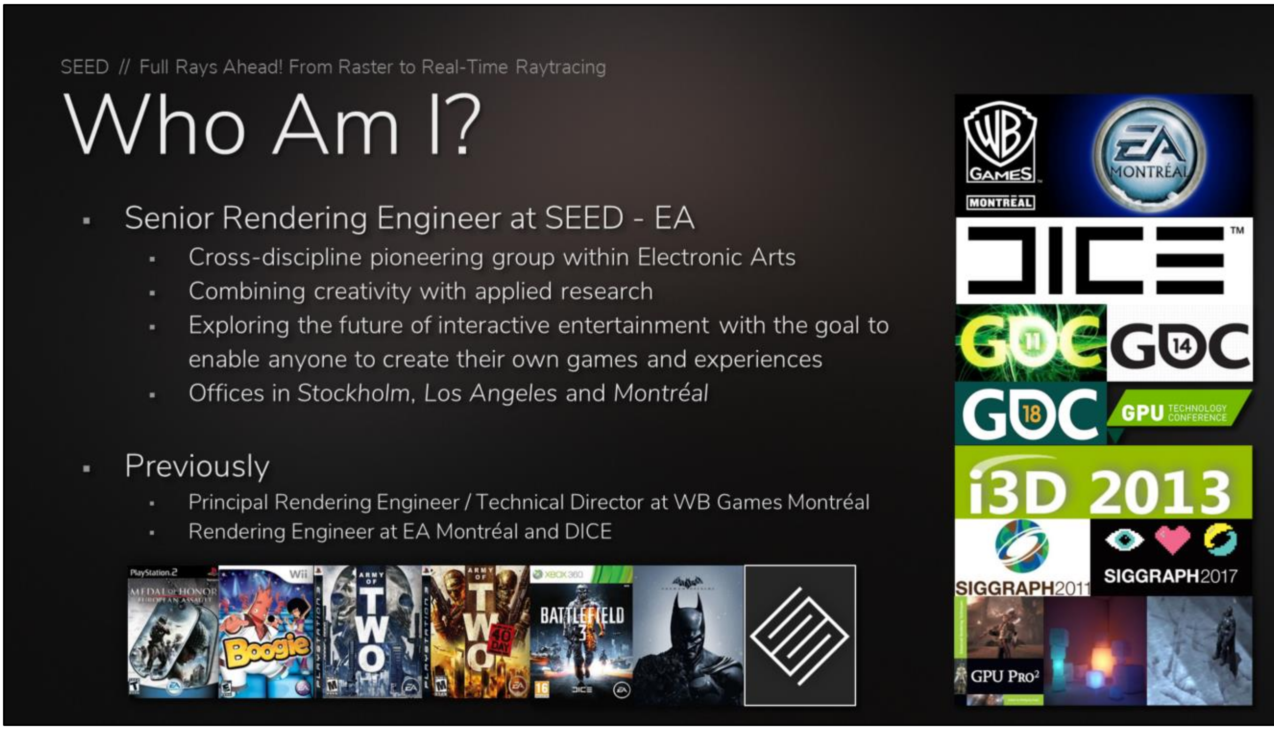
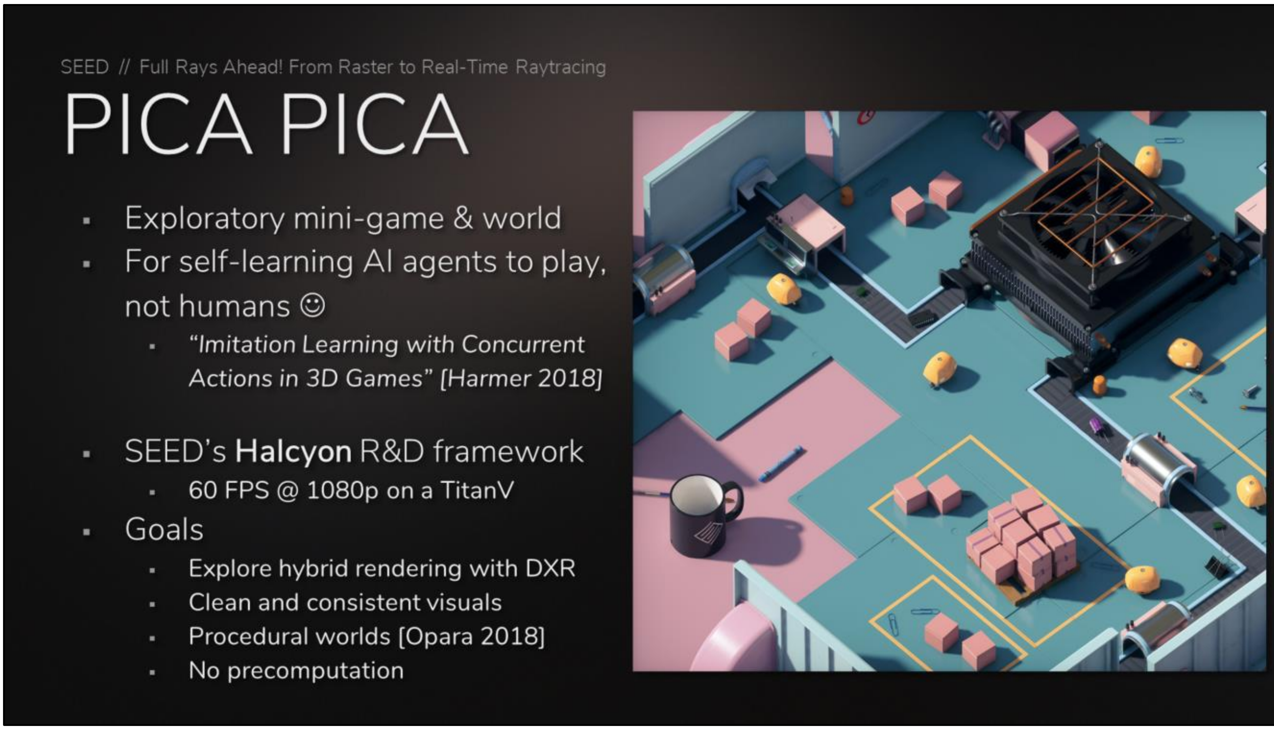
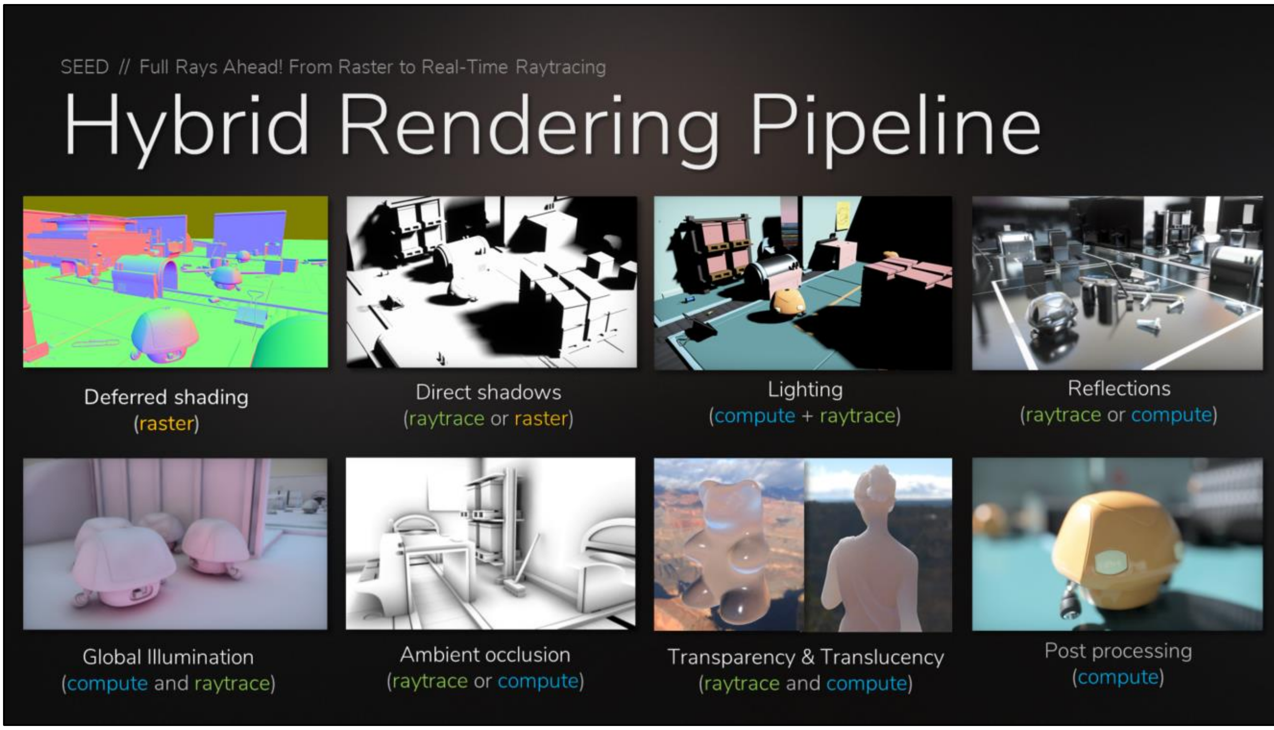

- Hi everyone and thanks Chris for having me as part of this course!
- My goal now is to give you some insights on how us at SEED handled the transition from raster & compute to real-time raytracing using DXR.
- I only have 20 minutes so will try to give you as much info, but we can definitely chat after if you have any questions whatsoever.



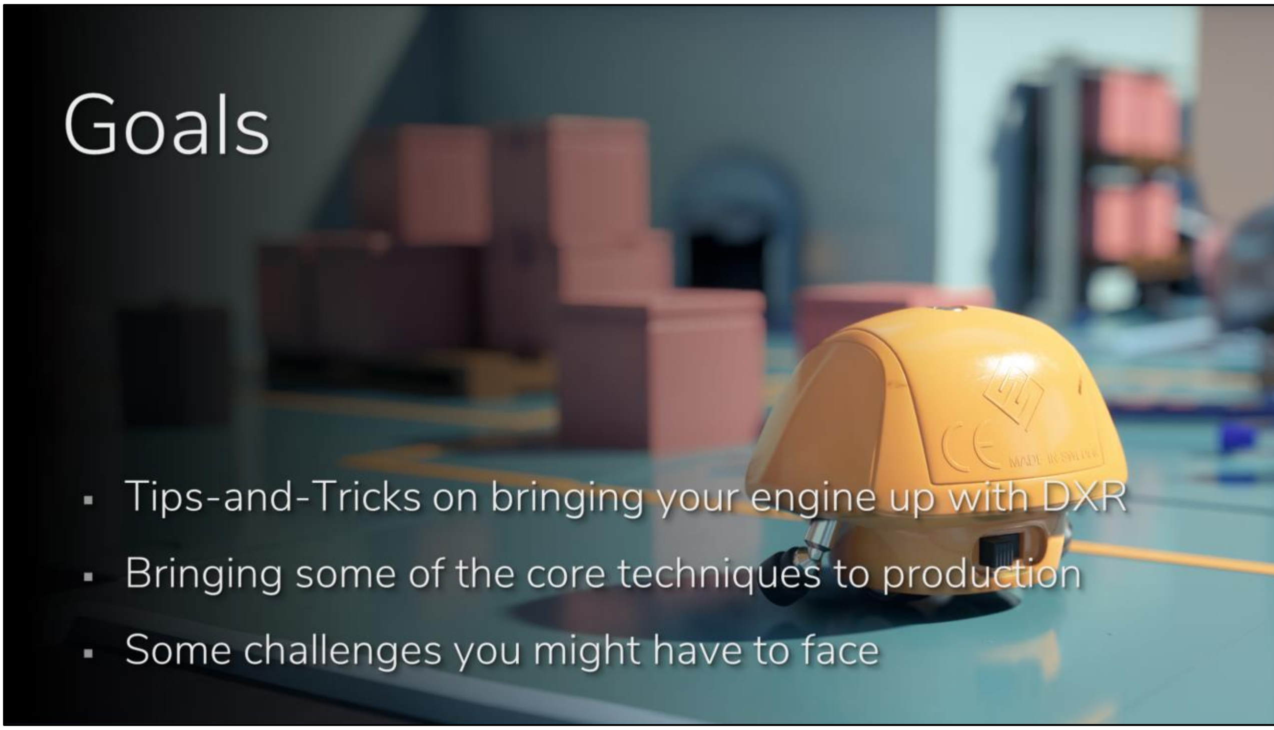

- So who is SEED? For those of you who don't know, we're a technical and creative research division inside Electronic Arts.
- We are a cross-disciplinary team with a mission to explore the future of interactive entertainment, with the goal to enable anyone to create their own games and experiences.
- We have offices in Stockholm, Los Angeles and Montreal.
- One of our recent projects was an experiment with hybrid real-time rendering, deep learning agents, and procedural level generation.
- We presented this at this year's GDC. In case you haven't see it, let's check it out!




- PICA PICA is a mini-game that we built for AI agents rather than humans.
- Using reinforcement learning, the agents learn to navigate and interact with the environment. They run around and fix the various machines, so that conveyor belts keep running efficiently. If you are interested in the AI part, do check out our paper in arXiv. The paper talks about how we train the agents, what they see, and how they communicate with game logic.
- We built the mini-game from the ground up in our in-house Halcyon R&D framework. It is a flexible experimentation framework that is still young, yet already capable of rendering shiny pixels.
- We've had the opportunity to be involved early on with DirectX Raytracing, thanks to our partners NVIDIA and Microsoft, to explore some of the possibilities behind this piece of technology.
- We decided to create something a bit different and unusual, less AAA than what you usually expect from an EA title.
- For this demo we wanted cute visuals that would be clean and stylized, yet grounded in physically-based rendering. We wanted to showcase the strengths of raytracing, while also taking into account our small art department of 2 people.
- We used procedural level generation with an algorithm that drove layout and asset placement. You should check out Anastasia's various talks on the matter in case you want to know more how the world was procedurally generated.
- Pica pica runs at 60 fps on a titanV in 1080p

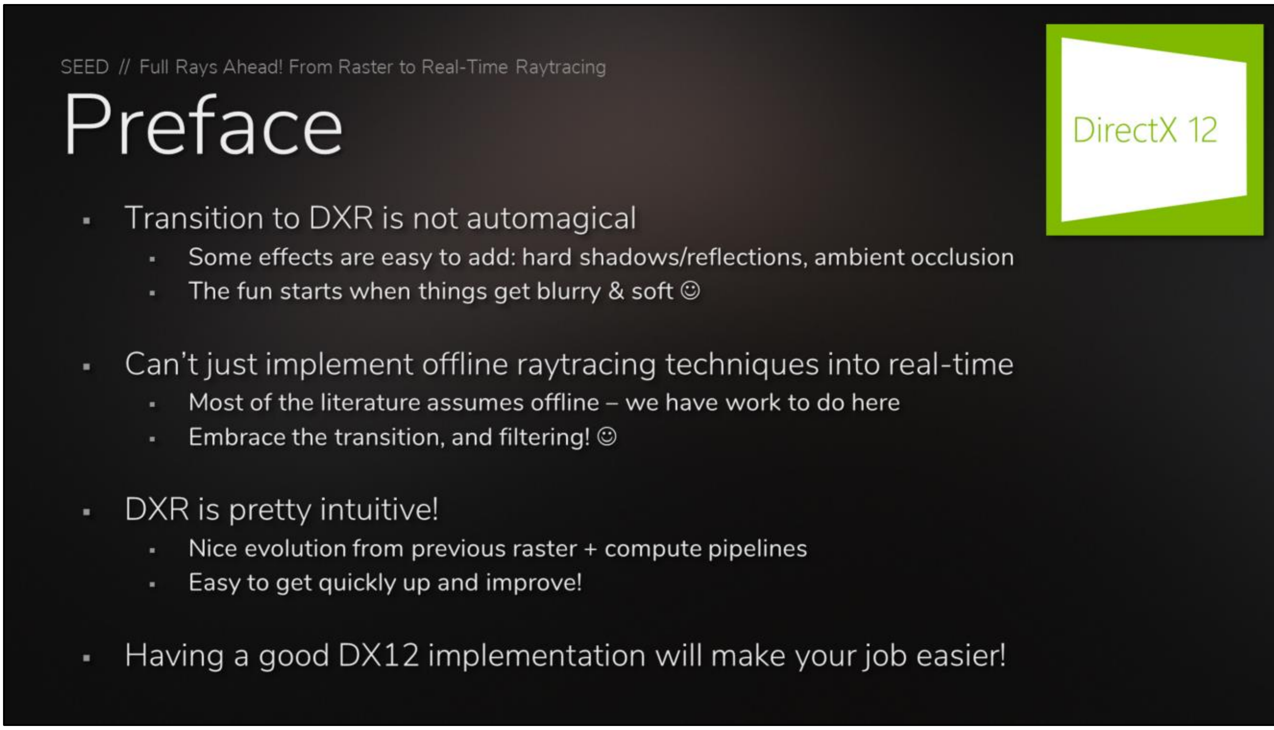



- And so to achieve this we built a hybrid of classic rendering techniques with rasterization and compute, with some raytracing on top. We take advantage of the interop between rasterization, compute and raytracing.
- We have a standard deferred renderer with compute-based lighting, and a pretty standard post-processing stack.
- We can render shadows via raytracing or cascaded shadow maps.
- Reflections can be raytraced or screen-space raymarched. Same story for ambient occlusion.
- Only global illumination and translucency requires raytracing.

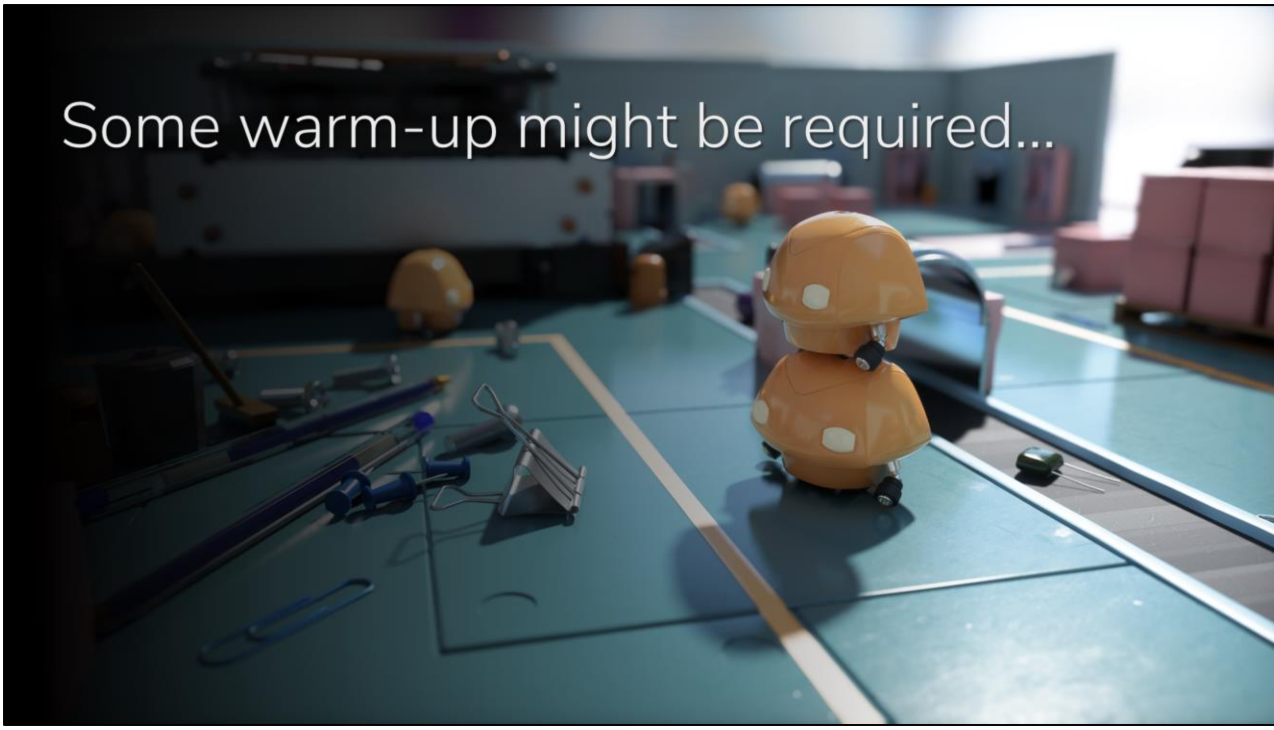



- With this said, the goal of this presentation is to provide you with some tips and tricks on how to evolve your engine from being raster and compute to supporting raytracing with DXR.
- I'm hoping to give you some practical insights on bringing some easy-to-do core raytracing techniques to production, but also warn you about some challenges you might face as you go through this awesome transition.

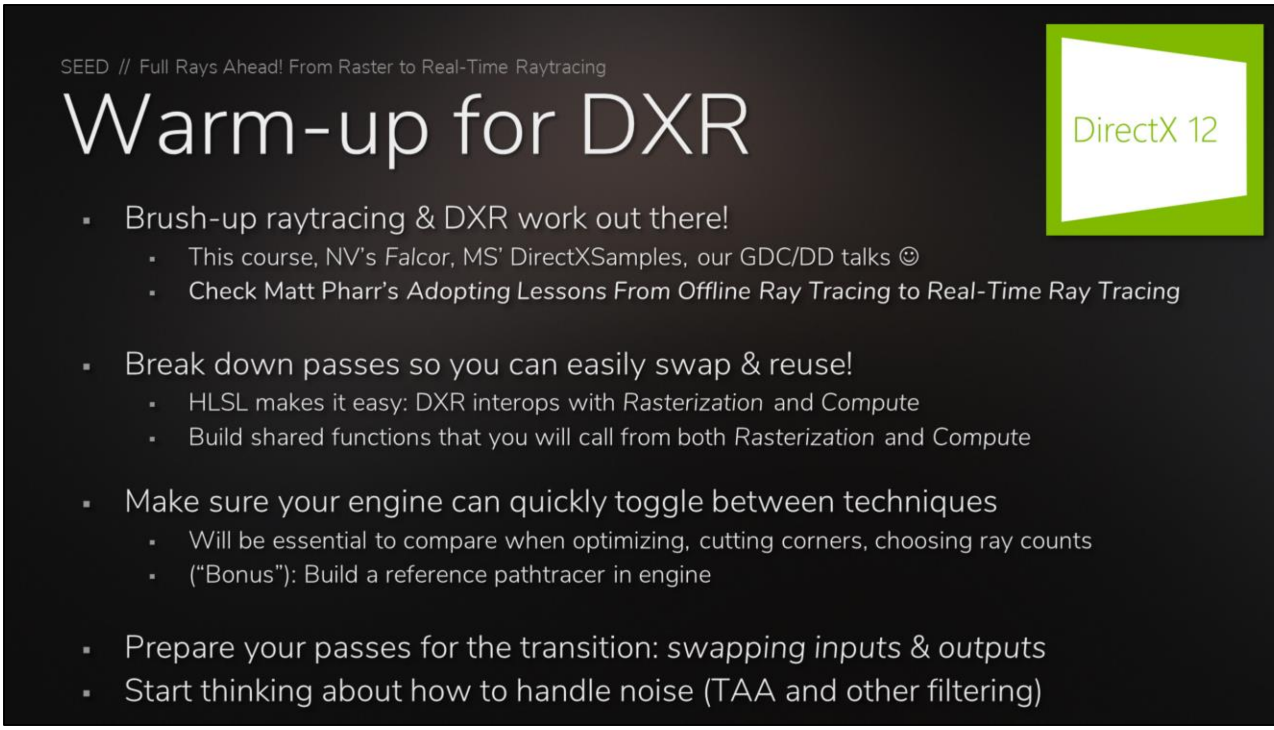

- So one question one might have is… well I'm a developer and I want to take advantage of DXR… so what should I do?



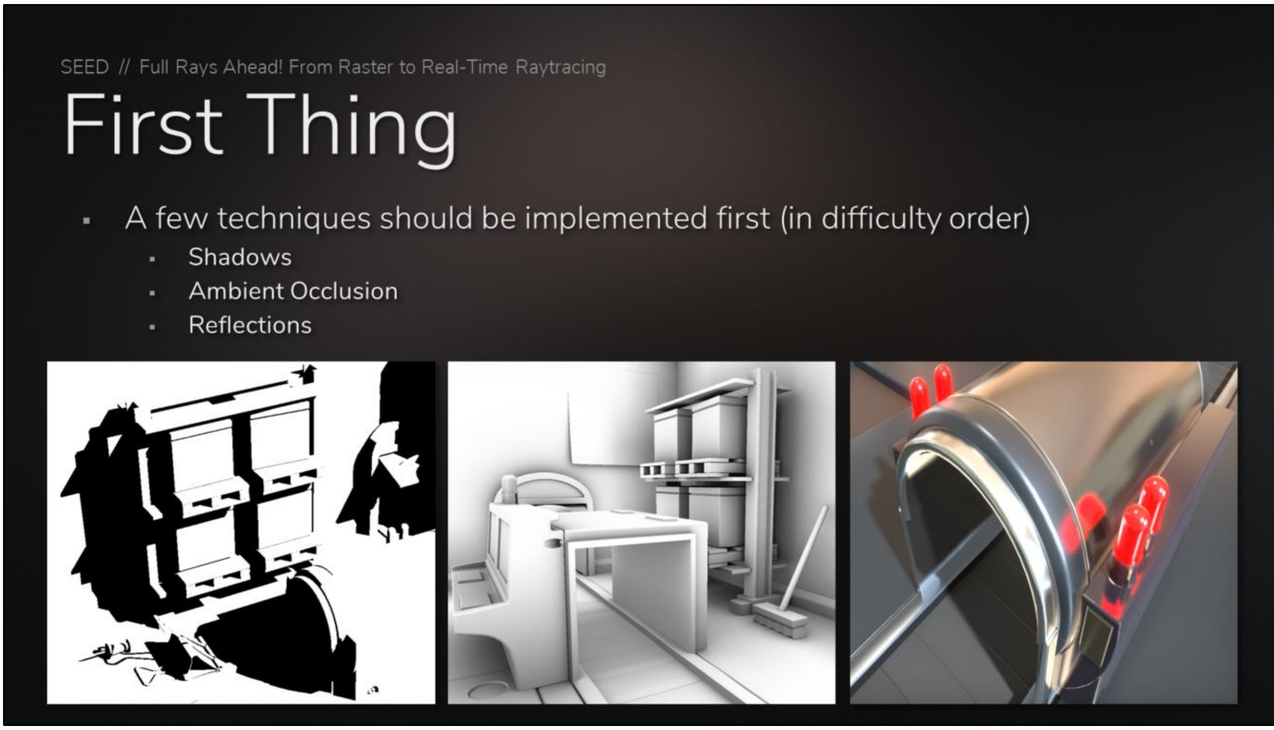

- One should know that the transition to DXR won't be automagical. Some effects and techniques are easy to add, such as hard shadows, hard reflections, and ambient occlusion. But once that's done you'll see that quickly the fun starts when things get blurry.
- Actually most raytracing techniques from books assume offline raytracing, so there some work to do there if one wants to adapt some techniques for real-time. This means you need to embrace filtering and adapting techniques to the constraints of real-time.
- That said, from the previous talks you saw that the API is pretty intuitive, so out of the box that's neat! It should be easy to get up and running, implement stuff and improve. If your engine already has a good DirectX12 implementation, then your job should be pretty easy.
- You need to have a good grasp on your resources, and how those get packaged together for the scene that you put together for raytracing and the various new concepts, such as *Shader Table* and *Acceleration Structures*.



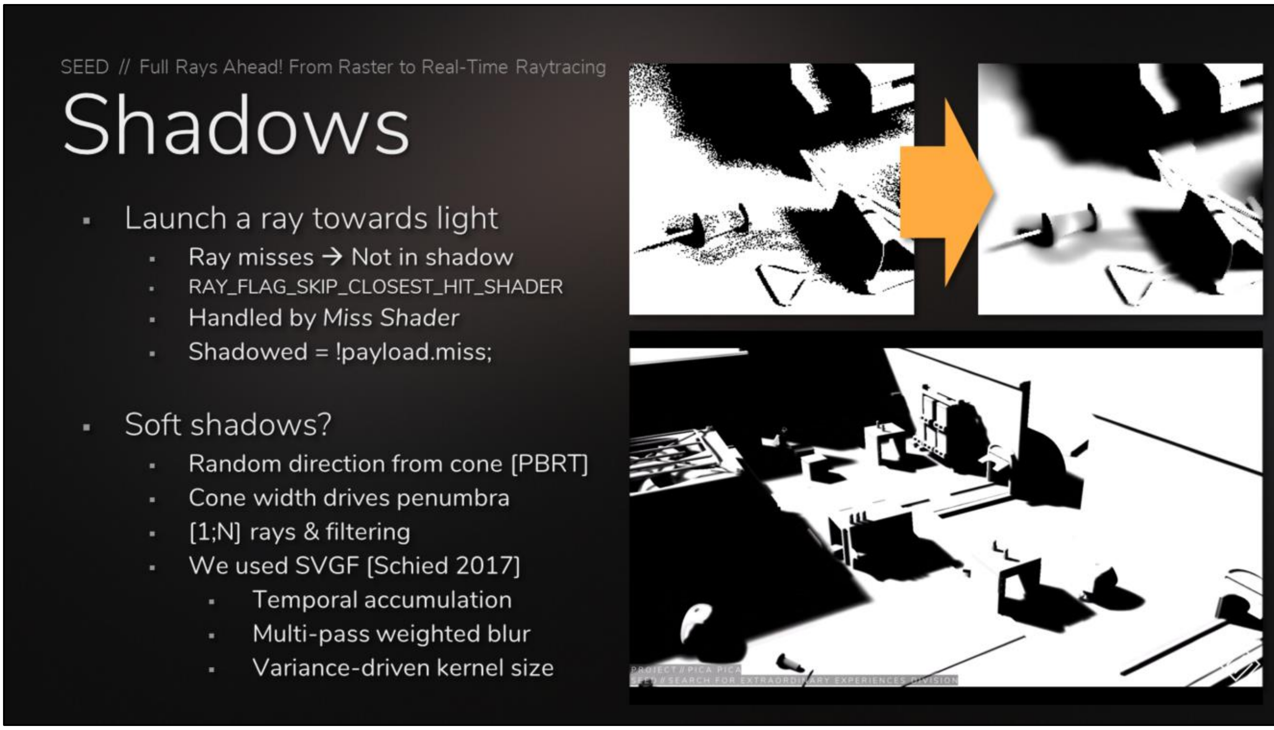

- To get there some stretching and warm-up might be required...



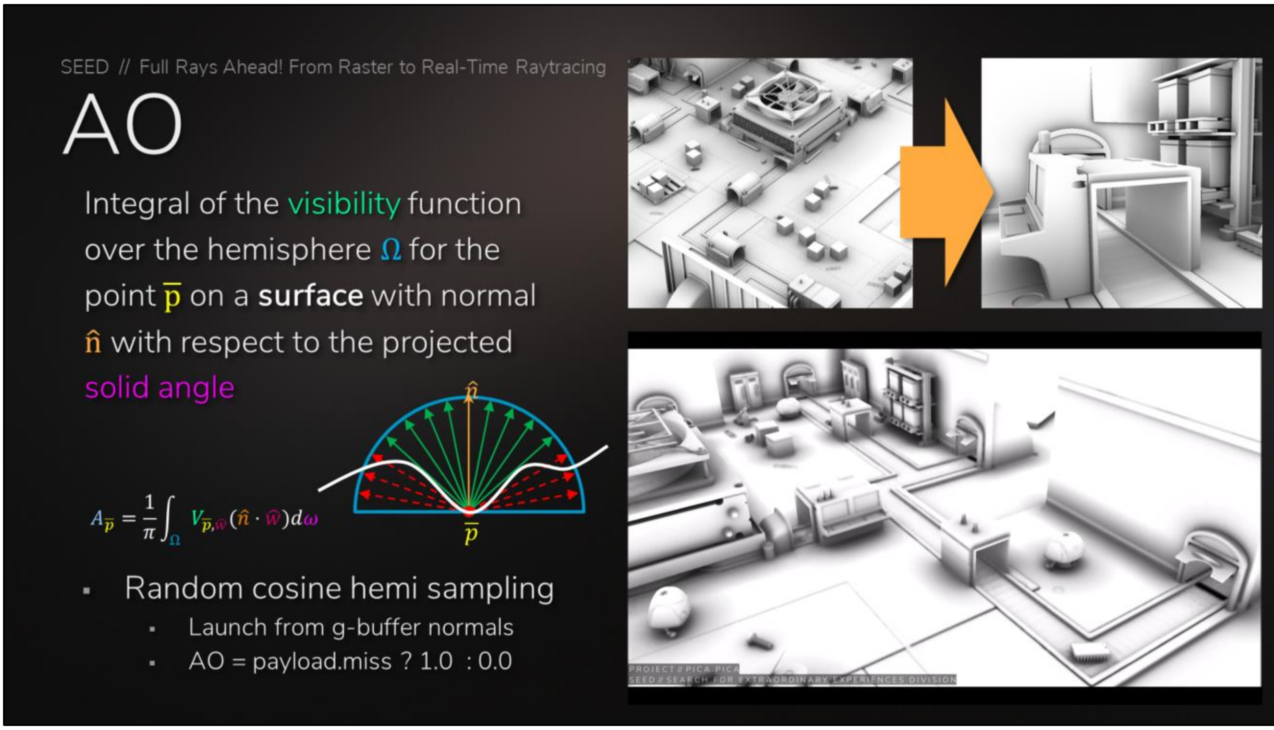

- First, brush up on raytracing and DXR. You can do this by using the resources provided by this course, but also the various open source frameworks that are out there, but also our GDC and Digital Dragon talks.
- **Additionally check out yesterday's presentation from Matt Pharr on adopting lessons from offline ray tracing to real-time ray tracing, from the advances in real-time rendering course. Pretty awesome stuff there about managing variance and noise!**
- Let's get practical. First you'll need to break down passes so that code can be swapped and re-used. And so with HLSL it's easy since D3D interops with raster, raytracing and compute. You should start by building shared functions that you can call from any stage, such as your material evaluation lighting code, various helpers
- One thing that will really help is making sure your engine can quickly toggle between techniques. This will be essential for comparing, cutting corners, optimizing and choosing ray counts.
- Preparing your passes for being able to swap inputs and outputs. For example, break your shadows into a full screen shadow mask, so that you can swap CSMs with RT shadows. The same idea applies for screen-space reflections and raytraced reflections.




- There are a few techniques you should implement first, that give you the most bang for the buck.
- In difficulty order that's shadows, ambient occlusion, then reflections.

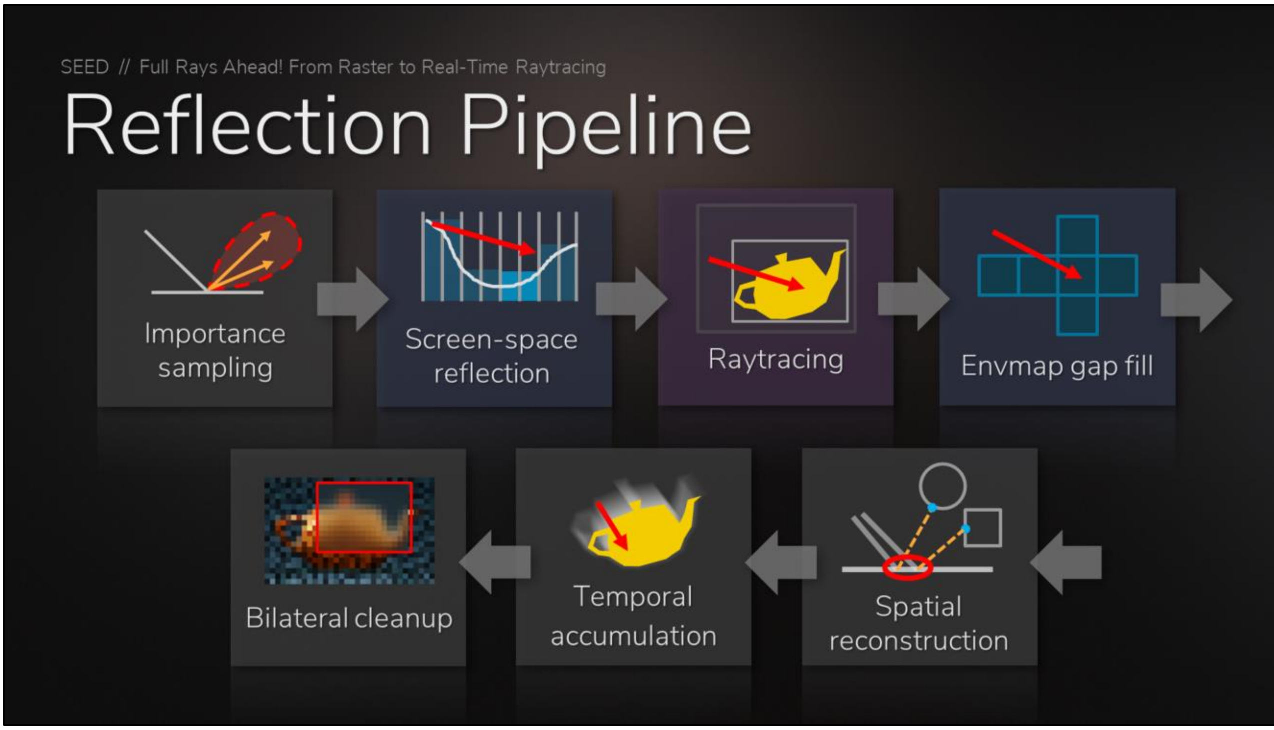



- First, raytraced shadows are great because they perfectly ground objects in the scene. This is not too complicated to implement. Just launch a ray towards the light, and if the ray misses you're not in shadow
- Hard shadows are great… but soft shadows are definitely better to convey scale and more representative of the real world
- This can be implemented by sampling random directions in a cone towards the light
- 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
- In our case, we first reproject results TAA-style, and accumulate shadow and variance
- We then apply an SVGF-like filter. We modified SVGF to make it more responsive for shadows. We coupled it with a color bounding box clamp similar to the one in temporal-antialiasing, with a single scalar value which is cheap to evaluate. We used Marco Salvi's variance-based bounding box method, with a 5x5 kernel estimating the standard deviation in the new frame. The result is not perfect, but the artifacts aren't noticeable with full shading.

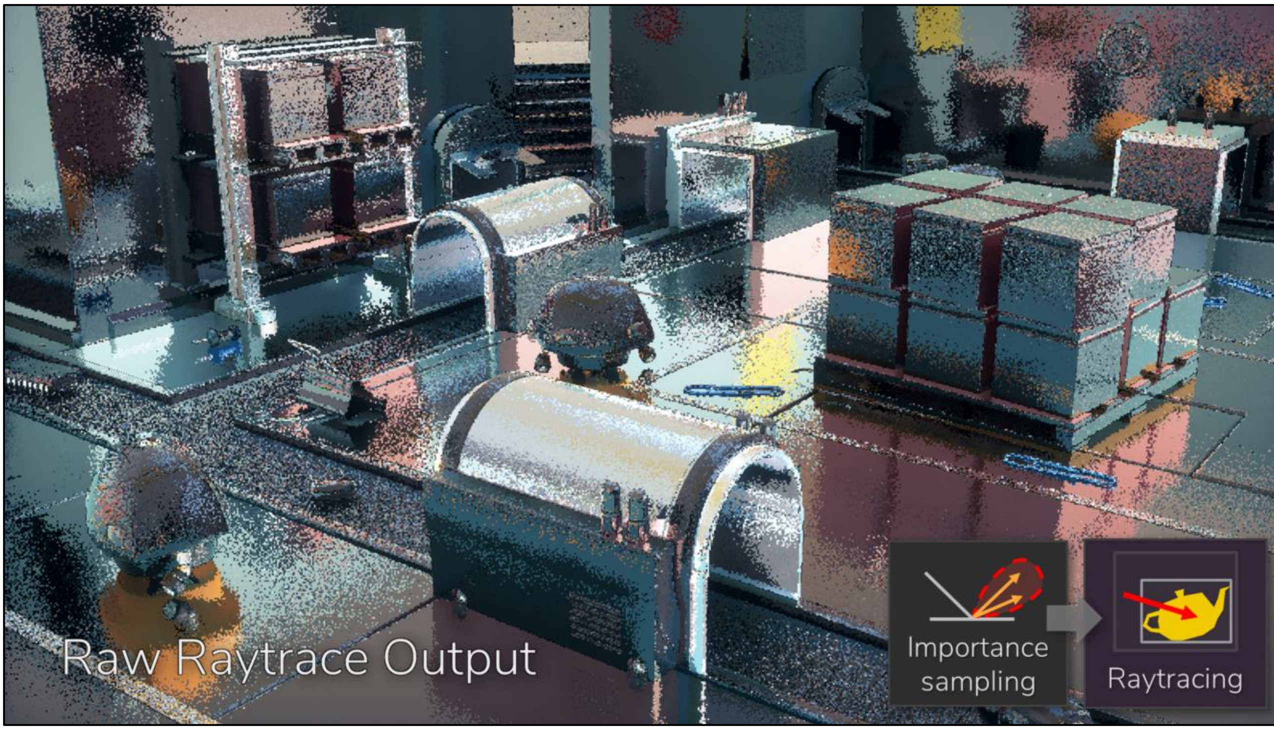



- Another technique that maps and scales well to real-time ray tracing is of course ambient occlusion, which we apply to indirect lighting.
- Being the integral of the visibility function over the hemisphere, we get more grounded results because all the random directions used during sampling actually end up in the scene, unlike with screen space techniques where rays can go outside the screen, or where the hitpoint is simply not visible.
- Just like in the literature, this is done by doing cosine hemispherical sampling around the normal.
- In our case we launch rays using the gbuffer normal, use the miss shader to figure out if we've hit something
- You can launch more than 1 ray per frame, but even with one ray per frame you should get some nice gradients after a few frames.
- We apply a similar filter as the one for the shadows

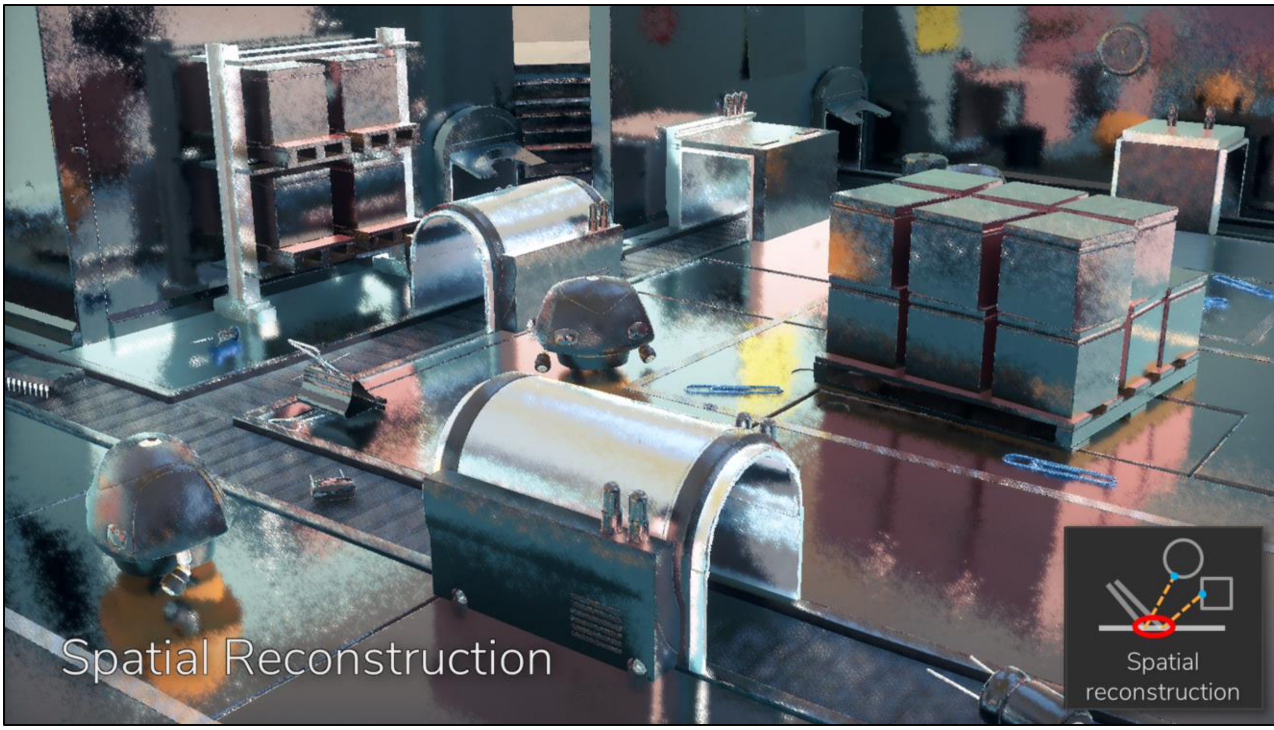

- Just like for AO, the ability to launch rays from the g-buffer also allows us to trace reflections.
- We raytrace them at half resolution, so for every four pixels you get one reflection ray. This gives us a quarter of a ray per pixel. Then at the hit point shadows will typically be sampled with another ray. This totals to half a ray per pixel.
- This works for hard reflections, but we also support arbitrary normal and varying roughness.
- We do this by applying spatiotemporal reconstruction and filtering
- Our first approach combined it with SSR for performance, but in the end we just raytraced for simplicity and uniformity.
- We have presented some extended info on how we do this, so check out our GDC 2018 and Digital Dragons talks




- Without going into too many details, which you can get from the talks, here's a high-level summary of the raytracing pipeline.
- We start by generating rays via BRDF importance sampling. This gives us rays that follow the properties of our materials.
- Scene intersection can then be done either by screen-space raymarching or raytracing. As I said in the end we only raytrace but support both.
- Once intersections have been found, we proceed to reconstruct the reflected image.
- Our kernel reuses ray hit information across pixels, upsampling the image to full-resolution.
- It also calculates useful information for a temporal accumulation pass.
- Finally we run a last-chance noise cleanup in the form of a cross-bilateral filter.




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

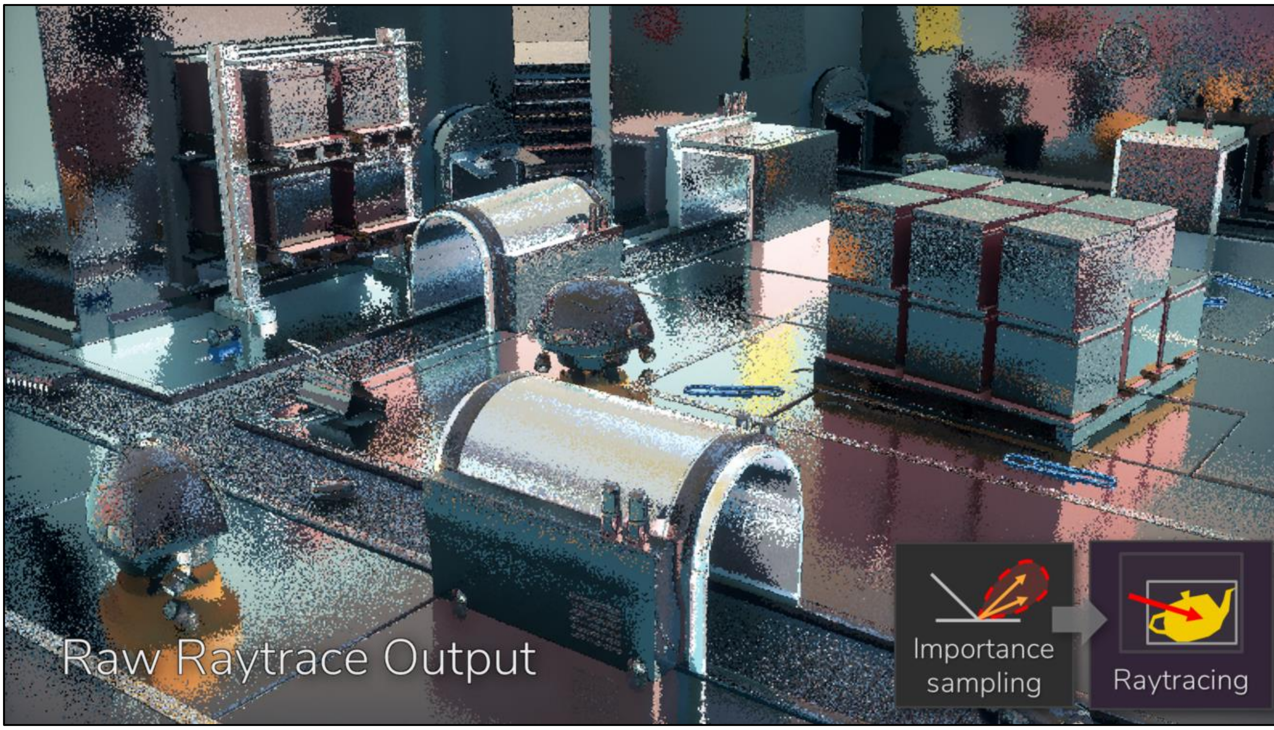



- This is after the spatial reconstruction

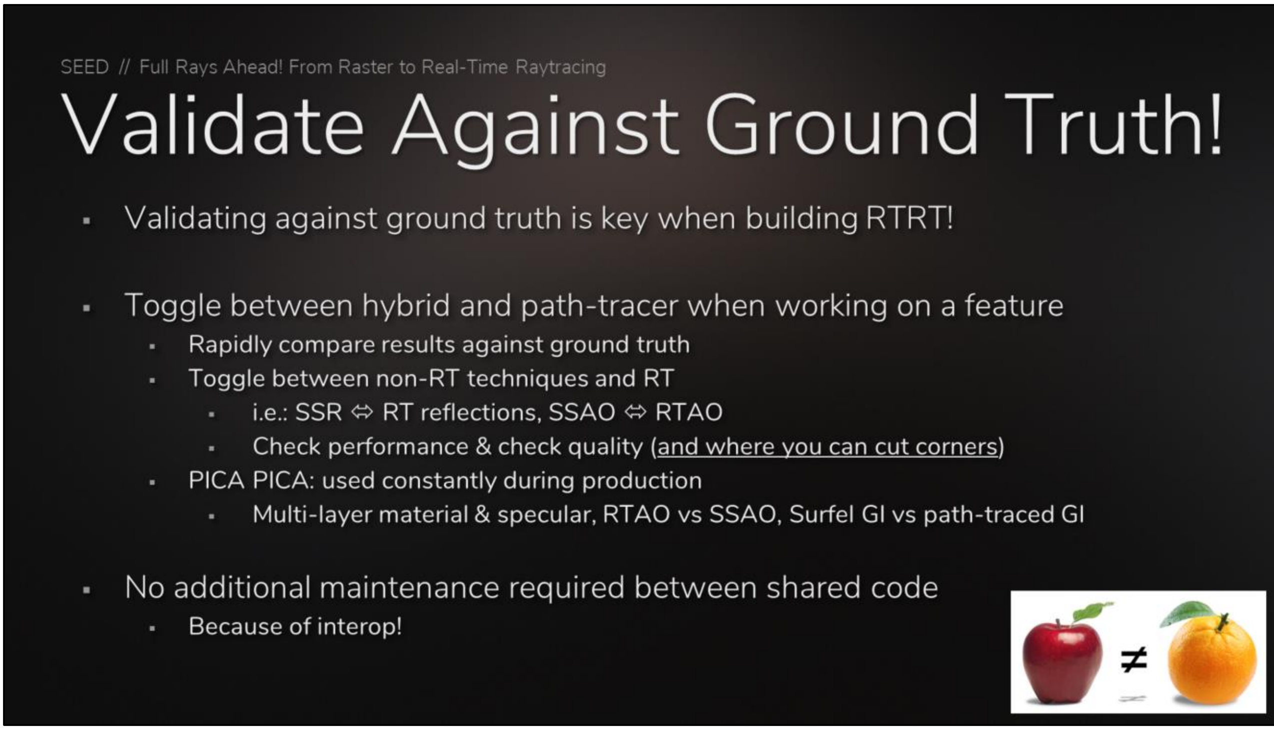

- Followed by temporal accumulation & bilateral to further clean up the image


- And finally by a much simpler bilateral filter that removes up some of the remaining noise.
- It overblurs a bit, but we needed it 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, we use a variance estimate from the spatial reconstruction pass and use it to scale down the bilateral kernel size and sample count


- With temporal anti-aliasing, we then 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

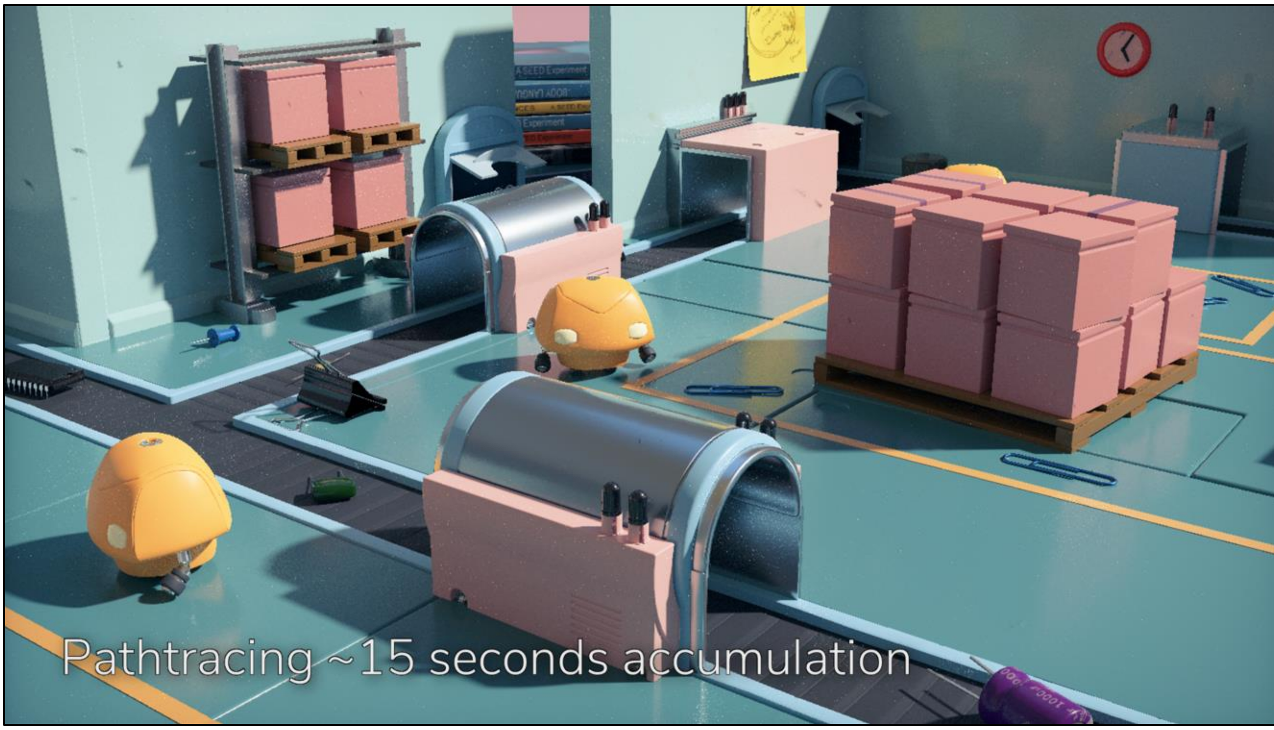



- Going back to the raw output, it’s quite a change!

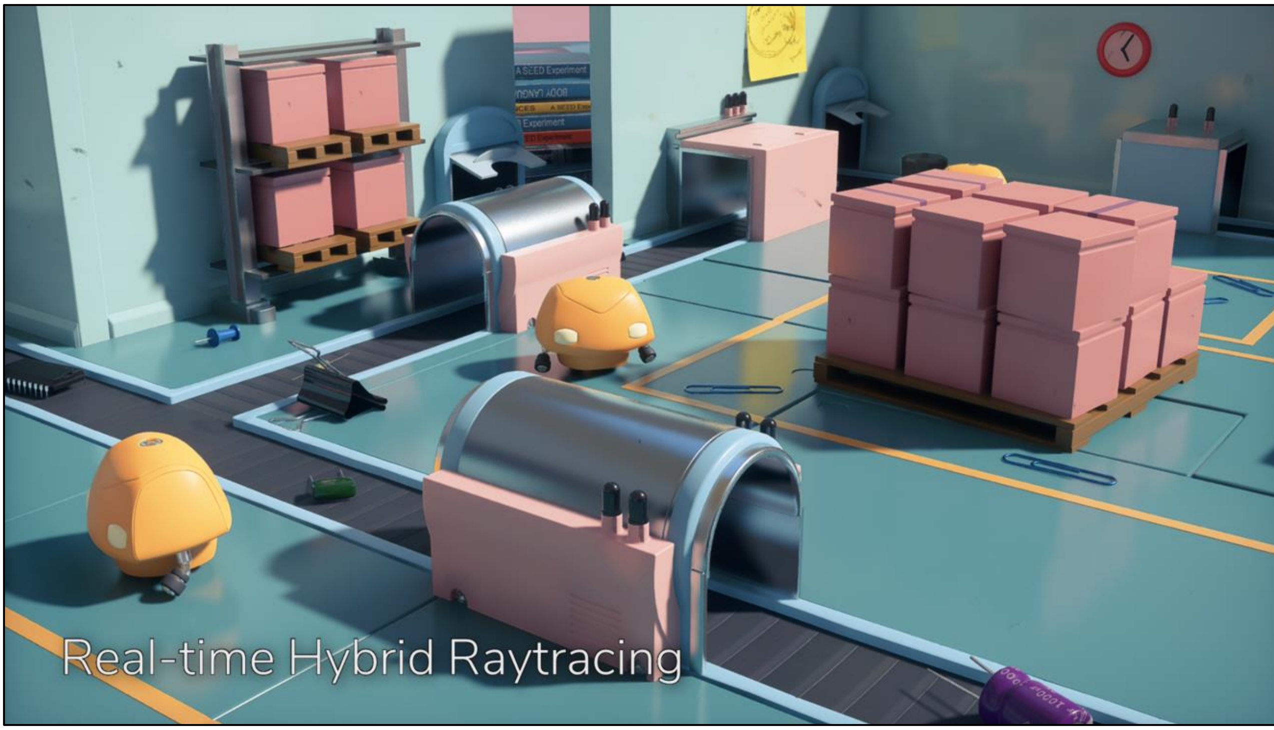



- In the case of reflections, and any other technique you want to implement with hybrid rendering, it's also key to compare against ground truth.
- Internally we've built a path-tracer that acts as our ground truth, which we can toggle when working on a feature.
- Actually we used this constantly during PICA PICA.
- Also because of interop there is minimal to almost no maintenance required, so having this ground truth comparison tool shouldn't add too much work.

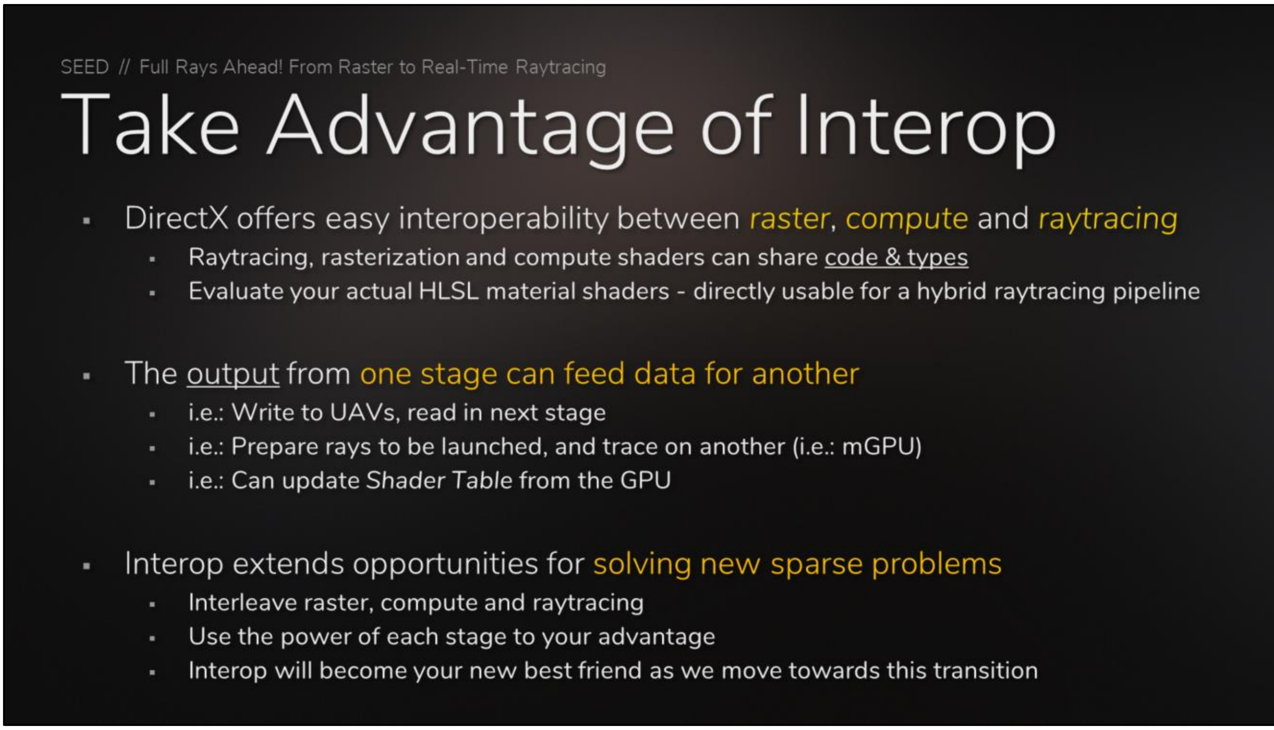
- Image: http://www.cristanwilliams.com/b/2012/01/10/apple-separatism/



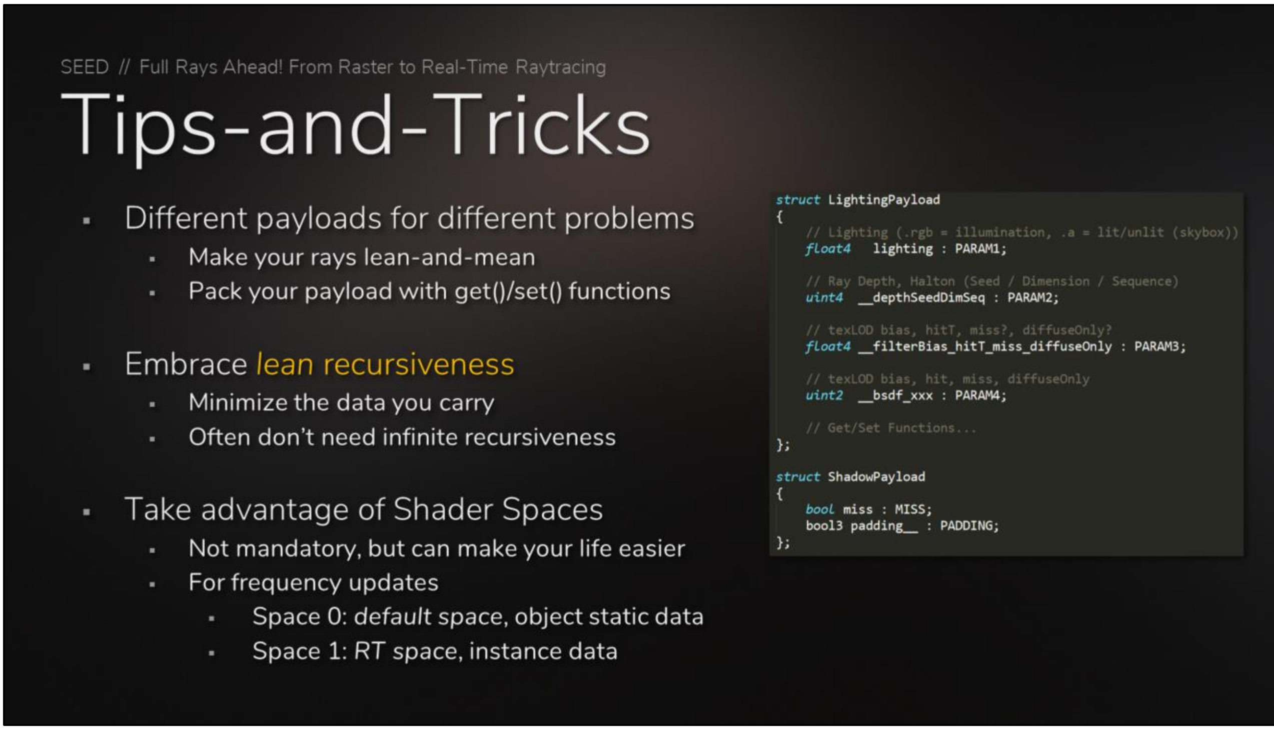

- This is what it looks like when we toggle the path tracer and wait 15 seconds. The image looks nice but has some noise because of difficult light paths and caustics.



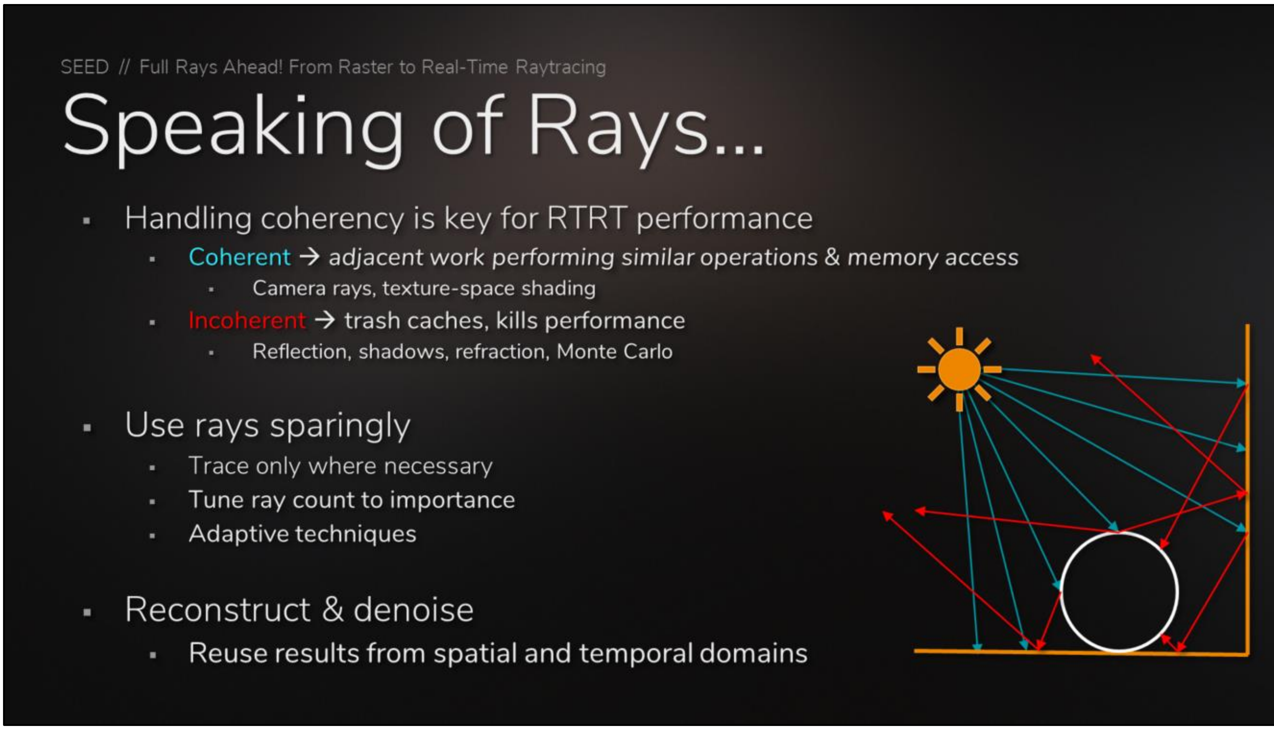

- And this is with out hybrid approach on the latest hardware at 60 FPS on a titan V.
- It’s not quite the same as the path traced version since it’s missing caustics and some small-scale interreflections, but still pretty decent.




- If we continue talking about practical things, make sure to take advantage of DirectX's interop, where you can easily share code, types and results between stages.
- For example you can evaluate your actual HLSL material shaders from the raytracing stage. Quite convenient. No conversions required.
- Actually you should be taking advantage of each stage's strength and use that as you build your hybrid pipeline.
- Interop allows for solving new sparse problems and with raytracing as a new tool in the bag, interop makes even more sense these days and will become your new best friend.




- Here are some additional practical tips and tricks when working with DXR.
- Since you'll be launching many rays, and as mentioned by Chris, the importance of minimizing what each ray carries can influence performance.
- Depending on what kind of rays you're launching, you can have different kinds of payloads, some leaner than others.
- Embrace lean recursiveness. Make your performance and IHVs happy by not having infinite recursiveness. You often don't need it.
- Finally you can also use shader spaces to organize your data. Makes it useful to keep things organized especially when things get updated at different frequencies.




- Speaking of rays, handling coherency is key for real-time raytracing 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
- And so use ray sparingly and trace only when necessary, tuning ray count to importance and adaptively. And set the max recursivity on the shader state object.
- The is no way right now in HLSL to specify if rays to be launched are expected to be coherent or incoherent. Providing a hint to IHVs could be good. Who knows maybe we'll be able to do this in the future, as the API keeps evolving.
- As you saw from our work, reconstruction and denoising allows for great results, so take advantage of spatial and temporal domains. Lots of great work in that field lately.

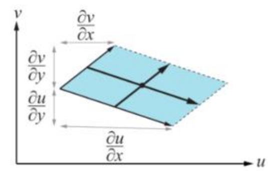
# Texture Level-of-Detail

What about texture level of detail?

- Mipmapping [Williams 1983] is the standard method to avoid texture aliasing:

$$\lambda(x, y) = \log_2 \lceil \rho(x, y) \rceil$$

$$\rho(x, y) = \max \left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2} \right\}$$




Left: level-of-detail (λ), partial derivatives and the parallelogram-approximated texture-space footprint of a pixel. Right: mipmap chain

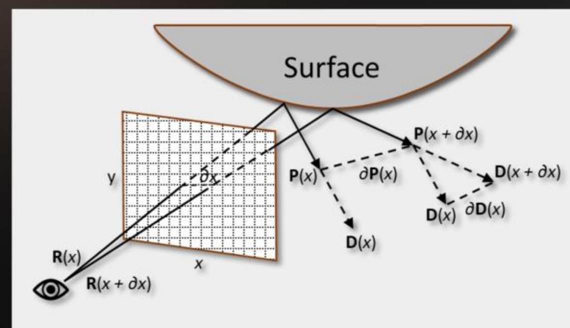
- Screen-space pixel maps to approximately one texel in the mipmap hierarchy
- Supported by all GPUs for rasterization via shading quad and derivatives

What about texture LOD?

# Texture Level-of-Detail

**No shading quads for ray tracing!**

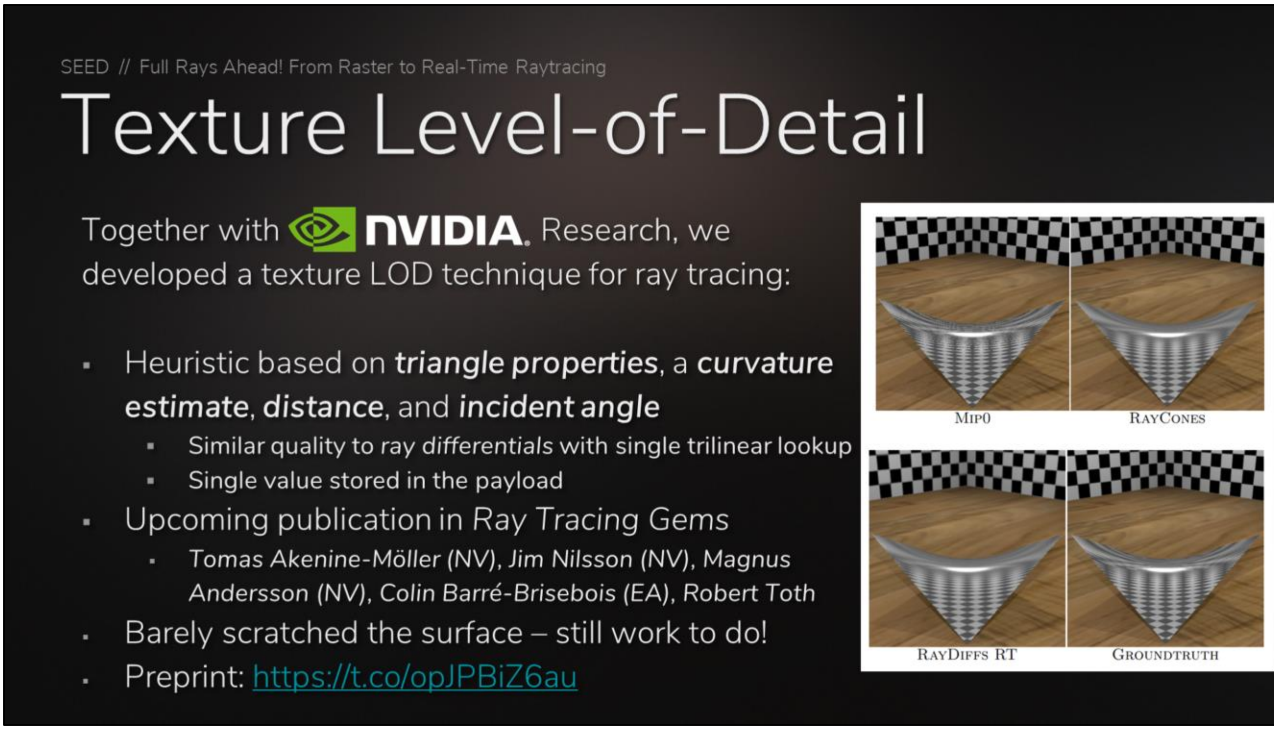
- Traditionally: *Ray Differentials*
  - Estimates the footprint of a pixel by computing world-space derivatives of the ray with respect to the image plane
  - Have to differentiate (virtual offset) rays
  - Heavier payload (12 floats) for subsequent rays (can) affect performance. Optimize!



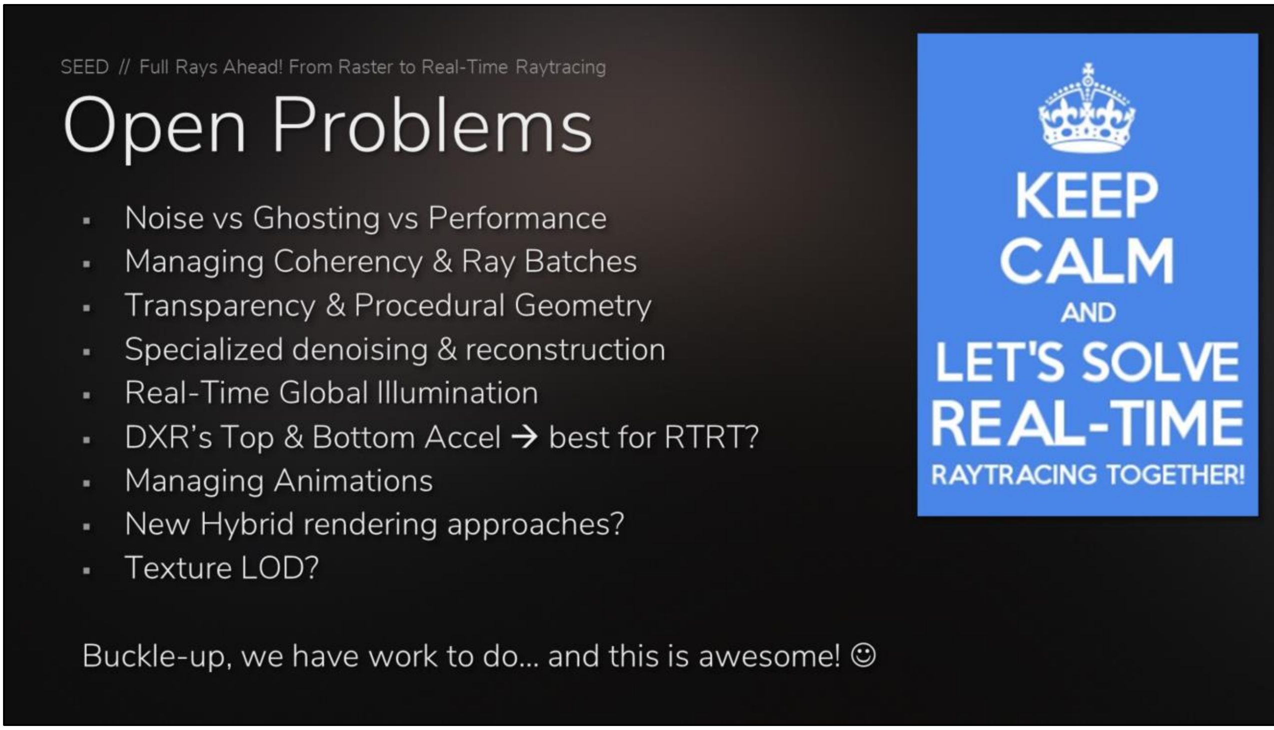
Ray Differentials [Igehy99]

- Alternative: always sample mip 0 with bilinear filtering (with extra samples)
  - Leads to aliasing and additional performance cost

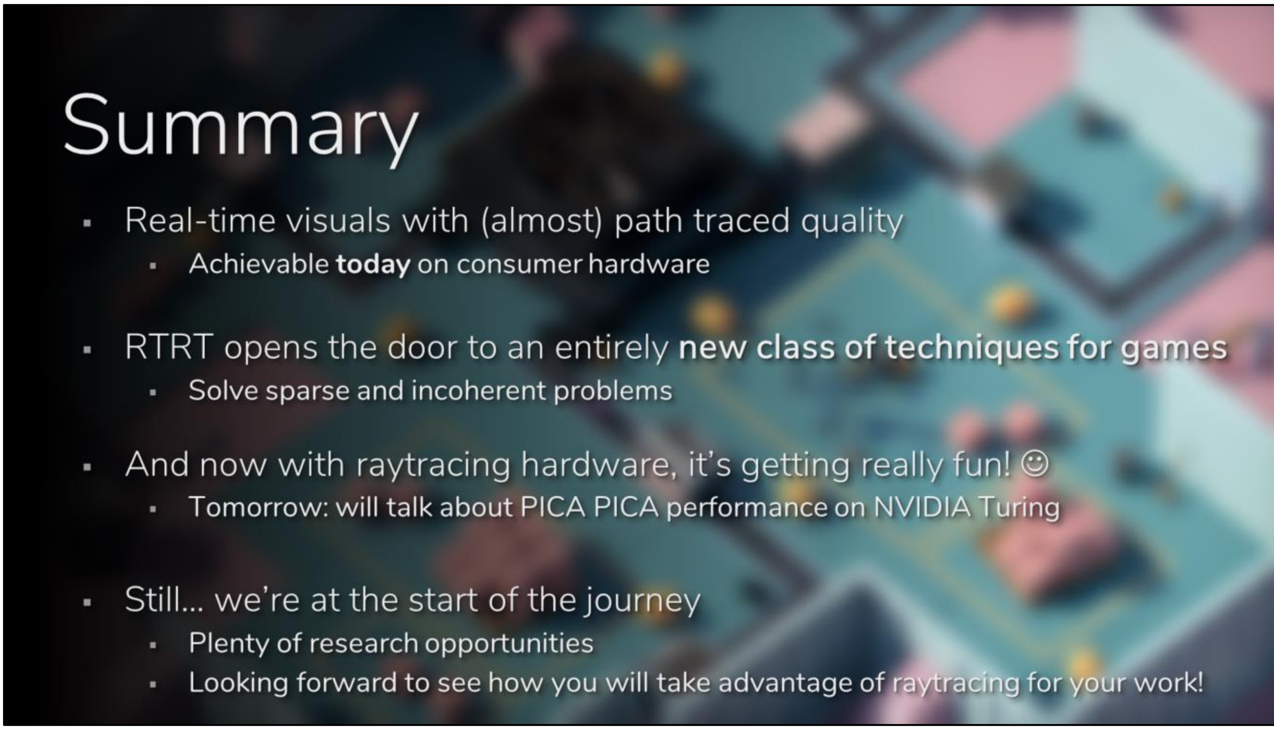
- As many of you are aware, we don’t have shading quads for ray tracing




- We've barely scratched the surface:
- Works great for materials where the texture sampling is very clear
- Need to look at anisotropy & dependent texture lookups in complex shader graphs
- Technique assumes static Uvs, so need to look at recomputing texture coordinates on the fly?
- For pica pica we used it but the difference is minimal because of the super clean visuals
- Even with detail normal on all objects, with a ton of TAA on top sampling mip0 was not noisy
- Does affect performance, so you don't want to only sample mip0
- A good start though, but still work to do here. Make sure to check out the pre-print

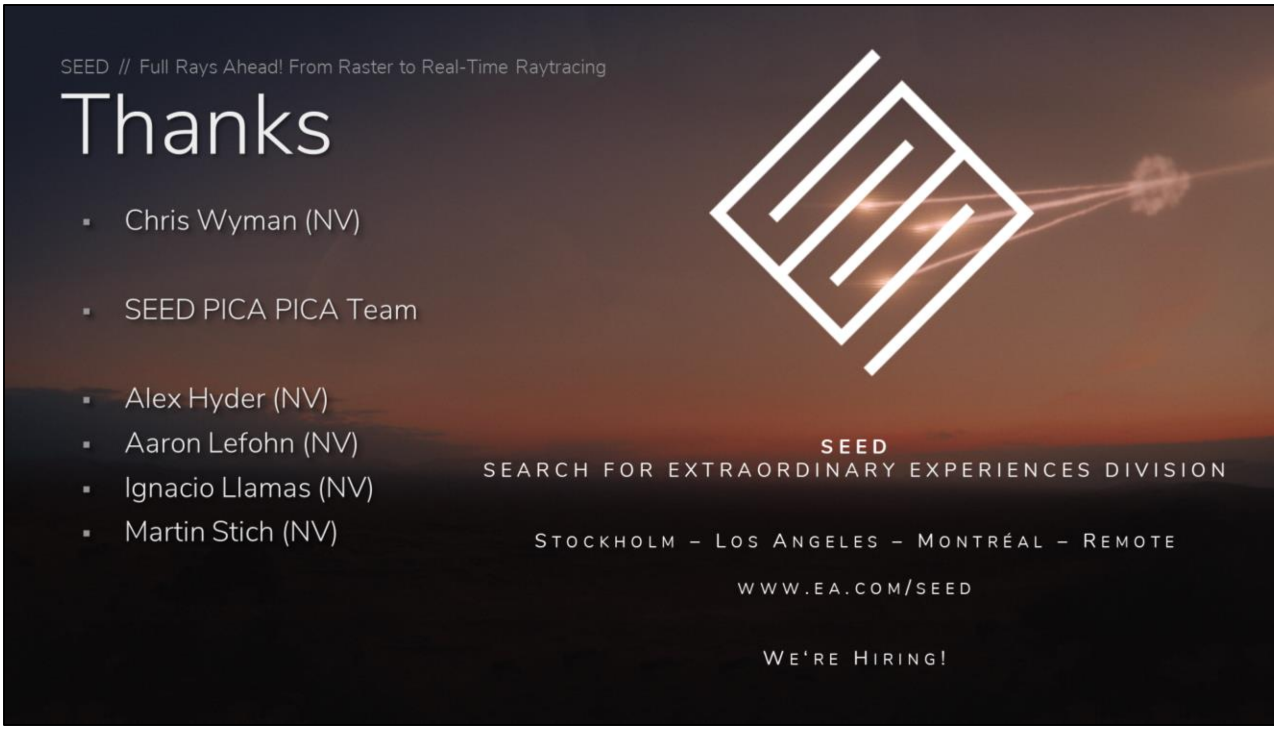



- And this was just a glimpse on some of the stuff you may have to tackle, and we haven't solved everything. Still lots of work to do!
- There are a bunch of open problems that we'll have to tackle together in the years to come, and this is awesome!
- Check out my HPG keynote this year where I go into detail about some of these topics
- Hopefully some great collaborations between offline raytracing experts and the game industry!




- To wrap things up, raytracing makes it possible to replace refined hacks with unified approaches, making it possible to finally phase out some screen space techniques that are artifact-prone.
- Out of the box, raytracing also enabled higher quality visuals without some of the painful artist time.
- There is no free lunch though, and considerable efforts need to be put into reconstruction and filtering.
- And now with raytracing hardware the possibilities are even more awesome! Tomorrow I will give some performance numbers on the new hardware, compared to TitanV. It's pretty impressive!
- But don't forget that raytracing is also just another tool in the box, and should be used wisely where it makes the most sense.
- I think it's very encouraging that we can begin to approach the quality of path tracing with just over two rays per pixel.
- Looking forward to what you come up with and how you can take advantage of hybrid rendering with raster, compute and raytracing


- Academia always asks for content from the games people and often doesn't get it.
- And so for SIGGRAPH we have decided to release all the assets from PICA PICA
- You can download them via Sketchfab and on our website, use them in your research for free, build your ray tracing pipeline and compare with ours.
- Challenge Accepted?


SEED // Full Rays Ahead! From Raster to Real-Time Raytracing
Thanks
Chris Wyman (NV)
SEED PICA PICA Team
Alex Hyder (NV)
Aaron Lefohn (NV)
Ignacio Llamas (NV)
Martin Stich (NV)
SEED
SEARCH FOR EXTRAORDINARY EXPERIENCES DIVISION
STOCKHOLM – LOS ANGELES – MONTRÉAL – REMOTE
WWW.EA.COM/SEED
WE'RE HIRING!


Questions?