Irradiance Volumes for Games

Chris Oat
ATI Research

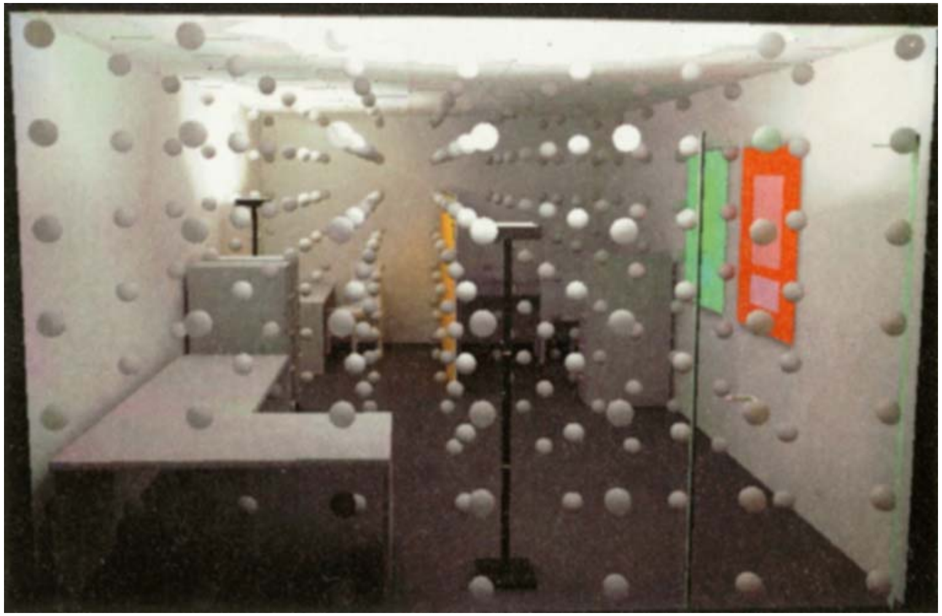
Overview

- Introduction and Motivation
- Review
  - Radiance, Irradiance, Transfer
- Spherical Harmonics
  - Projection, Gradients, Evaluation
- Irradiance Volume
  - Uniform Subdivision, Adaptive Subdivision, Interpolation
- Summary

Motivation

- One discontinuity between real time and non-real time rendering is the use of global illumination for physically based, realistic lighting
- Light mapping approximates global illumination on the surface of static scene geometry but light maps do not address dynamic objects that move through the scene
  - Result in beautifully rendered, globally illuminated scenes that contain unrealistic, locally lit dynamic objects
- Solution: Precomputed Irradiance Volumes for static scenes and Precomputed Radiance Transfer for objects within those scenes

The Irradiance Volume



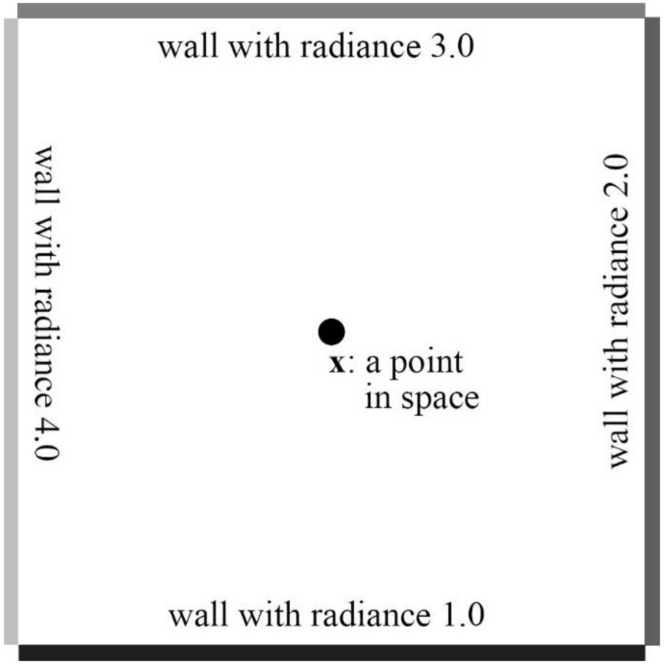
From [Greger]

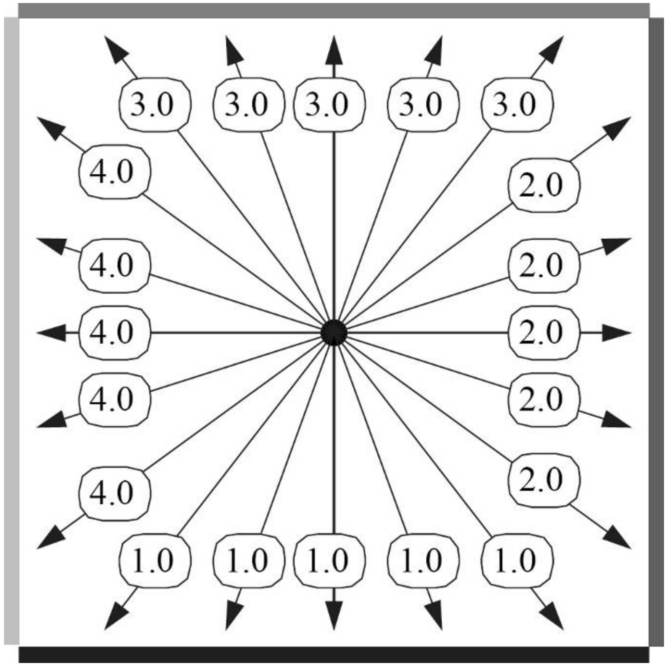
- This is what we're trying to achieve
- We aim to solve as much of the global illumination calculation during preprocess time
- A 3D light map: volume of diffuse lighting samples

Used in *Ruby: Dangerous Curves*

- These techniques were used as a drop in replacement for diffuse lighting in the *Ruby: Dangerous Curves* demo
- At the very least, these techniques could serve as an ambient lighting solution in your games
- Before diving into the details it is necessary to have a basic familiarity with: *radiance*, *irradiance*, and *transfer*

Radiance

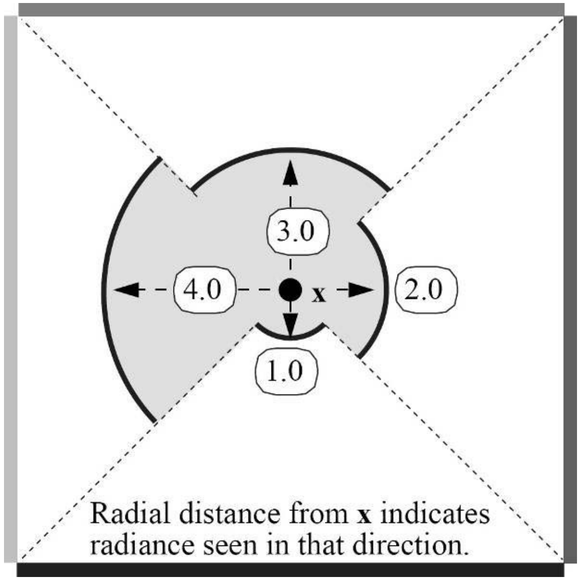




[Greger]

> Radiance is the emitted energy per unit time in a given direction from a unit area of an emitting surface

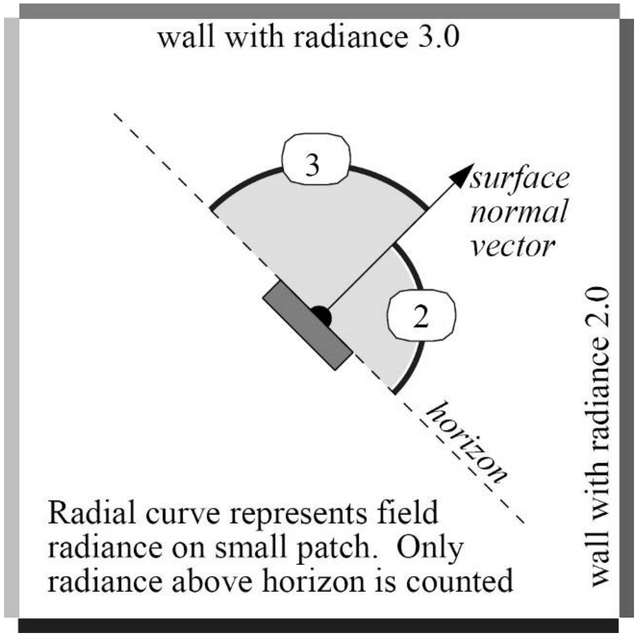
Radiance



From [Greger]

- We could capture radiance at a point for *all* directions by placing a camera at the point, rendering the surrounding scene into a cube map and scaling each texel by its projected solid angle
- This cube map would represent the radiance for all directions at the point where it was captured, this is known as the ***radiance distribution function***
- The *radiance distribution function* is not necessarily continuous, even in very simple environments
- There is a *radiance distribution function* at every point in space: ***radiance*** is a 5D function (3 spatial dimensions and 2 directional dimensions)

Radiance



From [Greger]

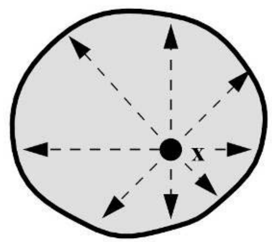
- The radiance of a surface is a function of its BRDF and incident radiance
- The incident radiance defined on the hemisphere of incoming directions is called the *field-radiance function*

Irradiance

- The radiance of a purely diffuse surface is defined in terms of the surface's *irradiance*
- Irradiance is an integral of the field-radiance function multiplied by the Lambertian cosine term over a hemisphere

$$I(p, N_p) = \int_{\Omega} L(p, \vec{\omega}_i)(N_p \bullet \vec{\omega}_i) d\omega_i$$

Irradiance



Radial distance from **x** is the irradiance for hypothetical differential surface with surface normal aligned to that direction.

From [Greger]

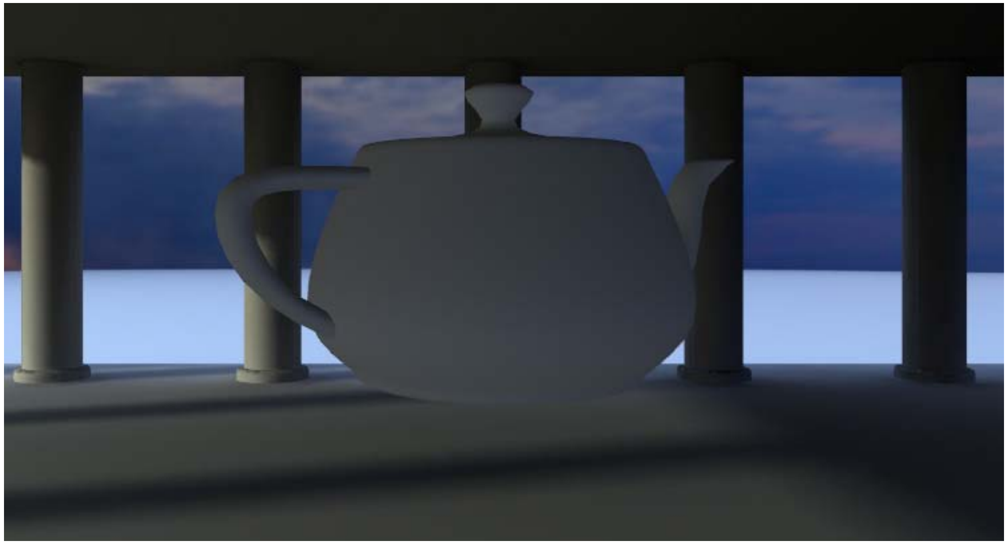
- We could compute irradiance at a point for *all* possible orientations of a small patch:
  - For each orientation, compute a convolution of the field radiance with a cosine kernel
- The result of this convolution for all orientations would be an irradiance distribution function
- The irradiance distribution function looks like a radiance distribution function except much blurrier because of the averaging process (convolution with cosine kernel)
- The irradiance distribution function is continuous over directions

Irradiance

- The irradiance distribution function can be computed for every point in space: irradiance is a 5D function (3 spatial dimensions and 2 directional dimensions)
- Evaluating the irradiance distribution function in the direction of a surface normal gives us irradiance at that surface location
- Computing irradiance distribution functions on demand is possible but can be costly. An obvious optimization is to precompute irradiance distribution functions for a scene at preprocess time and then use this precomputed data at runtime

Rendering with Irradiance

- The Irradiance Distribution Function at a point can be stored using a "Diffuse Cube Map"
- The cube map is indexed with an object's surface normal



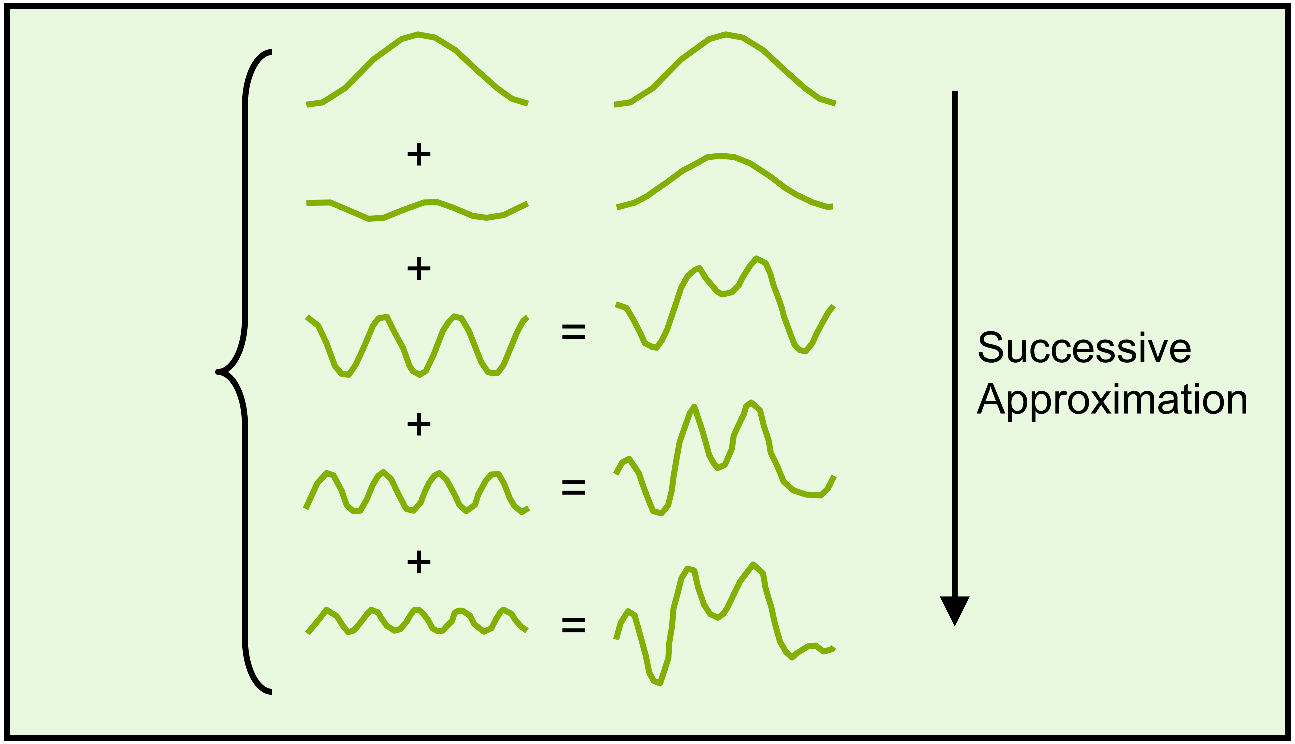
Efficient Storage of Irradiance

> If our objects move through the scene, an irradiance distribution function will be needed for many points
> As a preprocess, capture the lighting environment at many points in the scene. We now have a volume of irradiance distribution functions
> We're still left with the cost of storing cubemaps for many different points in our scene as well as the bandwidth overhead of indexing these maps at render time
> Instead, compress irradiance maps by representing each as a vector of spherical harmonic coefficients. This reduces both the storage and bandwidth costs

Spherical Harmonics

- Infinite series of spherical functions that may be used as basis functions to store a frequency space approximation of an environment map
- Microsoft's DirectX SDK includes functions for projecting a cubemap into a representative set of spherical harmonic coefficients (as well as functions for scaling and rotating spherical harmonics)
- For code snippets that will help your write your own spherical harmonic helper functions, see Robin Green's *Spherical Harmonic Lighting: The Gritty Details*

Fourier Theory

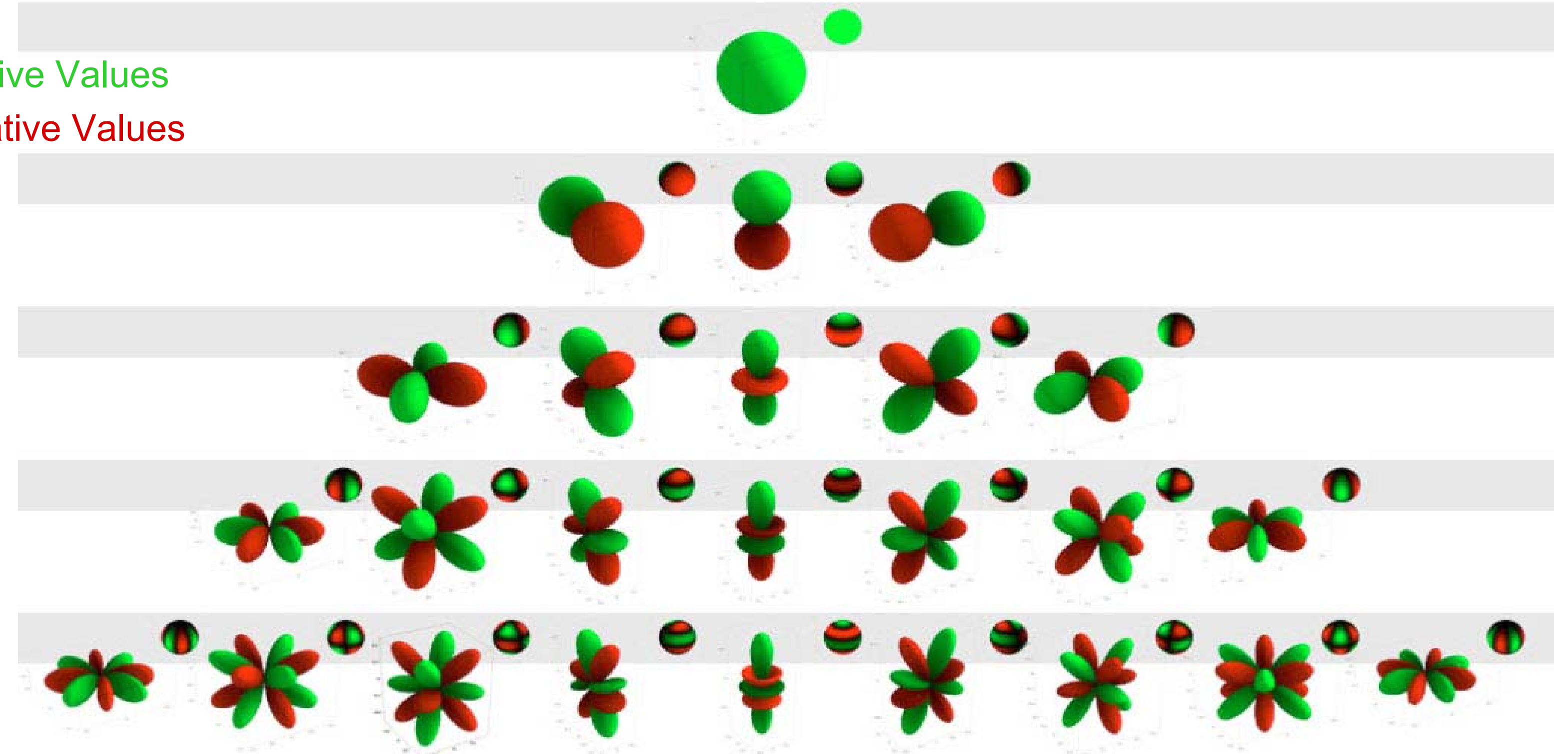


- Recall that it is possible to represent any 1D signal as a sum of appropriately scaled and shifted sine waves
- Spherical harmonics are the same idea on a sphere!

Spherical Harmonic Basis

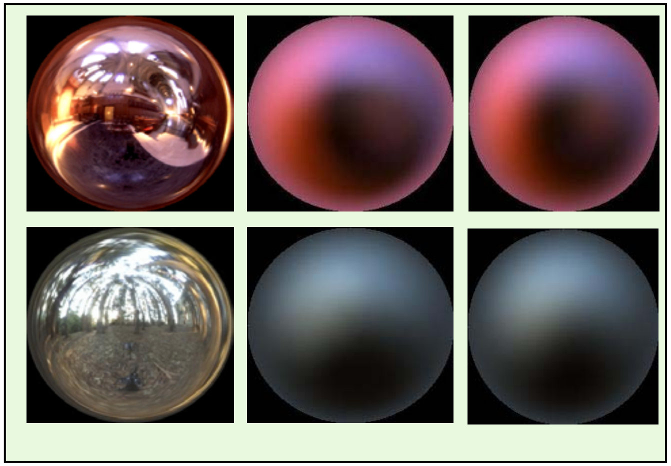
Positive Values

Negative Values



From [Green]

SH Projection: Storage and Computation WIN



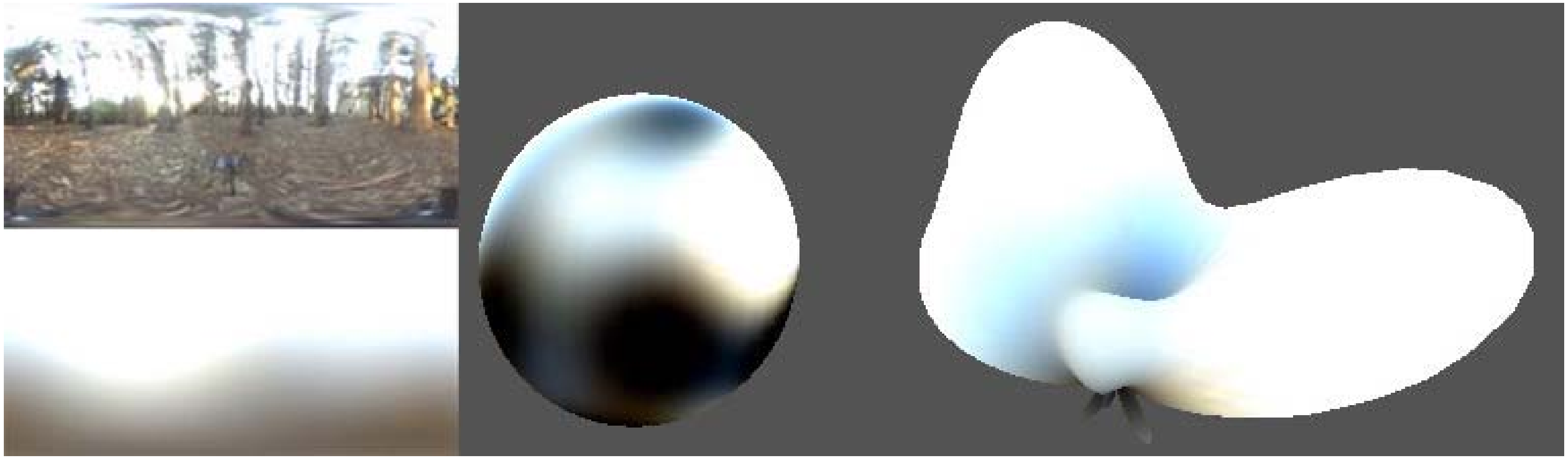
[Ramamoorthi]

Original Environment Map | **Filtered Environment Map** | **SH Representation**

- Projecting an environment map into 3rd order spherical harmonics effectively gives you the irradiance distribution function [Ramamoorthi]
- Projection into 3rd order SH is not only a storage win but a preprocessing win too since SH projection is much faster than convolving an environment map with a cosine kernel for all possible normal orientations

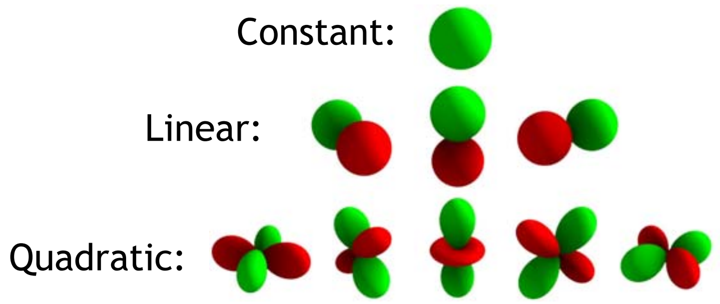
Spherical Harmonics

- Once an environment map has been projected into spherical harmonics, the coefficients can be used to evaluate the original map in a given direction
- Storing these coefficients VS constants allows us to compute irradiance per-vertex rather than having to sample a cubemap per-pixel



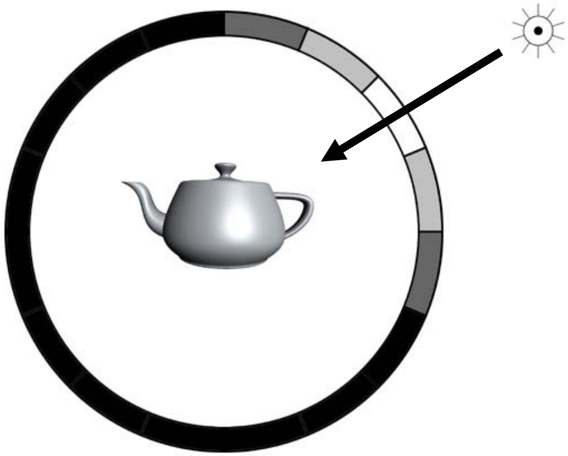
SH Evaluation With Normal

```
float4 cAr; // first 4 red irradiance coefficients
float4 cAg; // first 4 green irradiance coefficients
float4 cAb; // first 4 blue irradiance coefficients
float4 cBr; // second 4 red irradiance coefficients
float4 cBg; // second 4 green irradiance coefficients
float4 cBb; // second 4 blue irradiance coefficients
float4 cC;  // last 1 irradiance coefficient for red, blue and green

float3 x1, x2, x3;

// Linear + constant polynomial terms
x1.r = dot(cAr, vNormal);
x1.g = dot(cAg, vNormal);
x1.b = dot(cAb, vNormal);

// 4 of the quadratic polynomials
float4 vB = vNormal.xyzz * vNormal.yzzx;
x2.r = dot(cBr, vB);
x2.g = dot(cBg, vB);
x2.b = dot(cBb, vB);

// Final quadratic polynomial
float vC = vNormal.x*vNormal.x - vNormal.y*vNormal.y;
x3 = cC.rgb * vC;

Output.Diffuse.rgb = x1 + x2 + x3;
```



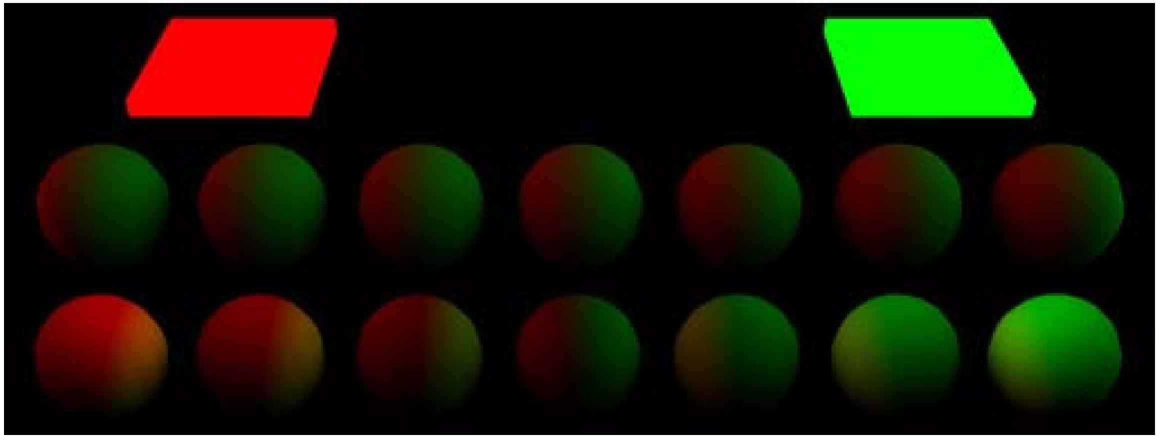
[Shader Code From DirectX SDK]

One Irradiance Sample – One Point in Space



- Irradiance samples only store irradiance for a single point in space
- This really only works well if the lighting environment is infinitely distant (just like a cubic environment map)
- This error can be very noticeable when the lighting environment isn't truly distant

Spherical Harmonic Irradiance Gradients



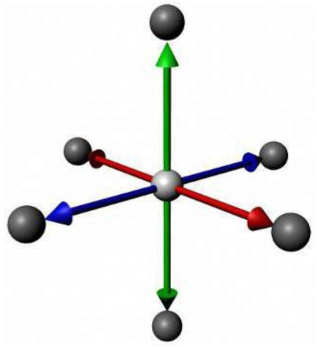
- If an irradiance sample is used to shade the surface of an object, the potential error increases the further we move away from the point at which the irradiance sample was generated
- Irradiance gradients allow us to store the rate at which irradiance changes with respect to translations about the sample

Spherical Harmonic Irradiance Gradients

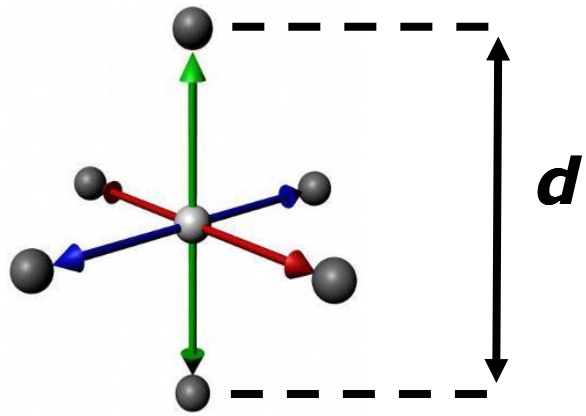
> Translational gradients for spherical harmonic irradiance samples may be computed in a number of ways [Annen]…

> One simple way to find the gradients is to use central differencing to estimate the partial derivatives of the spherical harmonic irradiance coefficients

> Project 6 additional irradiance functions into spherical harmonics and perform central differencing on each of the coefficients



Central Differencing



$$\nabla_y = \frac{y_{+1} - y_{-1}}{d}$$

- Subtract the coefficients for samples taken at a small offset in the +Y and –Y directions
- Divide by distance between the samples
- This gives you an estimate of the partial derivative with respect to *y* for each coefficient
- Do this for the other two axes as well…
- You now have a 3D gradient vector for each spherical harmonic coefficient

First-Order Taylor Expansion

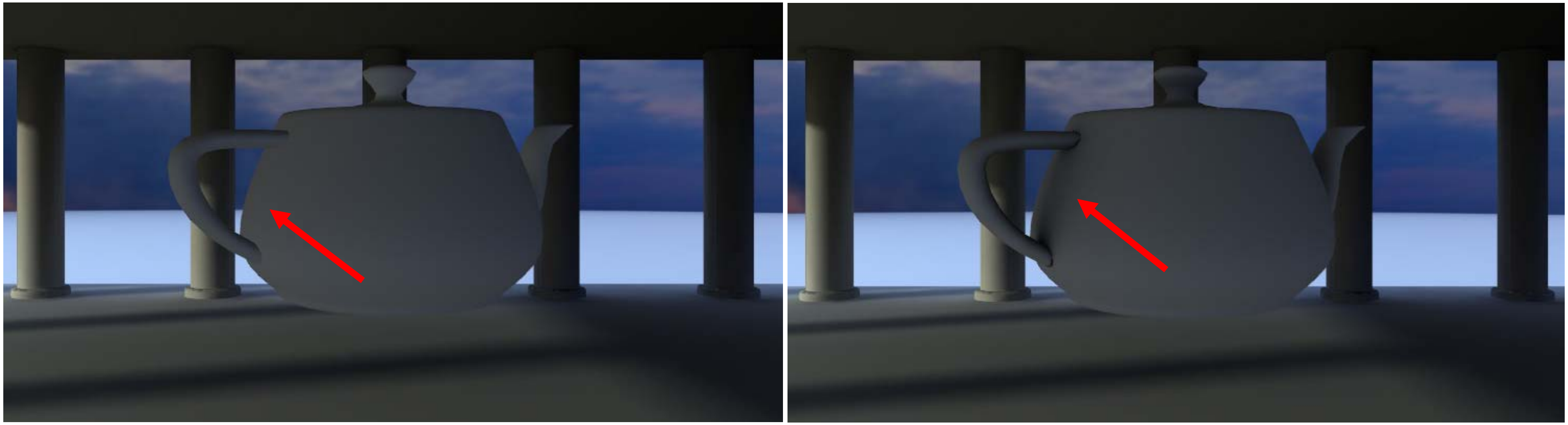
- At render time, the gradient may be used to extrapolate a new irradiance function
- Compute world space vector from the location at which the sample was generated to the point being rendered
- This vector is then dotted with the gradient vector and added to the original sample to extrapolate a new irradiance function

$$I_i' = I_i + (\nabla I_i \cdot d)$$

- *Ii′* is the *i*th spherical harmonic coefficient of the extrapolated irradiance function, *Ii* is the *i*th spherical harmonic coefficient of the stored irradiance sample, ∇I_i is the irradiance gradient for the *i*th irradiance coefficient and ***d*** is a non-unit vector from the original sample location to the point being rendered

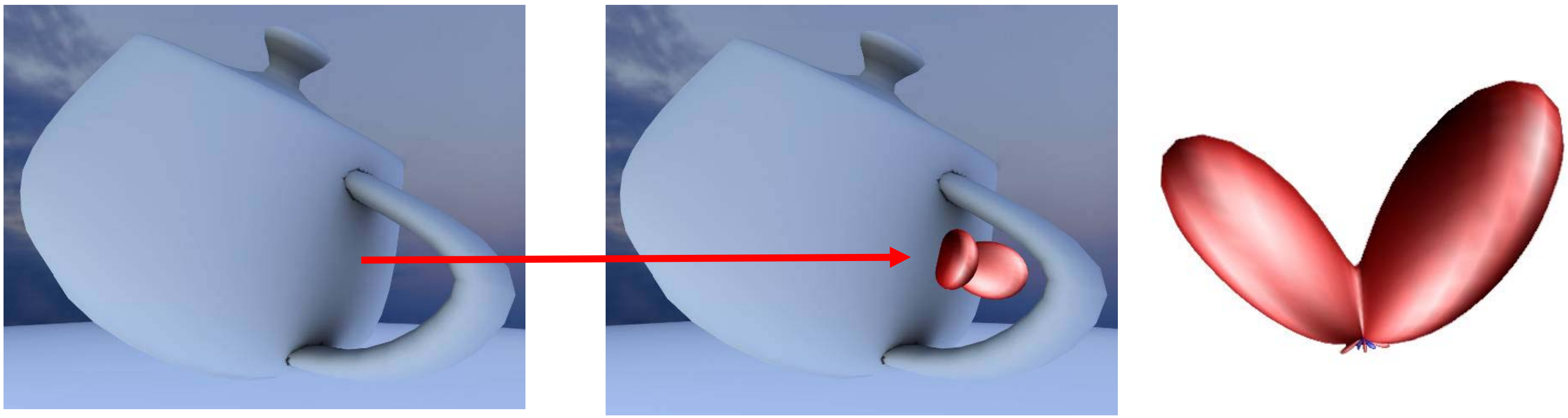
```
// Compute vector from original irradiance sample position to the position that is being shaded
float3 vSampleOffset = (vPos - vIrradianceSamplePosWS);

// Arrays for the extrapolated 4th order (16 coefficients per color channel) spherical harmonic irradiance
float4 vIrradNewRed[4]; float4 vIrradNewGreen[4]; float4 vIrradNewBlue[4];

// Extrapolate new irradiance for 4th order spherical harmonic irradiance sample
for ( int index = 0; index < 4; index++ )
{
    vIrradNewRed[index] = float4( dot(vSampleOffset, vIrradianceGradientRedOS[index*4 + 0]),
                                  dot(vSampleOffset, vIrradianceGradientRedOS[index*4 + 1]),
                                  dot(vSampleOffset, vIrradianceGradientRedOS[index*4 + 2]),
                                  dot(vSampleOffset, vIrradianceGradientRedOS[index*4 + 3]) );

    vIrradNewGreen[index] = float4( dot(vSampleOffset, vIrradianceGradientGreenOS[index*4 + 0]),
                                    dot(vSampleOffset, vIrradianceGradientGreenOS[index*4 + 1]),
                                    dot(vSampleOffset, vIrradianceGradientGreenOS[index*4 + 2]),
                                    dot(vSampleOffset, vIrradianceGradientGreenOS[index*4 + 3]) );

    vIrradNewBlue[index] = float4( dot(vSampleOffset, vIrradianceGradientBlueOS[index*4 + 0]),
                                   dot(vSampleOffset, vIrradianceGradientBlueOS[index*4 + 1]),
                                   dot(vSampleOffset, vIrradianceGradientBlueOS[index*4 + 2]),
                                   dot(vSampleOffset, vIrradianceGradientBlueOS[index*4 + 3]) );

    vIrradNewRed[index] = vIrradNewRed[index] + vIrradianceSampleRed[index];
    vIrradNewGreen[index] = vIrradNewGreen[index] + vIrradianceSampleGreen[index];
    vIrradNewBlue[index] = vIrradNewBlue[index] + vIrradianceSampleBlue[index];
}
```

What about Self Occlusion, Bounced Lighting?



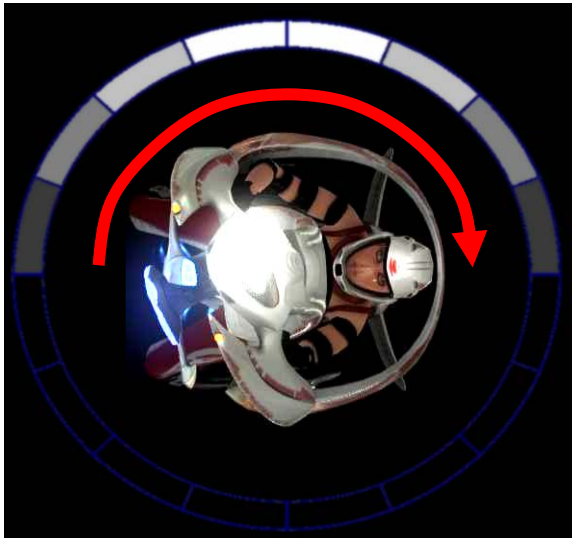
- Gradients improve the usefulness of each sample but we still haven't solved all our problems…
- One limitation of irradiance mapping is that it doesn't account for an object's self occlusion or for bounced lighting from the object itself
- This additional light transport complexity can be accounted for by generating pre-computed radiance transfer (PRT) functions for points on the object's surface

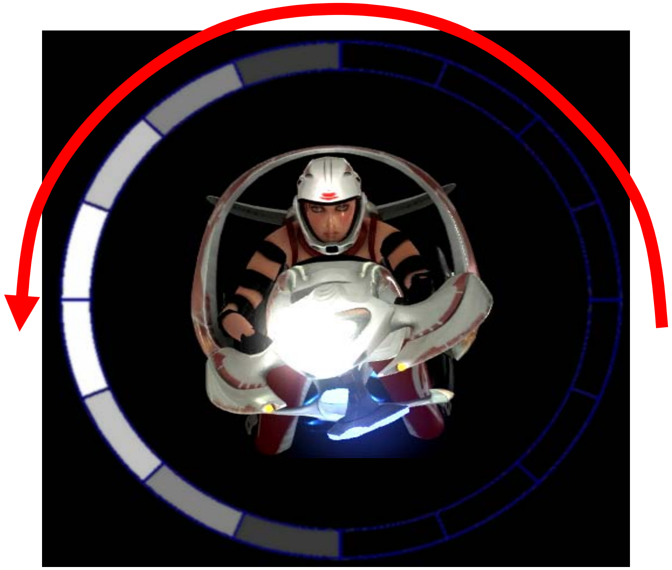
Precomputed Radiance Transfer



- Radiance Transfer maps incident radiance to reflected radiance
- PRT require incident *radiance*, we're dealing will *irradiance?!*
  - If you project an environment map into 3rd order SH and evaluate with a surface normal then the SH data represents irradiance
  - If you project an environment map into SH and integrate the product of the environment and transfer functions then the SH data represents low-frequency incident radiance (where "low-frequency" is relative to the order of the SH projection)
  - As long as we're assuming low-frequency, the data is the same… the difference is semantic
- If stored as SH, the integral of (Incident Radiance * Transfer) reduces to a dot product of two vectors (the vectors contain SH coefficients for incident radiance and transfer)

Handling Rotation



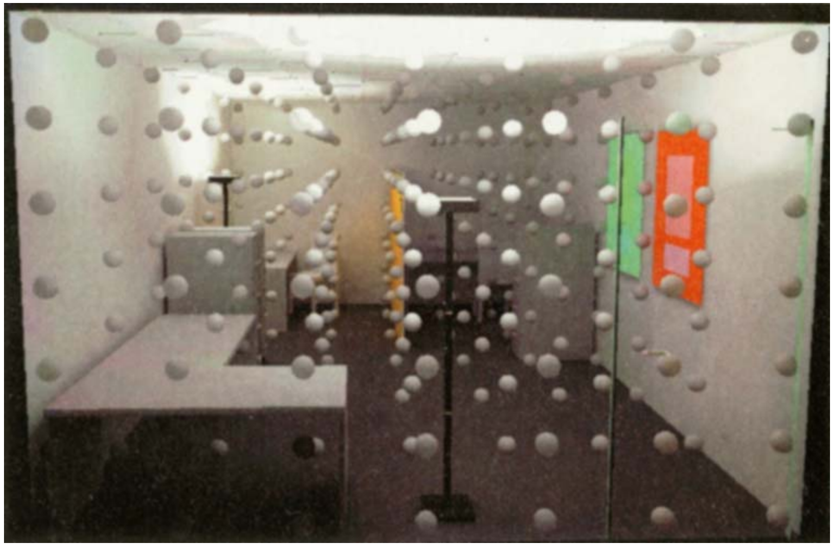


- If using samples for irradiance distribution, the surface normal used for finding irradiance should be transformed into world space (skinned) before evaluating the SH function
- If using the samples for PRT, the transfer function can not be easily rotated on the GPU so instead rotate the lighting environment by the inverse model transform on the CPU

Irradiance Volume: Background

- Irradiance volumes have been used by the film industry as an acceleration technique for high quality, photorealistic offline rendering
- The volumes store irradiance distribution functions for points in space by utilizing a spatial partitioning structure that serves as a cache
- Sampling the volume allows the for the global illumination of a point in space to be quickly calculated
- Spherical harmonics allow irradiance volumes to be efficiently stored and evaluated
- These volumes are compatible with precomputed radiance transfer and allow for fast, efficient and realistic rendering in real time applications such as games

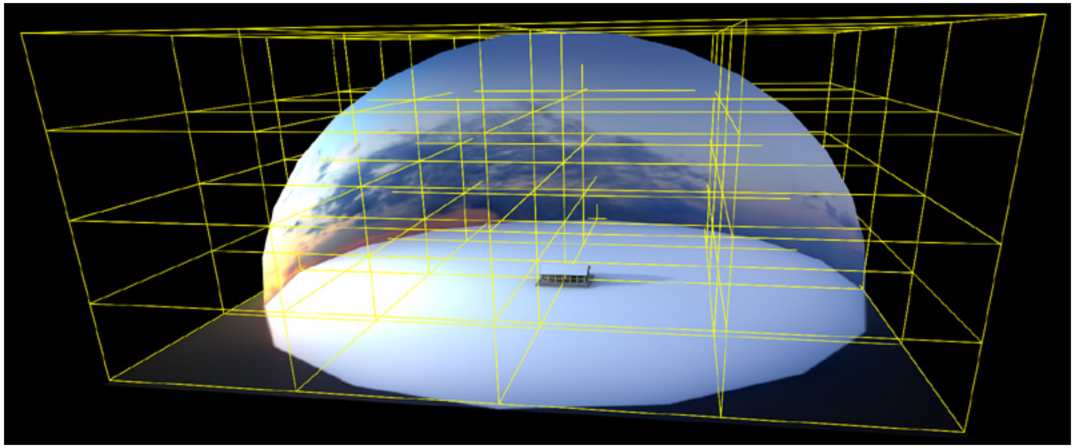
The Irradiance Volume



From [Greger]

- A grid of irradiance samples is taken throughout the scene
- At render time, the volume is queried and near-by irradiance samples are interpolated to estimate the global illumination at a point in the scene

Uniform Volume Subdivision



- Subdividing a scene into evenly spaced voxels is one way to generate and store irradiance samples
- Irradiance samples should be computed for each of the eight corners of all the voxels
- A uniform grid is easy to implement but quickly becomes unwieldy for large, complex scenes that require many levels of subdivision

Adaptive Volume Subdivision

- Choosing an adaptive subdivision scheme such as an octree will allow you to only subdivide the volume where subdivision is beneficial
  - For a given scene, some areas will have slowly changing irradiance and can be subdivided coarsely
  - Areas with quickly changing irradiance will need to be subdivided more finely

Adaptive Octree Subdivision

- Knowing which areas of your scene need further subdivision is a challenging problem
- For example, a character standing just inside a house will appear shadowed on a sunny day but if the character moves over the threshold of the door and into the sunlight they should appear much brighter; irradiance can change very quickly
- We need a way to find areas of rapidly changing irradiance so that these areas can be more finely subdivided

Adaptive Subdivision

- Since irradiance sampling is done as a preprocess, one option is to use a brute force method that starts by super-sampling irradiance using a highly subdivided uniform grid
- After this super-sampled volume is found, redundant voxels may be discarded by comparing irradiance samples at child nodes using some error tolerance to determine if a voxel was unnecessarily subdivided
- This brute force method isn’t perfect though because it assumes you know the maximum level of subdivision or super-sampling that is needed for a given scene
- Instead, certain heuristics may be used to detect voxels that might benefit from further subdivision

Subdivision Heuristics

- Measuring irradiance gradients and flagging voxels where the irradiance is known to change quickly with respect to translation (large gradient) is one way to test if further subdivision is necessary
- Testing gradients isn't perfect though, because this will only subdivide areas where you know that irradiance changes rapidly. There may still be areas that have small gradients but contain sub-regions with quickly changing irradiance
- Subdivide any voxels that **contain scene geometry** [Greger]
- Find the **harmonic mean** of scene depth at a sample point to determine when subdivision is needed [Pharr]
- The idea is that areas that contain a lot of geometry will have more rapidly changing irradiance
  - Not a bad assumption, the more geometry surrounding a sample point the more opportunities for shadows, bounced lighting, etc…
  - In the center of a room, lighting doesn't change much. As one approaches the walls things get interesting.

Harmonic Mean of Scene Depth

- Shoot a bunch of rays out from the irradiance sample's position
- Compute the harmonic mean of distance traveled by all rays before intersection

$$HM = \frac{N}{\sum_{i}^{N} \frac{1}{d_i}}$$

- *N* is the total number of rays fired, and d*i* is the distance that the *i*th ray traveled before intersecting scene geometry
- The harmonic mean is then used as an upper-bound for the sample's usefulness. If the neighboring irradiance samples are further away than this upper-bound, then their associated voxels should be subdivided
- The harmonic mean is chosen over the arithmetic mean (or linear average) because large depth values—due to infinite depth if no geometry exists in a given direction—would quickly bias the arithmetic mean to a large value

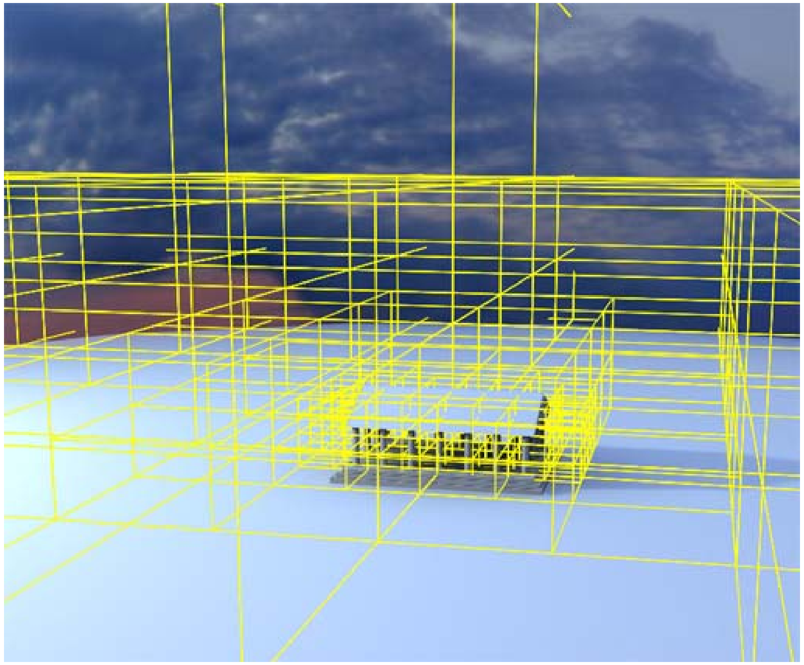
Using the GPU: Harmonic Mean of Scene Depth

- For each sample location, render the scene into each face of a floating point cubemap
- The scene should be drawn with a shader that outputs: 1/depth
- Read the cubemap back into system memory and find the harmonic mean

Using the GPU: Voxel Contains Scene Geometry

- If you're already reading back scene depth for the harmonic mean test, you can also use this data to determine if any scene geometry exists inside the voxel
  - Scene depth is sampled at the voxel corners, so only some of the cubemap texels should be used to test for scene intersection
- Alternatively, you could use occlusion queries:
  - Place the camera at the center of a voxel
  - Render into each face of a cubemap
    - First draw quads for each face of the voxel
    - Second draw the scene
  - If any of the scene's draw calls pass the occlusion query, a part of the scene is inside the voxel

Adaptive Subdivision

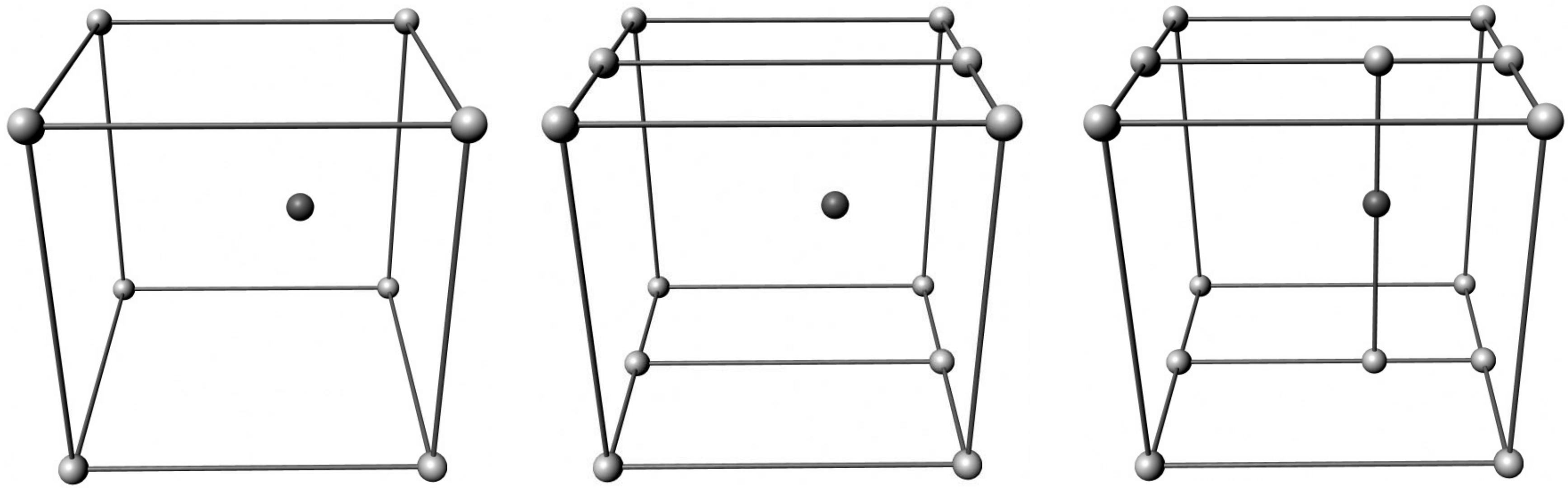


- Specify a **`Min`** and **`Max`** level of subdivision
- Allow thresholds to be specified for each subdivision heuristic
- After you've fully sampled the volume, go back and reject any redundant samples: if a voxel has been subdivided and it's children don't differ enough from the parent, these samples may be culled

Sampling the Volume

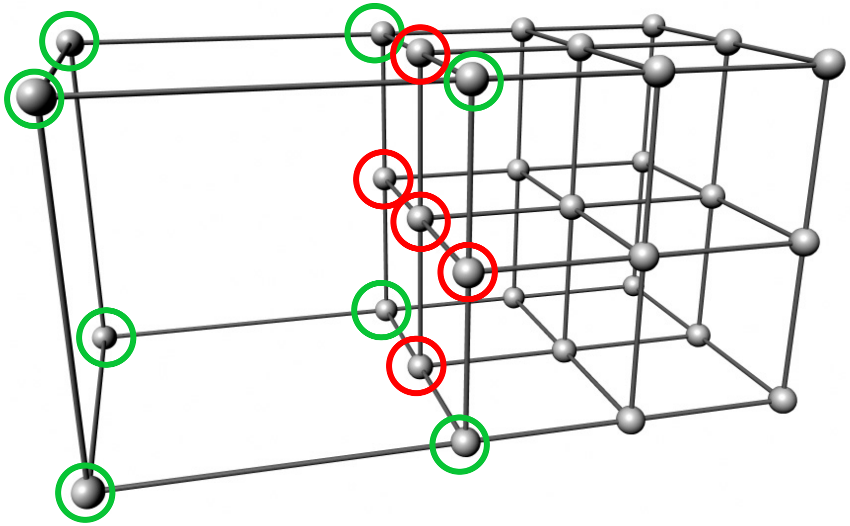
- If you’re using an octree, search the tree for the voxel that contains the object’s centroid
- Use the surrounding samples to determine irradiance
  - Interpolate surrounding samples (trilinear)
  - Find a weighted sum of surrounding samples (weighted by 1/distance)

Trilinear Interpolation



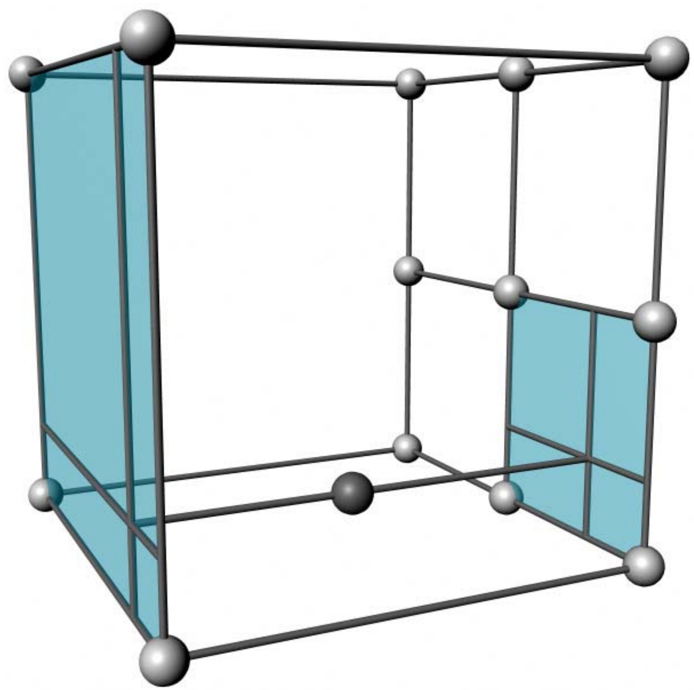
- Seven LERPs of the spherical harmonic coefficients
- Works well for uniformly subdivided volumes
- Adaptively subdivided volumes require slightly more care

Trilinear Interpolation



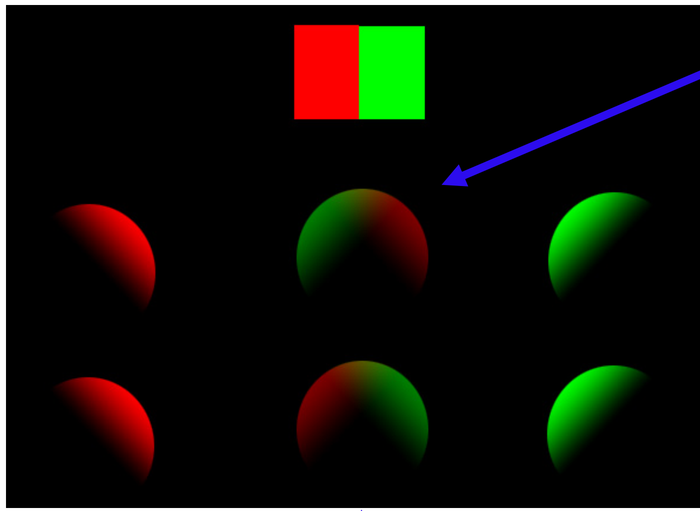
- When transitioning between voxels that have been adaptively subdivided, naïve trilinear interpolation can produce popping artifacts
- As an object moves from finely subdivided voxels to coarsely subdivided voxels, some of the sample data will suddenly be ignored

Trilinear Interpolation



- To prevent popping, continue using samples from subdivided neighbors for interpolation
- Each octree node should store pointers to samples that lie on each face

Using Gradients for Interpolation

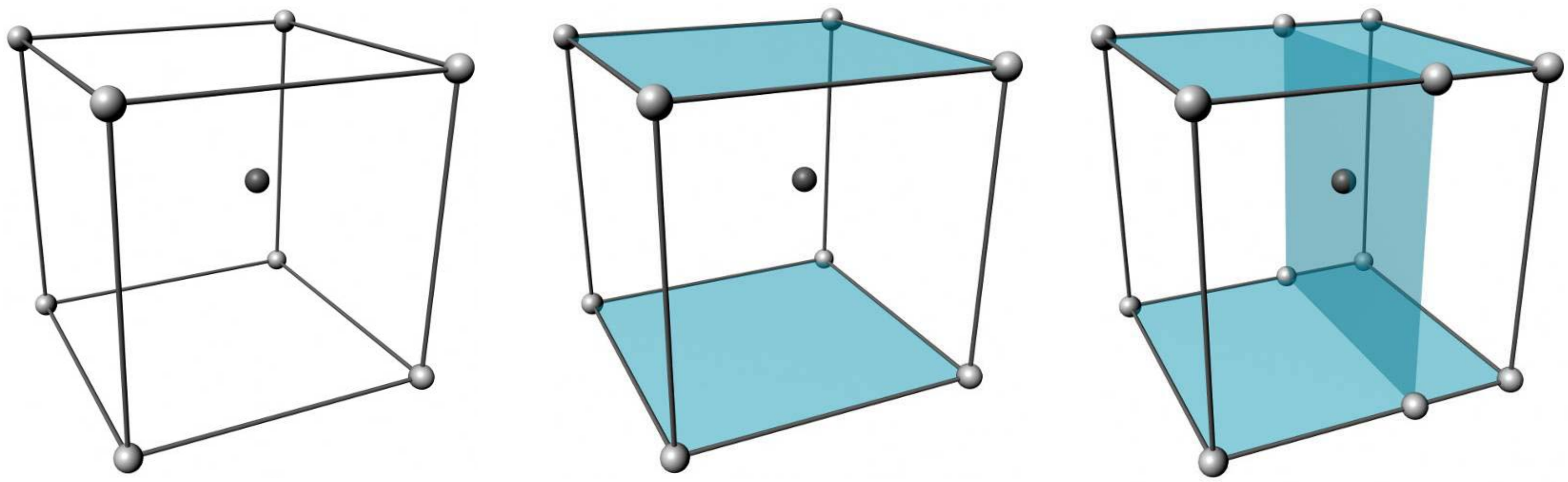


Linear interpolation between left and right samples

Gradients used for first-order Taylor expansion before interpolation

- Before using a sample for interpolation, evaluate the first-order Taylor expansion, then interpolate as usual.

Tricubic Interpolation



Use samples and gradients to construct cubic patches for interpolation. Hermite patches are well suited for this since they only require four control points and four tangents (gradients).

GPU Memory Requirements (Constant Store)

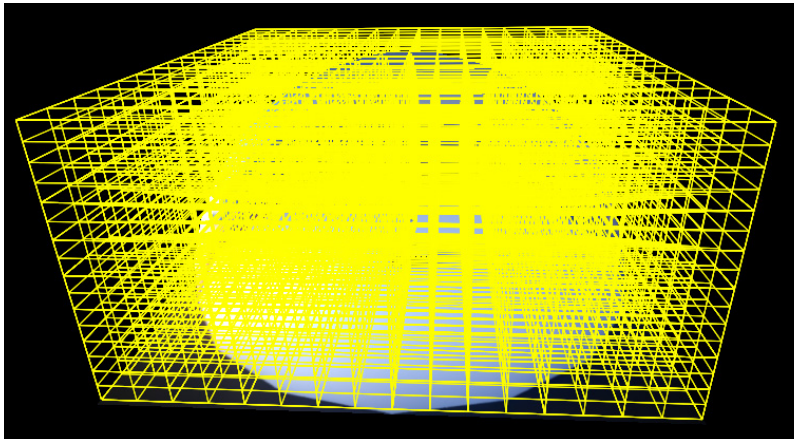
6th order SH approximation for R, G and B:	**108 floats**
6th order SH gradients for R, G, and B:	**324 floats**
	------------
	432 floats / sample
3rd order SH approximation for R, G, and B:	**27 floats**
3rd order SH gradients for R, G, and B:	**81 floats**
	------------
	108 floats / sample

Modern GPUs can typically store 1024 to 2048 floats in VS constant store

GPU Memory Requirements (Constant Store)

- If you have enough constant store available, you can send all nearby samples and their gradients to the vertex shader and do the interpolation per-vertex
- If this is too costly for you, interpolate on the CPU and send a single interpolated sample and interpolated gradient to the vertex shader
  - We did this for *Ruby: Dangerous Curves* and were very pleased with the results

System Memory Requirements

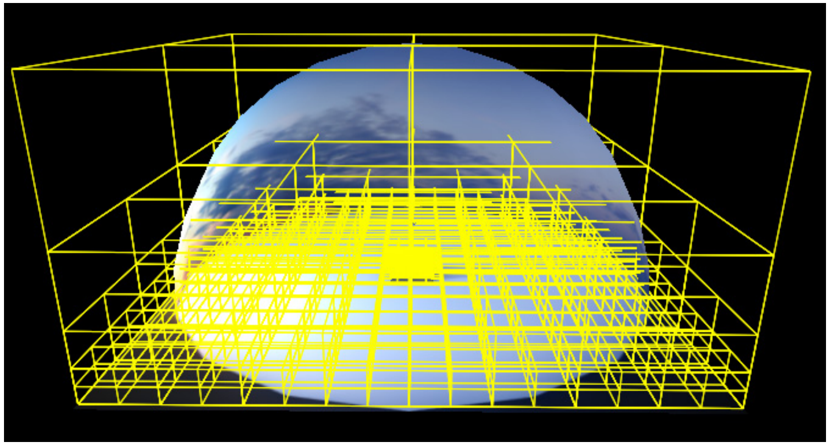


Uniform Subdivision Scene:

4913 Unique Samples

3rd Order SH + Gradients: ~2MB

6th Order SH + Gradients: ~8.2MB



Adaptive Subdivision Scene:

2301 Unique Samples

3rd Order SH + Gradients: ~970kB

6th Order SH + Gradients: ~3.8MB

Pros:

- Fast, efficient global illumination: A 3D light map for characters
- Much smaller memory cost compared to diffuse cubemaps
- Scalable: use higher/lower order SH approximations depending on needs
- Compatible with lower-end hardware

Cons:

- Doesn't handle dynamic lighting well
- Articulated characters are tricky
  - Works fine if evaluating irradiance samples with a vertex normal but PRT can be problematic
  - Instead of using Spherical Harmonic basis functions…
    - Valve uses a Cartesian basis in HalfLife2 (Ambient cube): http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf
    - Zonal Harmonics are more GPU rotation friendly. See Microsoft's GDC 2004 talk on LDPRT

Conclusion

- A lighting technique for dynamic characters in static scenes
- Compact storage of diffuse lighting functions using Spherical Harmonics for many points in a scene
- First order derivatives are used for Taylor series expansion of the incident lighting functions to increase the accuracy of each sample
- Adaptive scheme using an octree for efficiently subdividing a scene
- Interpolation between samples

Ruby: Dangerous Curves

- We used a technique, similar to the one presented today, for diffuse lighting in *Ruby: Dangerous Curves*
- We cheated a little though, rather than storing an entire volume, we only stored samples along each character's animation spline
- Rather than parameterize the samples by position, we parameterized by time

References

- G. Greger, P. Shirley, P. Hubbard, and D. Greenberg, The Irradiance Volume. *IEEE Computer Graphics & Applications,* 18(2):32-43, 1998.
- Cohen, Wallace. *Radiosity and Realistic Image Synthesis,* Academic Press Professional, Cambridge, 1993.
- J. Arvo. *Analytic Methods for Simulated Light Transport,* PhD thesis, Yale University, December 1995.
- Brennan, C., "Diffuse Cube Mapping", *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, Wolfgang Engel, ed., Wordware Publishing, 2002, pp. 287-289.
- Paul E. Debevec. *Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography*. SIGGRAPH 1998.
- Ramamoorthi, R., and Hanrahan, P., *An Efficient Representation for Irradiance Environment Maps*, SIGGRAPH 2001, 497-500.
- Green, R., *Spherical Harmonic Lighting: The Gritty Details*. 2003. Available from: http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.html
- Tomas Annen, Jan Kautz, Fredo Durand, and Hans-Peter Seidel, *Spherical Harmonic Gradients for Mid-Range Illumination*, Proceedings of Eurographics Symposium on Rendering, June 2004
- Sloan, P.-P., Kautz, J., Snyder, J., *Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments*, SIGGRAPH 2002.
- Pharr, M., Humphreys, G., *Physically Based Rendering*, Morgan Kaufmann, San Francisco, 2004.

Thank you

I’d like to thank Jan Kautz and Peter-Pike Sloan for providing their input on spherical harmonic gradients and Paul Debevec for his light probes (available for download from http://www.devec.org).

Questions?

Chris Oat

coat@ati.com

These slides are available for download:
http://www.ati.com/developer/