Naty Hoffman

Electronic Arts

naty@io.com

GameDevelopers Conference

Make Better Games.

Hi, I’m Naty Hoffman and today I’ll be talking about Advanced Real-Time Illumination Techniques.

Please turn your cellphones and pagers off, and don’t forget to fill in the feedback form and hand it in after the talk.

Outline

- Game Graphics– What's Missing?
  - The Rendering Equation
  - Simple Lighting
  - The Evil Ambient Term
  - Local Vs. Global Illumination
- Advanced Techniques
  - Polynomial Texture Maps
  - Spherical Harmonic Lighting
  - Precomputed Transfer Functions

Game Graphics Today

- Per-pixel lighting
- Complex materials
- Shadows from geometry
- Effects: depth of field, light halos, etc.
- Extremely detailed

What's Missing?

- The lighting model in current use is very limited
- There are several deficiencies which, if removed, would increase realism

GameDevelopers Conference

Make Better Games.

I will show some techniques which can be done on today's hardware which address some of these deficiencies. They are not commonly used in games yet though.

The Rendering Equation

$$L_{\text{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)} L_{\text{in}}\left(x,\vec{\Psi}\right) f_{\text{r}}\left(x,\vec{\Theta},\vec{\Psi}\right) \vec{N}(x)\cdot\vec{\Psi}\, d\omega_{\Psi}$$

GameDevelopers Conference

Make Better Games.

Let's start this off completely on the wrong foot – with a big complicated equation. This is the rendering equation, first formulated by James Kajiya in 1986. This form is slightly different from the one in Kajiya's paper: the notation is different and the emissive term has been removed.

The Rendering Equation

$$L_{\mathrm{ex}}(x,\vec{\Theta}) = \int_{\Omega(x)} L_{\mathrm{in}}(x,\vec{\Psi})\, f_{\mathrm{r}}(x,\vec{\Theta},\vec{\Psi})\, \vec{N}(x)\cdot\vec{\Psi}\, d\omega_{\Psi}$$

GameDevelopers Conference
Make Better Games.

Lowercase “x” is the point we are currently shading,…

The Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)} L_{\mathrm{in}}\left(x,\vec{\Psi}\right) f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}\right)\vec{N}(x)\cdot\vec{\Psi}\,d\omega_{\Psi}$$

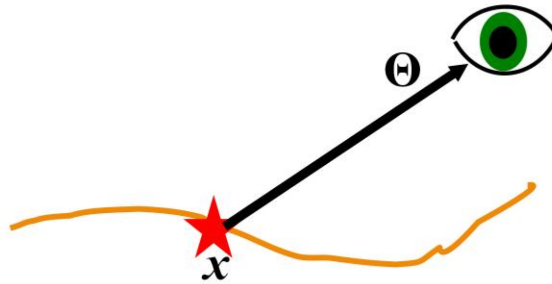
GameDevelopers Conference

Make Better Games.

…and the uppercase L’s represent radiance **<u>in</u>**coming or **<u>ex</u>**iting that point.

The Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)} L_{\mathrm{in}}\left(x,\vec{\Psi}\right) f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}\right) \vec{N}(x)\cdot\vec{\Psi}\, d\omega_{\Psi}$$



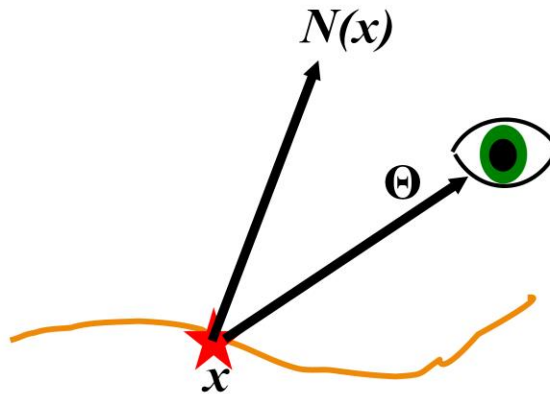
GameDevelopers Conference

Make Better Games.

Uppercase theta is the direction to the eye…

The Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)= \int_{\Omega(x)} L_{\mathrm{in}}\left(x,\vec{\Psi}\right) f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}\right) \vec{N}(x) \cdot \vec{\Psi}\, d\omega_{\Psi}$$



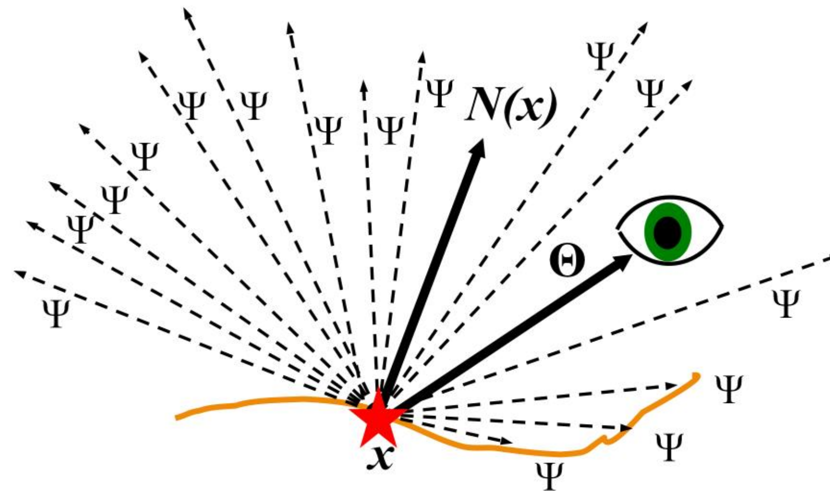
GameDevelopers Conference

Make Better Games.

…N(x) is the surface normal at x…

The Rendering Equation

$$L_{\text{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)} L_{\text{in}}\left(x,\vec{\Psi}\right) f_{\text{r}}\left(x,\vec{\Theta},\vec{\Psi}\right) \vec{N}(x)\cdot\vec{\Psi}\, d\omega_{\Psi}$$



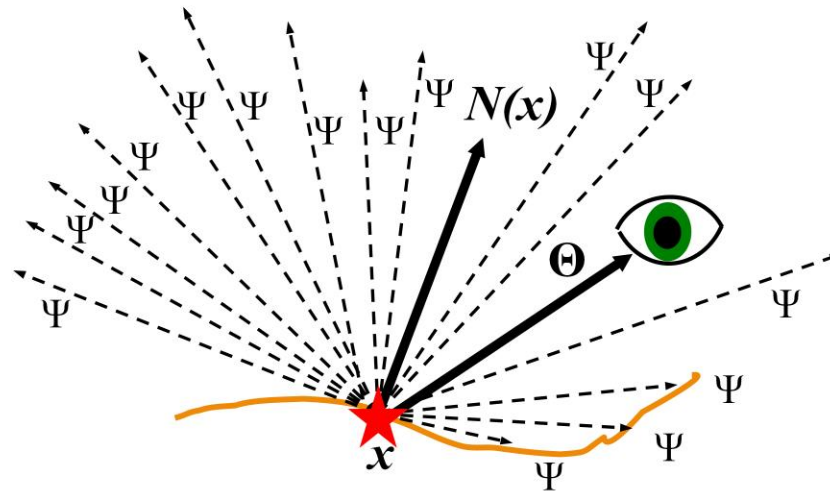
GameDevelopers Conference

Make Better Games.

…and uppercase psi is the incoming direction, which is swept through the hemisphere around N(x) as a variable of integration.

The Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)}L_{\mathrm{in}}\left(x,\vec{\Psi}\right)f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}\right)\vec{N}(x)\cdot\vec{\Psi}\,d\omega_{\Psi}$$



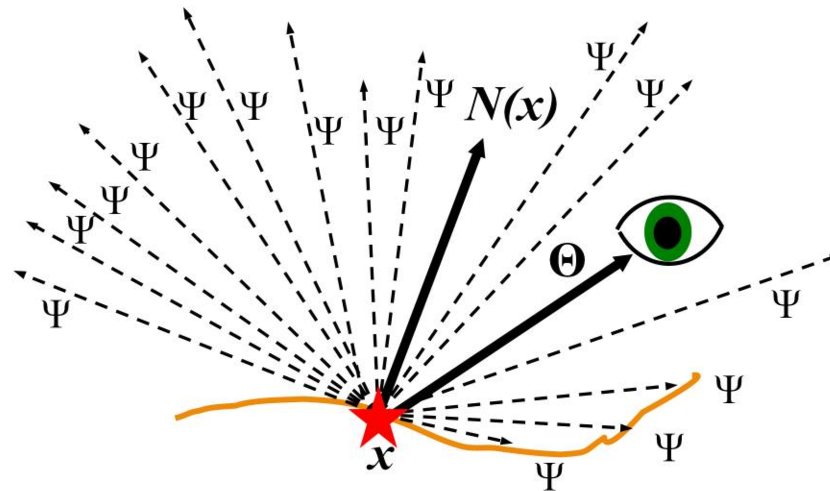
GameDevelopers Conference

Make Better Games.

The hemisphere around N(x) is uppercase omega(x)).

The Rendering Equation

$$L_{ex}\left(x,\vec{\Theta}\right)= \int_{\Omega(x)} L_{in}\left(x,\vec{\Psi}\right) f_r\left(x,\vec{\Theta},\vec{\Psi}\right) \vec{N}(x)\cdot\vec{\Psi}\, d\omega_{\Psi}$$

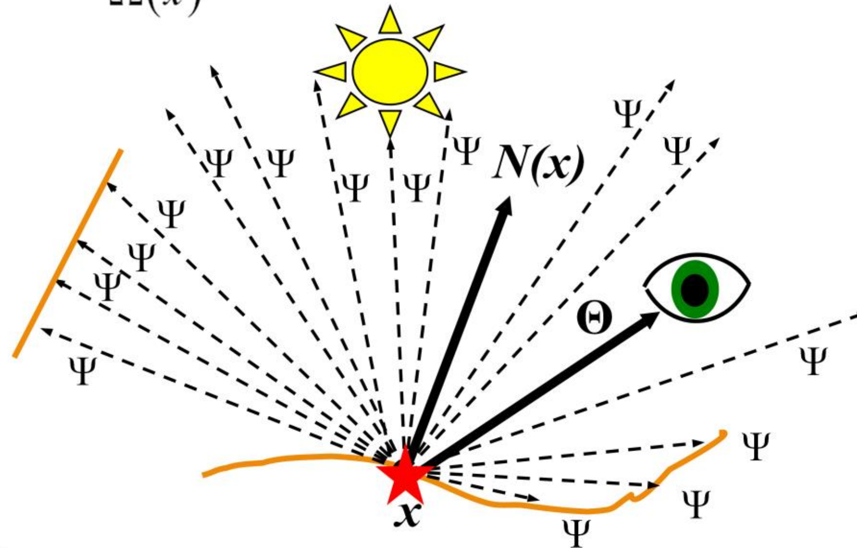


GameDevelopers Conference
Make Better Games.

The function fr() is the BRDF (Bidirectional Reflectance Distribution Function, which relates incoming and outgoing light intensities at various directions).

The Rendering Equation

$$L_{\text{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)} L_{\text{in}}\left(x,\vec{\Psi}\right)f_{\text{r}}\left(x,\vec{\Theta},\vec{\Psi}\right)\vec{N}(x)\cdot\vec{\Psi}\,d\omega_{\Psi}$$

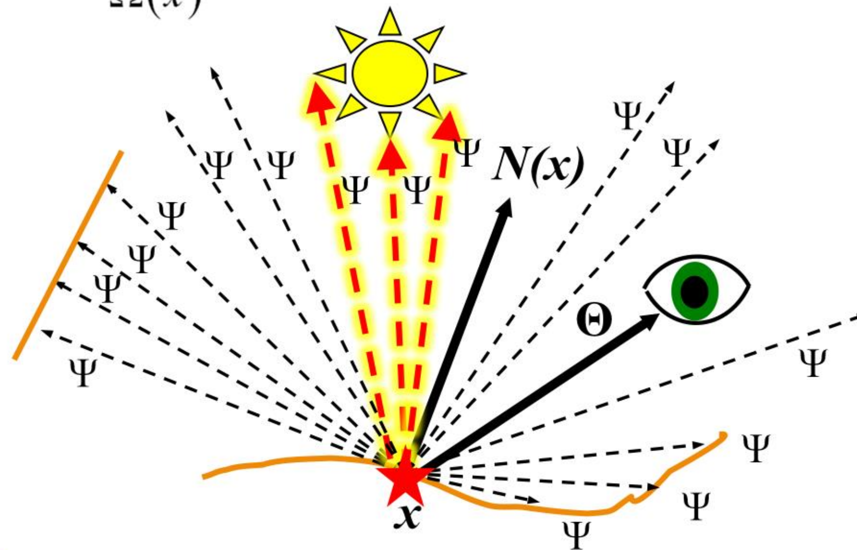


GameDevelopers Conference
Make Better Games.

Note that there are Psi directions with some incoming light all over the hemisphere.

The Rendering Equation

$$L_{\text{ex}}\left(x,\vec{\Theta}\right)=\int\limits_{\Omega(x)} L_{\text{in}}\left(x,\vec{\Psi}\right) f_{\text{r}}\left(x,\vec{\Theta},\vec{\Psi}\right)\vec{N}(x)\cdot\vec{\Psi}\,d\omega_{\Psi}$$



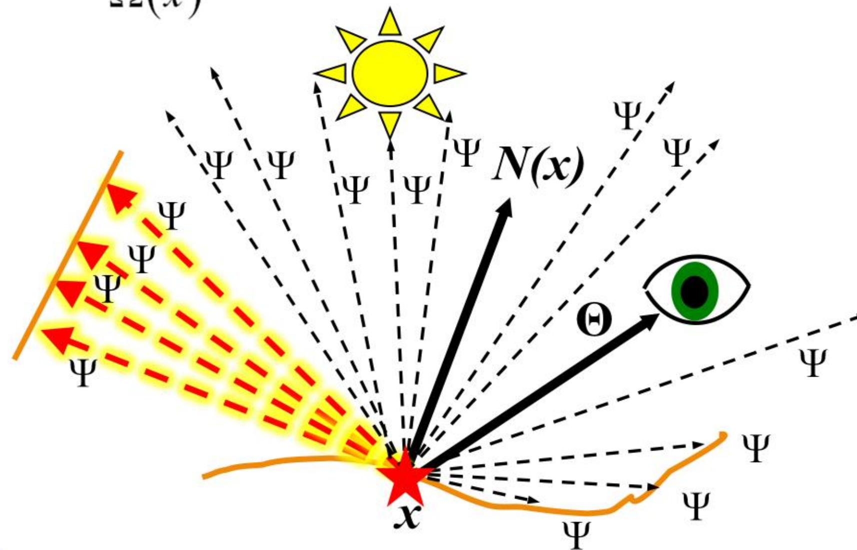
GameDevelopers Conference

Make Better Games.

Some of these directions come from area light sources (all real-world light sources cover some angular area, or solid angle. Their illumination is caused by a combination of their radiance and solid angle)…

The Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)} L_{\mathrm{in}}\left(x,\vec{\Psi}\right) f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}\right) \vec{N}(x)\cdot\vec{\Psi}\, d\omega_{\Psi}$$



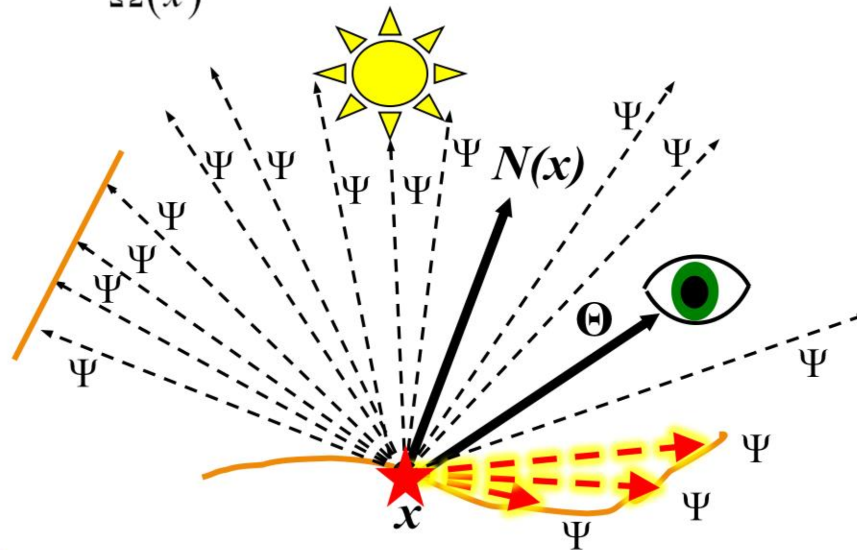
GameDevelopers Conference

Make Better Games.

…some from other objects…

The Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)=\int_{\Omega(x)} L_{\mathrm{in}}\left(x,\vec{\Psi}\right) f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}\right)\vec{N}(x)\cdot\vec{\Psi}\, d\omega_{\Psi}$$



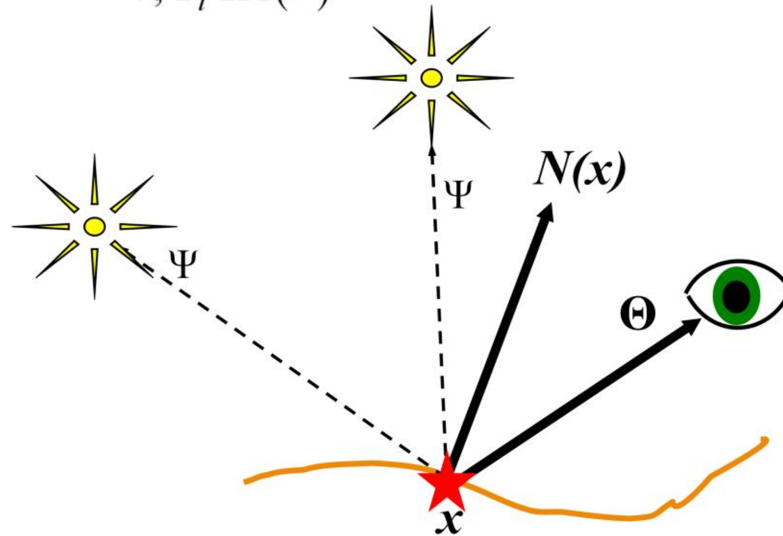
GameDevelopers Conference

Make Better Games.

…some from other parts of the same object. All these directions have some incoming light.

Real-Time Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)= \sum_{i,\vec{\Psi}_i\in\Omega(x)} \left(L\omega\right)_i f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}_i\right)\vec{N}(x)\cdot\vec{\Psi}_i + L_{\mathrm{am}}$$



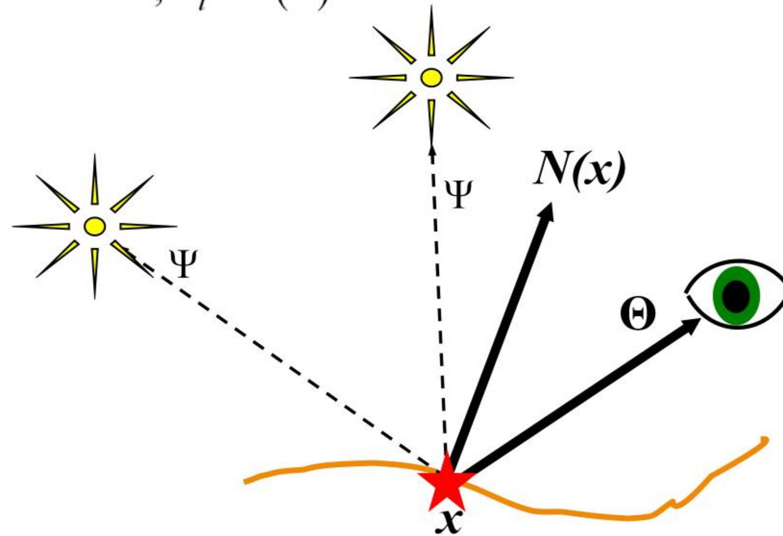
GameDevelopers Conference

Make Better Games.

This is the equation we currently use in games.

Real-Time Rendering Equation

$$L_{\text{ex}}\left(x,\vec{\Theta}\right)= \sum_{i,\vec{\Psi}_i\in\Omega(x)} (L\omega)_i \, f_{\text{r}}\left(x,\vec{\Theta},\vec{\Psi}_i\right) \vec{N}(x)\cdot\vec{\Psi}_i + L_{\text{am}}$$



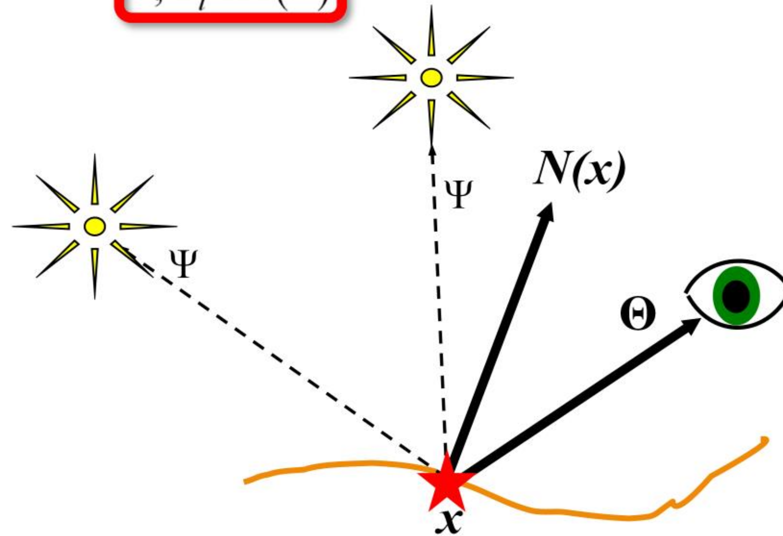
GameDevelopers Conference

Make Better Games.

We can have a general BRDF (there are many well-understood real-time techniques for going beyond the Phong lighting equation—outside the scope of this talk though)…

Real-Time Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)=\sum_{i,\vec{\Psi}_i\in\Omega(x)}\left(L\omega\right)_i f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}_i\right)\vec{N}(x)\cdot\vec{\Psi}_i+L_{\mathrm{am}}$$



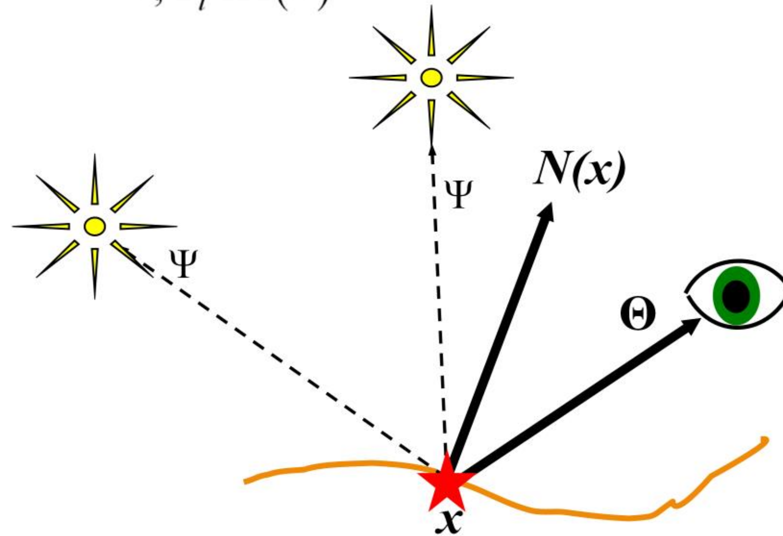
GameDevelopers Conference

Make Better Games.

…but now the incoming light is restricted to a handful of discrete directions. Each of these is an actual light source (no reflected light off objects) and these light sources are strange…

Real-Time Rendering Equation

$$L_{\mathrm{ex}}(x,\vec{\Theta}) = \sum_{i,\vec{\Psi}_i \in \Omega(x)} (L\omega)_i f_{\mathrm{r}}(x,\vec{\Theta},\vec{\Psi}_i)\vec{N}(x)\cdot\vec{\Psi}_i + L_{\mathrm{am}}$$



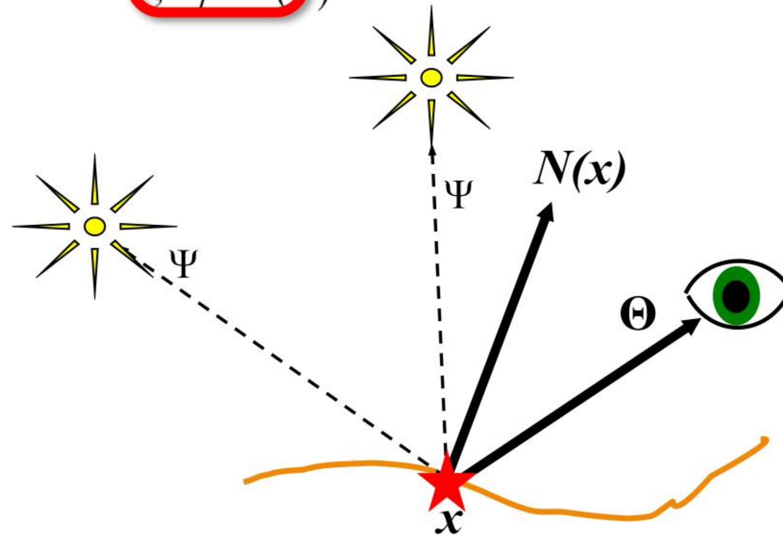
GameDevelopers Conference

Make Better Games.

…all their intensity (which is represented by l times omega) is squeezed into a single ray. This would mean their radiance is infinite, this is unlike any light source found in reality.

Real-Time Rendering Equation

$$L_{\text{ex}}\left(x,\vec{\Theta}\right)=\sum_{i,\vec{\Psi}_i\in\Omega(x)}\left(L\omega\right)_i f_{\text{r}}\left(x,\vec{\Theta},\vec{\Psi}_i\right)\vec{N}(x)\cdot\vec{\Psi}_i+L_{\text{am}}$$



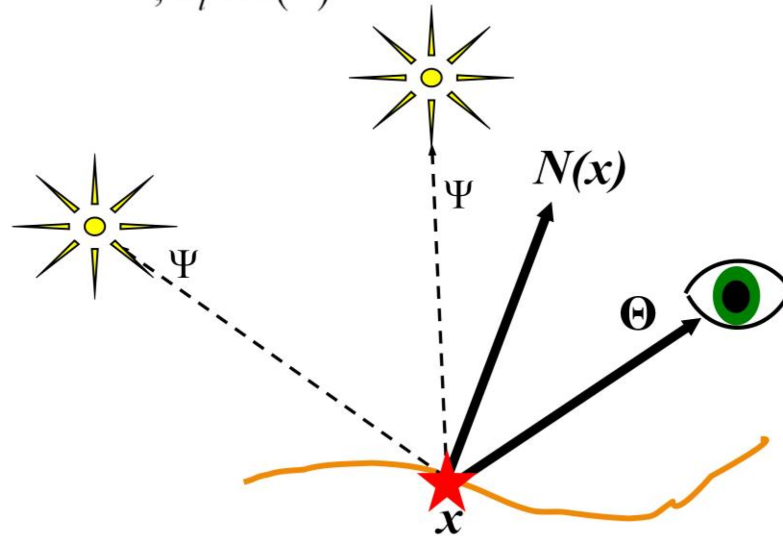
GameDevelopers Conference

Make Better Games.

This does have the advantage of simplifying the integral into a summation.

Real-Time Rendering Equation

$$L_{\mathrm{ex}}\left(x,\vec{\Theta}\right)=\sum_{i,\vec{\Psi}_i\in\Omega(x)}\left(L\omega\right)_i f_{\mathrm{r}}\left(x,\vec{\Theta},\vec{\Psi}_i\right)\vec{N}(x)\cdot\vec{\Psi}_i+L_{\mathrm{am}}$$



GameDevelopers Conference

Make Better Games.

The ambient factor is another issue. It is a constant radiance which is added to the result of the calculation.

We’ll get back to the ambient factor in a moment, but for now let me just make a statement…

Ambient Factor =

GameDevelopers Conference

Make Better Games.

Just to let you know where I stand…

Simple Lighting

- Real scenes have a rich *light environment*
  - A continuous function of radiance for each incoming direction
  - Real-time model assumes light environment is zero everywhere except for a handful of directions
- Environment Mapping is an Exception
  - But only for mirror-like surfaces
  - Hard to generalize to a large game world

Simple Lighting

- **This simplified light environment leads to several problems:**
  - **Can cause some parts of scene to be too dark since environment is mostly zero**
    - **Common cure (ambient) is worse than disease**
  - **'Delta Function' light sources**
    - **Physically implausible**
    - **Unrealistically sharp lighting**

GameDevelopers Conference

Make Better Games.

Delta functions are functions with zero support and a finite integral, which means that they have non-finite values.

Simple Lighting

- This simplified light environment leads to several problems:
  - Can cause some parts of scene to be too dark since environment is mostly zero
    - Common cure (ambient) is worse than disease
  - 'Delta Function' light sources
    - Physically implausible
    - Unrealistically sharp lighting
  - Extremely poor compared to real scenes

GameDevelopers Conference

Make Better Games.

Even in sunlit outdoor scenes, besides the one strong light source objects are also lit by the sky, reflections from the ground, etc.

And after the sun has set, or in overcast days, there is no direct light at all.

Simple Lighting

- This simplified light environment leads to several problems:
  - Can cause some parts of scene to be too dark since environment is mostly zero
    - Common cure (ambient) is worse than disease
  - ‘Delta Function’ light sources
    - Physically implausible
    - Unrealistically sharp lighting
  - Extremely poor compared to real scenes
    - Especially indoor scenes

GameDevelopers Conference

Make Better Games.

Indoor scenes are an especially poor match for the ‘handful of point lights’ scenario – in many cases there is no direct light at all.

Example

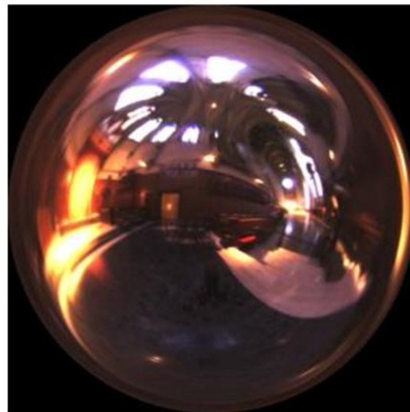
GameDevelopers Conference

Make Better Games.

Nothing in this room is lit by a direct light source. A small part is lit by the sky (itself indirect light from the sun) but most of it is lit by indirect light bouncing off other objects, walls, etc. An object in this room, to be rendered correctly, would need to use a complex light environment – a handful of point lights won't cut it.

Image-Based Lighting

- **Paul Debevec – light probes**
  - **From real scenes**



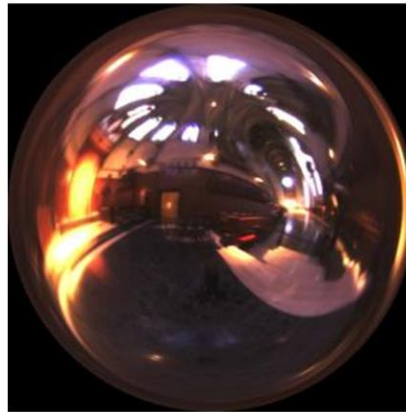
GameDevelopers Conference

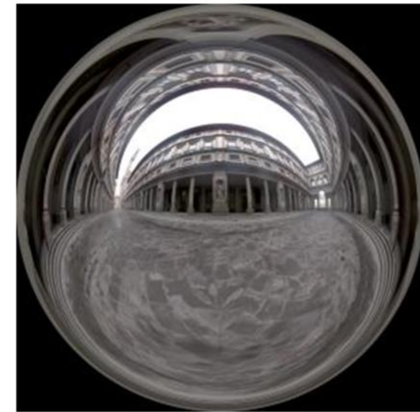
Make Better Games.

Paul Debevec at USC has been doing some amazing work recently, on capturing light environments from real-world scenes and using them to render synthetic objects. I recommend attending his talk this Saturday afternoon. The rendering techniques used in his original papers are non-real time…

Image-Based Lighting

- **Paul Debevec – light probes**
  - **From real scenes**
  - **High dynamic range**





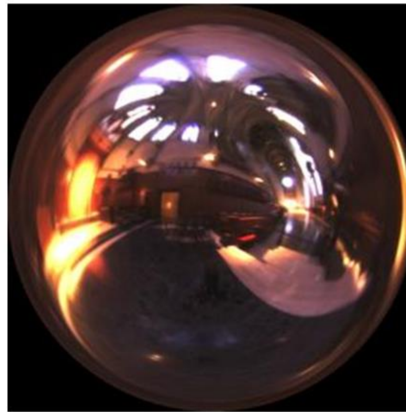
GameDevelopers Conference

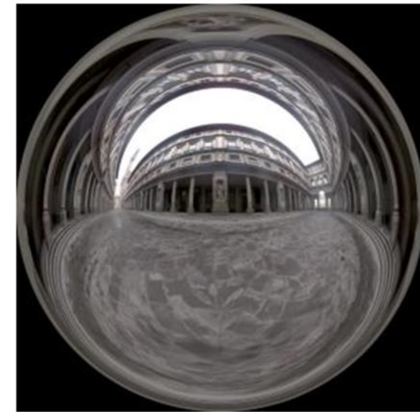
Make Better Games.

…but his capture techniques are important for anyone trying to add more realism to rendered scenes. These light environments are high dynamic range (radiance values vary a lot, as much as one to a million in some environments). This can be a problem with the 8-bit formats common in real-time rendering.

Image-Based Lighting

- **Paul Debevec – light probes**
  - From real scenes
  - High dynamic range





GameDevelopers Conference

Make Better Games.

The environments are captured as multiple photographic exposures of a mirrored ball. I hope these images convey the richness (if not the dynamic range) of real-world light environments and the hopelessness of trying to capture them with four point lights and an ambient term. Speaking of the devil…

The Evil Ambient Term

- Adding a constant incoming radiance to the real-time 'light environment' would be reasonable
  - Given that most of it is dark
- However, that is not what the ambient term does!
  - Adds to the result of the lighting calculation
  - Equivalent to all surfaces glowing…

The Evil Ambient Term

GameDevelopers Conference

Make Better Games.

Ambient factor is a non-physical hack. Can it be viewed as a reasonable approximation of indirect lighting? This image (from 3D Studio MAX help files) shows a model lit by completely diffuse lighting —the light environment is constant white in all directions. A correct rendering looks like this.

The Evil Ambient Term

GameDevelopers Conference

Make Better Games.

This is what the ambient term looks like. Not remotely a reasonable approximation even by “looks kinda OK if you squint” standards.

Local vs. Global Illumination

- **Local illumination**
  - **Only the point properties affect its color**

GameDevelopers
Conference

Make Better Games.

And the last pair of images brings us to another problem. The lighting algorithms in use in games today are all strictly ‘local illumination’. However, this leads to a loss in realism.

Local vs. Global Illumination

- **Local illumination**
  - Only the point properties affect its color
- **Global illumination**
  - The bouncing of light between objects in the scene is fully simulated
  - The color of each point is affected by many (all?) other points in the scene

GameDevelopers Conference

Make Better Games.

All the detail in the tank render came from global illumination effects. The rendering equation is recursive; each point's shading depends on other points. Thus global illumination solutions need to be iterative, not closed-form. Local illumination lacks this recursion. It's much faster, but at a large cost to realism.

Local vs. Global Illumination

- **Local illumination**
  - Only the point properties affect its color
- **Global illumination**
  - The bouncing of light between objects in the scene is fully simulated
  - The color of each point is affected by many (all?) other points in the scene

GameDevelopers Conference

Make Better Games.

Since local illumination ignores light bouncing to different parts of the scene, the resulting images are unrealistically dark in places. The common solution for this problem is… you guessed it, the ambient term. There are two exceptions to the 'local illumination' rule for games:…

Local vs. Global Illumination

- Local illumination
  - Only the point properties affect its color
- Global illumination
  - The bouncing of light between objects in the scene is fully simulated
  - The color of each point is affected by many (all?) other points in the scene
- Local + shadows
  - Games now do local + shadows.
  - But usually not soft shadows or shadows from per-pixel details (bumps)

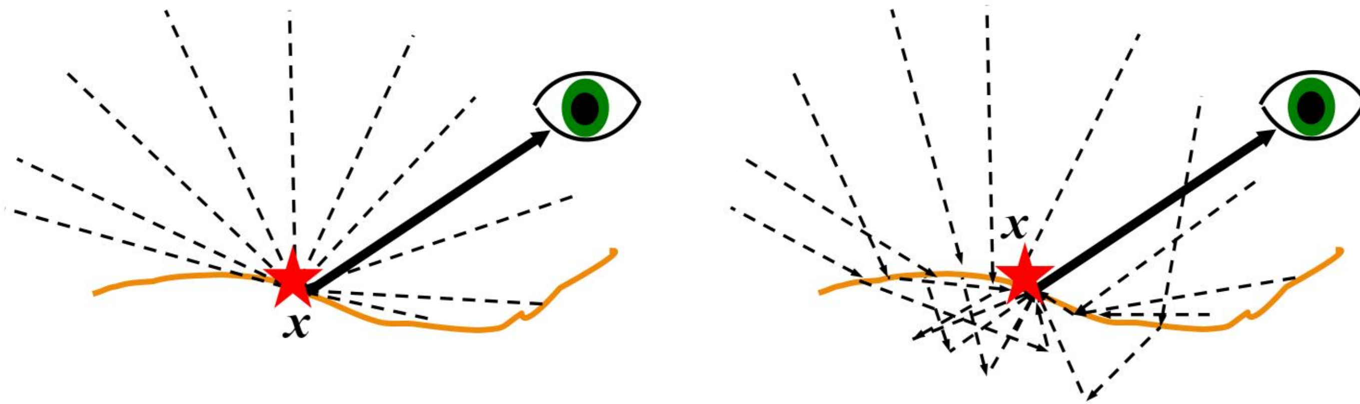
GameDevelopers Conference

Make Better Games.

…one is shadows, which is a global illumination effect. The other exception is static scenes, where for some years we have been running offline global illumination solutions and storing them in lightmaps. Neither of these is a complete solution.

Beyond the Rendering Equation

- **Subsurface scattering**



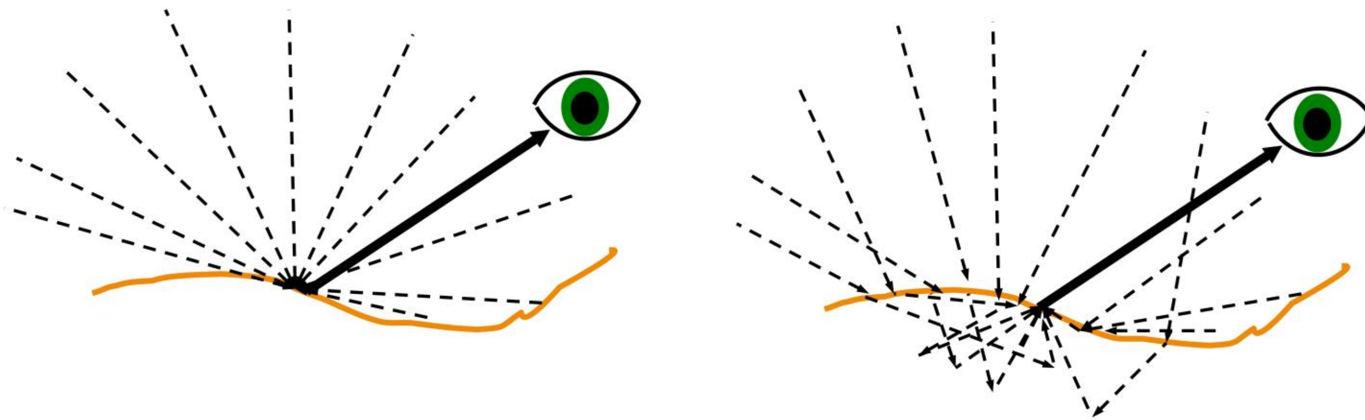
GameDevelopers Conference

Make Better Games.

The rendering equation itself doesn't cover all phenomena needed for photorealistic rendering. It assumes that outgoing light from a point ***x*** only depends on incoming light at ***x*** (on left). But in many materials (e.g. skin) light from other points is scattered under the surface and emerges at ***x*** (on right).

Beyond the Rendering Equation

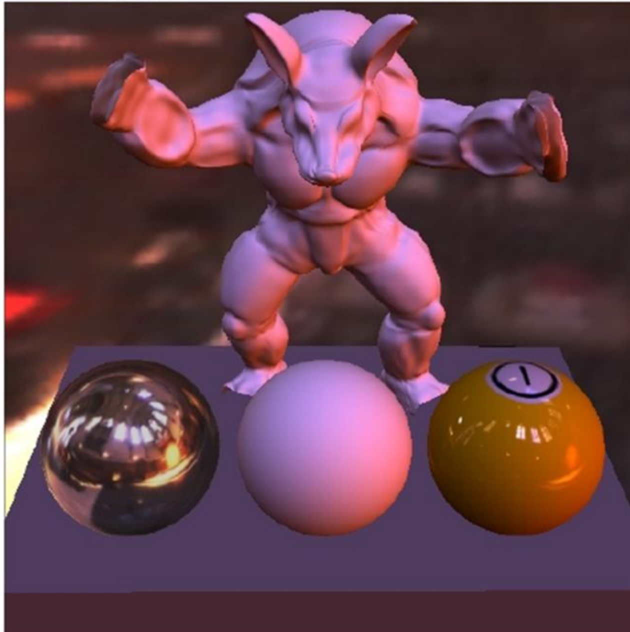
- **Subsurface scattering**



GameDevelopers Conference

Make Better Games.

These are phenomena even non-real-time rendering has a hard time coping with, but in some cases they can be important for realism.



GameDevelopers Conference

Make Better Games.

So, after we have seen what’s missing in the commonly used techniques and what we would like to do, here are some techniques which enable solving some of those problems.

Polynomial Texture Maps

- 2001: Malzbender, Gelb, Wolters (HP)
- Image-based technique
- Data can be extracted from real or virtual surfaces

GameDevelopers Conference

Make Better Games.

The first advanced technique I'll present is Polynomial Texture Maps (PTMs), first presented at SIGGRAPH in 2001. This is an image-based technique, using a series of images of a surface (lit from different directions) as input. These may be photographs of real surfaces, or offline renders of virtual surfaces.

Polynomial Texture Maps

- 2001: Malzbender, Gelb, Wolters (HP)
- Image-based technique
- Data can be extracted from real or virtual surfaces
- Full global illumination
  - Can handle subsurface scattering
  - Only diffuse surfaces
- No light environments
  - Point / directional lights only
  - Easier to integrate into existing games

GameDevelopers Conference
Make Better Games.

PTMs can handle full global illumination (even subsurface scattering), but are limited to view-independent, or diffuse surfaces. The point/directional light limitation has its upside – it makes it easier to integrate PTMs into any games that use per-pixel lighting, without having to change how lights are defined.

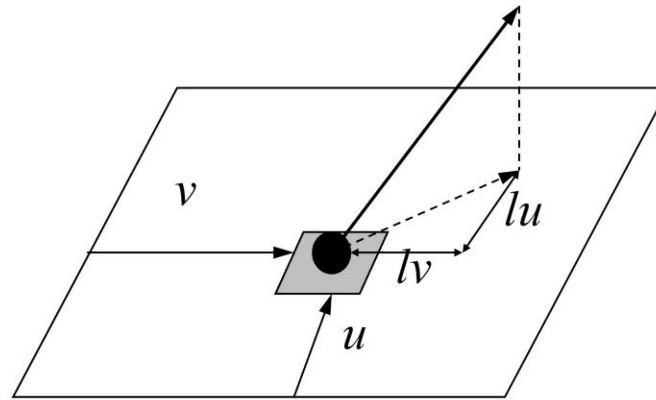
Polynomial Texture Maps

- **Basic Idea:**
  - **For each point, capture illumination as a function of local light direction.**
  - **Use a bivariate quadratic polynomial**
    - **Captures most surfaces with low error**
    - **Cheap to evaluate**
    - **Simple to fit to data**

Polynomial Texture Maps

- **Variables**
  - **2D projection of light direction into texture (tangent) space**

$$L(u,v;l_u,l_v) = a_0(u,v)l_u^2 + a_1(u,v)l_v^2 + a_2(u,v)l_u l_v + a_3(u,v)l_u + a_4(u,v)l_v + a_5(u,v)$$



GameDevelopers Conference

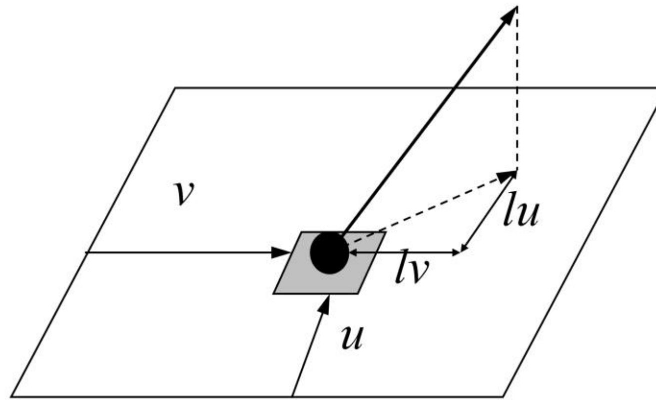
Make Better Games.

You can have 3 polynomials per pixel (one each for R, G and B), but it’s more efficient to separate RGB and just have a polynomial for luminance. Note that the polynomial is completely defined by the values of the six coefficients, which vary from texel to texel in the polynomial texture map.

Polynomial Texture Maps

- **Variables**
  - **2D projection of light direction into texture (tangent) space**

$$L(u,v;l_u,l_v) = a_0(u,v)l_u^2 + a_1(u,v)l_v^2 + a_2(u,v)l_u l_v + a_3(u,v)l_u + a_4(u,v)l_v + a_5(u,v)$$



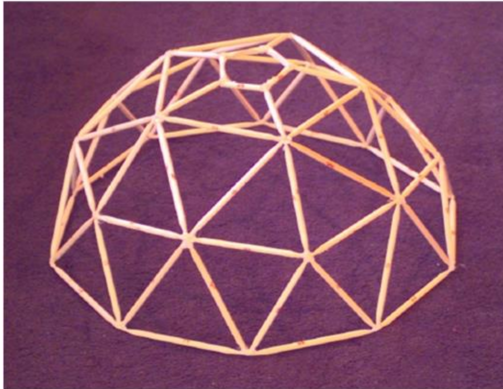
GameDevelopers Conference

Make Better Games.

Note that projecting the light direction to a tangent space at each vertex and interpolating the result to be used in the pixel shader is exactly the same as in bump mapping, and in fact the vertex shaders for both are the same.

Polynomial Texture Maps

- **Capture**
  - **With camera and light rig from real object**



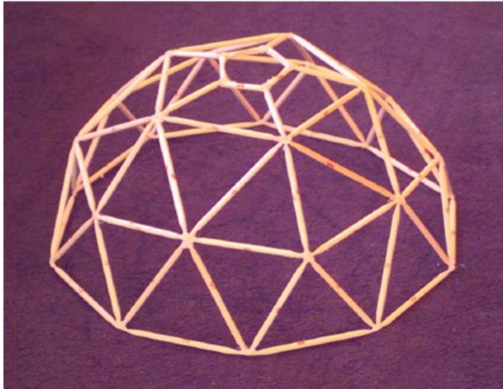
  - **Or with virtual rig from 3D model**
- **Fit polynomials to data using SVD**

GameDevelopers Conference
Make Better Games.

The paper used two capture rigs – a simple guide for placing camera and lights, and a more elaborate rig with automatically activated lights. In either case, the end result is a set of images (the paper used 40-50 images per set) of the same surface from multiple different (known) light directions.

Polynomial Texture Maps

- **Capture**
  - **With camera and light rig from real object**



  - **Or with virtual rig from 3D model**
- **Fit polynomials to data using SVD**

GameDevelopers Conference

Make Better Games.

Each pixel's luminance across images is treated as data points for a luminance function (of light direction), a polynomial fitted to those points—the paper used SVD (single value decomposition)—and its coefficients stored in textures. The tools are freely available, so there is no need to re-implement them.

Polynomial Texture Maps

- Storage
  - Each coefficient can be stored in 8 bits
    - Global scale and bias values are also stored
  - Can pack 6 coefficients into two textures
    - Plus a third texture for RGB
    - Alpha channels can be used for gloss, etc.
  - Or use different color space
    - Instead of LRGB, use YCbCr
    - Store Cb, Cr and 6 coefficients for Y polynomial
    - Can then pack whole thing into two textures

Storing separate luminance and RGB is actually redundant. There are color spaces like YCbCr which are easily transformed to/from RGB and which have one luminance and two chromaticity channels.

Polynomial Texture Maps

- L(lu,lv) is symmetrical about Z plane
  - Backfacing pixels are lit the same as front-facing
  - Solutions – when light behind surface:
    - Modulate with a darkening factor, or
    - Zero out some of the polynomial terms, or
    - Extend lu, lv beyond unit circle

GameDevelopers Conference
Make Better Games.

Since the polynomials are symmetrical about the Z=0 plane, backfacing lights need to be handled somehow. HP used a darkening factor; my implementation extends the 2D light vector beyond the unit circle, which gives a result consistent with lighting values in other directions.

Polynomial Texture Maps

- **Demo (HP Implementation)**

GameDevelopers Conference

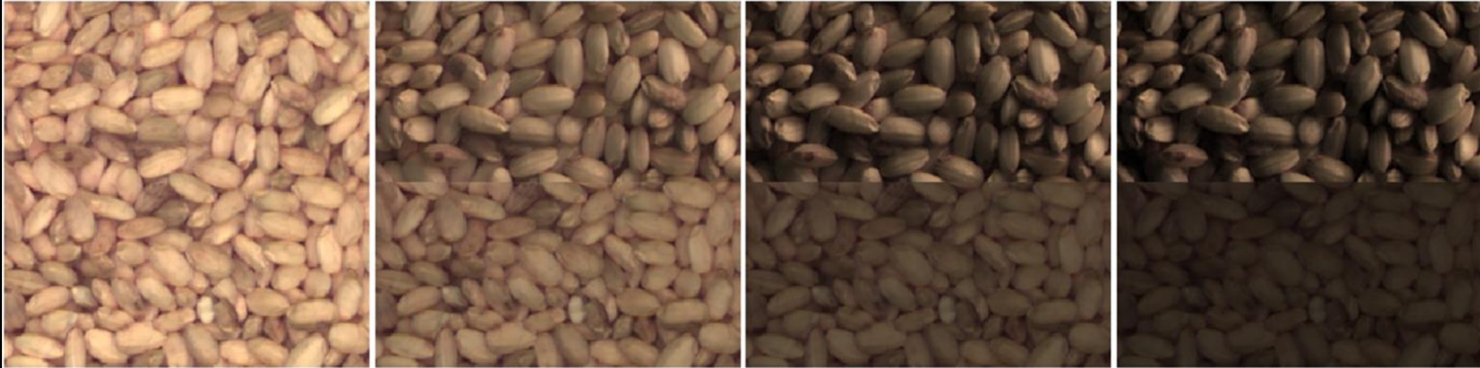
Make Better Games.

<Show HP viewer w. split quad between flat plane and seeds PTM>

Note that this can do anything which is not view-dependent – full global illumination with interreflections, subsurface scattering, etc. Note that half of the plane looks like seeds, while the other half looks like it a picture of seeds.

Polynomial Texture Maps

- **Demo (HP Implementation)**



GameDevelopers Conference

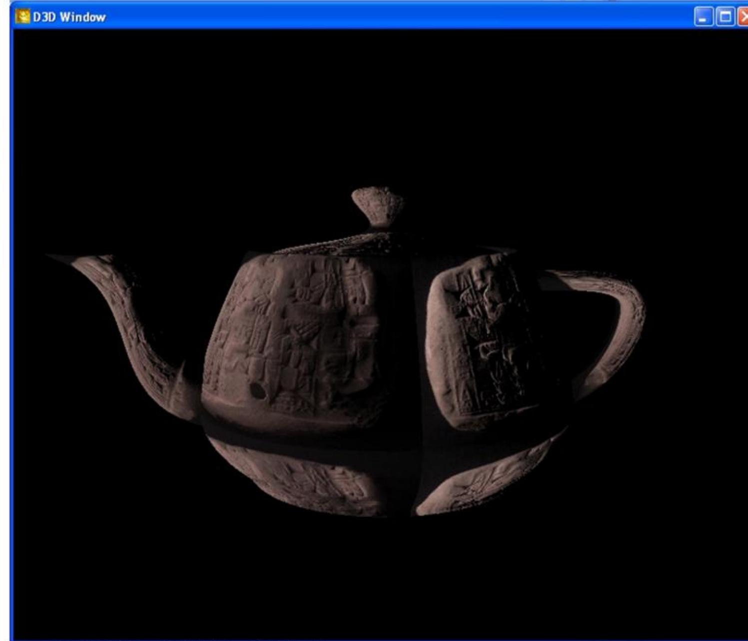
Make Better Games.

<Switch to pebbles PTM, turn split-quad off and switch to cylinder, then to teapot. Move light and object around a bit>

Note that the “terminator” line between light and shadow follows the contour of the surface detail.

Polynomial Texture Maps

- **Demo (HLSL Implementation)**



GameDevelopers Conference

Make Better Games.

This “RenderMonkey” demo (on Radeon 9700 Pro) uses HLSL adapted from assembly by Arcot Preetham (ATI). <Open RM PTM project in read only mode. Workspace and output off, fullscreen on. Start with cracked quad, then teapot. Move light around, then move ‘mode’ to turn off the backfacing shadows.>

Vertex Shader Source

```
struct VS_OUTPUT
{
   float4 Pos:    POSITION;
   float4 Tex:    TEXCOORD0;
   float3 Light:  TEXCOORD1;
};
VS_OUTPUT main(
   float4 vPosition: POSITION,
   float4 vNormal:   NORMAL,
   float4 vTex:      TEXCOORD,
   float4 vTangent:  TANGENT,
   float4 vBinormal: BINORMAL )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   Out.Pos = mul(view_proj_matrix, vPosition);
   float3 light_vector;

   Out.Tex = vTex;
   light_vector = normalize(mul(inv_view_matrix, light) - Out.Pos);

   Out.Light[0] = dot(light_vector, vTangent.xyz);
   Out.Light[1] = dot(light_vector, vBinormal.xyz);
   Out.Light[2] = dot(light_vector, vNormal.xyz);

   return Out;
}
```

The nifty syntax color coding is also from RenderMonkey.

We can see that the vertex shader code is identical to a standard bump-mapping implementation. Just transform the light vector into the local tangent space and pass it down as a texture coordinate.

```
float4 main(
   float4 Tex:     TEXCOORD0,
   float3 Light:    TEXCOORD1 ) : COLOR
{
   float3 lu2_lv2_lulv;
   float4 c;
   float3 a012;
   float3 a345;

   // Normalize light direction
   Light = texCUBE(normalizer, Light) * 2.0 - 1.0;

   // z-extrapolation
   if (mode > 0.0f && Light.z < 0.0) {
      Light.xy = normalize(Light.xy);
      Light.xy *= (1.0 - Light.z);
   }
   Light.z = 1.0;

   // Prepare higher-order terms
   lu2_lv2_lulv = Light.xyx * Light.xyy;

   // read higher-order coeffs from texture and unbias
   a012 = tex2D(a012_map, Tex) * 2.0 - 1.0;

   // read lower-order coeffs from texture and unbias
   // (a5 isn't biased, just halved)
   a345 = tex2D(a345_map, Tex) * 2.0 - 1.0;
   a345[2] += 1.0;

   // Evaluate polynomial
   c = dot(lu2_lv2_lulv, a012) + dot(Light, a345);

   // Multiply by rgb factor
   c = c * tex2D(rgb_map, Tex);

   return c;
}
```

Pixel Shader Source

Make Better Games.

The pixel shader is where the interesting stuff happens. Normalizing light direction helps with backfacing shadow code but isn't strictly needed. The 'z-extrapolation' code (toggled by "mode") handles backfacing lights. The core is just two vector multiplies, two texture reads, two dot products and a vector add.

PTMs in Games

- PTMs can be used in any type of game
  - Anywhere you would use bump mapping
- HP tools can also be used to capture bump map + color from photographs
  - For use in standard bump-mapping

GameDevelopers Conference

Make Better Games.

Note that HP also have a technique to capture an ordinary bump map from the PTM is needed (basically find direction of highest luminance), which might be useful if you have a real-world surface you want to use as a material and you don’t want to use PTMs for rendering.

Spherical Harmonic Lighting

- 2001: Ramamoorthi, Hanrahan (Stanford)
- Conceptual breakthrough
- Enables arbitrary light environments
- But purely local illumination
- Only diffuse surfaces

GameDevelopers Conference
Make Better Games.

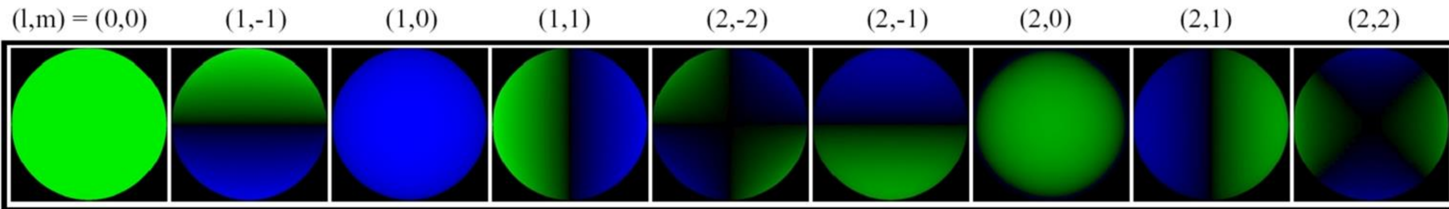
Ramamoorthi and Hanrahan introduced a new way of thinking about rendering as well as a new rendering technique. The underlying ideas are also used in the next technique we will discuss. This technique enables using any arbitrary light environment for local illumination of a diffuse surface.

Spherical Harmonic Lighting

- **Basic Concept**
  - Look at the rendering equation in frequency space
  - Convolution of light environment and BRDF turns into multiplication of coefficients in frequency space
  - Use Spherical Harmonic basis functions
    - Natural basis for functions on sphere

Spherical Harmonic Lighting

- **SH Basis Functions**



- **Diffuse BRDF is low-pass filter**
  - **Can ignore all coefficients beyond first 9**
  - **So any environment can be represented by 9 RGB values**
    - **At least for diffuse lighting purposes**

GameDevelopers Conference

Make Better Games.

Here are the first 9 (1 x 0th-order, 3 x 1st-order, and 5 x 2nd-order) spherical harmonic basis functions (green=positive, blue=negative). Simple polynomials in x, y and z. Key idea of the paper: Lambertian BRDF is low-pass filter without significant coefficients after 1st nine, so lighting only needs 9 coefficients.

Spherical Harmonic Lighting

- **Function for diffuse illumination by arbitrary light environment**
  - Takes normal in light space as input
  - Uses SH coefficients as constants

$$E(\mathbf{n}) = \mathbf{n}^t M \mathbf{n} \qquad M = \begin{pmatrix} c_1 L_{22} & c_1 L_{2-2} & c_1 L_{21} & c_2 L_{11} \\ c_1 L_{2-2} & -c_1 L_{22} & c_1 L_{2-1} & c_2 L_{1-1} \\ c_1 L_{21} & c_1 L_{2-1} & c_3 L_{20} & c_2 L_{10} \\ c_2 L_{11} & c_2 L_{1-1} & c_2 L_{10} & c_4 L_{00} - c_5 L_{20} \end{pmatrix}$$

$$c_1 = 0.429043 \quad c_2 = 0.511664$$

$$c_3 = 0.743125 \quad c_4 = 0.886227 \quad c_5 = 0.247708$$

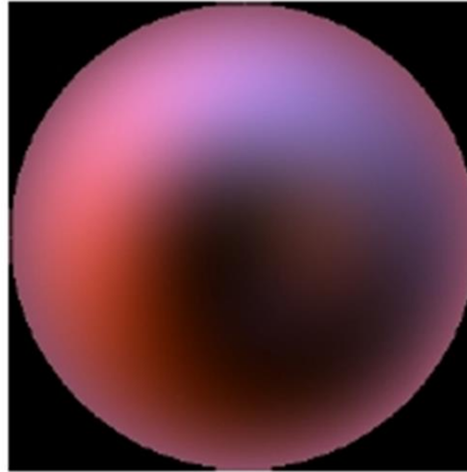
$$\begin{aligned} E(\mathbf{n}) &= c_1 L_{22} \left(x^2 - y^2\right) + c_3 L_{20} z^2 + c_4 L_{00} - c_5 L_{20} \\ &+ 2c_1 \left(L_{2-2} xy + L_{21} xz + L_{2-1} yz\right) \\ &+ 2c_2 \left(L_{11} x + L_{1-1} y + L_{10} z\right) \end{aligned}$$

Make Better Games.

Ramamoorthi & Hanrahan used SH analysis to find a closed-form equation which can compute diffuse lighting in any light environment as a function of the normal. This equation is shown here in two forms, a matrix form and a polynomial form. The matrix form is more appropriate for programmable HW.

Spherical Harmonic Lighting

- **Demo (HLSL)**

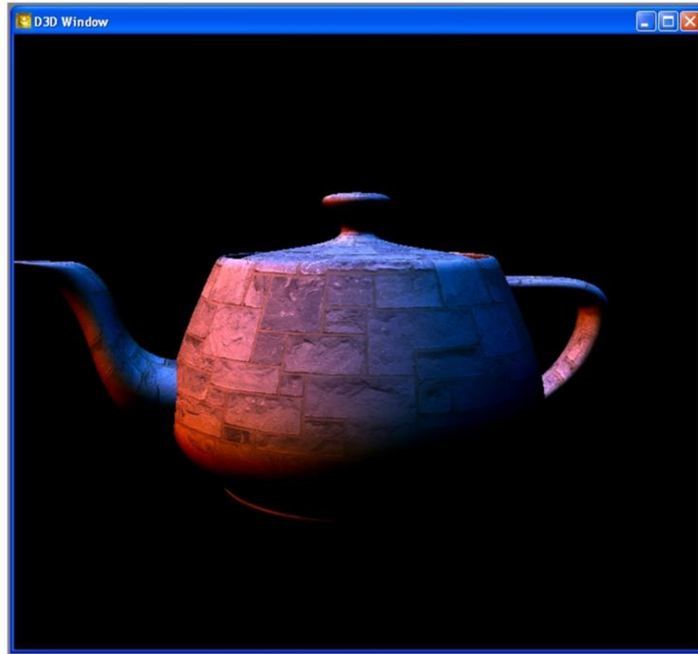


	Grace Cathedral		
L_{00}	.79	.44	.54
L_{1-1}	.39	.35	.60
L_{10}	-.34	-.18	-.27
L_{11}	-.29	-.06	.01
L_{2-2}	-.11	-.05	-.12
L_{2-1}	-.26	-.22	-.47
L_{20}	-.16	-.09	-.15
L_{21}	.56	.21	.14
L_{22}	.21	-.05	-.30

SH demos use Debevec's Grace Cathedral light probe. Here we see the light probe and its diffuse lighting result, which is represented by 18 coefficients (from Ramamoorthi and Hanrahan 2001). The authors analyzed the error compared to a full simulation and found it was quite small (about 1%).

Spherical Harmonic Lighting

- **Demo (HLSL – Per-Vertex)**



GameDevelopers Conference

Make Better Games.

RenderMonkey demo of per-vertex spherical harmonics lighting. The shaders are in HLSL using vertex shader model 1.1 and pixel shader model 1.1. (could used fixed-function for the pixels). The demo is running on a Radeon 9700 Pro. <Load per-vertex SH project. Rotate the teapot a bit – don't spend too long>

Vertex Shader Source

```
struct VS_OUTPUT
{
   float4 Pos:    POSITION;
   float4 Diff:   COLOR;
   float4 Tex:    TEXCOORD;
};
VS_OUTPUT main(
   float4 vPosition: POSITION,
   float4 vNormal:   NORMAL,
   float4 vTex:      TEXCOORD )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   Out.Pos = mul( view_proj_matrix, vPosition);

   Out.Tex = vTex;

   // Rotate normal from object/world space to light (view space)
   // (in RenderMonkey, lights are defined in view space).
   float4 normal4 = float4(vNormal.x, vNormal.y, vNormal.z, 0.0);
   normal4 = mul(view_matrix, normal4);
   normal4.w = 1.0;

   // Evaluate spherical harmonic
   Out.Diff.r = dot(mul(r_sh_matrix, normal4), normal4);
   Out.Diff.g = dot(mul(g_sh_matrix, normal4), normal4);
   Out.Diff.b = dot(mul(b_sh_matrix, normal4), normal4);
   Out.Diff.a = 1.0;

   return Out;
}
```

Here the computation occurs per-vertex. You can see that the normal is rotated to worldspace, and then we do multiplies with the SH matrix to find the lighting color.

Pixel Shader Source

```
float4 main(
    float4   Diff: COLOR,
    float4   Tex:  TEXCOORD ) : COLOR
{
     float4 c = Diff * tex2D(rgb_map, Tex) * 2.0;

     return c;
}
```

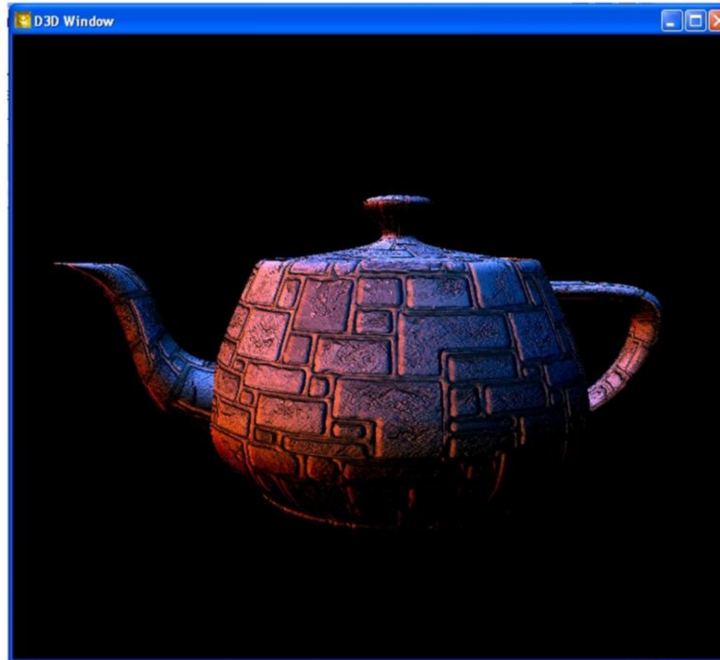
GameDevelopers Conference

Make Better Games.

The pixel shader is pretty bare-bones here.

Spherical Harmonic Lighting

- **Demo (HLSL – Per-Pixel)**



GameDevelopers Conference

Make Better Games.

This demo is running the lighting calculations per-pixel.

<Load per-pixel SH project, rotate around a bit, make sure to show the spout pointing to the left since that is the side where the bumps are correct (the teapot model has reversed tangent spaces on one side)>

Vertex Shader Source

```
struct VS_OUTPUT
{
   float4 Pos:       POSITION;
   float4 Tex:       TEXCOORD0;
   float3 Normal:    TEXCOORD1;
   float3 Tangent:   TEXCOORD2;
   float3 Binormal:  TEXCOORD3;
};
VS_OUTPUT main(
   float4 vPosition: POSITION,
   float4 vNormal:   NORMAL,
   float4 vTex:      TEXCOORD,
   float4 vTangent:  TANGENT,
   float4 vBinormal: BINORMAL )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   Out.Pos = mul(view_proj_matrix, vPosition);

   Out.Tex = vTex;

   // Rotate tangent basis from object/world space to view
   // space (in RenderMonkey lights are defined in view space)
   float4 nvec = float4(vNormal.x, vNormal.y, vNormal.z, 0.0);
   float4 tvec = float4(vTangent.x, vTangent.y, vTangent.z, 0.0);
   float4 bvec = float4(vBinormal.x, vBinormal.y, vBinormal.z, 0.0);
   Out.Normal = mul(view_matrix, nvec);
   Out.Tangent = mul(view_matrix, tvec);
   Out.Binormal = mul(view_matrix, bvec);

   return Out;
}
```

This is the kind of vertex shader you might see for correct environment-mapped bump-mapping or other cases where you need to do per-pixel transforms of vectors. We are transforming the tangent basis to the light space, and then sending the whole thing down in texture coordinates.

Pixel Shader Source

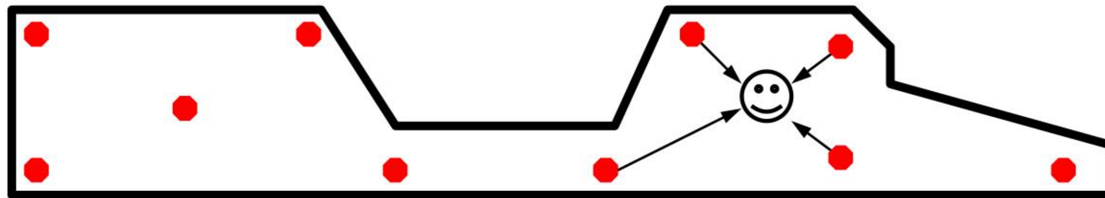
```
float4 main(
    float4 Tex:         TEXCOORD0,
    float3 Normal:      TEXCOORD1,
    float3 Tangent:     TEXCOORD2,
    float3 Binormal:    TEXCOORD3 ) : COLOR
{
    float4 c;
    float3x3 rotation;
    float3 normal3;
    float4 normal4;

    // Matrix to transform from tangent space into light space)
    rotation = float3x3(Tangent, Binormal, Normal);

    // Get normal
    normal3 = tex2D(bump_map, Tex) * 2.0 - 1.0;

    // Transform normal into light space
    normal3 = mul(normal3, rotation);

    normal4 = float4(normal3, 1.0);

    // Evaluate spherical harmonic
    c.r = dot(mul(r_sh_matrix, normal4), normal4);
    c.g = dot(mul(g_sh_matrix, normal4), normal4);
    c.b = dot(mul(b_sh_matrix, normal4), normal4);
    c.a = 1.0;

    // Multiply by rgb factor (and scale by two)
    c = c * tex2D(rgb_map, Tex) * 2.0;

    return c;
}
```

r Games.

Here the pixel shader rotates the bumped normal into light space, and then does the calculation with the SH matrices to find the light value. It is fairly similar to the vertex shader for the per-vertex SH lighting (unsurprisingly).

SH Lighting in Games

- Per-vertex or per-pixel normals
- Precalc light environments in level
  - Store at key points – interpolate between as characters move



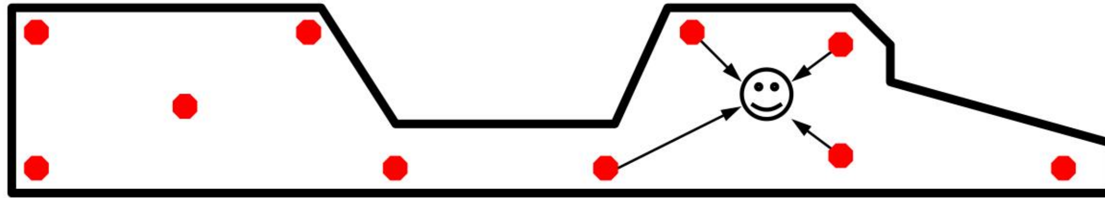
GameDevelopers Conference

Make Better Games.

It works best with baked lighting. This shows part of a game level. Lighting is sampled at key points (red dots) and stored (18 numbers: 9 coefficients x 3 color channels). As a character (happy face) moves about the level, SH coefficients interpolated from the nearest sample points are used to light him.

SH Lighting in Games

- Per-vertex or per-pixel normals
- Precalc light environments in level
  - Store at key points – interpolate between as characters move
  - Dynamic lights can be added in as needed



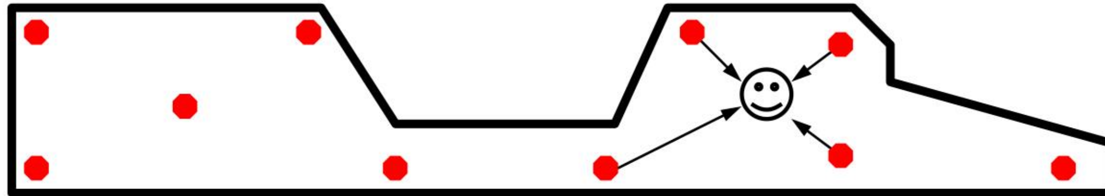
GameDevelopers Conference

Make Better Games.

SH coefficients for any dynamic lights can be computed on the fly and added in (adding or interpolate lighting by adding or interpolating SH coefficients is a key benefit of SH lighting).

SH Lighting in Games

- **Per-vertex or per-pixel normals**
- **Precalc light environments in level**
  - **Store at key points – interpolate between as characters move**
  - **Dynamic lights can be added in as needed**
  - **Outdoor scene can have single precalculated environment**



GameDevelopers Conference

Make Better Games.

Indoor games should show this technique to best effect, but outdoor games can benefit as well – imagine a sunset sky with a rich palette of colors, used to light the objects in the scene. This technique can also be combined with environment mapping for specular to great effect.

Precomputed Transfer Functions

- 2002: Sloan, Kautz, Snyder (Microsoft Research & Max-Planck-Institut)
- Based on same theoretical underpinnings as last technique (SH)
- Combines many strengths of last two
  - Full global illumination, scattering
  - Almost-arbitrary light environments
- Also limited to diffuse materials

GameDevelopers Conference
Make Better Games.

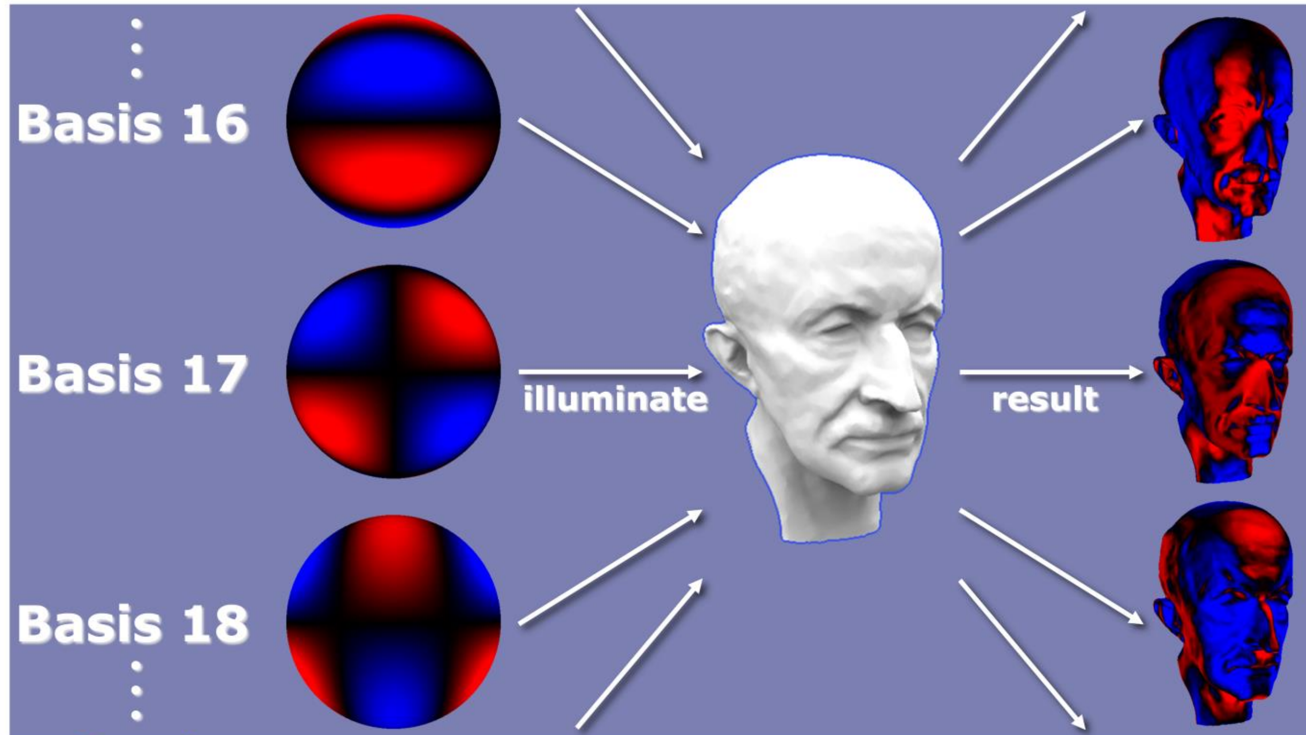
Precomputed transfer does not handle high-frequency lighting environments, it is best suited to relatively low-frequency lighting. If a high-frequency light environment is used, it will effectively be low-pass filtered by this technique, resulting in lighting which is less sharp than the original lighting.

Precomputed Transfer Functions

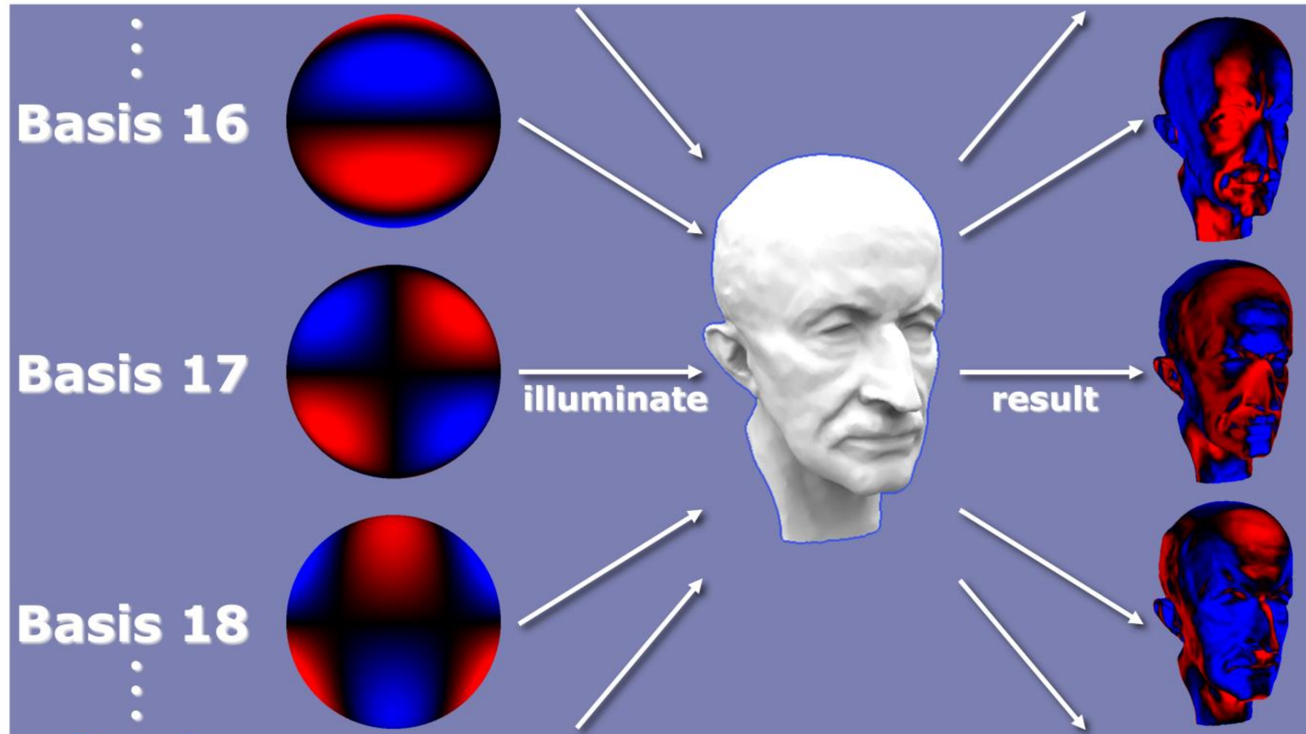
- Represent surface transfer function (response to light) as SH coefficients
  - Handles any non-view-dependent effects
    - interreflections, subsurface scattering, even caustics cast onto diffuse receivers
  - Store SH coefficients over surface of object (in a texture or at vertices)
  - Each coefficient can be thought of as the object illuminated with the basis function

GameDevelopers Conference
Make Better Games.

Can be seen as lighting an object with SH basis functions one by one, each time doing a GI simulation & storing the result (though SH basis functions aren't really lights; they have negative values). When rendering, these 'lighting solutions' are weighted by SH coefficients of incident lighting and combined.



As seen in this image (courtesy of Peter-Pike Sloan).



One way to do this is with a GI package which supports negative light. Otherwise, basis function can be split into negative and positive parts, each part used separately and the results combined. This could even enable capturing coefficients from real objects, given a rig which can handle the lighting.

Precomputed Transfer Functions

- **Rendering**
  - Just do a dot-product between the SH coefficients of the transfer functions and those of the incident light.
- **How many SH coefficients?**
  - Depends on how sharp the incident lighting needs to be
  - In most cases 16 or 25 coefficients
  - Usually TF coefficients are monochrome while lighting coefficients are RGB

GameDevelopers Conference
Make Better Games.

Rendering is simple once we have SH coefficients for transfer functions and incident lighting. 16 monochrome coefficients combined with 16 RGB coefficients means 12 vector dot-products and 9 adds. This is more than DX8 hardware can do in one pass, but DX9 HW should be able to handle it easily.

Precomputed Transfer Functions

- Technique can also do other things
  - Glossy transfer
  - Volume transfer
  - Neighborhood transfer
- But these are all currently too expensive for real-time
  - Requiring storing and multiplying matrices of coefficients instead of vectors

Precomputed Transfer Function

- **Demo – thanks to Peter-Pike Sloan, Microsoft**

GameDevelopers Conference

Make Better Games.

<Show Peter-Pike's PRT demo. Diffuse PRT with head and skull models. Compare just irradiance mapping with PRT, rotate the object and the light a bit, maybe switch light environments.> This technique has recently been extended to support glossy transfer and higher-order coefficients much more efficiently.

PTFs in Games

- Similar considerations to SH lighting
  - Precalc environments, add lights in
- Authoring similar to PTMs
  - Microsoft will publish tools
  - Or use global illumination solution of choice with basis functions as light source
  - Capturing from real objects would be more difficult

GameDevelopers Conference

Make Better Games.

The Global illumination solution needs to be able to handle negative lights, or need to separate positive and negative parts.

As discussed earlier, a light rig like Debevec's might be able to pull capture off if the positive and negative parts of the basis functions are separated.

```
; xform the point

m4x4 oPos, v0, c0

mul r0,v1,c4
mad r0,v2,c5,r0
mad r0,v3,c6,r0
mad r0,v4,c7,r0
mad r0,v5,c9,r0
mad r0,v6,c10,r0
mad r0.x,v7.x,c11.x,r0.x

dp4 oD0.x, r0, c8

mul r0,v1,c12
mad r0,v2,c13,r0
mad r0,v3,c14,r0
mad r0,v4,c15,r0
mad r0,v5,c16,r0
mad r0,v6,c17,r0
mad r0.x,v7.x,c11.y,r0.x

dp4 oD0.y, r0, c8

mul r0,v1,c18
mad r0,v2,c19,r0
mad r0,v3,c20,r0
mad r0,v4,c21,r0
mad r0,v5,c22,r0
mad r0,v6,c23,r0
mad r0.x,v7.x,c11.z,r0.x

dp4 oD0.z, r0, c8
```

Vertex Shader Source

Make Better Games.

If we do this per-vertex, the vertex shader will look like this and the pixel shader will be pretty trivial. If applying the technique per-pixel, then the pixel shader would look more or less like this code.

Conclusions

- **These techniques can be used today to enhance realism**
  - **The techniques can be combined**
    - **e.g. PTM + PTF for a light environment which is mostly low-frequency but has one sharp light**
  - **This is just the beginning**
    - **The research community is building on this work to make full use of modern programmable hardware**
    - **Expect performance and quality improvements**

GameDevelopers Conference

Make Better Games.

The SH techniques have already been extended to handle different BRDFs, and there is interesting work happening on reducing computational cost for the precomputed transfer functions technique.

Acknowledgements

- **Very special thanks to Peter-Pike Sloan and the Microsoft Windows Gaming and Graphics Technologies Group for PC, demo and other help**
- **Thanks to Arcot Preetham at ATI for initial PTM implementation and data**

References

- Kajiya, J. The Rendering Equation. SIGGRAPH 86
- Malzbender, T., Gelb, D., and Wolters, H. Polynomial Texture Maps. SIGGRAPH 2001 http://www.hpl.hp.com/ptm/
- Ramamoorthi, R. and Hanrahan, P. An Efficient Representation for Irradiance Environment Maps. SIGGRAPH 2001 http://graphics.stanford.edu/papers/envmap/
- Sloan, P-P., Kautz, J., and Snyder, J. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. SIGGRAPH 2002 http://research.microsoft.com/~ppsloan/

Questions?