Physically Based Shader Design in arnold

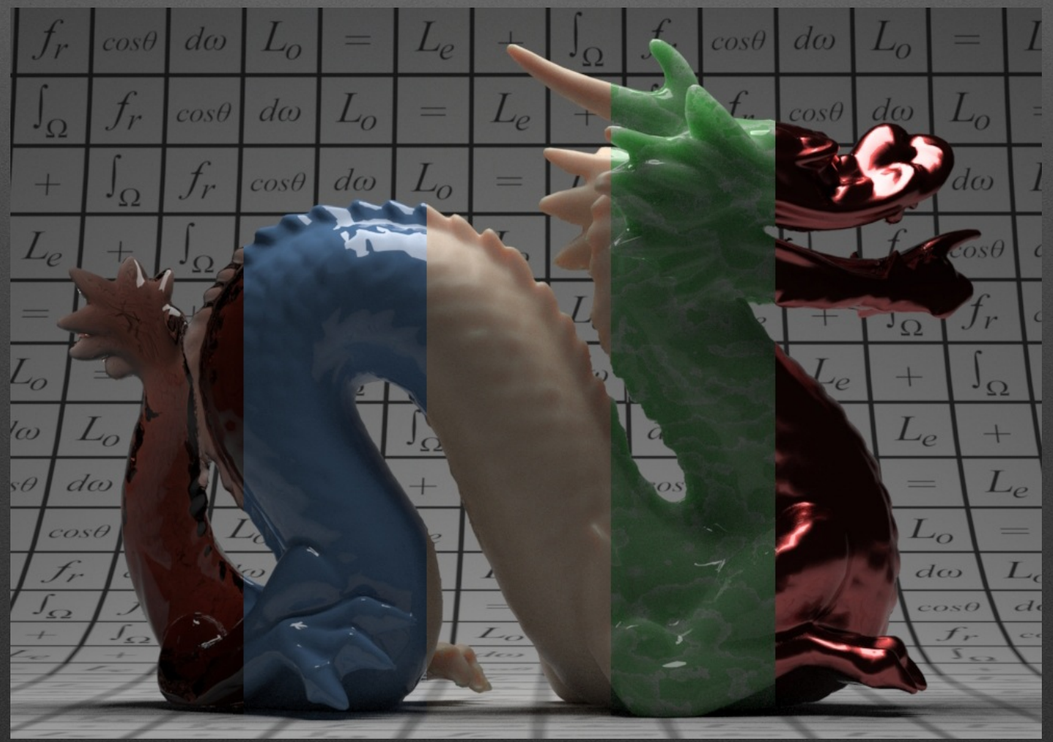
Anders Langlands

- Until recently, I was a Visual Effects Supervisor at MPC.
- I started at MPC in 2003 as a Shader Writer.
- Head of lighting for a while.
- Most recently Visual Effects Supervisor on X-Men: DOFP.
- I've spent a lot of time thinking about shading and lighting pipelines from different perspectives.

alShaders

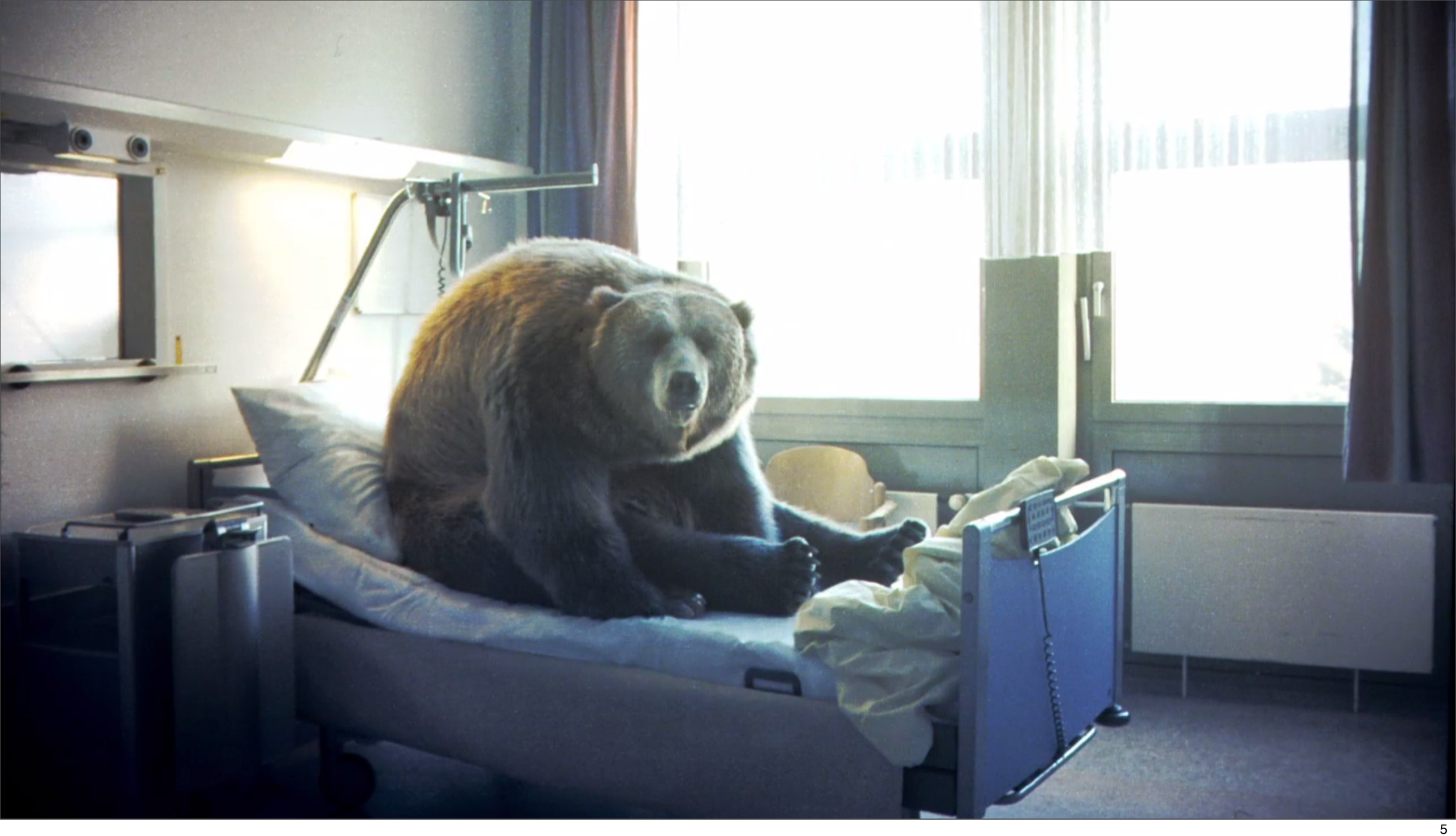
- Open-source, physically based, production shader library

- Seeing action at several studios around the world



- Today I'm going to be talking about alShaders, which is an open-source, physically based, complete production shader library that I've been developing in my spare time since 2012.
- In the two years since I started, more and more people around the world have been using it, which of course means I get more and more bug reports and that makes the almost complete lack of documentation quite embarrassing.


HUP




:: www.uglykids.org:: phil@armageddon-industries.com

STORM
STUDIOS

Why?

- Default Arnold shader library is a bit limited
- Provide a reference implementation for other users
- Experiment with new algorithms
- Because it’s FUN!

- So why would I do something like this, particularly when I open myself up to bug reports and user feature requests?
- When playing with Arnold, I found the default shader library a bit limited.
- Some good 3rd party shaders (e.g. Kettle) but no complete libraries.
- Wanted to make it open source, so others could contribute, and learn.
- Wanted to have a playground to experiment with new algorithms & techniques.
- Wanted to have fun!

What should it contain?

- Surface shader

- The most important part is the surface shader.
- It should be able to handle any type of hard surface: skin, metal, plastic, wood etc.

What should it contain?

- Surface shader

- Layer shader

- Then you can extend the range of materials by adding a layer shader.
- To do effects of one material over another, blended with weight maps.

What should it contain?

- Surface shader
- Layer shader
- Hair shader

- Of course you need a hair shader.
- Because characters without silly hair styles quickly get boring!

What should it contain?

- Surface shader
- Layer shader
- Hair shader
- Pattern generation

- Most textures are obviously painted: Photoshop, Mari, ZBrush.
- But it helps to have pattern generation for adding variation.
- By combining pattern generation types, you can create lots of different effects.

What should it contain?

- Surface shader
- Layer shader
- Hair shader
- Pattern generation
- Utility nodes

- And that's all glued together with utility nodes.
- Nodes for combining colors and floats together, remapping their ranges, etc.

What should it look like?

- Next question is what should it look like?
- How should it be structured?
- What sort of algorithms should we use?

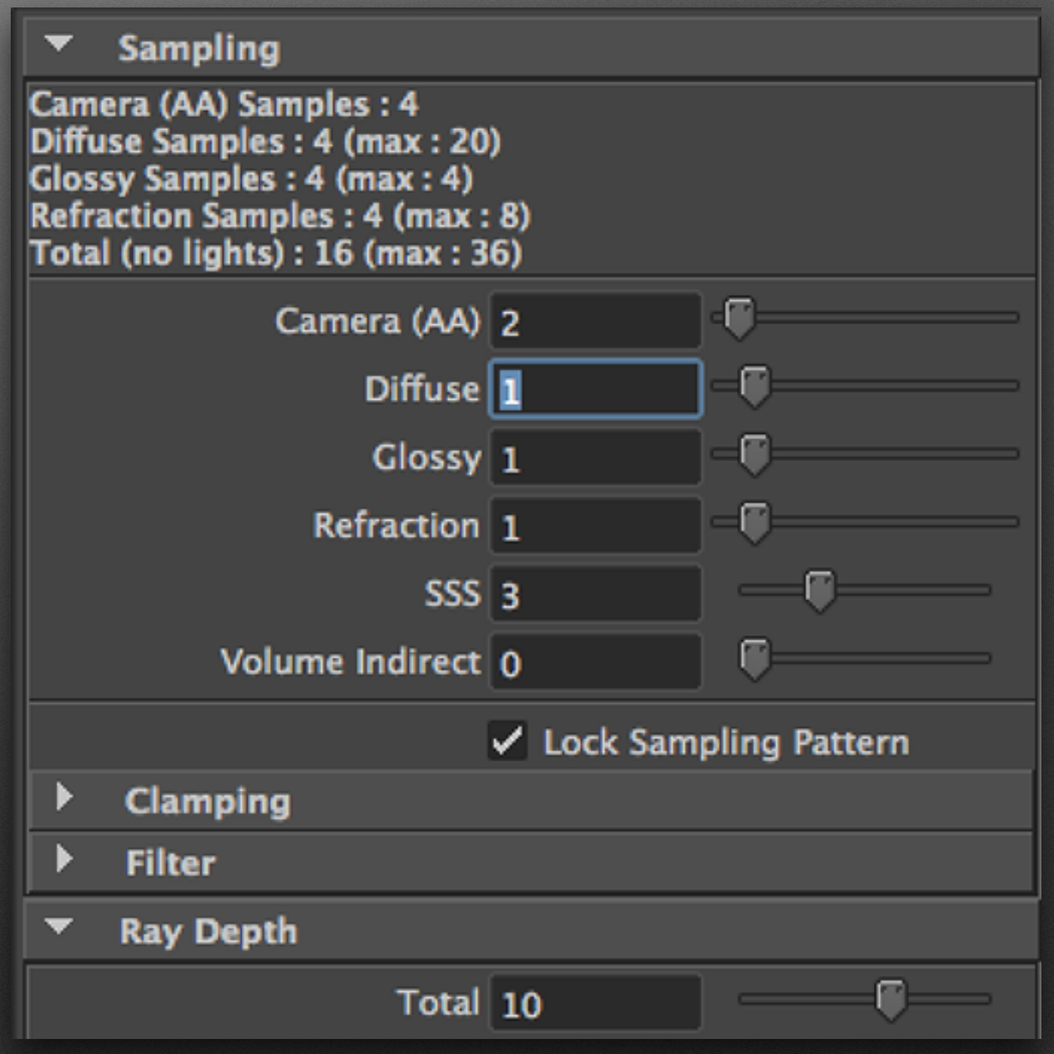
arnold Design

- Forward path tracer
- Extremely fast
- Extremely low memory usage
- C API

- Best idea is to play to a renderer's strengths.
- I've spent a large part of my career trying to turn Pixar's PRMan into something it's not.
- So we look to Arnold's design to inspire ours.

arnold Philosophy

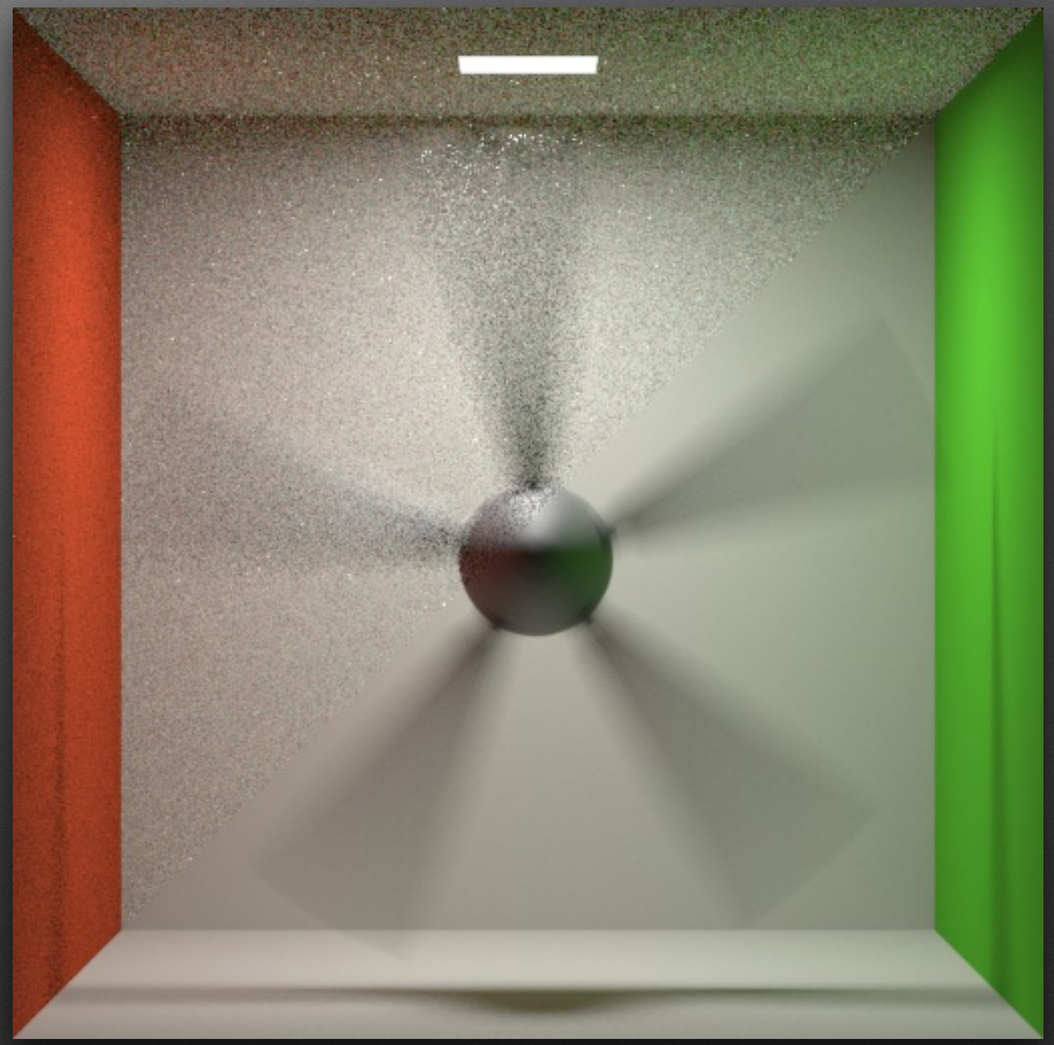
- Render everything in one pass
  - No baking
- Hide complexity from the user
  - Few knobs



- Arnold's philosophy is basically to keep everything as simple as humanly possible. That means no pre-passes or baking of illumination for irradiance caches or point clouds. People should be able to see and interact with images in as close to realtime as possible.
- We want easy-to-understand algorithms with as few controls as possible, so we should set good defaults and reparameterize where necessary.

arnold Philosophy

- Do one thing extremely well
  - Use Arnold's sampling routines
- Aim for production
  - Motion blur = high camera samples



- Arnold is a brute-force, unidirectional path tracer and it gets its efficiency by being extremely good at sampling and sample distribution, so we should use Arnold's sampling routines wherever possible.
- It's for real-world production, which means we're going to be rendering with motion blur, depth of field and huge amounts of geometry and textures, so we need to optimize our algorithms for high camera sample counts.

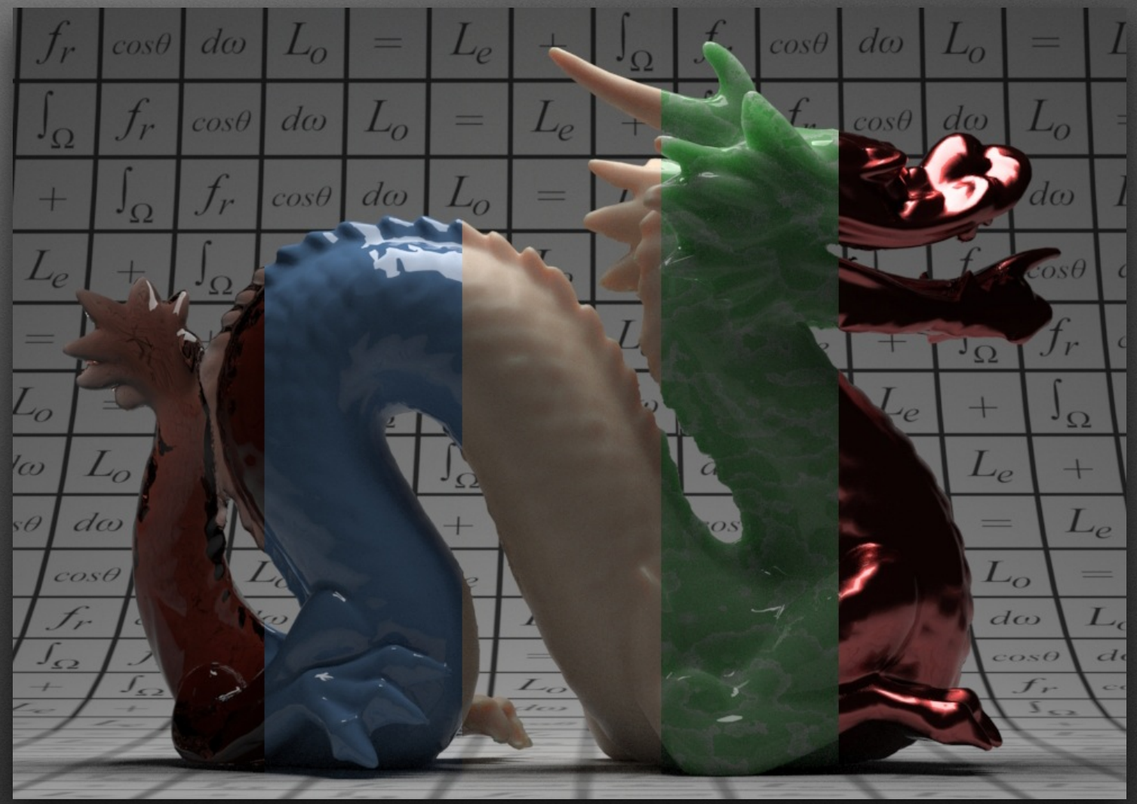
alSurface

- General-purpose surface shader

- Create (nearly) every material type

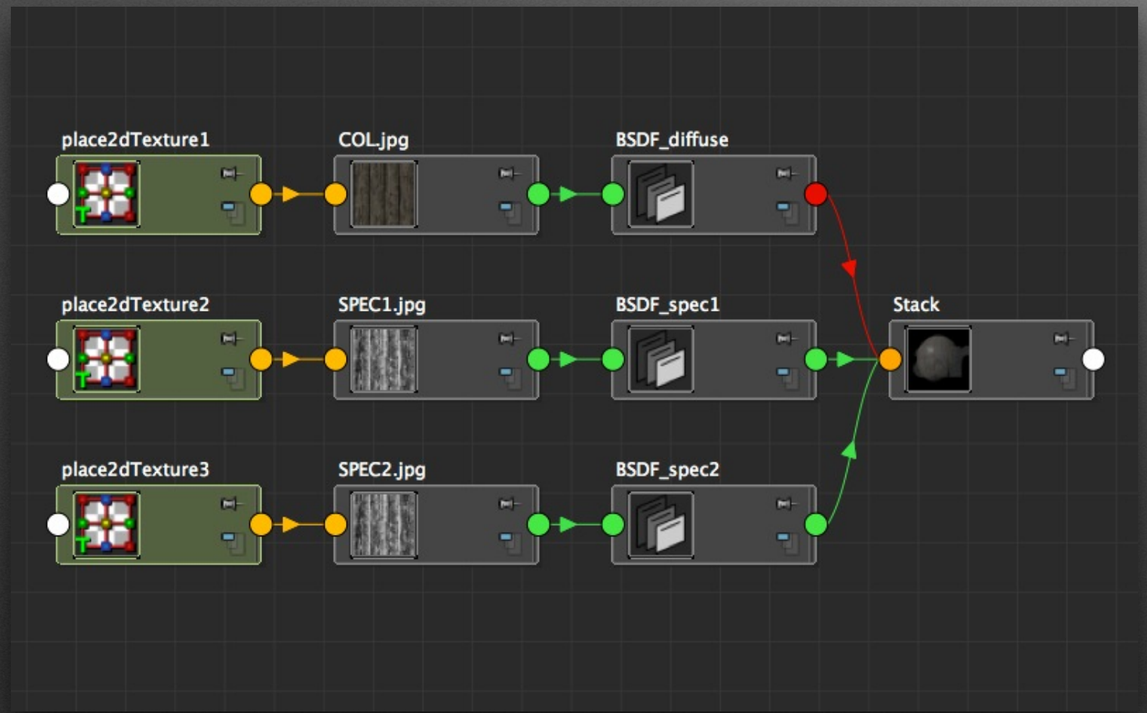
- Energy conserving

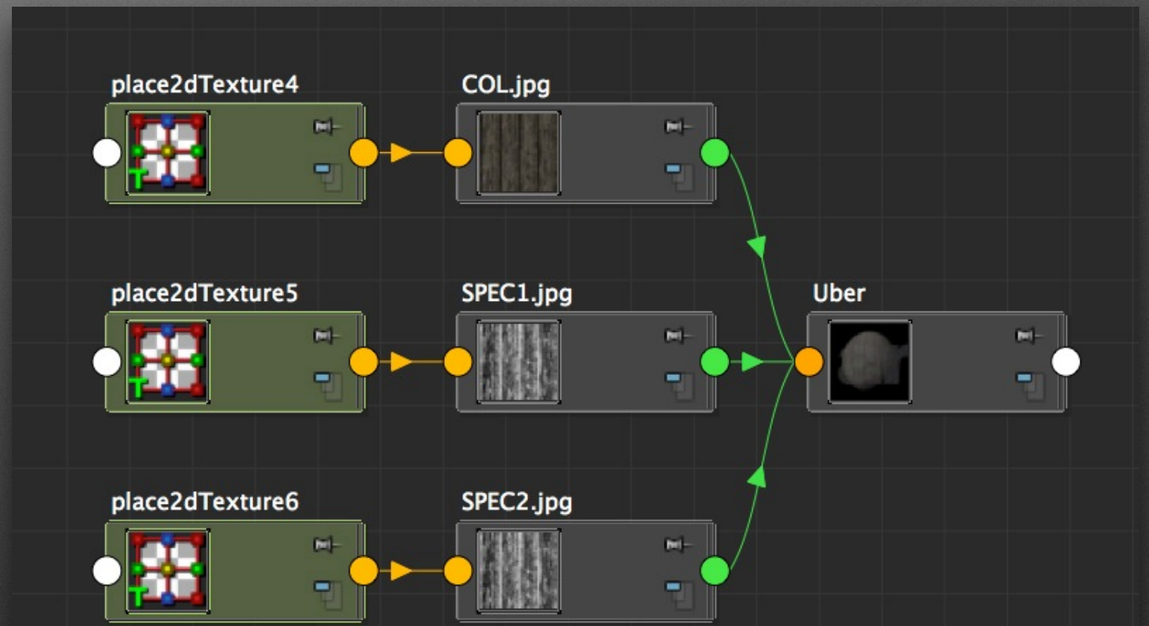
- Uber-shader model



- So I'm mainly going to be talking about the design of the library's surface shader, alSurface.
- The first thing to say about it is that it's an uber shader.

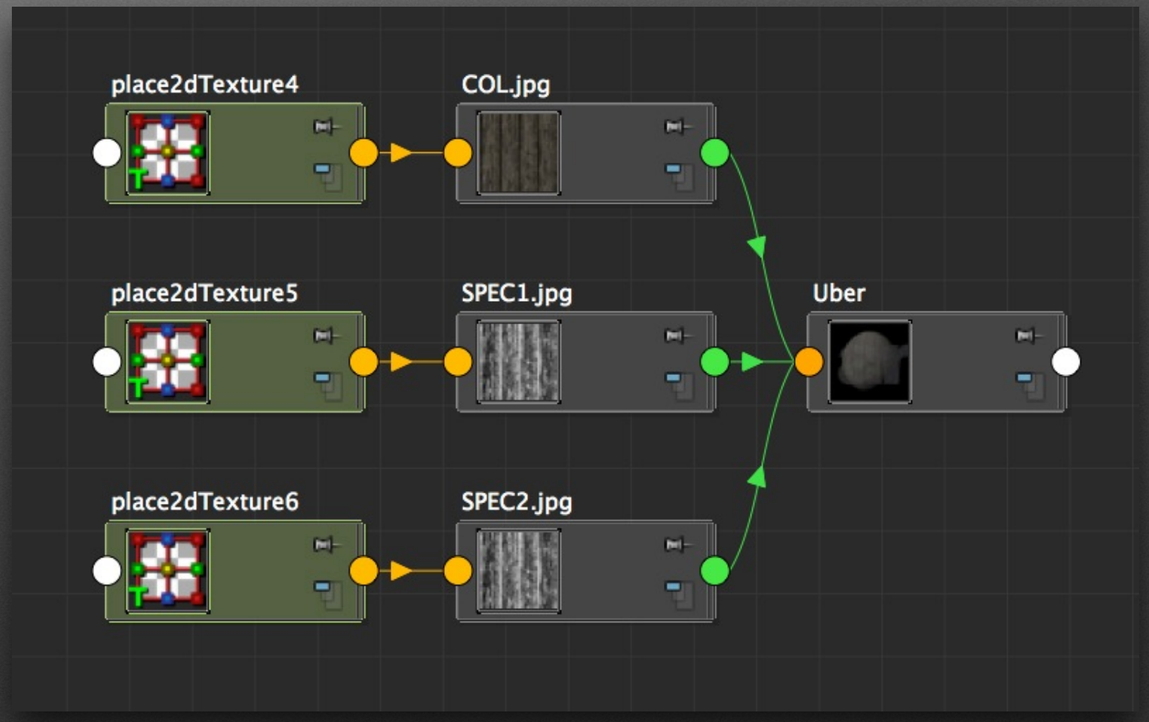
Uber-Shader vs. BSDF Stack





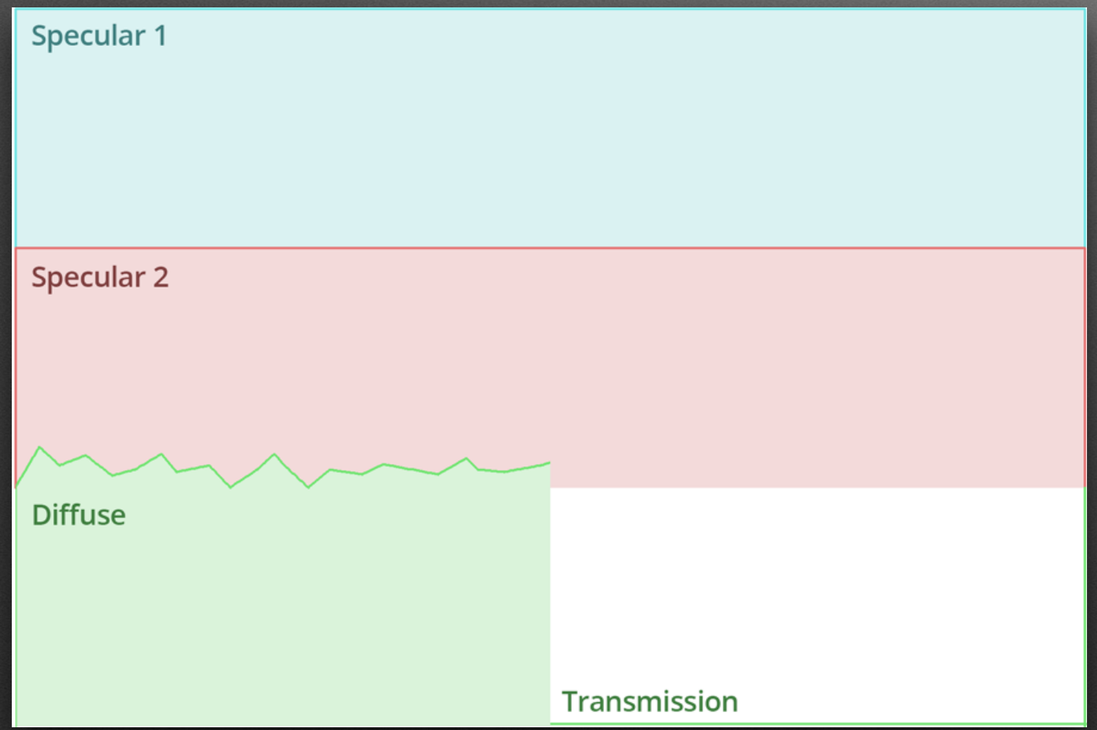
- I've written both uber-shader libraries and BSDF-stack libraries for production.
- Each have their strengths and weaknesses.
- A BSDF stack allows you to make any combination of individual BSDF lobes (diffuse, specular) you like.
- While an uber shader packages a few BSDFs together to form a complete material.
- An Uber shader is normally combined with a layer shader to extend its functionality.
- So it's similar, but your basic material chunk is bigger.

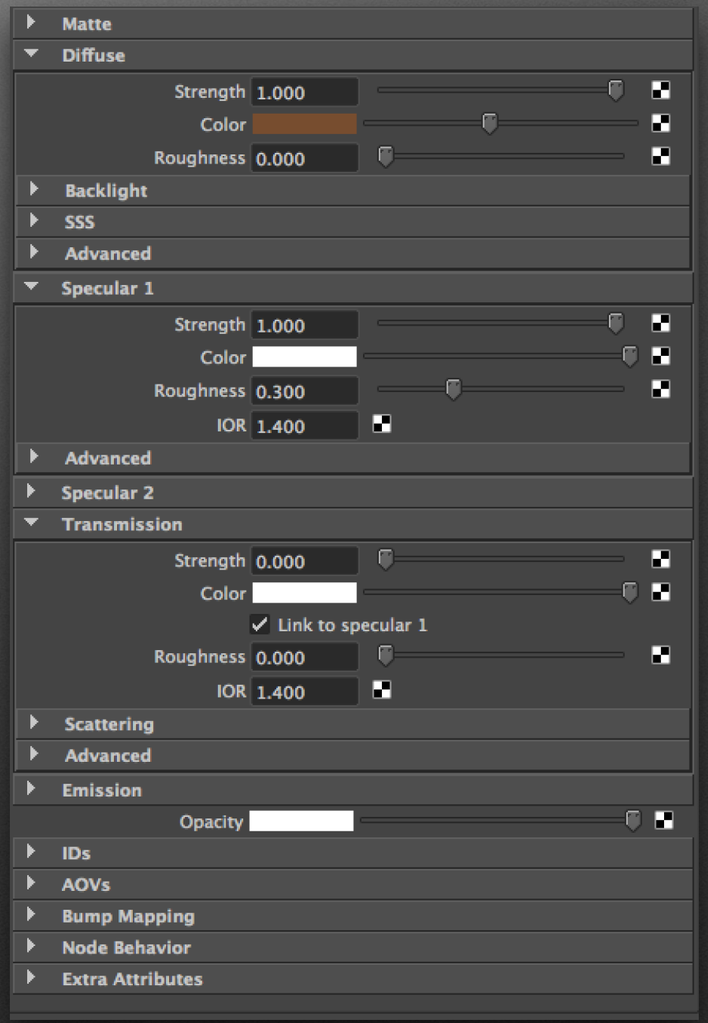
Uber-Shader vs. BSDF Stack



- We use an uber-shader model because it's simpler to use, easier to optimize and easier to ensure physical correctness.
- More details about the trade-offs in the course notes.

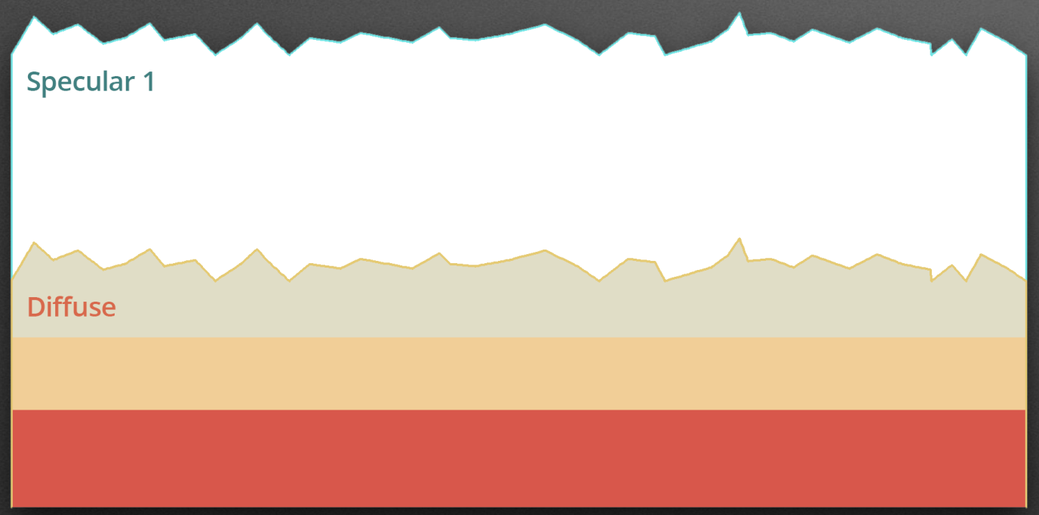
alSurface Layered Model

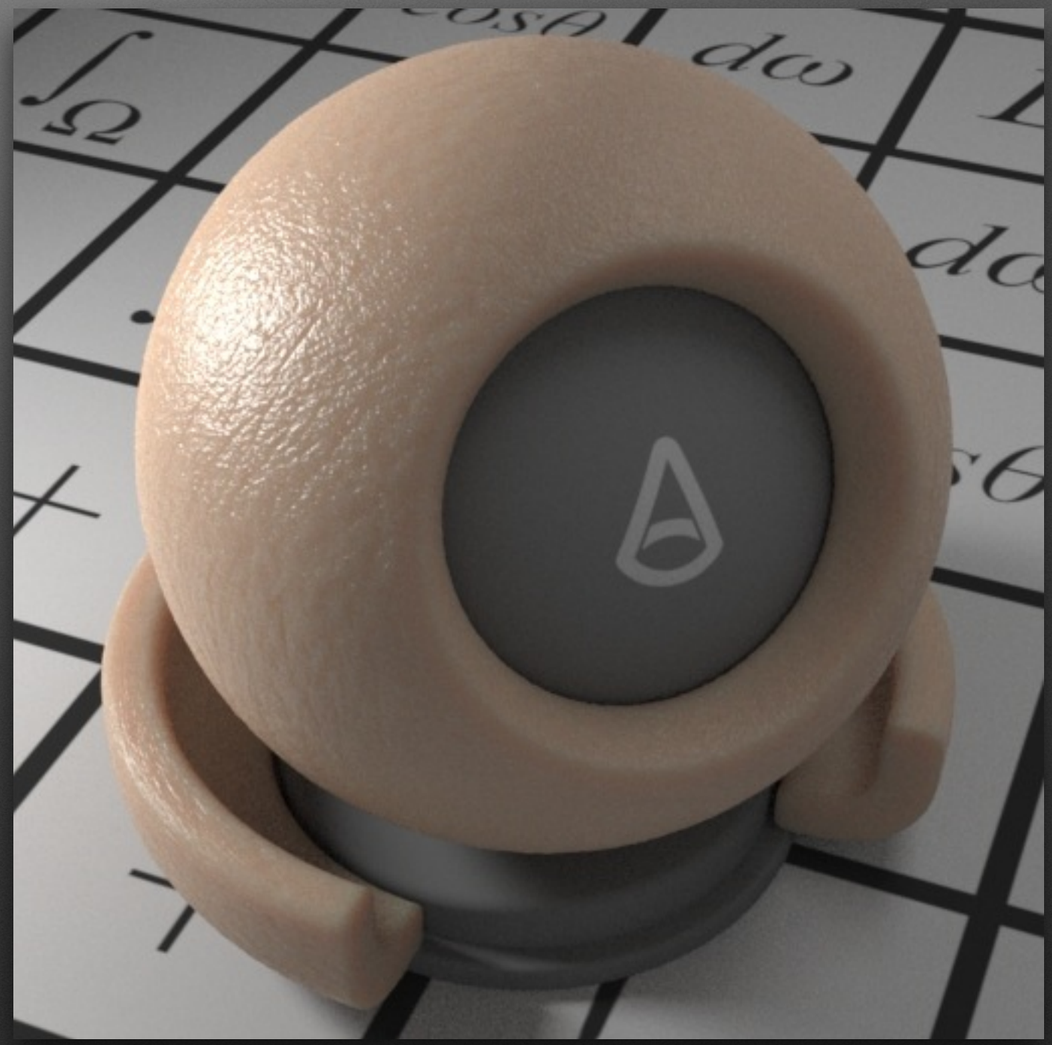




- So the structure of alSurface is like this…
- Conceptually it represents a material as a series of layers with a definite order.
- Either a diffuse or transmissive base, with one or two specular layers on top.
- Out of the box, this allows us to create 90% of the materials you will ever need.
- Because layering is explicitly controlled, we can make it very efficient and ensure energy conservation.

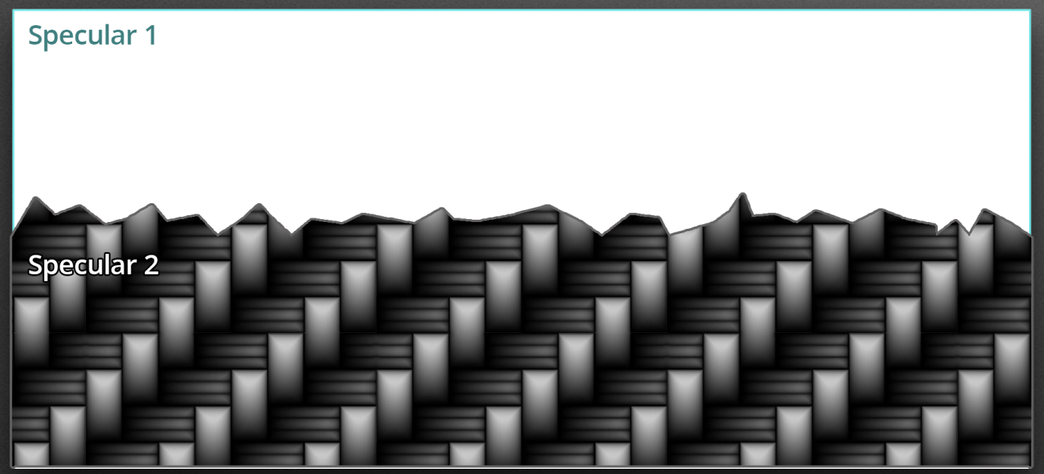
alSurface Layered Model





- For example, skin is just a diffuse (subsurface scattering) base with a rough specular on top.
- Arnold kindly provides us with a very efficient, multilayered, importance-sampled SSS model, which Solid Angle published last year.

alSurface Layered Model

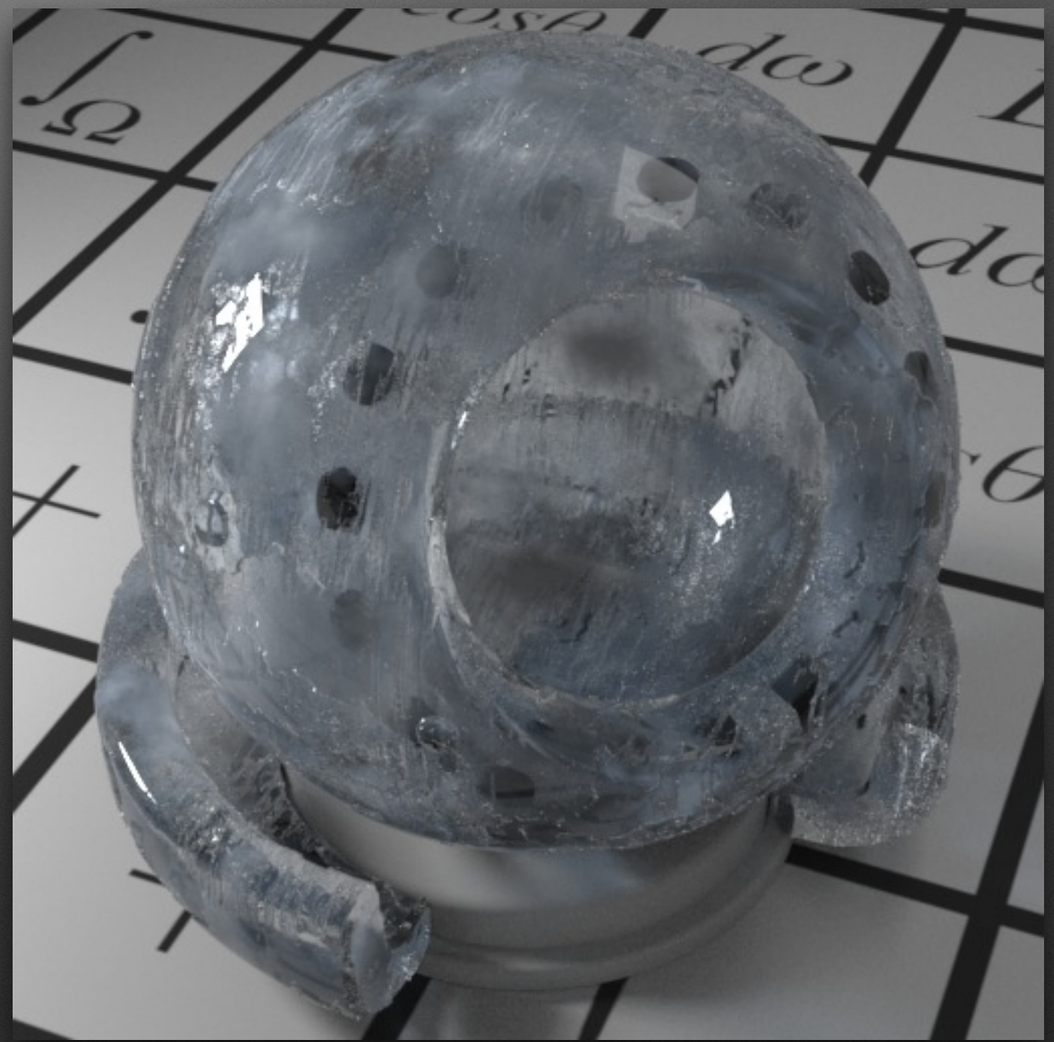


This carbon fiber material is a textured anisotropic layer with a clear-coat layer on top

alSurface Layered Model

Specular 1

Tranmission

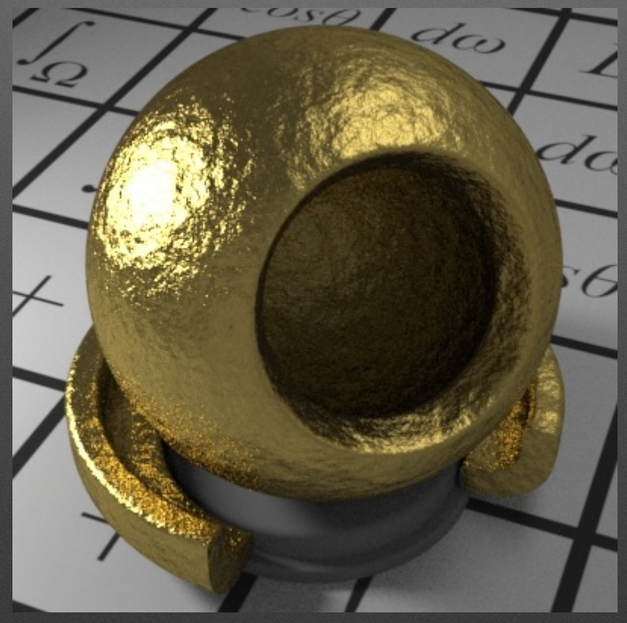


...and this ice material is a transmissive, single-scattering base layer with a smooth specular layer on top.

alSurface Layered Model

- Combine with alLayer shader for more

+

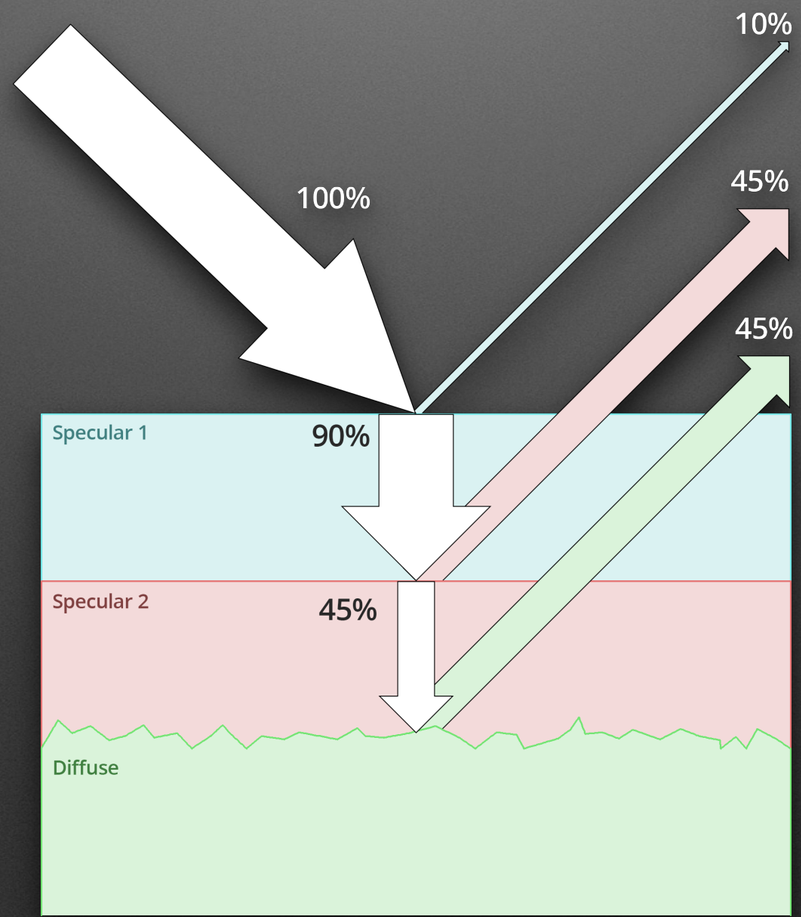


=

If we want even more flexibility we can just combine two materials with the layer shader.

Layer Energy Conservation

- Energy reduced by Fresnel transmission at each interface

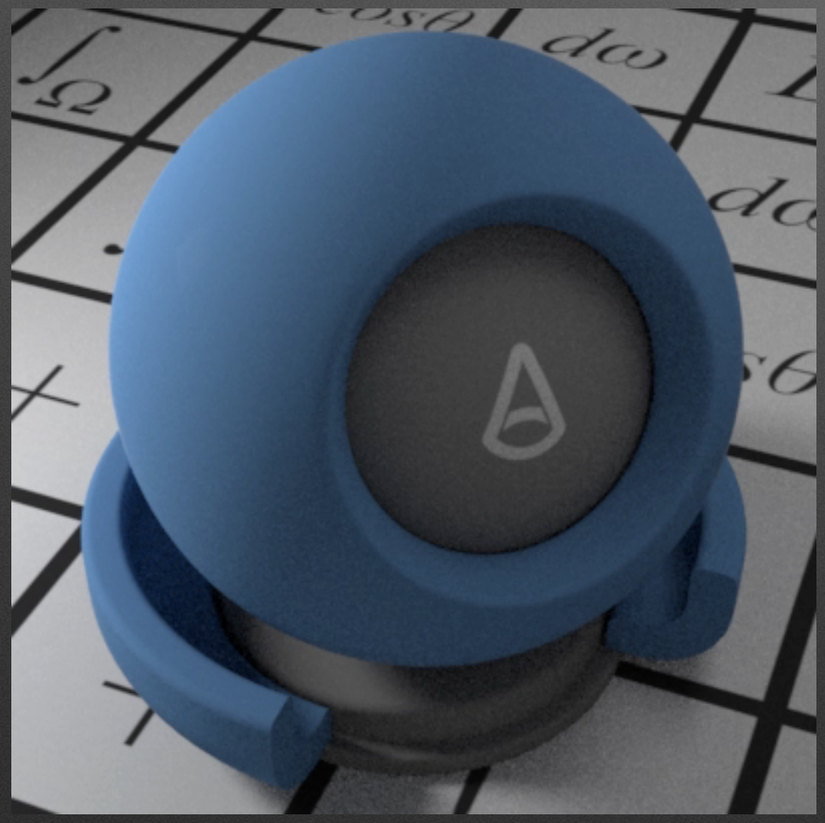


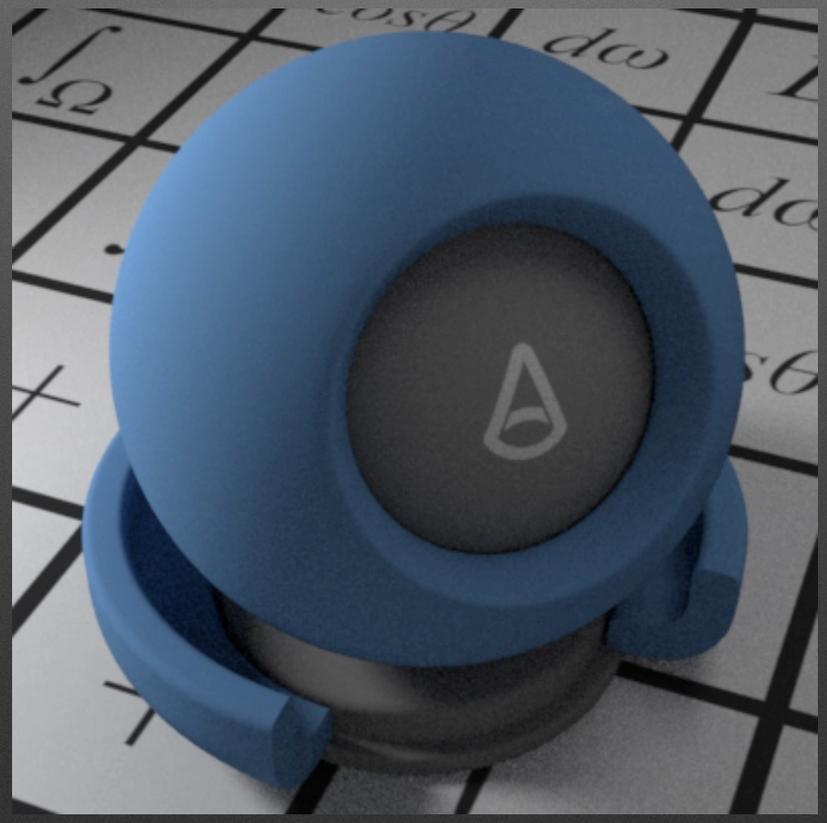
- The energy conservation model is simple and easy to understand.
- Each layer is an infinitely thin, microfacet scattering interface, which can either reflect or transmit light according to the Fresnel function. Any light that's not reflected or absorbed – according to the Fresnel function – is assumed to be transmitted.
- So we just evaluate each layer in a top-down order, integrating the Fresnel transmission as we go.
- At the same time, each layer then gets the chance to reflect a portion of the total transmitted light.
- So the amount of light reaching each layer is solely dependent on the index of refraction (IOR) of the layers above it.
- What's also important is that you can't turn this off: the user only has IOR controls for the layers and that's it. This means that the shader will always conserve energy and users don't have to worry about whether they're setting something right or not.

Layer Energy Conservation

So here I'm just increasing the index of refraction of the clear coat of this plastic shader and the layers balance accordingly.
(See supplemental videos.)

Layer Energy Conservation





So here I'm just increasing the index of refraction of the clear coat of this plastic shader and the layers balance accordingly.
(See supplemental videos.)

Rough Fresnel

```
reflection = fresnel(AiV3Dot(N, V), eta);
```


Now, it’s very important when computing Fresnel to do so correctly. The way we all used to do it was to just compute it based on the surface normal and the view direction, which is fine for perfectly specular BRDFs.

Rough Fresnel

```
reflection = fresnel(AiV3Dot(N, V), eta);
```

```
reflection = fresnel(AiV3Dot(H, L), eta);
```


But for rough BSDFs you really need to do it based on the normal of each microfacet (i.e, the half-angle vector) and the light direction.

Rough Fresnel

Otherwise you get artificially bright/dark edges, and dull highlights at facing angles as seen in the image on the left, which looks very bad compared to the correct way, as seen in the image on the right.

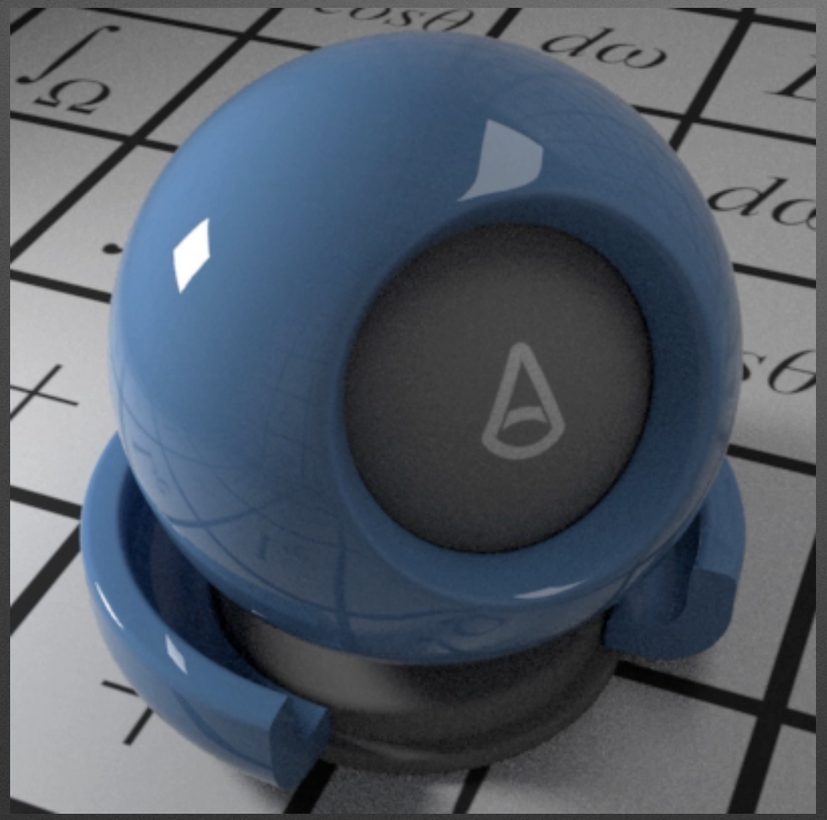
Rough Fresnel

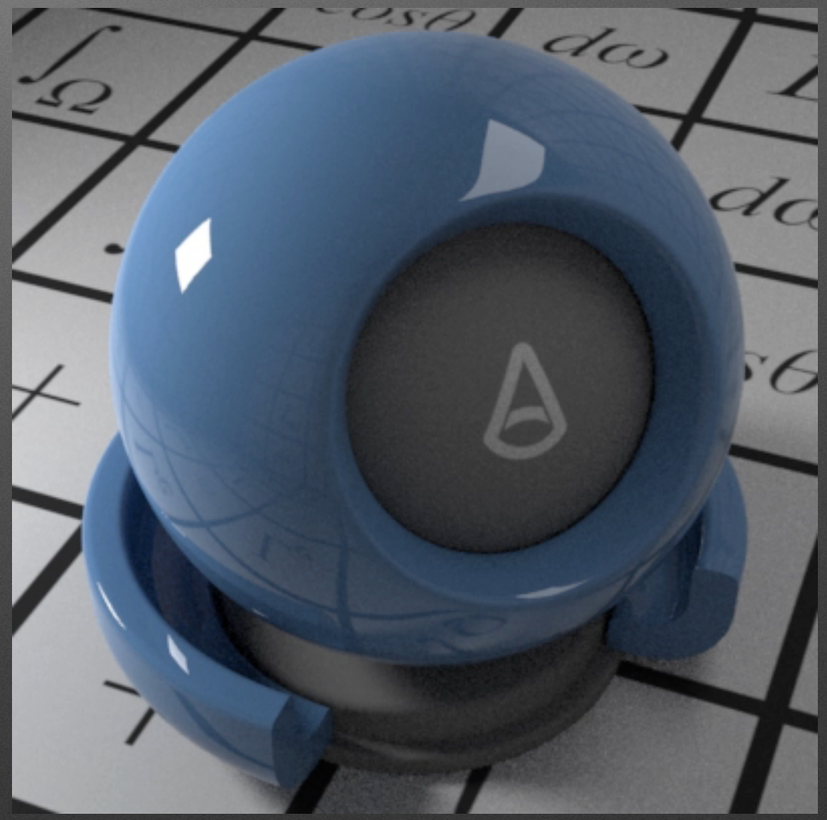
Otherwise you get artificially bright/dark edges, and dull highlights at facing angles as seen in the image on the left, which looks very bad compared to the correct way, as seen in the image on the right.

Rough Fresnel

Here's a wedge showing the same effect. You can see that the two methods start off looking the same but as roughness is increased the old-school method breaks down.
(See supplemental video.)

Rough Fresnel





Here's a wedge showing the same effect. You can see that the two methods start off looking the same but as roughness is increased the old-school method breaks down.
(See supplemental video.)

Improving Efficiency

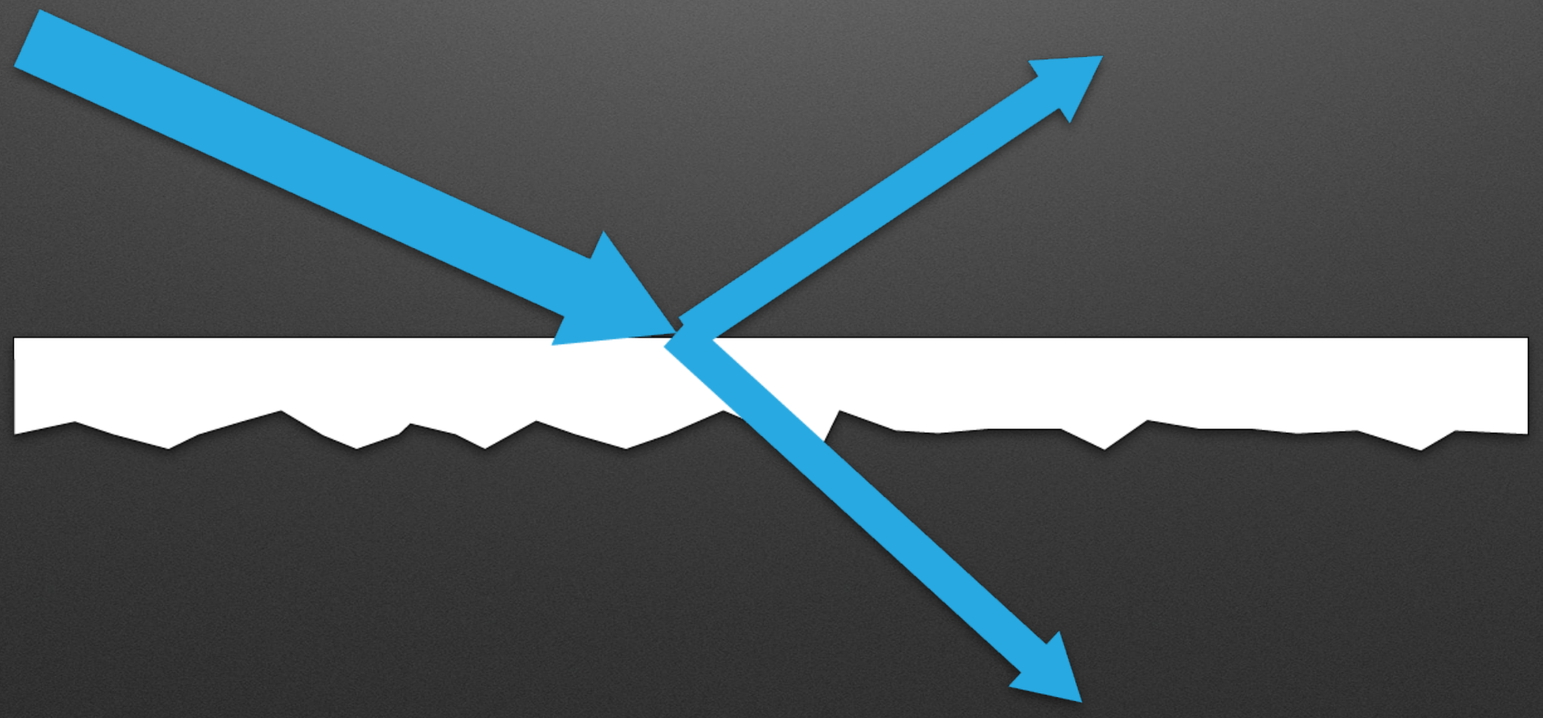
- So that's basically how it works.
- Next question is how do we make it faster?
- Because we're aiming for production, we know we're going to need high sample counts to deal with motion blur.
- Fortunately, Russian Roulette can be used reduce the number of rays we're tracing, without visibly increasing noise.

Russian Roulette - Splitting

- The first place we can do this is splitting.
- Rendering believable glass means many bounces of reflection and refraction (10 here).
- This can get expensive!

Naive Approach

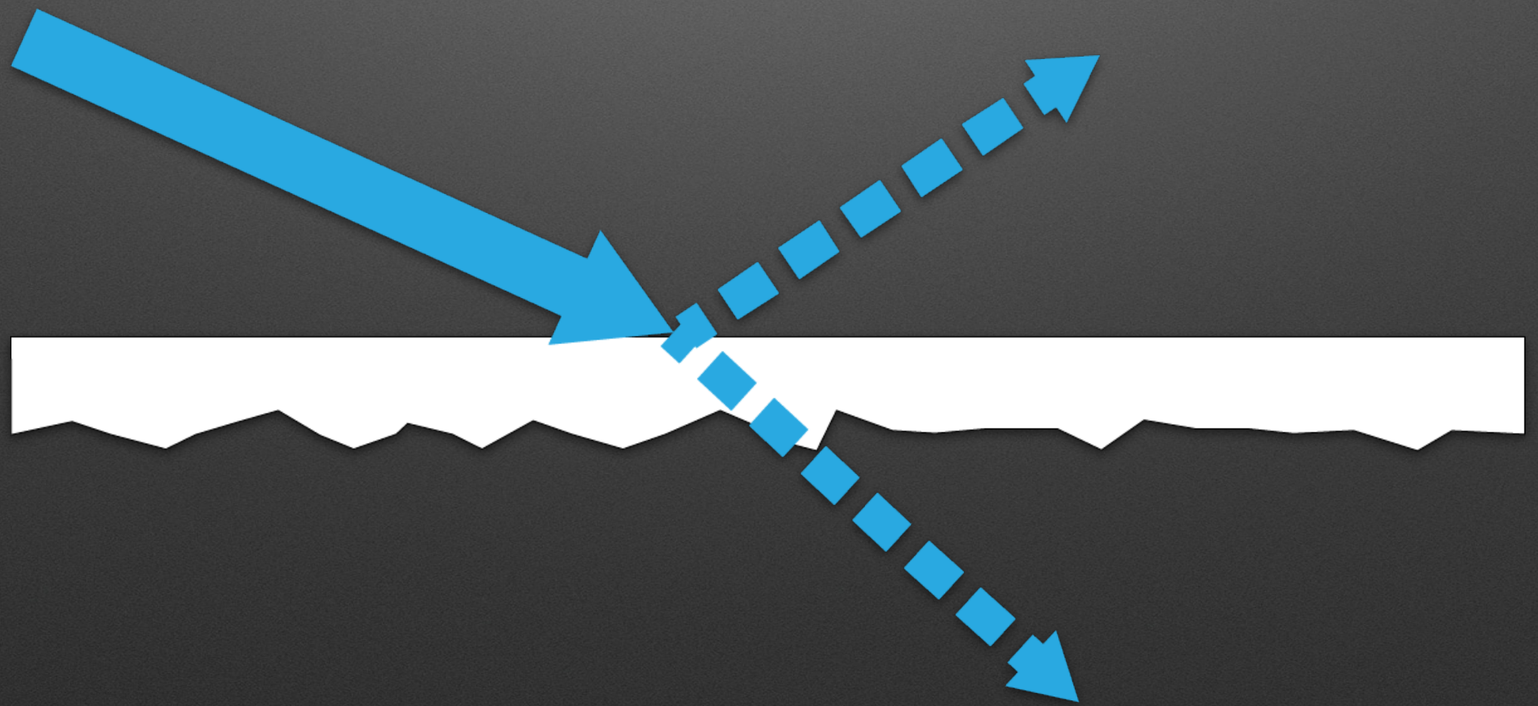
```
result = reflection() * fresnel()
       + transmission() * (1-fresnel);
```



- The naive approach just evaluates both reflection and transmission and scales them by the Fresnel function.
- Because you're tracing two rays at each intersection, the ray counts grow exponentially.

Russian Roulette - Splitting

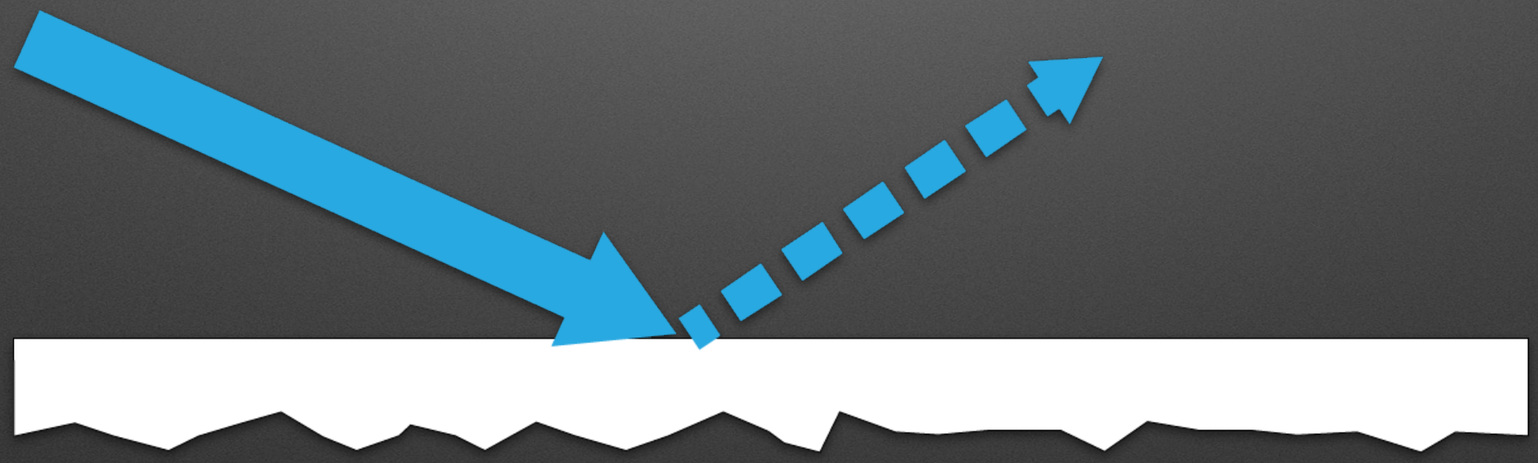
```
if (rand() < fresnel())
  result = reflection();
else
  result = transmission();
```



Instead, we can just randomly choose to evaluate one or the other in proportion to the Fresnel function.

Russian Roulette - Splitting

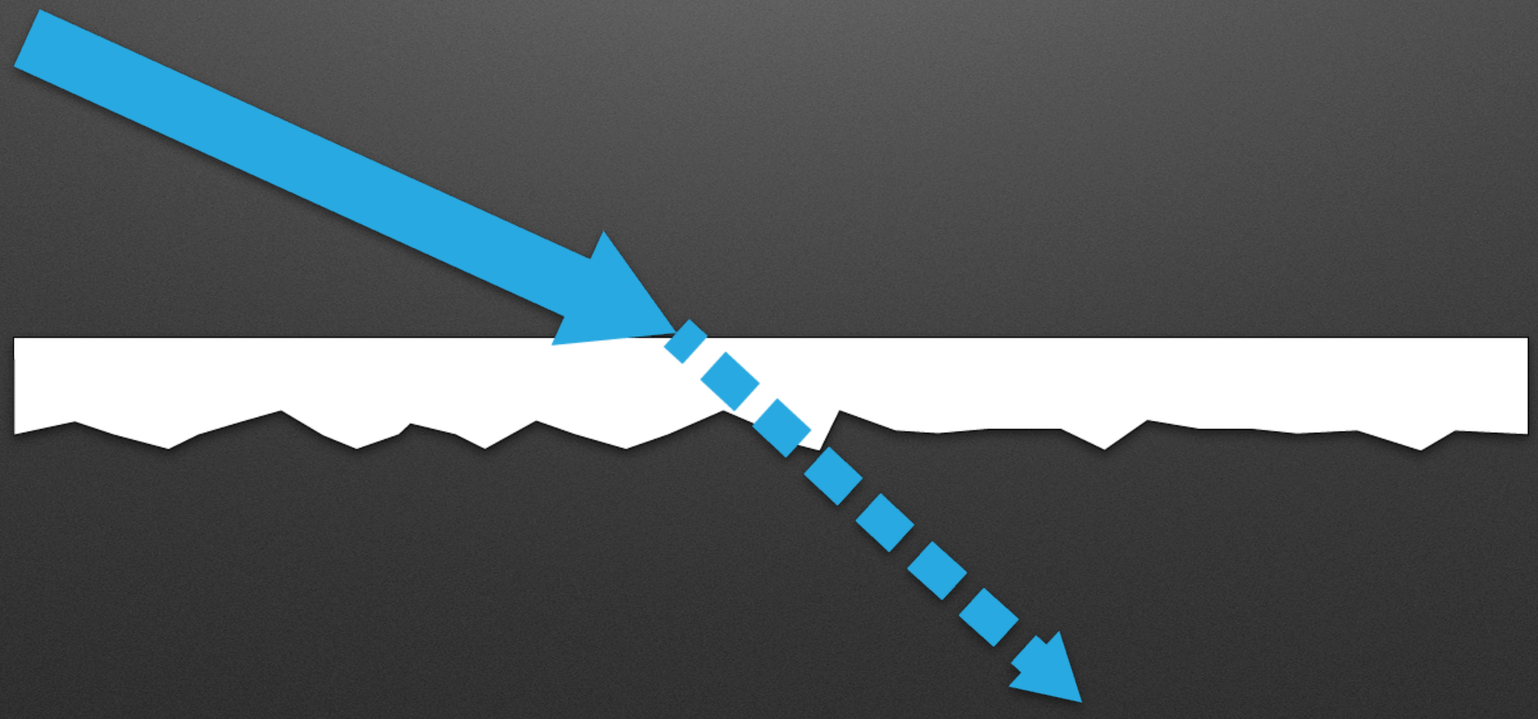
```
if (rand() < fresnel())
  result = reflection();
else
  result = transmission();
```



Instead, we can just randomly choose to evaluate one or the other in proportion to the Fresnel function.

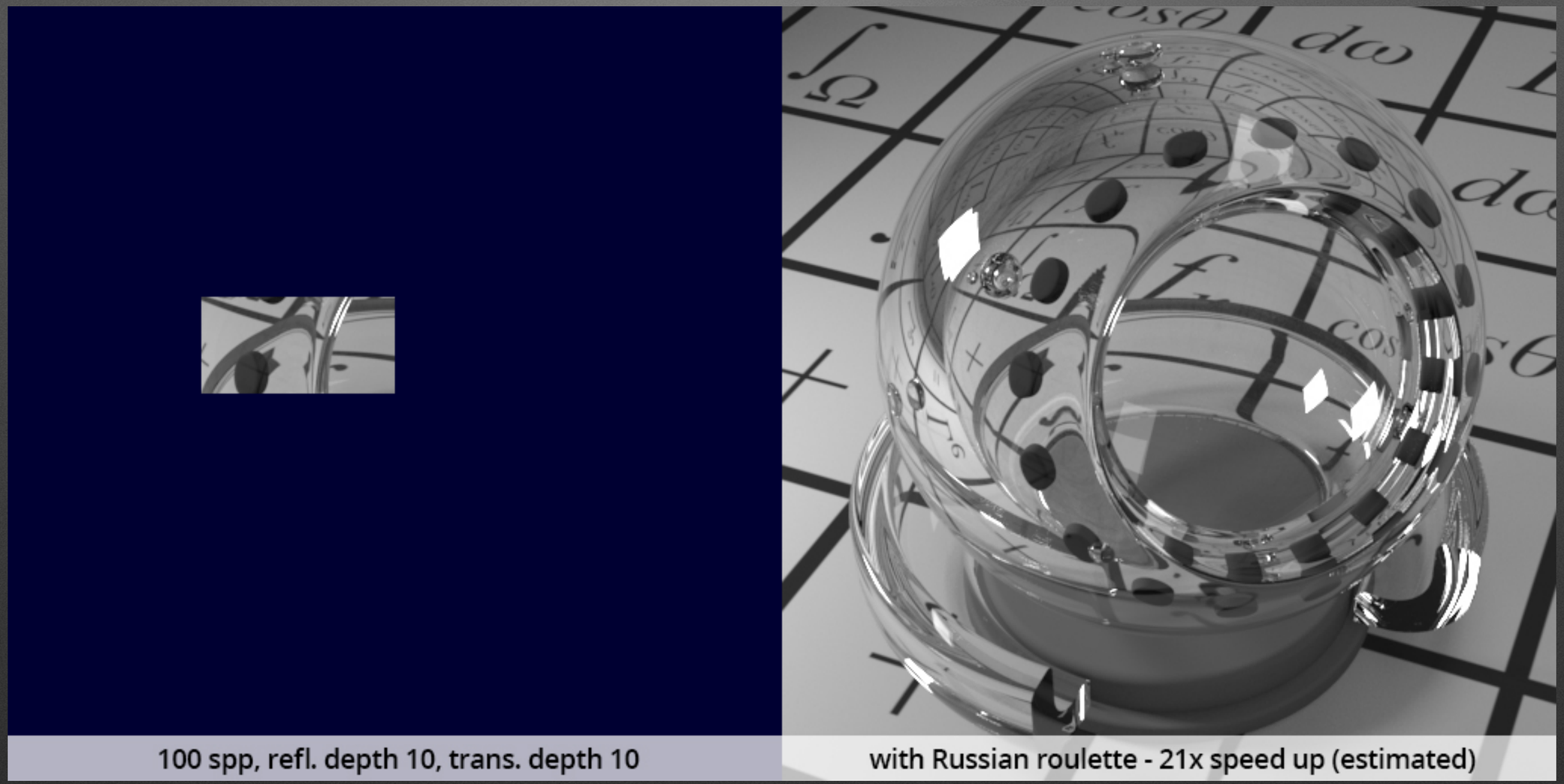
Russian Roulette - Splitting

```
if (rand() < fresnel())
  result = reflection();
else
  result = transmission();
```



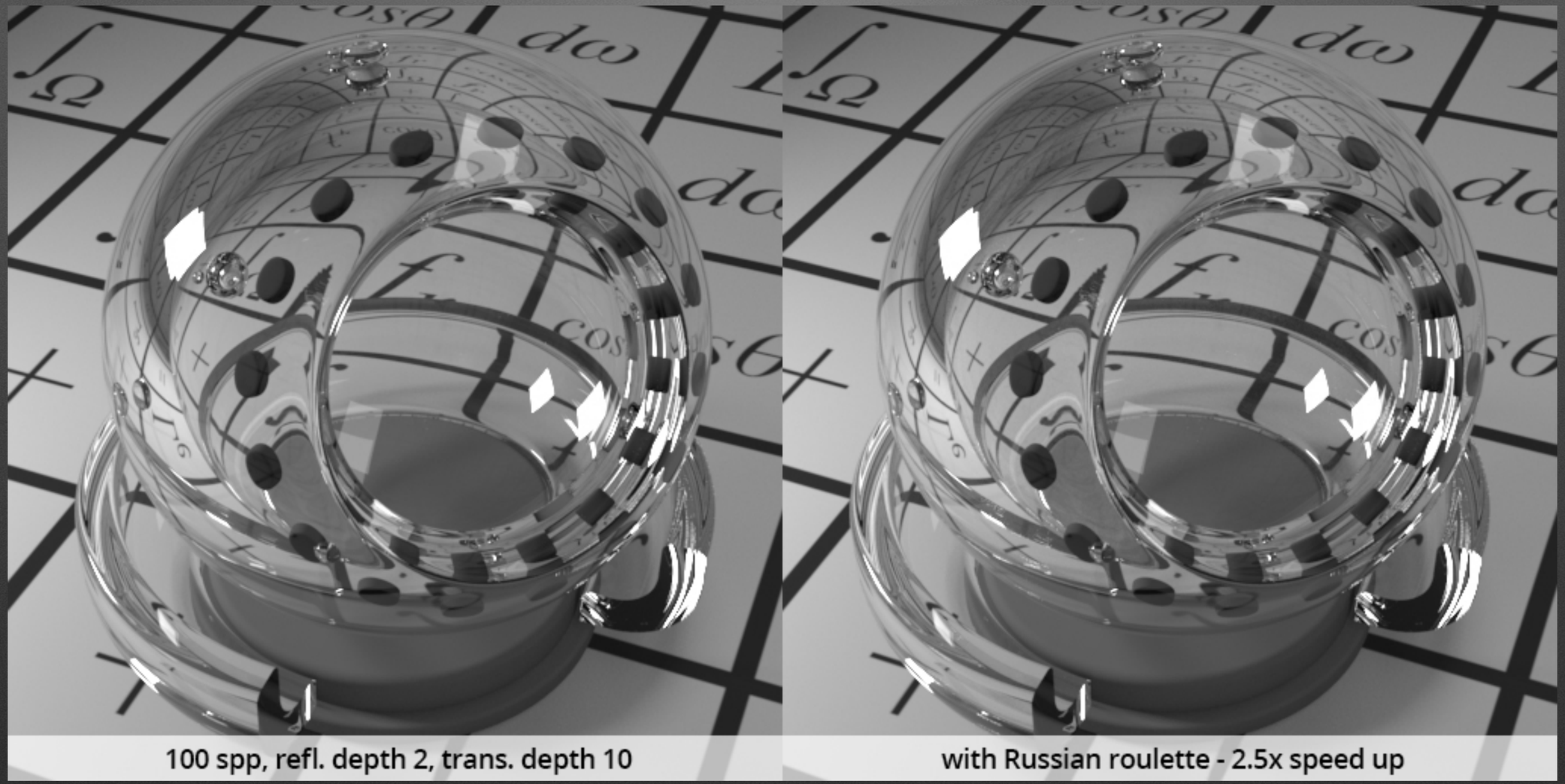
Instead, we can just randomly choose to evaluate one or the other in proportion to the Fresnel function.

Russian Roulette - Splitting



- …and that speeds things up dramatically!
- With 10 bounces each of reflection and transmission, I couldn't be bothered to wait for the render without RR to finish, but it was going to be at least 21x slower than the RR version.

Russian Roulette - Splitting

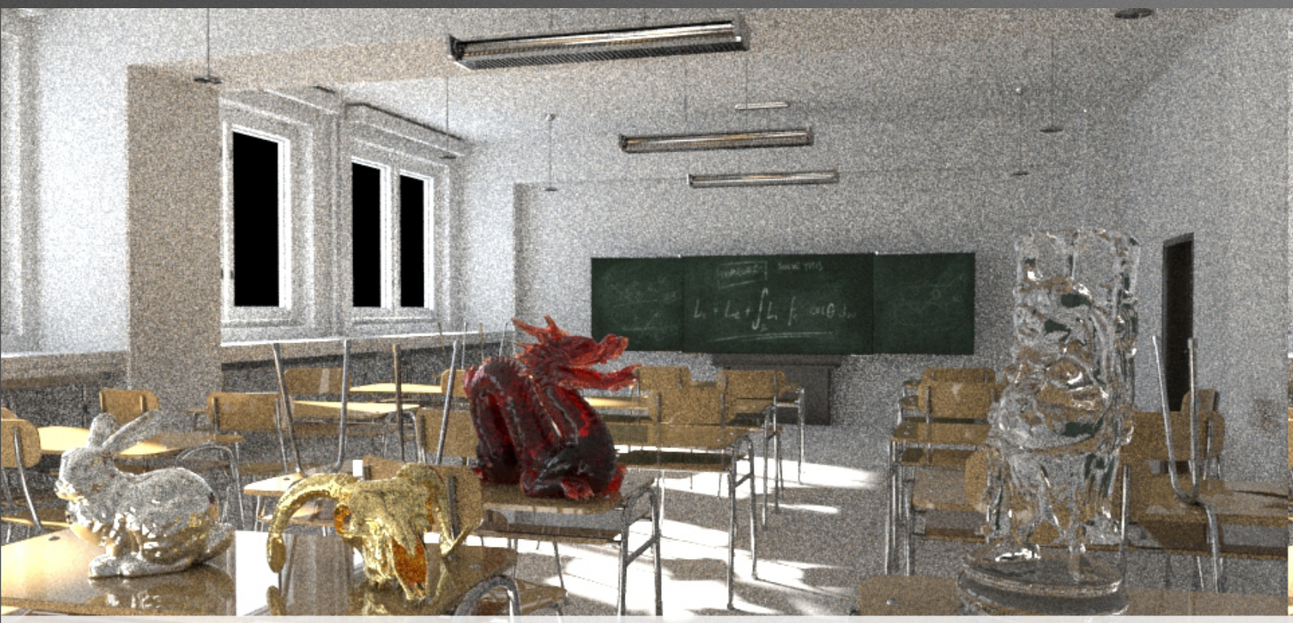


100 spp, refl. depth 2, trans. depth 10

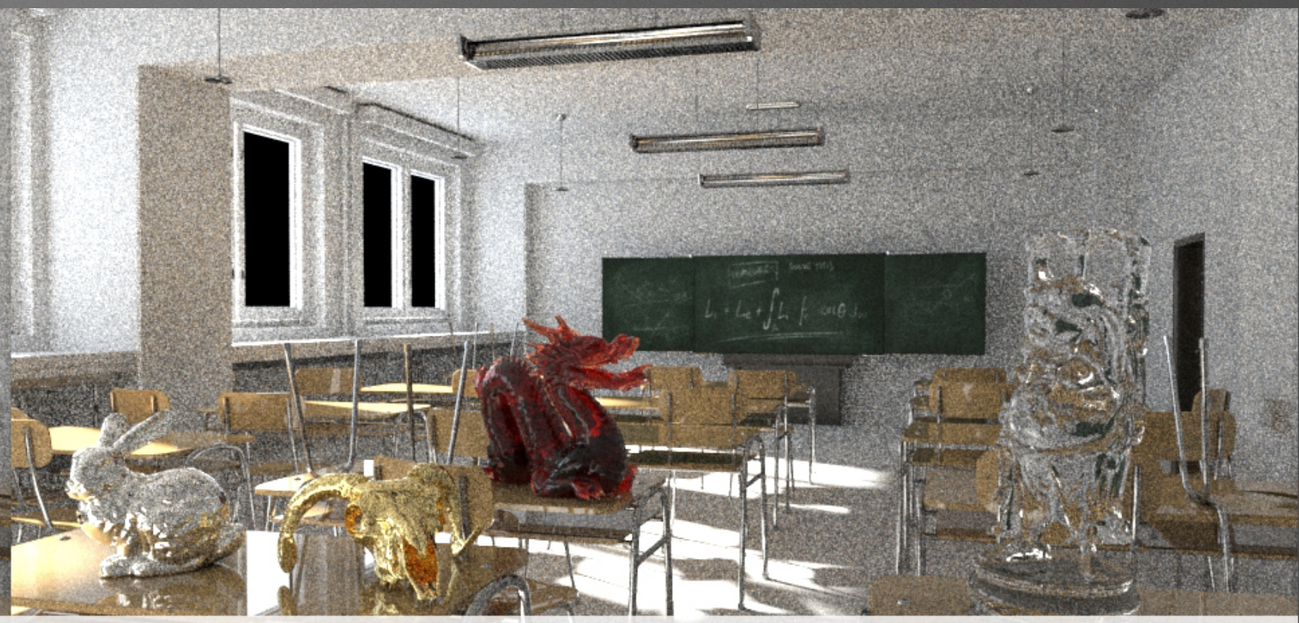
with Russian roulette - 2.5x speed up

- Limiting the reflection bounces to 2 we still get a dramatic speedup.
- At the sort of sample counts we want to use to resolve motion blur, there's no extra noise visible in the RR render.

Path Termination



Russian roulette disabled, depth 45



Russian roulette enabled, depth 45 - 2.5x faster

without

with = 2.5x speedup

- We can also use Russian Roulette to speed up other types of scenes with many bounces, by limiting the number of bounces we do to the ones that actually matter to the image.
- Here we've reduced the render time of this classroom scene by 2.5 times, with a barely noticeable increase in noise.

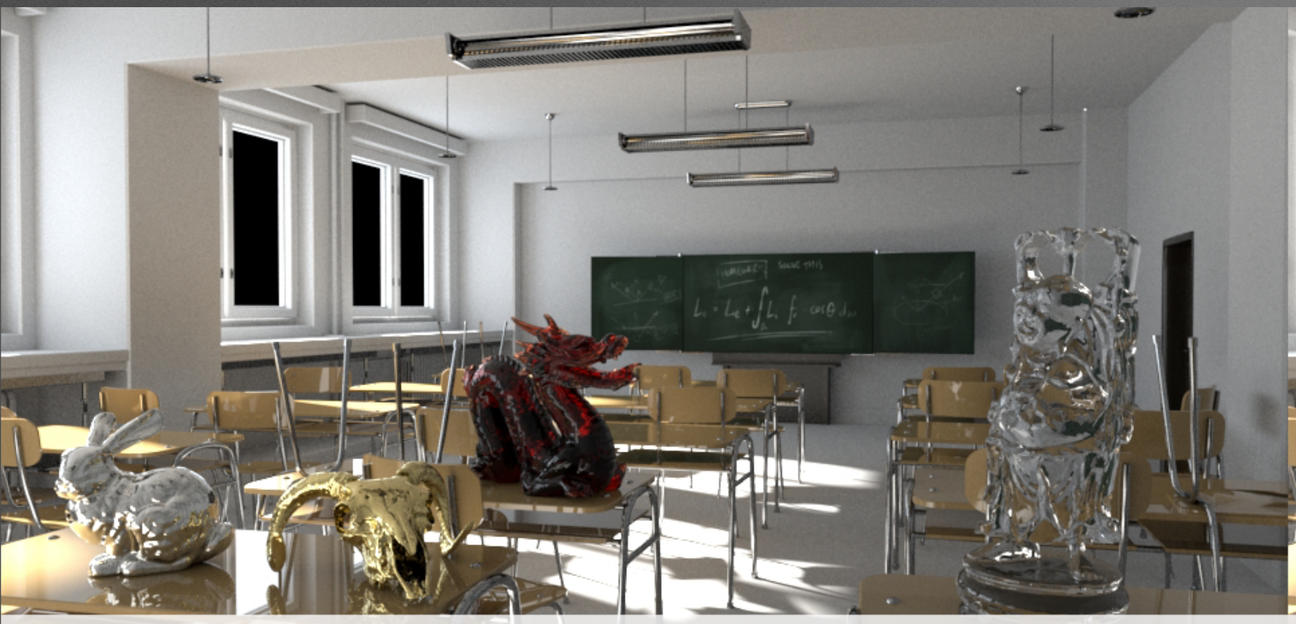
Path Termination

- Probabilistically kill paths when their contribution is low

```
if (rand() < throughput)
  result = trace() / throughput;
else
  result = AI_RGB_BLACK;
```

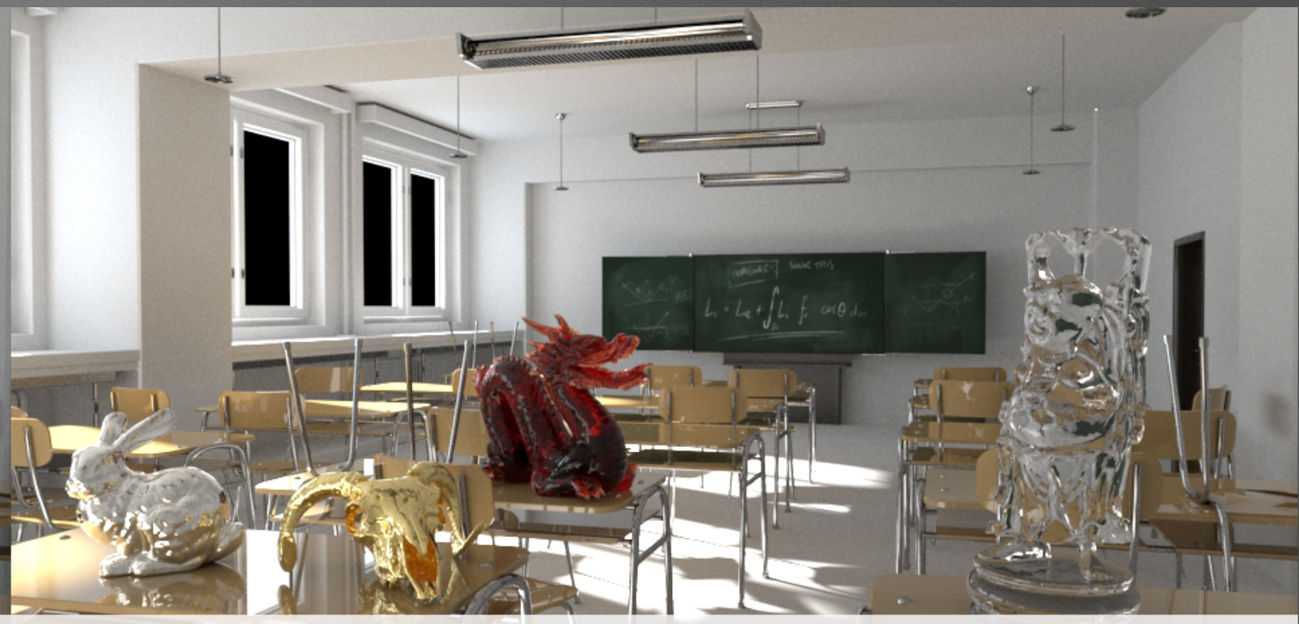

After a few bounces in a typical scene, many paths will become quite dark, so we track the throughput along each path as we trace it using message passing, and randomly choose to stop tracing when continuing wouldn't add much to the image.

Path Termination



Russian roulette enabled, depth 3

3 bounces



Russian roulette enabled, depth 45 - 25% slower

45 bounces - 25% slower

- Alternatively, we can increase the number of bounces with little impact on render time.
- Going from 3 to 45 bounces increases rendering time by only 25%.

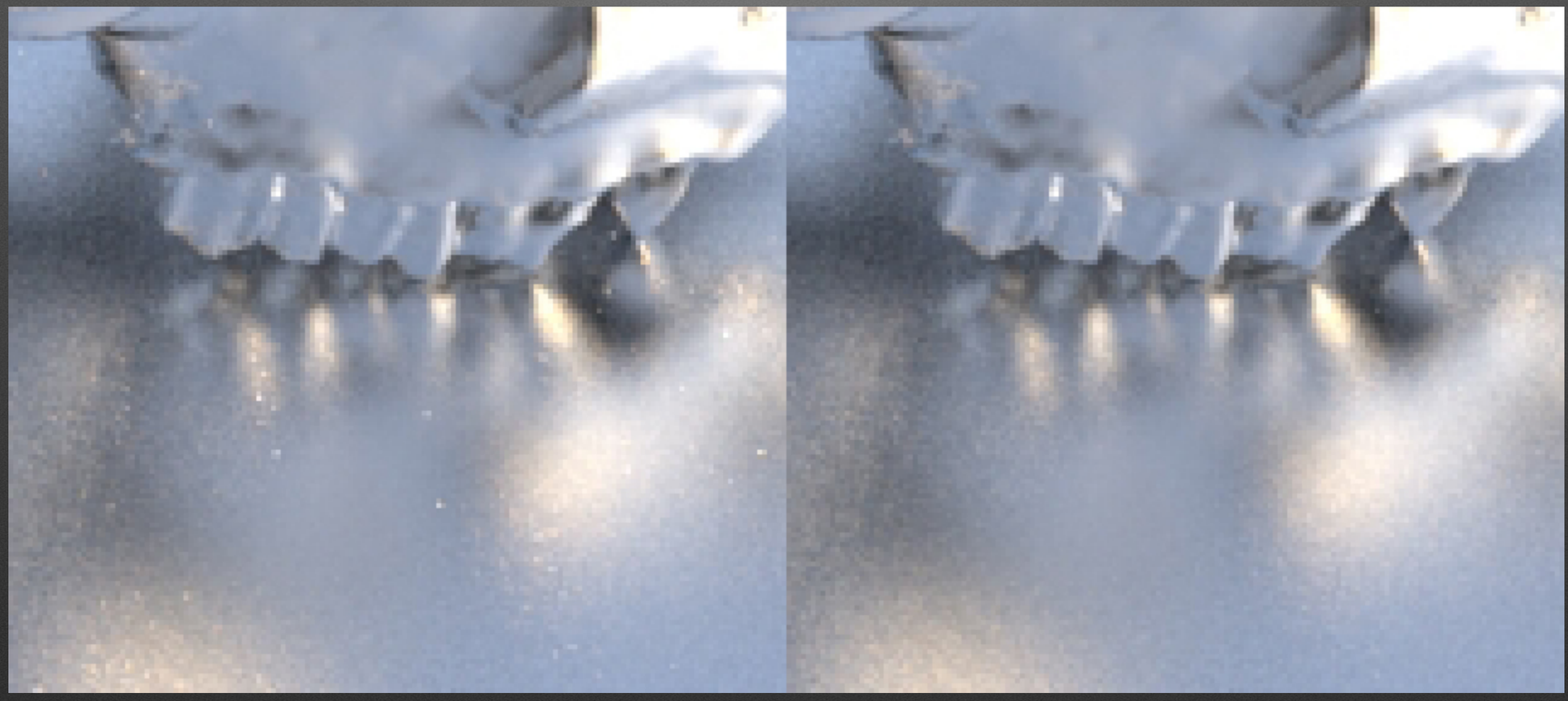
Sample Clamping

- Problem: unlikely paths can create fireflies

RR techniques are great, but you can occasionally get fireflies when sampling paths with a very low probability.

Sample Clamping

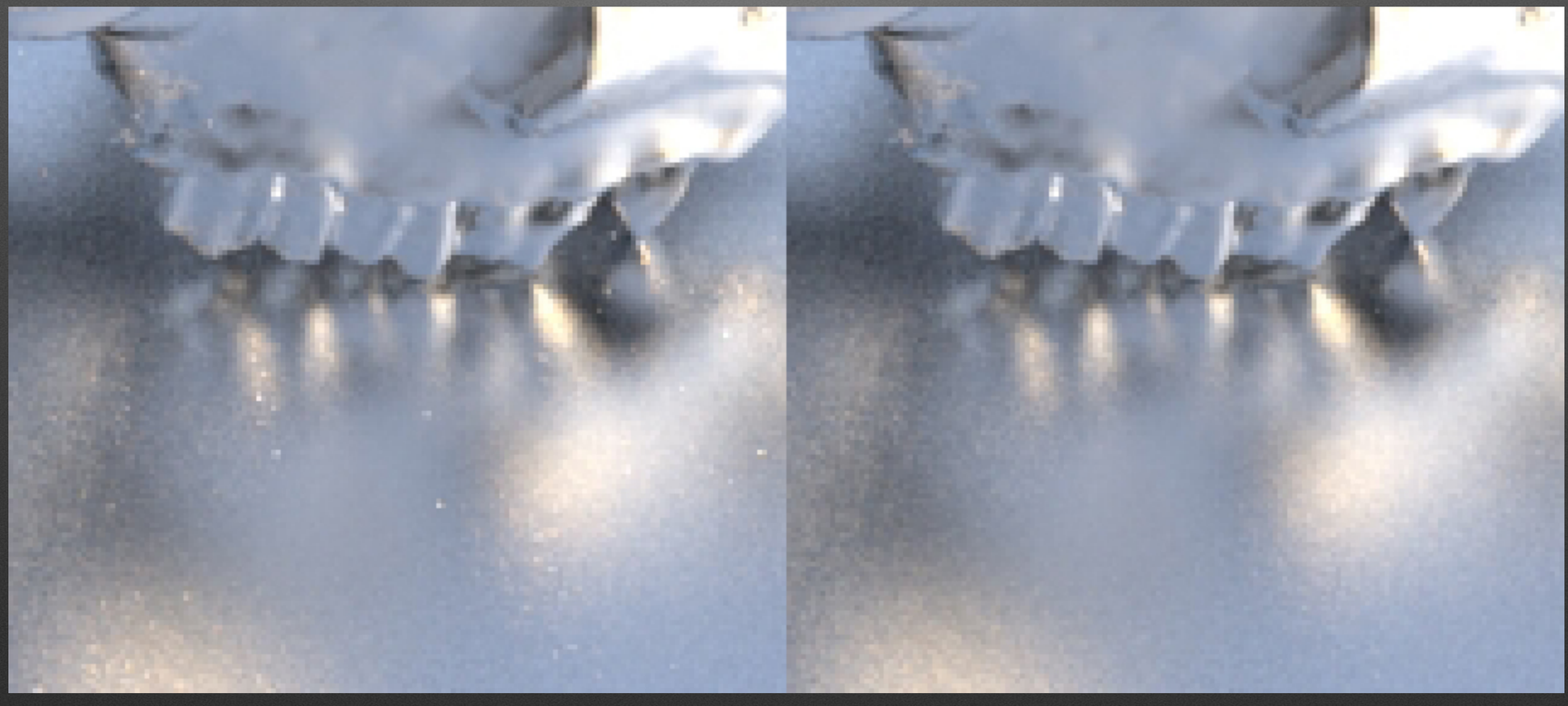
- Problem: unlikely paths can create fireflies



Here we can see lots of little bright pixels in the image on the left.

Sample Clamping

- Solution: clamp paths to some reasonable maximum (10-20)

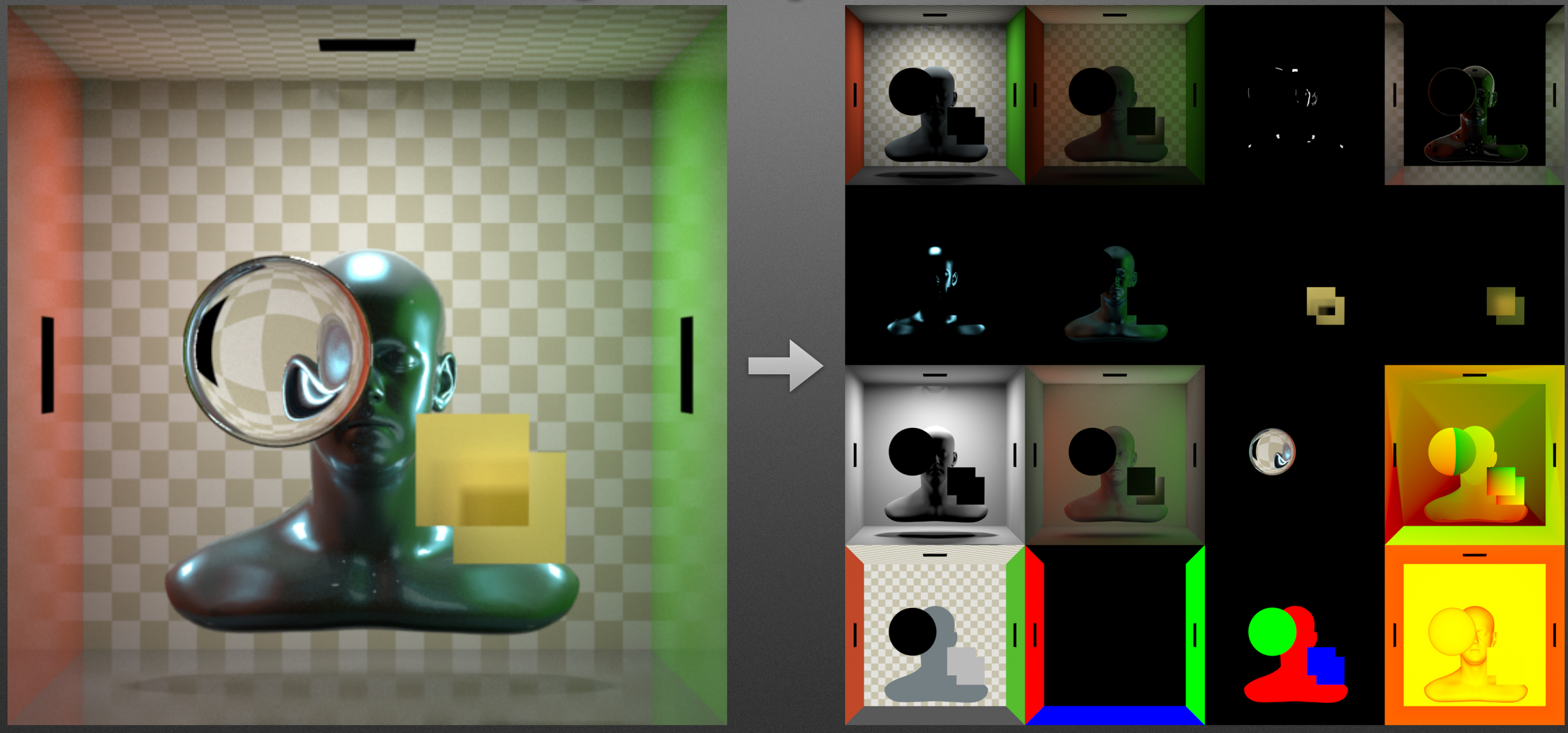


By clamping indirect paths to some reasonable maximum brightness (e.g., 10–20), we can kill the fireflies without otherwise affecting the image.

Outputs

Any shader used in a production pipeline needs to generate a bunch of extra outputs for controlling the render in compositing.

Arbitrary Output Variables



We support 4 main categories of AOVs:
1) Shading outputs (diffuse, specular etc.)
2) Data outputs (normal, uv, depth etc.)
3) ID outputs (user-specified RGB colors, which can be mattes for comp, patterns)
4) Light groups (which I'll come to in a minute)

Arbitrary Output Variables

- Prefer AOVs that sum linearly to beauty render
- Avoid potentially complex relationships (e.g. LPEs)
- Prefer physically correct rebalancing…

- As a general rule, we prefer simplicity and consistency in AOVs as well.
- It's very important that shading and lighting AOVs should just sum up linearly to the beauty render, so that lighting's output to the compositing team is consistent and easy to understand.
- We don't want any complex relationships that would make it more difficult to do that (so no light path expressions).
- Shading AOVs were only really included for legacy reasons (people expect them), but they're dangerous: tweaking them breaks physical correctness immediately, since you're rebalancing the contribution of individual layers without correctly affecting the global illumination in the scene.
- Instead we prefer to use light groups…

Light Groups

- Per-light AOVs
- Each light tagged with an integer attribute
- Accumulate exitant radiance into an array for each layer
- Pass sum of arrays back up the ray tree

- Basically we tag each light with an integer attribute that tells the surface shader what group it belongs to.
- Each group then gets output in a separate AOV.
- By accumulating the lights' contribution at each bounce, we can output the global illumination for each group.
- That means we can rebalance the lighting in comp and have it be mathematically identical to changing the lighting and re-rendering.

Light Groups

- In this slide, the beauty render is in the top left and the other images are the light groups that I've tagged.
- You can see we've got the sun coming in through the window, the sky light, the fluorescents and the lamps in the ceiling.
- This all came from a single render, and each light group includes every bounce of diffuse, specular, transmission and scattering.

Light Groups

...and then we can rebalance them in comp to create different looks, by grading each group separately and adding them back together. The result is just as physically correct as doing another render.

Physically Based Shader Design in arnold

- Arnold is awesome!
- Design a shader library to play to its strengths
- Optimize for realistic AA (motion blur, DOF etc.)
- Enforce physical correctness
- Have fun!

- So to recap…
- Arnold is awesome!
- If we're designing a shader library for it, we should play to its strengths and follow its philosophy.
- We should optimize for high sample counts and this allows us to use tricks like Russian Roulette to dramatically reduce render times.
- Obviously, we enforce physical correctness as much as possible.
- And hopefully we can have some fun while we're doing it!

Physically Based Shader Design in arnold

- Acknowledgements:
  - Marcos and the rest of the “Solids”
  - Jules Stevenson for Kettle
    - https://bitbucket.org/Kettle/kettle_uber/wiki/Home
  - Espen Nordahl for his great contributions
  - Everyone who keeps making great images

- I should thank Marcos, obviously, and the rest of the Solid Angle team for making Arnold so awesome in the first place and for their support in the development of these shaders.
- Jules Stevenson’s Kettle shaders were a massive help when first learning how to use the Arnold API.
- And a big shout out to Espen Nordahl for adding many new shaders to the library and fixing some stupid mistakes I made.
- And everyone who’s ever made a cool picture with the library, reported a bug, or just nagged me to make it better.

Physically Based Shader Design in arnold

- Credits:
  - Images courtesy of Nozon, Pascal Floerks, Brett Sinclair, Psyop, Phil Amelung & Daniel Hennies, Storm
  - Head model by Lee Perry-Smith

I'd also like to thank the artists who made some of the images I've used in this presentation. Please see the course notes for details.

Physically Based Shader Design in arnold

- Loads more detail in the course notes!
- https://bitbucket.org/anderslanglands/alshaders

- There's loads more detail and extra goodies in the course notes.
- And of course you can check out the code on Bitbucket.
- I apologize if some of it makes your eyes bleed; I promise I'll clean it up eventually.

Thank you!