



Hello everyone and welcome. Thanks Alex for having me part of this course. My name is Colin Barré-Brisebois, and I work at SEED. This talk is a snapshot of the current state of the art in game raytracing, as well as a bunch of challenges the game dev community is looking into, and might want some help from its friends in academia. ☺

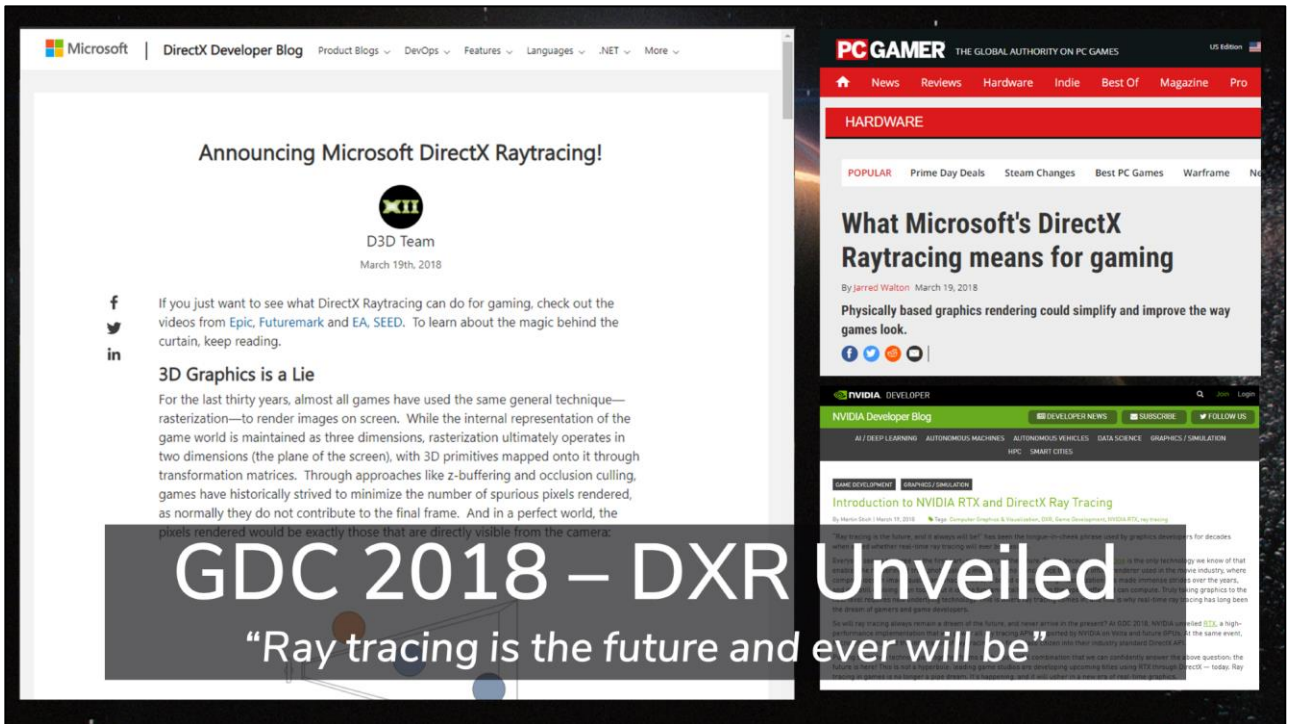


SEED

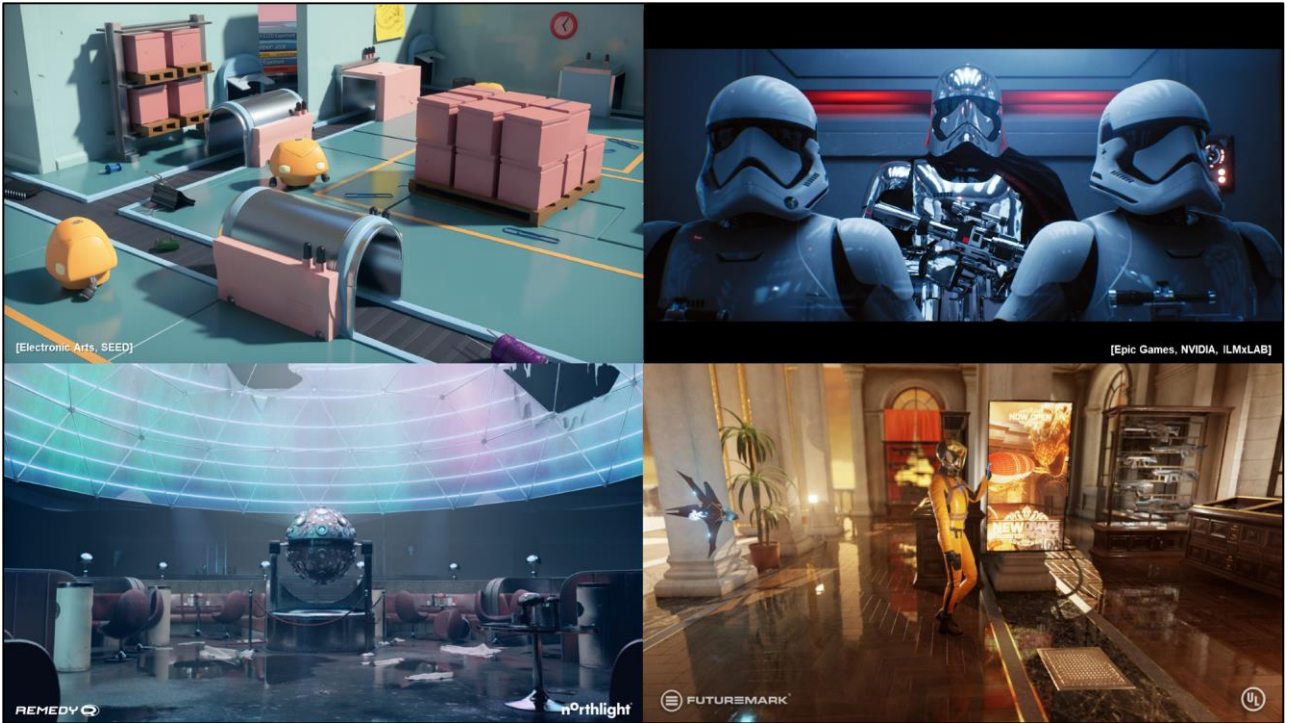
In case you don't know who SEED is, we're a technical and creative research division of Electronic Arts. We exist at EA as a cross-disciplinary team to deliver and foster disruptive innovation, for our games and our players. We focus on long-term applied research, but also try to stay relevant to the present by delivering artifacts along the way. Actually you might've seen some of our work for the launch of DirectX Raytracing, in collaboration with NVIDIA and Microsoft. We also have a bunch of presentations and publications under our belt. Actually, more than 40. I invite you to check our website, seed.ea.com.



Let's start with a question: how did a bunch of game developers started taking ray tracing seriously, and considered it as a robust solution to ship in their game products?




It should be said that several folks have played with and delivered games that rely on some aspects of ray tracing, so it’s not a new topic. But one can’t deny that the world of gamedev was taken by surprise at GDC 2018, when Microsoft announced DirectX Ray Tracing. A lot of minds were blown, but also a lot of unanswered questions, especially in terms of what is possible to achieve, at performance?



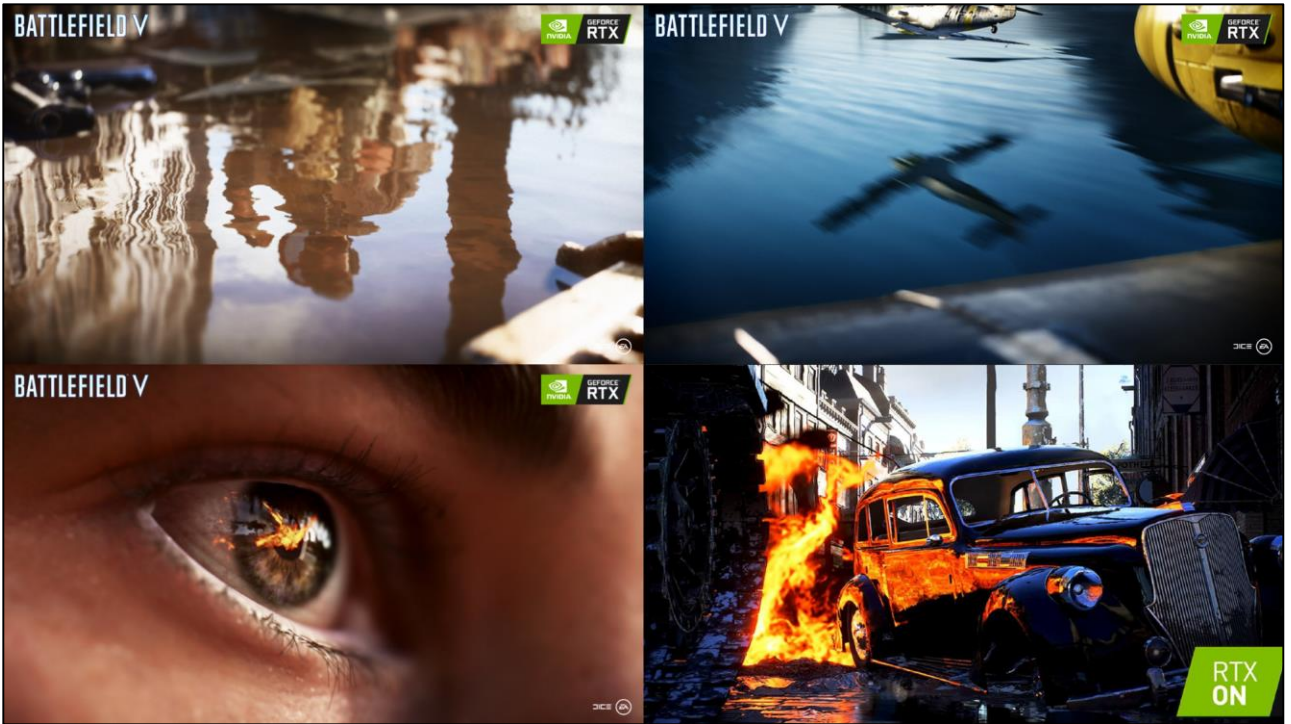
At SEED we felt very lucky to have been involved early on with Microsoft and NVIDIA, to see what could be done with this technology. The hybrid rendering pipeline we built for PICA PICA, allowed us to create visuals that are augmented with ray tracing and feature an almost path-traced quality look, at 2.5 samples per pixel. This was really challenging to build, but extremely fun too!

There was also this really cool demo from our Finnish friends at Remedy, featuring a bunch of ray tracing techniques in their Northlight engine, including reflections, ambient occlusion, indirect lighting and ray traced shadows. There was also this great demo from the folks at Futuremark.



Real-Time Ray Tracing in Software and Hardware

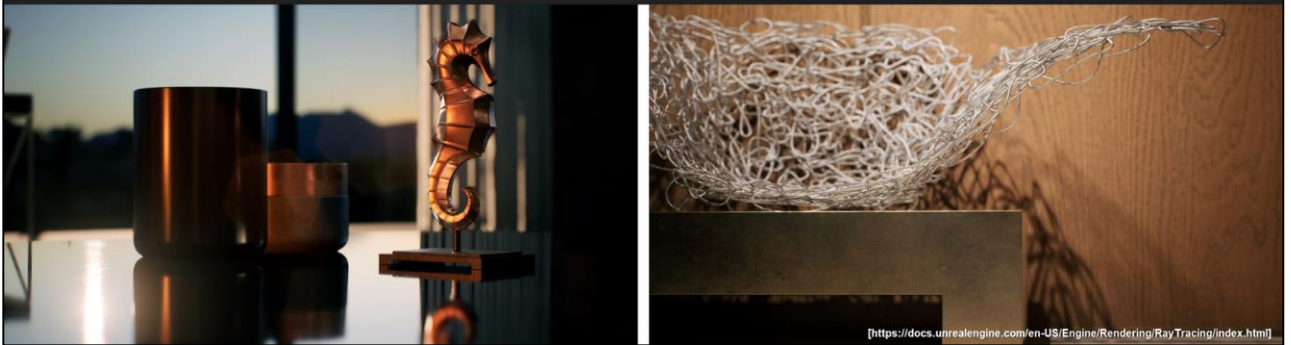
So real-time ray tracing was off to a great start! Especially since a few months later NVIDIA announced its new architecture titled Turing, which we all know accelerates ray tracing in hardware.



DICE's Battlefield 5 was the first game that shipped with real-time hybrid ray tracing using DXR, powered by EA's Frostbite engine. It features really awesome hybrid ray-traced reflections. Make sure to check our Jan and Johannes talk from GDC 2019.



Real-Time Hybrid Ray Tracing in Unreal Engine 4



Other big game engines like Unreal 4 also adopted ray tracing, and since release 4.22 ray tracing is now available for all to experiment with. They support both a hybrid mode, and a path-tracer reference mode, to compare against ground truth.



[Courtesy of Epic Games, Goodbye Kansas, Deep Forest Films]



[Courtesy of Epic Games, Goodbye Kansas, Deep Forest Films]



Rendered in real-time with Unity

[Tatarchuk 2019, Courtesy of Unity Technologies]



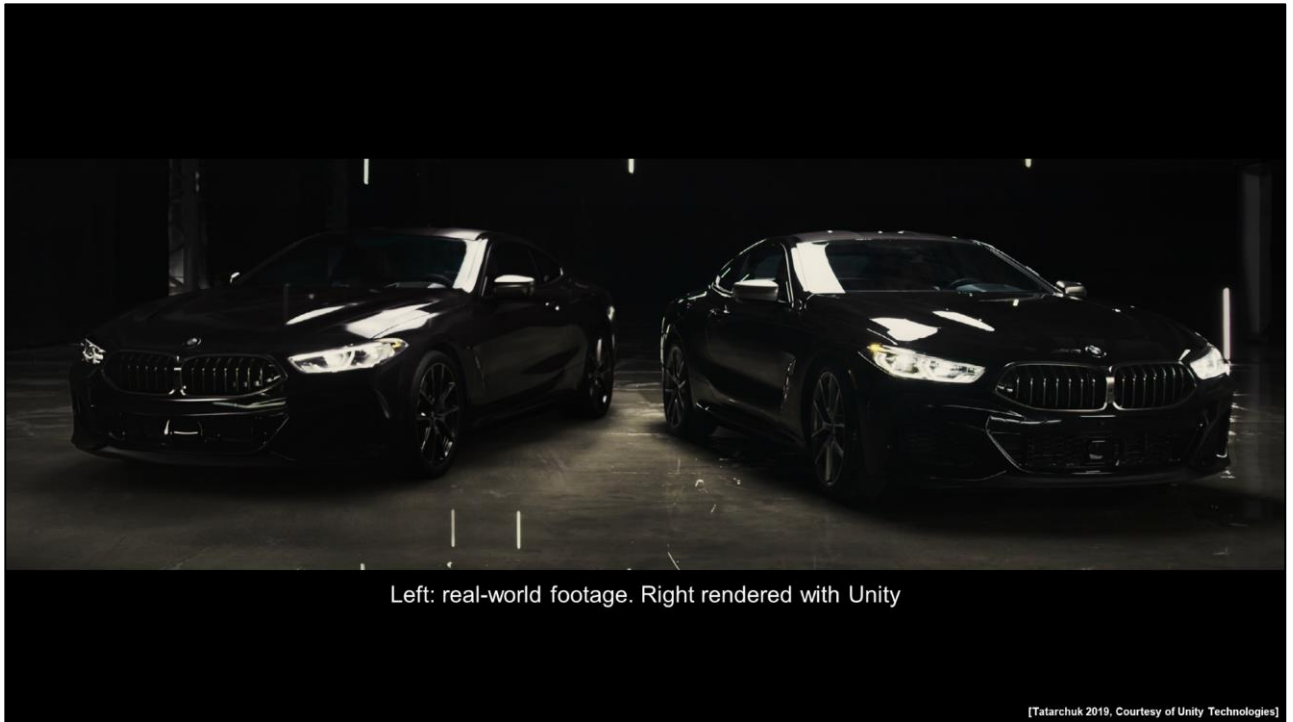
In this film they rendered at interactive rates on an RTX 2080TI in 4K [it was a 28M poly car], blurring the line between what is real and what is rendered.



Left: real-world footage. Right rendered with Unity

[Tatarchuk 2019, Courtesy of Unity Technologies]

This video shows off some of the effect they support, such as global reflections, multi-layer transparency with refraction, area lights, shadows, ambient occlusion and so much more



And here they placed a CG car into the filmed scene with the real-world car. Unless you look at subtitles, it's hard to tell which one is real.



- A bunch of games have also announced support for ray tracing, including Control, Metro Exodus, and Shadow of The Tombraider.
- Lately, Quake 2 was modified by Christoph Schied and features real-time path tracing, with reflections, refraction, shadows, ambient occlusion and GI.
- That's not the quake 2 I remember, and back then it looked awesome.
- This version just looks amazing. Mind completely blown.

And many more...

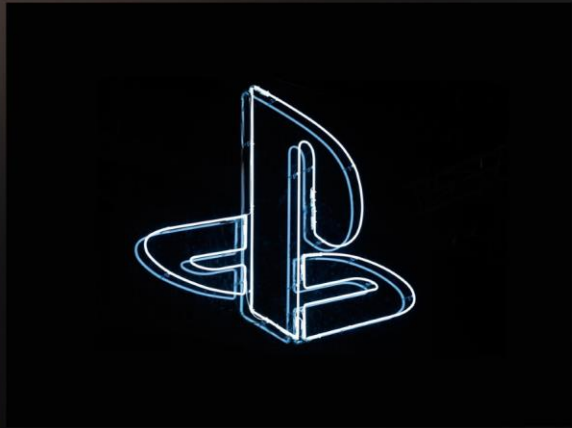
- Assetto Corsa
- Atomic Heart
- Call of Duty: Modern Warfare
- Cyberpunk 2077
- Enlisted
- Justice
- JX3
- Mech Warrior V: Mercenaries
- Project DH
- Stay in the Light
- Vampire: The Masquerade – Bloodlines 2
- Watch Dogs: Legion
- Wolfenstein: Youngblood

Just the beginning of real-time ray tracing making its way into game products

We're in for a great ride, and the work is not done! This is super exciting! 😊

And this is just the beginning of real-time ray tracing making its way into our products. So no, we're not done with ray tracing. We're in for a great ride, and we actually have quite a lot of work to do, which is super exciting!

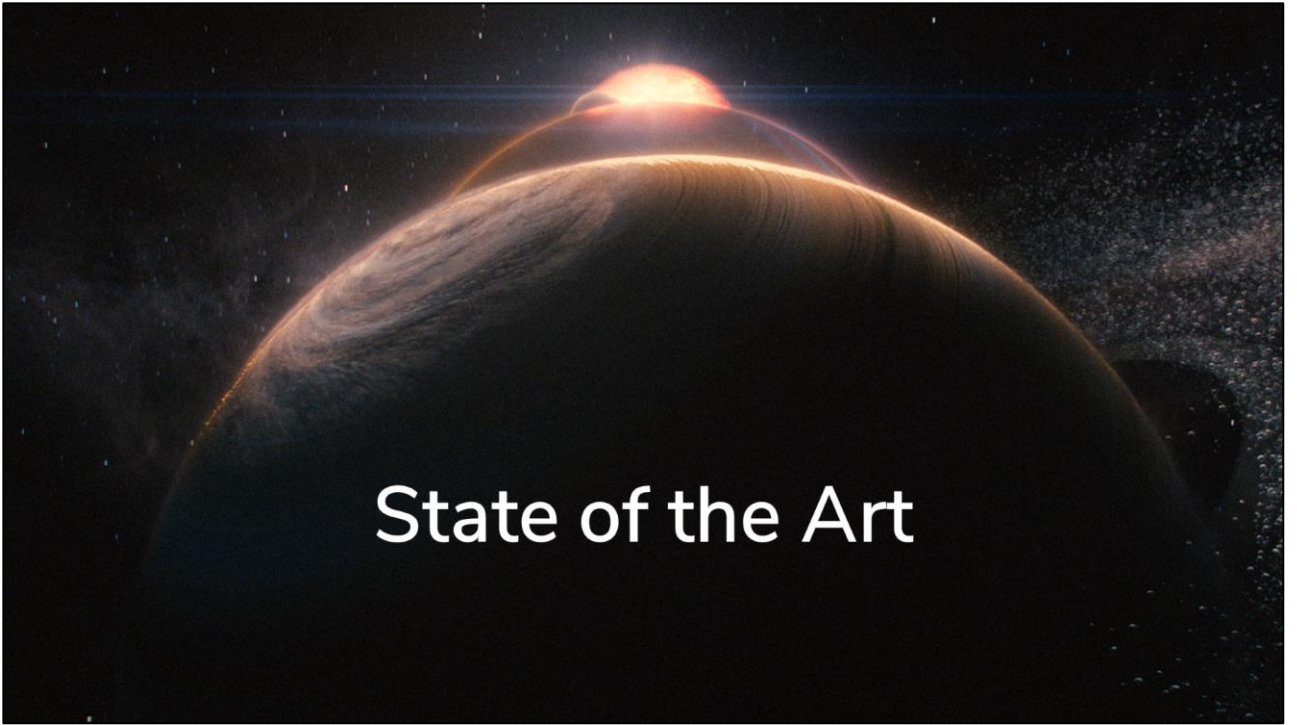
Future Consoles



→ Microsoft: E3 2019 Keynote, June 9th 2019, [link](#)

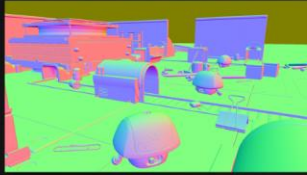
→ SONY: What to Expect From SONY's Next-Gen PlayStation, Wired Magazine, April 16th 2019, [link](#)

- Oh yeah, and let's not forget that two major console makers have announced support for ray tracing in their next iteration.
- So awesome!

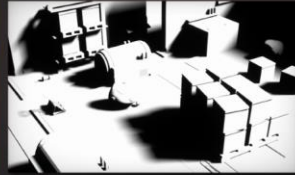


State of the Art

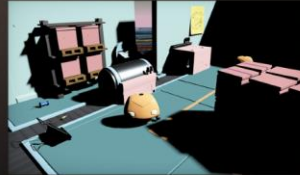
Hybrid Rendering Pipeline



Deferred shading
(*raster*)



Direct shadows
(*ray trace or raster*)



Lighting
(*compute + ray trace*)



Reflections
(*ray trace or compute*)



Global Illumination
(*compute and ray trace*)



Ambient occlusion
(*ray trace or compute*)



Transparency & Translucency
(*ray trace and compute*)



Post processing
(*compute*)

- The common denominator of these products is that they are mostly all built with a hybrid rendering pipeline that takes advantage of the best of rasterization, compute and raytracing
- The idea is that some techniques are built by chaining some stages after another, because that stage is best at doing what it does. For example, chaining ray tracing after generating relevant info stored in a UAV from a compute shader. Or grabbing all the hits from ray tracing, and shading them in a compute shader.
- To do this most have a standard deferred renderer with compute-based lighting, and a pretty standard post-fx stack.
- Parts of the pipeline are injected with ray tracing.



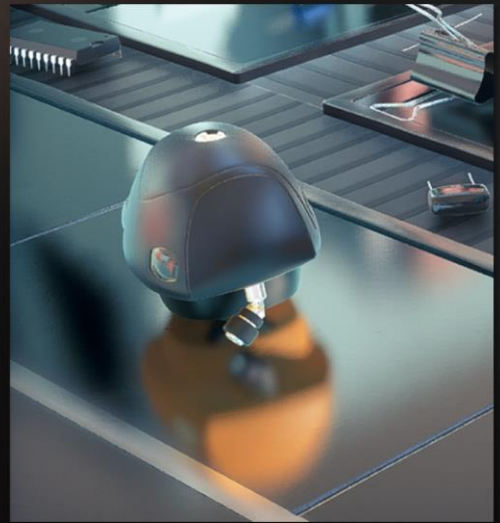
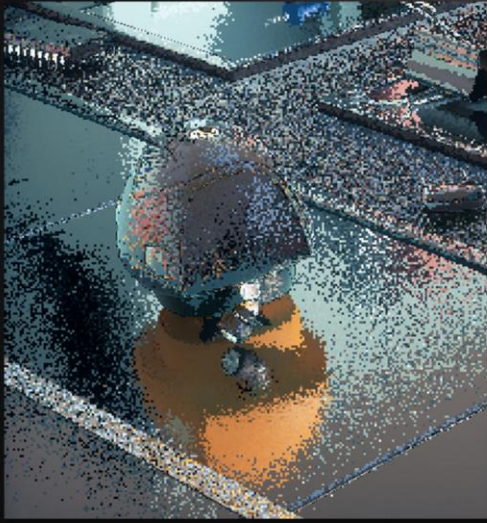
Let's first talk about reflections!



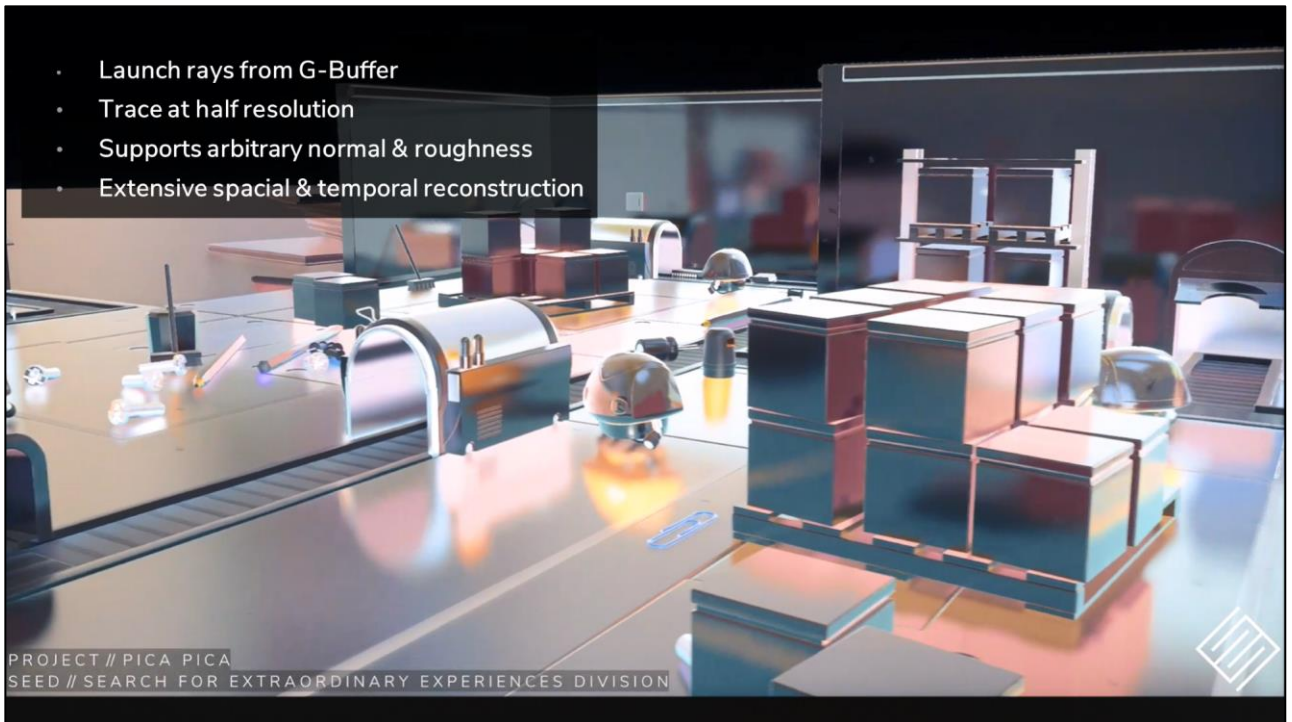
"It Just Works": Ray-Traced Reflections in 'Battlefield V' [Deligiannis 2019]

- Like I said Battlefield shipped with awesome reflections.
- This short video is an exert from Jan and Johannes' presentation, and really shows how much proper reflections add to a scene

Reflections

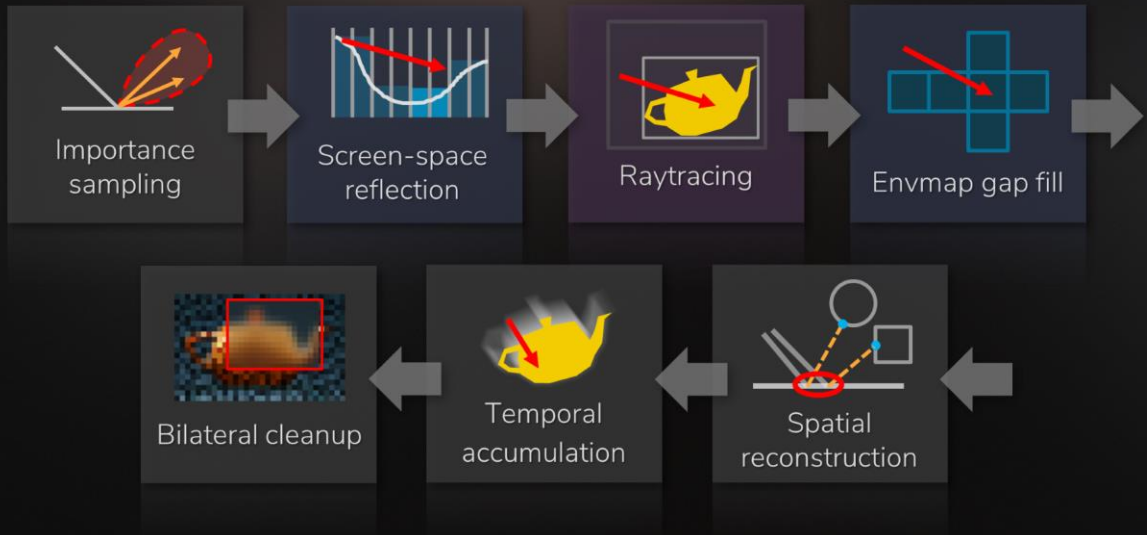


- The current idea with real-time ray-traced reflections is that we need more than 1 ray per pixel to fully capture the range of rough-to-smooth materials that a physically-based pipeline can describe.
- This becomes even more complex with multi-layer materials, even for the simple case like a material that has a base layer, and a finish.
- Perfectly sharp reflections are somewhat easy. The fun really starts when roughness goes up, or when surfaces become smooth.

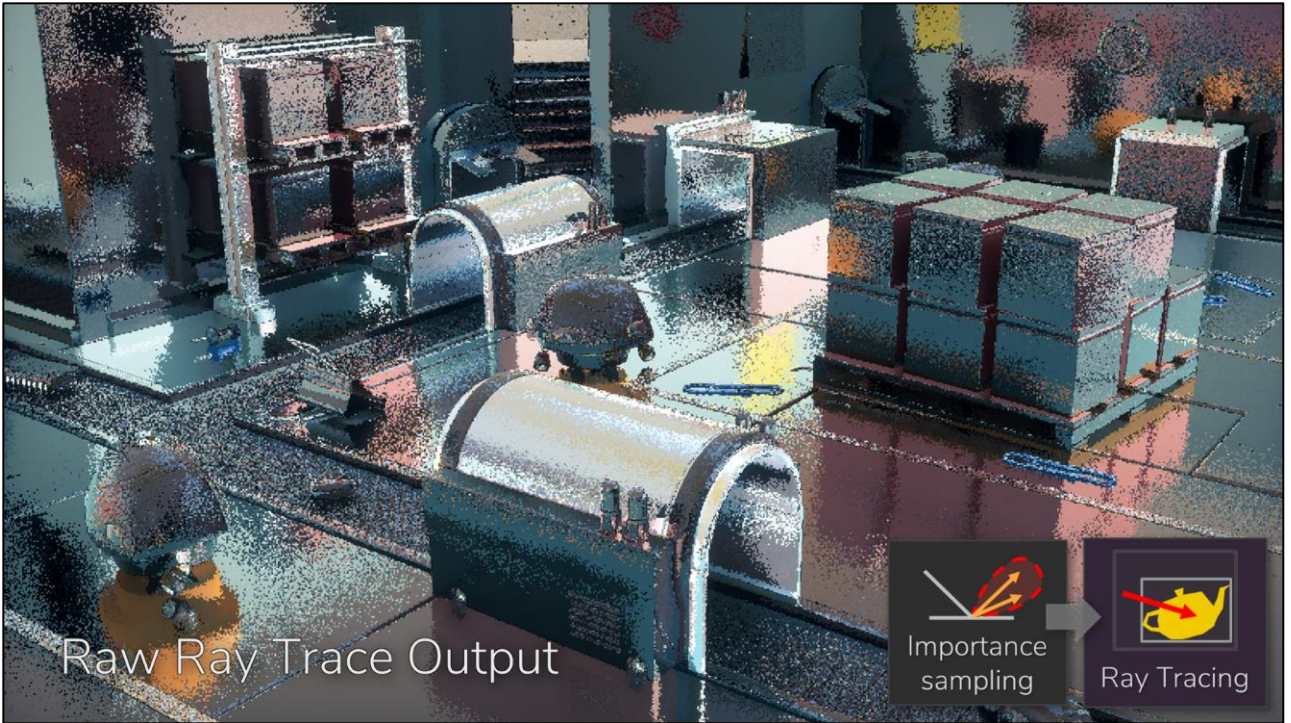


- In the context of a hybrid pipeline, reflections rays are launched from the g-buffer
- You can trace in full resolution, but for performance reason people have done it in half-resolution, which gives you one reflection ray for every 4 pixels.
- Then at the hit point, shadows will typically be sampled with another ray. This totals to $\frac{1}{2}$ ray per pixel. Alternatively you can sample your existing shadow maps, if don't want to launch a recursive rays, for performance reasons.
- If you want to support varying roughness and arbitrary normals, you will have to do some amount of reconstruction and filtering.
- Some approaches have a maximal roughness level, so you will have to find another source, like prefiltered environment maps. You can also combine this with screen space reflections for performance.

Reflection Pipeline



- Here's a high-level summary of a hybrid ray tracing pipeline for reflections.
- First you generate rays via BRDF importance sampling, which gives you rays that follow the properties of materials.
- Scene intersection can then be done either by screen-space raymarching or ray tracing. In the video I just showed we only ray trace.
- Once intersections are found, you reconstruct the reflected image. This is either done in-place, or separately for improved coherency. I'll talk about this in a few slides.
- Your upsampling kernel reuses ray hit information across pixels when upsampling the image to full-resolution.
- Often a last-chance noise cleanup in the form of a cross-bilateral filter runs as the last step.



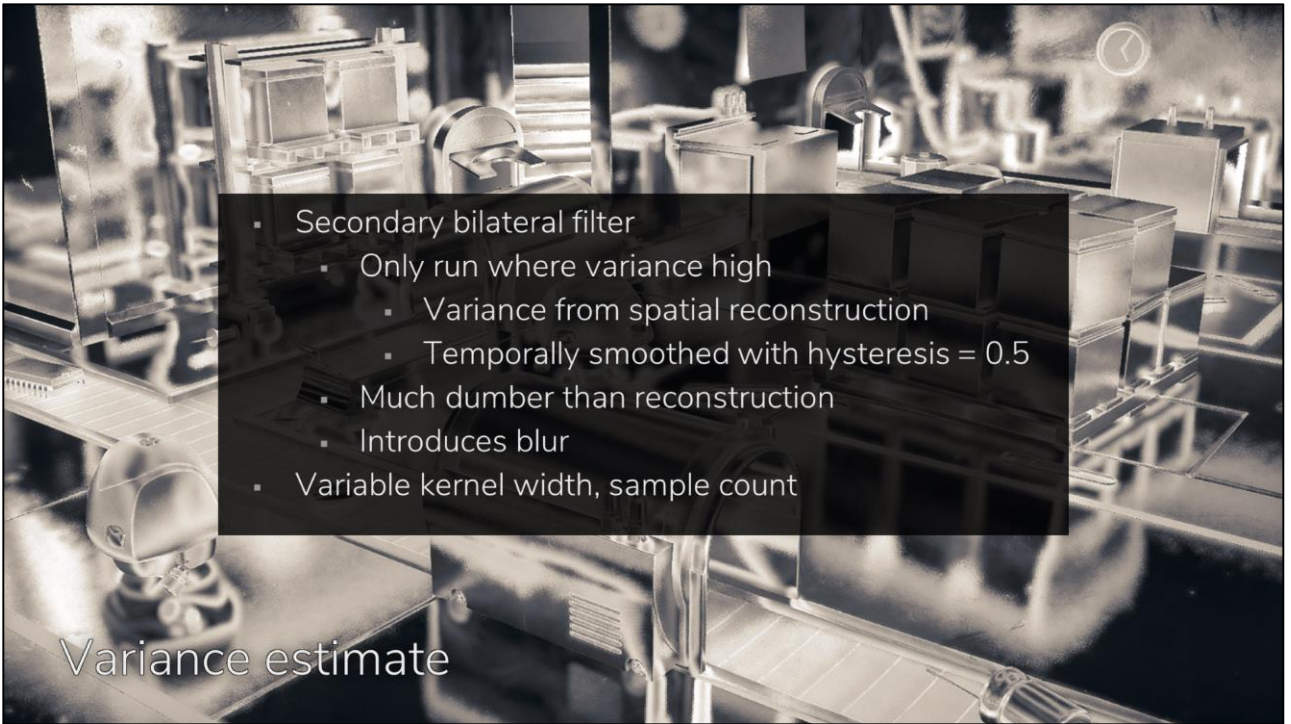
Looking only at the reflections, this is the raw results we get at 1 reflection ray every 4 pixels.



- And this is what the spatial filter does with it. The output is still noisy, but it is now full rez, and it gives us variance reduction similar to actually shooting 16 rays per pixel.
- Every full resolution pixel basically uses a set of ray hits to reconstruct its reflection.
- It's a fancy weighted average where the local pixel's BRDF is used to weigh contributions.



- Followed by temporal accumulation



- Secondary bilateral filter
 - Only run where variance high
 - Variance from spatial reconstruction
 - Temporally smoothed with hysteresis = 0.5
 - Much dumber than reconstruction
 - Introduces blur
 - Variable kernel width, sample count

Variance estimate

- And finally by a much simpler bilateral filter that removes up some of the remaining noise.
- It overblurs a bit, but it's needed for some of the rougher reflections.
- Compared to SSR, ray tracing is trickier because we can't cheat with a blurred version of the screen for pre-filtered radiance
- There's much more noise compared to SSR, so our filters need to be more aggressive too.
- To prevent it from overblurring, the variance estimate from the spatial reconstruction pass is used to scale down the bilateral kernel size and sample count



+Bilateral cleanup

Bilateral cleanup



- And then of course we sprinkle some TAA on top, because TAA "fixes everything" and the remaining noise, and we get a pretty clean image.
- Considering this comes from one quarter rays per pixel per frame, and works with dynamic camera and object movement, it's quite awesome what can be done when reusing spatial and temporal data



Raw Ray-traced Output

- Going back to the raw output, for comparison

Alternatively, Hybrid RT + SSR



As mentioned, you can also combine screen space reflections with ray tracing. This is what Jan and Johannes from DICE have presented back at GDC, and featured in Battlefield 5.

Hybrid RT & SSR

- Figure out which pixels can rely on screen space results
 - Otherwise, trace in world

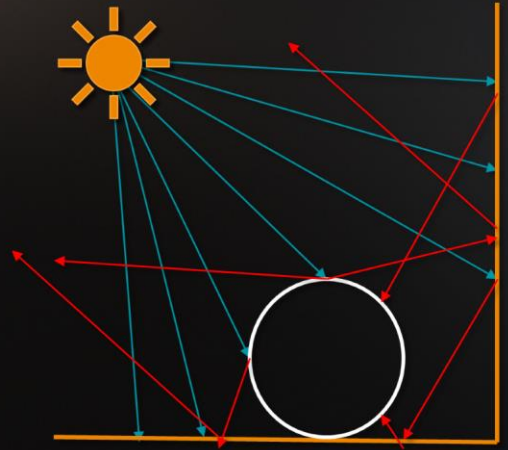
- Performance [Deligiannis 2019]
 - Variable Rate Tracing
 - More rays for water & grazing angles
 - Ray Binning



- The general idea when blending between SSR and ray tracing is that one has to figure out which pixels can rely on screen-space results. If that's the case, you can use that result. Otherwise you trace in the world.
- The challenge here is achieving that fine balance and aligning results from screen space, with results from world space tracing.
- Once done the results can look great, as shown here. Here's an overview of their whole pipeline, with some performance numbers on a 2080TI. This should give you a good idea of some of the steps needed to achieve this.
- Additionally the folks at DICE have presented an approach that tweaking the ray count shows significant performance improvement, and binning the rays.
 - Red high ray count
 - Blue low ray count
 - Yellow in Between

Managing Coherency

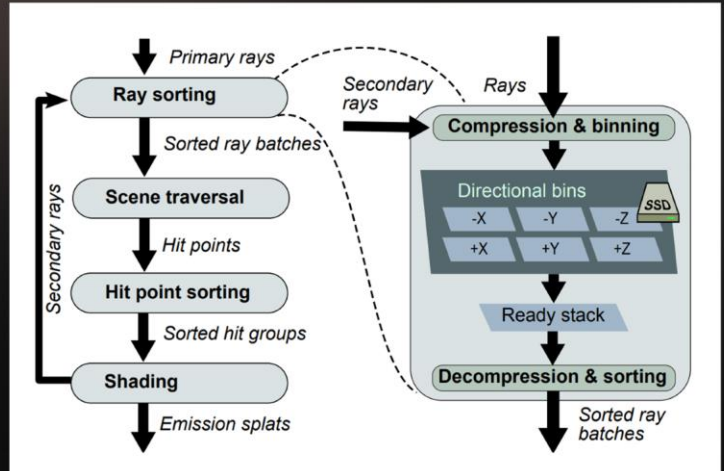
- **Coherency** is key for RTRT performance
 - **Coherent** → adjacent work performing similar operations & memory access
 - Camera rays, texture-space shading
 - **Incoherent** → thrash caches, kills performance
 - Reflection, shadows, refraction, Monte Carlo
- You're on your own: hardware won't take care of it for you



- Managing coherency is key for real-time ray tracing performance
- You will get some adjacent rays that perform similar operations and memory accesses, and those will perform well, while some might trash cache and affect performance
- Depending on what techniques you implement, you will have to keep this in mind, as the current hardware won't do this for you.
- Can't expect out-of-core ray sorting and coherency construction from total mess. Still need to tackle coherency upfront in the techniques & algorithms we develop.

Managing Coherency

- **Inspiration from Offline**
 - Sort large out-of-core ray batches & ray-hits for deferred sharing
- A few options: [Aalto2018] [Benyoub2019] [Deligiannis 2019]
 - Use shadow maps for reflection shadows
 - Split ray tracing and shading
 - Group shading per material
 - Limit tracing on roughness



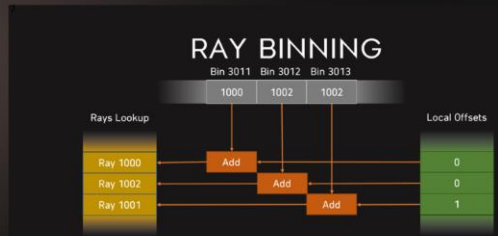
Sorted Deferred Shading for Production Path Tracing [Eisenacher 2013]

- Here we can take inspiration from offline, and this is what several have demonstrated.
- You can use shadow maps for reflection shadows, and therefore not do recursive rays.
- You can bin rays, and split tracing and shading
- You can group shading per material
- And also limit tracing based on roughness.

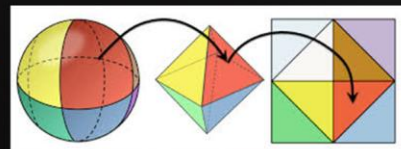
Ray Binning

Group rays that are directionally aligned to maximize coherency [Deligiannis 2019] [Benyoub 2019] [Majercik 2019]

1. Split the screen in (32x32) tiles
2. Generate (random) rays
3. Sort rays in octahedral space for ray direction binning
4. For each bin, launch rays
5. Gather hit results in G-Buffer
6. Shade in Compute Shader



"It Just Works": Ray-Traced Reflections in 'Battlefield V' [Deligiannis 2019]



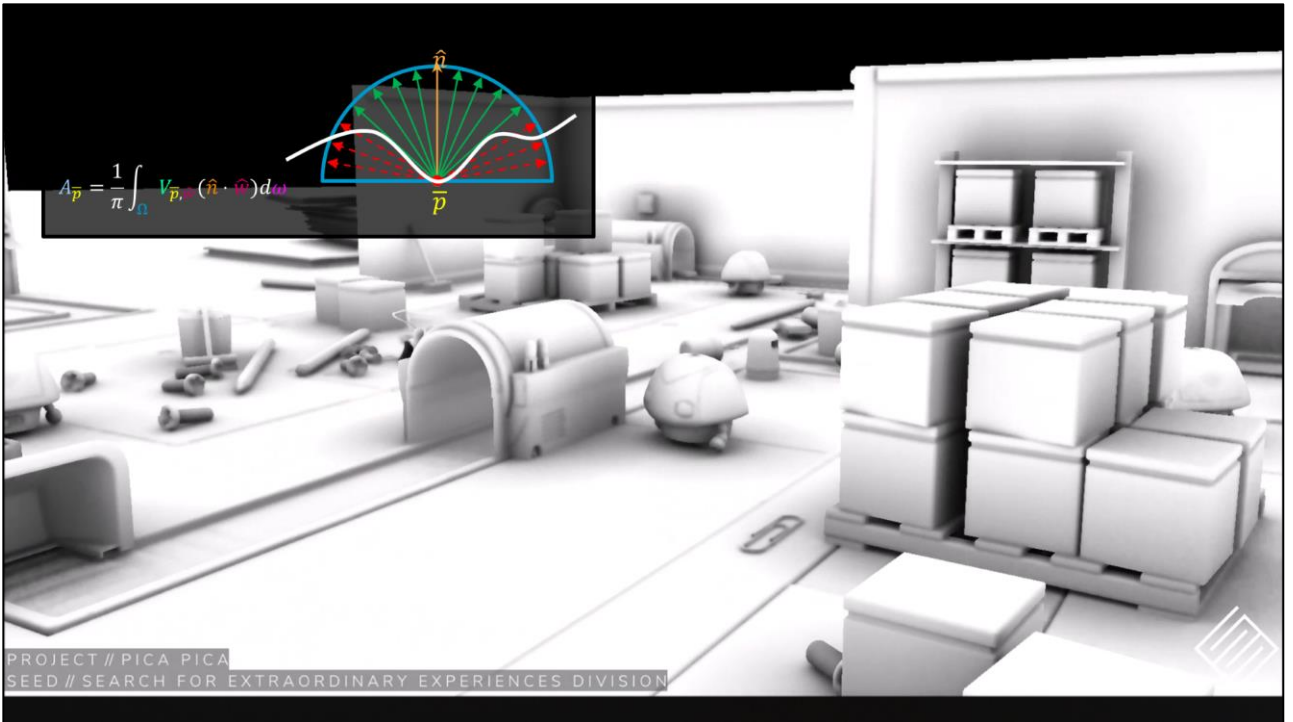
A Survey of Efficient Representations for Independent Unit Vectors [Cigolle 2014]

- Speaking of managing coherency, grouping rays that are directionally aligned to maximize coherency
- The general idea here is to split the screen in tiles, and sort randomly generated rays in some kind of space that allows you to bucket them by direction.
- Octahedral space is perfect for this.
- Then, for each bin you launch the rays and gather the hits, and output to a UAV or Gbuffer
- You can then light and shade those hits results in a compute shader, which gives you better control over SIMD usage.



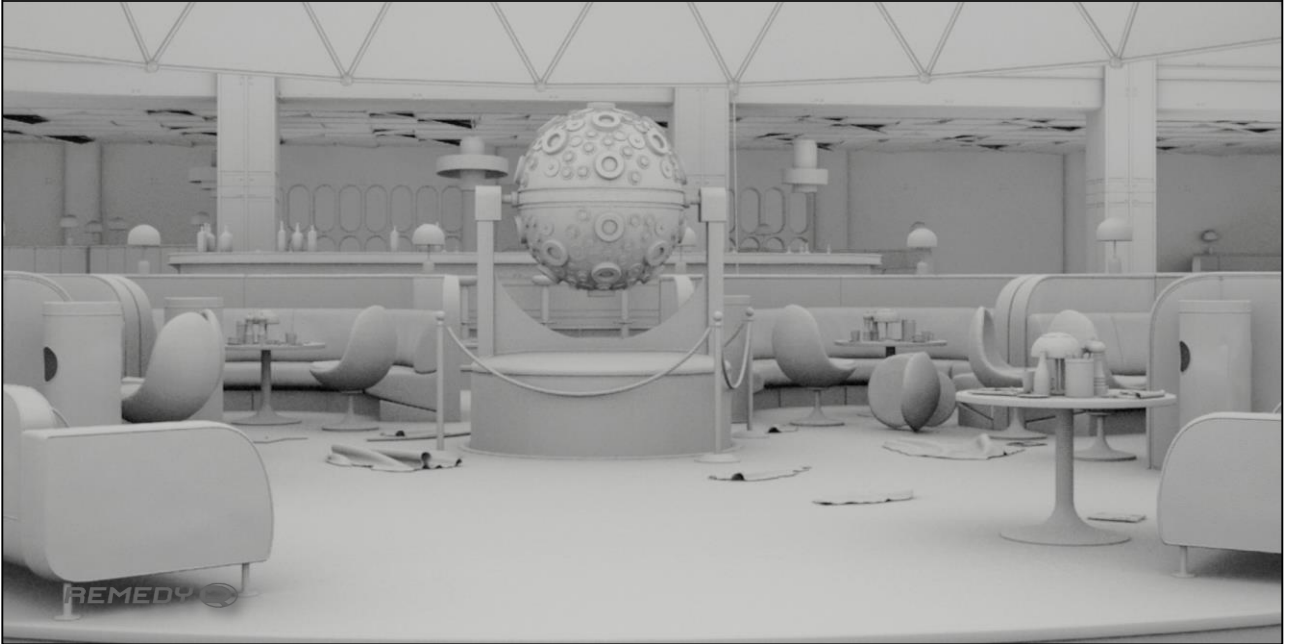
Ambient Occlusion

- Another technique that maps and scales well to real-time ray tracing is of course ambient occlusion.

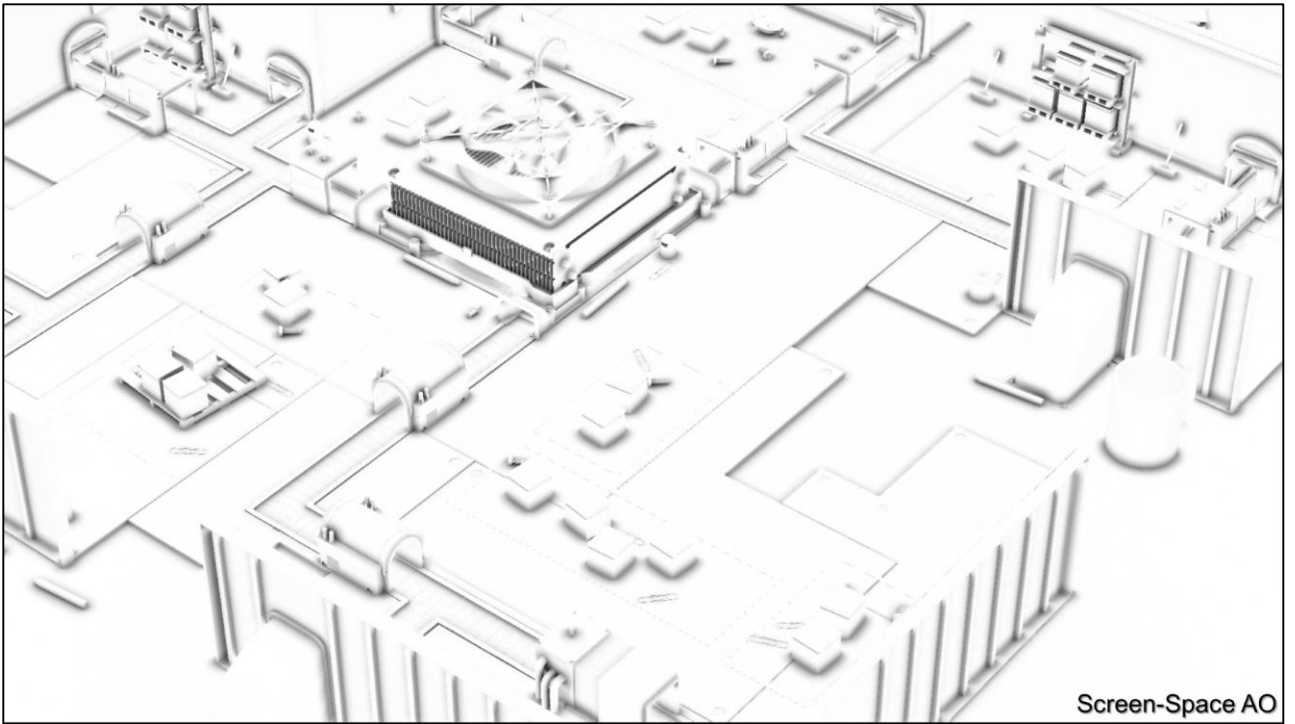


- Being the integral of the visibility function over the hemisphere, we get more grounded results because all the random directions used during sampling actually end up in the scene, unlike with screen space techniques where rays can go outside the screen or behind geometry, where the hitpoint is not visible.
- Just like in the literature, this is done by doing cosine hemispherical sampling around the normal.
- Rays are typically launched from the gbuffer, and the miss shader is used to figure out if we've hit something
- You can launch more than 1 ray per frame, but if you limit the ray distance distance even with one ray per frame you should get some nice gradients
- You'll most likely need to filter and reconstruct, as AO can be a bit noisy.

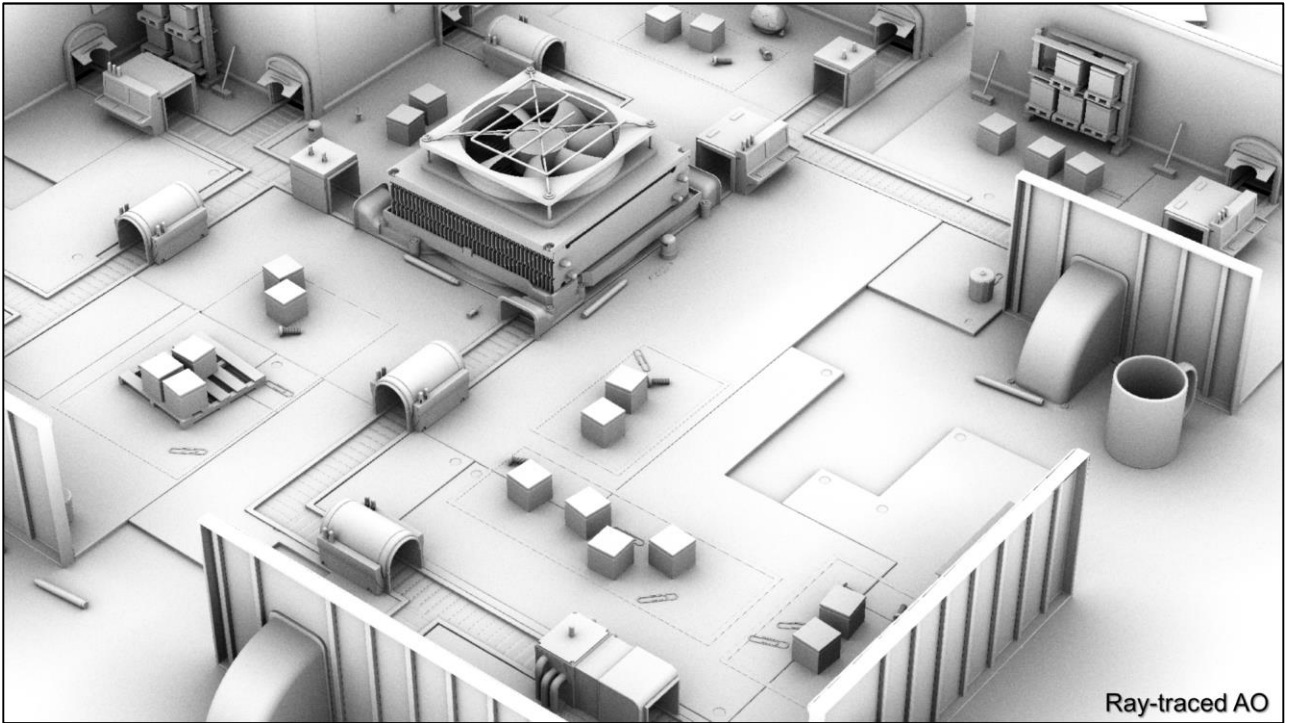
Remedy's Ray-traced Ambient Occlusion in Northlight Engine [Aalto2018]



Some really great results from the folks at Remedy, in the Northlight Engine



And if we compare with screen space AO, we can totally see that ray traced AO takes it to another level



Ray-traced AO

Overall it looks so much more grounded!



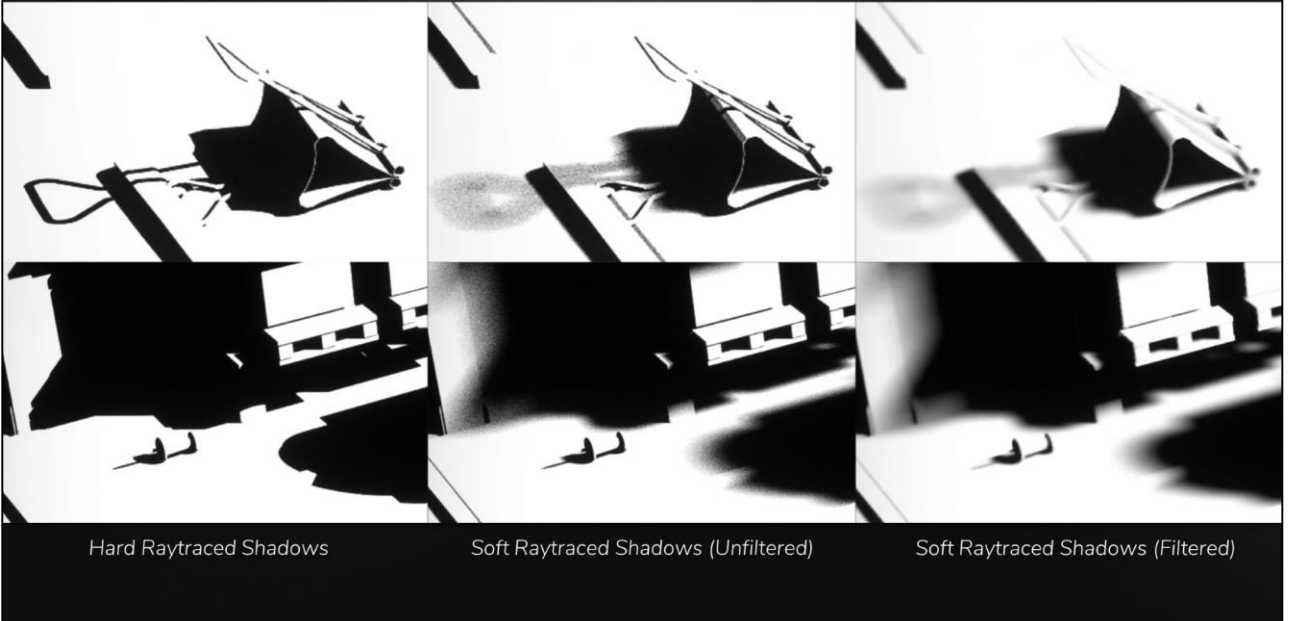
- Raytraced shadows is obviously another technique where ray tracing shines. Those are great because they perfectly ground objects in the scene.



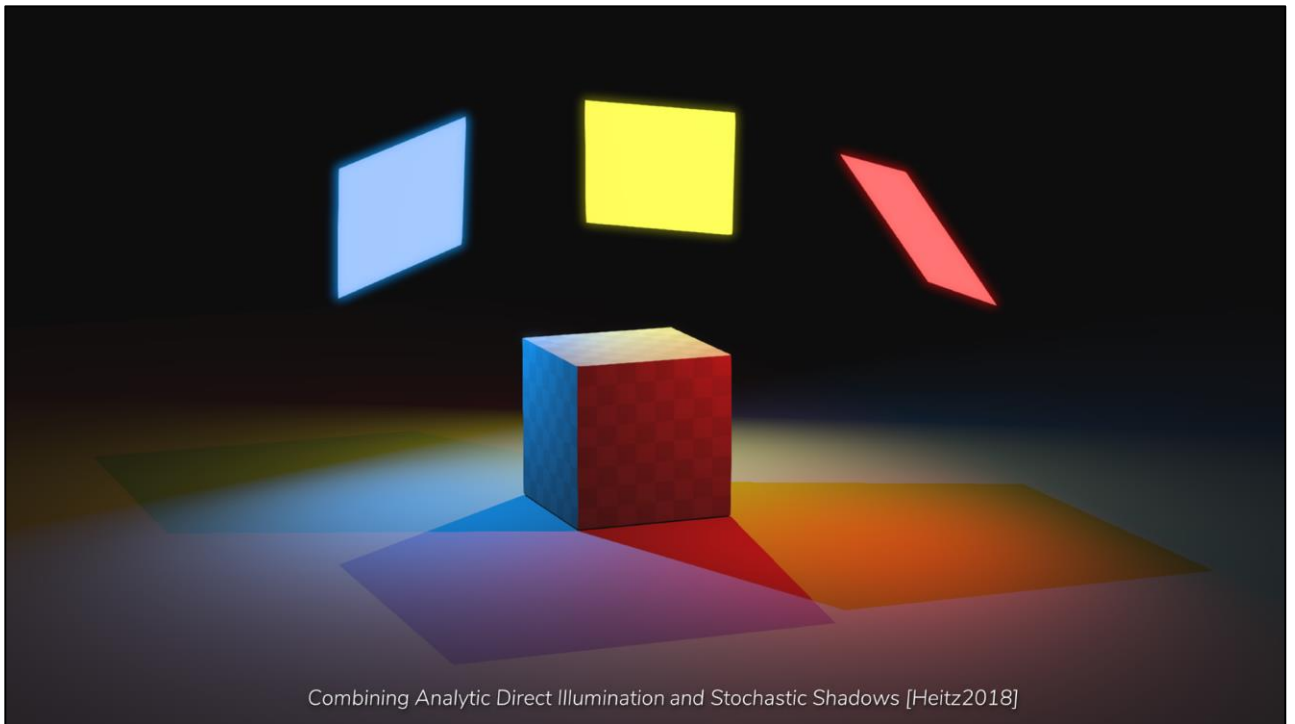
Here a scene from Unreal Engine 4 that shows how perfect shadows really help in making a visually-convincing and cohesive image.



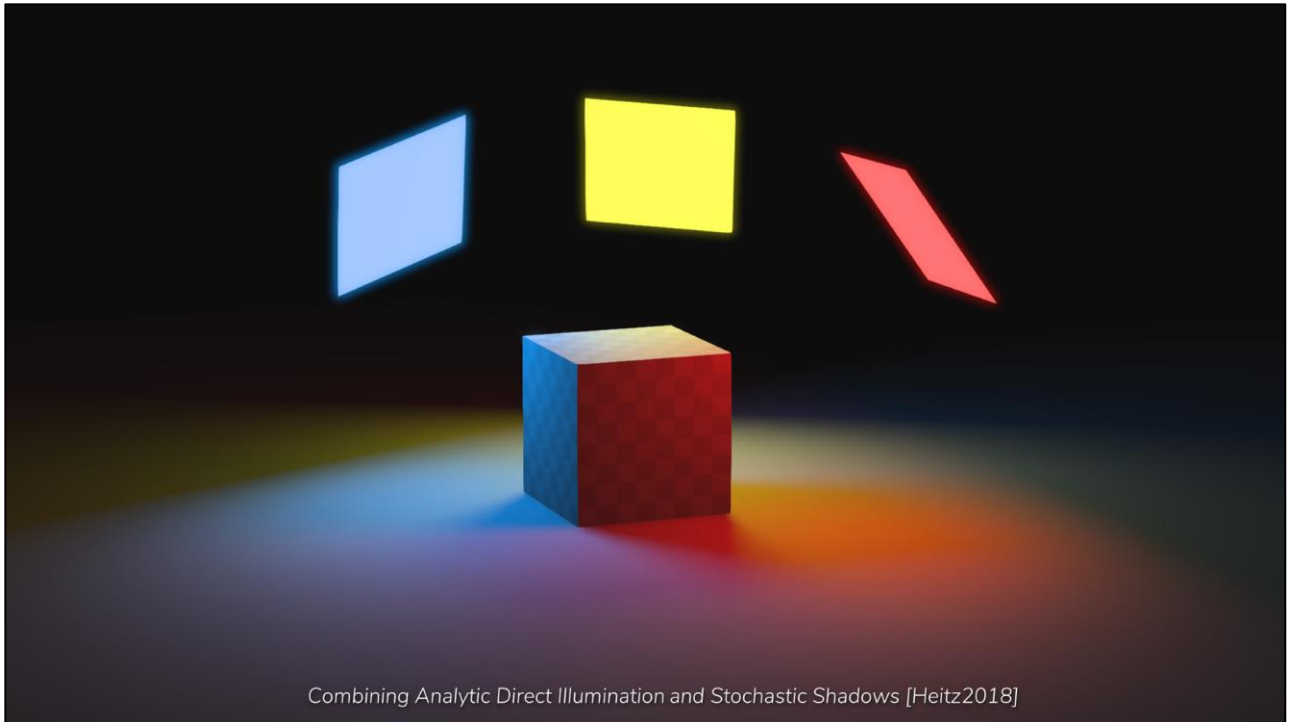
- This is not too complicated to implement. Just launch a ray towards the light, and if the ray misses you're not in shadow
- Hard shadows are great... but soft shadows are definitely better to convey scale and more representative of the real world
- This can be implemented by sampling random directions in a cone towards the light, treating it like an area light
- The wider the cone angle, the softer shadows get but the more noise you'll get, so we have to filter it
- You can launch more than one ray, but will still require some filtering



- Let's zoom on some details. We get nice contact hardening and it just works.
- But much better can be done here.



- And to this, Heitz, Hill and McGuire have demonstrated an approach that combined analytic direct illumination and stochastic shadows.
- This approach is also implemented in Unity, their soft area light shadows.
- In their paper, they propose a ratio estimator that allows correctly combining analytic illumination techniques with stochastic raytraced shadows.



- By splitting shadowed illumination in two parts -- the analytical part and the stochastic part -- their method demonstrates how one can obtain sharp and noise-free shading in the unshadowed part of the image, analytically, and visually-convincing shadows via stochastic ray tracing.
- The advantage of stochastic evaluation only where needed is that the final result only has noise in the shadows, whereas the rest is handled analytically.
- They also denoise shadows separately from illumination, so high-frequency shading details is kept. This technique is really awesome.

Transparent Shadows

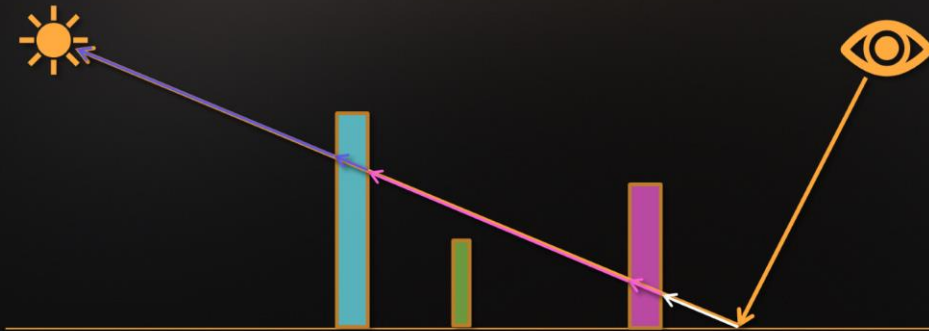
- Shadows from transparency
 - Hard to get right with raster [McGuire17]
- Let's flip it around
 - Trace towards light, like opaque
 - Accumulate absorption
 - Product of colors
 - Thin film approximation
 - Density absorption easy extension
 - Can also be soft!



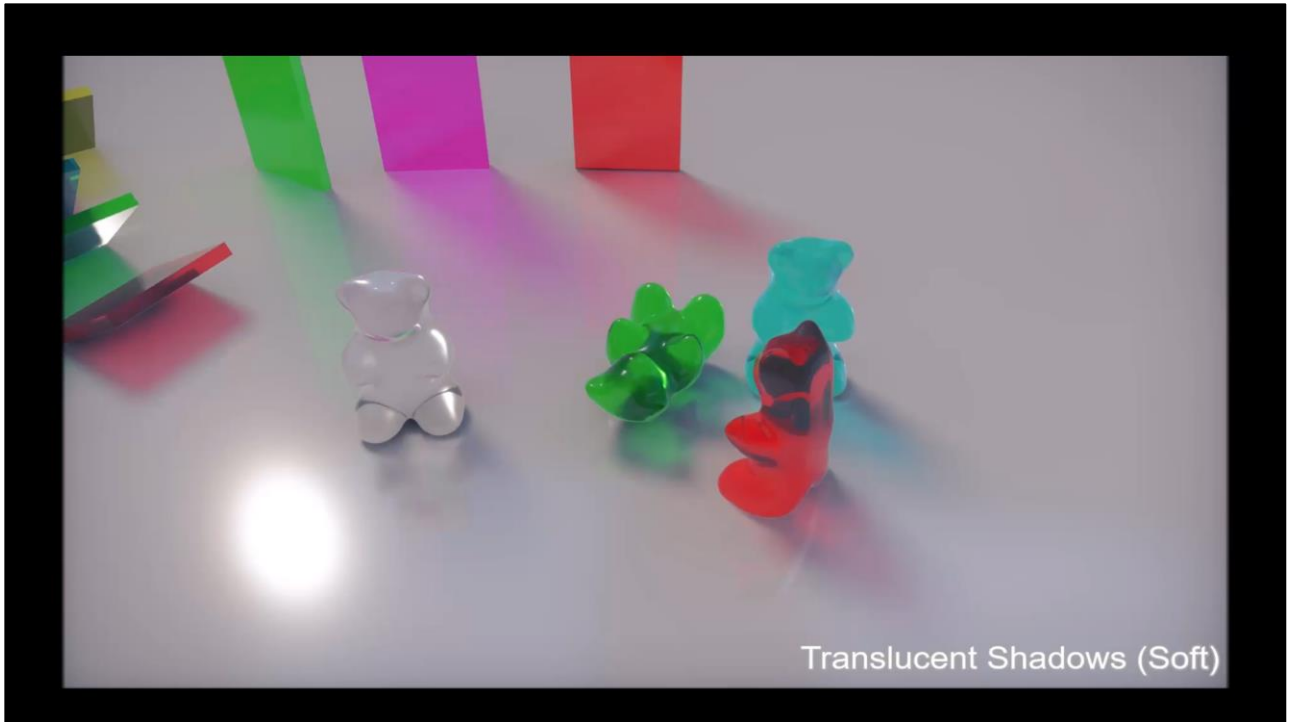
- Opaque shadows are cool, but transparent shadows for thick homogenous mediums are even better!
- Transparency is a hard problem in real-time graphics, but with ray tracing new alternatives are possible
- In our case we replace the regular shadow tracing code with a recursive ray trace through transparent surfaces
- As the light travels through the medium we accumulate absorption, multiplicatively
- Our current implementation treats this as a thin film approximation, where we assume all the color is on the surface, for performance
- Just like our opaque shadows, transparent shadows can also be soft! We filter them with a similar SVGF-inspired filter.

Transparent Shadows (1/)

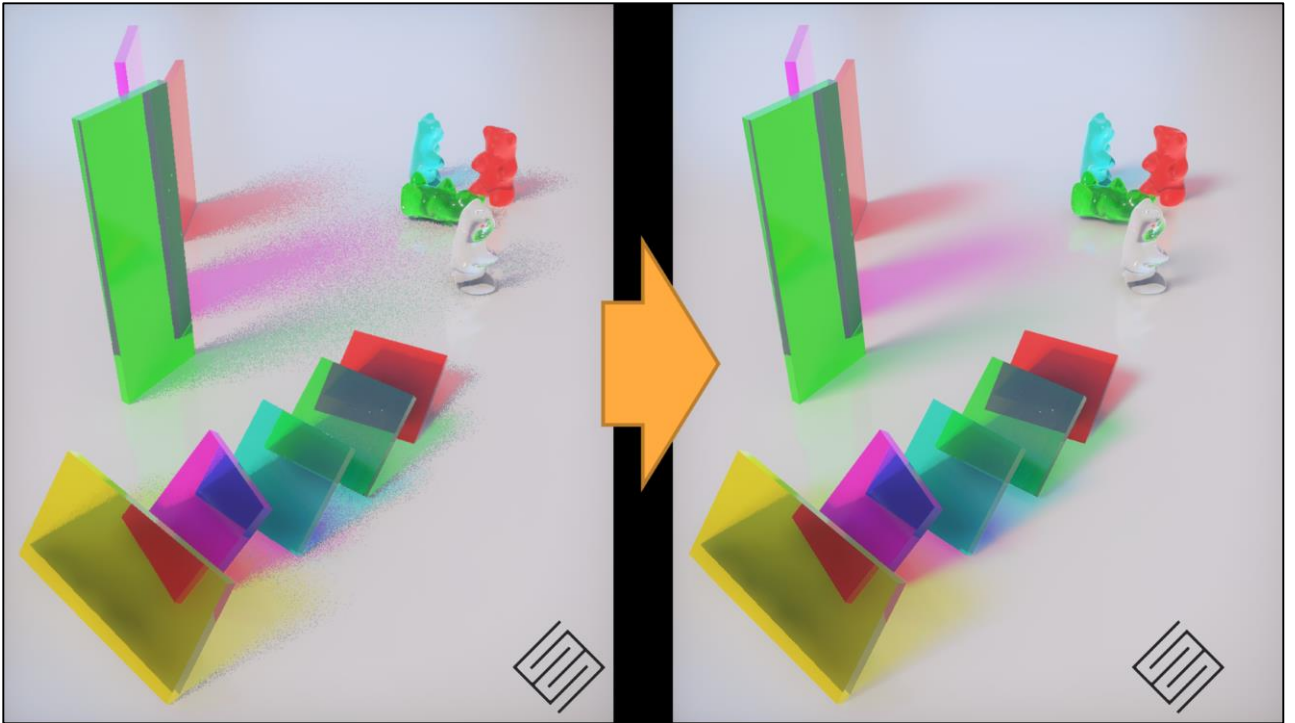
- Keep tracing until
 - Hit opaque surface or all light is absorbed or miss



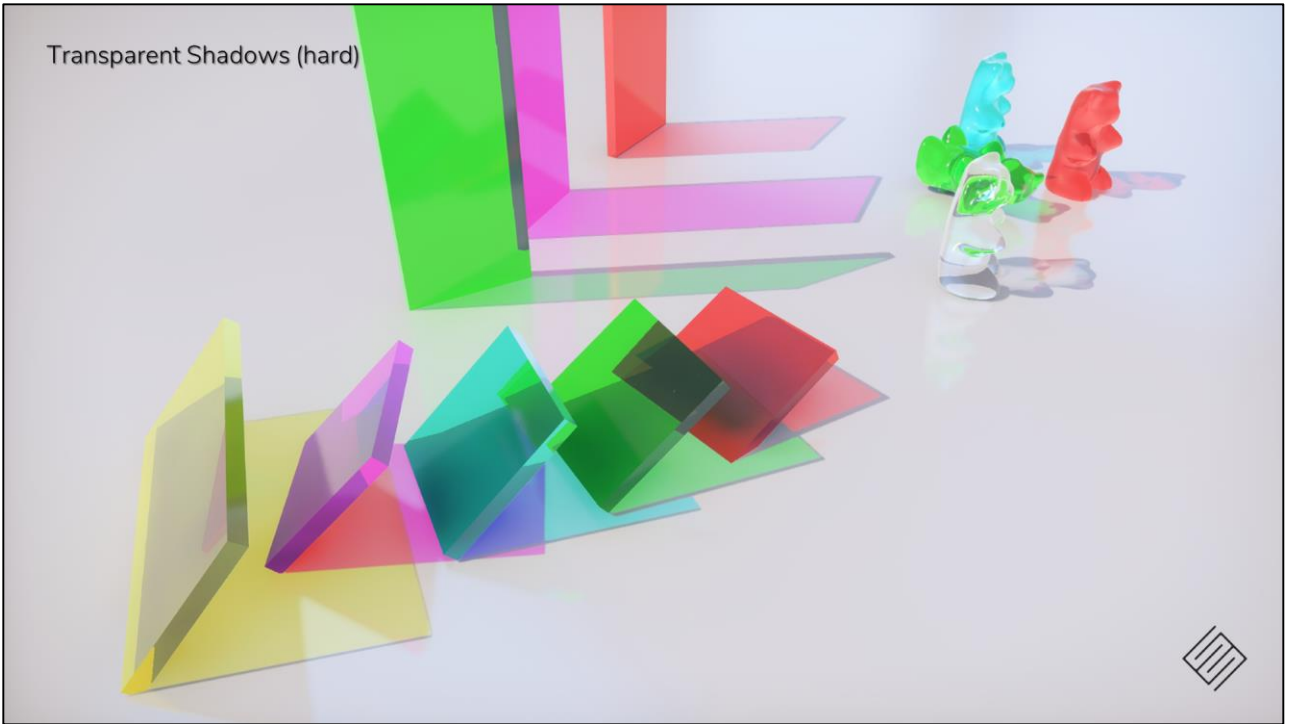
- This is a simple illustration of what happens
- For any surface that needs shadowing, we shoot a ray towards the light
- If we hit an opaque surface, or if we miss everything, we can stop
- If we hit a transparent surface however, we accumulate absorption based on the albedo of the object
- We keep doing this until 1. all light is absorbed, or 2. We miss in the trace. 2. We hit an opaque surface



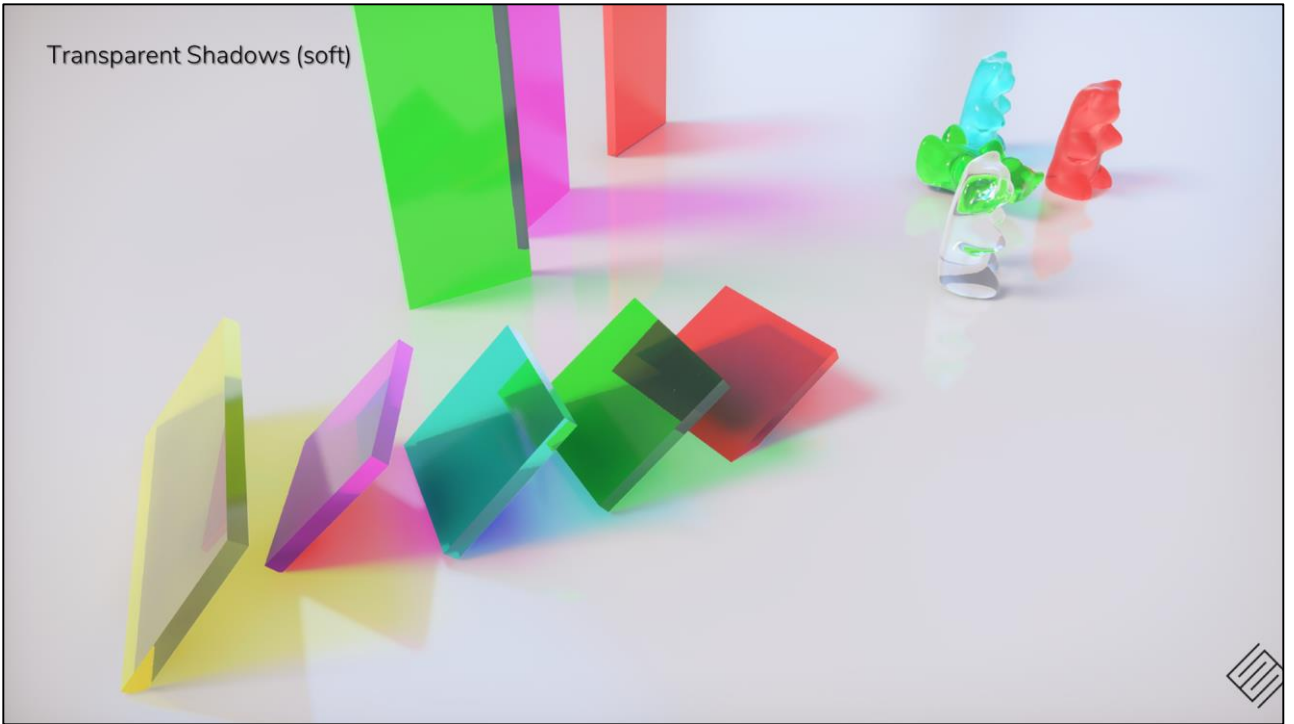
- This is not caustics, as we ignore caustic effects with our approach
- We do take Fresnel into account on the boundaries though
- It is also important to note that the regular Schlick approximation falls apart when the IOR on the incident side of the medium is higher than the far side
- We use Total-Internal-Reflection Fresnel, and filter the results with our tweaked SVGF that we mentioned in previous talks



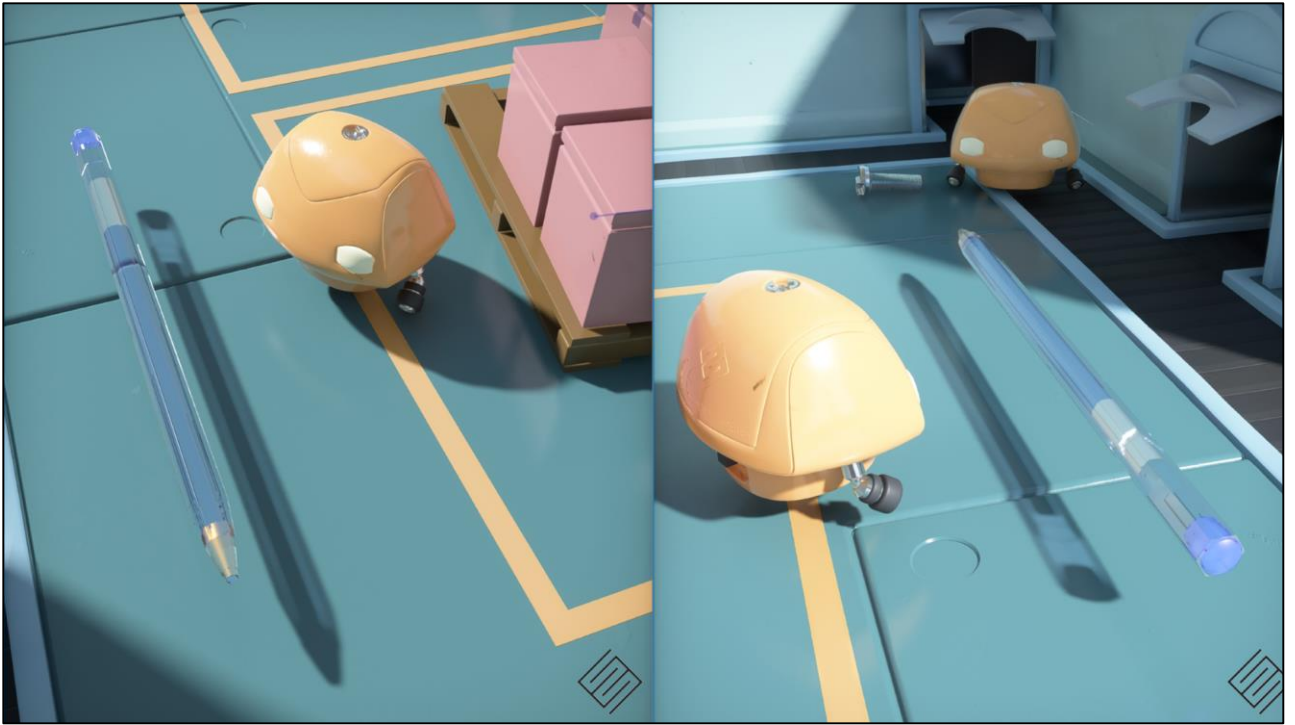
Unfiltered results on the left, and filtered results on the right.



Once again, and notice how the colors blend together as the light travels inside the glass.



Same thing here, but softly filtered.



This allows us to fix super important issues like these, where the shadow of the pen's ink tube clearly shows from the light travelling in the plastic casing.

A space scene featuring a large planet with a prominent ring system, likely Saturn, in the foreground. The planet is partially illuminated by a bright sun or star in the background, creating a lens flare effect. The background is a dark, starry space with a visible galaxy or nebula on the right side.

Transparency & Translucency

Transparency & Translucency

- Raytracing enables **accurate light scattering**
- **Transparency**
 - Order-independent (OIT)
 - Multiple index-of-refraction transitions
 - Variable roughness, refractions and absorption
- **Translucency**
 - Light scattering inside homogeneous medium
- We do this in **texture-space**
 - Handle view-dependent terms & dynamic changes to the environment

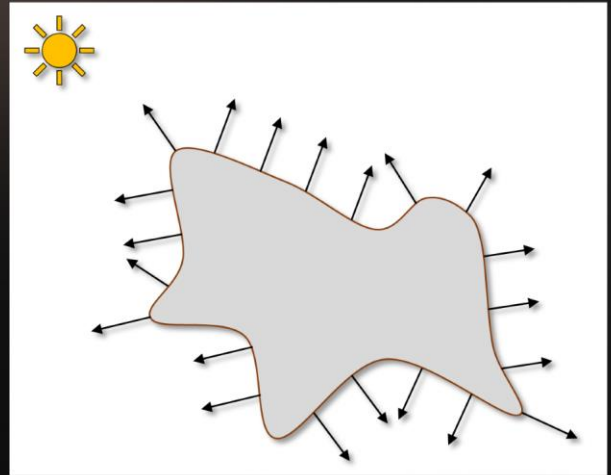
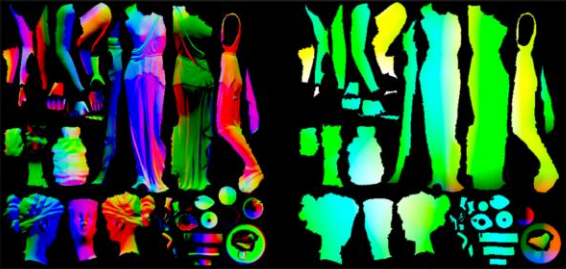


Texture-Space Glass and Translucency

- Ray tracing enables accurate light scattering for both transparency and subsurface translucency
- It's now possible to properly represent order independent transparency, variable roughness, IOR transitions as well as absorption
- For PICA PICA we did this in texture space

Translucency Breakdown

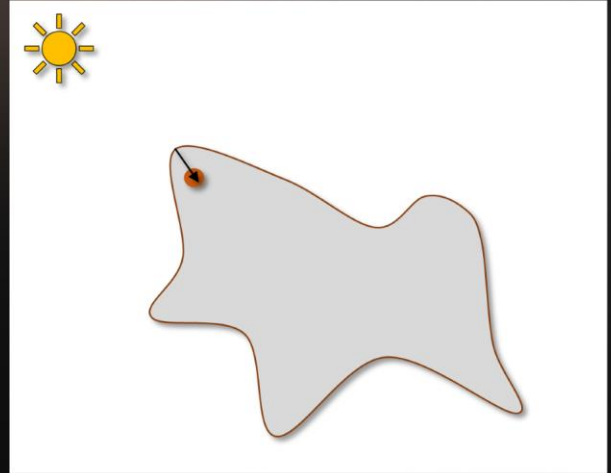
- For every valid position & normal



Here's a breakdown of how we compute translucency

Translucency Breakdown

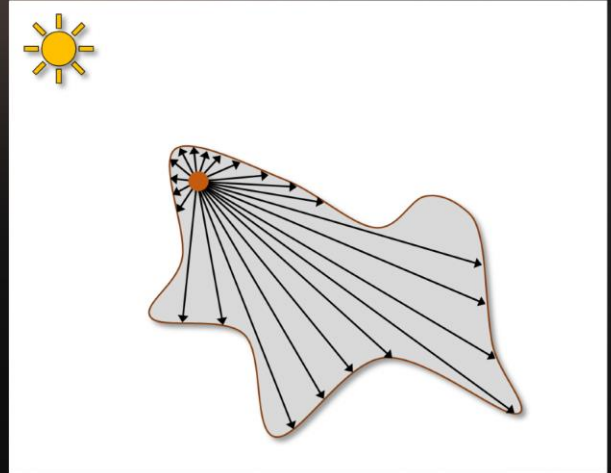
- For every valid position & normal
- Flip normal and push (ray) inside



Here's a breakdown of how we compute translucency

Translucency Breakdown

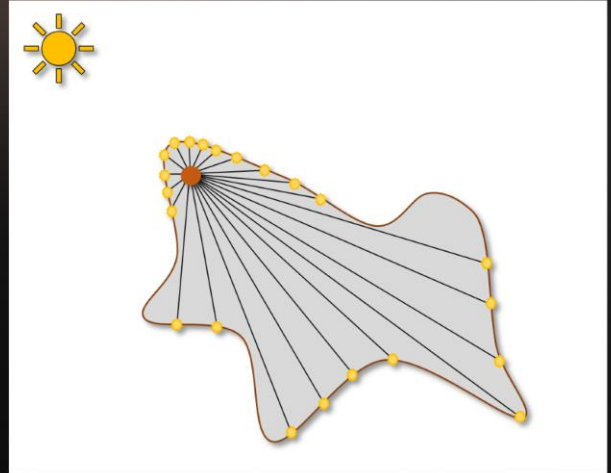
- For every valid position & normal
- Flip normal and push (ray) inside
- Launch rays in uniform sphere dist.
 - (*Importance-sample phase function*)



Here's a breakdown of how we compute translucency

Translucency Breakdown

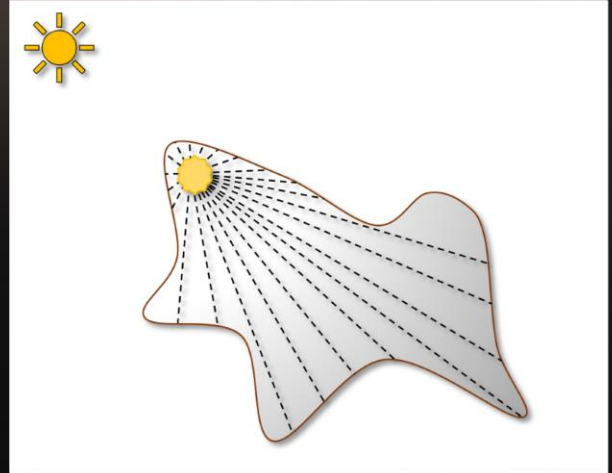
- For every valid position & normal
- Flip normal and push (ray) inside
- Launch rays in uniform sphere dist.
 - (*Importance-sample phase function*)
- Compute lighting at intersection



Here's a breakdown of how we compute translucency

Translucency Breakdown

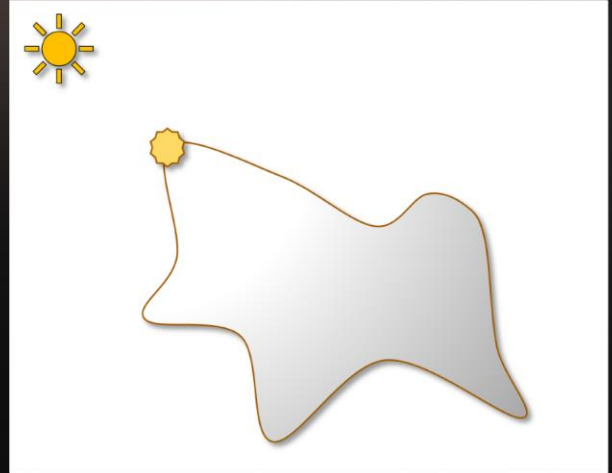
- For every valid position & normal
- Flip normal and push (ray) inside
- Launch rays in uniform sphere dist.
 - (*Importance-sample phase function*)
- Compute lighting at intersection
- Gather all samples



Here's a breakdown of how we compute translucency

Translucency Breakdown

- For every valid position & normal
- Flip normal and push (ray) inside
- Launch rays in uniform sphere dist.
 - (*Importance-sample phase function*)
- Compute lighting at intersection
- Gather all samples
- Update value in texture



Here's a breakdown of how we compute translucency

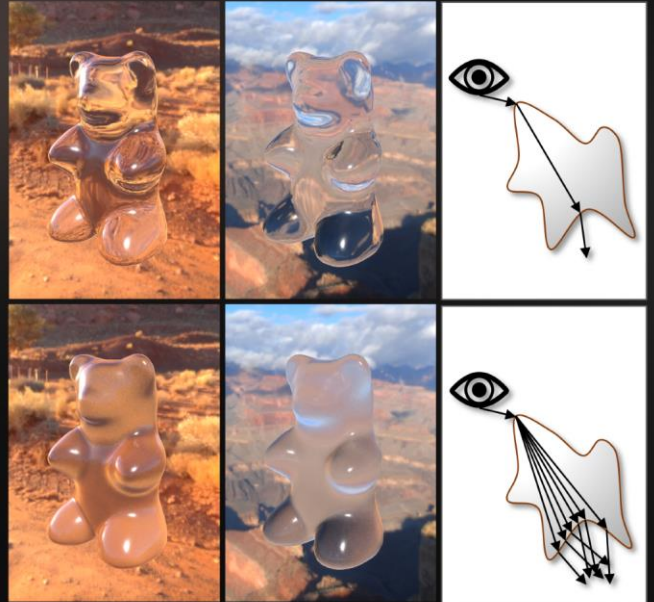


This is the kind of results we get. At GDC we talked about how we interpolate results to make them reactive, so check out the slides on our website for additional details.

Transparency

Works for clear and rough glass

- **Clear**
 - No filtering required
- **Rough**
 - *Microfacet Models for Refraction through Rough Surfaces* [Walter2007]
 - More samples + temporal filtering
- Apply phase function & BTDF



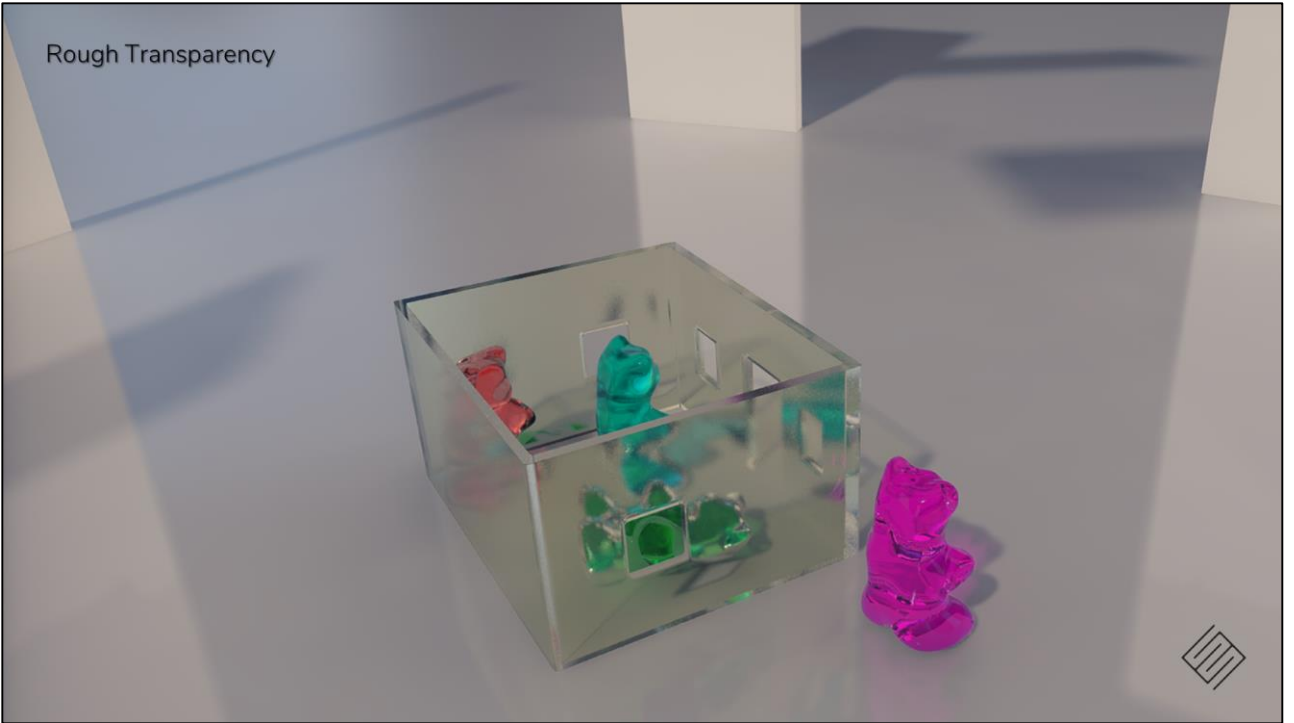
- Transparency on the other hand works both clear and rough glass
- For clear glass no filtering is required
- For rough glass, we use Walter's parametrization and importance sample GGX roughness
- For super rough, more samples are needed to get rid of noise, but one can also use temporal filtering. Easier to do in texture-space!
- The examples here show a very simple material model for glass, but something more complex can be done

Rough Transparency

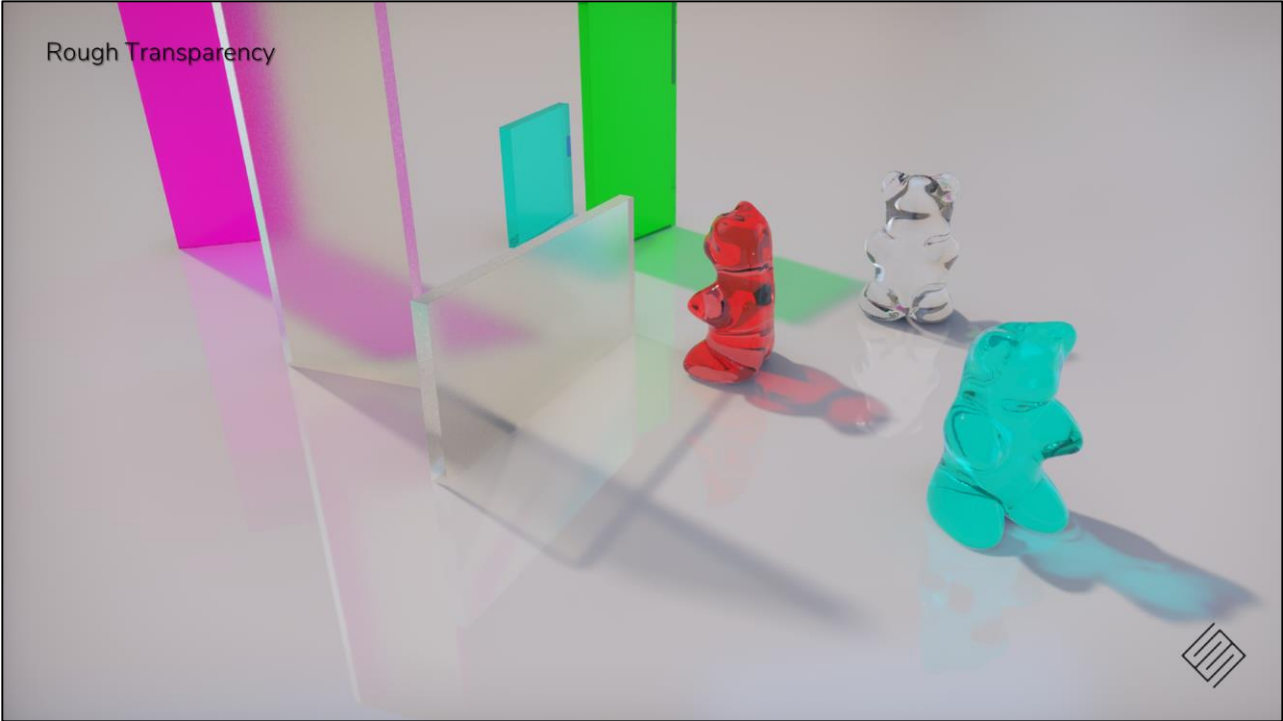


- And so transparent surfaces, you need a model that can handle refraction as well as reflection
- This is what it looks like with Walter
- Walter handles the rough light interactions on air to object boundaries, but also provides physically accurate solutions for any surface-to-surface light transport
- We importance sample the GGX distribution, using this method, for both reflected and refracted rays on internal and external boundaries
- This is obviously more expensive, but still manageable and quite nice. We feel like we can optimize it more.
- Naïve Implementation @ 512x512 in texture space is 3.5ms on Turing

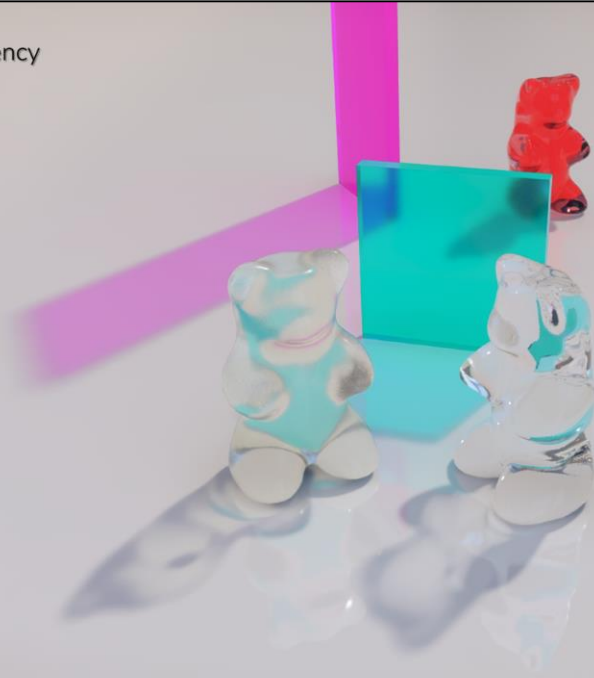
Rough Transparency



Rough Transparency



Rough Transparency





Many Lights

Many Lights

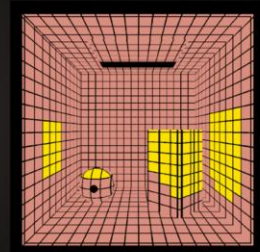
Which (most affecting) lights to choose?

Acceleration structure-based selection:

- Unity: camera-oriented acceleration structure [Benyoub 2019] [Tatarchuk 2019]
- BFV: horizontal plane light list [Deligiannis 2019]

Light Importance Sampling:

- *Dynamic Many-Light Sampling for Real-Time Ray Tracing* [Moreau 2019]
- *Stochastic Lightcuts* [Yuksel 2019]



Camera-oriented Acceleration Structure for Lights [Tatarchuk 2019] [Benyoub 2019]

PER CELL LIGHT LISTS



"It Just Works": Ray-Traced Reflections in 'Battlefield V' [Deligiannis 2019]

- A lot of the things I've just discussed handle a limited number of lights.
- While one can decide to process all the lights, sometimes that's not an option, for performance reasons
- A few approaches are possible here.
- Unity relies on a camera oriented acceleration structure, like the image on the right. We a hit is shaded, the light of lights from where that hit resides is queried, so you don't end up sampling the whole scene. Battlefield 5 relies on a horizontal plane light list.
- The first kind relies on acceleration structures to tell a pixel which lights are to be sampled. This is common for deferred and clustered shading.
- The second kind, are importance-sampling based.
- Recently Moreau, Phar and Clarbeg have released a paper at HPG a few weeks ago. Their paper describes a hierarchical light sampling data structure based on a two-level BVH, that enables interactive direct lighting from 10,000s emissive triangles. This enables a future where real-time scenes could be lit with only emissive meshes, which is really awesome. I really recommend checking it out!



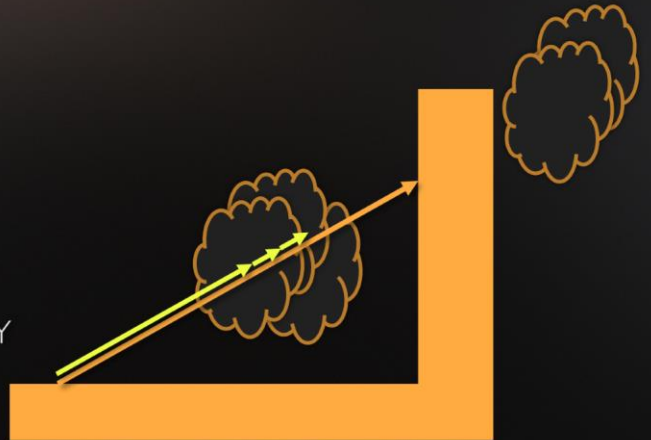
What about particles? Having VFX support for ray tracing is key to make sure explosions appear in the upside down as they appear in the scene.



The problem is that particles are built as billboards, and expected to be facing the camera. That doesn't work in reflections.

Particles

- Particles have to be ray-aligned
 - Not perfect since VFX are often designed assuming view-aligned particles
- Battlefield V [Deligiannis 2019]
 1. Shoot ray in Opaque TLAS
 2. Shoot again in Particle TLAS
 1. Limit length from opaque hit
 3. Blend particles with opaque hit
- Rotate odd particles 90 deg. around Y



[Deligiannis 2019]

- The way the folks at DICE have solved this is by orienting the particles towards the ray.
- The general scheme is to maintain two top level acceleration structures: one for opaque geometry, and one for particles
- You first shoot a ray in the opaque one, and if there's a hit you store that length
- You then launch another ray, this time in the particle acceleration structure, and limit that ray length from the opaque hit length
- You then blend particles in the scene accordingly
- Another trick that seems to have worked for them is to rotate odd particles by 90 degrees.




[Deligiannis 2019]

So again without aligning particles towards the ray



[Deligiannis 2019]

- And here's the final result.
- It's not perfect, but can work very well for your case as well. Especially for fast explosions and smoke.

A space scene featuring a large planet with a prominent ring system, likely Saturn, in the foreground. The planet is partially illuminated by a bright sun or star in the background, creating a lens flare effect. The background is a dark, star-filled space with a visible galaxy or nebula on the right side.

**Other Things
(yet still super important!)**

Ray-traced GI

Ray-traced approaches making their way:

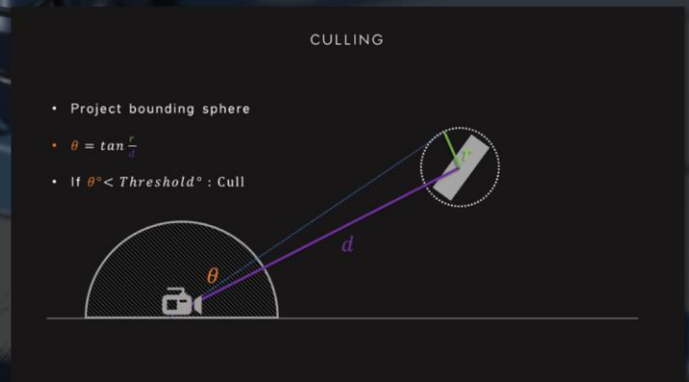
- Surfels [Stachowiak 2018]
- Grid [Aalto 2018]
- Probes [Majercik 2019]



- We've also seen a bunch of techniques benefit of the ray tracing hardware capabilities
- These techniques rely on various caching mechanisms to accumulate results over multiple frames, and accelerate sampling
- Could do a full talk on this, but that will have to be for some other day.

Culling

- Can't rely on Frustum Culling, since rays are in world space
- In the case one can't have all objects in the BVH, have to find new culling heuristic
- Projected bounding sphere [Deligiannis 2019]

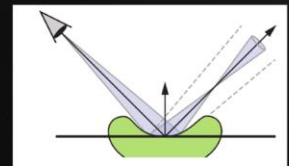
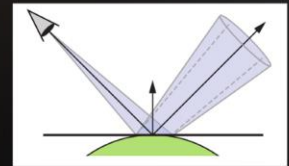
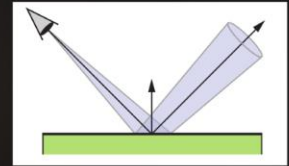


"It Just Works": Ray-Traced Reflections in 'Battlefield V' [Deligiannis 2019]

- Culling is another interesting one, since you can't rely on frustum culling anymore. For example, you can see reflections from objects behind you
- Also in the case where you can't have all your objects in the BVH, you have to find a new heuristic
- Can't just use distance, as you might want some far away buildings in reflections
- The DICE guys have relied on a projected bounding sphere, which seems to do the job

Texture LOD

- **No automatic texture LOD with ray tracing**
 - Inherent of the rasterization pipeline: pixel quad derivatives
- In Ray Tracing Gems [Akenine-Möller 2019]
 - *Ray Differentials* [Igehy 1999] vs *Ray Cones* [Amanatides 1984]
 - Heuristic based on triangle properties, a curvature estimate, distance, and incident angle
 - Similar quality to ray differentials with single trilinear lookup. Single value stored in the payload
- Barely scratched the surface – still work to do!
 - Still some needed improvements. Come get inspired at the talk! ☺
 - *Texture Level of Detail Strategies for Real-Time Ray Tracing*. In Ray Tracing Gems 1.1 Session Room: 501AB Wed @ 2PM



Ray Cones

- Texture LOD is another interesting one: there is no automatic texture mip selection with ray tracing, because pixel quad derivatives exist only for rasterization
- People have often relied on ray differentials, but it has some performance implications
- In ray tracing gems, we discuss an alternative technique based on cones.
- I'll talk about this on Wednesday at 2PM
- But you'll see that it's not perfect, and we still have some things to improve

Performance Good Practices

- Minimize recursion: favor fire-and-forget / tail-recursive techniques
- Consider manual scheduling (sorting/binning) [Aalto 2018] [Deligiannis 2019]
- Minimize/pack payload and attributes
- Optimize your *any-hit* shaders (or don't use them at all)
- BLAS build & update on an async queue: define update vs rebuild metric for your case, over multiple frames, overlapped with raster
- TLAS: Build instead of Update & don't include the skybox (do in Miss Shader)
- More [Tips & Tricks: Ray Tracing Best Practices](#) [Dunn 2019]

- A bunch of performance good practices have emerged from the initial group of products that have adopted real-time ray tracing.
- Some are obvious, and some are based on limitations of the current hardware and API implementations
- A lot of it is case by case for your game, but the following should be generally good advice for now
- Especially minimizing recursion and adopting fire-and-forget tail-recursive techniques
- And building your own metric for BVH update vs refit
- There's this great article by Alex Dunn from NVIDIA, and I really recommend checking it out.



Let's briefly touch on some open challenges for real-time ray tracing and video games, where you could possibly have an impact as a researcher.

Game Constraints



Complex & Large Scale Environments in EA BioWare's Anthem



Need robust techniques

for video games:

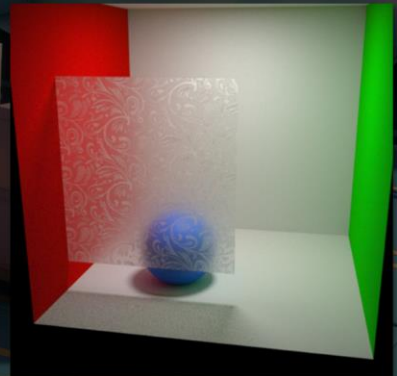
- Many animated characters
- Dynamic environments
- Moving foliage & vegetation
- Massive open worlds
- User generated content
- User created experiences

- In the context of a game we need to support many animated characters, lots of foliage, potentially in a massive open world that evolves
- Also possibly some user generated content and created experiences that you might not be able to process on the fly, without any art clean-up
- Techniques have to be robust, and constraints well exposed in publications. This last one really really helps!



Transparency

- **Transparency** is far from being solved!
 - Glass
 - Clear/rough + filtered + shadowed
 - Particles & Volumetric Effects
 - Can use miss shader to update volumes / clipmaps
 - Ray marching in hit shaders?
 - Non-trivial blending & filtering
- PICA PICA: **texture-space OIT** with refractions and scattering
 - Not perfect, but one step closer
- Denoising techniques don't work so well with transparency (and 1-2 spp)



Transparency in the Maxwell Renderer

- Transparency. Even with real-time raytracing, this is definitely not solved.
- When one looks at the images on the right... we still have work to do to reach that quality level in real-time, at 1 sample per pixel
- Actually a lot of transparent effects require non-trivial blending with the rest of the scene, and non-trivial filtering.
- It's also a challenge when it comes to blending volumetrics, particles and fog with other effects....
- For transparency we came up with a texture-space OIT technique, but you have to deal with temporal issues.
- The thing is, with 1 sample per pixel, and throwing some Monte Carlo in there, most denoising techniques generally don't work so well with transparency or partial coverage



Partial Coverage

- **Foliage**
 - Can still do alpha test (i.e.: any-hit)
 - Animated becomes a real problem
- **Defocus effects** such as motion blur and depth of field are still intractable
- Need partial coverage denoising @ 1-2 spp



PBRT (Matt Pharr, Wenzel Jakob, Greg Humphreys)



© Disney/Pixar

Disney / Pixar

- Same thing if we talk about partial coverage
- We can still do alpha testing in the hit shaders, and could use some pre-filtering. But as soon as it starts moving, it can become a problem both for performance and visually.
- If you're building a jungle or a scene like PBRT, or the forest on the island from Moana, it's a different story than just a tree here and there.
- Some folks have even considered not using any-hit shaders, and modeled leaves with triangles. Can work too depending on your case.
- Other types of partial coverage effects that get affected such as DOF and motion blur also fall into this trap
- And so, current denoising techniques don't work well with this kind of partial visibility in real-time. And often this is because we only have 1 sample per pixel and assume everything is opaque.

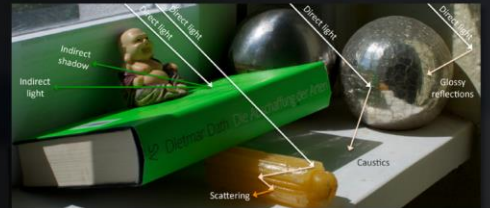
Future Ray Tracing Research

- Literature has to **adjust to real-time game ray tracing constraints**
 - Games → budgeted amount rays/frame, rays/pixel, fixed frame times & memory budgets
 - Games → Light transport caches: surfels, voxels, lightmaps
- Moving towards decoupled shading & variable rate ray tracing
 - Texture-space Techniques
 - Caching of materials and partial solutions
 - Split the BRDF: view vs view-independent terms
- Perf R&D: Efficient sampling / integration strategies & reconstruction filtering

- In terms of future ray tracing research, if we talk about literature, literature has to adjust itself to real-time constraints, and not just “correct raytracing”.
- The metrics listed here become important for papers to get adopted by games
- Ray tracing also funnels exploration of texture space techniques, and variable rate ray tracing
- We still have a bunch of work to do on sampling and integration strategies, and reconstruction

Global Illumination

- **Open problem even in offline rendering**
 - Variance still too high
 - Can reduce frequency
 - Prefiltering, path-space filtering
 - Denoising & reconstruction
 - Pinhole GI & lighting is not solved
- **Incoherent** shading → **intractable** performance
 - Have to resort to caching to amortize shading
 - PICA PICA: caching of GI via surfels
 - Issues: only spawn surfels from what you see
- Need to solve GI for **user-generated content** →



[Ritchell 2011]



- GI is another good one: Real-time raytracing doesn't completely solve real-time GI
- It's still a problem for offline, where scenes can take hours to resolve, with difficult paths with caustics for example.
- There are many workarounds in offline, but they don't necessarily map to real-time.
- For real-time we absolutely have to resort to caching, like techniques I mentioned a few slides ago. Some techniques work better than others, especially if artists don't have to do manual UV unwraps for lightmaps or proxy geometry for GI.
- We also need to solve GI for user-generated content, where you can't expect any upfront parametrization
- So even with TRRT, we're definitely not done here

Sparse BVH Tracing

- **We assume ray-triangle intersection as the end-all-be-all, but what if we stopped the ray higher up the tree?**
 - Treat AABBs like voxels
 - Akin to beam tracing [Heckbert 1984], or “ray bundles”
 - Explore algorithms that would benefit from broad tracing
 - *Global Illumination Using Ray-Bundle Tracing* [Tokuyoshi 2012]
 - *Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields* [Majercik 2019]
 - Cone Tracing [Crassin 2011]
 - Sound Propagation
 - Acceleration is currently ray-tri & ray-AABB
 - Feels like it’s all here.
 - Expose trace “LOD” controls to developers at API level

- The current model for real-time ray tracing also assumes that ray-triangle intersection is the end-all-be-all, but what if we stopped the ray higher up the tree
- This could enable new types of tracing, like beam tracing or ray bundles
- Could unlock a family of algorithms that require broad tracing, in real-time
- Right now, it feels like it’s all there, and one just needs to expose this as a LOD control for developers.

Evolving Engines for Hybrid RT

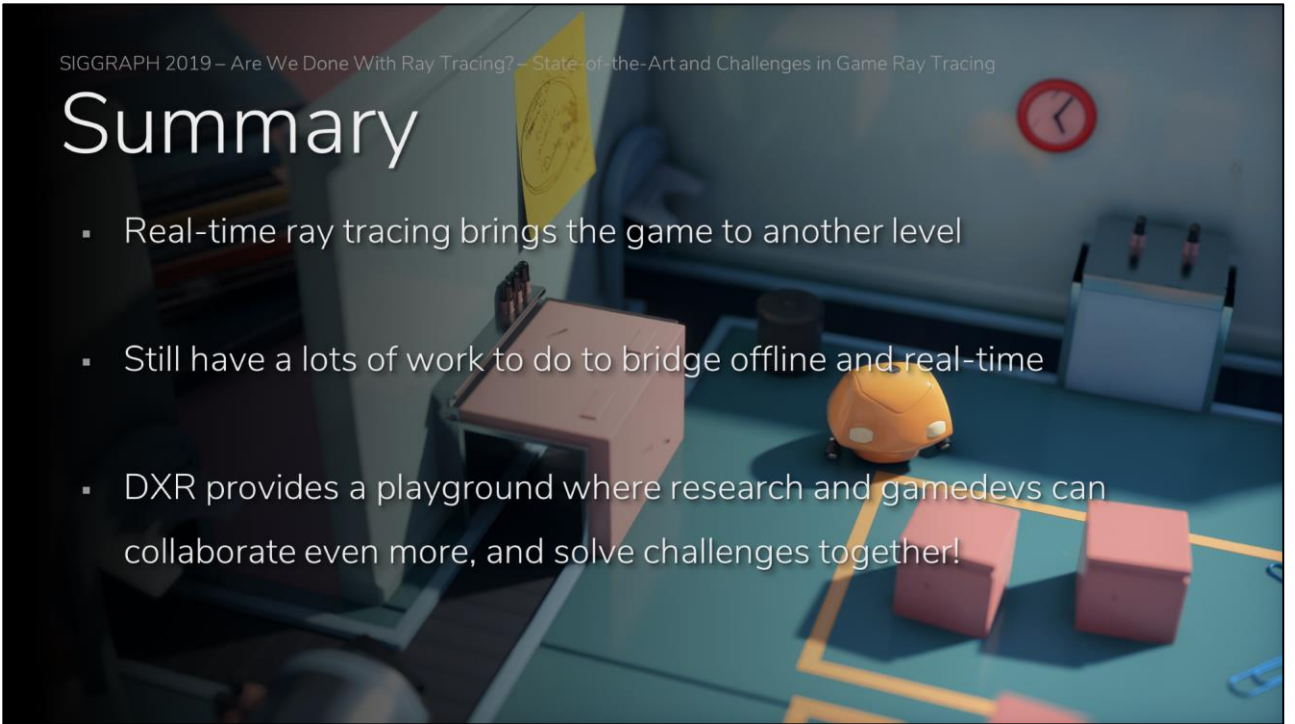
- Bindless
- Visibility-Buffer
- Texture-space Ray Tracing
- Raster still shows some advantages for primary opaque visibility over RT
- ...
- Scene Management for RT
- Shader Compilation [Deligiannis 2019]



- Right now it might be a bit duct-tapey for the first round of products, but we also have a bunch of work on the engine side to evolve our engines to maximize this hybrid pipeline idea
- Runtime is important, but also pipeline related challenges around the massive number of shader permutation engines generate

Summary

- Real-time ray tracing brings the game to another level
- Still have a lots of work to do to bridge offline and real-time
- DXR provides a playground where research and game devs can collaborate even more, and solve challenges together!

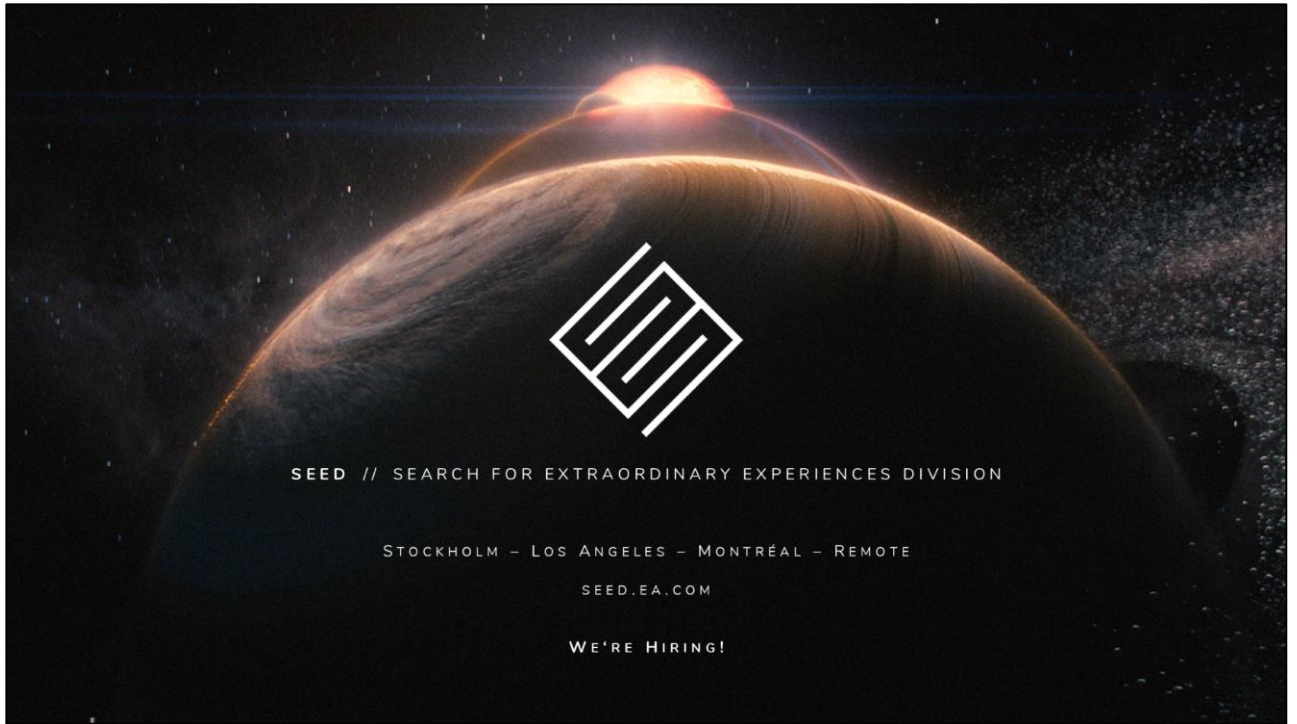


Thank You!

- SEED
- Alex Keller (NVIDIA)
- Natasha Tatarchuk (Unity Technologies)
- Marcus Wassmer (Epic Games)
- Jan Schmid and Johannes Deligiannis (DICE)
- Jon Greenberg (for BVH & beam tracing discussions)

References

- [Aalto 2018] Aalto, Tatu. "Experiments with DirectX Raytracing in Remedy's Northlight Engine", [online](#).
- [Amanatides 1984] Amanatides, John. "Ray Tracing with Cones", [online](#).
- [Andersson & Barré-Brisebois 2018] Andersson, Johan and Barré-Brisebois, Colin. "Shiny Pixels and Beyond: Real-Time Ray Tracing at SEED", [online](#).
- [Benyoub 2019] Benyoub, Anis. "Leveraging Ray Tracing Hardware Acceleration In Unity", [online](#).
- [Epic, NVIDIA and ILMxLAB 2019] "Epic Games demonstrates real-time ray tracing in Unreal Engine 4 with ILMxLAB and NVIDIA" [online](#).
- [Heckbert 1984] Heckbert, Paul and Hanrahan, Pat. *Beam Tracing Polygonal Objects*", [online](#).
- [Igehy 1999] Igehy, Homan. "Tracing Ray Differentials", [online](#).
- [Majercik 2019] Majercik, Alexander. Guertin, Jean-Philippe. Nowrouzezahrai, Derek. McGuire, Morgan. "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields", [online](#).
- [Macedo 2016] de Macedo, Daniel Valente. Rodrigues, Maria Andréia F. "Real-time dynamic reflections for realistic rendering of 3D scenes", [online](#).
- [Ritschel 2011] Ritschel, Tobias et. al. "The State of the Art in Interactive Global Illumination", [online](#).
- [Schied 2017] Schied, Christoph et. al. "Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination", [online](#).
- [Stachowiak 2018] Stachowiak, Tomasz. "Stochastic All The Things: Ray Tracing in Hybrid Real-Time Rendering", [online](#).
- [Tokuyoshi 2012] Tokuyoshi, Yusuke and Sekine, Takashi. "Global Illumination Using Ray-Bundle Tracing", [online](#).
- [Tatarchuk 2019] Tatarchuk, Natalya. "Towards Filmic Quality at 30 FPS: Real-Time Ray Tracing for Practical Game Engine Pipelines", [online](#).
- [Yuksel 2019] Yuksel, Cem. "Stochastic Lightcuts", [online](#).



On one last note, we would like to point out that we're hiring for multiple positions at SEED. If you're interested, please give us a shout!