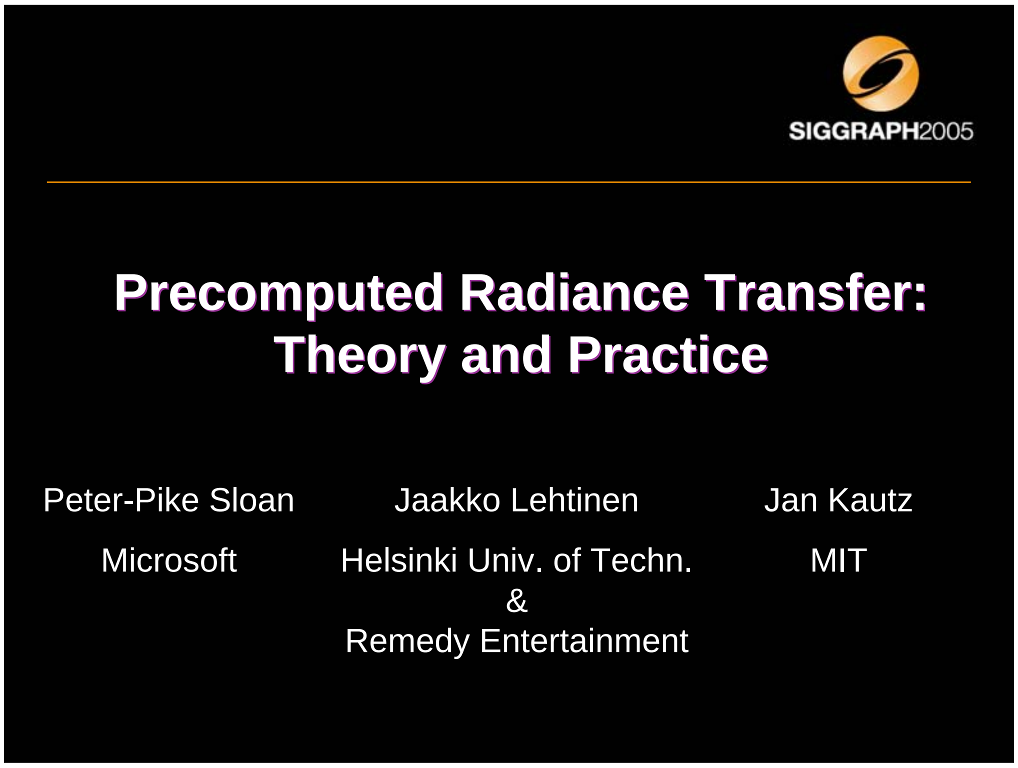
SIGGRAPH2005



SIGGRAPH2005

Precomputed Radiance Transfer: Theory and Practice

Peter-Pike Sloan — Microsoft

Jaakko Lehtinen — Helsinki Univ. of Techn. & Remedy Entertainment

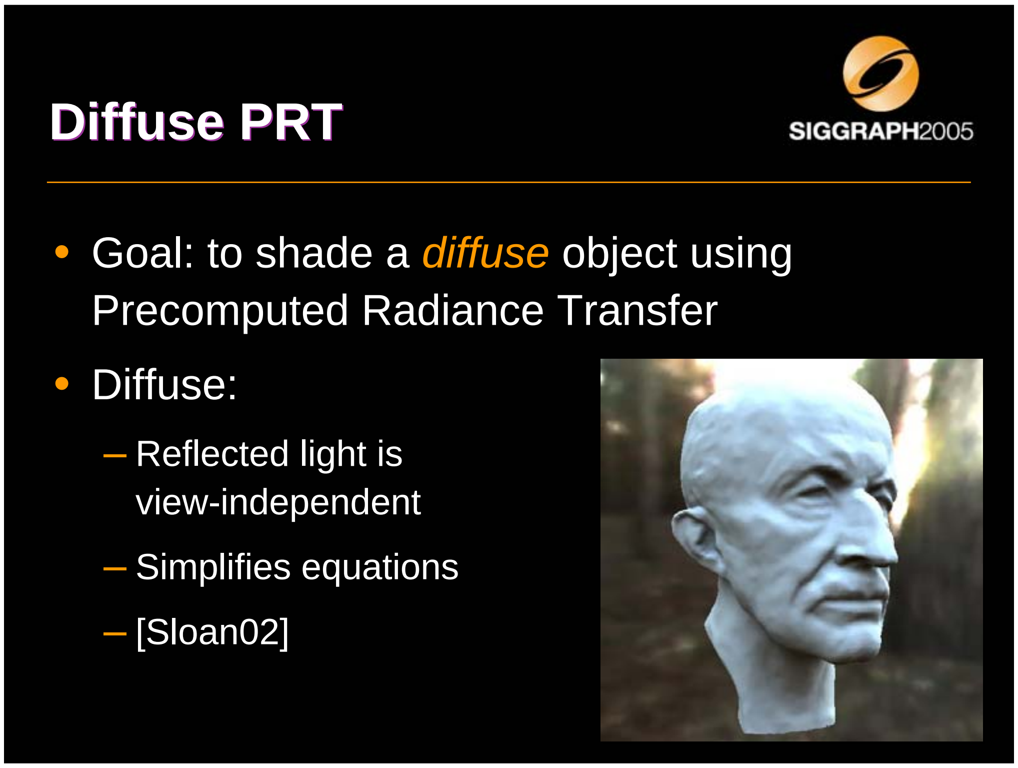
Jan Kautz — MIT

SIGGRAPH2005

Diffuse PRT

Jan Kautz

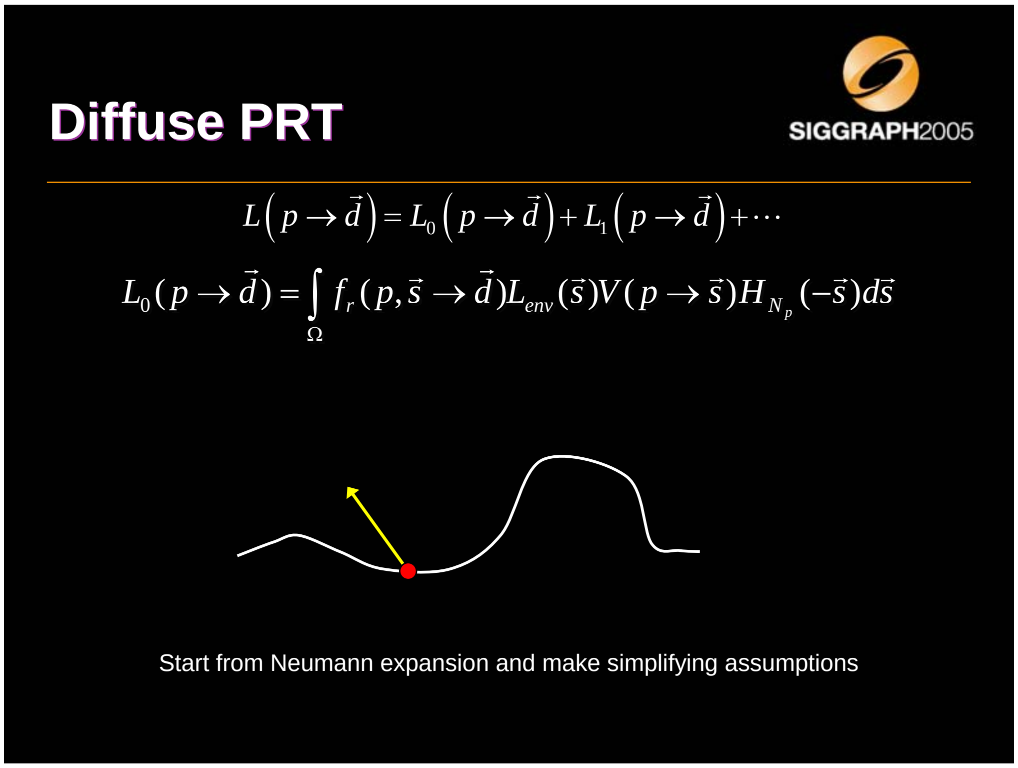
MIT



Diffuse PRT

SIGGRAPH2005

- Goal: to shade a *diffuse* object using Precomputed Radiance Transfer
- Diffuse:
  - Reflected light is view-independent
  - Simplifies equations
  - [Sloan02]

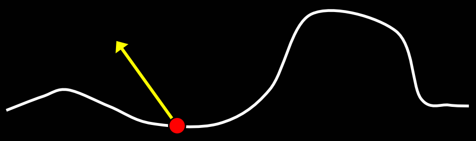


Diffuse PRT

SIGGRAPH2005

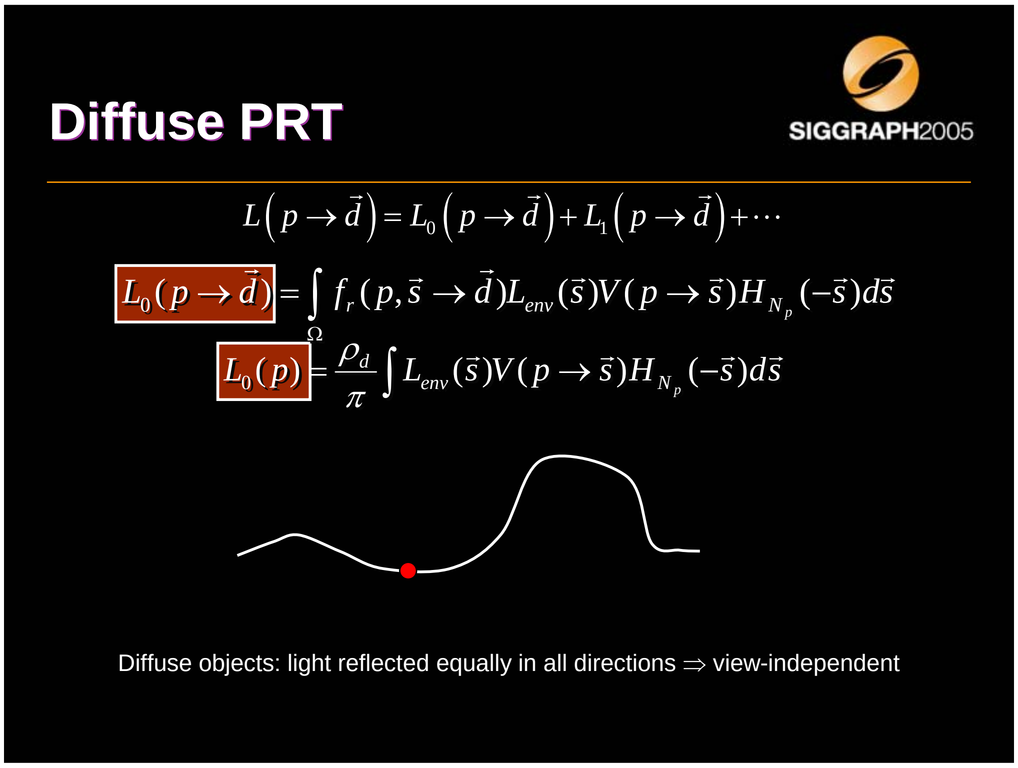
$$L\left(p \to \vec{d}\right) = L_0\left(p \to \vec{d}\right) + L_1\left(p \to \vec{d}\right) + \cdots$$

$$L_0(p \to \vec{d}) = \int_\Omega f_r(p, \vec{s} \to \vec{d}) L_{env}(\vec{s}) V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$



Start from Neumann expansion and make simplifying assumptions

To derive PRT for the diffuse case we are going to start with just the direct term from the Neumann expansion of the rendering equation and make several simplifying assumptions.

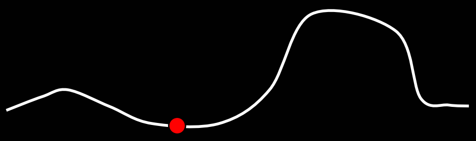


Diffuse PRT

$$L\left(p \to \vec{d}\right) = L_0\left(p \to \vec{d}\right) + L_1\left(p \to \vec{d}\right) + \cdots$$

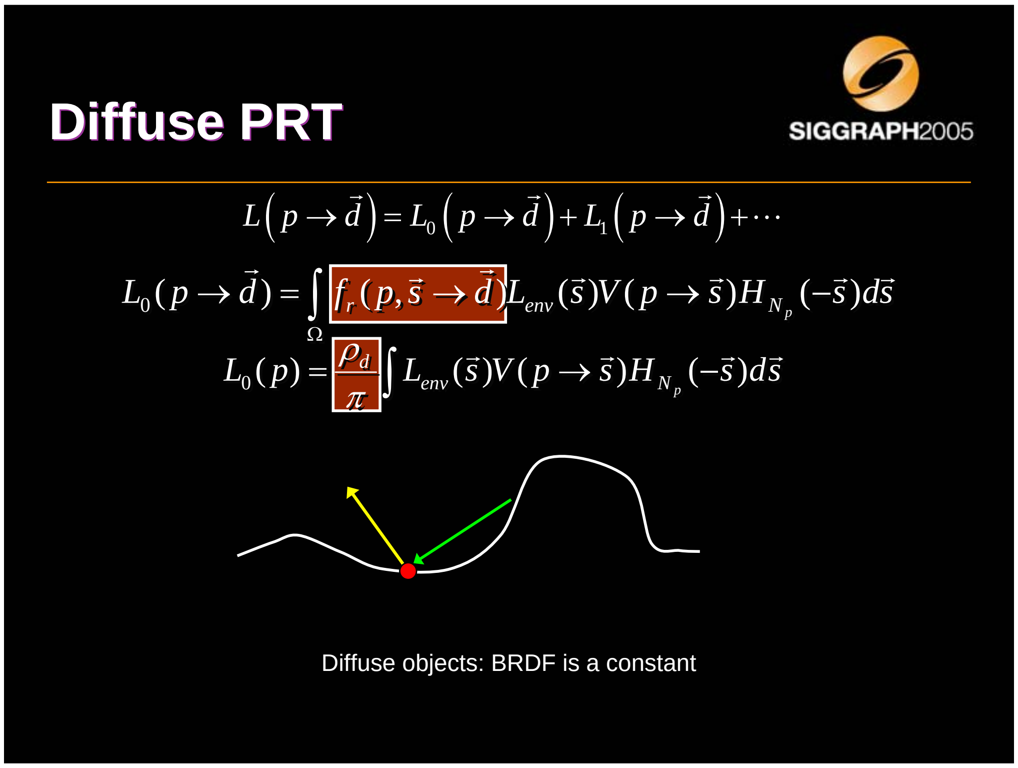
$$\boxed{L_0(p \to \vec{d})} = \int_{\Omega} f_r(p, \vec{s} \to \vec{d}) L_{env}(\vec{s}) V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$

$$\boxed{L_0(p)} = \frac{\rho_d}{\pi} \int L_{env}(\vec{s}) V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$



Diffuse objects: light reflected equally in all directions $\Rightarrow$ view-independent

The bottom equation is the “simplified” form. First, for diffuse objects light is reflected equally in all directions, so outgoing radiance is independent of view direction.

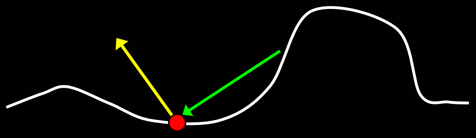


Diffuse PRT

$$L\left(p \to \vec{d}\right) = L_0\left(p \to \vec{d}\right) + L_1\left(p \to \vec{d}\right) + \cdots$$

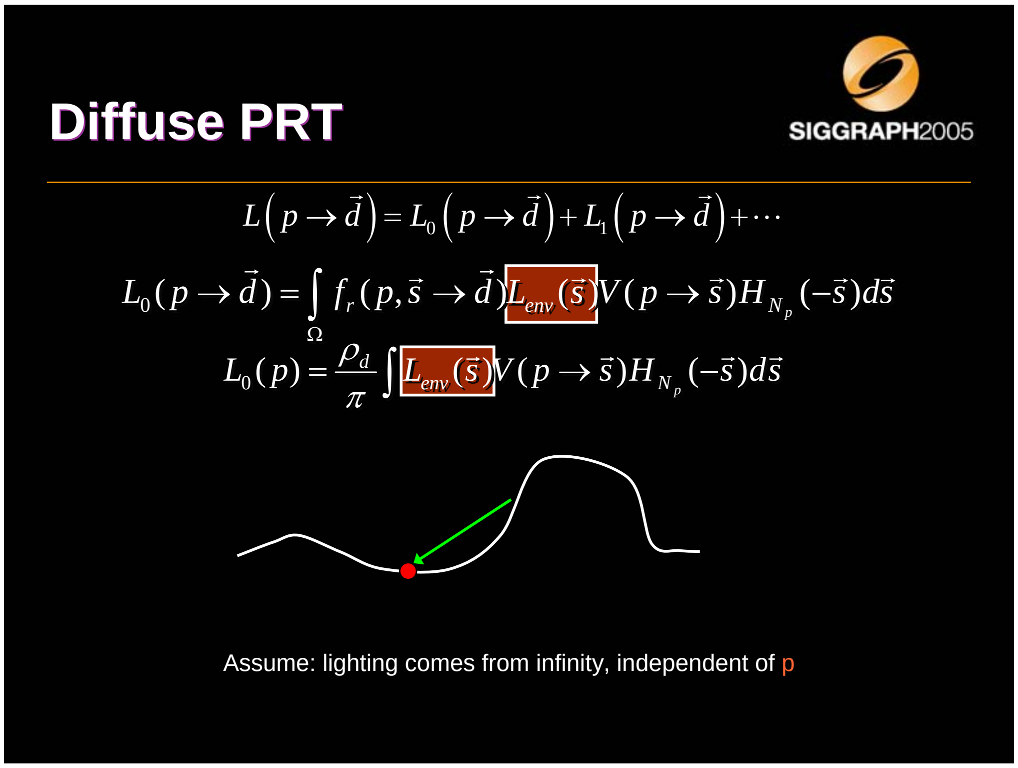
$$L_0(p \to \vec{d}) = \int_{\Omega} \boxed{f_r(p, \vec{s} \to \vec{d})} L_{env}(\vec{s}) V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$

$$L_0(p) = \boxed{\frac{\rho_d}{\pi}} \int L_{env}(\vec{s}) V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$



Diffuse objects: BRDF is a constant

This also means the BRDF is just a constant (and independent of direction) so it can be pulled out of the integral. Rho_d represents the diffuse reflectivity of the surface, and is a number between 0 and 1. The divide by Pi enforces energy conservation.



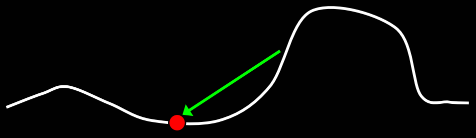
Diffuse PRT

SIGGRAPH2005

$$L\left(p \to \vec{d}\right) = L_0\left(p \to \vec{d}\right) + L_1\left(p \to \vec{d}\right) + \cdots$$

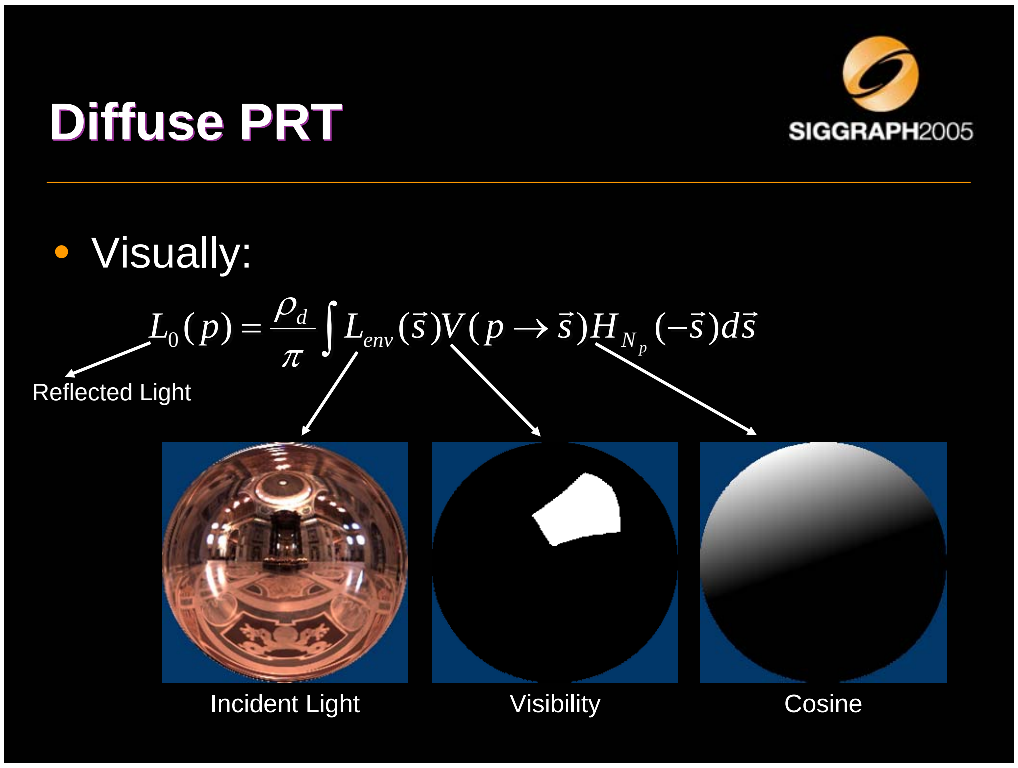
$$L_0(p \to \vec{d}) = \int_\Omega f_r(p, \vec{s} \to \vec{d}) \boxed{L_{env}(\vec{s})} V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$

$$L_0(p) = \frac{\rho_d}{\pi} \int \boxed{L_{env}(\vec{s})} V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$



Assume: lighting comes from infinity, independent of p

As before, we assume the source radiance function is at infinity, this means we only need to concern ourselves with the direction s.



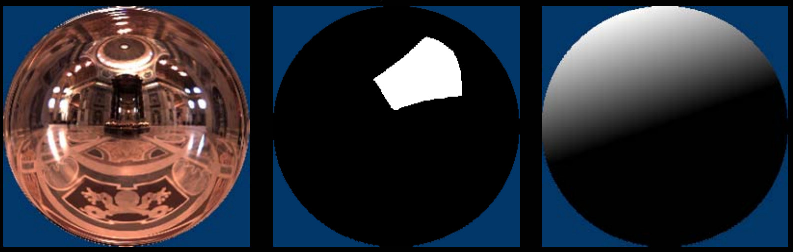
Diffuse PRT

SIGGRAPH2005

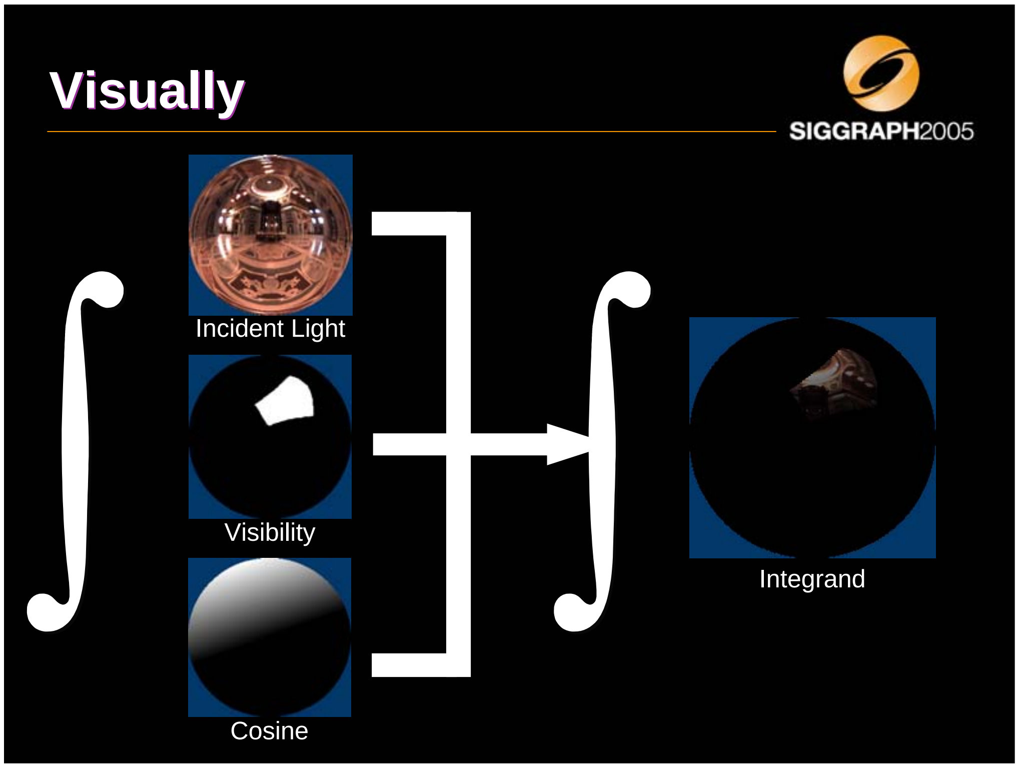
- Visually:

$$L_0(p) = \frac{\rho_d}{\pi} \int L_{env}(\vec{s}) V(p \to \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$

Reflected Light



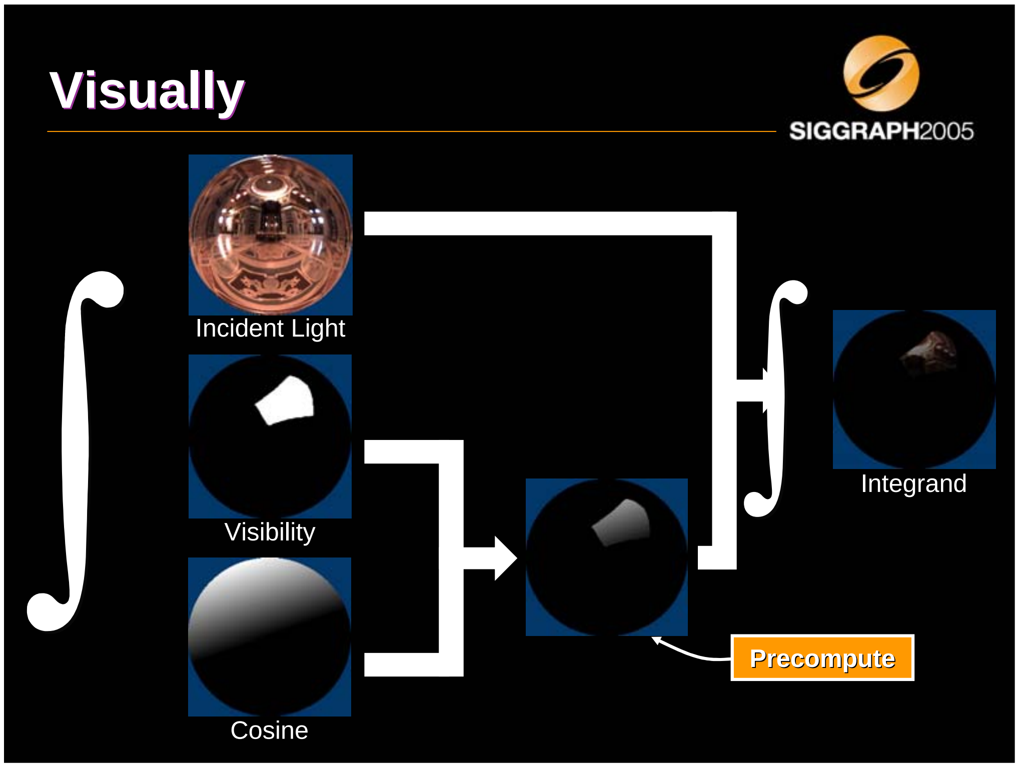
Incident Light Visibility Cosine



Visually

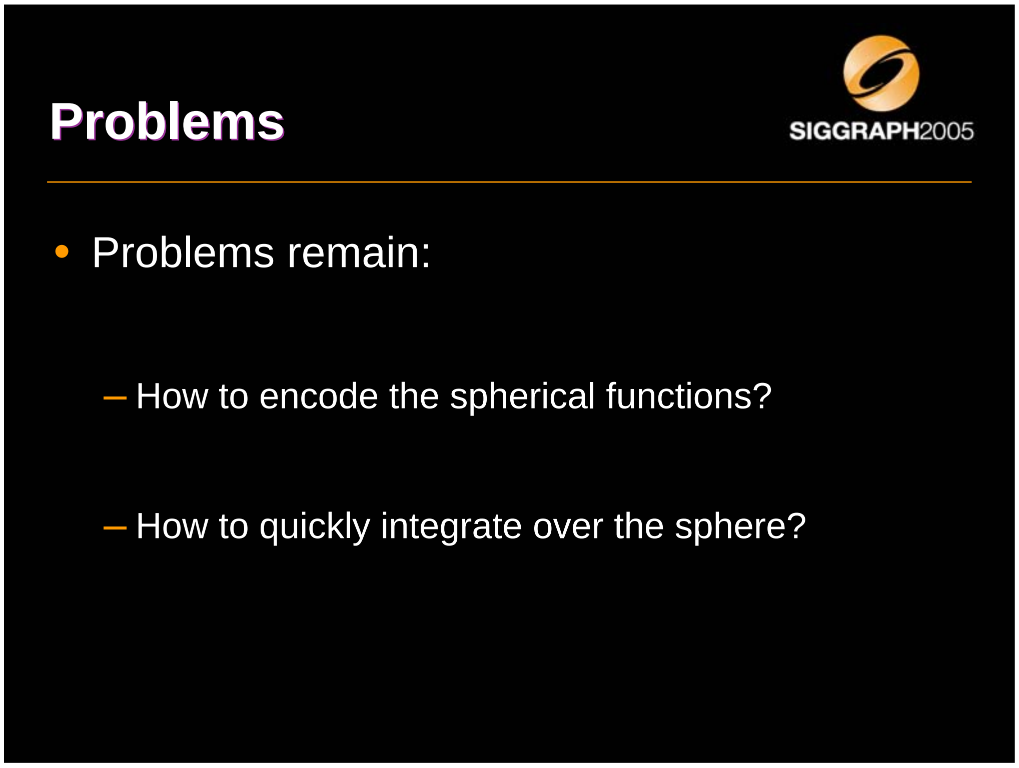
SIGGRAPH2005

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



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.

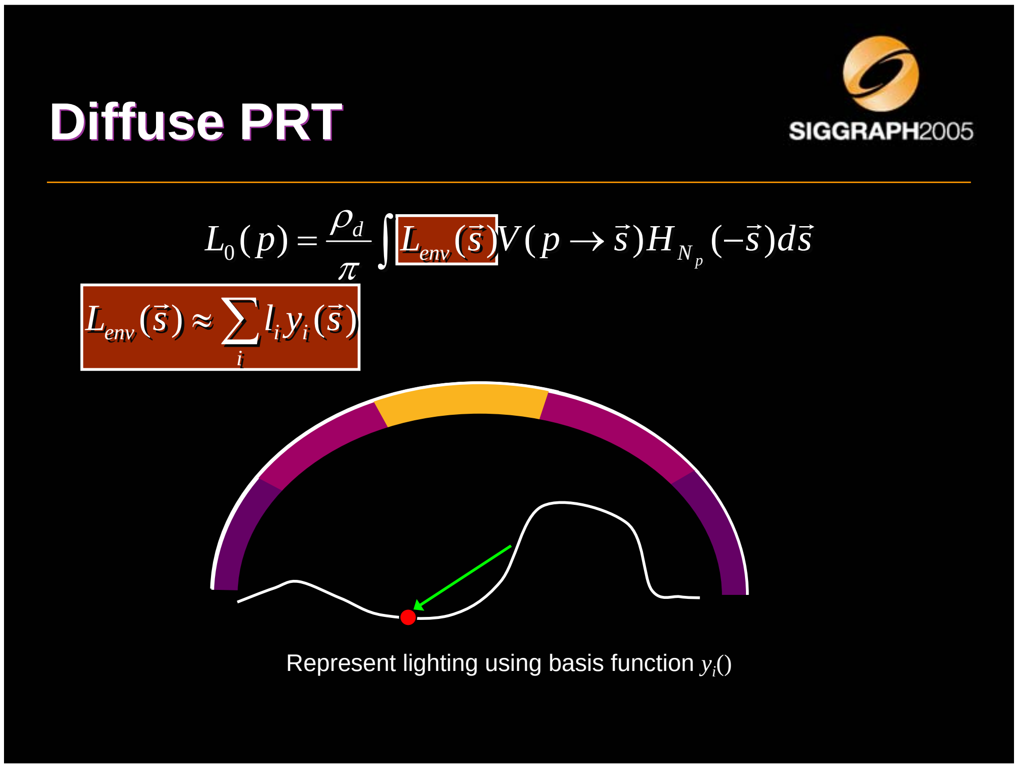


Problems

SIGGRAPH2005

- 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.

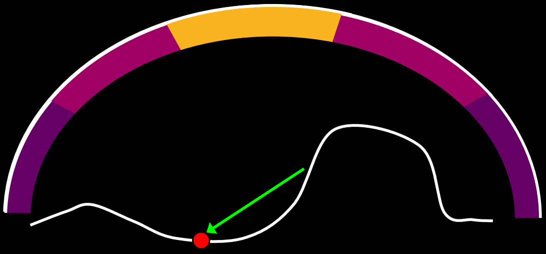


Diffuse PRT

SIGGRAPH2005

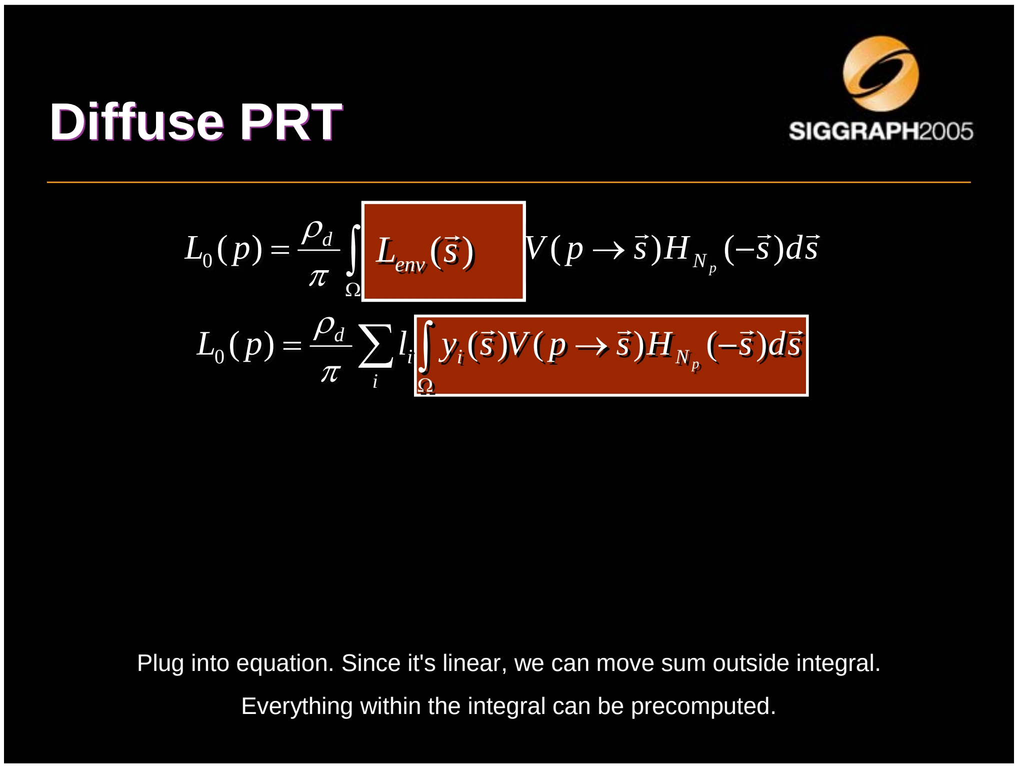
$$L_0(p) = \frac{\rho_d}{\pi} \int \boxed{L_{env}(\vec{s})} V(p \rightarrow \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$

$$\boxed{L_{env}(\vec{s}) \approx \sum_i l_i y_i(\vec{s})}$$



Represent lighting using basis function $y_i()$

Now we are going to approximate the source radiance function with its projection into a set of basis functions on the sphere (denoted $y_i()$ in this equation.) The l_i are the projection coefficients of a particular lighting environment. For didactic purposes we are using piecewise constant basis functions.



Diffuse PRT

SIGGRAPH2005

$$L_0(p) = \frac{\rho_d}{\pi} \int_\Omega \boxed{L_{env}(\vec{s})} V(p \rightarrow \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}$$

$$L_0(p) = \frac{\rho_d}{\pi} \sum_i l_i \boxed{\int_\Omega y_i(\vec{s}) V(p \rightarrow \vec{s}) H_{N_p}(-\vec{s}) d\vec{s}}$$

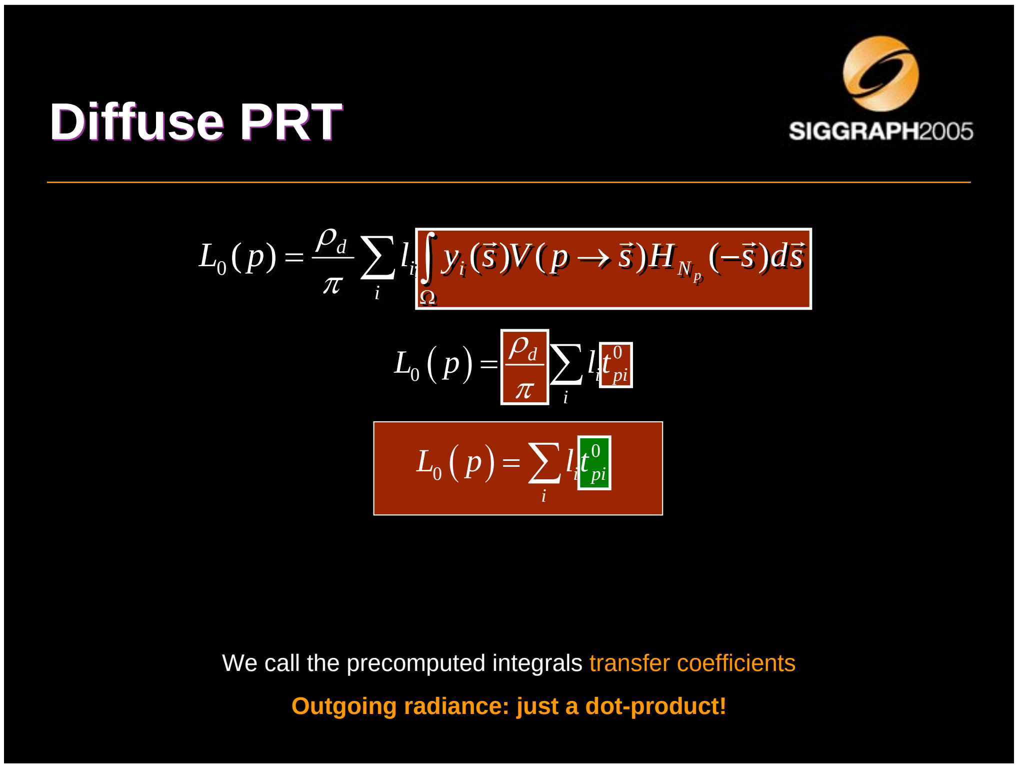
Plug into equation. Since it's linear, we can move sum outside integral.

Everything within the integral can be precomputed.

We can plug this approximation directly into the reflected radiance equation.

Manipulating this expression exploiting the fact that integration is a linear operator (sum of integrals = integral of sums), we can generate the following equivalent expression.

The important thing to note about the highlighted integral is that it is independent of the actual lighting environment being used, so it can be precomputed.



Diffuse PRT

SIGGRAPH2005

$$L_0(p) = \frac{\rho_d}{\pi}\sum_i l_i \boxed{\int_\Omega y_i(\vec{s})V(p\to\vec{s})H_{N_p}(-\vec{s})d\vec{s}}$$

$$L_0(p) = \boxed{\frac{\rho_d}{\pi}}\sum_i l_i \boxed{t^0_{pi}}$$

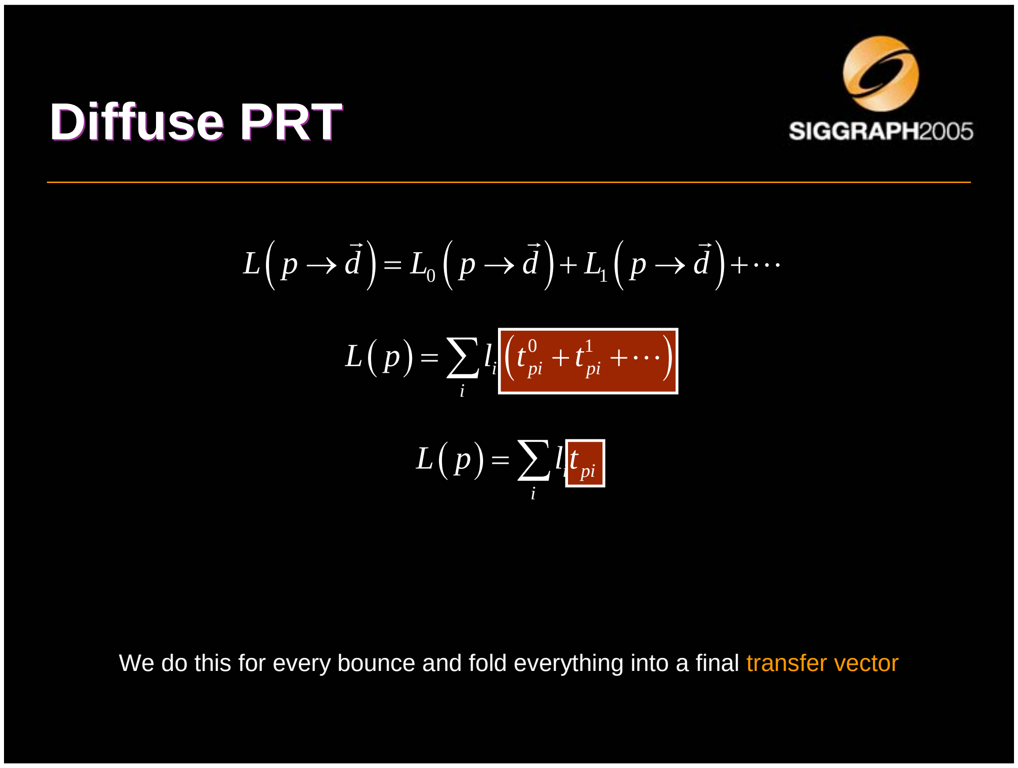
$$\boxed{L_0(p) = \sum_i l_i \boxed{t^0_{pi}}}$$

We call the precomputed integrals transfer coefficients

Outgoing radiance: just a dot-product!

This integral represents a transfer coefficient – it maps how direct lighting in basis function I becomes outgoing radiance at point p. The set of transfer coefficients is a transfer vector that maps lighting into outgoing radiance.

We can optionally fold the diffuse reflectivity into the transfer vector as well.



Diffuse PRT

SIGGRAPH2005

$$L\left(p \to \vec{d}\right) = L_0\left(p \to \vec{d}\right) + L_1\left(p \to \vec{d}\right) + \cdots$$

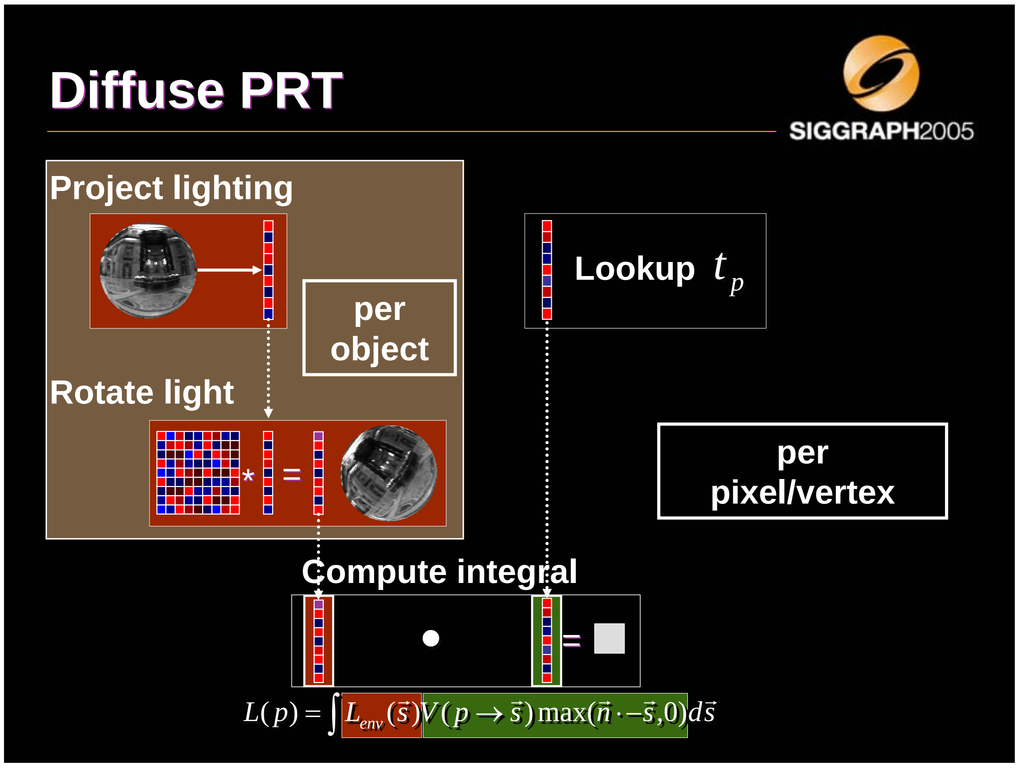
$$L(p) = \sum_i l_i \boxed{\left(t_{pi}^0 + t_{pi}^1 + \cdots\right)}$$

$$L(p) = \sum_i l_i \boxed{t_{pi}}$$

We do this for every bounce and fold everything into a final transfer vector

A similar process can be used to model the other bounces, so that a final vector can be computed and used to map source radiance to outgoing radiance at every point on the object.

Outgoing radiance is then just the dot product of the lights projection coefficients with the transfer vector.



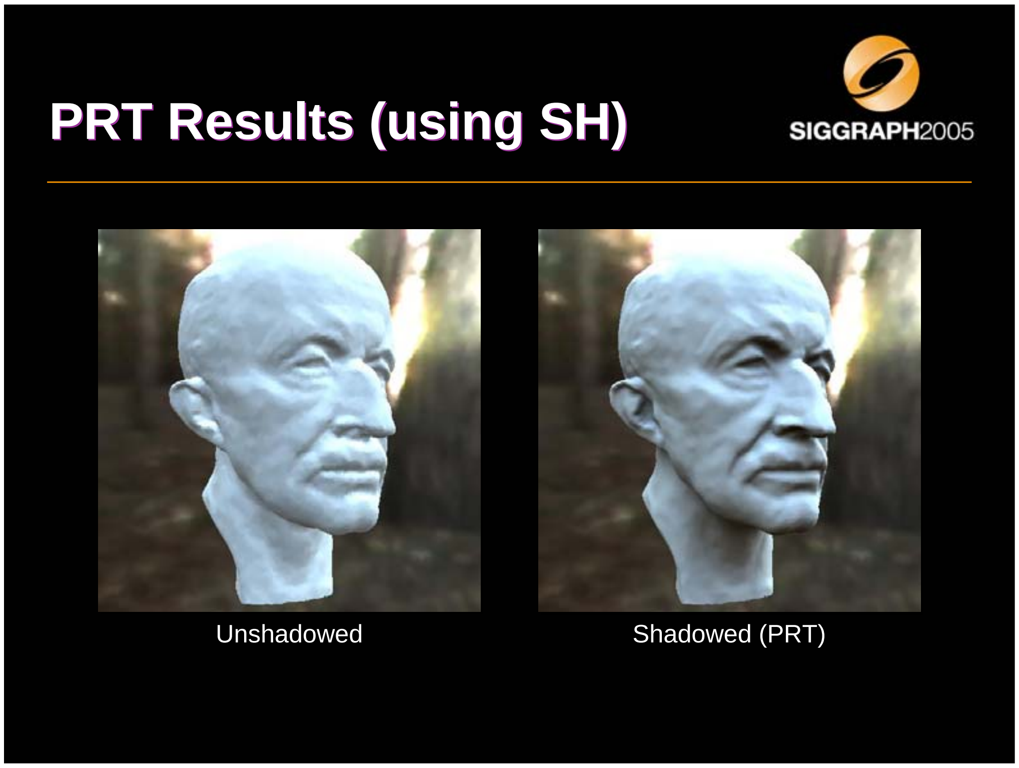
Diffuse PRT

$$L(p) = \int L_{env}(\vec{s}) V(p \to \vec{s}) \max(\vec{n} \cdot -\vec{s}, 0) d\vec{s}$$

This shows the rendering process.

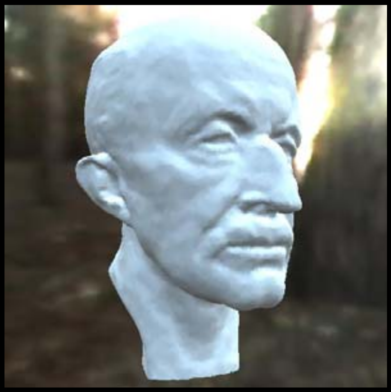
We project the lighting into the basis (integral against basis functions). If the object is rotated wrt. to the lighting, we need to apply the inverse rotation to the lighting vector (in case of SH, use rotation matrix).

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 outgoing radiance at that point.

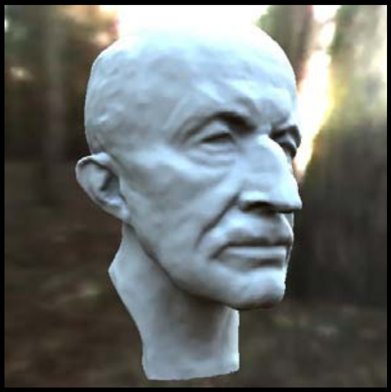


PRT Results (using SH)

SIGGRAPH2005



Unshadowed

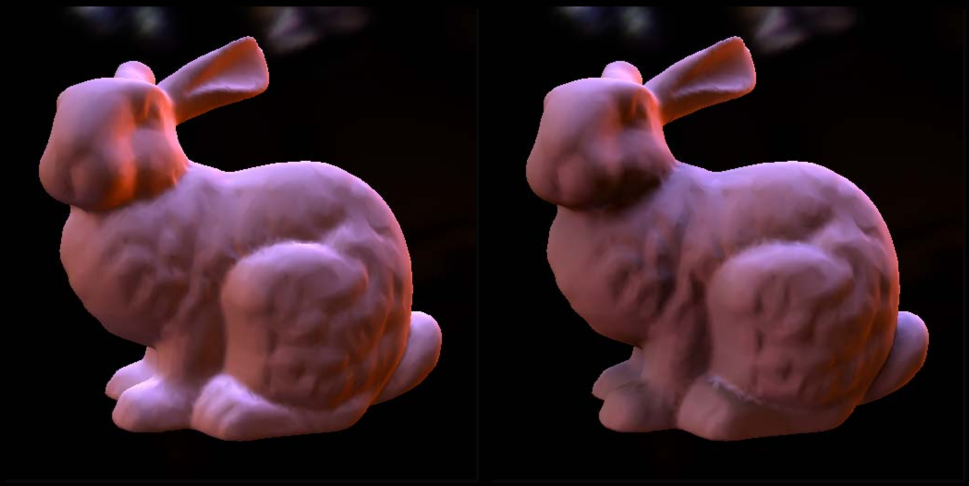


Shadowed (PRT)

On the left, you can see results down with previous techniques (no shadowing), and on the right using Precomputed Radiance Transfer.

PRT Results (using SH)

SIGGRAPH2005

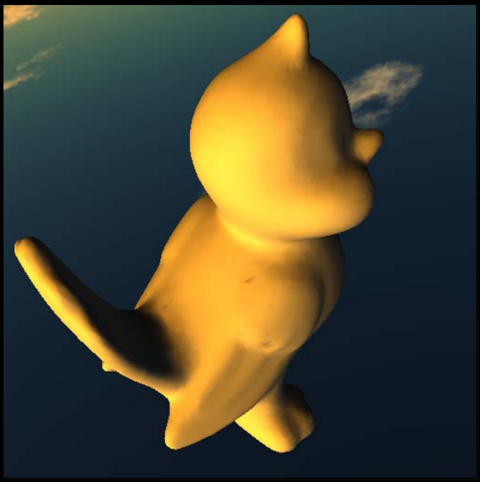


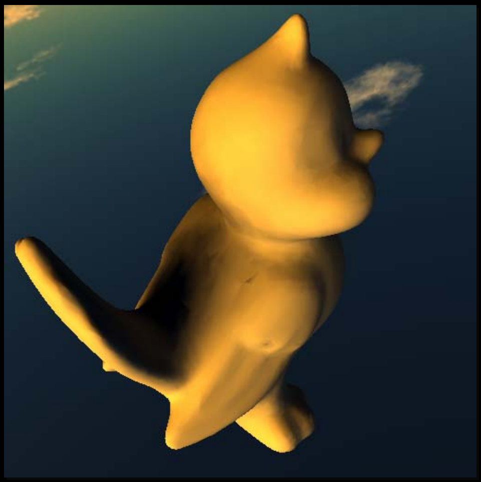
Unshadowed

Shadowed (PRT)

PRT Results (using SH)

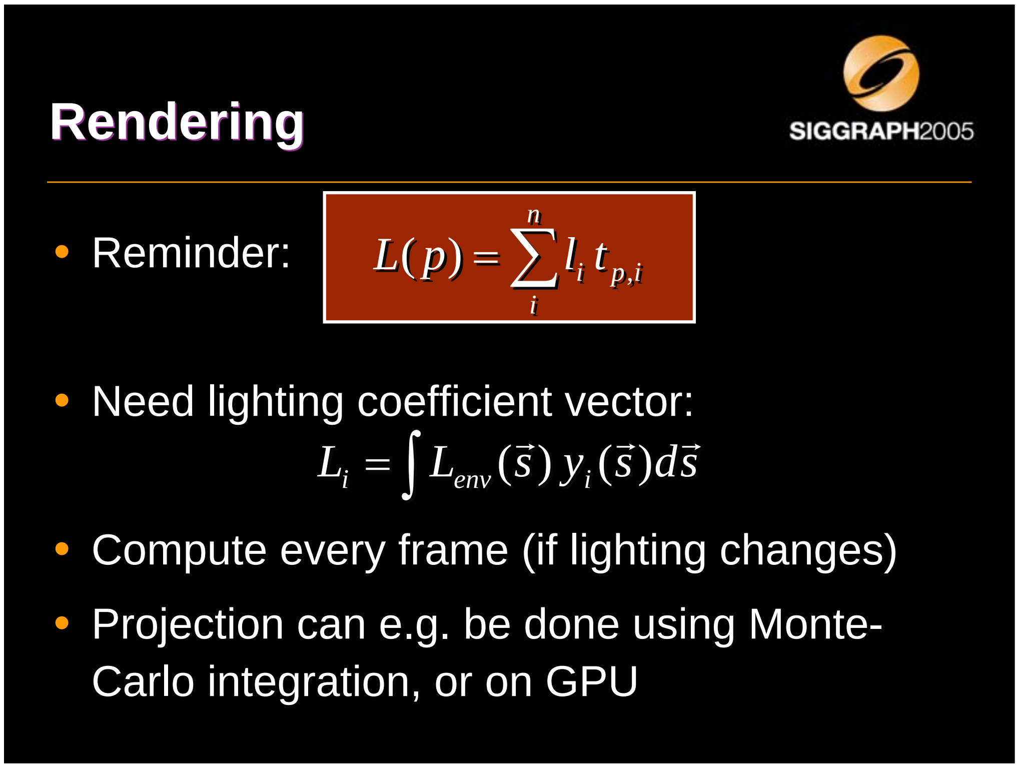
SIGGRAPH2005


Unshadowed


Shadowed (PRT)

SIGGRAPH2005

Demos



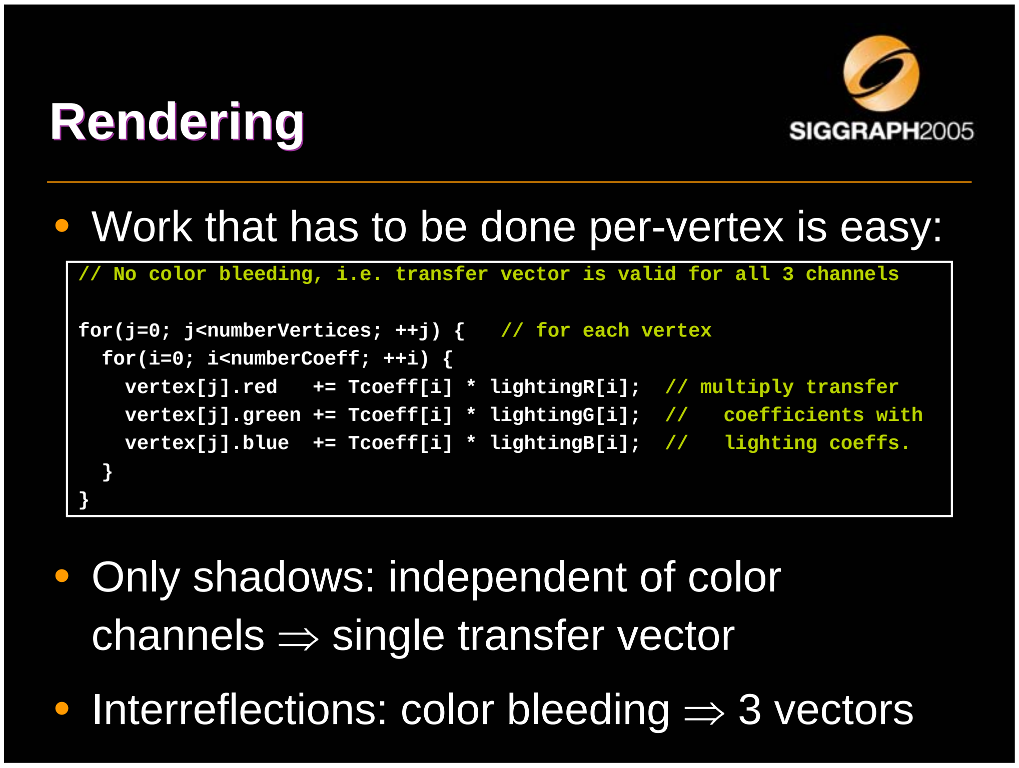
Rendering

SIGGRAPH2005

- Reminder: $L(p) = \sum_{i}^{n} l_i \, t_{p,i}$
- Need lighting coefficient vector:
  $$L_i = \int L_{env}(\vec{s}) \, y_i(\vec{s}) d\vec{s}$$
- Compute every frame (if lighting changes)
- Projection can e.g. be done using Monte-Carlo integration, or on GPU

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

SIGGRAPH2005

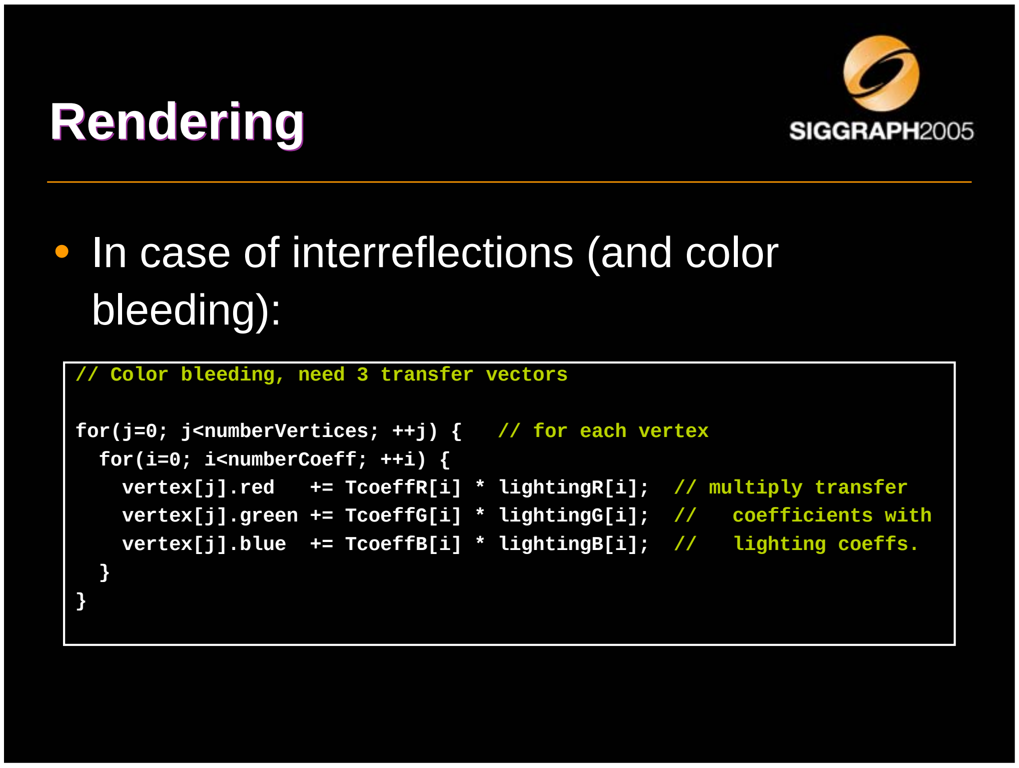
- Work that has to be done per-vertex is easy:

```
// 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.



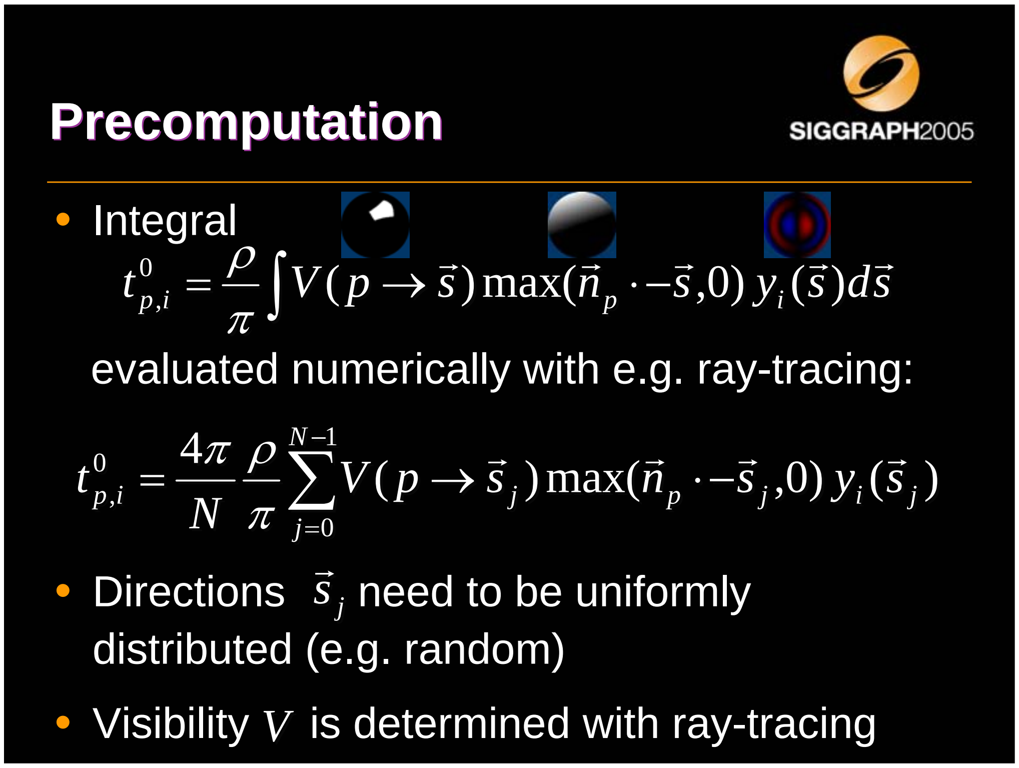
Rendering

SIGGRAPH2005

- 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.
  }
}
```



Precomputation

SIGGRAPH2005

- Integral

$$t^0_{p,i} = \frac{\rho}{\pi}\int V(p \to \vec{s})\max(\vec{n}_p \cdot -\vec{s},0)\, y_i(\vec{s})\,d\vec{s}$$

evaluated numerically with e.g. ray-tracing:

$$t^0_{p,i} = \frac{4\pi}{N}\frac{\rho}{\pi}\sum_{j=0}^{N-1} V(p \to \vec{s}_j)\max(\vec{n}_p \cdot -\vec{s}_j,0)\, y_i(\vec{s}_j)$$

- Directions $\vec{s}_j$ need to be uniformly distributed (e.g. random)
- Visibility V 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 *s_j*, which we use to sample the integrand. All the contributions are then summed up and weighted by 4*pi/(#samples).

The visibility V(p->s) needs to be computed at every point. The easiest way to do this, is to use ray-tracing.

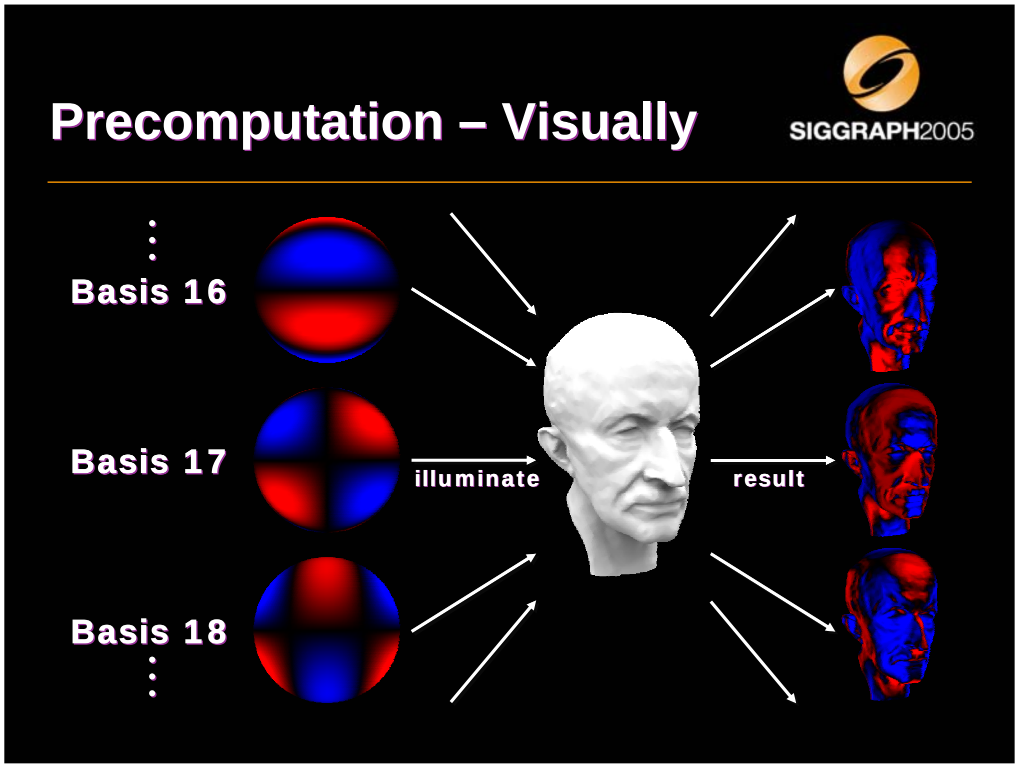
Aisde: uniform random directions can be generated the following way.

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

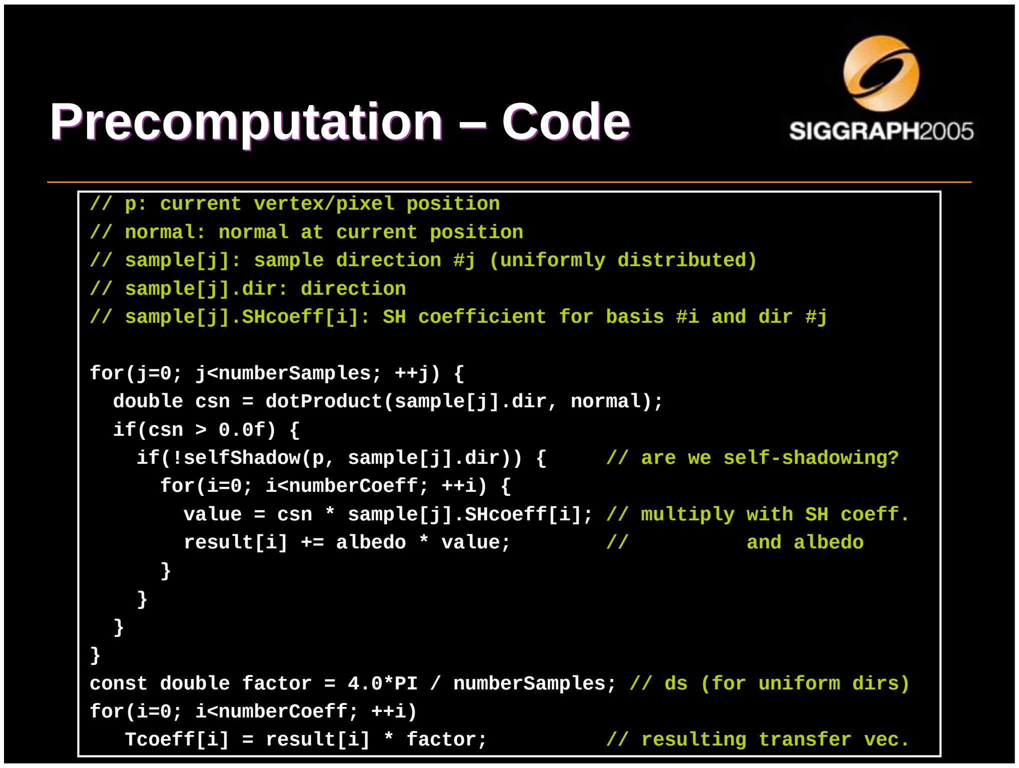
SIGGRAPH2005

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

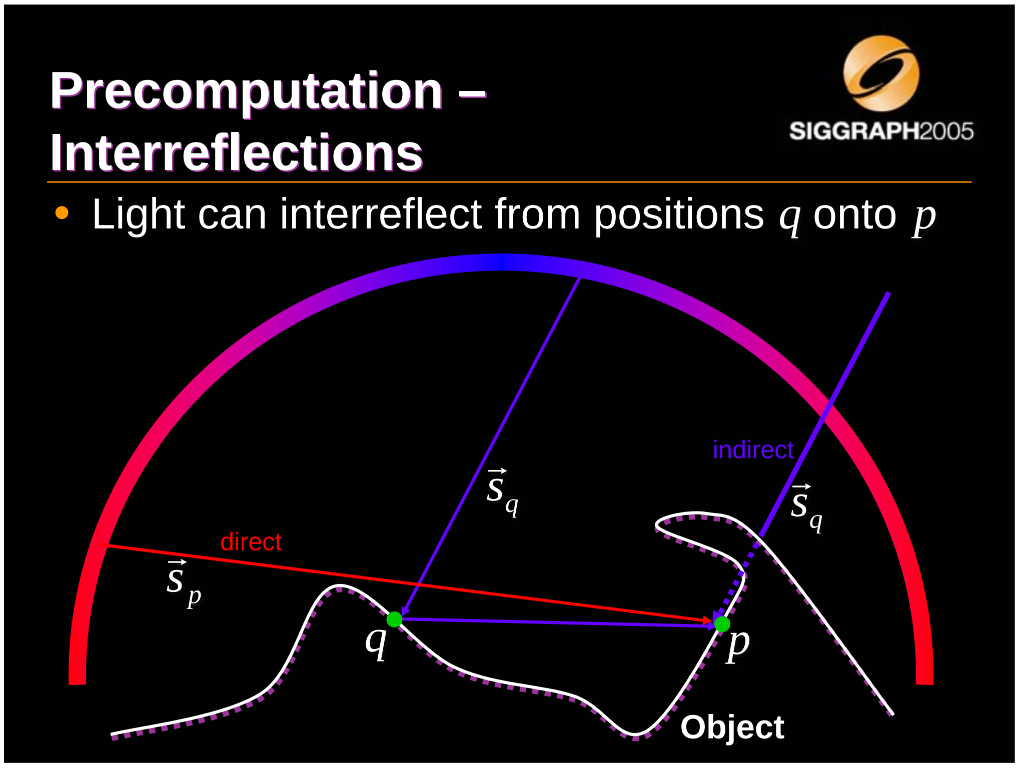
SIGGRAPH2005

```
// 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.
```

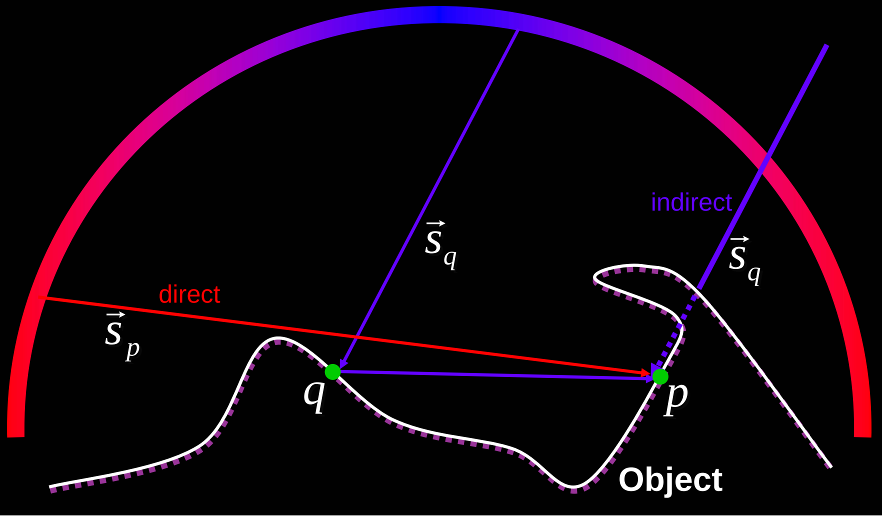
Pseudo-code for the precomputation.

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 – Interreflections

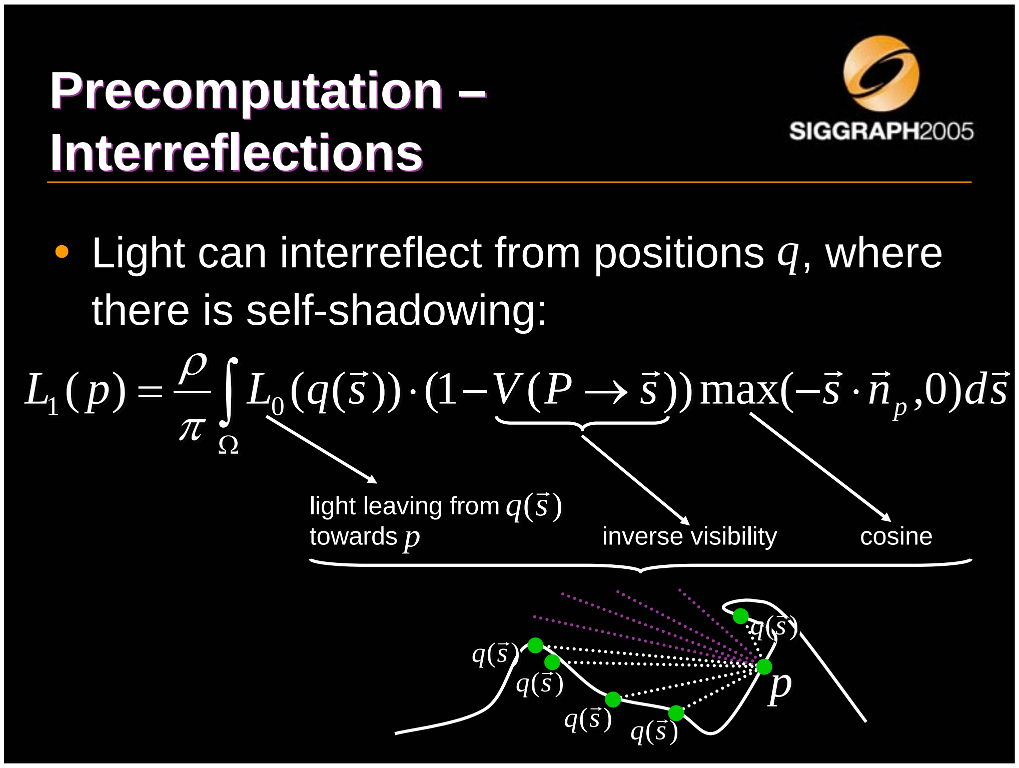
- Light can interreflect from positions q onto p



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

SIGGRAPH2005

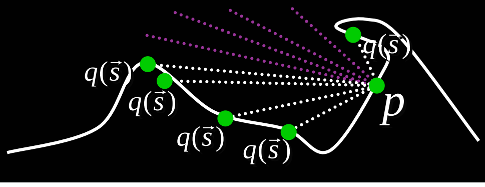
- Light can interreflect from positions q, where there is self-shadowing:

$$L_1(p) = \frac{\rho}{\pi}\int_\Omega L_0(q(\vec{s}))\cdot(1-\underbrace{V(P\to\vec{s})})\max(-\vec{s}\cdot\vec{n}_p,0)d\vec{s}$$

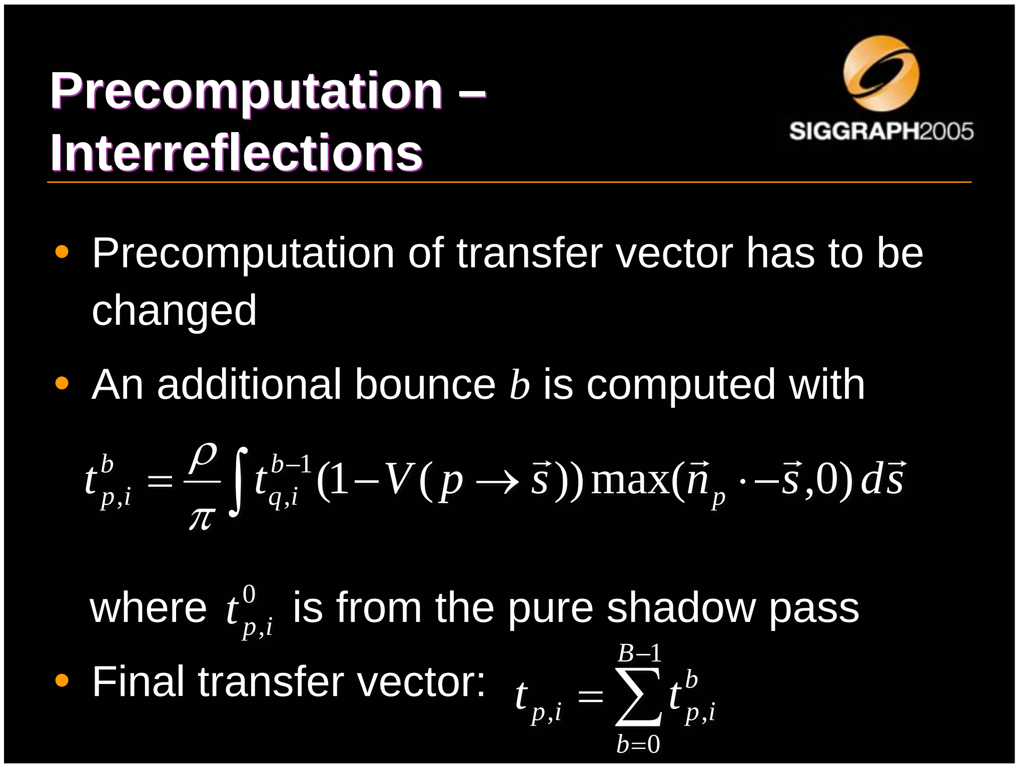
light leaving from $q(\vec{s})$ towards p

inverse visibility

cosine



More formally, we do not only have direct illumination *L_0*, 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

$$t_{p,i}^{b} = \frac{\rho}{\pi} \int t_{q,i}^{b-1} (1 - V(p \rightarrow \vec{s})) \max(\vec{n}_p \cdot -\vec{s}, 0)\, d\vec{s}$$

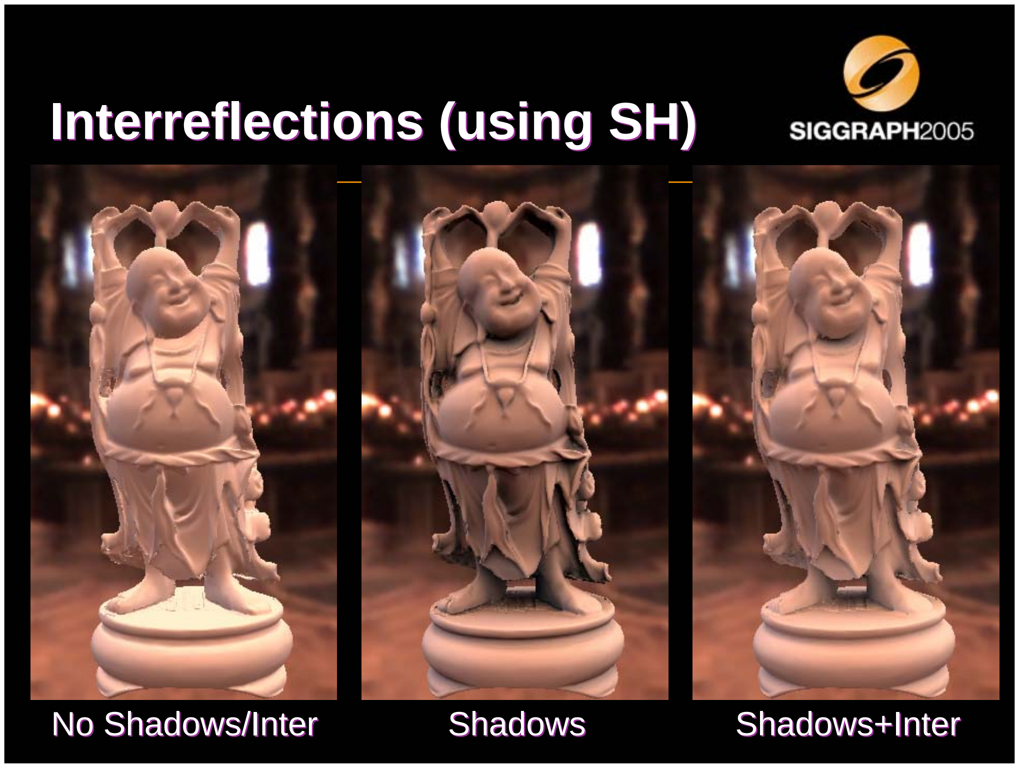
where $t_{p,i}^{0}$ is from the pure shadow pass

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

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

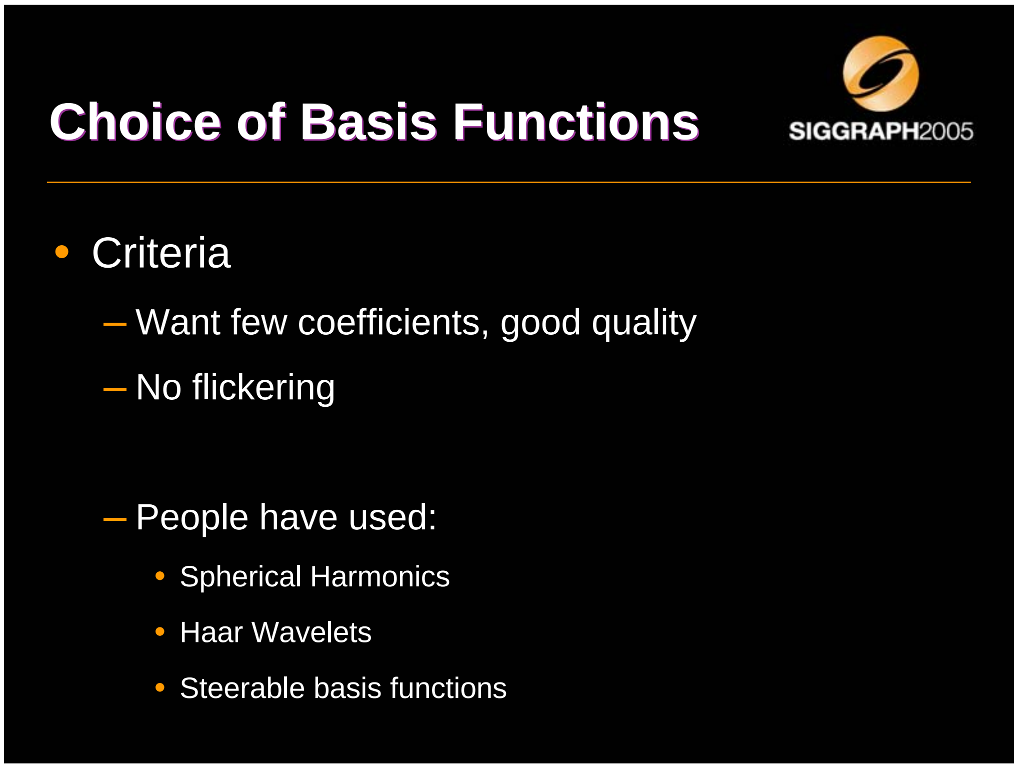
Each additional bounce *b* generates a vector *T^b_p = [t^b_{p,0}, …]*, which is computed as shown on the slide. Each of these additional transfer vectors is for a certain bounce.

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



Interreflections (using SH)

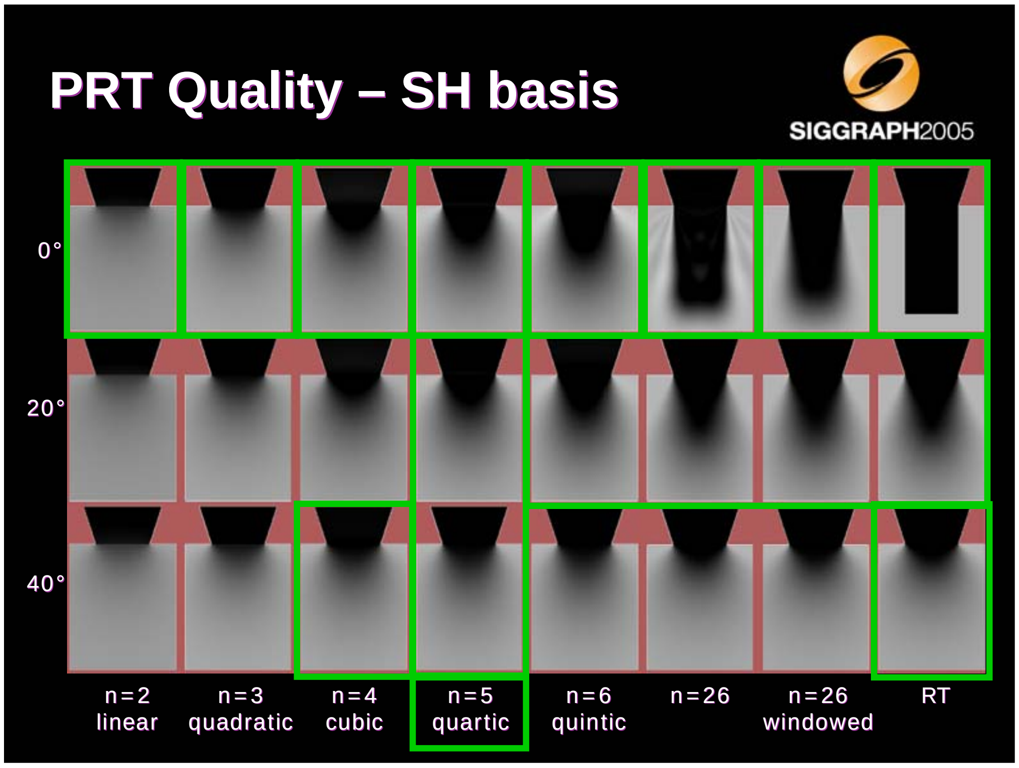
This set of images shows the buddha model lit in the same lighting environment, without shadows, with shadows and with shadows and inter reflections.



Choice of Basis Functions

SIGGRAPH2005

- Criteria
  - Want few coefficients, good quality
  - No flickering
  - People have used:
    - Spherical Harmonics
    - Haar Wavelets
    - Steerable basis functions



PRT Quality – SH basis

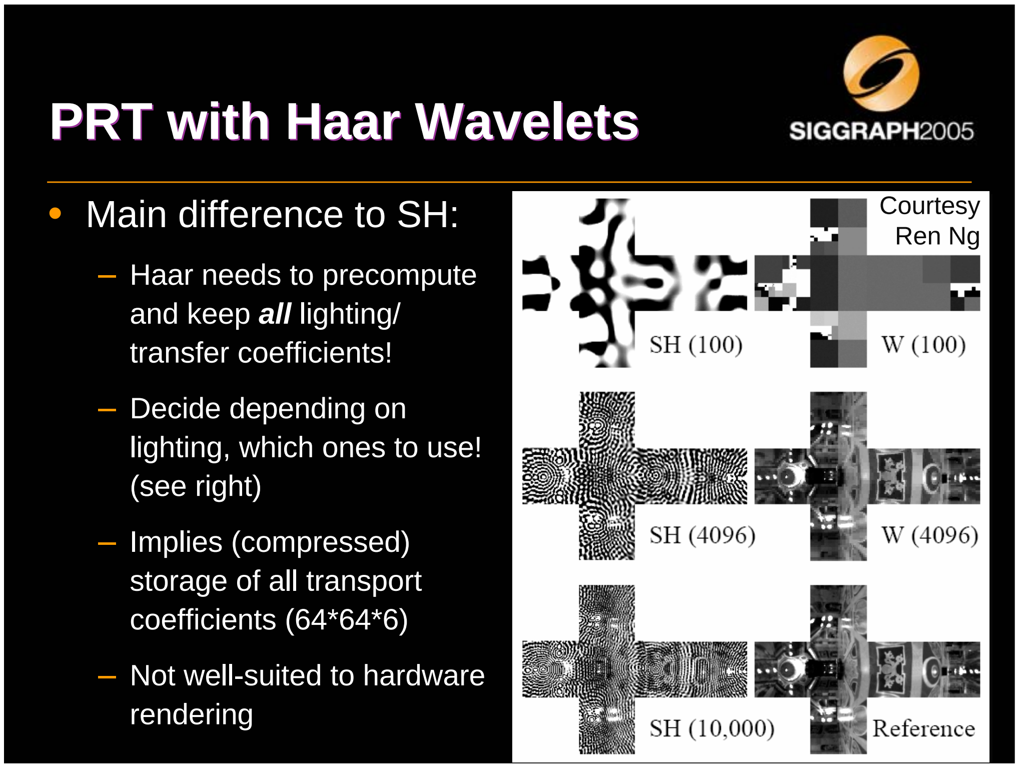
SIGGRAPH2005

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^2 = number of basis functions). Very right: exact solution.

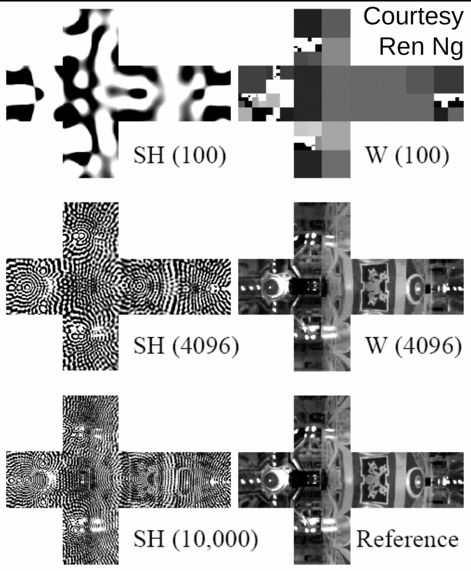
As stated before, lighting is assumed low-frequency, i.e. point light doesn't work well, but large area lights do!



PRT with Haar Wavelets

SIGGRAPH2005

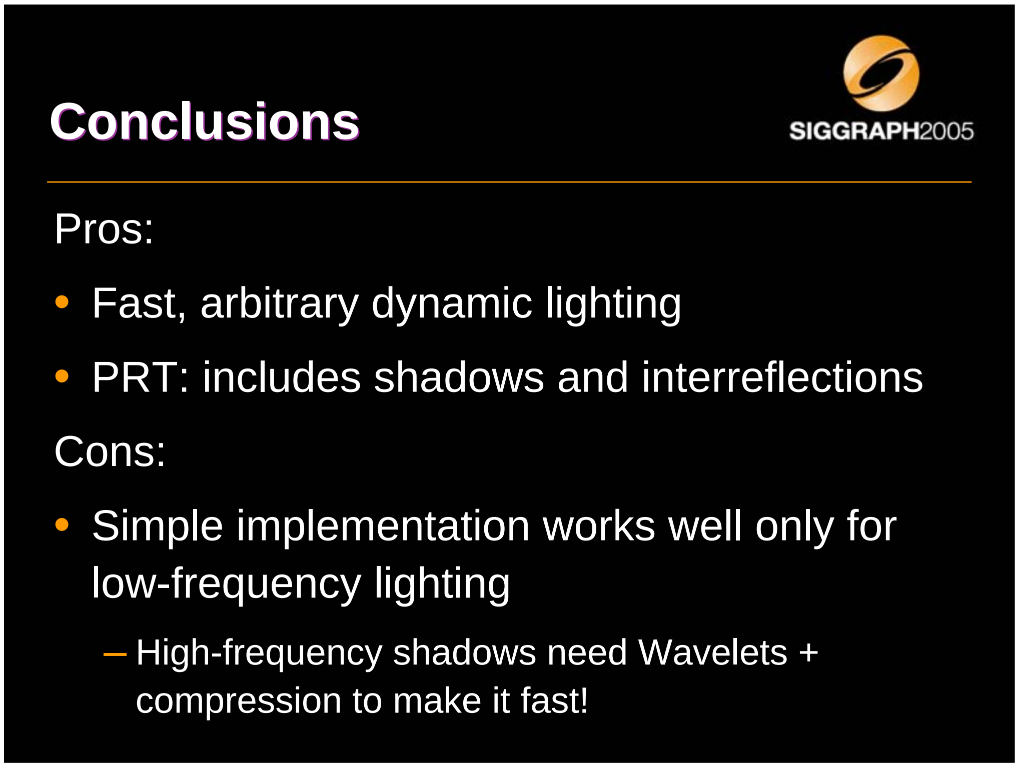
- Main difference to SH:
  - Haar needs to precompute and keep ***all*** lighting/ transfer coefficients!
  - Decide depending on lighting, which ones to use! (see right)
  - Implies (compressed) storage of all transport coefficients (64*64*6)
  - Not well-suited to hardware rendering



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 and stored!
2) Because which of the actual N coefficients are 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 great on a CPU)



Conclusions

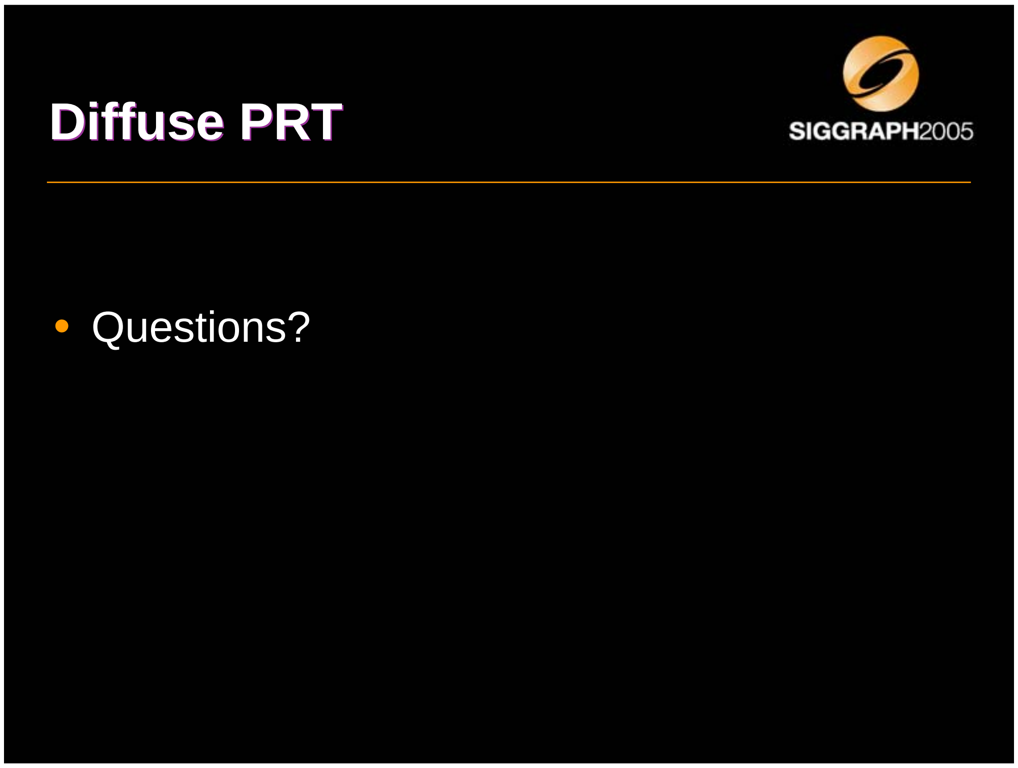
SIGGRAPH2005

Pros:

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

Cons:

- Simple implementation works well only for low-frequency lighting
  - High-frequency shadows need Wavelets + compression to make it fast!



Diffuse PRT

SIGGRAPH2005

- Questions?

SIGGRAPH2005