The Indirect Lighting Pipeline of God of War

Josh Hobson

1st Party studio, part of Sony Playstation
Responsible for the God of War series
This talk will cover high-level end-to-end pipeline with a focus on concepts, motivations, situations, and some technical depth. Where possible, I will integrate dates and behind the scenes info.
What this talk is NOT

- Theory or proofs
- Code
- Implementation Instructions

It’s not intended to be implementation instructions or proofs. While we do bring some new ideas to the table, this is largely built on existing work.
First let’s go over some terminology.
All level editing is done in maya there is no level editor
Wad - Streamable unit
- scene heirarchy
- Composed of a series of overlaid maya files
- heap/object lifetime (this is important for reliability/repeatability since there’s never a full teardown)

Wads contain “Refnodes” serve as general purpose links for
- Prefab workflow - object instances
- Level editing workflow layering (art/design, subdivision, vis, lights, entities)
- These can be recursive
Hallmark of the series - no loading screens
- Playable end to end, no discrete “levels”
- Conveyor belt loading
- What we load and when is data driven by level design and art
- Design/art responsible for breakdown of wads
- S-hallways, traversal pacing to hide/anticipate loading ahead and unloading behind

Wads can be stacked
- Kratos’ house is a wad that’s contained in the surrounding forest wad
In mid-2015, at our first playable, we had an indirect lighting pipeline that used lightmaps and loosely placed diffuse probes. It’s not an uncommon solution but it comes with tradeoffs. There’s a visual disparity between dynamic and static objects that’s reminiscent of old cartoons where you can tell something is about to move because it’s cell-painted in the animated layer instead of the matte painted background. Additionally there’s a complex pairing of lightmaps to instances. This is further complicated due to the generic and recursive nature of our refnodes. For example you might have an object on a table, the table could be instanced several times in a house and then the house might be instanced several times. The number of instances can explode quickly. Keeping track of what is and isn’t up to date is complex. UVs and probe placement is laborious. Changing mesh topology requires new UVs and rebaking. Moving lights and objects can require re-placing probes. All of this can be helped by tools but it’s still time consuming and painful.
We wanted to try something different. I’d used 3D textures for lighting in the previous console generation and it was affordable. Consistent look for static and dynamic objects. We have a relatively small lighting team, we want them focused on lighting not data management.
Keeping it simple, GI volumes are loosely placed boxes in Maya. Inside there’s a series of 3D textures that contain lighting baked data. Since we’re only storing indirect lighting resolution does not need to be high. Even larger than meter per voxel can be enough.
Four 3D textures

- 2 band spherical harmonics
- RGB bounce and sky visibility + monochrome bounce (1)
- Float16 (more later)

(1) Gilibert and Stefanov "Deferred Radiance Transfer Volumes – Global Illumination in Far Cry 3", GDC 2012

4 3D SH textures
RGB – bounce lighting from analytic light sources
Alpha – sky vis + monochrome bounce
Test scene
Here's an example of a few slices from the GI volume. Splitting sky and bounce helps SH encoding maintain directionality. With 2-band SH opposing directions cancel each other out, leaving only the constant 0-band to represent the data. Sky comes largely from above and bounce from below.
You can think of the sky visibility term like a directional ao. The sky lighting is joined at runtime, this means we can change the sky without rebaking. We express sky lighting as a SH encoding of cubemap, generated during build.
We do change sky lighting/rotate at runtime, mostly during transitions.
4 closest on screen
Sort by authored priority
We want this simple on the GPU. Shader walks in order, stops on first intersection
We’ll talk a little about optimizations shortly.
Because the voxel density is low we get self occlusion and light leaking artifacts. This is a similar effect as shadow acne. We’re motivated to use hw filtering and avoid unrolling the 8 taps for manual filtering. To solve this we simply offset the GI sample away from the receiving surface along the normal. This effectively inflates objects when sampling the GI so they’re sampling outside of themselves.
Sample offset on moving object reads like specular, looks shiny. We already had a bit in our gbuffer for characters. Don’t apply offset for characters (they didn’t participate in the bake anyway).
Video showing character GI looking like spec when moving because of normal offset. Real-time shader update shows problem go away on Kratos.
There’s a literal corner case where close-by normals cause the normal offset to poke through walls or the floor revealing un-occluded sky.
Light Leaking – Corner Case

Glowing from sampling below plane
Over occlusion more acceptable than under
Box is not great for organic environments
-we intend to investigate alternatives going forward
Cubemaps are sparse compared to the GI, which means spec lighting can leak into occluded areas.
Cubemap Normalization
Cubemap Normalization

Objectives

- Keep a natural balance between diffuse and specular ambient lighting
- Use cubemaps for angular detail and GI for spatial detail.

A natural balance between diffuse and spec is more important than pixel correct reflections.
Metals need to benefit from GI bake
Ideally should leverage cubemap angular detail + GI spatial detail.
Cubemap Normalization

Generate spherical harmonics from cubemap during build

Use spherical harmonics to remove low frequency detail

Replace with low frequency detail from GI
Again, the composition of reflections may be incorrect but the intensity/balance should be correct. This lets us get away with fewer cubemaps overall.
Example scene. Highlighted block and overhang occlude the corner of the hallway.
Example scene. Highlighted block and overhang occlude the corner of the hallway.
Cubemap sees the block, not the dark corner.
Original example.
Normalization on. Clearly contents of cubemap still incorrect (chrome ball) but the intensities are more correct.
Hard to tell they’re behind the block
Spec diffuse balance is way off (upper left sphere)
Notice the chrome ball is brighter where reflections point out of the corner like you’d expect.  
Note: The corner on the left of the image isn’t water-tight, there’s an opening about the size of a tennis ball letting light in.
The back wall looks wet (remember there's an overhang above).
Normalization on.
To keep ourselves honest, you can see that when the objects get closer to the cubemap capture location the effects of normalization lessen. These aren’t perfectly on the capture location, and the change is small. This is what you would expect because as the cubemap and the GI agree on what they see, they essentially cancel each other out.
This wasn’t really based on any particular paper however it ends up being pretty much identical to the call of duty technique (lazarov, 2013). I’ve spoken with other studios who’ve tried this (or similar) and some reported abandoning it because they didn’t like the results.
To fix the issues we reduce directionality and saturation in cubemap SH (denominator/div by 0).
If you have problems resulting from particles or transparencies in your cubemap (that don’t participate in the bake), you could probably use the GI at the cubemap capture location instead of SH from the cubemap as the denominator. We considered this but never ended up needing to try it.
Here's an example of the discoloration that results from clamping and channel separation.
We used the ragdoll capsules for our character AO. It worked well but AO ended up having competing goals with ragdoll and we narrowly cleared the AO limit during finaling. Ragdoll can inflate capsules for various reasons, and potentially add extra to avoid falling through the world. We plan to make this a custom pipeline going forward.

Ambient Occlusion
SSAO
AO maps
Character AO capsules similar to the Last of Us constant + directional term (4)
For God of War we used the ragdoll capsules
- Almost ran over limit late in production
- Will be custom going forward

Add capsules
Add ssao
Add ao maps
Volumetric Fog

Fog uses Bart Wronski’s technique. (5)

GI volumes (0 band only) are sampled when constructing the lighting texture.

Particles use fog lighting texture for cheaper lighting. Get GI for free. Bart’s idea.

Consistency is important so the fog also uses the GI. The fog uses only band0 (single texture fetch) of the GI as an optimization we made for the e3 2016 reveal of the game. We used to store a texture per channel, had to reswizzle data to be a texture per band.

Optimization

Accelerated as part of our tiled lighting. Each 8x8 tile has a mask of potentially intersecting GI volumes, loaded as SGPR.

Runs in parallel with shadow rendering on async compute. Shader applies both ambient diffuse and spec.

GI and cubemaps are applied together in one pass.
Nova was developed in the PS3 generation. It has shipped lots of great games and I believe is still used.

Baking Background

Nova bakes (e3 2016)

- 1st party tool developed by Sony research ATG
- Carried over from lightmap and probe workflow
- Good starting point, known quantity
...But it wasn’t great for iteration.
- It has its own data format that is principally redundant to ours and is built only for Nova.
- The material conversion is complex for our materials. They were baked to vertex colors as an optimization.
- Are the results correct? Does Nova model their lights the same as we do? Spotlights were problematic for this reason. It was hard to verify that you’ve converted your parameters correctly. Left us to verify visually (which isn’t trustworthy).

Additionally
- Scattering/translucent materials were possible with Nova but another tricky conversion.
- No volumetric fog.
The Nova pipeline largely relied on automated testing to stay functioning. The (small) lighting team was the only group really using the tool and it was easy for even seemingly benign material changes to break the Nova pipeline.

Baking in-Engine Motivation

Leverage built data
• Lighters already build the levels to see results
• Custom building for light baking is wasteful
• Fewer pipelines -> fewer points of failure

Real-time results
• Lighting workflow is iterative
• We already support live-update changes from Maya
I get asked why we bake on the PS4 instead of the PC.

- PC does share some but not all data, so it would be an extra build
- We were trying to get the lighters off their own support island. Our PC build isn’t really used in any workflow so it, like Nova relies on automated testing to stay functioning.
- Generally, we would rather have everyone looking at results on PS4 (it’s what we’re shipping after all)
- Most importantly, we could have abandoned PC build if the maintenance became more than we could afford. It wasn’t critical to ship the game.

Caldera (“The Lake of Nine”) ended up only baking on PS4Pro because it required extra debug memory to bake.
Baking in Engine

“Fast Global Illumination Baking via Ray-Bundles” (6)

• Early rays have fewer bounces, converges on multi-bounce.

• Rather than looping over rays for every point, loop over points for every ray.

Recommendation from Bart Wronski. Ideas borrowed from Stephen Hill.


Credit to Stephen Hill for adaptation for GI volumes
The first step is to shrink-wrap all the geometry into a cloud of surfels, each containing associated materials properties like normal, position, albedo, etc. To do this we hijack the rasterizer, rendering with no render or depth targets and no back-face culling. It simply samples triangles on a regular grid for us. We use atomics to collect the surfels as we go into one list.

We have a lot of shaders so adding a GI bake permutation would slow the build down or create an ugly build-the-level-for-GI-baking workflow. Fortunately, we already had a debug shader permutation that we use for a variety of debug/instrumentation views. This shader obviously doesn’t need to be performant so it was a convenient place to add a static branch for emitting the surfels.
Example: creating surfels on each axis
Again, but viewing normals.

This is the first of several times that I’m going to stress how important it is to build good debugging in _while_ developing a large feature like this.
Accumulate lighting in surfels rather than lightmaps, credit again to Stephen Hill.

Light Injection

Original technique accumulated intermediate results in lightmaps

- This is done in the persistent set of surfels instead
- Lighting holds emissive when surfel is created

Direct light is evaluated for every surfel for every light (added to emissive)

- Uses the same lighting calculations/models as the main render
Ray Processing

N rays

• Project surfels into linked lists
• Sort the lists
• Walk list, transfer light between surfels
• Resolve to GI volume
For each ray we build an orthographic projection around the area we care about. We project each surfel to a texture where each texel is the head pointer of a linked list. We use an atomic swap to append new surfel as the new head.
Next we sort the rays using a bubble sort. It was easy to implement and validate. It’s surprisingly fast and never became the highest priority to optimize.
Again it's extremely important to build good debug rendering. Here’s a view to help validate the sort order of lists. It renders directly from the surfel cloud with links between them. Improperly sorted links show up as red. Head surfels are green, tails are blue, otherwise cyan.
How could you validate 100k linked lists over 1000's of rays without it?
Now that the lists are sorted we need to exchange lighting. Simply walk the list and treat neighbors as light sources. Back-facing surfels are black. Head surfel sees unoccluded sky, sky is accumulated separately from lighting.
Lastly, we take the voxel center, project it into the list and walk until it’s between two surfels. Then we simply encode the lit surfel as SH. Then we progress to the next ray.
Video showing the bake in GI view. Less than 14 seconds (realtime) to bake 512 rays, but the image stabilizes in far fewer rays. Artists can cancel, move objects/lights and restart. You can see how this would allow them to quickly iterate.
GI Baking Execution

State machine
Schedules GPU work by inserting passes into the renderer
Issues multiple state updates a frame (all necessary states are buffered)
Targets 80ms a frame. Assumes GPU time scales linearly with passes, adjusts accordingly.

80ms target means most of the frame is spent on baking but maintains high enough framerate to be responsive.
Objects like doors and breakables would be excluded from bake. Lights generally should participate except accent lights in cinematics and potentially special fx. For the e3 2016 demo there were several different suns with different angles and intensities to sell the progression of time. Transitions were strategically hidden. To avoid seams in the bakes, all suns were marked as do-not-bake and a single bake-only sun was added from an average direction and intensity.

GI volume resolution will be based on ideal meters per voxel up until it reaches the memory budget, in which case it will reduce resolution to fit.
Entire categories of lights can be disabled during the bake for convenience. For example lights marked as special fx could be disabled during the bake.
Level from beginning of the game
Original troll arena in e3 2016
Load additional wad
Quick bake
(realtime)
This video shows surfel debug mode, lighting, sky vis, albedo, normal. Baking progress while in surfel debug mode.

(realtime)
We support single step debugging through the bake. Here I single step while in linked list debug mode.
Here’s an example of a more typical iterative workflow. Lights and objects being moved around several times re-baking along the way. You can see the surfels update after the sphere moves, there’s no precomputation of the scene required. Bakes are easily cancelled and restarted. You can see that bright/small lighting causes fireflies initially but it smooths out with more rays. (realtime)
As I mentioned before, having fog participate in the bake is one of the motivators for moving away from Nova. Fog can be a significant light source in a scene. For this Bart modified his technique to use an orthographic projection for the lighting and scattering textures around the GI volume. We use a cubemap for anything outside.
We modify the light transfer to sample the fog textures, taking multiple taps to accumulate extinction and in-scattering.
For every ray direction (cont’d)

- Apply extinction and in-scattering between voxel center and surfel
- Variable num taps, fixed step size.

For resolve we do the same thing but from voxel center to surfel
Scene, no fog
Add fog
No fog in bake
Fog in bake
No fog in bake
With fog in bake
-Directionality on block at the left comes from the column of light, quite a drastic change.
Thin (2-sided) we add 2 surfels at the same location. We use some bit hacking to guarantee proper sort ordering for each ray.
Block of ice + spot light
Variable Resolution
Variable Resolution Motivation

Lighting is relatively constant in open spaces
Lighting varies most near surfaces
It would be nice to redistribute resolution near surfaces
Can’t explode bake times, ideally bakes still scale with sampling density

Indirect lighting changes are low frequency in open air, away from objects.
Variable resolution uses essentially a 3D virtual texture. This is a technique I’d developed in the previous generation. Indirection texture is usually 4 meters per voxel, sometimes larger. Each indirection voxel can be backed by a different sized block in the atlas texture.
On the right you see an example section of an atlas texture, scrambled blocks of different resolutions packed together. On the left you see a piece of the indirection texture, containing location and size information for the blocks in the atlas texture. The indirection texture’s job essentially is to unscramble the atlas.
Determining Resolution

After surfels are collected, before lighting

Small number of rays to calc average distance from indirection voxels. (< 10 sec)

Read back on the cpu.

Start with indirection voxels as 1x1x1.

Promote the voxel that reduces error the most. Repeat until memory budget has been met.

Run over a small number of rays (256-512) at indirection resolution to determine the average-nearest-distance to a surfel from the indirection voxel center in all directions. Then we use a greedy algorithm starting with all voxels as 1x1x1 that promotes the voxel that we estimate reduces error the most based on the average-nearest-distance. We repeat this until we’ve spent our memory budget. This allows us to know our final resolution _before_ we do any baking.
So how do we estimate error? There wasn’t a lot of time to derive a perfect mapping so we went with something empirical. The basic idea is we assume all surfaces have high frequency detail. If you move an observer point that is close to the surface perpendicularly the observed error would be large. Whereas an observer far away moving that same distance would have a small error. So we use that parallax angle as our error metric.
Here's a debug visualization showing voxel resolution. This GI volume is uniform.
A variable resolution GI volume using the same memory budget.
Here's a video to hopefully help explain the technique using that same debug view. I'll sweep a plane across the volume so you can see the resolution change in open spaces. Colder colors are lower res, warmer are higher res.
Again, time was limited so there are probably better solutions for packing the atlas texture. This is essentially a 3D pooled allocator for 32x32x32 sections. Each 32x32x32 section is filled as tightly as it can be with the blocks remaining. Then, we simply find the best dimensions for the texture based on the number of sections. In the ideal case the number of sections is a cube, this is never the case so we search the possible width and height options to find one that minimizes waste.
Now that we know the final layout of the atlas texture we can actually bake. Only the resolve stage of the baking changes with variable res. We need some way of mapping atlas voxels to world positions. With uniform volumes we break the voxels into 4x4x4 wavefronts. With variable res we break the individual blocks into 4x4x4 wavefronts too. However, anything that’s not a multiple of 4 will have wasted lanes. This didn’t end up being a significant performance hit.
Keeping hardware filtering is even more important now with the indirection texture, it too would unroll to 8 samples + indirection logic for each + 8 atlas taps. So we treat voxel centers as box corners, this means that we transition to the next block before we’d start filtering outside our current block (and from sampling the random neighboring block in the atlas texture).

Seams should only occur where sampling density is lower than lighting frequency and where neighbors have different resolutions
- Under-sampling means error estimate failed or we ran out of memory
- Going forward I’d like to experiment with using real lighting information to drive the resolution. Maybe replacing the average-nearest-depth pass with instead a lighting variance over the voxel. This would also avoid wasting samples close to objects that don’t actually have interesting lighting features.

Block Borders

Voxel centers are corners. Essentially a voxel border.
- Allows hardware filtering.

Seams? Only where under-sampled and neighbors differ in resolution

Better error estimator should help
- Ideally closer match in sampling res and signal frequency.
Precompute solver for each pair of neighbor options. Then run across each axis separately. Corners and edges have multiple constraints so we have to run multiple iterations.
Can take minutes to compute on the CPU.
There are better options but we ran out of time.
Before fixup
After fixup
Here's a video of me sweeping that plane across the volume again, this time with the GI results.
Variable Resolution


Optional and opt-in

- Helped trim memory or improve visuals depending on the level.
- Small divergent branch to alter UVs used when applying GI volume
Uniform
Variable res with the same memory budget
We prioritized variable res over compression because it seemed higher risk and the window to implement would close sooner than compression. We could do compression really late right?

Total energy of SH is represented in the constant term, can be used to normalize second band. Store only constant (band0) as float, and the second band as ldr.
After. Whoops
What happened? Well, technically the interpolation changes with this compression. Normally this wouldn’t be an issue, light maps tend to be higher res, in fact Frostbite presented only a month or so after we’d scrapped it and moved on. For them this approach worked great for lightmaps. So I believe the problem is two fold 1) resolution and 2) high contrast lighting changes from inside and outside of the object coupled with that we’d tuned the normal offset distance down to prevent some over-darkening artifacts which I believe left us with a subtle amount of self-occlusion. When the interpolation changed it exaggerated that self-occlusion. We’ll do some more investigation into this, but at a month before gold we didn’t have time to open back up the normal offset again. Fortunately perf and memory were ok. Disk space was our main motivator to get compression in.
So we simply min/max compressed the second band instead. We tested lots of levels and found that banding artifacts were not that common. So, it was defaulted on and the lighters were gracious enough to go back through and verify wads. They turned off compression where there were banding artifacts. Savings (and bake times) were worth it.
Runtime numbers are estimates (removing portions of the combined shader), there are interaction effects between cubemaps and GI. The AO number isn’t the total AO time, just the portion of the combined shader that applying it cost.

Bake times: 15mins is the more typical worst case. Lake of nine was the absolute worst case variable res, it took 20 mins.
Stats

GI Volumes
- 300 total
- 80 variable res
- 250 compressed

Disc
- 2.9 GB for GI
- 30% total savings by compression
Thanks

Bart Wronski
Stephen Hill
Santa Monica Studio Rendering Team
Santa Monica Studio Lighting Team
References

(1) Gilabert and Stefanov "Deferred Radiance Transfer Volumes – Global Illumination in Far Cry 3", GDC 2012

(2) LaGarde and Zanuttini, "Local Image-based Lighting With Parallax-corrected Cubemaps", Siggraph 2012

(3) Lazarov, "Getting More Physical in Call of Duty: Black Ops II", Siggraph 2013

(4) Iwanicki, "Lighting Technology of 'The Last of Us'", Siggraph 2013


(8) Hazel, "Converting SH Radiance to Irradiance", 2017

(9) O'Donnell, "Precomputed Global Illumination in Frostbite", GDC 2018
We’re hiring for what’s next!

We’re expanding our family across disciplines and would love to meet you. Please visit sms.playstation.com/careers for all openings or drop us a line at sms@sony.com.
Thank you!
josh_hobson@playstation.sony.com

*SIE Worldwide Studios*
Blending Between GI Volumes

GI Volume bakes were sometimes manual

- Some setups involved manually altering the setup before baking
- For that reason we could not automatically re-bake the whole game.
- Having all special setups saved as data going forward is a goal so we can automate.

No blending between volumes at run time.

Offline tool identifies overlaps

- Resamples lower priority volume from higher priority.
- Bilinear smooths edges around borders.

Offline blending only
- Where there’s overlap, lower priority GI volumes sample higher priority and replace contents