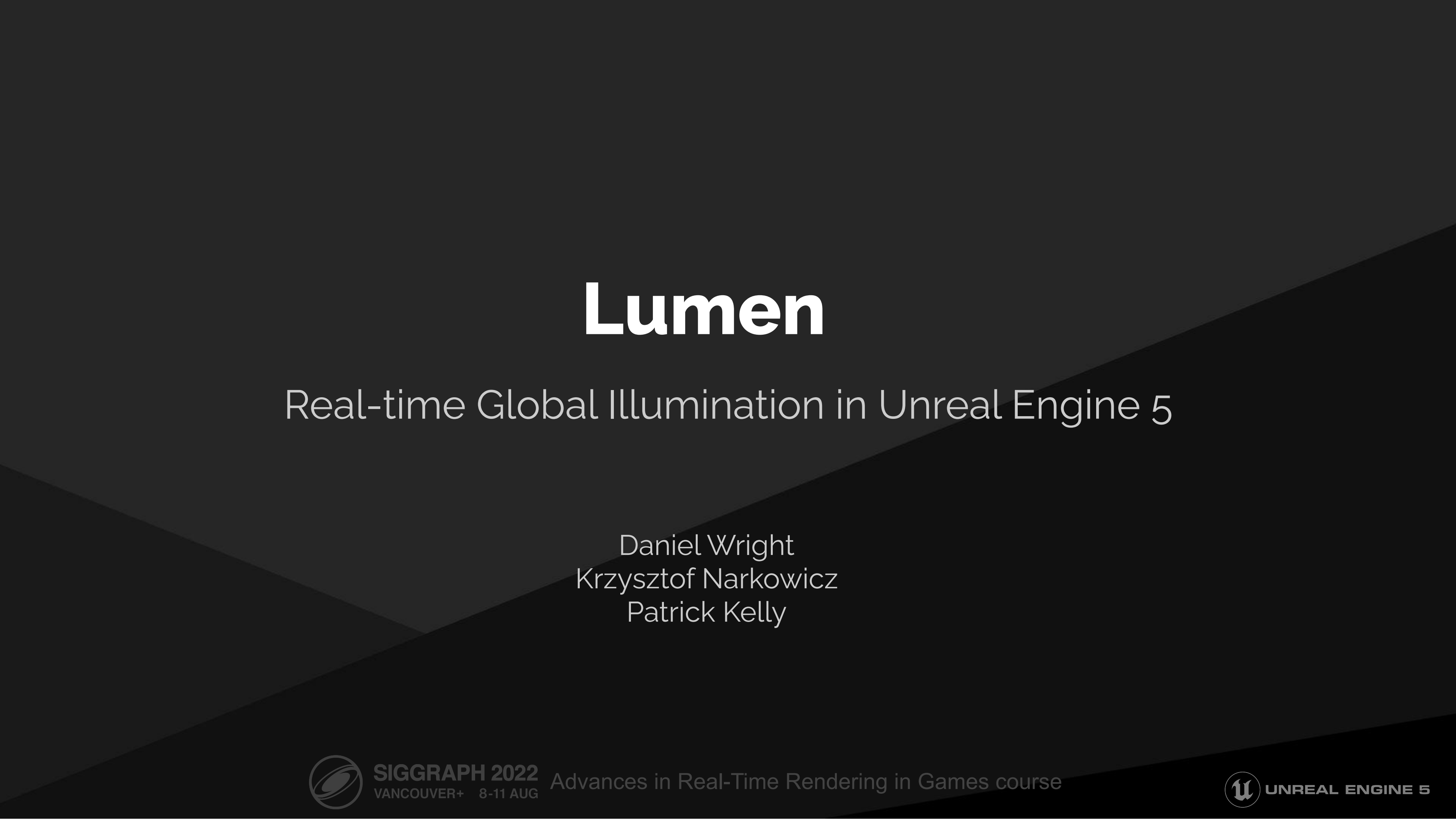


# Lumen

## Real-time Global Illumination in Unreal Engine 5

Daniel Wright
Krzysztof Narkowicz
Patrick Kelly

SIGGRAPH 2022
VANCOUVER+ 8-11 AUG
Advances in Real-Time Rendering in Games course

UNREAL ENGINE 5

Hi, I’m Daniel Wright, and today with my co-workers Krzysztof Narkowicz and Patrick Kelly, we’re presenting Lumen, our real-time Global Illumination system in Unreal Engine 5.

# The Dream - dynamic indirect lighting

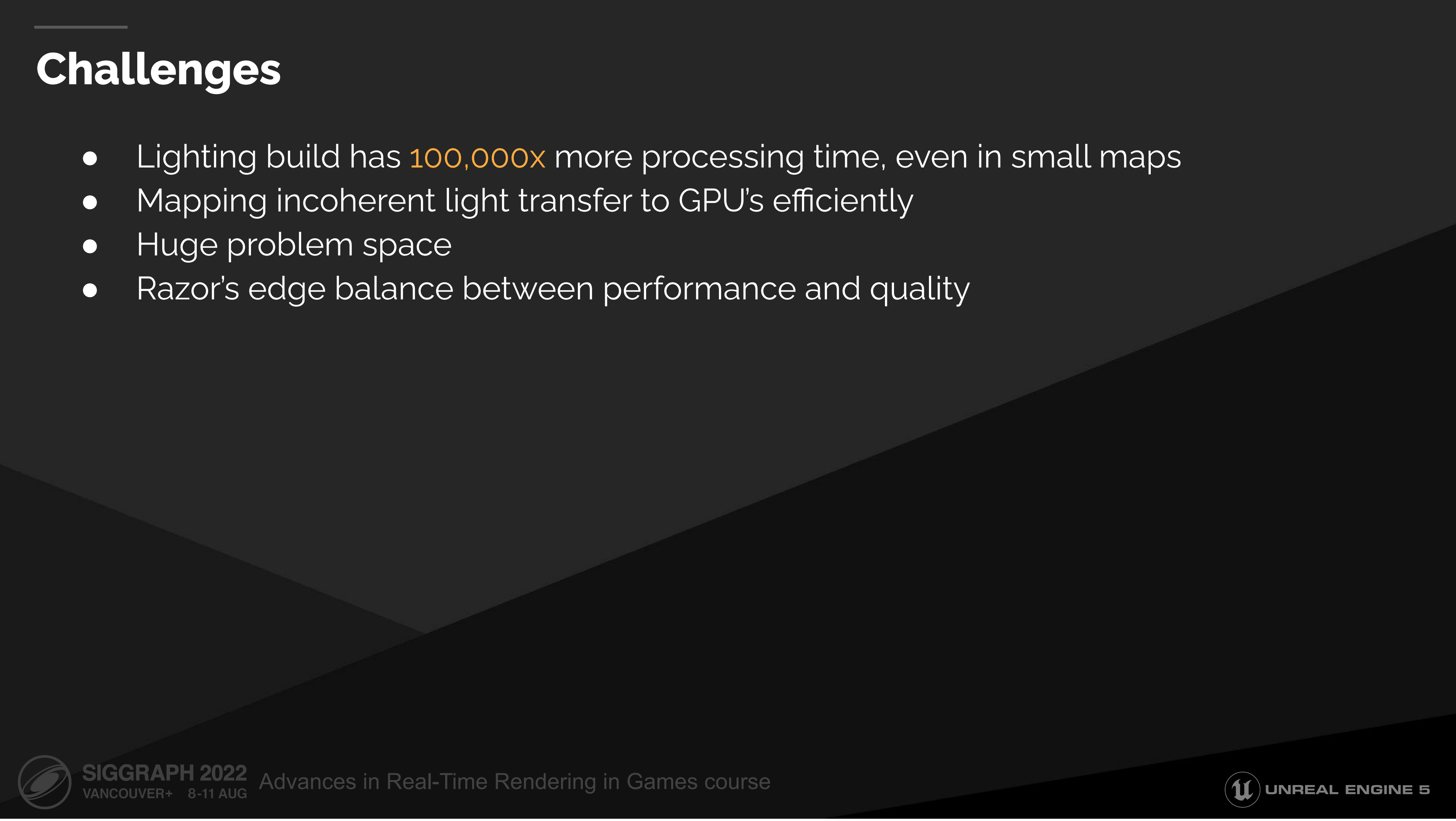
- Unlock new ways for players to interact with game worlds
- Instant results for lighting artists
  - No more lighting builds
- Huge open worlds that couldn't have ever been baked
- Indoor quality comparable to baked lighting

We've always dreamed of having fully dynamic indirect lighting. This would unlock new ways for players to interact with game worlds. Baked lighting constrains even simple interactions like opening a door or destroying a wall. If those simple interactions would be solved with dynamic indirect lighting, what kind of new complex interactions would we see?

We also wanted a much better workflow for lighting artists. Instead of waiting around for minutes or even hours for the lighting build to finish and to see the results of their work, we would like them to see their results immediately, and how much better would the quality of their lighting be if they could see the results in realtime?

We also wanted huge open worlds that couldn't have ever been baked, and to solve all of the problems that come from using baked lighting in a big production, where hundreds of people are changing the level every day and the baked lighting is never up-to-date.

It's not enough to solve dynamic indirect lighting with quality that works for outdoors, we want to solve it with quality comparable to baked lighting, so we get all the lighting details and indirect shadows that baked lighting can do.



## Challenges

- Lighting build has 100,000x more processing time, even in small maps
- Mapping incoherent light transfer to GPU's efficiently
- Huge problem space
- Razor's edge balance between performance and quality

In terms of challenges, the lighting build that we're trying to match the quality of has a hundred thousand times more processing time than solving indirect lighting in realtime, even in small maps where there's not a big difference between what the two are operating on.

Global Illumination is fundamentally incoherent, and it's a challenge to solve that light transfer efficiently when GPU's are designed for memory and execution coherency.

It's also a huge problem space, there are so many different choices we could make, it's a challenge just to know which paths can work and which can't.

The margins for success with realtime global illumination are so small, that it's hard to satisfy both performance and quality at the same time. It's like standing on top of a ridge where a small movement in either direction causes a drop in performance or quality.

Now I'll do an overview of Lumen's algorithms at the highest level, and then we'll go through them in more detail later.

# Fundamental problem #1:
# **How to trace rays?**

- Hardware Ray Tracing is great, but
  - Need options to scale down
    - PC market, 60fps console
  - And handle heavily overlapping meshes that are slow with HWRT
- Also want a Software Ray Tracing path that addresses these

The first problem that we have to solve for realtime indirect lighting is: how are we going to actually trace rays through the world?

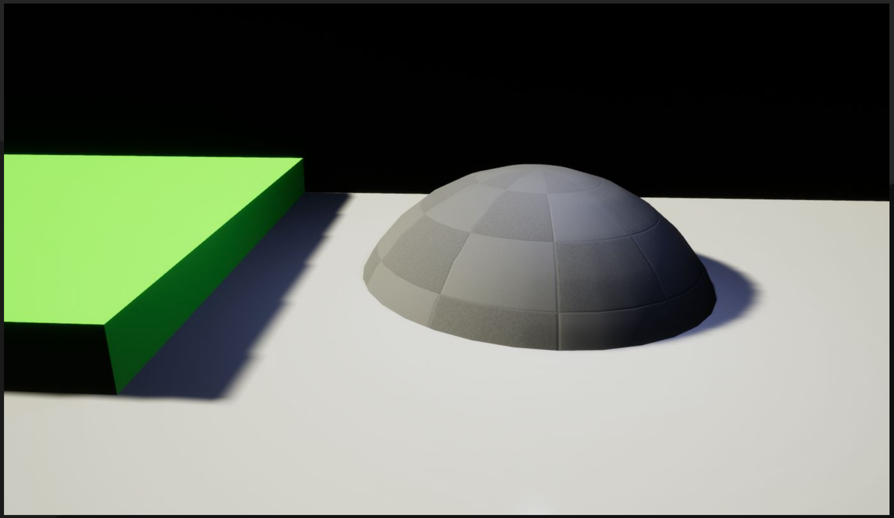
Hardware Ray Tracing is great and it is the future, but we need options to scale down. In the PC market there are still plenty of video cards that don’t support hardware ray tracing, and console Hardware Ray Tracing is not that fast.

We also want to handle scenes that have heavily overlapping meshes, which are slow with Hardware Ray Tracing’s two level acceleration structure.

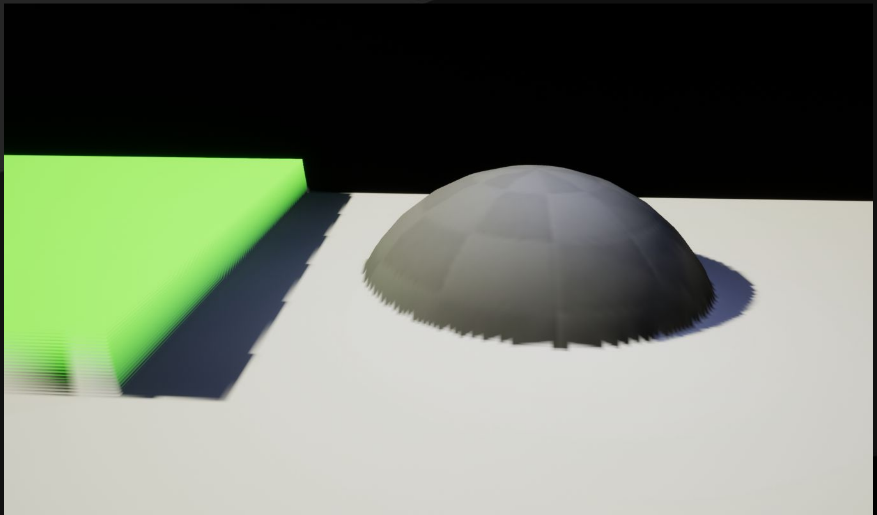
So we’re going to develop a Software Ray Tracing path that addresses these limitations.

# Early experiment: **Ray tracing the scene as heightfields (cards)**

- Construct by capturing the scene's surfaces with 2d ortho cameras
  - Similar to Rasterized Bounding Volume Hierarchies [Novak 2012]
- High spatial resolution compared to voxels
- Fast tracing leveraging heightfield properties - POM [Tatarchuk 2006]
- But, unreliable coverage leads to leaking



Triangles



Heightfields

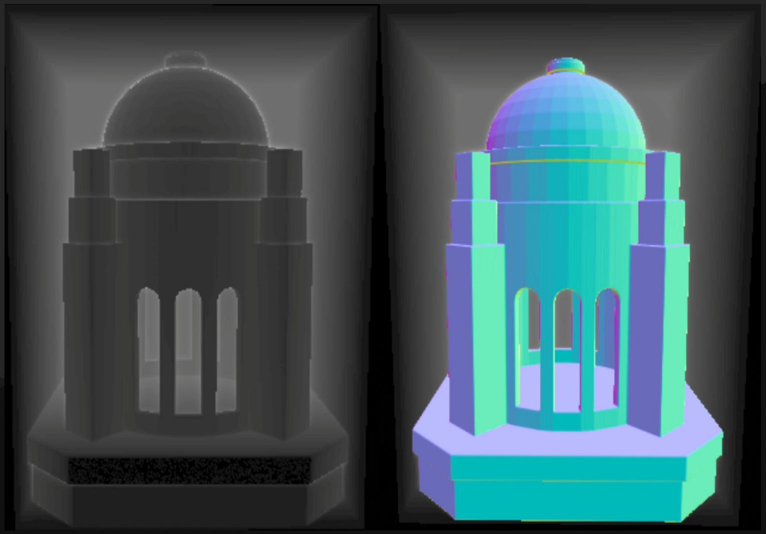
When we started working on Software Ray Tracing, one of the first things we tried was to capture the scene using a bunch of orthographic cameras, giving what we call cards.  We then ray trace through the card heightfields, and sample the card lighting when the ray hits.

Because this is a 2d surface representation, it gives high spatial resolution compared to 3d representations like voxels, and we get fast software tracing by leveraging the heightfield properties, just like Parallax Occlusion Mapping.

But ultimately it's impossible to cover the entire scene with heightfields, and areas that are missing coverage cause leaking.

# Mesh Signed Distance Field tracing [Wright 2015]

- Reliable occlusion unlike card heightfields
- Fast software tracing - sphere tracing
- Intersection gives hit position and normal
  - But no further attributes, don't know material or lighting



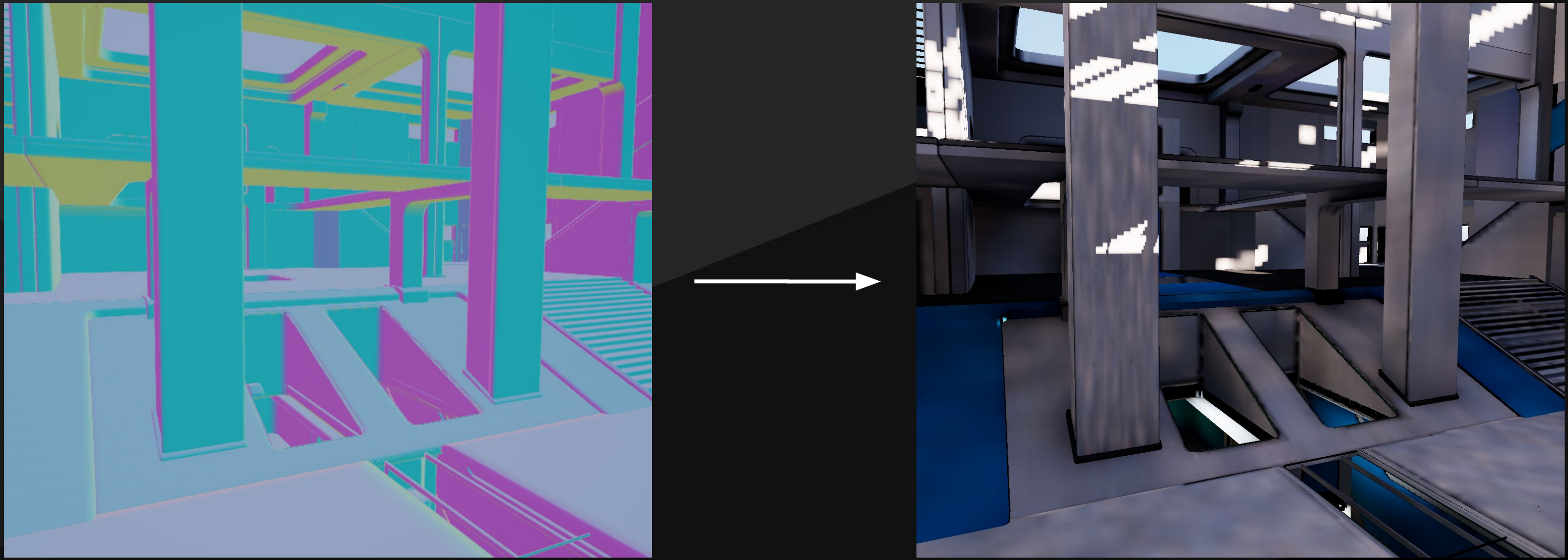
Hit position

Normal (from gradient)

So instead we settled on Mesh Signed Distance Fields for our Software Ray Tracing geometry representation.  These give reliable occlusion, all areas have coverage, and we still get fast software ray tracing through sphere tracing, which skips through empty space.
The intersection with the distance field surface only gives us the hit position and normal, we can't find the material attributes or the lighting.

# Lighting from cards (Surface Cache)

- Interpolate at SDF hit position + normal
- High resolution lighting cache
- Unreliable coverage results in lost energy, rather than leaking



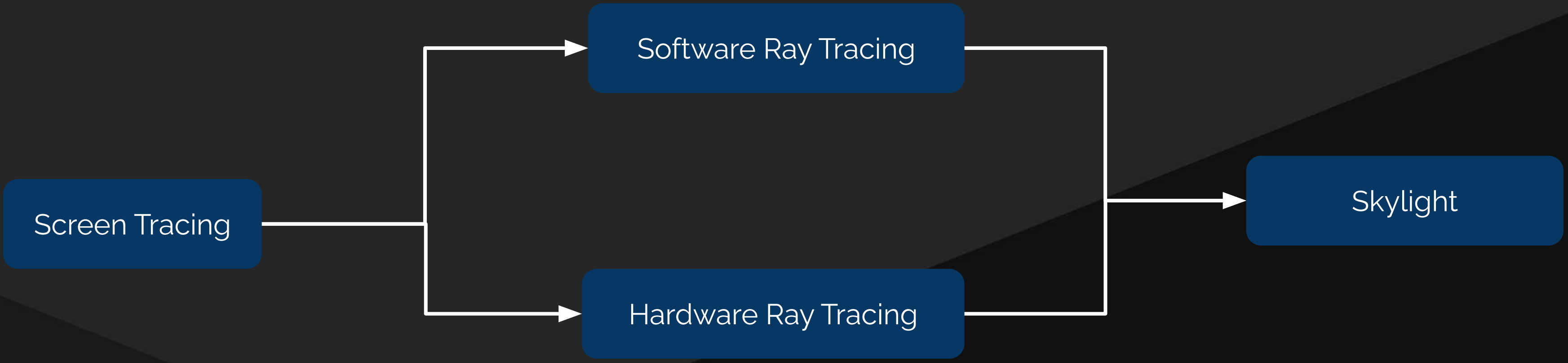
We interpolate the lighting where the ray trace hit from the cards, which we call the Surface Cache.  Areas that are missing coverage only result in lost energy, instead of leaking.  Ray tracing the card heightfields didn't work, but using them for lighting does.

# Surface Caching bonuses

- Shares material evaluation and lighting work between indirect rays
- Caches across frames, gives direct control over update cost
- Fast paths for Hardware Ray Tracing

Surface Caching has a lot of other benefits, it shares material evaluation between rays, gives us direct control over the update of the surface cache, and unlocks fast paths for Hardware Ray Tracing, which Patrick will cover in depth later.

# Ray Tracing pipeline



Screen Tracing

Software Ray Tracing

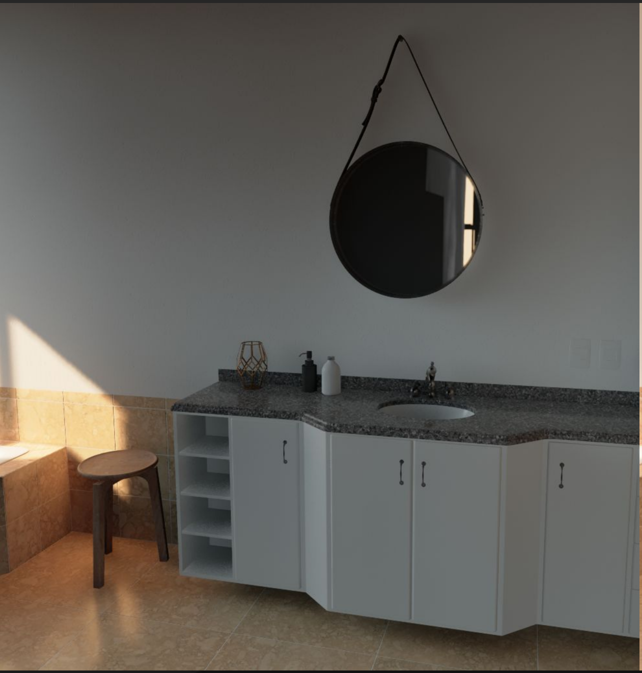
Hardware Ray Tracing

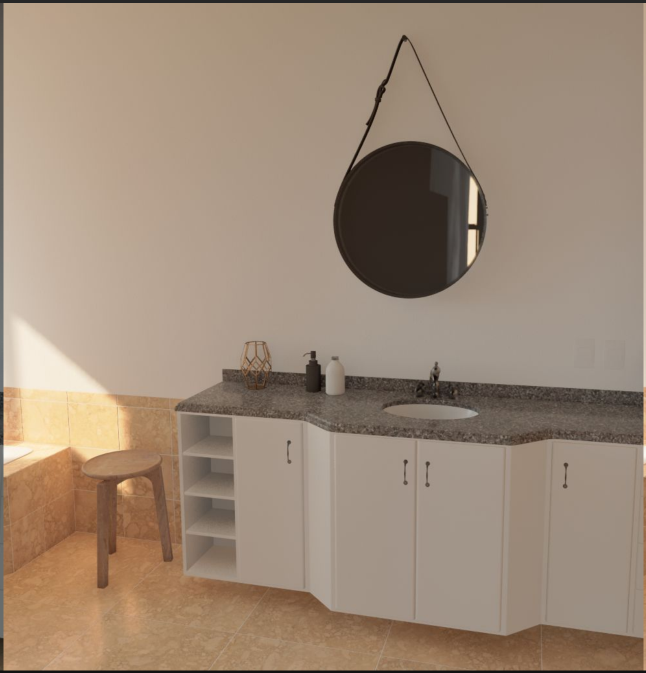
Skylight

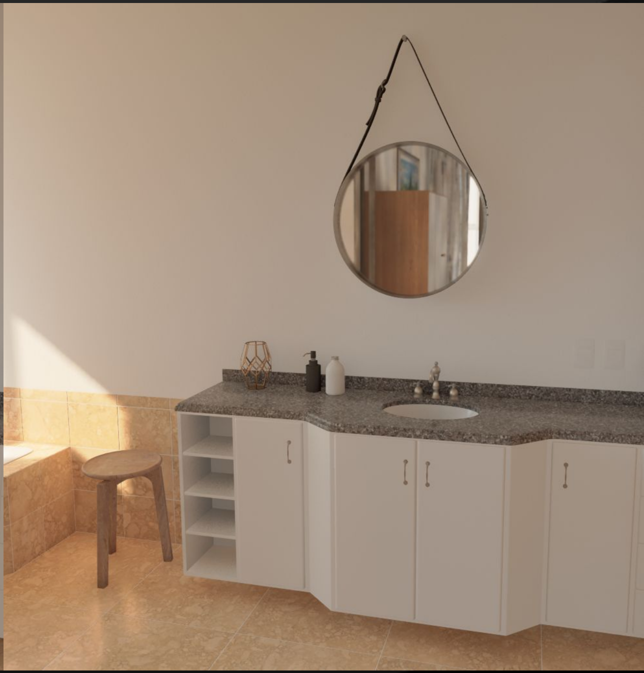
Here’s Lumen’s ray tracing pipeline, starting from Screen Tracing, which we run first, because it’s the most accurate near the start of the ray, then we do Software Ray Tracing or Hardware, depending on what is configured, and finally skylight if the ray misses any geometry.

Fundamental problem #2:

## How to solve the whole indirect lighting path?


Single bounce


Multi-bounce diffuse

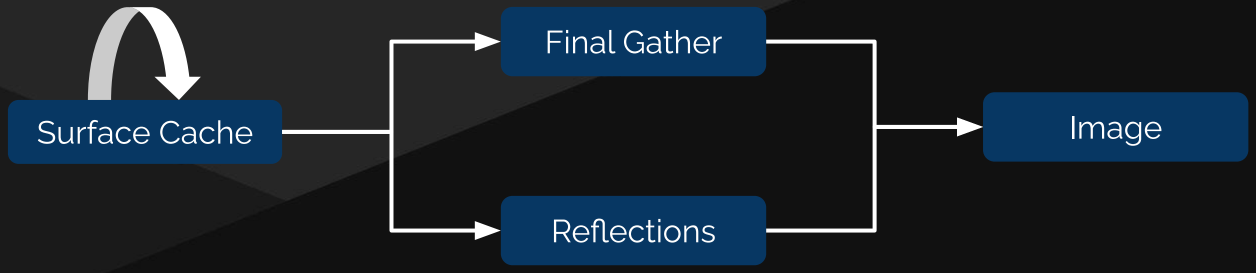

GI seen in reflections

The next fundamental problem in realtime global illumination is solving the whole indirect lighting path.

It’s not enough to just provide a single bounce of indirect lighting, we need multibounce diffuse for indoor scenes and we need global illumination seen in reflections.

# First bounce is the most important, needs careful sampling

- Final Gather (diffuse)
- Reflection denoising (specular)
- Multi-bounce diffuse through Surface Cache feedback

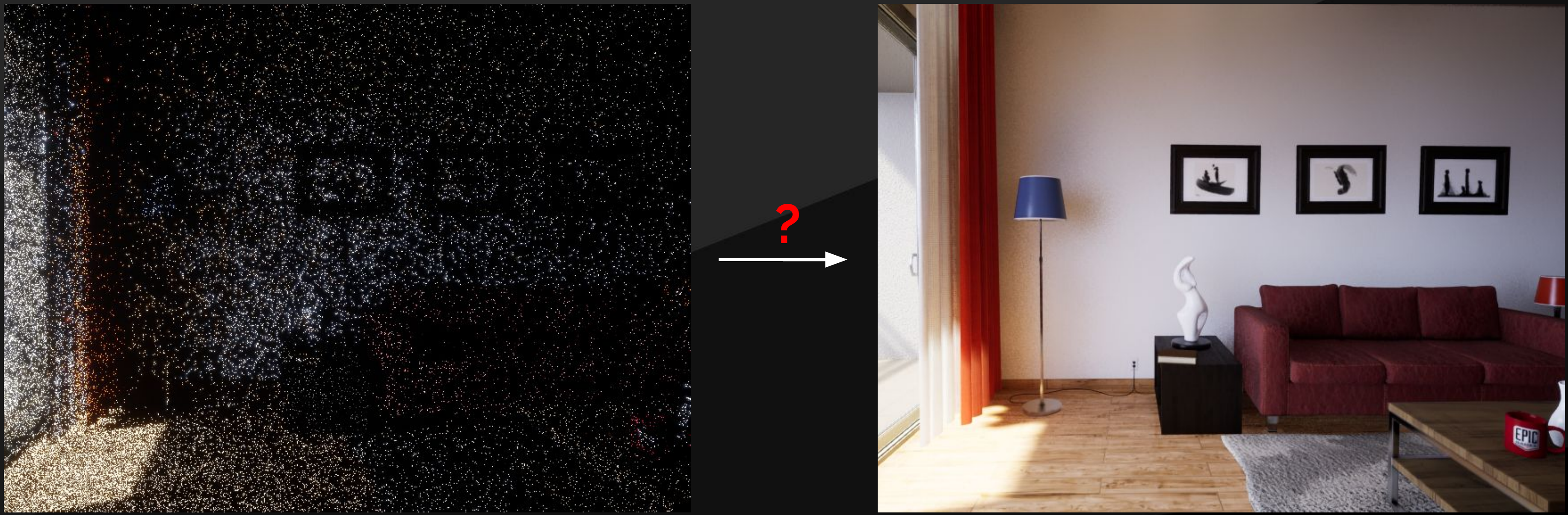


The first bounce is the most important, so we're going to separate that out and solve it with dedicated techniques.  For diffuse that's called the Final Gather, and for specular that's our reflection denoising.

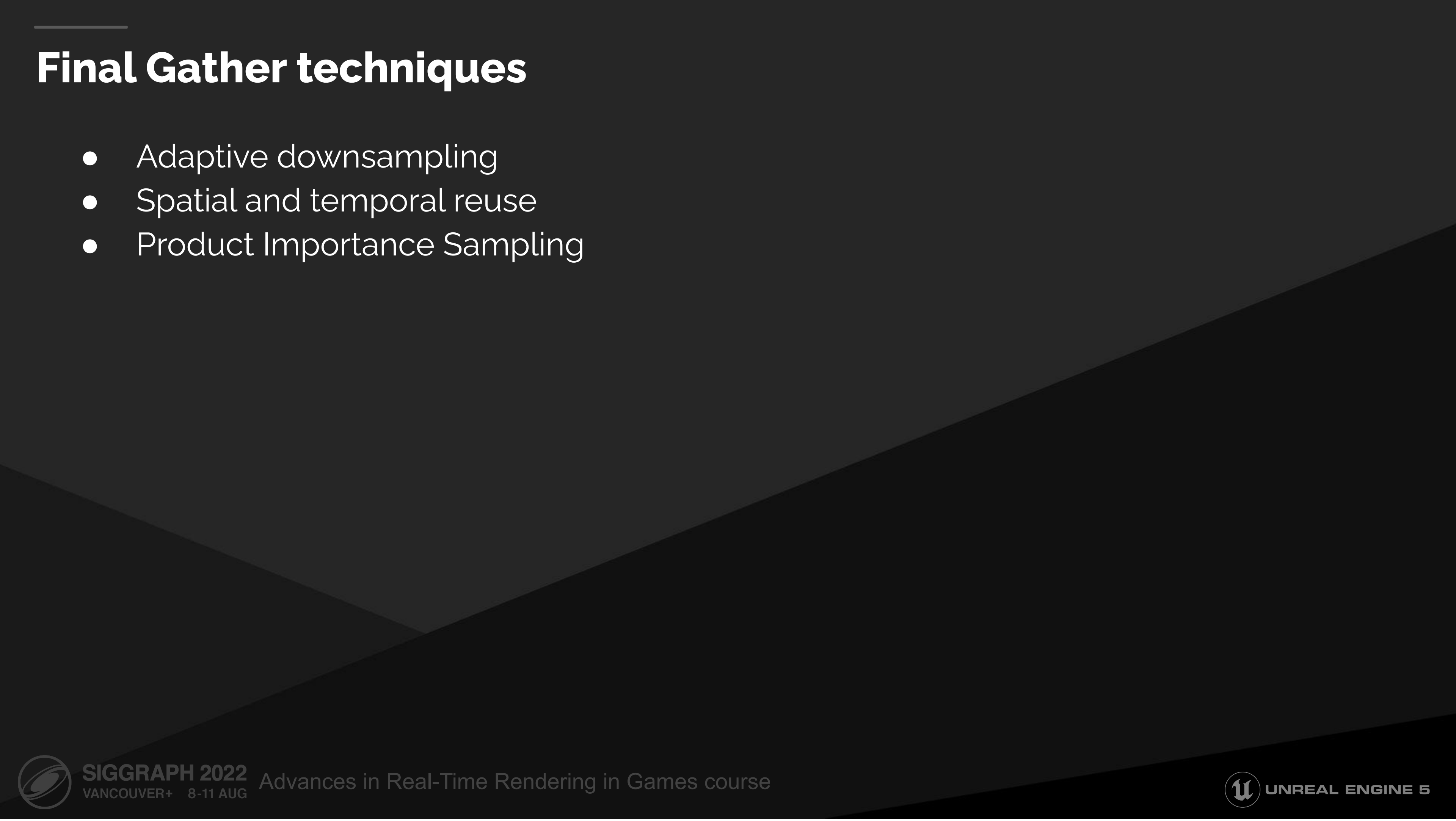
Any bounces after the first will be solved through our surface caching using feedback.  We'll do a gather from the surface cache, which will read from itself, and each update we'll propagate another bounce of indirect lighting.

# Fundamental problem #3:
# **How to solve noise in the light transfer?**

- We can't even afford one ray per pixel
  - But need hundreds of effective samples for high quality indoor GI



The third fundamental problem that we have to solve for realtime global illumination is the noise in the light transfer, considering that we can't even afford one ray per pixel for diffuse GI, but we actually need hundreds of effective samples for high quality indoors.



# Final Gather techniques

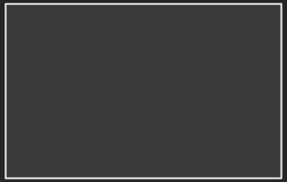
- Adaptive downsampling
- Spatial and temporal reuse
- Product Importance Sampling

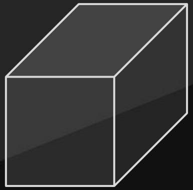
That’s where all of our Final Gather techniques come in. We’re going to use adaptive downsampling to trace the fewest number of rays possible, spatial and temporal reuse to make the best use of those rays, and Product Importance Sampling to send them in the best directions possible. I’ll go into more detail in how these work later.

# Final Gather domains

Opaque - Screen space

- ○ Contiguous 2.5d



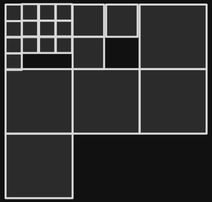


Transparency and Fog - Camera aligned volume

- ○ Contiguous 3d

Surface Cache - Texture space

- ○ Noncontiguous 2d



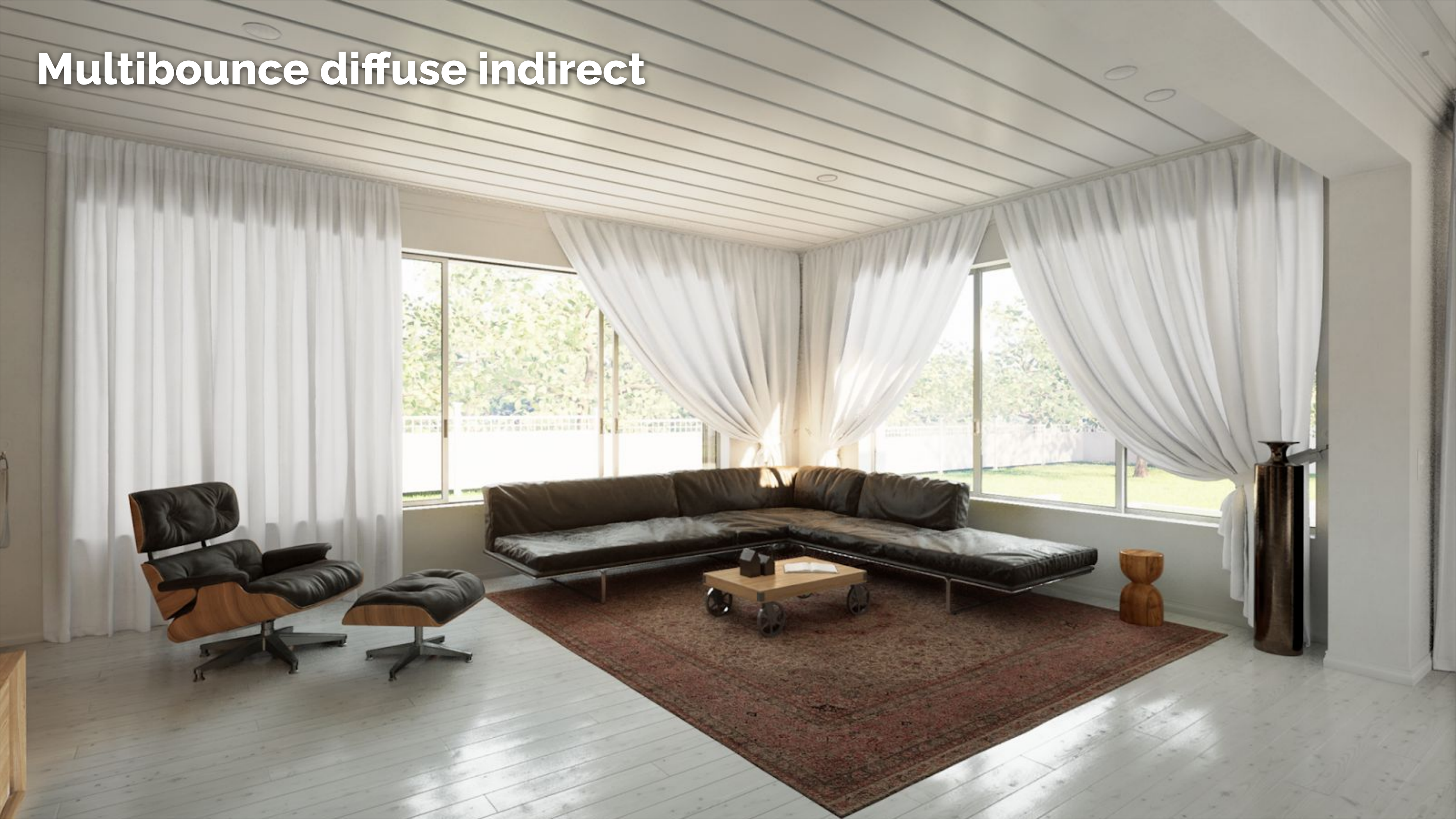
We can't just solve the Final Gather for opaque though, we need to also solve it for Transparency and Fog. For opaque we're operating in a 2d domain, while fog needs to be solved for all the points in the visible frustum, and for the surface cache we're gathering in a texture space domain.

# Reflection denoising

- Spatial and temporal reuse
- Bilateral filter
- Reuse diffuse rays

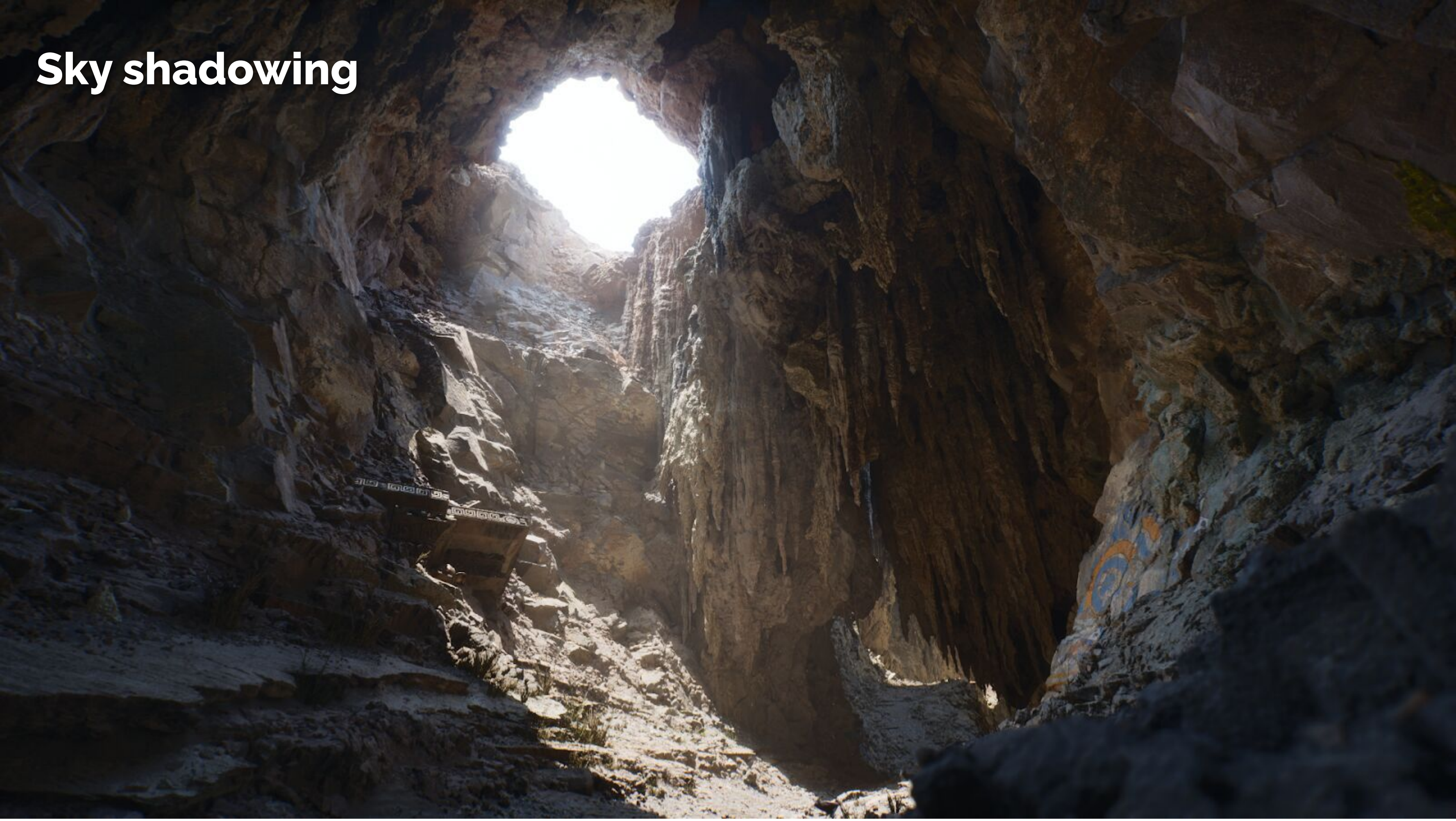
To solve the noise in reflections we're going to use spatial and temporal reuse to make the best use of the rays, and we're going to reuse the diffuse rays where possible.

# Multibounce diffuse indirect



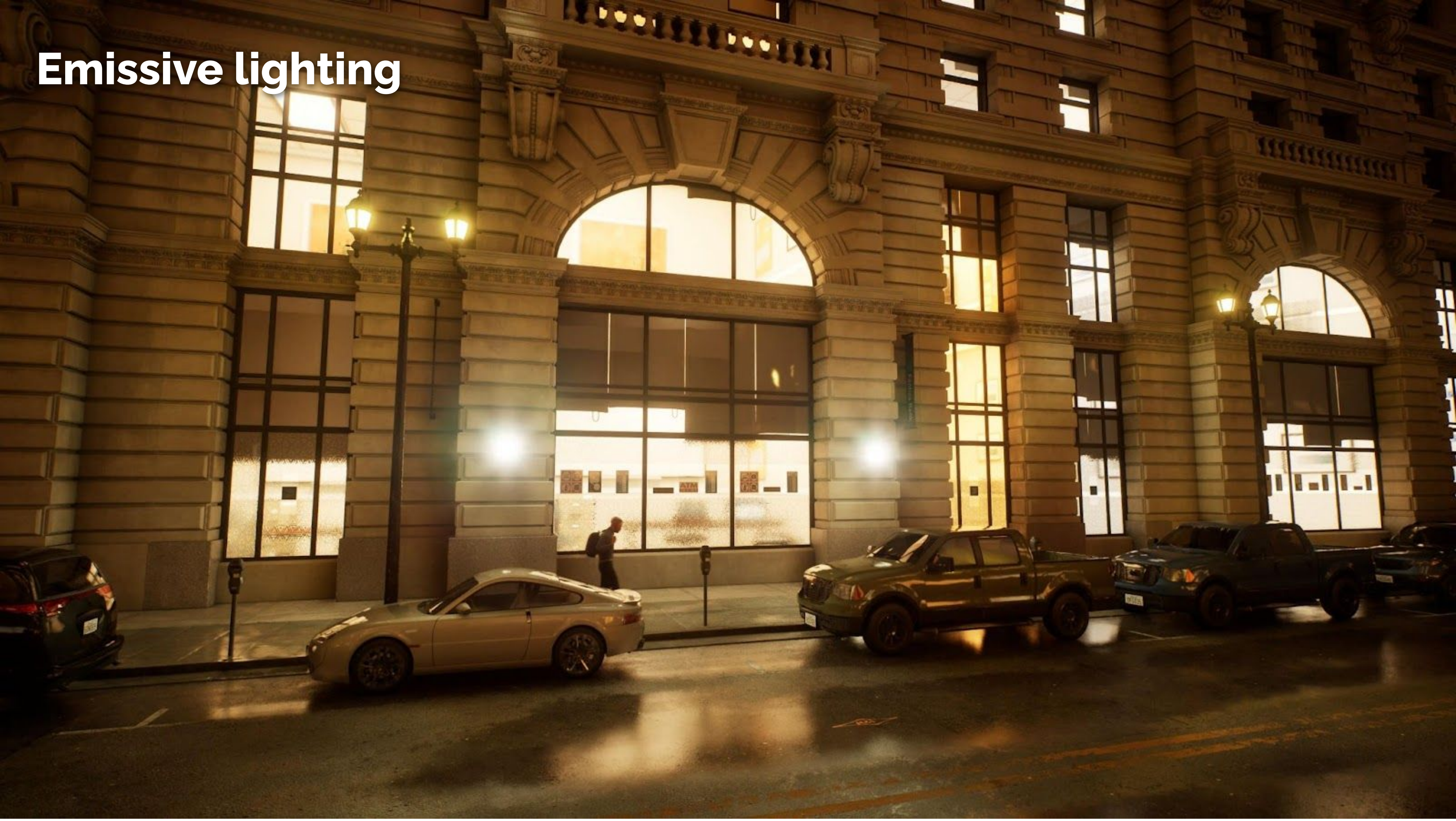
Putting all of that together, Lumen solves multi-bounce indirect lighting

# Sky shadowing



Sky shadowing

# Emissive lighting

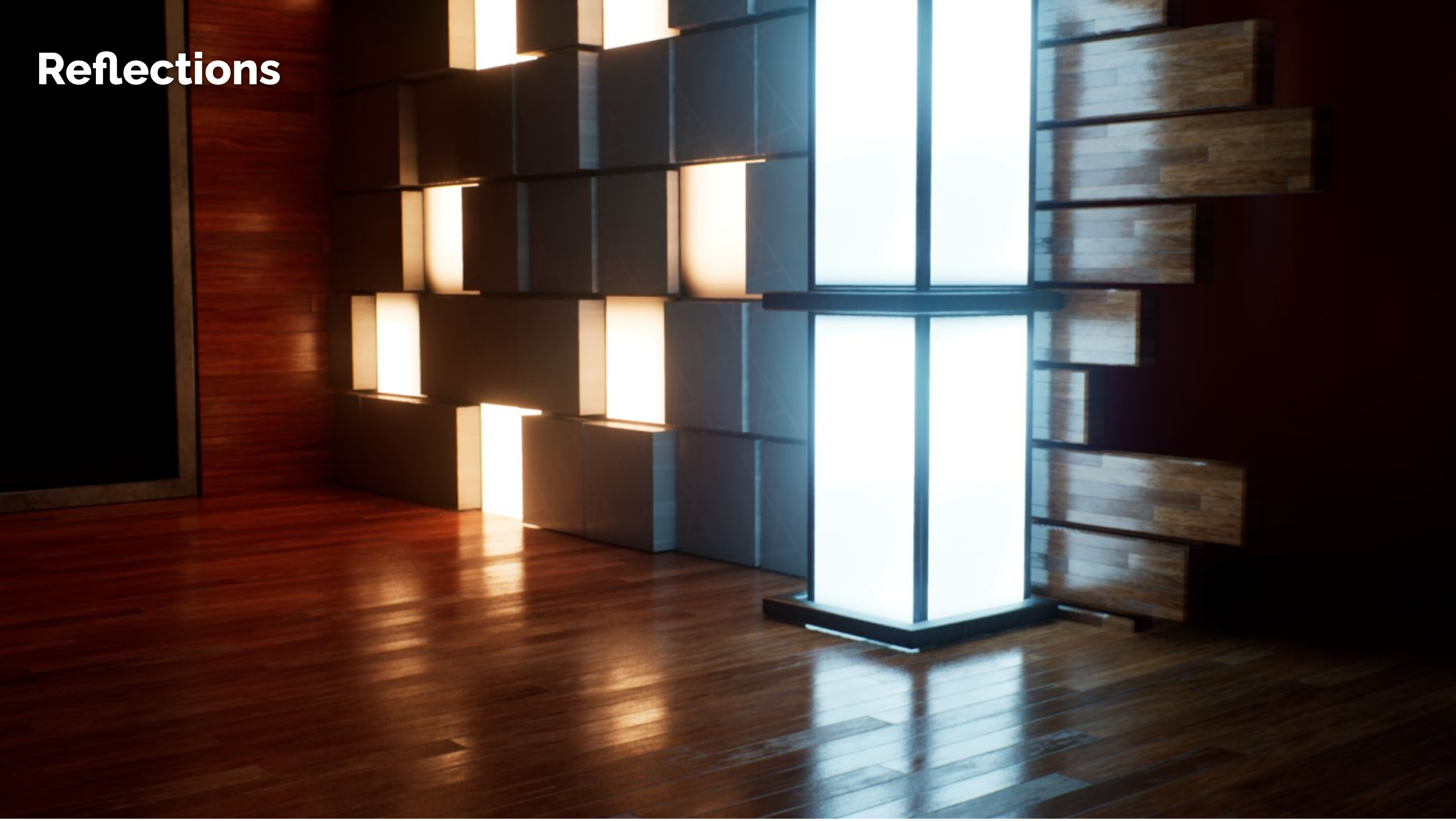


And emissive lighting, without artists needing to place individual light sources, although there is a limit to how small and bright those emissive areas can be without causing noise artifacts.

# GI on Volumetric Fog

Lumen solves GI on volumetric fog and transparency,

# Reflections



and Lumen solves reflections.  It provides all of these features very efficiently with Surface Caching, and it can use either Software or Hardware Ray Tracing, depending on the project’s needs.

# Outline

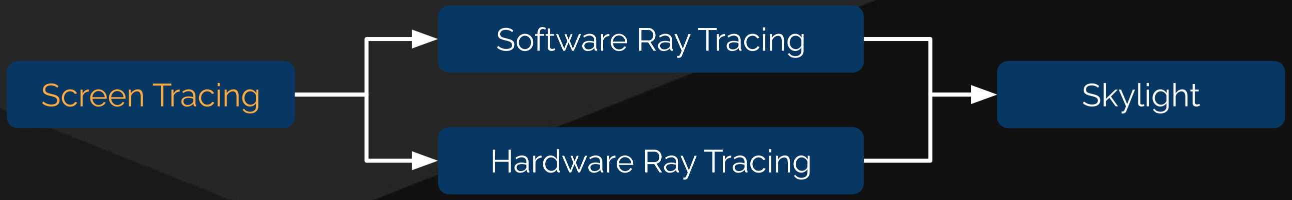
- Ray Tracing pipeline
  - Screen Tracing
  - Software Ray Tracing
  - Surface Cache
  - Hardware Ray Tracing
  - Tracing Performance
- Final Gather
- Reflections
- Performance and Scalability

That’s the end of the introduction, now we’re going to do a deep dive into each area of Lumen, starting with the Ray Tracing pipeline, and moving on to the Final Gather, Reflections and Performance and Scalability.

When we get to Software Ray Tracing I’m going to hand off to my colleague Krzysztof.

# Hybrid ray tracing pipeline

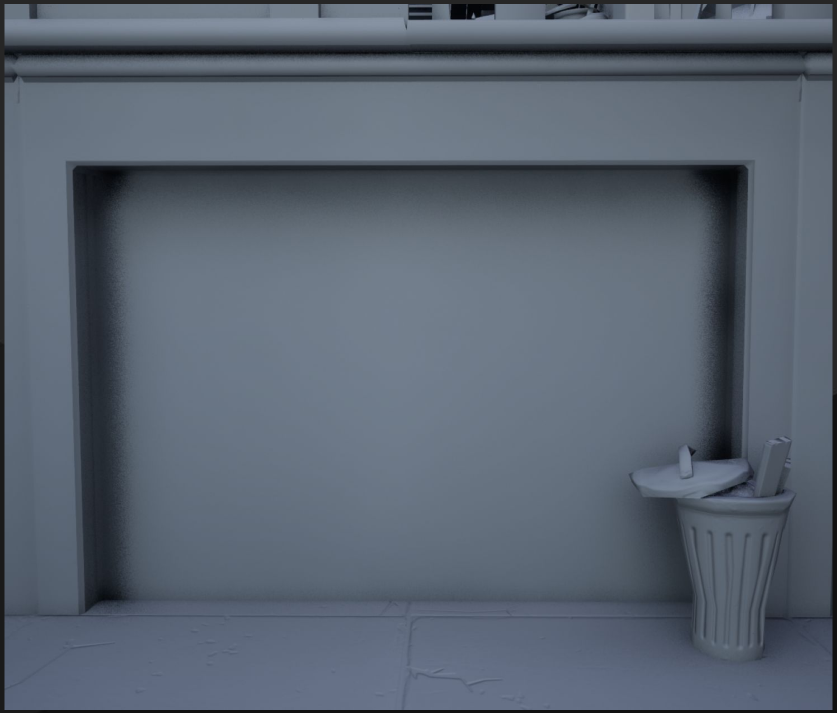
- Each tracing method hands off to the next
  - Write TraceDistance, bHit
- Build on the strengths of each



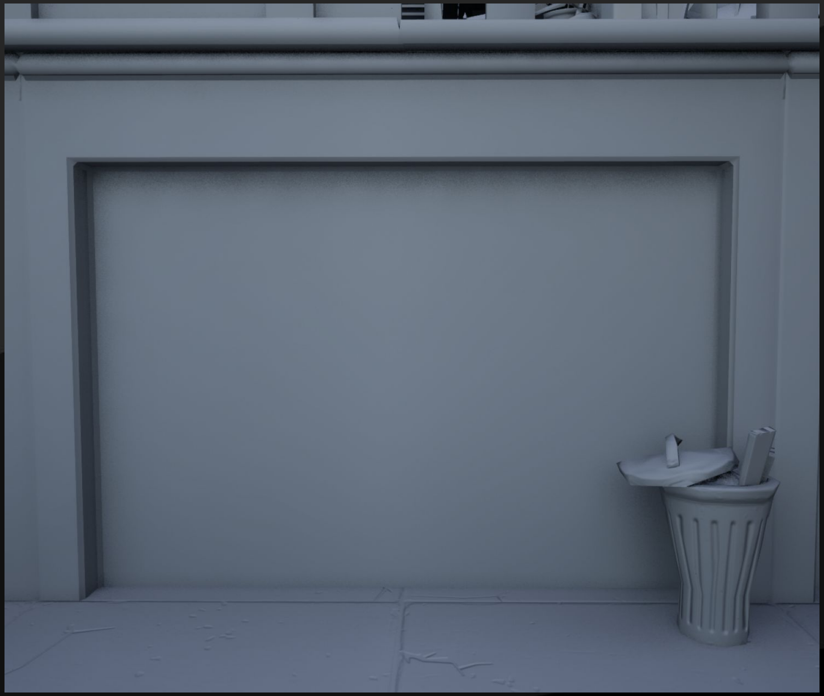
Lumen uses a hybrid ray tracing pipeline, which allows us to mix and match different techniques.  Screen tracing goes first, and then each tracing method hands off to the next, by writing out how far the ray traversed, and whether it found a hit or not.  The next tracing method can then resume the ray, picking up where the previous method left off.

# Screen tracing benefits

- Handles mismatches between the GBuffer and ray tracing scene



Self intersection artifacts



Fixed with Screen traces

Both Software and Hardware Ray Tracing have mismatches with the rasterized GBuffer for reasons that we'll get into later. Screen traces are good at handling these mismatches, which can cause self-intersection, in the case of starting inside the representation, or the mismatches can cause leaking.

# Screen tracing benefits

- Geometry types not represented by main tracing methods
- Detailed GI and occlusion at intersections

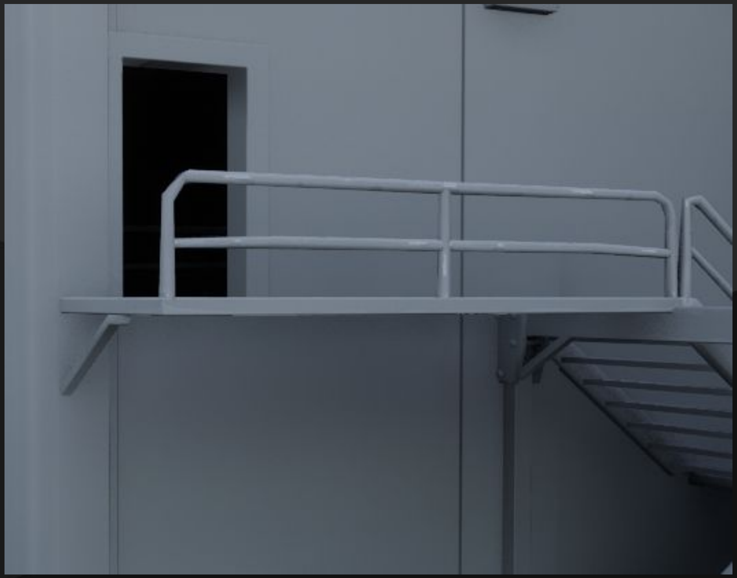
UNREAL ENGINE 5

Screen traces are also good at handling geometry types not represented by the main tracing methods, for example our Software Ray Tracing doesn't handle skinned meshes, but we can still get indirect shadows from a third person character using Screen traces.

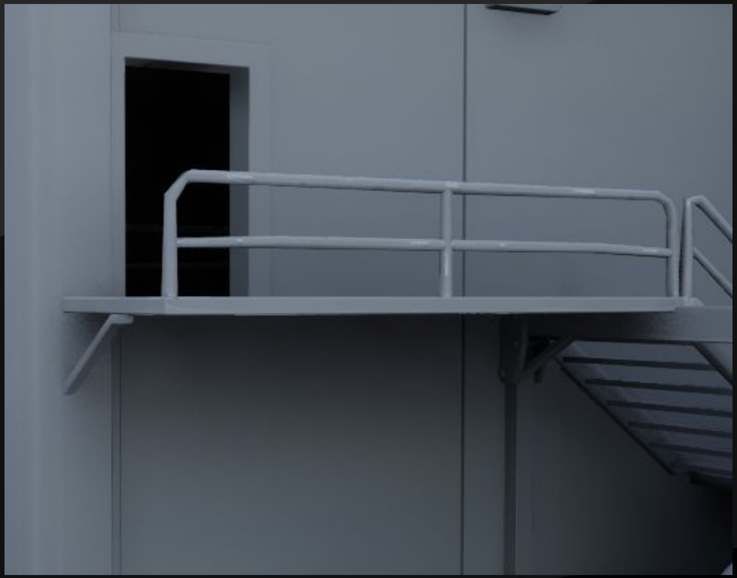
Screen traces work at any scale, so they work for detailed GI, no matter how much you zoom in.

# Linear steps skip thin occluders

- Tables, doors, walkways, etc



Linear steps

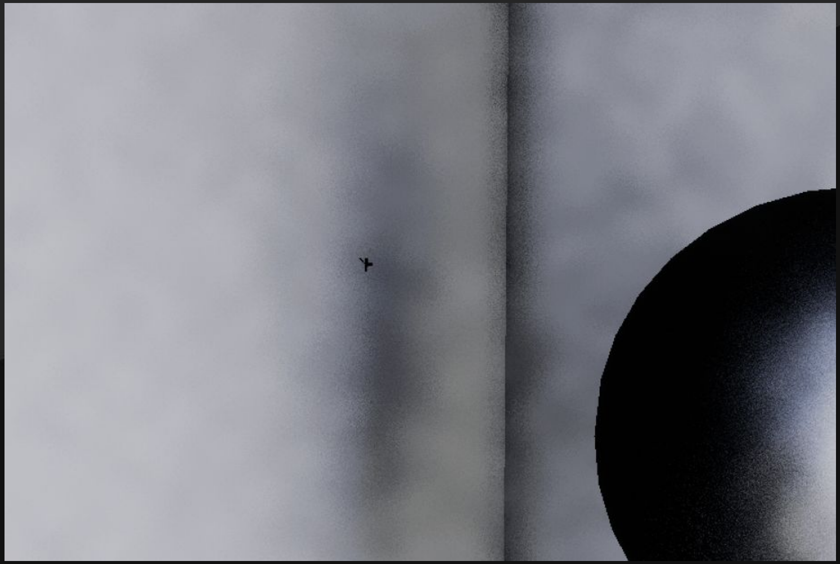


Reference

Screen traces don’t work well with linear steps, which skip thin objects, like the walkway shown here, which causes leaking.

# Hierarchical Z Buffer traversal

- Stackless walk of Closest HZB mips [Uludag 2014]
- Limit iteration count for grazing angle rays
- Diffuse rays use half res for faster tracing



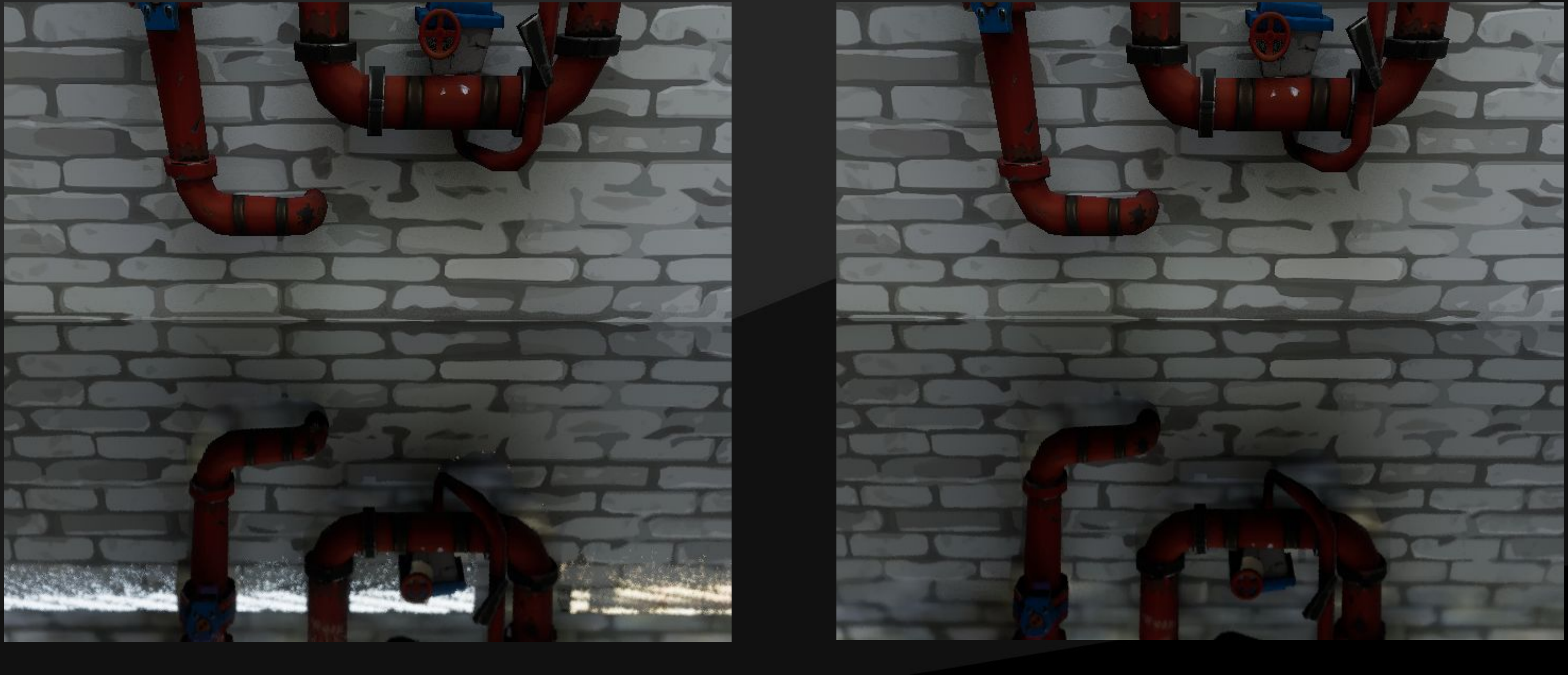
Steps 5 - 50

Instead we use a HZB traversal, which is a stackless walk of the Closest HZB mips.  We limit the iteration count for grazing angle rays, like a ray going parallel to a wall, and our diffuse rays use half resolution for faster tracing.

The animation here is showing the HZB traversal progress after every 5 steps.

# Hand off to the next tracing method accurately

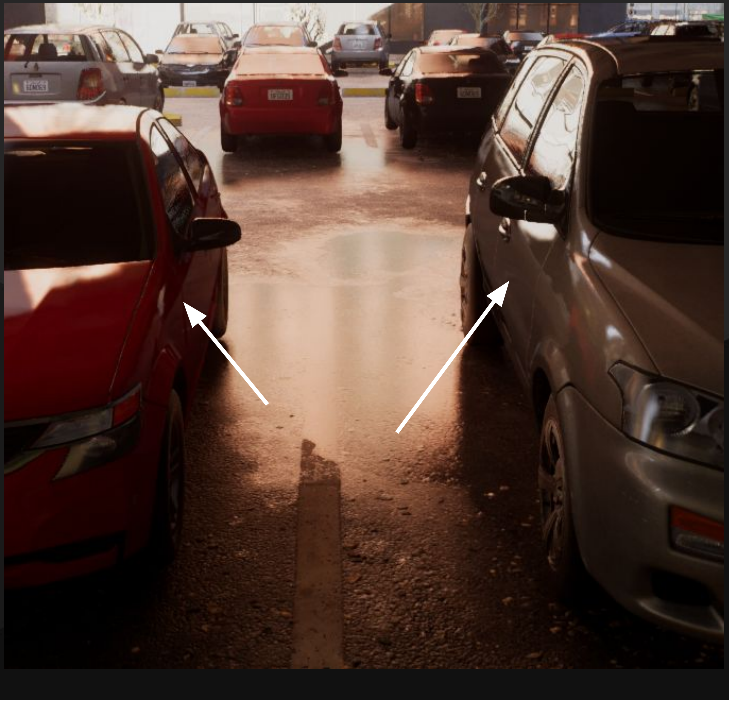
- Step back to last unoccluded position
  - This happens when ray goes behind a surface or offscreen

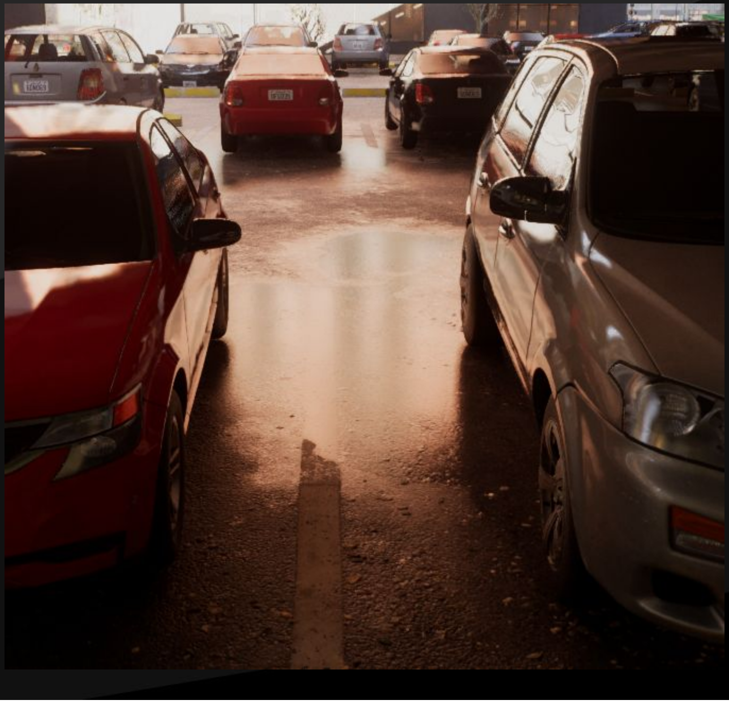


We have to make sure we hand off to the next tracing method accurately or we'll cause it to leak.  We do this by stepping back to the last unoccluded position, whenever the ray goes behind a surface or off the screen.

# Screen trace results

- Net quality win, despite downsides
  - The more it does, the more you notice when it fails
- Small performance win with HWRT

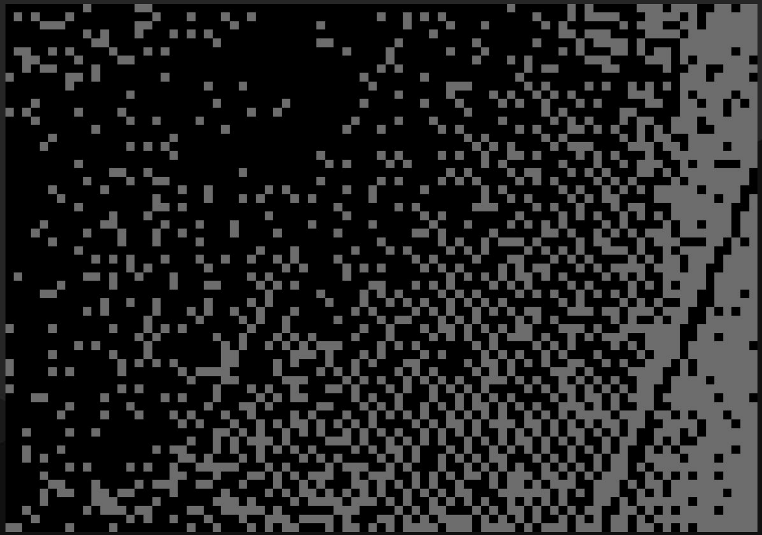




Ultimately, Screen traces are a net quality win, although the more they do, the more you notice when they fail. With Hardware Ray Tracing they also give a small performance win as they allow most of the rays to escape the complex surface geometry.

# Compact rays after Screen Tracing

- Some rays have been solved, remove empty tracing lanes
- Up to 50% tracing speedup

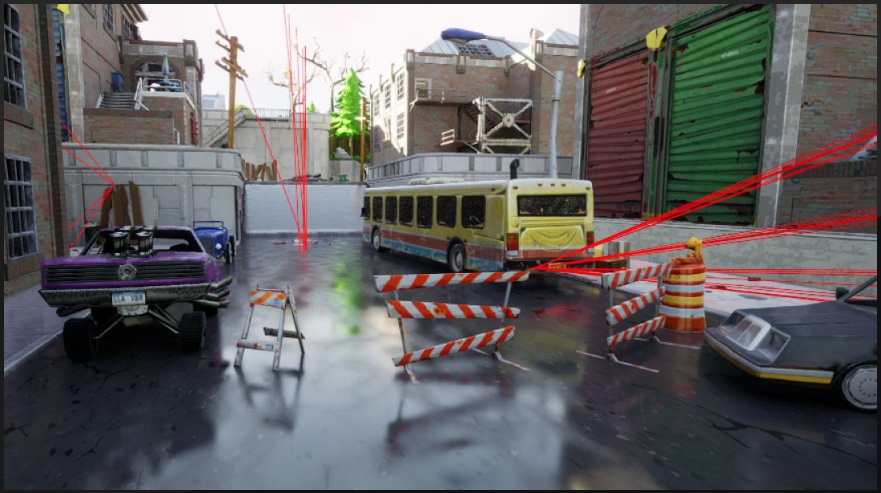


Hits and misses after Screen Tracing

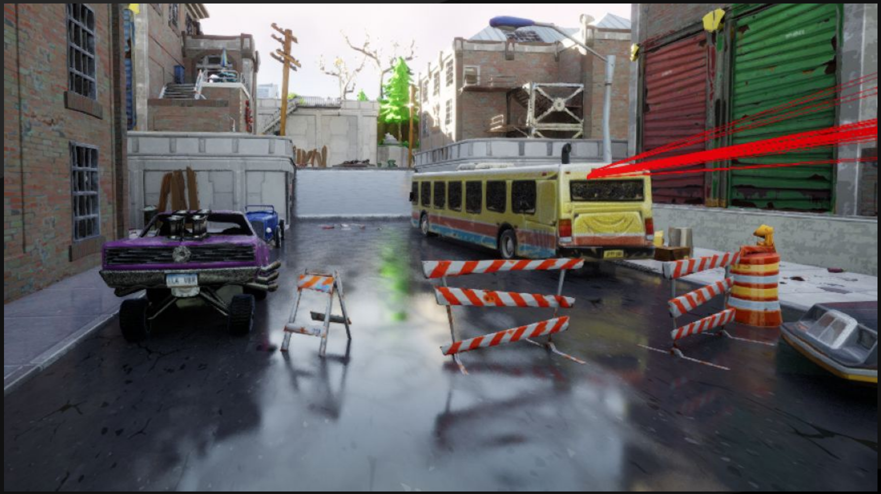
After Screen Traces run, some of the rays have been solved by Screen trace hits, and the rest still need to be traced further.  Instead of running the next tracing path on all of these empty tracing lanes, first we do a compaction, which gives a significant speedup.

# Coherency preserving compaction

- Z-order within large threadgroup (128x128)
- Waveops prefix sum gives local offset
- Atomics for global offset
- 21% faster reflection tracing than local atomics



Atomic compaction



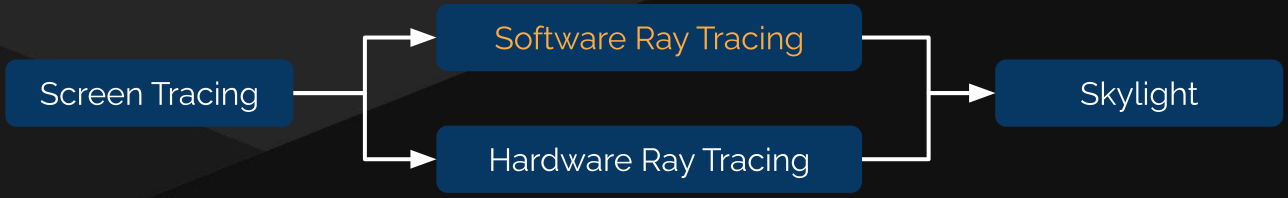
Prefix sum compaction

The easiest way to do a compaction is to use local atomics to allocate the compacted index. That has the effect of scrambling the rays, which you can see on the left. The red lines are all the different rays within a single wave, and they're now starting from different positions within the scene.

We solved that by doing an order preserving compaction, which does a fast prefix sum within a much larger threadgroup to allocate the compacted index.

Now I'm going to hand off to my colleague Krzysztof.

# Software Ray Tracing



Screen Tracing

Software Ray Tracing

Hardware Ray Tracing

Skylight

Hi, my name is Krzysztof and I’m going to present Software Ray Tracing and Surface Cache in Lumen.

# Why Software Ray Tracing?

- Option to scale down
  - Some GPUs don't support HWRT
  - 60 fps on consoles
  - Two level BVH isn't ideal for overlapping instances
- Full control over tracing code
  - Can't replace HWRT, but can pick different tradeoffs

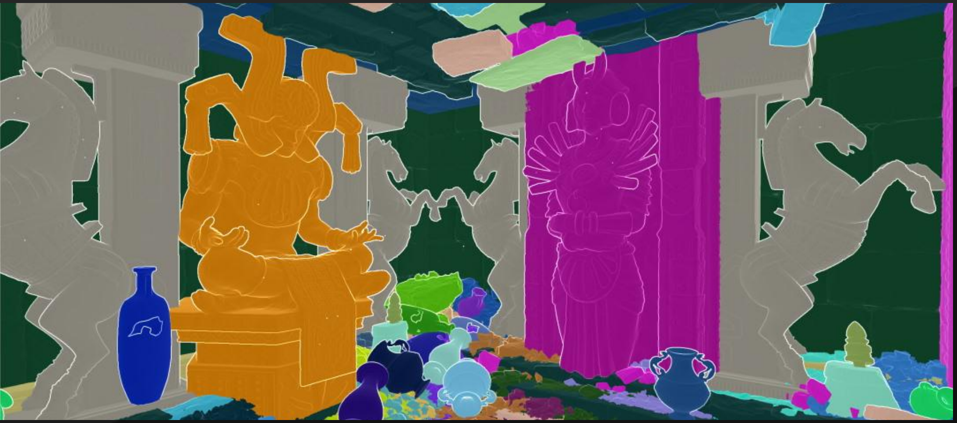
The first question is why even do any kind of software ray tracing when a hardware solution is available.

Unreal Engine supports different platforms and different types of content and we need different tools for handling such a wild variety of use cases. This is where our main motivation for software tracing comes in. We want to run Lumen on hardware without DXR support and we want to be able to scale down.

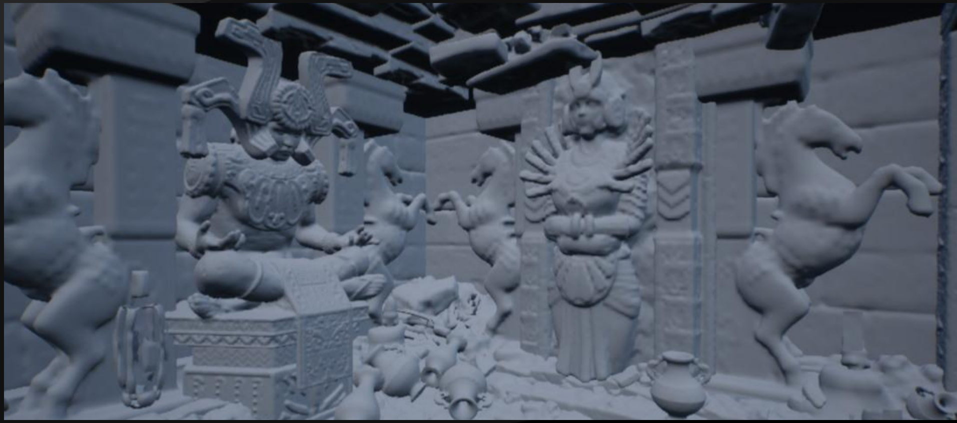
We cannot ever replace hardware ray tracing, but we have full control over the tracing loop and we can make different tradeoffs. For example, overlapping instances in a BVH are an issue as ray needs to visit each one of them in order to find the closest hit and we have no ability to change this acceleration structure.

# Overview

- Two primitives
  - Mesh SDF
  - Landscape heightfields
- Two level structure to leverage instances for storage
  - Array of instances on GPU
- SDF already used in UE4 [Wright 2015]
  - Will focus on new developments



SDF Instances



SDF Scene

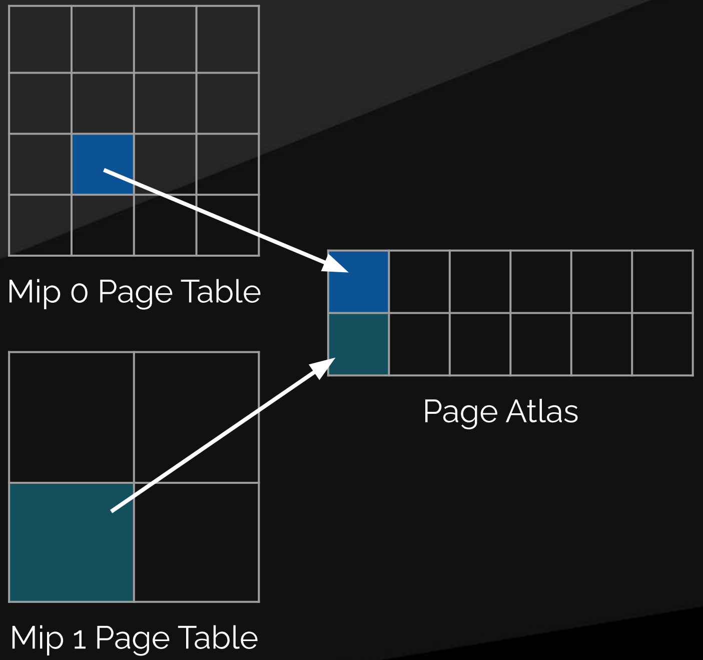
On a high level we have two basic primitives - a distance field per mesh and a heightfield per Landscape component.

Those primitives are stored in a two level structure, where on the bottom level we have our primitives and top level is a flat instance descriptor array. This approach allows us to leverage instances for storage and decreases memory usage, which is important for any kind of a volumetric mesh representation.

Distance fields aren't new in Unreal Engine and in this talk I'm going to focus on new developments which were required for Lumen.

# Mesh SDF

- Generated on mesh import
  - Resolution based on mesh size
  - Embree point query
  - Trace rays and count triangle back faces for sign
  - ~0.6ms to build a large 1.5M tri mesh
- Virtual volume texture per mip
  - Sparse 8^3 bricks with 0.5 texel border
  - Narrow band SDF
  - [-4;+4] voxel distance in 1 byte



We generate mesh distance fields during mesh import and store them alongside other mesh data.

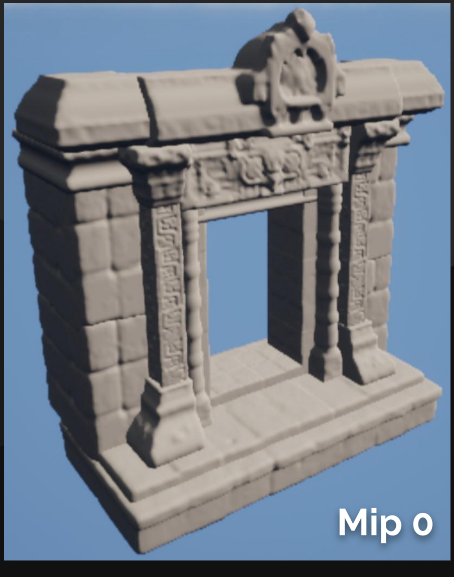
For generation we use Embree point query to efficiently find distance to the nearest triangle. We also cast 64 rays from every voxel and count backface hits to decide whether we are inside or outside the geometry, which determines the distance field sign.

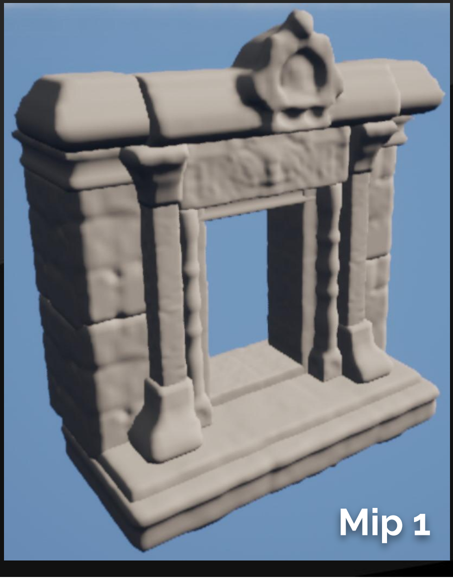
Volumetric structures don’t scale well with resolution and are quite memory intensive, so we store only a narrow band distance field inside a mip mapped virtual volume texture.

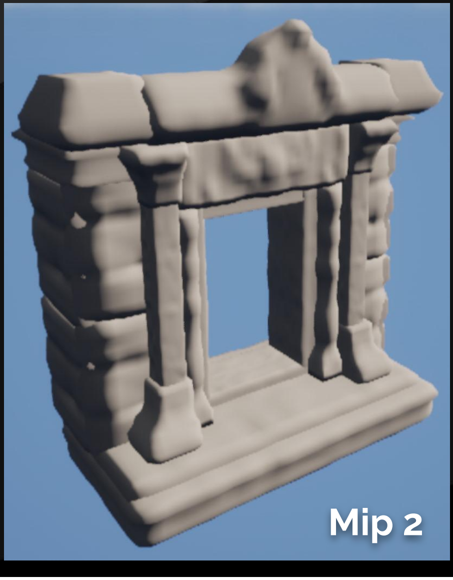
Mip0 resolution is based on the mesh size and mesh import settings, and then mip1 halves the resolution and doubles max object space distance, and so on.

# Mesh SDF Streaming

- Every frame GPU gathers requests
  - Min distance to camera
- CPU downloads requests and streams pages in/out
- Fixed memory pool (320mb) of bricks
  - Linear allocator - no fragmentation







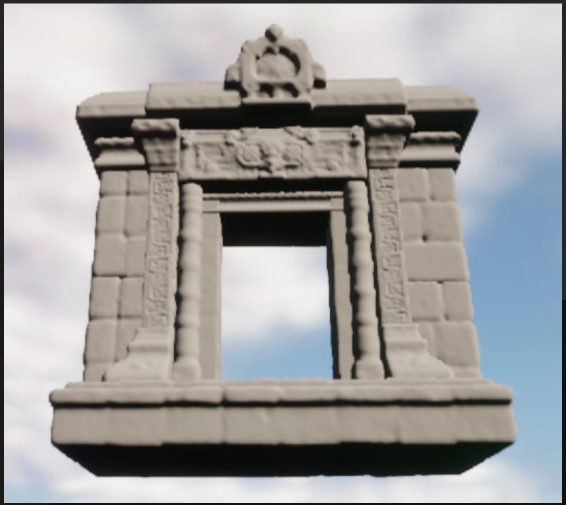
Then every frame we dispatch a shader to loop over all instances. This shader computes the required mip level for every distance field asset based on the distance from the camera.

Next we download those requests to CPU and stream in additional distance field mips or drop mip maps which aren’t used anymore.

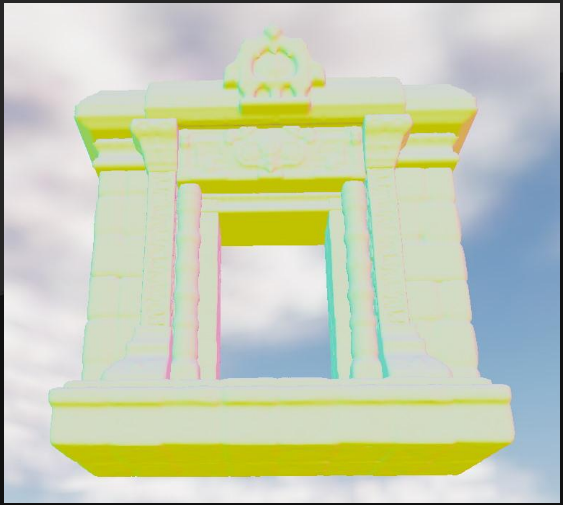
Distance field bricks are stored inside a fixed size pool, which is managed by a simple linear allocator. It’s a convenient setup, as we don’t need to deal with any kind of a variable sized 3d allocations or resulting fragmentation.

# Mesh SDF Tracing

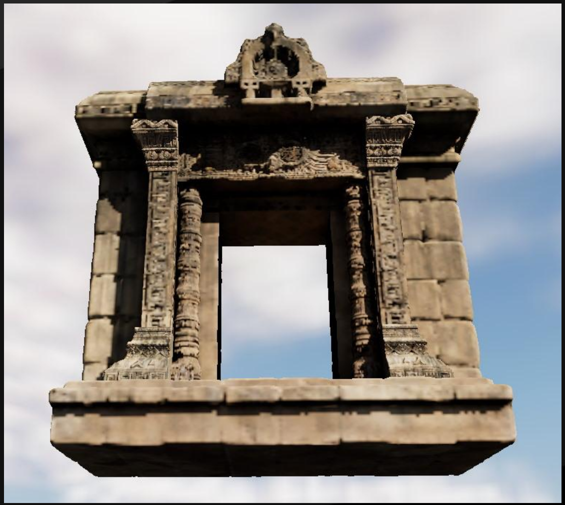
- Raymarch and step mips [Aaltonen 2012]
  - Force 64th iteration to hit
- Evaluate normal at hit point
  - 6 samples, +/-0.5 voxel for gradient
- Sample surface cache



Mesh SDF



Mesh SDF Normal



Shaded Mesh SDF

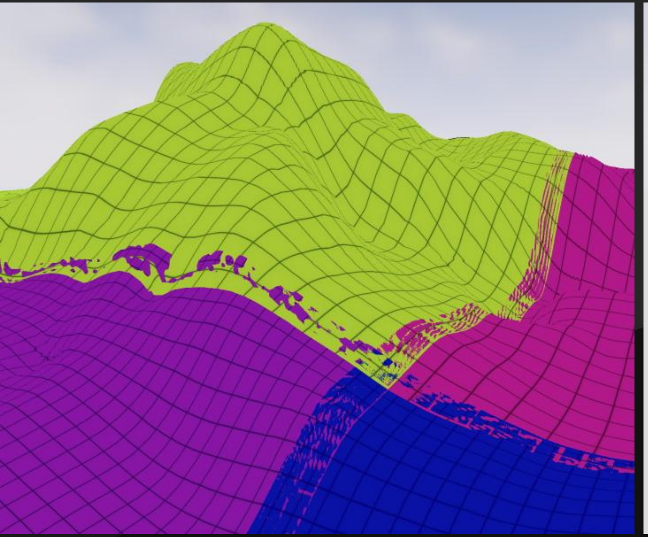
For mesh distance field tracing we use mipmaps to accelerate ray marching. When we get too close to the surface, we use a higher and more precise mip, and when we get too far away, then we switch to a lower one in order to step through the empty space faster. This is similar to what Sebastien did for Claybook.

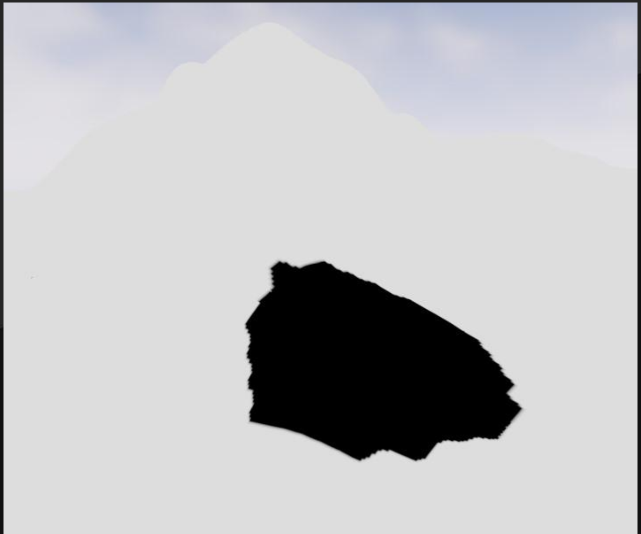
We also limit mesh distance field ray marching iterations to 64 for performance reasons, and if we hit this limit then we stop traversal and report a hit at a current ray distance.

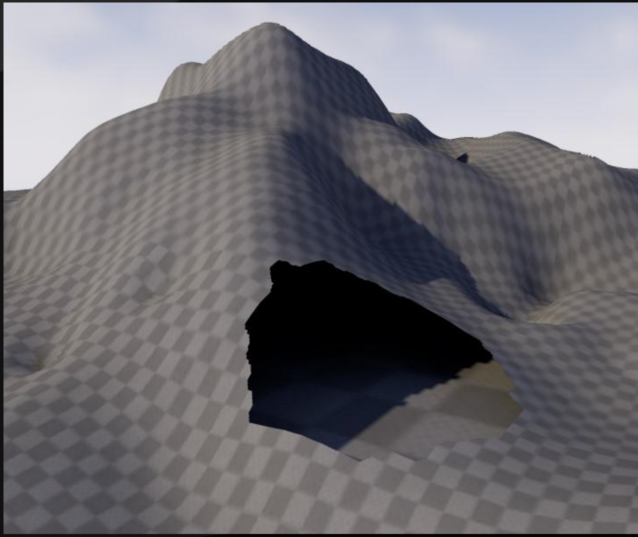
Finally after finding the hit, we use central differencing with 6 samples to compute the geometry normal. This normal is later used to sample material and lighting from the surface cache.

# Heightfields

- Raymarch heightfield
  - Find zero crossing
  - LERP last two steps for final hitT
- Evaluate surface cache on hit
  - Continue tracing if hit point is transparent


Heightfields


Opacity Mask


Raymarched Scene

Landscape is divided into components and we have a single heightfield in surface cache for each component.

On the top level, heightfield instances are handled the same way as mesh distance field instances, reusing culling and traversal code.

Bottom level is different, as per instance instead of a 3d distance field, we raymarch a 2d heightfield and try to find a zero crossing. After finding two samples, where one is above and the other is below heightfield, we linearly interpolate between them to approximate the final hit point.

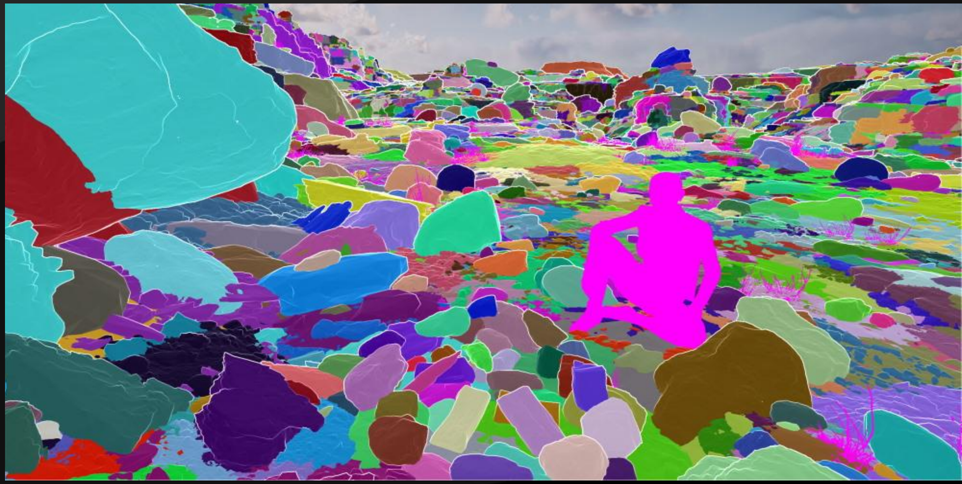
Having this hit point we can now evaluate opacity from surface cache to decide whether we should accept this hit or we should skip it and continue tracing below the heightfield.

After accepting a hit we evaluate surface cache at that point to compute ray radiance.

# Acceleration Structure

- Can't just loop over all instances
- Tried BVH and grid
  - Too slow with long incoherent rays
  - Ray has to check every overlapping mesh
- Can afford only short rays
  - Continue tracing using a different method

Scene



Instance visualization

At this point we have all the data in the memory and we know how to trace an individual instance. Now we need to figure out how to trace the entire scene as we cannot just loop over all instances in the scene and raymarch each one of them.

We tried BVH and grids. Those are really nice acceleration structures as you can build them once per frame and then reuse them in multiple passes. Unfortunately performance of long incoherent rays wasn't good enough. Software BVH traversal has a quite complex kernel. Grids have complex handling of objects spanning multiple cells. On top of that scenes with overlapping instances require to raymarch each one of them in order to find the closest hit.

Finally we decided to simplify this problem and only trace short rays, and when the ray footprint gets wider we would switch to a different tracing method.

# Runtime Scene LOD

- Coarse and fast rays
  - Diffuse rays don't need to be precise after leaving surface
  - Opportunity to simplify and merge scene
- Offline scene merging - inflexible
- Voxel Cone Tracing - leaky
- Voxel bit bricks - slow
  - 1 bit per voxel in a 8^3 brick
  - Distance Field to accelerate?

This was an important realization that we need a precise scene representation only for the first segment of a ray and after that we can switch to a coarse scene representation. This also gave us an opportunity to solve the object overlap issue, as now we can merge entire scene into some simplified global representation.

We tried different approaches here.

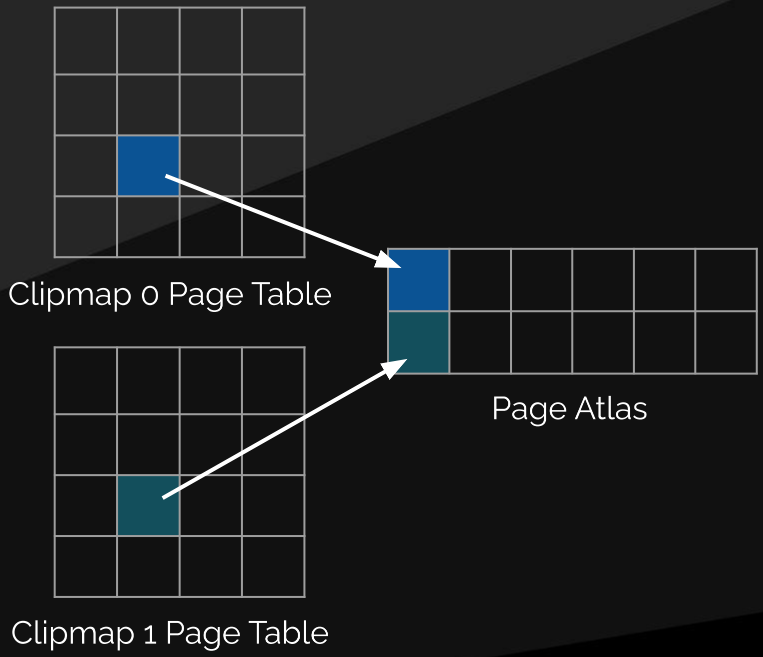
Obvious one is to merge the entire scene as a part of a scene build step, but that’s a quite restrictive workflow and it doesn't support dynamic objects.

We tried runtime voxelization and voxel cone tracing, but merging geometry properties into a volume causes lots of leaking, especially in the lower mip maps.

We also tried voxel bit bricks, where we stored 1 bit per voxel to mark whether it contains geometry or not. Simple ray marching of bit bricks was surprisingly slow and after adding a proximity map for acceleration, we just decided to drop voxels and arrived at a Global Distance Field.

# Global SDF

- Merge instances at runtime
- 4 clipmaps of 256^3
- Virtual volume texture per clipmap
    - Sparse 8^3 bricks with 0.5 texel border
    - Narrow band SDF
    - [-4;+4] voxel distance in 1 byte



Clipmap 0 Page Table

Page Atlas

Clipmap 1 Page Table

The global distance field merges all mesh distance fields and heightfields into a set of clipmaps centered around the camera. Mesh distance fields and heightfields are a perfect mesh representation for this as they are simple to merge and LOD at runtime.

By default we use 4 sparse clipmaps of virtual volume textures. Every clipmap stores distance field bricks and every brick stores a narrow band distance field.

This is a similar setup to mesh distance fields, but instead of a mipmap hierarchy we use a clipmap hierarchy as we want to simplify scene further away from the camera.

# Global SDF Caching

- Full update is slow
- Timesplice and LOD per clipmap
- Dynamic cached separately from static
  - Most of the objects don’t move
- Update only modified regions
  - Track updates and build a list of modified bricks
  - Cull objects to modified bricks
  - Sample SDF for fine culling
  - Allocate and free bricks

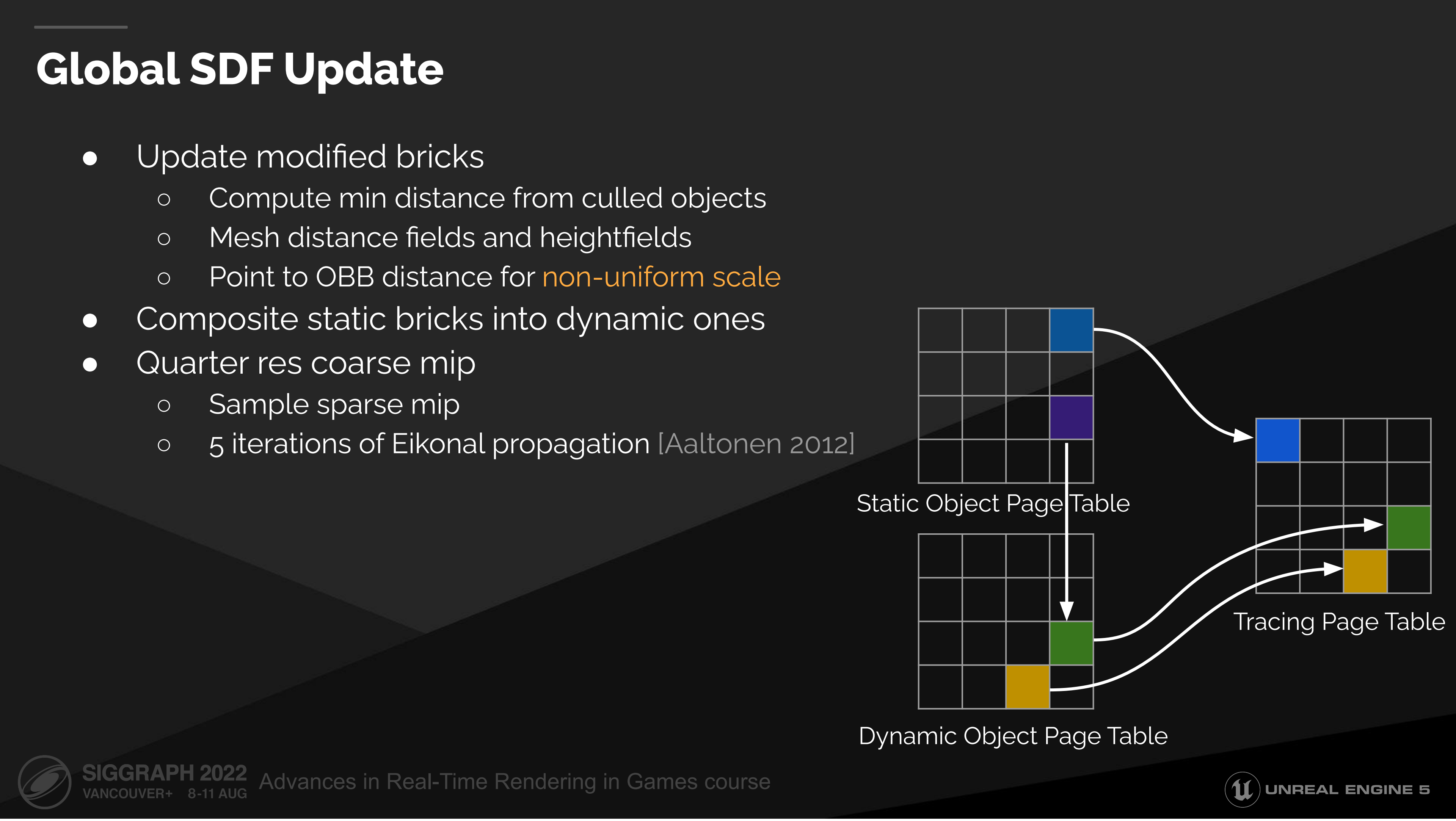
Merging all objects in the scene is expensive, so we need to aggressively cache and update only what changed since the last frame.

We also time splice updates by updating further clipmaps less often and we have separate LOD settings per clipmap, which allows us to drop smaller objects in the distance. This really helps with update performance as largest climaps also have the largest number of instances to merge.

Usually only a few objects in the scene move and the rest is just completely static. We leverage this by splitting our cache into dynamic and static bricks, so when a car moves we don't need to recompute static buildings around it.

For cached updates we track all scene modifications and build a list of modified bricks on the GPU. Next we cull all the objects in the scene to the current clipmap and then cull resulting list to modified bricks. During the last culling step we sample mesh distance fields for more accurate culling that checking analytical object bounds.

At this point we have a list of modified bricks which need to be updated and a list of culled objects per modified brick. We can now allocate and deallocate persistent bricks and proceed to updating them.



# Global SDF Update

- Update modified bricks
  - Compute min distance from culled objects
  - Mesh distance fields and heightfields
  - Point to OBB distance for non-uniform scale
- Composite static bricks into dynamic ones
- Quarter res coarse mip
  - Sample sparse mip
  - 5 iterations of Eikonal propagation [Aaltonen 2012]

Static Object Page Table

Dynamic Object Page Table

Tracing Page Table

In order to update a single brick we loop over all objects affecting it and compute min distance per voxel.

One issue here is that instances can have non-uniform scale, but distances stored in a distance field are valid only for an uniform object scale.

We tried finding the nearest point through an analytical gradient and then recomputing distance from it, but it didn’t work well in practice due to the limited distance field resolution. In the end what worked for us is simply to bound the distance field using distance to analytical object bounds. Most of the non-uniformly scaled objects are also simple shapes like walls so it works really well in practice.

Also when updating dynamic bricks, we need to composite overlapping static ones in order to combine both caches into a final distance field for tracing.

Finally we update our coarse mip, which is a quarter res non sparse distance field volume and is used to accelerate empty space skipping. We use the coarse mip instead of stepping through the clipmap levels as our clipmaps have different LOD levels and objects may be missing from the largest ones.

Coarse mip is quite low resolution, so we always update entire volume by sampling the global distance field and then doing a few Eikonal propagation iterations to expand it.

# Global SDF Tracing

- Raymarch through clipmaps
  - Coarse mip to accelerate
- Compute normal
- Sample surface cache

Global SDF

Global SDF Normal

Shaded Global SDF

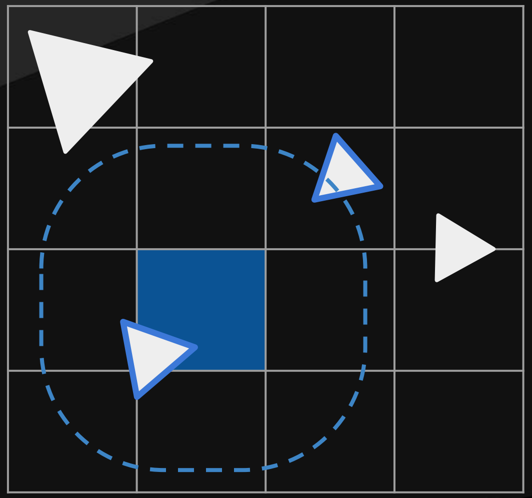
For global distance field tracing we loop over clipmaps starting from the smallest one and raymarch each one of them until we find a hit.

Every step we first sample the continuous coarse mipmap and if we are close to the surface then we also sample sparse bricks.

Finally when we find a hit, we compute surface gradient using 6 taps and sample the surface cache to get lighting.

# Mesh Object Traversal for GI

- Limit detail traces to 2m
  - Continue rays using Global SDF
- Cull objects to frustum
- Mark used froxels from screen pixels
- Cull objects to used influence froxels
  - Sample SDF for fine culling
  - Compact culled list
- Tracing
  - Load a single cull grid cell
  - Trace objects to the last found hit



Now when we have a far field trace fallback method we can come back to the mesh distance field tracing.

With the assumption of tracing only short rays we don’t need BVH or world space grids anymore. Instead we can cull objects to an influence froxel grid, where every cell contains a list of all objects which need to be intersected if a ray starts from that cell.

To produce this list, we first cull all scene objects to frustum. Then we mark froxels which contain any geometry, so that we don’t waste time on froxels which won’t be ever used during the tracing pass.

Next we cull objects to marked froxel cells. First object culling test is just a rough bounds test and the second one is a precise distance field sample.

Finally we compact culled lists into a continuous array of objects.

When tracing from pixels for GI or reflections, we will load an appropriate cell, loop over all objects in it and ray march them till the last found hit. This results in a very simple and coherent tracing kernel.

# Mesh Object Traversal for Directional Shadows

- Full length rays
- Cull objects to light space 2d cells
  - Rasterize bounding volumes
  - Sample SDF for fine culling
  - Compact culled list
- Tracing
  - Load a single cull grid cell
  - Trace objects until any hit

Directional shadow rays are parallel and we cannot depend on a cone footprint getting wider here. Which means that we need to trace full length rays.

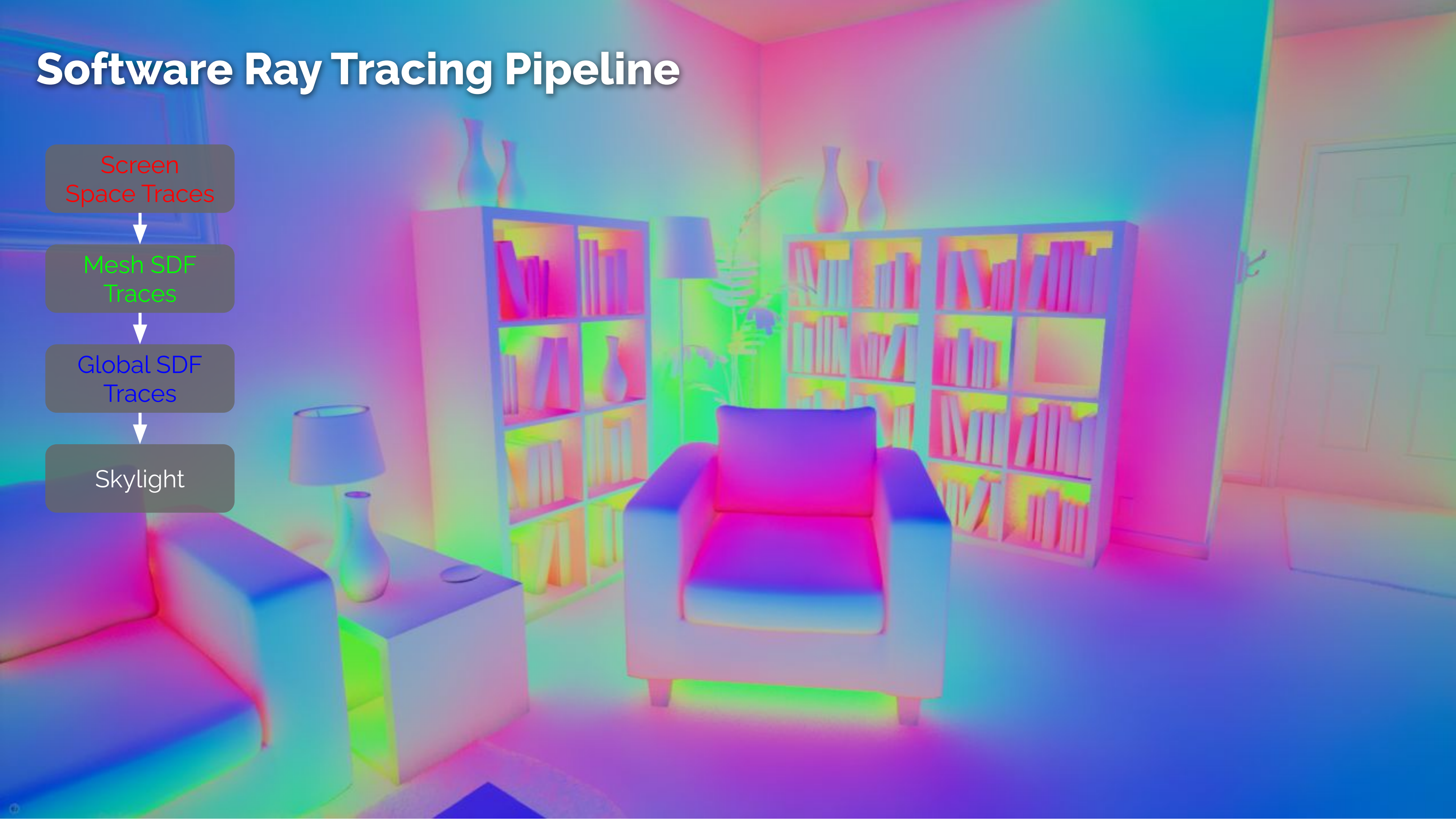
To do that we cull objects to a light space 2d grid, where every cell contains array of objects to intersect when starting ray from that cell.

Next to fill this grid we scatter objects by rasterizing their object oriented bounds. Inside the pixel shader we do extra fine culling by sampling the mesh distance field.

Finally culled lists are compacted.

And when tracing shadow rays, we will load an appropriate cell, loop over all objects inside it and ray march them until we find any hit.

# Software Ray Tracing Pipeline

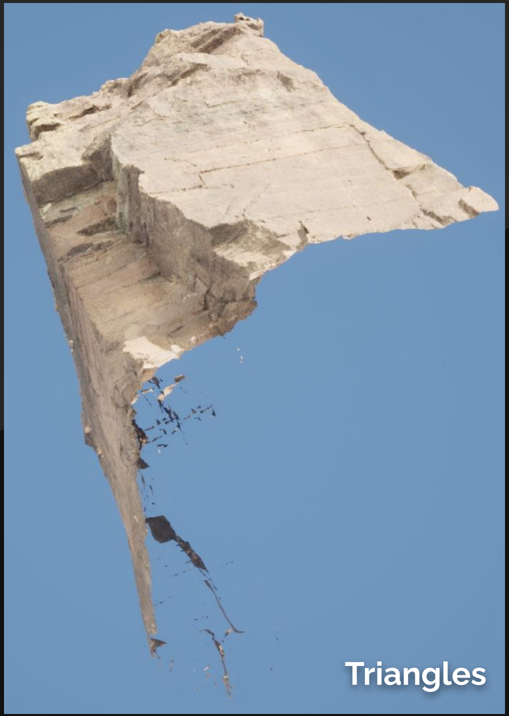


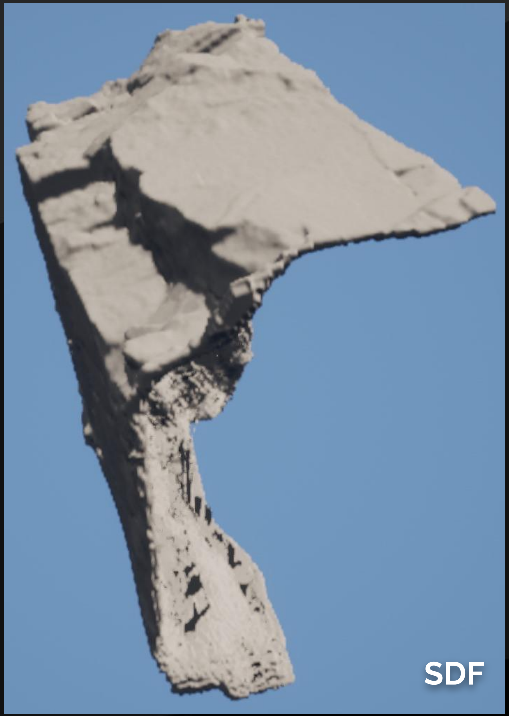
Now we have all the pieces and we can combine them into a full software ray tracing pipeline.

We start from screen space traces.
Then we continue unresolved rays using short mesh distance field traces.
Next we continue using global distance field traces.
And finally any ray miss will sample the skylight.

# One sided surfaces in SDF

- Meshes may not be closed
- Infinite negative region
- Add virtual surface to close
  - Wrap after 4 voxels





Before we conclude, there a few issues with distance fields which we had to solve to make it practical.

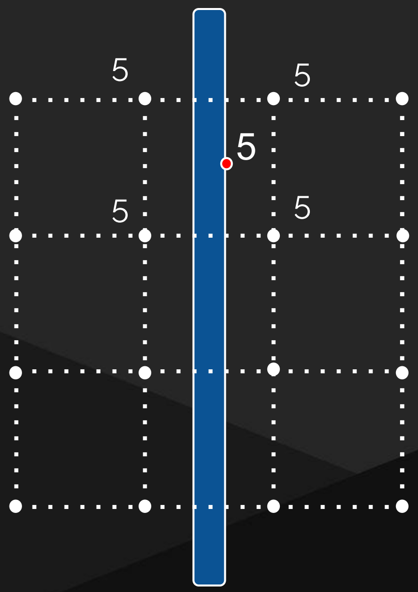
First one is that many meshes aren't closed. This often happens with scanned meshes or simply meshes which aren't supposed to be seen from the opposite side. It's not an issue for rasterizer, but in case of a distance field it produces a negative region which will stick out from the geometry and break tracing.

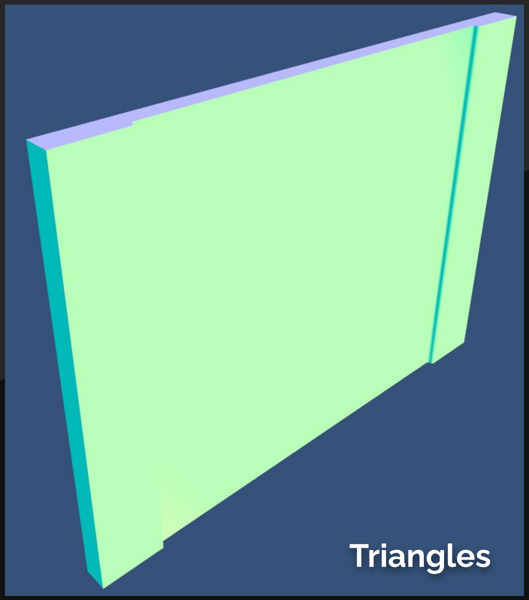
To solve this problem, during the distance field generation we insert a virtual surface after 4 voxels. Or in other words we wrap negative distance after 4 voxels.

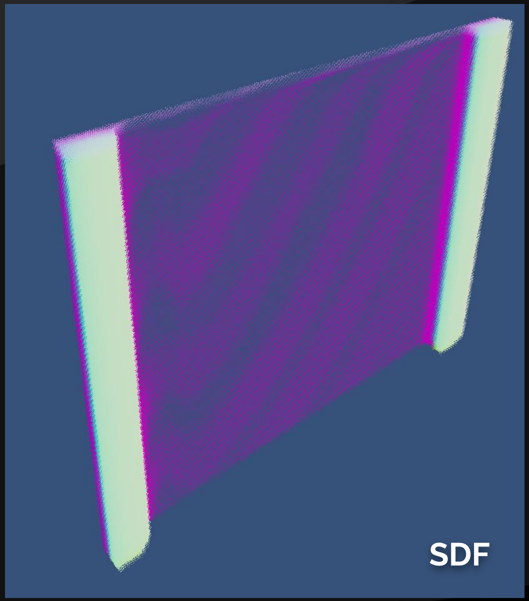
This is not a perfect solution, as it still causes mismatches between raster and ray marching, but it's better than having a huge negative region.

# Thin meshes in SDF

- May not ever cross zero
- Voxel size relative error







Another issue are thin meshes.

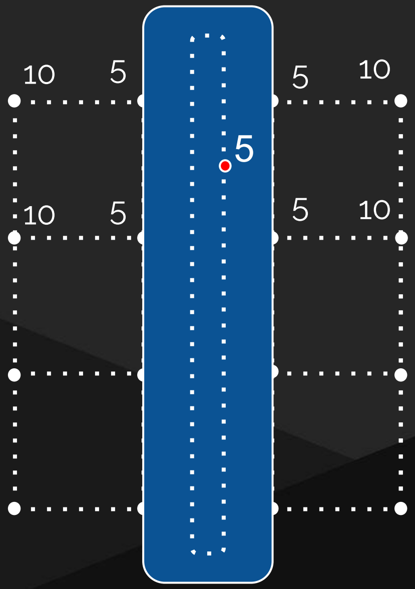
Discrete distance field representation is limited by the resolution and can't represent detail smaller than distance between two voxels.

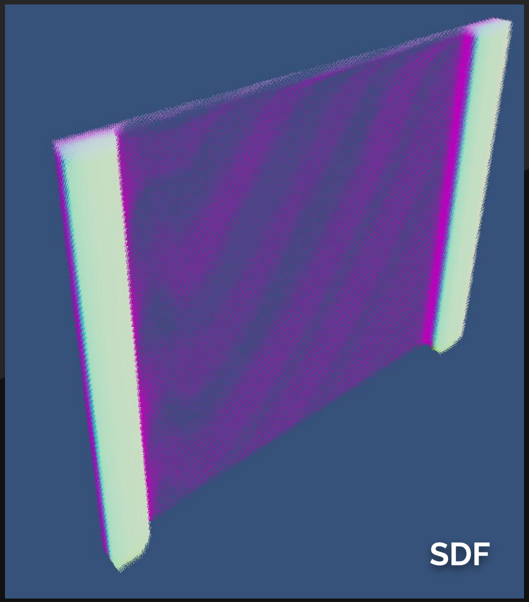
In this diagram you can see an example of a thin wall, which is placed between the sampling points. Evaluating such distance field won't ever result in a zero or negative distance and ray marcher won't ever register a hit. Gradient computation will be also incorrect as gradient around this wall will be zero.

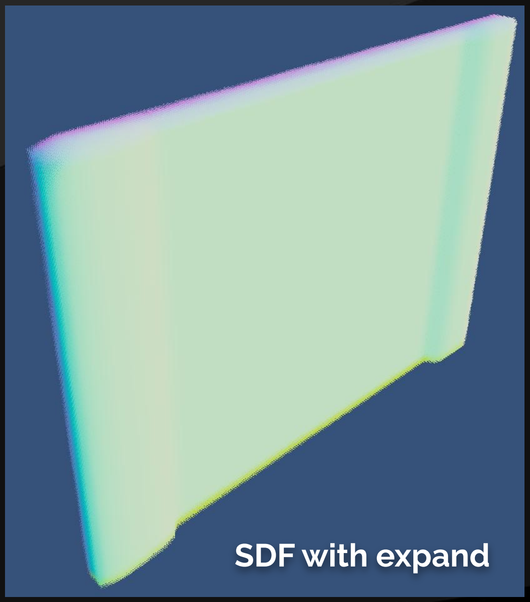
This can be a disaster in common scenarios, where we have dark indoors and bright outdoors and even a single ray going through a wall can cause massive light leaks.

# SDF Expand

- Half voxel expand to fix leaking
  - Pullback fixes gradient
- Lost contact shadows due to surface bias







To overcome this issue we need to expand out distance fields by a half of voxel’s diagonal.

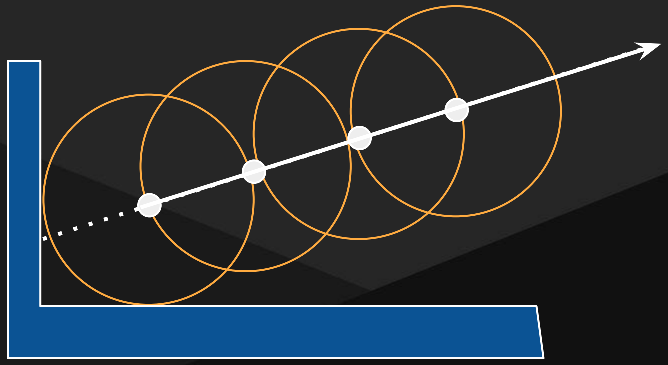
This expand fixes leaking and now we can reliably hit any thin surface. Gradient also will be fixed, as we will be computing it further away from the surface where we have reliable distance field values.

Expand is done at runtime, which allows us to preserve original distance field data.

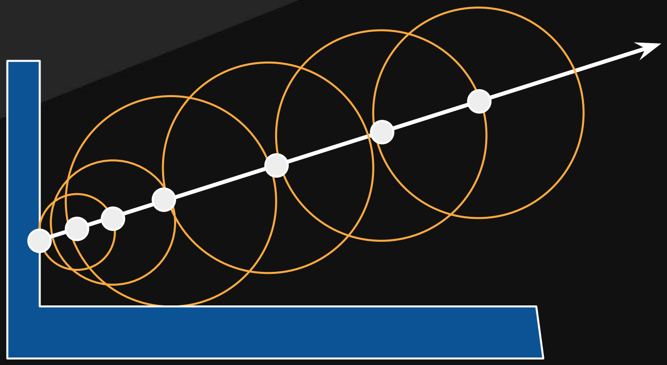
Downside with the expand is that it causes over-occlusion. Also we need a large surface bias to escape the surface, which breaks contact shadows.

# SDF Tracing Surface Bias

- How to expand surface without a huge bias?
- Start ray at t=0 without expand
- Linearly increase expand with distance
- Falloff expand at the end of a shadow ray



Surface bias

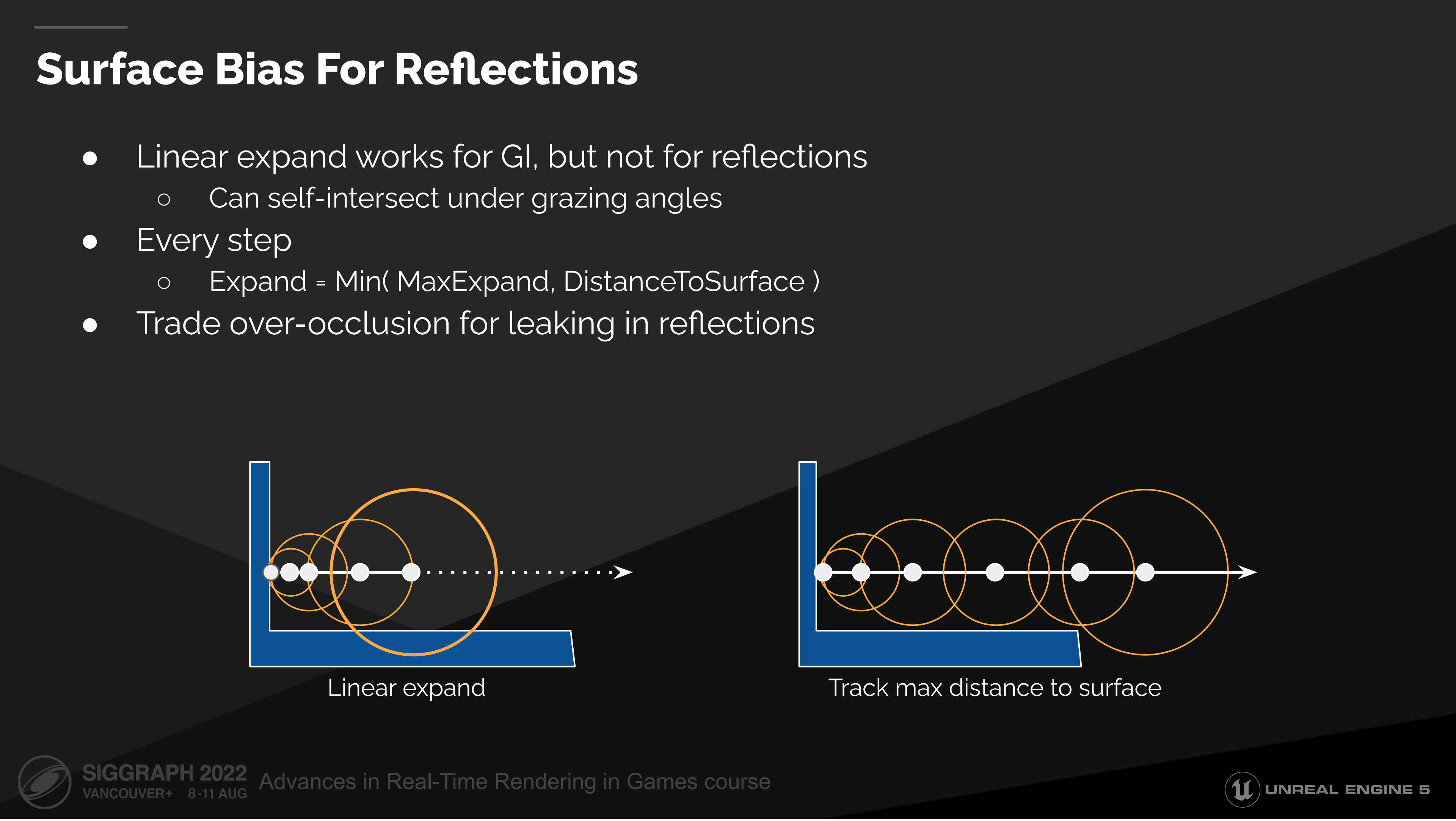


Linear expand

Let’s see how we can improve surface bias issue.

We preserve the original distance field data and expand surfaces at runtime, which allows us to start at the surface and then linearly increase expand as we move further away from it. This way we can trace that initial ray segment instead of just skipping it and losing all contact shadows.

In a similar fashion we need to falloff the expand back to 0 at the end of a shadow ray, so that it doesn’t hit any surface on which the light source is placed.



# Surface Bias For Reflections

- Linear expand works for GI, but not for reflections
  - Can self-intersect under grazing angles
- Every step
  - Expand = Min( MaxExpand, DistanceToSurface )
- Trade over-occlusion for leaking in reflections

Linear expand

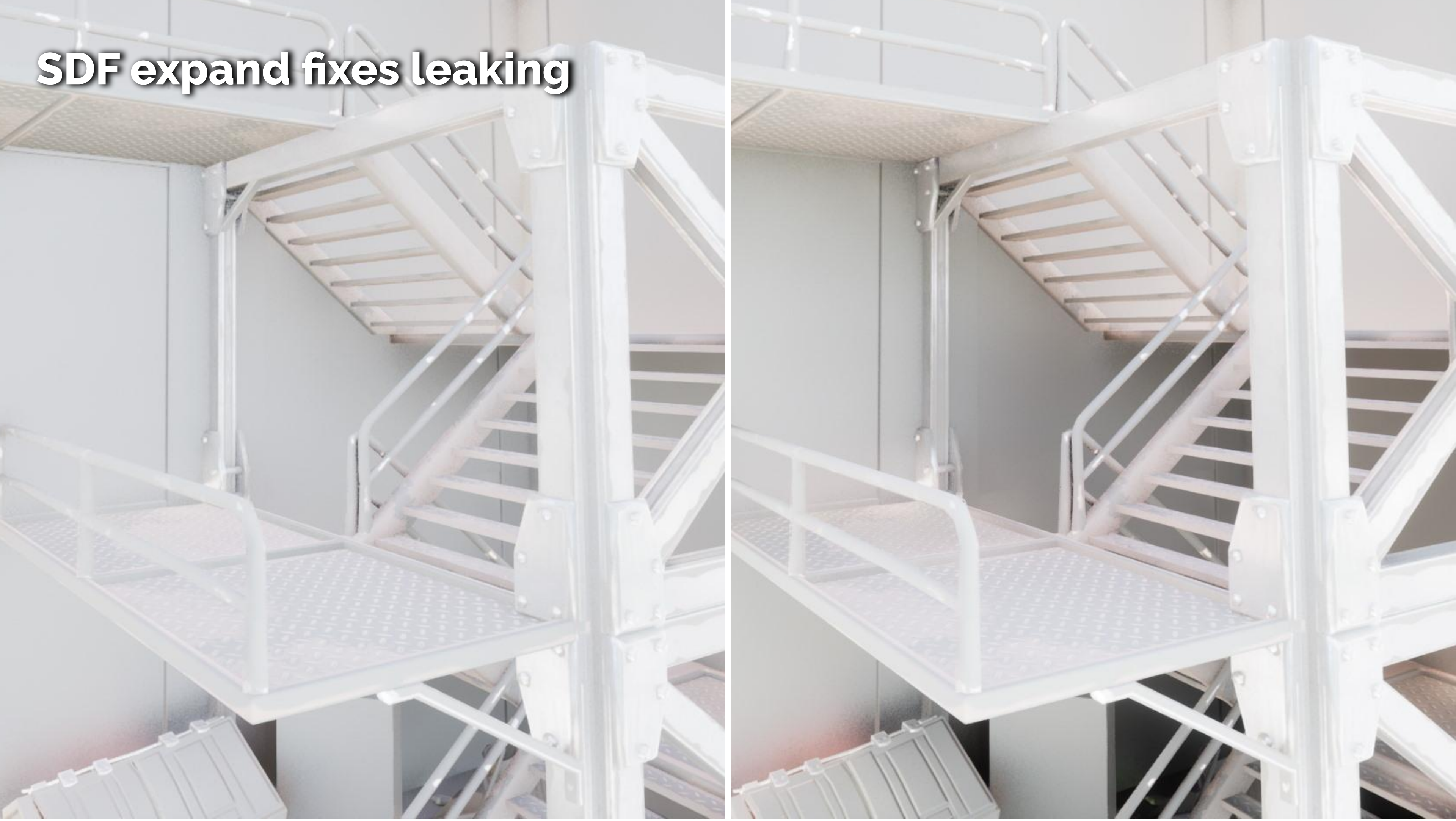
Track max distance to surface

Unfortunately this heuristics doesn’t work for glazing angles, as expand increases too fast and at some point ray will self-intersect.

This is fine for global illumination and diffuse rays, but it doesn’t look good in reflections, where we would prefer a bit of leaking instead of over-occluding.

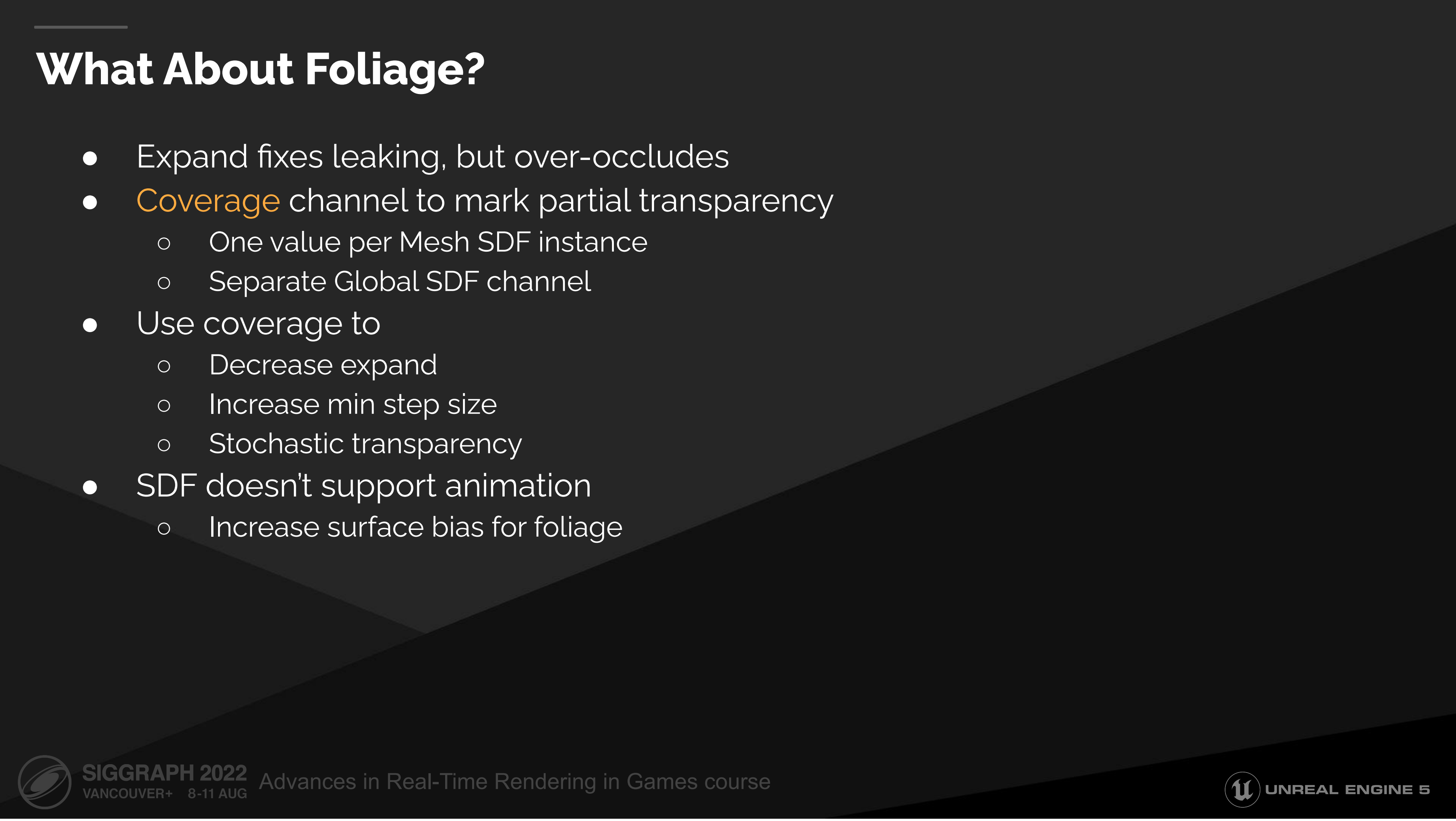
We solve this with our second heuristic for reflections. Every step we expand as much as possible based on the current distance to the surface. This makes sure that we will always escape the initial surface.

# SDF expand fixes leaking



Here’s the example of distance field expand in practice.

We manage to preserve contact shadows and still reliably hit thin surfaces.



# What About Foliage?

- Expand fixes leaking, but over-occludes
- Coverage channel to mark partial transparency
  - One value per Mesh SDF instance
  - Separate Global SDF channel
- Use coverage to
  - Decrease expand
  - Increase min step size
  - Stochastic transparency
- SDF doesn’t support animation
  - Increase surface bias for foliage

Distance field expand works great for solid geometry like walls, but it’s not great for foliage where it can completely prevent any light from passing between the leaves.

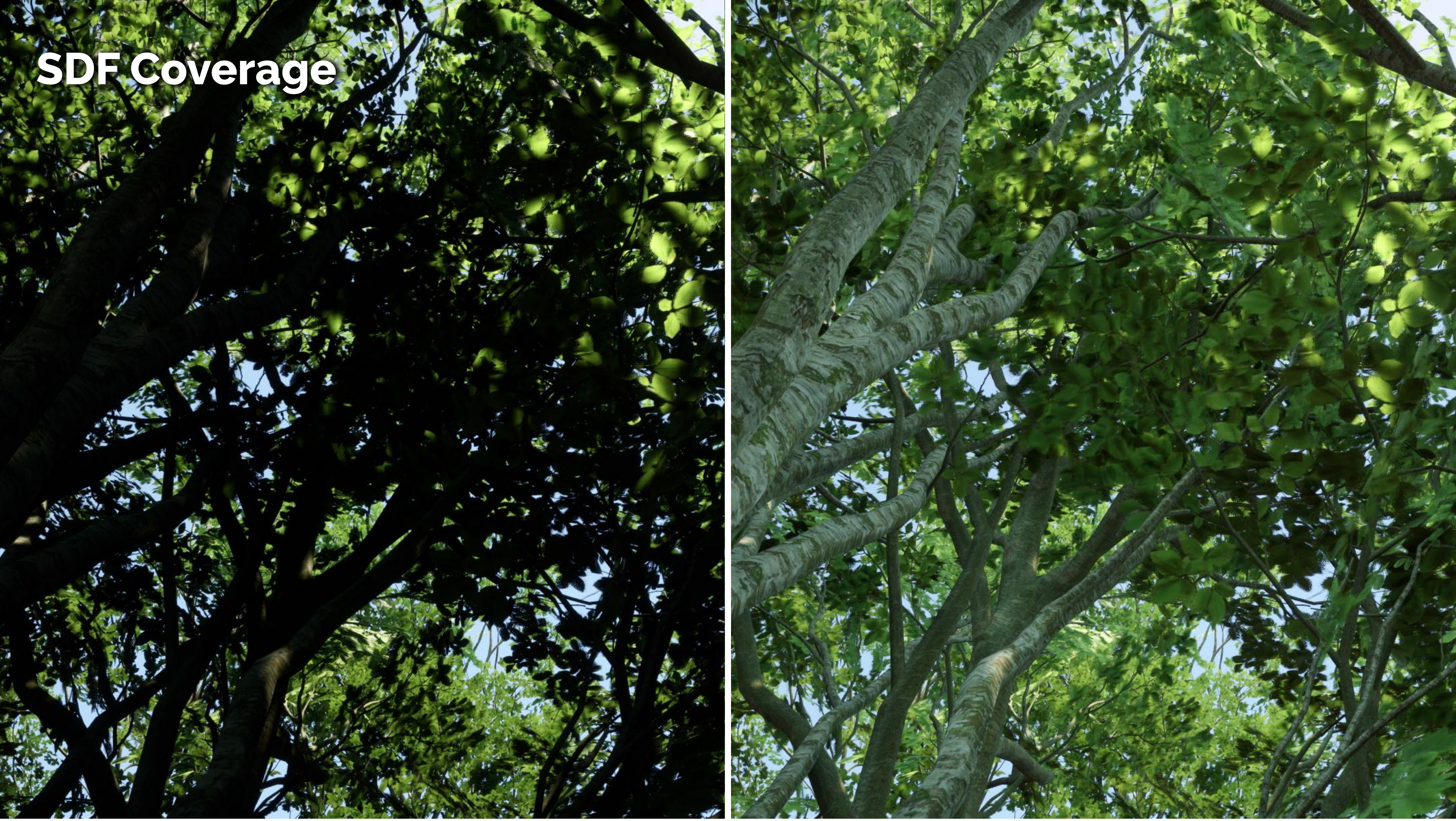
To solve this we had to introduce another heuristic, which we named coverage.

We mark distance field instances based on the two-sided material and then resample this data into a separate global distance field channel. Coverage allows us to distinguish solid thin surfaces which should block all the light, from surfaces with partial transparency, which should let some light pass through.

During ray marching every step we sample the coverage and based on it we increase raymarching step size and decrease expand. Additionally we use coverage for stochastic transparency on every hit to decide whether we should accept this hit or we should continue tracing.

Another issue with foliage is that it's usually animated and precomputed distance fields don't support animation. This is usually manifesting as self-shadowing issues and we fix them by adding an extra surface bias on foliage.

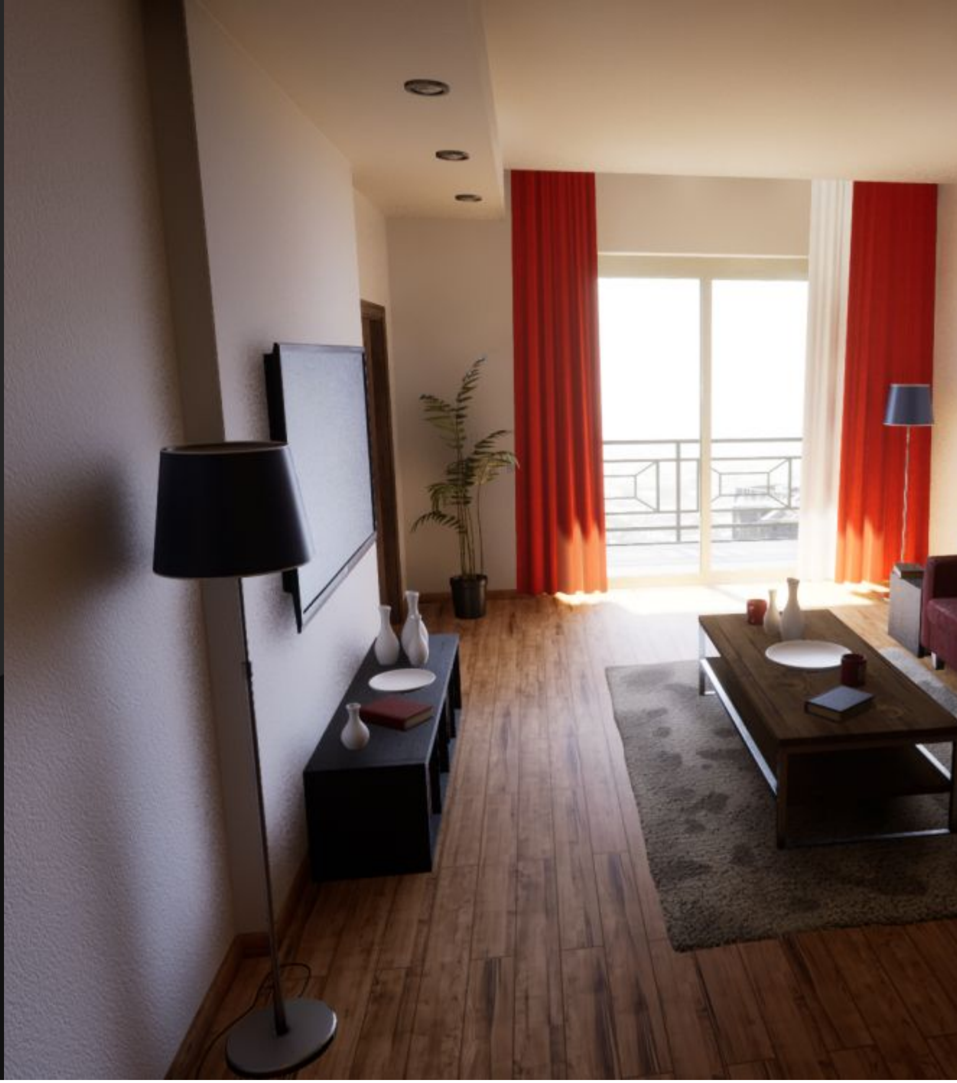
# SDF Coverage



With the coverage channel and extra bias trees don't block all the light anymore and some light can pass through the leaves and bounce between the trees.

# Software Ray Tracing Benefits

- GI and rough reflections
- Supported on all platforms
- Versatile utility
- Runtime LOD for complex scenes



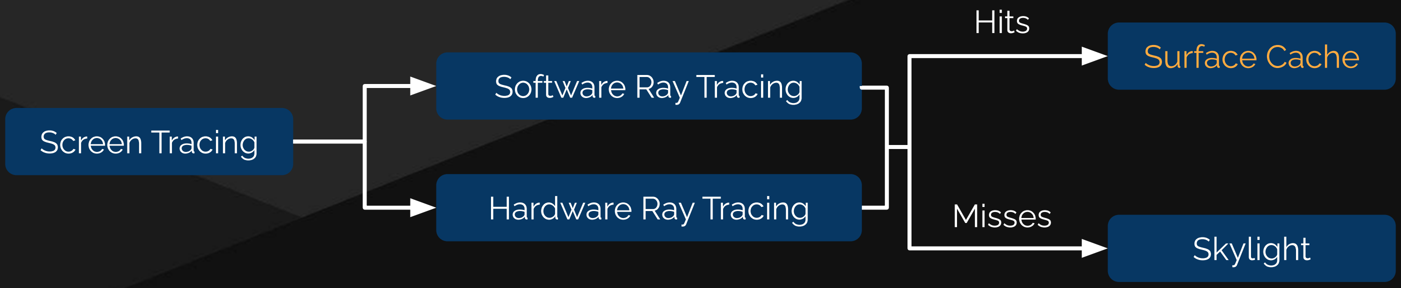
While distance fields aren’t perfect for mirror reflections, they are pretty good for GI and rough reflections. This scene on the right is almost fully lit by indirect lighting and as you can see distance fields just do a fine job here, resolving all the small details like indirect shadows from the lamp or TV on the wall.

Distance fields don’t require any special hardware and are supported on all platforms.

They are also a versatile engine utility, sharing scene setup work between Lumen and various other use cases like particles or physics collisions.

Finally distance fields allow us to scale down and support complex scenes with lots of instance overlap by merging entire scenes at runtime to a single global distance field.

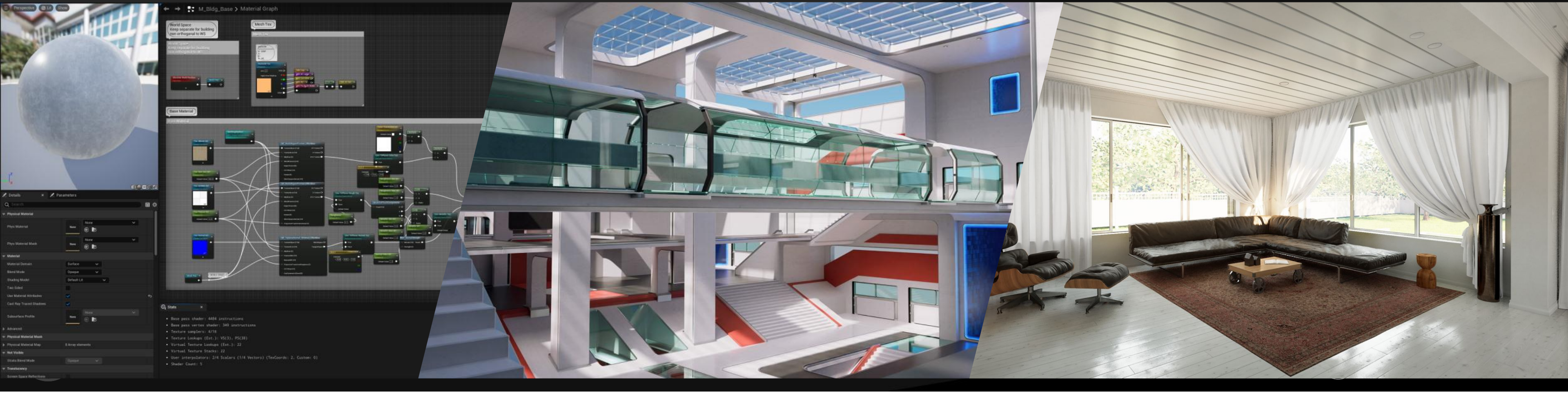
# Surface Cache



Now we need to address an elephant in the room. Namely how do we evaluate materials and lighting for distance field traces?

# Why Surface Cache?

- Evaluate SDF ray hits
  - SDFs have no UVs or other vertex attributes
- Cache expensive operations
  - Material graphs
  - Direct lighting
  - Multiple bounces



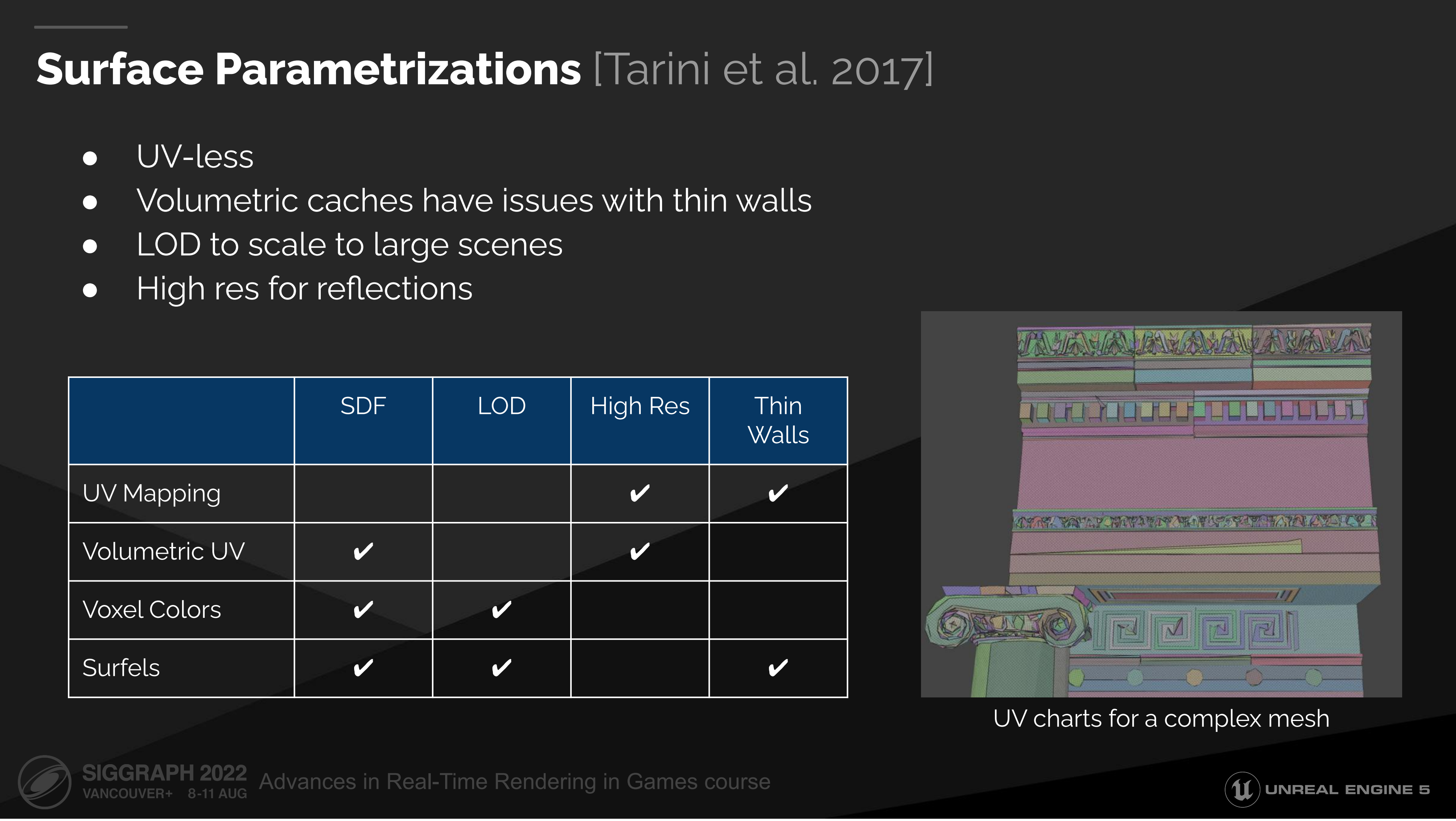
Distance fields don't have any vertex attributes and we can't run material shaders on them. We have access only to position, normal and mesh instance data. This means that we need some kind of a UV-less surface representation to be able to shade those hits.

We would like also to reuse this representation for caching various computations and ray paths as we can't afford recursive multi-bounce ray tracing.

Custom material graphs can be quite complex and costly to evaluate on every ray hit.

Direct lighting with multiple shadow casting lights can be expensive.

Multiple bounces are even more expensive, as now for every ray hit we need to recursively trace multiple rays and evaluate material and lighting for each one of them.



## Surface Parametrizations [Tarini et al. 2017]

- UV-less
- Volumetric caches have issues with thin walls
- LOD to scale to large scenes
- High res for reflections

| | SDF | LOD | High Res | Thin Walls |
|---|---|---|---|---|
| UV Mapping | | | ✔ | ✔ |
| Volumetric UV | ✔ | | ✔ | |
| Voxel Colors | ✔ | ✔ | | |
| Surfels | ✔ | ✔ | | ✔ |

UV charts for a complex mesh

Outside of being able to evaluate distance field hits, we also have other requirements for the surface parametrization. We would like it to be surface space based, as volumetric ones have issues with representing thin walls and can cause light leaks. We need a scalable solution, which can parametrize huge number of instances in complex scenes. Finally we would like it be able to scale up in resolution, so that we can also use it for reflections.

There are multiple surface parameterizations, but most of them didn’t work for us.

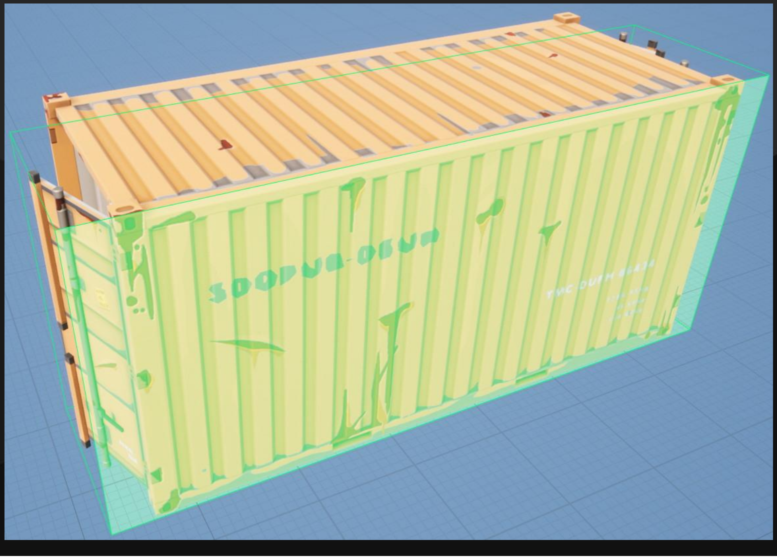
UVs are not a good fit for complex meshes as they can generate lots of UV charts which are non trivial to merge. They also require vertex attributes which we don’t have access to.

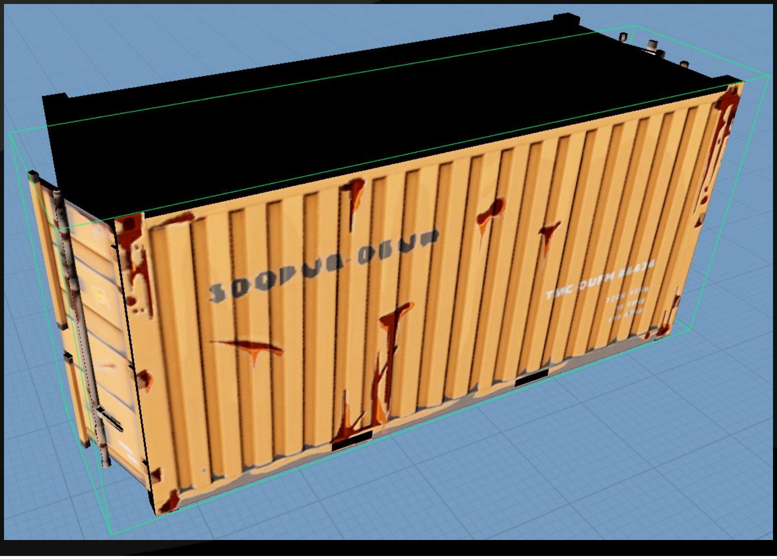
Volumetric UVs can’t represent thin walls and it’s unclear how to LOD those in the distance.

Voxel colors and surfels have a limited resolution and can’t be used for reflections.

## Cards

- Projected surfaces per mesh
  - Like clustered surfels
- Fast lookup
- Can capture at runtime
- Can represent two sided surfaces





For Lumen we decided to use projected cards, which can be also described as uniform rectangular clusters of surfels.

Cards are projection based, so we don’t need any vertex attributes to evaluate them.

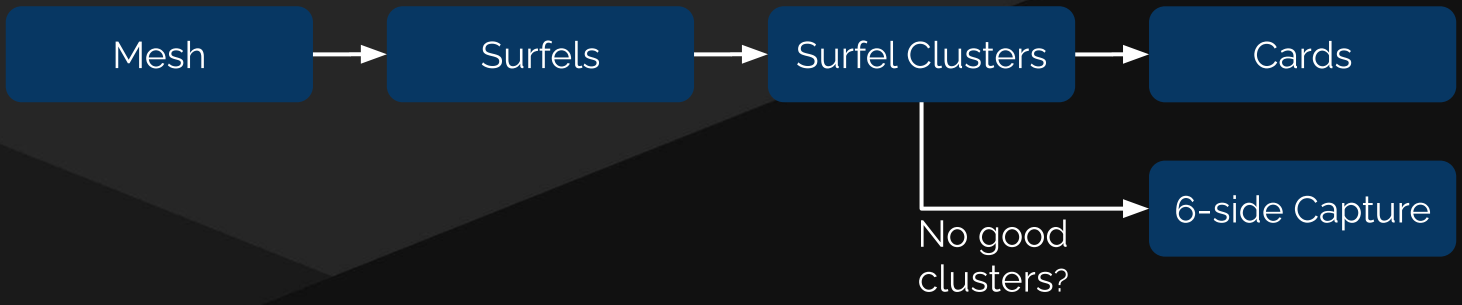
They are fast to lookup due to their regular structure.

Cards can be captured at runtime and scale to any resolution without having to bake anything.

They can also represent a two-sided thin wall.

# Card Generation

- Axis-aligned cards per mesh
- Generated at mesh import time
- ~0.2ms to build a large 1.5M tri mesh



Mesh → Surfels → Surfel Clusters → Cards

No good clusters? → 6-side Capture

In order to place cards on a mesh we run a precomputation step during mesh import.

All cards are axis-aligned, which simplifies generation and lookup. We tried freely oriented ones, but those were hard to place and extra flexibility wasn't really worth it in the end.

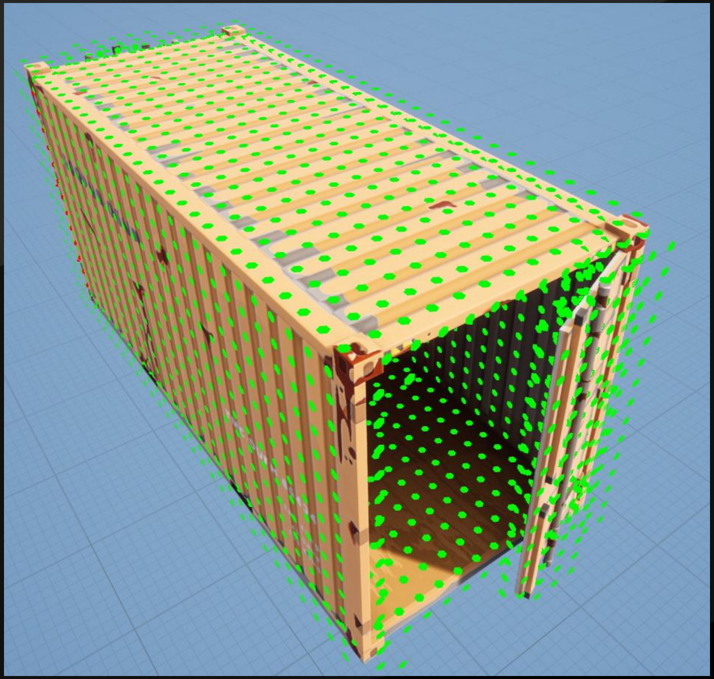
Generation starts from simplifying input triangle data by voxelizing it into a set of axis-aligned surfels.

Then we cluster resulting surfels using a K-means inspired clustering algorithm and resulting clusters are converted into cards.

If anything goes wrong during the generation. Like if mesh is too hard to unwrap or is too small, then we fallback to a 6 side cubemap like projection.

# Card Generation - Surfels

- Simplify triangle mesh
- Voxelize to axis aligned surfels
  - Trace 64 rays through object per 2d cell
  - Surfel coverage based on % of ray hits
  - Store previous ray hit for clustering
- Trace 64 rays over hemisphere from every surfel
  - Discard surfels inside geometry
    - If too many triangle back face hits
  - Surfel occlusion based on average distance to hits



First generation step is to convert triangle mesh into axis-aligned surfels in order to simply the mesh by removing small details.

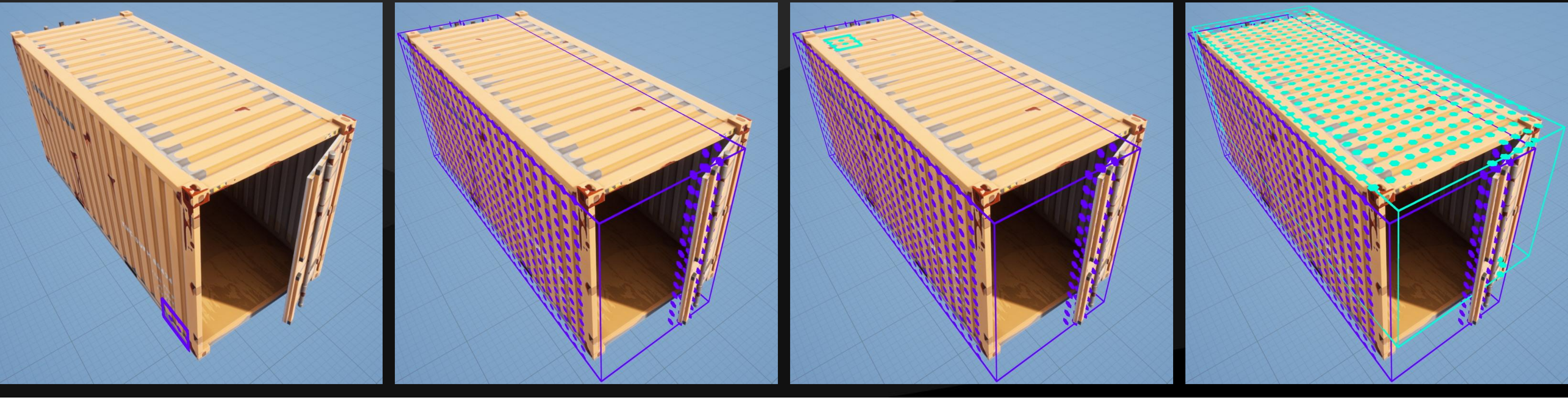
We voxelize mesh by casting 64 rays per 2d cell through the object. We sum ray hits for every 3d cell and spawn surfels above some threshold.

Number of hits is also stored as surfel's coverage, which will be later useful to evaluate how useful a given cluster is. Additionally we store previous ray hit position, which will be used to determine whether surfel is visible from cluster's near plane.

Next for every surfel we trace 64 rays and count the number of triangle back face hits. If most of those hits are back faces then given surfel is inside geometry and we can discard it. We also compute surfel's occlusion based on the average distance to hits. Occlusion will be used to determine how important it is to cluster a given surfel.

# Card Generation - Initial Clusters

- Pick random surfel as seed and iteratively grow cluster
  - Weight surfel candidates by
    - Distance from bounds and cluster aspect
    - Surfel occlusion and whether surfel is visible
  - Restart growing with centroid as new seed
  - Accept cluster if above coverage threshold
- Continue until all surfels have been tried or clustered



Next we generate our initial clusters.

We do this by picking an unused surfel and iteratively growing a cluster from it.

First we weight all unassigned surfels by distance to the cluster bounds to prefer closest ones.
We also weight by surfel occlusion to prefer the most important ones.
We weight by cluster ratio in order to prefer square clusters and finally we check whether a given surfel is visible from the cluster's near plane.

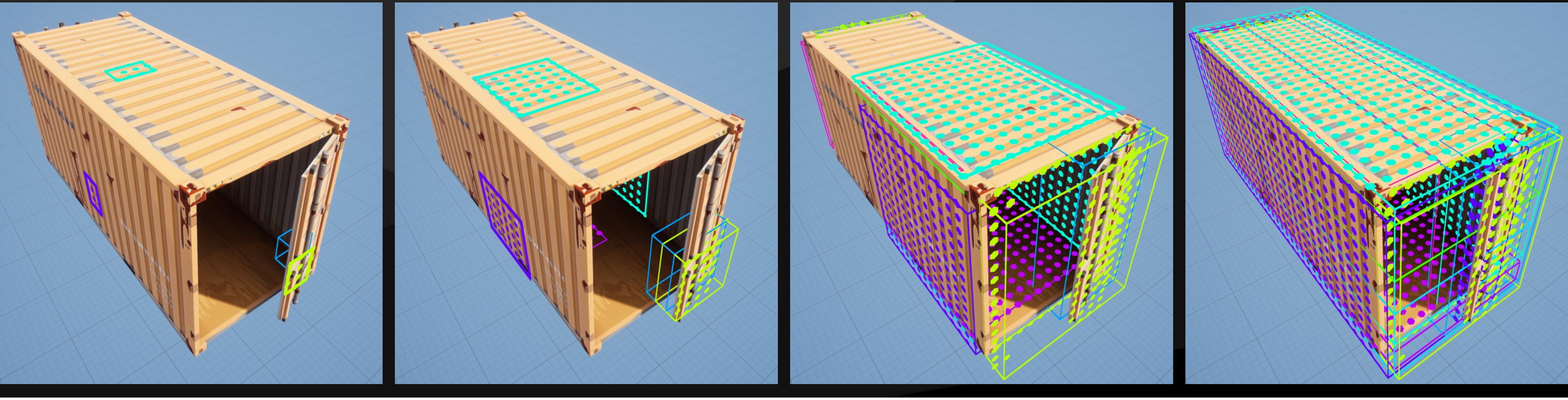
Next we proceed adding best surfels until we run out of valid surfel candidates. At this point we re-compute cluster's centroid and restart growing from it.

After hitting a limit of re-growths or when clusters don't change anymore, we add cluster to our list and try to find the next unused surfel.

After this step we have our mesh fully covered with clusters, but those clusters may not be globally optimal.

# Card Generation - Cluster Optimization

- Until clusters don't change anymore
  - Regrow all clusters in parallel from their centroids
  - Remove bad clusters
  - Insert new clusters
  - Use cluster centroids as new seeds
- Pick N clusters with highest coverage as cards



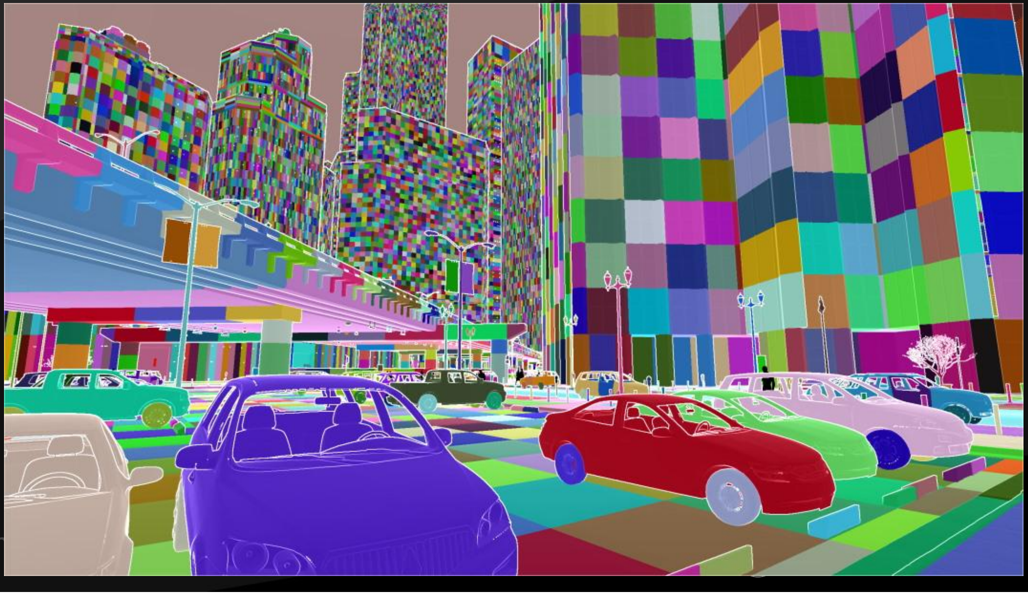
Final step is a global optimization, where we re-grow all clusters in parallel from their current centroids. Again we do a few iterations of parallel growing until we hit a limit or clusters don't change anymore.

Parallel growing may cause too small clusters or empty space, so after every iteration we need to remove too small clusters and try to insert new ones in empty space.

Finally we sort our clusters by coverage and pick user selected number of most important ones to convert them into cards.

# Card Management

- Two conflicting needs
- Many low res cards for GI
  - Cover around the camera for multiple GI bounces
- Few high res cards for reflections
  - Cover specific surface parts seen in reflections
- "The Matrix Awakens"
  - 1.5M loaded instances
  - Plenty of reflective surfaces



Now we have cards per mesh and need to somehow manage them in the scene.

There are two conflicting needs here.

On one hand we want to have many small cards for handling multiple GI bounces. We don't need a lot of resolution here, as GI is rather low frequency, but it's important to cover everything consistently.

On the other hand we want a few high resolution cards on selected surfaces for reflections. Here we need lots of resolution, as in case of mirror reflections, cards need to match screen pixel density.

Extreme case of this was our latest Matrix Awakens demo, where we had lots of instances to cover around the camera to handle light bouncing between the buildings. And at the same time we had multiple reflective surfaces which required high resolution cards.

# Virtual Surface Cache

- Low res always resident pages
  - Track objects around camera
  - Allocate based on distance and size
  - Cull cards below some threshold
- On demand high res pages
  - Only allocate pages requested by GPU feedback
  - Heap to track when page was last used
  - Deallocate oldest when out of memory

Those two requirements lead us to a virtual surface cache.

For GI we use low resolution always resident pages. Those pages are allocated around the camera based on the distance. We also have an LOD scheme in a form of removing too small cards.

For reflections we use sparse and optional on demand pages. Those are allocated based on the reflection ray hits and deallocated when they no longer in use.

# Surface Cache GPU Feedback

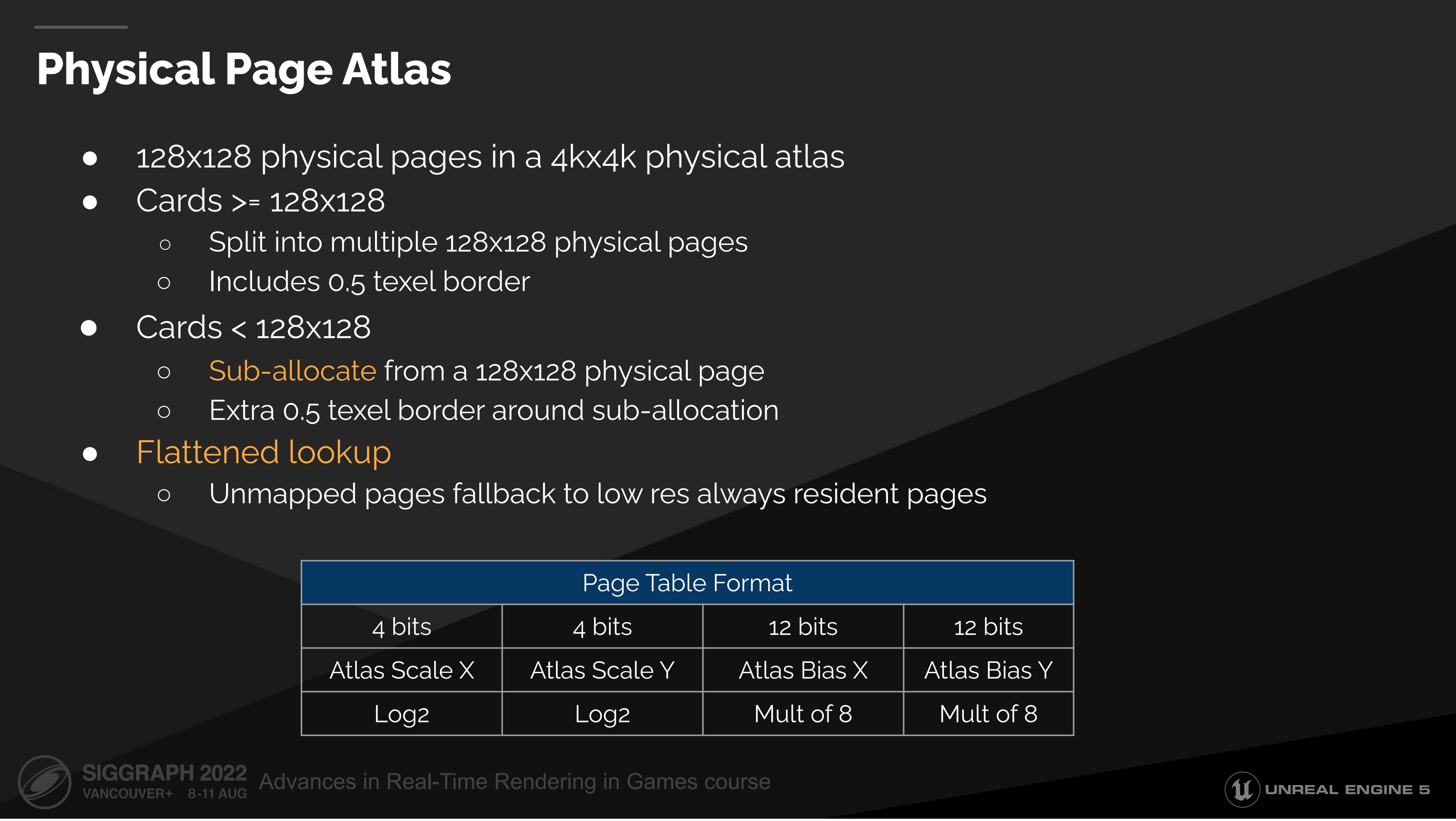
- On ray hit
  - Stochastically select one sample per ray
  - Update last used time for page
  - Write requested mip to feedback buffer
- Merge feedback buffer on GPU
  - Insert into a GPU hash table
  - Compact hash table into an array
- Download merged requests to CPU
- Map and unmap pages

On every ray hit we write surface cache feedback, so that we can optimally select resolution and update rate for every page.

Ray hit samples and blends multiple pages, so first we need to stochastically select the most important one. Next we update its last used time and write requested mip level to a feedback buffer.

We compact this feedback buffer on the GPU by inserting all request into a GPU hash table and then compacting it. Final array of requests contains unique pages with number of hits per page.

This request array is later downloaded to the CPU, where we can sort it and use it to map and unmap pages.



# Physical Page Atlas

- 128x128 physical pages in a 4kx4k physical atlas
- Cards >= 128x128
  - Split into multiple 128x128 physical pages
  - Includes 0.5 texel border
- Cards < 128x128
  - Sub-allocate from a 128x128 physical page
  - Extra 0.5 texel border around sub-allocation
- Flattened lookup
  - Unmapped pages fallback to low res always resident pages

| Page Table Format | | | |
|---|---|---|---|
| 4 bits | 4 bits | 12 bits | 12 bits |
| Atlas Scale X | Atlas Scale Y | Atlas Bias X | Atlas Bias Y |
| Log2 | Log2 | Mult of 8 | Mult of 8 |

Cards larger than the page size are split into physical pages and allocated separately.

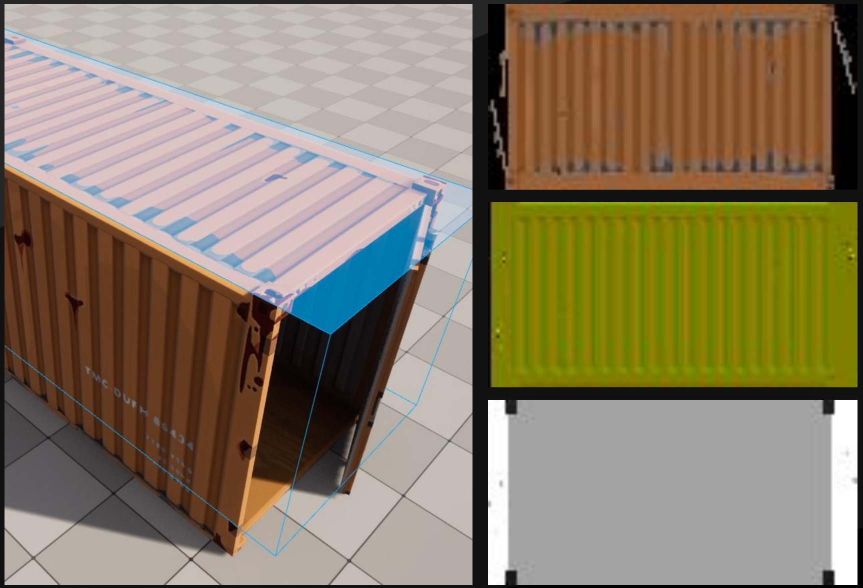
Cards smaller than page size are sub-allocated. This means that we map a single physical page and allocate multiple smaller cards from it using a 2d allocator.

This is important, as it allows us have large pages which don’t waste much memory on borders and at the same time we can support small allocations which don’t need to be rounded up to the physical page size.

We flatten our lookups, so that if we request a missing high res page, then page table will automatically point to a lower resolution always resident one. This allows us to sample surface cache in a single lookup, without having to recursively search for the fallback page.

# Card Capture

- Resample mesh and material data at runtime
  - Render single mesh with an ortho camera
- Fixed texel budget per frame (512x512)
  - Prioritize by distance and GPU feedback
- Slowly update by recapturing oldest pages
- x10 faster with Nanite [Karis et al. 2021]
  - Single vis buffer draw for all cards
  - One dispatch per material
  - Optimal LOD level

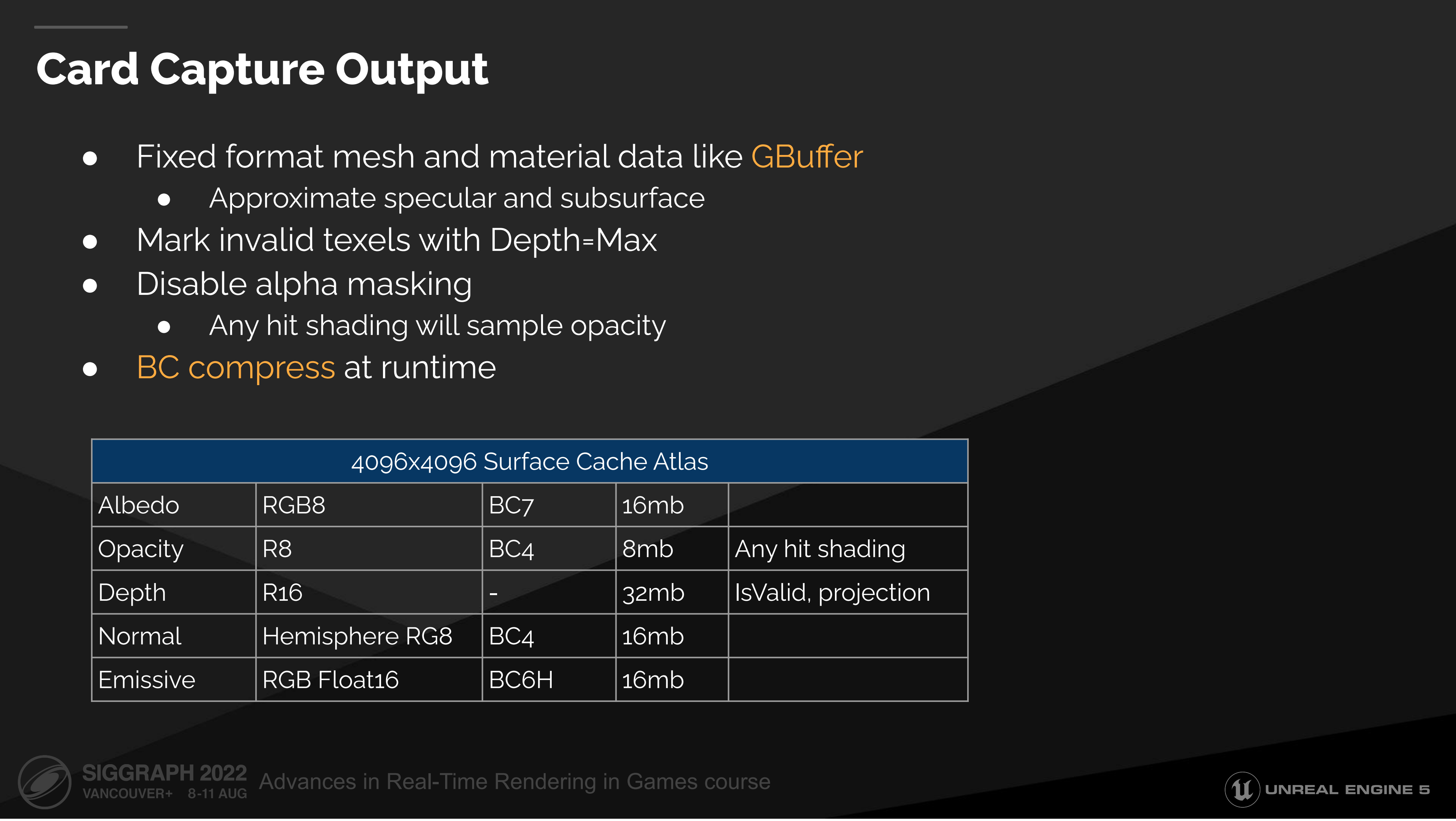


We also need to populate cards with mesh and material data, which later will be projected onto surfaces.

We do this at runtime by rendering meshes into cards with an ortho camera and writing surface properties like albedo or normals. Doing this at runtime is convenient as it allows us to scale up in resolution and support material changes without having to manage any precomputed data.

Card capture is cached and updated using a fixed budget. Every frame we gather page update requests and sort them by distance from the camera and when they were last used. Then we pick a fixed number of most important pages to update and capture them. Additionally in order to support animated materials we also update a small number of oldest pages per frame.

Traditionally rendering many small meshes would be slow due to LOD and many small draw calls. With Nanite we can render all geometry in a single draw call and we have a continuous LOD level to simplify meshes rendered into those tiny targets. This greatly speeds up rendering and allows us to capture cards more often.



# Card Capture Output

- Fixed format mesh and material data like GBuffer
  - Approximate specular and subsurface
- Mark invalid texels with Depth=Max
- Disable alpha masking
  - Any hit shading will sample opacity
- BC compress at runtime

| 4096x4096 Surface Cache Atlas | | | | |
|---|---|---|---|---|
| Albedo | RGB8 | BC7 | 16mb | |
| Opacity | R8 | BC4 | 8mb | Any hit shading |
| Depth | R16 | - | 32mb | IsValid, projection |
| Normal | Hemisphere RG8 | BC4 | 16mb | |
| Emissive | RGB Float16 | BC6H | 16mb | |

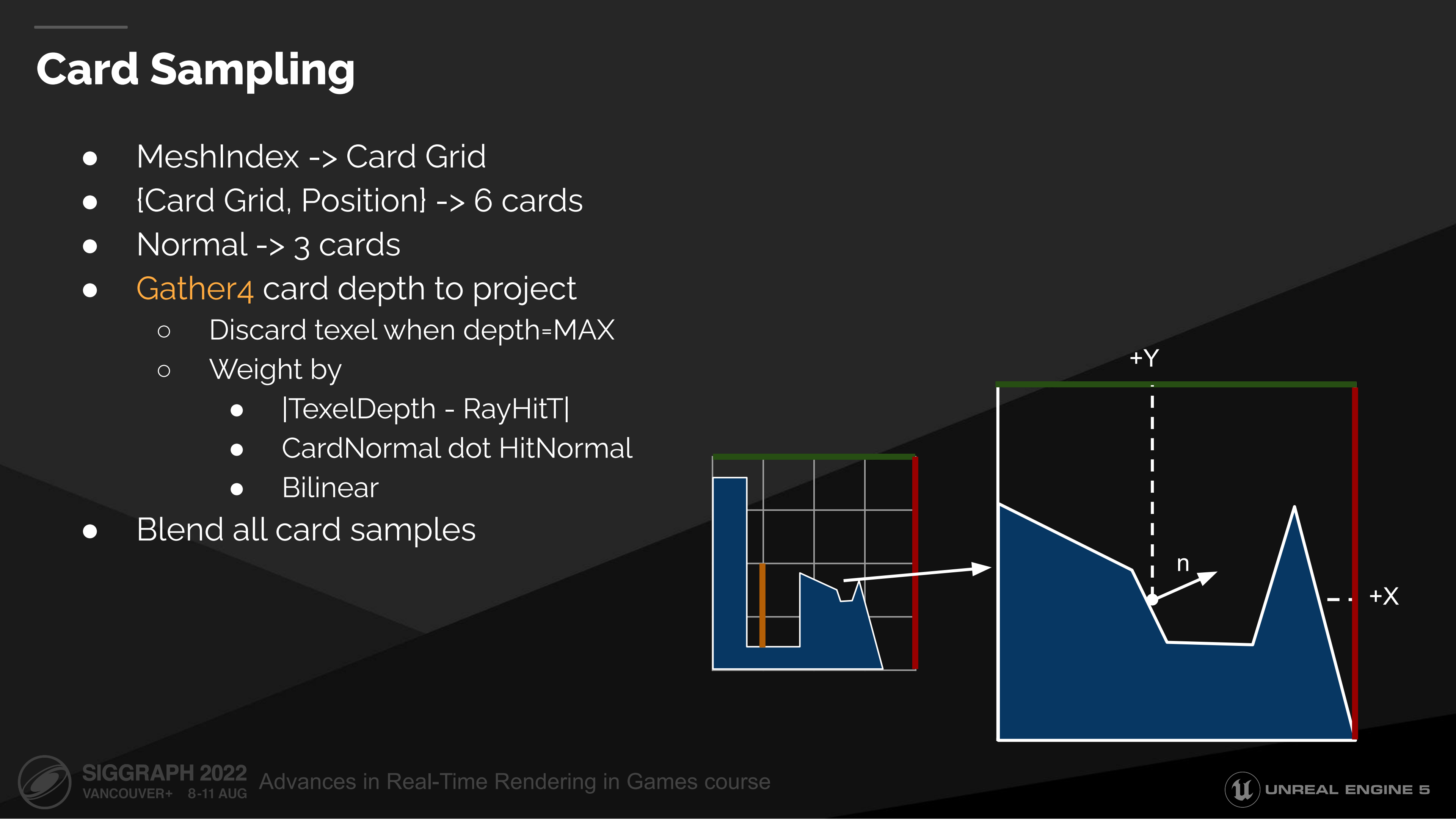
Card capture resamples material and mesh data into a fixed and view independent GBuffer like structure.

We approximate specular and subsurface response by modifying albedo to account for the lost energy.

We also mark invalid texels, so that later we know which texels don’t contain valid data and can’t be sampled.

During capture we disable alpha masking, as we need to distinguish lack of surface cache data from an alpha masked surface point. It will be useful later for running any hit shader without having to deal with material shaders.

Finally captured surface cache data is BC compressed at runtime in order to minimize the resulting memory overhead.



## Card Sampling

- MeshIndex -> Card Grid
- {Card Grid, Position} -> 6 cards
- Normal -> 3 cards
- Gather4 card depth to project
  - Discard texel when depth=MAX
  - Weight by
    - |TexelDepth - RayHitT|
    - CardNormal dot HitNormal
    - Bilinear
- Blend all card samples

Now we have all the data ready and we can sample the surface cache.

We start from looking up the card grid based on the mesh index. Then we lookup a cell in that grid to get 6 cards. Based on the surface normal we pick 3 cards to project.

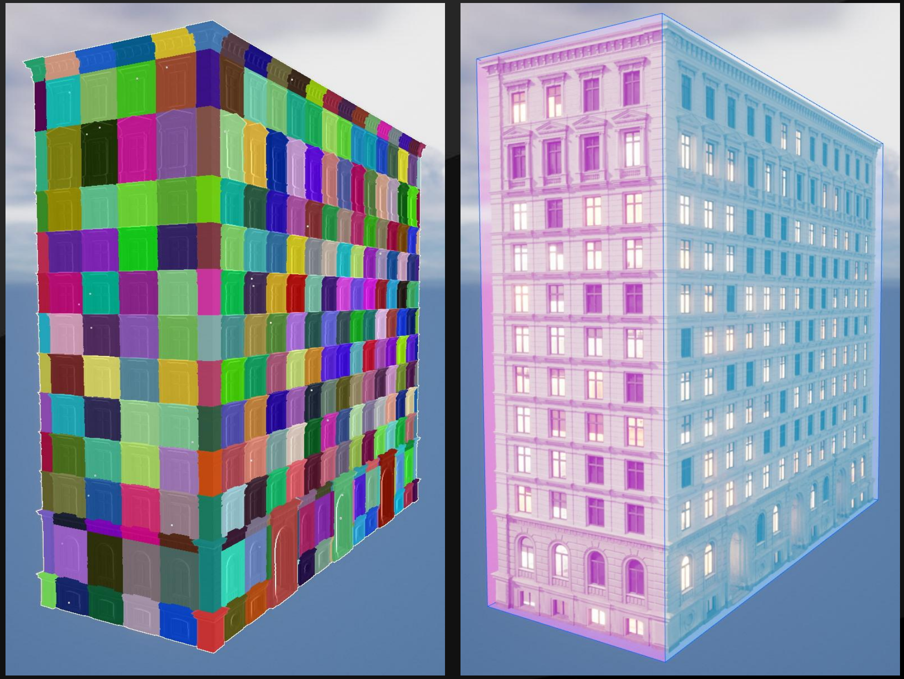
Then we sample all 3 cards. For every card we gather 4 depths from surface cache in order to do manual bilinear filtering.

We weight each texel by delta between stored depth in surface cache and ray hit depth to discard occluded samples. We also weight texels by card projection normal to prevent projection from stretching. Then we discard texels marked as invalid.

Finally all samples are blended together to compute the final surface cache properties at the hit point.

# Card Merging

- Too many small instances
  - Buildings in the distance
  - Scattered pebbles
- Merge cards for LOD
  - Group small instances
  - Group based on tags
  - 6 card cubemap like capture
- Fast capture with Nanite
  - Render entire building into card



For some content we were running into limits on how far we can scale cards. This was an issue with many small instances aggregating into larger objects, like this building here, where we would have either to spawn too many small cards or to drop the entire object.

Our solution to this was to merge cards at runtime.

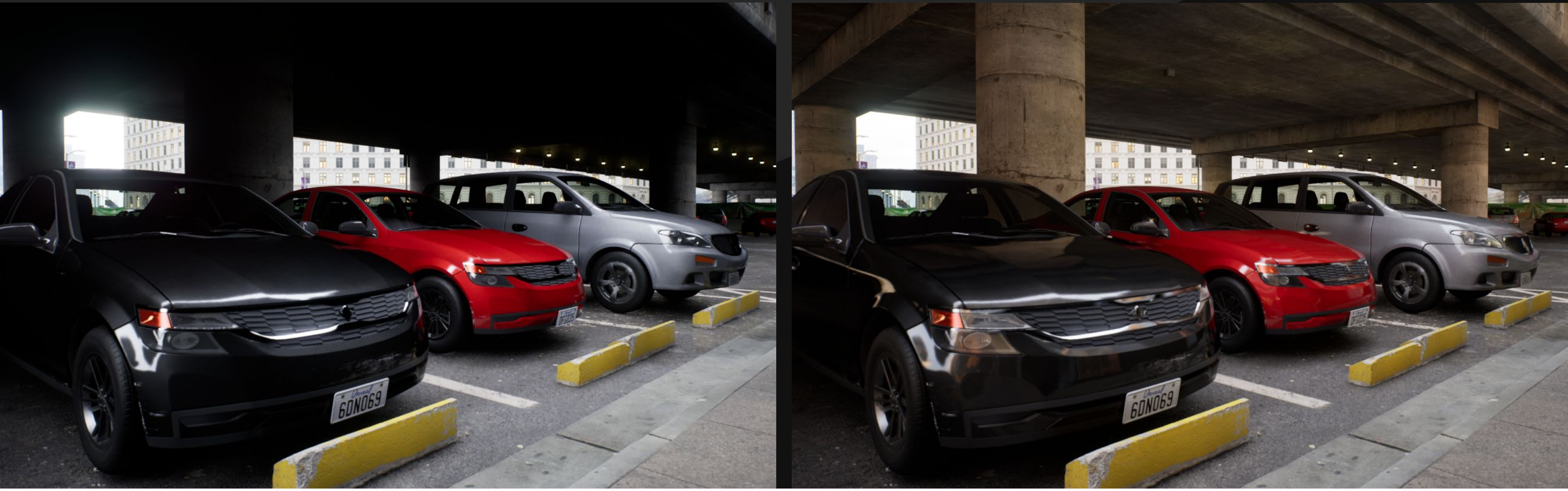
We do this either by automatically finding groups of small overlapping instances or by using user supplied group tags.

Each such group gets 6 cards to capture it from 6 sides, like a cubemap. This is a pretty good approximation assuming that the viewer is outside of that aggregated group, which is usually the case.

Finally cards are captured by rendering an entire group into each card, which again is pretty fast thanks for Nanite.

# What About Lighting?

- Material is now in surface cache
- How to compute lighting on hit?
  - Shadow rays are expensive
  - Multiple GI bounces even more
  - Can’t afford longer paths than 1 trace to surface



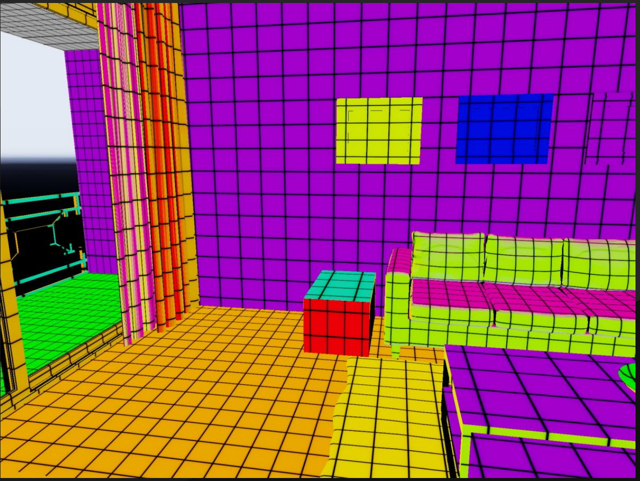
Now we have the material data in the surface cache, but still somehow need to compute lighting.

Here in this screenshot you can see how important multiple bounces are. Without them half of the scene is black and reflections disappear.

Direct lighting with multiple shadow rays or indirect with recursive tracing is quite expensive. In most cases we can’t really afford even a single extra ray and ideally all lighting would just come from the surface cache.

# Surface Cache Lighting

- Cache direct and indirect lighting (n+2 bounce)
- Tracing from texels
  - Surface bias
- Traces can start inside the geometry
  - Rays that hit triangle back faces return zero radiance



Surface Cache Mapping

Direct Lighting

Indirect Lighting

Surface cache contains all mesh and material data and we can use it to evaluate and cache lighting.

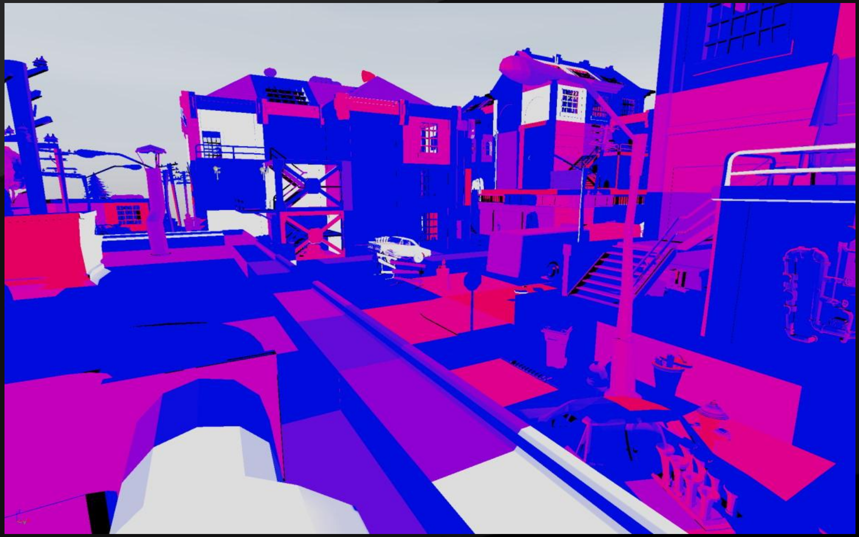
This is similar to lightmapping and we also run into similar issues.

When tracing from texels, we need an appropriate bias based on the surface normal and ray direction to escape surfaces.

Texels may also be inside geometry and they can cause leaking through the walls due to bilinear filtering. We fix this by discarding rays which hit triangle back faces, effectively making texels inside geometry black.

# Update Strategy

- Fixed budget
  - Select pages to update based on priority
  - Priority = LastUsed - LastUpdated
- Priority queue using radix sort
  - Build histogram
  - Update buckets until hitting budget
  - 1024x1024 texels for direct lighting
  - 512x512 texels for indirect lighting
- Resample previous lighting if available



Lighting is too expensive to evaluate every frame, so we use caching and update only a subset of surface cache every frame.

We select pages to update based on when pages were last used and last updated.

Last used is based on the GPU feedback writing current frame number on every ray hit. Last updated is incremented on every page update.

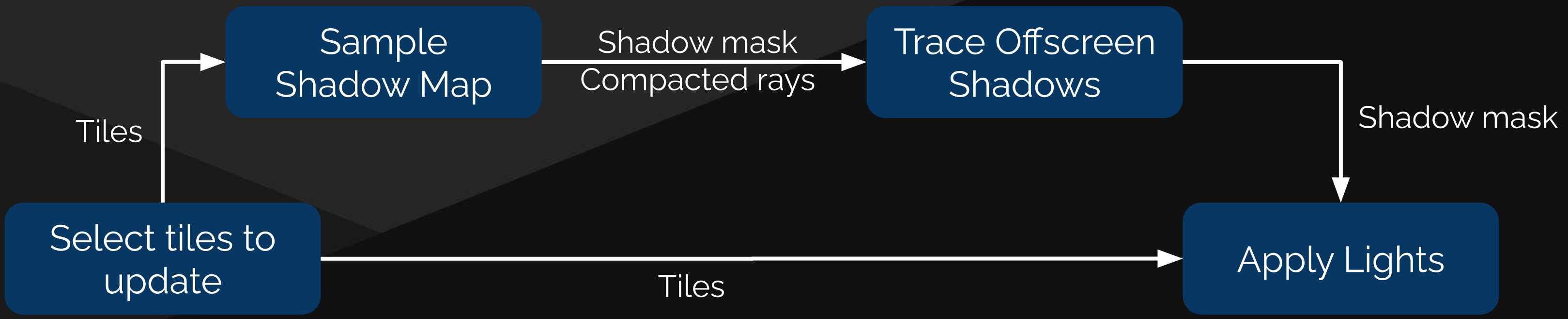
Those two properties are tracked separately for direct and indirect lighting, as they both have different update ratios. Specifically we update cheaper direct lighting much faster than indirect.

In order to select a fixed number of most important pages, we build a histogram and select items from consecutive buckets until we hit our budget.

Finally sometimes we need to resize a card or map a new page. In that case we try to resample previous lighting if it's available, so that we don't discard all those expensive calculations.

# Direct Lighting

- Cull lights to 8x8 tiles in z-order
- Select first 8 lights per tile
- 1 bit shadow mask
- Output compacted rays for tracing



Sample Shadow Map

Trace Offscreen Shadows

Select tiles to update

Apply Lights

After selecting pages to update we splice them into 8x8 tiles. We output those tiles in a z-order to maximize trace coherency.

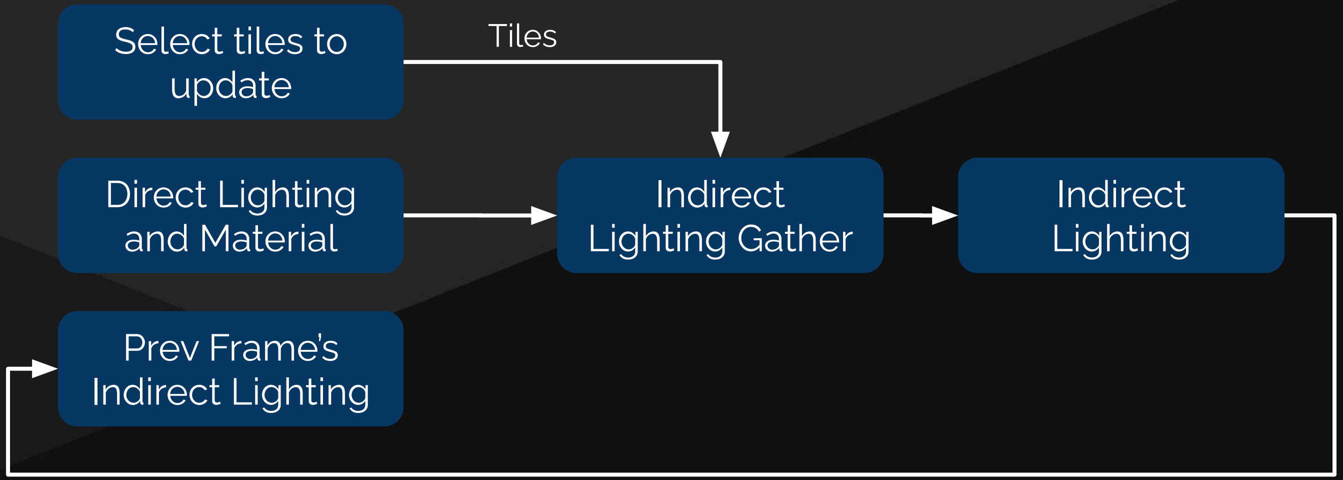
Then for every tile we select up to 8 lights. Currently we just pick the first 8 lights and it works fine for our use cases, but in future we would like to use a smarter light selection strategy.

For every light we have 1 bit in a shadow mask which will be used to composite multiple shadow methods. We first populate this shadow mask by sampling available shadow maps. During this pass we also build a compacted list of shadow rays, which couldn't be resolved by shadow maps and will need to be traced. Usually those are texels located behind the camera. Next we trace shadow rays to finish the shadow mask.

Finally we run our light pass and we compute lighting values using this shadow mask.

# Indirect Lighting

- Like final gather, but 2nd bounce in surface space
- N+2 bounces through feedback
- Below 1/16th of Lumen's budget



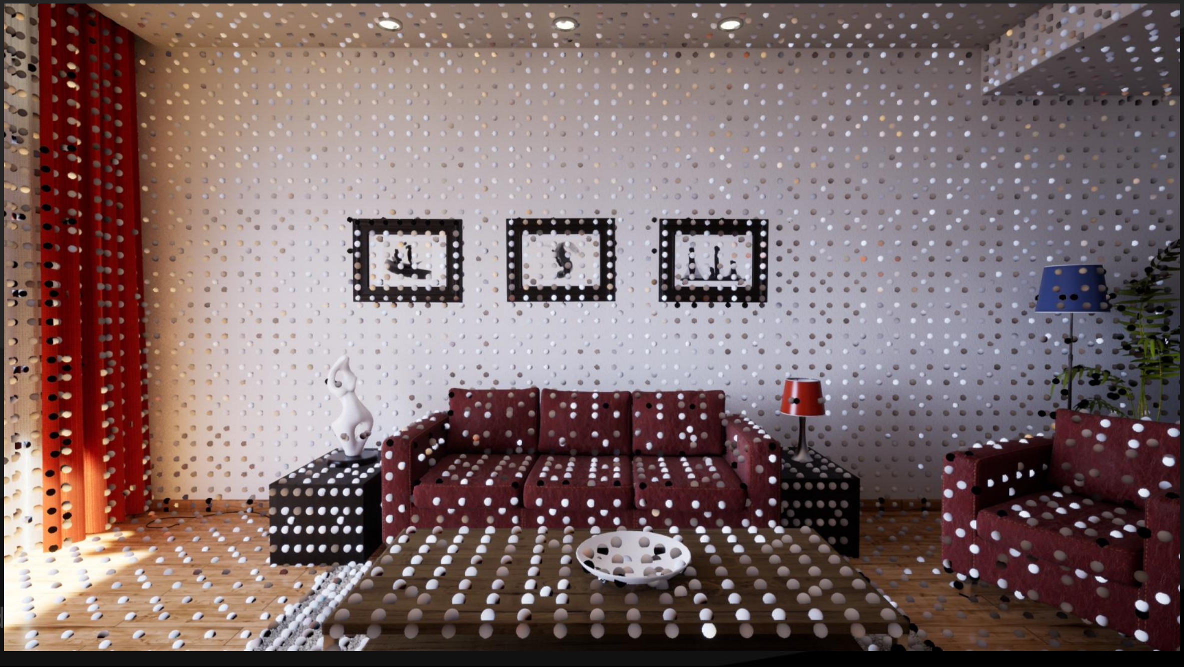
Indirect lighting is more challenging as here we basically need to run the final gather in surface space to compute the secondary bounce.

In order to support multiple bounces for every indirect ray hit we sample current frame's direct lighting and last frame's indirect lighting. So for every frame we compute the first two bounces and the following bounces are then feedback based.

One important callout is that we have a very limited budget and we trade off quality for performance here. This means that not only do we select a low number of pages to update, but also that we have a quite limited ray budget.

# Indirect Lighting Probes

- 4x4 hemispherical probe per 4x4 tile
- 4 frame temporal jitter
  - Jitter probe placement and ray directions
  - Frame index per page



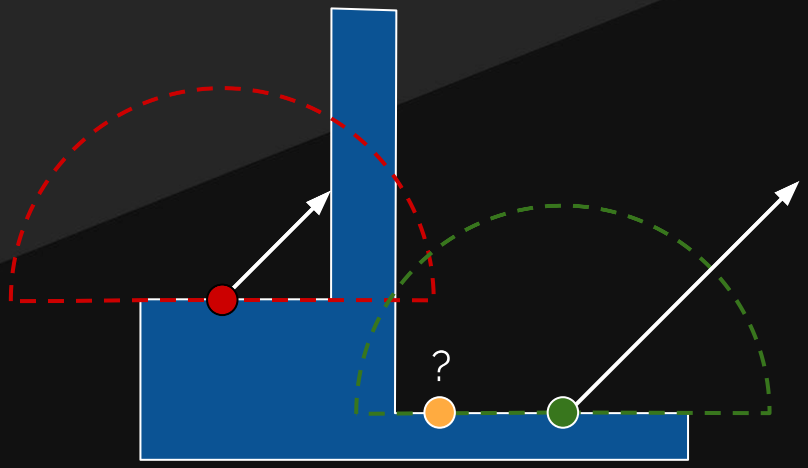
Ideally we would trace at least 64 rays from every texel, but that’s just too expensive. Instead we place a hemispherical probe on 4x4 tile and trace only from probe texels. This enables downsampled tracing and still preserves surface normal detail.

We jitter probe locations and probe trace directions every frame based on the frame index, which is stored per surface cache page.

Resulting probes are quite noisy due to a low number of traces, so we will also need some spatial and temporal reuse to clean that up.

# Indirect Lighting Gather

- Bilinear interpolation of 4 probes
  - Plane weighting
  - Visibility weighting using probe hitT
- 4 frame temporal accumulation to atlas
  - Accumulated frame num per texel



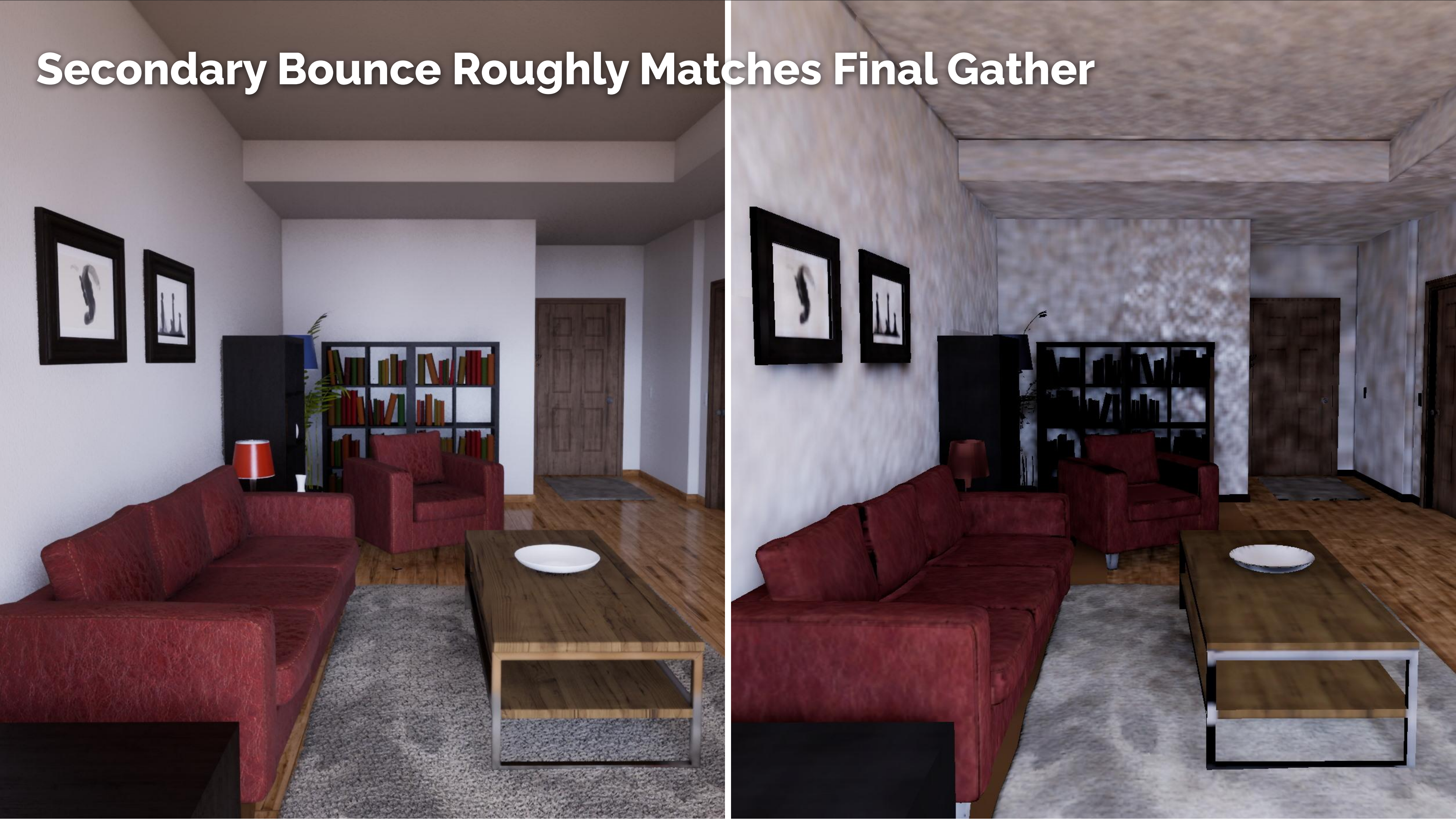
Evaluation point is behind the red probe
Also red probe doesn't see the evaluation point

For every texel we take 4 closest probes and then interpolate between them to compute indirect lighting.

During interpolation we have two heuristics to minimize leaking, as some of the probes could be behind a wall. First heuristic to weight every hemispherical probe by its plane in order to skip texels behind it. Second one is to use the probe's depth map to check visibility between probe and evaluated texel.

Finally interpolated results are temporarily blended into the indirect lighting atlas. Alongside this atlas we keep a current number of accumulated frames. Indirect lighting update rate is quite low and we need to limit the total number of accumulated frames to 4 in order to minimize ghosting.

# Secondary Bounce Roughly Matches Final Gather



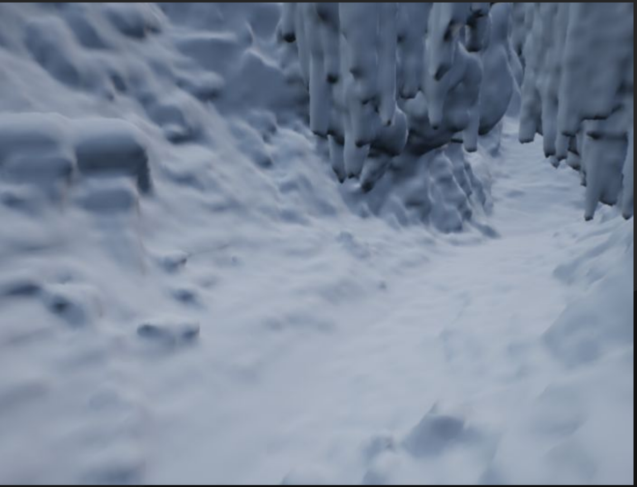
Here’s a comparison of pixel final gather against surface cache indirect lighting or in other words our secondary bounce.

Secondary bounce has lower quality due to performance constraints, but still it roughly matches pixel gather.

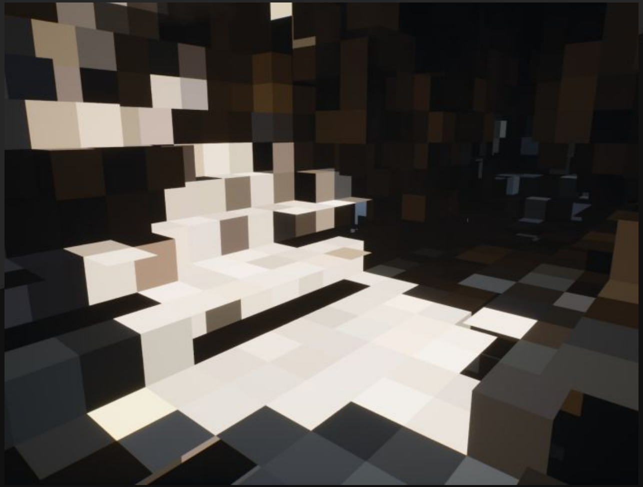
The quality level is fine for the diffuse multi-bounce or rough reflections and usually is good enough for mirror reflections.

# Voxel Lighting

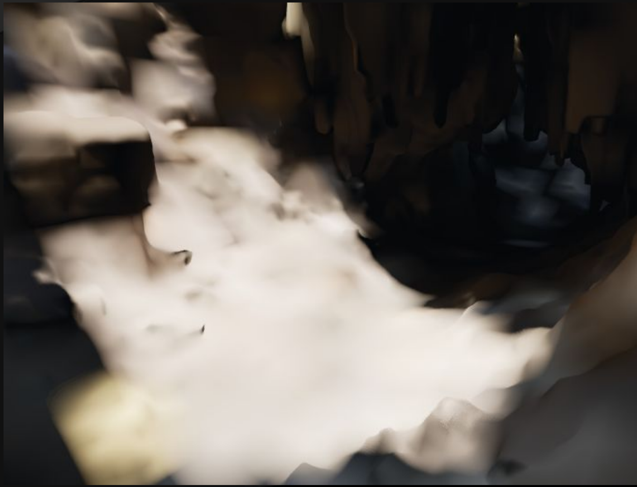
- Global SDF can't sample surface cache
- Clipmap volume of merged cards
    - 4 clipmaps of 64x64x64 voxels
    - Radiance per 6 directions per voxel
    - Sample and interpolate 3 directions using normal
    - Normalize using weights from alpha channel



Global SDF



Voxel Lighting



Shaded Global SDF

One last issue is that we can't sample surface cache directly from the global distance field traces. Global distance field is a merged scene representation and we don't know which mesh instance was hit.

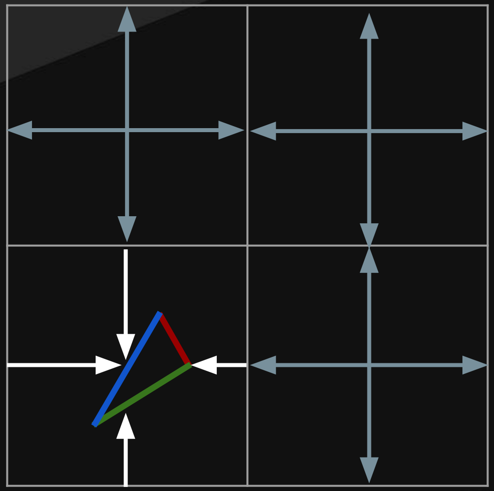
To fix this we also merge all cards into a set of global clipmaps centered around the camera.

Every voxel stores radiance per axis-aligned direction and during sampling we interpolate between directions and nearby voxels.

We weight every sample by weight stored in the alpha channel. This weight allows us to account for the missing cards and for card re-projection onto a fixed world space axis.

# Voxel Lighting Visibility Buffer

- Track object updates
- Build list of modified bricks on GPU
- Voxelize modified bricks by tracing rays
  - 6 rays per voxel
  - Cull objects to 4^3 bricks
  - One thread per mesh SDF per trace
  - InterlockedMin write to a visibility buffer
- Cache hits in a visibility buffer
  - 24 bit Mesh Index | 8 bit HitT



Updating the entire volume every frame is too expensive and we need to leverage caching.

Similar to the global distance field update we track scene modifications and update only modified bricks.

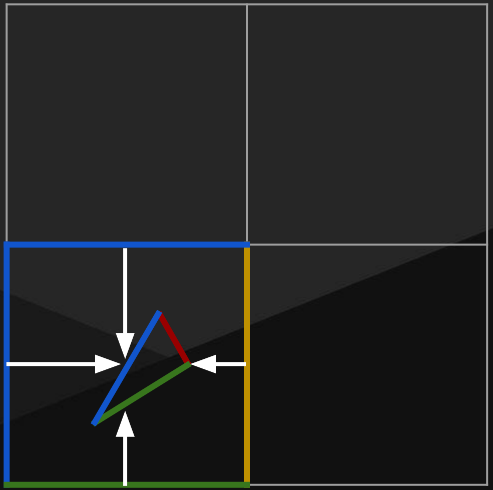
For every modified brick we build a list of objects inside it.

Next we run the ray tracing pass with one trace per thread. Every trace raymarches a single object and outputs to a visibility buffer using an atomic min.

This allows us to cache traces and update them only when some mesh moves in the scene. We cache geometry here instead of lighting, as lighting changes are hard to track and lighting can change every frame.

# Voxel Lighting Update

- Shade entire visibility buffer every frame
  - Compact active lanes
  - Sample lighting from surface cache
- Store projection weights in alpha



Next we shade the entire visibility buffer every frame.

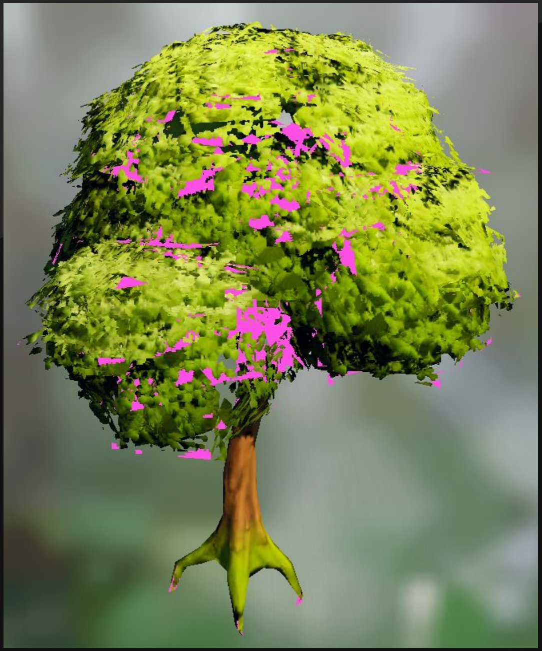
First we need to compact the visibility buffer as it's quite sparse.

After the compaction we sample surface cache for every valid visibility buffer entry to compute the final lighting.

During this step we also compute projection weights which will be stored in the alpha channel of the voxel lighting volume.

# Surface Cache Limitations

- Voxel lighting is too coarse
- No animation support
- Coverage issues
  - Limited number of surface layers



Missing coverage marked pink

As for the limitations of surface cache, the main one is quality of the voxel lighting which is just too coarse. This is something that we will have to address in the future.

Cards are placed during mesh import and don't support mesh animation. We alleviate some of this by increasing depth weight bias on foliage, which works reasonably well for small deformations, but doesn't work for characters.

Some meshes like trees that have too many layers can't be unwrapped using a reasonable number of layers. This can be noticeable in reflections, but it's not a big issue for diffuse rays as it will just cause a bit of missing energy.

# Summary

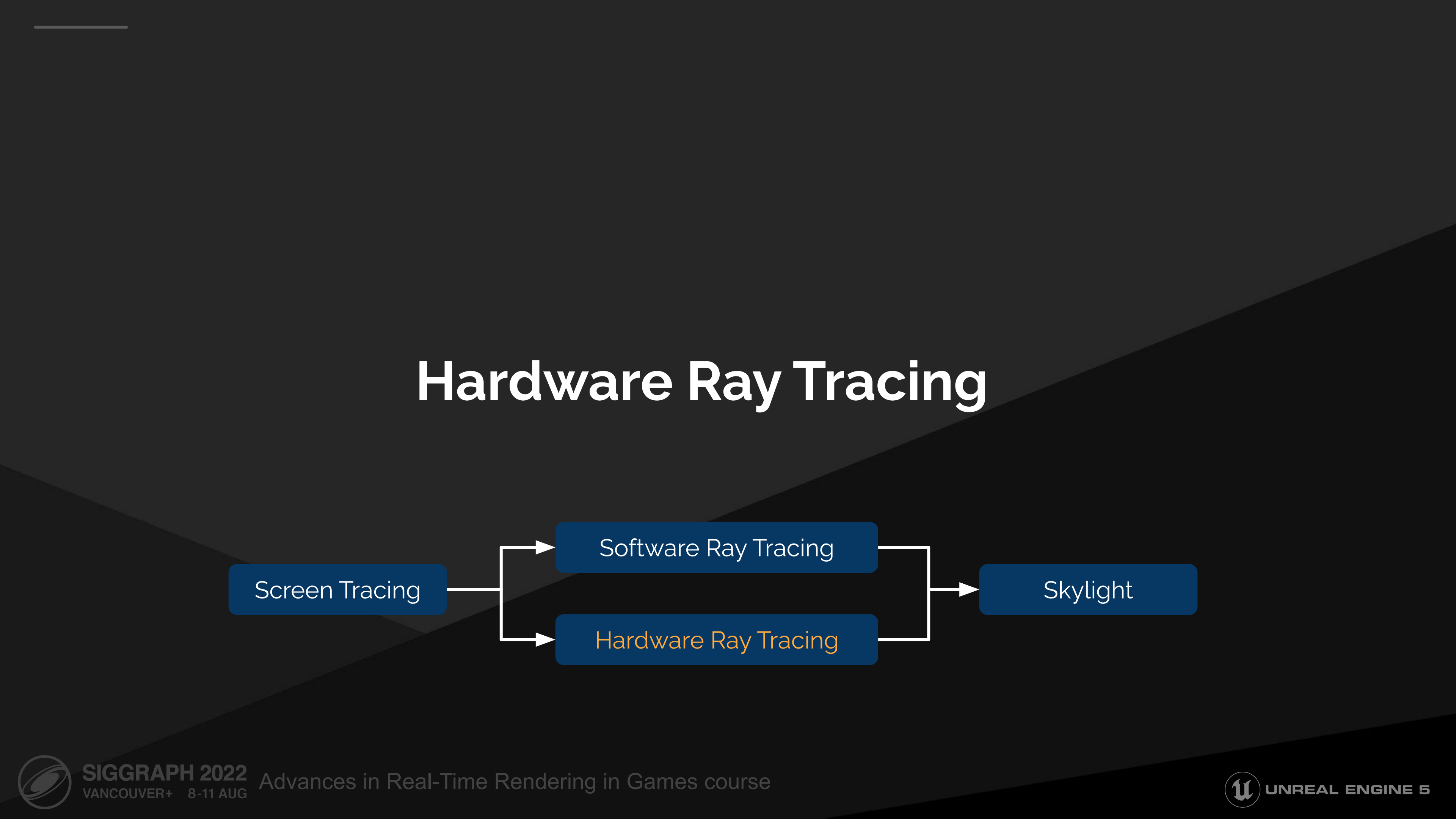
- Makes SDF tracing possible
- Fast lighting and material evaluation
    - Software and Hardware Ray Tracing
- Multiple bounces

The good parts about surface cache is that it enables distance field tracing.

It also is a great utility for caching various expensive computations. This is useful not only for distance field tracing, but also for hardware ray tracing where it allows to skip expensive material and lighting evaluation on hit.

Finally it enables high quality multiple bounces, which are important for believable GI and reflections.

Thank you for listening and now I’m going to hand off to Patrick.



# Hardware Ray Tracing

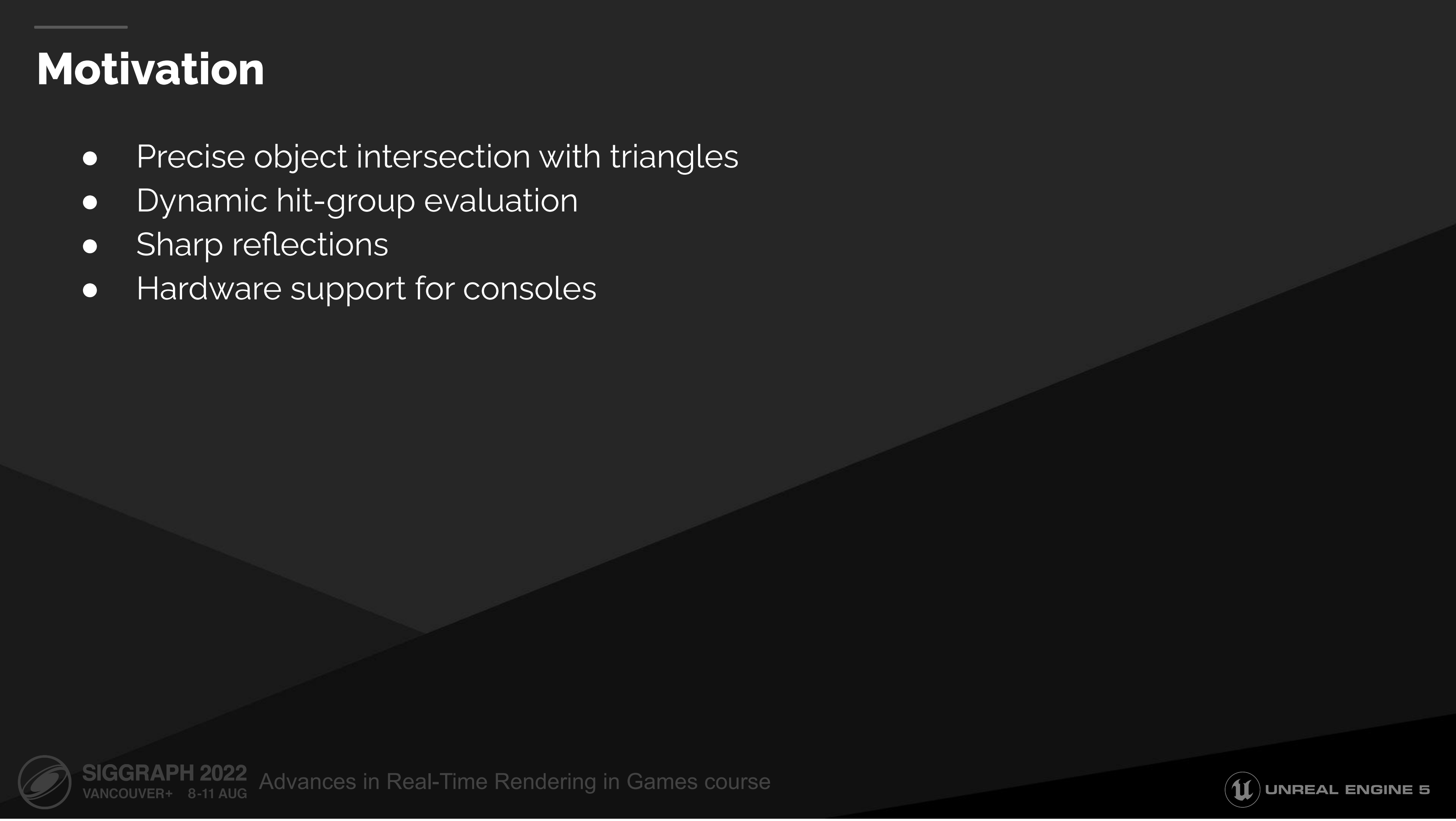
Screen Tracing

Software Ray Tracing

Hardware Ray Tracing

Skylight

Hi. My name is Patrick and I’ll be presenting our use of hardware ray tracing in Lumen.



# Motivation

- Precise object intersection with triangles
- Dynamic hit-group evaluation
- Sharp reflections
- Hardware support for consoles

Before I begin, I’d like to present some initial motivations for this work.

We were interested in using hardware ray tracing for several reasons. We can use hardware ray tracing to precisely calculate ray-object intersection with raw triangle data. It enables dynamic material and lighting evaluation, which makes sharp, off-screen reflections possible. And finally, the introduction of hardware ray tracing to modern consoles makes the technique much more widely available than it was before.

# Overview

- Initial experiments with reflections
  - UE4 model
  - Surface-cache model
  - Hybrid model
  - Incorporating opacity
- The Matrix Awakens with Hardware Ray Tracing (HWRT)
  - Nanite fallback meshes
  - Far-field representation
- Tiered-tracing pipeline

I’ll start by outlining some of our initial experiments with hardware ray tracing and Lumen. I’ll briefly discuss the incorporation of the UE4 model and its shortcomings. I’ll then present an alternative model, which minimizes runtime cost by using the surface cache and demonstrate how mixing both models can provide a qualitative advantage; ultimately arriving at the two evaluation modes we support in Lumen, today.

I’ll also present some practical solutions for representing opacity in our system, and discuss how Lumen's GPU-driven pipeline changed our invocation of hardware ray tracing.

I’ll highlight additional traversal-related complexities that we encountered during the creation of "The Matrix Awakens: An Unreal Engine Experience," such as incorporating Nanite fallback meshes and extending our trace distance into the far-field.

Finally, I’ll conclude by presenting the full hardware ray-tracing pipeline released in UE5.

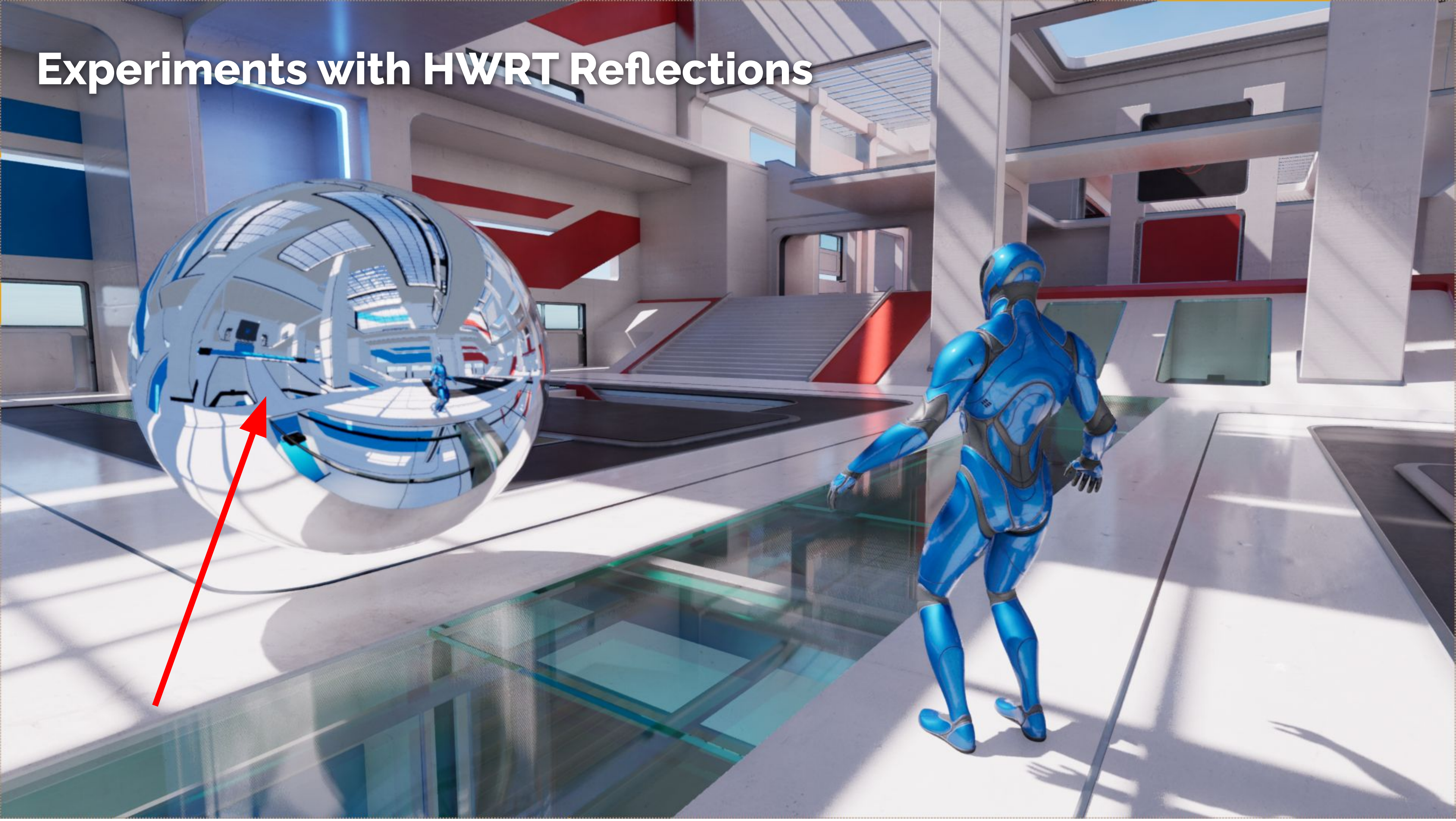
# Experiments with HWRT Reflections

- UE4 Model
  - 4.22 Ray-Traced Reflections
    - Dynamic material evaluation
    - Dynamic lighting evaluation

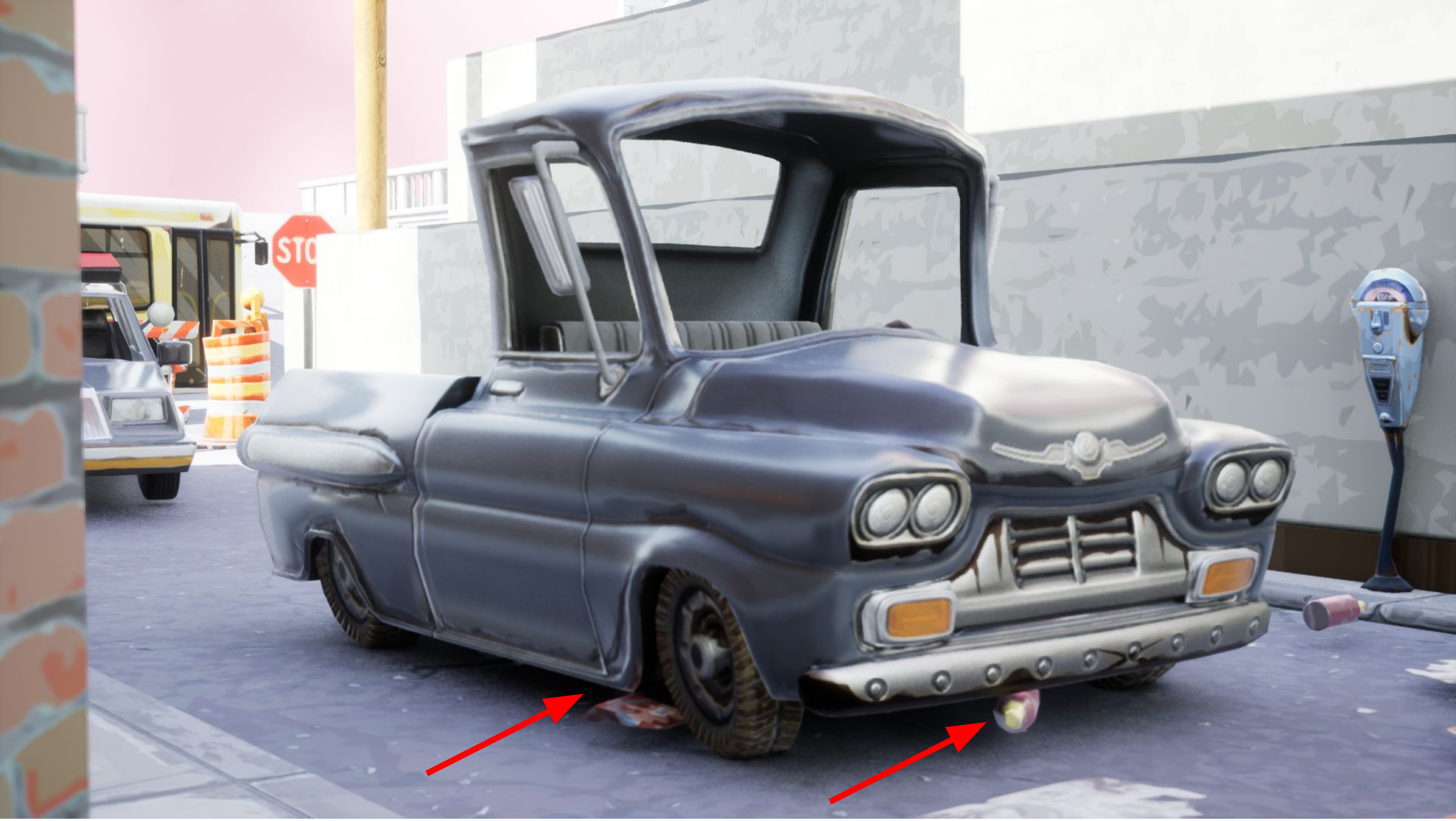
Image courtesy of Porsche, Epic Games and NVIDIA

Initial deployment of HWRT in Lumen began with reflections. And for this task, it seemed most appropriate to start by incorporating the UE4 Ray-Traced Reflections model. While integration of Ray-Traced Reflections provided an immediate solution for dynamic material and lighting evaluation, it was clear from the beginning that the shortcomings of the UE4 model would need to be addressed before we could deliver an integrated solution.

# Experiments with HWRT Reflections



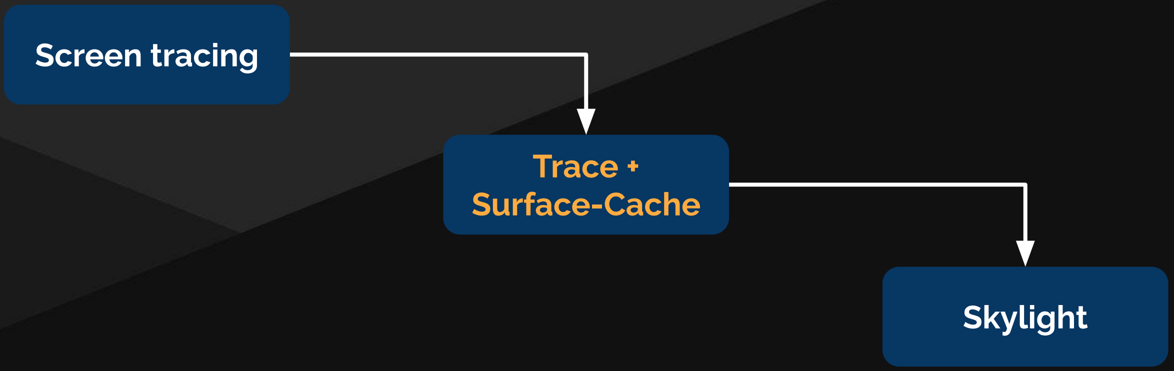
The most notable problem we encountered when deploying the UE4 reflection model was the lack of proper specular occlusion. The mirror ball shows the artifact clearly in our Lyra sample, where the unshadowed SkyLight adds an unnatural blue hue to the interior.



The lack of correct specular occlusion can also be seen in this image.

# Surface-Cache Pipeline

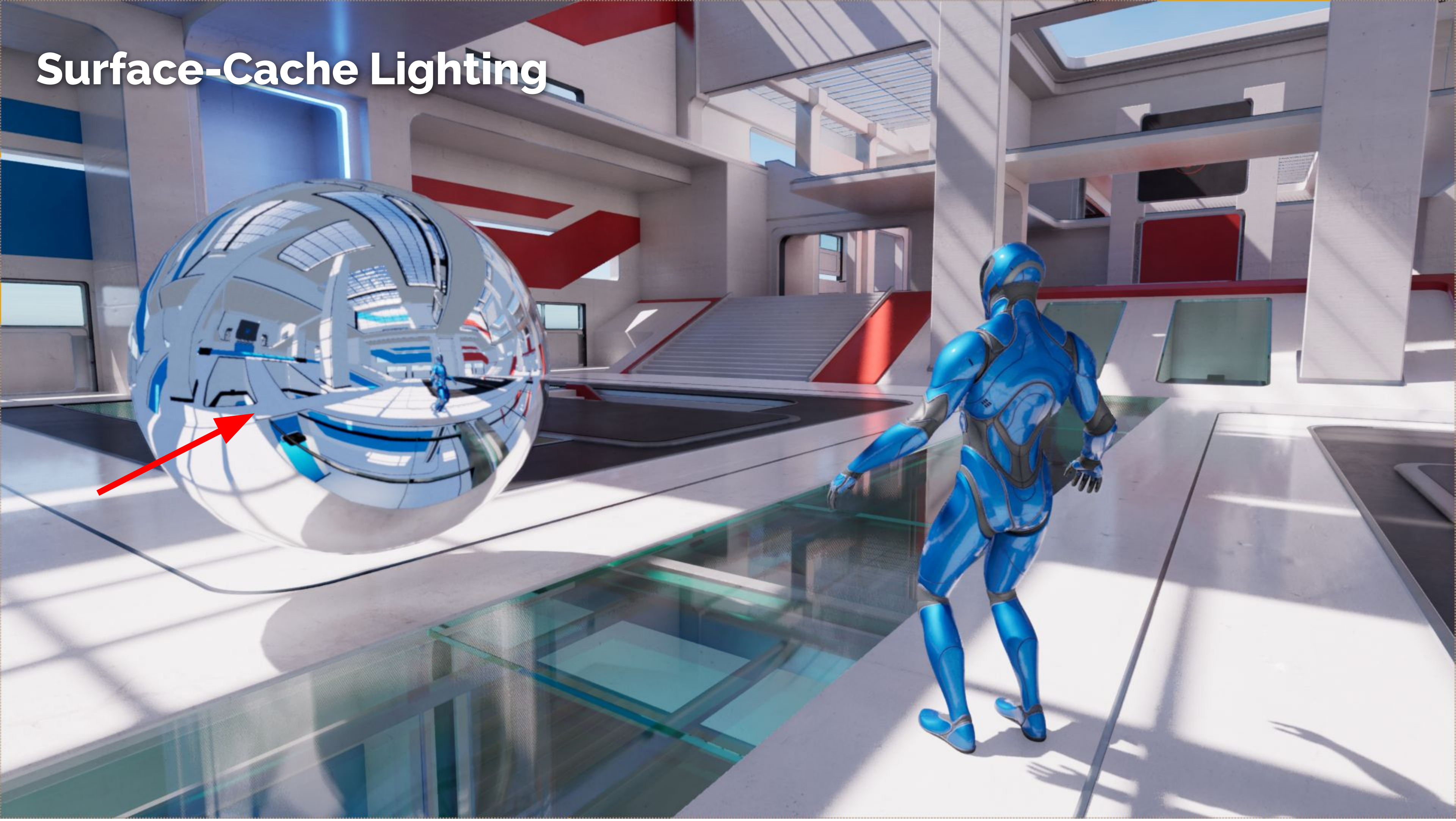
- Design for performance by embracing the Surface Cache
  - No Any-Hit Shader (AHS), by forcing opaque
  - Modify Closest-Hit Shader (CHS) to only fetch:
    - geometry normal,
    - surface cache parameterization
  - Apply lighting as normal-weighted surface-cache evaluation



At the same time, we also realized that it would be an interesting experiment to forego dynamic evaluation and rely on the surface cache exclusively. Deciding that this invocation could ultimately become Lumen hardware ray-tracing's "fast path," we also decided to impose additional constraints on the pipeline, and eliminated the use of an any-hit shader by forcing all objects in the BVH to be opaque.

For this model, we replaced the set of material-dependent closest-hit shaders with a single closest-hit shader, and this shader only fetches only the data necessary to extract the geometric normal and surface cache parameterization. The ray-generation shader then applies lighting as a normal-weighted surface-cache evaluation.

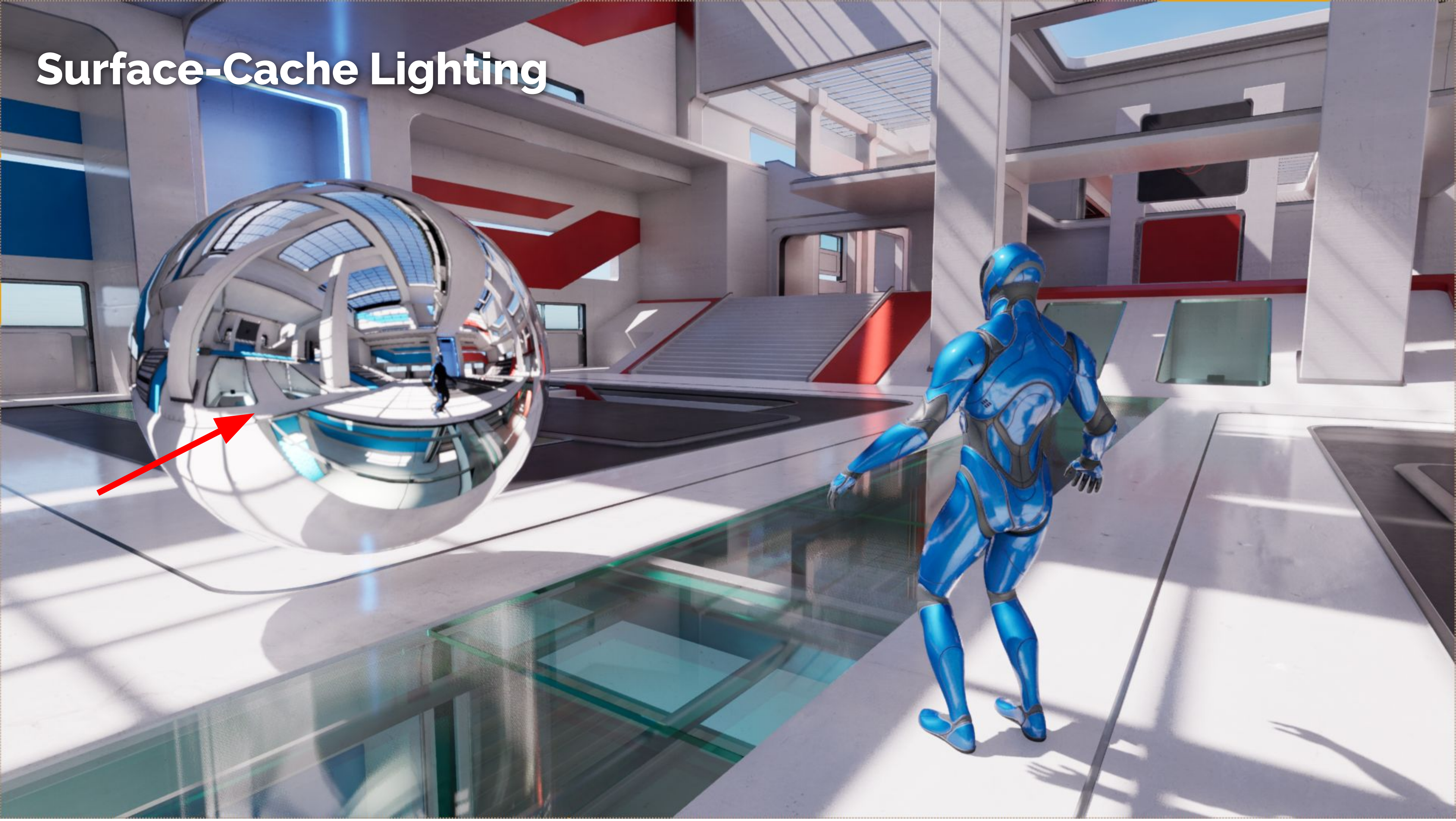
We call this pipeline the Surface-Cache Pipeline.



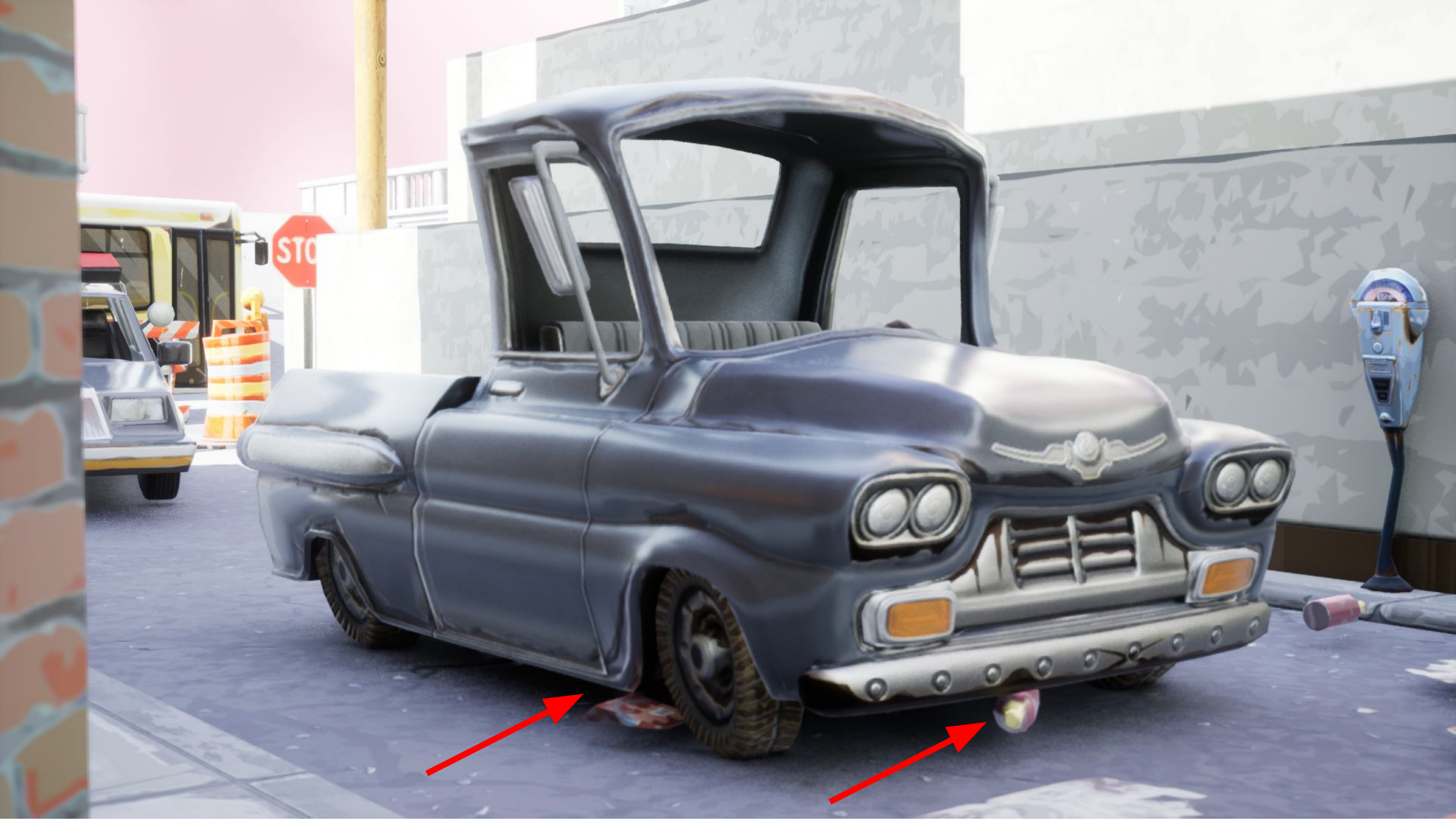
# Surface-Cache Lighting

Utilizing the Surface-Cache, we get the direct and indirect lighting in a single evaluation. By using the Surface-Cache, we can overcome the unshadowed SkyLight problem previously presented.

# Surface-Cache Lighting

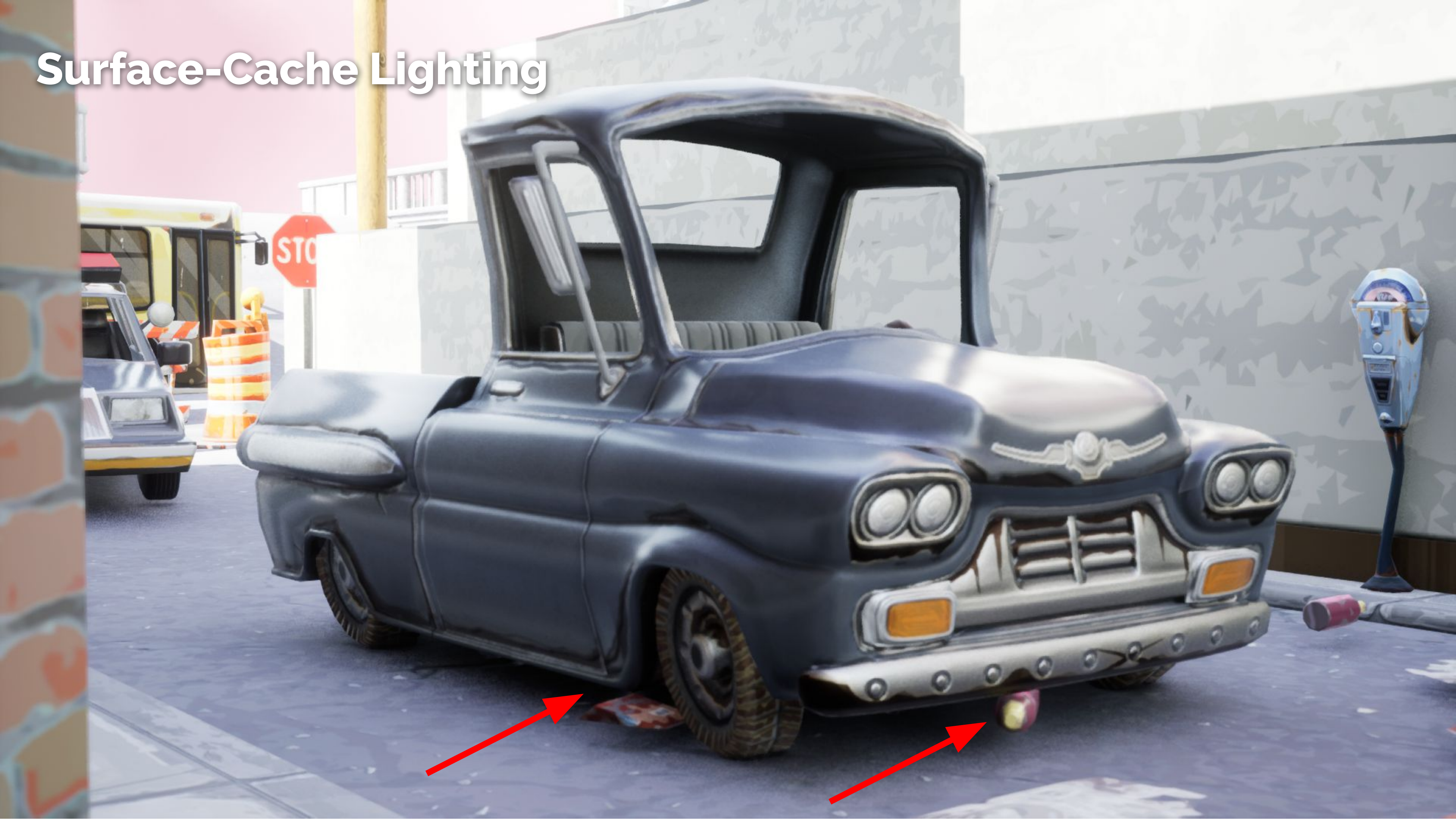


Utilizing the Surface-Cache, we get the direct and indirect lighting in a single evaluation. By  using the Surface-Cache, we can overcome the unshadowed SkyLight problem previously presented.



And it corrects the specular occlusion previously mentioned.

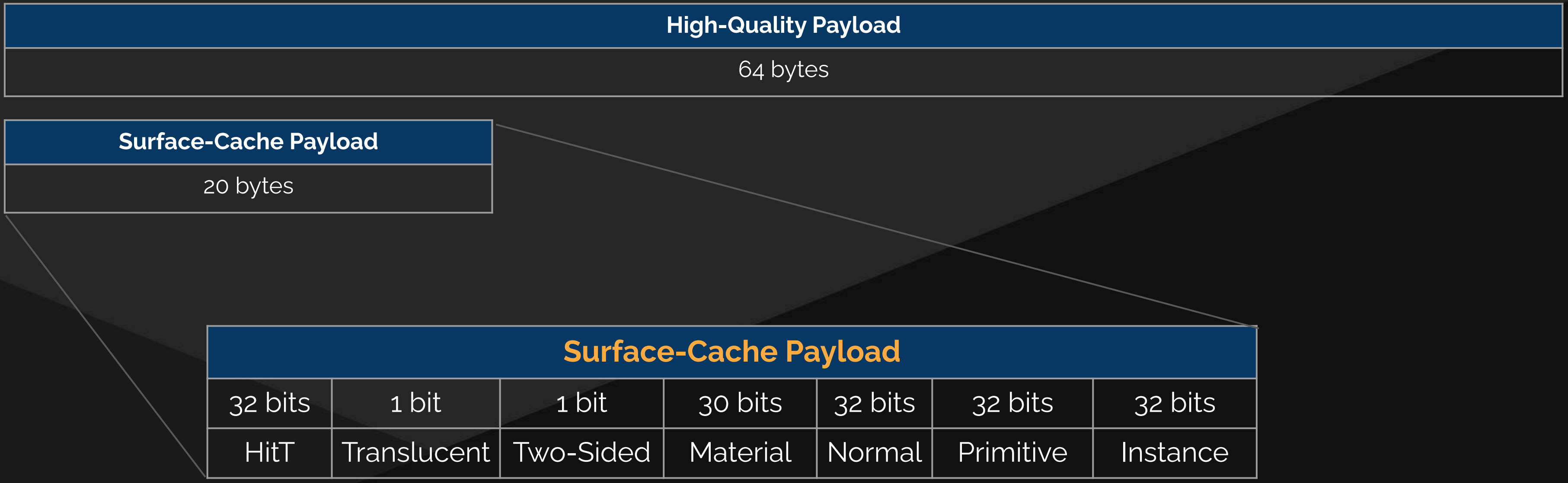
# Surface-Cache Lighting



And it corrects the specular occlusion previously mentioned.

# Surface-Cache Payload

- Replaces 64-byte High-Quality payload with 20-byte Surface-Cache payload



| High-Quality Payload |
|---|
| 64 bytes |

| Surface-Cache Payload |
|---|
| 20 bytes |

| Surface-Cache Payload | | | | | | |
|---|---|---|---|---|---|---|
| 32 bits | 1 bit | 1 bit | 30 bits | 32 bits | 32 bits | 32 bits |
| HitT | Translucent | Two-Sided | Material | Normal | Primitive | Instance |

With the aggressive optimizations to the Surface-Cache pipeline, we created a new payload structure to minimize bandwidth pressure. The high-quality payload used in the UE4 model requires 64-bytes to store GBuffer-like parameters for dynamic lighting. This includes parameters such as BaseColor, Normal, Roughness, Opacity, Specular, and more. In comparison, the Surface-Cache payload only requires 20-bytes to store the necessary parameters for a Surface-Cache lighting lookup.

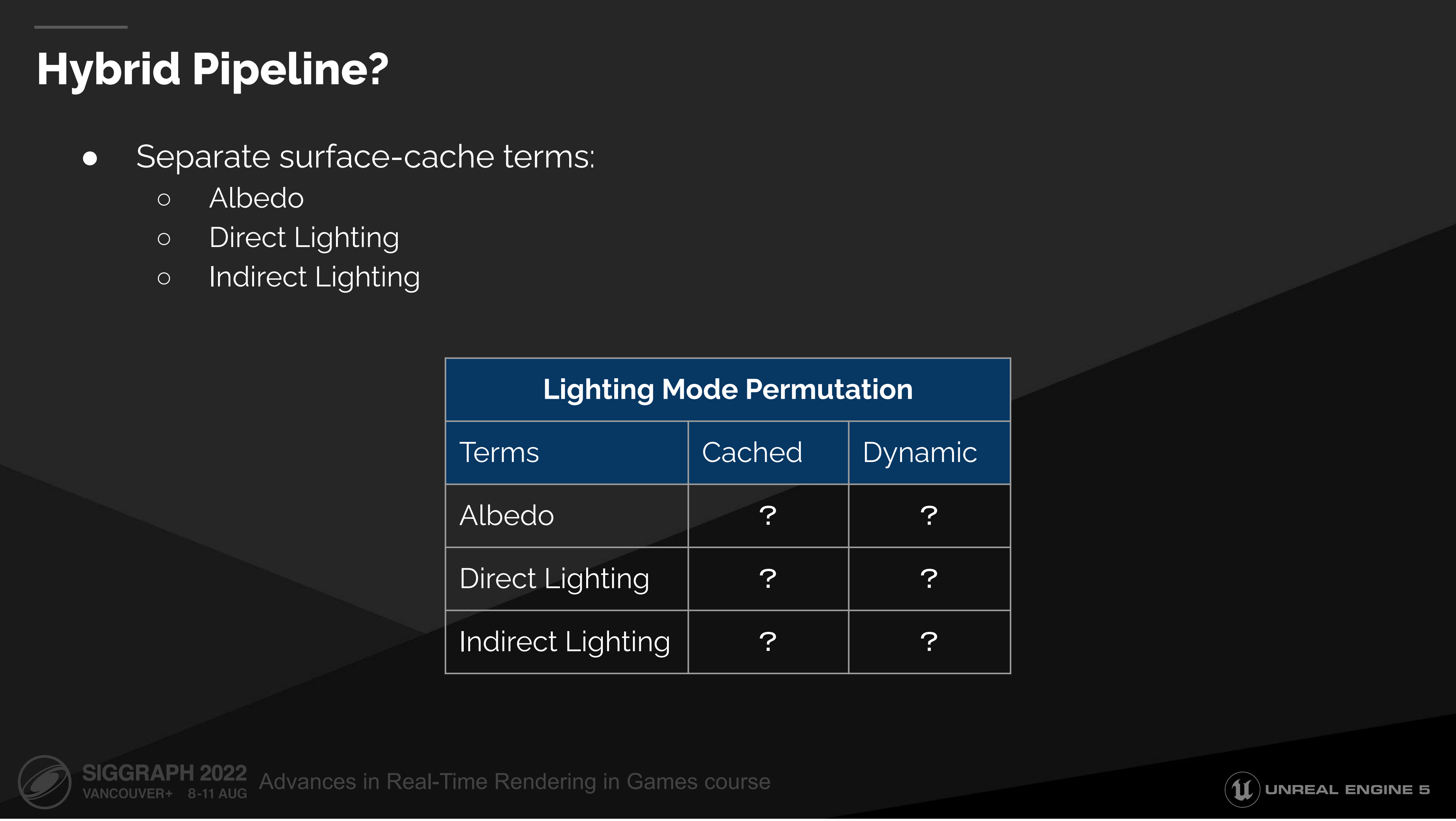
The Surface-Cache payload parameter set is listed below. The bits designated to the Material are not necessary for the Surface-Cache pipeline, but I will explain its purpose later.

# Surface-Cache Pipeline

- Shader Binding Table
  - Removes expensive bindings loop during shader assignment
  - Hit-group indexing still required to reconstruct normals on PC

The Surface-Cache pipeline also simplifies construction of the shader binding table. With the new model, the binding loop during no longer needs to fetch material-dependent resources. For a large number of instances, this creates a noticeable savings with respect to CPU time.

Unfortunately, we cannot completely eliminate the binding loop entirely, as vertex and index buffer bindings are required to reconstruct surface normals in DXR, and UE5 does not currently use bindless resources.



# Hybrid Pipeline?

- Separate surface-cache terms:
  - Albedo
  - Direct Lighting
  - Indirect Lighting

| Lighting Mode Permutation | | |
|---|---|---|
| Terms | Cached | Dynamic |
| Albedo | ? | ? |
| Direct Lighting | ? | ? |
| Indirect Lighting | ? | ? |

After constructing the two models, we anticipated that mixing the two strategies could provide a natural scaling control to govern the performance vs quality tradeoff. We accomplished this by conditionally separating surface-cache terms, such as albedo, direct lighting, and indirect lighting depending upon the desired level of dynamic evaluation. This also presented a mechanism to incorporate the surface-cache indirect lighting with the dynamic evaluation of UE4 model, eliminating a fundamental problem with unshadowed SkyLight.

Even though we could separately control whether to dynamically evaluate the material or the lighting, we found that partial dynamic evaluation was just as expensive as full dynamic evaluation.

# Ray Lighting Modes

- Surface Cache
- Hit Lighting
  - Incorporates sorted-deferred optimization [Tatu Aalto 2018] [Kelly et al 2021]

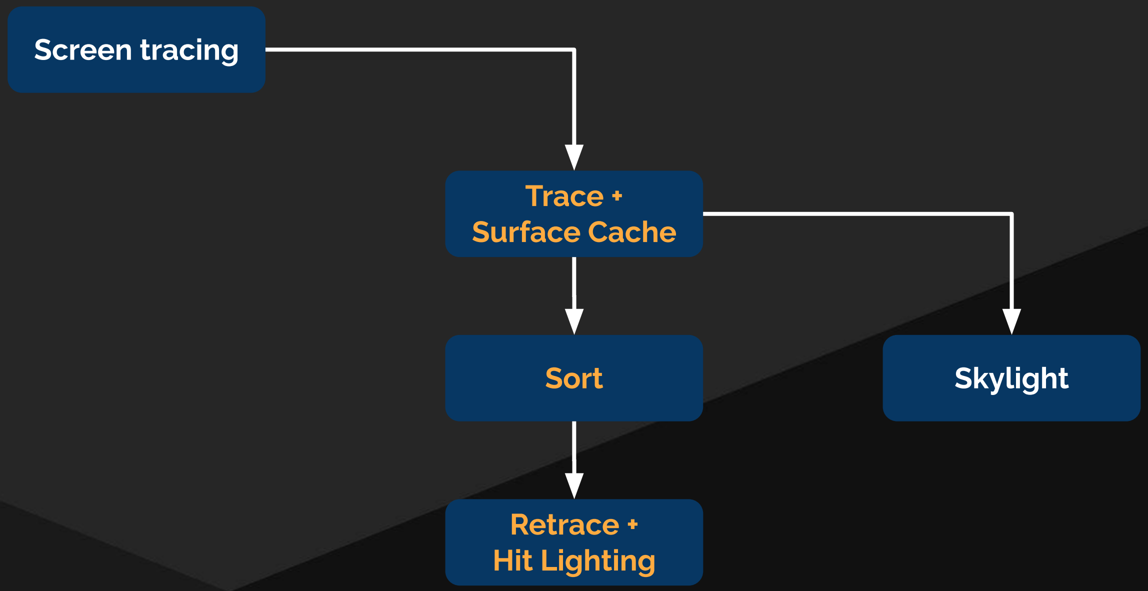
| Surface Cache | | |
|---|---|---|
| Terms | Cached | Dynamic |
| Albedo | ✔ | |
| Direct Lighting | ✔ | |
| Indirect Lighting | ✔ | |

| Hit Lighting | | |
|---|---|---|
| Terms | Cached | Dynamic |
| Albedo | | ✔ |
| Direct Lighting | | ✔ |
| Indirect Lighting | ✔ | |

With this finding, we only expose two lighting configurations to the artist. The Surface-Cache mode implements the previously described Surface-Cache model. And the Hit-Lighting mode represents the modified UE4 model, which now incorporates indirect lighting from the surface cache. At the moment, we only expose the Hit-Lighting model for reflections.

It is important to point out that the Hit-Lighting model implements a modified version of the previously published Sorted-Deferred tracing pipeline [Tatu Aalto 2018] [Kelly et al 2021]. And since the sorted-deferred optimization requires exporting the Material ID, this is why it is included in the Surface-Cache Payload.

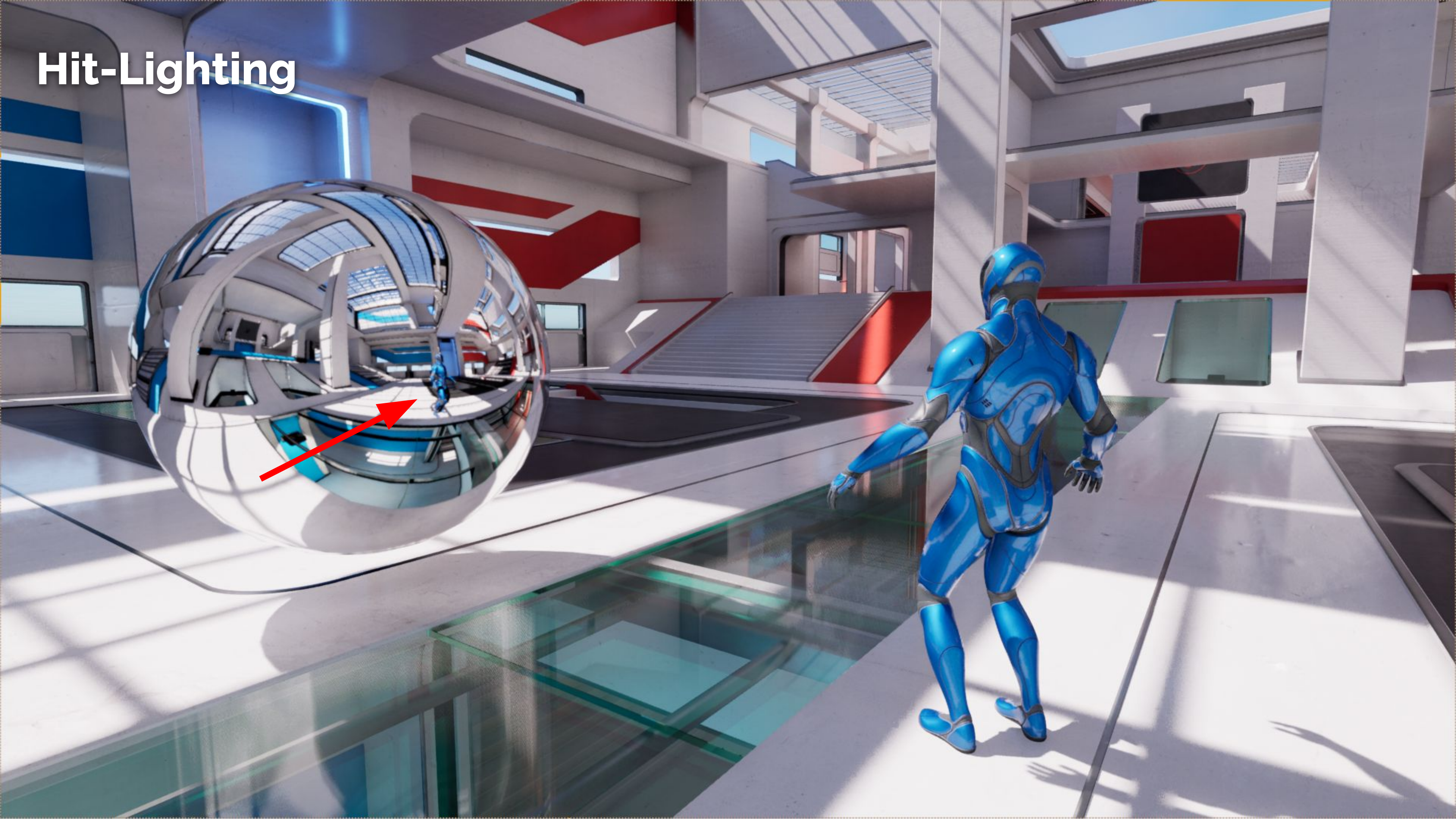
# Hit-Lighting Pipeline



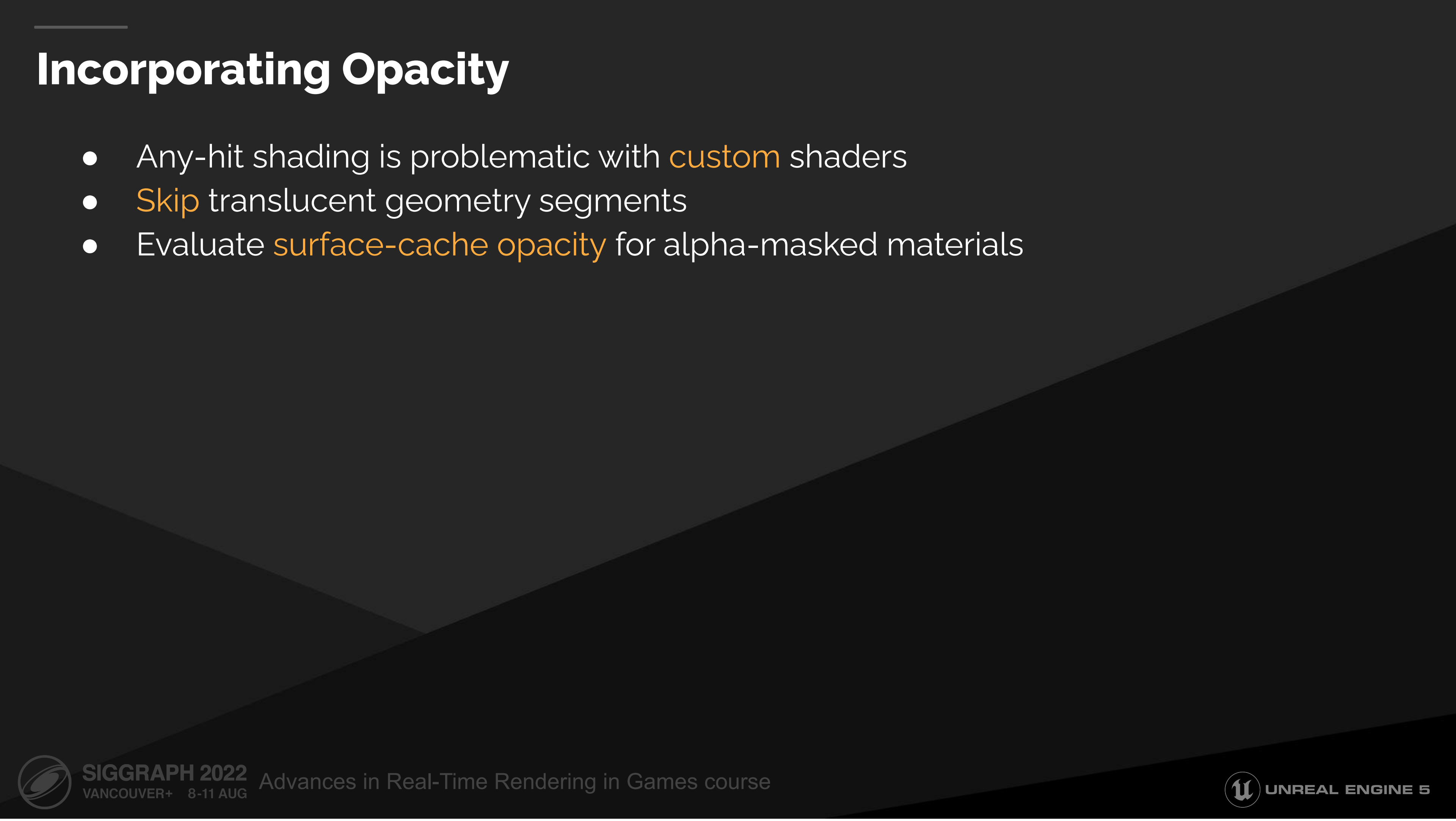
This allows us to repurpose the Surface-Cache tracing stage as the prerequisite to the Hit-Lighting pipeline. It also grants us the flexibility to optionally invoke dynamic evaluation on a per-ray basis. For instance, we have experimented with this idea to optionally invoke Hit-Lighting on meshes which have no Surface Cache parameterization.

Here is the Hit-Lighting pipeline, which adds sorting and re-tracing after initially executing the Surface-Cache tracing pass.

# Hit-Lighting



In the Lyra example, we can see that Hit-Lighting provides direct illumination to the reflection of the skeletal mesh in the mirror ball. Since skeletal meshes do not have a Surface-Cache parameterization, this effect is only possible with the Hit-Lighting model.



## Incorporating Opacity

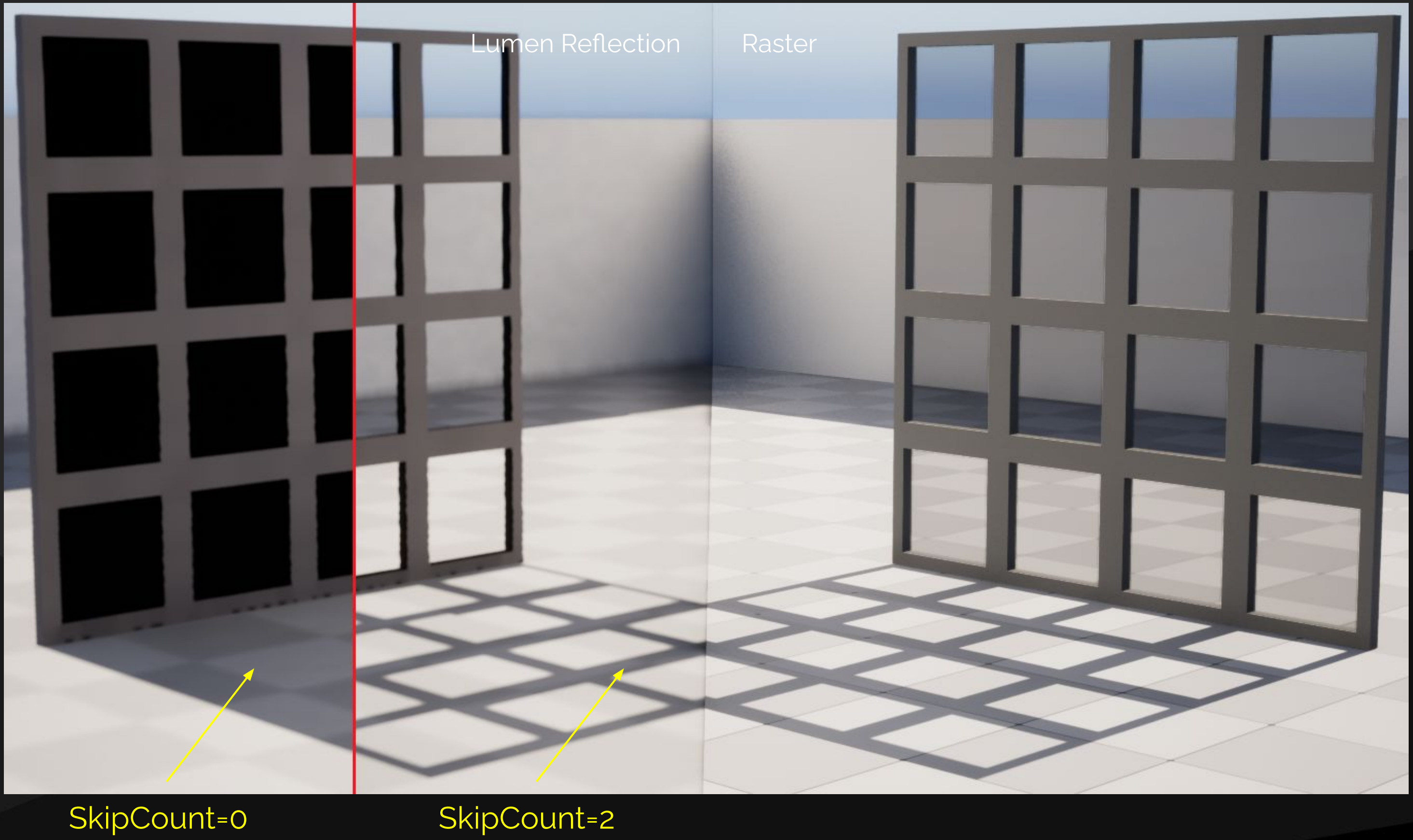
- Any-hit shading is problematic with custom shaders
- Skip translucent geometry segments
- Evaluate surface-cache opacity for alpha-masked materials

The Surface-Cache pipeline presents additional challenges when dealing with partially opaque geometry. As mentioned previously, when employing the Surface-Cache model, we explicitly treat all geometry in the BVH as being fully opaque. We do this because invoking any-hit shading incurs a noticeable run-time penalty that we wish to avoid.

In an attempt to overcome noticeable problems with opaque traversal, we employ two material-dependent strategies. For translucent materials, we choose to skip those meshes entirely. For alpha-masked materials, we evaluate the surface-cache opacity and decide to skip meshes which are less than 50% opaque.

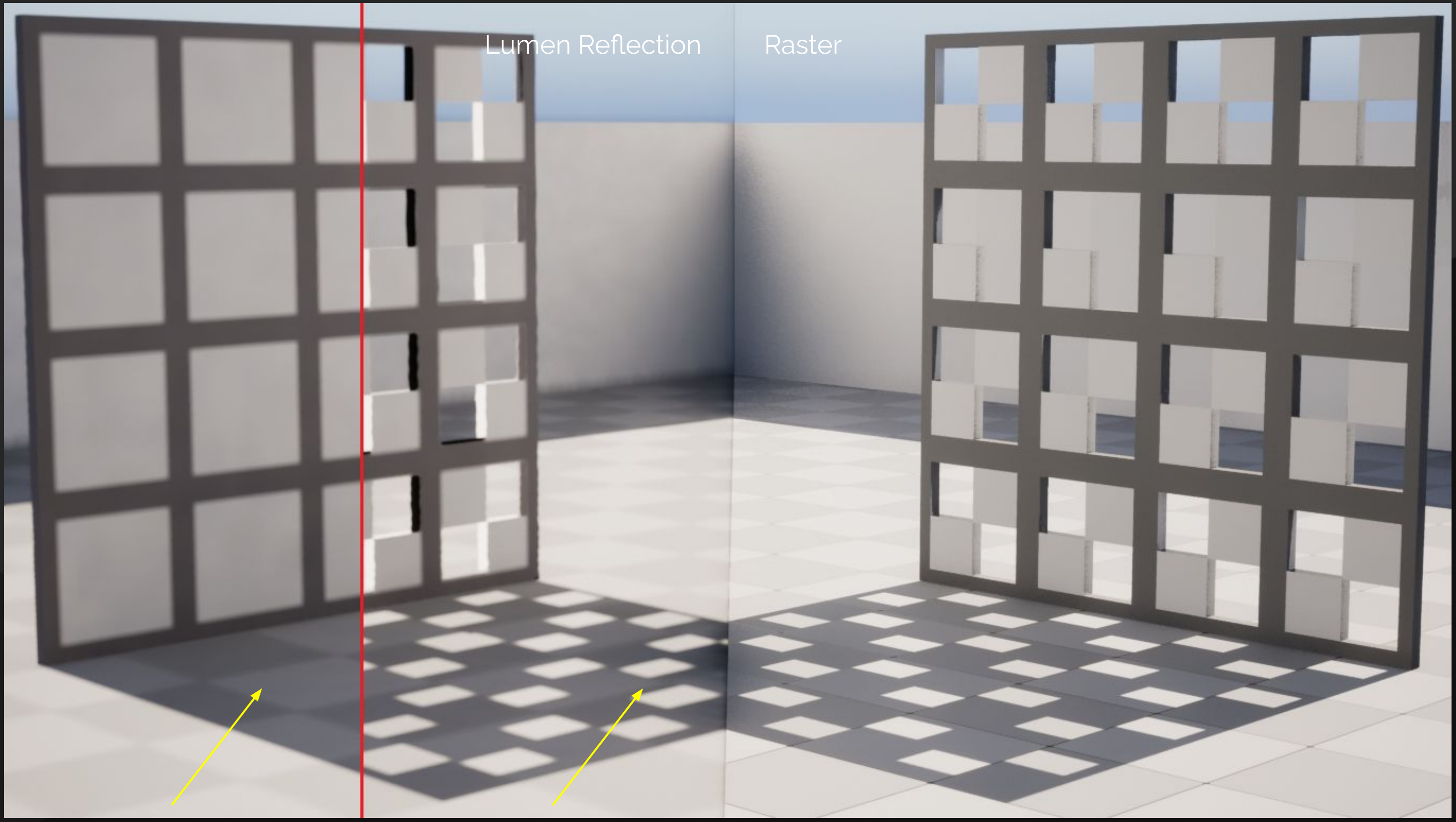
Since we chose to omit an any-hit shader, we must iteratively traverse through the scene in the ray-generation shader whenever we encounter a partially opaque surface. This iteration count is governed by a MaxTranslucentSkipCount shader parameter.

# MaxTranslucentSkipCount



This example shows the application of MaxTranslucentSkipCount when encountering a translucent geometry segment.

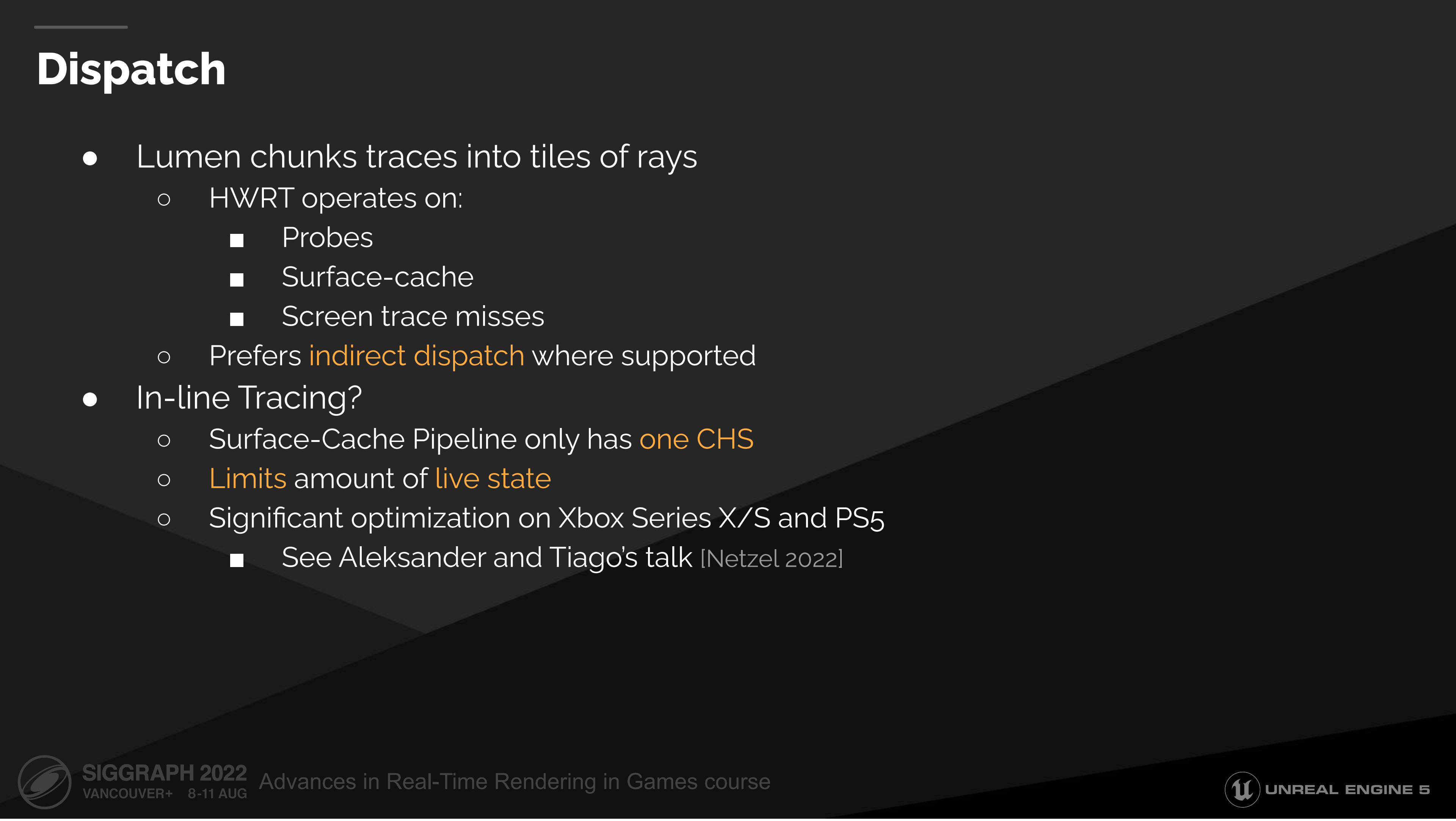
# Alpha Masking



No Mask

Alpha Mask

This example shows the application of evaluating surface-cache opacity to reconstruct the geometry’s alpha mask during traversal.



## Dispatch

- Lumen chunks traces into tiles of rays
    - HWRT operates on:
        - Probes
        - Surface-cache
        - Screen trace misses
    - Prefers indirect dispatch where supported
- In-line Tracing?
    - Surface-Cache Pipeline only has one CHS
    - Limits amount of live state
    - Significant optimization on Xbox Series X/S and PS5
        - See Aleksander and Tiago's talk [Netzel 2022]

Lumen's GPU-driven pipeline required additional dispatch control that was not present in UE4. Instead of operating directly on screen pixels, Lumen passes operate on probes, surface cache texels, and screen tiles. In many instances, hardware ray tracing is a secondary trace type and operates as the fall-through technique. For these reasons, it was essential that we utilize indirect dispatch where supported. For this reason, Lumen prefers DXR 1.1 semantics on PC.

We should also point out another useful feature in DXR 1.1: Inline ray tracing. The RayQuery interface avoids the complexity of the shader binding table and this allows for the hardware traversal in standard compute and pixel shaders. Utilizing ray queries also offers the compiler significant opportunity to optimize. In the ray-generation case, it is strongly recommended to minimize the amount of "live state" spanning a TraceRay() call. With Inline ray tracing, the compiler can minimize this without developer intervention.

Except for the need to supply mesh-varying vertex and index buffer data to hit-group shaders, the Surface-Cache pipeline could use inline ray tracing. This auxiliary buffers are only a requirement for PC; however, as console ray tracing intrinsics already provide access to the geometric normal as part of the ray-tracing Hit structure. Because of this, we actually do leverage inline ray tracing on consoles and benefit from a noticeable speedup on certain platforms. To learn more about this console specialization, please see Aleksander and Tiago's talk.

# The Matrix Awakens

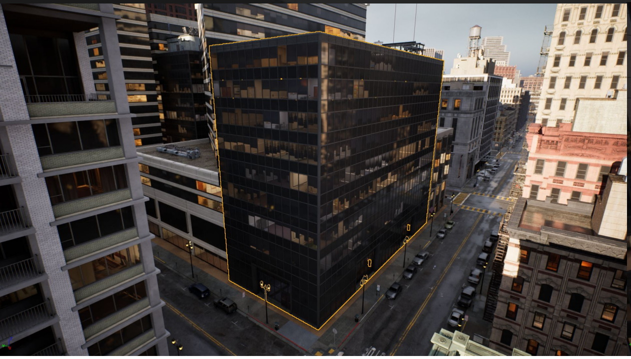
- Complicated scene construction
  - Instance count
  - Dynamic objects
  - Sharp reflections
- Shipping on console
  - Less computation and slower traversal
  - More customization over the RT pipeline
    - Pre-built BLAS for Static Meshes
    - See Aleksander and Tiago’s talk!

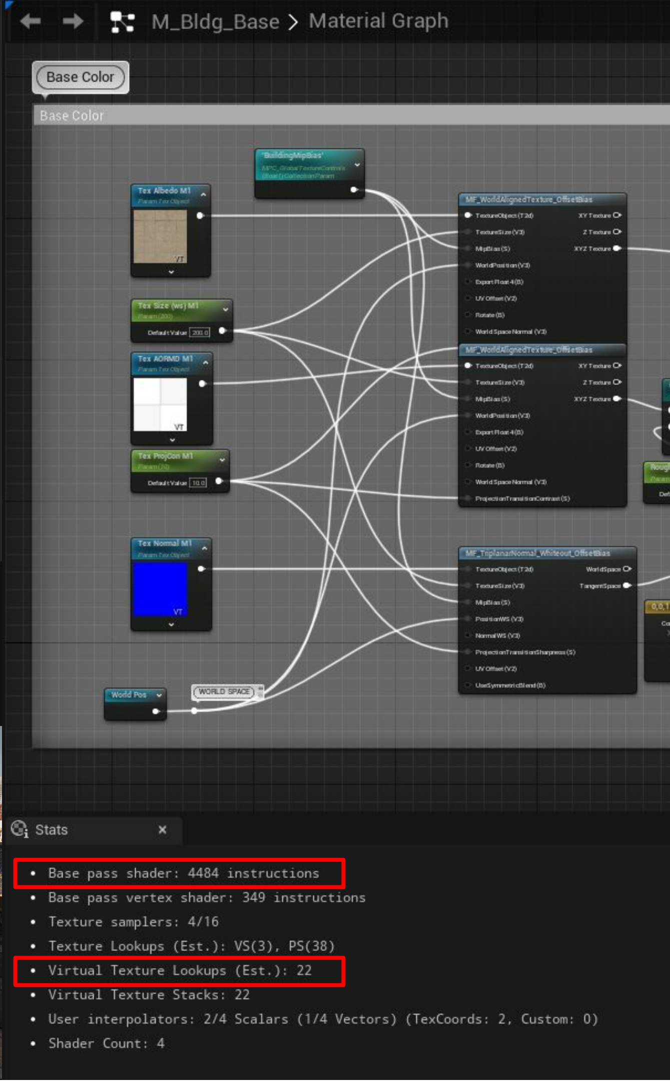
The Matrix Awakens: An Unreal Engine Experience presented some difficult challenges for the ray-tracing model. The open world demanded operating with a large instance count, which could easily overwhelm our top-level acceleration structure rebuild time. The sheer number of active cars and pedestrians in the world required a large number of dynamic refits to our bottom-level acceleration structures. Car paint and glass materials would not be realistic without mirror reflections.

The demo’s target shipping platform was Xbox Series X/S and PS5. These machines have native ray tracing support, but with less overall computational power and slower traversal than what is available on high-end PC systems. However, console APIs offer greater flexibility in the ray-tracing pipeline, which we could leverage to our advantage. For instance, construction of acceleration structures for static meshes can be pre-built and streamed, significantly reducing frame-time dedicated to building and refitting bottom-level acceleration structures. For more details on console-specific optimizations, we refer again to Aleksander and Tiago's talk.

# The Matrix Awakens

- Hit-Lighting performance suffers…
  - Master material complexity
  - Dozens of 4K virtual textures per material
- Use Surface-Cache lighting



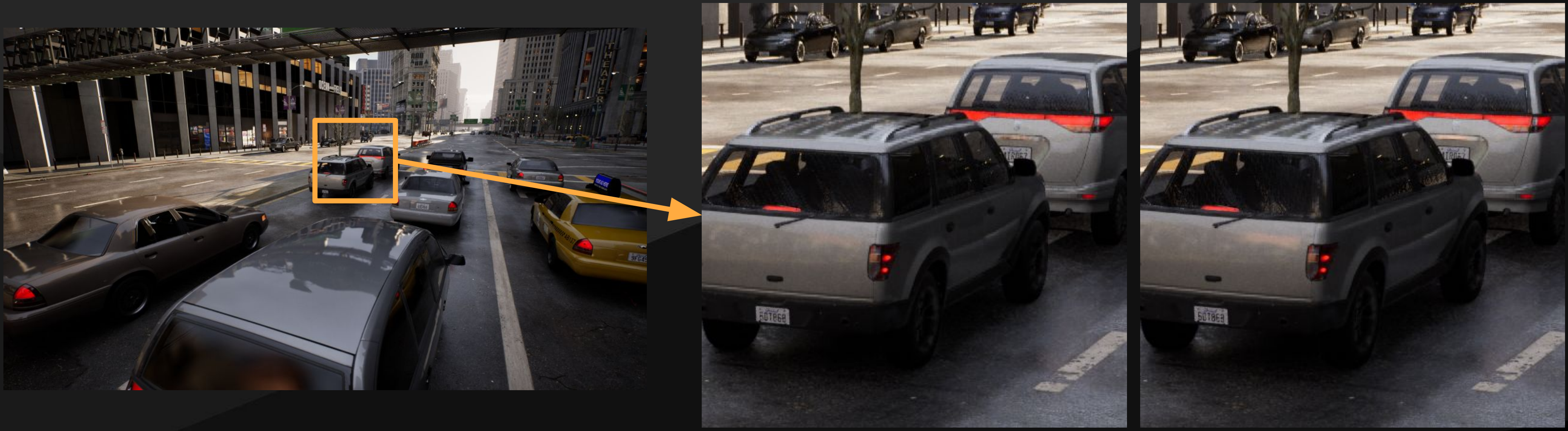


```
• Base pass shader: 4484 instructions
• Base pass vertex shader: 349 instructions
• Texture samplers: 4/16
• Texture Lookups (Est.): VS(3), PS(38)
• Virtual Texture Lookups (Est.): 22
• Virtual Texture Stacks: 22
• User interpolators: 2/4 Scalars (1/4 Vectors) (TexCoords: 2, Custom: 0)
• Shader Count: 4
```

Unfortunately, the material complexity in the demo made reliance on the Hit-Lighting model infeasible. Along with high instruction counts, our master materials invoked dozens of virtual texture fetches. Dynamic material and lighting evaluation was simply not performant. We had initial hopes for invoking Hit-Lighting on dynamic objects as mentioned previously, but we simply ran out of budget.

# “The Matrix Awakens” - UE5 demo

| PS5, 1080p | | |
|---|---|---|
| | HWRT + Surface Cache | HWRT + Hit Lighting |
| Reflection tracing | 2.44 ms | 11.54 ms |



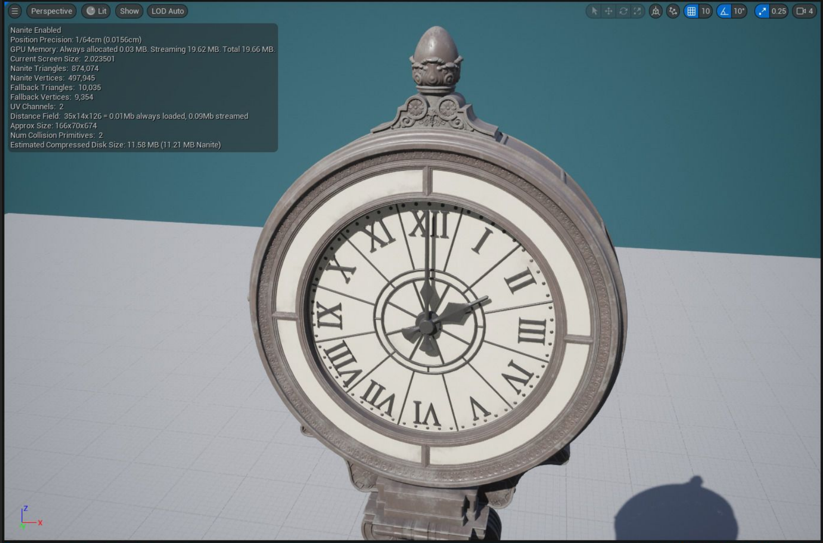
HWRT + Surface Cache

HWRT + Hit Lighting

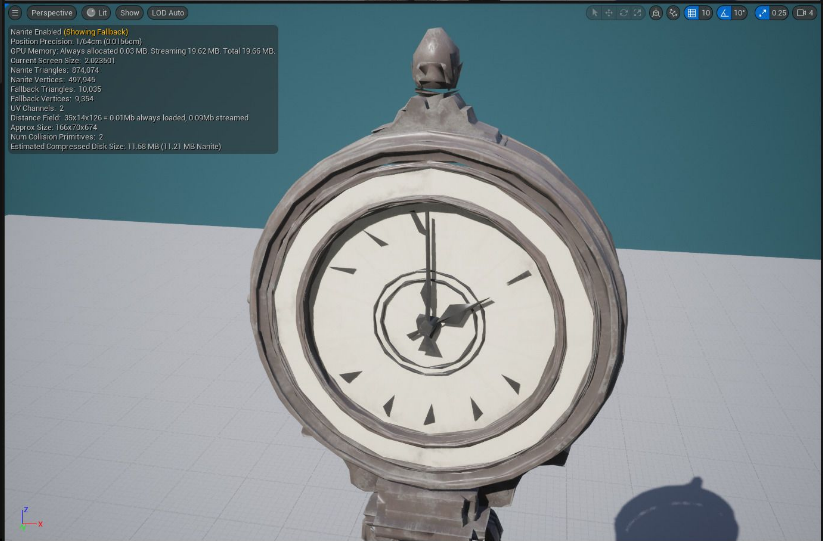
This timing report shows the staggering evaluation cost of invoking Hit-Lighting with our complicated materials on PS5. Interestingly, Hit-Lighting did not provide a substantial qualitative benefit, either. This is due to the strong dependence on the SkyLight term in our demo, which both pipelines now fetch from the Surface-Cache.

# Tracing with Nanite

- Nanite geometry is complex
  - Too large to store natively in BLAS
  - Vertex/Index buffer submission
    - DXR requires CPU submission
    - Nanite clusters are created on GPU
  - Active research area..
- Nanite fallback meshes
  - Simplified proxy geometry
  - No topological guarantees with original
- Self-intersection is a problem!



```
Nanite Enabled
Position Precision: 1/64cm (0.0156cm)
GPU Memory: Always allocated 0.03 MB. Streaming 19.62 MB. Total 19.66 MB.
Current Screen Size:  2.023501
Nanite Triangles:  874,074
Nanite Vertices:  497,945
Fallback Triangles:  10,035
Fallback Vertices:  9,354
UV Channels:  2
Distance Field:  35x14x126 = 0.01Mb always loaded, 0.09Mb streamed
Approx Size: 166x70x674
Num Collision Primitives:  2
Estimated Compressed Disk Size: 11.58 MB (11.21 MB Nanite)
```



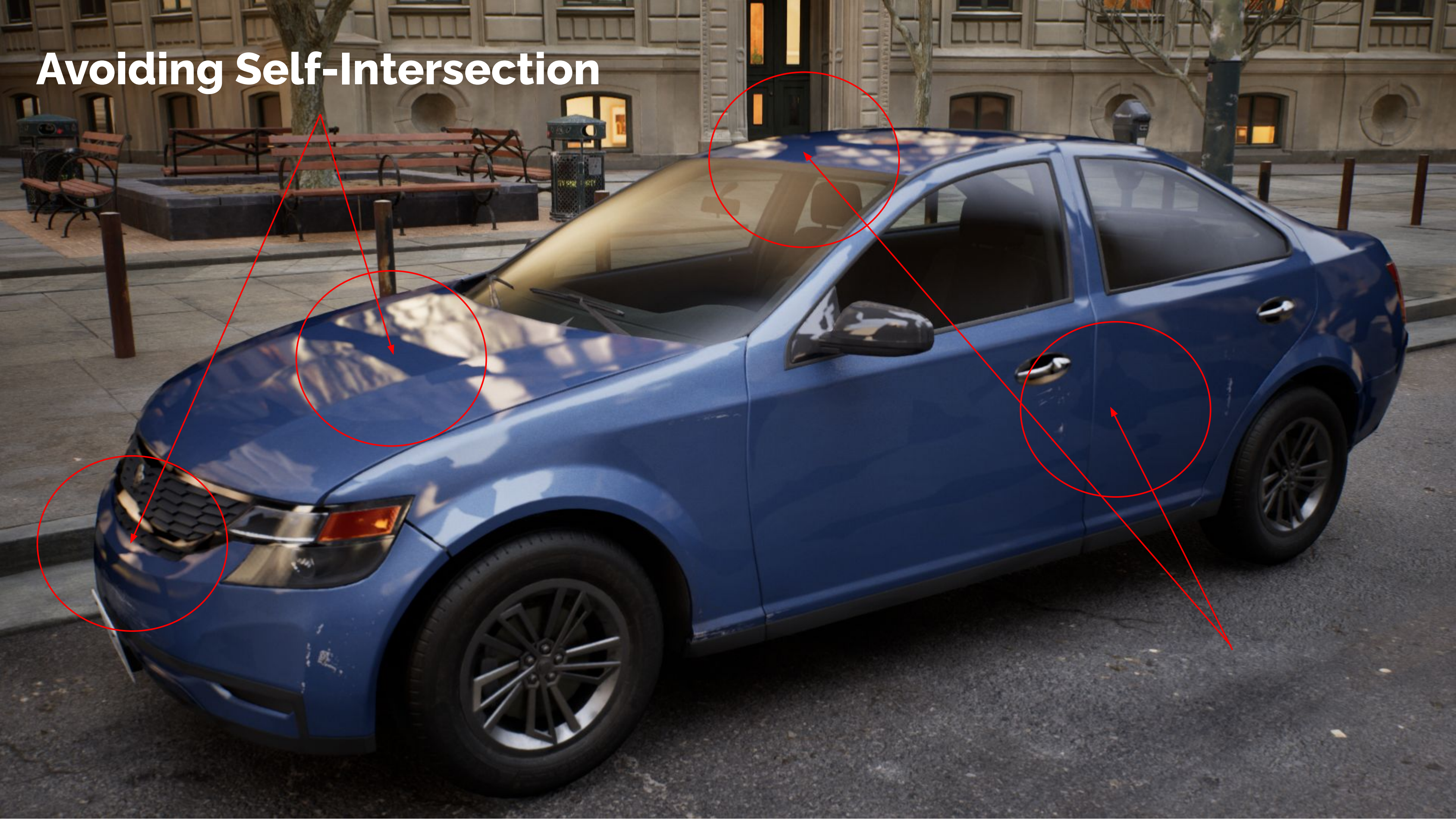
```
Nanite Enabled (Showing Fallback)
Position Precision: 1/64cm (0.0156cm)
GPU Memory: Always allocated 0.03 MB. Streaming 19.62 MB. Total 19.66 MB.
Current Screen Size:  2.023501
Nanite Triangles:  874,074
Nanite Vertices:  497,945
Fallback Triangles:  10,035
Fallback Vertices:  9,354
UV Channels:  2
Distance Field:  35x14x126 = 0.01Mb always loaded, 0.09Mb streamed
Approx Size: 166x70x674
Num Collision Primitives:  2
Estimated Compressed Disk Size: 11.58 MB (11.21 MB Nanite)
```

The vast majority of assets in the Matrix Awakens are rendered in Nanite. This presents an immediate complication with hardware ray tracing as we do not have the capability to support native Nanite geometric resolution in our acceleration structures. There are multiple reasons for this, and I have highlighted a few. The simple answer is that high-quality Nanite support for ray tracing is still an active area of research.

Instead, we must make some concessions with approximate geometric representations. We use Nanite fallback meshes as simplified representations of the rasterized geometry and store them in bottom-level acceleration structures. These fallback meshes bring their own set of challenges, as they do not provide any topological guarantee with the base mesh.

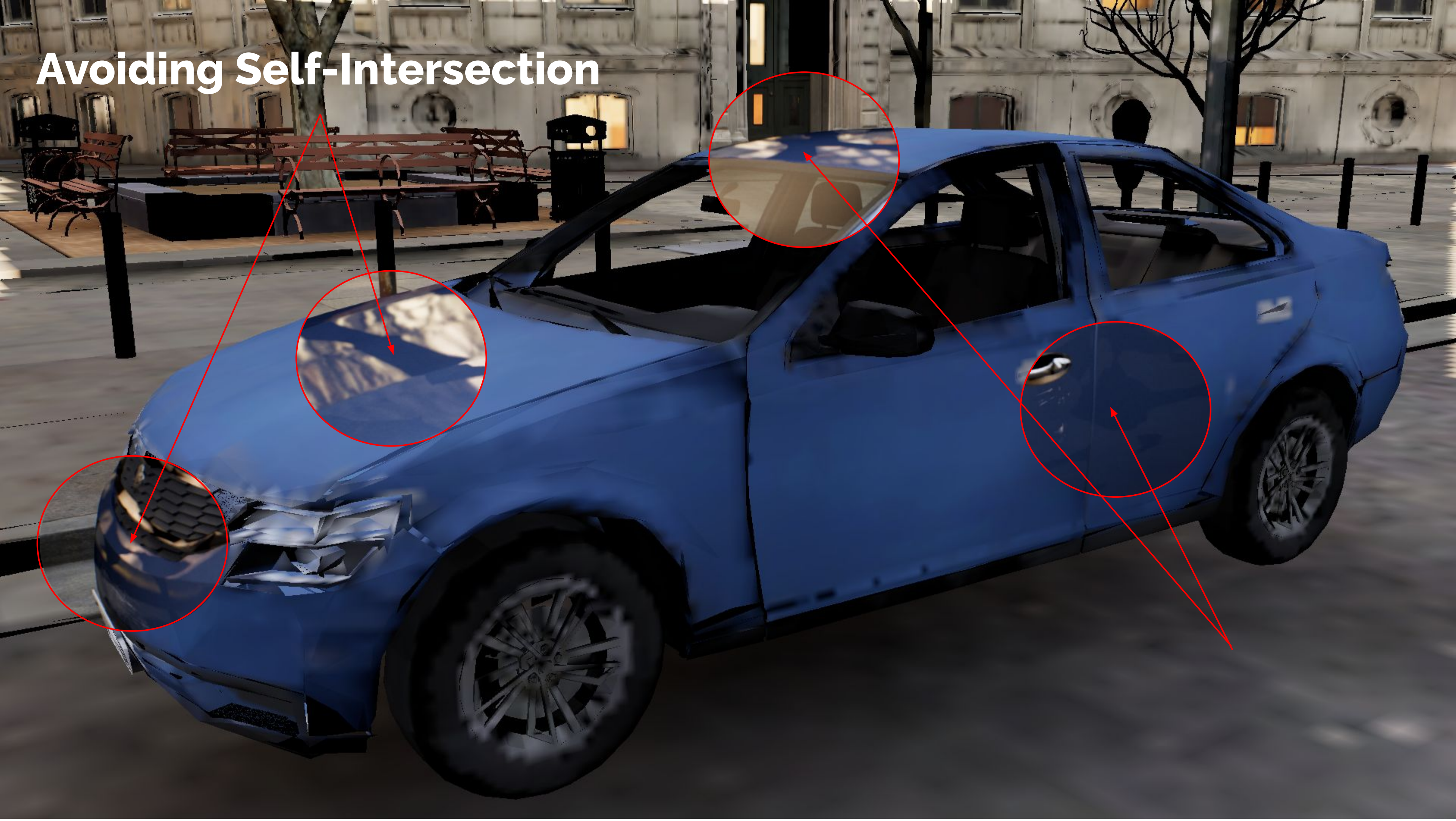
Tracing directly from the GBuffer presents some obvious challenges when working with Nanite fallback meshes. The approximate shape creates the potential for self-intersections that cannot easily be overcome with a traditional ray bias.

# Avoiding Self-Intersection



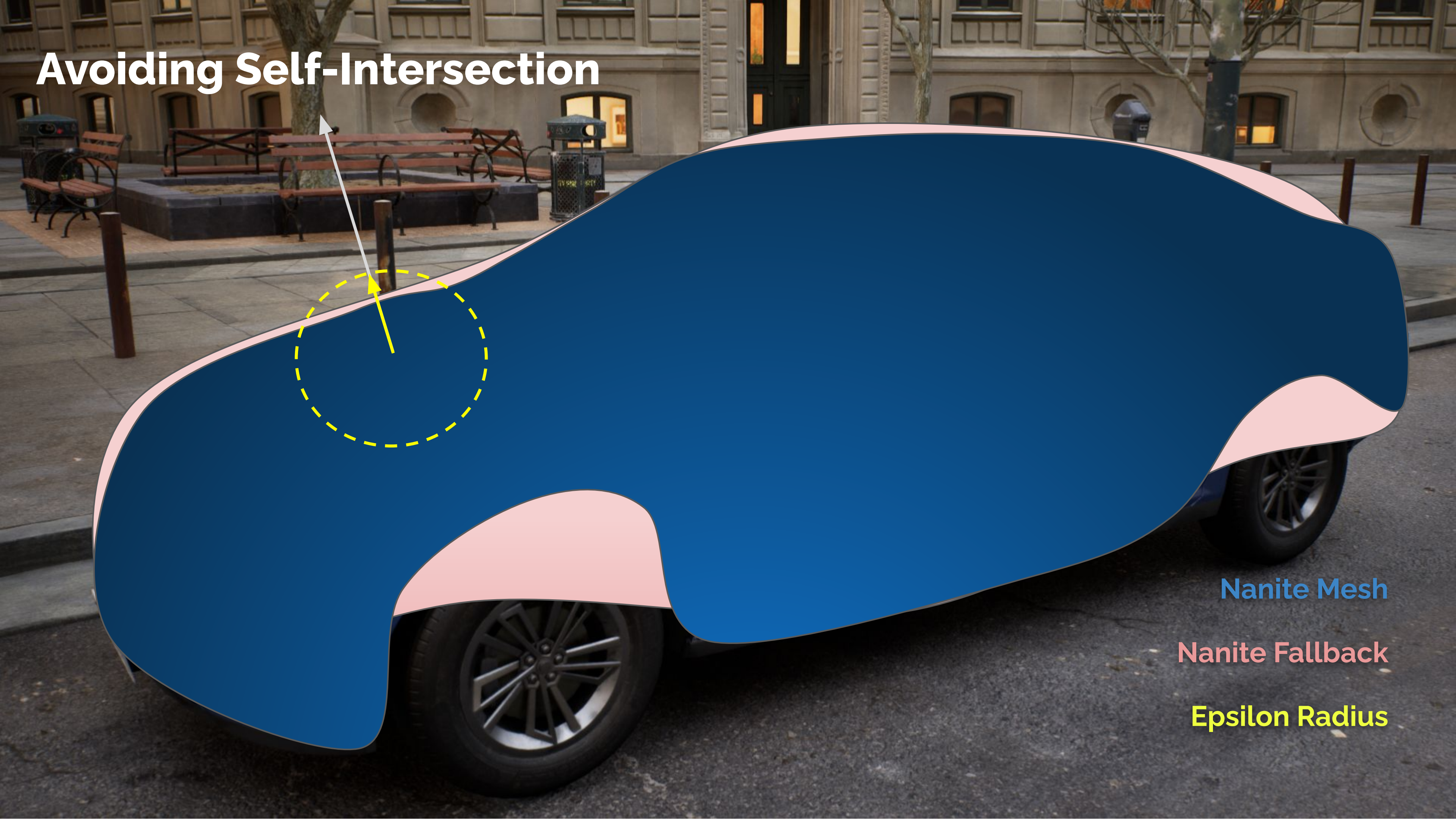
Here is an example of the common self-intersection artifacts we encountered when rendering the vehicles in the demo. I have specifically disabled screen traces to show the prevalence of the artifacts when relying only on the ray-tracing scene.

# Avoiding Self-Intersection



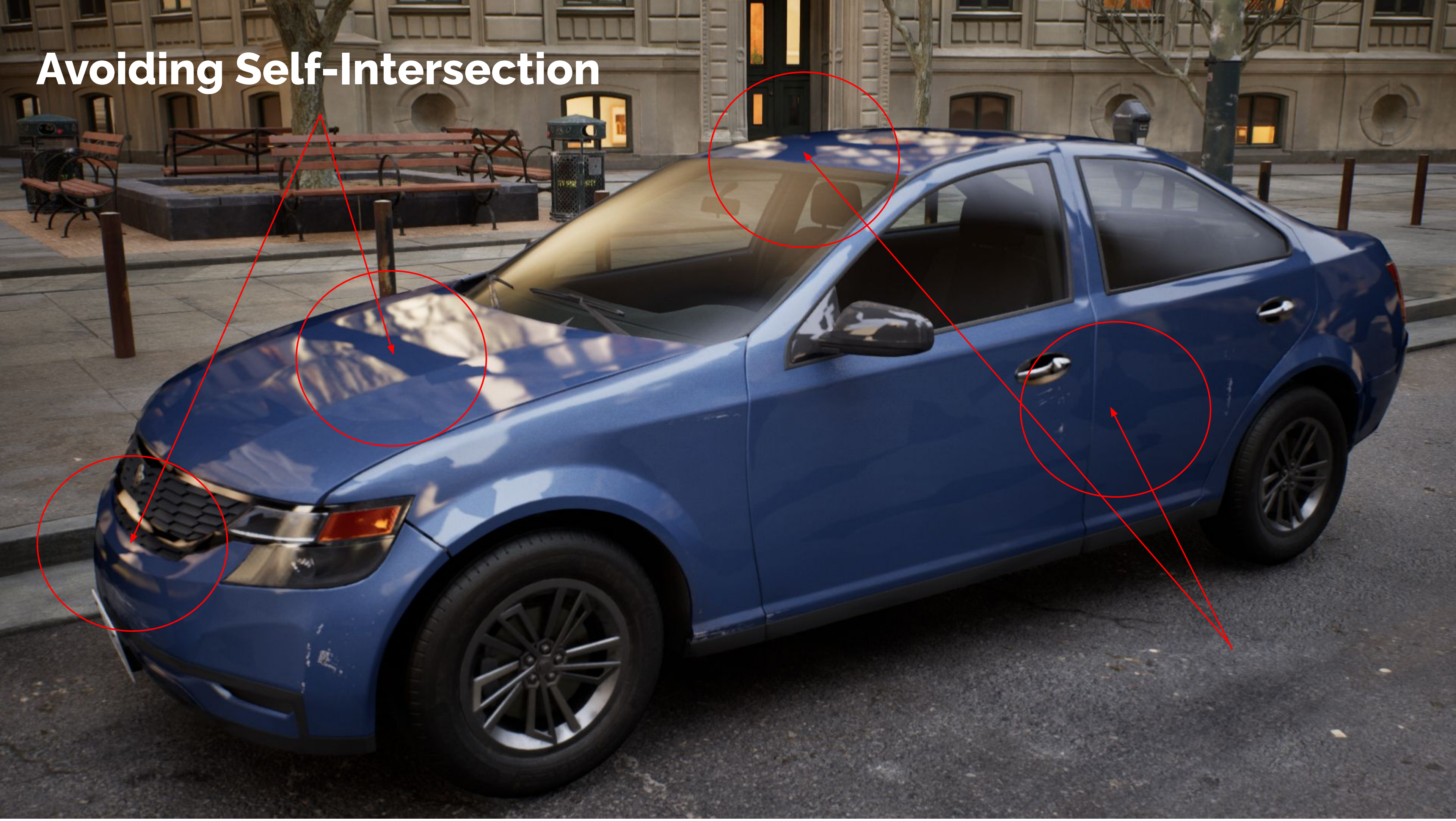
Here is the same shot overlaid with the LumenScene representation to give you an idea of the type of geometric mismatch we must overcome.

# Avoiding Self-Intersection

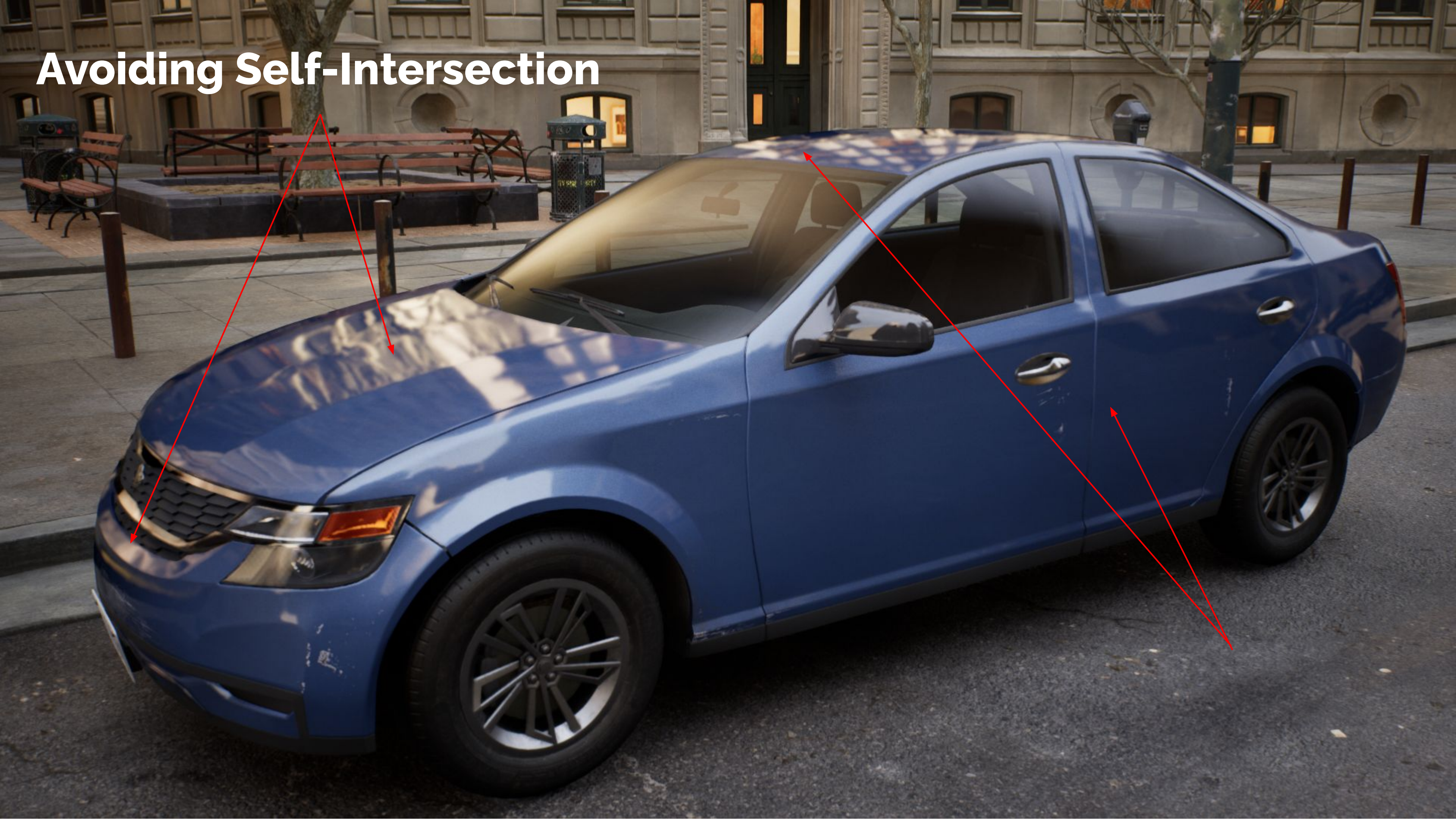


Ray tracing against multiple geometric levels-of-detail is not a new problem, however. Tabellion and Lamorlette presented a solution for this issue back in 2004 [Tabellion et al 2004]. Unfortunately, a proper implementation was too expensive for our budget. Instead, we modified our traversal algorithm to first cast a short ray, to some defined epsilon-distance, ignoring back faces. Only after successfully traversing this distance did we cast a long ray, without any sidedness properties. This figure illustrates the process.

# Avoiding Self-Intersection



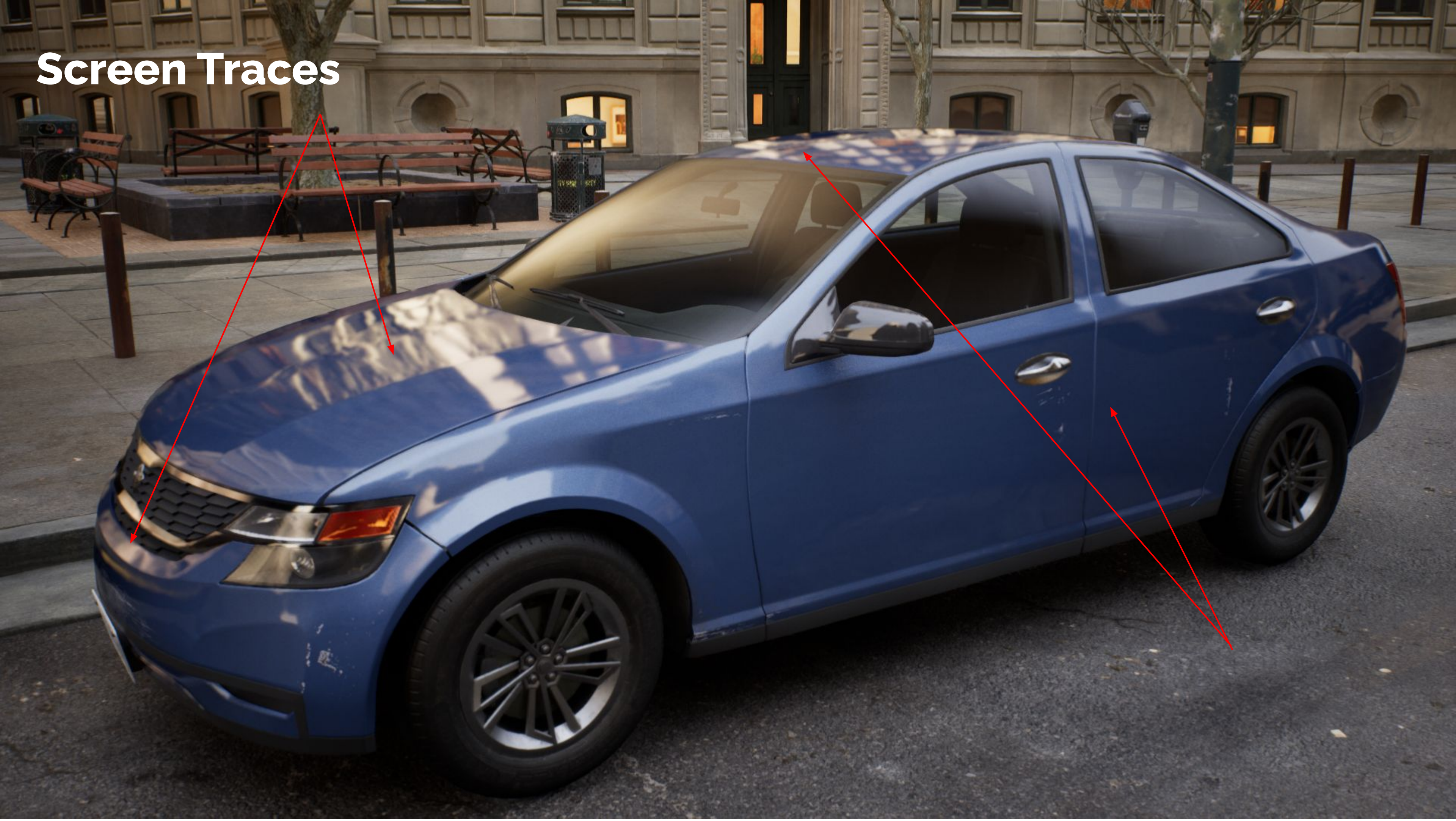
Applying the technique removes the self-intersection artifacts that were present before.



# Avoiding Self-Intersection

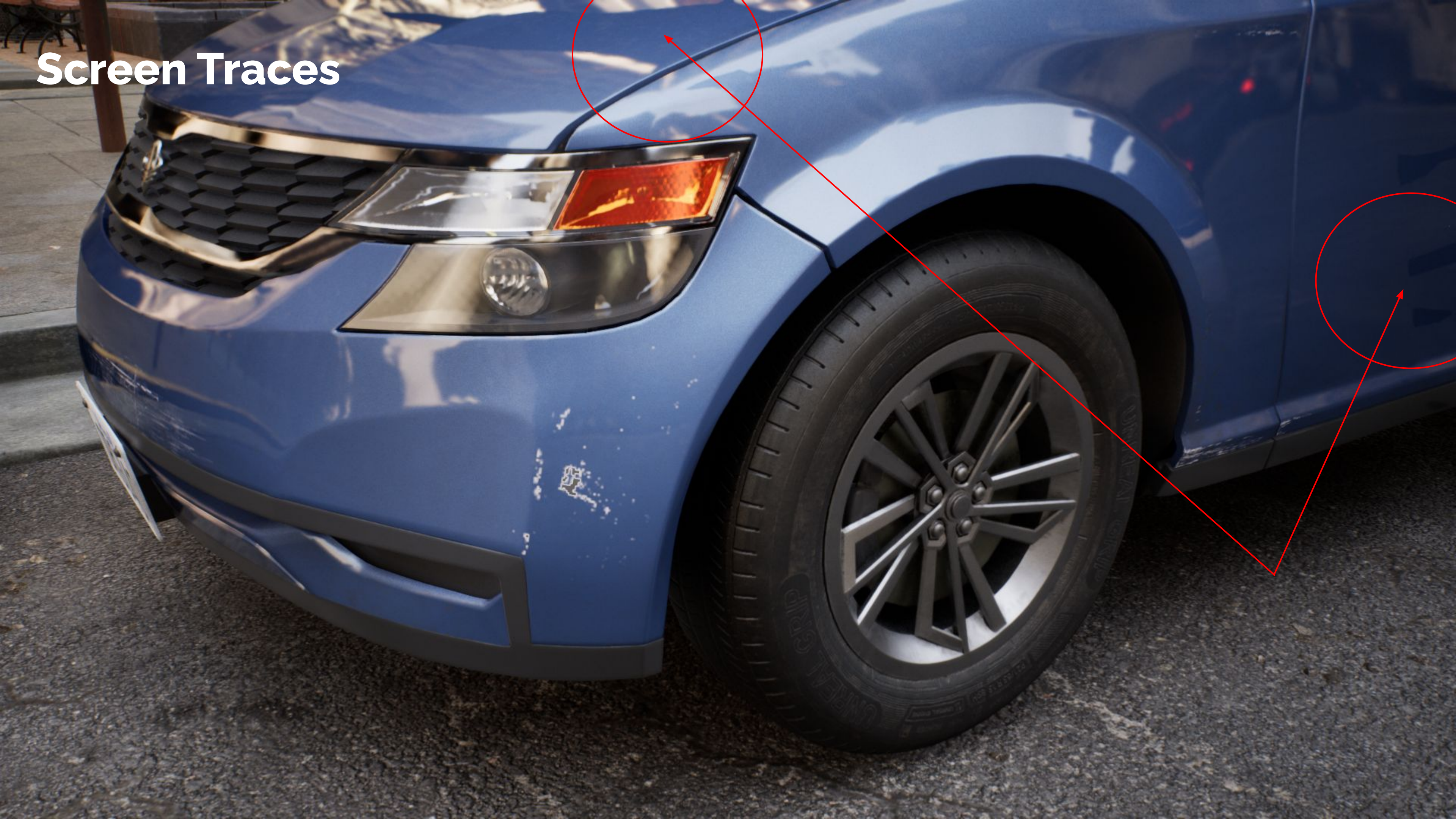
Applying the technique removes the self-intersection artifacts that were present before.

# Screen Traces



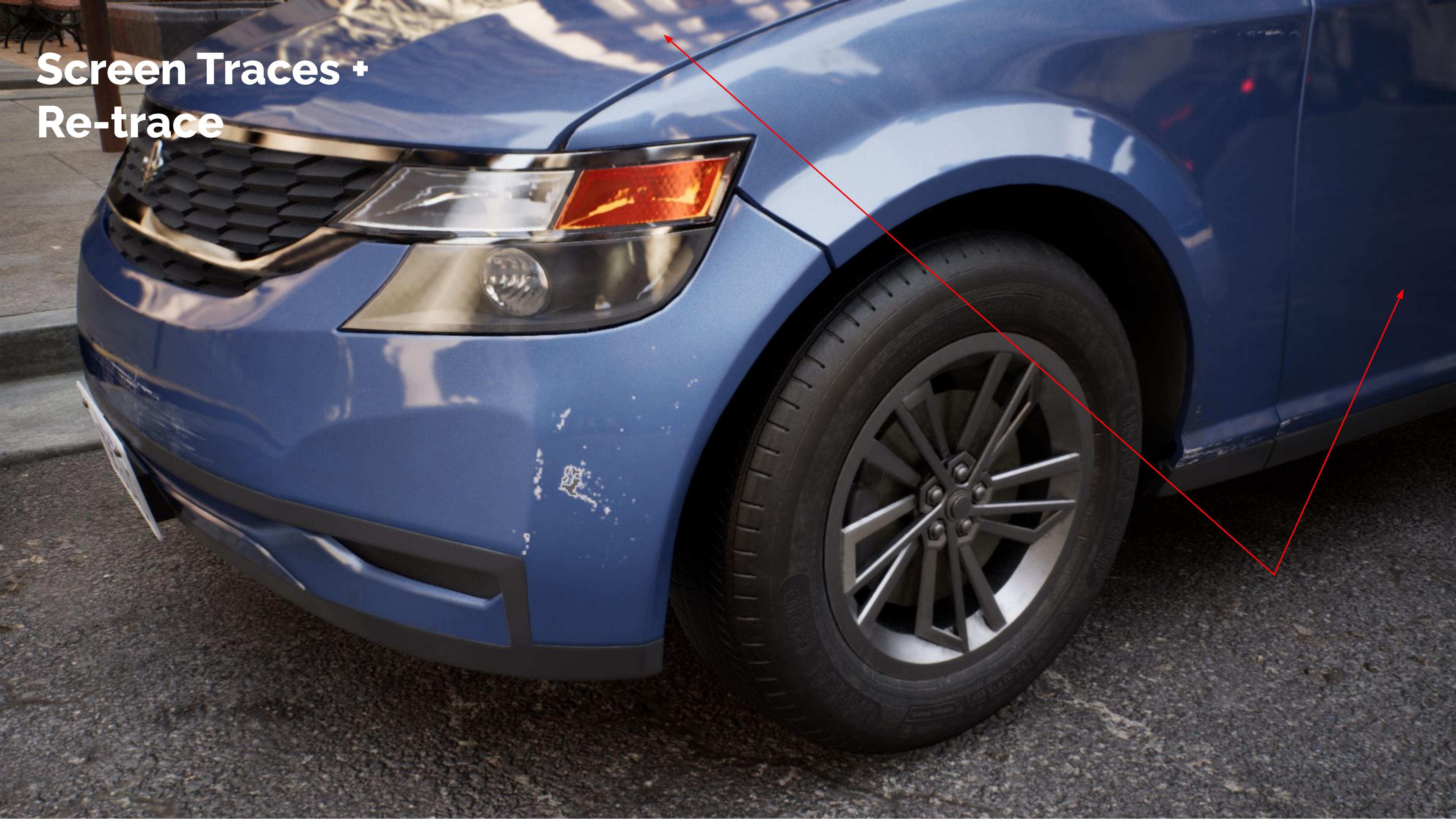
Screen traces also overcome self-intersection artifacts, by providing hardware traversal with a starting t-value that aligns with the bounding view frustum.

# Screen Traces

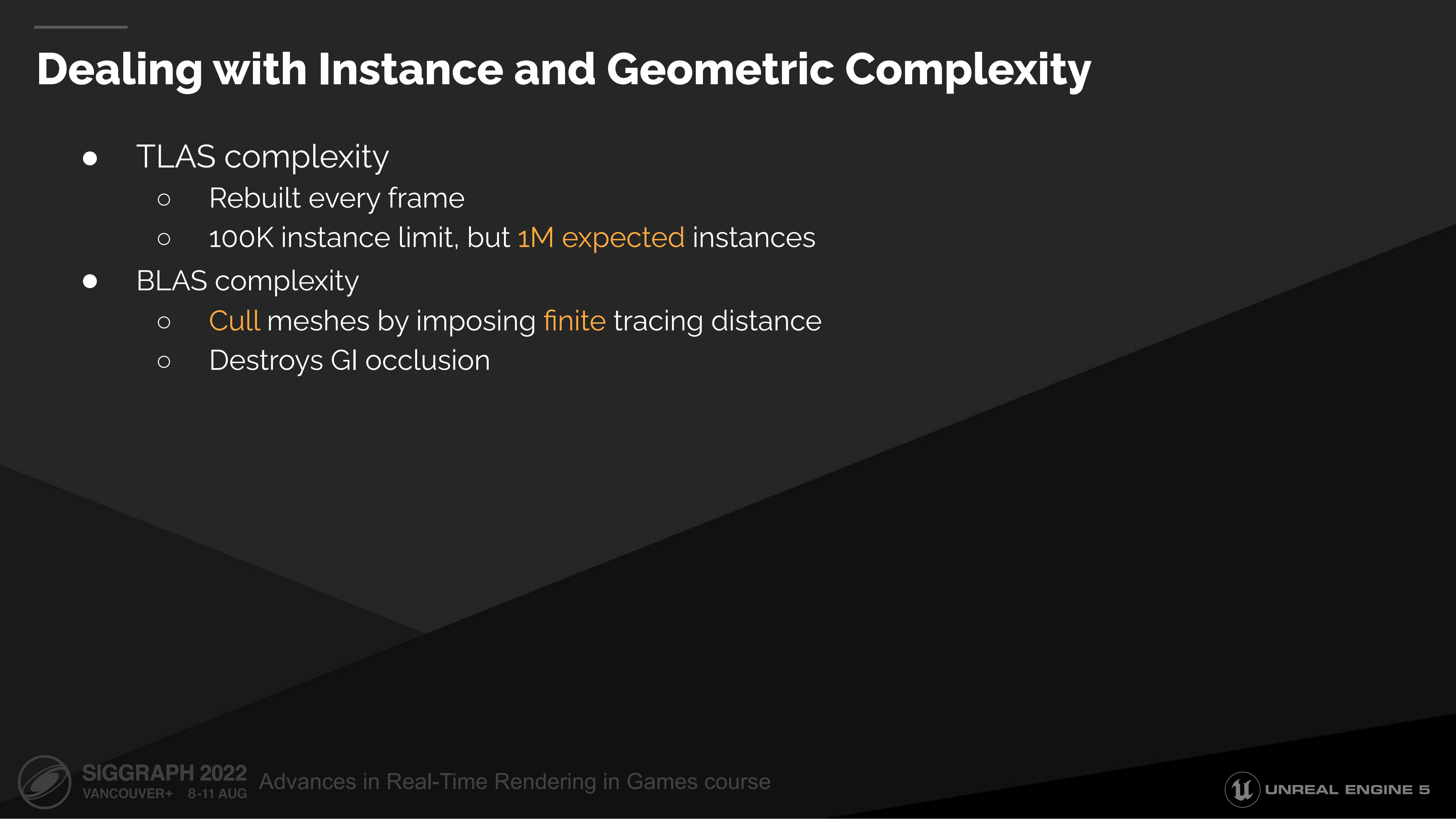


But the inherent bounds of the view frustum make the technique unreliable near the edges of the frame.

# Screen Traces + Re-trace



So we end up using both techniques.



# Dealing with Instance and Geometric Complexity

- TLAS complexity
  - Rebuilt every frame
  - 100K instance limit, but 1M expected instances
- BLAS complexity
  - Cull meshes by imposing finite tracing distance
  - Destroys GI occlusion

As mentioned earlier, the sheer number of active instances in the demo placed significant strain on the hardware ray-tracing model. Our system rebuilds the top-level acceleration structure every frame and early experiments suggested that our model would be limited to 100K instances. At the time of the experiment, content dressing was approaching 500k active instances, with the intention of reaching 1M. The rebuild cost was not the only issue, as ray traversal became extremely costly due to the large triangle count.

It was clear that we needed to reign in the maximum allowable trace distance to fit in budget. By aligning the maximum trace distance with the ray tracing mesh culling distance, we could throttle the system to match our initial requirements. We found that limiting trace distance to 200 meters was ultimately required to hit our goal.

However, limiting trace distance had a profoundly negative impact on the overall look. Car reflections no longer showed the skyline in the distance. But more importantly, accurate sky occlusion from the global illumination solver was completely gone.

We needed a solution that could address this leaking.

# Dealing with Instance and Geometric Complexity

- Use World Partition HLOD
  - Merges instances
  - Simplified geometry
  - Invoked at a different rate than raster
- Mutual exclusion of representations enforced with a “far-field” ray-mask

We chose to utilize the world partition system and its HLOD representation to accomplish this goal. Under the HLOD system, meshes are both simplified and merged together, creating amalgamations for geometry in the distance.

In the typical case, HLOD representations are direct replacements for rasterized geometry; however, due to limitations with performant trace distance, we needed to incorporate the HLOD representation before the rasterizer would typically need this substitution. Because of this, we were often presented with two different mesh representations occupying the same space in our top-level acceleration structure.

To overcome this inconsistency, we submit both representations, but force mutual exclusion during ray traversal with a ray-mask. We tag the HLOD meshes as belonging to the "far-field" set of geometry.

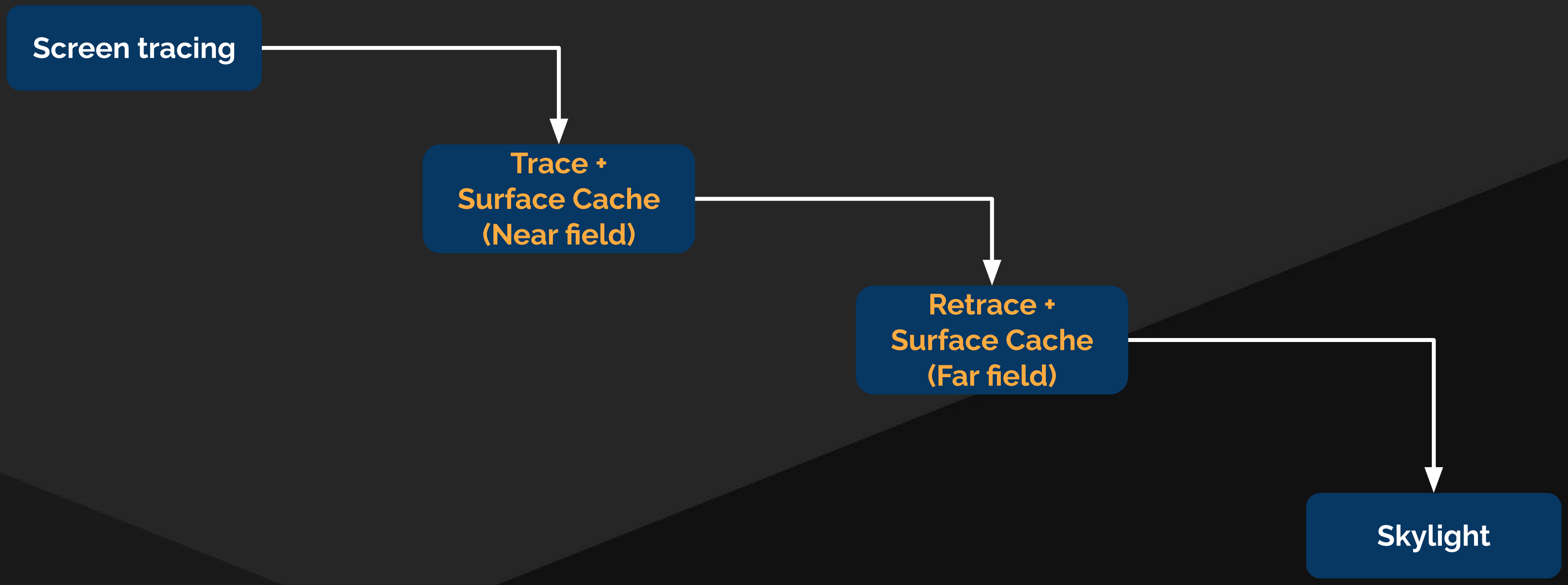
# Tracing with Far-Field

- Ordered traversal
  - Near-field = (t-min, culling distance)
  - Far-field = (culling distance, t-max)
- Unordered traversal
  - Order is reversed, far-field traversal is faster

Ray traversal now comprises the union of traversal operations against both the "near-field" and "far-field" geometry. For ordered traversal, we first trace against the near-field geometry to a distance aligned with our mesh culling distance. Rays that miss the near-field are re-queued for traversal against the far-field.

We reverse the order of operations for shadow rays that do not require ordered traversal. We choose to do this because the far-field representation contains fewer instances and less geometry than the near-field. And this makes far-field faster to traverse.

# Tiered-tracing Pipeline



Screen tracing

Trace + Surface Cache (Near field)

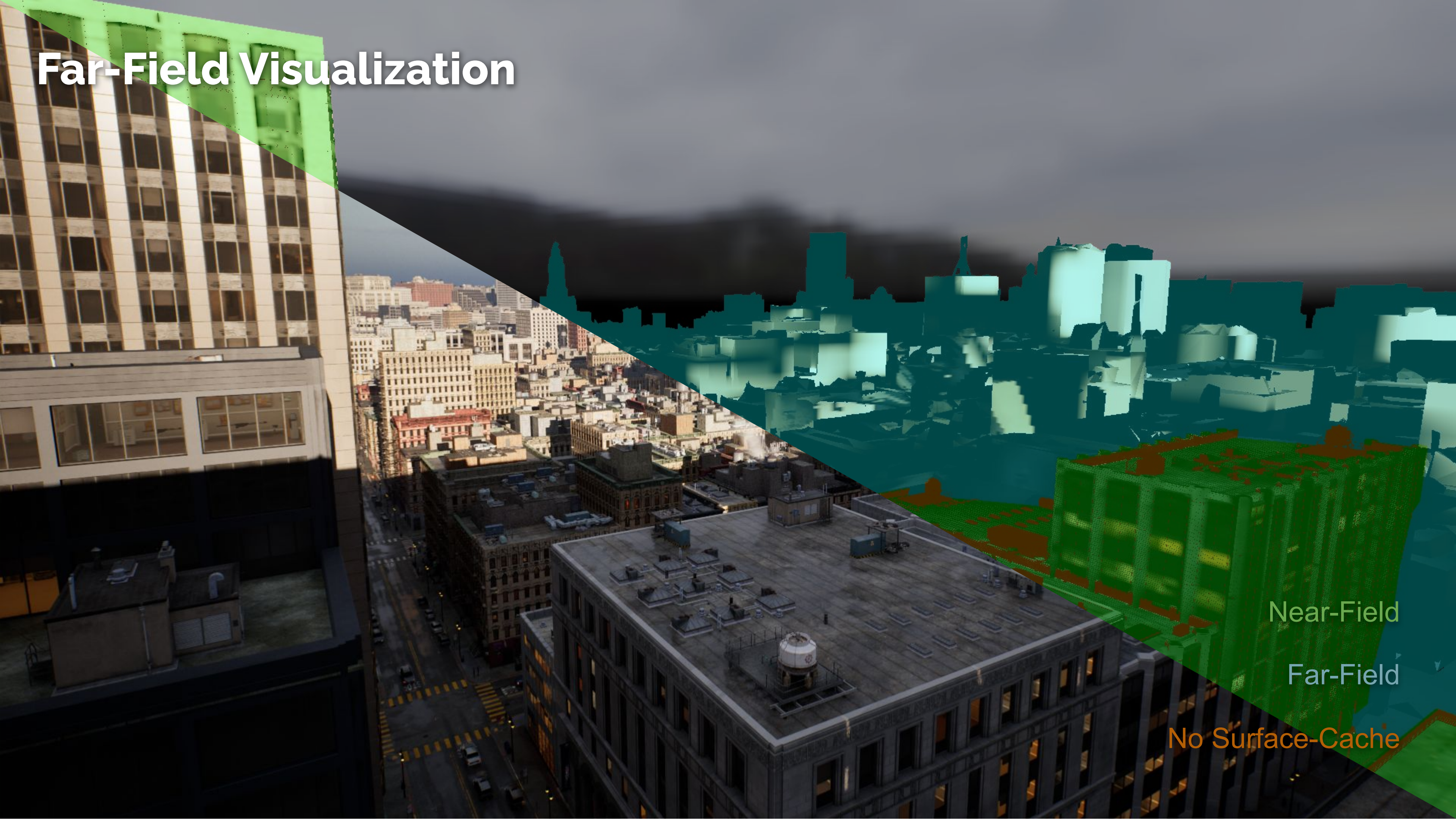
Retrace + Surface Cache (Far field)

Skylight

Incorporation of far-field traces into ordered traversal places a new set of stages into our original hardware traversal pipeline. We add an intermediary compaction step, collating near-field misses into new ray-tiles, and then a subsequent indirect dispatch to trace the ray-tiles against the far-field representation.

For now, I have temporarily removed the Hit-Lighting model from our pipeline.

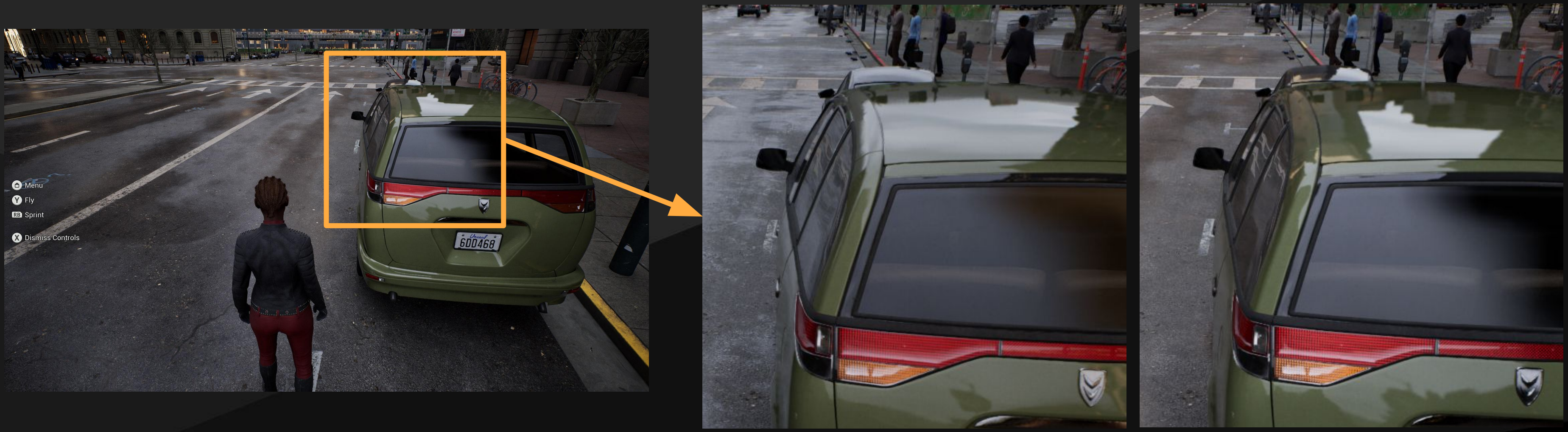
# Far-Field Visualization



This visualization shows the delineation of near-field and far-field geometry. Geometry which does not have a surface cache entry is also highlighted.

# “The Matrix Awakens” - UE5 demo

| PS5, 1080p | | |
|---|---|---|
| | Baseline | Overlapping |
| Near-Field | 1.65 ms | 2.37 ms |
| Far-Field | N/A | 1.04 ms |



Baseline

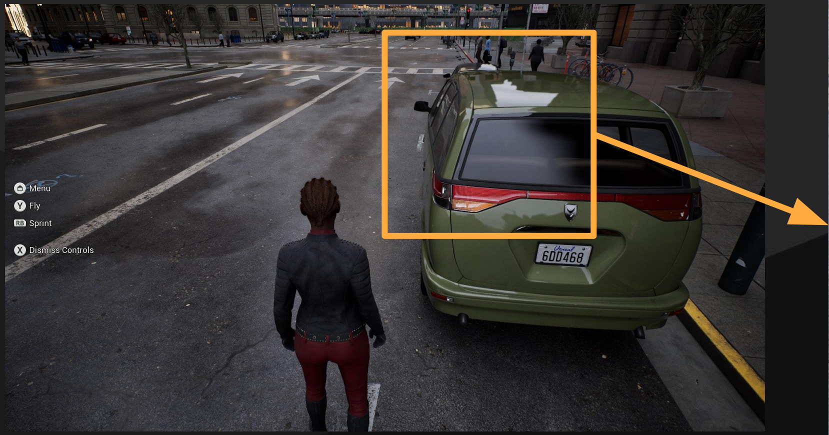
Overlapping

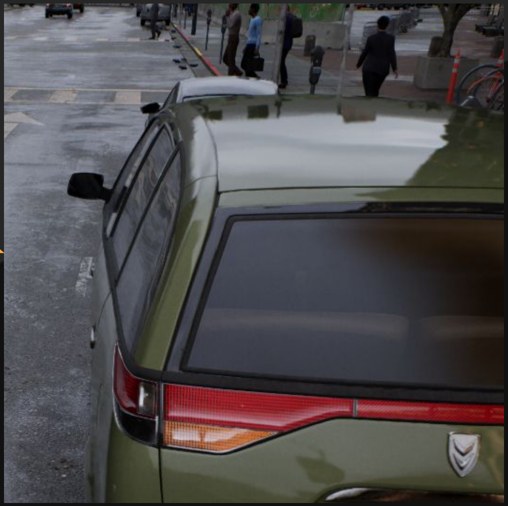
Submitting both near-field and far-field representations to the same top-level acceleration structure is NOT ideal. Doing so creates needless geometric overlap. While the ray-mask removes unnecessary traversal, the damage to the top-level acceleration structure build is substantial. Our early experiments with overlaying the far-field representation revealed a show-stopping 44% penalty to all near-field traversal costs.

A proper solution would be to support multiple top-level acceleration structures, which would also avoid the need to use the ray-mask mentioned previously. We were hesitant to take on this architectural change mid-production under the already aggressive development schedule, but we needed to do something.

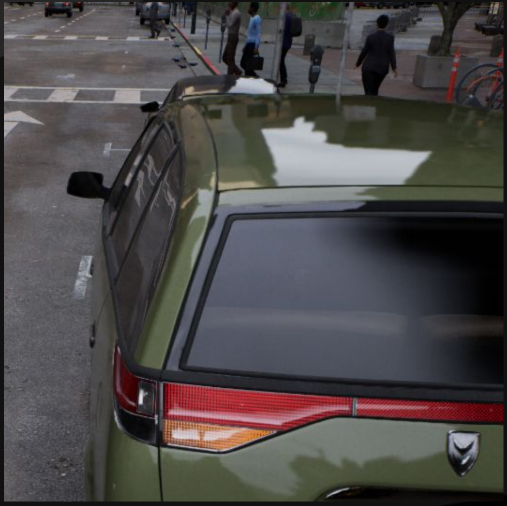
# “The Matrix Awakens” - UE5 demo

| PS5, 1080p | | | |
|---|---|---|---|
| | Baseline | Overlapping | Translated |
| Near-Field | 1.65 ms | 2.37 ms | 1.77 ms |
| Far-Field | N/A | 1.04 ms | 0.60 ms |





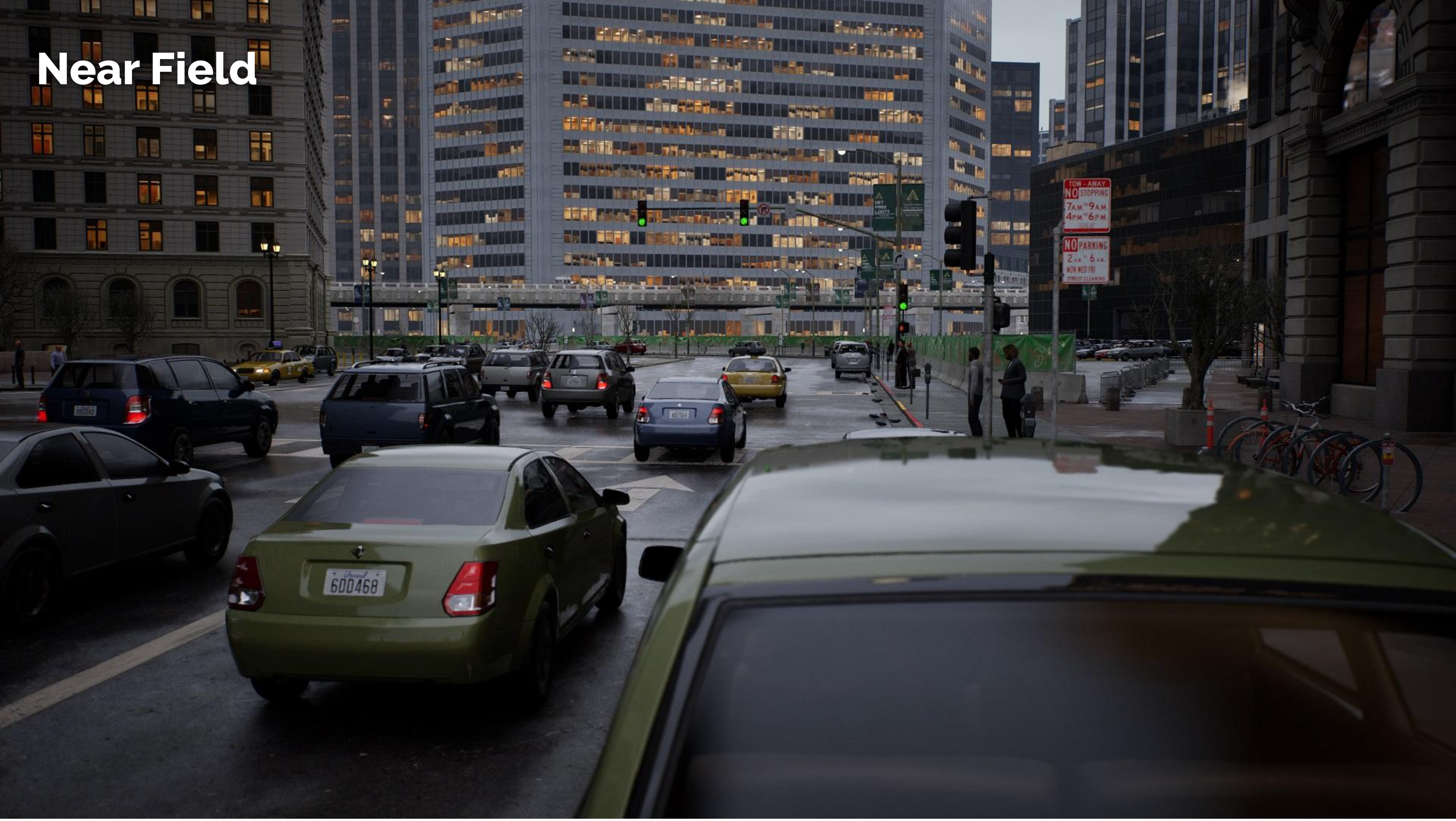
Baseline



Translated

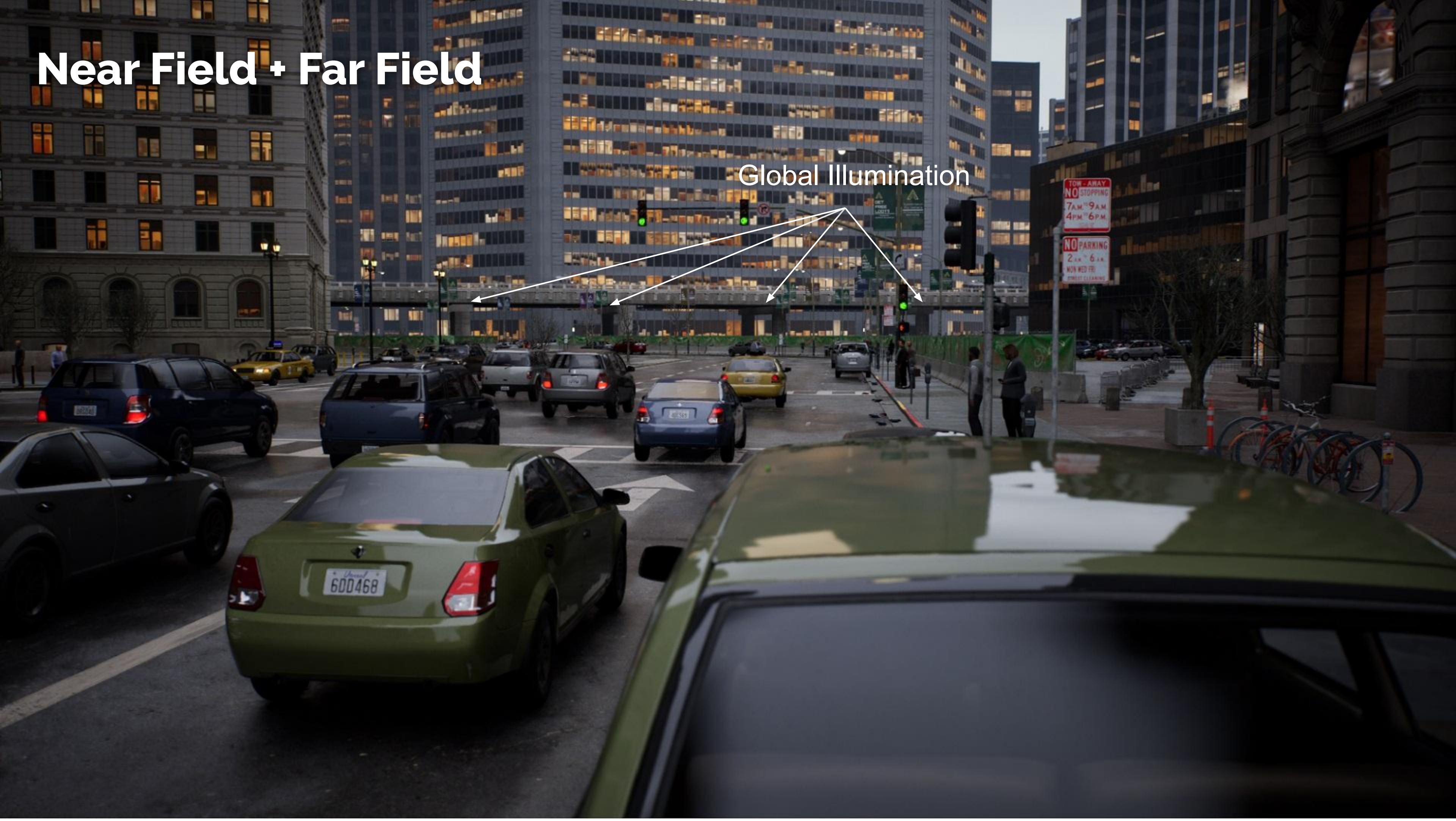
Following an earlier hypothesis, it was suggested that maybe applying a global translational offset to the far-field geometry might alleviate some of the burden. So we gave this idea a shot. In the end, we do suffer a penalty for incorporating the geometry within the same acceleration structure, but the translational offset significantly reduces the overhead cost.

# Near Field

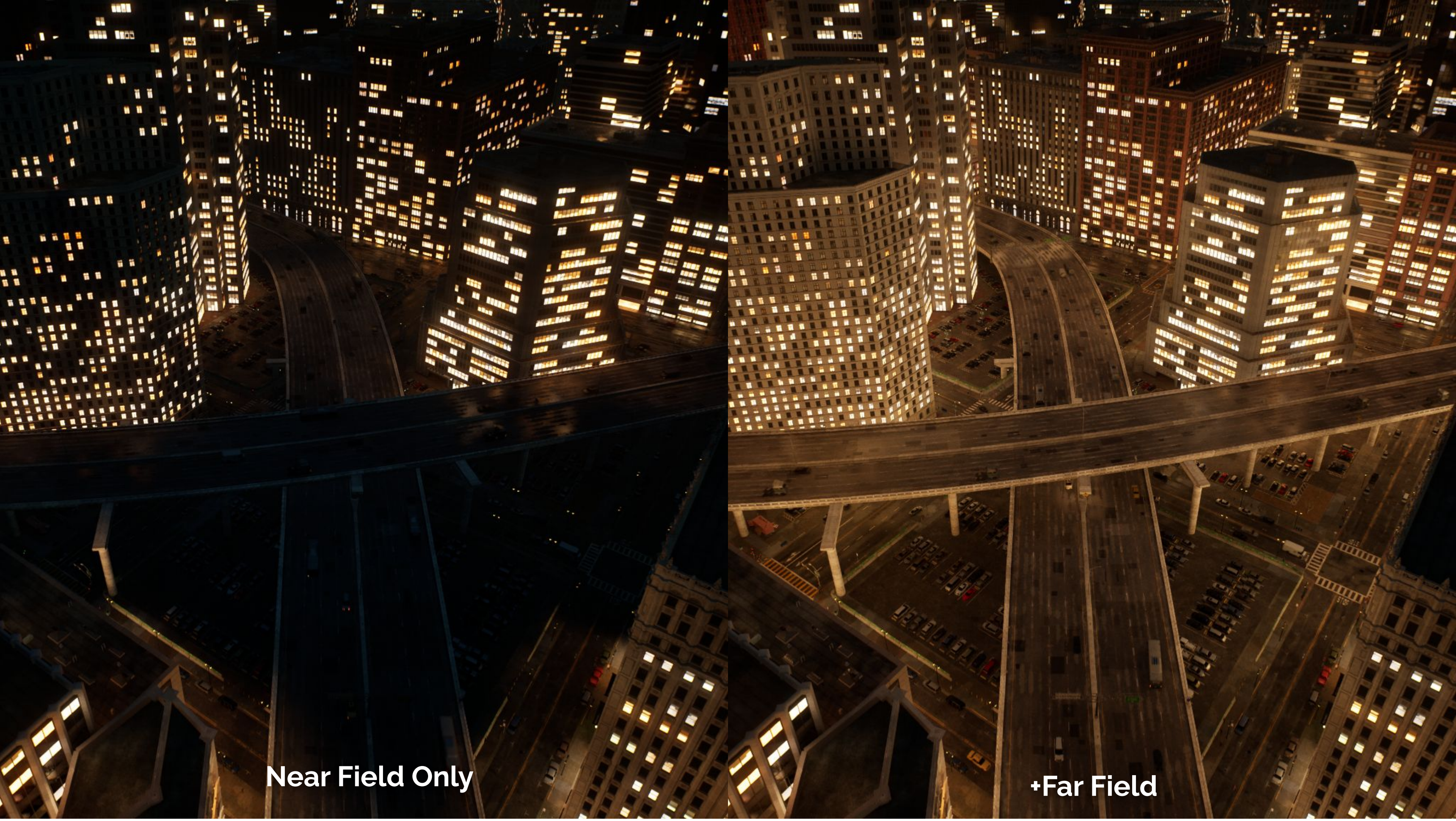


This animation shows the application of far-field tracing to the entire set of Lumen passes. While reflection occlusion has the most striking influence, you can see its impact to the global illumination contribution in the distance.

# Near Field + Far Field



This animation shows the application of far-field tracing to the entire set of Lumen passes. While reflection occlusion has the most striking influence, you can see its impact to the global illumination contribution in the distance.

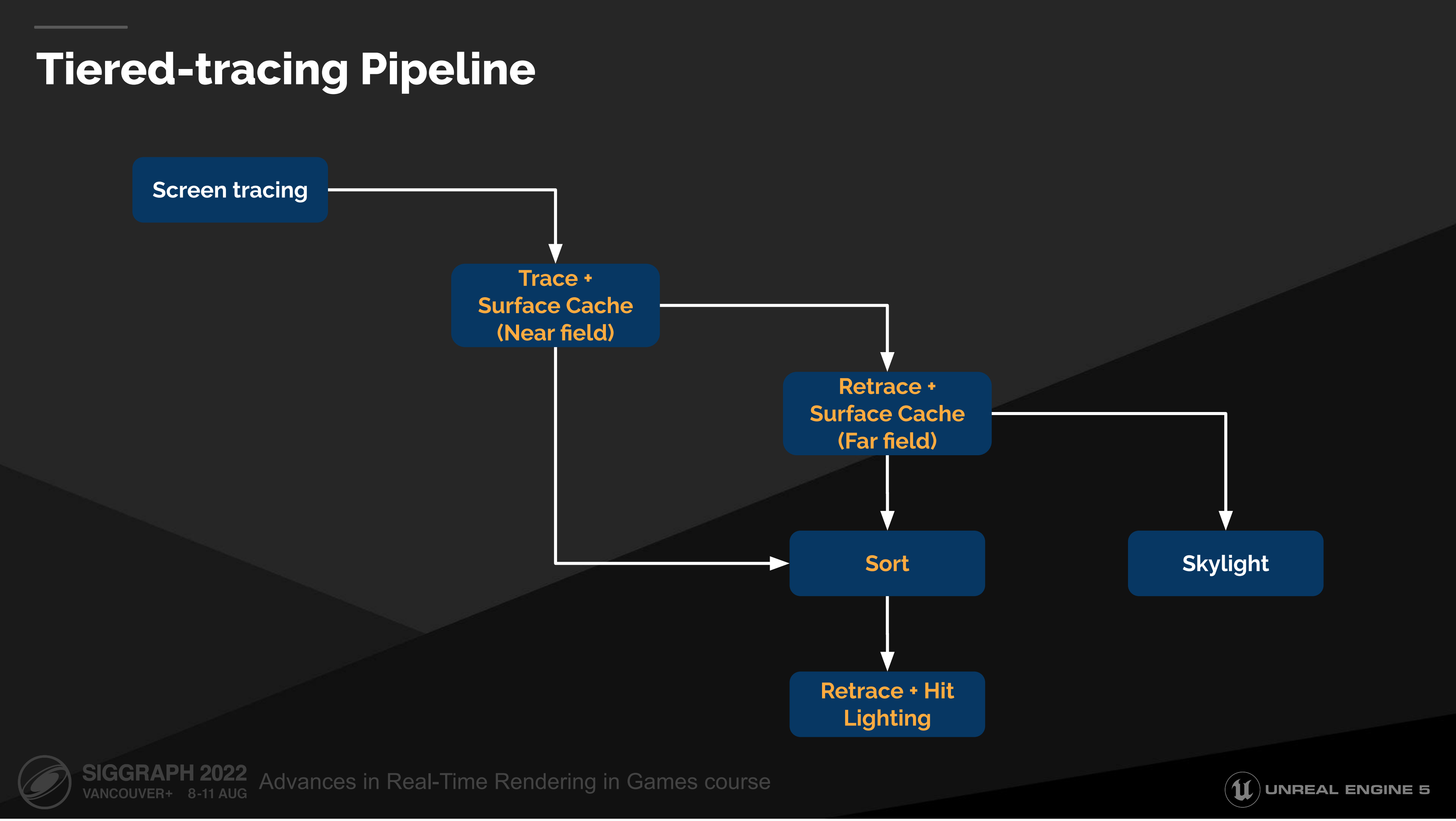


Here is a more obvious depiction of the quality improvement with the addition of the far-field tracing stage.

# Tiered-tracing Pipeline

- Shading models
  - Surface Cache
  - Hit-Lighting
- Traversal models
  - Near-field
  - Far-Field

At this point, I have discussed all of the building blocks necessary to reveal our final tiered-traversal pipeline. To provide a brief recap, we present the artist with two shading models: the Surface-Cache model for speed, and the Hit-Lighting model for quality. We also provide a mechanism to gracefully cascade from a high-quality, near-field geometric representation to a low-quality, but more performant, fear-field geometric representation in the distance.



# Tiered-tracing Pipeline

Screen tracing

Trace + Surface Cache (Near field)

Retrace + Surface Cache (Far field)

Sort

Skylight

Retrace + Hit Lighting

With some reordering of stages, we can minimize the dispatch costs by first resolving all Surface-Cache stages before optionally requeuing results for Hit-Lighting. We do this by cascading through both geometric representations. Hits are then compacted and optionally requeued for Hit-Lighting, while misses cascade to apply SkyLight evaluation.

# Summary

- Fast paths with Surface Cache
- Huge view ranges with Far Field
- Robust handling of LOD mismatches

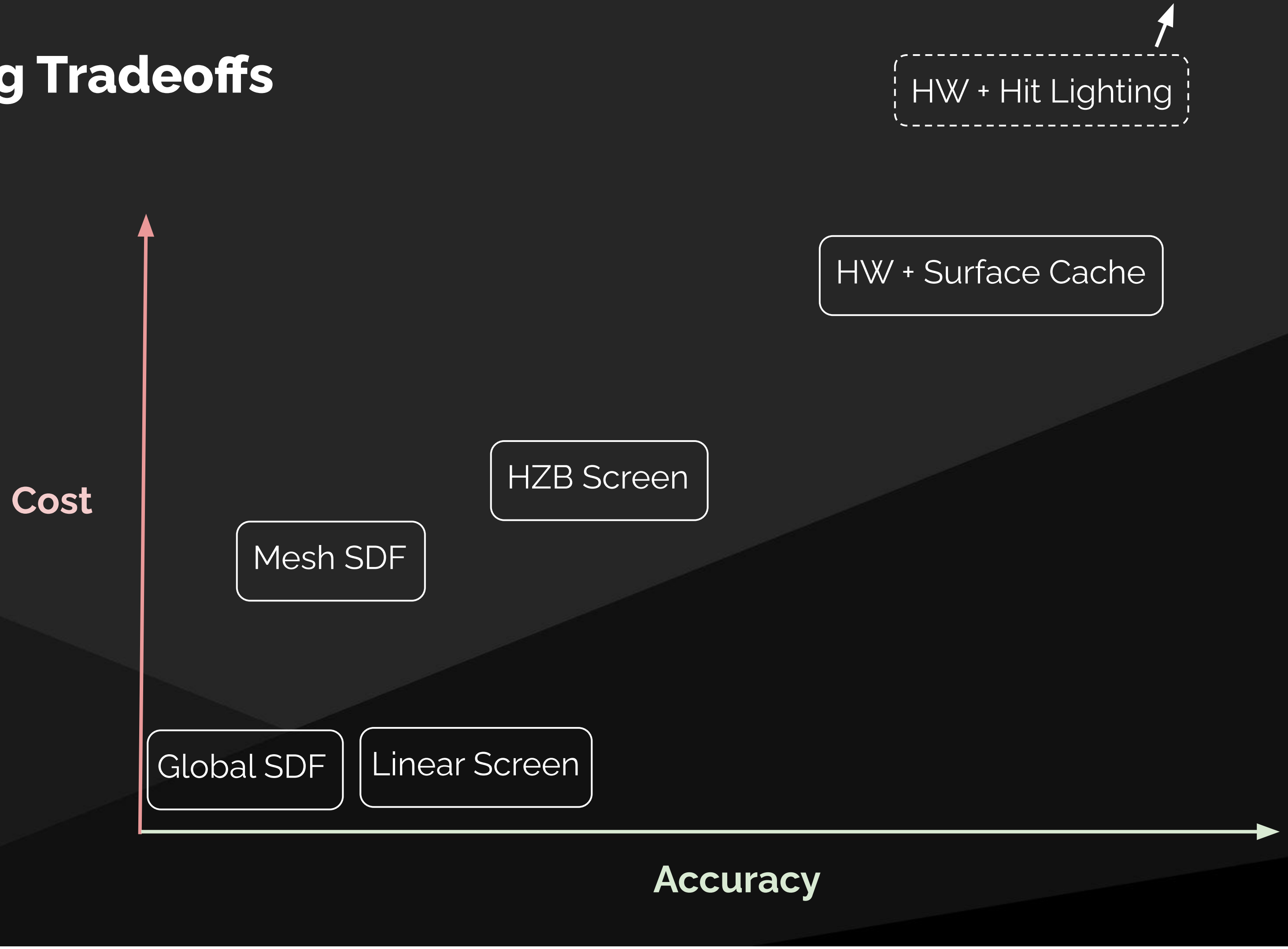
Our hardware tracing model has changed quite a bit from the original UE4 model. We have shown that by utilizing the Surface Cache, we can construct a minimalistic traversal scheme that is designed for performance. We have shown that adding a far-field geometric representation has addressed overwhelming geometric instance complexity, while allowing us to significantly extend our ray traversal distance at the time same time. And we have shown how to deal with inherent geometric LOD mismatches when incorporating complicated Nanite assets.

Thanks for watching. I'll now hand it back to Daniel for the remainder of our talk.

# Tracing Performance

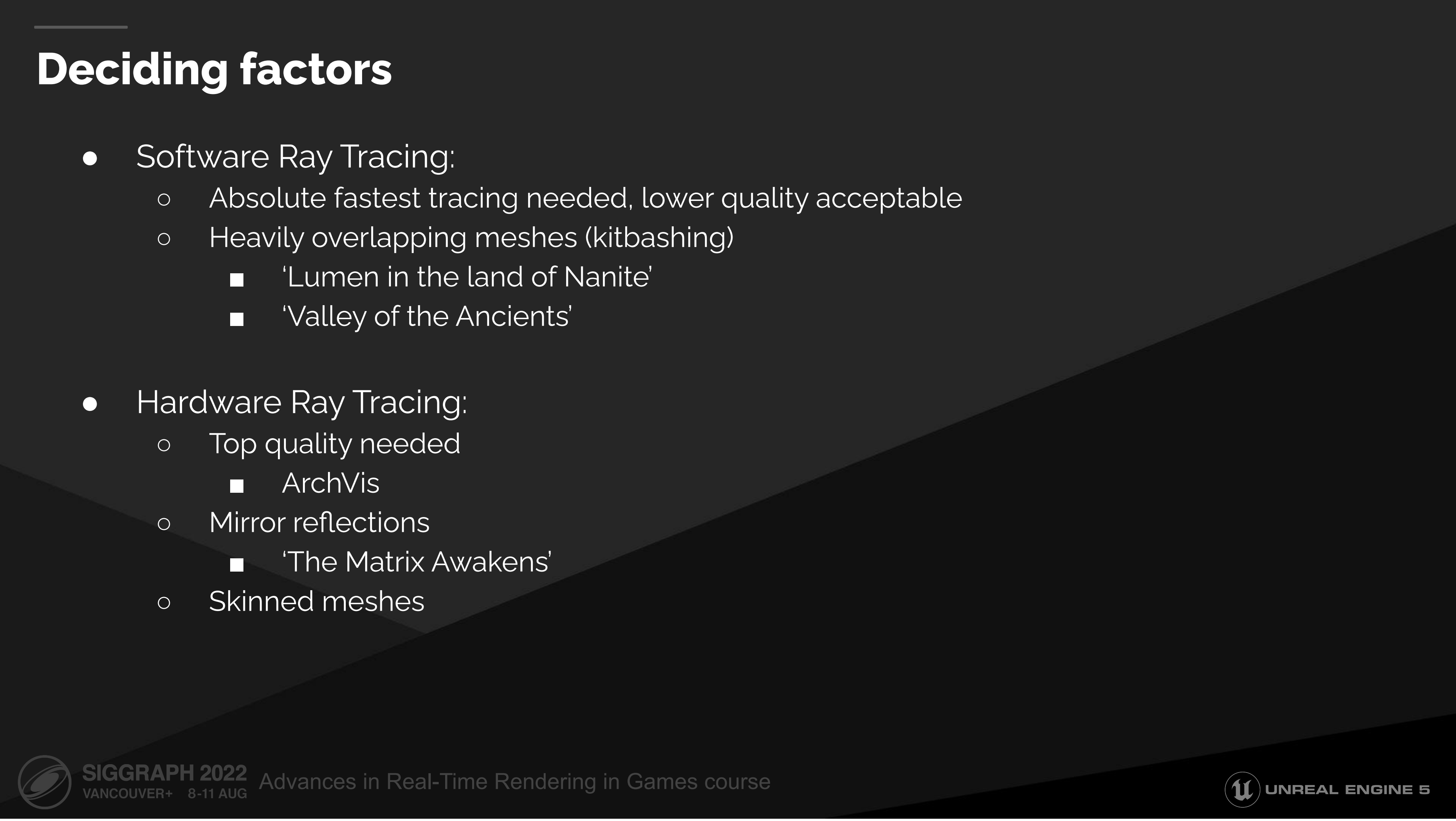
So now I’m going to wrap up the Ray Tracing Pipeline section with a performance comparison of the tracing methods.

# Tracing Tradeoffs



If we graph them all by cost and accuracy, we can see that the Global Distance Field tracing is the fastest, but also the least accurate, and it needs to be complemented with a more accurate method, like the screen traces, or Mesh Distance Field tracing.

Hardware ray tracing is very accurate, but also very expensive, and doesn't really have a way to scale down. Hardware ray tracing with hit lighting is so expensive that it's off the graph in cost.



## Deciding factors

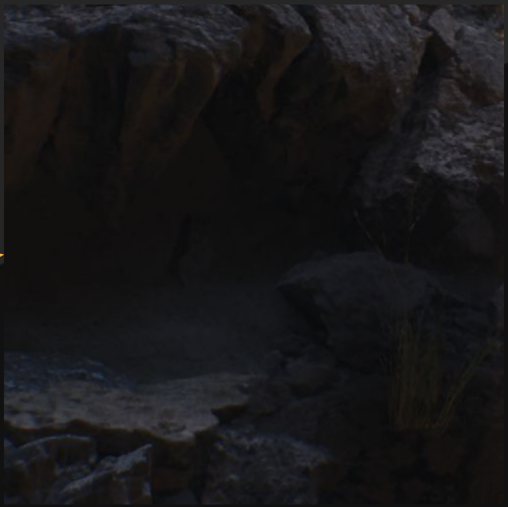
- Software Ray Tracing:
  - Absolute fastest tracing needed, lower quality acceptable
  - Heavily overlapping meshes (kitbashing)
    - ‘Lumen in the land of Nanite’
    - ‘Valley of the Ancients’
- Hardware Ray Tracing:
  - Top quality needed
    - ArchVis
  - Mirror reflections
    - ‘The Matrix Awakens’
  - Skinned meshes

Projects that use Lumen have to choose which tracing method they’re going to use. Software Ray Tracing is the best choice for projects that need the absolute fastest tracing possible, like 60 frames per second on next generation consoles. Projects that have lots of overlapping meshes built using kitbashing should also use Software Ray Tracing, which was the case in our tech demos ‘Lumen in the land of Nanite’ and the ‘Valley of the Ancients’.

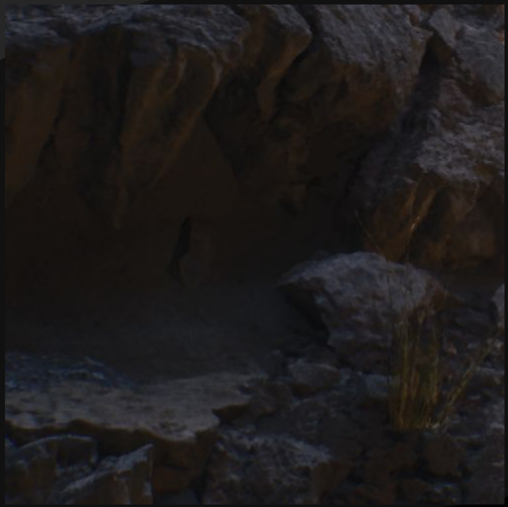
Projects should use Hardware Ray Tracing if they need the absolute top quality possible, like Architectural Visualization. Projects should also use Hardware Ray Tracing if they need mirror reflections, like in ‘The Matrix Awakens’, or skinned meshes affecting the indirect lighting in a significant way.

# “Lumen in the Land of Nanite” - UE5 demo

| PS5, 1080p | | |
|---|---|---|
| | Global SDF | HWRT |
| Final Gather tracing | 0.94 ms | 10.03 ms |
| Reflection tracing | 0.04 ms | 0.46 ms |



Global SDF

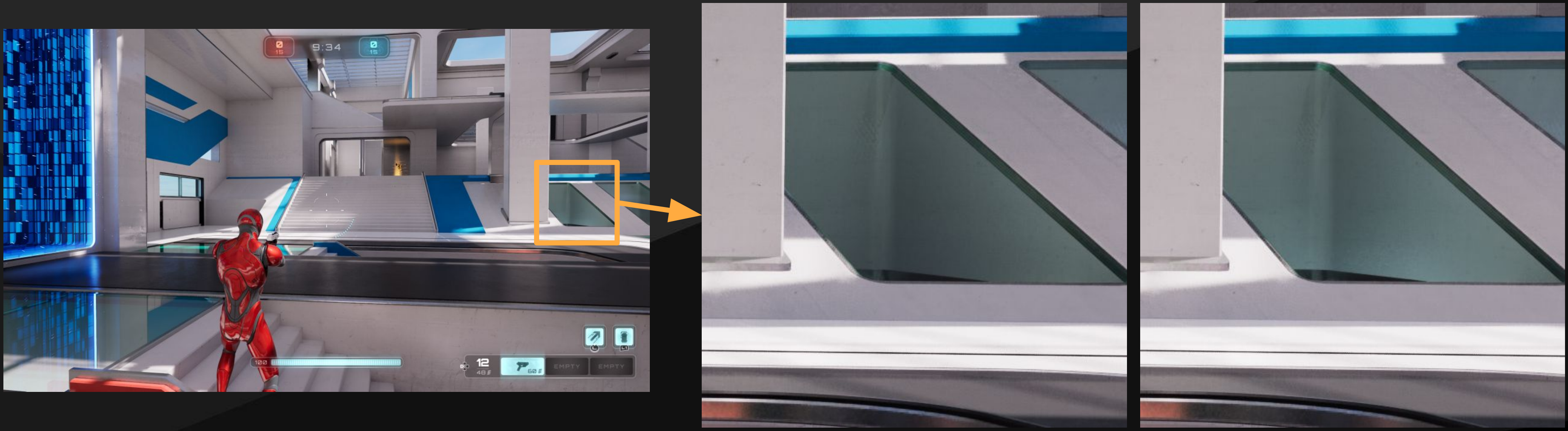


HWRT

So let’s look at the performance of the two tracing methods in a couple of different scenes, starting with ‘Lumen in the land of Nanite’, which had a huge number of overlapping meshes. This content was effectively built as a Nanite stress test, and every single point along the cave surface has about a hundred meshes overlapping it. With Hardware Ray Tracing, the ray has to traverse every single overlapping mesh, whereas the Software Ray Tracing has a fast merged version. Hardware Ray Tracing is so expensive in this content that we could have never shipped with it.

# “Lyra” - UE5 sample game

| PS5, 1080p | | |
|---|---|---|
| | Global SDF | HWRT |
| Final Gather tracing | 1.21 ms | 1.41 ms |
| Reflection tracing | 1.58 ms | 2.03 ms |



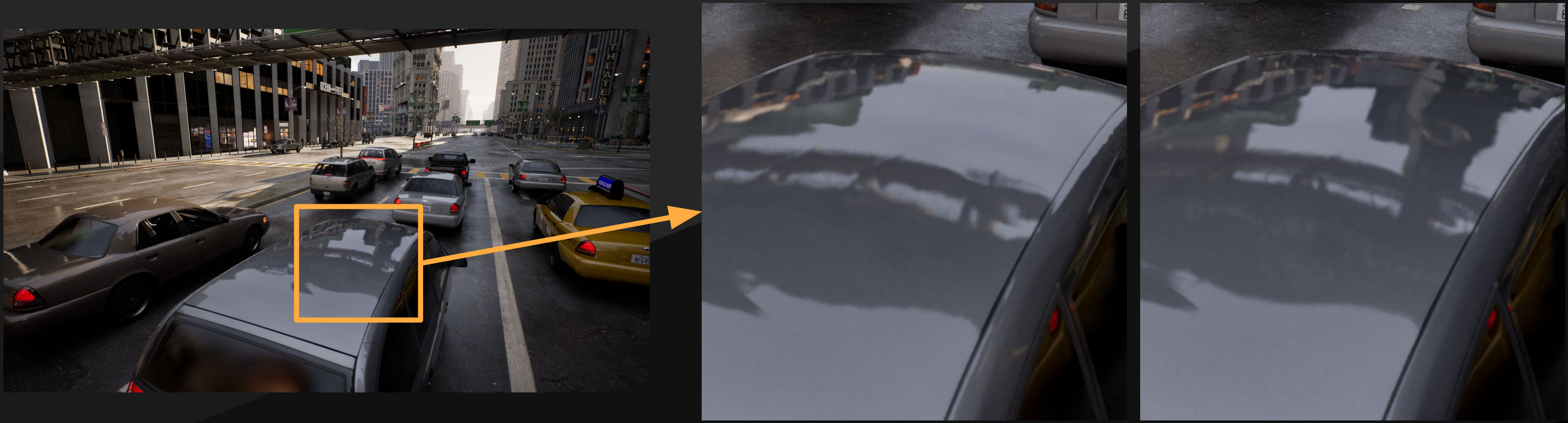
Global SDF

HWRT

In the Lyra UE5 sample game, it’s a different situation. The geometry here is not really overlapping, so Hardware Ray Tracing performs very well. The cost difference between the two techniques is not that large, and neither is the quality difference. It could really go either way, depending on what the hardware supports.

# “The Matrix Awakens” - UE5 demo

| PS5, 1080p | | | |
|---|---|---|---|
| | SWRT | HWRT | HWRT with Far Field |
| Final Gather tracing | 1.83 ms | 1.72ms | 2.13 ms |
| Reflection tracing | 1.76 ms | 1.79ms | 2.52 ms |



SWRT

HWRT with Far Field

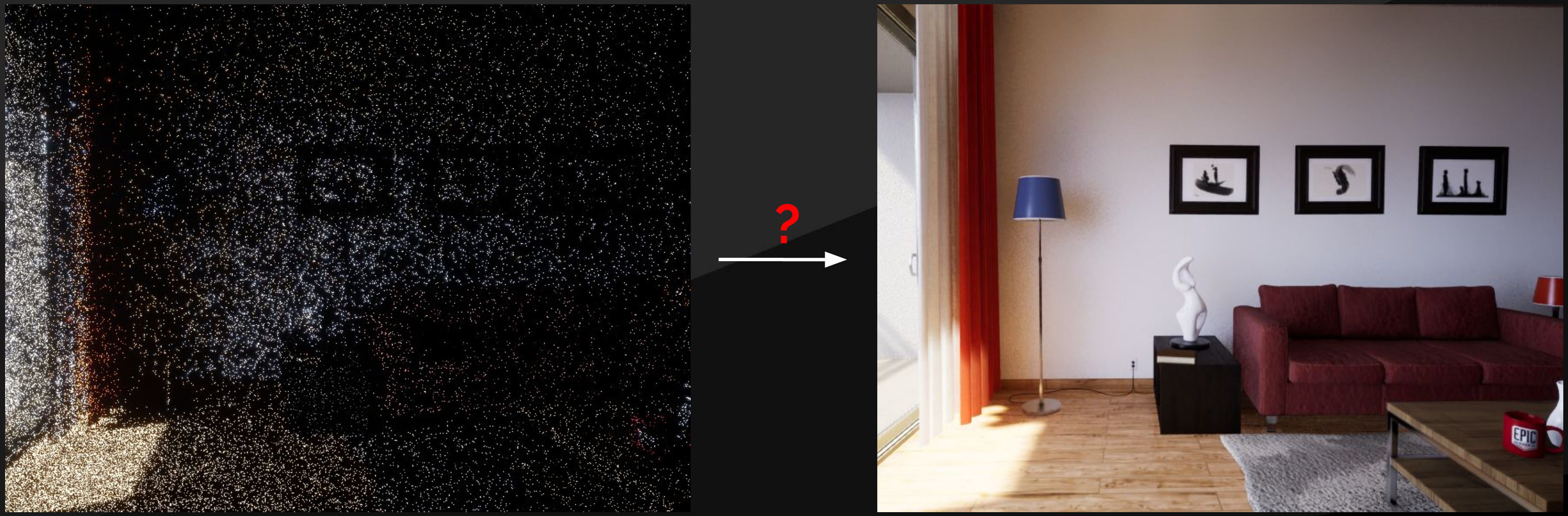
In ‘The Matrix Awakens’ tech demo, the cost of both tracing methods is nearly the same.  Hardware Ray Tracing gives higher quality reflections and supports GI over huge view ranges with Far Field, so it’s the better choice.

# Final Gather

So now moving on to Lumen’s Final Gather,

# How to solve noise in the light transfer?

- We can't even afford one ray per pixel
  - But need hundreds of effective samples for high quality indoor GI



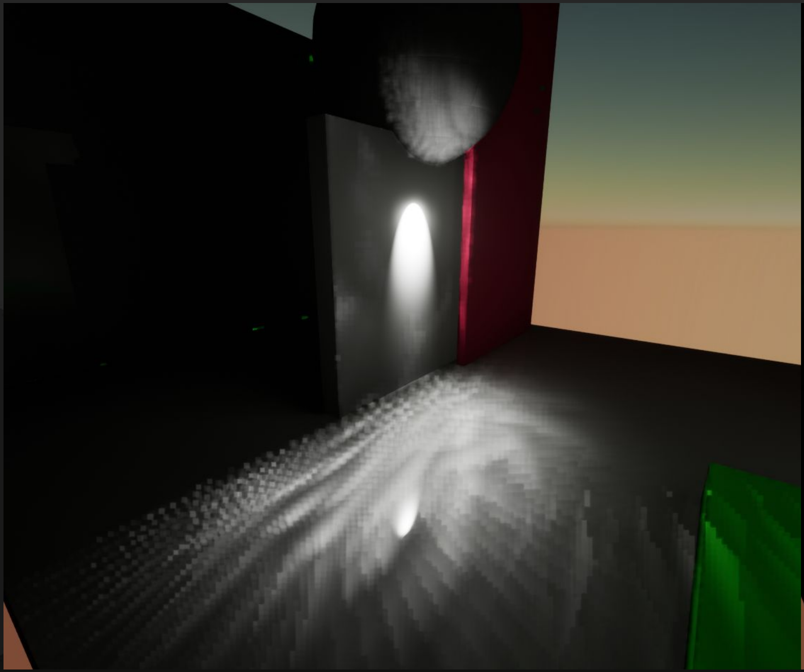
1 path per pixel

The third fundamental problem we had to solve for realtime indirect lighting was the noise in the light transfer.

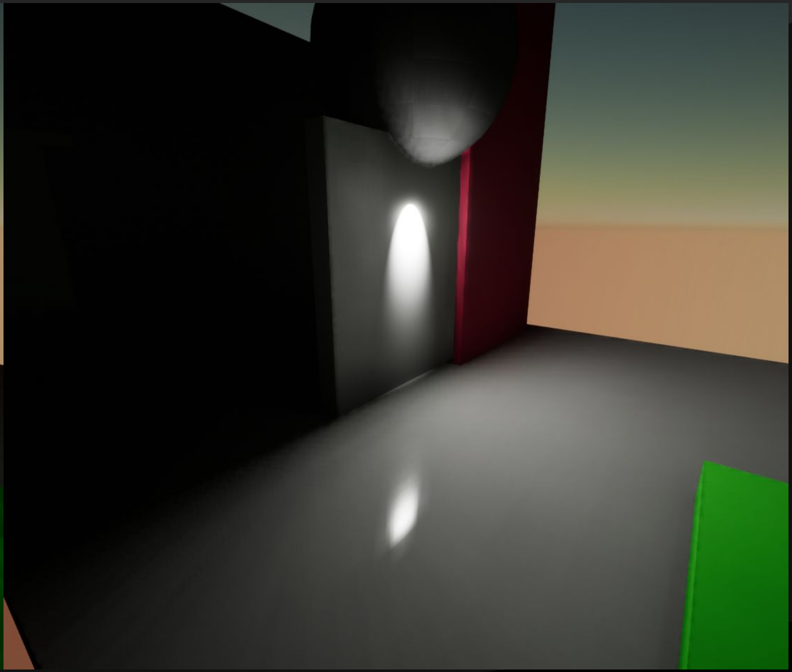
We can't even afford a single ray per pixel, and on the left here you can see what a path per pixel looks like, but we need hundreds of effective samples for high quality indoor GI.

# Early experiment: Prefiltered cone tracing

- One cone = many rays
  - Cone tracing makes the Final Gather trivial



Rays

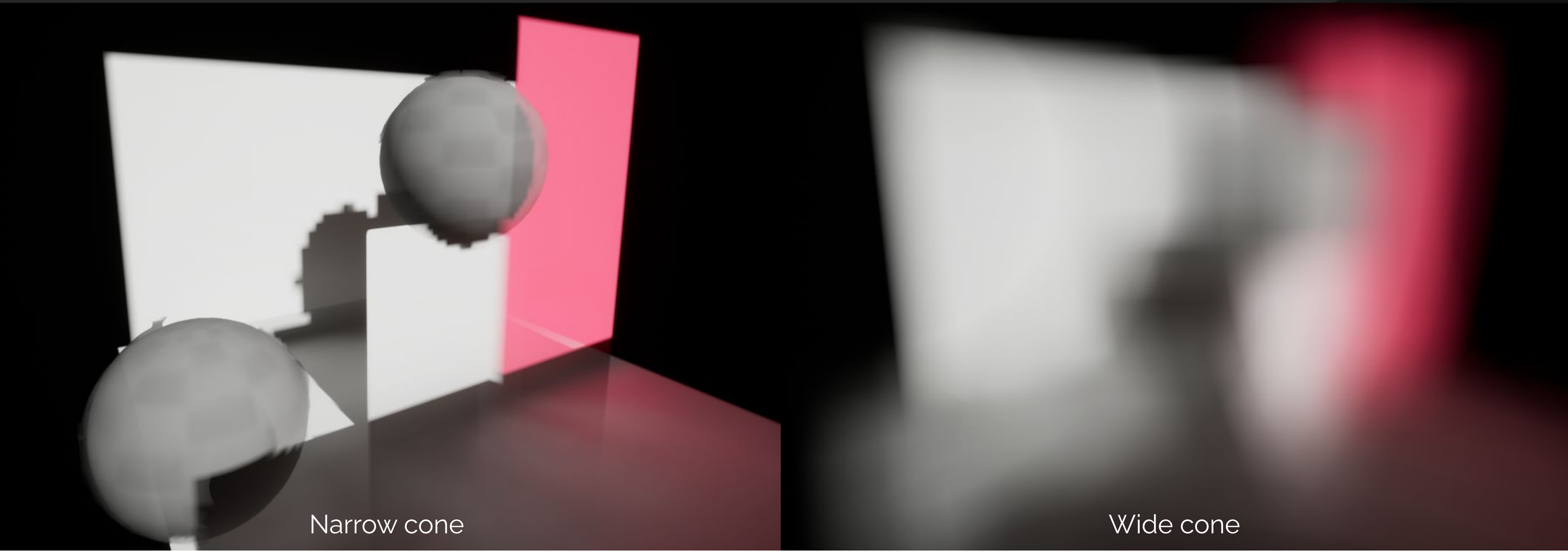


Cones

One of our early experiments was prefiltered cone tracing.  It’s very difficult to implement, but if you can pull it off, tracing a single cone gives you the results of many rays.  Cones can be very effective at solving noise and they essentially make the Final Gather trivial.

# Mesh SDF Cone Tracing

- Surface Cache mip selection based on cone intersection size
- Cone occlusion approximation using distance to surface
- Separate surfaces blended with Weighted Blended OIT [McGuire 2013]



We implemented cone tracing against Mesh Signed Distance Fields.  Whenever the cone intersects a surface, we calculate the mip of the surface cache to sample using the size of the cone intersection, giving prefiltered lighting for cones that intersect.

When the cone has a near miss with the surface, it's only partially occluded so this becomes a transparency problem.  We can approximate how much the cone was occluded using the distance from the cone axis to the surface, which we can get from the distance field.

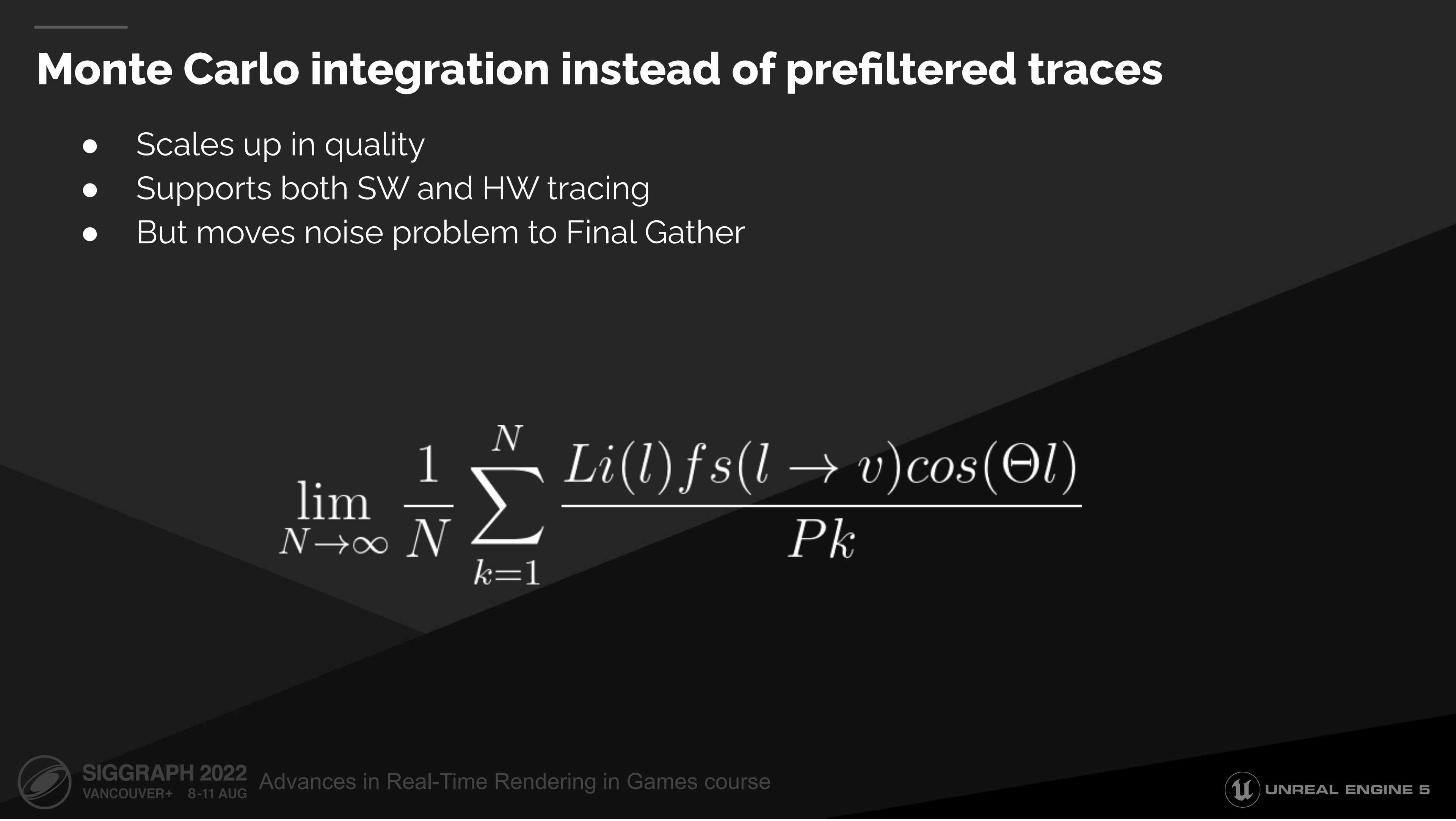
We then need to combine multiple partial hits from different meshes that are out of order.  We solved this with Weighted Blended OIT, which has significant leaking for primary rays over a large distance, but had much less leaking over smaller distances for diffuse indirect rays.

You can see on the right that the cone trace works because there are no hard edges, which would cause noise.

# Cone traces very effective at solving noise, but

- Leaking or over-occlusion unavoidable
    - Can never resolve lighting through small distant window
- Can't support HWRT

Cone traces were very effective at solving noise, but at the end of the day we just couldn't solve leaking in all cases.  We'd always have to choose between leaking or over-occlusion, and we could never resolve the lighting from a small distant window.
It also only worked with Software Ray Tracing.



# Monte Carlo integration instead of prefiltered traces

- Scales up in quality
- Supports both SW and HW tracing
- But moves noise problem to Final Gather

$$\lim_{N\to\infty} \frac{1}{N}\sum_{k=1}^{N} \frac{Li(l)fs(l \to v)cos(\Theta l)}{Pk}$$

So instead we’re going to do Monte Carlo integration.  That scales up in quality to the highest end, and supports any kind of ray tracing, but it moves the whole noise problem to the Final Gather.

# Previous real-time approach:
# **Irradiance Field** [Tatarchuk 2012]

Trace from probes in volume, pre-integrate irradiance, interpolate to pixels

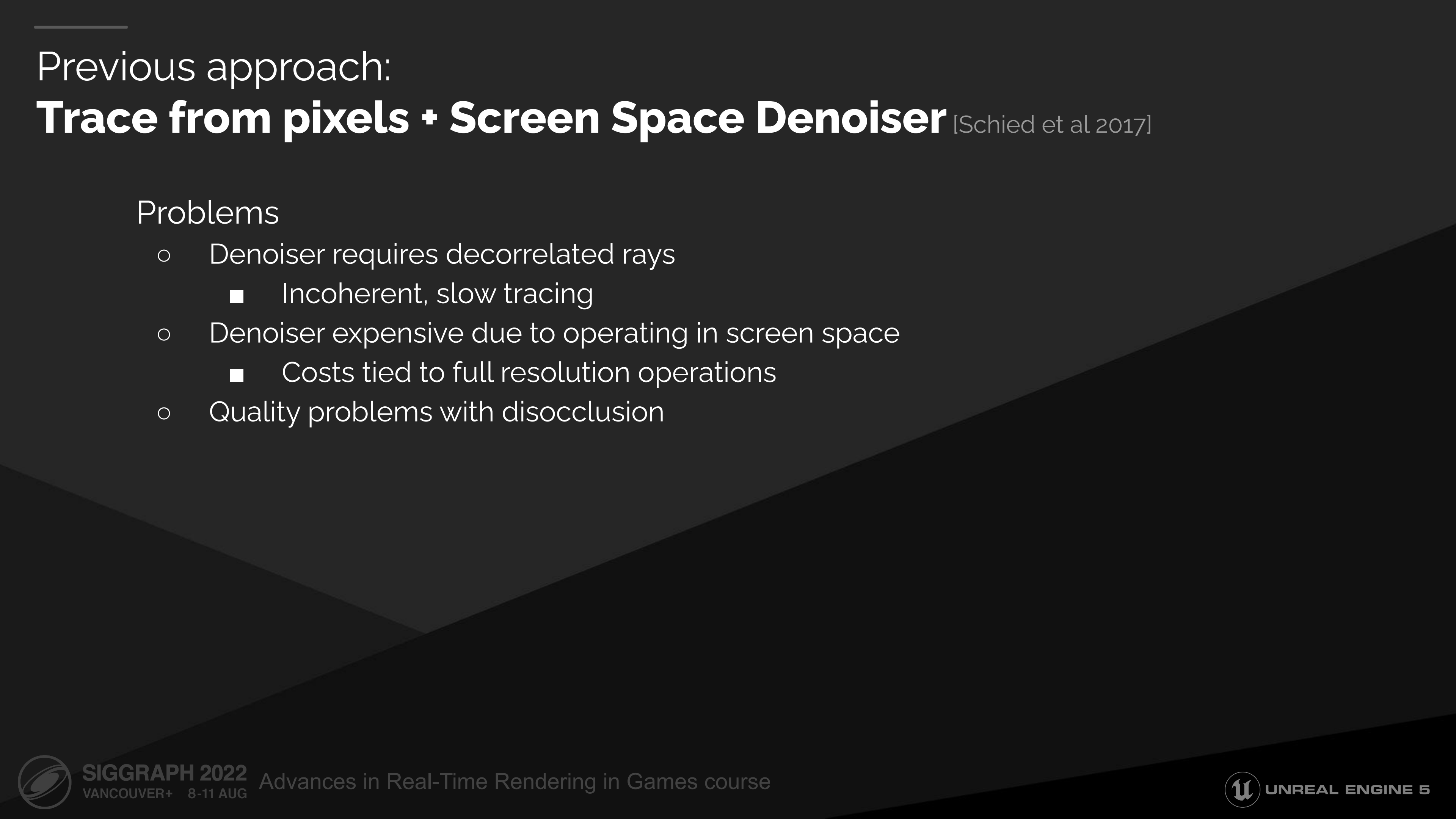
Problems

- Irradiance was computed at the probe, not the pixel
  - Leaking and over-occlusion
  - Probe placement challenges
  - Slow lighting update
- Low spatial resolution
  - Distinctive flat look

The most popular approach for solving diffuse light transfer is the Irradiance Field.  Irradiance Fields trace from probes placed in a volume, then pre-integrate irradiance at the probe position, and then interpolate that to the pixels on your screen.

The biggest problem with this technique is that irradiance was calculated at the probe, not the pixel, which causes leaking and over-occlusion and makes the probe placement very difficult.

It's also a volumetric representation so you can only afford a low spatial resolution which causes the GI to look flat.



# Previous approach:
## **Trace from pixels + Screen Space Denoiser** [Schied et al 2017]

Problems

- ○ Denoiser requires decorrelated rays
  - ■ Incoherent, slow tracing
- ○ Denoiser expensive due to operating in screen space
  - ■ Costs tied to full resolution operations
- ○ Quality problems with disocclusion

On the other end of the spectrum, you can trace from the actual pixels on the screen and then try to solve the noise after the fact with a screen space denoiser.

That has its own set of problems.  The denoiser requires a decorrelated set of rays to get full coverage of the hemisphere, which means they are incoherent and slow to trace.

The denoiser operations become very expensive because they are operating in screen space, there's no opportunity for downsampled filtering.  Denoisers have problems with disocclusion, where there just aren't enough samples to converge in newly revealed areas.

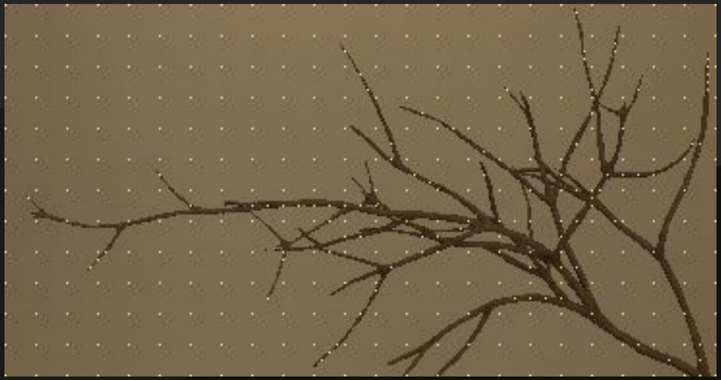
# Need to find something in between

- Want the accuracy of tracing from pixels
  - With significantly less cost than existing approaches that do

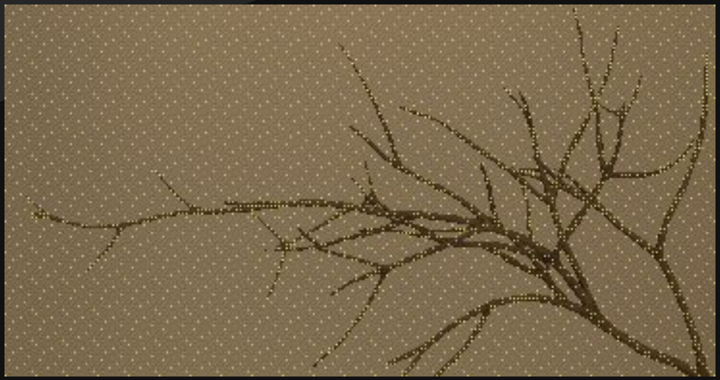
We would like to find something in between.  We want the accuracy of tracing from the pixels on the screen, like indirect shadows, but significantly less cost than the approaches that do.

## Our approach:
# **Screen Space Radiance Caching**

- Trace from probes placed on pixels (screen probes)
  - Adaptive downsampling
- Interpolate to pixels within the same plane
- Jitter placement grid and temporally accumulate



Frame 0



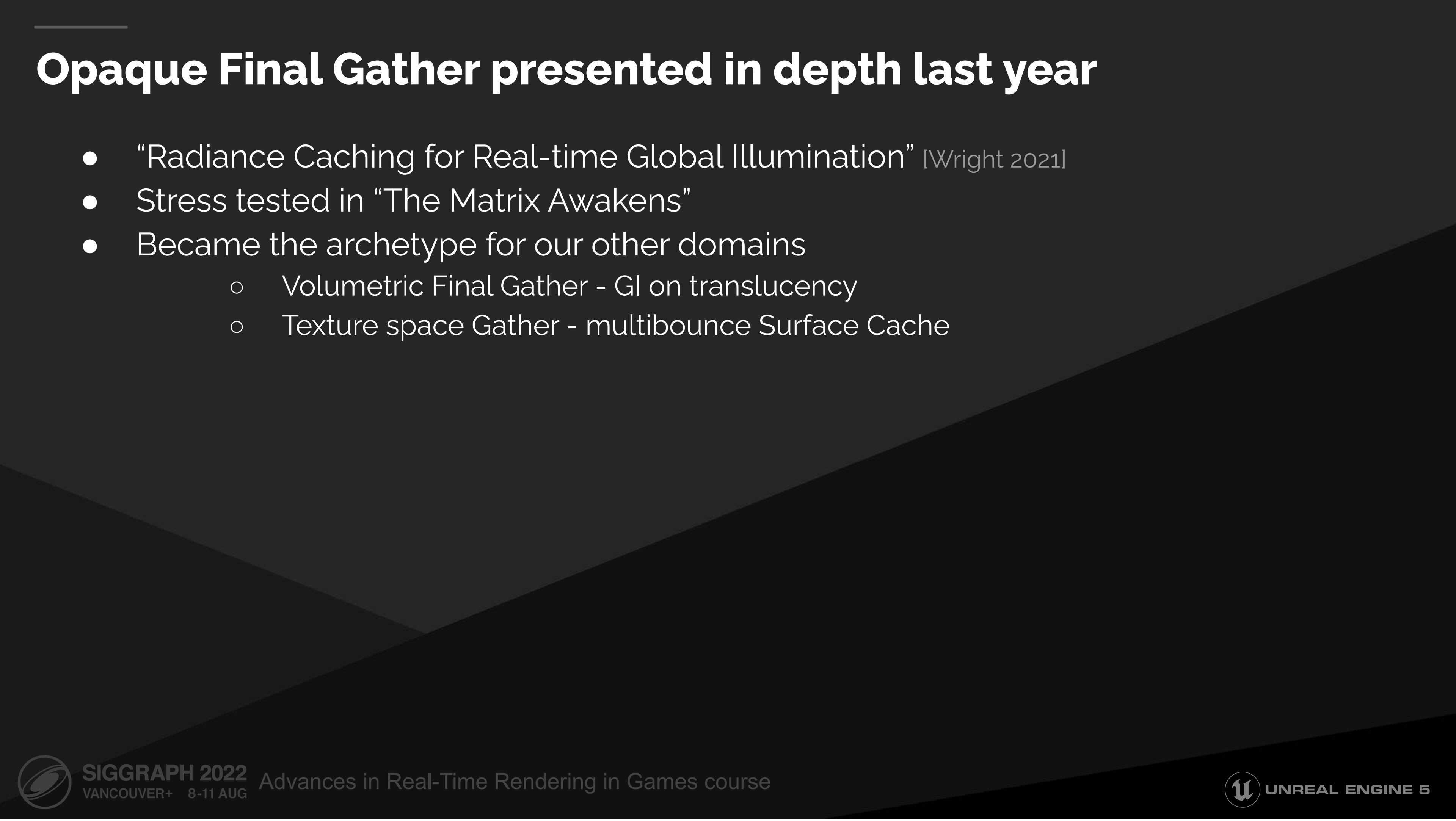
Temporal accumulation

Our approach is Screen Space Radiance Caching.  We trace from probes which are placed on pixels on the screen, which we call screen probes.

This is effectively adaptive downsampling.  You can see on the left that the probes are placed uniformly, except where there's more detailed geometry, where we place more probes.

After we place the probes and trace from them, we interpolate their radiance to any other pixels within the same plane, which limits leaking from the interpolation to pixels in the same plane, which is hard to notice.

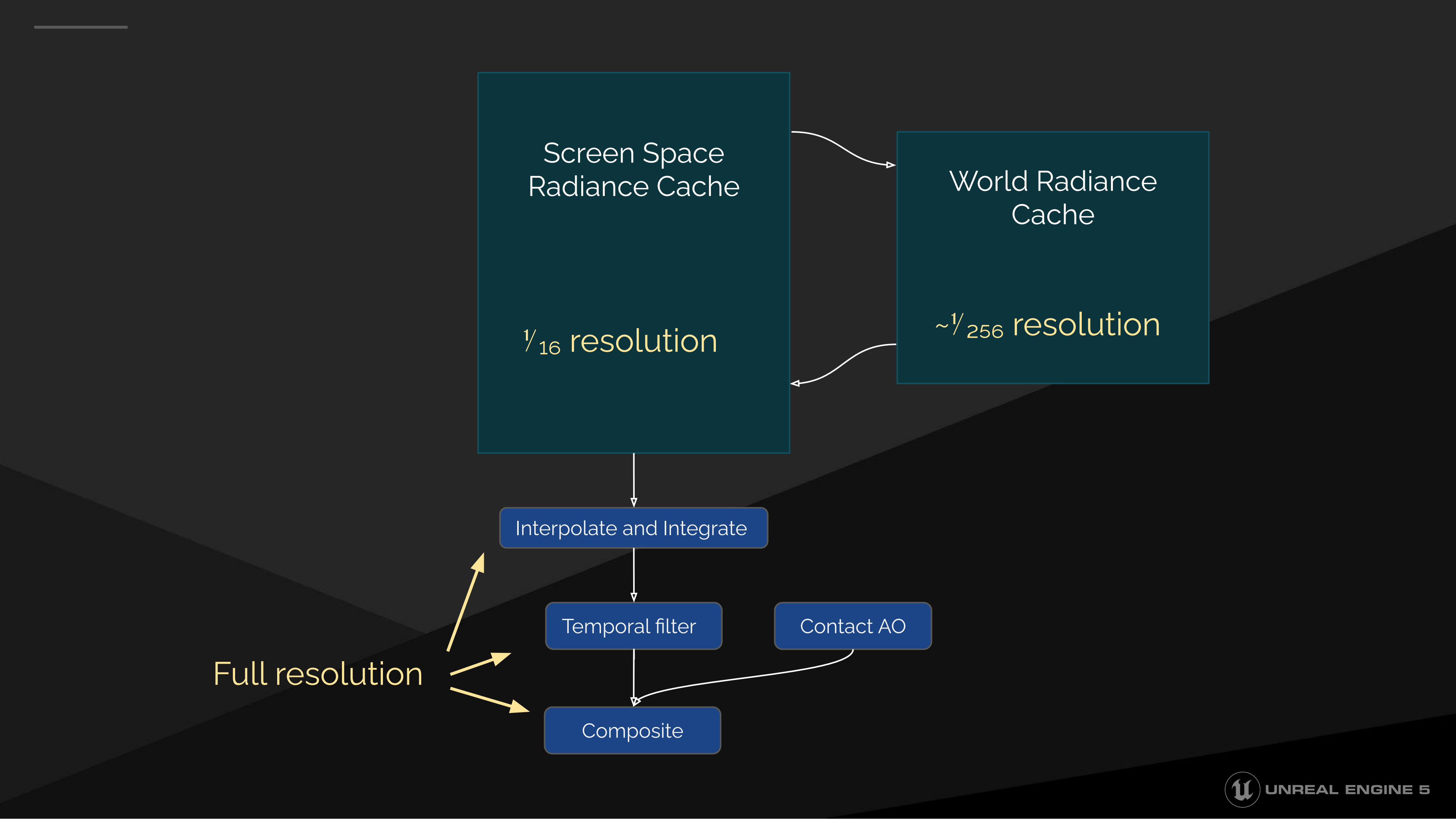
We jitter the placement grid of the probes over frames and temporally accumulate to get good coverage.



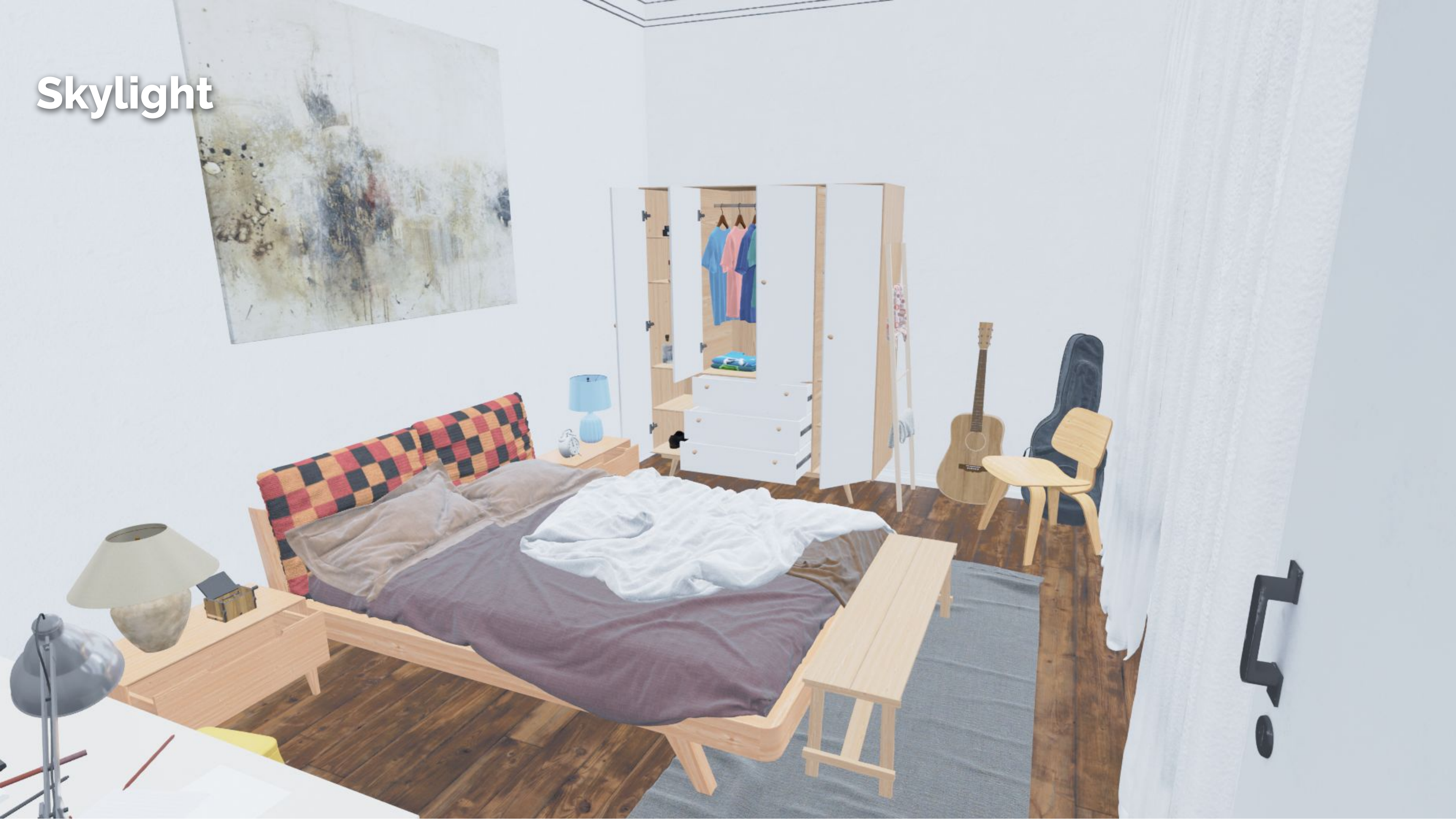
# Opaque Final Gather presented in depth last year

- "Radiance Caching for Real-time Global Illumination" [Wright 2021]
- Stress tested in "The Matrix Awakens"
- Became the archetype for our other domains
  - Volumetric Final Gather - GI on translucency
  - Texture space Gather - multibounce Surface Cache

I presented Lumen's Opaque Final Gather in last year's Advances course, but since then we've stress tested it in 'The Matrix Awakens' tech demo. It became the archetype for our other domains, the Volumetric Final Gather, and the texture space gather, so I want to share some of the insights that we've gained since then.



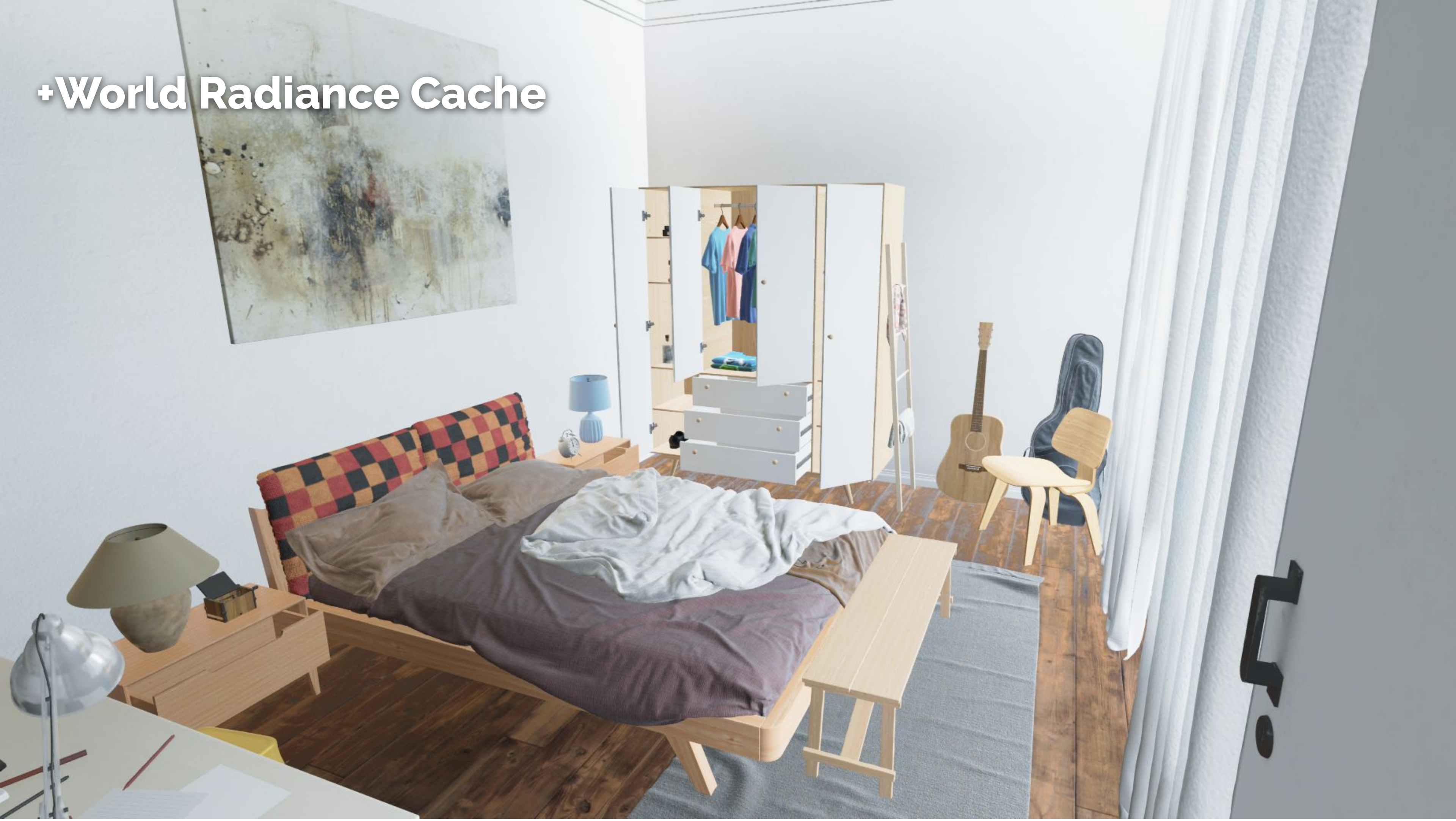
The opaque Final Gather has three main parts, first there's the Screen Space Radiance Cache which is operating at 1/16th resolution in each dimension.  It's backed up by the World Space Radiance Cache, which is handling distant lighting at a much lower resolution.  At full resolution there's the interpolation, integration, the temporal filter and Contact AO.



# Skylight

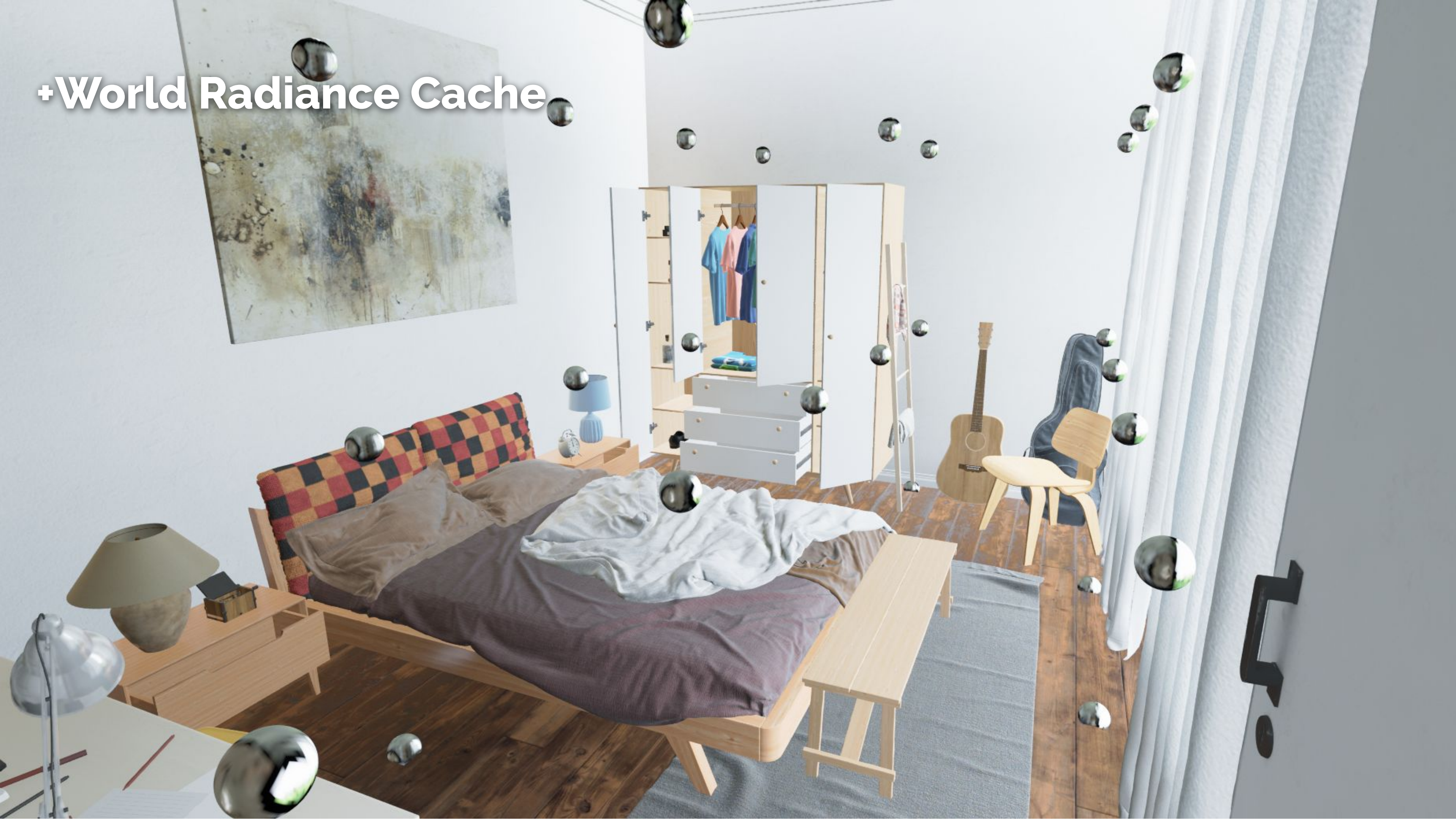
Here’s what a scene looks like, using the Final Gather, starting from the most distant lighting to shortest range.  Starting out here’s just the skylight,

# +World Radiance Cache



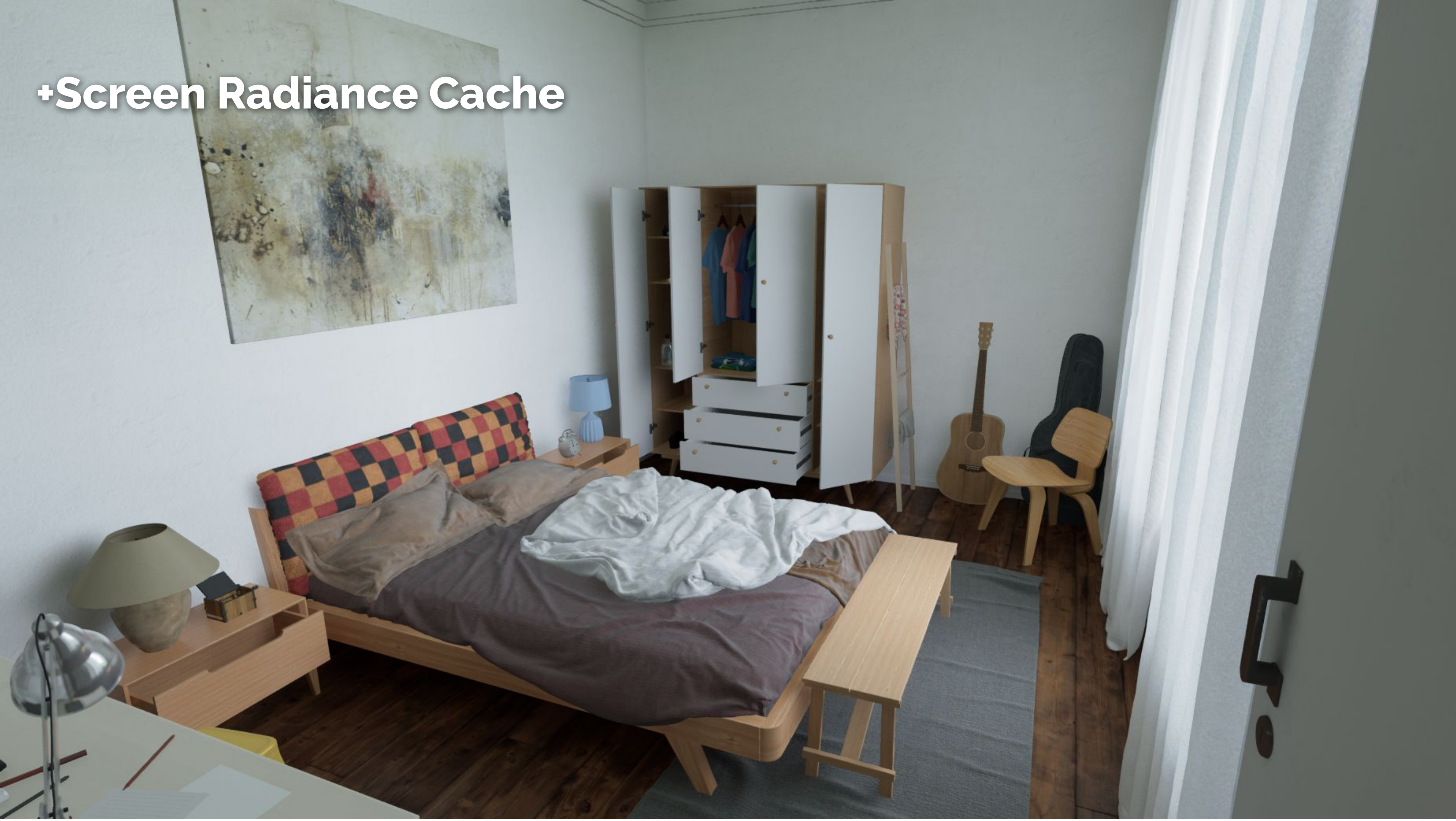
And then adding the World Space Radiance Cache, which is solving lighting further than two meters, which is only the wall on the left in this scene.

# +World Radiance Cache

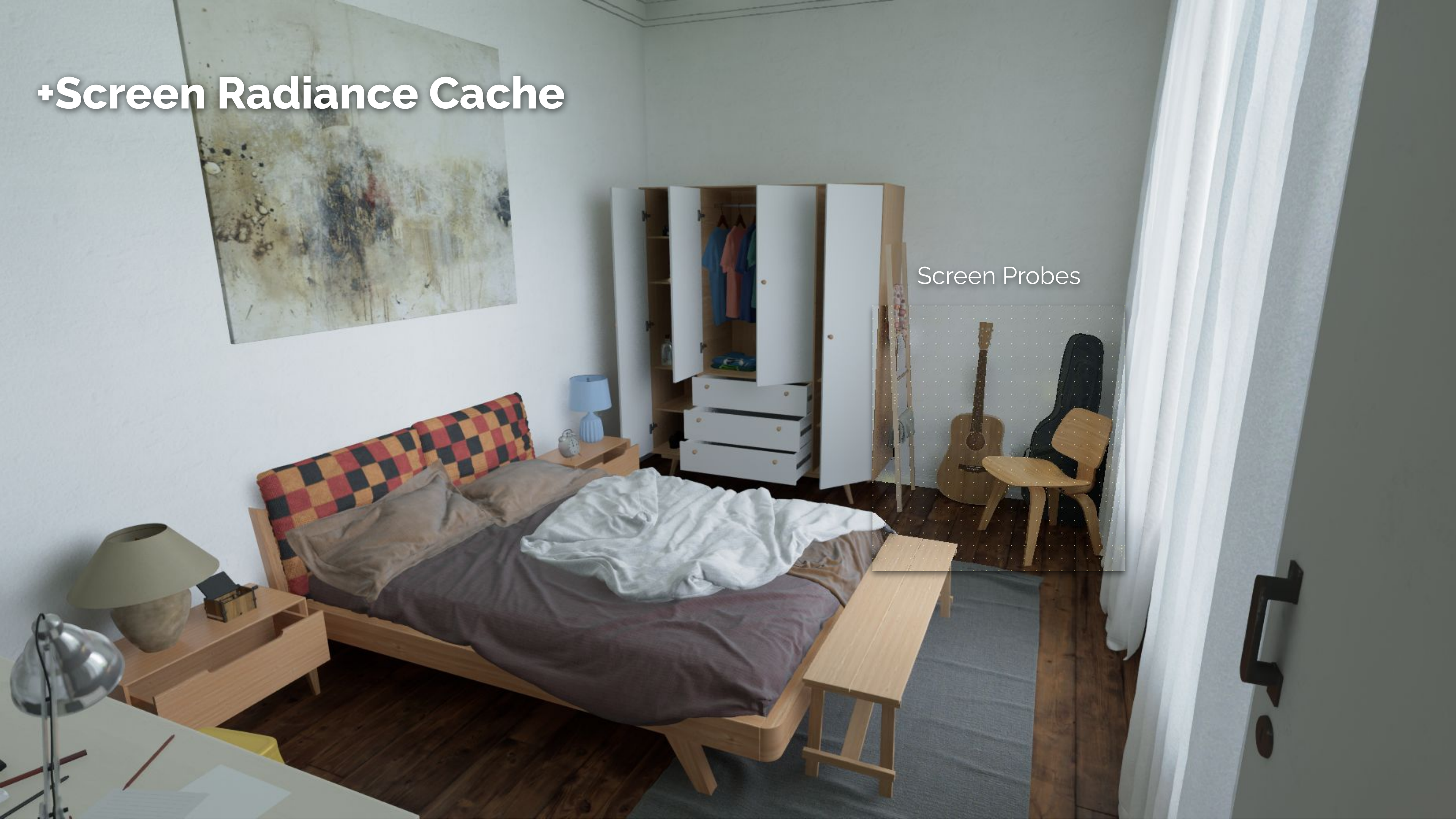


When we look at the probes in the World Space Radiance Cache you can see that the probes on the wall have resolved the window very accurately, but the probes on the right are seeing through the wall.

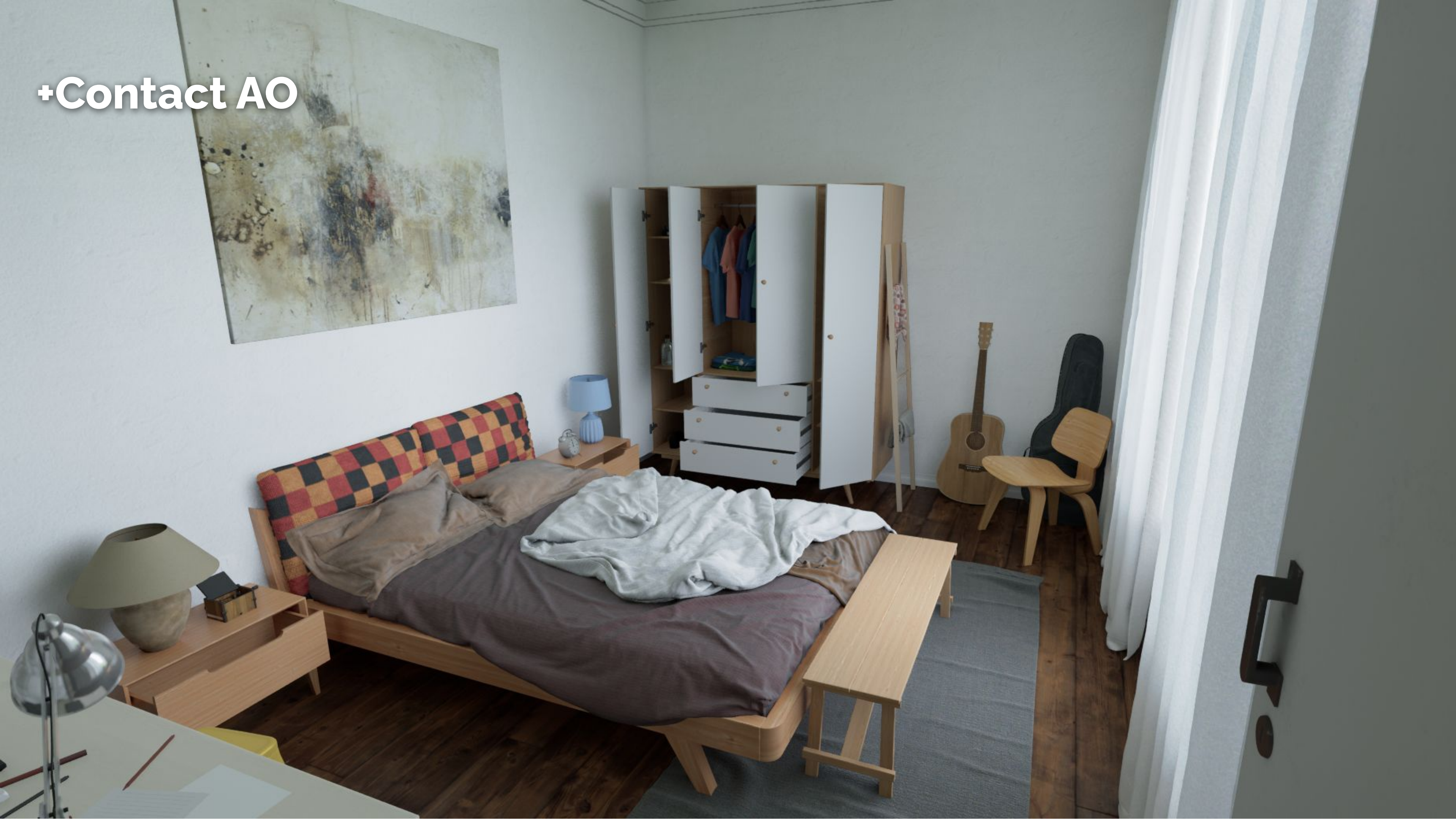
# +Screen Radiance Cache



Then adding the Screen Space Radiance Cache, which is capturing all the nearby lighting, so we have indirect shadows, but in a downsampled space.

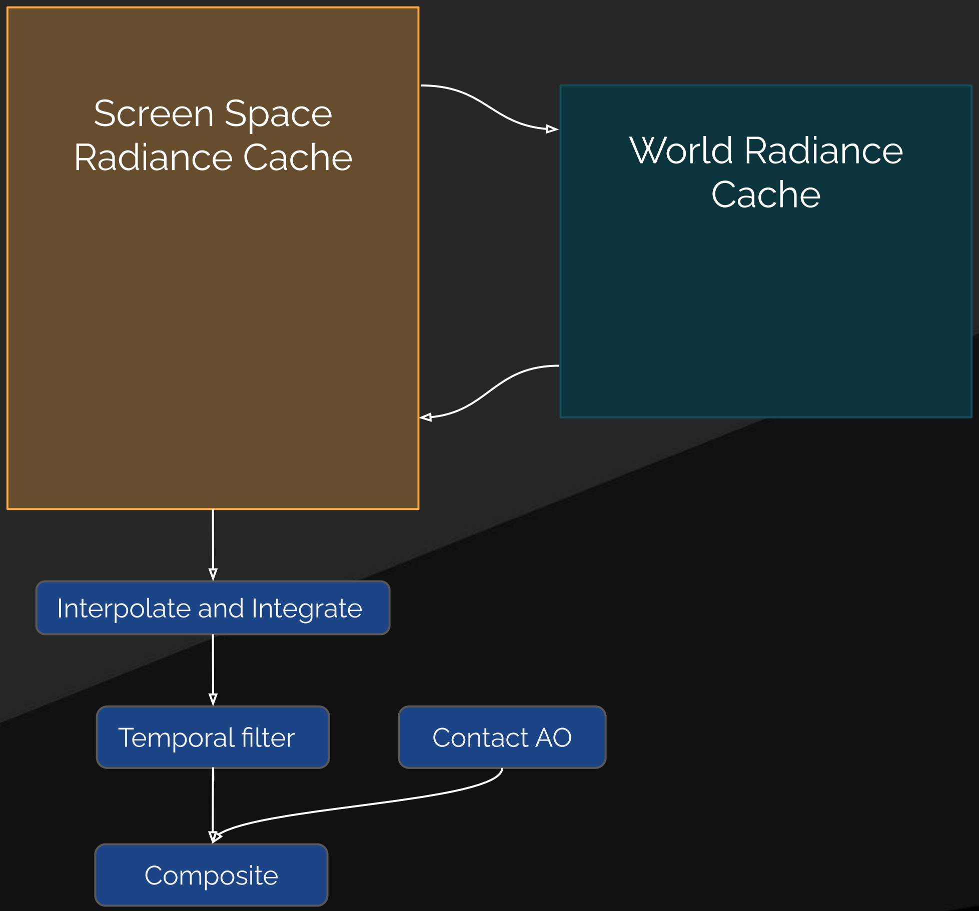


Here’s where the Screen Probes are placed.  They’re uniformly placed except where there’s detailed geometry, where we subdivide the grid and place more.



# +Contact AO

Finally here’s what it looks like with Contact AO, which is making up for some shadowing detail that we lost by downsampling the Screen Radiance Cache.



Screen Space Radiance Cache

World Radiance Cache

Interpolate and Integrate

Temporal filter

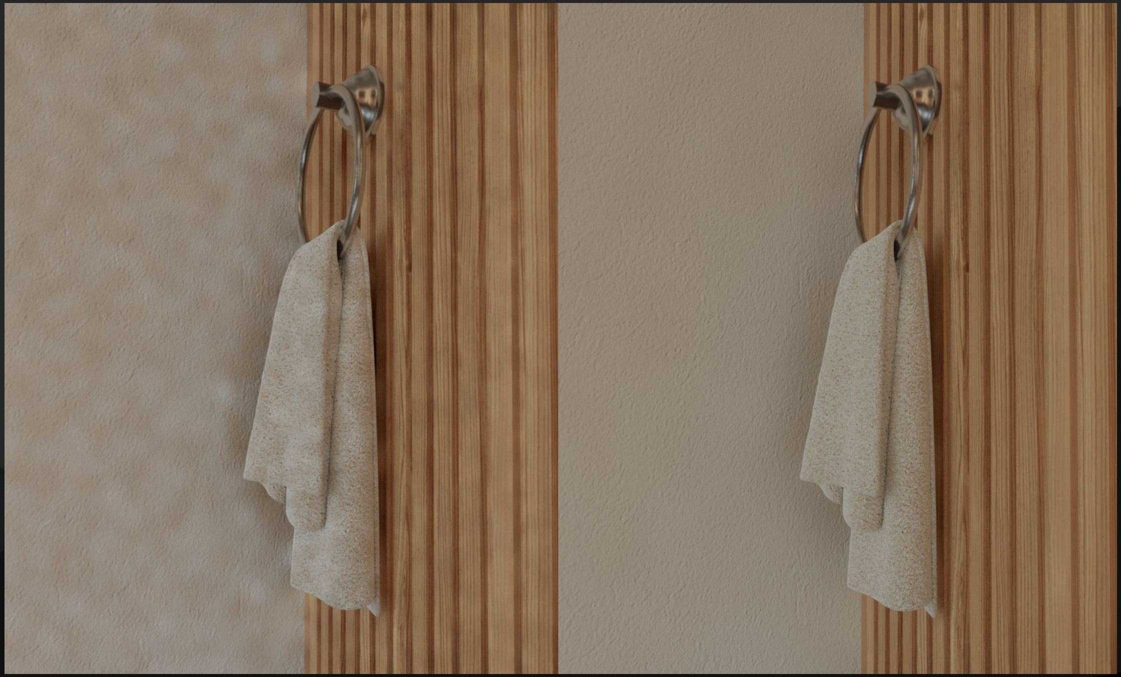
Contact AO

Composite

Looking at the Screen Space Radiance Cache in more detail,

# Filtering in Radiance Cache space

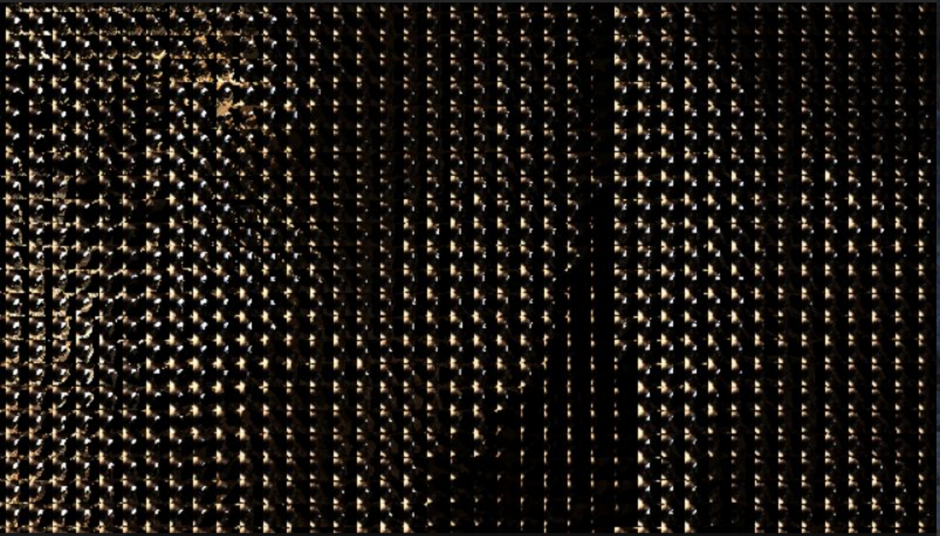
- Large spatial filter for cheap
  - 3^2 in probe space vs 48^2 screen space



It provides large spatial filters for cheap, because we're filtering in the Radiance Cache space and not in screen space.  A very small 3x3 filtering kernel in the probe space gives us the same noise reduction that a large 48x48 screen space filtering kernel would do.  We only have to load the positions of the probes, rather than all the positions and normals of all the pixels for error weighting.

# Importance sample incoming lighting

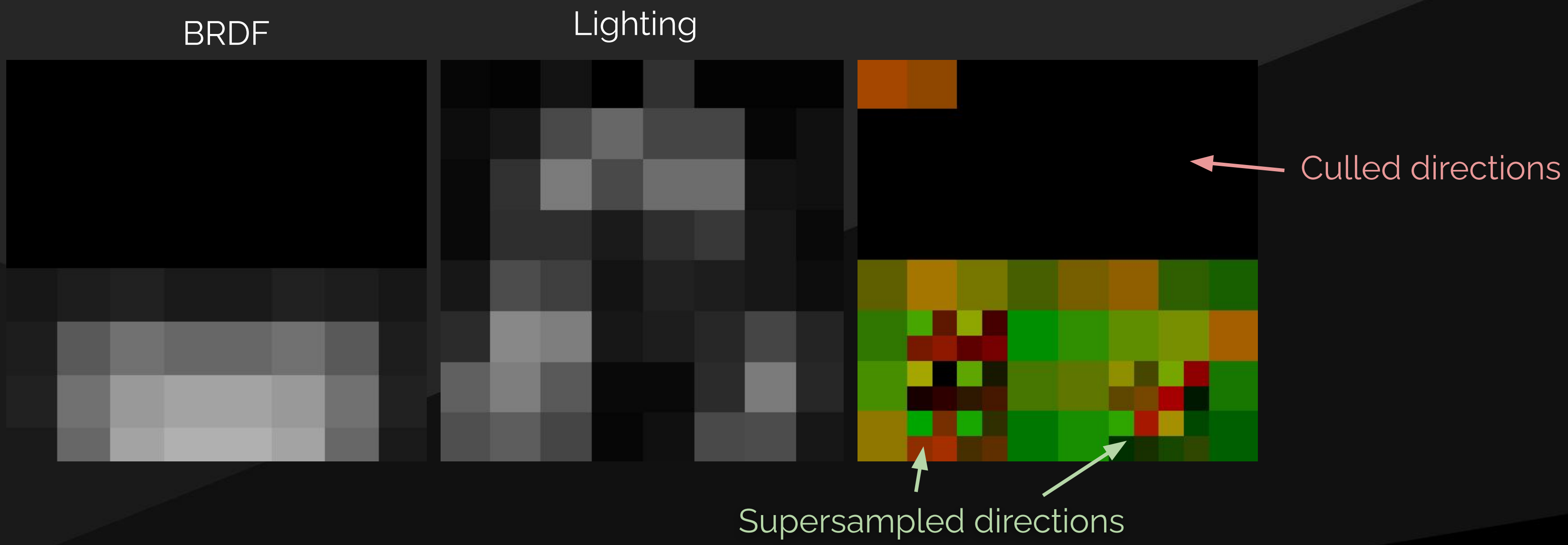
- Importance sampling is only as good as the importance estimate
- We have an accurate estimate of incoming lighting from:
  - Last frame's Screen Radiance Cache, reprojected
  - World Radiance Cache



We importance sample the incoming lighting.  Importance sampling is only as good as the importance estimate, and we have a very accurate estimate of the incoming lighting from last frame's Screen Space Radiance Cache, reprojected into the current frame.  We can very efficiently find all of last frames' rays because they're indexed by direction as well as position in the radiance cache.  Where the reprojection fails, like the edges of the screen, we fall back to the World Space Radiance Cache and still have effective importance sampling.

# Product Importance Sampling

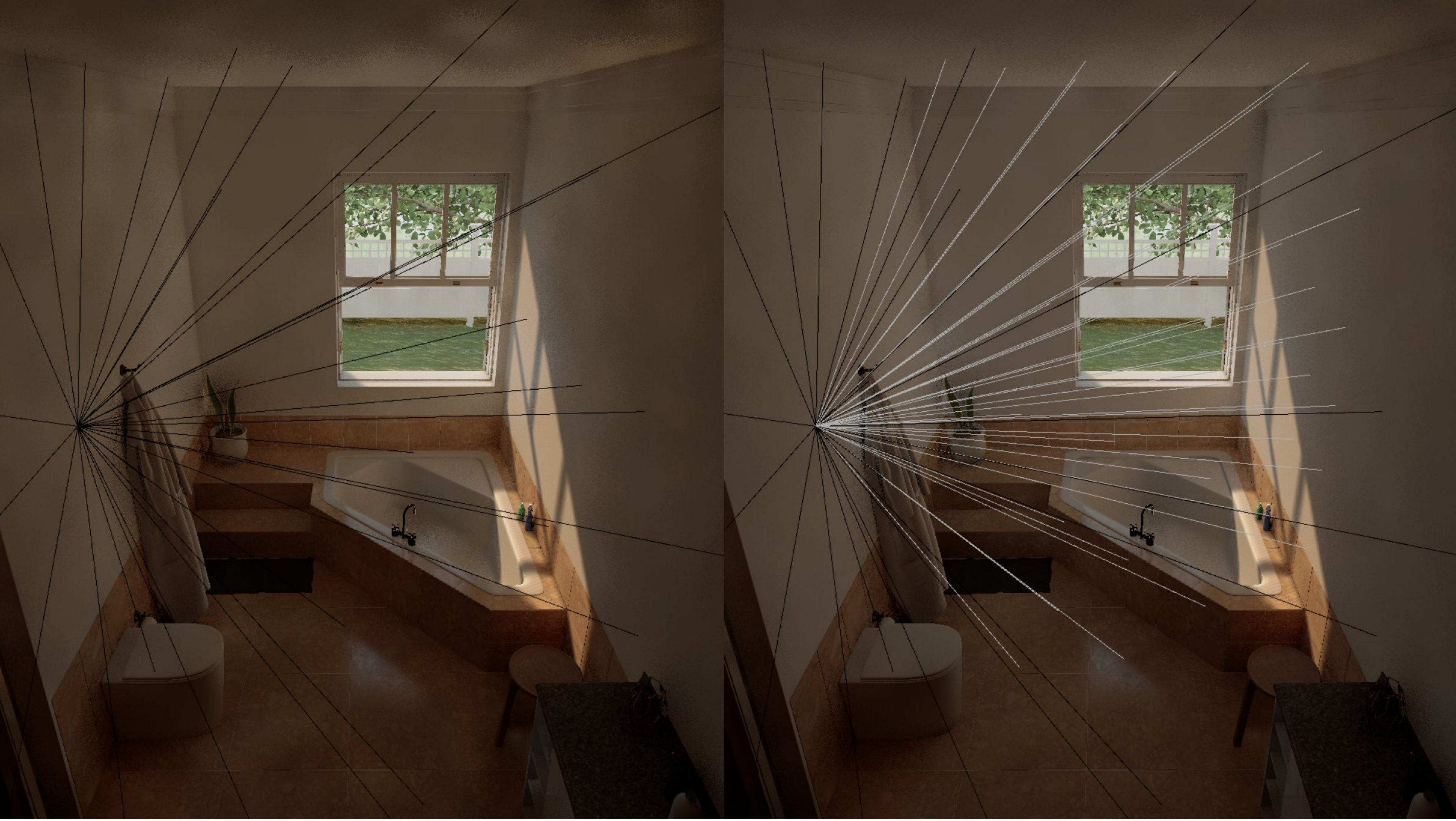
- Better than importance sampling only BRDF or Lighting
  - Or Multiple Importance Sampling, which discards rays
- Cull unimportant ray directions, supersample the most important directions



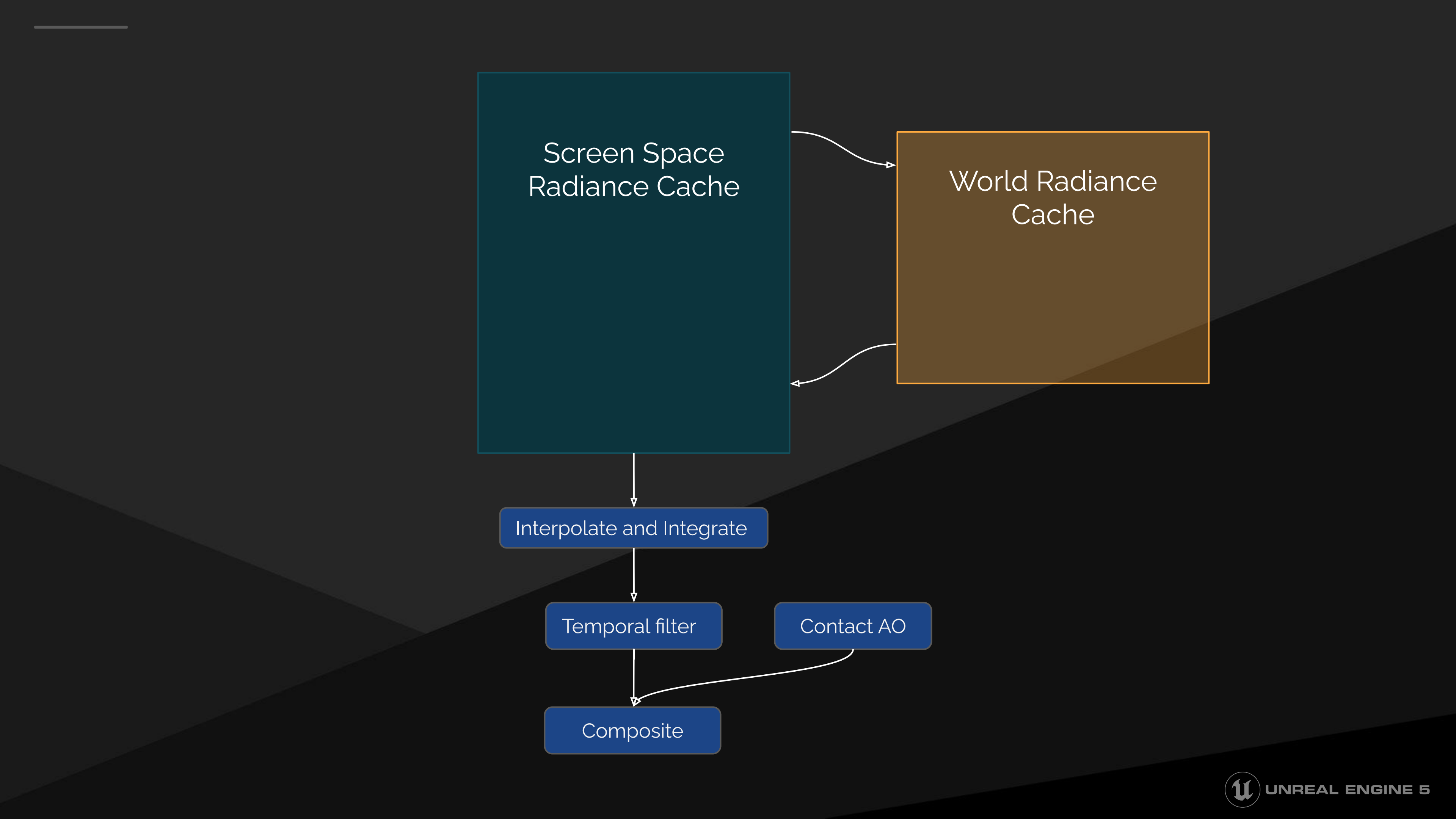
We're operating in a downsampled space, so we can afford to launch a whole threadgroup per probe to do better sampling. Product Importance Sampling is normally only possible in offline rendering, but we can do it in real-time.

Product Importance Sampling is better than importance sampling only the BRDF or the Lighting, and it's better than Multiple Importance Sampling, which throws away tracing work when the direction had a low weight in the other distribution.

On the left we have the BRDF for a probe on a wall, where only the directions in a single hemisphere matter. In the middle we know the incoming lighting from the previous frame, which tells us that most of the lighting is coming from these two directions. We then reassign the useless rays into the directions that are the most important in the product.



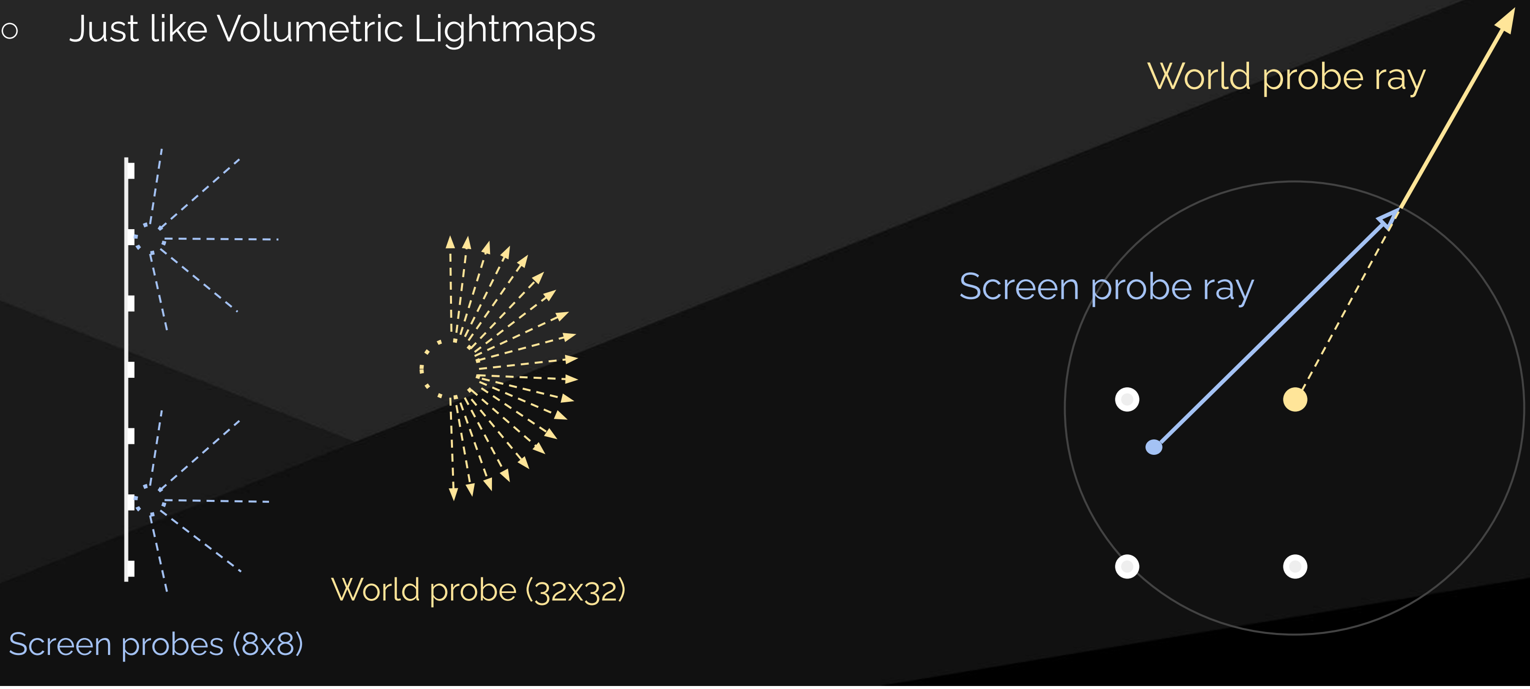
Here the white rays are the ones we've generated to supersample the most important directions.  We're getting the quality of tracing four times more rays without actually tracing any more rays.



Now looking at the World Space Radiance Cache in more detail,

# World space Radiance Cache

- Separate sampling for distant Radiance
  - Shorten Screen probe trace, interpolate from World probe on miss
  - Higher directional resolution (16 : 1), lower spatial resolution
- Stable error since world space - easy to hide
  - Just like Volumetric Lightmaps


World probe ray

Screen probe ray

World probe (32x32)

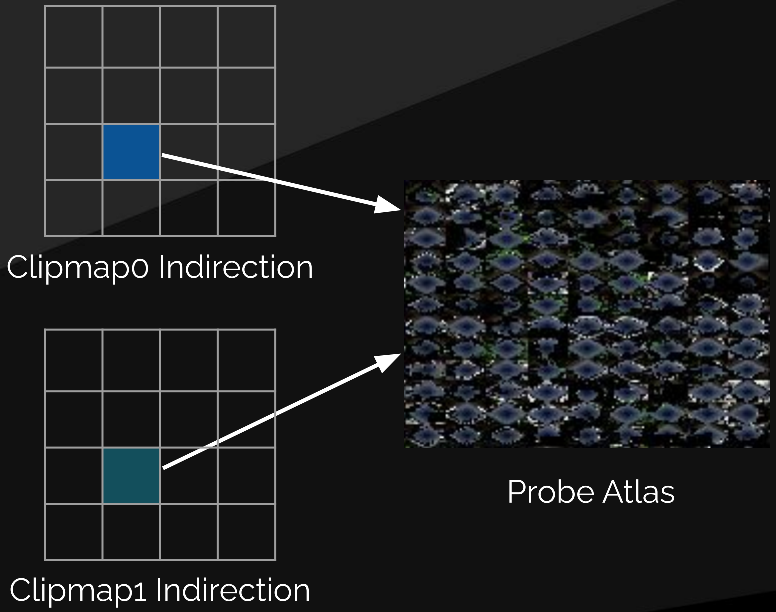
Screen probes (8x8)

The World Space Radiance Cache is handling distant lighting, with higher directional resolution but lower spatial resolution.  It solves the problem where all of the lighting in a room is coming from a small distant window, which was missed with a small number of rays.  We integrate the two radiance caches by shortening the Screen probe ray, and when it misses we interpolate from the World Space Radiance Cache.

The World Space Radiance Cache has stable error because the probe positions are stable, so it’s easy to hide.

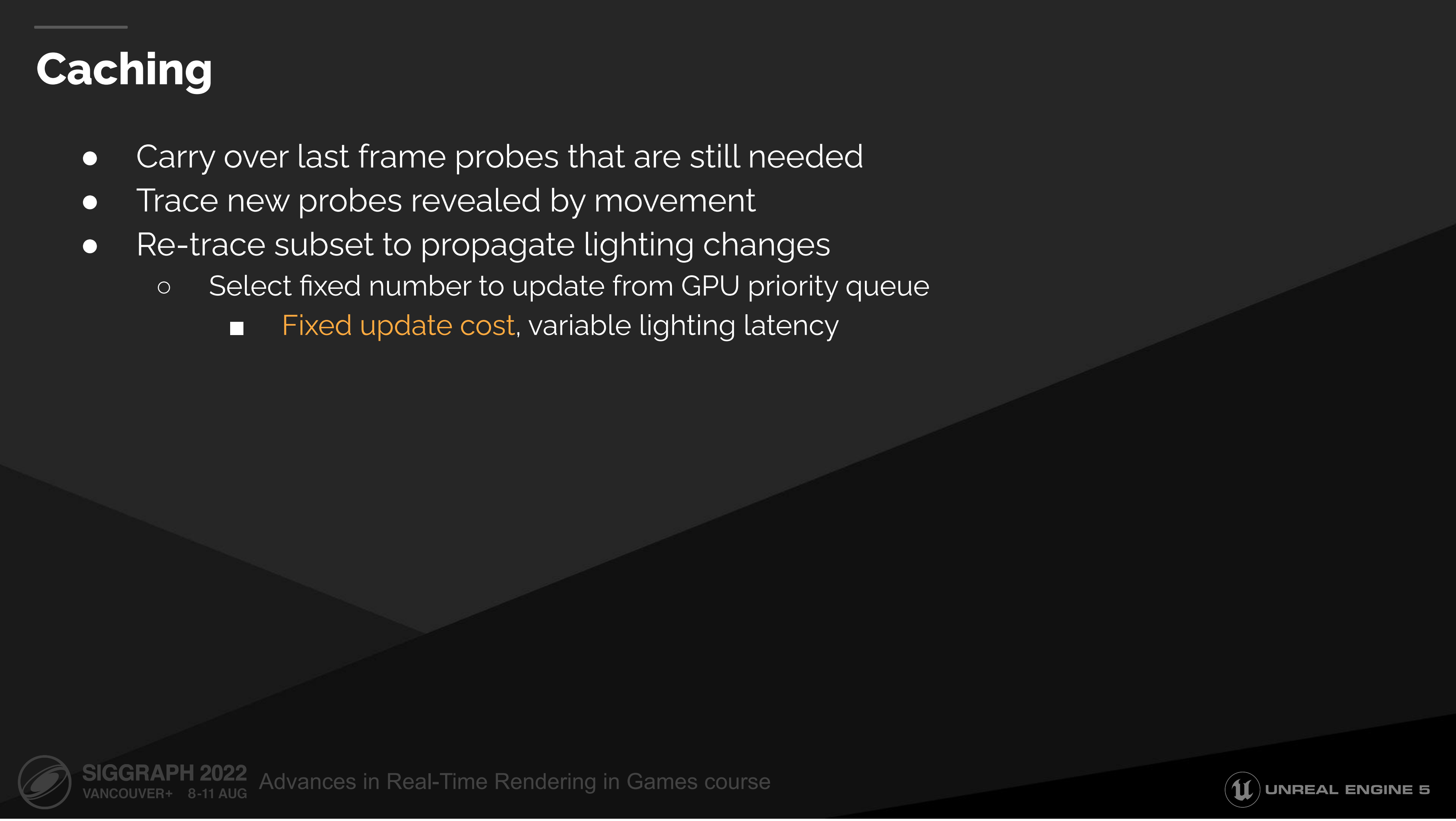
# Sparse coverage

- Clipmap distribution maintains bounded screen size
- Persistent allocations with free list



The World Space Radiance Cache has sparse coverage, and we use a clipmap distribution to maintain a bounded screen distance between the probes, to make sure that we don’t over-sample, or under-sample.

The probes use persistent allocations, so they can survive across frames.



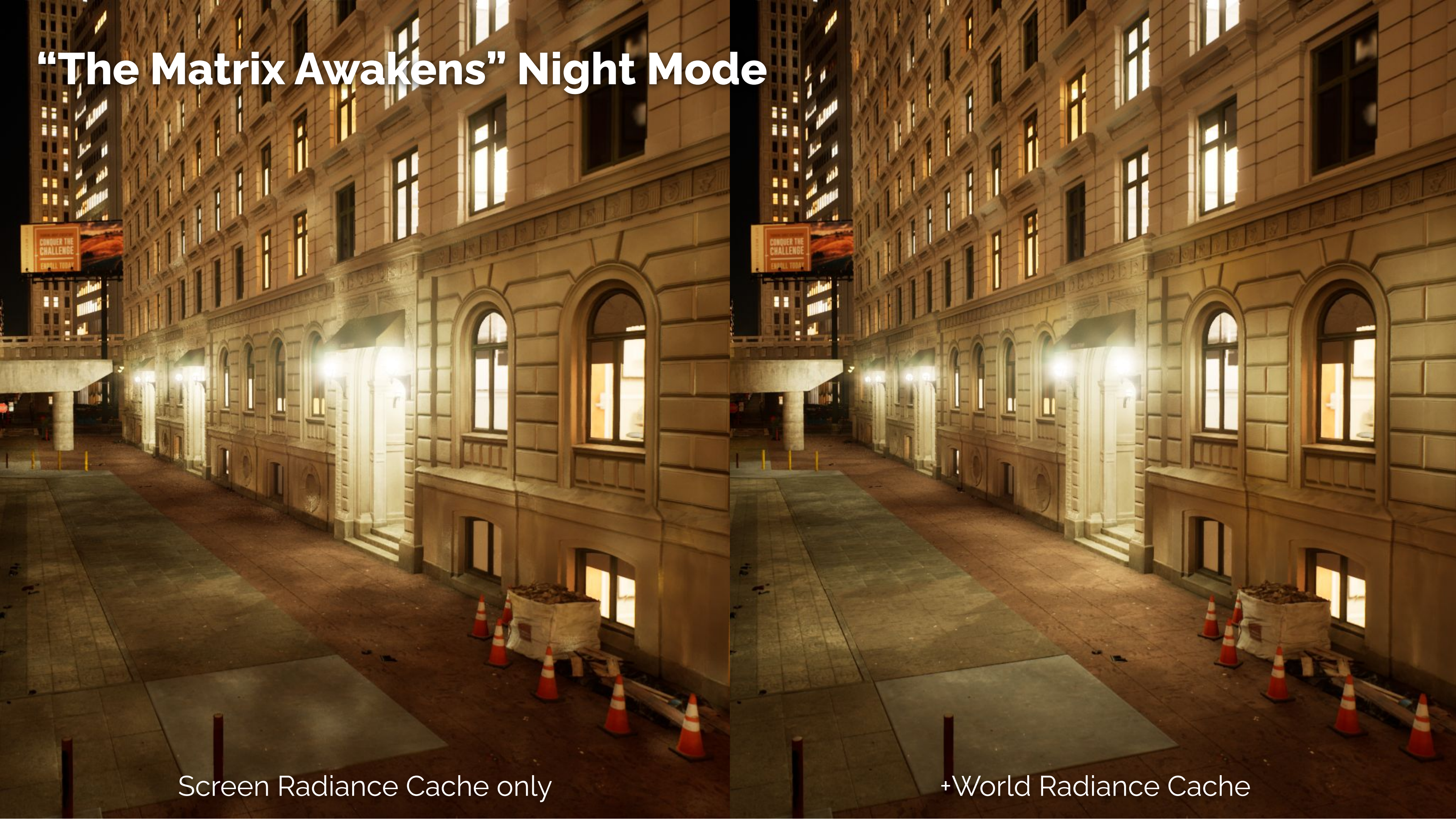
## Caching

- Carry over last frame probes that are still needed
- Trace new probes revealed by movement
- Re-trace subset to propagate lighting changes
    - Select fixed number to update from GPU priority queue
        - Fixed update cost, variable lighting latency

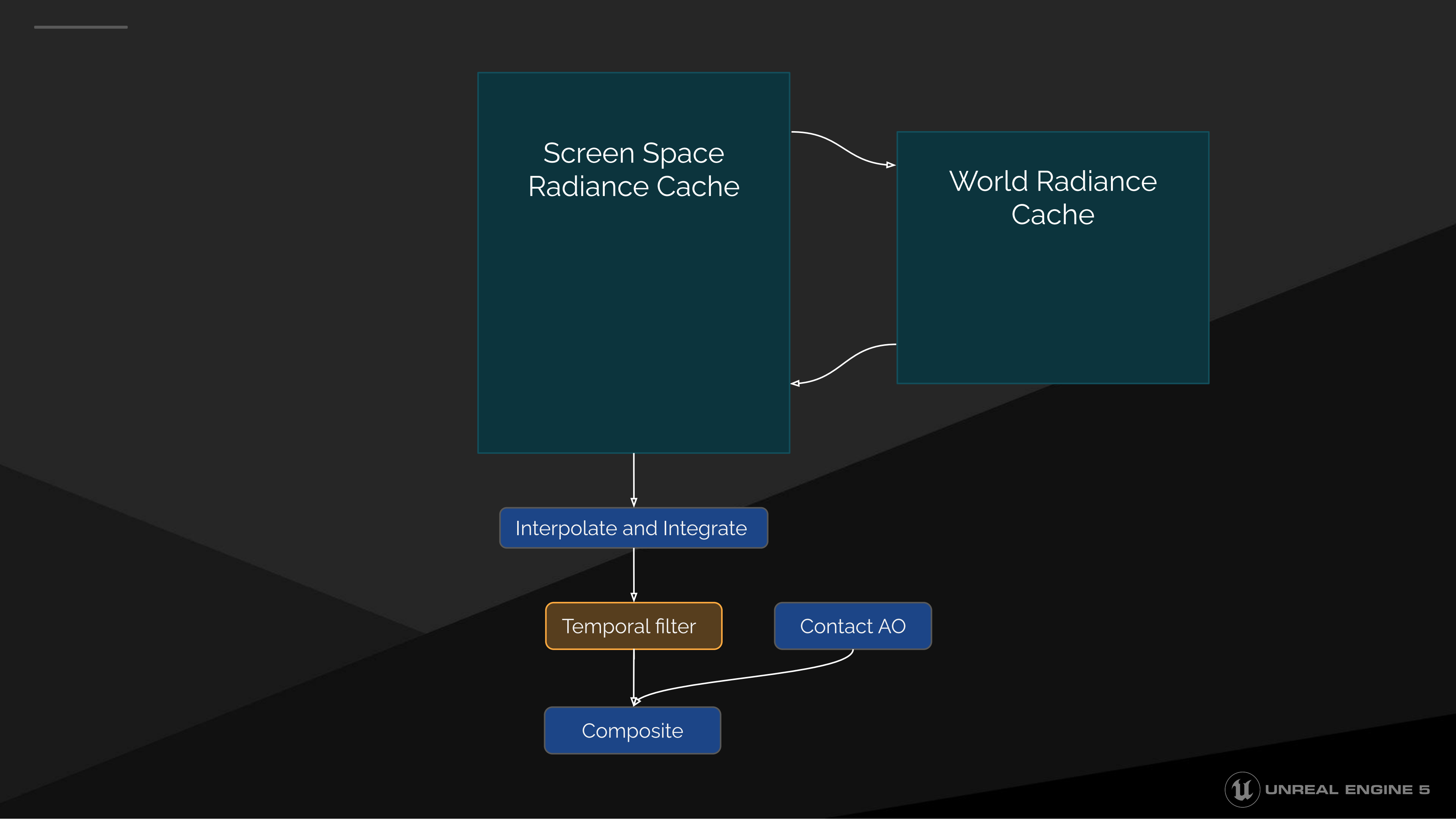
We cache them by carrying over the previous frame’s probes that are still needed in the new frame.  Then we trace new probes for positions that were revealed by camera, or scene movement.

We re-trace a subset of those probes to propagate lighting changes through the world.  Since last year we’ve improved this by using a GPU priority queue to select a fixed number of probes to update, which gives us a fixed update cost for the whole cache, even though it’s operating on variable input.

# “The Matrix Awakens” Night Mode



“The Matrix Awakens” has an experimental night mode, which is lit entirely with emissive meshes.  Most of the lighting in this scene is coming from the small, bright light bulb meshes so handling their direct lighting through our GI method is an absolute stress test, as we’re not explicitly sampling the light sources.  The World Space Radiance Cache solves their direct lighting much more accurately with its higher directional resolution, and is temporally stable, which doesn’t come across in this screenshot comparison.



Screen Space Radiance Cache

World Radiance Cache

Interpolate and Integrate

Temporal filter

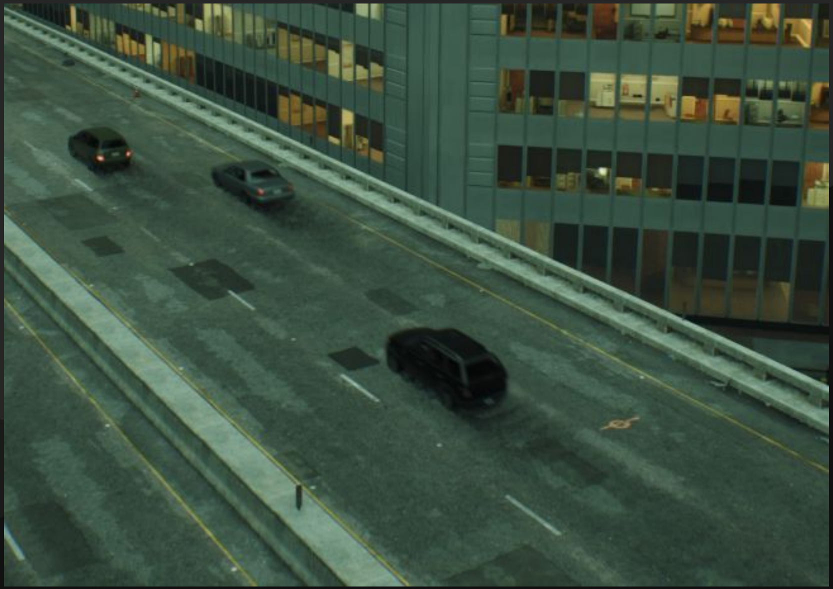
Contact AO

Composite

Now moving on to the full resolution temporal filter,

# Temporal filter

- Jittering probe position requires reliable temporal filter
- Using depth + normal rejection
  - Stable results, but also slow reaction to lighting changes - ghosting

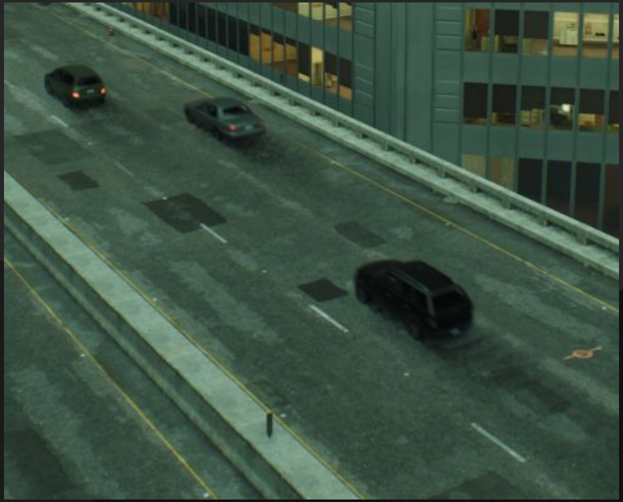


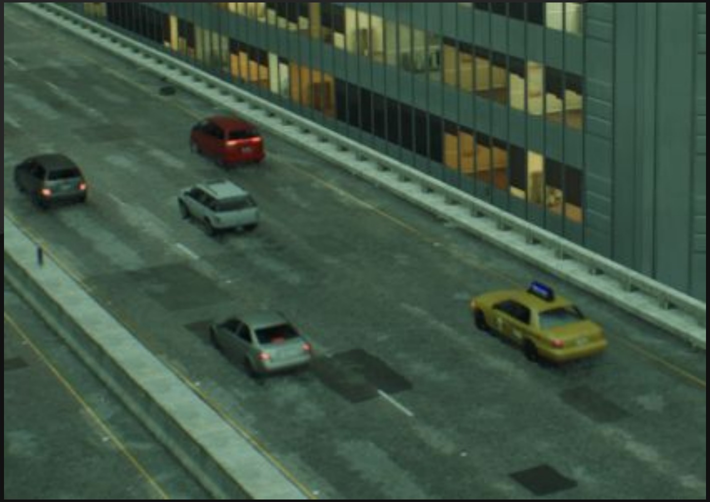
The temporal filter is needed to cover our jittering of the probe positions and directions.  We need a stable temporal filter, so we can't use neighborhood clamp, instead we reject the history based on depth and normal differences.

This gives very stable results, which we need, but also causes a slow reaction to lighting changes, which shows up as ghosting behind moving objects.

# Improvements

- Faster temporal accumulation when traces hit fast moving object
- Apply shortest range occlusion (16 pixels) after temporal filter

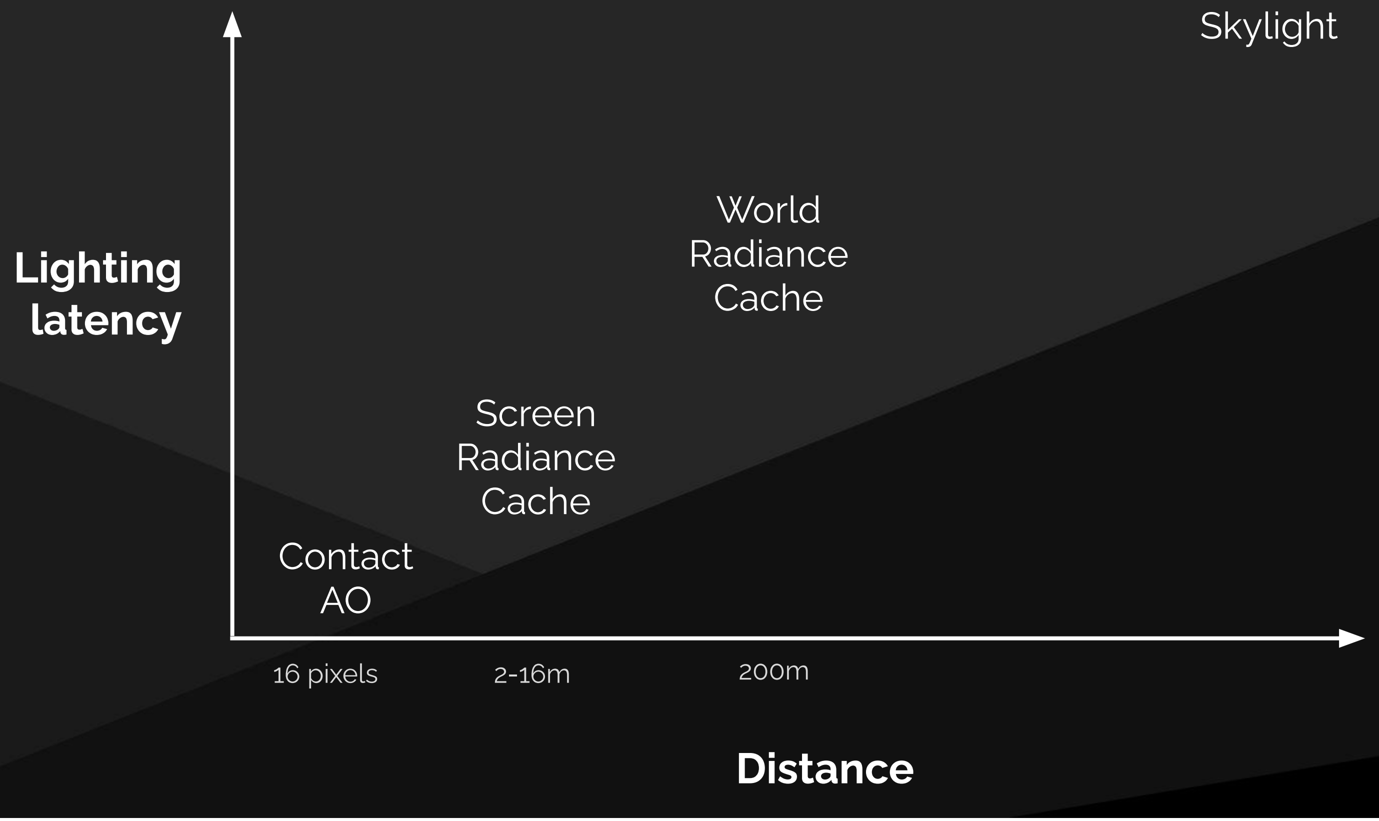




We improve this by detecting when traces hit fast moving objects, and speeding up the temporal filter for pixels that have most of their lighting coming from fast moving objects.

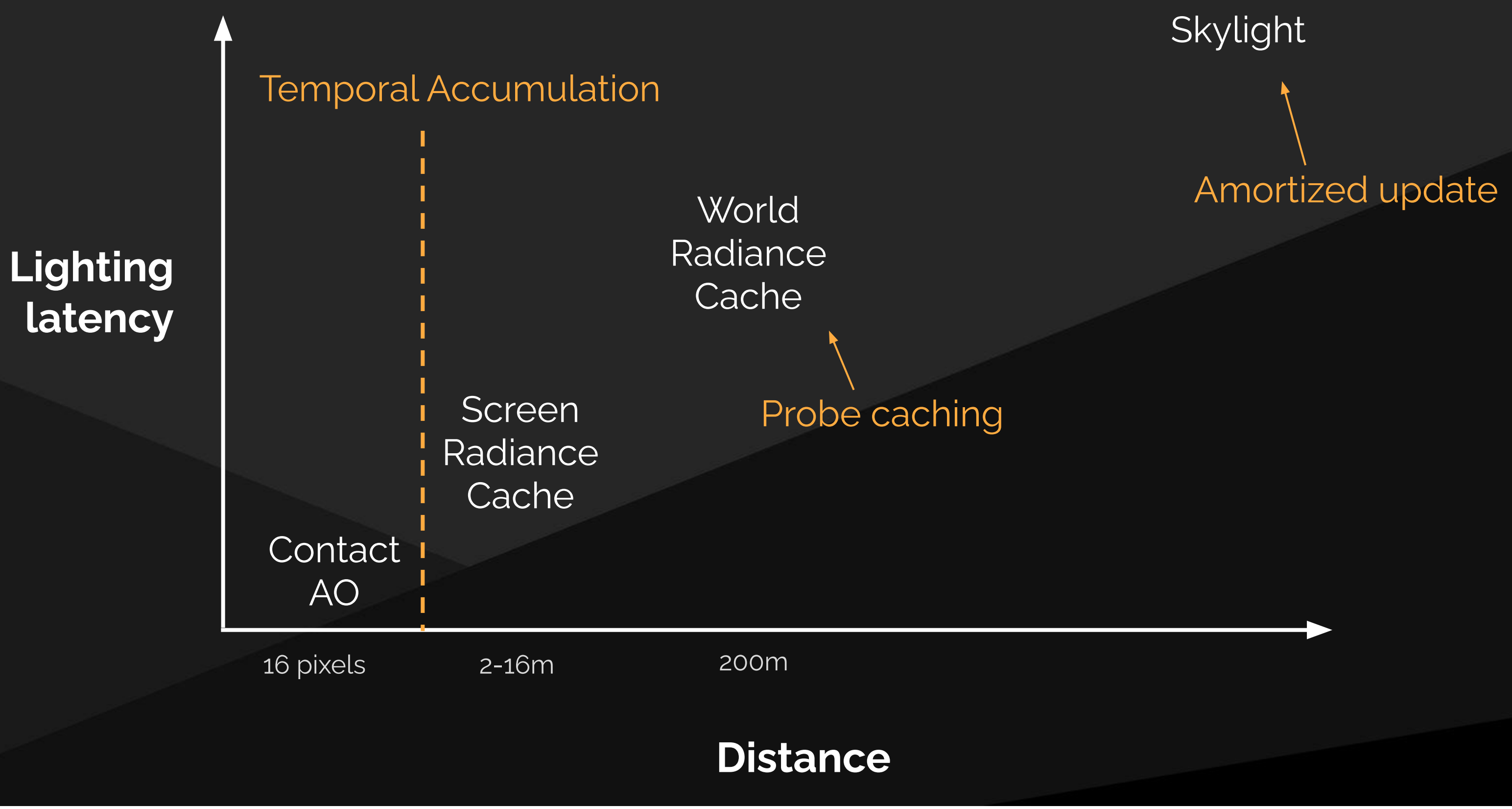
We also apply the shortest range occlusion after the temporal filter, so it won't have any lag.

# Caching opportunity by distance



With most dynamic scene changes, we can afford more latency in the lighting propagation in the distant lighting.  The shortest range indirect lighting can't afford any latency, but the skylight can get away with a lot of latency.  It's interesting that we're already setup to take advantage of this caching opportunity because our Final Gather separates the radiance into distance ranges and solves them with different techniques.

# Caching opportunity by distance



We can take advantage of allowable latency in the Screen Space Radiance Cache by temporally accumulating it, while the World Space Radiance Cache can reuse whole probes from previous frames, and the skylight can be slowly updated over many frames.

# Translucency and Fog GI challenges

- Have to support any number of lit translucent layers
- GI needed everywhere in the visible depth range for fog
- Volume vs surface
- Much smaller budget than opaque

UNREAL ENGINE 5

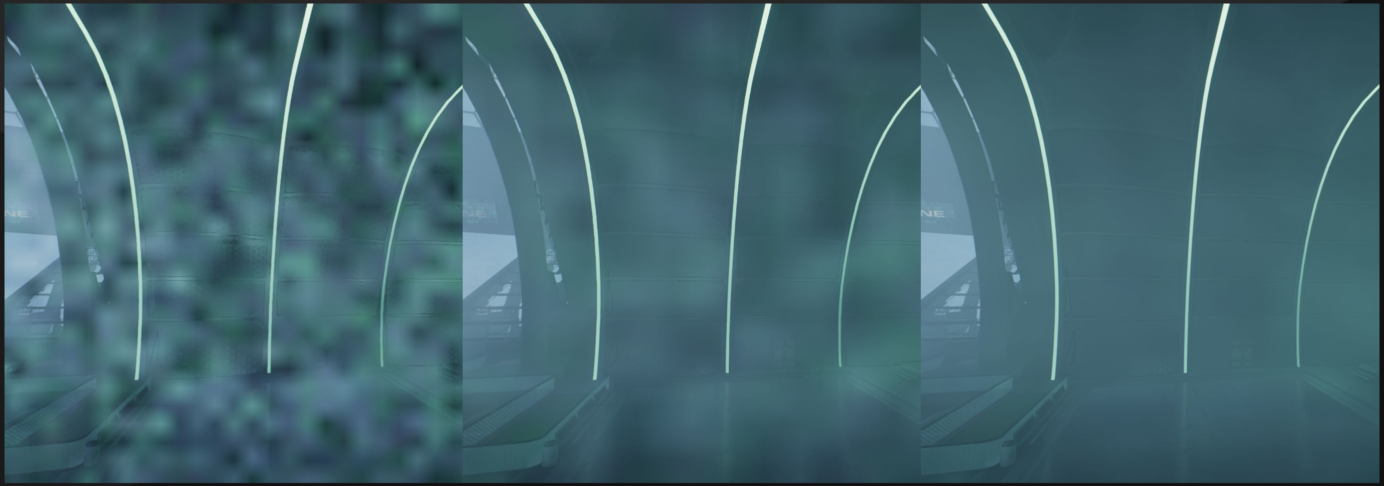
Now moving on to GI on Translucency and Fog.  For Translucency we have to support any number of layers, and you can see what those translucent dust particles look like if we don’t shadow the skylight.

For Fog we need to solve GI everywhere in the visible depth range, and we need to solve it over the entire incoming sphere, instead of a hemisphere with a normal.

We also have about 1/8th the budget of the Opaque Final Gather, so we’re going to need a very fast technique.

# Volumetric Final Gather

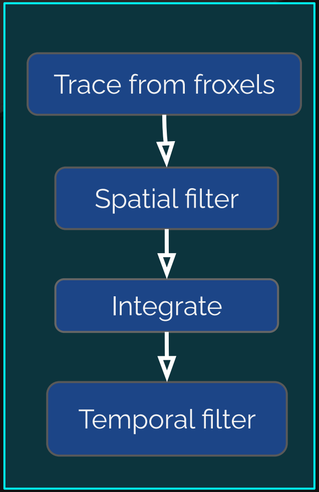
- Cover the view frustum with a probe volume (froxel grid)
  - Trace 3x3 or 4x4 octahedral probes
  - Skip invisible probes with HZB occlusion tests
- Spatially filter and temporally accumulate
- Pre-integrate into SH2 Irradiance
- Forward translucency pass interpolates



Raw Traces

+Spatial filter

+Temporal Accumulation



Trace from froxels

Spatial filter

Integrate

Temporal filter

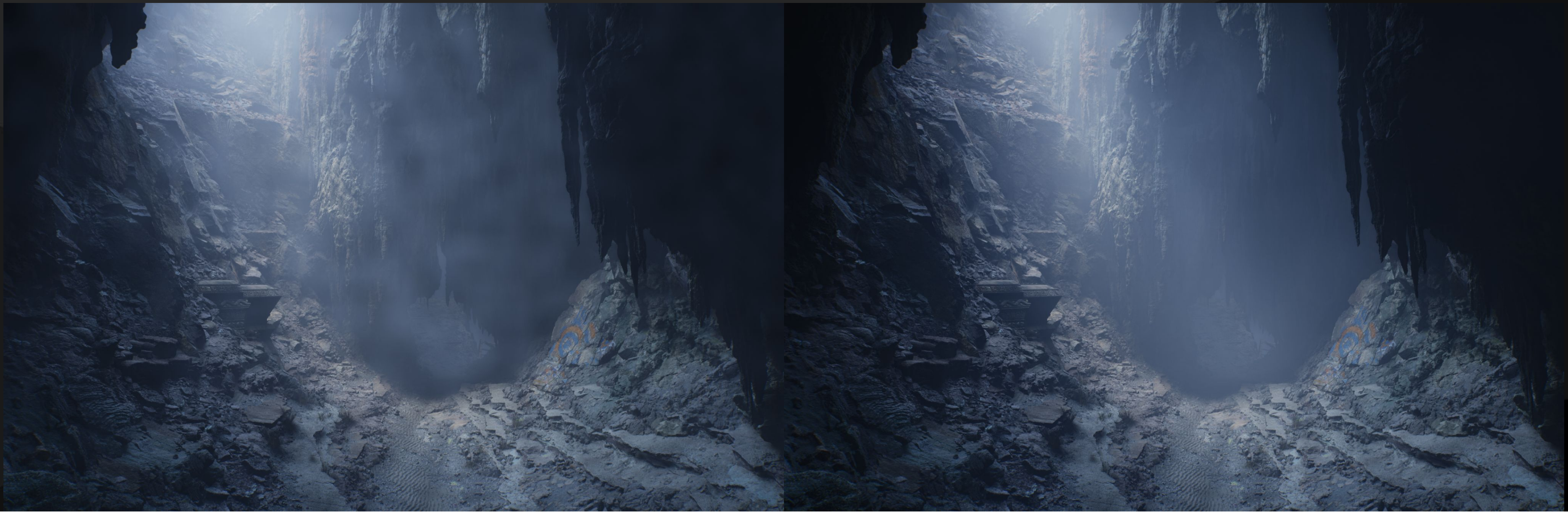
Our Volumetric Final Gather covers the view frustum with a probe volume, which is a froxel grid.  We trace octahedral probes and skip invisible probes determined with an HZB test.

After tracing to find Radiance, we do a spatial filter on the radiance and temporally accumulate it to reduce noise with our small number of traces.

We then pre-integrate into Spherical Harmonic Irradiance and the forward translucency pass, or the volumetric fog pass, interpolates the irradiance.

# Even 4x4 traces not enough for distant lighting

- Use another World Radiance Cache for distant lighting
  - Higher directional resolution - 16x16
  - Stable distant lighting

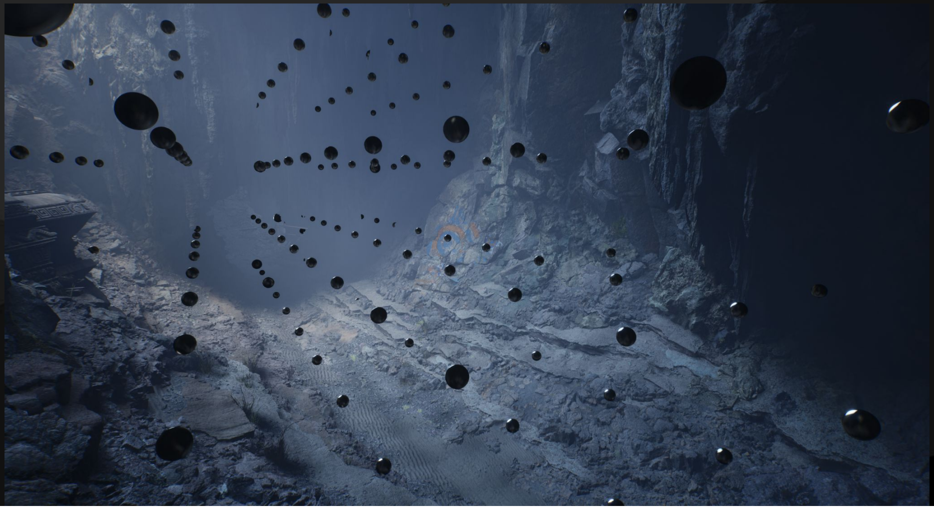


Even with the larger number of traces, we can’t solve the noise in distant lighting, like this cave that’s lit up entirely by the sky lighting coming through a small hole.

We use another World Space Radiance Cache for the distant lighting, which again has higher directional resolution and gives stable distant lighting.

# Translucency World Radiance Cache details

- Place world probes around visible froxel grid
- Trace and prefilter radiance into mip maps
- Interpolate when shortened froxel traces miss geometry
  - Mips reduce aliasing, 16 world rays : 1 froxel ray
- Overlap with opaque World Radiance Cache
  - Becomes almost free



To populate this World Radiance Cache we need to place probes everywhere around the visible froxel grid.  We then trace from the probes and prefilter the radiance into mip maps.

We shorten the Froxel traces and when they miss, we interpolate from the world probes.  There are 16 times more world probe rays than froxel rays so the mips reduce aliasing.

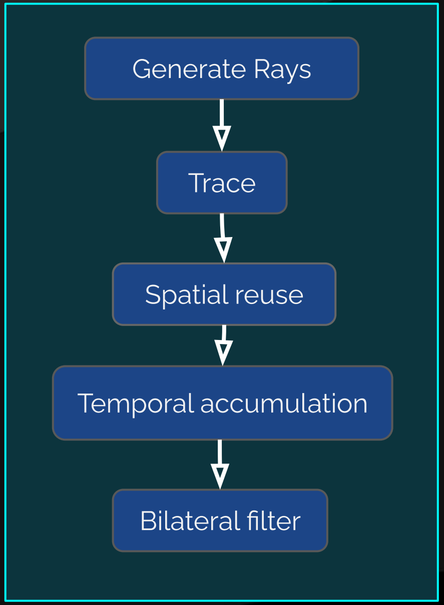
We overlap this new translucency Radiance Cache with the Opaque World Radiance Cache, so that it's many dispatches fit in the gaps and it's almost free.

# Reflections

Now moving on to Lumen Reflections

# Stochastic integration with screen space denoising

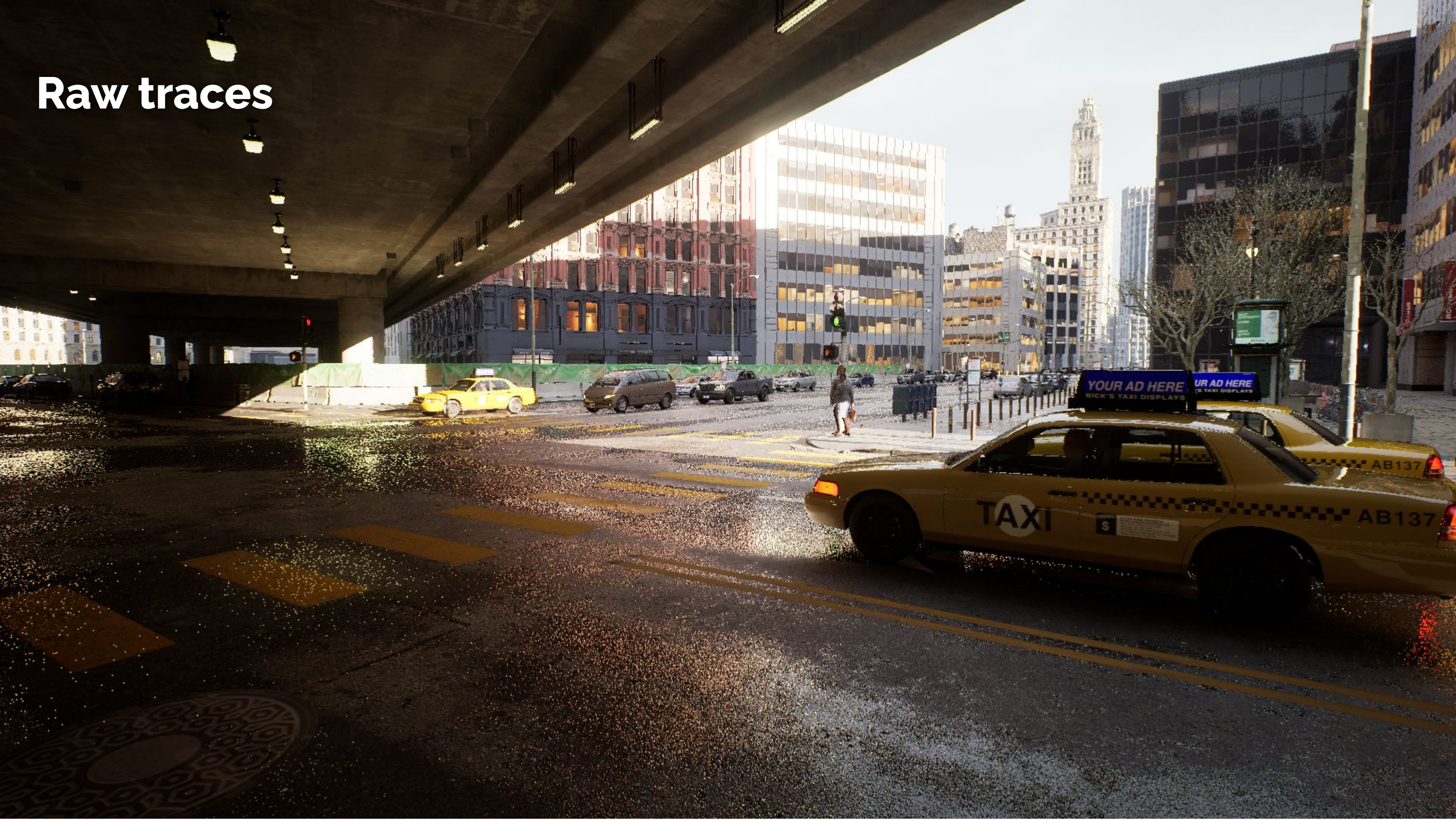
- “Stochastic Screen-Space Reflections”
- “Stochastic All The Things: Raytracing in Hybrid Real-Time Rendering”
  - [Stachowiak 2015 & 2018]



For reflections we use stochastic integration with screen space denoising, based on these excellent talks from Tomasz Stachowiak.

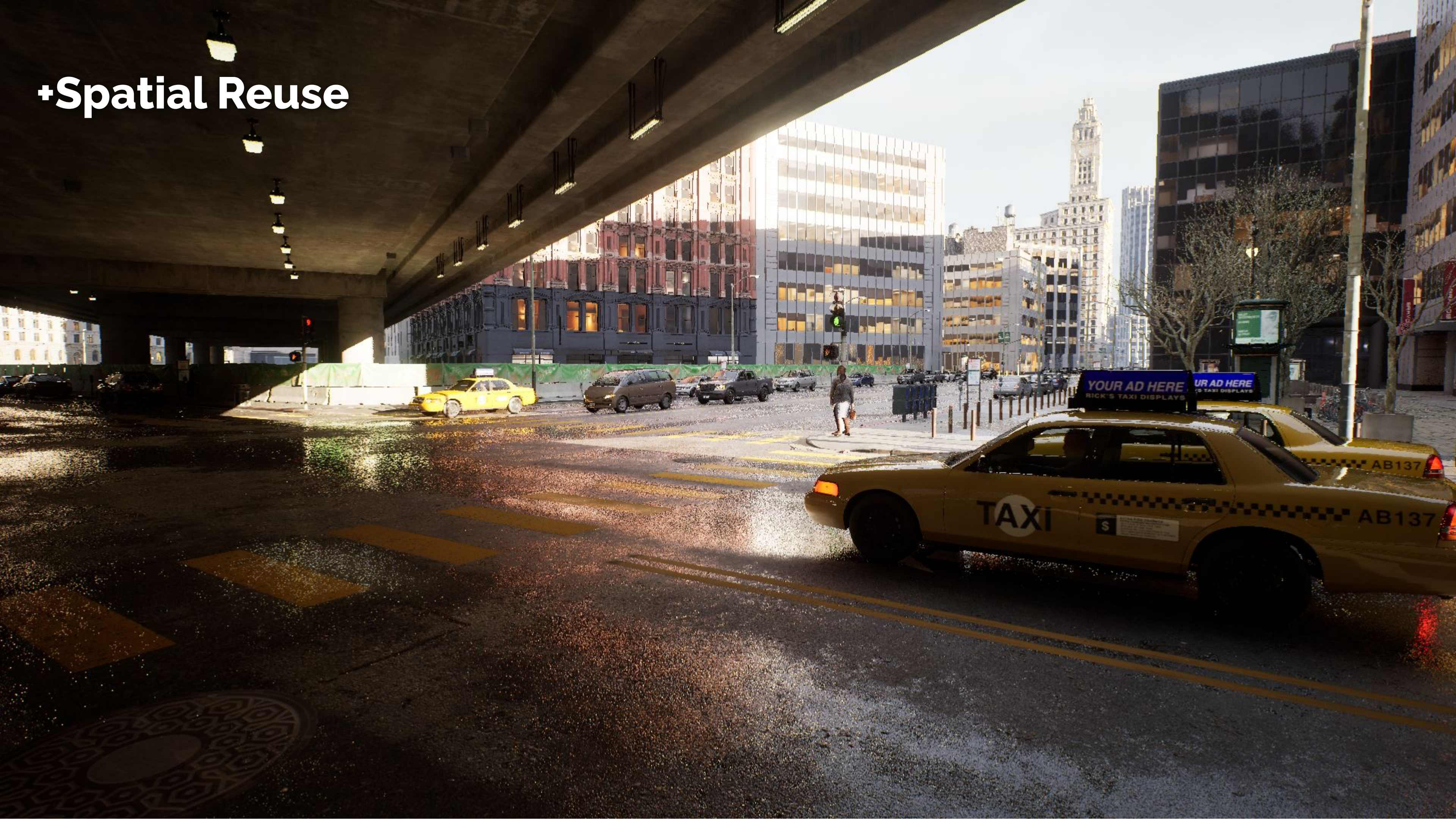
Looking at our pipeline at a high level, first we generate rays by importance sampling the visible GGX lobe, then we trace the rays using our ray tracing pipeline.  Then we run our spatial reuse pass, which looks at screen space neighbors and reweights them based on their BRDF. Then we do a temporal accumulation, and finally a bilateral filter to clean up any remaining noise.

# Raw traces



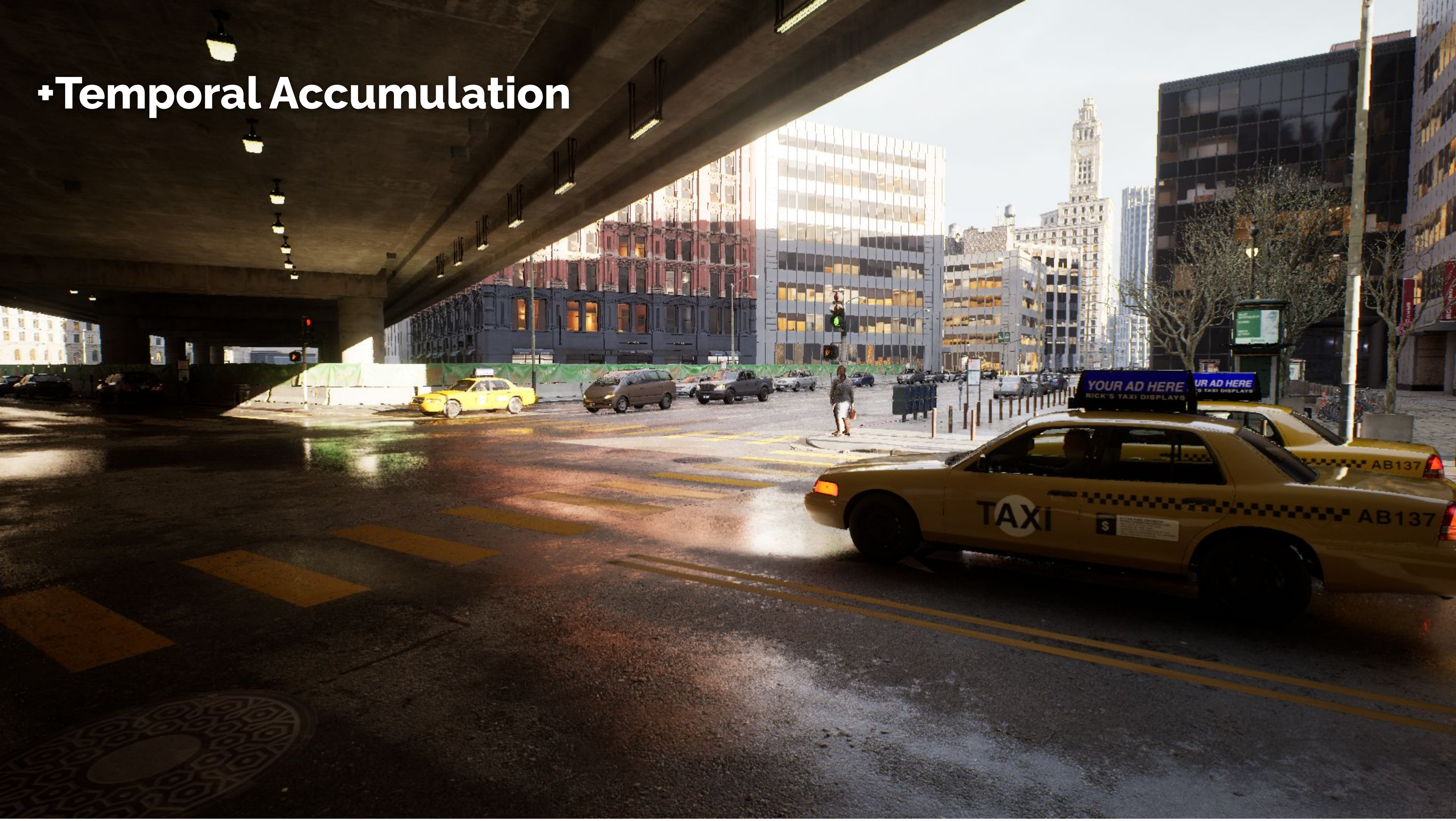
Here’s what those steps look like in “The Matrix Awakens”.  Starting from the raw traces we have a lot of noise,

# +Spatial Reuse



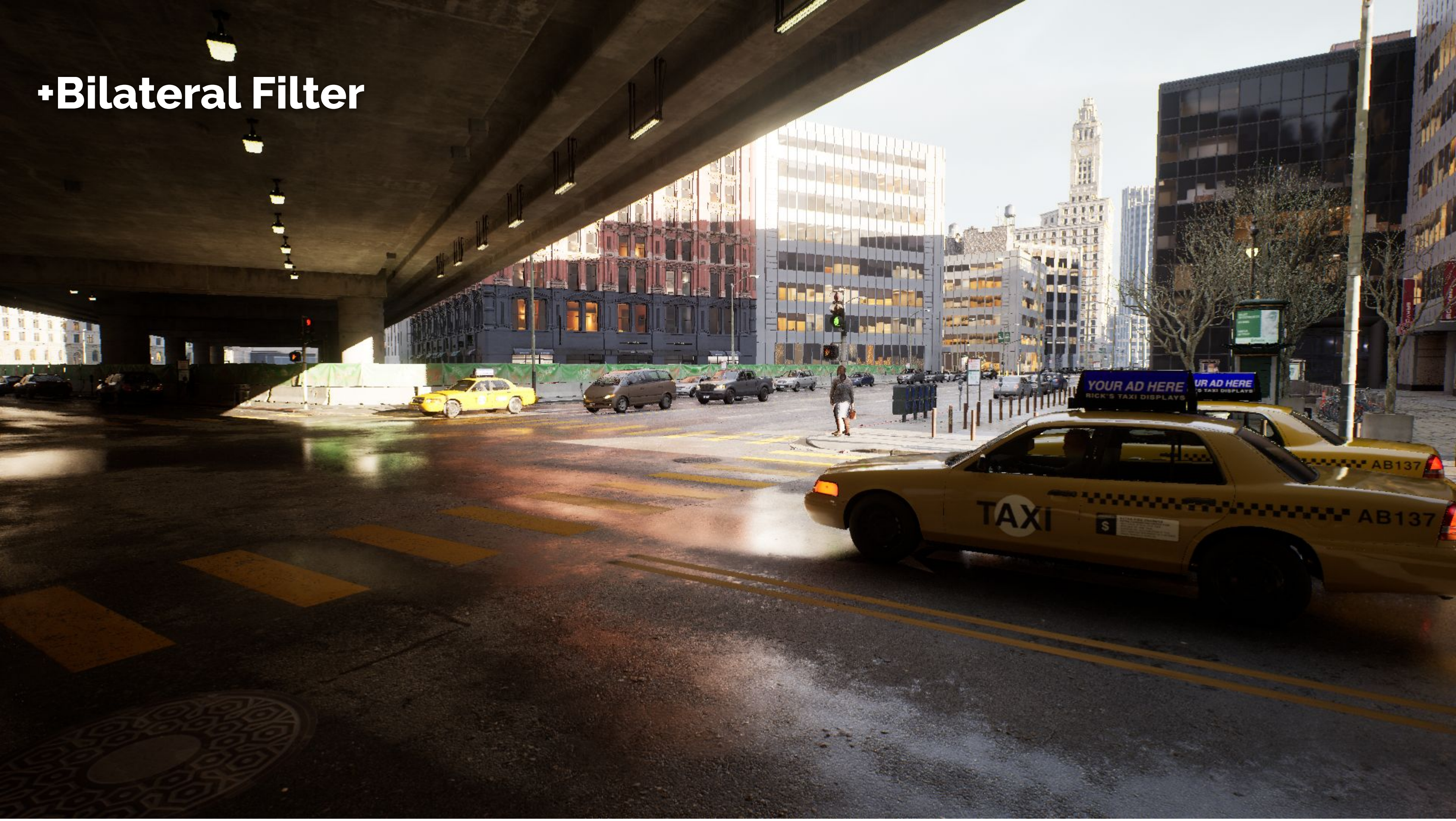
which is reduced after doing spatial reuse, but still visible.

# +Temporal Accumulation



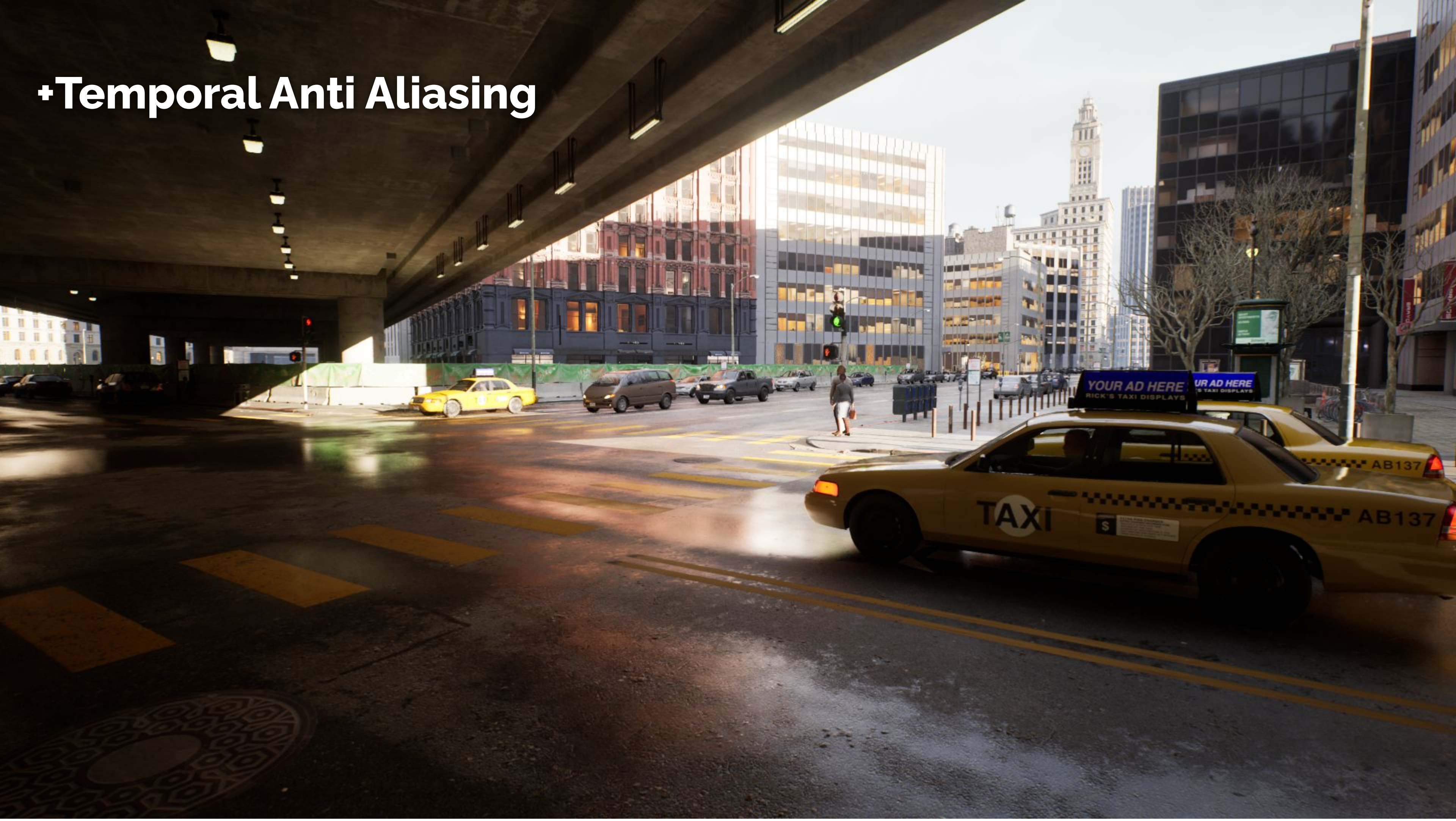
Then we apply Temporal Accumulation and the fireflies are significantly reduced, but there’s still some noise in the very bright areas.

# +Bilateral Filter



We clean these up with the bilateral filter,

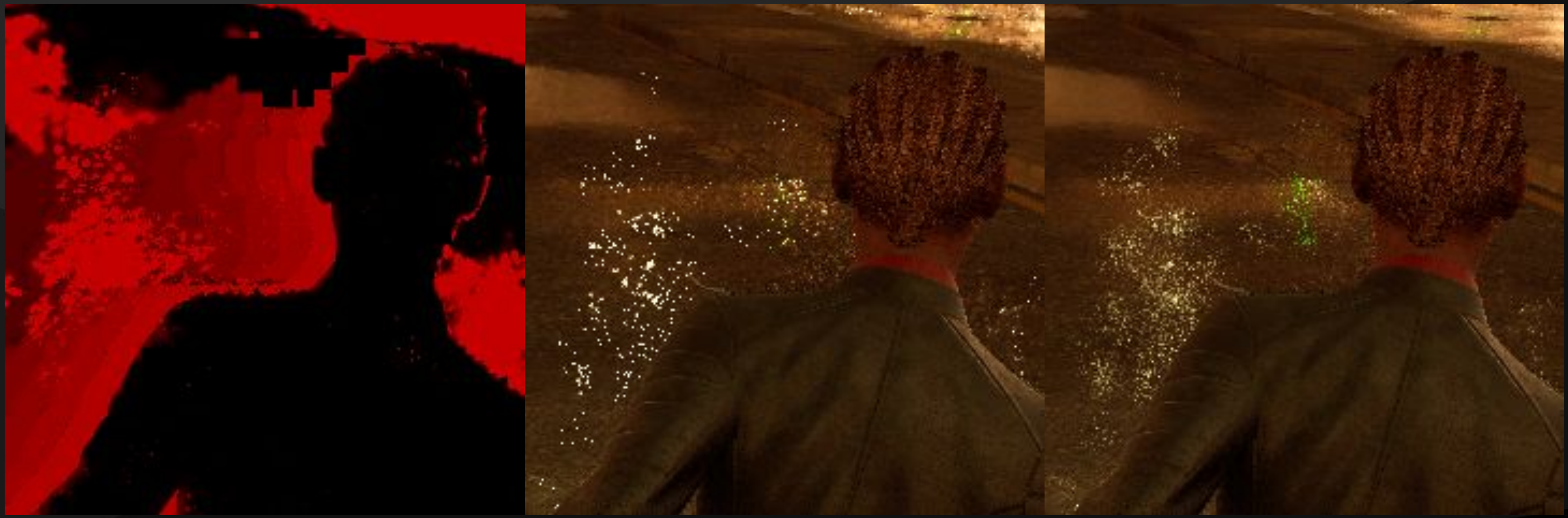
# +Temporal Anti Aliasing



And finally the full frame Temporal Anti Aliasing which cleans it up a bit more.

# Bilateral Filter as a last ditch effort

- Run on areas with high variance after the spatial reuse
- Force on newly revealed areas with double strength filter
- Tonemapped weighting to remove fireflies



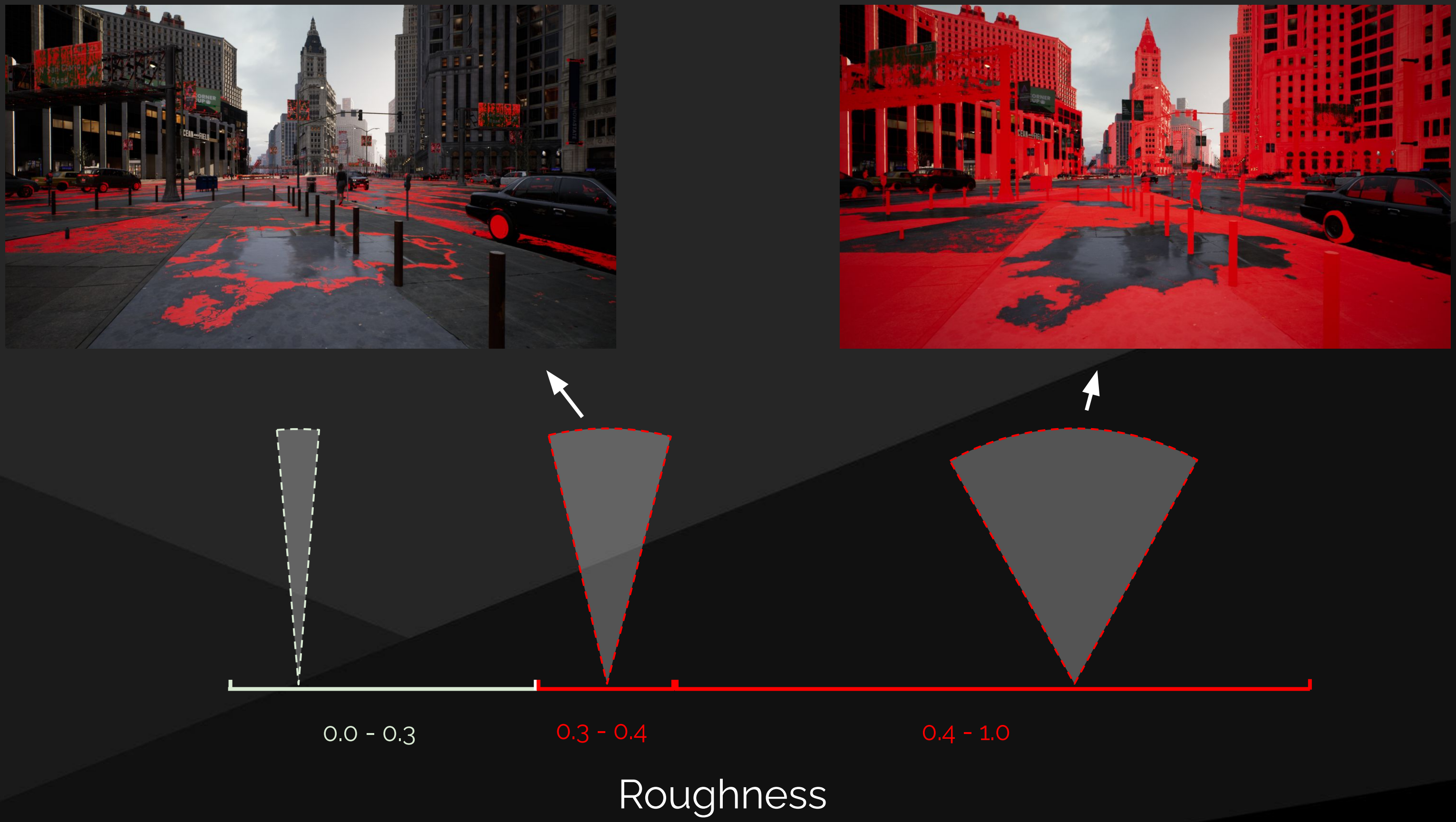
Variance + Disocclusion

Original (no TAA)

After Bilateral Filter (no TAA)

The Bilateral Filter is our last ditch effort for when the physically based reuse isn't enough. We run it on areas that had high variance after the spatial reuse pass, and we force it on at double strength in areas that were newly revealed by disocclusion, which don't have any temporal history. We use tonemapped weighting in the Bilateral Filter to remove fireflies, which would crush our highlights if used in the spatial reuse pass, but works perfectly here.

# Dealing with incoherency

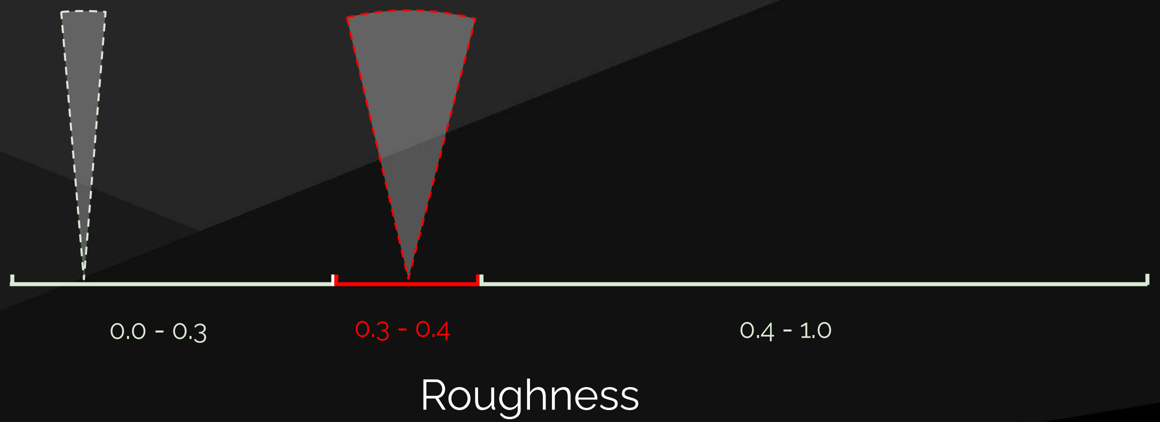


One of the problems we have with reflections is the slow tracing of incoherent rays. When the material’s roughness approaches 1, the GGX lobe approaches diffuse and our rays become very incoherent.

The roughest reflections from .4 to 1 often cover half the screen and they are a significant optimization opportunity. Then there are the glossy reflections from .3 to .4 which require more directionality but are also quite slow to trace.

# For rough reflections

- Screen Radiance Cache has enough directional resolution (8x8)
  - Skip reflection rays
  - Resample Screen Radiance Cache
    - Importance sample GGX, interpolate radiance from probes
- Reduces cost of reflections 50-70%

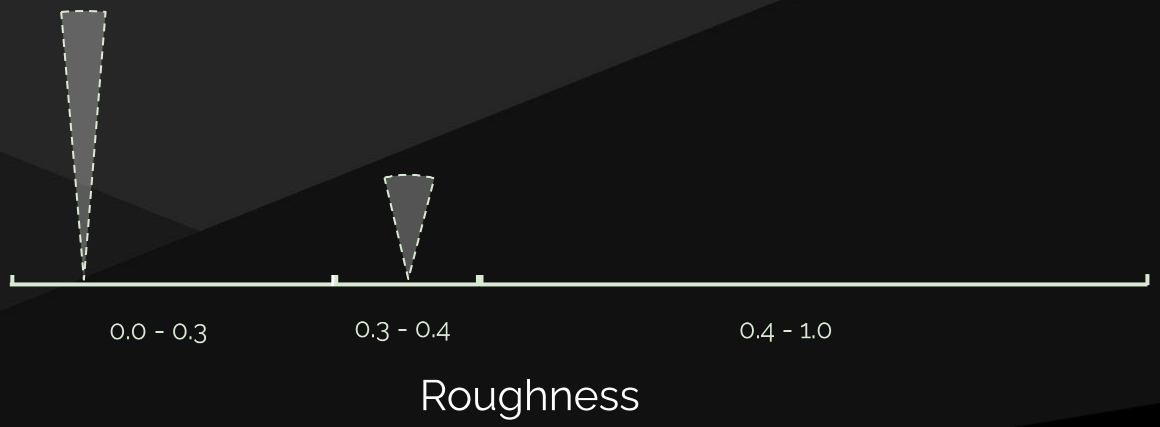


We can solve the incoherency in the roughest reflections by just reusing the work that we did for Diffuse GI. The Screen Space Radiance Cache has enough directional resolution for a wide specular lobe, and we can just resample it, by importance sampling the GGX lobe to get a direction, and then interpolating radiance from the screen probes.

That reduces the cost of reflections, depending on how much of the screen can skip the reflection rays, but in most scenes we see the cost reduced by 50% to 70%.

# For glossy reflections

- World Radiance Cache has enough directional resolution (32x32)
  - Shorten the reflection ray and interpolate from World Radiance Cache on miss
- Removes the need for ray binning and sorting
  - Rays are already ordered by origin
  - Reduced directional divergence by shortening
- Reduces cost of road reflections by 16%



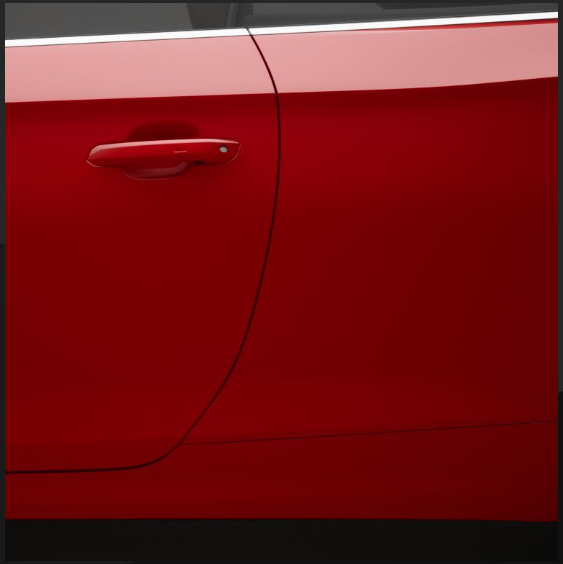
For that middle range of glossy reflections, the Screen Space Radiance Cache doesn't have enough directional resolution, but the World Space Radiance Cache does.  We shorten the reflection ray, and interpolate from the World Space Radiance Cache on miss.

This removes the need for ray binning and sorting because we've reduced directional divergence by shortening the rays, and they're already ordered by their origins.

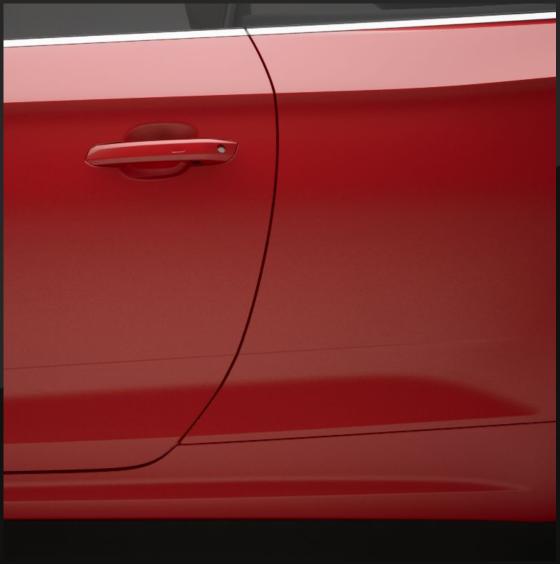
In "The Matrix Awakens", we see the cost of the road reflections reduced by another 16%.

# Clear Coat double reflections efficiently through reuse

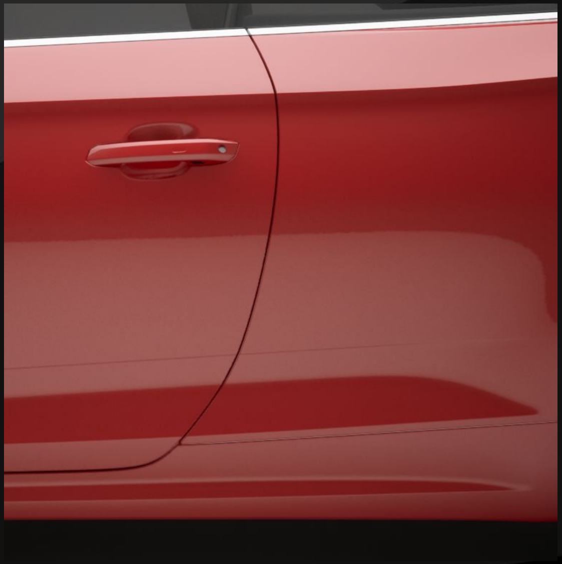
- Glossy bottom paint layer: reuse Screen Radiance Cache
- Smooth Clear Coat: Reflection pipeline



Diffuse GI



+Paint reflection

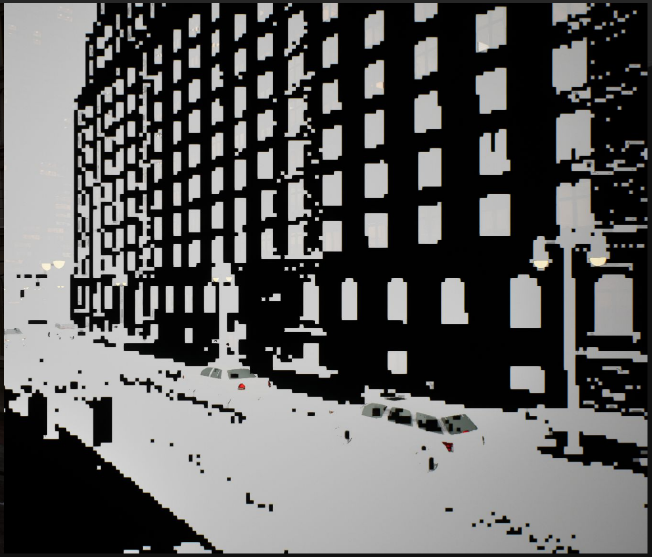


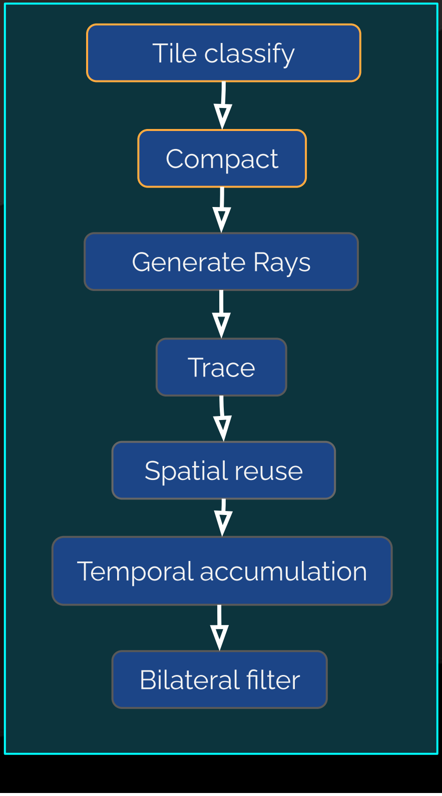
+Clear Coat reflection

We also have very efficient Clear Coat reflections through reuse.  The glossy bottom paint layer can reuse the Screen Space Radiance Cache, and we only have to trace new rays for the top Clear Coat layer.

# Tile based pipeline

- Skip sky and areas that reused diffuse rays efficiently
- Can run multiple times on a small part of the screen efficiently
  - Opaque, Translucency, Water
- Compact tile list with z-order to preserve ray origin coherency



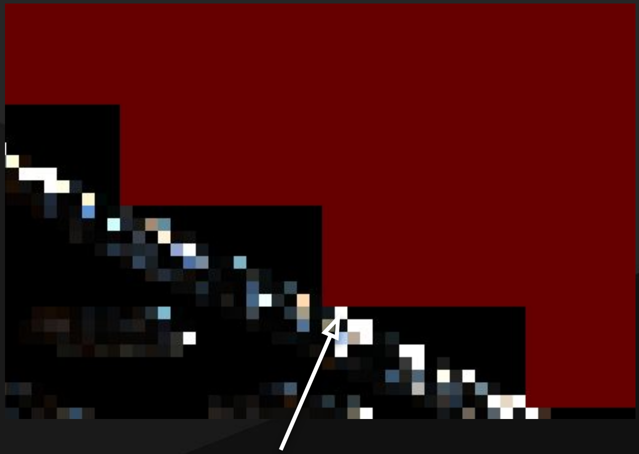


Our reflection pipeline is based on tiles, which allows us to skip the sky and areas that reused the diffuse rays very efficiently.  That's important because we have many dispatches in the pipeline, which you can see on the right, and even more in the tracing pipeline, and we can operate on just the parts of the screen that need work.

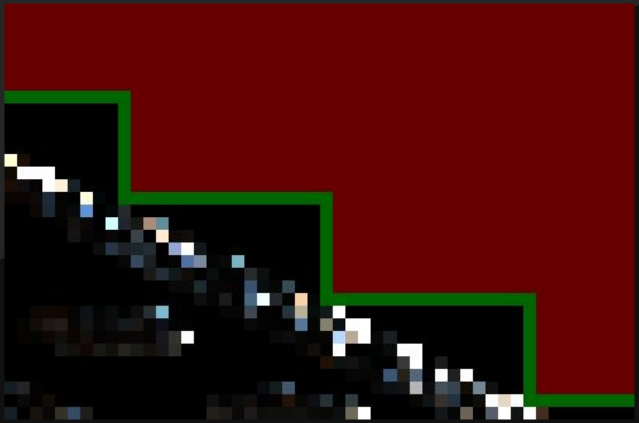
We actually run our entire reflection pipeline multiple times, depending on the scene.  We run it at least once for opaque, but we might run it again for translucency reflections and for water reflections, so it's important that we can operate only on the parts of the screen that need it.

# Temporal filter neighborhood clamp reads out of tile

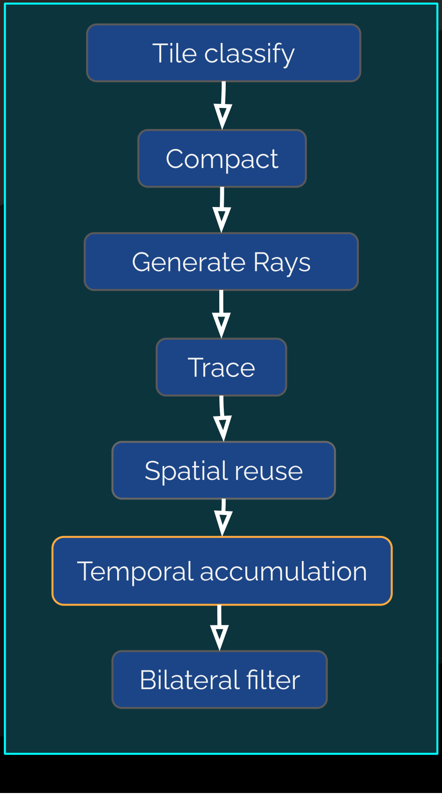
- Clear unused regions of texture, or branch in temporal filter?
- Faster: Spatial reuse shader reassigns threads to clear tile border if neighbor is culled



Pixel has uninitialized neighbors



Tile borders cleared



Tile classify

Compact

Generate Rays

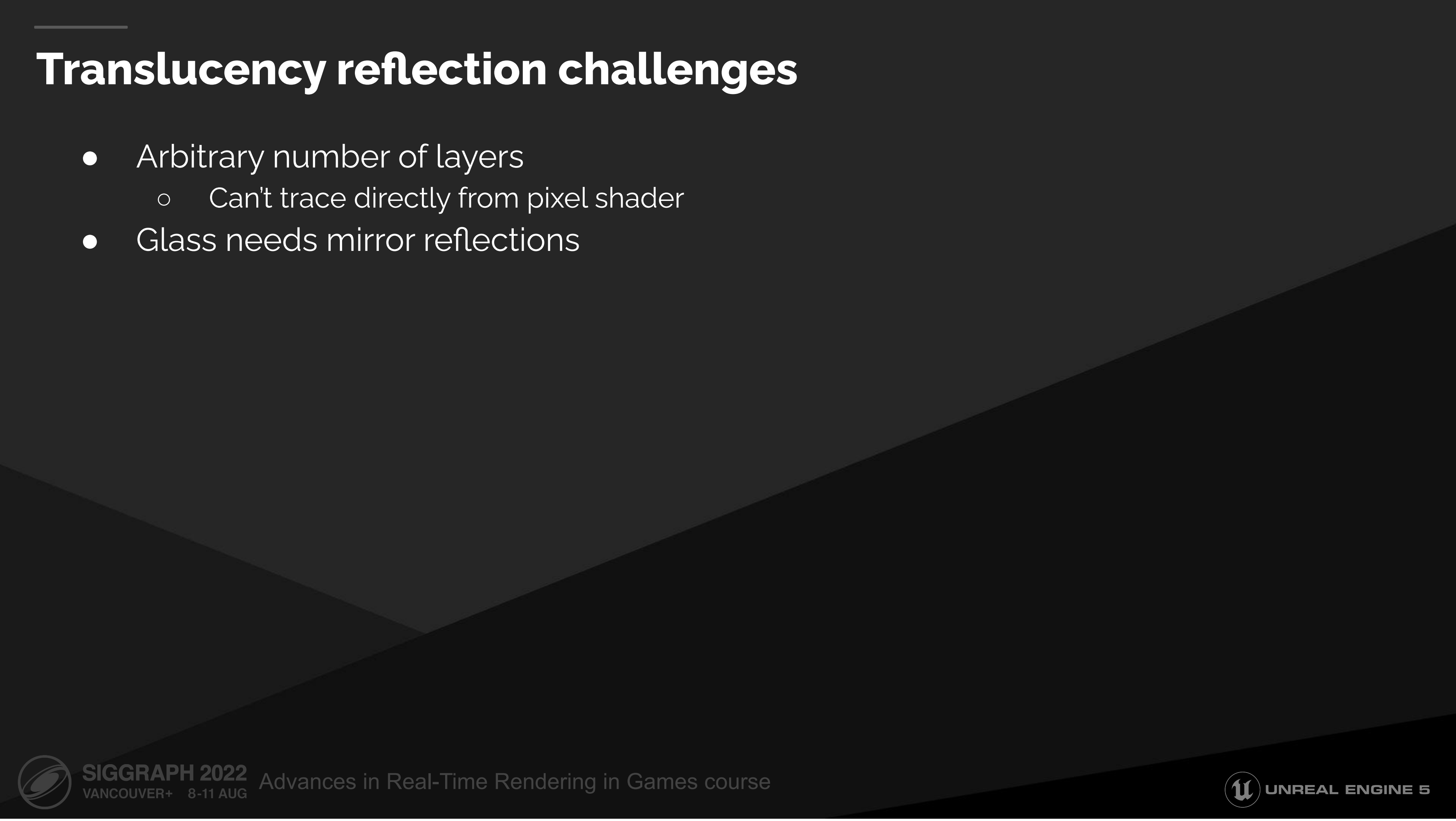
Trace

Spatial reuse

Temporal accumulation

Bilateral filter

There's a gotcha with implementing a tile based reflection pipeline - all of the denoising passes read from neighbors which may not have been processed. For the temporal filter's neighborhood clamp, we could clear the unused regions of the texture, or branch in the temporal filter. It's slightly faster to have the pass that runs before the temporal filter clear the tile border. We can only clear texels in unused tiles, to avoid a race condition with the other threads of the spatial reuse pass.



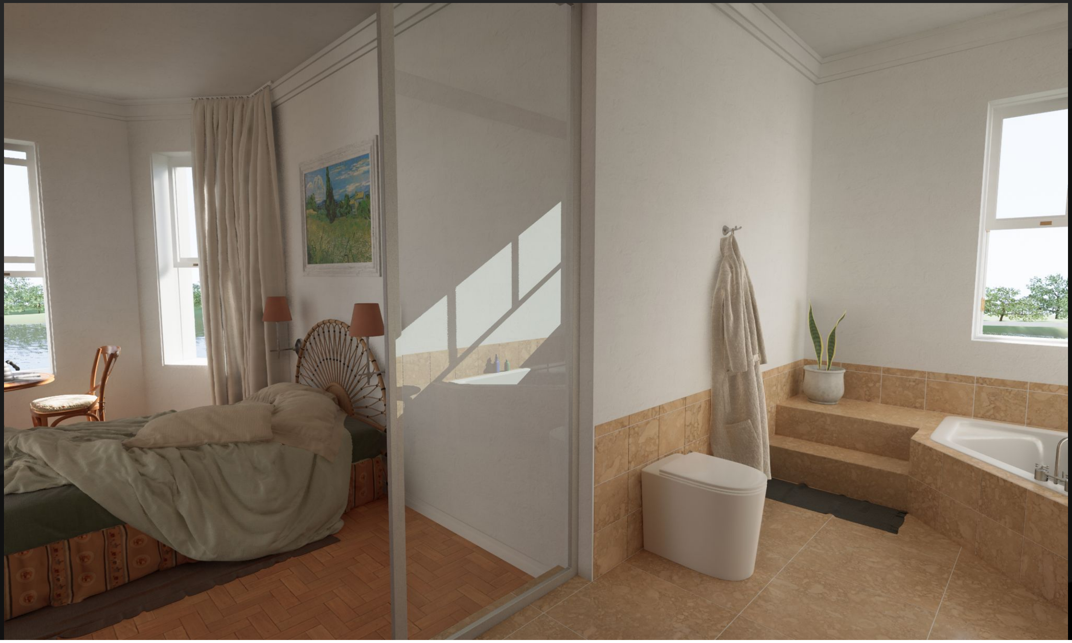
# Translucency reflection challenges

- Arbitrary number of layers
  - Can’t trace directly from pixel shader
- Glass needs mirror reflections

For reflections on translucency, we need to support an arbitrary number of layers, and we can’t trace directly from the pixel shader, so we need to solve them outside the pixel shader, which is then interpolated to the pixel shader.  At the same time, glass needs mirror reflections, so we can’t do any interpolation there.

# Front layer traces new reflection rays

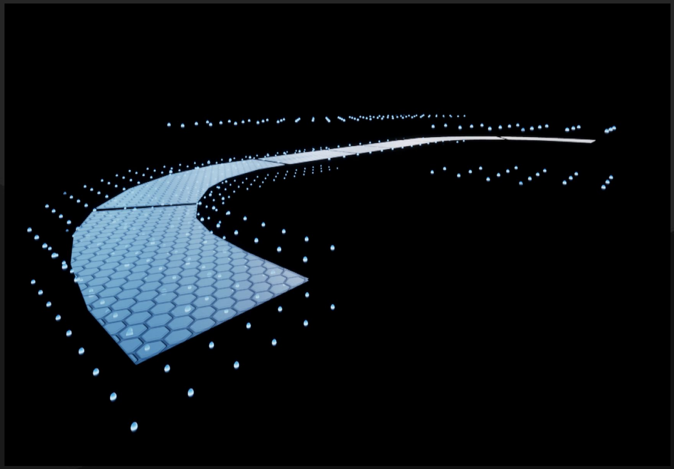
- Extract the front layer minimal GBuffer with depth peeling
- Run reflection pipeline on the valid pixels
  - Force to mirror, disable denoiser to reduce overhead

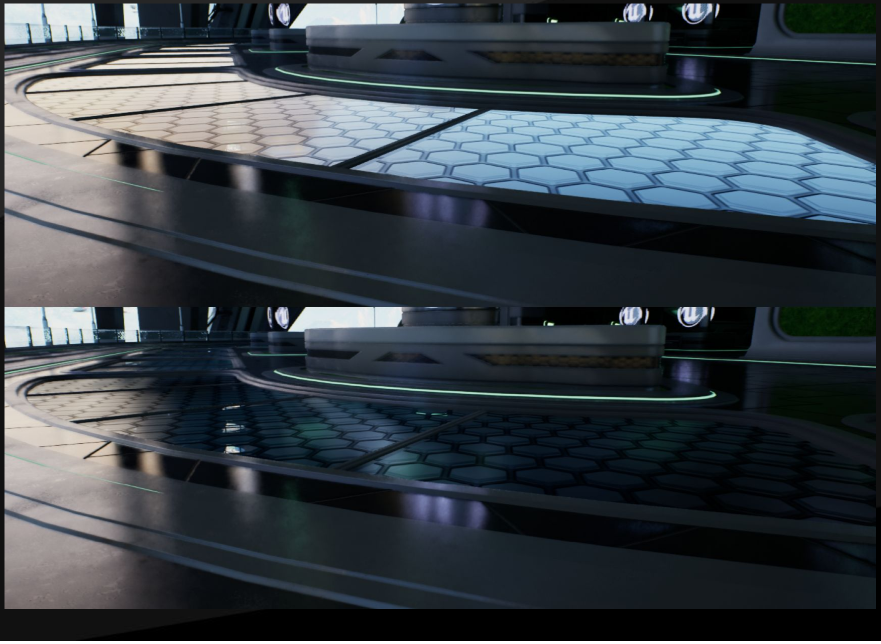


To provide the glass reflections, we extract the frontmost layer of the translucency into a minimal GBuffer using depth peeling. We then run the reflection pipeline again, only on the valid pixels, with the denoiser disabled to reduce the pipeline overhead.

# Additional layers use World Radiance Cache

- Same Radiance Cache as Opaque Final Gather, share work
- Mark probes that need to be cached
  - Rasterize translucent surfaces at low resolution
- Translucency forward shading interpolates from Radiance Cache



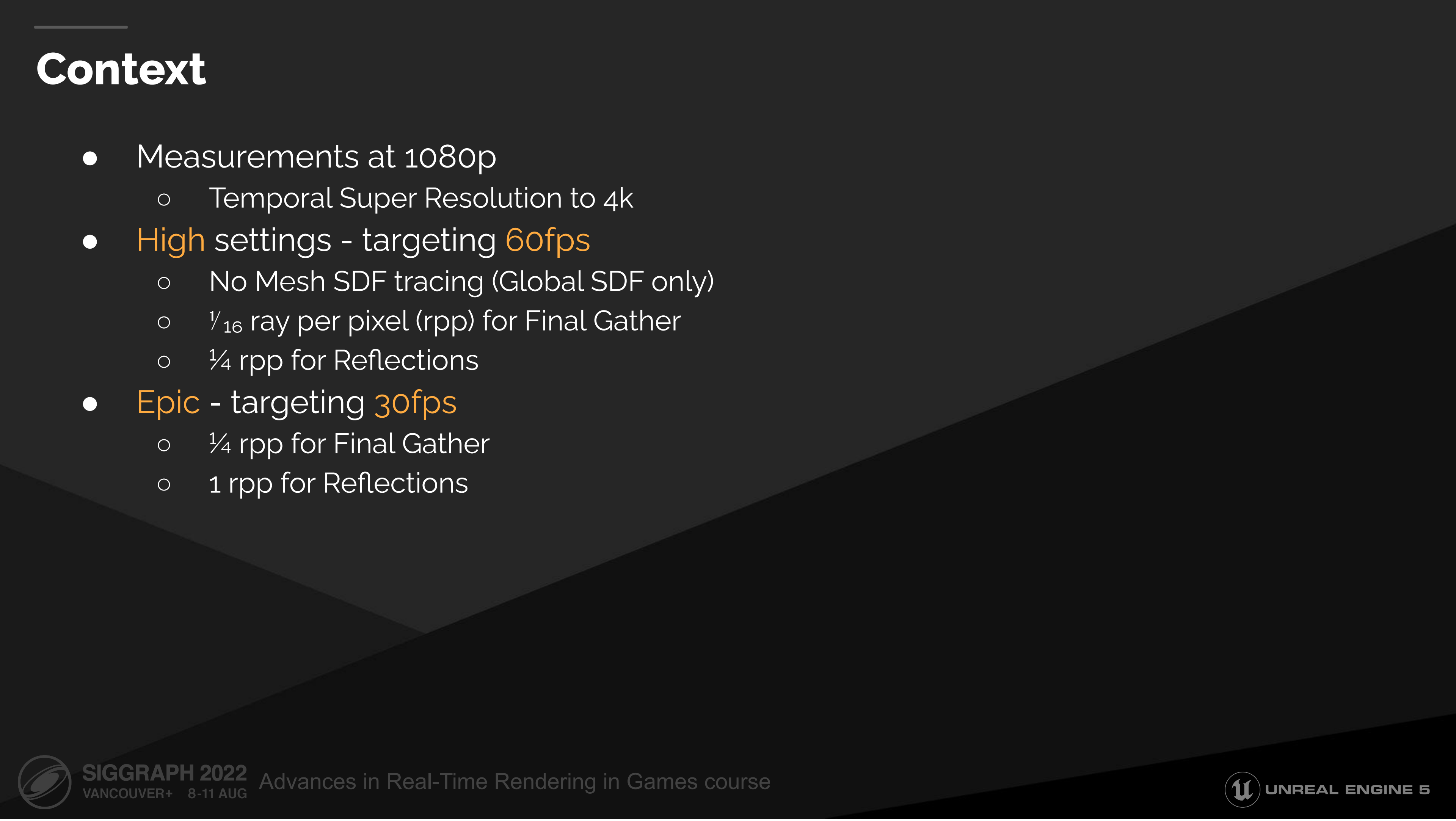


For the rest of the layers, we use the World Space Radiance Cache again.  We use the same Radiance Cache as the Opaque Final Gather, we just need to place some more probes.  We mark the new probes by rasterizing the translucent surfaces at a low resolution, and then mark the probes that are needed at each pixel's position.

Then we rasterize the translucency with any number of layers, and their pixel shader interpolates from the Radiance Cache for glossy reflections.

# Performance and scalability

Moving on to Lumen’s performance and scalability



# Context

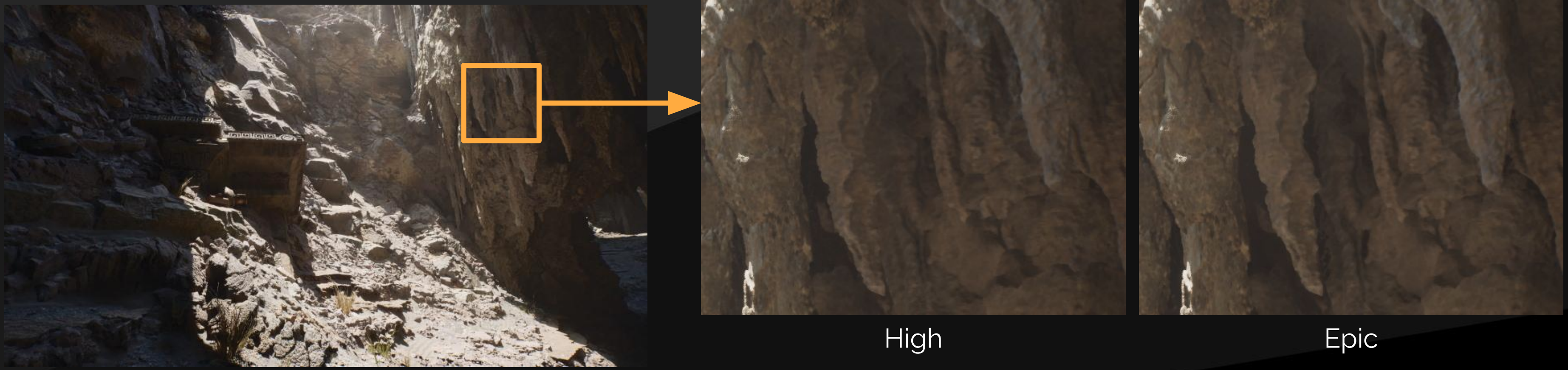
- Measurements at 1080p
  - Temporal Super Resolution to 4k
- High settings - targeting 60fps
  - No Mesh SDF tracing (Global SDF only)
  - $^{1}/_{16}$ ray per pixel (rpp) for Final Gather
  - ¼ rpp for Reflections
- Epic - targeting 30fps
  - ¼ rpp for Final Gather
  - 1 rpp for Reflections

All of the measurements shown here were captured at 1080p resolution, with Temporal Super Resolution outputting at 4k. We find that this gives much better final image quality than if we had run Lumen natively at 4k, with really low quality settings.

Our 'High' settings are targeting 60 frames per second, and our 'Epic' settings are targeting 30 frames per second. The 'High' settings don't have any Mesh SDF tracing, they're only using Global SDF tracing, the fastest method, and the 'Epic' settings have four times more rays per pixel.

# “Lumen in the Land of Nanite” - UE5 demo

| SWRT, Global SDF only, PS5 | | |
|---|---|---|
| | High ($^1/_{16}$ rpp) | Epic (¼ rpp) |
| Surface Cache | 0.47 ms | 0.93 ms |
| Final Gather | 2.07 ms | 3.46 ms |
| Reflections | 0.23 ms | 0.24 ms |
| Total | 2.77 ms | 4.63 ms |



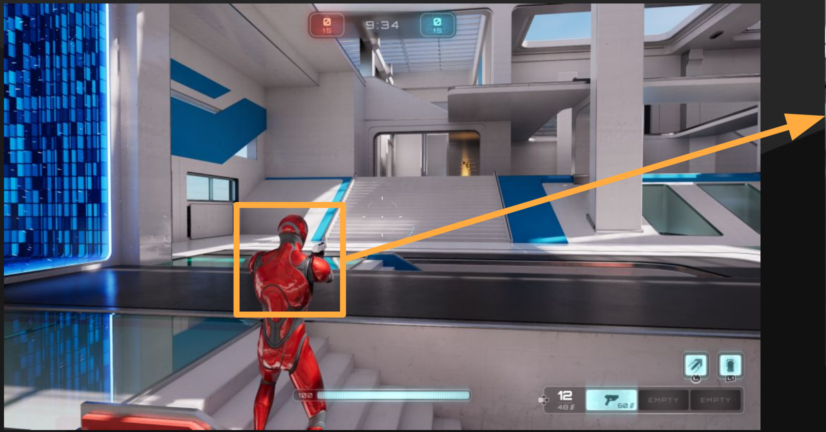
High

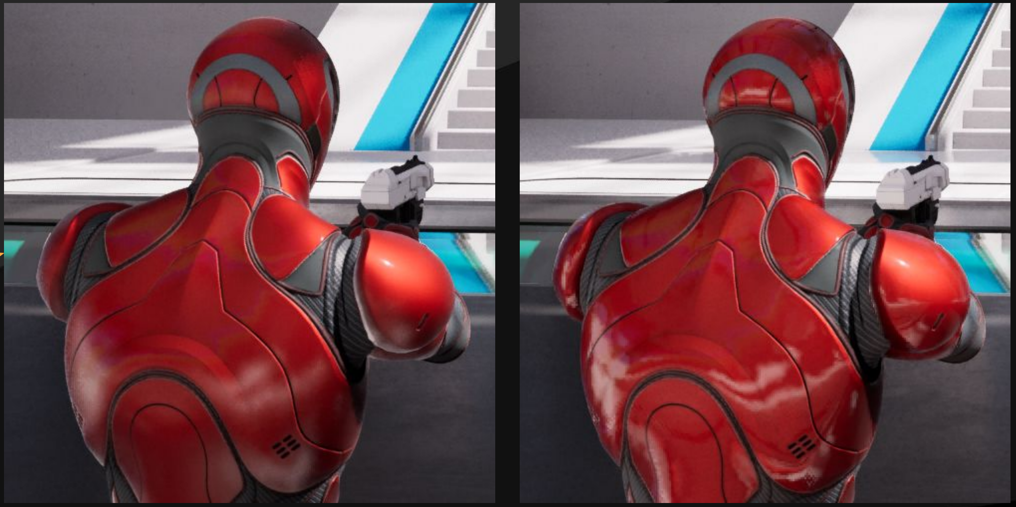
Epic

In “Lumen in the Land of Nanite”, there aren’t any smooth materials, so our reflection cost is very small. Our total Lumen cost is 2.8 milliseconds on High, which can easily fit in a 60 frames per second budget. On Epic quality, we’re doing four times more rays per pixel, which has a total cost of 4.6ms, and shows up as more accurate indirect shadows, and better temporal stability.

# “Lyra” - UE5 sample game

| SWRT, PS5 | | |
|---|---|---|
| | High (1⁄16 rpp) | Epic (¼ rpp) |
| Surface Cache | 0.36 ms | 0.66 ms |
| Final Gather | 1.65 ms | 3.30 ms |
| Reflections | 2.28 ms | 4.74 ms |
| Total | 4.29 ms | 8.70 ms |





High

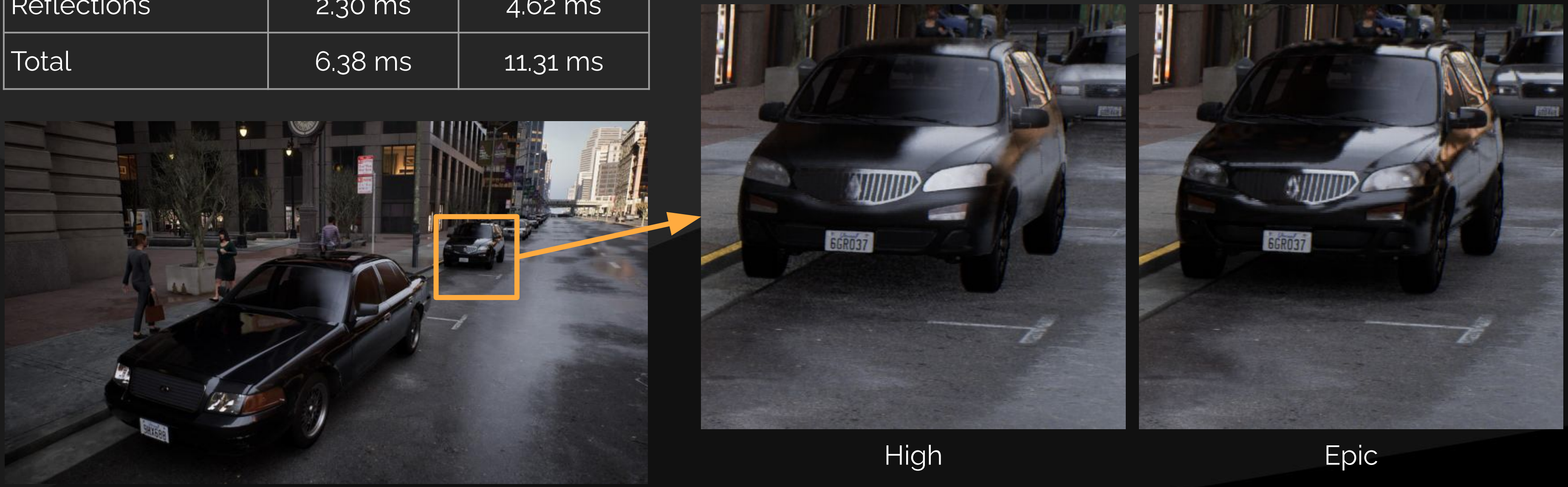
Epic

In our “Lyra” UE5 sample game, which is using Software Ray Tracing, our total Lumen cost is 4.3 milliseconds on High.  This scene looks simple, but with its clean diffuse textures and smooth materials, there’s a lot of work for realtime GI and reflections to do.

On Epic settings we have full resolution reflections, which you can see on the right.

# “The Matrix Awakens” - UE5 demo

| HWRT with Far Field, PS5 | | |
|---|---|---|
| | High (1⁄16 rpp) | Epic (¼ rpp) |
| Surface Cache | 1.03 ms | 1.17 ms |
| Final Gather | 3.05 ms | 5.52 ms |
| Reflections | 2.30 ms | 4.62 ms |
| Total | 6.38 ms | 11.31 ms |



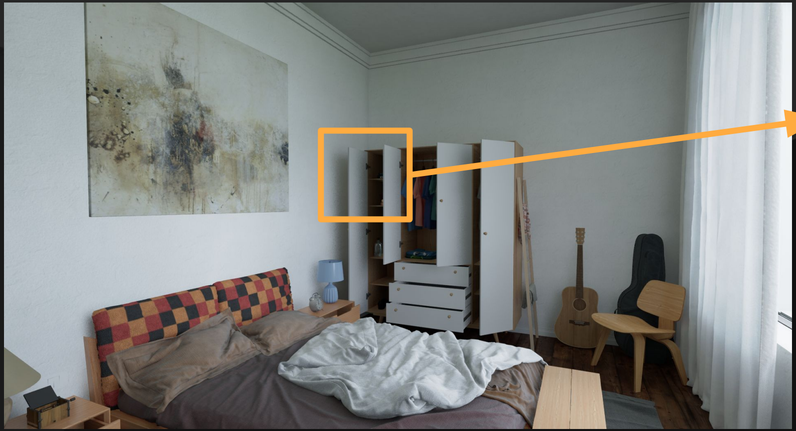
High

Epic

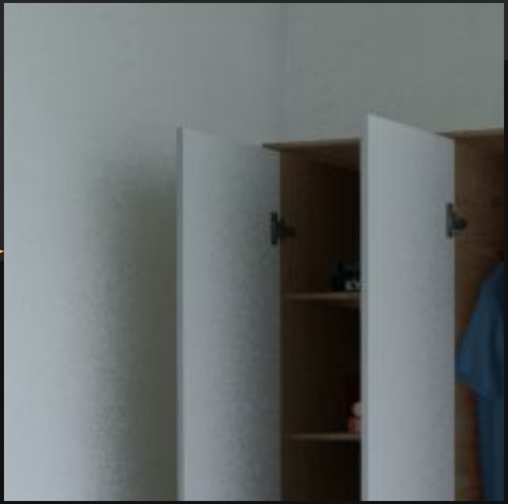
In “The Matrix Awakens”, we’re using Hardware Ray Tracing with Far Field, and it’s a much more complex scene so our tracing costs are higher.  On High settings the total Lumen cost is 6.4 milliseconds, while Epic has significantly higher quality at 11.3 milliseconds.

# “Lake House” - Architectural Visualization

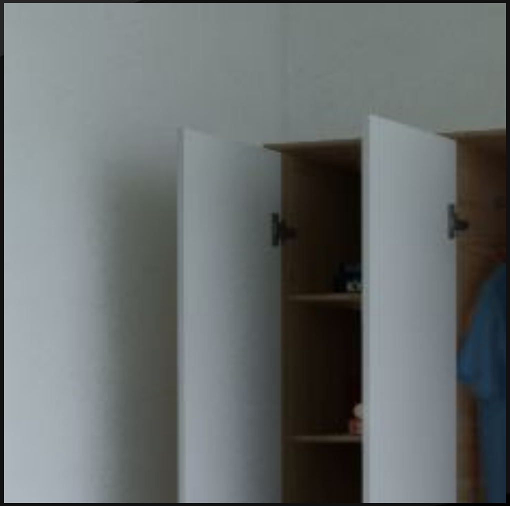
| HWRT on 2080 TI | | |
|---|---|---|
| | Epic (¼ rpp) | ArchVis (1rpp) |
| Surface Cache | 1.68 ms | 1.68 ms |
| Final Gather | 4.01 ms | 8.01 ms |
| Reflections | 1.56 ms | 1.56 ms |
| Total | 7.25 ms | 11.25 ms |



Scene courtesy of Rafael Reis @ UE4Arch

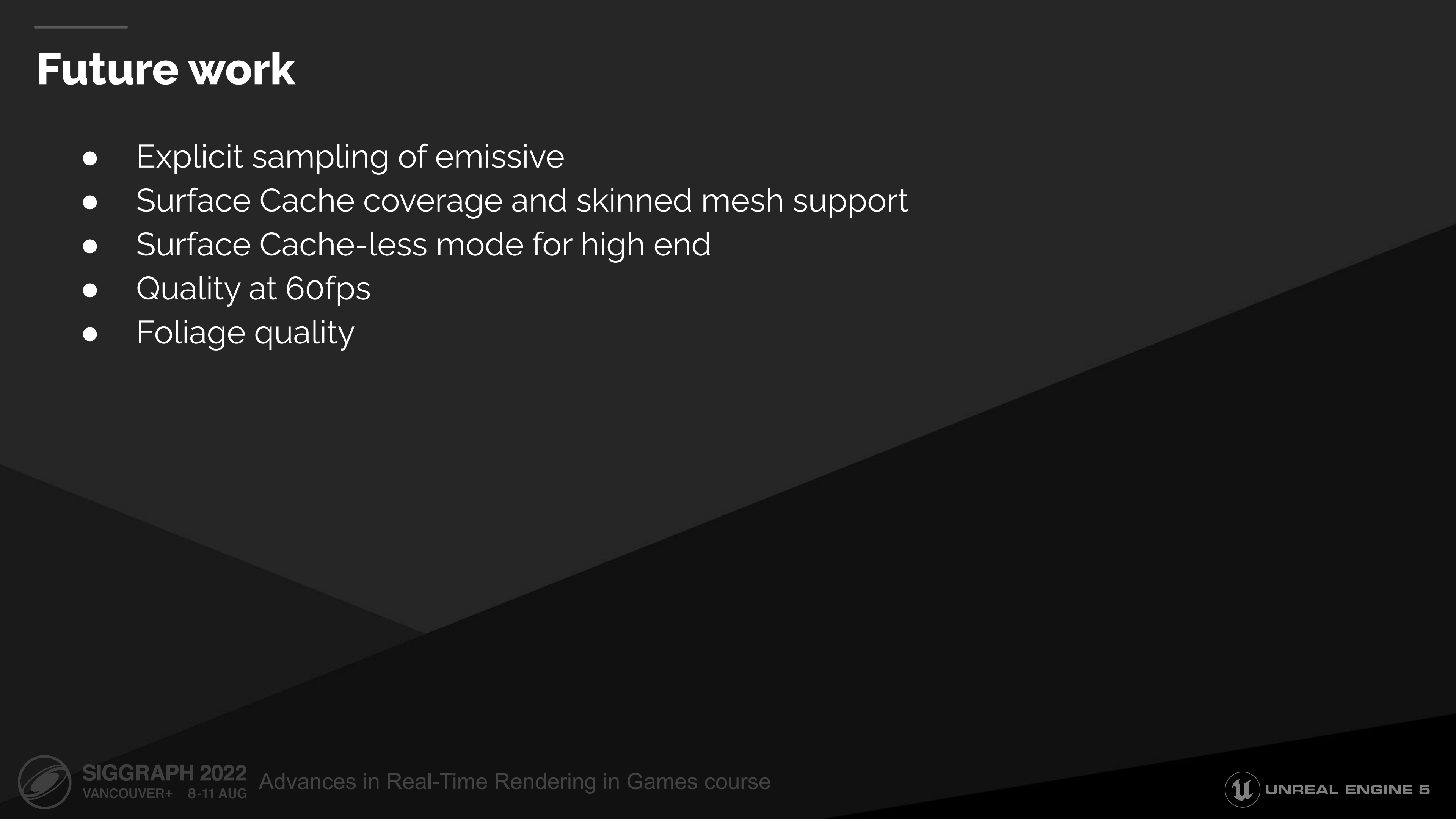


Epic (¼ rpp)



ArchVis (1rpp)

Now looking at an Architectural Visualization scene, running on a 2080 TI. On Epic settings Lumen costs 7.3 milliseconds, which has a little bit of noise visible when you zoom in on one area of the screen. We can crank up the Final Gather to 1 ray per pixel, and now we have extremely smooth indirect lighting.



## Future work

- Explicit sampling of emissive
- Surface Cache coverage and skinned mesh support
- Surface Cache-less mode for high end
- Quality at 60fps
- Foliage quality

For future work, we’d like to have explicit sampling of emissive meshes in our downsampled radiance cache.  We’re still working to improve the coverage of our Surface Cache, and to support skinned meshes.

We’d like to have a mode that doesn’t use the surface cache at all, which will probably be very expensive and only viable for high end.

We’re always working on the quality of these techniques at 60 frames per second, where the budget is tight, and we’re still working on foliage quality.

# Acknowledgements

Advances - Natalya Tatarchuk

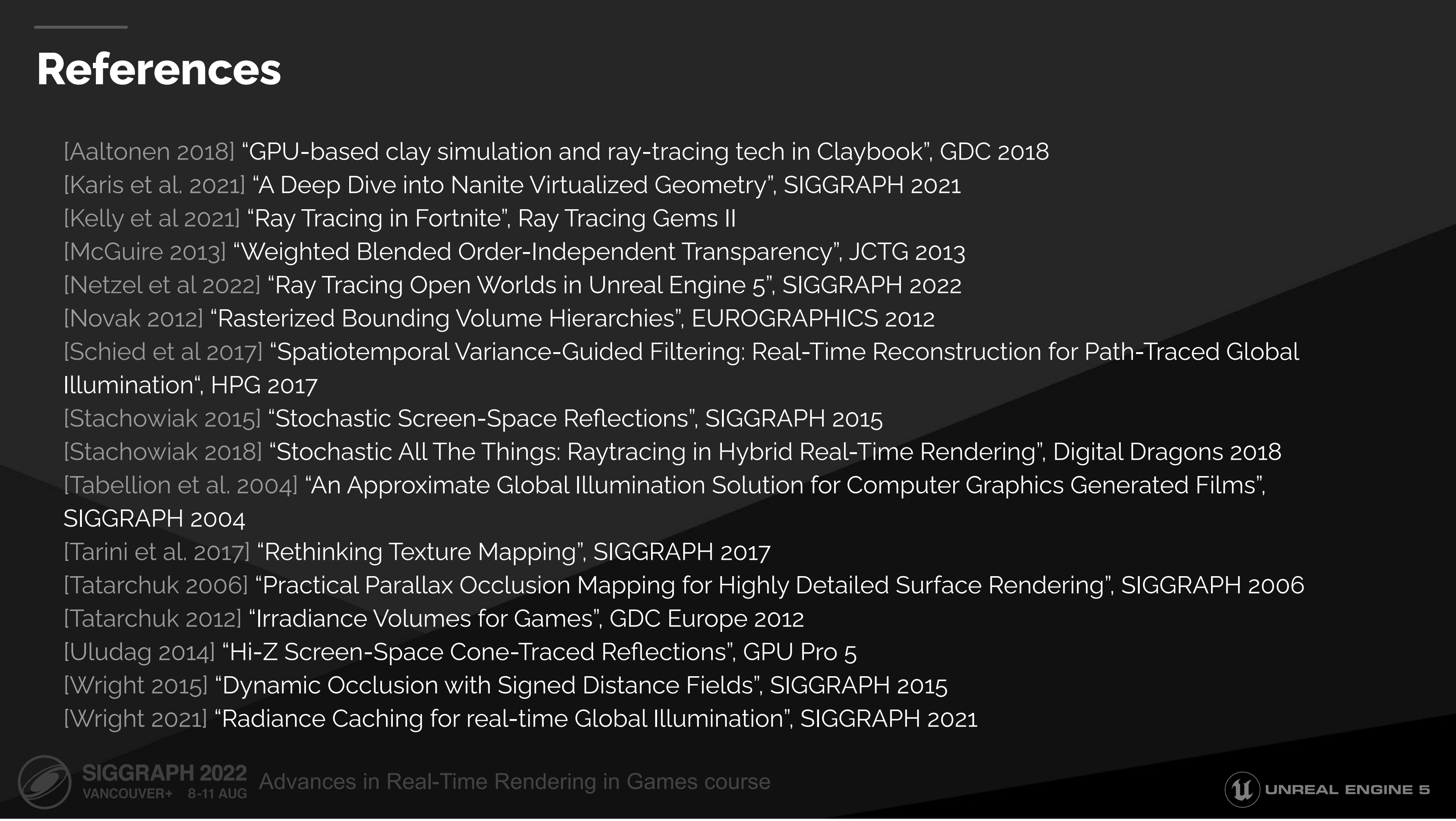
Entire UE Rendering team

- Hardware Ray Tracing
  - Yuriy O'Donnell
  - Juan Canada
  - Aleksander Netzel
  - Kenzo ter Elst
  - Yujiang Wang
  - Tiago Costa
  - Tiantian Xie
  - Ryan Vance

- Rendering
  - Zachary Bethel
  - Guillaume Abadie
  - Rune Stubbe
  - Brian Karis

- Special Projects
  - Jerome Platteaux
  - Scott Clifford
  - Pete Sumanaseni
  - Quentin Marmier
  - Votch Levi
  - Patrick Enfedaque
  - Sébastien Lussier

We'd like to thank the entire Unreal Engine Rendering Team, especially everyone that worked on Hardware Ray Tracing, and we'd like to thank the Special Projects team at Epic that makes such amazing tech demos.  And thank you to Natasha for the Advances course.



# References

[Aaltonen 2018] “GPU-based clay simulation and ray-tracing tech in Claybook”, GDC 2018

[Karis et al. 2021] “A Deep Dive into Nanite Virtualized Geometry”, SIGGRAPH 2021

[Kelly et al 2021] “Ray Tracing in Fortnite”, Ray Tracing Gems II

[McGuire 2013] “Weighted Blended Order-Independent Transparency”, JCTG 2013

[Netzel et al 2022] “Ray Tracing Open Worlds in Unreal Engine 5”, SIGGRAPH 2022

[Novak 2012] “Rasterized Bounding Volume Hierarchies”, EUROGRAPHICS 2012

[Schied et al 2017] “Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination“, HPG 2017

[Stachowiak 2015] “Stochastic Screen-Space Reflections”, SIGGRAPH 2015

[Stachowiak 2018] “Stochastic All The Things: Raytracing in Hybrid Real-Time Rendering”, Digital Dragons 2018

[Tabellion et al. 2004] “An Approximate Global Illumination Solution for Computer Graphics Generated Films”, SIGGRAPH 2004

[Tarini et al. 2017] “Rethinking Texture Mapping”, SIGGRAPH 2017

[Tatarchuk 2006] “Practical Parallax Occlusion Mapping for Highly Detailed Surface Rendering”, SIGGRAPH 2006

[Tatarchuk 2012] “Irradiance Volumes for Games”, GDC Europe 2012

[Uludag 2014] “Hi-Z Screen-Space Cone-Traced Reflections”, GPU Pro 5

[Wright 2015] “Dynamic Occlusion with Signed Distance Fields”, SIGGRAPH 2015

[Wright 2021] “Radiance Caching for real-time Global Illumination”, SIGGRAPH 2021

Here are the references from this talk, and thank you very much for joining us.