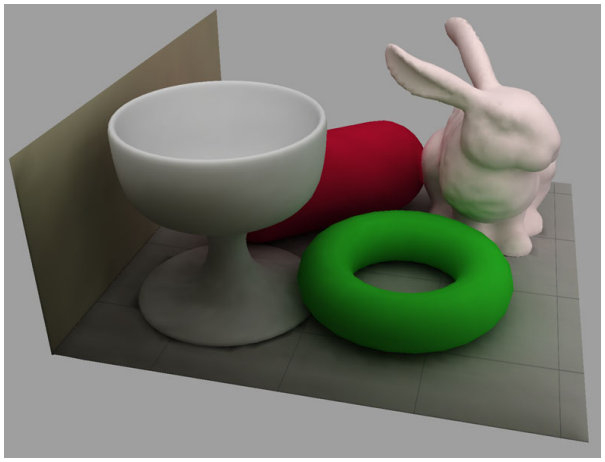
Spherical Harmonic Lighting: The Gritty Details

Robin Green,
R&D Programmer,
Sony Computer Entertainment America
robin_green@playstation.sony.com
January 16, 2003

Introduction

Spherical Harmonic lighting (SH lighting) is a technique for calculating the lighting on 3D models from area light sources that allows us to capture, relight and display global illumination style images in real time. It was introduced in a paper at Siggraph 2002 by Sloan, Kautz and Snyder as a technique for ultra realistic lighting of models. Looking a little closer at it's derivation we can show that it is in fact a toolbox of interrelated techniques that the games community can use to good effect.

The results are compelling and the code to compute them is actually straightforward to write, but the paper that introduces it assumes a lot of background knowledge from the first time reader. This paper is an attempt to provide this background, add some insights into the "why" questions, and hopefully give you all you need to add SH lighting to your game.



Illumination Calculations

If you have spent any time coding a 3D engine you should be familiar with the common lighting models, and concepts like specular highlights, diffuse colors and ambient lights should be second nature to you.

The simplest lighting model that you probably use is the *diffuse surface reflection model* sometimes known as "dot product lighting". For each light source the intensity (often expressed as an RGB colour) is multiplied by the scalar dot product between the unit surface normal **N** and the unit vector towards the light source **L**. This value is then multiplied by the surface colour giving the final reflected result:

Equation 1. Diffuse Surface Reflection

$$I = surfacecol * \sum_{i=1}^{nlights} lightcol_i * (\mathbf{N} \cdot \mathbf{L}_i)$$

One way of looking at this code fragment is to say that it first calculates the total amount of incoming light from all directions, then scales it by the cosine of the angle between **N** and **L** and multiplies the result by the surface reflection function (which for a diffuse surface is just a constant colour for all directions).

This is a simplification of the *rendering equation*, a formulation of the problem of producing images in computer graphics that is

based only on physics. It is the gold standard by which all realistic computer graphics lighting must be measured.

The problem with the rendering equation is that it is difficult to compute, and definitely not a real-time friendly operation. It is an integral over a hemisphere of directions where *L*, the light intensity function we are looking to calculate, appears on both sides of the equation:

$$L(\mathbf{x},\vec{\omega}_o) = L_e(\mathbf{x},\vec{\omega}_o) + \int_S f_r(\mathbf{x},\vec{\omega}_i \rightarrow \vec{\omega}_o) L(\mathbf{x}',\vec{\omega}_i) G(\mathbf{x},\mathbf{x}') V(\mathbf{x},\mathbf{x}') d\omega_i$$

Equation 2. The Rendering Equation, differential angle form. For more background on differential angles, see "*Radiosity and Realistic Image Synthesis*" by Cohen and Wallace, Academic Press, 1993

where

$L(\mathbf{x},\vec{\omega}_o)$ = the intensity reflected from position **x** in direction ω_o

$L_e(\mathbf{x},\vec{\omega}_o)$ = the light emitted from **x** by this object itself

$f_r(\mathbf{x},\vec{\omega}_i \rightarrow \vec{\omega}_o)$ = the BRDF of the surface at point **x**, transforming incoming light ω_i to reflected light ω_o

$L(\mathbf{x}',\vec{\omega}_i)$ = light from **x′** on another object arriving along ω_i

$G(\mathbf{x},\mathbf{x}')$ = the geometric relationship between **x** and **x′**

$V(\mathbf{x},\mathbf{x}')$ = a visibility test, returns 1 if **x** can see **x′**, 0 otherwise

To anyone used to writing raytracers the only difficult concept is the use of *differential angles* to represent rays and solving the actual integral itself, but the background is quite simple. I urge you to learn more about global illumination solutions as for this tutorial all I can give you is a taster.

Imagine that time is stationary. Picture a volume of space filled with photons – each cube of space can be said to have a constant *photon density*. Picture this field of photons in linear motion. At the very heart of rendering, we need to find out how many photons collide with a stationary surface for each unit of time, a value called *flux* which measured in *joules/second* or *watts*.

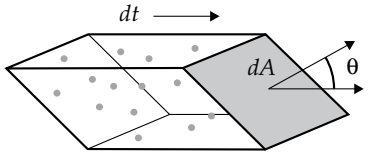


Figure 1. The origins of projected angle – calculating flux density.

To calculate the flux, note that all the photons that will hit the surface within a unit of time *t* lie within the volume swept behind the surface in the direction of flow – as the angle gets shallower, less photons hit the surface per unit of time because the swept volume is smaller. The relationship turns out to be proportional to the cosine of the angle between the surface normal and the direction of flow. As the lengths of the sides of this patch tend towards zero, we get the differential area of the patch. Dividing

the flux by the differential area, we get a value called the *irradiance*, measured in *watts/meter*^{2}.

That's the idea for one small patch and one direction of incoming light, but the relationship also holds when we consider all directions visible from a surface, as shown in Figure 2. As the angle gets shallower the *projected area* approaches zero. This is the reason behind the cosine (or dot product) in the diffuse shading equation.

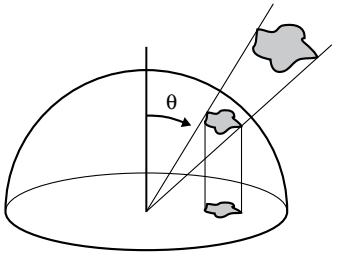


Figure 2. Projected Solid Angle.

That was just a quick taste of the background to the rendering equation, but I hope you can see that by looking more carefully at the derivation of our shading models and paying attention to the details we should be able calculate extremely realistic images based on purely physical principles. The only question left is how can we do this in real time? As a games programmer you may look at the rendering equation and throw your hands up in horror. An integral inside the inner loop of a shader? And how do we "integrate over a hemisphere"? How does this relate to any kind of hardware operations we have available? The rest of this document aims to take these abstract definitions and propose methods for calculating global illumination solutions and displaying them in a game friendly manner.

Monte Carlo Integration

We have a function to integrate, the function describing incoming light intensity, but we have no idea what that function looks like so there is no way we can calculate a result symbolically. The key to unlocking the puzzle is called *Monte Carlo Integration*, and it's all related to probability.

For more, see Siggraph 2001, "*State of the Art in Monte Carlo Ray Tracing*", Course 29

Peter Shirley,"*Realistic Ray Tracing*", A. K. Peters, 2001

Matt Pharr, "*Design of a Realistic Image Synthesis System*", 2002, available at http://graphics.stanford.edu/~mmp/book.pdf

First, some background in probability theory. A *random variable* is a value that lies within a specific domain with some distribution of values. For example, rolling a single 6-sided dice will return a discrete value from the set $f_i = \{1,2,3,4,5,6\}$ where each value has equal probability of occurring of $p_i = 1/6$. The *cumulative distribution function P(x)* is just a function that tells us the probability that when we roll a die, we will roll a value less than x. For example the probability that rolling a die will return a value less than 4 is $P(4) = 2/3$. If a variable has equal probability of taking any value within it's range it is known as a *uniform random*

variable. Rolling a die returns a *discrete* value for each turn – there are no fractional values it could return - but random variables can also have a *continuous* range, for example picking a random number within the range [3,7]. One continuous variable that turns up time and time again is the uniform random variable that produces values in the range [0,1) (i.e. including 0, excluding 1), and it is so useful for generating samples from other distributions it is sometimes called the *canonical random variable* and we will denoted it with the symbol ξ.

The function we are most likely to be working with is the *probability density function* (PDF) which tells us the relative probability that a variable will take on a specific value, and is defined as the derivative of the cumulative distribution function. Random variables are said to be distributed according to a particular PDF, and this is denoted by writing $f(x) \sim p(x)$. PDFs have positive or zero values for every valid number in the range and the function must integrate to 1.

$$\int_{-\infty}^{+\infty} p(x)dx = 1 \quad \text{where} \quad p(x) \geq 0$$

The probability that a variable x will take a value in the range $[a,b]$ is just the integral between a and b of the PDF.

$$P(x \in [a,b]) = \int_a^b p(x)dx$$

Every function using a random variable has an average value, the mean value, that it will tend to return most often if you take many, many samples. This is termed the *expected value* of the function, written $E[f(x)]$, which is calculated as:

$$E[f(x)] = \int f(x)p(x)dx$$

For example, let's find the expected or mean value for $f(x)=2-x$ over the range [0..2]. In order for the function to integrate to 1 over the range, we need to set $p(x)=1/2$. The integral gives us:

$$\begin{aligned} E[2-x] &= \int_0^2 \frac{2-x}{2} dx \\ &= \left[x - \frac{x^2}{4} \right]_0^2 = 1 \end{aligned}$$

Another way of calculating the expected value of a function is to take the mean of a large number of random samples from the function, which can be shown to converge towards the correct

answer as the number of samples approaches infinity (called the *Law of Large Numbers*):

$$E[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

We can combine these two results in one of the sneakiest tricks in the whole of Engineering Mathematics to give us an estimate of the integral of a function

$$\int f(x)dx = \int \frac{f(x)}{p(x)} p(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x)}{p(x)}$$

We just take lots and lots of point samples of the function f(x), scale each one by the PDF, sum the result and divide by the number of samples at the end. Intuitively you can see that samples with a higher probability of occurring are given less weight in the final answer. The other intuition from this equation is that distributions that are similar to the function being integrated have less variance in their answers, ending up with only a single sample being needed when *p(x) = f(x).* Another way of writing the Monte Carlo Estimator is to multiply each sample by a weighting function *w(x) = 1/p(x)* instead of dividing by the probability, leading us to the final form of the Monte Carlo Estimator:

Equation 3. The Monte Carlo Estimator

$$\int f(x) \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)w(x_i)$$

If we can guarantee that *p(x)* is a uniform distribution over the space we want to sample then we can just take point samples of our function, sum them, divide by the number of samples times *w(x)* and we have calculated an approximation to the integral of our function saving many multiplies. We know from the rendering equation that we want to integrate over the surface of a sphere, so all we need to do is generate evenly distributed points (more technically called *unbiased random samples*) over the surface of a sphere. Taking a pair of independent canonical random numbers ξ_x and ξ_y we can map this "square" of random values into spherical coordinates using the transform:

Equation 4. Mapping [0..1,0..1] random numbers into spherical coordinates.

$$\left(2\arccos\left(\sqrt{1-\xi_x}\right), 2\pi\xi_y\right) \rightarrow (\theta, \varphi)$$

The probability that we will sample any point on the surface of this unit sphere is the same for all samples, meaning that our weighting function is just the constant value 1/the surface area of a sphere, giving us a weighting function of $w(x) = 4\pi$. The resulting distribution of points is shown below.

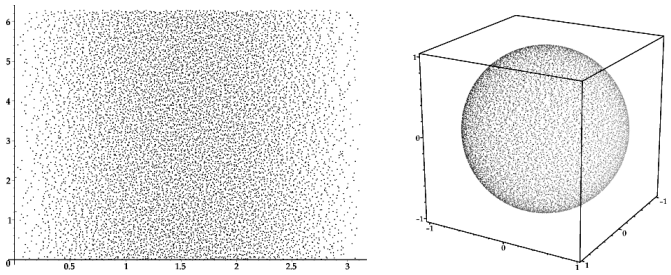


Figure 3. 10,000 unbiased stratified samples on a sphere. Presented in (θ,φ) angle space and in 3D projection.

An additional tool, to lower the variance of our sampling scheme, is to generate a grid of *jittered* samples. Divide the input square into N×N sample cells and pick a random point inside each cell. This sampling technique is called *stratified sampling* and it is provable that the sum of variances for each cell will never be higher than the variance for random samples over the whole range, and is often much lower. There are many more sampling tricks to make Monte Carlo integration more accurate for fewer samples, but this is all that is necessary for basic SH lighting.

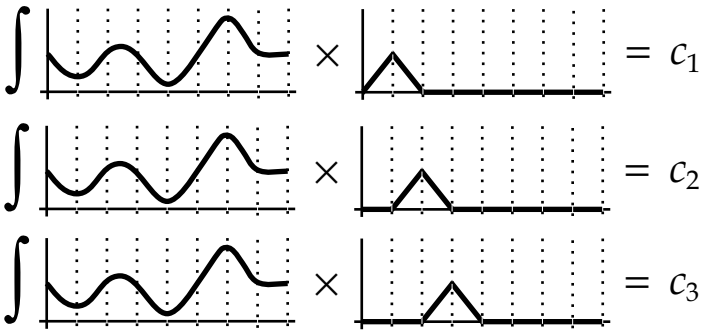
Here is some code for setting up a table of jittered samples. Don't worry, we'll be defining the meaning of the function `SH()` in a moment.

```
struct SHSample {
  Vector3d sph;
  Vector3d vec;
  double *coeff;
};

void SH_setup_spherical_samples(SHSample samples[], int sqrt_n_samples)
{
  // fill an N*N*2 array with uniformly distributed
  // samples across the sphere using jittered stratification
  int i=0; // array index
  double oneoverN = 1.0/sqrt_n_samples;
  for(int a=0; a<sqrt_n_samples; a++) {
    for(int b=0; b<sqrt_n_samples; b++) {
      // generate unbiased distribution of spherical coords
      double x = (a + random()) * oneoverN;  // do not reuse results
      double y = (b + random()) * oneoverN;  // each sample must be random
      double theta = 2.0 * acos(sqrt(1.0 - x));
      double phi = 2.0 * PI * y;
      samples[i].sph = Vector3d(theta,phi,1.0);
      // convert spherical coords to unit vector
      Vector3d vec(sin(theta)*cos(phi), sin(theta)*sin(phi), cos(theta));
      samples[i].vec = vec;
      // precompute all SH coefficients for this sample
      for(int l=0; l<n_bands; ++l) {
        for(int m=-l; m<=l; ++m) {
          int index = l*(l+1)+m;
          samples[i].coeff[index] = SH(l,m,theta,phi);
```

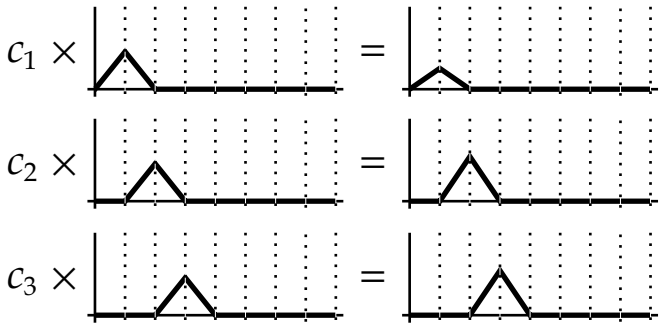
```
        }
      }
      ++i;
    }
  }
}
```

Orthogonal Basis Functions

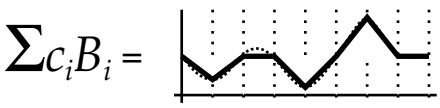
The SH lighting paper assumes knowledge of the use of *basis functions*. Basis functions are small pieces of signal that can be scaled and combined to produce an approximation to an original function, and the process of working out how much of each basis function to sum is called *projection*. To approximate a function using basis functions we must work out a scalar value that represents how much the original function *f(x)* is like the each basis function $B_i(x)$. We do this by integrating the product $f(x)B_i(x)$ over the full domain of *f*.



Using this projection process over all our basis functions returns a vector of approximation coefficients. If we scale the corresponding basis function by the coefficients…



… and sum the results we obtain our approximated function.



In the above example we have used a set of *linear basis functions*, giving us a *piecewise linear approximation* to the input function. There are many basis functions we can use, but some of the most interesting are grouped into a family of functions mathematicians call the *orthogonal polynomials*.

Orthogonal polynomials are sets of polynomials that have an intriguing property – when you integrate the product of any two of them, if they are the same you get a constant value and if they are different you get zero.

$$\int_{-1}^{1} F_m(x)F_n(x)dx = \begin{cases} 0 & \text{for } n \neq m \\ c & \text{for } n = m \end{cases}$$

We can also specify the more rigorous rule that integrating the product of two of these polynomials must return either 0 or 1, and this sub-family of functions are known as the *orthonormal basis functions*. Intuitively, it's a like the functions do not "overlap" each other's influence while still occupying the same space, the same effect that allows the Fourier transform to break a signal into it's component sine waves.

These families of polynomials are often named after the mathematicians who studied them, names like Chebyshev, Jacobi and Hermite. The one family we are most interested in are called the Legendre polynomials, specifically the *Associated Legendre Polynomials*. Traditionally represented by the symbol P, the associated Legendre polynomials have two arguments l and m, are defined over the range [–1,1] and return real numbers (as opposed to the ordinary Legendre Polynomials which return complex values – be careful not to confuse the two).

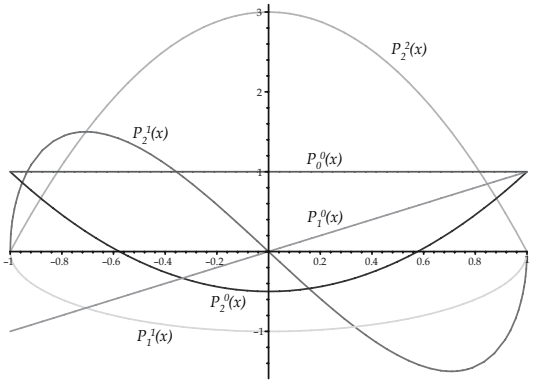


Figure 4. The first six associated Legendre polynomials.

The two arguments l and m break the family of polynomials into *bands* of functions where the argument l is the *band index* and takes any positive integer value starting from 0, and the argument m takes any integer value in the range $[0,l]$. Inside a band the polynomials are orthogonal w.r.t. a constant term and between

bands they are orthogonal with a different constant. We can diagram this as a triangular grid of functions per band, giving us a total of *n(n+1)* coefficients for an *n* band approximation:

$$P_0^0(x)$$
$$P_1^0(x), P_1^1(x)$$
$$P_2^0(x), P_2^1(x), P_2^2(x)$$
$$\dots$$

The process for evaluating Legendre polynomials turns out to be quite involved, which is why they're rarely used for approximating 1D functions. The usual mathematical definition of the series is defined in terms of derivatives of imaginary numbers and requires a series of nasty cancellations of values that alternate in sign and this is not a floating point friendly process. Instead we turn to a set of *recurrence relations* (i.e. a recursive definition) that generate the current polynomial from earlier results in the series. There are only three rules we need:

1 $$(l-m)P_l^m = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m$$

The main term of the recurrence takes the two previous bands *l*–1 and *l*–2 and generates a new higher band *l* from them.

2 $$P_m^m = (-1)^m(2m-1)!!(1-x^2)^{m/2}$$

The expression is the best place to start from as it is the only rule that needs no previous values. Note that *x*!! is the *double factorial function* which, as (2*m*–1) is always odd, returns the product of all odd integers less than or equal to *x*. We can use $P_0^0(x) = 1$ as the initial state for an iterative loop that hoists us up from 0 to *m*.

3 $$P_{m+1}^m = x(2m+1)P_m^m$$

This expression allows us to lift a term to a higher band.

For more, see "*Numerical Methods in C: The Art of Scientific Computing*", Cambridge University Press, 1992, pp 252-254

The method for evaluating the function is first to try to generate the highest P_m^m possible using rule 2, which if *l*=*m* is the final answer. Since *m*<*l* in all remaining cases, all that is left is to raise the band until we meet the required *l*. We do this by calculating P_{m+1}^m using rule 3 only once (stopping if *l*=*m*+1) and finally iterating rule 1 until the correct answer is found (noting that using rule 1 has less floating point roundoff error than iterating rule 3).

```
double P(int l,int m,double x)
{
  // evaluate an Associated Legendre Polynomial P(l,m,x) at x
  double pmm = 1.0;
  if(m>0) {
    double somx2 = sqrt((1.0-x)*(1.0+x));
```

```
    double fact = 1.0;
    for(int i=1; i<=m; i++) {
      pmm *= (-fact) * somx2;
      fact += 2.0;
    }
  }
  if(l==m) return pmm;
  double pmmp1 = x * (2.0*m+1.0) * pmm;
  if(l==m+1) return pmmp1;
  double pll = 0.0;
  for(int ll=m+2; ll<=l; ++ll) {
    pll = ( (2.0*ll-1.0)*x*pmmp1-(ll+m-1.0)*pmm ) / (ll-m);
    pmm = pmmp1;
    pmmp1 = pll;
  }
  return pll;
}
```

Spherical Harmonics

This is all fine for 1D functions, but what use is it on the 2D surface of a sphere? The associated Legendre polynomials are at the heart of the *Spherical Harmonics*, a mathematical system analogous to the Fourier transform but defined across the surface of a sphere. The SH functions in general are defined on imaginary numbers but we are only interested in approximating real functions over the sphere (i.e. light intensity fields), so in this document we will be working only with the *Real Spherical Harmonics*. When we refer to an SH function we will only be talking about the Real Spherical Harmonic functions.

Given the standard parameterization of points on the surface of a unit sphere into spherical coordinates (which we will look at more closely in a later section on coordinate systems):

Equation 5. The standard conversion from spherical to Cartesian coordinates.

$$(\sin\theta\cos\varphi,\ \sin\theta\sin\varphi,\ \cos\theta)\rightarrow(x,y,z)$$

the SH function is traditionally represented by the symbol y

Equation 6. The real spherical harmonic function y.

$$y_l^m(\theta,\varphi)=\begin{cases}\sqrt{2}K_l^m\cos(m\varphi)P_l^m(\cos\theta), & m>0\\ \sqrt{2}K_l^m\sin(-m\varphi)P_l^{-m}(\cos\theta), & m<0\\ K_l^0P_l^0(\cos\theta), & m=0\end{cases}$$

where P is the same associated Legendre polynomials we look at earlier and K is just a scaling factor to normalize the functions:

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}$$

In order to generate all the SH functions, the parameters l and m are defined slightly differently from the Legendre polynomials – l is still a positive integer starting from 0, but m takes signed integer values from $-l$ to l.

$$y_l^m(\theta,\varphi) \quad \text{where } l \in \mathbf{R}^+,\ -l \le m \le l$$

Sometimes it is useful to think of the SH functions occurring in a specific order so that we can flatten them into a 1D vector, so we will also define the sequence y_i

$$y_l^m(\theta,\varphi) = y_i(\theta,\varphi) \quad \text{where } i = l(l+1)+m$$

The code for evaluating an SH function looks like this:

```
double K(int l, int m)
{
  // renormalisation constant for SH function
  double temp = ((2.0*l+1.0)*factorial(l-m)) / (4.0*PI*factorial(l+m));
  return sqrt(temp);
}

double SH(int l, int m, double theta, double phi)
{
  // return a point sample of a Spherical Harmonic basis function
  // l is the band, range [0..N]
  // m in the range [-l..l]
  // theta in the range [0..Pi]
  // phi in the range [0..2*Pi]
  const double sqrt2 = sqrt(2.0);
  if(m==0) return K(l,0)*P(l,m,cos(theta));
  else if(m>0) return sqrt2*K(l,m)*cos(m*phi)*P(l,m,cos(theta));
  else return sqrt2*K(l,-m)*sin(-m*phi)*P(l,-m,cos(theta));
}
```

(Note: the fastest and most accurate way to implement `factorial(x)` is as a table of precalculated floating point values. You will never need more than 33 entries in the table.)

Traditionally, at about this point, papers using the SH functions like to show you tables of confusing polynomials, but I think it's more interesting to show what the functions actually look like when plotted as spherical functions.

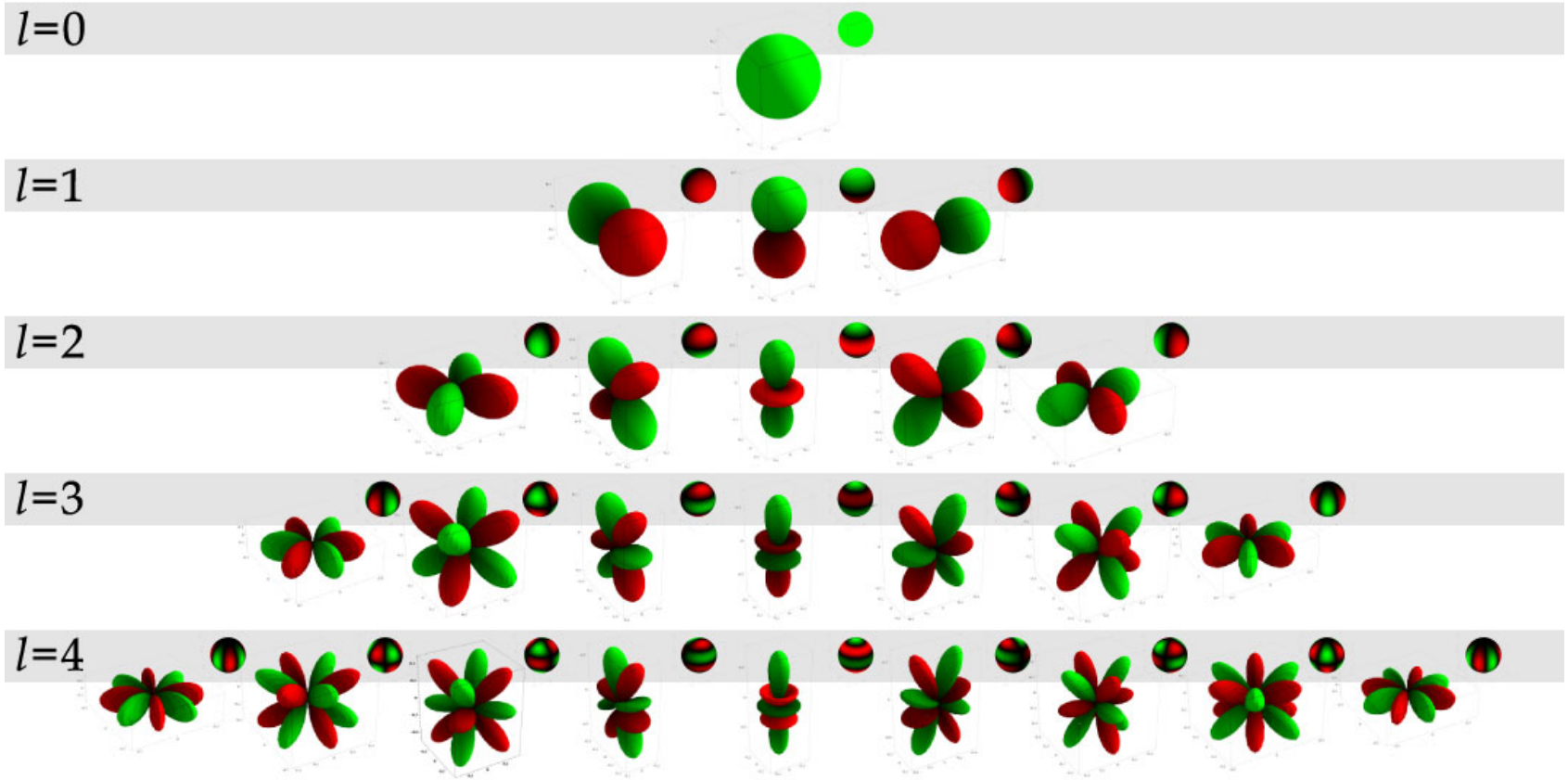


Figure 5. The first 5 SH bands plotted as unsigned spherical functions by distance from the origin and by colour on a unit sphere. Green (light gray) are positive values and red (dark gray) are negative.

Note how the first band is just a constant positive value – if you render a self-shadowing model using just the 0-band coefficients the resulting looks just like an accessibility shader with points deep in crevices (high curvature) shaded darker than points on flat surfaces. The $l = 1$ band coefficients cover signals that have only one cycle per sphere and each one points along the x, y, or z-axis and, as you will see later, linear combinations of just these functions give us very good approximations to the cosine term in the diffuse surface reflectance model.

SH Projection

The process for projecting a spherical function into SH coefficients is very simple. To calculate a single coefficient for a specific band you just integrate the product of your function f and the SH function y, in effect working out how much your function is like the basis function:

$$c_l^m = \int_S f(s) y_l^m(s) ds$$

(Note: The equation above is very carefully written not to include any mention of the parameterization we will use to generate points on the surface of the sphere – the value s merely represents some choice of a sample point. We will transform these equations into concrete, parameterized versions that we can actually calculate with in a moment, but for now we will stick with the abstract idea of sample points over the sphere S.)

To reconstruct the approximated function (notated by f capped with a tilde), we just take the reverse process and sum scaled copies of the corresponding SH functions:

$$\tilde{f}(s) = \sum_{l=0}^{n-1} \sum_{m=-l}^{l} c_l^m y_l^m(s) = \sum_{i=0}^{n^2} c_i y_i(s)$$

Now you can see why an n-th order approximation will require n^2 coefficients. It can be proven that the true function f could be reconstructed if we summed the infinite series of all SH coefficients, so every reconstruction we will make will be an approximation to the true function, technically known as a *band–limited* approximation where band–limiting is just the process of breaking a signal into it's component frequencies and removing frequencies higher than some threshold.

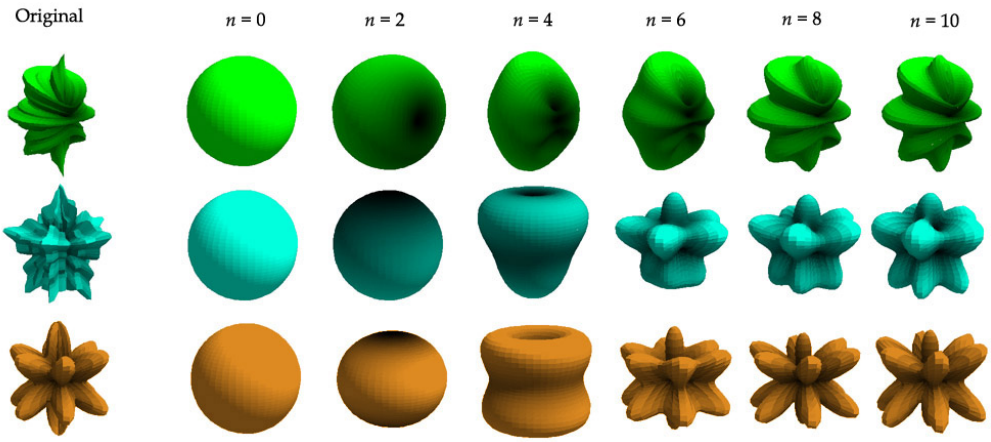


Figure 6. SH projection of functions with increasing orders of approximation.

Let's work through a concrete example of projecting a function into SH coefficients using Monte Carlo integration. First we need to decide on a parameterization of our sphere, so let's use the spherical coordinate system we defined earlier for the SH functions. Let's also choose a nice low-frequency function to integrate so that we don't have to generate too many coefficients to illustrate our point. How about two large monochromatic light sources at 90 degrees to each other and slightly rotated off-axis. We'll define these directly in spherical coordinates for now, but in our full program we'll be using a ray tracer to evaluate functions like this directly from geometry.

$$light(\theta,\varphi) = \max(0, 5\cos(\theta)-4)+ \max(0, -4\sin(\theta-\pi)*\cos(\varphi-2.5)-3)$$

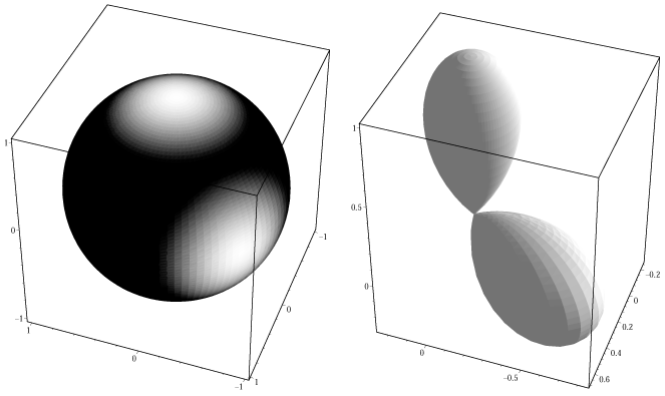


Figure 7. An example lighting function displayed as a color and a spherical plot.

Integrating some function *f* in spherical coordinates is done using the formula:

$$\int_0^{2\pi}\int_0^{\pi} f(\theta,\varphi)\sin\theta\, d\theta\, d\varphi$$

(Why the sin(θ) in there? Remember that integration is all about summing small patches of area on the surface of the sphere, and the integral is just the limit as the edge lengths of these square patches tend to zero. In this rectangular spherical coordinate parameterization, patches around the equator are going to have more effect on the final answer than the tiny patches around the pole and the sin(θ) term encodes this effect. Don't worry, it's about to disappear…)

Remember that to project a function into SH coefficients we want to integrate the product of the function and an SH function so we can write out our parameterised function for one coefficient as:

$$c_i = \int_0^{2\pi}\int_0^{\pi} light(\theta,\varphi)y_i(\theta,\varphi)\sin\theta\, d\phi\, d\varphi$$

This equation is great for symbolic integration using a package like Mathematica or Maple, but we have to do this numerically. We must evaluate this integral using Monte Carlo integration, so recalling the Monte Carlo estimator from earlier:

$$\int_S f\,ds \approx \frac{1}{N}\sum_{j=1}^{N} f(x_j)\,w(x_j)$$

where x_j is our array of pre-calculated samples and the function f is the product $f(x_j) = light(x_j)y_i(x_j)$.

As we have chosen all our samples to be unbiased w.r.t. area on the sphere, each sample has equal probability of appearing anywhere on the sphere giving us a probability function of $p(x_j) = 1/4\pi$ and so a constant weighting function $w(x_j) = 1/p(x_j) = 4\pi$. Also, the use of unbiased samples means that any other parameterization of the sphere would yield the same set of samples with the same probabilities, so we have magically factored out the parameterization of the sphere and our sin(θ) term disappears.

$$\begin{aligned} c_i &= \frac{1}{N}\sum_{j=1}^{N} light(x_j)y_i(x_j)4\pi \\ &= \frac{4\pi}{N}\sum_{j=1}^{N} light(x_j)y_i(x_j) \end{aligned}$$

Using the `SH_setup_spherical_samples` function from earlier we can precalculate our jittered, unbiased set of samples and the SH coefficients for each band we want to SH project. Our integration code is then just a simple loop of multiply-accumulates

into the correct elements of the SH vector, followed by a simple rescale of the results:

```
typedef double (*SH_polar_fn)(double theta, double phi);

void SH_project_polar_function(SH_polar_fn fn, const SHSample samples[],
double result[])
{
  const double weight = 4.0*PI;
  // for each sample
  for(int i=0; i<n_samples; ++i) {
    double theta = samples[i].sph.x;
    double phi   = samples[i].sph.y;
    for(int n=0; n<n_coeff; ++n) {
      result[n] += fn(theta,phi) * samples[i].coeff[n];
    }
  }
  // divide the result by weight and number of samples
  double factor = weight / n_samples;
  for(i=0; i<n_coeff; ++i) {
    result[i] = result[i] * factor;
  }
}
```

Applying this process to the light source we defined earlier with 10,000 samples over 4 bands gives us this vector of coefficients:

$$
\begin{bmatrix} 0.39925, \\ -0.21075, \ \ 0.28687, 0.28277, \\ -0.31530, -0.00040, 0.13159, \ \ 0.00098, -0.09359, \\ -0.00072, \ \ 0.12290, 0.30458, -0.16427, -0.00062, -0.09126 \end{bmatrix}
$$

Reconstructing the SH functions for checking purposes from these 16 coefficients is simply a case of calculating a weighted sum of the basis functions:

$$
\begin{aligned}
\tilde{f}(s) &= c_1 y_1(s) + c_2 y_2(s) + c_3 y_3(s) + \ldots \\
&= \sum_{i=1}^{n^2} c_i y_i(s)
\end{aligned}
$$

giving us this low frequency approximated light source:

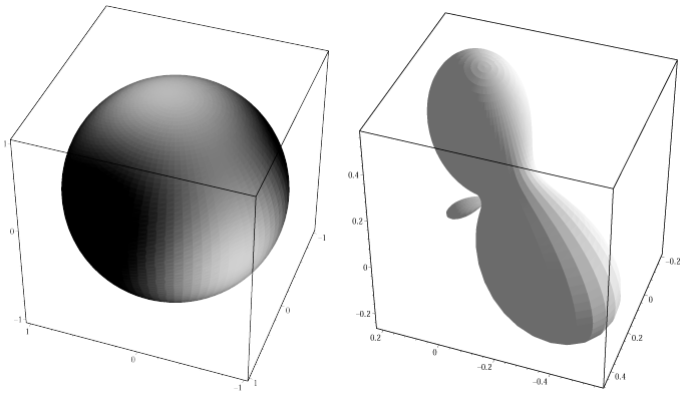


Figure 8. The reconstructed low frequency lighting function displayed as a color and a spherical plot.

Not a bad approximation given that it's only 16 coefficients. Note that it does seem to have some residual "fins" sticking out the back, and these will be manifested as unexpected illumination on the dark side of an object. These are caused by the high frequency components of the lighting function when we clamp it at 0 with the `max()` function – the discontinuity gives rise to "ringing" in the reconstructed signal. With higher and higher order approximations these fins will eventually disappear but a better method, which we'll come to in the section on designing light sources, is to *window* your input data by pre-filtering it with a Gaussian before SH-projecting it. Even though we're working on the surface of a sphere, all the old rules of signal processing still apply.

Properties of SH Functions

The SH functions have a bunch of interesting properties that make them more desirable for our purposes than other basis functions we could choose. Firstly, the SH functions are not just orthogonal but orthonormal, meaning if we integrate $y_i y_j$ for any pair of i and j, the calculation will return 1 if $i = j$ and 0 if $i \neq j$.

The SH functions are also *rotationally invariant*, meaning that if a function g is a rotated copy of function f, then after SH projection it is true that:

$$\tilde{g}(s) = \tilde{f}(R(s))$$

In other words, SH projecting the rotated function g will give you exactly the same results as if you had rotated the input to f before SH projecting. You're right, that is pretty confusing. It may not sound like such a big deal but this property is something that a lot of other compression methods cannot claim, e.g. JPEG's Discrete Cosine Transform encoding is not translation invariant which is what gives us the blocky look under high compression. In practical terms it means that by using SH functions we can guarantee that when we animate scenes, move lights or rotate

models, the intensity of lighting will not fluctuate, crawl, pulse or have any other objectionable artifacts.

The next property is the killer one. We need to do lighting, so in general terms we will be taking some description of incoming illumination and multiplying it by some kind of description of the surface reflectance (which we will be calling a *transfer function*) to get the resulting reflected light, but we need to do this over the entire sphere of incoming light. We need to integrate:

$$\int_S L(s)t(s)ds$$

where L is the incoming light and t is the transfer function. If we project both the illumination and transfer functions into SH coefficients then orthogonality guarantees that the integral of the function's products is the same as the dot product of their coefficients:

Equation 7. Integrating the product of two SH functions by evaluating a dot product of their coefficients.

$$\int_S \tilde{L}(s)\tilde{t}(s)ds = \sum_{i=0}^{n^2} L_i\, t_i$$

We have collapsed an integration over the sphere into a single dot product over the SH coefficients, just a series of multiply-adds. This is the key to the whole process – by projecting functions into *SH space* we can convert integration over a sphere into a very fast operation.

This dot product returns a single scalar value which is the result of the integration, but there is another technique we can use for transforming SH functions. Here's how the argument goes:

Say we have some arbitrary spherical light source function $a(s)$ that we don't know yet. We also have some shadowing function for a particular point on the surface $b(s)$ that describes how light at that point is shadowed (e.g. there's a nose above us that will block light coming from that direction), and we can evaluate it using a ray tracer. We want a way to transform the coefficients of the incoming light into another set of coefficients for a light that has been masked by the shadow function, and we'll call the result $c(s)$. We can construct a linear operation that maps the SH projection of the light source $a(s)$ directly to the SH projection of the shadowed light source $c(s)$ using a *transfer matrix* without having to know the lighting function $a(s)$. To build the transfer matrix $\mathbf{M}$, where each element of the matrix are indexed by i and j, the calculation is:

Equation 8. The triple product for calculating elements of a transfer matrix.

$$\mathbf{M}_{ij} = \int_S b(s)y_i(s)y_j(s)\,ds$$

The result is a matrix that we can use to transform from a light source into a shadowed light source using a simple matrix-vector multiply:

Equation 9. Applying a transfer matrix to a vector of SH coefficients.

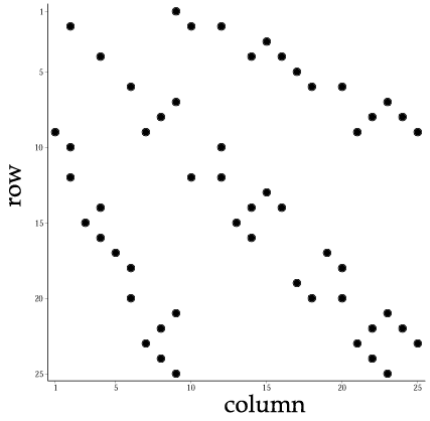
$$c_i = \sum_{j=1}^{n^2} \mathbf{M}_{ij} a_i$$

Let's try an explicit example. What would happen if we found a magic shadowing function that looks exactly like one of the SH functions; for example what if $b(s) = y_2^2(s)$? This means we will be calculating a *triple product* of SH functions:

$$\int_S y_2^2(s) y_i(s) y_j(s)\, ds$$

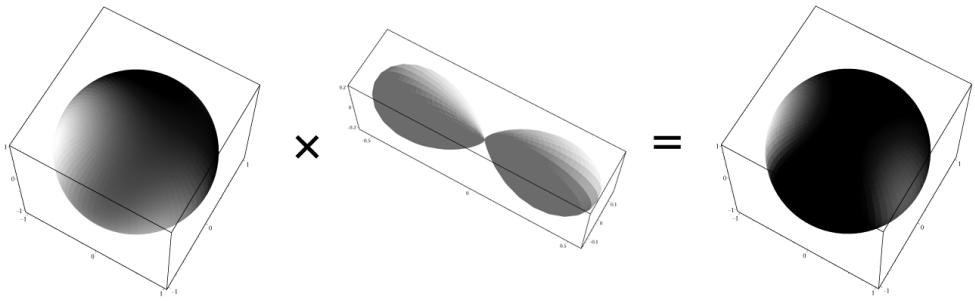
The resulting matrix is mostly sparse, and a plot of the non-zero elements in a 25×25 matrix looks like this:

Figure 9. The non-zero entries in matrix **M** in our contrived triple-product transfer matrix example.



Applying the matrix to the SH coefficients of a light source gives us another vector of SH coefficients, which gives us exactly the same results as if we had multiplied the light source with the mask before SH projecting:

Figure 10. The result of applying the example transfer matrix to a light source.



As you can see this is a different way of recording shadowing at a point on a model without forcing us to do the final integral, and

we will use it to good effect when we preprocess view dependant glossy specular surfaces.

Rotating Spherical Harmonics

The last property of SH functions is the most difficult to code, and probably the place where most people will get stuck. We have asserted that SH functions are rotationally invariant, but how do we actually rotate an SH projected function? The answer is not simple. The first question to answer is "what form of rotation are you talking about?" Do you want rotations in terms of Euler angles (α,β,γ), and if so which order of axes are you rotating about? *XYZ*, *ZYX* or *ZYZ*? How about specifying an axis and angle rotation by using a quaternion? How about generalizing rotations into 3×3 rotation matrices with all the associated redundancy of symmetries? Despite Sloan, Kautz and Snyder saying that SH functions have "simple rotation", they aren't telling the whole story.

Kautz et al, "Fast, Arbitrary BRDF Shading", 13th Eurographics Workshop on Rendering, 2002, Section 3 and Appendix.

What we can say about the SH rotation process, from the rules of orthogonality, is that it is a linear operation and that coefficients between bands do not interact. In practical terms this means that we can rotate a vector of SH coefficients into another vector of SH coefficients using a single $n^2 \times n^2$ rotation matrix and that the matrix will be *block diagonal sparse*, looking something like this:

$$R_{SH} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \cdots \\ 0 & 0 & 0 & 0 & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

So true, *using* the rotation operation could be seen as "a simple computation" once you have the rotation coefficients handy but constructing the rotation coefficients efficiently is far from simple.

As a quick aside, let's look at another representation for the SH functions. So far we have been expressing SH functions in terms of spherical coordinates, but we can just as easily convert them to implicit functions on (x,y,z) by substituting in the spherical to Cartesian coordinate conversion formula and canceling out terms. Doing so we come up with a surprisingly simple set of expressions:

Equation 10. Cartesian version of the first few real SH functions.

	$m=-2$	$m=-1$	$m=0$	$m=1$	$m=2$
$l=0$			$\frac{1}{2}\sqrt{\frac{1}{\pi}}$		
$l=1$		$\frac{1}{2}\sqrt{\frac{3}{\pi}}\frac{y}{r}$	$\frac{1}{2}\sqrt{\frac{3}{\pi}}\frac{z}{r}$	$\frac{1}{2}\sqrt{\frac{3}{\pi}}\frac{x}{r}$	
$l=2$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{yx}{r^2}$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{yz}{r^2}$	$\frac{1}{4}\sqrt{\frac{5}{\pi}}\frac{2z^2-x^2-y^2}{r^2}$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{zx}{r^2}$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{x^2-y^2}{r^2}$

where

$$r=\sqrt{x^2+y^2+z^2} \quad \left(\text{n.b. usually } r=1\right)$$

To use these functions simply pick a point (x,y,z) on the unit sphere and crank it through the equation above to calculate the SH coefficient in that direction. It is possible to use this form of equation for SH projection, but they turn out to be more useful to us as symbolic expressions.

We can build a rotation matrix for SH functions by building a matrix where each element is calculated using symbolic integrating of a rotated SH sample with an unrotated version:

$$\mathbf{M}_{ij}=\int_S y_i(\mathbf{R}s)\,y_j(s)\,ds$$

This will build a $n^2\times n^2$ matrix of expressions that will map an unrotated vector of SH coefficients into a rotated one. For example, using the explicitly parameterised formulation:

$$\mathbf{M}_{ij}=\int_o^{2\pi}\int_0^{\pi} y_i(\theta,\varphi+\alpha)\,y_j(\theta,\varphi)\sin(\theta)\,d\theta\,d\varphi$$

for the first three bands gives us a 9×9 matrix for rotating about the z-axis:

$$\mathbf{Z}_\alpha = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & 0 & \sin(\alpha) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\sin(\alpha) & 0 & \cos(\alpha) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cos(2\alpha) & 0 & 0 & 0 & \sin(2a) \\ 0 & 0 & 0 & 0 & 0 & \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 0 & -\sin(2\alpha) & 0 & 0 & 0 & \cos(2\alpha) \end{bmatrix}$$

This matrix expands into higher bands as you would expect, with band N using the sine and cosine of Nα.

This technique looks great for low order SH functions – you simply decompose any rotation into a series of simpler rotations and recompose the results. In reality it quickly turns into a royal pain-in-the-ass for anything larger than a 2nd order SH function.

Firstly, what is the minimum number of rotations we need to allow us to transform an SH function to any possible orientation? If we use a *ZYZ* formulation we can get away with only two rotations, and one of them we already have the formula for! So, how to rotate about the *y*-axis? We can decompose it into a rotation of 90° about the *x*-axis, a general rotation about the *z*-axis followed finally a rotation by -90° about the *x*-axis. Great, the *x*-axis rotation is a fixed angle so we can just tabulate it as an array of constant floats:

$$\mathbf{X}_{-90} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{bmatrix} \quad \mathbf{X}_{+90} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & -\frac{\sqrt{3}}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{bmatrix}$$

Taking a step back, let's look at the computational cost of this process. In matrix notation, we are calculating:

$$\mathbf{R}_{SH}(\alpha, \beta, \gamma) = \mathbf{Z}_\gamma \, \mathbf{X}_{-90} \, \mathbf{Z}_\beta \, \mathbf{X}_{+90} \, \mathbf{Z}_\alpha$$

That's 4 different 9×9 matrix multiplications, plus associated trig functions. Given that cost of matrix-matrix multiplication is $O(n^3)$, a naïve implementation would use 2916 multiply adds. We can use the sparsity of the matrices to get this down to around 340 multiplies for a 5th order rotation, but it's not as cheap as it could be.

How about combining the rotations and multiplying through the whole operation into one big explicit expression? For bands higher than 1 this turns out to produce some very scary trig expressions. Here is the matrix for the first two bands:

$$\mathbf{R}_{SH}(\alpha,\beta,\gamma)=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \begin{pmatrix}\cos\alpha\cos\gamma \\ -\sin\alpha\sin\gamma\cos\beta\end{pmatrix} & \sin\alpha\sin\beta & \begin{pmatrix}\cos\alpha\sin\gamma \\ +\sin\alpha\cos\gamma\cos\beta\end{pmatrix} \\ 0 & \sin\gamma\sin\beta & \cos\beta & -\cos\gamma\sin\beta \\ 0 & \begin{pmatrix}-\cos\alpha\sin\gamma\cos\beta \\ -\sin\alpha\cos\gamma\end{pmatrix} & \cos\alpha\sin\beta & \begin{pmatrix}\cos\alpha\cos\gamma\cos\beta \\ -\sin\alpha\sin\gamma\end{pmatrix} \end{bmatrix}$$

Equation 11. Analytical solution for the SH rotation matrix by ZYZ Euler angles (α,β,γ) for the first two bands.

This matrix is useful for debugging, but to use it we have to convert our game engine to ZYZ rotations with all the associated gimbal lock problems when rotations end up aligning. Wasn't this the reason we all converted to quaternions? There is a trick we can use to prevent us from entering this arena of pain and also speed up the calculation. One of the fundamental properties of rotation matrices in 3D is their numerous symmetries and we can exploit these to our advantage. Given an ordinary 3×3 rotation matrix **R**:

$$\mathbf{R}=\begin{bmatrix} R_{XX} & R_{XY} & R_{XZ} \\ R_{YX} & R_{YY} & R_{YZ} \\ R_{ZX} & R_{ZY} & R_{ZZ} \end{bmatrix}$$

we can reconstruct the trigonometric functions of the *ZYZ* Euler angles (α,β,γ) directly using these identities:

$$\cos\beta = R_{ZZ} \qquad \sin\beta = \sqrt{1-R_{ZZ}^2}$$

$$\cos\alpha = \frac{R_{ZX}}{\sin\beta} \qquad \sin\alpha = \frac{R_{ZY}}{\sin\beta}$$

$$\cos\gamma = \frac{-R_{XZ}}{\sin\beta} \qquad \sin\gamma = \frac{R_{YZ}}{\sin\beta}$$

This formulation breaks down when $\sin\beta = 0$, but in this case the rotation matrix would be written:

$$\mathbf{R}_{SH}\left(\alpha,\begin{Bmatrix}0\\ \pi\end{Bmatrix},\gamma\right)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos(\alpha\pm\gamma) & 0 & \pm\sin(\alpha\pm\gamma)\\ 0 & 0 & \pm 1 & 0\\ 0 & -\sin(\alpha\pm\gamma) & 0 & \pm\cos(\alpha\pm\gamma)\end{bmatrix}$$

So we can arbitrarily decide how to divide the total *z*-rotation between α and γ. Forcing $\gamma = 0$ gives us the special case identities:

$$\cos\alpha = R_{YY} \qquad \sin\alpha = -R_{YX}$$

$$\cos\gamma = 1 \qquad \sin\lambda = 0$$

Both of these formulations work fine as a stopgap for the first couple of bands of SH coefficients, but we really need a fast, general case solution for any number of bands. This is where we hit the unexplored, bleeding edge of current game research.

To solve this we need to dig into the roots of where the SH functions originally came from. The SH functions were originally devised to describe the distribution of angular momentum in a single atom at the quantum level and this explains why the arguments *l* and *m* are integers – they are the indivisible *quantum numbers* describing the state of the atom. The area of science with most experience of writing programs using SH functions is called *Computational Chemistry*, in which researchers try to model the interactions of atoms inside molecules at the quantum level to better understand how they function. As recently as 1999 papers of basic research have been published describing new ways of rotating real SH functions that are far more efficient that the traditional *Wigner D functions*. The emphasis of these papers until recently has been on explicit formula for axis-aligned special cases (great for modeling lattices of atoms), but we need a more general formulation for our computer graphics work.

We really need a set of *recurrence relations*, recursive functions that build SH rotation matrices for band *l*+1 from band *l*. This way, low order approximations are guaranteed less compute complexity than higher order ones. Research has uncovered three papers on recurrence relations for rotating real spherical harmonics and they have many similarities:

1 Ivanic J and Ruedenberg K, "*Rotation Matrices for Real Spherical Harmonics, Direct Determination by Recursion*", J. Phys Chem. A, Vol. 100, 1996, pp 6342-6347. See also "*Additions and Corrections: Rotation Matrices for Real Spherical Harmonics*", J. Phys Chem. A, Vol. 102, No.45, 1998, pp 9099-9100

2 Choi, Cheol Ho et al, "*Rapid and stable determination of rotation matrices between spherical harmonics by direct recursion*", J. Chem. Phys. Vol 111, No. 19, 1999, pp 8825-8831

3 Blanco, Miguel A et al, "*Evaluation of the rotation matrices in the basis of real spherical harmonics*", J. Molecular Structure (Theochem), 419, 1997, pp19-27

I have successfully implemented Blanco's paper but would recommend either Ivanic or Choi's papers as they are more efficient algorithms, if you can fight your way through the math. Choi reports in a personal email that working in the complex space is up to 10 times faster than working in the real space but requires a complex to real postprocess ("Complex makes life easier!" he writes). I am working on implementing both algorithms for a later paper. For a little more information on Ivanic's algorithm, see Appendix 1 of this document.

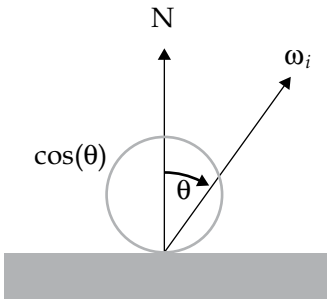
SH Lighting Diffuse Surfaces

Now we have covered the properties of SH functions and coded up some tools for manipulating them we can finally get to use them for generating some lighting for our models. We will go through a number of lighting techniques showing how each one is defined and how it is implemented.

Let's assume we have already loaded a description of our polygonal model into an internal database. We are only interested in point sampling the world using a raytracer, so all we need is a list of vertices, normals and triangles. First we process the vertex-normal pairs into a set of unique "lighting points" by duplicating vertices with multiple normals and concatenating vertices that are shared across smooth surfaces (this can be done very quickly using an STL map). Next we loop through all the lighting points in the model calculating a *transfer function* for each. The transfer function is a function that when dotted with an incoming luminance function (i.e. multiplied and integrated), gives us the approximated lighting for that point.

There are three different types of transfer function we can generate for diffuse surfaces, each one progressively more complex to calculate so we'll go through them in order.

1 **Diffuse Unshadowed Transfer**
Returning to the Rendering Equation we can strip it down to it's barest essentials, just a light source and a surface point assumed to lie on some oriented flat plane, to generate an unshadowed image using only direct illumination



$$L(\mathbf{x},\omega_o)=\int_S f_r(\mathbf{x},\omega_o,\omega_i)L_i(\mathbf{x},\omega_i)H(\mathbf{x},\omega_i)d\omega_i$$

where:

$L(\mathbf{x},\omega_o)$ = the amount of light leaving point **x** along vector ω_o
$f_r(\mathbf{x},\omega_o,\omega_i)$ = The BRDF at point **x**
$L_i(\mathbf{x},\omega_i)$ = incoming light at point **x** along vector ω_i
$H(\mathbf{x},\omega_i)$ = the *geometric* or *cosine term*, as described earlier.

Remembering that a diffuse BRDF reflects light equally in all directions, so we can optimize this equation quite dramatically. Light is reflected equally so the lighting is *view independent* and our viewing angle ω_o disappears. Our BRDF is just a simple, constant scalar which can be taken outside of the integral leaving us with just three elements: The light source L_i, a simplified cosine term and a linear scale factor.

$$L_{DU}(\mathbf{x})=\frac{\rho_{\mathbf{x}}}{\pi}\int_S L_i(\mathbf{x},\omega_i)\max(\mathbf{N}_{\mathbf{x}}\bullet\omega_i,0)d\omega_i$$

where

$\rho_{\mathbf{x}}$ = the surface albedo at point **x**.
$\mathbf{N}_{\mathbf{x}}$ = the surface normal at point **x**.

(Note for pedants: here we have switched to using the albedo $\rho_{\mathbf{x}}$ as the measure of reflectivity for Lambertian diffuse reflection. The albedo is the ratio of emitted radiance over irradiance, and for diffuse surfaces it collapses to $\pi\rho_{\mathbf{x}}$ so we need only specify a scalar $\rho_{\mathbf{x}}$ that takes on values in the range [0,1]. Just like RGB colors.)

For proof, see Cohen and Wallace, "Radiosity and Realistic Image Synthesis", Academic Press Professionsal, 1993, pp32-33

Separating out the light source from the transfer function (which we shall call M^{DU} for diffuse unshadowed transfer) we get the function we are looking to approximate with an SH function. It's just the geometric term, the cosine of the angle between the normal and the light source, clamped at zero:

$$M^{DU} = \max(\mathbf{N} \bullet \mathbf{s}, 0)$$

In order to calculate this function we will take our list of precalculated rays and SH coefficients and run through them, dotting each one with the surface normal at our sample point to see if it is inside the upper hemisphere.

```
// for each sample
for(int i=0; i<n_samples; ++i) {
  // calculate cosine term for this sample
  double H = DotProduct(sample[i].vec, normal);
  if(H > 0.0) {
    // ray inside upper hemisphere so...
    // SH project over all bands into the sum vector
      for(int j=0; j<n_coeff; ++j) {
        value = Hs * sample[i].coeff[j];
        result[j + red_offset  ] += albedo_red   * value;
        result[j + green_offset] += albedo_green * value;
        result[j + blue_offset ] += albedo_blue  * value;
      }
  } else {
    // ray not in upper hemisphere
  }
}
// divide the result by probability / number of samples
double factor = area / n_samples;
for(i=0; i<3*n_coeff; ++i) {
  coeff[i] = result[i] * factor;
}
```

This will produce pictures much like normal dot-product lighting, except using arbitrarily complex SH area light sources.

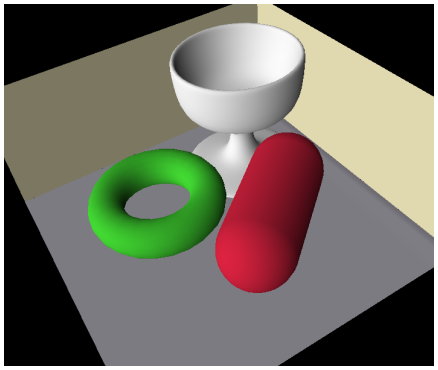


Figure 11. A rendering using 5th order diffuse unshadowed SH transfer functions (25 coefficients). In effect, lighting using just the geometric term.

2 Shadowed Diffuse Transfer

By adding a visibility term to the simplified Rendering Equation,

we can add self shadowing to the lighting model, and our lighting starts to really get interesting.

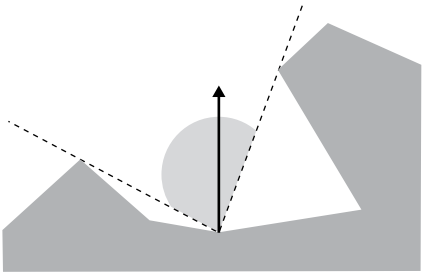
$$L(\mathbf{x}) = \frac{\rho_{\mathbf{x}}}{\pi} \int_{\Omega} L_i(\mathbf{x}, \omega_i) V(\omega_i) \max(\mathbf{N}_{\mathbf{x}} \bullet \omega_i, 0) d\omega_i$$

where

$V(\omega_i)$ = visibility test that returns 0 if the ray ω_i is blocked by self, 1 otherwise.

With only this tiny change, points on the surface of an object are no longer assumed to be sitting on an infinite plane but instead interact with their surrounding geometry and have access to their fields of view of incoming area light sources. The resulting transfer function for diffuse shadowed lighting M^{DS} is:

$$M^{DS} = V(\omega_i) \max(\mathbf{N} \bullet \omega_i, 0)$$



This one effect, sometimes called occluded ambient, is the most powerful difference between traditional CG and Global Illumination images. Calculating the transfer function requires us to trace a ray from the current point through the polygon database to find any hits – note that we don't need any geometric information from the hit, just a boolean that it occurred.

```
// for each sample
for(int i=0; i<n_samples; ++i) {
  // calculate cosine term for this sample
  Hs = DotProduct(sample[i].vec, normal);
  if(Hs > 0.0) {
    // ray inside upper hemisphere...
    if(!self_shadow(pos,sample[i].vec)) {
      // ray hits nothing, add in its the contribution:
      for(int j=0; j<n_coeff; ++j) {
        value = Hs * sample[i].coeff[j];
        result[j + red_offset  ] += albedo_red   * value;
        result[j + green_offset] += albedo_green * value;
        result[j + blue_offset ] += albedo_blue  * value;
      }
```

```
    } else {
      // ray hits self...
    }
  } else {
    // ray not in upper hemisphere
  }
}
// divide the result by number of samples
double factor = area / n_samples;
for(i=0; i<3*n_coeff; ++i) {
  coeff[i] = result[i] * factor;
}
```

How do we implement the `self_shadow()` function? That's up to your raytracer, but there are some things to note about the SH projection process make the code easier to write.

Firstly, if you decide to use an acceleration data structure like a voxel grid, hierarchical bounding box tree or BSP tree, remember that every ray origin will be inside the object's bounding box so there is no need for an initial ray-box intersection to find the starting point for the raytrace. Just hash the starting point into the data structure and start there.

Secondly, the point of `self_shadow` is to find occlusions so we want our shadow feelers to hit the object and we are going to be throwing out rays in all directions from a vertex *on* the model. The problem is that polygons adjacent to the vertex are always going to be coplanar to many rays. Any ray tested against a polygon that included the ray origin as a vertex will at best return a hit at the origin (requiring an epsilon tests to exclude hits too close to the origin) or at worst a hit somewhere along the ray. This can lead to incorrect shadowing on your object if you are not careful, so if possible retain some face adjacency information when loading your model and exclude polygons that share the current vertex from your shadow tests.

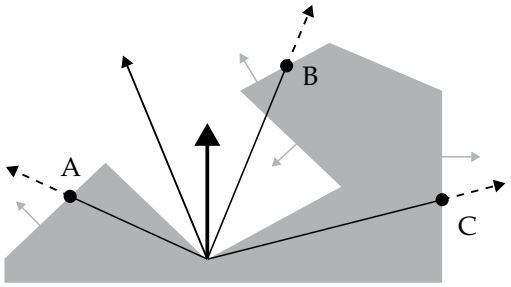


Figure 12. Examples of ray-polygon gotchas when tracing rays from a vertex on a polygon model.

The third issue is to do with single sided polygon tests. In order for ray-poly tests to correctly return occlusion we cannot use single-sided polygon tests. Excluding a ray-polygon test because the polygon "faces away" from the ray does not work if the ray

started from within the model, (intersections *A* and *C* in the illustration above) as is the case for many shadow rays. This limitation also means that, in general, we must have *manifold objects* – objects with fully enclosed skins. Objects with self intersecting surfaces will have incorrect shadowing (after reconstruction with Gouraud shading) if there are no vertices at the intersections. Be careful when lighting single-thickness objects that you are sure that any shadow feeler will be correctly occluded by other objects (as the illustration below fails to do correctly on the back wall!)

Shadow rays are also free to return an intersection as soon as an occluding polygon is found (intersection *B*) as the order of intersection is not important, just that one exists.

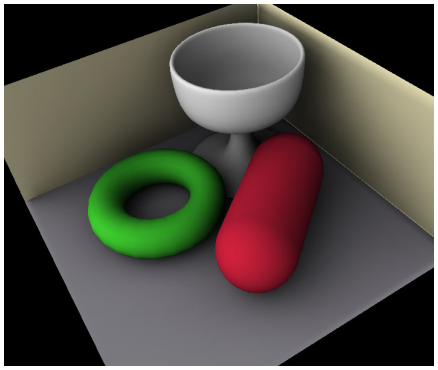


Figure 13. A rendering using 5th order diffuse shadowed SH transfer functions. Note the soft shadowing from the constant hemisphere light source.

3 **Diffuse Interreflected Transfer**

The last method of diffuse lighting is the most striking. The interesting part of the Rendering Equation is where it recursively adds in light not arriving directly from a light source, but as secondary reflected light from other polygons visible to a point on the model. Expressing this in an integral we can write:

$$L_{DI}(\mathbf{x}) = L_{DS}(\mathbf{x}) + \frac{\rho_{\mathbf{x}}}{\pi}\int_{\Omega}\bar{L}(\mathbf{x}', \omega_i)\left(1 - V(\omega_i)\right)\max(\mathbf{N}_{\mathbf{x}} \bullet \omega_i, 0)\, d\omega_i$$

where

$L_{DS}(\mathbf{x})$ = diffuse shadowed lighting from previous section.

$V(\omega_i)$ = visibility test from previous section.

$\bar{L}(\mathbf{x}', \omega_i)$ = light reflected from another point **x′** on the same model towards point **x**.

The interreflected light transfer function is difficult to mathematically describe compactly and not really that

illuminating, but the algorithm for generating it is easier to describe. There are four steps:

a) For each shading point **x** on the model, calculate the direct lighting transfer function at that point (i.e. diffuse shadowed lighting from the previous section).

b) Next, fire a rays from your current point until one hits another triangle on the object. Linearly interpolate the SH functions at each corner of the triangle using the barycentric coordinates of the hit. This transfer function is the amount of light being reflected back towards your shading point.

c) Multiply this reflected light by the dot product between the ray and the surface normal at **x** (a vector-scalar multiply that scales how much this interreflected light is reflected according to the cosine term) and sum it into an empty SH vector. Once all rays have been cast, divide the accumulated values by the number of samples and Monte Carlo weighting term as usual.

d) Once all shading points have been calculated, this new set of SH vectors is one bounce of diffusely interreflected light, and only the interreflected light. For additional bounces, repeat the raytracing using this new set of values as the starting light intensities at each triangle vertex. Repeat until no energy has been transferred, or until you reach N bounces. Finally, sum all bounces plus the direct illumination into one list of SH vectors.

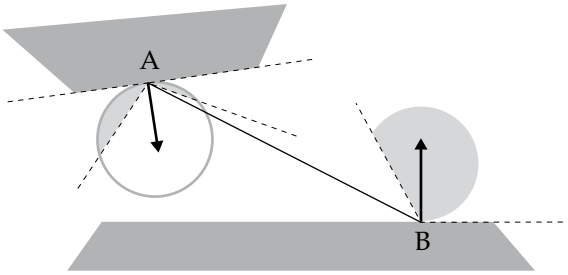
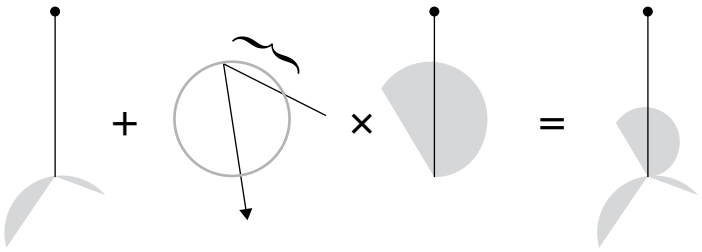


Figure 14. A single sample of diffuse self transfer, the downward facing plane A fires a ray.

Geometrically, the idea of interreflected light is simple. Each point on the model already knows how much direct illumination it has, encoded in the form of a transfer function. We fire rays to find sample points that can reflect light back onto our position and add a cosine weighted copy of that transfer function back into our own. For example, point *A* in the illustration above has fired a ray and hit point *B*. The transfer function at *B* is added to *A* like this:



Note that all SH functions occur in the same coordinate system so summing them is a valid operation. See how point *A*, when lit by an SH lightsource, will be illuminated by light from above, even though it cannot directly see any. The assumption here is that illumination doesn't vary across the model (i.e. point *B* has exactly the same lighting function as point *A*). This is the key to SH lighting: Low frequency light sources and very small light source variance across an object.

This time your raytracer has more work to do. Not only do you have to find intersections with the model, but you need to find the closest hit. If the closest hit faces towards the lighting point we need to find the exact SH lighting function at that point. We can speed up the process by remembering which rays in the previous pass were occluded by self as these are the ones that will be reflecting light back at us. Two ways of doing this – the SH lighting paper uses a subdivided icosahedron of "buckets" where each bucket contains the rays that exit through it's triangle. Each bucket is marked with a bit saying whether any rays in it have hit self. My program went for the simpler per-lighting-point STL `vector<bool>` where a set bit indicates a self hit. Much less elegant, wasteful of huge chunks of memory but surprisingly quick. But, hey, what is cheap RAM for anyway if you can't waste it to speed up an offline preprocessor? Note that all reads and writes of the STL container are done in order, so the container could be optimized for uni-directional iteration.

```
void self_transfer_sh()
{
  const double area = 4.0*PI;
  double *sh_buffer[n_bounces+1];     // list of light bounce buffers.

  // allocate and clear buffers for self transferred light
  sh_buffer[0] = sh_coeff;    // already calculated from direct lighting
  for(int i=1; i<=n_bounces; ++i) {
    sh_buffer[i] = new double[n_lighting * 3 * n_coeff];
    memset(sh_buffer[i], 0, n_lighting*3*n_coeff*sizeof(double));
  }

  // for each bounce of light
  for(int bounce=1; bounce<=n_bounces; ++bounce) {
    // loop through all lighting points redistributing self light
```

```
for(int i=0; i<n_lighting; ++i) {
  // find rays that hit self
  bitvector::iterator j;
  int n = 0;
  double u = 0.0, v = 0.0, w = 0.0;
  Face *fptr = 0;
  double sh[3*n_coeff];
  // get the surface albedo of the lighting point.
  double albedo_red   = mlist[plist[i].material].kd.x / PI;
  double albedo_green = mlist[plist[i].material].kd.y / PI;
  double albedo_blue  = mlist[plist[i].material].kd.z / PI;
  // loop through boolean vector looking for a ray that hits self…
  for(j=hit_self[i].begin(); j!=hit_self[i].end(); ++n,++j) {
    if(*j) {
      // calc H cosine term about surface normal
      float Hs = DotProduct(sample[n].vec, plist[i].norm);
      // if ray inside hemisphere, continue processing.
      if(Hs > 0.0) {
        // trace ray to find tri and (u,v,w) barycentric coords of hit
        u = v = w = 0.0;
        fptr = 0;
        bool ret = raytrace_closest_triangle(plist[i].pos,
                                             sample[n].vec,
                                             face_ptr, u, v);
        // if (surprise, surprise) the ray hits something...
        if(ret) {
          // lerp vertex SH vector to get SH at hit point
          w = 1.0 - (u+v);
          double *ptr0 = sh_buffer[bounce-1] +
                         face_ptr->vert[0]*3*n_coeff;
          double *ptr1 = sh_buffer[bounce-1] +
                         face_ptr->vert[1]*3*n_coeff;
          double *ptr2 = sh_buffer[bounce-1] +
                         face_ptr->vert[2]*3*n_coeff;
          for(int k=0; k<3*n_coeff; ++k) {
              sh[k] = u*(*ptr0++) + v*(*ptr1++) + w*(*ptr2++);
          }
          // sum reflected SH light for this vertex
          for(k=0; k<n_coeff; ++k) {
            sh_buffer[bounce][i*3*n_coeff + k+0*n_coeff] +=
                             albedo_red   * Hs * sh[k+0*n_coeff];
            sh_buffer[bounce][i*3*n_coeff + k+1*n_coeff] +=
                             albedo_green * Hs * sh[k+1*n_coeff];
            sh_buffer[bounce][i*3*n_coeff + k+2*n_coeff] +=
                             albedo_blue  * Hs * sh[k+2*n_coeff];
          }
        } // ray test
      } // hemisphere test
    } // hit self bit is true
  } // loop for bool vector
} // each lighting point
// divide through by n_samples
```

```
    const double factor = area / n_samples;
    double *ptr = sh_buffer[bounce];
    for(int j=0; j<n_lighting * 3 * n_coeff; ++j)
      *ptr++ *= factor;
    }
  } // loop over all bounces

  // sum all bounces of self transferred light back into sh_coeff
  for(i=1; i<=n_bounces; ++i) {
    double *ptra = sh_buffer[0];
    double *ptrb = sh_buffer[i];
    for(int j=0; j<n_lighting * 3 * n_coeff; ++j)
      *ptra++ += *ptrb++;

  // deallocate SH buffers
  for(i=1; i<=n_bounces; ++i) {
    delete[] sh_buffer[i];
  }
  return;
}
```

Global illumination and radiosity programmers will recognize this is a *probabilistic gathering solution*, where each point on the model blindly searches for neighbors that can see it and drags in light from the outside world, with the effect that the model is slowly lit by completely solving one lighting point at a time. It is quite possible to implement a *non-probabilistic* version that has pre-calculated which vertices are visible to each other and loops over only them. Another option is to implement a *shooting solution* where each surface distributes it's energy outwards to every surface it can see so that the lights slowly come up, converging on the solution all at once. (It's is good to keep a sorted list or heap of lighting points ordered by intensity so that the lights come up as fast as possible.) Both techniques have their place, especially when we introduce emissive surfaces into the preprocessor. The drawback of the non-probabilistic methods are that we have to calculate SH coefficients for each ray as we generate them instead of blasting through a list of precalculated vectors and SH coefficients. It's a balancing act between accuracy and efficiency but compared to the cost of ray-model intersections, calculating a few SH coefficients costs peanuts. (Importance sampling is another area to play with, especially when we start using glossy BRDFs).

Our new, more picky raytracer will have more issues with non manifold objects and coplanar polygons than the self shadowing version as it has to find barycentric coordinates for each hit. If you find that vertices inside concavities are returning bad values, one quick fix up you can do is to offset the origin of the ray a small epsilon distance along the geometric mean normal of the vertex

(add all normals of polygons adjacent to the vertex and renormalize). This will usually move the ray origin outside of the model and give better looking interreflection deep into concavities at the cost of a slightly increased number of false positive self hits.

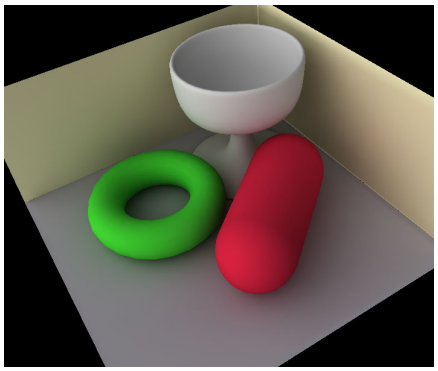


Figure 15. A rendering using 5th order diffuse SH transfer functions with self interreflection. Note the soft shadows and color bleeding onto the white cup.

Using SH lighting over time will start to affect how you look at 3D models. You will start looking for flat shaded models without shadows baked into the texture as the SH lighting will handle all this for you. You will start to recognize how often artists leave out subtle concavities in real-time models as normal lighting just does not show them up. You will start asking for models that break up large flat areas into grids of triangles so that these areas get more lighting samples across their span, helping to capture shadows more effectively. Over time you will develop an eye for models that will look good under SH lighting and see how to adapt other models to put vertices in the interesting lighting points. Putting the tools of SH lighting into the hands of your artists is the best way to update your art path for SH lighting, as low order SH preprocessing can be done at near interactive rates.

Rendering SH Diffuse Surfaces

Now we have a set of SH coefficients for each vertex, how do we build a renderer using current graphics hardware that will give us real-time frame rates? Going back to the properties of SH functions, the basic calculation for SH lighting is the dot product between an SH projected light source and the SH transfer function:

$$\int_S \tilde{L}(s)\tilde{t}(s)ds = \sum_{i=0}^{n^2} L_i\, t_i$$
$$= L_0 t_0 + L_1 t_1 + L_2 t_2 + L_3 t_3 \ldots$$

This calculation will give us a single channel light intensity for a vertex, and we approximate the complete solution over an object by filling in the gaps between vertices using Gouraud shading. Remember that all SH calculations happen in object space so if

you want to reorient the object in world space or rotate a lighting function, you will need to rotate the light into object space first.

At this point we get some choices to make. If we assume that the illumination is a white light source (the same for red, green and blue), the calculation per vertex is:

```
for(int j=0; j<n_coeff; ++j) {
  vertex[i].red   += light[j] * vertex[i].sh_red[j];
  vertex[i].green += light[j] * vertex[i].sh_green[j];
  vertex[i].blue  += light[j] * vertex[i].sh_blue[j];
}
```

A colored light source will have a different SH function for each color channel, making calculation:

```
for(int j=0; j<n_coeff; ++j) {
  vertex[i].red   += light_red[j]  * vertex[i].sh_red[j];
  vertex[i].green += light_green[j]* vertex[i].sh_green[j];
  vertex[i].blue  += light_blue[j] * vertex[i].sh_blue[j];
}
```

Note that nowhere in either calculation is the surface normal mentioned – it's already implicitly encoded in the SH coefficients as is the ambient term, so we get diffuse shading with occluded ambient built in.

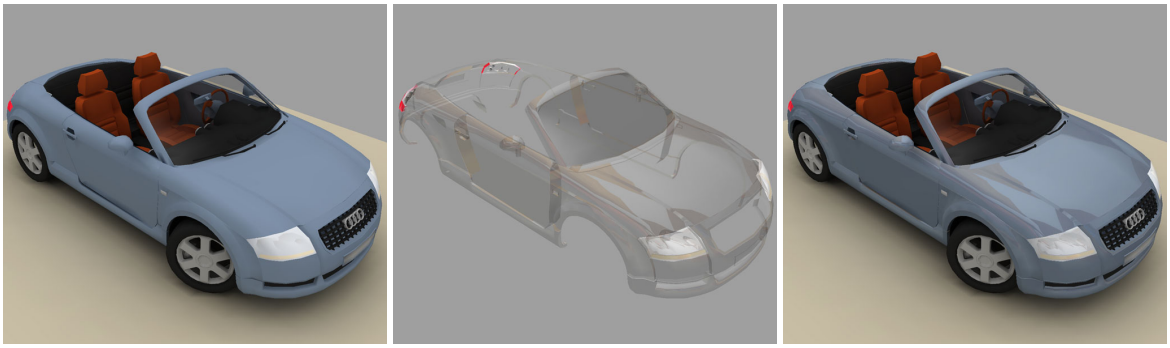
One optimization is to note that if the model has not too much emphasis on hard shadows (like a human face), the cosine term is well approximated by just the first two orders of SH functions, giving us a 4-coefficient drop in replacement for the normal diffuse shading term that gives us soft self-shadowing for no extra multiply-adds per vertex and no extra storage. This 4-coefficient system also translates well into 4-way SIMD operations, using exactly the same code as a 3 parallel light source plus ambient matrix.

Another option is to use a single channel white light source, encode just self shadowing and ignore self transfer. The resulting lighting calculation requires only one transfer function and one colour per vertex and can be re-colored on the fly:

```
for(int j=0; j<n_coeff; ++j) {
  vertex[i].red   += k_red   * light[j] * vertex[i].sh[j];
  vertex[i].green += k_green * light[j] * vertex[i].sh[j];
  vertex[i].blue  += k_blue  * light[j] * vertex[i].sh[j];
}
```

We can mix and match SH lighting and normal lighting because

light sums linearly, so we can use SH lighting for the ambient and diffuse part of a shader and add a specular term over the top. This is exactly what we did for the image of the car at the beginning of this document, first rendering the car body diffusely with SH lighting and alpha blending a pre-filtered specular environment map over the top.



So far we have been using infinite light sources that are the same all over an object, but there is a way to fake a local light source illuminating the model. This only works if we have only self-shadowing and no self-transfer as it breaks the assumption of low lighting variance across the model. To do local lighting we construct a small number of samples (e.g. 6 or 8) of the local lighting environment from a few well spaced precalculated points on the surface of the model, calculating the lighting function as if the model was absent. If there are emitters close by each of these spherical samples will see the local light source at slightly different positions. We reconstruct correct local lighting by calculating a weighted sum of the lighting samples for each vertex, weighting by the distance from the vertex to the light sample point. The example in Sloan, Kautz and Snyder's SH paper used the weighting scheme:

$$w(i,j)=\left(\frac{1}{dist(i,j)}\right)^{n}$$

between vertex i and lighting point j where n = 10 in the example image of Max Planck to emphasize the effect of local lighting.

Other researchers have found a quicker, looser method. Take the N lighting sample positions and construct a very low poly triangulated polyhedron. Next break the model into groups of vertices based on which polygon they are closest to. To precalculate the distribution of light to the vertices in a group we project the vertex onto the lighting triangle and calculate the 2D barycentric coordinate of that projected point. We then use these weights to lerp the three light sources for that vertex. This

technique has the added advantage that it can be calculated in hardware.

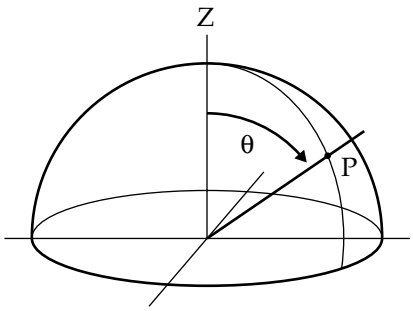
We have only scratched the surface of techniques for reconstructing SH lighting so have a go and see if you can come up with some better and more flexible effects. The current holy grail is to extend SH lighting from being a static model technique to one that works on animated models. Blending SH vectors the same way we use blending matrices for joints works well for the cosine term, but the shadow function V is just too random and unpredictable to represent as the linear combination of a few snapshots. More work on this problem is needed.

Creating Light Sources

Single channel CIE Sky Models available from http://www.softcom.net/users/daylight/thesis.pdf

In 1955, the CIE, the international standards body tasked with defining the science of illumination and colour, produced a paper defining three standard reference light sources by which architectural designs should be judged. The standard is available from the CIE for around €10 and is copyright, so I can only give you the single-channel versions stolen from the web.

Starting with the CIE Overcast model which gives the lighting function for an evenly overcast day:



$$L_\beta = L_z \frac{1 + 2\sin\beta}{3}$$

where

L_Z = sky luminance at the zenith (top of the sky) in kilo candela per meter^2.
β = angle between zenith and P, in radians.

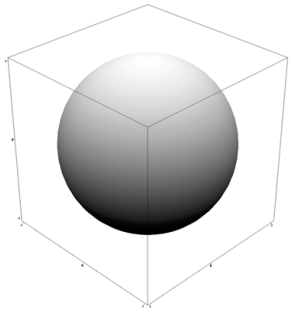
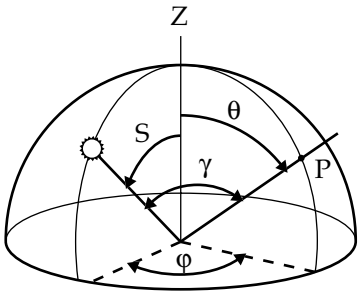


Figure 16. CIE Overcast Sky illuminance model.

The CIE Clear Sky model is a bit more complex as we have to take into account the position of the sun and it's different scattering effects as it nears the horizon.



$$L_{\theta,\varphi} = L_Z \frac{\left(0.91 + 10e^{-3\gamma} + 0.45\cos^2\gamma\right)\left(1 - e^{\frac{-0.32}{\cos\theta}}\right)}{\left(0.91 + 10e^{-3S} + 0.45\cos^2 S\right)\left(1 - e^{-0.32}\right)}$$

where

$L_{\theta,\varphi}$ = luminance at point P at (θ,φ) on the sphere, in kcd/m².

L_Z = sky luminance at the zenith, in kcd/m².

θ = angle between zenith and P, radians

α = azimuth angle (on the ground) between sun and P, radians.

S = angle between sun and zenith, radians.

γ = planar angle between the sun and P, radians

Quite an impressive equation. The resulting light source is very realistic.

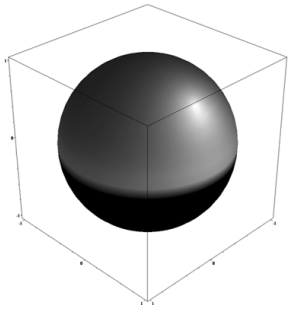


Figure 17. CIE Clear Sky illuminance model.

The CIE Partly Cloudy model is a halfway house between the two and a little simpler than the clear sky model.

$$L_{\theta,\varphi} = L_Z \frac{\left(0.526 + 5e^{-1.5\gamma}\right)\left(1 - e^{\frac{-0.8}{\cos\gamma}}\right)}{\left(0.526 + 5e^{-1.5S}\right)\left(1 - e^{-0.8}\right)}$$

where the definitions for the Clear Sky model still hold.

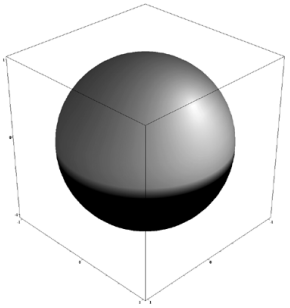


Figure 18. CIE Partly Cloudy illuminance model.

Another type of illumination model we can use are the *high dynamic range light probes* made popular by Paul Debevec. HDR images are encoded bitmaps of floating point RGB values captured from the real world using a special process of camera calibration and multiple exposures that encode the full range of light energy in a scene. Using the freely available program HDRShop we can generate angle map projections of HDR scenes and save them out as arrays of float RGB pixels.

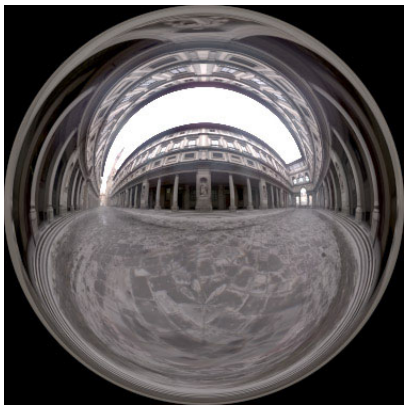


Figure 19. An example HDR light probe as an N×N bitmap of RGB float values using an angle map projection.

The angle map projection is simple to access using either spherical or Cartesian coordinates, and we can plug this straight into an SH projection function giving us SH coefficients that approximate the lighting function.

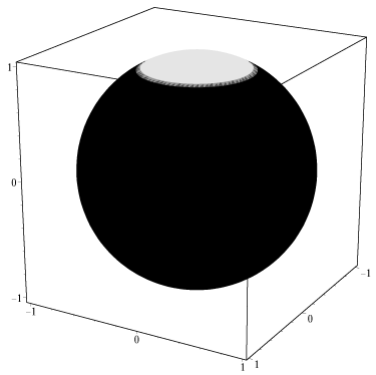
```
typedef Vector3d (*SH_vector_fn_rgb)(float dx, float dy, float dz);

Vector3d hdr_lightsource(Vector3d *hdr_image, int image_size,
                         float dx,float dy, float dz)
{
  // assume angle map projection
  const float one_over_pi = 1.0f / PI;
  float invl = 1.0f / sqrtf(dx*dx+dy*dy);
  float r = one_over_pi * acosf(dz) * invl;
  float u = dx * r;           // -1..1
  float v = dy * r;           // -1..1
  // map to pixel coordinates
  int x = int(u * image_size + image_size) >> 1;
  int y = int(v * image_size + image_size) >> 1;
  // return the float RGB value at (x,y)
  return hdr_image[y*image_size + x];
}

void SH_project_vector_function_rgb( SH_vector_fn_rgb fn,
                                     int n_bands,
                                     int sqrt_n_samples,
                                     const SHSample sh_samples[],
                                     Vector3d result[] )
{
  int n_coeff = n_bands*n_bands;
  int n_samples = sqrt_n_samples*sqrt_n_samples;
  const double area = 4.0*PI;

  // for each sample
  for(int i=0; i<n_samples; ++i) {
    Vector3d color = fn(sh_samples[i].vec.x,
                        sh_samples[i].vec.y,
                        sh_samples[i].vec.z);
```

```
    for(int n=0; n<n_coeff; ++n) {
      result[n] += color * float(sh_samples[i].coeff[n]);
    }
  }
  // divide the result by number of samples
  double factor = area / n_samples;
  for(i=0; i<n_coeff; ++i) {
    result[i] = result[i] * factor;   // NOTE: vector-scalar multiply
  }
}
```

The real problem with HDR images once you start playing with them is that there is no standard normal intensity for an HDR image. There is no clear "brightest" setting to scale them to, so you have to start building renderers that have exposure settings built in from day one. It looks like exposure settings are part of the future of renderers.

The final form of light source we can generate are totally synthetic light sources, generated directly as SH coefficients. By symbolically integrating a spherical function f(theta,phi) that returns 1 if theta is less than a threshold *t*, we can create a circular light source around the *z*-axis:

$$f(t,\theta,\varphi)=\begin{cases}1 & (t-\theta)>0\\ 0 & otherwise\end{cases}$$



We can then use symbolic integration in Mathematica or Maple to generate an analytical solution for the SH projection of the light:

$$c_i=\int_{\varphi=0}^{2\pi}\int_{\theta=0}^{\pi} f(t,\theta,\varphi)y_i(\theta,\varphi)\sin\theta\,d\theta\,d\varphi$$

This gives us a direct algorithm for generating a light source. All we need do next is rotate this light source to the correct

orientation using our SH rotation code, and sum it with other light sources.

```
inline double is_positive(double x) { return x>0 ? 1.0 : 0.0; }

void synth_light(double cutoff, double sh[])
{
  // clear all values to 0.0
  memset(sh, 0, 16*sizeof(double));
  // symbolic integration automatically generated by Maple.
  double t3 = is_positive(-cutoff + PI);
  double t5 = cos(cutoff);
  double t6 = t3*t5;
  double t9 = is_positive(-cutoff);
  double t11 = t9*t5;
  double t14 = sin(cutoff);
  double t15 = t14*t14;
  double t16 = t3*t15;
  double t18 = t9*t15;
  double t21 = t5*t5;
  double t22 = t21*t5;
  double t31 = t21*t21;
  sh[0] = 3.544907702 - 1.772453851*t3 - 1.772453851*t6 -
          1.772453851*t9 + 1.772453851*t11;
  sh[2] = 1.534990062*t16 - 1.534990062*t18;
  sh[6] = -1.98166365*t3*t22 + 1.98166365*t6 + 1.98166365*t9*t22 -
           1.98166365*t11;
  sh[12] = 2.930920062*t3 - 2.930920062*t3*t31 - 3.517104075*t16 -
           2.930920062*t9 + 2.930920062*t9*t31 + 3.517104075*t18;
}
```

Alex Evans of Lionhead has suggested using a similar technique for creating *proxy shadows* on SH light sources. Firstly, each object gets it's own copy of the common lighting function. Next, instead of generating a "cap" of light we generate the inverse function – everything except the cap. We next find out where external objects are in object space and calculate a rotation from the that direction to the *z*-axis. We then rotate the lighting function with that rotation and "multiply" it with the SH light source to subtract light in that direction.

There are two ways to implement this multiplication, one by circular convolution and one by generating an analytical *transfer matrix* by symbolic integration. Either way we must rotate the light so that the shadow direction is along the *z*-axis, do the convolution and rotate the light back to object space so we can carry on with lighting as normal. That's two SH rotations per shadow per object.

Advanced SH Techniques

SH lighting of glossy surfaces will be covered in an extended version of this paper, available soon from:

http://research.scea.com/

hopefully incorporating notes and example code on:

- Untangling the mix of coordinate spaces.
- Using graphics hardware to calculate SH transfer functions.
- Extending the preprocessor for models with mixed materials.
- Extending the preprocessor for static, emissive surfaces.
- Adding reflected caustics to the preprocessor.
- Diffuse volumetric lighting for relighting clouds in real time.
- Projected transfer for sampling the shadow around an object.
- Real Time Translucency.

Conclusion

Hopefully we have shown how SH lighting can be used to produce extremely realistic images on both static and dynamic models using just a little extra processing power. We have shown how to preprocess your static models with interreflection, how to generate transfer functions for diffuse shadowing and how it affects the way you design your models. SH lighting can be used as a drop-in replacement for the diffuse and ambient terms of a normal renderer for static objects, and we have covered a range of options for updating global and local lighting in real time.

I hope you will find a use for SH lighting for your next project so that we can all take game graphics to the next stage of realism. I am looking forward to seeing how people use SH functions for unexpected tasks that require encoding and manipulating spherical functions. Remember to write up and share your ideas!

Acknowledgements

Dominic Mallinson and Attila Vass of SCEA R&D for their support and giving me the freedom to pursue this area of research as far as I have. Gabor Nagy for working so hard to add SH lighting to his 3D editing system Equinox (downloadable at `http://www.equinox3d.com/`) so that we could both play with it interactively. Alex Evans of Lionhead for his rotation matrices, the proxy shadow and barycentric local lighting ideas and general enthusiasm and willingness to share. Peter-Pike Sloan, Jan Kautz and John Snyder for their generous help and putting up with my endless questions. Professor Klaus Ruedenberg, Dr Joe Ivanic and

Professor Cheol Ho Choi for generously sharing their code and support in mastering the alien world of SH Rotations. Charles Poynton for the assurance that typography and design is important, even for technical documents. My wife Christiane, for keeping my head in the real world.

Appendix 1: Fast SH Rotations

The 1996 paper "Rotation Matrices for Real Spherical Harmonics" by Ivanic et al (with errata factored in) contains the following recurrence relations for generating SH rotation matrices. Firstly we need to permute the normal 3×3 rotation matrix into a new order and rename it R_{mn}:

$$R_{mn} = \begin{bmatrix} R_{YY} & R_{YZ} & R_{YX} \\ R_{ZY} & R_{ZZ} & R_{ZX} \\ R_{XY} & R_{XZ} & R_{XX} \end{bmatrix} = \begin{bmatrix} R_{-1,-1} & R_{-1,0} & R_{-1,1} \\ R_{0,-1} & R_{0,0} & R_{0,1} \\ R_{1,-1} & R_{1,0} & R_{1,1} \end{bmatrix}$$

The SH rotation matrix M is defined for band l as

$$M^l_{mn} = u^l_{mn} U^l_{mn} + v^l_{mn} V^l_{mn} + w^l_{mn} W^l_{mn}$$

where integer arguments m and n vary from $-l$ to l to fill in the entries of the matrix. The following tables define expressions for coefficients u, v, w as well as functions U, V, W and P.

	$m = 0$	$m > 0$	$m < 0$
U^l_{mn}	${}_0P^l_{0n}$	${}_0P^l_{mn}$	${}_0P^l_{mn}$
V^l_{mn}	${}_1P^l_{1n} + {}_{-1}P^l_{-1n}$	${}_1P^l_{m-1,n}\sqrt{1+\delta_{m1}} - {}_{-1}P^l_{-m+1,n}(1-\delta_{m1})$	${}_1P^l_{m+1,n}(1+\delta_{m,-1}) + {}_{-1}P^l_{-m-1,n}\sqrt{1-\delta_{m,-1}}$
W^l_{mn}		${}_1P^l_{m+1,n} + {}_{-1}P^l_{-m-1,n}$	${}_1P^l_{m-1,n} - {}_{-1}P^l_{-m+1,n}$

	$\lvert n\rvert < l$	$\lvert n\rvert = l$
u^l_{mn}	$\sqrt{\frac{(l+m)(l-m)}{(l+n)(l-n)}}$	$\sqrt{\frac{(l+m)(l-m)}{(2l)(2l-1)}}$
v^l_{mn}	$\frac{1}{2}\sqrt{\frac{(1+\delta_{m0})(l+\lvert m\rvert-1)(l+\lvert m\rvert)}{(l+n)(l-n)}}(1-2\delta_{m0})$	$\frac{1}{2}\sqrt{\frac{(1+\delta_{m0})(l+\lvert m\rvert-1)(l+\lvert m\rvert)}{(2l)(2l-1)}}(1-2\delta_{m0})$
w^l_{mn}	$-\frac{1}{2}\sqrt{\frac{(l-\lvert m\rvert-1)(l-\lvert m\rvert)}{(l+n)(l-n)}}(1-\delta_{m0})$	$-\frac{1}{2}\sqrt{\frac{(l+\lvert m\rvert-1)(l+\lvert m\rvert)}{(2l)(2l-1)}}(1-\delta_{m0})$

	$\lvert b\rvert < l$	$b = l$	$b = -l$
${}_iP_{ab}^{l}$	$R_{i,0}M_{ab}^{l-1}$	$R_{i,1}M_{a,l-1}^{l-1} - R_{i,-1}M_{a,-l+1}^{l-1}$	$R_{i,1}M_{a,-l+1}^{l-1} + R_{i,-1}M_{a,l-1}^{l-1}$

(Note: If you get hold of the original paper, you will see that I have renamed variables m' to n and R^l to M^l for clarity. The original authors admit that the printing was less than clear.)