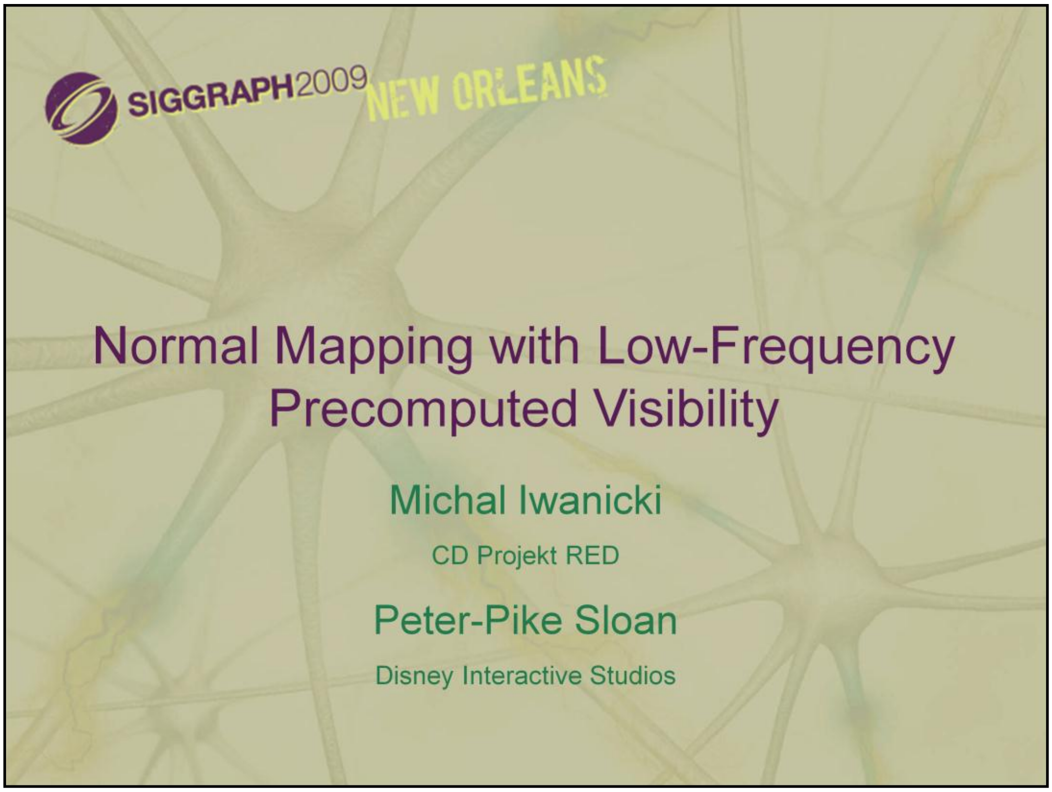
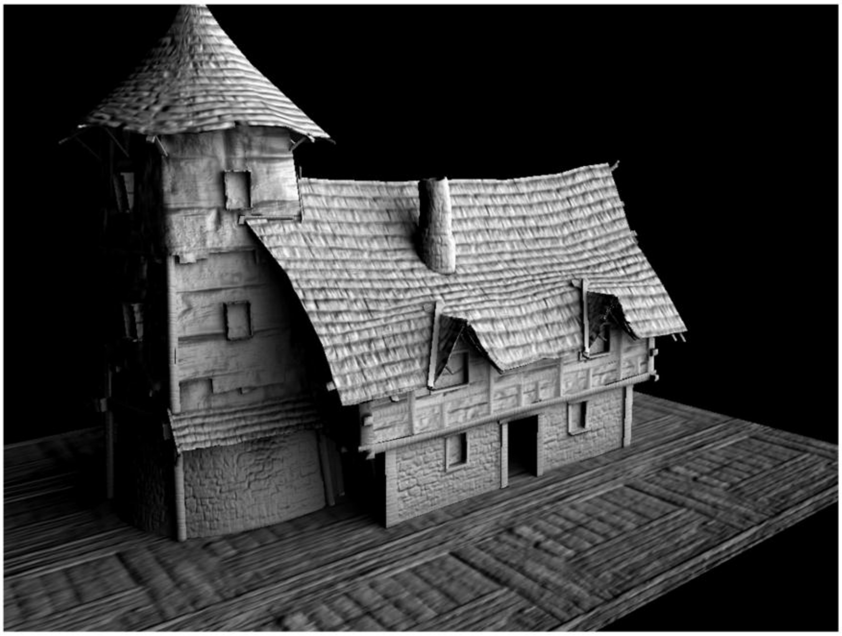


Hi everyone, my name is Michal Iwanicki, I work at CD Projekt RED, Polish game developer, this talk is titled ‘Normal mapping for low-frequency precomputed visibility’.

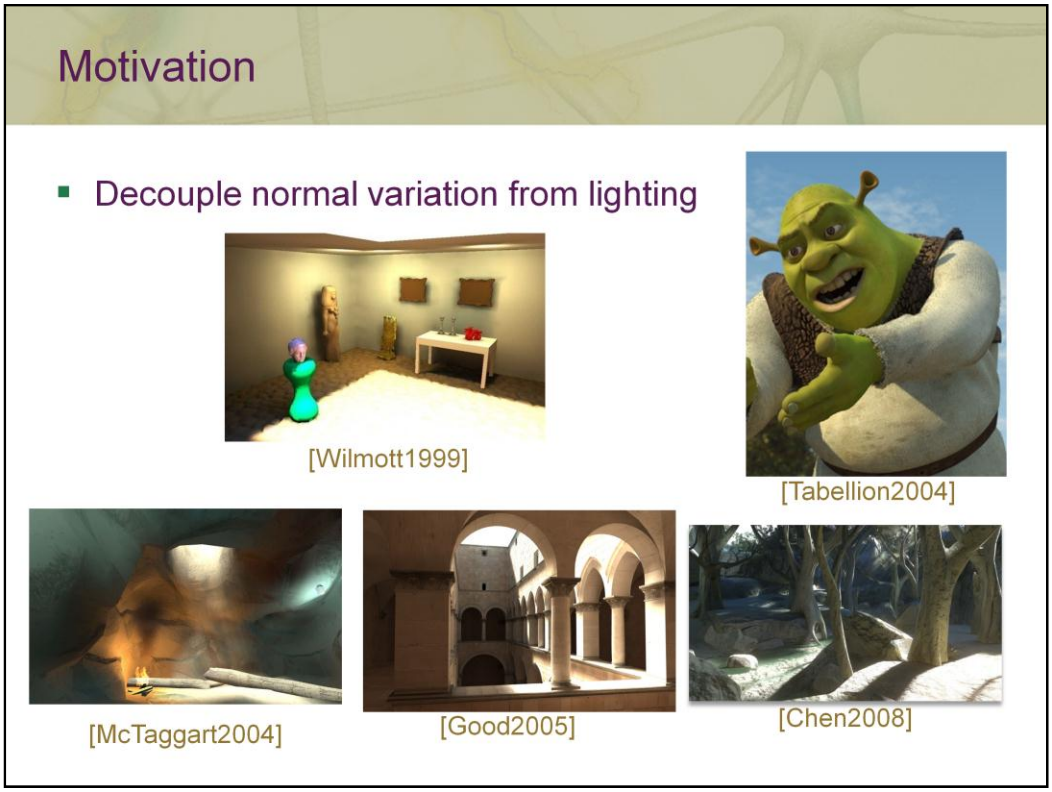
Today, we’re going to show you a way of combining PRT-like lighting with normal mapping, in a low-cost technique well suited for games and other real-time uses.

Here is an image of a house rendered with direct-light only PRT, while it has nice soft shadow, all details are created with geomety, the technique cannot use normal maps

Quick teaser…



Here we have the same model rendered using our techique, as you can see with normal mapping applied.



The goal of this project was to decouple high frequency normal variation from gross lighting.

There have been several attempts to do so for static lighting, like the work by Andrew Wilmott, who used a vector irradiance formulation of radiosity to accelerate precomputation. This later became the foundation for radiosity normal mapping used by Valve and many others.

Similar ideas have also been used in off-line rendering.

And, while those work really well, all assume static lighting. I personally work for a company which is focused on RPGs – and for such we simply cannot rely on a single static lighting environment. We need a solution that allows us to change lighting dynamically – to show passing time, changing weather etc.

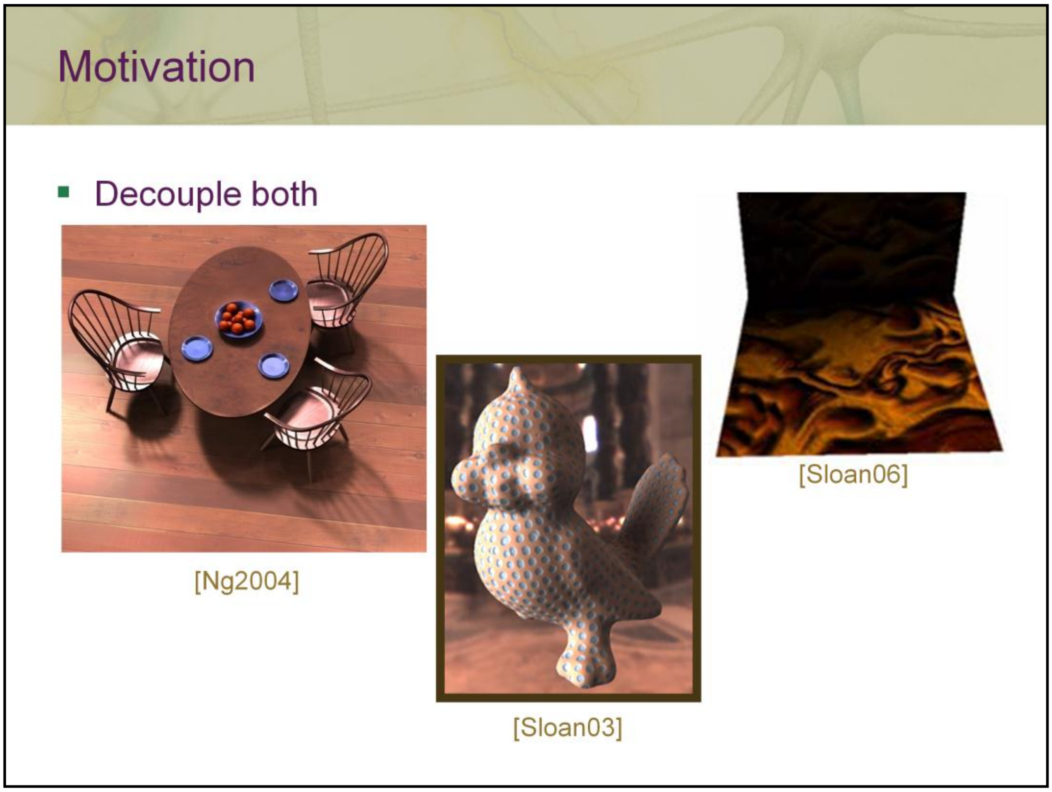
Motivation

- Decouple environment lighting

[Ramamoorthi01]

[Sloan02]

… and, there are solutions that allow us to dynamically change lighting. But they either don't model shadows or do not decouple normal variation.



Some solutions decouple both factors – local surface variation and lighting like the triple products by Ren Ng, bi-scale radiance transfer, and normalmapped prt by Peter-Pike. The first two are quite heavyweight techniques, not really suitable for games, the last one, to model local effect needs to store complex transfer matrices which also makes if diffucult to use in typical game-scenario. Our technique requires less memory, but can only handle soft shadows, not indirect lighting.

Decoupling normal variation from visibility

- Reflectance equation for direct lighting, diffuse surface:

$$I = \int_{\Omega} L(\vec{v}) V(\vec{v}) H(\vec{v}) d\omega$$

- Triple product integral
  - L changes every frame
  - V constant, unique at each scene point, smooth
  - H (normal) high frequency change, normal maps

So, let's start with a reflectans equation for a rigid object, with diffuse surface lit by distant lighting. Lets focus on direct only illumination.

Reflectance equation for such case is given by the following equtions – integral of lighting environmen (L) times visibility function (1 when light is visible, 0 when obstructed) times clamped cosine oriented along normal at given point over a sphere.

Let's formulate our assumpsions, our needs with regards to those three elements:
- we want light to by dynamic, possibly changing every frame, but we want it to be global for the whole scene
- visibility is unique for every point on the mesh, so it can't be tiled/etc, but as the model is rigid it doesn't change and, thus, can be precomputed. We can also assume that local surface details dont contribute to visibility, so it can be stored at a relatively low frequency
- the last element is cosine term, directly related to the normal vector. To model this element we want to use normalmaps, so it will change with highest frequency

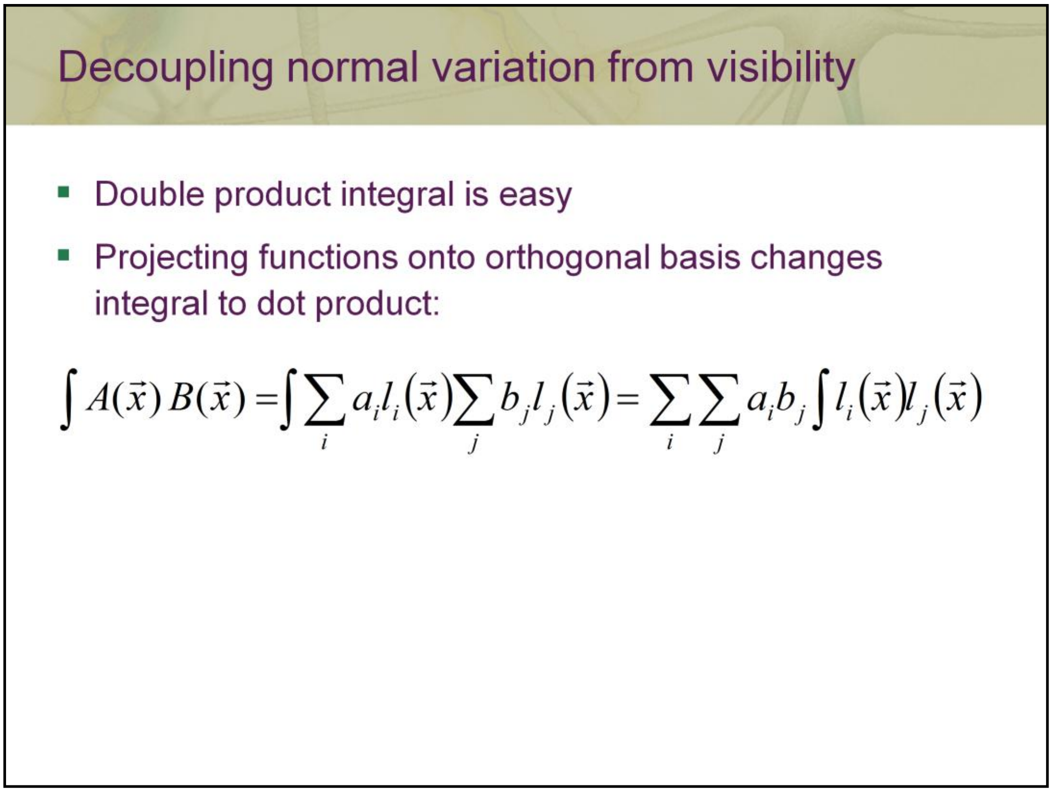
As you see it's a triple product integral.

Triple Products

- Various ways to compute
  - Using tripling coefficients [Ng2004]
  - Compute product of two functions and the double product rule

There are ways of computing such integrals. In terms of computer graphics the topic has been studied by Ren Ng et al. But like said before, this is really „heavyweight" technique.

One other option is to merge two of the functions and compute double product integral, which turns out to be a lot easier



This on the contarty is relatively easy.
Let’s say we have two functions, A and B. Let’s project them both on a common basis l. Properties of integerals and sums, allow us to rearrange it to the form on te right. And while this final form is still difficult to compute…

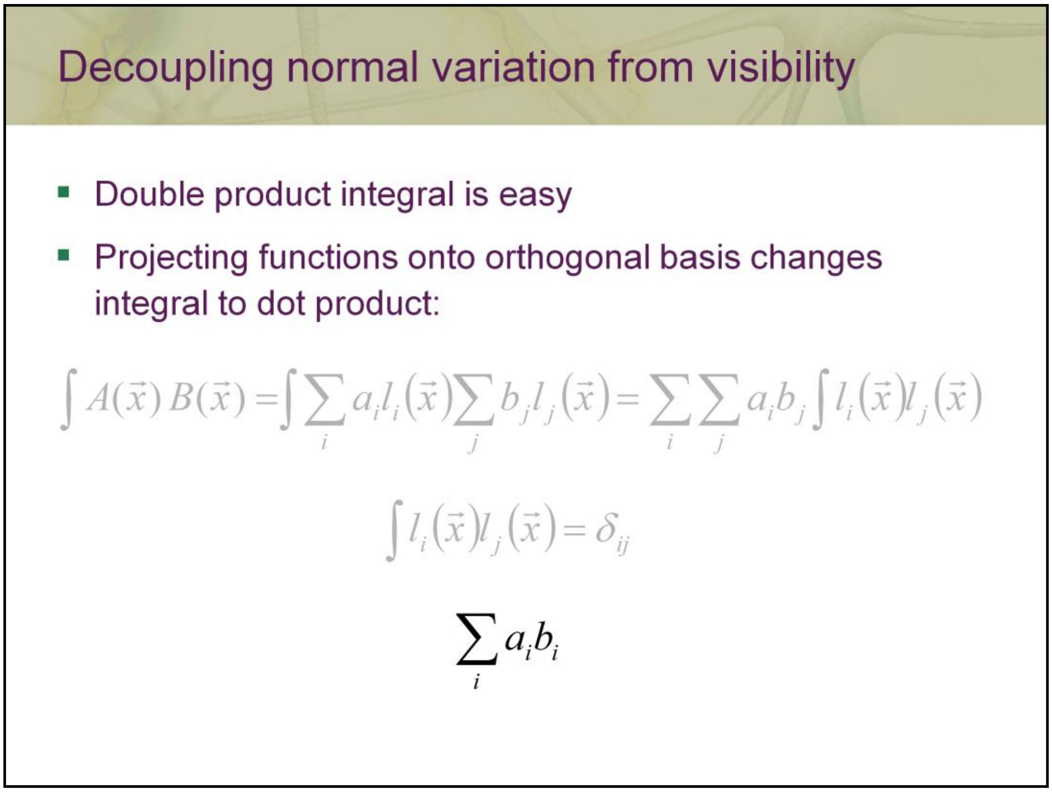
Decoupling normal variation from visibility

- Double product integral is easy
- Projecting functions onto orthogonal basis changes integral to dot product:

$$\int A(\vec{x})\,B(\vec{x}) = \int \sum_i a_i l_i(\vec{x}) \sum_j b_j l_j(\vec{x}) = \sum_i \sum_j a_i b_j \int l_i(\vec{x}) l_j(\vec{x})$$

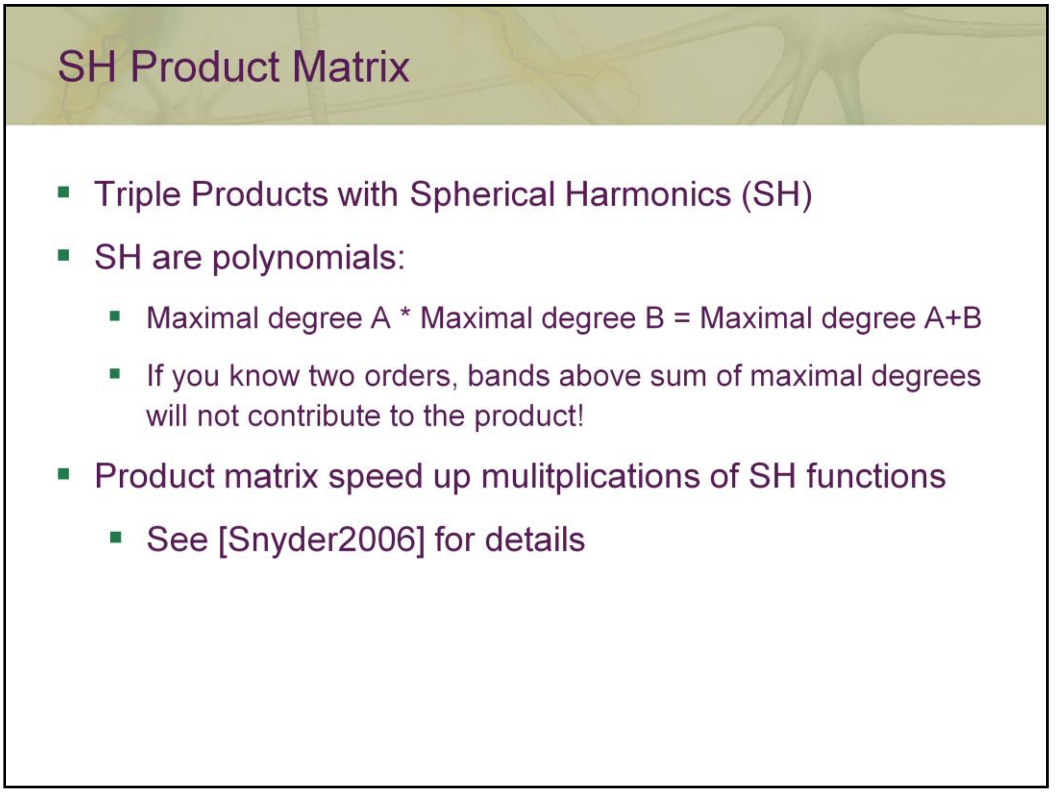
$$\int l_i(\vec{x}) l_j(\vec{x}) = \delta_{ij}$$

… if we decide to use an orthogonal basis, product of 2 basis functions will be Kronecker delta function, equal to 1 if the indices are equal and 0 if not



Which simplifies our integral to simple dot procuct between two sets of coefficients.

Most of the works in last few years used spherical harmonics, they are REALLY well studied, so we've just decided to stick to them.



As we're talking about spherical harmonic, it is worth to mention 2 things, which in fact are not really obvious.

- SH basis functions are simply polynomials, so when multipling two, let's say of degree A and B, the product will be a polynomial of a degree A+B.

Since SH are an orthogonal basis, it means then when we multiply 3 functions, that if you know the band limit of two of them, energy above the sum of their degrees will no contribute to the product. So when multiplying order 4 visibility and by order 3 clamped cosine, there is no need to use lighting of order higher then 7.

The second imporant thing is that we want to speed up multiple multiplications, if a function is a constant, we can use so-called product matrix. John Snyder has a very nice technical report that covers how to efficiently compute SH products and product matrices efficiently on his web page.

Decoupling normal variation from visibility

- [Sloan2002] uses SH as basis and combines V*H and L resulting in:

$$I = \int_{\Omega} L(\vec{v}) V(\vec{v}) H_{\vec{n}}(\vec{v}) d\omega = \sum_i l_i b_{\vec{n}i}$$

- Transfer function is parameterized by lighting
  - Normal is burned in
- So to capture small surface details, data must be stored with high resolution.

So let’s get back to the reflectance equation, and see how PRT utilized this „combine 2, and use double product” rule.
What PRT does is it combines visibility and cosine term into a single function, projects it to SH, and then, separtly projects visiblity to SH.
All that is required at runtime is to compute dot product bewteen the resulting sets of coefficients, but as you have probably noticed visibility is merged together with normal.
But, since visibility (and thus b vector) is unique across the mesh, we cannot tile or somehow reuse those values. Even though the visibility varies smoothly across the mesh, to capture small normal variations we need to store this data at very high resolution – which is not really practical.

Decoupling normal variation from visibility

- Small surface details don’t really contribute to visibility
- The idea is to combine equation components the other way – project L*H and V onto SH

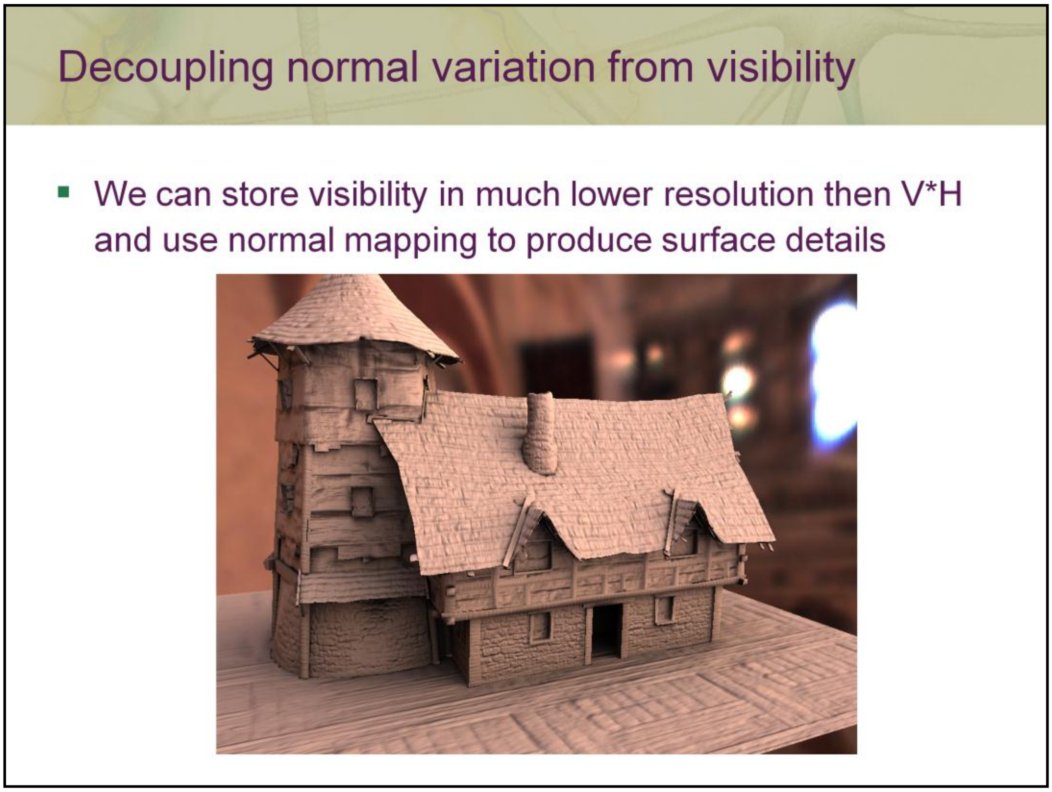
$$I = \int_{\Omega} \boxed{L(\vec{v}) H_{\vec{n}}(\vec{v})}\boxed{V(\vec{v})} d\omega = \sum_i lh_{\vec{n}i} v_i$$

So, the key to our idea is to combine the functions in different way.

At the very beginning we assumed that those small normal variations don’t contribute to visibility, so we dont need to combine those two terms togheter. We project visibility into one set of SH coefficient and then compute product of L times H and project the result into another set of coefficients.

As you can see in such way we can fullfill all of our initial assumptions. Visibility V is unique, but it is also constant (model is rigid) and relatively low-frequency and can be precomputed. LHn is unique for given direction and lighting environment, but as we show in a few seconds in can be easly precomputed and tabulated.

Exacly as with PRT, all that is required at runtime is to grab two sets of coefficients and compute a dot product.



As the visibility is stored totaly independently of the surface normal we can normalmaps to create local surface details, togheter with all, well known tricks – blending, tiling, etc.

Decoupling normal variation from visibility

- We store visibility function in textures, in a manner similar to lightmaps
- Multiple coefficients can be stored in separate textures or atlased (first option is more texture cache friendly)
- In the simplest form as floating-point textures, but more sophisticated encoding schemes can be used to save storage space/texture bandwidth [Ko2008]

Let's see how we store the data needed for the computation.

Storing visibility is pretty straightforward – as the model is rigid we can precompute it and store it in a manner similar to lightmaps – we create a unique UV mapping and for every texel store SH coefficients. The number of coefficients depends on the order of the visibility function we want to use. The higher order, the more accureate shadows we get, but we end up with higher number or coefficents.

We used order 3, 4, 6 with respectively 9, 16, 36 coefficients.

To store them we can either atlas the the textures, or use multiple textures, the second option being more texture-cache friendly.
In the simplest form we can just store floating-point values, but we can also use some more sophisticated encoding, like the one presented by Jerome Ko, saving both texture memory and bandwidth.

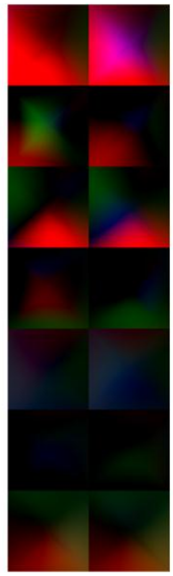
Decoupling normal variation from visibility

- L*H is dependant on direction of normal
- What direction should we store L*H for?
- All directions!

More interesting is how we store lighting time cosine function – as it depends on normal direction.
We have to tabulate value of this product for all the possible directions of normal vector.

Storing L*H

- Dual-paraboloid projection [Heidrich1998]
  - Combined with square to disk mapping (to ensure C0 continuity [Shirley1997])
  - We atlas multiple coefficients in a single texture



And while natural solution for such lookup table is cubemap, the problem is that, on the contary to visibility, we need to store those coefficients for each color channel – which gives us 27 coeffcients for order 3, which is quite a lot – it gives us 7 cubes maps. Additionally, on older hardware we dont get filtering between cubemap faces. So we have chosen dual paraboloid map as a final form of our look-up table. 2 hemispheres are laid out side by side on the texture, consequent coefficents are atlases one below the other. To ensure C0 continuity we combine it with squre-to-disc mapping. This texture doesn't hae to be large, 32x32 for single hemisphere is enough.

Decoupling lighting environment

- For different lighting environment we need different L*H textures
- We can blend between some pregenerated textures
  - If the changes are not drastic (e.g. during daytime) blending can be done on CPU
- Or we can generate them on fly
  - Can be done very effectively by tabulating all required Hn functions and multipling them by lighting SH product matrix (using SSE)

So now we have decoupled visibility from normal variation, but we also wanted to dynamically change lighting conditions.
As you may expect, for different lighting environments we need separete L*H lookup textures.
As those textures contain SH coefficients, we can prepare several and blend then at runtime – either on CPU or GPU.
Or, we can generate them on fly. The key here is the formentioned product matrix. We start by tabulatig cosine function for all required directions prior to rendering, and then, each frame multiply them by product matrix created from lighting function, obtaing a LHn texture. SSE makes it fast enough for real-time use.

A6

Compression of Visibility

- To reduce storage requirements (especially for higher order SH) we may compress visibility
- Rewriting the equation for PCA-compressed visibility gives us:

$$I = \int_{\Omega} L(v) \sum_{n=1}^{M} \left(w_n V_n(\vec{v}) + V_e(\vec{v}) \right) H(\vec{v}) d\omega =$$

$$\sum_{n=1}^{M} w_n \left(\int_{\Omega} L(\vec{v}) V_n(\vec{v}) H(\vec{v}) d\omega \right) + \int_{\Omega} L(\vec{v}) V_e(\vec{v}) H(\vec{v}) d\omega$$

And while this works really nice, there are some minor issues – mainly storage costs for visibility function.
To get more accrate results we need to use higher order spherical harmonics – which result in high number of coefficients. To reduce the amount of data wee need to store we decided to try to compress visibility functions.

We use PCA for that. Rewriting the reflectance equation to use compressed visiblity gives us the following form, which after few rearrangements reduces to sum of integral incorporating visibility mean and weighted sum of integrals with principal components.

Compression of visiblity

- In such case instead of V function we store M weights per visiblity texel
- Instead of LHn textures we store M+1 textures with precomputed products calculated for PCA basis vectors
  - We generate them using product matrix derived from PCA basis vectors
- We don't even need to store them explicitly, but pass L*PCA basis to shaders and perform convolution with H on the fly (when using 3rd order approximation of cosine function we only need to send 9 coefficients per color channel)

So now, instead of storing visibity we only need to store M weights, and instead storing LH texture we store those preintegrated textures. They are ordinary textures, storing just the color data – in fact, as you may have notices those are just irradiance environment maps.

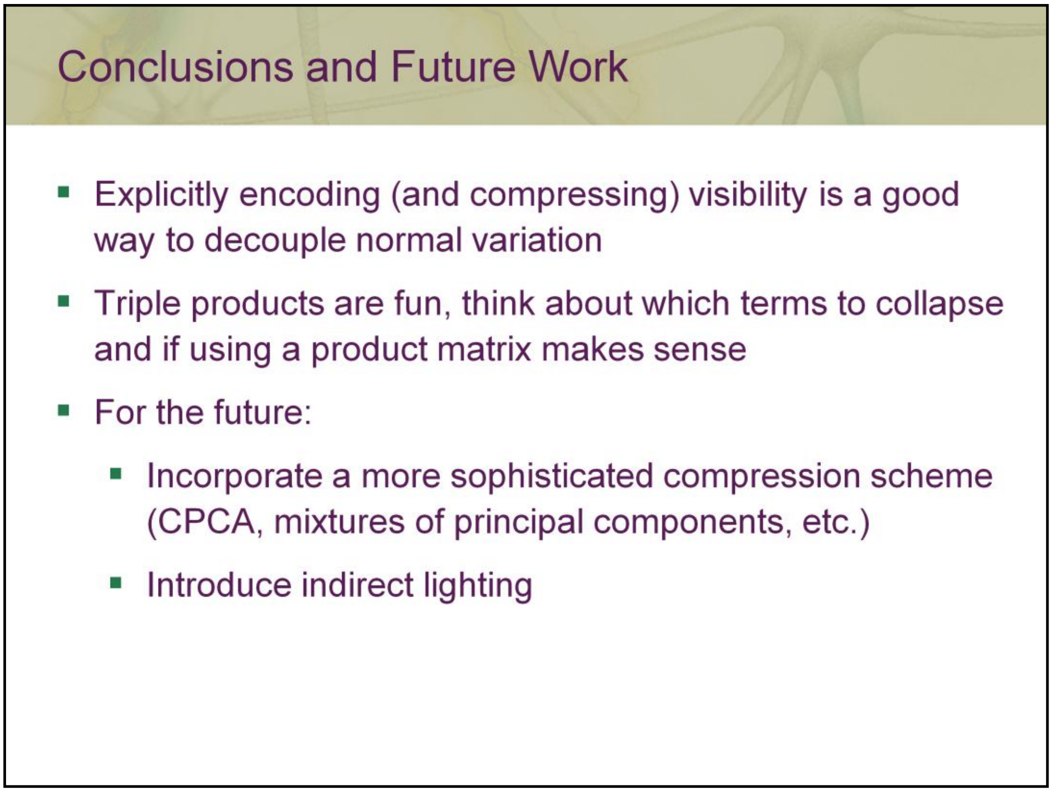
We don't even need to store them explicitly as texture, but just send those PCA basis vectors multiplied by lighting to the GPU and perform convolution with H on the fly.

Compression of visiblity

- Compressed visibility turns out not only to be more memory-efficient but also faster

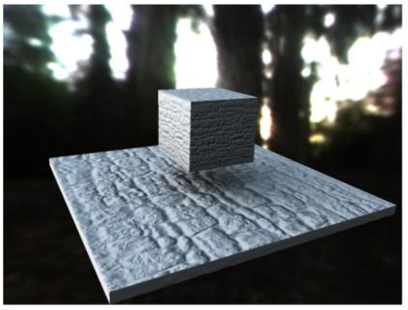
	With LHn texture update	Without LHn texture update
Uncompressed 6th	107 fps	110 fps
Uncompressed 4th	282 fps	302 fps
PCA (24)	153 fps	161 fps
PCA (16)	273 fps	289 fps
PCA (12)	370 fps	400 fps

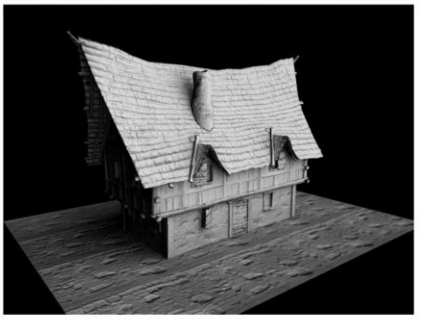
Table compares performance of several options. Measurement were done on Core2 CPU with NVIDIA Quadro 3600M.
Table shows performance for uncompressed visibility of order 6 and 4, and compressed version using 24, 16 and 12 PCA vectors. As you can compression is a huge win, especially that this way, we're not bound to any specific SH order, we can use higher order, and then approximate it with as many coefficients as we need to.

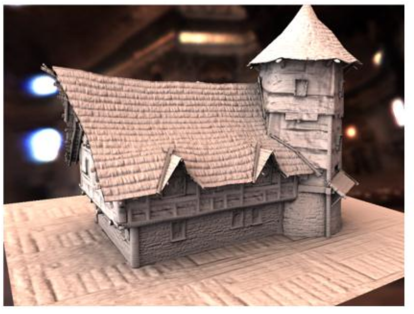


To conclude: explicitply encoding and compressing visibility is a good way to decouple normal variaion.
For the future, we would like to experiment with more sophisitcaed compression schemes, like CPCA, and introduce indirect lighting.

Demo







Acknowledgements

- Tomasz Polit, Michał Buczkowski for models
- Paul Debevec for Light Probes
- Hao Chen, Otavio Good, Garry McTaggart, Ren Ng, Eric Tabellion, Andrew Willmott for figures from papers
- Dreamworks for figure from 2004 Siggraph paper
- Questions?