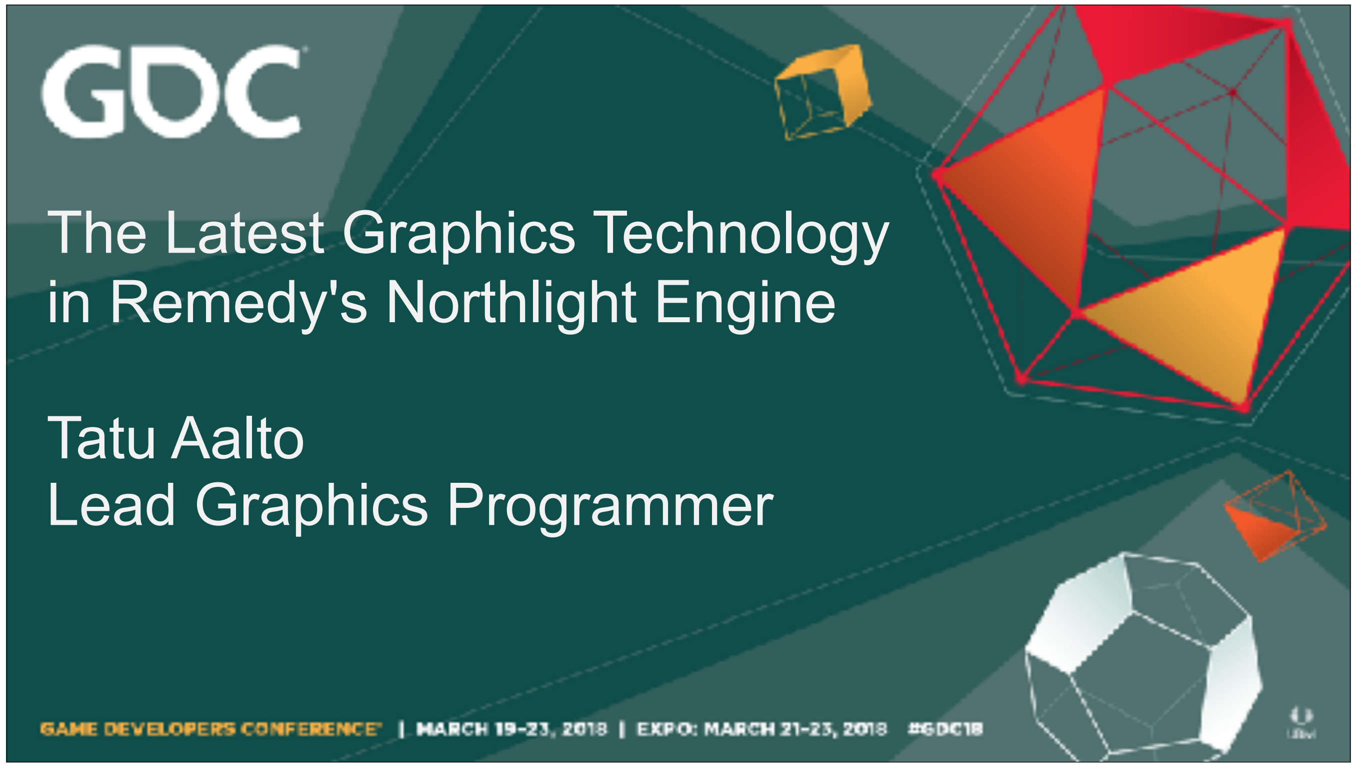
The Latest Graphics Technology in Remedy's Northlight Engine

Tatu Aalto
Lead Graphics Programmer



GDC

The Latest Graphics Technology in Remedy's Northlight Engine

Tatu Aalto
Lead Graphics Programmer

GDC

~~The Latest Graphics Technology in Remedy's Northlight Engine~~

Tatu Aalto
Lead Graphics Programmer

GAME DEVELOPERS CONFERENCE® | MARCH 19-23, 2018 | EXPO: MARCH 21-23, 2018 #GDC18

Experiments with DirectX Raytracing in Remedy's Northlight Engine

Tatu Aalto
Lead Graphics Programmer

Remedy Entertainment plc.

Quantum Break

Alan Wake

Max Payne

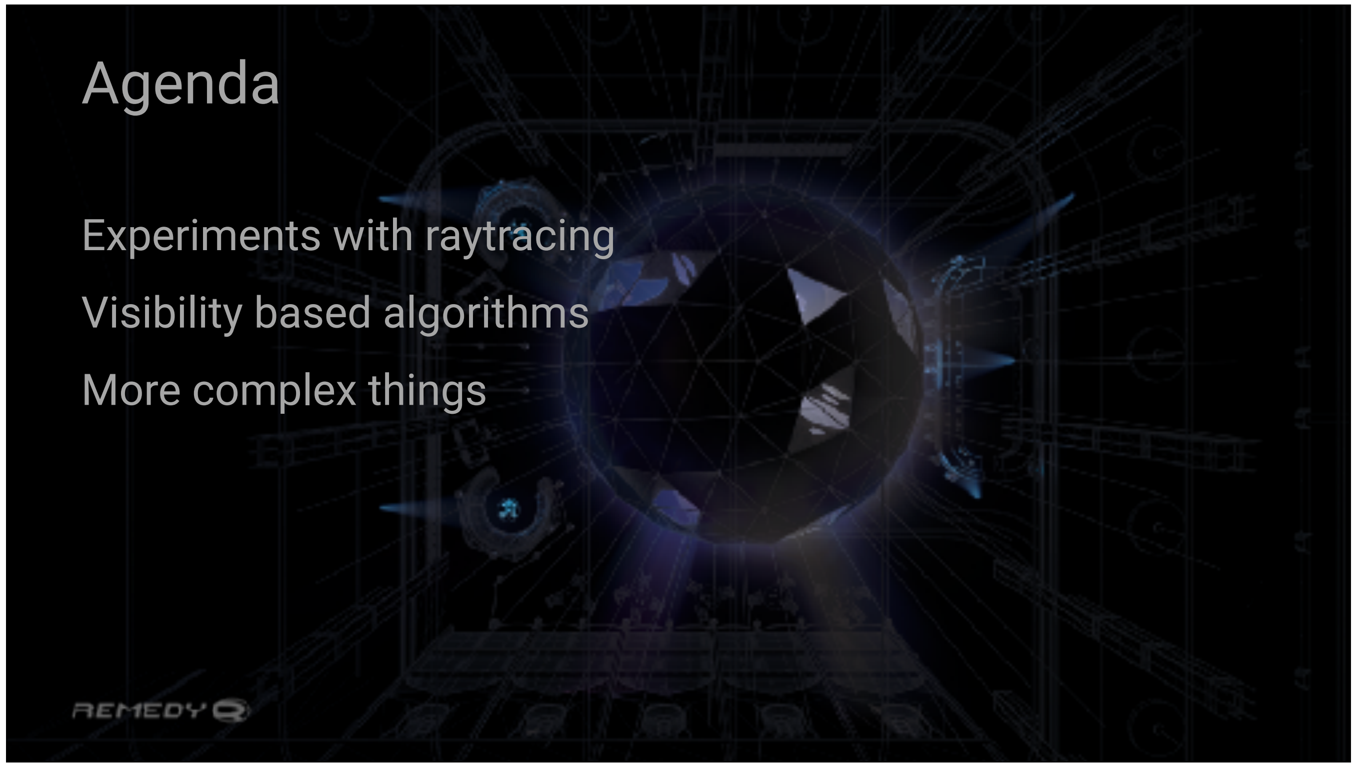
Death Rally

P7

CrossFire 2

Company intro here

Demo video



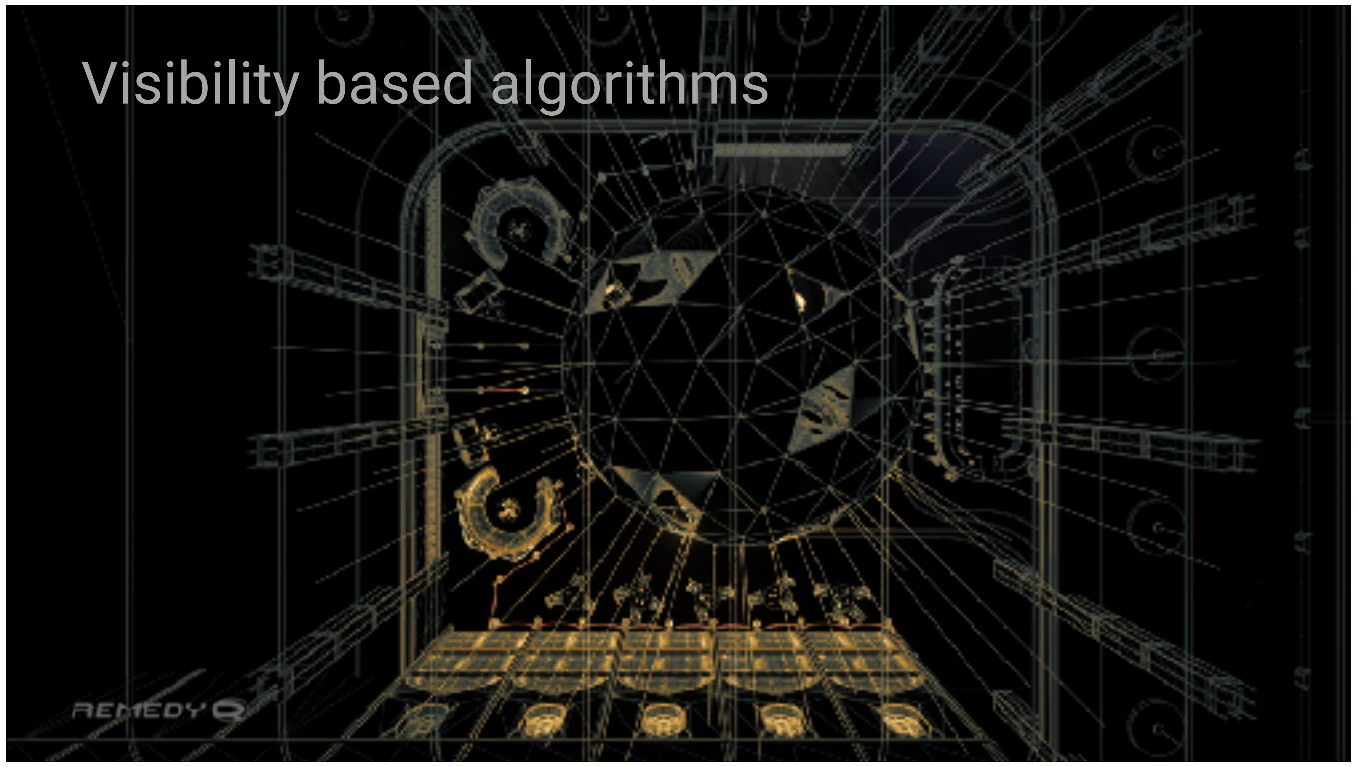
Agenda

Experiments with raytracing

Visibility based algorithms

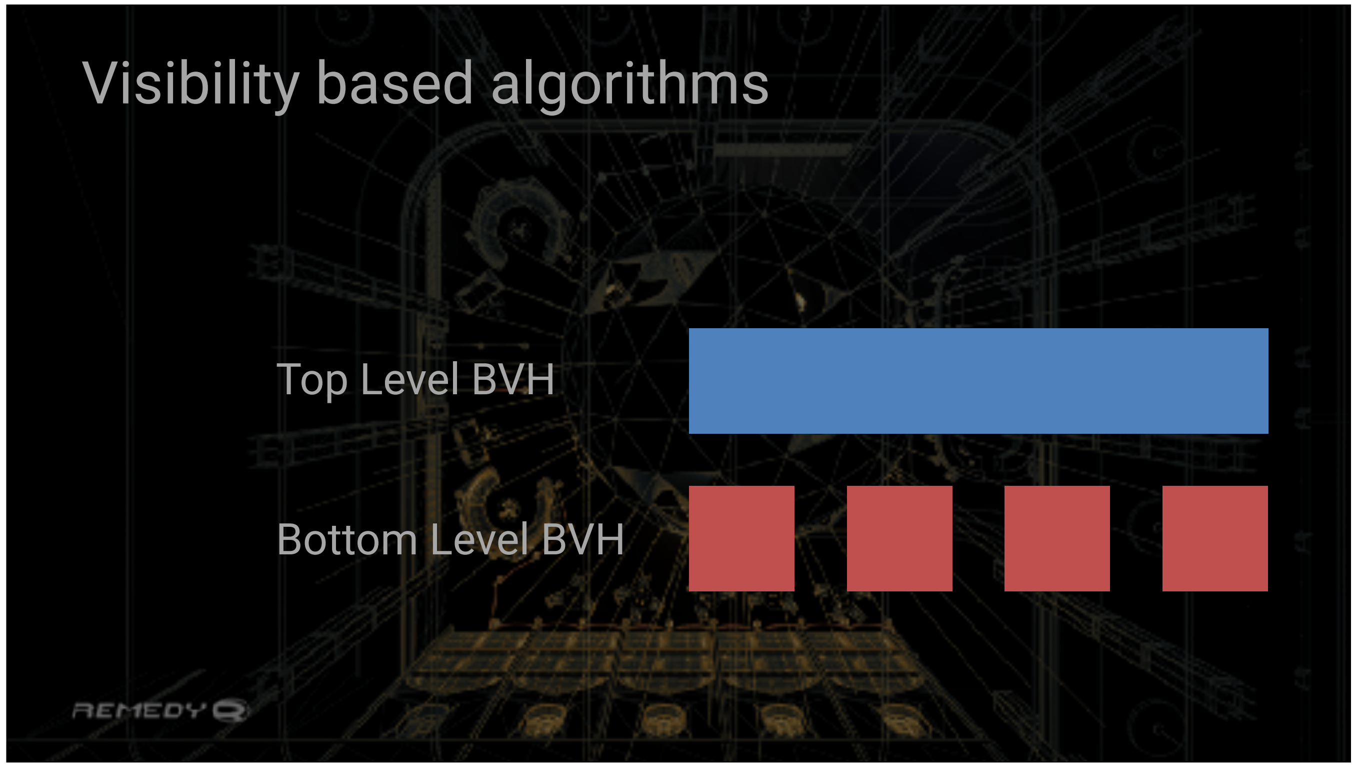
More complex things

My agenda on this presentation will be pretty straight forward. I’ll do a very quick introduction into high level DirectX raytracing concepts, and then I’ll walk you through couple examples on what we have tried out and some of the findings on what to expect when exploring. We’ve started of by targeting to make a reference implementations of existing algorithms by using raytracing. The hope is, that this will help us refining and balancing the looks of rasterisation algorithms when targeting console platforms for example.



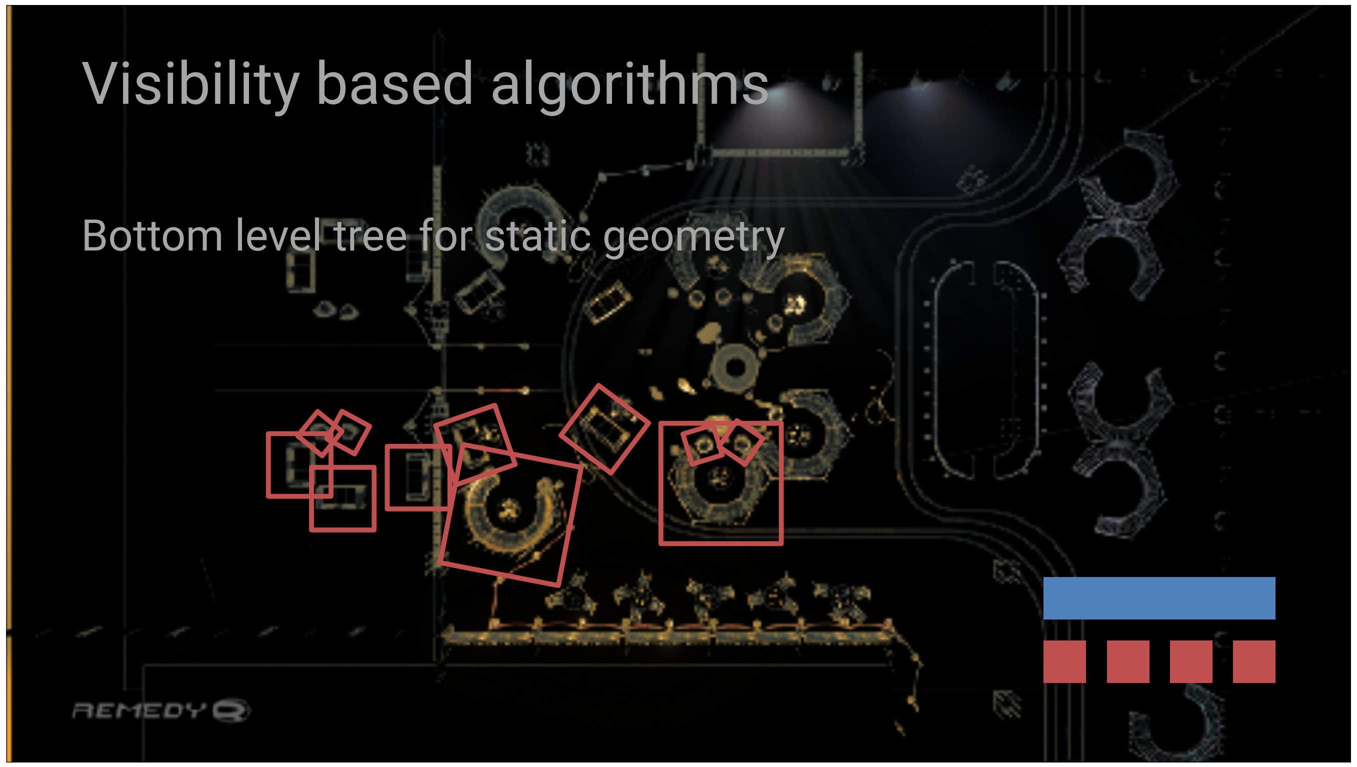
Visibility based algorithms

Let’s start with very high level introduction on DirectX raytracing. I will not go through the API in this presentation, but I’ll try to give you some background on what you need to do before you can start shooting rays.



Visibility based algorithms

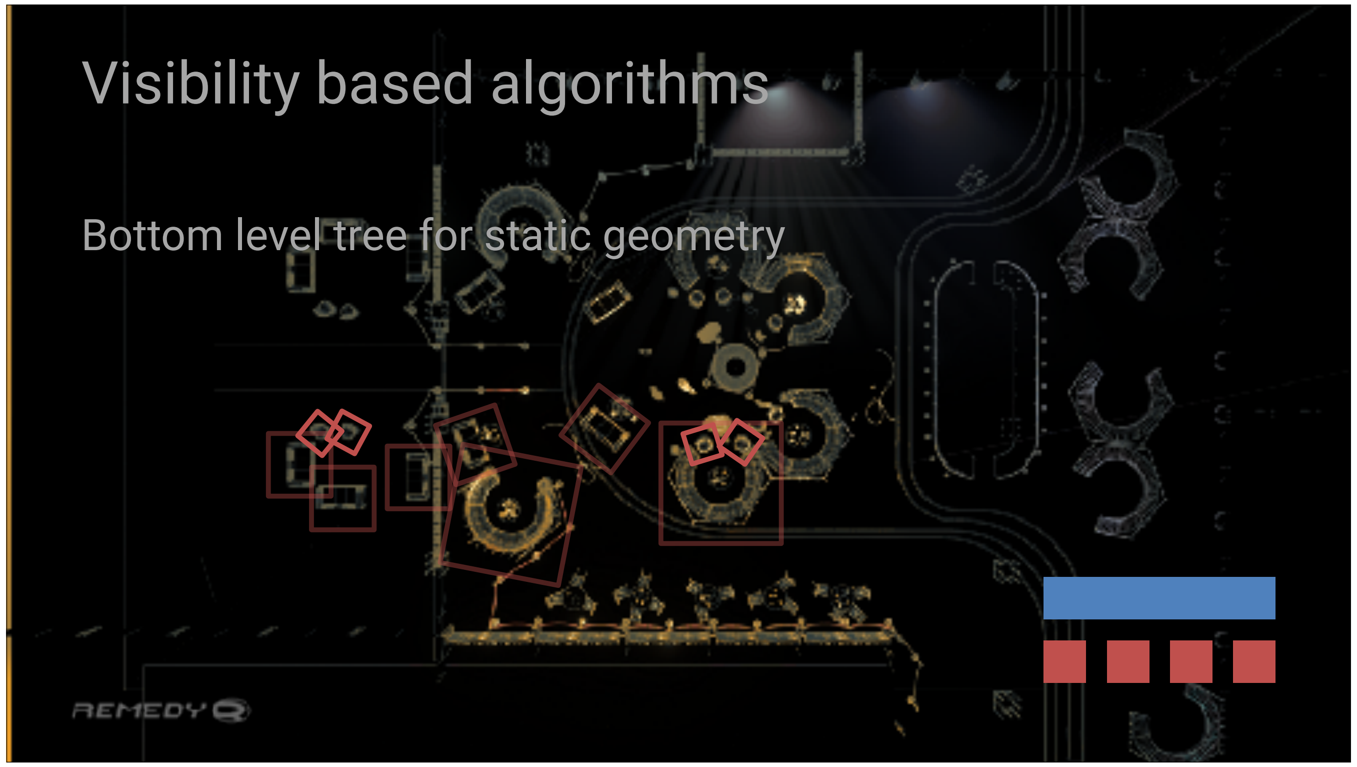
Ray traversal API is built on top of two different high level acceleration structures, Bottom and Top level bounding volume hierarchies. The idea is, that bottom level hierarchy is constructed to hold geometry, while the top level contains bottom level trees. Simple way to think about this is that each mesh should be a single bottom level tree. And top level tree is your scene that contains bunch of bottom level tree instances with some transformation.



Visibility based algorithms

Bottom level tree for static geometry

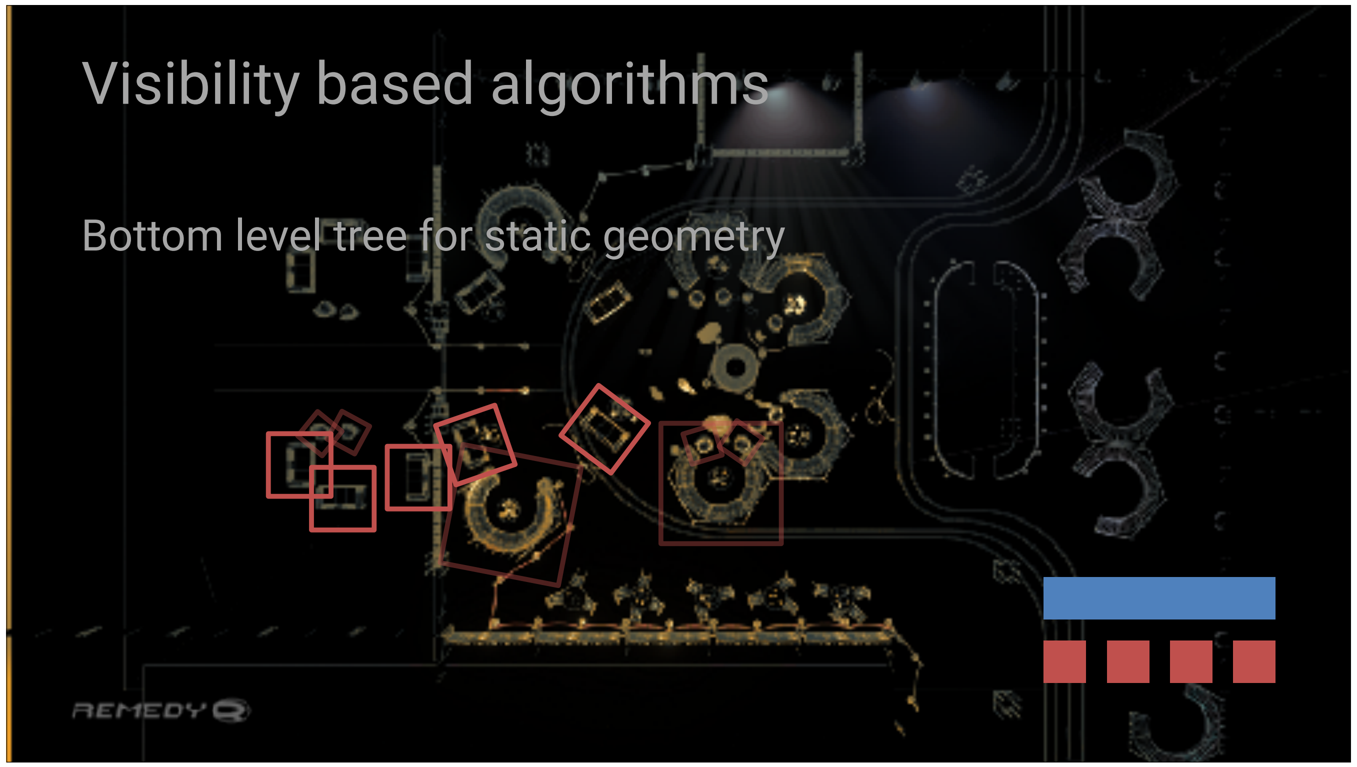
The first thing you are going need is the bottom level bounding volume hierarchy for the static parts of your scene. In this picture, each red square represent bounds of a single bottom level tree. Building bottom level tree is very simple. In API, you'll have a function, that takes normal DirectX GPU geometry and index buffers, and returns bottom level tree build around the geometry. Building is done on GPU, and the algorithms are hidden from the user. For the scene you saw in the demo, building of all the trees takes roughly XX ms. This is for a scene that hold XX geometry instances with XX triangles.



Visibility based algorithms

Bottom level tree for static geometry

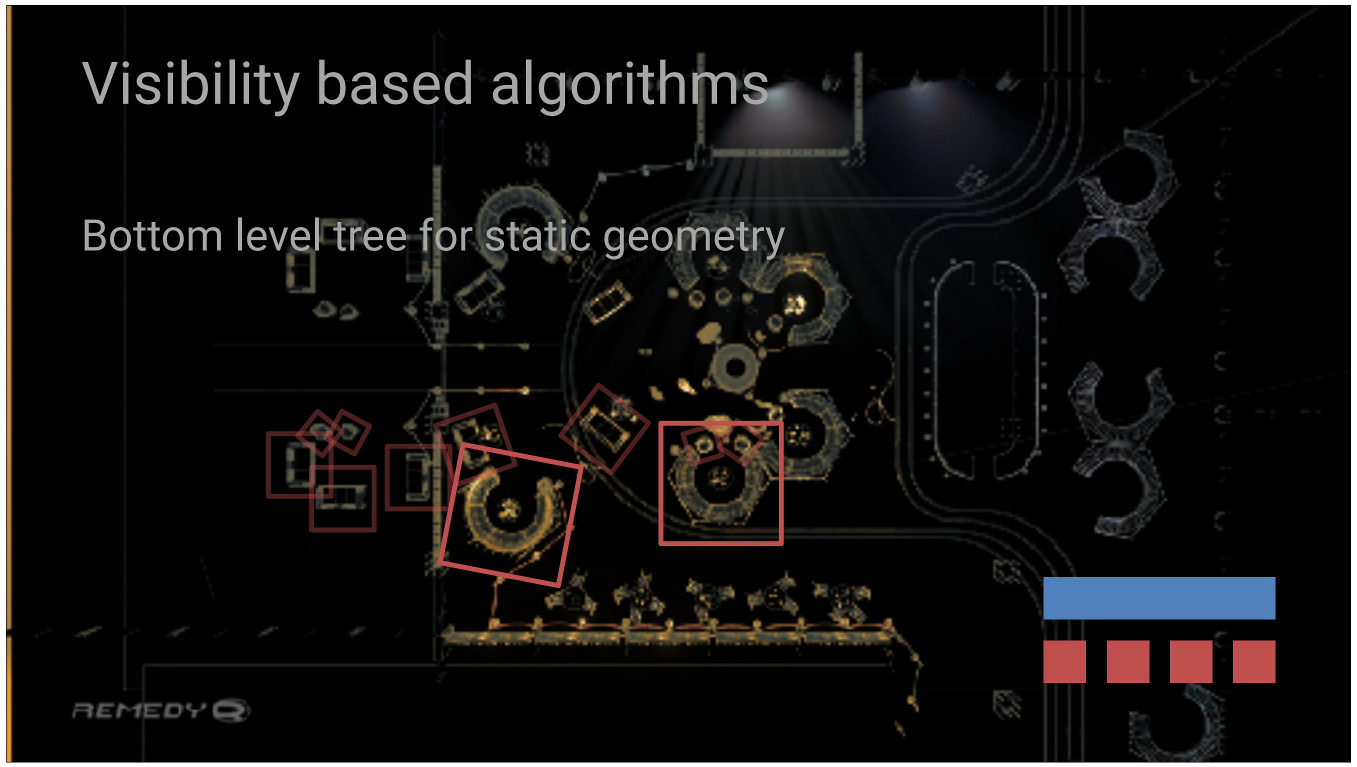
So one thing to note here. The highlight I've picked in this picture, contain only tree unique bottom level trees. These three trees can be instanced multiple times around the scene with different transformations. These smallest boxes are copies of the same chair mesh.



Visibility based algorithms

Bottom level tree for static geometry

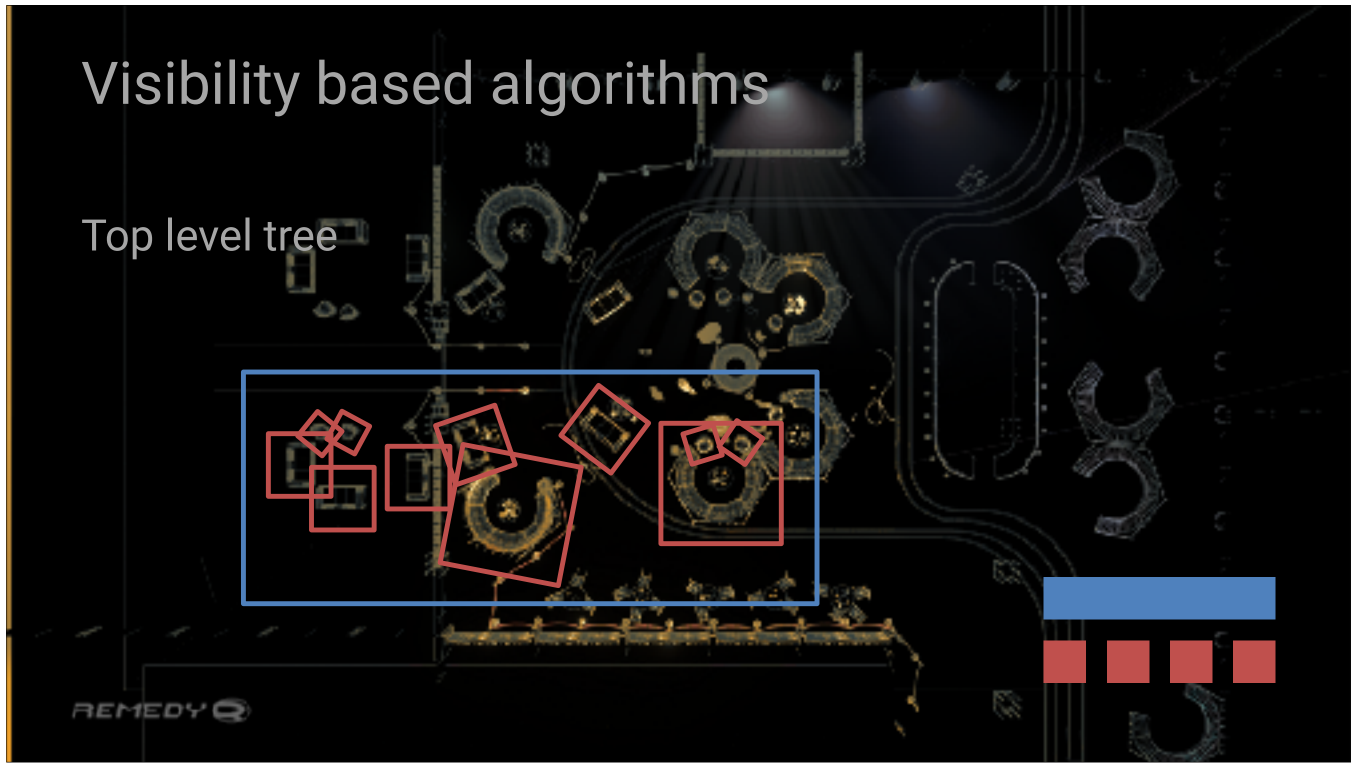
The mid size trees are small sofas.



Visibility based algorithms

Bottom level tree for static geometry

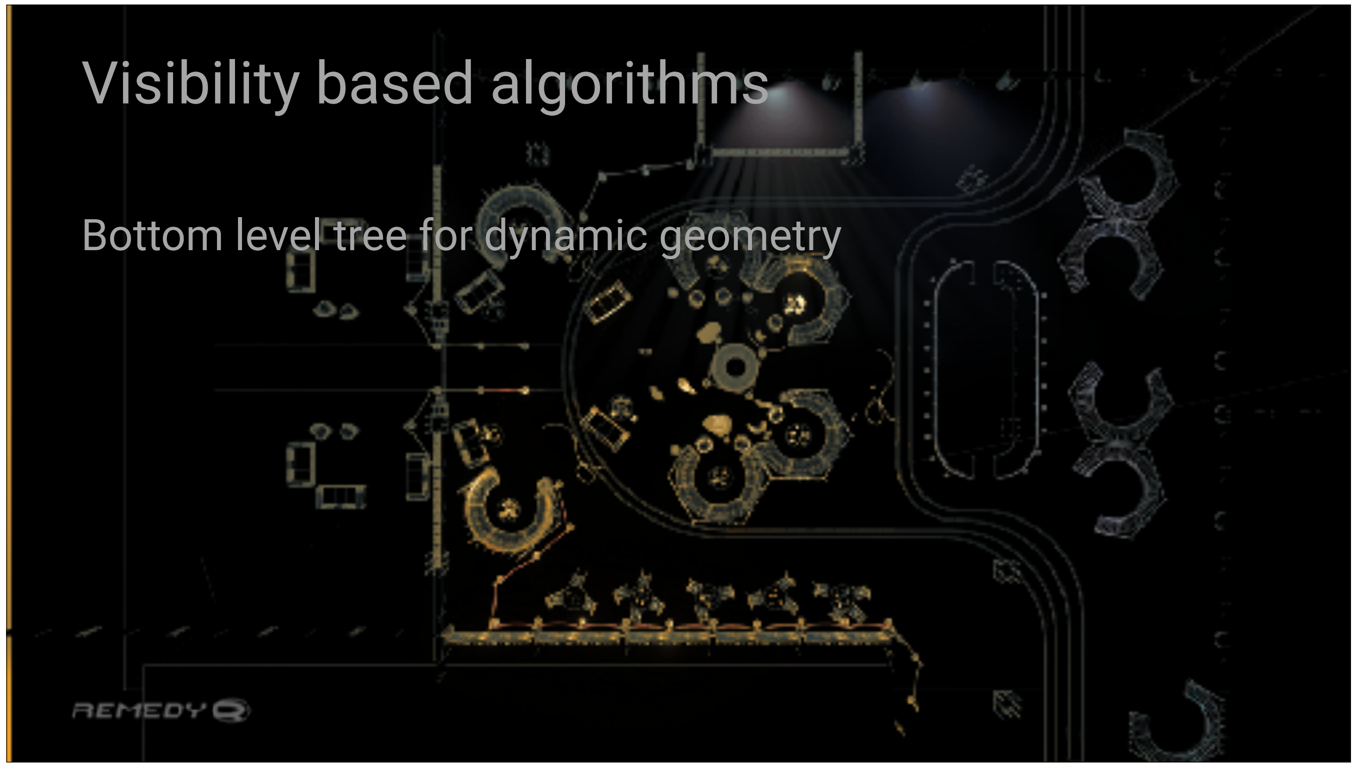
And the biggest ones are large round sofas.



Visibility based algorithms

Top level tree

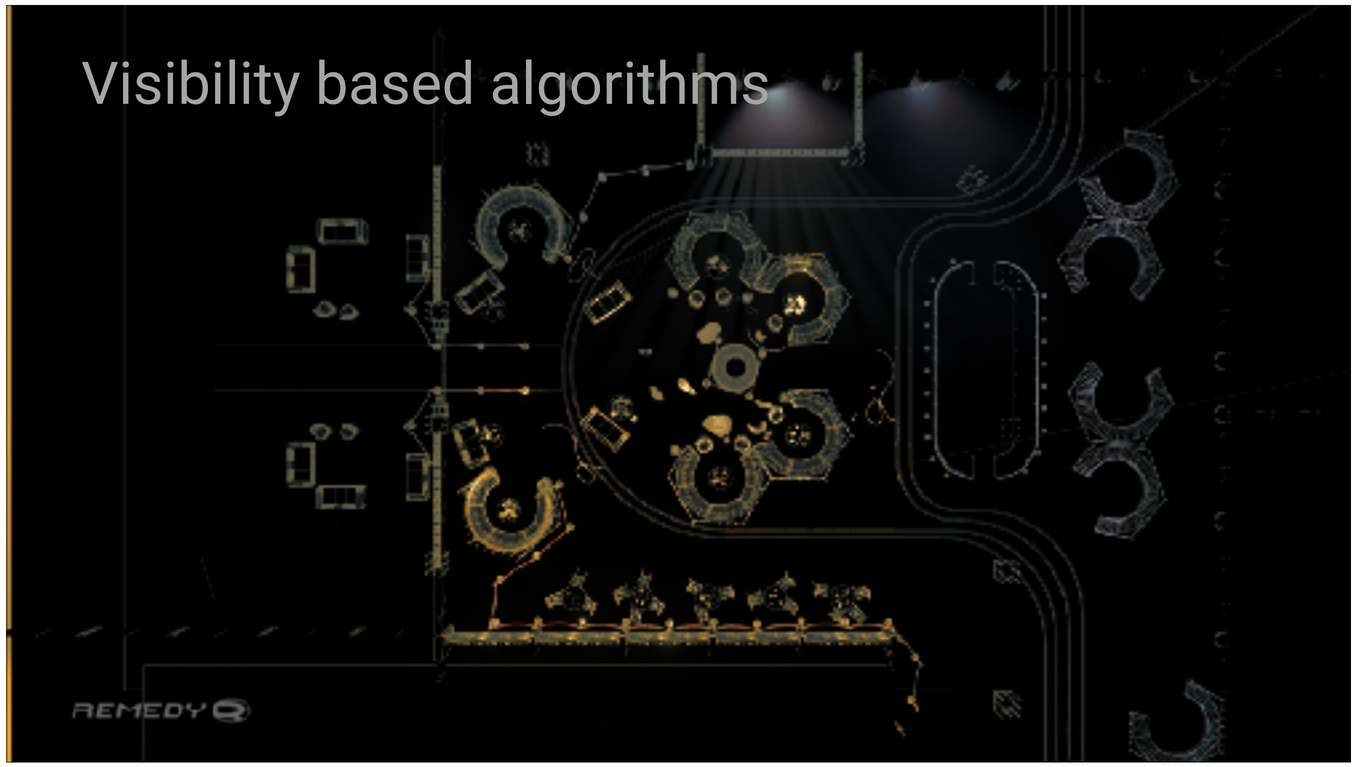
In order to build a scene for raytracing, you'll need to insert these bottom level trees into a top level tree. This is again very simple thing to do. The API provides you a function that takes in a number of bottom level tree instances with transformations. Building a top level tree is supposed to be a lot faster than bottom level trees. Intent is that you can move the objects as you will, and build a top level tree from scratch every frame. In our demo level, building a top level tree takes roughly XX ms



Visibility based algorithms

Bottom level tree for dynamic geometry

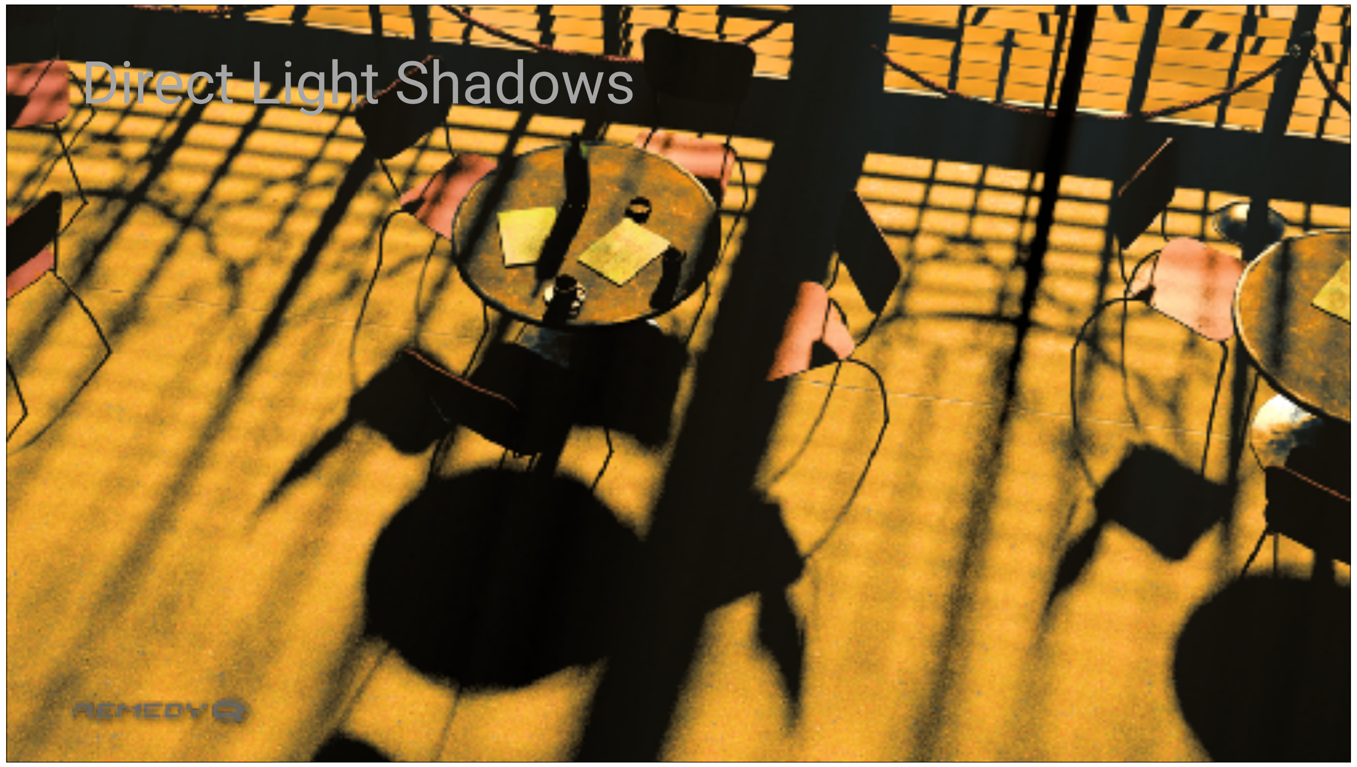
Deformed geometry like skinned character needs special handling, as you can’t deform bottom level tree. You will need to first create buffer that contains deformed geometry, and then feed that into tree builder API. Many engines run with such config by default, but we haven’t been using pre-deformed geometry, so we needed to add support for this. Quickest way to get it done for us was to create simple compute shader that calculates skinning.



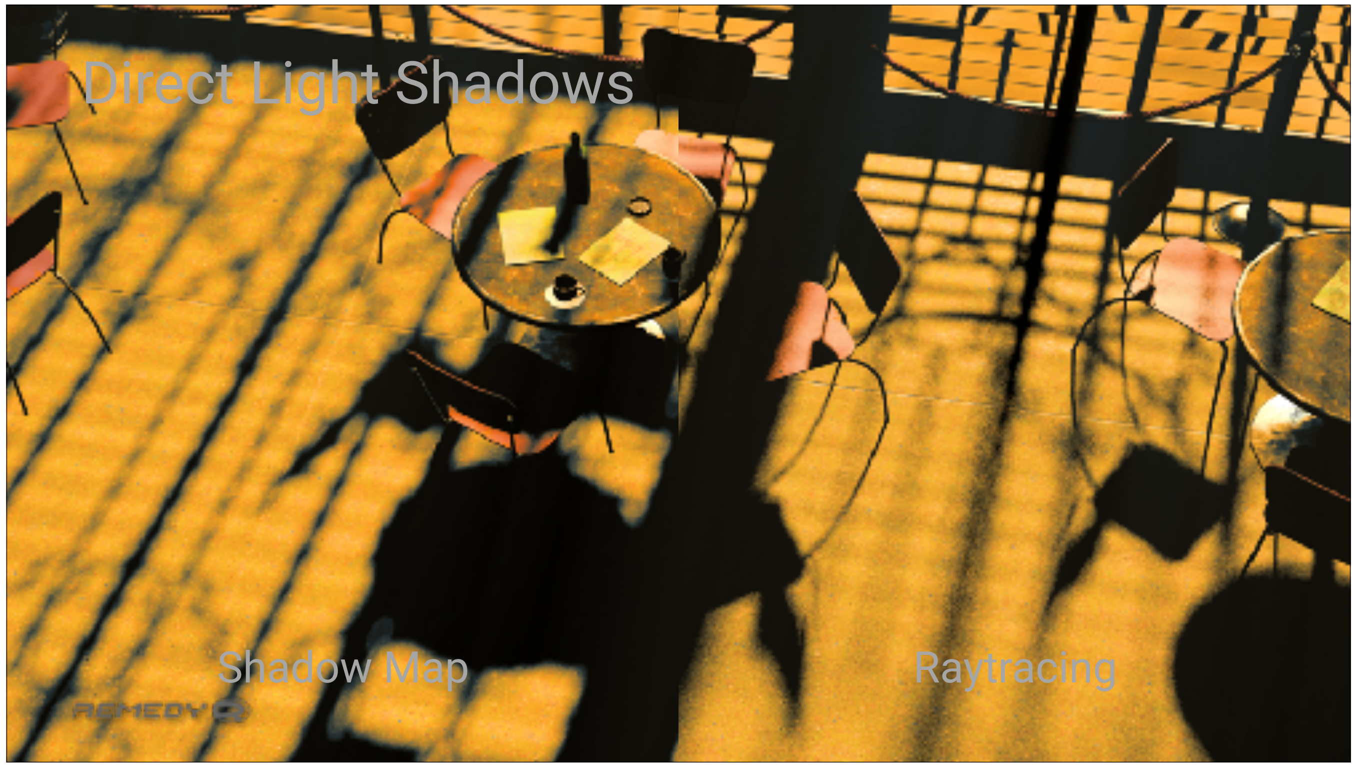
Visibility based algorithms

That hopefully gives you a rough picture on how the API works. Lets get into use cases that can be implemented after getting basic ray traversal scene built.

Direct Light Shadows

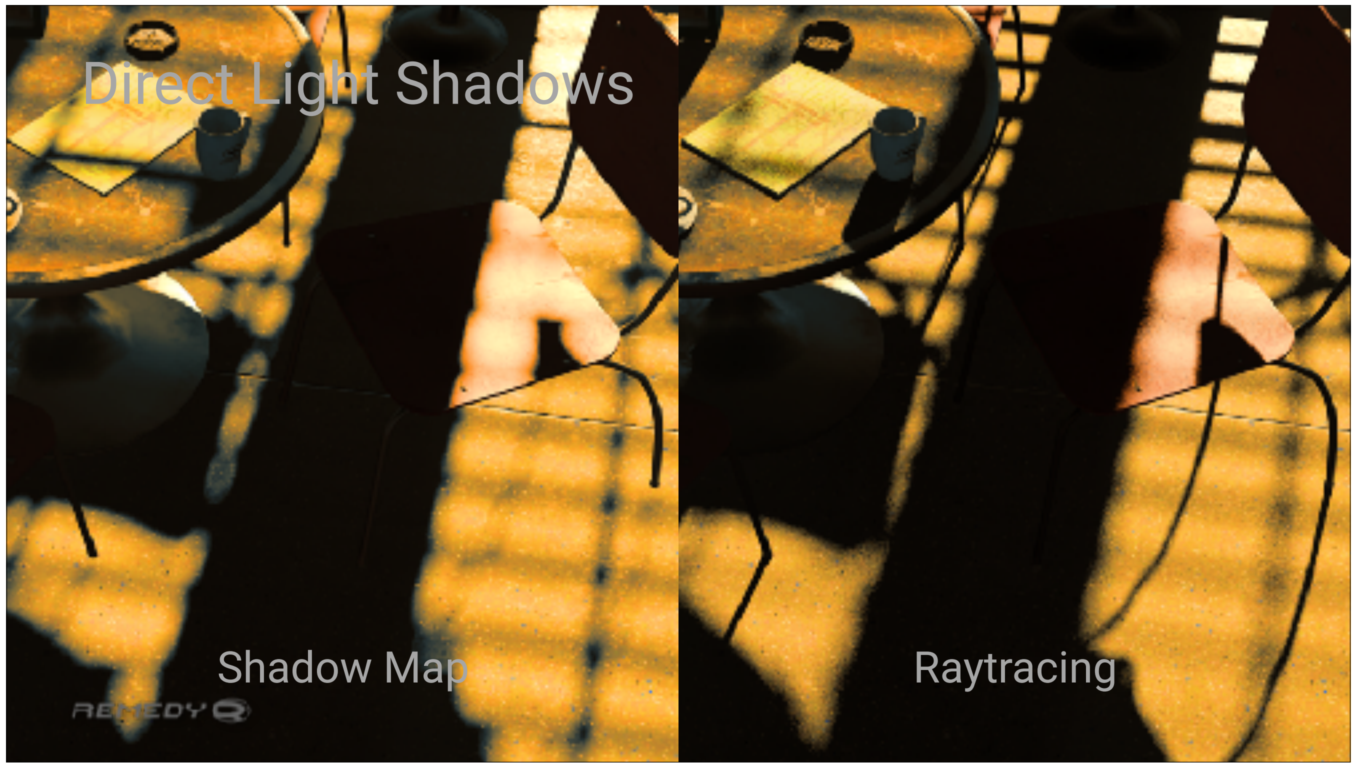


After getting basic data filled, you can start experimenting with raytraced algorithms like shadows. In this picture, we have replaced sunlight cascaded shadow maps with shooting rays. If you happen to fill directional light shadows into fullscreen buffer for filtering like us, it’s very easy to just replace cascade map lookup shader with raytracing kernel that writes into the same texture.



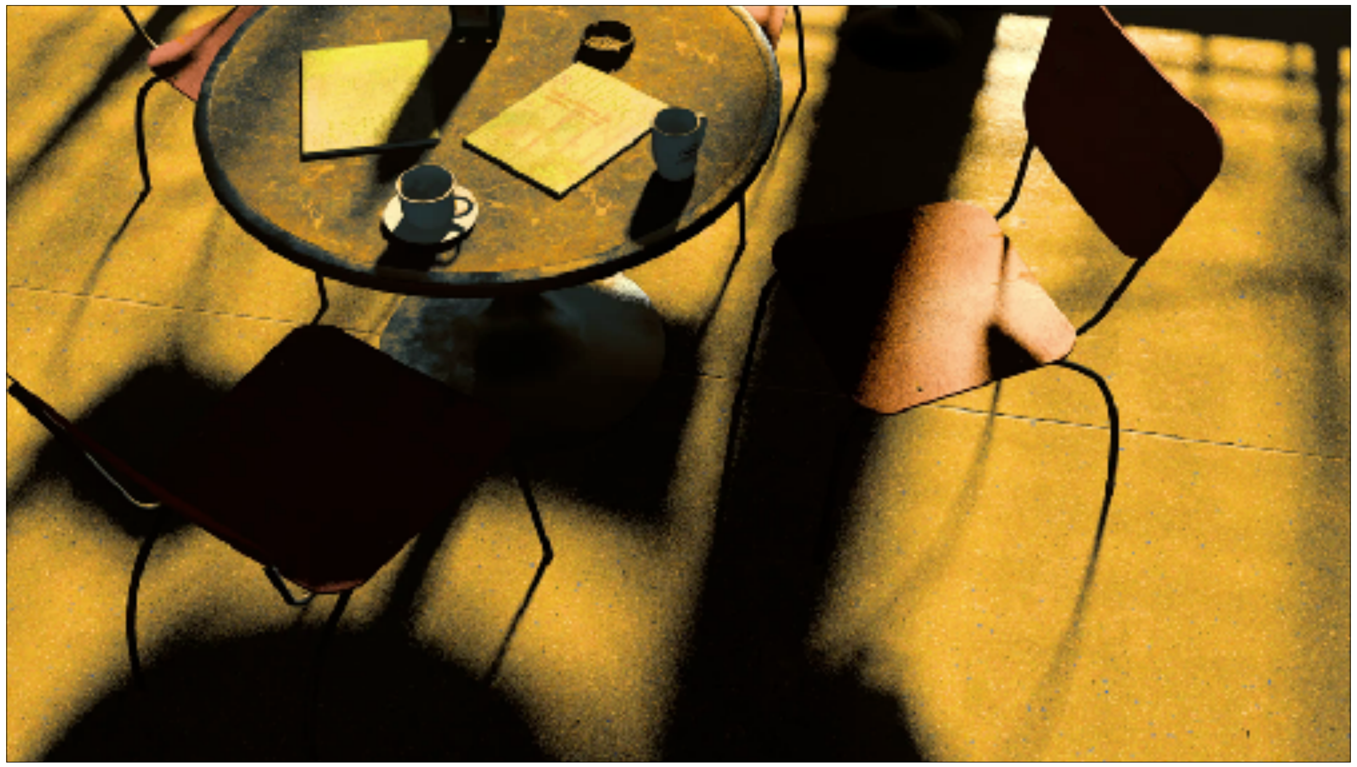
Here you can see a comparison using shadow maps on the left side, and raytraced shadows on the right side. In order to be fair, I must note that the left side hasn’t been crafted to absolute best quality you can achieve with shadow maps. This is just something we get out of engine with default good quality settings. Anyway, the amount of detail gained with accurate shadows is pretty remarkable.

Direct Light Shadows

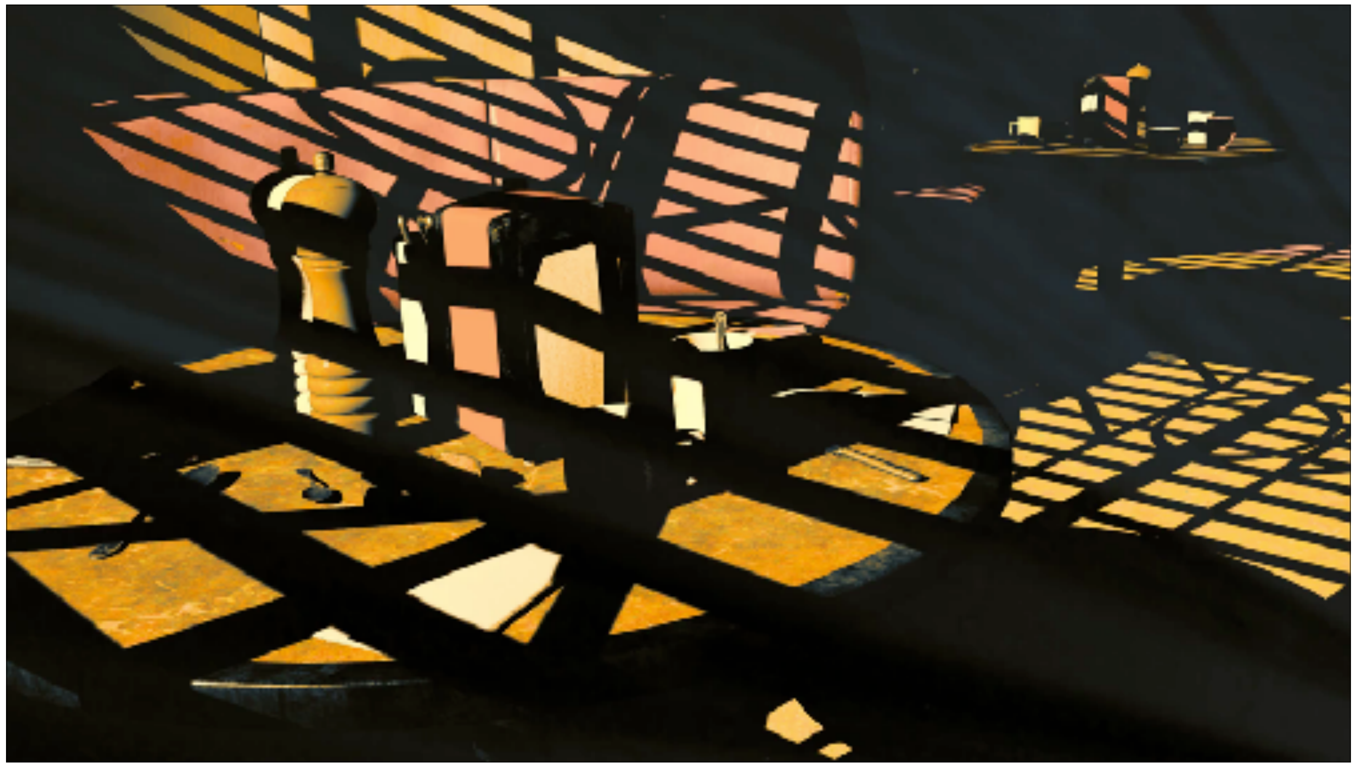


Here you can see closeup comparison between shadow maps and shadows with raytracing. Image with raytracing is using eight samples per pixel and does not have spatial filtering. Shadow map image on left side, is using 16 PCF samples and spatial filter applied on top of the screen space buffer.

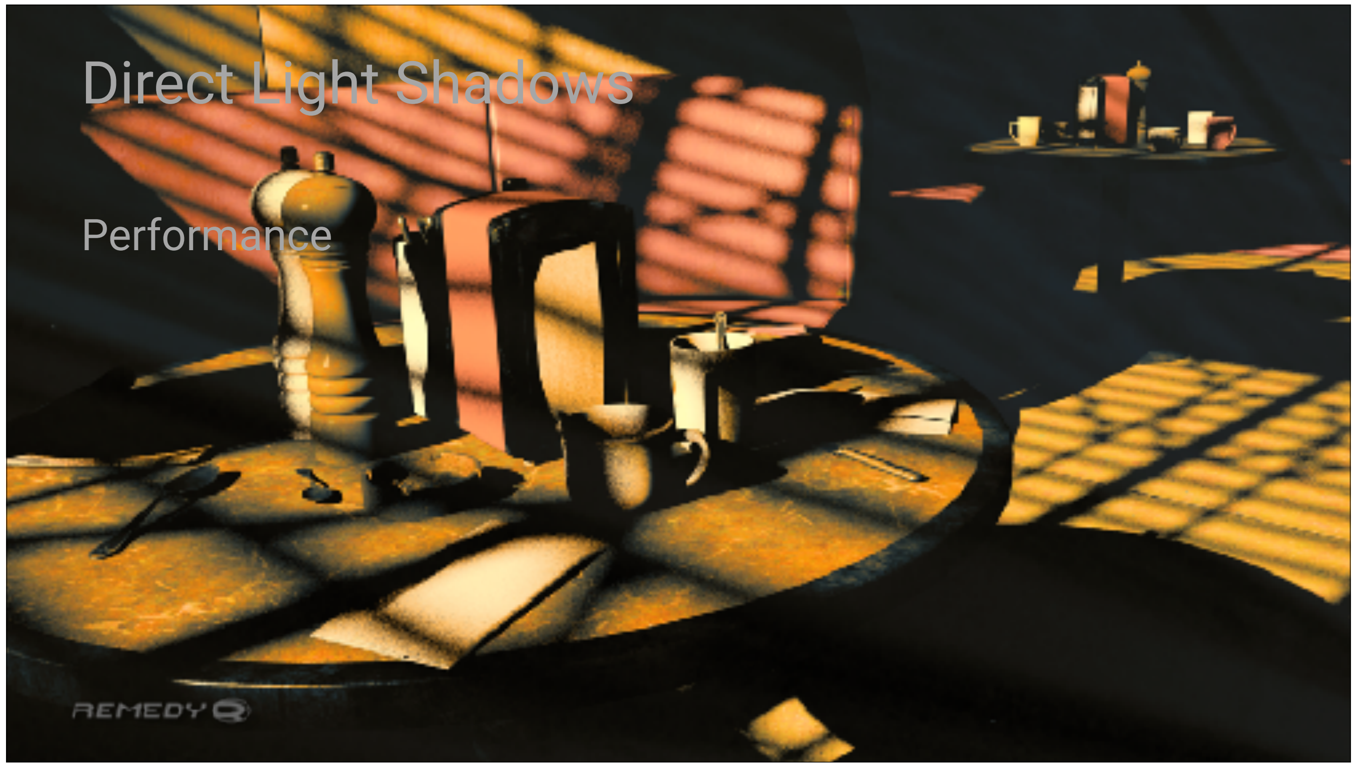
In addition to accuracy, it’s easy to do real area shadows with raytracing. Clip here is again using eight samples. Sampling pattern is basic blue noise, that is changing between frames, over sequence of four uncorrelated patterns. For each sample on the pixel, we offset the same blue noise with sobol sequence and wrap the random values on disk.



And here is the closeup I showed you earlier with animated area shadows. As you can expect with raytracing, contact points are accurate, and you get nice mix of sharp and soft shadows. While this is nothing new and radical on algorithm level, its very nice to be able to go and look your existing game scenes to see how much detail is lost without accurate shadows.



And here is one more example. Good accurate shadows are very good example of the amount detail that you can get into scenes without going up with display resolution. One could argue that it doesn’t make much sense to go towards big resolutions before we can sample lower ones with good quality.

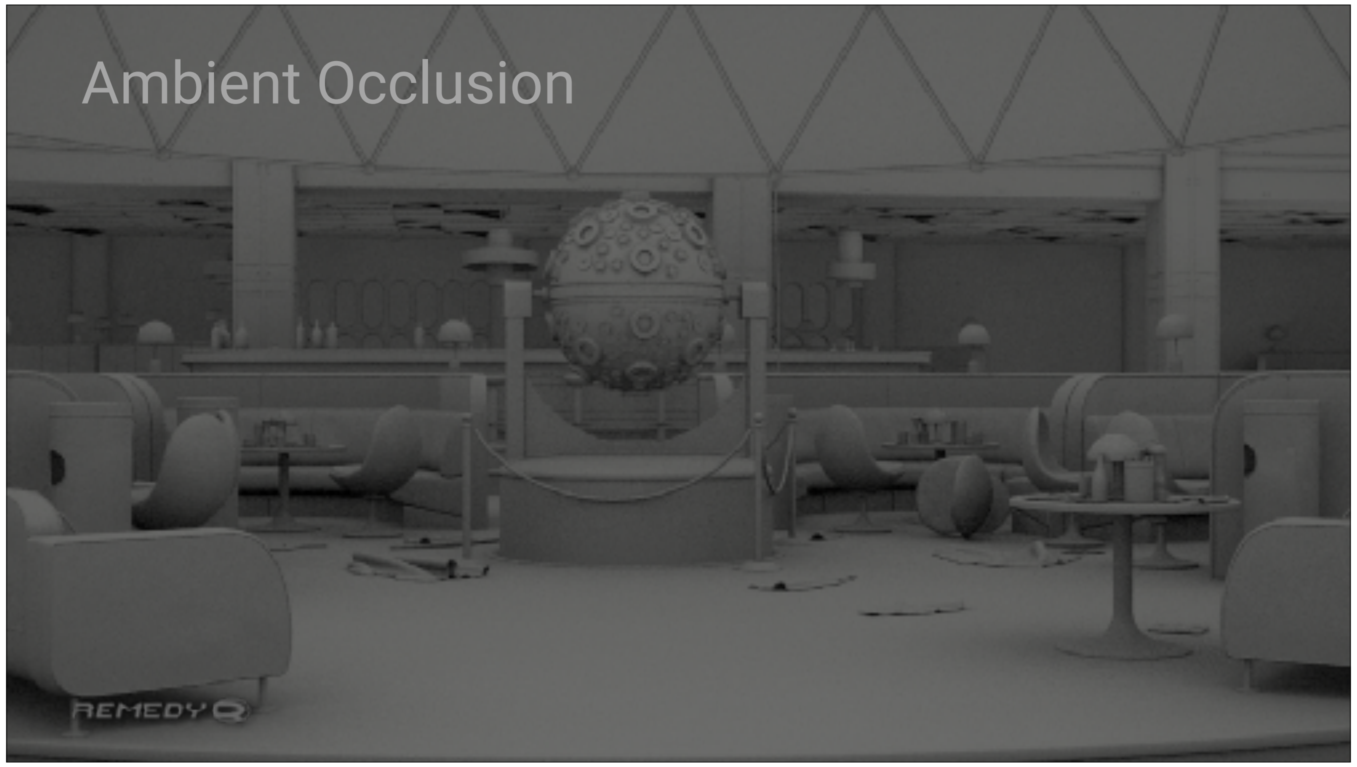


Direct Light Shadows

Performance

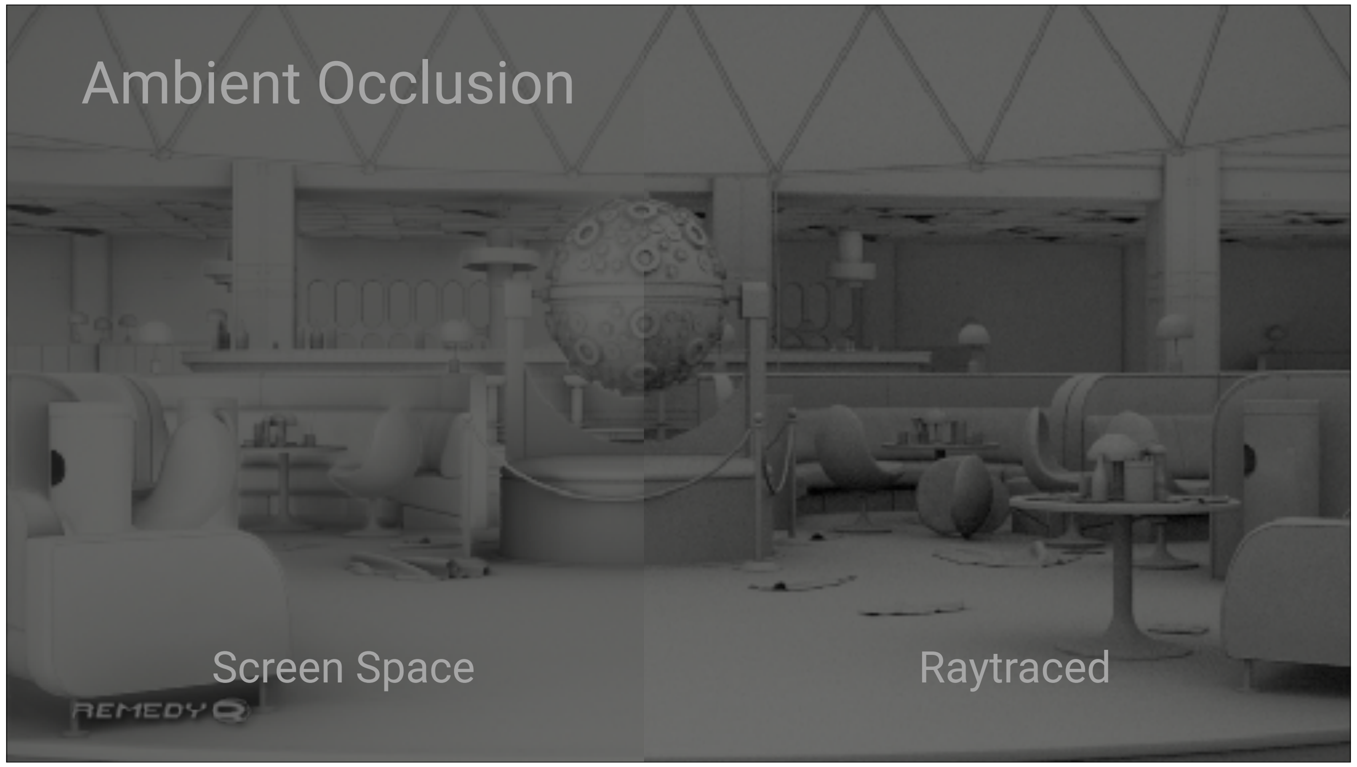
Performance isn’t quite there to ship a game with all shadows raytraced, but this is definitely interactive with modern hardware. We have roughly XX million triangles in this demo scene, and shooting a single shadow ray per pixel on fullHD setup costs around XX milliseconds. That gives you something like XX shadow rays per second.

Ambient Occlusion

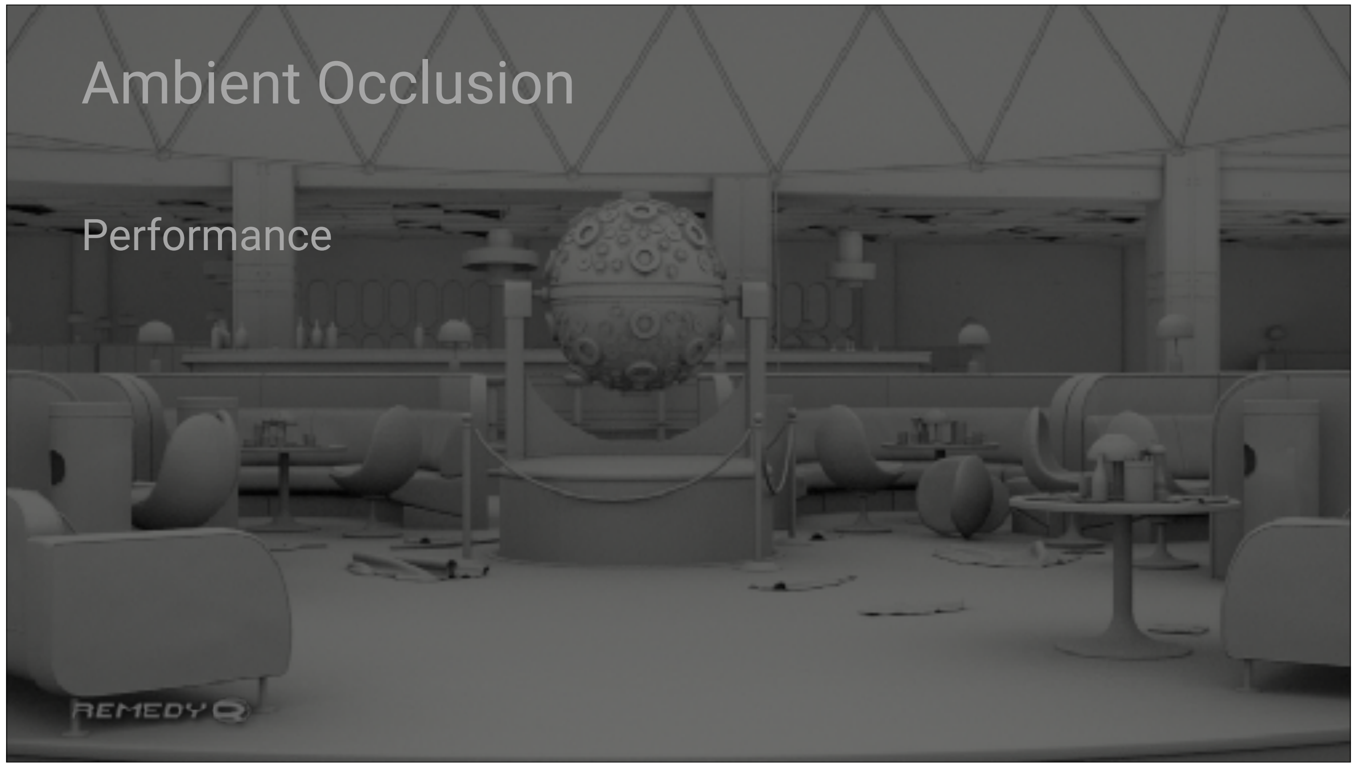


Ambient occlusion is other simple visibility based algorithm that can be easily tested out with raytracing. Image you see on this slide is taken by shooting 4 four samples per pixel using cosine distribution. This is also added as a direct replacement to what our screenspace ambient occlusion outputs, so it is easy to compare results of different algorithms.

Ambient Occlusion



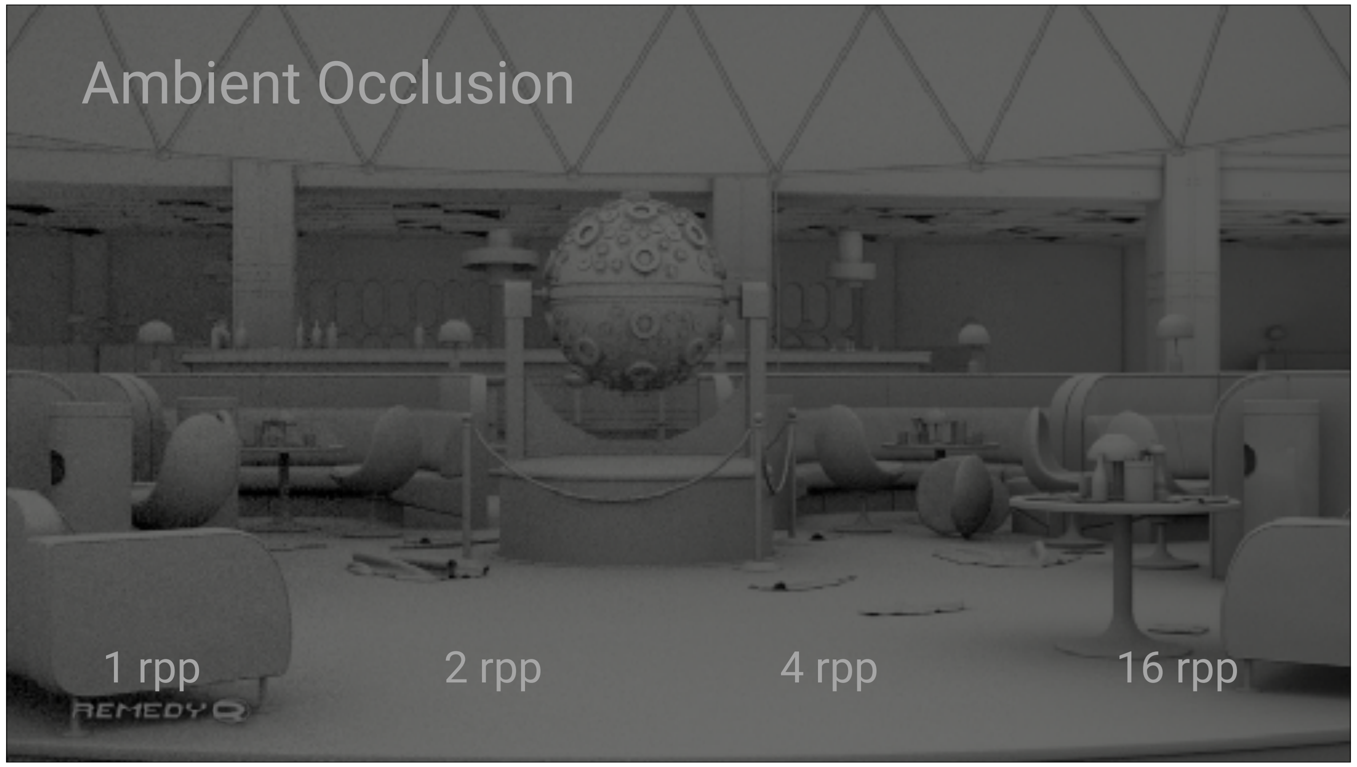
This is split screen, with screen space ambient occlusion on the left side, and raytraced ambient occlusion on the right. While our screen space technique does pretty good job on grounding things, it is pretty obvious that it is lacking information. Screen space algorithms can’t know what is outside the screen or behind the surfaces that are directly visible on camera.



Ambient Occlusion

Performance

Shooting rays is of course more expensive that traversing screen space depth. In our quick prototype implementation, shooting a single ray with maximum distance of four meters, costs roughly 5 milliseconds. The cost scales pretty much linearly, so shooting 16 rays for full HD picture, that I have in this image, takes roughly 80 milliseconds.



Finally, here you can see a picture with comparison of different ray counts. You should note that these are all taken using temporal anti-aliasing.

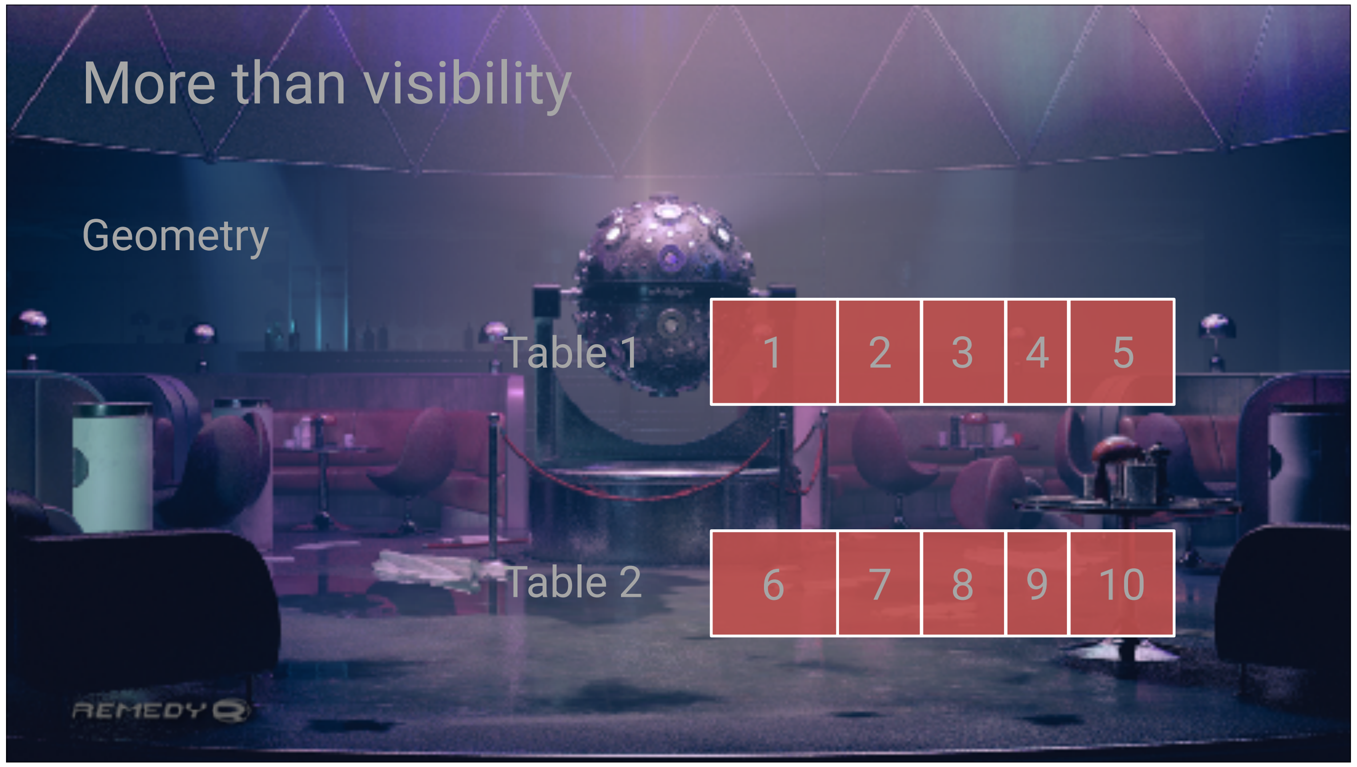
More than visibility

Shadows and ambient occlusion are examples of algorithms that you can implement with a simple and very limited set of data. Doing more interesting things, requires additional data. And also, in order to get something like alpha testing or transparency working with shadows or AO, you will need more data.

More than visibility

Geometry

When raytracing hits geometry, it will pass position of the intersection in object, scene and triangle base to intersection shader. Any other geometry data like UVs or normals, you will need to read manually from somewhere. There is few options to choose from. New DirectX comes with a new binding API, where you can set different shader root signature for each instance of a bottom level piece in the scene.

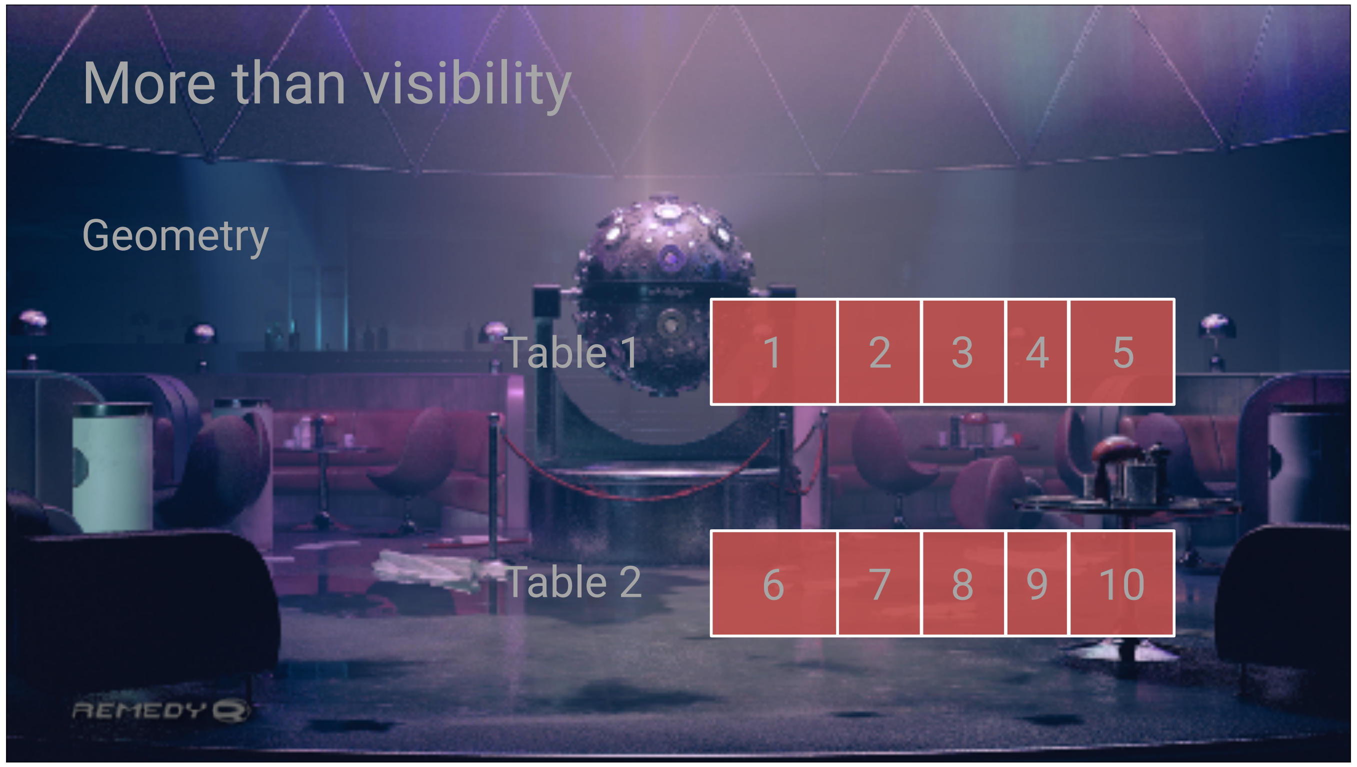


More than visibility

Geometry

Table 1	1	2	3	4	5
Table 2	6	7	8	9	10

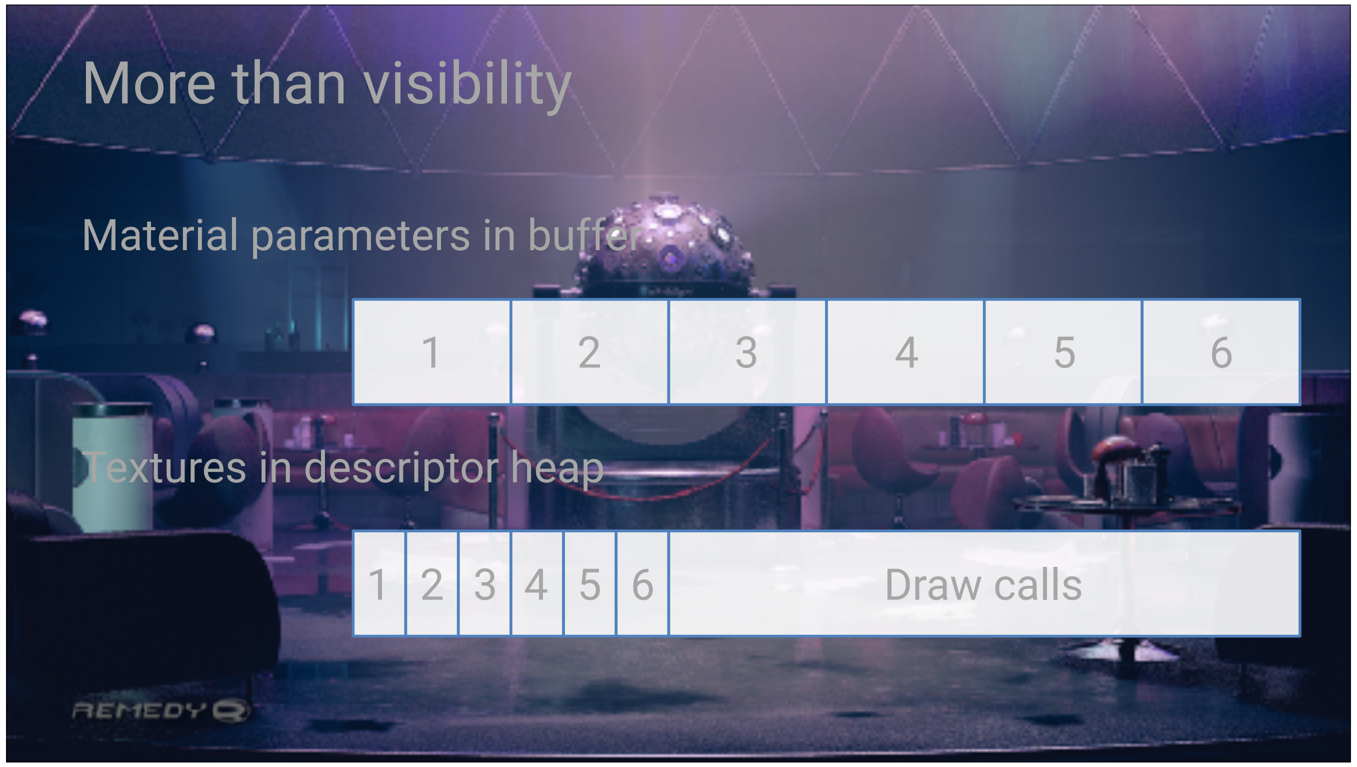
So let’s say, I have two instances of a table. Each table has five pieces for different materials, like in the diagram on this slide. I can now define separate root signature for all the pieces, so in total I have ten slots for root signatures. Each of the root signature can then have a custom bindings for vertex and index buffer for example.


More than visibility

Geometry

Table 1	1	2	3	4	5
Table 2	6	7	8	9	10

Other option we tried first, was to reserve separate space from a descriptor heap for all the buffers in the scene, and then index those manually from intersection shader code. In the end, doing bindings using multiple root signatures turned out to be quite a lot faster, so we ended up going with that.



More than visibility

Material parameters in buffer

1	2	3	4	5	6

Textures in descriptor heap

1	2	3	4	5	6	Draw calls

For material binding, the timing with raytracing experiments happened to fit perfectly for us. When hearing first time about the API, we had just completed the first version of our new material binding style that didn’t need any per draw call bindings for material data. Our approach is to assign unique index for each material and use that to lookup all the data that is associated with the material. We pack all the parameters into a single large buffer, where each material has unified size block for storage. Textures are stored in a separate space in descriptor heap, and indexed with the same material ID as other data.

More than visibility

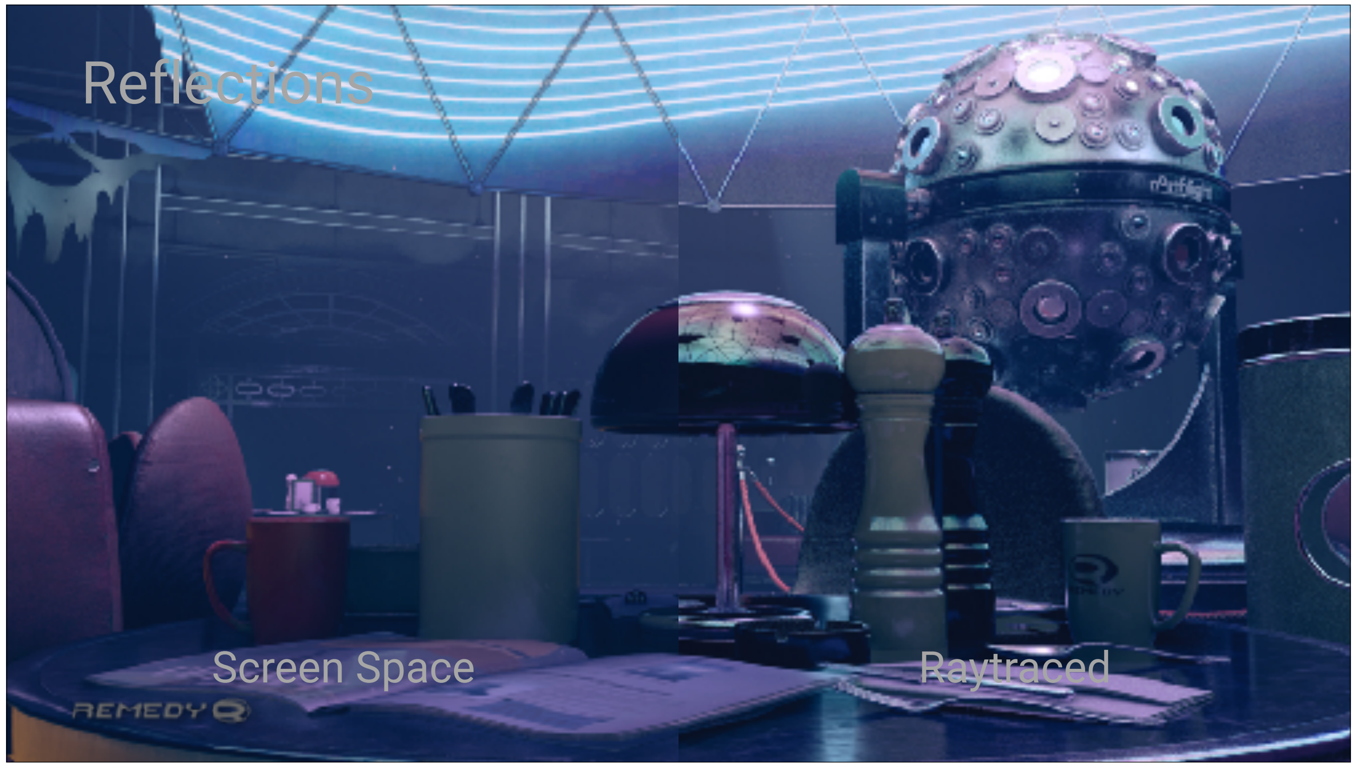
Lighting

Since Quantum Break, we already have all the lighting data bound so that everything can be used in a single shader call. For us, this means that our shadows and projection maps are stored in couple atlas textures, and parameters for each light can be found in few buffers.

Reflections

Ok, let’s get a bit more interesting use cases now that we have packed more data to be available on the raytracing kernel. I’ll start with obvious, so the next section is dedicated to implementing raytraced reflections. These are also implemented as a direct replacement for the screen space reflections that we have in engine.

Reflections



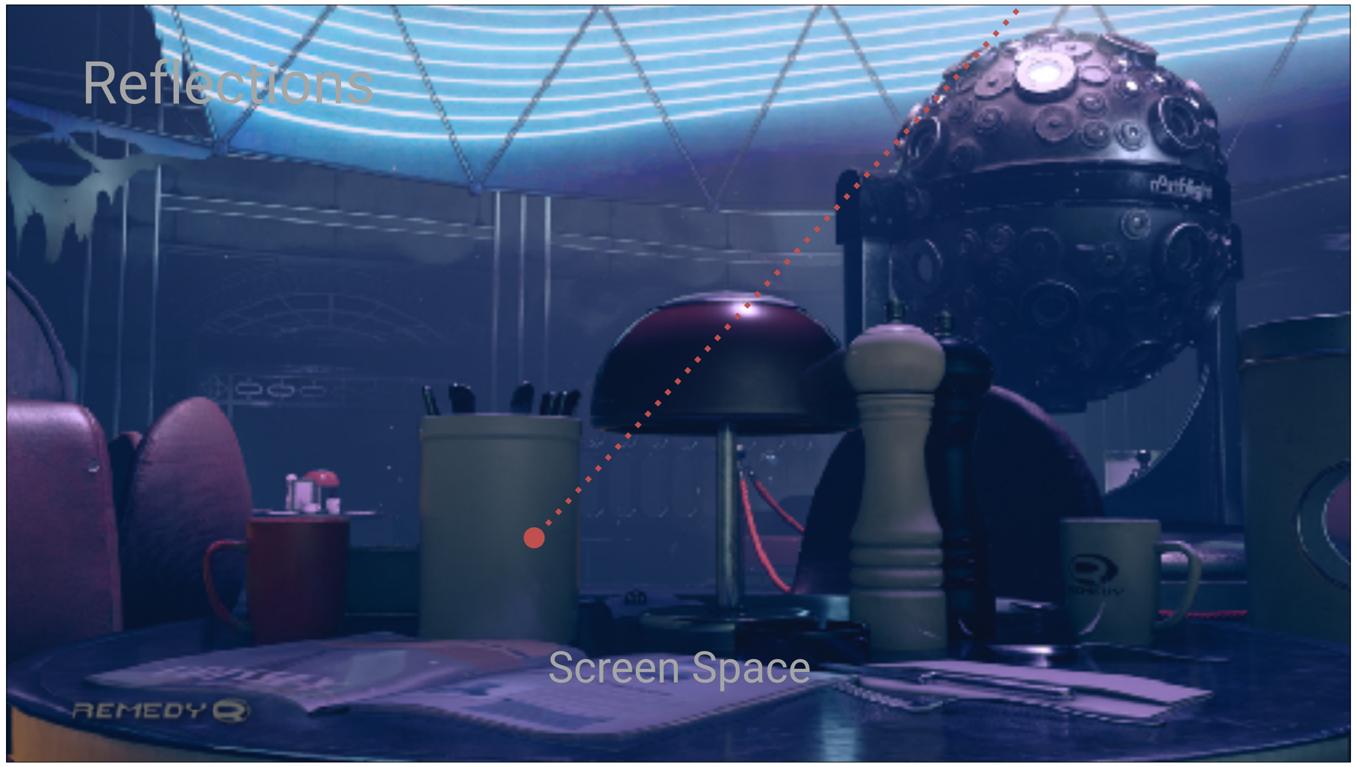
for demo we simply replaced screen space reflections with raytraced ones

Reflections

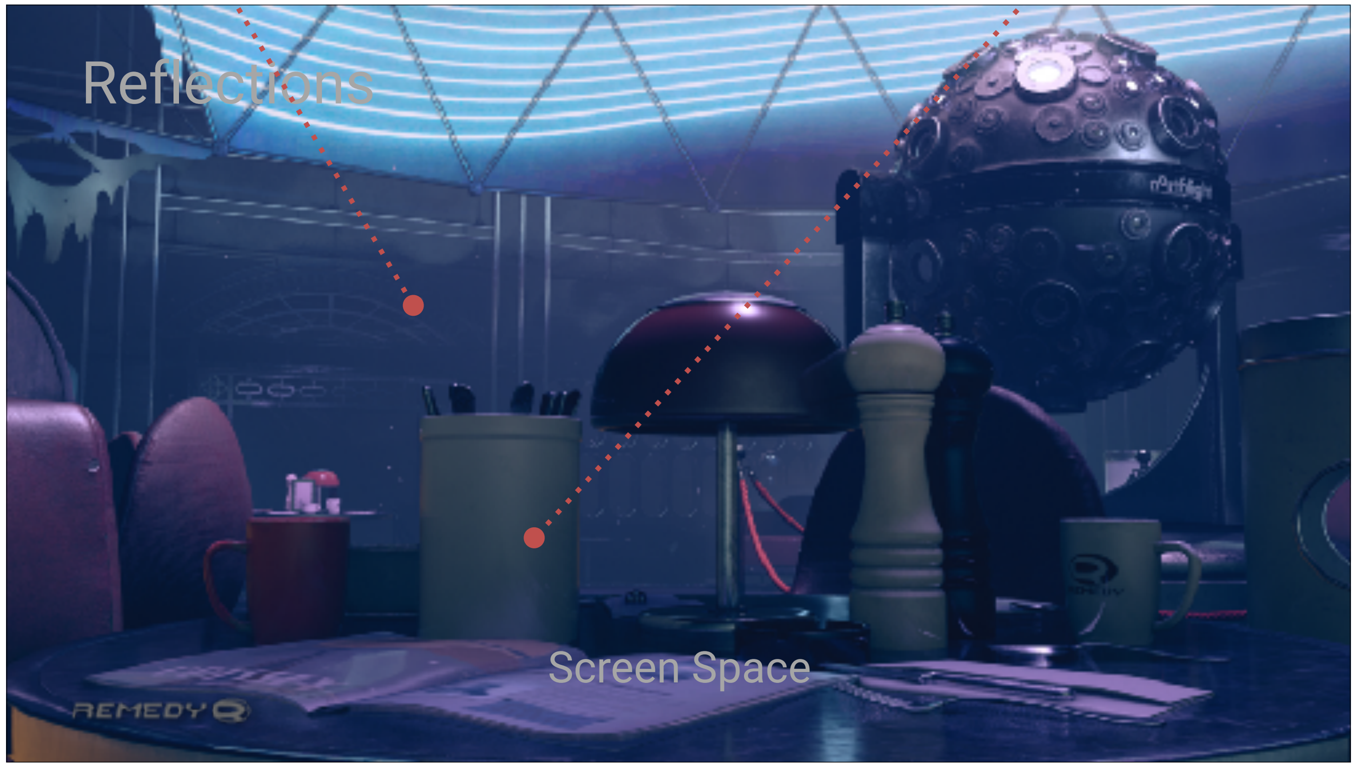
Screen Space
REMEDY

Reflection ray generation works the same way in both the screen space and raytrace algorithms.

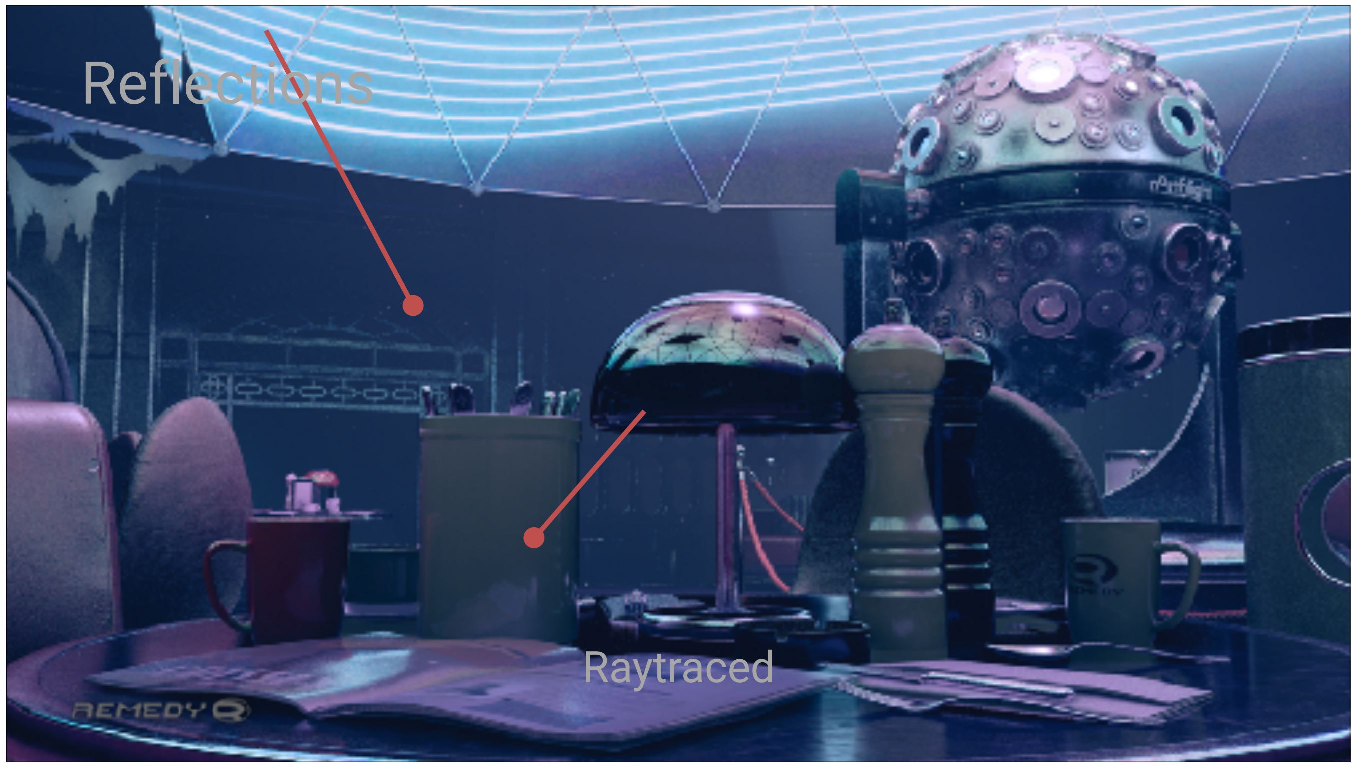
For each sample on screen, you read properties of the surface from the geometry buffer. Then based on smoothness, you pick a random direction from ggx distribution and transform it according to geometry normal.



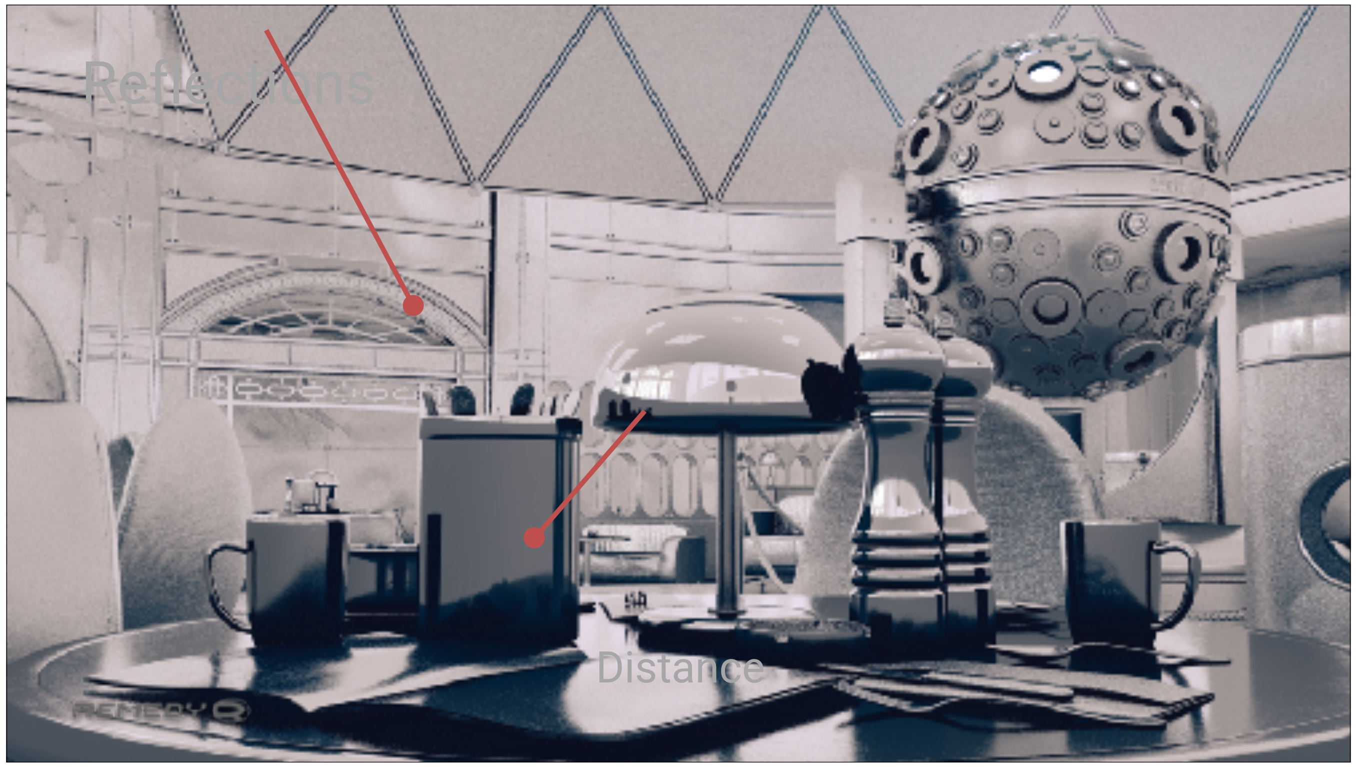
Our screen space algorithm treats reflection direction as a 2d line and takes some number of samples between screen edge and reflection start position. With holes in sampling, it is possible to miss occluders that are on the way.



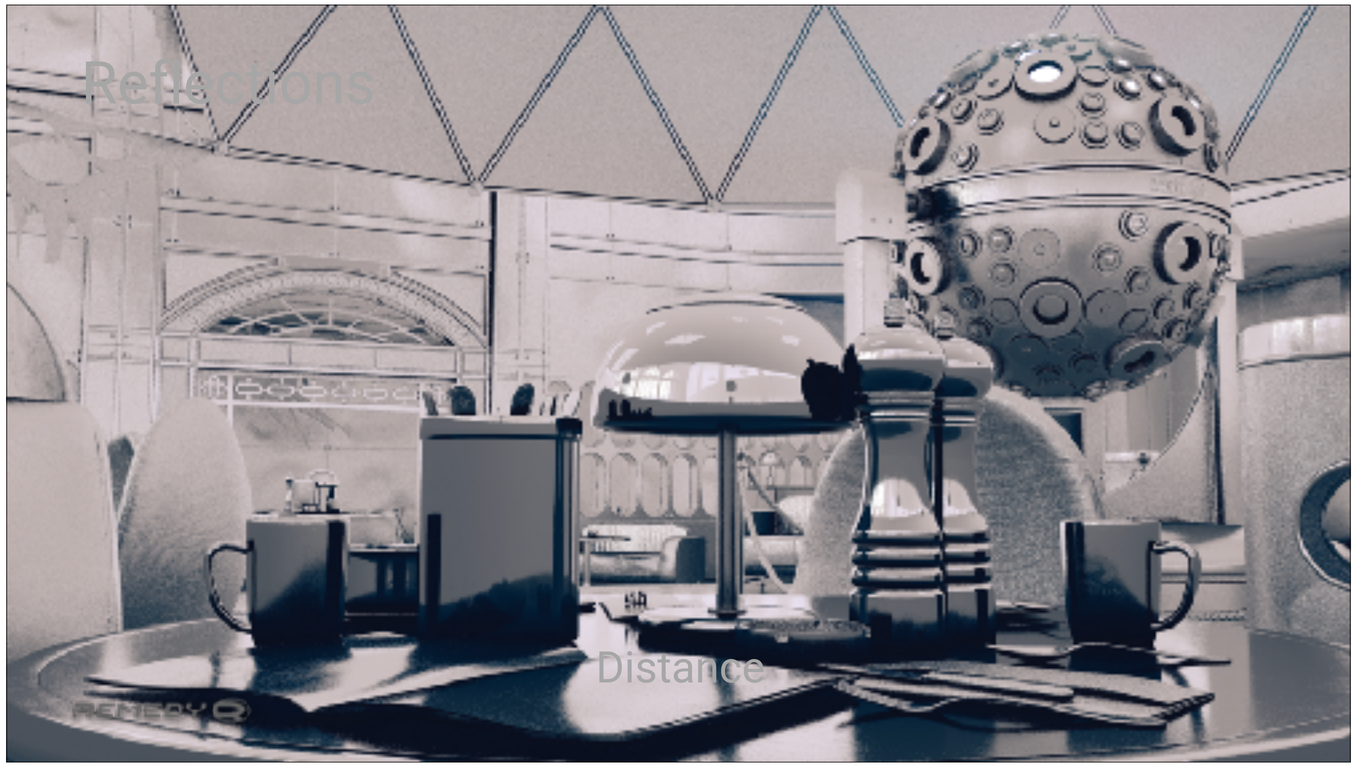
Also, it is possible that screen space doesn’t contain information on hit location, and you need to guess for instance thickness of this lamp, and if that would actually block the ray.



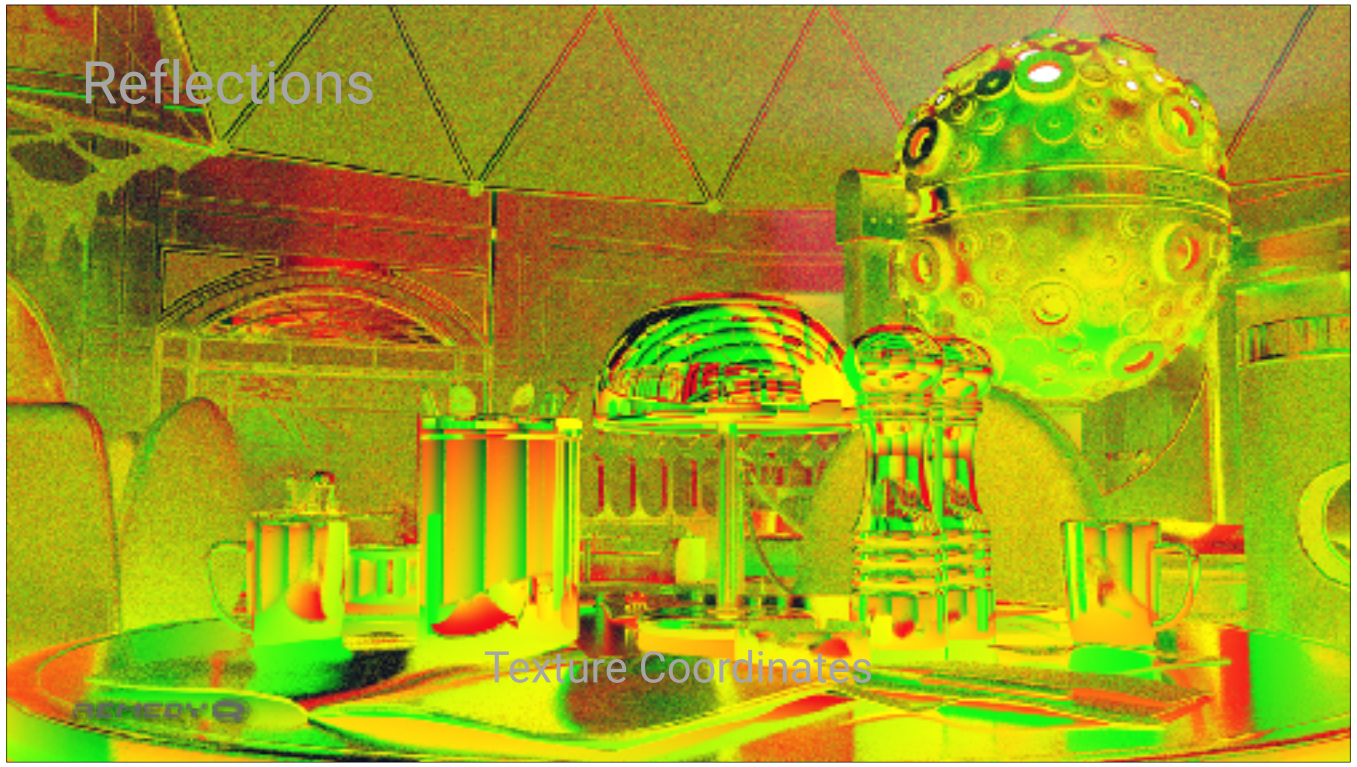
Geometry raytrace obviously doesn’t miss any geometry that is outside the screen, or behind directly visible surfaces. On a geometry hit, we need to first figure out properties of the surface, that was hit.



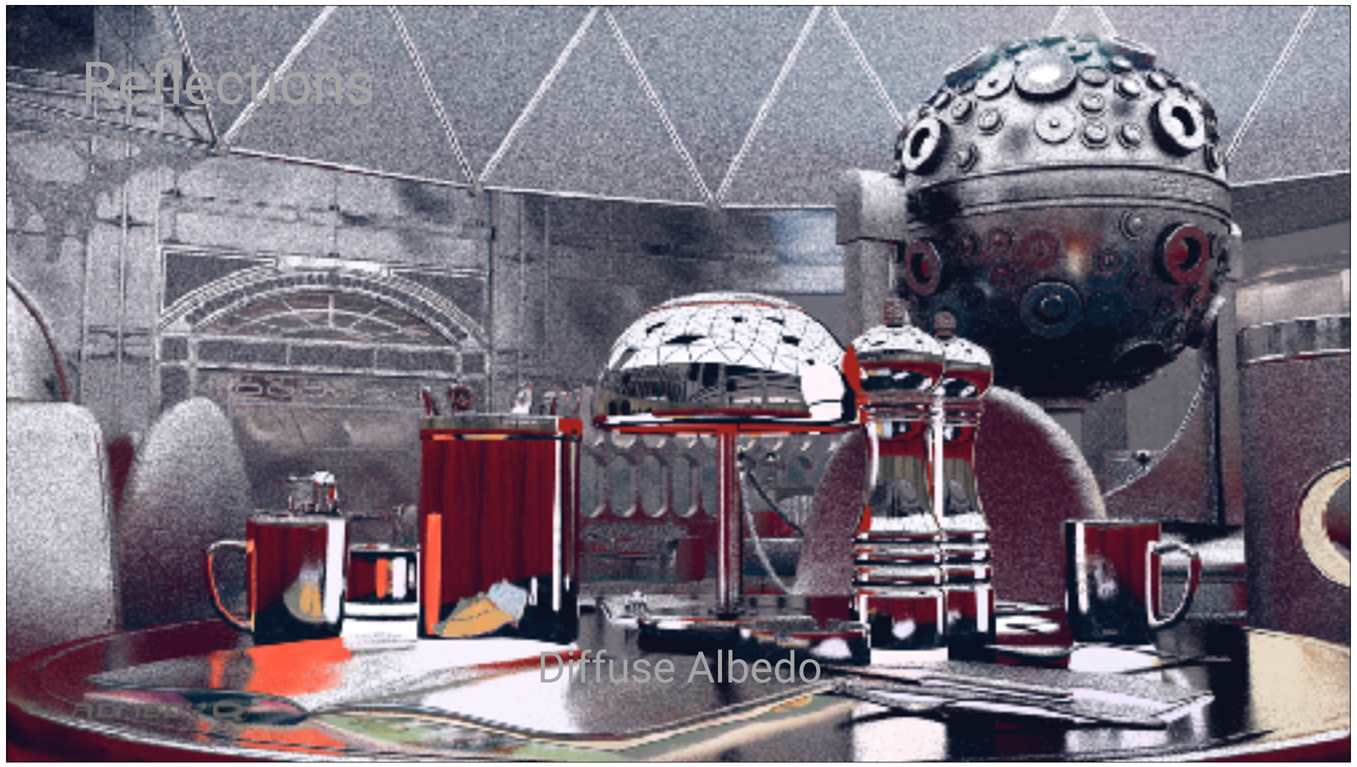
You also get distance to hit location, that I have visualised in this image. Dark values mean that hit position toward reflection was close to surface, while white indicates that the position was far away. So you can see that contact points to objects on the table are black, and rays that have gone out from window are white.



In order to sample surface properties on the surface that reflection ray hit, you need information we discussed in earlier section. From geometry data, you need to first figure our UV coordinate of the hit surface.

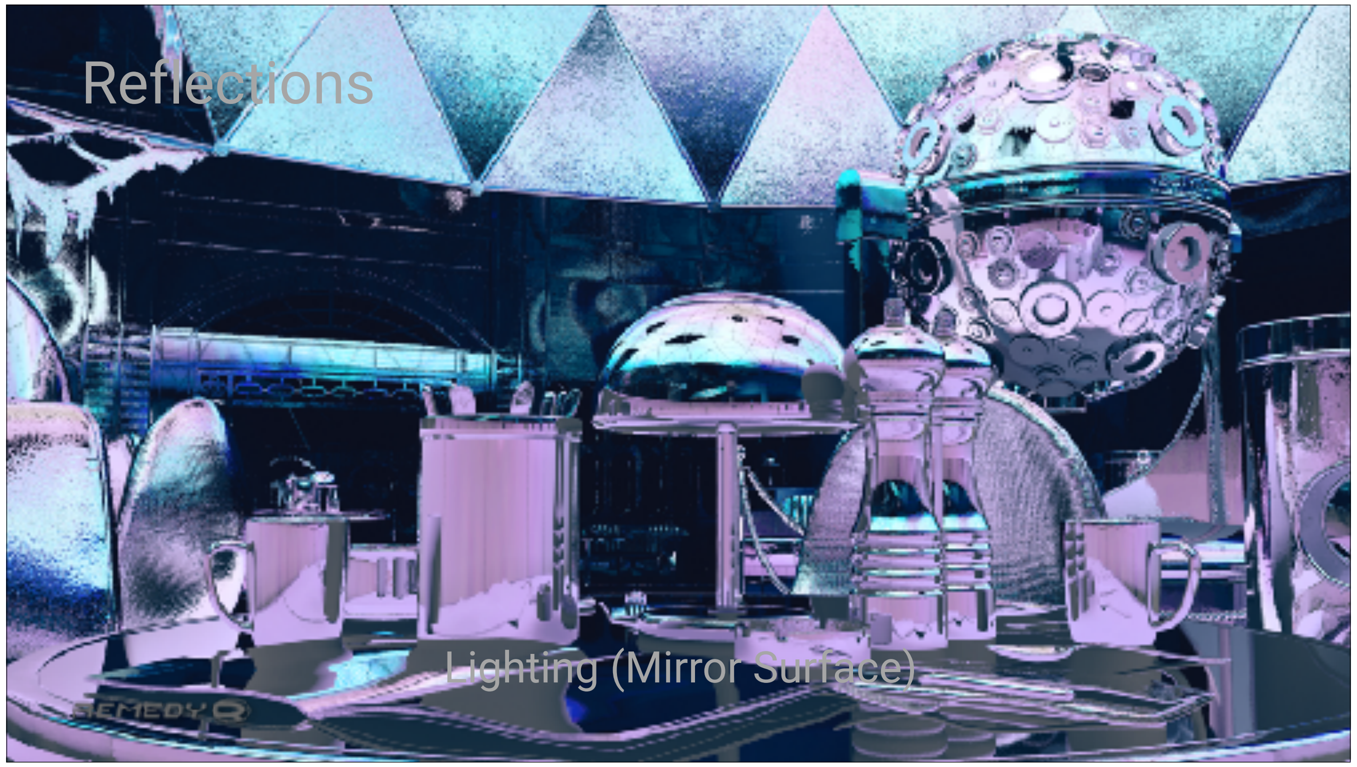


Here you can see UVs of the hit position. Quite a messy looking thing, I know.



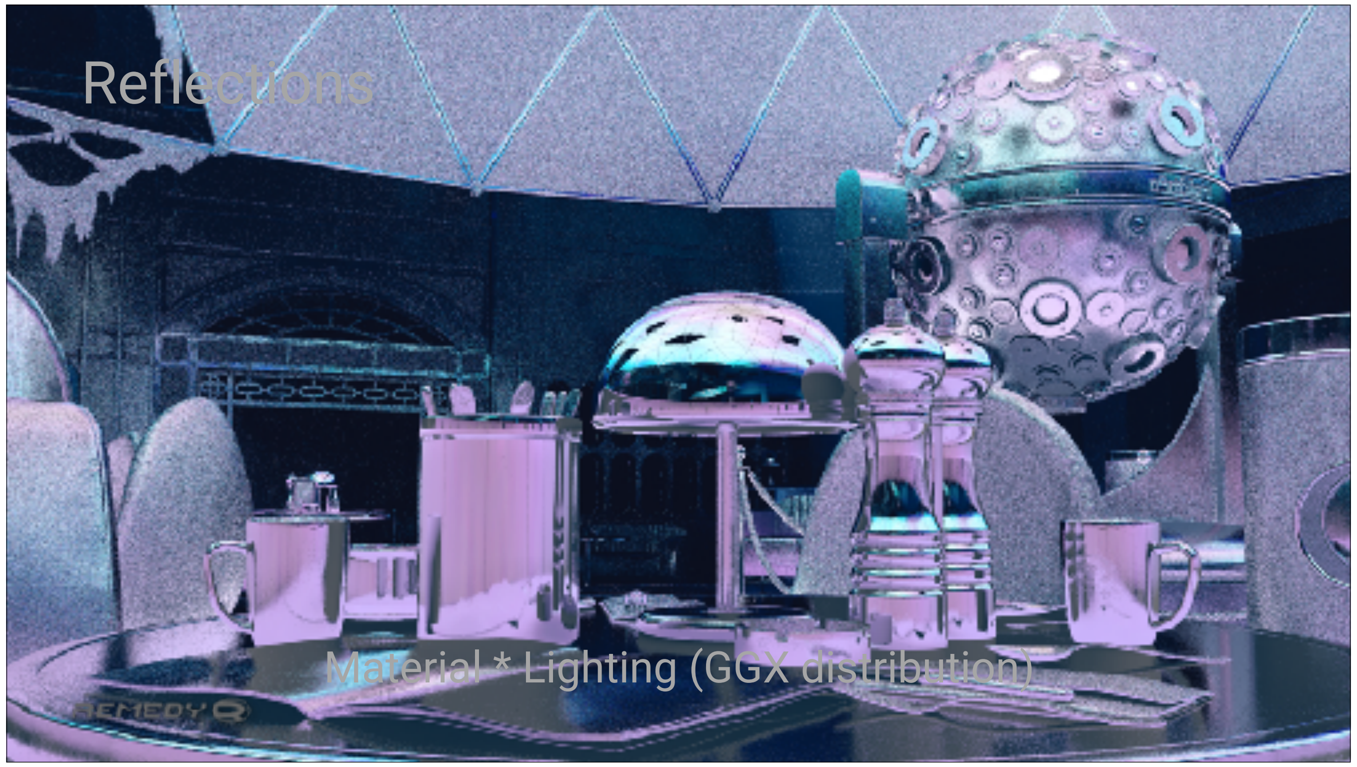
This is slightly less messy.

With UV coordinate and material info, you can sample diffuse albedo colour from texture. If your material is using alpha testing, you can discard sample based on alpha value here. After you have all the needed surface properties, it’s time to evaluate lighting and combine results.



Here, I have lighting only at the hit location.

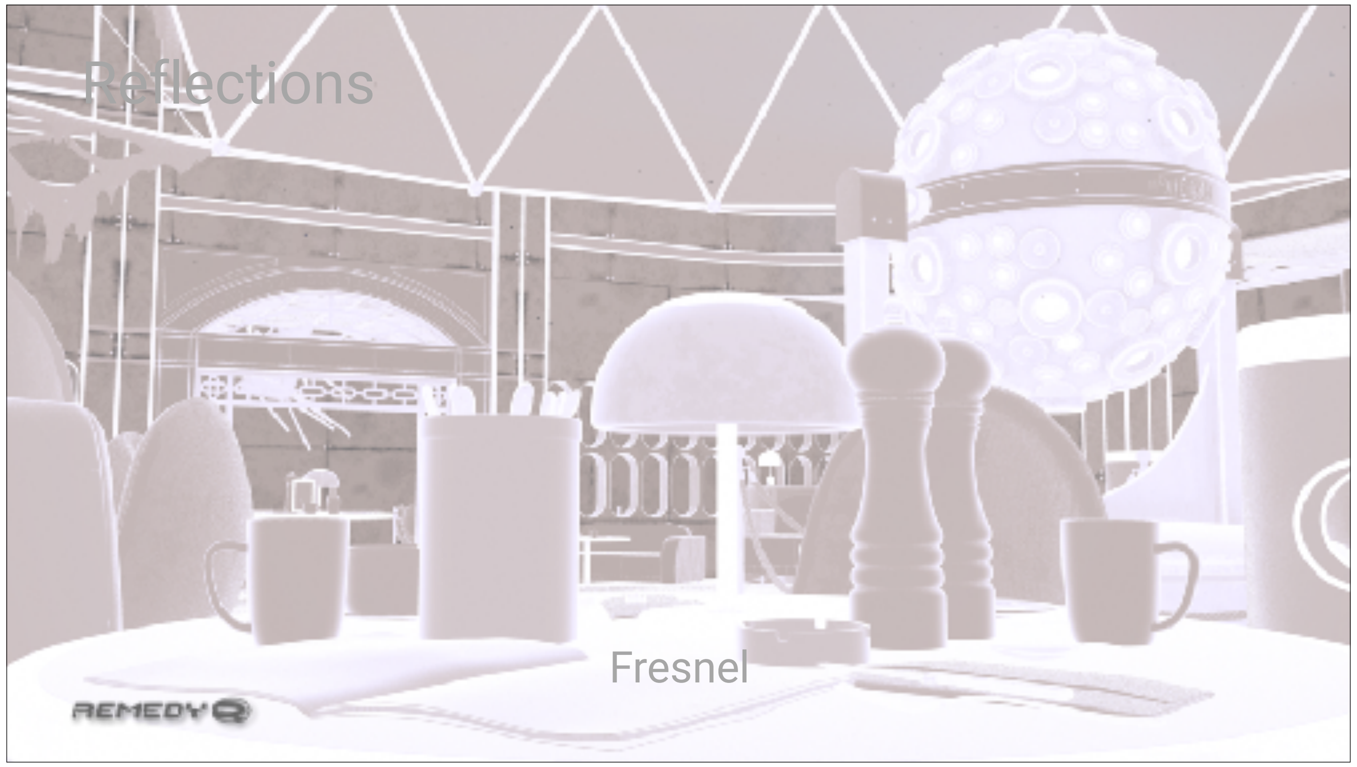
Reflections



THIS HAS WRONG PICTURE!!

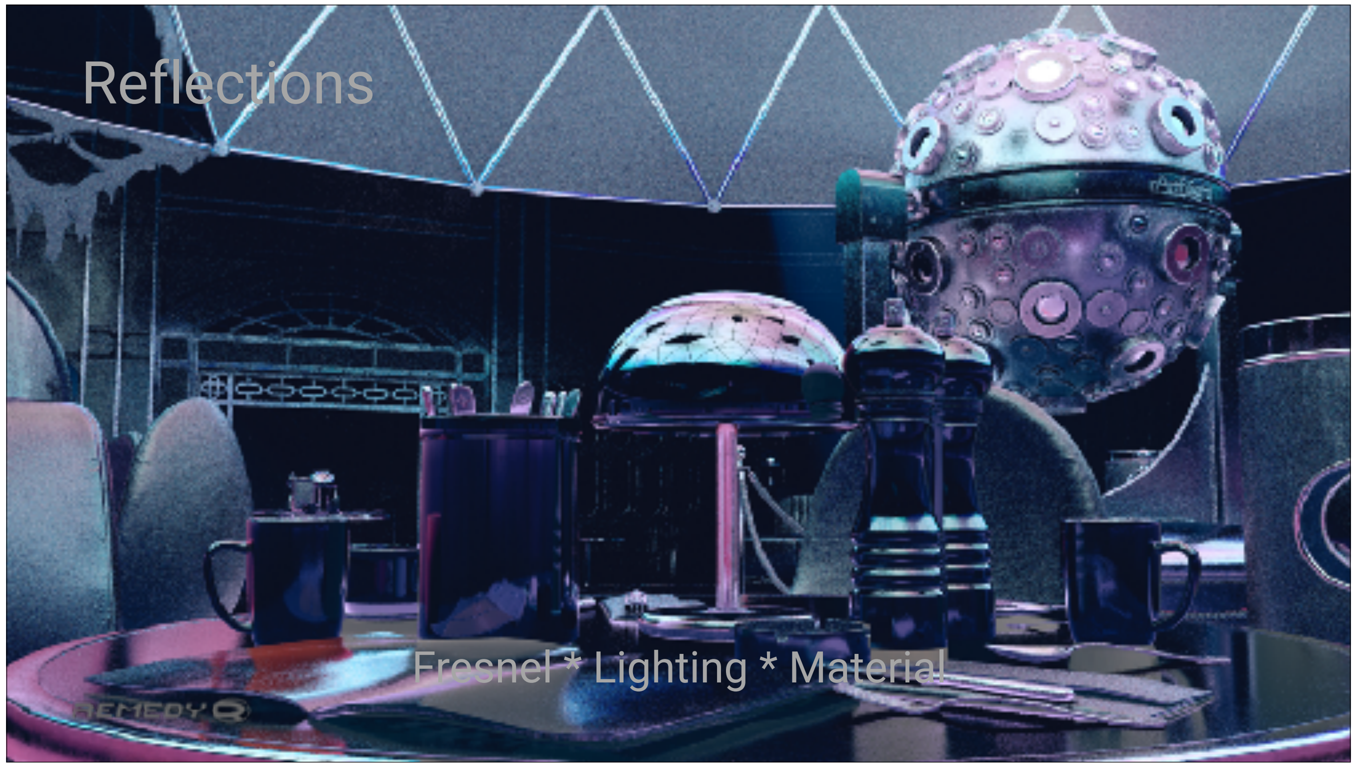
An this is combined result of lighting and surface properties. In order to make this behave nicely, we need to add fresnel term based on surface properties of source pixel.

Reflections



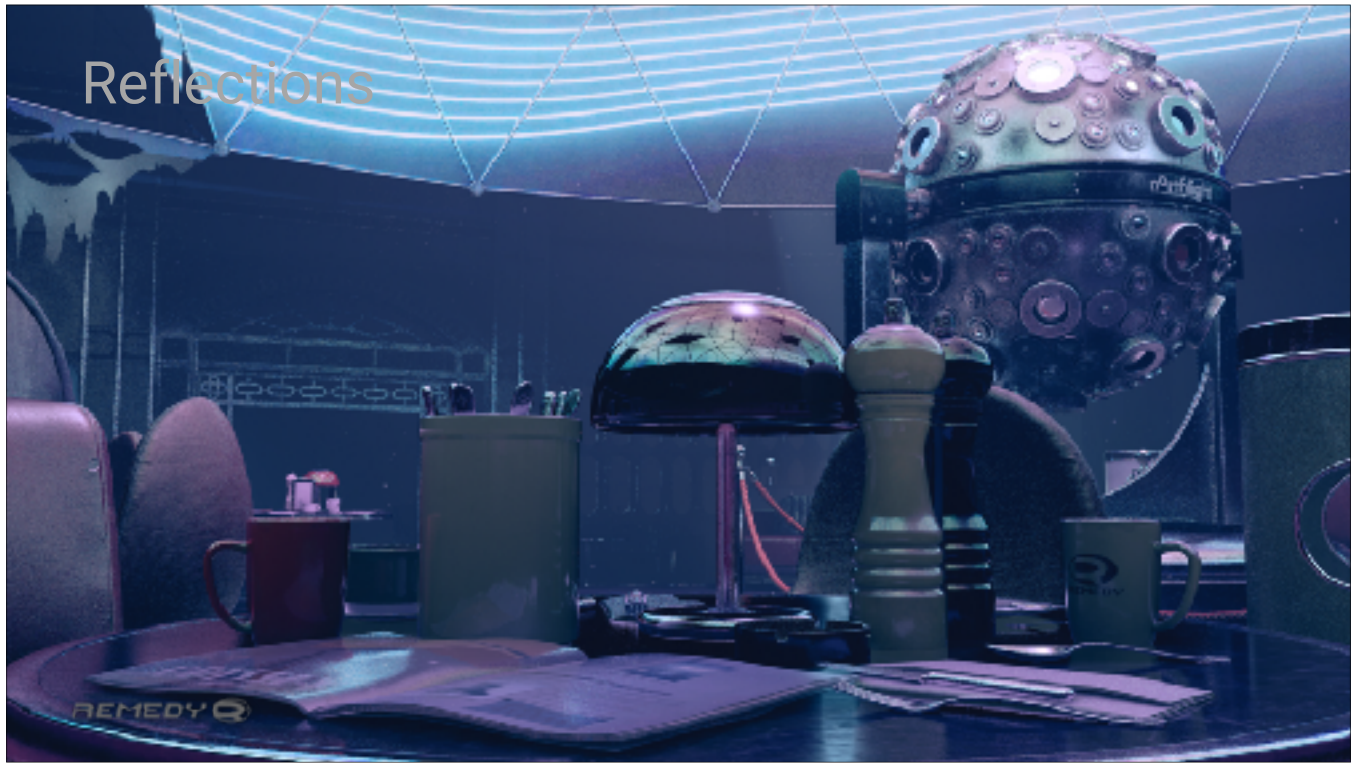
This is the basic schlick fresnel term we are using in the demo

Reflections

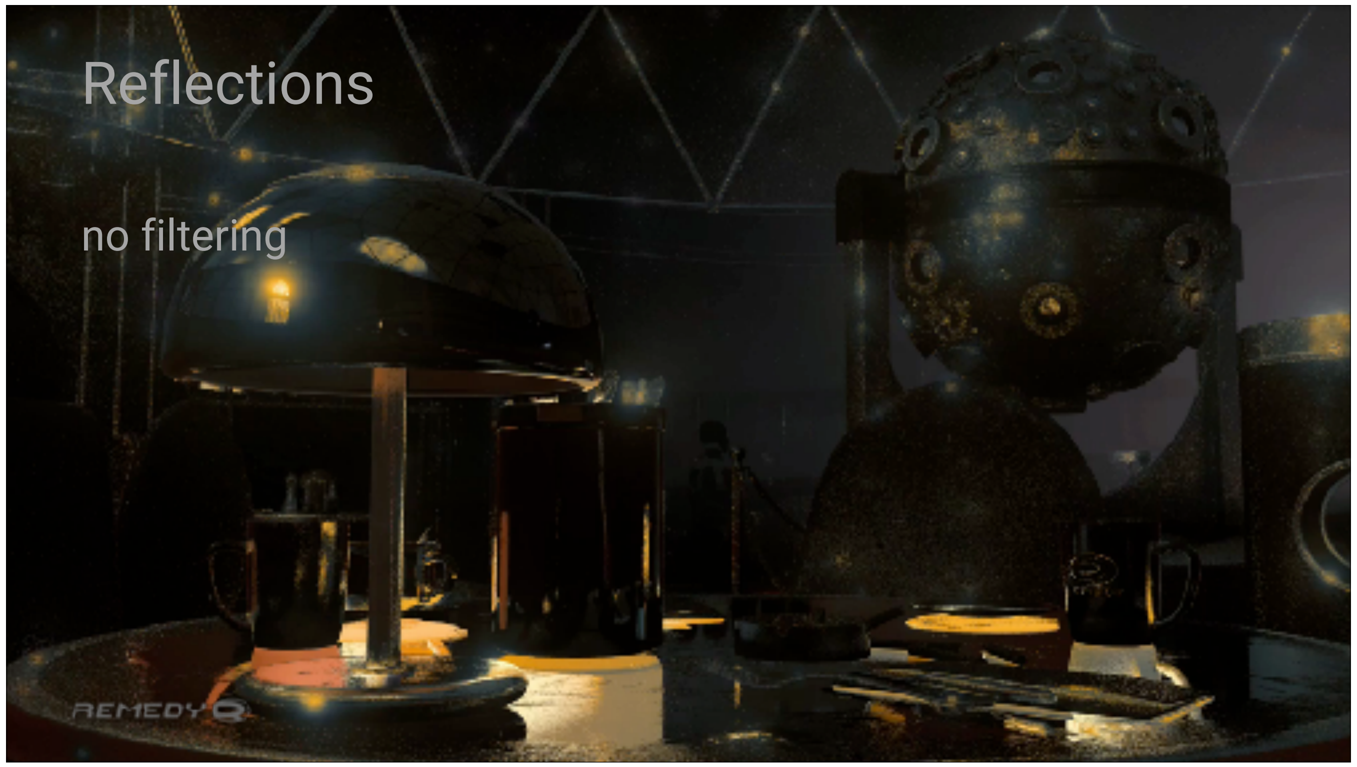

Fresnel * Lighting * Material

And here we have final result of reflections in the demo. This can now be combined with indirect diffuse lighting and full direct lighting.

Reflections

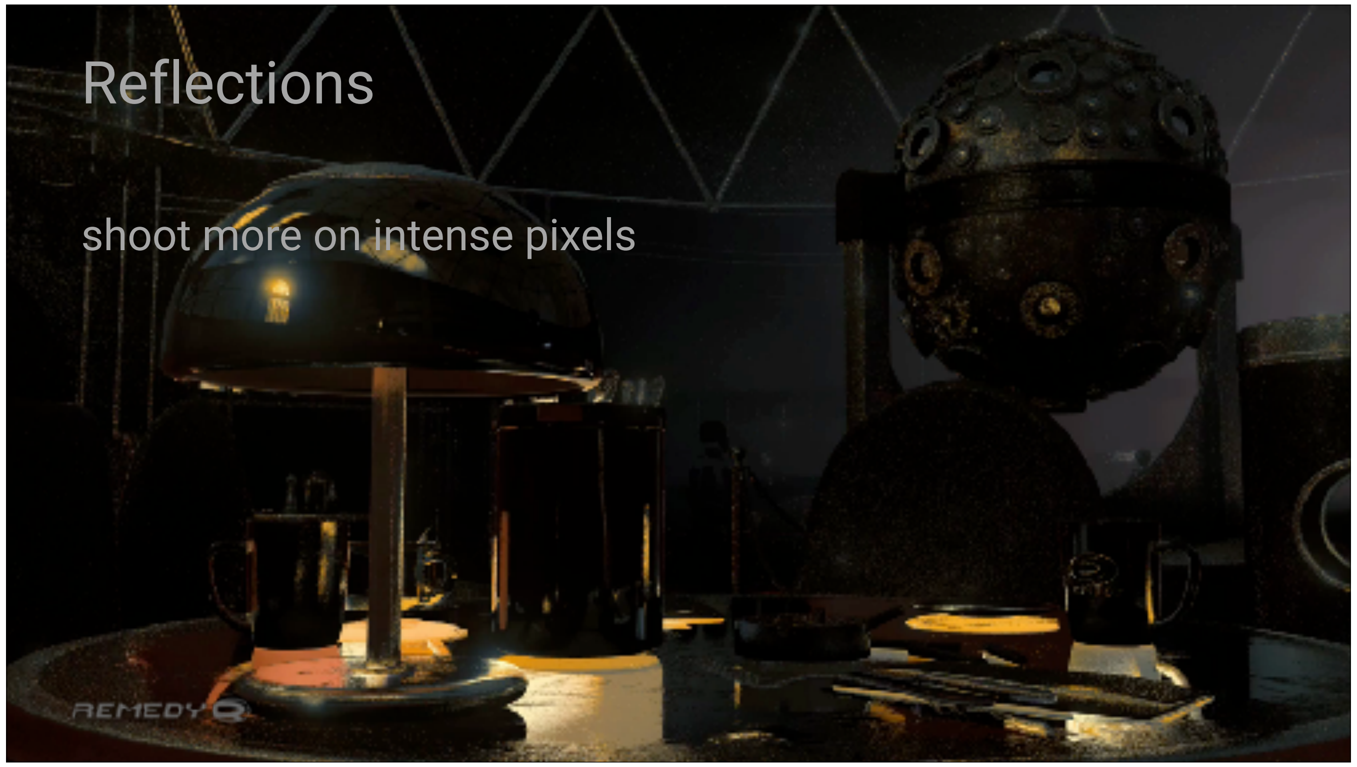


We didn’t really have much time to explore with filtering methods for reducing noise in reflections. The image you are seeing here has a lot of samples. I have a short loop to demonstrate how the results are when shooting single reflection ray per pixel without any noise reduction. The following is still using temporal anti-aliasing.



I’ve taken this with a different lighting setup, in order to capture a bit harder case. Unlike the previous image, lighting is from the start of the demo, where we have intense sunlight hitting in from windows.

Noise is clearly sequential as we use loop the same samples over four frames as that fits well with our temporal anti-aliasing. This loop has bloom enabled to make bright spots more clearly visible.



Reflections

shoot more on intense pixels

Our simple approach, that is nearly free, is to add few samples to pixels that are two times brighter than the white point. This is very cheap in terms of implementation complexity and quality of course. Better approach would be to use some more advanced algorithm like gradient domain sampling.

Reflections

and combine with small damping

In addition to shooting more rays, we are also damping pixels that we think might cause flickering.

Reflections

Performance

rays/sec

In the demo, we are using single ray per pixel, resulting in rather noisy picture. As we are running in fullHD, this means we shoot roughly 2 million rays per frame, costing around ??? 10ms. So if we were running reflection rays only, that would mean roughly 200 million rays per second.

Reflections

Performance

rays vs surface+lighting

From the overall time, roughly xx% is spent on traversal, and xx% is taken by resolving surface properties and evaluating lighting.

Reflections

Performance

coherent vs non-coherent rays

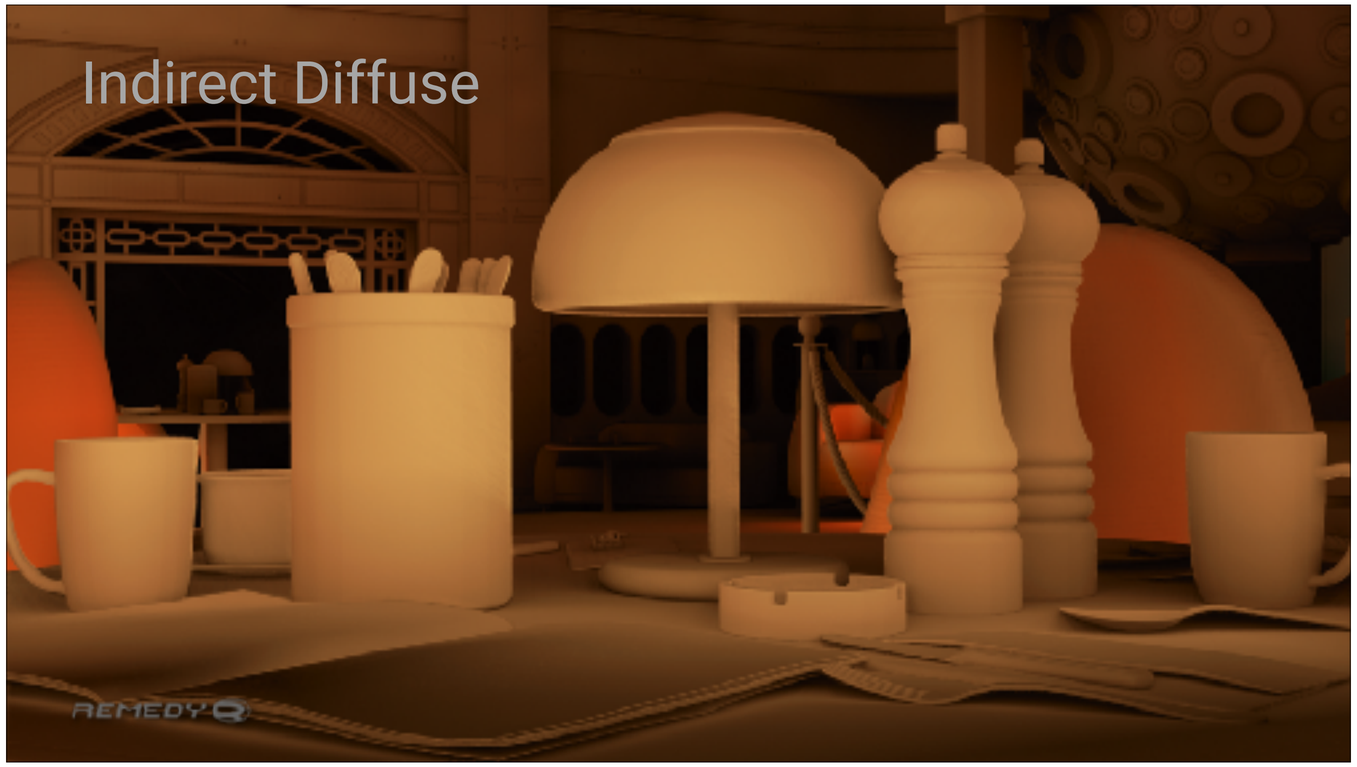
image: coherent vs non-coherent

Difference between coherent and non-coherent (mirror and glossy reflections) rays is around xx% in this demo level

Reflections

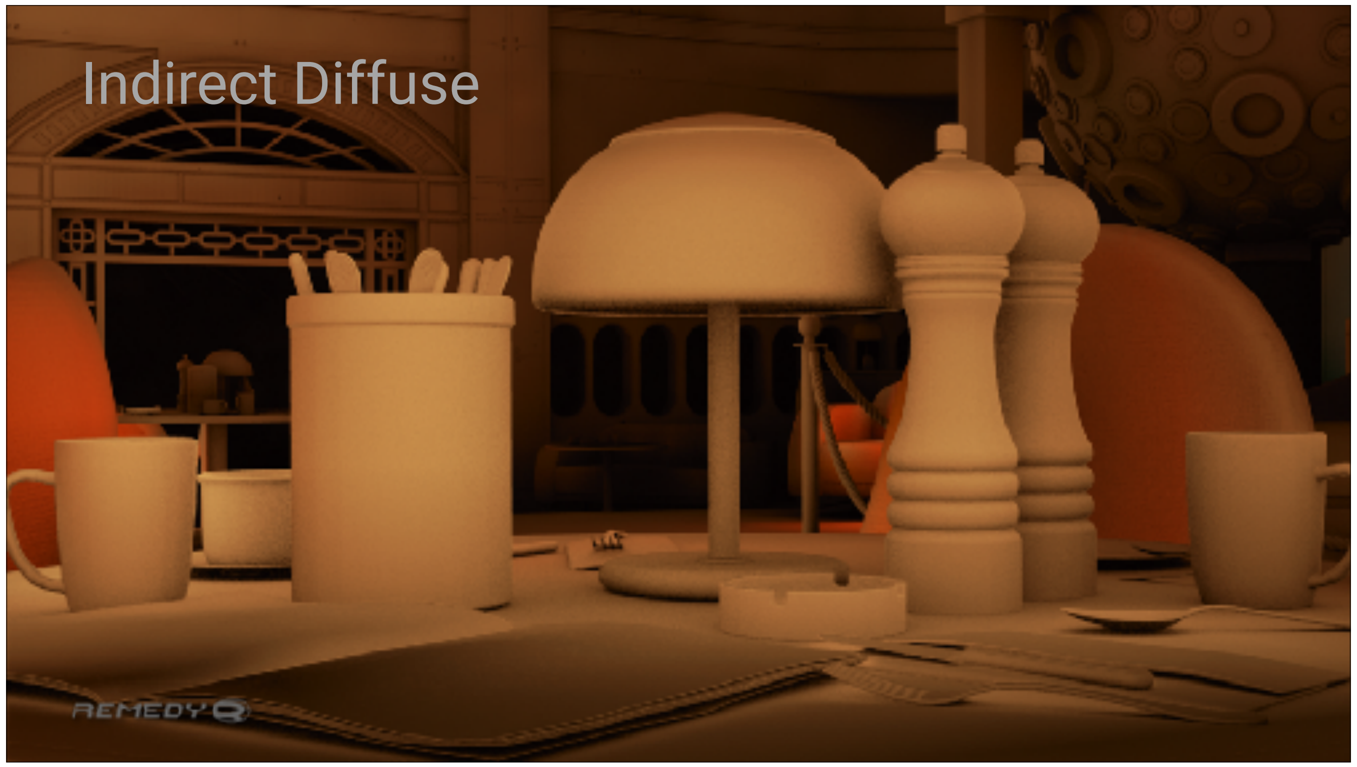
optimise by using shadow maps

In order to minimise the time taken to resolve lighting on reflection hit location, we are using shadow maps on point and spot lights. For directional light, we use raytracing, as our cascaded shadow maps don’t have valid information outside the view frustum. It would probably be possible to combine shadow maps on view frustum, and use raytracing only as a fallback, but we didn’t have time to test that.



Lets jump to diffuse component of global illumination. In this picture I have our volume based diffuse global illumination, as it is currently added by default in engine. Voxel resolution of the GI data is 25 cm, and we combine results with screen space ambient occlusion. As showed you earlier, we have a path to replace screen space AO, with raytraced one. Let’s have a look on how that looks.

-Ok, after showing raytraced reflections, its pretty obvious to jump on to showing diffuse global illumination.

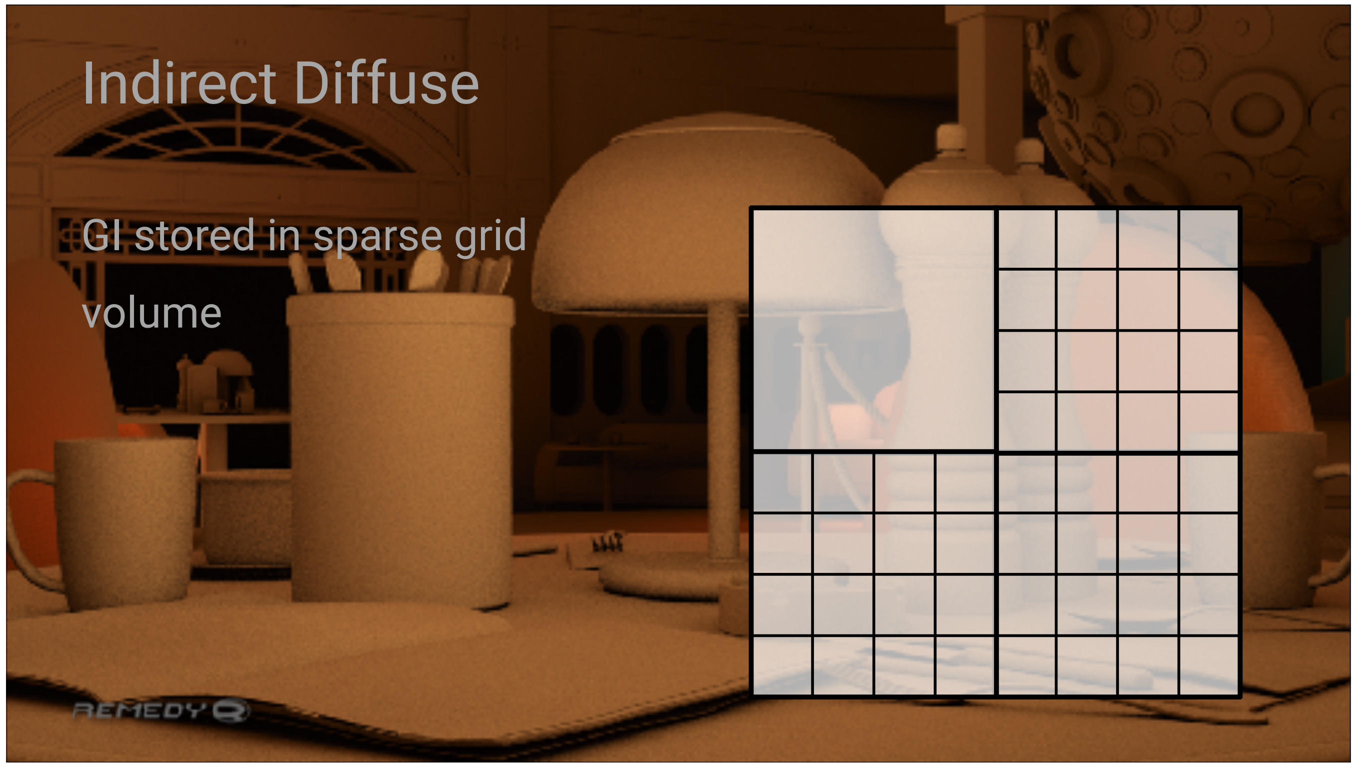


This is the same diffuse global illumination data, combined with raytraced ambient occlusion. Results are better, but since we have packed all the information about lighting and materials, we can do more interesting stuff.

Indirect Diffuse

Similar to AO, lots of non-coherent rays

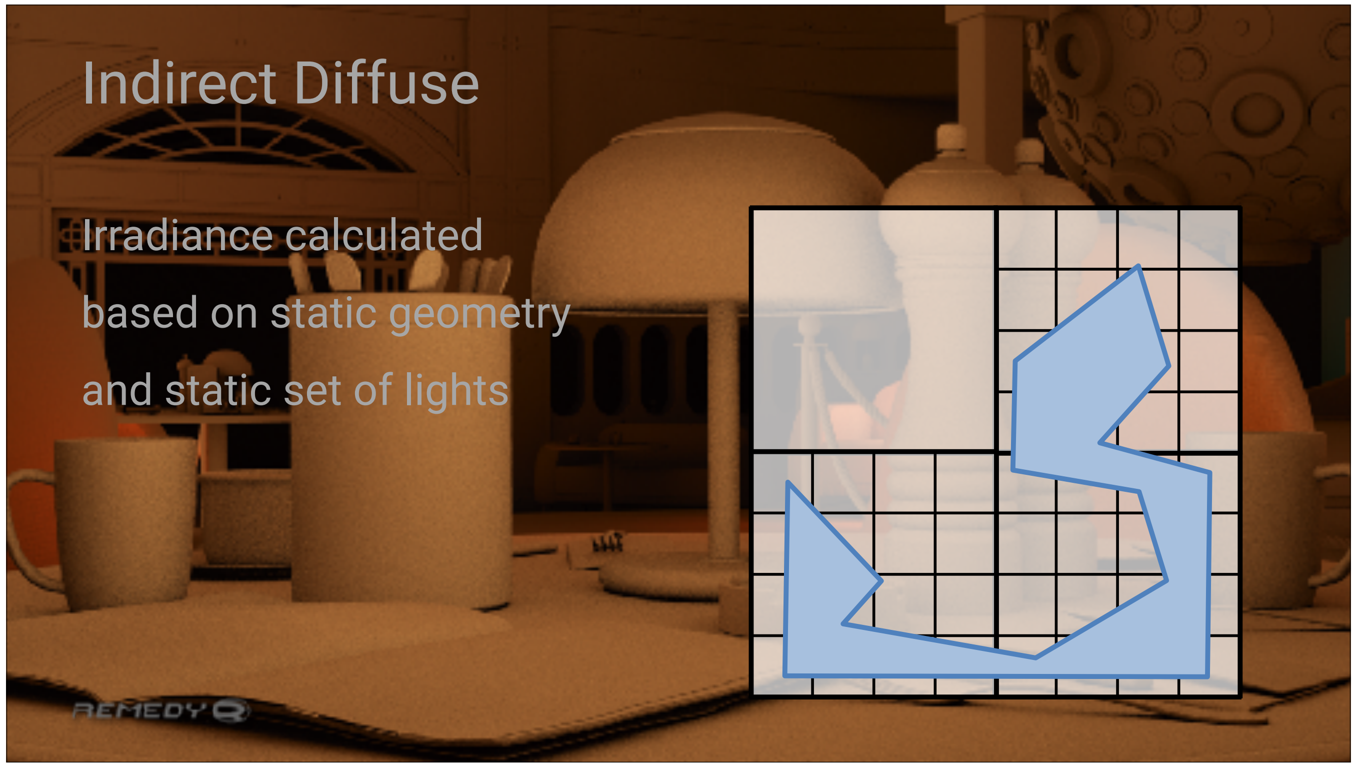
Instead of just shooting a lot of rays around to resolve indirect diffuse lighting, I'll be talking a bit about improving volumetric global illumination sampling and resolving near field global illumination by using raytracing. I'll quickly break this down with few diagrams.



Indirect Diffuse

GI stored in sparse grid volume

Our pre-computed global illumination is stored in sparse volume texture. The grid on the screen represents the data. Each crossing in this simplified 2d figure contains indirect lighting data that has been calculated with pathtracer.



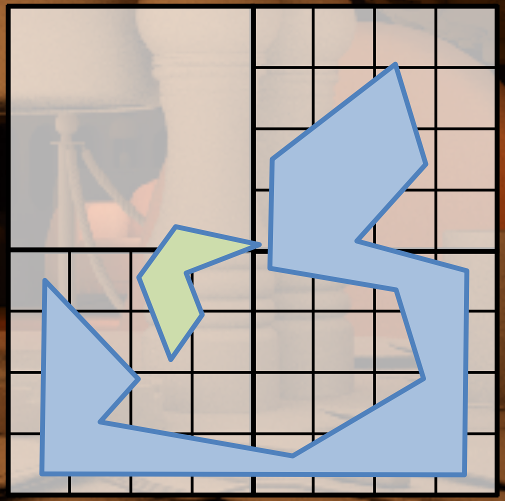
Indirect Diffuse

Irradiance calculated based on static geometry and static set of lights

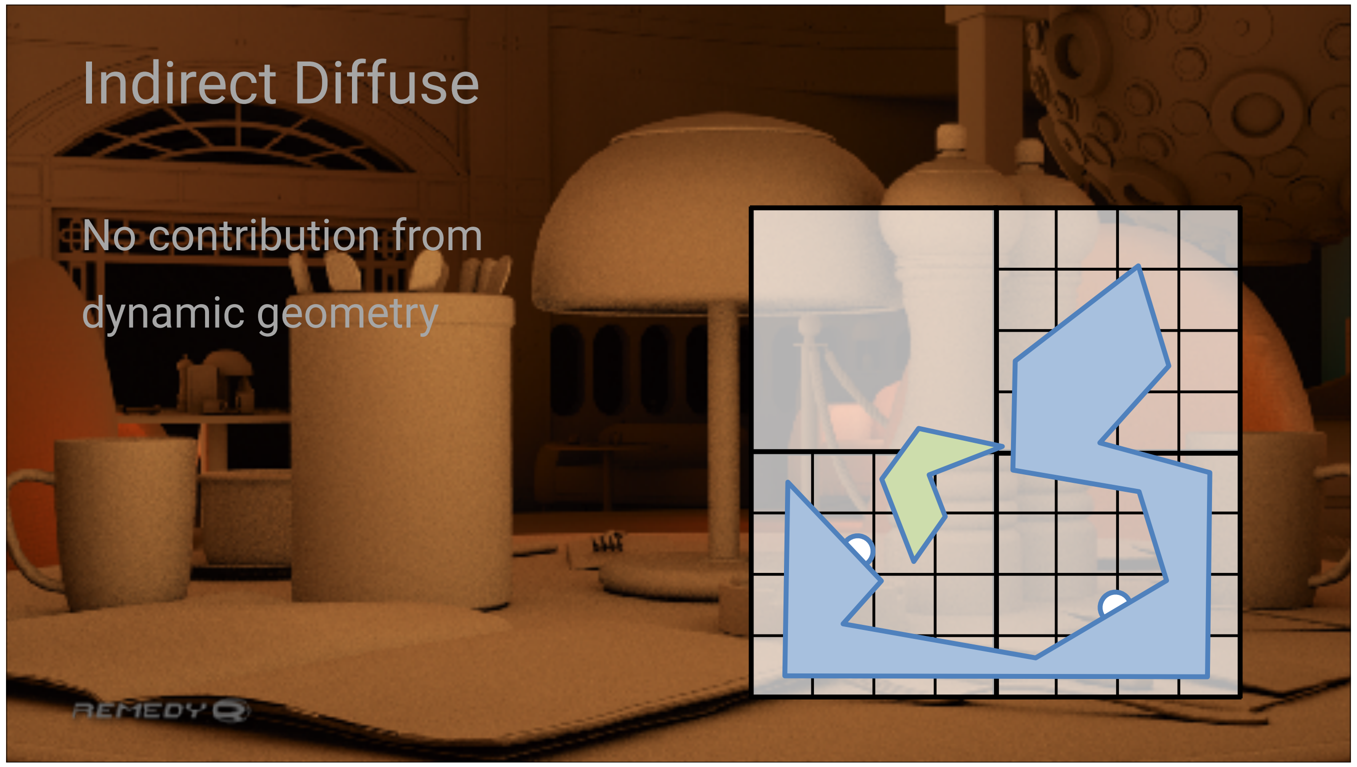
The blue shape on top of the global illlumination volume, is static geometry that we use for calculating the data. You can see that the dense parts of the GI volume are on the areas that intersect with geometry.

Indirect Diffuse

Dynamic geometry can receive light, but does not contribute to calculated irradiance



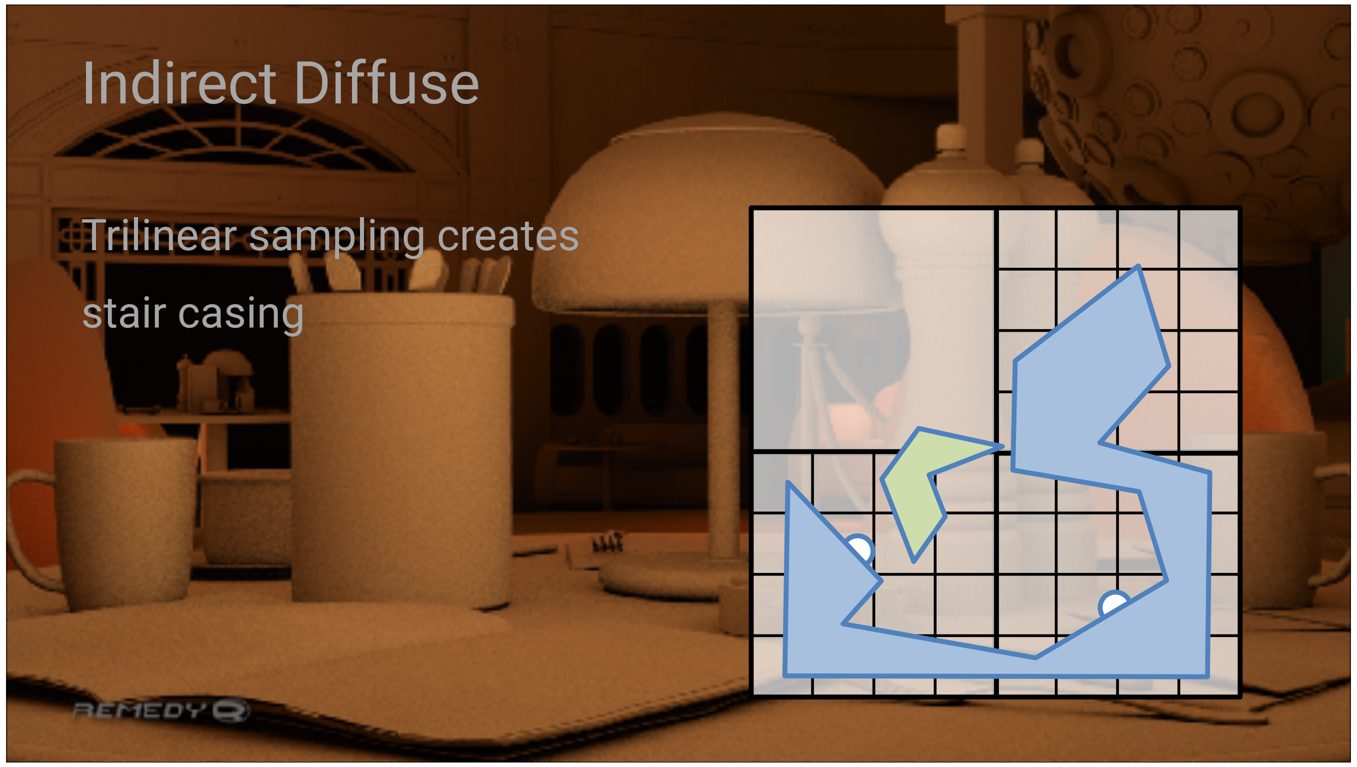
In addition to static geometry, our scenes of course contain a lot of dynamic geometry. The smaller green piece in the middle of picture is dynamic geometry. Dynamic geometry is not used in pre-calculation, so part of it resides on low resolution area of volume.



Indirect Diffuse

No contribution from dynamic geometry

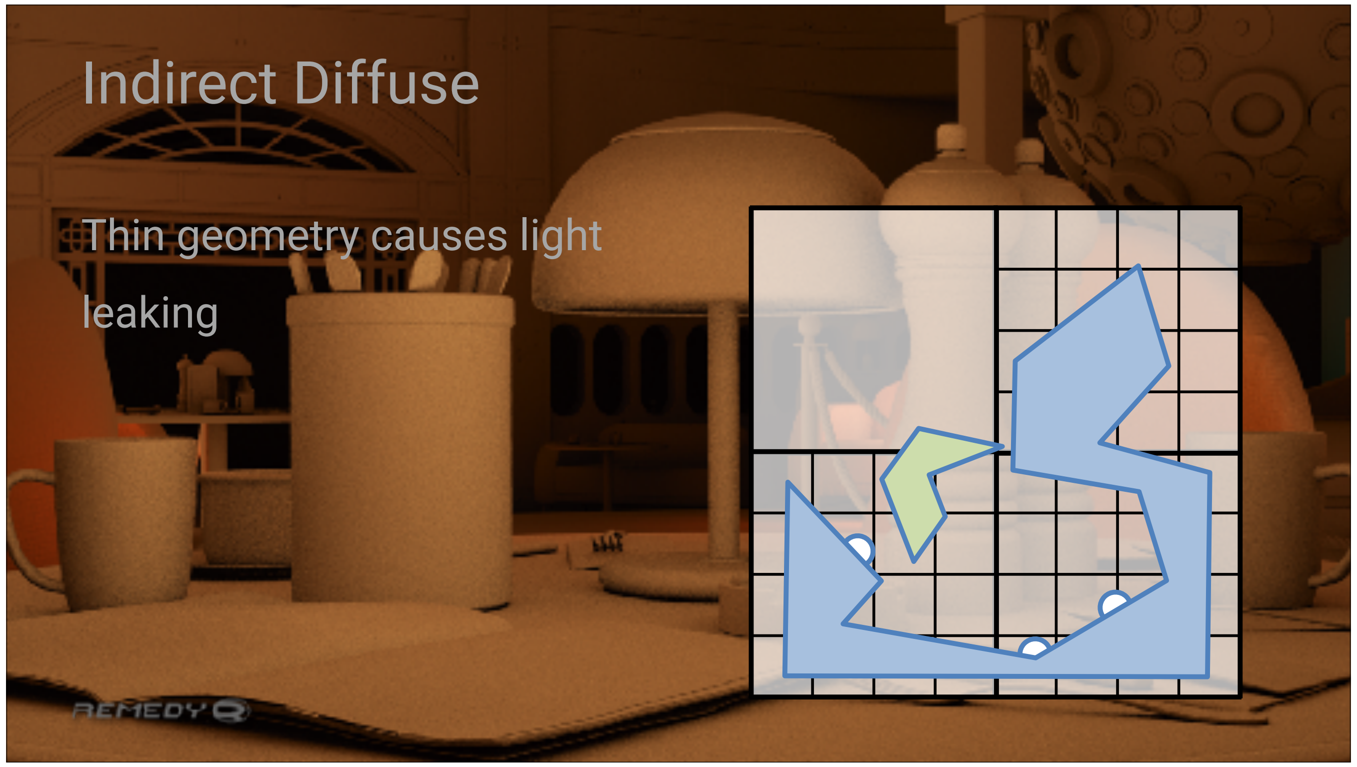
If we sample the lighting directly from the global illumination volume, we will miss dynamic geometry completely. We've been relying screen space ambient occlusion to tie dynamic geometry to rest of the scene. Even though using ambient occlusion works quite well, it darkens the image in unnatural way.



Indirect Diffuse

Trilinear sampling creates stair casing

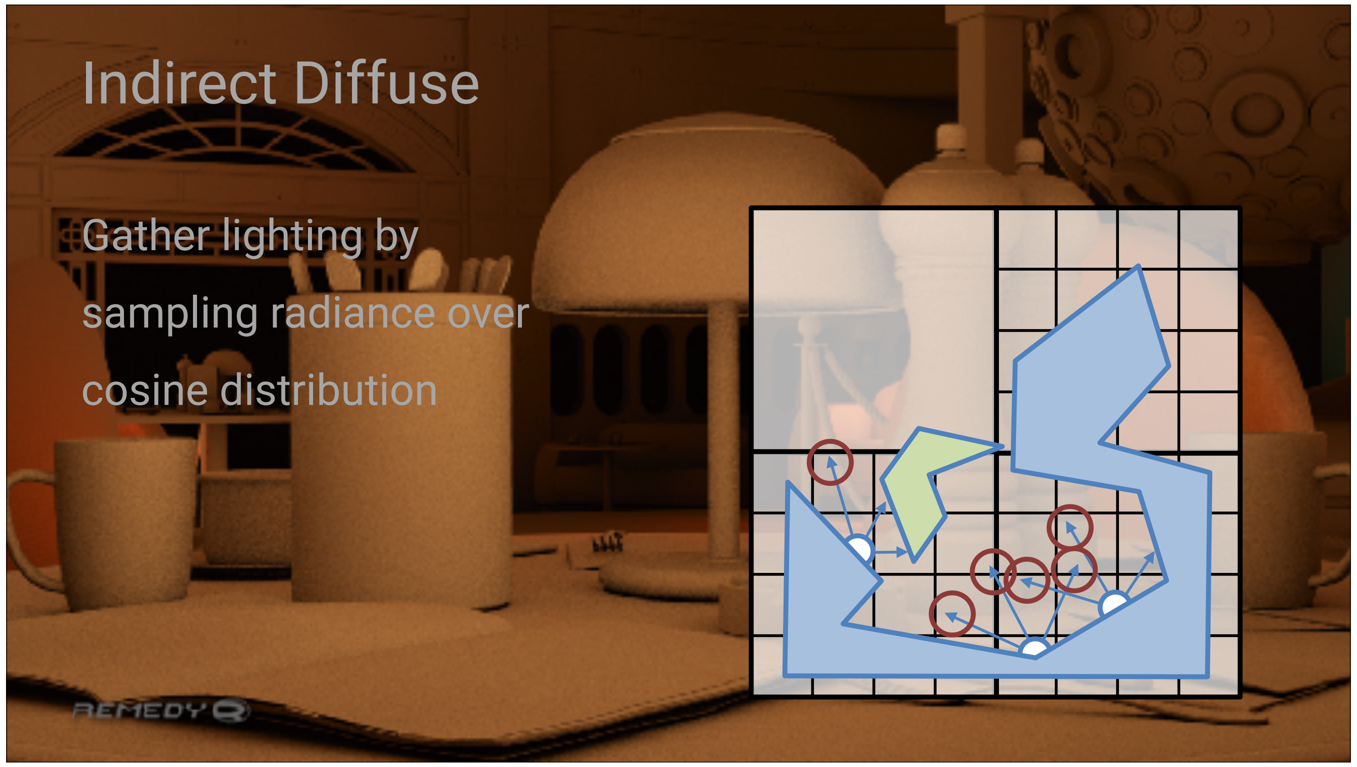
Other problem is that trilinear sampling of the volume data on surfaces that are not aligned to volume grid is also prone to stair stepping.



Indirect Diffuse

Thin geometry causes light leaking

Our sampling will also be very likely to leak if we have any thin geometry in the scene. We used geometry normal based fudge factor in Quantum Break to avoid some leaking, but the problem is really hard to solve entirely. Also, by altering interpolation, you easily end up just introducing new artefacts.



Indirect Diffuse

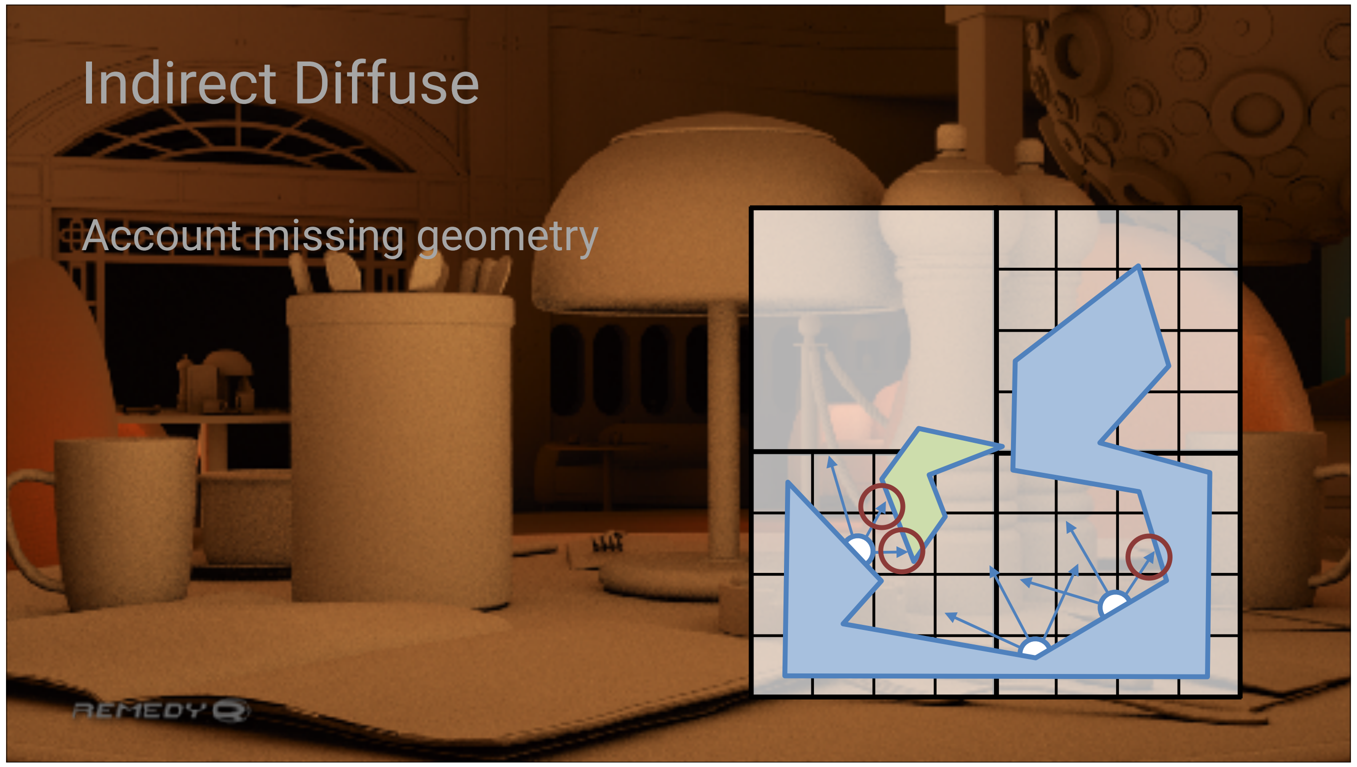
Gather lighting by sampling radiance over cosine distribution

Instead of sampling volume directly on surface, we can shoot short rays and sample global illumination on locations that didn't hit geometry. This will reduce leaking, and results in smoother interpolation.

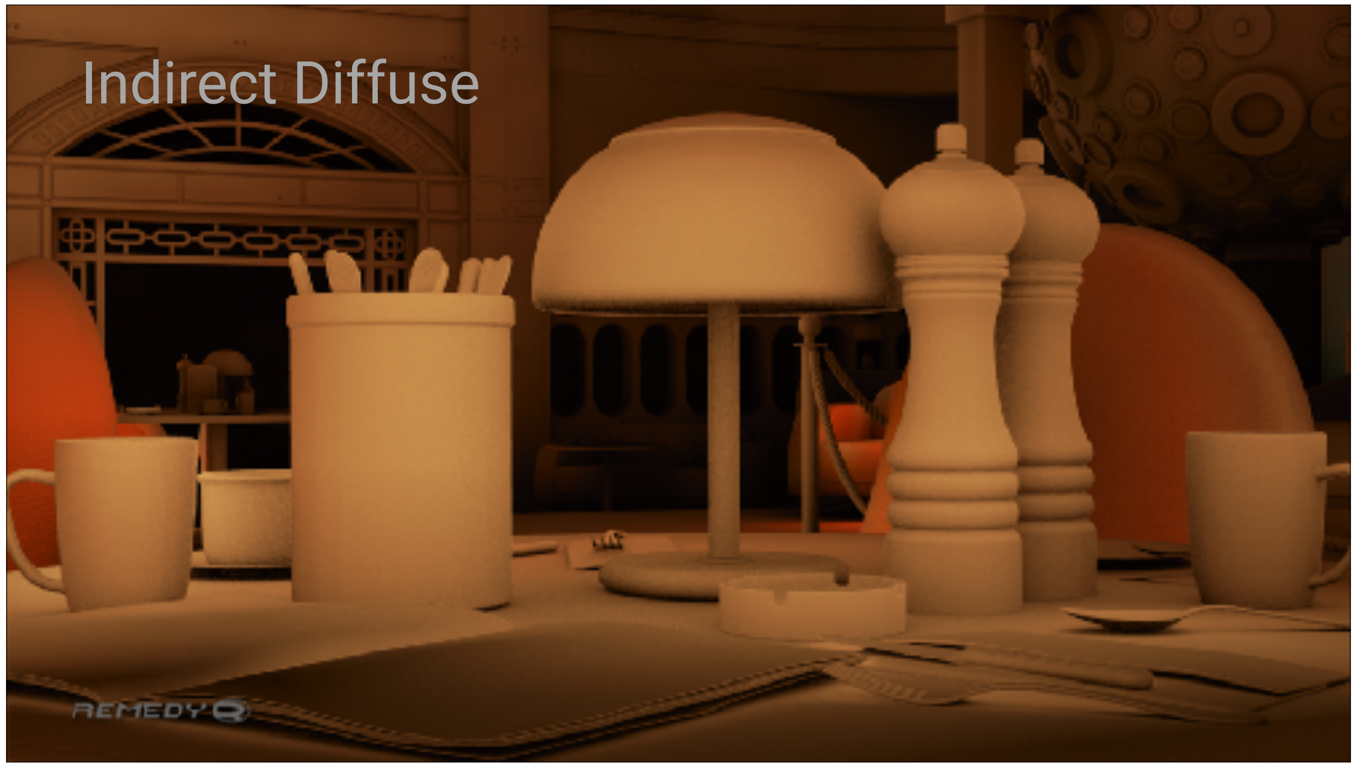
- Can still be close to geometry, but detecting that efficiently would require modifications to GI data, so we didn't try anything on that.

Indirect Diffuse

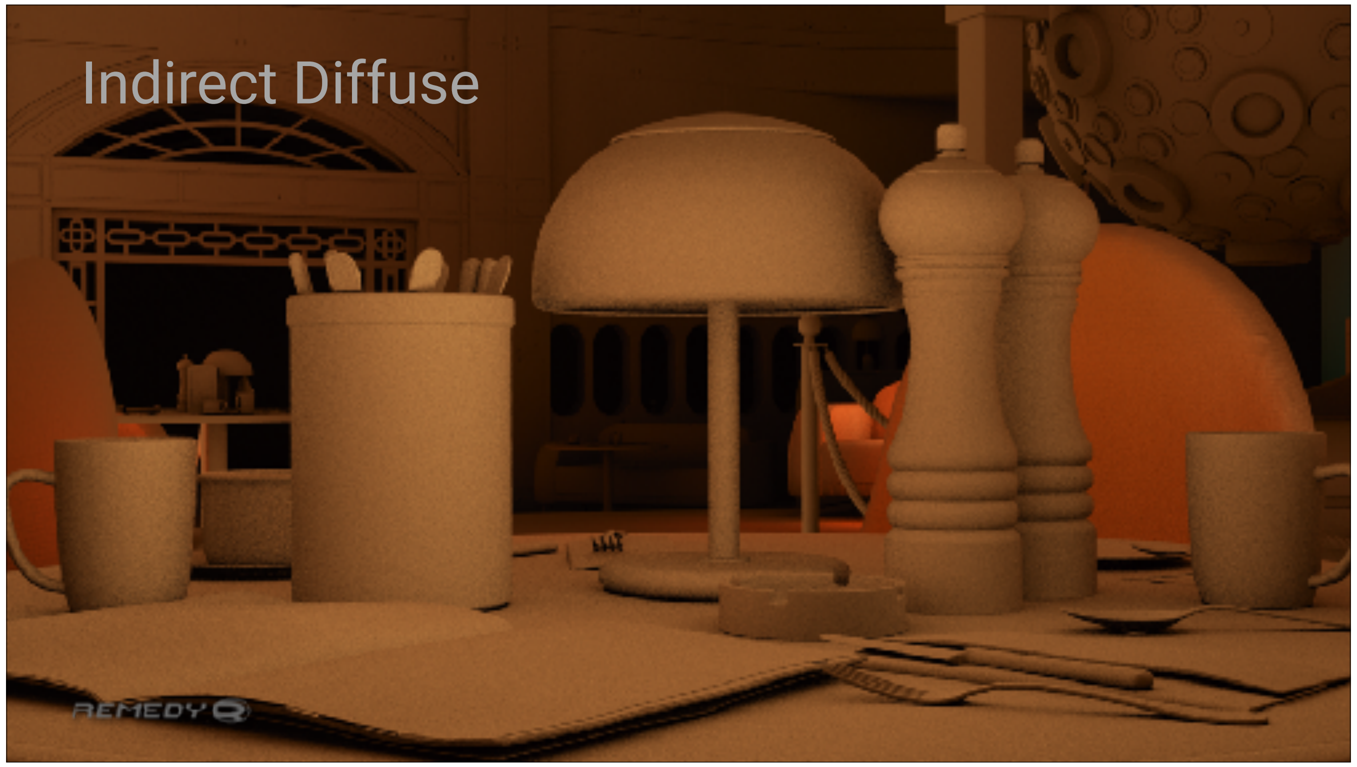
Account missing geometry



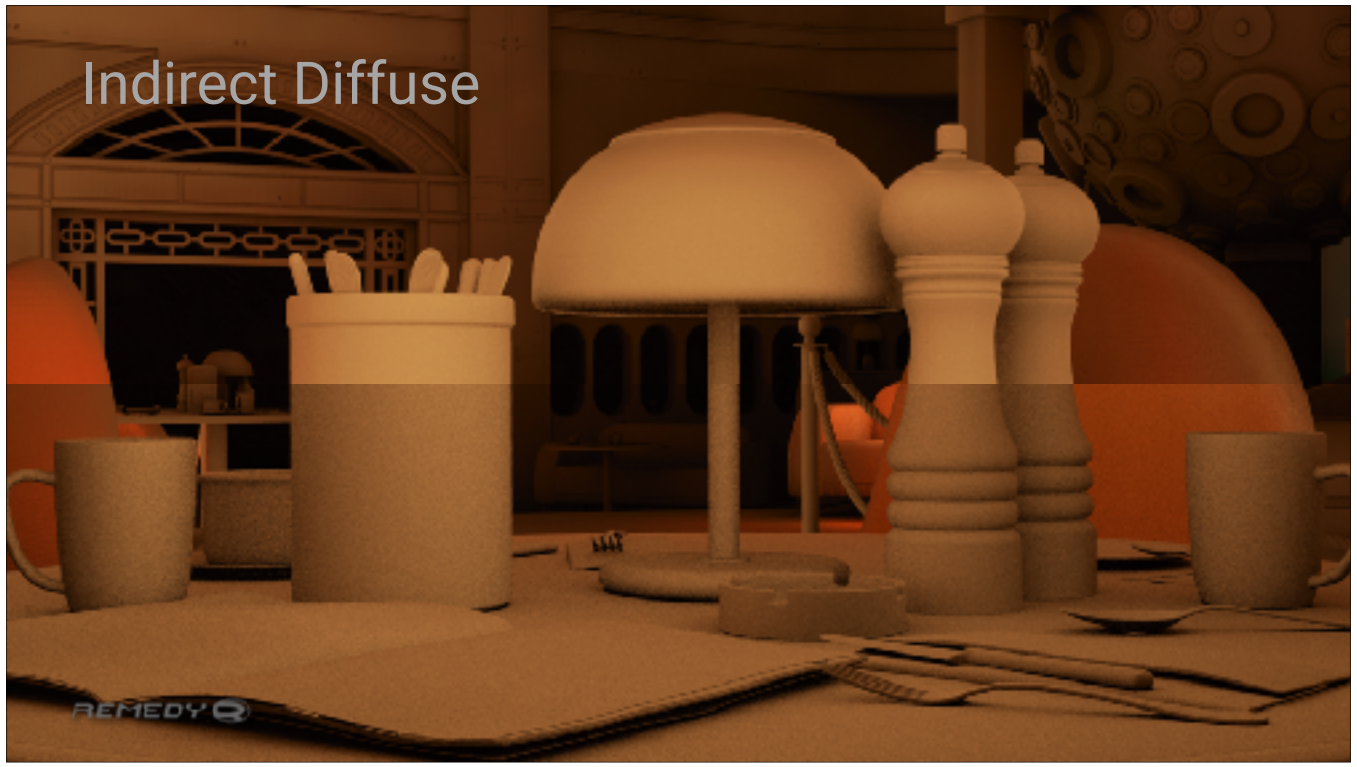
Simplest thing to do when hitting a geometry is to consider the sample black. This pretty much the same as calculating ambient occlusion, with benefits I mentioned earlier. Let’s look at comparison between this, and direct sampling of volume with ambient occlusion.



This is GI volume sampled directly and combined with raytraced ambient occlusion.

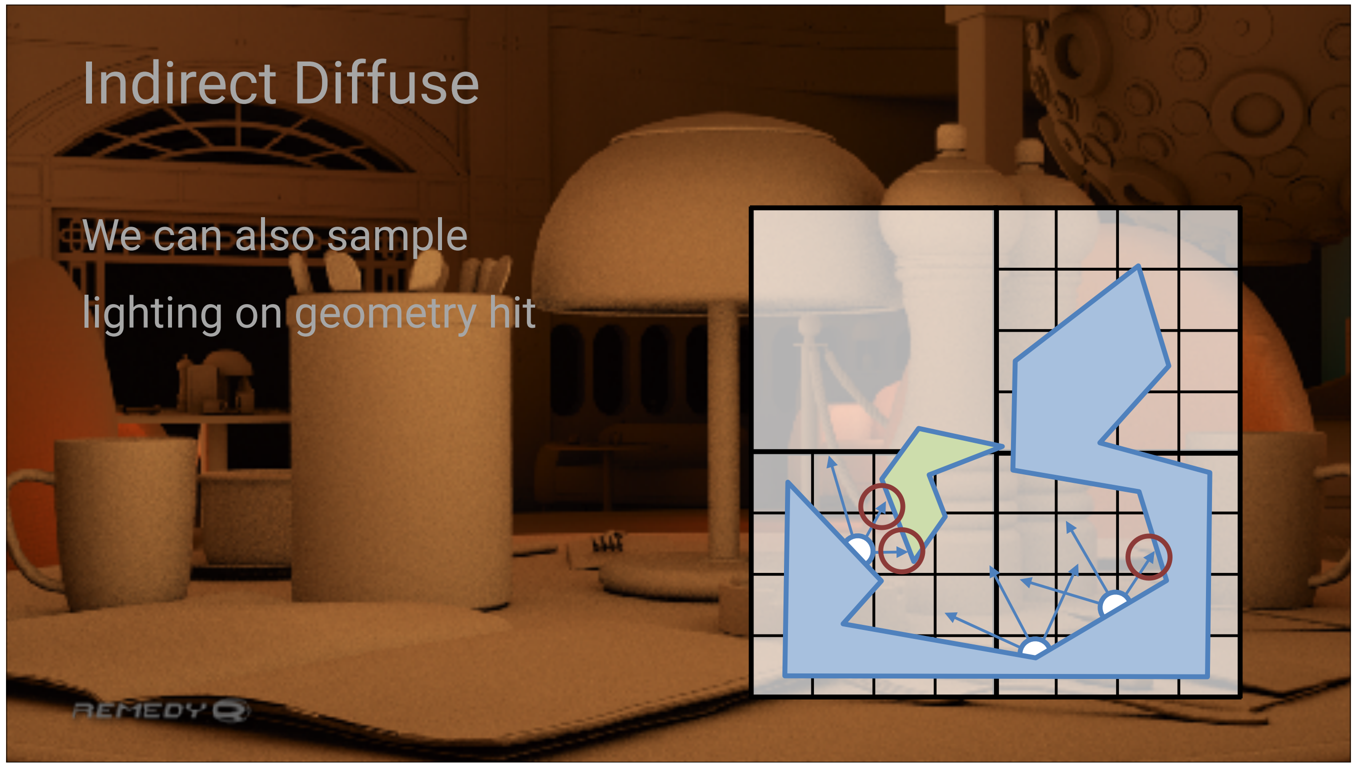


This is the same global illumination data sampled with few rays and and geometry hit locations considered black.



This is split image with upper part being the direct sampling with AO.

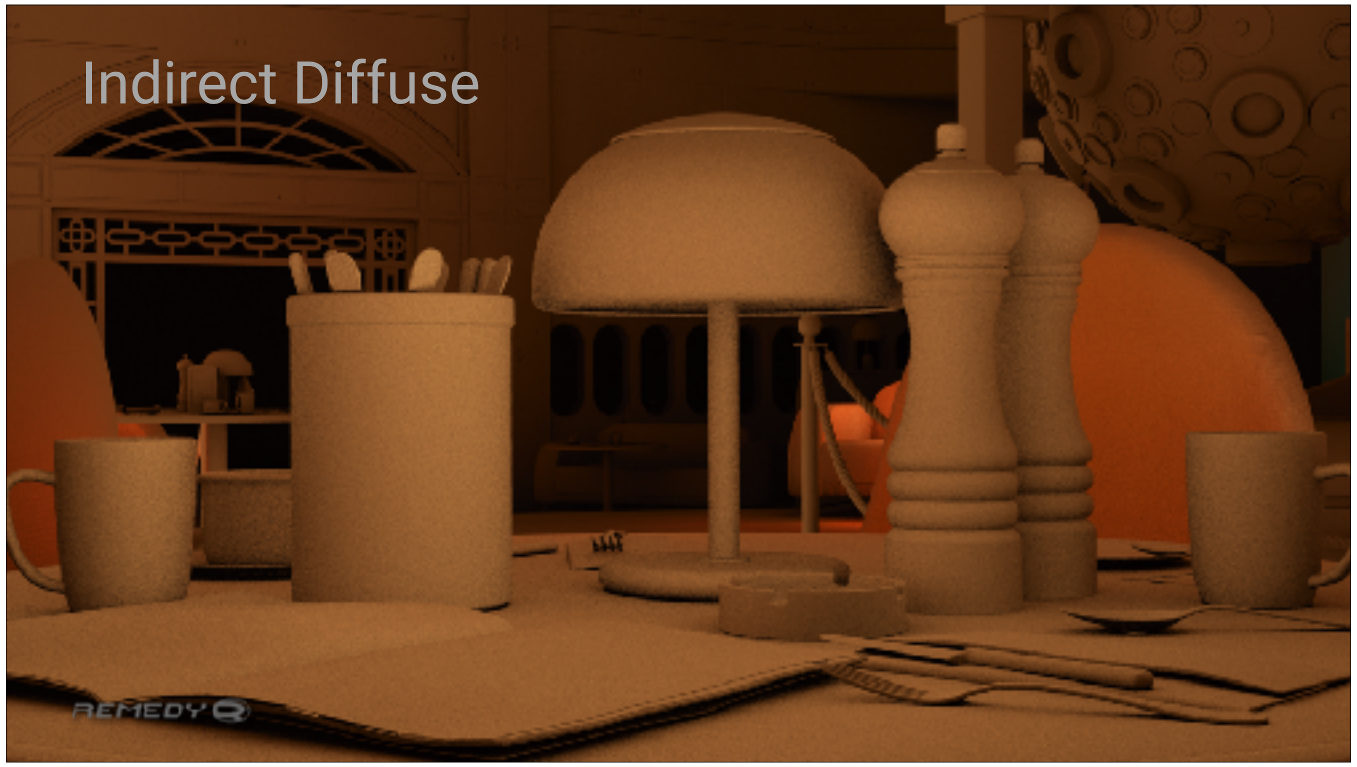
- did we manage to fix direct light leaking to surface itself? check!



Indirect Diffuse

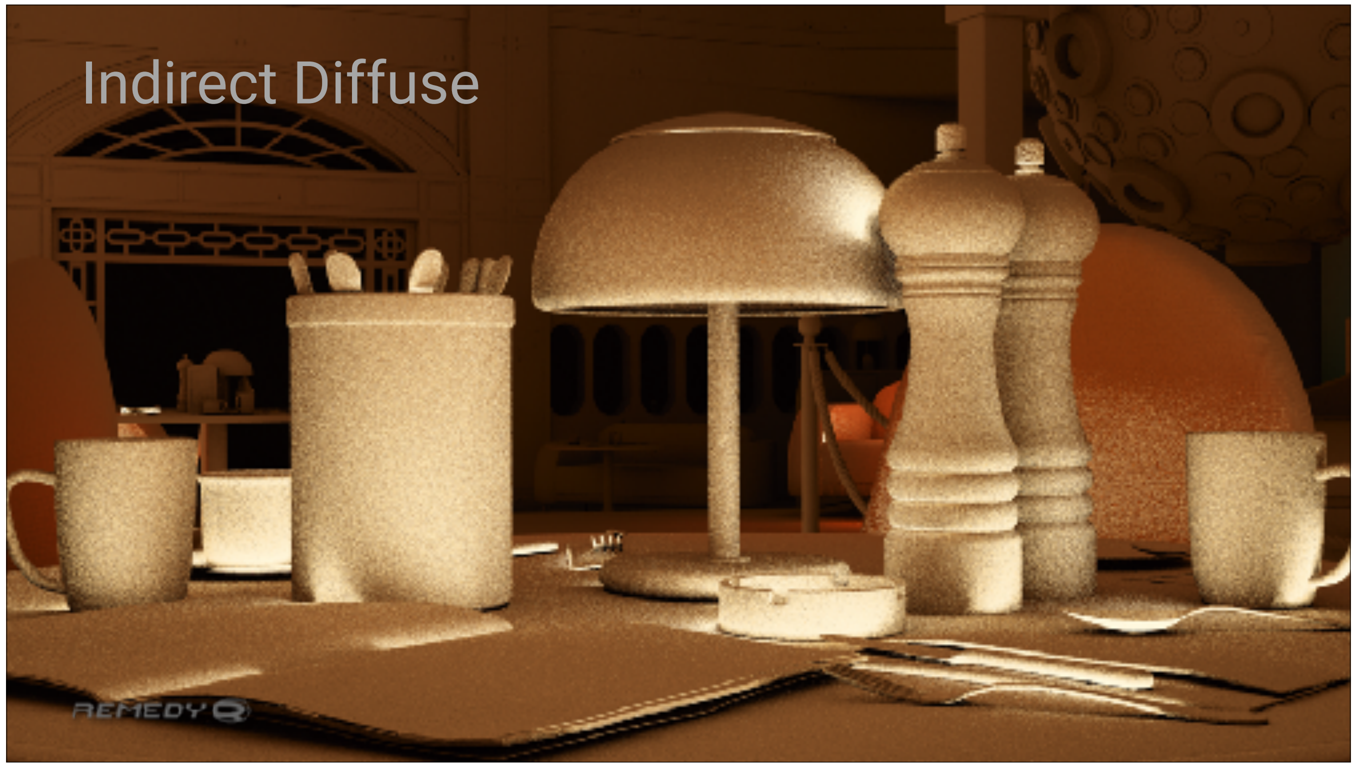
We can also sample lighting on geometry hit

Next obvious thing to try out is to sample actual lighting on the geometry hit locations.



This is the same global illumination data sampled with few rays and and geometry hit locations considered black.

Indirect Diffuse



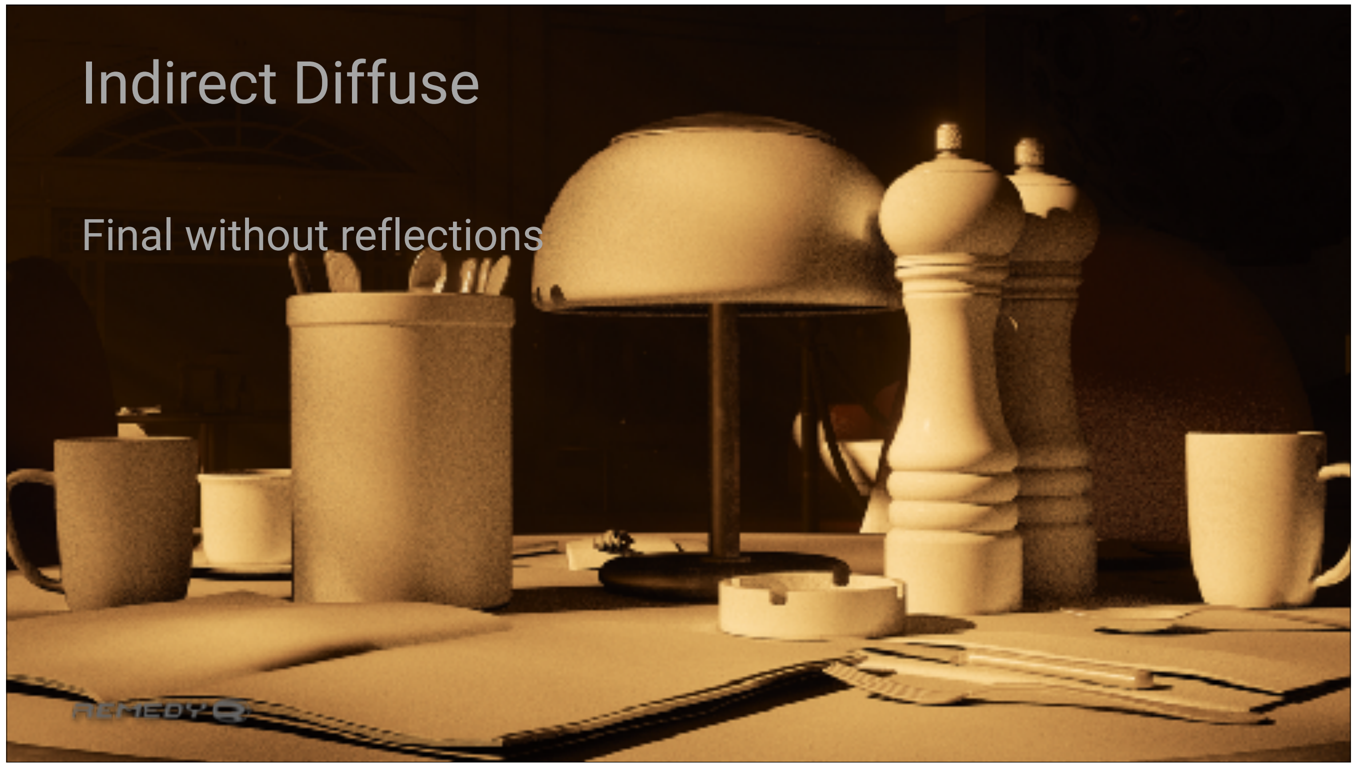
This will give you single bounce near field global illumination.

Indirect Diffuse

Visualise direct lighting that was used?

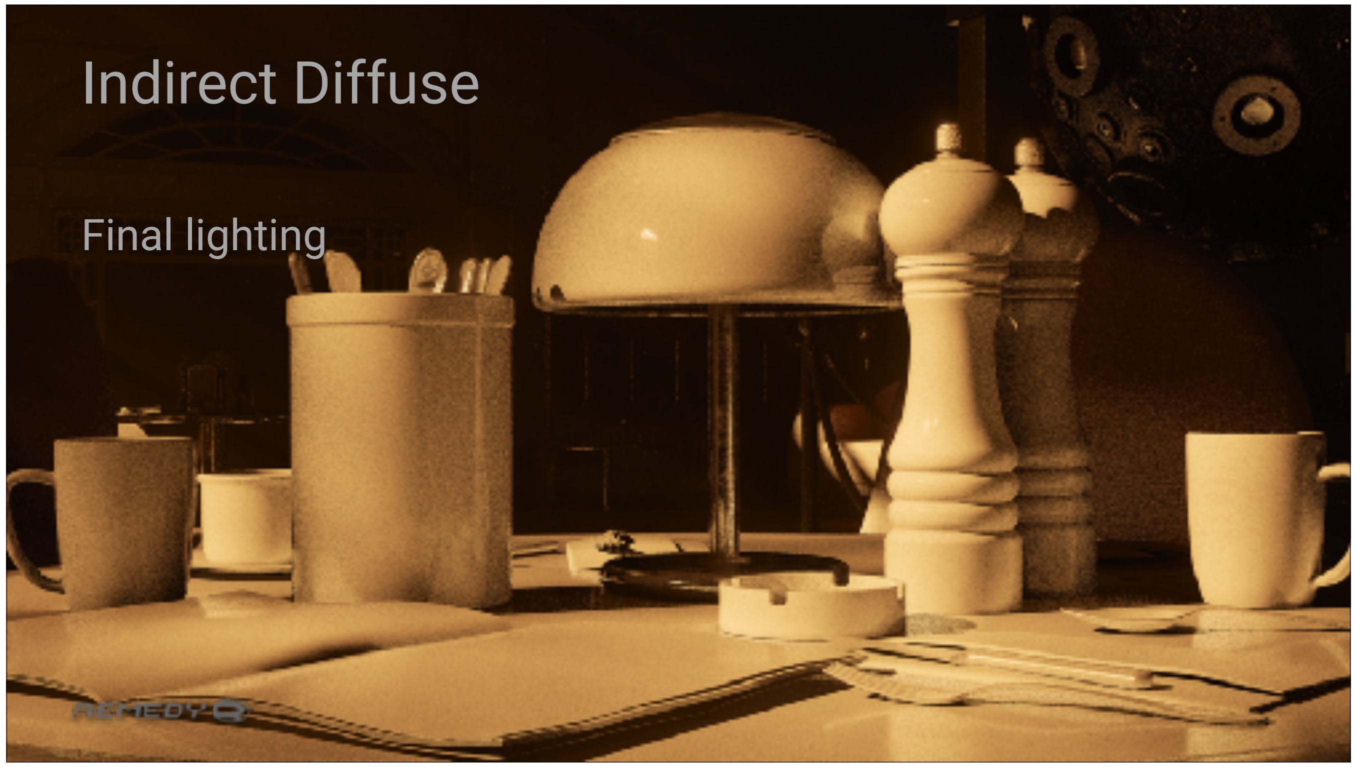
Indirect Diffuse

Final without reflections



Indirect Diffuse

Final lighting



Indirect Diffuse

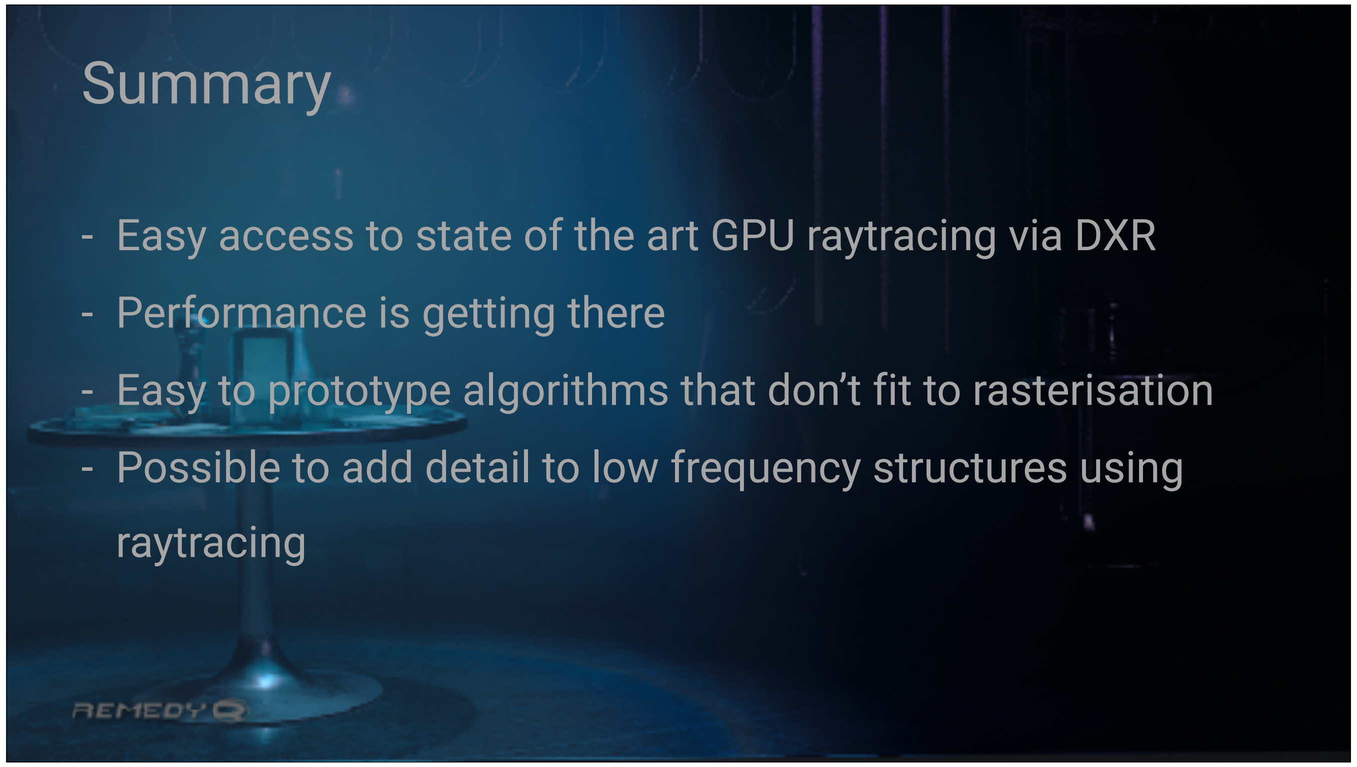
Final image

Indirect Diffuse
REMEDY

direct interpolation

Indirect Diffuse
REMEDY

rt interpolation



Summary

- Easy access to state of the art GPU raytracing via DXR
- Performance is getting there
- Easy to prototype algorithms that don't fit to rasterisation
- Possible to add detail to low frequency structures using raytracing

Pablo Fernandez

Juho Jousmäki

Sami Kastarinen

Benjamin Lindquist

Stuart MacDonald

Janne Pulkkinen

Juha Sjöholm (NVIDIA)

Teppo Ylitalo

Hiring

Remedy is hiring

Questions?