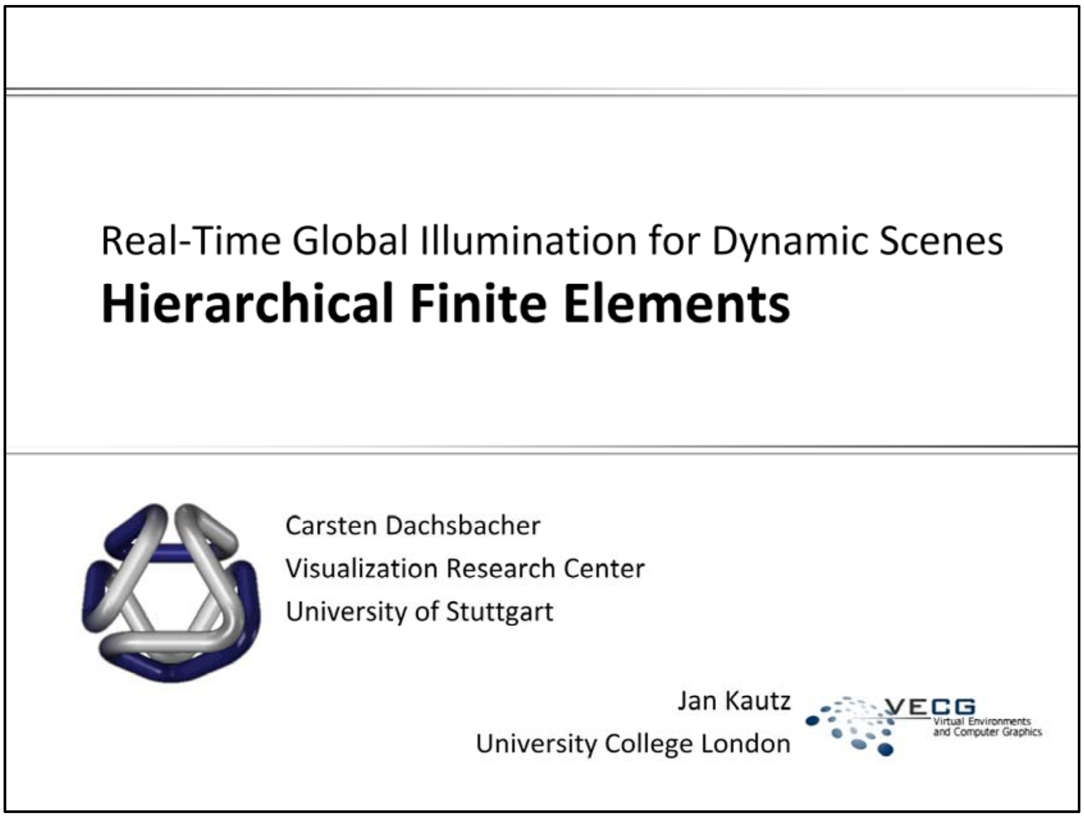
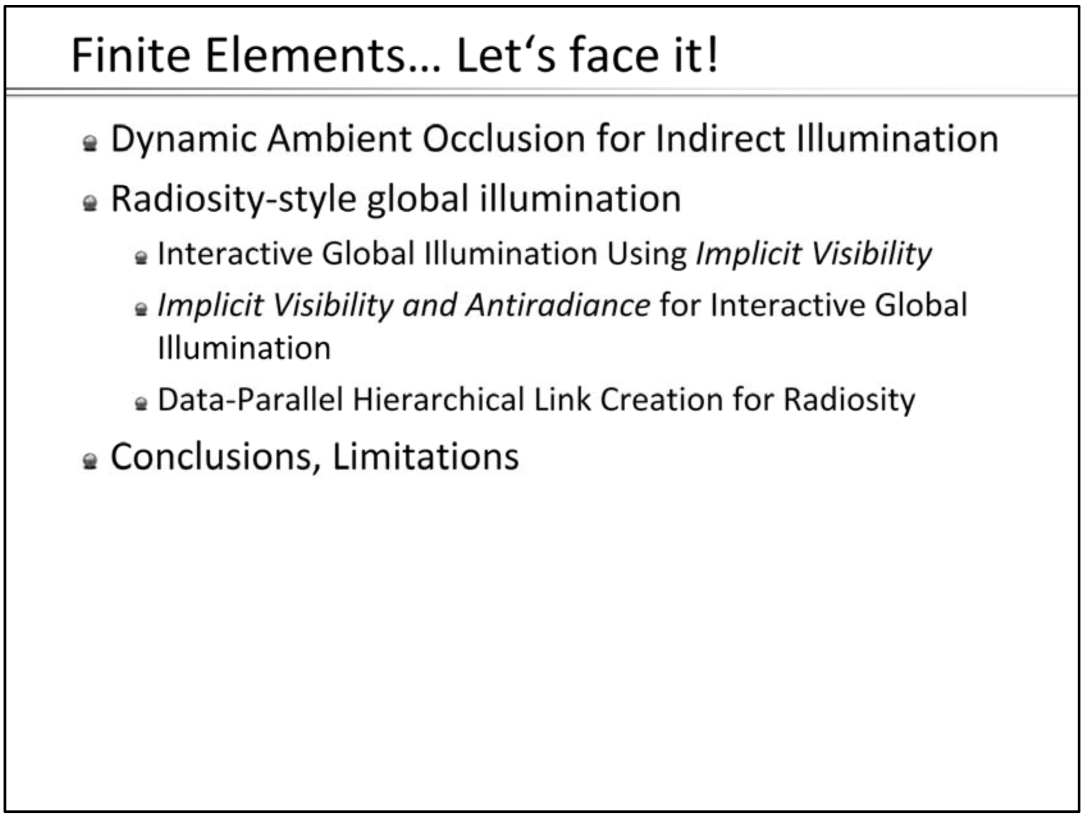


Real-Time Global Illumination for Dynamic Scenes

Hierarchical Finite Elements

Carsten Dachsbacher
Visualization Research Center
University of Stuttgart

Jan Kautz
University College London



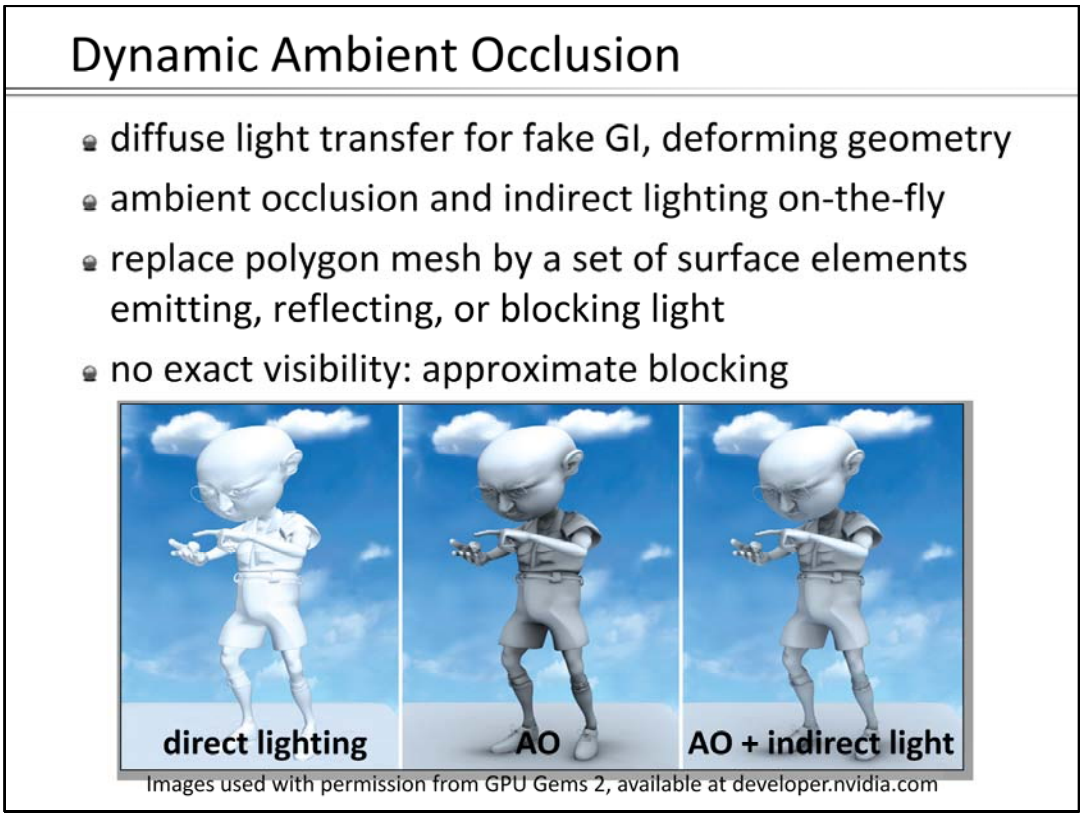
Finite Elements… Let‘s face it!

- Dynamic Ambient Occlusion for Indirect Illumination
- Radiosity-style global illumination
  - Interactive Global Illumination Using *Implicit Visibility*
  - *Implicit Visibility and Antiradiance* for Interactive Global Illumination
  - Data-Parallel Hierarchical Link Creation for Radiosity
- Conclusions, Limitations

Welcome to the 3rd block of our tutorial which will cover finite element methods for global illumination at real-time, or at least interactive, frame rates.

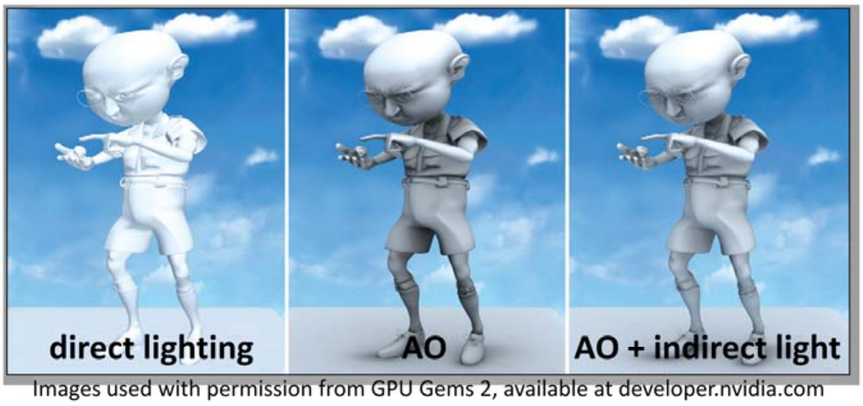
We will address two categories here, first Bunnells early approach to AO and indirect illumination based on finite elements on the GPU.

And second, variants of the radiosity algorithm that have been modified to become GPU-friendlier and thus closer to real-time.



Dynamic Ambient Occlusion

- diffuse light transfer for fake GI, deforming geometry
- ambient occlusion and indirect lighting on-the-fly
- replace polygon mesh by a set of surface elements emitting, reflecting, or blocking light
- no exact visibility: approximate blocking



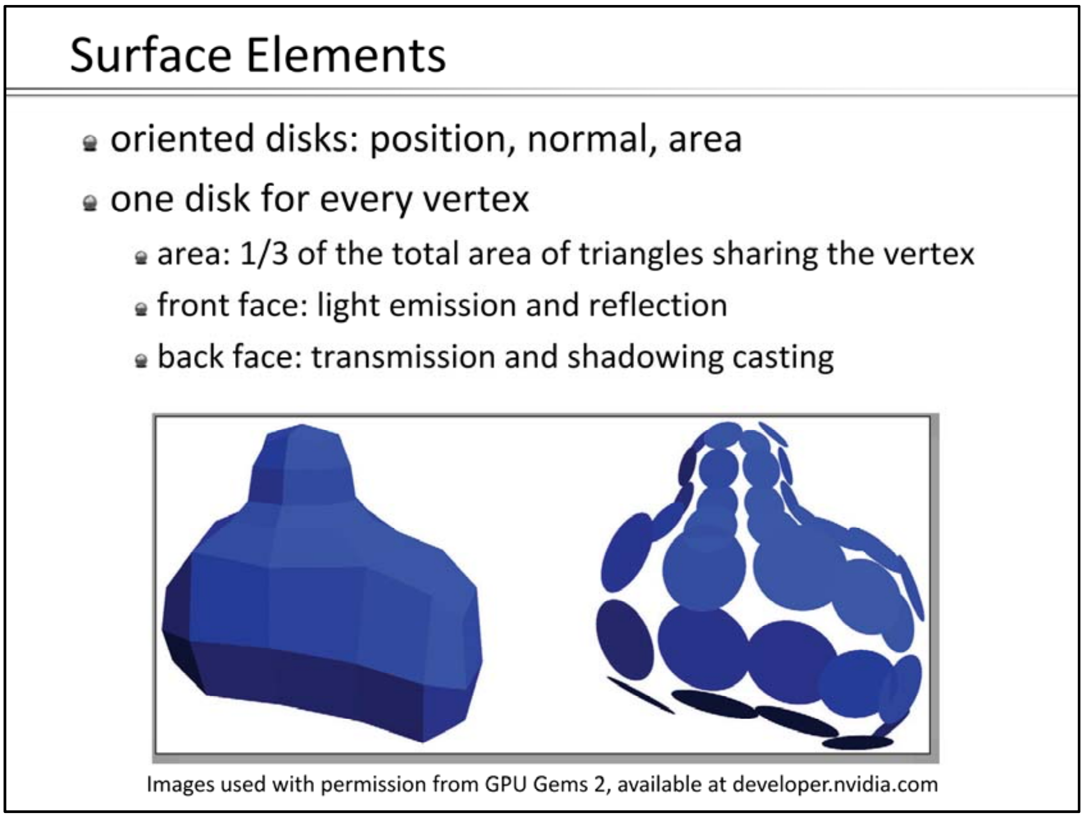
Images used with permission from GPU Gems 2, available at developer.nvidia.com

Bunnell’s technique approximates diffuse light transfer for fake global illumination effects and is able to handle deforming geometry.

Using a GPU is’s fast enough to compute ambient occlusion (center) and indirect lighting (right) on the fly.

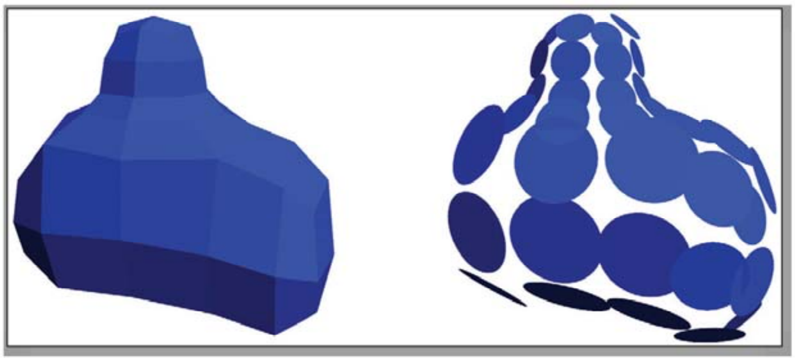
At the core of this technique, the lighting computation is carried out using a set of surface elements, that emit, transmit, or reflect light and that can shadow each other.

The trick which makes this method fast is that the visibility between those elements for computing ambient occlusion and indirect lighting is not computed exactly, but approximated iteratively.



Surface Elements

- oriented disks: position, normal, area
- one disk for every vertex
  - area: 1/3 of the total area of triangles sharing the vertex
  - front face: light emission and reflection
  - back face: transmission and shadowing casting



Images used with permission from GPU Gems 2, available at developer.nvidia.com

The first step is to convert the meshes to surface elements.

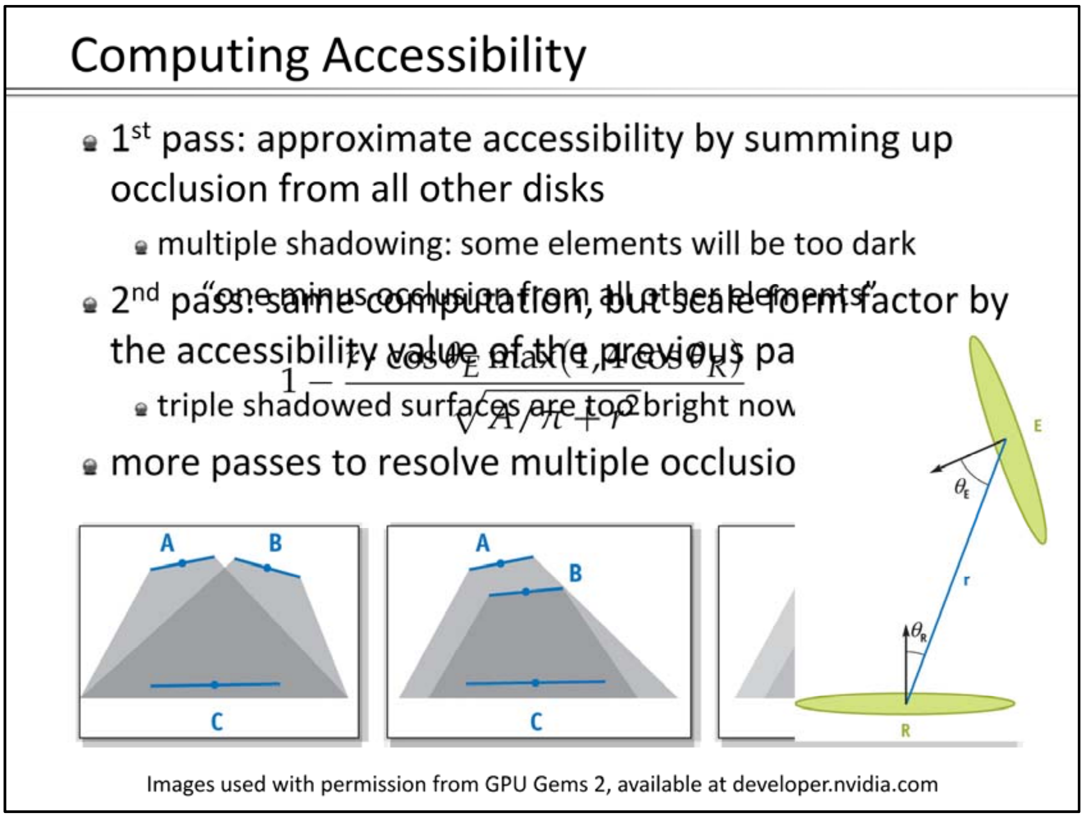
For this, one disc element is created at every vertex of the mesh:

The position and the orientation is directly obtained from the vertex data.

The size of the disc is chosen such that its area equals the sum of one third of the area of all triangles sharing the vertex.

… or one-fourth of the area of adjacent quads.

When computing light transport, light is emitted and reflected on the front sides of the discs and light is transmitted and shadows are cast from the back.



Computing Accessibility

- 1st pass: approximate accessibility by summing up occlusion from all other disks
  - multiple shadowing: some elements will be too dark
- 2nd pass: same computation, but scale form factor by the accessibility value of the previous pa
  - triple shadowed surfaces are too bright now
- more passes to resolve multiple occlusio

“one minus occlusion from all other elements”

$$1 - \frac{r \cos\theta_E \max(1, 4\cos\theta_R)}{\sqrt{A/\pi + r^2}}$$

Images used with permission from GPU Gems 2, available at developer.nvidia.com

Let’s first see how the accessibility of a disc is computed – this takes place in multiple passes.

In the first pass, the accessibility of each element is approximated by summing up the occlusion from all other elements using a disc-to-disc form factor, and subtracting the sum from 1.

I.e. in this example, A and B are shadowing C.

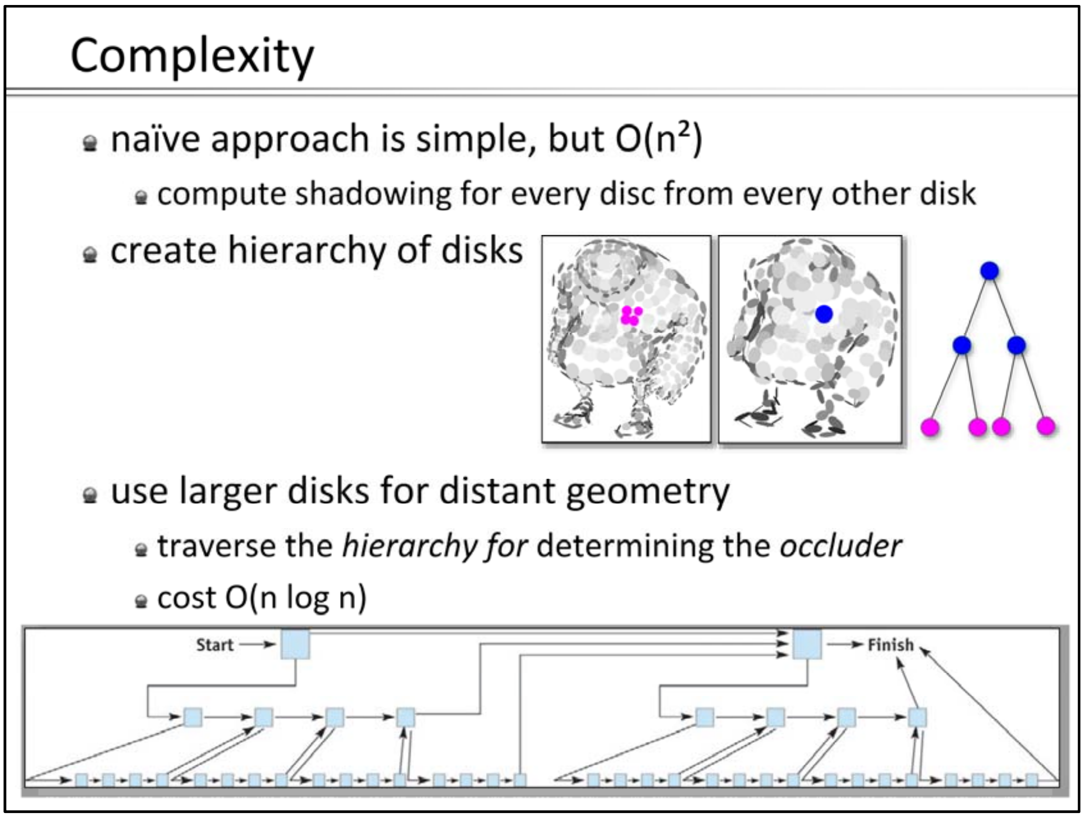
After the first pass, some elements will generally be too dark because other elements that are in shadow casted shadows as well.

For this, a second pass is used, which is essentially doing the same calculation, but this time each form factor is scaled by the emitter element’s accessibility from the previous pass.

I.e. B is already blocked almost entirely by A, so it does not contribute much to the blocking of C.

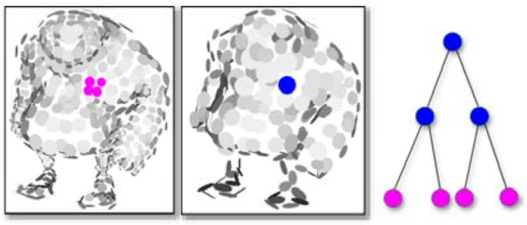
After the second pass, double shadowing is removed. However, surfaces that are triple shadowed or more will generally be too light.

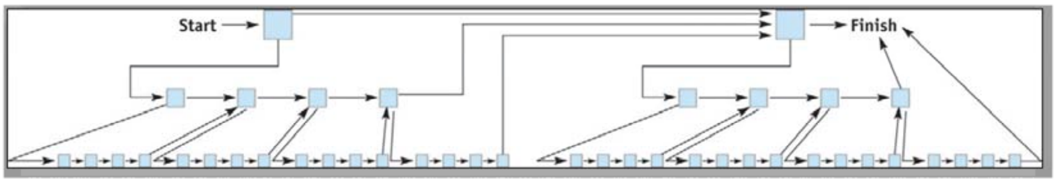
More passes can then be used to get better approximations – Bunnell proposes to blend the results of pass 1 and 2 which ends up being close to the final result.



Complexity

- naïve approach is simple, but O(n²)
  - compute shadowing for every disc from every other disk
- create hierarchy of disks
- use larger disks for distant geometry
  - traverse the *hierarchy for* determining the *occluder*
  - cost O(n log n)





A naïve implementation of this method will have a quadratic complexity: for every element we compute the occlusion from all others.

However, we can also create a hierarchy of elements, by successively merging discs to obtain coarser elements.

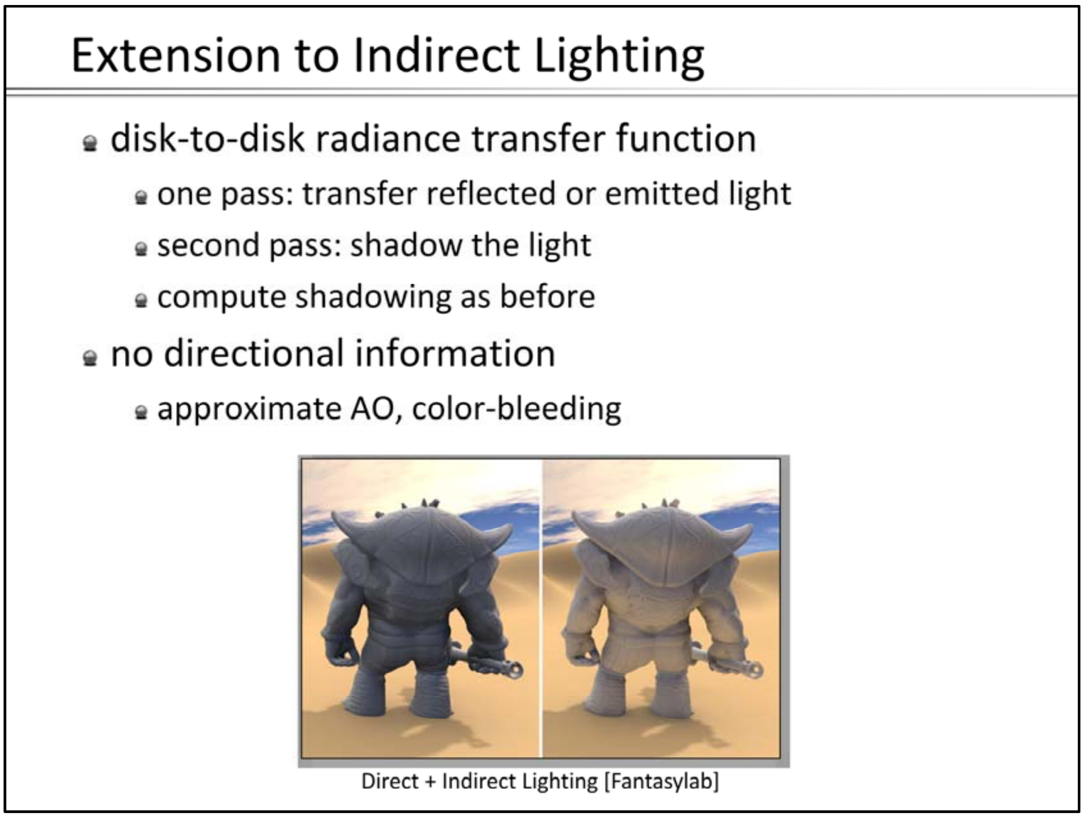
We then still compute the accessibility for every element, but we use the hierarchy for determining the blockers:

we always start from the root of the hierarchy, i.e. the coarsest disc representation, and

whenever the currently considered blocker element is far enough away from the receiver or small enough it is used for the computation.

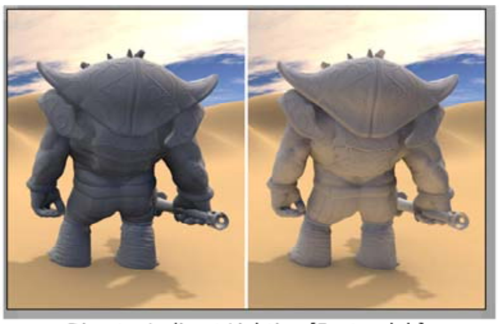
Otherwise the children of the current blocker (if any) are traversed instead.

This blocker-side hierarchy can be efficiently used on a GPU, and the complexity of the algorithm goes from O(n²) to O(n log n).



Extension to Indirect Lighting

- disk-to-disk radiance transfer function
  - one pass: transfer reflected or emitted light
  - second pass: shadow the light
  - compute shadowing as before
- no directional information
  - approximate AO, color-bleeding



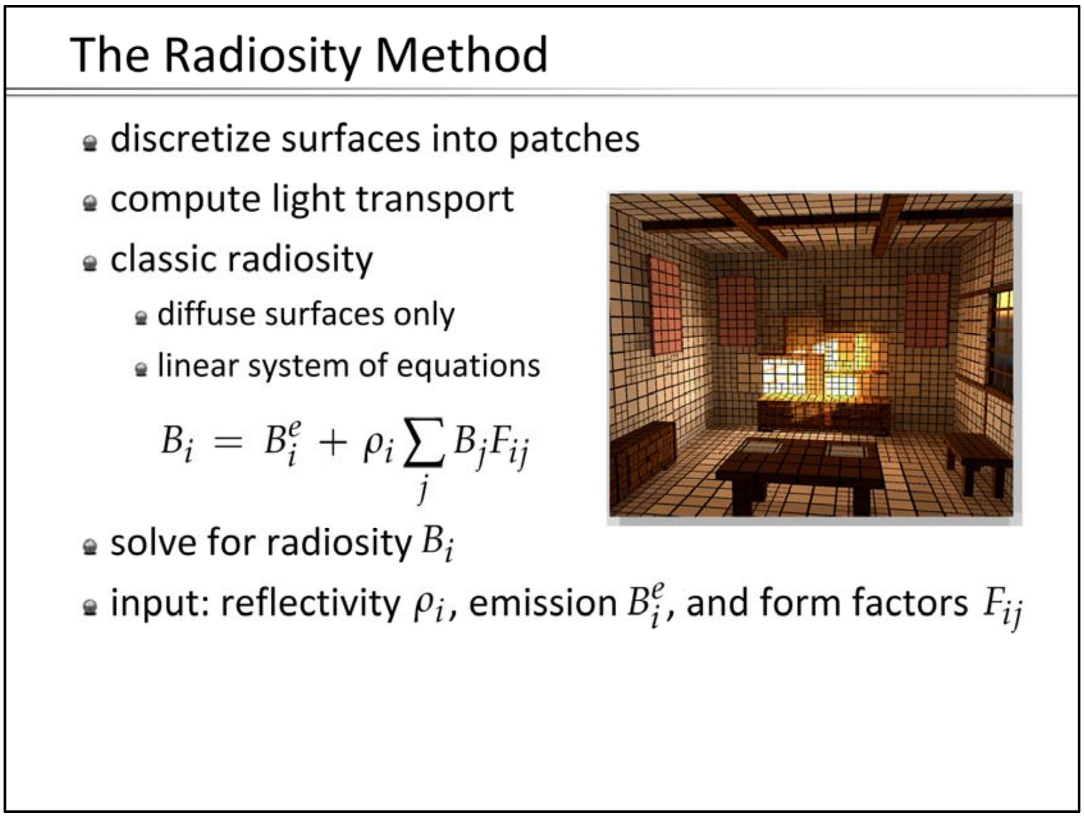
Direct + Indirect Lighting [Fantasylab]

This method can also be extended to account for (fake) indirect lighting.

The idea is that a single bounce of indirect light can be added using a variation of the ambient occlusion method:

We can compute the disk-to-disk radiance transfer, and thus perform one pass to transfer the reflected or emitted light (for light sources) and further passes to shadow the light.

Although this method produces visually pleasing ambient occlusion effects, and color bleeding, it is important to note that it computes a coarse approximation of GI only. This is due to the fact that directional information of radiance is completely ignored.



The Radiosity Method

- discretize surfaces into patches
- compute light transport
- classic radiosity
  - diffuse surfaces only
  - linear system of equations

$$B_i = B_i^e + \rho_i \sum_j B_j F_{ij}$$

- solve for radiosity B_i
- input: reflectivity ρ_i, emission B_i^e, and form factors F_{ij}

Now we‘re switching to radiosity methods that are able to compute real GI.

The classic radiosity method discretizes all surfaces in a scene into a set of small surface patches, and assumes that all of them are pure diffuse reflections.

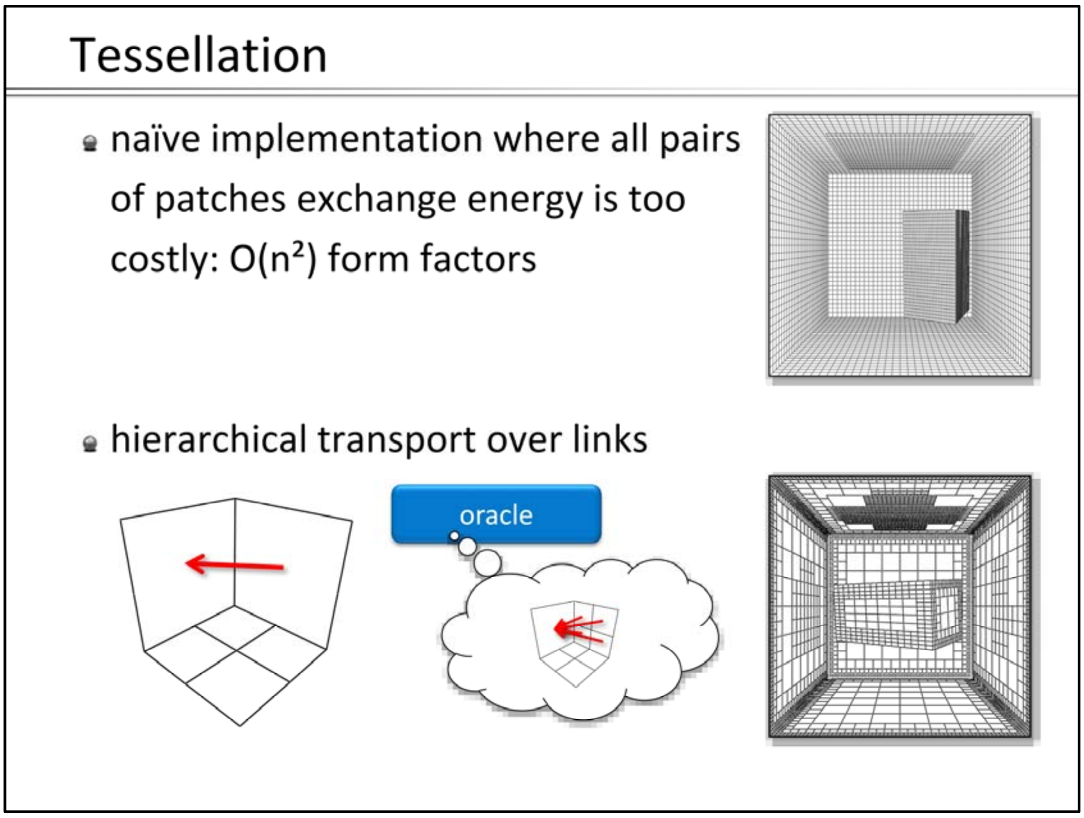
To obatin a solution to the rendering equation, the light transport between these patches is computed until the equilibrium of light distribution in the scene is reached.

Under these assumptions, we can express this computation as a linear system of equations, for which we know the emission and reflectivity of the patches.

And we can compute the form factors which denote the fraction of energy that is leaving one patch and reaching another one.

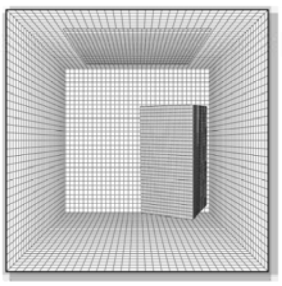
I.e. they tell us how energy is transferred between patches.

Well given all this, we simply need to solve the equations for the patch radiosities…

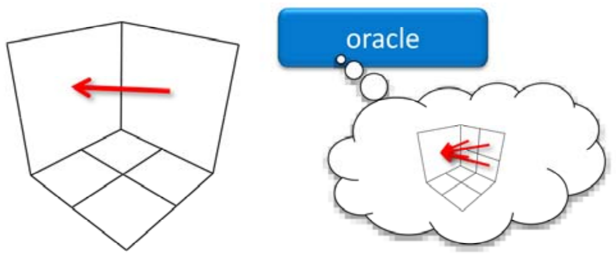


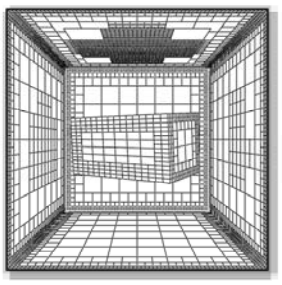
Tessellation

- naïve implementation where all pairs of patches exchange energy is too costly: O(n²) form factors



- hierarchical transport over links





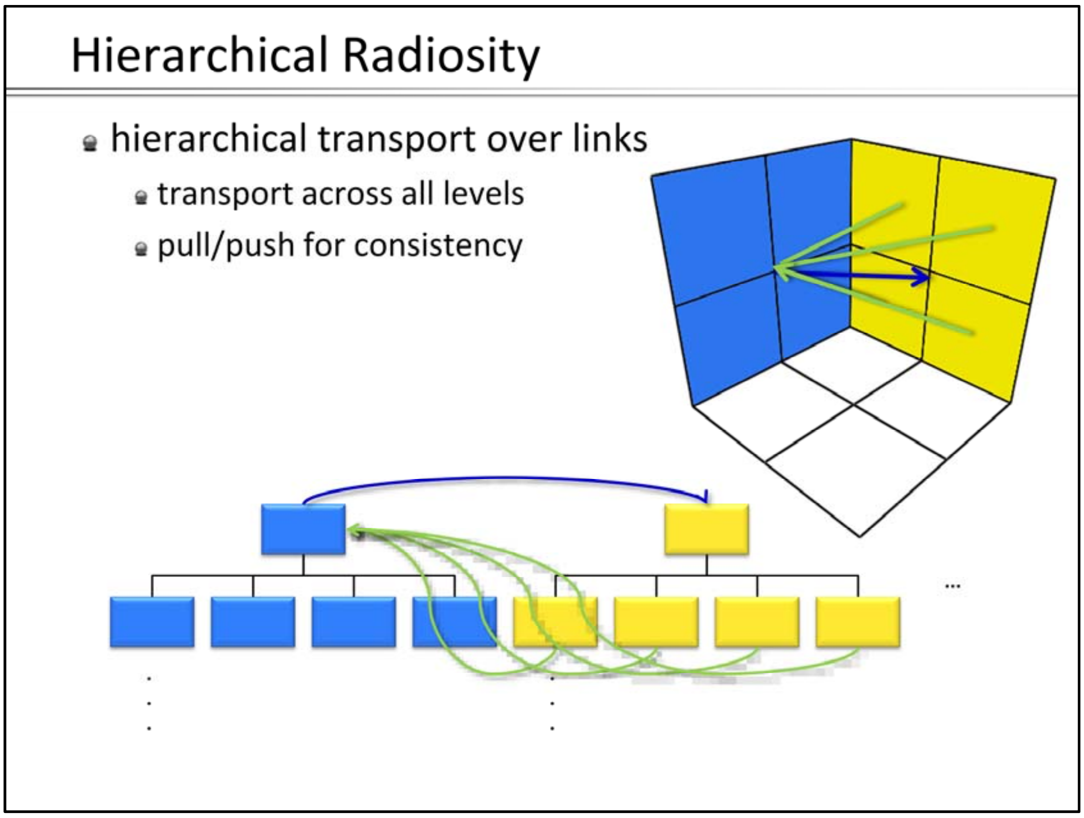
In practice, a naive implementation where all pairs of patches exchange energy is way too costly again.

And thus a hierarchy of patches is created:

In principle we starting from a coarse input mesh where all patches mutually exchange energy.

In hierarchical radiosity this is called linking: we transport energy over links from one patch to another.

Then a so-called oracle decides whether the transport over a link is too large, e.g. if patches are close and large, and in this case, one or both patches are sub-divided and links between their children are established.

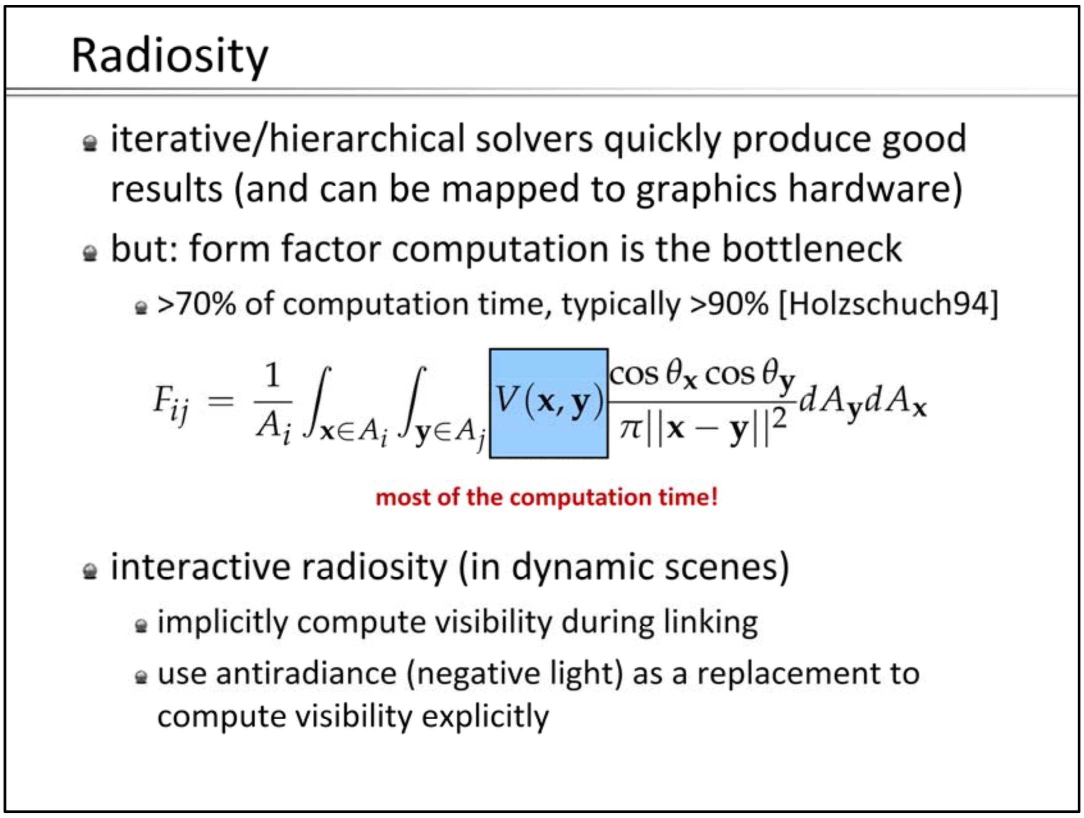


Hierarchical Radiosity

- hierarchical transport over links
  - transport across all levels
  - pull/push for consistency

And from this we get a patch hierarchy and a set of links exchanging energy across all levels.

To ensure consistency push pull steps are used which propagate the energy vertically in the hierarchy.


Radiosity

- iterative/hierarchical solvers quickly produce good results (and can be mapped to graphics hardware)
- but: form factor computation is the bottleneck
  - >70% of computation time, typically >90% [Holzschuch94]

$$F_{ij} = \frac{1}{A_i}\int_{\mathbf{x}\in A_i}\int_{\mathbf{y}\in A_j} \boxed{V(\mathbf{x},\mathbf{y})} \frac{\cos\theta_{\mathbf{x}}\cos\theta_{\mathbf{y}}}{\pi||\mathbf{x}-\mathbf{y}||^2} dA_{\mathbf{y}} dA_{\mathbf{x}}$$

most of the computation time!

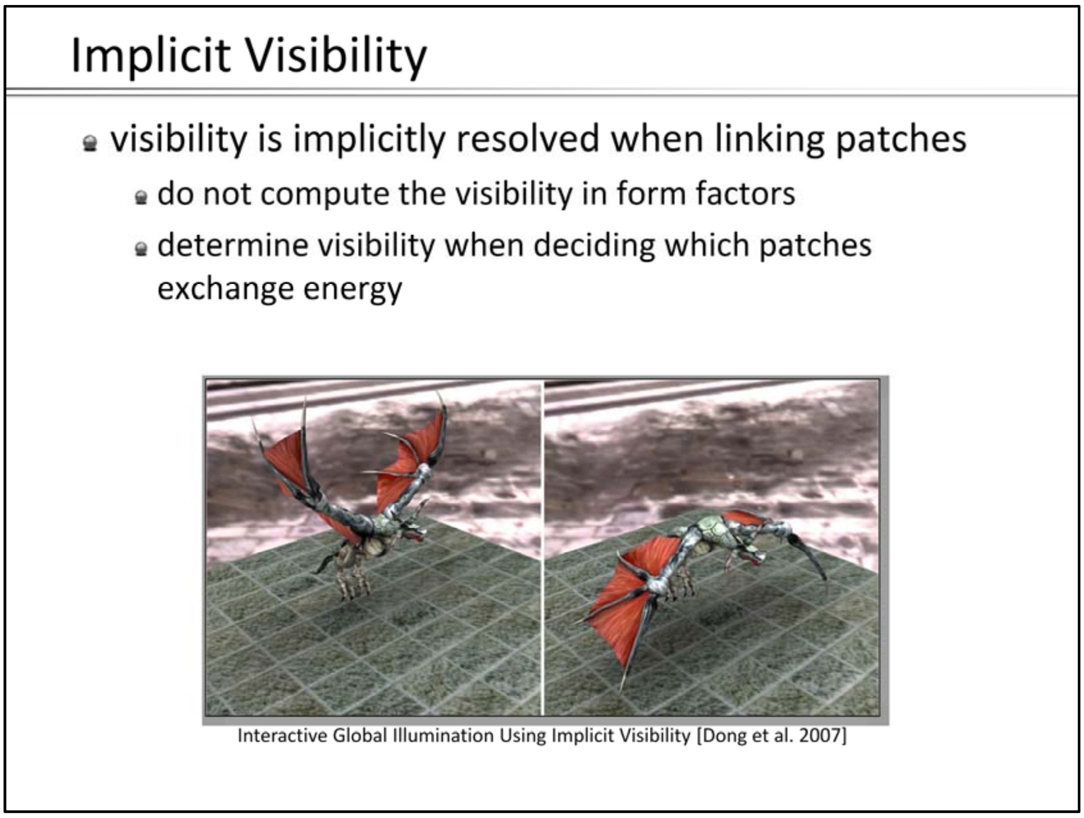
- interactive radiosity (in dynamic scenes)
  - implicitly compute visibility during linking
  - use antiradiance (negative light) as a replacement to compute visibility explicitly

Now a solution can be computed very quickly, also on graphics hardware, once we have the hierarchy and the form factors.

But the bottleneck is computing the form factors where we need the numerical integration of the double integral and evaluation of the visibility function.

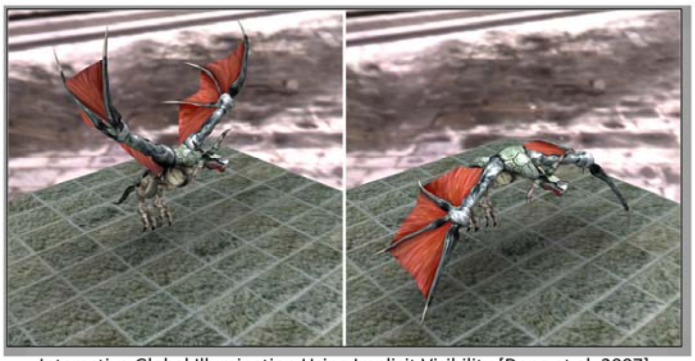
Typically the time spent for the form factors is 70%, most often more than 90% of the total computation time!

Obviously we need a different solution to achieve interactive radiosity in dynamic scenes where form factors need to be recomputed all the time, and I‘ll present two you two options in the next couple of minutes.



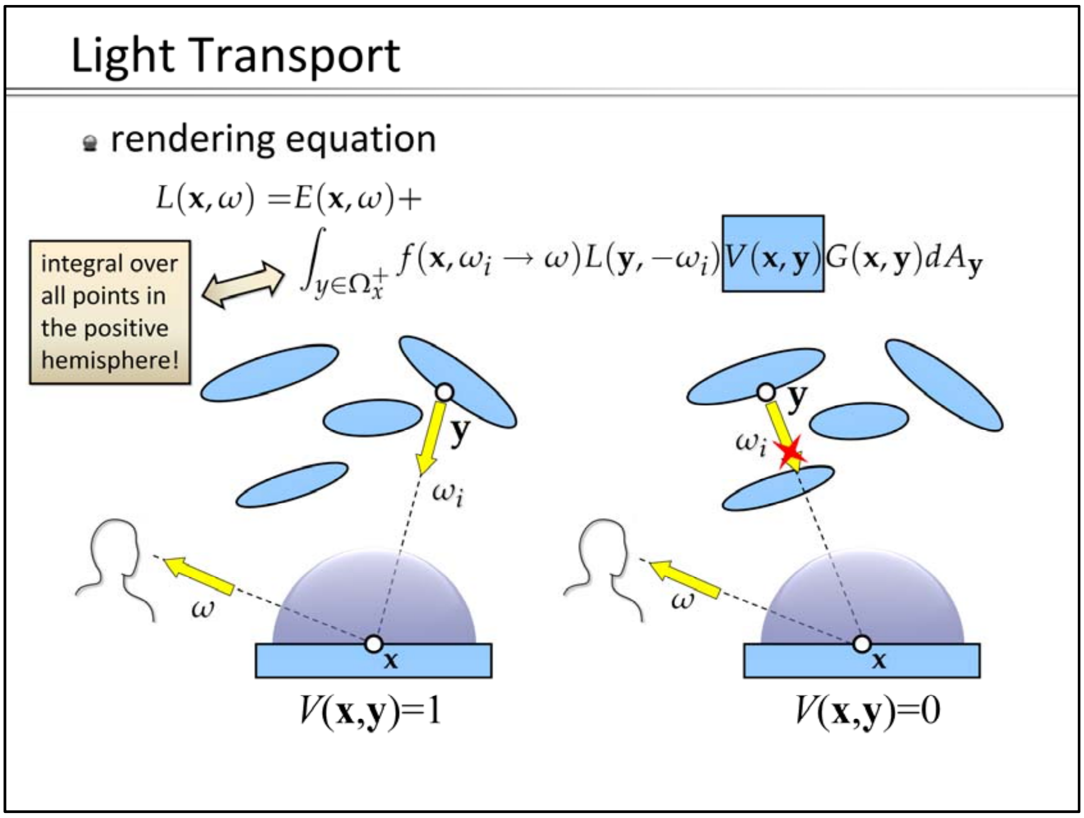
Implicit Visibility

- visibility is implicitly resolved when linking patches
  - do not compute the visibility in form factors
  - determine visibility when deciding which patches exchange energy


Interactive Global Illumination Using Implicit Visibility [Dong et al. 2007]

The first one is Dong et al.'s method which tries to resolve the visibility in the linking phase of the hierarchical radiosity method.

The idea is, not to compute the visibility in the form factor, but only keep links, over which energy is exchanged, and not create links which will have a small contribution to the light transport.



Light Transport

- rendering equation

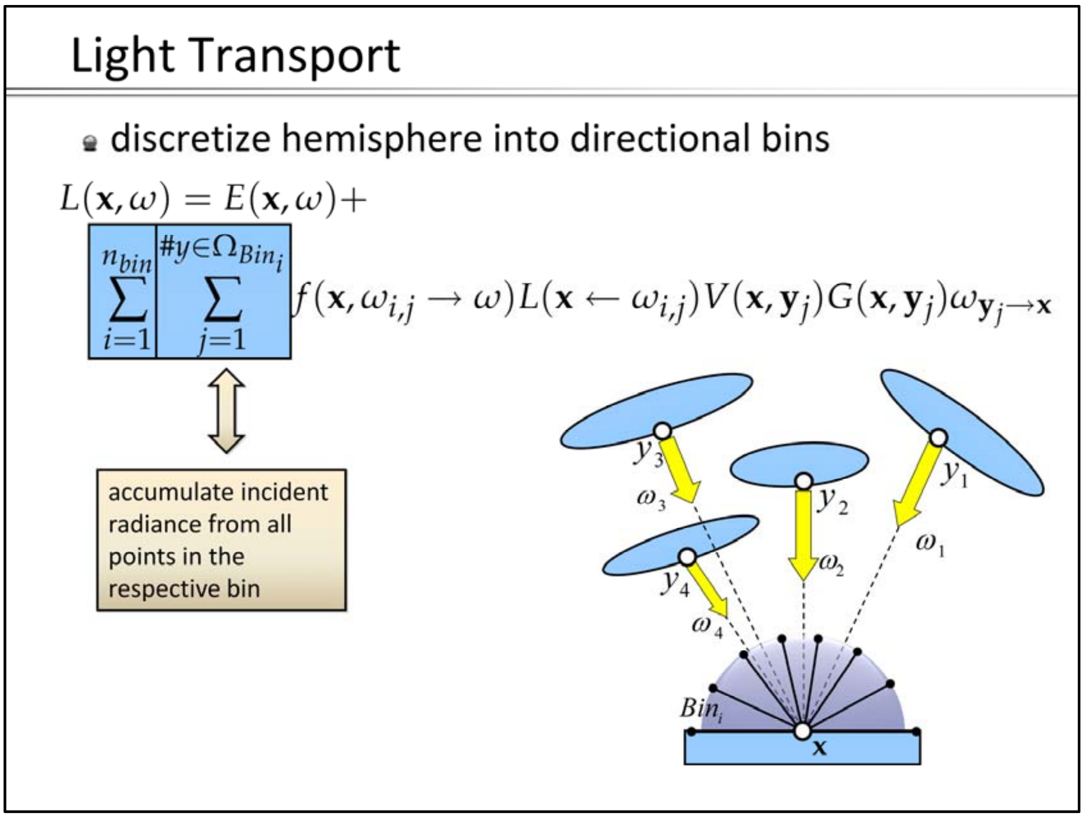
$$L(\mathbf{x},\omega) = E(\mathbf{x},\omega) + \int_{y\in\Omega_x^+} f(\mathbf{x},\omega_i \rightarrow \omega) L(\mathbf{y},-\omega_i) \boxed{V(\mathbf{x},\mathbf{y})} G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$

integral over all points in the positive hemisphere!

y
ω_i
ω
x
$V(\mathbf{x},\mathbf{y})=1$
y
ω_i
ω
x
$V(\mathbf{x},\mathbf{y})=0$

This idea can be best explained, when looking at the directional formulation of the rendering equation.

Here the visibility function appears in the reflection integral, and it equals 1 is light transport between two surface points takes place, and 0 otherwise.



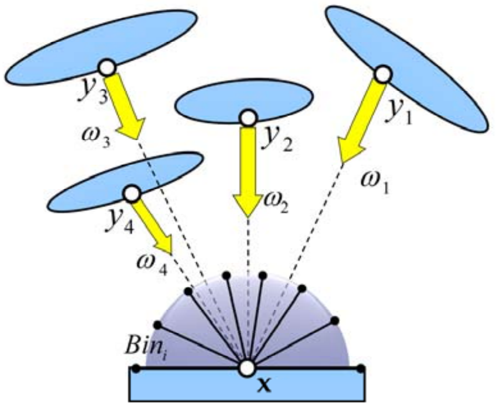
Light Transport

- discretize hemisphere into directional bins

$$L(\mathbf{x},\omega) = E(\mathbf{x},\omega) + \boxed{\sum_{i=1}^{n_{bin}}}\boxed{\sum_{j=1}^{\#y\in\Omega_{Bin_i}}} f(\mathbf{x},\omega_{i,j}\to\omega)L(\mathbf{x}\leftarrow\omega_{i,j})V(\mathbf{x},\mathbf{y}_j)G(\mathbf{x},\mathbf{y}_j)\omega_{\mathbf{y}_j\to\mathbf{x}}$$

$$\Updownarrow$$

accumulate incident radiance from all points in the respective bin



Dong’s method now approximates the rendering equation, by discretizing the hemisphere into a collection of directional bins, this is the first summation.

And next, we can accumulate the incident radiance at the surface point x, from all surface points y which lie in the respective bins.

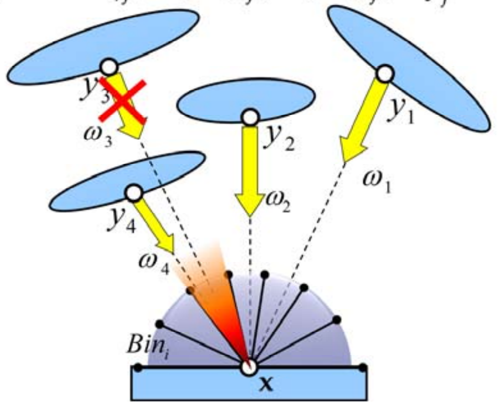
Note that the radiance values from those points are always multiplied with the visibility function!

Light Transport

- discretize visibility function: only one element visible in each bin → only one link per bin

$$K_i(\mathbf{x},\omega) = \sum_{j=1}^{\#y\in\Omega_{Bin_i}} f(\mathbf{x},\omega_{i,j}\to\omega)L(\mathbf{x}\leftarrow\omega_{i,j})V(\mathbf{x},\mathbf{y}_j)G(\mathbf{x},\mathbf{y}_j)\omega_{\mathbf{y}_j\to\mathbf{x}}$$

- determine visibility implicitly during linking
- when a link is created:
  - determine bin
  - if it's the closest link: keep
  - otherwise: omit

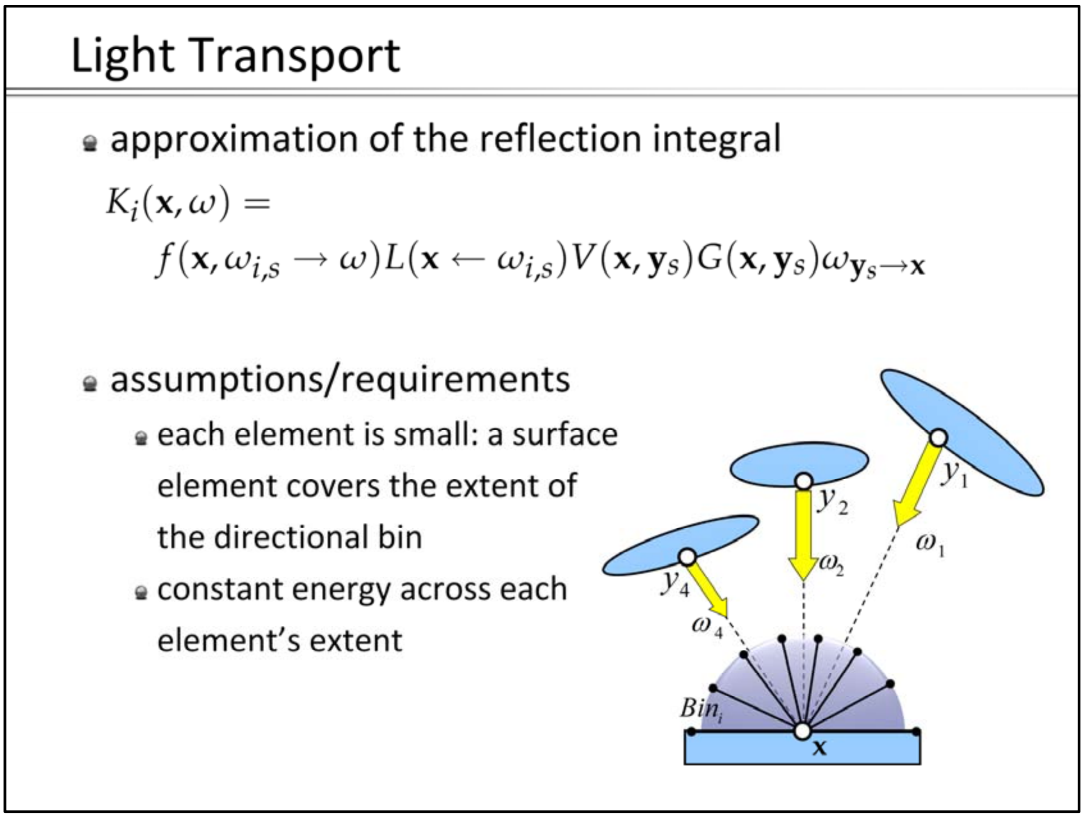


The key idea of this method is now to discretize the visibility function such that no more than one element is visible for each bin.

In this example, two surface points y3 and y4 lie in the same bin, and since only one can be visible, we take y4, because it is closer to x.

The point is, that this visibility can be determined implicitly during the linking phase:

Whenever a link is to be created, its direction is discretized, and a lookup into the list of bins tells us whether to store the link (and remove another one), or to omit it.

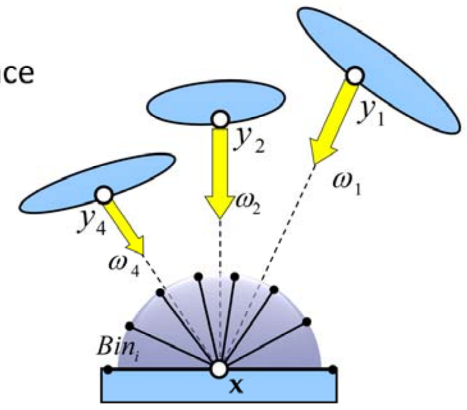


Light Transport

- approximation of the reflection integral

$$K_i(\mathbf{x},\omega) = f(\mathbf{x},\omega_{i,s} \rightarrow \omega)L(\mathbf{x} \leftarrow \omega_{i,s})V(\mathbf{x},\mathbf{y}_s)G(\mathbf{x},\mathbf{y}_s)\omega_{\mathbf{y}_s \rightarrow \mathbf{x}}$$

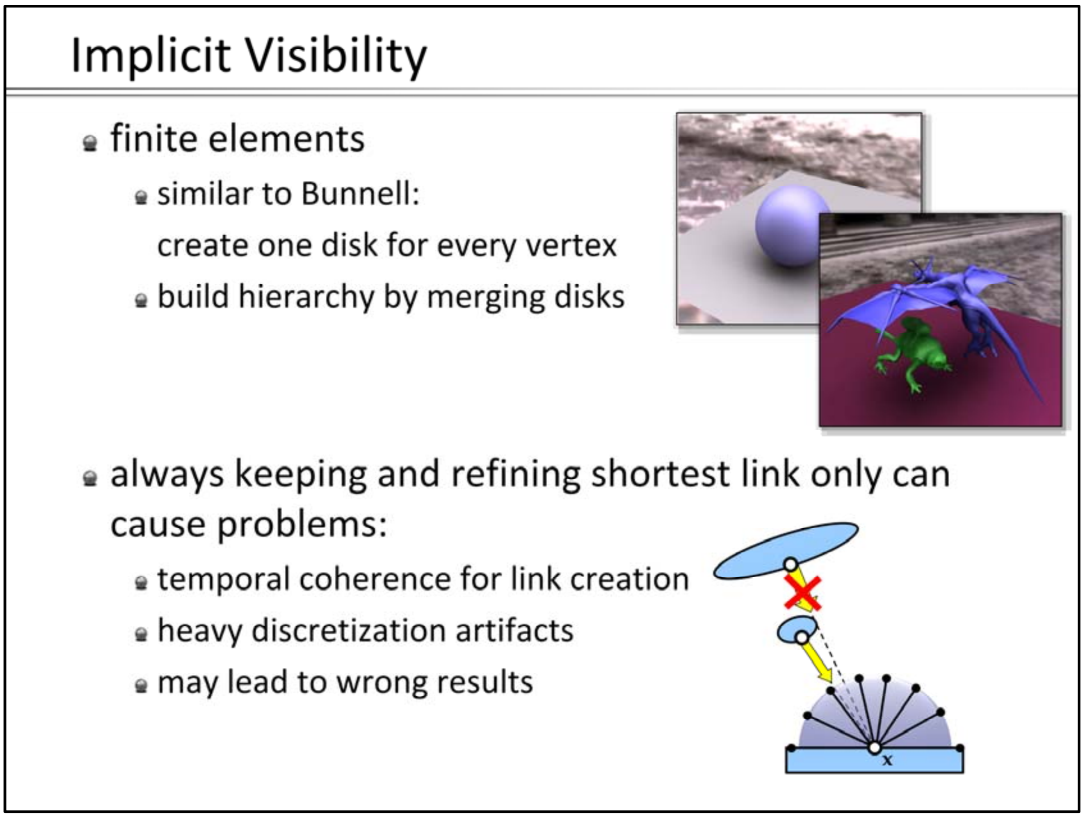
- assumptions/requirements
  - each element is small: a surface element covers the extent of the directional bin
  - constant energy across each element's extent



These simplifications are based on some assumptions or requirements that need to be met to ensure good rendering quality.

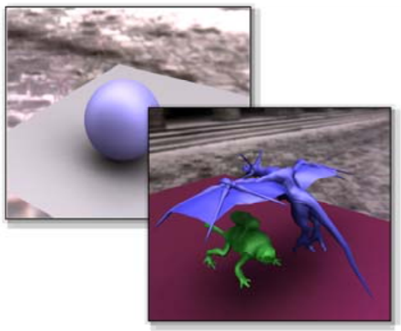
First of all, elements should be small and subtend a small solid angle only, ideally the solid angle equal to the size of a directional bin.

And second, the energy should be constant across the element but this is a common assumption for many radiosity methods.

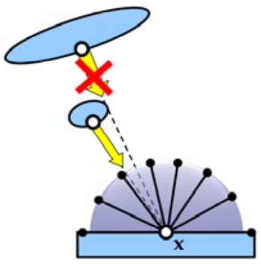


Implicit Visibility

- finite elements
  - similar to Bunnell: create one disk for every vertex
  - build hierarchy by merging disks



- always keeping and refining shortest link only can cause problems:
  - temporal coherence for link creation
  - heavy discretization artifacts
  - may lead to wrong results

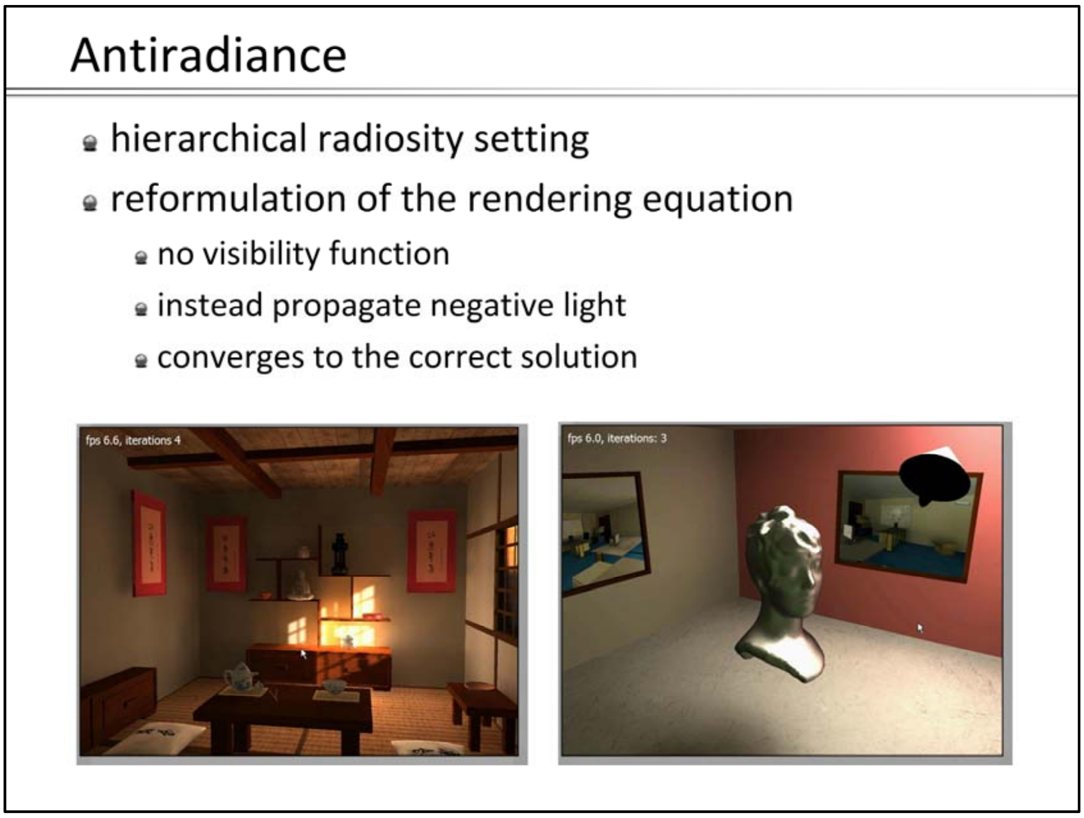


The finite elements used in this method are disc elements similar to Bunnell‘s method.

The underlying idea to keep only one link for a discretized direction, however, causes problems in many situations:

First of all, temporal coherence is an issue, due to the heavy discretization.

And second: the solution can be arbitrarily wrong, as seen in this example, where a very small surface blocks the light transport from a big one entirely



Antiradiance

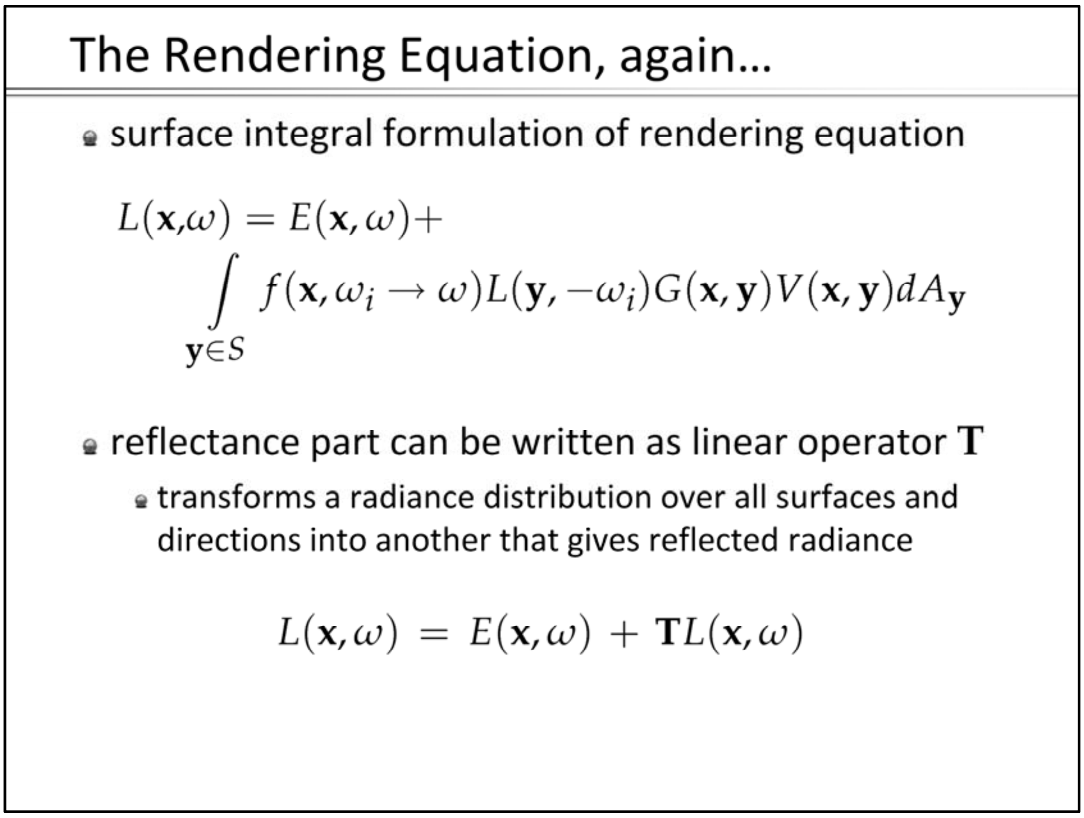
- hierarchical radiosity setting
- reformulation of the rendering equation
  - no visibility function
  - instead propagate negative light
  - converges to the correct solution

fps 6.6, iterations 4

fps 6.0, iterations: 3

The second option for radiosity on GPUs in the antiradiance method, which is also based on hierarchical radiosity, but uses a reformulation of the rendering equation which does not require visibility computations.

Instead, negative light in generated and propagated to cast shadows and compensate for occlusion (which is not computed).



The Rendering Equation, again…

- surface integral formulation of rendering equation

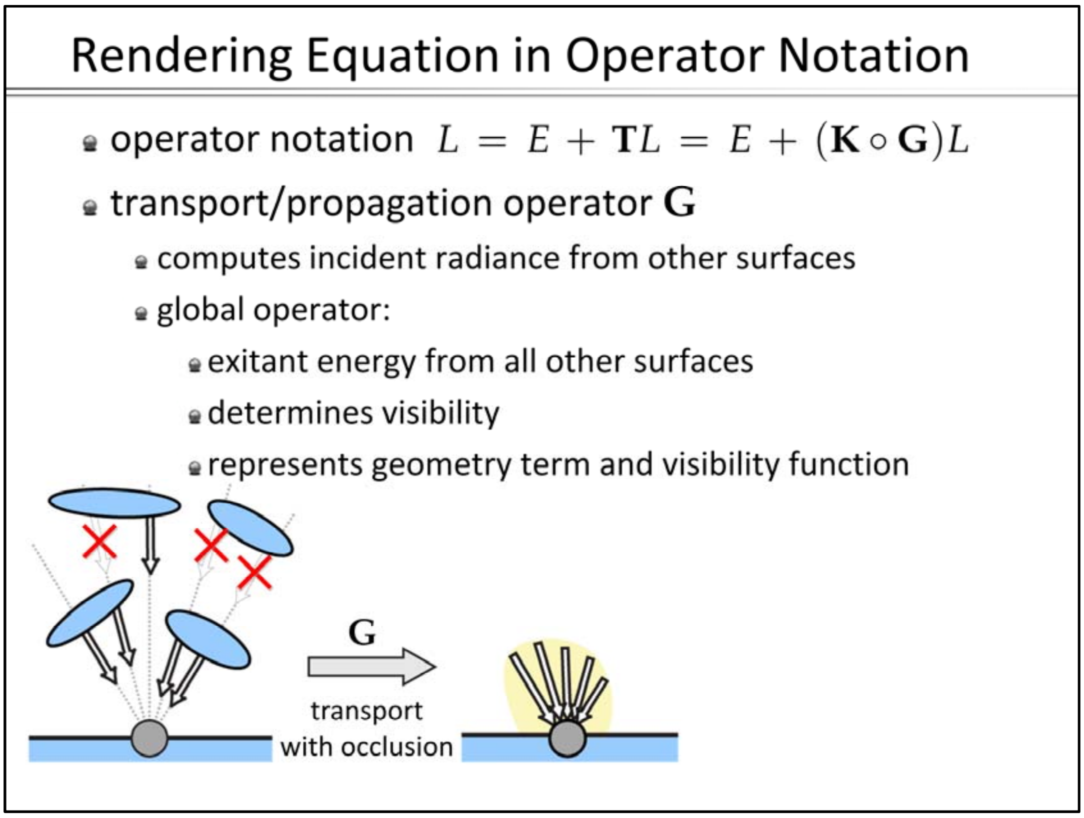
$$L(\mathbf{x},\omega) = E(\mathbf{x},\omega) + \int_{\mathbf{y}\in S} f(\mathbf{x},\omega_i \to \omega) L(\mathbf{y},-\omega_i) G(\mathbf{x},\mathbf{y}) V(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$

- reflectance part can be written as linear operator $\mathbf{T}$
  - transforms a radiance distribution over all surfaces and directions into another that gives reflected radiance

$$L(\mathbf{x},\omega) = E(\mathbf{x},\omega) + \mathbf{T}L(\mathbf{x},\omega)$$

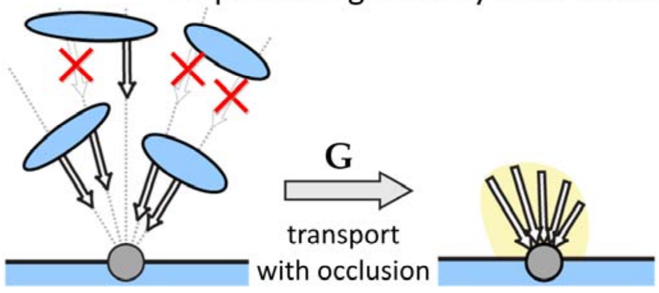
In order to explain the underlying idea, we will use an operator notation for the rendering equation which is shown here in it‘s surface integral formulation.

We can use a linear transport operator T for the integral, which essentially transforms a radiance distribution over all surfaces and directions into another distribution that gives us the reflected radiance.



Rendering Equation in Operator Notation

- operator notation $L = E + \mathbf{T}L = E + (\mathbf{K} \circ \mathbf{G})L$
- transport/propagation operator $\mathbf{G}$
  - computes incident radiance from other surfaces
  - global operator:
    - exitant energy from all other surfaces
    - determines visibility
    - represents geometry term and visibility function

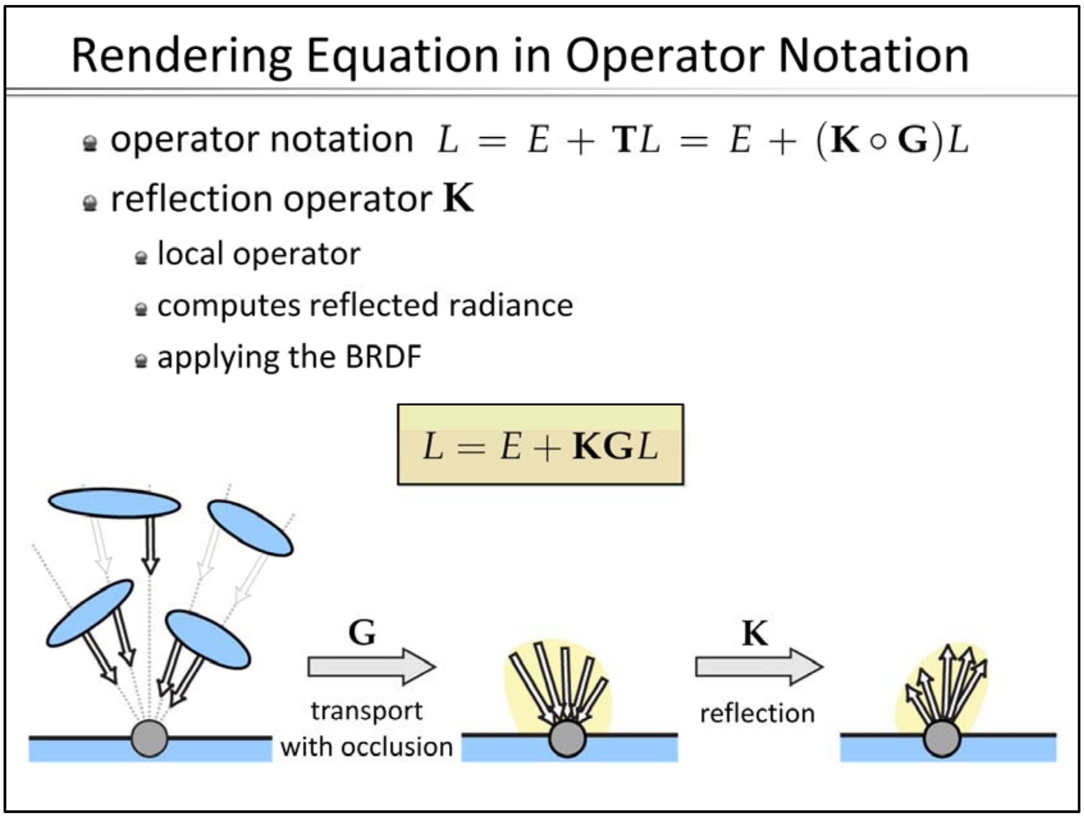


Actually this linear operator can be seen as a concatenation of two other operators.

The first one, the propagation operator G deals with the propagation of light. It computes the incident radiance at a surface point.

And to do so this operators considers all other surfaces to detect if and how much of their emitted energy reaches another surface.

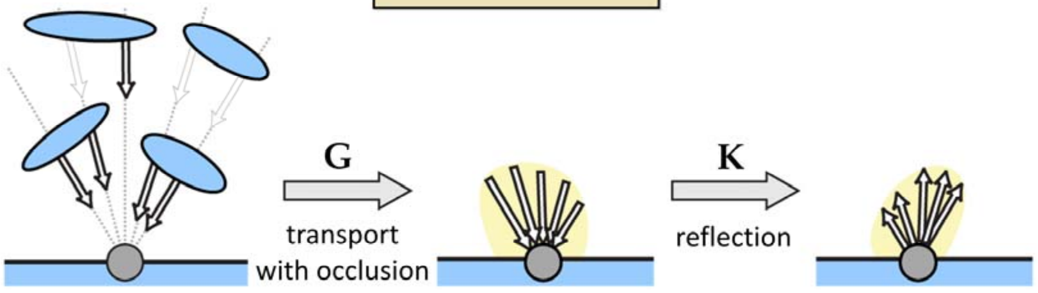
This of course requires the computation of visibility!



Rendering Equation in Operator Notation

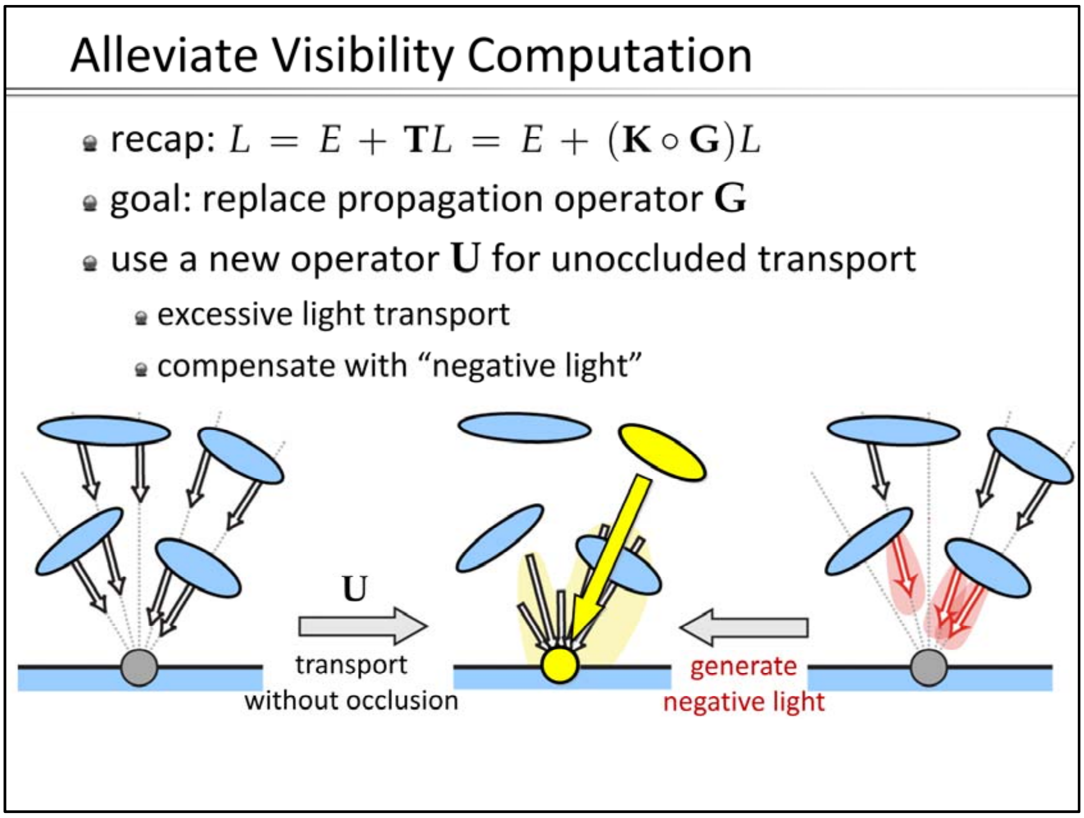
- operator notation $L = E + \mathbf{T}L = E + (\mathbf{K} \circ \mathbf{G})L$
- reflection operator $\mathbf{K}$
  - local operator
  - computes reflected radiance
  - applying the BRDF

$$\boxed{L = E + \mathbf{KG}L}$$



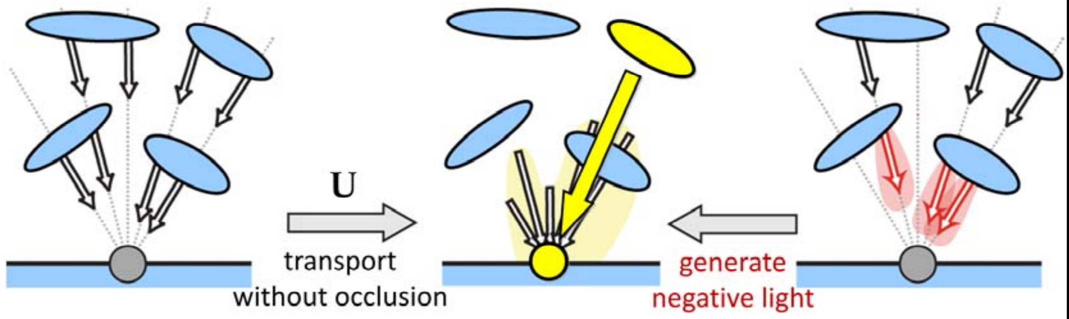
The local reflection operator K works on a single surface point only and computes the distribution of the reflected radiance.

Both operators G and K together replace the integral term in the rendering equation and it can be written as shown here (now without the position and direction parameters).



Alleviate Visibility Computation

- recap: $L = E + \mathbf{T}L = E + (\mathbf{K} \circ \mathbf{G})L$
- goal: replace propagation operator $\mathbf{G}$
- use a new operator $\mathbf{U}$ for unoccluded transport
  - excessive light transport
  - compensate with “negative light”



The goal of the antiradiance method is to get rid of the costly explicit computation of visibility, which is part of the transport operator G.

And the intuition that led to the reformulation is shown here:

We define a new transport operator U which transports light but simply ignores occlusion

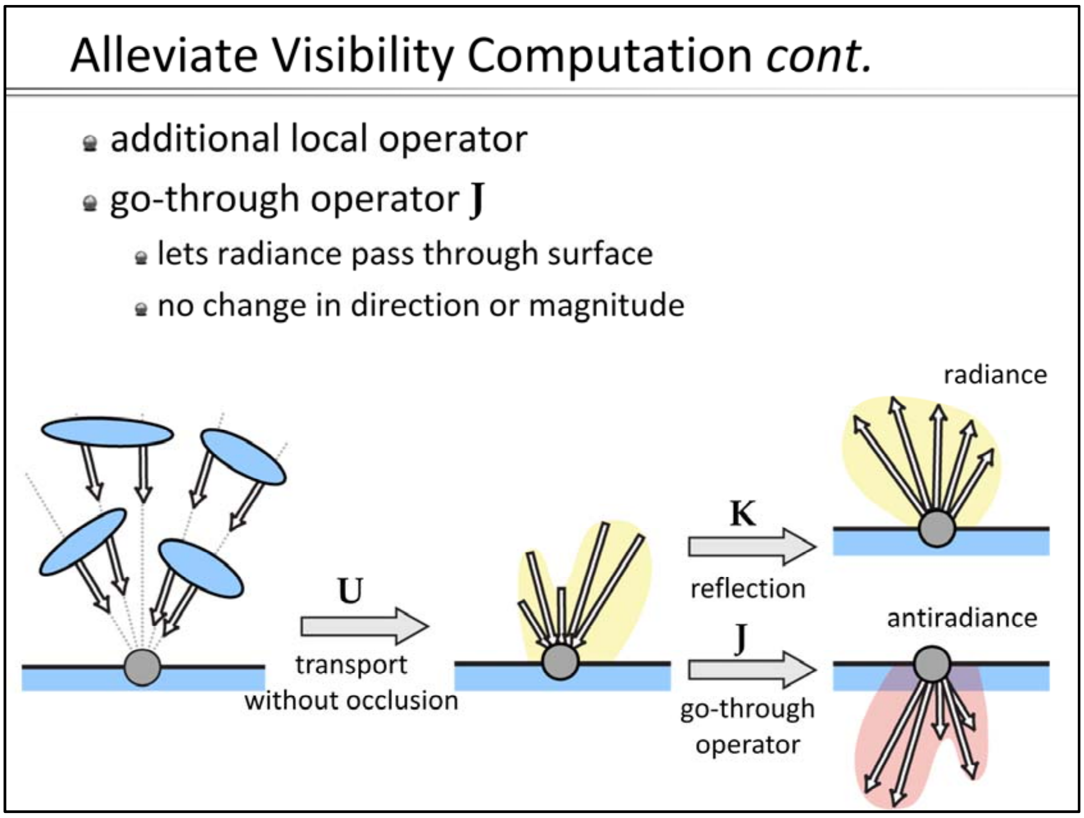
It is easier to compute since the visibility function, which is costly to evaluate, is not required anymore

This means, that now every pair of surfaces exchanges energy,

for example the highlighted surface on the top-right exchanges energy with the bottom surface.

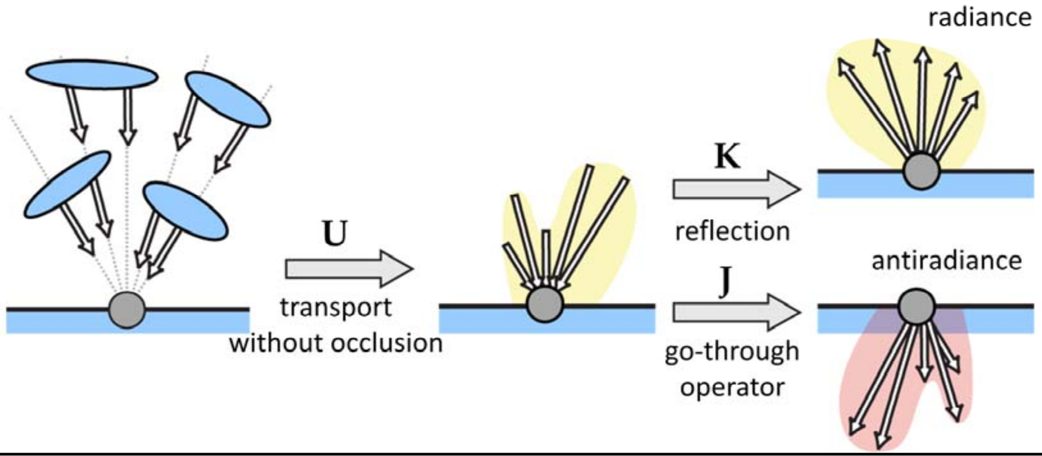
This unoccluded transport is easier to compute, because we don’t have to verify if there’s a blocker or not.

On the other hand, we have to compensate for the resulting excessive light transport – and this will be done using negative light, which is generated on the back of all surfaces from incident radiance on their front-side.



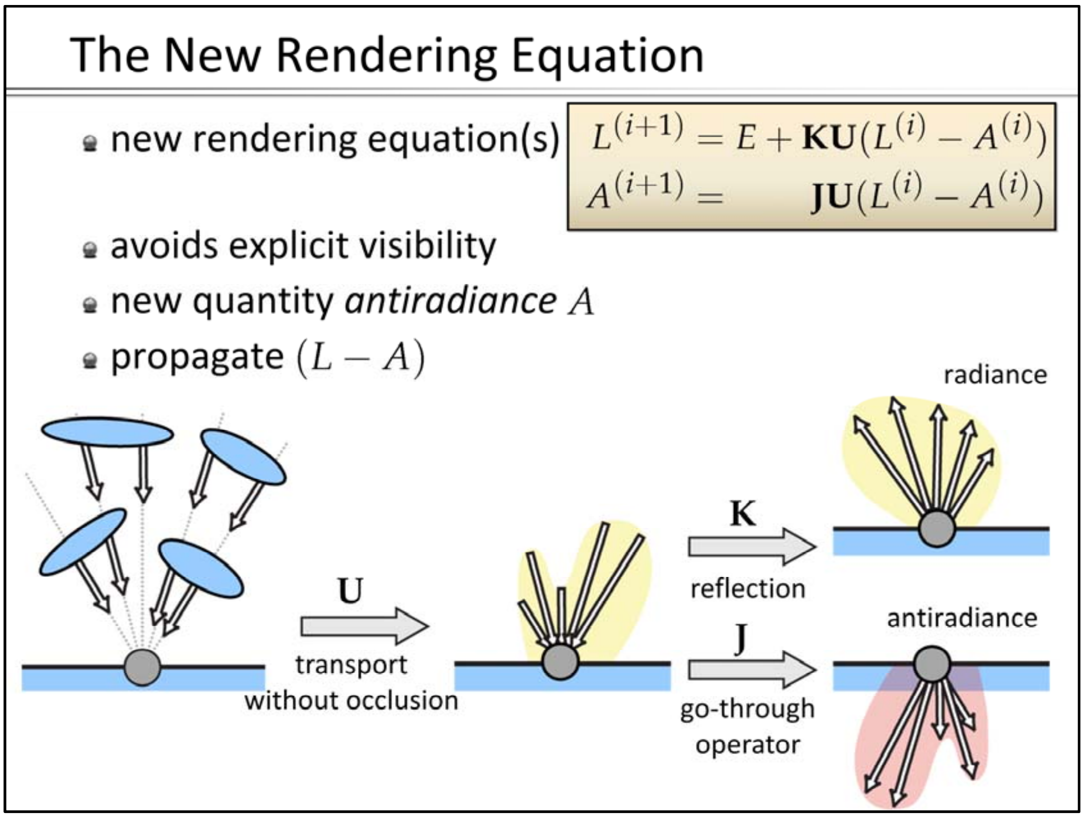
Alleviate Visibility Computation *cont.*

- additional local operator
- go-through operator $\mathbf{J}$
  - lets radiance pass through surface
  - no change in direction or magnitude



To do so, a second local operator, the “go-through operator J” is used.

This new operator lets radiance pass through surfaces without changing its direction or magnitude, effectively generating the negative light.



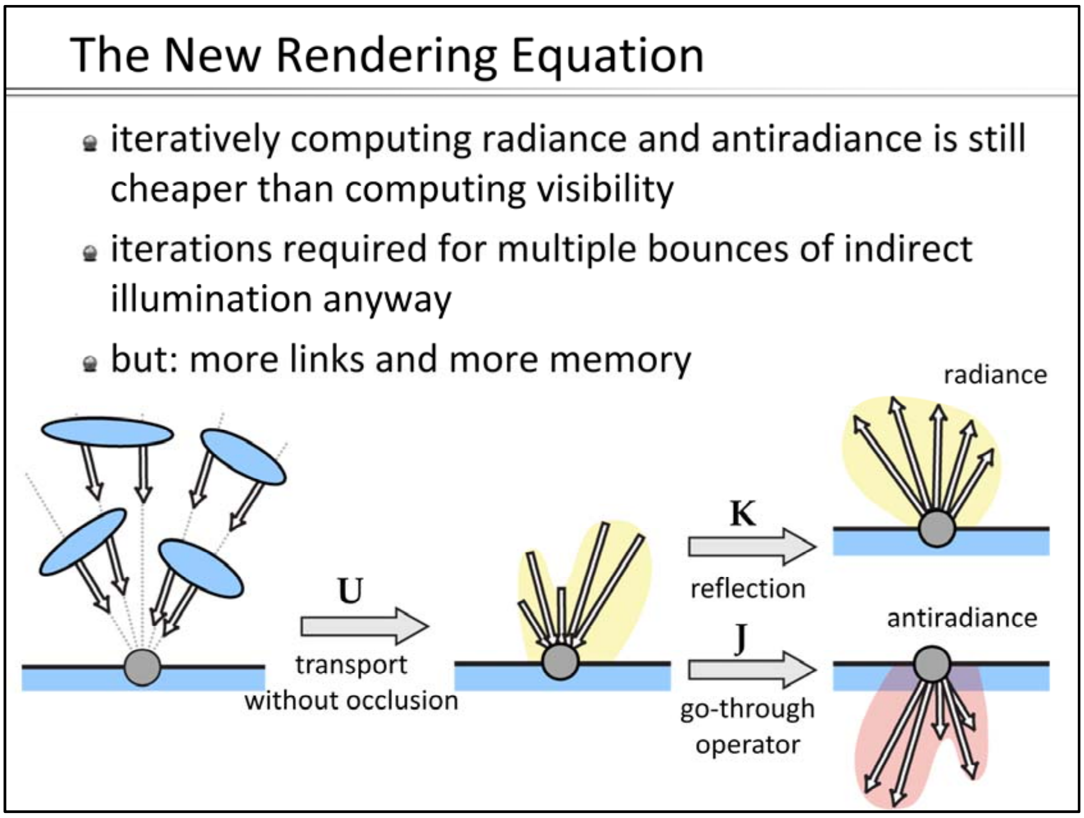
The New Rendering Equation

- new rendering equation(s) $$L^{(i+1)} = E + \mathbf{KU}(L^{(i)} - A^{(i)})$$ $$A^{(i+1)} = \mathbf{JU}(L^{(i)} - A^{(i)})$$
- avoids explicit visibility
- new quantity *antiradiance* A
- propagate $(L - A)$

radiance
K
reflection
U
transport
without occlusion
antiradiance
J
go-through
operator

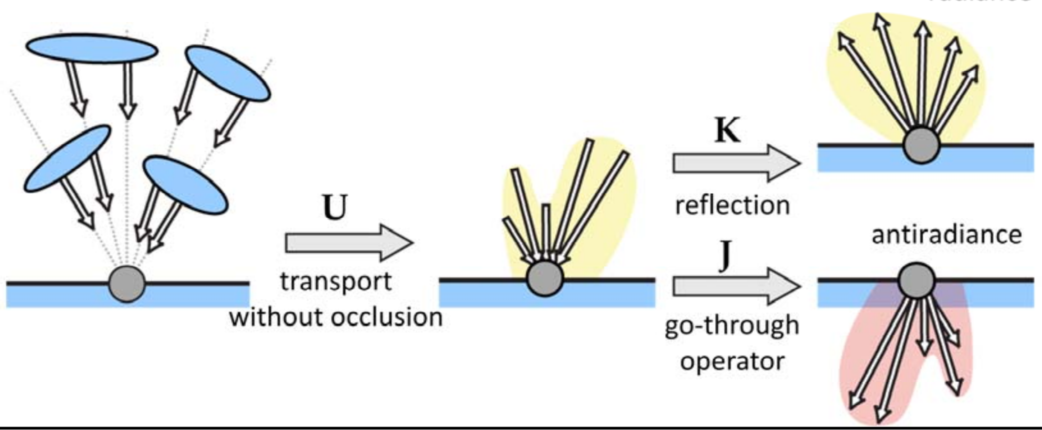
After putting these ideas together, please refer to the paper for the details, this ends up in a new formulation of the rendering equation which now uses a new quantity called antiradiance and a transport operator which is easy to compute because it does not consider visibility.

Using this formulation, we need to iteratively solve for two equations and propagate the difference of radiance and antiradiance.



The New Rendering Equation

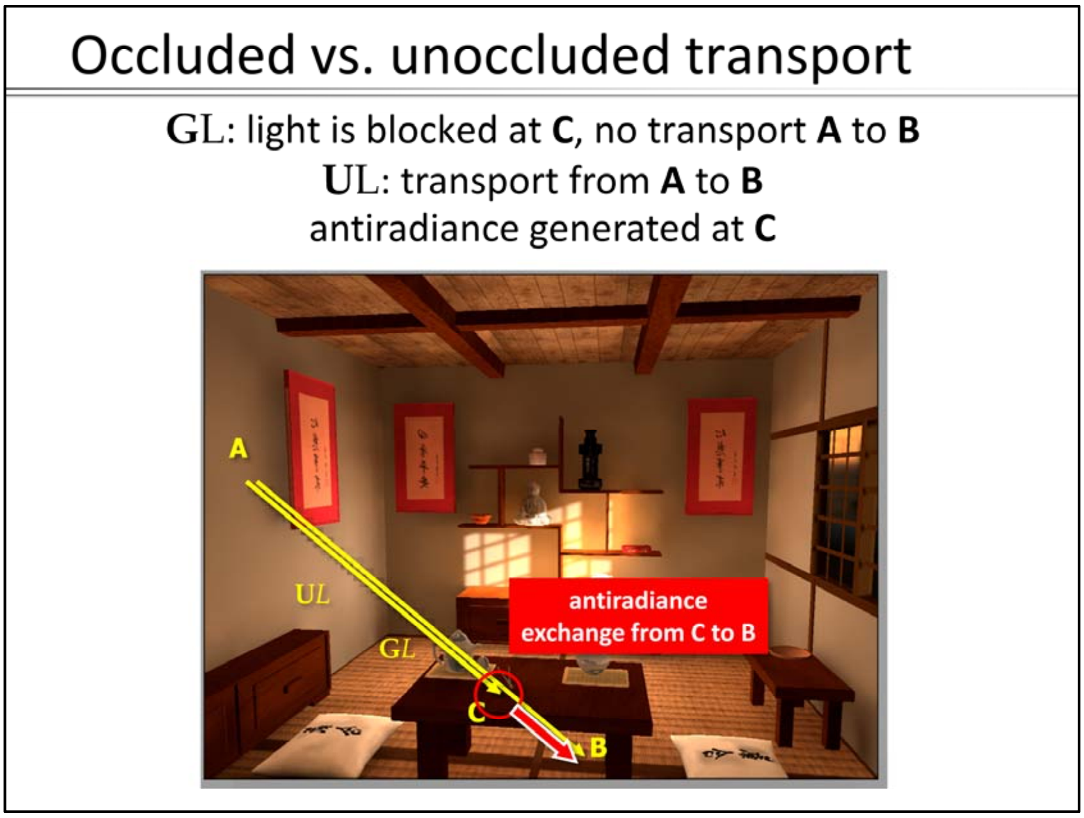
- iteratively computing radiance and antiradiance is still cheaper than computing visibility
- iterations required for multiple bounces of indirect illumination anyway
- but: more links and more memory



Computing a solution now requires an iteration and the computation of radiance and antiradiance, but this proved to be much faster than computing form factors with visibility.

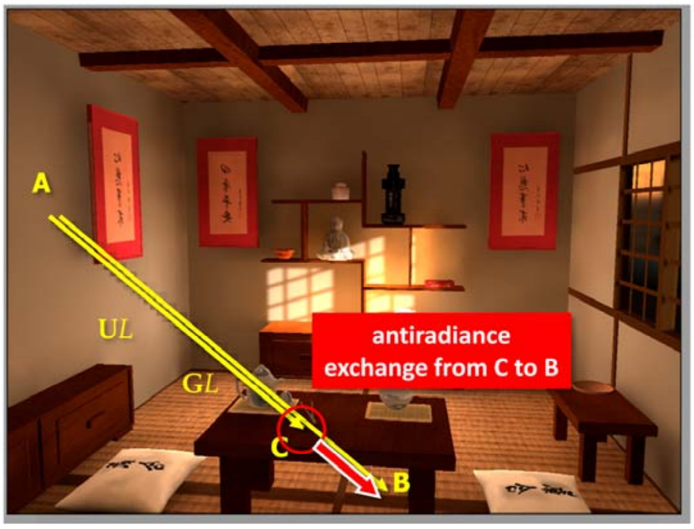
Moreover, multiple iterations would be necessary for multiple bounces of indirect illumination anyway.

Of course not everything comes for free: in general, using antiradiance you will need more links to transport the negative light between patches, and you will need more memory to store the antiradiance for every patch which is a directional quantity.



Occluded vs. unoccluded transport

GL: light is blocked at **C**, no transport **A** to **B**
UL: transport from **A** to **B**
antiradiance generated at **C**



So let’s see what antiradiance means for global illumination?

In this example, we’re going to transport energy from point A to point B.

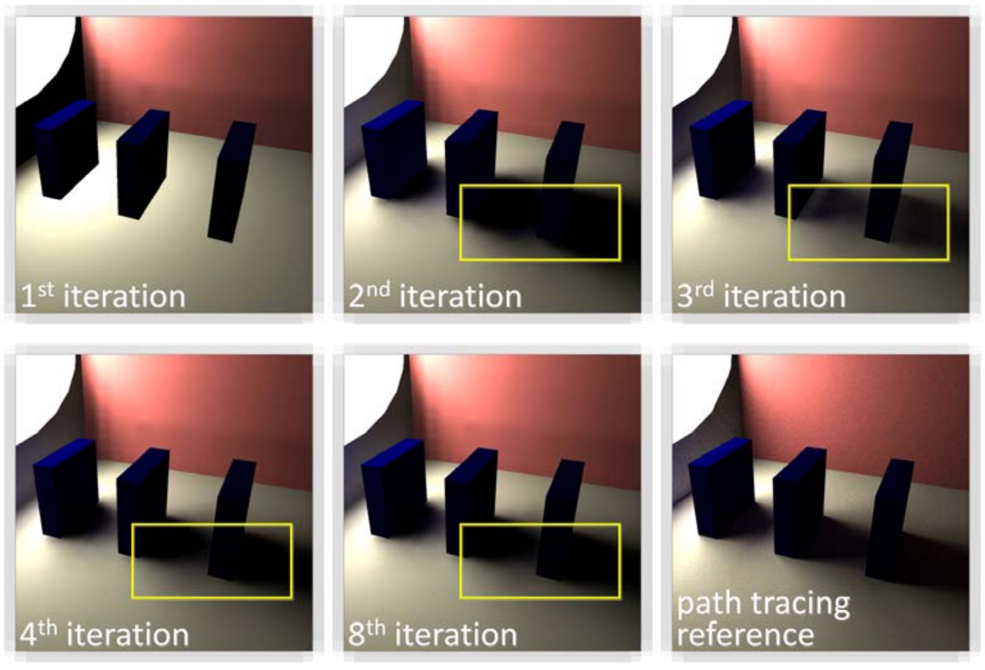
The standard procedure is to determine if there’s any surface blocking the path.

In this scene, the table – at point C - is blocking the light.

With unoccluded transport we don’t need to check what’s inbetween A to B and light is transferred directly.

But this is not a problem: we also transport light to C where antiradiance is generated and afterwards transported to B, where it cancels out the extraneously transported light, thus producing the correct result.

Symmetric Iteration



Here you can see the whole thing working in a complete scene:

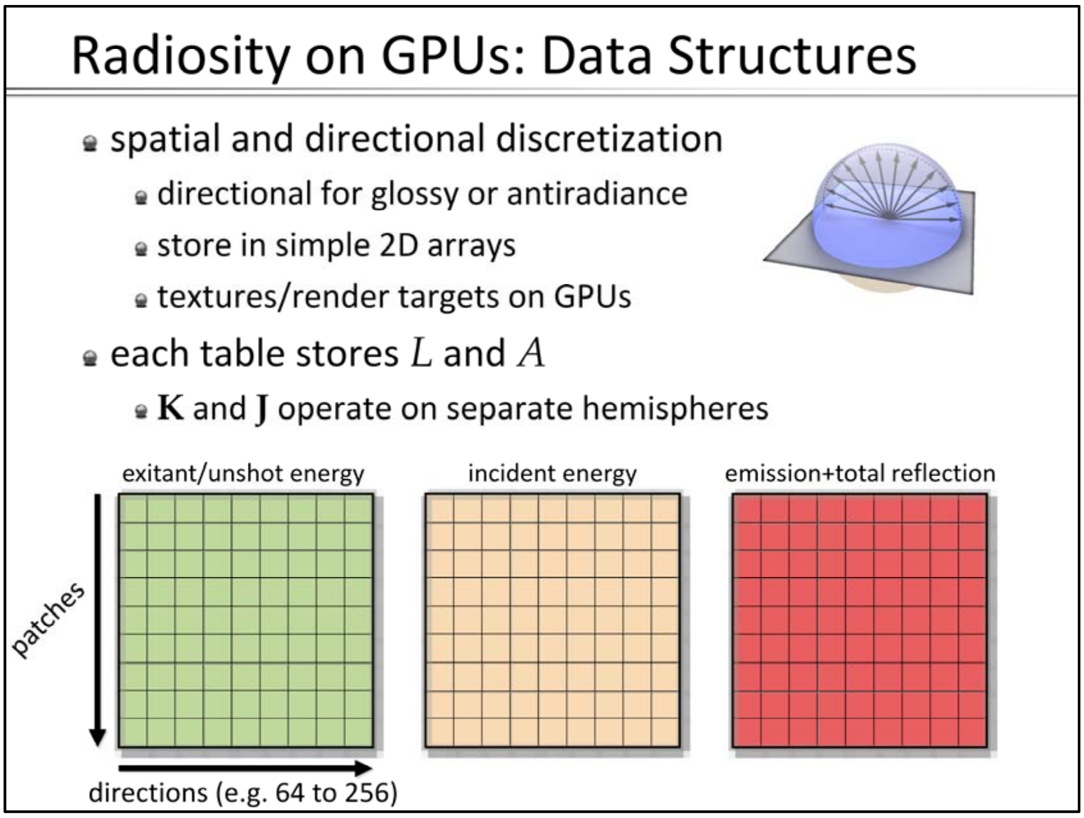
This is the result after 1 iteration, where light bounces off a white wall on the left – this is basically one-bounce indirect light without shadows from indirect illumination.

After this iteration antiradiance is generated for the first time and propagated to the scene in the second iteration.

In the area inside the yellow box there's now too much negative light coming from multiple blockers and in fact displayed radiance values are locally negative.

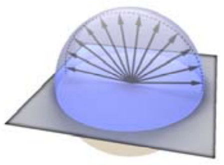
In the following iterations, the interplay between radiance and antiradiance progressively convergences to the correct solution.

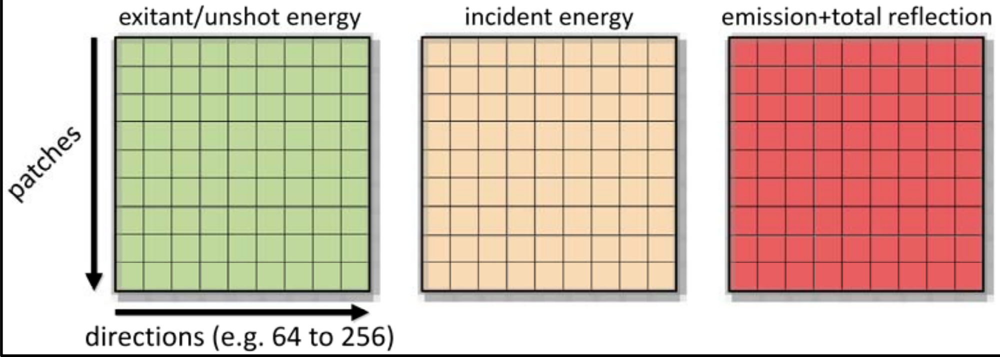
In the lower-right image, you can see a comparison to a path tracing image which demonstrates the correctness of the technique.



Radiosity on GPUs: Data Structures

- spatial and directional discretization
  - directional for glossy or antiradiance
  - store in simple 2D arrays
  - textures/render targets on GPUs
- each table stores L and A
  - $\mathbf{K}$ and $\mathbf{J}$ operate on separate hemispheres





Computing radiosity solutions on the GPU is pretty easy with both variants that I explained once you have the tessellation and the links

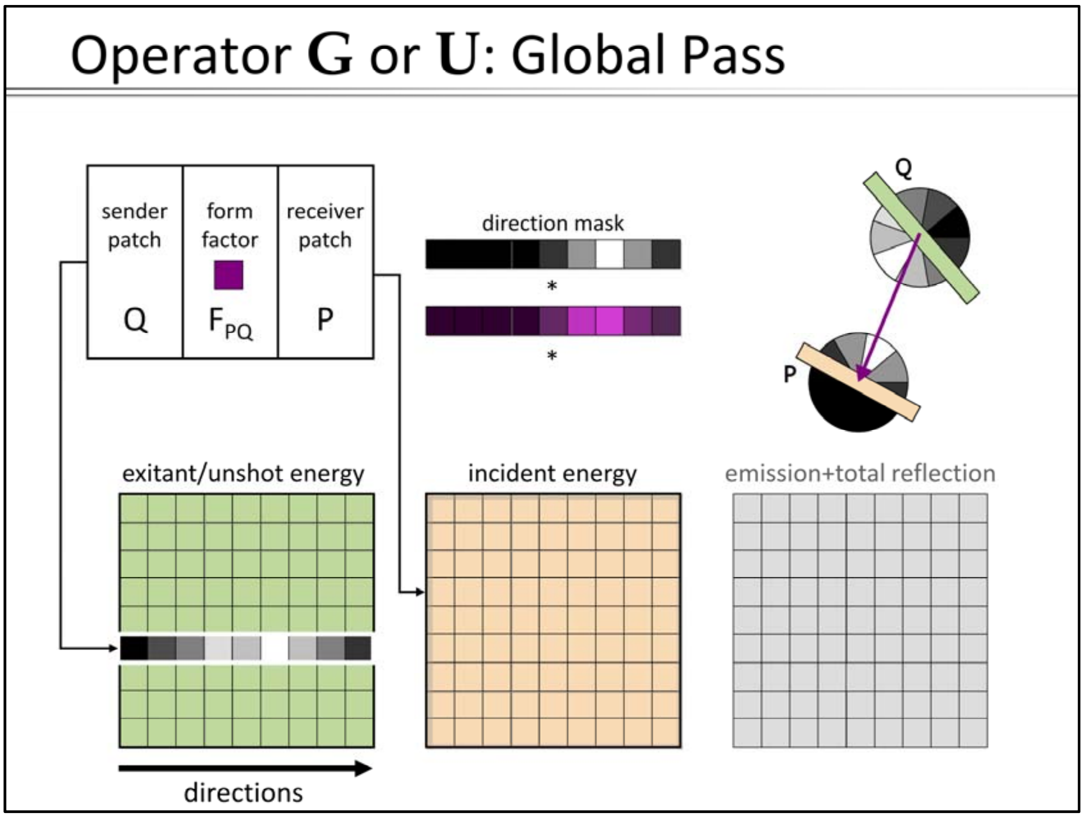
In addition to the spatial discretization we also need to discretize the directions to store the radiance (and possibly antiradiance) values for every patch.

This allows the use of simple 2D arrays to represent the entire radiance distribution in the scene.

And 2D arrays are the perfect data structures for computations on GPUs.

In a shooting style radiosity solver, we store 3 such arrays: exitant and incident energy for every patch and every direction, and we also keep track of the total emitted plus reflected energy of every patch, which defines the brightness of the surfaces used for final display.

It’s possible to store radiance and antiradiance in those tables together, because the two local operators K and J operate and separate hemispheres and thus do not interfere.



Operator $\mathbf{G}$ or $\mathbf{U}$: Global Pass

This slide outlines the computation of an energy propagation given a patch hierarchy and a link list:

First we have a sender patch Q, from which energy is transported to a receiver P.

We look at the distribution of exitant energy of the sender patch.

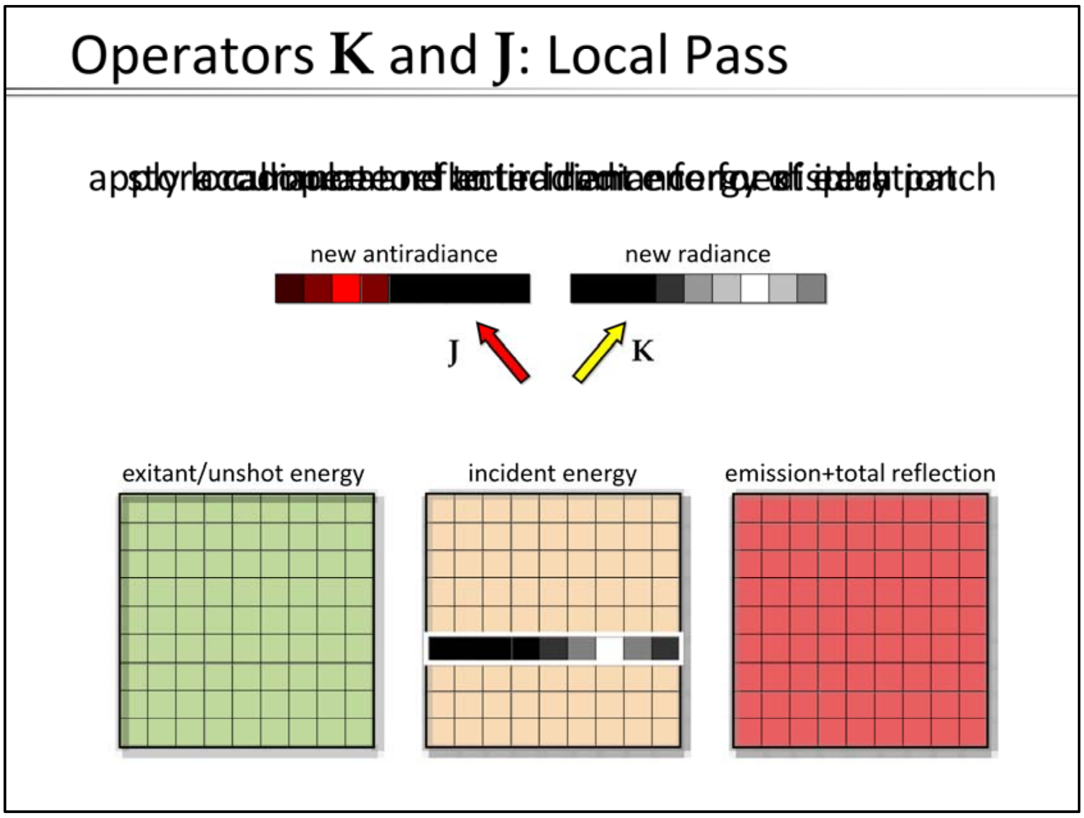
And as this patch only spans a certain solid angle, as seen from the receiver, the energy transport does not affect all directional bins at the receiver.

To account for this, the unaffected bins are masked out using a direction mask, which is shown here.

The amount of transported energy is given by the form factor, which does not contain visibility anymore (when using Dong's method, or antiradiance) and thus can be computed independently of all other patches. It is either precomputed and stored in a table, or computed on the fly.

Finally, these three components are multiplied together and added to the incident energy record of the receiver patch.

All these operations can be easily mapped onto graphics hardware and arranged such that we can perform many propagations in parallel.



Operators $\mathbf{K}$ and $\mathbf{J}$: Local Pass

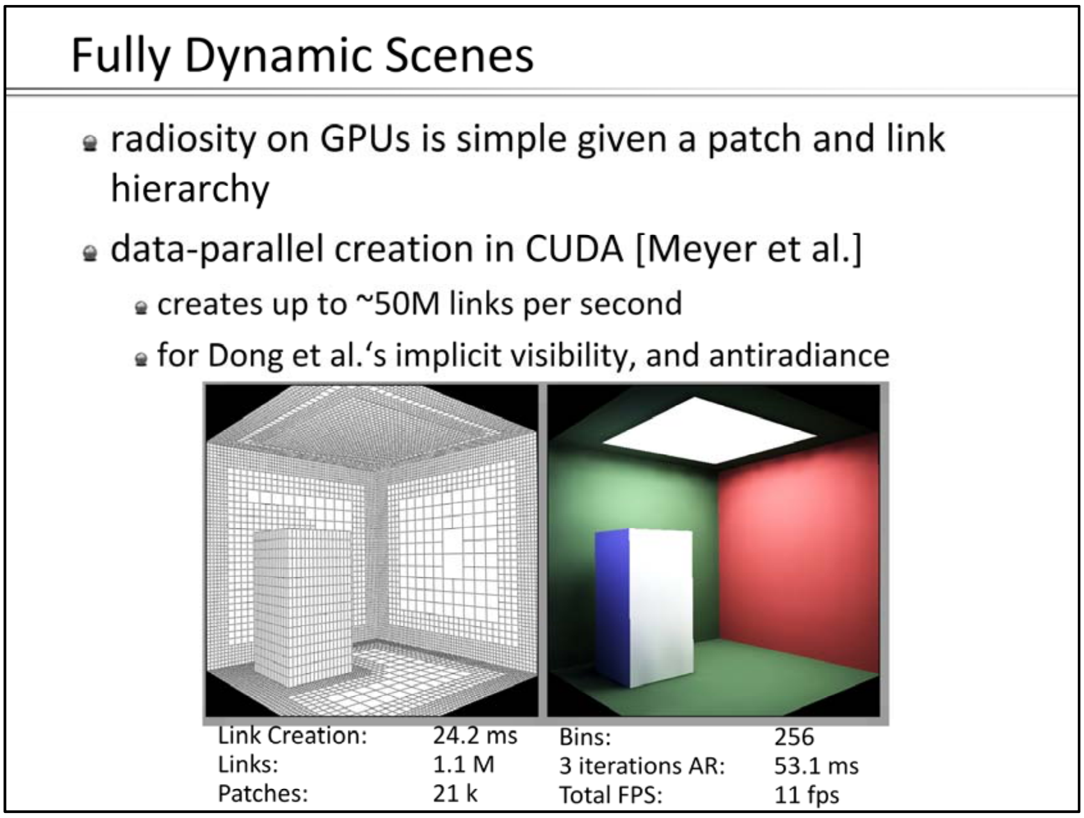
apstlyorlecadliapnecraetoerfslecteiredidiaenticaenfcoergey,dfispelaclhaytpatch

In the local pass, we take the incident energy distribution of each element independently and apply the local operators to it (J of course only when using the antiradiance method).

Thus we get the antiradiance and radiance distribution for the next iteration.

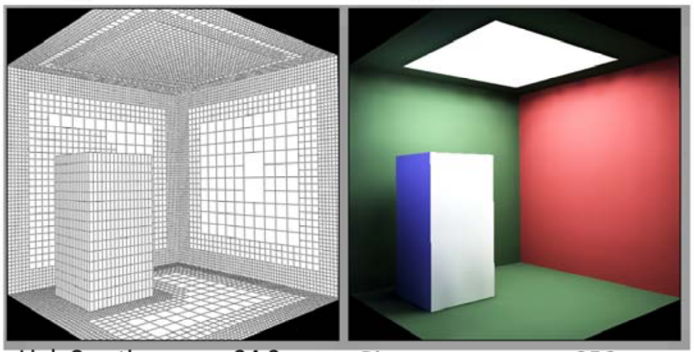
As I mentioned before, K and J operate on separate hemispheres, their results do not interfere and we can store both quantities together in one data structure.

The reflected radiance alone is accumulated in the third data structure and used for final display.



Fully Dynamic Scenes

- radiosity on GPUs is simple given a patch and link hierarchy
- data-parallel creation in CUDA [Meyer et al.]
  - creates up to ~50M links per second
  - for Dong et al.'s implicit visibility, and antiradiance

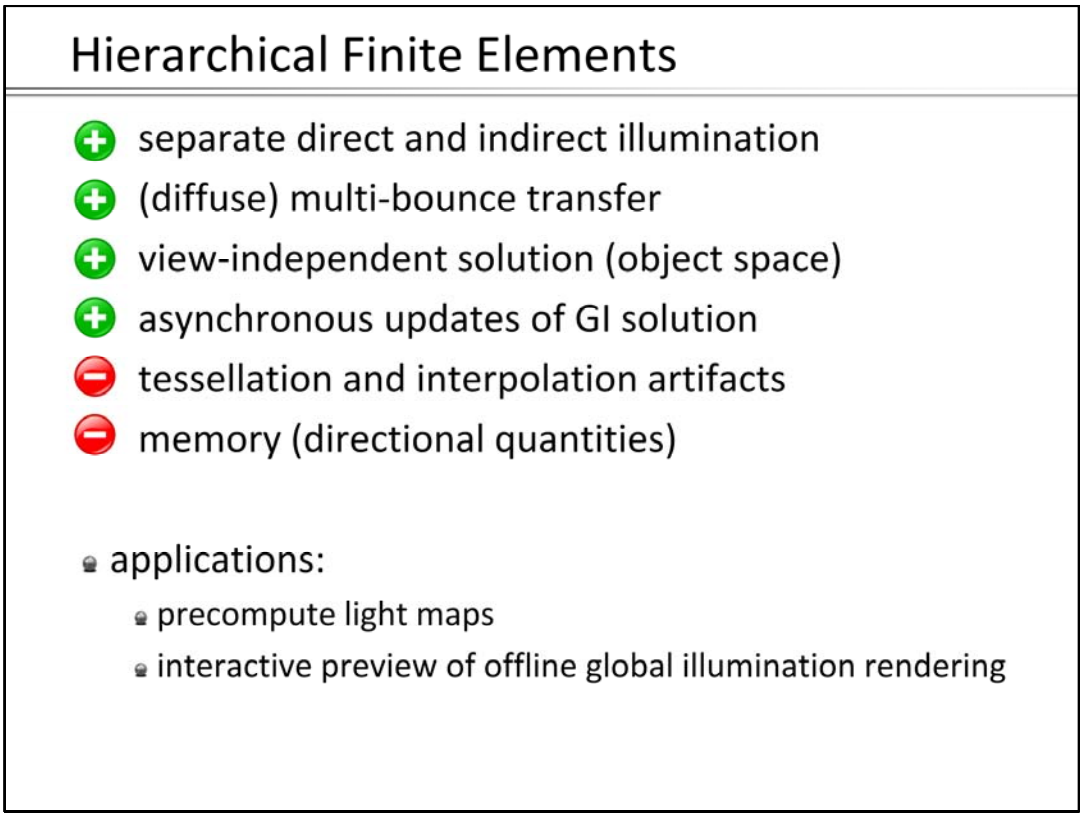


Link Creation:	24.2 ms	Bins:	256
Links:	1.1 M	3 iterations AR:	53.1 ms
Patches:	21 k	Total FPS:	11 fps

Now radiosity on GPUs is simple once you have the patches and links.

Fortunately this problem has already been addressed, and I'd like to point you to recent work, where a data-parallel method for creating link and patch hierarchies, suitable for Dong's implicit visibility and for the antiradiance method, has been demonstrated using a CUDA implementation.

They were able to create 50 mio links per second in fully dynamic scenes.



Hierarchical Finite Elements

- ➕ separate direct and indirect illumination
- ➕ (diffuse) multi-bounce transfer
- ➕ view-independent solution (object space)
- ➕ asynchronous updates of GI solution
- ➖ tessellation and interpolation artifacts
- ➖ memory (directional quantities)

- applications:
  - precompute light maps
  - interactive preview of offline global illumination rendering

good about the finite element techniques is, that you can compute the indirect illumation at a low-resolution, and combine this with any direct lighting technique, e.g. soft-shadow mapping.

Radiosity is particularly well working for scenes with multiple diffuse or moderately glossy bounces.

And a radiosity solution – once computed and stored - is also view-independent and can be reused until the scene geometry or lighting changes.

If the performance is not good enough, then an asynchronous or incremental update of the indirect illumination might still be an option for your application.

However, there are also inherent difficulties: with any finite element representation you always bump into the problem of interpolating the solution, and you will often suffer from light leaking and other artifacts.

And finally, memory is an issue: you have to store light information for every patch, and if you use antiradiance or glossy materials then the memory consumption will be quite high in complex scenes and you need to store a directional radiance distribution and not just a single radiosity value per patch.

So – are finite element methods useful at all?

I mainly see two possible applications at the moment: they might be interesting for precomputing lighting and storing it in lightmaps, and second, radiosity on GPUs might be an option for interactively rendering previews of off-line global illumination renderers, e.g. Monte-Carlo ray tracing.