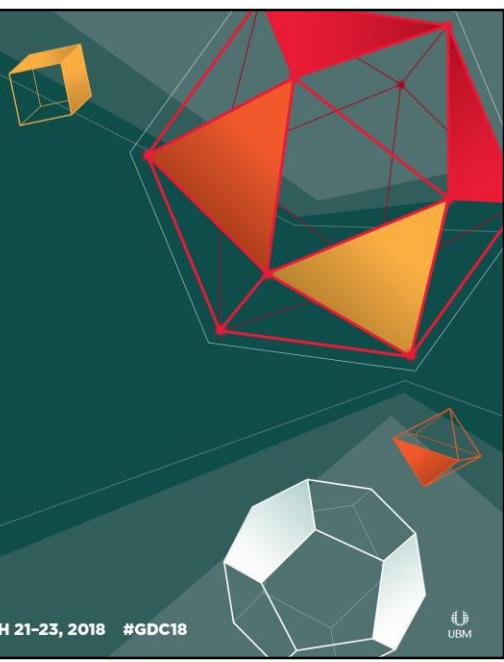




Precomputed Global Illumination in Frostbite

Yuriy O'Donnell
Rendering Engineer

GAME DEVELOPERS CONFERENCE® | MARCH 19–23, 2018 | EXPO: MARCH 21–23, 2018 #GDC18



Agenda

- Introduction
- Part I: Path-traced spherical harmonics lightmaps in Frostbite
 - Motivation
 - Diffuse lighting
 - Efficient encoding
 - Approximate specular lighting
- Part II: A bag of tricks
 - Hemisphere and texel sampling
 - Rendering convergence detection
 - Dealing with overlapping geometry
 - Ensuring correct bilinear interpolation
 - Efficient lightmap atlas packing



Introduction



Flux: Frostbite path tracer

- High quality static lighting
- Lightmaps and irradiance volumes
 - Direct and indirect lighting
 - Static ambient occlusion, sky visibility
- Initially created for FIFA and Madden
- Now used in Star Wars Battlefront II



FROSTBITE

A little bit of history.

We have developed our own lightmap baking solution during FIFA17 Frostbite transition. Previously, FIFA artists used Maya to bake night-time lightmaps for their stadiums. We needed a solution that was more integrated with the rest of the engine. Additionally, we wanted to have something that we can maintain and extend according to the needs of our projects.

When Star Wars Battlefront II started, we decided that our baker could be used for the project to achieve better lighting quality than what was possible in Battlefront I.

Flux method

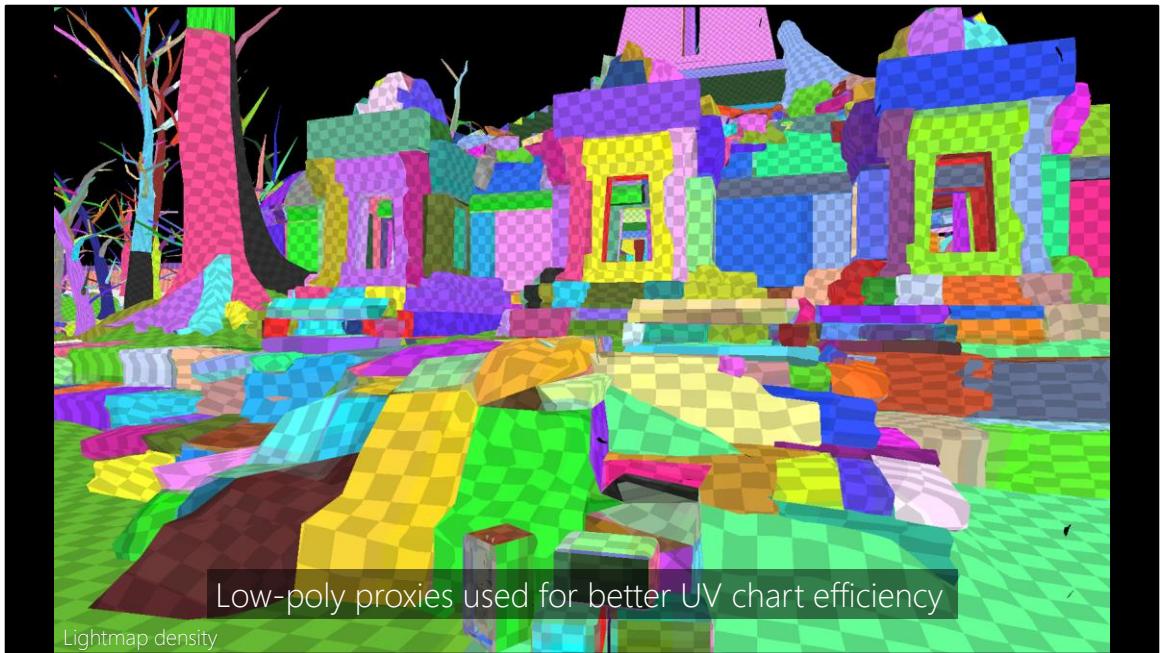
- Unidirectional path tracing with next event estimation
 - Brute force
- CPU implementation for baking
 - Intel Embree
 - Incredibuild XGE
- GPU implementation for real-time artist workflows
 - See [Interactive Global Illumination in Frostbite](#) [Hillaire18]



Our lightmap baker, **Flux**, is a brute force (naïve) unidirectional path tracer that uses basic BRDF importance sampling and next event estimation (explicit light source sampling).

It is primarily a CPU-based solution, which uses Intel Embree for BVH construction/traversal and Incredibuild XGE for distributing computations over machines in our studios.

We have recently also implemented a GPU back-end that's based on a pre-release version of RTX. It's aimed at improving artist workflows by providing a fully interactive global illumination authoring system that allows full scene editing (geometry, materials, lights). We have presented some of the implementation details of our GPU back-end at GDC2018.



Let's look at some of the inputs and outputs of our lightmap renderer.

Frostbite was a very early adopter of Geomerics Enlighten [Martin10] middleware, which is used in most of our games to provide semi-dynamic GI and interactive lighting artist workflows. Having full compatibility of the assets and essentially zero workflow changes was one of the design goals of our custom baker, so that it could be a trivial drop-in replacement for teams that only need static GI.

We use low-poly proxy geometry during baking. Proxies are created for all mesh assets that require lightmaps. A low-LOD version of the asset may be sometimes used, though in general, creating good proxies is still a pretty labor-intensive process for artists.

Lightmap UVs are created only for proxies (either manually or using Enlighten auto-UV library). Having low-poly version of geometry is quite desirable when it comes to lightmap UV efficiency, as low-poly meshes typically have much fewer UV charts/islands and therefore fewer lightmap texels are wasted.



Ideally, each lightmap texel should be representing as a single plane in the mesh. If this is not the case (i.e. when geometry is much higher resolution / high frequency), some artifacts may appear due to self-shadowing.

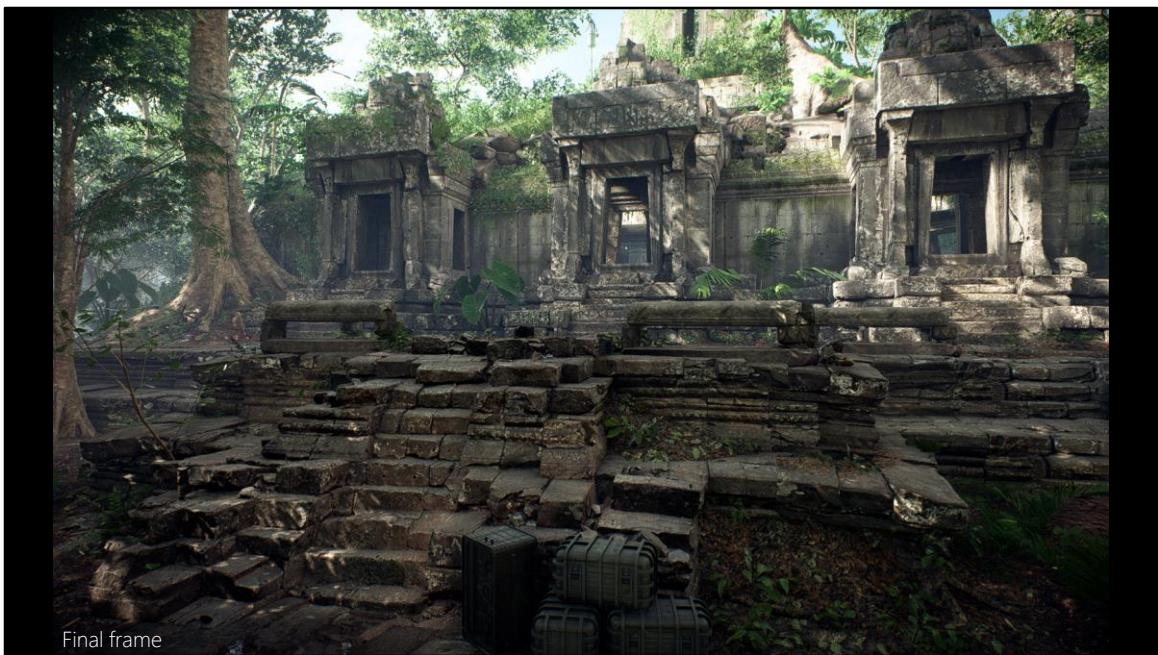


Lightmap UVs are projected from proxies to all LODs of the final display geometry in the mesh pipeline. Frostbite uses Enlighten SDK to perform lightmap UV projection.

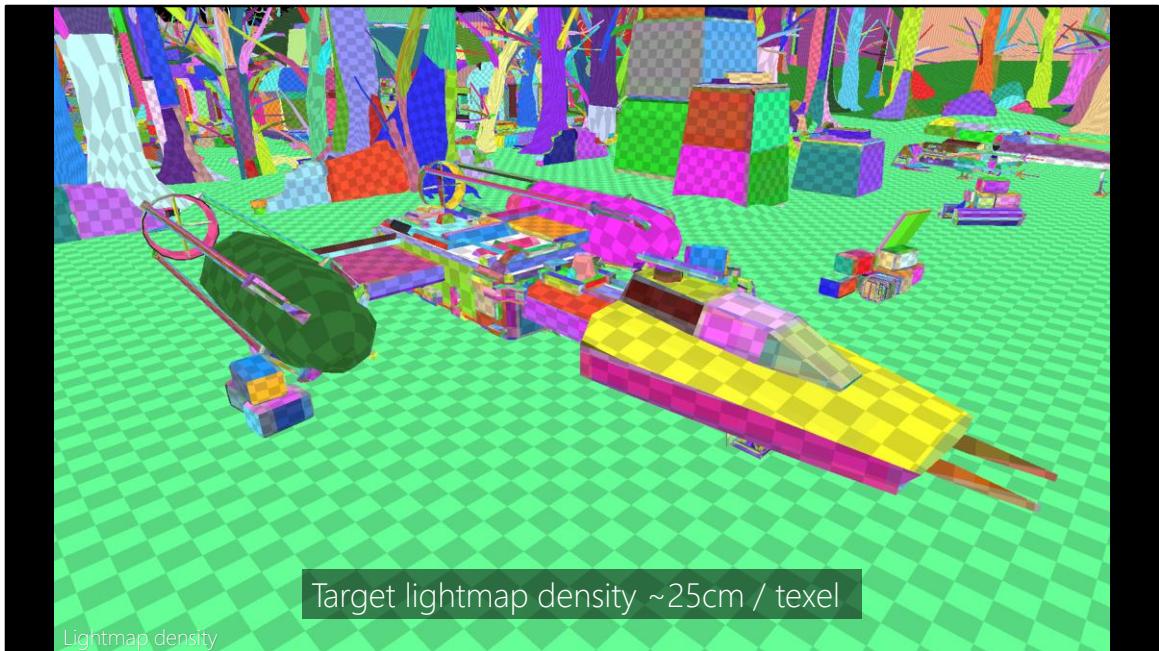


Direct + indirect lighting, final geometry

Lightmaps in SWBF2 only contain indirect lighting. All direct contributions (except sky) are computed dynamically.



Final frame



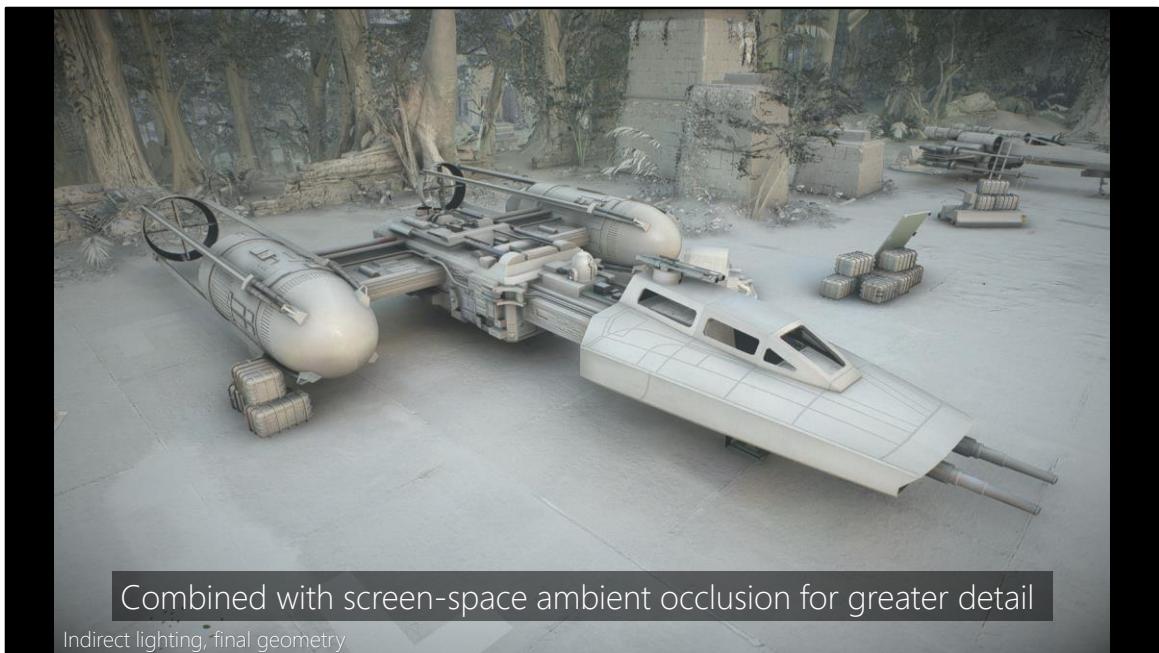
Target lightmap density for Star Wars Battlefront was ~50cm / texel. Lighting density for Star Wars Battlefront II was 25cm / texel (~4x more data).



Baked indirect lighting is fairly coarse

Indirect lighting, low-poly GI scene

Baked undirect lighting was fairly coarse. It was augmented with a screen-space technique (Horizon-based Ambient Occlusion).



Combined with screen-space ambient occlusion for greater detail

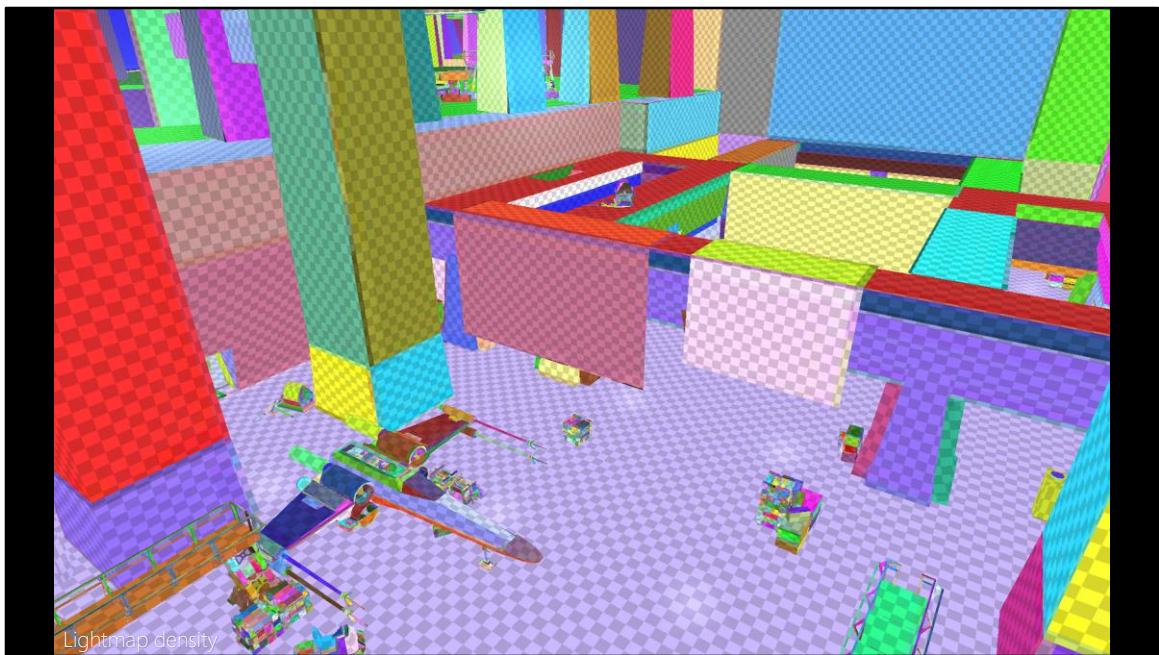
Indirect lighting, final geometry



HBAO adds some short-range, high-frequency lighting detail that's missing from lightmaps.



Final frame



Lightmap density

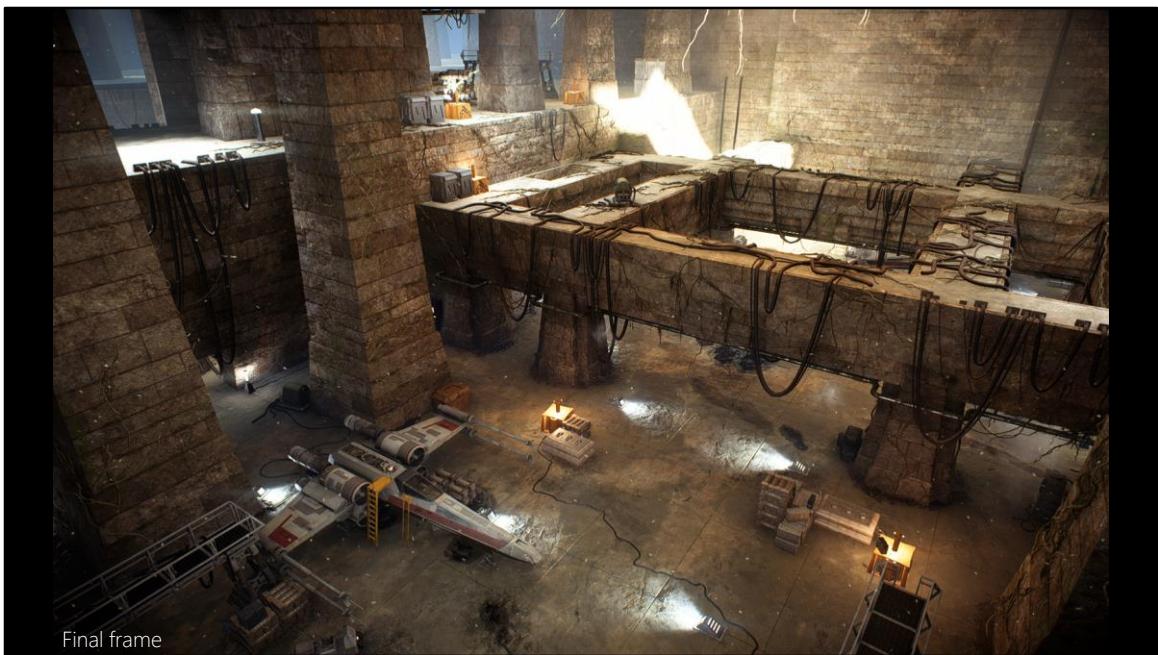


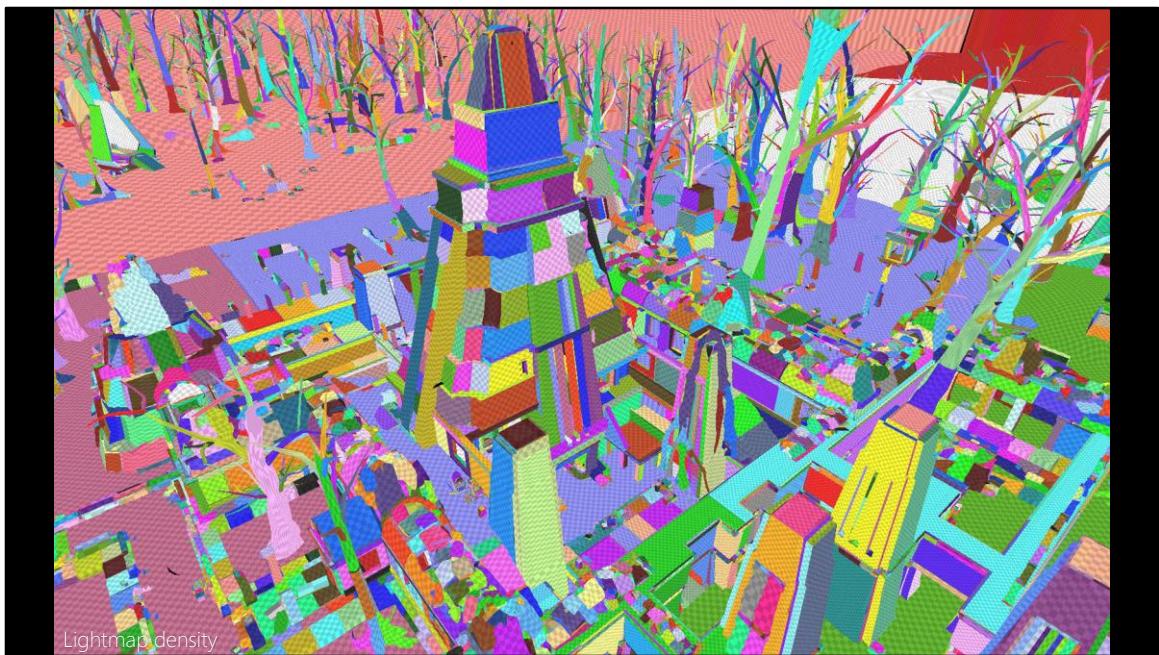
Indirect lighting, low-poly GI scene



Indirect lighting, final geometry







Lightmap density



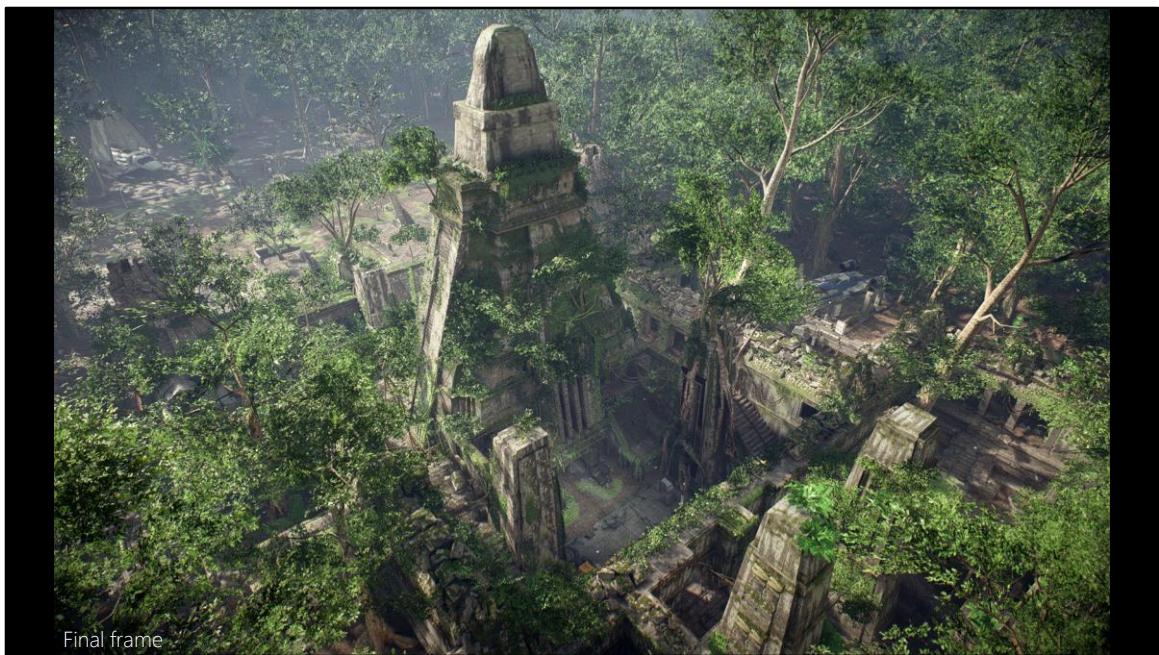
Indirect lighting, low-poly GI scene



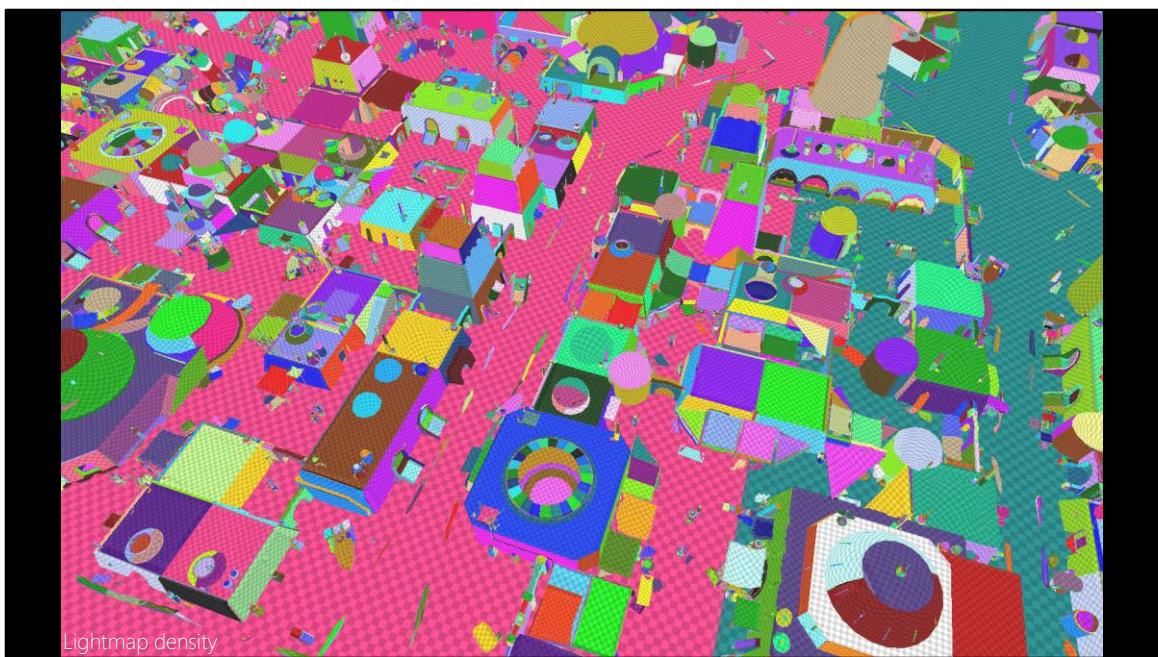
Indirect lighting, final geometry

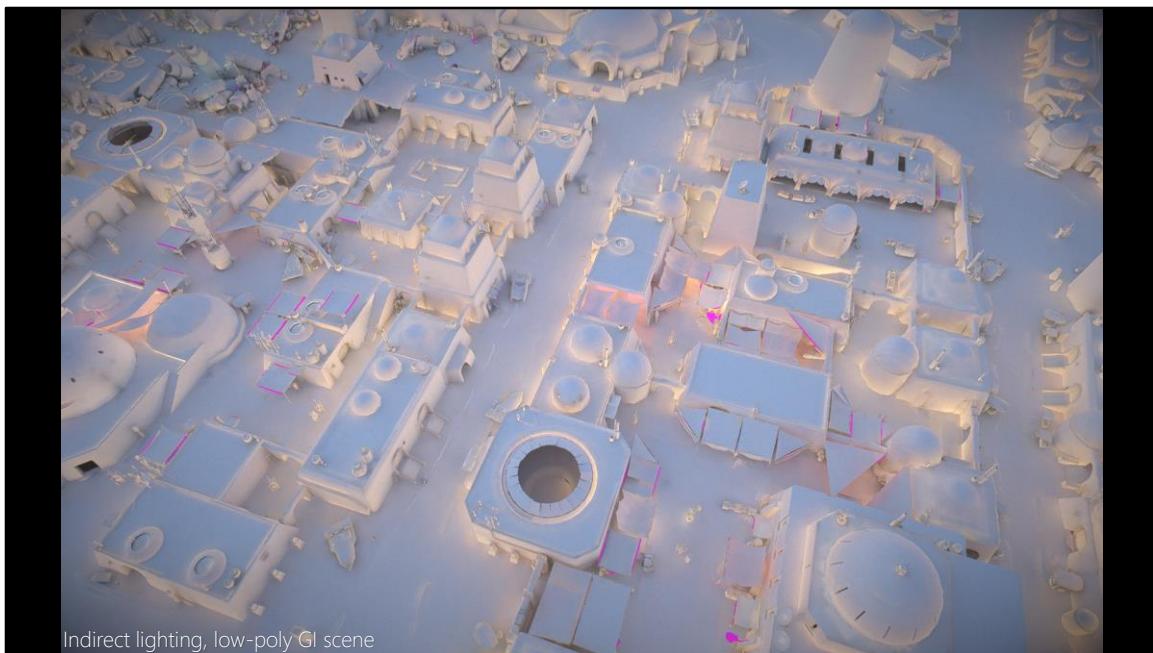


Direct + indirect lighting, final geometry

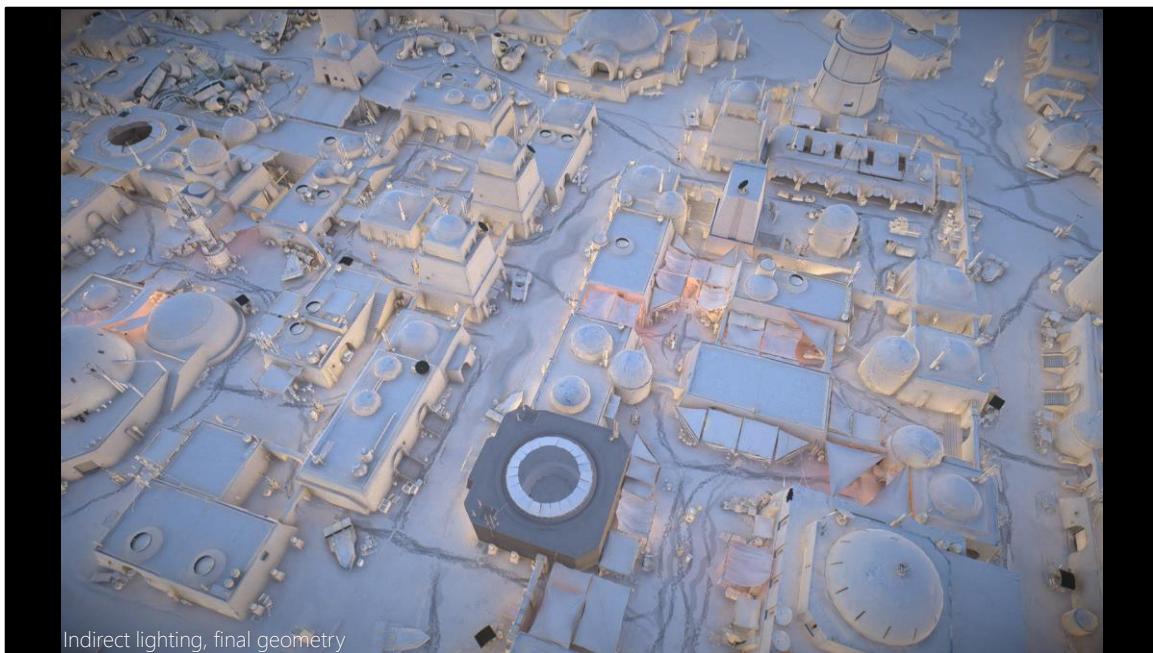


Final frame





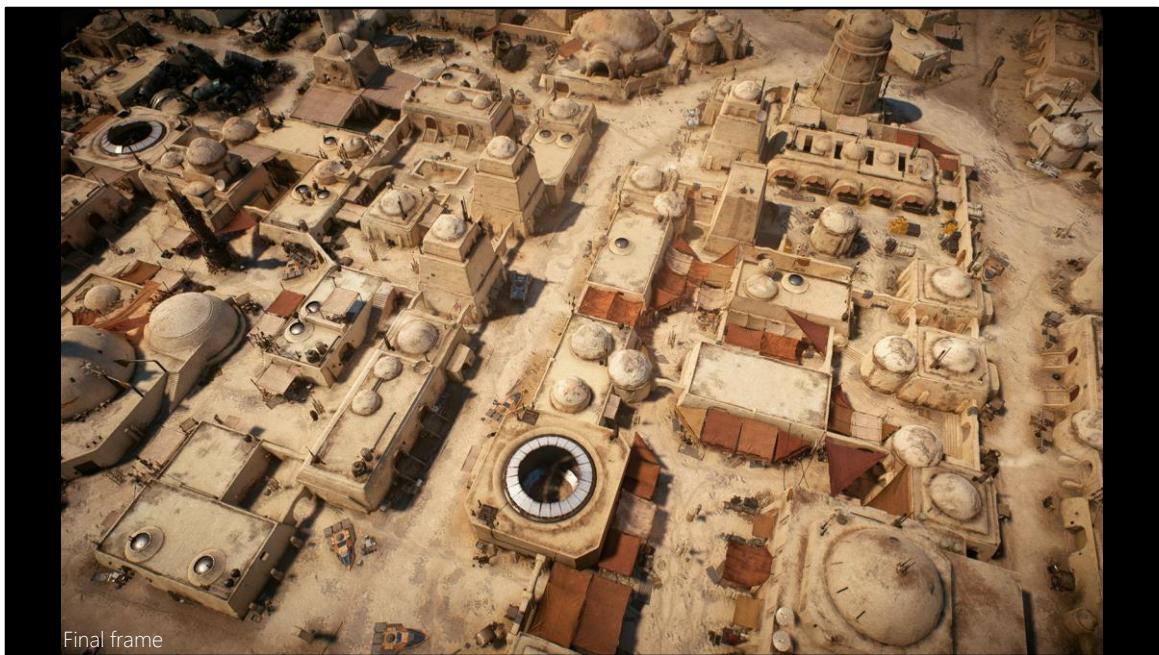
Indirect lighting, low-poly GI scene



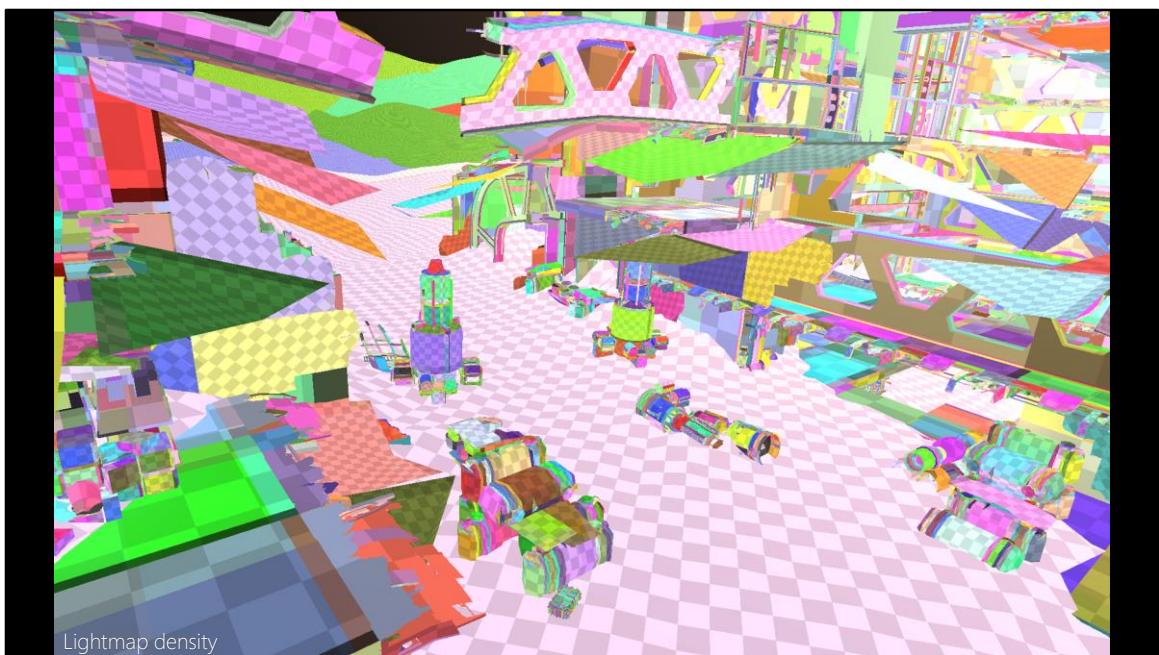
Indirect lighting, final geometry



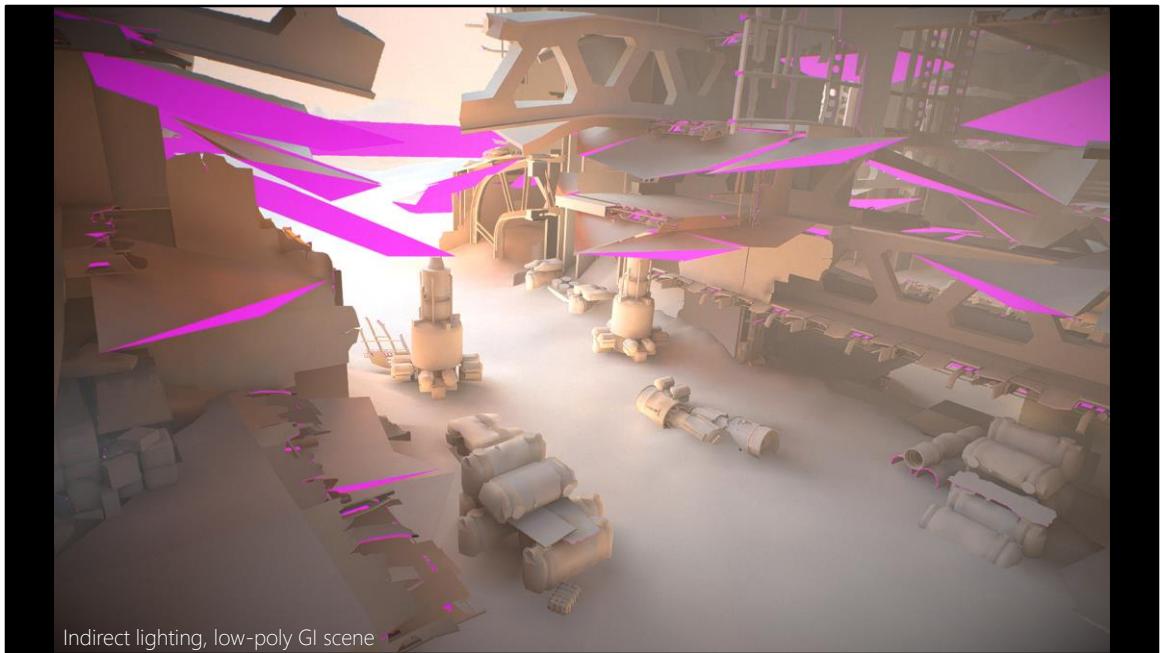
Direct + indirect lighting, final geometry



Final frame



Lightmap density



Indirect lighting, low-poly GI scene

The magenta surfaces in this screenshot represent visible back-faces. This is typically useful for artists to quickly identify errors in proxy geometry.

In this particular case, the surfaces correspond to some cloth that scatters light.



Indirect lighting, final geometry



Direct + indirect lighting, final geometry



Final frame

Part I: Path-traced SH lightmaps in Frostbite

- Motivation
- Diffuse lighting
- Efficient encoding
- Approximate specular lighting



Why path tracing?

- Elegant and unified approach for all light types (sun, sky, local lights)
- Easy to implement and understand
- Embarrassingly parallel
- Easy to validate against ground truth
- Progressive rendering is possible



We chose path tracing because it was a very simple approach to implement that could still solve all our intended use case pretty well.

It was trivial to distribute over a cluster of machines in our studios, since computations for every lightmap texel are completely independent.

It was also easy to validate our implementation against other path tracers, such as Mitsuba. We used this for testing implementation of our area light source sampling in particular.

Last but not least, we have aimed from the beginning to use our framework for progressive / interactive lighting artist workflows. Path tracing was a natural choice, as it does not require any preprocessing steps (other than building the acceleration structures).

Why not path tracing?

- Expensive
 - 8 megapixel lightmap
 - 100k rays traced / pixel primary + secondary + shadow
 - 800bn rays in total
 - >200 CPU core-hours to bake*
- Some scenes require **huge** number of rays
 - Need BDPT, MLT, photon mapping or other algorithms to solve efficiently
 - Radiosity can also work great, with some quality loss



* Assuming performance of 1M rays / second on 1 CPU core



Sounds great, huh!? Well, the reality that people typically don't talk about is that brute force path tracing (most commonly used approach) is very computationally expensive. While getting a noisy [preview quality] image can be almost instant, getting the final clean result can take hundreds of core-hours. Distributed computations (XGE / SN-DBS) and GPUs help tremendously, though it is still quite slow.

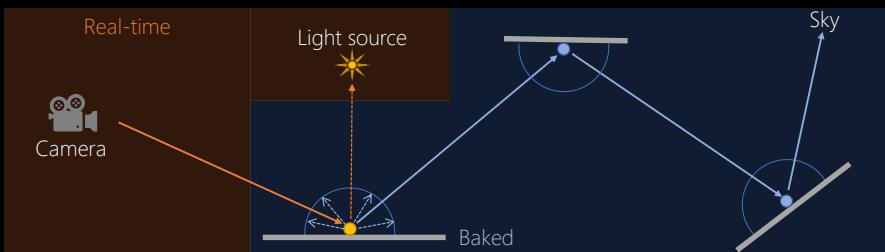
Some scenes take a completely impractical amount of time to converge and really require smarter solutions, such as bi-directional path tracing, Metropolis light transport, photon mapping, irradiance caching, radiosity, etc.

Post-process denoising also helps.

Baked Global Illumination (1)

$$L_o(x, \omega_o) = L_e(x) + \int_{\Omega} L_i(x, \omega_i) f_r(x, \omega_o, \omega_i) (\omega_i \cdot n) d\omega_i$$

- Baked GI is a database/cache, keyed by position x
 - Stores partial solution to the Rendering Equation



When people talk about precomputed global illumination, they invariably mean splitting the rendering solution into a pre-computed and run-time parts. Partial R.E. solutions are computed offline and stored in a form that can be indexed using a world position. Lightmaps, irradiance probes, etc. are all just variations on the idea that have been used in games since Quake1 (or earlier?). Luckily, it's possible to split the R.E. integral more or less arbitrarily and combine various partial solutions that are computed offline or in real-time to get various degrees of approximation for global illumination.

Baked Global Illumination (2)

$$L_o(x, \omega_o) = L_e(x) + \int_{\Omega} L_i(x, \omega_i) f_r(x, \omega_o, \omega_i) (\omega_i \cdot n) d\omega_i$$

- Eye vector ω_o is not known (no camera during baking)
- Shading normal n may be not known at baking time
 - Normal maps or geometry LODs may be used at run-time
- Flux bakes **spherical irradiance function** in lightmaps and probes
 - Assuming basic diffuse BRDF (i.e. $f_r = \frac{k}{\pi}$) and $n_{shading} = n_{face}$
 - Evaluated using per-pixel surface normal n and base color k at runtime
 - Using Spherical Harmonics



FROSTBITE

There are few problems with this general approach. Eye vector and final shading surface normal may be not known at baking time. Eye vector will come from dynamic camera, normal will come from different geometry LODs and normal maps. It is therefore necessary to store the GI data in a form that can be evaluated later using the final dynamic parameters.

The common approaches are to bake either radiance or irradiance function. Spherical harmonics are commonly used. Alternatives include spherical gaussians, H-basis, radiosity normal mapping, ambient + highlight direction, etc.

We use low-order L1 spherical harmonics to store irradiance data in our lightmaps.

Spherical Harmonics

- Baking
- Encoding
- Evaluation

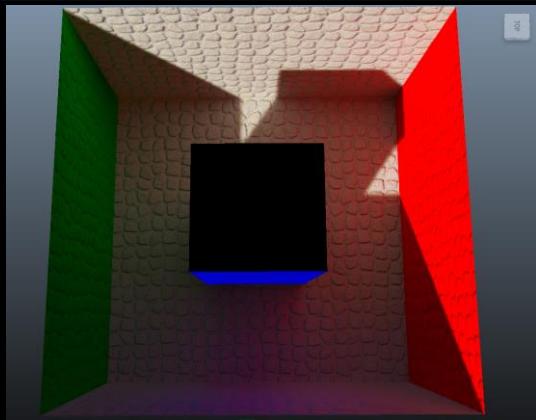


Directional Lightmaps in Mirror's Edge

I want normal maps to look
as good as they did in
Mirror's Edge



Oscar Carlen
Lighting artist
Mirror's Edge, Battlefield



Beäst

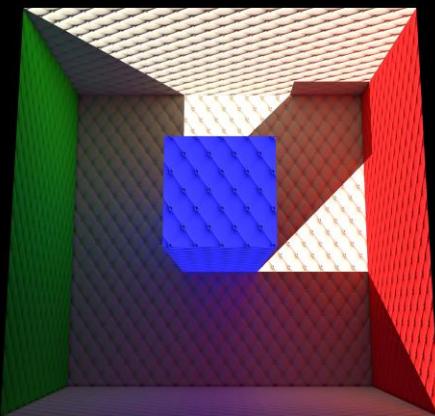
 FROSTBITE

Directional Lightmaps in Battlefront II

Challenge accepted!



Yuriy O'Donnell
Global Illumination
Enthusiast



FROSTBITE

Why spherical harmonics?

- No tangent frame required
 - Unlike RNM [Green07] or \mathcal{H} -Basis [Habel10]
- Chroma-separated RGB directional lighting
 - Unlike ambient + highlight direction (AHD) [Iwanicki13]
- High contrast diffuse lighting
- Approximate indirect specular lighting
- Good compression options for RGB L1 SH



After evaluating many different schemes for storing precomputed global illumination data (spherical gaussians, RNM / HL2 basis, ambient cube, H-basis, ambient+directional, etc.) we have settled on RGB L1 spherical harmonics for our lightmaps (12 coefficients total per texel).

We found SH to be the best compromise of storage footprint, evaluation cost and quality for our particular use cases (60Hz ~1080p action games).

Baking spherical harmonics (1)

For rays on a uniform sphere:

```
radianceSh += 4pi/N * shEvaluateL1(rayDirection) * rayRadiance;
```

```
SHL1 shEvaluateL1(vec3 p)
{
    float Y0 = 0.282095f; // sqrt(1/fourPi)
    float Y1 = 0.488603f; // sqrt(3/fourPi)
    SHL1 sh;
    sh[0] = Y0;
    sh[1] = Y1 * p.y;
    sh[2] = Y1 * p.z;
    sh[3] = Y1 * p.x;
    return sh;
}
```



$$L_{00} = \sqrt{\frac{1}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) \quad L_{1-1} = \sqrt{\frac{3}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) y_i \quad L_{10} = \sqrt{\frac{3}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) z_i \quad L_{11} = \sqrt{\frac{3}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) x_i$$



FROSTBITE

Spherical harmonics are nothing new. The process of baking and evaluating them is quite well covered in the literature already. I've included a sample implementation here just for completeness / convenience.

The typical process is to first compute SH representation of the radiance at a particular point using Monte Carlo sampling. We can generate random rays on a sphere (or hemisphere for lightmaps) and project the radiance in those ray directions onto SH basis, as shown on the slide.

The result is a set of SH coefficients that represent the spherical radiance function (i.e. amount of light coming from a particular single direction).

Baking spherical harmonics (2)

- Convolve with clamped cosine lobe to get the irradiance

```
irradianceSh = shApplyDiffuseConvolutionL1(radianceSh);
```

```
void shApplyDiffuseConvolutionL1(SHL1& sh)
{
    float A0 = 0.886227f; // pi/sqrt(fourPi)
    float A1 = 1.023326f; // sqrt(pi/3)
    sh[0] *= A0;
    sh[1] *= A1;
    sh[2] *= A1;
    sh[3] *= A1;
}
```



$$A_0 = \frac{\pi}{\sqrt{4\pi}} \quad A_1 = \sqrt{\frac{\pi}{3}}$$

SH coefficients for Lambertian BRDF
AKA clamped cosine lobe

$$E_{lm} = A_l L_{lm}$$

FROSTBITE

For rendering, we need to compute the amount of light that falls on a surface with a particular normal from all directions on a hemisphere around it. In other words, we need to compute **irradiance** at a particular point on the geometry surface.

This is achieved by convolving the radiance function in SH form with the SH form of our BRDF (clamped cosine).

SH is a frequency-space representation of a signal and a convolution in this form is simple per-component multiplication of the SH basis factors.

This process is quite well known and previously described in detail [Ramamoorthi01].
<http://cseweb.ucsd.edu/~ravir/papers/envmap/envmap.pdf>
<http://graphics.stanford.edu/papers/invlamb/josa.pdf>

Simplifying irradiance SH (1)

- `shEvaluateL1` can be merged with `shApplyDiffuseConvolutionL1`

```
SHL1 shEvaluateL1(vec3 p)
{
    float Y0 = 0.282095f; // sqrt(1/fourPi)
    float Y1 = 0.488603f; // sqrt(3/fourPi)
    SHL1 sh;
    sh[0] = Y0;
    sh[1] = Y1 * p.y;
    sh[2] = Y1 * p.z;
    sh[3] = Y1 * p.x;
    return sh;
}
```

```
void shApplyDiffuseConvolutionL1(SHL1& sh)
{
    float A0 = 0.886227f; // pi/sqrt(fourPi)
    float A1 = 1.023326f; // sqrt(pi/3)
    sh[0] *= A0;
    sh[1] *= A1;
    sh[2] *= A1;
    sh[3] *= A1;
}
```

```
SHL1 shEvaluateDiffuseL1(vec3 p)
{
    float AY0 = 0.25f;
    float AY1 = 0.50f;
    SHL1 sh;
    sh[0] = AY0;
    sh[1] = AY1 * p.y;
    sh[2] = AY1 * p.z;
    sh[3] = AY1 * p.x;
    return sh;
}
```



What is interesting is that we can just merge the SH coefficients of our BRDF and the SH basis normalization factors. Of course this is most certainly not a crazy coincidence, just algebra. I've included some more detailed walk-through of the simplification as bonus slides.

Radiance derivation

Assuming SH truncated to L1 and integration over a sphere

$$L(\theta, \varphi) = \sum_{l,m} L_{lm} Y_{lm}(\theta, \varphi)$$

$$Y_{00} = \sqrt{\frac{1}{4\pi}} \quad Y_{1m} = \sqrt{\frac{3}{4\pi}} (x; y; z)$$

$$SH \text{ basis functions}$$

$$L_{lm} = \int_{\theta=0}^{\pi} \int_{\varphi=0}^{2\pi} L(\theta, \varphi) Y_{lm}(\theta, \varphi) \sin\theta, d\theta, d\varphi$$

Solve using Monte Carlo

$$L_{00} = \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) \sqrt{\frac{1}{4\pi}}$$

$$L_{1m} = \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) \sqrt{\frac{3}{4\pi}} (x_i; y_i; z_i)$$



Let's just step through all the simplification steps for fun :)

Based on:

<http://cseweb.ucsd.edu/~ravir/papers/envmap/envmap.pdf>

<http://graphics.stanford.edu/papers/invlamb/josa.pdf>

Irradiance derivation

Assuming SH truncated to L1 and integration over a sphere

$$E(\theta, \varphi) = \sum_{l,m} \sqrt{\frac{4\pi}{2l+1}} A_l L_{lm} Y_{lm}(\theta, \varphi)$$

$$A(\theta) = \max[\cos\theta, 0] \quad A(\theta) = \sum_l A_l Y_{l0}(\theta)$$

$$A_0 = \frac{\pi}{\sqrt{4\pi}} \quad A_1 = \sqrt{\frac{\pi}{3}} \quad Y_{00} = \sqrt{\frac{1}{4\pi}} \quad Y_{1m} = \sqrt{\frac{3}{4\pi}} (x; y; z)$$

SH coefficients for Lambertian BRDF
AKA clamped cosine lobe

SH basis functions

$$E(\theta, \varphi) = \sqrt{\frac{4\pi}{2\times 0+1}} A_0 L_{00} Y_{00} + \sqrt{\frac{4\pi}{2\times 1+1}} A_1 L_{1-1} Y_{1-1}(\theta, \varphi) + \sqrt{\frac{4\pi}{2\times 1+1}} A_1 L_{10} Y_{10}(\theta, \varphi) + \sqrt{\frac{4\pi}{2\times 1+1}} A_1 L_{11} Y_{11}(\theta, \varphi)$$

$$E(x, y, z) = \sqrt{\frac{4\pi}{1}} \sqrt{\frac{1}{4\pi}} \sqrt{\frac{\pi}{4\pi}} L_{00} + \sqrt{\frac{4\pi}{3}} \sqrt{\frac{3}{4\pi}} \sqrt{\frac{\pi}{3}} L_{1-1} y + \sqrt{\frac{4\pi}{3}} \sqrt{\frac{3}{4\pi}} \sqrt{\frac{\pi}{3}} L_{10} z + \sqrt{\frac{4\pi}{3}} \sqrt{\frac{3}{4\pi}} \sqrt{\frac{\pi}{3}} L_{11} x$$

$$E(x, y, z) = 0.25 \sqrt{4\pi} L_{00} + 0.5 \sqrt{\frac{4\pi}{3}} L_{1-1} y + 0.5 \sqrt{\frac{4\pi}{3}} L_{10} z + 0.5 \sqrt{\frac{4\pi}{3}} L_{11} x$$

$$E(x, y, z) = 0.886227 L_{00} + 1.0233 L_{1-1} y + 1.0233 L_{10} z + 1.0233 L_{11} x$$



FROSTBITE

Let's just step through all the simplification steps for fun :)

Based on:

<http://cseweb.ucsd.edu/~ravir/papers/envmap/envmap.pdf>

<http://graphics.stanford.edu/papers/invlamb/josa.pdf>

Combining radiance & irradiance

Assuming SH truncated to L1 and integration over a sphere

Radiance

$$L_{1-1} = \sqrt{\frac{3}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) y_i$$

$$L_{00} = \sqrt{\frac{1}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i)$$

$$L_{10} = \sqrt{\frac{3}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) z_i$$

$$L_{11} = \sqrt{\frac{3}{4\pi}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) x_i$$

Combined formulation

$$E_{1-1} = 0.5 \sqrt{\frac{3}{4\pi}} \sqrt{\frac{4\pi}{3}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) y_i$$

$$E_{00} = 0.25 \sqrt{\frac{1}{4\pi}} \sqrt{4\pi} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i)$$

$$E_{10} = 0.5 \sqrt{\frac{3}{4\pi}} \sqrt{\frac{4\pi}{3}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) z_i$$

$$E_{11} = 0.5 \sqrt{\frac{3}{4\pi}} \sqrt{\frac{4\pi}{3}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) x_i$$

Irradiance

$$E(\textcolor{violet}{x}, \textcolor{blue}{y}, \textcolor{red}{z}) = 0.25 \sqrt{4\pi} L_{00} + 0.5 \sqrt{\frac{4\pi}{3}} L_{1-1} \textcolor{blue}{y} + 0.5 \sqrt{\frac{4\pi}{3}} L_{10} \textcolor{red}{z} + 0.5 \sqrt{\frac{4\pi}{3}} L_{11} \textcolor{violet}{x}$$

$$E(\textcolor{violet}{x}, \textcolor{blue}{y}, \textcolor{red}{z}) = E_{00} + E_{1-1} \textcolor{blue}{y} + E_{10} \textcolor{red}{z} + E_{11} \textcolor{violet}{x}$$



FROSTBITE

Let's just step through all the simplification steps for fun :)

Based on:

<http://cseweb.ucsd.edu/~ravir/papers/envmap/envmap.pdf>

<http://graphics.stanford.edu/papers/invlamb/josa.pdf>

Simplifying irradiance integral

Combined formulation
over full sphere

$$E_{00} = 0.25 \sqrt{\frac{1}{4\pi}} \sqrt{4\pi} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i)$$

$$E_{1-1} = 0.5 \sqrt{\frac{3}{4\pi}} \sqrt{\frac{4\pi}{3}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) y_i$$

$$E_{10} = 0.5 \sqrt{\frac{3}{4\pi}} \sqrt{\frac{4\pi}{3}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) z_i$$

$$E_{11} = 0.5 \sqrt{\frac{3}{4\pi}} \sqrt{\frac{4\pi}{3}} \frac{4\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) x_i$$

Final simplified form for Monte Carlo integration
over full sphere

$$E_{00} = \frac{\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i)$$

$$E_{1-1} = \frac{2\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) y_i$$

$$E_{10} = \frac{2\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) z_i$$

$$E_{11} = \frac{2\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) x_i$$

over hemisphere

$$E_{00} = \frac{\pi}{2N} \sum_{i=1}^N L(x_i, y_i, z_i)$$

$$E_{1-1} = \frac{\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) y_i$$

$$E_{10} = \frac{\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) z_i$$

$$E_{11} = \frac{\pi}{N} \sum_{i=1}^N L(x_i, y_i, z_i) x_i$$



FROSTBITE

Let's just step through all the simplification steps for fun :)

Based on:

<http://cseweb.ucsd.edu/~ravir/papers/envmap/envmap.pdf>

<http://graphics.stanford.edu/papers/invlamb/josa.pdf>

Simplifying irradiance SH (2)

- More opportunities for cancellation:
 - $1/\pi$ in the BRDF
 - 4π in spherical Monte Carlo integration (2π for hemispherical)
- Final simplified L1 SH irradiance Monte Carlo integration

$$E_{00} = \frac{1}{N} \sum_{i=1}^N \mathbf{L}(x_i, y_i, z_i) \quad E_{1-1} = \frac{2}{N} \sum_{i=1}^N \mathbf{L}(x_i, y_i, z_i) y_i \quad E_{10} = \frac{2}{N} \sum_{i=1}^N \mathbf{L}(x_i, y_i, z_i) z_i \quad E_{11} = \frac{2}{N} \sum_{i=1}^N \mathbf{L}(x_i, y_i, z_i) x_i$$

For rays on a uniform sphere:

```
irradianceSH += SHL1rgb(rayRadiance, 2 * rayRadiance * rayDirection) / N;
```



The cool thing that comes out of all this is that our L0 coefficients simply contain the average radiance, while L1 contains weighted-average radiance direction.

Graham Hazel describes a simplification process that arrives to the same result:
<https://grahamhazel.com/blog/2017/12/22/converting-sh-radiance-to-irradiance>

Simplifying irradiance SH (3)

- Factor out 2 and $L0$ from $L1$ to bring its magnitude into 0..1 range

For rays on a uniform sphere:

```
irradianceSH += SHL1rgb(rayRadiance, rayRadiance * rayDirection) / N;
```

Then `irradianceSH.L1 /= irradianceSH.L0;`

Evaluation shader:

```
result = (0.5 + dot(irradianceSH.L1, normal)) * irradianceSH.L0 * 2.0;
```



Might be negative and already includes $\frac{1}{\pi}$ BRDF term



For storage efficiency, we can further take out the 2 and $L0$ factors from our $L1$ coefficients. This is good for storage / encoding, as it brings the magnitude of our $L1$ data into 0..1 range.

Reconstruction shader is then also simplified, as shown on the slide. The pseudocode evaluates the irradiance function using the surface normal and outputs the outgoing radiance (i.e. reflected light) that's only missing the albedo factor.

Note that result of SH evaluation may be negative, since magnitude of $L1$ vector is potentially twice that of $L0$. This artifact is known as “ringing”.

Irradiance SH lightmap encoding

- Use 4 RGB textures to store 12 SH coefficients
 - L_0 coefficients in HDR (BC6H texture)
 - L_1 coefficients in LDR (3x BC7 or BC1 textures)
- Total footprint for RGB SH lightmaps:
 - 32 bits (4 bytes) / texel for BC6+BC7, high quality mode
 - 20 bits (2.5 bytes) / texel for BC6+BC1, low quality mode
- Example:
 - 4096 x 4096 lightmap
 - 64MB at 32 bits / texel, 40MB at 20 bits/texel



Now that our data is in a nice and convenient range, we can encode it using standard block compression formats.

L_0 coefficients represent the average radiance and have high dynamic range. BC6HU was a naturally good fit for them.

We use a single BC6HU texture to store 3x L_0 coefficients (RGB).

L_1 coefficients represent a weighted average light direction. Since we've factored out $2*L_0$, the data is now in a convenient 0..1 range.

We can use an LDR texture format to store L_1 coefficients. We need 3 textures to store 9 L_1 coefficients.

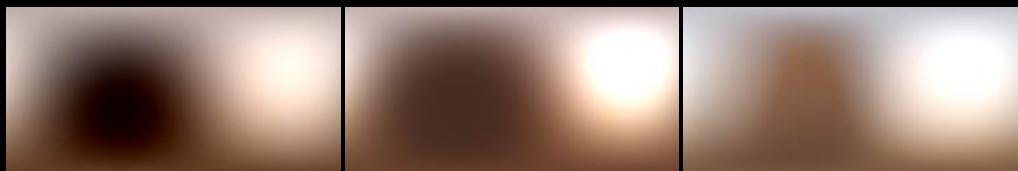
BC7 is a great choice from quality point of view, but BC1 also works pretty well.

Non-linear SH lightmap evaluation

- Proposed by Geomerics [Hazel15]
- Improves contrast
- Guarantees only positive results (no ringing)
- Guarantees correct total energy



Radiance environment probe
Image by Bernhard Vogl



Diffuse irradiance L1 SH

Diffuse irradiance non-linear L1 SH
[Hazel15]

Diffuse irradiance
Ground truth Monte Carlo



<https://grahamhazel.com/blog/2017/12/22/converting-sh-radiance-to-irradiance>

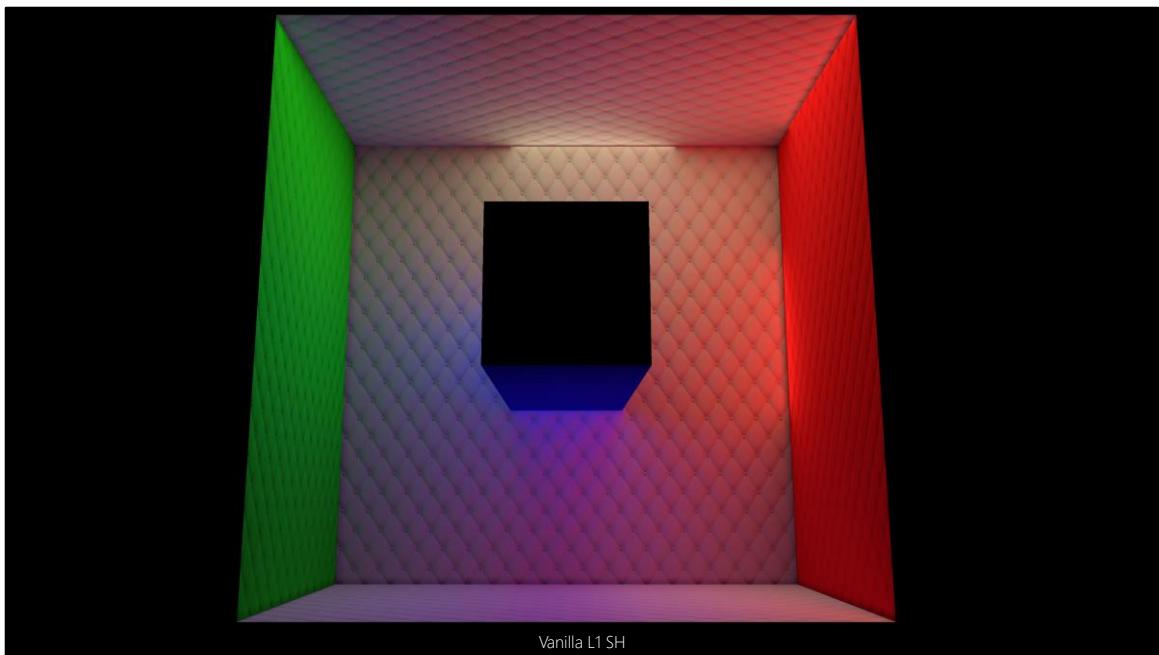


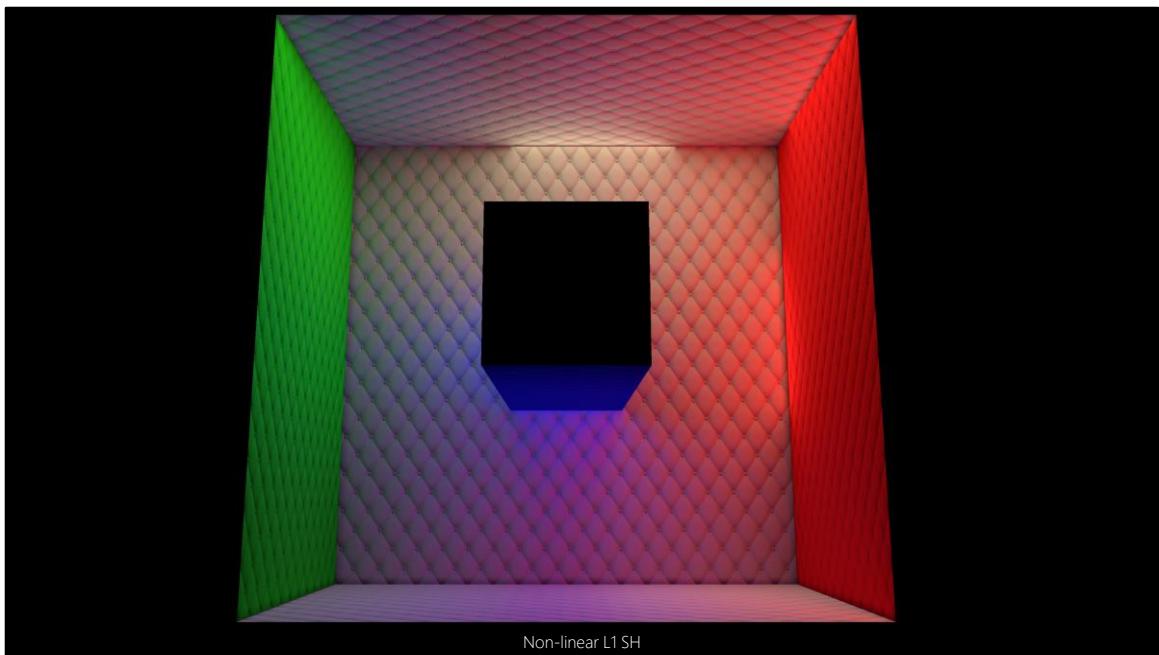
While simple L1 SH lighting is quite fast and intuitive, we can do a lot better. Frostbite uses a technique proposed by Graham Hazel & Chris Doran from Geomerics:
https://web.archive.org/web/20160313132301/http://www.geomerics.com/wp-content/uploads/2015/08/CEDEC_Geomerics_ReconstructingDiffuseLighting1.pdf
<https://grahamhazel.com/blog/2017/12/22/converting-sh-radiance-to-irradiance>

It was originally described as a solution to the ringing artifacts (unnaturally dark / negative values of the function) for L1 SH probe-based lighting. Artifacts appear on the side of the sphere opposite of the highly directional signal with high magnitude (i.e. on the side opposite the window in the example on the slide).

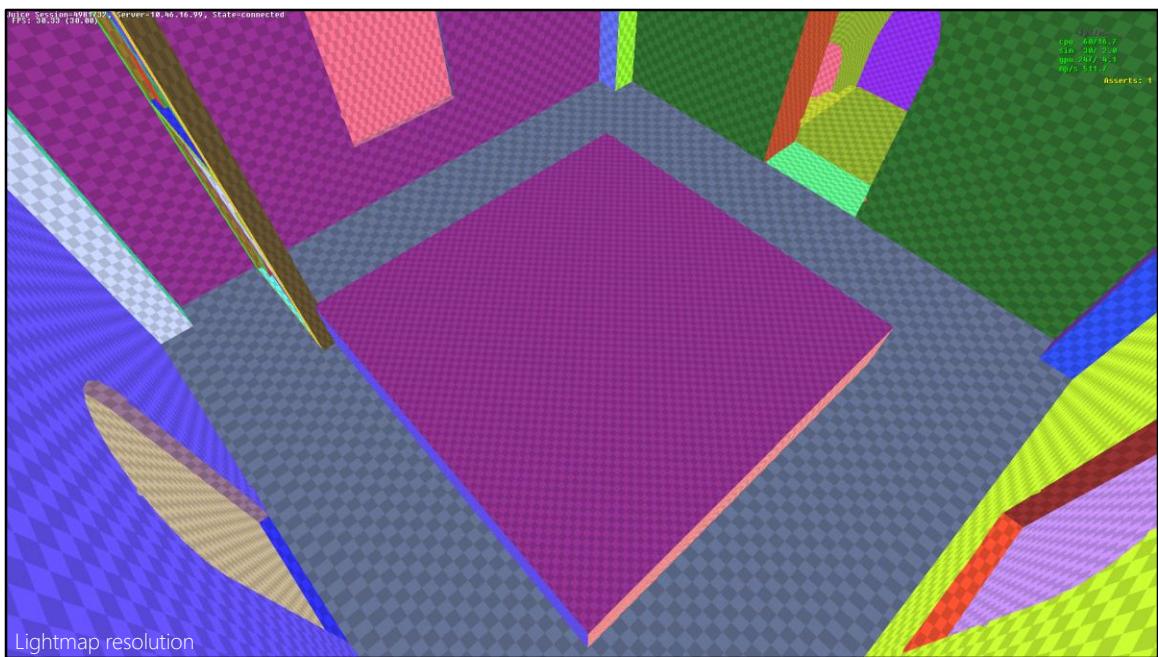
Geomerics non-linear irradiance reconstruction produces only positive results, while preserving the total energy of the function.

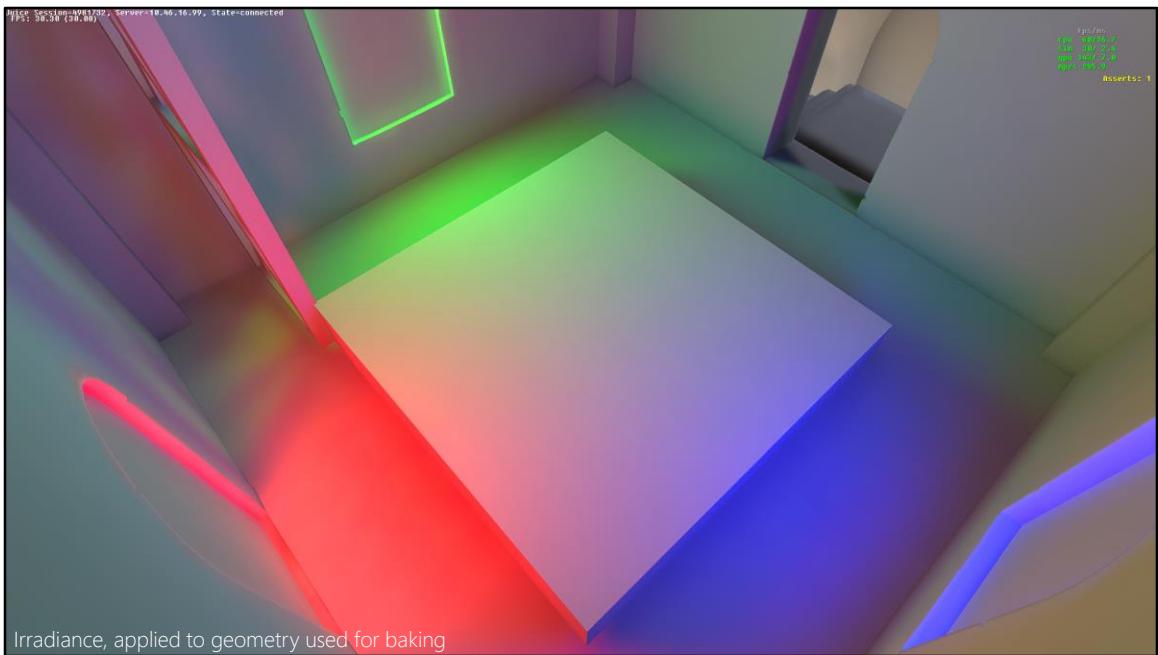
Ringing is not a big issue for lightmaps, since we typically don't see the geometry from "below" of the hemisphere around lightmap texel. However, non-linear reconstruction also happens to improve lighting contrast (as long as light mostly comes from one direction).

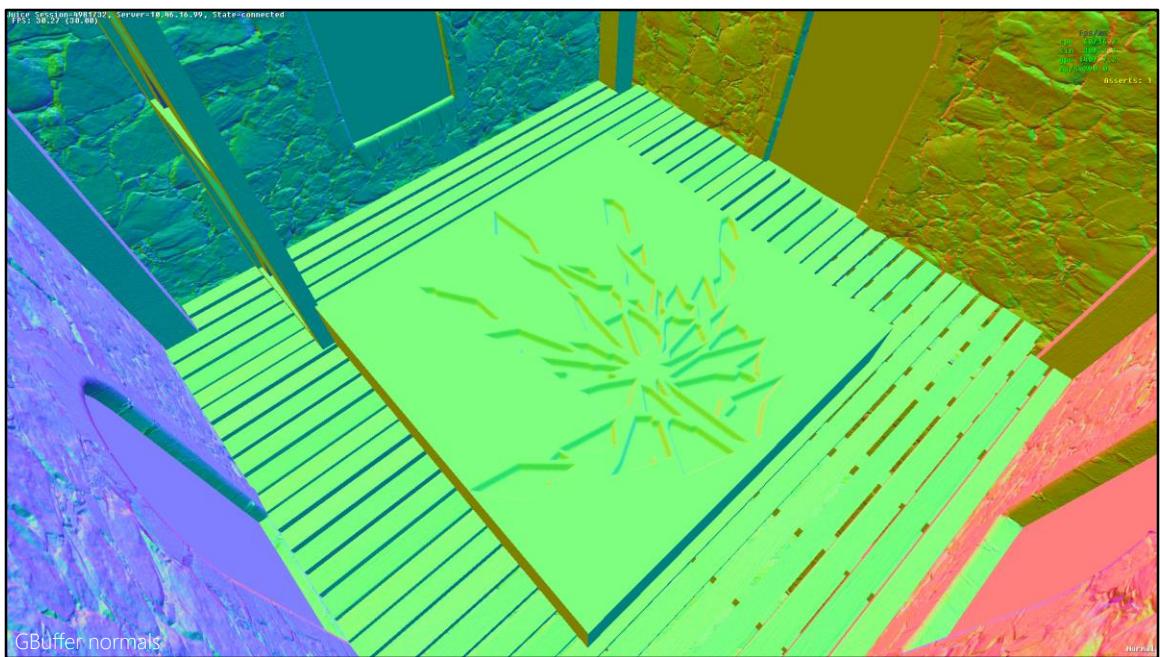


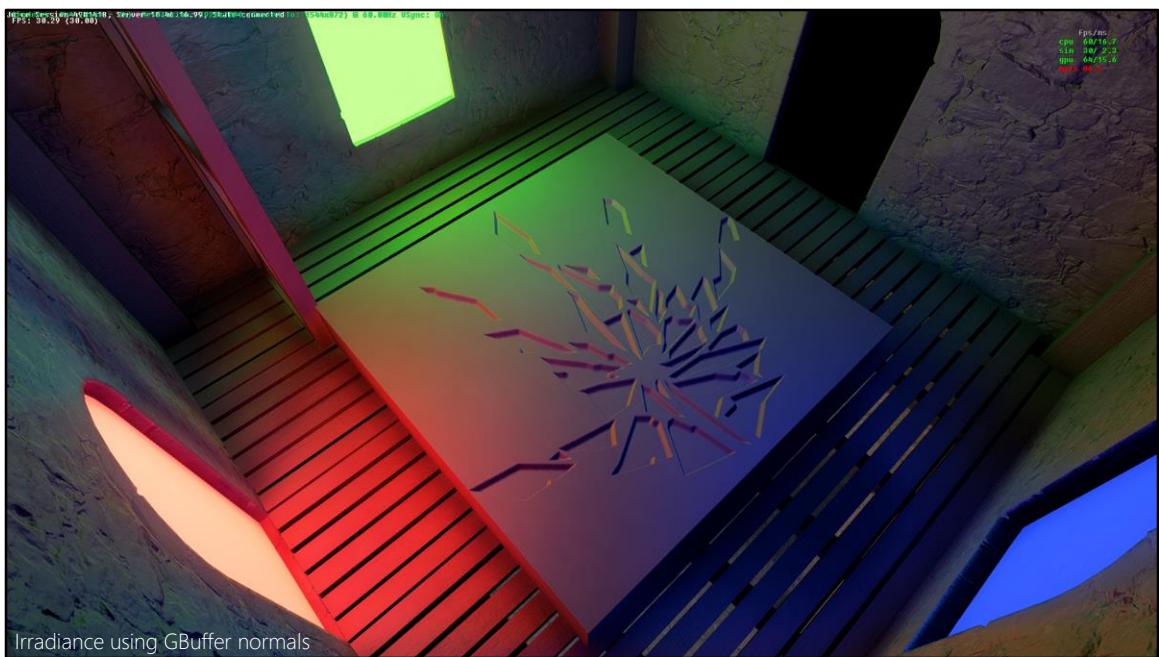


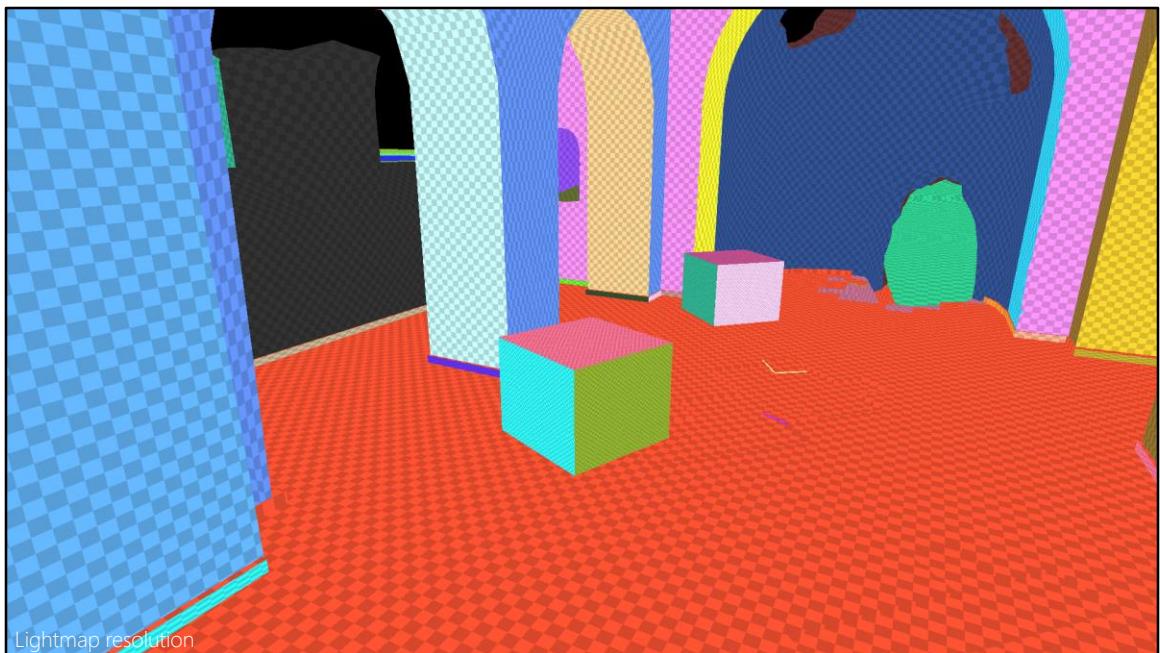
Non-linear L1 SH







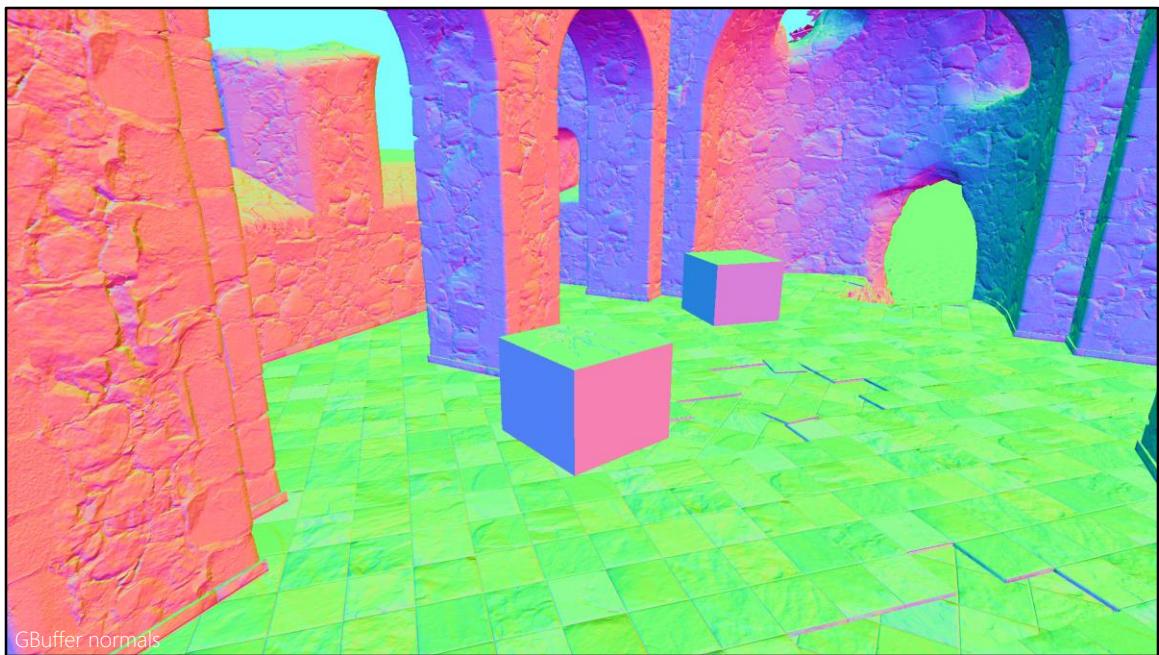




Lightmap resolution



Irradiance, applied to geometry used for baking





Irradiance using GBuffer normals

Approximate specular lighting



Doing things properly

- Lighting and Material of Halo 3 [Chen08] describes specular lighting
- Lightmaps contain radiance L2 SH data
- Uses lookup textures of SH coefficients keyed by BRDF parameters
 - Baked offline from roughness, view angle in XZ plane
 - Uses the fact that isotropic specular BRDFs are symmetrical
- Uses SH rotation to transform BRDF SH from local to world space
- SH dot product between BRDF and Radiance SH to get final specular
- Fundamentally solid, but quite a lot of ALU and texture lookups ☺



http://developer.amd.com/wordpress/media/2013/01/Chapter01-Chen-Lighting_and_Material_of_Halo3.pdf

Implementation using lookup textures is available in BakingLab [Pettineo16].

General idea for approximate specular

- Use L1 Spherical Harmonics data in lightmaps
- Derive principal light direction from SH (just normalize L1 data)
- Estimate the “spread” or “focus” based on length of L1 band
 - `float focus = length(lightmap.L1); // [0..1]`
- Create an imaginary light source using estimated parameters
- Plug the numbers into our standard GGX specular formula



Since we know that L1 data contains weighted average light direction, we can just normalize it to get a representative light source direction for the purposes of specular lighting approximation.

Approximate specular lightmap shader

- Use L1 magnitude to estimate how *focused* the lighting is
 - Similar principle to non-linear diffuse SH lighting [Hazel2015]
 - Close to 0.0 means that light comes from many directions
 - Close to 1.0 means that light mostly comes from one direction
- Adjust surface roughness as a **fast** area light approximation
 - Make highlight softer when heuristic suggests omni-directional lighting
 - Arbitrary **empirical approximation** based purely on visually pleasing and plausible result

```
// Approximate an area light by adjusting smoothness/roughness

float lightmapDirectionLength = length(lightmapDirection); // value in range [0..1]
float3 L = lightmapDirection / lightmapDirectionLength;
float adjustedSmoothness = linearSmoothness * sqrt(lightmapDirectionLength);

// Proceed with standard GGX specular maths
```



Our approach is based purely on empirical fitting (AKA randomly tweaking stuff based on a gut feeling and keeping results that look good).

We adjust the roughness of the surface based on a heuristic tries to guess how focused or spread the lighting data is.

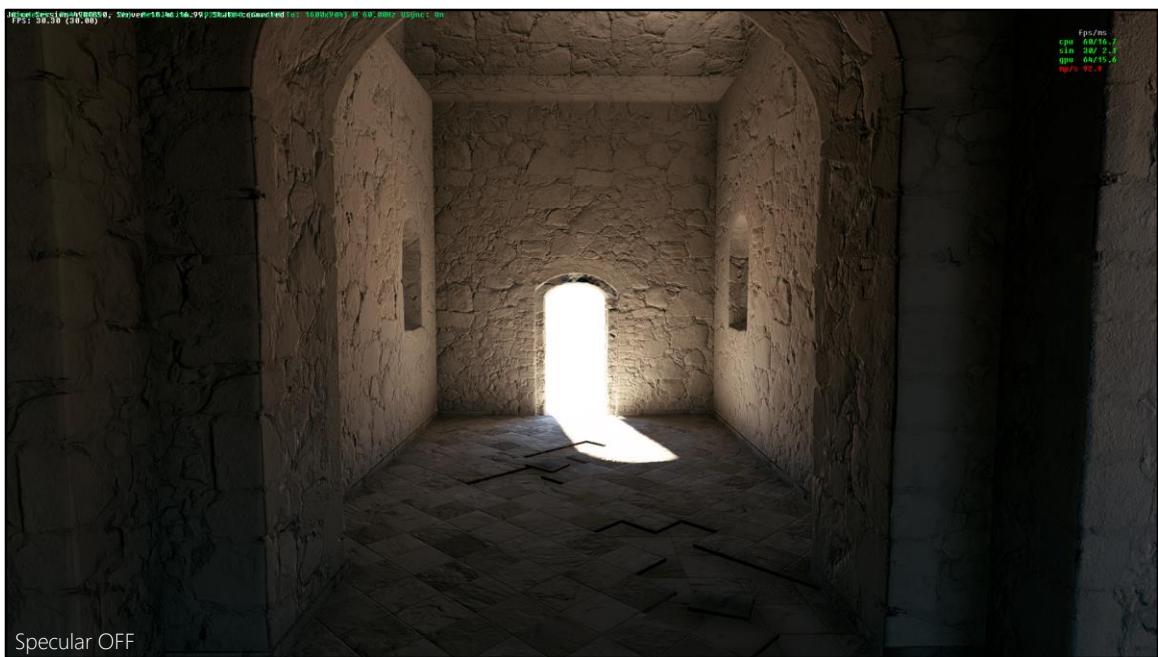
Our heuristic looks at the magnitude of our encoded L1 vector (average of 3 vectors stored in our lightmaps for RGB).

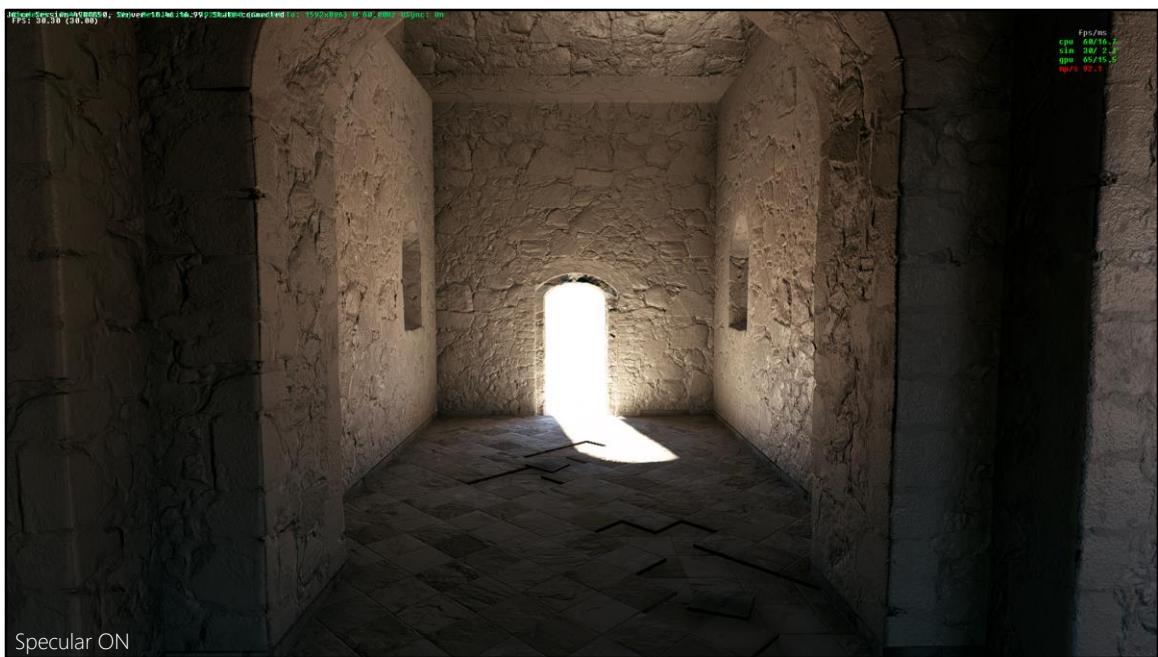
Magnitude close to 0 means that light contributions from many different directions cancel each other out, so the highlight should be softer.

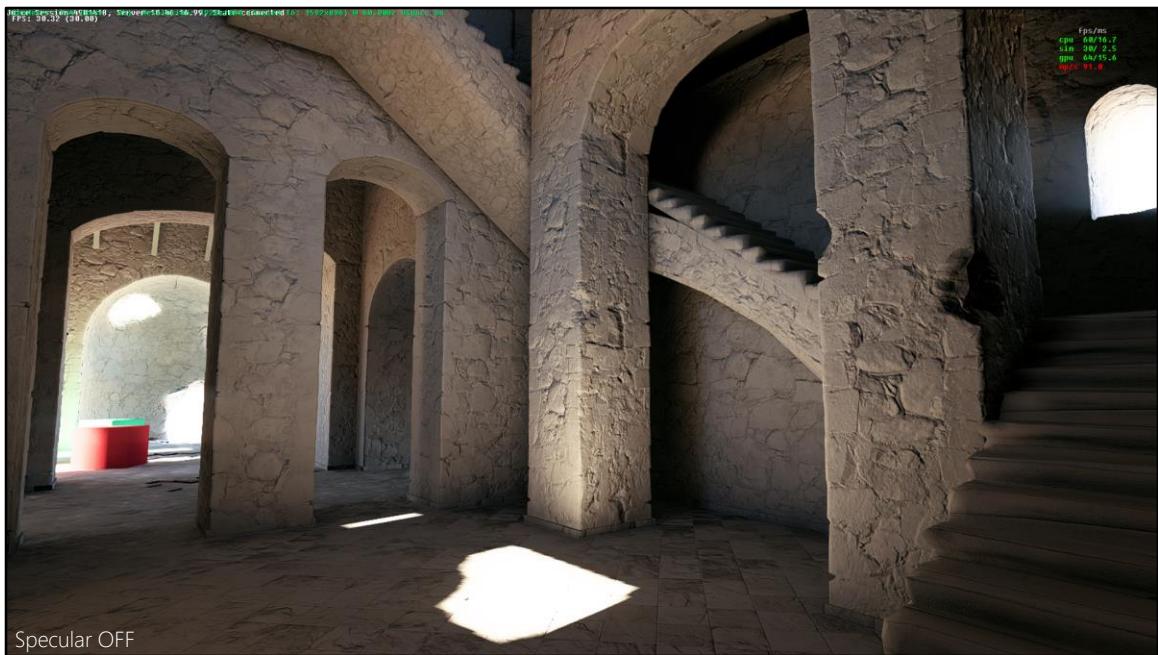
Magnitude close to 1 means that lighting is unidirectional, so the highlight should be sharper.

We can use the magnitude heuristic to adjust the surface roughness that we plug into our standard GGX specular formula.

Of course, this is nothing new and other games have used similar approaches in the past, though it is not often publicized.











Diffuse only



Diffuse + screen-space reflections



Diffuse + approximate SH specular

When compared to SSR, approx. specular lightmaps look different. This is to be expected, since we only have a single specular highlight direction. Results are still quite plausible. Since lightmaps are evaluated per-pixel, the overall image looks sharper compared to SSR which is rendered at half-resolution.

Some issues require art workarounds

- Single light direction isn't enough
- Turn off specular lightmaps per asset
- Only use specular within certain roughness range
- Use other reflection tech instead
 - Local reflection cubemaps
 - Screen-space reflections
 - Planar reflections



Since lightmaps only contain a single average light direction per texel, artifacts are noticeable on smooth surfaces in areas where dominant light direction changes smoothly.

There's not much that can be done about it, since we simply don't have the necessary data in L1 SH to accurately reconstruct specular lighting. The only option is to mitigate the issue by fading out specular lightmaps on surfaces with high smoothness. Other reflection techniques (SSR, reflection probes, planar reflections) are typically used for such surfaces any way.

We also gave artists some explicit control that allows them to turn off specular contribution per material. This happens to be helpful for GPU performance optimization.

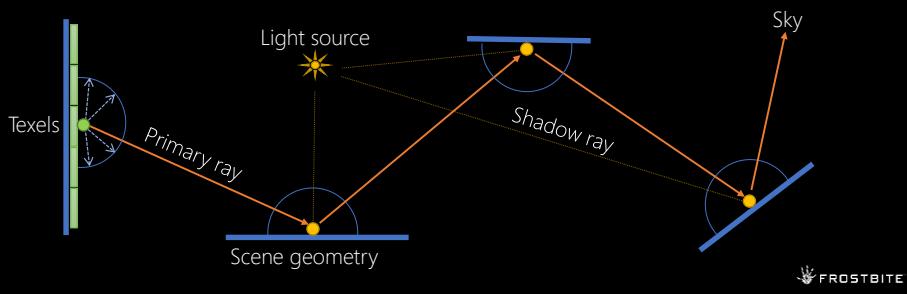
Part II: A bag of tricks

- Hemisphere and texel sampling
- Rendering convergence detection
- Dealing with overlapping geometry
- Ensuring correct bilinear interpolation
- Efficient lightmap atlas packing



Baking lightmaps with path tracing (recap)

- Generate sample points on texels
- Trace paths in a hemisphere from texel sample points



Quick recap: to bake lightmaps with path tracing, we need to trace some rays and accumulate their contribution in texels.

In this section we'll cover a few techniques related to texel and hemisphere sampling.

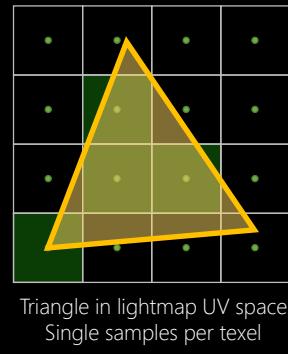
Texel sampling

Generating ray origins



Texel sampling (1)

- Sample per texel center is an obvious choice
... which does not work too well
- Aliasing is a big problem, since lightmaps are usually quite low resolution
- Large parts of the texel may overlap polygons, but texel center may not
- Could fill in such pixels through dilation
 - May not always be possible
 - May lead to visual artifacts



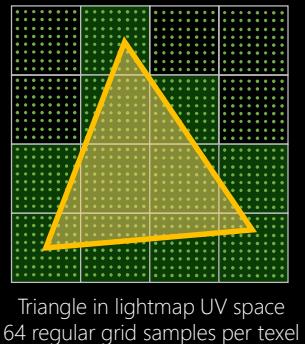
Sampling texel centers to pick ray origins is an obvious choice. It may work well enough for high-resolution lightmaps, however at our texel density this is a pretty poor strategy.

Aliasing is a big problem. A texel may have significant contribution to the final lighting of a surface, but we may be actually missing data for it due to under-sampling.

Dilation helps to fill any holes, but there are pathological cases that are basically impossible to solve through it. For example, a small triangle that occupies a 2x2 texel block may actually not overlap any texel centers. In this case there is no sensible way to produce lighting for it, other than sampling neighbors in **world space**.

Texel sampling (2)

- Use super-sampling to get many points per texel
- Flux uses up to 64 points per texel
 - We use **Hammersley** sequence
 - Regular grid also works just fine for this purpose
- Points that overlap triangles added to valid list
- Rays traced from all valid points per texel
 - Uniform-randomly selected per hemisphere ray
- Helps with aliasing in direct lightmaps too



An obvious solution to aliasing is **super-sampling**.

We generate 64 sample points on the texel using Hammersley sequence. Every texel sample that overlaps with geometry in lightmap space is added to a per-texel list. For every hemisphere ray, we uniformly-randomly pick a ray origin from the list of valid samples for the texel that we render.

Using super-sampling also helps with aliasing in direct lightmaps, where some shadows may end up too sharp and unnatural.

While this is not a particularly sound strategy from signal reconstruction point of view (we treat texels as little squares, rather than splatting sample contributions into neighbors using some filter, such as gaussian), it has the advantage of keeping calculations for all texels completely independent.

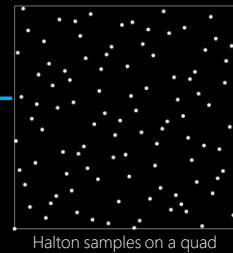
Hemisphere sampling

Generating ray directions



Hemisphere sampling

- Use Halton sequence, as it allows progressive rendering
 - Don't need to specify number of rays up-front
 - Keep tracing until we have traced "enough"
- Apply per-texel randomization to avoid correlation artifacts



Halton samples on a quad



16 samples per texel
(no randomization)



16 samples per texel
(with randomization)

By default, we use Halton sequence to generate hemisphere rays. This is a very commonly used low discrepancy, quasi Monte Carlo / pseudo-random sequence. We chose it due to being **progressive**. I.e. we do not need to pre-specify the number of samples that we intend to take, unlike other QMC approaches such as stratified sampling, Hammersley sequence, etc.

Since the sequence is deterministic, we need to apply some randomization per texel to trade correlation artifacts for noise. Correlation artifacts happen because neighboring texels happen to use exactly the same set of ray directions and hit the same features, resulting in very similar lighting data. The artifacts due to correlation look like bands of similar lighting values, which are much more difficult to get rid of with a post-process filter compared to high-frequency noise that we get with per-texel randomization.

Progressive sampling allows us to keep tracing rays for a texel until we have traced "enough", i.e. until our Monte Carlo estimate has converged.

Hemisphere sampling code

```
vec3 sampleHaltonHemisphere(int sampleIndex, vec2 offset)
{
    // Generate 2 uniformly-distributed values in range 0 to 1
    float u, v;
    sampleHalton(sampleIndex, &u, &v);

    // Apply per-texel randomization
    u = fract(u + offset.x);
    v = fract(v + offset.y);

    // Transform unit square sample to uniform hemisphere direction
    float cosTheta = u;
    float sinTheta = sqrtf(1.0f - cosTheta * cosTheta);
    float sinPhi, cosPhi;
    sinCos(v * twoPi, &sinPhi, &cosPhi);
    return vec3(cosPhi * sinTheta, sinPhi * sinTheta, cosTheta);
}
```



Sample pseudocode, just for convenience.

How many rays is “enough”?

- Required number of samples can be vastly different per texel



Indoor scenes require more rays



Outdoor scenes require fewer



The required ray count may be vastly different between different areas of the same level. Indoor scenes require more hemisphere samples, as radiance variance is generally much higher. This happens because the probability of a light path bouncing several times and then hitting a window is quite low, while the lighting contribution of such low probability path may be extremely high. Environments with high variance require many more hemisphere samples to get noise-free result. Conversely, areas that are mostly outdoors will likely produce paths that are very short. Rays traced from outdoor surfaces generally will immediately hit the sky and terminate. Of course, sky itself may be a source of high variance. We mitigate this by not including the sun in our environment maps and using pre-filtering (i.e. blurring and down-sampling the envmaps).

What we want is a generally low level of noise in the final lightmaps, while tracing as few rays as we can get away with. This requires some sort of automatic convergence detection and a progressive renderer.

Path tracing convergence detection

- Use **confidence intervals** and **standard error** (not `std::err`)
 - Similar to **adaptive sampling** meta-integrator in Mitsuba [Jakob10]
- Commonly used in statistics to **measure uncertainty**
 - Confidence interval for a Monte Carlo estimator gives us error bounds*!
 - * *With certain probability, assuming normal distribution of the data*
 - $\bar{x} \pm SEM \times zScore$ (z-score for 95% confidence interval = 1.96)
 - $SEM = \sigma_{\bar{x}} = \sqrt{Var(x)/n}$
- Define stopping condition in terms of **standard error**
 - $\sigma_{\bar{x}} \times z < \bar{x} \times E$
 - $t = E/z$
 - $\sigma_{\bar{x}} < \bar{x}t$

```
// Accept 5% error using 95% confidence interval
double threshold = 0.05 / 1.96;
double standardError = sqrt(variance / sampleCount);
bool shouldStop = standardError < mean * threshold;
```



FROSTBITE

Our approach is based on similar principles to adaptive sampling meta-integrator in Mitsuba: **confidence intervals** and **standard error of the mean (SEM)**.

Monte Carlo estimate of a function (such as our rendering equation) gives us the expected value of that function, AKA the mean. We can compute a N% confidence interval from our Monte Carlo samples, which will give us a value range which will contain the **true mean** with N% probability. Confidence interval is computed using the sample mean, standard error [of the mean] and critical value (somewhat incorrectly called "zScore" on the slides). Critical value represents the number of standard deviations from the mean of the normal distribution that creates a range containing N% of the values. For example, 95% of the values in a normal distribution are contained within 1.96 standard deviations from the mean. 99% of the values are within 2.58 standard deviations.

Error bounds computed in this way can be used as a stopping condition for path tracing. We stop taking further samples when relative error falls below a chosen threshold. We combine the critical value and the error percentage threshold into a single threshold parameter that we expose to the user.

Running variance calculation

```
int count = 0;
float mean = 0;
float meanDistSquared = 0;

while(keepSampling)
{
    float sample = getSample();

    // Update mean and mean^2
    count++;
    float delta = sample - mean;
    mean += delta / count;
    meanDistSquared += delta * (sample - mean);

    // Calculate current running variance
    float variance = meanDistSquared / (count-1);
}
```

Welford's algorithm

https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance



Just given for convenience.

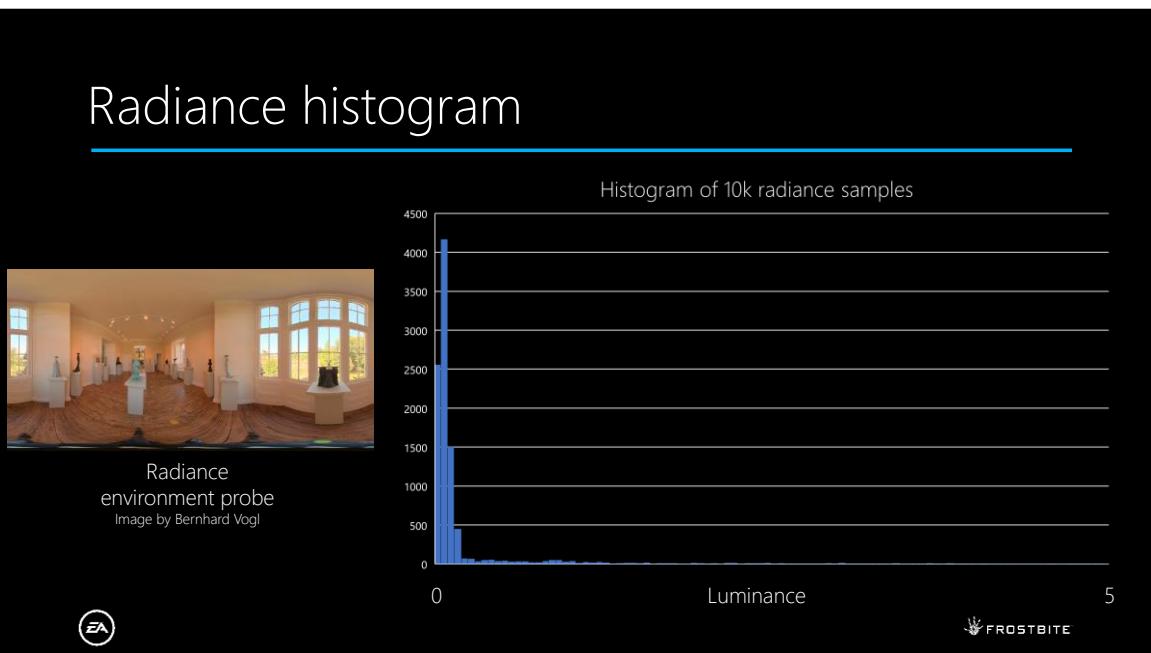
Are radiance samples normally distributed?



All this assumes that the data follows the normal distribution, which is a pretty generous assumption. In reality, radiance samples aren't normally distributed. Though it's close enough for the stopping heuristic to still work.

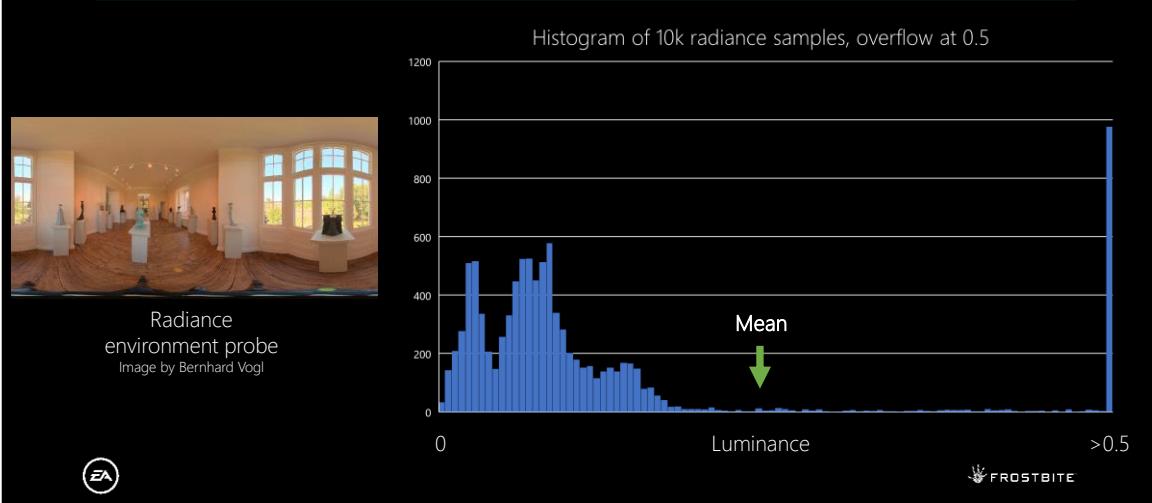
The confidence interval that we calculate just becomes less accurate. This means that rendering may stop earlier than it would if samples were *actually* normally distributed.

Radiance histogram

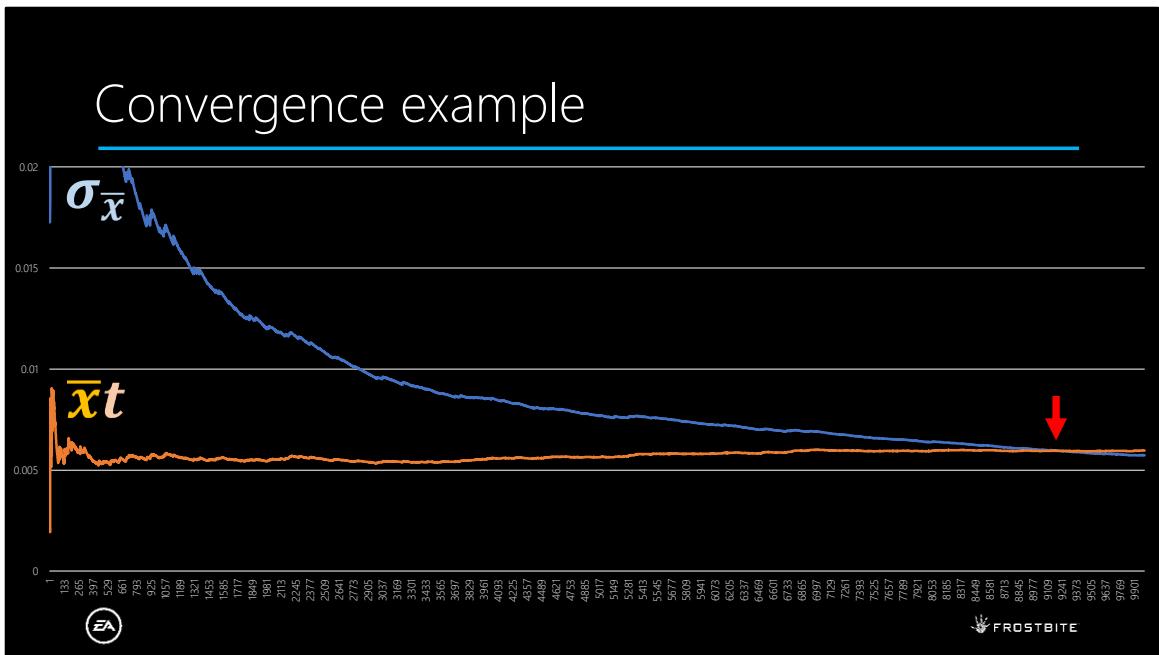


Radiance samples typically have a long tail.

Radiance histogram (truncated)



The long positive tail and lack of negative values to compensate for it shifts the mean value to the right.

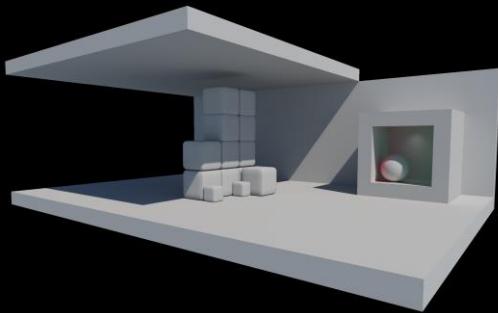


If we plot the standard error of our samples and our sample mean scaled by the error threshold, we'll see that the graphs will intersect at some point.

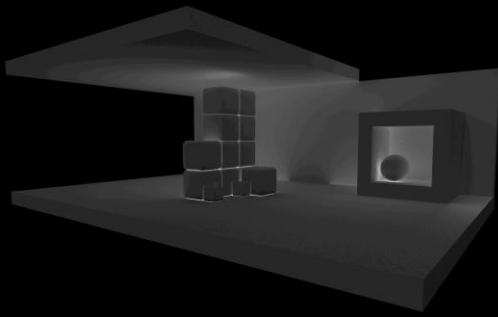
Setting a larger error threshold will shift the orange graph up and the rendering will stop earlier. On the other hand, a higher sample variance will make the standard error higher and shift the blue graph up, so rendering will stop later.

We always initially trace 10% of the total ray budget to get a good estimate of the variance and then evaluate stopping condition after tracing every subsequent 10%.

Does it work?



Lightmap rendered with up to 25k samples per texel



Heat map showing actual rendered sample count
as fraction of total sample budget



FROSTBITE

This works quite well in practice. We get exactly what we were after: fewer rays traced in the open areas and more rays near geometry.

More on standard error and confidence intervals:

<http://www.stat.yale.edu/Courses/1997-98/101/confint.htm>

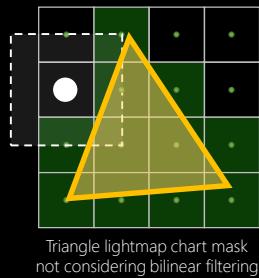
http://www.ucl.ac.uk/ich/short-courses-events/about-stats-courses/stats-rm/Chapter_6_Content/relationship_confintervals_pvalues

Lightmap atlas packing

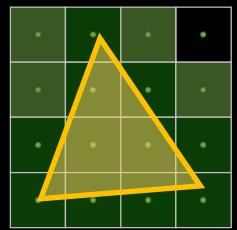


Ensuring correct bilinear interpolation

- Generate **chart masks** for groups of triangles that share texels
- Each texel affects a 2x2 area around its center due to bilinear filtering



Triangle lightmap chart mask
not considering bilinear filtering



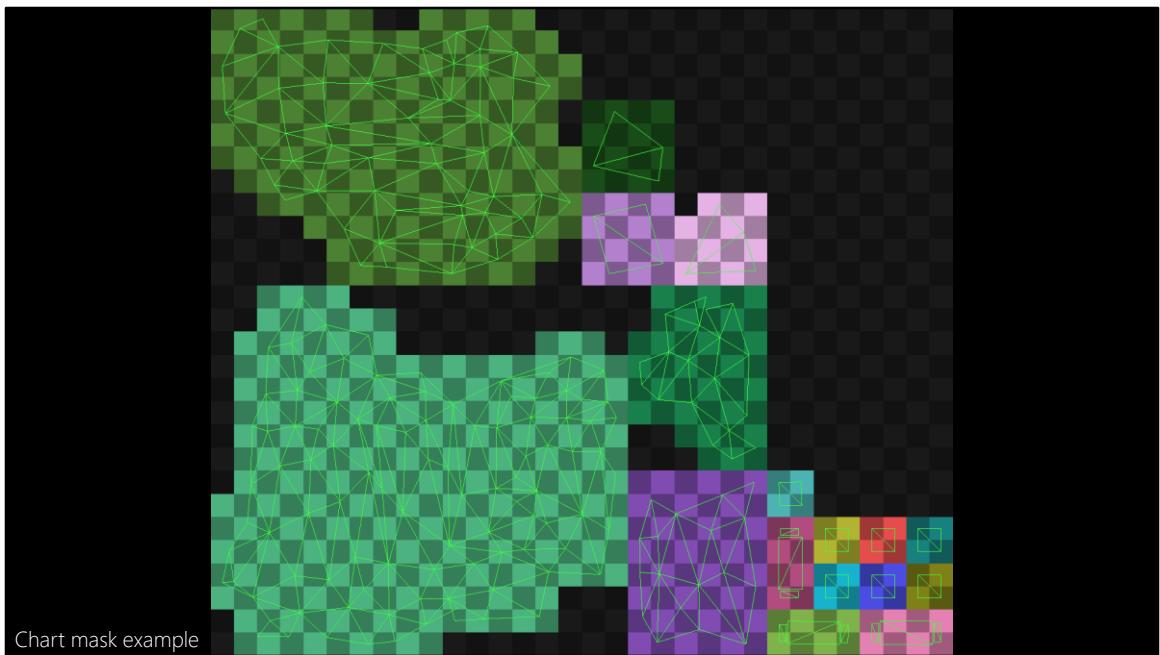
Triangle lightmap chart mask
taking bilinear filtering into account



Before we can pack our lightmap, we need to detect UV charts/islands and mark all texels that belong to all the charts.

Chart masks are computed by intersecting a 2x2 bounding box around each texel center with lightmap-space geometry. If the box overlaps a triangle, the texel gets assigned a chart ID of that triangle. If multiple triangles are overlapped with different IDs, then those triangles actually belong to the same chart (they will use some of the same texels through bilinear interpolation).

To minimize the footprint of each chart, we can scale and offset UVs of each chart such that the UV-space bounding box of each chart is aligned to texel centers. Doing this allows us to tightly pack small charts without introducing any extra padding.

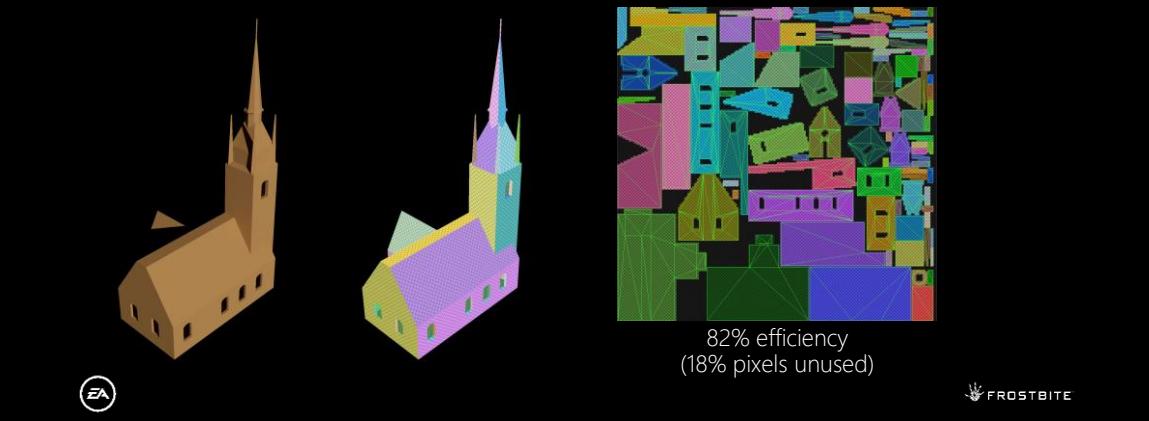


Different colors represent independent charts. Smallest chart is 2x2 (or 4x4 when block compression is used for final textures).

We do not collapse charts to Nx1 or 1x1 footprint, as we typically want to have some lighting gradient over surfaces. However, it is possible that some charts can be safely collapsed after baking (i.e. after all values are known). Similar technique (and more) is described by Richard Mitton in his blog: www.codersnotes.com/notes/lightmap-tricks

Lightmap packing efficiency issues (1)

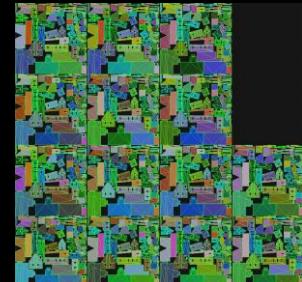
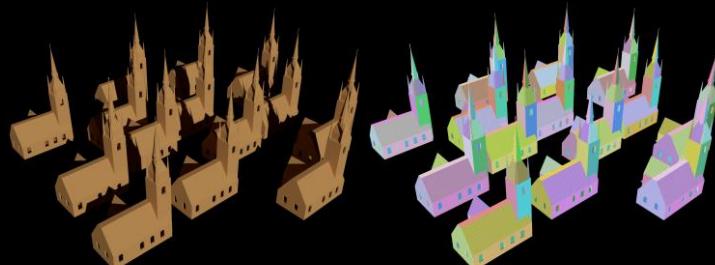
- Lightmap UVs are typically created and packed per mesh



Once we have the chart masks, we can pack them. The typical lightmap packing process first packs UVs for each mesh independently. Some efficiency is lost here, due to irregularly shaped charts.

Lightmap packing efficiency issues (2)

- Meshes are instanced in the scene many times
- Per-mesh lightmaps are packed into an atlas
 - Final atlas packing itself is not perfect
 - All inefficiencies add up



72% efficiency
(28% pixels unused)

 FROSTBITE

All lightmaps for mesh instances in the scene are typically packed into one or more atlases, for efficiency and batching purposes.

Atlas packing leads to some additional inefficiency that adds to the per-mesh packing waste. Typical lightmap utilization that we see in final production levels is ~50-75%, which is quite bad.

Lightmap packing goal

- Pack lightmap UV charts for **entire level** into single atlas
- Avoid waste due to per-mesh / instance packing
- Avoid adding unique lightmap UV stream to all meshes
- Deal with >8 megapixel lightmaps and >200k UV charts

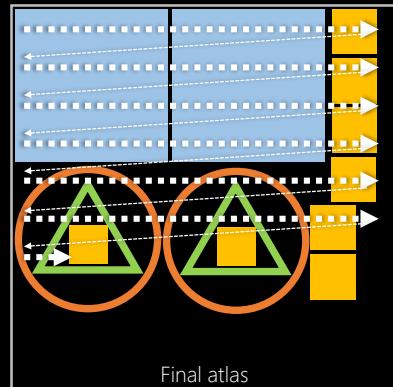


Ideally, what we want to do is pack all charts of all mesh instances in the entire scene into a single atlas. The aim is to perform a global optimization that will avoid the inefficiencies of local optimizations.

We wanted to implement a scheme that would be fast (our lightmaps are quite large) and would not require adding a secondary unique UV stream to all geometry.

Global packing algorithm

- Generate *bit masks* for every chart
 - Accounting for bilinear filtering
- Sort charts by their bounding box perimeters
 - Large to small
- Create an empty bit mask for the output atlas
- Insert each chart into the output atlas one by one
 - Brute force iterate through all atlas pixels
 - Intersect chart bit mask with atlas bit mask
 - Splat chart into the atlas mask if there is no intersection
 - *Static Lighting Tricks in Halo 4 [Boulton13]*
- This works, but is obviously very slow



The global packing algorithm that we've implemented is similar to one described in Static Lighting Tricks in Halo 4 GDC 2013 talk.

We generate bit masks for every chart and then sort them by bounds perimeter (actually just width+height, since that's equivalent). We then insert chart bit masks into an output atlas bitmask one at a time, from largest to smallest. The aim is to fill any holes produced during packing of large charts using smaller charts. The perimeter sorting heuristic is design to pack more "difficult" charts before "easy" ones. For example, 1024x1 chart is considered harder to pack compared to 32x32.

Packing is quite brute force / naïve. We intersect bitmasks of all charts with every possible location in the final atlas bitmask by sweeping the chart though all rows. The only optimization that we use here is testing up to 64 bits of the chart at a time against the bits in the atlas.

This is pretty slow and isn't really practical for typical Frostbite levels.

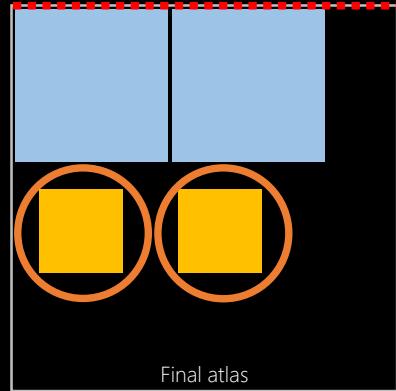
<https://archive.org/details/GDC2013Boulton>

Packing algorithm optimization

- Don't test every location!
- Start packing from the last successful row
- Reset start row when chart size decreases

```
int startRow = 0;
int sortKey = INT_MAX;
while (!packingQueue.empty())
{
    LightmapPackingNode node = packingQueue.pop();
    if (node.sortKey < sortKey)
    {
        startRow = 0;
        sortKey = node.sortKey;
    }
    startRow = bruteForcePack(atlas, node, startRow);
}
```

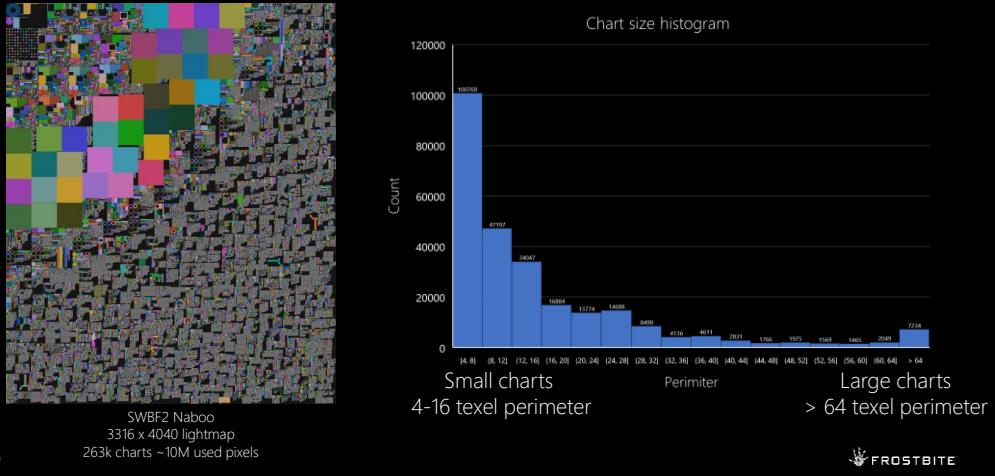
- Packing time drops from ∞ to <10s for production scale levels



We have added a very simple heuristic that allows us to skip brute force tests for large portions of the atlas, dramatically improving performance.

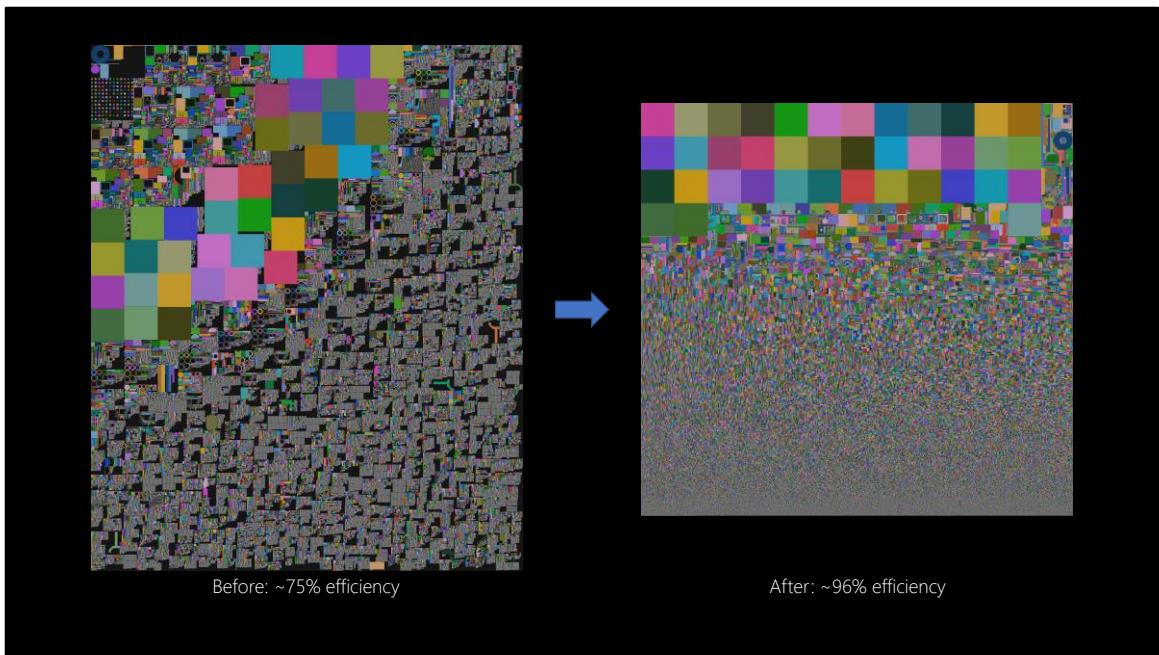
The idea is to simply start packing each chart from the row where we previously managed to fit a similarly-sized chart. When we move on to smaller chart size, we reset the packing start row to the beginning of the atlas, since smaller charts have a higher possibility to fit in some hole produced by larger charts.

Lightmap packing example

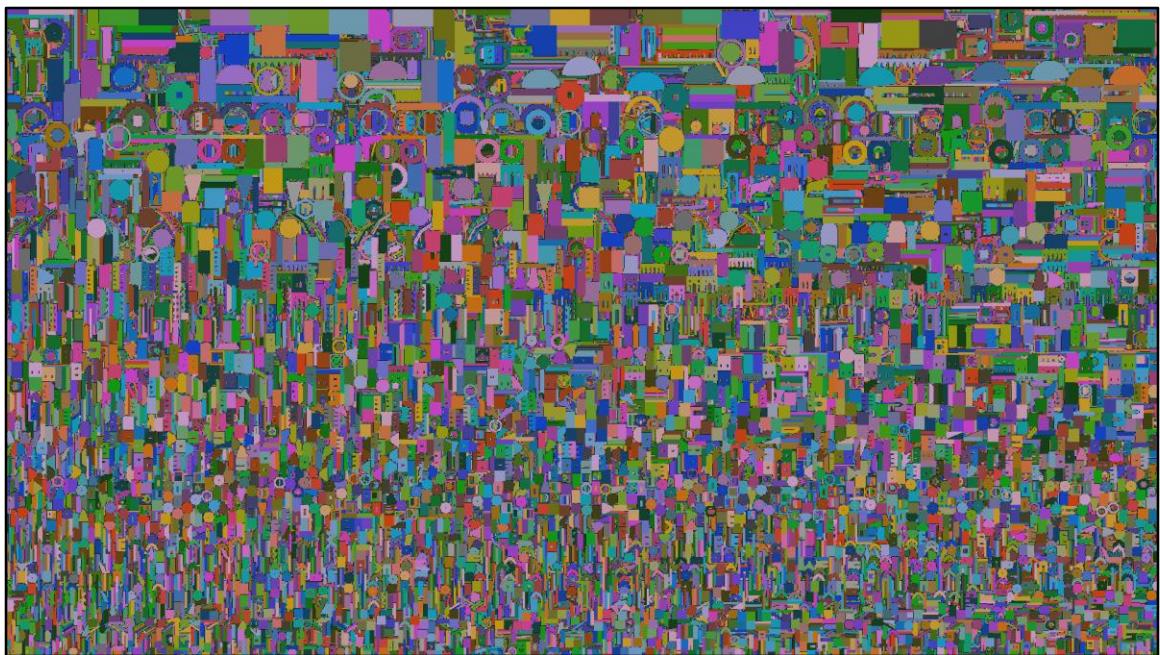


Example lightmap chart visualization from SWBF2. The chart size distribution histogram shows that there are vastly more tiny charts compared to large charts.

It also shows that there aren't too many different chart size buckets, which means that we don't end up resetting the packing row too often.



Lightmap for Naboo. Input: 3316 x 4040 (13MP), 10.1MP used, ~75% efficiency.
Output: 3316 x 3168, ~96% efficiency. 263k charts. 10sec packing time on 1 core @
2.6Ghz.



World's most difficult "Where's Waldo" puzzle.



No, there wasn't actually a Waldo on the last slide.

Unpacking transforms in the shader (1)

- Packing algorithm generates an array of 2d transforms
- Scale & offset **per chart** (not per instance)
- Need to access correct final packed UVs *somewhat*
 - Don't want to have a unique secondary UV stream
 - Don't want to break instancing

```
// Apply per-instance lightmap transform
float2 getLightmapUV(Vertex vertex, InstanceData instance)
{
    return vertex.uv * instance.lightmapScale
        + instance.lightmapOffset;
}
```

```
// Apply per-chart lightmap transform
float2 getLightmapUV(Vertex vertex, InstanceData instance)
{
    ????????????
    ???????????
}
```



Using the globally-packed lightmap requires jumping through a few hoops. While previously we could just provide a per-instance scale and offset for lightmap UVs, we can no longer do that.

Each mesh will have a few charts. Each chart now requires a unique transform. Moreover, each instance of each mesh will potentially have charts scattered through the atlas in a completely unique way.

Unpacking transforms in the shader (2)

- Store the `chart index` for each mesh vertex
- Write all lightmap chart transforms for the scene into a single buffer
 - Sorted by mesh instance index and chart index
- Store an offset into this buffer for each mesh `instance`
- Fetch correct final UV transform by combining two indices

```
float2 getLightmapUV(Vertex vertex, InstanceData instance)
{
    float4 uvTransform = uvTransformBuffer[vertex.chartId + instance.chartOffset];
    return vertex.uv * uvTransform.xy + uvTransform.zw;
}
```

- Extra indirection in the VS, but no measurable performance impact!



Our solution to this problem is to put all chart transforms into a single buffer, that's sorted by instance index and per-mesh chart index. We can then send the offset into the transform buffer as instance data. Per-instance offset is then added to the per-vertex chart index to get the final chart transform index.

Wrapping up



Takeaways

- Spherical harmonics lightmaps are great
- Path tracing is great
- One simple lightmap packing trick



Thanks

Diede Apers
Oscar Carlen
Sébastien Hillaire
Charles de Rousiers
Tomasz Stachowiak
Alban Wood

Christina Coffin
Chris Doran
Graham Hazel
Intel Embree team



References

- [Boulton13] Static Lighting Tricks in Halo 4, GDC 2013
- [Chen08] Lighting and Material of Halo 3, SIGGRAPH 2008
- [Green07] Efficient Self-Shadowed Radiosity Normal Mapping, SIGGRAPH 2007
- [Habel10] Efficient Irradiance Normal Mapping, SI3D 2010
- [Hazel15] Reconstructing Diffuse Lighting from Spherical Harmonic Data, CEDEC2015
- [Hillaire18] Interactive Global Illumination in Frostbite, GDC 2018
- [Iwanicki13] Lighting Technology of "The Last Of Us", SIGGRAPH 2013
- [Jakob10] Mitsuba Renderer
- [Martin10] A Real Time Radiosity Architecture for Video Games, SIGGRAPH 2010
- [Ramamoorthi01] An Efficient Representation for Irradiance Environment Maps, SIGGRAPH 2001



Questions?



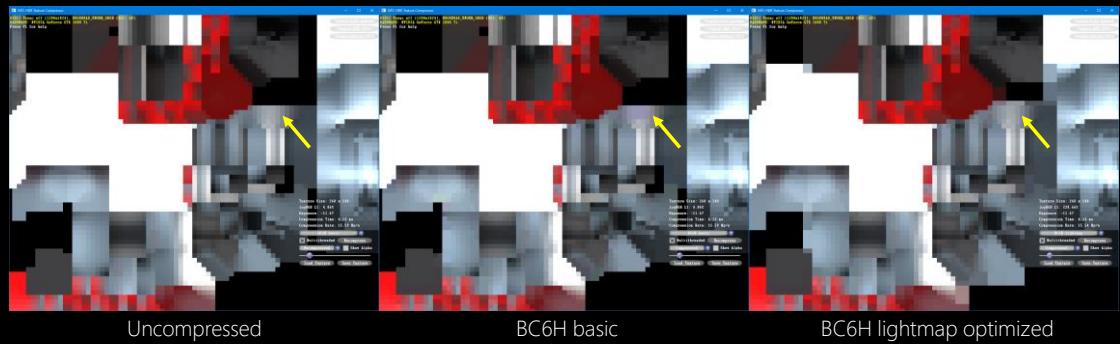
@YuriyODonnell



Bonus slides



Improving BC6H block compression (1a)



Uncompressed

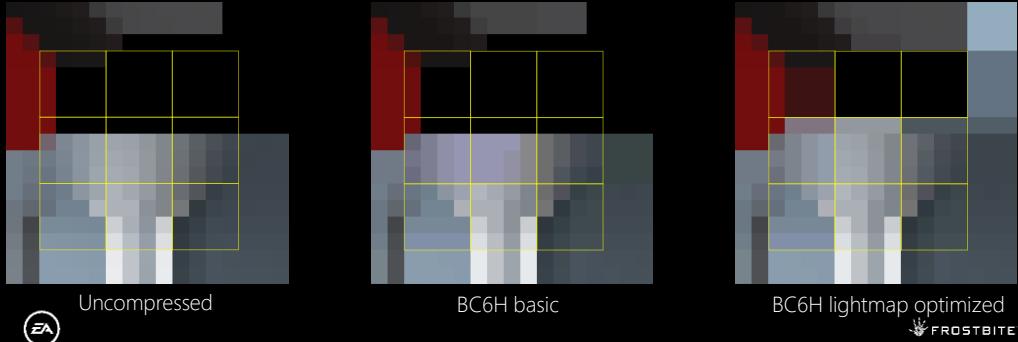
BC6H basic

BC6H lightmap optimized



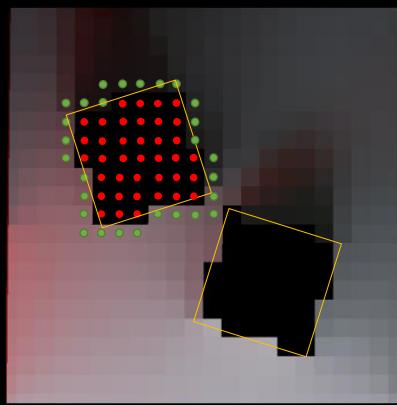
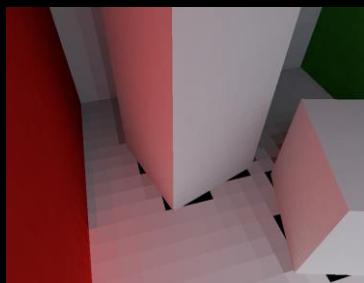
Improving BC6H block compression (1b)

- Unused pixels in the lightmap cause BC6H compression artifacts
- Zeros expand the 4x4 color block bounding box, reducing precision
- Replace invalid pixels with **average valid** value within each 4x4 block
- Keeps min, max and mean values of the color block intact



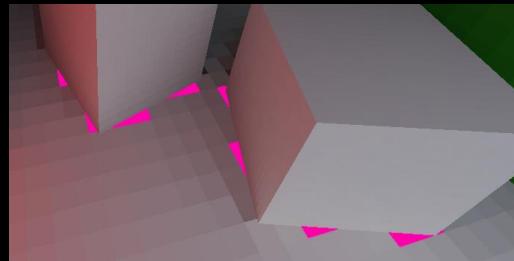
Dealing with overlapping geometry (1)

- Texels may be inside scene geometry
- Rays from those texels will hit back faces
- Ignore contribution from back face rays
- Discard texel if >50% of rays hit back faces

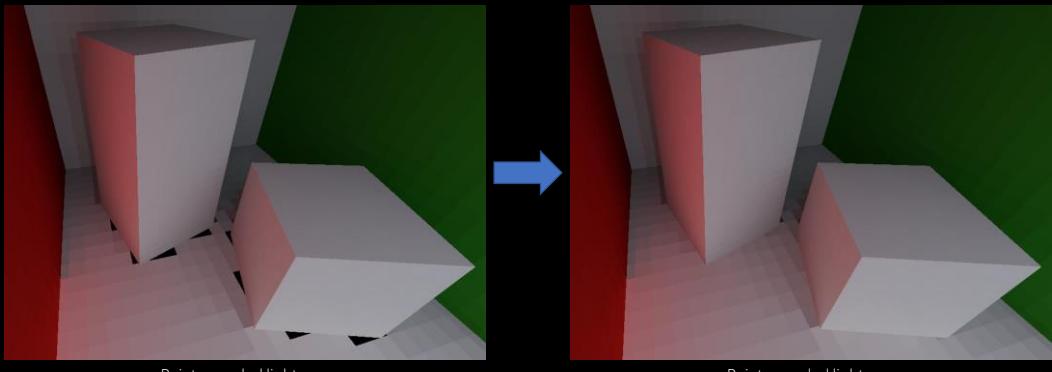


Ensuring correct bilinear interpolation (1)

- Lightmap may contain some invalid texels after baking
 - Texels that were discarded due to visible back faces (being inside geometry)
 - Texels that are part of the chart, but don't overlap UV space triangles
- Fill in any invalid texels with data from valid 8 neighbors (3x3 filter)



Ensuring correct bilinear interpolation (2a)



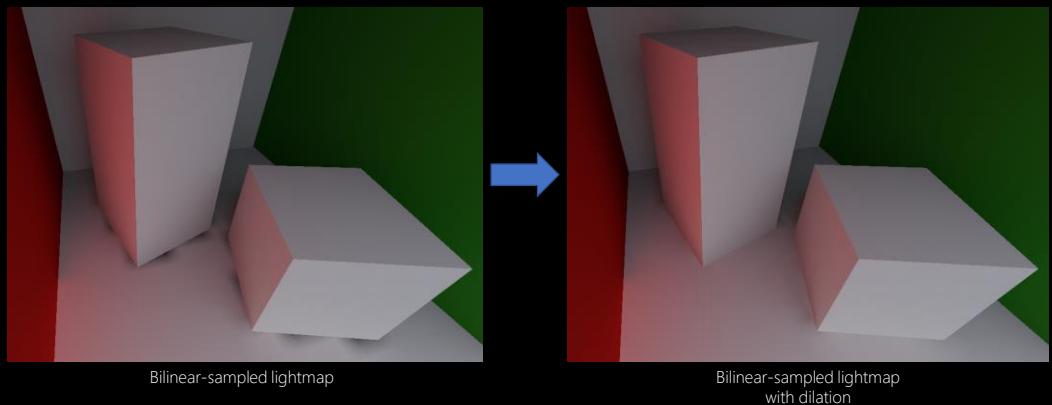
Point-sampled lightmap

Point-sampled lightmap
with dilation



FROSTBITE

Ensuring correct bilinear interpolation (2b)



FROSTBITE