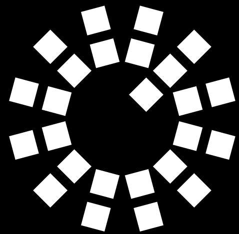

SONY PICTURES
imageworks
25TH ANNIVERSARY

30 JULY – 3 AUGUST Los Angeles
SIGGRAPH2017

Volume Rendering Techniques at Imageworks

Christopher Kulla

SIGGRAPH 2017

Thank you for the introduction.

My name is Chris Kulla and I am going to be talking about how Volume Rendering works in our renderer at Sony Imageworks.

Outline

System level overview of volume rendering in our renderer:

- Medium Tracking
- Ray Marching
- Importance Sampling

We've heard a lot about the theory of volume rendering so far, but I want to give more of a systems level overview of how volume rendering is implemented in our renderer.

Before I begin, I do want to clarify that while our renderer does share a common ancestry with the Arnold renderer from Solid Angle - I am going to be talking about our version of it which has diverged for quite some time now. I'm actually going to be talking more about this divergence on Wednesday afternoon in the Production Path Tracing course.

For today we're going to focus on volumes, specifically these three areas:

Medium Tracking, Ray Marching, And Importance Sampling

Medium Tracking

(Finding relevant volumes along each ray)

The first topic is medium tracking. Now I apologize for the overloaded meaning of the word tracking here. When I say tracking I mean “keeping track of”. That is, how do we discover which volumes are relevant along each ray.

Medium Tracking

Our renderer distinguishes two cases:

- Volumes Primitives (IOR $= 1$, no boundary)
  - Overlap is *additive*, all mediums are summed
  - Need list of overlapping intervals along the ray
- Surface Defined Volumes (IOR ≥ 1, with boundary)
  - Overlap is *exclusive*, one medium "wins"
  - Priority defined by the artist
  - Each surface hit must know medium on either side

We have different tracking techniques for each case.

Our renderer actually distinguishes between two cases that serve different use cases.

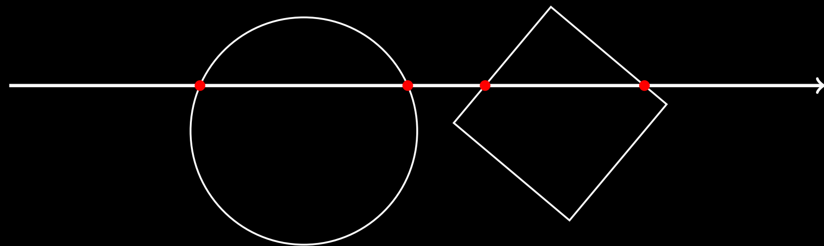
First we have volume *primitives*, which don't have any boundary. For these the overlap rules are additive: meaning all the properties have to be summed. To do this - we keep track of a list of overlapping intervals along the ray.

Second, we have surface defined volumes. These are actually defined through surface shading, in other words we hit the boundary and the surface shader will tell us if there is a medium inside the shape or not. Here the overlap rules are exclusive and driven by a priority system.

So let me go through the implementation of each case.

Medium Tracking - Volume Primitives

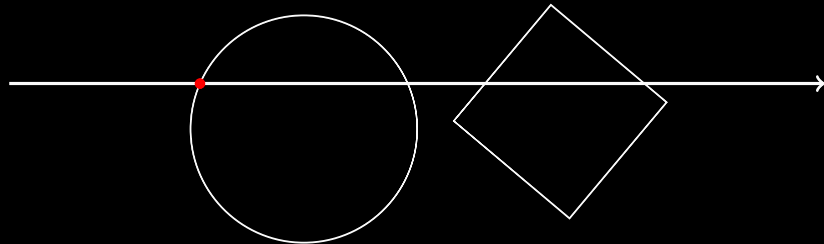
Surface primitives return hit *points*



```
struct surfhit { float t; int primID; }
```


A raytracer represents surface hits by measuring the distance along a ray to a particular point. This is usually represented as a single value t and some identifier of the primitive we hit.

Surface primitives return hit *points* (keep nearest)

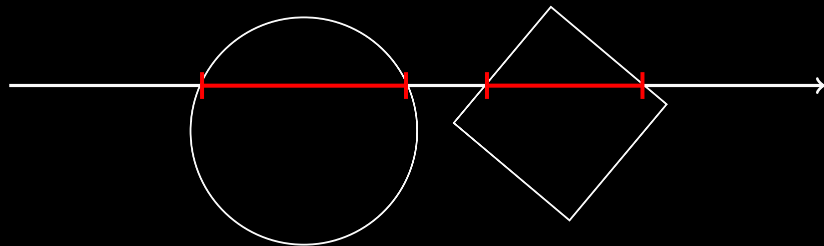


```
struct surfhit { float t; int primID; }
```


Because the ray tracer usually only cares about the nearest primitive, that's all it needs. Any hit that's closer than the one found so far updates the hit, otherwise its discarded.

Medium Tracking - Volume Primitives

Volume hits return *intervals*



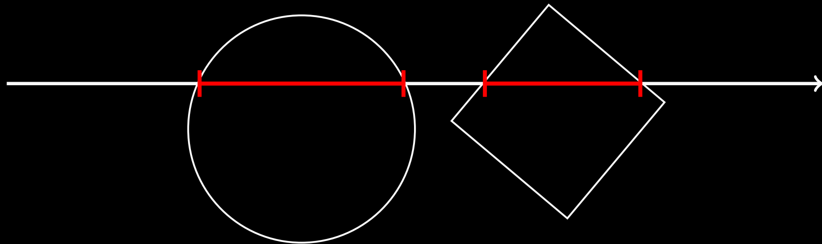
```
struct volhit { float t0, t1; int primID;                  }
```


So for volume *primitives* in our renderer we actually return something different which is *intervals* instead of hit points.

That means they define a start and end along the ray.

Medium Tracking - Volume Primitives

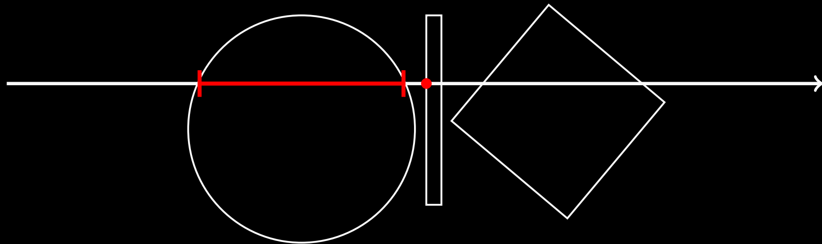
Volume hits return *intervals* (keep all)



```
struct volhit { float t0, t1; int primID; volhit* next; }
```


And because volumes have transmission, we actually keep *all* the intervals we find. We store this as a linked list, which is usually a terrible data structure but in this context, the list is really short lived and the memory comes from a memory pool so it stays very cache friendly.

Volume hits return *intervals* (keep all unoccluded)

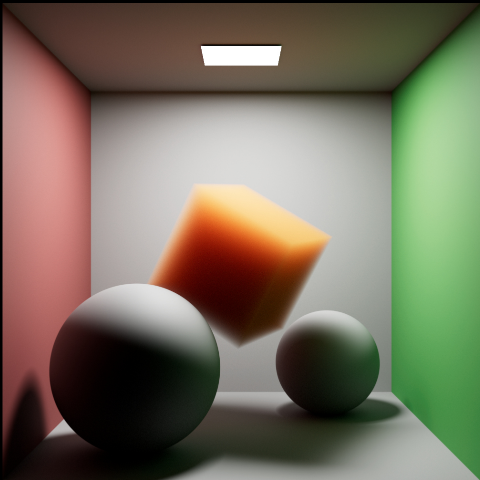


```
struct volhit { float t0, t1; int primID; volhit* next; }
```


Of course, if we do hit a surface we can discard any volume intervals that lie beyond that point.

In our implementation this happens after we return from the ray tracing call so we just need to do a single pruning pass.

Medium Tracking - Volume Primitives

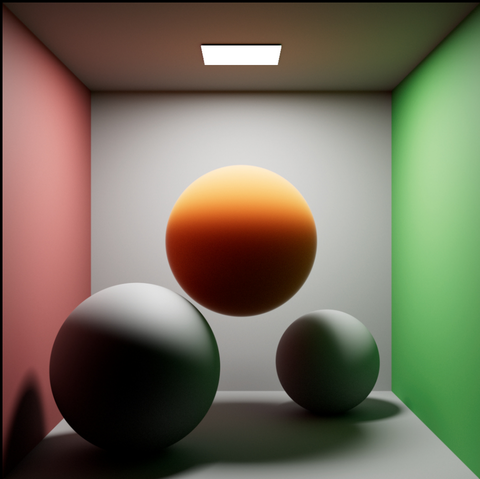


Convex shapes are easy as the regular intersection routines already produce a front and back hit.

Here are some examples of the various volume primitives we support:

Convex shapes are the easiest to implement because the intersection test already returns a front and back hit point. So we just make an interval instead of treating them as two potential surface hits.

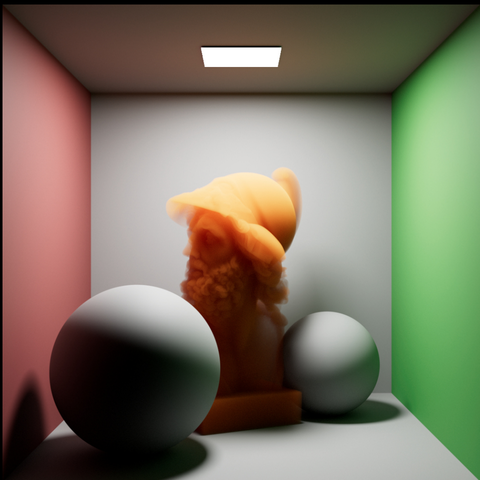
Medium Tracking - Volume Primitives



Convex shapes are easy as the regular intersection routines already produce a front and back hit.

Here's a sphere.

Medium Tracking - Volume Primitives



Mesh defined volumes require counting intersections (even or odd).

Bounding box limits ambiguous cases.

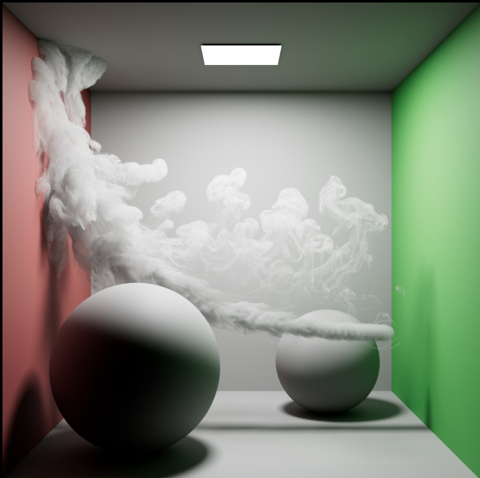
Each ray figures out overlap from scratch, no history required!

Meshes are a bit more complicated. In order to turn surface hits into intervals, we actually need to count intersections. Depending upon if we hit an even or odd number, we can decide how to pair up those hits and turn them into intervals.

As you can imagine, there are some ambiguous cases like if you only found a single intersection but started outside the shape. So we always use the objects bounding box as a kind of sanity check.

The nice thing about this approach is that the overlap between the ray and volume is figured out from scratch without any kind of history required. On the other hand, its a bit more expensive because each ray needs to trace all the way through the mesh each time.

Medium Tracking - Volume Primitives



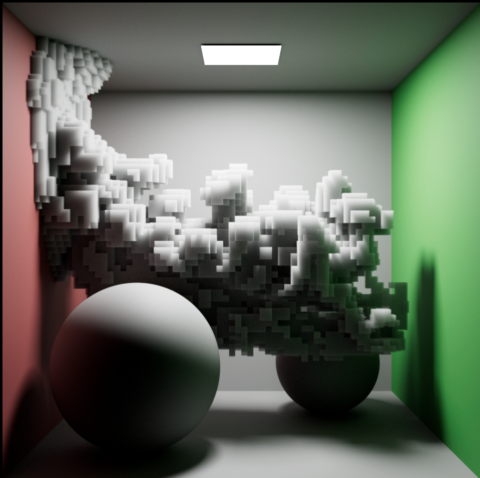
Sparse grids use a two level structure to minimize marching through empty space.

Dense grids can be made sparse on load.

The next and actually most common volume primitive is the sparse grid. This is what we use to render voxel data, usually from fluid simulations.

We take advantage of the sparse structure of most volumes and organize the data into two levels, blocks and voxels. Even if the input data is dense for some reason, we convert it to be sparse as we load the data.

Medium Tracking - Volume Primitives



Sparse grids use a two level structure to minimize marching through empty space.

Dense grids can be made sparse on load.

Here is what the block structure of that fluid simulation looks like. This is what gets intersected for finding overlap. The regular 3D grid traversal algorithm already provides the hits in order, we just turn them into intervals as we go.

We also make sure to merge the intervals as we go, so if you go through several blocks at once we still just end up with a single interval.

Medium Tracking - Surface Defined Volumes

- Surfaces can define an interior medium
- Artists describe intent through priorities
- Also defines IOR at boundaries

That was volume primitives, now lets talk about surface defined volumes.

This is the case where the surface shader is the one responsible for deciding what the medium properties are.

From the point of view of the artist, everything is contained in the surface shader. In this picture I just have a glass shader and a liquid shader.

When the two meshes overlap each other, we rely on the artist to define which takes takes precedence by giving each a priority.

Medium Tracking - Surface Defined Volumes

- Surfaces can define an interior medium
- Artists describe intent through priorities
- Also defines IOR at boundaries
- Critical for correct rendering of liquids

This is really critical for rendering liquids correctly. Here I've shown how things look if you just model the liquid as slightly smaller than the glass. All kinds of extra reflections and refractions happens in that small air gap.

Medium Tracking - Surface Defined Volumes

- Surfaces can define an interior medium
- Artists describe intent through priorities
- Also defines IOR at boundaries
- Critical for correct rendering of liquids
- Liquid is modeled slightly overlapping glass

By modeling the liquid slightly overlapping the glass, we get the right picture because we get a clean transition from one medium directly to the next and also because we can get the right ratio of refractive indices.

Medium Tracking - Surface Defined Volumes

"Simple Nested Dielectrics in Ray Traced Images"
Schmidt and Budge, JGT 2002

- Each ray maintains a stack of mediums it has entered
- At each interface, decide which medium "wins"
- Some hits will be discarded (but still update the stack)

Our implementation pretty much follows the only paper I know of on this topic. It was published in 2002 by Charles Schmidt and Brian Budge.

The basic idea is that each ray maintains the stack of all mediums its entered so far.

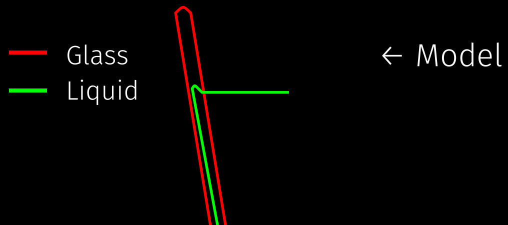
At each interface there are rules to decide which medium "wins" (based on their priority). This is what decides if the surface hit is actually accepted or not.

But either way, the stack is still updated so we remember that we've entered or left the given medium.

Medium Tracking - Surface Defined Volumes

"Simple Nested Dielectrics in Ray Traced Images"
Schmidt and Budge, JGT 2002

- Each ray maintains a stack of mediums it has entered
- At each interface, decide which medium "wins"
- Some hits will be discarded (but still update the stack)



← Model

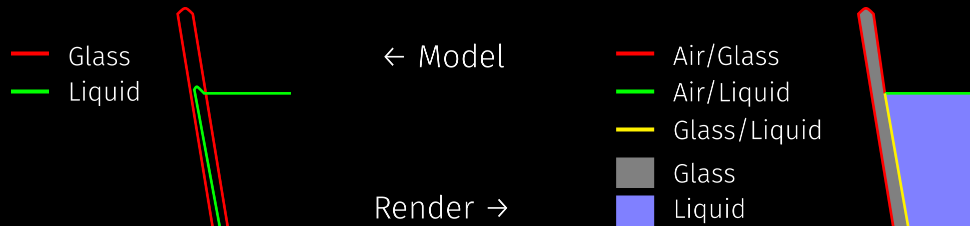
Here is a cross section view of how we might model the example from before.

There are just two meshes here, each with just a surface shader. The important thing is how the liquid is modeled to slightly intersect the glass.

Medium Tracking - Surface Defined Volumes

"Simple Nested Dielectrics in Ray Traced Images"
Schmidt and Budge, JGT 2002

- Each ray maintains a stack of mediums it has entered
- At each interface, decide which medium "wins"
- Some hits will be discarded (but still update the stack)



What this turns into during the render is more like this. Because the glass has higher priority than the liquid, it "wins" where they overlap.

Also because we maintain that stack, when we leave the glass and travel into the liquid, we already have the liquid in our stack.

That tells us we are making a transition from glass to liquid and therefore we can calculate the right IOR for that boundary without having had to model it separately.

As far as the artist is concerned, all they had to do was define the IORs relative to vacuum - which is how they are used to thinking about those numbers.

Medium Tracking - Surface Defined Volumes

Some details not described in paper:

- Handling of shadow rays
- Rules for equal priorities
- Establishing starting medium

When implementing all this we ran into a few questions that weren't really covered in the paper. So I'd like to discuss those now.

Medium Tracking - Shadow Rays

Needed for approximate caustics or embedded lights:

- Ray tracer discovers hits in unsorted order

The first is what to do on shadow rays. Now in reality, shadow rays shouldn't be able to go through refractive interfaces – but in production rendering we sometimes allow this because its a much cheaper way to approximate caustics.

It also comes up when lights are embedded inside mediums.

The ray tracer discovers shadow hits in unsorted order...

Medium Tracking - Shadow Rays

Needed for approximate caustics or embedded lights:

- Ray tracer discovers hits in unsorted order
- Defer any hits of priority $\neq$ Off

…so we actually need to defer the processing for all the hits that have a non-default priority.

Medium Tracking - Shadow Rays

Needed for approximate caustics or embedded lights:

- Ray tracer discovers hits in unsorted order
- Defer any hits of priority $\neq$ Off
- Resolve intervals once all hits are known (if not otherwise shadowed)

So we maintain a list of all the hits we get and then we sort them into intervals once we know all of them.

Of course that sorting step is only needed if we didn't already get shadowed by some opaque object.

Medium Tracking - Shadow Rays

Needed for approximate caustics or embedded lights:

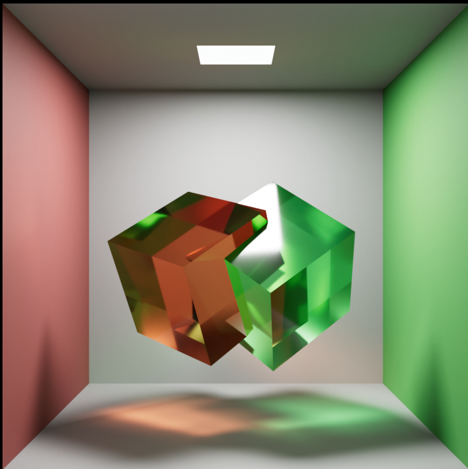
- Ray tracer discovers hits in unsorted order
- Defer any hits of priority $\neq$ Off
- Resolve intervals once all hits are known (if not otherwise shadowed)
- Good performance depends on using priorities sparingly

This probably sounds a bit expensive - and it definitely can be. So we make sure we just use priorities only when we really need to.

And that's really only when shapes need to overlap for some reason, which is normally a small fraction of what is in the scene.

Its also worth mentioning that the expense only applies to rays that hit these objects. There's no cost if you don't get near these objects.

Medium Tracking - Different Priorities



Left > Right

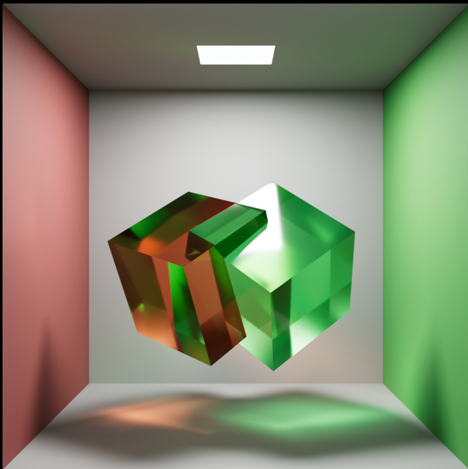
- Red cube has higher priority
- Red cube visible inside green cube
- Green cube invisible inside red cube

So here we have the priority system in action:

I have two partially overlapping cubes. The shader is just setting plain absorption with no scattering so we can see what's going on inside.

When the red cube has higher priority, the green cube isn't visible through it.

Medium Tracking - Different Priorities



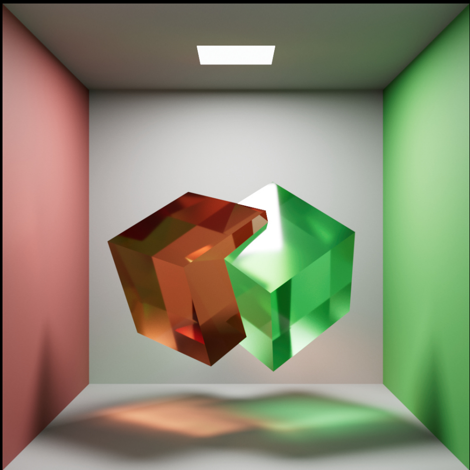
Left < Right

- Green cube has higher priority
- Green cube visible inside red cube
- Red cube invisible inside green cube

And when the green cube has higher priority, the red cube disappears inside of it.

So as you write the code, you then have to decide - how should we handle the case where both priorities are equal? The paper didn't really mention this.

Medium Tracking - Equal Priorities



Left = Right

- All interior hits are ignored!
- Allows merging objects at render time
- Leaving a medium maintains current properties if next object on stack is of equal priority

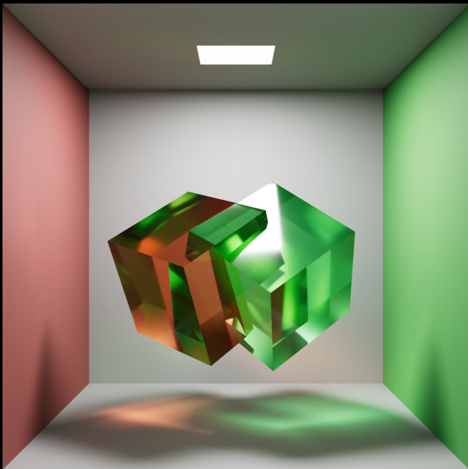
We decided it would be most helpful if equal priority mediums actually “merge”. That means all of the internal hits disappear.

This actually ended up being really handy for getting rid of internal geometry.

Its worth pointing out that when we leave an object, if the next one on the stack has the same priority - we actually keep the medium properties from the object we are leaving.

So in this example - leaving the red cube while you are still inside the green cube keeps the ray tagged with the red medium properties. That’s important because otherwise you could see a discontinuity there.

Medium Tracking - Equal Priorities



Left = Right = Off

- All interior hits are ignored!
- Allows merging objects at render time
- Leaving a medium maintains current properties if next object on stack is of equal priority
- Priority=Off objects are always visible

Its also worth mentioning that when we reach our highest possible priority the internal hits are visible no matter what.

This is important because it keeps that case fast but also because it makes the rules simpler for the artists. Priority Off means the object will be visible no matter what.

Priorities are stored as integers (smaller means higher priority).

To avoid confusion, we hide the integers from artists and present a drop-down menu:

- Off (0)
- Very High (50)
- High (100)
- Normal (200)
- Low (300)
- Very Low (400)

Left gaps in numbering for special cases (none encountered so far).

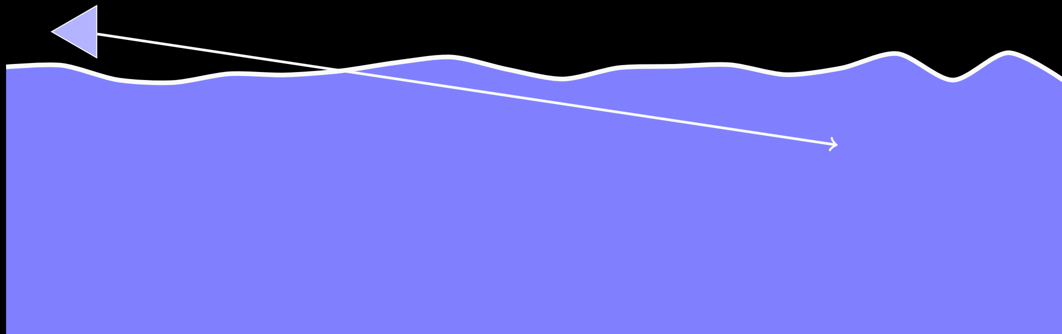
Just like in the paper - we decided to internally store priorities as integers, where smaller means higher priority. That convention is nice because it makes 0 be the default.

But in conversation with artists this was always very confusing. So we decided to make our UI show just a few hardcoded levels with names instead of numbers.

You'll notice that we left ourselves some gaps in how we numbered these, because we were worried we would run into corner cases where we had to slot an object between two already decided priority levels – but this hasn't happened so far.

Medium Tracking - Initialization

Rays that start in a vacuum behave correctly

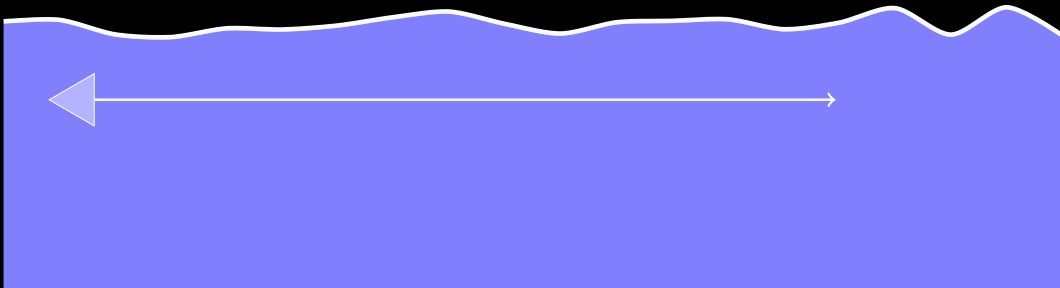


The last implementation detail to discuss is how we actually decide the starting state of the ray stack.

When the camera starts in empty space, everything is fine. An empty stack is the correct start state.

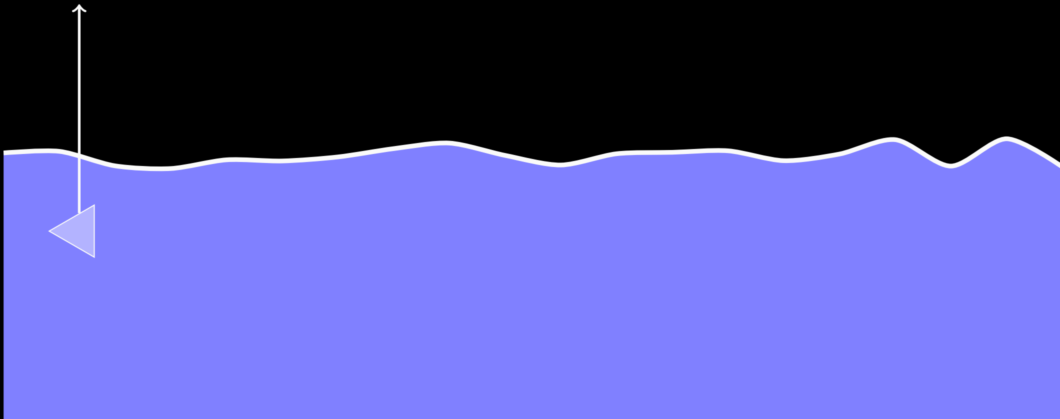
Medium Tracking - Initialization

Rays that start inside a medium need to know the starting stack



But if the camera start inside a medium (underwater in this case) we need to have some way to specify the starting stack.

Notice how this first ray here doesn't even intersect the surface of the water all. Even if we made sure to add sides to close the water volume - we might hit some other objects first.

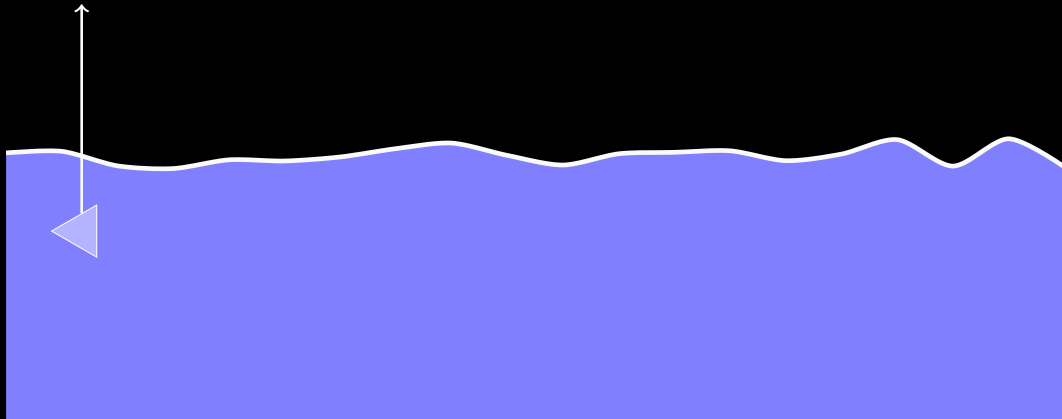


Our solution is to fire a single probe ray at the start of the render, going all the way through all surfaces.

Why did we pick up? Well its mostly motivated this water case, which is probably the only case where this is needed. Ocean surfaces are typically modeled as displaced planes...

Medium Tracking - Initialization

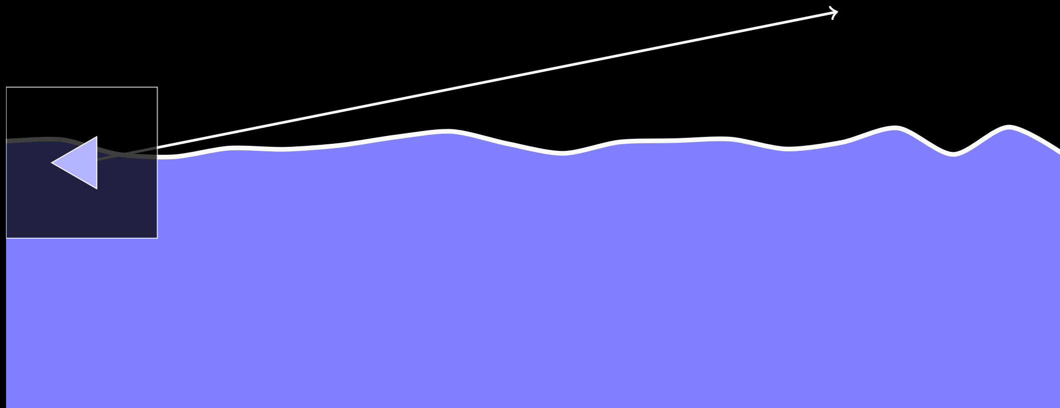
Water surface doesn't need to be closed!



…and it was helpful to our artists not to have to worry about somehow closing off the surface with fake walls.

Medium Tracking - Initialization

High priority empty medium can act as clipping plane



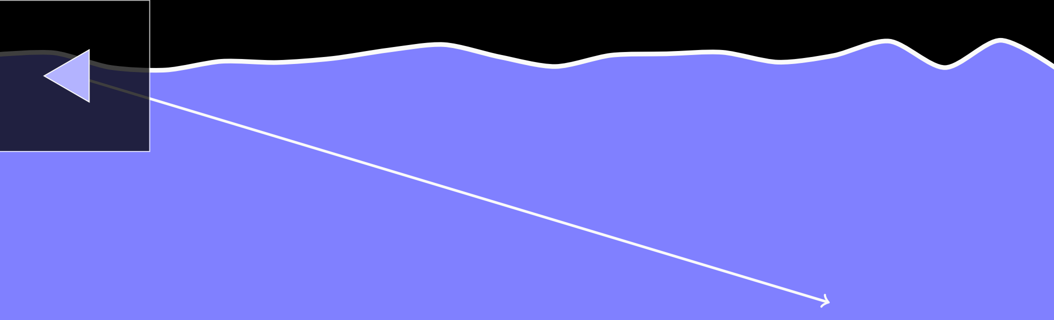
Another case we ran into was what to do when the camera is hovering near the surface and wants to see above and below the water at the same time.

We happen to be working on a shark movie at the moment, so this case really did come up in production - even though I can't show any renders of it today.

We use a high priority box that acts like a clipping plane. This makes sure we can cross the water to find its medium without actually registering a hit right away. This means we can see both above and below the water in the same frame. And the artist never has to worry about manually tagging anything.

Medium Tracking - Initialization

Camera rays can see above and below water in the same image

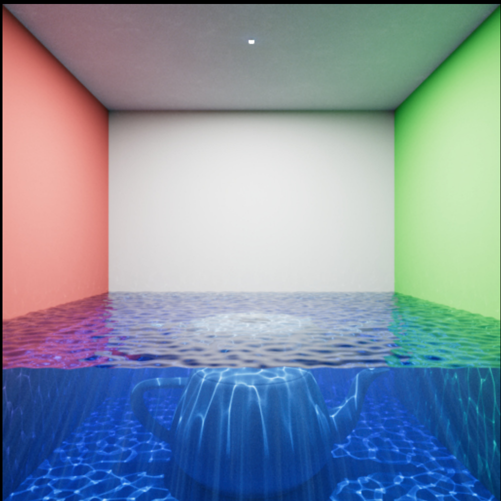


Medium Tracking

Camera outside

Like I said I can't show the images from the real production shots this came up on because the movie is not out yet. So we'll have to make do with a Cornell box.

Medium Tracking



Camera outside, see below the water

Here the camera is outside the water, but we use a high priority box around it to be able to see underwater

Medium Tracking

Camera inside, see above water

Now the camera starts inside the water, but we can still see above the water

Medium Tracking

Camera inside

And now the camera is fully inside. Again, this plane is animating up - but I didn't ever have to manually tell the camera it was inside any particular medium – the renderer figures it out automatically.

Medium Tracking - Summary

Two ways of representing volumes:

- Volume Primitives (Additive overlap)
  - Clouds, Smoke, Fire
- Surface Defined Volumes (Exclusive overlap)
  - Glass, Liquids, Subsurface

Both methods co-exist. Volume primitives combine additively with the surface defined medium decided by the current stack (e.g.: under-water explosions).

That pretty much covers the topic of medium tracking. Just to summarize, our renderer has two kinds of volumes:

Volume primitives like clouds, smoke or fire that combine additively

And surface defined volumes that have exclusive overlap. We use those for things like glass, liquids or subsurface.

Of course both of these methods coexist. So going back to the underwater example, any explosion that happens underwater will properly combine additively with the ambient water medium.

Ray Marching

(Capturing volume properties along each ray)

Now lets move out to ray marching. All I just described was how to find where the volume are along the ray. Now we need to actually execute the volume shaders.

Ray Marching

Each segment captured by medium tracking defines:

- Interval: $[t_{\text{start}}, t_{\text{end}}]$
- Shader to execute (Volume primitives)
- Constant properties (Surface defined volumes)
- Step size

All segments are split into non-overlapping intervals and sorted. Where volumes overlap, a single segment can refer to multiple primitives.

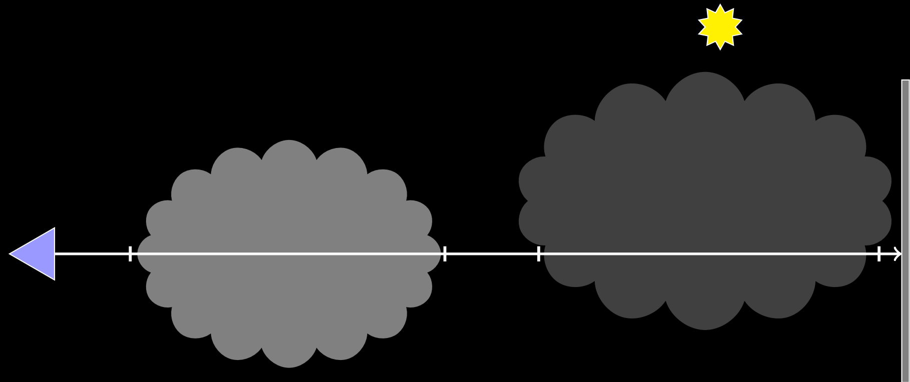
Medium tracking provides a list of segments, where each one defines:

Start and end intervals

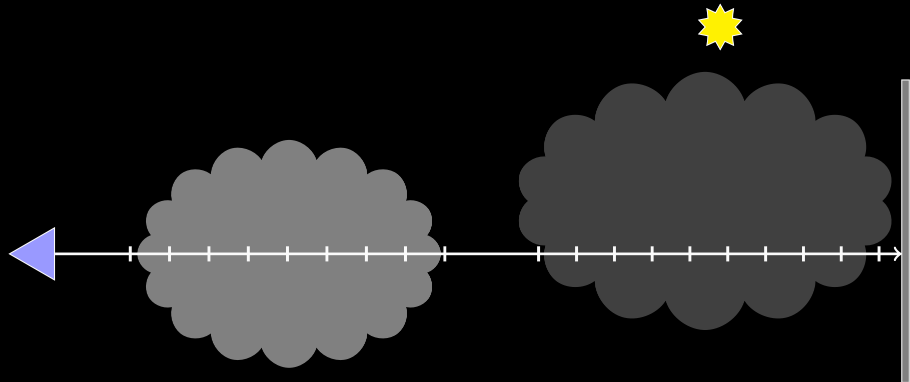
Either a shader to execute if the segment was from a volume primitive or a set of medium properties if we came from a surface defined volume.

And lastly a step size that tells us how many times we need to run the shader if we have one. For example the sparse grid primitive defaults this to about 1 step per voxel.

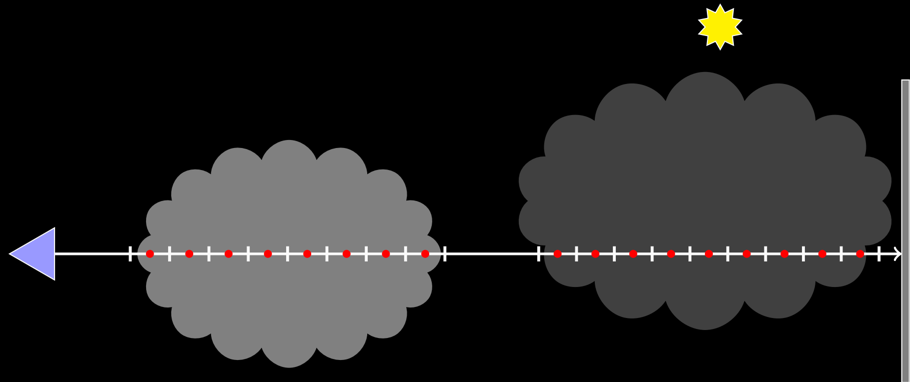
We also have to split the segments into disjoint spans, where each span has the list of all the primitives it covers.



Here is just a visual of what I just said. We start with some intervals given to us by the medium tracking phase.



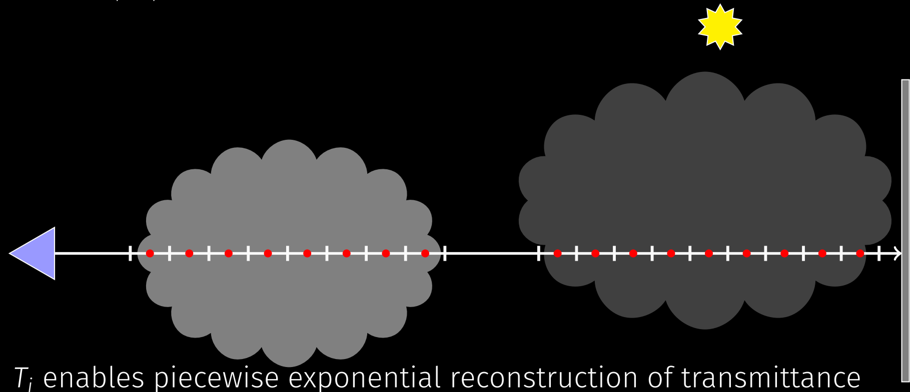
We break those intervals into segments that we assume to be homogeneous by the step size.



And then start executing shaders from front to back

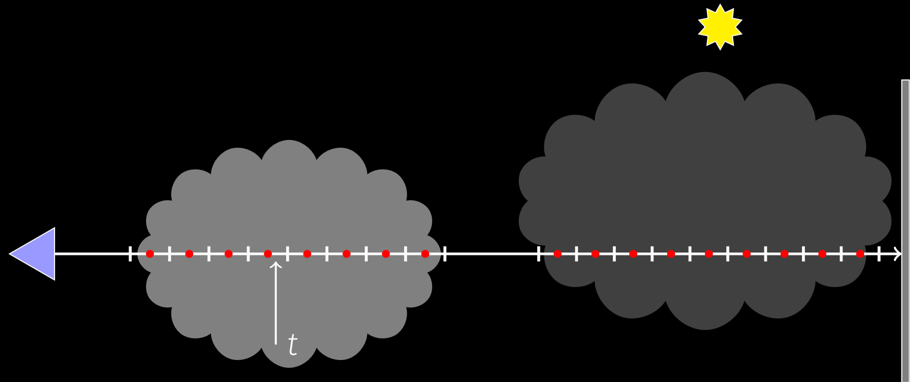
Ray Marching

Store σ_{s_i},σ_{t_i},$T_i = T_{i-1}e^{-\sigma_{t_{i-1}}\Delta_{i-1}}$ and phase function



At every step, we store the medium properties and estimate the transmittance so far.

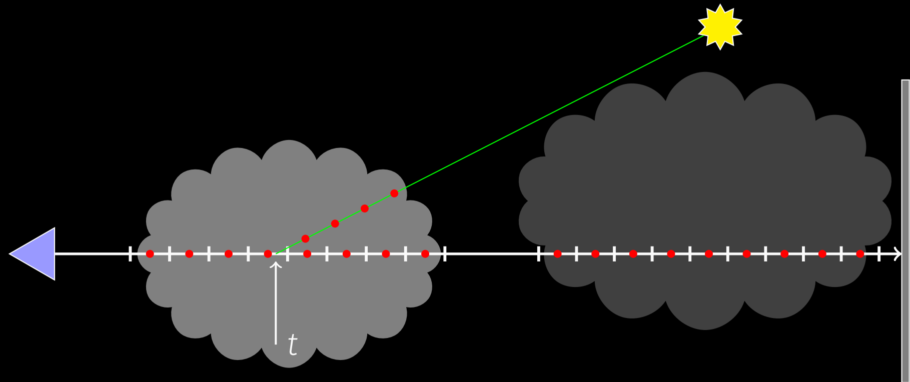
This discretized transmittance together with the extinction inside that step means we can reconstruct a piecewise exponential representation of the transmittance along the ray.



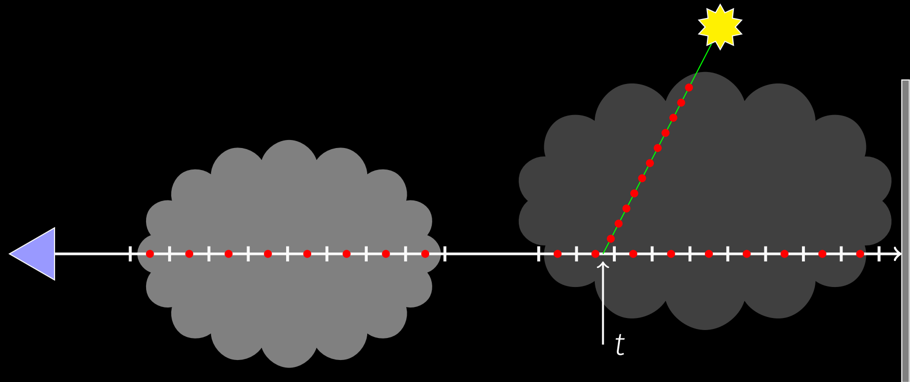
This means that given any distance t along the ray, we can quickly locate it by binary search

Ray Marching

Can calculate lighting at any point (using any pdf)

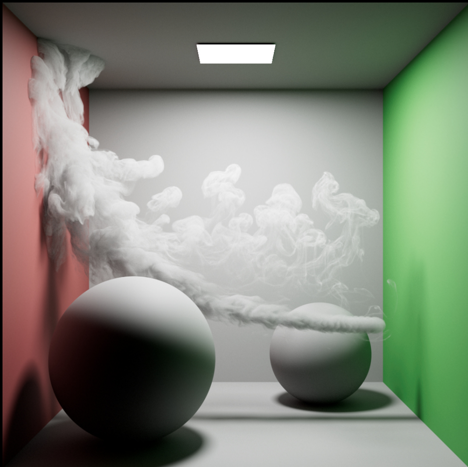


And then we have all the information we need to compute the lighting integral.



We are free to choose those distances however we like - and I will talk about that in a minute.

Ray Marching Bias



Step Size = 1 Voxel

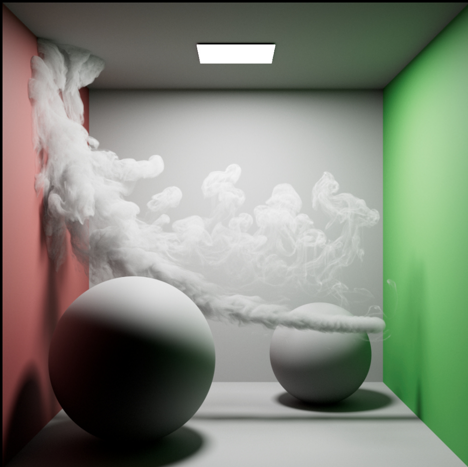
Ray marching underestimates transmittance when step size is too large (bias).

I just want to briefly discuss the bias that comes from doing ray marching this way. When we execute a shader over a small step, we are making the incorrect assumption that the volume in homogeneous in that step.

When the step size is too large, we are actually going to underestimate transmittance. This is of course a form of bias, which is something we've been trying to move away from in the shift to path tracing techniques.

But the visual error you get here is a bit different from other forms of bias in rendering.

Ray Marching Bias

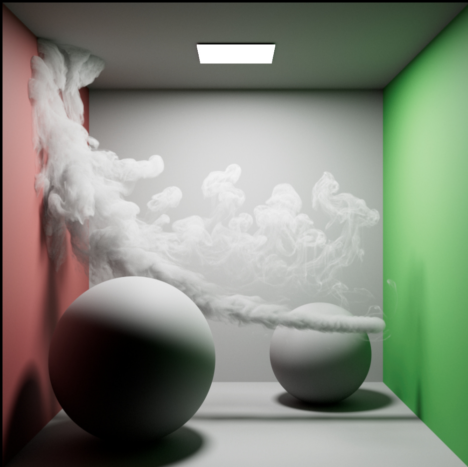


Step Size = 2 Voxels

Ray marching underestimates transmittance when step size is too large (bias).

Here I am increasing the step size by multiples of the voxel size

Ray Marching Bias



Step Size = 4 Voxels

Ray marching underestimates transmittance when step size is too large (bias).

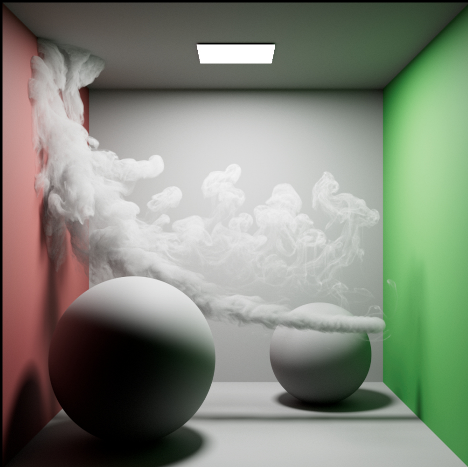
Slight change in density is not always objectionable!

But here even at 4 voxels per step, we still don't really see any significant change – even though the render was 4 times faster.

Its worth pointing out that the error only manifests itself where the assumption that the medium is not homogeneous over the step is not true.

Anywhere that the volume *is* homogeneous, ray marching with a large step is fine. In fact we take advantage of this if we detect for example that a particular block of voxels all have the same value.

Ray Marching Bias



Step Size = 8 Voxels

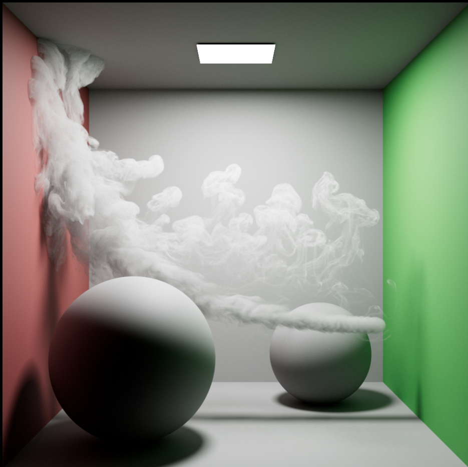
Ray marching underestimates transmittance when step size is too large (bias).

Slight change in density is not always objectionable!

Step size trades accuracy for speed.

Once we go too far, we can start to see visible differences to the reference – but the picture is still plausible. In other words, we are exchanging speed for a less accuracy but we don’t get any extra noise.

Ray Marching Bias



Step Size = 16 Voxels

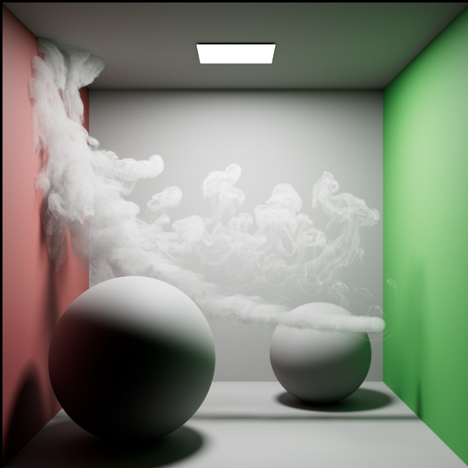
Ray marching underestimates transmittance when step size is too large (bias).

Slight change in density is not always objectionable!

Step size trades accuracy for speed.

I'll keep going...

Ray Marching Bias



Step Size = 32 Voxels

Ray marching underestimates transmittance when step size is too large (bias).

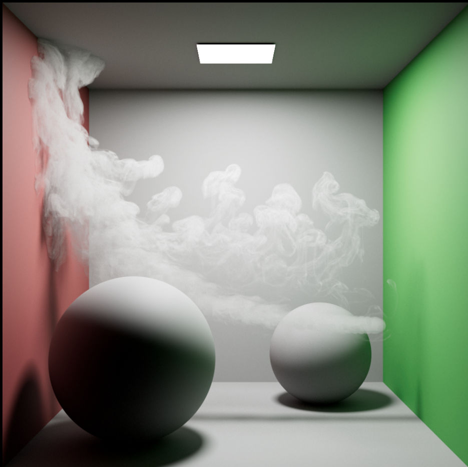
Slight change in density is not always objectionable!

Step size trades accuracy for speed.

Now the renders are starting to look different than our reference, but are still nice to look at. Which brings me to the second point, that the bias you get here is not necessarily objectionable. The artist might be ok with trading a slight loss in density in exchange for a faster render.

Sometimes you can nudge the volume parameters to make up for this a little bit - but not always of course.

Ray Marching Bias



Step Size = 64 Voxels

Ray marching underestimates transmittance when step size is too large (bias).

Slight change in density is not always objectionable!

Step size trades accuracy for speed.

Unbiased methods trade speed for extra noise (beneficial for higher order scattering)

Now in contrast, the unbiased methods of ray marching that you heard about earlier always give the correct image but you can only trade performance for extra noise.

So far we've been happy with the behavior of this biased ray marching approach and the bias hasn't been too much of an issue…

…but it would be nice to do a more detailed analysis of the bias/variance trade-offs between this approach and the unbiased estimators we heard about earlier.

Decoupled Ray Marching

A few more things happen during the ray marching stage:

- Surface shaders are run (for all transparent hits)
- Emission is summed up
- Alpha and holdouts are computed

No lighting calculations have been performed yet!

We call this process *decoupled ray marching*.

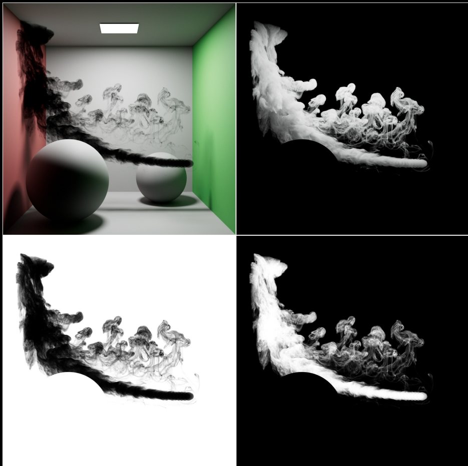
A few other things happen during this ray marching phase:

We also run all the surface shaders, potentially going through any transparent hits. This means we can do the volume light sampling just once, even if there were discrete opacity changes like from hair for example. This really helps when you have fur embedded inside a volume.

We also add up any emission (for cases like fire), and we calculate any alpha and any holdouts if this is a camera ray.

But just to be clear, we still haven't done any lighting calculations. We're just gathering information about what to do in the next phase. That's why we call this *decoupled ray marching* because the ray marching is *decoupled* from the lighting calculations.

Holdouts



Important for isolating volumes in compositing.

Holdouts defined by σ_h (same units as σ_a)

Holdout through step i is:

$$\mathrm{bg_{vis}} \mathrel{+}= T_i \frac{\sigma_{h_i}}{\sigma_{t_i}} \left(1 - e^{-\sigma_{t_i}\Delta_i}\right)$$

Since I mentioned holdouts, I just want to explain how this is tracked, since its an important production feature.

First, we keep track of the holdout amount using the same units as extinction - inverse length. That's because we need overlapping volumes to compose correctly, even if only of them is flagged to be held out.

At every step, we compute the holdout amount with the formula you see here.

This computes what we call "background visibility". This makes all the math additive which simplifies the formulas. Alpha is computed at the end as just one minus that value. This representation is also convenient in the surface case. Anywhere we would like alpha to be 0 we just accumulate background visibility of 1 based on the current ray weight.

Decoupled Ray Marching

From this discretized representation of volumes (and surfaces) along the ray, we now need to decide:

- Where to sample direct lighting?
- Where to sample indirect lighting?

So now that we have all this information about what happening along the ray, we just need to decide where to sample direct lighting and how to extend the path for indirect lighting.

Importance Sampling for Single Scattering

Which brings me to the last phase - importance sampling.

I’m going to discuss direct lighting or single scattering first.

$$L(\mathsf{x}, \vec{\mathsf{w}}) = \int_a^b e^{-\int_0^t \sigma_t(\mathsf{x}_s)\,\mathrm{d}s} \left(L_e\left(\mathsf{x}_t, \vec{\mathsf{w}}\right) + \sigma_s\left(\mathsf{x}_t\right) \int_{S^2} \rho(\vec{\mathsf{w}}, \vec{\mathsf{v}}) L(\mathsf{x}_t, \vec{\mathsf{v}})\,\mathrm{d}\vec{\mathsf{v}} \right) \mathrm{d}t$$

Homogeneous media ▼

$$L(\mathsf{x}, \vec{\mathsf{w}}) = \int_a^b e^{-\sigma_t t} \left(\sigma_s \int_{S^2} \rho(\vec{\mathsf{w}}, \vec{\mathsf{v}}) L(\mathsf{x}_t, \vec{\mathsf{v}})\,\mathrm{d}\vec{\mathsf{v}} \right) \mathrm{d}t$$

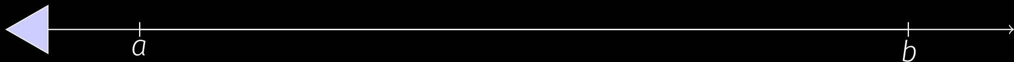
Point light ▼

$$L(\mathsf{x}, \vec{\mathsf{w}}) = \sigma_s \int_a^b e^{-\sigma_t t} \rho(\vec{\mathsf{w}}, \overrightarrow{\mathsf{x}_t \mathsf{c}}) L(\mathsf{x}_t, \overrightarrow{\mathsf{x}_t \mathsf{c}})\,\mathrm{d}t$$

In order to tackle the easiest problem first, we're going to simplify the full volume rendering equation down to just homogeneous mediums and point lights. This allows us to collapse most of the integrals and just focus on one.

Single Scattering Equation for Point Light

Integrate point light contribution along the ray

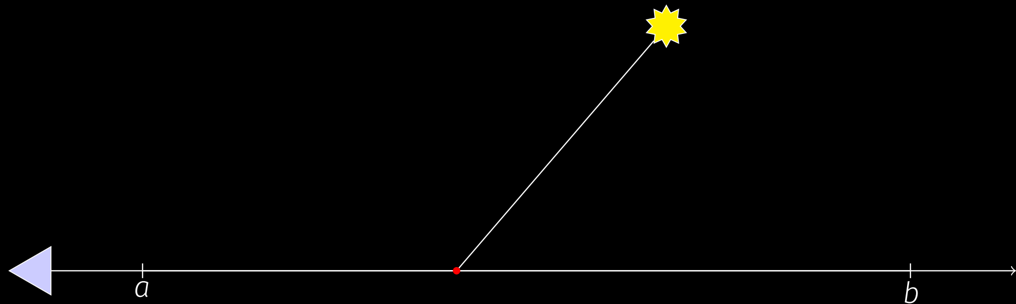


$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b \qquad \mathrm{d}t$$

I am just going to quickly run through the terms in this equation so we see what each one does.

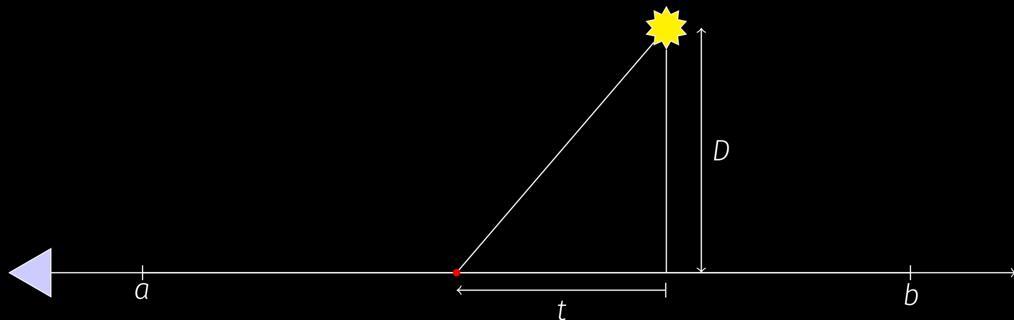
Single Scattering Equation for Point Light

Radiance reaching the ray varies as $1/r^2$

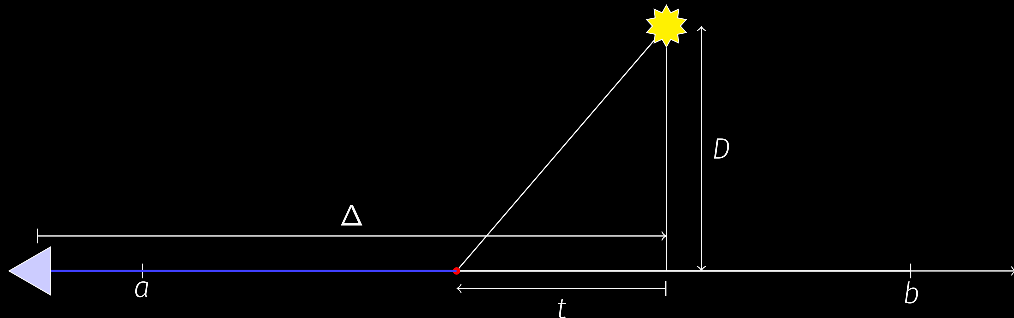


$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b \frac{\Phi}{\|\mathbf{x}_t - \mathbf{c}\|^2} \, \mathrm{d}t$$

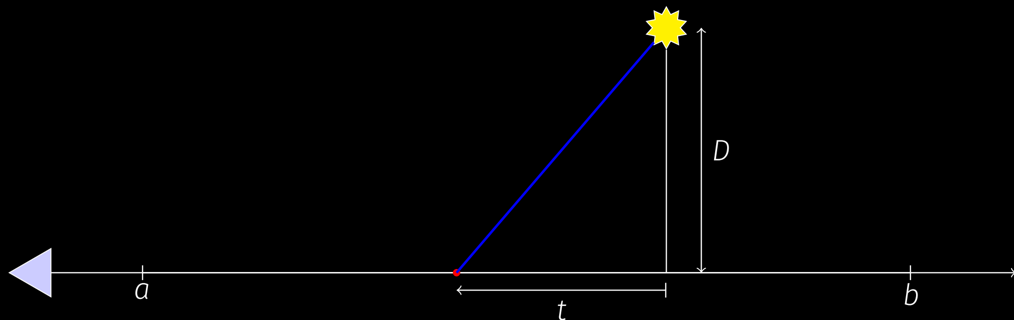
The radiance reaching the ray varies as $1/r^2$.



We can express that in terms of the distance t along the ray and the distance D between the ray and the light.



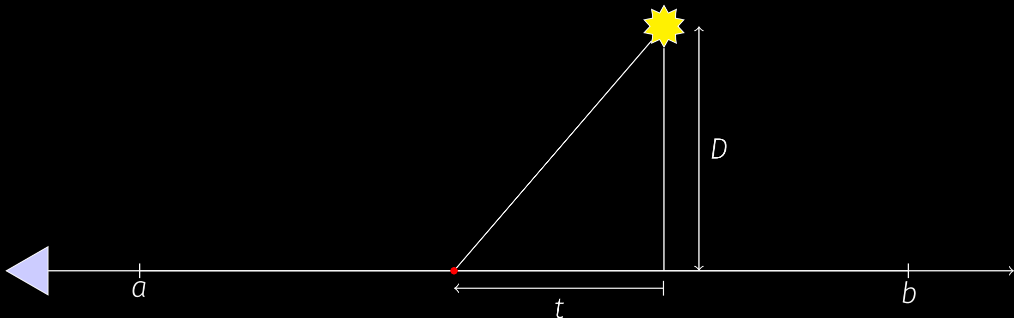
Now we add extinction along the viewing ray. The Δ here is because I've shifted to origin to be under the light....



…and now extinction along the shadow ray

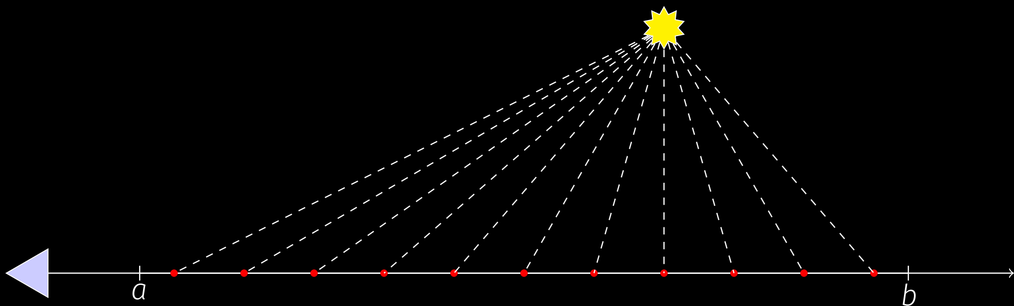
Single Scattering Equation for Point Light

Finally add phase function and visibility



$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t\sqrt{D^2+t^2}} \rho(\vec{\mathbf{w}}, \overrightarrow{\mathbf{x}_t\mathbf{c}}) V(\mathbf{x}_t, \mathbf{c}) \frac{\Phi}{D^2+t^2}\,\mathrm{d}t$$

And then the rest is just the phase function and visibility.



So the question is - how should we distribute samples for Monte Carlo integration?

Importance sampling for point lights

$$L(\mathrm{x}, \vec{\mathrm{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t \sqrt{D^2+t^2}} \rho\left(\vec{\mathrm{w}}, \vec{\mathrm{v}}_t\right) V(\mathrm{c}, \mathrm{x}_t) \frac{\Phi}{D^2+t^2} \mathrm{d}t$$

Only two terms can be easily integrated and inverted

There are really only two terms in this expression that can be importance sampled easily.

Importance sampling for point lights

$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b e^{-\sigma_t (t+\Delta)} e^{-\sigma_t \sqrt{D^2+t^2}} \rho\left(\vec{\mathbf{w}}, \vec{\mathbf{v}}_t\right) V(\mathbf{c}, \mathbf{x}_t) \frac{\Phi}{D^2+t^2} \mathrm{d}t$$

Only two terms can be easily integrated and inverted

Transmission $e^{-\sigma_t t}$

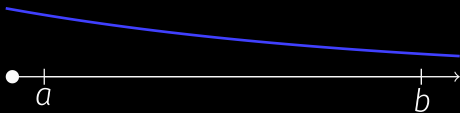
First is the transmission, which was discussed earlier.

Importance sampling for point lights

$$L(\mathrm{x}, \vec{\mathrm{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t\sqrt{D^2+t^2}} \rho\left(\vec{\mathrm{w}}, \vec{\mathrm{v}_t}\right) V\left(\mathrm{c}, \mathrm{x}_t\right) \frac{\Phi}{D^2+t^2}\,\mathrm{d}t$$

Only two terms can be easily integrated and inverted

Transmission $e^{-\sigma_t t}$



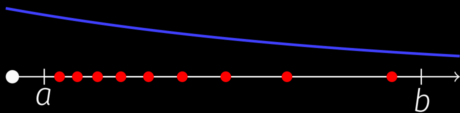
Notice that this function is really slowly varying and always between 0 and 1.

Importance sampling for point lights

$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t\sqrt{D^2+t^2}} \rho\left(\vec{\mathbf{w}}, \vec{\mathbf{v}_t}\right) V(\mathbf{c}, \mathbf{x}_t) \frac{\Phi}{D^2+t^2} \mathrm{d}t$$

Only two terms can be easily integrated and inverted

Transmission $e^{-\sigma_t t}$



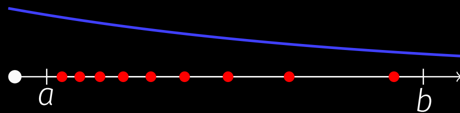
Sampling according to this will tend to place samples close to the ray origin

Importance sampling for point lights

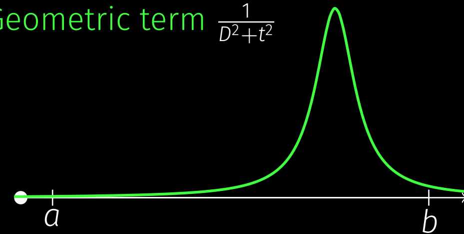
$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t\sqrt{D^2+t^2}} \rho\left(\vec{\mathbf{w}}, \vec{\mathbf{v}_t}\right) V(\mathbf{c}, \mathbf{x}_t) \frac{\Phi}{D^2+t^2}\,\mathrm{d}t$$

Only two terms can be easily integrated and inverted

Transmission $e^{-\sigma_t t}$



Geometric term $\frac{1}{D^2+t^2}$

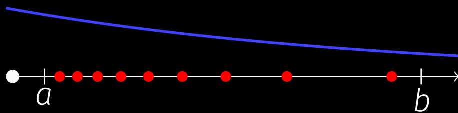


Second is the geometric term.

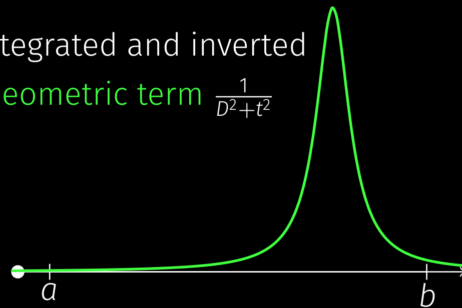
$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t\sqrt{D^2+t^2}} \rho\left(\vec{\mathbf{w}}, \vec{\mathbf{v}_t}\right) V(\mathbf{c}, \mathbf{x}_t) \frac{\Phi}{D^2+t^2} \mathrm{d}t$$

Only two terms can be easily integrated and inverted

Transmission $e^{-\sigma_t t}$



Geometric term $\frac{1}{D^2+t^2}$



This one actually has a large spike that can get arbitrarily large as we get closer to the light source.

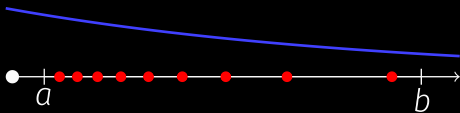
Importance sampling for point lights

$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t\sqrt{D^2+t^2}} \rho\left(\vec{\mathbf{w}}, \vec{\mathbf{v}_t}\right) V(\mathbf{c}, \mathbf{x}_t) \frac{\Phi}{D^2+t^2} \mathrm{d}t$$

Only two terms can be easily integrated and inverted

Transmission $e^{-\sigma_t t}$

Geometric term $\frac{1}{D^2+t^2}$



This one actually has a large spike that can get arbitrarily large as we get closer to the light source.

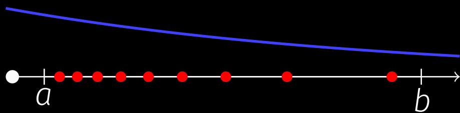
Importance sampling for point lights

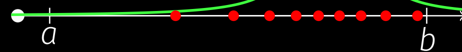
$$L(\mathbf{x}, \vec{\mathbf{w}}) = \sigma_s \int_a^b e^{-\sigma_t(t+\Delta)} e^{-\sigma_t\sqrt{D^2+t^2}} \rho\left(\vec{\mathbf{w}}, \vec{\mathbf{v}_t}\right) V(\mathbf{c}, \mathbf{x}_t) \frac{\Phi}{D^2+t^2} \mathrm{d}t$$

Only two terms can be easily integrated and inverted

Transmission $e^{-\sigma_t t}$

Geometric term $\frac{1}{D^2+t^2}$





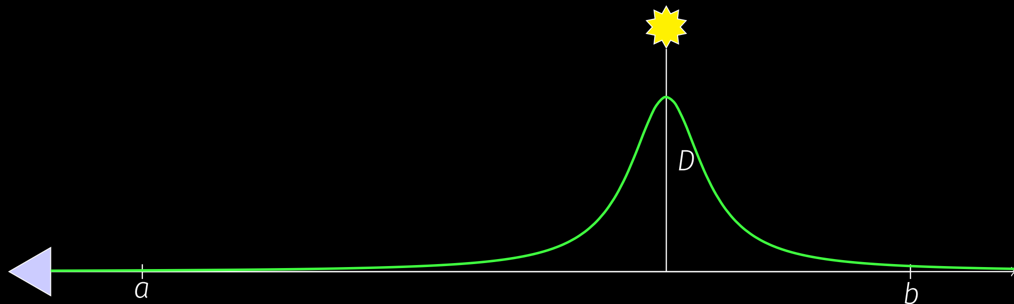
So placing samples here will be much more effective at reducing variance because the overall product is going to be most strongly influenced by this term.

I do need to mention the “Joint Importance Sampling” paper from Siggraph Asia, 2013 that introduced additional estimators that also include the phase function. The only problem is they are a bit hard to implement in our renderer because we allow the phase function to change along the ray. But the paper itself is definitely still worth checking out.

Importance sampling for point lights

Goal is to get a pdf proportional to geometric term:

$$\text{pdf}(t) \propto \frac{1}{D^2+t^2}$$

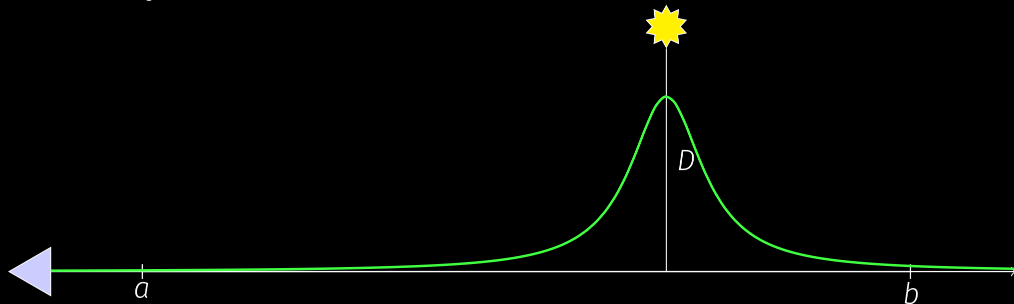


You might have already recognized this PDF as a Cauchy distribution, but I'll work through the derivation just to highlight the geometric interpretation.

Importance sampling for point lights

Integrate pdf to obtain cdf:

$$\mathrm{cdf}(t) = \int \frac{1}{D^2+t^2}\,dt = \frac{1}{D}\tan^{-1}\frac{t}{D}$$

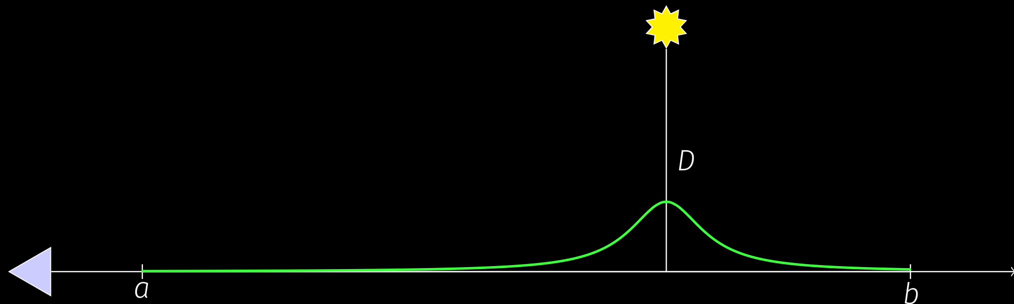


We start by doing the integral which is just the arctangent

Importance sampling for point lights

Use cdf to normalize over $[a, b]$:

$$\mathrm{pdf}(t) = \frac{D}{(\tan^{-1}\frac{b}{D} - \tan^{-1}\frac{a}{D})(D^2+t^2)}$$

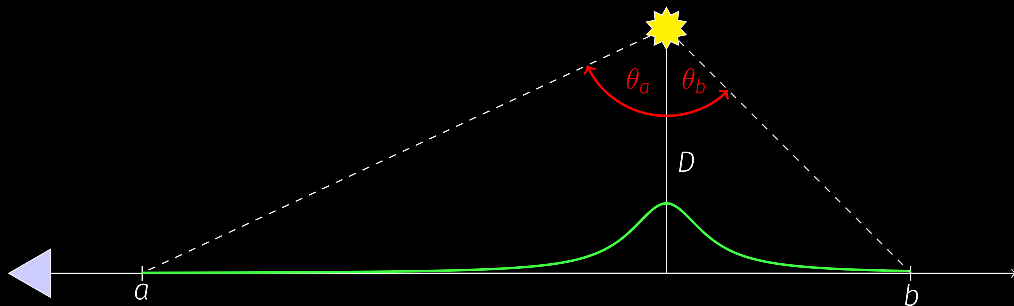


That lets us normalize the expression so it becomes a valid PDF

Importance sampling for point lights

Use cdf to normalize over $[a, b]$:

$$\mathrm{pdf}(t) = \frac{D}{(\theta_b - \theta_a)(D^2 + t^2)}$$

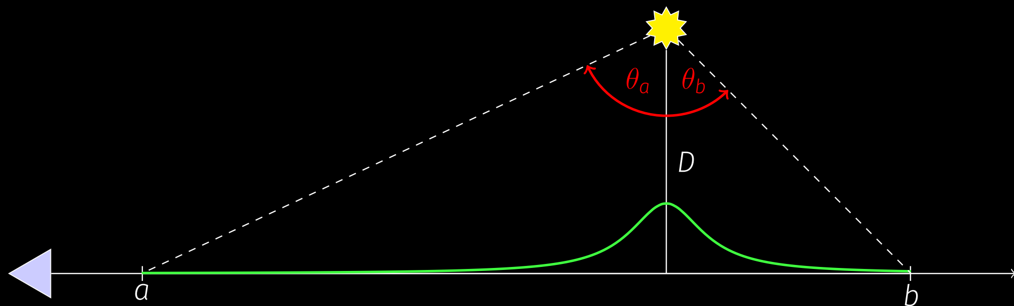


Those arctangents actually have a very precise geometric meaning: they correspond to the angle toward the endpoints of the ray.

Importance sampling for point lights

Invert cdf to obtain distribution for $\xi \in [0, 1)$:

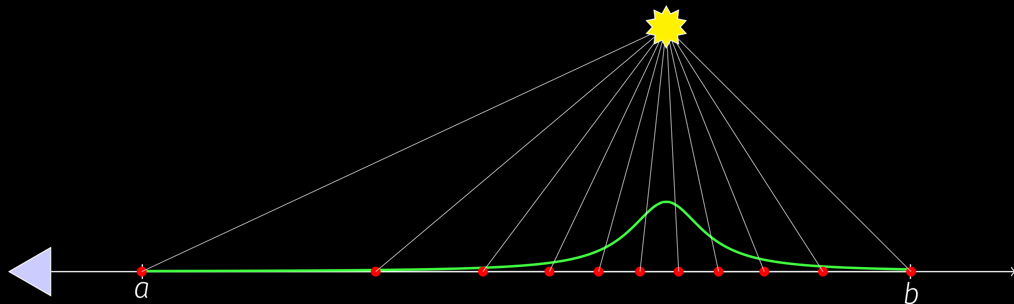
$$t(\xi) = D \tan((1 - \xi)\theta_a + \xi\theta_b)$$



And now we can get the final expression for the inverse CDF which lets us distribute points along the ray. Again looking at the term inside the tangent, you'll notice we're just doing a linear interpolation between those angles.

Sample distribution is *equiangular*

$$t(\xi) = D \tan\left((1-\xi)\,\theta_a + \xi\theta_b\right)$$



This means equal steps in the random variable make steps of equal angles. Thats why we call this distribution *equi-angular*.

Importance sampling for point lights

If light source lies on the ray ($D = 0$) use:

$$\mathrm{pdf}(t) = \frac{ab}{(b-a)t^2}$$
$$\mathrm{t}(\xi) = \frac{ab}{b+(a-b)\xi}$$

Rare corner case but can occur in “flashlight” configuration.

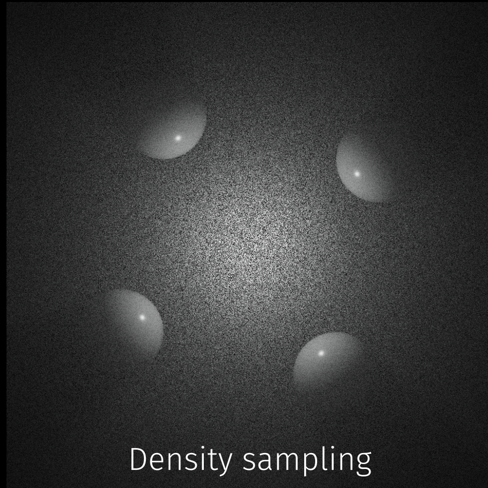
You might have noticed in the formulas I just showed that things can still break down if D is exactly 0.

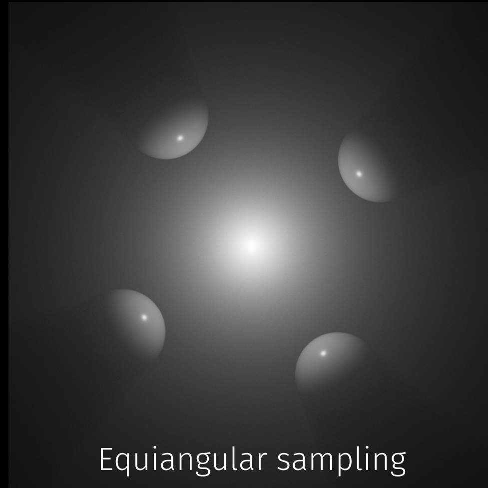
But you can still get a simple closed form formula for this case.

It might sound like something that shouldn’t really matter but this actually came up for us in production when an artist put a spotlight in exactly the same position as the camera. This is sometimes called the “flashlight” configuration.

It only takes 2 or 3 extra lines of code, so might as well add it. It also makes for a good unit test.

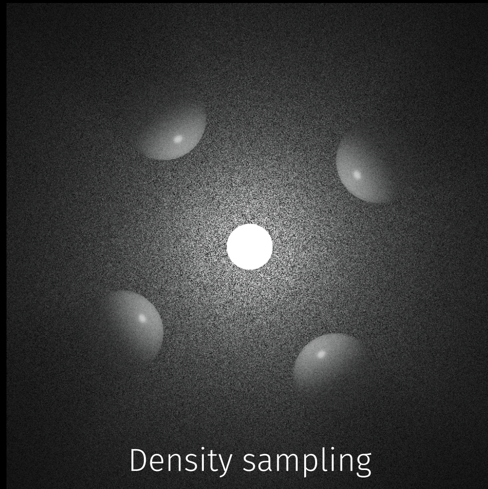
Results with 16 samples/pixel

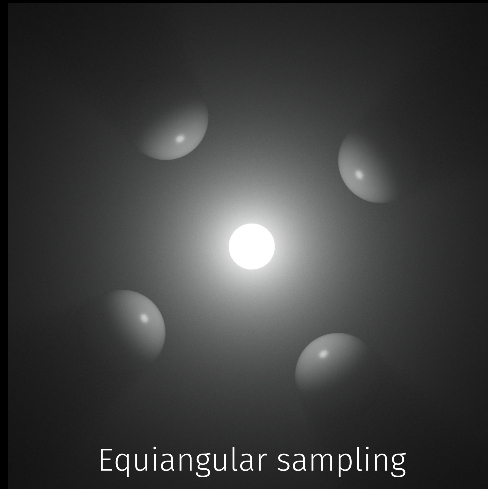

Density sampling


Equiangular sampling

So lets look at some results. As we expected - equi-angular sampling does a much better job at capturing the lighting around the point light.

SPHERICAL LIGHTS CAN USE SAME EQUATIONS! ($\Omega \propto 1/r^2$)





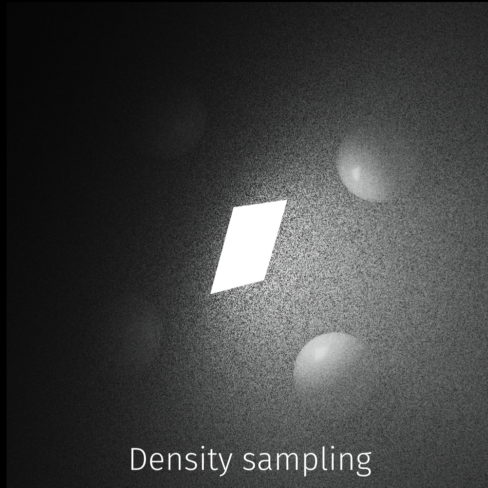
In fact, even though I described everything so far assuming a single point light - it works for area lights too.

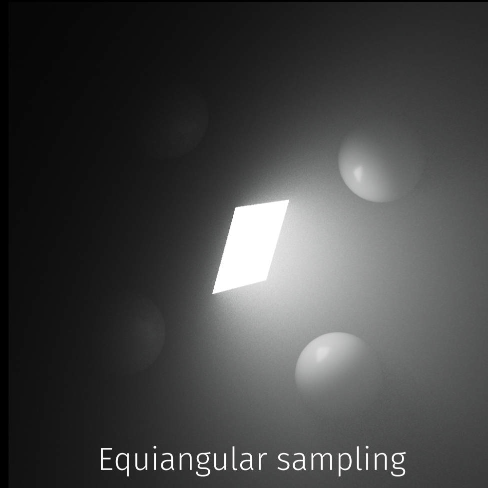
It turns out that the solid angle of area lights also varies as $1/r^2$.

So here I have a spherical light that shows a similar improvement with exactly the same equations.

There are two steps here, first we pick a point along the ray with equi-angular sampling, and then we fire a ray towards the light using the regular solid angle based sampling.

Quad lights can use same equations! ($\Omega \propto 1/r^2$)

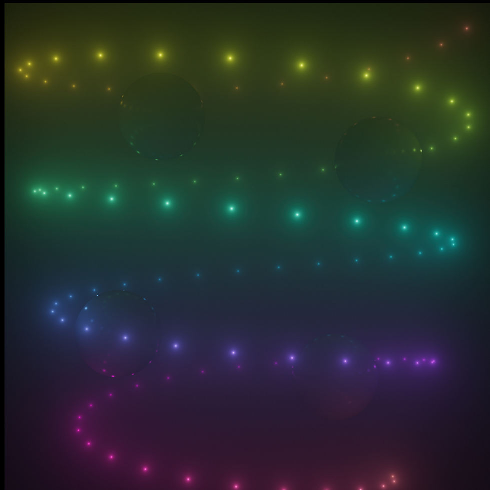




Other shapes like quads work as well.

As long as you can implement solid angle sampling from an arbitrary point to the light - doing equiangular sampling to pick that point will give good results.

Many lights



"Importance Sampling of Many Lights With Adaptive Tree Splitting"

Monday, 3:45PM, Room 402AB

The idea can even be extended to many lights. I won't go into the details of this here. My coworker Alex Conty will be giving a talk on this topic tomorrow.

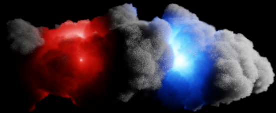
Mesh lights

"Importance Sampling of Many Lights With Adaptive Tree Splitting"

Monday, 3:45PM, Room 402AB

Mesh lights will be covered in that talk as well - we handle those as collections of triangular lights.

Multiple Importance Sampling

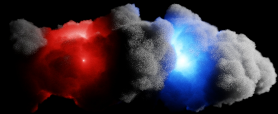


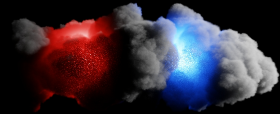
Equi-angular Sampling
$(\mathrm{pdf}(t) \propto 1/r^2)$

I described equiangular sampling inside homogeneous media for simplicity, but remember from the ray marching step that we've actually stored a complete representation of the volume along the ray.

So like I said before, we can use this stored copy of transmittance and scattering coefficients to compute the lighting at any point along the ray.

Equi-angular sampling is still effective here, for example with the point lights inside the cloud.



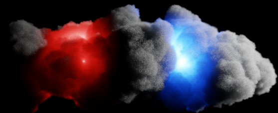


Equi-angular Sampling
$(\mathrm{pdf}(t) \propto 1/r^2)$

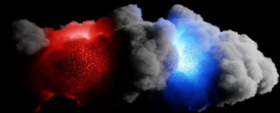
Density Sampling
$(\mathrm{pdf}(t) \propto \sigma_{s_i} T_i)$

But sampling proportionally to transmittance and scattering has some advantages as well. There's a third distant light in this scene that still shows a bit of noise near the "surface" of the cloud and this goes away with density sampling because it focuses more samples there.

Multiple Importance Sampling



Equi-angular Sampling
$(\mathrm{pdf}(t) \propto 1/r^2)$



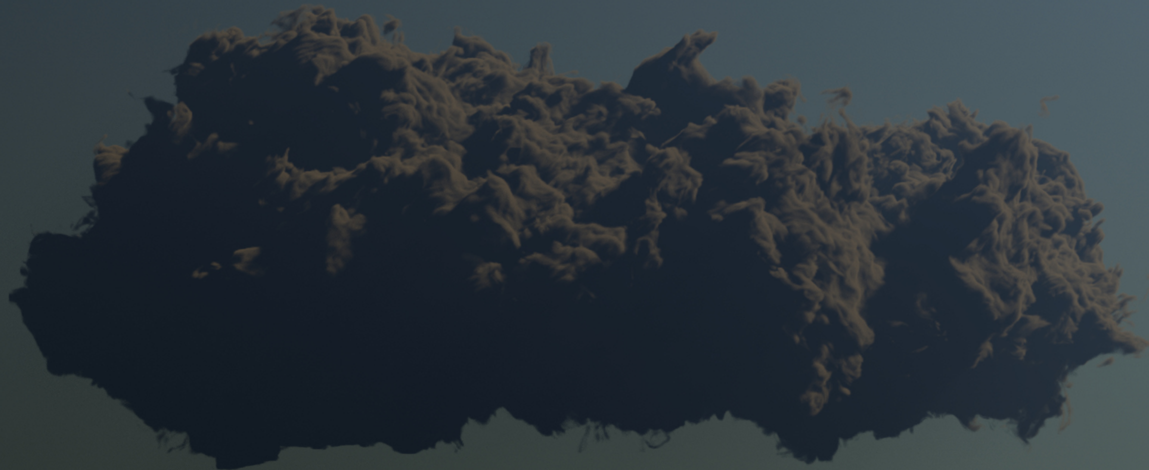
Density Sampling
$(\mathrm{pdf}(t) \propto \sigma_{s_i} T_i)$

MIS
(best of both)

Of course like any time we have two sampling techniques that have orthogonal strengths, combining them by multiple importance sampling lets us get the best of both worlds.

Optimizations for Multiple Scattering

The last thing left to cover is multiple scattering.



For a long time, we just assumed we wouldn't be able to afford this - but it really is very important.

Here I've made a scene with a volumetric cloud surrounded by atmosphere. That atmosphere is assigned to a roughly planet sized sphere, and there is just a single distant light. So every pixel here is lit through a volume.

With only single scattering - the picture doesn't look like a cloud at all. The directly lit parts are gray and the shadowed part is all flat.

Even just adding a single bounce makes a dramatic difference to the color of the sky.

I’ll keep going by doubling the trace depth…

4...

8…

16...

32...

At 64 bounces - we’re still seeing new details come out…

128...

256…

512...

1024…Now at this point things have sort of stabilized. But I'm sure we could see a few more areas brighten up if we kept going.

Of course tracing up this depth was roughly a thousand times more expensive than the first picture, because we have up to a thousand segments per path instead of one.

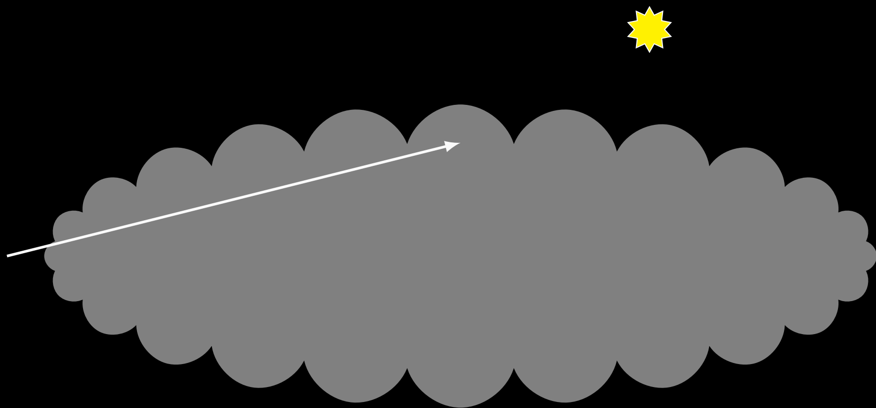
Optimizations for Multiple Scattering

Computing single scattering for every path segment can be wasteful, particularly in volumes which generate many short bounces.

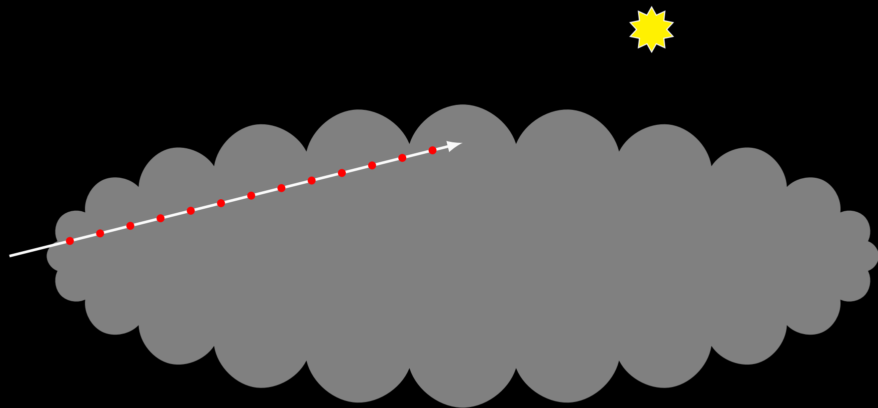
We can defer single scattering until the whole path has been generated, and stochastically perform single scattering on a subset of these segments.

The only optimization we really do to help is to recognize that doing next-event estimation for all segments or vertices in a long path can actually be wasteful. When the medium is dense, we can end up with lots series of segments that make similar contributions.

So we actually wait until we have an entire path generated before deciding which segments need to perform single scattering.



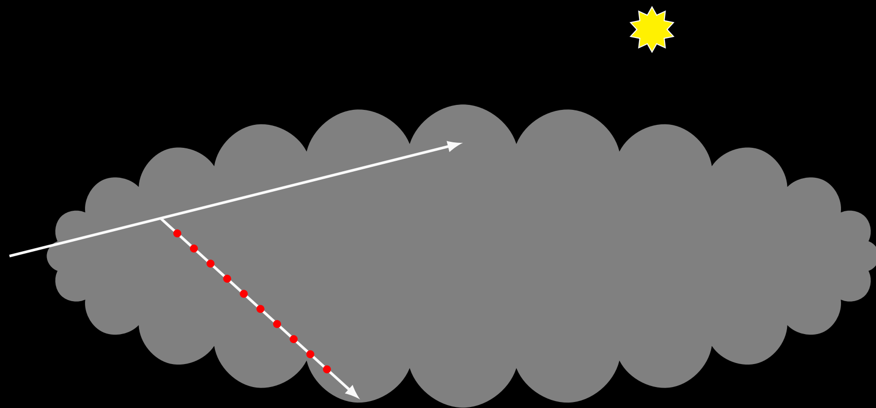
Let me illustrate this with a diagram. The ray enters the cloud, and I'm going to show how we build up the path.



Each ray does some ray marching until it leaves the cloud or reaches some opacity threshold.

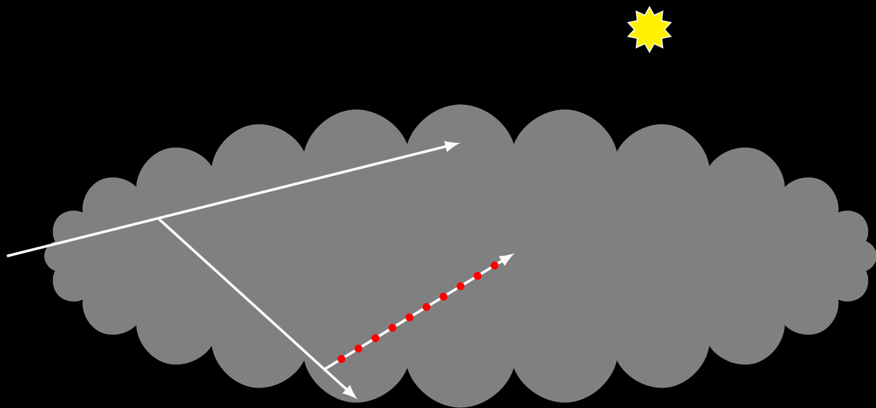
Multiple Scattering

Next ray chosen by discrete pdf $\propto \sigma_{s_i} T_i$



We choose the next ray to trace based on a discrete pdf proportional to scattering times transmission.

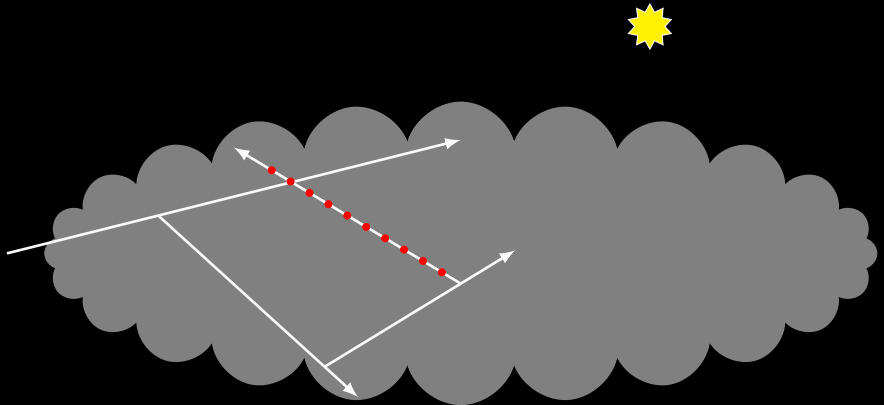
Russian roulette can terminate paths early



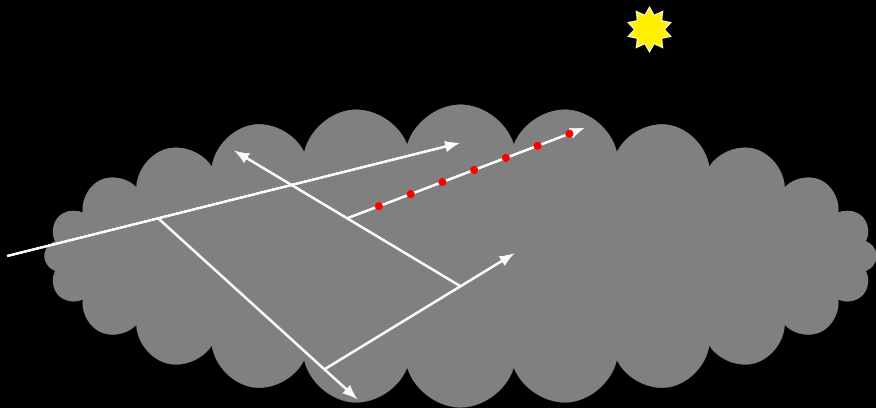
Now we can apply Russian roulette to terminate any paths that don't carry much energy

Multiple Scattering

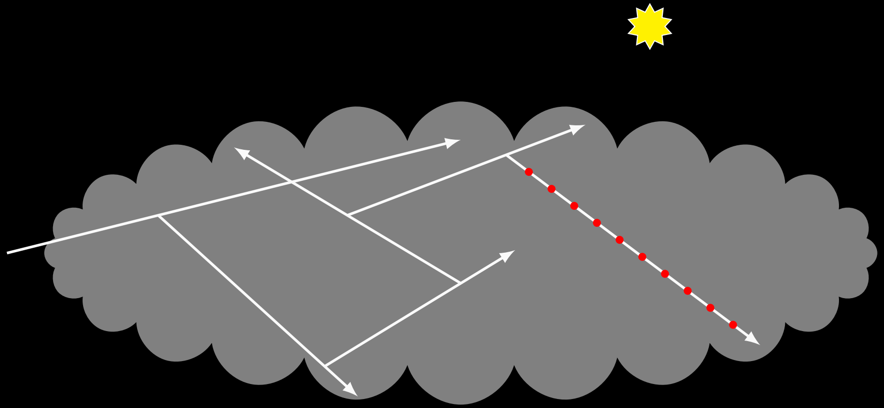
RR equalizes the weight of surviving paths



This has the effect of roughly equalizing the weight of paths that do survive



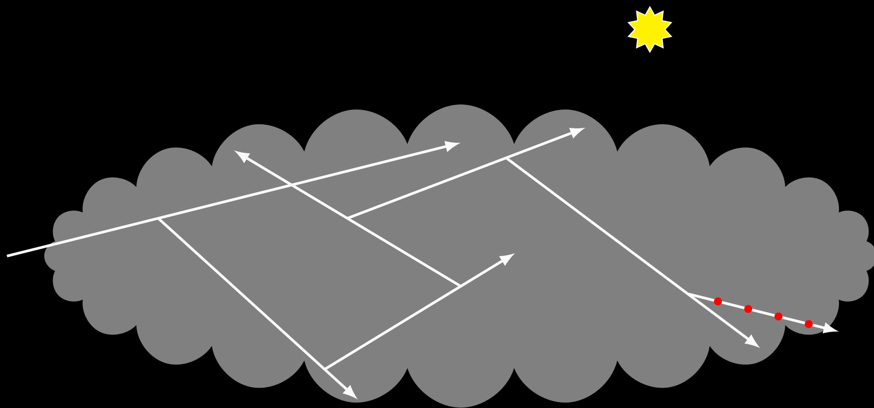
But Russian roulette doesn't help at all when albedo is close to 1. And this is precisely the case where multiple scattering is really important - like clouds.



...

Multiple Scattering

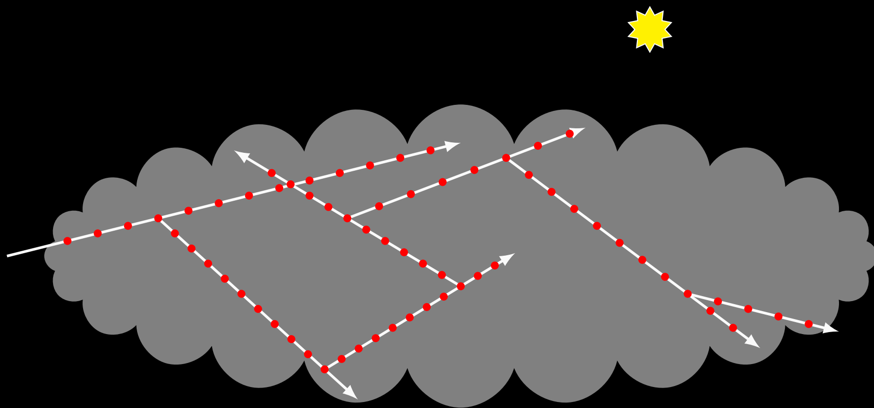
When path is complete, we have many segments!



So when our path is complete - we have this whole collection of segments...

Multiple Scattering

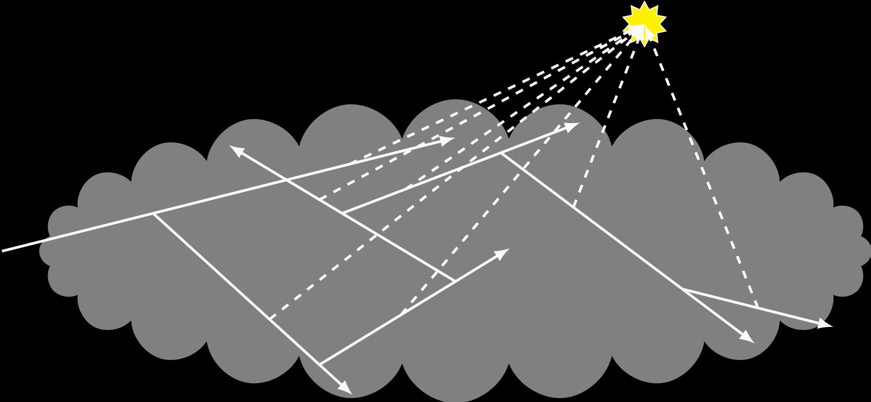
Each has a full representation of volume along it



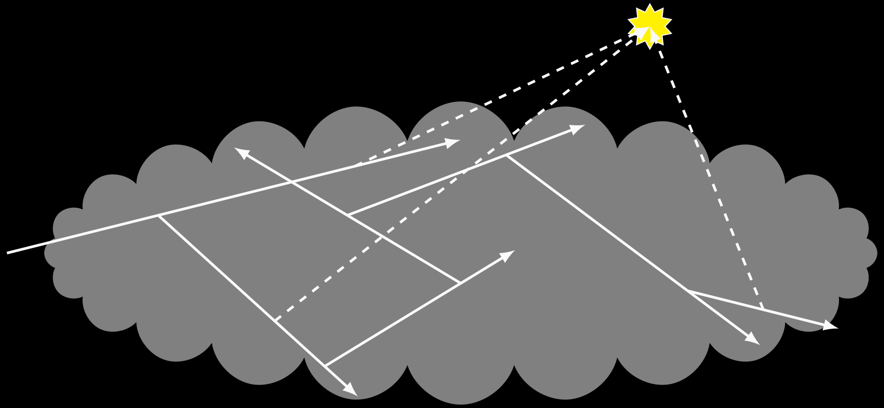
…and each one stores a full description of the volume along it.

Multiple Scattering

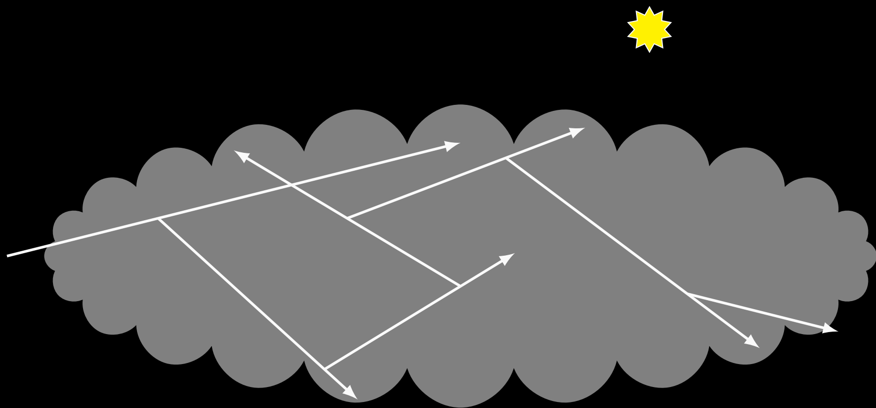
We could connect each to the light …



We could just connect each one to the light…



…or just some random subset of them



This is the classic trade-off between efficiency and variance.

In our tests, we've found this paid off. It sped up renders without adding too much noise. But I don't want to oversell this as a solution either. The frames of the cloud at really high depth that I showed a minute ago are still very expensive.

The Spectral Decomposition Tracking paper being presented this year at Siggraph looks like a much more promising technique and its something we would like to investigate.

Conclusion

So let me just wrap up now with a few more observations…

Hindsights

- Stack based medium tracking is complex, but necessary

Stack based medium tracking is tricky to implement and carrying extra state on each ray can be a roadblock to efficiently vectorizing the renderer.

At the same time, we haven't found any better way to correctly render liquids and glass. And now that we have this system in place, productions are leveraging it more and more.

Hindsights

- Stack based medium tracking is complex, but necessary
- Volumes and SSS share the same code, only tracking method changes

Because we have these two ways of tracking volumes, we've been able to unify subsurface scattering and volumes. They no longer are really different things in our system, which has really helped make the system more predictable. Artists are free to put volumes inside refractive objects and can expect the same results as if they had used subsurface scattering defined through the surface shader.

We'll be discussing this a bit more in the physically based shading course this afternoon.

Hindsights

- Stack based medium tracking is complex, but necessary
- Volumes and SSS share the same code, only tracking method changes
- Decoupled ray marching provides good pdfs, but requires many lookups

Next - decoupled ray marching provides really good pdfs - but it does requires lots of lookups which makes it expensive over many bounces.

Hindsights

- Stack based medium tracking is complex, but necessary
- Volumes and SSS share the same code, only tracking method changes
- Decoupled ray marching provides good pdfs, but requires many lookups
- Equiangular sampling essential when lights are inside volumes

Equiangular sampling is really essential when lights are inside volumes. They're a much stronger source of variance so you definitely need a dedicated sampling technique for this case.

- Stack based medium tracking is complex, but necessary
- Volumes and SSS share the same code, only tracking method changes
- Decoupled ray marching provides good pdfs, but requires many lookups
- Equiangular sampling essential when lights are inside volumes
- Multiple scattering still slow when albedo is high

And lastly as I just mentioned – multiple scattering is still slow when the albedo is really high.

I mentioned one way to help reduce the cost a little by firing fewer shadow rays, but we see this as just a short term solution. We really just need better algorithms to handle this case well. Again the spectral decomposition tracking paper from this year looks very promising.

Acknowledgements

Rendering Team:

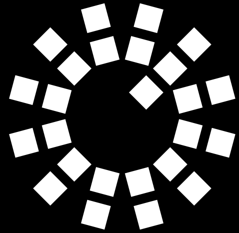
- Alex Conty
- Cliff Stein
- Larry Gritz

Alumni:

- Magnus Wrenninge

I'd like to acknowledge my co-workers on the rendering team at Imageworks, in particular Alex Conty who actually implemented the messy details of medium stacks and the multiple scattering optimizations.

I also want to acknowledge Magnus who participated in the very early design of volume rendering support in our renderer and did a lot of the early testing of it.



Thank You! Questions?

To hear more about our renderer please see:

- Course: “Physically Based Shading In Theory and Practice”, Sunday 2:00PM, Room 150/151
- Talk: “Importance Sampling of Many Lights With Adaptive Tree Splitting” - Monday 3:45PM, Room 402AB
- Course: “Path Tracing in Production - Part 2: Making Movies”, Wednesday 2:00PM, Room 408AB

With that I would like to conclude my talk.

Here are the references to the other courses I’ll be participating in, both this afternoon and on Wednesday.

Thank you for listening and I’m happy to take any questions.