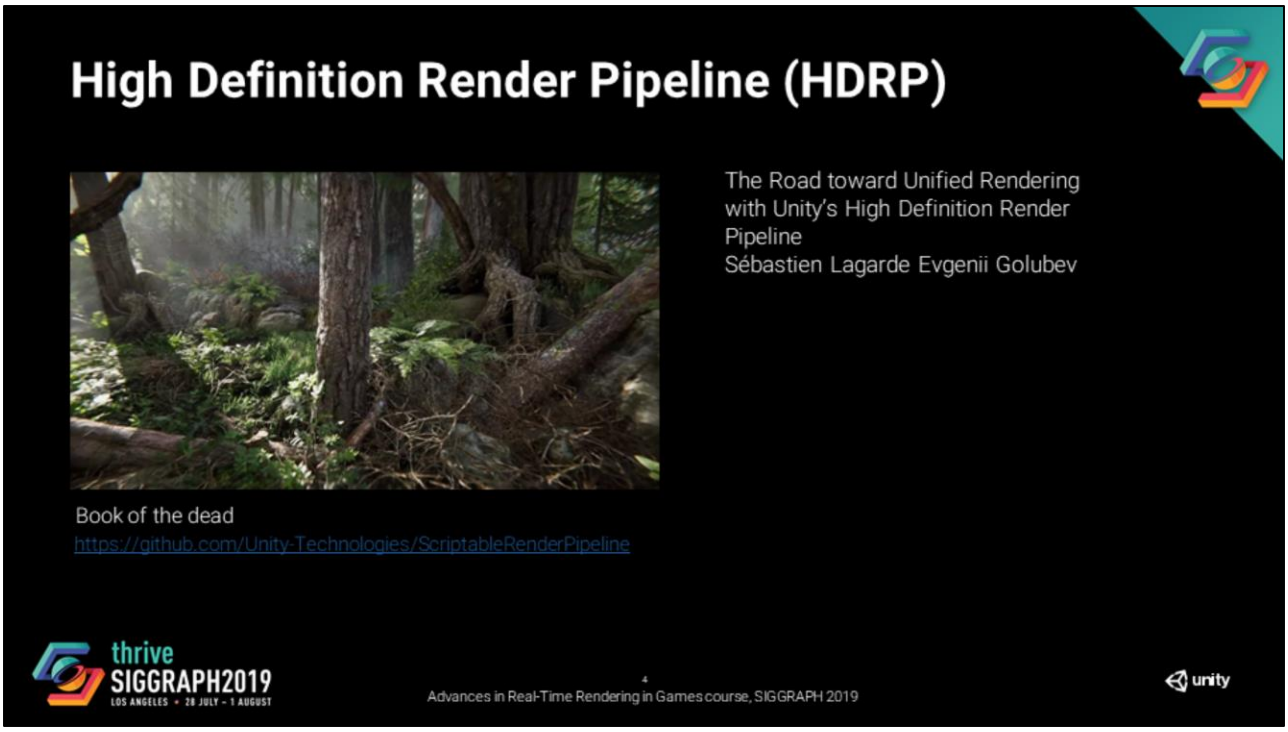

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST
unity
2019


The goal of this presentation is to share with you the lessons that we learned while trying to take advantage of ray tracing hardware acceleration in the Unity game engine.

The slides with speaker notes will be available after the presentation if you want to look more closely at something (you are reading them)!

# PHOTOGRAPHY & RECORDING ENCOURAGED

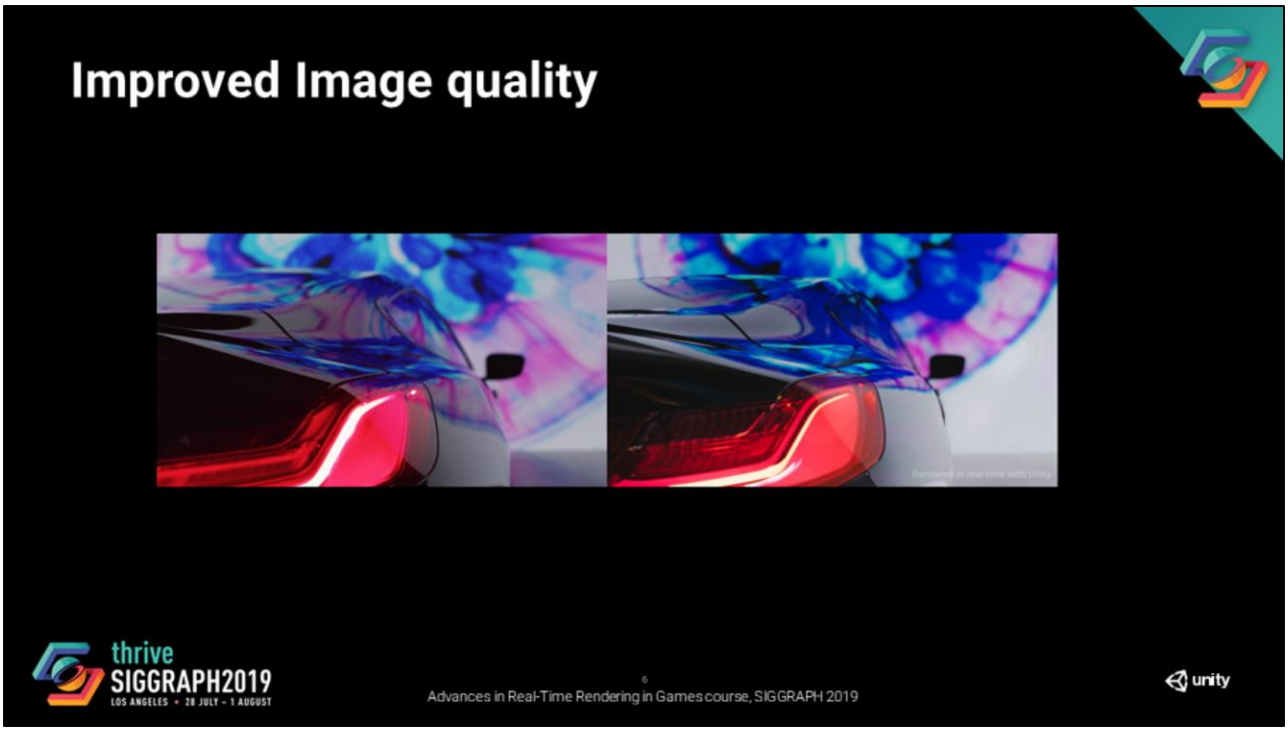



You can see this talk as a sequel to the talk « The Road toward Unified Rendering with Unity's High Definition Render Pipeline" presented last year in the same track.

If you are interested in additional details about the render pipeline in which the work presented today was done. Feel free to check our github repository or the slides of the previous talk.

https://github.com/Unity-Technologies/ScriptableRenderPipeline
http://advances.realtimerendering.com/s2018/index.htm

That’s said, let’s look at a video.




This video was part of the « Reality vs illusion » project.

Some of the shots in the video are real-world footage, others are rendered using our real-time renderer. The VFX breakdown can be found at: https://blogs.unity3d.com/2019/04/11/reality-vs-illusion/ if you want to know exactly what is going on during the video.

In collaboration with Light & Shadow, the goal for us was to explore and share what real-time ray tracing brings to a game engine-based production.

While I believe that benefits of real-time ray tracing are sometimes questionable. For this demo, the quality improvements that we were able to achieve using real time ray tracing are clear.

## Improved Image quality

Left: real-word footage. Right rendered with Unity



The results of the project were the video that I showed then and a real-time demo that I will show at the end of the presentation

In our journey of exploring real-time ray tracing, we have implemented a good number of effects.

## Ray Tracing Effects

Ambient Occlusion



The first one is ray traced ambient occlusion. From our tests, it is always an improvement when compared to techniques like screen space techniques, but at a non-negligible cost.

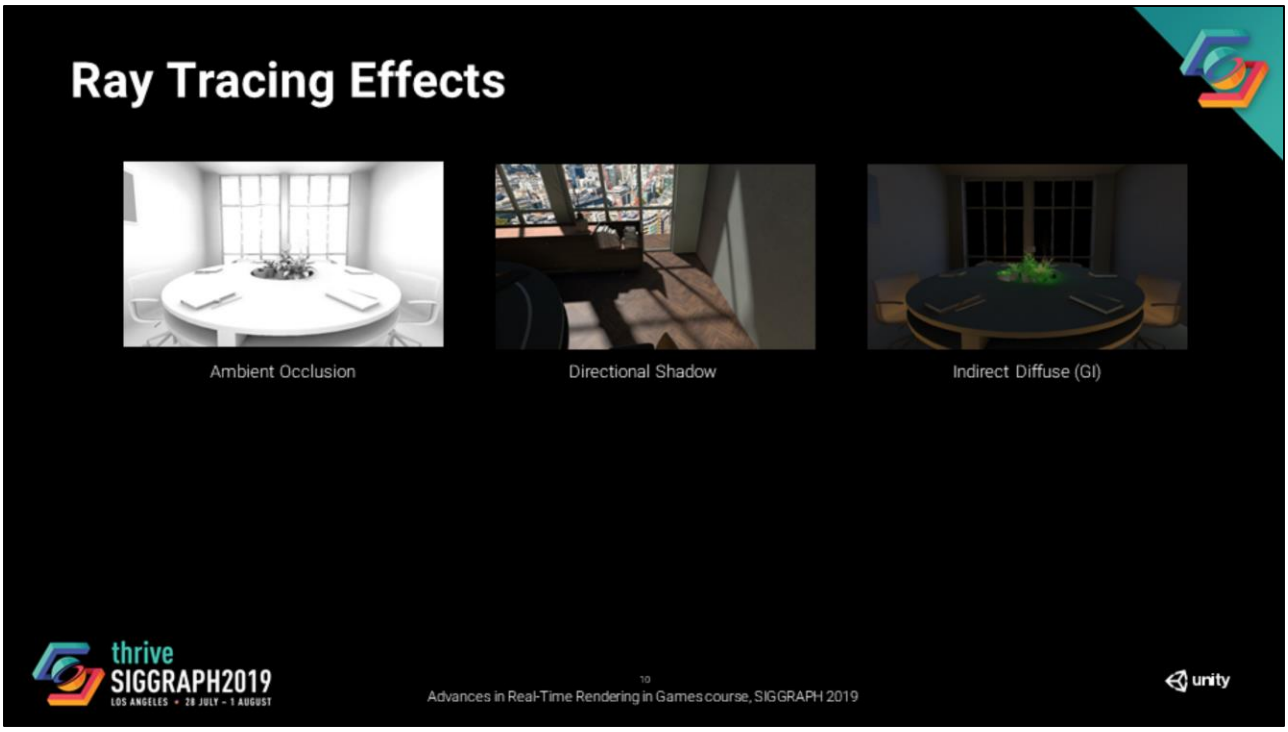
## Ray Tracing Effects

Ambient Occlusion

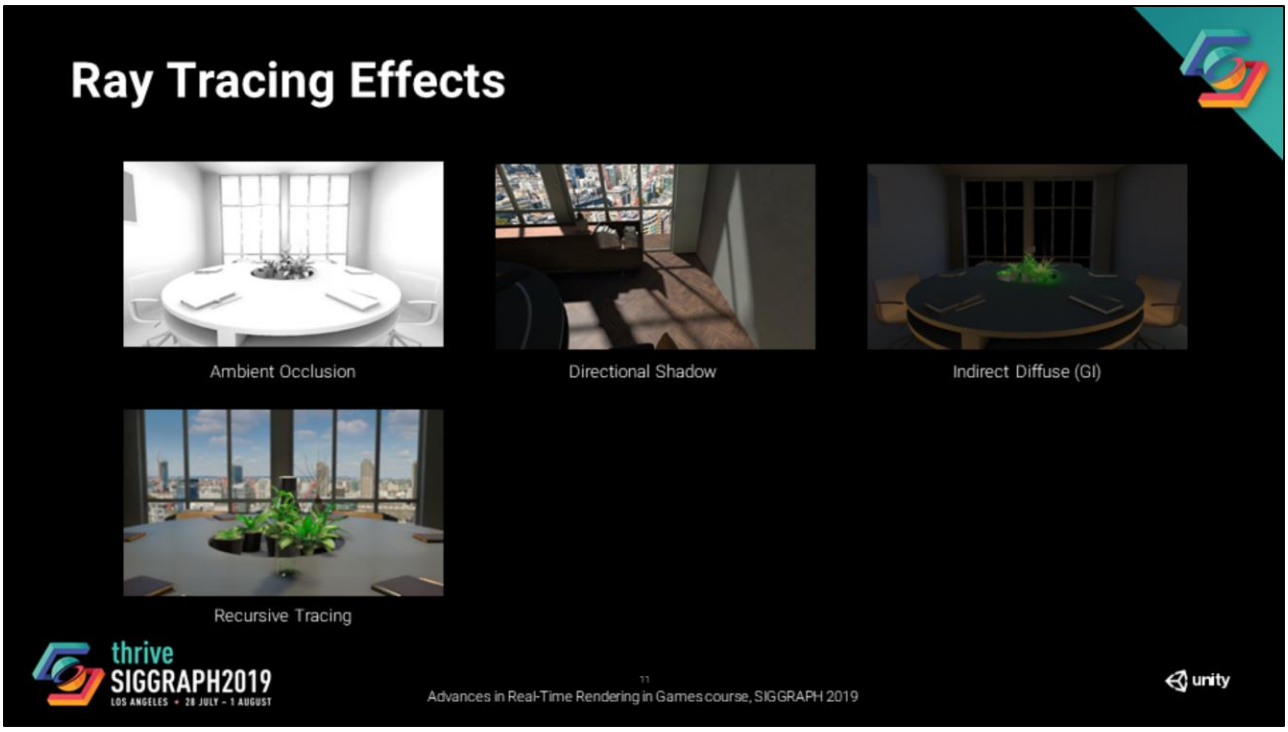
Directional Shadow



The next effect is soft directional shadows. It evaluates the penumbra of an infinite disk light and it avoids the artifacts of shadow map filtering techniques (which is always nice).




The next one is indirect diffuse or, more commonly in games, GI. It is an alternative to light probes or lightmaps which require baking. It can replace, or be combined with SSGI.

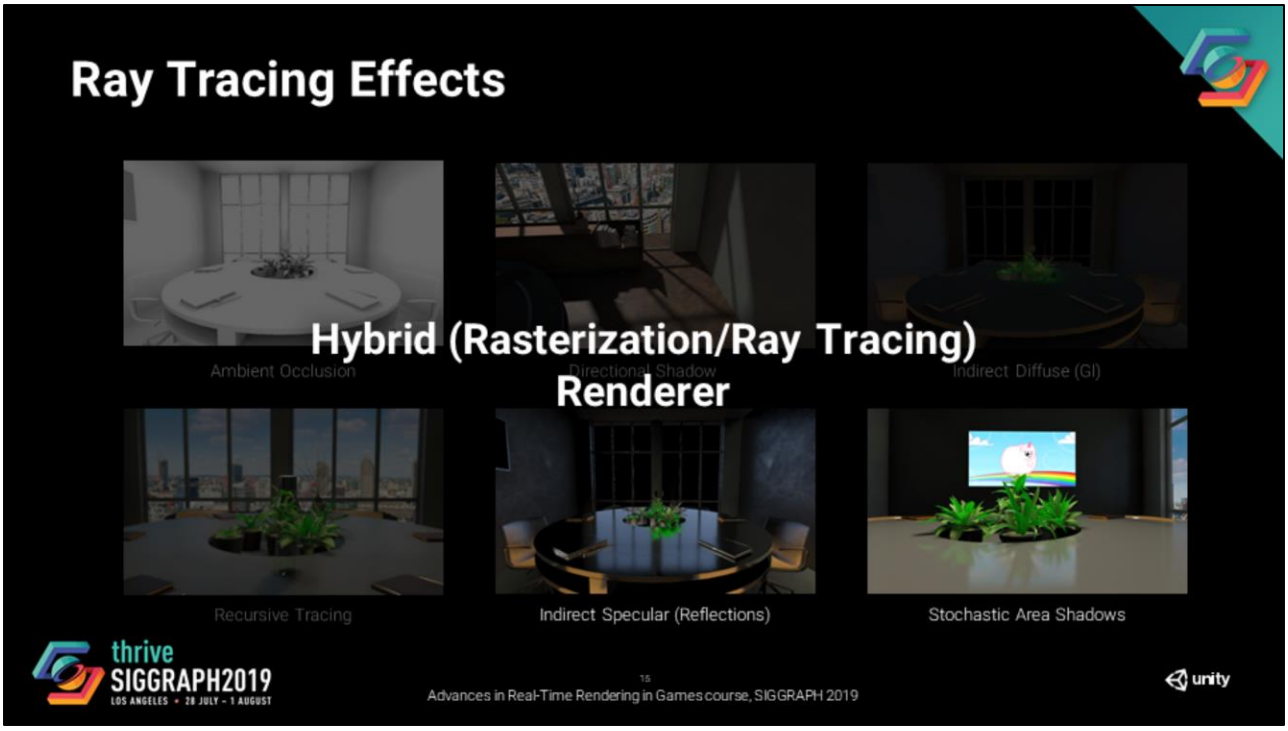



The next effect is what we call recursive tracing, we use it to render complex transparent light paths and that is how we can properly render the head and tail lights in the car demo.

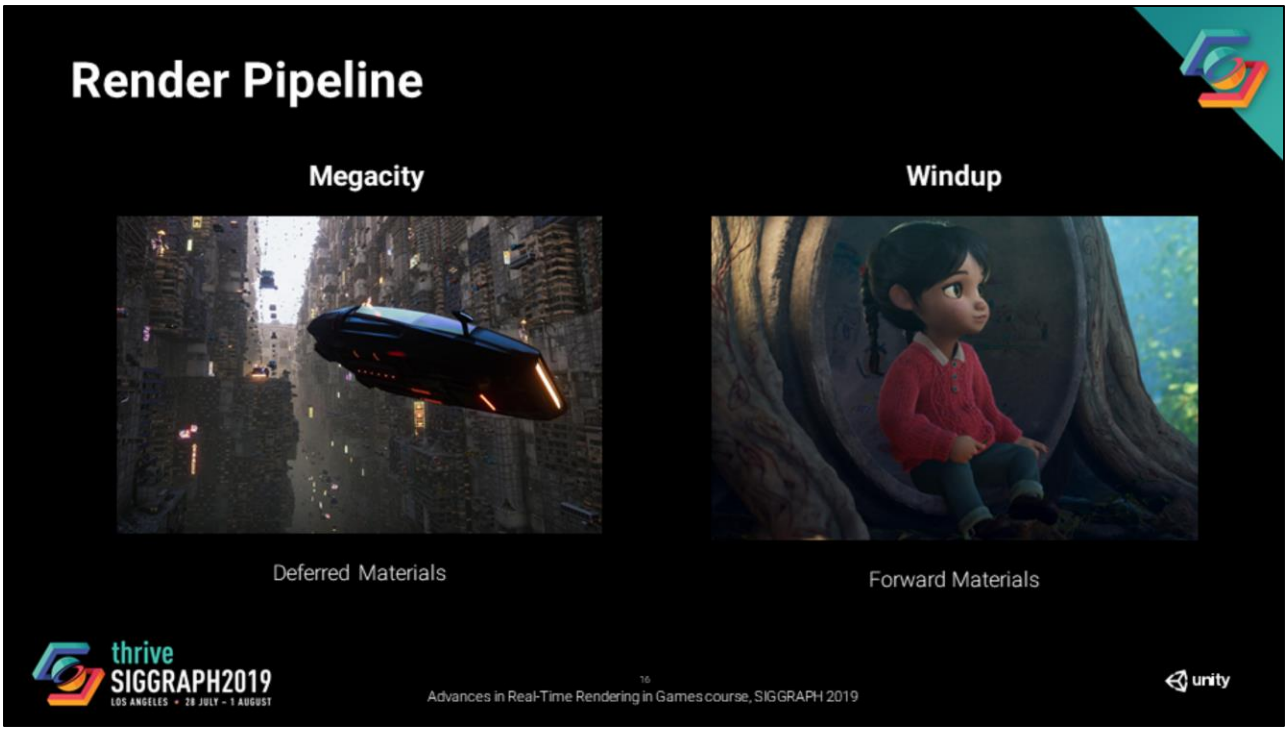

The next one is indirect specular, we also call it more commonly reflections in games.


And finally stochastic area shadows.


In this presentation, I'll be mainly talking about the later two effects. They are the most interesting ones and the points that I'll be making are also valid for the other effects.



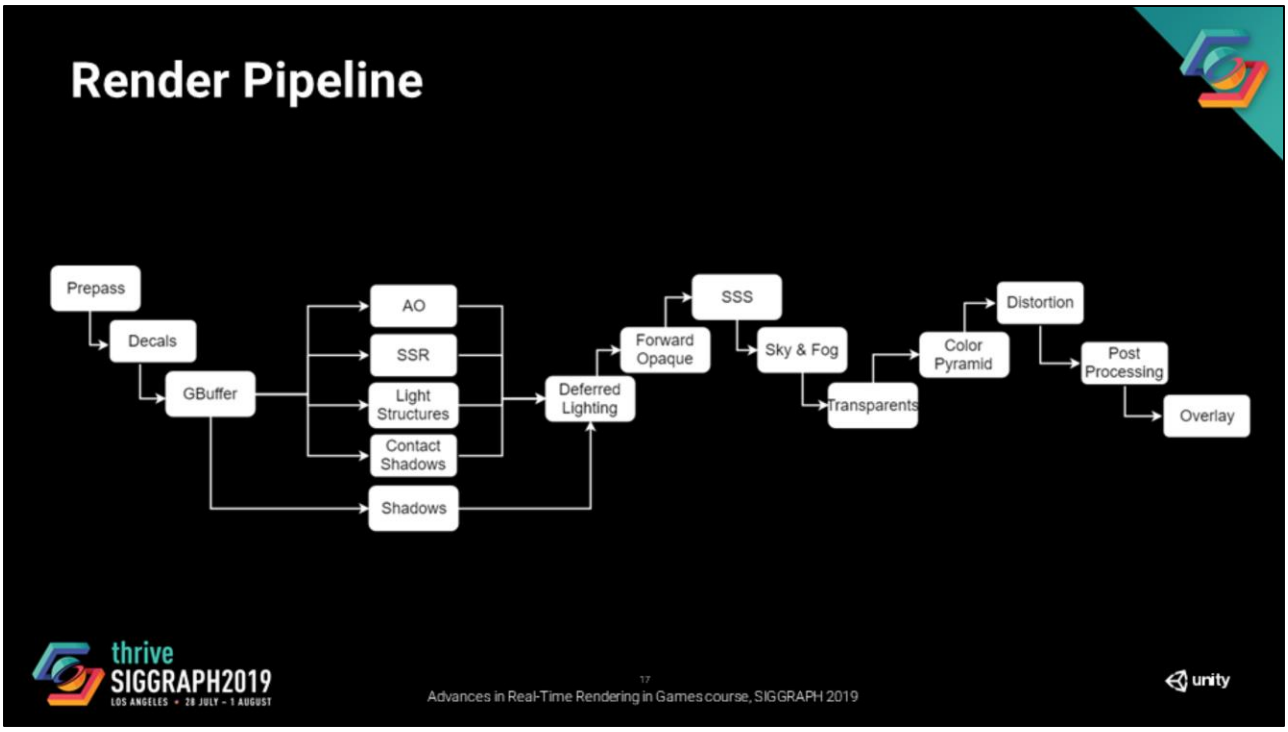

One more important thing to mention at this point is that we are going for a hybrid render pipeline. By that, I mean that we use rasterization for most of the render pipeline and we use ray tracing for specific effects. To compose the final image, we combine the result from rasterization and ray tracing.




Before we dive into the details of the changes that ray tracing brings to the render pipeline, it's important to understand what it's built on.

In our case, we start from a render pipeline that is a hybrid of deferred and forward. By that I mean that it has two types of materials for opaque objects: deferred materials (for most of the objects in a scene) and forward materials.

The forward materials are complex and do not fit into a gbuffer due to their large number of parameters (for instance hair, fabric, and eyes shaders in the image on the right). This is important to keep in mind for the rest of the presentation.




This a simplified version of the render pipeline.

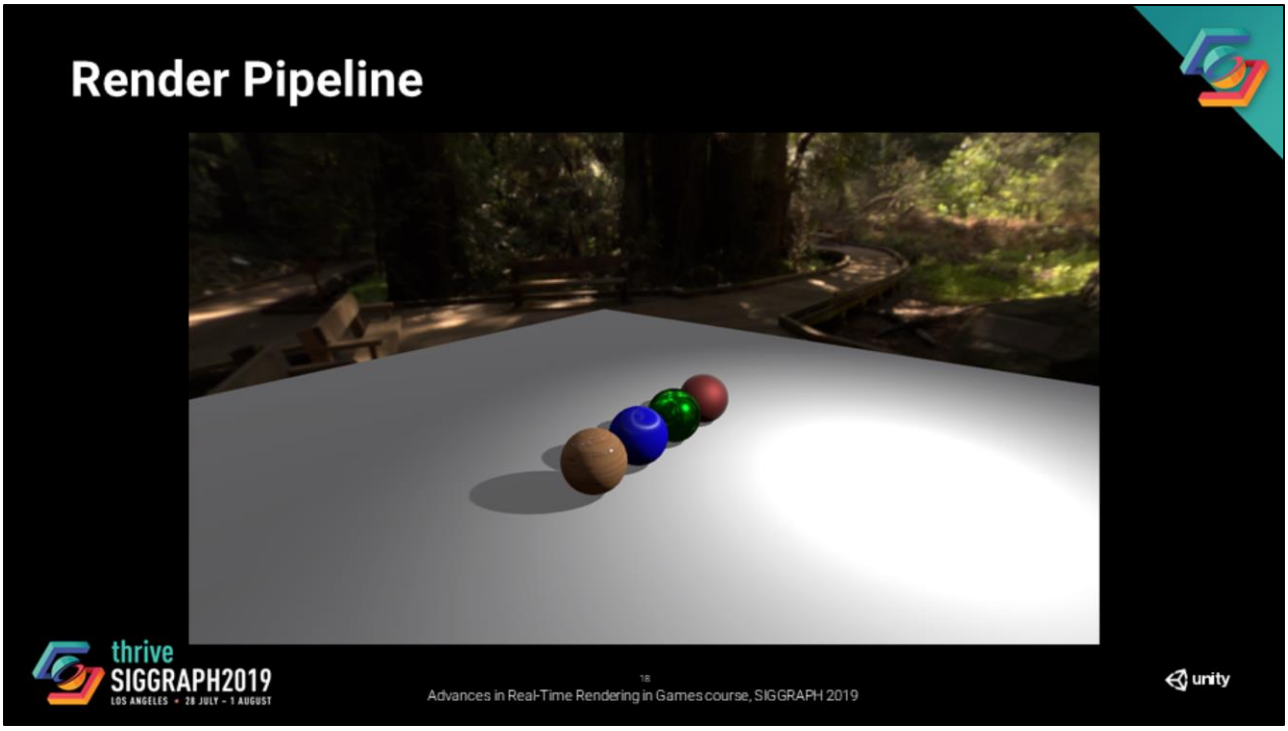
It starts with an optional prepass, then we have our decal buffer pass. We then render the gbuffer for our main lighting model (Lit).

Screen space lighting effects (SSAO, SSR, Contact Shadows) and light structures are evaluated, and shadow maps are rasterized.

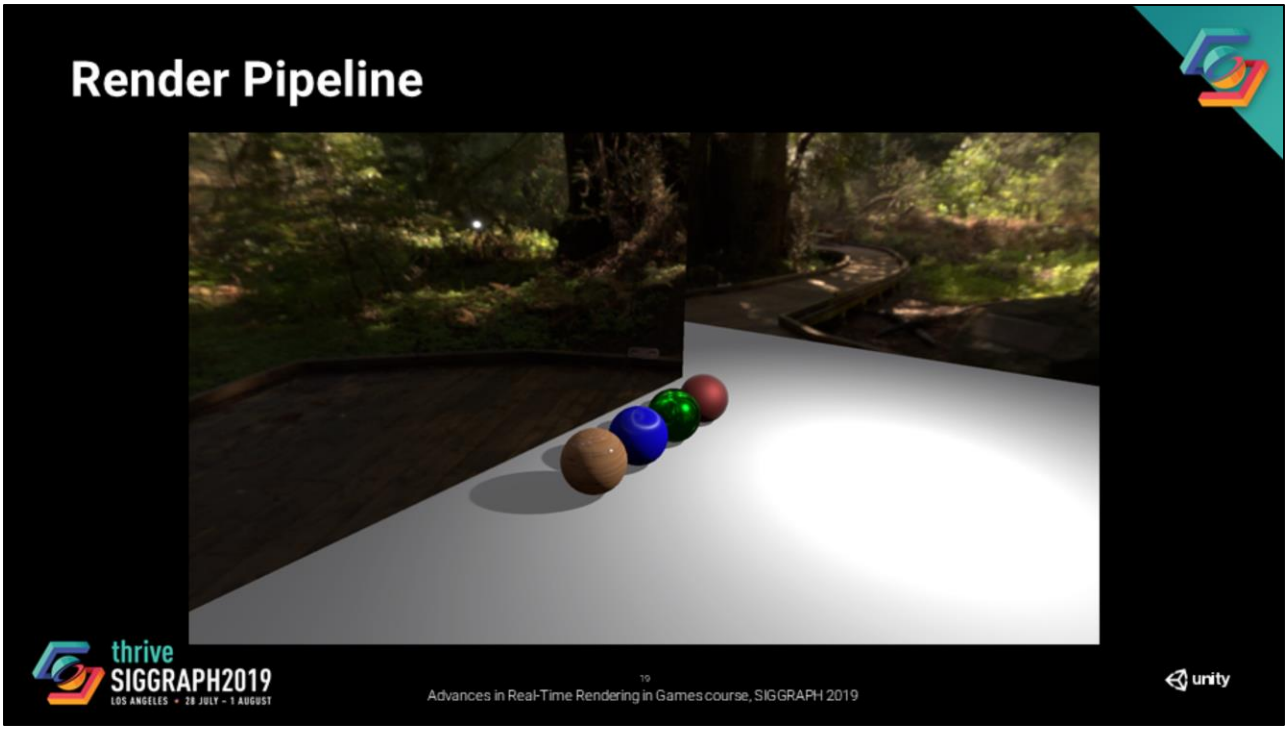
When all of that is done, we compute the lighting for the main lighting model (in the deferred pass), followed by the lighting of the other models (fabric, hair, eye, etc).

We evaluate and combine the separable subsurface scattering. Then evaluate the sky and the volumetric scattering.

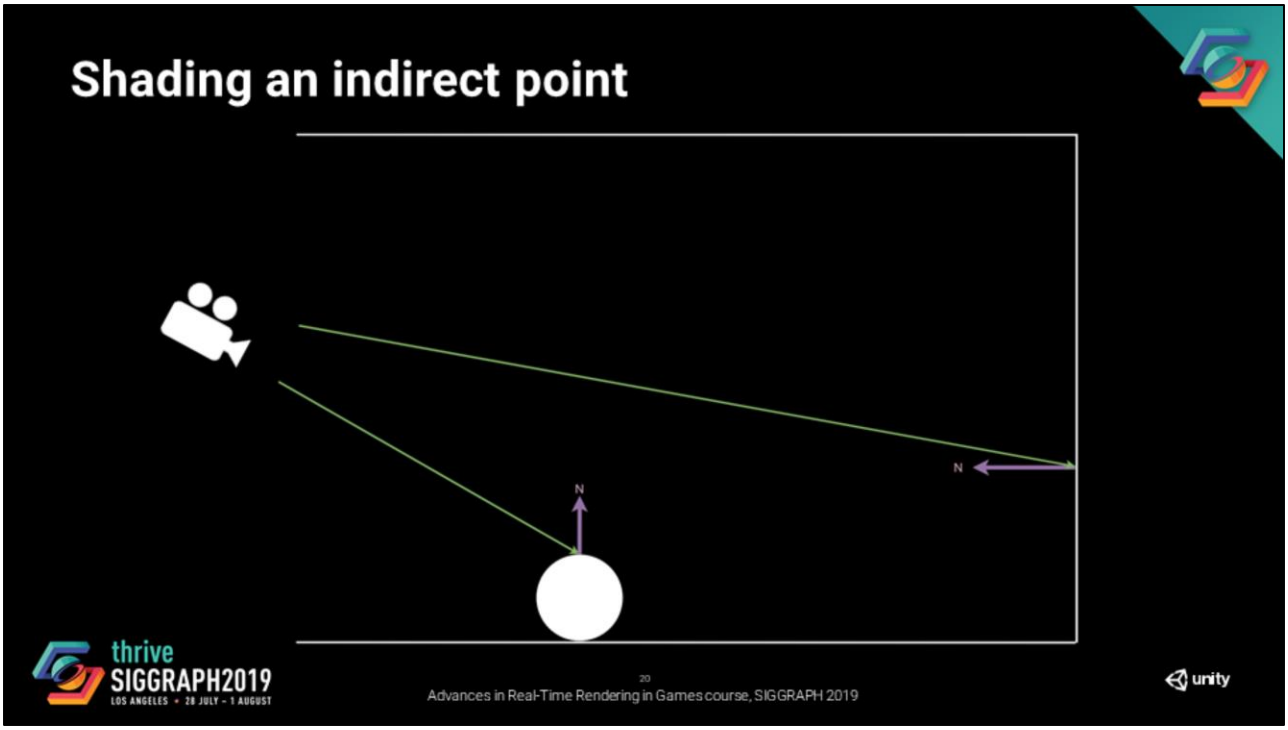
The transparent pipeline is executed after, then we evaluate the color pyramid and we finish by distortion, post processing, and finally overlay.



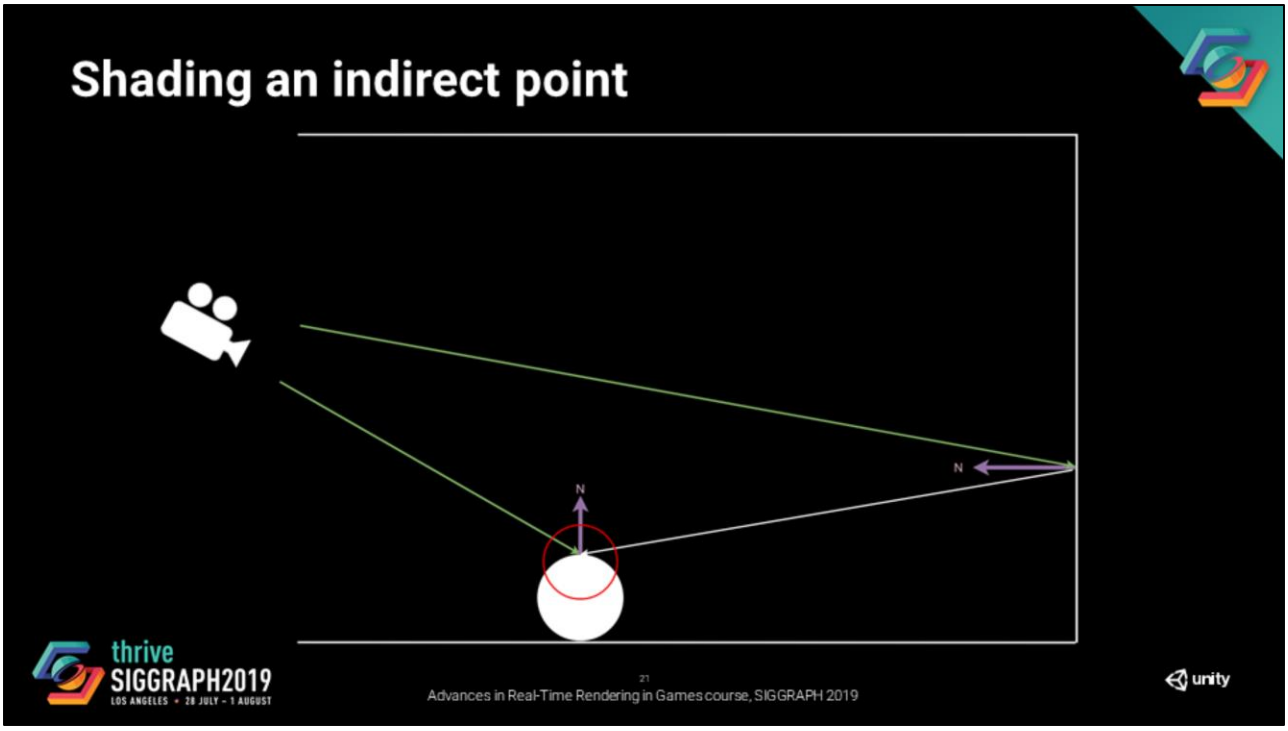

With the pipeline that we just described, we are able to render complex materials (here are a few variants of our lit model (left to right: clear coat, anisotropy, specular color, iridescence).



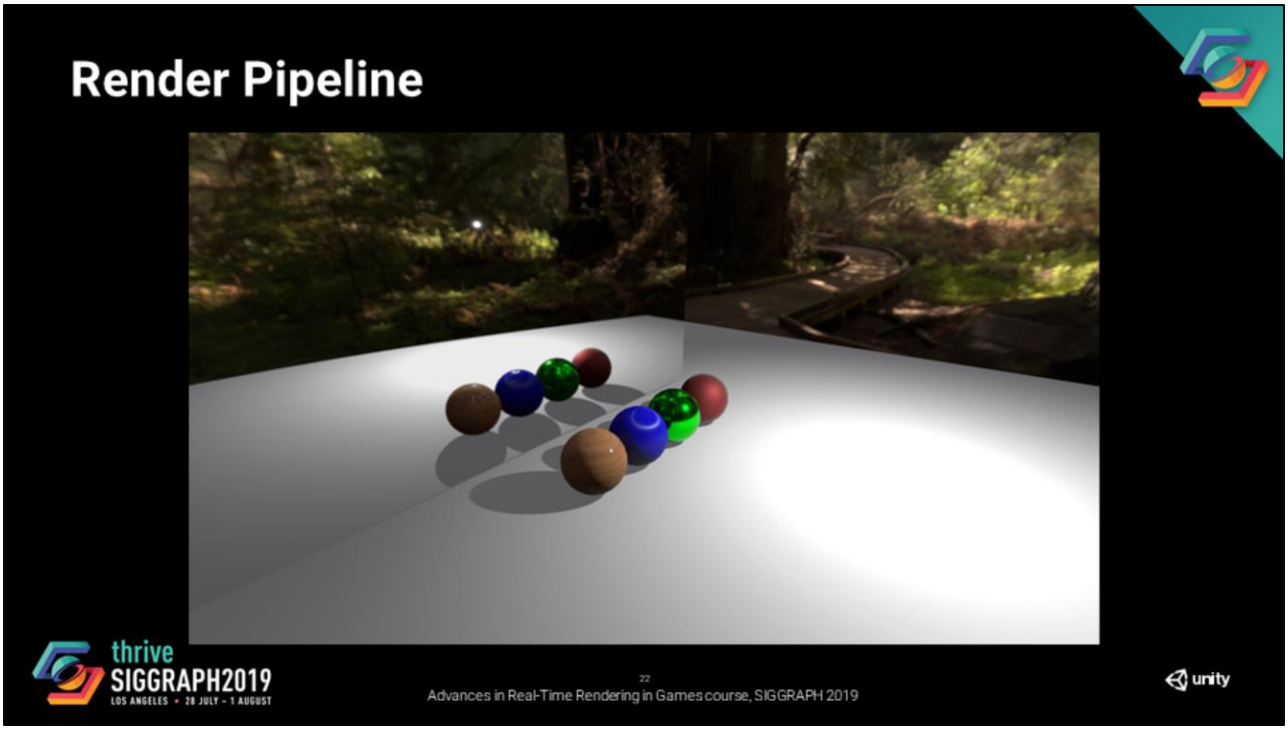

But what happens if we add a mirror for instance?




These are two points seen by the camera



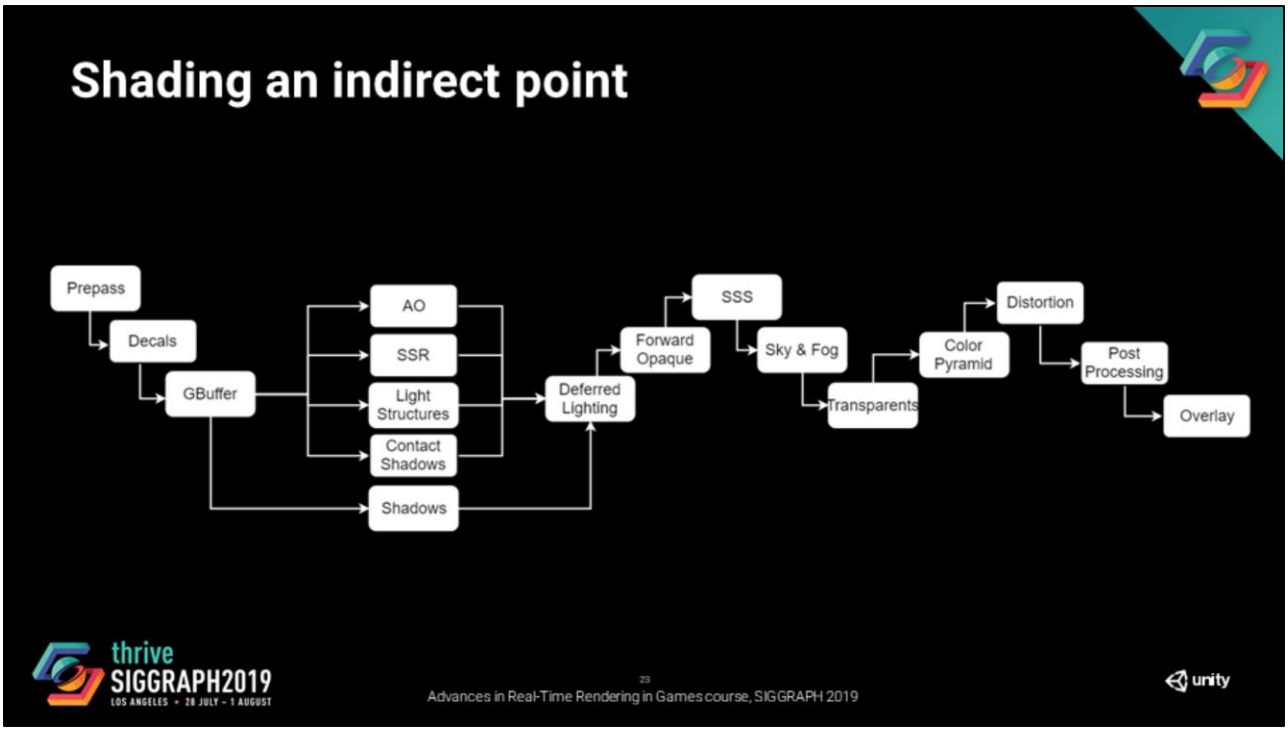

And we want to achieve the same lighting evaluation at the point in red for both the rasterized and the ray traced pipelines.



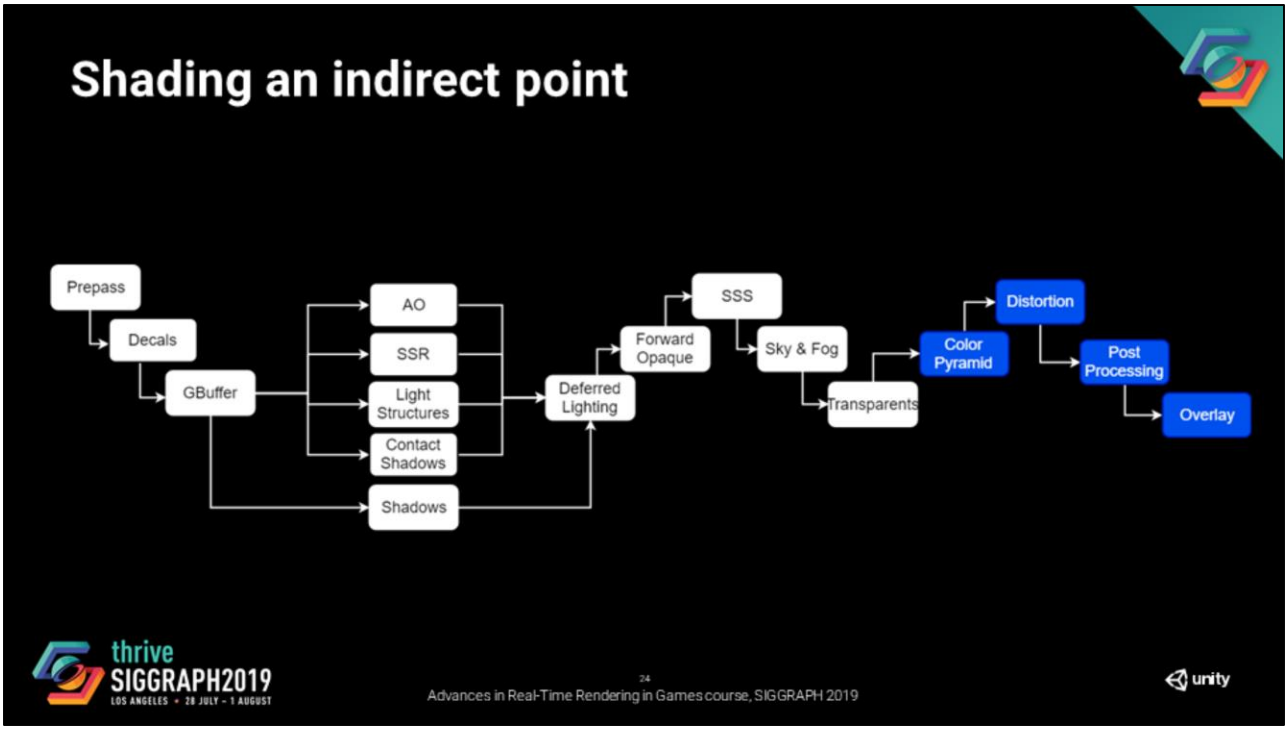

By rendering them the same way, we would get something like this.

A very interesting part of these new ray tracing APIs, is the ability to share the shader code between rasterization and ray tracing shaders.

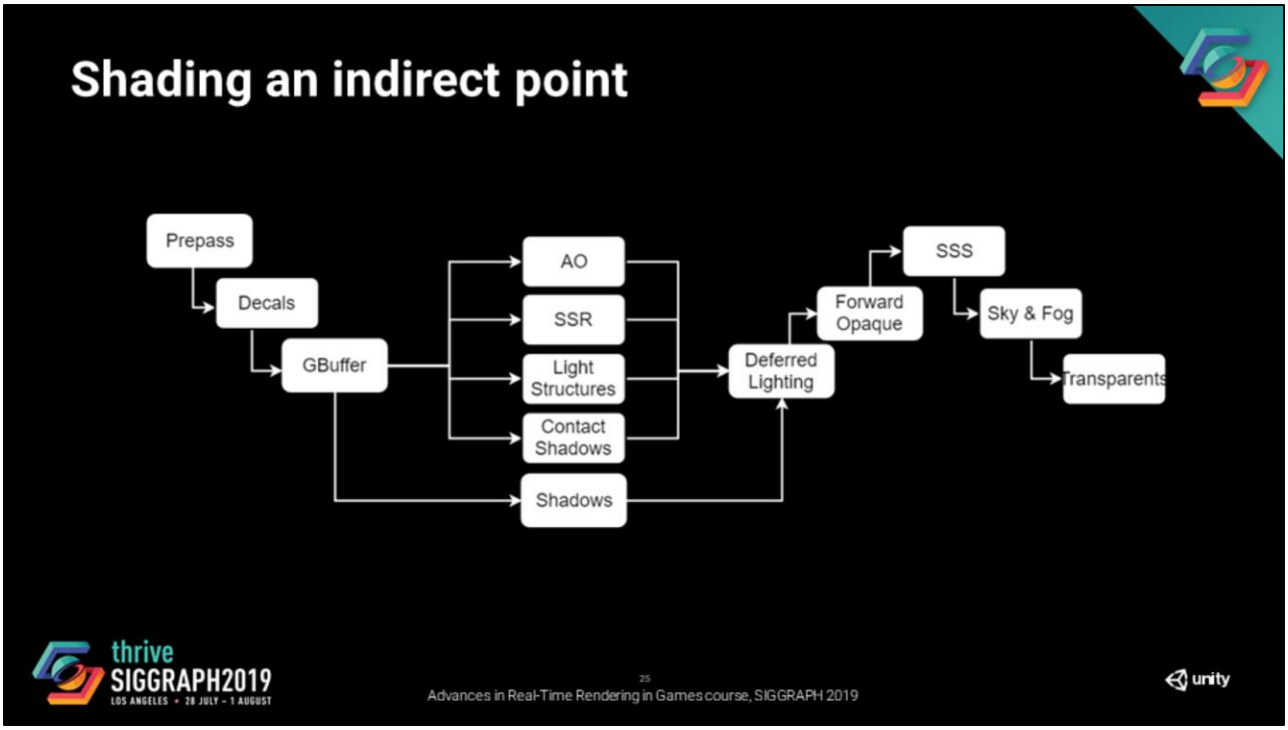
What does that mean for the lighting code for instance?



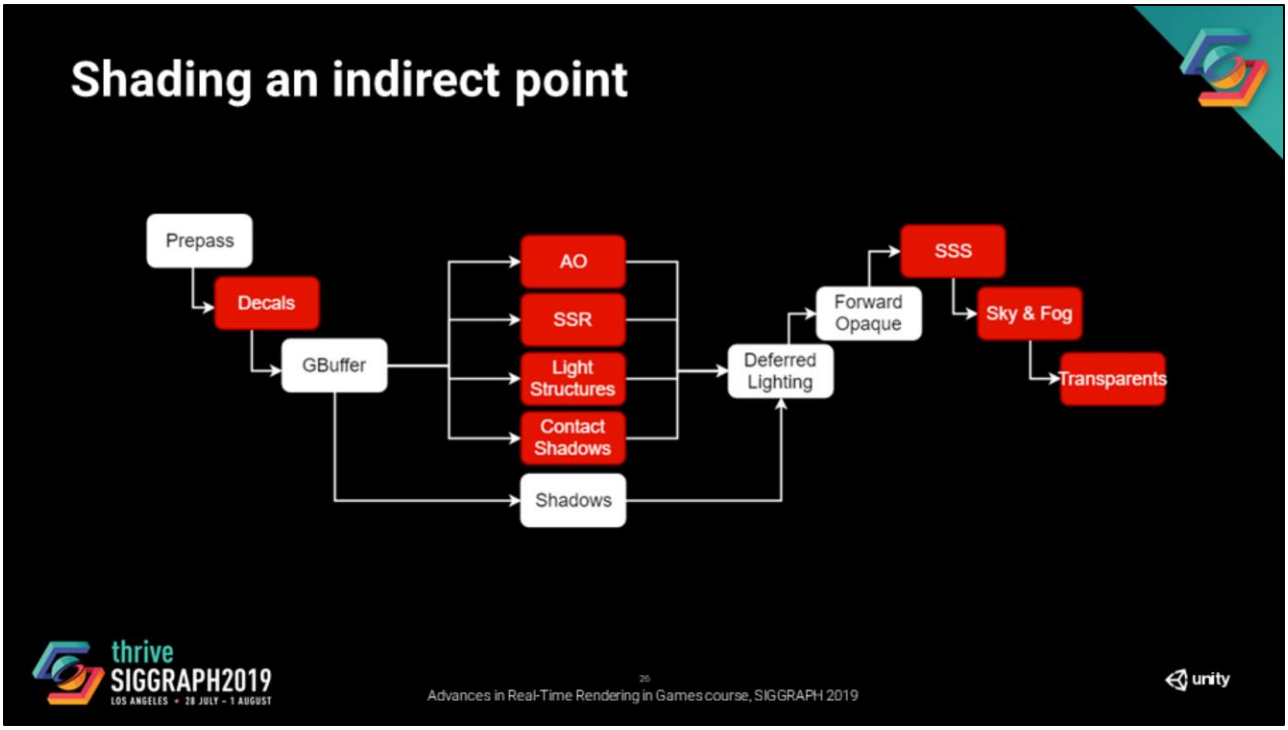

It means that to a certain extent all the information that is used to evaluate the lighting for rasterization should be available in some form to shade an indirect point with ray tracing.




First, this part of the pipeline is evaluated on the final image. So it does not really matter for our exercise, we can ignore them.




We are thus left with this. Let’s color in red everything that is screen space information.




That is pretty much the whole pipeline.

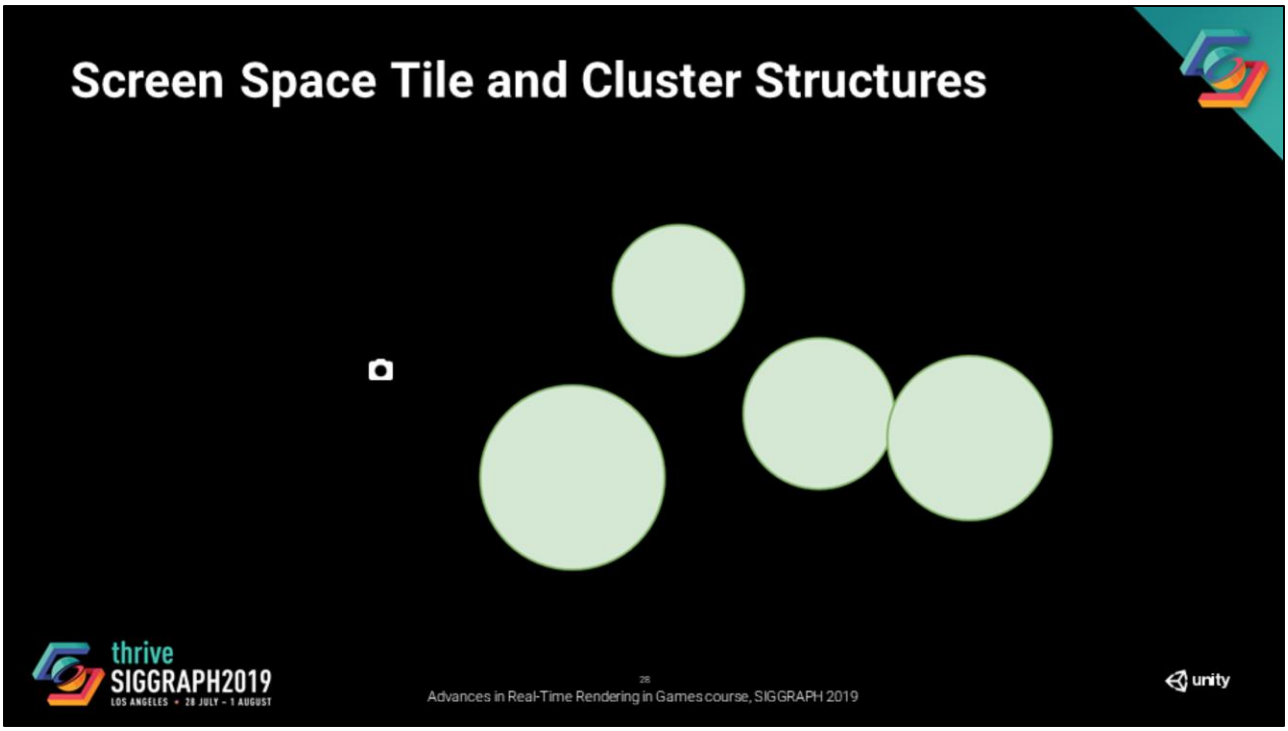
These are problems that we need to solve in order to achieve equivalent quality lighting at indirect points with the same shader code.

## Light Structures

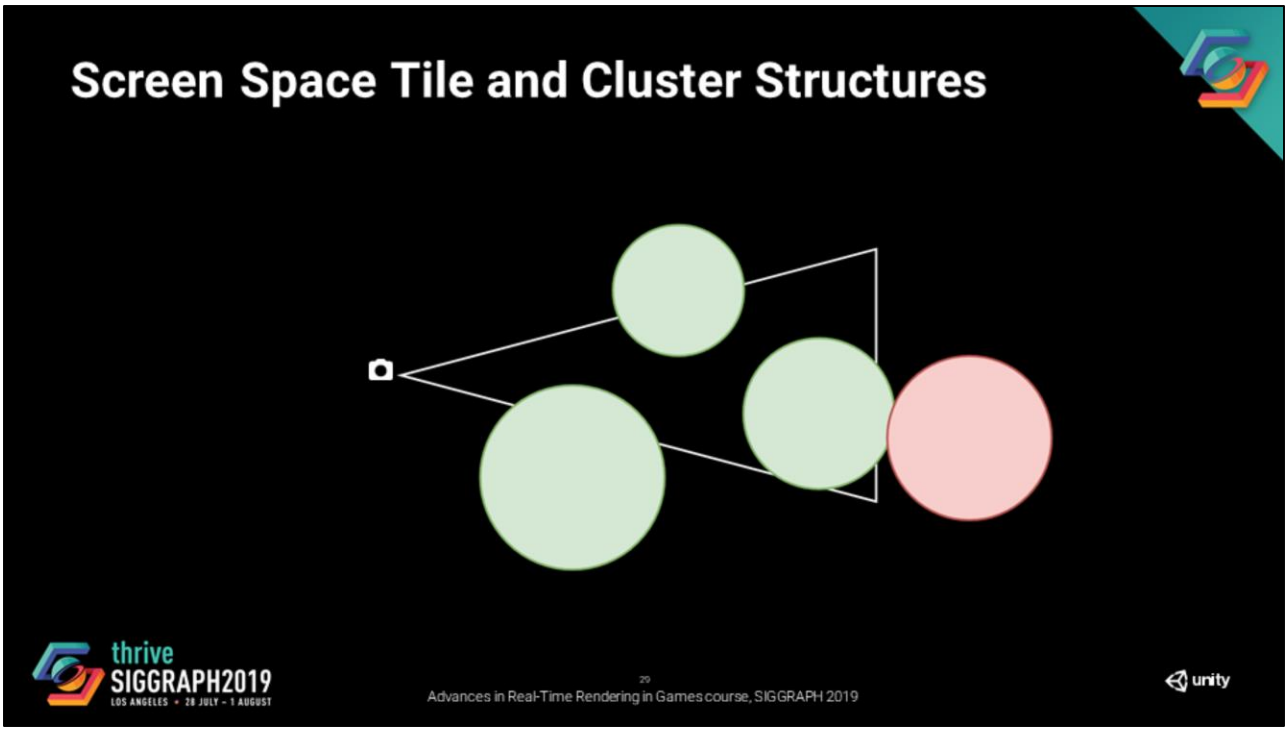



Let’s look what we can do for the light structures.

Obviously, you need them when your scene looks like this.

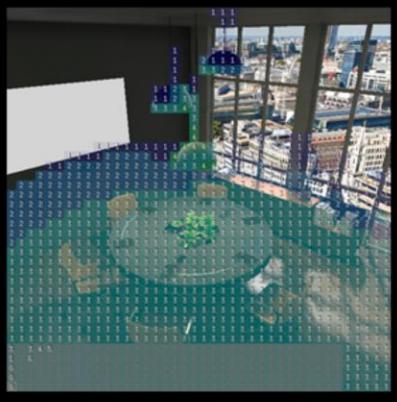



For rasterization, we usually start from a camera and a set of light volumes.

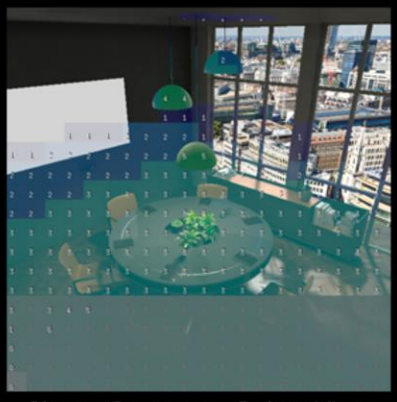



We cull those volumes using the camera frustrum

## Screen Space Tile and Cluster Structures



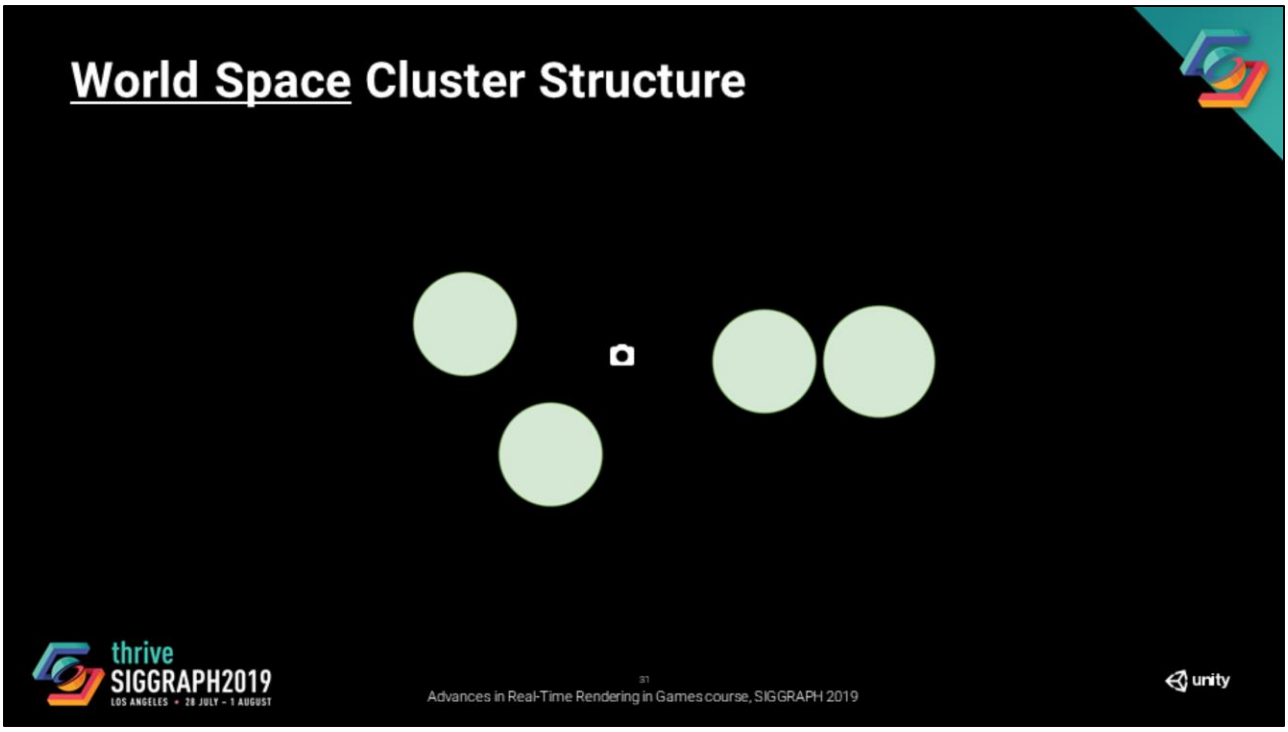
Tile Structure – Debug View



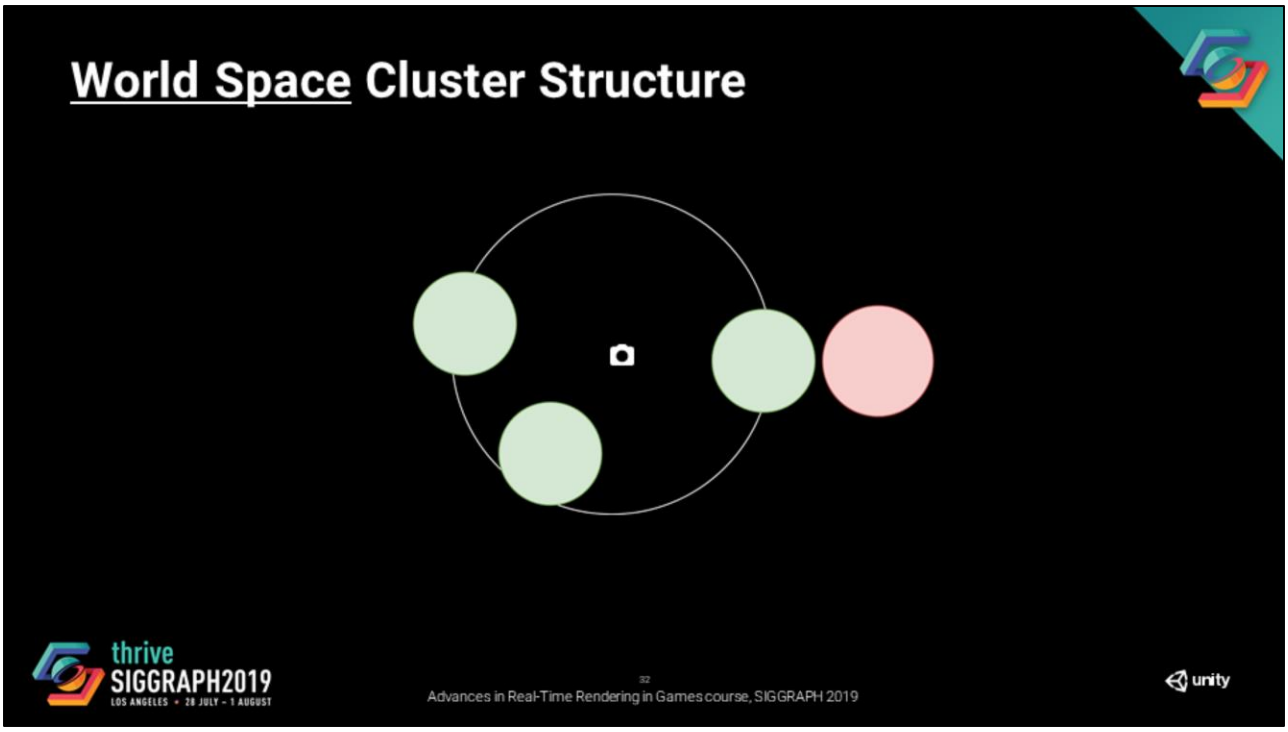
Cluster Structure – Debug View



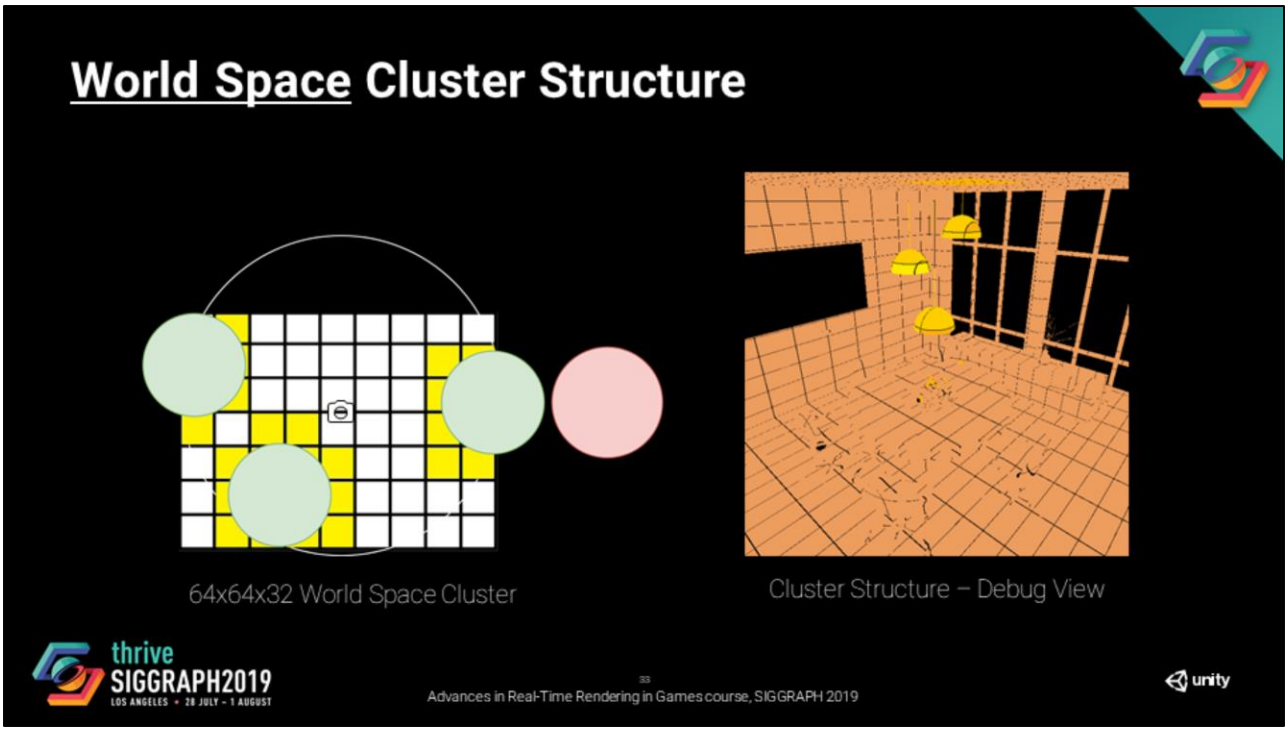
Then we evaluate the tiled and cluster structures. These are host light lists that we use to light the screen space fragments.



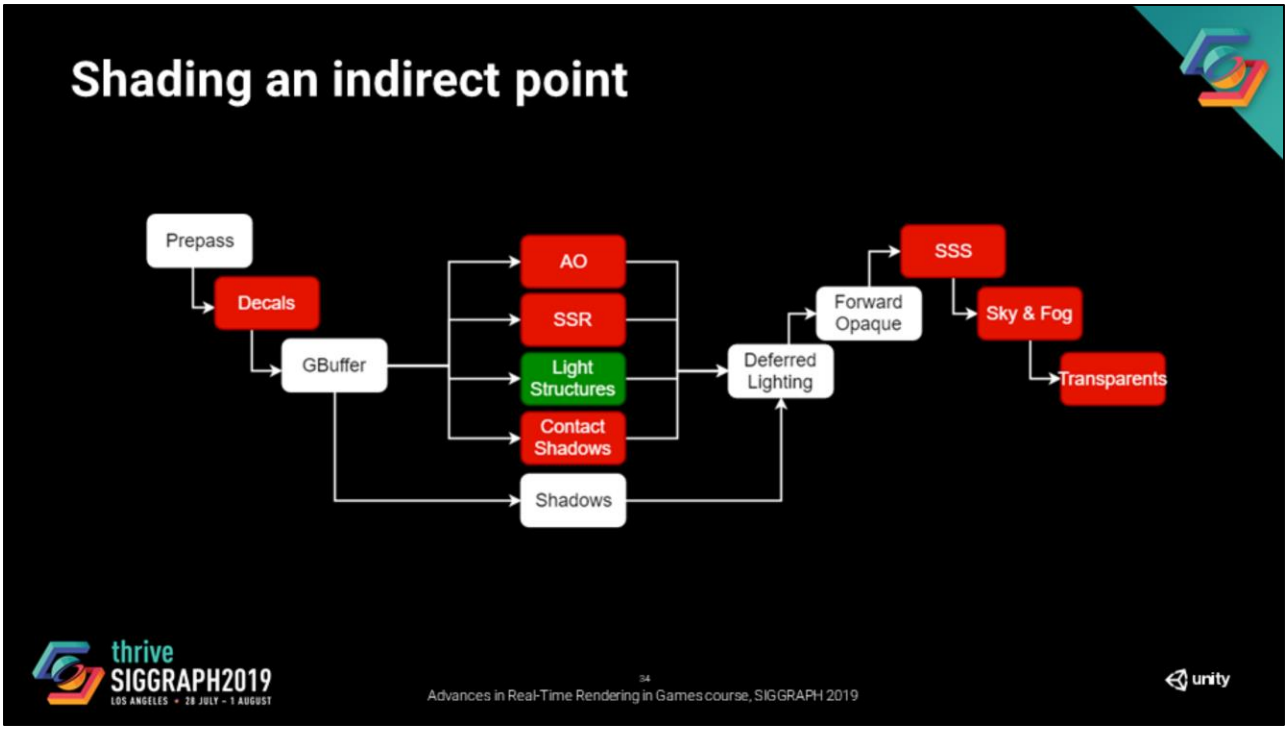

The approach for ray tracing is quite similar, we start with our camera and the same set of light volumes in our scene.




We do a culling pass to define the set of lights that we need to include in our structure, this time it is not with with the frustrum, but in proximity of the camera.




Then we build a structure that hosts a light lists per cell like it would for the screen space structures. This structure has a fixed resolution and centers around the light volumes while staying in the previously defined camera neighborhood.

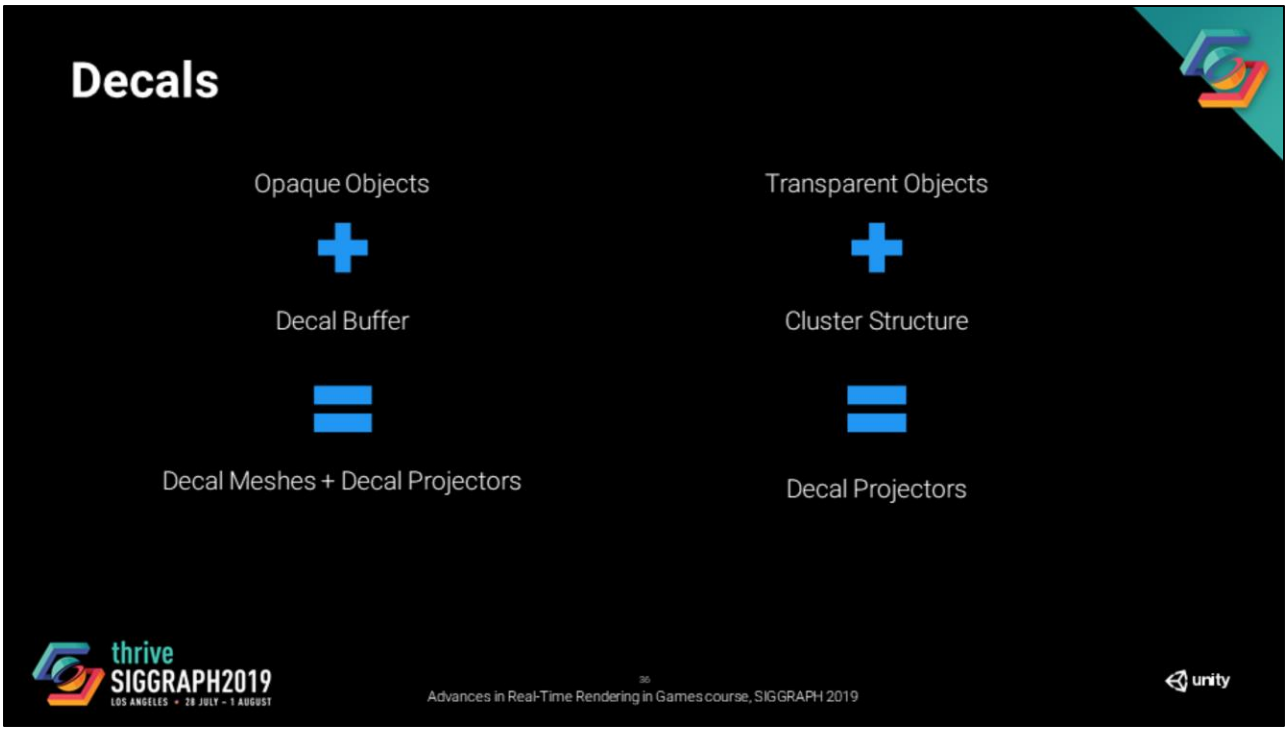



This allows us to have an equivalent to the rasterization light structures. That said, we are not done.

Let’s investigate the other missing data. The next one we are covering is…


Decals. As you can see, they make a pretty big different on an image.

We need to have a solution for these primitives.




In the render pipeline we are starting from manages decals in two ways

For opaque object, we support decal meshes and decal projectors and handle them using the decal buffer.

For transparent object, we only support decal projectors and handle them through the cluster structure (same structure as for lights).

## Decals

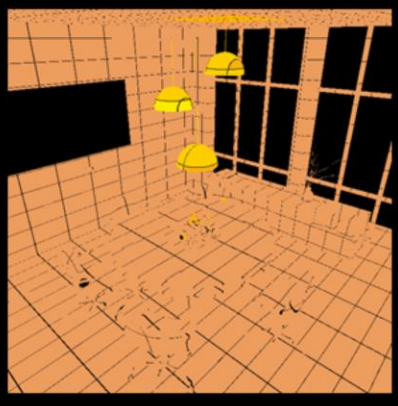
Indirect Point

+

World Space Cluster

=

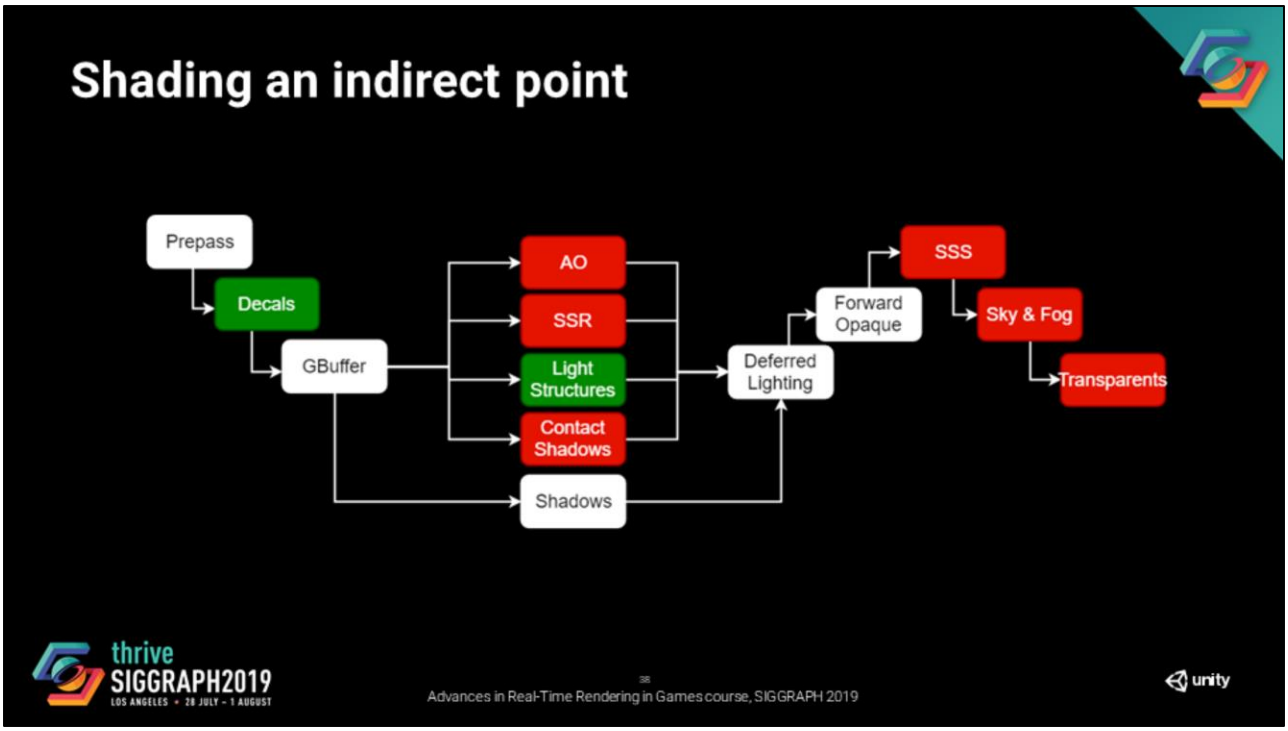
Decal Projectors



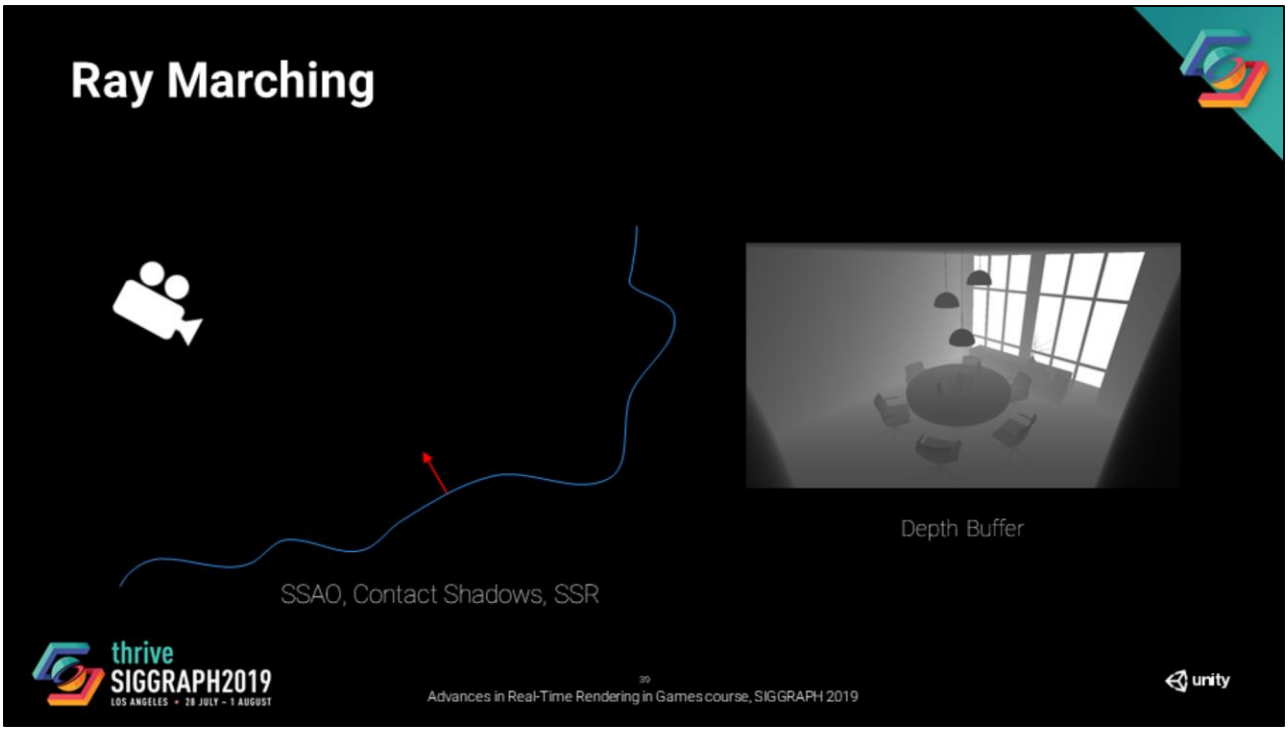
Cluster Structure – Debug View



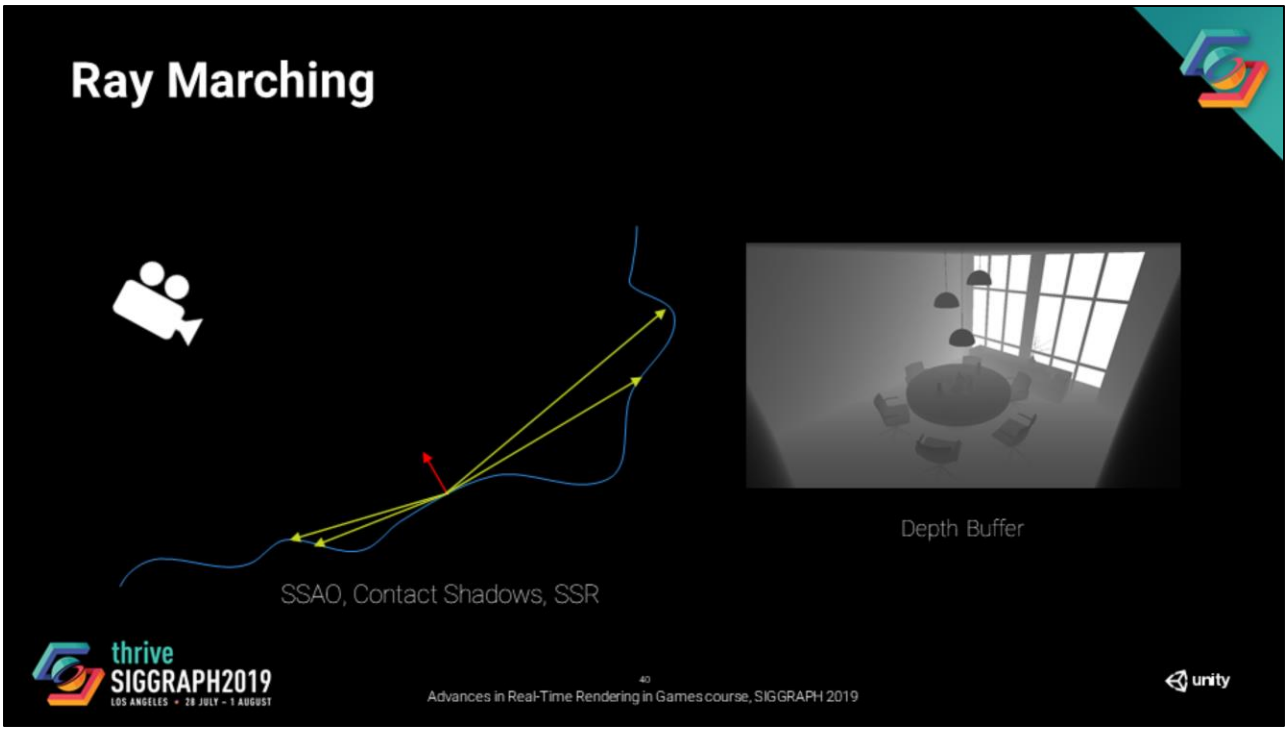
We now have an equivalent to the rasterization cluster structure. If we limit ourselves to decal projectors in ray tracing (which is most use cases of decals) we have an approach for decals.




That « solves » it for decals, let’s investigate the other missing data.

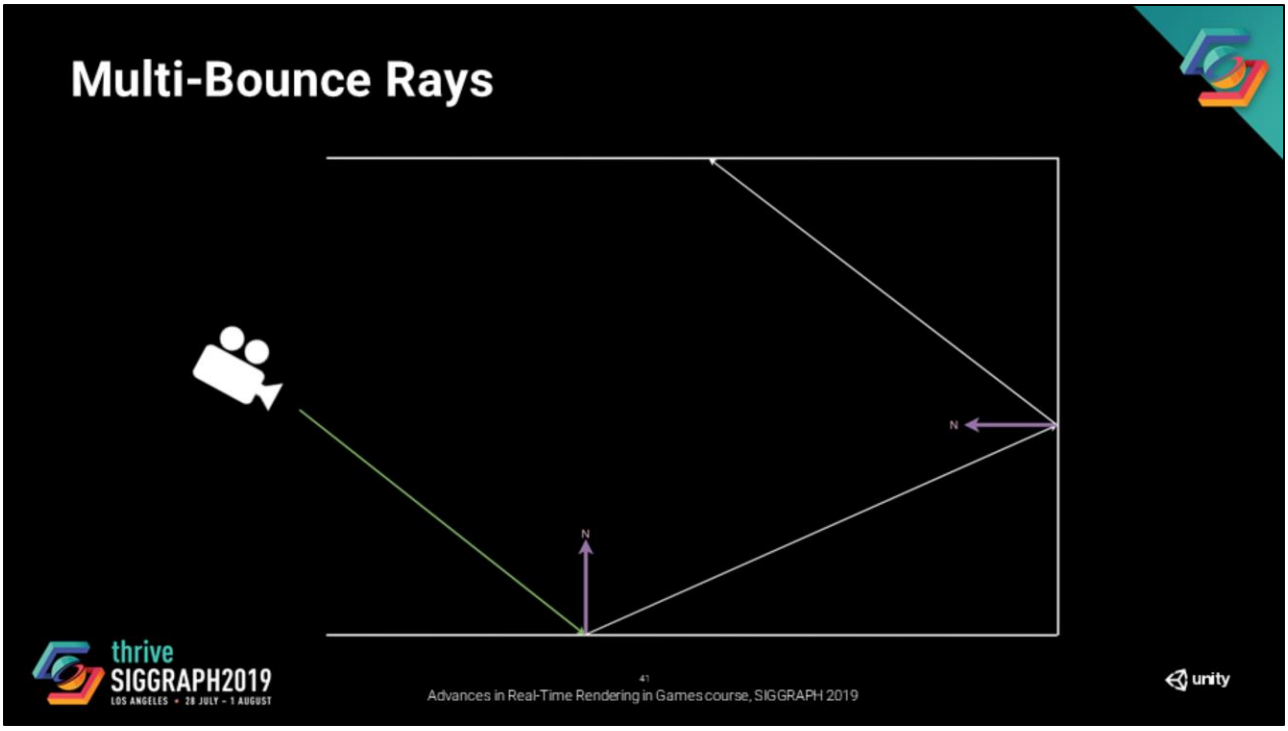



For our screen space lighting effects (SSAO, SSR, Contact Shadows), we use ray marching to evaluate the signals.




The goal here is not to explain what ray marching is.

Simply to point that it fetches geometry or lighting information of an other point in the scene.



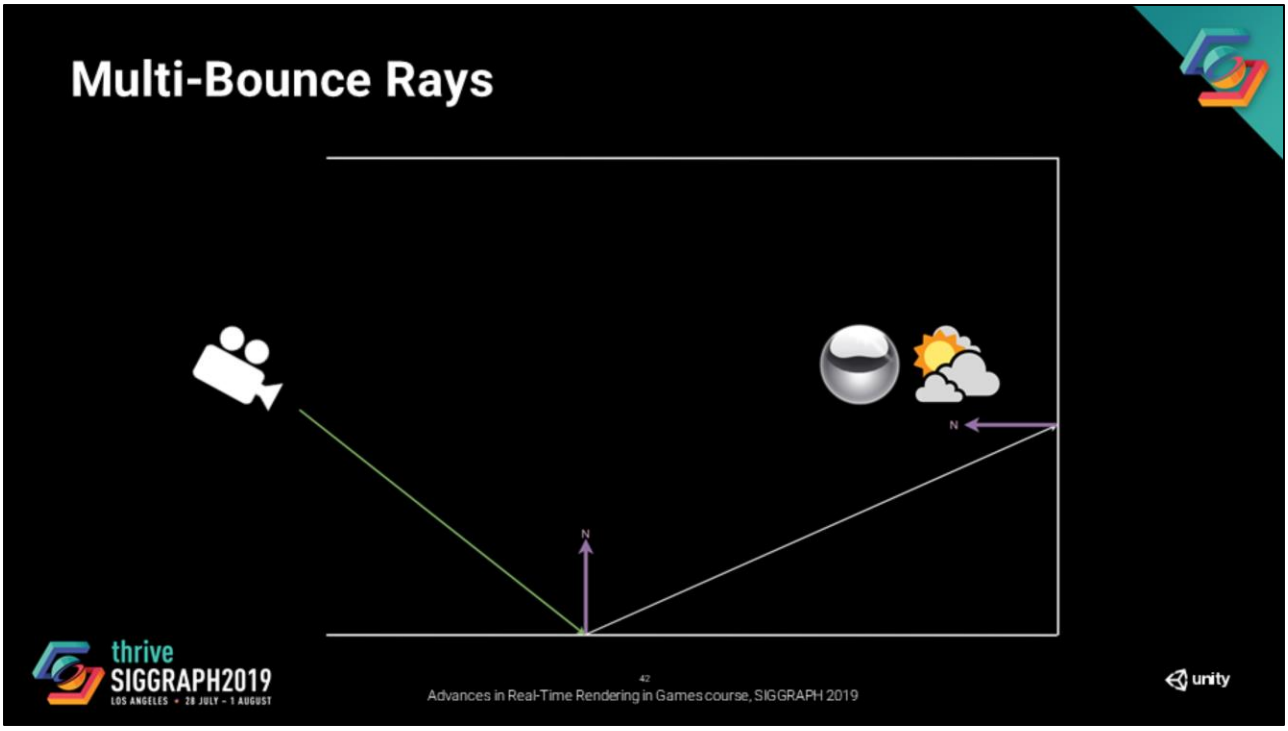

And that sounds like a straight translation to ray tracing, right?

Indeed, the first option that we have is to use multibounce to solve these problem.

This has theoretical advantages, such as solving our problems without baking.

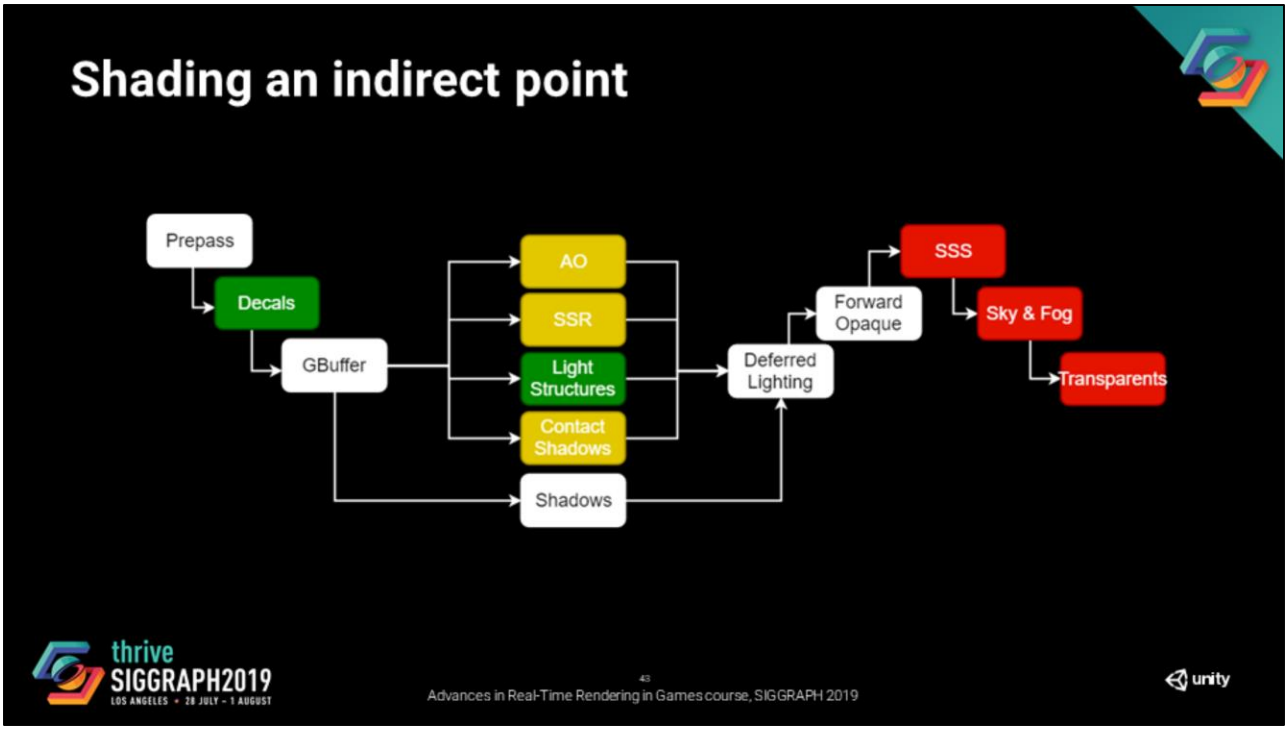
However, with a similar sample per pixel count it we are evaluating a bigger sample-space (two additional dimensions, or more depending on what we are doing). This means that either we need more samples (a lot more) or we need to filter more aggressively (which impacts the overall image quality).

The other major drawback for this approach is performance, this is not viable for games (we will see how expensive this can be later in the presentation).



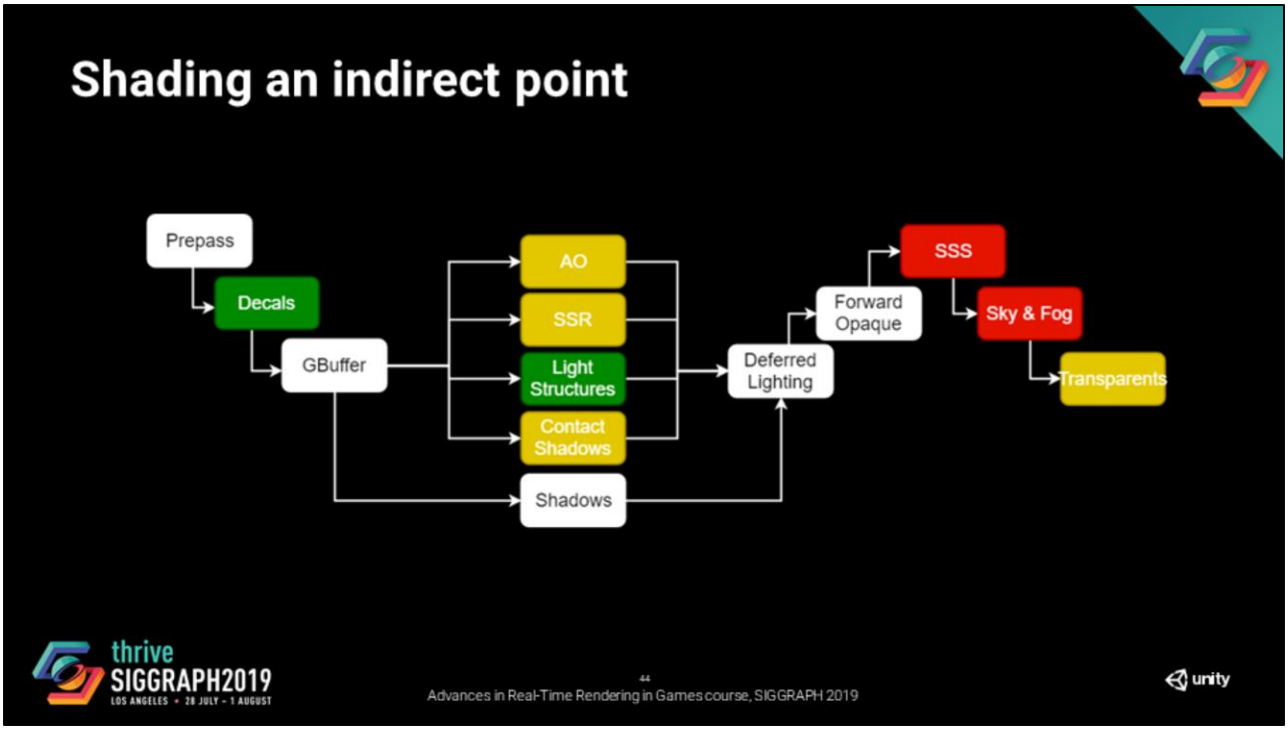

The second option is to settle for approximations. Such as reflection probes, light probes, shadow maps, etc.

For most cases this is enough given the signal that we are evaluating is a rough lobe and does not require as much precision as the screen space pixels.



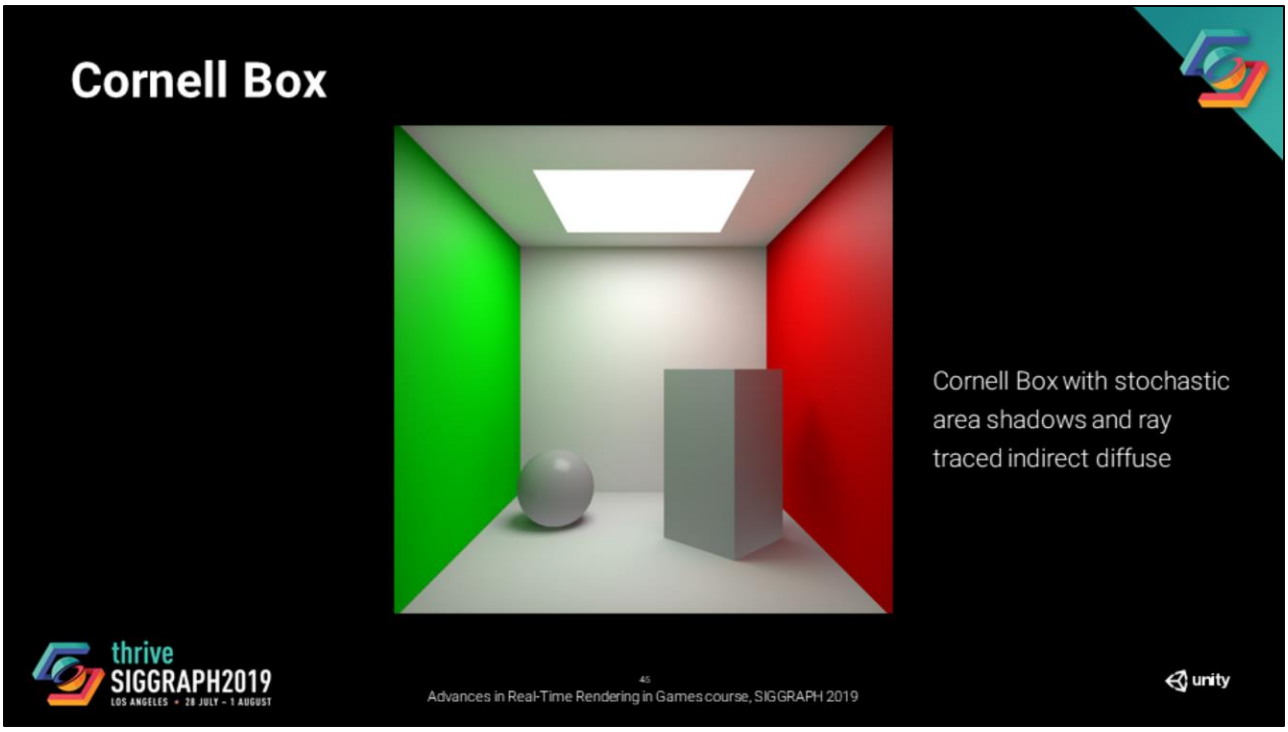

For yellow steps our solution is either multi-bounce or approximations.

The next thing we are looking at is transparency. It is seems like a good candidate for ray tracing right?



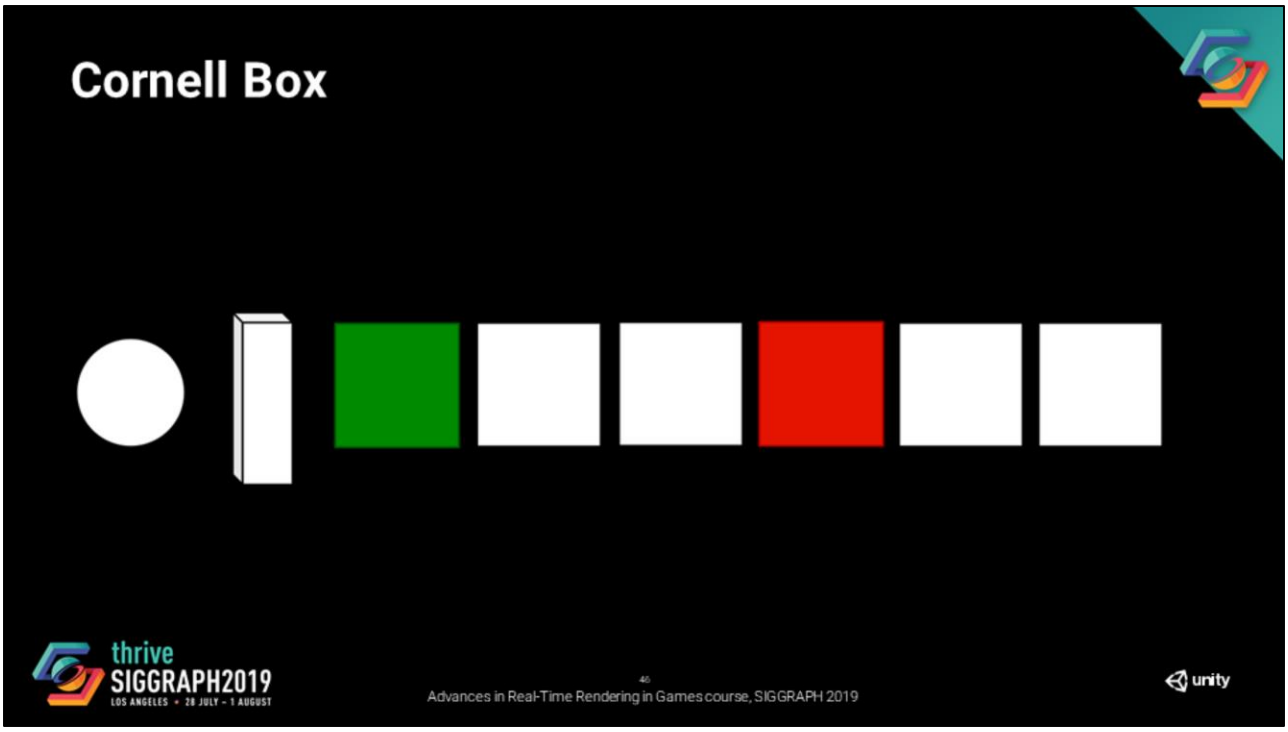

It turns out the same problems that we have mentionned for the screen space lighting are also valid for transparents (multi-dimentionality, performance, filtering). Right now, we have solutions but they do not scale for games.

For SSS and volumetric scattering we are still exploring the options.



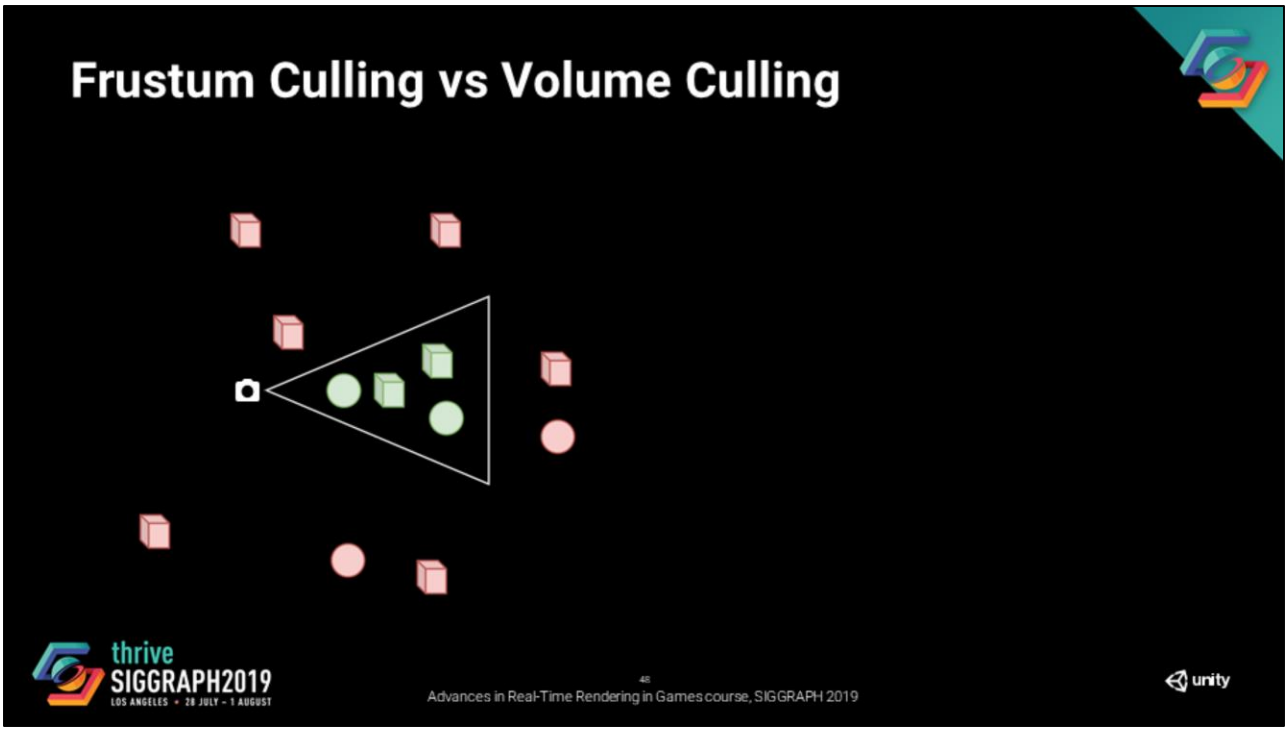

Now that we have an idea of what information we require, let’s figure out what are the constraints that apply on the data.

Let’s take a very simple example, a cornell box.

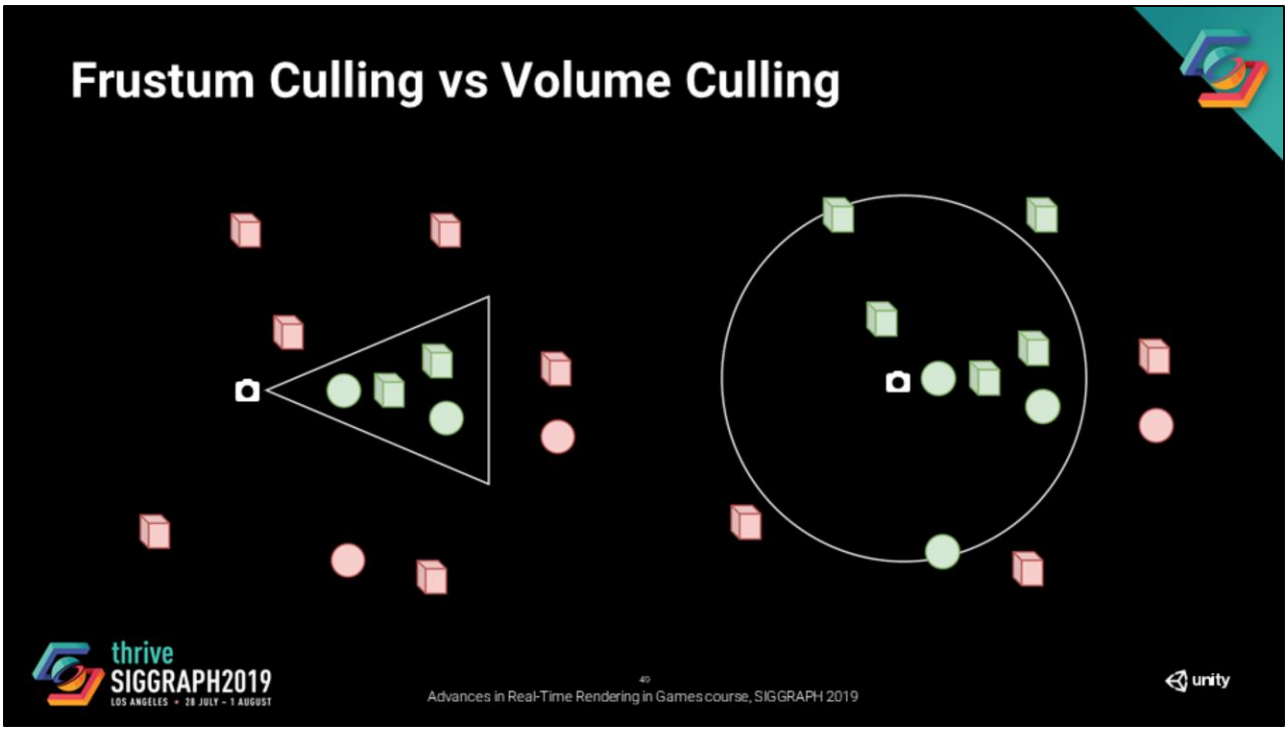



We can decompose this scene as a set of entities.

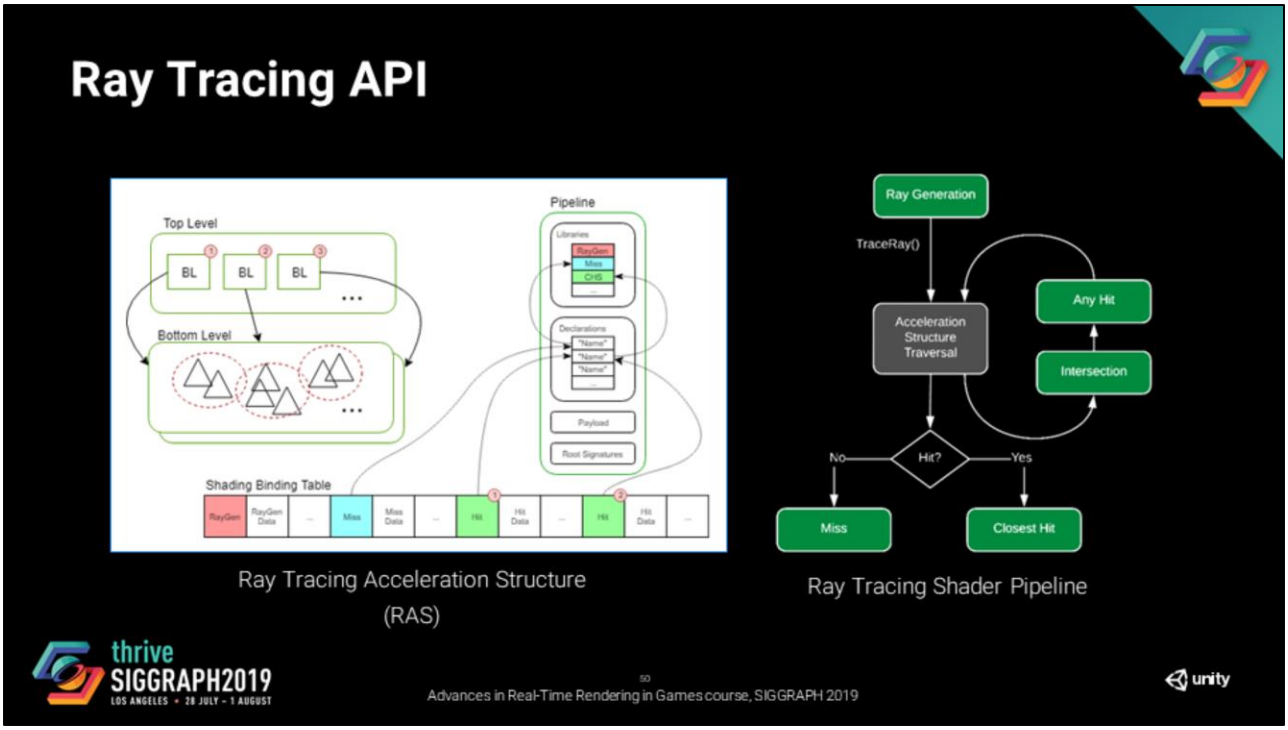

These entities have material properties, meshes, and tansforms.




When we render the scene using rasterization, we have a culling pass that defines the set of objects that we should feed to our draw indexed call.




For ray tracing, the idea is the same. However, this time the set of instances is used to build the ray tracing acceleration structure.

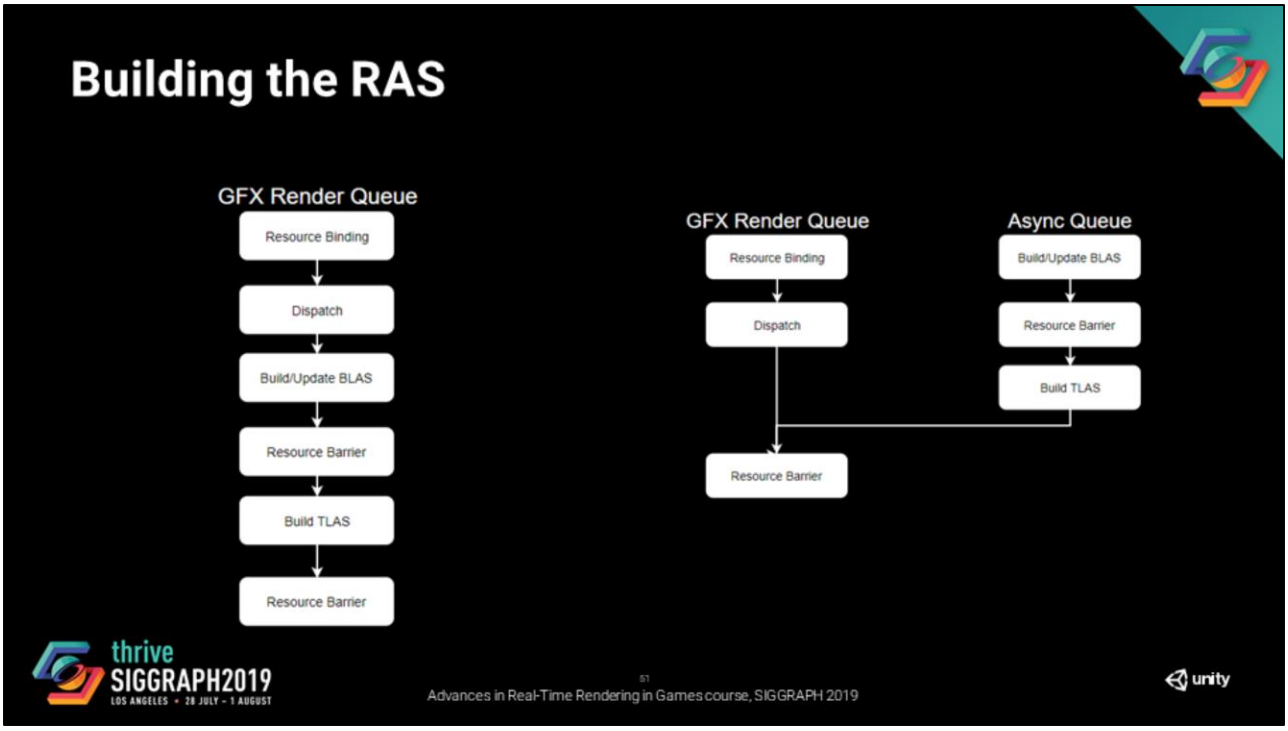



This presentation is not meant to be a ray tracing API tutorial. So here is a link to the most complete API description i've seen so far [https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html](https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html)

That said, I'd like to refresh two things:

- Raytracing acceleration structures (RAS) are two-leveled BVHs (TLAS and BLAS) and they need to be bound to a set of shaders and resources.

- The diagram on the right the that I'll be using a later in this presentation. It represents the shader stages of the raytracing shader pipeline (it is missing the callable shaders, but I am referring to them in this presentation).

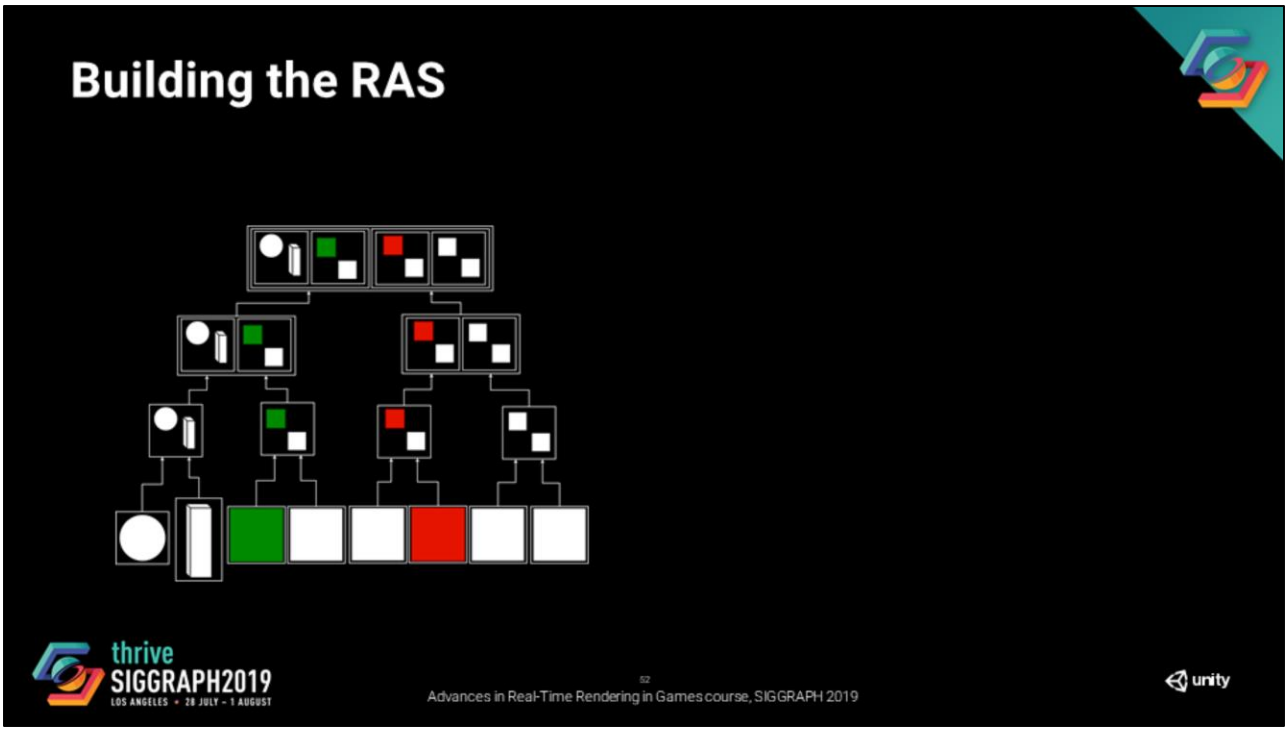



There is couple of options available to us on how to build the RAS.

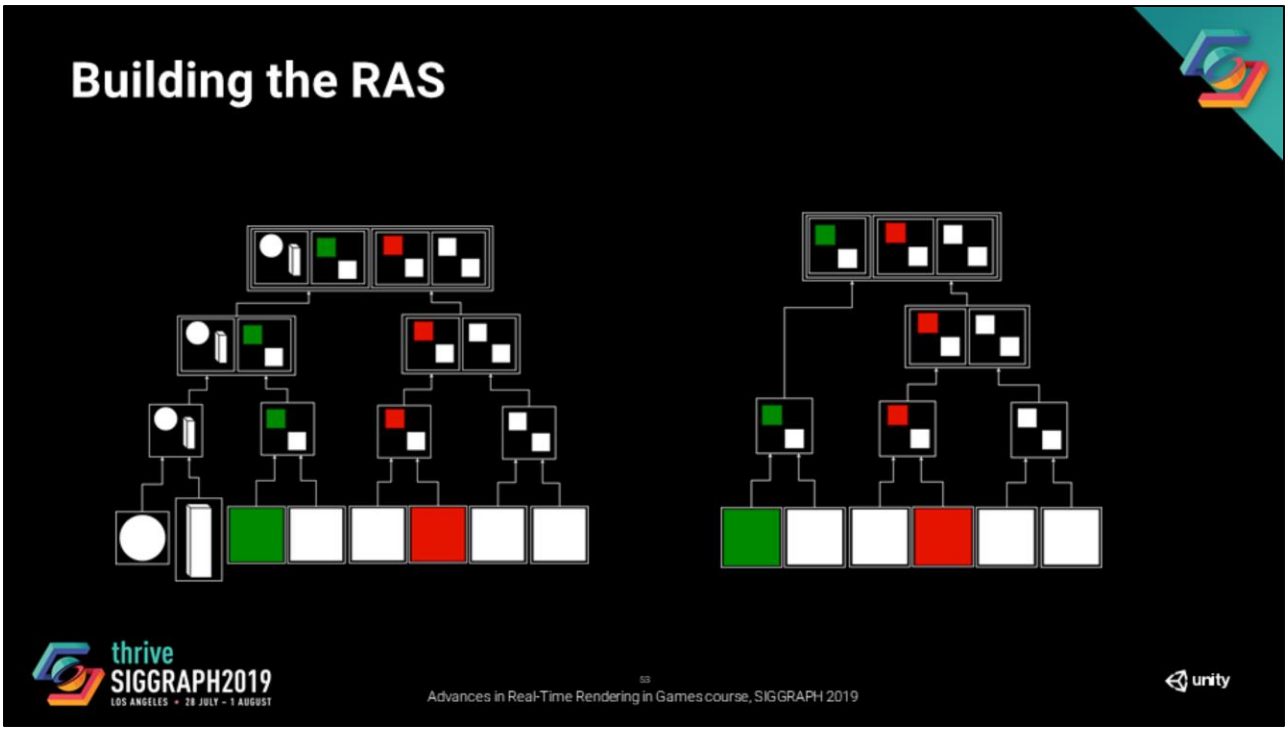
First of all, we can either do it in the gfx queue or the async queue. That depends on if you have other dispatches that you can use to hide the cost of the build before the first use of the RAS. It is entirely dependent on your pipeline. Note that if you are already making heavy usage of the async queue, you should measure it before using the second solution.

The second thing is, as soon as your scene is going to get anything bigger than couple cubes, It is going to get very expensive to rebuild/update all the BLAS (especially so if you have couple of skinned meshes, or partcile systems), you need to define a budget per frame and then, based on your budget, define a prioriy queue for your rebuild and updates.

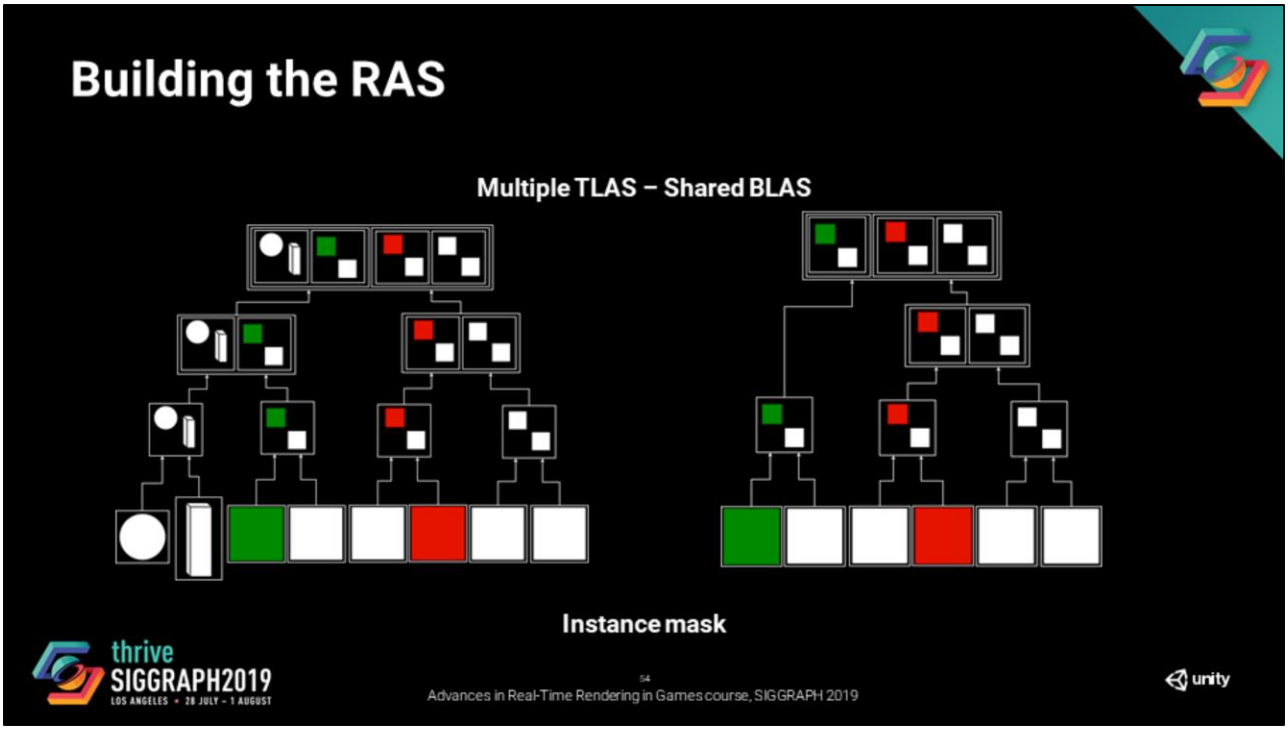
Then you rebuild your TLAS (either way).




If we do this, we endup with a TLAS for the whole scene, which is what you would want in most cases.



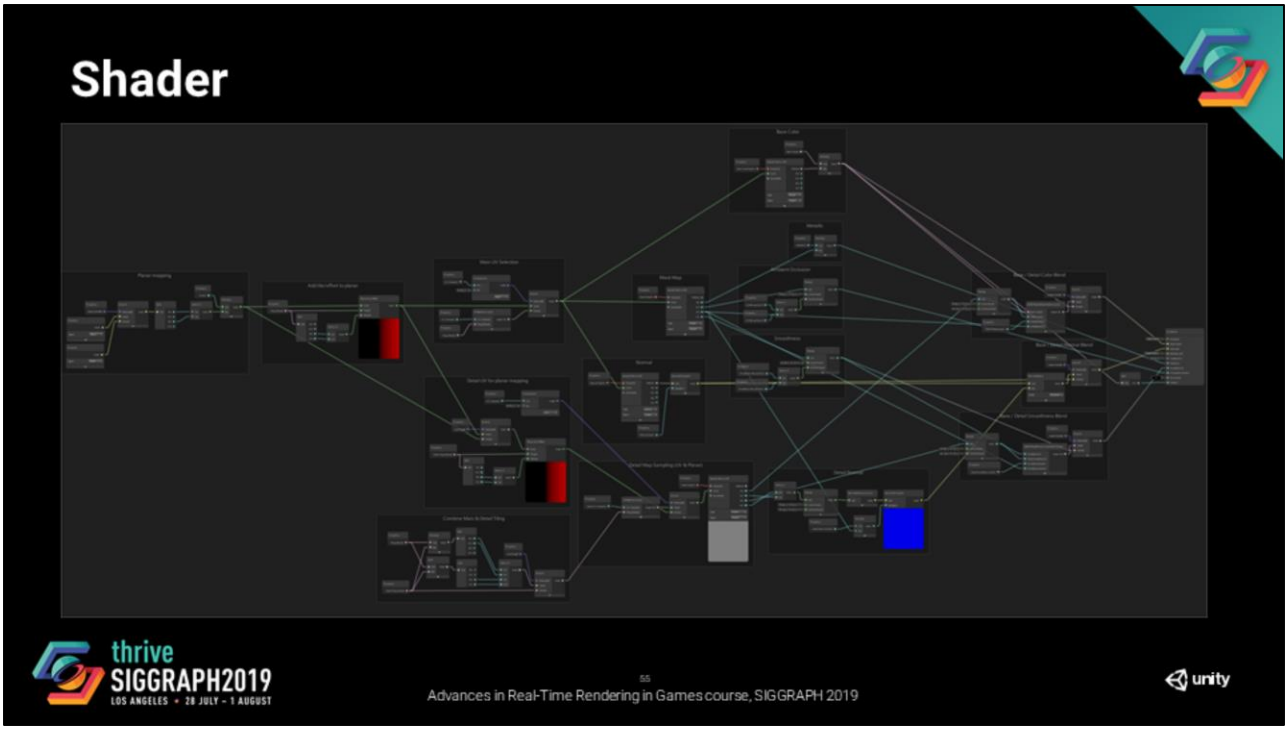

However, sometimes for artistic or performance reasons, we do not want to have the same parts of the scene for different effects (or you may have multiple cameras in different regions of your virtual world).




Then we have two options:

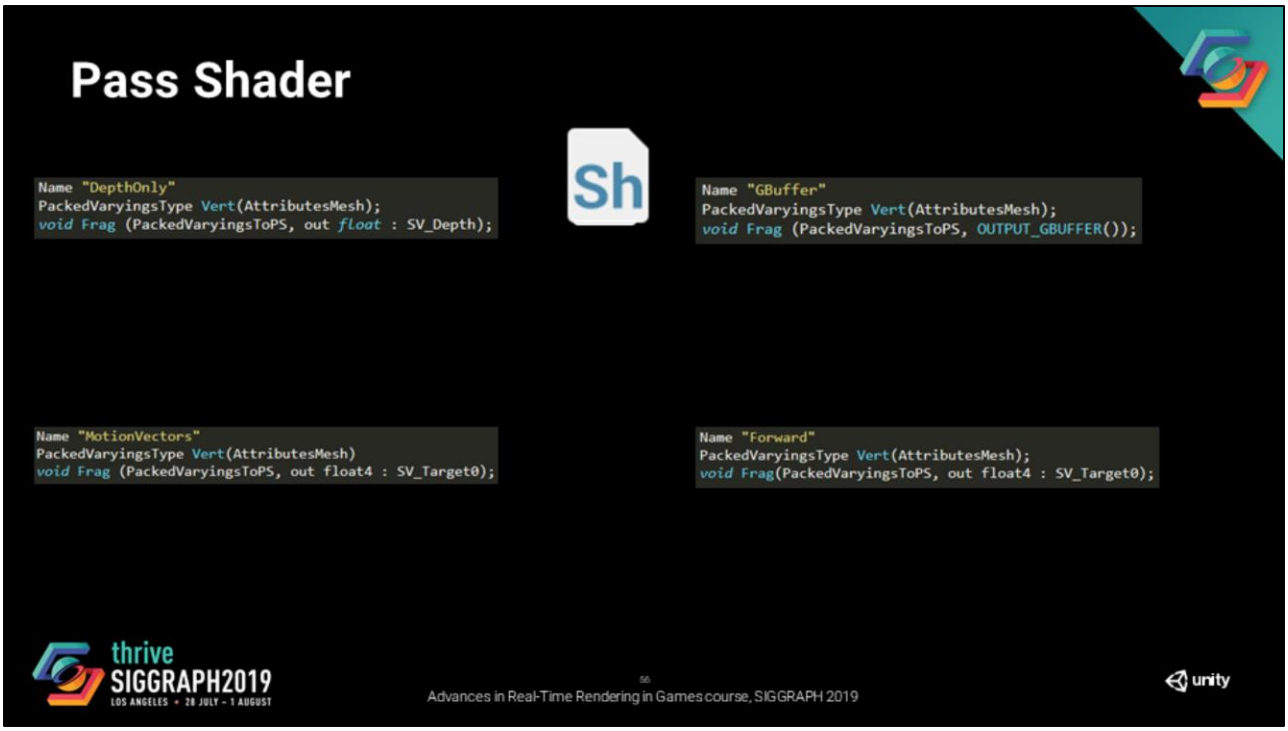
- Share the BLASs among multiple TLASs (but that means building multiple TLASs every frame).
- Build only one TLAS, but use instance masks to exclude objects (but that means paying the full traversal cost for every ray for the TLAS).

There is no absolute best. From our measures, the right choice is really content dependent. The important piece information here is that you should measure and pick the best choice for your configuration.




The next point that I would like to cover is the set of shaders that should be bound to the RAS.

In most game engines, users are able to describe extensively their shaders as graphs. This is a simple example, it can get much worse than this.




In rasterization, we usually generate a set of shaders from that graph and use them for the passes in our pipeline. Each shader usually maps to a single render pipeline pass.


The approach we found to be good for ray tracing shaders is pretty similar. From that graph we generate a set of shaders. Here are the shaders that we settled for and use to render the effects previously presented.

These shaders are classified by the type of the effect meaning that they are dependent on the type of signal we want to evaluate at the indirect point. For instance:

# Effect Type Shader

```
Name "IndirectDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```

```
Name "ForwardDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```

```
Name "VisibilityDXR"
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```

```
Name "GBufferDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```



We use indirect for indirect signals (specular and diffuse). Both effects are different, but the information that we compute at indirect point is (almost) the same.

# Effect Type Shader

```
Name "IndirectDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```


```
Name "ForwardDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```

```
Name "VisibilityDXR"
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```



```
Name "GBufferDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```

thrive SIGGRAPH2019 LOS ANGELES • 28 JULY – 1 AUGUST



unity

Visibility is used for ambient occlusion, directional shadows, and stochastic area shadows. Note that this shader does not need the closest hit shader, which is significantly cheaper.

# Effect Type Shader

```
Name "IndirectDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```


```
Name "ForwardDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```

```
Name "VisibilityDXR"
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```



```
Name "GBufferDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```



Forward is used to evaluate recursive tracing

# Effect Type Shader

```
Name "IndirectDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```


```
Name "ForwardDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```

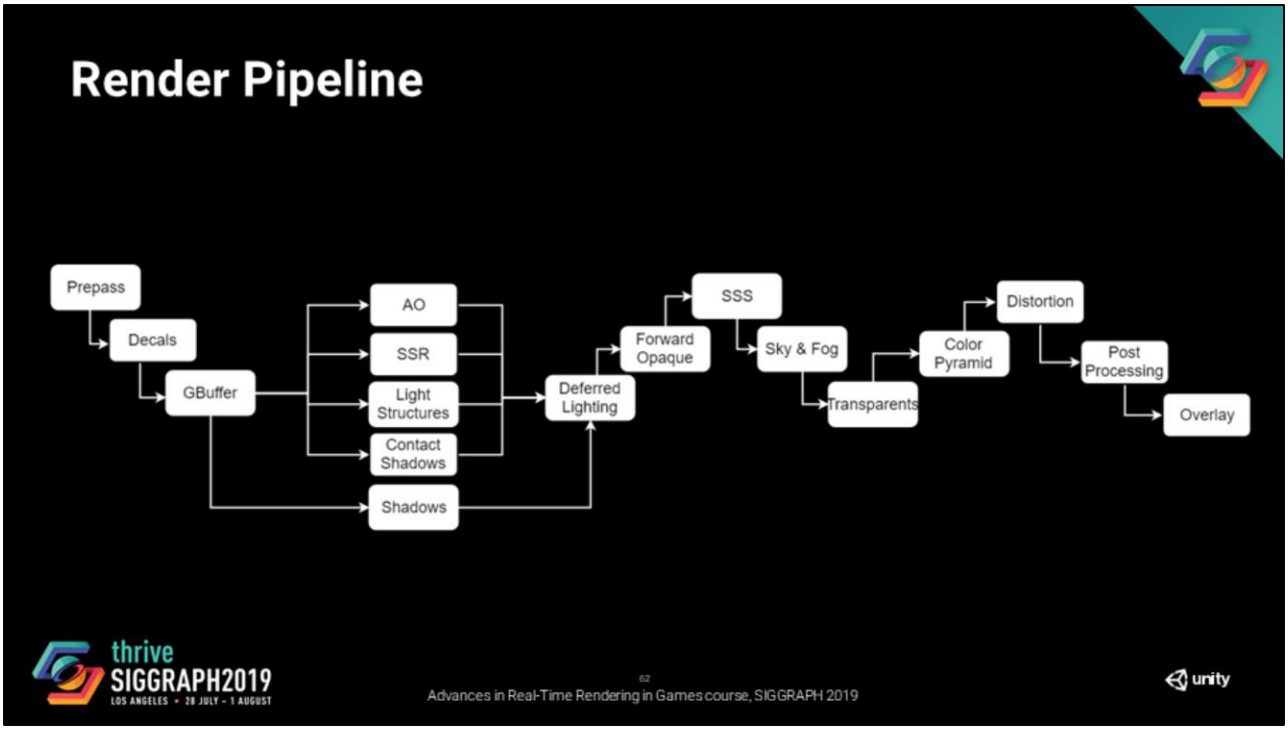
```
Name "VisibilityDXR"
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```



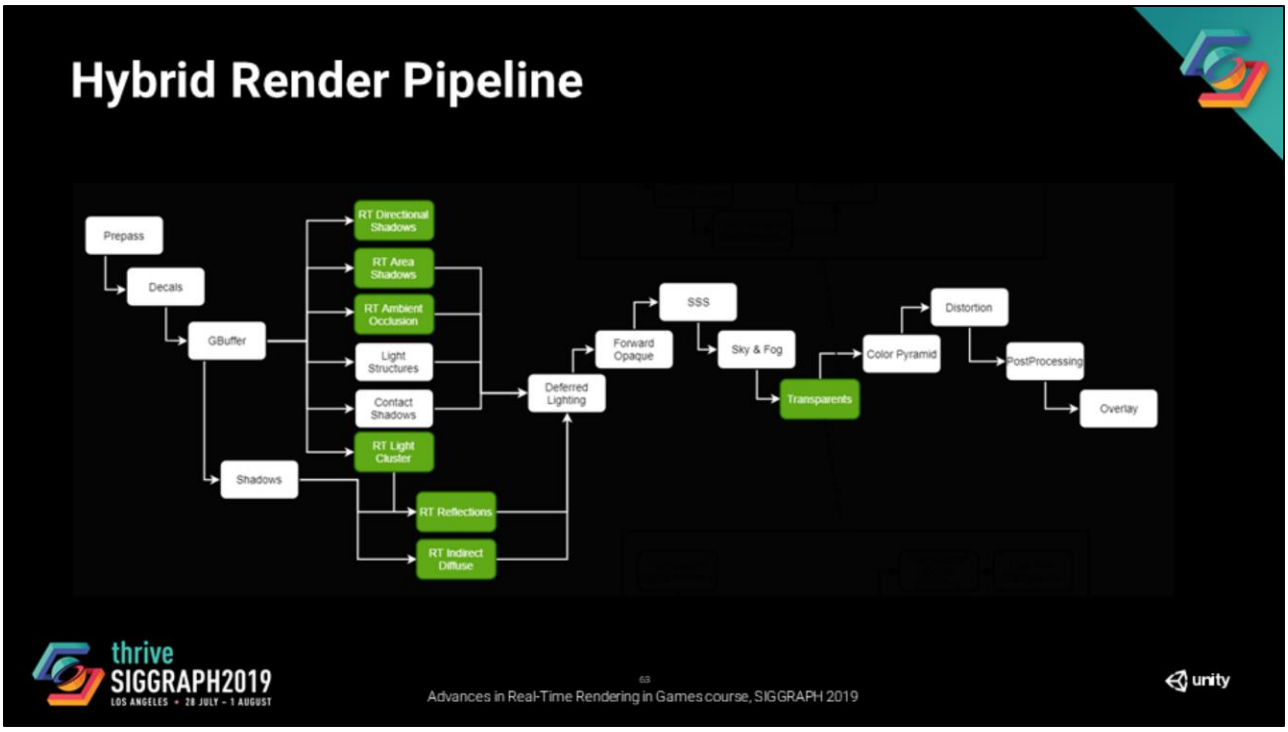
```
Name "GBufferDXR"
[shader("closesthit")]
void ClosestHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
[shader("anyhit")]
void AnyHitMain(inout: SV_RayPayload, : SV_IntersectionAttributes);
```



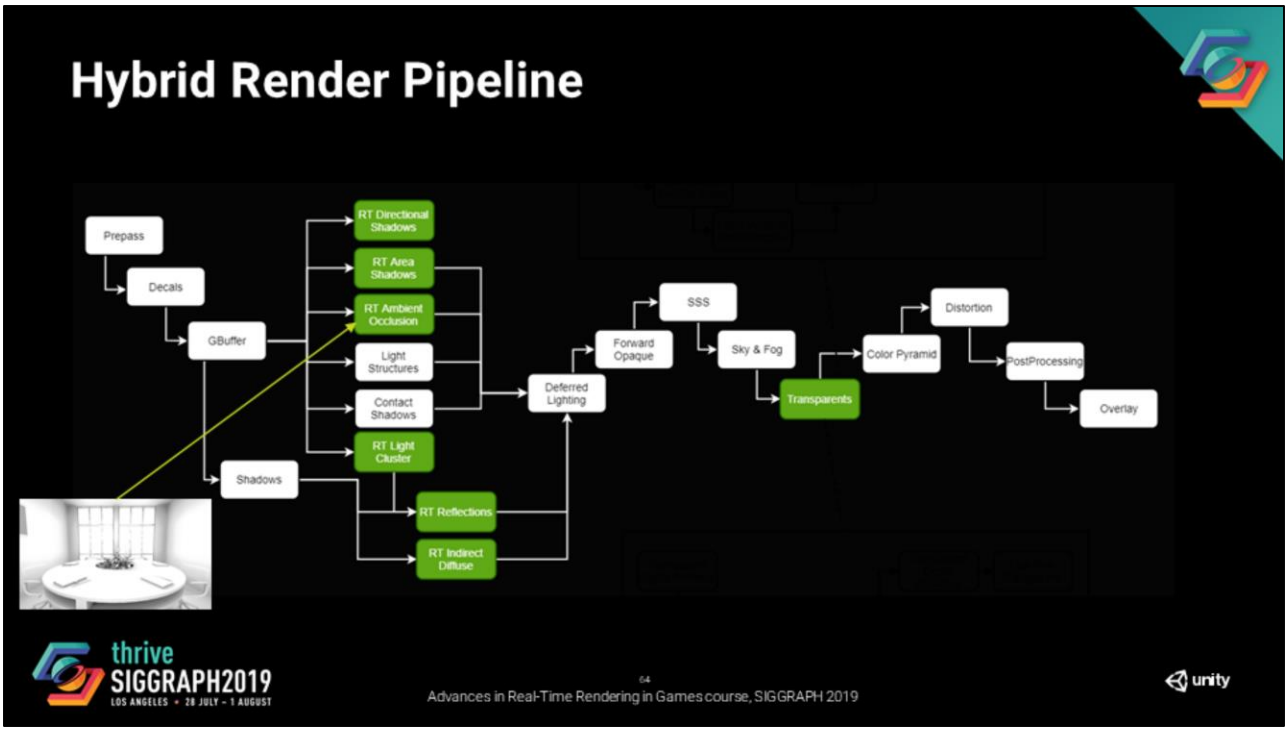
And i'll cover the last one « Gbuffer » later in this presentation.



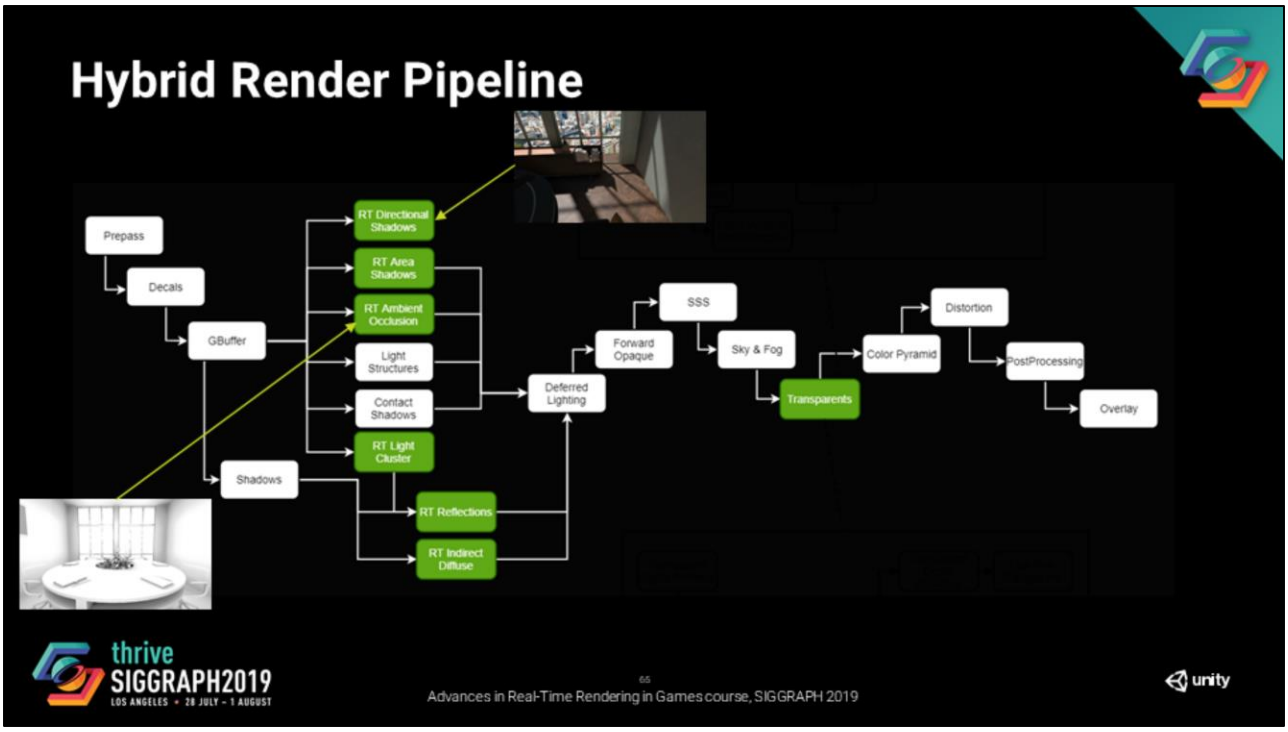

All of that said, let’s go back to where we came from. This is how the render pipeline looked initially.



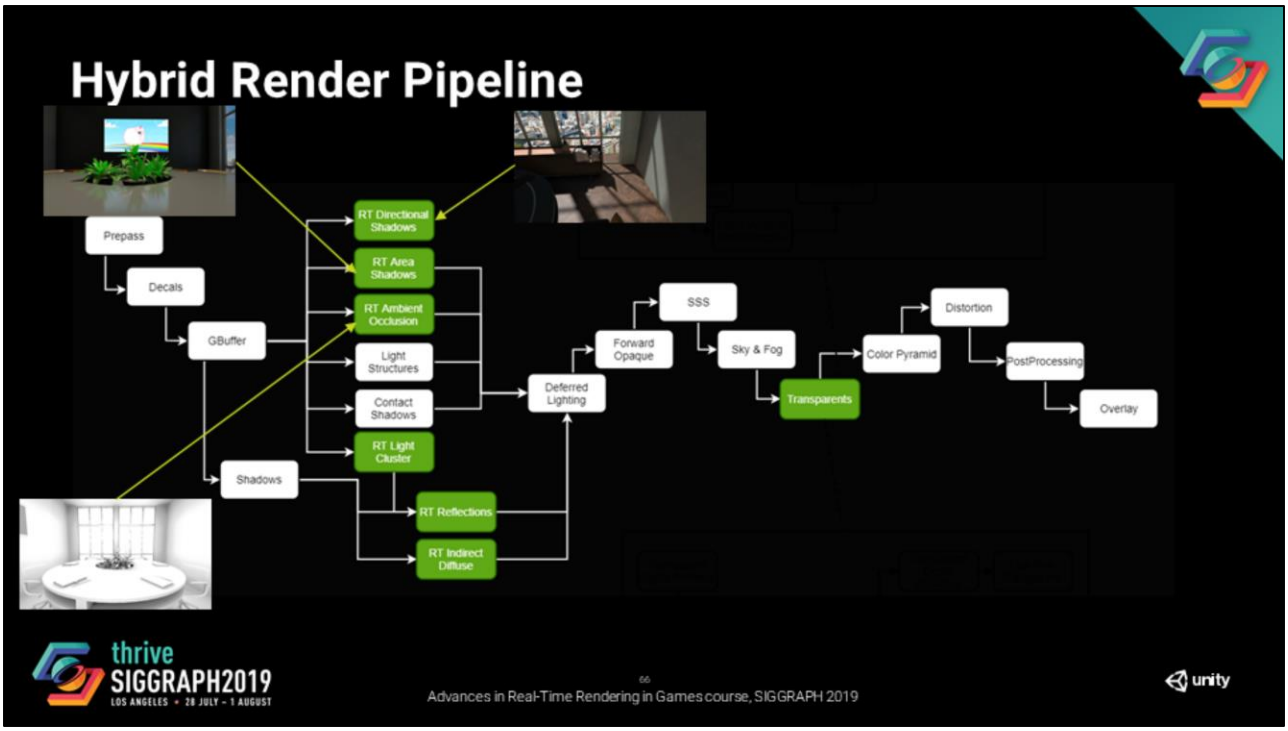

And this is how it looks now if we include everything that we described so far.



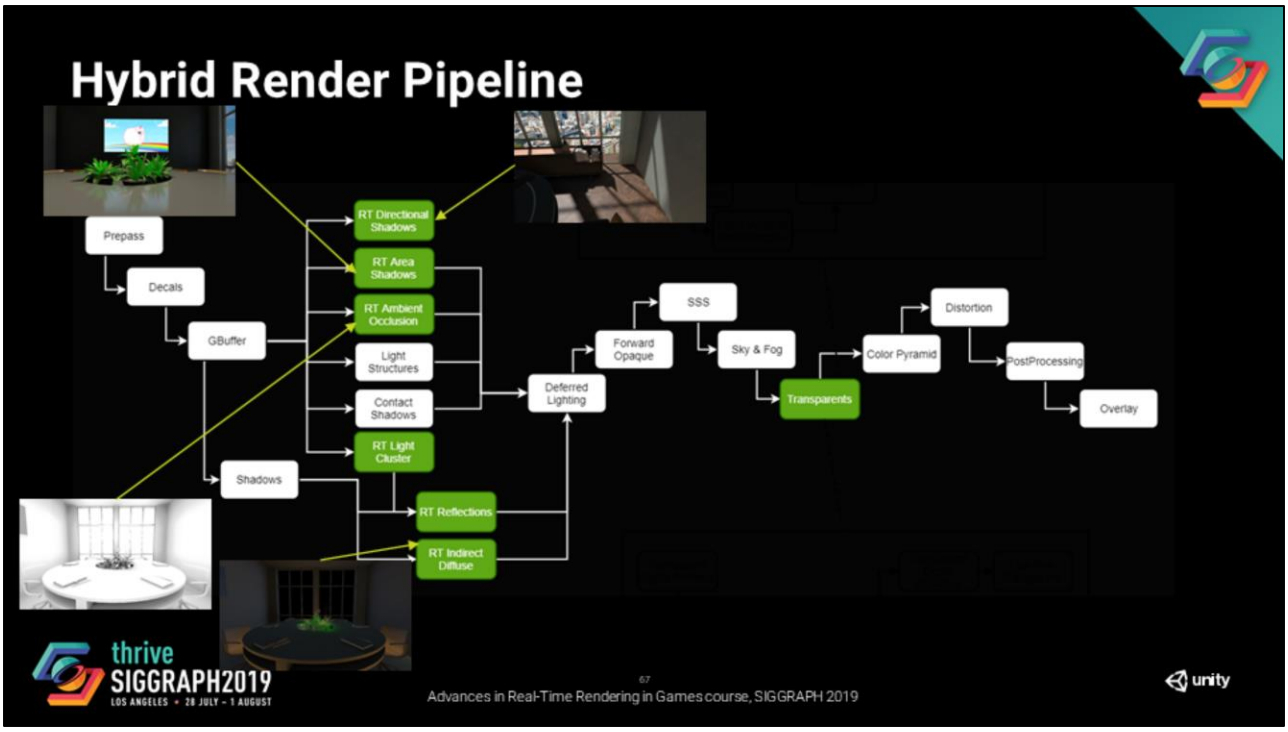

Ray traced ambient occlusion replaces SSAO, no major changes here.



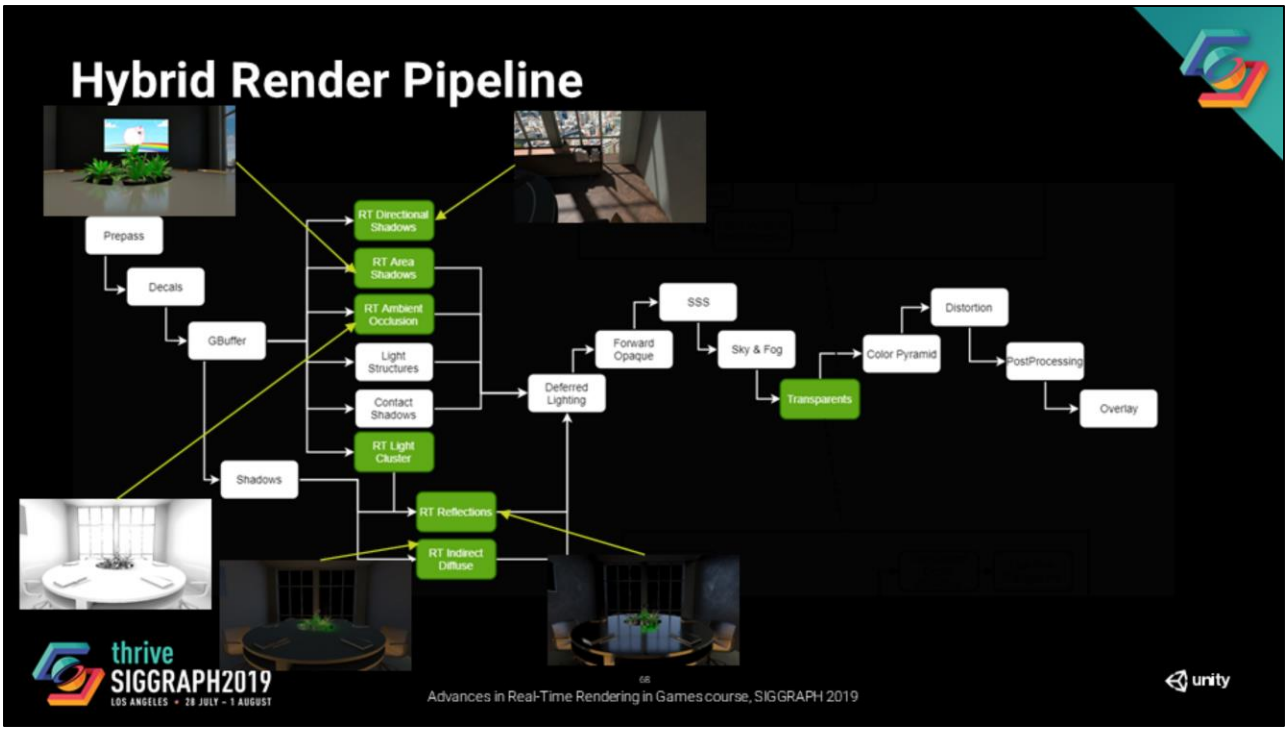

Ray traced directional shadow is a new step.



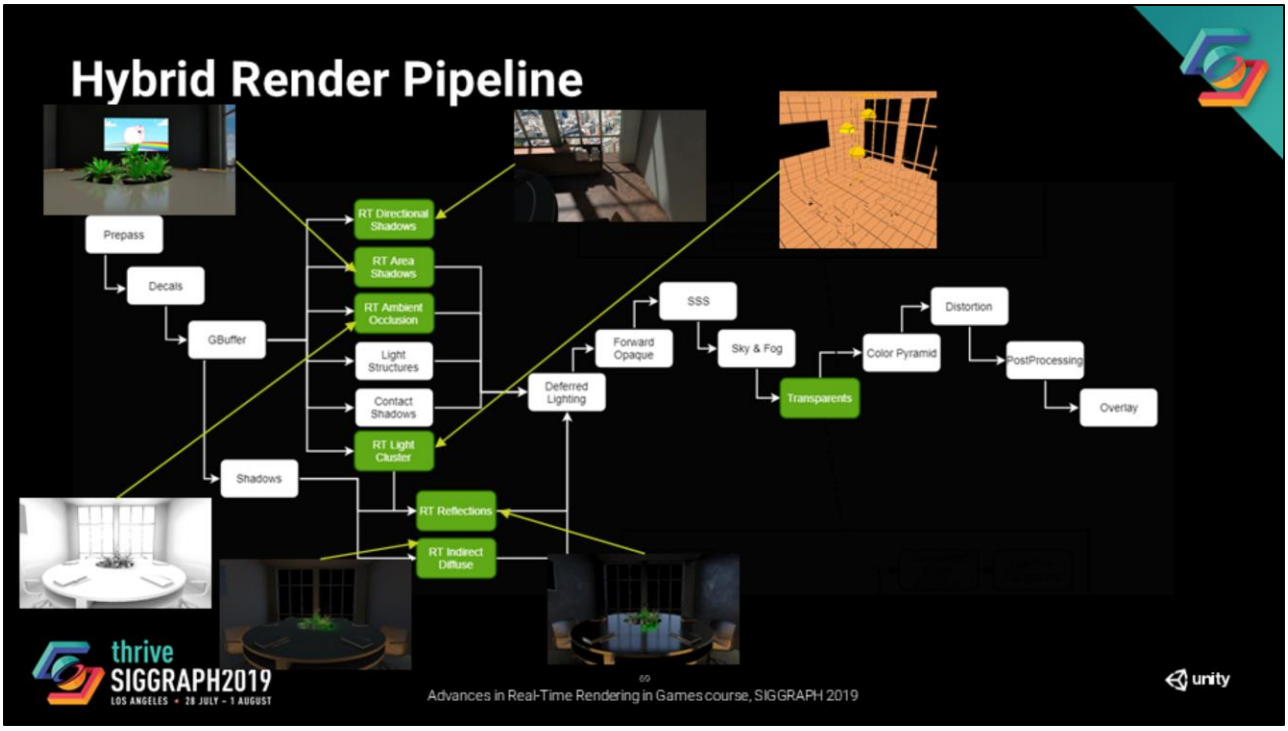

Stochastic Area Shadow is also a new step.



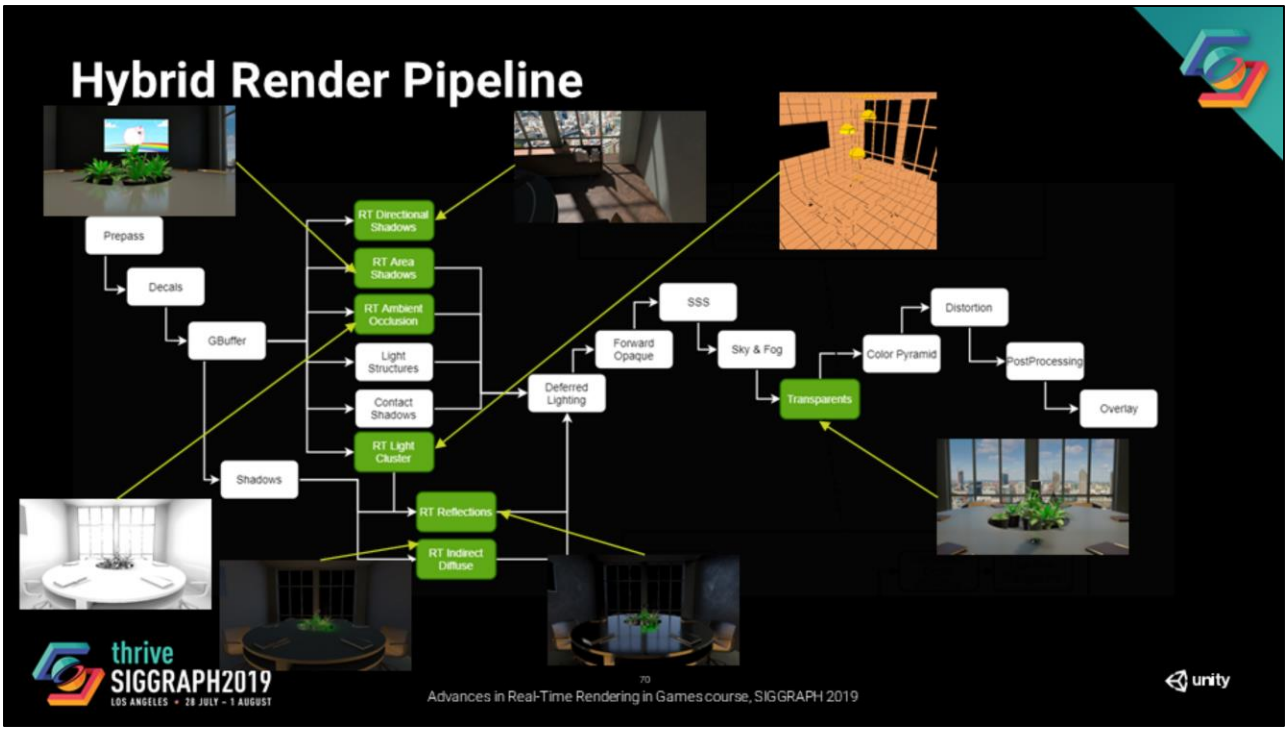

Indirect diffuse now depends on shadow maps and the ray tracing light cluster



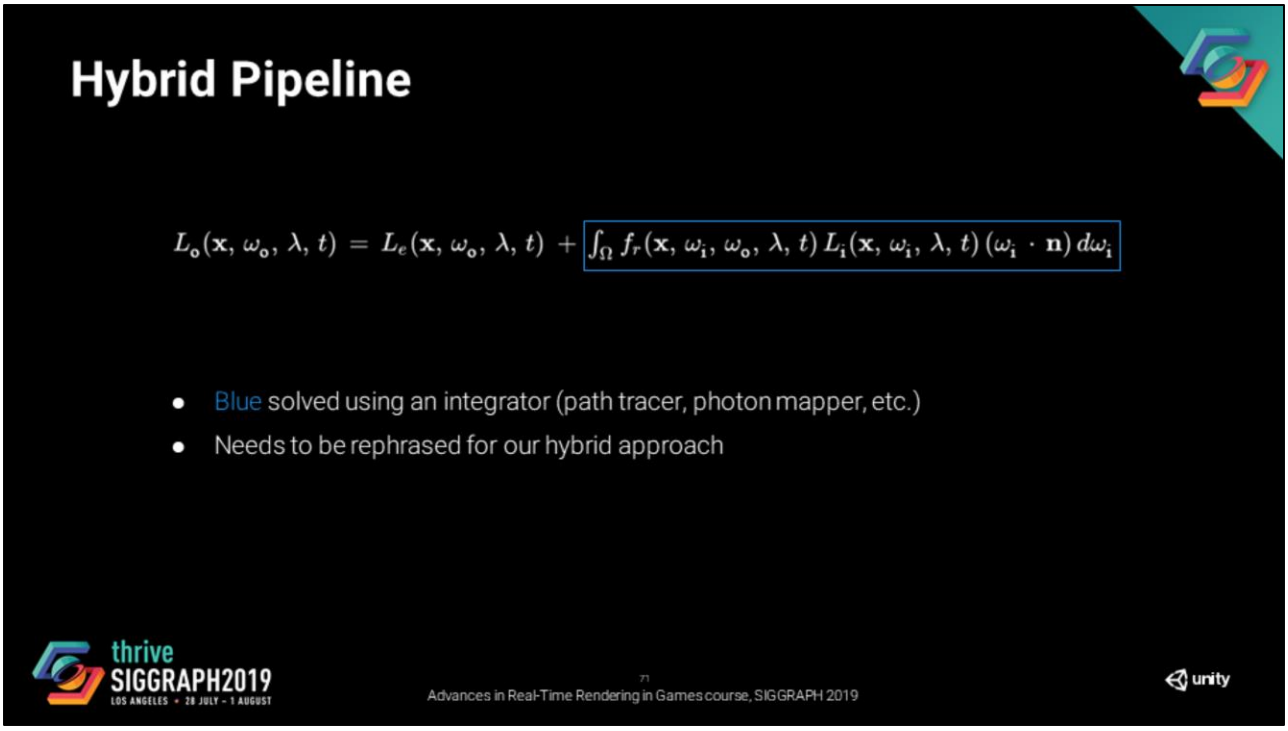

Same thing for reflections




Light cluster is here.



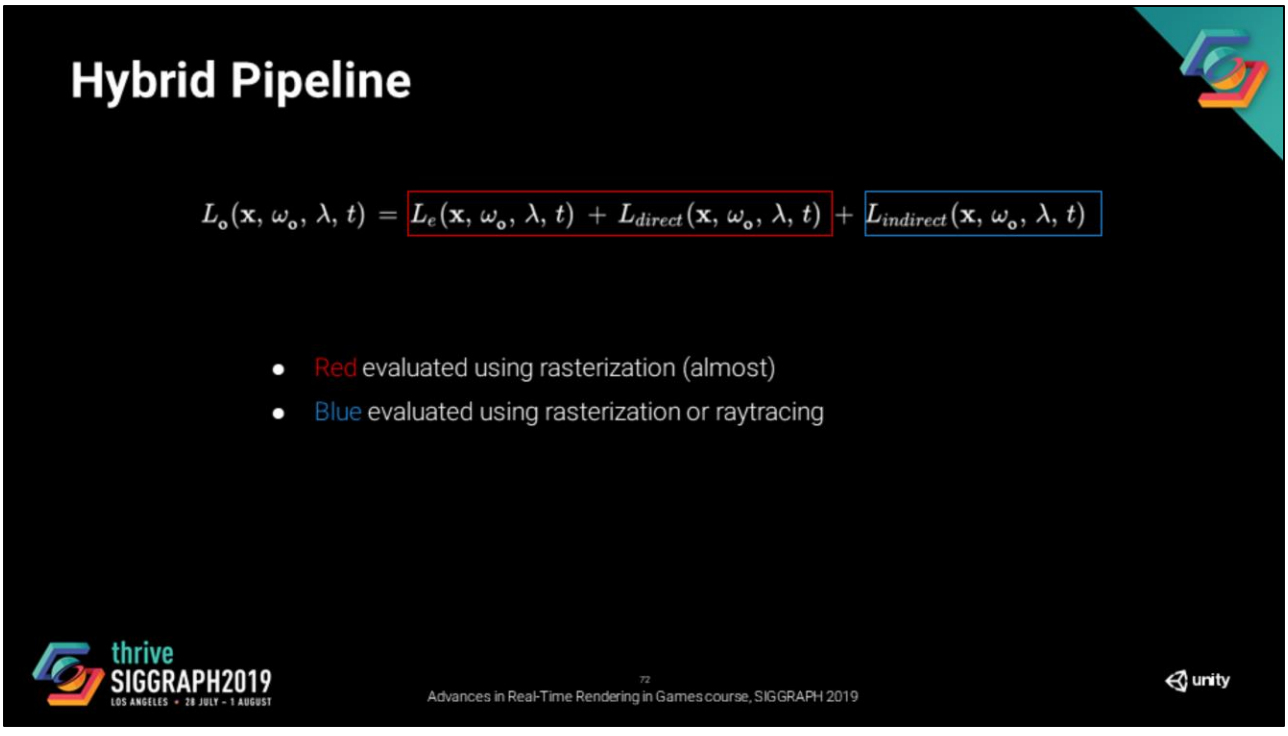

And recursive tracing is a replacement to the transparent pipeline for the objects that are rendered using it.




So far, we have covered all the plumbing that needed to be done to be able to call dispatch rays with the right object and the right material. Now let's figure out what exactly we should be computing.

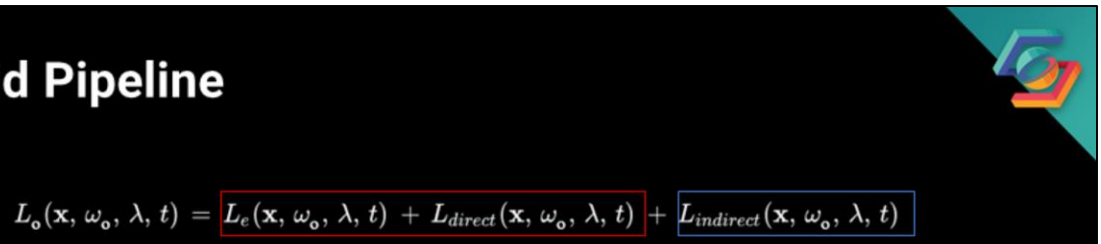
We start from the rendering equation, the part in blue is what is usually evaluated using an integrator, but given that we are going for a hybrid approach, we need to rephrase it a bit.

# Hybrid Pipeline




We decompose it as the set of emissive and direct lighting on one side (evaluated using rasterization) and indirect lighting (evaluated with ray tracing) on the other side.

# Hybrid Pipeline



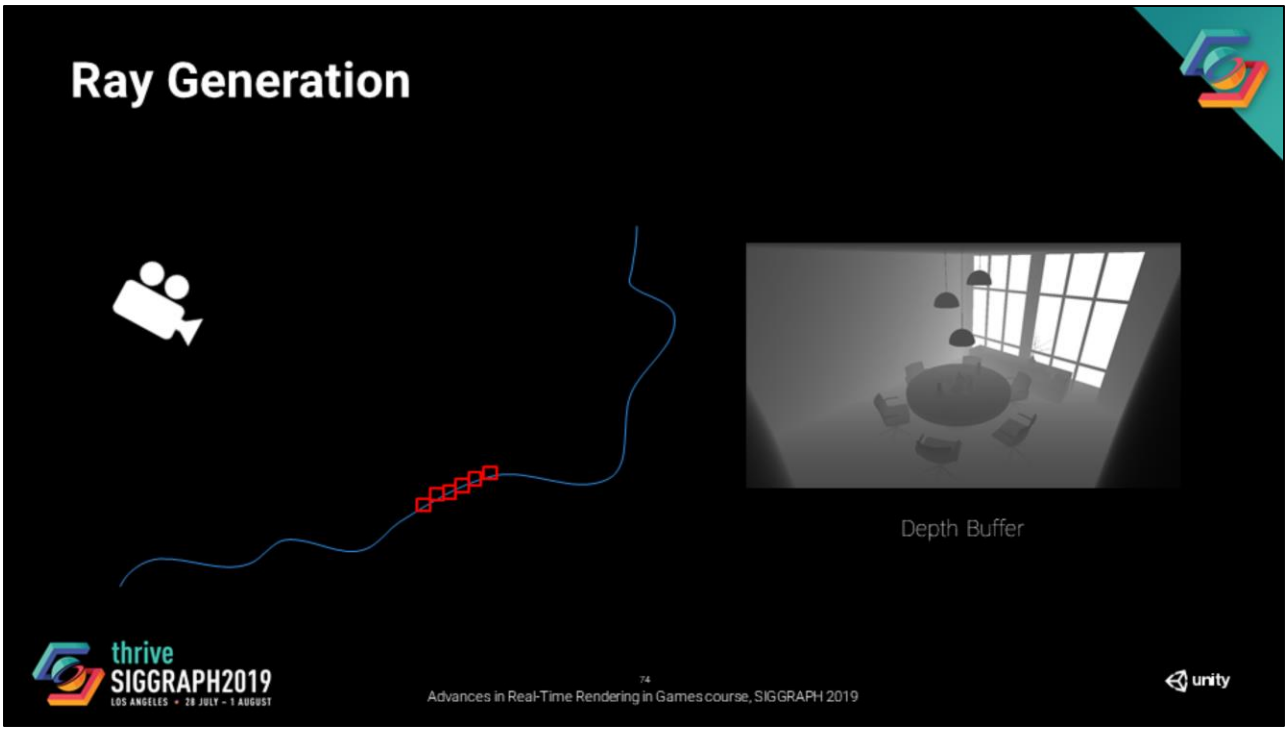
$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + L_{direct}(\mathbf{x}, \omega_o, \lambda, t) + L_{indirect}(\mathbf{x}, \omega_o, \lambda, t)$$

- Red evaluated using rasterization (almost)
- Blue evaluated using rasterization or raytracing

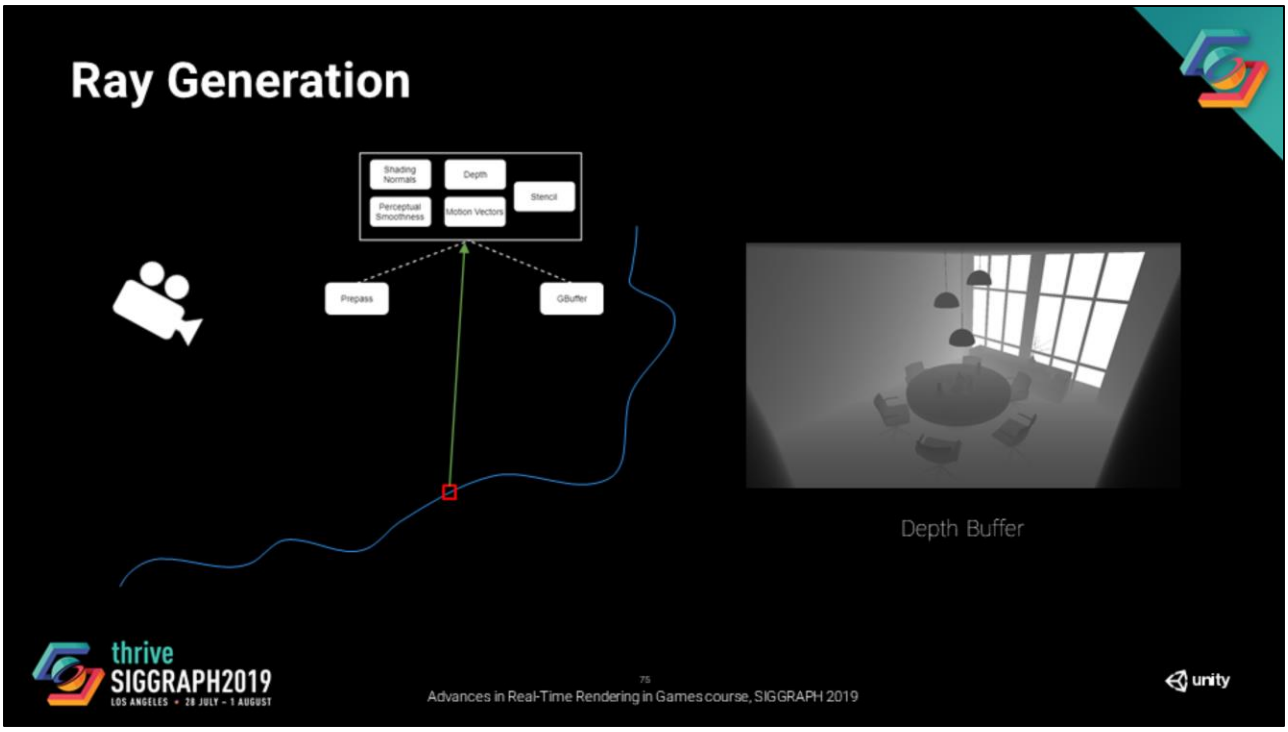
$$L_{indirect}(\mathbf{x}, \omega_o, \lambda, t) = L_{specular}(\mathbf{x}, \omega_o, \lambda, t) + L_{diffuse}(\mathbf{x}, \omega_o, \lambda, t)$$



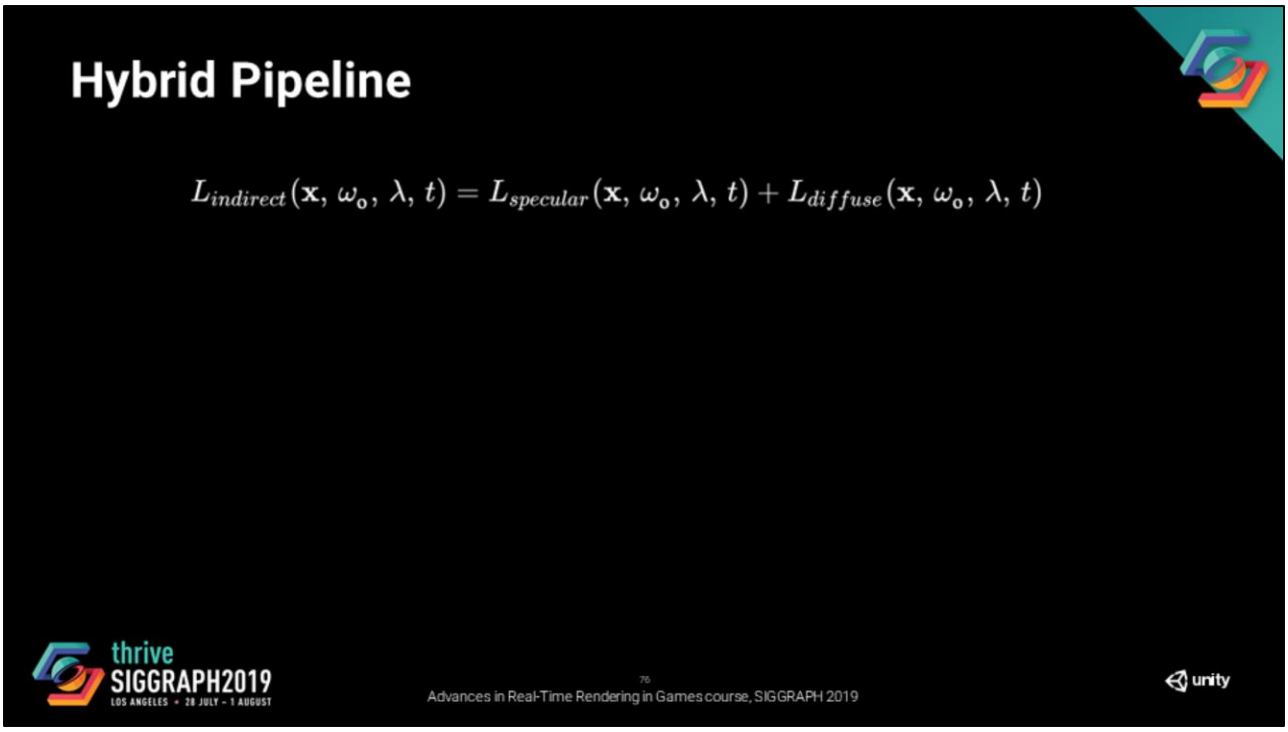
Let’s focus on indirect lighting first. We can decompose it as the sum of indirect specular and indirect diffuse lighting.




To evaluate those terms, we start from the depth buffer.




For every pixel on the screen, the guaranteed information that we have independetly of our lighting model is shading normal, smoothness, depth, motion vector, and stencil. These are the shared screen space buffers between the prepass and the gbuffer pass.




However, these two signals are dependent on the full material we are trying to evaluate.

# Hybrid Pipeline

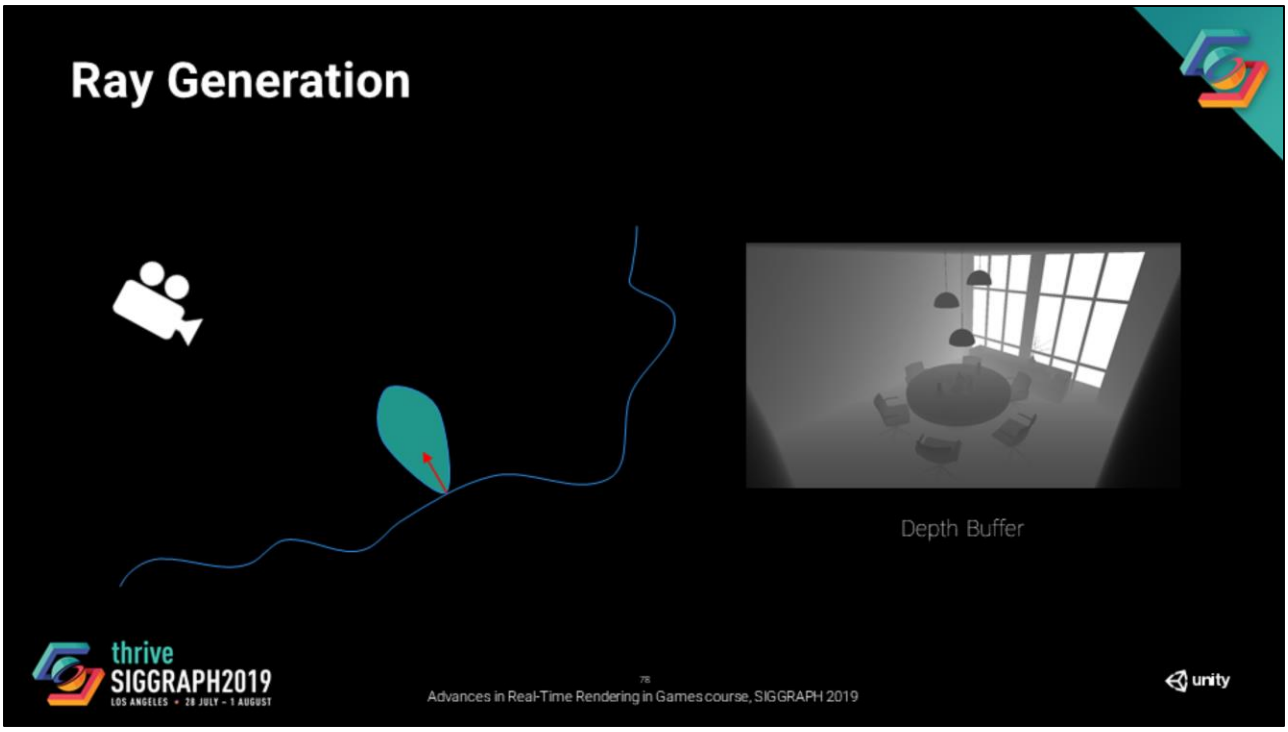
$$L_{indirect}(\mathbf{x}, \omega_o, \lambda, t) = L_{specular}(\mathbf{x}, \omega_o, \lambda, t) + L_{diffuse}(\mathbf{x}, \omega_o, \lambda, t)$$

$$L_{\text{indirect specular}} \approx \int_\Omega \frac{F\,G\,D}{4(\omega_i \cdot \mathbf{n})(\mathbf{v} \cdot \mathbf{n})} L_{\text{indirect}}(\omega_i \cdot \mathbf{n})\, d\omega_i \qquad L_{\text{indirect diffuse}} \approx \int_\Omega \frac{\rho}{\pi} L_{\text{indirect}}(\omega_i \cdot \mathbf{n})\, d\omega_i$$

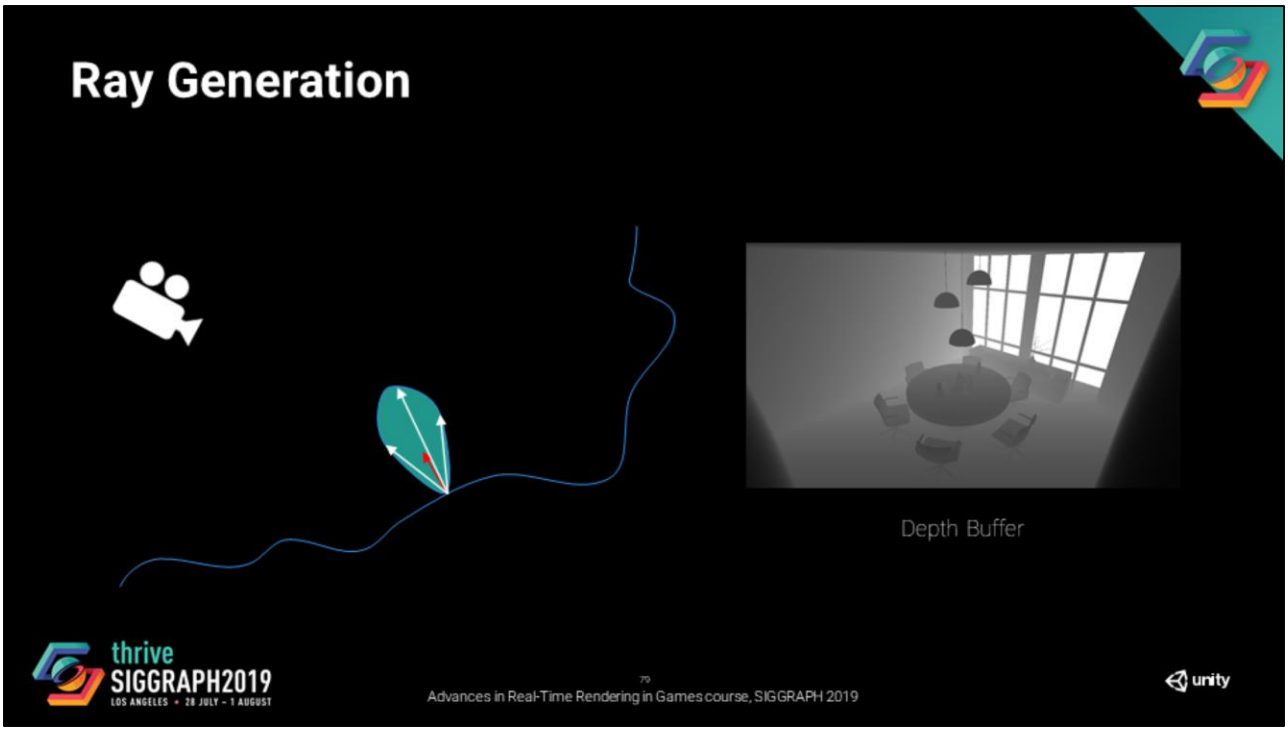
- Indirect specular is approximated as an Isotropic GGX lobe
- Indirect diffuse is approximates as a Lambert lobe
- Ambient occlusion affect indirect diffuse



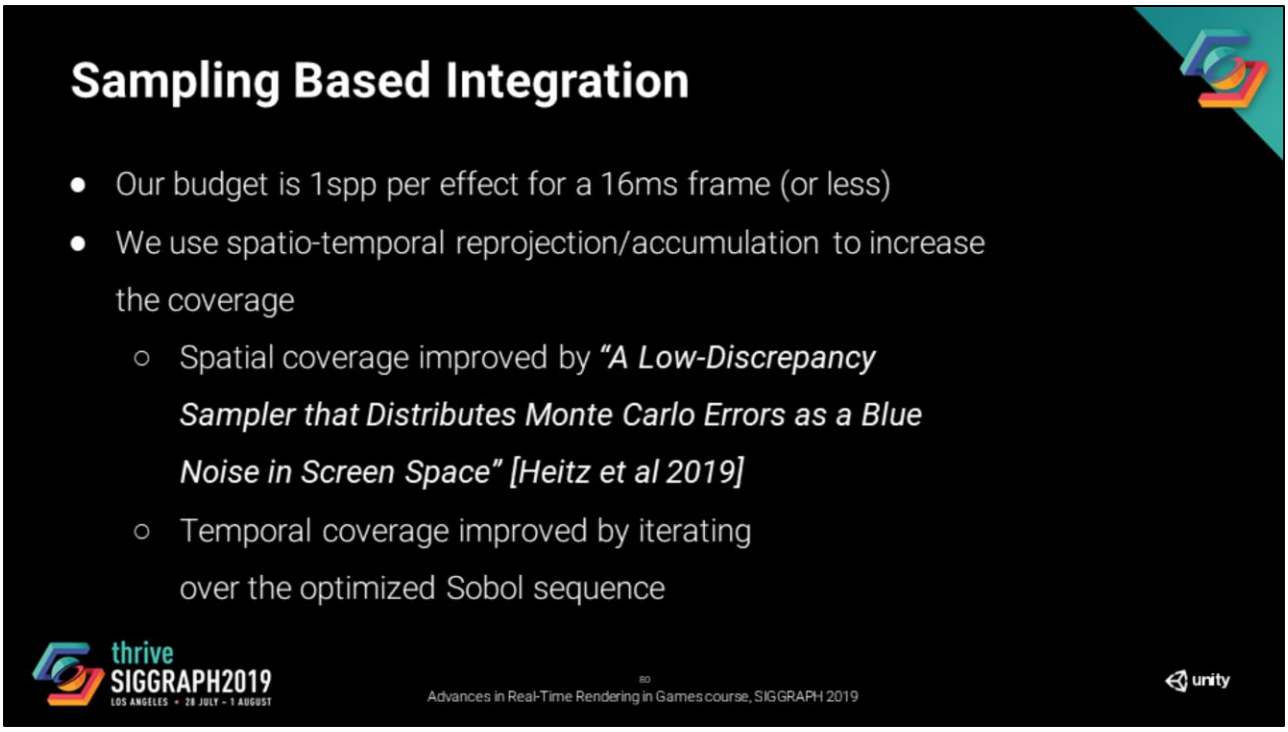
Given the fairly small amount of information we have, we decided to approximate the indirect specular as an isotropic ggx lobel, and indirect diffuse as a lambert lobe that will allow us to evaluate the lighting with only the screen space data that we have.




Then, based on that information and on the signal that we are trying to evaluate, we define a lobe and thus an important sampling profile.




Then the next step is to generate a set of directions to evaluate the lobe.




First, let's define precisely how we are going to generate our samples.

Let's suppose our budget is one sample per pixel per effect per frame (if we target 16 millisecond frames). In reality, our budget is lower than that, but let's assume that is what we will have.

If you consider a single sample by its own, there is no notion of coverage.

The idea is to try to maximize the coverage by accumulating the signal over time but also to make sure that neighboring pixels have a different coverage of the sample space so when we filter, we end up with enough information to produce a stable signal over the frames.

To do that, we use the recent paper

# Sampling Based Integration

```
Buffer<int> _OwenScrambledSequence;
Buffer<int> _ScramblingTile;
Buffer<int> _RankingTile;

float OptimizedBlueNoise8SPP(uint2 pixel, int sampleIndex, int sampleDimension)
{
    // Wrap args
    pixel = pixel & 127;
    sampleIndex = sampleIndex & 8;
    sampleDimension = sampleDimension & 8;

    // xor index based on optimized ranking
    int rankedSampleIndex = sampleIndex ^ _RankingTile[sampleDimension + (pixel.x + pixel.y * 128) * 8];

    // fetch value in sequence
    int value = _OwenScrambledSequence[sampleDimension + rankedSampleIndex * 256];

    // If the dimension is optimized, xor sequence value based on optimized scrambling
    value = value ^ _ScramblingTile[sampleDimension + (pixel.x + pixel.y * 128) * 8];

    // convert to float and return
    float v = (0.5f + value) / 256.0f;

    return v;
}
```

Owen Scrambled Sobol Sequence

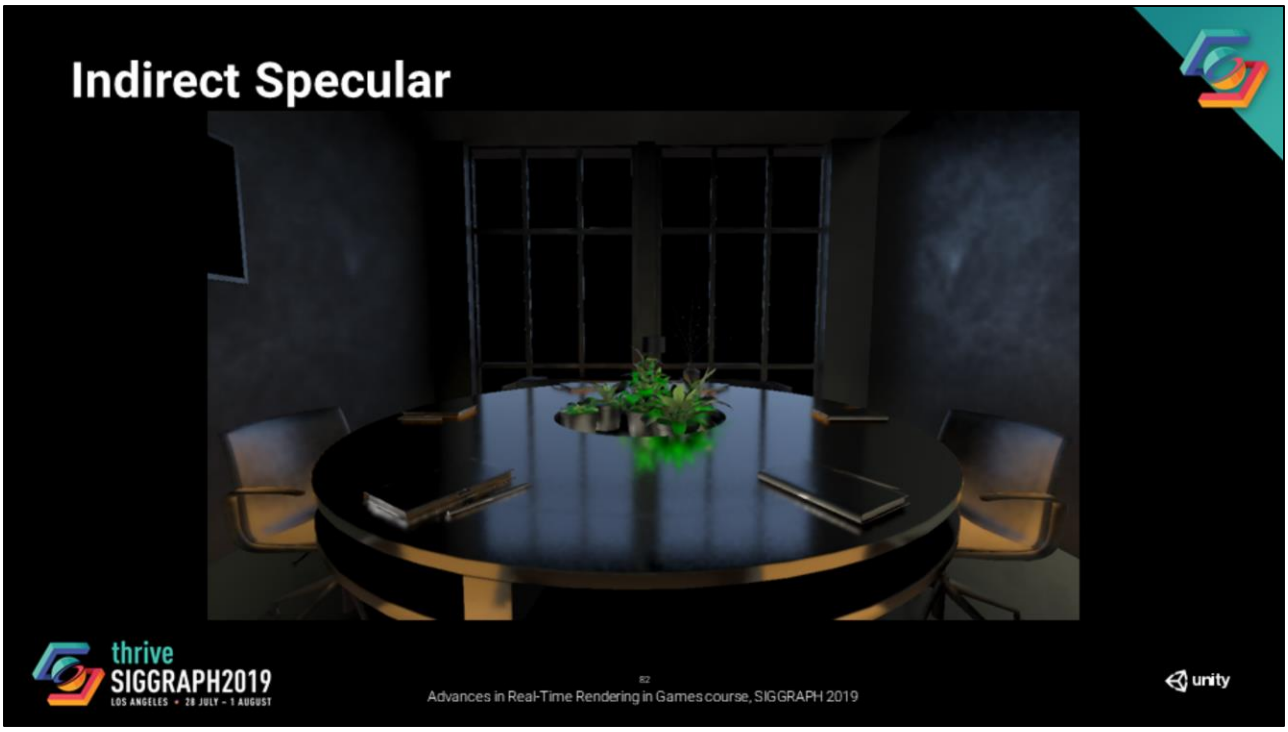
https://eheitzresearch.wordpress.com/762-2/



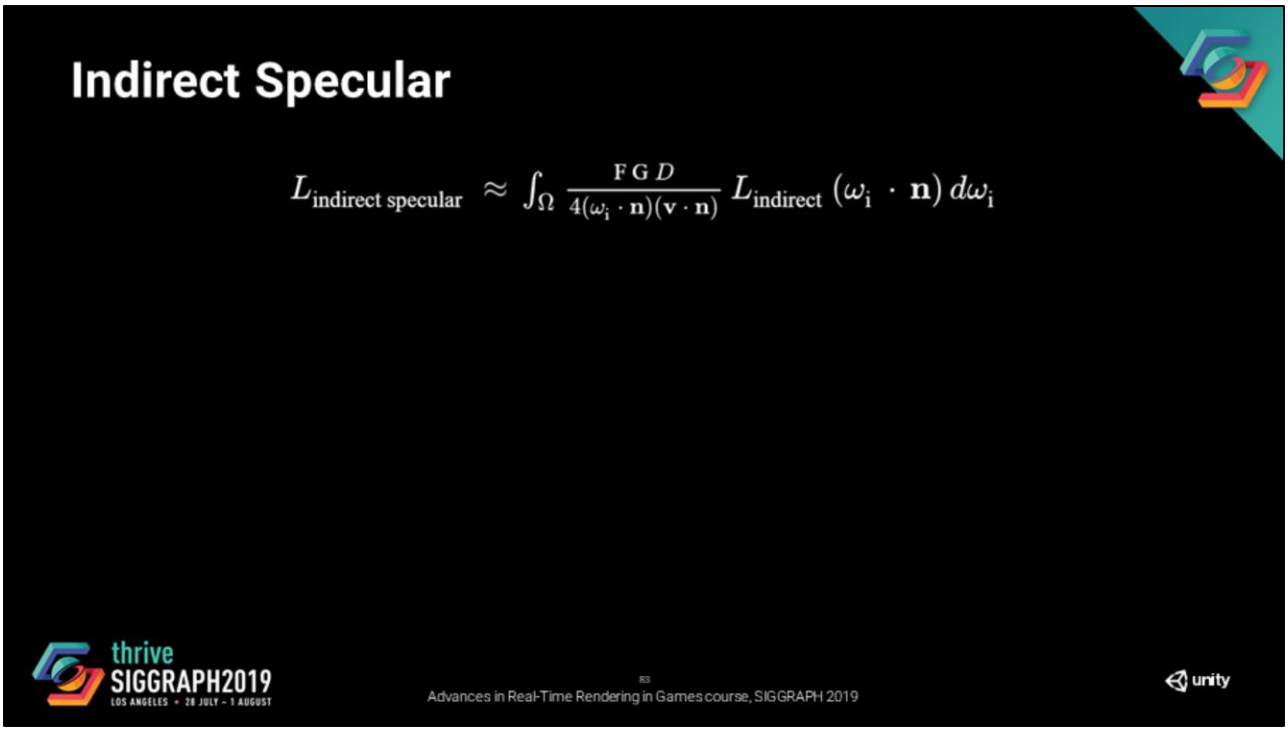
In practice this is what the code looks like. And this is what we use for all the effects
If we target a convergence time of 8 frames (which is what TAA does), so we use the 8spp version of the algorithm.

Pixel is the screen coordinate of the current pixel.
Sample index is basically the frame index modulo 8
And sample dimension is (0, 1) for the indirect samples and more for additional dimensions

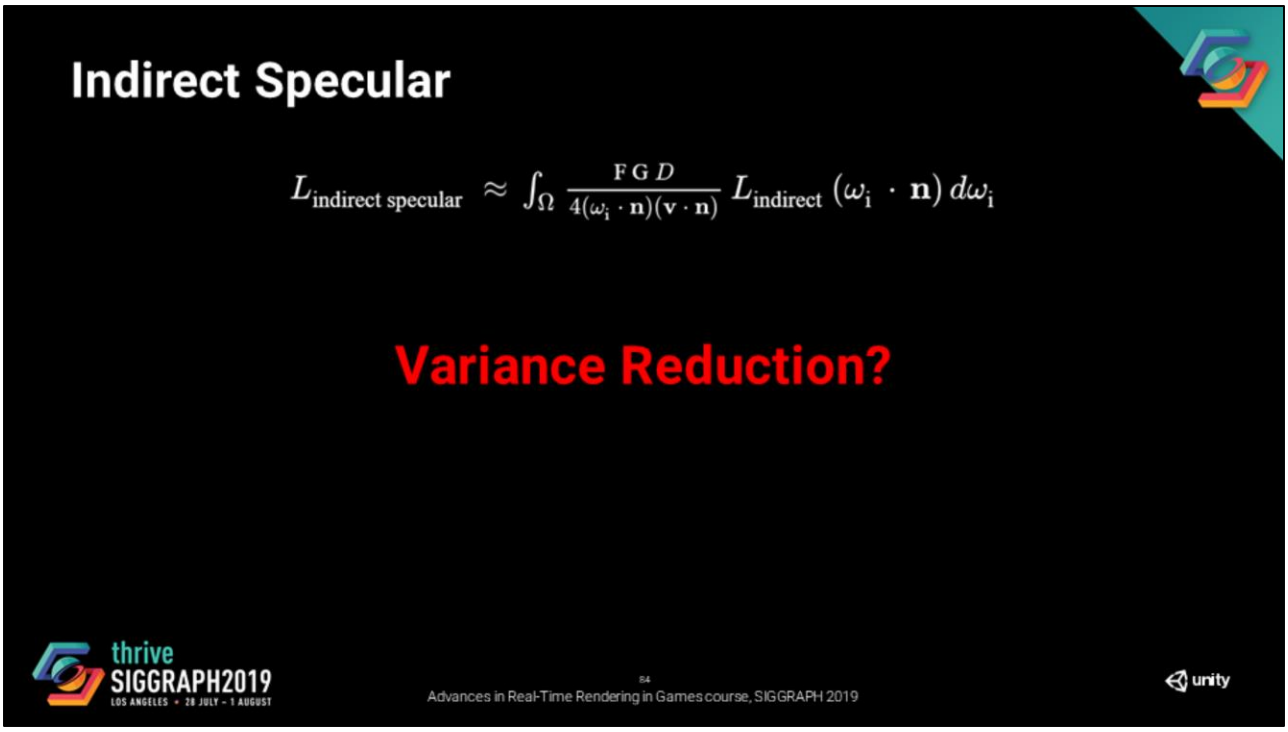
The full code and data can be found on Eric Heitz's website here:
https://eheitzresearch.wordpress.com/762-2/



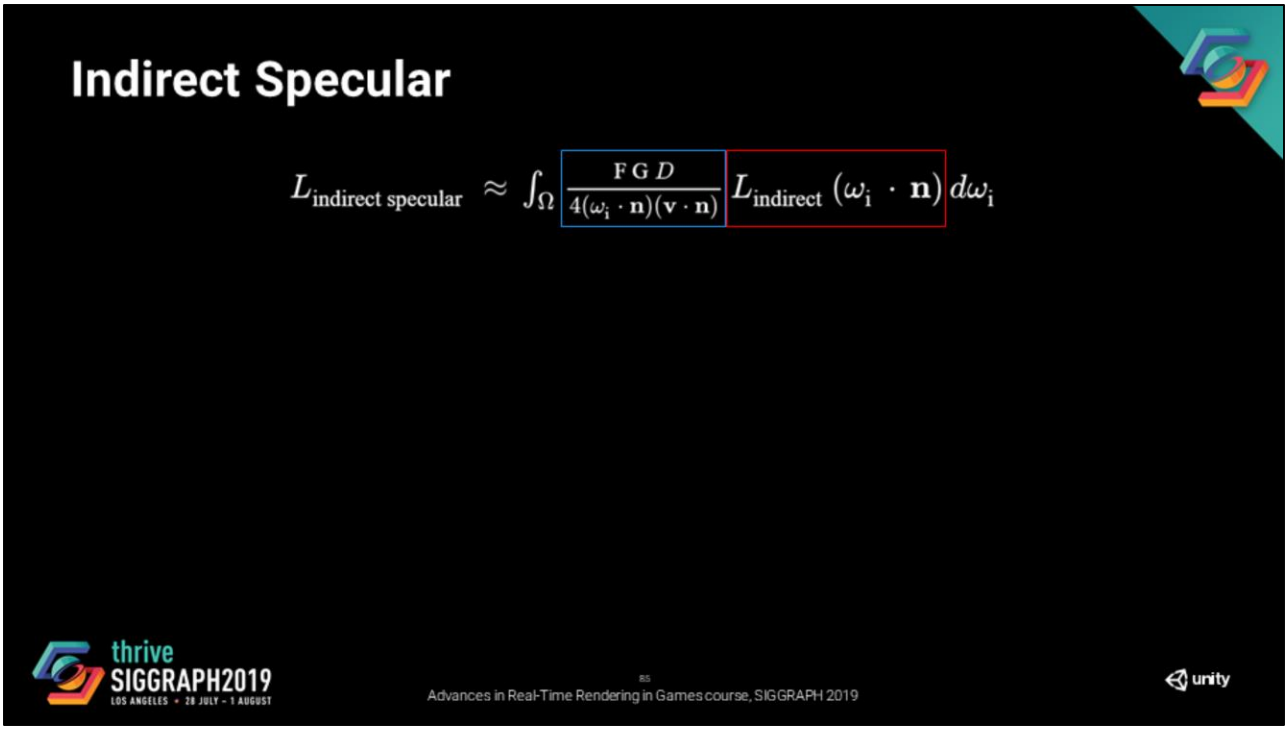

Now that we are able to generate our samples, let’s focus on the first effect that I wanted to talk about: indirect specular.




As mentioned before, we start from the approximated version that we settled for. One option is to go for the full integral.




But the question we want to ask ourselves: is there is a way to do variance reduction, achieve faster convergence with less samples and thus have less to denoise.




The approach we have settled for here is the split sum approximation by karis

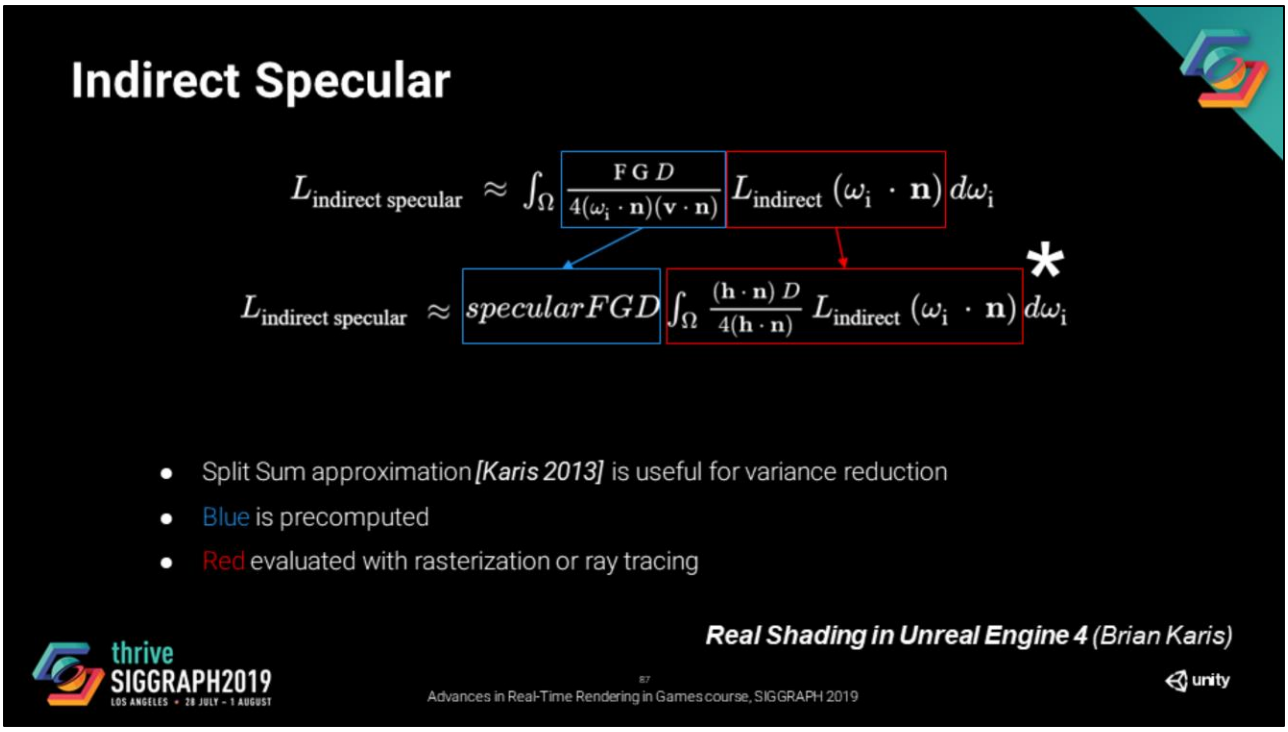
# Indirect Specular

$$L_{\text{indirect specular}} \approx \int_{\Omega} \frac{\mathrm{F}\,\mathrm{G}\,D}{4(\omega_i \cdot \mathbf{n})(\mathbf{v} \cdot \mathbf{n})} L_{\text{indirect}}(\omega_i \cdot \mathbf{n})\, d\omega_i$$

$$L_{\text{indirect specular}} \approx specularFGD \int_{\Omega} \frac{(\mathbf{h} \cdot \mathbf{n})\, D}{4(\mathbf{h} \cdot \mathbf{n})} L_{\text{indirect}}(\omega_i \cdot \mathbf{n})\, d\omega_i \quad *$$



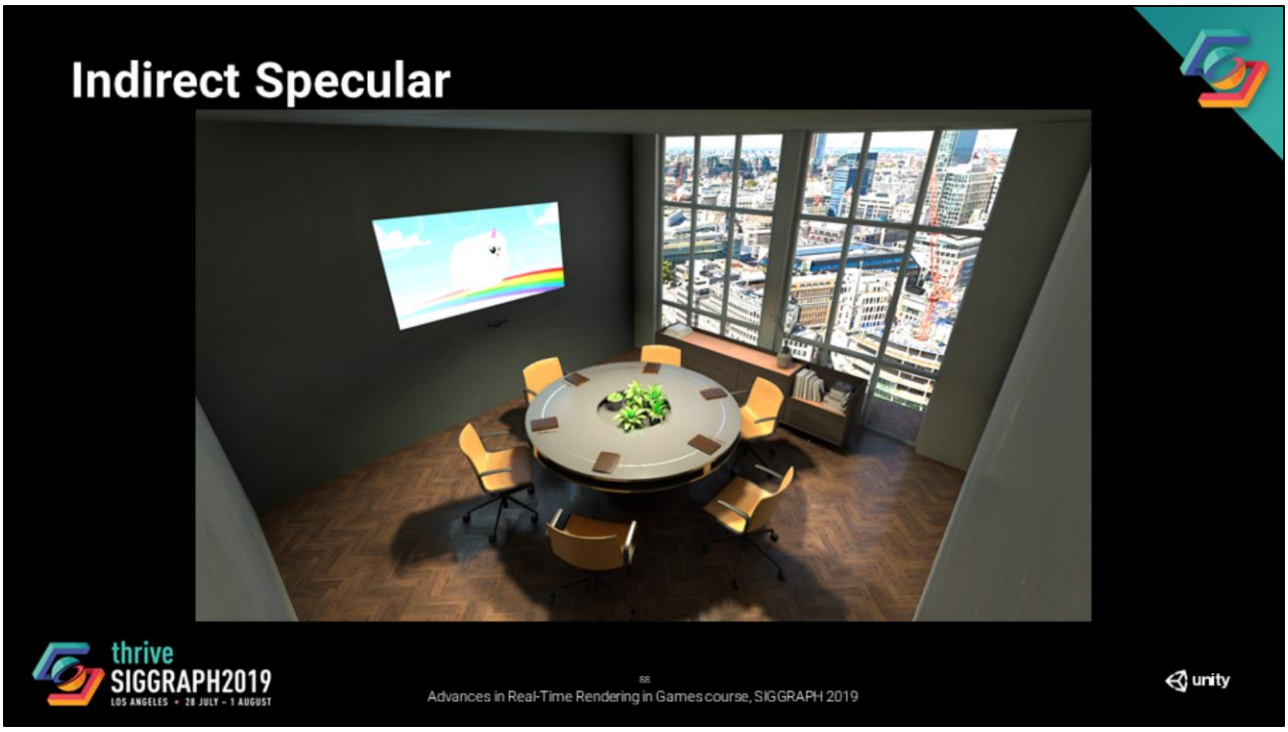
The equation is decomposed as two terms



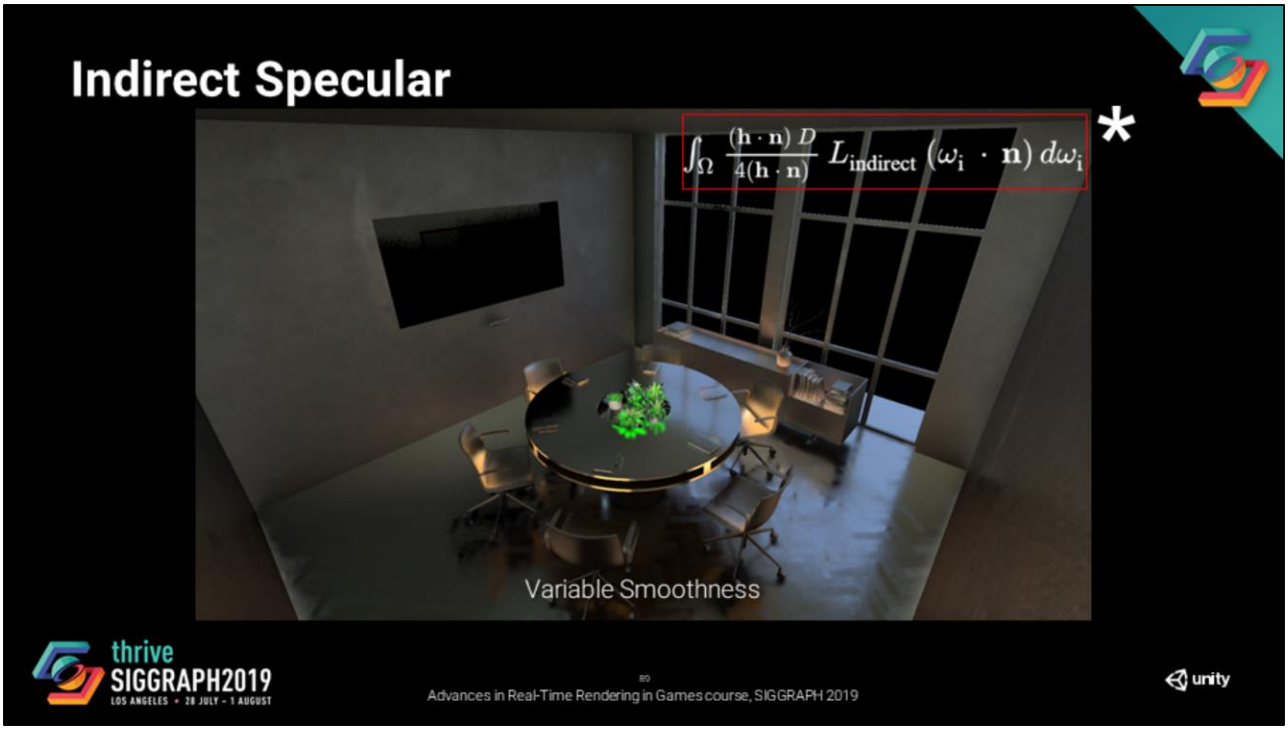

The blue term is precomputed and stored in a texture.

And the red term is evaluated with rasterizarion techniques or ray tracing.

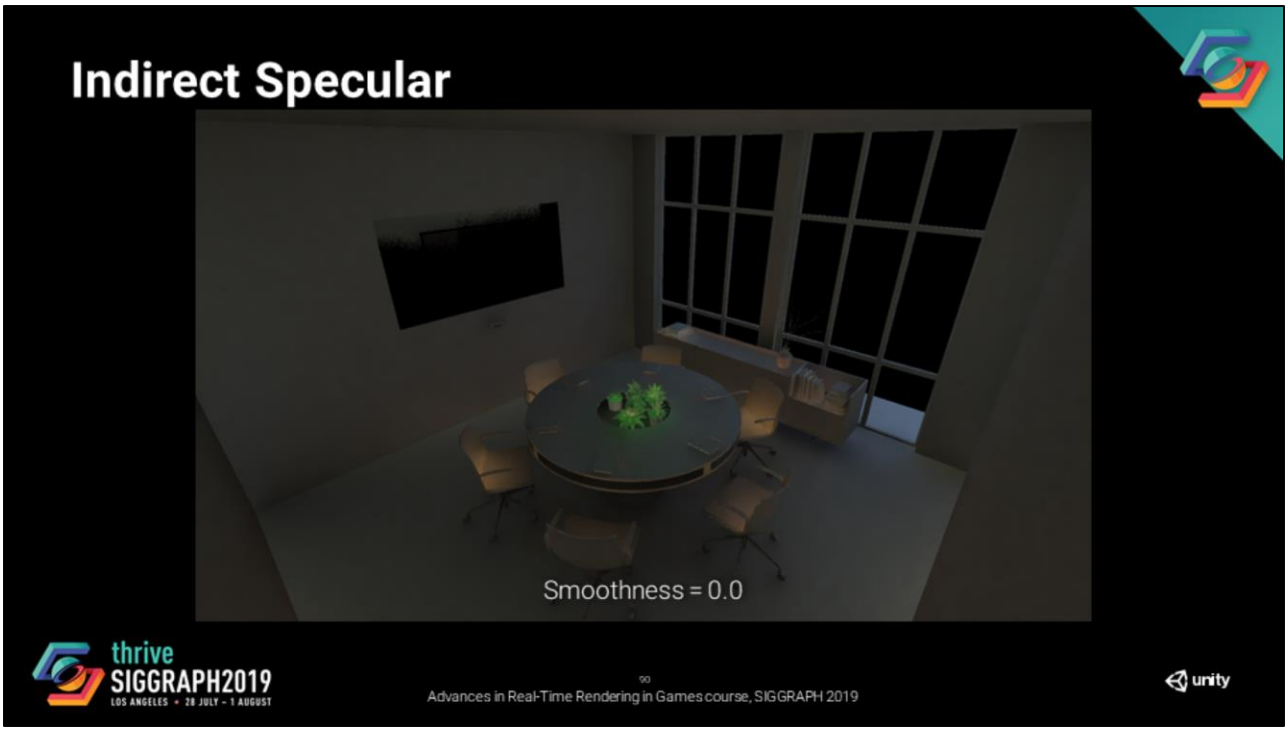
This main problem with this approximation is that it implies a biased estimator. However, the bias of this approach has been evaluated as acceptable (this is already what we do for rasterization).



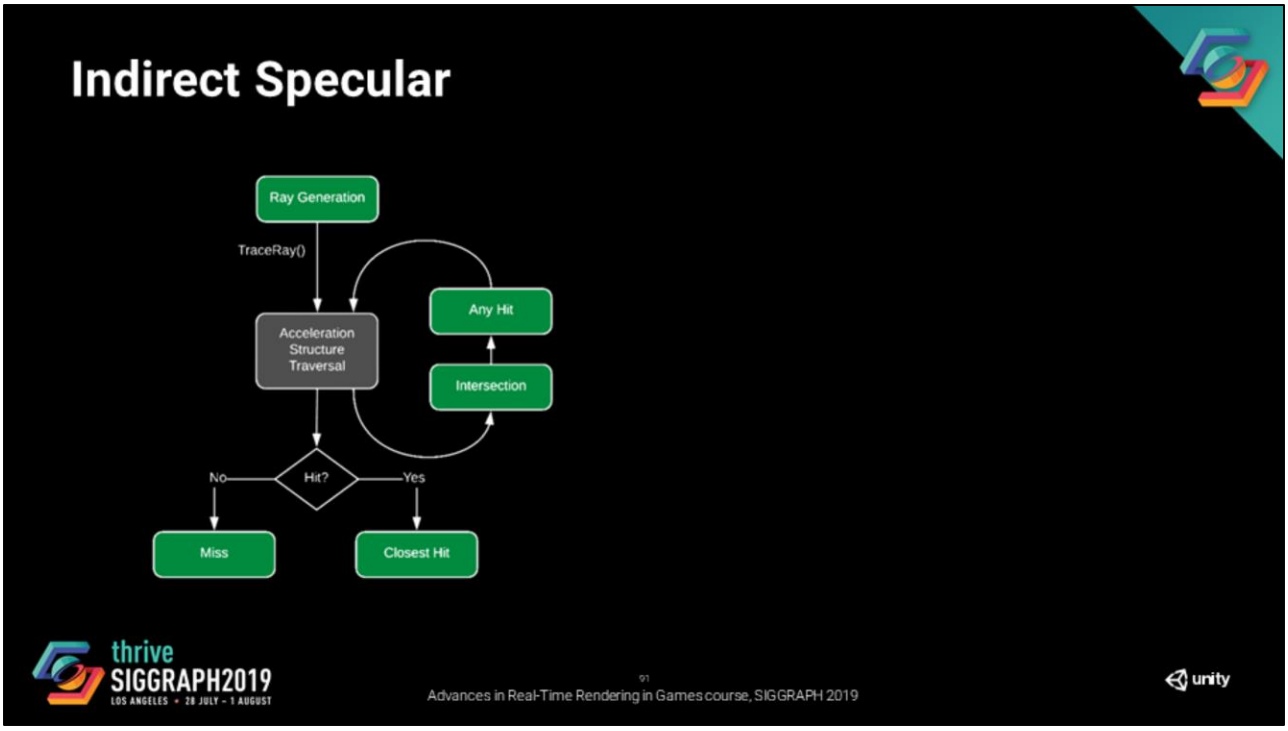

Let’s start with a final render of the scene




This is what the red term looks like. This is not a bad configuration, most surface are kind of smooth, rays will thus go roughly in the same direction

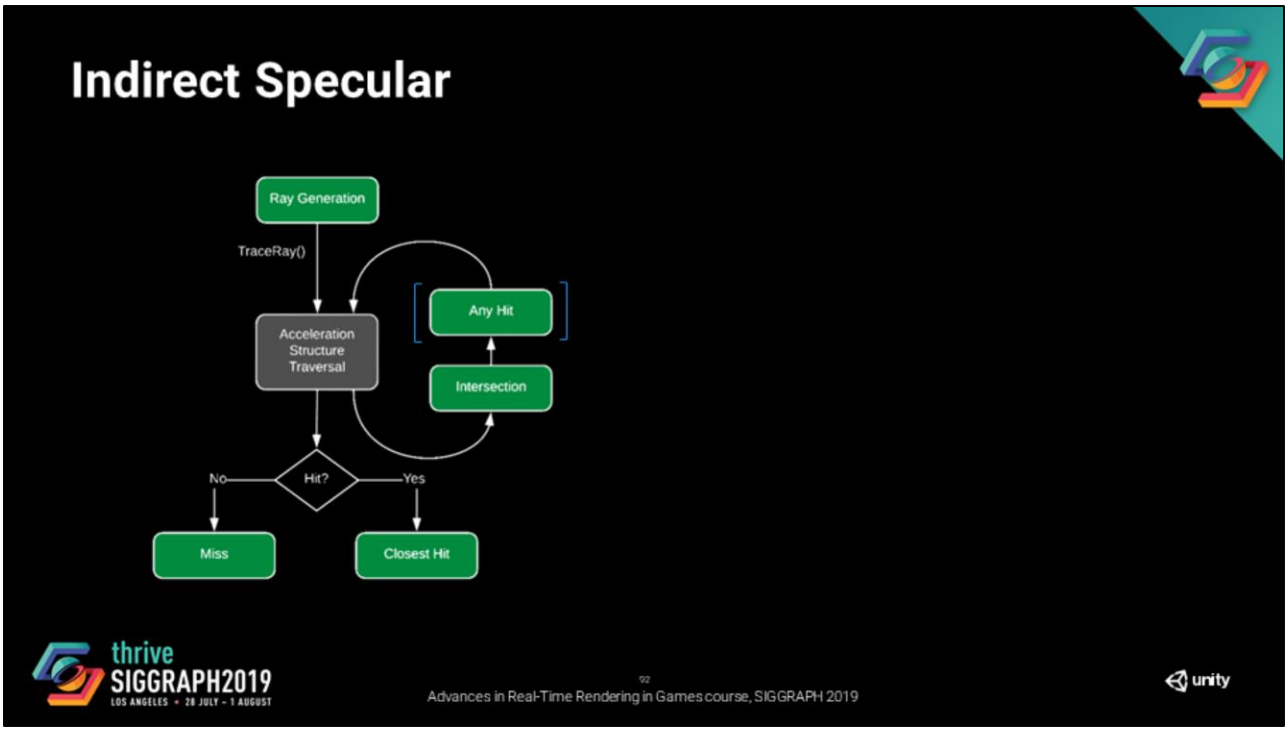



Just to make things a bit more challenging performance-wise, let’s force the smoothness of everything to be 0.0 so that rays are completely incoherent.



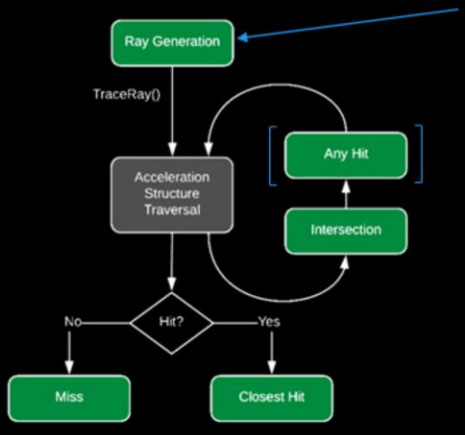

Everything is solved right? (no), let’s implement reflections!

If you remember correctly in the dxr introduction, i showed this diagram.




So the first important thing to point out is that we want to avoid using any hit shaders as much as possible. Given that they are triggered for every potential intersection, they have a huge impact on the execution time of ray traversal.

# Indirect Specular



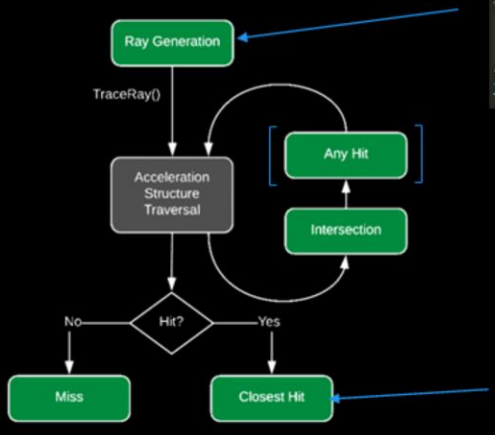

```
for(int sampleIdx = 0; sampleIdx < numSamples; ++sampleIdx)
{
    newRay = GenerateRay();
    TraceRay(newRay);
    totalLighting += newRay.lighting;
}
finalColor = totalLighting / (numSamples);
```



Ray generation shader is the entry point of our evaluation. For every pixel in our frame, we loop through the number of samples (more than one for the sake of the exercise).

Based on that sample index, and the pseudo random number generator that we defined earlier, we generate a new direction (in this presentation I am not covering importance sampling).

# Indirect Specular

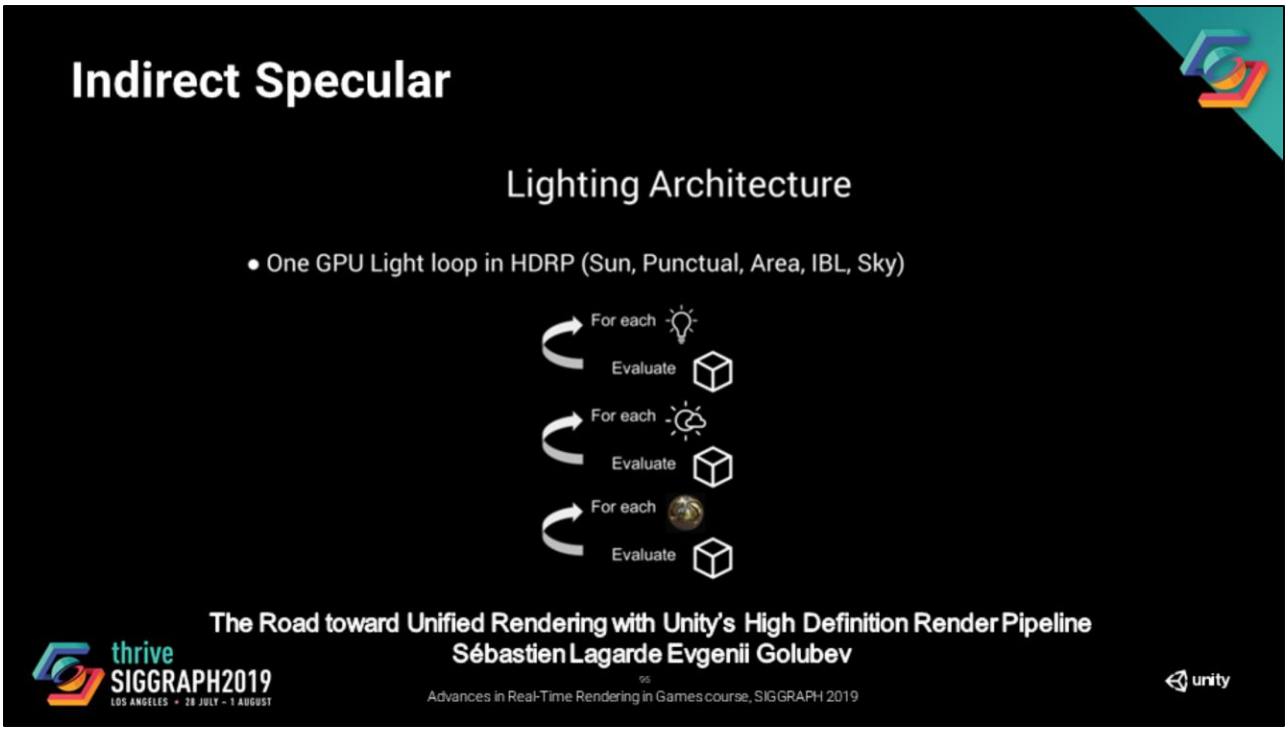



```
for(int sampleIdx = 0; sampleIdx < numSamples; ++sampleIdx)
{
    newRay = GenerateRay();
    TraceRay(newRay);
    totalLighting += newRay.lighting;
}
finalColor = totalLighting / (numSamples);
```



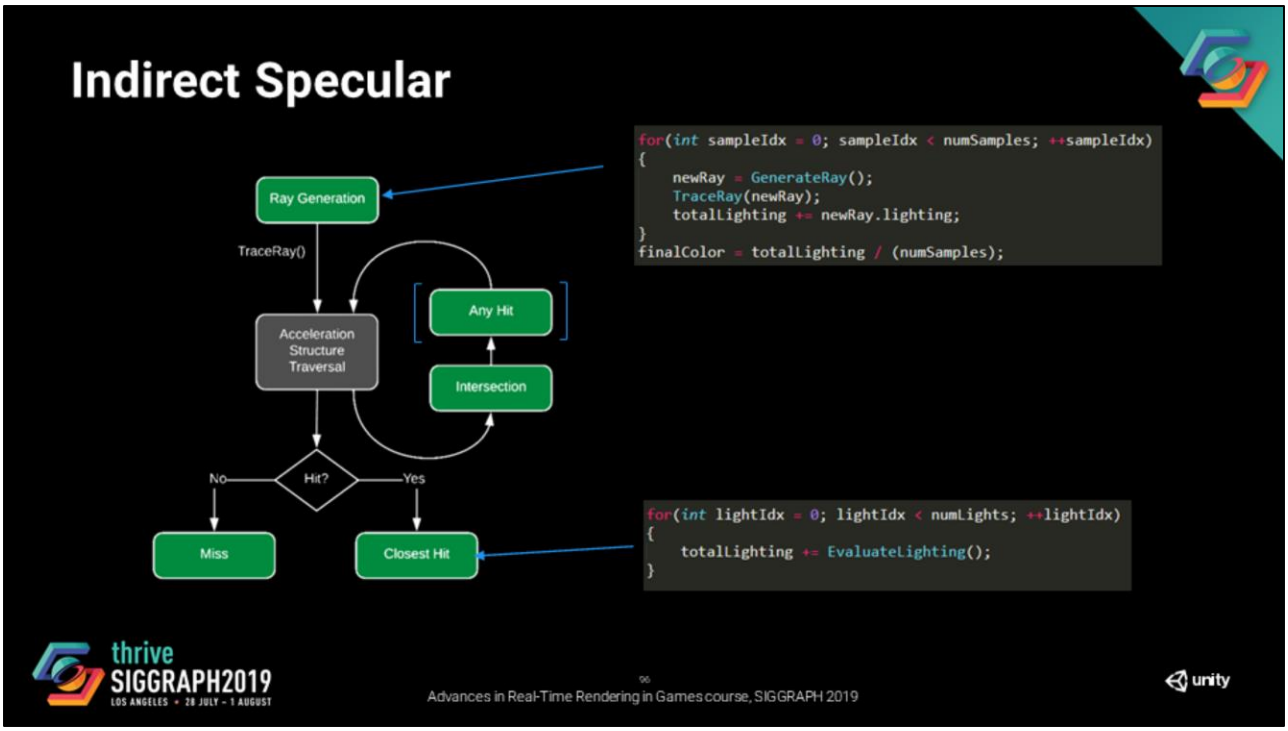
We then trace a ray in our scene and we get a closest intersection (or none, but then we just pick the radiance of the sky returned by the miss shader).

At this point we need to evaluate the lighting at the intersection point



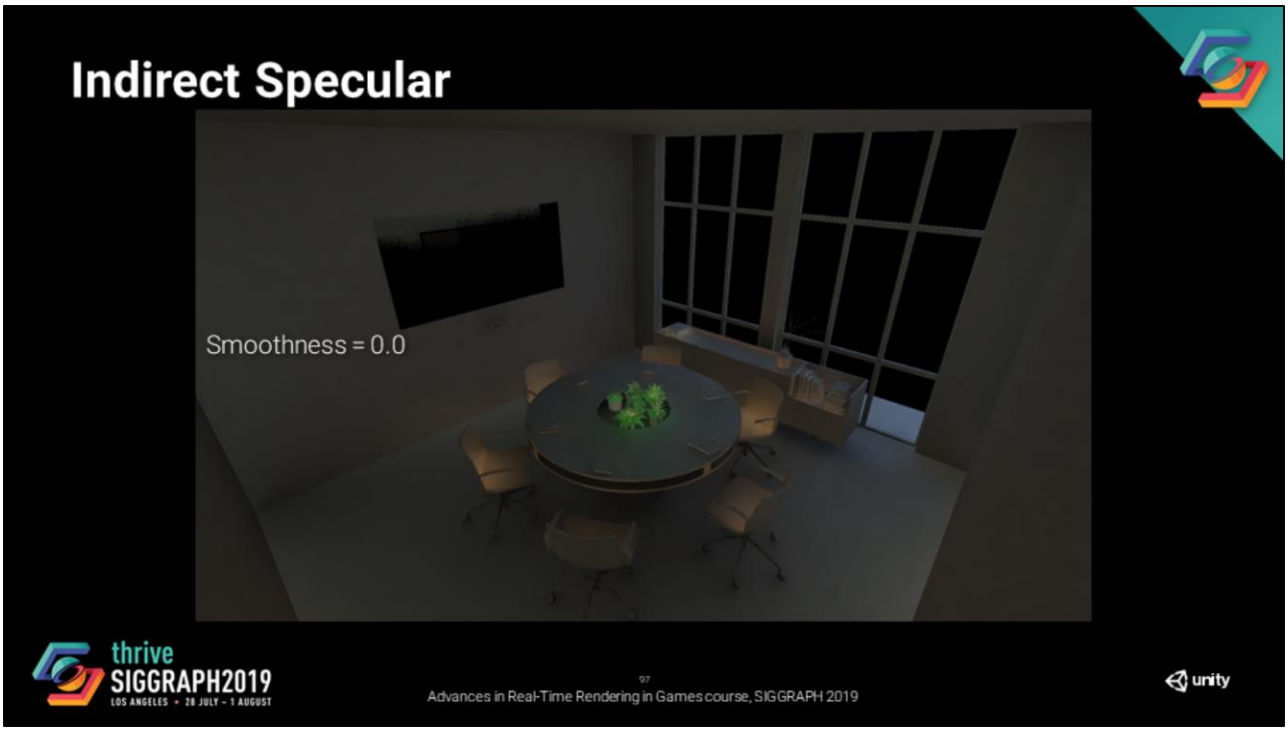

Given that we have a valid cluster structure, we can run our lightloop. The shorter the lightloop is, the better, but just in case the user wants to have full lighting, he can run all of it.

An interesting point to raise here is that we are not doing a next event estimation based approach. It is probably something we should explore, but it will introduce additional variance into our integration and we already have very few samples.




So we loop through the set of lights that were fetched thanks the light cluster, the lighting is accumulated and returned to the ray generation shader.

At that point, we accumulate the lighting of this ray, normalize the result and the result shown in the screen shot.



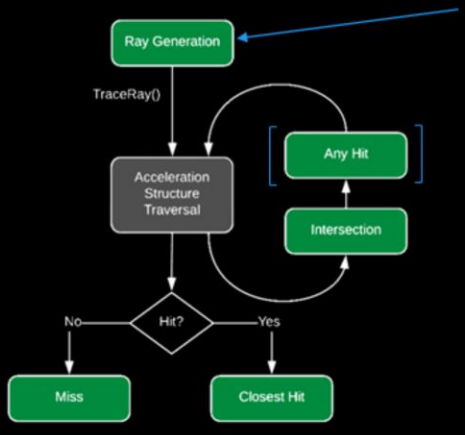

Great, we are done, right?

There is a small problem.


It is very very expensive and not viable for anything close to real-time.
This is happening because we are messing with the scheduling of ray bvh traversal.


We thus need to try couple things to damage control this.

# Indirect Specular



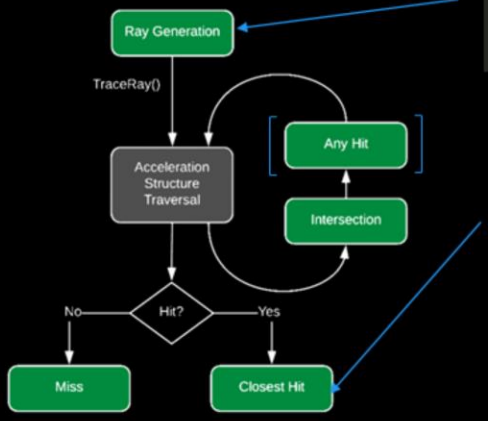

```
for(int sampleIdx = 0; sampleIdx < numSamples; ++sampleIdx)
{
    newRay = GenerateRay();
    TraceRay(newRay);
    totalLighting += LightLoop(newRay.gbuffer0, newRay.gbuffer1,...);
}
finalColor = totalLighting / (numSamples);
```



The first thing we can do is move the lightloop outside of the closest hit shader. The approach here is to avoid messing the scheduling of rays bvh traversal. So we need to reduce the complexity of our closest hit shader

We do pretty much the same thing as the previous slide. For every sample, we generate a ray and then trace that ray.

# Indirect Specular

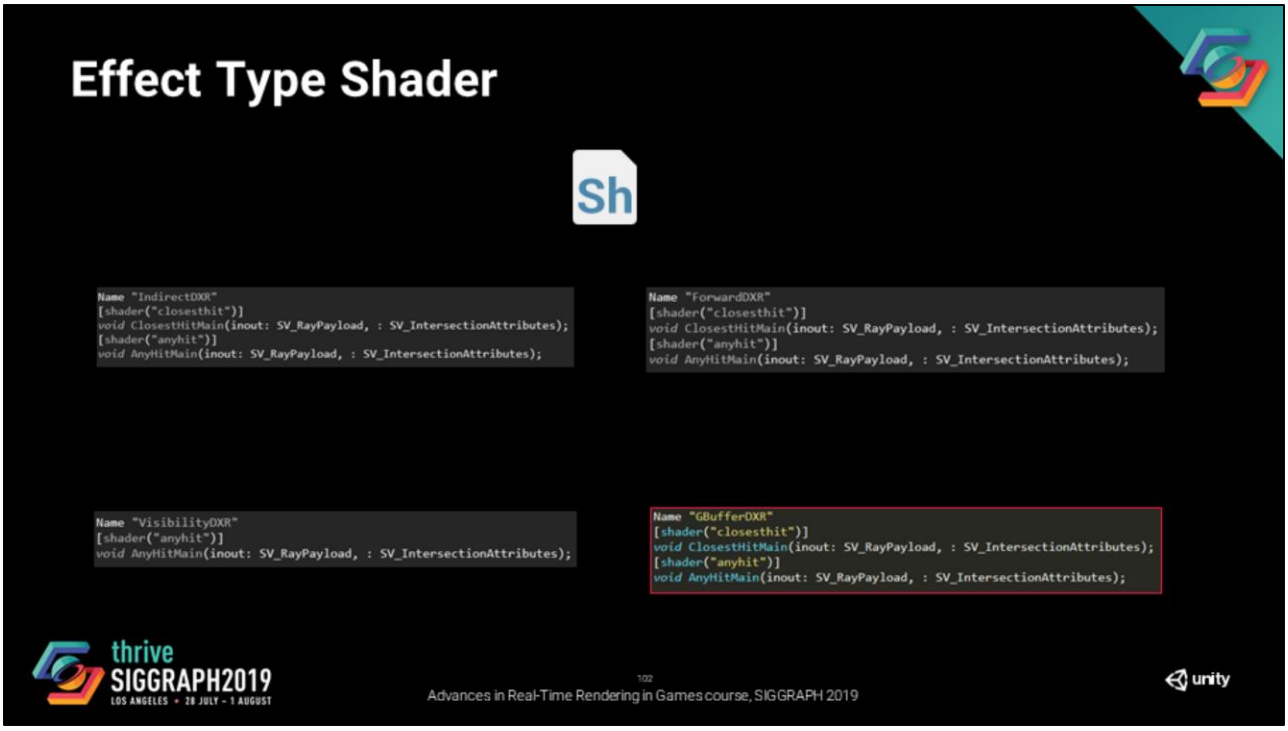



```
for(int sampleIdx = 0; sampleIdx < numSamples; ++sampleIdx)
{
    newRay = GenerateRay();
    TraceRay(newRay);
    totalLighting += LightLoop(newRay.gbuffer0, newRay.gbuffer1,...);
}
finalColor = totalLighting / (numSamples);
```

```
currentRay.gbuffer0 = ...
currentRay.gbuffer1 = ...
currentRay.gbuffer2 = ...
currentRay.gbuffer3 = ...
```

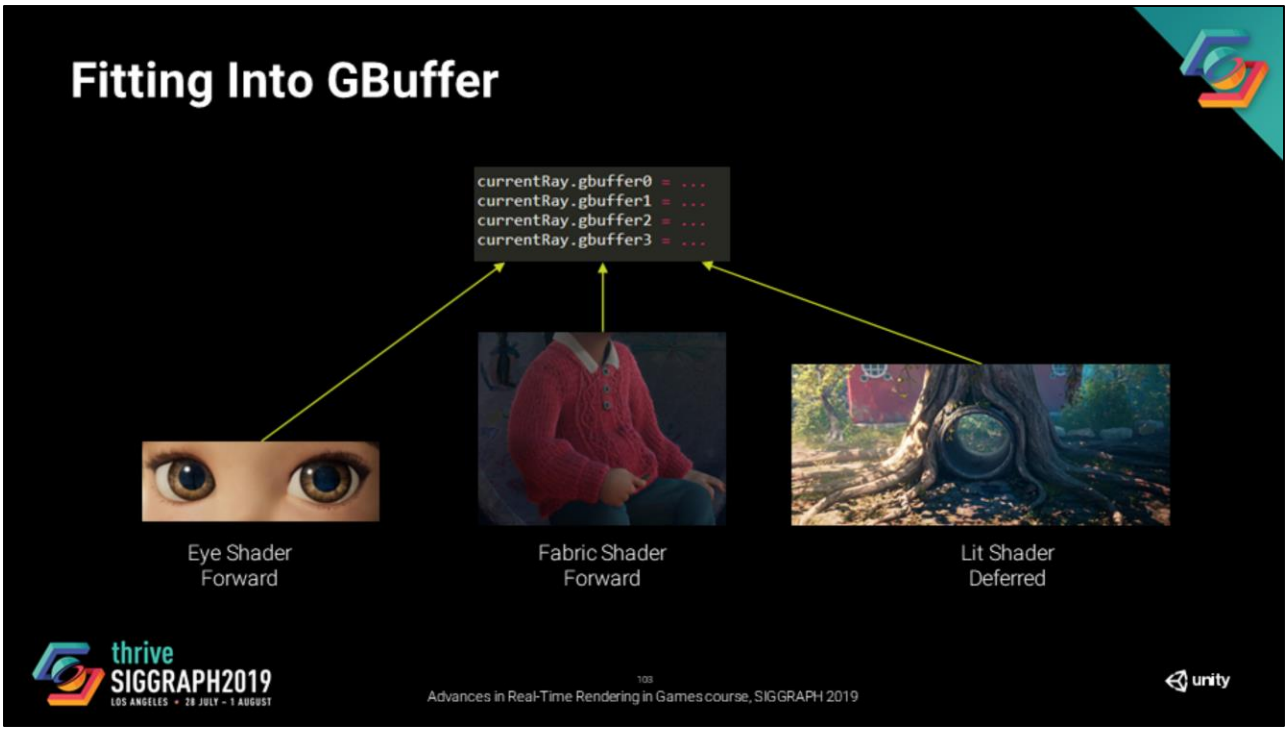


We then get our intersection and this time, instead of evaluating the lighting, we compress the material data into a gbuffer.




This is when that fourth effect type shader Gbuffer is going to be useful.

But this raises a couple of questions…

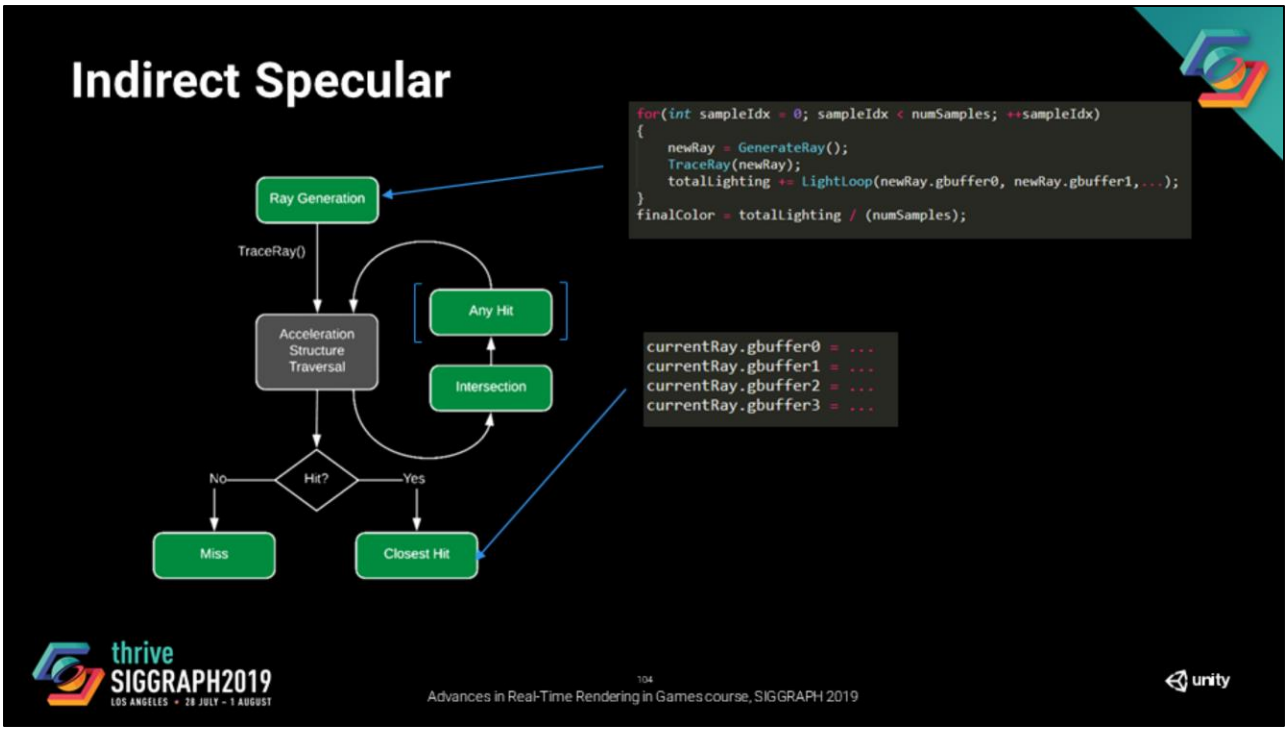



How do we manage the fact that we have multiple lighting models. Here are three examples: an eye shader, a fabric shader and our main Lit shader.

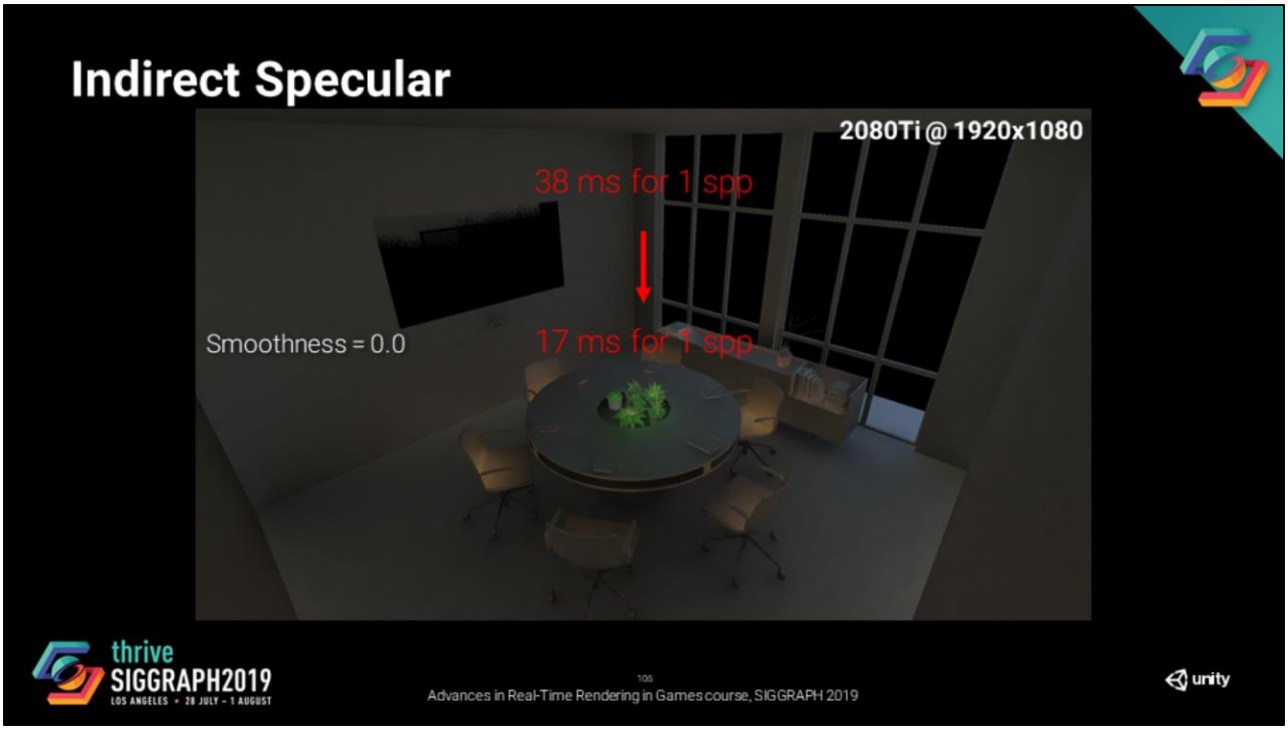
Currently we have settled for two options

- A performance mode where we have an automatic fit of all the lighting models into our main lighting model

- A quality mode where we do the lighting in the closest hit for the alternative lighting models and return it as a single channel of the gbuffer. The main lighting model data are still stored in the gbuffer.




Then we return this to the ray generation shader, where we run the lightloop like before, normalize and we are done, right?

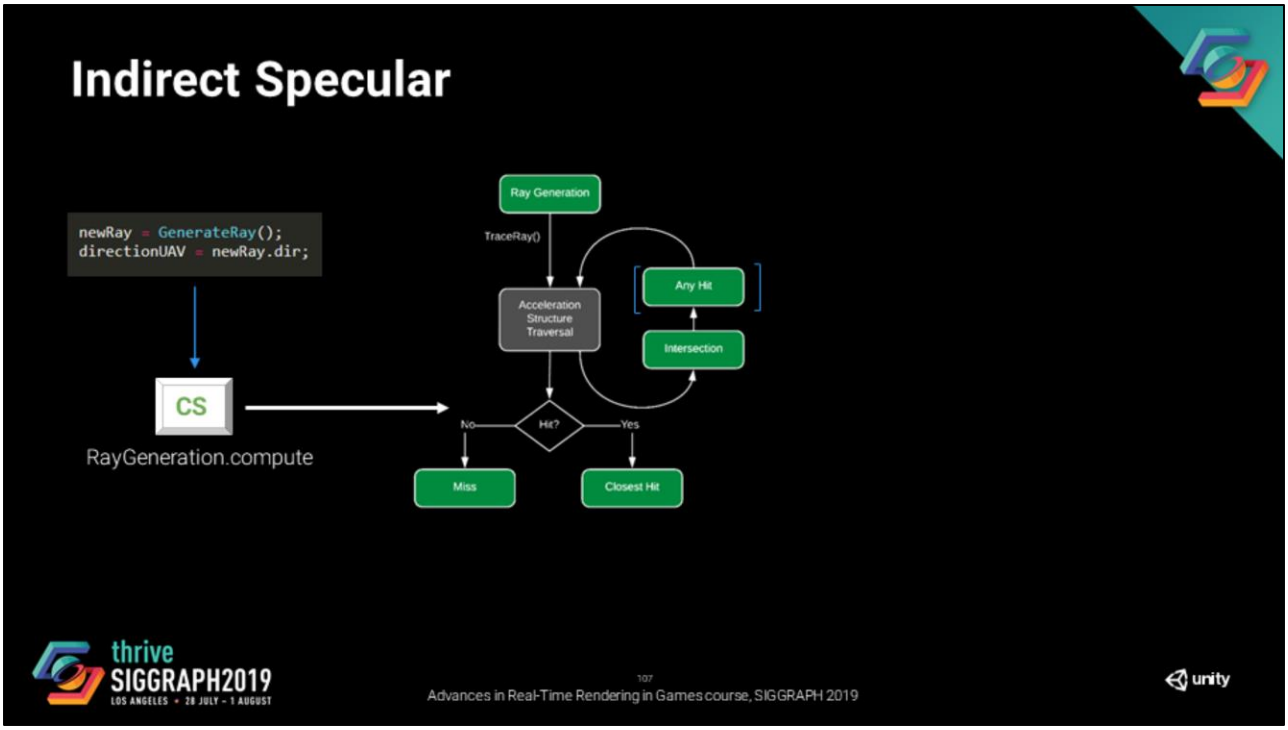



By doing this, we get better performance. It is still bad, but at least we are getting somewhere.

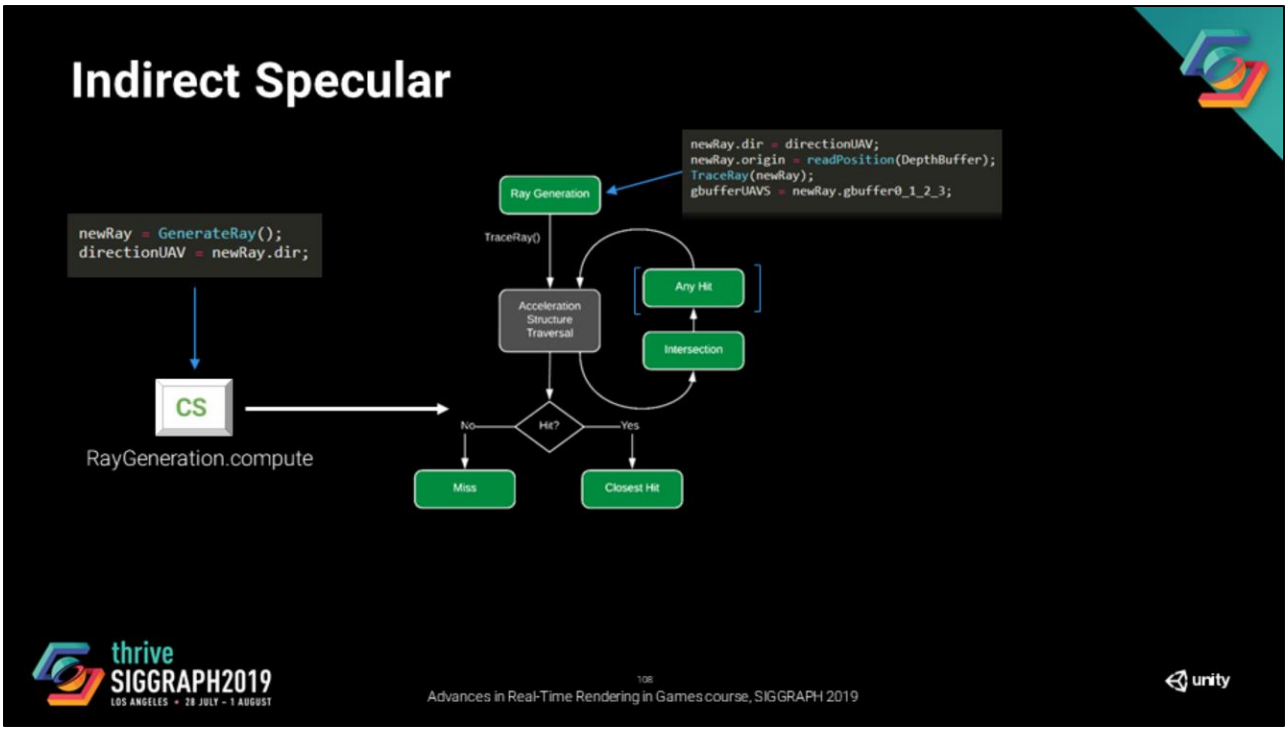
The intuition that we are getting from this is: the less we interfere with the ray traversal scheduling, the better it is.

Let’s continue following that track and see how it goes. Let’s offload as much as possible to compute shaders.


So now we have a compute shader that generate the directions of our rays. Directions are stored into a UAV.

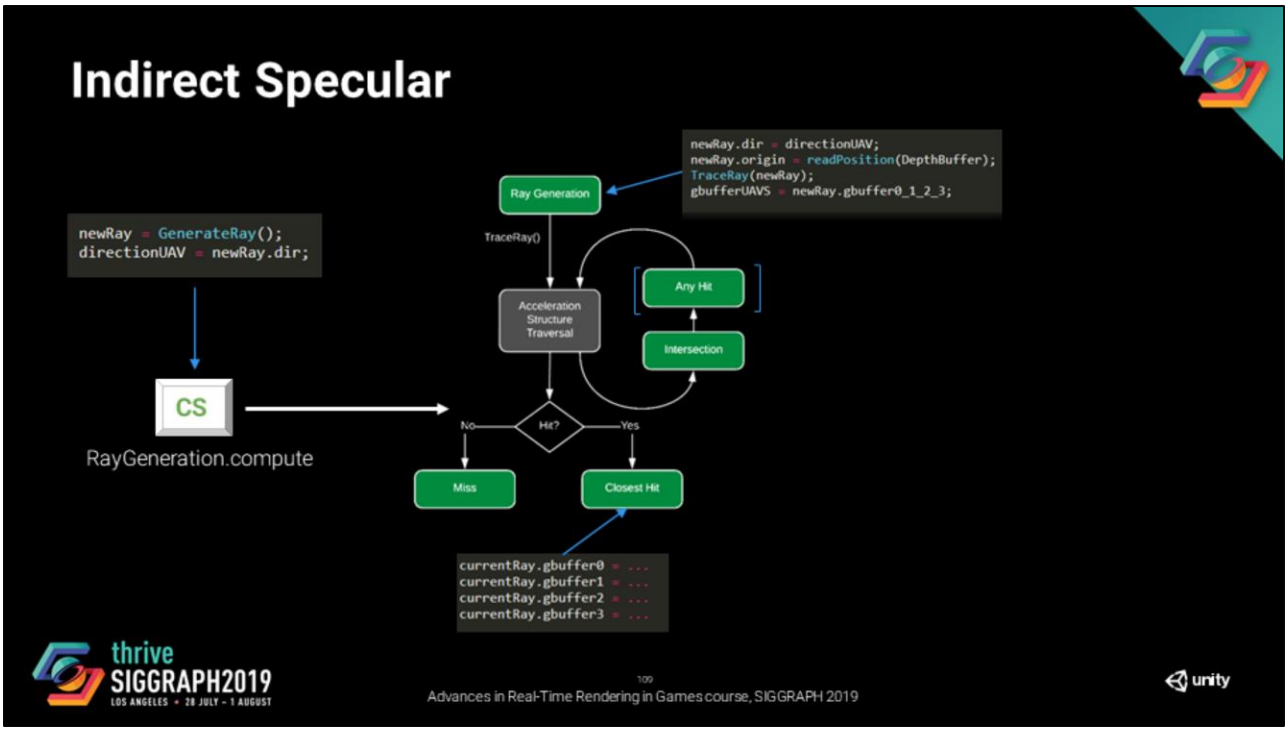



Then we call the raytracing shader pipeline.



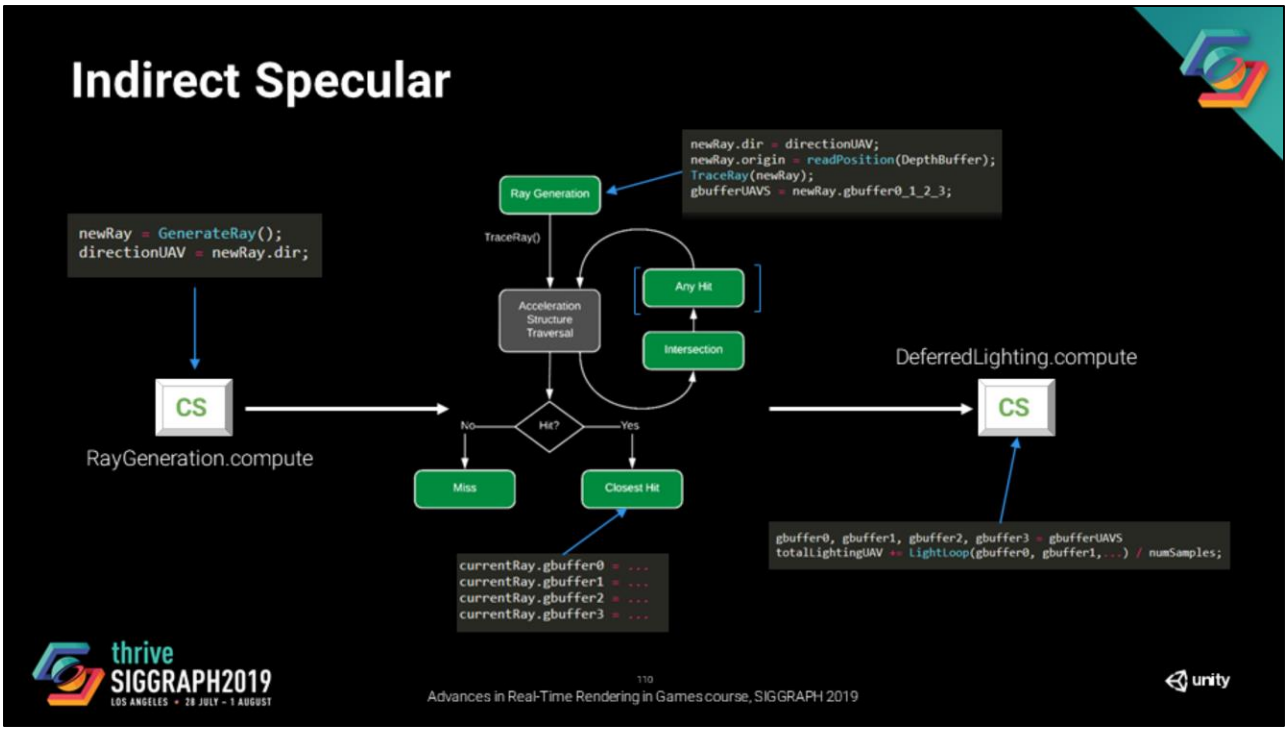

We read the directions and position from the depth buffer and diretion UAV, build our ray.

Trace the ray in the scene.



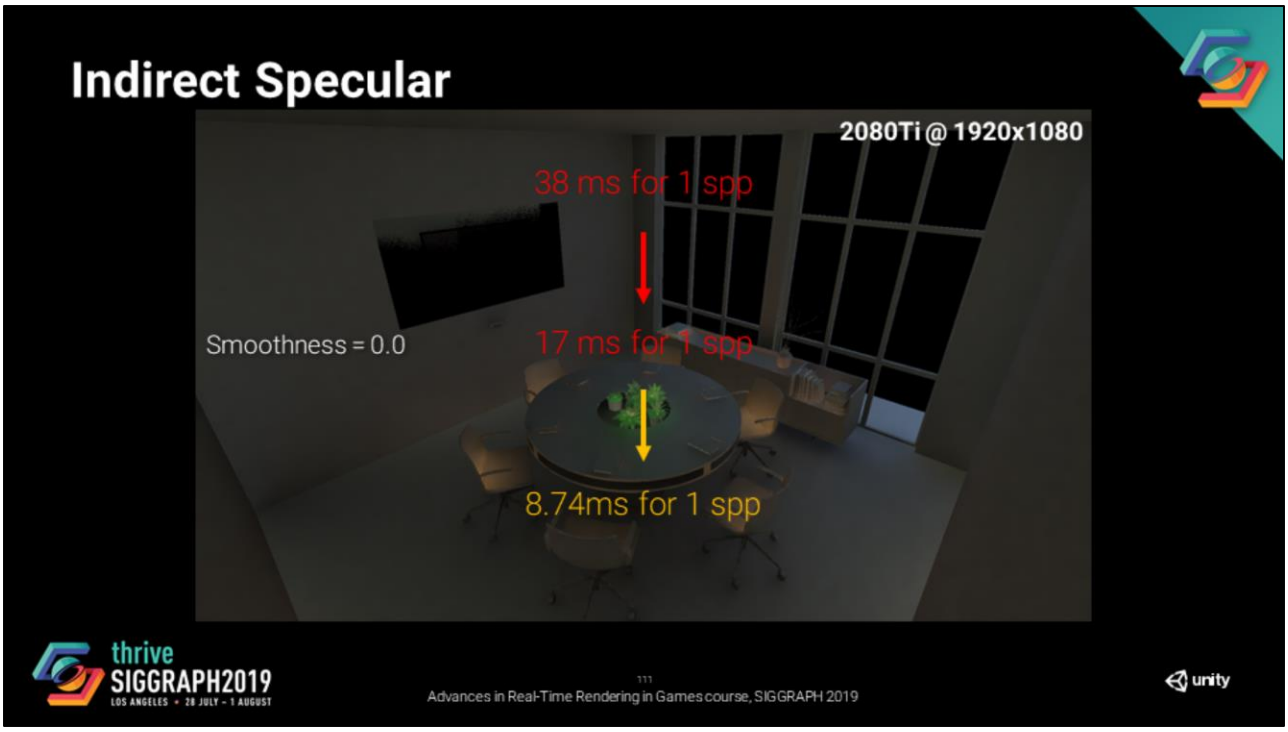

Store the data into a gbuffer in the same way we did in the previous example.

Then this data is returned to the ray generation through the payload. This time, instead of evaluating lighting, we store the gbuffer in UAVs.



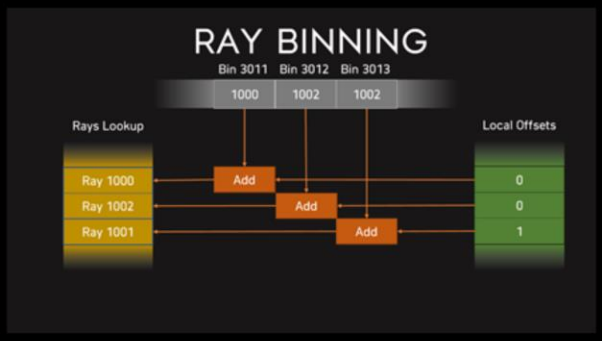

After that, we run the deferred lighting compute shader the same way we would do for rasterization (almost).

And we get the visual same result.




This is a significant improvement. However, we still can do better without compromising quality.

# Ray Binning



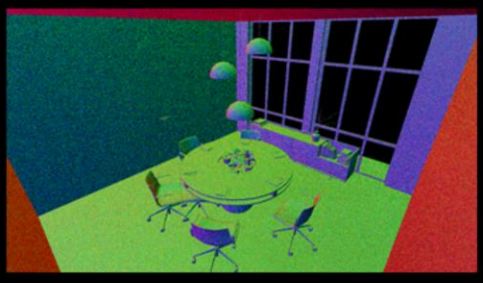

*"It Just Works": Ray-Traced Reflections in 'Battlefield V'*
*Johannes Deligiannis Jan Schmid EA DICE*



Let’s try something that is similar to what was presented at the talk ”It Just Works”: Ray-Traced Reflections in ’Battlefield V’ by DICE. Ray binning!

## Ray Binning

Depth Buffer



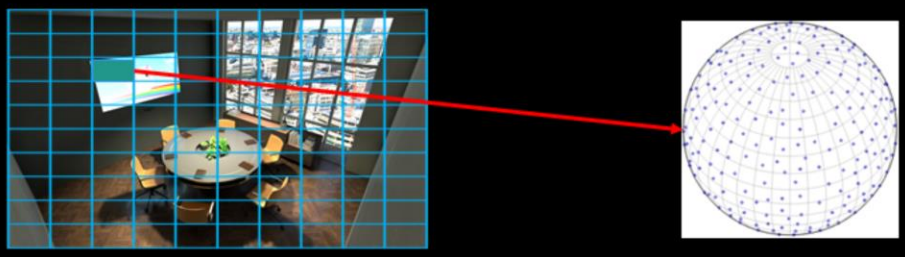
Direction Buffer



For every pixel on our screen, we have a screen space buffer that holds the origin of the rays and another one that holds the direction of the rays.

These two sets of information are also the two constraints that we want to use for binning.

## Ray Binning



*Screen tiles for binning*

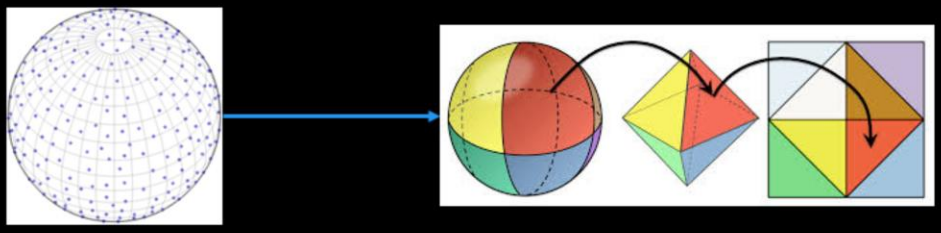
*Generated rays in the unit sphere for a tile*



Our approach is, by binning together directions in the same tile, we get the spatial constraint that we are looking for.

We can then focus on binning by direction.

# Ray Binning



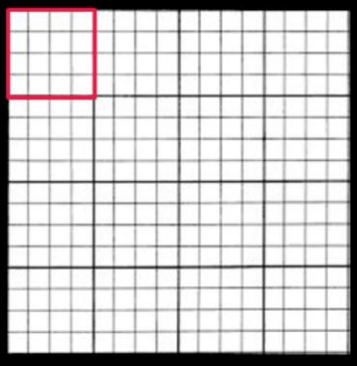
*Generated rays in the unit sphere for a tile*

*Octahedral space for ray direction binning*
Survey of Efficient Representations for Independent Unit Vectors [Cigolle et al 2014]



A good space to do the binning for directions is the octahedral space.

# Ray Binning

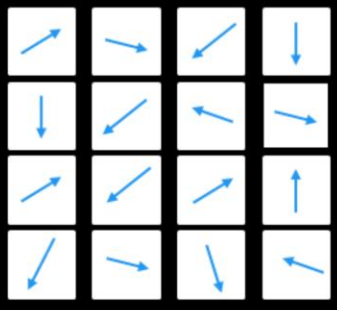


*4x4 Pixels Compute Groups*



Let’s go through the algorithm and let’s do it for a 4x4 tile.

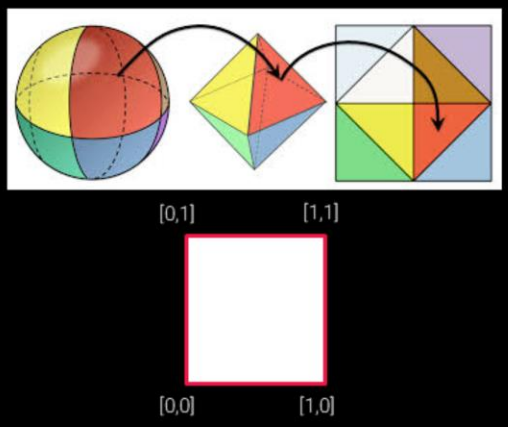
# Ray Binning





For every pixel in the tile, we have a direction

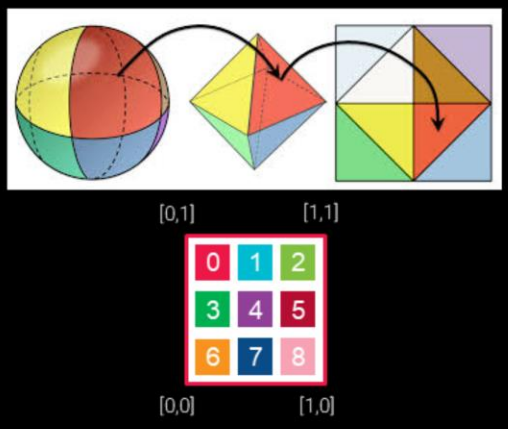
# Ray Binning






The octahedral space transforms directions into the unit square [0,1]x[0,1]

# Ray Binning

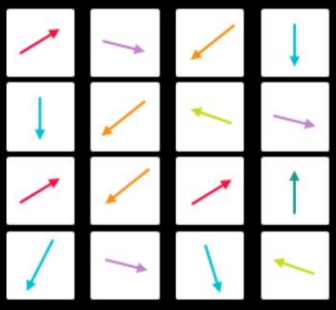





We define our set of bins (note that they have a specific order).

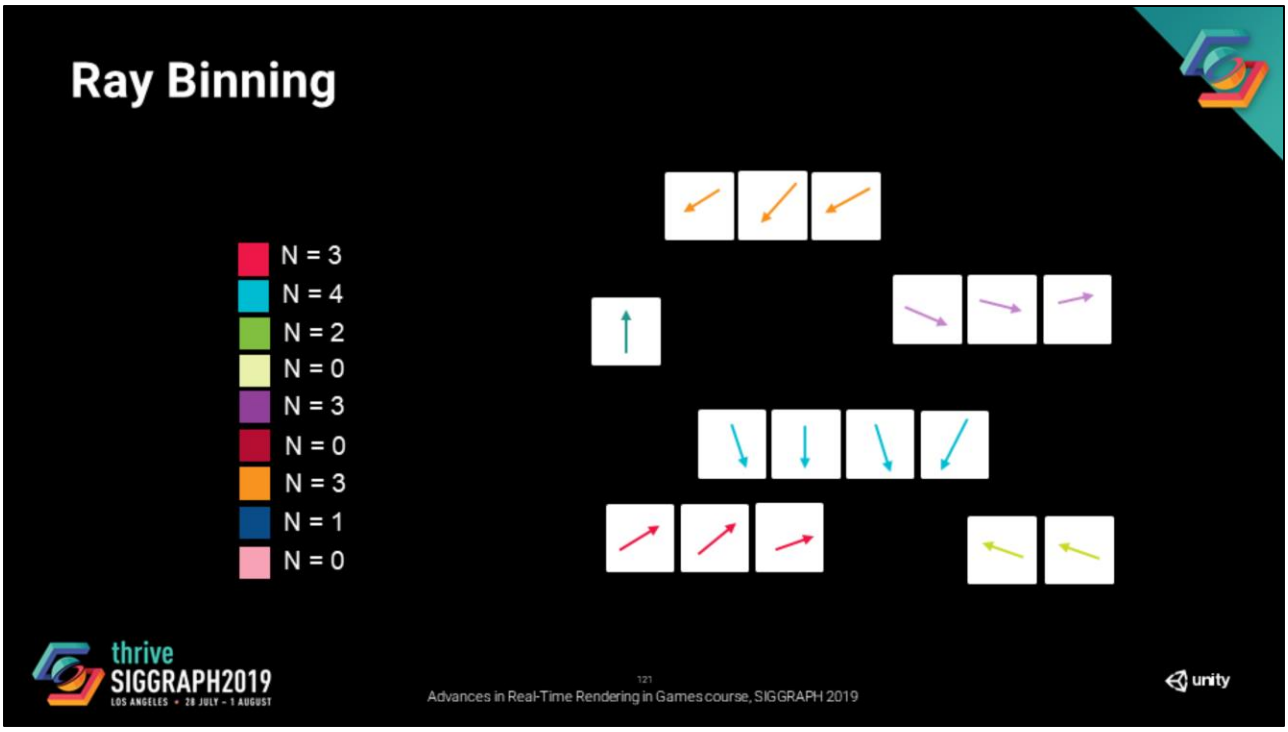
Just to make this more readable, let's use colors to identify our bins.

# Ray Binning



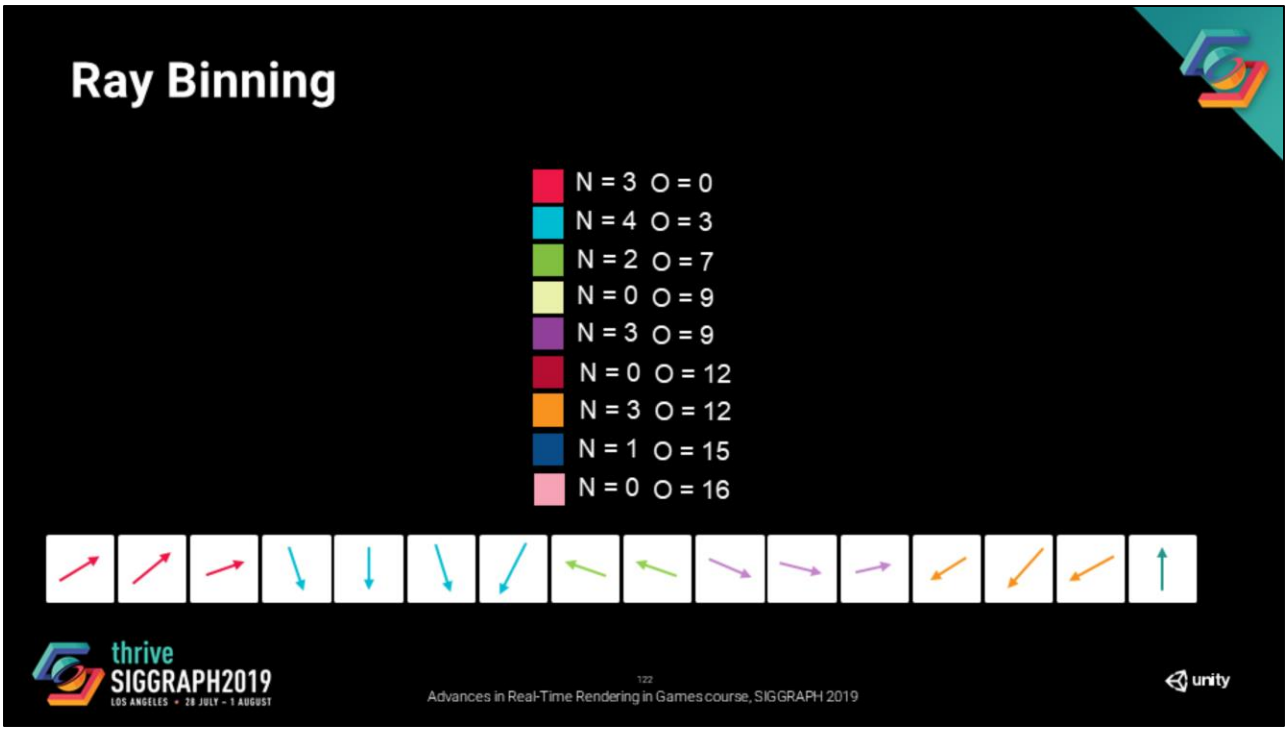


Then we are able to identify the bin for every direction.

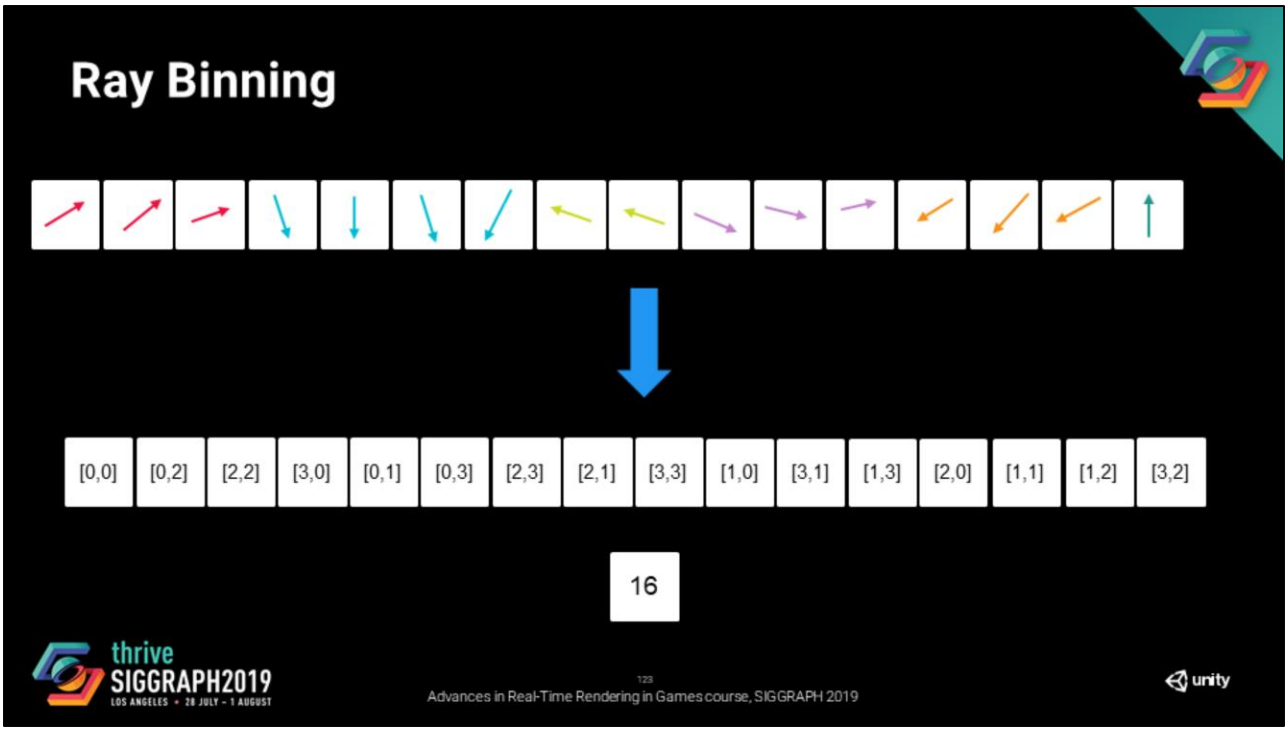



Using an atomic_add we are able to compute two things:

- The number pixels that are in each bin
- The offset of each pixel in its bin




Then we compute the global list of directions and thus the offset of each bin in this global list of directions



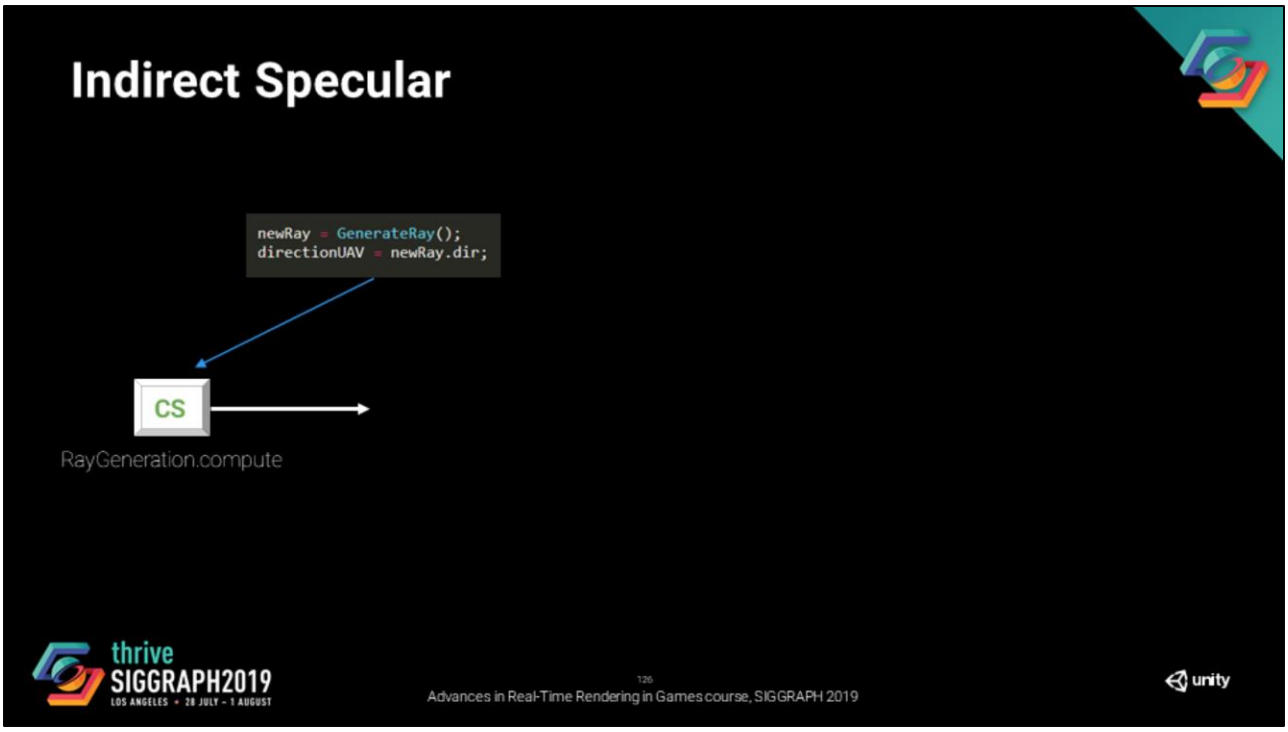

The output of the binning algorithm are two buffers:

- Pixel coordinates sorted by bins
- The number of valid rays in this tile (16 in our test case, all rays were valid)

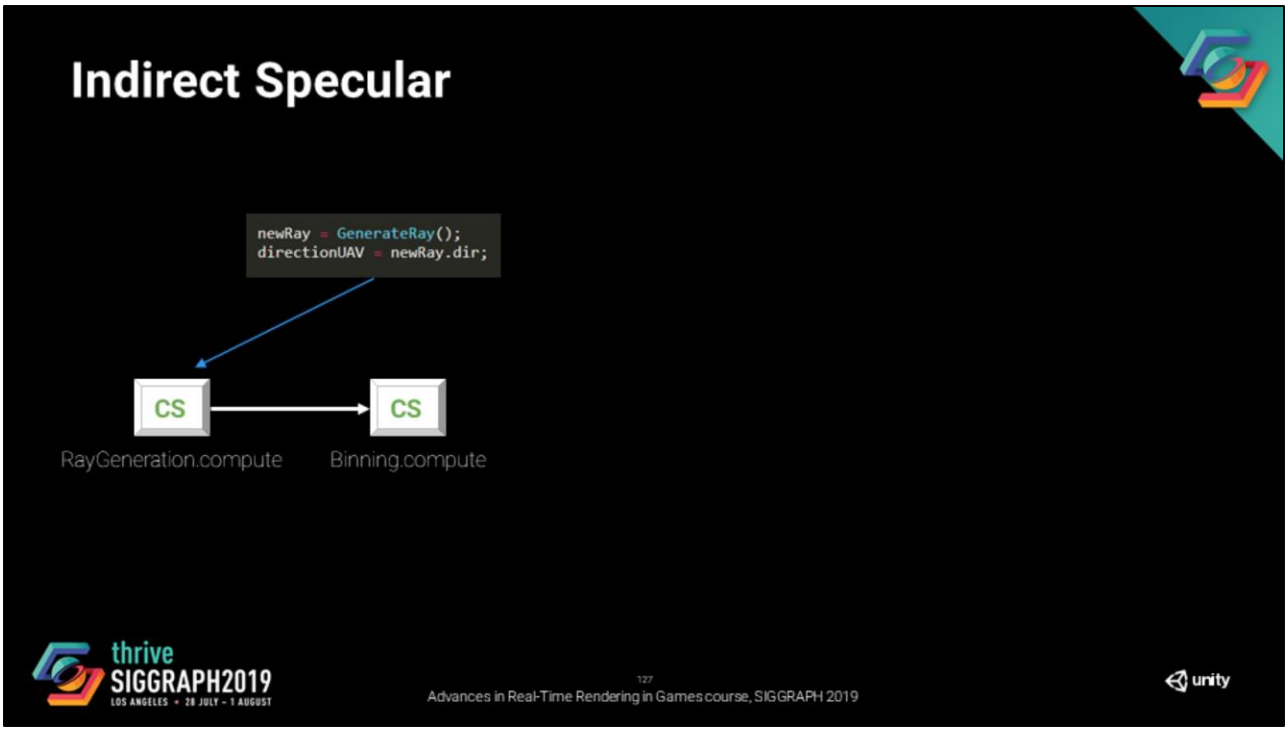

And that is what we do for binning.

Here you have the actual code. If you want to check it out or even improve it, see our github respository


Now that we have an improvement, let’s change our implementation so that we can use ray binning

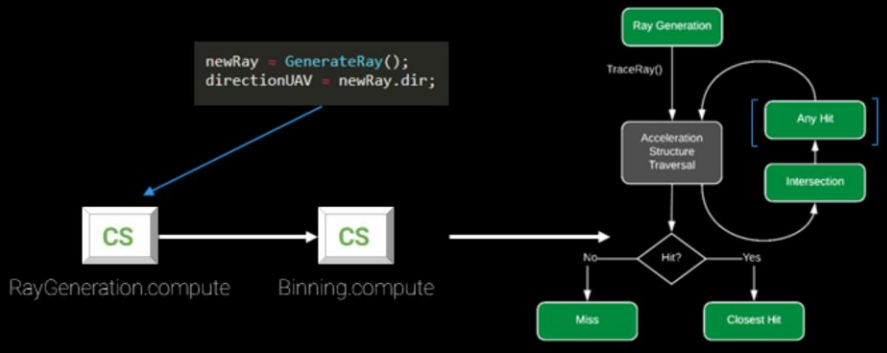



From the previous step, we have already extracted the ray generation (which is handy).



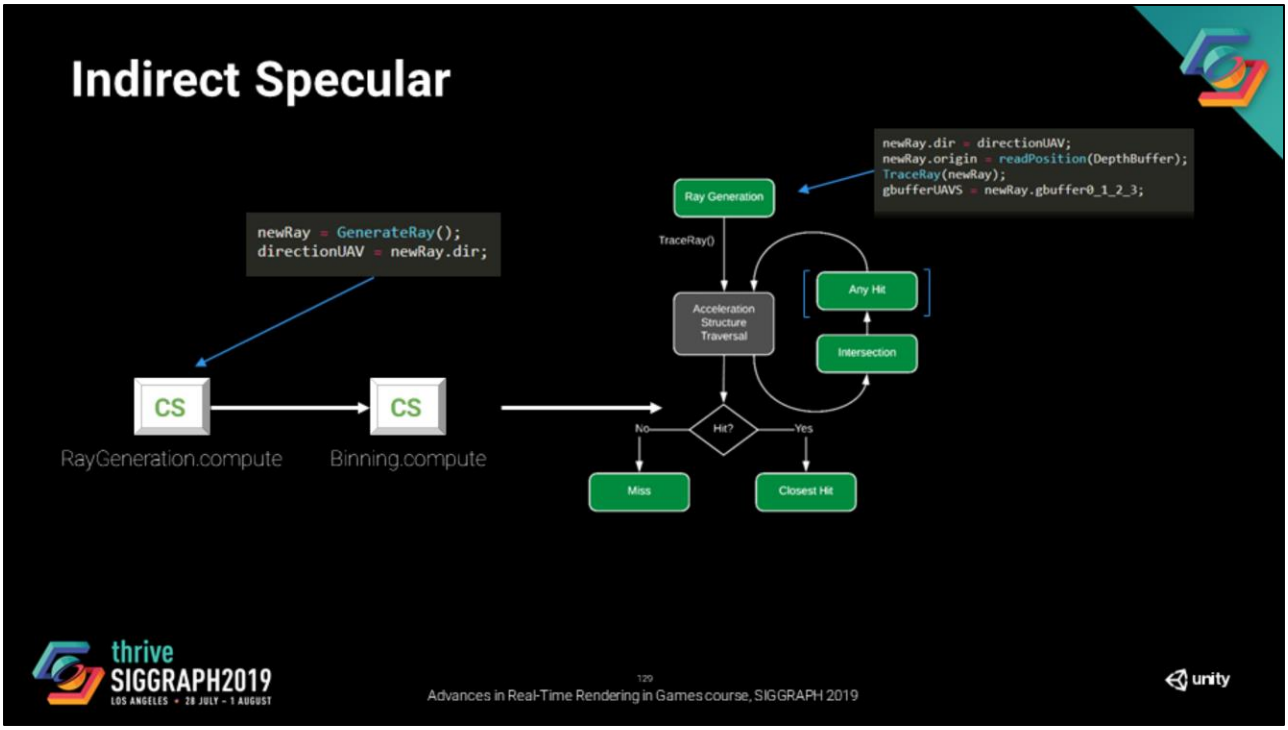

Now we can run the ray binning pass that we just described

# Indirect Specular



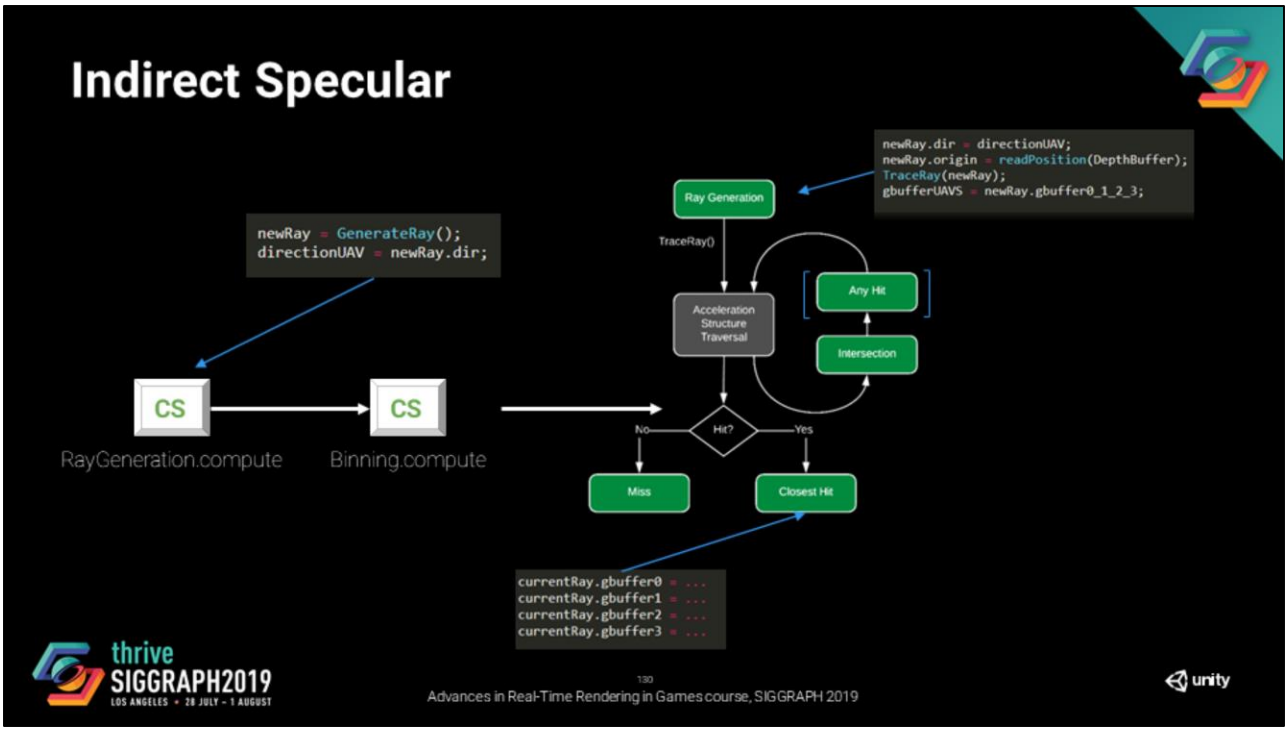



Then we move to our ray tracing shader pipeline



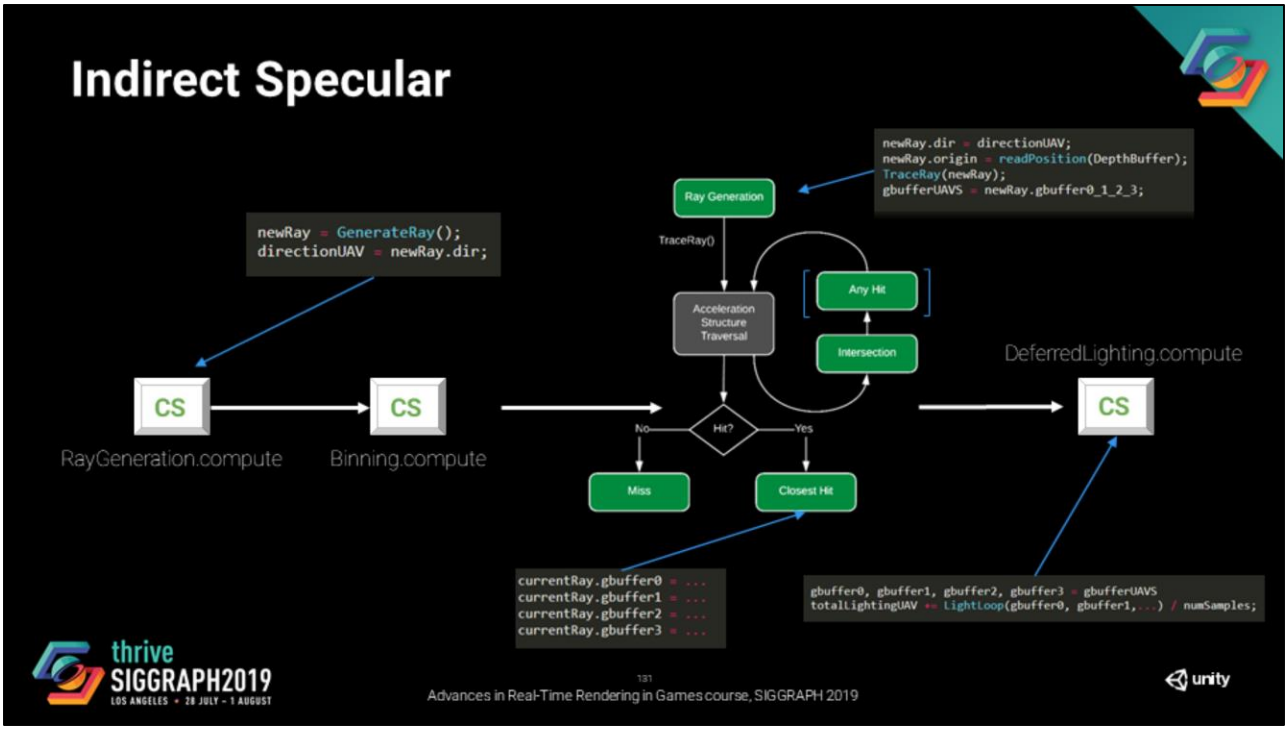

From the binned pass we are going to read the directions and depth buffer

Then its about the same as the previous step. We trace the ray




We Store the material data into the payload gbuffer

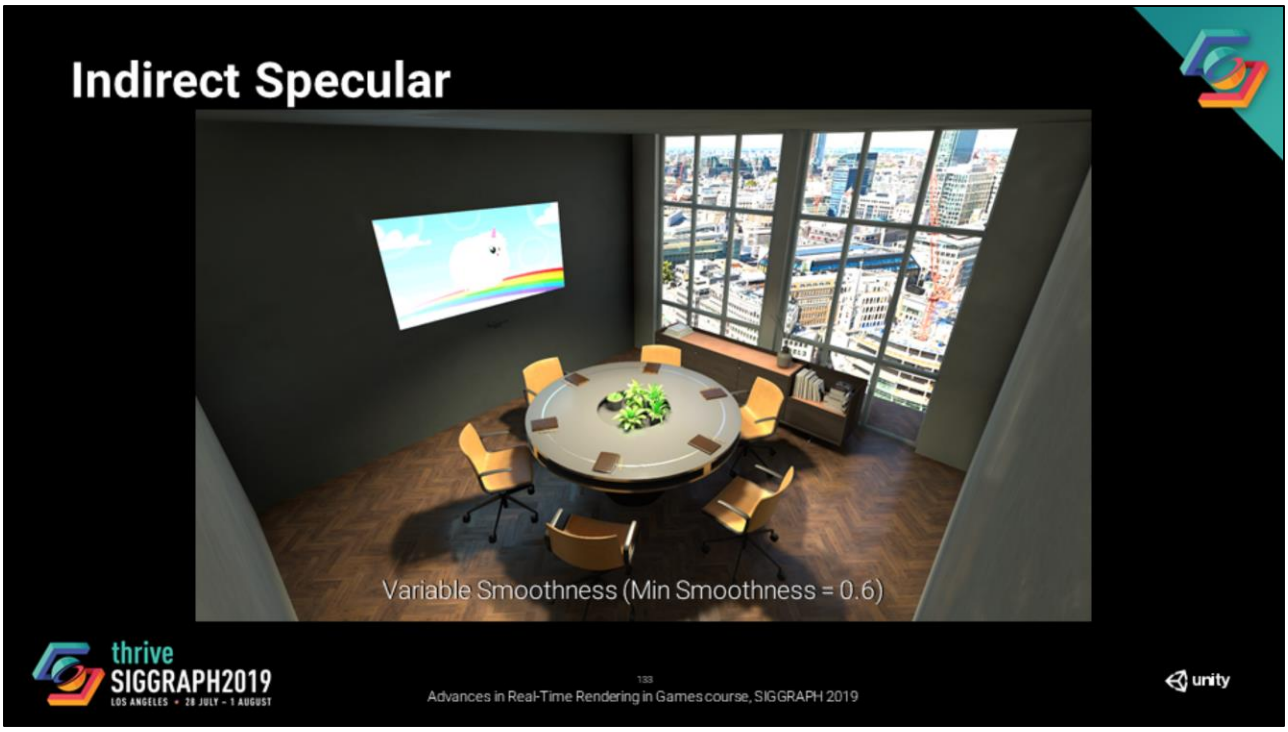
Then output it to UAVs



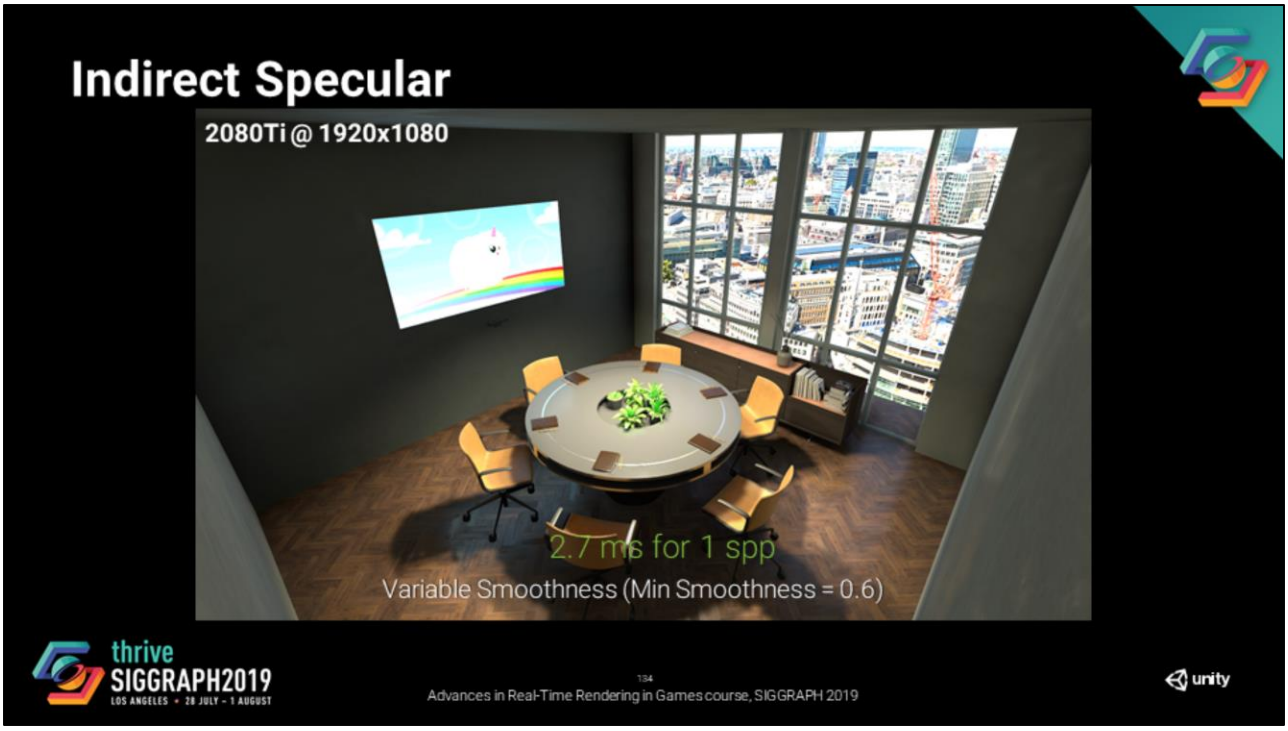

Then we run our deferred lighting


Almost a 2ms improvement, not bad. This starts to be viable for a real-time application, but not quite for games.

However, remember that the case that we are profiling, is the worst case possible.




Our original case is a more reasonable configuration. It is also acceptable to fallback to an other indirect specular technique below a given smoothness, 0.6 in this screenshot




This time is still a bit high. But I think it is acceptable.

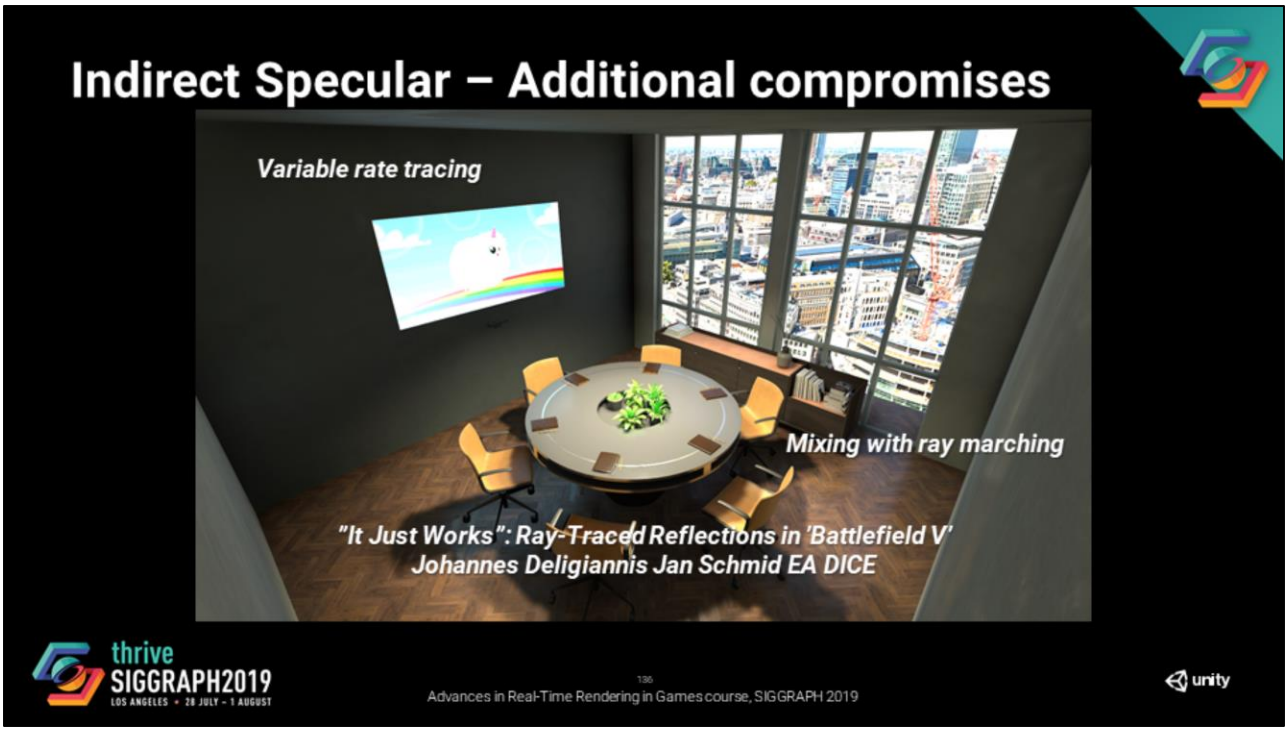
Please note that in the additional materials, you will find a table that details all these timings


Until now, we have not made any quality compromise (except the compression of the material data into the gbuffer). But if the time that we have is still too high (which is something that is understandable), there is additional things that we have explored to reduce the execution time.

You can provide LOD/Raytracing nodes in your shader graphs to reduce the complexity of closest hit shaders (thus the cost of ray tracing).

You can also use a simplfiied lighting mode for the deferred lighting (for us takes the simplest variant of our main lighting model), this reduces the cost of the lighting pass and the memory pressure on the ray tracing shaders (if you managed to export less parameters).

You can evalute the effect in half resolution and upscale it (this reduces both ray tracing and light evaluation, but increases the filtering).

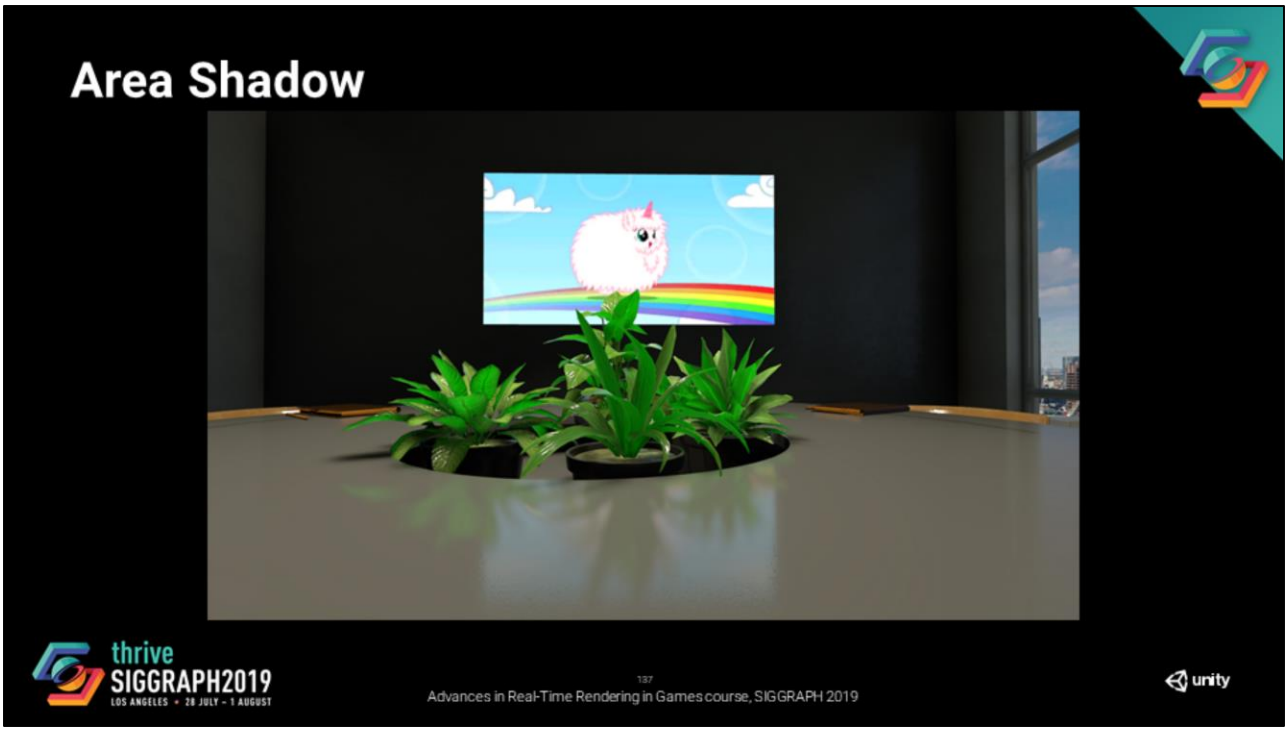



Based on your content, this may not be enough.

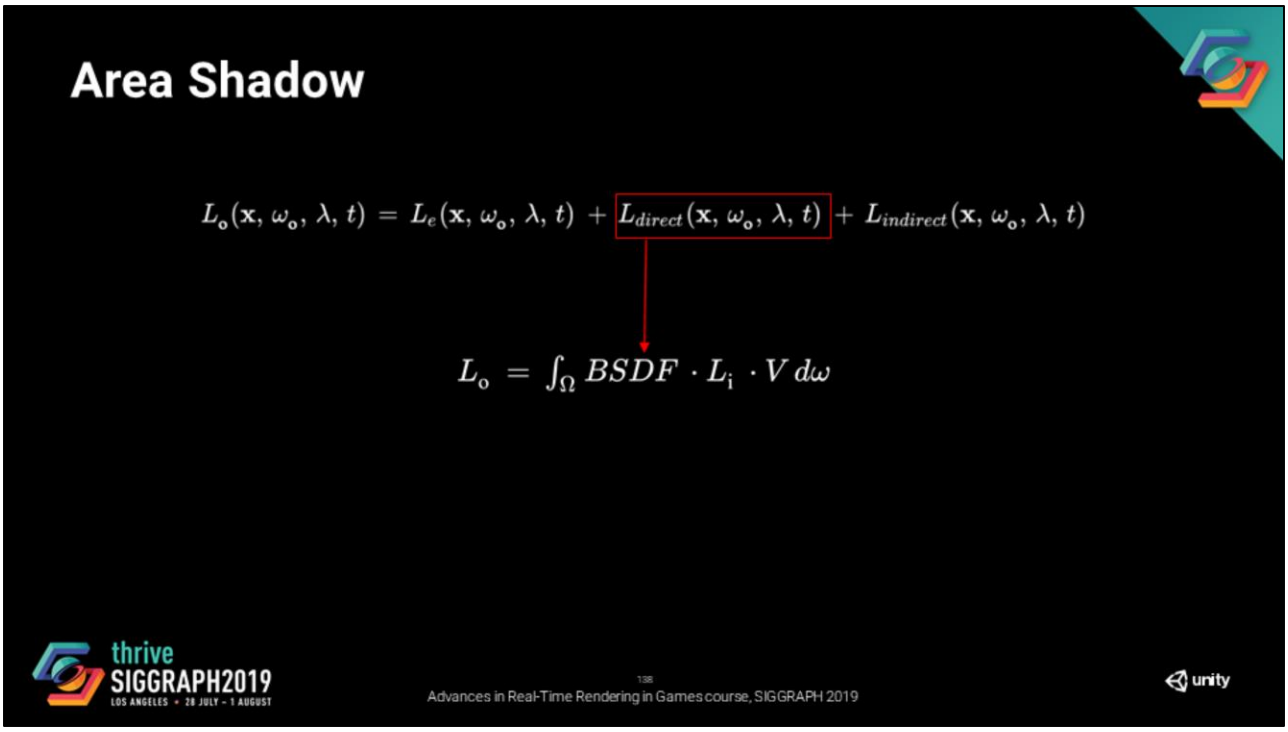
In the talk that i mentionned earlier, there is couple of additional tricks that can help.

You can do variable ray tracing (meaning that you can go much lower than half res in certain areas) and also mix the effect with ray marching for additional performance.

That covers it for indirect specular.




The next subject that I would like to cover is area shadows




Earlier I said that ray tracing was used to evaluate indirect lighting. The thing is we also use it for some parts of direct lighting.

For an area light, this is the formula for evaluating the direct lighting of an area light on a given point

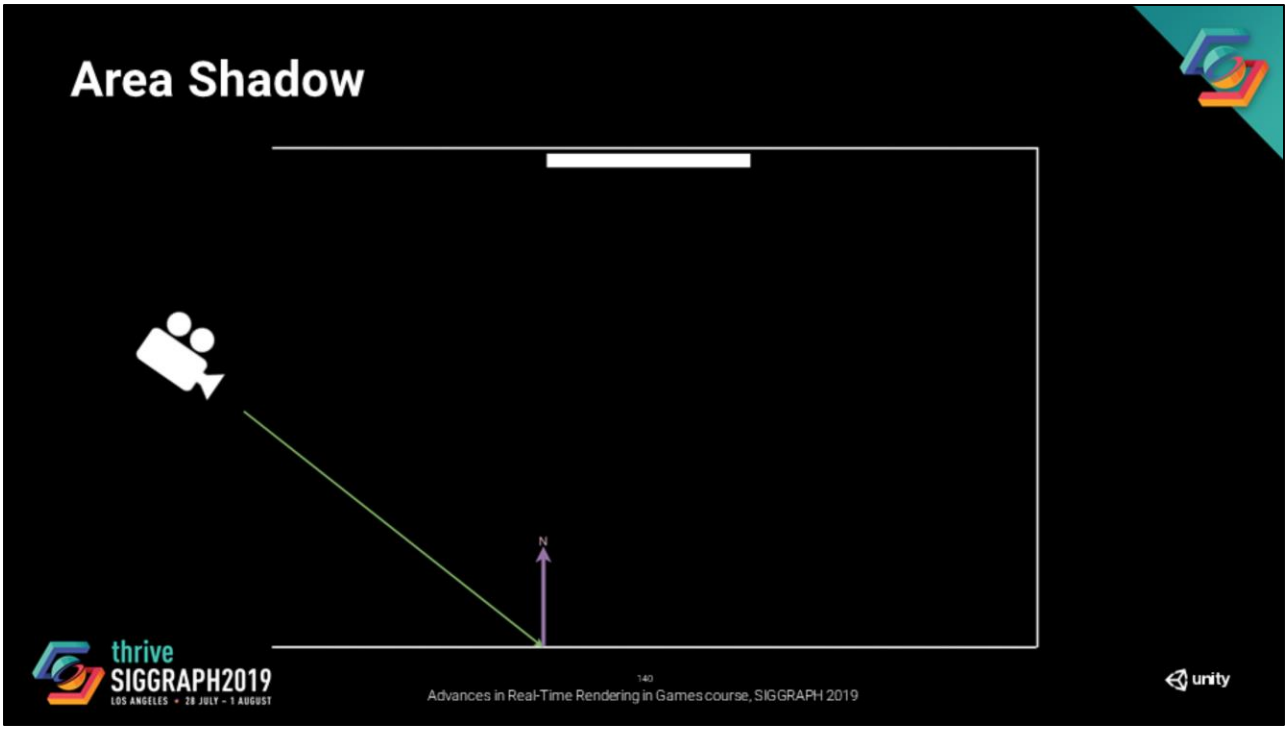
# Area Shadow

$$L_o = \int_\Omega BSDF \cdot L_i \cdot V \, d\omega$$

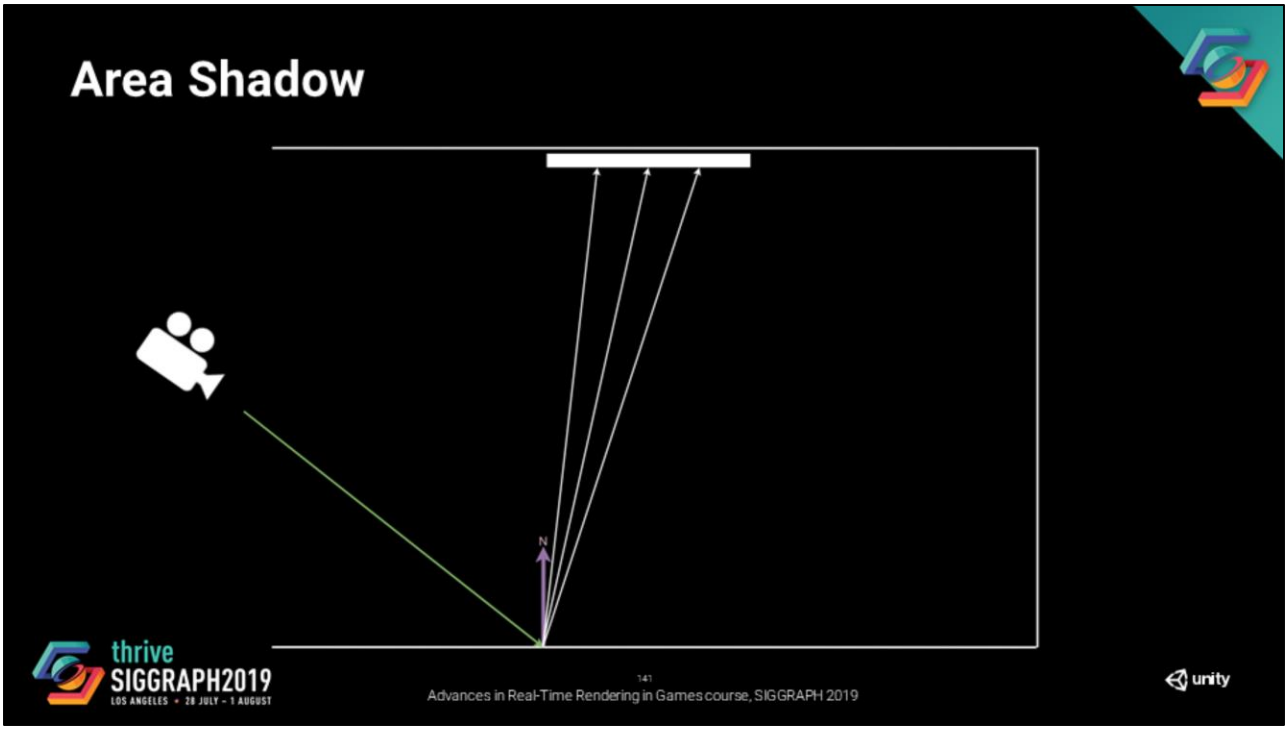
*Sponza Scene - Reference*



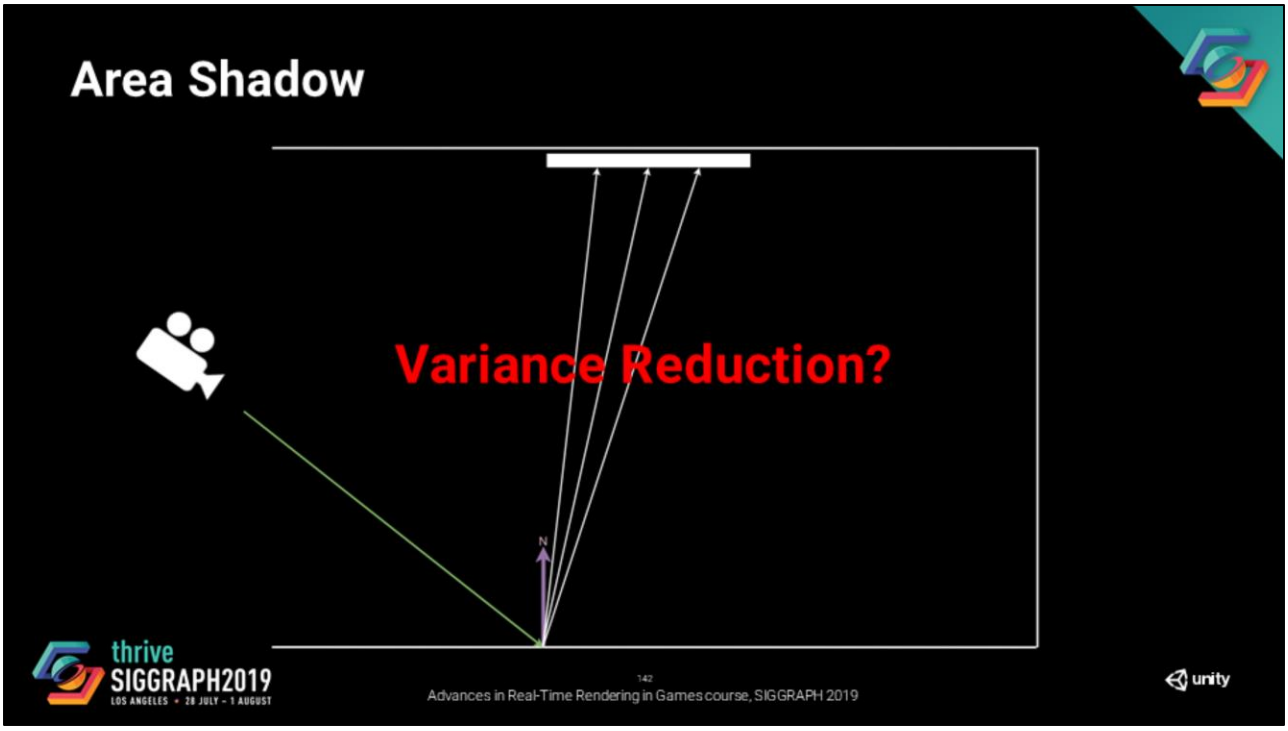
If we use this in an offline integrator, this is the result that we would get (and that is what we are looking for)




Maybe then we should do the same in our case! For a given point…




We generate samples on the light source, evaluate the lighting for these directions, accumulate, normalize, and get the correct result.




But again, we need to search for a way to reduce the variance of our integration.

# Area Shadow

$$L_o \approx \int_\Omega BSDF \cdot L_i \, d\omega$$

*Sponza Scene - LTC only*

"Real-Time Polygonal-Light Shading with Linearly Transformed Cosines"
*Eric Heitz, Jonathan Dupuy, Stephen Hill and David Neubelt*



In rasterization, we use the linearly transfomed cosines approximation for evaluating the radiance of an area light on a given point.

## Area Shadow

$$L_o \approx \frac{\| \int_\Omega BSDF \cdot L_i \cdot V \|}{\| \int_\Omega BSDF \cdot L_i \|} \int_\Omega BSDF \cdot L_i$$

*Sponza Scene - LTC + Stochastic Shadow*

"Combining Analytic Direct Illumination and Stochastic Shadows"
*Eric Heitz, Stephen Hill and Morgan McGuire*



Using this analytic approximation to evaluate a ratio approximator, we have a pretty efficient variance reduction method and that is what this paper ( ) is about.

# Area Shadow – Visibility

$$L_o = \int_\Omega BSDF \cdot L_i \, d\omega \int_\Omega L_i \cdot V \, d\omega$$

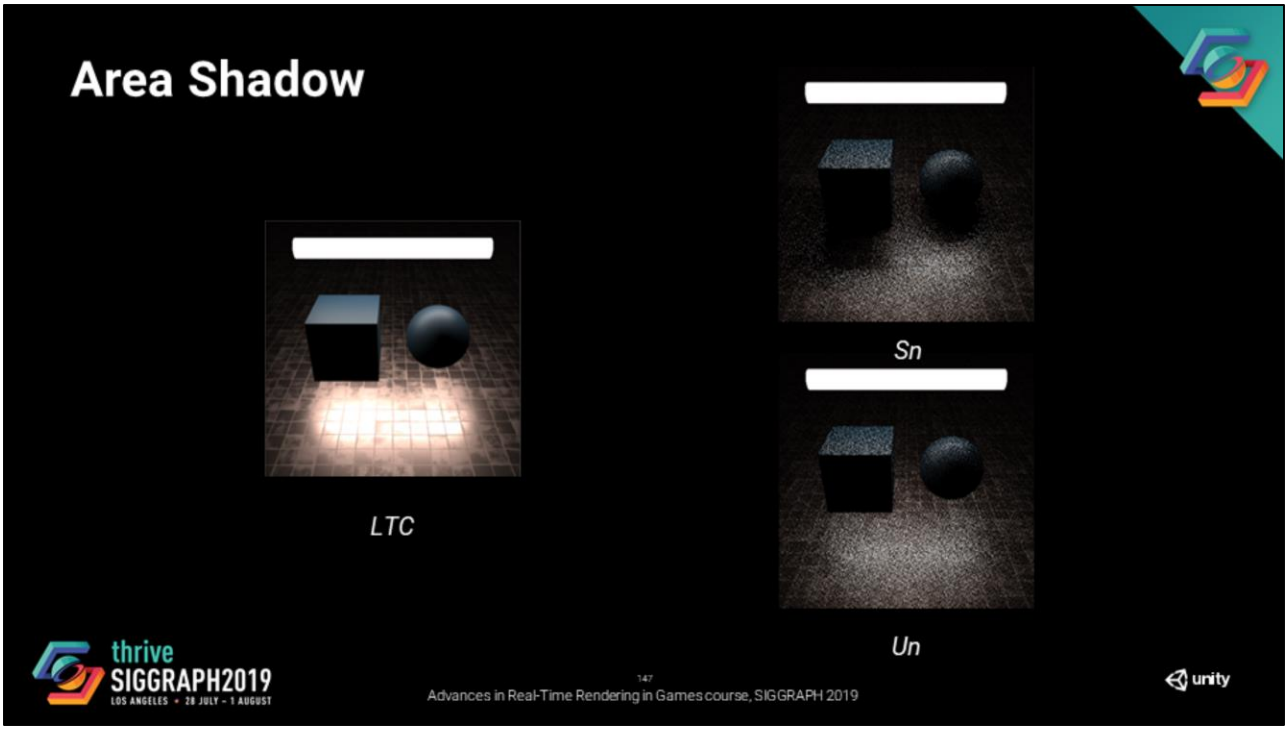


It is also possible to evaluate an « Occlusion » term by doing the integration of the visibility term of the area light. It looks fine.

## Area Shadow – Stochastic



But it does not correctly capture the visibility term for specular lobes



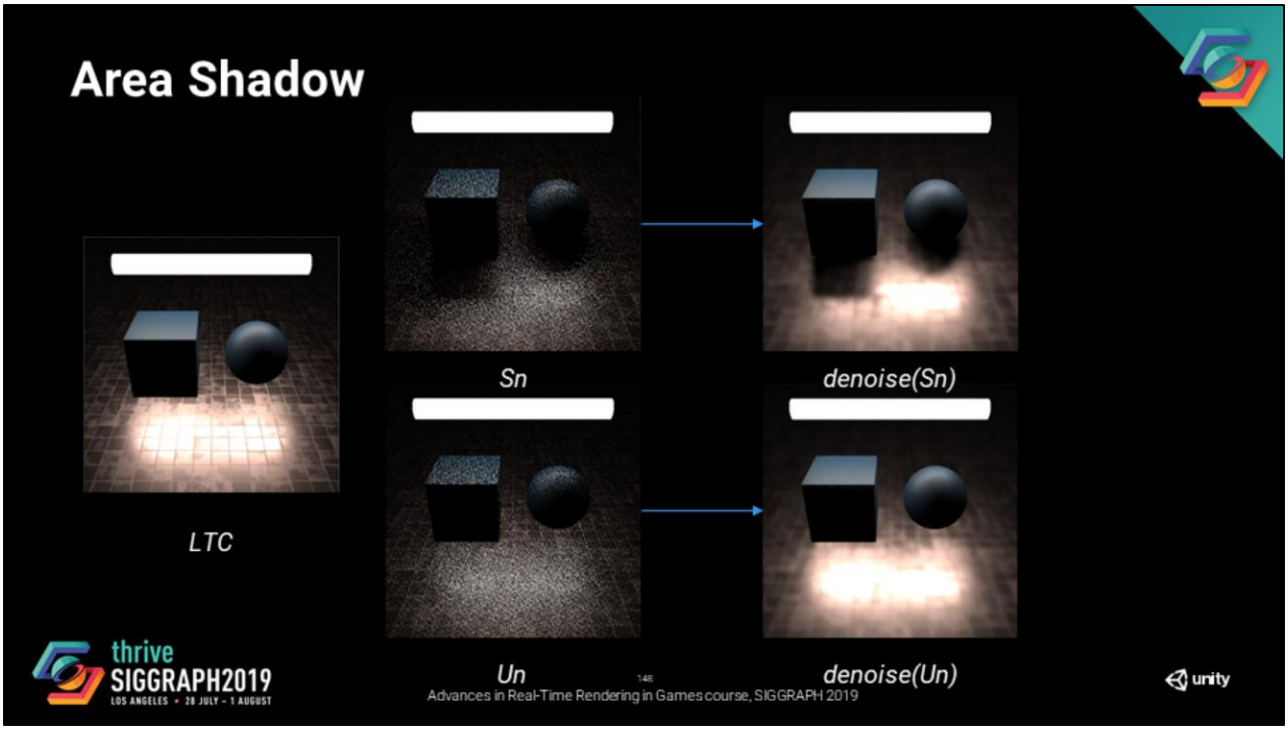

Just in case you are not familliar with the paper, I am going to do a quick reminider of the stochastic area shadow algorithm

We first start by evaluating the analytic value of the lighing

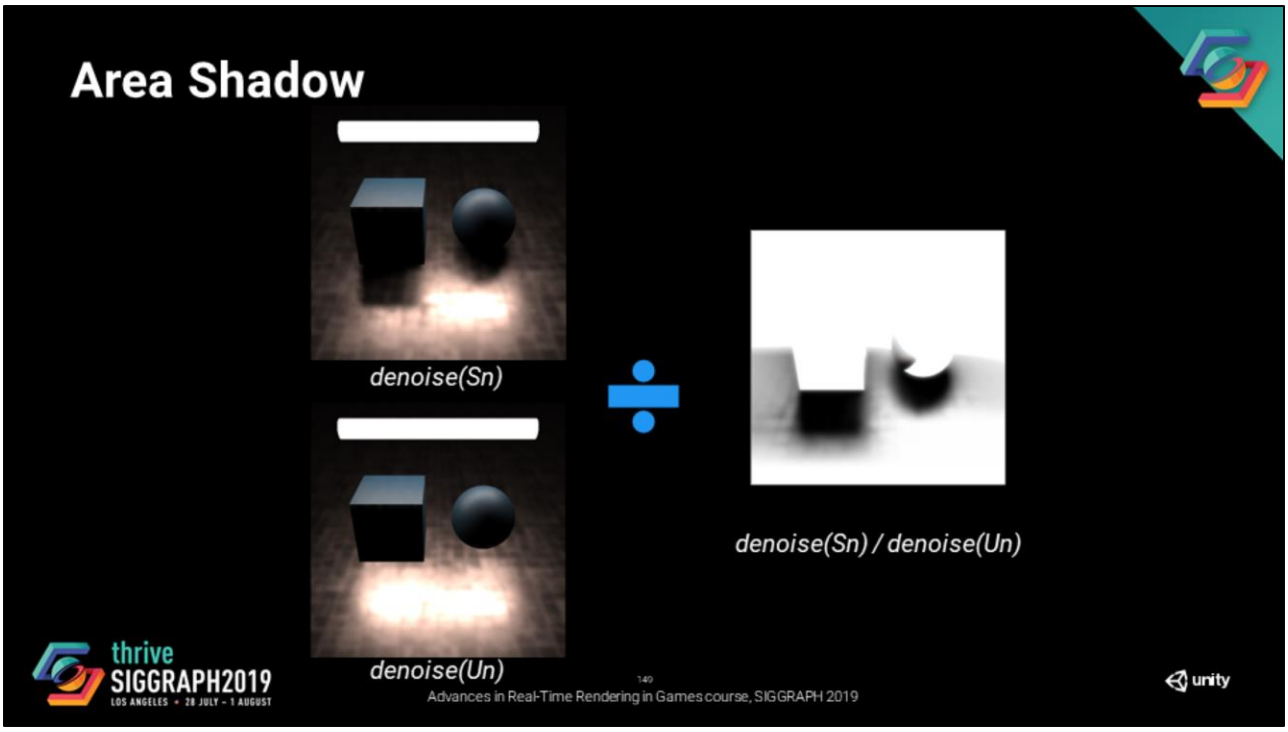
Then we need to produce two buffers:

- Sn which is the integration of the area light including the visibility term

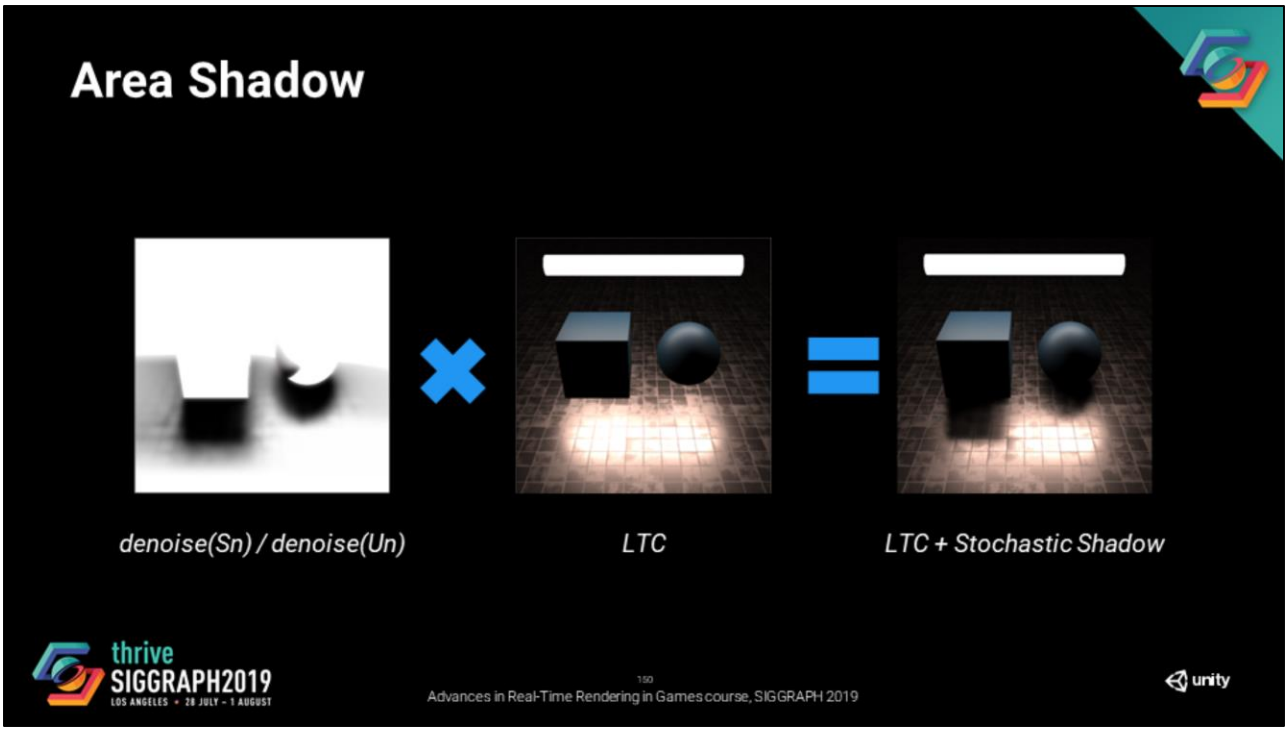
- Un is the same integration, but without the visibility term




Then the two buffers SN and Un are denoised independently but with the same kernels.

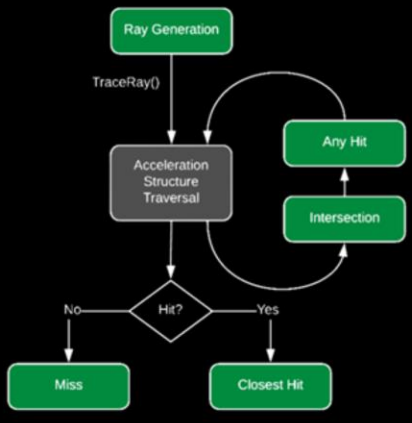



Then we divide and get our stochastic shadow




Finally, we multiply by the analyitic value and get the final result that we are looking for!

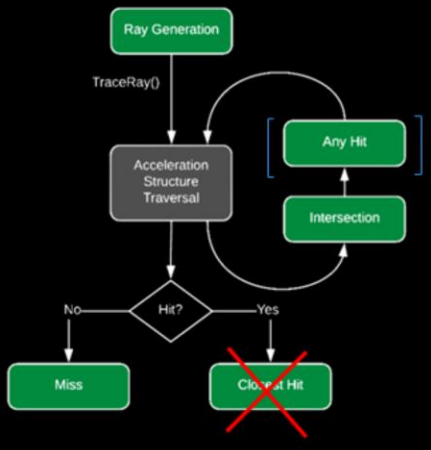
# Area Shadow - Stochastic





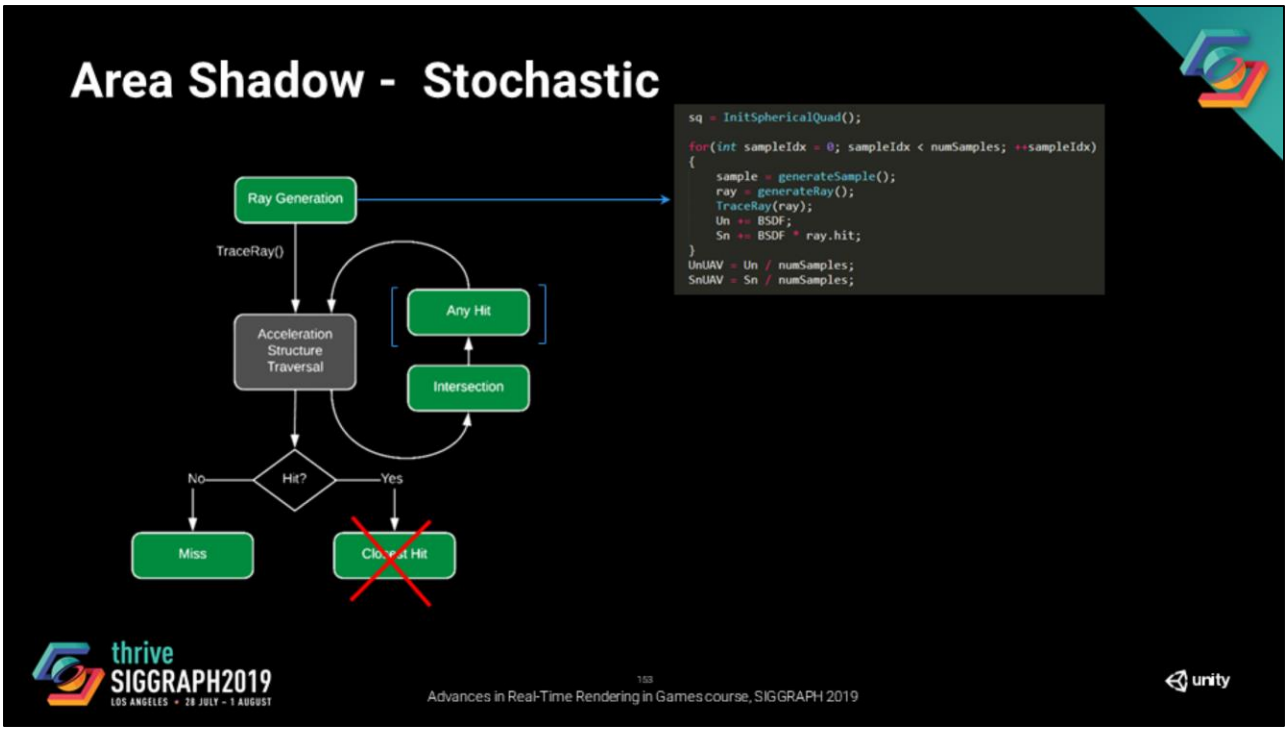

That said, let’s go through an other implementation. We start with the ray tracing shader pipeline

# Area Shadow - Stochastic





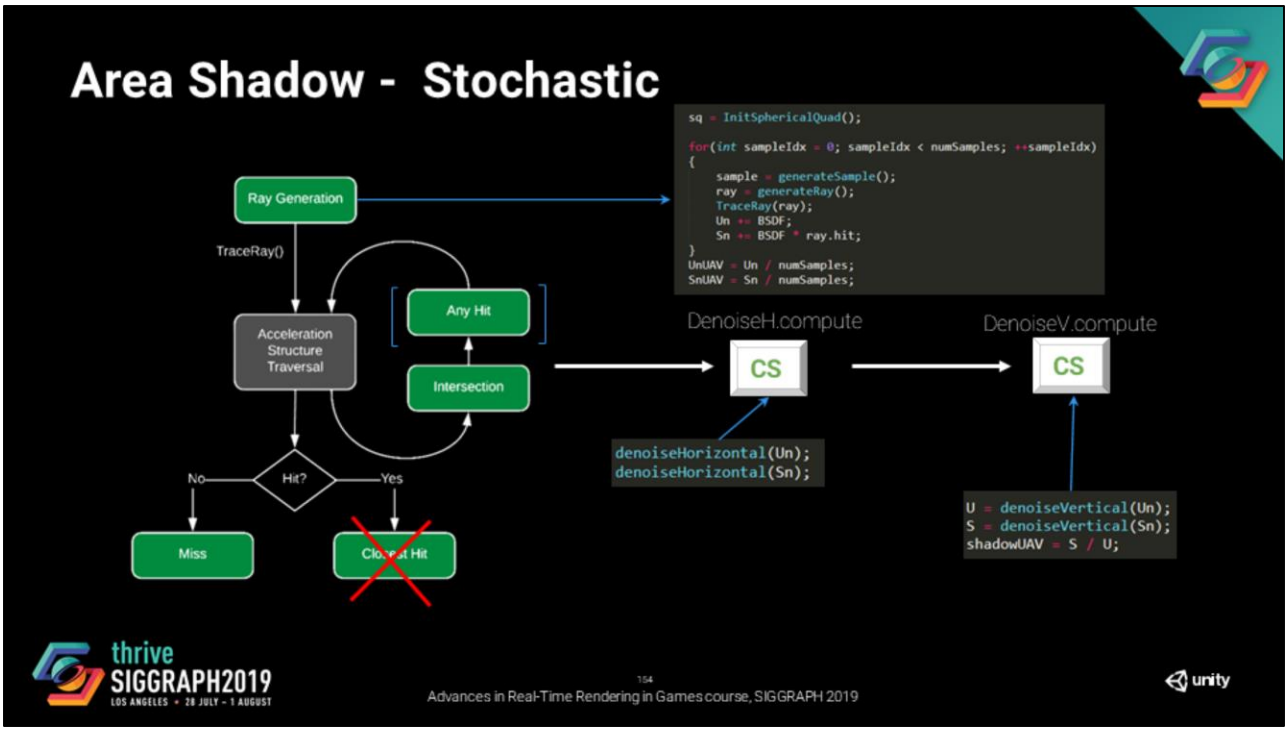

This is a visibility evaluation, we do not use the closest hit shader. We also avoid using the any hit shader which drastically impacts the cost of the effect.




At this stage, our entry point is the ray generation shader.

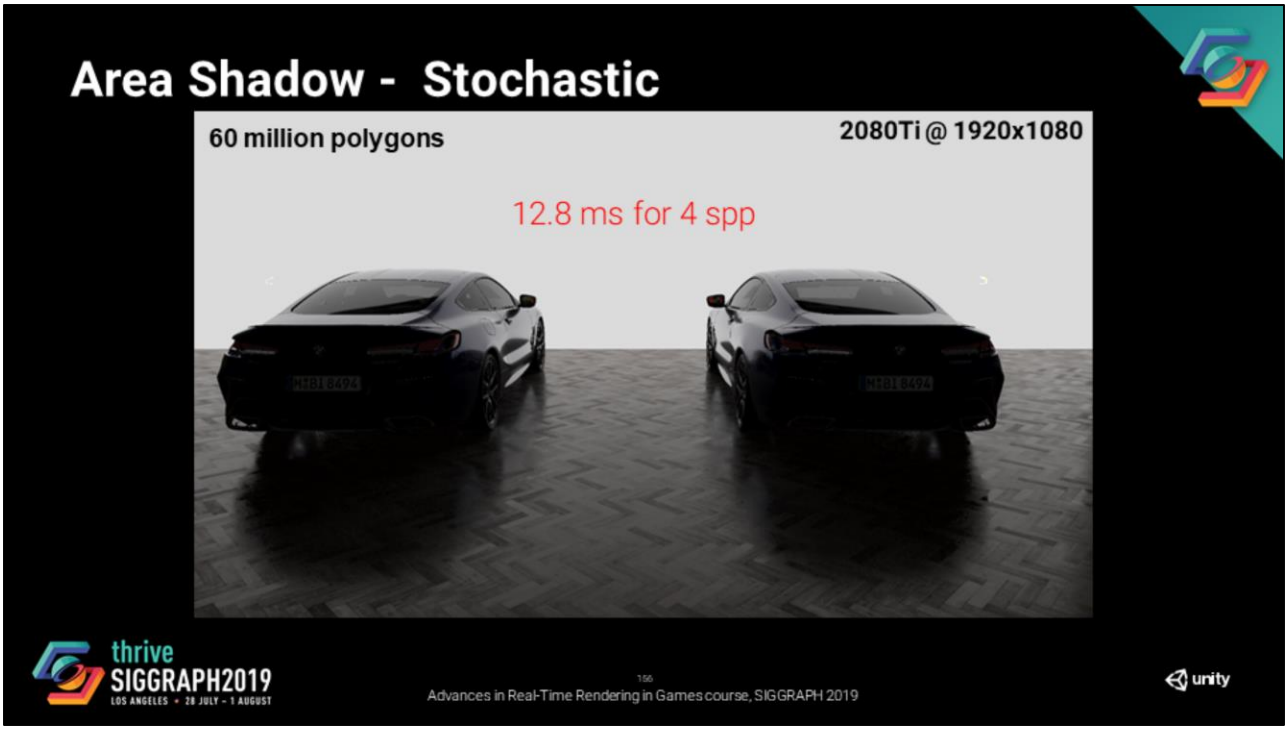
We start by initializing the sampling struct for the light source. Then for every sample that we want to evaluate, we generate a new sample, generate the ray, and trace the ray.

Then our miss shader notifies us if an intersection was detected. We accumulate Un and SN, normalize.



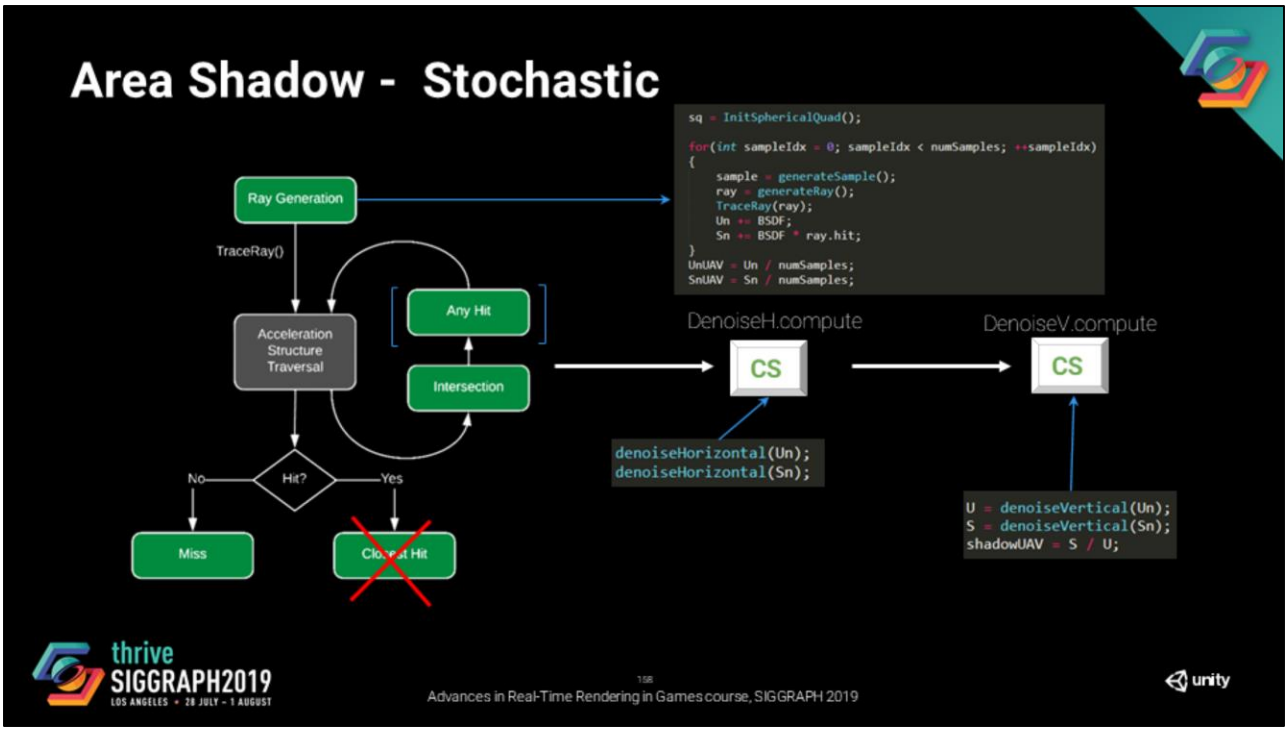

Then the next step is denoising. Separable bilateral filters are not really a thing, but it happens to work well enough, so we use that.


So we get the result we are looking for!

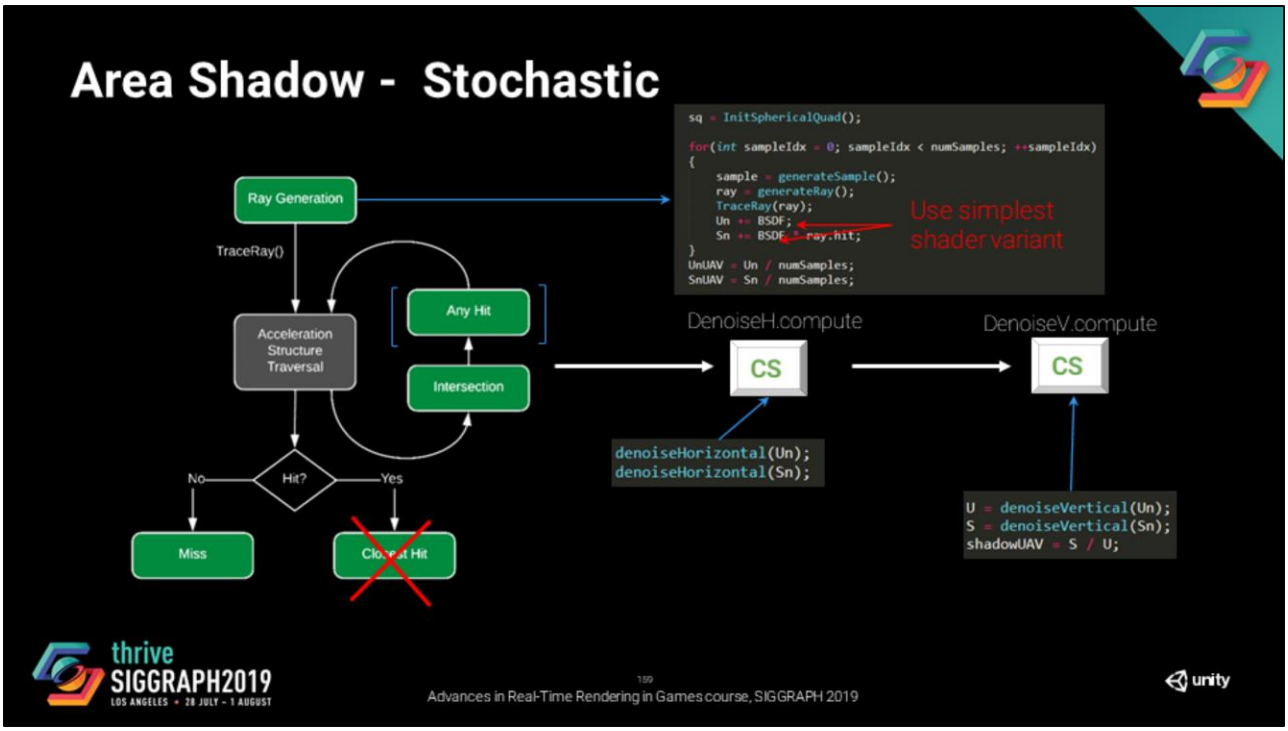



But again, same problem as before.
Even if the scene is pretty big (60 million polys), this is quite slow. Note that this includes filtering.


We definitily can do better.



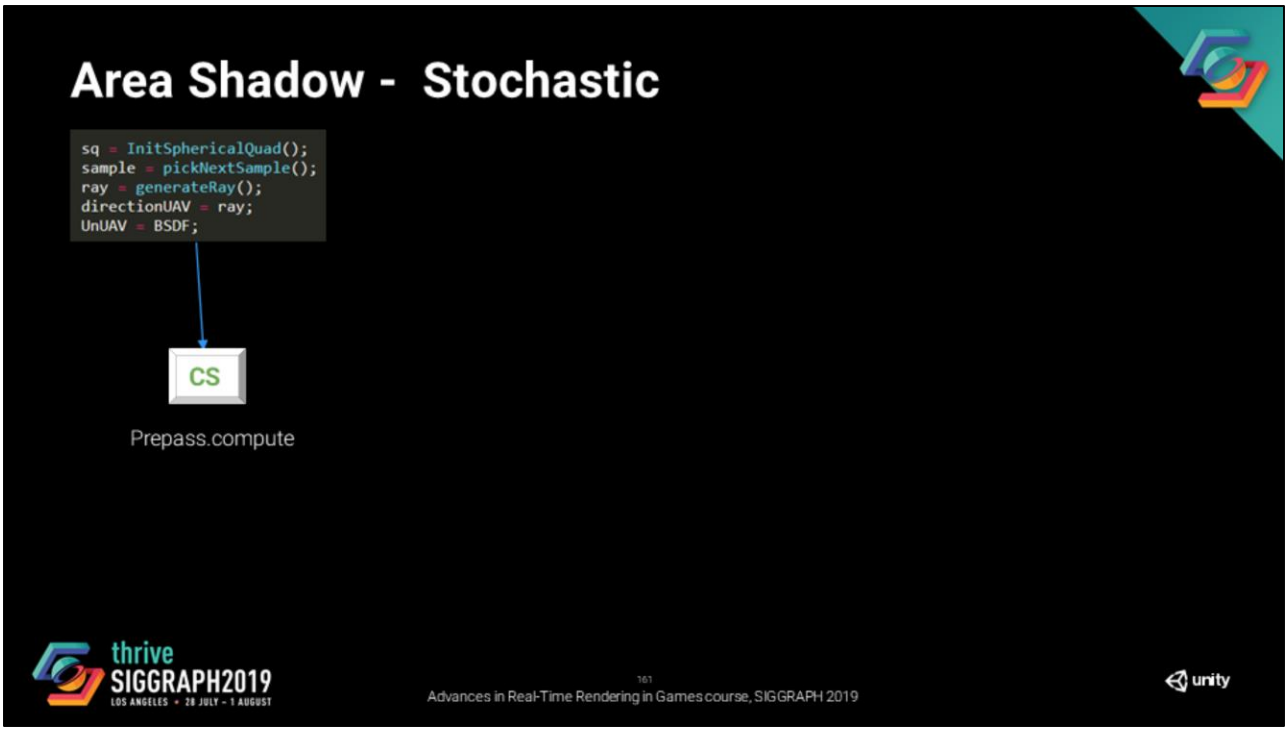

The first things that we can do, and that is a quite easy optimisation, is to use a simpler shader variant when computing U and Un.



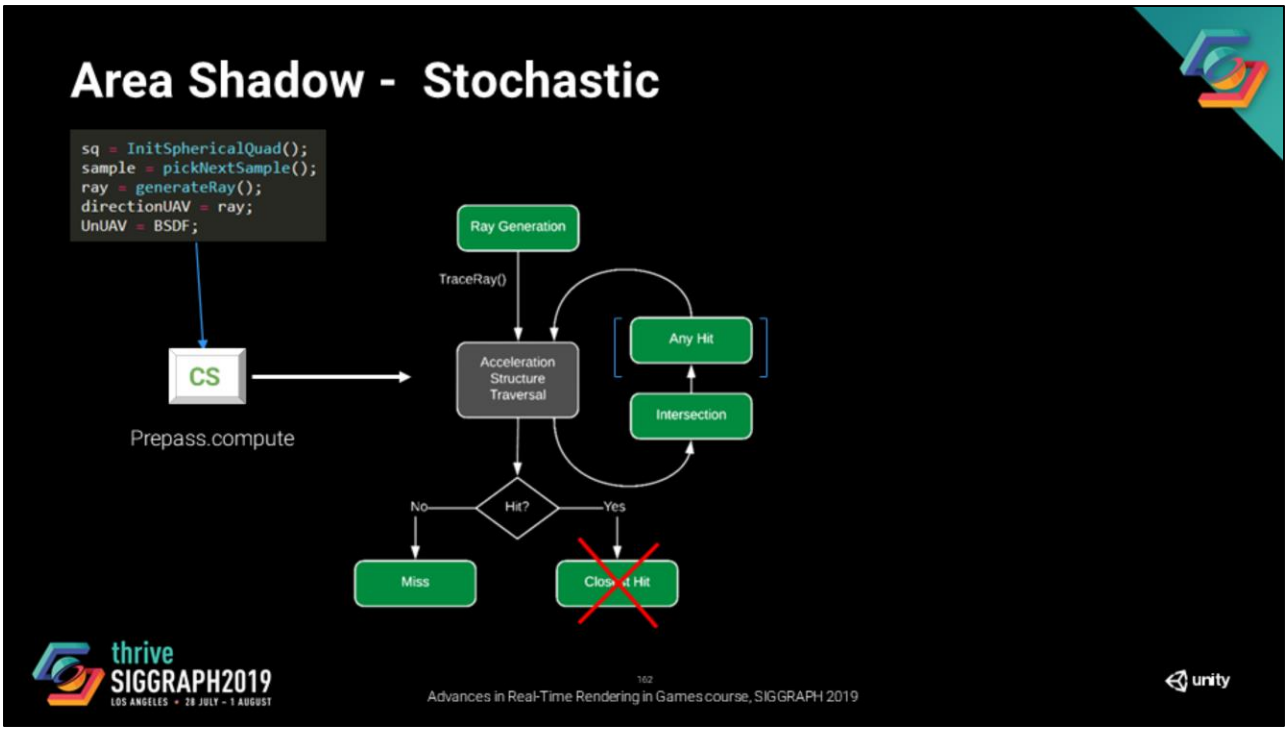

That reduces significantly the register pressure in the ray tracing shader and thus gives better performance.


With this simple approximation, we get something that improves the execution time.

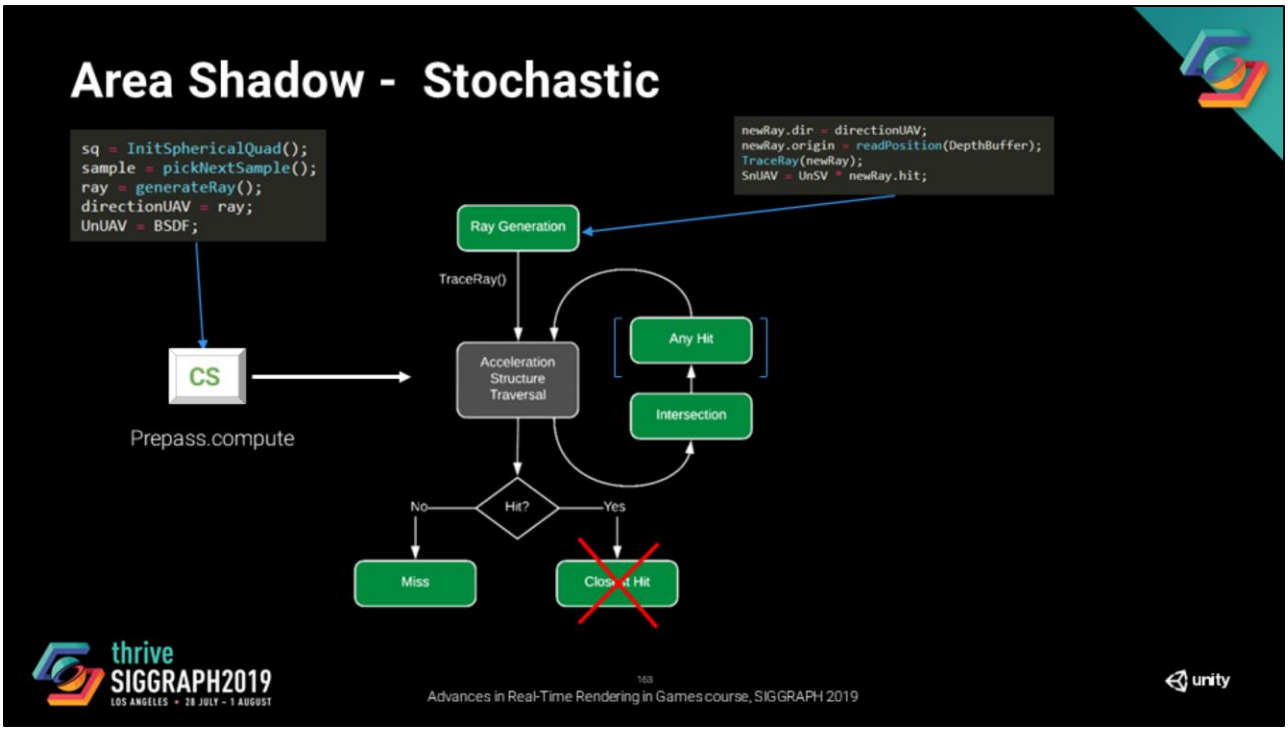
On the other hand, we've learned from reflections that offloading to compute shaders is a good tip. Let's try that.



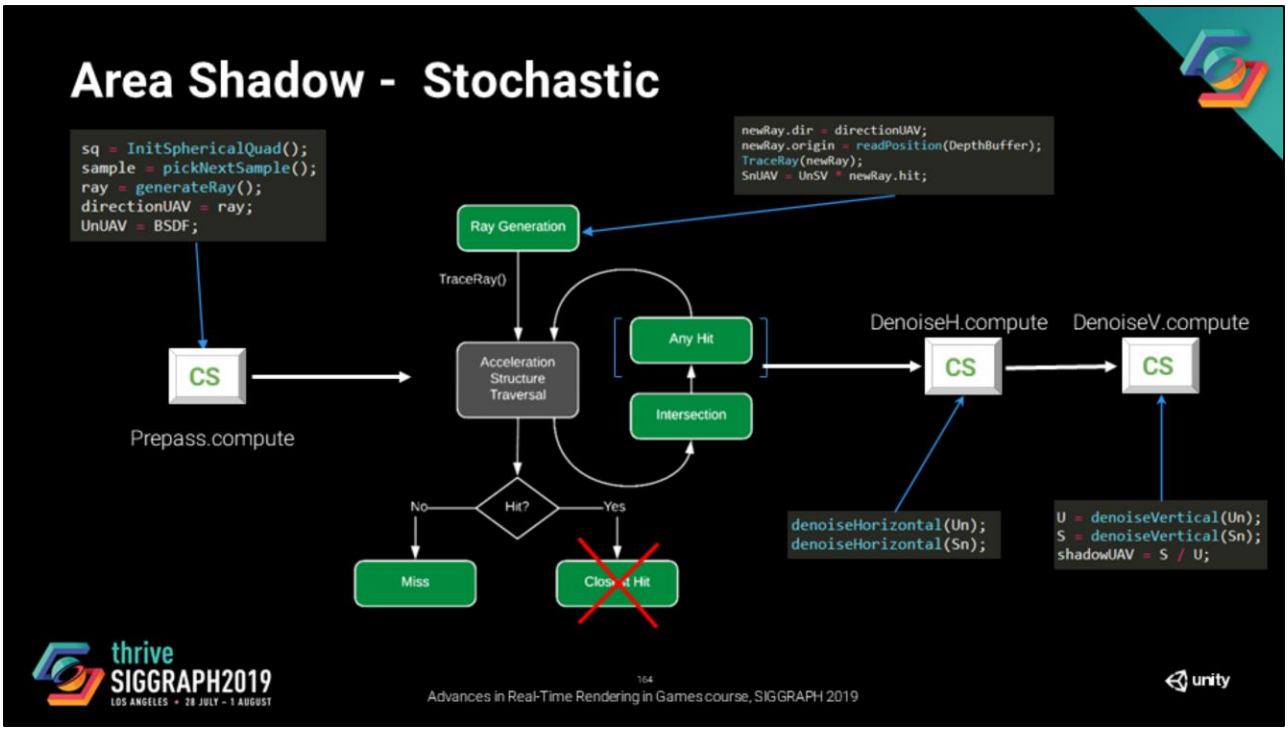

We start with a prepass shader that generates the direction and evaluates the current unshadowed lighting



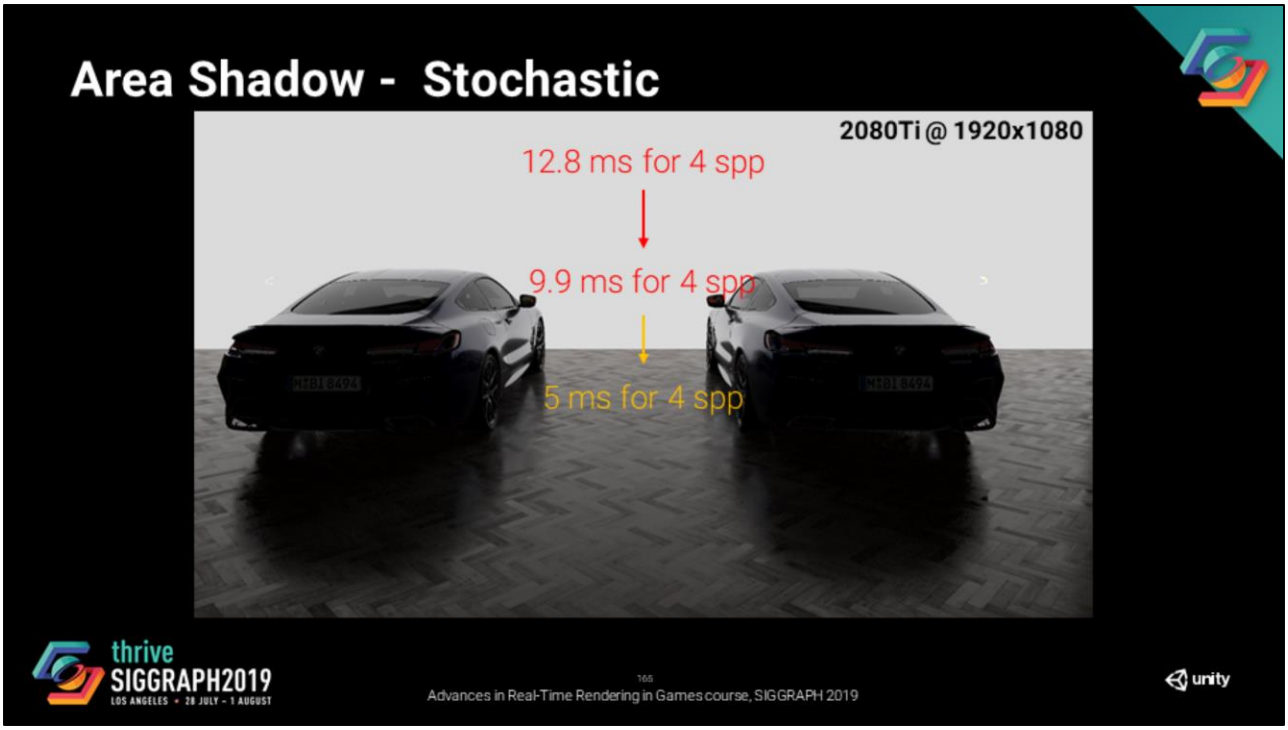

Then we run our ray tracing pipeline



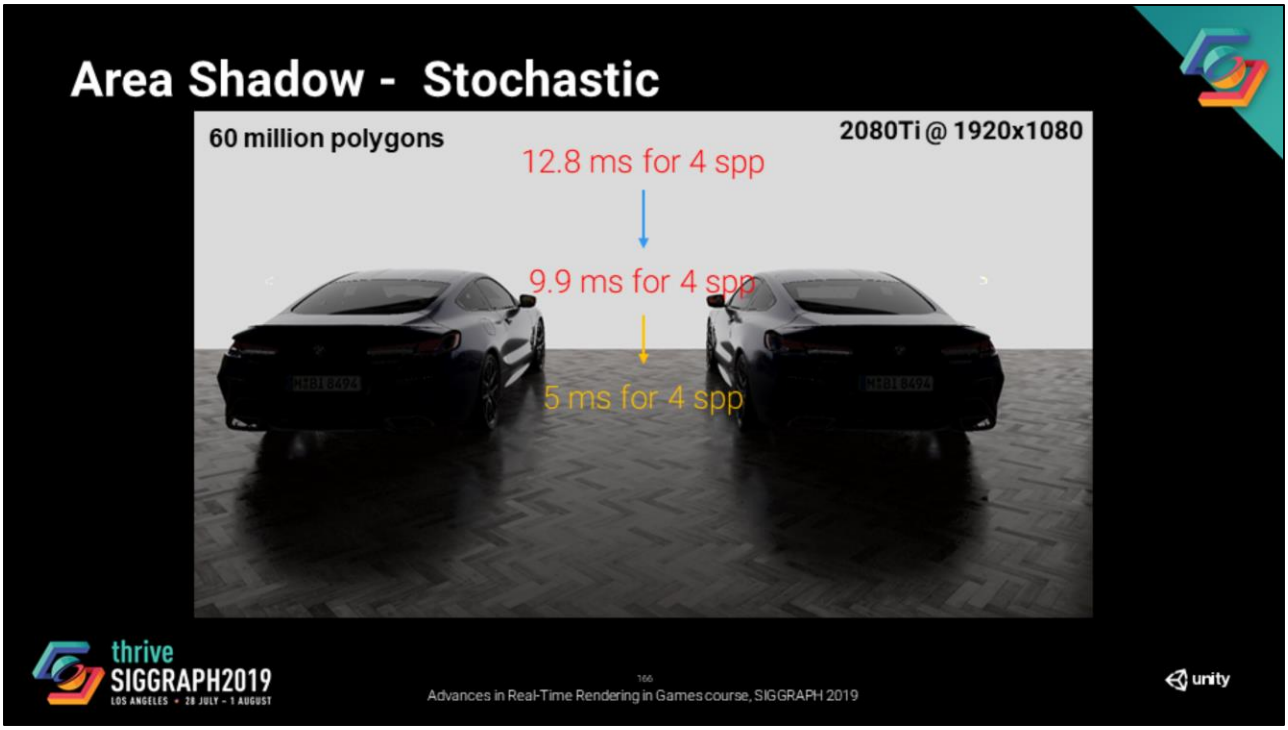

We read the direction from the UAV, recompose the ray that we are currently casting, trace it and store the shadowed term.




And then we run our denoiser



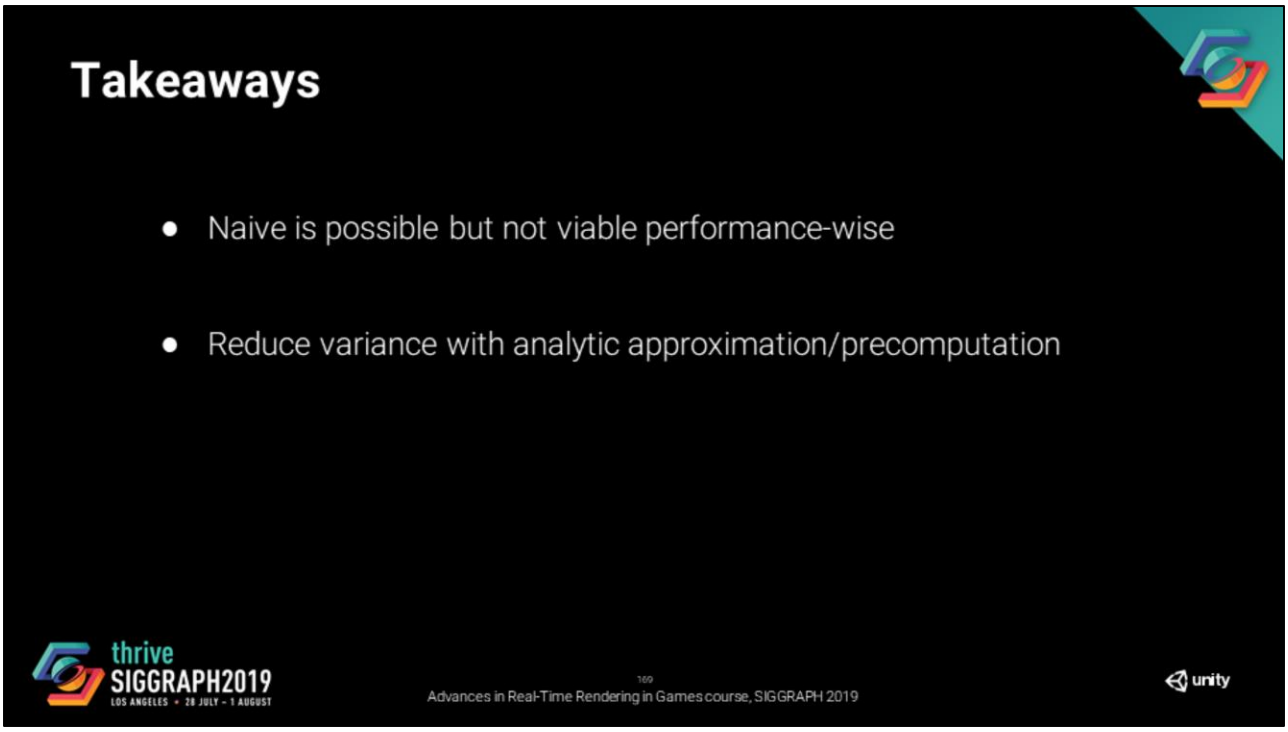

It is a significant improvement, but it is still quite expensive and there is room for improvement, either on the raytracing part itself or on the denoising.



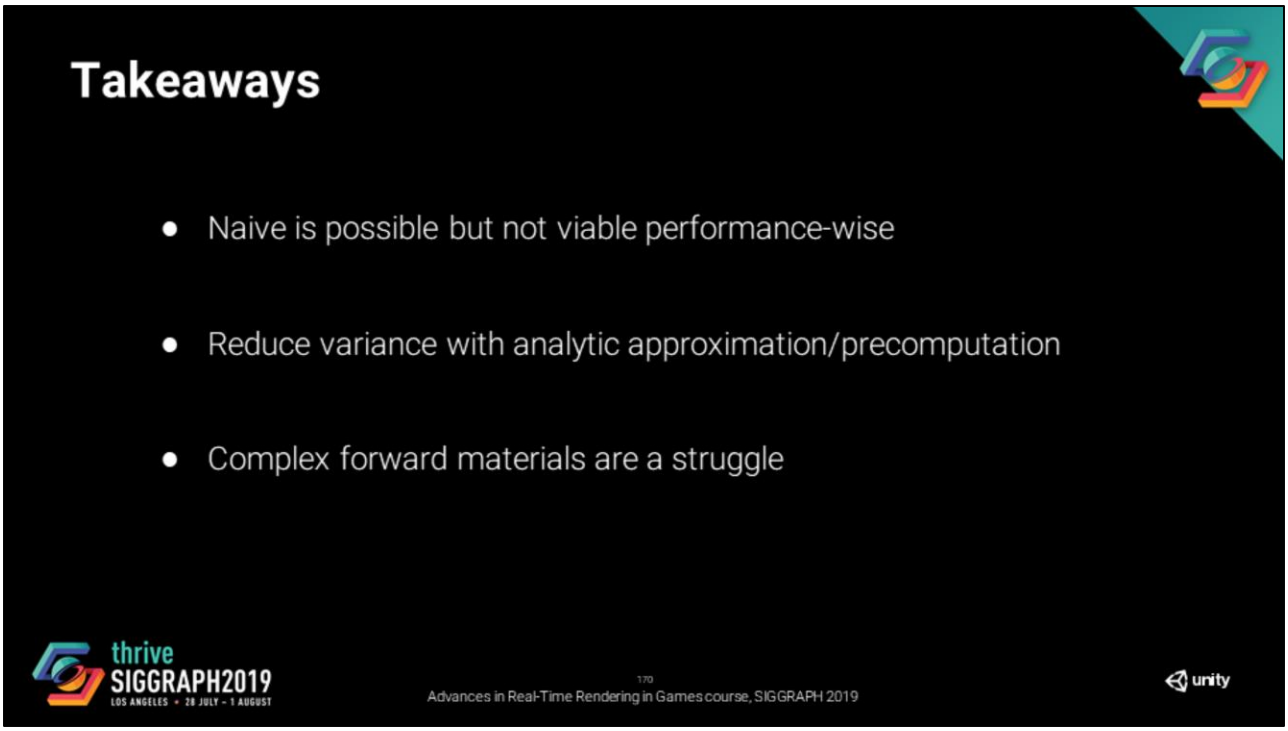

But again this scene is about 60 million polygons, which is quite heavy.


Alright, at this point I covered pretty much everything that I wanted to talk about. I'd like to do a recap of the lessons that we've learned and I think are worth sharing

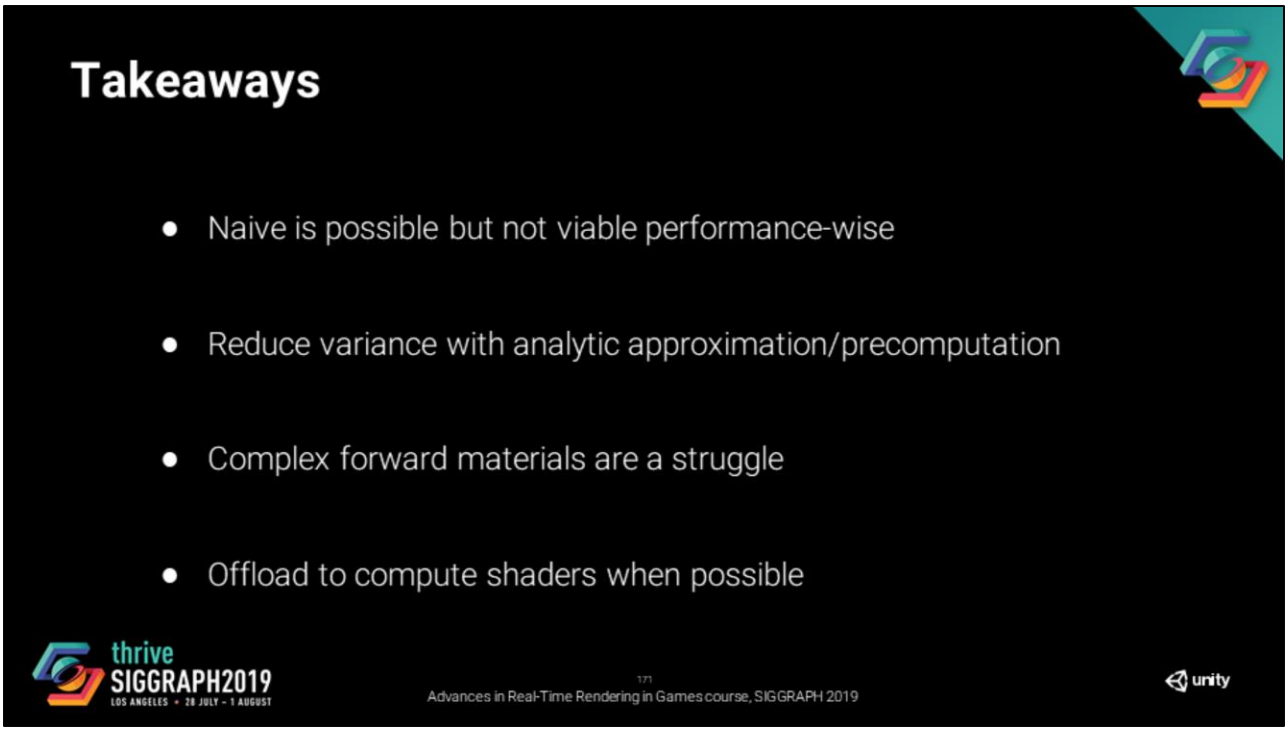

As you've seen through this presentation, the intuitive implementation is possible. However, it is not viable if you are trying to ship a game or a real-time application with content that is not trivial. It is important to measure everything that you do with this API, it can get very expensive quickly.



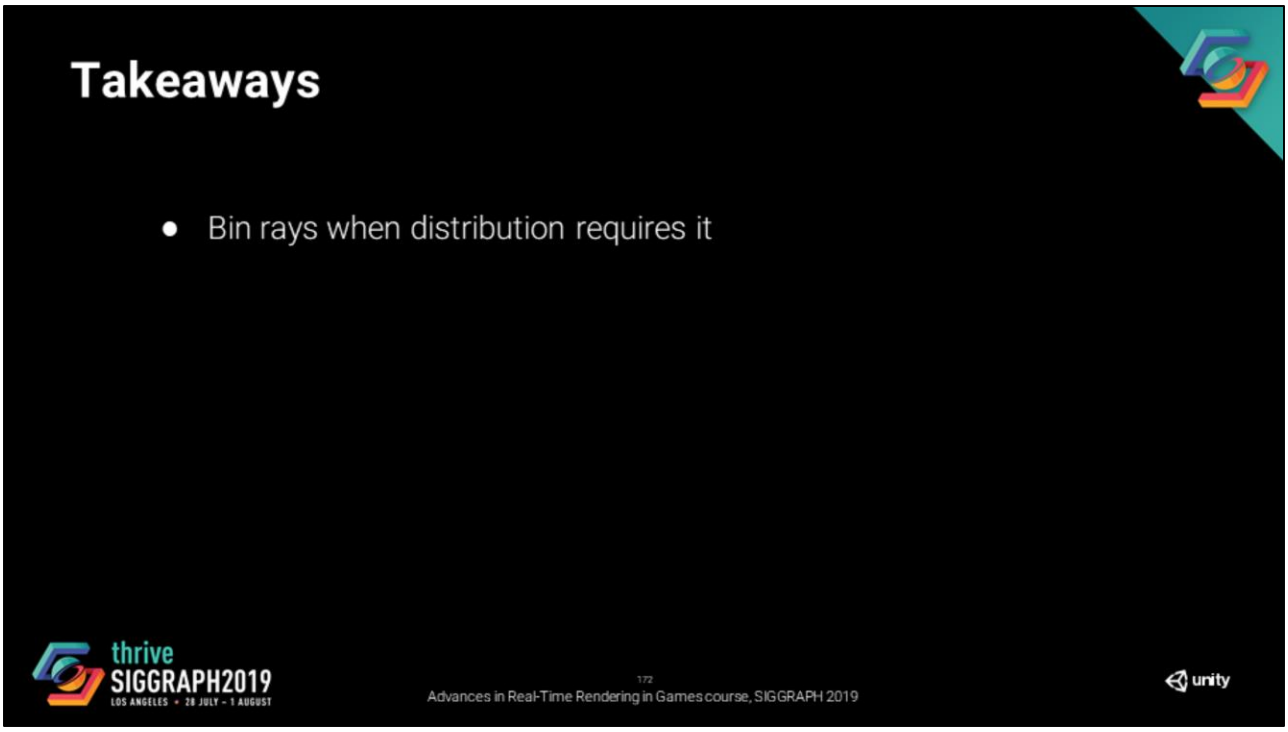

The second point is that it is very important to try to reduce the variance of whatever you are integrating, we've seen it through the two examples that I covered. You can use importance sampling, consistent estimators or even biased estimators if you make sure to compare your result to a reference.




If you have complex forward materials that will not fit into a gbuffer. It is going to be a struggle. I covered the solution that we settled for but there are more options.

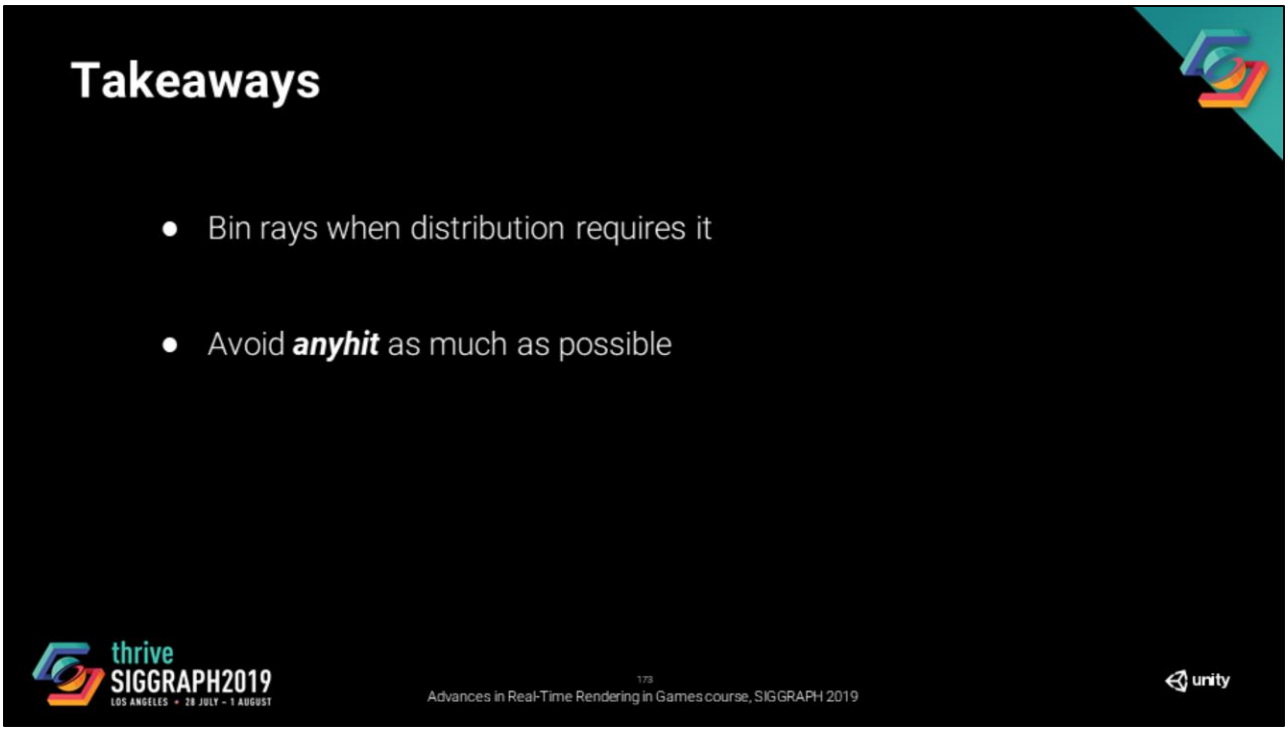



A good tip is that you should always think of offloading work to compute shaders. We've seen that either interfering with the ray traversal process is problematic, plus compute shaders have features like LDS that allow you to implement more efficently.



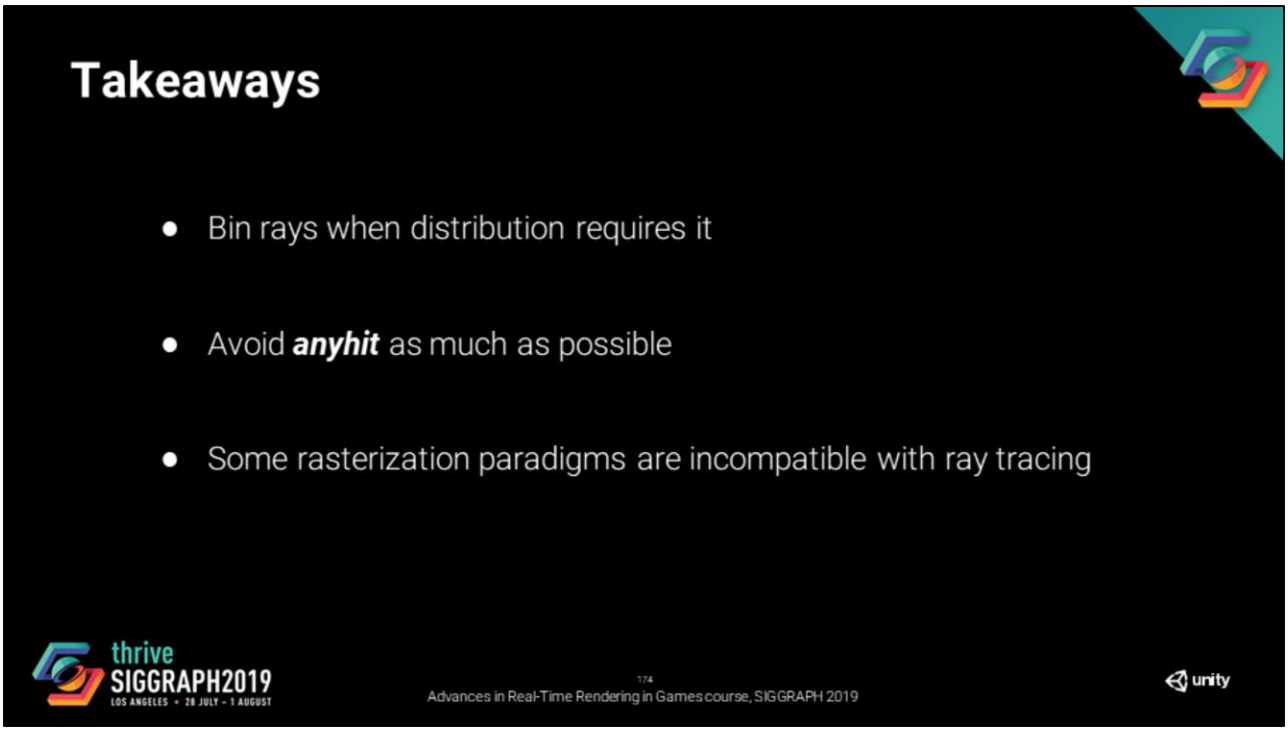

The cost of incoherent rays is a game changer. I presented our current approach, but I do not think that the problem is solved.

If you find a better way, please implement and share it!

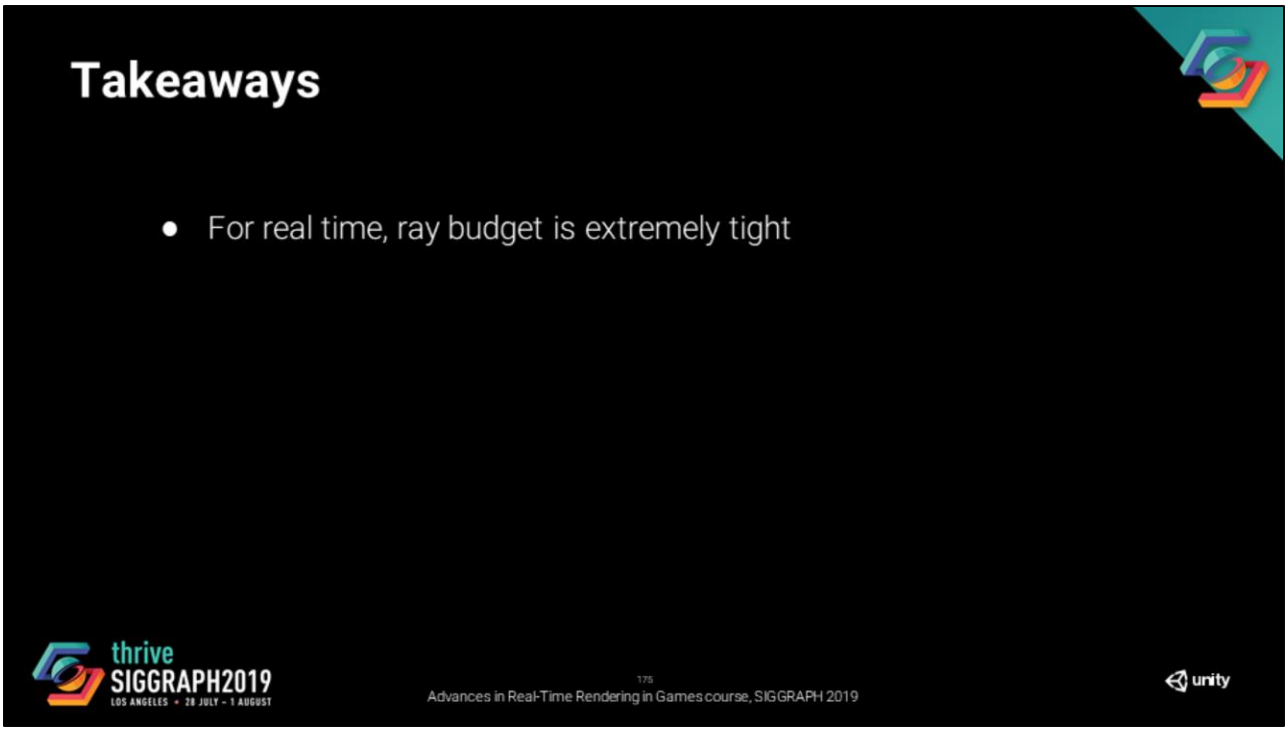



I am stating the obvious here, but any hit shaders are triggered for every potential intersection.

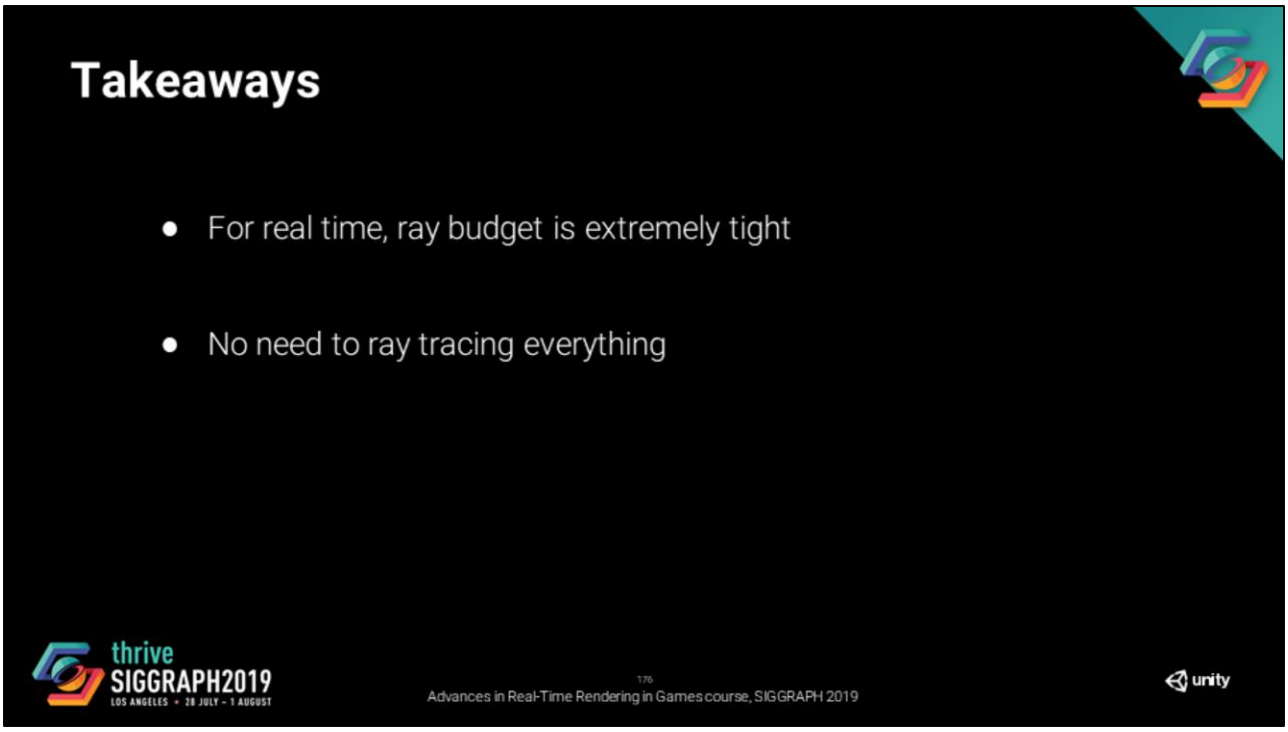
However, it is probably the most effective way to particle systems, cutout materials. We thus cannot pretened they do not exist.



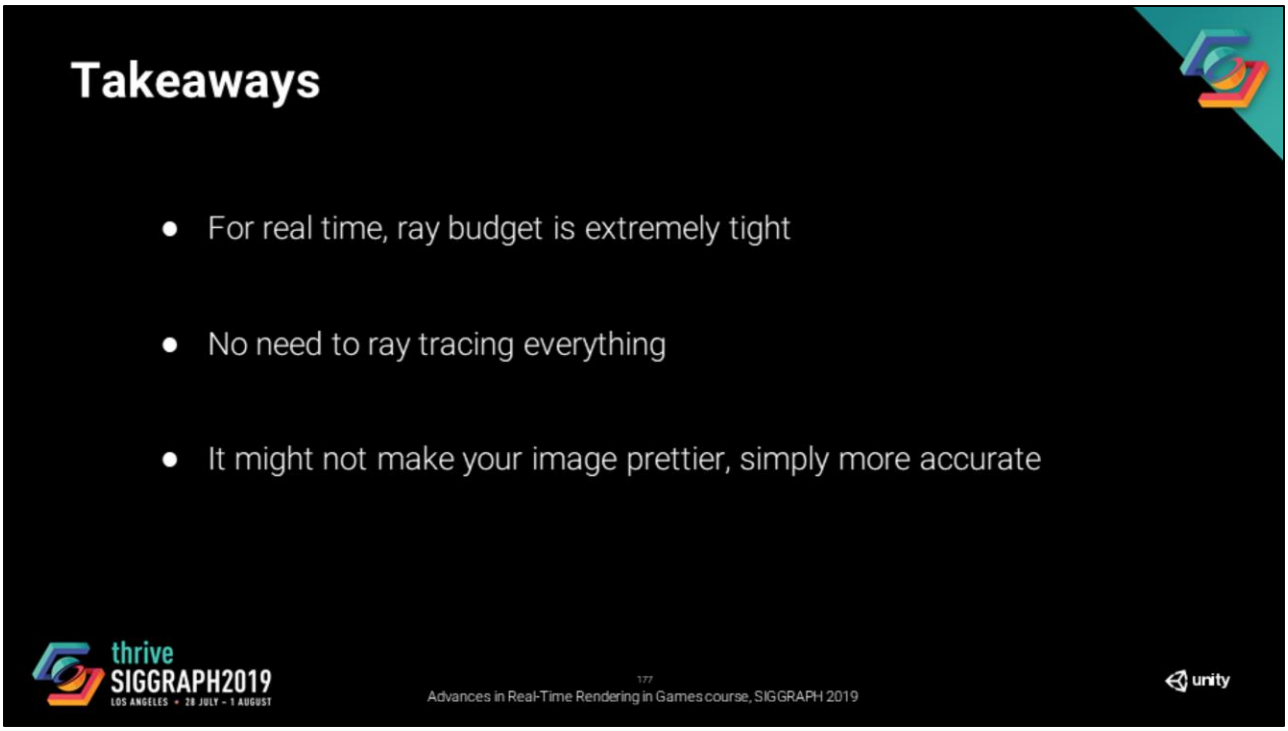

You are going to struggle to make ray tracing fit in your render pipeline. We had to and you will probably need to rethink part of the pipeline if you integrate it.




Ealier in the presentation, I said that our budget was about 1spp per effect. Depending on your content and deployment target, that is probably an over estimation (especially if you are going for multiple effects). Thus, you will spend a lot of energy trying to trace less rays.

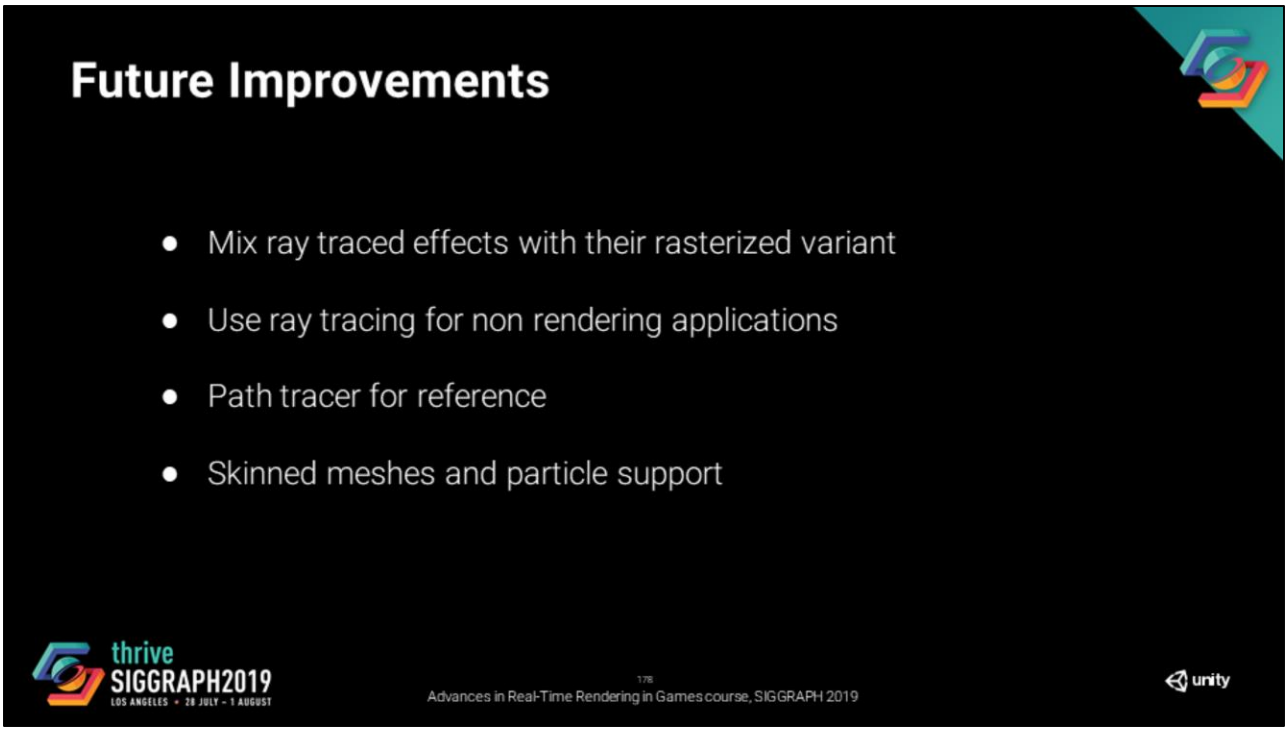



Ray tracing is fun, but ray tracing is not the answer to everything. To take the example of lighting an indirect point, having shadow maps is a huge advantage, no need to spend additional computing time fetching that information.




Sometimes, ray tracing is not worth the trouble.

If used correctly it may make your image more accurate, but it won't necessarly be prettier. Make sure that it fits the concept arts of what you are trying to display.




I would like to finish with the things that we are, or will be, working on in the coming months.

We want to explore the ability to mix effects with rasterization variant to reduce the cost of effects

SSR with indirect specular

SSGI with indirect diffuse

Contact shadows with ray trace shdows.

We want to expose ray tracing for non rendering applications

We are currently working on a path tracer for reference

And we will be working on skinned mesh and particle support.

# Acknowledgement

Unity Graphics Team
Sebastien Lagarde (Unity)
Kate McFadden (Unity)
Dany Ayoub (Unity)
Alexandre Cepisul (L&S)
Awen Couellan (L&S)
Aymeric du Chéné (L&S)
Mike Geig (Unity)
Emmanuel Turquin (Unity)
Natasha Tatarchuk (Unity)
Laurent Harduin (Unity)
Ionut Nedelcu (Unity)
Arnaud Carré (Unity)
Joel de Vahl (Unity)
Tim Cooper (Unity)
Eric Heitz (Unity)
Francesco Cifariello Ciardi (Unity)
Antoine Lelievre (Unity)
Cyril Jover (Unity)
Lewis Jordan (Unity)
Jesper Mortensen (Unity)
Tian Ning (Unity)
Melissa Chou (Unity)




# Questions

https://github.com/Unity-Technologies/ScriptableRenderPipeline

# Indirect Specular

| Small Room Scene - Roughness override 1.0 - 1SPP - Min Smoothness 0.0 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Test ID | Deferred | Binning | Tile Size (resolution) | Bin Resolution | Ray Binning Pass (ms) | Ray Trace (ms) | Deferred Shading (ms) | Effect Cost (ms) |
| Base | Off | Off | 0 | 0 | 0 | 38 | 0 | 38 |
| A0 | On | Off | 0 | 0 | 0 | 7.3 | 1.44 | 8.74 |
| A1 | On | On | 16 | 16 | 0.141 | 5.8 | 1.437 | 7.378 |
| A2 | On | On | 32 | 16 | 0.126 | 5.3 | 1.44 | 6.866 |

| Small Room Scene - Roughness override 1.0 - 1SPP - Min Smoothness 0.0 | | | | |
|---|---|---|---|---|
| Test ID | Type | Ray Trace (ms) | Deferred Shading (ms) | Effect Cost (ms) |
| Base | Default | 38 | 0 | 38 |
| A0 | Prepass + SemiDeferred | 17 | 0 | 17 |
| A1 | Prepass + Deferred | 7.5 | 1.4 | 8.9 |

# Indirect Specular

| Small Room Scene - Variable Roughness - 1SPP - Min Smoothness 0.6 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Test ID | Deferred | Binning | Tile Size (resolution) | Bin Resolution | Ray Binning Pass (ms) | Ray Trace (ms) | Deferred Shading (ms) | Effect Cost (ms) |
| Base | Off | Off | 0 | 0 | 0 | 8.6 | 0 | 8.6 |
| A0 | On | Off | 0 | 0 | 0 | 1.961 | 0.633 | 2.594 |
| A1 | On | On | 16 | 16 | 0.132 | 1.8 | 0.634 | 2.566 |
| A2 | On | On | 32 | 16 | 0.1085 | 1.69 | 0.637 | 2.4355 |