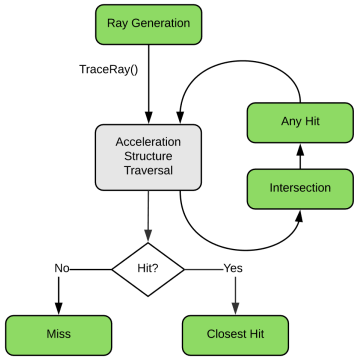
I'm Matt Pharr and today I'm going to talk about ray tracing.

One thing to make clear from the start: I'm not here to say that it's going to be the end of rasterization, and I'm not expecting you're going to start path tracing everything in your games starting tomorrow. I think ray tracing's a fantastic tool to have in the toolbox. I do think it makes sense to start tracing some rays, and in time, to trace some more. I think you'll love it.

This talk is about being smart about those rays and making the most of the GPU's ray-tracing capabilities.

DIRECTX RAY TRACING

- New addition to DX12, announced at GDC 2018
- Ray tracing is now a first-class member of the real-time graphics pipeline
  - Tight integration with raster and compute
  - Shared buffers, textures between raster and ray tracing
  - Trace rays from HLSL shaders



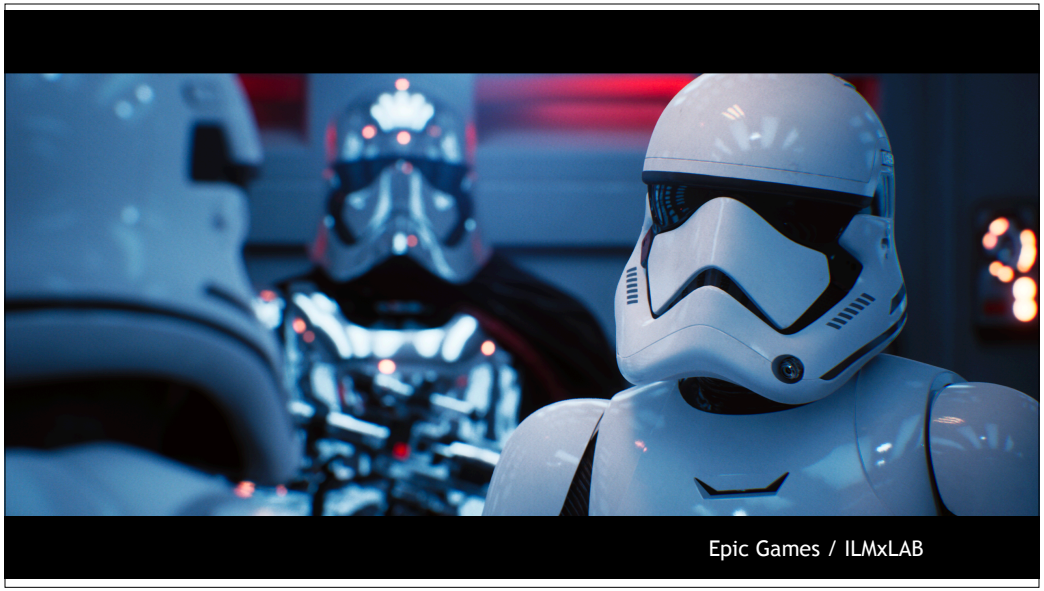
Real time ray tracing has arrived. [At the time the talk was given, Turing was half an hour away from being announced.]

At GDC, DXR was announced, making ray tracing a first-class component of the real-time graphics pipeline. We've got close integration with raster and compute, with shared buffers, textures, and shaders.

Of course, you could do real-time ray tracing on GPUs before, but the interop with raster was fiddly, and writing a high-performance ray tracer is really hard. Efficiently building acceleration structures on a massively parallel processor like a GPU is a real challenge, as you know if you've read some of those papers.

DXR lets the IHVs handle a lot of that for you and gives you the capabilities cleanly through DirectX. [And also lets them accelerate ray tracing with hardware. :-)]

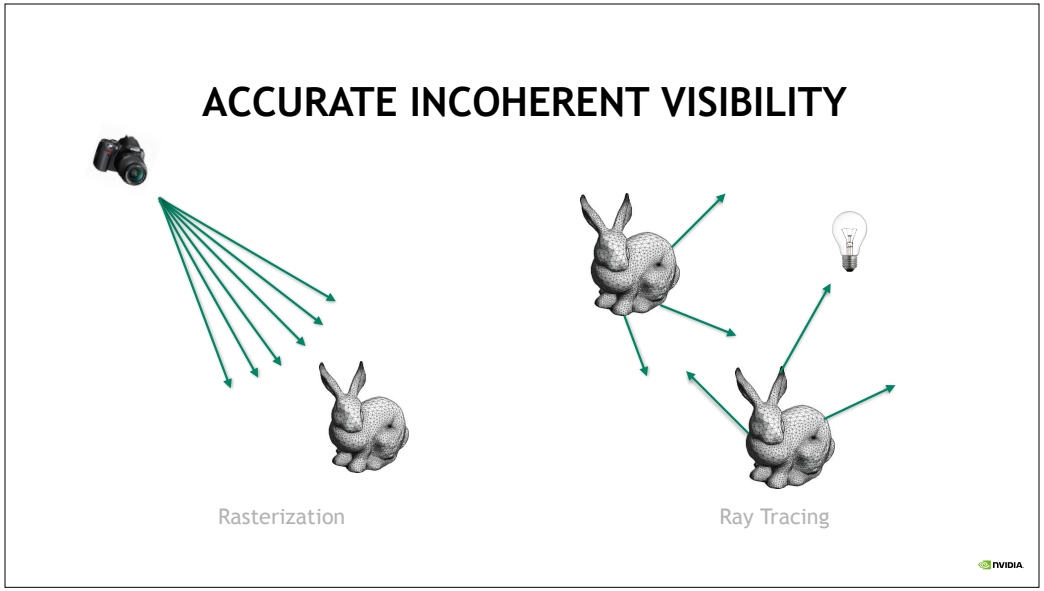
You can just focus on deciding which rays to trace and what to use them for.



People have shown amazing stuff with it already. Lighting and shading effects that are extremely difficult if not impossible to do with rasterizers. This is an amazing image from Epic and ILMxLAB, shown running in real-time at GDC. The reflections are incredible; doing them this well with a rasterizer only is hard to imagine. It looks like offline rendered content.

Here’s another example that I’m sure everyone’s also seen: PICA PICA from EA’s SEED group.

Colin Barré-Brisebois gave a great talk at HPG this year about PICA PICA; definitely check out his slides if you didn’t see the talk. There’s lots of great information there. They’re doing ray tracing, clever sampling and denoising, and it looks utterly beautiful. You’re not used to seeing things that look like this in real-time rendering.



Let’s boil down what this change means and think about it purely in terms of visibility algorithms. Ignore global illumination and stuff like that for now and just think about the geometric problem of determining visibility.

Rasterization gives you highly efficient visibility, from:

- a (homogeneous) point

- In regularly distributed directions

- In many of directions at once (pixels)

If you don’t have all three of those in the visibility you’re computing, you’re lost.

With ray tracing, now we’ve got the ability to do incoherent arbitrary visibility tests.

That’s the most exciting change in real-time graphics since programmable shading. If you think about the history of it all, there hasn’t been any choice in visibility algorithms for real-time rendering since storing a per-pixel depth and z-buffering first became possible (in the 70s, that was!)

INFLUENCE OF RASTERIZATION-ONLY

- Can I see the light?
  - PCF shadow maps, stencil shadows, perspective shadow maps, cascaded shadow maps, SDSMs, AVSMs, variance shadow maps, exponential shadow maps, exponential variance shadow maps, light probes, light maps, …
- What's around me?
  - Environment maps, reflection maps, depth peeling, screen-space visibility (SSAO, etc.), sparse voxel octrees, stochastic transparency, billboard clouds, …
- VR distortion correction passes

NVIDIA

Only having a rasterizer for the visibility has deeply influenced the algorithms people use in real-time rendering..

That constraint has been a source of a ton of brilliant work and it's gotten us to where we are now, but it's been fighting against the limitations of a rasterization-only graphics pipeline.

It's not easy to choose the right algorithm—it's scene and asset specific. So many great things have been invented, but they've been fighting against the limitations of only having a rasterizer for visibility. Now we can look beyond all that.

To be clear, I'm not suggesting that all this stuff–or even rasterization–will go away, but now it's not the *only* choice, and that's fantastic.

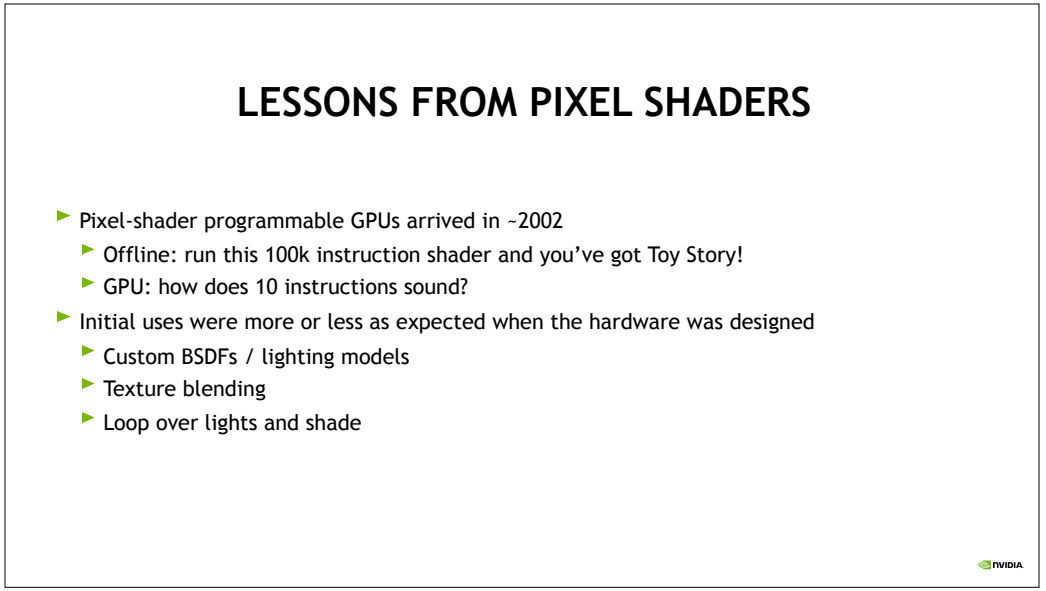
HAVING ALL OF PHYSICALLY-BASED RENDERING

- PBR materials have hugely improved the game development pipeline over the past n years
  - Accurate simulation of light scattering from surfaces
- The simulation of light arriving at materials has been, necessarily, incomplete
  - In reality, everything exchanges light
  - Visibility is only coherent to the camera and (maybe) to the lights; the rest is highly incoherent
  - Hard to do with rasterization only (much good work in this area notwithstanding)

NVIDIA

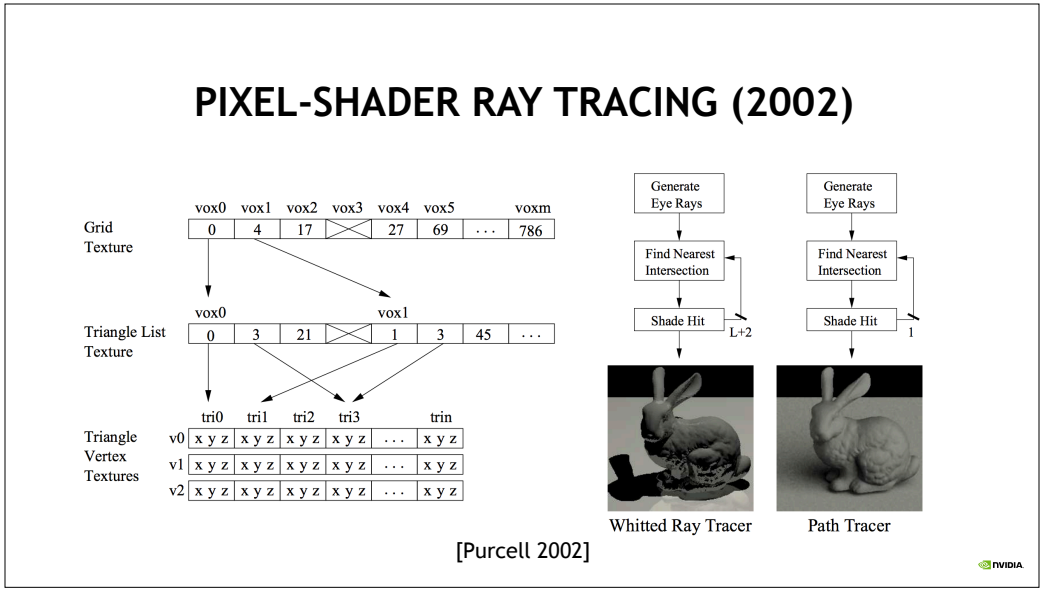
Physically-based materials have been only half of what's needed for realistic real-time rendering. Those have had a huge impact, but the problem is that at each point you're shading, you need to know the distribution of light arriving there. The visibility queries you want to do to do that are highly incoherent—not a good fit for a rasterizer.

Here also, there's been tons of clever work trying to make the most of the rasterizer, but now we can do something that lets us accurately figure that out—trace rays. That's why the SEED and Epic demos looked so great.



So we've got GPU ray tracing now, and that's great. But it's not like you can trace thousands of rays per pixel. We've been here before, though; 15 or so years ago, pixel programmability arrived to GPUs. You could look to offline rendering, which offered amazing images if you could run massive shaders, but GPUs couldn't do anything near that at the time.

At first, people used GPU programmability in pretty straightforward ways—the obvious next steps from register combiners and multipass texturing. Imagery improved, and that was great.



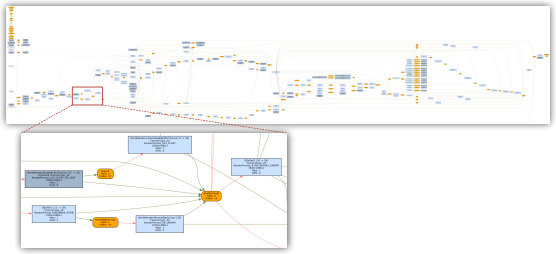
But then it got really interesting really quickly. It wasn't a matter of sitting around and waiting for GPUs to be able to run offline shaders, but instead people started finding really creative things to do with the capabilities that were available to them.

I believe that the first instance of the idea of "draw a full screen quad and do some other computation in a pixel shader" dates from this paper by Tim Purcell and collaborators. As it turns out, they were doing ray tracing. They encoded the scene and the acceleration structures in textures and used dependent texture lookups to traverse them. It's all straightforward stuff now, but it was a crazy new way of thinking at the time.

And it wasn't easy: there limited control flow on the GPU, so they had to do things like issue a draw per loop iteration and use occlusion query to figure out when to stop. But GPUs got better and more flexible really quickly.

RAPID DIVERGENCE

- Programmable shading -> programmable graphics
- Compute-shader driven rendering
- Led to GPU compute and CUDA…



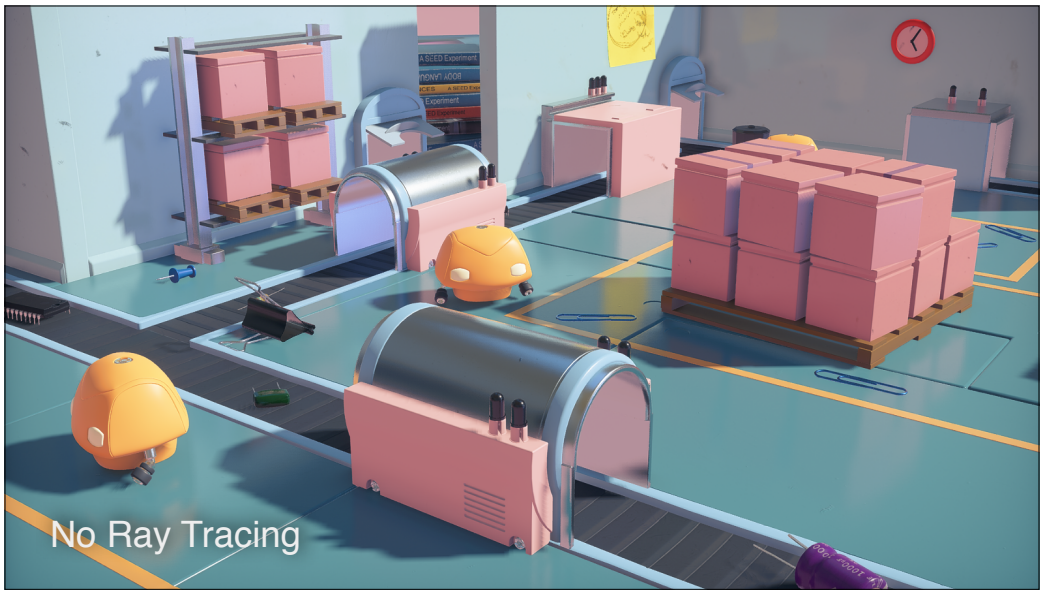
Frostbite FrameGraph [O'Donnell 2017]

NVIDIA

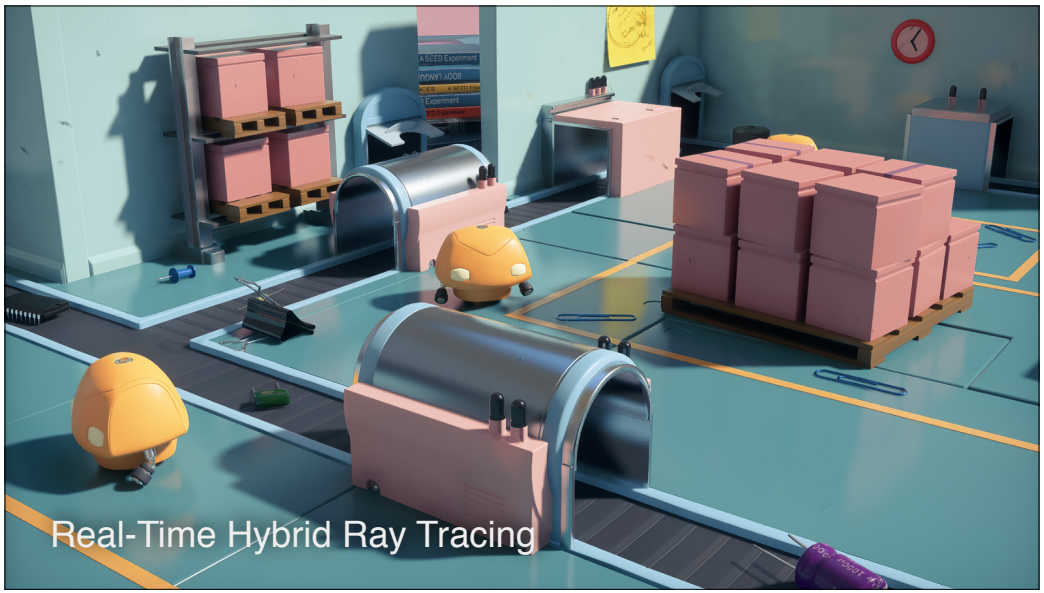
Soon enough, people were both creating data structures and traversing them in shaders.

We quickly saw close interop between compute and graphics; increasingly real-time rendering became a matter of renderer-driven data processing on the GPU—less and less about the traditional stages of the graphics pipeline

The lesson is that it's hard to predict, but amazing things start happening when the world's clever graphics programmers can put their hands on a new tool and start figuring out how to use it. And that's how I think it's going to go with real-time ray tracing.

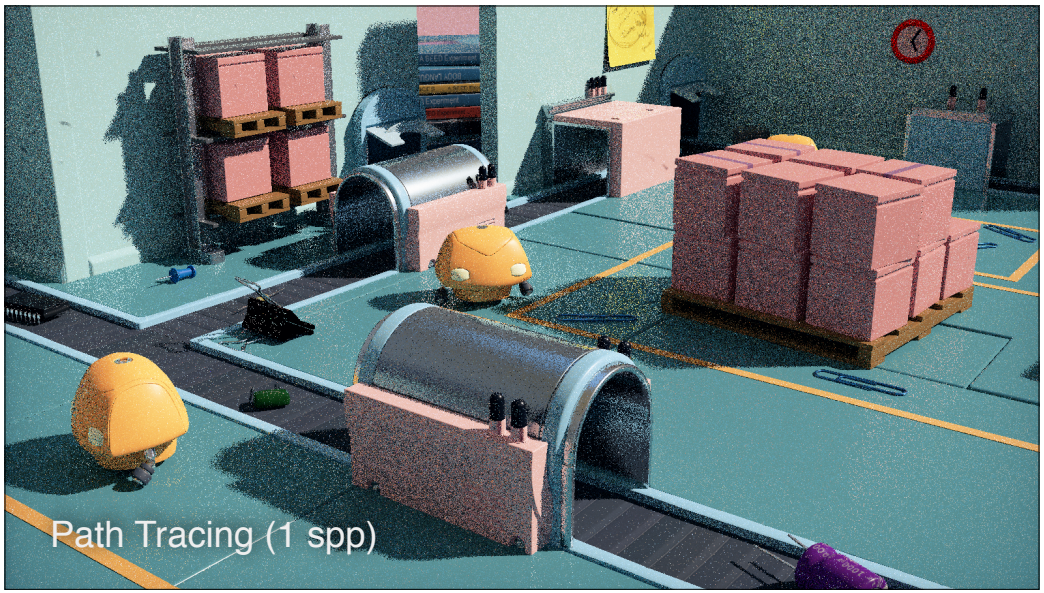


It’s already starting to happen. Here are some more examples from the PICA PICA demo. Here’s the scene rendered with rasterization. Nice enough, but not amazing.

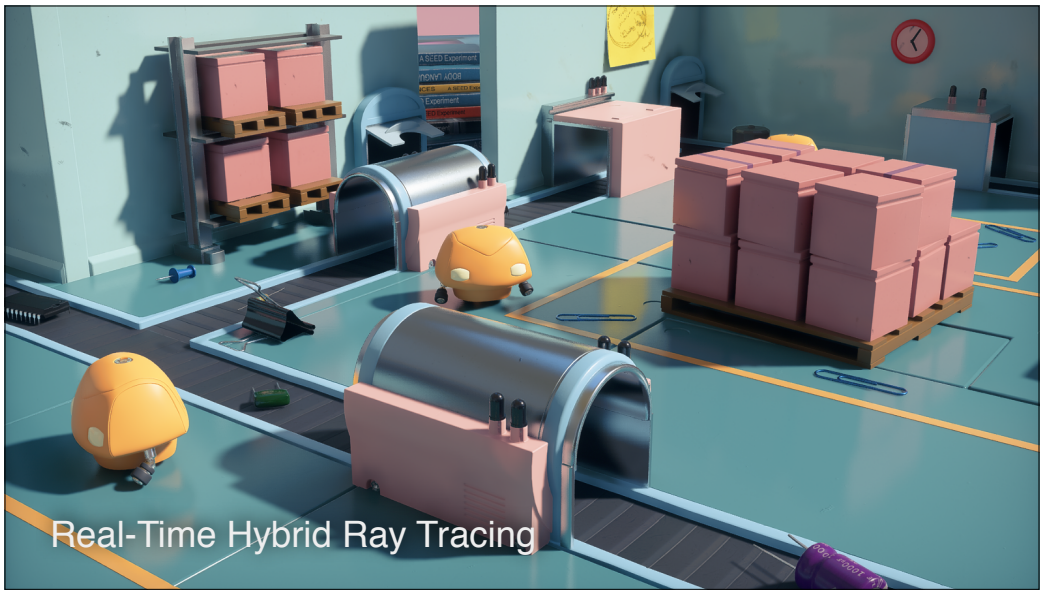


With a hybrid approach, they got this at 60 fps on a TitanV. And on an RTX GPU with dedicated ray tracing hardware, it’s about 6x faster.

There’s glossy and indirect reflection and accurate AO. Even those paperclip shadows help ground them. It all reads “reality” in a way that the other one didn’t. And that’s thanks to accurate visibility computations from ray tracing



They didn’t trace a ton of rays to do it, either. They laid down a g-buffer for primary visibility and then traced about one ray per pixel. Here’s what that gives you, unfiltered—it’s pretty noisy.



Given that, they did a lot of work filtering samples to eliminate noise—both neighboring pixels and reprojected pixels from previous frames is a big part of making this work. I'm not going to go into that today, but it's a key part of making it practical.

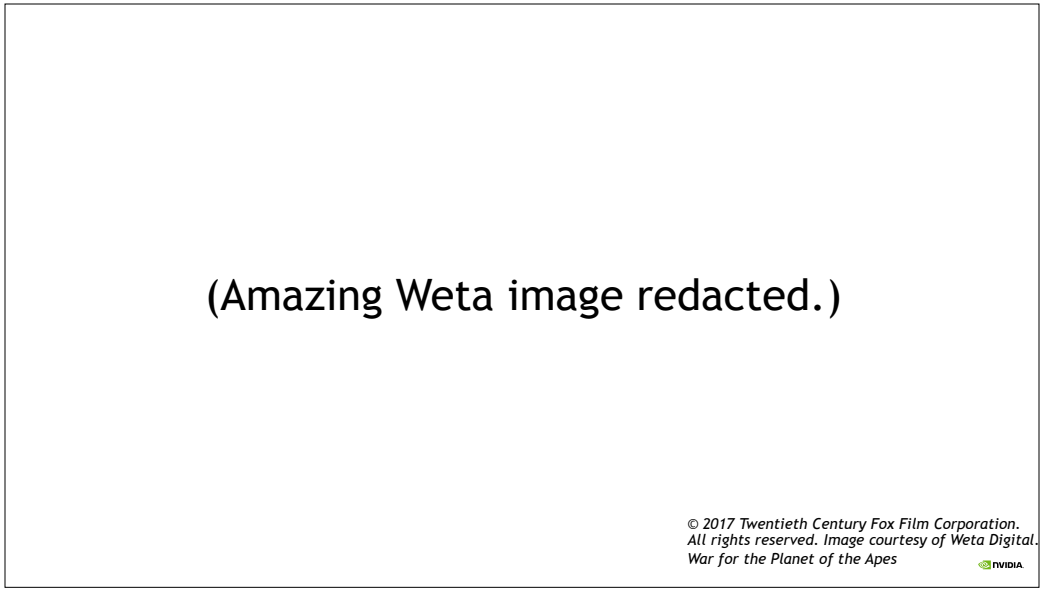
OFFLINE RENDERING'S EXPERIENCE

- ~10 years ago, multi-pass rasterization hit its breaking point
  - Long iterations for artists, unwieldy workflows, rendering artifacts from visibility approximations
  - Prebaking and caching illumination usually works… Until it doesn't (╯°□°)╯ ┻━┻
  - Couldn't simulate light transport as accurately as desired
- Adopted path tracing: unified light transport algorithm that handles everything
  - Primitives: surfaces, hair, volumetrics, …
  - Reflection: all types of BSDFs, BSSRDFs, …
  - Lights: point lights, area lights, environment map lights, …

NVIDIA

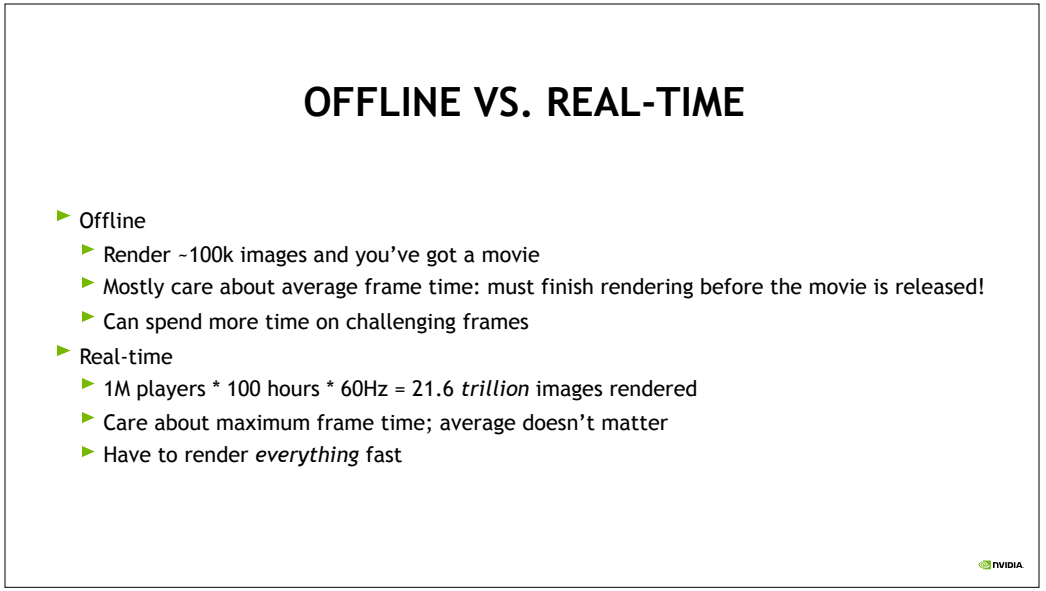
Ten years ago, offline rendering was in a similar place to real-time rendering today: rasterization was their workhorse visibility algorithm, but they really wanted more accurate lighting, which meant more accurate visibility. They had all sorts of precomputation and techniques based on rasterization, but it was really unwieldy. Tons of processing time before rendering really got started, and a long iteration cycle for artists.

Eventually, it became too much, and they made the transition to path tracing. It was painful, but really wonderful once they got there—they got rid of so many special cases: things like a volumetric effect shadowing hair just work now that it's all path traced, while before it could well be that the volumetric shadows algorithm wouldn't support casting shadows on hair—they just lived in totally different worlds.



And this is where they are today. Here's the ape Caesar from *Planet of the Apes 3*. It's a gorgeous image, fully ray traced, and it's insanely beautiful. If you've got amazing artists, a good ray tracer, and lots of time, the sky's the limit.

But this took hours to render, and 10s to 100s of thousands of rays per pixel. We're nowhere near being able to do that in real time today. I don't think this is going to happen in real-time tomorrow, but if you can make it happen soon, you're awesome.



To be completely clear, there are huge differences between the two types of rendering. To be honest, they're almost completely different realms.
These differences deeply influence the art and production pipelines.
e.g. if an asset is over modeled for film, meh, don't worry about it, probably not worth the human time to fix versus slightly longer renders
So even with ray tracing as the future for real-time, I believe that there will still never be complete convergence between the two areas.

OFFLINE RAY TRACING LESSONS ARE RELEVANT

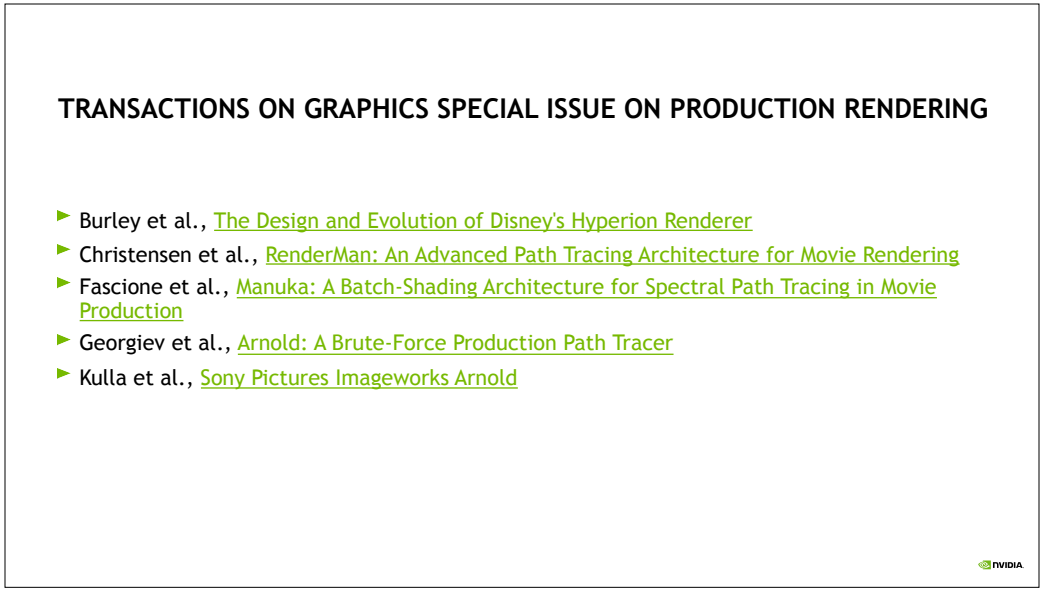
- Needed substantial performance improvements
  - Some of it was faster ray intersections
    - Lots of great work there, much of it applicable to real-time ray tracing
  - Most of it was about tracing the right rays
    - (And not tracing the wrong rays)
    - Enormous amounts of subtlety there; the focus of the rest of the talk

NVIDIA

So why are we talking about offline rendering in the real-time rendering course?

Offline figured a lot of stuff out that's applicable to real-time; they had to figure out a lot of things related to efficiency and tracing the right rays. That stuff is completely applicable to real-time and it's important to know.

We're going to see that it's worthwhile to do a decent amount of work to figure out which rays to trace in the first place. Being smart about that is even more important if you can't just trace a whole bunch of rays until you're happy with the result, as is the case with real-time.



Speaking of the offline rendering folks, the latest issue of ACM Transactions on Graphics is a special issue on production rendering, featuring 5 papers from the developers of today's most widely-used film renderers. There's almost 100 pages of information about implementing offline renderers, explaining the constraints they face, the problems they have to solve, and the approaches they've taken.

They're all doing ray tracing—path tracing—but beyond that, there's a great variety in design decisions. There's a lot of experience distilled there, well worth the time to read through.

LEARNING A FEW LESSONS FROM OFFLINE

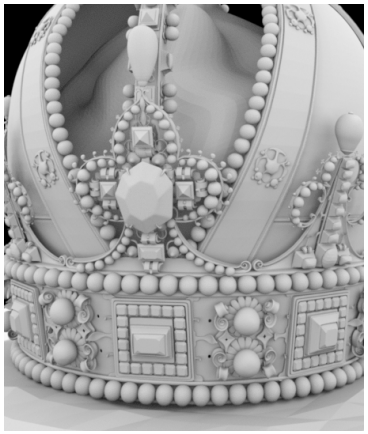
- Assumption: using path tracing (broadly defined) for some part of rendering
  - Not everything, not necessarily everywhere
  - …and then giving it to a denoiser
- Four topics for today:
  - Choose your rays wisely (and why you must)
  - Generate your (not)-random numbers carefully
  - Spend your ray budget where it's most useful
  - Understand and defend against error

NVIDIA

Again, the assumption here is that you're going to trace rays for some things, some of the time. The rasterizer is still a great thing, and worth using when it works well. For those rays, the rest of the talk will be about making the most of them and understanding some of the pitfalls that can lead to really high error if you're not careful about them.


CHOOSE YOUR RAYS
WISELY
21

MOTIVATING (SIMPLE) EXAMPLE: AO



NVIDIA

I'm going to start out using ambient occlusion as an example to motivate these ideas. Many of you have implemented the algorithm already, for example via SSAO, so some of this should be familiar ground. If you're an AO expert, the next few slides may be very familiar, but stick with me and we'll go some interesting places.

MONTE CARLO INTEGRATION 101

Basic estimator $$E\left[\frac{1}{N}\sum_{i=1}^{N} f(X_i)\right] = \int_{[0,1]^n} f(x)\,\mathrm{d}x \qquad X_i \in [0,1]^n$$

NVIDIA

Monte Carlo integration is a nice way to integrate functions that's particularly useful for rendering. Here's the definition of what's called the basic Monte Carlo estimator. It says that you can compute an estimate of the value of an integral basically just by picking some random points X_i in the domain of the function, evaluating the function there, and then averaging the function values.

It's applicable to integrating over any number of dimensions, which is nice, and the number of points you evaluate the function at can be arbitrary—in particular, it doesn't grow exponentially with the number of dimensions like many other numerical integration methods.

Another nice thing about it is that you only need to be able to evaluate the integrand at individual points; you don't need any more information about the function's behavior—bounds or derivatives or whatever.

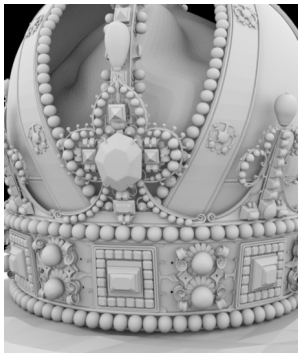
Monte Carlo integration also a natural fit for ray tracing, which tells you what's going on with the lighting, or whatever you're integrating, at a single point.

Note that Monte Carlo *on average* gives you the value of the integral. Sometimes it'll be too high, sometimes it'll be too low. In images, that error is manifested as noise.

MOTIVATING (SIMPLE) EXAMPLE: AO

$$\frac{1}{\pi}\int_{\Omega} V(\omega)\,\cos\theta\,\mathrm{d}\omega \approx \frac{1}{N\pi}\sum_{i}^{N} V(\omega_i)\,\cos\theta_i$$

```
float ao(float3 p, float3 n, int count) {
  float sum = 0;
  for (int i = 0; i < count; ++i) {
     float3 dir = sample_dir(n, i);
     if (unoccluded(p, dir))
       sum += dot(n, dir);
  }
  return sum / (Pi * count);
}
```



NVIDIA

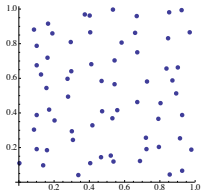
Ok, so let's apply the Monte Carlo estimator to ambient occlusion. We've got AO expressed as an integral on the left, where the 1/pi term normalizes the result to be in the range from 0 to 1, and we're just saying that we can compute AO by checking the visibility in a bunch of directions, weighting the visibility by the cosine theta term for that direction, and averaging.

The code below implements the idea, where we assume that sample_dir() returns a random direction over the hemisphere and unoccluded() determines if there's a blocker in the given direction. You could use a screen-space test there in place of ray tracing, of course.

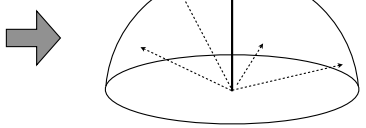
Note that if you've implemented SSAO you know this way of choosing directions not the best choice and we'll get to that in a second, but I'm going to explain exactly why that is.

UNIFORM HEMISPHERE SAMPLING

$$(x, y, z) = \left(\sqrt{1-\xi_1^2}\cos(2\pi\xi_2), \sqrt{1-\xi_1^2}\sin(2\pi\xi_2), \xi_1\right)$$



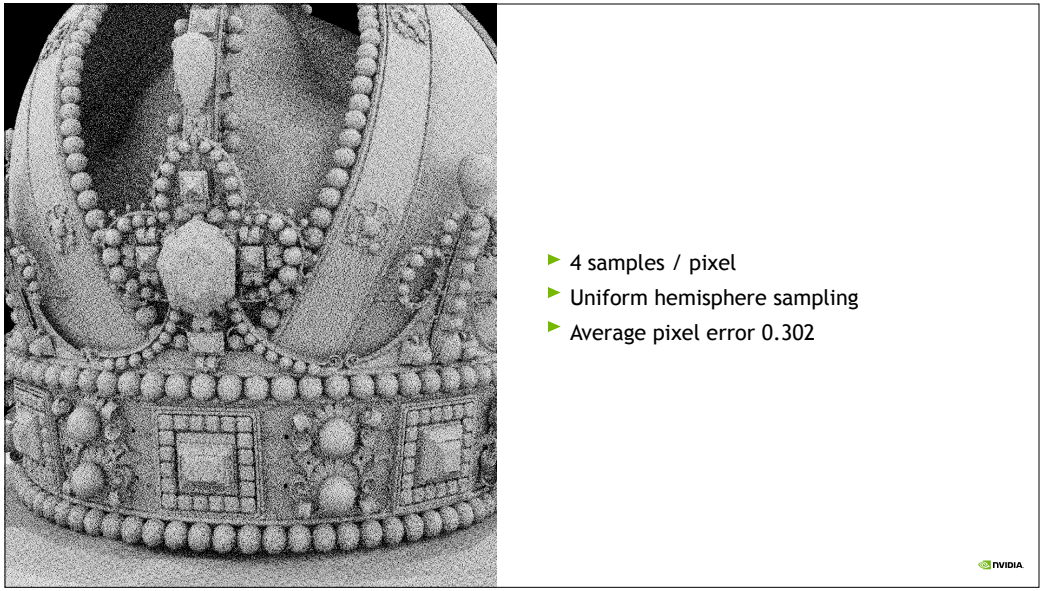
$(\xi_1, \xi_2) \in [0, 1)^2$



NVIDIA

How do we choose that random direction?

There are all sorts of mappings from 2D values in the unit square to directions on the surface of the unit hemisphere; here's one of them. It gives a direction assuming that the normal is aligned with the z axis, so you need to then transform the sampled direction to the frame around your actual surface normal. Just a few dot products to do that.



And here’s what we get, tracing 4 rays per pixel. If we compare to a reference image, we can determine that the average pixel error is about 0.3. Not terrible, but we’ll see it could be better.

MONTE CARLO INTEGRATION 101

"Biased" estimator $$E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] = \int f(x)\,\mathrm{d}x \qquad X_i \sim p(x)$$

Before we chose all possible directions on the hemisphere with equal probability. Think about that for a second, though—all the directions the horizon are sort of wasted since the cosine theta value will be very small. You're looking at the prospect of tracing a ray that will either contribute some small epsilon to your AO value or zero, depending on if it's occluded. It's not going to make much of a difference to the final result and in a sense, the ray is kind of wasted.

Monte Carlo integration also lets you sample non-uniformly. You can define a probability distribution and use that to choose your directions. For AO, you want to take more of them where cos theta is larger and fewer toward the horizon.

Here's another Monte Carlo estimator that lets you do that. Now the samples X_i are drawn from a probability distribution p(); there are more samples where p() is large and fewer where p() is small.

Now note that division by p(x_i) in the estimator. What's happening is, if you take more samples in some part of the domain you're integrating over, then p is larger there. In turn, you correct for those excess samples by dividing by p, reducing their contribution so that you get the right answer.

MONTE CARLO INTEGRATION 101

"Biased" estimator $$E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] = \int f(x)\,\mathrm{d}x \qquad X_i \sim p(x)$$

Want $\frac{f(X_i)}{p(X_i)} \approx c$

NVIDIA

If you sample more where f is large and less where it's low, then each sample makes about the same contribution. That's a really good thing.

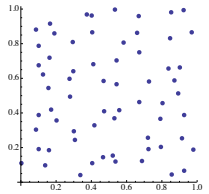
Intuitively, that's good from an efficiency perspective—a low contribution ray is sort of not worth the effort to trace; you want all rays to have the same contribution

Thought exercise: what if the ratio is always the same? You only need a single ray to estimate the integral perfectly. Woop.

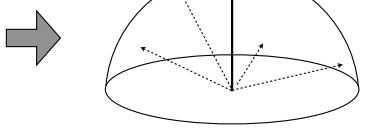
To be able to do that, it turns out that means that you need to be able to evaluate the integral analytically, in which case you don't need Monte Carlo anyway. Here we don't know the visibility function in closed form so that's not possible anyway, but we're doing better by eliminating the variation due to the cosine term.

COSINE-WEIGHTED HEMISPHERE SAMPLING

$$(x, y, z) = (\sqrt{\xi_1}\cos(2\pi\xi_2), \sqrt{\xi_1}\sin(2\pi\xi_2), \sqrt{1-\xi_1})$$



$$(\xi_1, \xi_2) \in [0, 1)^2$$



NVIDIA

Here's a recipe for taking numbers from the 2D unit square and turning them into cosine-weighted hemispherical directions. In other words, the probability of a ray being sampled is proportional to the cosine theta term—more rays around the normal, down to fewer of them around the horizon.

APPLICATION TO AO

Function $$f(\omega) = \frac{V(\omega)\cos\theta}{\pi}$$

"Biased" estimator $$E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{\pi}\frac{V(\omega_i)\ \cos\theta_i}{p(\omega_i)}\right]$$

Cosine sampling $$p(\omega) = \frac{\cos\theta}{\pi}$$

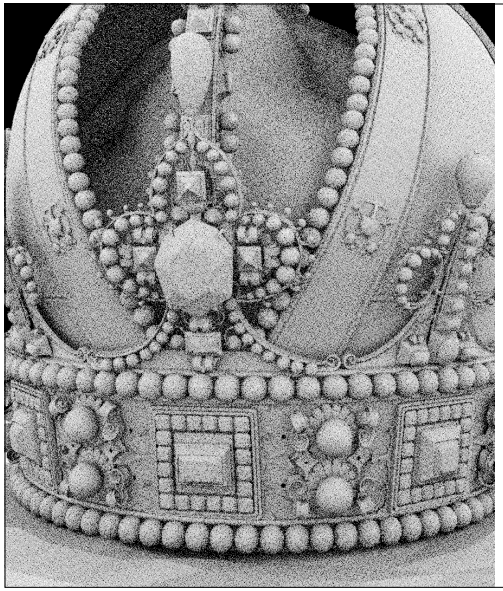
Final estimator $$E\left[\frac{1}{N}\sum_{i=1}^{N}V(\omega_i)\right]$$

NVIDIA

Let's take the AO function and substitute it into that second Monte Carlo estimator. After simplification, we get to the final estimator at the bottom; we can see that every ray contributes either zero or one. That's a great place to be—we're not tracing any more of those rays that make a small contribution. All of the rays have the same weight and in a sense, all contribute the same amount of information.

In general, you can't always achieve that ideal—all rays contribute the same—but the point's really important to keep in mind: how much information is this ray giving me?

So we've taken the long way to get to the known best practice for AO. but now we've got some mathematical vocabulary to talk about ways of doing this stuff.



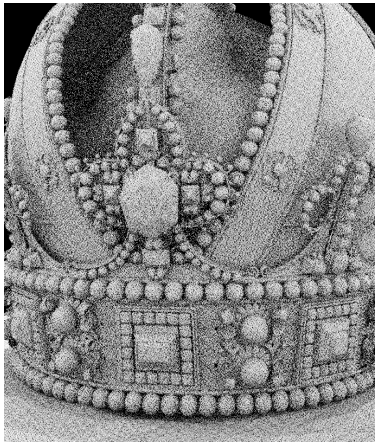
- 4 samples / pixel
- Cosine-weighted sampling
- Average pixel error 0.229

- Error is 24% less than uniform sampling

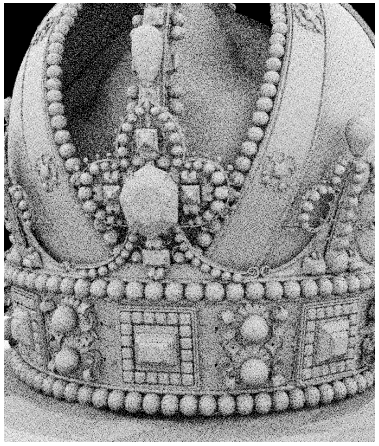
NVIDIA

And here’s the result. Again, 4 samples per pixel, but now with cosine-weighted sampling. Error got substantially better, 24% less than with uniform sampling, with the exact same number of rays traced.

4 uniform samples / pixel, error 0.302

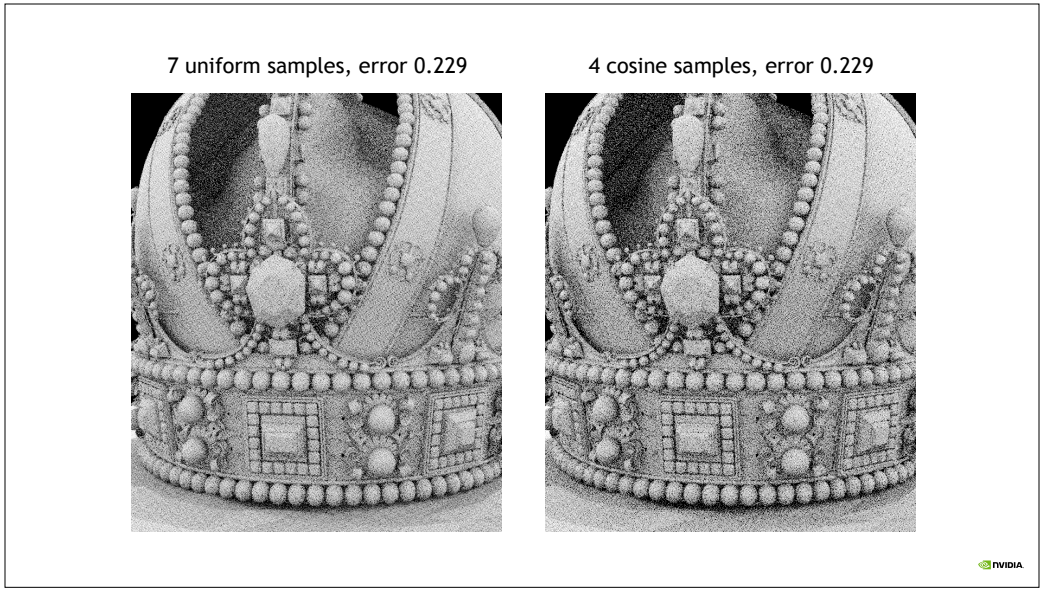


4 cosine samples / pixel, error 0.229



NVIDIA

Here are the two images side by side. The error difference is pretty clear visually.



Next I did a a little experiment, taking more uniform samples until I got the same error as with cosine sampling. I had expected I'd overshoot or undershoot, but as it turned out, it worked out nearly exactly. 7 uniform rays gives essentially the same error as 4 cosine-weighted rays. That's 1.75x more rays with the worse sampling method to get the same quality results.

Put another way, if you'd been doing uniform sampling and switched to cosine-weighted, it's like your GPU was instantly able to trace 1.75x as many rays as before.

COSINE-WEIGHTED IS BARELY MORE WORK

$$(x, y, z) = \left(\sqrt{1-\xi_1^2}\cos(2\pi\xi_2), \sqrt{1-\xi_1^2}\sin(2\pi\xi_2), \xi_1\right)$$

Uniform

$$(x, y, z) = (\sqrt{\xi_1}\cos(2\pi\xi_2), \sqrt{\xi_1}\sin(2\pi\xi_2), \sqrt{1-\xi_1})$$

Cosine-Weighted

NVIDIA

The great thing about this is that cosine-weighted sampling is barely any more work. Here are the two sampling recipes, right next to each other. The amount of work is basically the same, so there's really no reason not to do cosine-weighted sampling in this case.

CHARACTERIZING ERROR: VARIANCE

Definition: $V[Y] \equiv E[(Y - E[Y])^2]$
$= E[Y^2] - E[Y]^2$

e.g. $Y = f(X)$
$E[Y] = \int f(x)\,\mathrm{d}x$

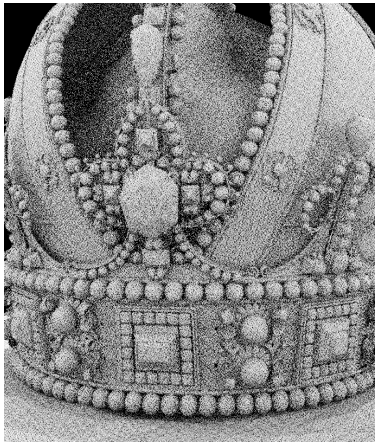
NVIDIA

A little more math.

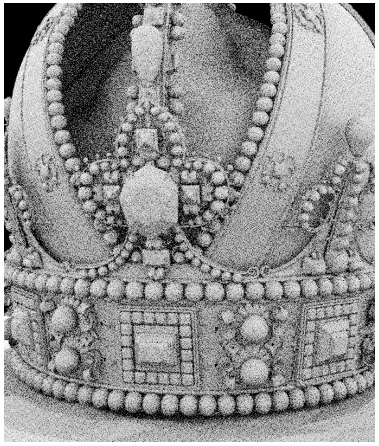
Variance is a good way to look at how good a Monte Carlo estimator is. It's basically a measurement of the expected squared distance between what your estimator gives you and the correct result. Here it's written in terms of an expectation over random variables, so it's basically defined over all of the possible random results that you might compute.

Low variance is good—that says that most of the time, your estimator is close to the thing you're integrating.

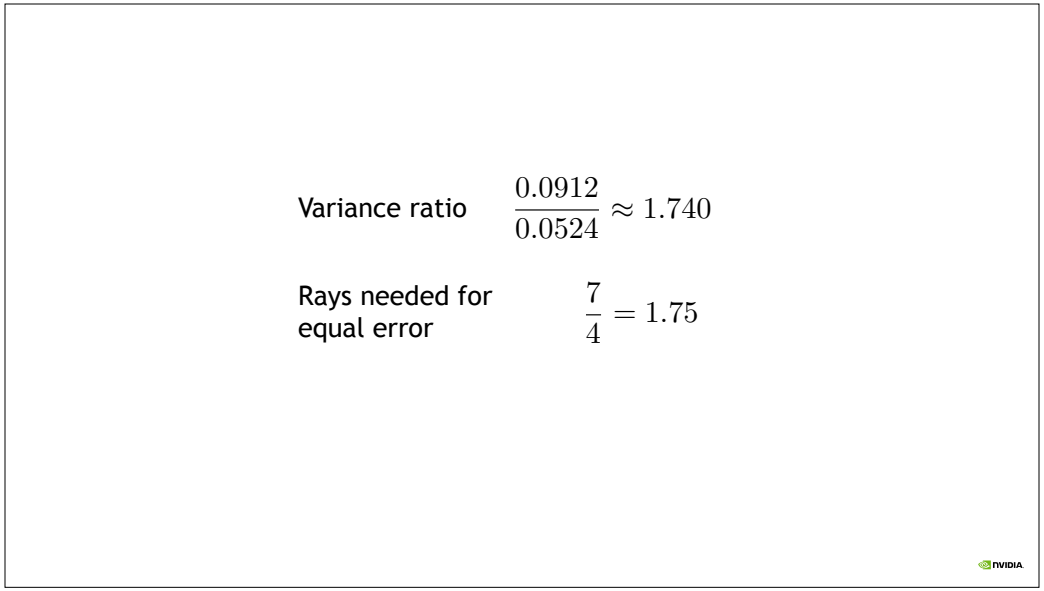
4 uniform samples, variance 0.0912



4 cosine samples, variance 0.0524



Let’s compare the variance of these two images.



Now, just for fun, let's throw a few numbers together.

First, we've got the ratio of the variance of the two images. The uniformly sampled rays give 1.74x higher variance than the cosine-sampled rays.

Then, we've got the ratio of rays traced to get an equal error result. 7 uniform rays gives the same error as 4 cosine weighted rays. That's a ratio of 1.75x.

Well that's interesting. Those values are nearly the same…

VARIANCE DECREASES LINEARLY WITH SAMPLE COUNT

Variance ratio $\frac{0.0912}{0.0524} \approx 1.740$

Rays needed for equal error $\frac{7}{4} = 1.75$

- **BUT:** Variance is squared error
- → Error goes down with the **square** of the number of samples 😨

NVIDIA

And it turns out that's a thing. With a little math you can show that variance decreases linearly with more samples. If you want half the variance, take twice as many samples. That's what we're seeing empirically.

So let's stop and digest that. For what may seem like an innocuous difference in sampling strategy—uniform versus cosine-weighted, we have a massive difference in efficiency.

This is really important so I'm going to repeat it:

- Variance is squared error

- Variance decreases linearly with more samples: 2x more samples, half the variance.

- And that means that error decreases quadratically with more samples: 4x more samples, half the error.

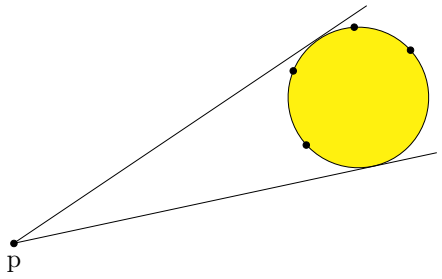
That 4x means that it's really important to choose your samples well. "Just trace more rays" won't get you there.

You see this if you write an interactive ray tracer. Wow does it start quickly—just a few samples and you're starting to get a nice result. And at the very beginning, that 4x isn't so bad. From one ray to 4 rays is just 3 more. But if you trace a few hundred rays, maybe you still have noise, and the prospect of 4x more of them is pretty unappealing.

SAMPLING SPHERICAL LIGHTS: UNIFORM

Uniform:

$$x = \cos(2\pi\xi_2)\sqrt{1-z^2}$$
$$y = \sin(2\pi\xi_2)\sqrt{1-z^2}$$
$$z = 1 - 2\xi_1$$



NVIDIA

We could go through lots more examples of this idea of how to sample well, but I want to discuss just two more, related to lighting and shadows.

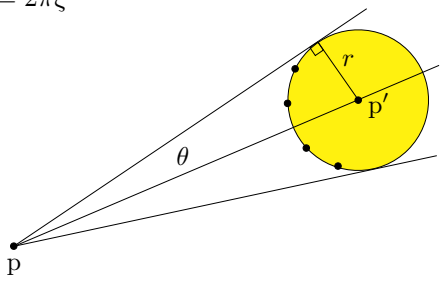
Consider computing illumination from a spherical light source at a point p. You want to choose some point on the light source and trace a ray to them to see if there's a blocker. Here, we're using a technique to uniformly sample points on the sphere—all points on the surface, sampled with equal probability.

Looking at the sample points here, you can see the problem. Some of the points are on the backside of the sphere and there's no way they'll be visible—the front of the sphere blocks them. In fact, you can never see more than half of a sphere from a point outside of it. You see half if it's infinitely far away, and less as it gets closer. Think about a point right next to the sphere; it'll barely see any of it.

SAMPLING SPHERICAL LIGHTS: VISIBLE SURFACE

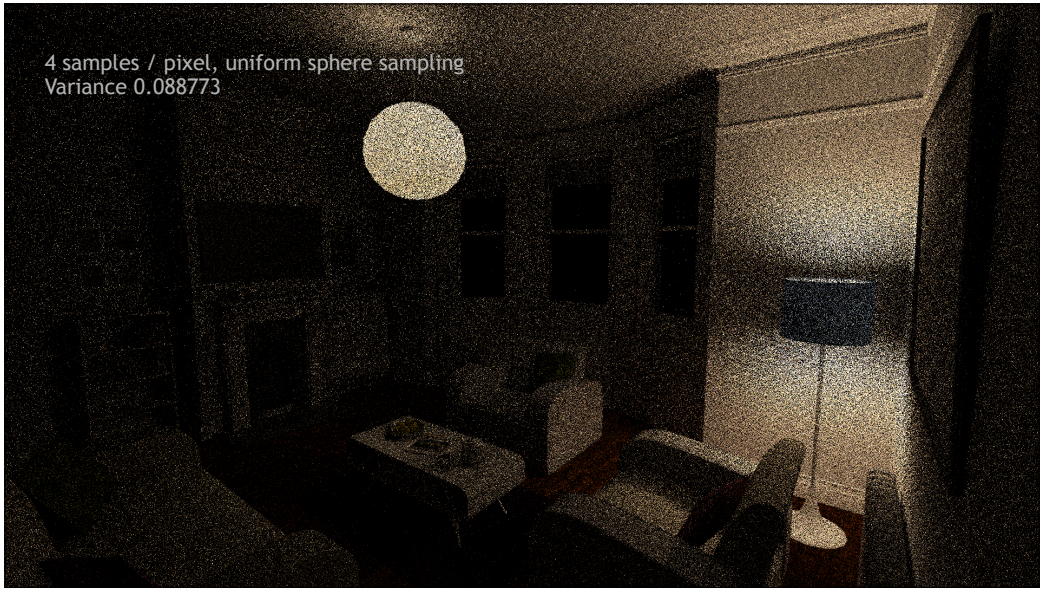
$$\cos\theta' = (1-\xi) + \xi\cos\theta \qquad \phi = 2\pi\xi$$

$$p(\omega) = \frac{1}{2\pi(1-\cos\theta)}$$



NVIDIA

A better thing to do is to sample points on the visible portion of the sphere. You can do that by considering the cone that bounds the sphere from the point and uniformly choosing directions within the cone. It's not much more work, and now all of your rays will be to parts of the sphere that are possibly visible to the point (depending on occluders).

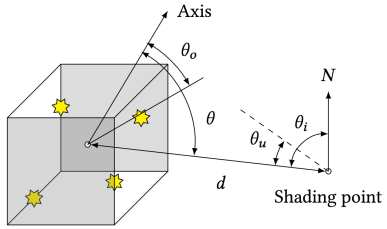


Let’s see how that works out for us. Here’s a scene with two spherical light sources, rendered with 4 rays per pixel, sampling points on the entire sphere.

And here it is sampling within the cone—the better technique. Variance decreases by 2.90x! Again, it’d take 2.9x as many rays with the inferior sampling technique to get the same quality.

CHOOSING A LIGHT SOURCE

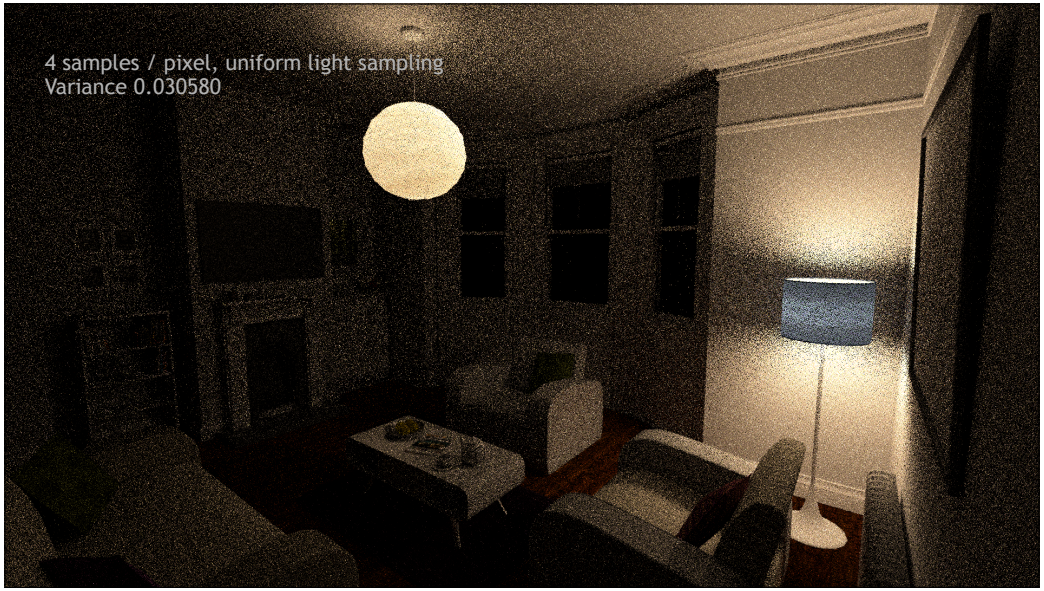
[Conty2018]



- Store lights in a BVH
- Cluster by position, emission direction and spread
- Given a shading point, can quickly cull unimportant lights, find ones to focus on

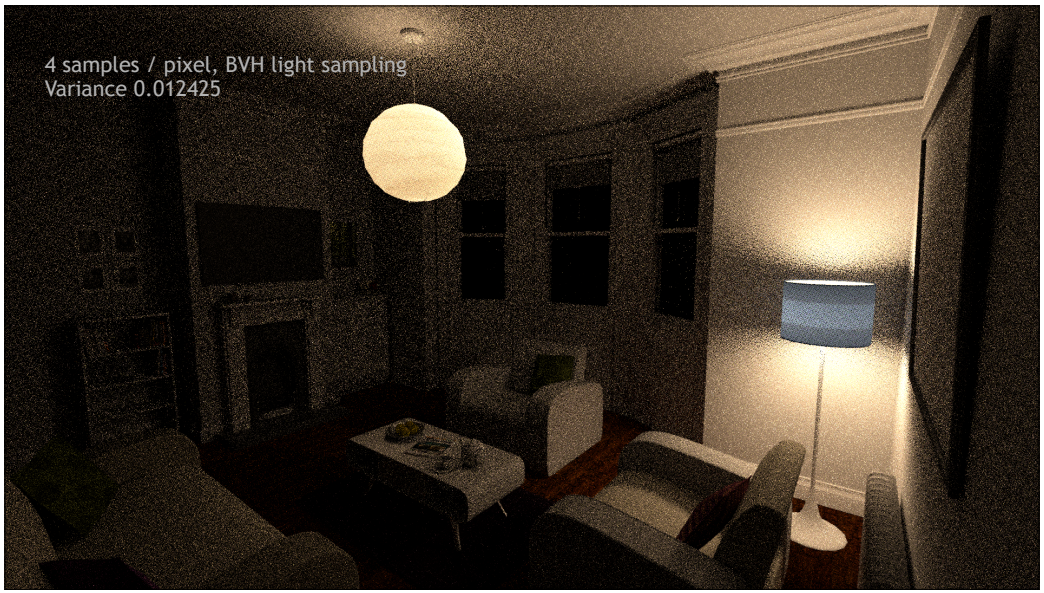
NVIDIA

One more light source example. There was a neat paper at HPG this year from some folks at Sony Pictures Imageworks about rendering scenes like this, with hundreds of thousands of light sources. Thanks to light's r^2 falloff, you can still render scenes like this efficiently, but the trick is to quickly figure out which lights have the highest potential contribution to the point being shaded.



I implemented their algorithm and was impressed with how well it worked. What surprised me was what happened even with scenes with just a few lights.

Here's that scene again, where we left off before—sampling points on the light within the visible cone. 4 rays per pixel. Each ray randomly chooses between the lights with equal probability—half the time tracing a shadow ray to one, half the time tracing to the other.

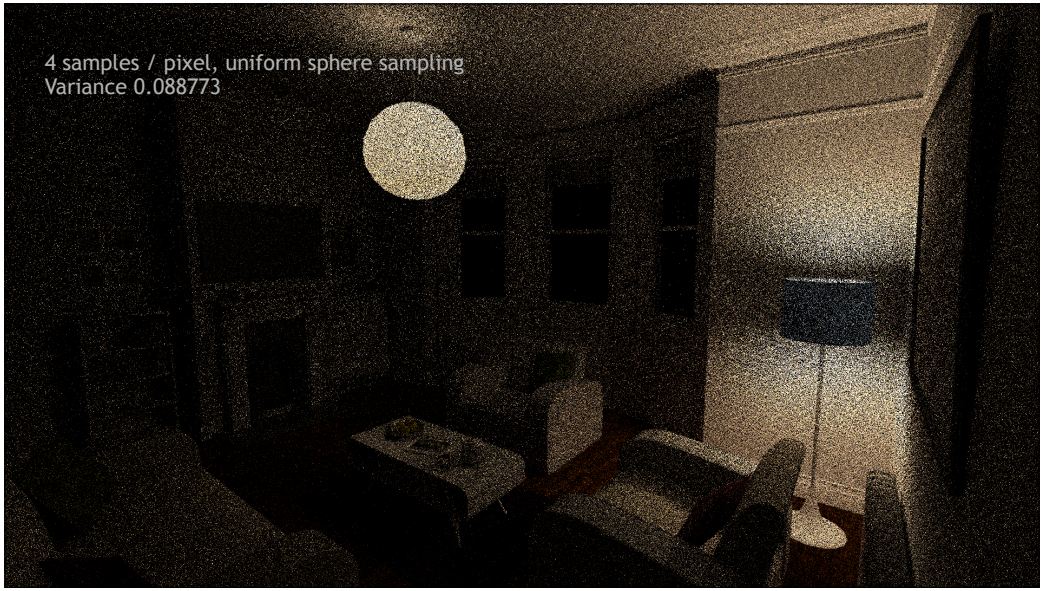


Here's the scene rendered using Sony's light sampling technique. Variance is 2.46x lower! Look at the difference on the wall by the light with the blue shade, for example—there you're close to that light, and the other's making a relatively small contribution. It's not worth tracing so many rays to it.

I was really surprised by how much this reduced variance in this case—I didn't expect it at all ahead of time, but it makes sense when you stop and think about it. And the lookup is really cheap–it's pretty efficient math.

Again, you would need 2.46x more rays to get the same noise level with choosing between the two lights with equal probability.

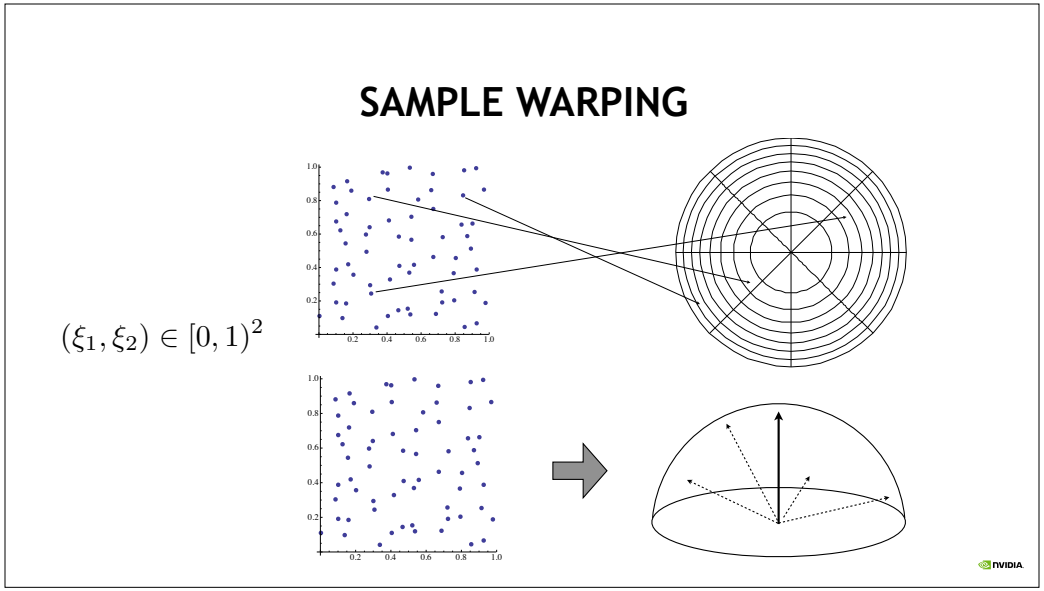
Put these two together, and it's overall a 7.1x variance improvement accounting for choosing a light and choosing the point on the light well, or you'd need 7x more rays to get this quality with the worse approaches (which aren't actually terrible!)



Here we are, back where we started. 7.1x more variance.

I love this stuff and could go on and on. There's how you take samples from BSDFs, there's how you combine multiple sampling techniques with multiple importance sampling. That's super important for reducing variance as well, but we're going to move on to the second topic.

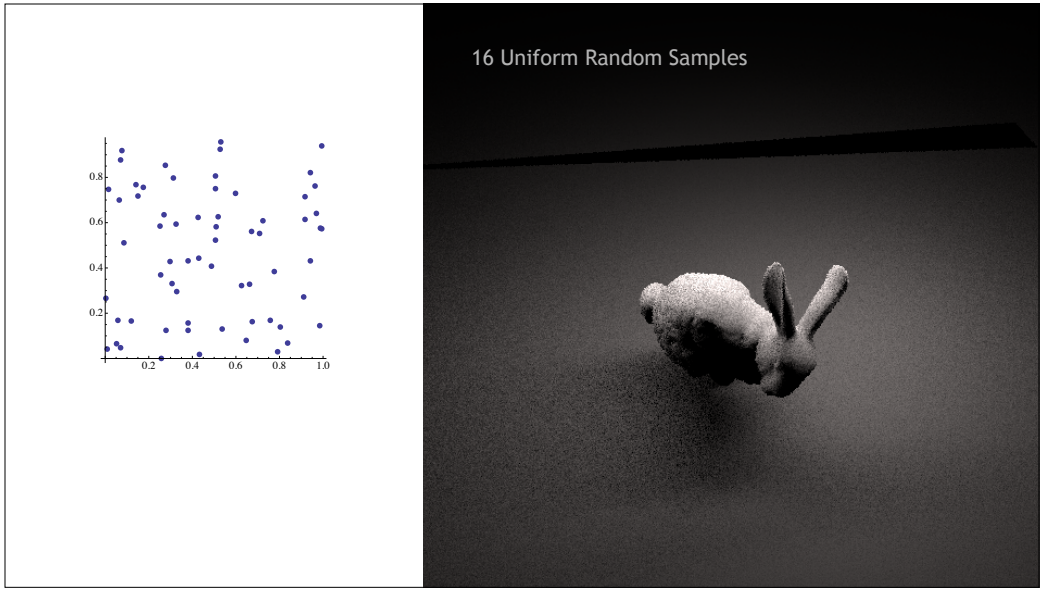

CHOOSE YOUR (NOT) RANDOM
NUMBERS CAREFULLY



All of these sampling techniques are based on warping random numbers from the unit square to some other domain—to the hemisphere, to the sphere, to the cone surrounding a sphere, to a disk. You can also generate samples from a BSDF's scattering distribution, or choose directions to an IBL light source.

There are tons of those out there, but they all start with these values between 0 and 1.

There's a nice orthogonality in this: there's "what are those values you start out with" and then there's "how do you warp them to the distribution of the thing that you want to sample to use that second Monte Carlo estimator". Now we'll talk about those values. It turns out that they really matter, too.



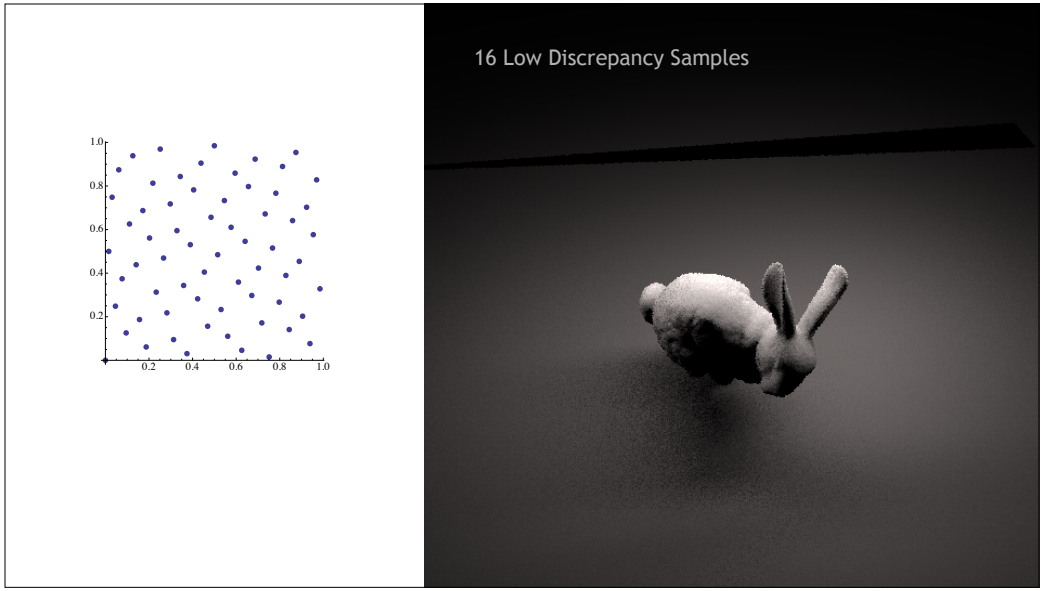
Here is a plot of 64 random samples and then an image rendered with 16 random samples per pixel. There's a long skinny triangle area light illuminating the bunny—you can see the black back side of it above the bunny.

Don't do this.

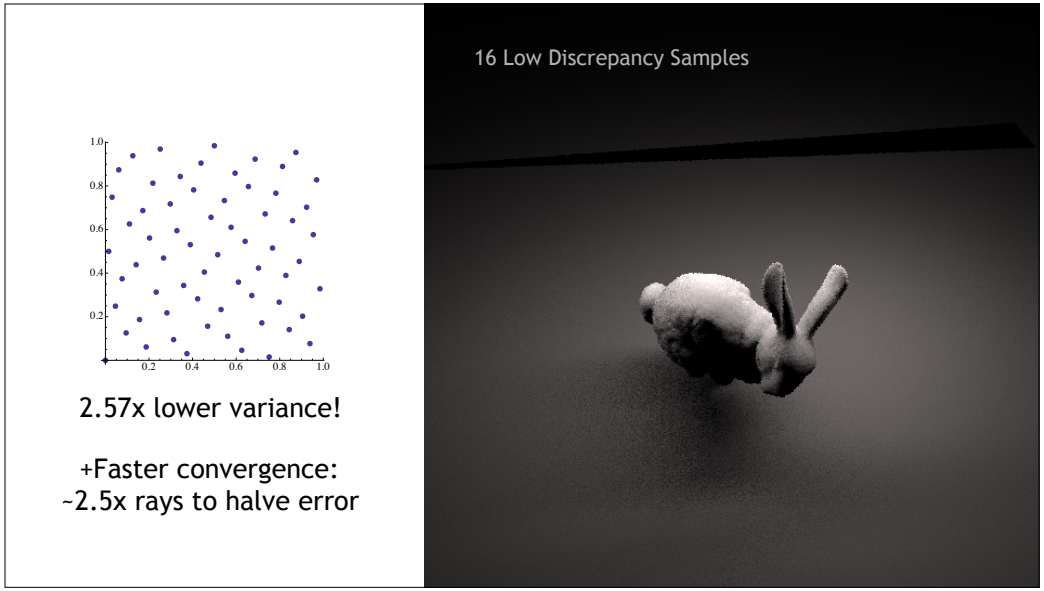
It's the easy way to do it—just pick a random number, then another one, one after another.

But what happens is that the samples sometimes clump together. You can see that in the plot here. When the samples clump together, that means that when you warp them to some domain, like the surface of a light source, you choose two nearby points in that domain.

In turn, the rays you trace will be very close together and so the second one doesn't really give you any new information. You'd much rather trace a ray to somewhere new and be better spread out across the light.



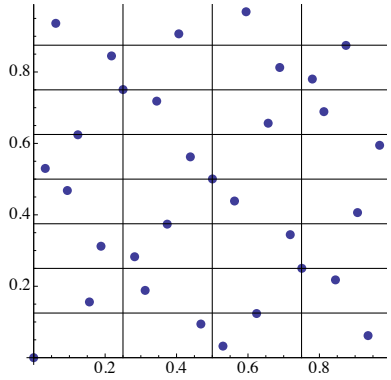
Here's that scene rendered with this sampling pattern. It's not random. You can see a nice structure to the samples—they're not clumped together. You can see that the image is cleaner for the same number of rays.



And the variance numbers show it. Nearly a 2.6x reduction in variance for the same number of rays.

There's more: it turns out that with really good sampling patterns like this one, the error actually goes down at a faster rate than I said before. With this, it's not 4x the rays to halve the error, but about 2.5x rays. That's a nicer place to be.

STRATIFIED SAMPLING (4X8)



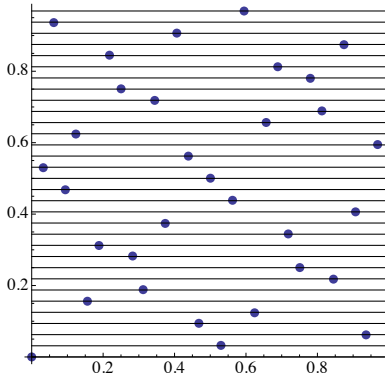
How do you improve on uniform random sampling?

A first step is stratification: split the [0,1] domain into boxes and place one sample point inside each box. That eliminates some of the worst case clumping and ensures that the points have decent overall coverage.

However, it means that you need to choose the number of samples ahead of time and that the total has to be the product of two integers. That's kind of a bummer.

You can still get sample clumping, too. Worst case, you could have four samples all meeting at the shared corner between two boxes, though we don't have that here. It's easy to do, though—super efficient, barely any more work than uniform random.

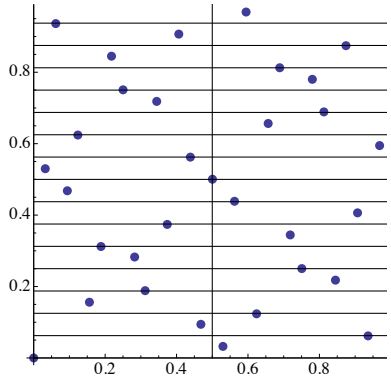
ELEMENTARY INTERVALS (1X32)



Stratification is good, but there are even better sampling patterns. It turns out that this is one of them.

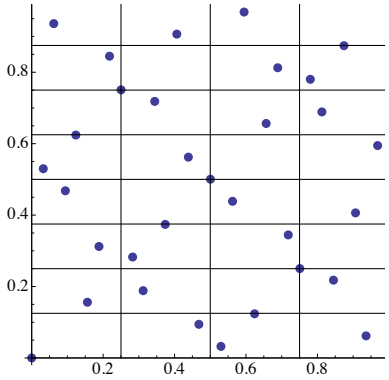
Let's consider different ways of splitting the domain into boxes. Here we've got 32 horizontal boxes, with one sample in each box. That ensures that, for example, if you project the sample points onto the y axis, they won't be clumped up. That turns out to be helpful.

ELEMENTARY INTERVALS (2X16)



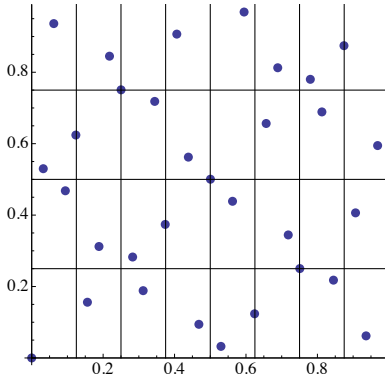
Here are the same sample points, but now we've stratified this way: 2x16 boxes. Again, one sample point is inside each box, so they're well distributed in this way.

ELEMENTARY INTERVALS (4X8)

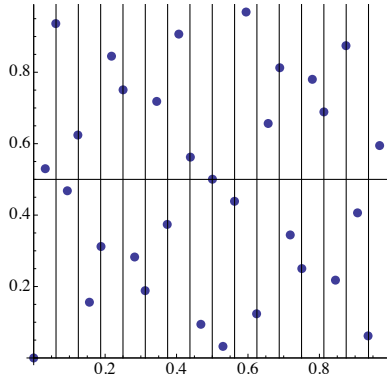


Another stratification. Same point set. You can probably see where this is going by now.

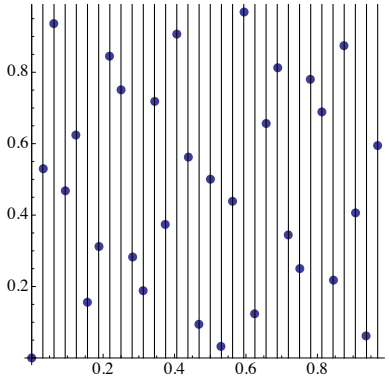
ELEMENTARY INTERVALS (8X4)



ELEMENTARY INTERVALS (16X2)



ELEMENTARY INTERVALS (32X1)



NVIDIA

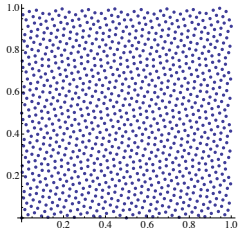
And finally, a 32x1 stratification. All of them satisfied by this set of points.

This worked really well for that scene with the bunny and area light; when the sample points are really stretched out across the light source, they're still well-distributed in one dimension, which leads to less error in the result.

It's kind of crazy; I'm not sure I'd have believed it was possible to generate points that fulfill all of those constraints in the first place. Even better, the algorithms that generate them are quite efficient—it's just a handful of instructions if you implement them well. There's some fun bit twiddling involved along the way as well.

WHAT'S IMPORTANT IN SAMPLING PATTERNS?

- Low discrepancy: ~generalized stratification
- Blue-noise: ~how close different samples get
- Progressive patterns
  - Can take an arbitrary number of them and (some) prefixes are well-distributed
- Many ways to get some of the above; see the pmj02 sequence [Christensen 2018] for the state-of-the-art in getting all of them



Blue noise point set

There's been lots of work on these sorts of sampling patterns. You can make a research career out of the topic, if you're so inclined.

What we've been looking at are what are called low discrepancy points. They have fast convergence but can exhibit harsh aliasing artifacts at low sampling rates. Examples include Halton, Hammersley, and Sobol sequences.

Another school of thought is to optimize for what's called blue noise—basically a guarantee that points don't get too close together. The error starts out more pleasing visually—it's higher frequency, and the human visual system is ok with that—but it's recently been proven that it doesn't converge very efficiently. Asymptotically, it's actually as bad as uniform random samples.

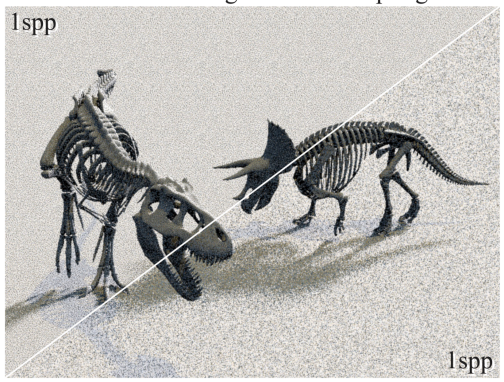
Progressive patterns are nice—prefixes of them are well-distributed. That's nice when you're doing adaptive sampling. You can keep taking more samples until you're happy and know that in the aggregate, they're well distributed. You don't get that with a stratified pattern, for example.

We're just now starting to see convergence among these methods. Check out this recent paper from Pixar from EGSR this year. It's one of the best options currently, IMHO, and also has all the references to previous work if you want to dig into this.

BLUE-NOISE DITHERED SAMPLING

Our dithered light source sampling

[Georgiev2016]



Random pixel decorrelation

NVIDIA

One last thing about sampling for making images. Numerical error isn't everything when a human's in the loop.

Here's an image from a cool paper from some of the folks who work on Solid Angle's Arnold renderer. Both halves of the image have the exact same numeric error. That's crazy—you'd never expect it, but it's true.

What they're doing is optimizing sample positions across neighboring pixels, not just within them. They make sure that nearby pixels have sample values that are quite different. In turn, that decorrelates their error. Otherwise, as we see in the bottom right, clumps of nearby pixels may end up taking nearby sample values and in turn all having roughly the same amount of error. That leads to that lower-frequency noise in the error.

It's a huge perceptual benefit, again because human vision doesn't mind high frequency error as much as lower frequency error.

Everyone wants to make the best image they can for a given number of rays.

Trace more rays where error is high, stop tracing rays where error is low.

Lots of work in this area.

VARIANCE-DRIVEN SAMPLING

- Simple idea:
  - Periodically estimate variance at each pixel based on samples taken so far
  - Sample more where variance is high
  - Better: sample more where variance / estimated value is high
  - Do this *after* tone map, etc.
- Offline (quality driven): once a pixel's variance is low enough, stop working on it
- Real-time (framerate driven): take more samples where variance is highest

NVIDIA

Next up, determining how much variance you're seeing and using it to your advantage.

We already know how increasing the sampling rate reduces variance. So, if we want to minimize variance, we want to trace more rays where variance is high.

In offline rendering, there's usually a quality target: render until the error is acceptable. They compute per-pixel variance and then keep working on pixels until their variance is low enough.

In real-time, there's a time target: the framerate. We'll want to instead think of having a fixed ray budget and want to use it where it's most useful. Variance is great for that, too.

Here we can see why progressive sampling patterns are really helpful: we don't know how many rays we're going to trace in a pixel at the start, so as we trace more, we'd like to make sure they're not clumped up with the rays we've already traced.

COMPUTING THE SAMPLE VARIANCE

```
float sample_variance(float samples[], int n) {
  float sum = 0, sum_sq = 0;
  for (int i = 0; i < n; ++i) {
    sum += samples[i];
    sum_sq += samples[i] * samples[i];
  }
  return sum_sq / (n * (n - 1))) -
         sum * sum / ((n - 1) * n * n);
}
```

Important: sample variance is an *estimate* of the true variance

Given a bunch of samples, it's easy to estimate how much variance there is in their corresponding Monte Carlo estimate. Here's a function to do it.

Note that it's easy to compute variance incrementally; you just need maintain the sum and sum of squared sample values, updating them each time you take another sample.

One note: this is the *sample variance*, which is to say that it's the variance based on the values of the samples we've taken. There can be error in this estimate, as we'll see visually in a second.

Ok, so here's a car rendered with four samples per pixel. It's lit by a HDR environment map. It's pretty noisy, but on the other hand, it's pretty amazing what four samples per pixel will give you. You can definitely tell what it's going to look like when it's converged.

And here's the per-pixel sample variance. It tells us a lot.

All in all, it's a good guide to which pixels are noisy. Note that the speckles on the car door that are both reported as having high variance here are places where there are spikes in the rendered image.

Next, note that the shadow boundary underneath the right side of the car has low variance—you don't see it at all. That's good—we don't need more samples there. If instead we were using color contrast or the like to decide where to sample, you'd think that edge needed work, but it really doesn't.

There is a problem, though. Look at the reflection of the metal wheels. There ought to be a nice blurry reflection there, and there would be with a lot more rays. Many of the pixels where that reflection should be think they have low variance, though—that's the problem with sample variance. They've traced their four rays and haven't found anything interesting—all of their rays had about the same contribution, so they're thinking hey, maybe I'm done.

The problem is that they haven't found that bright shiny wheel. If they had, they'd know there was something going on that needed more sampling.

SAMPLE VARIANCE IS JUST AN ESTIMATE

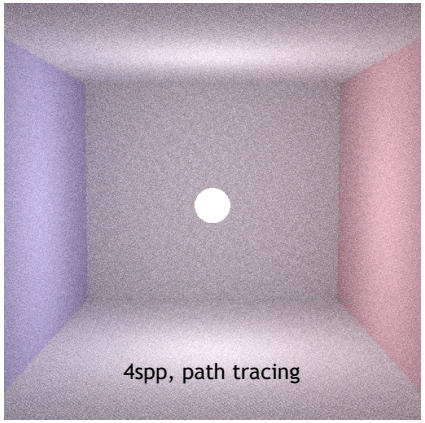
- TONS of work on estimating it for denoising
  - [Zwicker 2015] Recent advances in adaptive sampling and reconstruction for MC rendering
- General ideas:
  - Incorporate sample variance at nearby pixels
  - Maybe weighted by how close auxiliary features (position, normal, etc.) are

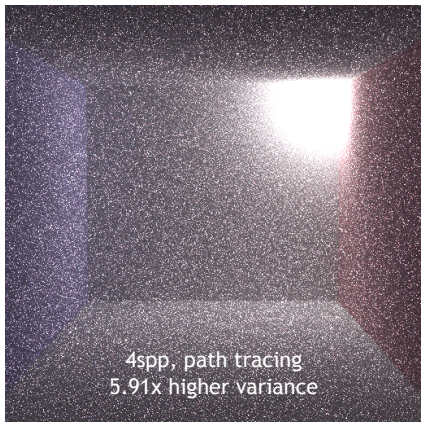
This issue is well known; there's lots that can be done. In general, a lot of it boils down to incorporating neighboring pixels' variance estimates. This survey article on the topic is a good one.

The SEED team at EA used variance as part of their denoising pipeline. The idea is that if variance is high, then you should blur with a wider kernel, grabbing more values from nearby pixels. If variance is low, then you’re good—no need to blur very much.


DISALLOW THE
“FIREFLY”

MODERN LIGHT TRANSPORT ALGORITHMS AREN'T ROBUST





NVIDIA

A tale of two scenes…

On the left, a small spherical light source in the middle of a box, rendered with 4 samples per pixel. This is a *really* easy scene. Any time you trace a ray to the light, it's guaranteed to get illumination—there are no occluders between the walls and the light and we're sampling the cone, so always have a point on the light.

On the right, all I've done is moved the light to the corner of the box. Variance is nearly 6x higher. What happened? I thought path tracing could render everything.

The problem is that now there's a bright source of indirect light from the walls around the light source. Consider a pixel on the floor now—the direct lighting to the sphere is easy, but think about a ray traced to sample indirect lighting. If it happens to hit a wall in the corner, it'll decide there's tons of indirect light—"look at this bright thing I found". If it doesn't, then it won't think there's much indirect light at all. Hence, you get this really noisy image.

THE CURSE OF HIGH VARIANCE

- Once a high-variance sample sneaks in, you're in big trouble
- Consider uniform sampling of

$$f(x) = \begin{cases} 1 & x < .999 \\ 100 & \text{otherwise} \end{cases}$$

- 6 samples: (1, 1, 1, 1, 1, 100) ≈ 17.5
- Take 6 more: (1, 1, 1, 1, 1, 100, 1, 1, 1, 1, 1, 1) ≈ 9.25

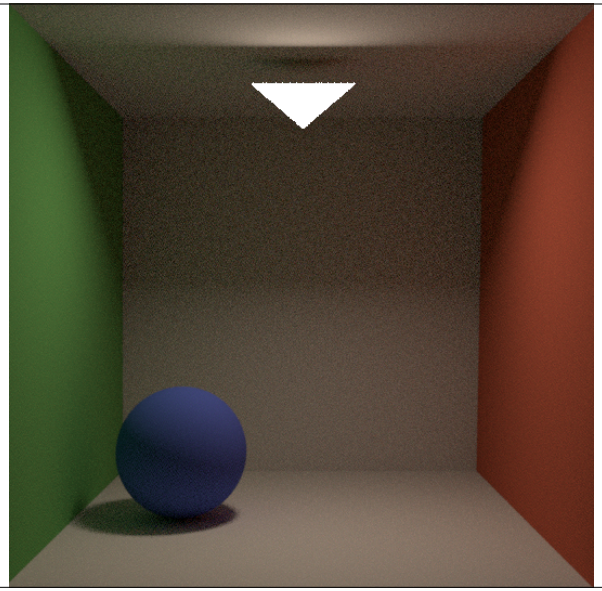
- Recall that variance goes down linearly with number of samples…

NVIDIA

The problem with this kind of variation is that it takes a long time to fix with more samples. Consider this little function over [0,1] that's basically 1 everywhere but spikes to 100 in a small part of the domain. It integrates to 1.1.

Now start taking samples from it. If you hit that place where it's high, now your estimate has spiked really high. You need tons more 1-valued samples to average that out.

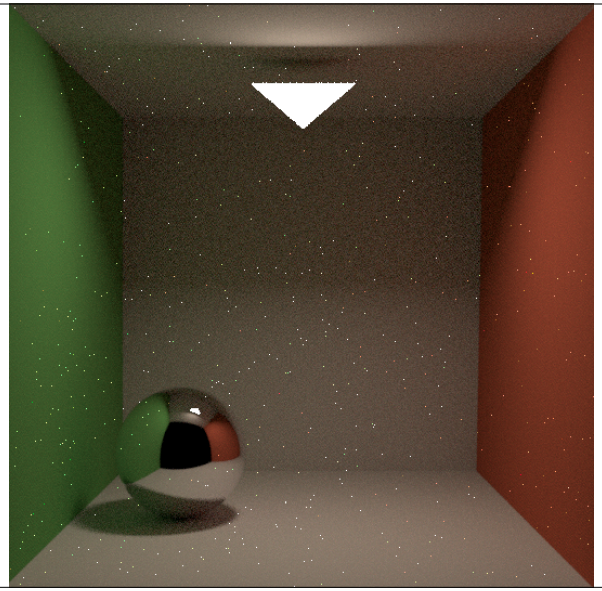
Path Traced, 16 samples



Let’s see another example of this problem and then a solution.

This is an easy scene, and with 16 samples per pixel we get a pretty clean image.

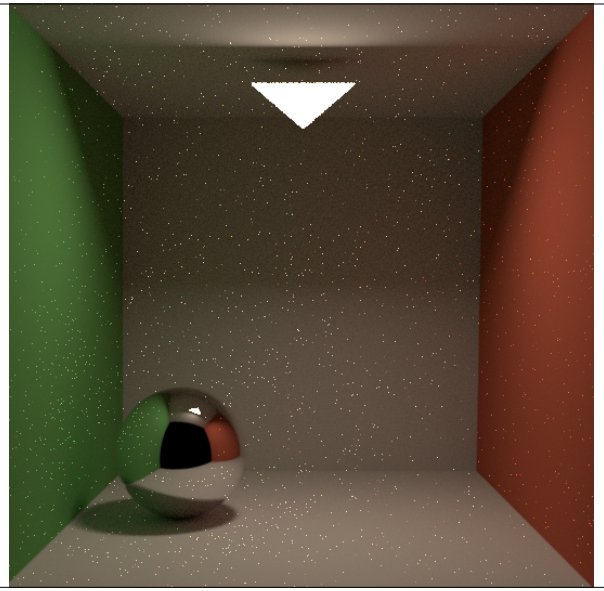
Path Traced, 16 samples



Now let’s replace the material on the sphere with a mirror.

I thought ray tracing was supposed to be good at mirrored spheres. [Wait for uproarious laughter.]

Path Traced, 64 samples



Ok, let’s trace more rays to make variance go down.

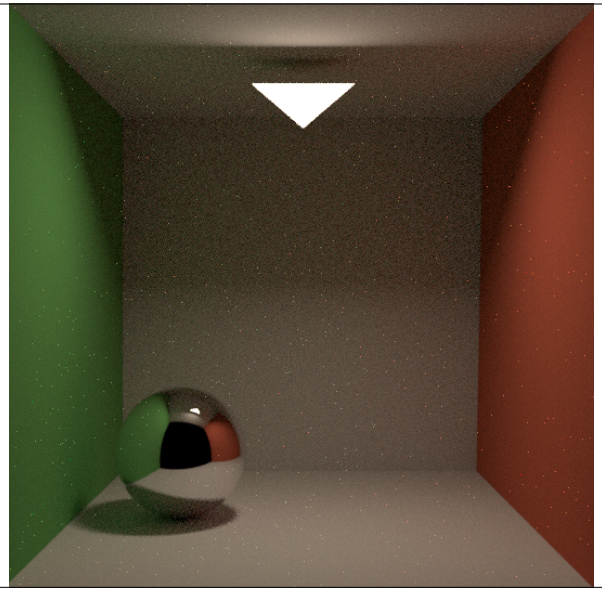
OMG more noise.

We’re slowly getting better with the first set of spikes, but now we’ve got more of them since other pixels happened to find that rare but bright bath.

If you see this kind of spiky noise, you need to figure out what’s going on. It’s deadly otherwise. It’d take forever to fix it with more rays.

Path Traced, 16 samples
Clamped: c = min(c, 3)

19.9x less variance!
~10% darker

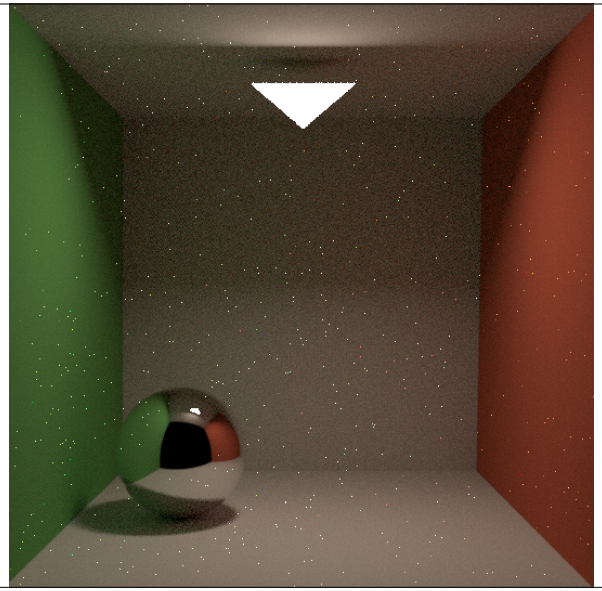


Clamping. It’s a dirty hack and I apologize, but it works and it’s widely used in practice.

A full implementation of the algorithm is there on the slide. That’s all there is to it—pick some maximum path contribution and if you find anything brighter than that, clamp it to the maximum. It’s really effective—look how much lower the variance is.

Now, it does make the image darker, here about 10%. That makes sense—we are discarding a good part of the contribution of those paths and they are indeed perfectly legitimate light-carrying paths.

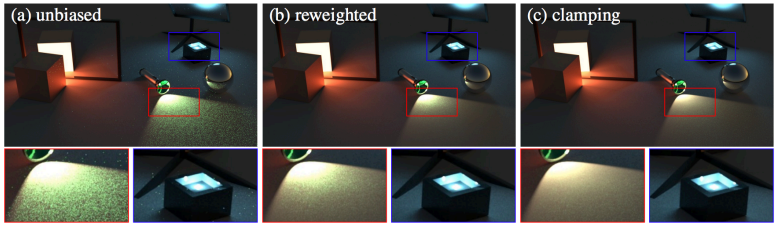
Path Traced, 16 samples



Here we are back where we started, 16 rays per pixel, no clamping. It’s a big difference.

MORE SOPHISTICATED OPTIONS

- Density-based outlier rejection [Decoro 2010]
  - Save all samples, analyze and filter out outliers
- [Zirr 2018]: Split samples into a few separate images based on brightness, then reweight based on a statistical analysis

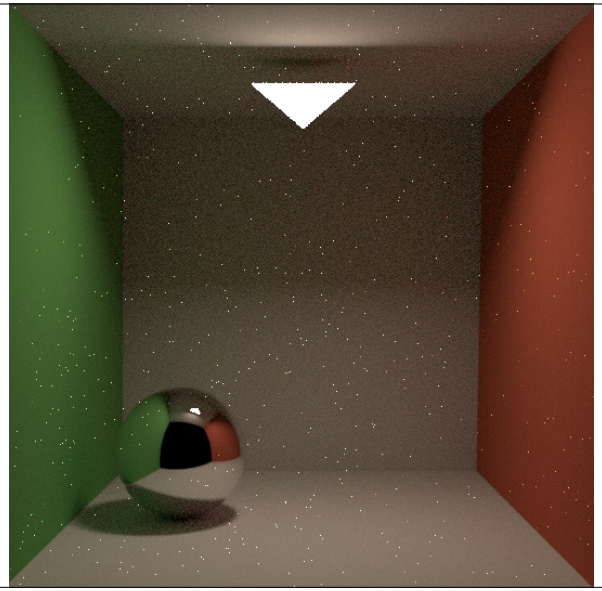


NVIDIA

There are much more sophisticated ways to do this. If you save the contributions of all of your samples, you can sort them and discard the large ones before averaging. Of course, that requires saving all of your sample values, which is expensive.

There was a nice paper at EGSR this year that split the image into a few buffers, each one responsible for a different brightness range. It then filtered those down, discarding the outliers, in a fairly principled way.

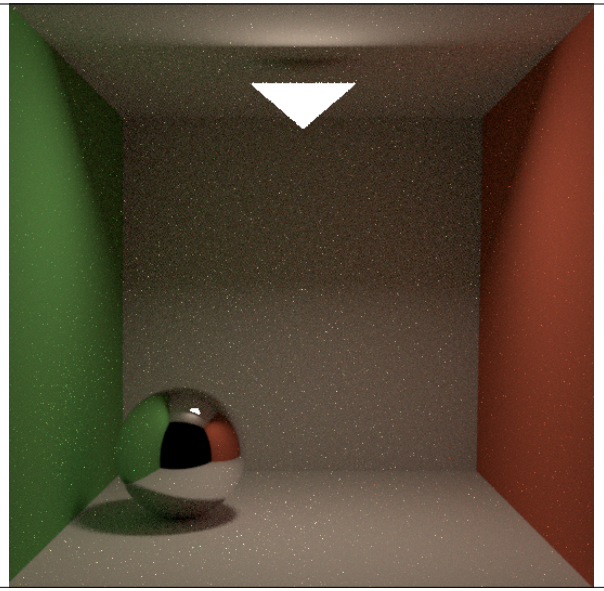
Path Traced, 16 samples



One more technique that’s good to know about: path regularization. Here we are back to the noisy image we started with.

Path Traced, 16 samples
Regularized ([Kaplanyan 2013])

9.8x less variance



With path regularization, you're basically roughening your BRDFs when they're seen indirectly, which in turn has the effect of blurring out the indirect lighting in the scene. In turn, you don't have those really localized paths that carry tons of light. Here we get a 9.8x variance reduction with path regularization. Not as good as clamping, but you don't have that darkening, which is nice.


WRAP-UP

SUMMARY

- Choose your rays well: importance sampling and good sampling patterns
- Use variance to:
  - understand where you're getting into trouble,
  - drive where to apply adaptive sampling,
  - and more accurately do post-render denoising
- Variance is insidious; if it comes to it, fight back with clamping and regularization
  - Fireflies are a sign of a big problem somewhere and need to be chased down and eliminated

We saw all these cases of 2x lower variance if you do this or if you do that, and that corresponds to that many fewer rays for the same quality compared to the relatively worse approach.

These are all multiplicative as well, so all in sum, it's a huge difference if you do this stuff or not. We saw roughly 18x from all the stuff together today—if you put together good sampling patterns and choosing lights and points on lights well.

Unfortunately, it turns out that it's easy to present what seems like straightforward input to light transport algorithms that turns out to give really high variance. I talked about clamping and path regularization as ways to work around it but my real hope is that we'll see rapid progress from having so many more graphics programmers looking at these problems. Just as there's been this incredible set of results after programmable shading came to GPUs, I'm confident that the same will happen with ray tracing.

OPEN QUESTIONS

- Impact of more and more denoising: the (conventional) renderer doesn't have to get it perfect
- The role of the rasterizer in the new world: it's still really fast!
  - Still worth using when visibility is coherent
  - Conservative rasterization to build data structures that can robustly determine visibility?
    - "Visible", "Occluded", "I don't know; please trace a ray"
- Frame-to-frame reuse and reprojection: performance vs. artifacts

ACKNOWLEDGEMENTS

- Pat Hanrahan
- Colin Barré-Brisebois
- Aaron Lefohn
- Chris Wyman
- Luca Fascione


Thanks!

REFERENCES

[Christensen 2018] Per Christensen, Andrew Kensler, and Charlie Kilpatrick, Progressive Multi-Jittered Sample Sequences, EGSR 2018.

[Conty 2018] Alejandro Conty and Christopher Kulla, Importance Sampling of Many Lights With Adaptive Tree Splitting, High Performance Graphics 2018

[DeCoro 2010] Christopher DeCoro, Tim Weyrich, and Szymon Rusinkiewicz, Density-based Outlier Rejection in Monte Carlo Rendering, Pacific Graphics 2010.

[Georgiev 2016] Iliyan Georgiev and Marcos Fajardo, Blue Noise Dithered Sampling, SIGGRAPH 2016 Sketches.

[Hachisuka 2004] Toshiya Hachisuka, High-Quality Global Illumination Rendering Using Rasterization, in *GPU Gems 2*.

[Kaplanyan 2013] Anton S. Kaplanyan and Carsten Dachsbacher, Path Space Regularization for Holistic and Robust Light Transport, Eurographics 2013.

[O'Donnell 2017] Yuriy O'Donnell, FRAMEGRAPH: Extensible Rendering Architecture in Frostbite, GDC 2017.

[Purcell 2002] Tim Purcell, Ian Buck, William Mark, and Pat Hanrahan, Ray Tracing on Programmable Graphics Hardware, SIGGRAPH 2002 papers.

[Zirr 2018] Tobias Zirr, Johannes Hanika, and Carsten Dachsbacher, Reweighting firefly samples for improved finite-sample Monte Carlo estimates, EGSR 2018.

[Zwicker 2015], Matthias Zwicker et al., Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering, Eurographics STAR 2015.


NVIDIA