Efficient Spherical Harmonic Evaluation

Peter-Pike Sloan
NVIDIA Corporation

Abstract

The real spherical harmonics have been used extensively in computer graphics, but the conventional representation is in terms of spherical coordinates and involves expensive trigonometric functions. While the polynomial form is often listed for low orders, directly evaluating the basis functions independently is inefficient. This paper will describe in detail how recurrence relations can be used to generate pre-factored evaluation code that is smaller, more efficient, and presents a performance comparison of several alternative techniques to evaluate the spherical harmonics.

1. Introduction

While spherical harmonics represent complex functions on the sphere, the real spherical harmonics (RSH) have been used extensively in graphics [Ramamoorthi and Hanrahan 2001; Sloan et al. 2002] and games [Chen 2008]. They are the spherical analog to the Fourier basis on the unit circle and, conceptually, are just a representation of spherical functions. While they have several important properties and efficient algorithms exist for convolution, computing various integrals, computing products of spherical functions and rotation [Sloan 2008], this paper focuses on how to evaluate the basis for a given direction on the unit sphere. This is one of the most common operations on SH, for everything from evaluating irradiance environment maps [Ramamoorthi and Hanrahan 2001] to projection of analytic light sources. The most common mathematical form in the literature is

$$y_l^m = \begin{cases} \sqrt{2}K_l^m \cos(m\phi)P_l^m(\cos\theta), & m > 0, \\ \sqrt{2}K_l^m \sin(|m|\phi)P_l^{|m|}(\cos\theta), & m < 0, \\ K_l^m P_l^m(\cos\theta), & m = 0. \end{cases} \tag{1}$$

where P_l^m are the associated Legendre polynomials and K_l^m are the normalization constants

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

They are indexed by band l and function in a band m, where l is a non-negative integer, and m is an integer in $[-l,l]$ in band l. An order O SH consists of all the bands between 0 and $O-1$ which has O^2 basis functions. This form is convenient for symbolic computations and evaluating analytic integrals, but it is expensive to evaluate at run-time. SH can also be represented as polynomials of a point on the unit sphere,[1] but they become quite complex, particularly at higher orders. The basis is orthogonal, closed under rotations, and, using a small number of bands, can accurately represent smooth functions.

1.1. Related Work

In a previous paper, Sloan [2008] gave a cursory description of some recurrence relations and a vague mention of how they might be used, but no explicit algorithm or code was given. Snyder [2006] described nicely how to compute the products of real SH, and this paper, in the same spirit, attempts to do so for evaluation. Many equations for recurrence relations/properties of spherical harmonics can be found in a text book [Varshalovich et al. 1988], but be forewarned, that these are for the complex spherical harmonics, so they have to be tweaked to work for the real spherical harmonics.

Another paper [Green 2003] has code for evaluating the RSH in spherical coordinates, but it is 2–3 orders of magnitude slower than the techniques presented in this paper. There also was a detailed code example of how to efficiently evaluate order 3 SH [Sloan 2003] on GPU shaders; this paper, however, is focused on higher orders. For vectorized GPU's, we would still recommend using that technique, but on GPU's that have scalar lanes, it is not necessary.

Code generators for specific algorithms have existed for a long time; FFTW [Frigo and Johnson 2005] is a good example. The code generator here is fairly simple and just generates scalar or SSE C code that can be compiled and linked to a given program.

2. SH Code Generator

To generate efficient SH code, we will exploit some common recurrence relations for the associated Legendre polynomials, and a clever trick from a former colleague at Microsoft, John Snyder, that makes them applicable to Cartesian coordinates, instead of the usual spherical ones. The equations for the associated Legendre polynomials are

$$P_l^m(x) = (1-x^2)^{m/2} \frac{d^m}{dx^m} (P_l(x)) \tag{2}$$

[1] Band l represents the polynomial basis of degree l when restricted to the unit sphere. Above the linear functions $l = 1$, this is a smaller basis than when considered over all $\mathbb{R}^3$.

where P is a Legendre polynomial. When $x = \cos\theta$ it simplifies to

$$P_l^m(\cos\theta) = (\sin\theta)^m \frac{d^m}{dx^m}\left(P_l(\cos\theta)\right). \tag{3}$$

In Cartesian coordinates $z = \cos\theta$, so if we divide P_l^m by $(\sin\theta)^m$, we are left with an expression that is simply a polynomial in z. Looking at Equation (1), one can simply absorb this $(\sin\theta)^m$ term into the ϕ terms, which leaves that part as a polynomial in x and y when expanded from $m\phi$ to ϕ using the trigonometric addition theorem.[2]

To evaluate Equation 1 in terms of Cartesian coordinates, we use the following recurrence relations:

$$P_m^m = (1-2m)P_{m-1}^{m-1}, \tag{4a}$$

$$P_{m+1}^m = (2m+1)zP_m^m, \tag{4b}$$

$$P_l^m = \frac{(2l-1)zP_{l-1}^m - (l+m-1)P_{l-2}^m}{l-m}, \tag{4c}$$

$$P_{m+2}^m = \frac{(2m+3)(2m+1)P_m^m z^2 - (2m+1)P_m^m}{2}, \tag{4d}$$

$$P_{m+3}^m = \frac{zP_m^m((2m+5)(2m+3)(2m+1)z^2 - 3(4m^2+8m+3))}{6}. \tag{4e}$$

Equation (4a) has the $(\sin\theta)^m$ term factored out, and all of the other recurrence relations build off this one. Most of the work is done using Equation (4c). Rules (4d) and (4e) are constructed by simply plugging in (4a) and (4b) into Equation (4c) and factoring constant expressions. Using these relations, you first compute the Y_l^0 functions, then iterate through the m terms for each relevant band from smallest to largest, building up the x, y terms using the trigonometric addition theorem.

Even when precomputing P_m^m and K_l^m in tables for a fixed order, evaluating these recurrence relations on the fly turns out to be slower than just using explicit polynomials for several reasons:

1. Products of constants involving K_l^m have to be multiplied through the recurrence relations.

2. Terms in the relations that are simple functions of l and m have to be computed.

3. There is flow control overhead for iterating through bands.

As an alternative, we wrote a program that, for a given order, evaluates the recurrence relations and generates code that propagates constant terms aggressively. Precision is a big problem, since naive evaluation of K_l^m will have precision problems. The code is given in Listing 1, but, for higher orders, you should either code it

[2]This theorem can be used to express $\cos(m\phi)$ or $\sin(m\phi)$ as a sum of terms where each one has exactly m of $\sin(\phi)$ or $\cos(\phi)$, which, when paired with $\sin(\theta)^m$, leaves you with a polynomial in x and y.

```
/* SH Normalization function:
 K(l,m) = sqrt((2*l + 1)(l - |m|)!/(4 Pi (l + |m|)!)
 The factorials mostly cancel out, you don't want overflow.
 To really be robust, you need to include this with the
  evaluation of Plm -- particularly for large m */
double K(const unsigned int l, const int m) {
    const unsigned int cAM = abs(m);
    double uVal = 1;// must be double

    for (unsigned int k = l + cAM; k > (l - cAM); k--)  uVal *= k;

    return sqrt( (2.0 * l + 1.0) / ( 4 * PI * uVal ) );
}
```

Listing 1. K_l^m code.

up in something like Mathematica or use an arbitrary precision arithmetic library—as l gets large, P_m^m becomes a huge number and K_l^m becomes very small.

There are two other optimizations in the code generator that can lead to increased performance. On some architectures, the dependencies in the instruction sequence cause issues, but if you interleave a pair of SH evaluations by construction every other instruction will be completely independent, leading to higher performance. Finally, you can generate vectorized code that evaluates the SIMD units width of SH evaluations in parallel. This leads to a significant performance win, depending on the architecture. On the Nintendo Wii I had to employ both of these modifications to get the 2X speedup one would expect from moving to the 2-wide vector instructions in that architecture; there was no speedup just vectorizing a pair of evaluations together. When generating vector code, the final results are interleaved—they either need to be de-interleaved by code, or code that uses them downstream has to understand this layout. A concrete example is using this to fill an environment map; this has been successfully used on several Wii titles [Ownby et al. 2010].

3. Results

We benchmarked various SH evaluation codes at orders up to 10. The techniques, illustrated from top to bottom in Table 1 are using recurrence relations in spherical coordinates (GRITY) [Green 2003], recurrence relations in Cartesian coordinates (Recur),[3] explicit polynomials for each basis function (Poly) [Sloan 2008], the output of the code generator for scalar code (RecCG) and, finally, vectorized output (RecSSE). The benchmark uses the same set of 160,000 random directions[4] and takes a

[3]Both these techniques precomputed a table of K_l^m values and P_m^m values. For the Gritty code 25% less time was needed.

[4]They are represented in Cartesian and spherical coordinates as a precomputation. Evaluation code does not have any data-dependent branches, so the directions are unimportant.

Algorithm	3	4	6	8	10
GRITY[3]	85.89	126.84	300.96	625.41	1158.86
Recur[3]	22.82	38.19	86.24	172.36	295.83
Poly	4.29	9.76	26.77	52.74	85.19
RecCG	4.06	7.59	21.14	39.64	64.17
RecSSE	0.95	2.01	5.35	9.65	15.82

Table 1. Comparison of SH evaluation algorithms at various orders. RecCG and RecSSE are the ones from this paper, timings are in nanoseconds per SH evaluation.

```
void SHNewEval3(const float fX, const float fY, const float fZ,
                float* __restrict pSH) {
    float fC0, fC1, fS0, fS1, fTmpA, fTmpB, fTmpC;
    float fZ2 = fZ * fZ;

    pSH[0] = 0.2820947917738781f;
    pSH[2] = 0.4886025119029199f * fZ;
    pSH[6] = 0.9461746957575601f * fZ2 + -0.3153915652525201f;
    fC0 = fX;
    fS0 = fY;

    fTmpA = -0.48860251190292f;
    pSH[3] = fTmpA * fC0;
    pSH[1] = fTmpA * fS0;
    fTmpB = -1.092548430592079f * fZ;
    pSH[7] = fTmpB * fC0;
    pSH[5] = fTmpB * fS0;
    fC1 = fX*fC0 - fY*fS0;
    fS1 = fX*fS0 + fY*fC0;

    fTmpC = 0.5462742152960395f;
    pSH[8] = fTmpC * fC1;
    pSH[4] = fTmpC * fS1;
}
```

Listing 2. Order 3 SH evaluation code.

minimum of 100 evaluation runs. The output of the code generator is always faster than the raw polynomials and generates less code compared to the raw polynomials.[5] Timings are on a 3.5 GHz Intel i7 processor.

Listing 2 is example output for the quadratic SH; see the supplemental documents for higher-order/vectorized code examples. The order of the coefficients uses the standard mapping to a single index: $i = l(l+1)+m$.

[5] At 10th order, the code generator functions use 2898 bytes, while the polynomials use 4339 bytes.

Acknowledgements

I am particularly indebted to John Snyder; the approach taken in this paper was his idea, in particular, the clever absorption of the $\sin(\theta)^m$-term from the associated Legendre Polynomials into the ϕ-dependent that enabled a fairly straightforward application of the recurrence relations expressed in θ, ϕ to the Cartesian coordinates. Morgan McGuire gave valuable feedback on an earlier draft of this paper.

References

CHEN, H. 2008. Lighting and Materials of Halo3. In *Game Developers Conference*. http://gdcvault.com/play/253/Lighting-and-Material-of-HALO. 84

FRIGO, M., AND JOHNSON, S. G. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2, 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation". 85

GREEN, R. 2003. Spherical Harmonic Lighting: The Gritty Details. In *Game Developers Conference*. http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf. 85, 87

OWNBY, J.-P., HALL, R., AND HALL, C. 2010. Rendering Techniques in Toy Story 3. In *SIGGRAPH 2010 Course: Advances in Real-Time Rendering in 3D Graphics and Games*, ACM Press, New York. 87

RAMAMOORTHI, R., AND HANRAHAN, P. 2001. An efficient representation for irradiance environment maps. In *SIGGRAPH 2001 Conference Proceedings, August 12–17, 2001, Los Angeles, CA*, ACM Press, New York, 497–500. 84

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics 21*, 3 (July), 527–536. 84

SLOAN, P.-P. 2003. Efficient evaluation of irradiance environment maps. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*, W. Engel, Ed. Wordware, Plano, Texas. 85

SLOAN, P.-P. 2008. Stupid Spherical Harmonics (SH) Tricks. In *Game Developers Conference*. http://www.gdcvault.com/play/273/Stupid-Spherical-Harmonics-(SH). 84, 85, 87

SNYDER, J. 2006. Code Generation and Factoring for Fast Evaluation of Low-order Spherical Harmonic Products and Squares. Tech. Rep. MSR-TR-2006-53, Microsoft Research. 85

VARSHALOVICH, D. A., MOSKALEV, A. N., AND KHERSONSKII, V. K. 1988. *Quantum Theory of Angular Momentum*. World Scientific, Singapore. 85

Supplemental Materials

The supplemental materials include a Visual Studio solution that calls the code and generates a file with evaluation code for orders 3 through 10. The file SHEvalCodeGen.cpp has the single relevant entry point, BuildSHEvalCode and lmax is the degree to generate. In that file,

there is a #define SSE that needs to be uncommented to generate vectorized code. There are also two files that are the output of the code generator: SHEval.cpp and SHEvalSSE.cpp.

Author Contact Information

Peter-Pike Sloan
NVIDIA Corporation
11431 Willows Road NE
Suite 200
Redmond, WA 98052
ppjsloan@gmail.com

Peter-Pike Sloan, Efficient Spherical Harmonic Evaluation, *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, 84–90, 2013
http://jcgt.org/published/0002/02/06/

Received:	2013-07-01		
Recommended:	2013-08-09	Corresponding Editor:	Cindy Grimm
Published:	2013-09-08	Editor-in-Chief:	Morgan McGuire