
Physically-Based Reflectance for Games

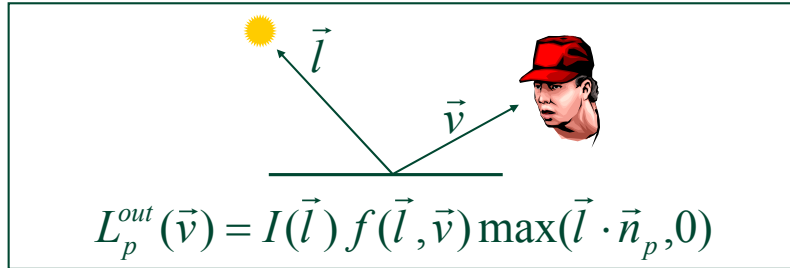
11:15 - 12:00: Reflectance Rendering
with Environment Map Lighting

Jan Kautz



Shading Computation

- Computer games often only use point lights

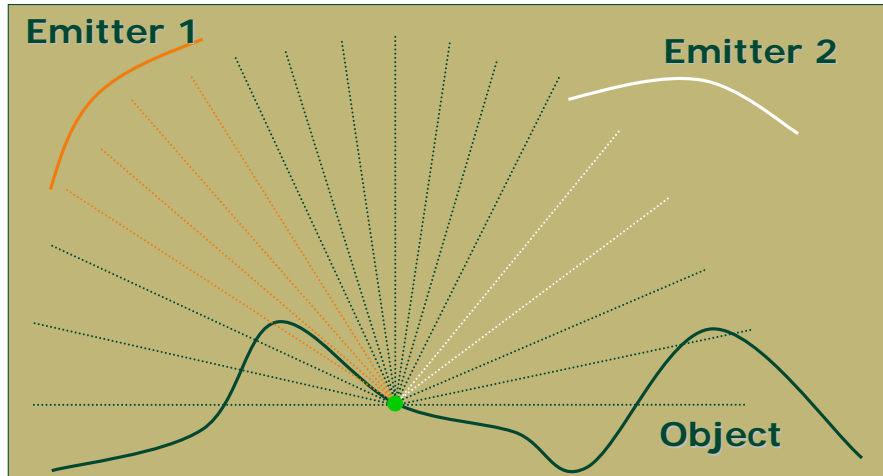


- Easy to compute
- But rarely occurs in reality



Shading Computation

- Actually: light arrives from many directions



Shading Computation

- Results are more realistic



Point Light



Global Incident Light

- Want to use this in real-time somehow

→ **Environment Maps** are a solution



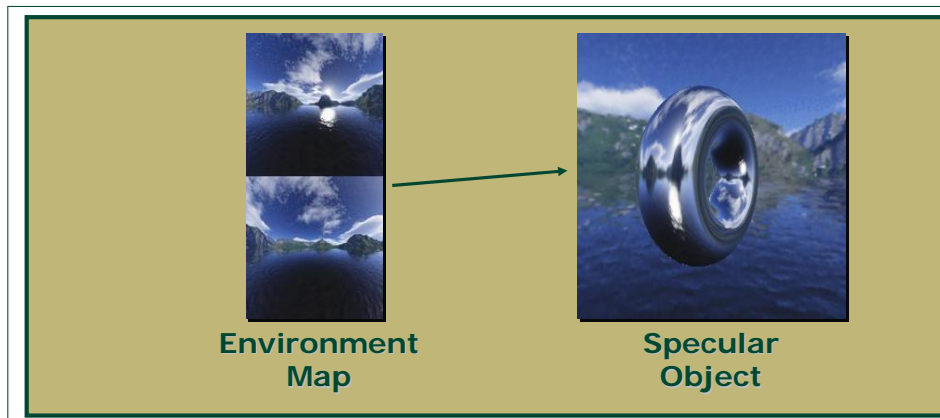
Reflectance Rendering with Environment Map Lighting

- Environment Maps
- Filtered Environment Maps
 - Diffuse Reflections
 - Glossy Reflections
- Anti-Aliasing
- Precomputed Radiance Transfer



In this section, we shall discuss the practical considerations for rendering reflectance models with more general lighting, such as environment maps. First we will cover methods for using environment map lighting with the simplest types of reflection: perfectly diffuse (Lambertian) and perfectly specular (mirror). Next, we discuss methods for rendering more general types of reflections with environment maps. Following that, we will discuss issues related to anti-aliasing when rendering reflection models with environment maps. Finally we shall discuss a special class of rendering algorithms for general lighting, precomputed radiance transfer, in the context of various reflection types.

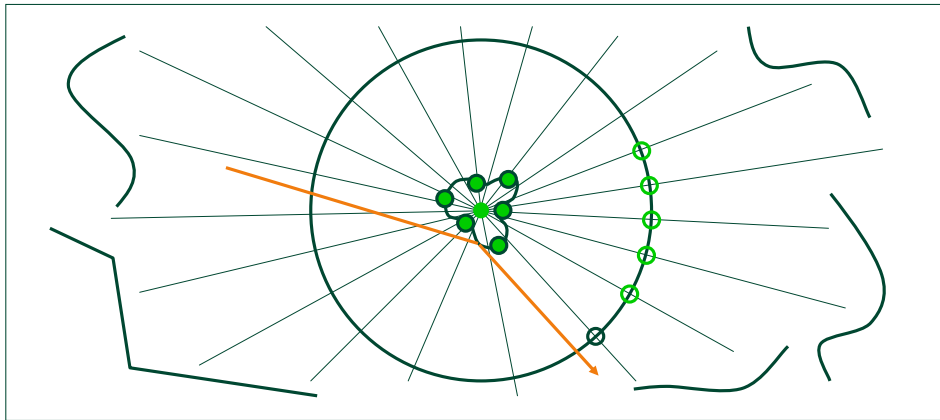
Environment Maps



- **Definition:**
 - 2D texture for directional information
 - Radiance arriving at a single point.
- **Assumptions:**
 - Environment map at infinity \Rightarrow valid for whole object

Environment maps are essentially 2D textures that store directional information. The directional information is the incident lighting arriving at a single point. This information can be used to render reflections off mirroring objects, such as the torus shown on the upper right corner. To this end, the reflected viewing direction at a surface point is used to lookup into the environment map. This makes the implicit assumption that the environment is infinitely far away.

Environment Maps



- **Definition:**

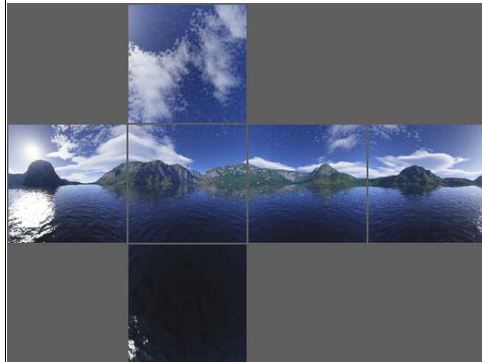
- 2D texture for directional information
- Radiance arriving at a single point.

- **Assumptions:**

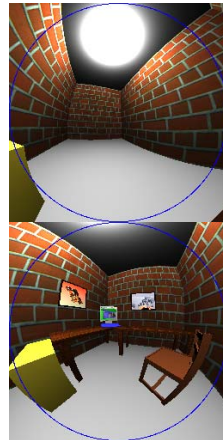
- Environment map at infinity \Rightarrow valid for whole object

Here we see how the assumption that the environment is infinitely distant slightly changes the direction that is used to lookup into the map.

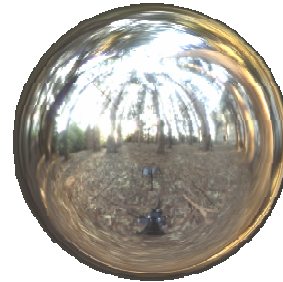
Environment Maps: Parameterizations



Cube Map (most common)



Parabolic Map



Sphere Map



There are different ways to store this spherical information.

- Cube Maps: a commonly used format nowadays, that is supported by GPUs
- Parabolic Maps: a parameterization that stores two hemispheres separately.
- Sphere Map: original format supported by GPUs (OpenGL). This parameterization corresponds to the reflection off a metal sphere.

Filtered Environment Maps

- Theory
- Diffuse Reflections
 - Spherical Harmonics
- Glossy Reflections
 - Prefiltering
 - On-the-fly Filtering
- Implementation and Production Issues

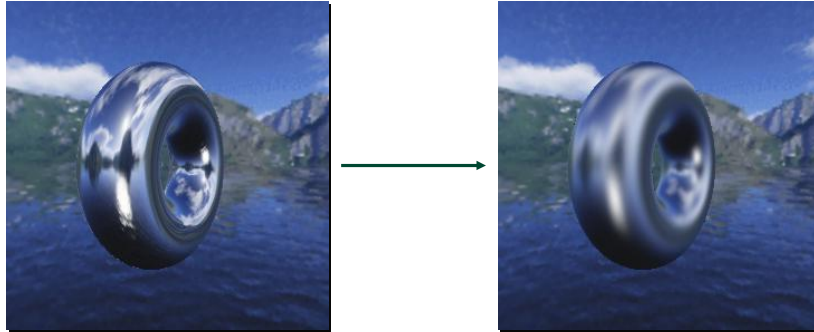


SIGGRAPH2006

We will first discuss general reflections with environment maps. Next we will discuss diffuse reflections and using spherical harmonics to represent lighting on diffuse surfaces. Glossy reflections will be treated afterwards, and finally we will discuss implementation and production issues related to environment maps.

Filtered Environment Maps

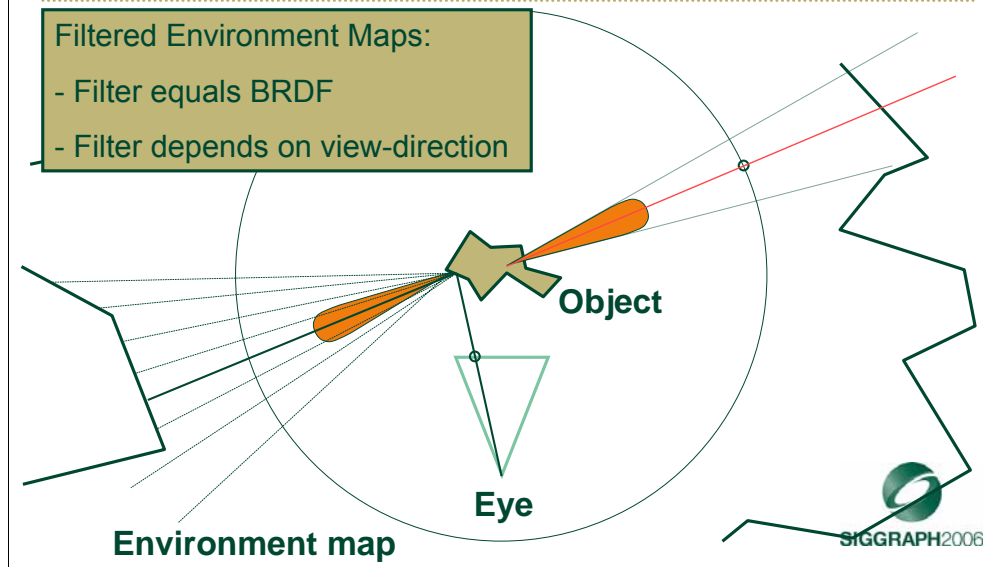
- Goal: **different materials**
⇒ diffuse and glossy reflections



⇒ Necessary: use of other BRDFs

The goal of filtered environment maps is to produce reflective objects that are not just mirrors. E.g., that can include purely diffuse object reflecting an environment or more glossy objects, as shown here.

Filtered Environment Maps



A filtered environment map stores a filtered version of the original map. Glossy surfaces reflect light from a larger cone of directions, which depends on the material (BRDF). A filtered environment map applies the BRDF, i.e. it convolves the original map to get a “blurry” version. In theory the filter (BRDF) depends on the direction of the viewer.

Filtered Environment Maps

- Environment maps:
 - Store incident light
- Filtered environment maps:
 - Store **reflected** light for all possible surface orientations and view directions (prefiltering)
 - Index into environment map with e.g.
 - Reflected view direction or surface normal direction
 - Depends on chosen reflectance model

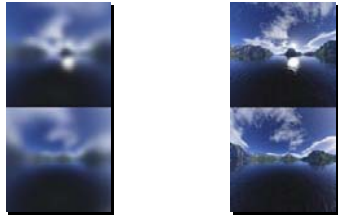


Filtered environment maps store reflected light instead of incident light.

Filtered Environment Maps

- General filtered environment maps:

$$L_e(\vec{v}, \vec{n}, \vec{t}) = \int_{\Omega} \underbrace{L_{env}(\vec{l})}_{\text{Environment Map}} \underbrace{f_r(\omega(\vec{v}, \vec{n}, \vec{t}), \omega(\vec{l}, \vec{n}, \vec{t}))(\vec{n} \cdot \vec{l})}_{\text{BRDF}} d\vec{l}$$



BRDF

- Depends on (global) view direction, tangent frame
- Output: 5D table \Rightarrow too expensive !!



SIGGRAPH2006

Reflected radiance is computed by integrating all incident light multiplied by the BRDF and the cosine between the normal and the lighting. Here we see that the BRDF depends on the coordinates of the view and lighting direction in the local coordinate system at the current point (BRDF requires local directions, the function $w()$ converts from global to local coordinate system). The outgoing radiance depends on the viewing direction, the surface normal and the tangent (in case of anisotropic BRDFs). Tabulating this ends up being a 5D table!

Filtered Environment Maps

- Goal: avoid high memory consumption
- Solutions:
 - E.g. Certain BRDFs \Rightarrow output only 2D
 - E.g. Arbitrary BRDFs \Rightarrow approximations



Fortunately, it is possible to avoid the high memory requirements by choosing appropriate BRDFs or approximating them the right way.

Filtered Environment Maps

- Theory
- Diffuse Reflections
 - Spherical Harmonics
- Glossy Reflections
 - Prefiltering
 - On-the-fly Filtering
- Implementation and Production Issues

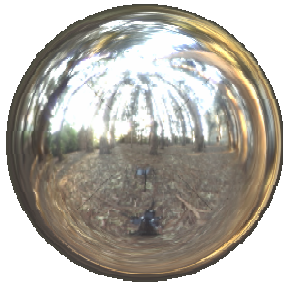


SIGGRAPH2006

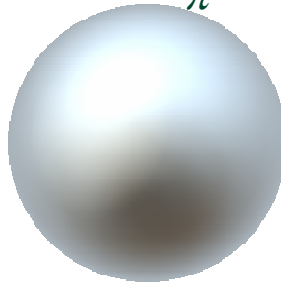
Now we will discuss using spherical harmonics to represent lighting on diffuse surfaces (irradiance environment maps).

Diffuse Environment Maps

- Diffuse prefiltering [Greene '86]
- Results in 2D map parameterized over the surface normal:
$$L_e(\vec{n}) = \frac{\rho_d}{\pi} \int_{\Omega} L_{env}(\vec{l})(\vec{n} \cdot \vec{l}) d\vec{l}$$



Original



Diffuse



Result

The first material we want to look at are diffuse materials. The BRDF of a diffuse material is just a constant ρ_d . The integral of the lighting and the cosine between the normal and integration direction only depends on the surface normal. Hence the filtered environment map also only depends on the normal, which makes it again a 2D environment map.

Notice how much smoother the diffusely prefiltered environment map looks like.

Diffuse Environment Maps

- Brute-force filtering is **very** slow
 - Can be tens of minutes for large environment maps
- Observation:
 - Diffuse environment maps are low-frequency
 - ⇒ due to large filter kernel
 - Hence, filtering in frequency-space is faster



SIGGRAPH2006

Filtering can be done brute-force (by applying the $\cos\theta_i$ filter at every texel of the environment map). But that can be slow.

Due to the large filter kernel (the cosine kernel extends over a full hemisphere) and the resulting low-frequency environment map, filtering is fast in frequency-space.

Diffuse Environment Maps Using Spherical Harmonics

- Proposed by [Ramamoorthi01]
- Project lighting into spherical harmonics:

$$l_{env,k} = \int L_{env}(\vec{l}) y_k(\vec{l}) d\vec{l}$$

- Convolution with cosine-kernel

$$L_{diffuse}(\vec{n}) = \int L_{env}(\vec{l}) (\vec{n} \cdot \vec{l}) d\vec{l}$$

becomes

$$L_{diffuse}(\vec{n}) = \frac{\rho_d}{\pi} \sum_{k=0}^8 A_k l_{env,k} y_k(\vec{n})$$



SIGGRAPH2006

First proposed by [Ramamoorthi01], frequency-space filtering is performed with Spherical Harmonics (the Fourier equivalent over the sphere). If the lighting is represented in SH, then the convolution becomes a simple scaled sum between the coefficients of the lighting (projection to be explained in a bit) and the coefficients of the cosine (times some convolution constants).

The exact definitions of SH can be looked up on the web. If only the first few bands are used, it is more efficient to use explicit formulas, which can be derived by using Maple for example.

The definitions for the first 25 basis functions (in `ylm_array[]`). Input is a direction.

```

float x = dir[0];
float y = dir[1];
float z = dir[2];
float x2, y2, z2;

ylm_array[0] = 0.282095f; //l,m = 0,0 // 1/sqrt(4pi)
ylm_array[1] = 0.488603f * y; //1,-1 //sqrt(3/4pi)
ylm_array[2] = 0.488603f * z; //1,0
ylm_array[3] = 0.488603f * x; //1,1

x2 = x*x; y2 = y*y; z2 = z*z;
ylm_array[4] = 1.092548f * x * y; //2,-2 // sqrt(15/4pi)
ylm_array[5] = 1.092548f * y * z; //2,-1
ylm_array[6] = 0.315392f * (3.f*z2 - 1.f); //2,0 // sqrt(5/16pi)
ylm_array[7] = 1.092548f * x * z; //2, 1
ylm_array[8] = 0.546274f * (x2 - y2); //2,2 // sqrt( 15/16pi )

const float fY30const = 0.373176332590115391414395913199f; //0.25f*sqrt(7.f/M_PI);
const float fY31const = 0.457045799464465736158020696916f; //1.f/8.0f*sqrt(42.f/M_PI);
const float fY32const = 1.445305721320277027694690077199f; //0.25f*sqrt(105.f/M_PI);
const float fY33const = 0.590043589926643510345610277541f; //1.f/8.f*sqrt(70.f/M_PI);
ylm_array[ 9] = fY33const*y*(3.f*x2 - y2); //3,-3
ylm_array[10] = fY32const*2.f*x*y*z; // 3,-2
ylm_array[11] = fY31const*y*(5.f*z2-1.f); // 3,-1
ylm_array[12] = fY30const*z*(5.f*z2-3.f); // 3,0
ylm_array[13] = fY31const*x*(5.f*z2-1.f); // 3,1
ylm_array[14] = fY32const*z*(x2-y2); // 3,2
ylm_array[15] = fY33const*x*(x2-3.f*y2); // 3,3

const float fY40const = 0.84628437532163443042211917734116f; // 3.0f/2.0f/sqrt(M_PI);
const float fY41const = 0.66904654355728916795211238971191f; // 3.0f/8.0f*sqrt(10.f/M_PI);
const float fY42const = 0.47308734787878000904634053544357f; // 3.0f/8.0f*sqrt(5.f/M_PI);
const float fY43const = 1.7701307697799305310368308326245f; // 3.0f/8.0f*sqrt(70.f/M_PI);
const float fY44const = 0.62583573544917613458664052360509f; // 3.0f*sqrt(35.0f/M_PI)/16.f;
ylm_array[16] = fY44const*4.f*x*y*(x2-y2); // 4,-4
ylm_array[17] = fY43const*y*z*(3.f*x2-y2); // 4,-3
ylm_array[18] = fY42const*2.f*y*x*(7.f*z2-1.f); // 4,-2
ylm_array[19] = fY41const*y*z*(7.f*z2-3.f); //4,-1
ylm_array[20] = fY40const*(z2*z2 - 3.f*z2*(x2+y2) + 3.0f/8.0f*(x2+y2)*(x2+y2)); // 4,0
ylm_array[21] = fY41const*x*z*(7.f*z2-3.f); // 4,1
ylm_array[22] = fY42const*(x2-y2)*(7.f*z2-1.f); // 4,2
ylm_array[23] = fY43const*x*z*(x2 - 3.f*y2); // 4,3
ylm_array[24] = fY44const*(x2*x2 - 6.f*x2*y2 + y2*y2); // 4,4

```

Diffuse Environment Maps Using Spherical Harmonics

- Convolution

$$L_{diffuse}(\vec{n}) = \frac{\rho_d}{\pi} \sum_{k=0}^8 A_k l_{env,k} y_k(\vec{n})$$

- With convolution constants: A_k
- Simple sum instead of integral!
- $y_k(\omega)$ are simple (quadratic) polynomials
- Only **15 instructions** in shader



SIGGRAPH2006

The constants are:

$$A_0 = \pi$$

$$A_1 = A_2 = A_3 = 2\pi/3$$

$$A_4 = \dots = A_8 = \pi/4$$

See previous page for $y_k(\omega)$

Diffuse Environment Maps Using Spherical Harmonics

- Projection into Spherical Harmonics
 - Integrate basis functions against fixed HDR map

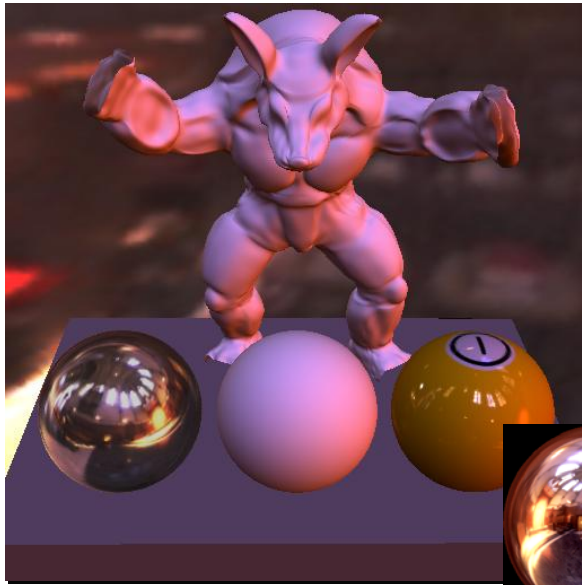
$$l_{env,k} = \int_{\Omega} L_{env}(\vec{l}) y_k(\vec{l}) d\vec{l}$$
$$l_{env,k} = \int \text{img} \cdot \text{img}$$

- 1) Monte-Carlo integration
- 2) Precompute maps in same space (e.g. cube map) that contain the basis functions \Rightarrow big dot-product for each coeff.

This is a standard scenario for projecting an HDR environment into SH basis functions.

If the environment map is given as a cube map, it is necessary to know the solid angle of a texel in the cube map. It is: $4/((x^2+y^2+z^2)^{3/2})$, where $[x \ y \ z]$ is the vector to the texel (not normalized).

Diffuse Environment Maps



IMAGES BY R. RAMAMOORTHY



full integral



SH



input

Filtered Environment Maps

- Theory
- Diffuse Reflections
 - Spherical Harmonics
- Glossy Reflections
 - Prefiltering
 - On-the-fly Filtering
- Implementation and Production Issues



SIGGRAPH2006

We will now consider various issues involved with the use of environment maps to render glossy reflections.

Glossy Reflections

- Reminder: store **reflected** light

$$L_e(\vec{v}, \vec{n}, \vec{t}) = \int_{\Omega} L_{env}(\vec{l}) f_r(\omega(\vec{v}, \vec{n}, \vec{t}), \omega(\vec{l}, \vec{n}, \vec{t})) (\vec{n} \cdot \vec{l}) d\vec{l}$$

- Table is
 - 4D for general isotropic BRDF
 - 5D for general anisotropic BRDF



SIGGRAPH2006

Reminder: Reflected radiance is computed by integrating all incident light multiplied by the BRDF and the cosine between the normal and the lighting. Here we see that the BRDF depends on the coordinates of the view and lighting direction in the local coordinate system at the current point (BRDF requires local directions). The outgoing radiance depends on the viewing direction, the surface normal and the tangent (in case of anisotropic BRDFs). Tabulating this ends up being a 5D table!

Tabulating this ends up being a 5D table in case of anisotropic BRDFs (2D for viewing direction, 3D for tangent frame), and 4D for isotropic BRDFs (2D+2D).

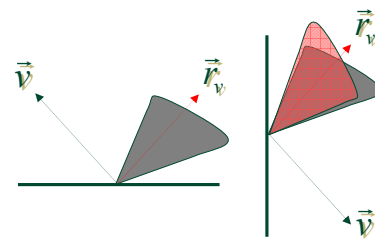
Glossy Reflections

- Intuition: lobe size/shape changes with view



- Solution: fixed lobe shape?

- In global coordinate system of the environment map, the **shape still changes**.



The reason for this is that the lobe size/shape of the BRDF changes with the viewing direction. For example reflections become sharper at grazing angles.

Even a fixed lobe shape is not sufficient, as the change from the local BRDF coordinate system to global environment map coordinate system still results in different shapes (i.e. dimensionality is still high).

Glossy Reflections

- Nonetheless, certain BRDFs (or approximations) reduce dimensionality
- Best example: Phong model
- Reason: its lobe shape is
 - fixed
 - rotationally symmetric



The solution is to find a BRDF that has a fixed lobe-shape and that the lobe shape is rotationally symmetric (so change from local to coordinate system doesn't matter).

Glossy Reflections: Phong

- Phong prefiltering [Miller84, Heidrich99]
 - Outgoing radiance of a Phong material at one point is a 2D function of the reflected viewing direction
 - Phong BRDF (global coords):

$$f_r(\vec{v}, \vec{l}) = k_s (\vec{r}_v \cdot \vec{l})^N / (\vec{n} \cdot \vec{l})$$

- “Blurred” environment map resulting from shift-invariant Phong filter:

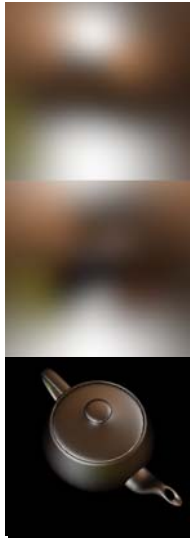
$$L_e(\vec{r}_v) = k_s \int_{\Omega} (\vec{r}_v \cdot \vec{l})^N L_{env}(\vec{l}) d\vec{l}$$



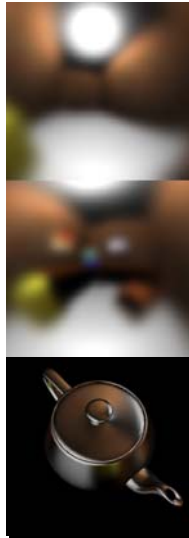
SIGGRAPH2006

The best example is the Phong BRDF, as defined here (original definition using global coordinate system). Note that the Phong material is physically not plausible, but results in 2D environment maps.

Glossy Reflections: Phong



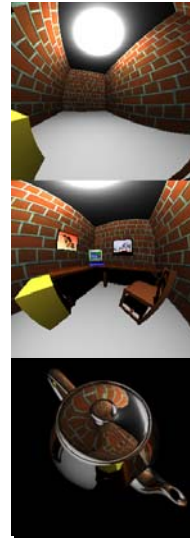
N=10



N=100



N=1000



perfect

Here we have an example of Phong environment maps, for different exponents $N=10, 100, 1000$.

Glossy Reflections: Phong + Diffuse (Combined with Fresnel)



IMAGES BY W. HEIDRICH



Here we combined Phong filtered environment maps with diffuse environment maps. The combination is governed by the Fresnel equations. This results in very realistic looking materials.

Glossy Reflections: Phong Problem

- Phong model is not realistic
- Real materials are sharper at grazing angles:



IMAGES BY E. LAFORTUNE

The Phong model is not realistic, as its lobe shape remains constant for all viewing directions! Real reflections become sharper at grazing angles, e.g. when looking at a sheet of paper at very grazing angles will show a visible glossy reflection.

Glossy Reflections: Real Materials

- Lobe becomes narrower and longer at grazing angles:



- For increased realism, want to model that
 - Need to use other BRDFs
 - **Without** increasing dimensionality of filtered environment map



I.e., the lobe shape changes. For more realism, we want to include this effect in our filtered environment maps.

Glossy Reflections: Lafortune Model

- Lafortune model
 - Derived from Phong model
 - Keeps lobe shape as in Phong, but modifies viewing direction to achieve off-specular lobes
 - Allows for anisotropic reflections



The first example that allows this (to some limited extent) is the Lafortune model. It is basically a Phong model, but slightly more realistic, allowing a more realistic class of materials to be represented.

Glossy Reflections: Lafortune Model

- [McAllister02] proposed to use it for environment maps:

$$L_{laf}(\vec{r}_w) = (\vec{n} \cdot \vec{r}_v) k_s \int_{\Omega} (\vec{r}_w \cdot \vec{l})^N L_{env}(\vec{l}) d\vec{l}$$

with $\vec{r}_w = (C_x v_x, C_y v_y, C_z v_z)$

- Note that $(\vec{n} \cdot \vec{l})$ is moved outside the integral as $(\vec{n} \cdot \vec{r}_v)$ which is an approximation.
 - Ok for sharp lobes, otherwise inaccurate



SIGGRAPH2006

Plugging the model into the reflectance equation, we arrive at the above equation. Note how similar the integral is to the original Phong environment maps.

The main difference is the direction r_w , which isn't necessarily just the reflected view direction, it can be any scaled version of the view direction, which allows for off-specular reflections.

Note further how the term $(n \cdot l)$, which should be inside the integral is simply moved outside by approximating it with $(n \cdot r_v)$. This is incorrect of course, but for high exponents N , the influence of $(n \cdot l)$ is rather small anyway (it doesn't vary much where $(r_w \cdot l)^N \neq 0$), so moving it outside the integral is an acceptable approximation. For small N , the approximation is very crude.

Glossy Reflection: Lafortune Example



IMAGE BY D. MCALLISTER

Additionally:

- Stored several 2D filtered environment maps as 3D stack (varying exponent)
- Per-pixel lookups to achieve spatial variation



Example of rendering with Lafortune example.

Here the authors have prefiltered an environment map with different exponents and stored it in a mip-mapped cube map. At each texel, they have a roughness map, which governs in which level of the mip-map to index. You can also note that the Lafortune model allows for anisotropies.

Glossy Reflections: BRDF Approximations [Kautz00]

- Approximate BRDF [Kautz00] with
 - rotationally-symmetric &
 - constant lobe
- E.g:



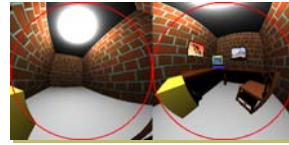
[Kautz00] proposed to approximate BRDFs with a rotationally-symmetric and fixed lobe such that filtered environment maps would remain 2D (like the Phong lobe).

Glossy Reflections: BRDF Approximations [Kautz00]

- Given:



+

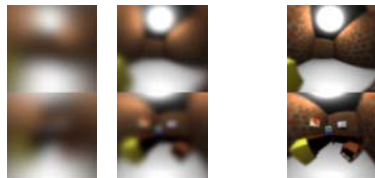


Environment Map

- Fit rotationally symmetric lobes:



- Filter environment map with lobes:



Given some arbitrary shaped lobe, you can see here rotationally-symmetric lobes and the environment maps filtered with it.

You can still see how the width/length of the lobe changes with viewing direction.

Glossy Reflections: BRDF Approximations [Kautz00]

- Any BRDF can then be used for filtering:
 - for good quality: need **separate lobe** for each view
 - results in 3D environment map (stack of 2D maps)
 - otherwise: use **single (scaled) lobe** for each view
 - basically like Phong or Lafortune
- Similar approximation as Lafortune needed
 - move $(n \cdot l)$ outside integral, or otherwise the lobes are not rotationally symmetric

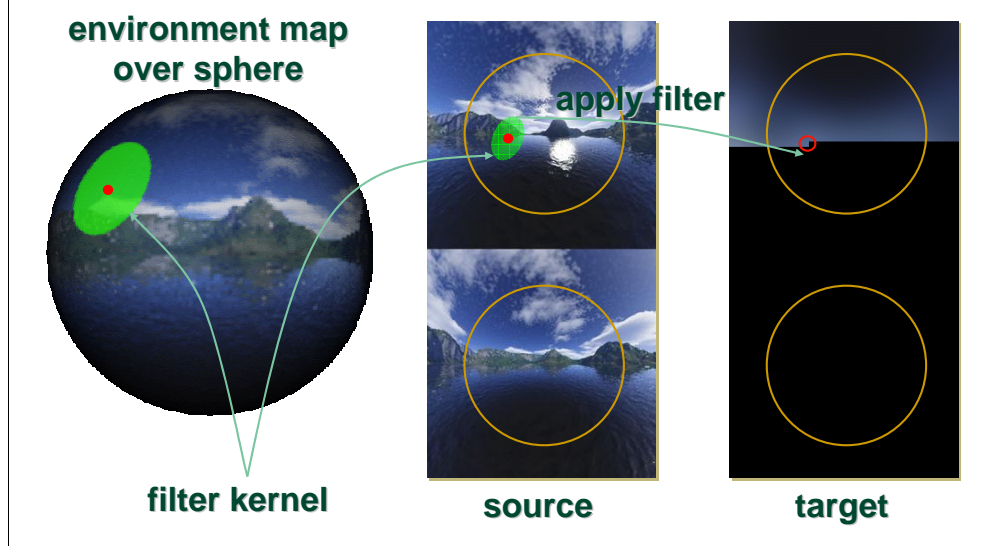


For realistic results, it is necessary to use these separate lobes and filter the environment map with it. This results in a stack of 2D environment maps, which are indexed with the elevation angle of the viewing direction (anisotropic BRDFs are not considered).

The authors also propose to approximate the BRDF with one (scaled) fixed lobe, which then is very similar to the Phong/Lafortune BRDFs (but based on measured BRDFs, which they approximate).

Again, $(n \cdot l)$ is moved outside the integral.

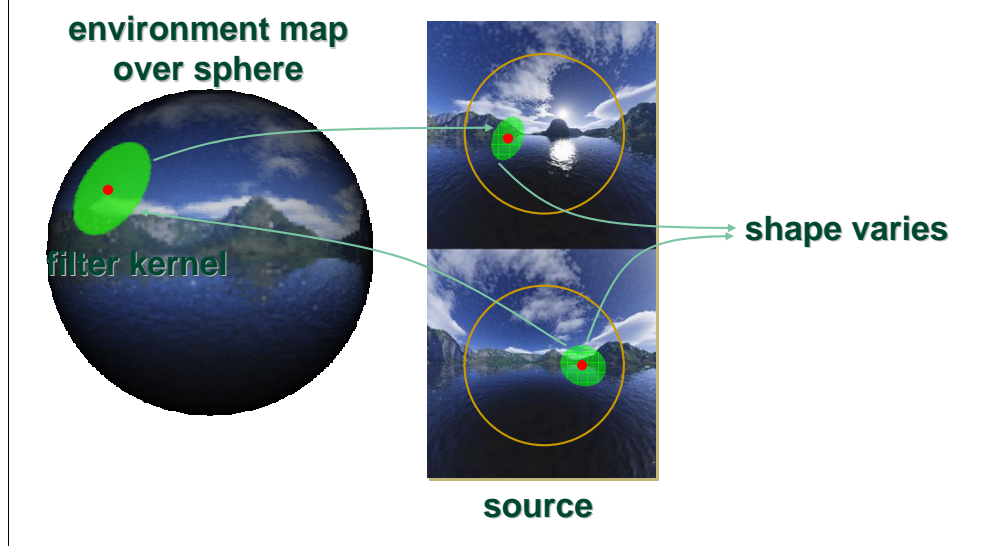
Glossy Reflections: How to Filter a Cube Map?



Now we have some idea on how prefiltered environment maps work. We don't know exactly how to filter one.

It is quite simple: for each texel in the target (filtered) environment map, we apply a convolution filter. This filter needs to be mapped from the sphere (the domain over which it is defined) into the texture domain of the environment map.

Glossy Reflections: How to Filter a Cube Map?



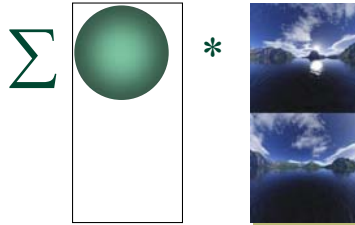
This usually means that the filter is spatially-varying in the environment map-domain, as there is no mapping from the sphere to the plane that doesn't introduce distortions.

Glossy Reflections: How to Filter a Cube Map?

- Easy implementation: **brute force**

— For each target pixel,  go over the full

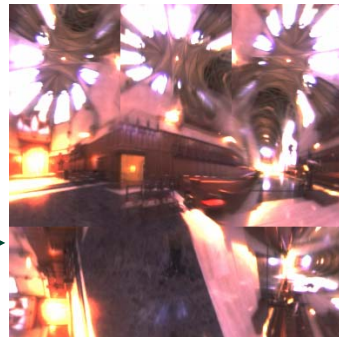
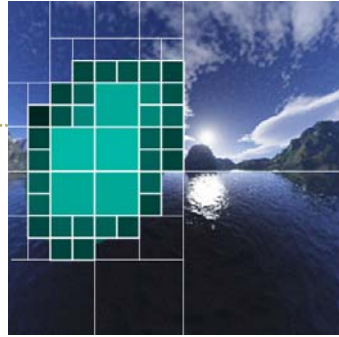
input map, and do:
(take solid angle
of each pixel into
account)



A simple implementation just does brute-force filtering.

Glossy Reflections: How to Filter a Cube Map?

- Faster (seconds), less accurate:
 - Hierarchically [Kautz00] →
 - Angular filtering with cut-off [CubeMapGen-ATI05]
 - Filter each face separately, surrounded by other faces [Ashikhmin03] →
 - In frequency domain [S2kit]



Faster and less accurate filtering can be done hierarchically [Kautz00].

Angular extent filtering: ATI's `cubemapgen` tool does correct filtering, but clamps the filter at a certain threshold and only filters within the bound of the filter. Much faster than doing brute-force.

Another method which can be used in the case of cube maps, is filtering each face individually but including neighboring faces around it to make sure borders are (somewhat) handled. As can be seen in the pictures, there is no way to fill in the corners around the center face.

Finally, the filtering can be performed in the frequency domain (S2kit has sample source code to do this).

Filtered Environment Maps

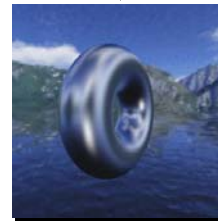
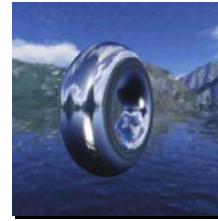
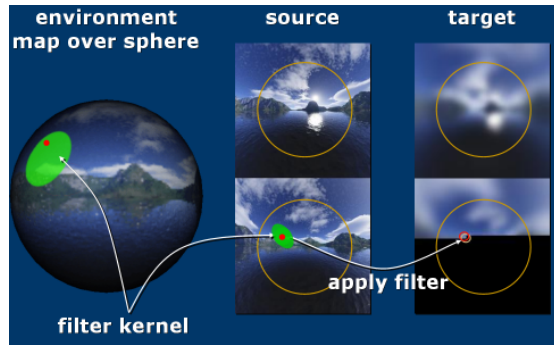
- Theory
- Diffuse Reflections
 - Spherical Harmonics
- Glossy Reflections
 - Prefiltering
 - On-the-fly Filtering
- Implementation and Production Issues



SIGGRAPH2006

We will first consider various issues involved with the use of environment maps to render mirror and diffuse reflections. Next we will discuss using spherical harmonics to represent lighting on diffuse surfaces (irradiance environment maps). Finally we will discuss implementation and production issues related to diffuse and mirror reflections of environment maps.

Filtered Environment Maps: Filtering Process



Environment Map Filtering

- ⇒ kernel shift-variant in texture space (for all: cube map, sphere map, etc.)
- ⇒ convolution is expensive

As said before, filtering can be expensive, which is mainly due to the shift-variant filter in texture space.

Filtered Environment Maps: Dynamic Glossy Reflections

- If environment changes, want to create and filter environment map **dynamically**.
[Kautz00] [Ashikhmin03][Kautz04]
 - Render environment into cube/parabolic map
 - Filter environment using graphics hardware
 - Box Filtering
 - Actual Convolution



But if a scene is changing dynamically, we also want dynamic reflections.

Easy solution: render scene into a cube map, and filter it using hardware.

Dynamic Glossy Environment Maps: Box-Filtering

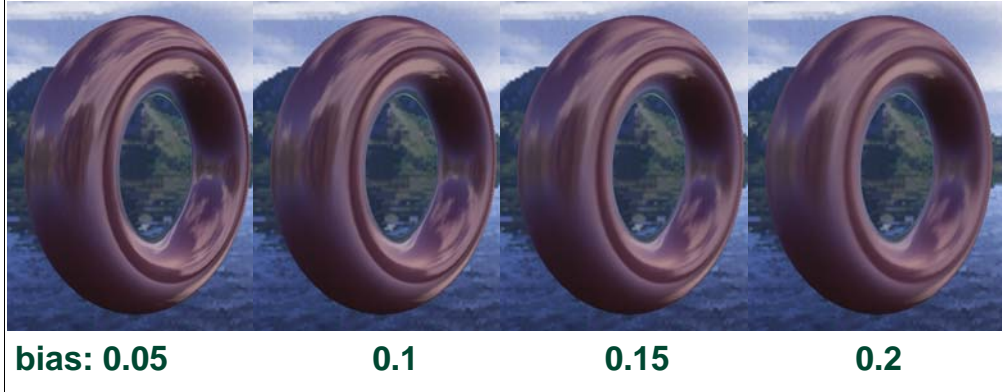


[Ashikhmin03][Kautz04]

The actual filter that is used to filter an environment map is not easy to tell for a viewer, so why not just use an inexpensive box-filter and store a mip-mapped cube map.

Dynamic Glossy Environment Maps

- Box filtering is supported by hardware (automipmap)
- Select glossiness: globally (**GL_EXT_lod_bias**) or per pixel (**texCUBEbias()**)



The filtering can be done automatically using the `auto_mipmap` feature of GPUs. Glossiness can then be selected on a per-object level (`EXT_lod_bias`) or by changing the LOD level per-pixel.

Dynamic Glossy Environment Maps



Phong: $N = 225$

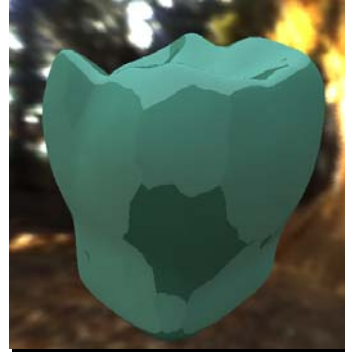


Box: bias = 2.25

Dynamic Glossy Environment Maps: Naïve implementation



LOD bias = 0.0



LOD bias = 9.4

Problem: cube faces are filtered independently

Solution: filtering **with border!** → slower again

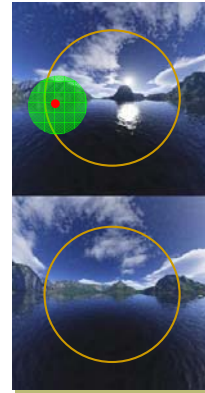


Unfortunately, there is one big drawback, the filtering is not done with borders taken into account, which means that the faces boundaries will show up at blurrier levels of the MIP-map!

Now easy way around this, other than slower filtering with borders again! → Change to different parameterization

Dynamic Glossy Environment Maps: Parabolic Maps

- Parabolic maps can include borders
 - GPU filtering is easier
 - Here the border is shown, and one can see that a filter kernel can extend outside the area of the hemisphere.
- Disadvantages:
 - Need to write own lookup into parabolic maps



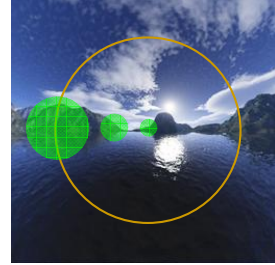
Parabolic maps can be used instead, where border can be used.

Problem, one needs to write a lookup into the parabolic map, which isn't that complicated, but it does cost a few instructions.

The other problem is that when a constant filter kernel is mapped to the center of a parabolic map, it is much smaller than at the boundaries. Can be fixed with [Kautz00]

Dynamic Glossy Environment Maps: Parabolic Maps

- Disadvantages:
 - Kernel becomes spatially varying
- Fix with [Kautz00]:
 - Filter with smallest & biggest kernel
 - Blend between two versions to get “correctly” filtered result
 - Can choose filter (e.g. Phong, approximations, box, ...)



Problem, one needs to write a lookup into the parabolic map, which isn't that complicated, but it does cost a few instructions.

The other problem is that when a constant filter kernel is mapped to the center of a parabolic map, it is much smaller than at the boundaries. Can be fixed with [Kautz00] by filtering with smallest & largest kernel and then interpolating between them to give “correct” kernel

Filtered Environment Maps

- Other techniques
 - Unified Approach To Prefiltered Environment Maps (using parabolic maps) [Kautz00]
 - Frequency-Space Environment Map Rendering [Ramamoorthi02]
 - Fast, Arbitrary BRDF Shading [Kautz02]



Filtered Environment Maps – Production Issues

- Distant Lighting Assumption
 - All techniques assume a distant environment
 - Looks odd, when object moves around and reflections don't change!
- Solutions
 - Static scene:
 - Sample incident light on grid (diffuse → SH) and interpolate
 - Half-Life 2 uses an artist-defined sampling of incident light

The assumption of distant lighting can be disturbing, when an object moves around within an environment.

For static scenes, the incident lighting can be precomputed (e.g. on a grid, e.g. in SH for diffuse reflections) and interpolated.

Half-Life 2 places samples of environment maps throughout the scene; the locations are controlled by the artist.

Filtered Environment Maps – Production Issues

- Dynamic scenes
 - Distant lighting not main issue
 - Need to re-generate environment map anyway!
 - Sample incident light on-the-fly
 - Render scene into cube map
 - On older hardware: 6 rendering passes
 - With DirectX 10 can do in **one** pass (geometry shaders)
 - How to do filtering, see before...



In dynamic scenes, one would like to regenerate the environment map. In case of cube maps that requires 6 passes, one for each face! This is very costly, and therefore not done on current hardware.

Next generation hardware will allow to render this in one pass by using geometry shaders!

Filtered Environment Maps – Production Issues

- Cube Maps: hardware does not filter across faces!
 - I.e. a bilinear lookup will not take samples from adjacent faces
 - Idea: use texture borders
 - Problem: often not supported by hardware
 - Fix: make sure borders of faces are similar...
 - ATI's **CubeMapGen** does this



Another problem is that currently the GPU does not filter across cube map faces. That is a bilinear lookup at the border of a face will **not** lookup into the adjacent face!

Texture borders (GL) are usually not supported by GPUs. ATI's **CubeMapGen** tool fixes this by making the borders of faces more similar.

Filtered Environment Maps – Production Issues

- HDR compression
 - For realistic effects, environment maps need to be high-dynamic range
 - So far no good compression for float textures
 - Solution: see this year's papers program!



Environment maps should be high-dynamic range to achieve realistic effects, but there are no compression techniques yet.

See this year's technical program for a solution.

Filtered Environment Maps – Production Issues

- So, what algorithm should I use then?

→ depends



Filtered Environment Maps – Production Issues

- On-the-fly vs. Prefiltering
 - Prefiltering can achieve higher quality reflections
 - But: not quite clear how much better Cosine-lobe filtering vs. box-filtering is
 - Prefiltering is slower
 - On-the-fly
 - Problem: filtering individual faces when using cube maps
 - wrong results
 - Parabolic maps: need some more shader instructions



SIGGRAPH2006

Filtered Environment Maps – Production Issues

- Spatially-Varying BRDFs?
 - Store (pre-)filtered environment map as mip-mapped cube map/parabolic map
 - Per-pixel roughness → select level l of mip-map
 - Want to make sure not to exceed maximum LOD l_{max} (determined by hardware to avoid aliasing)
 - Clamp mip-level “by hand” and then do a texCUBElod instruction (pre-PS3.0: use texCUBEbias)



SIGGRAPH2006

Reflectance Rendering with Environment Map Lighting

- Environment Maps
- Filtered Environment Maps
 - Diffuse Reflections
 - Glossy Reflections
- Anti-Aliasing
- Precomputed Radiance Transfer



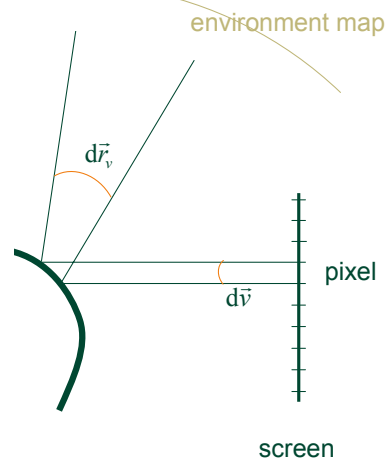
Now we will discuss issues related to anti-aliasing when rendering reflection models with environment maps.

Environment Map Anti-Aliasing

- Aliasing due to curved surfaces

- Envmap may be minified
- Need to anti-alias it

- Done automatically by the GPU!

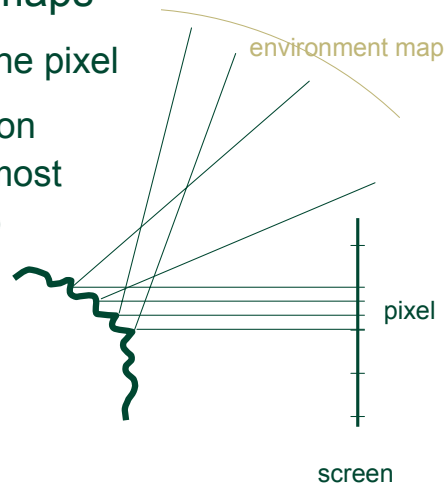


Minification as well as magnification may occur when rendering with environment maps. For example, a convex, curved surface can reflect light from a larger area of the environment map towards a single pixel on the screen. To avoid aliasing artifacts, the environment map needs to be filtered, just like any other texture needs to be filtered in this case. GPUs perform this filtering automatically, and the programmer does not need to deal with it explicitly.

Environment Map Anti-Aliasing

- Aliasing due to bump maps

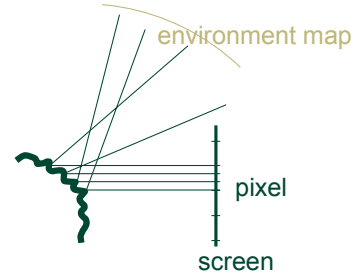
- Multiple bumps within one pixel
- Accurate filtered reflection is nearly impossible (almost random reflected views)



The situation is more difficult, when rendering environment maps together with bump maps. Underneath a single pixel, there might be several bumps that are all reflecting in different directions. Accurate filtering is almost impossible in this situation (unless very heavy super-sampling is used, which is not an option with current GPUs).

Environment Map Anti-Aliasing

- [Schilling97]
 - Use covariance matrix of distribution of normals (for mip-maps of bump map)
 - Change derivatives in cube map lookup based on covariance value
 - An implementation using TEXDD (supply derivatives by hand) is interesting, but hasn't been tried...



A. Schilling proposed to approximate the variation of normals underneath a screen pixel and then to use that knowledge when looking up the environment map. He proposes to store a covariance matrix of the normals underneath each texel of the bump map in a MIP-mapped fashion (i.e., at coarser resolution, the area taken into account becomes bigger). The covariance matrix is then used to perform an anisotropic texture lookup into the environment map. Please see the paper for details.

It might be feasible now to perform all the necessary calculations in a pixel shader. The anisotropic lookup can be performed with the TEXDD instruction (explicit derivatives needed). Nobody has tried this yet, but it does seem interesting.

Reflectance Rendering with Environment Map Lighting

- Environment Maps
- Filtered Environment Maps
 - Diffuse Reflections
 - Glossy Reflections
- Anti-Aliasing
- Precomputed Radiance Transfer



Finally, we discuss a special class of rendering algorithms for general lighting, precomputed radiance transfer, in the context of various reflection types.

Precomputed Radiance Transfer

- So far: no self-shadowing taken into account
- But is much more realistic
- Want to do it in **real-time** for changing illumination



No Shadows



Shadows

Precomputed Radiance Transfer

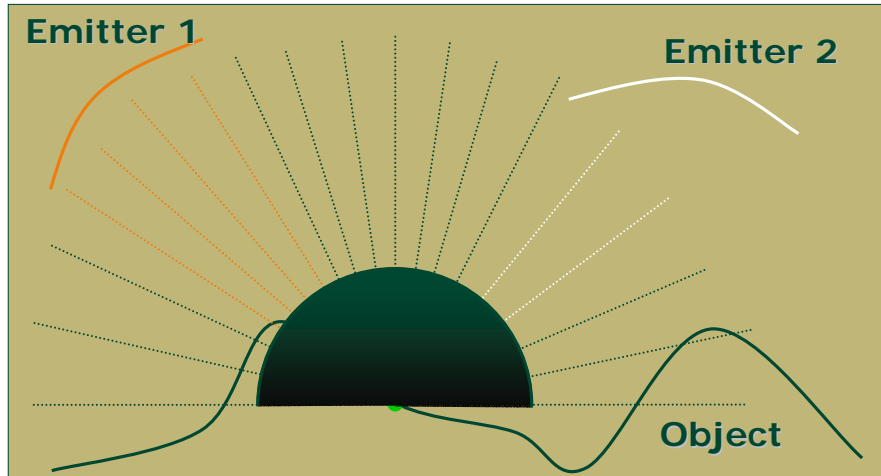
- Diffuse Reflection [Sloan02]
- Other BRDFs
- Implementation and Production Issues



Another method for rendering environment map lighting, which also takes other effects such as self-shadowing, interreflections and subsurface scattering into account, is precomputed radiance transfer (PRT). We will first discuss using PRT on Lambertian surfaces, followed by more general reflection models, and finally implementation and production issues.

Global Illumination (Reflectance Equation)

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

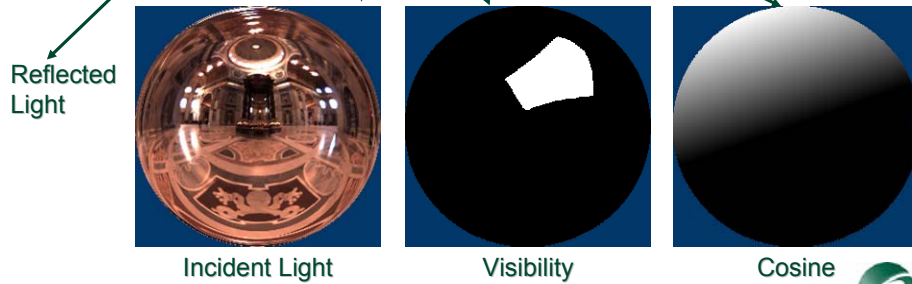


To compute exit radiance from a point p , we need to integrate all incident lighting against the visibility function and the diffuse BRDF (dot-product between the normal and the light direction).

Precomputed Radiance Transfer Reflectance Equation

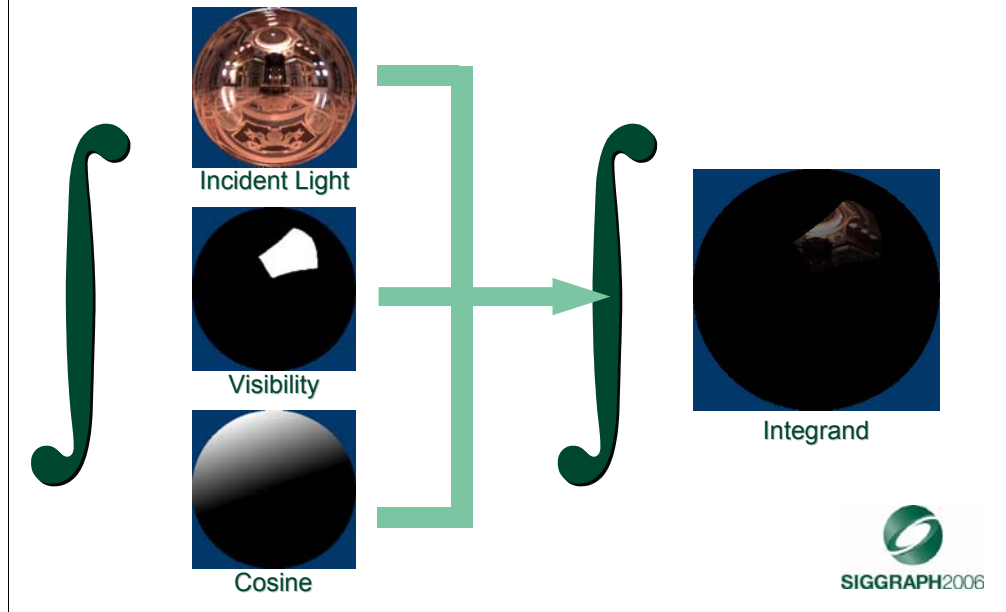
- Math:

$$L_{e,p}(\vec{v}) = \int_{\Omega} L_{env}(\vec{l}) V_p(\vec{l}) \max(\vec{n}_p \cdot \vec{l}, 0) d\vec{l}$$



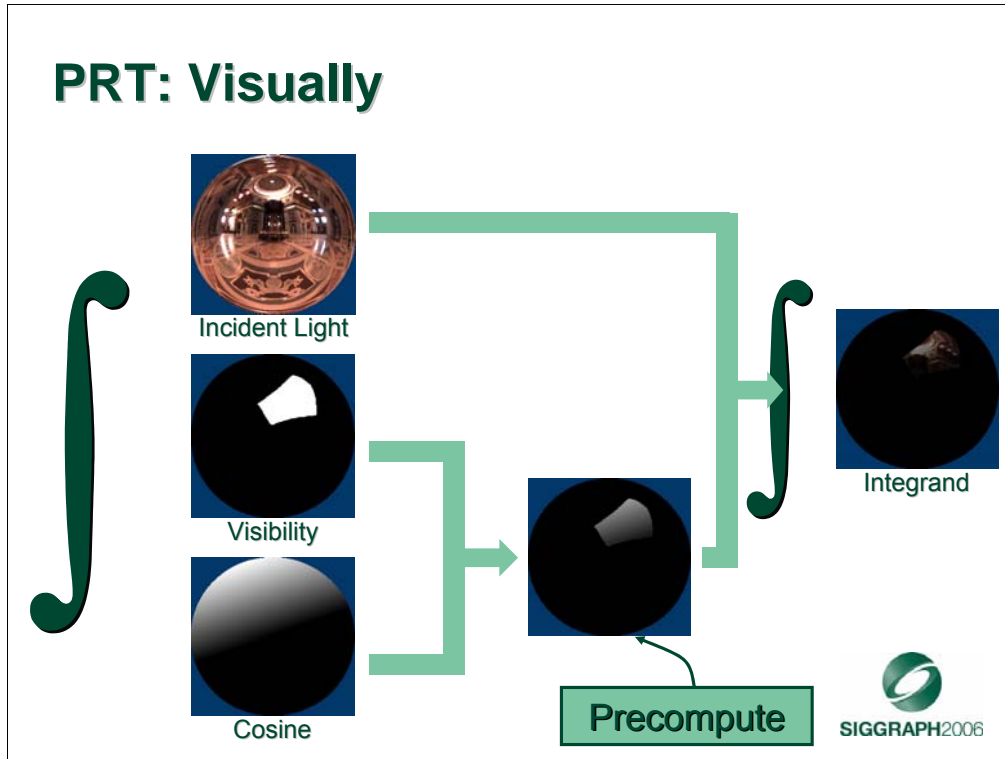
Computation written down more accurately.

Reflectance Equation: Visually



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

PRT: 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.

PRT

- Questions 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.

PRT

Reflectance Equation: Rewrite

- Math:

$$L_{e,p}(\vec{v}) = \int L_{env}(\vec{l}) \underbrace{V_p(\vec{l}) \max(\vec{l} \cdot \vec{n}_p, 0)}_{T_p(\vec{l})} d\vec{l}$$

- Rewrite with $T_p(\vec{l}) = V_p(\vec{l}) \max(\vec{l} \cdot \vec{n}_p, 0)$ 

- This is the **transfer function**

— Encodes:

- Visibility
- Shading



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

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.

PRT

Reflectance Equation: Rewrite

- Math:



$$L_{e,p}(\vec{v}) = \int L_{env}(\vec{l}) V_p(\vec{l}) \max(\vec{l} \cdot \vec{n}_p, 0) d\vec{l}$$

- Plug new $T_p(\vec{l})$ into Equation:

$$L_{e,p}(\vec{v}) = \int \underbrace{L_{env}(\vec{l})}_{\text{into SH}} \underbrace{T_p(\vec{l})}_{\text{into SH}} d\vec{l}$$

light function: L_{env} transfer: T_p


⇒ project **lighting** and **transfer** into SH

Now, when we plug the new $T_p(s)$ 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.

PRT – Evaluating the Integral

- The integral


$$L_{e,p}(\vec{v}) = \int L_{env}(\vec{l}) T_p(\vec{l}) d\vec{l}$$

becomes

$$L_{e,p}(\vec{v}) = \sum_k^n l_{env,k} t_{p,k}$$

"light vector" →
"transfer vector" →

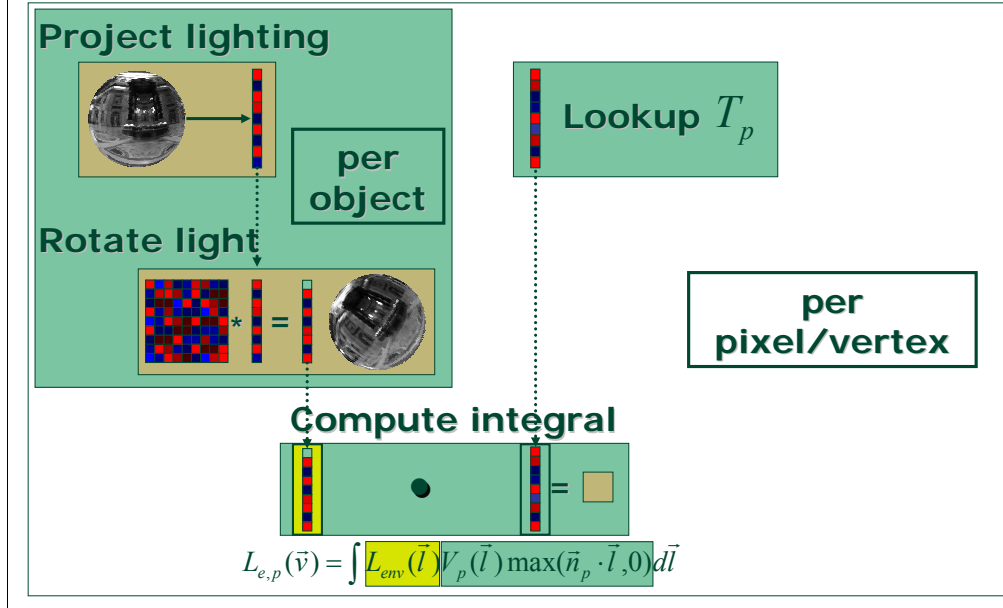
A *simple* dot-product!!!!

(All examples use n=25 coefficients)



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

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



Unshadowed



Shadowed



PRT Results



PRT Rendering

- Reminder:

$$L_{e,p} = \sum_k^n l_{env,k} t_{p,k}$$

- Need lighting coefficient vector:

$$l_{env,k} = \int L_{env}(\vec{l}) y_k(\vec{l}) d\vec{l}$$

- Compute every frame (if lighting changes)
- Projection can e.g. be done using Monte-Carlo integration, or on GPU



SIGGRAPH2006

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).

PRT Rendering – Dynamic Lighting

- Sample dynamic lighting:
 - Render scene from center of object p
 - 6 times: for each cube face
 - Compute lighting coefficients (SH/Haar):

$$l_{env,k} = \int \text{img} \cdot \text{hemisph}$$


- No need to rotate lighting then



PRT Rendering

- 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

So far, 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.

PRT Rendering

- 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.
  }
}
```



SIGGRAPH2006

What does this mean?

- **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?

- Negative:
 - Models are assumed to be *static*



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).

PRT – Precomputation

- Integral

$$t_{p,k}^0 = \frac{\rho_d}{\pi} \int V(p \rightarrow \vec{l}) \max(\vec{n}_p \cdot -\vec{l}, 0) y_k(\vec{l}) d\vec{l}$$

evaluated numerically with e.g. ray-tracing:

$$t_{p,k}^0 = \frac{4\pi}{N} \frac{\rho_d}{\pi} \sum_{j=0}^{N-1} V(p \rightarrow \vec{l}_j) \max(\vec{n}_p \cdot -\vec{l}_j, 0) y_k(\vec{l}_j)$$

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

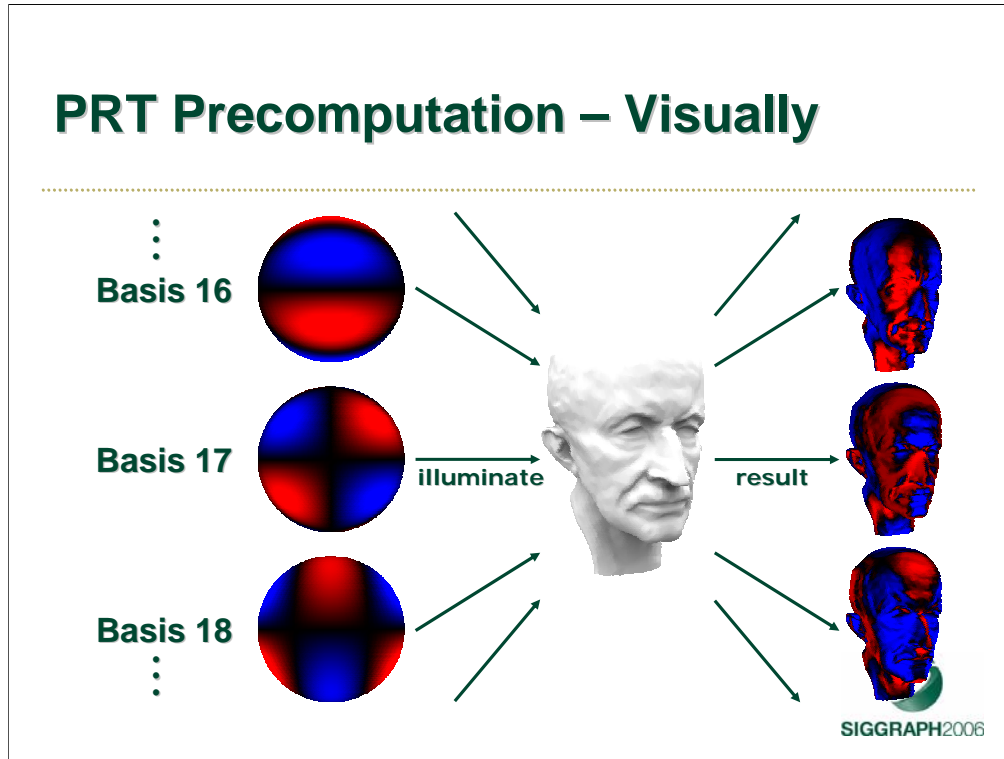
The main question is how to evaluate the integral. We will evaluate it numerically using Monte-Carlo integration. This basically means, that we generate a random (and uniform) set of directions s_j , which we use to sample the integrand. All the contributions are then summed up and weighted by $4\pi/(\#\text{samples})$.

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

 Aside: 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

PRT 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.

PRT Precomputation – Code

```
// 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.

PRT – Basis Functions

- Original technique uses Spherical Harmonics
- Wavelets are a good alternative [Ng03]
 - Better quality for a given amount of coefficients
 - [Ng03] cannot be directly done on GPU
 - Using compression [Sloan03], most of the computation can be done on the GPU



SIGGRAPH2006

Precomputed Radiance Transfer

- Diffuse Reflection
- Other BRDFs
- Implementation and Production Issues



We will now discuss methods for using PRT with more general reflection models.

Precomputed Radiance Transfer

- General BRDFs

- Ongoing subject of research, see this year's papers.

- Difficult due to additional BRDF term:

$$L_{e,p}(\vec{v}) = \int L_{env}(\vec{l}) V_p(\vec{l}) f(\omega(\vec{l}), \omega(\vec{v})) \max(\vec{l} \cdot \vec{n}_p, 0) d\vec{l}$$

- Could say: like envmap rendering, but with visibility

- Won't go into details in this course



SIGGRAPH2006

General BRDFs are ongoing research. The difficulty arises from the use of an arbitrary BRDF $f()$. As before, the BRDF requires local coordinates, and the function $\omega()$ converts from global to local coordinates. The treatment of arbitrary BRDFs is outside the scope of this course. The audience is referred to last year's course on PRT.

Precomputed Radiance Transfer

- General BRDFs
 - Commonalities of various techniques:
 - Need to store more data
 - Use compression
 - Slower run-time than pure diffuse BRDF
 - Problems with these techniques:
 - Still too slow for games
 - Question: *do glossy reflections need self-shadowing?*



Precomputed Radiance Transfer

- Diffuse Reflection
- Other BRDFs
- Implementation and Production Issues



We will now cover implementation and production issues relating to the techniques we have just discussed.

PRT: Production Issues

- Albedo maps
 - Could be stored as part of the transfer coefficients
 - But often textures are sampled at higher resolution
 - Better: multiply albedo with transfer at run-time
 - If inter-reflections are included this is tricky, since they depend on albedos.



SIGGRAPH2006

PRT: Production Issues

- Normal maps
 - If normal maps are included in precomputation
 - works just fine
 - But
 - Normal maps often at different resolution than PRT maps
 - Solution:
 - Normal Mapping for Precomputed Radiance Transfer [Sloan06]



PRT: Production Issues

- Should I use PRT?
 - If you are using light maps anyway
 - PRT is not that different really!
 - If you are using dynamic lighting (point lights ...)
 - More difficult to answer
 - Combination of the two is unexplored...

