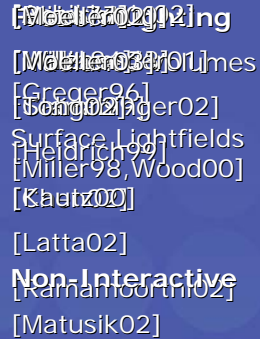




**SIGGRAPH**2004

## **Precomputed Radiance Transfer**

Jan Kautz, Massachusetts Institute of Technology



# Motivation – What we want



SIGGRAPH2004

- What we want:
  - Illuminate objects with environment maps
  - Change lighting on-the-fly
  - Include self-shadowing and interreflections
  - In real-time



No Shadows



Shadows

## Background – Spherical Harmonics



- Spherical Harmonics  $y_i(\vec{s})$ :
  - Orthonormal basis over the sphere
  - Analogous to Fourier transform over 1D circle
- Projection:  $f_i = \int_{\Omega} f(\vec{s}) y_i(\vec{s}) d\vec{s}$
- Reconstruction:  $\tilde{f}(\vec{s}) = \sum_i^N f_i y_i(\vec{s})$ 
  - Note, this is an approximation!

The basis functions we used are the spherical harmonics – they are equivalent to the fourier basis on the plane, but mapped to the sphere.

They have several nice properties:

Since the basis is orthonormal projection is simple, evaluation is also very simple.

## Background – Spherical Harmonics



SIGGRAPH2004

- Important properties:
  - Rotational invariance  $\Rightarrow$  no aliasing artifacts (no wobbling etc...)

- Integration:

$$\int_{\Omega} \tilde{a}(\vec{s}) \tilde{b}(\vec{s}) d\vec{s} = \sum_{i=1}^n a_i b_i$$

- Rotation: linear transform on coefficients (matrix-vector multiplication, will not explain)

The most important property for our application is that they are rotationally invariant. This means that given some lighting environment on the sphere, you can project that lighting environment into SH, rotate the basis functions and integrate them against themselves (ie: rotating the projection and re-projecting) you get identical results to rotating the original lighting environment and projecting it into the SH basis.

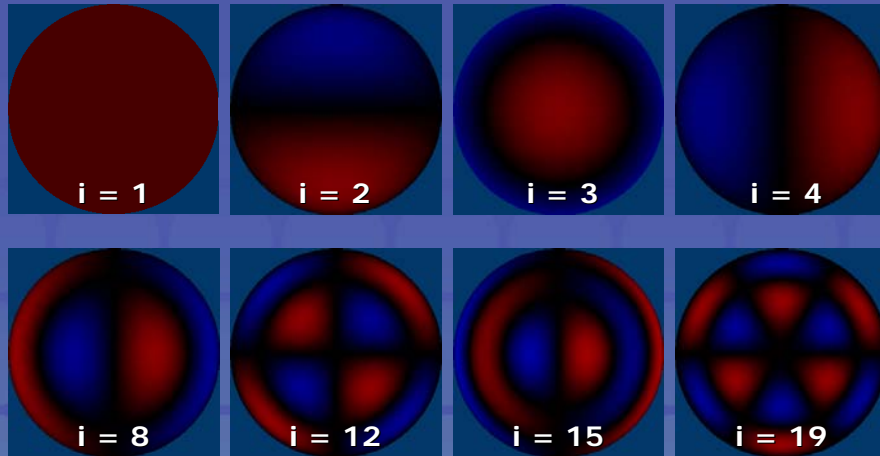
Integration of a product of two functions, which are represented in SH becomes a simple dot-product! Something we will exploit later on.

Rotation is simple (efficient evaluation formulae, just a linear operator on the SH coefs). We won't explain it in this talk.

# Background – Spherical Harmonics



- Basis functions (examples):



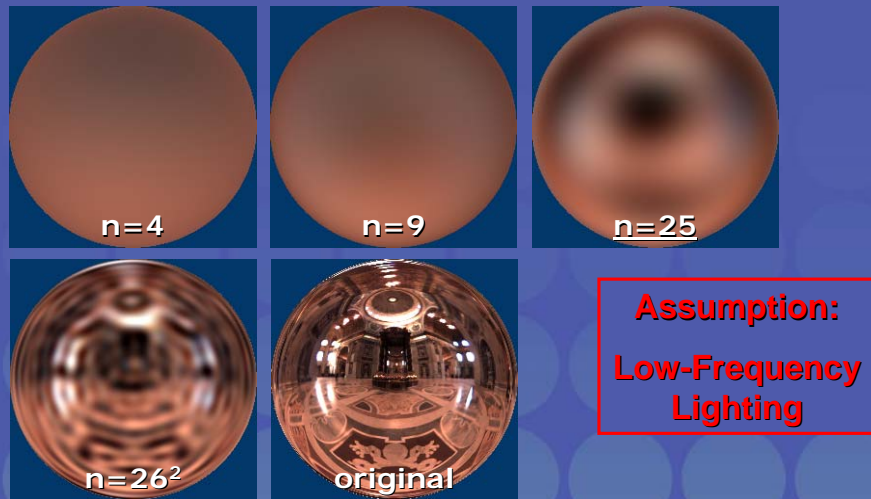
Examples of a few basis functions. They are defined over the sphere. We show them here as sphere maps.

Note, that the higher basis function have higher frequencies too (like Fourier).

# Background – Spherical Harmonics



- Example: projection of environment



Here, we first convert a spherical function (environment map) into SH and then reconstruct the function (formula from before).

Again, the more basis functions are used, the higher frequencies can be represented...

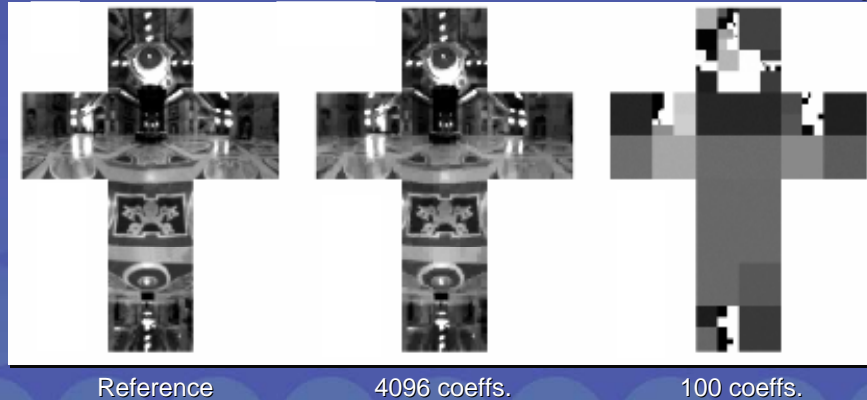
For all examples in this talk, we've used 25 coefficients (assuming low-frequency lighting).

## Background – Haar Wavelets



- Example: projection of environment

Courtesy  
Ren Ng



Alternatively, the Haar wavelet basis functions can be used (instead of SH). We will talk briefly about this at the end of the talk.

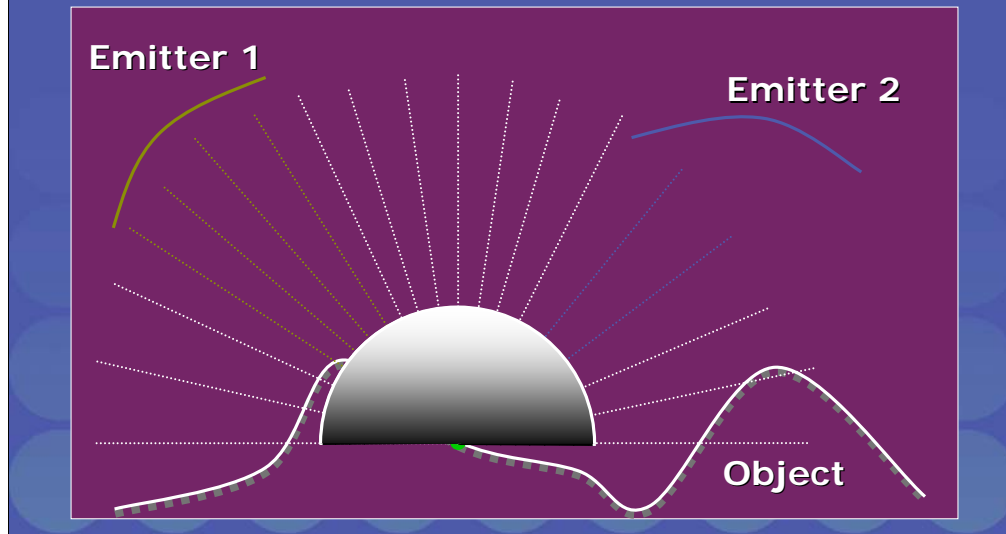
It should be noted, that wavelets are good at representing all-frequency detail (e.g. with 100 coeffs the bright windows are represented well, the not so important darker areas (floor) is represented with less accuracy).



# Global Illumination (Rendering Equation)



- Integrate incident  $\text{light} * V() * \text{diffuse BRDF}$



To compute exit radiance from a point  $p$ , we need to integrate all incident lighting against the visibility function and the diffuse BRDF (dot-product between the normal and the light direction).

# Rendering Equation



SIGGRAPH2004

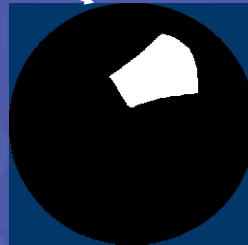
- Math:

$$L_p^{out}(\vec{v}) = \int L_p^{in}(\vec{s}) V_p(\vec{s}) \max(\vec{s} \cdot \vec{n}_p, 0) d\vec{s}$$

Reflected  
Light



Incident Light



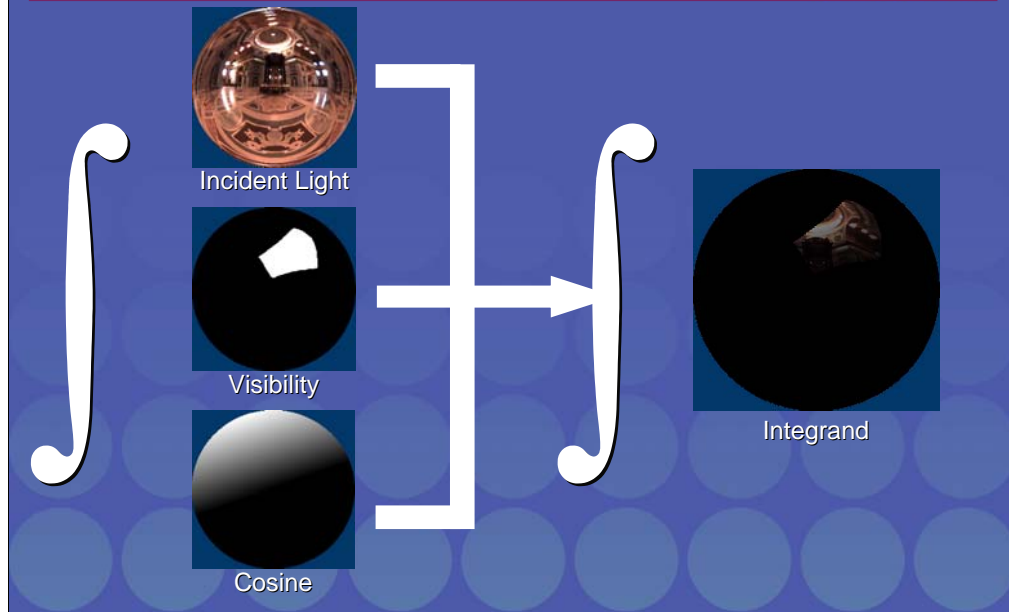
Visibility



Cosine

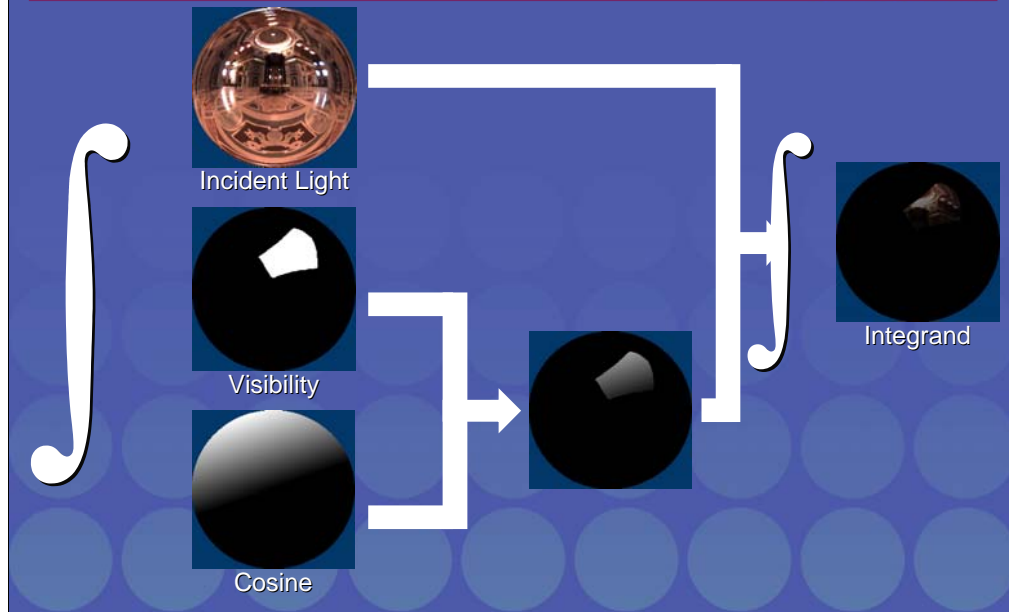
Same thing written down more accurately.

# Rendering Equation – Visually



Visually, we integrate the product of three functions (light, visibility, and cosine).

# Precomputed Radiance Transfer – Visually



The main trick we are going to use for precomputed radiance transfer (*PRT*) is to combine the visibility and the cosine into one function (*cosine-weighted visibility* or *transfer function*), which we integrate against the lighting.

# Problems



SIGGRAPH2004

- Problems remain:
  - How to encode the spherical functions?
  - How to quickly integrate over the sphere?


This is not useful per se. We still need to encode the two spherical functions (lighting, cosine-weighted visibility/transfer function). Furthermore, we need to perform the integration of the product of the two functions quickly.

## Rendering Equation – Rewrite



- Math:

$$L_p^{out}(\vec{v}) = \int L^in(\vec{s}) V(\vec{s}) \max(\vec{s} \cdot \vec{n}_p, 0) d\vec{s}$$

- Rewrite with  $T(\vec{s}) = V(\vec{s}) \max(\vec{s} \cdot \vec{n}_p, 0)$  
- This is the ***transfer function***
  - Encodes:
    - Visibility
    - Shading
    - Implicitly: normal as well (no need to store it)

Using some more math again, we get the transfer function  $T(s)$ .

Note, that this function is defined over the full sphere. It also implicitly encodes the normal at the point  $p$ ! So, for rendering no explicit normal will be needed.

# Rendering Equation – Rewrite





- Math:

$$L_p^{out}(\vec{v}) = \int L^in(\vec{s}) V(\vec{s}) \max(\vec{s} \cdot \vec{n}_p, 0) d\vec{s}$$

- Plug new  $T(\vec{s})$  into Equation:

$$L_p^{out}(\vec{v}) = \int \underbrace{L^in(\vec{s})}_{\text{into SH}} \underbrace{T(\vec{s})}_{\text{into SH}} d\vec{s}$$

 light function:  $L_t^{in}$       transfer:  $T$  

$\Rightarrow$  project *lighting* and *transfer* into SH

Now, when we plug the new  $T(s)$  into the rendering equation, we see that we have an integral of a product of two functions. We remember, that this special case boils down to a dot-product of coefficient vectors, when the two functions are represented in SH.

This is exactly, what we will do. We project the incident lighting and the transfer function into SH.

# Evaluating the Integral



SIGGRAPH2004

- The integral

$$L_p^{out}(\vec{v}) = \int L^{in}(\vec{s}) T(\vec{s}) d\vec{s}$$

becomes

$$L_p^{out}(\vec{v}) = \sum_i^n L_i^{in} T_i$$

"light vector" →  
"transfer vector" →

A *simple* dot-product!!!!

(All examples use n=25 coefficients)

Then the expensive integral becomes a simple product between two coefficient vectors.



# What does this mean?



SIGGRAPH2004

- **Positive:**

- Shadow computation is *independent* of *number* or *size* of light sources!
- Soft shadows are *cheaper* than hard shadows
- Transfer vectors need to be computed (can be done offline)
- Lighting coefficients computed at run-time (3ms)

This has a number of implications:

Shadow computation/shading is independent of the number or the size of the light sources!  
All the lighting is encoded in the lighting vector, which is independent of that.

Rendering this kind of shadows is extremely cheap. It is in fact cheaper than rendering hard shadows!

The transfer vectors can be computed off-line, thus incurring no performance penalty at run-time.

The lighting vector for the incident light can be computed at run-time (fast enough, takes a few milliseconds).

# What does this mean?



SIGGRAPH2004

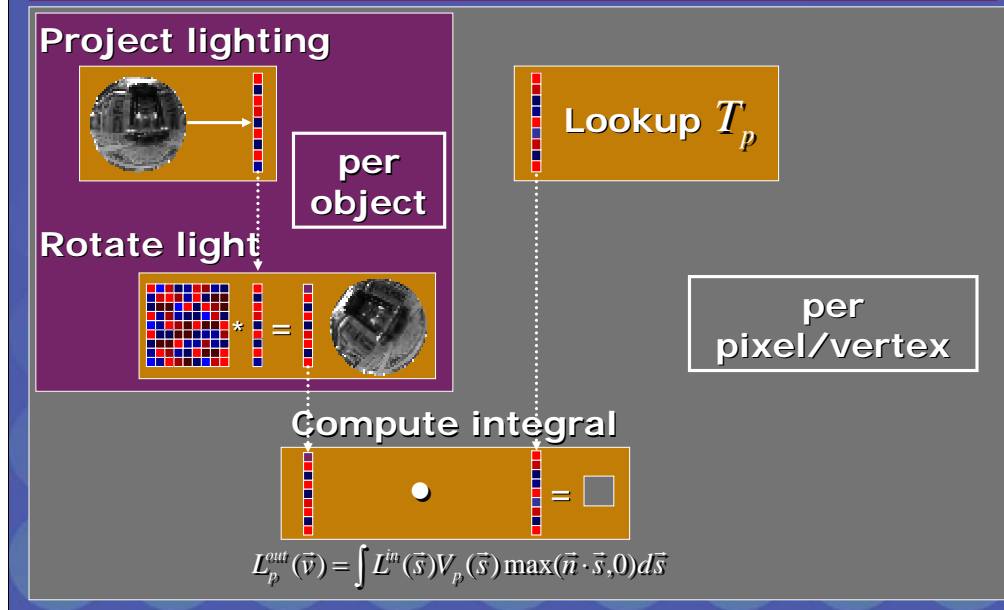
- **Negative:**

- Models are assumed to be *static*
- Assumes all points on surface have *same incident illumination* (no shadows over half the object)

The precomputation of transfer coefficients means that the models have to be static!

Also, there is an implicit assumption, that all points on the surface receive the same incident illumination (environment map assumption). This implies that no half-shadow can be cast over the object (unless, it's part of the object preprocess).

# Precomputed Radiance Transfer



This shows the rendering process.

We project the lighting into SH (integral against basis functions). If the object is rotated wrt. to the lighting, we need to apply the inverse rotation to the lighting vector (using the SH rotation matrix).

At run-time, we need to lookup the transfer vector at every pixel (or vertex, depending on implementation). A (vertex/pixel)-shader then computes the dot-product between the coefficient vectors. The result of this computation is the exitant radiance at that point.

# PRT Results



Unshadowed



Shadowed

# PRT Results



Unshadowed



Shadowed

# PRT Results



SIGGRAPH2004



Unshadowed



Shadowed

# PRT Results



## PRT Results

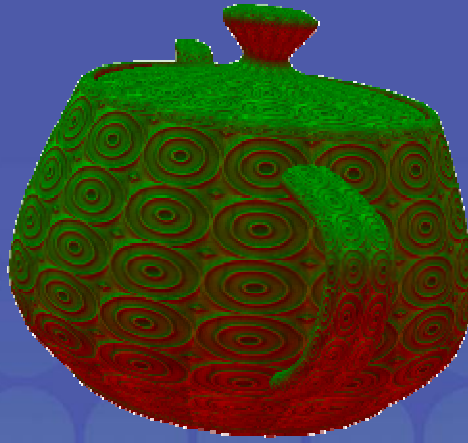


- Diffuse volume: 32x32x32 grid
- Runs 40fps on 2.2Ghz P4, ATI 8500
- Here: dynamic illumination

The same technique can be applied to diffuse volumes. Here we have a transfer vector at each voxel, instead of each pixel/vertex.



## PRT Results



- Bump Map: 128x128 texel
- 50fps on 1.0Ghz Athlon, ATI 8500

Since, the normal is implicitly encoded in the transfer function, we can easily do bump mapping as well...

## More Details



SIGGRAPH2004

- How is precomputation done (for transfer vectors)?
- How is rendering done exactly?
- How can we reduce storage requirements?

Some detail has been lacking so far, which we will explain in the following.

# Precomputation



SIGGRAPH2004

- We need to **project** the transfer function  $T_p(\vec{s})$  into SH (at every point  $p$ ):

$$T_i = \int T_p(\vec{s}) y_i(\vec{s}) d\vec{s}$$

$$\text{with: } T_p(\vec{s}) = \frac{\rho}{\pi} V_p(\vec{s}) \max(\vec{n}_p \cdot \vec{s}, 0)$$

albedo ← visibility ← cosine

- Boils down to: Integral against basis functions

As we've seen before, both the lighting and the transfer function need to be projected into SH. Here we will talk about the projection of the transfer function.

As introduced in the Background-Section, projecting a function into SH boils down to integrating that function against the SH basis functions. This results in a vector of coefficients.

As a reminder: the transfer function is the visibility times the dot-product between the normal  $n$  and the sample direction  $s$  (multiplied by the albedo, which says how reflective the surface is).

# Precomputation



- Integral

$$T_i = \frac{\rho}{\pi} \int V_p(\vec{s}) \max(\vec{n}_p \cdot \vec{s}, 0) y_i(\vec{s}) d\vec{s}$$

evaluated numerically with ray-tracing:

$$T_i = \frac{4\pi}{N} \frac{\rho}{\pi} \sum_{j=0}^{N-1} V_p(\vec{s}_j) \max(\vec{n}_p \cdot \vec{s}_j, 0) y_i(\vec{s}_j)$$

- Directions  $\vec{s}_j$  need to be uniformly distributed (e.g. random)
- Visibility  $V_p$  is determined with ray-tracing

The main question is how to evaluate the integral. We will evaluate it numerically using Monte-Carlo integration. This basically means, that we generate a random (and uniform) set of directions  $\vec{s}_j$ , which we use to sample the integrand. All the contributions are then summed up and weighted by  $4\pi/(\text{\#samples})$ .

The visibility  $V_p()$  needs to be computed at every point. The easiest way to do this, is to use ray-tracing.

-----  
Aisde: uniform random directions can be generated the following way.

- 1) Generate random points in the 2D unit square (x,y)
- 2) These are mapped onto the sphere with:

$$\theta = 2 \arccos(\sqrt{1-x})$$

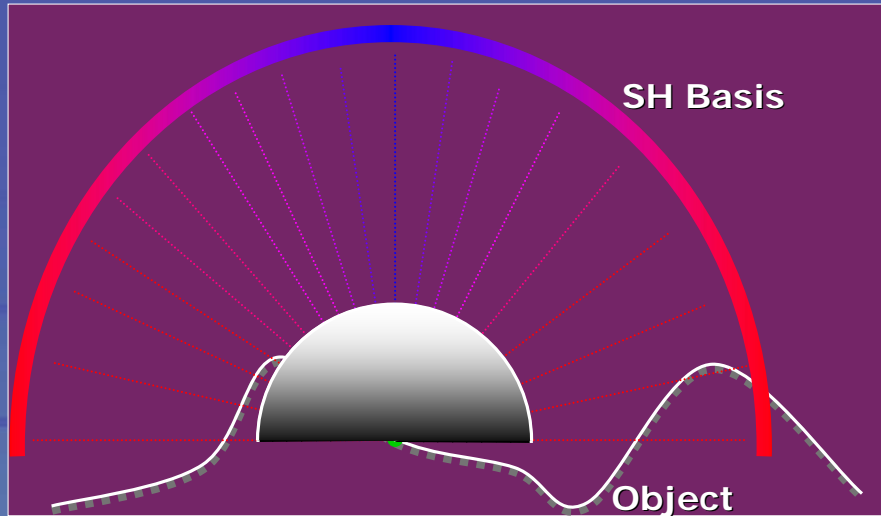
$$\phi = 2y\pi$$

## Precomputation – Visually



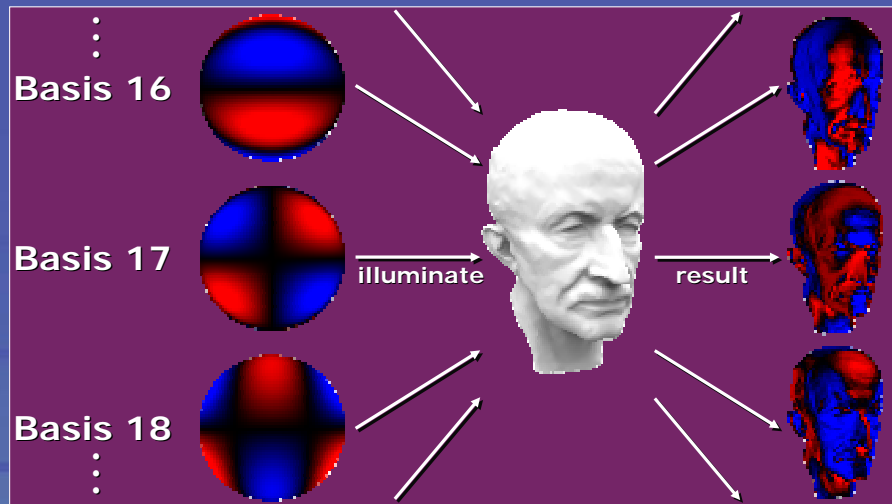
SIGGRAPH2004

- Integrate  $V_p(\vec{s}) \cdot \max(\vec{n}_p \cdot \vec{s}, 0) \cdot y_i(\vec{s})$



Visual explanation 1).

## Precomputation – Visually



Visual explanation 2):

This slide illustrates the precomputation for direct lighting. Each image on the right is generated by placing the head model into a lighting environment that simply consists of the corresponding basis function (SH basis in this case illustrated on the left.) This just requires rendering software that can deal with negative lights.

The result is a spatially varying set of transfer coefficients shown on the right.

To reconstruct reflected radiance just compute a linear combination of the transfer coefficient images scaled by the corresponding coefficient for the lighting environment.

# Precomputation – Code



SIGGRAPH2004

```
// p: current vertex/pixel position
// normal: normal at current position
// sample[j]: sample direction #j (uniformly distributed)
// sample[j].dir: direction
// sample[j].SHcoeff[i]: SH coefficient for basis #i and dir #j

for(j=0; j<numberSamples; ++j) {
    double csn = dotProduct(sample[j].dir, normal);
    if(csn > 0.0f) {
        if(!selfShadow(p, sample[j].dir)) { // are we self-shadowing?
            for(i=0; i<numberCoeff; ++i) {
                value = csn * sample[j].SHcoeff[i]; // multiply with SH coeff.
                result[i] += albedo * value; // and albedo
            }
        }
    }
}

const double factor = 4.0*PI / numberSamples; // ds (for uniform dirs)
for(i=0; i<numberCoeff; ++i)
    Tcoeff[i] = result[i] * factor; // resulting transfer vec.
```

Pseudo-code for the precomputation.

The function `selfShadow( p, sample[j].dir )` traces a ray from position `p` in direction `sample[j].dir`. It returns true if there it hits the object, and false otherwise.

## Precomputation – Bump Map



- Basically: no difference!
- We use per-pixel normal instead of interpolated normal...
- Self-shadowed bump maps are possible:
  - selfShadow() needs to account for that
- Again: run-time does **not** change

Bump mapping is really easy to incorporate (if transfer vectors are stored in a texture).

The precomputation algorithm from the previous slide remains the same. Only the normal needs to be looked up from a bump map in the precomputation phase!

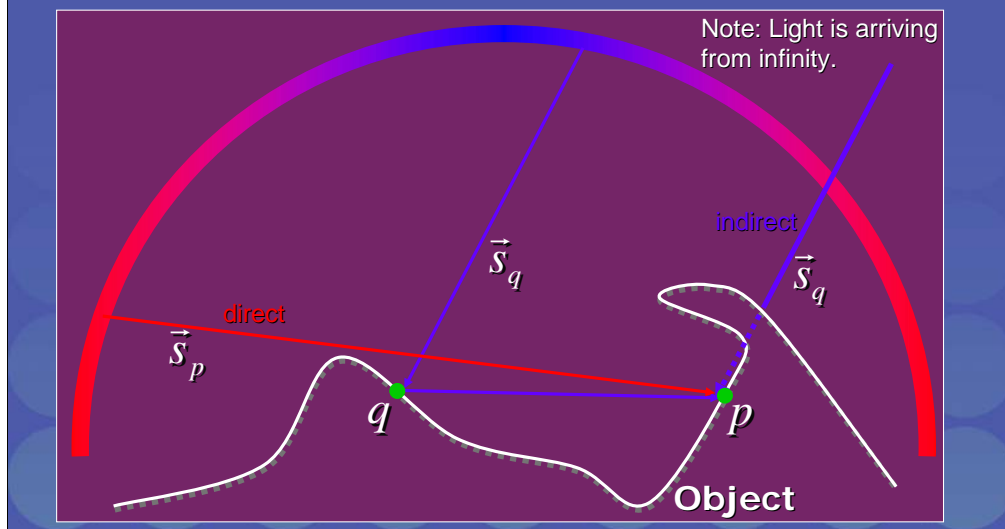
If the bump map is too incorporate shadows from the bumps, then the selfShadow() function needs to be augmented to test for intersections with the bump map (e.g. convert to a heightfield for intersection tests).



## Precomputation – Interreflections



- Light can interreflect from positions  $q$  onto  $p$



Not only shadows can be included into PRT, but also interreflections.

Light arriving at a point  $q$  can be subsequently scattered onto a point  $p$ . I.e. light arriving from  $s_q$  can arrive at  $p$ , although there is may be no direct path (along  $s_q$ ) to  $p$  (as in this example).

Note, that light is arriving from infinity, so both shown direction  $s_q$  originate from the same point in infinity.

## Precomputation – Interreflections



- Light can interreflect from positions  $q$ , where there is self-shadowing (no direct visibility of sphere!):

$$L_p^{out}(\vec{v}) = L^{DS} + \int L_q^{out}(\vec{s}) \underbrace{(1 - V_p(\vec{s}))}_{\text{inverse visibility}} \underbrace{\max(\vec{s} \cdot \vec{n}_p, 0)}_{\text{cosine}} d\vec{s}$$

direct illumination      light leaving from  $q$  towards  $p$

More formally, we do not only have direct illumination  $L^{DS}$  (Direct Shadowed), but also light arriving from directions  $s$ , where there is self-shadowing (i.e.  $1 - V_p(s)$ ). The light arrives from positions  $q$ , which are the first hit along  $s$ .

## Precomputation – Interreflections



- Precomputation of transfer vector has to be changed
- An additional bounce  $b$  is computed with

$$T_{p,i}^b = \frac{\rho_p}{\pi} \int T_{q,i}^{b-1} (1 - V_p(\vec{s})) \max(\vec{n}_p \cdot \vec{s}, 0) d\vec{s}$$

where  $T_{p,i}^0$  is from the pure shadow pass

- Final transfer vector:  $T_{p,i} = \sum_{b=0}^{B-1} T_{p,i}^b$

***Run-time remains the same!***

To account for interreflections, the precomputation has to be changed again.

Each additional bounce  $b$  generates a vector  $T^b_{p,i}$ , which is computed as shown on the slide. Each of these additional transfer vectors is for a certain bounce.

To get the final transfer vector, they have to be added. Again, the run-time remains the same!

## Interreflections – Results



SIGGRAPH2004



No Shadows/Inter



Shadows



Shadows+Inter

This set of images shows the buddha model lit in the same lighting environment, without shadows, with shadows and with shadows and inter reflections.

# Rendering



- Reminder:

$$L_p^{out}(\vec{v}) = \sum_i^n L_i^{in} T_i$$

- Need lighting coefficient vector:

$$L_i = \int L^{in}(\vec{s}) y_i(\vec{s}) d\vec{s}$$

- Compute every frame (if lighting changes)
- Projection can e.g. be done using Monte-Carlo integration (see before)

Rendering is just the dot-product between the coefficient vectors of the light and the transfer.

The lighting coefficient vector is computed as the integral of the lighting against the basis functions (see slides about transfer coefficient computation).

# Rendering



SIGGRAPH2004

- Work that has to be done per-vertex is very simple:

```
// No color bleeding, i.e. transfer vector is valid for all 3 channels

for(j=0; j<numberVertices; ++j) { // for each vertex
    for(i=0; i<numberCoeff; ++i) {
        vertex[j].red   += Tcoeff[i] * lightingR[i]; // multiply transfer
        vertex[j].green += Tcoeff[i] * lightingG[i]; // coefficients with
        vertex[j].blue  += Tcoeff[i] * lightingB[i]; // lighting coeffs.
    }
}
```

- Only shadows: independent of color channels  $\Rightarrow$  single transfer vector
- Interreflections: color bleeding  $\Rightarrow$  3 vectors

Sofar, the transfer coefficient could be single-channel only (given that the 3-channel albedo is multiplied onto the result later on). If there are interreflections, color bleeding will happen and the albedo cannot be factored outside the precomputation. This makes 3-channel transfer vectors necessary, see next slide.

- In case of interreflections (and color bleeding):

```
// Color bleeding, need 3 transfer vectors

for(j=0; j<numberVertices; ++j) {    // for each vertex
    for(i=0; i<numberCoeff; ++i) {
        vertex[j].red   += TcoeffR[i] * lightingR[i]; // multiply transfer
        vertex[j].green += TcoeffG[i] * lightingG[i]; // coefficients with
        vertex[j].blue  += TcoeffB[i] * lightingB[i]; // lighting coeffs.
    }
}
```

# Extensions



SIGGRAPH2004

- Reducing Storage Cost
- Animated Objects
- Other Materials
- Other Basis Functions (not SH)

In the following we talk about 4 extensions.

- 1) How to reduce the storage cost
- 2) How to incorporate animated objects
- 3) Other materials (other than diffuse)
- 4) Use of other basis functions



# Reducing Storage Cost



SIGGRAPH2004

- Original method:
  - Store transfer vector at each pixel/vertex
  - Each transfer vector has 25 coefficients (can be 8bit if scaled/biased appropriately)
  - Original paper suggest to just store 24 coeffs. Pack into 6 textures
  - For a 256x256 texture, that's 1.5MB
- Reduce storage somehow

The original storage cost is fairly high (25 coefficients per texel).

# Reducing Storage Cost



SIGGRAPH2004

- Reduction:
  - Don't need high-resolution texture – shadows are very smooth (comparable to light maps)
  - Use standard texture compression
- Better Reduction:
  - Use Principle Component Analysis (PCA)
  - Use Vector Quantization (VQ)
  - Use combination (Clustered PCA)

Standard texture compression can work, but there are better techniques.

## Reducing Storage Cost – PCA



- PCA:
  - Take all  $T_p$  and run PCA on it
  - Result:  $T_p \approx T^0 + \sum_{k=1}^K w_p^k T^k$
  - Where:  $w_p^k$  are weights that change per pixel
  - And:  $T^k$  are basis vectors
  - Since original transfer vectors  $T_p$  can be very different, a high  $K$  is needed for good quality

PCA (Principle Component Analysis):

Plug all the vectors  $T_p$  (for all  $p$ ) into PCA, and you will get above result/approximation.

The quality of the approximation depends on the number of basis vectors. If the variation in the  $T_p$  is high, a high number of basis vectors is needed.

The beauty of this is, that now we only need to store weights per pixel (hopefully far less than there are original coefficients). Unfortunately, this might not be the case, rendering pure PCA not very useful (or only limited to certain objects, where  $T_p$  is well-approximated with less than 25 basis vectors).

## Reducing Storage Cost – VQ



- VQ:
  - Take all  $T_p$  and run VQ on it
  - Result:
    - Each  $T_p$  gets a unique  $T^k$  associated
    - Store  $k$  at each pixel, instead of actual vector
  - Number of  $T^k$  is much smaller than the number of  $T_p$
- Problem:
  - Indices are stored in textures
    - Cannot do mip-mapping on them!
  - Visible quantization artifacts

VQ:

We plug all  $T_p$  into VQ and get a set of representative  $T^k$  back. Each  $T_p$  is uniquely associated with a  $T^k$ , but different  $T_p$  may map to the same  $T^k$  (quantization).

Now we only need to store the index at each pixel, instead of the actual vector.

Storage is hereby reduced, as the codebook (containing the  $T^k$ ) is usually much smaller than the number of original vectors  $T_p$ .

Unfortunately, quantization artifacts may be visible and mip-mapping is not possible anymore.

## Reducing Storage Cost – VQ+PCA



- VQ+PCA:
  - Take all  $T_p$  and run VQ on it
  - For each VQ transfer vector  $T^k$ 
    - Find all  $T_p$  that map to it (cluster)
    - Do PCA only on those
      - Works much better than before, since the vectors in a cluster are very similar (because of VQ)
  - Works well, but complicates rendering a bit
  - But also makes it faster (few needed dot-products can be computed on CPU, GPU only does weighted sum for PCA)

The best compression technique combines both algorithms.

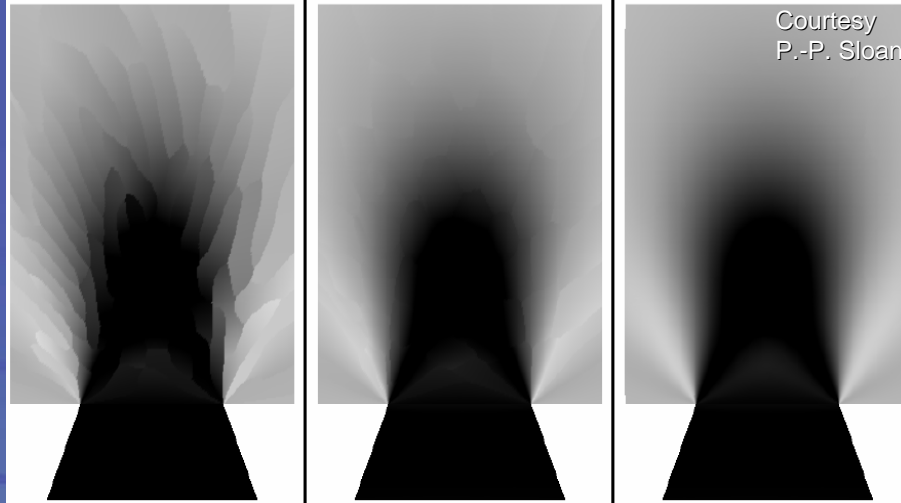
First run VQ. All the  $T_p$  that map to the same  $T^k$  are considered a cluster. Within that cluster PCA is used, which now works much better, because all  $T_p$  of a cluster are similar (due to VQ).

Rendering is slightly different now. The dot-products between  $T_p$  and lighting vector  $L$  is now:  $T_p \text{ dot } L = \text{SUM}(w_k T_k) \text{ dot } L = \text{SUM}(w_k (T^k \text{ dot } L)) = \text{SUM}(w_k R^k)$

$R^k$  can be computed on the CPU, since the  $T^k$  don't change per-pixel, but only the weights. Only the  $\text{SUM}(w_k \wedge R^k)$  is then done on the GPU.

Attention has to be paid to the different areas of an object, that belong to different clusters (they have different basis vectors).

# Compression – Comparison



Courtesy  
P.-P. Sloan

SH Order 10, VQ+PCA  
(2 PCA Vectors, 64 clus.)

SH Order 10, VQ+PCA  
(3 PCA vectors, 64 clus.)

SH Order 10, VQ+PCA  
(4 PCA vectors, 64 clus.)

Left: VQ producing 64 clusters. 2 PCA Vectors (mean + first) - Quantization artifacts are still visible (varies only linearly within a cluster)

Middle: VQ producing 64 clusters. 3 PCA Vectors (mean + first, second) - Artifacts almost gone (variation is now bilinear)

Right: VQ producing 64 clusters. 4 PCA Vectors (mean + first, second, third) - Visually as good as original (variation is trilinear within cluster)

# Animated Objects



SIGGRAPH2004

- So far, objects were assumed *static*
- This is ok for e.g. architectural walkthroughs, but not for games
- Need to extend PRT to animated models
- Two kinds of models:
  - Key-framed
  - Fully dynamic

Assumption so far was: static models (because of precomputation).

# Animated Objects



SIGGRAPH2004

- Key-framed animations:
  - Compute PRT solution for (sub-set) of key-frames
  - Interpolate in-between (coefficients can be interpolated)
- Problems:
  - Very high storage costs
    - Solution: Do VQ+PCA on all precomputed frames
  - High computation cost (PRT for each frame)
    - Solution: None really

Key-framed objects can be easily used with PRT. Just precompute transfer coefficient for every key-frame and interpolate coefficients within key-frames (as the rest is interpolated as well).

Storage cost can be decrease by using the same compression technique as before.

The precomputation becomes a lot more expensive... (Nothing really to prevent this).



# Animated Objects



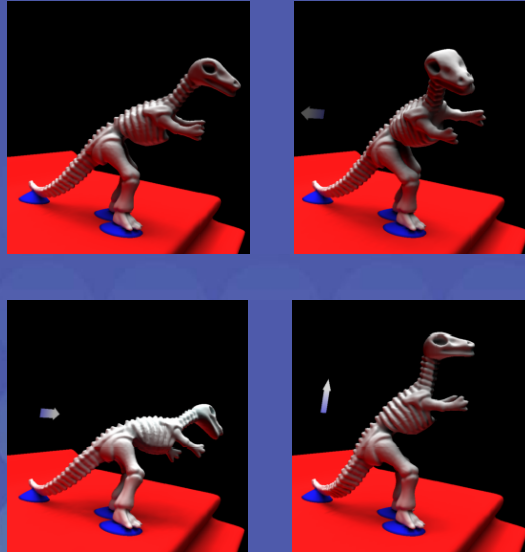
SIGGRAPH2004

- Fully dynamic models:
  - No individual frames, for which PRT could be precomputed
  - Would need to do everything on-the-fly
  - ***No solution yet!***

Fully dynamic models:

No solution in sight. Visibility changes radically (and in unknown ways). Computation of transfer vectors would need to be done on-the-fly.

## Example: Animation with PRT



Courtesy  
Doug James

Example of a key-framed animation.

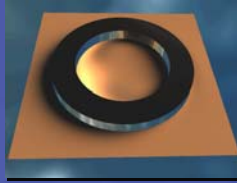
## Other Effects / Materials



SIGGRAPH2004

- The following effects can be incorporated easily:

- Caustics:



- Subsurface-Scattering:



Courtesy  
P.-P. Sloan

- View-dependent effects require transfer matrices (see original paper)
  - E.g., glossy materials

Other effects that can be incorporated with an enhance pre-process: caustics and subsurface-scattering.

View-dependent effects require transfer matrices and not vectors (see original PRT paper).

## PRT with glossy BRDFs



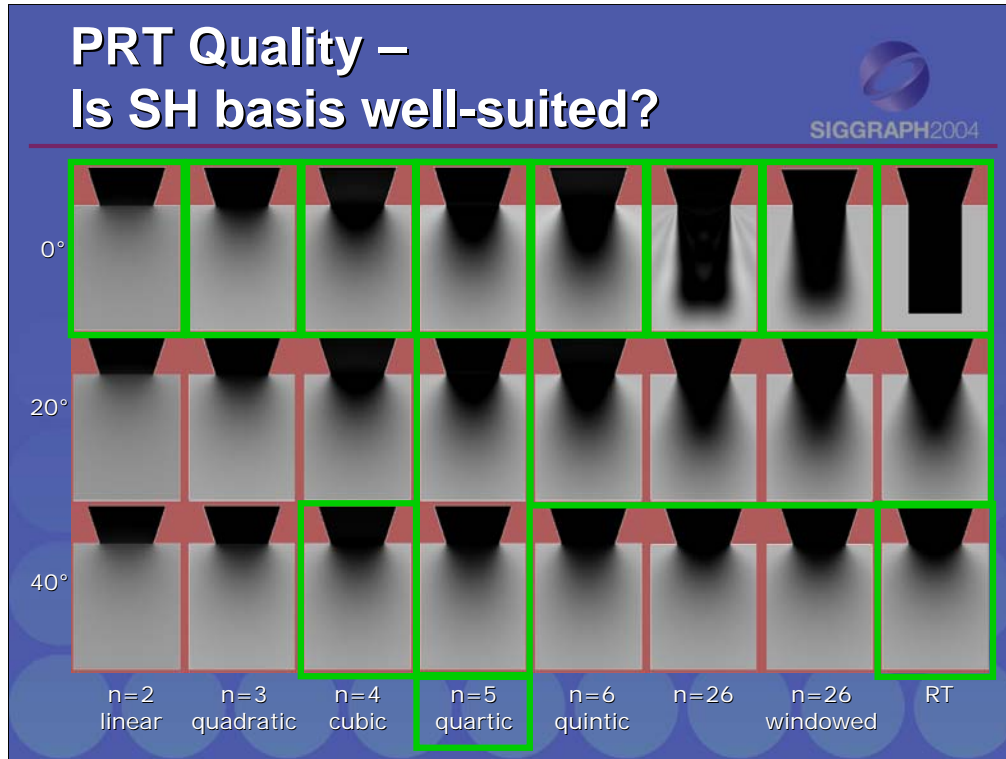
SIGGRAPH2004



Phong



Measured Vinyl



Quality of SH solution.

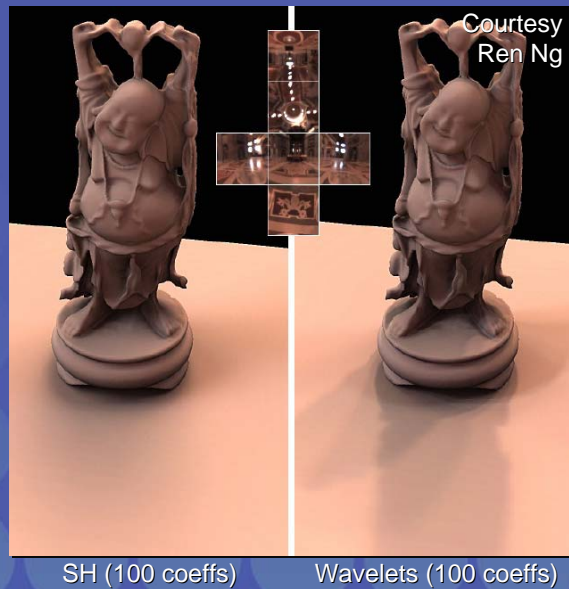
0 degree (point light) source, 20 degree light source, 40 degree light-source.

Light is blocked by a blocker casting a shadow onto the receiver plane. Different order of SH is shown (order<sup>2</sup> = number of basis functions). Very right: exact solution.

As stated before, lighting is assumed low-frequency, i.e. point light doesn't work well, but large area lights do!

## Other Basis Functions

- SH produce very soft shadows
- Alternative: Haar Wavelets [Ng et al. 02]
- Quality improves quite a bit
- Idea remains the same:
  - Per-pixel dot-product to compute shading

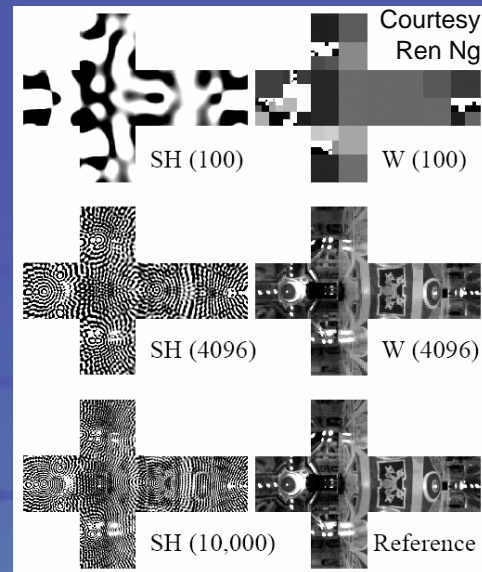


As we've noticed SH make shadows smooth (few coefficients are used, hence the low-frequency lighting assumption). Even if more are used (say 100), still low-frequency.

Alternative: Use Haar Wavelets. Quality improves quite a bit. Basic algorithm remains the same.

# PRT with Haar Wavelets

- Main difference to SH:
  - Haar needs to precompute *all* lighting/transfer coefficients!
  - Decide depending on lighting, which ones to use! (see right)
  - Implies (compressed) storage of all transport coefficients ( $64 \times 64 \times 6$ )
  - Not well-suited to hardware rendering



As shown in the comparison on the right, with more coefficients, wavelets do much better represent the lighting than the SH (which show a lot of ringing artifacts).

There are a few differences when using Haar instead of SH:

- 1) All transfer coefficients need to be computed!
- 2) Because the actual  $N$  coefficients used, is decided at run-time based on the lighting's most important  $N$  coefficients ( $N=100$  seems sufficient).
- 3) This requires all transfer coefficients to be stored as well (can be compressed well, like lossy wavelet compressed images).
- 4) Since the coefficients to be used change at run-time, this is not well-suited to a GPU implementation (but works fine on CPU)

# Conclusions



SIGGRAPH2004

## Pros:

- Fast, arbitrary dynamic lighting
- PRT: includes shadows and interreflections

## Cons:

- Works only well for low-frequency lighting
- Animated models are difficult to handle





- Thanks to:
  - ATI & NVIDIA for hardware donations
  - Paul Debevec for HDR environments

Thank you!