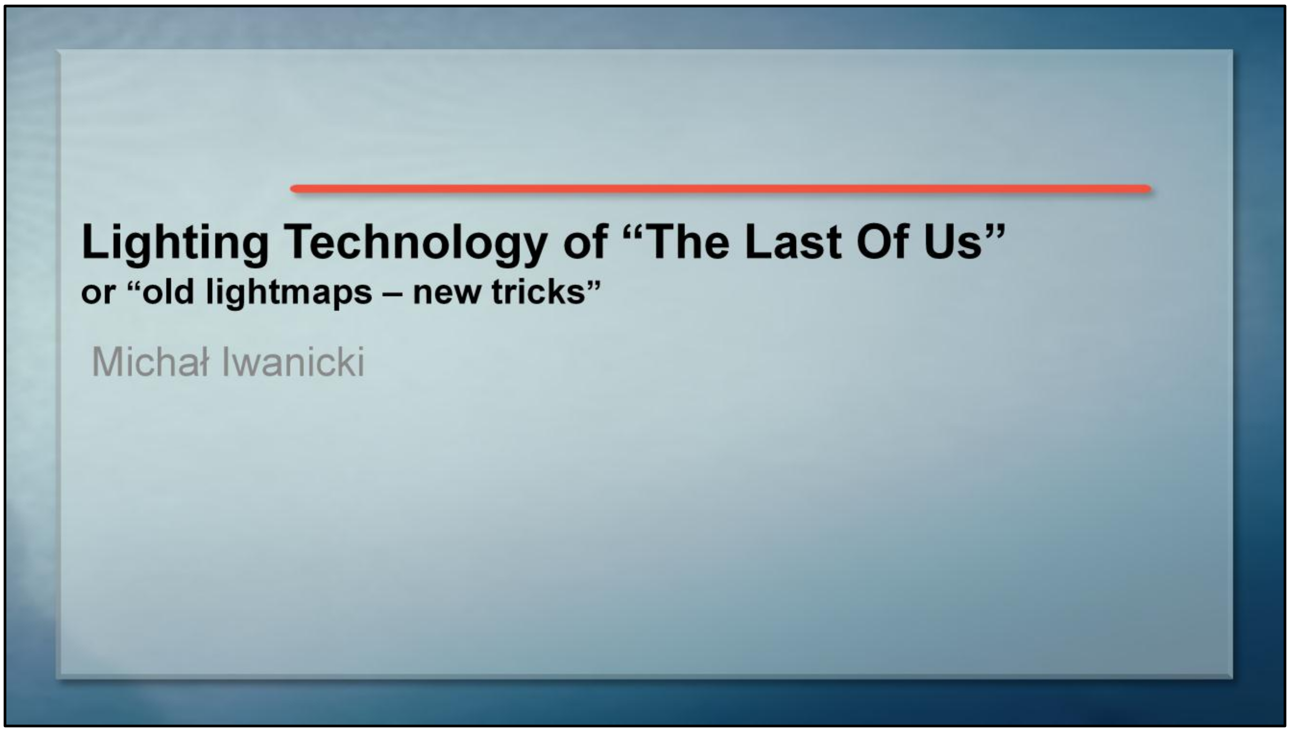
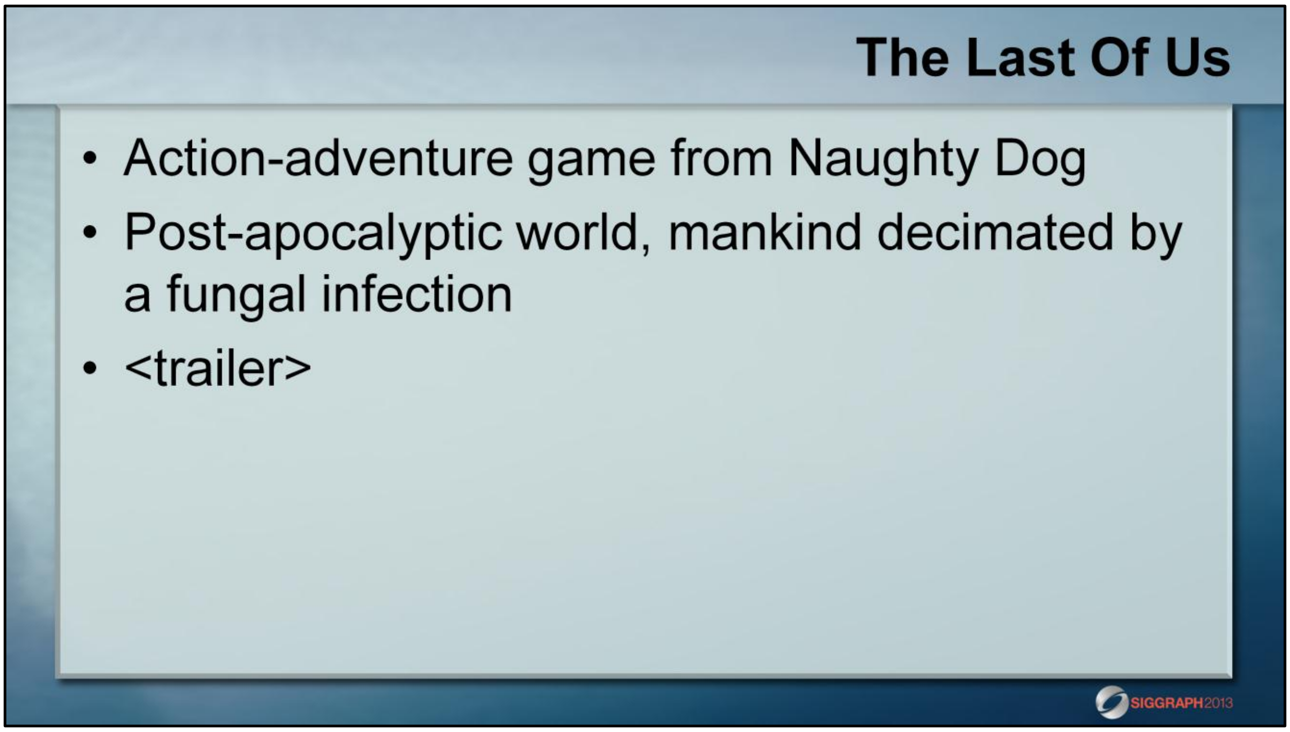


Lighting Technology of “The Last Of Us”
or “old lightmaps – new tricks”

Michał Iwanicki

Hi everyone, my name is Michał Iwanicki. I’m an engine programmer at Naughty Dog and this talk is entitled: “Lighting technology of The Last of Us”, but I should have called it “old lightmaps – new tricks”

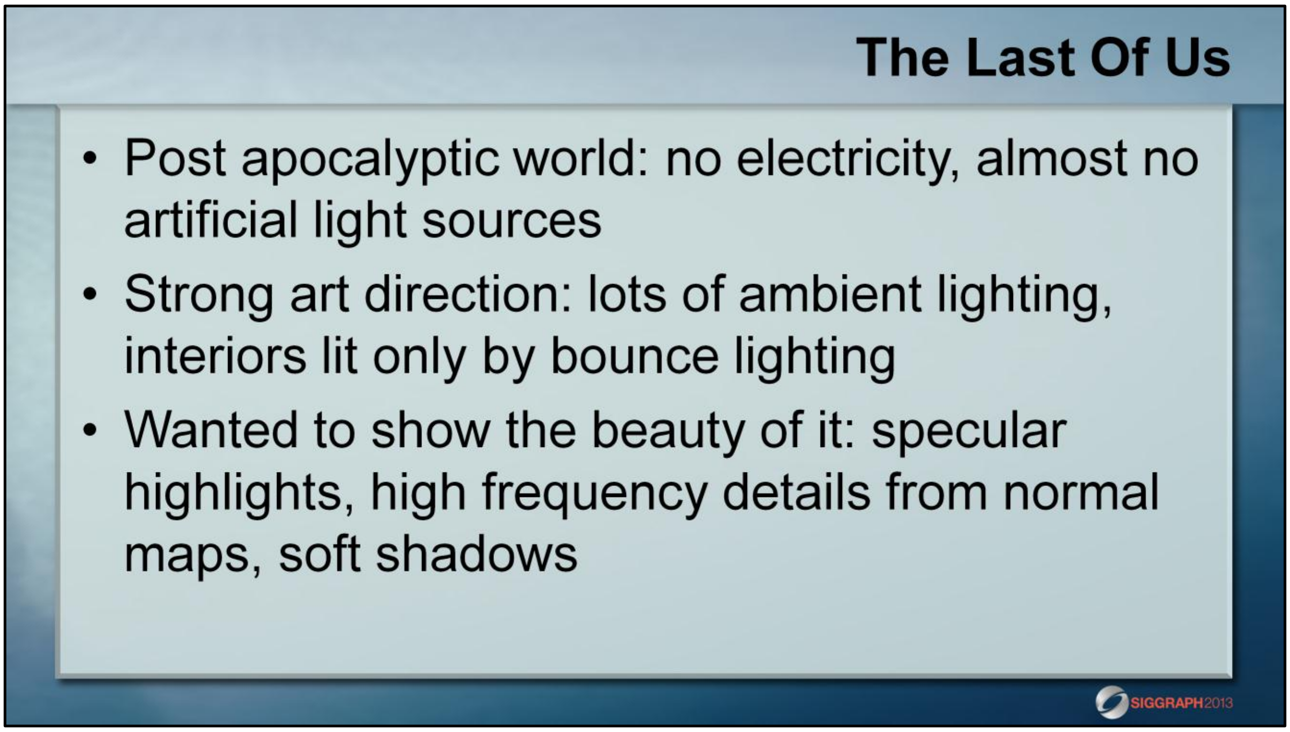


The Last Of Us

- Action-adventure game from Naughty Dog
- Post-apocalyptic world, mankind decimated by a fungal infection
- <trailer>

SIGGRAPH2013

The Last Of Us is an action-adventure, survival-horror game, with a heavy emphasis on story and characters. It takes place in the near future, where mankind has been decimated by a mind-controlling fungus. If you haven't seen the game yet, here's a short trailer…

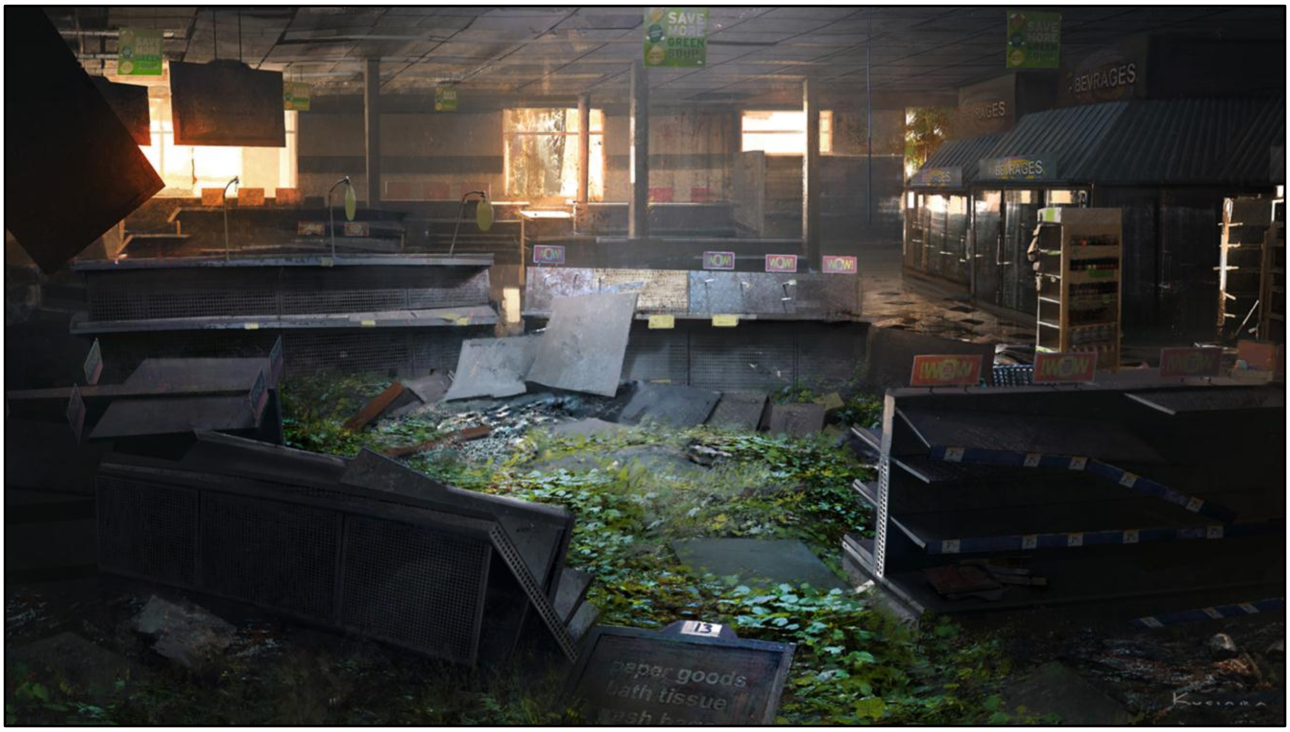


The Last Of Us

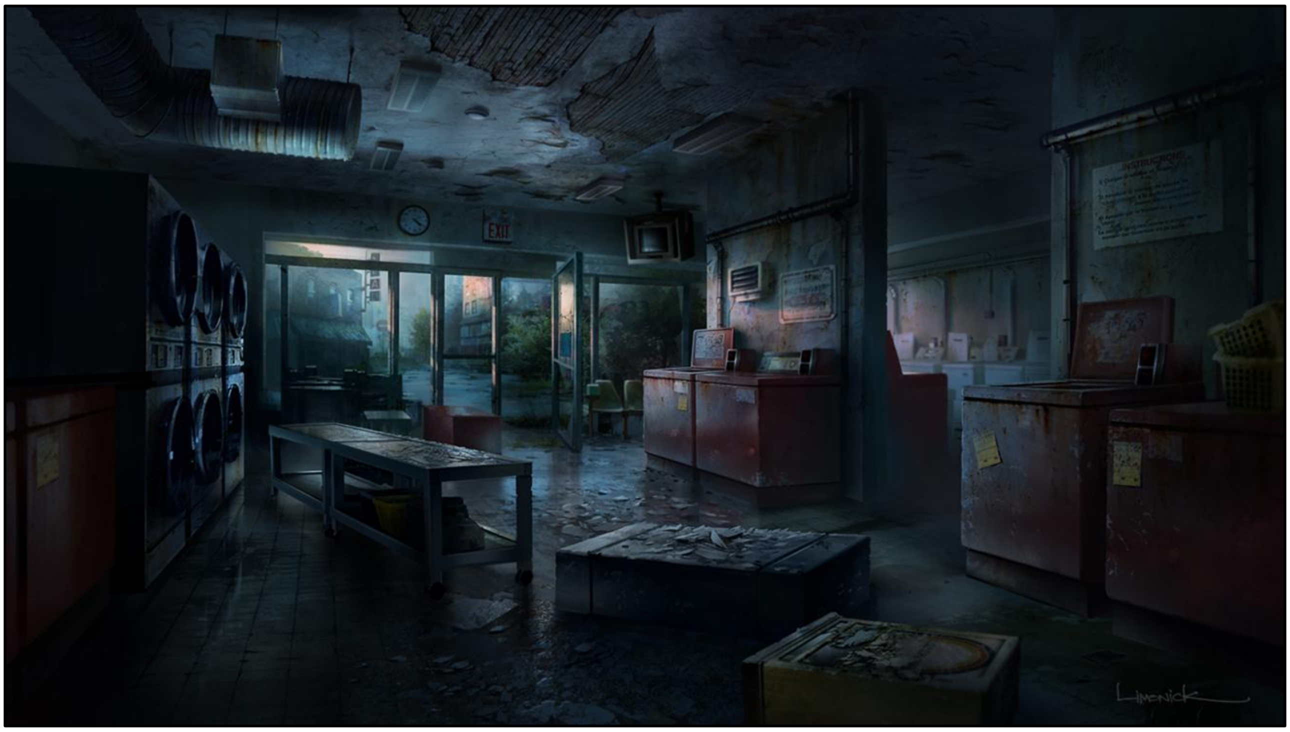
- Post apocalyptic world: no electricity, almost no artificial light sources
- Strong art direction: lots of ambient lighting, interiors lit only by bounce lighting
- Wanted to show the beauty of it: specular highlights, high frequency details from normal maps, soft shadows

SIGGRAPH2013

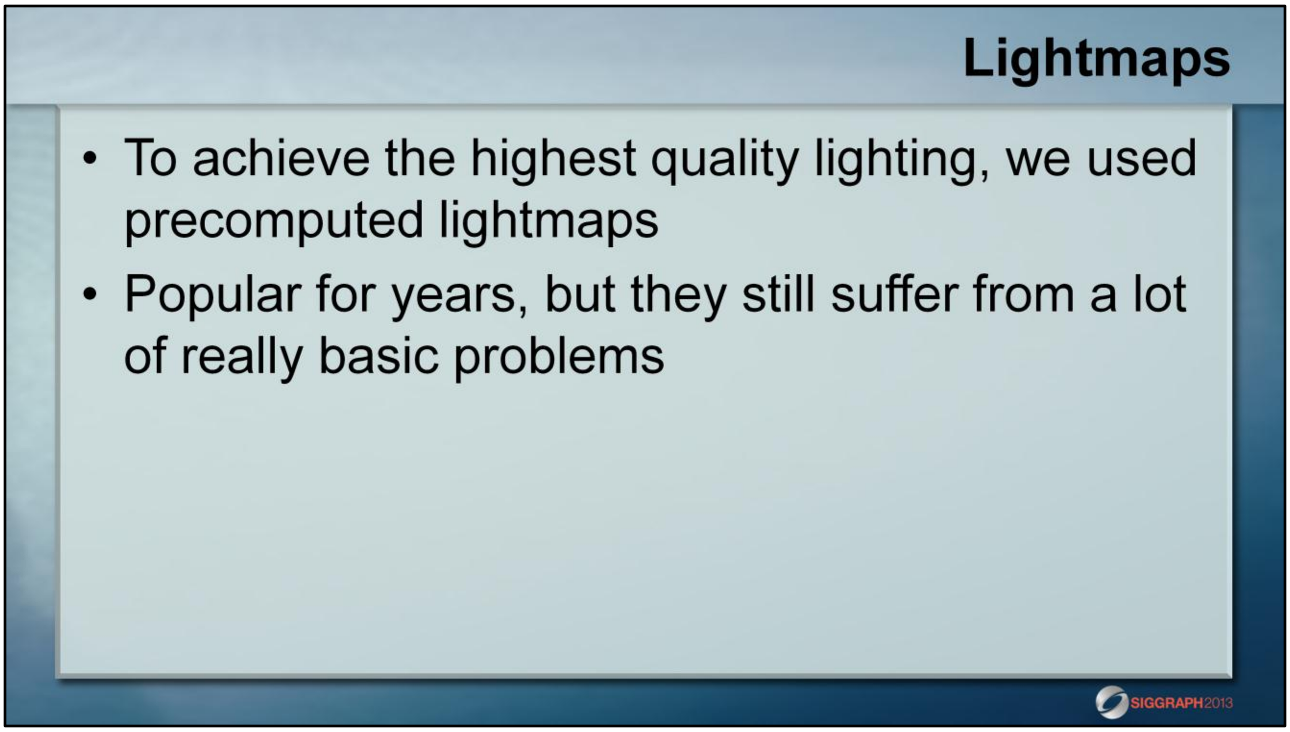
Since the game takes place in a post-apocalyptic world, there's almost no electricity, so no artificial light sources. The majority of the lighting comes from the sun and the sky, so most of the environments are lit just by bounce lighting.
There was a very strong art direction: we wanted to show the beauty of this lighting, its softness, how it interacts with different surfaces, the soft shadows it creates. We wanted to show specular highlights of the indirect illumination and show all those subtle details modeled with normal maps.



Just to give you an overview of what we were after, here's some concept art…



As you can see, all the lighting is very soft, but you still get a very clear read of all the surface details.

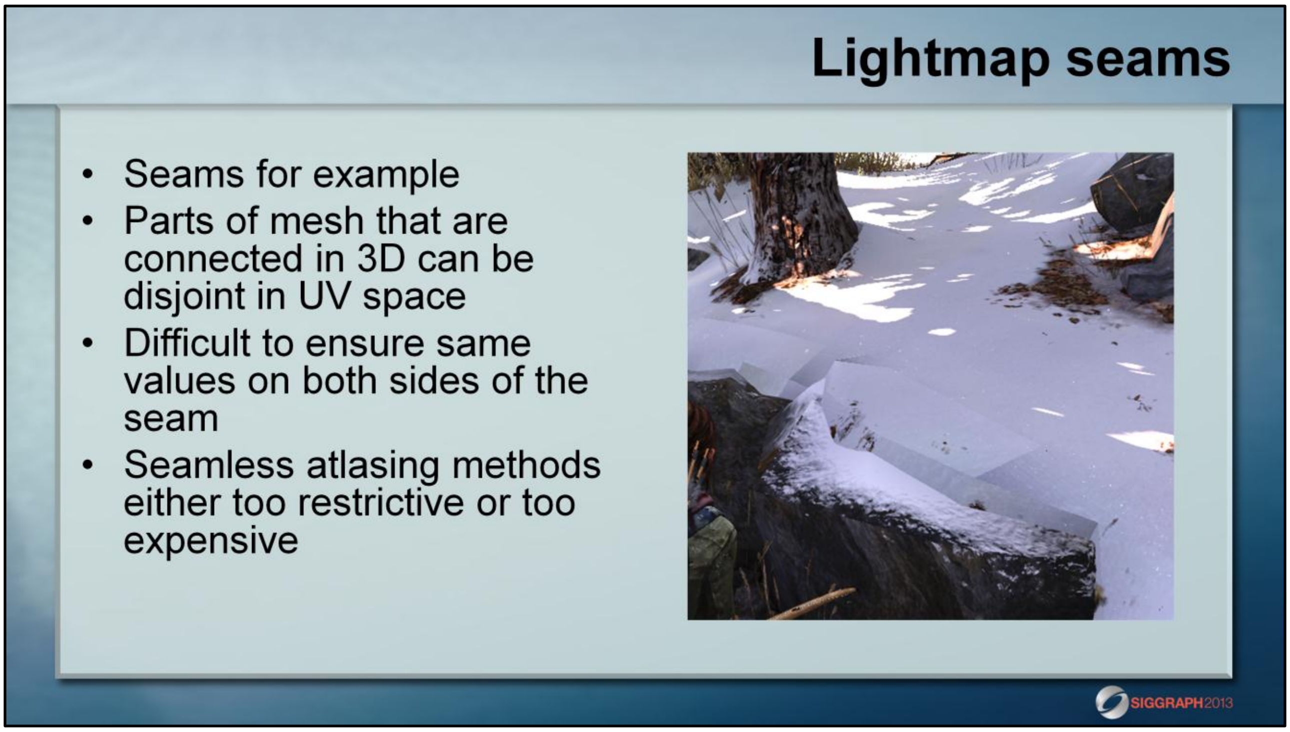


Lightmaps

- To achieve the highest quality lighting, we used precomputed lightmaps
- Popular for years, but they still suffer from a lot of really basic problems

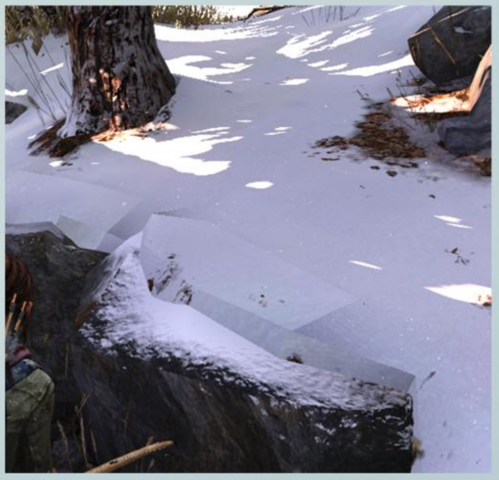
SIGGRAPH2013

From the very beginning, we knew that to achieve this level of quality, relying purely on runtime lighting would not be enough. We knew that we had to compute our lighting offline, so we decided to use lightmaps.
The problem with lightmaps is that even though the game industry has been using them for years now (since Quake 1?), they still suffer from really fundamental issues…



Lightmap seams

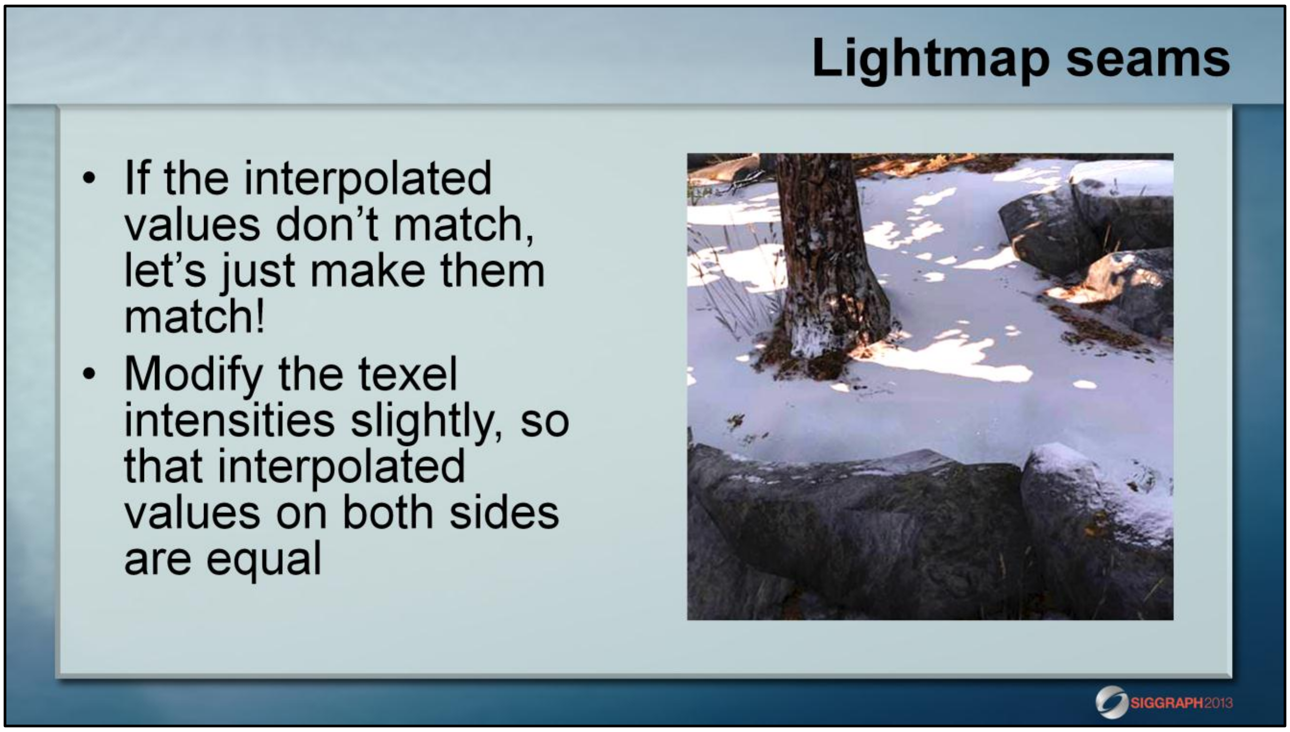
- Seams for example
- Parts of mesh that are connected in 3D can be disjoint in UV space
- Difficult to ensure same values on both sides of the seam
- Seamless atlasing methods either too restrictive or too expensive



SIGGRAPH2013

Like the seams: the discontinuities in the lighting.
It can happen that parts of the mesh that are contiguous in 3D space are mapped into disjoint regions in UV space, and the interpolated lighting values along split edges don't quite match, which causes visible seams.
While methods to deal with this do exist, they either generate additional cost at runtime or impose restrictions on shape and/or placement of charts in uv space, leading to inefficiencies in the uv space usage, which we wanted to avoid.

Seamless atlasing methods can be used but with the size of our levels, they were too costly to integrate into our pipeline.

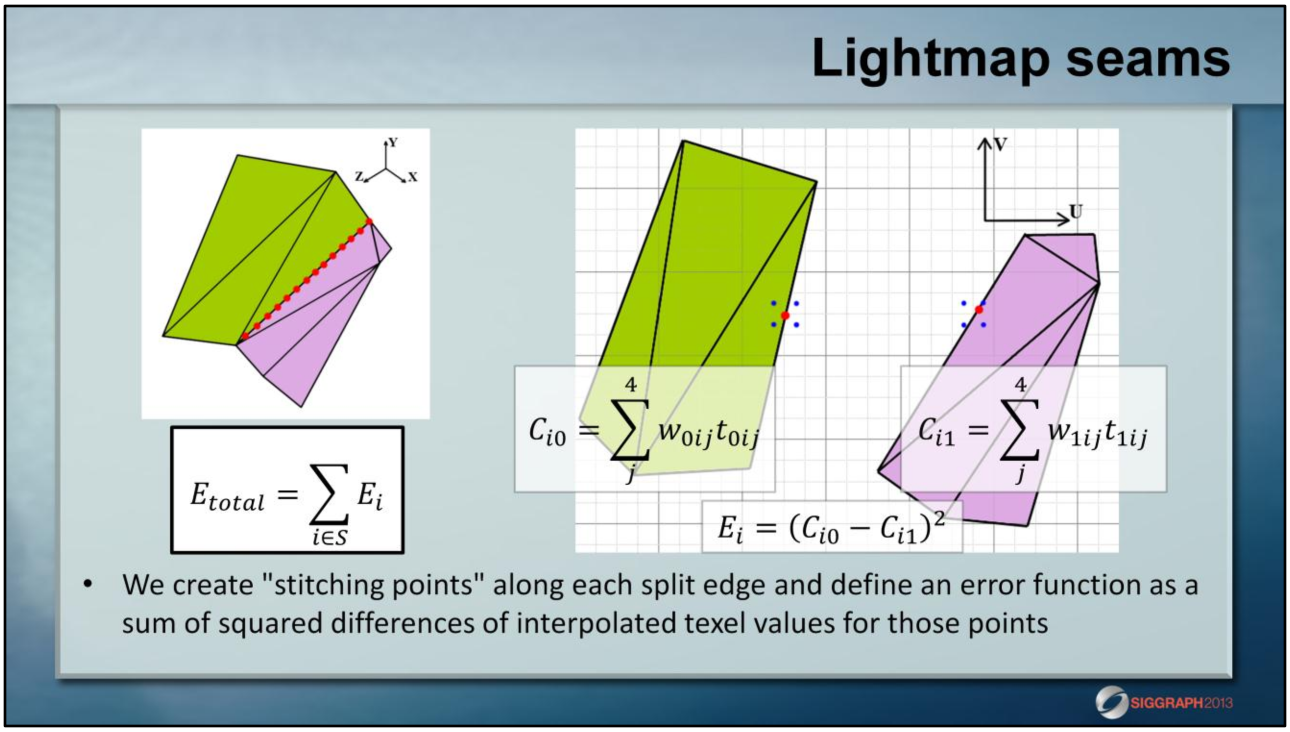


Lightmap seams

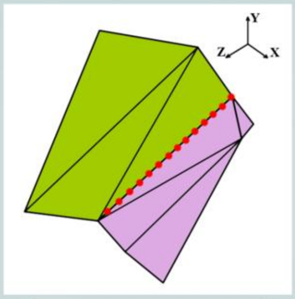
- If the interpolated values don't match, let's just make them match!
- Modify the texel intensities slightly, so that interpolated values on both sides are equal

SIGGRAPH2013

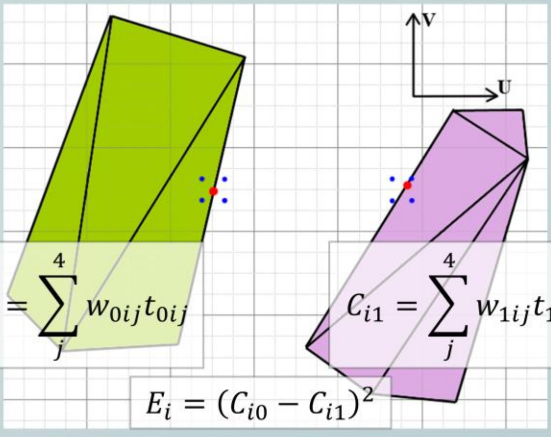
We found a pretty simple, yet effective solution: if the interpolated values on both sides of split edges don't match, let's just make them match, by slightly modifying the intensities of the surrounding texels.



Lightmap seams



$$E_{total} = \sum_{i \in S} E_i$$



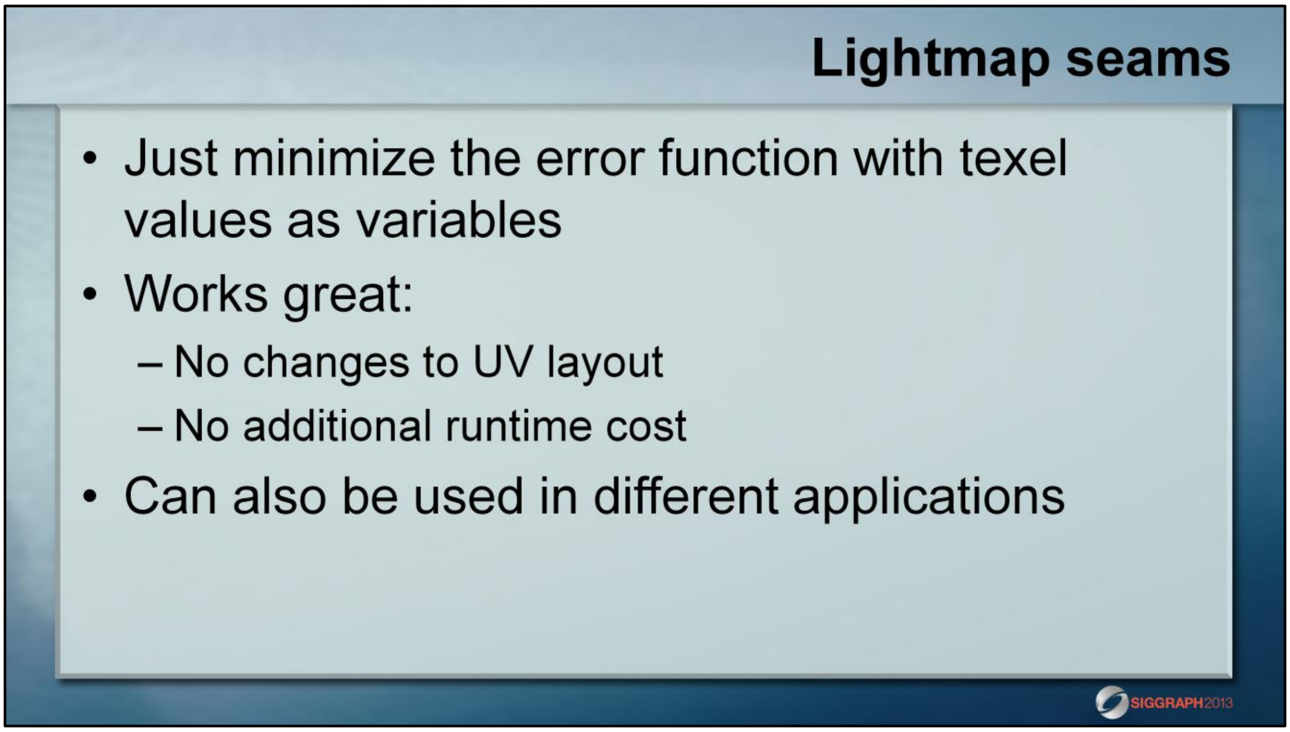
$$C_{i0} = \sum_{j}^{4} w_{0ij} t_{0ij}$$

$$C_{i1} = \sum_{j}^{4} w_{1ij} t_{1ij}$$

$$E_i = (C_{i0} - C_{i1})^2$$

- We create "stitching points" along each split edge and define an error function as a sum of squared differences of interpolated texel values for those points

Let’s take a look at the example: a mesh in 3D space is atlased into 2 disjoint regions in UV space. Along the split edge we create a number of “stitching points” (we create three points per texel, since the bilinear interpolation follows a quadratic curve).
Let’s examine one of those stitching points. The value in the stitching point for the chart on the left, Ci0, is the interpolated value of four surrounding texels, marked in blue.
Same with the value on the other side of the edge: Ci1.
We want Ci0 and Ci1 to be equal.
We can define the error for this stitching point as a squared difference between them.
And then we define the error function for the whole lightmap, which is the sum of the errors for individual stitching points.

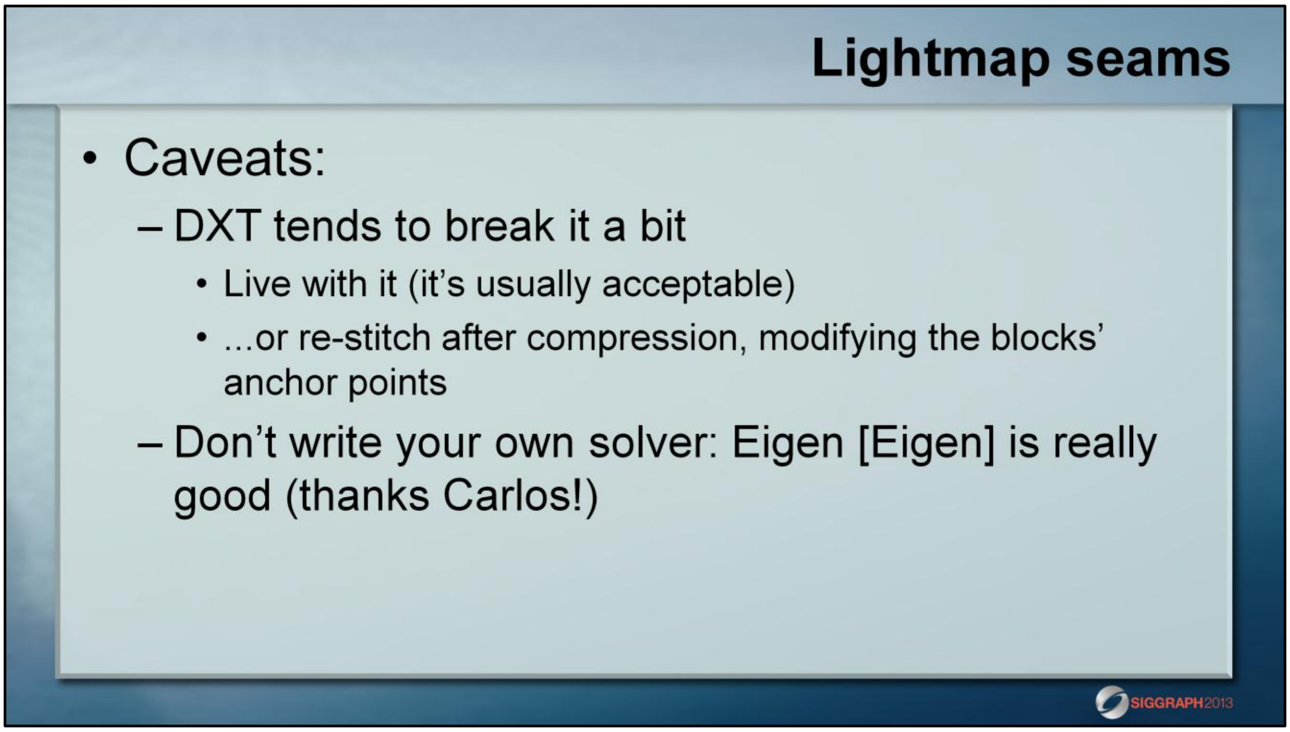


Lightmap seams

- Just minimize the error function with texel values as variables
- Works great:
  - No changes to UV layout
  - No additional runtime cost
- Can also be used in different applications

SIGGRAPH2013

Once we have the error function we simply minimize it using least squares, treating texel values as variables. The optimization modifies the texel values slightly and makes sure that the total error is minimized. To make sure that the values don't deviate from the originally calculated ones, we add additional constraints that penalize deviation from the calculated values, with a user-controllable strength.
It works surprisingly well: it doesn't require any changes to the UV layout and doesn't generate any additional cost at runtime; it can be easily applied to already existing maps.
This isn't limited to lightmaps: the method can be used on other types of maps as well.

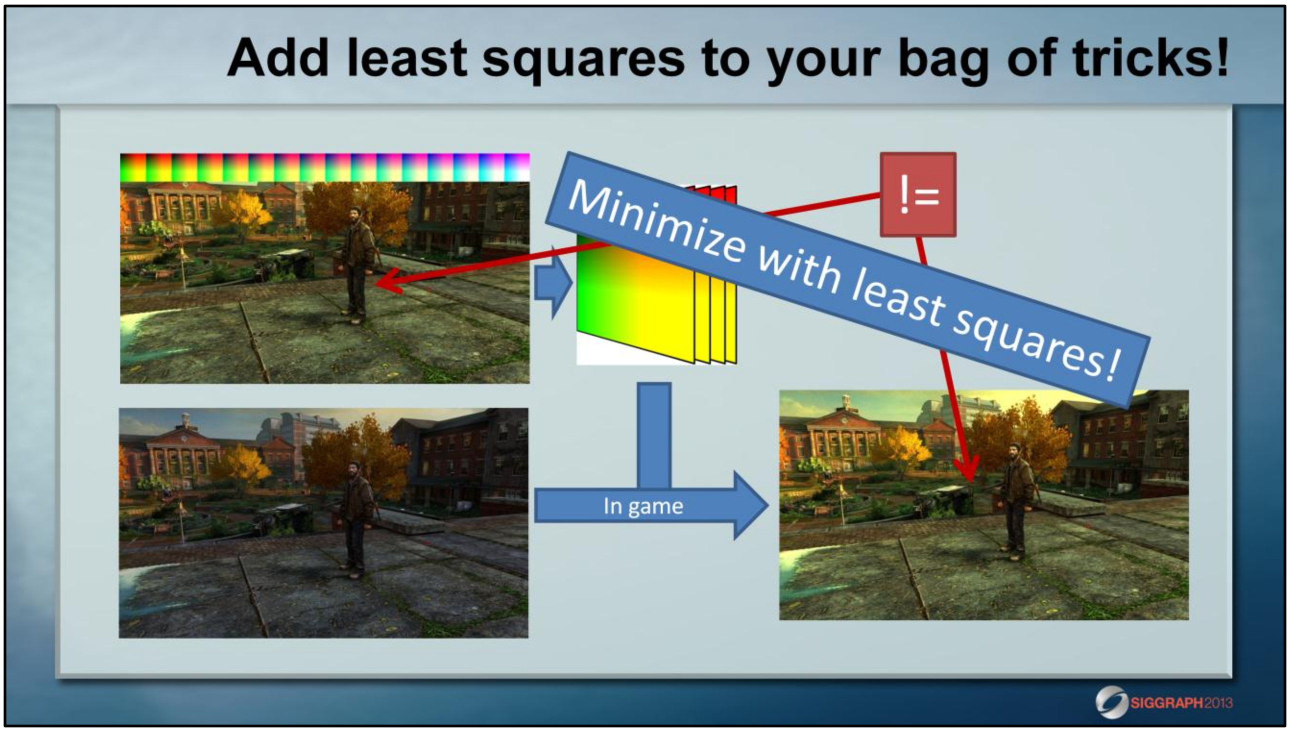


Lightmap seams

- Caveats:
  - DXT tends to break it a bit
    - Live with it (it's usually acceptable)
    - ...or re-stitch after compression, modifying the blocks' anchor points
  - Don't write your own solver: Eigen [Eigen] is really good (thanks Carlos!)

SIGGRAPH2013

There are some caveats however. First of all: block compression, which tends to break the stitching a bit. In most cases it still looks ok, but when it doesn't we have an option to re-stitch the compressed texture, this time modifying blocks' anchor points instead of individual pixels.
The second thing is the linear solver: just don't write your own, it's just a bad idea. There are some pretty good ones available, and if you don't like the hassle of integrating big libraries, there's Eigen which is all template based and comes as a collection of header files.

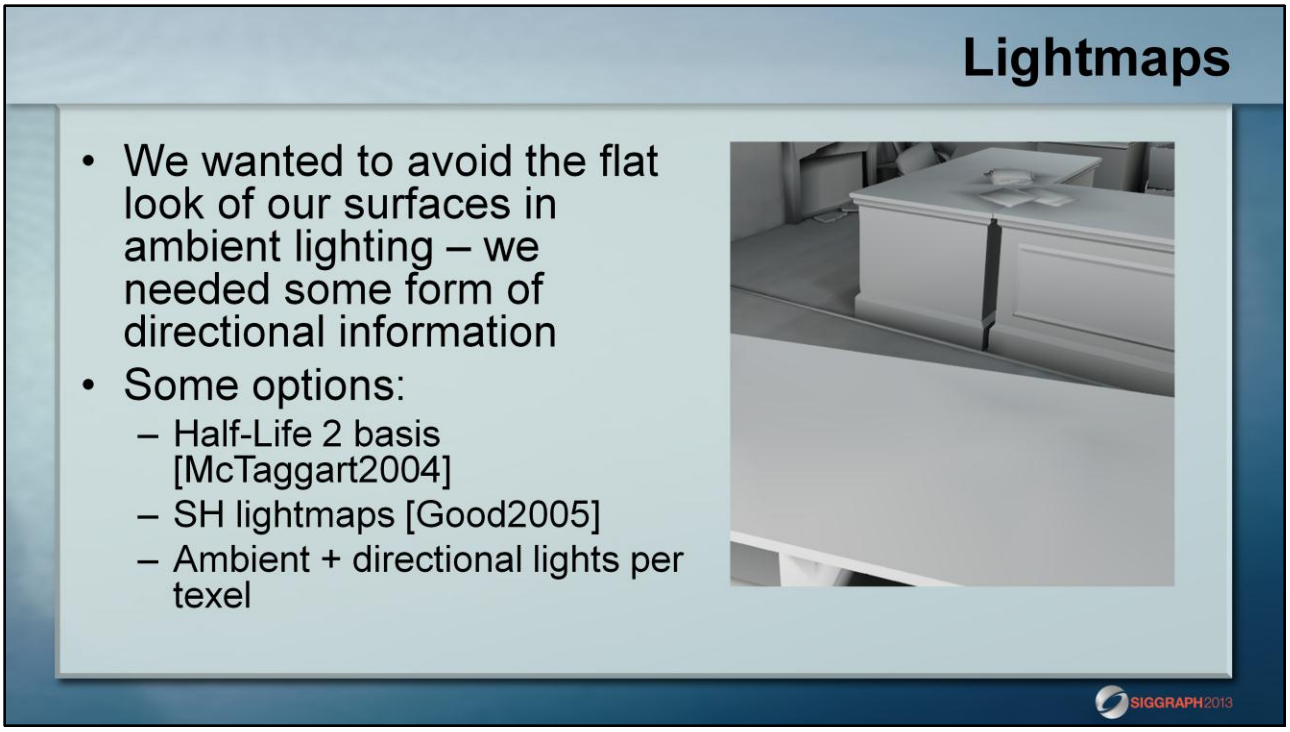


Add least squares to your bag of tricks!

This is one, small off-topic slide, but I just couldn't resist: just add least-squares minimization to your bag of tricks. It's really useful but often overlooked by the game developers community. Once you use it, you suddenly see tons of other applications. On of them is the pipeline for 3D LUTs used for color corrections in games:

The workflow usually looks like this:
- Take screenshot
- Embed identity volume texture in the image
- Massage in Photoshop
- Extract LUT
- Then in game, you render your scene, take the extracted LUT, apply it and you get the same result as in Photoshop

Well, the problem is that it's not the same. Due to quantization of various components, you can get pretty serious discolorations.
What we do: we minimize the difference between the image from Photoshop and the image in game. The error function is defined as a squared difference between those two images, and we modify the volume texture so that it's minimized.
It works much better than just using the straight volume texture.

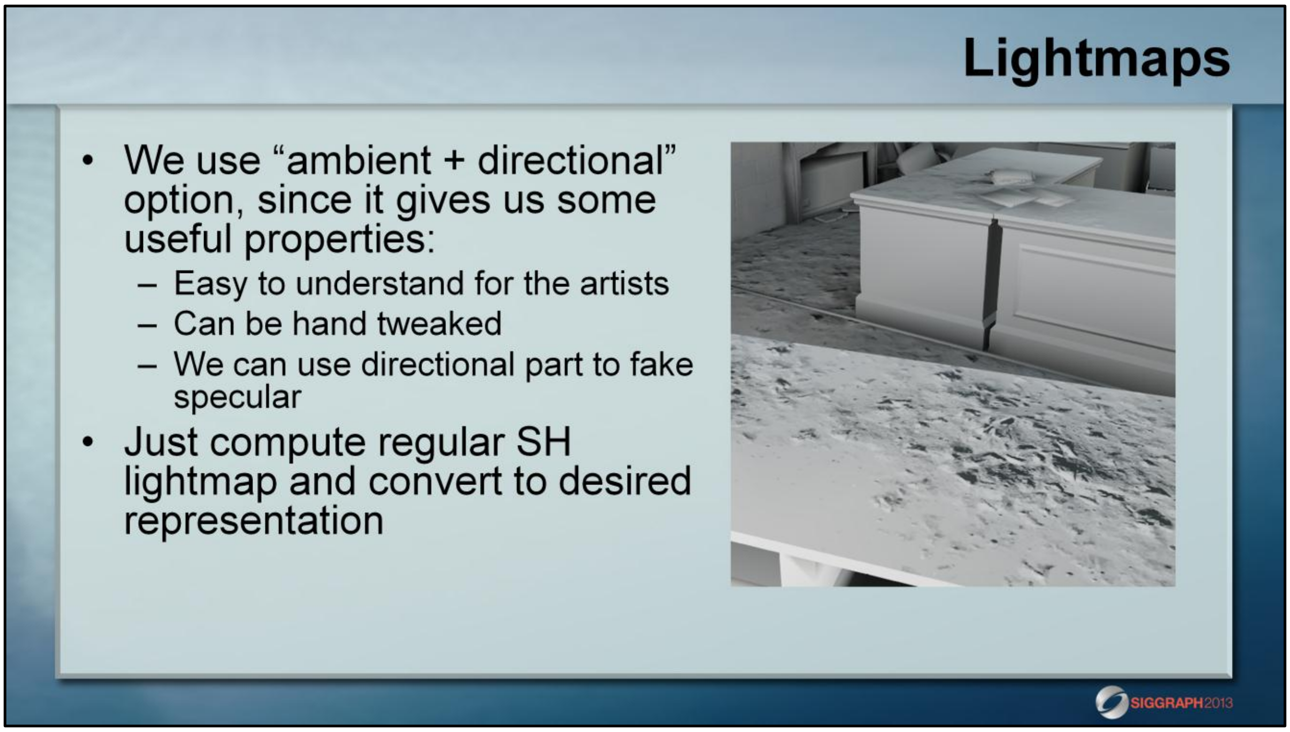


Lightmaps

- We wanted to avoid the flat look of our surfaces in ambient lighting – we needed some form of directional information
- Some options:
  - Half-Life 2 basis [McTaggart2004]
  - SH lightmaps [Good2005]
  - Ambient + directional lights per texel

SIGGRAPH2013

But going back to lighting…
We wanted to avoid the flat look of our surfaces in the absence of direct lighting, so we knew that we needed some sort of directional information in the lightmaps.
We evaluated different options, like the HL2 basis and spherical harmonic lightmaps, but we settled on a very simple representation: a pair of ambient and directional lights stored for every lightmap texel.



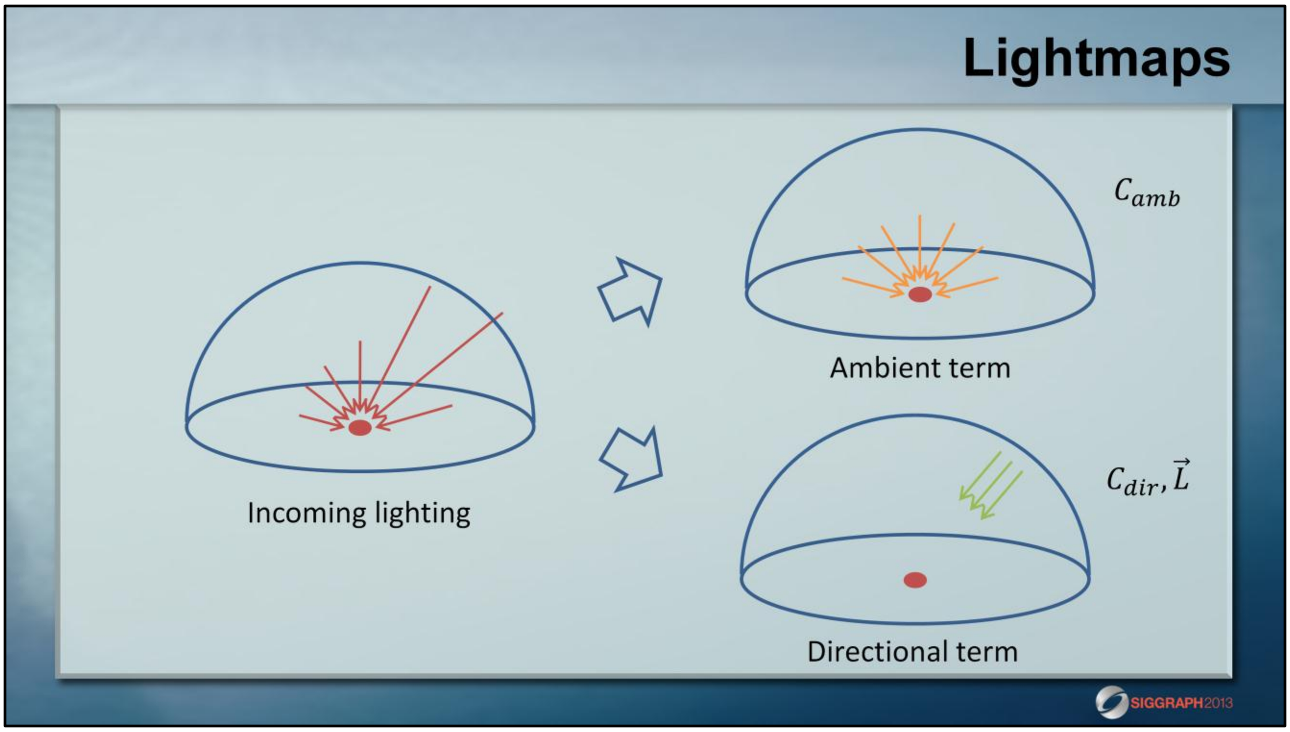
Lightmaps

- We use "ambient + directional" option, since it gives us some useful properties:
  - Easy to understand for the artists
  - Can be hand tweaked
  - We can use directional part to fake specular
- Just compute regular SH lightmap and convert to desired representation

SIGGRAPH2013

Why? It has some really nice properties:
1. It's very easy to understand by the artists, which is extremely useful, since they like investigating the outputs from the bake process
2. It can be hand tweaked to some extent
3. And we have the direction of the dominant light, which we can also use to get some additional effects.

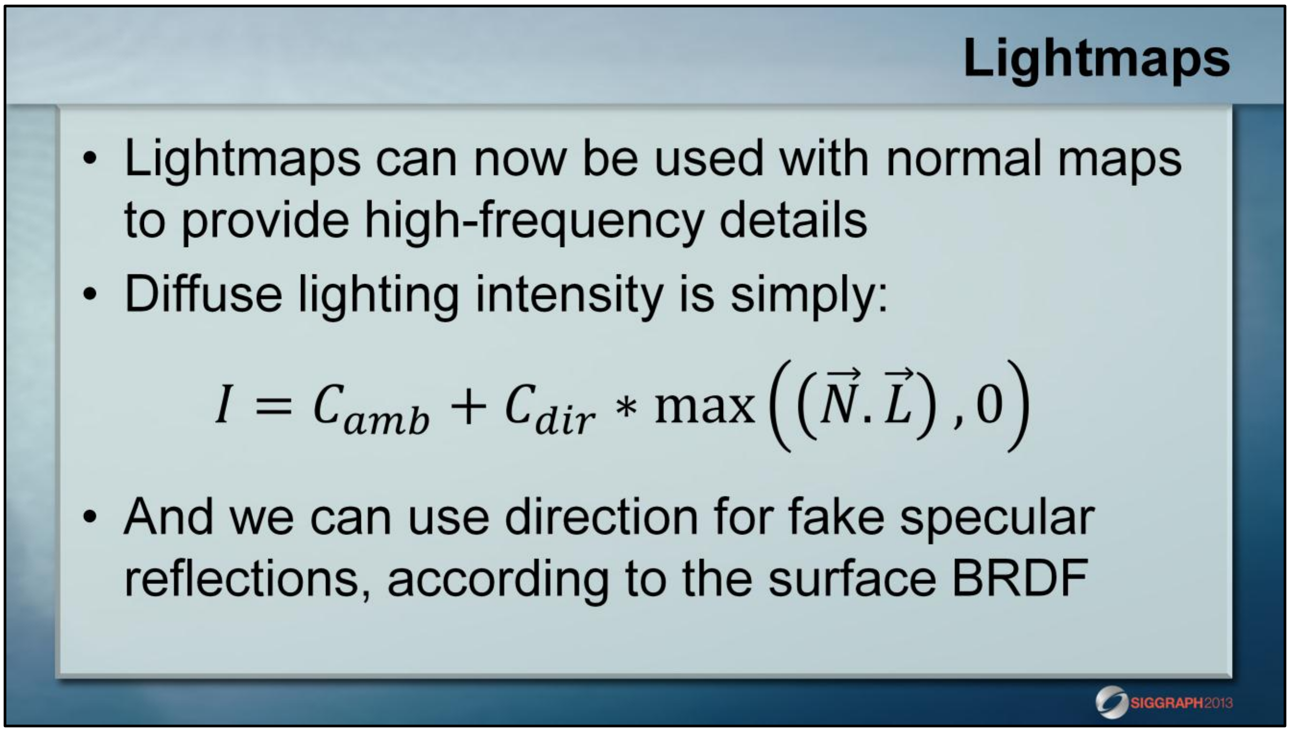
Lightmaps



For each texel of our lightmap, the baking tool generates some distribution of incoming lighting, and we divide is into two components:

- Ambient term: which is just a single color, intensity of the uniform lighting arriving from all directions
- Directional term: which is a direction vector and a color – intensity of lighting arriving from that direction

We generate this representation from an SH lightmap generated by our global-illumination bake tool.
We do it in a way that minimizes the difference in the final lighting produced by those two representations.



Lightmaps

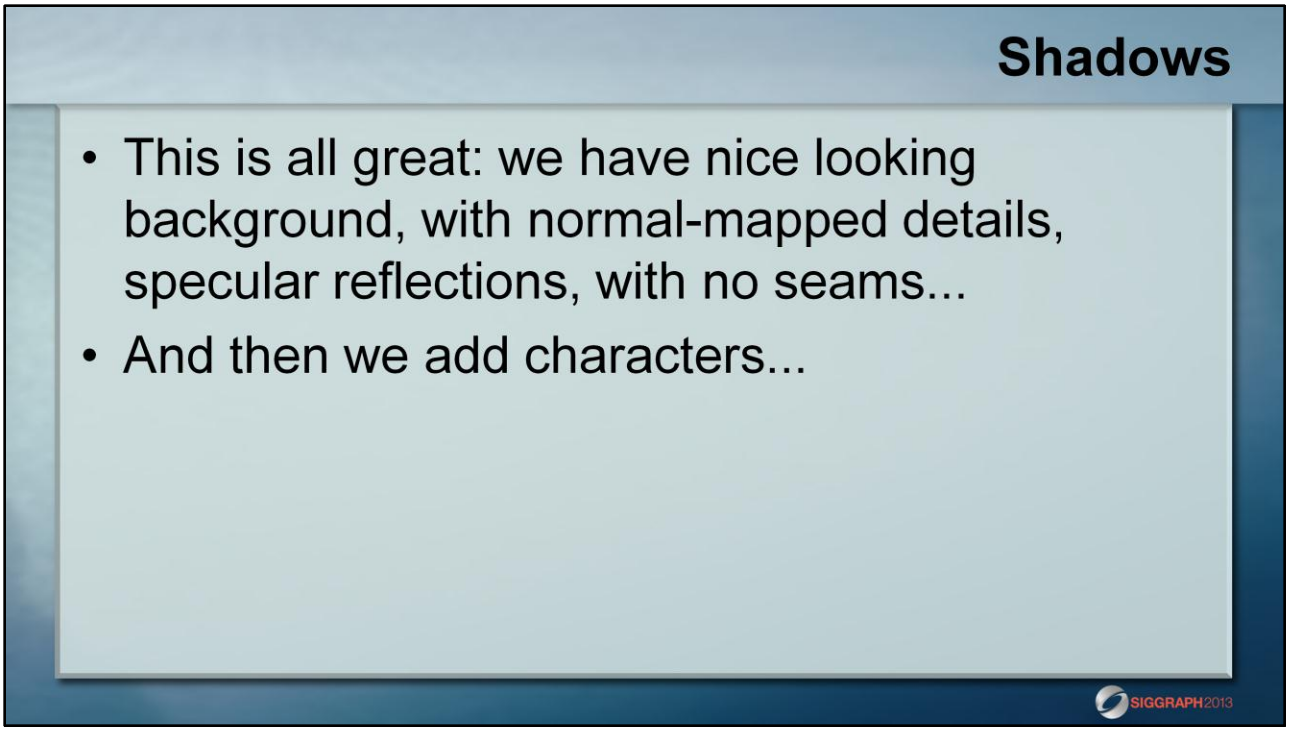
- Lightmaps can now be used with normal maps to provide high-frequency details
- Diffuse lighting intensity is simply:

$$I = C_{amb} + C_{dir} * \max\left(\left(\vec{N}.\vec{L}\right), 0\right)$$

- And we can use direction for fake specular reflections, according to the surface BRDF

SIGGRAPH2013

With this representation, we can use normal maps to provide high frequency details.
The intensity of the diffuse lighting is simply: ambient term + directional color * (n.l)
For glossy surfaces we also use that direction to add a specular highlight.



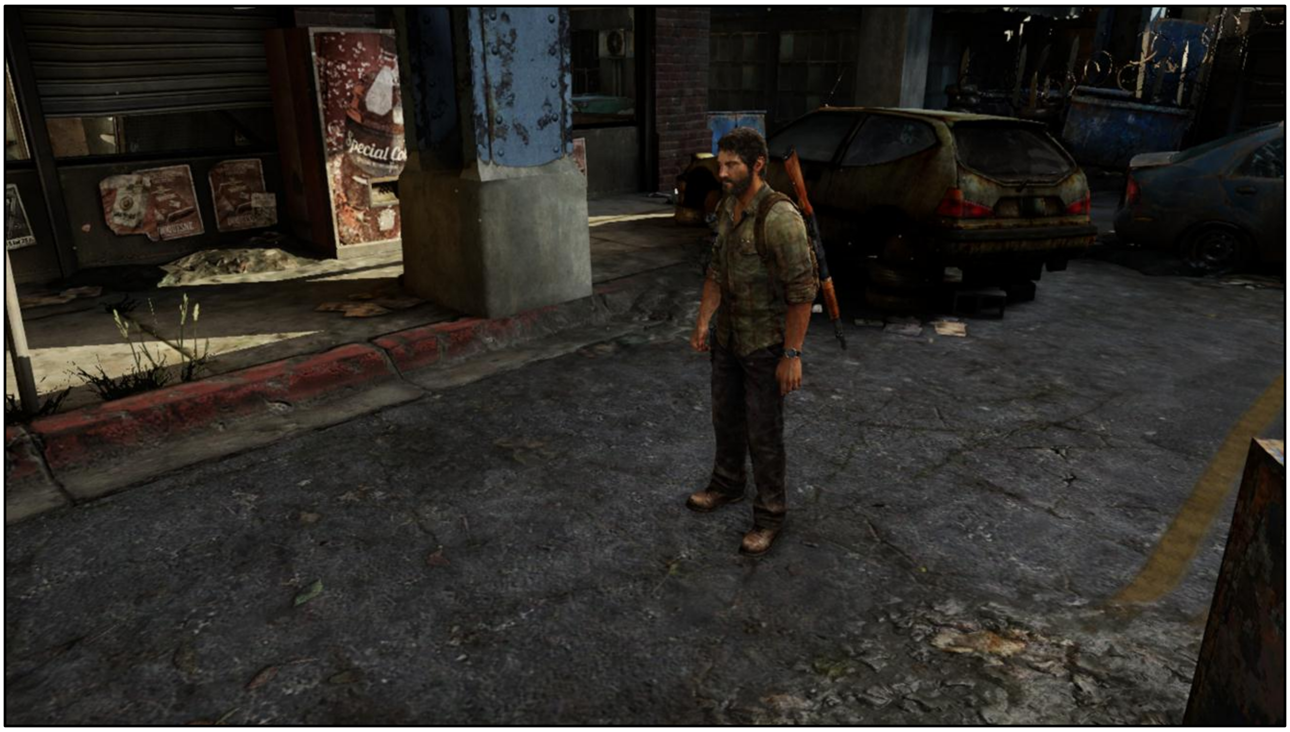
Shadows

- This is all great: we have nice looking background, with normal-mapped details, specular reflections, with no seams...
- And then we add characters...

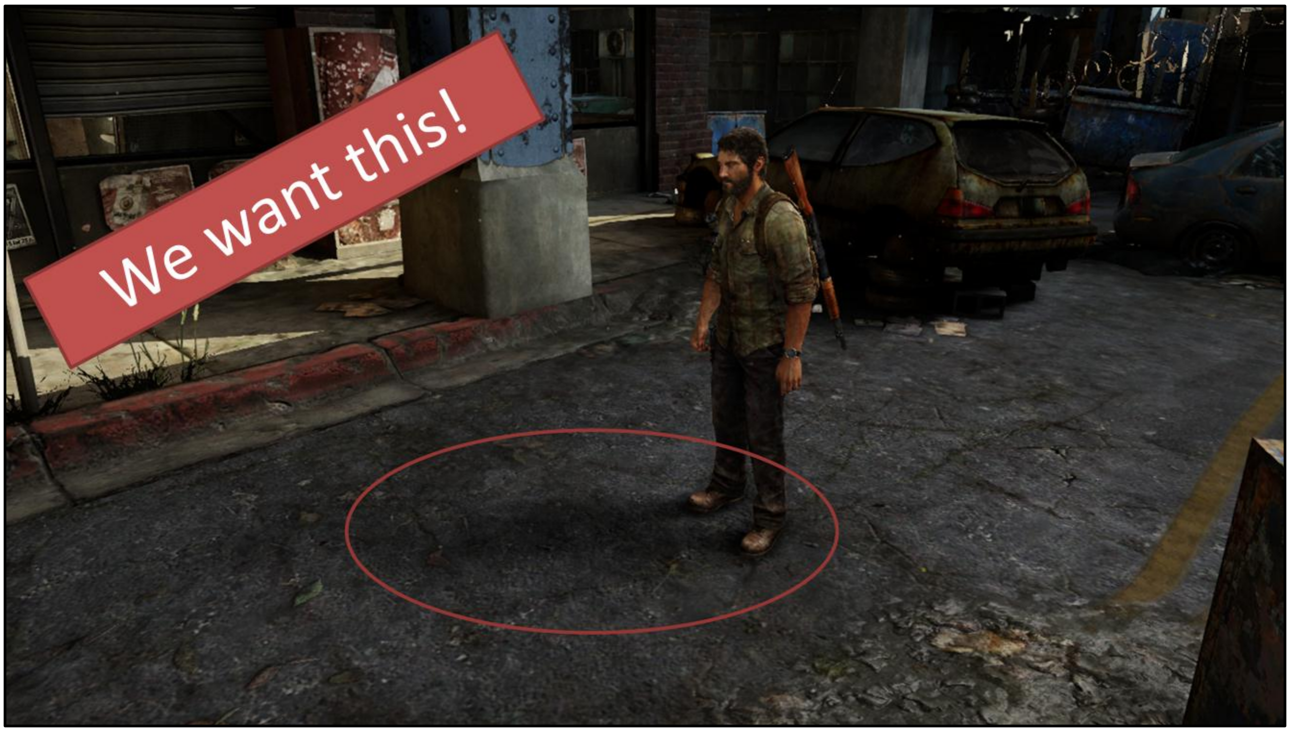
SIGGRAPH2013

And this is all great, but then we add characters…

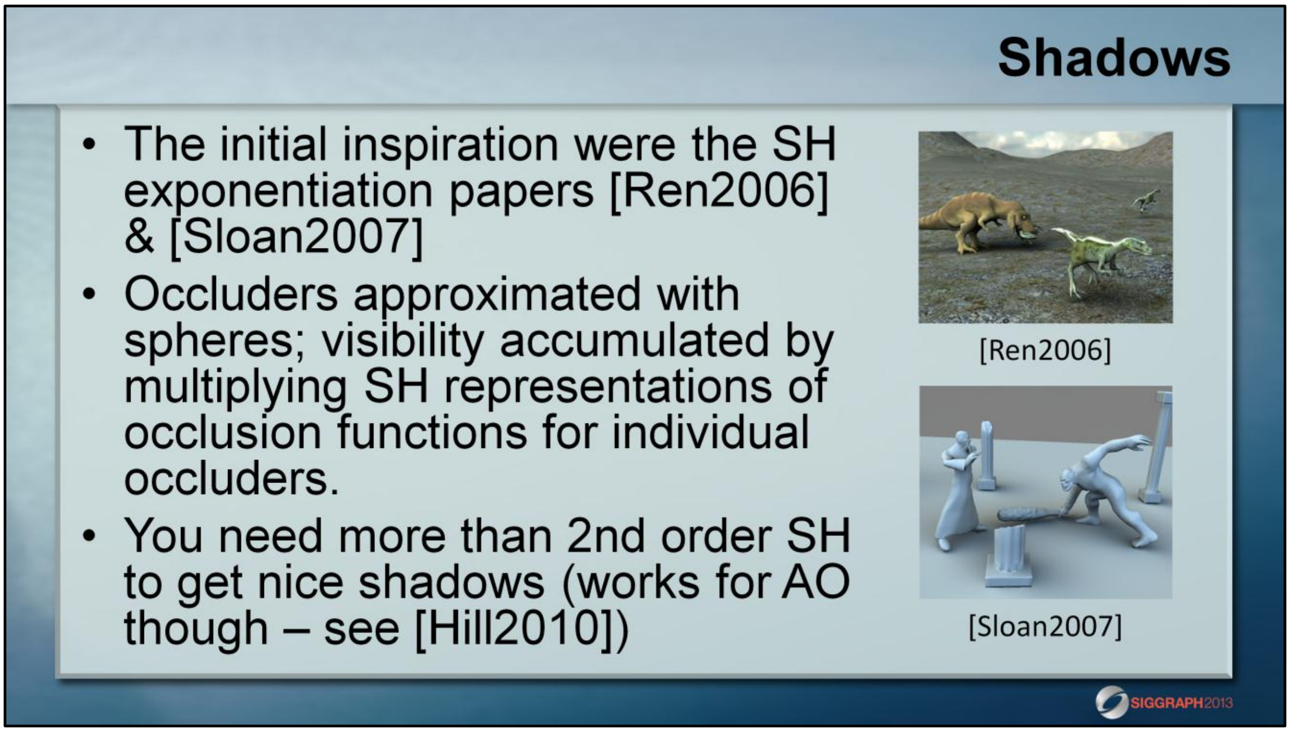
Special Co



EEEE, wrong!
The character looks as if he was a sticker slapped on top of the screen.



We want this! Where character properly interacts with the lighting, is nicely grounded, casts a believable shadow. He actually looks like it's part of the environment, not something added later on.



Shadows

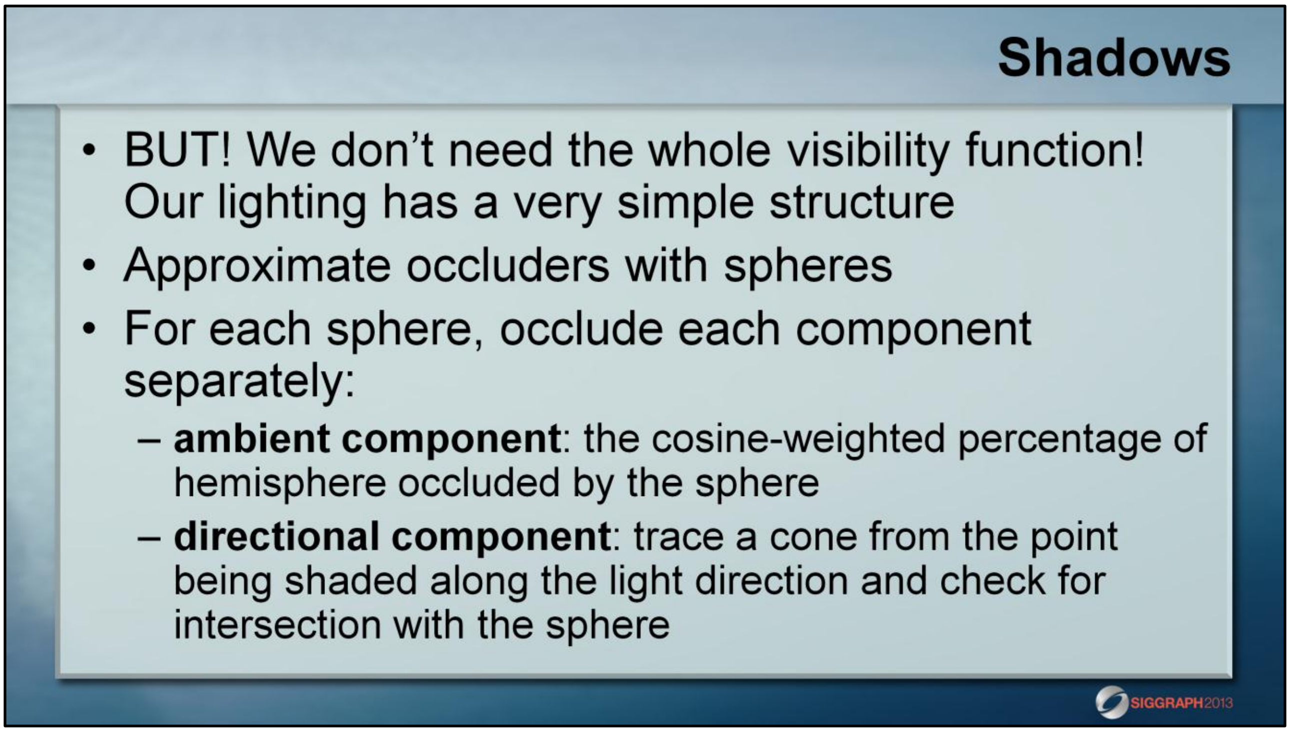
- The initial inspiration were the SH exponentiation papers [Ren2006] & [Sloan2007]
- Occluders approximated with spheres; visibility accumulated by multiplying SH representations of occlusion functions for individual occluders.
- You need more than 2nd order SH to get nice shadows (works for AO though – see [Hill2010])

[Ren2006]

[Sloan2007]

SIGGRAPH2013

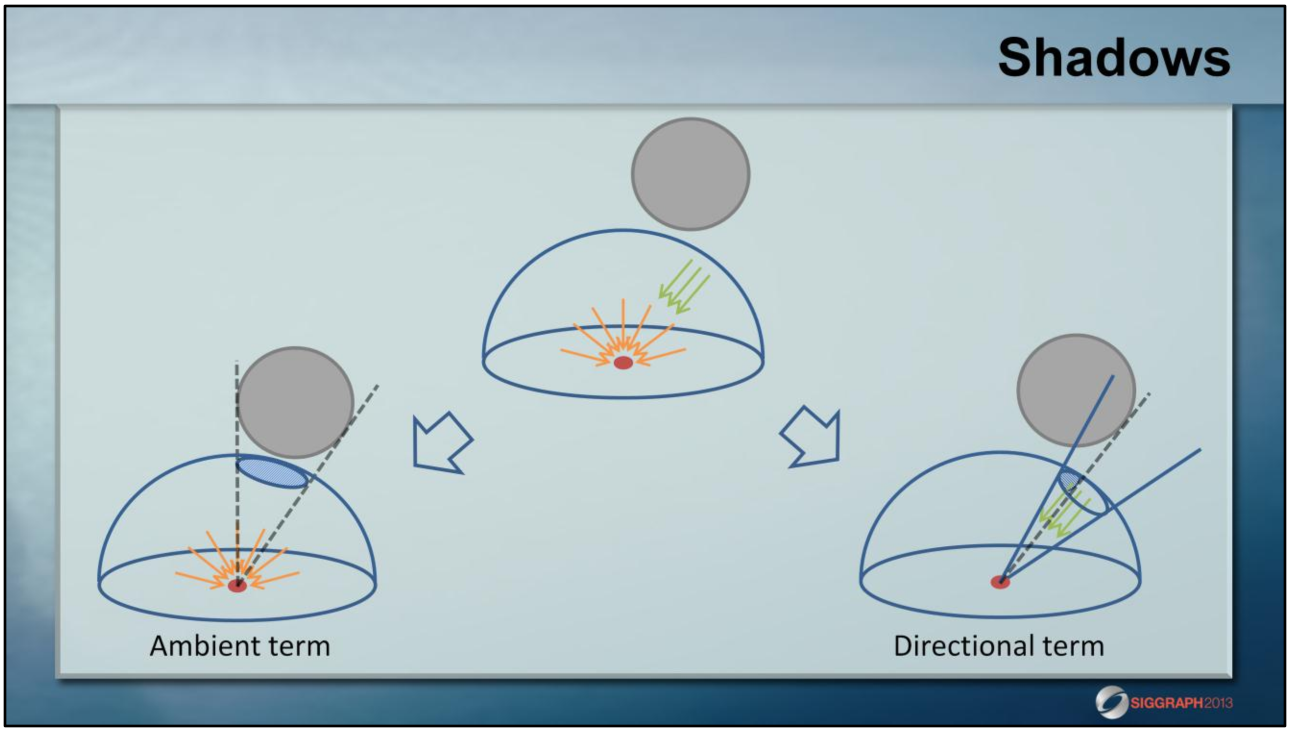
The origianal inspiration came from SH exponentiation papers. They approximate occluders with sets of spheres and combine visibility functions described as SH vectors, by multiplying them together in a fancy way.
The problem is that to get long shadows, you need a high SH order. 2nd order SH works fine for AO – Stephen Hill did something similar in Splinter Cell Conviction – but for shadows it’s just not enough.



Shadows

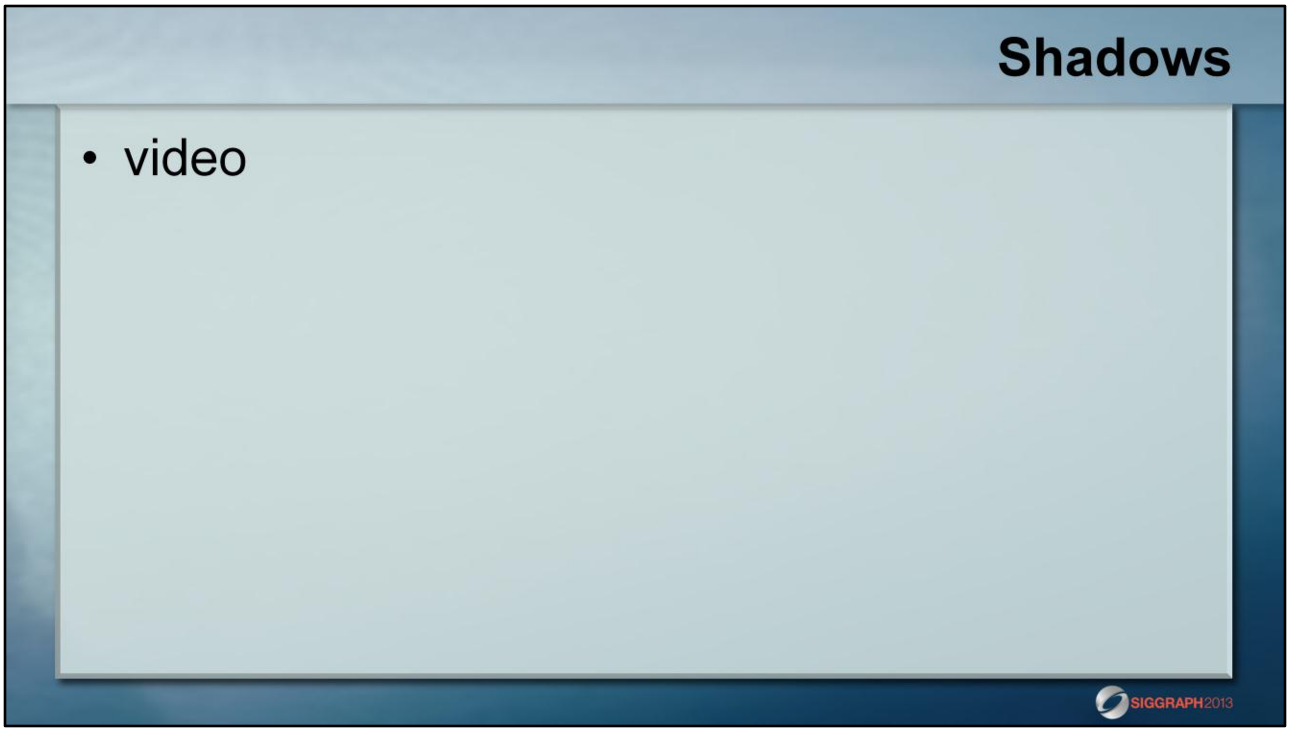
- BUT! We don’t need the whole visibility function! Our lighting has a very simple structure
- Approximate occluders with spheres
- For each sphere, occlude each component separately:
  - **ambient component**: the cosine-weighted percentage of hemisphere occluded by the sphere
  - **directional component**: trace a cone from the point being shaded along the light direction and check for intersection with the sphere

BUT! Our representation of lighting comes to the rescue. It has a very simple structure: it’s just an ambient component and a directional component. We don’t need to worry about the whole visibility function, we just need to occlude those two parts appropriately.
Just like the papers mentioned, we approximate the occluders with spheres. Then, for every point we shade, for each of those occluding spheres:
- For the ambient component, we compute the occlusion of the as a cosine-weighted percentage of hemisphere occluded by the sphere
- For the directional component we trace a cone from a point being shaded, in the direction of the dominant lighting and check for the intersections with the occluder sphere



Shadows

Here’s the breakdown of how it works:
We have some lighting at a point in space and a occluder and how the occlusion values are calculated for both components

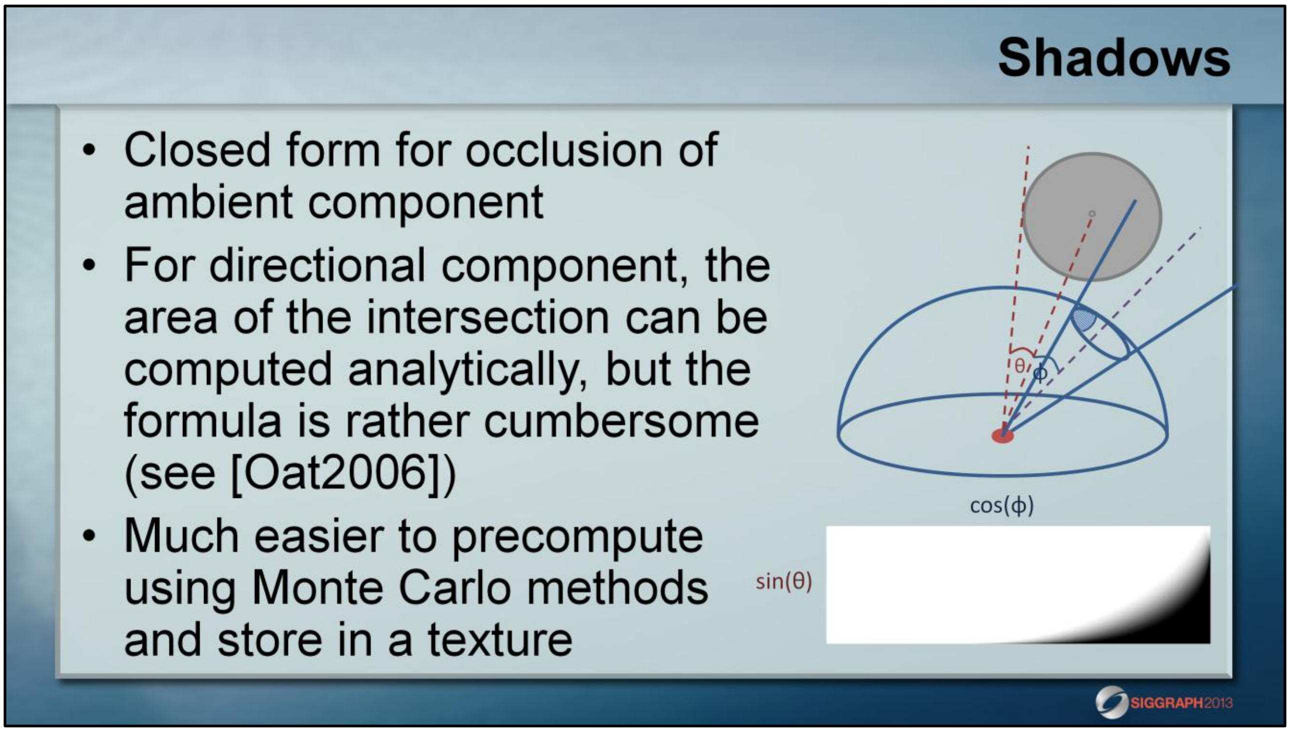


Shadows

- video

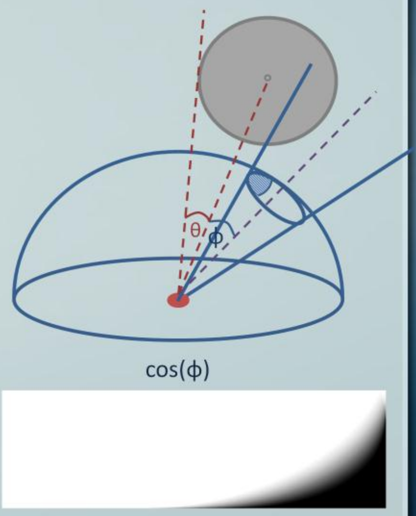
SIGGRAPH2013

This is how it works in practice – first without, then with the occlusion calculations.



Shadows

- Closed form for occlusion of ambient component
- For directional component, the area of the intersection can be computed analytically, but the formula is rather cumbersome (see [Oat2006])
- Much easier to precompute using Monte Carlo methods and store in a texture

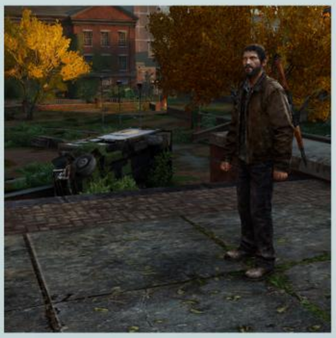


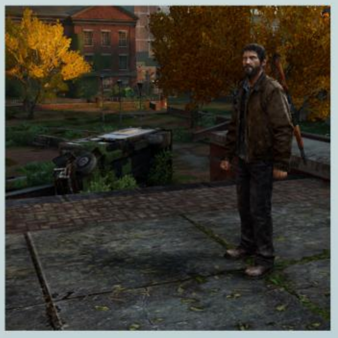
SIGGRAPH2013

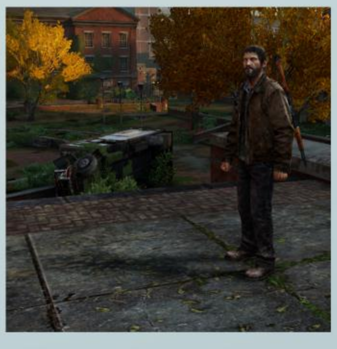
Now some technical details:
For the ambient component, there's a closed form solution for the occlusion value: it's just the cosine lobe integrated over the solid angle subtended by the sphere divided the cosine lobe integrated over the entire hemisphere.
There is a formula to calculate the occlusion of the directional component as well, but it's rather cumbersome. We've found that it's much easier to just precompute it using Monte Carlo methods and store it in a texture. For a given cone angle, the occlusion value is a function of the angle subtended by the occluder (theta) and the angle between the cone axis and the vector pointing towards the occluder center (phi). We can store this function in a 2D texture, and for different cone angles we can stack several of them in a 3D texture.

Shadows

- Cone angle is exposed as a parameter to the artists – it controls the penumbra size

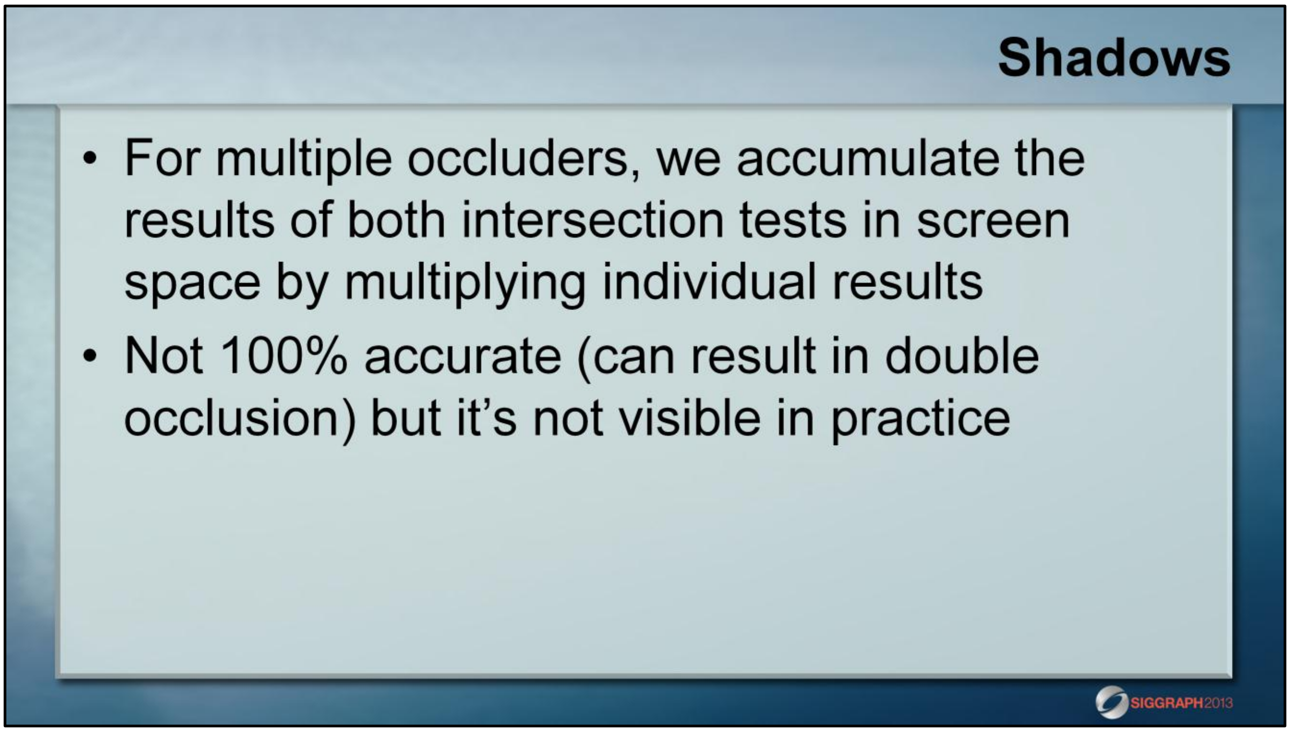

60 deg.


30 deg.


10 deg.

SIGGRAPH2013

The angle of the cone we use for tracing is chosen arbitrarily. It controls the penumbra size, and we exposed it to our artists. They can change it on per-level basis, but there's nothing to prevent using different angles for every pixel on the screen.

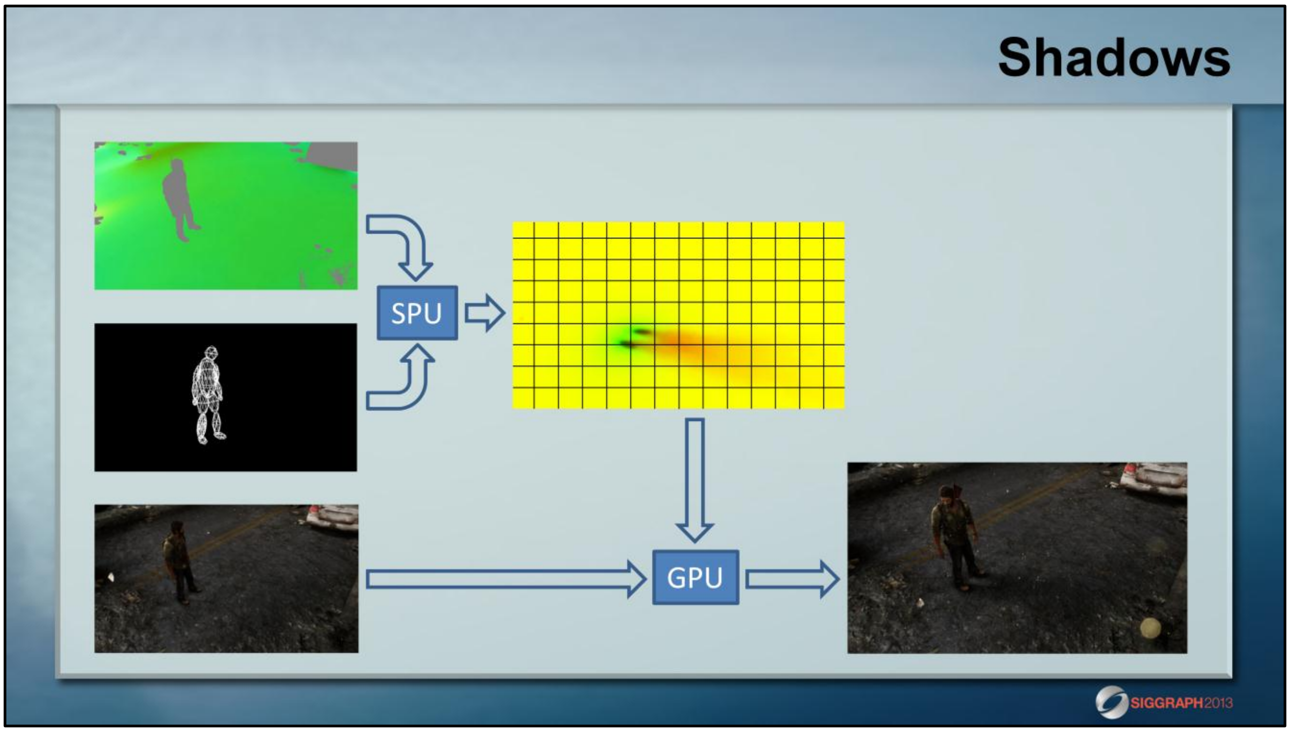


Shadows

- For multiple occluders, we accumulate the results of both intersection tests in screen space by multiplying individual results
- Not 100% accurate (can result in double occlusion) but it’s not visible in practice

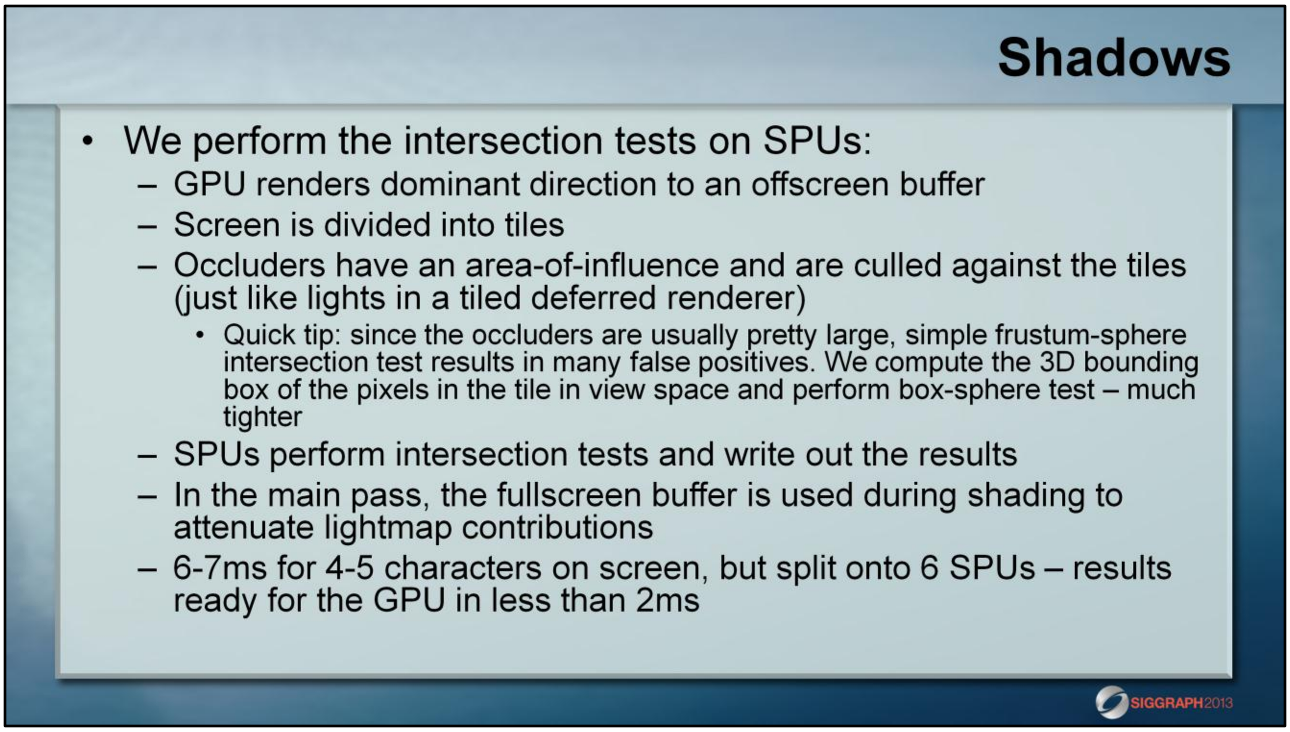
SIGGRAPH2013

We accumulate the results of both intersection tests for individual spheres, by multiplying results. This is not 100% accurate and in theory can cause double occlusion, but it’s not visible in practice (I don’t see it even though I know what to look for).



Here are some more low-level detail details:

We do all the intersection tests on the SPU. We start by rendering the dominant direction into a offscreen buffer. It is passed to the SPUs together with the descriptions of the occluders.
SPUs divide the image into tiles and check which occluders influence each tile. Each occluder has a “sphere of influence” – so it casts shadows only inside that sphere – and we check if those spheres of influence intersect our tiles. It’s pretty similar to how you do culling of lights for tiled deferred lighting. One problem is that the tiles are pretty small compared to the size of the spheres, and if you cull using typical frustum-sphere intersection test you get a lot of false positives. We eliminate them by calculating the view space bounding box of the pixels within a tile and testing this box against the sphere of influence, which is a much tighter test.
Once we have the list of occluders influencing the tile, we compute the occlusion for ambient and directional components, accumulate the results for individual spheres and write out the results to a texture.
This texture is used by the GPU in the main rendering pass to attenuate the lighting fetched from the lightmap.

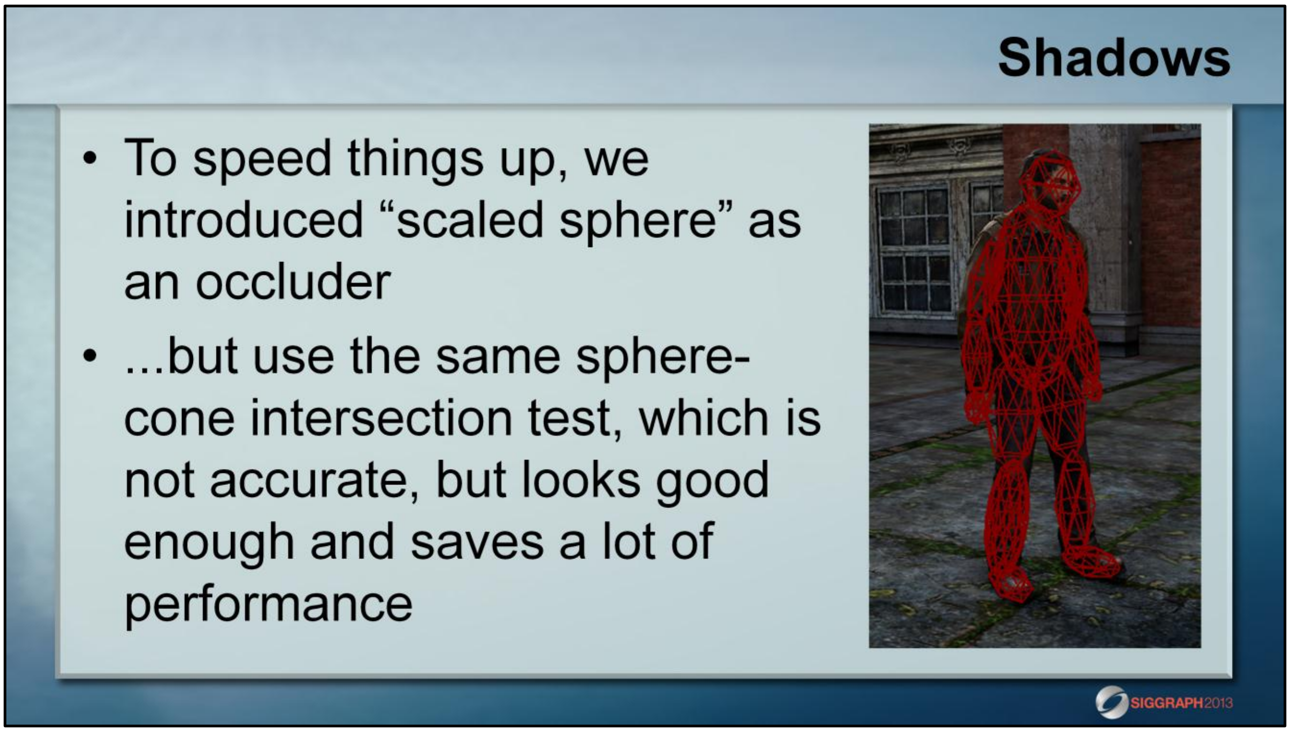


Shadows

- We perform the intersection tests on SPUs:
  - GPU renders dominant direction to an offscreen buffer
  - Screen is divided into tiles
  - Occluders have an area-of-influence and are culled against the tiles (just like lights in a tiled deferred renderer)
    - Quick tip: since the occluders are usually pretty large, simple frustum-sphere intersection test results in many false positives. We compute the 3D bounding box of the pixels in the tile in view space and perform box-sphere test – much tighter
  - SPUs perform intersection tests and write out the results
  - In the main pass, the fullscreen buffer is used during shading to attenuate lightmap contributions
  - 6-7ms for 4-5 characters on screen, but split onto 6 SPUs – results ready for the GPU in less than 2ms

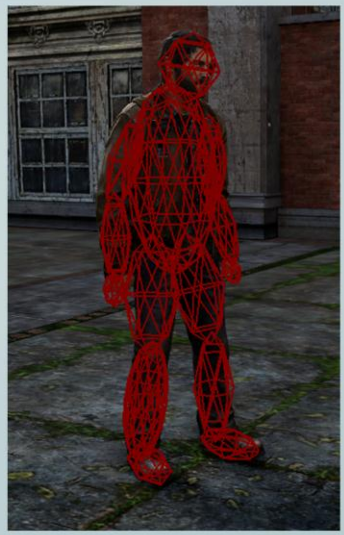
SIGGRAPH2013

For 4-5 characters on screen it takes around 6-7ms, but it's split onto 6 SPUs, so the results are ready in less than 2ms, and the GPU is doing some other stuff in the meantime anyway, so it doesn't wait for the SPUs.



Shadows

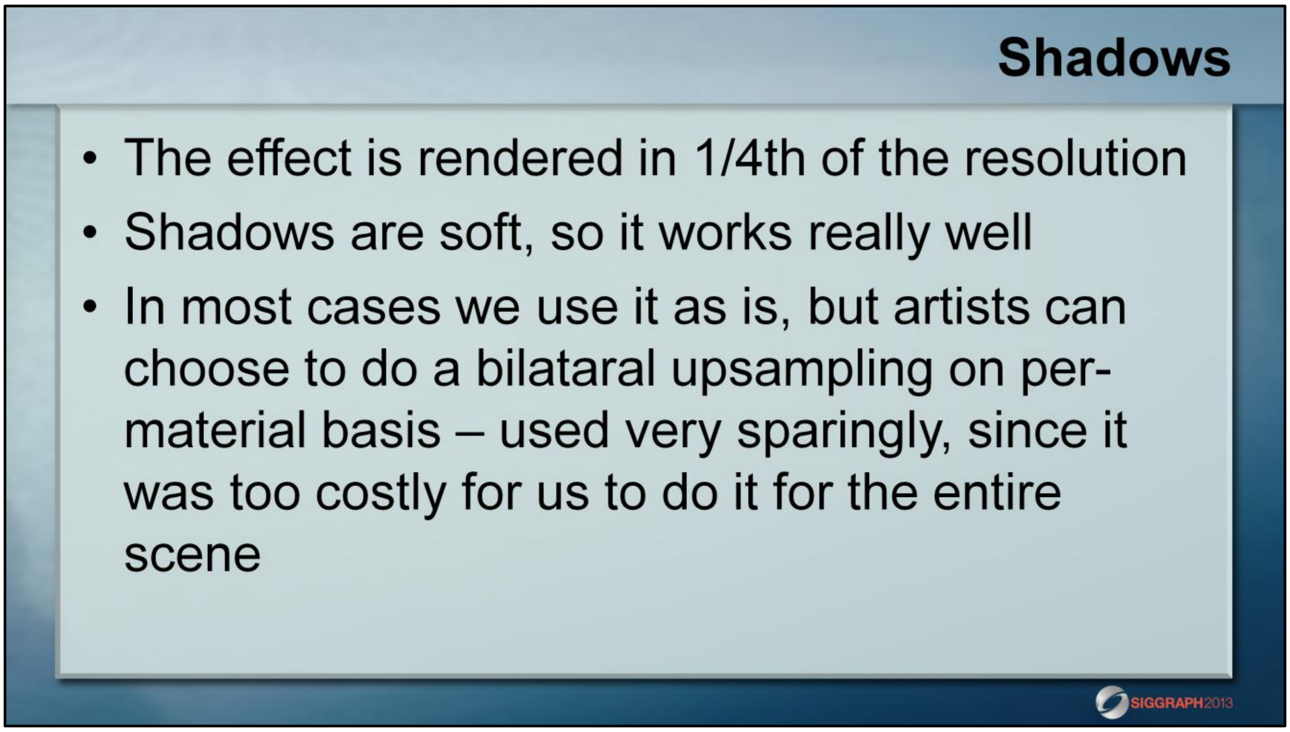
- To speed things up, we introduced “scaled sphere” as an occluder
- ...but use the same sphere-cone intersection test, which is not accurate, but looks good enough and saves a lot of performance



SIGGRAPH2013

A few more tricks to get better performance:

We introduced the ellipsoid as an occluder too – it’s just a sphere scaled along one axis. When we test for the intersections, we transform the cone to the space of the ellipsoid and scale it down along the main axis, so the ellipsoid becomes a sphere. The direction of the cone is scaled as well, but we don’t change the cone angle. We should squash it along the scaling axis, but it would cause it to become non-radially symmetric, which would make the intersection test more complicated. We leave it as is and perform the intersection as before.
This is not accurate, but looks good and saves us a lot of performance, since we can greatly reduce the number of spheres we use.

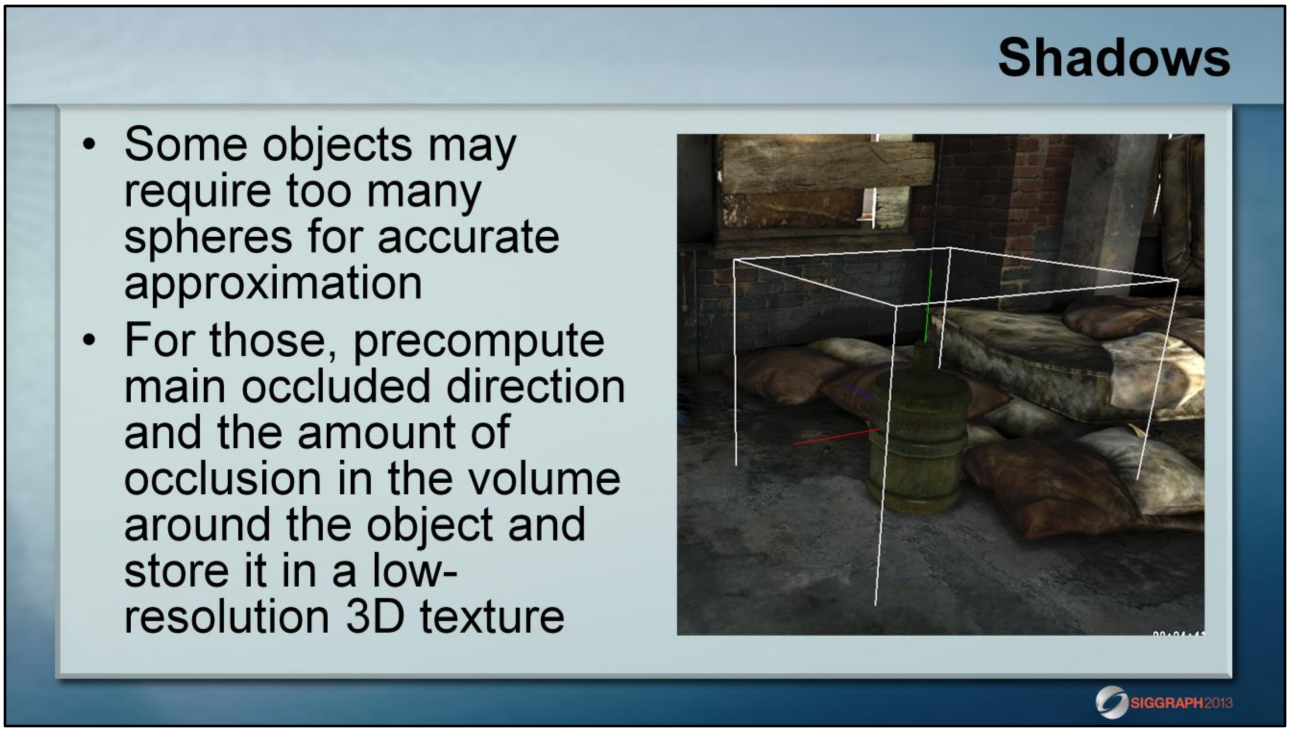


Shadows

- The effect is rendered in 1/4th of the resolution
- Shadows are soft, so it works really well
- In most cases we use it as is, but artists can choose to do a bilataral upsampling on per-material basis – used very sparingly, since it was too costly for us to do it for the entire scene

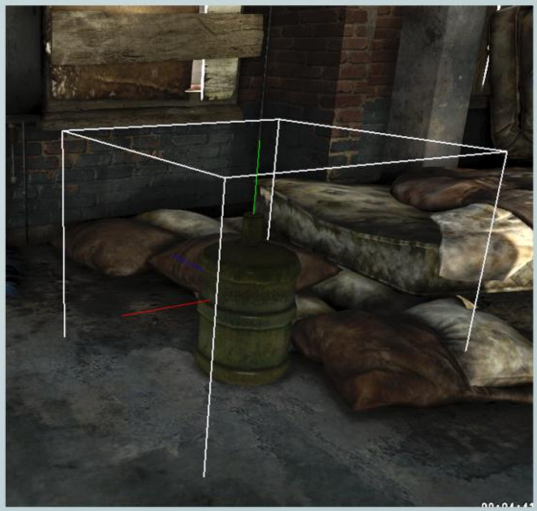
SIGGRAPH2013

The whole effect is rendered in ¼ resolution, but due to the low-frequency nature of those shadows, it works really well.
Most objects use the result texture as is, just upscaling. We added bilateral upsample as an option for some of the objects, but it's used very sparingly, since it's a bit too costly for us.



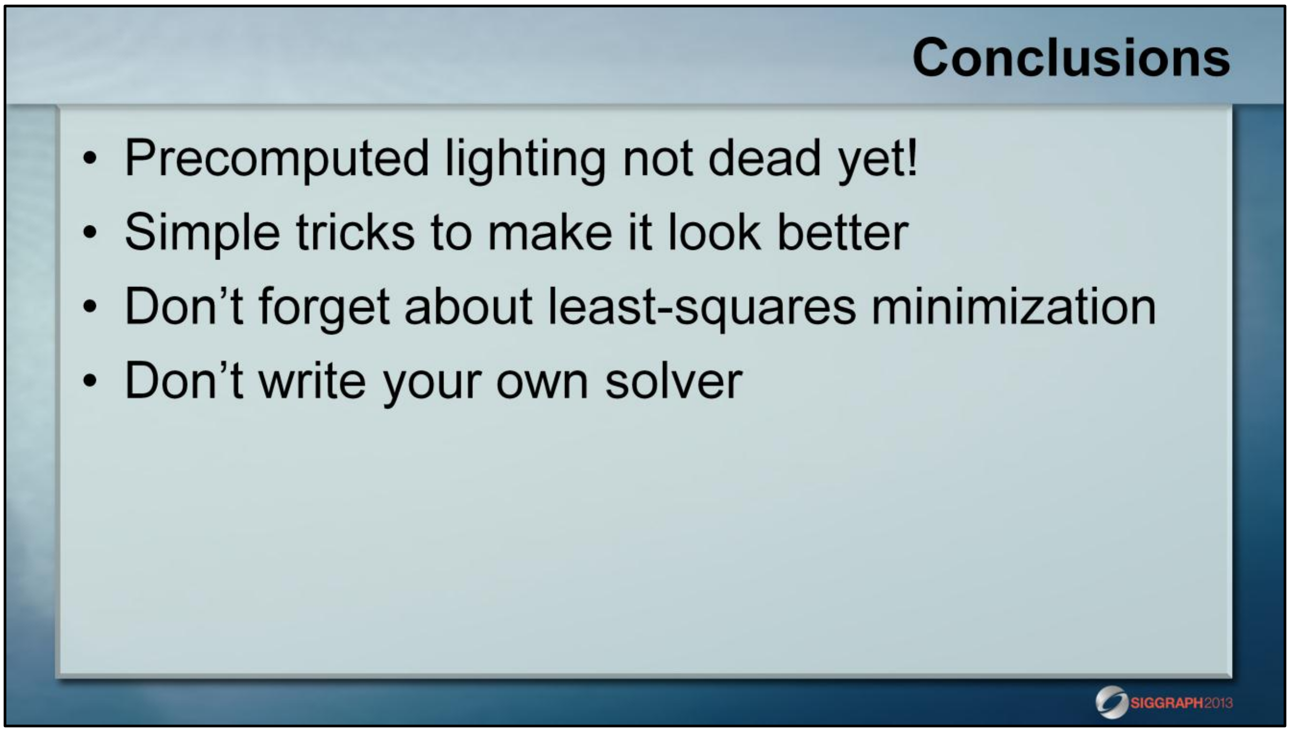
Shadows

- Some objects may require too many spheres for accurate approximation
- For those, precompute main occluded direction and the amount of occlusion in the volume around the object and store it in a low-resolution 3D texture



SIGGRAPH2013

For the objects that would require too many spheres to approximate, we also support precomputing occlusion information and storing it in a 3D texture. We store an average, unoccluded direction and the spread of the occluded direction. With this setup, the intersection test becomes a cone-cone intersection (which in fact is the same as cone-sphere, so we use the same lookup texture for this).The occlusion from those objects is composited on top of the regular ones.

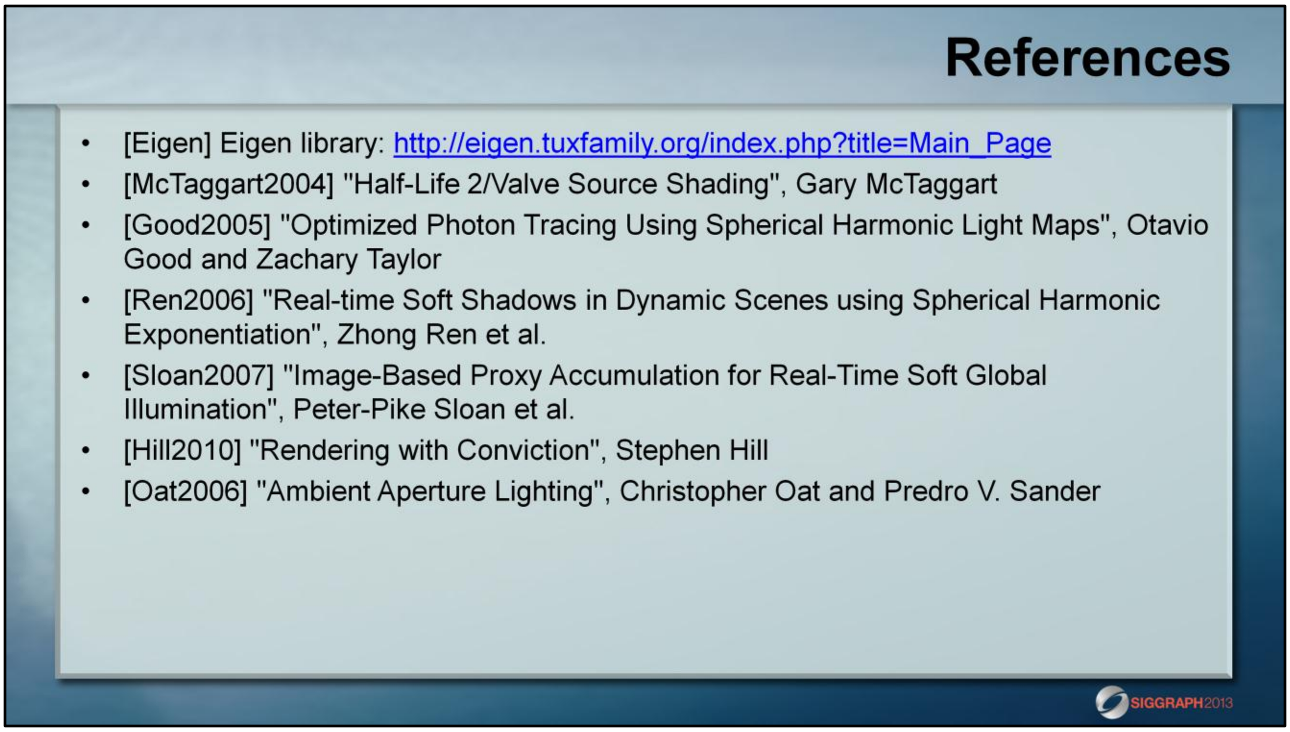


Conclusions

- Precomputed lighting not dead yet!
- Simple tricks to make it look better
- Don't forget about least-squares minimization
- Don't write your own solver

SIGGRAPH2013

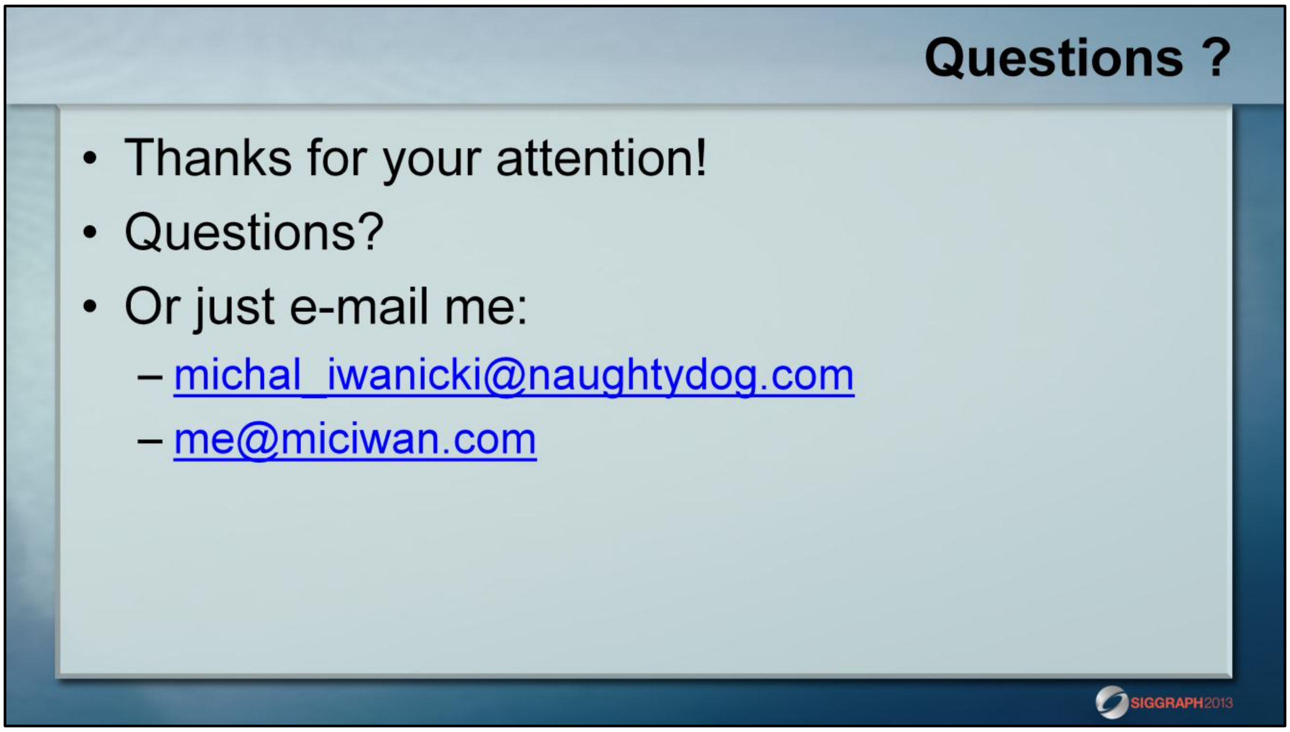
To conclude: precomputed lighting is definitely not dead yet. With some relatively simple tricks you can make it look much better.
Don't forget about least-squares minimization, and don't write your own linear solver for it!


References

- [Eigen] Eigen library: http://eigen.tuxfamily.org/index.php?title=Main_Page
- [McTaggart2004] "Half-Life 2/Valve Source Shading", Gary McTaggart
- [Good2005] "Optimized Photon Tracing Using Spherical Harmonic Light Maps", Otavio Good and Zachary Taylor
- [Ren2006] "Real-time Soft Shadows in Dynamic Scenes using Spherical Harmonic Exponentiation", Zhong Ren et al.
- [Sloan2007] "Image-Based Proxy Accumulation for Real-Time Soft Global Illumination", Peter-Pike Sloan et al.
- [Hill2010] "Rendering with Conviction", Stephen Hill
- [Oat2006] "Ambient Aperture Lighting", Christopher Oat and Predro V. Sander

SIGGRAPH2013

Here are some references

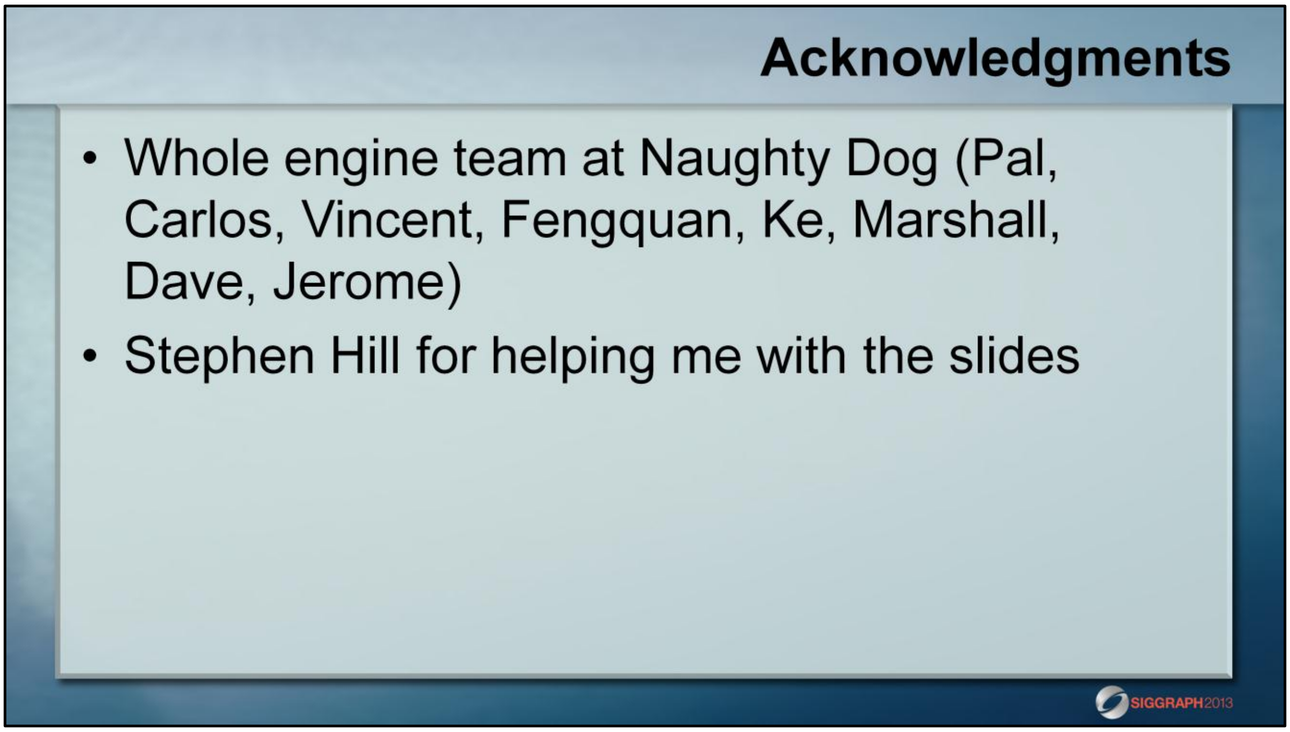


Questions ?

- Thanks for your attention!
- Questions?
- Or just e-mail me:
  - michal_iwanicki@naughtydog.com
  - me@miciwan.com

SIGGRAPH2013

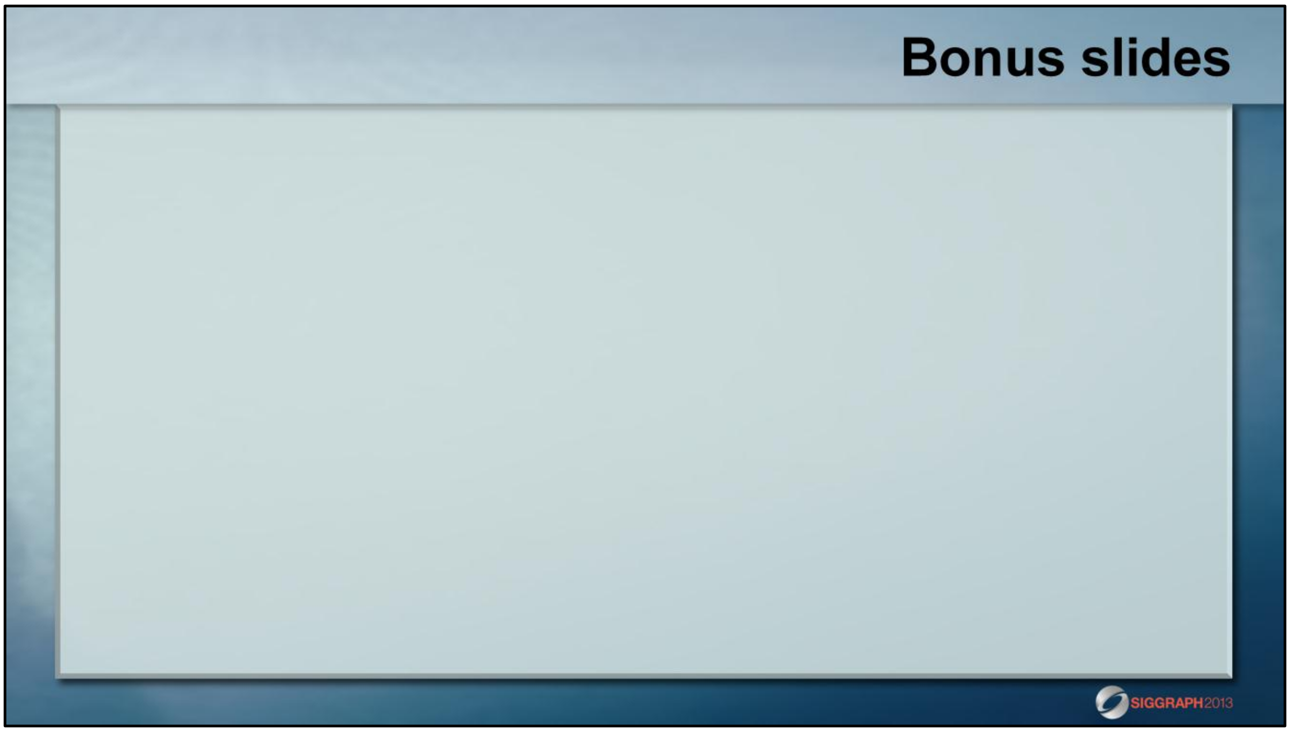
Thanks for your attention. If you have any questions now I'll be more than happy to answer them!



Acknowledgments

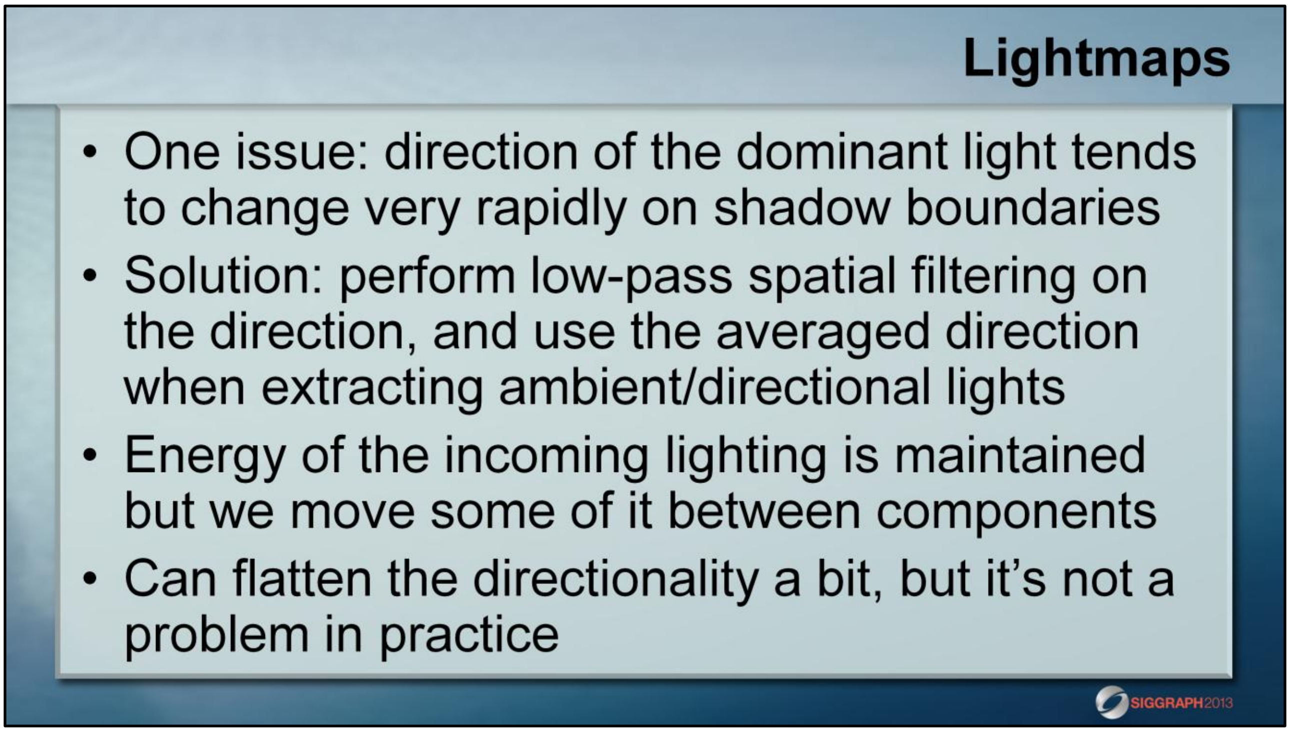
- Whole engine team at Naughty Dog (Pal, Carlos, Vincent, Fengquan, Ke, Marshall, Dave, Jerome)
- Stephen Hill for helping me with the slides

SIGGRAPH2013



Bonus slides

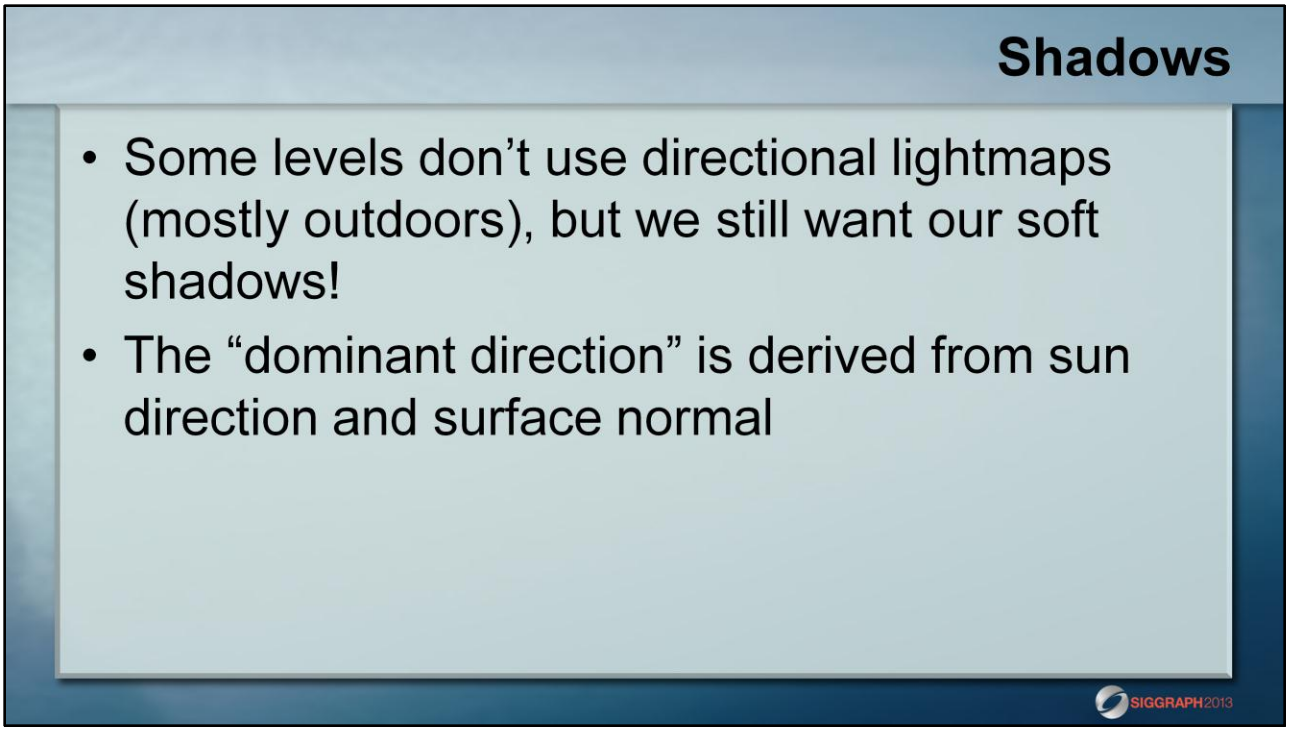
SIGGRAPH2013



Lightmaps

- One issue: direction of the dominant light tends to change very rapidly on shadow boundaries
- Solution: perform low-pass spatial filtering on the direction, and use the averaged direction when extracting ambient/directional lights
- Energy of the incoming lighting is maintained but we move some of it between components
- Can flatten the directionality a bit, but it's not a problem in practice

SIGGRAPH2013



Shadows

- Some levels don't use directional lightmaps (mostly outdoors), but we still want our soft shadows!
- The "dominant direction" is derived from sun direction and surface normal