Real-Time Samurai Cinema

Lighting, Atmosphere, and Tonemapping in

Jasmin Patry
Sucker Punch Productions

Speaker notes

Hi! My name is Jasmin Patry, and I work at Sucker Punch Productions as a lead rendering engineer. I'm honored and excited to be talking to you about some of the techniques that we used to create a real-time "samurai cinema" experience in Ghost of Tsushima.

Introduction

GHOST

- Sucker Punch is a part of PlayStation Studios (formerly Sony Interactive Entertainment Worldwide Studios)
- Peak size: 160 people (including 25 QA)
- Previous games include:
  - *Sly Cooper* (*1*, *2*, & *3*)
  - *Infamous* (*1*, *2*, *Festival of Blood*, *Second Son*, *First Light*)

Speaker notes

If you're not familiar with Sucker Punch, we've been a part of Sony Interactive Entertainment since 2011, and are located in Bellevue, Washington, near Seattle.
[next]
During the development of Ghost, we reached a peak of 160 people working in the studio,
[next]
and our previous games include
[next]
the Sly Cooper series,
[next]
and the Infamous series of games.

Introduction

- *Ghost of Tsushima* released in July 2020 for the PlayStation 4
- *Legends* multiplayer expansion released in October 2020
  - Also added 60 FPS support for PS5

Speaker notes

Last summer, we released Ghost of Tsushima for the PlayStation 4.
[next]
A free co-op multiplayer expansion followed in the fall,
[next]
along with 60 FPS support when playing on the PS5 in backwards-compatibility mode.

Speaker notes

Ghost is an open-world action adventure set in 13th-century feudal Japan...

Speaker notes

... in which you take on the role of samurai Jin Sakai...

Speaker notes

... the lone survivor of the Mongol invasion of the island of Tsushima...

Speaker notes

... as he fights to liberate his home.

Introduction | Other Talks

Samurai Shading in Ghost of Tsushima
Jasmin Patry

Procedural Grass in Ghost of Tsushima
Eric Wohllaib

Samurai Landscapes: Building and Rendering Tsushima Island on PS4
Matthew Pohlmann

Blowing from the West: Simulating Wind in Ghost of Tsushima
Bill Rockenbeck

Zen of Streaming: Building and Loading Ghost of Tsushima
Adrian Bentley

Speaker notes

Before launching into my material, I wanted to plug some of the other talks that may be relevant to some of you.
[next]
At last year's SIGGRAPH I gave a talk as part of the Physically Based Shading course regarding some of the shading research that we did for Ghost.
[next]
There were several talks at GDC this year; Eric Wohllaib gave a talk about procedural grass;
[next]
Matthew Pohlmann gave a talk discussing world building and rendering;
[next]
Bill Rockenbeck talked about wind and cloth;
[next]
and Adrian Bentley discussed how we got the game to load so quickly, among other things.

- Transport players to 1274 Japan
  - Huge open world
    - Beautiful, living, breathing
  - Continuous time of day with dynamic weather
- Recreate the style of classic samurai cinema
  - “Kurosawa Mode”
  - Dramatic lighting, wind, clouds, haze, and fog

Speaker notes

Our goals with Ghost included what we called the "time-machine": we wanted to immerse players in our version of Tsushima from 1274.
[next]
This meant creating a large, dynamic open world -- much, much larger than our previous games --
[next]
that was breathtakingly beautiful and felt alive.
[next]
It also meant creating a dynamic weather and continuous time of day system, which we hadn't done before.
[next]
We also wanted to recreate the style of classic samurai cinema --
[next]
Ghost even has a "Kurosawa mode" in honor of the legendary filmmaker --
[next]
which meant that we needed dramatic lighting, biting wind, and expressive skies and atmospherics.

Introduction | Lighting

GHOST

- Art direction called for “stylized realism”
- Lighting models are physically based
- Materials authored with physically plausible values, some photogrammetry
- Indirect lighting computed at runtime from dynamic sky and local light transfer data
- Physically based sky model, used for sky, clouds, haze, and fog particles
- Rendered in HDR and tonemapped with custom techniques
- Lighting can deviate from physical correctness
  - Artists can globally adjust to achieve desired look

Speaker notes

Now I'll talk a little bit about the lighting in Ghost; our art direction called for "stylized realism" -- so we weren't aiming for photorealism on this project.
[next]
Our lighting models are physically based, however; they are energy conserving, use the GGX specular NDF with optional anisotropy; they use the Smith visibility function and Schlick Fresnel approximation. Diffuse lighting is Lambertian with optional energy-conserving translucency and fuzziness models. Lights are authored using physical units, and we support energy conserving area lights.
[next]
Our materials are authored using physically plausible parameters, with some photogrammetry. This allowed our materials to respond to light in a consistent way.
[next]
Our indirect lighting consists of dynamically updated spherical harmonics and reflection probes, based on the current sky and directional light, plus local light transfer data.
[next]
Speaking of skies, we use a physically based multi-scattering atmospheric scattering model for all of our atmospheric lighting, including clouds, haze, and particles.
[next]
Our scenes are rendered in HDR and tonemapped using a set of custom techniques that were necessary to achieve our desired look.
[next]
Since our goal was not photorealism, we do allow artists to deviate from physical correctness at the lighting stage, using a set of knobs that allow them to globally adjust various lighting parameters (such as ambient diffuse and specular balance, sky brightness, etc.). This is typically done at the granularity of the weather and time of day states in our weather system.
[next]
In this talk I'll elaborate on the three topics highlighted here.

Introduction | Overview

GHOST

- Indirect lighting
  - Diffuse
  - Specular
- Atmospheric volumetric lighting
  - Skies & clouds
  - Haze
  - Particles
- Tonemapping
  - Local tonemapping operator
  - Custom tonemapping color space and white balance
  - Purkinje shift (low-light vision simulation)

Speaker notes

This talk will be divided into three sections:
[next]
In the first, I'll discuss our indirect lighting techniques,
[next]
first with regards to diffuse lighting,
[next]
following by indirect specular lighting.
[next]
I'll then discuss our approach to atmospheric volumetric lighting,
[next]
which was used to light our skies, clouds,
[next]
volumetric haze,
[next]
and particles.
[next]
Finally I'll present our tonemapping techniques,
[next]
including our local tonemapping operator,
[next]
our color grading approach using white balance and a custom tonemapping color space,
[next]
as well as our simulation of the low-light behavior of the human visual system to help render convincing night time scenes.

Speaker notes

Now, let's talk about indirect lighting.

Indirect Lighting | Diffuse

- *Infamous: Second Son* used static degree 2 SH irradiance probes in tetrahedral meshes
  - See Adrian Bentley's GDC 2014 Talk, “Engine Postmortem of inFAMOUS: Second Son”

Speaker notes

We'll start with diffuse lighting. In Infamous: Second Son, we used quadratic irradiance probes, arranged in tetrahedral meshes, to provide the indirect diffuse lighting for the entire world.
[next]
You can see Adrian Bentley's GDC 2014 talk for more details.

Speaker notes

This worked quite well for our requirements, that is, static time of day and weather, with changes only happening behind loads.

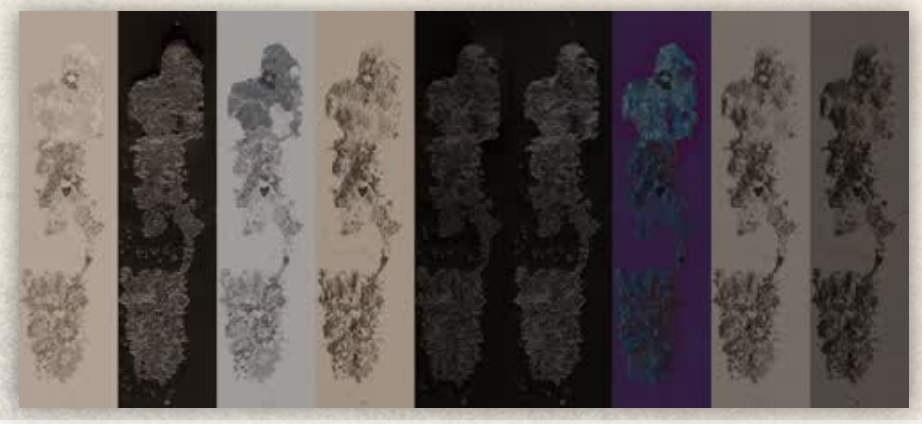
Speaker notes

Here is the same scene without local SH probes.

Indirect Lighting | Diffuse

GHOST

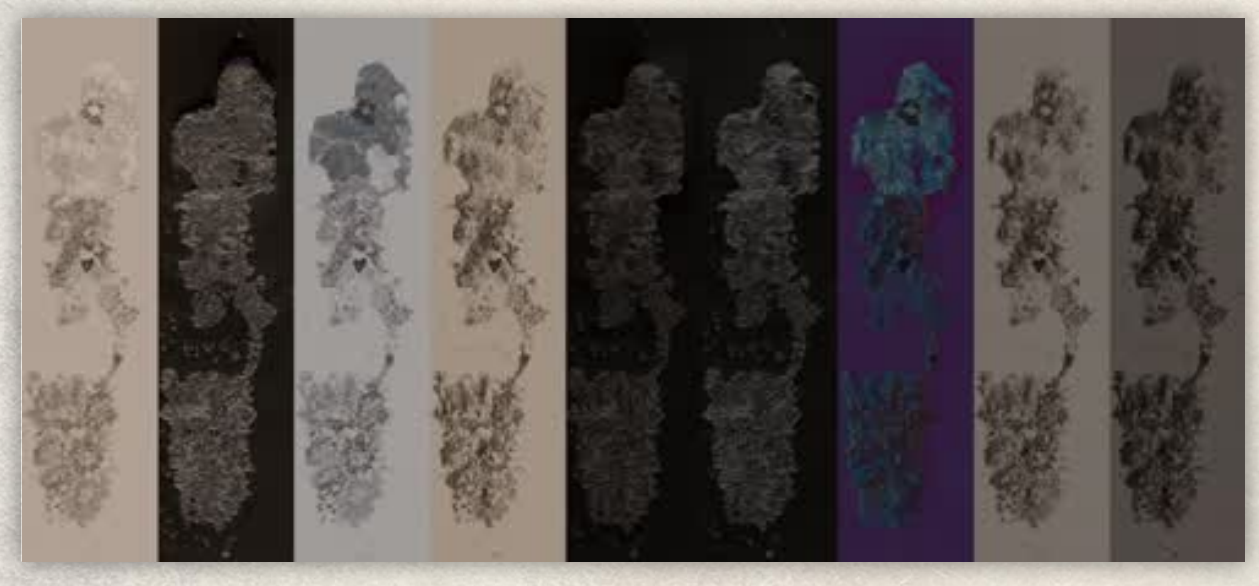
- Tsushima's size and dynamic time of day and weather required a new approach
- Regular grid of quadratic SH probes
  - 16x16x3 per 200m square tile, 20x80 tiles



Speaker notes

In Ghost, we had to light a much larger world, and we had to deal with dynamic time of day and weather for the first time, so we had to make some changes to our approach.
[next]
To cover the island, we use a regular grid of probes,
[next]
arranged at 12.5 m intervals along the ground, and at elevations of 1.5, 10, and 30m above ground level.

y_0^0 y_1^{-1} y_1^0 y_1^1 y_2^{-2} y_2^{-1} y_2^0 y_2^1 y_2^2



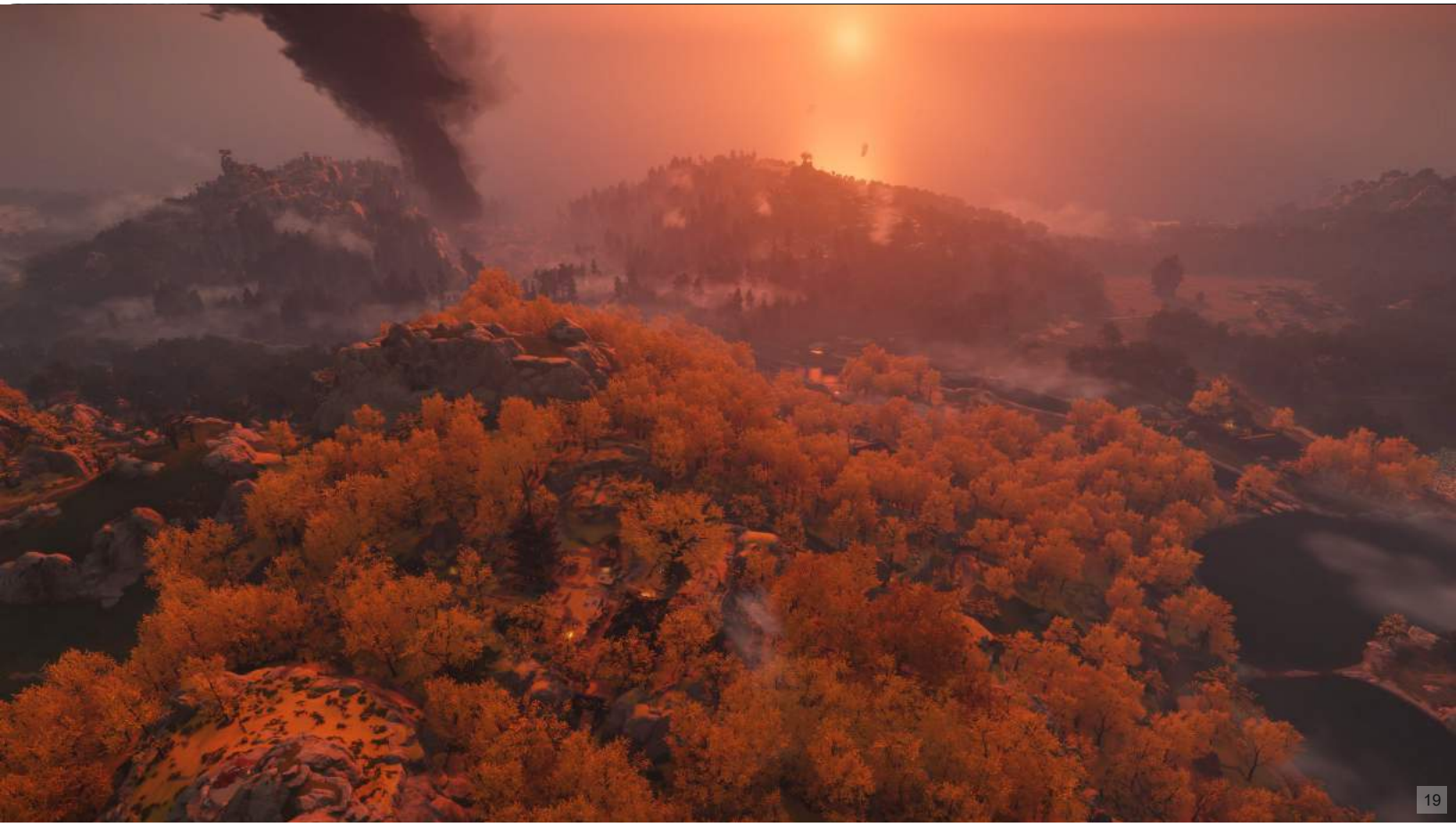
Speaker notes

Here's a 24 hour time of day cycle showing the absolute values of the nine SH bands over the entire island. The quick changes are happening at sunrise and sunset.

- Tetrahedral meshes used in for more complex cases
  - Towns, villages, farmsteads, castles, etc.
  - Streamed in with location
  - Overrides regular grid, blends at interface

Speaker notes

This spacing of probes was too coarse to accurately light interiors, for example, so we used tetrahedral meshes (or "tet meshes")
[next]
in places such as towns, villages, farmsteads, etc.
[next]
These were streamed as separate units, and so the tet meshes were streamed with them.
[next]
When loaded, the tet meshes override the grid, and they blend at the boundary.

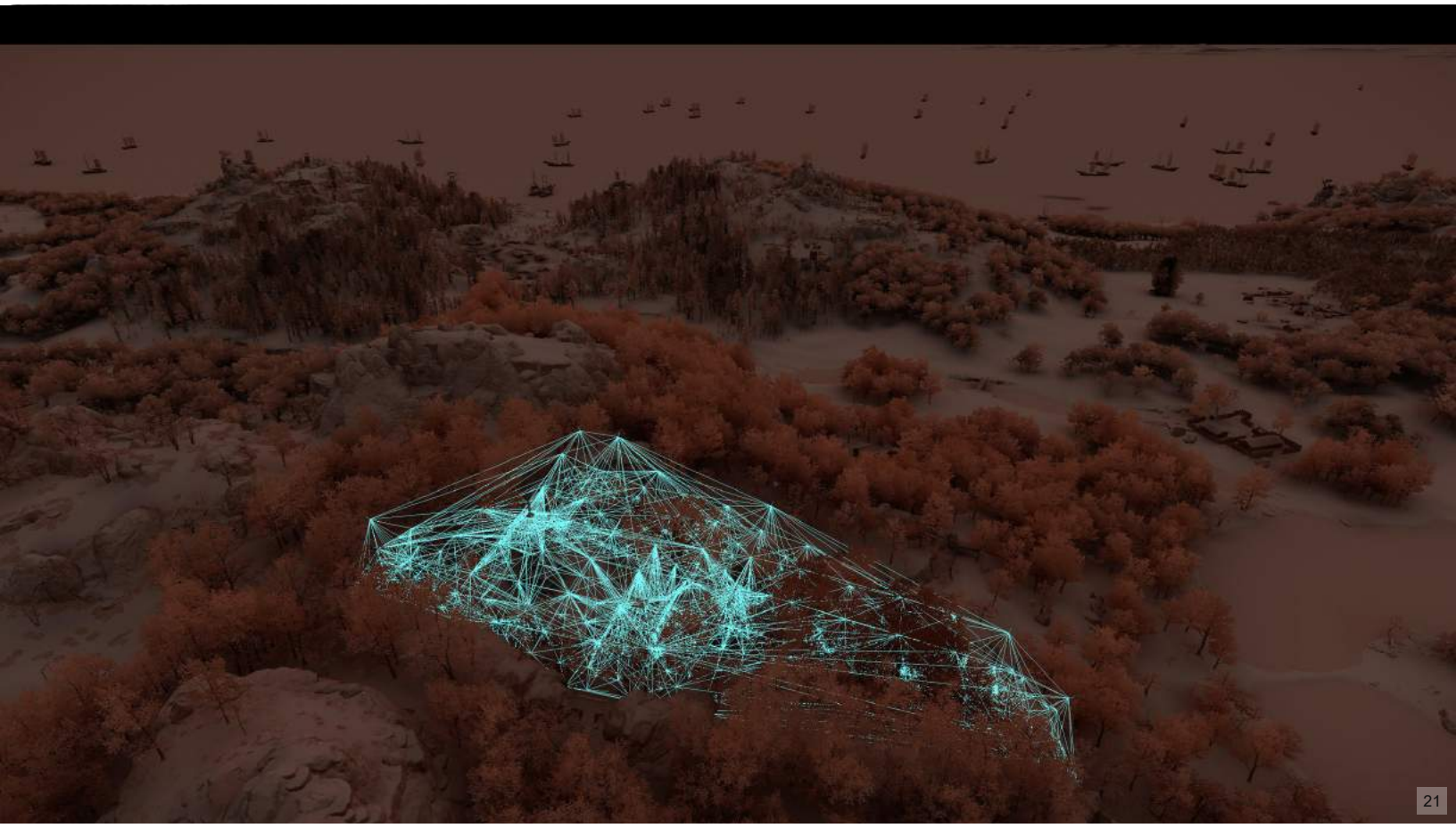


Speaker notes

In the foreground is the Golden Temple location from the game...

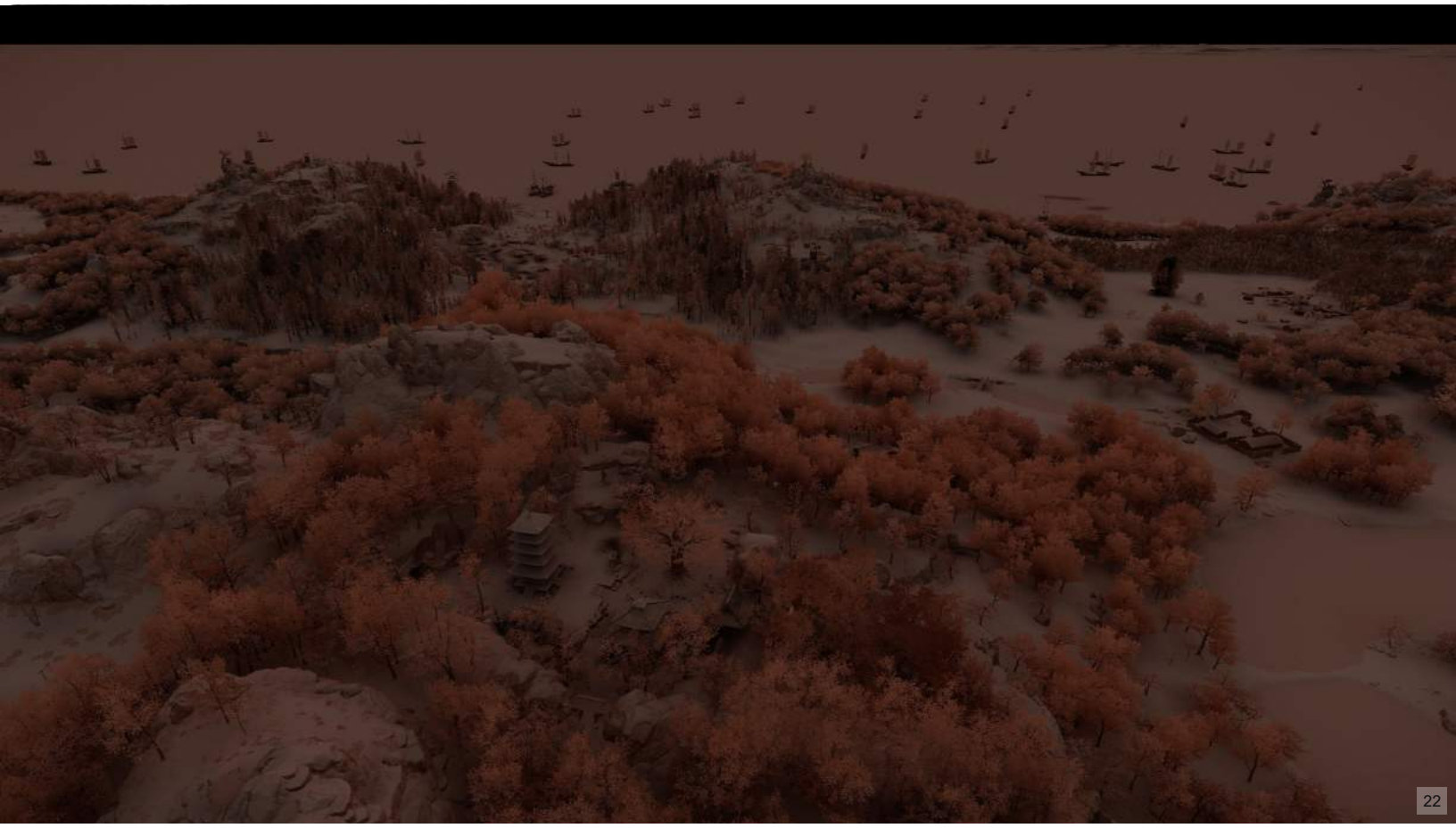
Speaker notes

... and here you can see some of its tet mesh.



Speaker notes

This buffer shows the diffuse indirect lighting...



Speaker notes

... and as you can see, there is a smooth transition between the regular grid lighting and the tet mesh lighting.

Runtime Relighting

- Need to update our irradiance probes at runtime
- Offline, capture:
  - Irradiance

Speaker notes

Because of the game's dynamic time of day cycle and dynamic weather, we need to update our irradiance probes at runtime.
[next]
Offline, instead of capturing
[next]
irradiance...

Runtime Relighting

- Need to update our irradiance probes at runtime
- Offline, capture:
  - ~~Irradiance~~
  - Sky visibility encoded in degree 2 SH (mono)
- At runtime:
  - Project sky to SH
  - Multiply sky SH by sky visibility
  - Convolve with Lambertian cosine lobe

Speaker notes

[next]
... we capture the SH representation of sky visibility,
[next]
At runtime,
[next]
we compute the SH projection of the sky,
[next]
and then update the probes by multiplying the sky luminance by the sky visibility,
[next]
and convolve the result with the cosine lobe as you do with conventional irradiance probes. The result is used just like ordinary irradiance probes for the purposes of lighting.

Bounced Sky Light

- This works, but gives direct sky lighting only
  - No bounced light
- Full transfer matrix?
  - Too much memory (9X)
  - Takes too long to capture
- Approximate by assuming that sky light is ~constant over hemisphere
  - Light the world with a uniform white sky
  - Capture “bounced sky visibility”
- Multiply by average sky color (from 0th SH band)

Speaker notes

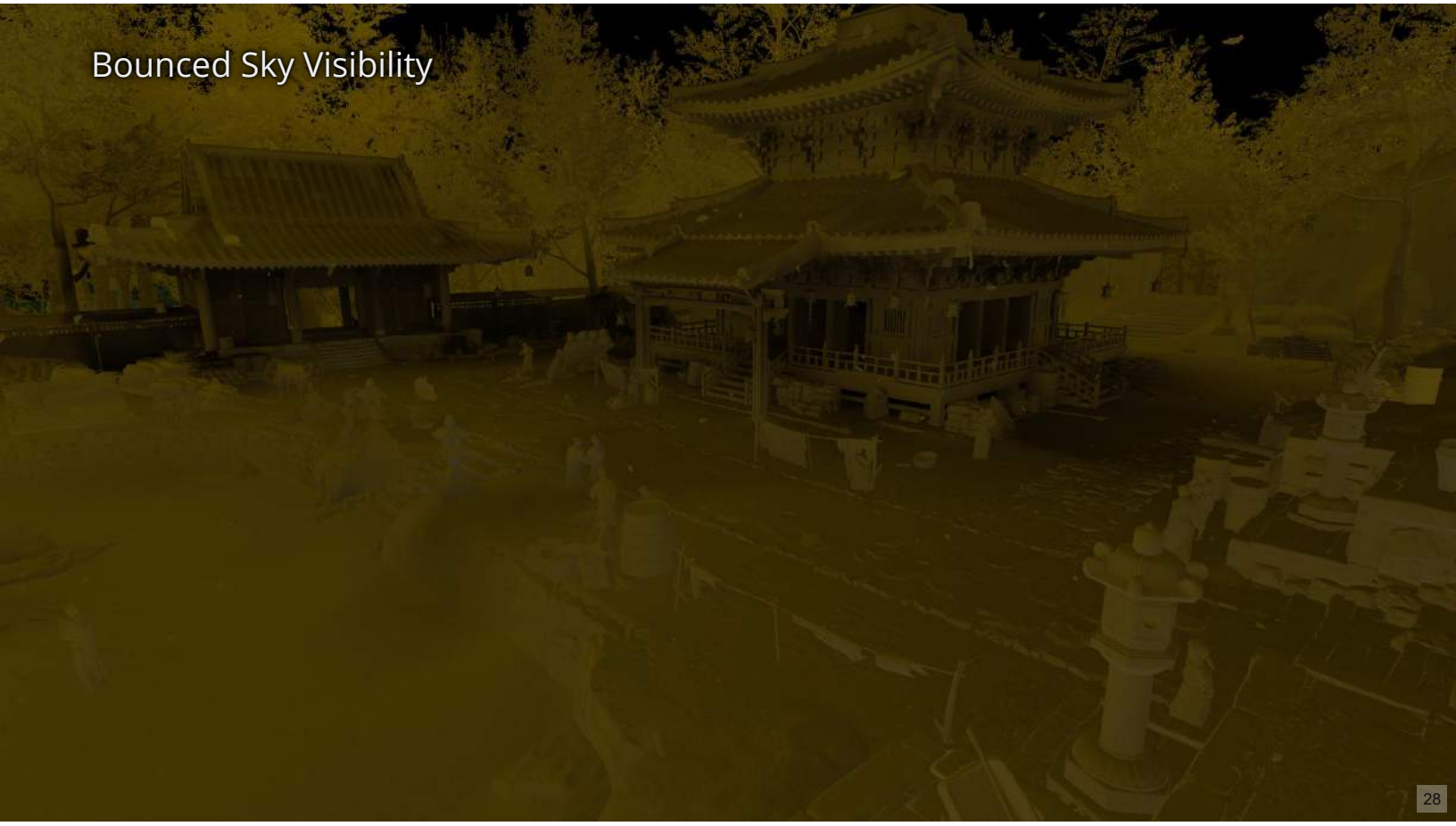
This approach works, but it gives direct sky lighting only --
[next]
that is, there is no bounced light from the environment.
[next]
In Infamous we'd experimented with radiance transfer matrices, which worked for small worlds,
[next]
but since they required at least nine times the storage, they would have taken too much memory,
[next]
and taken far too long to capture.
[next]
So, we opted for an approximation that is valid if the sky light is approximately constant over the hemisphere:
[next]
We capture bounced light by effectively lighting the world with a uniform white sky, using the sky visibility SH functions.
[next]
We then project this radiance to SH to capture bounced sky visibility.
[next]
At runtime, we multiply the bounced sky visibility by the average sky color, and add that to the previous result.

Speaker notes

Ok, let's look at some examples. Here we have a scene that's light with the sky's SH representation only -- that is, no sky visibility information is used.

Speaker notes

This is what the direct sky visibility data looks like...



Speaker notes

... and the bounced sky visibility data looks like this. The yellow colour comes from the bright yellow leaves of the trees surrounding the temple, and the fallen leaves on the ground.

Speaker notes

Here again is the scene without using the sky visibility data...

Speaker notes

... and here is what we get if we use the direct sky visibility.

Speaker notes

Adding the bounced sky visibility adds some yellow bounce light.

Adding Sun/Moon Bounce

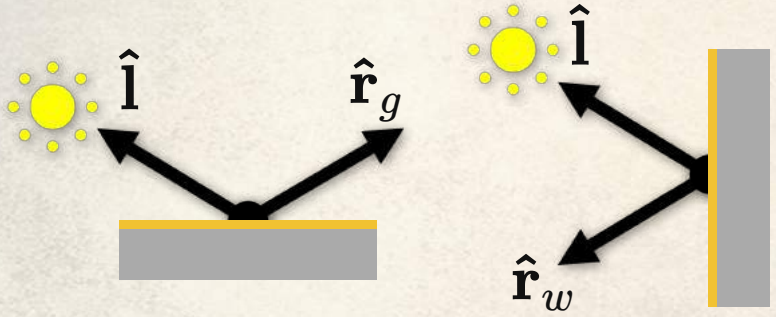
- Bouncing sky light by itself not sufficient
- Can we use the bounced sky visibility data somehow?
- Idea: consider a fixed set of of bounce directions
  - (bounce directions SH) ×
    (bounced sky visibility SH) ×
    (sun/moon irradiance)
  - Assumes that the cosine-weighted average sky visibility of the surfaces is a good estimate of shadowing
    - Close enough for our purposes

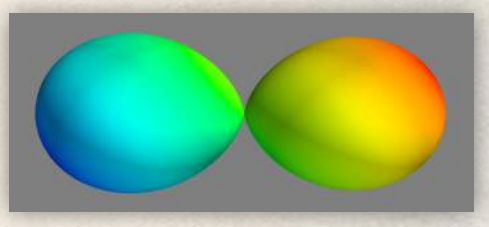
Speaker notes

However, bouncing sky lighting by itself is far from sufficient. In clear weather like in the previous example, most of the bounce light comes from the sun, so we needed a way to add this to our model.
[next]
We wanted to avoid having to store additional data, so we asked ourselves if we could make use of the bounced sky visibility data.
[next]
Our idea was to consider a small fixed set of bounce directions; this gave better result than assuming that bounce light was uniformly distributed over the sphere.
[next]
By multiplying the SH representation of these bounce directions times the bounced sky visibility SH, and then by sun or moon irradiance, we get an SH representation of the bounced light.
[next]
This makes the rather large assumption that the cosine-weighted average sky visibility of the surfaces captured by the bounced sky visibility is a good estimate of their shadowing.
[next]
There are many instances where this isn't the case, but it proved to be a good enough approximation for our purposes.

Adding Sun/Moon Bounce

- Reflect light in imaginary ground and wall





- Project $-\hat{\mathbf{r}}_g$ and $-\hat{\mathbf{r}}_w$ to SH; window to avoid negative lobes
- Scale by RGB light intensity (modulated by dynamic cloud shadow)
- Multiply by bounced sky SH and add to previous result

Speaker notes

Ok, so what bounce directions did we use? We observed that in practice most bounce light came from either horizontal or vertical surfaces, so we used imaginary
[next]
ground planes
[next]
and wall planes. We reflected the light in these planes; it turns out that these vectors r_g and r_w point in exactly opposite directions.
[next]
We project the (negated) reflected light directions into SH;
[next]
here's what that looks like (after deringing to make sure they're positive everywhere).
[next]
Then, we scale it by the light color (after applying cloud shadowing, which is dynamic so isn't captured by the sky visibility),
[next]
and multiply this by the bounced sky SH representation, and add it to the previous result.

Speaker notes

Here is a scene with no sun bounce...

Speaker notes

and here is the same scene with sun bounce added. As you can see, it makes quite a dramatic difference.

Speaker notes

Here's another example, this time inside the temple. Here we're not using sky visibility, so the lighting is very flat.

Sky Visibility

Speaker notes

Here's the direct sky visibility...

Bounced Sky Visibility

Speaker notes

and the bounced sky visibility.

Speaker notes

Here again is the temple without using the sky visibility data...

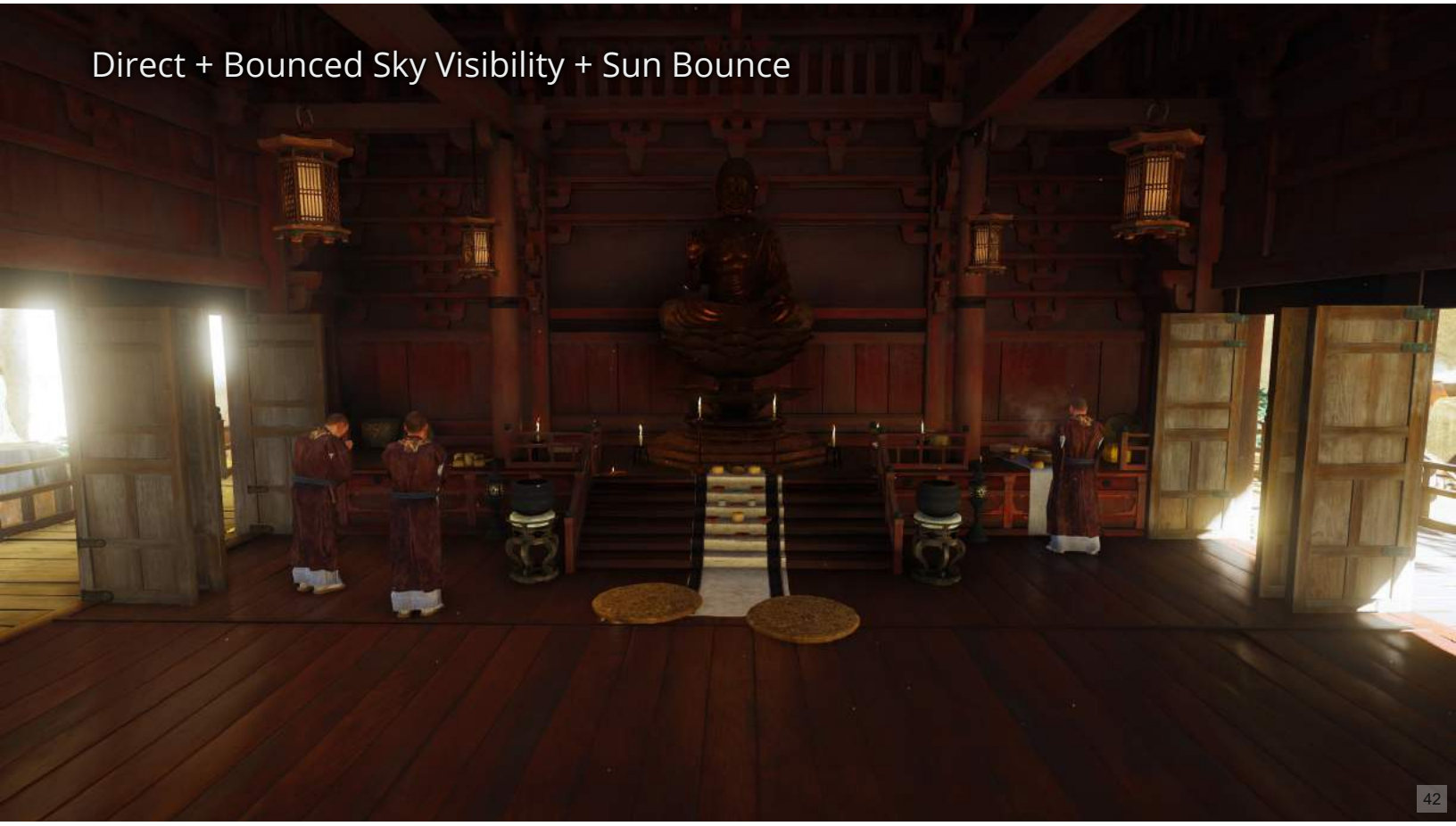
Direct Sky Visibility Only

Speaker notes

... and here is what we get if we use the direct sky visibility.

Speaker notes

Adding bounced sky light brightens thing up a bit (relative to outside)...



Speaker notes

... while adding bounced sunlight warms it up nicely.

Directionality Boost

- Degree 2 SH is relatively low frequency
- Indirect lighting looked flat in many areas
- Lerp towards delta in direction of linear SH maximum
  - Inexpensive to compute
- Separately computed for direct and bounced portions
- Used 25% boost in all weather states

Speaker notes

If you've worked with SH you know that quadratic SH can't represent high frequency data very well. We're multiplying quadratic SH functions together, which gives a degree 4 SH result, but to keep data sizes reasonable we truncate the higher order terms and only store up to degree 2.
[next]
Because of this, our lighting looked a little flat.
[next]
We resorted to another heuristic to improve this: we lerped towards the SH projection of a delta function in the direction of the linear SH maximum,
[next]
which was inexpensive to compute.
[next]
We did this separately for the direct and bounced portions of the irradiance probes.
[next]
In the end, we used a lerp factor of 25% throughout the game.

Speaker notes

Here is a scene without directionality boost applied...

Speaker notes

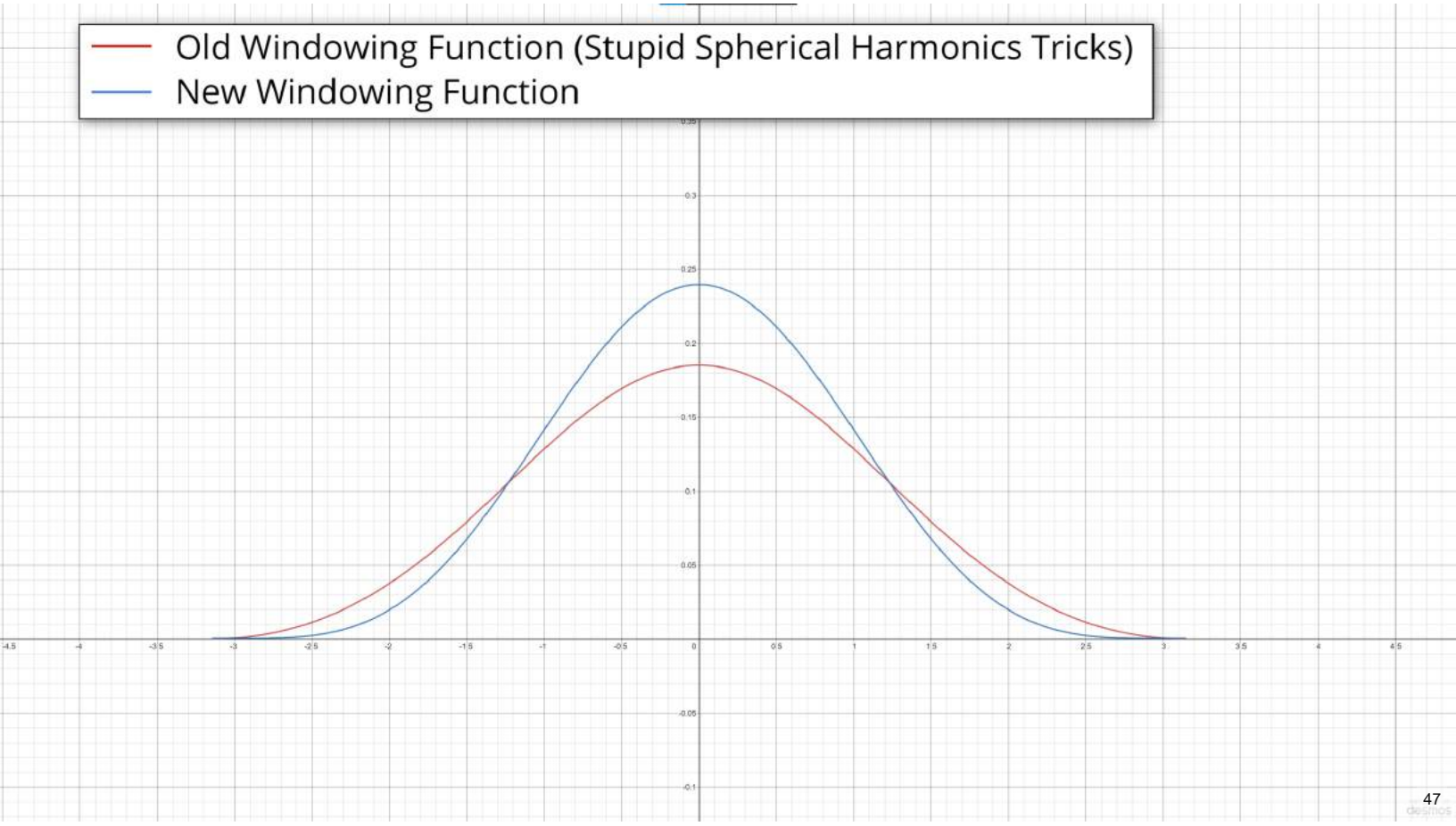
... and here is the same scene with a 25% boost. As you can see, it helps give added definition to the normal maps.

Deringing

- No guarantee that final irradiance will be positive everywhere
- Apply fixed extra deringing to sky visibility?
  - Reduces directionality and fidelity
- Instead dering sky luminance and final irradiance at runtime
  - Peter-Pike Sloan, “Deringing Spherical Harmonics”, SIGGRAPH Asia 2017
- One Newton step sufficient to find the minimum
- Used a depth 3 binary search tree computed on CPU

Speaker notes

If you've worked with SH you probably _also_ know that negative lobes are a problem. In this context, deringing is the process of modifying the SH representation to ensure that it's positive everywhere.
[next]
Initially, we tried applying an extra fixed deringing factor to our sky visibility, but that reduced the directionality and fidelity of the probes.
[next]
Instead, we dering sky luminance and the final irradiance probes at runtime on the GPU,
[next]
using the approach described in this paper by Peter-Pike Sloan.
[next]
In practice we found a single Newton step was sufficient for finding the SH minimum,
[next]
and we used a binary search tree of depth 3 to find the minimum deringing factor necessary to ensure that the SH was positive everywhere.



Speaker notes

The paper presents a new windowing function that gives sharper results than the old windowing function were were using in Infamous, as you can see here, where we're windowing the SH representation of a delta function. The red curve is using the old windowing function, while the blue curve is using the new one.

(Link to graph: https://www.desmos.com/calculator/bopiih3uka)

No Deringing

Speaker notes

Here is a scene that exhibits negative lobes near the window...

Speaker notes

... which is fixed by runtime deringing. Note that the lighting in the rest of the scene doesn't change, since we dering each probe independently.

Performance (Base PS4)

- Sky luminance ⇒ SH: 60 µs (including deringing)

Operation	Tetrahedral Mesh (~5000 probes)	Grid Update (73 tiles)
Full Update (baseline)	60 µs	123 µs
Direct only (no bounce)	30 µs (-50%)	110 µs (-11%)
Without Deringing	53 µs (-12%)	120 µs (-2%)
Without Dir. Boost	60 µs (no change)	123 µs (no change)

Speaker notes

Here are some performance number for the SH updates. Projecting the sky luminance to SH takes about 60 microseconds, including the deringing step. This step used 4096 samples, with one sample per thread, so this does not saturate the GPU (we ran all of our SH updates in async compute in parallel with our depth-only passes).

We updated one tetrahedral mesh per frame, with an average mesh being on the order of 5000 probes (which also does not saturate the GPU). For the regular grid, we updated 8x8 tiles per frame, plus the 3x3 tiles closest to the camera every frame. This was just enough to saturate the base PS4 GPU (though not the PS4 Pro).

Note that the regular grid data interleaved direct & bounce coefficients, so eliminating the bounce from the update has only a small impact on the overall performance (since cache pressure was the same).

Light Leaking

- In *Infamous*, worked around light leaking issues
  - Few interiors, thick walls
- Lots of interiors and thin walls in *Ghost*
- Tried augmenting probes with:
  - Cone occlusion
  - Occlusion planes
  - Occlusion maps
- None of these solved all our problems

Speaker notes

Light leaking arises when a brightly-lit irradiance probe "leaks through" an obstruction such as a wall or roof, causing unnaturally bright lighting on the other side. This was a problem that we managed to work around in Infamous,
[next]
mainly because we didn't have many interiors, and problematic situations could be fixed by placing extra probes in thick walls.
[next]
In Ghost, however, there were lots of interiors, and they all had _very thin_ walls, due to the architecture of the period.
[next]
We tried a variety of solutions to address this problem:
[next]
We added cones to each probe to indicate which neighboring probes were occluded (and should therefore not be sampled),
[next]
we tried adding occlusion planes to probes to represent walls and other obstructions,
[next]
and we tried low-res shadow maps,
[next]
but none of these approaches solved the problem to our satisfaction, and they also had very high performance overhead.

Light Leaking Solution

- Classify tetrahedral mesh probes as **interior** or **exterior**
- Assign surfaces a 0-1 **interior mask** w_{interior}
  - Geometry property or shader parameter
  - Vertex paint
  - Deferred decal
- Calculate barycentrics as normal, then multiply by weight:
  - w_{interior} for interior probes
  - $1 - w_{\text{interior}}$ for exterior probes
- Renormalize barycentrics (use original if all 0)

Speaker notes

We finally found a solution that was workable in terms of runtime overhead and extra authoring burden. First, we classified probes as either being _interior_ or _exterior_.
[next]
We then assign all surfaces an _interior mask_ value w, which defaults to 0.5. This value could be authored using either
[next]
a property on the geometry or shader,
[next]
a vertex color channel,
[next]
or applied separately as a deferred decal.
[next]
To perform the lighting, we calculate the tetrahedral barycentric coordinates as normal, but then we multiply each one by
[next]
w if the coordinate belongs to an interior probe,
[next]
or by 1-w if it belongs to an exterior probe.
[next]
We then renormalize the barycentrics and compute the weighted irradiance blend as usual. (In the event that all the weights are zero, we fall back to the original barycentrics.)

Speaker notes

Here is an interior scene lit without using the interior mask weights.

Speaker notes

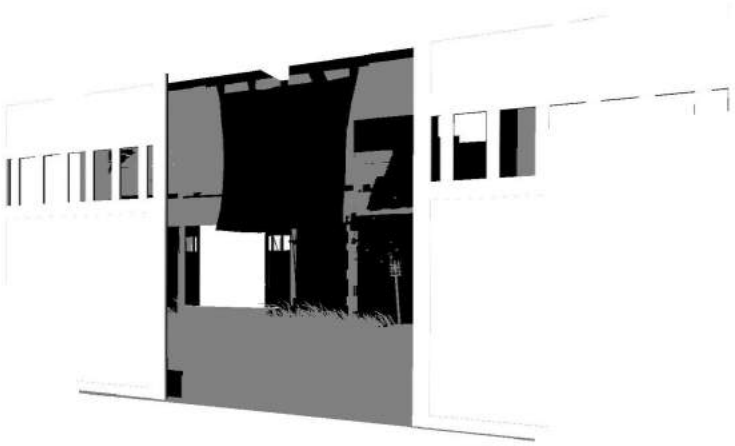
The back wall is brighter than it should be due to light leaking.

Speaker notes

Applying the interior mask fixes the light leaking issue.

Speaker notes

Interior Mask



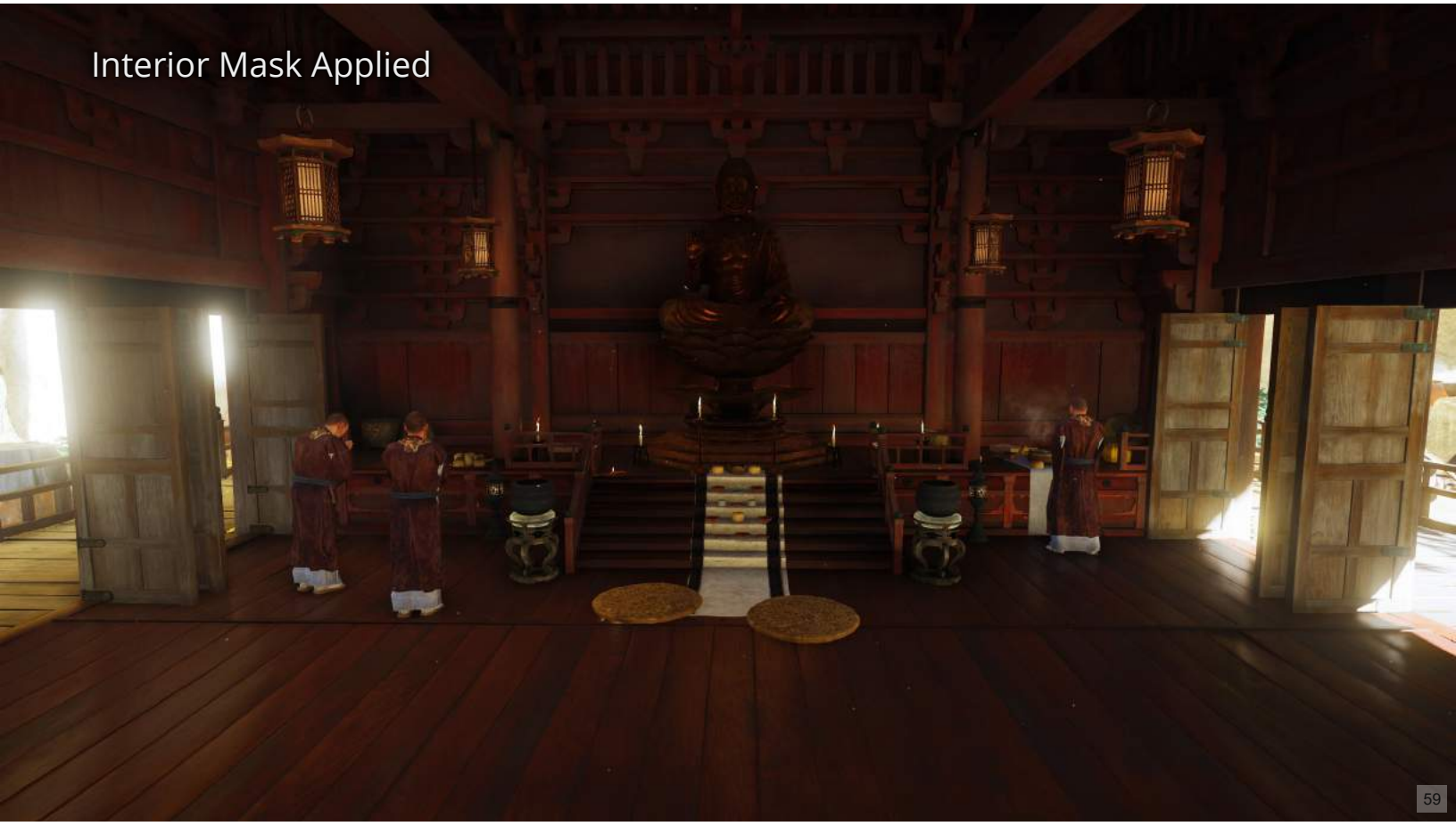
Speaker notes

This shows the value of the interior mask for the scene.

Speaker notes

Here is another example, first without the interior mask...

Interior Mask Applied



Speaker notes

... and here with the mask applied. Note how the light leaking in the upper part of the back wall is fixed.

Limitations

- Baked data is completely static
  - Objects that could move or be destroyed usually omitted from the bakes
- Bounced light assumption (shadowing ≈ sky visibility) is often wrong
  - Bounced light either too strong or too dim
- No indirect light from local lights

Speaker notes

Indirect Lighting | Specular

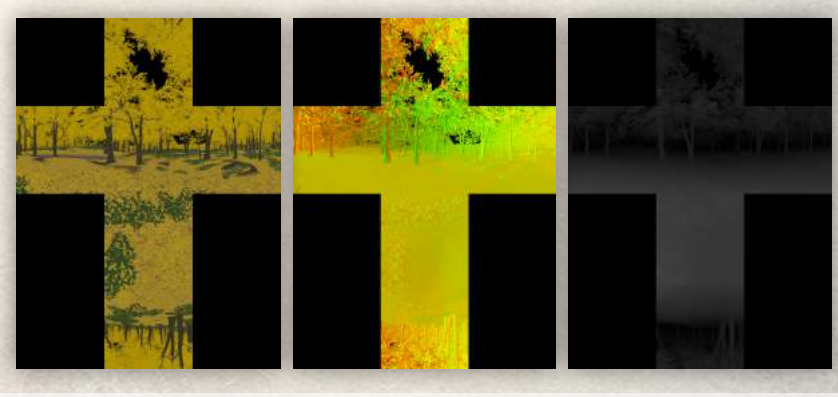
GHOST

- Seattle in *Infamous* used 230 static reflection probes
  - Already pushing memory limits
- Tsushima used 235
  - Extra data required for relighting
  - Streamed on demand
  - Per-biome default probes
  - Instanced interior probes
- Added support for nested probes
- Up to 128 relit probes at a time

Speaker notes

Ok, let's move on to indirect specular lighting. Infamous: Second Son used 230 static reflection probes,
[next]
which was already pushing it, memory-wise.
[next]
We ended up using 235 in Tsushima,
[next]
but because of the extra data required for relighting (1.5 times than the size of the compressed lit probes),
[next]
we switched to streaming them on demand.
[next]
We were able to reduce the number of probes needed by adding per-biome default probes,
[next]
as well as instancing many interior probes (e.g. for many smaller houses).
[next]
We also added support for one level of nesting (due to the many interiors in the game);
[next]
in total we supported up to 128 relit probes at any one time.

- Offline captured data:
  - Albedo cube map (BC1 format)
  - Normal + Depth cube map (BC6H format)
    - RG: Octahedral normal encoding
    - B: Depth (hyperbolic)
  - All cube maps 256x256x6



Speaker notes

We capture the following data for relighting:
[next]
An albedo cube map compressed to BC1,
[next]
and a normal + depth cube map compressed with BC6H.
[next]
We use the red and green channels to store octahedral normals,
[next]
and we we store hyperbolic depth in the blue channel.
[next]
By the way, all of our cube maps have size 256.

Reflection Probe Relighting

- Performed round-robin in async compute (one per frame)
- Shadowed with far shadow map atlas
  - Per-tile shadow map, 128x128 texels
- Single SH sample for indirect lighting
  - Also used as for reflection probe luminance for normalization
- Prefiltered using filtered importance sampling
- Compressed to BC6H using GPURealTimeBC6H
  - Reduced memory footprint and improved sampling performance

Speaker notes

To relight our probes, we pick one per frame and update it in async compute.
[next]
Since the probes volumes aren't necessarily covered by our cascaded shadow maps, we use our far shadow map atlas,
[next]
which has a 128x128 shadow map per 200m tile.
[next]
For performance we use a single SH sample for indirect lighting;
[next]
we also use this sample (converted to luminance) for reflection probe luminance normalization. This ensures that a probe used at its capture location will have the expected luminance, and it avoids having to project the probe to SH.
[next]
We pre-filter the probes using filtered importance sampling of the GGX NDF,
[next]
and compress them to BC6H using Krzysztof Narkowicz's excellent open source compressor.
[next]
This was very helpful in reducing the memory footprint, as well as preventing cache thrashing on surfaces like water, which have high gloss and highly varying normal maps.

Cube Map Shadow Tracing

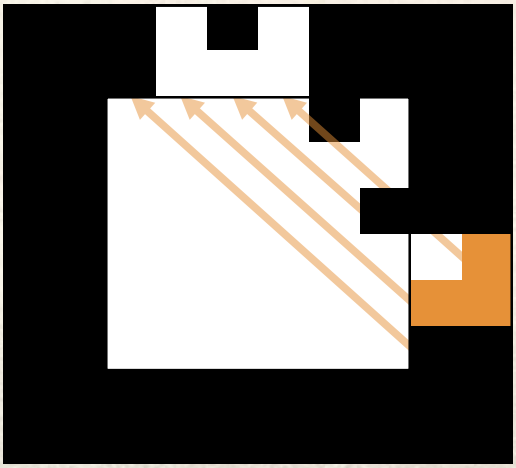
- Far shadow maps insufficient for shadowing interiors
  - Combination of low shadow resolution and LOD
- Insight: depth cube maps have lots of occlusion information
- When relighting each cube map texel:
  - Find intersection of directional light ray with cube map volume
  - Sample depth at intersection
    - Sky depth ⇒ unoccluded
    - Used 4x4 PCF

Speaker notes

One problem we ran into was that our interior probes weren't correctly shadowed,
[next]
because our far shadow maps weren't up to the task, partly due their low resolution, partly due to the LODing of many buildings when rendered to far shadow maps.
[next]
We realized that our cube maps contain lots of occlusion information, since they store depth, so we use that for shadowing.
[next]
When relighting a texel,
[next]
we trace back towards the light to find the intersection with the cube map volume,
[next]
and sample the depth from the cube map at that intersection.
[next]
We treat any sufficiently small value as being unoccluded (we're using 1-z depth buffers),
[next]
and to help soften the shadows we sample in a 4x4 footprint and apply PCF filtering.

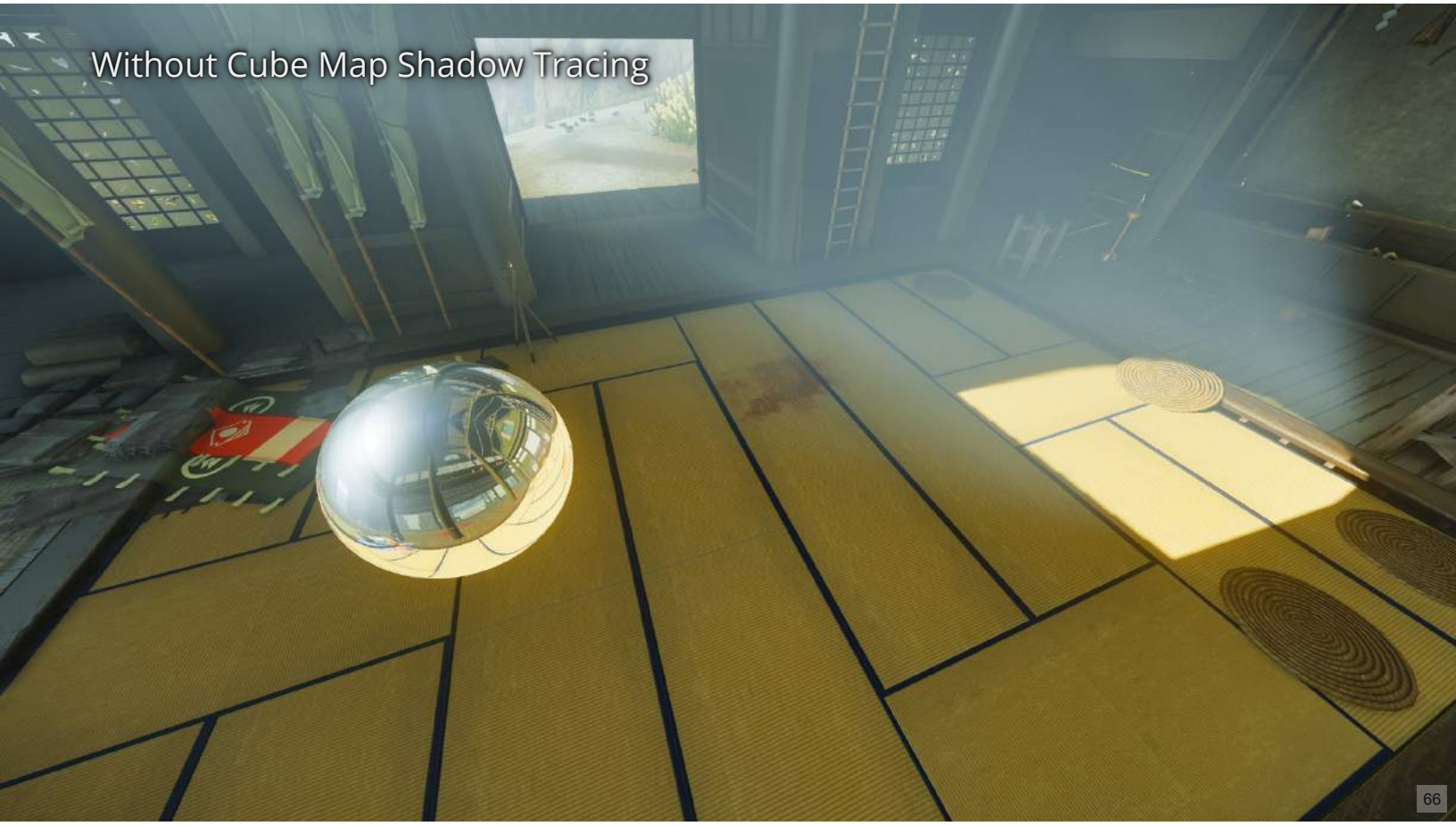
Cube Map Shadow Tracing

- Fairly crude, but cheap and effective



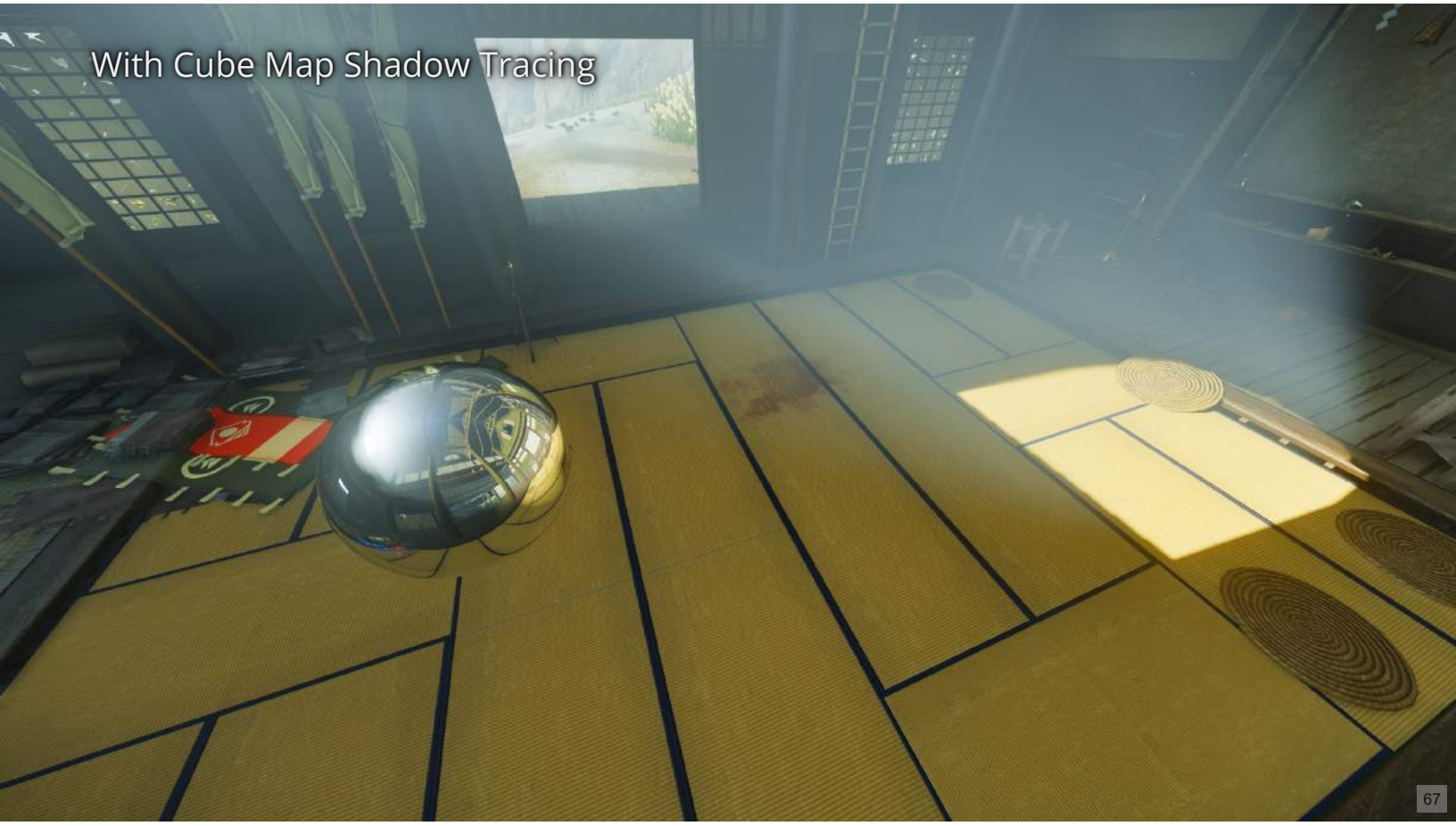
Speaker notes

This technique is fairly crude, and we could improve the accuracy by taking multiple depth samples along the ray, but this version worked well enough for our purposes. Let's step through a few sample rays.
[next]
Here's our first ray,
[next]
and it hit something,
[next]
so that texel is shadowed.
[next]
Our next ray
[next]
doesn't hit anything, so that texel is unshadowed.
[next]
With this ray we can see some of the problems with this approach, since it clearly should be shadowed,
[next]
but according to its depth sample it is not.
[next]
Finally, this ray
[next]
thinks it hit something,
[next]
so it's shadowed, though if we kept marching beyond the volume's bounds we'd perhaps find it to be unshadowed.



Speaker notes

Here's a scene where the probes aren't using cube map shadow tracing...



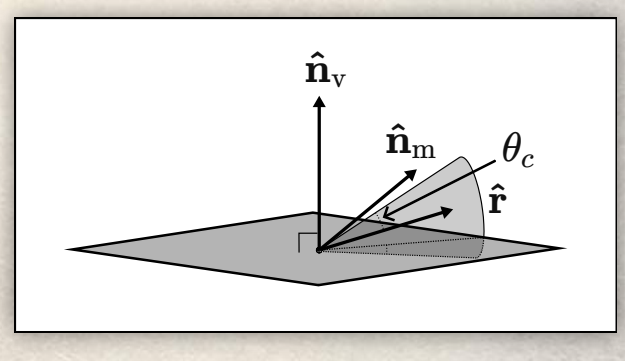
Speaker notes

... and this is the same scene with it enabled. As you can see it prevents much of the "glowing" visible in the previous version, and the floor and walls in the reflections closely match the scene.

Horizon Occlusion

- Account for occlusion of underlying geometry on reflection cone caused by tilt of normal-mapped normal $\hat{\mathbf{n}}_{\mathrm{m}}$ relative to vertex normal $\hat{\mathbf{n}}_{\mathrm{v}}$
- Fast, approximate, plausible results
- For GGX roughness α, the cone half-angle θ_c containing the fraction u_{e} of the energy is given by

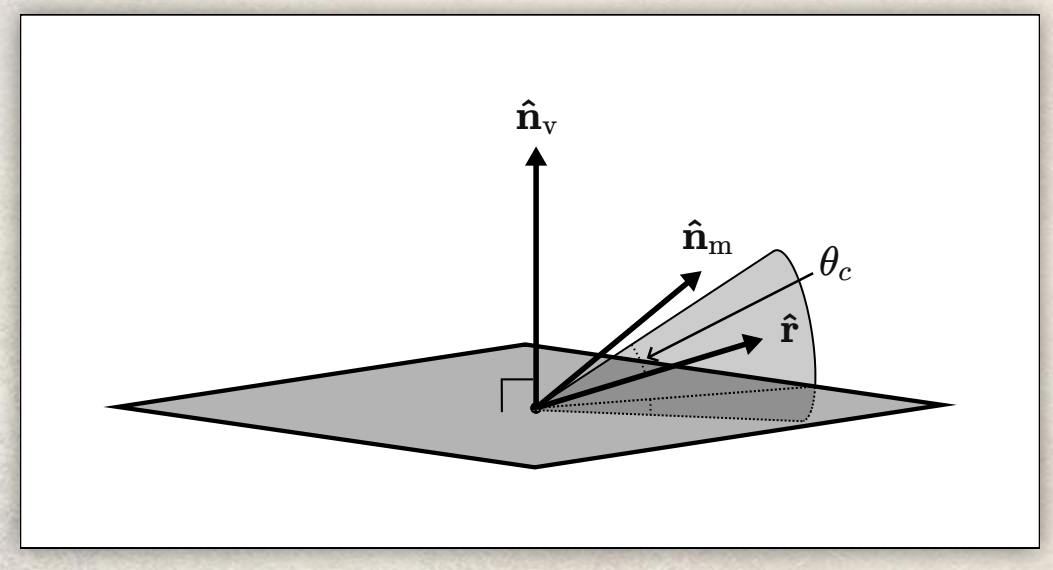
$$\tan\theta_c = \alpha\sqrt{\frac{u_{\mathrm{e}}}{1-u_{\mathrm{e}}}}$$



Speaker notes

When lighting with our reflection probes, our artists complained that the back sides of normal map bumps seemed too bright, since the reflected light should be partially occluded by the macro-geometry. So we developed what we called a "horizon occlusion" term to account for the tilt of the normal-mapped normal n_m relative to the vertex normal n_v.
[next]
The input space to this problem is four dimensional: one dimension for the surface roughness, one for the tilt of the normal-mapped normal relative to the vertex normal, and two for the view vector orientation. We wanted something that was fast while giving plausible results, so we developed this approximation.
[next]
First, we approximate the reflected rays with a cone, by using the following relation: for a GGX roughness alpha, the cone half-angle theta_c containing the fraction u_e of the energy is given by this equation. We used a value of 0.95 for u_e, and we fit a curve to make computing the cone angle less expensive.

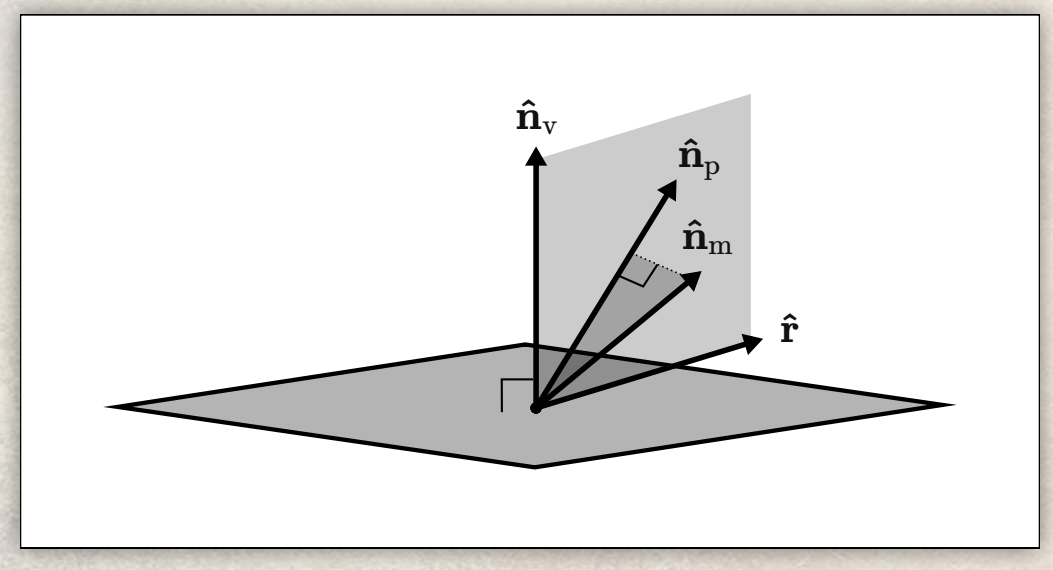
Horizon Occlusion



Speaker notes

To simply the problem...

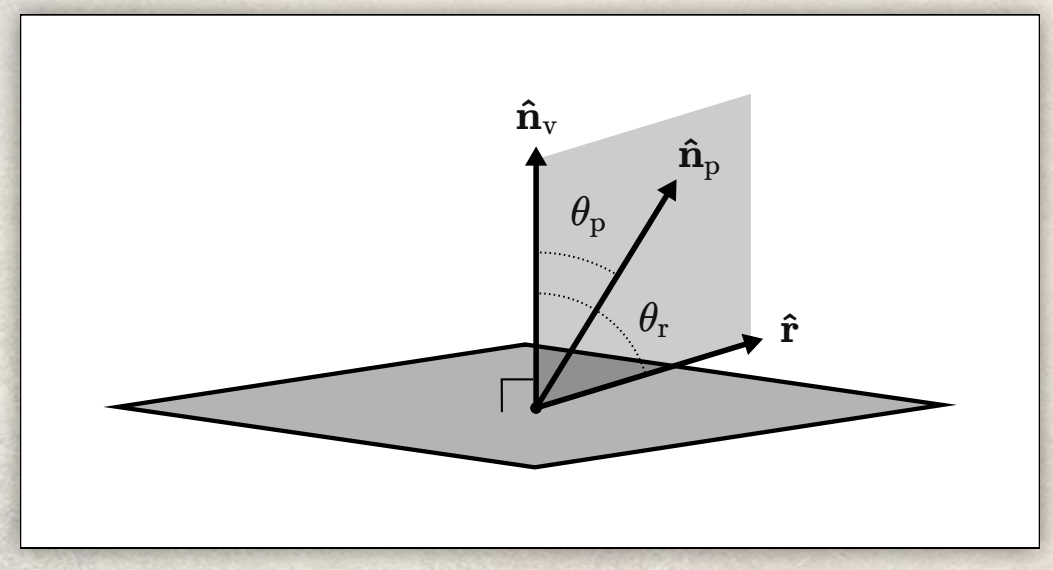
Horizon Occlusion



Speaker notes

... we first project the normal-mapped normal n_m into the plane formed by the reflection vector and the vertex normal; we'll call this projected normal n_p.

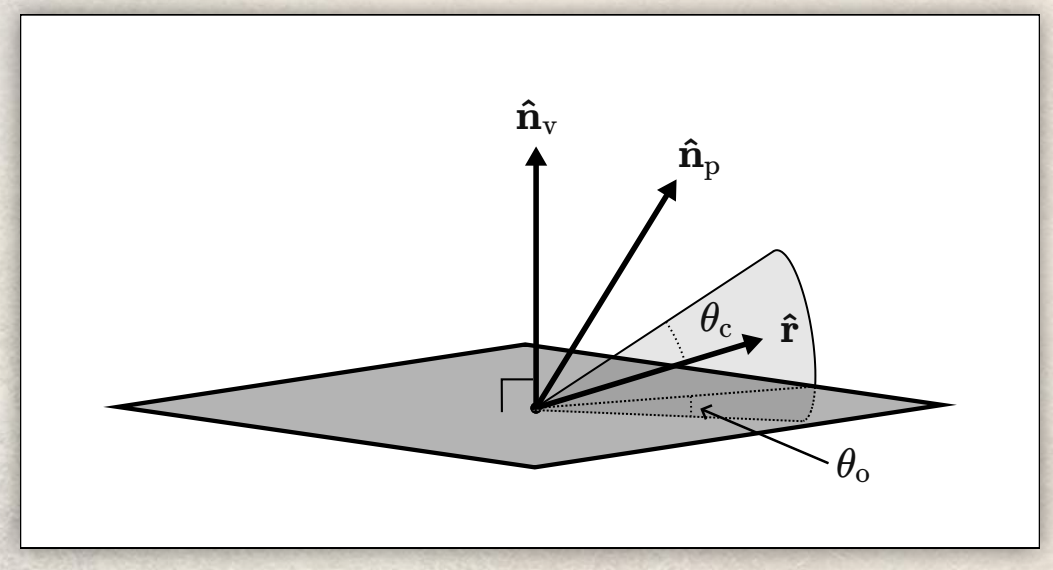
Horizon Occlusion



Speaker notes

Then, we let theta_p be the angle between this projected normal and the vertex normal, and theta_r be the tilt of the reflection vector.

Horizon Occlusion



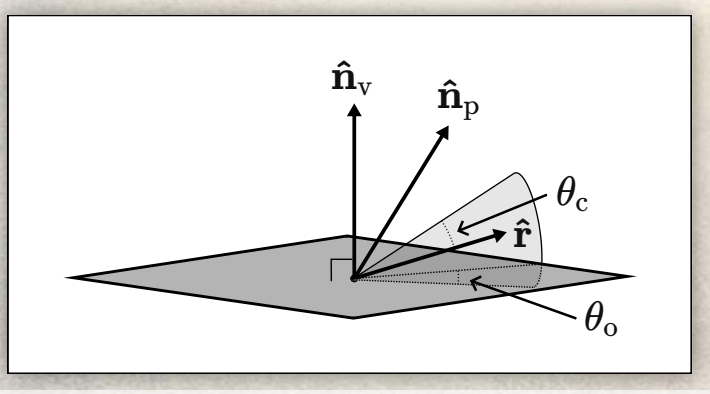
Speaker notes

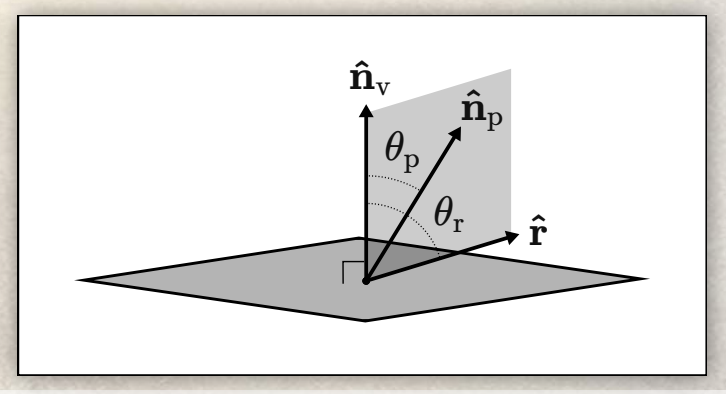
We define theta_o to be the angle that the cone dips below the surface...

Horizon Occlusion

$$\theta_o = \min\left(\theta_r + \theta_c - \frac{\pi}{2}, 2\theta_p\right)$$

$$u_o = u_e \,\text{smoothstep}\,(0, 2\theta_c, \theta_o)$$





Speaker notes

and we calculate it as follows. Since the BRDF already accounts for occlusion when theta_p is zero, and since a one degree change in theta_p results in a two degree change in theta_r, we clamp to two times theta_p, so that we only count extra occlusion.
[next]
We then compute the final occlusion amount by using the following approximate formula. Note that we never exceed u_e, which is the fraction of the energy in the cone (95% in our case).

Speaker notes

Here's a riverbed rendered with parallax occlusion mapping but without horizon occlusion...

Speaker notes

and here it is with horizon occlusion enabled; note how the back sides of the rocks no longer "glow" unrealistically.

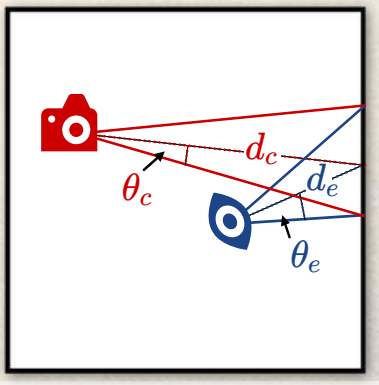
Roughness Parallax Compensation

- Parallax correction within reflection probe volumes distorts apparent roughness
- For current eye position d_e units from sample location, with cube map capture location d_c units away, we have

$$\frac{\tan\theta_e}{\tan\theta_c} = \frac{\alpha_e}{\alpha_c} \approx \frac{d_c}{d_e}$$

- So, when sampling the cube map we use

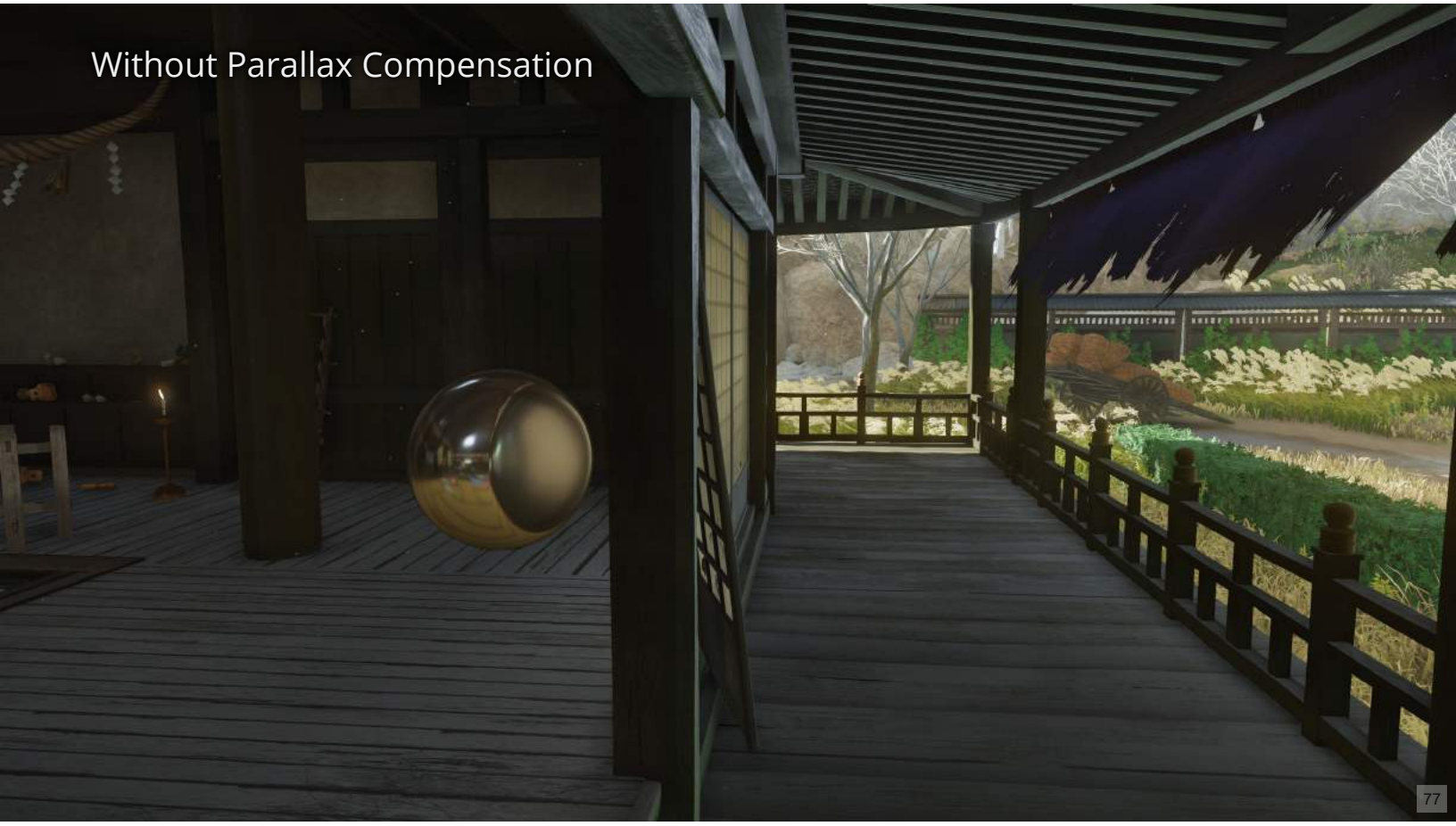
$$\alpha_c \approx \alpha_e \frac{d_e}{d_c}$$



Speaker notes

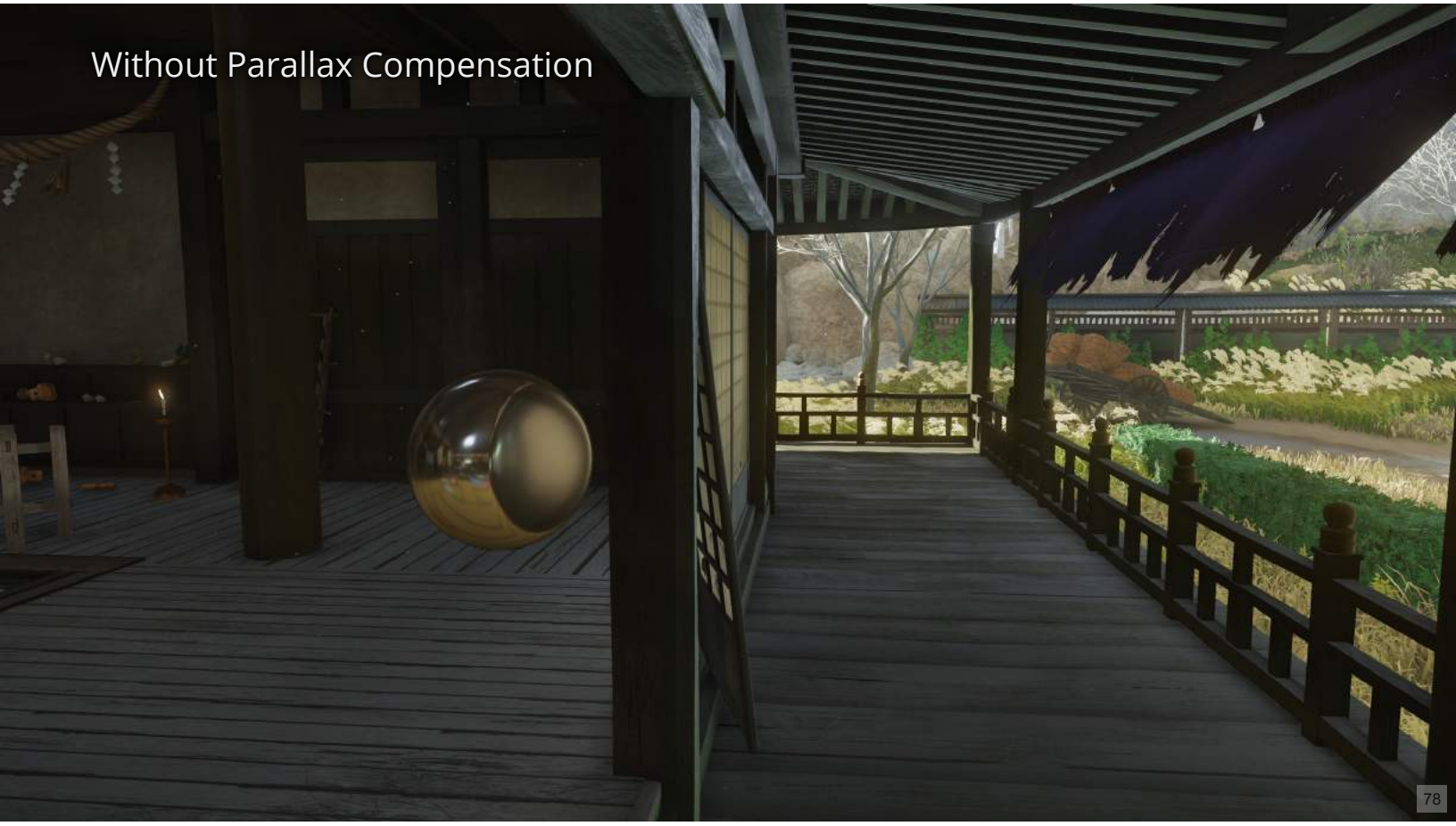
We apply parallax correction to our reflection probes; that is, we model the reflected environment using an authored bounding box, and compute the intersection of the reflection vector with that box, which gives us the cube map sample location.

In the diagram on the right, the cube map was captured from the point of view of the red camera, while the current view is represented by the blue eye. Let's say that the eye is d_e units away from the sample location, while the camera is d_c units away. We can reuse the formula we saw earlier relating the tangent of the cone angle to roughness for a given fraction of energy. Dividing the expressions for the two views cancels out everything but the roughness terms on the right (that is, alpha_e over alpha_c). We then observe that this is approximately equal to the ratio of the distances,
[next]
so we adjust the roughness by this factor when sampling the cube map.



Speaker notes

Here is a scene without parallax compensation...



Speaker notes

The reflection of the shoji screen in the slightly rough chrome sphere is blurrier than it should be.

Speaker notes

Applying parallax compensation fixes the issue.

Performance (Base PS4)

Step	Time (µs)
Relight reflection probe	347
Generate mipmaps	37
Convolve with GGX NDF	250
Compress to BC6H	60
Total	**700**

- Shadow tracing accounts for ~16% of the relighting cost
- Horizon occlusion accounts for ~3% of deferred lighting cost
- Roughness parallax compensation has negligible cost

Speaker notes

All of the reflection probe relighting steps were performed in async compute in parallel with our depth-only passes. We relit one probe per frame.

Speaker notes

I'll now discuss some of the techniques we used to light our skies, clouds, haze, and fog.

Atmospheric Lighting

GHOST

- All of our atmospheric lighting used pre-calculated 3D LUTs
  - Rayleigh and Mie inscattering (divided by phase functions) indexed by altitude, sun/moon polar angle, and view polar angle
  - Multiple scattering
  - Bruneton08, Elek09, Yusov13
  - Analytic Earth shadow term
- Precision issues when using this LUT to light clouds, haze, and fog
  - Extended with parallel LUT to store radiance divided by phase function
  - I.e., average irradiance

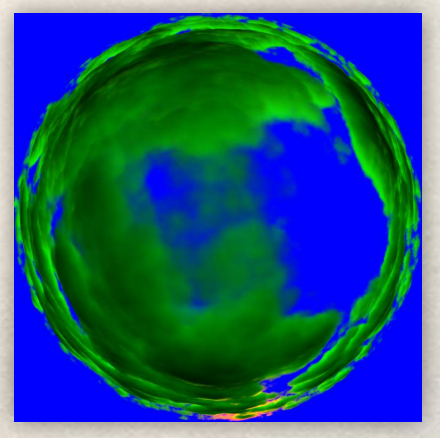
Speaker notes

As is pretty common these days, we used precomputed 3D lookup tables to light our skies. However, we also used them to light the other atmospheric phenomena in the game -- that is, clouds, volumetric haze, and fog particles.
[next]
The LUTs used for the sky stored Rayleigh and Mie inscattering (divided by their respective phase functions), indexed by altitude, directional light polar angle, and view polar angle.
[next]
The LUTs include multiple scattering bounces,
[next]
based on the papers by Bruneton, Elek, and Yusov.
[next]
Since the 3D LUTs do not account for azimuthal angle, we applied an analytic earth shadow term at runtime.
[next]
We ran into precision issues, especially at the horizon, when trying to use this LUT to light clouds, haze, and fog,
[next]
so we added parallel LUTs (indexed in the same way) that store the scattered Rayleigh and Mie light at each point. These are the values *prior* to multiplying by the local density, and divided by the phase functions as before.
[next]
This has units of lux and can be thought of as a kind of average irradiance at each point.

Atmospheric Lighting

GHOST

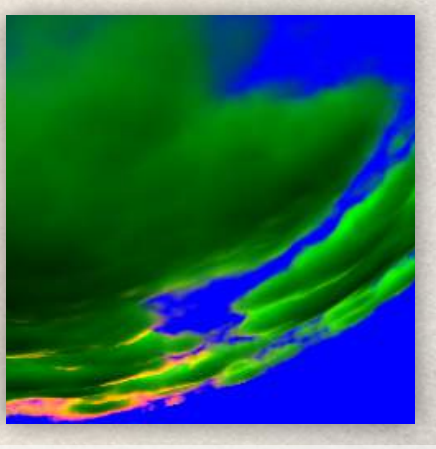
- Apply shadowing to the Mie irradiance, and ambient occlusion to the Rayleigh irradiance
  - Mie phase function is strongly forward-scattering
  - Rayleigh scattering is closer to uniform
- Haze uses average sky visibility for AO
  - Plus rain shadow map for interiors
- Clouds render to a paraboloid texture
  - Rgb11f
  - Red: Mie
  - Green: Rayleigh
  - Blue: Transmittance



Speaker notes

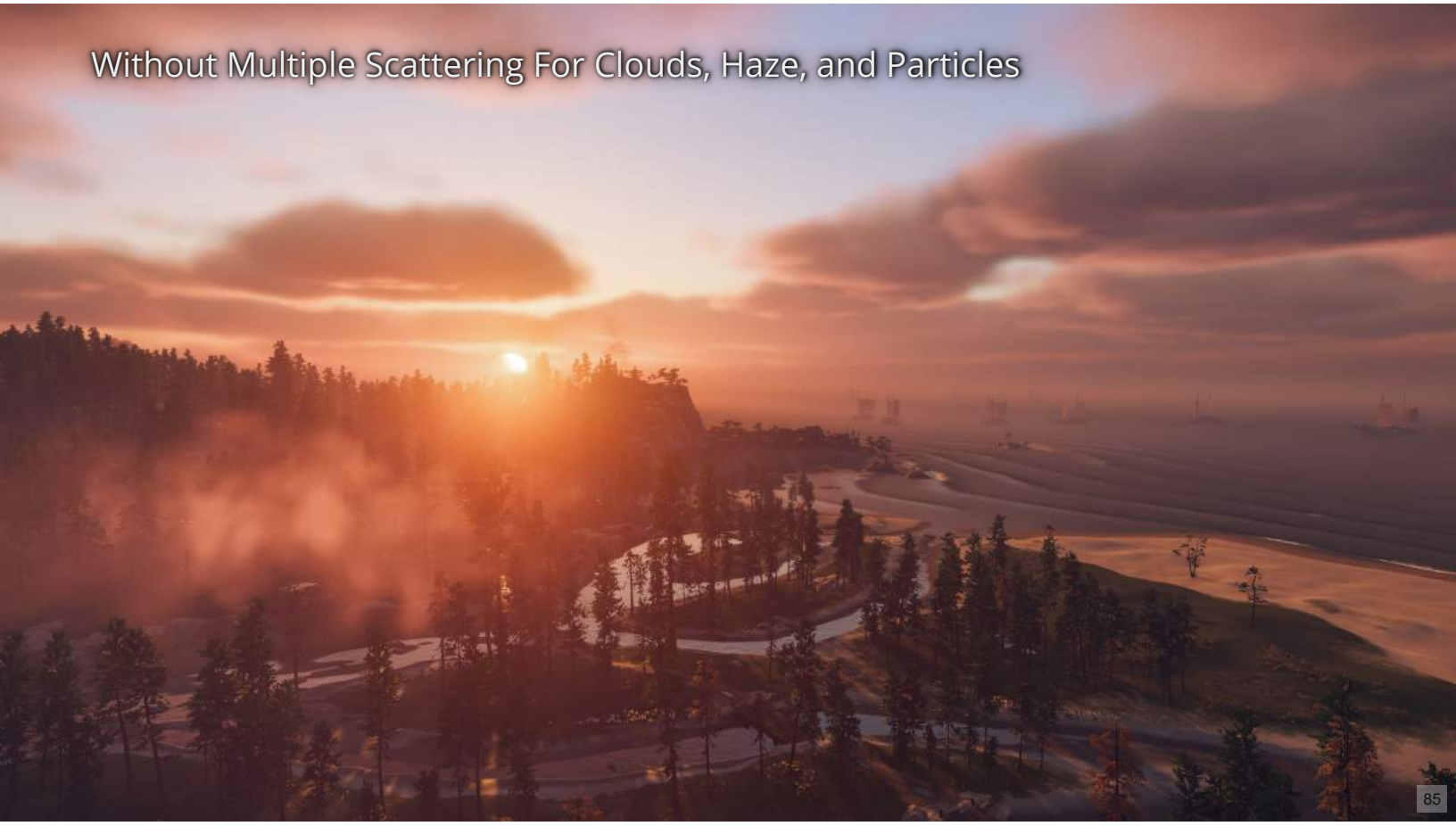
When using these irradiance LUTs for lighting, we apply shadowing to the Mie irradiance, AO to the Rayleigh irradiance.
[next]
This is because Mie scattering is strongly forward-scattering,
[next]
while Rayleigh scattering is much more uniform.
[next]
Volumetric haze samples average sky visibility from the indirect lighting data and uses that for AO,
[next]
supplemented by the rain shadow map to prevent overly-bright haze in interiors.
[next]
Clouds render to a paraboloid texture,
[next]
in Rgb11f format,
[next]
where the red channel stores the Mie scattering, which is then multiplied by the Mie irradiance from the LUT;
[next]
the green channel stores the Rayleigh scattering, which is then multiplied by the Rayleigh irradiance from the LUT;
[next]
and the blue channel stores the transmittance.

Atmospheric Lighting



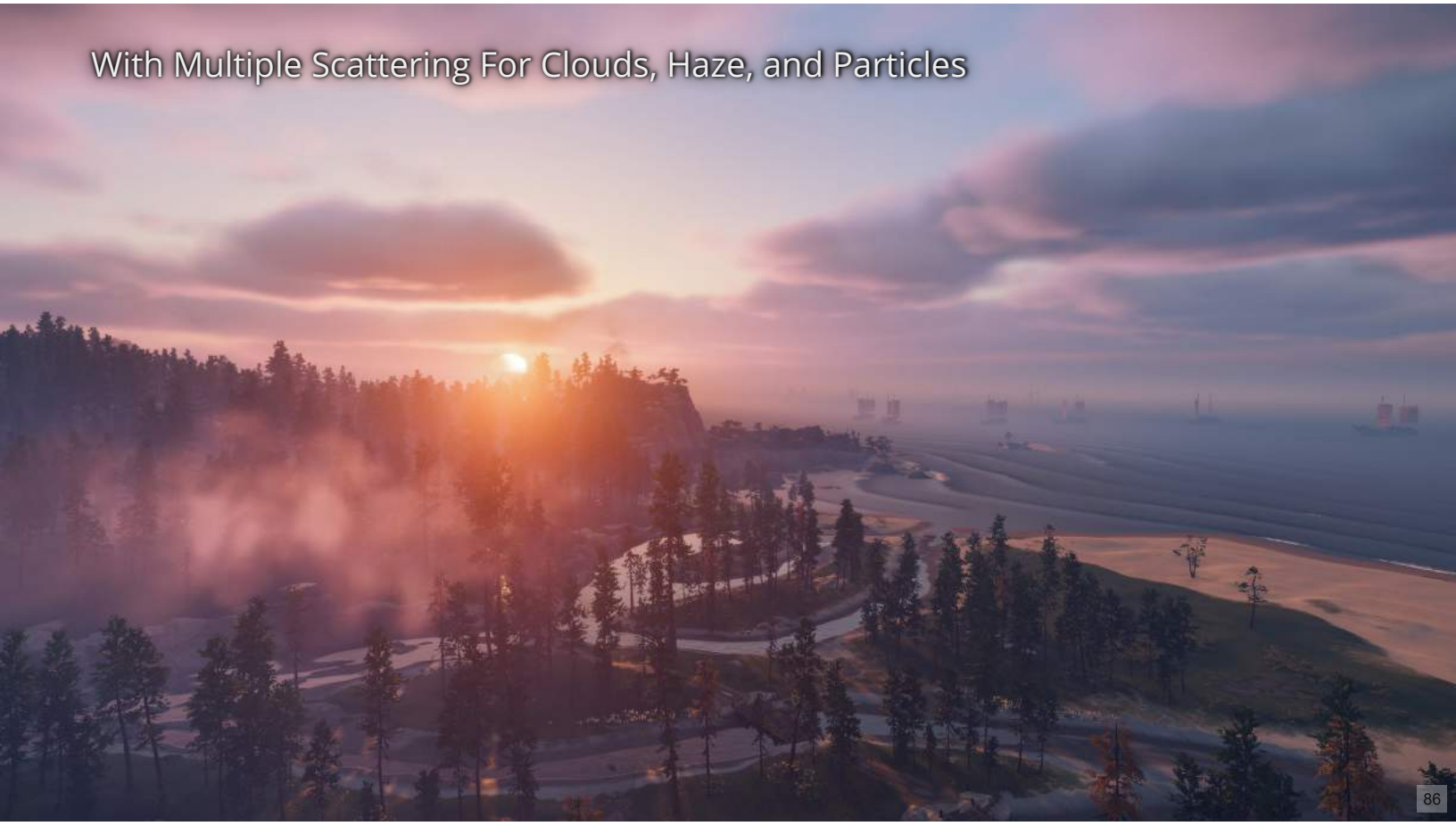
Speaker notes

This is the scene that corresponds to the cloud map,
[next]
and here I've zoomed into the part of the cloud map that is visible in the center of the scene.



Speaker notes

The benefit of using the LUTs for lighting is that it lets us include multiple scattering in our cloud, haze, and fog lighting. Here is a scene without multiple scattering...



Speaker notes

and here is the same scene after including multiple scattering; as you can see, it has a big impact on the look of scene, especially at sunrise and sunset.

Atmospheric Lighting

GHOST

- Each frame, resample 3D LUTs into 2D for current sun and moon angles
- Upsample using cubic filtering to avoid aliasing at sunrise and sunset
- Makes later lookups cheaper

Speaker notes

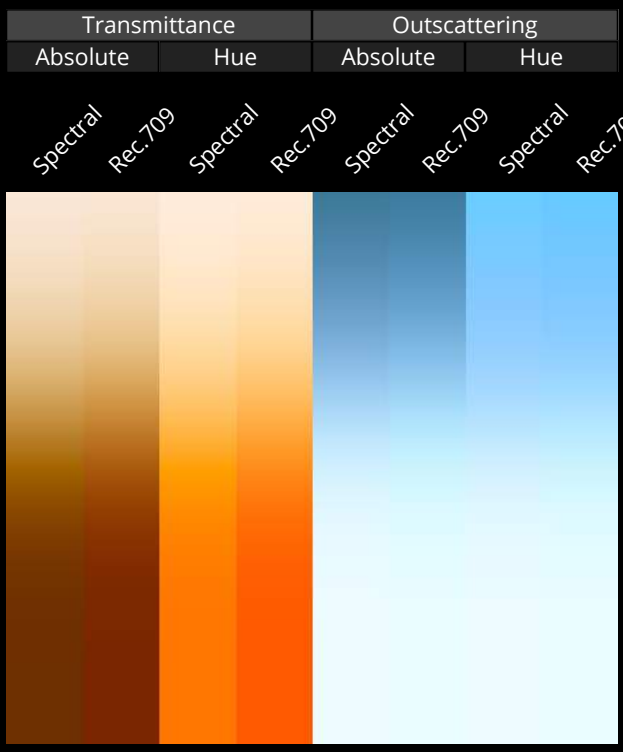
Now for some implementation details: each frame, we resample the 3D LUTs into 2D, using the current sun and moon angles.
[next]
We use cubic upsampling, which avoids aliasing issues at sunrise and sunset.
[next]
It also makes the LUT lookups cheaper.

Custom Rayleigh Color Space

- The atmosphere drives most of our world lighting, so getting Rayleigh scattering right was important
- A blog post by Christian Schüler gave us the idea to try to find a color space that gave results closer to spectral rendering
- Minimized the error in Rayleigh transmittance in two steps:
  1. Choosing three wavelengths (LMS) for our primaries with point-sampled Rayleigh coefficients
  2. Refined the Rayleigh coefficients for the color space chosen in step 1.

Speaker notes

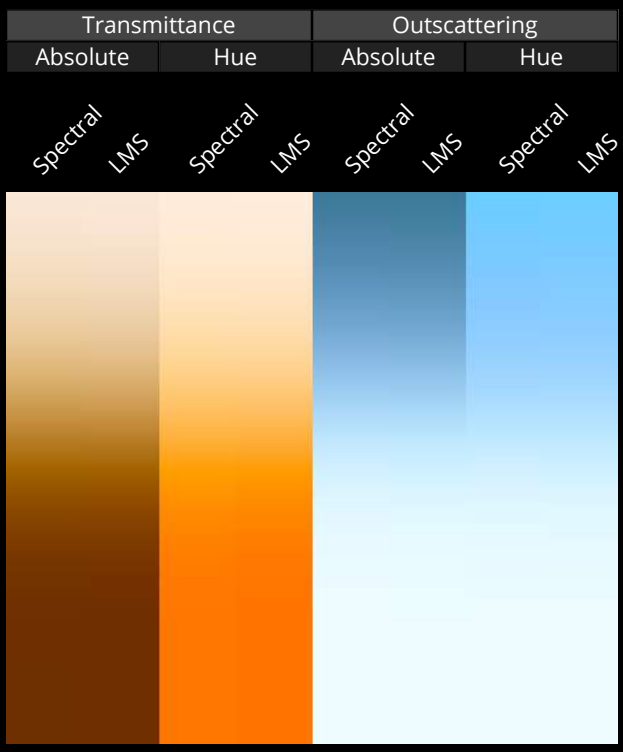
Since atmospheric scattering drives most of our world lighting, it was important for us to try to get Rayleigh scattering to be as accurate as possible.
[next]
We found a blog post by Christian Schüler that discussed using a custom color space to approximate spectral rendering, so we decided to give that try.
[next]
As in the blog post, we used spectral primaries for the color space. We minimized the error in Rayleigh transmittance over a range of sun angles, weighting shallow angles corresponding to sunrise and sunset more heavily. The optimization consisted of two steps:
[next]
First, we found the three wavelengths (long, medium, and short) for our spectral primaries that minimized this error, using point-sampled Rayleigh scattering coefficients at these wavelengths.
[next]
We then fixed the primaries and allowed the Rayleigh coefficients to vary, refining the result found in step 1.



Spectral vs Rec.709 with Bruneton08 Coefficients

Speaker notes

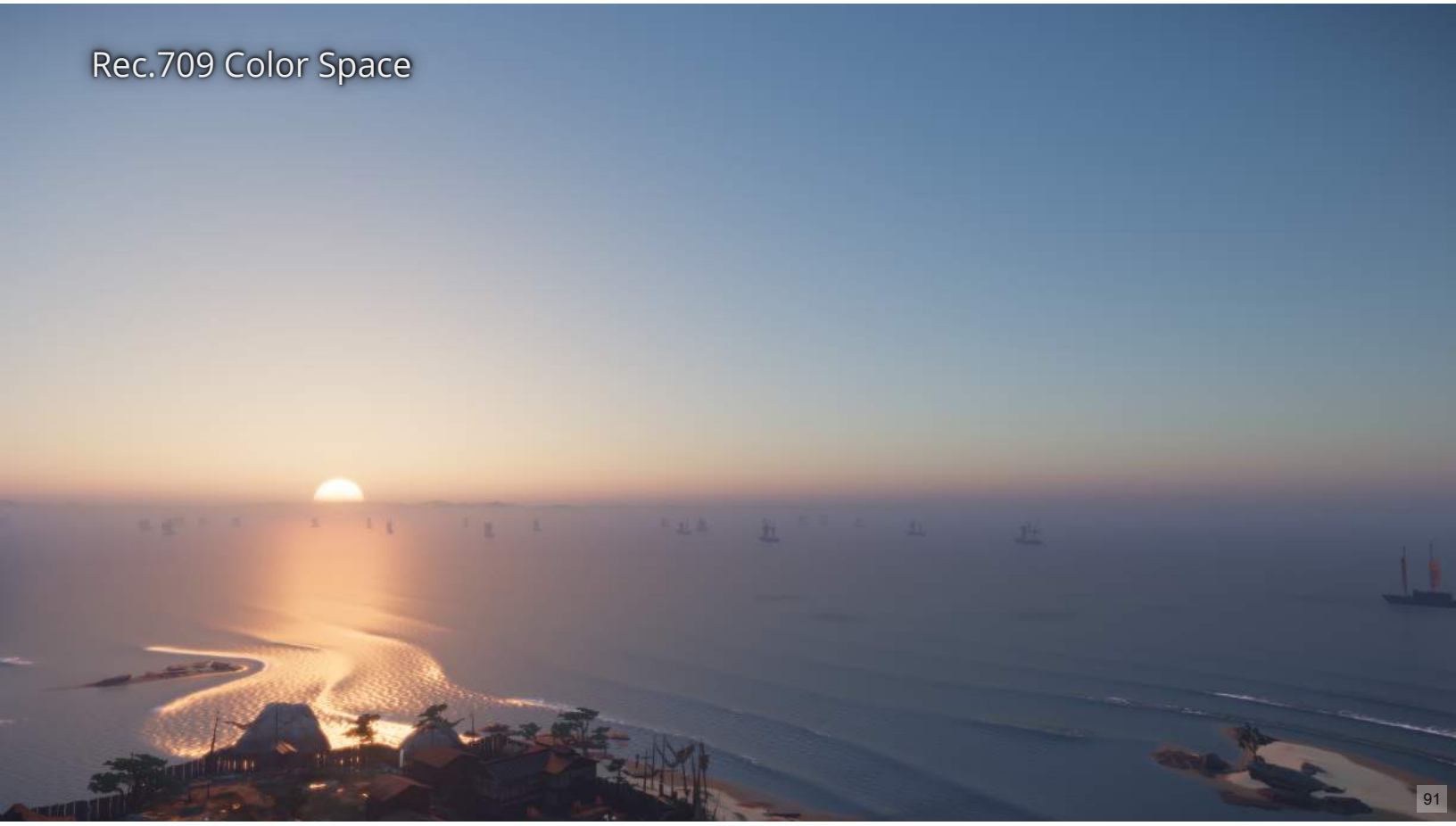
This shows a comparison of spectral Rayleigh scattering, versus scattering computed in Rec.709 RGB space, with Bruneton scattering coefficients. The left half shows the transmittance, while the right half shows the outscattering. As you can see there is a noticeable difference in both the absolute value of the transmittance, as well as the hue.



Spectral vs LMS Color Space

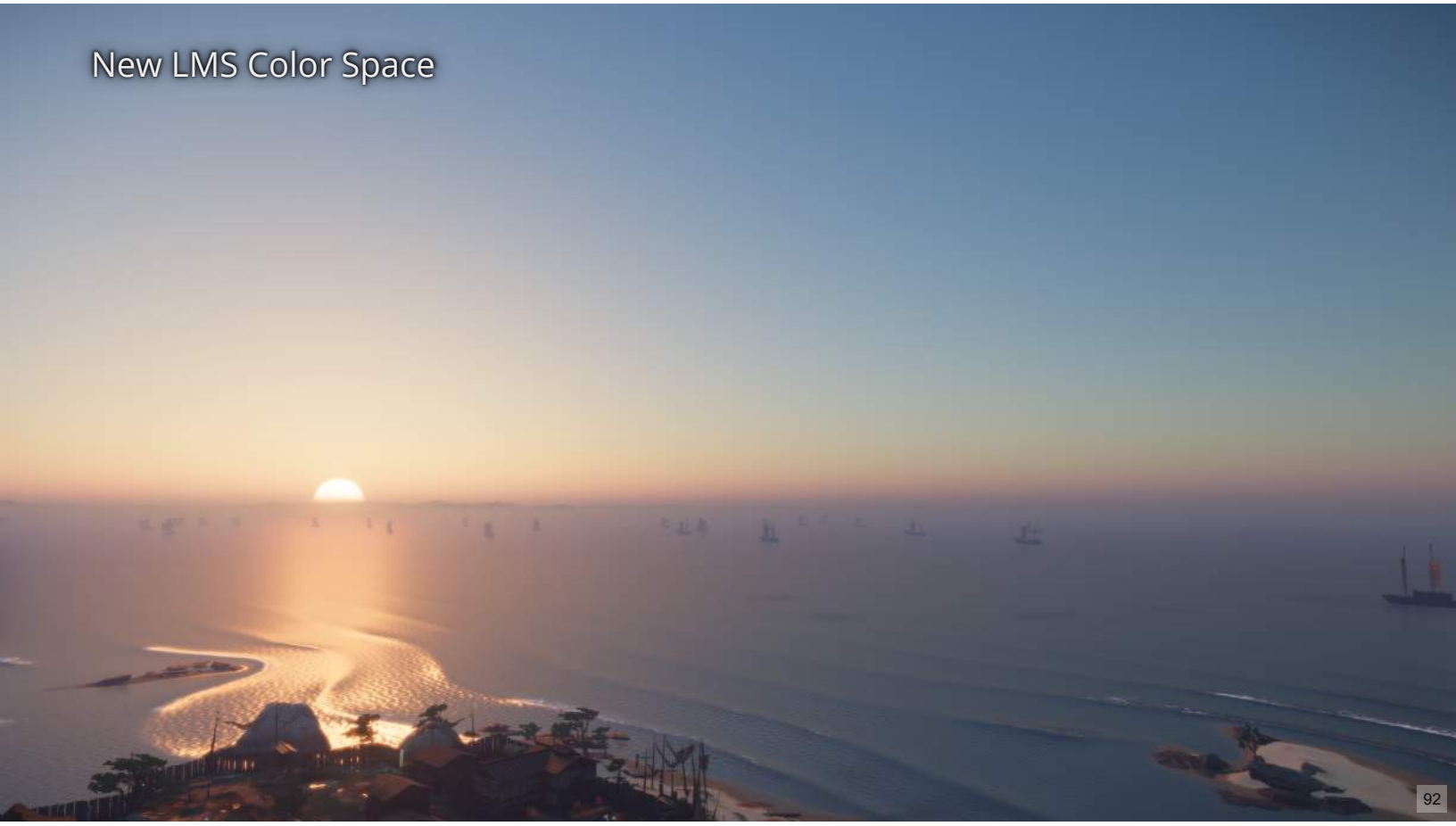
Speaker notes

Here is a comparison using our LMS color space and scattering coefficients. The results very closely match those of spectral rendering.



Speaker notes

Here is a scene using the Rec.709 color space with Bruneton08 coefficients...



Speaker notes

... and here is the same scene using the new LMS color space and scattering coefficients. The difference is most pronounced at sunrise and sunset, and helps to reduce the greenish hue in the sky that we sometimes noticed with the old color space.

Custom LMS Rayleigh Color Space

White point (D65):

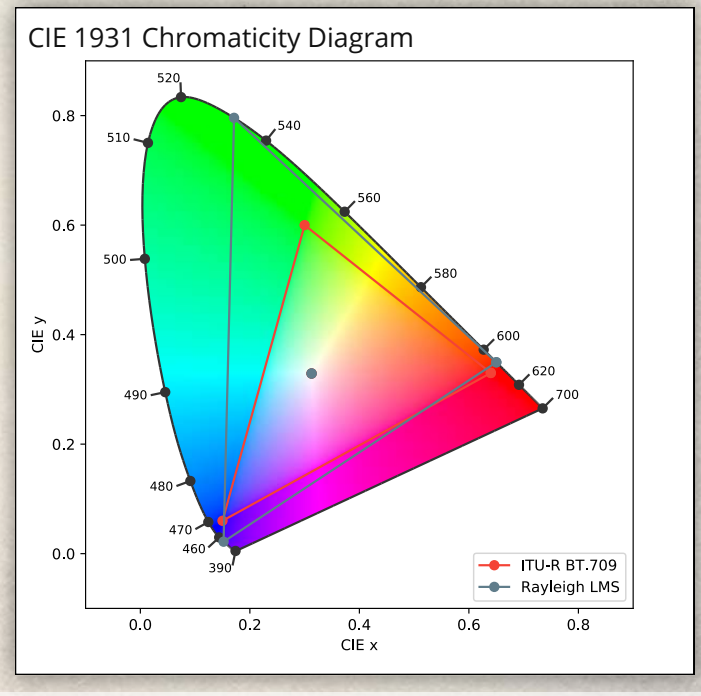
$$x_W = 0.3127$$
$$y_W = 0.3290$$

Primaries:

$$x_L = 0.6501 \quad | \quad x_M = 0.1711 \quad | \quad x_S = 0.1520$$
$$y_L = 0.3495 \quad | \quad y_M = 0.7959 \quad | \quad y_S = 0.0218$$

Rayleigh scattering coefficients:

$$\beta^R_{\text{LMS}} = \begin{bmatrix} 0.0076224 & 0.012935 & 0.024845 \end{bmatrix}^T \text{km}^{-1}$$



Speaker notes

Here are the details for the new color space. Note that we used the same D65 white point as Rec.709.

Atmospheric Lighting | Clouds

GHOST

- Schneider 2015, Hillaire 2016
- Clouds rendered to 768x768 paraboloid texture
  - Three textures:
    - Two for scrolling & lerping
    - One for rendering (timesliced over 60 frames)
- Antialiasing for cloud density:
  - Calculate derivatives of:
    - Radial cloud position with radial texture coordinate
    - Angular cloud position with angular texture coordinate
  - Reduce cloud density when derivative is high

Speaker notes

I'll now talk a little bit more about our cloud rendering. We built on the ideas presented by Andrew Schneider in this course in 2015, and by Sebastien Hillaire in the Physically Based Shading course in 2016.
[next]
As I mentioned earlier, we rendered our clouds to a paraboloid map that covers the entire hemisphere. For performance reasons it had to be fairly low-resolution -- only 768x768.
[next]
We used three of these textures;
[next]
two that we blend between while for scrolling in the direction of movement,
[next]
and one that is updated over 60 frames, to keep the overhead as low as possible.
[next]
Because of the low resolution, we needed to reduce the cloud density in areas where the cloud position changes quickly in UV space.
[next]
To do this we calculated the derivatives of:
[next]
the radial cloud position with respect to the radial texture coordinate,
[and]
and the angular cloud position with respect to the angular texture coordinate. Working in polar coordinates allowed us to find relatively simple analytical expressions for these derivatives.
[next]
We then reduced the cloud density when the maximum of these derivative was high.

Speaker notes

Here are clouds with density antialiasing turned off...

Speaker notes

... and here is what they look like with density antialiasing enabled. Note that the noisy sampling artifacts and pixelated edges all disappear.

- For Mie scattering, use Henyey-Greenstein phase function
  - Asymmetry g depends on how “deep” we are in cloud
  - Estimate using product of:
    - Transmittance to light
    - Transmittance of current raymarch segment
  - 0 $\Rightarrow$ Backscattering ($g \approx -0.15$)
  - 1 $\Rightarrow$ Forward Scattering ($g \approx 0.85$)
  - Gives “silver lining” for backlit clouds, dark edges for front-lit clouds
  - Scale backscattering by ~2.16 to account for multiple scattering
    - Based on simulation with albedo of 0.9

Speaker notes

The design of our phase function was very important in getting our clouds to look right. We used the Henyey-Greenstein phase function,
[next]
where the value of the asymmetry parameter g depends on how "deep" we are in the cloud. The idea is that forward-biased single scattering predominates in low-density, wispy parts of the cloud, while multiple scattering becomes more important as density increases, and also as the density in the direction of the light increases.
[next]
So, we use the product of
[next]
the transmittance to the light
[next]
and the transmittance of the current raymarch segment to lerp between
[next]
backscattering, when the product is 0,
[next]
and forward scattering, when the product is 1.
[next]
This naturally produces the "silver lining" present on backlit clouds, as well as the dark edges that Schneider described in his talk, without the use of his "powdered sugar" heuristic.
[next]
Because backscattering results from multiple scattering, we scaled the brightness of the backscattering phase function by 2.16.
[next]
This value was found by simulating a dense Mie scattering layer using our atmospheric multiple scattering model, with an albedo of 0.9. Note that the limit for this scale is 4, in the case of a uniform scattering layer with albedo of 1, which gives a white Lambertian surface.

Speaker notes

Here are the clouds using only forward scattering...

Speaker notes

... while here is what they look like using only backscattering.

Speaker notes

Combining forward and backscattering gives this result
[next]
with the dark edges of the clouds clearly visible here.

Speaker notes

Here is how the clouds look when backlit; note the silver lining effect on the top edge of the clouds.

- Wronski 2014, Drobot 2017a
- Froxel grid, 128 W x 64 H x 64 D
- Covers entire depth range (10 cm to 100 km)
- Exponential depth distribution: $\Delta z_{i+1} = 1.2\Delta z_i$
- Uses only analytic and integrable density functions
  1. Exponential height falloff: $d(z) = d_e e^{\frac{-z}{k_e}}$
  2. Sigmoidal height falloff:

$$d(z) = \text{lerp}\left(d_0, d_1, \frac{1}{2}\frac{z^2}{z^2 + k_s^2} + \frac{1}{2}\right)$$

- Local variation added with particles

Speaker notes

Ok, let's move on to our volumetric haze lighting and rendering. We built on the techniques described by Bart Wronski at SIGGRAPH 2014, and by Michal Drobot at Digital Dragons in 2017.
[next]
We used a low-resolution froxel grid, with a width of 128, and a height and depth of 64.
[next]
This covered the entire view frustum, from the near plane at 10 cm to the far plane at 100 km.
[next]
Depth slices grew exponentially, with each one being 20% thicker than the previous.
[next]
We restricted ourselves to a combination of two analytic and integrable density functions:
[next]
The first was a exponential height falloff,
[next]
while the second was a sigmoidal height falloff, with d_1 here being zero in almost all cases.
[next]
To add local variation in the other dimensions, we used particles, as I'll discuss shortly.

- Single compute dispatch populates froxel grid
  - Integrate inscattering front to back with quad swizzling using 4x4x4 threadgroups (Drobot 2017a)
  - Temporal filtering:
    - Shadow and ambient fraction (RG16f)
    - Local lighting (RGB11f)
  - Skip lighting of occluded froxels
    - Still compute shadow and ambient fraction to reduce reprojection errors
- Async compute (~0.5 ms on base PS4)

Speaker notes

The froxel grid is updated with a single compute dispatch;
[next]
it integrates inscattering from front to back with quad swizzling using 4x4x4 threadgroups.
[next]
We use temporal filtering to
[next]
smooth the jittered shadow and ambient occlusion samples,
[next]
as well as the jittered local light samples. Using a single Rgb11f buffer to filter the final light value resulted in objectionable banding; this arrangement lets use a different reprojection strategy for local lights,
[next]
and also lets us skip the lighting of occluded froxels. Both textures are filtered at full rate so the performance is about the same as sampling a single Rgba16f texture (before adding the early-outs).
[next]
We still update the shadow and ambient occlusion terms for occluded froxels, to reduce disocclusion artifacts.
[next]
All of this is performed in async compute in about 0.5 ms in a heavy scene with lots of local lights, on a base PS4.

Local Lighting

- Tiled light culling compute shader also outputs lights for haze
- Light divergence between threads in haze compute shader hurts performance
- Using flat bit arrays (Drobot 2017b) for haze lighting was a big speedup
  - For forward-shaded materials, faster to use sorted light indices

Speaker notes

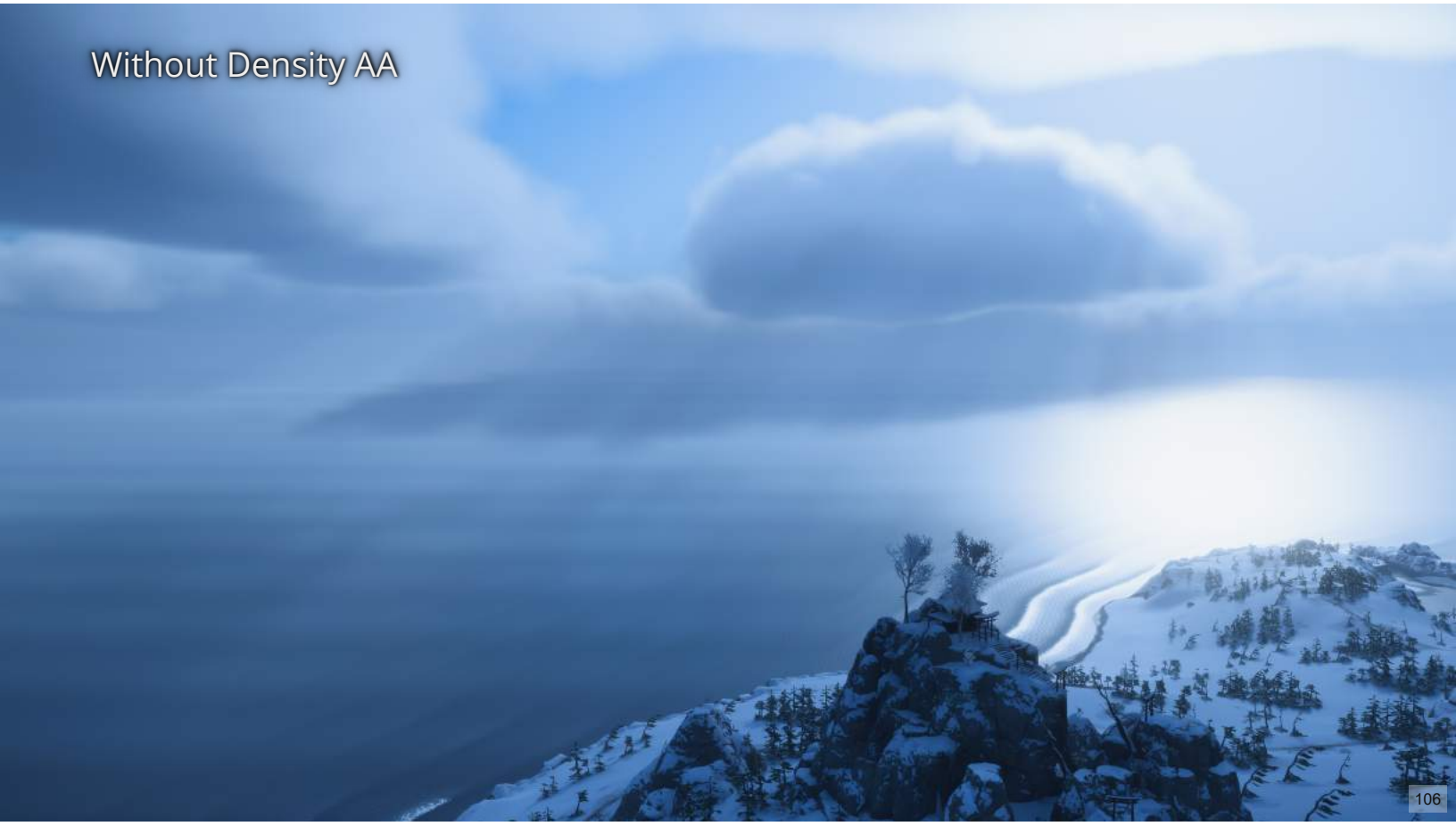
Our tiled light culling compute shader also outputs a set of per-tile lights for use by our volumetric haze.
[next]
Divergence in the set of lights between threads has a big performance impact,
[next]
so we adopted the flat bit array technique described by Michal Drobot in this course in 2017, which significantly sped up the shader, in some cases by nearly 2X.
[next]
As an interesting aside, for our forward-shaded materials, we found that it was a little bit faster to use a list of sorted light indices instead of bit arrays, since the overhead of iterating through the light list was a little bit lower than iterating through the bit arrays.

Anti-Aliasing

- Froxel volume has very low resolution but covers entire view frustum (10 cm to 100 km)
- Using traditional approach results in bad aliasing at horizon and with thin haze layers (density aliasing)
- Solve by storing inscattered radiance divided by opacity
- Re-apply opacity per-pixel when compositing into scene

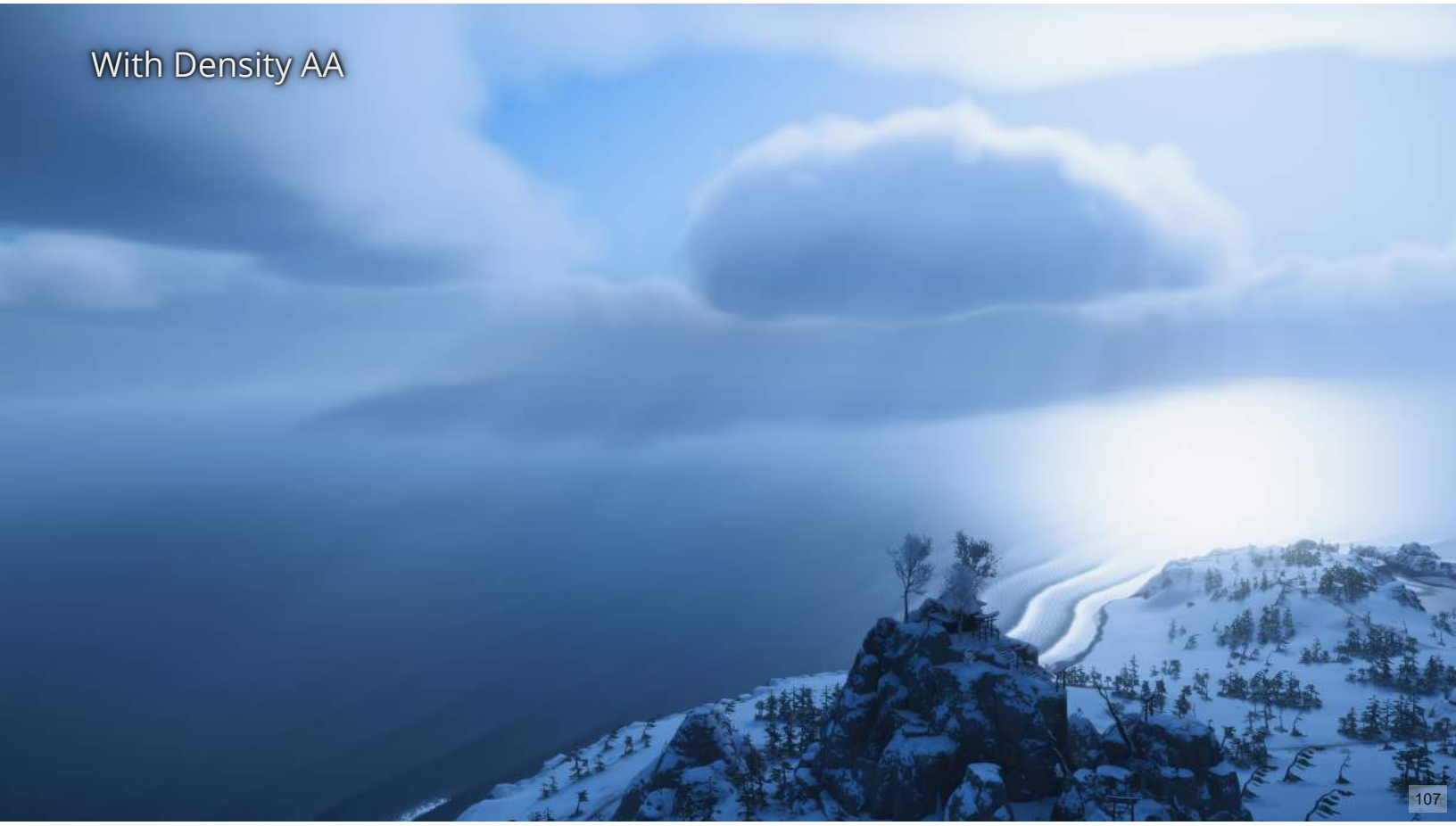
Speaker notes

Our froxel grid is very coarse, yet it covers the entire view frustum,
[next]
and using the traditional approach resulted in very bad aliasing at the horizon, and when using thin haze layers, due to the undersampled density changes.
[next]
We solved this by storing inscattered radiance divided by the opacity, or one minus the transmittance.
[next]
Because our density functions are analytic, we can recompute the transmittance per pixel, we then apply it when compositing into the scene. This also means we don't have to store opacity anywhere, so we're able to use an Rgb11f format for the volume.



Speaker notes

Here is a scene without density antialiasing; note the bands along the water parallel to the horizon.



Speaker notes

Applying density antialiasing significantly reduces these artifacts.

- Why does this work?
- First, assume that scattering $\sigma_s(x)$ and extinction $\sigma_e(x)$ are proportional, where c is the albedo:

$$\sigma_s(x) = c\sigma_e(x)$$

- Start with the volume rendering equation for the inscattered radiance over distance d:

$$L(d) = \int_0^d \sigma_s(x) \exp\left(-\int_0^x \sigma_e(t)\,\mathrm{d}t\right) L_i(x)\,\mathrm{d}x$$

- $L_i(x)$ is the integral over the sphere of the incoming radiance times the phase function.

Speaker notes

So that's a neat trick, but why does it work?
[next]
We'll start off with a fairly reasonable assumption that the scattering sigma_s is proportional to the extinction sigma_e, with the ratio being the albedo c.
[next]
Then consider the rendering equation for the inscattered radiance over distance d,
[next]
where L_i is the integral over the sphere of the incoming radiance times the phase function.

$$L(d) = \int_0^d \sigma_s(x) \exp\left(-\int_0^x \sigma_e(t)\,\mathrm{d}t\right) L_i(x)\,\mathrm{d}x$$

- Substitute for $\sigma_s(x)$:

$$L(d) = c\int_0^d \sigma_e(x) \exp\left(-\int_0^x \sigma_e(t)\,\mathrm{d}t\right) L_i(x)\,\mathrm{d}x$$

- Now assume that $L_i(x)$ is relatively constant. Then

$$L(d) \approx \widetilde{L}(d) = \frac{c}{d}\int_0^d L_i(x)\,\mathrm{d}x \int_0^d \sigma_e(x) \exp\left(-\int_0^x \sigma_e(t)\,\mathrm{d}t\right)\mathrm{d}x$$

Speaker notes

[next]
We then substitute for sigma_s, and pull the albedo c out in front of the integral.
[next]
Now, we'll assume that L_i is relatively constant, which in our case it generally is, except for shadowing. We can then separate the integral like so,

$$L(d) \approx \widetilde{L}(d) = \frac{c}{d} \int_0^d L_i(x)\,\mathrm{d}x \int_0^d \sigma_e(x) \exp\left(- \int_0^x \sigma_e(t)\,\mathrm{d}t \right) \mathrm{d}x$$

- Let $\overline{L_i}(d)$ be the average of $L_i(x)$ over $[0, d]$:

$$\widetilde{L}(d) = c\overline{L_i}(d) \int_0^d \sigma_e(x) \exp\left(- \int_0^x \sigma_e(t)\,\mathrm{d}t \right) \mathrm{d}x$$

- This integral simplifies to:

$$\widetilde{L}(d) = c\overline{L_i}(d) \left[1 - \exp\left(- \int_0^d \sigma_e(t)\,\mathrm{d}t \right) \right]$$

Speaker notes

[next]
and call the first integral L_i bar, since it's the average of L_i over the interval.
[next]
The remaining integral has simple analytic solution,

$$\widetilde{L}(d) = c\overline{L}_i(d)\left[1 - \exp\left(-\int_0^d \sigma_e(t)\,\mathrm{d}t\right)\right]$$

- The term in square brackets is just the opacity $\alpha(d)$:

$$\widetilde{L}(d) = c\overline{L}_i(d)\alpha(d)$$

- So, if we divide the inscattering $L(d)$ by $\alpha(d)$, we get:

$$\frac{L(d)}{\alpha(d)} \approx c\overline{L}_i(d) \sim \text{constant}$$

- Therefore we can expect that L/α is a smoother function than L.

Speaker notes

[next]
and the term in square brackets is just one minus the transmittance, or the opacity alpha
[next]
so if we divide by alpha, we get the albedo times the average of L_i as a function of d, which by assumption is constant.
[next]
So, to the extent that that assumption is true, L/alpha will be a smoother function than L.

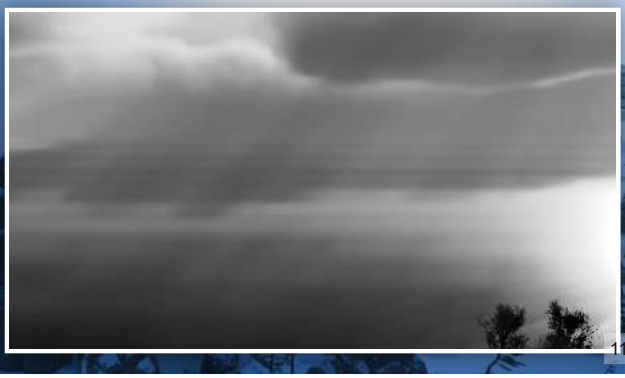
More Anti-Aliasing

- Use tricubic B-spline filtering to smooth and denoise
  - Relatively inexpensive on an Rgb11f buffer
    - Less than 50 us more than trilinear on PS4 Pro
  - See this Desmos graph (https://bit.ly/3wZ3aIs) for fast and accurate approximations of the weighting functions

Speaker notes

So that took care of aliasing due to density, but we aren't done. To reduce aliasing in all three dimensions, we sampled the volume using tricubic B-spline filtering, which requires 8 trilinear taps. This is surprisingly inexpensive -- it only cost 50 us more than trilinear sampling on a PS4 Pro at on a 3200x1800 checkerboard buffer.
[next]
To help optimize the ALU, we used an approximation for the tricubic weighting functions, which didn't affect the visual results at all; details are in the linked Desmos graph.

Trilinear Filtering



Speaker notes

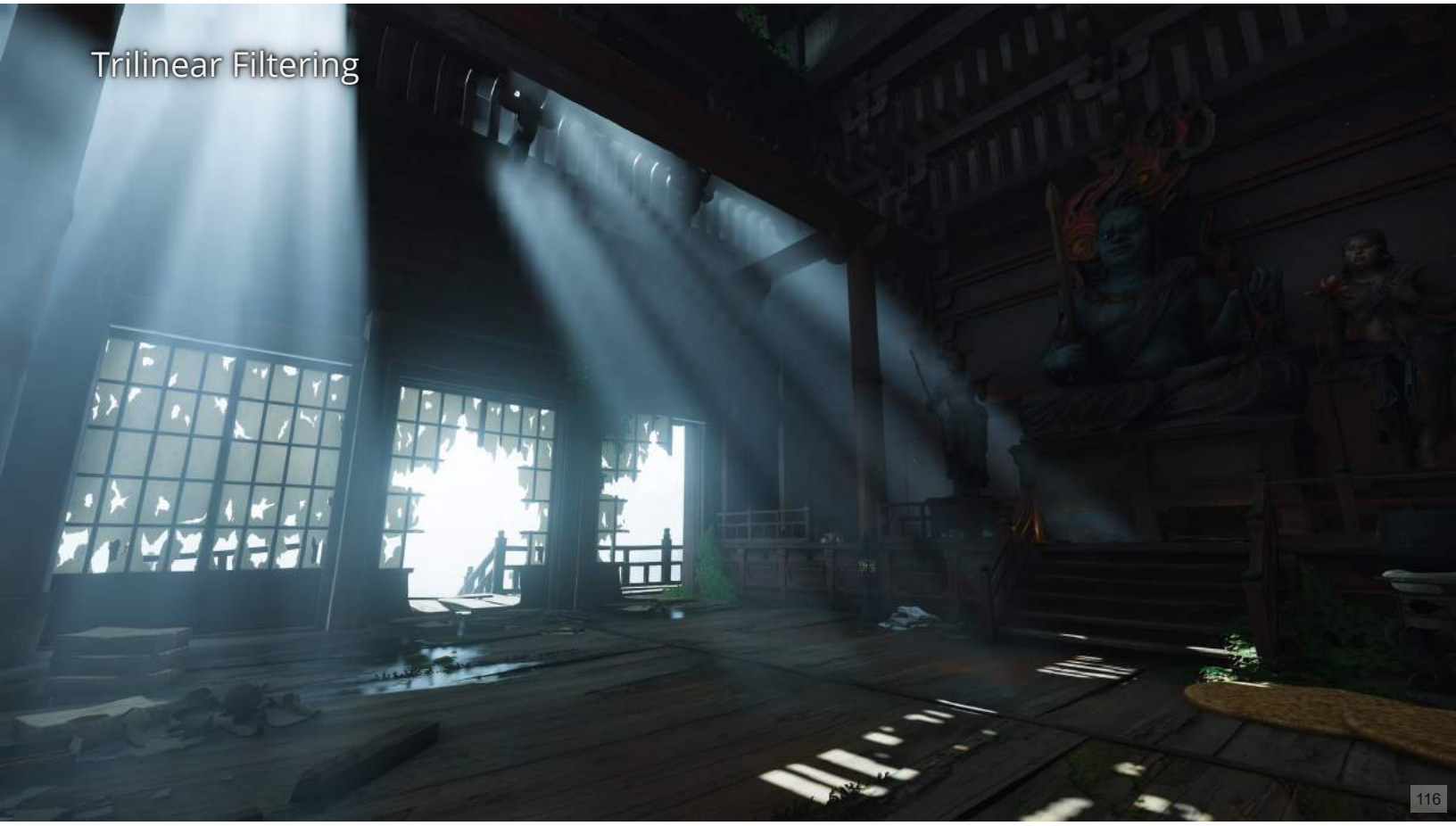
Here we're using trilinear filtering,
[next]
and here I've increased the contrast so you can see the aliasing more clearly.

Speaker notes

Bicubic filtering in x & y helps, but there's still some banding visible.

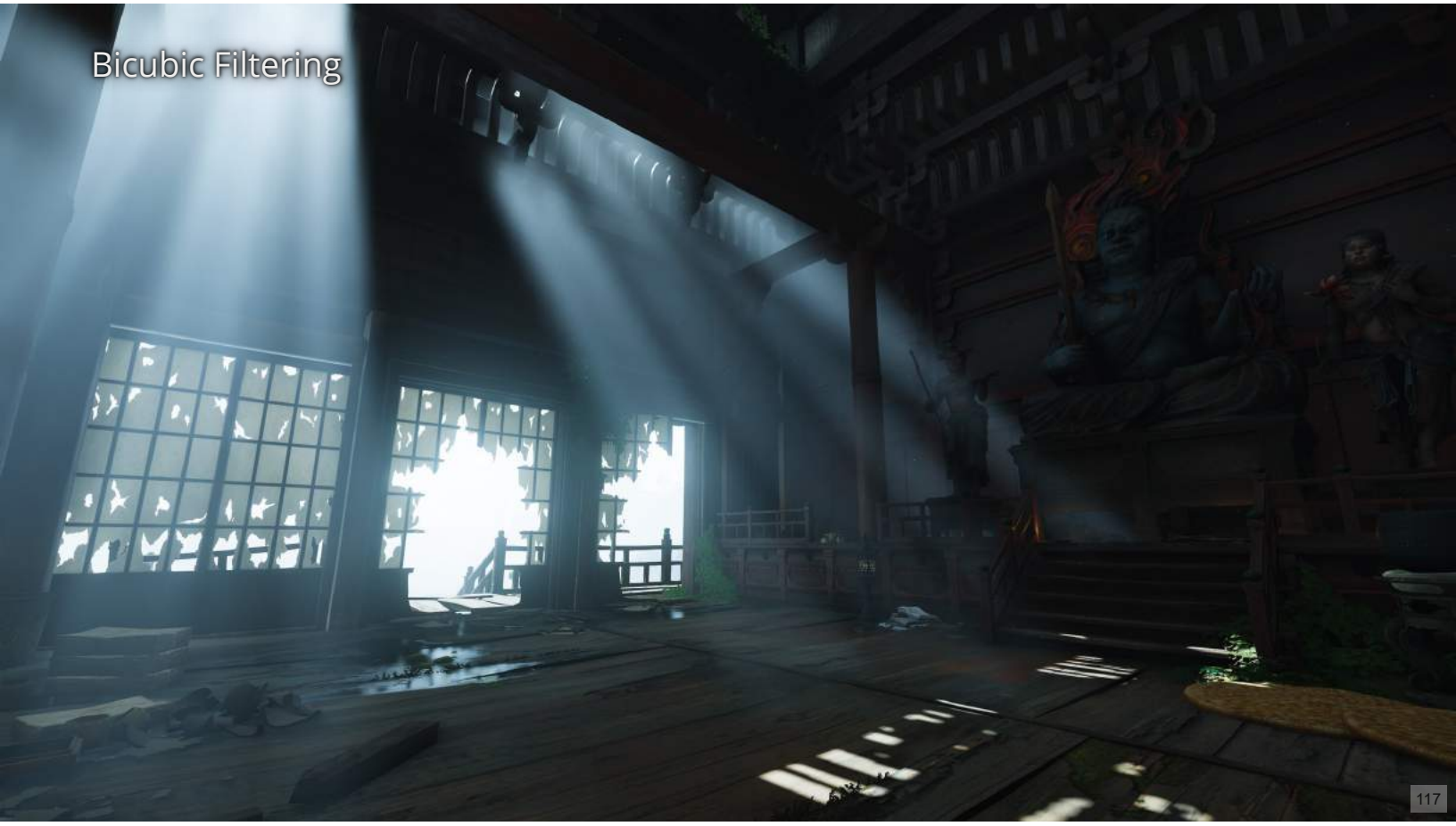
Speaker notes

Tricubic filtering almost completely eliminates the remaining aliasing.



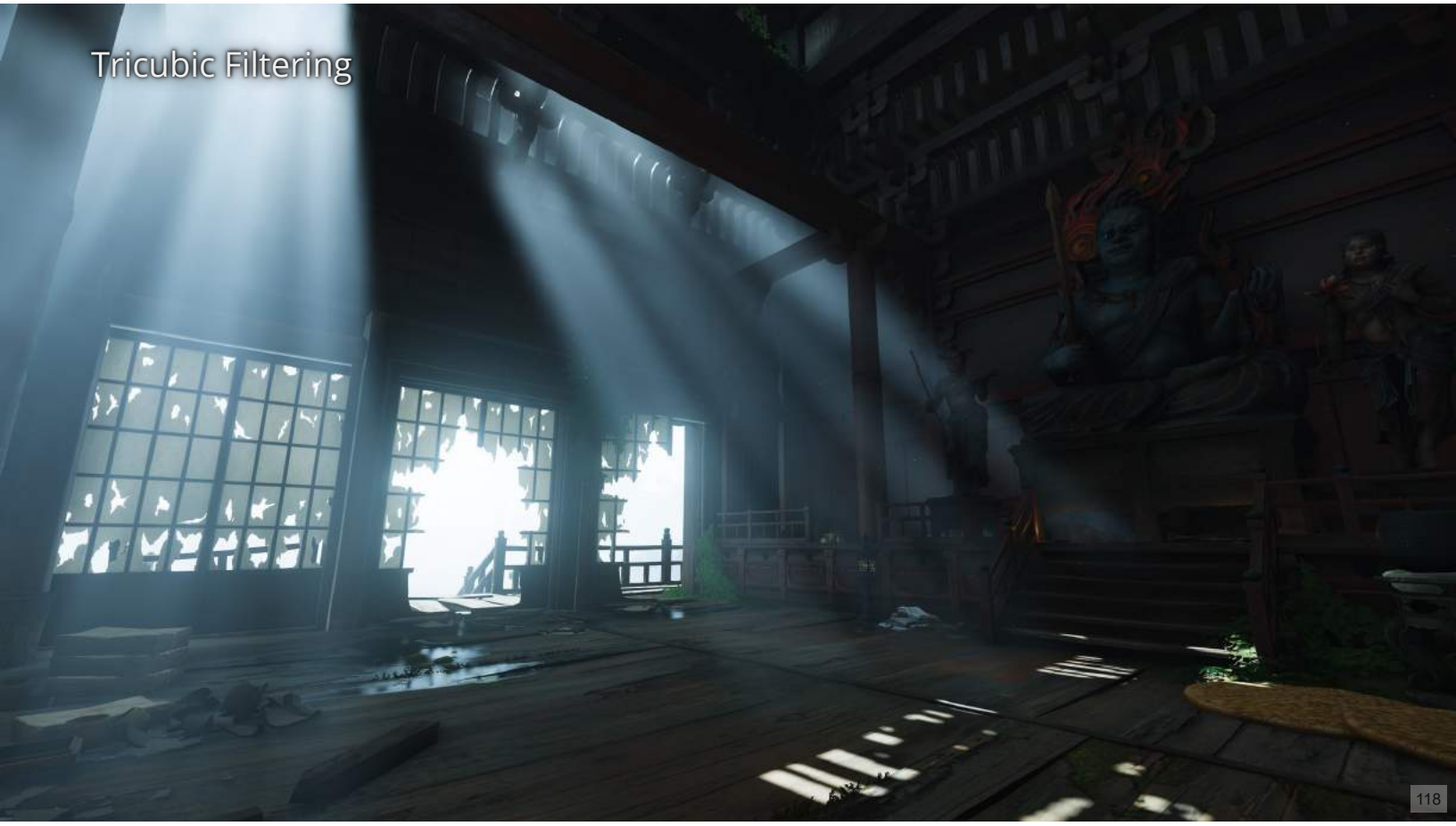
Speaker notes

Here's another example, first using trilinear filtering, with the aliasing in x & y being very apparent.



Speaker notes

Bicubic filtering fixes that...



Speaker notes

and tricubic filtering helps connect the light shafts to the openings.

Performance

Filter	Base PS4 (µs) (1080p)	PS4 Pro (µs) (1800cb)
Trilinear	363	430
Bicubic	430	443
Tricubic	667	483

Speaker notes

Note that 1800cb refers to 3200x1800 checkerboard resolution, where 1600x1800 pixels are rendered each frame.

- Optional “haze lighting” mode for particles
- In volumetric haze shader we also store $L_i(x)$
  - Sampled from LUTs, with phase functions, shadowing, and sky visibility fraction applied
  - Plus local light contribution
- Applied in particle shader by sampling froxel volume and multiplying $L_i(x)$ times opacity
  - Sample using bicubic filtering when opacity exceeds threshold.

Speaker notes

As I mentioned earlier, we used particles to add local fog. We did this by adding an optional "haze lighting" mode for particles.
[next]
In the volumetric haze shader we store L_i (from the earlier equations) in a parallel light volume;
[next]
this is the light sampled from the LUTs, times the phase functions, with shadowing and AO applied,
[next]
plus the local light contributions.
[next]
This is sampled in the particle shader by sampling the froxel light volume and multiplying by the particle opacity.
[next]
As an optimization, we use trilinear filtering until the opacity exceeds a threshold, at which point we switch to bicubic filtering. (Most of these particles are camera-facing so extra filtering in Z isn't necessary.)

Speaker notes

Here's a scene with particles lit using our standard particle lighting model...

Speaker notes

And here we've switched to the haze lighting model for the fog-like particles. This helps them fit better into the scene.

Multiple Scattering Approximation

- High-opacity particles are too dark when lit from the front
- Haze uses predominantly forward (Mie) scattering
- Inexpensive, approximate compensation:
  - On CPU, calculate ratio of a back-scattering phase function to haze Mie phase function for $\hat{\mathbf{v}} = \hat{\mathbf{l}}$ and $\hat{\mathbf{v}} = -\hat{\mathbf{l}}$
  - In shader, lerp between these two values based on $\hat{\mathbf{v}} \cdot \hat{\mathbf{l}}$ to get a multi-scattering scale factor $r_{\mathbf{MS}}$
  - Lerp from 1 to $r_{\mathbf{MS}}$ based on opacity, scale light
  - In practice, strength of effect was reduced for back scattering
    - Too bright when lacking Mie color contribution

Speaker notes

After adding particle haze lighting, our FX artists complained that the particles were too hard to see when lit from the front.
[next]
This was because haze assumes single scattering with the haze layer, and uses predominantly forward scattering.
[next]
To solve this problem, we added an inexpensive multi-scattering approximation term, which works as follows:
[next]
On the CPU, we calculate the ratio of a back-scattering phase function to the haze Mie phase function in the forward and back directions. We used the same Henyey-Greenstein phase function and asymmetry parameter of -0.15 to simulate multiple scattering, just like we did for clouds.
[next]
In the shader, we lerp between these two values based on the dot product between the view and light vectors; this gives us a multi-scattering scale factor r_MS.
[next]
We lerp from 1 to r_MS based on the opacity of particle, and use the resulting value to scale the particle luminance.
[next]
In practice, we reduced the strength of this effect for back-scattering especially,
[next]
since the particles appeared unnaturally lit when lacking the full Mie color contribution.

Speaker notes

Here are particles lit without the multi-scattering approximation...

Speaker notes

and here the multi-scattering approximation is on. As you can see, the particles gain more shape and definition.

Speaker notes

In the final section of this talk, I'll discuss some of the tonemapping techniques we used.

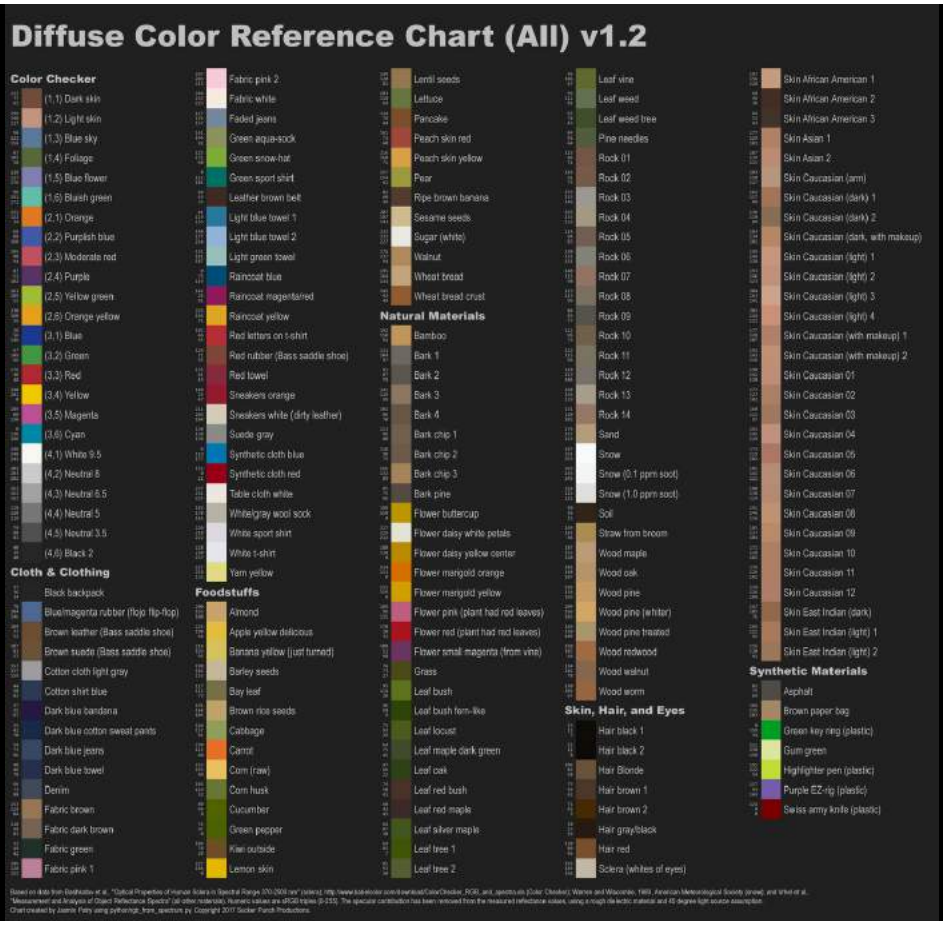
Tonemapping | Exposure

GHOST

- In the *Infamous* games, we struggled to maintain physically plausible diffuse albedo values
  - Exposure ⇔ Albedo feedback loop
  - Resulted in albedo values that were too dark
- Created a diffuse color reference chart

Speaker notes

First, some words about exposure. In Infamous: Second Son and First Light, we struggled to maintain physically plausible diffuse albedo values for our assets.
[next]
Our assumption was that the feedback loop between luminance-based exposure and albedo was largely responsible for this,
[next]
and resulted in diffuse albedo values that were too dark (and therefore the specular component was too bright in relation).
[next]
To help artists choose more physically-plausible albedo values, we created a diffuse color reference chart...



Speaker notes

which you can see here. It is based on the reflectance spectra of a variety of objects, converted to sRGB colors.

Tonemapping | Exposure

GHOST

- In the *Infamous* games, we struggled to maintain physically plausible diffuse albedo values
  - Exposure ⇔ Albedo feedback loop
  - Resulted in albedo values that were too dark
- Created a diffuse color reference chart
- Switched to a primarily illuminance-based exposure system
  - Use luminance only if highlight luminance exceeds threshold
    - E.g. Fire particles, bright specular highlights

Speaker notes

We also switched to a primarily illuminance-based exposure system, which takes albedo out of the equation altogether;
[next]
we only use luminance if the highlight luminance exceeds a threshold;
[next]
for example, because of fire particles or bright specular highlights.

Dealing with Dynamic Range

- Dim interior lighting
  - Large variation between interior and exterior luminance
  - Maintain visibility while avoiding “blow-outs”
- Skies drove most of our lighting
  - Brighter than environment
  - Wanted to avoid losing color
- See Bart Wronski's blog post for a discussion of this problem

Speaker notes

Because of the historical setting of the game, the interior lighting is generally very dim,
[next]
which results in a difference of up to 10 EV between the interiors and exteriors.
[next]
We wanted to maintain visibility as much as possible, which meant brightening parts of the scene that were too dark while avoiding blown-out lighting in the bright areas.
[next]
Since our skies drove our indirect lighting, it was often brighter than the environment,
[next]
but it was important to us artistically to prevent it from becoming too desaturated due to tonemapping.
[next]
A few months after we did this work, Bart Wronski wrote a blog post that addresses many of the same issues and suggests a similar solution, so I'll refer you to that for a more complete discussion.

Dealing with Dynamic Range

- In photography and cinematography:
  - Graduated filters
  - Special lighting rigs, bounce cards
  - Dodging & burning
  - Digital shadow/highlight adjustment
  - HDR photography using multiple exposures
- Human visual system capable of perceiving a very wide dynamic range
- Bilateral filter can be used to maintain image detail while reducing overall contrast

Speaker notes

But, I'll briefly go over some of the techniques used in photography and cinematography for dealing with these problems; they include the use of
[next]
graduated filters,
[next]
special lighting rigs and bounce cards,
[next]
dodging and burning during film development,
[next]
or these days, shadow and highlight adjustment,
[next]
as well as the use of multiple exposures in so-called HDR photography.
[next]
We also want to simulate dynamic range of the human eye, which is capable of perceiving a much higher dynamic range than a camera.
[next]
We knew that a bilateral filter could be used to maintain image detail while reducing overall contrast, so we decided to experiment with that to see if we could make it performant.

Speaker notes

Consider this scene no adjustments applied...

Speaker notes

If we reduce contrast by 50% we get this, which restores a lot of color but flattens the detail significantly. Note that this is an extreme adjustment for illustrative purposes only

Speaker notes

Using a bilateral filter we get this, which helps to retain a lot of the local contrast.

Tonemapping | Bilateral Grid

- Naïve bilateral filters are expensive
- The *Bilateral Grid* algorithm (Chen et al., 2007) allows efficient computation
  - Allows aggressive downsampling
    - We used 64 W x 32 H x 64 D (log luminance)
  - Naturally fits alongside luminance histogram generation
  - Overhead was ~250 us on PS4 at 1080p

Speaker notes

Naïve implementations of the bilateral filter are expensive,
[next]
but the Bilateral Grid algorithm allows them to be computed efficiently with a low memory footprint.
[next]
The algorithm allows the image data to be aggressively downsampled
[next]
(we used a volume of width 64, height 32 and depth 64)
[next]
and naturally fits alongside luminance histogram generation, used for dynamic exposure.
[next]
The overhead of the algorithm was approximately 250 us the PS4 at 1080p, so it wasn't too bad.

- Algorithm:
  - Populate grid with homogeneous log-luminance values $(w_i V_i, w_i)$
  - Initial weight for each sample based on linear Z weights
    - Each sample contributes to two nearest Z slices
    - Weighting in X & Y not necessary due to blurring
  - Apply Gaussian blur to grid
    - Wide blur in X & Y, smaller in Z (or can omit)
  - Bilaterally filtered value for a pixel obtained by trilinearly sampling grid and normalizing result

Speaker notes

Here's an overview of how it works:
[next]
First, we populate a volume (the "grid") with homogeneous log-luminance values, w_i V_i and w_i, where w_i is the weight and V_i is the log luminance.
[next]
The initial weights for each sample are chosen based on the linear Z axis weights;
[next]
that is, each sample contributes to the two nearest Z slices.
[next]
We can skip weighting in X & Y because we'll apply a wide blur in those axes.
[next]
So next, we apply a Gaussian blur to the grid,
[next]
using a wide blur in X & Y, and a smaller blur in Z. (Sometimes the Z blur is omitted altogether.)
[next]
Finally, the bilaterally filtered value for a pixel is obtained by trilinearly sampling the grid and normalizing the result.

Detail-Preserving Contrast Adjustment

$$I_o = c \times (B - M) + d \times (I_i - B) + M$$

- I_i: Input image log luminance
- I_o: Output log luminance
- B: Bilaterally filtered log luminance
- M: Midpoint log luminance
- c: Contrast scale
- d: Detail strength

Speaker notes

Here's the basic formula for detail-preserving contrast adjustment.
[next]
I_i is the input image's log luminance;
[next]
I_o is the output log luminance;
[next]
B is the bilaterally filtered log luminance;
[next]
M is the midpoint log luminance (that is, it's the point that the luminance values will be drawn towards as we reduce contrast);
[next]
c is the constrast scale;
[next]
and d is the detail strength.

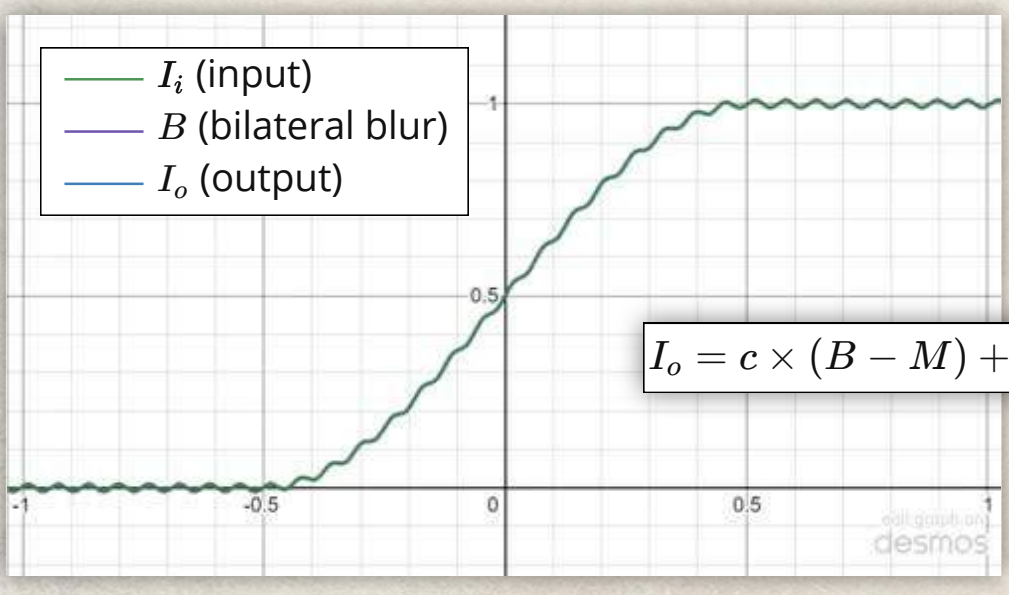
Ringing and Haloing Artifacts

- Bilaterally blurring smooth gradients can result in ringing
  - E.g., clouds, filtered shadows, specular highlights
- Can be reduced with larger luminance buckets (or wider Z blur)
  - Tends to increase halos

Speaker notes

This works fairly well, although we did encounter ringing artifacts when applied to scenes with smooth gradients.
[next]
This most often occurred on clouds, soft shadows, and specular highlights, as you can see
[next]
here.
[next]
These artifacts can be reduced by using larger luminance buckets (or a wider Z blur), but the tradeoff is increased haloing.

Simulated Ringing Scenario



$$I_o = c \times (B - M) + d \times (I_i - B) + M$$

Speaker notes

This animation demonstrates how ringing can occur. We start with the green input signal and bilaterally blur it to get the purple curve. We then produce the blue output signal
[next]
by reducing the contrast of the blurred curve,
[next]
and adding back the detail, which is the difference between the input and blurred signals.

(Link to graph: https://www.desmos.com/calculator/tieypyafig)

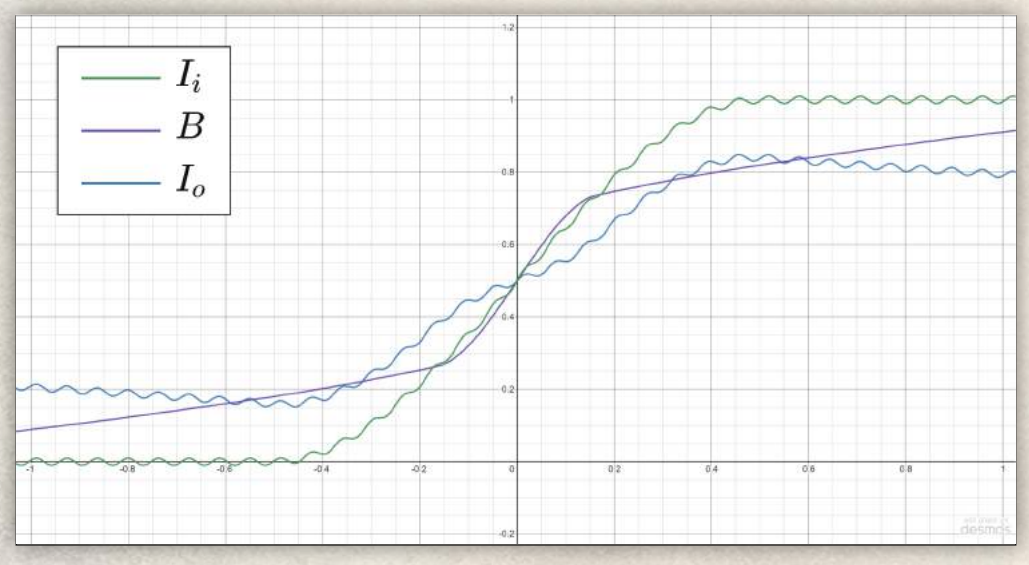
Eliminating Ringing Artifacts

- Blend between bilateral filter and very wide 2D Gaussian
  - We used 40% bilateral, 60% 2D Gaussian
- 2D Gaussian blur must be wide enough to avoid noticeable haloing
  - We used a radius 13 Gaussian blur on 64x32 image
  - Upsampled to 256x128 using bicubic B-spline filtering
    - Avoids bilinear artifacts
    - Adds extra blur

Speaker notes

To eliminate these artifacts, we adopted a hybrid approach by blending between the output of the bilateral filter and a very wide 2D Gaussian blur.
[next]
In the end we used a weighting of 40% for the bilateral blur and 60% for the Gaussian blur.
[next]
It's important that the 2D Gaussian be wide enough to avoid noticeable haloing;
[next]
we used a filter with a radius of 13 taps, which was used on a 64x32 image.
[next]
We then upsampled this to 256x128 using bicubic B-spline filtering filtering,
[next]
which avoided bilinear filtering artifacts,
[next]
and added even more blur.

Eliminating Ringing Artifacts



Speaker notes

This animation shows a lerp between a bilateral filter and a wide Gaussian filter; note how the ringing artifacts are reduced by mixing in more of the Gaussian blur.

(Link to graph: https://www.desmos.com/calculator/4afvbmckcv)

Speaker notes

This is the same image we saw before, with a 50% contrast reduction using the bilateral filter only.

Speaker notes

Here is the same image after mixing in the Gaussian blur. The appearance is more natural overall, while still retaining color in the skies.

Speaker notes

For comparison, here is the original image once again.

Speaker notes

And here is the scene using typical in-game settings for this time of day and weather state.

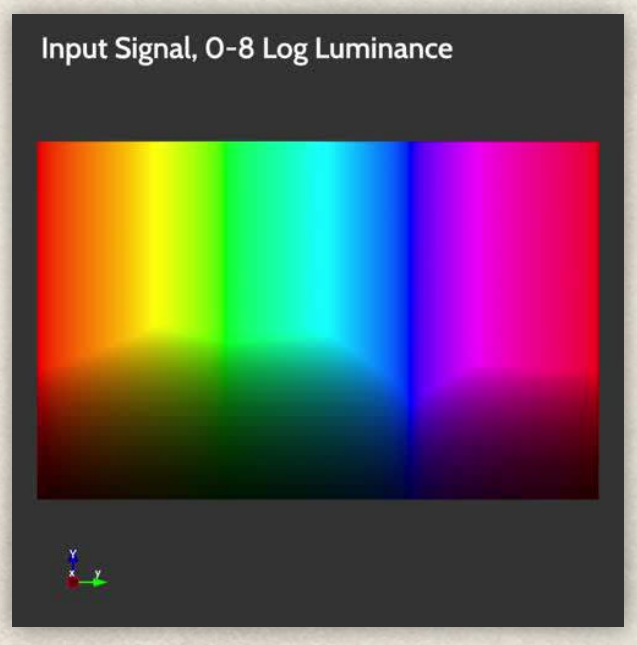
Speaker notes

I haven't talked much about the the
[next]
'd' parameter in this equation, but it's possible to adjust local contrast by varying it. Here I've set it to 1.5, which makes the scene a bit punchier. We used this sparingly, but it can be a useful tool to have.

- Per-channel tonemapping in the rendering color space clips saturated colors to cyan, magenta, and yellow
- Here we're just applying the per-channel "Reinhard" operator:

$$\mathbf{c}_i^{\text{out}} = \frac{\mathbf{c}_i^{\text{in}}}{1 + \mathbf{c}_i^{\text{in}}}$$

- Color scaled so no component exceeds 1.0



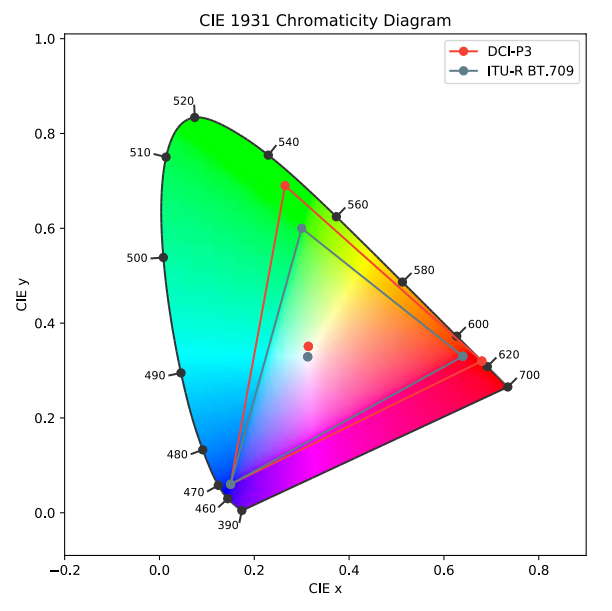
Speaker notes

It's well-known that per-channel tonemapping in the rendering color space clips saturated non-primary colors to cyan, magenta and yellow.
[next]
Here we're just applying the Reinhard operator per channel, to keep things simple.
[next]
Note that we've also scaled the input colors so that no component exceeds one.

Speaker notes

In practice, tonemapping this way gives pretty ugly results -- notice the saturated yellow at the sunset and specular highlight.

- Choose another color space?
  - Tried several:
    - ACES2065-1 (AP0)
    - ACEScg (AP1)
    - Rec.2020
    - DCI-P3



Speaker notes

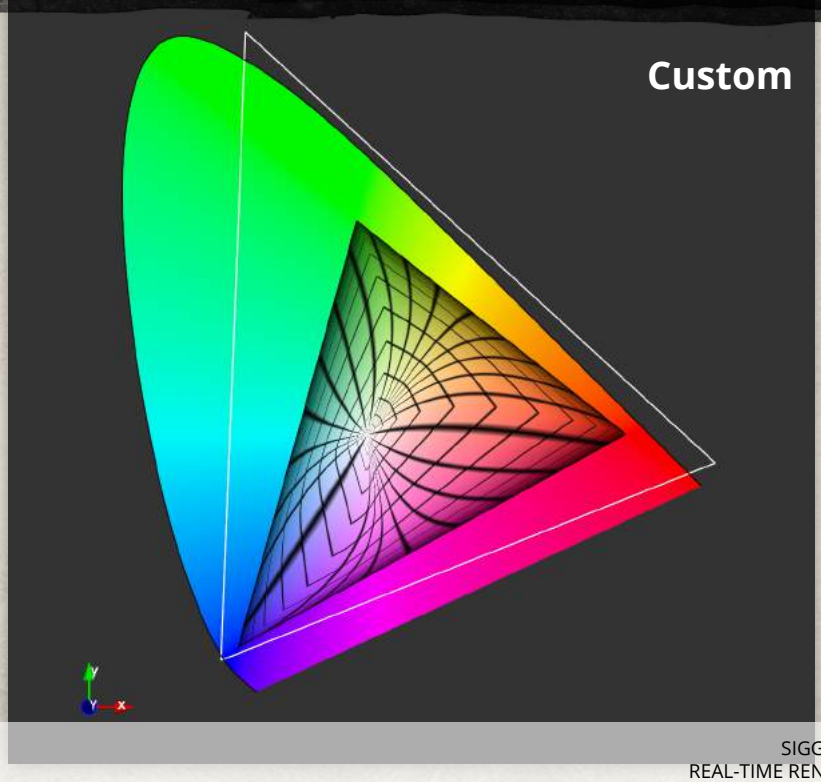
During a conversation in 2017, Brian Karis mentioned that he liked the roll-off-to-white behavior of ACEScg as a tonemapping space.
[next]
So we decided to that, along with several other color spaces, including
[next]
ACES AP0,
[next]
ACEScg (or AP1),
[next]
Rec.2020,
[next]
and DCI-P3.

- Choose another color space?
  - Tried several:
    - ACES2065-1 (AP0)
    - ACEScg (AP1)
    - Rec.2020
    - DCI-P3
  - ACEScg was the best of these, but still bent reds too much towards yellow
  - Adjusted red primary x coordinate (0.713 $\Rightarrow$ 0.75)

Speaker notes

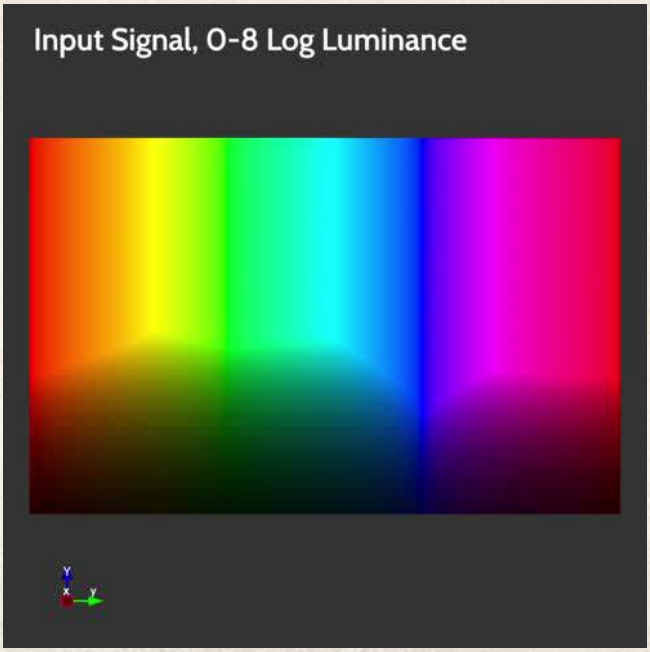
ACEScg was judged to be the best out of all of these, but we still felt that it bent reds too much towards yellow during tonemapping.
[next]
So we adjusted the x coordinate of the red primary from 0.713 to 0.75, and that gave us results we were all happy with.

Tonemapping| Color Space



Speaker notes

Here
[next]
you can see how ACEScg bends the lines of red and orange hue in input space towards yellow.
[next]
By adjusting the red primary, this shift is reduced.

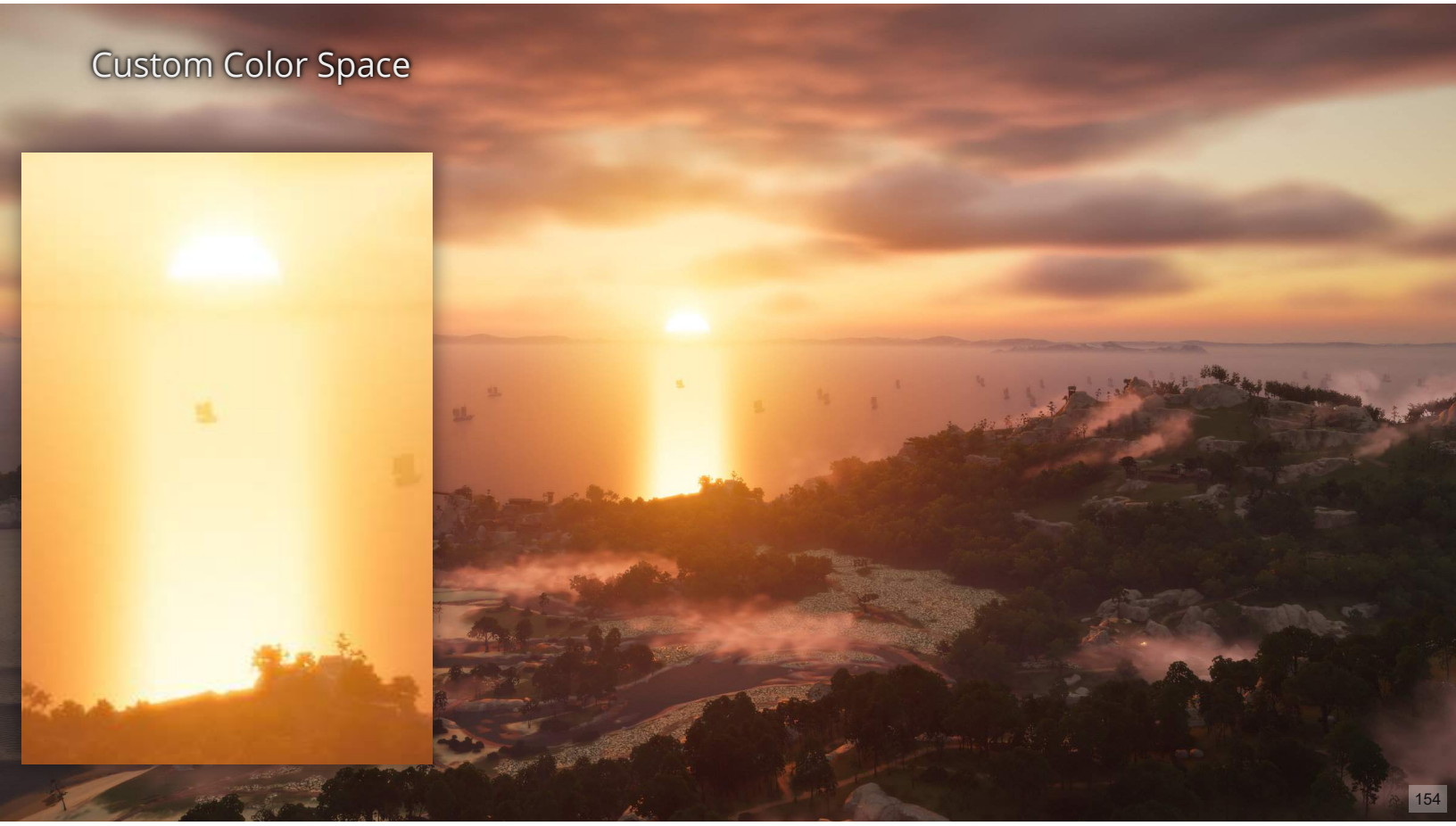


Speaker notes

This animation shows how the color space is transformed by tonemapping in our custom color space. Note that there's likely still room for improvement; note how colors near the blue primary are shifted fairly strongly towards magenta.

Speaker notes

This scene is using ACEScg for tonemapping.
[next]
Note the specular highlight in particular; our art director really disliked how sunsets tended towards this color when using ACEScg.



Speaker notes

Using the custom color space reduced the color shift and saturation just enough here, without adversely affecting the tonemapping of yellows, like in the golden forest.

Tonemapping | White Balance

- Working with old-school grading tools
  - Photoshop + 32x32x32 sRGB LUTs
  - Still favored by Art Director
- Needed to be conservative to avoid introducing banding
  - Too limiting in many cases
- Before tossing our existing pipeline and LUTs, we tried adding white balance controls
  - Free, since we can roll it into the Rec.709 ⇒ tonemap matrix
  - Worked so well that we didn't need to change our grading approach

Speaker notes

On Ghost we still used our fairly old-school color grading tools,
[next]
consisting of manipulating screenshots with an embedded 32x32x32 sRGB color LUT in Photoshop.
[next]
Our art director liked this workflow, so we weren't eager to change.
[next]
However, due to the limited resolution, we had to use it conservatively to avoid introducing banding artifacts,
[next]
which was too limiting in many cases.
[next]
I'd read a blog post by Bart Wronski discussing some ideas about white balance and games, and the idea of introducing a white balance control into our grading pipeline made a lot of sense, so we gave it a try.
[next]
It was free in terms of GPU performance, since we could roll it into the Rec.709 to tonemap color space transformation matrix.
[next]
Fortunately, it worked so well that we didn't end up needing to change our grading toolset.

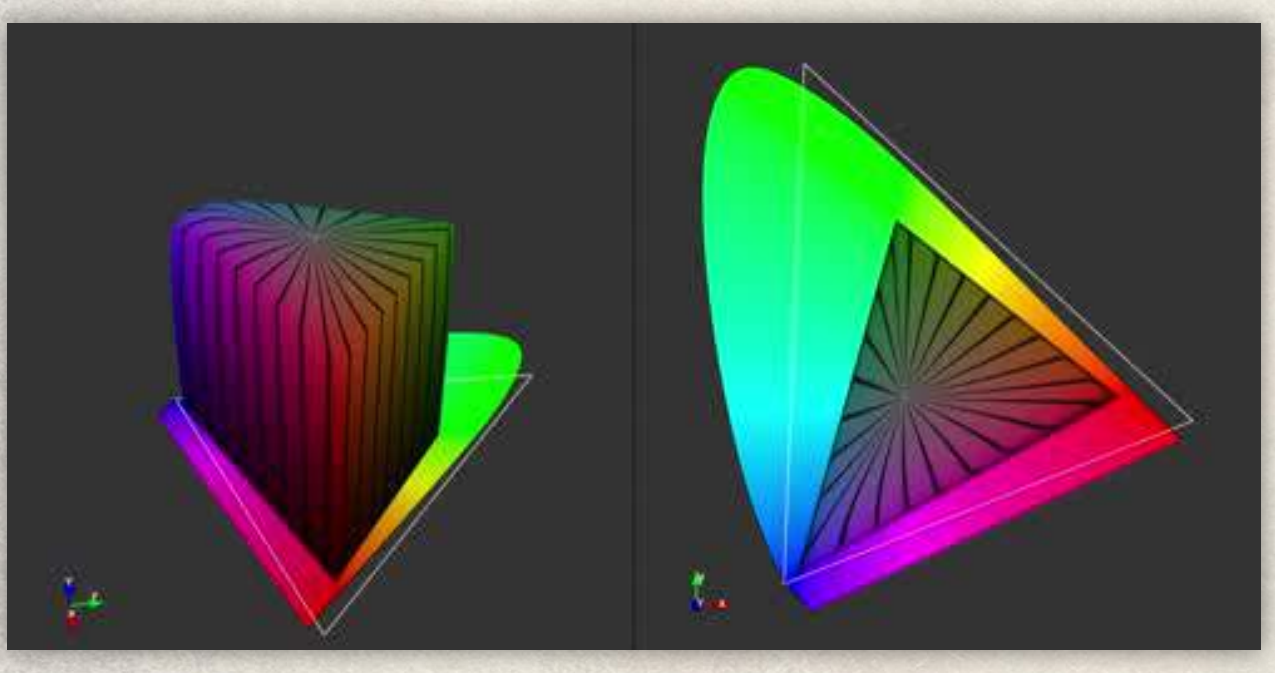
Tonemapping | White Balance

- Artist chose the color they wanted white to change **to**
  - Artists found this more intuitive than typical photographic white balance controls
- Implemented using chromatic adaptation matrix
  - Von Kries method using Bradford LMS space
- Does interesting things to the color space!

Speaker notes

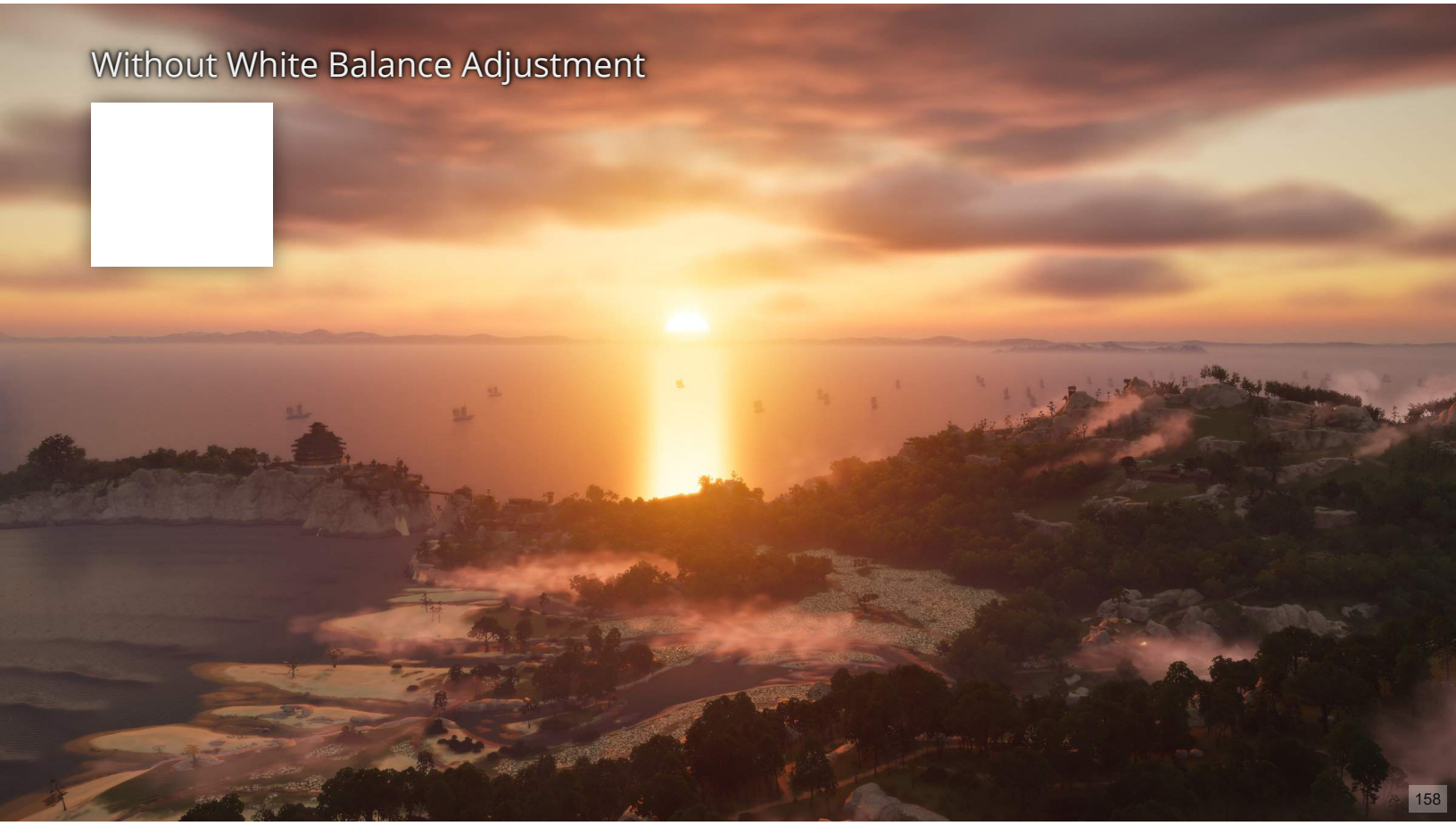
The interface we provided was a little different than traditional white balance tools. Rather than providing color temperature and hue sliders, we let the artist choose the color they wanted white to change *to*.
[next]
This is backwards from a tool like Lightroom, which lets you click on a pixel that should be neutral (like a gray card). So in Lightroom you pick the color that should change *to white*. But the way we did it, if you want the scene to be bluer, you pick a blue color, which the artists found more intuitive.
[next]
The transformation we used is called a chromatic adaptation matrix, and models how the human visual system adapts to different lighting environments.
[next]
Specifically, we used the Von Kries transform in Bradford LMS color space. This transforms colors to a color space that approximates the responses of the long, medium, and short cones in the retina, scales them by the ratio of the white points in this space, and then transforms them back.
[next]
This does interesting things to the color space...
[next]

Tonemapping| White Balance



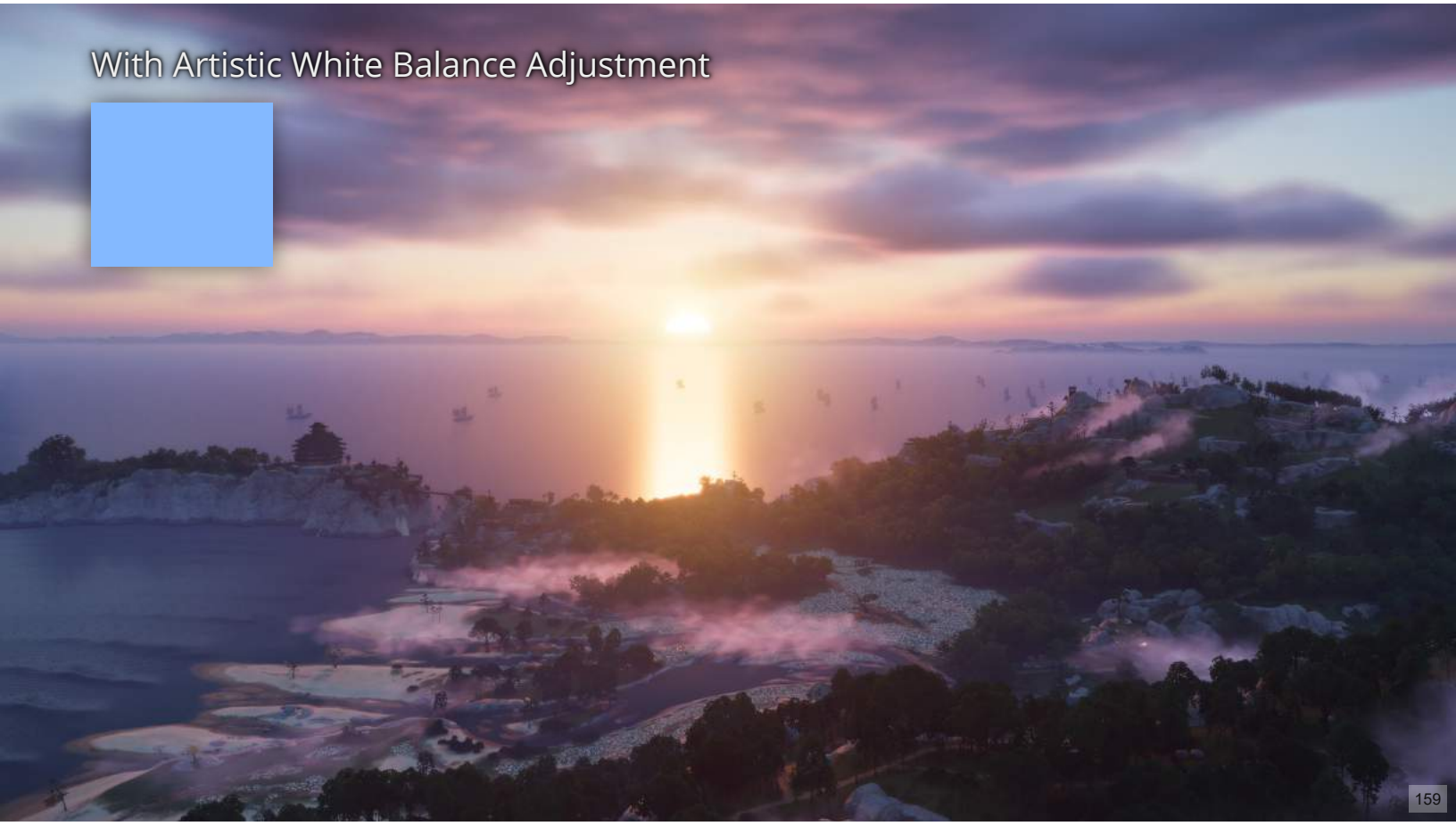
Speaker notes

This animation shows the color space being transformed to a light blue white point. Because our final color space after tonemapping is still Rec.709, bright colors still roll off to display white, as you can see when the outer layers of the color space are "peeled away" to show only low-saturation colors.



Speaker notes

Here is the scene without white balance adjustment...



Speaker notes

... and this is the scene after applying artistic white balance adjustment, where white was shifted to the blue in the square in the top-left.

Tonemapping | Purkinje Shift

- Two challenges when rendering night:
  - Make night feel like night, and not just darker day
  - Increase visibility in dark areas
- Research has been done in this area
  - Dingcai Cao et al., “Rod Contributions to Color Perception: Linear with Rod Contrast”, 2008.
  - Adam G. Kirk, “Tone Mapping for Low-Light Conditions”, 2011
  - Based on measurements done on the human visual system

Speaker notes

I'll now talk about one of the techniques we used to overcome the challenges we faced when trying to render nighttime scenes.
[next]
The first challenge was to make night *feel* like night, and not just like a darker version of daytime.
[next]
The second challenge was to make sure that the player was able to see clearly enough at night, so that they're able to play the game.
[next]
We came across some research in this area,
[next]
described in these two papers. The Kirk paper is based on the Cao et al. paper, but we found that following the equations and constants in the Cao paper gave better results, and I'll mostly use their symbols here.
[next]
The research is based on models and measurements of the human visual system, so I'll briefly go into some of the background.

Human Visual System

- Receptors:
  - Long, medium, short **cones** (normal color vision)
  - **Rods** (low-light vision, monochromatic)
- Vision is 3D, not 4D
  - Rods use same neural pathways as cones
- Night vision is not “black & white”
  - Color shift introduced by rods as they take over
    ⇒ *Purkinje Shift*

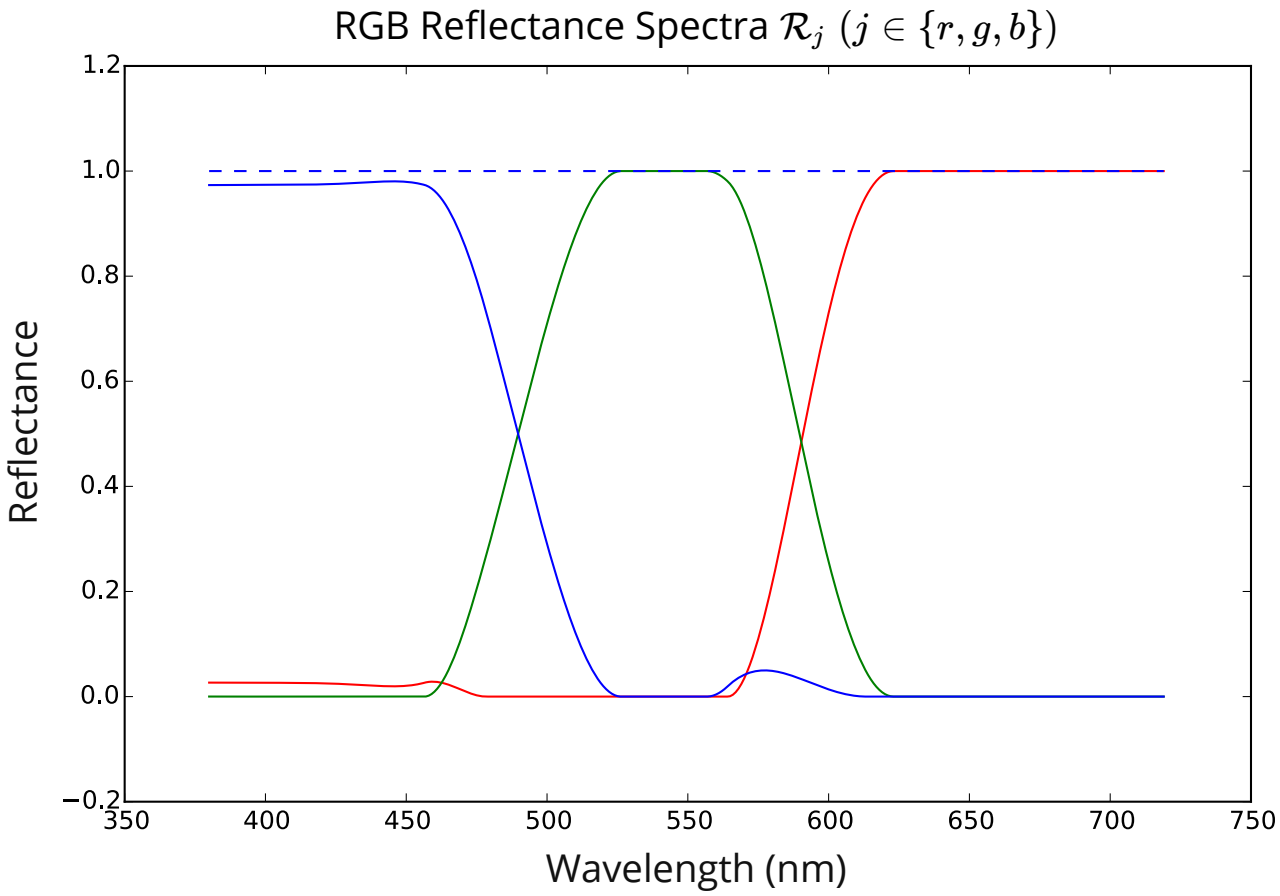
Speaker notes

Our retinas have four types of receptors:
[next]
three types of cones (long, medium, and short), which are responsible for regular color vision,
[next]
and rods, which are more sensitive to low light levels, and are monochromatic.
[next]
However, human vision is 3D, not 4D (in most people, at least),
[next]
which is because the rods use the same neural pathways as the cones.
[next]
The result is that low-light vision is not "black and white",
[next]
but there's a color shift introduced by the rods as they take over from the cones, which is sometimes called the Purkinje shift.

- The Purkinje shift can be modeled as a delta in *opponent color space*
  - Color space believed to be used by the visual system
- Need a way to convert between HDR scene colors and cone/rod excitations
  - Cone and rod wavelength response curves are available
  - So need scene RGB ⇒ spectrum
  - Brian Smits, “An RGB to Spectrum Conversion for Reflectances”, 2000

Speaker notes

This color change can be modeled as a delta in opponent color space,
[next]
which is the color space believed to be used within the visual system.
[next]
To use this technique, we need a way to convert between HDR scene colors and cone/rod excitations,
[next]
and while we could try using an existing LMS transform such as the previously-mentioned Bradford space, we'd still need a way to compute rod values.
Cone and rod wavelength response curves are available from Kirk and others,
[next]
so we need a way to convert scene RGB colors into a frequency spectrum.
[next]
This is an underspecified problem; we used the approach described by Smits in this paper, which finds a set of RGB basis spectra that maximize smoothness.



Speaker notes

Here are the reflectance spectra that we found using Smit's method; we constrained the RGB curves so they were between 0 and 1, and summed to 1.

- Calculate matrix from RGB to LMSR (using D65 illuminant)

$$\mathbf{M}_{i,j} = \int_{\Lambda} \mathcal{E}_i(\lambda)\mathcal{I}(\lambda)\mathcal{R}_j(\lambda)\,\mathrm{d}\lambda$$

- $\mathbf{M}$: Matrix to convert from RGB to LMSR
- $i \in \{L, M, S, R\}$
- $j \in \{r, g, b\}$
- λ, Λ: Wavelength & range of visible wavelengths
- $\mathcal{E}_i(\lambda)$: Receptor response curve
- $\mathcal{I}(\lambda)$: CIE D65 illuminant spectrum (Rec.709 white point)
- $\mathcal{R}_j(\lambda)$: Rec.709 reflectance spectrum

Speaker notes

First, we need to calculate a matrix to convert from Rec.709 RGB to LMSR space, using the Rec.709 D65 illuminant.
[next]
M is the conversion matrix;
[next]
i is the row index corresponding to the LMSR responses;
[next]
j is the column index corresponding to the red, green, and blue primaries;
[next]
lambda is the wavelength, and capital lambda is the range of visible wavelengths that we integrate over;
[next]
E_i is the response curve for receptor i;
[next]
I is the D65 illuminant spectrum;
[next]
and R_j is the reflectance spectrum for red, green or blue found using Smit's method.

- Convert from scene color $\mathbf{c}$ to LMSR response $\mathbf{q}$:

$$\mathbf{q} = \mathbf{M}\mathbf{c}$$

- Calculate *multiplicative gain control* $\mathbf{g}$:

$$\mathbf{g}_i = \left[1 + \frac{0.33}{\mathbf{m}_i}\left(\mathbf{q}_i + \mathbf{k}_i \mathbf{q}_R\right)\right]^{-\frac{1}{2}}$$

- $i \in \{L, M, S\}$
- $\mathbf{m}_i$: maximal cone sensitivity ($\mathbf{m} = [0.63721, 0.39242, 1.6064]$)
- $\mathbf{k}_i$: rod input strength with $\mathbf{k}_L = \mathbf{k}_M$
  - We used $\mathbf{k}_L = \mathbf{k}_M = 0.2, \mathbf{k}_S = 0.29$

Speaker notes

We then use this matrix to convert from the scene color c to the LMSR response q,
[next]
and then calculate what's known as the multiplicative gain control vector g:
[next]
which is only defined for the cones,
[next]
and where m is a constant vector describing cone sensitivity, given by this value,
[next]
and k is a vector describing rod input strength for the LMS pathways, with k_L and k_M restricted to be equal.
[next]
According to Cao, k decreases with increasing illuminance level, but we chose to follow Kirk and use constant values, given here. Note that there is some freedom in choosing these values, and experimenting with different values gives different results, so if you're implementing this you may want to try different values here.

- For a color $\hat{\mathbf{q}}$ in LMS space, the opponent space color $\mathbf{o}$ is given by:

$$\mathbf{o} = \mathbf{A}\hat{\mathbf{q}}$$

$$\mathbf{A} = \begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Speaker notes

If we let q-hat be a color in LMS space (dropping the R component), then the opponent space color is given by this equation,
[next]
with the matrix A given here.

- We can now calculate the incremental effect $\Delta\mathbf{o}$ that the rods have in opponent color space:

$$\Delta\mathbf{o} = \frac{K}{S}\begin{bmatrix} -(k_3 + r_w) & (1 + k_3 r_w) & 0 \\ pk_3 & (1-p)k_3 & 1 \\ pS & (1-p)S & 0 \end{bmatrix} \operatorname{diag}(\mathbf{k}) \operatorname{diag}(\mathbf{m})^{-1} \mathbf{g}\mathbf{q}_R$$

- K: Scaling constant ($K = 45.0$)
- S: Static saturation ($S = 10.0$)
- k_3: Surround strength of opponent signal ($k_3 = 0.6$)
- r_w: Ratio of responses for white light ($r_w = 0.139$)
- p: Relative weight of L cones ($p = 0.6189$)

Speaker notes

We're almost done; we can now calculate the incremental effect delta-o that the rods have in opponent color space,
[next]
given by this equation,
[next]
where the new terms are all constants, given here.

- The RGB color change is then given by:

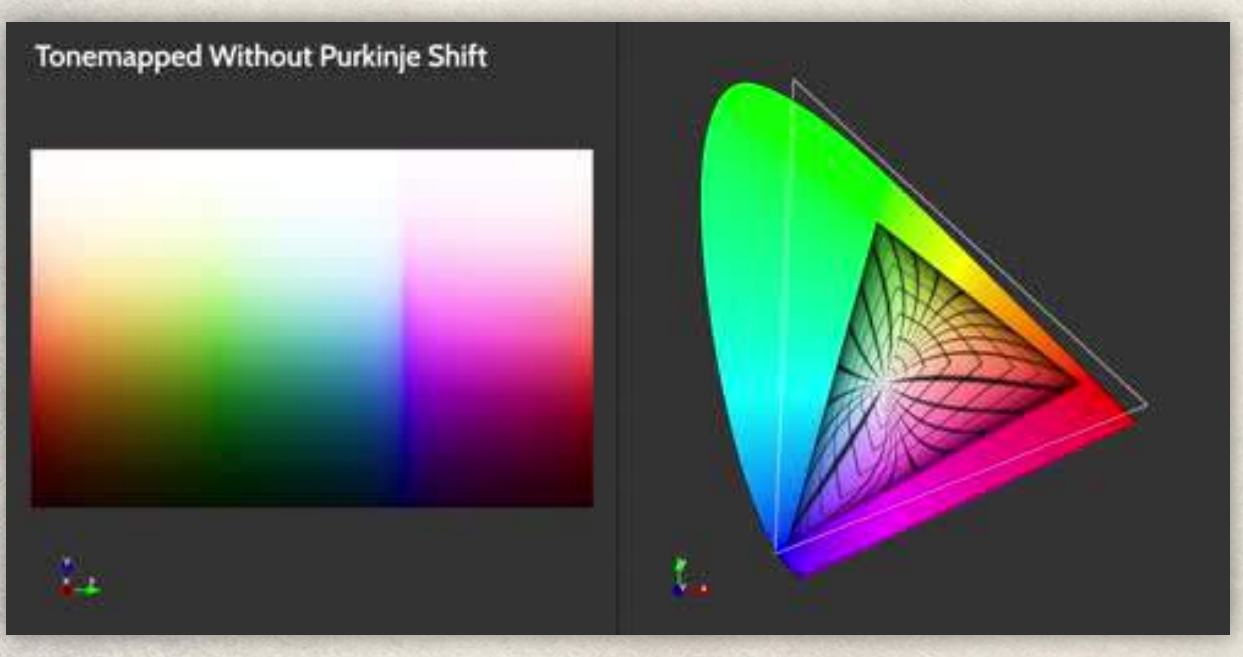
$$\Delta \mathbf{c} = \hat{\mathbf{M}}^{-1} \mathbf{A}^{-1} \Delta \mathbf{o}$$

- $\hat{\mathbf{M}}$: LMS 3x3 submatrix of $\mathbf{M}$
- Shader code is straightforward:

```
1 float3 PurkineShift(
2     float3 rgbLightHdr,
3     float4x3 matLmsrFromRgb,
4     float3x3 matRgbFromLmsGain)
5 {
6     float4 lmsr = mul(matLmsrFromRgb, rgbLightHdr);
7     float3 lmsGain = rsqrt(1.0f + lmsr.xyz);
8     return rgbLightHdr + mul(matRgbFromLmsGain, lmsGain) * lmsr.w;
9 }
```

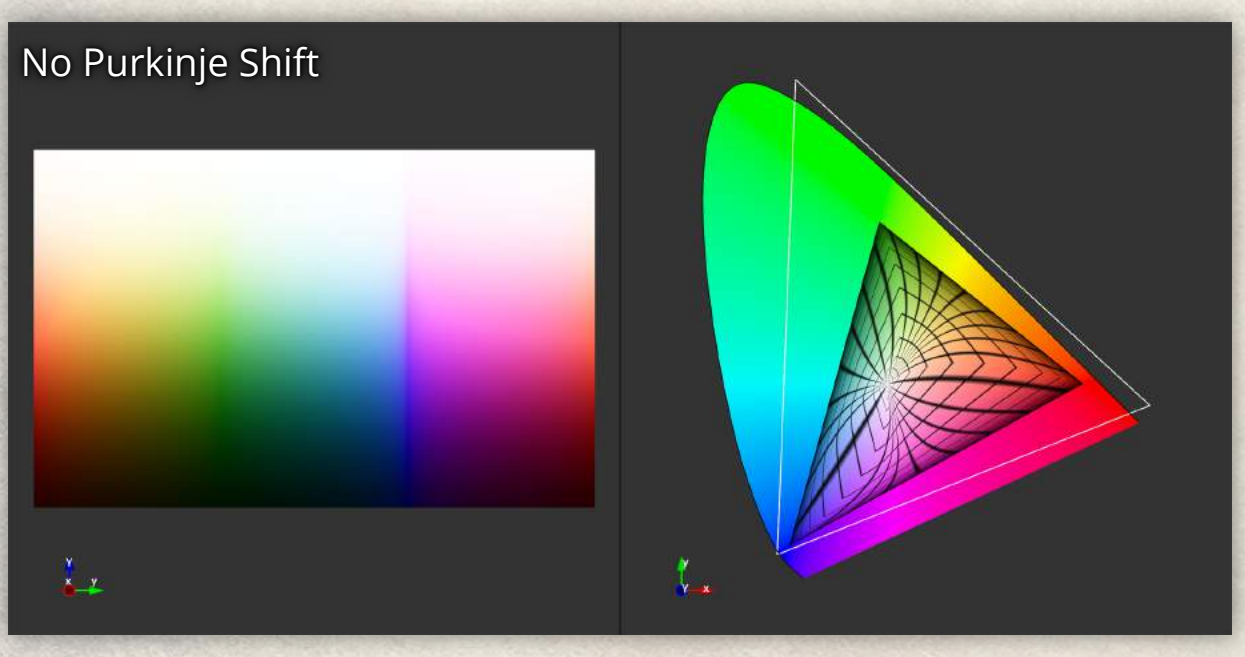

Speaker notes

Finally, we can compute the RGB color change using this formula,
[next]
where M-hat is the top-left 3x3 submatrix of M.
[next]
The shader code for implementing this on the GPU is straightforward -- just three lines of code.



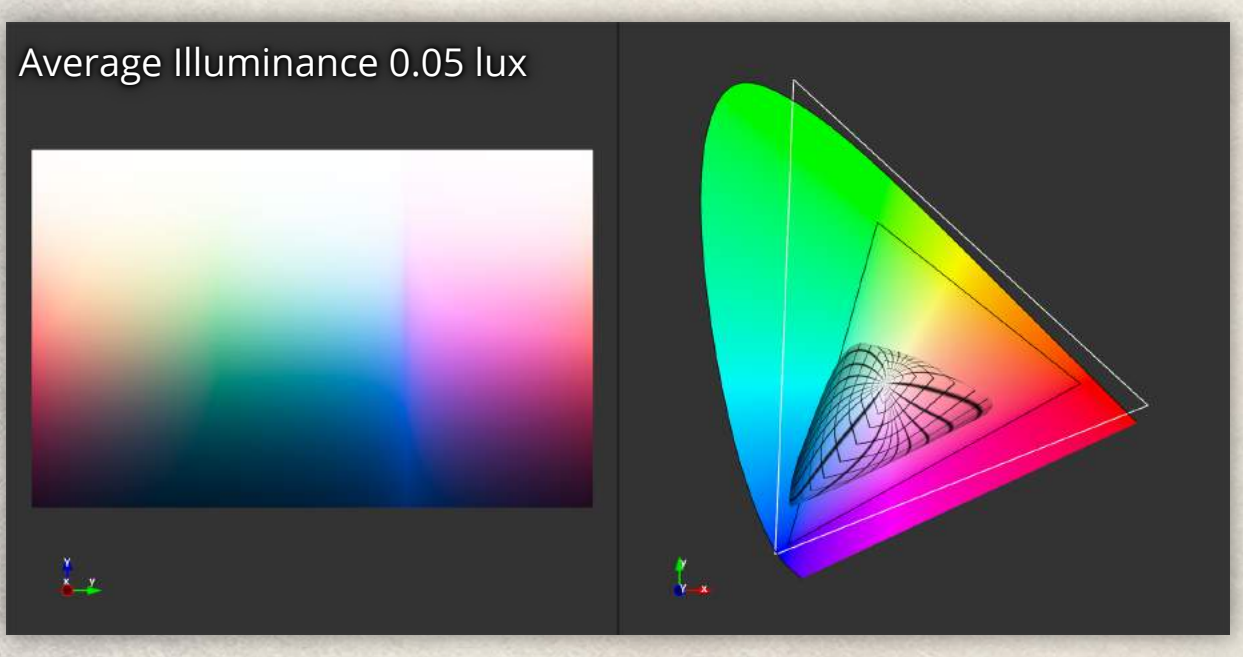
Speaker notes

This animation shows the effect of reducing the brightness of the light from bright light to an illuminance averaging about 0.05 lux. As you can see, the darker colors especially are brightened and blue shifted.



Speaker notes

To show the difference again, here is the output without any color shift,



Speaker notes

And here is the color shift at about 0.05 lux.

Speaker notes

Finally, here's an in-game example without Purkinje shift applied...

Speaker notes

and this is the same scene with Purkinje shift enabled; the average illuminance for this scene is about 0.05 lux. Note how the dark areas are brighter after applying the Purkinje effect, especially the dark blue water and sky.

Acknowledgements

- Sucker Punch rendering programmers:
  - Adrian Bentley
  - Bill Rockenbeck
  - Eric Wohllaib
  - Matthew Pohlmann
  - Tom Low
- The Sucker Punch lighting team: Jeremy Forbes, Rob Simpson, Gaby Soto, and Toby Tobler
- Jason Connell for the many discussions about sky color

Speaker notes

Before wrapping up, I want to tank all of the following people for their help and support...

Acknowledgements

- Matthijs De Smedt, Dave Elder, and Roman Margold for their contributions to this work
- Joanna Wang for her assistance with assets for this talk
- Everybody at Sucker Punch for making everything awesome
- Natalya Tatarchuk for organizing this course
- My family for their endless patience, support, and love

Speaker notes

References

1. [Ben14] Adrian Bentley. Engine postmortem of inFAMOUS: Second Son. Game Developer's Conference (video at https://www.gdcvault.com/play/1020399/Engine-Postmortem-of-inFAMOUS-Second and slides at https://www.suckerpunch.com/iss-engine-gdc2014/), 2014.
2. [BN08] Eric Bruneton and Fabrice Neyret. Precomputed atmospheric scattering. In *Special Issue: Proceedings of the 19th Eurographics Symposium on Rendering 2008*, Computer Graphics Forum. Wiley, 2008.
3. [CK07] Mark Colbert and Jařosláv Krivanek. GPU-based importance sampling. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 20. Addison-Wesley Professional, 1st edition, 2007.
4. [CPD07] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph.*, 26(3):103–es, July 2007.
5. [CPSZ08] D. Cao, J. Pokorny, V. C. Smith, and A. J. Zele. Rod contributions to color perception: linear with rod contrast. *Vision Res*, 48(26):2586–2592, Nov 2008.
6. [Cup12] Robert Cupisz. Light probe interpolation using tetrahedral tessellations. Game Developer's Conference, 2012.
7. [DD02] Frédo Durand and Julie Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Trans. Graph.*, 21(3):257–266, July 2002.
8. [DFD+15] Haarm-Pieter Duiker, Alexander Forsythe, Scott Dyer, Ray Feeney, Will McCown, Jim Houston, Andy Maltz, and Doug Walker. ACEScg: A common color encoding for visual effects applications. In *Proceedings of the 2015 Symposium on Digital Production*, DigiPro '15, page 53, New York, NY, USA, 2015. Association for Computing Machinery.
9. [Dro17a] Michal Drobot. Improved culling for tiled and clustered rendering. SIGGRAPH 2017 Advances in Real-Time Rendering in Games Course (slides available at http://advances.realtimerendering.com/s2017/2017_Sig_Improved_Culling_final.pdf), 2017.
10. [Dro17b] Michal Drobot. Rendering of Call of Duty: Infinite Warfare. Digital Dragons (slides available at https://research.activision.com/publications/archives/rendering-of-call-of-dutyinfinite-warfare), 2017.
11. [Ele09] Oskar Elek. Rendering parametrizable planetary atmospheres with multiple scattering in real-time. https://cgg.mff.cuni.cz/~oskar/projects/CESCG2009/Elek2009.pdf, 2009.

Speaker notes

Here are the references for this talk...

References

12. [Hil16] Sébastien Hillaire. Physically based sky, atmosphere and cloud rendering in frostbite. SIGGRAPH 2016 (slides available at https://blog.selfshadow.com/publications/s2016-shading-course/), 2016.
13. [KO11] Adam G. Kirk and James F. O'Brien. Perceptually based tone mapping for low-light conditions. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, New York, NY, USA, 2011. Association for Computing Machinery.
14. [Lin17] Bruce Lindbloom. Chromatic adaptation. http://www.brucelindbloom.com/index.html?Eqn_ChromAdapt.html, 2017.
15. [Nar20] Krzysztof Narkowicz. GPURealTimeBC6H. https://github.com/knarkowicz/GPURealTimeBC6H, 2020.
16. [PD09] Sylvain Paris and Frédo Durand. A fast approximation of the bilateral filter using a signal processing approach. *Int. J. Comput. Vision*, 81(1):24–52, January 2009.
17. [RV11] P. Ramachandran and G. Varoquaux. Mayavi: 3D Visualization of Scientific Data. *Computing in Science & Engineering*, 13(2):40–51, 2011.
18. [Sch15] Andrew Schneider. The real-time volumetric cloudscapes of horizon: Zero dawn. SIGGRAPH 2015 (slides available at https://advances.realtimerendering.com/s2015/), 2015.
19. [Sch18] Christian Schüler. Followup to atmospheric scattering—part 1: Overview. http://www.thetenthplanet.de/archives/4519, 2018.
20. [SH05] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. In Matt Pharr, editor, *GPU Gems 2*, chapter 20. Addison-Wesley Professional, 1st edition, 2005.
21. [Slo08] Peter-Pike Sloan. Stupid spherical harmonics (SH) tricks. Game Developer's Conference (also available at https://ppsloan.org/publications/), 2008.
22. [Slo17] Peter-Pike Sloan. Deringing spherical harmonics. In *SIGGRAPH Asia 2017 Technical Briefs*, SA '17, New York, NY, USA, 2017. Association for Computing Machinery.
23. [Smi99] Brian Smits. An RGB-to-spectrum conversion for reflectances. *J. Graph. Tools*, 4(4):11–22, December 1999.
24. [Wro14] Bartlomiej Wronski. Volumetric fog: Unified, compute shader based solution to atmospheric scattering. SIGGRAPH 2014 (slides available at https://bartwronski.com/publications/), 2014.

Speaker notes

References

25. [Wro15] Bartlomiej Wronski. White balance and physically based rendering pipelines: Part 1 – introduction. https://bartwronski.com/2015/10/14/white-balance-and-physically-based-rendering-pipelines-part-1-introduction/, 2015.
26. [Wro16] Bartlomiej Wronski. Localized tonemapping – is global exposure and global tonemapping operator enough for video games? https://bartwronski.com/2016/08/29/localized-tonemapping/, 2016.
27. [Yus13] Egor Yusov. Outdoor light scattering sample update. https://software.intel.com/content/www/us/en/develop/blogs/otdoor-light-scattering-sample-update.html, 2013.

Speaker notes

Speaker notes

.. and finally, I'd like to thank *you* very much for your attention; if you have any questions, my Twitter handle is @jasminpatry. Thanks again!