



Trace All The Rays!

State-of-the-Art and Challenges in Game Ray Tracing

Colin Barré-Brisebois
Head of Technology, SEED, Electronic Arts
@ZigguratVertigo



Good morning, good afternoon, and good evening, to wherever you are in the World. I hope everyone here is doing great and hope you and your families are safe.

My name is Colin Barré-Brisebois and I'm the Head of Technology at SEED, Electronic Arts.

In this keynote today I will chat about the dream that many have had for so many years where « ray tracing is the future and will ever be », with a focus on video games.

I will discuss the state of the art and various challenges we're facing. Hopefully my talk will give you sense of where we are at in the games industry with some of the challenges we're facing, but also inspire you for your future research in case you want it to make its way into video games and other real-time interactive mediums.

Before I begin I would like to thank Ulf and Ari for inviting me, and I would also like to thank Peter-Pike and Paul for their great keynotes.

Agenda

- SEED's Research Overview
- State of The Union: *real-time ray tracing in games*
- The Road Ahead: *considerations for real-time ray tracing in games*
- Open Problems

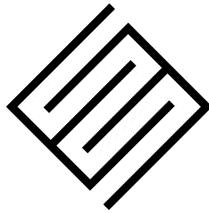
Here's the agenda for today's talk

First I will give a quick overview of SEED's research

I will then spend a few minutes looking at where we're at, today, with real-time ray tracing in games

I will then talk about the road ahead, and what are some things to consider when tailoring ray tracing research for making it applicable to games

Finally, I will talk about some of the open problems that remain. Hopefully this should spawn some ideas in your head that you can bring up during the Q&A, right after the talk.




SEED

So let me tell you a bit about SEED.
SEED stands for the Search for Extraordinary Experiences Division



SEED

Search for Extraordinary Experiences Division

- Applied R&D team at Electronic Arts
 - Est. 2015
 - ≈30 distributed across 6 offices:  Stockholm, Montréal, Los Angeles, Redwood Shores, London, Vancouver
 - seed.ea.com
 - @SEED
- Research
 - Cross-disciplinary applied research: art + engineering + creative + research
 - Run fast towards the future for the benefit of our games and studios
 - 6Mo-5Y+ applied R&D: short, medium, and long-term research portfolio
- Collaborate
 - Games Teams (DICE, BioWare, Madden, Motive, Respawn)
 - Central Groups (Frostbite, EA Create, EA Digital Platform)
 - External (Hardware, Software, Standards, Academia)
- Share
 - Present & Publish: 50+ publications & presentations
 - Open Source



In case you don't know who SEED is, we're a technical and creative research division of Electronic Arts Studios.

Established in 2015 our team of 30 is distributed in 6 locations across the world including Stockholm, Montreal, Los Angeles, Redwood Shores, London and Vancouver

You can find more info about us at seed.ea.com, or follow us on twitter
SEED exists at EA as a cross-disciplinary team where we combine art, engineering, creativity, and research to deliver disruptive innovation, for our games and our players.

We run fast towards the future and run in parallel to the current business constraints,

For the benefit of our games and all EA studios

By focusing on short, medium, and long term research portfolio gives us an opportunity to do research and always be delivering technology artifacts along the way

We collaborate with game teams, central groups, as well as many external partners in hardware, software, industry standards bodies, and academia.

Of course as an R&D group we have to present and publish. Over the years we've accumulated more than 50 publications and presentations

We also open source some of our work, so find us on github.

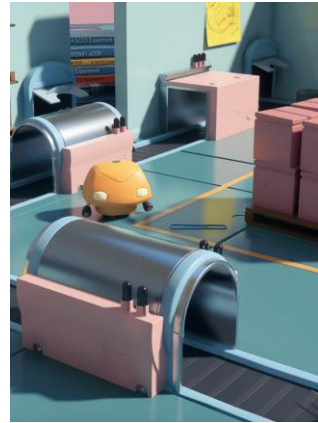
Research Vectors



Advanced Avatars



Deep Testing



Future Graphics

At SEED our research is split in 3 vectors

Advanced Avatars, Deep Testing, and Future Graphics

I will give you a quick overview of each vector so that you get a better idea about the topics we're focused on

Advanced Avatars

Emotionally-convincing Characters with Revolutionary Pipelines

- Transform performance capture at EA with machine learning and computer vision
- Artist-friendly, computer-assisted content creation workflows at scale & quality
- Visually-convincing character creation tools for accelerating work



Advanced Avatar, the first vector I want to bring up, is based around building a revolutionary pipeline for creating emotionally-convincing characters. At EA and SEED we are focused on transforming how we do performance capture for our games. We use machine learning and compute vision to build artist-friendly assisted workflows that enable content creation at scale and quality. This means character creation technology for accelerating the work of our art experts, so that they can do their best work, and iterate as fast as possible to really focus on the art.

Deep Testing

Human-Like Behavior Simulation for Game Testing

- Use ML to drive improvements to how we test games at scale
- Tools to help our testers find bugs faster, and improve overall product quality
- Human-like behavior that goes beyond current testing approaches



Our second vector, Deep Testing, is a direct application of Machine Learning for improving how we test games.

It is well known that games are getting bigger and bigger, and that with games running as a live-service, that gamers want more content, delivered faster. And this is where Machine Learning comes in and enables us to improve how we test these games at scale and speed, so that we can deliver better content faster and hopefully bug free.

Instead of taking a bruteforce approach of just throwing people at the problem, we use machine learning to build tools that help our tester's with tedious tasks, which means an overall improvement of quality-of-life for our developers, where they can focus on more important and fundamental testing.

Also, with neural networks learning how to play the games by themselves we can mimic human-like behaviors that go beyond current automated testing approaches, of scripted bots, where the game learns and plays like a human.

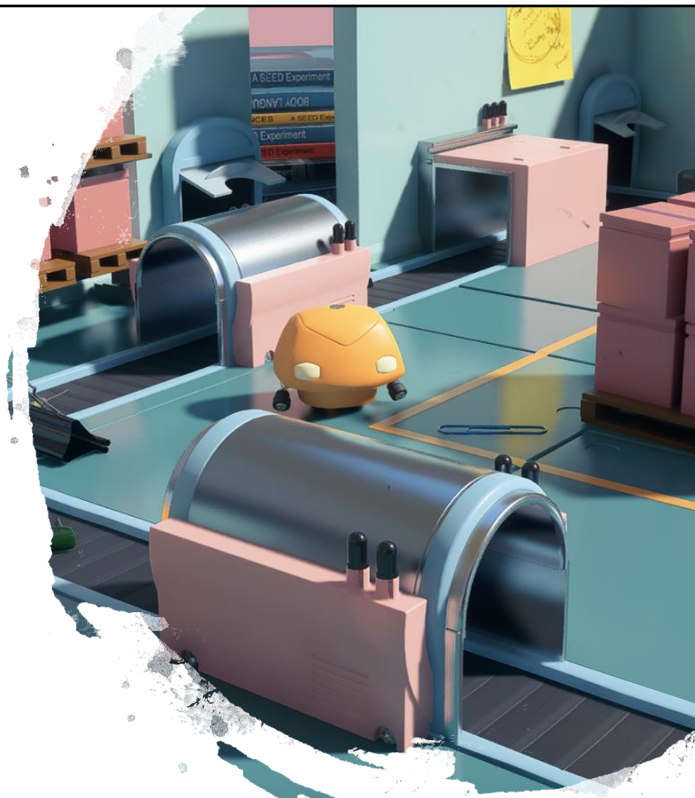
This means we find bugs faster, in ways that humans do, before they get out in the wild to millions of players.



Future Graphics

Breakthrough Visuals & Novel Content
Creation Techniques

- Push the boundaries of real-time visuals & simulation
- Technology to build & play in worlds that we could only dream of before
- Explore artist-friendly and pain-free paradigm shifts in how we build content



And finally, our last research vector, Future Graphics, is built around building novel content creation techniques and breakthrough visuals.

Real-time ray tracing comes to mind, but also other things like real-time global illumination, realistic physics simulations at a fraction of the cost, and new ways of authoring content. More on this at upcoming conferences.

Here we try to push the boundaries of real-time visuals and simulation, to enable building and playing in worlds that we could only dream before

As programmers we tend to want to optimize, which means sometimes coming up with approaches that are not always friendly for content creators. Having to build LODs for example, or having to unwrap UVs is one of those tasks that comes to mind. Here we explore radical shifts in how we build content, with a focus on making these new approaches artist-friendly and pain-free.

We also explore how these techniques can benefit some of the latest hardware, with next-generation consoles, well I guess current generation consoles now, but also future hardware architectures.

State of The Union

Let's talk about real-time ray tracing in games

Now that you know a bit more about SEED and what we do at EA, let's switch gears and talk about real-time ray tracing in games.

State of the Union

Announcing Microsoft DirectX Raytracing!



D3D Team
March 19th, 2018

If you just want to see what DirectX Raytracing can do for gaming, check out the videos from [Epic Futuremark](#) and [EA SEED](#). To learn about the magic behind the curtain, keep reading.

- 2018 – *DirectX Raytracing (DXR)* announced
- 2018 – Ray tracing hardware announced
- 2018 – Initial round of RT-powered PC titles
- 2019 – DXR v1.1 spec announced
- 2020 – Consoles ray tracing hardware announced
- 2020 – *Vulkan Ray Tracing API* spec finalized



Looking back, this journey of bringing real-time ray tracing to life has been quite awesome

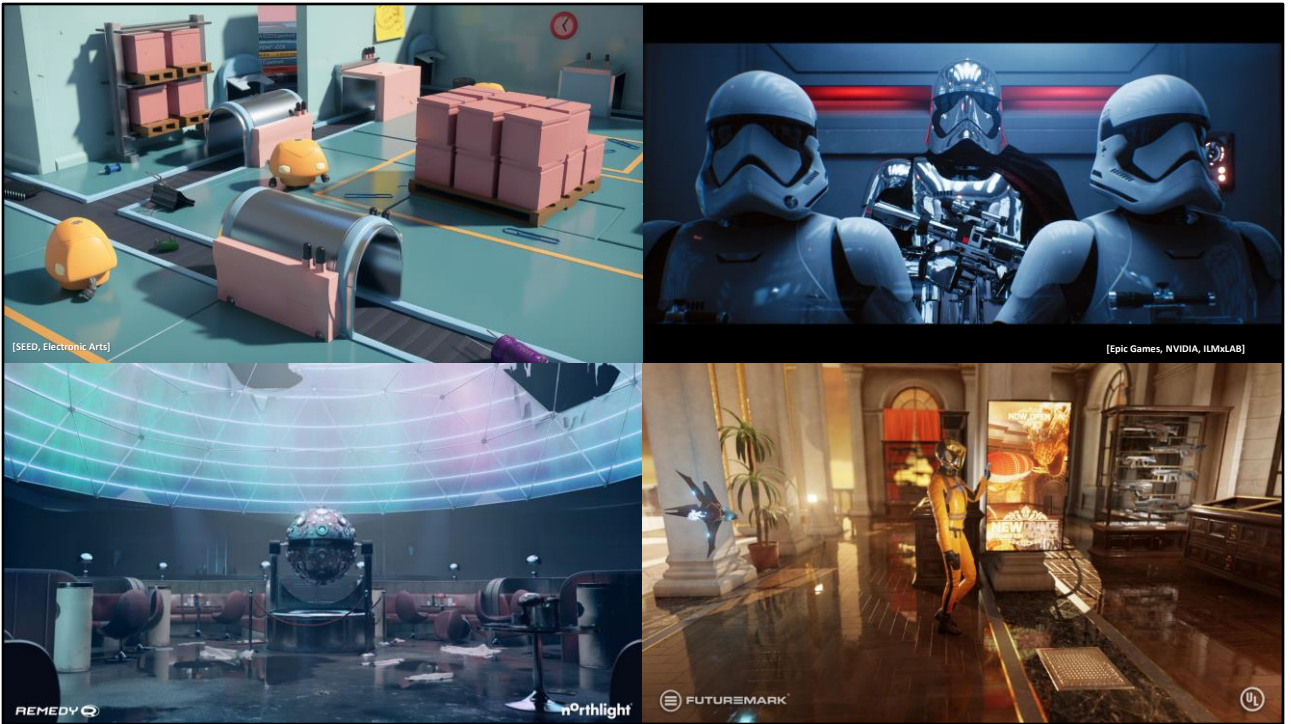
From the initial announcement of DirectX Ray Tracing back in 2018, where we partnered Microsoft, NVIDIA, to bridge the API, the GPU with the software, and also the great work from our friends at Epic, Future Mark and Remedy

To hardware acceleration being announced and an initial round of PC blockbusters supporting ray tracing like Battlefield 5, this was a good start to 2018!

Then in 2019 people had more time to play around with the API, and it evolved it with all the conversations with developers, which led to the version 1.1 of the DXR spec

Then, in 2020 a new generation of consoles got announced, with ray tracing support, which helped solidify and reinforce that real-time ray tracing is happening across consumer entertainment platforms

Then, in 2020, beyond DirectX, the Vulkan API ray tracing spec finalized by the committee and the various members from industry and the community feedback.



Looking back at these original 4 demos, 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. Speaking for ourselves the hybrid rendering pipeline we built for our PICA PICA demo, at the top left here, 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. Similarly another great demo from the folks at Futuremark, who always come up with really impressive showcases to push your GPU as far as it can



As mentioned, DICE's Battlefield 5 was one of 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.

And now beyond the initial round of games, a myriad of games followed suit and showcase ray tracing.

I counted 50, not all listed here, but the list keeps growing .

This is really encouraging to see, and totally understandable why everyone is excited about what real-time ray tracing can enable.

Reflections

- Hybrid: screen-space reflections (SSR) & ray tracing
- If valid screen ray
 - Use SSR with BRDF importance sampling [Stachowiak 2015]
- else
 - Trace in world & evaluate BRDF
- Results should blend and match, depending on shortcuts you've taken in your BRDF evaluation



Ray Traced Reflections in Battlefield V [Deligiannis 2019]

Let's talk a bit about the two most common techniques these games support, starting with reflections, which undeniably add a lot to the image

One can do perfect mirror reflections, though the more complex case here is to tackle rough and smooth surfaces.

This is typically done in a hybrid way, for performance reasons, by figuring out which pixels on screen can rely on screen-space reflections first.

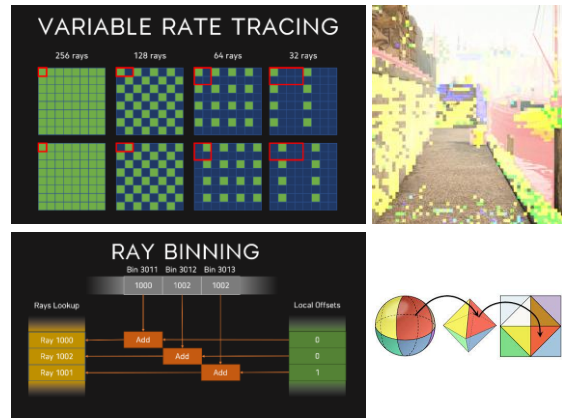
So, if the ray coming from the camera into the scene and its reflection end up on screen, like the puddle of water here from Battlefield V, with the results highlighted in orange, then you can use existing screen information for that reflection.

Otherwise, you trace a ray in the world and evaluate the BRDF. Here in blue.

Results are then merged, and technical should blend & match, but this all depends on some of the shortcuts you might've taken along your BRDF evaluation. Hopefully they fit well with each other.

Reflections

- Achieve performance
 - Variable Rate Tracing
 - Instead of 1:1 rays:pixels
 - Ray Binning
 - Launch rays in direction buckets
 - Shadow map sampling
 - Sample rasterized shadow maps (ie. CSM, local lights) instead of launching secondary rays



Battlefield V's ray tracing pipeline (Deligiannis 2019)

As you probably guessed this approach is taken mainly for performance reasons. Other tricks to achieve performance are needed.

Variable rate tracing, where you evaluate where you should launch more (or less) rays on the screen

Ray Binning, where you launch reflection rays pre-sorted by direction buckets, to help drive the GPU with a more predictable intersection workload.

You can even sample the shadow maps, instead of launching a secondary ray for the reflection's shadows.

As you can see, many tricks here and from the image below this is a glimpse of the whole pipeline that was built to have real-time reflections at performance in Battlefield V.

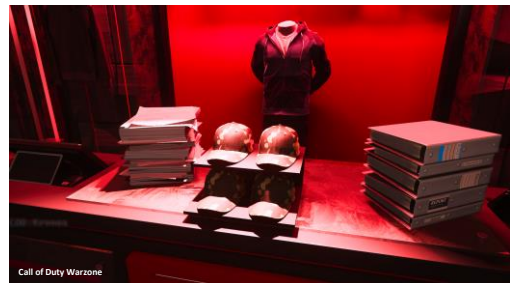
Please check out the talk from Johannes and Jan on this topic, for even more details on the whole pipeline.

Shadows

- Shadows really help in making a visually-convincing and cohesive image
- Contact-hardening, from sharp to rough
 - Hard shadows are great but...
 - Soft shadows convey scale and more representative of the real world
- Simple case: sun light
 - Launch ray towards light
 - If hit, you're in shadow
- Complicated case: area lights!
 - Probabilistic sampling of the area
 - Will require denoising [Olejnik & Kozlowski 2020]



<https://docs.unrealengine.com/en-US/RenderingAndGraphics/RayTracing/index.html>



<https://www.nvidia.com/en-us/geforce/guides/call-of-duty-modern-warfare-pc-graphics-and-performance-guide/>

Shadows is another popular real-time ray tracing technique, mainly for its “simplicity”, between quotes.

Here I say between quotes because at its core this is not too complicated to implement. Just launch a ray from the surface towards the light, and if it hits something you're in shadow, or alternatively if it misses you're not in shadow.

Hard shadows are great... but soft shadows are definitely better to convey scale and more representative of what happens in the real world

Though maybe your art direction wants hard shadows too, with minimal penumbra.

The simple case of a directional sun light 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

As you probably guessed the complex case is area lights, where you need to sample across the area. Here you need denoising, but you'll get really nice and soft shadows like the image on the right here, from Call of Duty. Really awesome blend of sharp and rough shadows, as you'd expect from real-world lights.

The Road Ahead

Considerations for real-time ray tracing research targeting games

Now that we've talked about the state of real-time ray tracing in games I would like to talk about some aspects of the road ahead for real-time ray tracing, from the angle of research

To be more precise, hopefully some of the items mentioned in the next slides should give you some insight into what we think about as game developers when it comes to ray tracing for our games

And some of our challenges. Also, how this could potentially drive and tailor some of your research in case you want to target games or other real-time mediums.

Gamedev RT vs Film RT

Production ray tracing solutions don't always map 1:1 to game ray tracing

- Film
 - Many courses at SIGGRAPH on production ray tracing, lately with a focus on path tracing
 - Complex shading, millions of lights, volume rendering...
 - 24hr+ frame times
 - Custom techniques to handle lack of convergence or noise, sometimes with no robust solution, where throwing more rays means significant production costs (time/\$)
- Games
 - Initial breadth of DXR games support a limited set of ray tracing features
 - Still ways to go before we move to full PT



Production raytracing solutions don't always map one to one to game Ray tracing solutions

For film you will find many courses at SIGGRAPH on production rendering with a focus on path tracing where we talk about complex shading, handling millions of lights, difficult volume rendering scenarios.

Really difficult usecases that render in 24 hour render times on massive clusters

Now, in games, we have a much more limited set of ray tracing features in order to fit in 33 or 16 milliseconds

Still ways to go before we move to full path tracing, like film, but we'll get there someday.

Hybrid Ray Tracing



Figure 1: Hybrid ray tracing in *PICA PICA*.

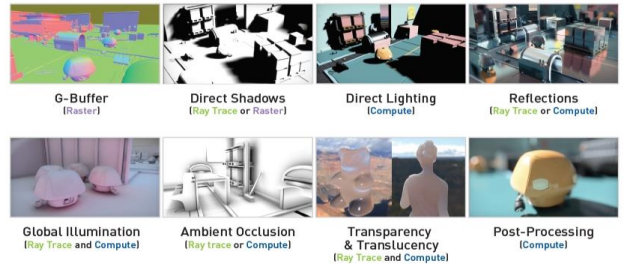


Figure 2: Hybrid rendering pipeline.

Hybrid Rendering for Real-Time Ray Tracing [Barré-Brisebois et al. 2018]

- A common denominator to current ray tracing in games
- Rely and combine the best of each stage
- Foreseen path for ray tracing at performance for the next (few) years



The first thing is that most of the work we do is tailored around maximizing the hardware that we have. Whether that's on PC or consoles.

A common denominator to achieve performance with these techniques is that they are built on this concept of a hybrid pipeline, where different aspects of rendering are solved with the best tool at hand

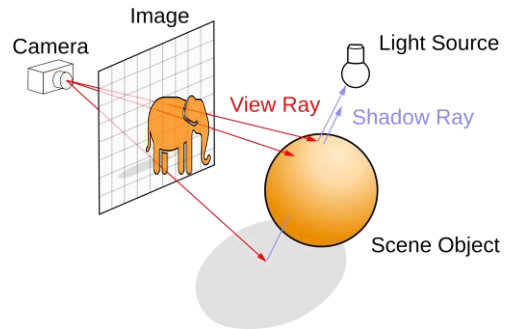
For example, you might use rasterization for some of the rendering. Rasterization is very good at what it does, and GPUs do it very well, so let's use it to its full potential.

Compute shaders as well, and the programmability that they unlock, for some aspects of the pipeline. Some techniques might also mix and match various stages, and that's OK if that's what is needed for performance.

The Game Ray Tracing Elephant (vs Academia)

Elephant in the room

- Ray tracing can diverge to how we currently think of rendering in games
 - What you look at is often what is assumed
- With RT, now need to have everything upfront & available for tracing rays
 - A lots of systems are not designed for this
 - Need to know all materials upfront
 - Rays can hit things that are really far – need to handle LODs
 - Also, need to handle animations outside the viewport



Now, the elephant in the room here, as we go through this transition and add more ray tracing to our games, at its core ray tracing can diverge to how we currently think of rendering in games

In games what you look at is often what is assumed to require processing

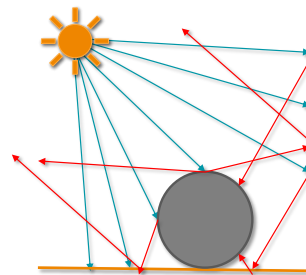
But, with ray tracing many systems need to be adapted where everything is expected to be available and provided upfront

Rays can end up anywhere in the scene, meaning that materials need to be known upfront

Rays can also hit objects that are really far, so you need to handle various Level of Details for geometry and textures, maybe things you don't typically have loaded and resident in memory

Animations is another one that comes to mind, of which I'll cover in a few slides

Adapting Game Systems to RT – World State



- What you look at is often what is assumed
 - « Don't render things you don't see »
- Can't solely rely on frustum culling since rays are in world space
- Cost: prohibitive update of all objects in BVH
 - Animations & dynamic geometry

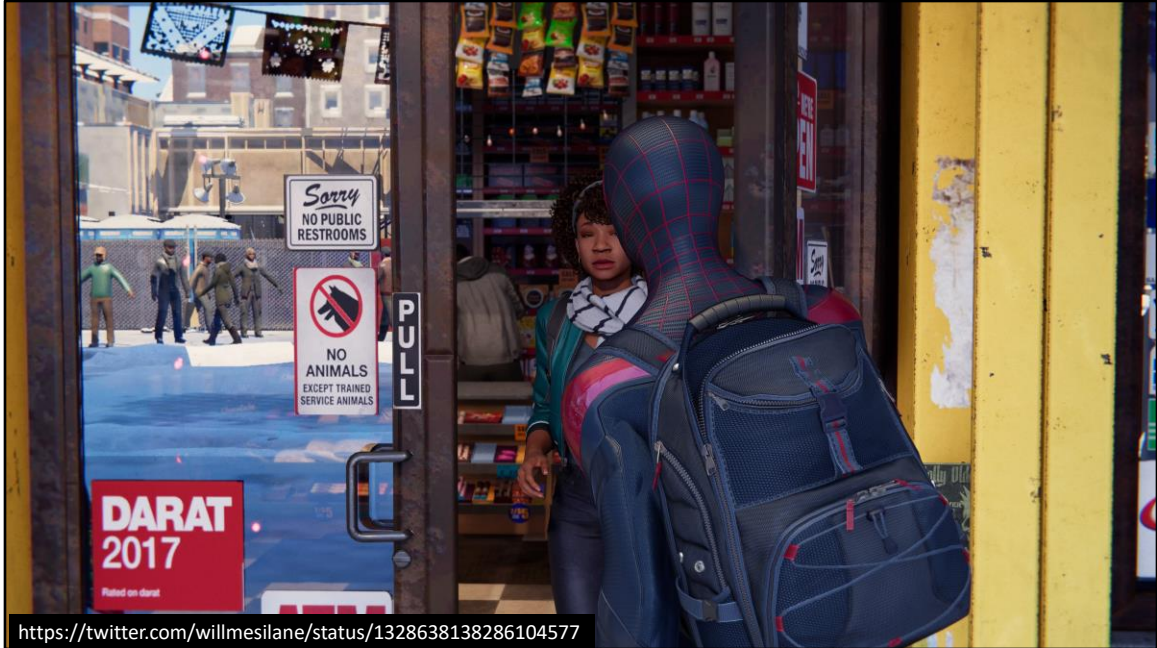
The first one that comes to mind is how you handle world state

Typically you don't render or even load things you don't see, which sounds pretty obvious.

With ray tracing now you can't rely on rendering workload reduction only with frustum calling, because rays are launched in world space and again can go behind or outside the frustum

And just loading everything and updating everything is not really a solution, and it's actually quite prohibitive to constantly update a BVH

This means you need to handle animations and dynamic geometry to the best of your ability, predict what needs updating, and clever do so to manage performance



Here's an example where in this game you can notice that some of the characters in the reflection are in what we call T-pose, meaning they are in a default state and are not being animated

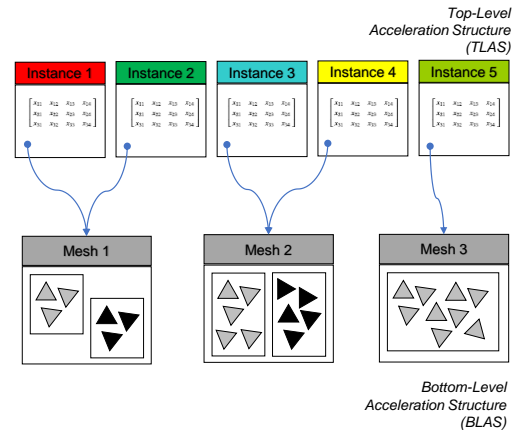
Their position in the world is right, but the animation was not updated.

Some other characters look like they are walking, so it's clearly driven by what is being budgeted per frame for updating animations with some sort of round-robin prioritization.

You can also notice that some objects are also lower resolution in terms of shading. Again, most likely for performance reasons of not having everything loaded at highest quality and being able to update every dynamic object, every frame, for a big chunk of the game world that can be reflected in that window.

Updating a BVH is Not Free

- Bounding Volume Hierarchy (BVH)
 - Responsible for accelerating ray traversal
 - Supported by hardware ray/triangle, ray/AABB intersection testing
- TLAS / BLAS Battlefield V [Deligiannis 2019]
 - TLAS: 20 000 instances
 - BLAS: 5000 meshes
 - Naive TLAS + BLAS build : 60ms
 - Staggered full & incremental BLAS rebuild
 - N-frames incremental before full rebuild
 - Don't rebuild if matrices haven't changed
 - BLAS: 50/frame
 - TLAS: 2800/frame
 - TLAS + BLAS build (GPU): 1.15ms
 - RayGen (GPU): 0.7ms → 0.8ms



If we talk about numbers, here's what I mean when I mention that updating a BVH is not free

As you know a BVH is the structure used to accelerate ray traversal in the scene
Latest generation GPUs accelerate this by providing hardware ray/triangle and ray/box intersection

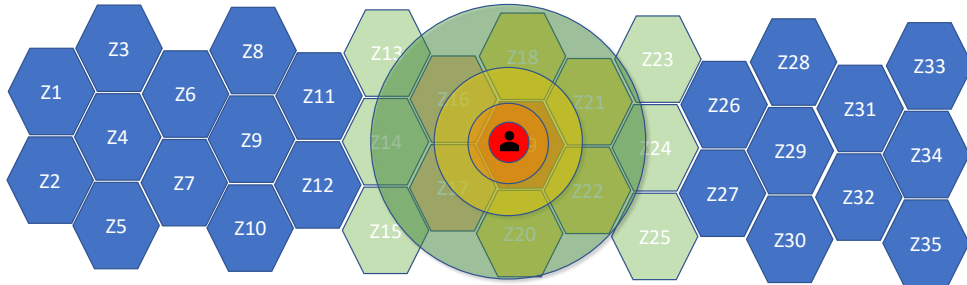
This is supported by a two-layer structure, top and bottom

In Battlefield 5, the example here is 20 000 top-level instances and 5000 bottom-level meshes

A naïve update of the whole BVH is 60 milliseconds

Instead of updating the whole thing, by cleverly balancing rebuilds and refits, you can bring this down to 1.15ms/frame while slightly augmenting the cost of your trace
This means that the BVH is not perfect every frame, but you've significantly reduced the update cost without affecting tracing too much

Adapting Game Systems to RT - Streaming



- Rays can go in any direction
 - Off-screen mirror reflections
- No more concept of loading based on camera orientation
 - Can't prioritize based on user-driven view
 - le.: camera, weapon scope
- Much higher resource residency required

LOD 0 - Near
LOD 1 - Mid
LOD 2 - Far
LOD 3 - Distant
LOD 4 - Vista

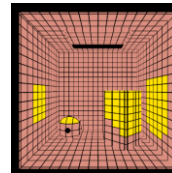
Streaming is another case that comes to mind that needs to be adjusted for ray tracing.

The well-used concept of positional loading prioritized by camera orientation goes out the window

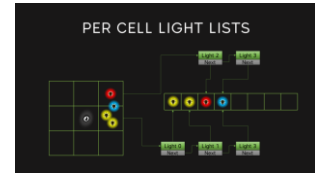
Ultimately this means that more resources need to stay resident at a high enough resolution so that visuals (and transitions) are coherent

Adapting Game Systems to RT – 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]
 - Spatiotemporal reservoir resampling [Bitterli 2020]



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



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

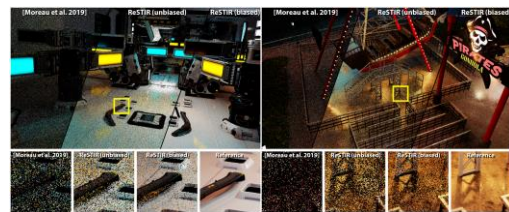
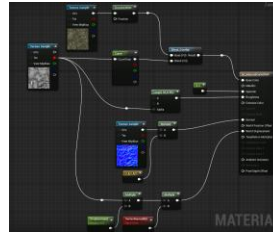


Fig. 1. Two complex scenes ray traced with direct lighting from many dynamic lights. (Left) A still from the Zero Day video [Vieljeux 2015] with 11,000 dynamic emissive triangles. (Right) A view of one ride in an Amusement Park scene containing 3.4 million dynamic emissive triangles. Both images show three methods running in equal time on a modern CPU, from left to right: Moreau et al. [2019]'s efficient light sampling BVL, our new unbiased estimator, and our new biased estimator. The Zero Day image is rendered in 35 ms and Amusement Park in 48 ms, both at 1000 x 1000 resolution. Zero Day (top-left), Pleasure Ship (bottom-left).

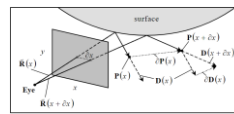
How to handle lights is another one that comes to mind.
 So how do you choose which lights to sample when a ray hits a surface?
 You can use a camera-oriented acceleration structure, like what Unity does
 Or you can use a horizontal plane with per-cell light lists
 Another approach is to treat this as an importance sampling problem, with some great papers by Moreau and Yuksel
 And lately the spatiotemporal reservoir resampling paper from Bitterli, that shows how reservoir sampling can accelerate convergence for both biased and unbiased use cases, and handle millions of lights in milliseconds.

Adapting Game Systems to RT – Materials

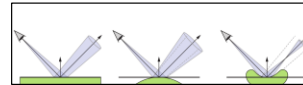
- Game engines rely heavily on artist-authored material graphs
 - Procedural Geometry
 - Vertex displacement, instance vertex animation
 - Transparency / alpha-mask
 - Transparency, particles, vegetation
 - Rasterization as a way to avoid geometry
 - RT: use any-hit shaders for parametric ray termination
 - Requires shader invocation at hit
 - Derivatives
 - $ddx(x)$, $ddy(x)$: texture sampling
 - Ray Differentials [Igehy 1999]
 - Ray Cones [Amanatides 1984]
 - Improved Shader and Texture Level of Detail Using Ray Cones [Akenine-Möller 2021]



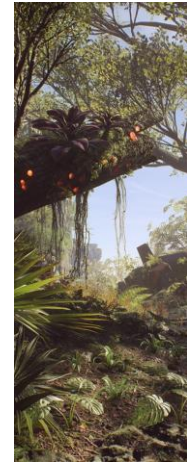
Unreal Engine Material Graph



Ray Differentials [Igehy 1999]



Ray Cones [Amanatides 1984]



Another item here is material graphs

Game engines heavily rely on artist-driven material graphs where a lot of features

assume concepts easily available for rasterization but not for ray tracing

Procedural geometry like vertex displacement or instance vertex animation in a

shader affects the geometry in the BLAS, so it will require refits and rebuilds. A refit

or a rebuild is not something a shader can typically trigger.

Transparency where the mask is used to clip is another example. That concept is

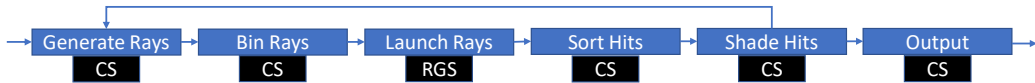
much more complex to implement with any-hit shaders, and requires shader

invocation which can affect performance.

Or pixel quad derivatives, to choose which mip level when sampling a texture. No

pixel quad for ray tracing.

Towards Decoupled Real-Time Ray Tracing



- A generalized pipeline for decoupled ray tracing [Eisenacher 2013]
 - Sort large out-of-core ray batches & ray-hits for deferred sharing & shading
 - Group work items together that make sense to optimize GPU workloads, occupancy, and overall performance
 - 6 stages that can be adapted based on your needs

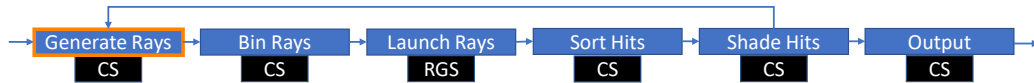
3 years in now into real-time ray tracing, and now that APIs are moving towards being able to launch rays from any shader stage, a trend is to move towards more decoupled ray tracing

Here's a high-level example of a generalized pipeline where large out-of-core ray batches & ray-hits are sorted for deferred sharing & shading

Group work items together that make sense, to optimize GPU workloads, occupancy, and overall performance

Basically 6 stages that can be adapted based on your needs

Towards Decoupled Real-Time Ray Tracing

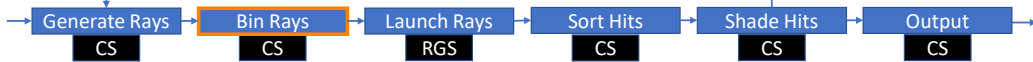


- **Generate Rays**

- Build a list of rays that are required for ray tracing
- Launch in (X) space: screen/view, texture, etc.
- Handle any resolution

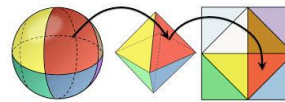
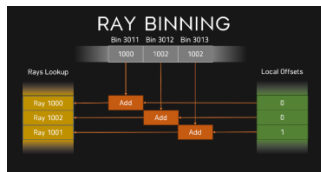
First step, in a compute shader, is to build a list of rays that you need for ray tracing
Here because this is general you can think of launching rays in screen space, in texture space or any other parametrization of your choice
Can also do this at any resolution

Towards Decoupled Real-Time Ray Tracing



- Bin Rays

- Group rays that are directionally aligned to maximize coherency [Deligiannis 2019] [Benyoub 2019] [Majercik 2019]
 - Sort rays in octahedral space for ray direction binning



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

Then you bin the rays to maximize coherency

One way to do this is to sort rays in some kind of space that allows you to bucket them by direction.

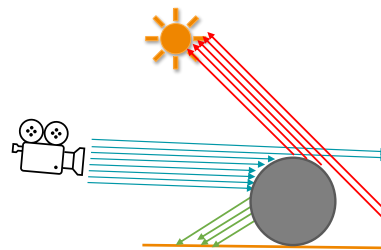
Octahedral space is perfect for this.

Binning can go beyond directions, like ray types that you know will do similar work, but direction is definitely a common way to do this

Towards Decoupled Real-Time Ray Tracing



- Launch Rays
 - For each bin, launch rays
- Coherency
 - Adjacent work performing similar operations & memory access
 - Primary rays
 - Shadow Rays
 - Reflection Rays
 - ...

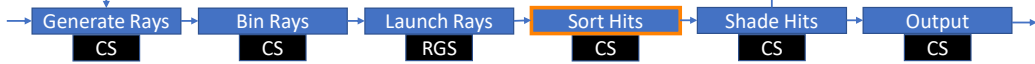


Then, for each bin you launch the rays

This is to manage coherency which is key for performance

So grouping things together that perform similar operations and memory accesses

Towards Decoupled Real-Time Ray Tracing



- Sort Hits

- By material ID
- GPUs like predictable workloads
 - Switch the focus to what costs: evaluating materials
 - To maximize GPU occupancy & wavefronts

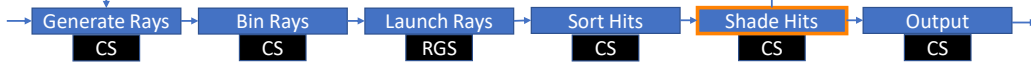
- Group and batch a series items to later shade



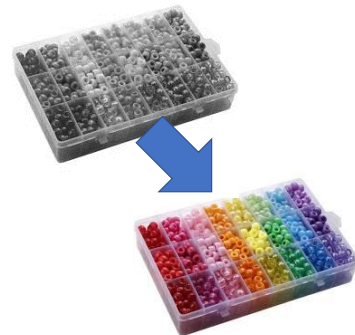
Beads by BEADNOVA (etsy)

Then, hits are gathered and sorted, by material ID, for example
You don't shade just yet, but instead sort all the unsorted hits, into something more organized and regrouped
GPUs like predictable workloads and shading random materials from random rays doesn't really align here
This is why this stage is important. By batching similar things, the overall GPU occupancy will later be improved

Towards Decoupled Real-Time Ray Tracing



- Shade Hits
 - Compute shader that runs based on each hit “type” to shade
- Secondary rays
 - Hits might require secondary shading (ie.: reflections, shadows, GI)
 - Queue secondary rays to *Generate Rays stage*



Beads by BEADNOVA (etsy)

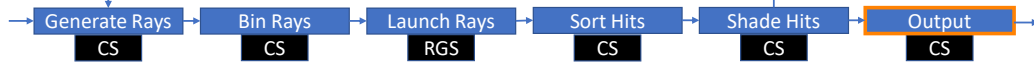
This next stage is when hits are shaded, where basically a compute shader runs based on each hit type to shade

By grouping common work items together, like materials of a certain type, you’re helping the GPU

This is also where you might launch secondary rays, for things like reflections or shadows, or global illumination

And those would enqueue back into the ray generation stage

Towards Decoupled Real-Time Ray Tracing



- Output
 - Reconstruction & reuse
 - Merge with other non-RT (ie.: volumetric)
- A pipeline that can be customized for you needs, oriented towards maximizing the GPU



Beads by BEADNOVA (Etsy)

Finally the last stage where the final result is put together

You might also handle reconstruction at different resolutions, reusing results, and merging with non-ray tracing effects

The devil is in the details, of course, especially when it comes to reusing and merging results at different rates and resolutions to reduce variance

But overall this should give you a good idea of a general decoupled pipeline for real-time ray tracing in games

Open Problems

Fun things to think about. Maybe you can help solve?



General RTRT Open Problems

- Scheduling
 - Instruction cache, occupancy
 - SIMD / data cache coherence
 - Just-in-time BLAS construction
 - Traversal Shaders to build the entire AS when necessary [Lee 2020]
- Decoupling
 - Reuse of intermediate results across paths & frames
- Noise
 - Optimal MIS to reduce variance [Pharr 2018]
 - Prefiltering + sampling + denoising



There are still a ton of general open problems

Open Problem #1

Massive & Custom Geometry Renderers

- Specialized “software” rasterizers
- Micro-polygon renderer
- Strand-based Hair
 - Strand-based software rasterizer
 - Geometry needed for BVH build
- Micro-polygon Geometry
 - Micro-polygon renderer
 - Constant streaming of geometry detail
 - “Pixel is the new triangle”
- How do we handle these?



Frostbite Engine Hair [Taillandier 2020]



Unreal Engine 5 Nanite

In this world of hybrid pipelines we see more and more implementations of custom geometry renderers


The amazing strand-based hair that the Frostbite team has presented and shipped with the EA Sport's FIFA team is a good example of a very optimized compute-based software rasterizer. With ray tracing now, this software rasterizer has to interface and feed into the ray tracing acceleration structures, which are triangle and AABB-based, and not just render the final pixel on screen

Also recently, Unreal Engine 5's micro-polygon geometry renderer Nanite, is bringing super high level of details to real-time where pixel is the new triangle. This means constant streaming of geometry detail, and in the world of ray tracing as we saw can greatly affect acceleration structures and performance. Same thing here.

So, how do we handle these new trends?

Open Problem #2

Limitations of two-level hierarchies

- A tree blowing in the wind?
- 2-level hierarchy
 - Increased storage via unique data
 - Increased pressure on building the BLAS
 - Top: forest
 - Bottom: tree
- 3-level hierarchy
 - Top: forest
 - Mid: tree branches/trunk
 - Refit
 - Bottom: leaf
 - Instance transform
- “All problems in computer science can be solved by another level of indirection” 



The next open problem is around challenges of a two-level hierarchy

Let's take the example of a tree blowing in the wind

You can implement this with a 2-level hierarchy where in the top level lives all the trees, so the forest really,

And the bottom is the actual tree, which needs rebuilds over time as it animates in the wind

This leads to increased pressure of rebuilding the bottom level as the leaves animate, and a lot of unique geometry

Alternatively, with a 3-level hierarchy where you can split the forest, the tree and its branches, and narrow down and reduce how much rebuilds and refits you need.

I'm sure you can think of other examples where another level of indirection could help.

What's the saying again “**All problems** in computer science **can be solved** by another level of **indirection**”?

Open Problem #3

Lenses (rifle/scope) & mirrors VS LODs & Streaming

- Expanding lens and impacts on LOD/streaming



<https://commons.wikimedia.org/w/index.php?curid=18200682>
By S10, Wikimedia Commons, CC BY-SA 3.0



<https://pixabay.com/photos/binoculars-looking-man-discovery-1209011/>

Another interesting problem is lenses and mirrors.

Because of reflections and refractions, it's now very difficult to predict where a ray will end up in the scene, and the impacts on LODs and streaming

A ray could end up reflecting or refracting into an unpredictable area, and considering that ray tracing expects things to be known upfront this can be an issue

Not necessarily a crash, but more adverse effects like things popping-in, or not visually being coherent

Users can notice this, and this can greatly affect the overall experience.

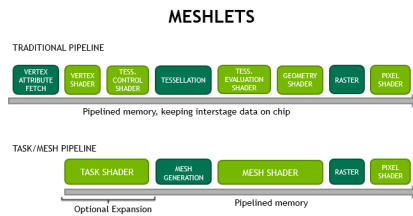
Open Problem #4

Mesh Shaders vs Ray Tracing

- Enables geometry expansion/reduction
 - Brings exciting & new possibilities

Arbitrary data packet that expands into an arbitrary set of primitives

- Hit shader
 - Hit a meshlet, require to invoke Task / Mesh Shader
 - Non-trivial expansion for ray-test (versus hardware ray-triangle or ray-AABB)
 - Trigger / feed-into BVH build?



Lastly, mesh shaders and ray tracing.

The concept of mesh shaders is a new exciting concept that simplifies the traditional rasterization pipeline with its vertex, tessellation and geometry shader stages and replaces it with 2 stages, task shaders and mesh shaders.

The challenge here between mesh shaders and ray tracing is the fact that arbitrary data packets, or meshlets, can expand into an arbitrary set of primitives.

In the case of building games with both mesh shaders and ray tracing, and trying to keep things coherent, a hit shader that arbitrarily hits a meshlet would then have to invoke a task or mesh shader, which means a non-trivial expansion for ray testing, compared to the hardware support for ray-triangle and ray-AABB. Sounds like quite the performance challenge, and so, maybe there is need for such an expansion to dynamically feed into the BVH builder for performance?

Some things to think about here.

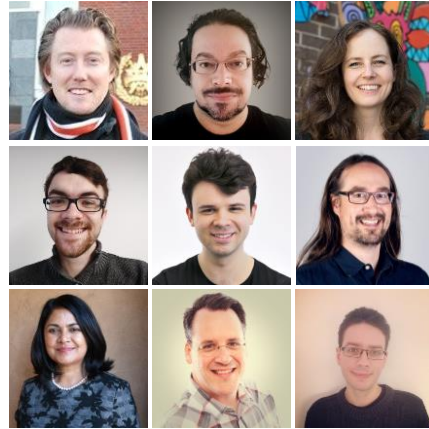
Summary

- Real-time ray tracing adoption in games has significantly moved forward since 2018
 - Many games on PC
 - Adoption on consoles too!
- Still have **many** challenges to solve
- We're in this together
 - Let's work together on this!

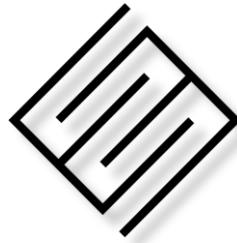


Thank You

- Ulf Assarsson (Chalmers)
- Ari Silvenoinnen (Activision)
- Jon Greenberg (SEED)
- Vicki Ferguson (SEED)
- Henrik Halén (SEED)
- Keven Villeneuve (SEED)
- Chris Lewin (SEED)
- Uma Jayaram (SEED)
- Jim Preston (SEED)
- Yuriy O'Donnell (Epic Games)



I would like to thank the following individuals, who were part of making this talk happen



SEED
seed.ea.com
@SEED

Of course, check out our website, seed.ea.com and on twitter @ SEED

References

- [Aalto 2018] Aalto, Tatu. *Experiments with DirectX Raytracing in Remedy's Northlight Engine*, GDC 2018.
- [Akenine-Möller 2021] Akenine-Möller, Tomas. Crassin, Cyril. Boksanaky, Jakub. Belcour, Laurent. Pantelev, Alexey. Wright, Oli. *Improved Shader and Texture Level of Detail Using Ray Cones*, Journal of Computer Graphics Techniques (JCGT), vol. 10, no. 1, 1-24, 2021.
- [Amanatides 1984] Amanatides, John. *Ray Tracing with Cones*, SIGGRAPH 1984.
- [Barré-Brisebois 2018] Barré-Brisebois, Colin. Halén, Henrik. Wihlidal, Graham. Lauritzen, Andrew. Bekkers, Jasper. Stachowiak, Tomasz and Andersson, Johan. *Hybrid Rendering for Real-Time Ray Tracing*, Chapter 25 in *Ray Tracing Gems*, edited by Eric Haines and Tomas Akenine-Möller, Apress, 2019.
- [Benyoub 2019] Benyoub, Anis. *Leveraging Ray Tracing Hardware Acceleration In Unity*, Digital Dragons 2019.
- [Bitterli 2020] Bitterli, Benedikt. Wyman, Chris. Pharr, Matt. Shirley, Peter. Lefohn, Aaron. Jarosz, Wojciech. *Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting*. ACM Transactions on Graphics, 2020.
- [Deligiannis 2019] Deligiannis, Johannes. Schmid, Jan. *"It Just Works": Ray-Traced Reflections in Battlefield V*, GDC 2019.
- [Epic 2018] Epic Games demonstrates real-time ray tracing in Unreal Engine 4 with ILMxLAB and NVIDIA, GDC 2018.
- [Epic 2020] A first look at Unreal Engine 5, <https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5>.
- [Eisenacher 2013] Eisenacher, Christian. Nichols, Gregory. Selle, Andrew. Burley, Brent. *Sorted Deferred Shading for Production Path Tracing*, EGSR 2013.
- [Igehy 1999] Igehy, Homan. *Tracing Ray Differentials*, SIGGRAPH 1999 Proceedings.
- [Keller 2019] Keller, Alexander. Viitanen, Timo. Barré-Brisebois, Colin. Schied, Christoph. McGuire, Morgan. *Are we done with ray tracing?* Course, SIGGRAPH 2019.
- [Karis 2020] Karis, Brian. Nanite. <https://twitter.com/BrianKaris/status/1283057629506367488?s=20>.
- [Lee 2020] Lee, Won-Jong. Liktor, Gabor. *Lazy Build of Acceleration Structures with Traversal Shaders*, SIGGRAPH Asia 2020.
- [Majercik 2019] Majercik, Alexander. Guertin, Jean-Philippe. Nowrouzezahrai, Derek. McGuire, Morgan. *Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields*. Journal of Computer Graphics Techniques (JCGT), vol. 8, no. 2, 1-30, 2019.
- [Microsoft] DirectX Raytracing (DXR) Spec: <https://github.com/microsoft/DirectX-Specs/blob/master/d3d11/Raytracing.md>.
- [Olejnik 2020] Olejnik, Michal. Pawel, Kozłowski. *Raytraced Shadows in Call of Duty: Modern Warfare*, Digital Dragons 2020.
- [Pharr 2018] Pharr, Matt. *Adopting Lessons from Offline Ray-Tracing to Real-Time Ray Tracing for Practical Pipelines*, "Advances in Real-Time Rendering in Games" course, SIGGRAPH 2018.
- [Schreier 2017] Schreier, Jason. *Horizon Zero Dawn Uses All Sorts Of Clever Tricks To Look So Good. Frustum Culling Video*, Kotaku, online. 2017.
- [Stachowiak 2015] Stachowiak, Tomasz. Yuludag, Yassin. *Stochastic Screen-Space Reflections*, Advances in Real-Time Rendering Course, SIGGRAPH 2015.
- [Tatarchuk 2019] Tatarchuk, Natalya. *Towards Filmic Quality at 30 FPS: Real-Time Ray Tracing for Practical Game Engine Pipelines*, GDC 2019.
- [Taillandier 2020] Taillandier, Robin. Valdes, Jon. *Every Strand Counts: Physics and Rendering Behind Frostbite's Hair*, Digital Dragons 2020.
- [Yukse 2019] Yukse, Cem. *Stochastic Lightcuts*, HPG 2019.



Merci! Thank You!

Colin Barré-Brisebois
Head of Technology, SEED, Electronic Arts
@ZigguratVertigo

