"Practical Path Guiding" in Production

Thomas Müller

Affiliation: NVIDIA

Work done while at: ETH zürich, Disney Research, cgl computer graphics laboratory

Hello everyone!

Around 3 years ago---while I was working at Disney---we first came into contact with path-guiding and we got really inspired by how it was able bridge the gap between simple unidirectional path tracing and some truly advanced bi-directional techniques.

So, naturally, we wanted to take the core ideas behind path-guiding and turn them into a *practical* algorithm that would fit Disney's production needs. We designed an algorithm with the goal of handling complicated geometry, having as few as possible tunable parameters, and not requiring an expensive pre-computation... and this led to a paper at EGSR named "practical path guiding".

"Practica
in P

as Müller

Wo

ter graphics laboratory

Eurographics Symposium on Rendering 2017
P. Sander and M. Zwicker
(Guest Editors)

Volume 36 (2017), Number 4

Practical Path Guiding for Efficient Light-Transport Simulation

Thomas Müller[1,2] Markus Gross[1,2] Jan Novák[2]

[1]ETH Zürich
[2]Disney Research

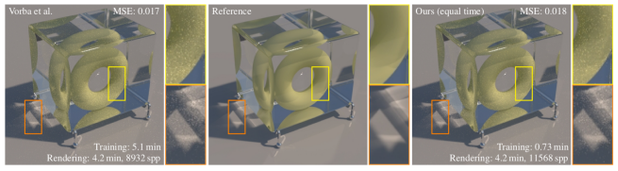


Figure 1: *Our method allows efficient guiding of path-tracing algorithms as demonstrated in the* TORUS *scene. We compare equal-time (4.2 min) renderings of our method (right) to the current state-of-the-art [VKS*14, VK16] (left). Our algorithm automatically estimates how much training time is optimal, displays a rendering preview during training, and requires no parameter tuning. Despite being fully unidirectional, our method achieves similar MSE values compared to Vorba et al.'s method, which trains bidirectionally.*

Abstract

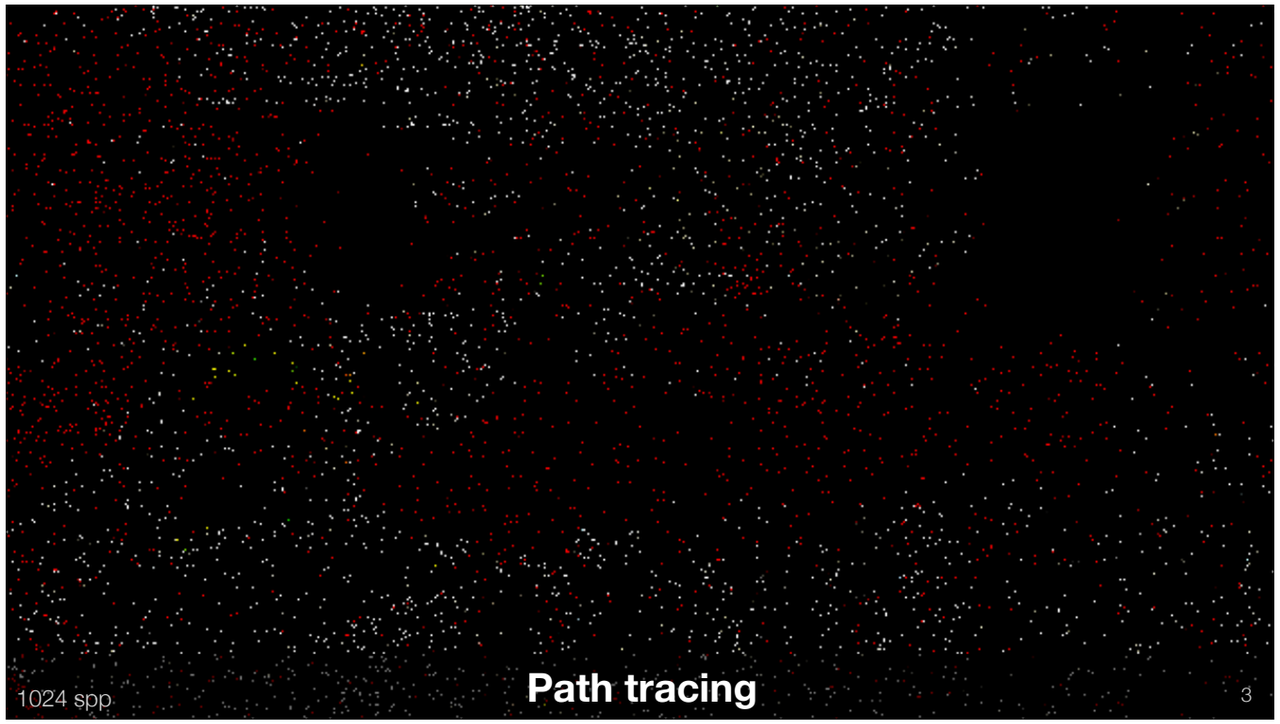
We present a robust, unbiased technique for intelligent light-path construction in path-tracing algorithms. Inspired by existing path-guiding *algorithms, our method learns an approximate representation of the scene's spatio-directional radiance field in an unbiased and iterative manner. To that end, we propose an adaptive spatio-directional hybrid data structure, referred to as* SD-tree*, for storing and sampling incident radiance. The SD-tree consists of an upper part—a binary tree that partitions the 3D spatial domain of the light field—and a lower part—a quadtree that partitions the 2D directional domain. We further present a principled way to automatically budget training and rendering computations to minimize the variance of the final image. Our method does not require tuning hyperparameters, although we allow limiting the memory footprint of the SD-tree. The aforementioned properties, its ease of implementation, and its stable performance make our method compatible with production environments. We demonstrate the merits of our method on scenes with difficult visibility, detailed geometry, and complex specular-glossy light transport, achieving better performance than previous state-of-the-art algorithms.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—; I.3.3 [Computer Graphics]: Picture/Image Generation—

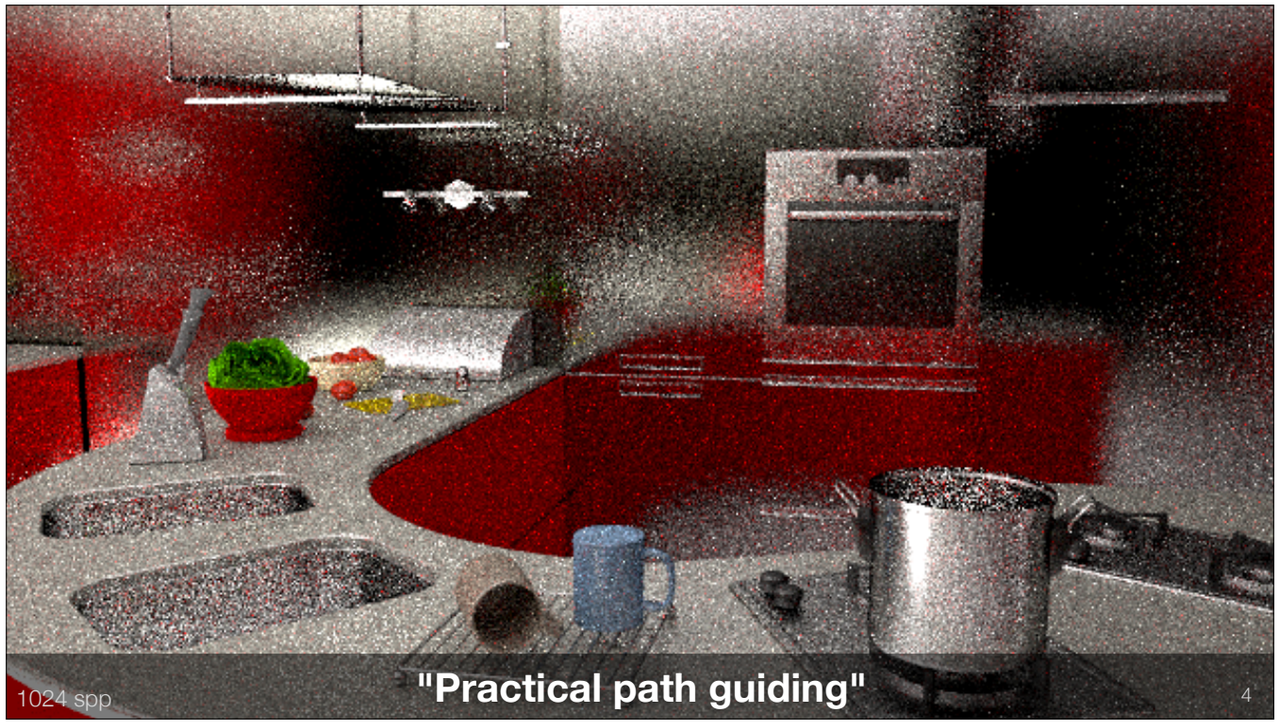
Within Disney, the paper quickly gained some traction---Pixar implemented it in RenderMan, and a little bit later I visited the Walt Disney Animation Studios and worked together with the Hyperion team to integrate it.

While doing that, we ran into a bunch of complications that I imagine many other people trying to implement the algorithm also have. I want to talk about a few of these complications and how we solved them.

These solutions actually turned out to be general, so regardless of which flavor of path-guiding you work with, hopefully they come in handy for you.

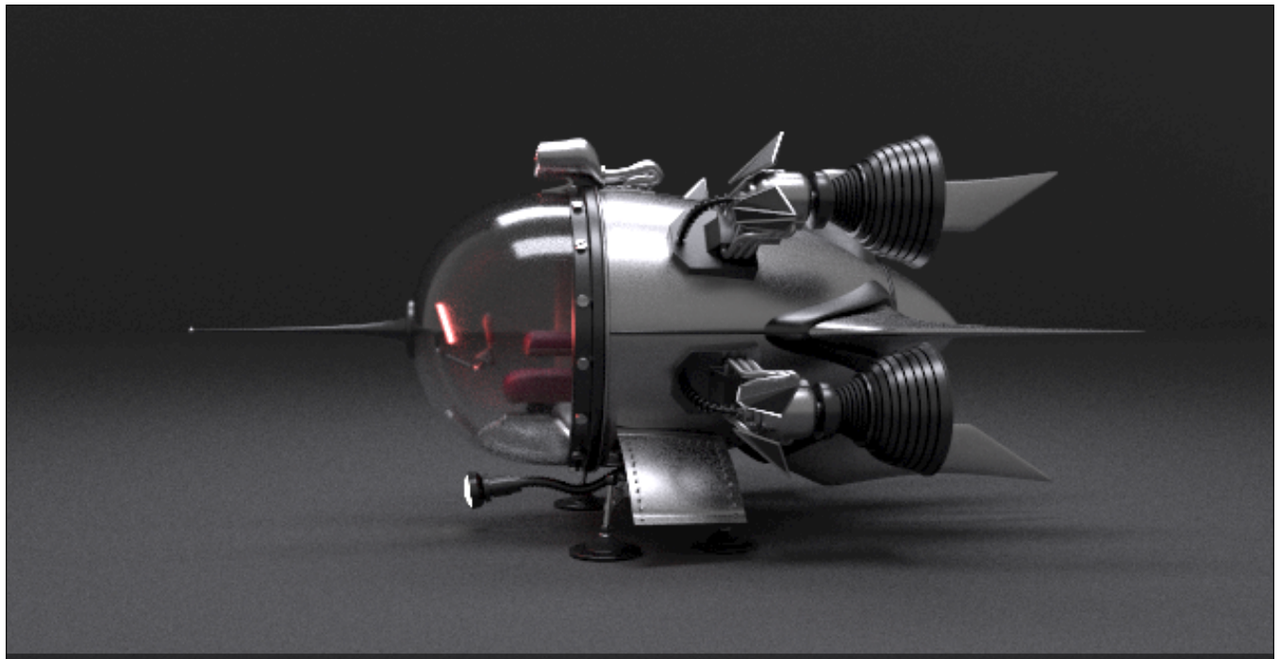


Alright, so this is what we observed after we implemented the original "practical path guiding" algorithm: rendering a really difficult scene, where plain old path tracing is clearly inadequate...



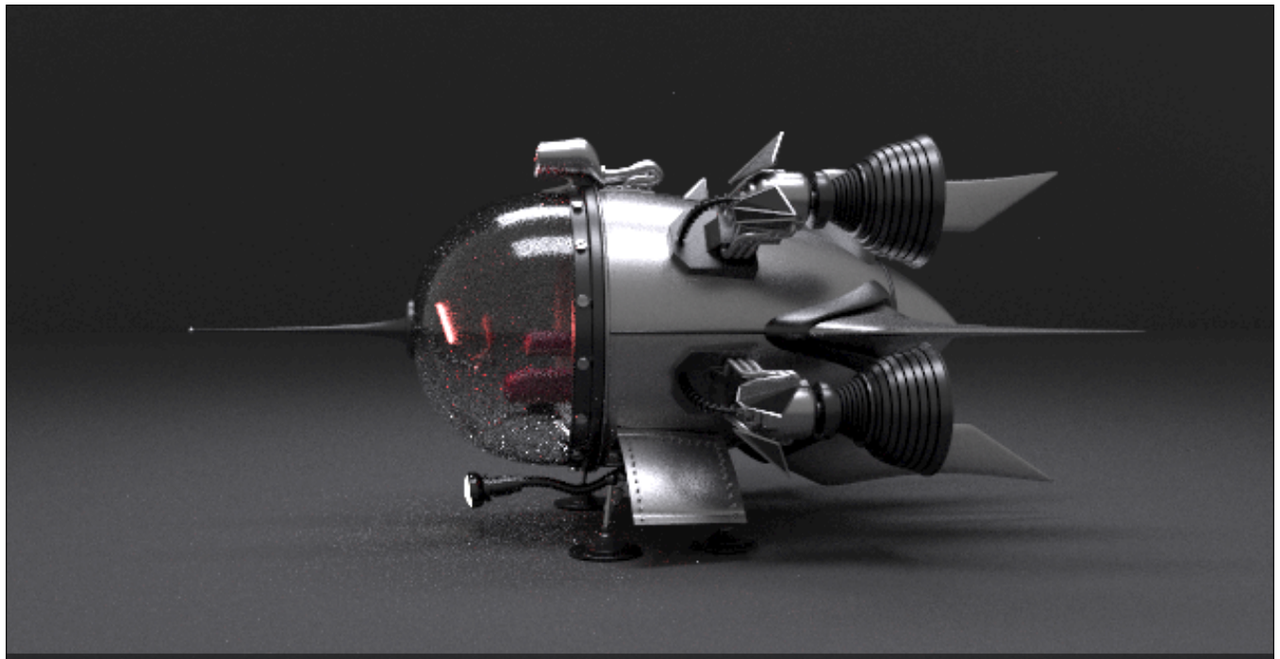
...path guiding helped a lot. So that's good... but many scenes in movie production are actually not so crazy.

That's because Artists *know* that putting glossy materials everywhere and then lighting the scene with multi-bounce indirect illumination asks for trouble, so they typically don't do it.



Instead, often you have easier scenes that are mostly directly lit, using relatively large light sources---like this one. Path tracing is pretty good here.

So, what happens when we turn on "practical path guiding" on such an easy scene. We would *hope* that the rendering get's at least a little better, but in reality...

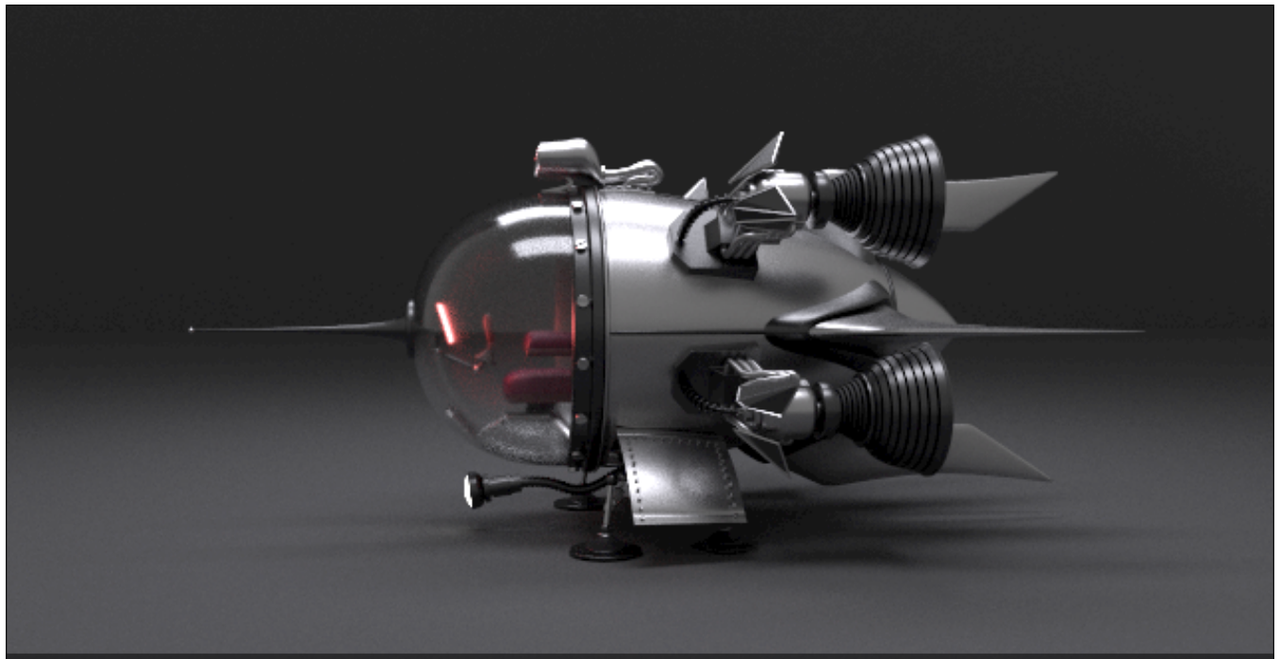


...we'd often get *worse* results! Not so practical after all!

Let me go back and forth between the two images just to illustrate the difference.

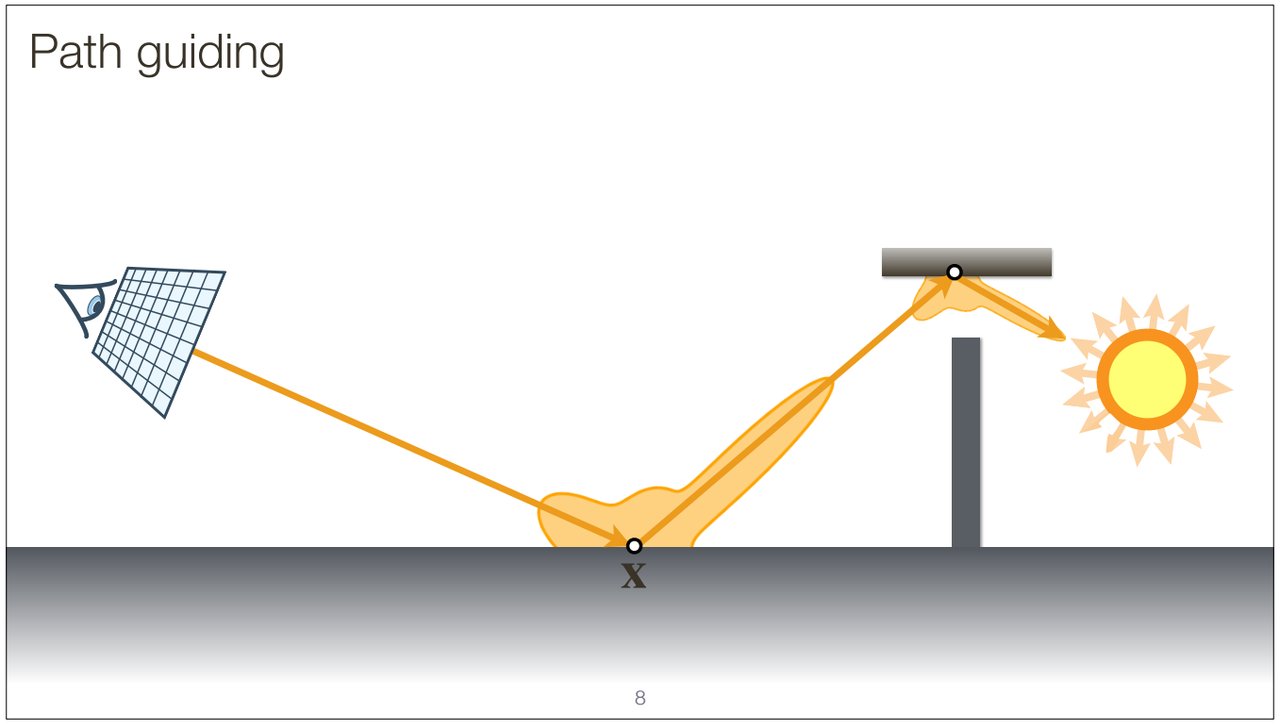
I'll go into detail about *why* this happens later on in the talk---the point I am trying to make here is that we *really don't want* path guiding to worsen cases that were fine beforehand. We want path guiding to be *always on and basically invisible to the user* and that's only possible if we eliminate the situations in which it performs worse than plain old path tracing. That's what I'll focus on in this talk. I'll be presenting **three** improvements to the "practical path guiding" algorithm that at least try to prevent it from becoming worse than the baseline.

Just to tease a little bit the kind of results we get:



This is what the improvements look like here. Now, path guiding actually *improves* over the regular path tracer.

I'll flip between the images to demonstrate this.



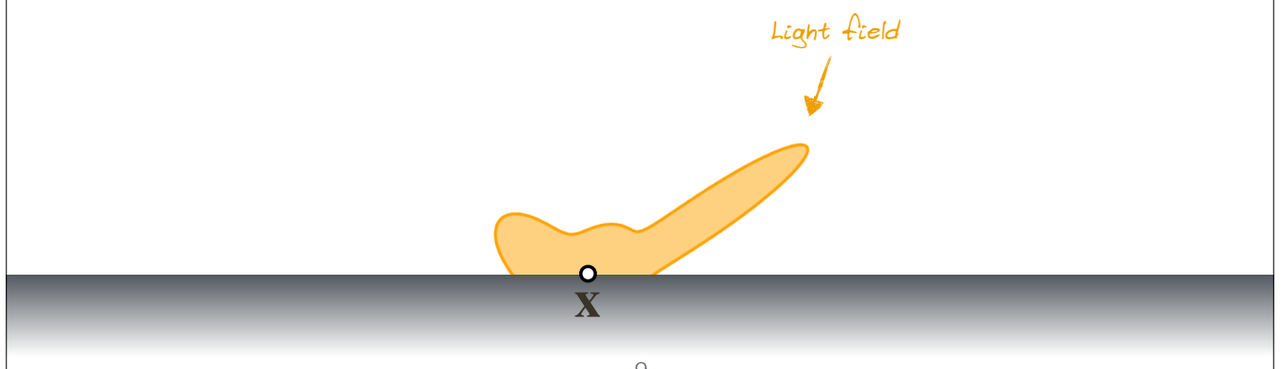
Okay, before we go into more details, let me quickly explain the base algorithm to bring us all on the same page.

As we trace a path, whenever we hit a surface, the path-guiding algorithm gives us a *directional distribution* of where most light is coming from. In order to continue the path, we sample from that distribution and we repeat the process.

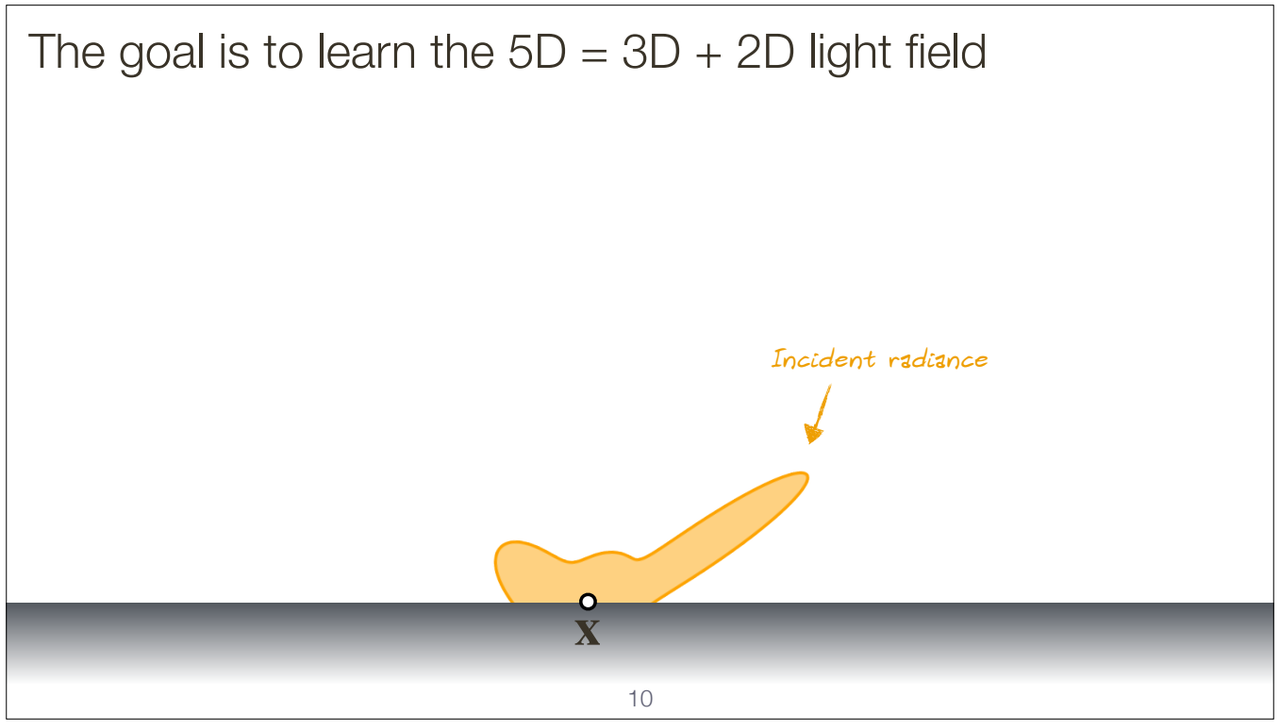
[click]

So for this new location up here, the algorithm gives us a new distribution that might leads us straight to the light source.

The goal is to learn the 5D = 3D + 2D light field

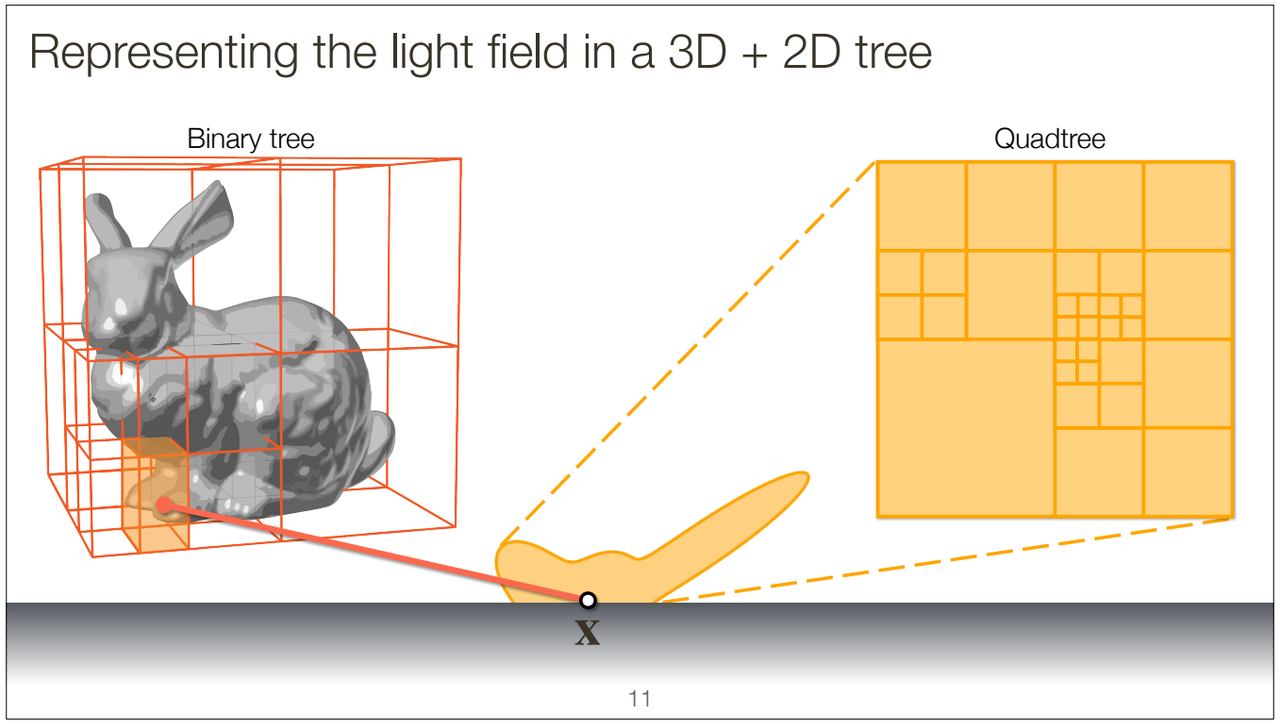


The "practical path guiding" algorithm uses the *light field* for guiding, or in rendering terms...



...the *incident radiance*.

This function has a 3-dimensional *spatial* component [click] and a 2-dimensional *directional* component [click], and these are represent as a...

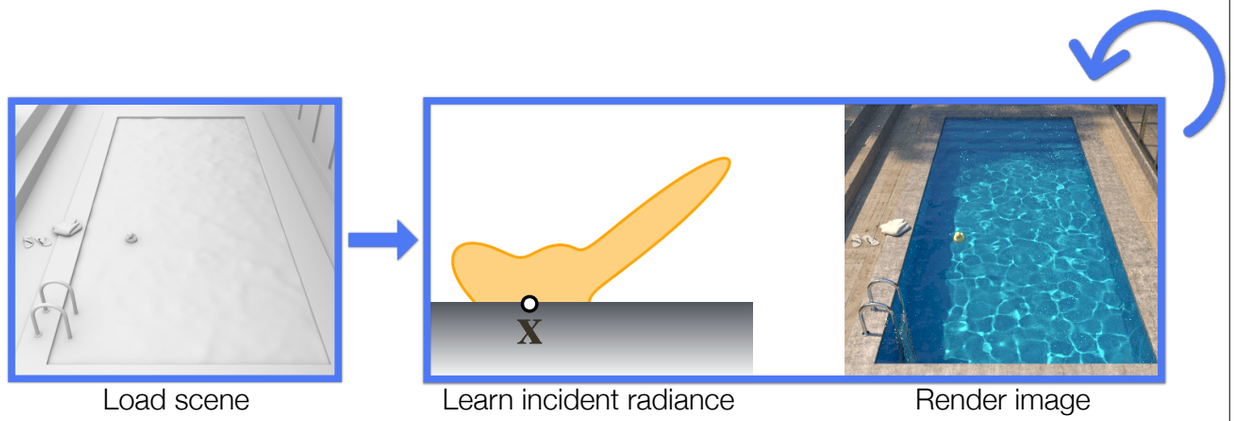


...binary tree that subdivides space, where *each leaf* of the binary tree contains a quadtree that subdivides the directional domain.

This overall tree structure is populated---or... learned---*during* rendering rather than in an expensive pre-computation... so, whenever we complete a light path, we not only record its radiance in the pixel that it started from, but we also also look at the incident radiance at *each vertex* of the path and splat all of these into the tree.

Each of these vertices has a position and a direction in which the light path was continued in... so that's a 5-dimensional coordinate that identifies one particular leaf node in the 5-dimensional tree into which the incident radiance *at* the vertex *from* that particular direction is splatted into.

Iterative learning during rendering

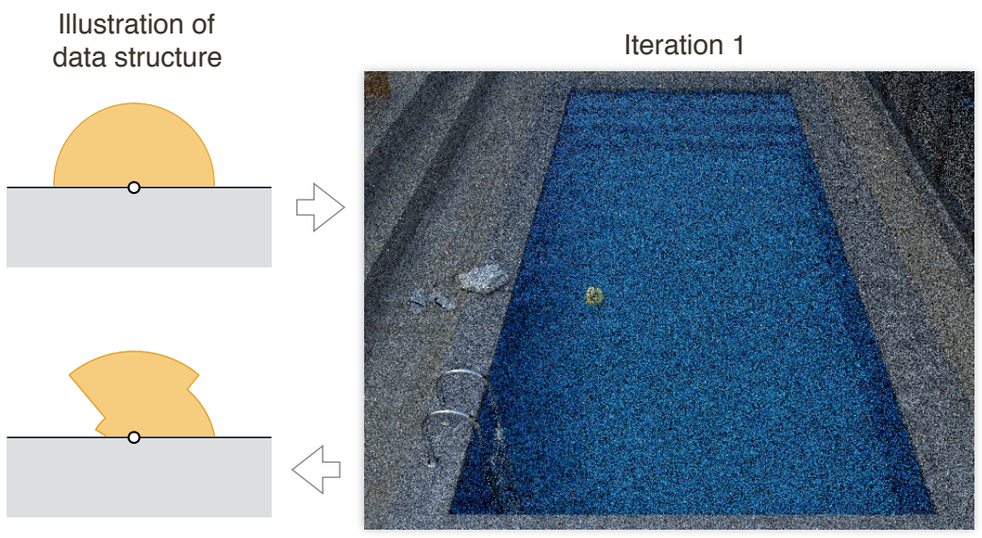


Here I am visualizing this in a flow-chart: the renderer starts by loading the scene and then the incident radiance is learned from the same light paths that are used to render the image---so the image starts rendering immediately; no precomputation---and as it gets less noisy the learned light field also gets more accurate. Future paths will then have even less noise.

Iterative learning during rendering

I'll now demonstrate how concretely this looks like while rendering this image of a swimming pool, where the caustics on the floor of the pool---SDS paths---are the difficult part.

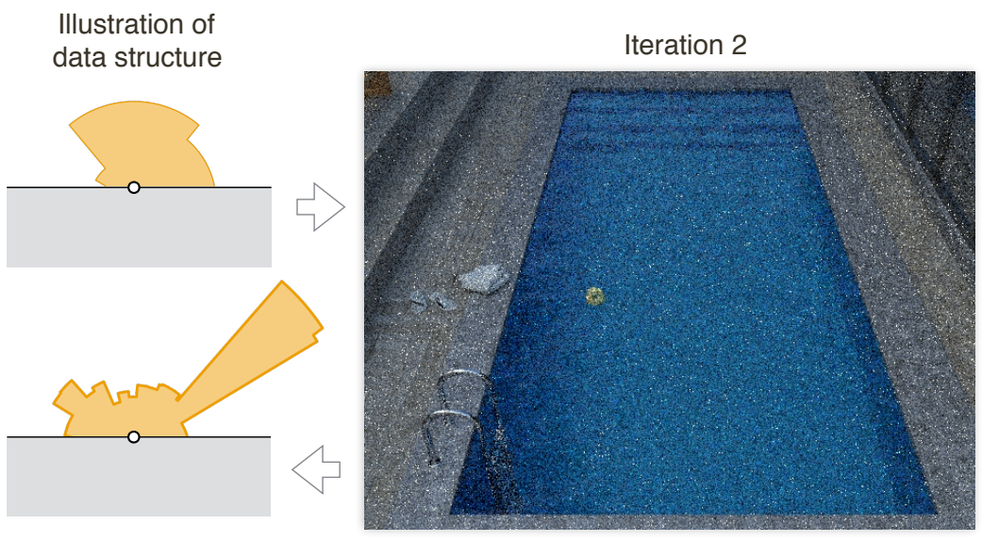
Iterative learning during rendering



We begin by initializing our path-guiding data structure to the uniform distribution and our image buffer to black.

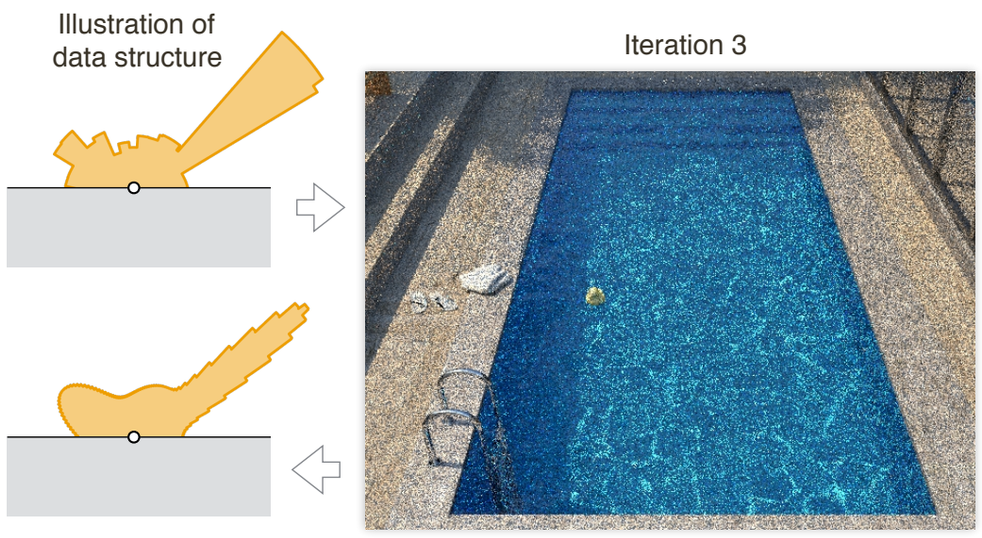
Then, we simulate some predetermined number of paths and use them to estimate not only the pixel values, but also a slightly better data structure for guiding in the future.

Iterative learning during rendering



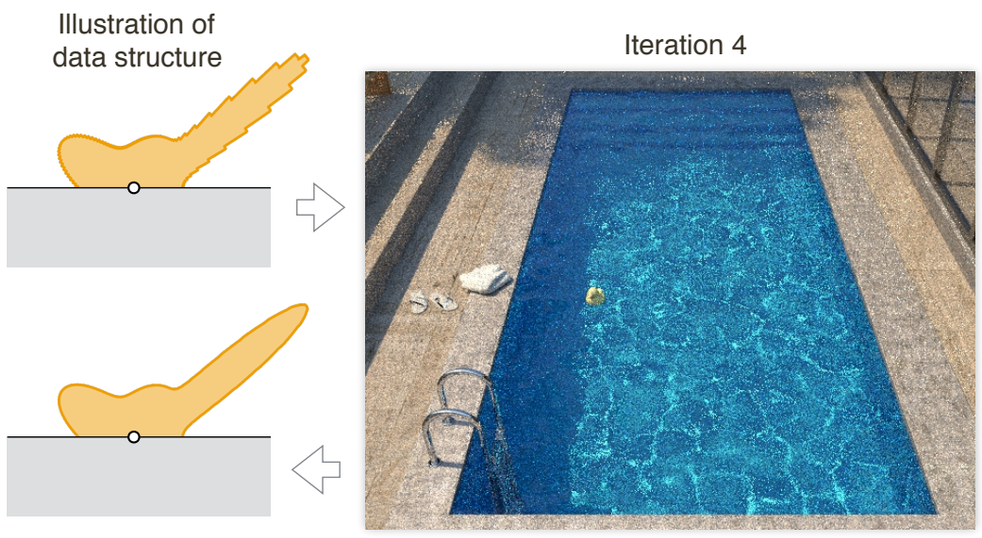
Then, in the next iteration, we *reset* the image to black and then use the better guiding data structure from the previous iteration to drive the simulation of additional light paths, which gives us new pixel values and produces an even better guiding data structure.

Iterative learning during rendering



We keep repeating this process---clearing the image and using the latest guiding data structure to trace the next paths---and we keep doing this until we have an image we are satisfied with. Then we stop rendering.

Iterative learning during rendering

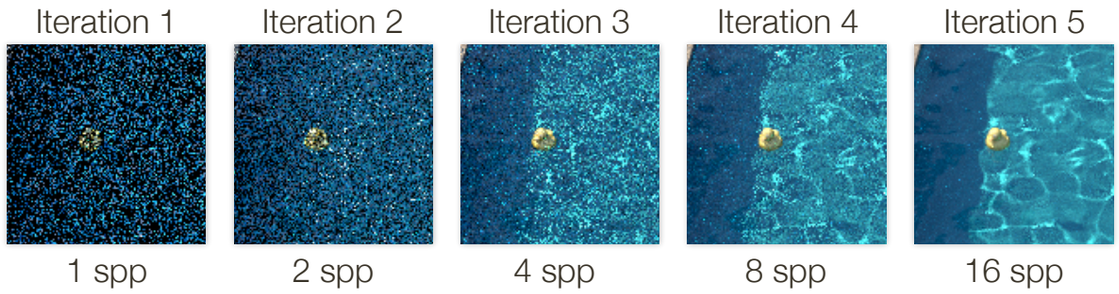


Let's talk about why we threw away the previous images. The reason is actually *not* to make the algorithm unbiased. The algorithm would be unbiased, even if we kept the previous images and averaged them out. The actual reason is that since we're creating the next image with a *better* guiding distribution, this next image might have so much less noise, that averaging it with the previous images would often give *higher* variance, because there are potentially fireflies in those previous images, or just more noise in general!

This is the case when the scene is hard to render and guiding is really important... *but sometimes the opposite can also be true*.

How to combine iterations?

[Disclaimer: the numbers in this slide are made up. Otherwise it'd be difficult to see the noise difference.]

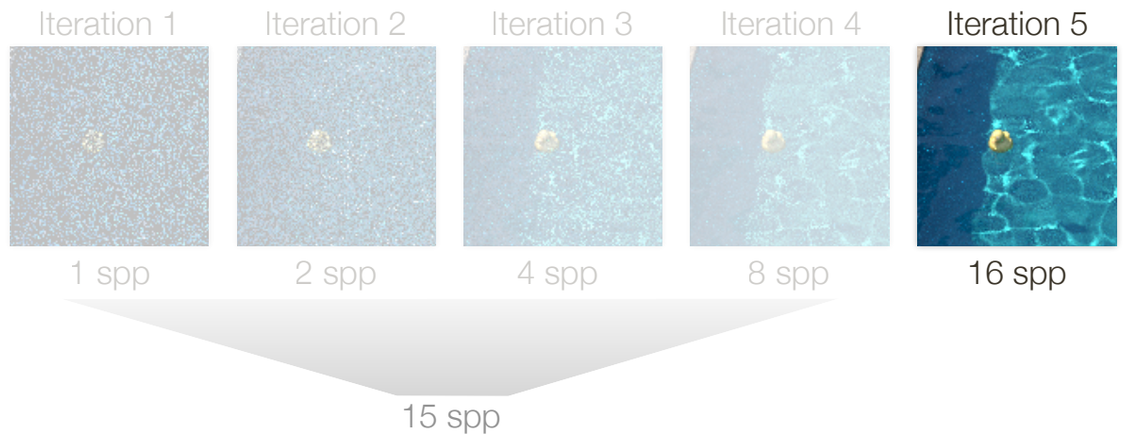


Let's look at the worst case that can happen with this scheme.

Suppose we got these images here from our iterations---and suppose that each consecutive iteration uses *twice* the number of samples as the previous iteration---that's what the "practical path guiding" algorithm does.

Original algorithm discards up to 50% of samples

[Disclaimer: the numbers in this slide are made up. Otherwise it'd be difficult to see the noise difference.]



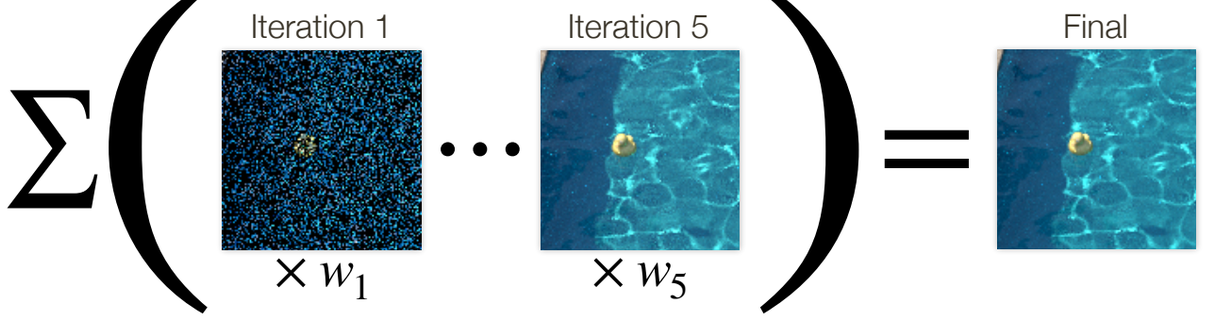
What you've seen before---throwing away all but the last iteration---amounts to discarding roughly *half* of the total sample count.

Now, this is not much of a problem if the first half of the samples was really noisy compared to the second half---for example because it took a long time for path guiding to learn something useful---because then we wouldn't want to use the initial noisy samples anyway.

But consider the case where path guiding learns really quickly! Then, those initial samples are essentially wasted for no good reason. So the worst-case situation is that we make rendering half as efficient as before, and while this is not *terrible*, it's still quite undesirable.

So... instead of either keeping all the images, or throwing them all away... is there perhaps a middle ground that gives us the best of both worlds?

Better: weighted average of iterations



Optimal weights? Inverse-variance of the samples! [Graybill and Deal 1995]
Details in course notes.

Turns out, there is! The idea is to build a *weighted average* of the iterations---so we sum up the images, but we weight them by scalars that some up to 1.

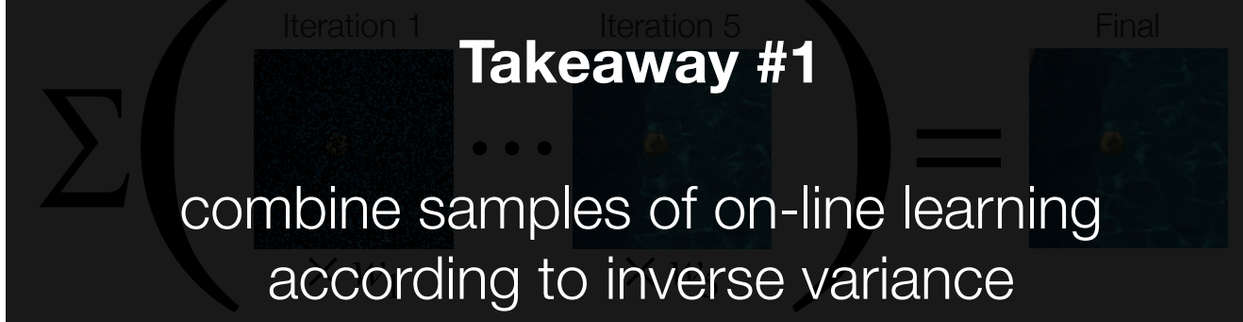
Figuring out the optimal weights is actually quite straightforward. Intuitively, if the variance of an iteration is large---so if that particular image was very noisy---then it should get a low weight, whereas if the variance of an iteration was low---that is, it had little noise---then it should get a large weight.

Mathematically, this amounts to weighting each iteration by its *inverse variance*... and this scheme actually results in the *least* variance in the combined result. And this scheme is a generalization of the two options we had before... if all the previous iterations had infinite variance, then the weights would be zero and we'd effectively throw them all away... and if the previous iterations all had the same variance per sample as the last one, then all the samples would be simply averaged.

Unfortunately, we don't actually know the exact variances of each iteration, but we can numerically estimate the variances during rendering which gives us accurate enough weights to perform this scheme.

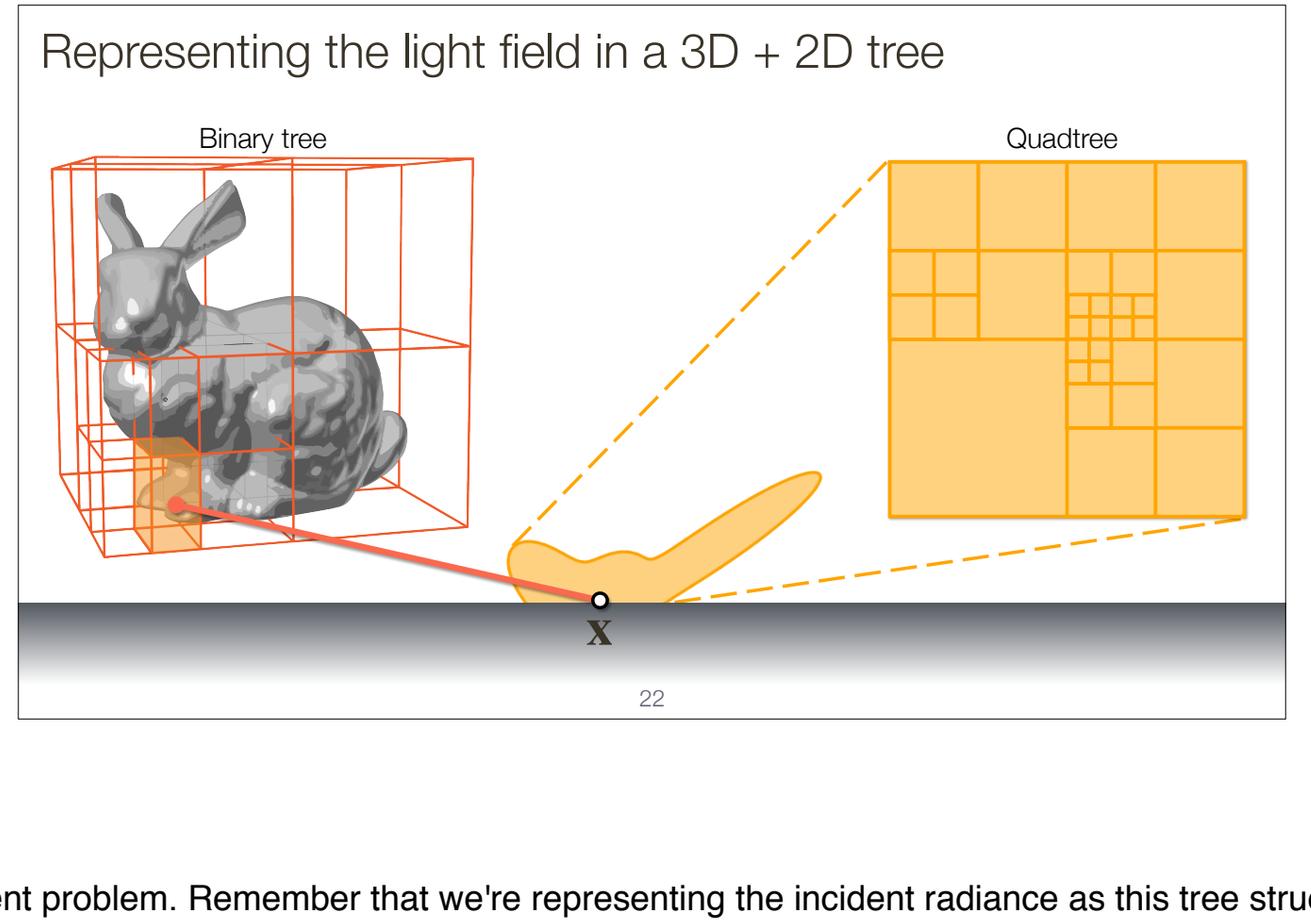
There are a few details involved in doing this without introducing a lot of bias, but I am referring you to the course notes for those details. The general idea is the important part.

Better: weighted average of iterations



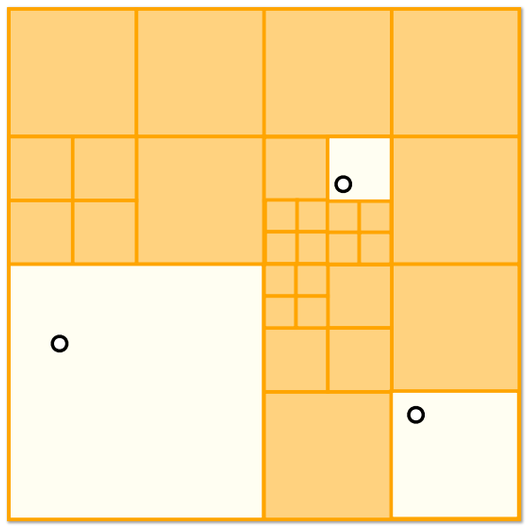
Optimal weights? Inverse-variance of the samples.
Details in course notes!

So to summarize---and this is the first take-home message of this presentation---when you learn on-line *during rendering*, then combining the samples according to their inverse variance gives you the best results, *regardless of whether the scene is difficult and path-guiding super important, or whether the scene is easy and path guiding does not help at all.*



Okay, let's context-switch to a different problem. Remember that we're representing the incident radiance as this tree structure, and we're populating it by splatting the radiance at path vertices into it.

Original algorithm: nearest-neighbor splatting

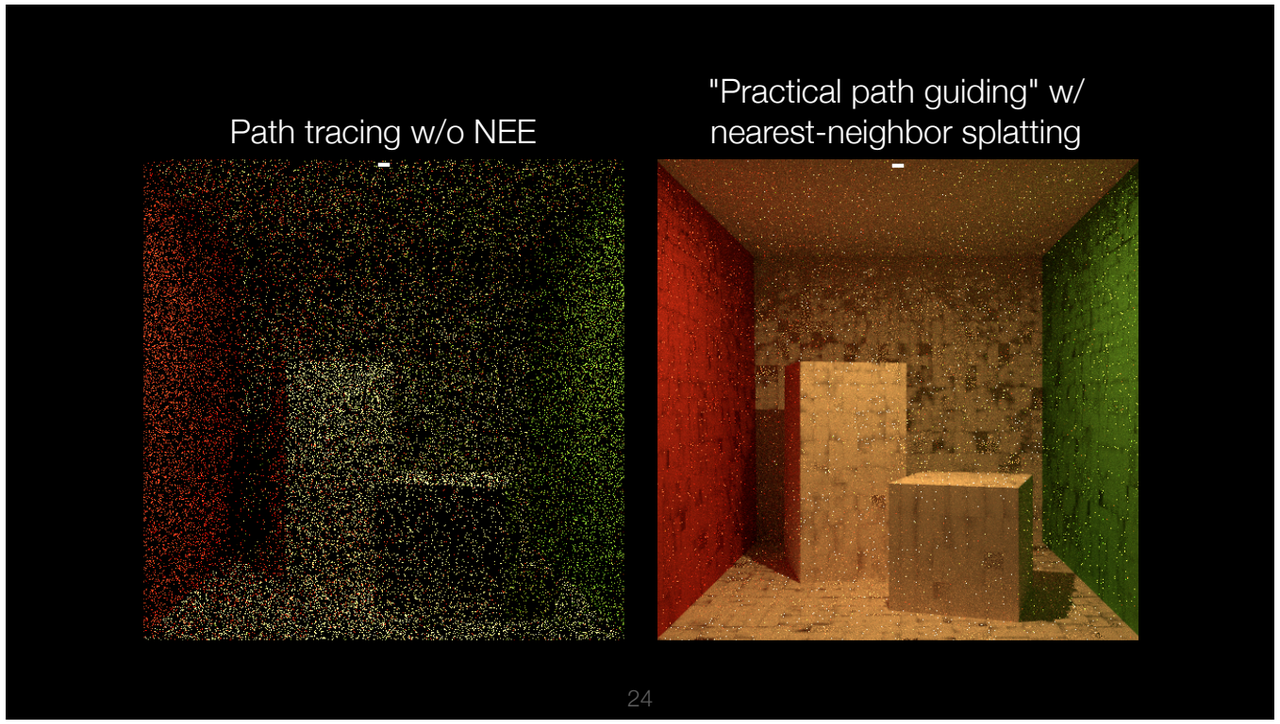


For the sake of visualization, let's look at just part of the tree. Whenever a sample comes in, the original algorithm would just deposit its radiance in whatever leaf the sample fell into.

So if another sample came in here [click], we'd record it in this leaf node, and if yet another sample came in here [click], we'd record it in this leaf node.

This means: we are doing *nearest-neighbor* splatting!

Unfortunately, we found some cases where this nearest-neighbor splatting does not work out so well.



Let me show you. Here's a Cornell box with a *tiny* light source and next-event estimation turned *off*.

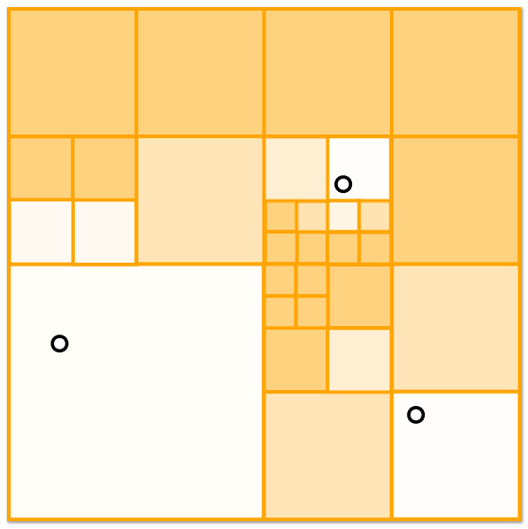
...and I'd just like to interject that you should probably *never* actually turn off next-event estimation---I am just doing it here to exaggerate the problem.

Anyway, a regular path tracer does pretty badly in this situation... "practical path guiding" with the nearest-neighbor splatting reduces that noise quite a lot, but it also produces *really ugly* artifacts. These artifacts are *not* bias, they are just *huge, sudden* variations in noise that are caused by the nearest-neighbor splatting into the tree data structure.

There are a number of subtle reasons *why* nearest-neighbor splatting causes these artifacts---they are discussed in detail in the course notes---but the gist of it is that nearest neighbor splatting leads to *non-uniform* learning of the incident radiance in some cases.

So we'd like to spread out the learning more uniformly across the tree structure. And the perfect tool for this is *filtering*.

Improvement: filtered splatting



So... how can we do filtered splatting into a tree with varying resolution?

It's actually really straightforward.

Whenever we want to splat a new sample, instead of splatting it into the nearest node [click], we take the node's footprint and center it around our sample. This is going to be our *filtering footprint*. So if the sample would have landed in a higher-resolution region of the tree, we would filter in a smaller region, and vice versa.

Then, we *distribute* the sample's radiance according to the *area overlap* of the footprint with its neighboring nodes [click].

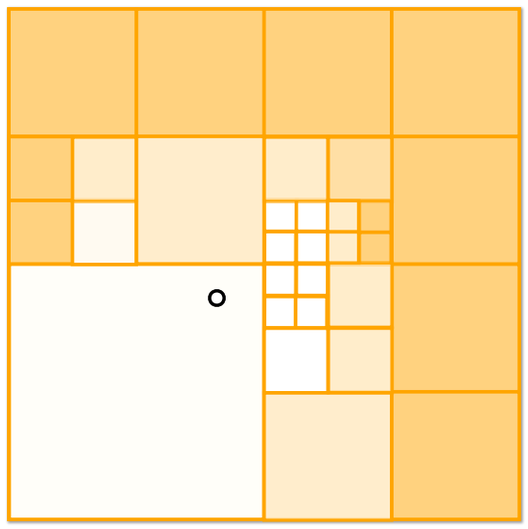
In this example, these two smaller nodes receive a larger density of radiance than the bigger nodes, because a larger portion of their area overlaps the footprint.

And that's it!

So if we got another sample here [click], it would be splatted like this.

And another sample here [click] would be splatted like this.

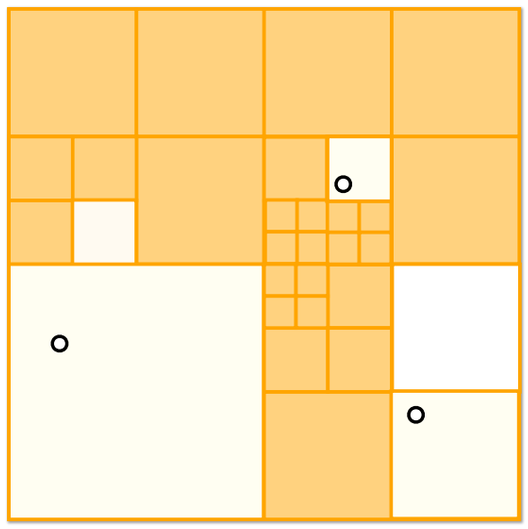
Improvement: filtered splatting



There's a catch, though. This filtering operation can become quite computationally expensive, because instead of touching a single node, it needs to touch *all* the overlapping nodes.

So when a sample falls into a large leaf that's right next to a high-resolution region, then the filter needs to visit a potentially very large number of other nodes.

Improvement: stochastic filtered splatting

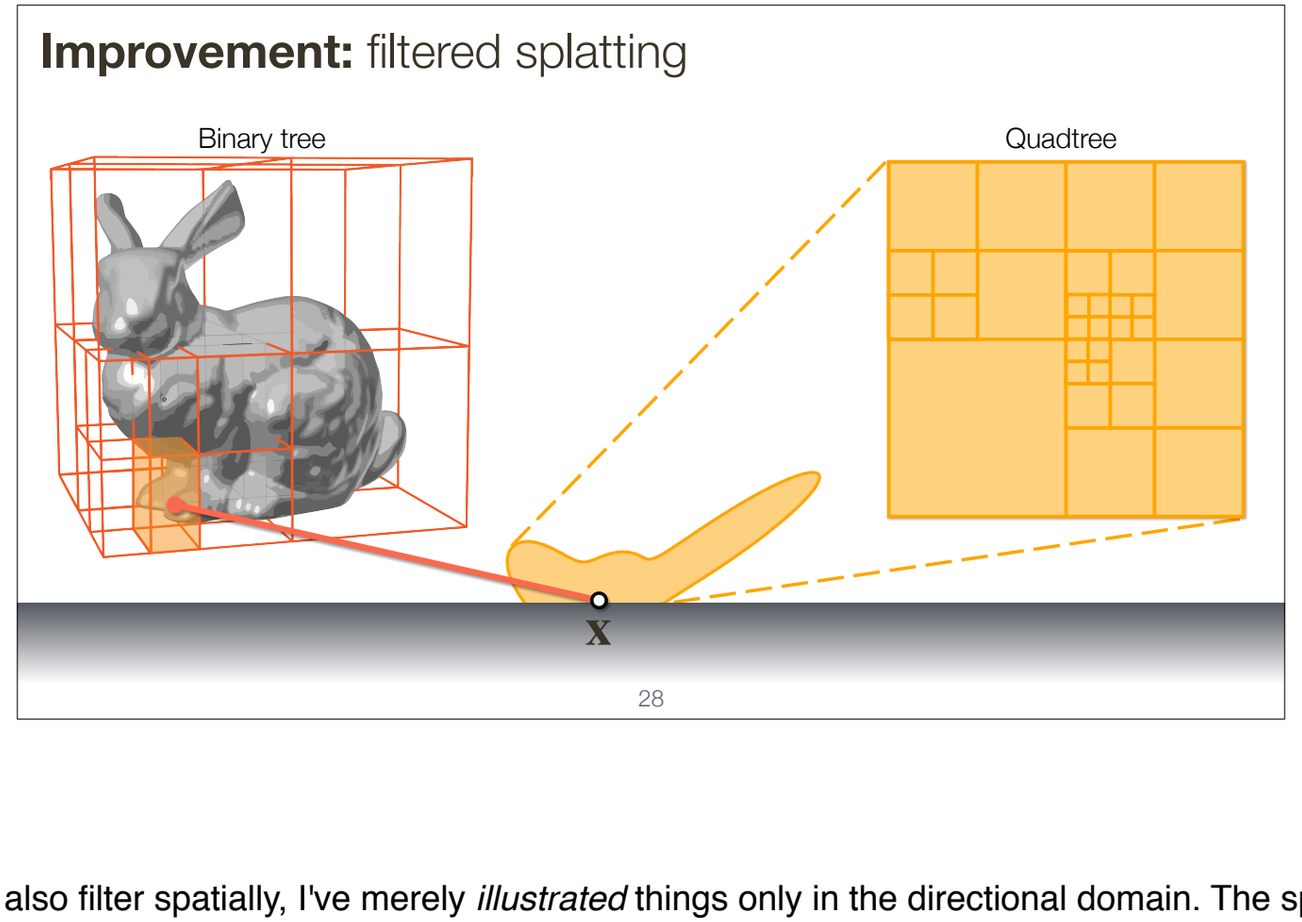


There's a way around that, though: the filtering can be done stochastically! It starts out the same as before [click]---we pick the node the sample fell in as the filter footprint and center it around the sample---but then instead of spreading out the radiance deterministically, we *randomly jitter* the sample within the footprint [click] and we splat *only* into the node that the sample ends up in [click].

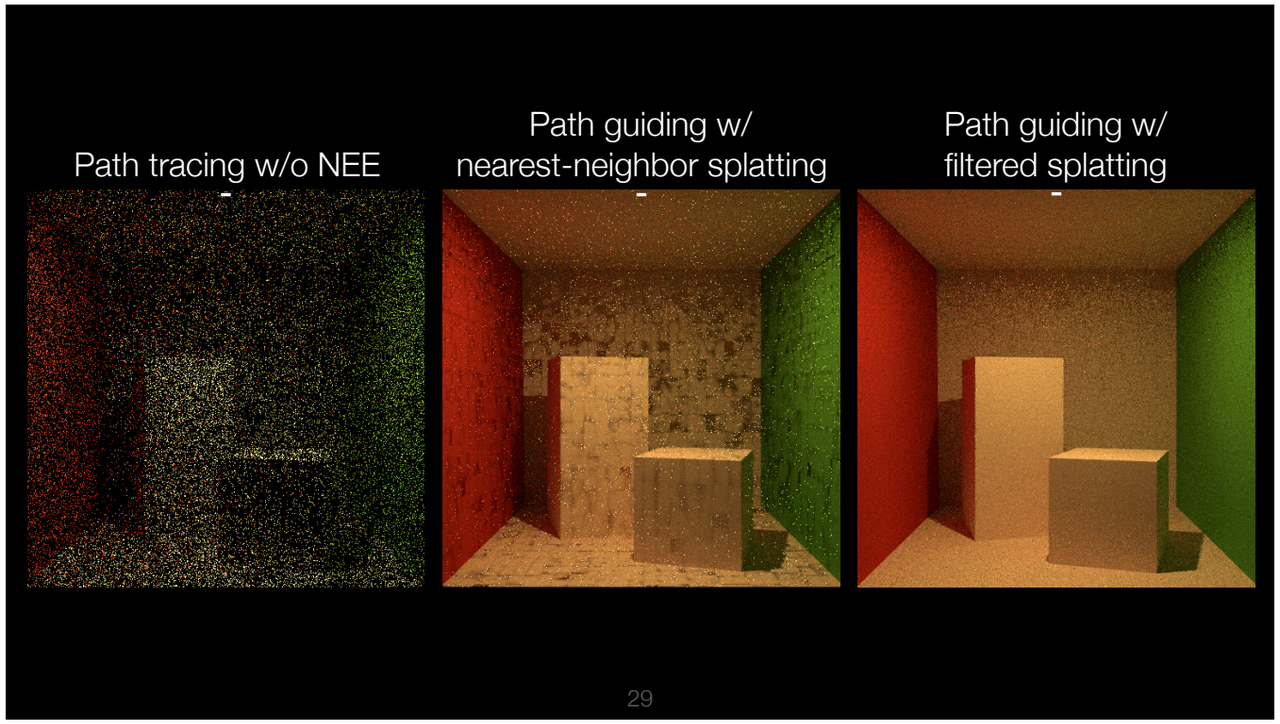
Here is an example where the sample ends up in the same node by chance [click].

And here is another example [click].

This approach only needs *two* look-ups for each splat, so it is cheap, but it gives *exactly the same* results in *expectation* as the deterministic filtering approach from before. So this allows trading performance for a little bit of additional noise.

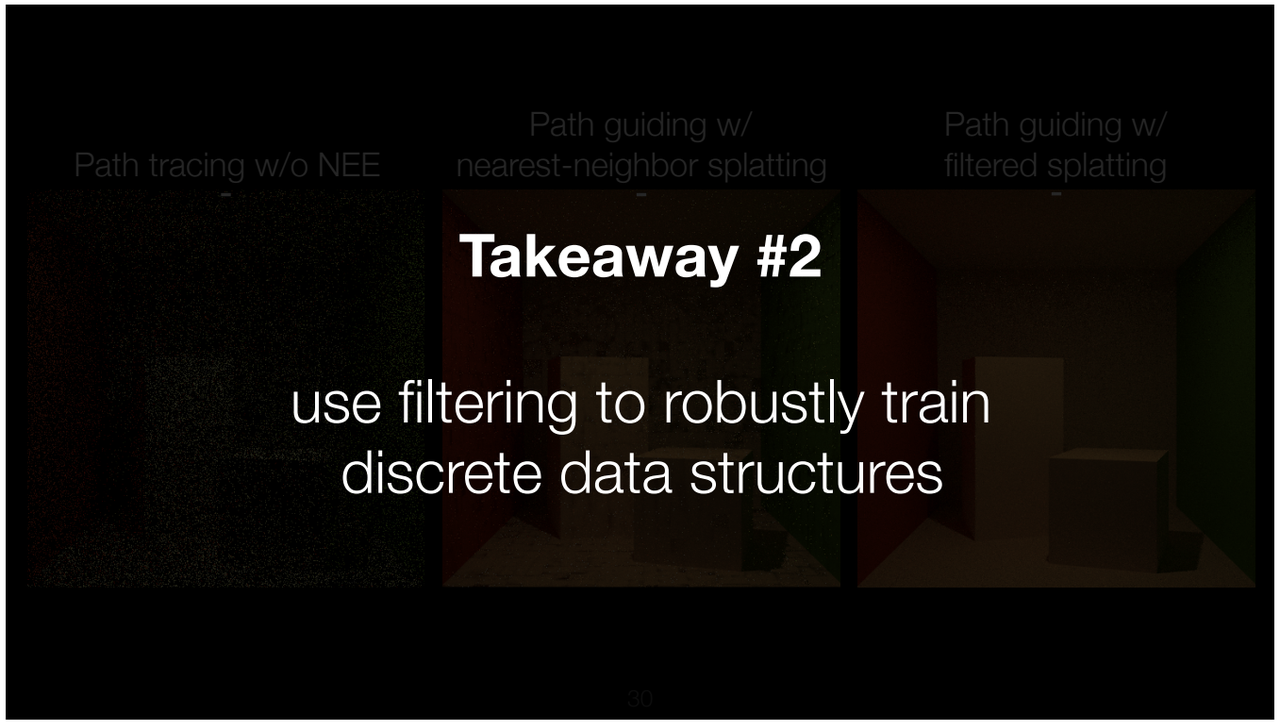


And just keep in mind that we would also filter spatially, I've merely *illustrated* things only in the directional domain. The spatial filtering works in exactly the same manner, only that we don't filter over areas, but instead over volumes.

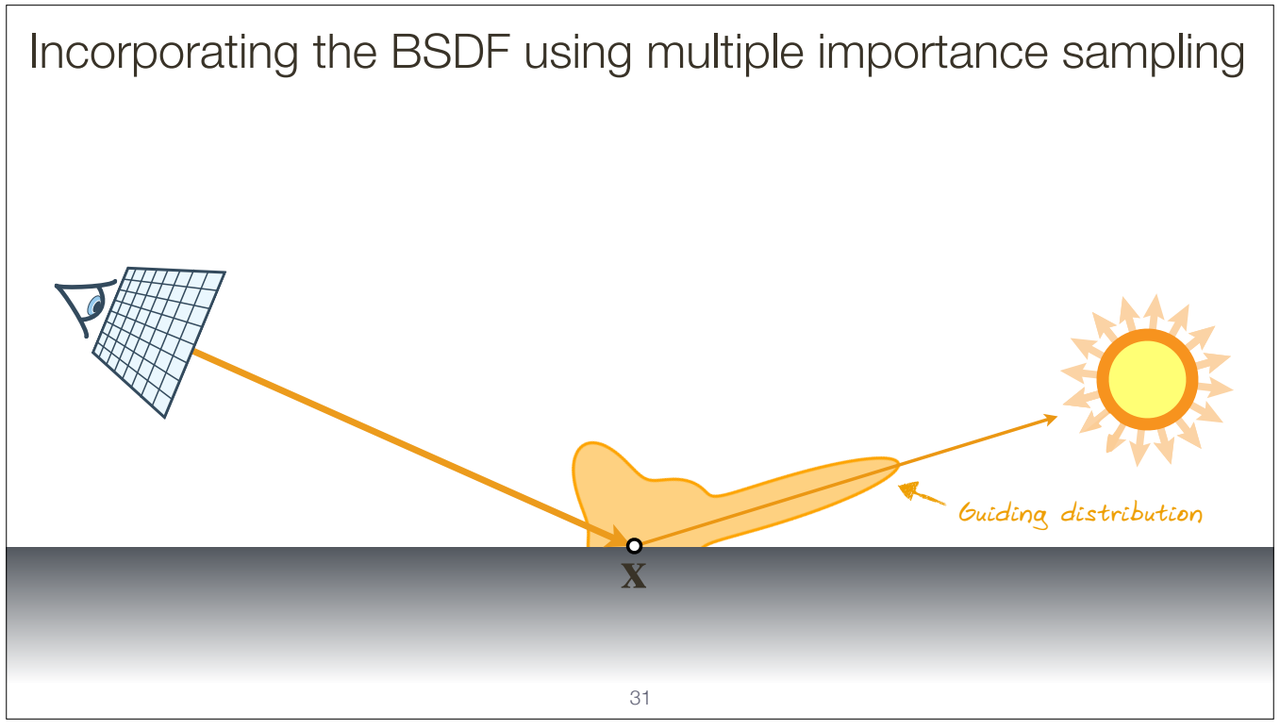


Okay, so this is how it looks like in the end. Here's the Cornell box from before, where regular path tracing performs pretty badly, and nearest neighbor sampling has ugly artifacts.

The filtering [click] removes these artifacts almost entirely. When I first saw these improvements I actually couldn't believe the large difference myself.



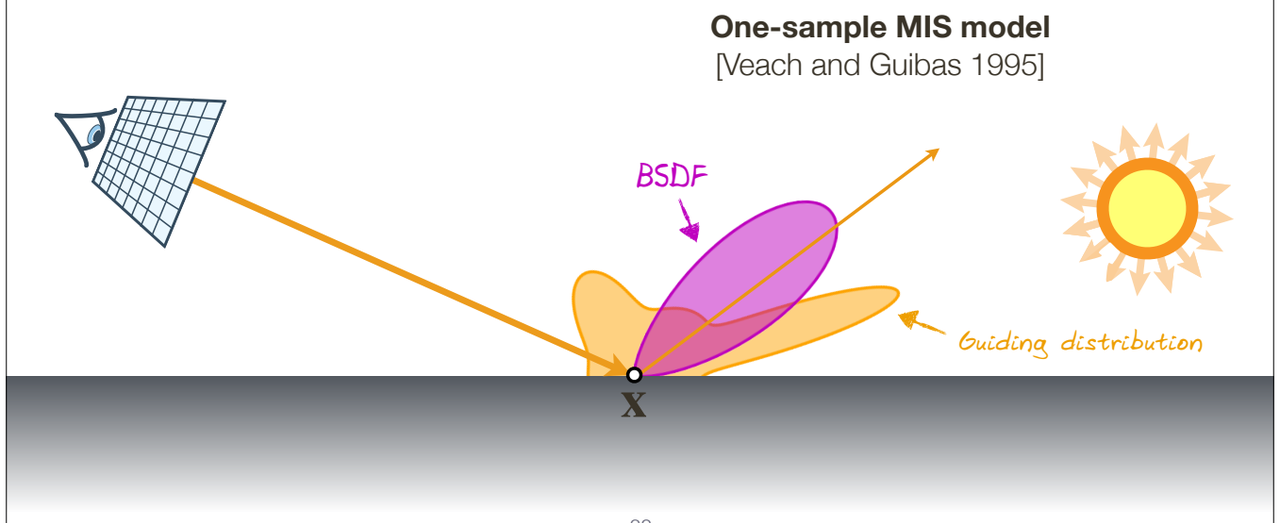
So the take-home message here is that filtering is *amazingly important* when using discrete data structures in a path-guiding setting. I know, it's common wisdom that one should filter whenever possible, but in this particular case it's more important than any others I've experienced in the past.



Okay, the last thing I want to talk about is how guiding is combined with BSDF sampling.

When reaching a vertex, we can either use the learned guiding distribution to sample the next direction... *or*...

Incorporating the BSDF using multiple importance sampling



...we can sample the next direction according to the BSDF, which is what a regular path tracer would do.

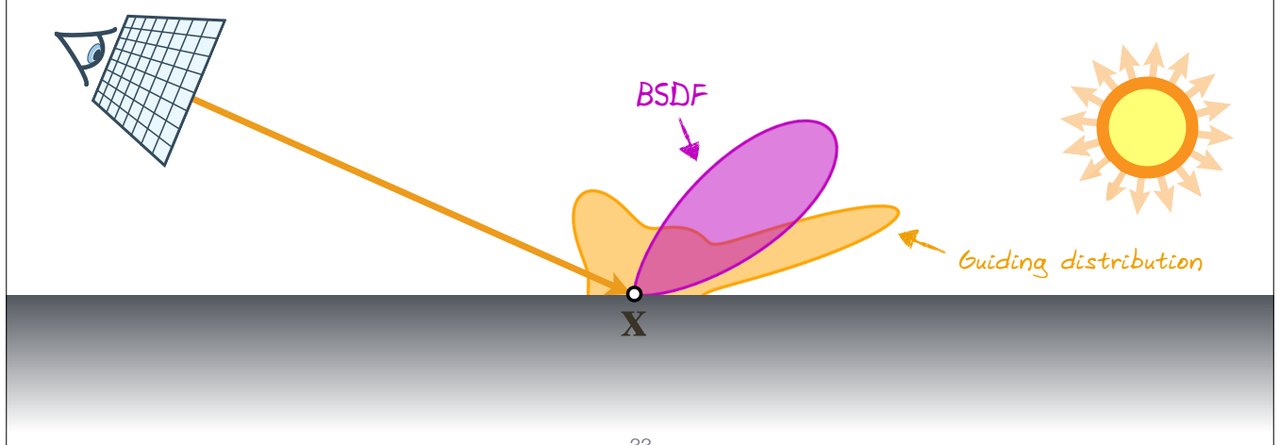
In practice, *either* of these techniques can be better, so pretty much all path-guiding algorithms out there *combine* BSDF sampling with guiding using the one-sample multiple-importance-sampling model.

Multiple importance sampling has two aspects to it: first, the samples are weighted according to some heuristic---most people use the balance heuristic in the one-sample model because there are certain optimality guarantees. Second, there is the question of how the samples are distributed between the techniques---so in this case how many samples should follow BSDF sampling and how many samples should be guided.

We focus here *only* on the second aspect: what should be the probability to select BSDF sampling versus the probability to select guiding?

Incorporating the BSDF using multiple importance sampling

$$p_{\text{final}}(\omega_{\text{i}}, \alpha) = \alpha\, p_{\text{BSDF}}(\omega_{\text{i}}) \;+\; (1 - \alpha)\, p_{\text{guide}}(\omega_{\text{i}})$$



Let's look at this a little more formally.

The *final* directional density is a linear blend between the BSDF density and the guiding density, where the selection probability---alpha---controls whether selecting the BSDF or guiding is more likely.

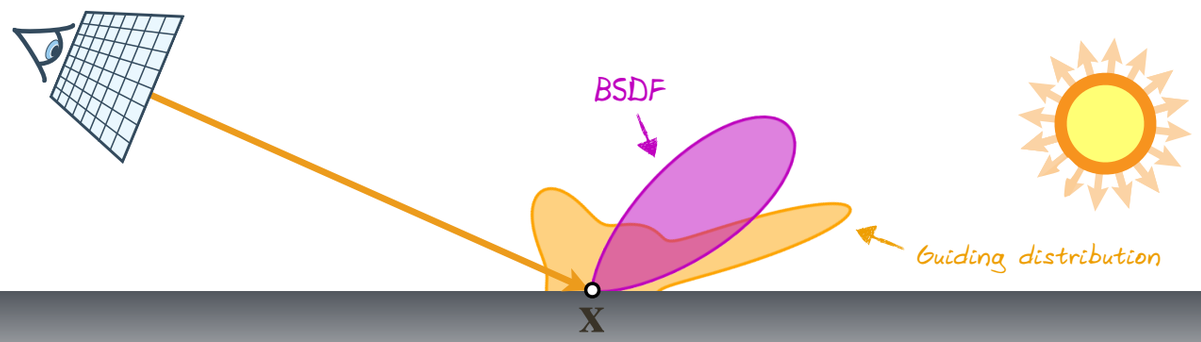
Most existing guiding algorithms simply set the selection probability to a fixed value like 50%---so there's a 50:50 chance of doing BSDF sampling or guiding.

What we are interested in is whether we can choose this selection probability more optimally... and ideally depending on where we are in the scene.

How to choose the selection probability α?

$$p_{\text{final}}(\omega_\text{i}, \alpha) = \alpha\, p_{\text{BSDF}}(\omega_\text{i}) \;+\; (1-\alpha)\, p_{\text{guide}}(\omega_\text{i}) \propto L(\omega_\text{i}) f(\omega_\text{i}) \cos\theta_\text{i}$$

Integrand of the rendering equation



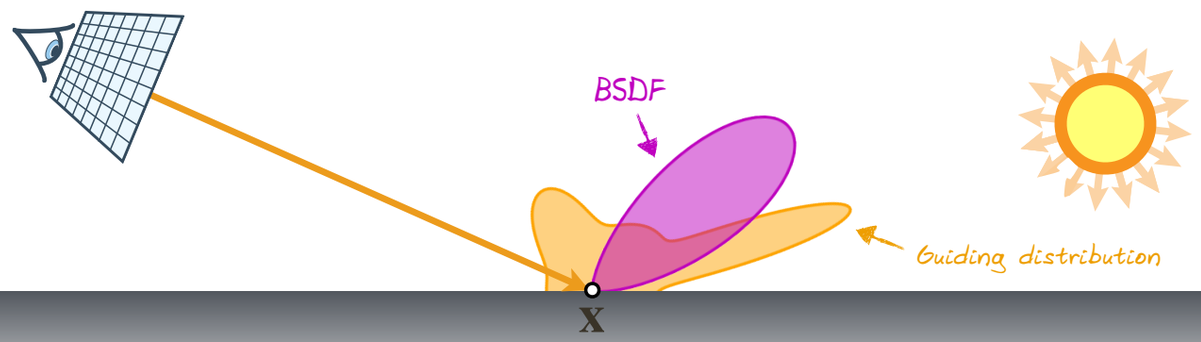
To do this, let's first look at what our goal is in the first place.

We want the final PDF to be as proportional as possible to the integrand of the rendering equation---the product of incident radiance, the BSDF, and the foreshortening term---because the better we sample the product, the lower our variance will be.

How to choose the selection probability α?

$$p_{\text{final}}(\omega_i, \alpha) = \alpha\, p_{\text{BSDF}}(\omega_i) + (1-\alpha)\, p_{\text{guide}}(\omega_i) = p_{\text{ideal}}(\omega_i)$$

"Ideal" PDF proportional to integrand



In other words, we want our final density to approximate an "ideal" density that is *exactly* proportional to the integrand.

How to choose the selection probability α?

Goal: $p_{\text{final}}(\omega_i, \alpha) = p_{\text{ideal}}(\omega_i)$

"Ideal" PDF proportional to integrand

Idea: frame as optimization problem.

Minimize: $D(p_{\text{ideal}} \,\|\, p_{\text{final}}\,; \alpha)$ such that $\alpha \in [0,1]$

So how do we get there? Clearly, just by linearly combining BSDF sampling and guiding, we can not perfectly match the product... but we *can* try to come as close to it as possible.

This is the important insight: we can frame it as an optimization problem.

We want to *minimize* some *divergence function* between the ideal density and our final density by tweaking the selection probability **alpha**. And, of course, we need the probability to be between 0 and 1.

How to choose the selection probability α?

Minimize: $D(p_{\text{ideal}} \,\|\, p_{\text{final}}\,; \alpha)$ such that $\alpha \in [0,1]$

if $D(p_{\text{ideal}} \,\|\, p_{\text{final}}\,; \alpha)$ is Kullback-Leibler divergence:

Stochastic gradient descent using:

$$-\frac{L(\omega_\text{i}) f(\omega_\text{i}) \cos\theta_\text{i}}{p_{\text{final}}(\omega_\text{i}, \alpha)} \cdot \frac{p_{\text{BSDF}}(\omega_\text{i}) - p_{\text{guiding}}(\omega_\text{i})}{p_{\text{final}}(\omega_\text{i}, \alpha)}$$

So... under this formulation, what would be a good divergence function to use?

Thankfully, this is something we already explored in a different setting, where we use *neural networks* for path guiding---and... if you're interested in that, I'll present it on Thursday.

What we found is that using the Kullback-Leibler divergence gives the best and most robust overall results. Sparing you all the math---it's in the course notes---we can solve this optimization problem using stochastic gradient descent on *alpha,* where the gradient is this particular expression.

It looks a little overwhelming at first glance, but let's look at the individual terms, which actually lead to a quite intuitive interpretation.

How to choose the selection probability α?

Minimize: $D(p_{\text{ideal}} \,\|\, p_{\text{final}}\,; \alpha)$ such that $\alpha \in [0,1]$

if $D(p_{\text{ideal}} \,\|\, p_{\text{final}}\,; \alpha)$ is Kullback-Leibler divergence:

Stochastic gradient descent using:

$$-\frac{L(\omega_i) f(\omega_i) \cos\theta_i}{p_{\text{final}}(\omega_i, \alpha)} \cdot \frac{p_{\text{BSDF}}(\omega_i) - p_{\text{guiding}}(\omega_i)}{p_{\text{final}}(\omega_i, \alpha)}$$

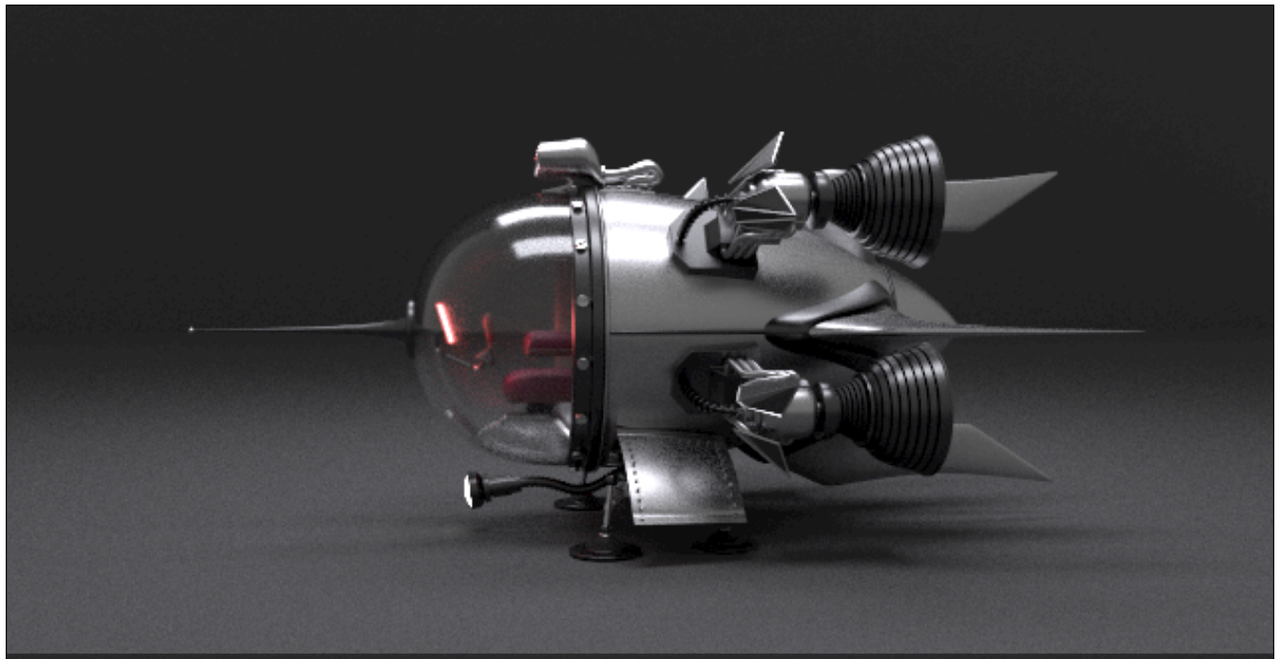
First, let's ignore the probabilities in the denominator. They're just there for making Monte Carlo work out. The interesting part is in the numerator.

On the left-hand side, we have the integrand of the rendering equation... so that means the gradient is larger when a large amount of light is being reflected, and smaller when little light is reflected. This means that gradient descent places more importance on more light, and less importance on little light---that makes intuitive sense.

Then, on the right-hand side, we have the *difference* between the BSDF density and the guiding density. So, this means if BSDF sampling places a larger probability on a direction than path guiding, the gradient is positive and gradient descent nudges the selection probability towards BSDF sampling... and otherwise---if guiding places more probability in that direction than BSDF sampling---gradient descent nudges the selection probability more towards guiding.

And in the end, wherever these nudges---the gradient descent steps---average out, there's our optimal selection probability between the two strategies. That's the selection probability that minimizes the Kullback-Leibler divergence between the ideal and our actual PDF.

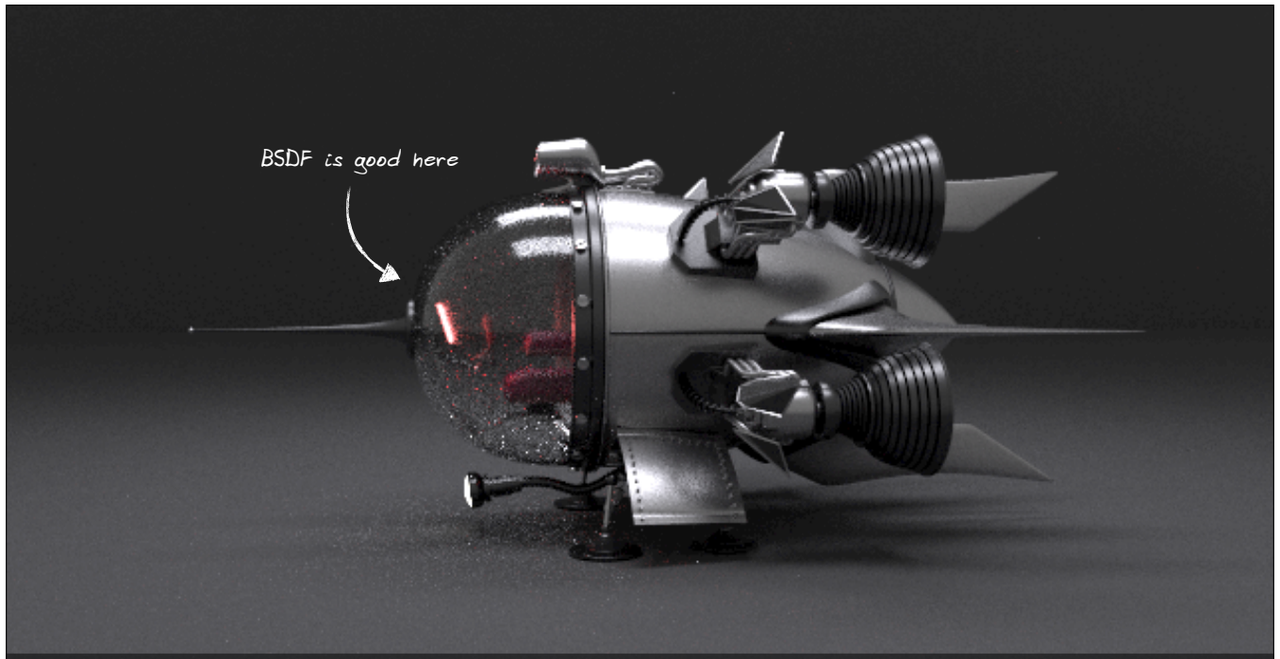
Let me show you an example where this really helps.



Here is the spaceship that I showed you in the beginning of the presentation... and this time I'm going to actually explain what's going on!

This scene is pretty easy to render for a regular path tracer---doesn't have a lot of noise.

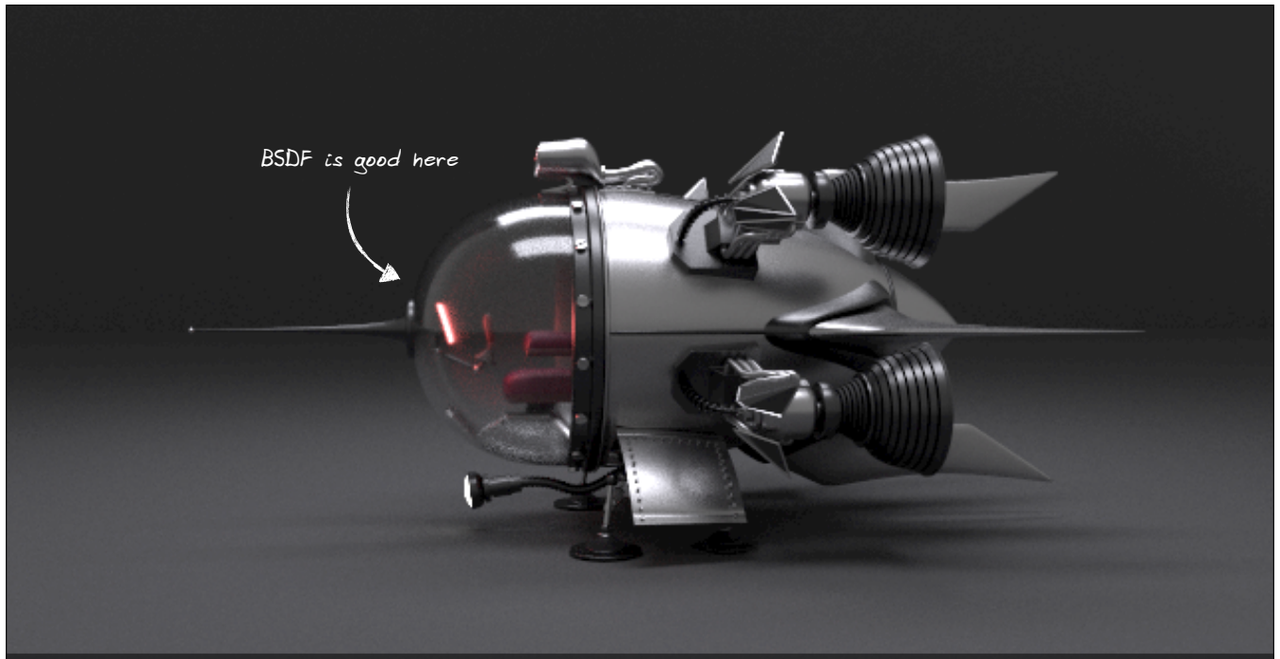
But when we enable plain old path guiding...



The image gets worse. *Particularly* on the cockpit of the spaceship.

That's because the cockpit glass has a just sliiiightly rough BSDF, which means that incident radiance doesn't really represent the product well, but the BSDF itself does! So the problem of the original practical path-guiding algorithm is that it guides paths according to incident radiance on the cockpit, where it really should just use BSDF sampling.

In theory, this is the perfect application of the learning scheme I just showed you. And indeed...



1024 spp

"Practical path guiding" with MIS learning

...with the learning scheme in place, the noise basically goes away! Let me flip back and forth a bunch of times.

Implementation

```
class SpatialLeaf()
    t, m, v, θ ← 0                                                          // Initialize state
    β_1 ← 0.9, β_2 ← 0.999, ε ← 10^−8, learningRate ← 0.01, regularization ← 0.01   // Hyperparameters
    function adamStep(∇_θ)
        t ← t + 1                                                           // Increment iteration counter
        l ← learningRate · sqrt(1 − β_1^t)/(1 − β_2^t)                      // Compute de-biased learning rate
        m ← β_1 · m_{t−1} + (1 − β_1) · ∇_θ                                 // Update first moment
        v ← β_2 · v_{t−1} + (1 − β_2) · ∇_θ · ∇_θ                           // Update second moment
        θ ← θ − l · m/(sqrt(v) + ε)                                         // Update parameter
    function misOptimizationStep(x, ω_i, ω_o, radianceEstimate, samplePdf)
        productEstimate ← radianceEstimate · f_s(ω_i) cos γ_i
        bsdfPdf ← p_{f_s}(ω_i|ω_o, x)
        learnedPdf ← isDiscrete(f_s(ω_i)) ? 0 : q(ω_i|x)
        spin lock (this)                                                    // Ensure θ is only optimized by one thread at a time
            α ← selectionProbability()
            combinedPdf ← α · bsdfPdf + (1 − α) · learnedPdf                // Equation 8
            ∇_α ← −productEstimate · (bsdfPdf − learnedPdf)/(samplePdf · combinedPdf)   // Equation 13
            ∇_θ ← ∇_α · α(1 − α)                                            // Chain rule
            regGradient ← regularization · θ                                // L2 regularization to avoid sigmoid saturation
            adamStep(∇_θ + regGradient)
    function selectionProbability()                                         // Called by the path tracer to use the learned probability
        return 1/(1 + e^−θ)                                                 // Sigmoid as in Equation 14
```

So surely this whole gradient descent stuff is complicated to implement, right?

Actually no! It's just around 20 lines of pseudo-code---and this includes all the implementation details plus a state-of-the-art stochastic-gradient-descent optimizer (Adam)! So it's really not that much.

There's also an open-source implementation of everything I've talked about so far---I'll have a link at the end of this talk.

Implementation

```
1  class SpatialLeaf()
2      t, m, v, θ ← 0                                                          // Initialize state
3      β1 ← 0.9, β2 ← 0.999, ε ← 10^−8, learningRate ← 0.01, regularization ← 0.01   // Hyperparameters
4      function adamStep(∇θ)
5          t ← t + 1                                                           // Increment iteration counter
6          l ← learningRate · √(1 − β2^t)/(1 − β1^t)                           // Compute de-biased learning rate
7          m ← β1 · m_{t−1} + (1 − β1) · ∇θ                                    // Update first moment
8          v ← β2 · v_{t−1} + (1 − β2) · ∇θ · ∇θ                               // Update second moment
9          θ ← θ − l · m/(√v + ε)                                              // Update parameter
10     function misOptimizationStep(x, ωi, ωo, radianceEstimate, samplePdf)
11         productEstimate ← radianceEstimate · fs(ωi) cos ϑi
12         [illegible]
13         [illegible]
14         spin lock (this)                                                    // Ensure θ is only optimized by one thread at a time
15         [illegible]
16         [illegible]                                                         // Equation 8
17             ∇α ← −productEstimate · (bsdfPdf − learnedPdf)/(samplePdf · combinedPdf)   // Equation 13
18             ∇θ ← ∇α · α(1 − α)                                              // Chain rule
19             regGradient ← regularization · θ                                // L2 regularization to avoid sigmoid saturation
20             adamStep(∇θ + regGradient)
21     function selectionProbability()                                         // Called by the path tracer to use the learned probability
22         return 1/(1 + e^−θ)                                                 // Sigmoid as in Equation 14
```

Takeaway #3

optimal MIS can be learned with stochastic gradient descent

The bottom line is that optimal selection probabilities for multiple-importance-sampling can be *learned* using stochastic gradient descent---we don't need to hard-code them---and this is *without involving neural networks* or other expensive stuff. The computational overhead is basically zero.

Results

Let me just show you some extra results that combine all of the improvements.

Here's an interior scene that is relatively simple to render with regular path tracing, but still...

...enabling "practical path guiding" reduces noise overall. That's because there nothing particularly special or difficult about this scene that would break the original algorithm. So this scene is a great benchmark to see whether our improvements have any negative effects in simple scenarios.

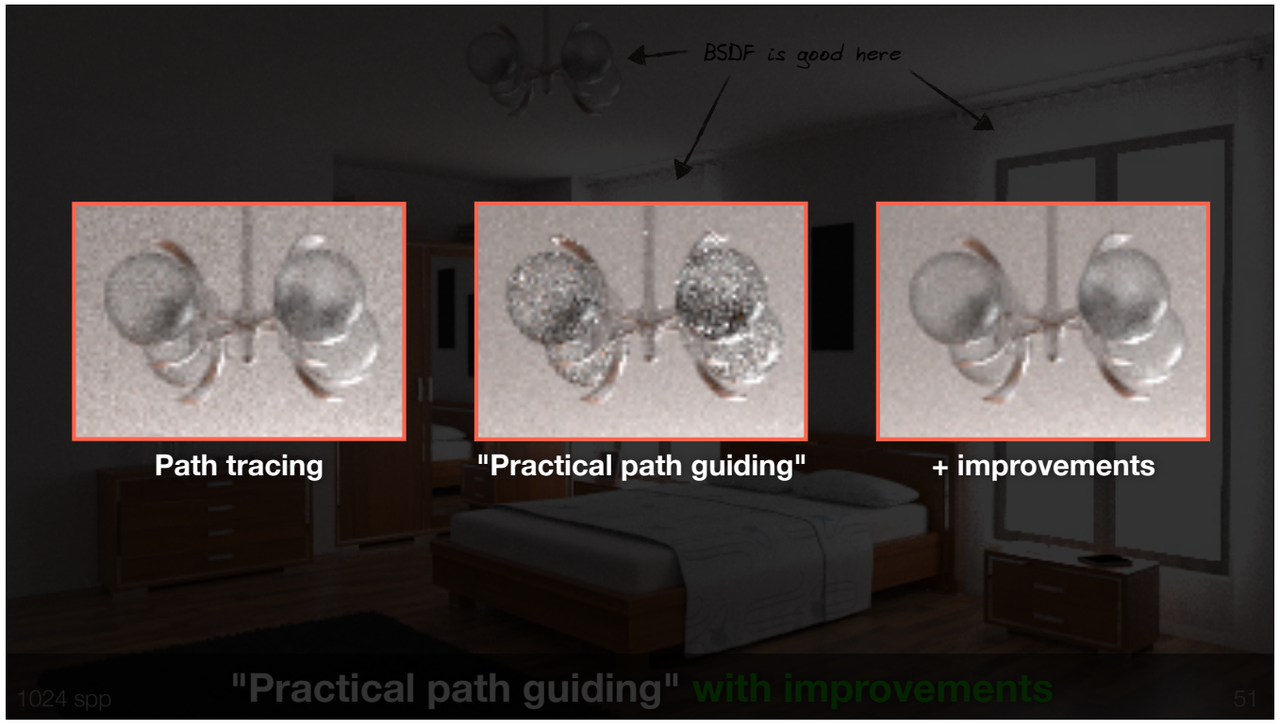
Switching to our improved path guiding in 3, 2, 1...

... I am going to flip back and forth.

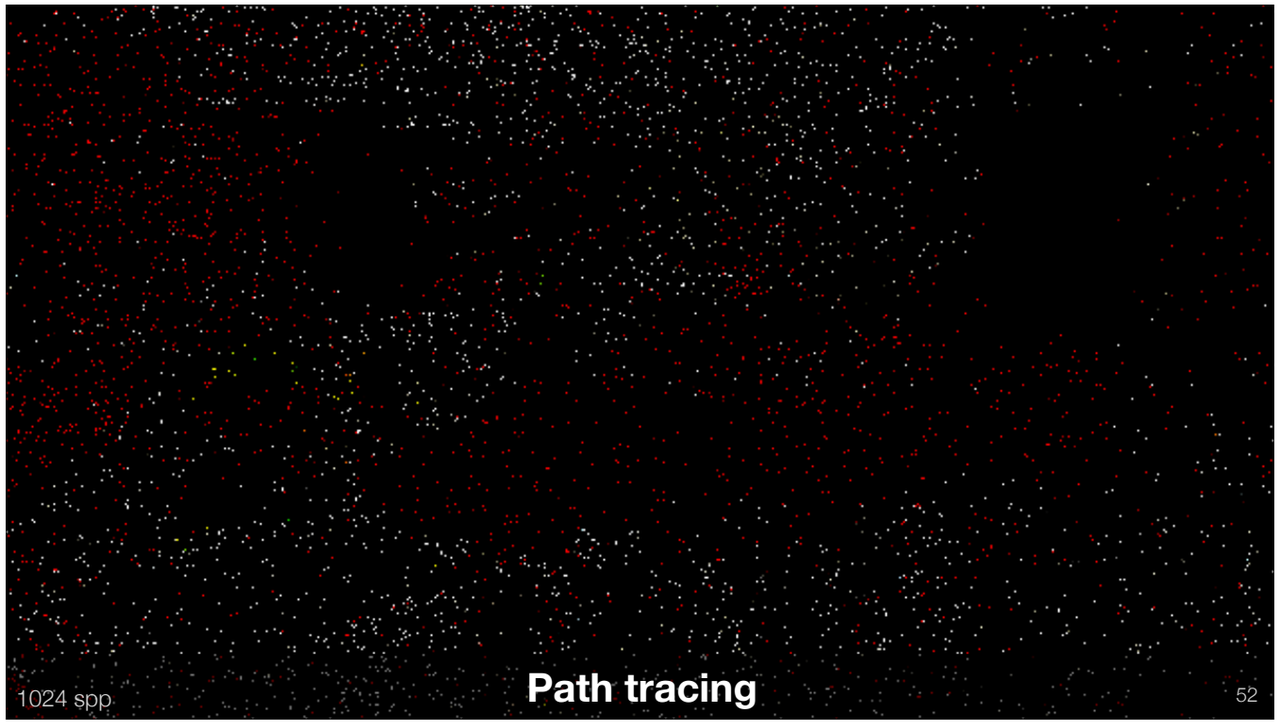
Here's another scene that's relatively simple to render with regular path tracing.

And here again, overall, the noise goes down, but in a few places shoots up dramatically because BSDF sampling is just way better there.

With our improvements, those places clean up again.

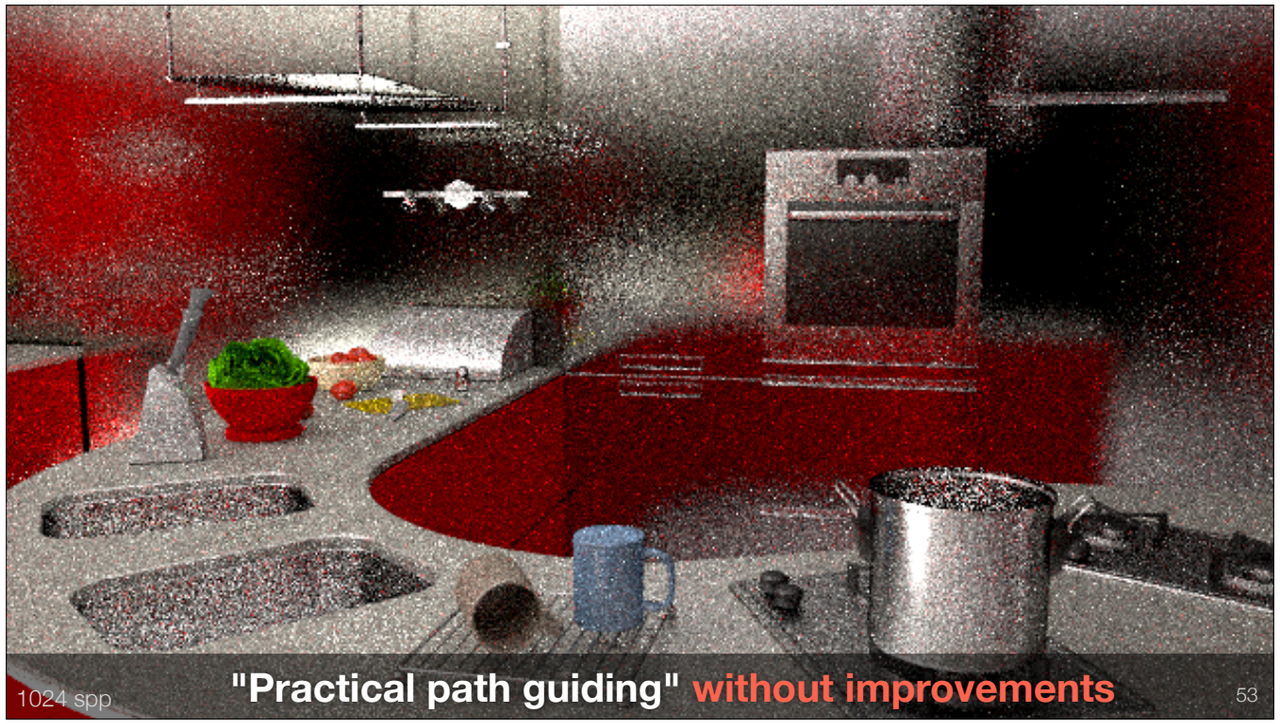


Here, just to highlight the differences: our improvements don't just improve above the original "practical-path-guiding" algorithm, but they also surpass the regular path tracer.



Now let's also have one last look at the difficult example from the beginning.

Path tracing is completely unusable, but enabling path guiding in 3, 2, 1...

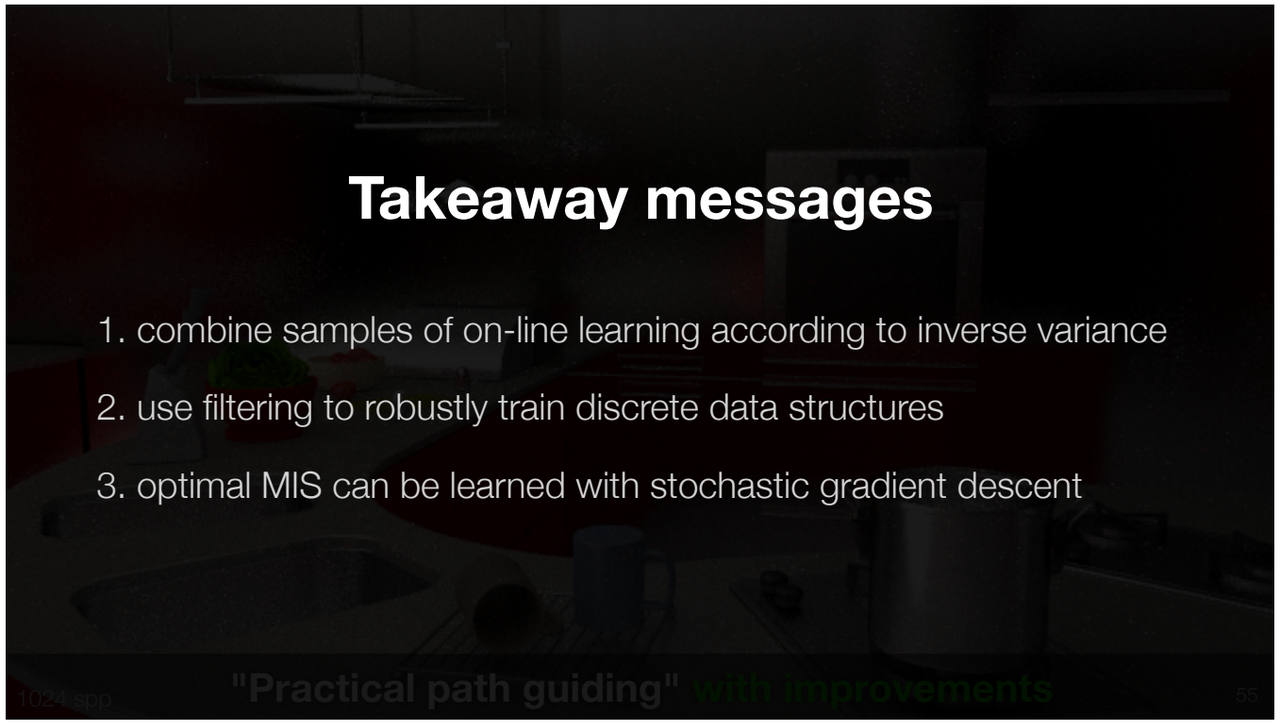


...we get a reasonable result. This scene contains lots of near-specular objects, so our improvements---in particular to the BSDF/guiding selection probability---should give a decent benefit.

Enabling them in 3, 2, 1...

...and difference is really quite significant.

Flipping back and forth...



I'd like to conclude by summarizing the take-away messages from before.

1. to avoid wasting samples when doing online training, *combine them according to their inverse variance*
2. filtering the samples when splatting them into some sort of discrete data structure for *guiding* is *really* important
3. we shouldn't use fixed selection probabilities between BSDF sampling and guiding anymore. We can actually *learn* the optimal probabilities using gradient descent and this doesn't just apply to radiance guiding but also to product guiding!

I'd like to point out here, that these points are *general*---they don't just work in conjunction with "practical path guiding", but they can be combined with pretty much all other path-guiding algorithms out there. So, for example, if you're using the product guiding from Herholz and colleagues you can still implement the learning of optimal selection probabilities for multiple importance sampling.

Thank you!

Code

Acknowledgments

Collaborators
Brian McWilliams
Jan Novák
Marios Papas
Markus Gross

Scenes
Benedikt Bitterli
Johannes Hanika
Ondřej Karlík
Nacimus
SlykDrako
thecali

Hyperion team
Andrew Fisher
Brent Burley
Dan Teece
Darren Robinson
David Adler
Joseph Schutte
Laura Lediaev
Mark Lee
Matt Chiang
Patrick Kelly
Peter Kutz
Rajesh Sharma
Ralf Habel
Wei-Feng Huang
Yining Karl Li

And with that I thank you for your attention!