

Hi everyone, I'm Michał and I'm here together with Peter-Pike and the title of our talk today is „Precomputed Lighting in Call Of Duty: Infinite Warfare”

ACKNOWLEDGEMENTS



- Central Tech
 - Adrien Dubouchet
 - Manny Ko
 - Dimitar Lazarov
 - Josiah Manson
 - Mike Stark
- Infinity Ward
 - Michał Drobot
 - Olin Georgescu
 - Ifedayo Isibor
 - Kyle McKisic
 - Peter Pon
- High Moon Studios
 - Martin Ecker
 - Stephane Etienne
- Sledgehammer Games
 - Dave Blizzard
 - Danny Chan
 - Stephanus
 - Luka Romel
 - Atsushi Seo

Before i start i wanted to say that *a lot* of people contributed to the ideas we will present here and we wanted to thank them all for that



In the unlikely case you haven't heard about Call Of Duty, it's a first person shooter with fast paced action, running 60 fps



The latest installment – Infinite Warfare – takes the player into space – we fight on Earth, but also on Moon, Mars, and in space!

HIGH LEVEL OVERVIEW



- Indirect lighting in CoD:IW is a combination of:
 - Lightmaps
 - Probe lighting
- Lightmaps – big structural geometry
 - Representation
 - Projection
- Light probes – everything else
 - Decoupling visibility from lighting
 - Representation of visibility
 - Interpolating the lighting
 - Light grid structure for storage and access
 - Generation
 - Baking
 - Visibility representation

And for all that we need some sort of rendering indirect lighting. It needs to look good and it needs to be fast – because, like i said, we’re running 60hz. So just to give you a quick overview of what we’re going to talk about – the indirect lighting in CoD:IW is a combination of lightmaps and probe lighting. Lightmaps are used fairly sparingly, on big, structural geometry only, and we’ll cover how we store them, how we bake and project to that representation.

Everything else is lit with probe lighting – and we’ll cover how we decouple lighting from visibility signal, how we store and interpolate them and also talk about the light grid data structure that we use to query the indirect lighting at runtime.

INDIRECT LIGHTING

ACTIVISION
CENTRAL TECH



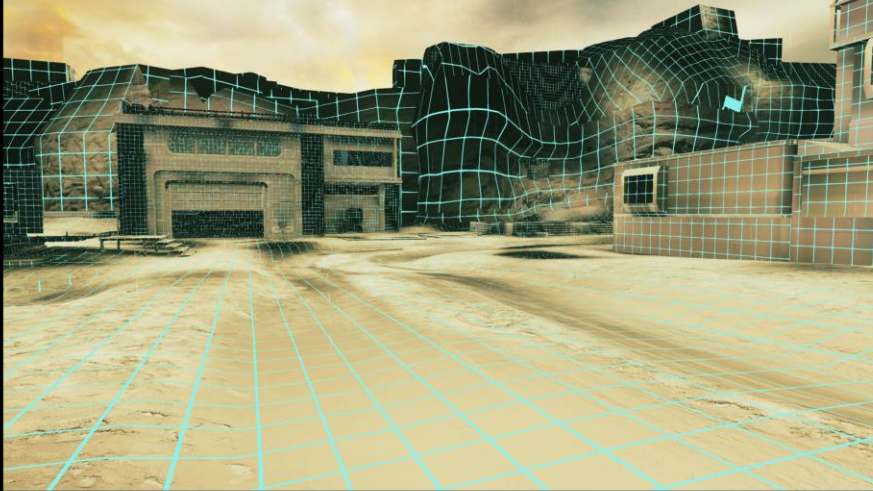
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017


This is an example scene rendered with just the indirect lighting

LIGHTMAPPED GEOMETRY

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

This is just the lightmapped objects – so as you can see only the big, base geometry

STATIC PROBE LIT GEOMETRY

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017


All those are static object that are lit with light probes

DYNAMIC PROBE LIT GEOMETRY

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

And those are the dynamic ones, also lit with probes

STRUCTURAL GEOMETRY

ACTIVISION
CENTRAL TECH

- Lightmaps
- (A)mbient + (H)ighlight (D)irection encoding
 - We explored other options but AHD gave biggest bang/\$
- In tangent space
- Hemi-octahedral encoding
- BC6H for the colors
 - option to disable compression on per-map basis



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

First the lightmaps – just briefly here, they’ve been around for a while, we like them. They are really simple and fast to look up – which is priceless for us. Also, their dimensionality matches the dimensionality of the surface. This way there’s a linear relationship between the area of the lightmapped geometry and the memory consumption.

To encode the directional signal and allow for local normal variation, we use what we call AHD encoding – which stands for Ambient and Highlight Direction – so a combination of ambient and directional light for every pixel of the lightmap. We looked at other representations, Peter-Pike will talk about this more in a bit, but AHD simply gave us the biggest bang for a buck. We store the direction in tangent space, which lets us to use hemi-octahedral encoding for it. For the two colors we compress them using BC6h, and while we had some minor quality issues with that, artists had an option to disable the compression on per-map basis.

EVERYTHING ELSE

- But lightmaps are not a silver bullet
- Bushes, small geometry etc.
- Previously mostly vertex lit or probe lit
 - Vertex lighting costly
 - every instance has an extra vertex stream with *directional* lighting data
 - Typical probe doesn't really look well
- We wanted some elegant solution for that

ACTIVISION
CENTRAL TECH



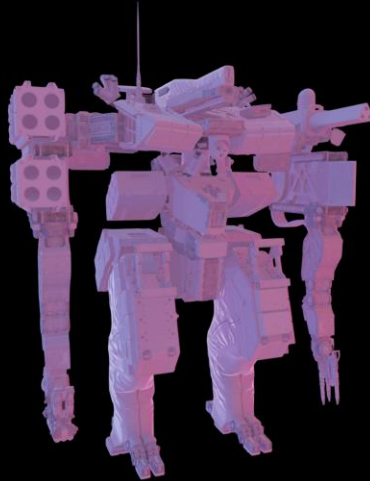
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

But we are aware that lightmaps are not a silver bullet – they work great in some cases, and totally don't in others. Things like small geometry, bushes, things with more volumetric structure, lightmap poorly. In previous games we use either vertex lighting or probe lighting for them. The problem was that vertex lighting looked great but was pretty costly, as every instance had to have a second vertex stream with the directional lighting information. The probe lighting on the other hand was pretty cheap, because we only had to store one lightprobe for such objects, but it didn't really look good, and we wanted to get a solution that would look good and be cheap.

TYPICAL PROBE LIGHTING

ACTIVISION
CENTRAL TECH



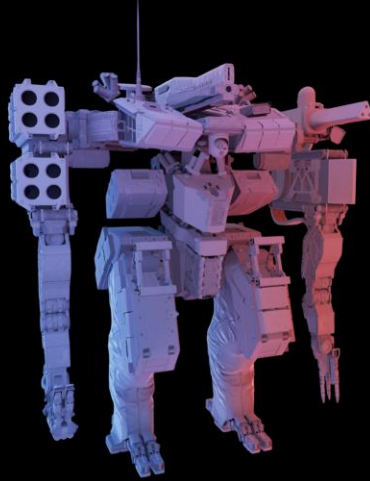
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017


Now, this is what the typical probe lighting usually looks like. And by „typical” i mean that you sample lighting in one location and use it for the whole object, it’s usually encoded as 3rd order spherical harmonics, it’s looked up by the normal, you do the convolution with the cosine on the fly and voila. And this of course gives you some directional variation, but in general looks fairly flat and uninteresting

WHAT WE WANT

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

What we really want is something like that – so with spatial variation over the object, with proper account for occlusion – something that would be comparable in quality with baking lighting per vertex.

DECOUPLE

- Rendering equation for diffuse surface under distant lighting with occlusion only:

$$L_o(\Omega) = \int_{\Omega} L_i(\vec{\omega}) V(\vec{\omega}) \cos(\vec{\omega}, \vec{n}) d\vec{\omega}$$

- Two terms with very different characteristics:
 - Incoming radiance – smooth and well behaved
 - Visibility – changes quickly, high frequency directional components
- Decouple the two!

So how did we approach the problem? If you look at the rendering equation, in a version with a distant lighting and visibility only, so we'll be ignoring bounces of the lighting off the object itself, there are two main term there – lighting and visibility. And they have very different characteristics – lighting is fairly well behaved, changes spatially in a smooth way, if we're considering diffuse response only, like here, we can treat is as varying smoothly directionally too. And visibility is nothing like that, it changes very rapidly across the object, has a lot of discontinuities. So the idea was to decouple the two and think of them separately.

DECOUPLE



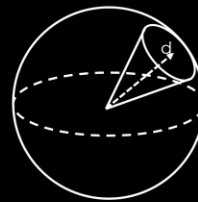
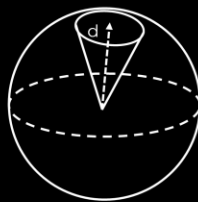
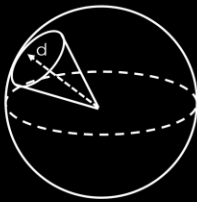
- Can reason about lighting and self visibility separately
 - encode in different ways
 - interpolate separately
- Saves tons of memory
 - visibility can be shared
 - lighting changes at low spatial rate – can be stored at lower resolution
- In our particular case:
 - Visibility stored at vertices
 - Lighting stored at some number of points on the object – more later

Because when we do this we can encode the two in a different way, we can interpolate them in a different way. And we can save a lot of memory, because the visibility is no longer unique for every instance – it can be shared between the copies of the object – and the lighting, even though it is unique, it changes at a low spatial rate so it can be stored in much lower resolution.

In our particular case we store visibility at the vertices and the lighting in some number of points scattered around the object – and we’re gonna talk about those two components now.

VISIBILITY

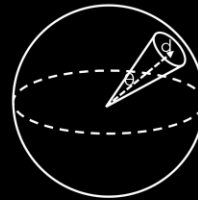
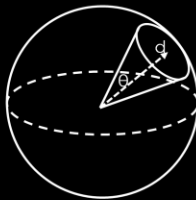
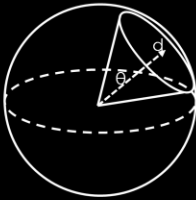
- Store visibility as a cone:
 - Axis



First the visibility – we store it as a scaled cone – so for every vertex on a mesh we store its axis – the main unoccluded direction

VISIBILITY

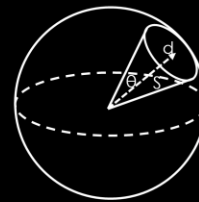
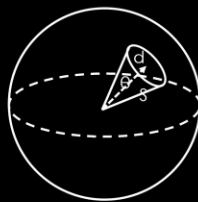
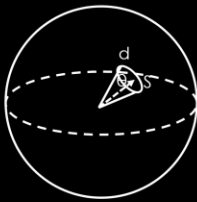
- Store visibility as a cone:
 - Axis
 - Cone angle



Then, we store the cone angle – so the wider the unoccluded portion of the hemisphere is, the wider the cone

VISIBILITY

- Store visibility as a cone:
 - Axis
 - Cone angle
 - Scale (0 to 1)
- Visibility is shared between instances
- Need to bake only once – during conversion



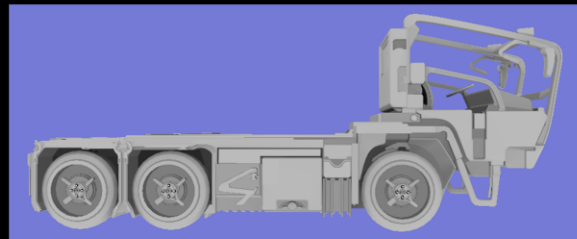
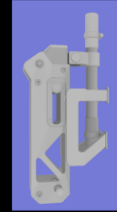
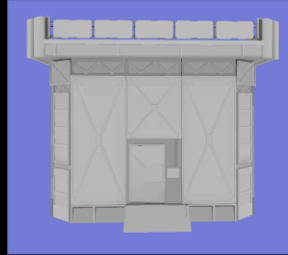
And then we store the scale – from 0 to 1, to have an extra degree of freedom, to be able to represent more stochastic visibility, which a single cone might not be a good representation of. So when a point is occluded from multiple directions, instead of compensating this with the smaller angle, we can use a wide angle and just scale it down.

And like i mentioned, visibility is shared between instances and baked only once, during mesh import.

BAKING VISIBILITY

- Generate 4th order SH
- Uniform sampling over the surface
- Validate sample
- Non-linear iterative smoother
 - Interpolate norm of linear SH as extra channel, rescale

ACTIVISION
CENTRAL TECH



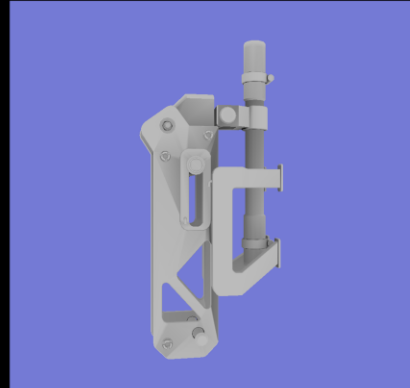
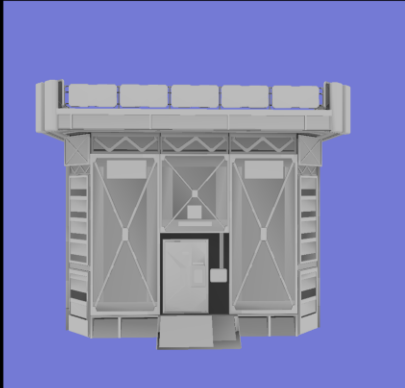
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

To get the visibility, we bake 4th order sh vector. We uniformly sample the mesh area, we validate each such sample – to check if it's not buried etc – and then we derive the values at the vertices from the baked samples. We do a couple iterations of smoothing and averaging of the baked data, to make the visibility signal cleaner

SELF VISIBILITY NO INPAINT NO BLUR

ACTIVISION
CENTRAL TECH

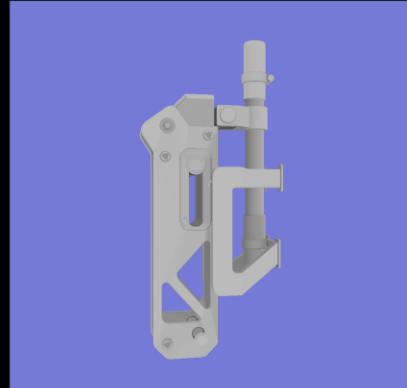


Advances in Real-Time Rendering in Games course – SIGGRAPH 2017


© 2017 Activision Publishing, Inc. SIGGRAPH2017

This is a comparison what that inpainting and blurring process gives – those are meshes baked with no post processing – as you can see they are slightly too dark in certain areas and there are some artifacts visible.

SELF VISIBILITY INPAINT AND BLUR



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

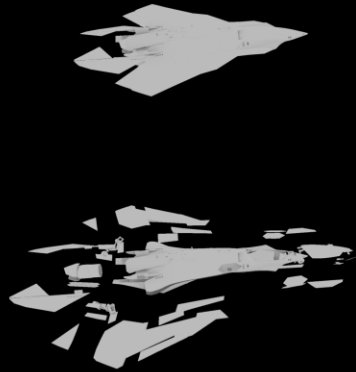
© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

And after all that post processing all those problems are gone


OTHER SELF VISIBILITY ISSUES

ACTIVISION
CENTRAL TECH

- Short rays for things like ship interiors, elevators
- Rigid objects bake components separately
 - Doors that might open
 - Landing gear
- Transparencies don't occlude
 - But they get self-vis signal
- No special handling of skinned object
 - Just make the baking rays shorter



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

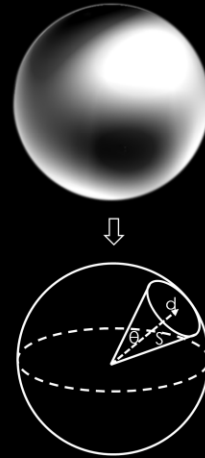
© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

A bunch of other things to mention here – we use shorter rays for the interiors – like ship insides, elevations – generally things that the player can go into – to make sure they are not „all occluded“. For rigid elements that can move – landing gear, doors etc – we bake each of such components separately. We don't do anything for the skinned models, they are just baked in the bind pose, but we also use shorter rays, to generate less occlusion in crotch, armpits etc.

FITTING CONE

ACTIVISION
CENTRAL TECH

- Least squares fit of the cone:
 - Optimal linear becomes axis
 - Non-linear fit for angle, but only 1d
 - Scale is just a ratio of visibility integrated over the cone to the cone integrated over itself



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

Now that we have the SH visibility for every vertex, we convert it to the cone representation i mentioned before. For the axis, we use the optimal linear direction from the SH. Fitting the scale and the angle requires a non linear fitting process, but it turns out the two things are actually fairly decoupled, and you can solve for the angle first – it's still a non linear solve, but a really simple one, as it's only 1d – and then calculate the scale as the ratio of the visibility integrated over the cone to the cone integrated over itself.

FITTING CONE



Baked visibility projected to SH, rotated to the frame of the cone

$$V_{SH}(\vec{n}) = \sum_i v_i Y_i(\vec{n})$$

Minimize the squared difference:

$$E = \int_{\Omega} (V_{cone}(S, \alpha, \vec{\omega}) - V_{SH}(\vec{\omega}))^2 d\omega = \int_{\Omega} (S \sum_i c_i(\alpha) Y_i(\vec{\omega}) - \sum_i v_i Y_i(\vec{\omega}))^2 d\omega$$

Partial derivatives w.r.t. degrees of freedom:

$$\frac{dE}{d\alpha} = \int_{\Omega} 2(S \sum_i c_i(\alpha) Y_i(\vec{\omega}) - \sum_i v_i Y_i(\vec{\omega})) S \sum_i \frac{dc_i(\alpha)}{d\alpha} Y_i(\vec{\omega}) d\omega$$

$$\frac{dE}{dS} = \int_{\Omega} 2(S \sum_i c_i(\alpha) Y_i(\vec{\omega}) - \sum_i v_i Y_i(\vec{\omega})) \sum_i c_i(\alpha) Y_i(\vec{\omega}) d\omega$$

Simplifying, rearranging, using SH orthonormality:

$$\frac{dE}{d\alpha} = 2S^2 \sum_i c_i(\alpha) \frac{dc_i(\alpha)}{d\alpha} - 2S \sum_i v_i \frac{dc_i(\alpha)}{d\alpha}$$

$$\frac{dE}{dS} = 2S \sum_i c_i(\alpha)^2 - 2 \sum_i c_i(\alpha) v_i(\alpha)$$

Setting partials to 0:

$$S \sum_i c_i(\alpha) \frac{dc_i(\alpha)}{d\alpha} = \sum_i v_i \frac{dc_i(\alpha)}{d\alpha} \Rightarrow \sum_i v_i \frac{dc_i(\alpha)}{d\alpha} = \frac{\sum_i v_i c_i(\alpha)}{\sum_i c_i(\alpha)^2} \sum_i c_i(\alpha) \frac{dc_i(\alpha)}{d\alpha} \Rightarrow$$

$$S = \frac{\sum_i v_i c_i(\alpha)}{\sum_i c_i(\alpha)^2} \Rightarrow$$

Scaled cone along +Z projected to SH (S: scale, alpha: cone angle)

$$V_{cone}(S, \alpha, \vec{n}) = S \sum_i c_i(\alpha) Y_i(\vec{n})$$

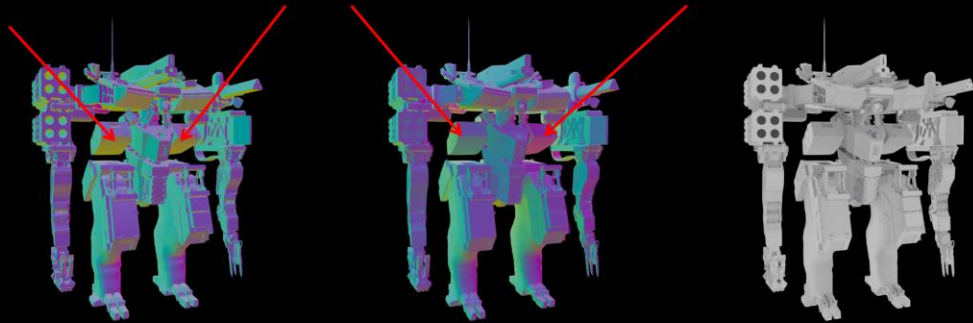
$$\begin{aligned} c_0(\alpha) &= -\sqrt{\pi}(-1 + \cos(\alpha)) \\ c_1(\alpha) &= \frac{1}{2}\sqrt{3\pi}\sin^2(\alpha) \\ c_2(\alpha) &= \frac{1}{2}\sqrt{5\pi}\sin^2(\alpha)\cos(\alpha) \\ c_{32}(\alpha) &= \frac{1}{10}\sqrt{7\pi}\sin^2(\alpha)(5\cos(2\alpha) + 3) \\ c_{1,3}(\alpha) &= 0 \\ c_{1,2,7,8}(\alpha) &= 0 \\ c_{0,3,11,13,14,15}(\alpha) &= 0 \\ \frac{dc_0(\alpha)}{d\alpha} &= \sqrt{\pi}\sin(\alpha) \\ \frac{dc_1(\alpha)}{d\alpha} &= \sqrt{3\pi}\cos(\alpha)\sin(\alpha) \\ \frac{dc_2(\alpha)}{d\alpha} &= \frac{1}{2}(3\sqrt{5\pi}\sin(3\alpha) - \sqrt{5\pi}\sin(\alpha)) \\ \frac{dc_{32}(\alpha)}{d\alpha} &= \frac{1}{10}(5\sqrt{7\pi}\sin(4\alpha) - 2\sqrt{7\pi}\sin(2\alpha)) \\ \frac{dc_{1,3}(\alpha)}{d\alpha} &= 0 \\ \frac{dc_{1,2,7,8}(\alpha)}{d\alpha} &= 0 \\ \frac{dc_{0,3,11,13,14,15}(\alpha)}{d\alpha} &= 0 \end{aligned}$$

Solve for alpha
Simple, 1d, non-linear solve
Compute S

The math for all this is here, you can go though that later, but it's actually pretty simple, just typical minimization of a quadratic error function

SELF VISIBILITY

ACTIVISION
CENTRAL TECH



Normal

Self visibility direction

Self visibility cone angle

Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

This is what the baked visibility looks like in debug visualization. As you can see the direction is fairly consistent with the normal – which of course makes sense – the main unoccluded direction will often be the normal direction – but in the occluded areas, like the robot armpits it deviates quite a bit. And the cosine of the angle looks fairly AO-ish – but that's because it is a form of AO actually – if compute the AO without the cosine term, this is actually what you get

LIGHTING



- Single lighting environment not enough
 - lighting can change drastically across large objects
- Could to gradients [Annen2004]
 - Quality/cost unsatisfactory
- Generate multiple sampling points
 - scattered around the object
 - number depending on object size
- Store the incoming lighting at those points and interpolate it across the whole object

Now, let's talk a bit about the lighting. One thing that we knew immediately was that a single lightprobe for the entire object is simply not enough – lighting can change pretty drastically across the objects, especially large ones. One option would be to use irradiance gradients, which do a form of 1st order Taylor expansion of the lighting spatially, but we weren't quite happy with it's quality to cost ratio – as in the end, it's still just one probe.

So for each object we actually use multiple sampling locations for the lighting – we scatter them around the object, with their number depending on object size – and we interpolate the lighting from those points across the whole object


DETERMINING SAMPLING POSITIONS

ACTIVISION
CENTRAL TECH

- Figure out N - the target number of probes – based on object size
- Sample object geometry uniformly, fairly densely
- Perform k-means clustering to assign samples to N clusters
- Final cluster centers after relaxation become sampling positions



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH2017

We figure out those sampling positions automatically. We first figure out how many we need – and this is just a heuristic based on object size. Then we sample the object uniformly and perform a k-means clustering to group those sample to N clusters. After few iterations, the centers of the final clusters become sampling positions.

INTERPOLATING THE LIGHTING

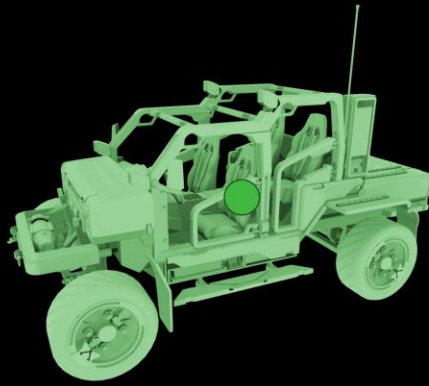


- Compute the covariance matrix of the set of sampling positions
- Relative magnitudes of the eigenvalues determine interpolation mode
- Eigenvectors define local space for interpolation

Given the set of sampling points we figure out the interpolation mode. For that we compute the covariance matrix of the set of sampling points and extract its eigenvalues and eigenvector. The structure of those determines how we perform the interpolation – the relative magnitude of the eigenvalues determines the interpolation mode and the eigenvectors form a local space in which the interpolation happens

SINGLE PROBE

ACTIVISION
CENTRAL TECH



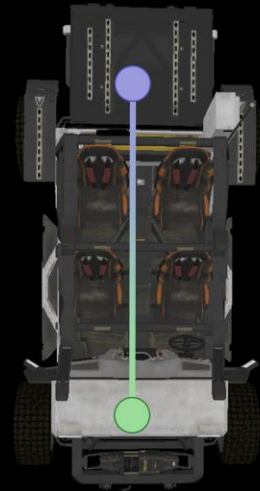
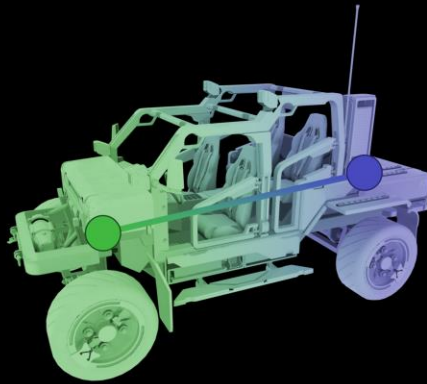
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

And this is what we mean by that. If there's only one probe for the whole object it's simple – whole object just uses the probe.

1D INTERPOLATION

ACTIVISION
CENTRAL TECH



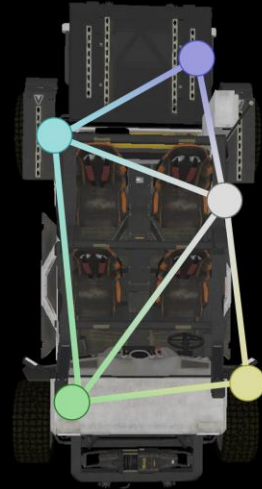
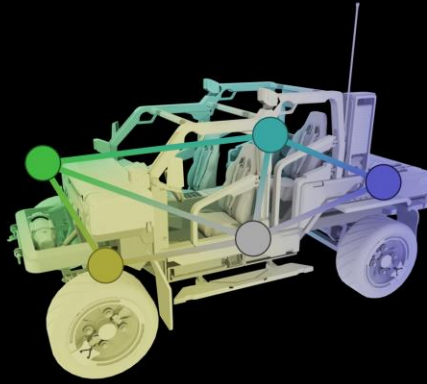
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

If we have more, but there's only one large eigenvalue it means that the sampling locations form a line. Each point on the object can be projected to that line and use the two closest probes to get the lighting, by interpolating between them. Here, we have only two probes, but if there was more and they still formed a line, they would also use the 1d interpolation mode

2D INTERPOLATION

ACTIVISION
CENTRAL TECH



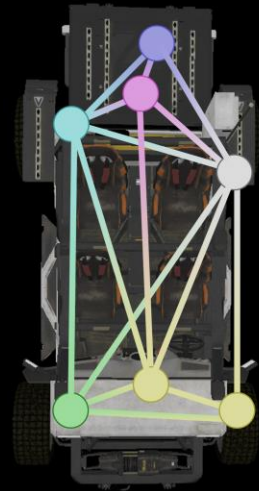
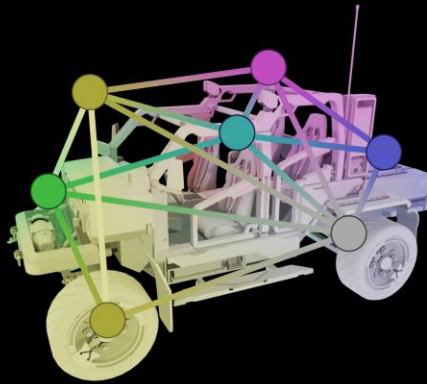
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

If there are two large eigenvalues, the sampling points form a plane. So we project everything onto this plane, triangulate the sampling point set and each point on the object uses the three probes that are the vertices of the triangle it projects to.

3D INTERPOLATION

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

If all three eigenvalues are large, the probes just form a 3d cloud – which we tetrahedralize, and each point on the object uses 4 probes – the vertices of the tet it's in – to get the lighting.

INTERPOLATING THE LIGHTING

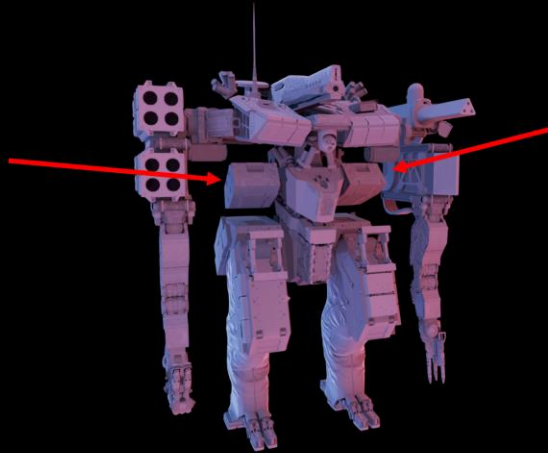


- Generate Delauney mesh with a given dimensionality
- Classify each vertex of a mesh to one simplex
- Store:
 - indices of the sampling points
 - interpolation weights
 - 2 x 4 bytes per vertex – shared between instances
- Skinned objects can explicitly attach sampling positions to particular bones

So to summarize, we generally generate Delauney mesh of a given dimensionality and classify each vertex of a mesh to one of the generated simplexes. We do it offline, and for each vertex of a mesh, we store the indices of the probes that a given vertex uses together with the interpolation weights. This is stored as two time 4 bytes per vertex – and again, this data is shared between instances. Additionally, skinned objects can bypass all that and explicitly attach sampling positions to particular bones.

SINGLE PROBE

ACTIVISION
CENTRAL TECH



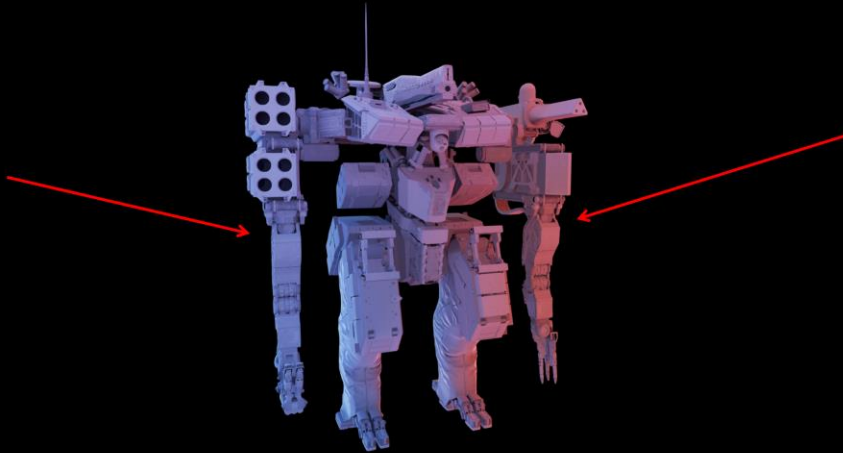
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

To show you the difference between using single probe and multiple probes – this is an object lit with a single probe. Even though it's a single probe, if you look at those two things in the center of the robot's body, they are lit differently – even though they are oriented the same way. This is what directional visibility gives you – those elements are mostly occluded in opposite directions, the light on that side gets removed with the visibility multiply and you still get different lighting

MULTIPLE PROBES

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

This is the same object lit with multiple probes – you get that color gradient going from left to right that wasn't there before. This might not really be that super important for that robot here, as it's fairly small, but we use the same method to light much larger objects too, and this is where the technique shines

SINGLE PROBE

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

This is single probe again

RUNTIME PART

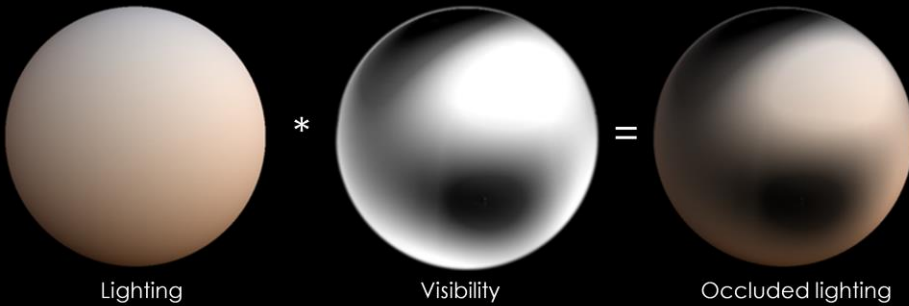


- Determine lighting for sampling points:
 - Static geometry has it baked
 - Dynamic geometry samples from the volumetric structure
- Vertex shader:
 - Grabs the sampling point indices and weights
 - Interpolates the lighting
 - Combines it with self-visibility to get occluded incoming lighting at given vertex
 - Encode it to AHD and pass to pixel shader
- Pixel shader:
 - $A + \text{saturate}(n.D) * H$

The runtime part is mostly fairly straightforward – we gather the sampling points for the visible objects, determine lighting at those points – for static geometry we just have it baked, for dynamic ones we sample the volumetric structure that stores the lighting for the level. Now, the vertex shader grabs the indices of the probes that should be used at a given vertex, the weights, does the interpolation, combines the lighting with the self-visibility that we talked about before. The result is still a spherical function, but we need to pass it to the pixel shader, so we encode it as an AHD, and the pixel shader just does $a + n.l$

TRICKY BIT

- Combining lighting with self visibility
- Want to mask parts of the probe that are occluded
 - Multiply lighting by visibility



The only tricky part in all that is combining lighting with visibility. Because we want to mask parts of the lighting that are not visible – so we want to multiply the lighting by the visibility, just like in the rendering equation earlier.

SH MULTIPLICATION



- Both lighting and visibility end up as 3rd order SH
- **SH multiply != component-wise multiplication!!!**
- Initially 3rd order multiply
 - Lighting rotated to the cone frame – cone axis pointing along +Z
 - Cone projection is just ZH – multiplication *way* simpler
 - Convert to AHD in that space
 - Rotate D back
- Mix different orders:
 - 2nd order visibility * 3rd order lighting (no need for rotations)
 - 3rd order visibility * 3rd order lighting for DC, 2nd order visibility * 3rd order lighting for remaining bands

Both visibility and lighting end up as 3rd order SH, and the problem is that SH multiply is not a component-wise multiplication. It's way more involved, coefficients get all intermingled. Initially we did however implement that – we took the lighting, we rotated it to the frame of the cone – where the projection of the cone is just a ZH, so the multiplication is way simpler, we converted the result of the multiply to AHD in that space, then rotated D back – and it all worked really well, we had it in that form for quite a while, but it could be cheaper


Then we realized that we can do mixed order multiplies – for instance do second order visibility and 3rd order sh – where a lot of the math simplifies, or you can do full 3rd x 3rd order for the DC component – as it's only a big dot product- and the remaining band as 2nd order x 3rd order

NO SELF VISIBILITY

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017


© 2017 Activision Publishing, Inc.  SIGGRAPH2017

This is the comparison.

DC VISIBILITY TIMES LIGHT



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017


© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

LINEAR VIS MULTIPLY

ACTIVISION
CENTRAL TECH




Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH2017

FULL DC PRODUCT LINEAR REST

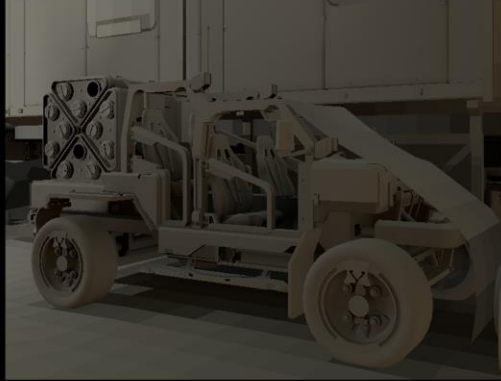


Advances in Real-Time Rendering in Games course – SIGGRAPH 2017


© 2017 Activision Publishing, Inc.  SIGGRAPH2017

FULL PRODUCT

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH2017

GCN NOTE



- GCN tweaked for a particular balance between VS and PS workloads
 - VS expected to finish XXX cycles after being launched
 - Will stall rasterizer otherwise
- Might not be a problem if enough PS work already pipelined
 - it was our case
- YMMV
 - on the next project we actually move a lot to pixel shaders due to different characteristics of the geometry
 - And also use different basis – see Ambient Dice for details [Iwanicki17]

One thing to note here is that it all happens at the vertex shader level. And the GCN – the gpu architecture used in the current generation consoles – was tweaked for a very particular balance of workload between vertex and pixel shaders. Vertex shaders are generally expected to finish x number of cycles after being dispatched, where x depends on different factors, like vgpr usage, parameter cache usage and so on. But the bottom line is that if they are not back on time, they will stall the rasterizer. It might not be a big problem – it wasn't for us here – if there's already enough of the PS work scheduled to cover that stall. But your mileage may vary – for instance on the next project we have different characteristics of the geometry that's rendered, and we actually had to move parts of that math to the pixel shades to save perf – things are done a bit differently, we use Ambient Dice as the representation of the lighting which we talked about during EGSR - but all this is a topic for another talk. This was also one of the motivations for using discrete sampling points for the lighting and interpolating the data by hand, instead of using 3d textures for instance and hardware interpolation. Because the vertex processing pipeline is so sensitive to latency, we use the scalar memory path as whenever possible, which has much lower latency of the lookup. Using texture wouldn't allow us to take advantage of such optimizations and decrease the overall performance.

LIGHTING



- We know how to interpolate the lighting and combine it with visibility
- We just need to know *what* that lighting is
- Two options:
 - Dynamic geometry: sample from light grid
 - Static geometry: bake

Now we know how to interpolate the lighting, how to combine it with the visibility, we just need to know what the lighting is.

And there are two options here – for static objects we bake it – and Peter-Pike will talk about some details on that – but for dynamic objects, we sample a volumetric data structure that I'll talk a bit now.

LIGHT GRID

ACTIVISION
CENTRAL TECH

- Light grid: our volumetric structure that holds the indirect lighting for the level
- Used by:
 - dynamic objects
 - volumetrics
 - VFX
- Designed to be accessed from GPU
 - couple tens of thousands of lookups per frame



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

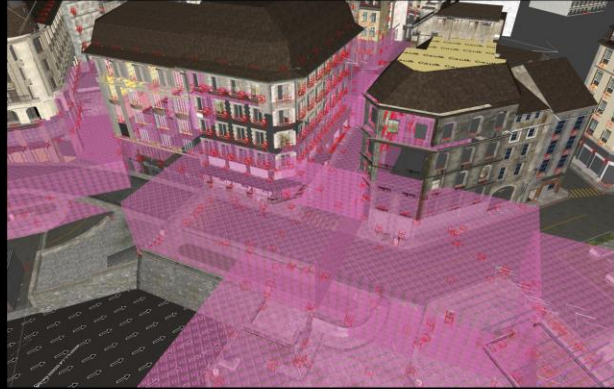
© 2017 Activision Publishing, Inc. SIGGRAPH2017

We call that structure light grid, it's being used by the dynamic objects, but also volumetrics, visual effects, decals. It was designed to be accessed from the GPU, but in the order of tens of thousands times per frame, not really per-pixel. Our motivation was that as the lighting becomes more complex, the data becomes more complex too, and it's more costly to perform the lookup – but the data, if you decouple the high frequency components – is fairly smooth – so there's no need to pay the lookup cost for every pixel – and we should rather resample to simpler representations to mitigate that cost – like those discrete sampling points I was talking about earlier


LIGHT GRID

ACTIVISION
CENTRAL TECH

- Don't want to manually place probes
- Previous system relied on manually placed volumes
 - re-use those



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

We knew that we didn't want the artists to place the probes manually. The previous system that we had relied on manually places volumes that enclosed the regions of interest and we decided to re-use them

LIGHT GRID

ACTIVISION
CENTRAL TECH

- Cannot just place uniform grid inside them
 - Transitions from ground to space – huge volumes
 - Too many probes even without that
- Non-uniform distribution of probes and tetrahedrization of the set [Cupisz12]
 - Trivial to interpolate, C0 continuity
 - Only 4 values to interpolate between



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

But we couldn't just uniformly place probes in them – we have those transitions into space, the volumes can be pretty large. So we settled on a non-uniform distribution of probes and a tetrahedrization of that set – much like solution that Robert talked about during GDC few years back. Tets are good, because you only have 4 values to interpolate between and the result is C0 continuous.

LIGHT GRID



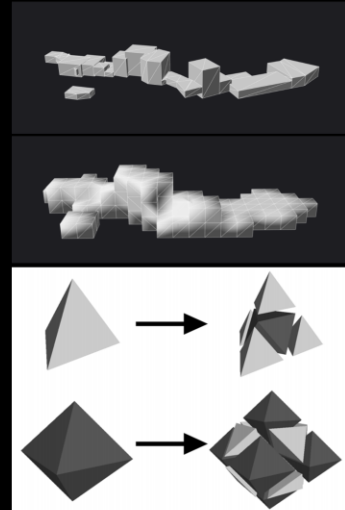
- Generate sampling points near geometry, some in empty space, generate tets out of this
 - Eeee.... no
- Delauney tetrahedrization finicky semi-random point sets
 - lot of long, thin tets
 - They create nasty artifacts at runtime
- Tried a lot of different ways to fix them
 - Add extra points near bad tets
 - Move the existing ones to be less bad
- The conclusion was that tets can do a decent job if they are fairly regular

So we did this with some probes that we generated fairly randomly near geometry – and it didn't really work well. Delauney tetrahedrization is fairly finicky when it comes to such random point sets – you end up with long, skinny tets, that behave really bad at runtime, generating temporal artifacts. And we spend a fair amount of time trying to fix them in various ways, but the bottom line was that tets can do a decent job pretty much only when they are fairly regular.

LIGHT GRID

- Start with tets that are good
 - ...and try not to mess them up too much
- Start with **very** rough voxelization of level's volumes
- Tetrahedralize voxelization
 - Only boxes in input = nice and regular tets
- Subdivide tetrahedrons
 - Scheme from [Schaefer04]
 - Tet -> 4 tets + oct
 - Oct -> 6 octs + 8 tets
 - Ensure that neighbors are at most 1 subdivision level apart
 - Subdivide near geometry, navmesh, manually placed regions of interest

ACTIVISION
CENTRAL TECH



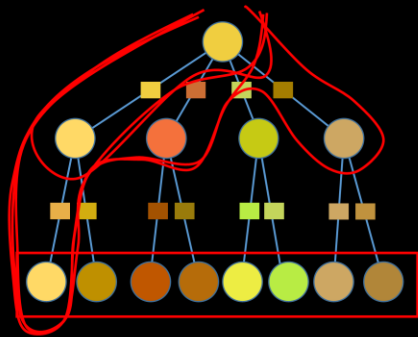
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

So the idea was to start with tets that are good and not mess them up. We take the volumes that artists placed, we generate a very rough voxelization of that. Those are only boxes, so the tets end up nice and regular. Next we subdivide them, according to the scheme by Scott Schaefer – where a tet gets split into 4 tets and octahedron, and octahedron gets split into 6 octahedrons and 8 tetrahedrons – all of them fairly close to being half of the size of the initial cell. And during that subdivision, we ensure that two neighboring cells are at most 1 subdivision level apart – which ensures that there are no badly formed cells. And we subdivide near geometry, navmesh, manually placed regions of interest.

LIGHT GRID

- Compute the rough lighting of the points at the bottom of the hierarchy
- For each subdivision in the hierarchy compute the error
 - squared difference between interpolated lighting at a given level and one level lower
 - analytically: integrals over the volume of the cell
- Starting from the top of the hierarchy refine again
 - Refinement priority based on the error, proximity to areas of interest etc.
- Final set of points is regular and aligns with both geometry and with signal
- The cells are not tets however – just re-mesh



This creates a lot of points at the very bottom, way too many for use. So we compute the lighting at the bottom of that subdivision hierarchy, then for each subdivision we compute the error – the squared difference between the lighting interpolated at a given level and the lighting computed using the data from one level lower. And then, starting from the top of the hierarchy, we subdivide again, this time with a heuristic that relies on that error – and things like distance to regions of interest, still conforming to that „neighbors being 1 level apart” rule. In the end, we get a set of points that’s really regular, and aligns with the geometry as well as with the lighting signal. The cells are not only tets however – as we generate those octahedrons too – but we just take the resulting probes and re-mesh them.

TETRAHEDRAL MESH

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017


This is what it looks like – just a cut through the tet mesh, the edge of that cut is being shown – as you can see the density changes, but in a smooth fashion – and in the areas where the player moves the density is higher

LIGHT GRID

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH2017

This is the resulting set of probes

LIGHT GRID VISIBILITY

ACTIVISION
CENTRAL TECH

- Lighting bleeds through walls ☹
 - Probes used even when not visible from lookup position
- Augment probes with some visibility information to limit their influence
- For each probe, for each tet it touches store a triangular depth map
 - Store barycentric of the first intersection: 0-1 range



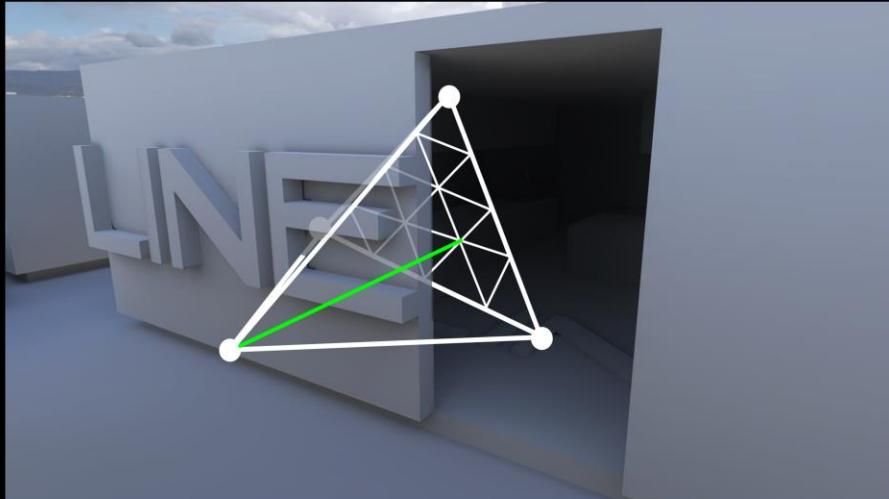
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

The only thing that's left is the visibility – if we just interpolate the lighting between the probes the lighting will bleed through walls – you'll lighting from the outside in the insides and vice versa. So our solution is to augment the probes with some visibility information to limit their influence. For each probe, we store a triangular depth map that stores the barycentrics of the first intersection on the ray to the opposite triangle

GENERATING VISIBILITY

ACTIVISION
CENTRAL TECH



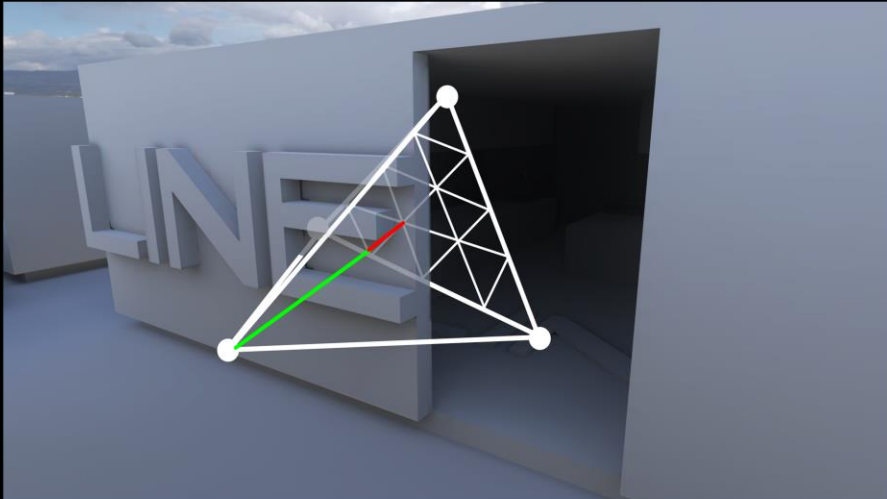
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

This is a graphical example, looking at the probe on the left, we shoot a ray towards the opposite face of the tet, it doesn't hit anything so we store 1.0

GENERATING VISIBILITY

ACTIVISION
CENTRAL TECH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

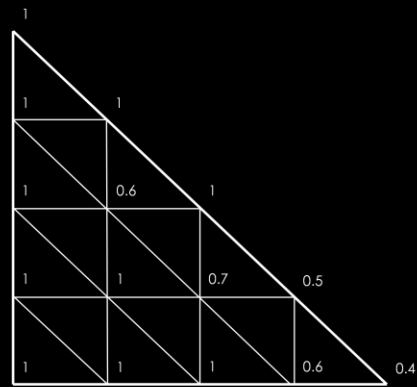
© 2017 Activision Publishing, Inc. SIGGRAPH2017

We shoot another ray, it hits something, so at that location we store something like 0.6

LIGHT GRID VISIBILITY

ACTIVISION
CENTRAL TECH

- Store 15 points, 8 bits of precision
 - 4 DWORDs total
- On finest subdivision level that gives us 2 bits per unit – more than enough



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

And we end up with that triangular depth map. We store 15 values, with 8 bits of precision, so total everything fits into 4DWORDs. And the precision, even though it might sound low, is actually more than enough – we get like 2 bits per unit on the finest subdivision level – and our unit is around 1inch.

ACCESSING VISIBILITY DATA



```
float GetDepthForTetrahedronProbe( float2 bary, uint4 encodedDepth )
{
    float sampleY = floor( bary.y * 4.0f );
    float sampleX = floor( bary.x * 4.0f );

    int baseIndex = (int)( ( 11.0f - sampleY ) * sampleX * 0.5f + sampleX * 0.5f );
    int lineLength = 5 - (int)sampleY;

    float2 baseBary = float2( sampleX * 0.25f, sampleY * 0.25f );
    int3 indices = int3( baseIndex + 1, baseIndex + lineLength, baseIndex );

    float3 sampleBary;
    sampleBary.x = ( bary.x - baseBary.x ) * 4.0f;
    sampleBary.y = ( bary.y - baseBary.y ) * 4.0f;

    if ( sampleBary.x + sampleBary.y > 1.0f )
    {
        // above the diagonal
        indices = int3( baseIndex + lineLength, baseIndex + 1, baseIndex + lineLength + 1 );

        float3 newSampleBary;
        newSampleBary.x = -( sampleBary.y - 1.0f );
        newSampleBary.y = -( sampleBary.x - 1.0f );
        sampleBary = newSampleBary;
    }

    sampleBary.z = 1.0f - sampleBary.x - sampleBary.y;
    float result = 0.0f;
    for( int i=0; i<3; ++i )
    {
        uint currIndex = indices[i];
        int intDepth = ( encodedDepth[currIndex / 4] >> ( ( currIndex << 4 ) * 8 ) ) & 0xFF;
        float depth = (float)intDepth / 255.0f;

        result += sampleBary[i] * depth;
    }

    return result;
}
```

This is the code to access that data from those 4 dwords based on the barycentrics, really simple, you can go through that after

LIGHT GRID RUNTIME



- Simple async jobs looks up lighting in the light grid
- Bootstrap tetrahedron search with:
 - tetrahedron used in previous frame
 - voxel tree – piggyback on structure used for other purpose – add a good starting tet to each node
- Walk across tets until all barycentrics are in $[0..1]$
 - otherwise move in the most negative direction
 - see [Cupisz12] for details
- Grab the probes, compute interpolation weights, mask by visibility if needed, interpolate, write out

The runtime part is fairly straightforward again – async running somewhere in the background early in the frame. We bootstrap the tetrahedron lookup with a tet index used in previous frame or if it's not available we have a fallback start tet in a data structure used for other purposes – we piggybacked on that. The search is a simple walk through the tests, evaluating the barycentrics until they are in 0-1 range – just like Robert described. Once we get to the proper tet, we grab the lighting, evaluate the visibility of the probes, interpolate and write out the result

LIGHT GRID RUNTIME



- Ended up adding slight temporal filtering of sampled lighting
 - Smooths lighting for fast moving objects
- Total of ~100k probes/level
 - ~60 MB for all the data
- All the lookups take ~0.2-0.4ms
 - But it's only a handful of wavefronts
 - They just hide somewhere in the early part of the frame

Few notes: we ended up using some slight temporal filtering on the output lighting – fast moving objects can flicker in some cases without it.

Target number of probes per level is around 100k, it all takes around 60 MB total – a lot of that data is actually visibility information, which could be compressed in some simple ways, but we never got to that. The lookups take on the order of 0.2-0.4ms, but it's only a handful of wavefronts, so when run on async pipe, they just hide somewhere and are virtually free.

BAKING

GENERAL BAKING



- Standard-ish baking algorithms (Embree for RT)
- Decouple indirect from self-vis for vertex bake
- Support emissive/point/line lights
- Validation + in-painting for "stuff jammed in stuff"
- De-noising
- Non-lightmapped objects bounce light via probes
- Seam solving for charts
- ...

There are a lot of things we do in our baking pipeline that I am not going to talk about here. I will just focus on how we sample lighting for the new data Michal talked about (lightgrid and static models with decoupled visibility), how we “de-ring” spherical harmonics and discuss lightmap encodings and the algorithm we use to project into AHD.

HEMISPHERICAL BASIS



- Quadratic SH is over-kill
- Ambient Cube [McTaggart04]
 - Fast, 3 coefficients, can struggle with seams
- Decompose SH to ambient and directional light is common
 - [Chen08][Lazarov13][Iwanicki13]
- H-Basis (linear SH least squares over hemisphere) [Habel2010]
 - Better bounce, but less directionality
- Eigenfunctions from Gramm matrix of SH over hemisphere

There are a lot of potential basis functions that can be used to encode lightmaps. We have investigated several of them, but ended up using AHD. The one we considered the most carefully was the H-Basis, which is equivalent to least squares projecting quadratic spherical harmonics to linear spherical harmonics over the hemisphere.

One new family of basis functions we looked at is a generalization of the H-basis. If you take any set of basis functions, say spherical harmonics, you can build the Gramm matrix over a domain (integral of pair of basis functions over the domain – hemisphere in our case, orthogonal basis have an identity Gramm matrix) and take it's eigen vectors. The linear combination from the coefficients of each eigen vector represents a new basis function (some of quadratics for example), and using K of them you can do better than the H basis in general.



SH3

Here is quadratic spherical harmonics, impractical, but the input for all the other basis. This lighting environment has a mix of directions, which makes it a tough case for AHD.



AHD

AHD has high contrast, the strong shadow terminator is obvious on the bottom of this image. This makes normal maps have higher contrast, and is something our lighters like. Since it only has a single direction, it can struggle with color bleeding if they are from different directions.



SH2

Linear SH without any least squares projection can ring (the red coloring is ringing), and it is a lot flatter than AHD/SH3.



Hbasis (SH2Hemi)

The Hbasis is flatter than AHD, it can ring (de-ringed in this example, which loses directional contrast.)



Hbasis (NoWindow)

Here is the hbasis without de-ringing, which has more contrast but also can go slightly negative. It might be you can window less aggressively (the windowing algorithm employed only changes the non local Z functions and guarantees that the function is strictly positive.)



Hbasis (SH2Hemi)

Here it is with windowing again.



AHD

And AHD for comparison.



Eigen3

Here are the first 3 eigen basis vectors, this is less memory than H-basis (3 basis functions vs. 4), but you would need to implement a de-ringer.



Eigen4

First 4 eigen vectors, same memory as h-basis, but better directionality, and also would need a de-ringer.



AHD

We ended up going with AHD, while they can have interpolation issues, they are inexpensive, and have high contrast with normal maps. You still have to determine how to projection lighting in them.

PROJECTION FROM SH TO AHD



- Two options:
 - Simple least-squares fit - for use in shader
 - SH optimal linear direction - D
 - More fancy solve for the bake time
 - Constrained fit of A/H
- Optionally filter hilite direction before solve
 - Direct lights or large changes in hilite direction cause interpolation artifacts

During the bake we generate world space SH data (3rd order for lightmaps, 4th order for models), and project into AHD at two spots. In the vertex shaders when using decoupled visibility (after multiplying with self-vis) and during the bake with additional constraints that I will describe next.

AHD can struggle if the hilite direction changes quickly. This mostly happens with lights that bake direct lighting into the lightmaps. There is an option to filter the optimal linear direction before AHD fitting that helps this, but was only used on a couple of maps.

LEAST SQUARES SH TO AHD



Source as SH (l - light env. convolved with cos): Target as SH (a - ambient light - just DC, b̄ - dir. light convolved with cos):

$$I_{SH}(\vec{n}) = \sum_i \hat{l}_i Y_i(\vec{n}) \quad I_{AHD}(\vec{n}) = A \sum_i a_i Y_i(\vec{n}) + \pi H \sum_i \hat{b}_i(\vec{d}) Y_i(\vec{n})$$

Minimize the squared difference:

$$E = \int_{\Omega} (I_{AHD}(\vec{\omega}) - I_{SH}(\vec{\omega}))^2 d\omega = \int_{\Omega} (A \sum_i a_i Y_i(\vec{\omega}) + \pi H \sum_i \hat{b}_i(\vec{d}) Y_i(\vec{\omega}) - \sum_i \hat{l}_i Y_i(\vec{\omega}))^2 d\omega$$

Partial derivatives w.r.t. degrees of freedom:

$$\frac{dE}{dA} = \int_{\Omega} 2(A \sum_i a_i Y_i(\vec{\omega}) + \pi H \sum_i \hat{b}_i(\vec{d}) Y_i(\vec{\omega}) - \sum_i \hat{l}_i Y_i(\vec{\omega})) \sum_i a_i Y_i(\vec{\omega}) d\omega$$

$$\frac{dE}{dH} = \int_{\Omega} 2(A \sum_i a_i Y_i(\vec{\omega}) + \pi H \sum_i \hat{b}_i(\vec{d}) Y_i(\vec{\omega}) - \sum_i \hat{l}_i Y_i(\vec{\omega})) \pi \sum_i \hat{b}_i(\vec{d}) Y_i(\vec{\omega}) d\omega$$

Simplifying, rearranging, using SH orthonormality:

$$\frac{dE}{dA} = 2A \sum_i a_i^2 + 2\pi H \sum_i a_i \hat{b}_i(\vec{d}) - 2 \sum_i a_i \hat{l}_i$$

$$\frac{dE}{dH} = 2A\pi \sum_i a_i \hat{b}_i(\vec{d}) + 2\pi^2 H \sum_i \hat{b}_i(\vec{d})^2 - 2\pi \sum_i \hat{b}_i(\vec{d}) \hat{l}_i$$

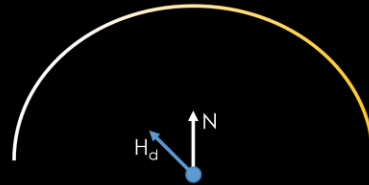
Setting partials to 0:

$$\begin{aligned} A \sum_i a_i^2 + \pi H \sum_i a_i \hat{b}_i(\vec{d}) &= \sum_i a_i \hat{l}_i \\ A\pi \sum_i a_i \hat{b}_i(\vec{d}) + \pi^2 H \sum_i \hat{b}_i(\vec{d})^2 &= \pi \sum_i \hat{b}_i(\vec{d}) \hat{l}_i \end{aligned} \quad \Rightarrow \quad \begin{aligned} a_0 &= 2\sqrt{\pi} \\ a_{1..n} &= 0 \\ \hat{b}_0(\vec{d}) &= \frac{1}{2\sqrt{\pi}} \\ \sum_i \hat{b}_i(\vec{d})^2 &= \frac{2}{3\pi} \end{aligned} \quad \Rightarrow \quad \begin{aligned} 4\pi A + \pi H &= 2\sqrt{\pi} \hat{l}_0 \\ \pi A + \frac{2\pi}{3} H &= \pi \sum_i \hat{b}_i(\vec{d}) \hat{l}_i \end{aligned} \quad \Rightarrow \quad \begin{aligned} A &= \frac{4\hat{l}_0}{5\sqrt{\pi}} - \frac{3}{5} \sum_i \hat{b}_i(\vec{d}) \hat{l}_i \\ H &= \frac{6}{5} \left(2 \sum_i \hat{b}_i(\vec{d}) \hat{l}_i - \frac{\hat{l}_0}{\sqrt{\pi}} \right) \end{aligned}$$

This is the derivation in the unconstrained case, and is what we use going from vertex to pixel shaders. You can go through the math off-line, it's just here for completeness.

AHD PROJECTION

- $A_c + \text{dot}(N, H_d) H_c$
 - A_c ambient color
 - H_c hilite color
 - H_d hilite direction
 - Luminance optimal linear
- $A_c + H_d \cdot z * H_c = C_z$
 - Scalar irradiance preserved without normal maps
- A_c and H_c non-negative

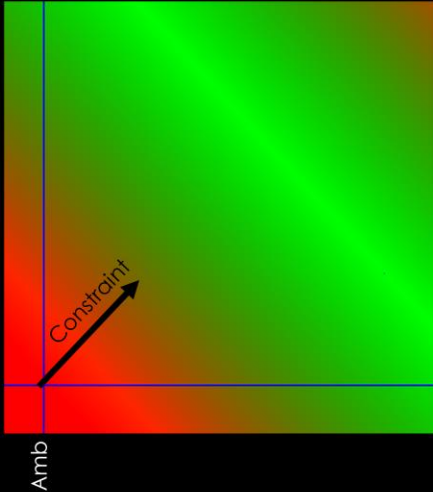


AHD stands for Ambient Hilite Direction. It is a simple model – just a directional and ambient light at each texel.

For our lightmaps the solver we used is a bit more sophisticated. If there is no normal map, we want to preserve scalar irradiance – so the fit is invariant to that.

We also want to force the ambient and hilite colors to be non-negative.

AHD PROJECTION



Constraint Equation (matrix):

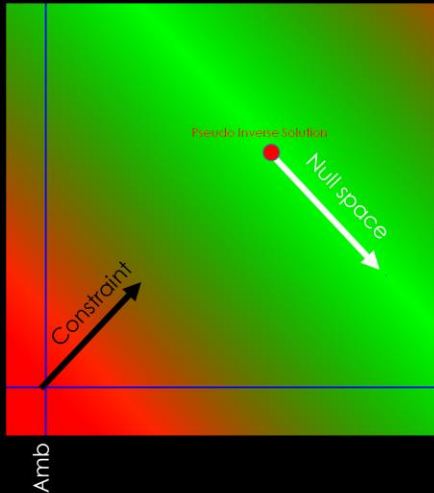
$$\begin{bmatrix} 1 & H_z \end{bmatrix} \begin{bmatrix} A_c \\ H_c \end{bmatrix} = C_z$$

If we look at the space of all ambient/hilite colors, we can build some intuition about how to solve using this constraint equation.

On the left we have ambient hilite space, and the color is the squared error with respect to the constraint. The black vector is the constraint axis.

The equation on the right is the constraint equation. It's a simple matrix that when multiplied by ambient/hilite colors, determines what outgoing radiance is for local Z.

AHD PROJECTION



Constraint Equation (matrix):

$$\begin{bmatrix} 1 & H_z \end{bmatrix} \begin{bmatrix} A_c \\ H_c \end{bmatrix} = C_z$$

Null space of constraint:

$$\begin{bmatrix} -H_z & 1 \end{bmatrix}$$

Pseudo Inverse:

$$\begin{bmatrix} 1 \\ 1 + H_z^2 \\ H_z \\ 1 + H_z^2 \end{bmatrix}$$

Hilite

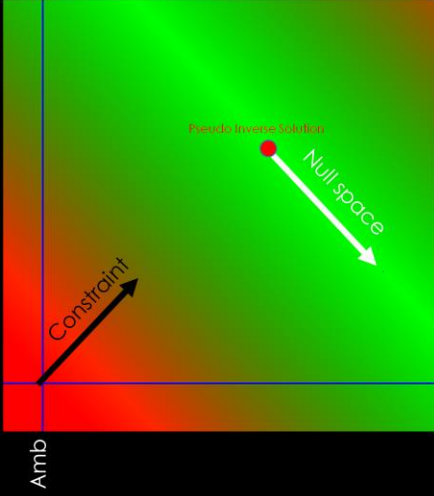
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH 2017

The white vector is the null-space – given any point in AH space, moving along the null-space of the constraint results in the same radiance when evaluated in the normal direction.

The null space of the constraint is easily computed using SVD, and we can also solve the 1D underconstrained problem using the pseudo-inverse (multiply it by C_z .) The red point is this solution, which due to the minimum norm property of the SVD is always the point that satisfies the constraint and is closest to the origin.

AHD PROJECTION



Constraint Equation (matrix):

$$\begin{bmatrix} 1 & H_z \end{bmatrix} \begin{bmatrix} A_c \\ H_c \end{bmatrix} = C_z$$

Null space of constraint:

$$\begin{bmatrix} -H_z & 1 \end{bmatrix}$$

Pseudo Inverse:

$$\begin{bmatrix} 1 \\ 1 + H_z^2 \\ H_z \\ 1 + H_z^2 \end{bmatrix}$$

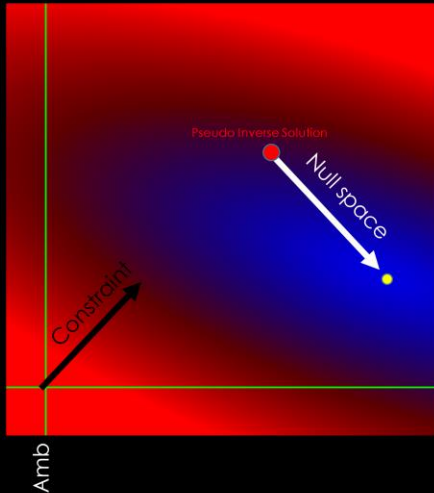
Parameterization of solution:

$$\begin{bmatrix} C_z \\ 1 + H_z^2 \\ C_z H_z \\ 1 + H_z^2 \end{bmatrix} + \begin{bmatrix} -H_z & 1 \end{bmatrix} z$$

Hillite

So there is a 1D parameterization of the points that satisfy the constraint, shown on the right. We want to now solve for a “z” coefficient that minimizes some other error.

AHD PROJECTION



Constraint Equation (matrix):

$$\begin{bmatrix} 1 & H_z \end{bmatrix} \begin{bmatrix} A_c \\ H_c \end{bmatrix} = C_z$$

Null space of constraint:

$$\begin{bmatrix} -H_z & 1 \end{bmatrix}$$

Pseudo Inverse:

$$\begin{bmatrix} 1 \\ 1 + H_z^2 \\ H_z \\ 1 + H_z^2 \end{bmatrix}$$

Parameterization of solution:

$$\begin{bmatrix} C_z \\ 1 + H_z^2 \\ C_z H_z \\ 1 + H_z^2 \end{bmatrix} + \begin{bmatrix} -H_z & 1 \end{bmatrix} z$$

Energy Functional (of null space coordinate):

$$E(z) = \int (f_z(s) - L(s))^2 \partial s$$

Hilite

Advances in Real-Time Rendering in Games course - SIGGRAPH 2017

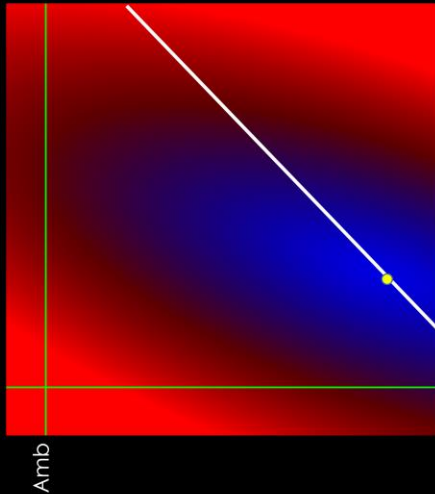
© 2017 Activision Publishing, Inc. SIGGRAPH 2017

In our case, the other error we minimize is squared error integrated over the hemisphere of the lighting environment.

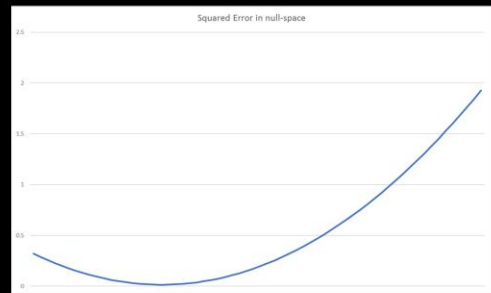
We visualize the error for this example, where blue is low error and red is high error.

This is just minimizing a 1D function, so is straightforward. You blow out f_z into SH, differentiate the error function with respect to z (the only DOF left) and solve for zero.

AHD PROJECTION



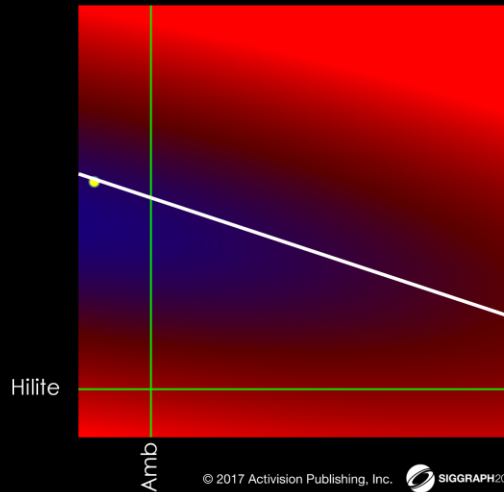
- Mix of 2 directional + ambient



This is a plot of the error along the line segment defined by the null space, and you can see we have a nice minimum that is a reasonable point in ambient/hilite space.

AHD PROJECTION TOUGH CASE

- Color channel not aligned with luminance
- Negative Hilite color is least squares

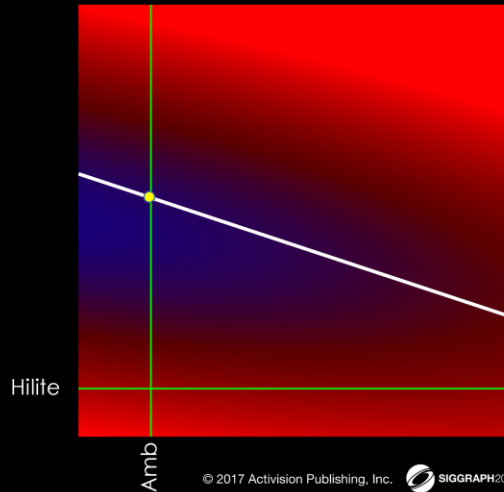
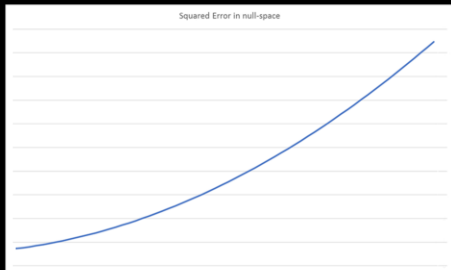


So why would either color ever be negative? This isn't that common, but where AHD struggles is when you have strong color bounce. Say you have a bright red-green light to the right, and a dim blue light to the left.

The optimal linear direction will point to the right, which causes the fit for blue to want to have a negative hilite color, ie: the lobe direction we have to use is in the wrong direction.

AHD PROJECTION TOUGH CASE

- Color channel not aligned with luminance
- Negative Hilite color is least squares



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

In this case we would end up with the intersection of the white line and Y axis (where hilite color is zero.)

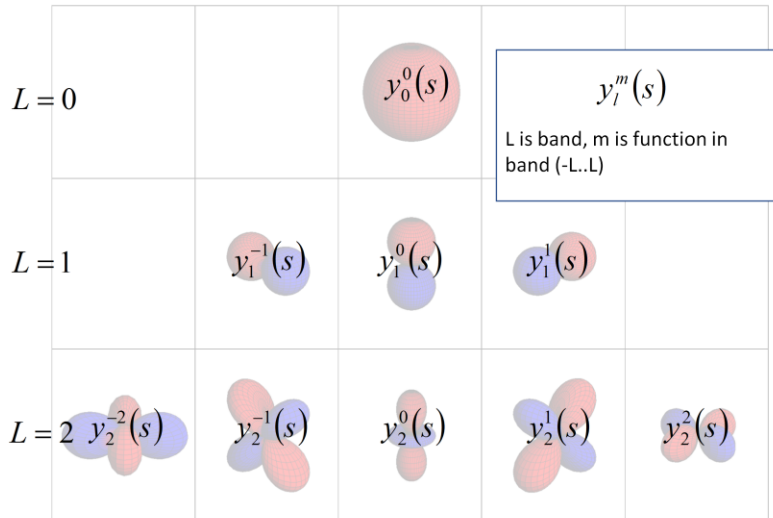
Given a convex optimization problem, with simple non-negativity constraints, the solution will always be the global minimum, or a point on the boundary of the valid region. In our case we have a 1D space, so we only need to test the two boundary points (ambient zero or hilite zero), and the one closest to the unconstrained minimum will always be the best.

Spherical Harmonics Interlude

I am going to have a brief digression to talk about ringing in spherical harmonics, and what we did to get around it for this project.

It turns out that having visibility multiplies at run-time makes ringing a larger problem than when just considering irradiance.

Spherical Harmonics

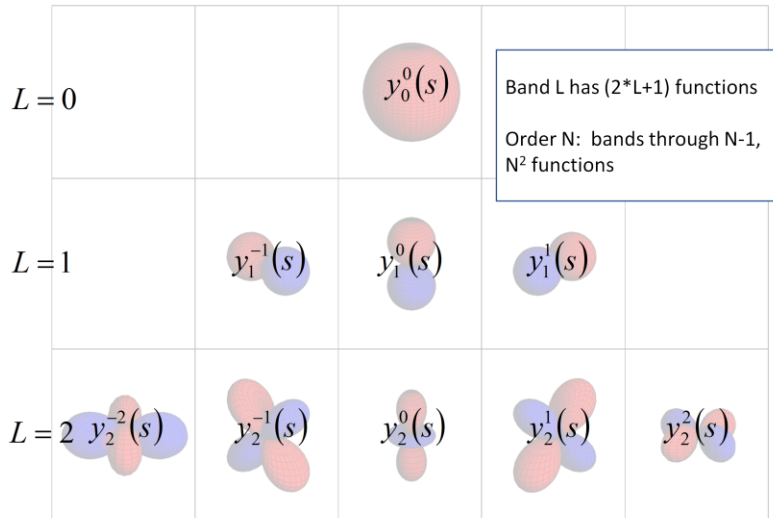


Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

The lower index of a basis function represents the band, the upper index a basis function in a band.

Spherical Harmonics

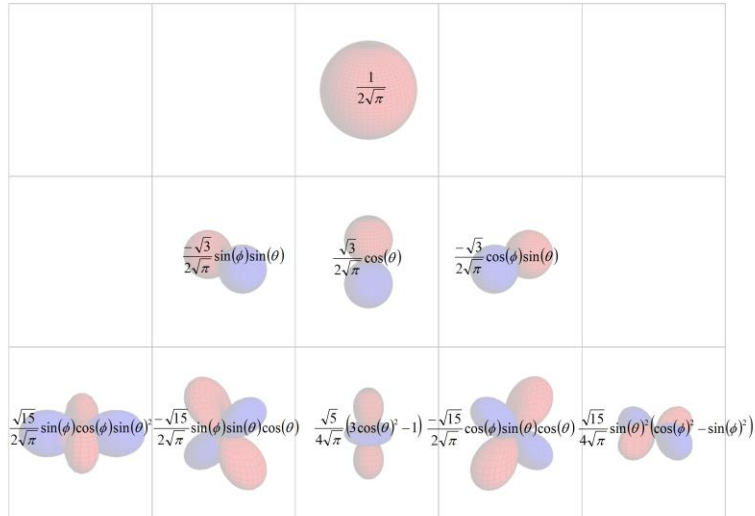


Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

Each band has $2l+1$ basis functions and the SH through order N consists of all the bands through N-1 and consists of N^2 functions.

Spherical Harmonics

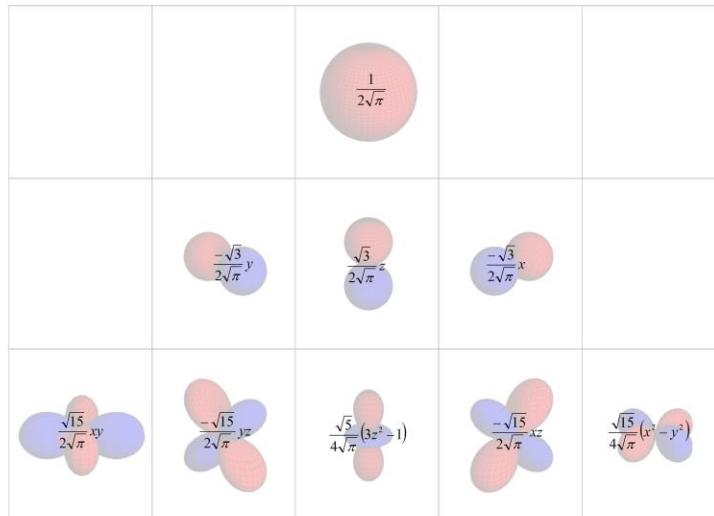


Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH 2017

The basis functions can be represented in spherical coordinates (shown here.) This is the most common definition shown in text books/the web, and is extremely useful when doing symbolic computations, but not so useful when writing code/shaders.

Spherical Harmonics

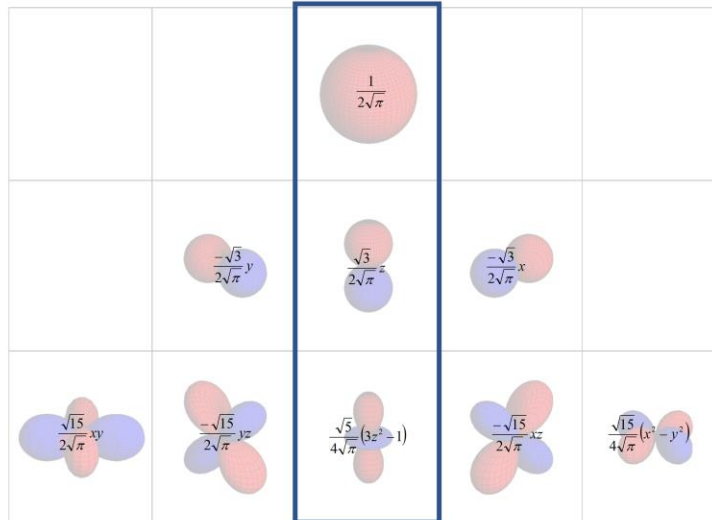


Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH 2017

In Cartesian coordinates the basis functions are simply polynomials. Shader and CPU evaluation code tends to use this representation.

Zonal Harmonics

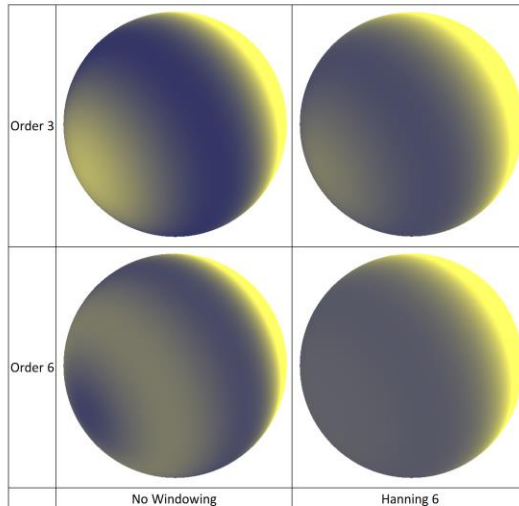


Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

We're going to make use of a subset of the spherical harmonics called "zonal harmonics", shown here in the center column. Zonal harmonics have only 1 basis function per band, and represent polynomials in Z. Any function that has circular symmetry around the Z axis when projected into SH only has non-zero coefficients corresponding to the zonal harmonics.

Ringing SH



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH 2017

Here is an example of color artifacts that can happen. You have a white ambient light and a bright yellow directional light.

The negative lobes pull out red and green, leaving the blue color artifacts. Windowing can get rid of these artifacts.

You also have the positive lobes (order 3) and negative lobes (order 6).

Windowing



- See stupid SH tricks [Sloan08] for various approaches
 - Attenuates high frequencies
 - Can pose as a regularized least squares projection
- Any global value will fix some probes, but hurt others
- Lighting signals should be strictly positive when converted to irradiance
- Apply the same filter to all color channels
 - Compute for each independently, take the most aggressive one
- Approach – find window that makes smallest point positive

There are a bunch of ways to combat ringing, the most straightforward ones are to window the high frequency coefficients. If this is just done globally though, it will over smooth lighting environments that actually aren't having any problems. So we want to window as little as possible to get eliminate the artifacts, in our case make sure that when convolved with a normalized cosine, the function is strictly positive.

One not on windowing is we want to apply a window that is the same for all 3 color channels, so that we don't get any funky coloring artifacts. To do this we compute the window on the color channels independently, and simply take the most conservative one.

For our solution, we want to quickly find the smallest value of the SH on the sphere, and use that routine inside a search over windowing values to guarantee a non-negative result.

In general this is minimizing a quadratic function over the sphere, and there will be lots of local minima, so instead we wanted a simple conservative bound.

Results Old



ACTIVISION
CENTRAL TECH


Here we show an earlier algorithm, that overly blurs the lighting.

Results New



ACTIVISION
CENTRAL TECH

Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

While the new one is strictly positive and has more angular detail.

Insights



- The linear band is **always** a ZH, so rotate to that frame (“optimal linear”)
 - Zeros out two basis functions, SH is the same
- Decompose to ZH + orthogonal complement
- ZH “most negative” is analytic
 - Minima of ZH (after convolution) is trivial to locate – quadratic polynomial in Z
 - Minima is one of $\{-1, 1, -b/(2a)\}$ (last only if $a > 0$)
- Binary search over window until strictly positive

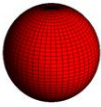



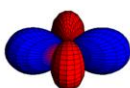
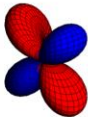

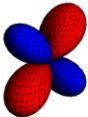

All linear functions have a coordinate system where the SH are just a ZH. The gradient of a linear function in R^3 has a direction, and if we make that Z, it will zero out the x/y SH basis functions, going from 2 to 3.

ZH have circular symmetry, which means they are a function of just Z, so to find the minimum you have a univariate polynomial, which is easy to solve. Given the coefficients of the quadratic polynomial ($ax^2 + bx + c$), the minimum is $-b/(2a)$ only if $a > 0$ (otherwise that’s a maximum) **or** one of the boundary points ($z = +/- 1$).


We can simply do a binary search over windowing parameters, but still need to account for the non ZH part (4 coefficients in this case.)

What about non-ZH?



M=-2	M=-1		<div style="border: 1px solid black; padding: 5px;"> +M / -M are rotated copies Any combination is rotated Conservative bound, f(Z) </div>	
			M=1 	M=2
				

Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

So we have 4 basis functions left, and there is a special property of those that we can exploit as well.

If $|M|$ is a non-zero constant, the basis functions are simply rotations in Z of each other (the angle I believe is $90/L$). This means that if we can reason about the shape of these functions, there is always a phi that has the worst case. So we simply need to turn that into a function of Z.

What about non-ZH

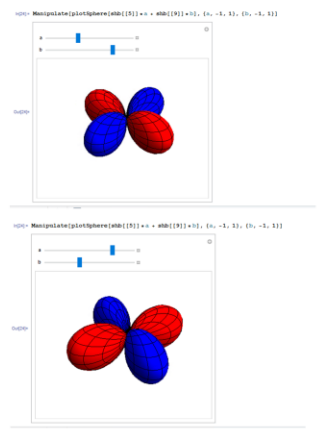


They simply rotate around Z (obvious from Z rotation matrices)

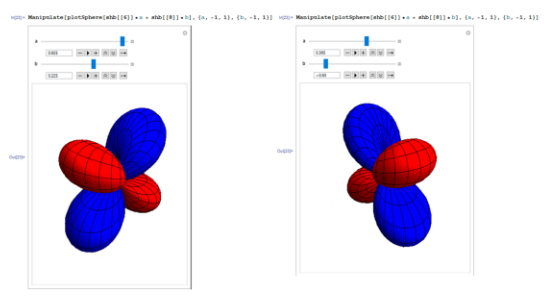
For what phase (phi) is Y(theta,phi) largest as a function of Z?

Minimum will always occur along that cross section, turn it into f(Z)

$|m|=2$



$|m|=1$



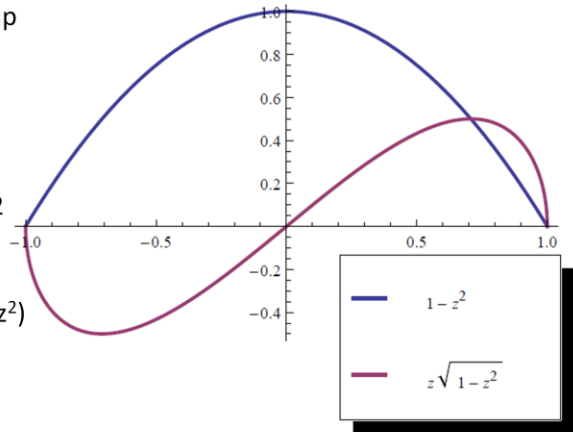
Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH 2017

This property is illustrated for $|m| = 1$ and $|m| = 2$ here, where any blend is clearly a rotation of one shape.

How to lump M's

- Any coefficient pair is a rotated function
 - Find worst case and make them line up
 - Preserve the norm of the pair
- Reparametrize as a function of Z
- $|M|=2$
 - $x^2+y^2+z^2 = 1$ (on sphere)
 - xy when $x=y \Rightarrow 2x^2+z^2=1 \Rightarrow x^2=(1-z^2)/2$
 - Just modify quadratic! This one is easy
- $|M|=1$
 - xz when $y=0 \Rightarrow x^2+z^2=1 \Rightarrow x=\sqrt{1-z^2}$
 - Ugh, tried to bound but not great
 - Just use Newtons method instead



So given the two basis functions for a band, say $|m| = 2$, we simply use a coefficient $\sqrt{b_0^2 + b_1^2}$, and can analyze the worst case for that.

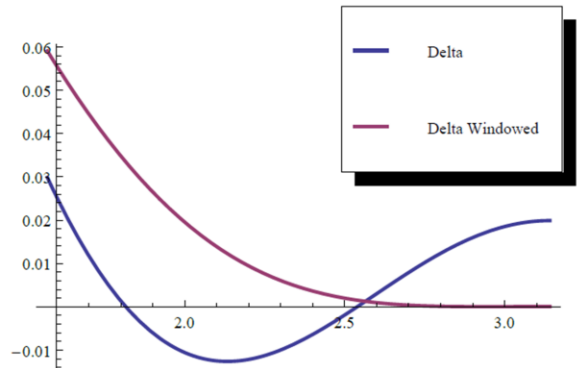
When $|m| = 2$, the worst case happens when $x=y$, so you simply replace the variable, and then make it a function of Z, and you get the blue curve above $(1-z^2)$, you take the negative lumped energy and can minimize this with the ZH function.

When $|m| = 1$, it is a non polynomial. So for this, we simply use newtons method with a good starting point, and we quickly get the minimum. We use a positive sign, since this function is negative in the lower hemisphere (opposite the optimal linear direction), which is where the most negative point tends to be.

Leveraging detecting the most negative value



- No windowing is one extreme
- Delta function is “peakiest” signal
 - Window so is strictly positive
 - Slight boost of linear to do this
- Search to find window that results in non-negative projection



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH 2017

So now that we have determined how to quickly determine if a SH has a negative, we need to come up with a family of windows. There are two extremes:

- 1) No windowing, which is simply a delta function.
- 2) A window function that forces any lighting environment to be strictly positive. For this we take a delta function (the peakiest function in SH), and window it so the tail (visualized in the graph above – from $\pi/2$ to π) is strictly positive. This is a windowed lancosz filter + a small attenuation of the linear SH to make the tail stay positive.

The intermediate points are just other less aggressive window functions from the same family.

What could be improved?



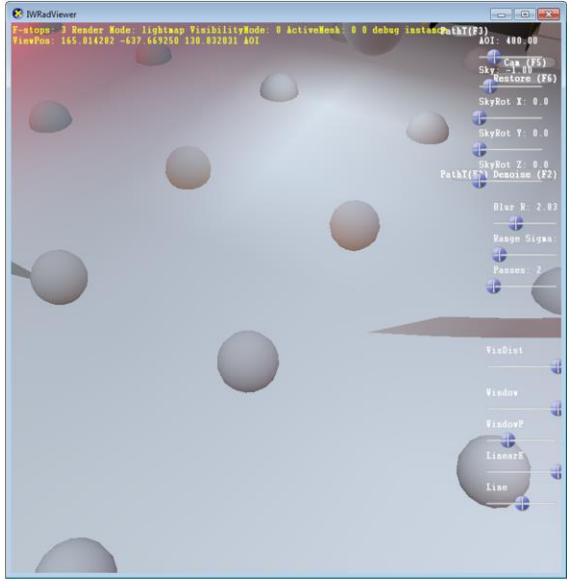
- This only deals with negative lobes, can be large positive lobe
 - Do extra convolution, hand tuned, after evaluating point lights – small amount
- Assumes perfect correlation between $|m| = 1$ and $|m| = 2$ bands
 - Can we down-weight one of them?
- Higher order just comes along (issue for products?)

So this only deals with the negative lobes, there can be rings from positive lobes as well (like the anti-podal location from a point light source), we do a small amount of extra windowing for direct lights before their radiance is added to SH to minimize this.

The only remaining approximation is that the $|m| = 1$ and $|m| = 2$ are perfectly in phase, so have maximal destructive interference. It is unclear if addressing this really helps, since the lower band can still have slight positive rings, but it might be possible to down weight them based on the relative phase that exists.

Finally, this is focused on quadratic SH, for higher orders we simply use the same windowing coefficients just evaluated at higher bands, but minimize using quadratic only. We use cubics internally as well sometimes.

Results Old



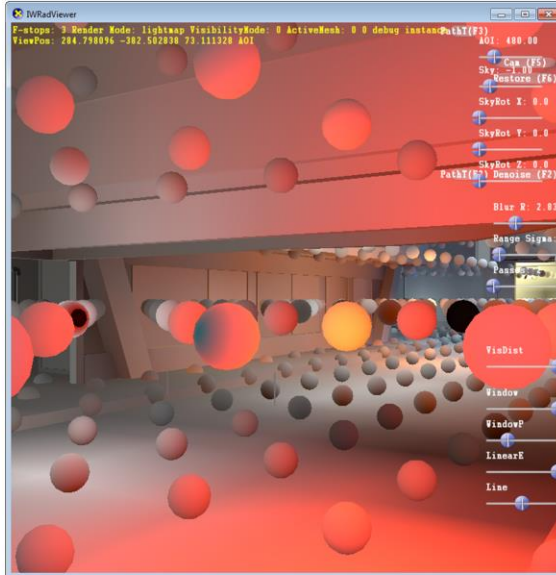
Here is a comparison again, old technique crushes directionality a bit too much.

Results New



New technique maintains it.

Results Old



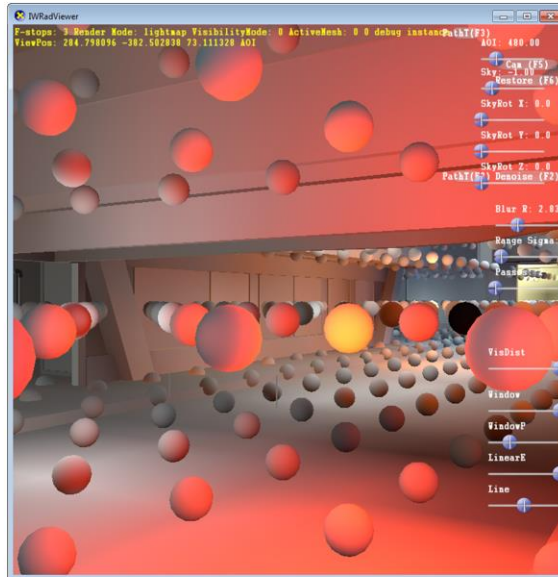
ACTIVISION
CENTRAL TECH

Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017

Here we see an area where the old technique over penalizes some lights, and makes other (black ring on the middle right) actually negative.

Results New



ACTIVISION
CENTRAL TECH

Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017


The new algorithm preserved directionality better, and is strictly positive.

BAKING FOR DECOUPLED LIGHTING



- Can bake
- But where?
- All across the object!

Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

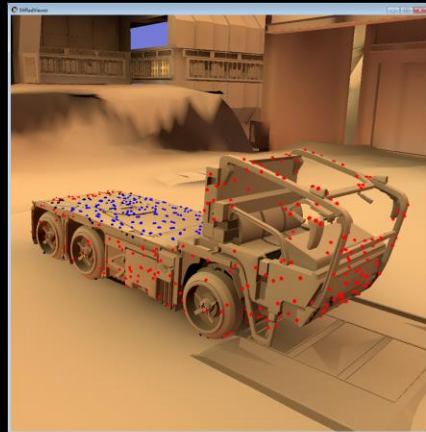
© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

Now we are going to talk about how we bake for static models with decoupled lighting.

We do this by sampling over the surface of the model, and then solving for lighting coefficients that when interpolated, multiplied by self-vis and evaluated at the sample locations minimize squared error.

STATIC GEO SAMPLE SELECTION

- Per Model
 - Generate multiple stratified sample sets in object space, based on number of probes, importance sample based on self-visibility
- Per Instance
 - draw from samples, until you have enough "valid" ones
 - Back face hits, particularly of structural/lightmapped objects
 - Caulk can be placed under ground to validate
 - Geometric and arithmetic mean distances (don't want buried samples)
 - Store bitmask for rays that don't self-intersect (for baking)



We have two phases. First for every model we generate lists of sample candidates in object space. These are done by importance sampling by self-vis over the model (areas that are very occluded are not visible to the player, so have less weight.)

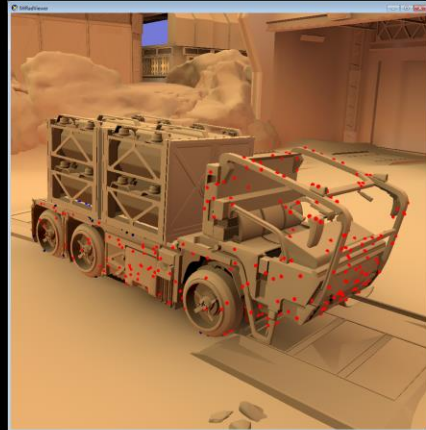
The second phase is per-instance, where we shoot a bunch of “feeler” rays out, to determine if a sample is buried, underground, etc. Lights can optionally place brushed in regions that should never be visible to help invalidate samples, though that’s only occasionally done in practice.

Here the red points are valid, and the blue are invalid.

STATIC GEO SAMPLE SELECTION

ACTIVISION
CENTRAL TECH

- Per Model
 - Generate multiple stratified sample sets in object space, based on number of probes, importance sample based on self-visibility
- Per Instance
 - draw from samples, until you have enough "valid" ones
 - Back face hits, particularly of structural/lightmapped objects
 - Caulk can be placed under ground to validate
 - Geometric and arithmetic mean distances (don't want buried samples)
 - Store bitmask for rays that don't self-intersect (for baking)



Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc. SIGGRAPH2017


When we draw more of the models near this truck, you can see that the points were invalidated because they were underneath the container things on the truck. If we included them, the lighting would be much darker and not match the surrounding areas as well.

STATIC GEO SOLVE



- Least squares solve for the probes so that *reconstructed* lighting – after multiplying with visibility and convolved with a cosine – matches the baked one 9 DOF per sample location
- Visibility product results in mixing of DOF, so can't solve independently
- Angular regularization is important
 - Non-negative DC component, ratio of linear to DC energy must be within bounds of directional lights (so things don't go crazy)
 - Exponential search for regularizer based on squared Laplacian

Advances in Real-Time Rendering in Games course – SIGGRAPH 2017

© 2017 Activision Publishing, Inc.  SIGGRAPH 2017

For the solve, we want to know what coefficients for the lighting, when interpolated to each sample and multiplied by self-vis, would describe the lighting data gathered at that sample.

Each point is then a 9x9 block matrix, where we multiply by self-vis (and convert to irradiance.)

Adding regularization to this solve is important, if you only have points on a plane, the “bottom hemisphere” could have any lighting values, but we don't want it to do anything crazy, since a titled normal could sample it, etc.

We use exponential search and a simple oracle to find the minimum amount of angular regularization (squared Laplacian over the sphere penalty term) that makes the lighting reasonable. Where reasonable has two criteria: 1) DC must be non-negative (negative DC can't happen from real light intensities) 2) the ratio of linear to DC energy must be less than a delta function (the peakiest light possible with SH).

BAKING LIGHT GRID



- We could just bake lighting at the vertices of the tet mesh, but the lighting there is not really what we want
- We want the final lighting, interpolated taking visibility into account to be correct, everywhere on the level
- Least squares again
 - Compute lighting in a number of points in each tet
 - Compute interpolation weight for those points as they are computed during reconstruction, taking visibility into account
 - Solve for lighting at the vertices of the tet mesh that generates that signal as closely as possible
 - Also add Laplacian regularizer (fancy term for penalizing rapid changes between neighboring probes – this also takes visibility into account – if probes don't see each other we don't really care)
- All this acts as a form of denoising/antialiasing – we can reduce the quality of lighting computations on individual samples

Similarly for the light grid, we super-sample the tets spatially, and solve for the lighting at the vertices that when interpolated explains the lighting at these sub-samples, but only the vertices that are directly visible.

We also add a spatial regularizer with a fixed weight – we want to make sure that the resulting lighting is smooth and doesn't have a lot of oscillations.

CONCLUSIONS



- Think of different components of the rendering equation separately
- With more detailed lighting the representations become more complex – resample to simpler structures to mitigate the cost
- Least squares is your friend, but don't forget to regularize

To conclude, splitting the rendering equation into different parts (as others have done) is generally a good idea. Splitting visibility from incident radiance helped improve quality significantly in our case.

Having different representations – heavy weight ones that are sparsely evaluated, combined with light weights ones that are densely evaluated, can be a good thing.

We use least squares a lot, but you almost always need to combine it with some form of regularization.

REFERENCES



- [Annen04] Annen, T, Kautz, J, Durand, T, Seidel, H-P, Spherical Harmonic Gradients for Mid-Range Illumination, EGSR 2004
- [Chen08] Chen, H, "Lighting and Materials of Halo3", GDC 2008
- [Cupisz12] Cupisz, R, "Light Probe Interpolation using Tetrahedral Tesselations", GDC 2012
- [Engelhardt08] Engelhardt, T, Daschbacher, C, "Octahedral Environment Maps", VMV 2008
- [McT04] McTaggart G, Half-life 2 source shading, GDC 2004
- [Lazarov13] Lazarov, D, "Getting More Physical in Call of Duty: Black Ops II", SIGGRAPH PBR Course 2013
- [Iwanicki13] Iwanicki, M, Lighting Technology of "The Last of Us", SIGGRAPH Talk 2013
- [Iwanicki17] Iwanicki, M, Sloan, P-P, Ambient Dice, EGSR 2017, EI&I
- [Schaefer04] Schaefer, S, Hackenberg, J, Warren, J, "Smooth Subdivision of Tetrahedral Meshes", SGP 2004
- [Sloan08] Sloan, P-P, "Stupid Spherical Harmonics Tricks", GDC 2008

ACKNOWLEDGEMENTS



- Central Tech
 - Adrien Dubouchet
 - Manny Ko
 - Dimitar Lazarov
 - Josiah Manson
 - Mike Stark
- Infinity Ward
 - Michał Drobot
 - Olin Georgescu
 - Ifedayo Isibor
 - Kyle McKisic
 - Peter Pon
- High Moon Studios
 - Martin Ecker
 - Stephane Etienne
- Sledgehammer Games
 - Dave Blizzard
 - Danny Chan
 - Stephanus
 - Luka Romel
 - Atsushi Seo

Before i start i wanted to say that *a lot* of people contributed to the ideas we will present here and we wanted to thank them all for that