SIGGRAPH 2025
Vancouver+ 10-14 August

# MegaLights: Stochastic Direct Lighting in Unreal Engine 5

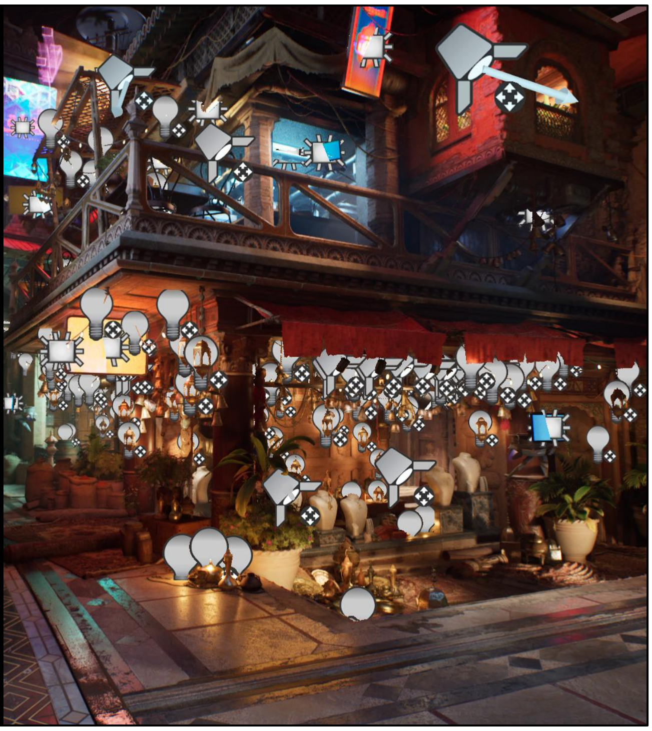
Krzysztof Narkowicz, Tiago Costa

SIGGRAPH 2025 Advances in Real-Time Rendering in Games *course*

Hi, my name is Krzysztof and together with Tiago Costa we'd like to present our new Stochastic Direct Lighting technique in Unreal Engine 5.

## Motivation

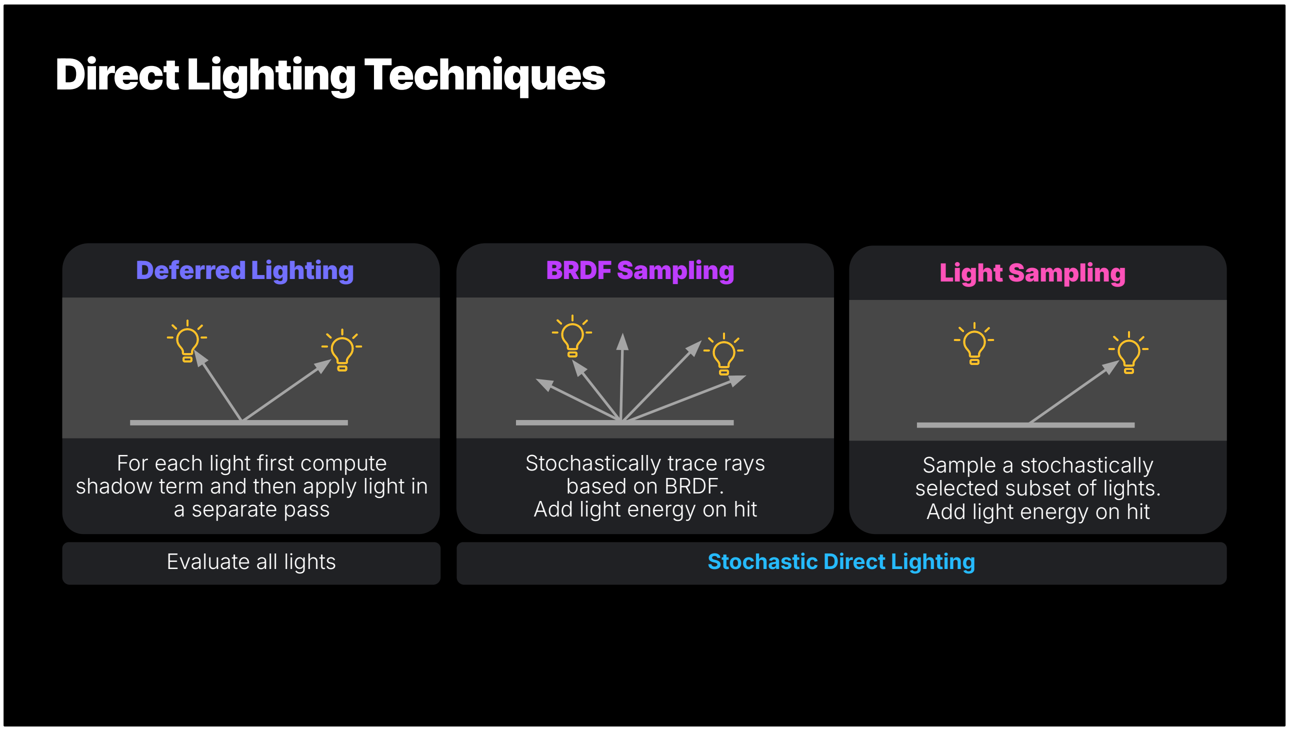
- 100+ shadowed lights on screen
  - Fully dynamic
  - Area lights
  - Complex BRDFs and light types
- **Workflows** are key to pretty pixels
  - No baking
- **Baseline** lighting method
  - Consoles and mid-range PC



Games constantly push the quality bar, want to use more lights, they want each light to be dynamic and to cast shadows and want to have area lights with realistic soft shadows. They also want to use complex BRDF, light sources or layered materials requiring multiple BRDF evaluations to simulate light passing through the material layers. Often to the point where just unshadowed light evaluation is becoming too expensive.

At the same time games getting more complex and I think we are now at the point where workflows are actually the main limiting factor of game graphics quality. Ideally artists should be able to work playfully without many limitations just like on this screenshot here. Which is maybe a bit excessive, but, well, this is what happened when artists figured out how to procedurally place lights.

Another key point that this has to be a baseline lighting technique. Something what artists can use as their main visual target during production. It can’t be an extra technique, which requires additional re-lighting pass by artists and would complicate workflows. It has to work at least on consoles and corresponding PC GPUs.



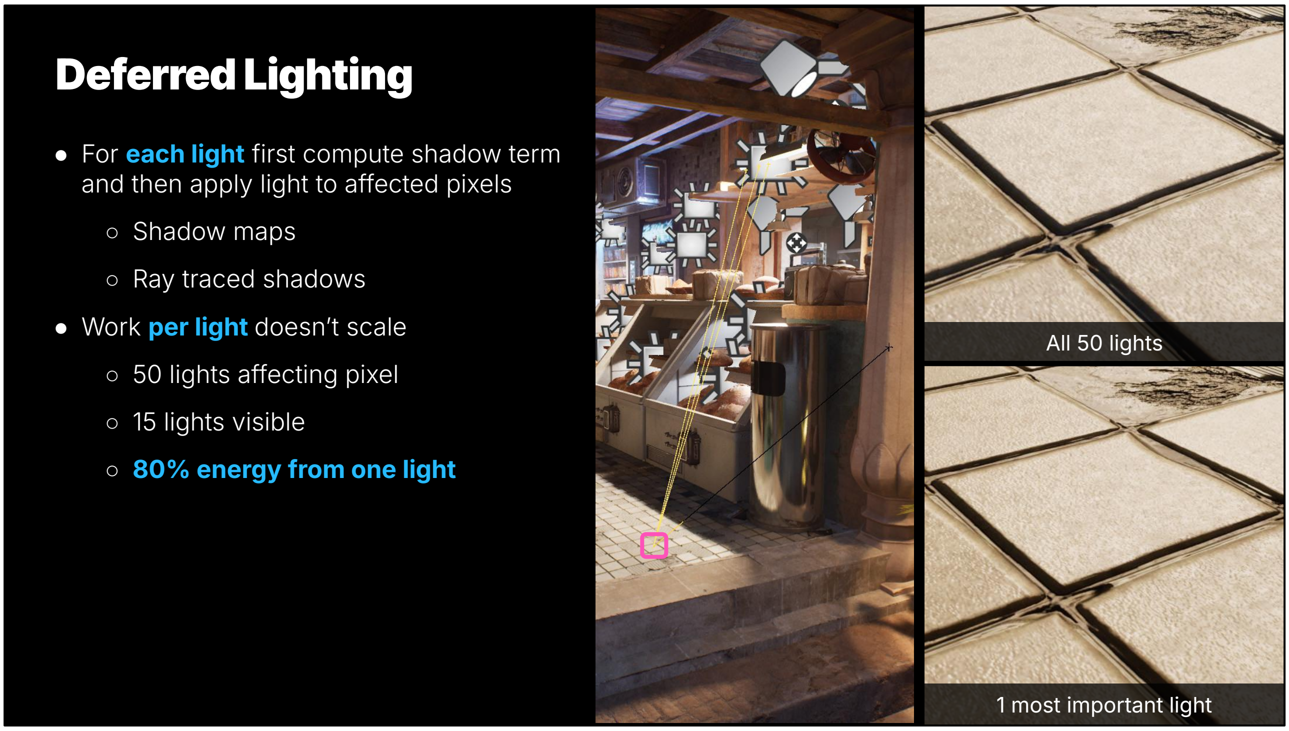
## Direct Lighting Techniques

Deferred Lighting
For each light first compute shadow term and then apply light in a separate pass
BRDF Sampling
Stochastically trace rays based on BRDF.
Add light energy on hit
Light Sampling
Sample a stochastically selected subset of lights.
Add light energy on hit
Evaluate all lights
Stochastic Direct Lighting

There are many ways how we could approach direct lighting.

The usual approach in games is that we do forward or deferred lighting, where for each light visible on screen we first precompute the shadow term using shadow maps or maybe ray trace shadows, denoise them and precompute a shadow mask. And then in a separate pass we apply lights one by one on screen combining their pre-integrated irradiance with a computed shadow term. This works great for a small number of lights, but it as the number of lights grows doing work per light can get prohibitively expensive.

Another way to do direct lighting, which is more common in the offline rendering, is to trace a fixed number of rays based on the BRDF or towards a stochastic subset of lights, and if we hit a light then we accumulate its energy. The main benefit here is that now performance is independent of lighting complexity. We only trace a fixed number of rays per pixel instead of doing work for each light.



## Deferred Lighting

- For **each light** first compute shadow term and then apply light to affected pixels
  - Shadow maps
  - Ray traced shadows
- Work **per light** doesn't scale
  - 50 lights affecting pixel
  - 15 lights visible
  - **80% energy from one light**

Given how popular deferred lighting is in games this was the first place where we started our investigation. We looked into ways how to speedup ray traced shadows, light evaluation and so on, but things just didn't really scale to our desired lighting complexity.

When we carefully looked at the content it became obvious why we aren't able to hit our goals. For example, this pixel is in attenuation range of 50 lights and for each light we need either to compute shadow maps or maybe ray trace and denoise shadows, but then only 15 lights affect that pixel at all, so we just did a lot of wasted work here.

Some of this cost can be hidden by caching shadow maps, but it doesn't really change scaling that much. There's only so much what we can cache before shadow cache invalidation caused by light, camera or object movement breaks performance. For example, in MegaLights demo we have bots flying around the scene creating 100s of moving lights, which completely break any caching schemes. Furthermore even if everything would be static, caching shadow map pages would require unreasonable amounts of memory. In our MegaLights demo enabling VSM (Virtual Shadow Maps) fills the the max allocation size of 4gb for the shadow map virtual page cache and still isn't able to fit all the required lights causing various artifacts.

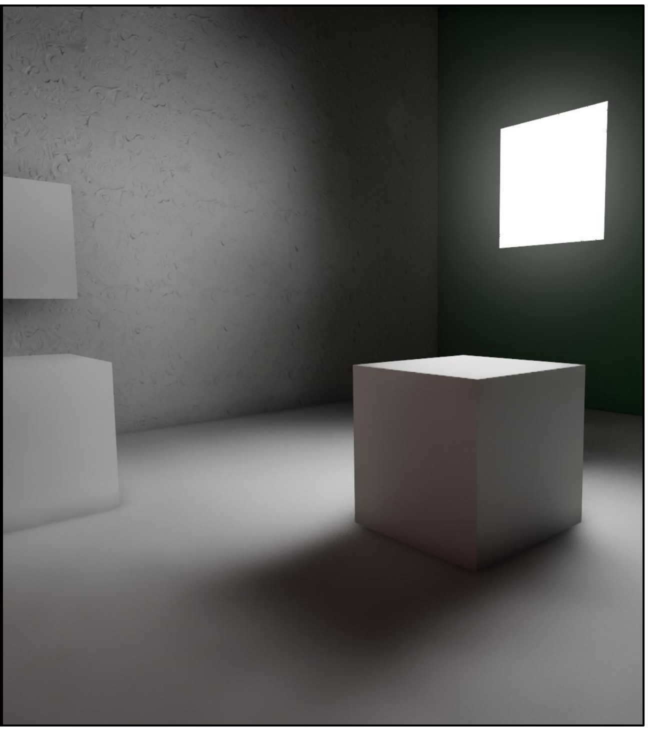
Ray traced shadows can hide some of the cost using adaptive tracing (adjusting number of traces per pixel or per tile) and using sparse shadow masks to save memory, but again this doesn't change overall scaling that much. Tracing rays and denoising is pretty expensive and doing it per light places a hard and relatively small limit on the number of lights.

Even if we would would somehow handle shadowing for free, there’s still an issue of evaluating lots of lights per pixel. In theory it’s only 15 lights, but those are complex light types and complex materials, which makes them too expensive on consoles even if they would be unshadowed.

If we look at those 15 visible lights, 80% of the energy is coming from a single light, so why are we trying to bruteforce those 50 lights? Why not just compute that single light at a high quality and approximate the rest?

## BRDF Sampling

- Just make it a **GI** problem?
- Hidden emissive surfaces are a common workaround in UE5
- Free, but quality is limited
  - BRDF sampling struggles to find small lights
- GI is already hard, don't have to make it even harder by trying to replace direct lighting

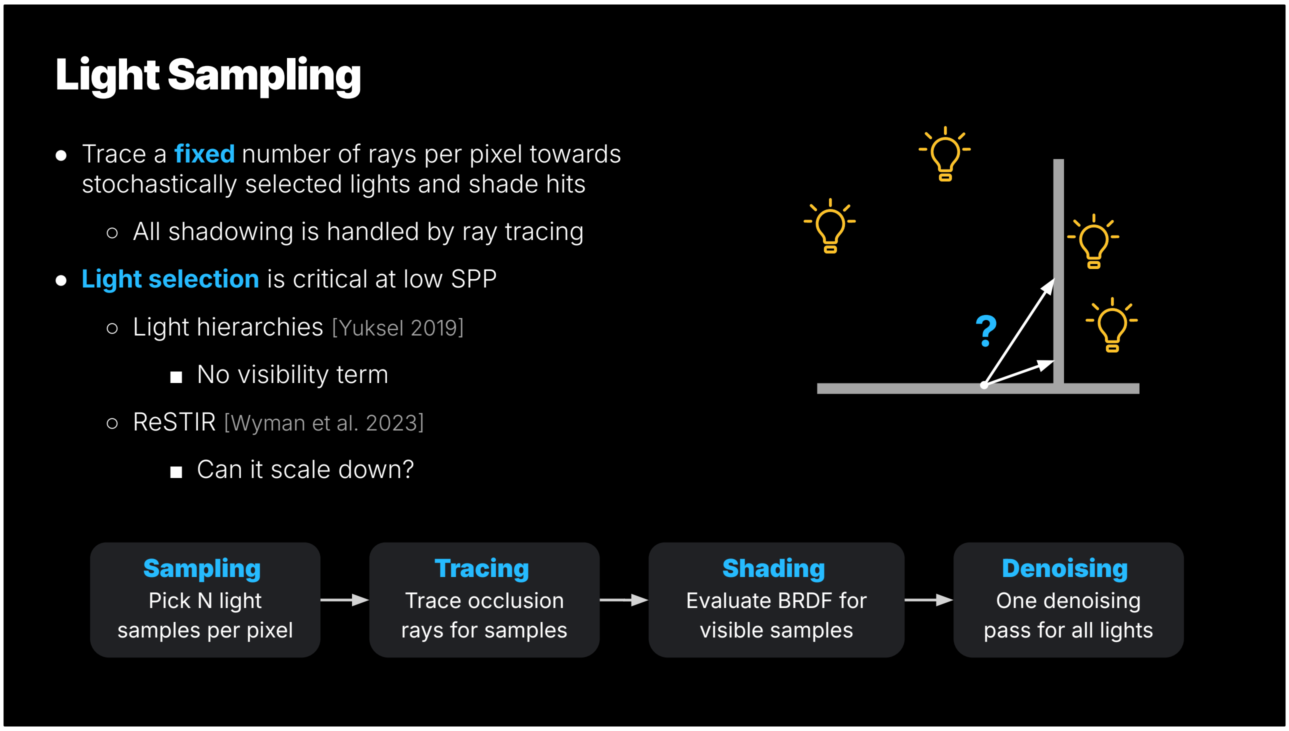


The other approach is to use BRDF sampling or basically making this a GI problem.

It's a common workaround in UE5. Artists place some kind of emissive meshes in the scene, hide them from the main view and use them as soft area light sources instead of analytical lights.

This is essentially free as we anyway have to compute GI, but comes with a bunch of quality issues. Small emissive surfaces or emissive surfaces which are far away, are really hard to find by just sampling the BRDF, even when augmented by Lumen (our GI/reflection system) ray guiding techniques.

The other problem is that real-time GI often needs help from the direct lighting. Even in offline rendering artists often use tricks like placing area lights in windows in order to speedup GI convergence. Here we are doing real-time GI, 60hz on consoles, and instead of helping GI with analytical lights we're instead asking it to also compute direct lighting.



## Light Sampling

- Trace a **fixed** number of rays per pixel towards stochastically selected lights and shade hits
  - All shadowing is handled by ray tracing
- **Light selection** is critical at low SPP
  - Light hierarchies [Yuksel 2019]
    - No visibility term
  - ReSTIR [Wyman et al. 2023]
    - Can it scale down?

**Sampling** Pick N light samples per pixel → **Tracing** Trace occlusion rays for samples → **Shading** Evaluate BRDF for visible samples → **Denoising** One denoising pass for all lights

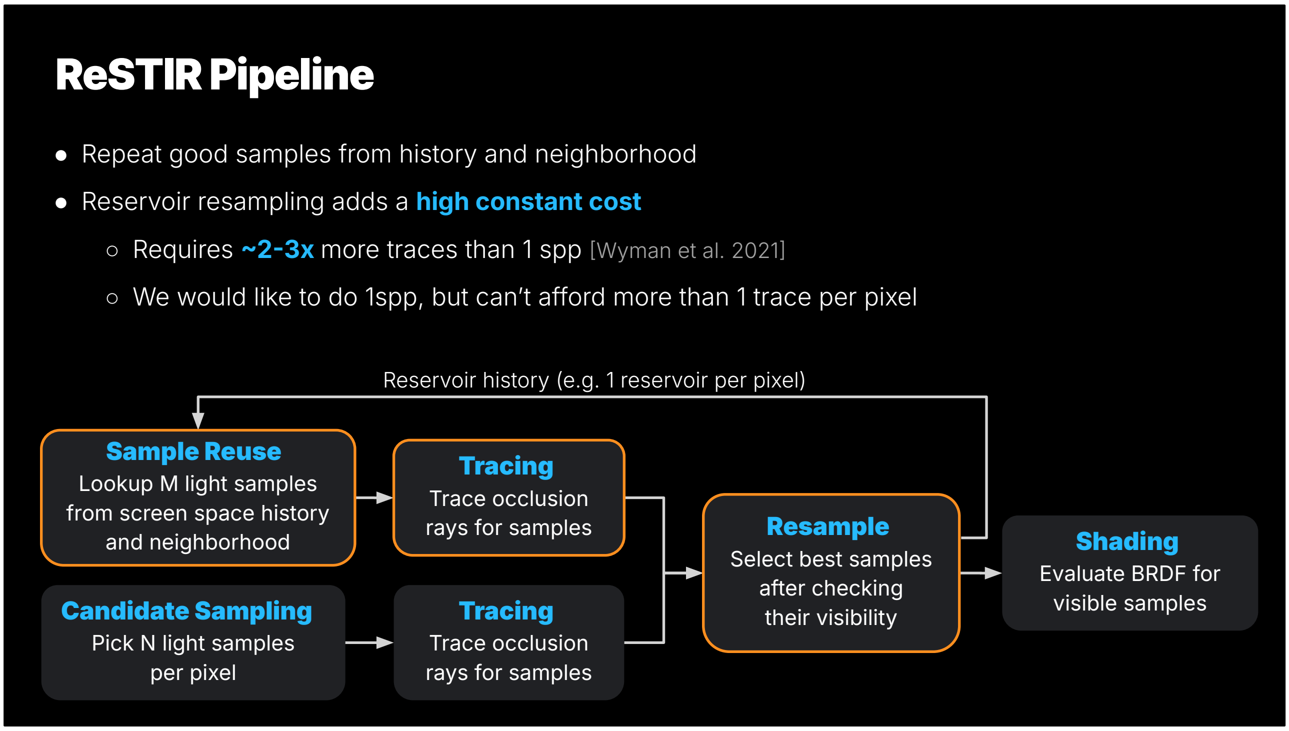
The third approach is build a new system, where we stochastically select a fixed subset of lights per pixel and trace rays towards them. If a ray hits a light then we shade that hit and accumulate light energy into a single render target. Finally we are going to denoise that render target after all tracing is done. This scales much better, as no matter how many light sources are there in the scene, we are always doing fixed amount of work per pixel.

Of course in practice it’s not that simple. On a console we can do maybe 1 ray per pixel and in that case light selection is is becoming extremely important as we want to maximize usage of each ray. For example, here on this diagram on the right we traced two rays towards 2 lights, which were really close to our shading point and seemed to be really important, but both of them are occluded, so we just wasted our entire budget without computing anything.

There are lots of approaches to light selection.

One of the most popular ones are different kinds of light hierarchies, where we first build some of hierarchy around lights and then try to pick the most important clusters. Pre-built hierarchy can speedup light sampling, but it doesn’t account for visibility and as we can see on this diagram visibility is a critical part of light selection. Light hierarchies can also enable some kind of light merging (hierarchical LOD), but in practice we run into too many issues with that (incorrect lighting, leaking etc.). Finally building this light hierarchy each frame can be pretty expensive, as this operation doesn’t translate well to GPUs.

The other approach which is really popular is ReSTIR. It can ship some amazing results on the high-end PC, but could we scale it down to console and so that it becomes our baseline lighting method?



## ReSTIR Pipeline

- Repeat good samples from history and neighborhood
- Reservoir resampling adds a **high constant cost**
  - Requires **~2-3x** more traces than 1 spp [Wyman et al. 2021]
  - We would like to do 1spp, but can't afford more than 1 trace per pixel

Reservoir history (e.g. 1 reservoir per pixel)
Sample Reuse
Lookup M light samples from screen space history and neighborhood
Tracing
Trace occlusion rays for samples
Candidate Sampling
Pick N light samples per pixel
Tracing
Trace occlusion rays for samples
Resample
Select best samples after checking their visibility
Shading
Evaluate BRDF for visible samples

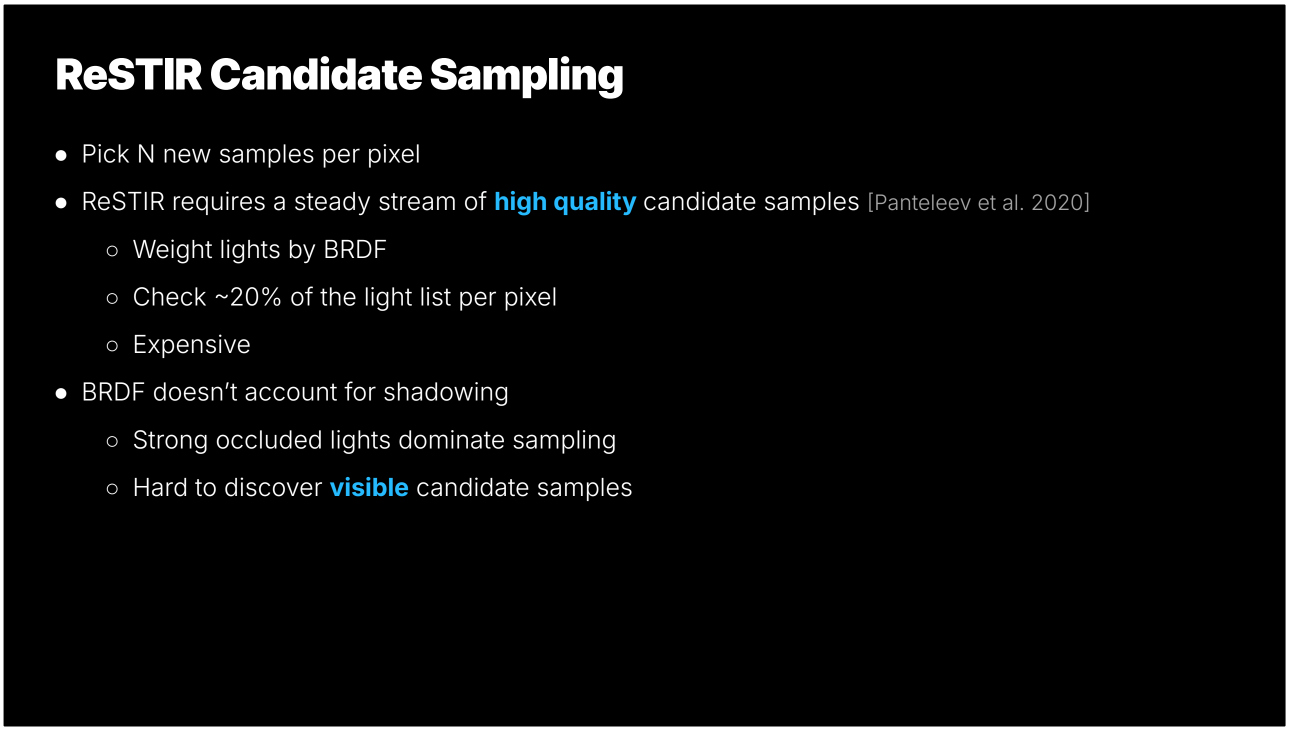
In order to answer that question we need first to take a look what ReSTIR actually does.

ReSTIR adds those few orange blocks into stochastic direct lighting, which will lookup a few samples from the history and a few neighbors, check their visibility by tracing rays and finally combine those reused samples with new candidate samples stochastically selected this frame.

The main idea here is that if we can't stochastically pick a good candidate sample, as for example all our selected samples turned out to be occluded, likely we have something better in the history or maybe our neighboring pixel was more lucky with its light selection.

ReSTIR greatly improves quality, but the downside is that this reuse adds a high constant cost. Not only the reuse operation itself is expensive, but most importantly we have to check visibility of each reused sample by tracing rays.

In order to achieve reasonable quality we would like to have at least 1 SPP, which would require 2-3 traces per pixel. It's maybe not a big deal on a high-end PC, but on a console we can't really trace more than 1 ray per pixel.



## ReSTIR Candidate Sampling

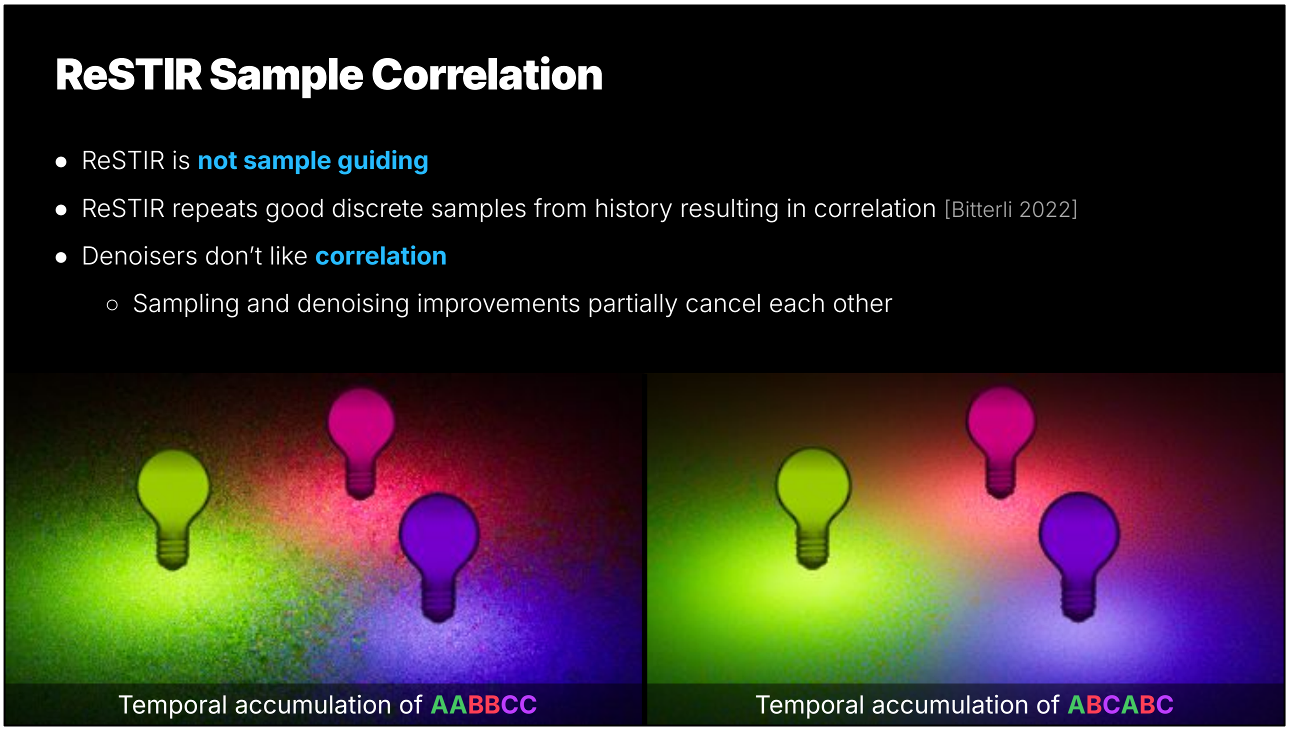
- Pick N new samples per pixel
- ReSTIR requires a steady stream of **high quality** candidate samples [Panteleev et al. 2020]
  - Weight lights by BRDF
  - Check ~20% of the light list per pixel
  - Expensive
- BRDF doesn't account for shadowing
  - Strong occluded lights dominate sampling
  - Hard to discover **visible** candidate samples

The other problem is candidate sampling.

In theory we could just pick some completely random lights per pixel, but in practice in order to achieve shippable quality we need to be selecting those lights really carefully. We need to weight each light by the BRDF and we need to be able to evaluate at least 20% of the light list per pixel. Which again maybe not a big deal on high-end PCs, but on consoles those costs accumulate really quickly, as we are basically doing unshadowed light evaluation for 20% of lights in our scene.

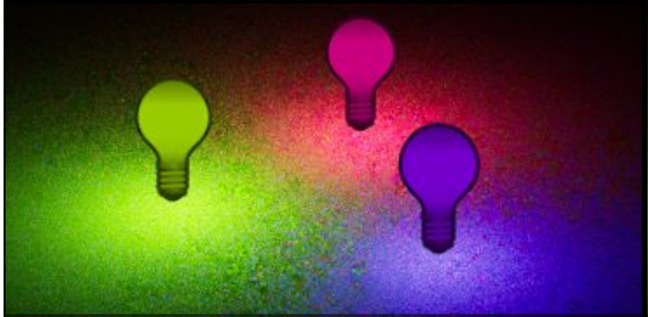
On top of that BRDF doesn't account for shadowing and strong occluded lights will receive a lot of samples making it even harder to discover good visible lights.

To sum up, we have this high constant cost, which we already can't afford, and we still need to solve a similar light sampling problem as before, just with a different constant.

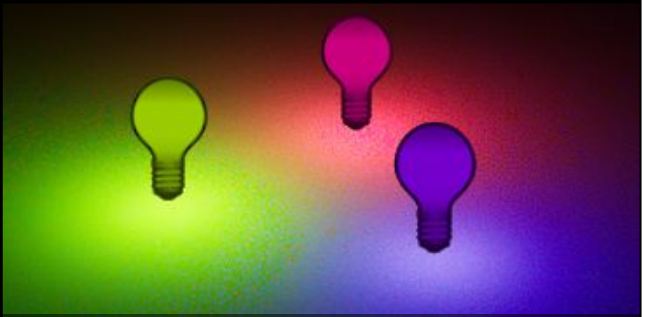


## ReSTIR Sample Correlation

- ReSTIR is **not sample guiding**
- ReSTIR repeats good discrete samples from history resulting in correlation [Bitterli 2022]
- Denoisers don't like **correlation**
  - Sampling and denoising improvements partially cancel each other



Temporal accumulation of AABBCC



Temporal accumulation of ABCABC

So a good question - is there something we can do better than ReSTIR?

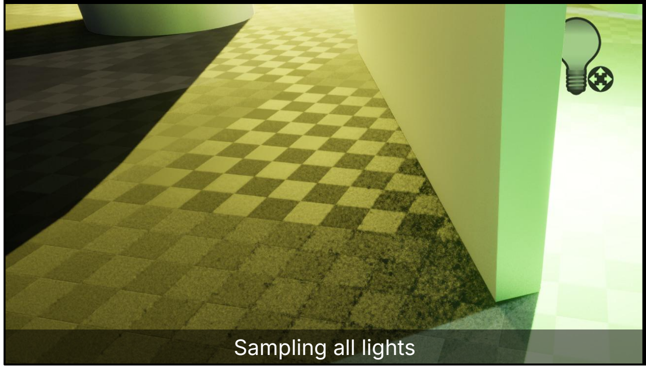
When playing around with stochastic light sampling we noticed that sometimes just simple sampling works better than ReSTIR. And explanation for that is pretty simple. Initial samples have a much higher chance to be repeated, as each frame there are exponentially more paths connecting those initial samples to current frame samples. This is the reason why in practice ReSTIR implementations clamp M (number of frames to keep sample in history), but even with clamp this clumping effect is still prominent. In order to fix it we would need to clamp M to 1, which would effectively disable any reuse and we would fallback to simple stochastic sampling. Those clumps of repeated samples make ReSTIR output less noisy, but it also greatly reduces effectiveness of denoiser. For the denoiser to work optimally we want nice non-correlated and alternating patterns of lights.

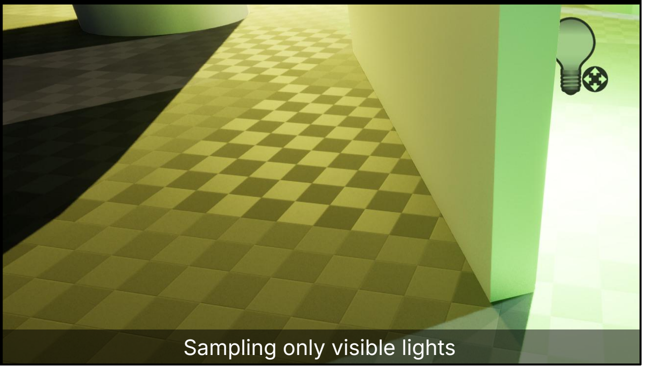
The other way to look at it is that both ReSTIR and denoiser do very similar screen space spatial and temporal filtering. With the difference being that ReSTIR reshades reused samples instead of reusing already shaded samples, has a longer temporal history (usually 2x longer) and that those samples don't have to go through history rectification. This in general allows ReSTIR to accumulate more temporal frames and more aggressively reuse neighbors, but it also scrambles sampling patterns which in some cases may actually lead to overall quality reduction.

So why not skip that spatio-temporal reuse and instead design our sampling for the denoiser, so that we can achieve similar quality much cheaper and be able to fit into our constrained target budget?

## Back To Square One

- Ideally we would sample a **visible light list** per pixel
    - List enables sample correlation reduction
    - Sampling hidden lights wastes samples and causes noise

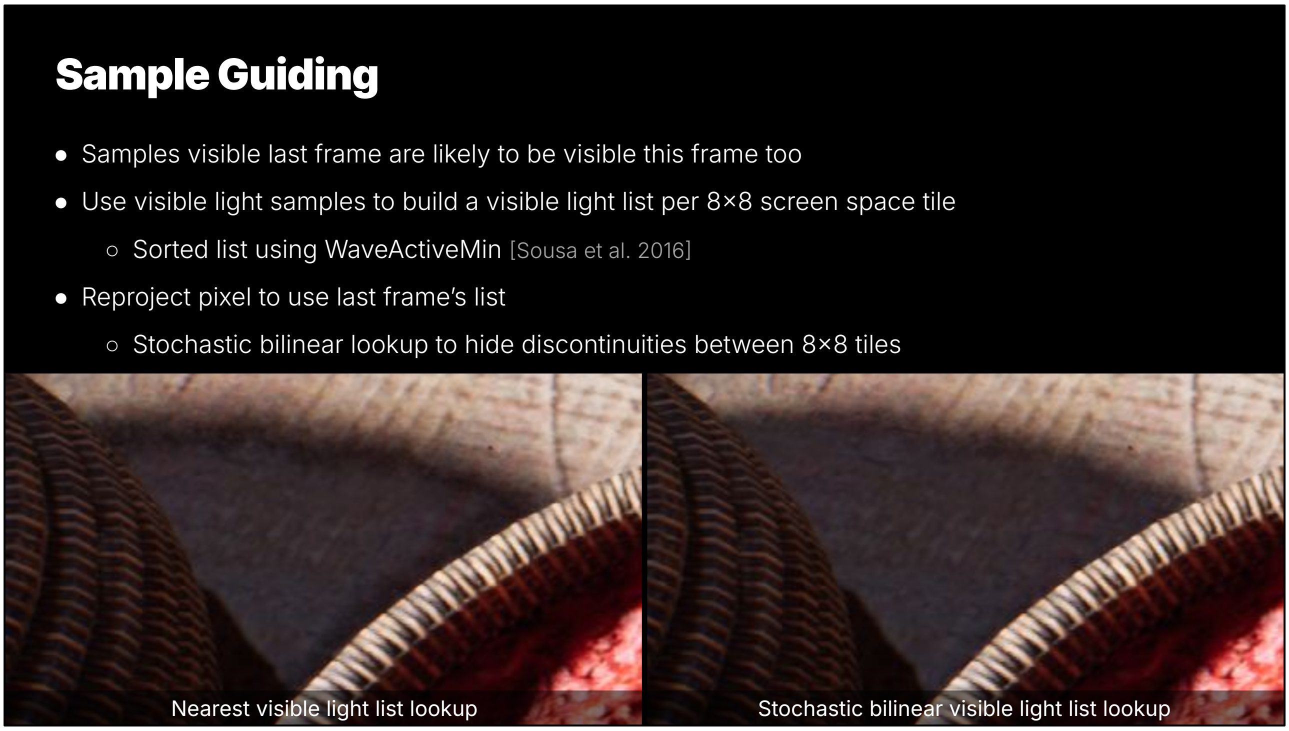

Sampling all lights


Sampling only visible lights

Ideally we would just sample a visible light list per pixel.

Having such a list would allow us to stochastically select a different light each frame generating nice patterns for the denoiser.

Sampling only visible lights would prevent us from wasting work on tracing towards hidden lights, which don’t contribute to the final pixel and would only waste our sample budget and increase noise.



## Sample Guiding

- Samples visible last frame are likely to be visible this frame too
- Use visible light samples to build a visible light list per 8×8 screen space tile
  - Sorted list using WaveActiveMin [Sousa et al. 2016]
- Reproject pixel to use last frame's list
  - Stochastic bilinear lookup to hide discontinuities between 8×8 tiles

Nearest visible light list lookup

Stochastic bilinear visible light list lookup

History could be a great source for such a visible light list, as samples which were visible last frame likely will be also visible next frame.

We can't really have a light list per pixel due to the memory overhead, but we could have it for a 8x8 screen space tile. Neighboring pixels have a high probability to share light visibility. In order to create such list, after the tracing pass, we gather all visible light samples and use WaveActiveMin output a sorted light list per 8x8 tile. In the next frame we can reproject our pixel in the sampling pass, lookup appropriate tile and select a light from that visible light list.

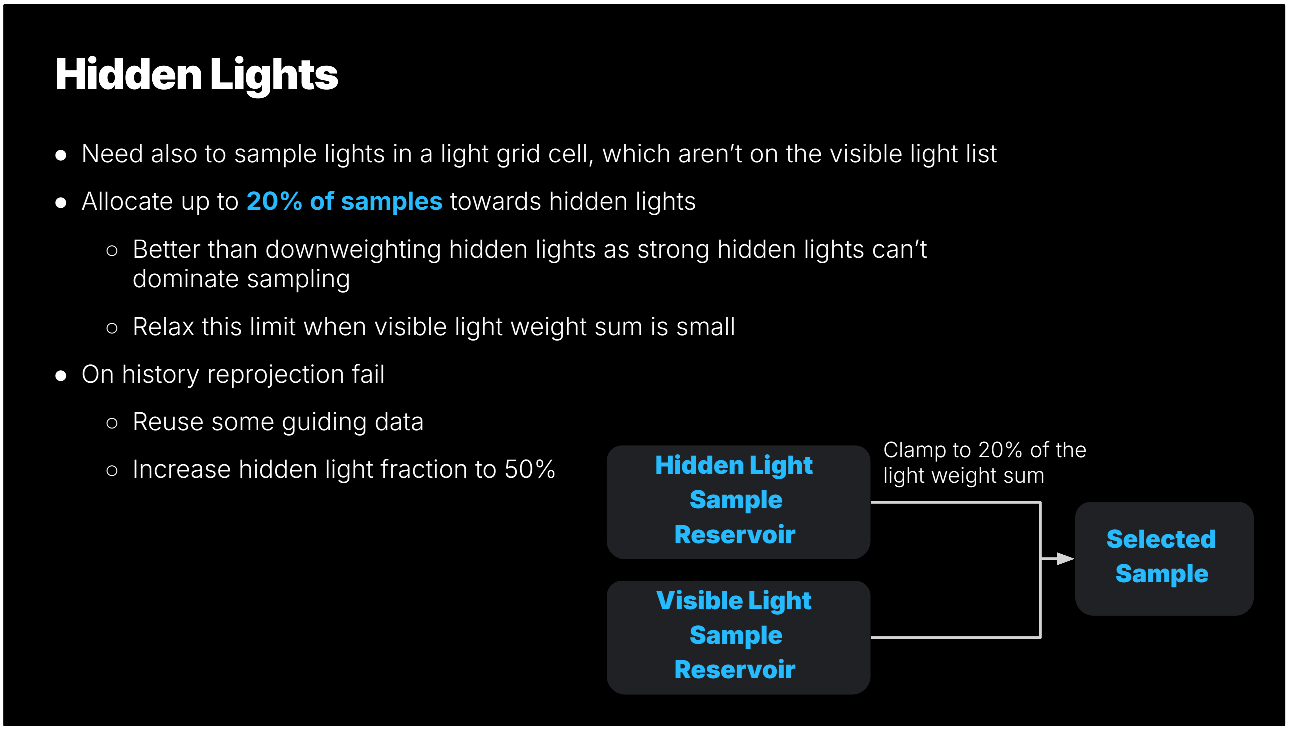
We tried also using various kinds of downweighting schemes instead of binary visibility, where each light would be weighted by the adjusted ratio of ray hits/misses, but it turned out to be less effective in practice as such schemes trace less rays for samples in penumbras. Usually we actually want the opposite, as penumbras can have a lot of noise and we don't want to undersample those.

8x8 tiles are pretty large and sometimes create some visible discontinuities. Essentially we are switching here from one light list to another on the tile border. In order to fix that we use a stochastic bilinear lookup, which will smoothly interpolate between the 4 closest list. Not only hiding those transitions, but also speeding up new visible light propagation on screen.

8x8 screen space tiles do reduce effectiveness of sample guiding on depth discontinuities, but in practice it's not that bad, as we accumulate light visibility per pixel.

Initially we used a bloom filter, which we find interesting, as bloom filters are both ingenious data structures and also quite uncommon in real-time rendering. Later we switched to explicit lists, as those allow to have extra payload per light and allow to sample such a list without having to iterate through all the lights inside a light grid cell.

Those lists can have also a small hardcoded size, as fundamentally their length is limited by the number of visible samples per 8x8 tile. The other way to look at it is that we if we shading 1SPP then even with temporal accumulation in denoiser we can’t really accumulate too many lights (no matter how we pick lights), which means that the max number of visible lights per tile is limited by the number of shaded samples per pixel.



## Hidden Lights

- Need also to sample lights in a light grid cell, which aren't on the visible light list
- Allocate up to **20% of samples** towards hidden lights
  - Better than downweighting hidden lights as strong hidden lights can't dominate sampling
  - Relax this limit when visible light weight sum is small
- On history reprojection fail
  - Reuse some guiding data
  - Increase hidden light fraction to 50%

Hidden Light Sample Reservoir
Clamp to 20% of the light weight sum
Visible Light Sample Reservoir
Selected Sample

We can't only sample the visible light list. Scenes are dynamic and hidden lights may become visible. We need to spend a certain part of our sample budget on discovering new visible lights.

What we do here is that we sample from both the visible and hidden light list and we clamp the hidden light list weight sum to a 20% of the total weight sum. This way we are able to allocate a fixed number of samples towards hidden lights.

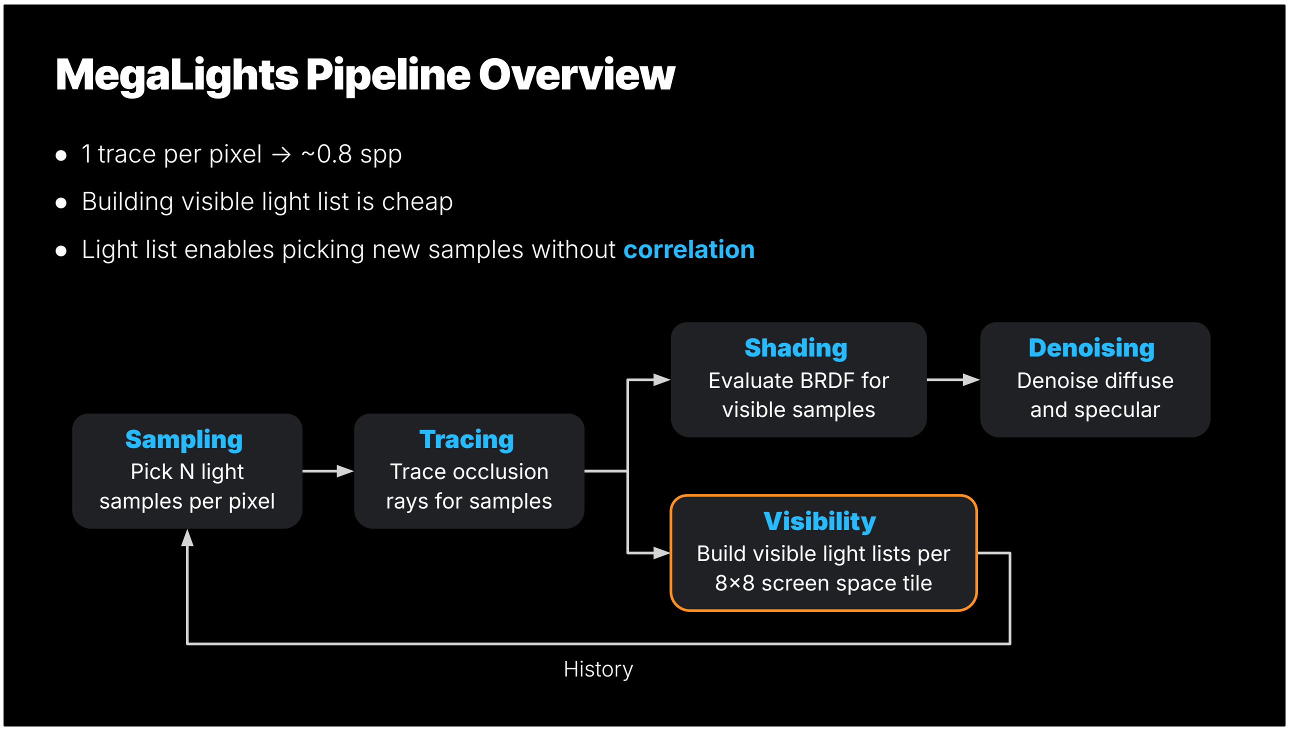
This can be done pretty easily by sampling visible lights into one reservoir, hidden lights into a second one and clamping hidden light reservoir weight just before merging them together. Both reservoirs use a separate random variable, so that they are uncorrelated. We also add hidden light reservoir based on the visible light list random variable, which allows us to maintain visible light list random variable noise properties.

20% is of course just an arbitrary tweakable number, which controls tradeoff between quality after convergence and quality during disocclusion / how fast technique reacts to scene changes.

Sometimes visible lights may be quite dim or maybe we don't have any local lights at all. In that case we relax that 20% clamp so that more rays can be traced towards hidden lights speeding up convergence on scene changes or disocclusion.

Hidden light budget works better than downweighting hidden lights, as strong hidden lights can’t ever dominate sampling (exceed 20% sample budget). Our method is pretty good at handling hard cases like strong completely occluded lights, which normally would dominate sampling and cause lots of noise.

Sometimes history reprojection fails due to object movement or due to out of screen sampling. In that case we still reuse closest tile, as likely it has some useful lights, but we also increase the ratio of hidden light samples in order to speedup new visible light discovery.



## MegaLights Pipeline Overview

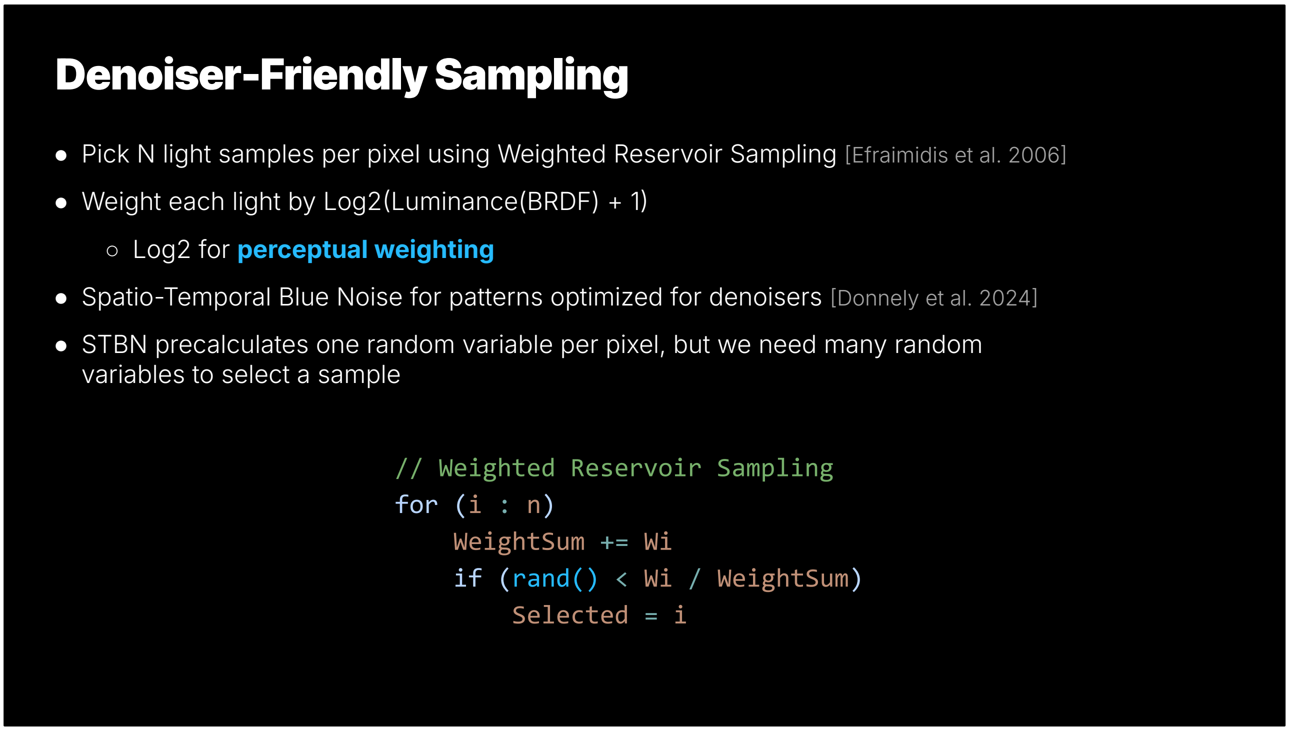
- 1 trace per pixel → ~0.8 spp
- Building visible light list is cheap
- Light list enables picking new samples without **correlation**

Sampling
Pick N light samples per pixel
Tracing
Trace occlusion rays for samples
Shading
Evaluate BRDF for visible samples
Denoising
Denoise diffuse and specular
Visibility
Build visible light lists per 8×8 screen space tile
History

Now when we have this visible light list we can look at our new pipeline. It's pretty similar to what we saw before, but we add this one orange block which is responsible for building a list of visible lights. Next frame this list is used for sampling lights allowing us to guide samples at a pretty small cost of building this list.

We do allocated some traces towards checking hidden lights, but it's a relatively small overhead. Most of the rays are being sent towards visible lights and contribute towards final pixels.

Finally we have a light list, which will allow us to generate nice patterns for the denoiser.



## Denoiser-Friendly Sampling

- Pick N light samples per pixel using Weighted Reservoir Sampling [Efraimidis et al. 2006]
- Weight each light by Log2(Luminance(BRDF) + 1)
    - Log2 for **perceptual weighting**
- Spatio-Temporal Blue Noise for patterns optimized for denoisers [Donnely et al. 2024]
- STBN precalculates one random variable per pixel, but we need many random variables to select a sample

```
// Weighted Reservoir Sampling
for (i : n)
    WeightSum += Wi
    if (rand() < Wi / WeightSum)
        Selected = i
```

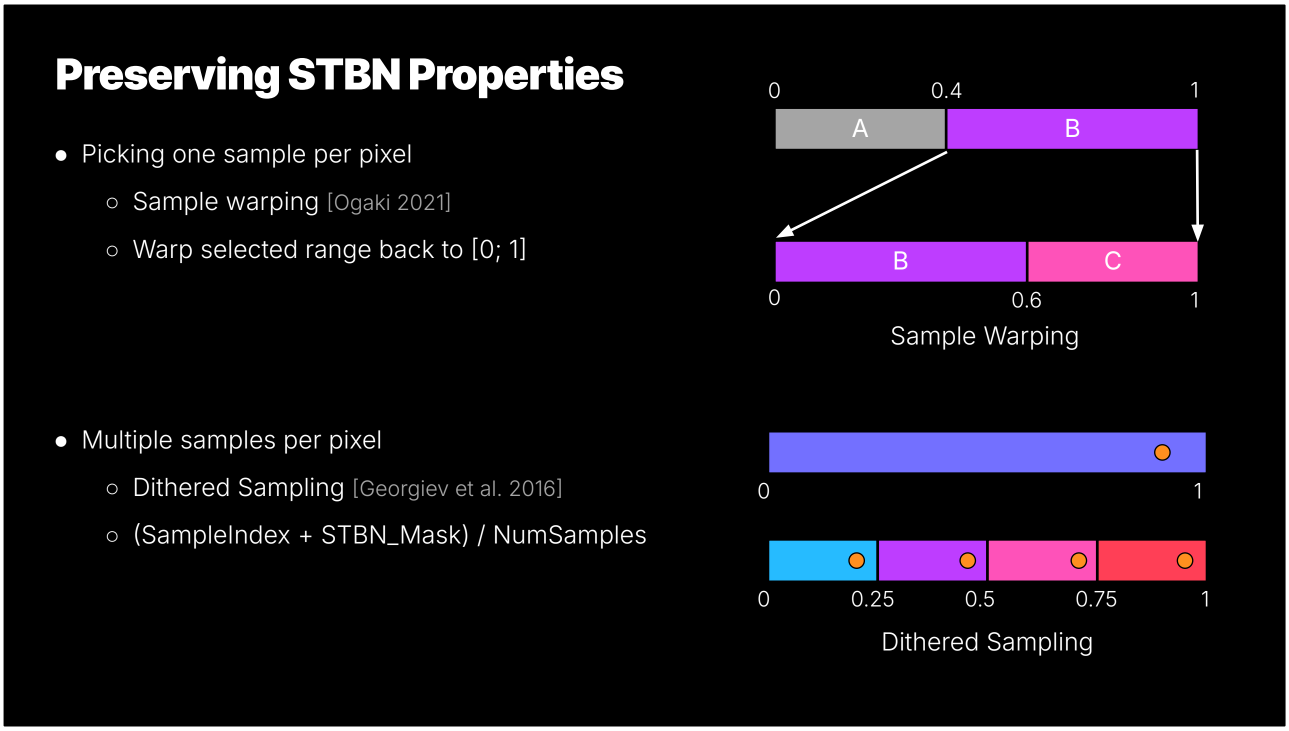
Now let’s make a good use of our light list.

In order to select our samples we use weighted reservoir sampling, which as you can see here on the bottom, is a very simple method to loop over all lights once and stochastically pick one based on their weights.

We weight each light by the luminance of BRDF using log for perceptual weighting. Perceptual weighting is a general quality improvement, but most importantly it allows us to downweight very strong lights which after the tonemapping won’t have as large impact on the final pixels, helping with the strong occluded light issue.

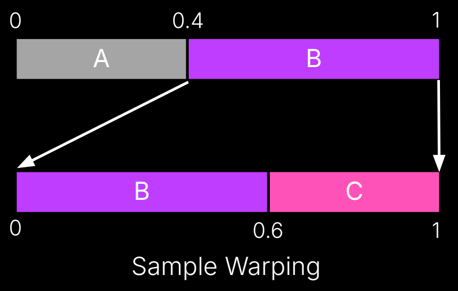
Spatio-Temporal Blue Noise provides lookup textures with noise patterns which are optimized for denoising. Those patterns are really good (and cheap) and we use them all over the engine for different kinds of stochastic operations in screen space.

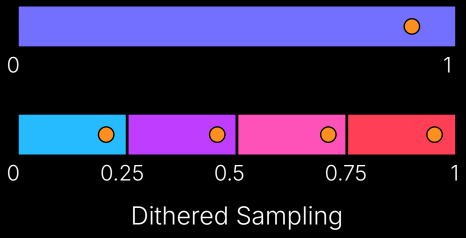
There’s only one small issue here. This lookup texture gives us a single random variable for each pixel, but as we can see here we need one random variable for each iteration of our sample selection loop. We can’t just sample STBN texture multiple times, as it was specifically precalculated to be sampled once for each pixel in order to maintain those nice STBN pattern properties.



## Preserving STBN Properties

- Picking one sample per pixel
  - Sample warping [Ogaki 2021]
  - Warp selected range back to [0; 1]
- Multiple samples per pixel
  - Dithered Sampling [Georgiev et al. 2016]
  - (SampleIndex + STBN_Mask) / NumSamples



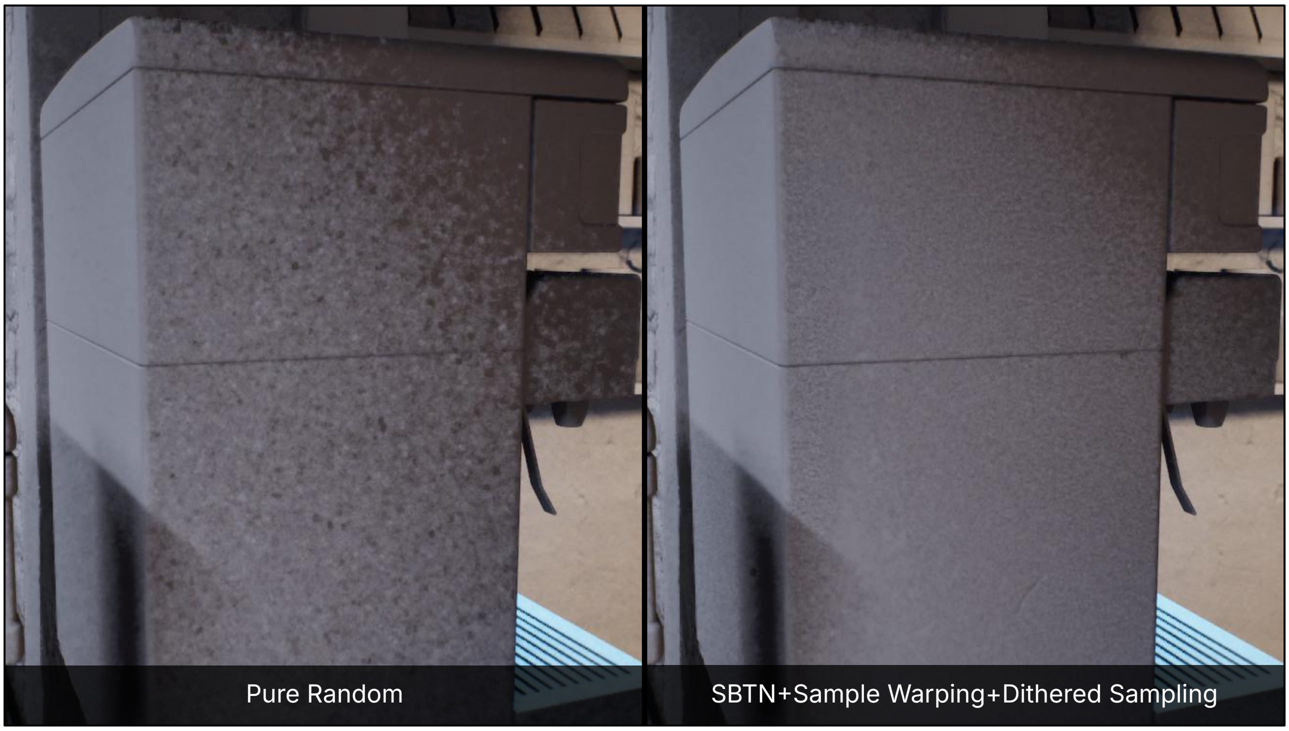


Thankfully there are a few tricks which allow us to reuse our random variable multiple times while still preserving original noise properties.

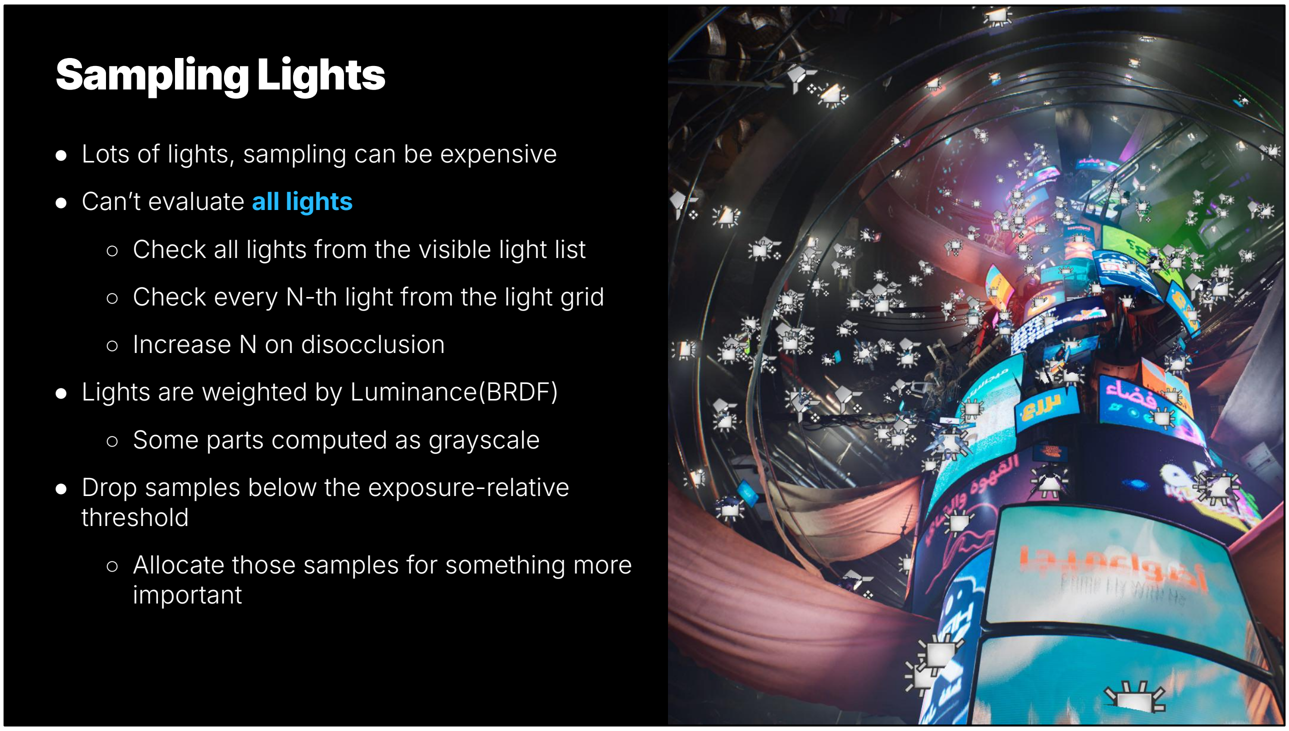
For the single sample selection we can remap selected range back to 0-1 after each light selection step.

If we want to trace multiple rays per pixel then we can use dithered sampling, which will remap our random variable into multiple segments.

Any kind of remap will use one bit of precision of our random variable, but in practice it's not an issue. We sample using two random variables, one for the visible light list and one for the hidden light list, which allows to redistribute this bit loss over two random variables. Additionally anyway we can't loop over too many lights per list due to the performance constraints.



All of this allows us not only to sample faster using just a single texture lookup, but most importantly it preserves those nice STBN properties. Which is a considerable quality improvement and it's completely free - it doesn't require any extra computations or traces.



## Sampling Lights

- Lots of lights, sampling can be expensive
- Can't evaluate **all lights**
  - Check all lights from the visible light list
  - Check every N-th light from the light grid
  - Increase N on disocclusion
- Lights are weighted by Luminance(BRDF)
  - Some parts computed as grayscale
- Drop samples below the exposure-relative threshold
  - Allocate those samples for something more important

Now we know how to select a light, but in large scenes looping over all lights in a light grid cell can get pretty expensive. Those light grid cells may contain 100s of lights.

What we do here is that we evaluate all the lights on the visible light list, and then we evaluate some lights from the light grid, skipping lights which were already evaluated from the visible light list. Each pixel evaluates a different stochastic subset, which makes it pretty incoherent, but also pretty effective at quickly picking up light changes. If we detect a history miss we increase number of evaluated lights from the light grid cell.

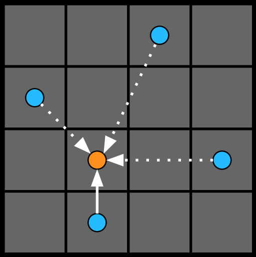
The entire algorithm just boils down to iterating over two sorted light lists with different strides and evaluating min light index of two lists.

One other simple optimization we can do here is that we need only the luminance of BRDF. We found that full grayscale BRDF computation (converting all inputs to grayscale) adds too much noise, but we were able to change some of our code like light component accumulator to grayscale reducing VGPR usage.

We also discard samples below a certain exposure-relative threshold. This threshold is set pretty low, so that it almost doesn't cause any bias (energy loss). This isn't a performance optimization, but it does help with quality. Basically if a light isn't strong enough to affect the final pixel then instead of wasting a ray to check its visibility, we can allocate that ray for something more important. This also removes some fireflies coming from very improbable and dim lights, which can be weighted pretty high on some frames.

## Downsampled Sampling

- Light weights are **similar between pixels**
- Temporally and spatially jittered checkerboard or 4-rooks pattern
- Stochastic bilinear upsampling using depth and normal weighting
- Reuse history from last frame when upsampling fails



1 sample every pixel

4 samples every 1/4th pixel

Light weights are usually pretty similar between pixels if their normals and depth are similar. We can use this to sample at a lower resolution. For example instead of picking 1 sample per pixel, we can pick 4 samples every 4 pixels reusing BRDF computations.

The important part here is that we need to use good sampling patterns like checkerboarding or 4-rooks. We also jitter those sampling patterns spatially and temporally. All of this greatly increases our chance to reconstruct the final pixel.

After sampling we will do a stochastic bilinear upsample based on depth and normal weights, which eventually should converge on the right result.

In some rare cases we may not have any valid pixels to upsample from - all weights may be zero due to large depth or normal discontinuity. For example, there may be a thin wire and we may not sampled any pixel belonging to it. In that case we reuse history without any kind of history rectification like neighborhood clamp.

Due to history rectification and upsampler this doesn't converge on the same result as full screen sampling and can make shadows somewhat softer (inbetween full res and half res). On the other hand it makes sampling almost four times faster, so it can be a pretty good tradeoff in practice.

Currently we're working on changing this to an adaptive sampling scheme in order to retain quality of full resolution sampling and better performance of lower resolution sampling.

## Directional Light

- Can dominate sampling
  - Major issue indoors, where directional light isn't visible at all
- Separate RT shadow pass? [Knapik el al. 2024]
  - Expensive
- Limit directional light to **50% of samples**
  - Relax this limit when local light's weight sum is small

Directional lights can cause a lot of issues for stochastic light sampling.

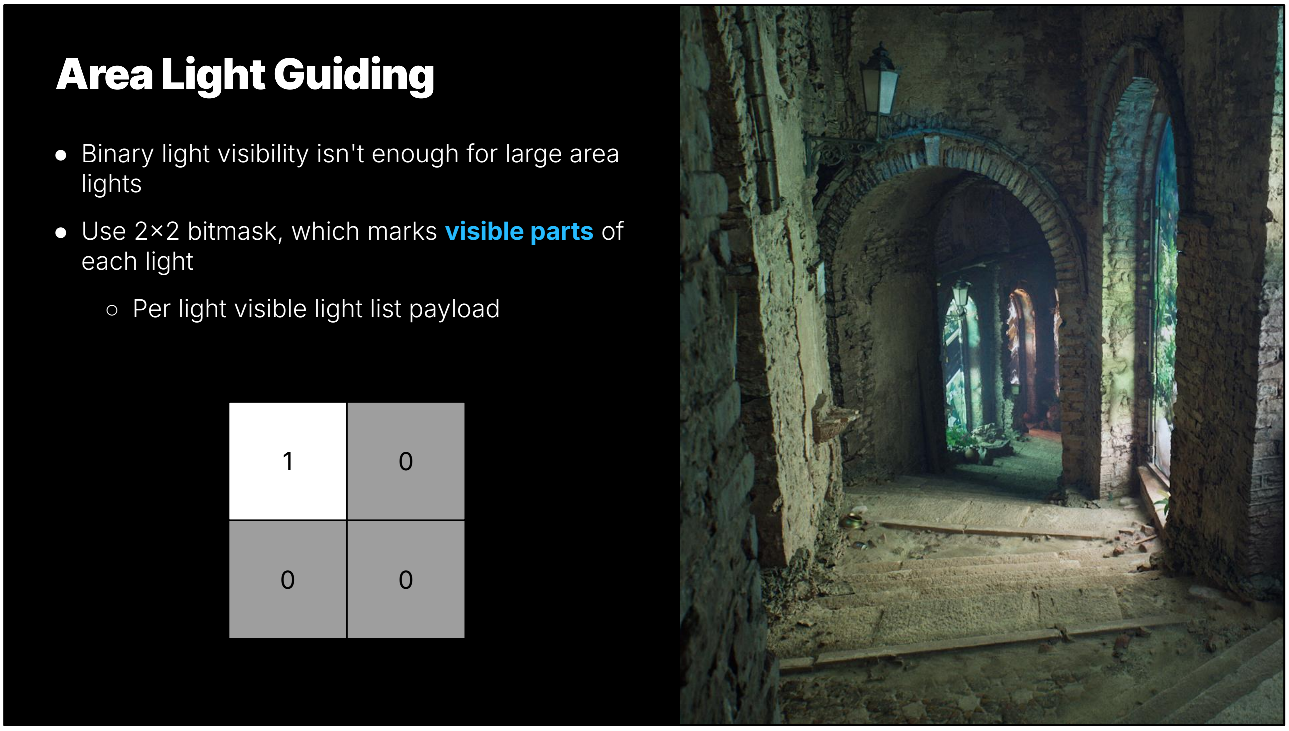
Directional light is usually a very strong light source, often thousands times stronger than anything else in the scene. When sampling based on the BRDF such lights will have a very high weights, which means that during light selection we're going to send almost all rays towards it and almost no rays are going to be used for local light sources, which can be an issue when trying to discover new visible lights in interiors.

A common solution would be to compute directional lights in a separate pass, for example by running a dedicated ray traced shadow pass for the directional light. This separate pass can also improve quality, as now we are able to run a separate shading and denoising pass only for the directional light. The downside is that this can be pretty expensive, as we have now to trace extra rays per pixel and run another shading and denoising pass, and may not always fit in a constrained game budget.

What we do instead is that we limit a fraction of the sample budget which can be used for the directional light. This is very similar to hidden lights, where we clamp directional light weight to a % of local light's weight sum. And we do this only when that local light's weight sum is above a certain threshold. This way if there are no local lights or local lights are very dim, we're able to allocate more samples towards the directional light than a fixed 50% budget.

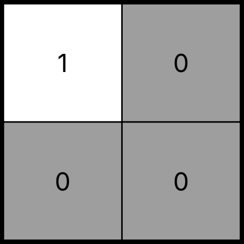
This scheme essentially works as a soft ray budget, where sometimes we trace more or less rays towards the directional light depending on the local lights affecting given pixel.

Other light types could also have the same issue as the directional light. For example, someone could use a very strong spotlight simulating sun. In practice we didn’t run into such issues. Less strong lights than sun are usually well handled by just perceptual weighting. 20% hidden light budget also improves handling of such lights, as we can guarantee that at most they will receive 20% of rays. Still in theory this scheme could be extended to select a list of such very strong lights and create a separate budget for them.



## Area Light Guiding

- Binary light visibility isn't enough for large area lights
- Use 2×2 bitmask, which marks **visible parts** of each light
  - Per light visible light list payload



Another thing we need to take a look are area lights.

As you can see here on this screenshot, those may be quite large and just the binary light visibility isn't really enough. A large part of such light may be occluded and we would be wasting samples tracing towards it.

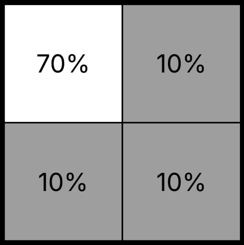
In order to improve this we keep an extra 2x2 bitmask, which marks visible parts of each light. This bitmask is just a payload in our visible light list and is built together with the visible light list.

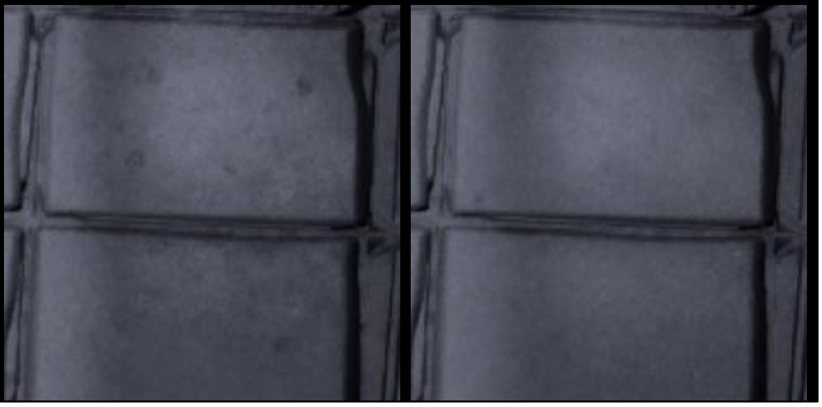
To be more specific 2x2 bitmask subdivides 2d random variable used for area light sampling. Area light sampling functions are designed to remap that 2d random variable from [0;1] square into light source area while preserving area so that they maximize quality of generated samples. This way it does work pretty well on any light source (capsules, spheres etc.).

Hardcoded 2x2 subdivision isn't ideal for all situations. Sometimes we would want to have more subdivisions, as some lights can be really large. In other cases like line lights we may want to have a different subdivisions ratio like 4x1. We're currently working on making this into an adaptive and hierarchical scheme.

## Area Light Sampling

- For each selected light
  - Pick a **point** on the light source
  - Downweight hidden parts
  - 2d STBN with 2d sample warping [Clarberg et al. 2016]

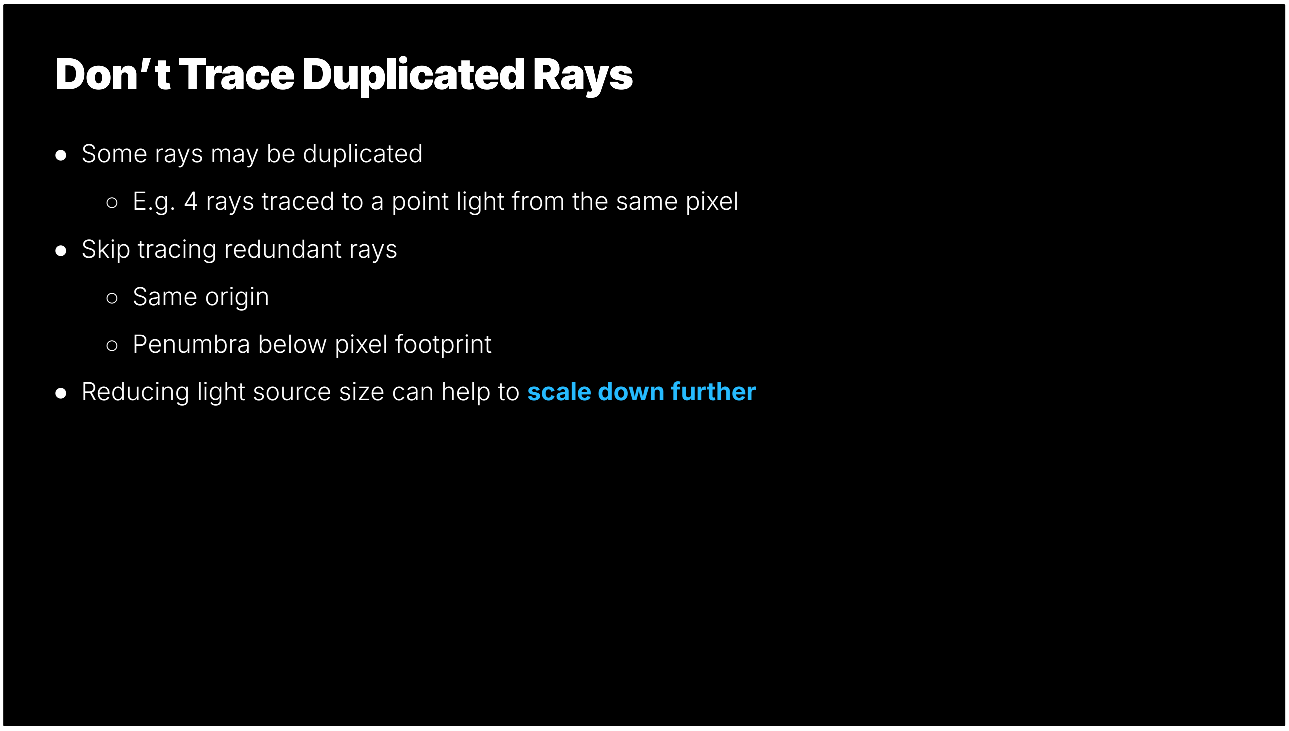




During sampling we first select N lights per pixel and then for each selected light we need to select a point on the light's surface.

This point selection will downweight hidden regions, which allows us to trace more rays towards the visible parts of the light reducing noise.

Each area light is sampled using 2d STBN noise with 2d sample warping in order to retain nice STBN properties after downweighting hidden regions. Those are a bit different flavors of STBN and warping than we saw before, as now we are working with 2d points.



## Don’t Trace Duplicated Rays

- Some rays may be duplicated
  - E.g. 4 rays traced to a point light from the same pixel
- Skip tracing redundant rays
  - Same origin
  - Penumbra below pixel footprint
- Reducing light source size can help to **scale down further**

Due to the stochastic nature of sampling some rays may be duplicated.

For example, we may have a strong point light near a surface and trace 4 identical rays towards that one point light from the same pixel. We don’t really need to trace all 4 identical rays. We can trace just 1 and reuse its visibility for the other 3.

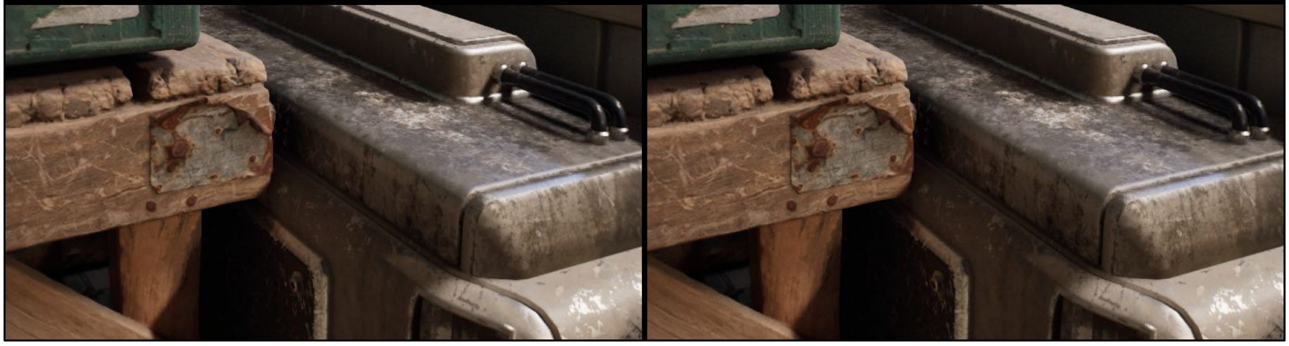
This may also happen with very small area lights which are far away, where penumbra is just too small.

It’s not a very common occurrence in content as artists love to use large area lights, but it does improve performance a bit without having any downside.

It also allows to tweak global penumbra size multiplier for scaling down further. This performance scalability multiplier will change the lighting look, as it's going to reduce area light source size, but it will also speedup traces as more rays will end up duplicated and will be skipped. This can be a good tradeoff for platforms like Steam Deck or high-end mobile, which are much slower than current generation consoles, but we still want to run the same content without manual scene relighting by artists.

## Ray Reuse

- Per-pixel rotated kernel
  - Kernel size based on depth and center sample RayHitT
  - Depth base edge stopping function
- Less traces, but softens contact shadows
- Great **option** for scaling down below **1 SPP**



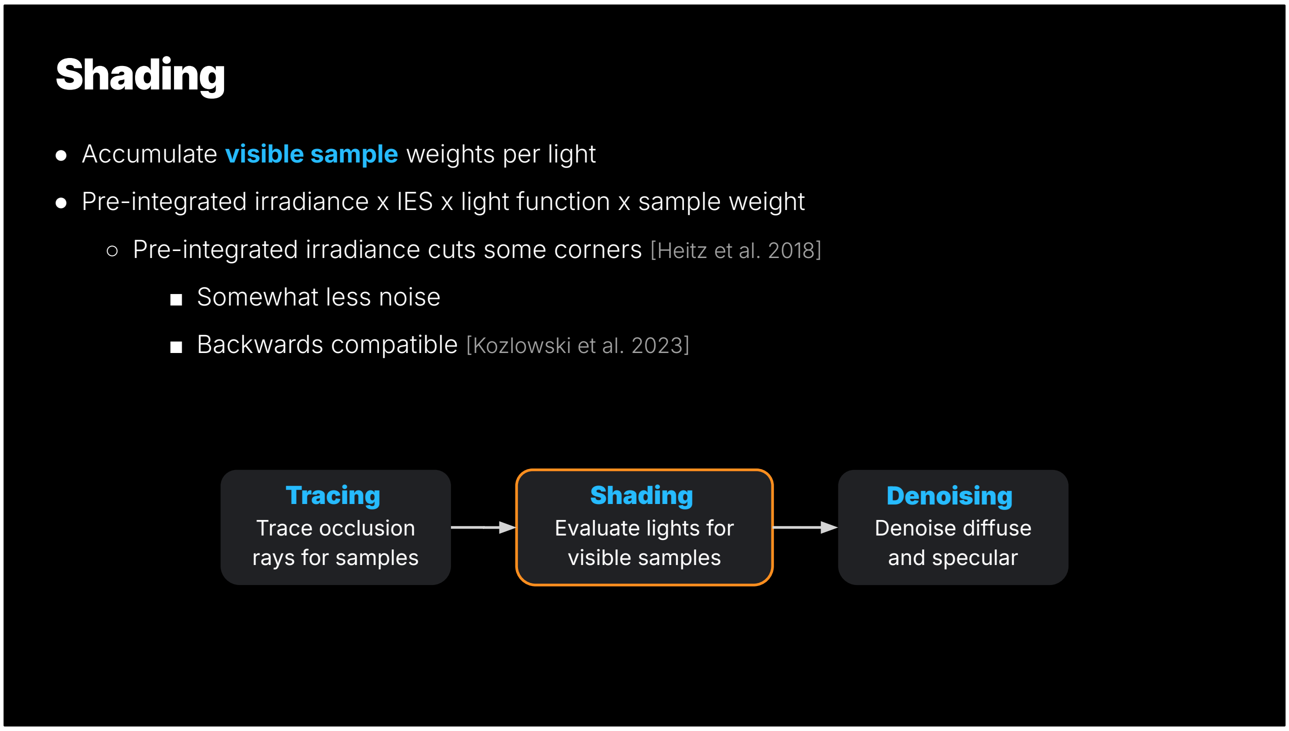
Another way to automatically scale down below current gen consoles is to reuse rays between pixels.

Instead of just shading the center sample we can lookup a few neighbors and see if their rays could be reused.

We use a screen space kernel, which is sized based on the distance to that pixel and ray hit distance. We also check a few heuristics like depth difference between pixels.

The end result has also to be reweighted, as we are picked light in a neighboring pixels with a different probability than it could happen in the center pixel.

It will blur contact shadows a but, but it allows use to trace less rays per pixel, while while keeping lighting stable and sharp.



## Shading

- Accumulate **visible sample** weights per light
- Pre-integrated irradiance x IES x light function x sample weight
  - Pre-integrated irradiance cuts some corners [Heitz et al. 2018]
    - Somewhat less noise
    - Backwards compatible [Kozlowski et al. 2023]

**Tracing**
Trace occlusion rays for samples

→ **Shading**
Evaluate lights for visible samples

→ **Denoising**
Denoise diffuse and specular

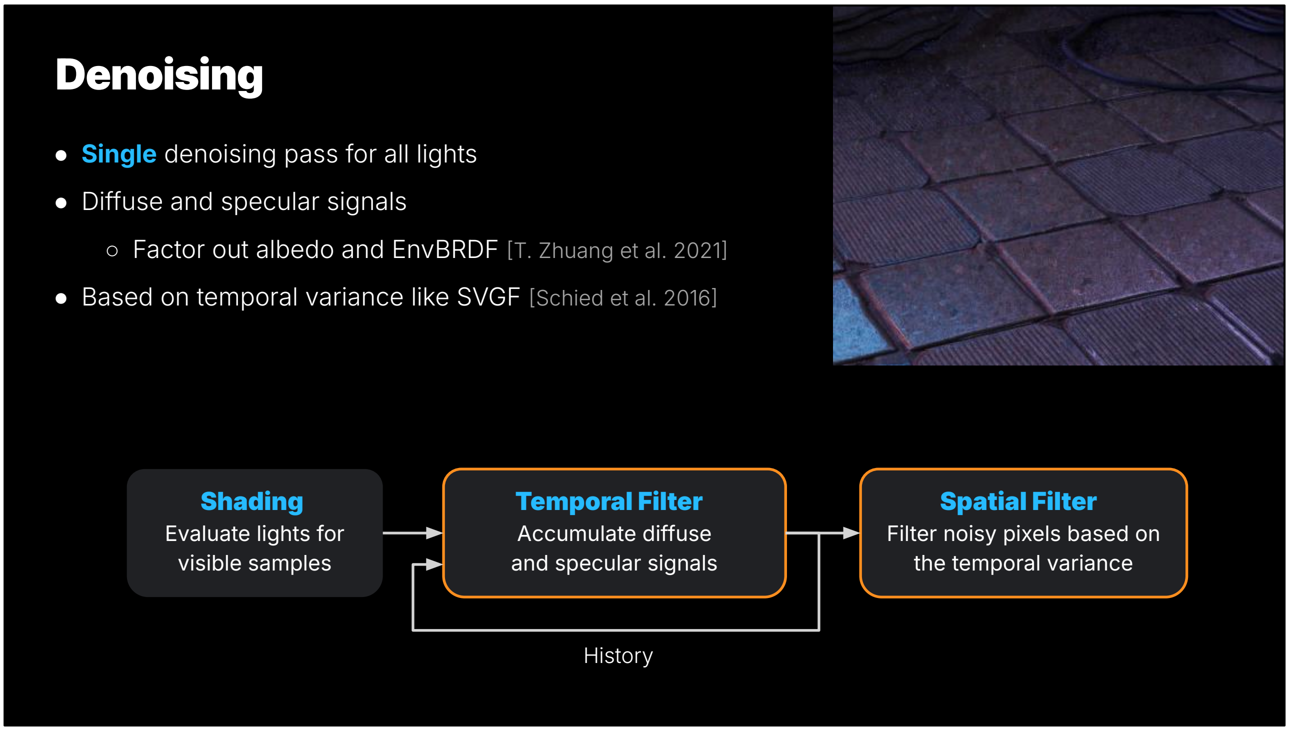
Finally we are ready to shade visible samples.

First we accumulate visible sample weights affecting given pixel. Then we multiply this accumulated weight by a pre-integrated light irradiance (analytic/precomputed light illumination).

Using pre-integrated irradiance means that we assume that we can separate shadowing from shading, which is not correct (introduction in [Heitz et al. 2018] has a great description of this approximation and what issues it can cause), but it makes some things a bit faster and less noisy.

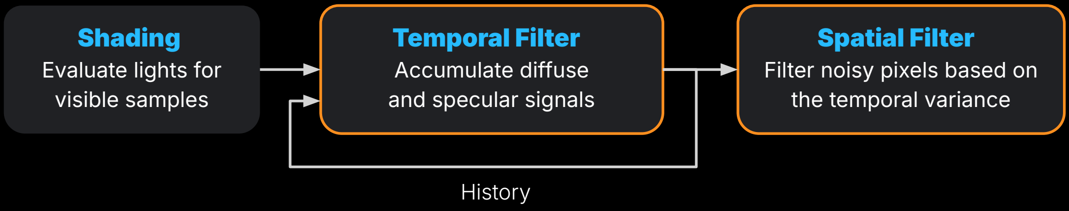
Most importantly it enables backwards compatibility, as our stochastic lights will looks exactly the same as the non-stochastic ones. Which is also the reasoning behind using this approximation for high-end PC ReSTIR implementations.

Still this is definitely something we want to address at some point, as it’s a common request for non-game use cases.



## Denoising

- **Single** denoising pass for all lights
- Diffuse and specular signals
  - Factor out albedo and EnvBRDF [T. Zhuang et al. 2021]
- Based on temporal variance like SVGF [Schied et al. 2016]


Shading
Evaluate lights for visible samples
Temporal Filter
Accumulate diffuse and specular signals
Spatial Filter
Filter noisy pixels based on the temporal variance
History

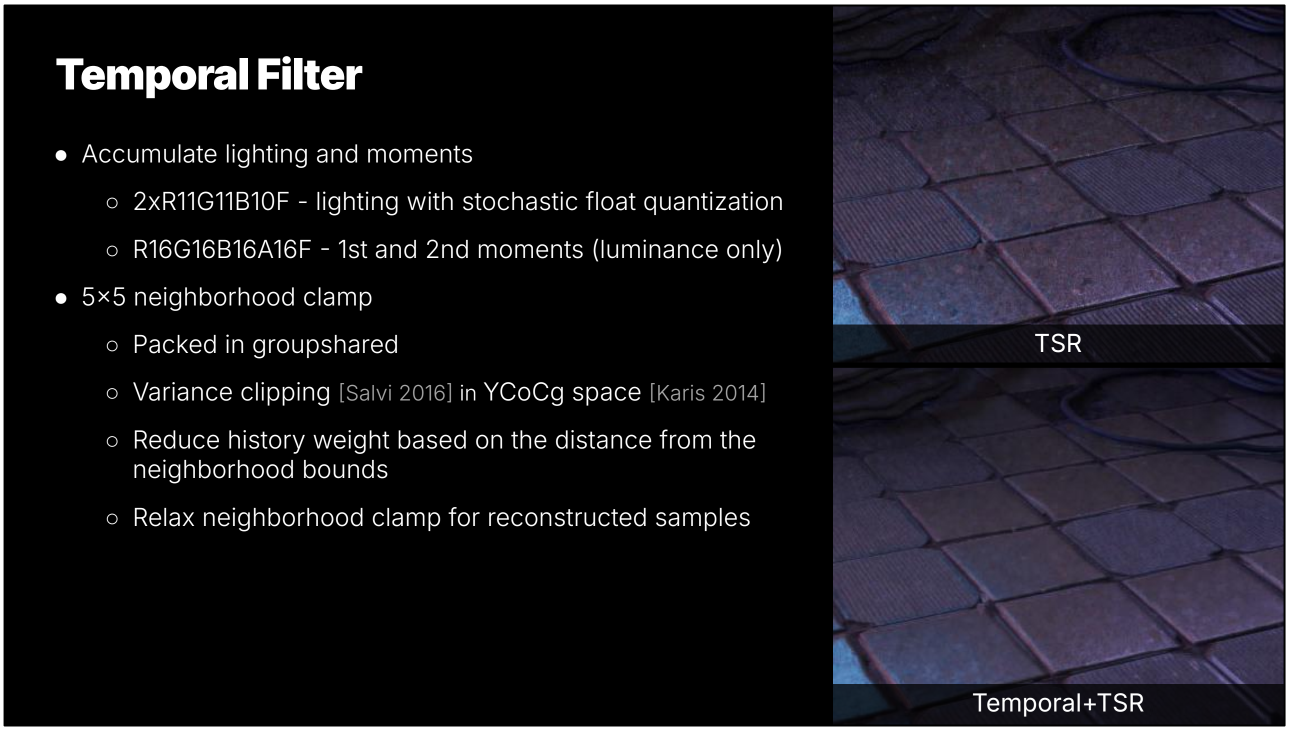
And after shading we run denoising.

Already with just TSR we're getting some pretty good results, but some spots definitively require a dedicated denoiser.

We run one denoising pass for all lights (all lighting is accumulated into 2 render targets) and we denoise denoise diffuse and specular signals separately.

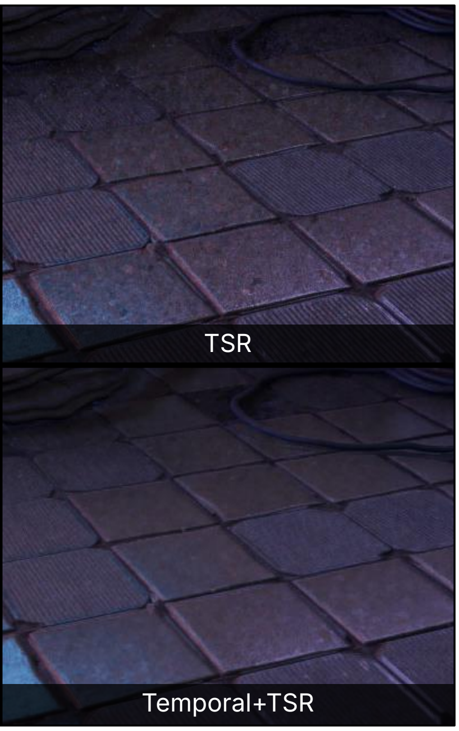
Before denoising we demodulate the signal, which is removing non-stochastic material parts of it. We don't want to be blurring material data like albedo.

The entire denoiser is based on the temporal variance idea introduced by SVGF. We track temporal variance of the signal, which we later use a metric of how noisy is given pixel, so that we know strong spatial filtering we need for that pixel. This way we only blur noisy parts of the image.



## Temporal Filter

- Accumulate lighting and moments
  - 2xR11G11B10F - lighting with stochastic float quantization
  - R16G16B16A16F - 1st and 2nd moments (luminance only)
- 5×5 neighborhood clamp
  - Packed in groupshared
  - Variance clipping [Salvi 2016] in YCoCg space [Karis 2014]
  - Reduce history weight based on the distance from the neighborhood bounds
  - Relax neighborhood clamp for reconstructed samples



On the temporal filter side this is where we reproject history and accumulate lighting and lighting moments.

Lighting is stored in two 32 render targets. One for diffuse and one for specular signal. We are able to use only 32 bit precision thanks to stochastic float quantization, which prevents various banding or color shifting artifacts. It seems to work pretty well in practice and we don't see any issues when comparing with a 64 bit reference.

We also store 1st and 2nd moment of lighting luminance, again one set of moments for diffuse and one for specular.

Float11 and Float16 have a pretty wide range (-2^16 to 2^16), but still in some extreme cases it can be exceeded. In order to prevent any issues with clamping we store all lighting values in pre-exposed space.

We use a 5x5 neighborhood clamp for history rectification. It's a pretty large region, so first we load and pack everything into groupshared.

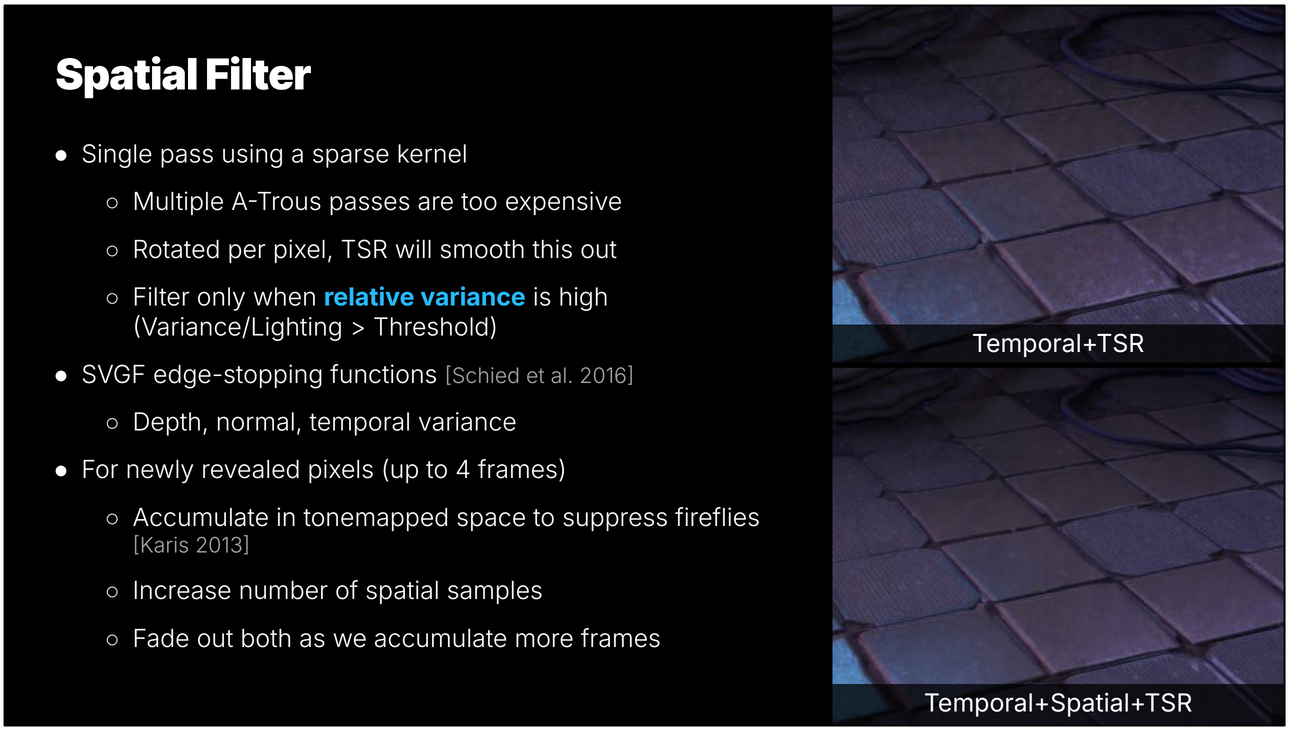
We run variance clipping in YCoCg space, which reduces color shifts during history rectification.

One interesting trick is that when a new data is outside of the neighborhood bounds we speedup history based on the distance from those bounds, which greatly reduces ghosting. The idea here is that if new data is really far away from the history then most likely we should discard that history pretty quickly instead of trying to rectify it and reuse. Here’s exactly how we compute it:

```
float3 ClampedHistoryDiffuseLighting =
clamp(HistoryDiffuseLighting, DiffuseNeighborhood.Center -
DiffuseNeighborhood.Extent, DiffuseNeighborhood.Center +
DiffuseNeighborhood.Extent);
float NormalizedDistanceToNeighborhood =
length(abs(ClampedHistoryDiffuseLighting -
HistoryDiffuseLighting) / max(DiffuseNeighborhood.Extent,
0.1f));
float DiffuseConfidence = saturate(1.0f -
NormalizedDistanceToNeighborhood);
```

DiffuseConfidence is used later to reduce the max number of accumulated frames in history (reduce history weight). We don’t allow it to drop to 0, as we want to blend-in at least a few history frames in order to prevent harsh transitions.

Another one is that when we use downsampling, some of the pixels are reconstructed and less reliable, so we relax the neighborhood clamp for them. This fixes some instability on highly discontinuous features (tiny metal gratings and such), where each frame input can look completely different due to the geometry and sub-pixel jitter.



## Spatial Filter

- Single pass using a sparse kernel
  - Multiple A-Trous passes are too expensive
  - Rotated per pixel, TSR will smooth this out
  - Filter only when **relative variance** is high (Variance/Lighting > Threshold)
- SVGF edge-stopping functions [Schied et al. 2016]
  - Depth, normal, temporal variance
- For newly revealed pixels (up to 4 frames)
  - Accumulate in tonemapped space to suppress fireflies [Karis 2013]
  - Increase number of spatial samples
  - Fade out both as we accumulate more frames

Finally we run a spatial filter in order to cleanup any noise left after temporal filtering. This filter is run after temporal accumulation into history (no recurrent blurring through history), which allows us to make the output sharper and quickly converge on the right result as we accumulate more data after disocclusion.

Spatial filter is a bit different from standard ones. Instead of multiple expensive A-Trous passes, we use a single sparse kernel, which is rotated per pixel. This works thanks to TSR (our temporal AA + upscaler), which can cleanup any filter holes.

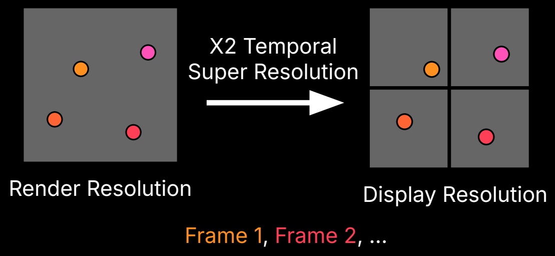
We only apply it to pixels where relatively temporal variance is high, which makes things both faster and sharper. We don’t need to output a perfect image here, as some noise can be handled by TSR.

Using absolute variance helps in cases like strong specular lighting, where a tiny error can cause large absolute temporal variance due to numerical precision issues, even if those pixels aren’t noisy.

For handling disocclusion we assume that input pixels have a constant temporal variance (we didn’t find estimating variance worth of extra overhead), increase the number of spatial samples and accumulate in tonemapped space which removes fireflies. As we accumulate more frames we fade out all those assumptions. After accumulating 4 frames of history data we completely disable all disocclusion handling.

## Problem With Denoising When Using Temporal Upscaling

- Denoising accumulates in **render** resolution
- Temporal upscaling accumulates in **display** resolution
- Denoiser accumulates unrelated pixels
    - Blurry lighting



There's one more issue with denoising, which we need to discuss here. Namely denoising accumulates and filters in lower resolution than upsampling.

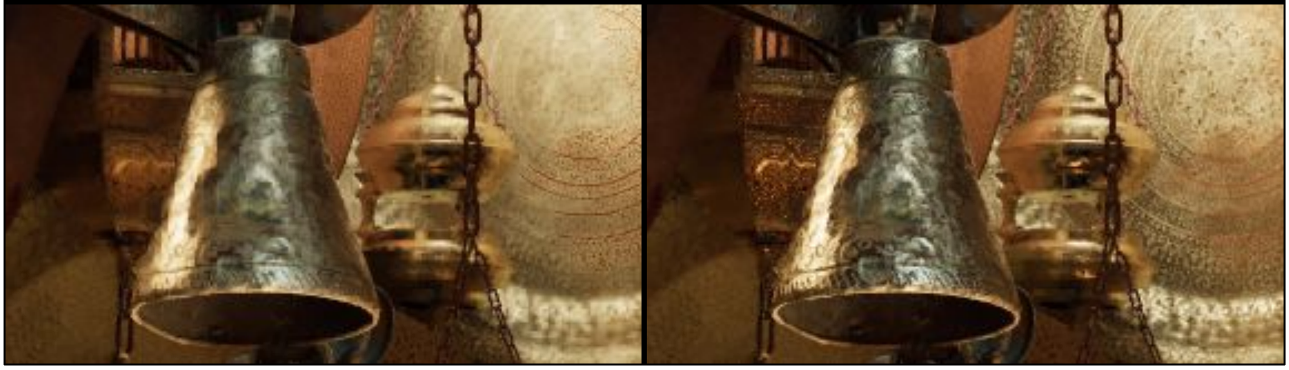
This means that a single pixel in denoisers history corresponds to multiple pixels after upsampling. So when denoising we either have to accumulate those unrelated pixels causing blurry lighting, or we need to discard history each frame causing noise.

This issue isn't that prominent with per-light denoisers. They denoise only the shadow term using shadow mask or ratio estimator, while lights are evaluated in a non-stochastic fashion each frame per pixel. In that case this issue is mostly visible only around penumbras. When denoising merged irradiance from multiple lights it's more problematic, as now it's blurring all lighting, which can be pretty visible, especially on faces where human eyes can quickly pickup lack of specular detail. We did try to do something similar as per-light ratio estimator denoisers by finding the most important light (BRDF) per pixel and factoring it out. This doesn't work well in practice as light selection itself is stochastic, so now that light selection noise will be visible on screen in more complex lighting scenarios. Factoring out BRDF itself isn't also great, as now all temporal and spatial filters will operate on incorrect values. Each pixel could be divided by a different important light and it can cause some weird artifacts in spots where most important light selection is discontinuous and we try to blend those values.

Ideally our denoiser's history should be also in display resolution, so that we can avoid this issue, but in practice this can be prohibitively expensive on consoles. It's common to render at 1080p and then upsample to 4k, which would require now for the denoiser to run temporal filtering at 4k making the entire pass ~4x slower. Still something what we want to research further, at least as a way to scale up to high-end PC GPUs.

## Shading Confidence

- Pass **some noise** to TSR
    - Reduce history weight and spatial filtering
- % of energy sampled from a visible light list is a good heuristic
- Requires some temporal stabilization



As we saw before there are usually only a few important lights per pixel. And if we are able to sample 80% of the energy for a pixel then we can reduce the amount of denoising or maybe even just pass signal directly to TSR.

We have a pretty good heuristic for that. Visible light list gives a total amount of energy which could affect a given pixel. We also know what we successfully sampled this frame. For example if there’s one light with a strong specular highlight responsible for 80% of the energy and we sampled it this frame then we don’t really need to do any denoising here and can pass that signal directly to TSR.

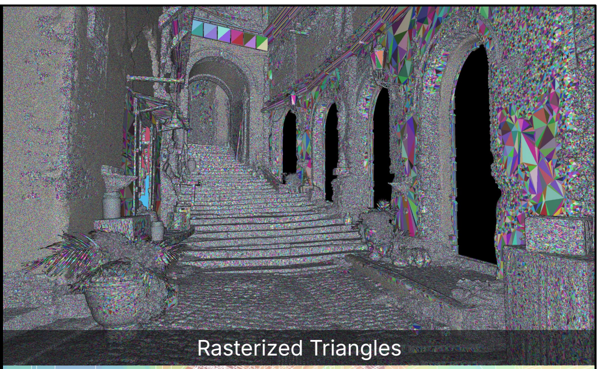
This heuristic requires a bit of temporal stabilization (temporal accumulation), as sampled light set may change a lot from frame to frame, but overall works pretty well in practice. It not only helps with output image sharpness, but also with common denoising issues like specular highlight ghosting or blurriness, which is hard to reproject correctly, especially on non-flat surfaces.

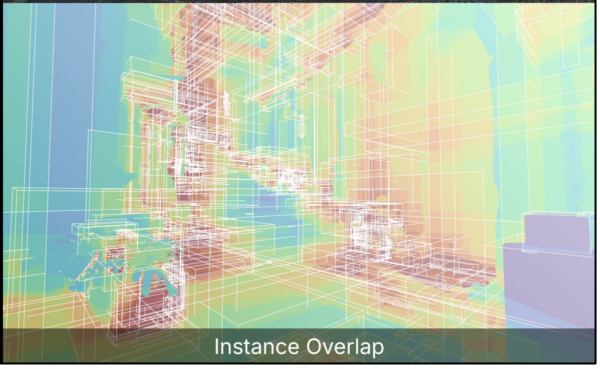
It may be hard to see here, but it really makes this cropped out image on right much sharper.

And now I’m going to hand it off to Tiago.

## Ray Tracing

- All shadowing is handled by tracing rays
- Can't represent everything in BVH
  - Highly detailed environments
  - High **memory overhead**
  - Expensive **build** and **tracing**
- Need:
  - Simplified meshes
  - Aggressive instance culling
  - Aggregate representations

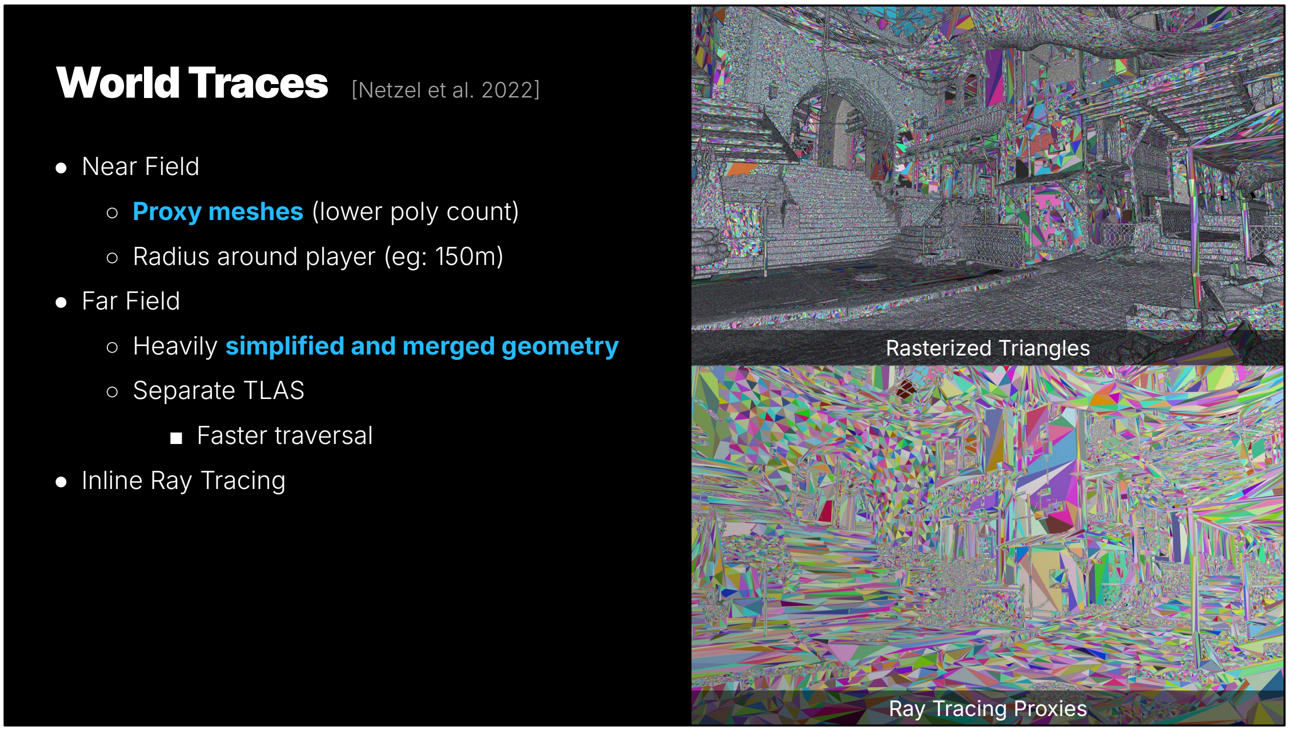

Rasterized Triangles


Instance Overlap

The main method we use to compute visibility for the light samples is hardware ray tracing, which allows us to avoid the overhead of rendering and maintaining multiple shadow maps per frame, which is especially problematic when dealing with large numbers of dynamic lights.

While shadow quality is extremely important, we can't represent all objects in BVHs with the amount of detail that is used to render the main view. First of all, there's a high memory overhead per vertex, and UE projects using Nanite tend to use highly detailed meshes. That also means BVH builds and tracing can become too expensive and on top of that, kitbashed environments result in a lot of instance overlap, which further increases tracing costs.

So instead we need to use simplified proxy meshes, as well as aggressive instance culling and aggregate representations to reduce the complexity of the levels in ray tracing.



## World Traces [Netzel et al. 2022]

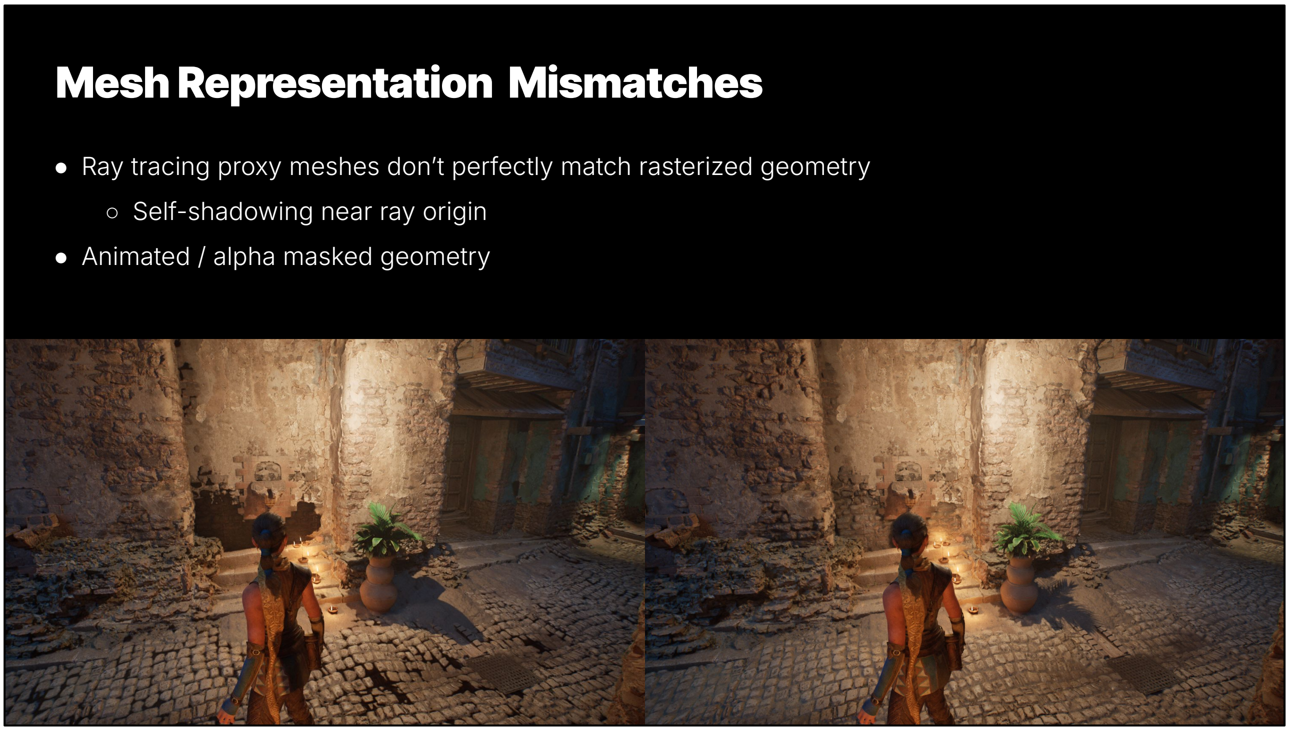
- Near Field
  - **Proxy meshes** (lower poly count)
  - Radius around player (eg: 150m)
- Far Field
  - Heavily **simplified and merged geometry**
  - Separate TLAS
    - Faster traversal
- Inline Ray Tracing

Rasterized Triangles
Ray Tracing Proxies

We use 2 ray tracing representations of the levels:

- In the near field, which covers an area of 150 meters around of the player camera, we use proxy meshes.
  - These are lower polygon representations of the original geometry, that provide a good balance between performance and accuracy, ideal for close proximity tracing.
- If rays don’t hit any near field mesh and extend outside it’s radius, we also trace against a more aggressively simplified representation of the level.
  - In this representation we merge instances and significantly reduce triangle count to keep ray traversal cost down.
  - The far field representation is stored in a separate TLAS, which makes those traces faster since the BVH complexity is lower.

And we use inline ray tracing by default, since we get better performance on current generation of consoles.



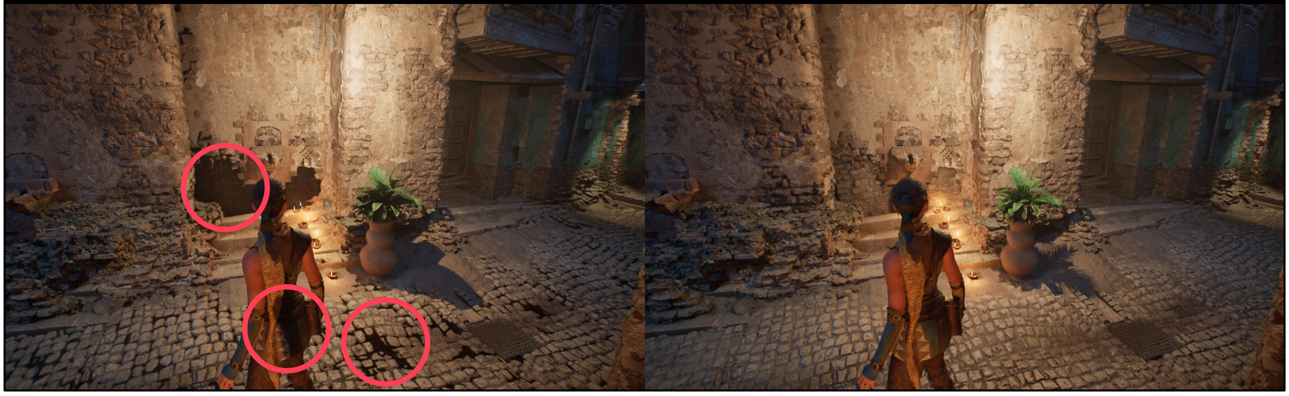
## Mesh Representation Mismatches

- Ray tracing proxy meshes don’t perfectly match rasterized geometry
  - Self-shadowing near ray origin
- Animated / alpha masked geometry

Using proxy meshes for ray tracing has one major downside which is mismatches between the proxy and the rasterized geometry.

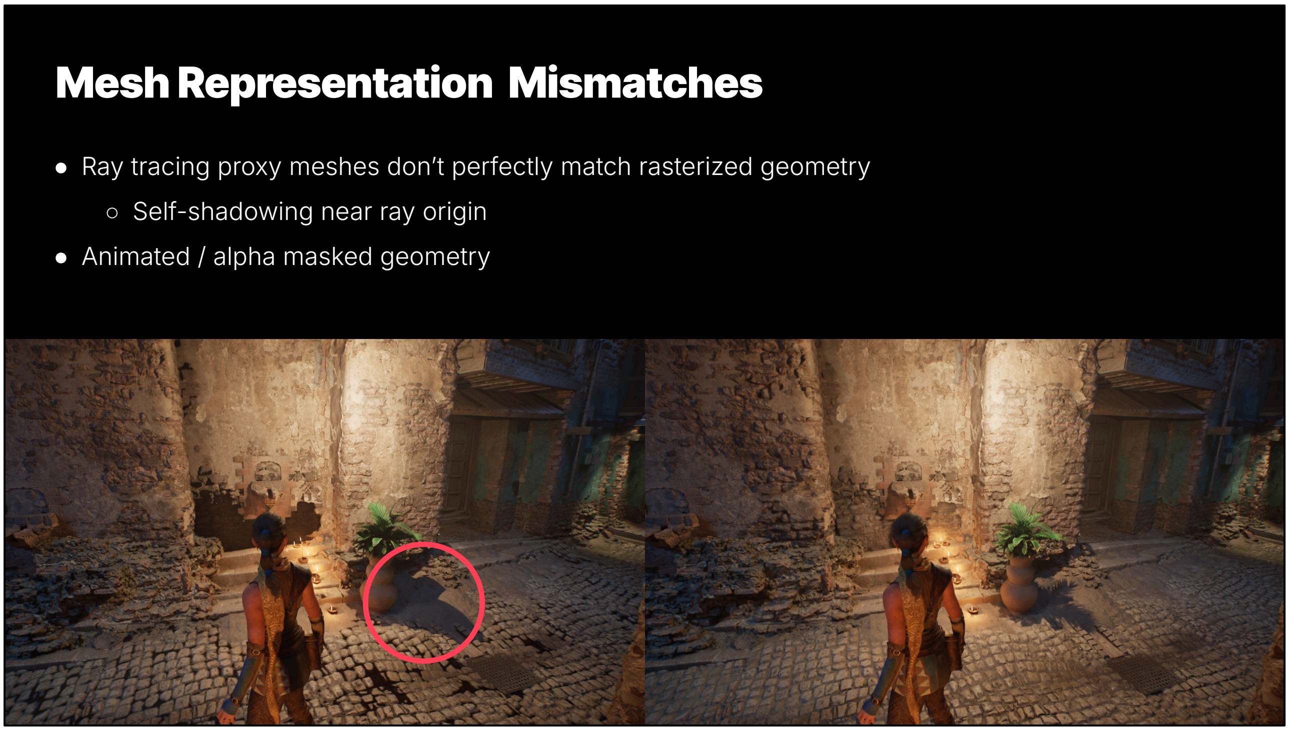
## Mesh Representation Mismatches

- Ray tracing proxy meshes don't perfectly match rasterized geometry
  - Self-shadowing near ray origin
- Animated / alpha masked geometry



Those mismatches cause incorrect self-shadowing, since rays are traced from the rasterized geometry surface and they might immediately intersect proxy mesh triangles.

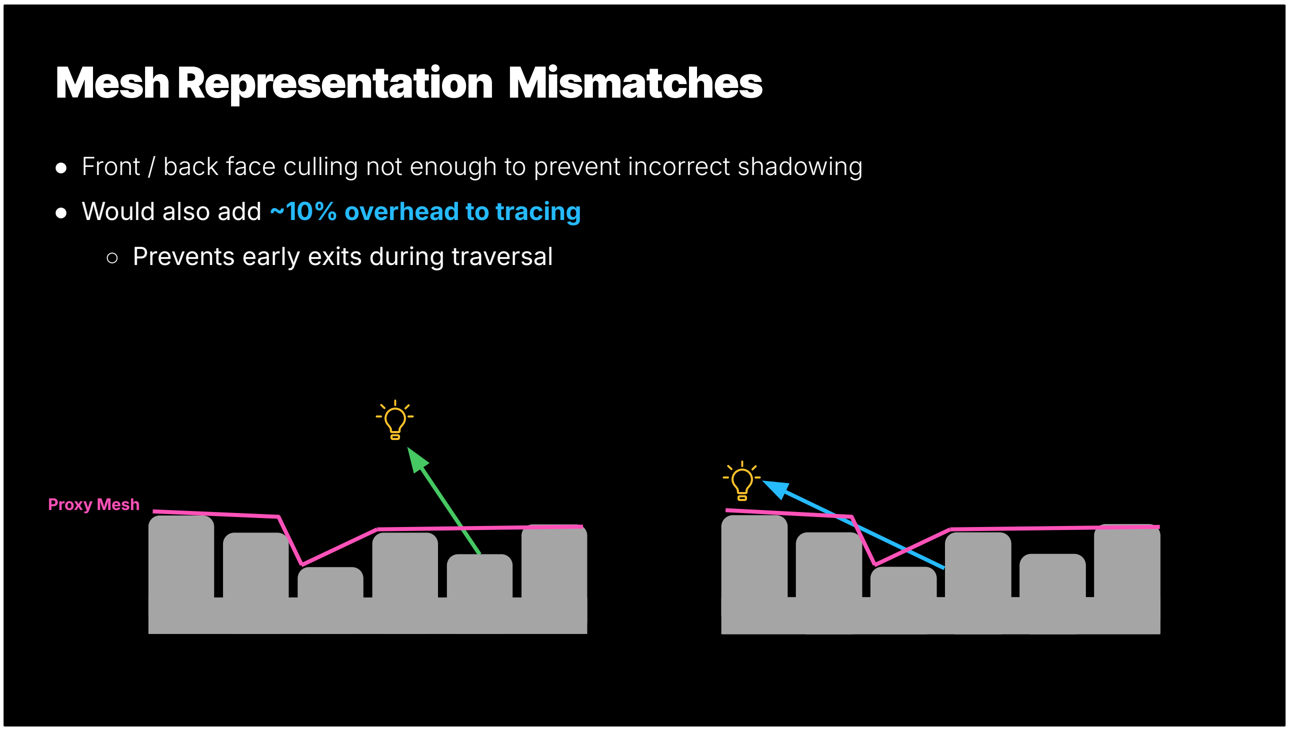
In these screenshots, we can see incorrect shadowing on the wall in front of the player, because that mesh uses tessellation that we can't efficiently represent in ray tracing. Similarly on the floor, there are mismatches that cause more self shadowing. And even on the player character there's simulated cloth mesh that is not well represented in ray tracing and results in an incorrect shadow.



## Mesh Representation Mismatches

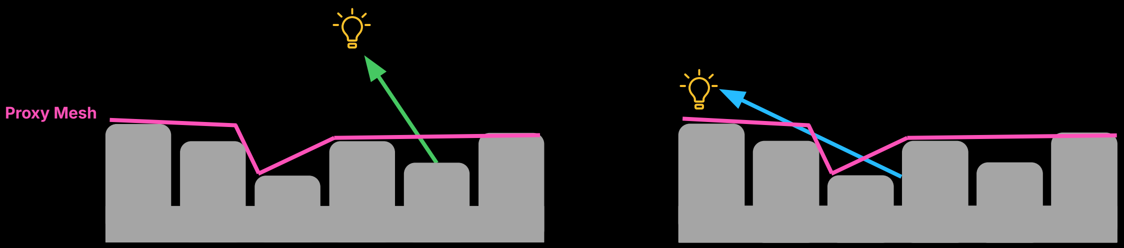
- Ray tracing proxy meshes don't perfectly match rasterized geometry
  - Self-shadowing near ray origin
- Animated / alpha masked geometry

There are also the usual issues with animated and alpha masked geometry, which are especially expensive to represent accurately and trace rays against.



## Mesh Representation Mismatches

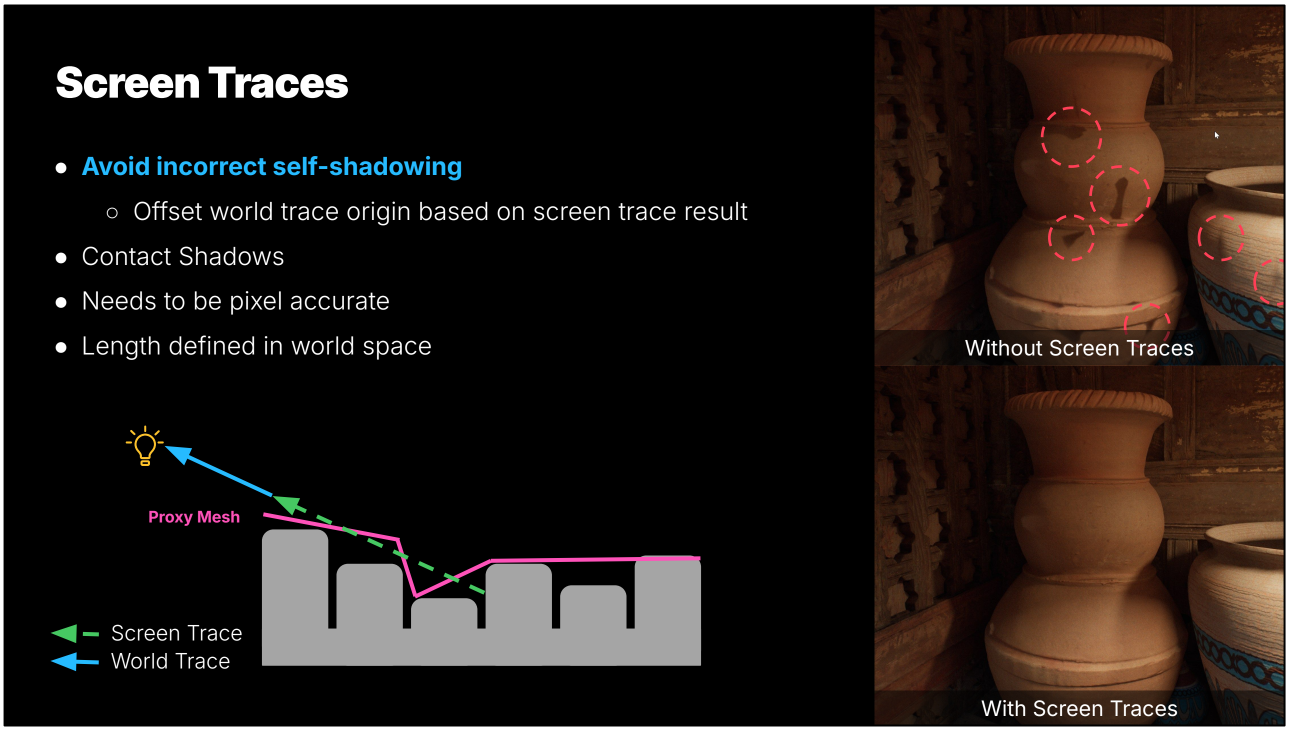
- Front / back face culling not enough to prevent incorrect shadowing
- Would also add **~10% overhead to tracing**
  - Prevents early exits during traversal



Unfortunately these mismatch issues can’t be avoided by simply using front or back face culling on the rays.
On the diagram on the left, we can see that back face culling is enough to prevent incorrect occlusion. However on right, we can see a situation where the ray also hits a front face of the proxy which results in incorrect occlusion.

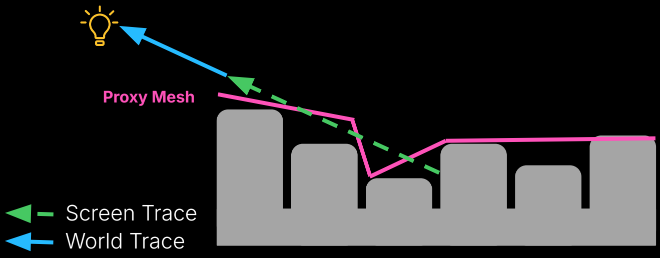
On top of that, using either front or back face culling adds about 10% overhead to tracing, since it prevents some early exists during the ray traversal.

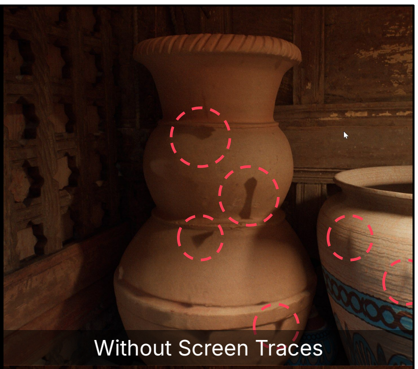
All of this means we can’t just rely on proxy meshes and instead need fallback mechanisms to workaround these issues.

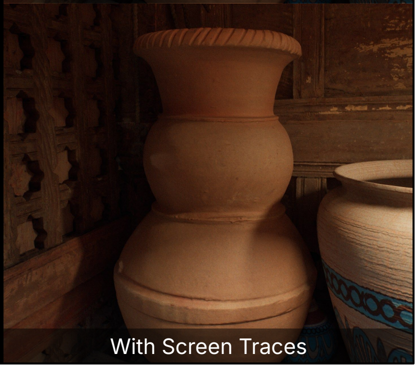


## Screen Traces

- **Avoid incorrect self-shadowing**
  - Offset world trace origin based on screen trace result
- Contact Shadows
- Needs to be pixel accurate
- Length defined in world space





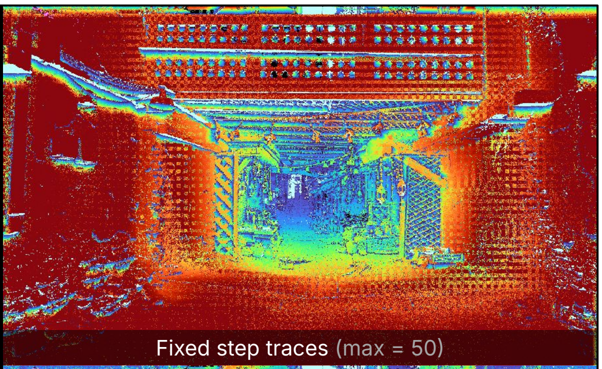


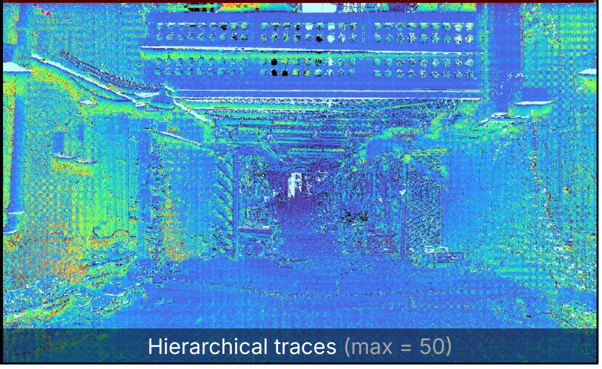
So our main method to avoid incorrect self-shadowing, is screen space ray tracing. Since screen rays are traced against the rasterized geometry depth buffer, we can rely on them to move away from surfaces without incorrectly intersecting the proxy geometry. We can then offset the origin of the world trace based on how far the screen trace got, similar to what this diagram shows, which avoids most of those self shadowing issues.

Another benefit is that we get contact shadows with detail that is not represented in the BVHs. However unlike typical contact shadows where we can usually get away with a small number of taps and short ray length, we need our screen traces to be pixel accurate otherwise we might skip over occluders, and the ray length is defined in world space which means in some cases we need to trace all the way across the screen, which is too expensive to do with fixed stepping.

## Screen Traces

- **HZB tracing** [Uludag 2014]
- Minimize screen space tracing artifacts
  - Assume low surface thickness
  - Limit ray length (eg: 20cm)
  - `min(Point, Bilinear)` depth [Stachowiak 2024]
- Increase thickness when instances are not represented in HWRT


Fixed step traces (max = 50)

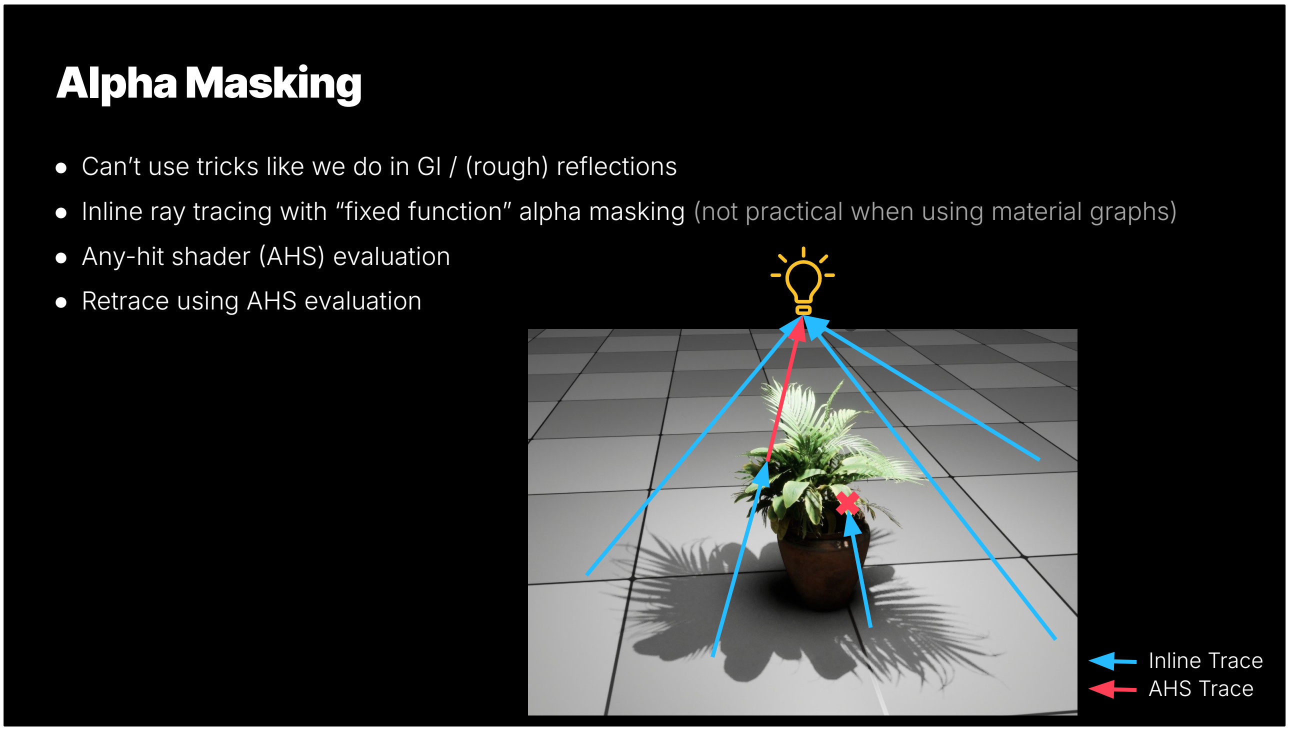

Hierarchical traces (max = 50)

We rely on HZB tracing to be able to quickly but accurately skip empty space without testing against every pixel along the ray.

We want to avoid typical screen space artifacts as much as possible:

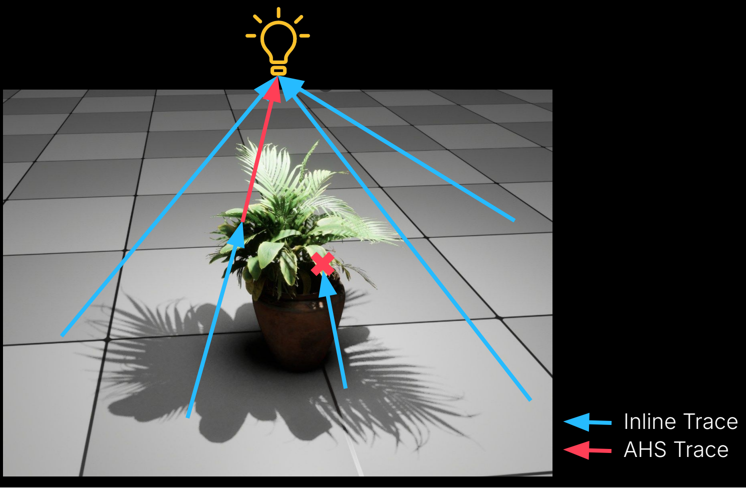
- So we assume a very conservative low surface thickness by default, meaning that in most cases we fallback to hardware ray tracing when rays go behind surfaces instead of assuming that there was an intersection.
- Another issue is that long screen traces can cause noticeable artifacts due to depth buffer aliasing, so in practice we limit the length to avoid those issues.
- Tracing against the depth buffer can also cause it's own self-shadowing issues, which we avoid by combining both point and bilinear sampled depth.

We also track whether instances are represented in ray tracing using a stencil bit, and if the ray goes behind one of those pixels we increase the estimated thickness to get a bit more contact shadows from those instances.



## Alpha Masking

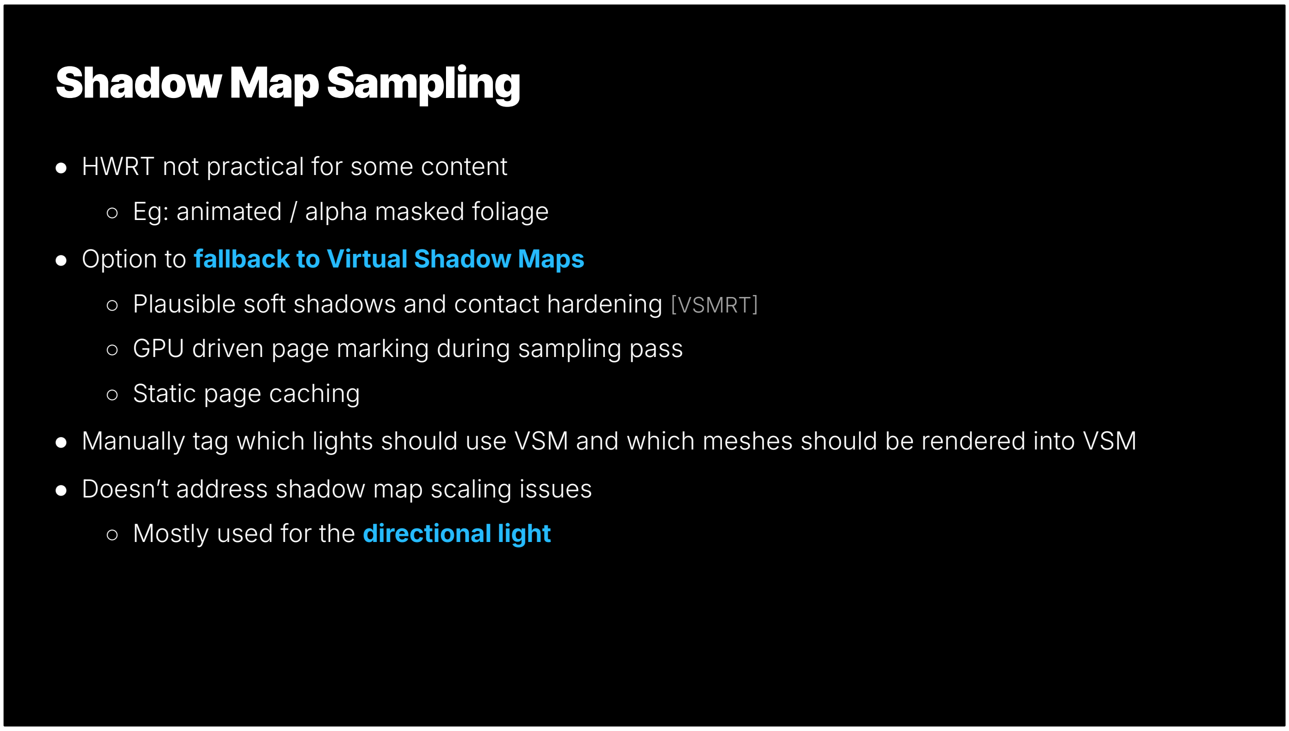
- Can't use tricks like we do in GI / (rough) reflections
- Inline ray tracing with "fixed function" alpha masking (not practical when using material graphs)
- Any-hit shader (AHS) evaluation
- Retrace using AHS evaluation



While for GI or reflections we can usually get away with tricks like shrinking or culling triangles during BVH build based on alpha mask to cheaply approximate occlusion, those methods don't work well for direct shadows.

Ideally, we would support some sort of "fixed function" alpha masking that we could directly evaluate when using inline ray tracing, but since artists are able to define the alpha mask using material graphs that's not really practical for us, so we are left with a few other options:

- We can trace all rays using full any hit evaluation, which gives us accurate results, but on the other hand we can't benefit from inline ray tracing, so it adds a lot of overhead on consoles.
- The second approach is to do the usual lightweight inline trace first, and if we hit an instance that needs material evaluation, we trace a "continuation" ray with any hit evaluation enabled. This approach gives us accuracy where needed but without losing inline ray tracing benefit for all rays.



## Shadow Map Sampling

- HWRT not practical for some content
  - Eg: animated / alpha masked foliage
- Option to **fallback to Virtual Shadow Maps**
  - Plausible soft shadows and contact hardening [VSMRT]
  - GPU driven page marking during sampling pass
  - Static page caching
- Manually tag which lights should use VSM and which meshes should be rendered into VSM
- Doesn't address shadow map scaling issues
  - Mostly used for the **directional light**

There are still situations and content that makes using hardware ray tracing simply not practical on current gen hardware:

- The overhead of running AHS might simply be too high, specially since it's usually used on dense foliage with lots of overlapping triangles.
- Or there might be heavily instanced animated meshes which require a lot of memory and time spent building BVHs.

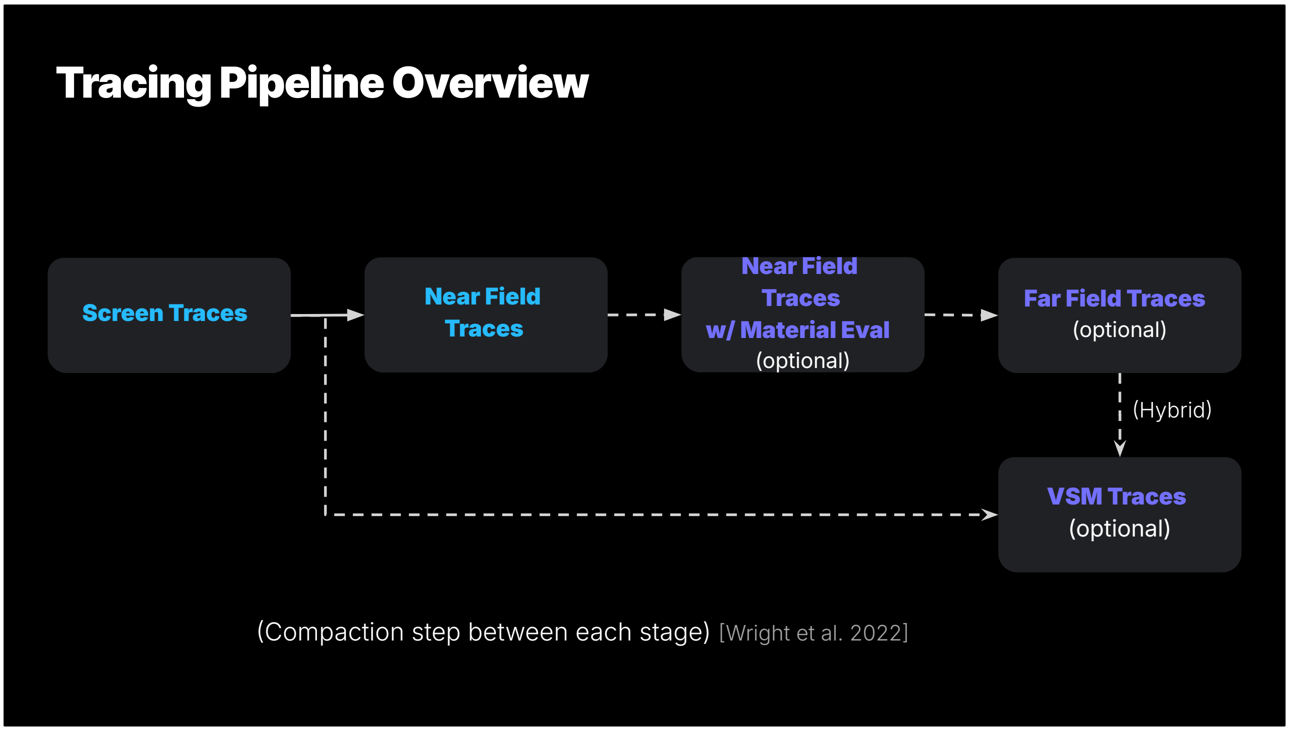
To workaround this, we also support falling back to virtual shadow maps instead of using hardware ray tracing for specific lights.

Virtual shadow maps support features that are useful for our purposes:

- They are capable of generating plausible soft shadows, so they mix well with ray traced shadows from other lights.
- During the light sampling pass, we can also mark which pages are sampled to avoid unnecessary work rendering meshes into the shadow maps.

This fallback method doesn't need to be a binary choice. Artists can manually tag which instances should be rendered into shadow maps, and rely on HWRT for instances that are well represented in ray tracing.

However, it's important to note that this fallback option doesn't address shadow map scaling issues. We have to pay for the BVH cost once (and anyway it's required for Lumen), but there's almost no per light overhead. With VSM it's different, as now each light has an extra overhead required to prepare shadow maps per light, and even with complex caching VSM schemes this can accumulate really quickly pushing the entire technique outside of a reasonable 60hz budget. So in practice this is used only for the directional light since representation mismatches tend to be more noticeable there and extending BVH to cover the entire scene view range would be pretty costly.



## Tracing Pipeline Overview

Screen Traces
Near Field Traces
Near Field Traces w/ Material Eval (optional)
Far Field Traces (optional)
(Hybrid)
VSM Traces (optional)
(Compaction step between each stage) [Wright et al. 2022]

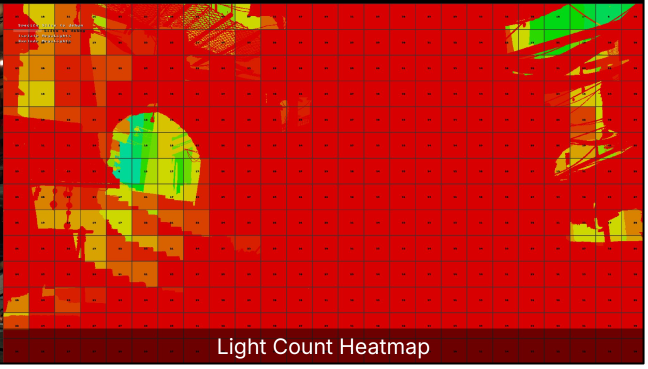
Putting it all together in this diagram:

- The core of our tracing pipeline consists of screen traces, which are always traced first to avoid incorrect self-shadowing, followed by near field traces, which handle most of the rays.
- If the ray hits an object that uses an alpha masked material, we have the option to retrace the ray with material evaluation.
- Similarly, if the ray extends past the near field radius, we can retrace against the far field representation of the level.
- And finally, for lights using VSM we also sample the shadow map.

We also run also compaction between each step to maintain coherency. This compaction step processes 16x16 trace tiles per group in order to maximize coherency.

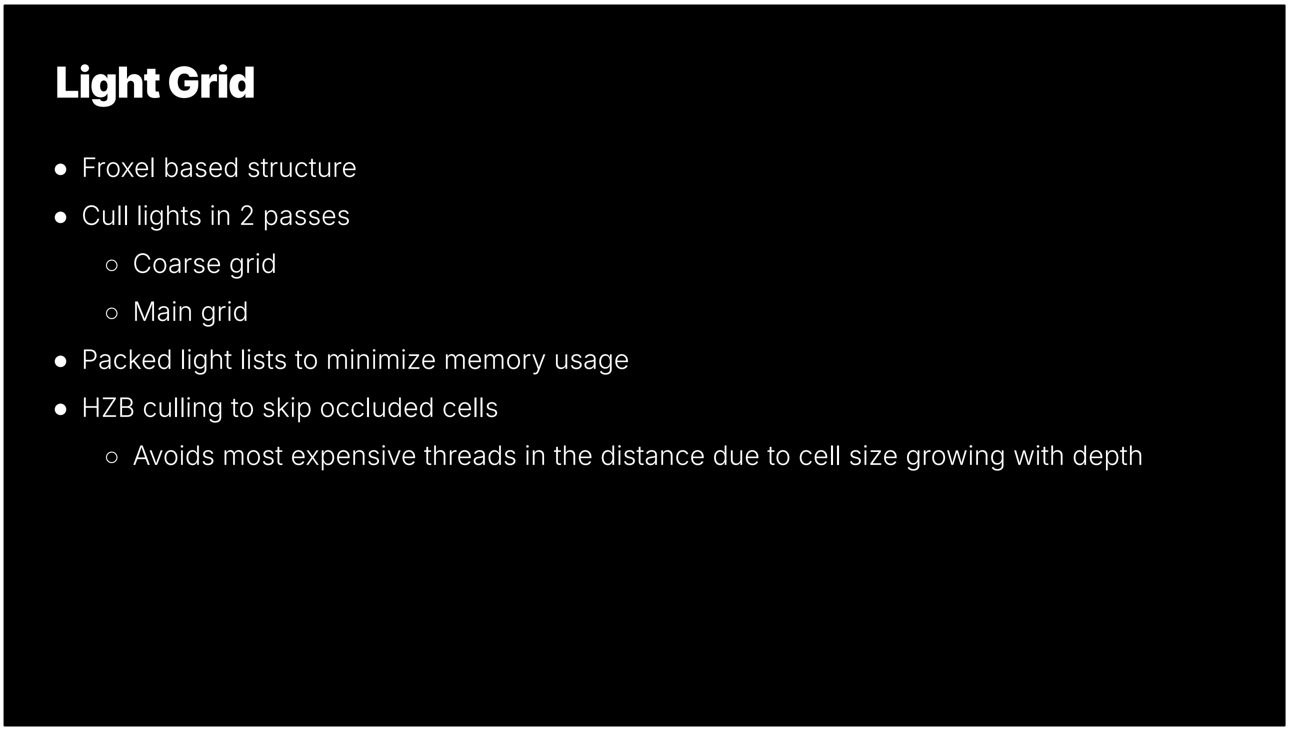
## Unconstrained lighting setups

- 1000s of lights in view frustum / level
  - Significant pressure on related systems
- Increased usage of complex light types and features
  - Eg: rect lights, textured lights



Light Count Heatmap

Most games are not actually shipping with thousands of lights, so even in a mature engine like UE, there was quite a bit of performance left to squeeze when we start pushing large numbers of lights.
For example, our light count heat map visualization was configured with 8 lights as the max threshold by default, which shows what the requirements were when it was implemented.
Artists also start to use complex light types (such as rect and textured lights) a lot more, so we need to make sure those are handled efficiently across the engine.



## Light Grid

- Froxel based structure
- Cull lights in 2 passes
    - Coarse grid
    - Main grid
- Packed light lists to minimize memory usage
- HZB culling to skip occluded cells
    - Avoids most expensive threads in the distance due to cell size growing with depth

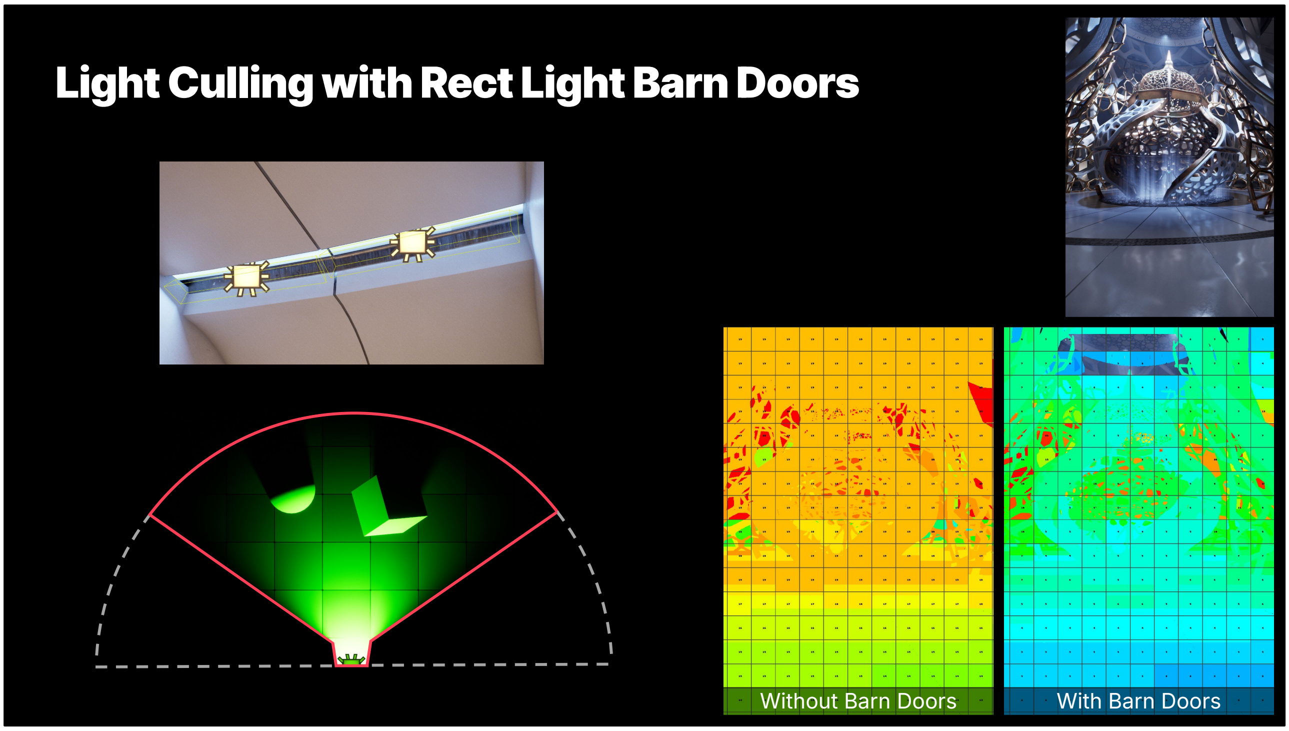
Our light grid implementation started to feel pressure when scaling up to thousands of lights, so we had to improve the build logic to better handle large numbers of lights.

Our updated implementation is described below, which reduced the time spent building the grid from over 0.6ms to under 0.2ms in the MegaLights demo. There has been a lot of research and publications about this topic in the last decade with highly efficient methods so this likely could be optimized further.

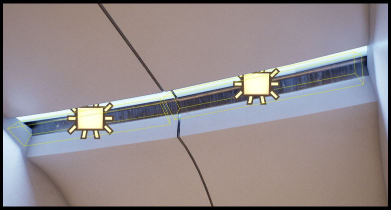
The existing froxel based structure was kept, but light injection was split into 2 passes. First we cull lights into a coarse grid using 1 thread group per cell, since it's quite slow to loop over all lights using a single thread. Then we use the results in the coarse grid to cull lights into the main grid that we use during rendering. In this second pass, the number of lights that need to be checked per cell is significantly lower so using a thread per cell is usually enough. A full thread group per cell can result in underutilization, so whether to use one thread or one thread group to process a cell can be decided dynamically based on the number of lights. Waves with 32 lanes can provide a good middle group on platforms that support it.

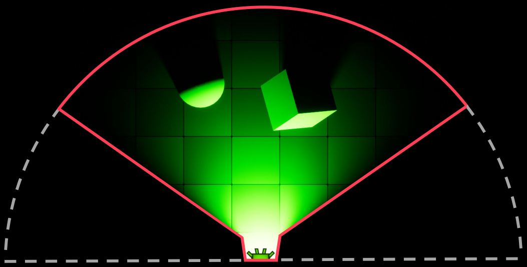
We need light lists of each cell to be packed, since allocating memory per grid cell for worst case would result in a lot of wasted memory. This requires counting the number of relevant lights per cell, allocating space in a buffer, and then writing the list indices. In order to avoid having to loop over the global list list twice (once for counting and once for writing), we first write the relevant light list into LDS (with spill over into a linked list stored in a shared buffer in global memory), allocate space in the actual grid buffer, and then copy the list from LDS + spillover linked list into that buffer.

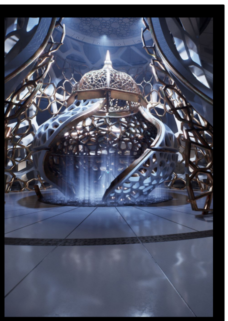
Often majority of light grid cells are occluded and HZB culling allows us to greatly reduce light grid building overhead. Due to the non-linear nature of the froxel grid, cell size grows with depth, so those distant cells are usually more expensive since they’re affected by many lights and threads can become bandwidth bound writing all those light indices.

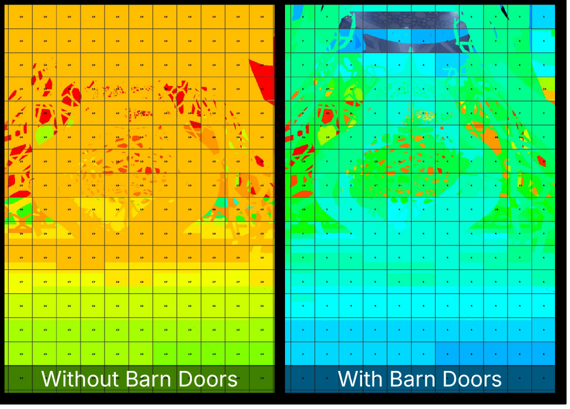


## Light Culling with Rect Light Barn Doors









Having tight culling bounds is very important to maximize performance. Similar to light grid building, this is something that has been extensively researched with many publications covering different aspects of it, but one type of light that seems to be often overlooked in terms of culling is rect lights (likely because they haven't been used much in games until now). We have found that by taking barn doors into account to tighten the culling we can significantly reduce the number of cells they affect.

## Tile Classification

- Need to minimize register pressure
- Classify based on both material and **light types**
- Specific shader permutations for rect and textured light support
    - **Increases occupancy by ~20%** on average

Register pressure and occupancy is also something we need to consider in our shaders, since they need to handle a bunch of material types and light features.
To address this, we run tile classification based on both material type, and the types of lights that affect each tile, so that we only have to pay for the worst case VGPR usage where it's necessary.
In particular, the logic to support rect and textured lights adds quite a bit of overhead, so we have specific shader permutations for those. In the MegaLights demo, doing this extra classification based on light types improved occupancy by around 20% with little overhead since we track which types of lights affect each light grid cell while building the light grid.

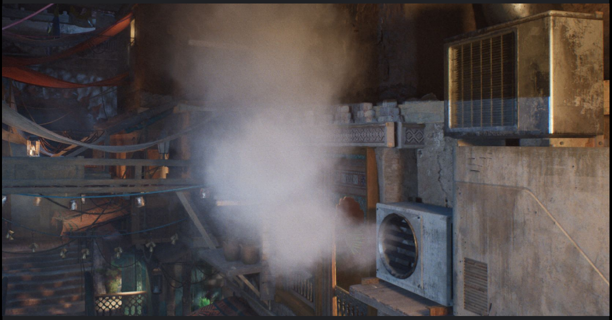
This wraps up our opaque pipeline, so let's move on to translucency and volumetric fog.

## Volumetric Fog and Translucency

**Volumetric Fog**

Froxel Grid

**Translucent Materials**

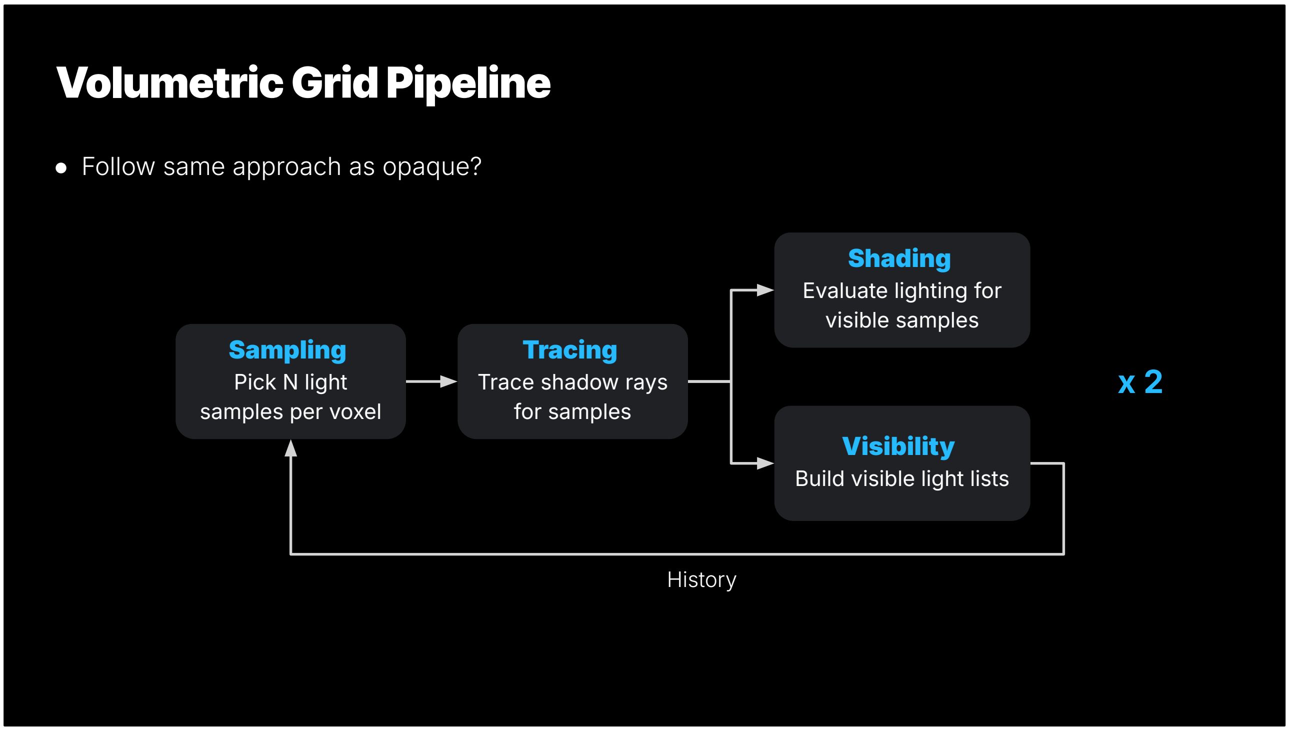


World Space Voxel Grid

In UE we calculate and store lighting for volumetrics in 2 structures:

- For volumetric fog, we use a camera aligned froxel grid.
- And for particles effects, we have a world space grid around the camera, where we store 2-band Spherical Harmonics per voxel.

These structures are updated every frame by injecting lights into them and shadowing is typically calculated using shadow maps. However when mainly relying on ray traced shadows we don't have shadow maps and injecting so many lights is also very expensive, so these structures have been upgraded to integrate with the new system.



## Volumetric Grid Pipeline

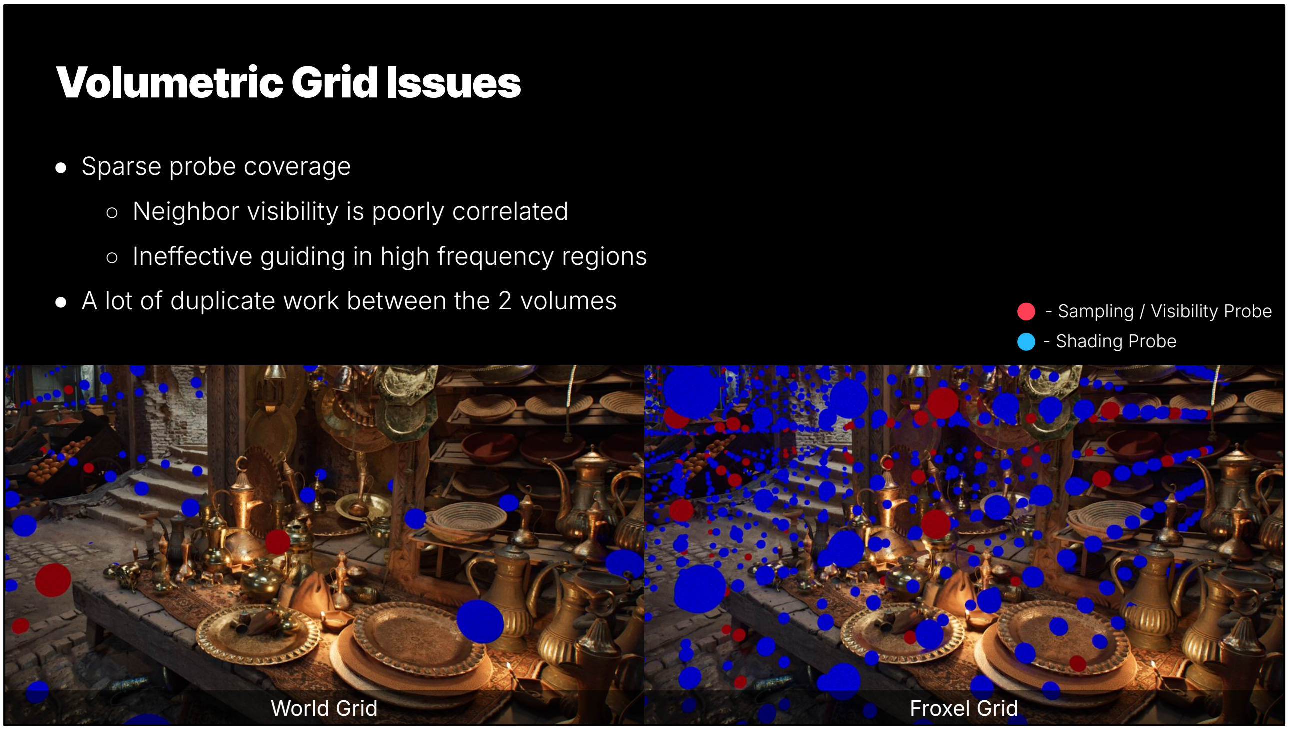
- Follow same approach as opaque?

Sampling
Pick N light samples per voxel
Tracing
Trace shadow rays for samples
Shading
Evaluate lighting for visible samples
Visibility
Build visible light lists
x 2
History

The general approach used for opaque pixels can be applied to volumes as well:

- We can run sampling at half res and pick N light samples per voxel
- Trace shadow rays
- And evaluate lighting for the visible samples using phase function for volumetric fog and the diffuse BRDF for the translucency volume
- Finally we build visible light lists for the next frame

And we have to do this twice, once for each volume.

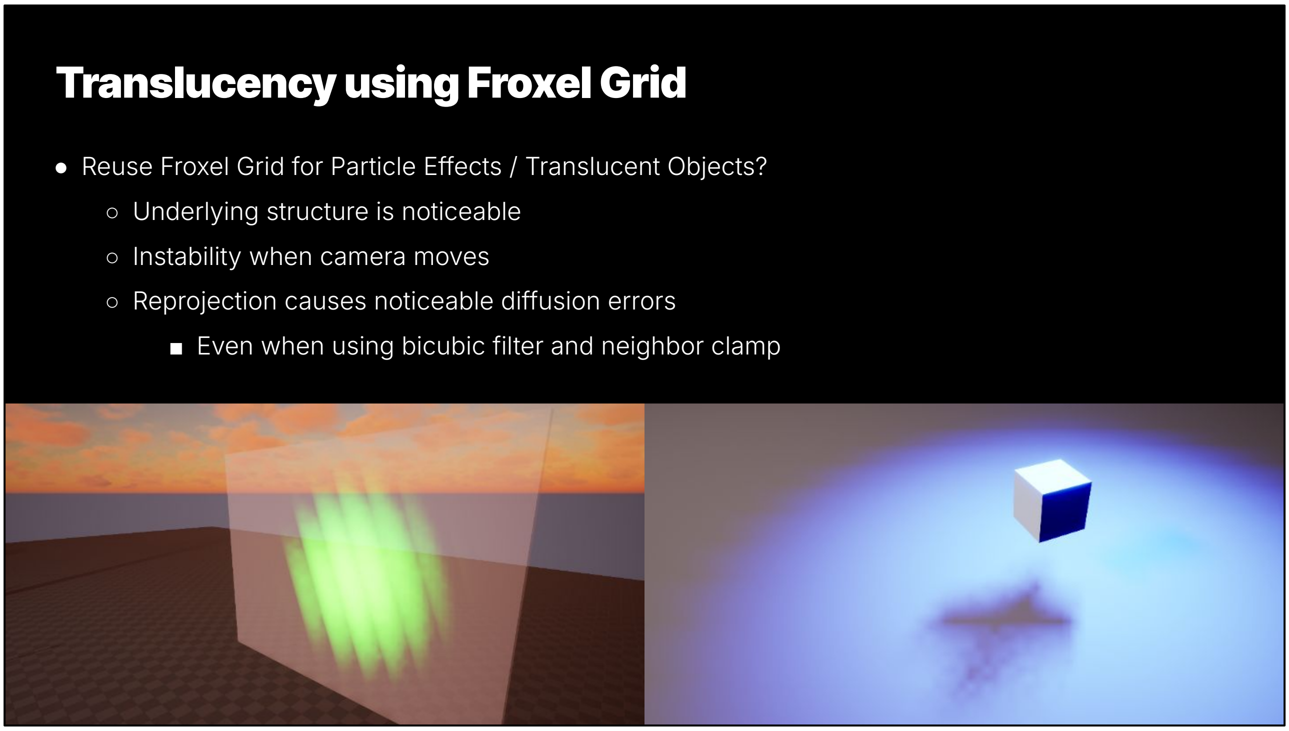


## Volumetric Grid Issues

- Sparse probe coverage
  - Neighbor visibility is poorly correlated
  - Ineffective guiding in high frequency regions
- A lot of duplicate work between the 2 volumes

There are however a few issues with this straightforward approach:

- First of all, probes are quite sparse, especially for the world space grid, so gathering visibility from nearby voxels leads to ineffective guiding in areas with high frequency shadowing
- And secondly, there's a lot of duplicate work between the 2 volumes
  - We are picking important lights and tracing shadow rays from a bunch of probes in the world
  - But as you can see in these screenshots, the coverage of the probes of those two volumes overlap significantly.

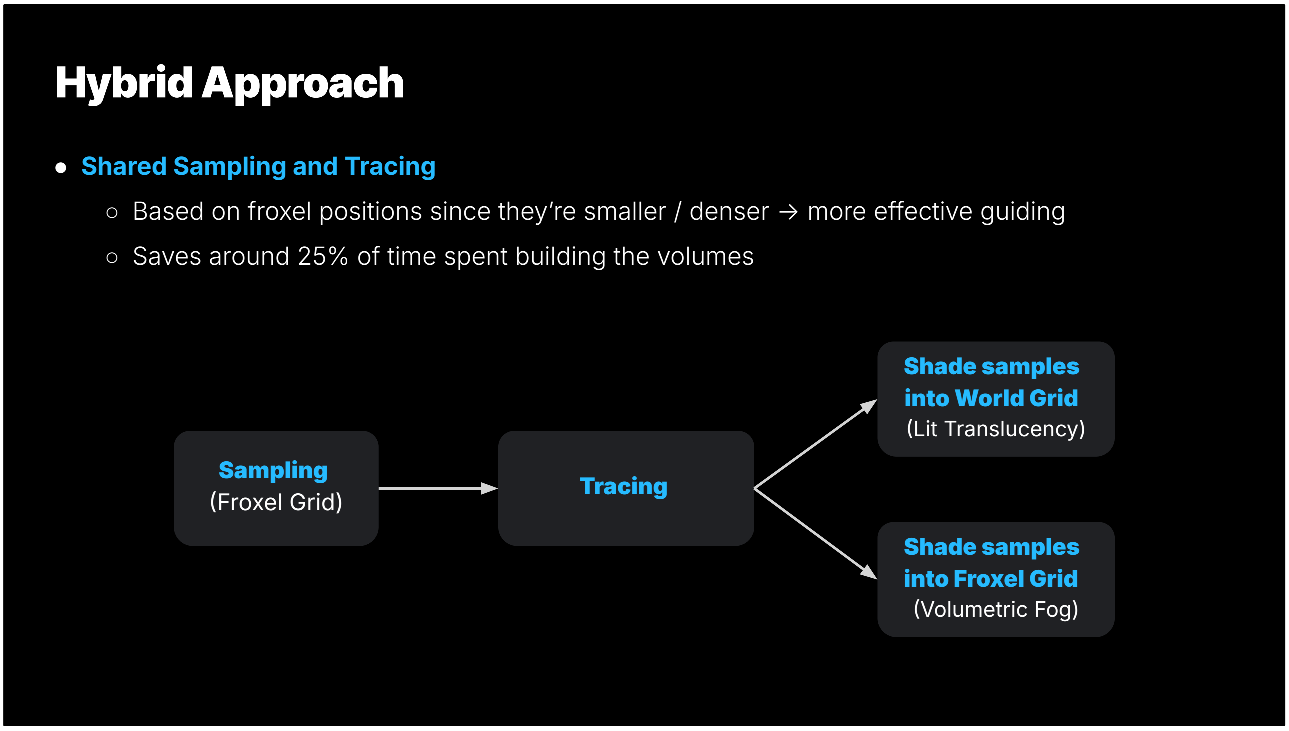


## Translucency using Froxel Grid

- Reuse Froxel Grid for Particle Effects / Translucent Objects?
  - Underlying structure is noticeable
  - Instability when camera moves
  - Reprojection causes noticeable diffusion errors
    - Even when using bicubic filter and neighbor clamp

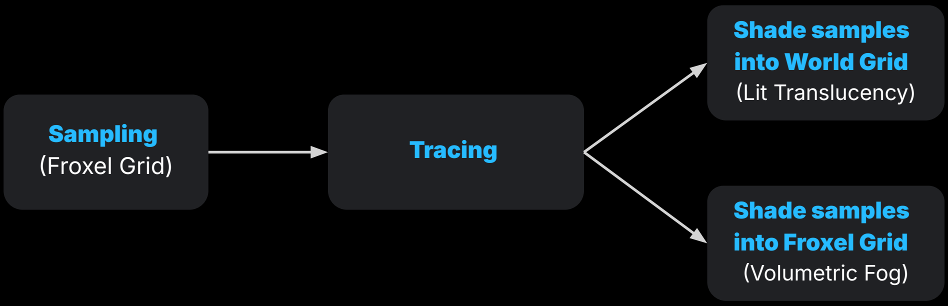
One idea is to simply extend the froxel grid structure to also store SH and use that to calculate lighting on particle effects and translucent objects. This seems like an efficient way to avoid duplicate work between the 2 volumes and improve guiding effectiveness on translucency since the froxels are denser. However, in practice this causes a few noticeable visual artifacts:

- First of all, the underlying structure of the froxel grid is quite noticeable, specially when the camera moves.
  - This sort of issue can also happen with world space grid but at least it's stable since it's not camera dependent, so it is not as distracting.
- We also need to use temporal filtering, but reprojection causes noticeable errors that we can't completely avoid.



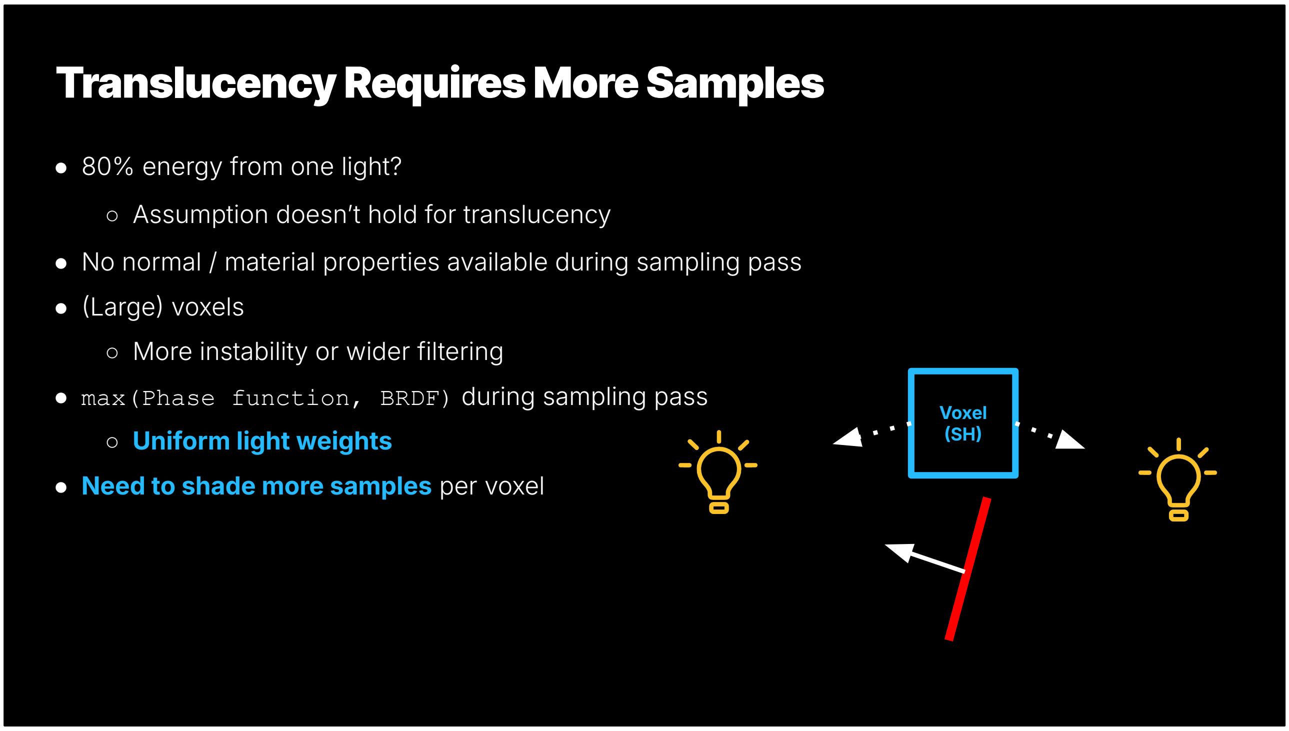
## Hybrid Approach

- **Shared Sampling and Tracing**
  - Based on froxel positions since they're smaller / denser → more effective guiding
  - Saves around 25% of time spent building the volumes



Another option is to use a hybrid solution, where only sampling and tracing are shared between the two volumes.
We run the sampling logic for probes based on froxel positions since they're denser, which means we can share visibility data between neighbor probes without hurting guiding effectiveness.
Then trace rays for those samples, and finally run 2 shading passes one for each volume, where each volume stochastically gathers samples from the closest sampling probes.

This approach saves around 25% of the time spent building the volumes since we don't need to run sampling and tracing twice.



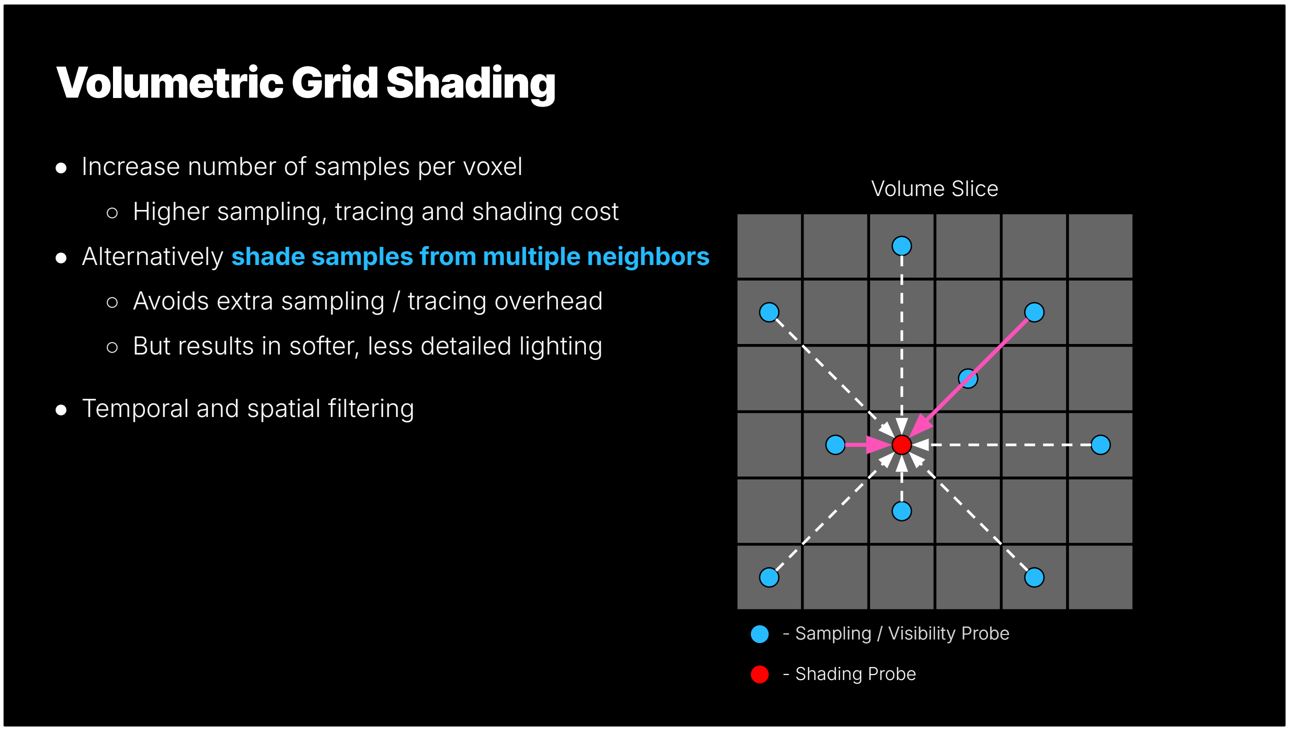
## Translucency Requires More Samples

- 80% energy from one light?
  - Assumption doesn't hold for translucency
- No normal / material properties available during sampling pass
- (Large) voxels
  - More instability or wider filtering
- `max(Phase function, BRDF)` during sampling pass
  - **Uniform light weights**
- **Need to shade more samples** per voxel

Earlier when talking about the opaque pipeline we mentioned the assumption that typically 80% of the energy comes from one light.
While for opaque surfaces we have access to the normal and material properties used for shading during sampling so we can get away with few samples per pixel, selecting samples for the translucency volume is more challenging.

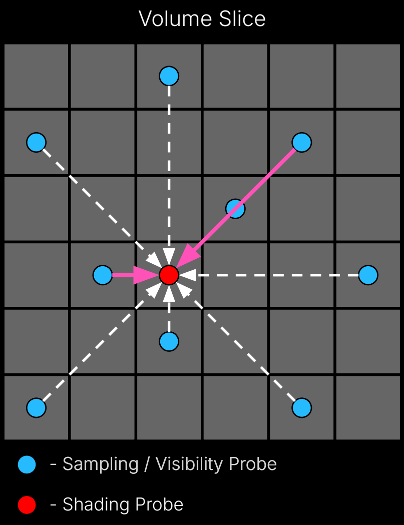
First of all, we are injecting the lights into spherical harmonics, and don't know which surfaces will be shaded using each voxel ahead of time.
The froxels are quite large relative to pixel size, so stochastically upsampling from neighbouring voxels can lead to instability or require wider filtering.
Since sampling is done in a unified volume we have consider both the phase function and the translucent BRDF which causes light weights to be more uniform.

The combination of these factors means we end up having to shade more samples per voxel than are usually required for opaque surfaces.



## Volumetric Grid Shading

- Increase number of samples per voxel
  - Higher sampling, tracing and shading cost
- Alternatively **shade samples from multiple neighbors**
  - Avoids extra sampling / tracing overhead
  - But results in softer, less detailed lighting
- Temporal and spatial filtering



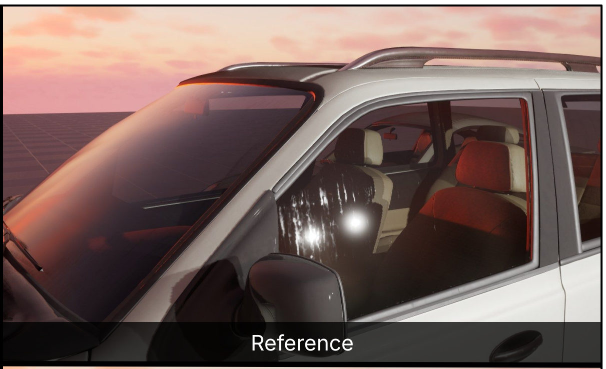
One easy way to do this is to increase the number of samples per voxel, however this makes all stages of the pipeline more expensive.

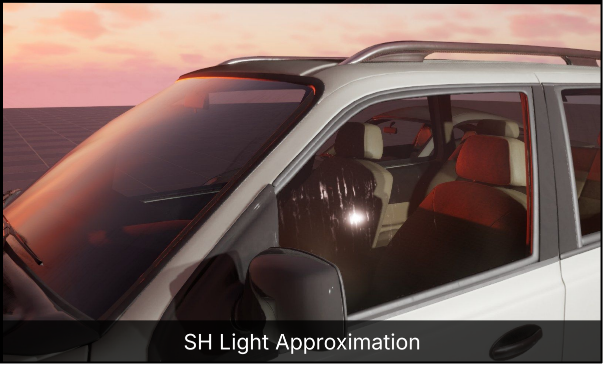
Alternatively, we can extend the stochastic upsampling we already do during shading, but instead of shading samples from a single neighbor, we can gather and shade samples from multiple neighbors. This avoids extra sampling and tracing overhead, but also results in some loss of lighting detail since it ends up blurring the visibility term.

Finally we also apply temporal and spatial filtering to further improve stability in the volumes.

## Forward Shading

- Some translucency requires higher quality lighting including **specular**
- Front Layer Translucency
  - Render translucency GBuffer
  - Run whole MegaLights pipeline again
  - Expensive and can only handle one layer
- Need lower frequency
  - Extract light from SH to approximate specular
  - Ongoing R&D



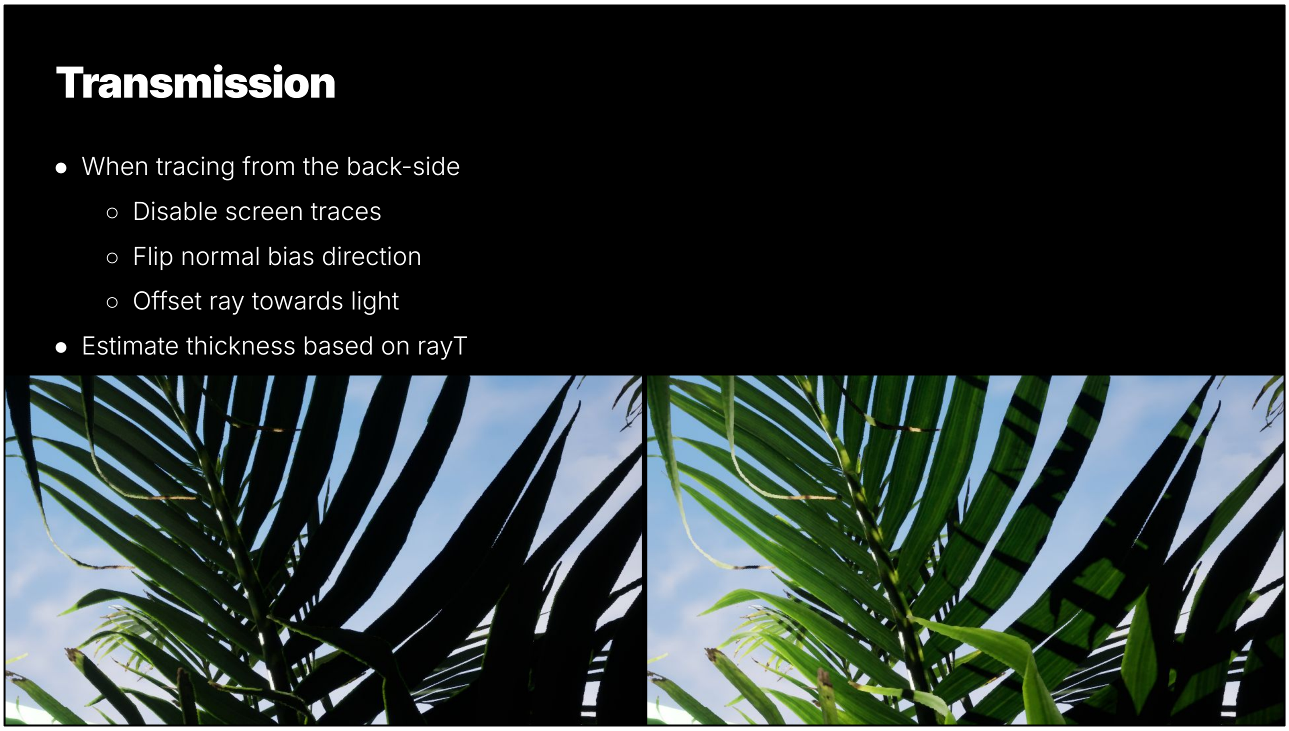


Some translucency such as glass requires higher quality lighting including specular.

A simple approach is to consider the front layer to be the most important and shade it by running the whole MegaLights pipeline again so it gets accurate lighting. However this is usually too expensive in practice since we can easily end up doubling the overall cost of lighting if the player gets close to those surfaces.

But we also need to support shading multiple layers so we need to handle this in a more scalable way. One cheap option is to light these surfaces using the translucency volume.  We can use the spherical harmonics that we already have, and extract the dominant direction to approximate specular lighting.  This can work well in simple cases, but as you can see in the screenshots, it can only support one specular highlight, and requires lighting in SH to be stable, otherwise the specular can noticeably jitter even when the lights are static.

For now we are using this method, but this is an area of ongoing work since we want to support better lighting quality on this type of translucent surfaces.



## Transmission

- When tracing from the back-side
  - Disable screen traces
  - Flip normal bias direction
  - Offset ray towards light
- Estimate thickness based on rayT

We also handle transmission, but only for thin surfaces at the moment, which greatly improves foliage lighting.

Transmission requires sometimes tracing from the back-side of the mesh and we have to make a few adjustments in that case. We disable screen space traces, as those can’t trace behind geometry. We also need to change our biasing and bias a bit towards the light. This way ray can skip behind the foliage geometry instead of immediately self-intersecting with it.

Distance from the back-side of the foliage to ray hit point gives as a distance estimate, which can be then plugged into our transmission term.

## Performance

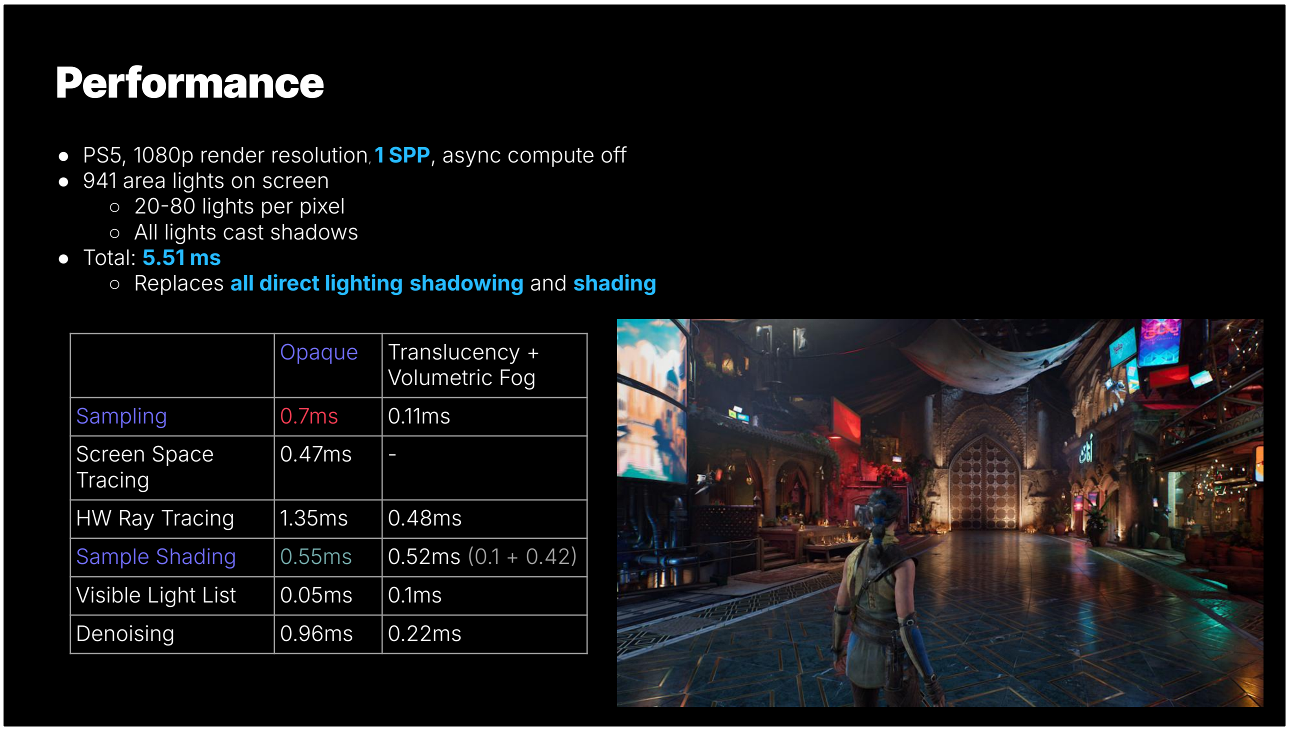
- PS5, 1080p render resolution, **1 SPP**, async compute off
- 941 area lights on screen
  - 20-80 lights per pixel
  - All lights cast shadows
- Total: **5.51 ms**
  - Replaces **all direct lighting shadowing** and **shading**

| | Opaque | Translucency + Volumetric Fog |
|---|---|---|
| Sampling | 0.7ms | 0.11ms |
| Screen Space Tracing | 0.47ms | - |
| HW Ray Tracing | 1.35ms | 0.48ms |
| Sample Shading | 0.55ms | 0.52ms (0.1 + 0.42) |
| Visible Light List | 0.05ms | 0.1ms |
| Denoising | 0.96ms | 0.22ms |

Let's now take a look at how this all comes together in terms of runtime performance.

These numbers are from the MegaLights demo running on a PlayStation 5 at 1080p render resolution, with 1 sample per pixel, and async compute disabled.
There are over 900 lights on screen, which results in around 20 to 80 lights per pixel.

The total cost is about 5.5ms covering all shadowing and shading cost for direct lighting.



## Performance

- PS5, 1080p render resolution, **1 SPP**, async compute off
- 941 area lights on screen
    - 20-80 lights per pixel
    - All lights cast shadows
- Total: **5.51 ms**
    - Replaces **all direct lighting shadowing** and **shading**

| | Opaque | Translucency + Volumetric Fog |
|---|---|---|
| Sampling | 0.7ms | 0.11ms |
| Screen Space Tracing | 0.47ms | - |
| HW Ray Tracing | 1.35ms | 0.48ms |
| Sample Shading | 0.55ms | 0.52ms (0.1 + 0.42) |
| Visible Light List | 0.05ms | 0.1ms |
| Denoising | 0.96ms | 0.22ms |

One interesting thing to note is that sampling for opaque surfaces takes more time than shading, despite running at half resolution.
That’s because sampling cost scales with the number of lights, which is very high in this scene, and a lot of those lights are rect lights which are expensive to sample and push occupancy down.
On the other hand, shading has a more fixed cost since it has a hard upper bound on how many samples are shaded per pixel, making it relatively stable since the cost is only affected by the tile type.

## Performance

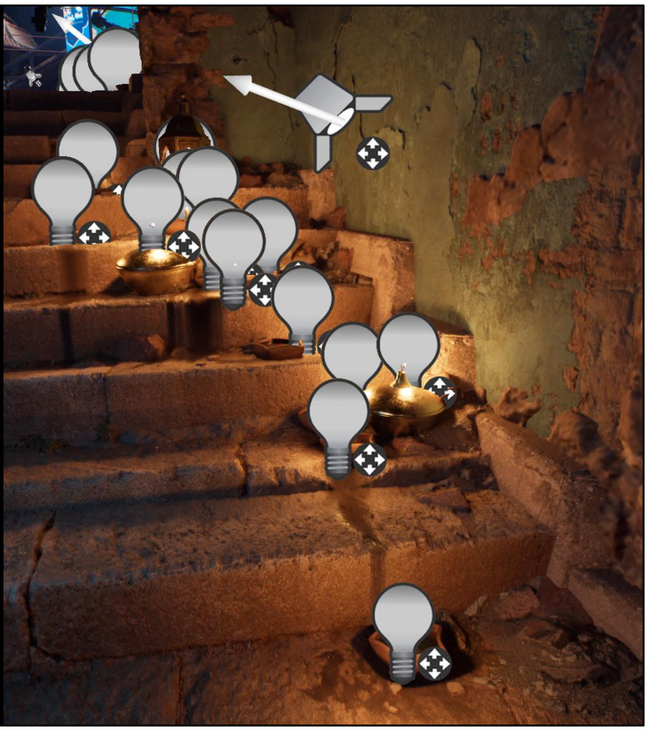
- PS5, 1080p render resolution, **1 SPP**, async compute off
- 941 area lights on screen
    - 20-80 lights per pixel
    - All lights cast shadows
- Total: **5.51 ms**
    - Replaces **all direct lighting shadowing** and **shading**

| | Opaque | Translucency + Volumetric Fog |
|---|---|---|
| Sampling | 0.7ms | 0.11ms |
| Screen Space Tracing | 0.47ms | - |
| HW Ray Tracing | 1.35ms | 0.48ms |
| Sample Shading | 0.55ms | 0.52ms (0.1 + 0.42) |
| Visible Light List | 0.05ms | 0.1ms |
| Denoising | 0.96ms | 0.22ms |

Another thing to notice is that translucency shading is almost as expensive as opaque, despite there being significantly fewer probes than there are pixels on the screen.
That's because, as we discussed earlier, translucency requires us to shade more samples per voxel, in order to improve stability and reduce noise.

## Practice

- Artists love it
  - Maybe even too much...

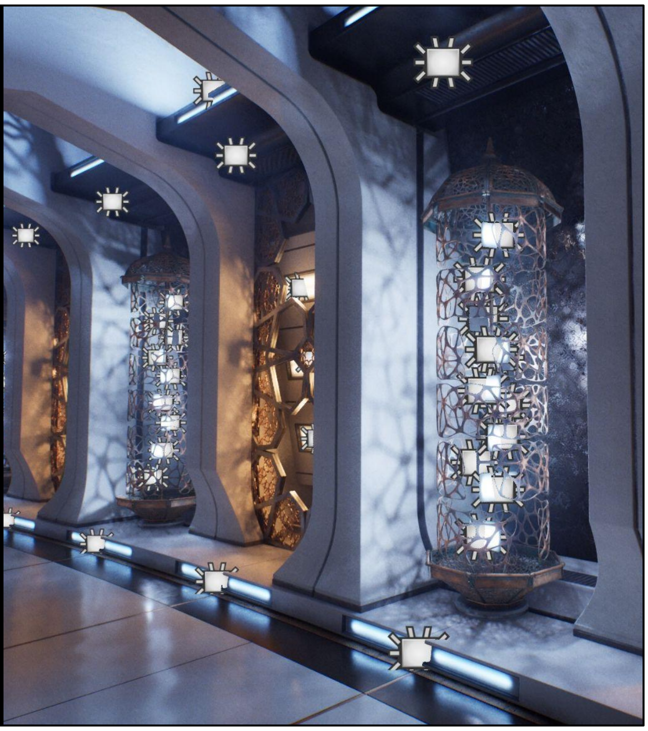


Overall, once artists start using with MegaLights, it quickly becomes clear how much fun they're having.

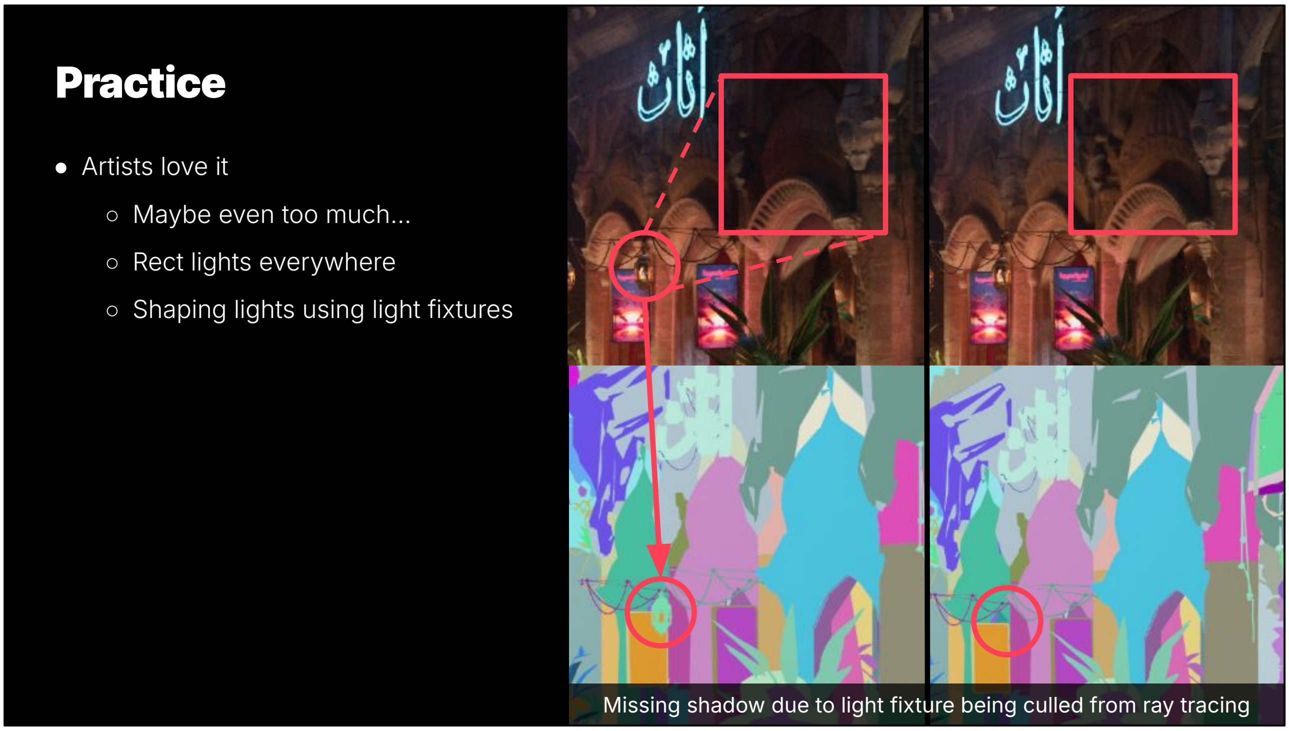
It was really interesting to see the progression as they got into a new mindset while working on the MegaLights demo. They started slowly, putting lights here and there, but that quickly changed and we started to see many more lights in the level.

## Practice

- Artists love it
  - Maybe even too much...
  - Rect lights everywhere



Then, as you can see in this screenshot, we also started to see usage of rect lights ramping up.



## Practice

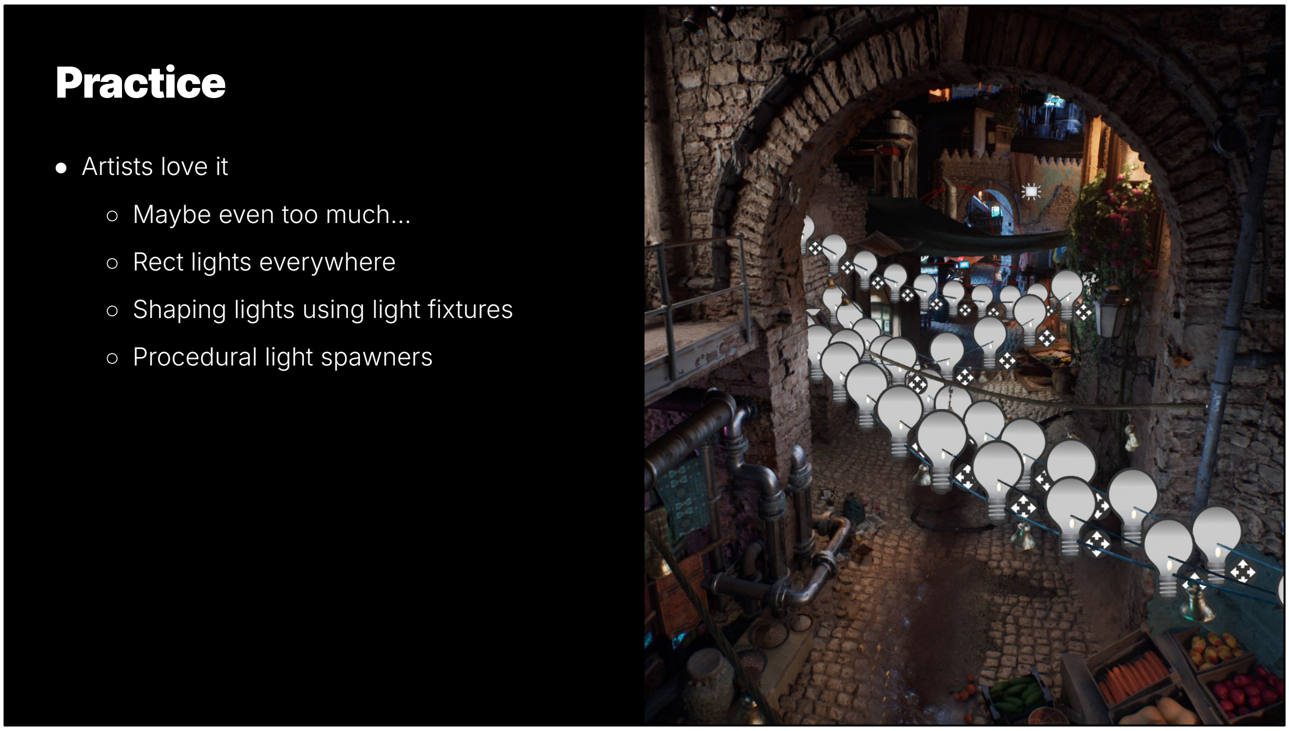
- Artists love it
  - Maybe even too much...
  - Rect lights everywhere
  - Shaping lights using light fixtures

Missing shadow due to light fixture being culled from ray tracing

We started to discover a lot of content where artists used meshes for light shaping, which makes it really hard to sample and resolve those lights correctly.

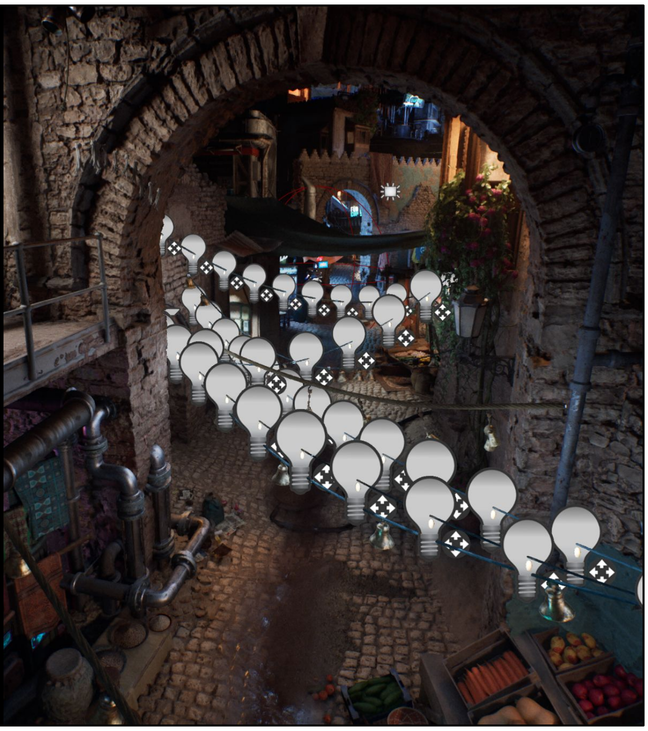
Ideally lights should be shaped using light functions (which can be cheaply evaluated during the sampling pass) and the fixture geometry skipped using a ray end bias specified per light.

Still in practice it wasn't such a big issue, so we shipped our demo with those mesh based fixtures. We just had to ask artists to manually tag them in the BVH, so that we don't end up automatically culling those in the distance, which would cause light leaking and significantly change the overall lighting.



## Practice

- Artists love it
  - Maybe even too much...
  - Rect lights everywhere
  - Shaping lights using light fixtures
  - Procedural light spawners



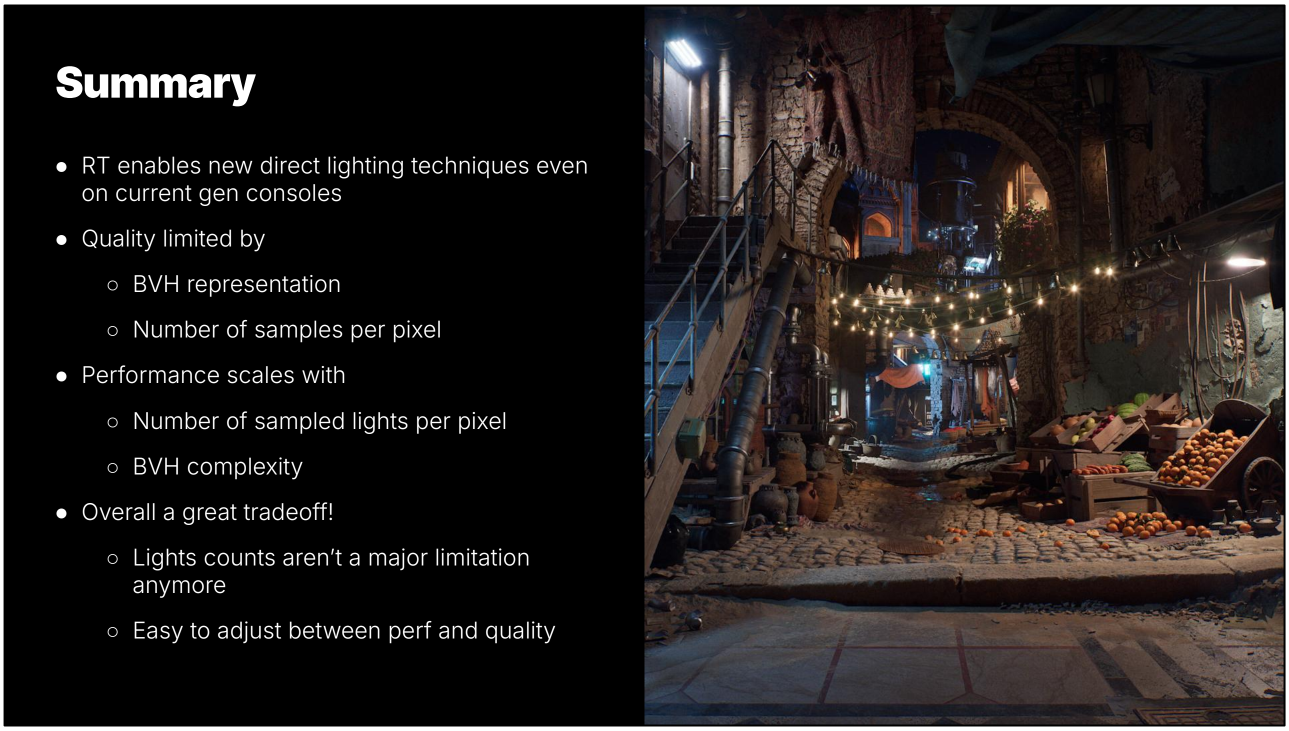
At some point artists got really comfortable, using tools like procedural light spawners to automatically place lights along splines.

## Content Optimizations

- Stochastic light sampling is much more forgiving
- Content optimizations are still relevant
  - Attenuation range
  - Spot light cone angles
  - Rect light barn doors
  - Light functions

The entire system scales much better and is more forgiving, often to the point where we were finding random textured area lights inside walls or lights made out of multiple light sources instead of using one light with an IES profile.

Still stochastic light sampling isn’t a magic bullet and light sources either reduce sampling pass performance or increase noise in the scene, and content optimizations like reducing attenuation range, cone angles or barn doors are relevant.



## Summary

- RT enables new direct lighting techniques even on current gen consoles
- Quality limited by
  - BVH representation
  - Number of samples per pixel
- Performance scales with
  - Number of sampled lights per pixel
  - BVH complexity
- Overall a great tradeoff!
  - Lights counts aren't a major limitation anymore
  - Easy to adjust between perf and quality

To wrap things up:

Ray tracing can enable new approaches to solving direct lighting, even on more resource constrained platforms like consoles.

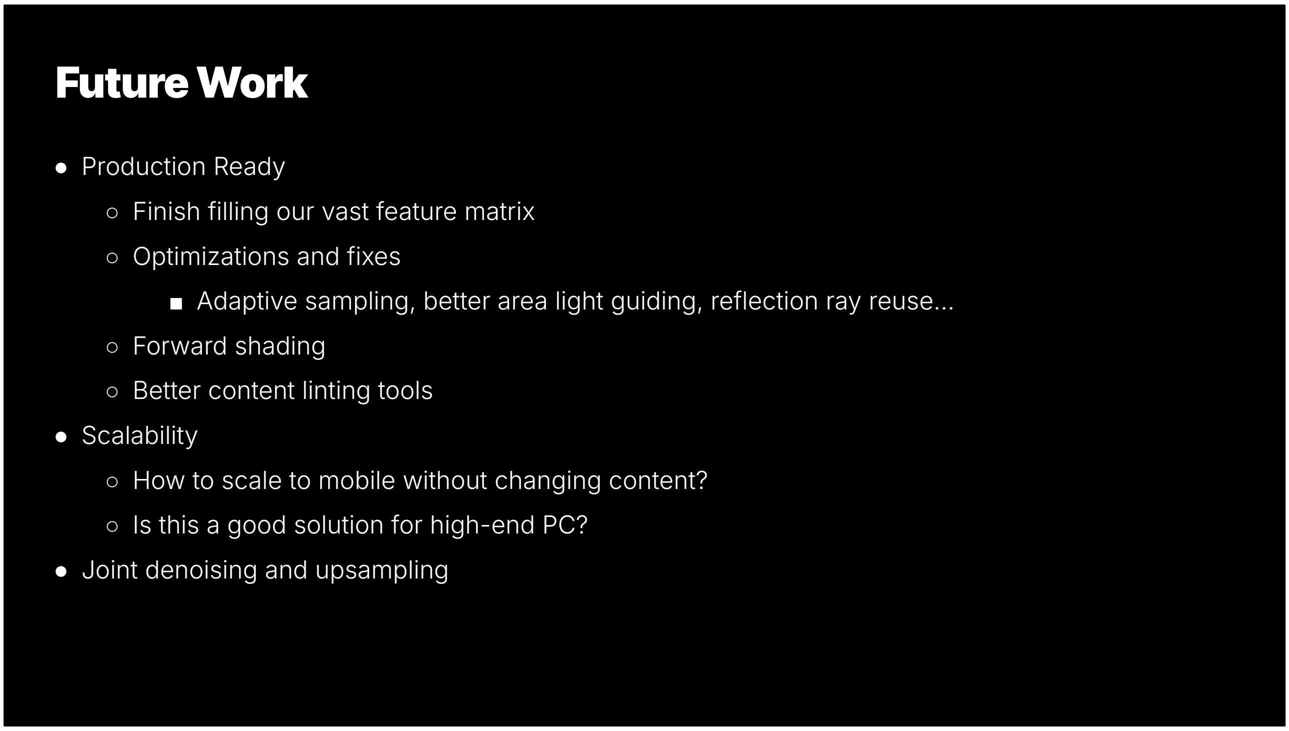
Of course, there are still quality limitations such as:
- How well meshes are represented in ray tracing
- And in most cases we are limited to 1 sample per pixel, so careful sampling and denoising are critical.

And the performance of this technique scales with number of lights sampled per pixel and ray tracing scene complexity.

We have scary limitations here and clearly light counts are still limited, but overall it turned out to be a pretty good tradeoff in practice. Artists can place lights much more freely before the system breaks. Additionally stochastic approaches excel at being able to freely adjust between performance and quality, which gives us more flexibility without having to resort to labor intensive manual light adjustments or maintaining multiple light setups for different scalability levels.

While BVH limitations are scary, in practice artists like to place area lights, which usually only require detailed shadow casters near the contact points. This first segment of the rays is reasonably well handled by screen space traces making it a reasonable solution for local lights. Directional light can be challenging, as it may require sharp shadows at large draw distances, which may not be practical with a BVH. In that case we can fallback to VSM. Either by relying exclusively on VSM or using the hybrid approach that combines both ray tracing and VSM tracing.

Temporal artifacts can be an issue during fast movement, as all stochastic light sampling methods depend on being able to accumulate and reuse data over multiple frames. Those aren't a prominent issue with small light counts with small light source sizes, as usually we can just sample one most important light per pixel and get almost instantaneous response. With large light sources or lots of important lights per pixel things do break down though somewhat, as now we have to depend on multiple frames of data and we don't have those. Still in practice for most games it seems to be working well enough and temporal artifacts are usually hidden by the denoiser. Those issues are also less problematic on higher-end GPUs where we can afford to trace more rays per pixel.



## Future Work

- Production Ready
  - Finish filling our vast feature matrix
  - Optimizations and fixes
    - Adaptive sampling, better area light guiding, reflection ray reuse...
  - Forward shading
  - Better content linting tools
- Scalability
  - How to scale to mobile without changing content?
  - Is this a good solution for high-end PC?
- Joint denoising and upsampling

In terms of future work, we are still working towards making MegaLights production ready. There's a bunch of optimizations, fixes and missing features that we still need to improve and implement.

For example, we want to be mixing BRDF sampling with explicit sampling. Explicit sampling is great for sampling lights, which are small or further away from the surface. BRDF sampling is great for large lights and for reflective BRDFs. We already have our BRDF rays (Lumen traces), so we can cheaply reuse those in order to improve direct lighting quality.

We are working on improvements to better support forward shading, using GPU driven feedback mechanisms to drive sample guiding based on the rendered forward shaded surfaces. We are also gathering feedback from licensees using this technology in production, which will allow us to prepare useful tooling for debugging and optimizing MegaLights.

Scalability to low-end (mobile) is still an open problem. Some game titles want to ship their content across a very wide range of hardware - from cheap mobile phones to high-end PC GPUs. Once content is authored using MegaLights it becomes challenging to reduce light count without requiring a lot of work for lighting artists, and we don't want to end up in a situation where artists have to maintain multiple lighting setups manually.

Another important area is denoising and upsampling. At the moment MegaLights runs it’s own denoising and the output then ends up being further processed by TSR, which results in various challenges. We want to look into making making those two more unified. Either by passing some extra information from the internal denoiser or by moving all denoising to TSR. There’s certainly a ton of new research using ML that shows very promising results in this area.

## Future Challenges - BVH

- BVH is the main limiting factor
  - Memory
  - BVH build times
  - Traversal performance
  - Manual optimization
- Geometry complexity is **constantly growing**!
  - Hair splines
  - Nanite Tessellation
  - Nanite Foliage

The main challenge for the future is mesh representation in hardware ray tracing.

While until now we have (sort of) been able to get away with low poly proxies and other shortcuts, since diffuse GI and rough reflections are quite forgiving. Shadows for direct lighting can easily expose the underlying ray tracing representation of meshes.

Dynamic geometry is still very much an unsolved problem and it can easily blow memory and performance budgets, making it currently impossible to accurately represent large numbers of animated instances.

Large amount of kitbashing or huge meshes with lots of empty area inside (like skybox mesh or caves modelled from a single mesh) can cause ray tracing performance issues even on the high-end PC GPUs. In theory this could be addressed by BVH rebraiding, but rebraiding is a pretty costly operation and requires much more memory, which isn't ideal as already BVH build times are challenging and BVH representation itself has a large memory overhead.

With Nanite (our virtualized geometry pipeline) we were able to hide most of the mesh optimization problems for rasterization, but ray tracing constraints bring those back requiring artists to carefully optimize their BVH representations.

While we are starting to see new extensions to ray tracing APIs that will enable more efficient representation of Nanite meshes, geometry complexity is constantly growing. There are new types of content that are even more challenging to represent in ray tracing, which will need to be addressed before we will be able to rely on pure ray tracing for shadowing.

## Acknowledgements

### Talk Review
Natalya Tatarchuk, Sébastien Lagarde, Brian Karis, Daniel Wright, Chris Kulla, Andrew Lauritzen, Charles de Rousiers, Aleksander Netzel, Kenzo ter Elst

### Thank You!
Daniel Wright, Charles de Rousiers, Jian Ru, Andrew Lauritzen, Dustin Hulm,
The UE5 Rendering Team

### MegaLights Demo Team
Wyeth Johnson, Daniel Woje, Isac Crafoord, Saga Alayyoubi, Carlos Cristerna, Nathaniel Morgan, Wiktor Öhman, Jakob Keudel

# References

## Sampling

- [Yuksel 2019] - “Stochastic Lightcuts”, HPG 2019
- [Efraimidis et al. 2006] - “Weighted random sampling with a reservoir”, Information Processing Letters, 2006
- [Donnely et al. 2024] - “Filter-Adapted Spatio-Temporal Sampling for Real-Time Rendering”, i3D 2024
- [Ogaki 2021] - “Vectorized Reservoir Sampling”, SIGGRAPH Asia 2021
- [Georgiev et al. 2016] - “Blue-noise Dithered Sampling”, SIGGRAPH 2016
- [Clarberg et al. 2016] - “Wavelet importance sampling: efficiently evaluating products of complex functions”, SIGGRAPH 2005

## ReSTIR

- [Wyman et al. 2023] - “A Gentle Introduction to ReSTIR: Path Reuse in Real-time”, SIGGRAPH 2023
- [Kozlowski et al. 2023] - “ReSTIR Integration in Cyberpunk 2077”, SIGGRAPH 2023
- [Wyman et al. 2021] - “Rearchitecting spatiotemporal resampling for production”, HPG 2021
- [Panteleev et al. 2020] - “Rendering Games With Millions of Ray-Traced Lights”, GTC 2020
- [Bitterli 2022]- “Correlations and Reuse for Fast and Accurate Physically Based Light Transport”, Thesis 2022
- [Knapik el al. 2024] - “The Evolution of the Real-Time Lighting Pipeline in Cyberpunk 2077”, GPU Zen 3, 2024

## Ray Tracing

- [Netzel et al. 2022] - “Ray Tracing Open Worlds in Unreal Engine 5”, SIGGRAPH 2022
- [Stachowiak 2024] - “Rendering Tiny Glades With Entirely Too Much Ray Marching”, GPC 2024

## Shadow Maps

- [VSMRT] - “Soft Shadows with Shadow Map Ray Tracing”, UE documentation

## References

### Global Illumination

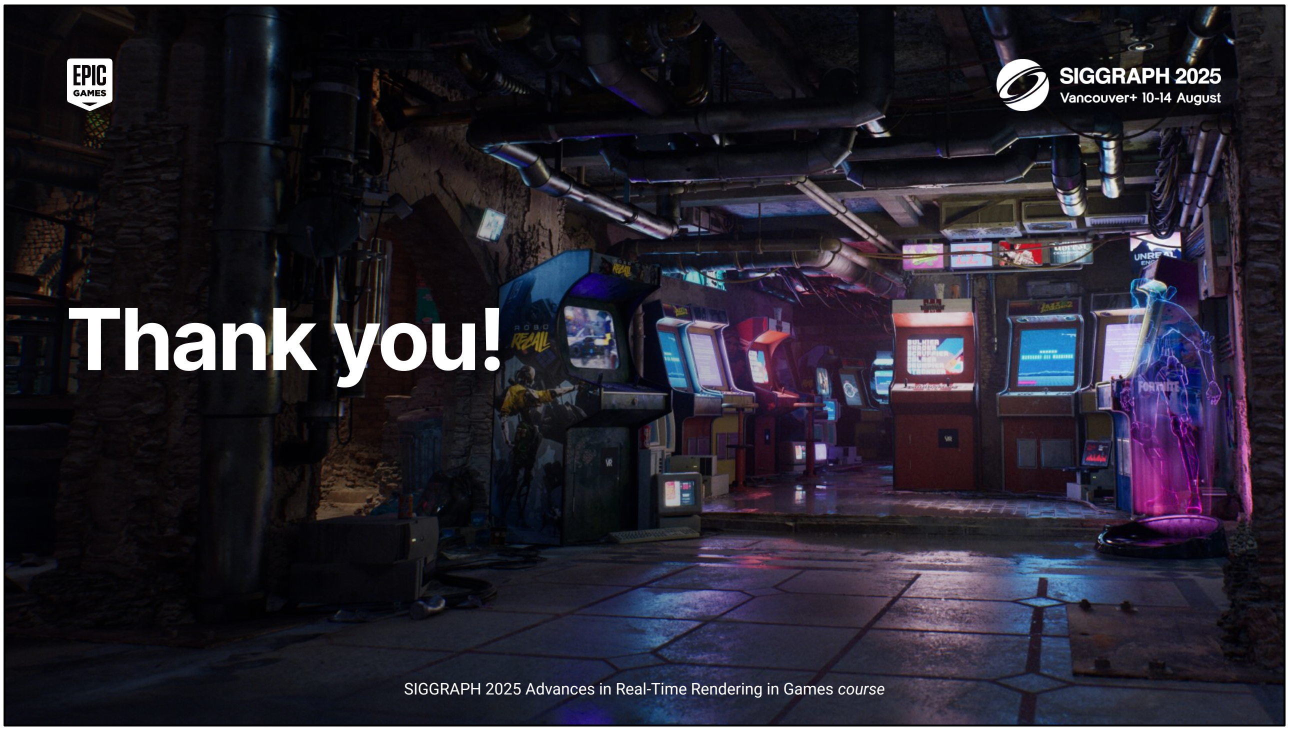
- [Wright et al. 2022] - “Lumen: Real-time Global Illumination in Unreal Engine 5”, SIGGRAPH 2022

### Light Lists

- [Sousa et al. 2016] - “The devil is in the details: idTech 666”, SIGGRAPH, 2016

### Denoising

- [Heitz et al. 2018] - “Combining Analytic Direct Illumination and Stochastic Shadows”, i3D 2018
- [Schied et al. 2016] - “Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination”, HPG 2017
- [T. Zhuang et al. 2021] - “Real-time Denoising Using BRDF Pre-integration Factorization”, CGF 2021
- [Karis 2013] - “Tone Mapping”, Blog 2013
- [Salvi 2016] - “An excursion in temporal supersampling”, GDC 2016
- [Karis 2014] - “High quality temporal supersampling”, SIGGRAPH 2014



# Thank you!

SIGGRAPH 2025 Vancouver+ 10-14 August