Efficient and Accurate Spherical Kernel integrals using Isotropic Decomposition

Supplemental Material

1 Projection onto the zonal harmonics basis

Spherical harmonics of degree-L form a closed space under the set of 3D rotations. That is, for any spherical harmonic y_l^m and for any given rotation $\mathbf{R}$, there exist coefficients $W_l^{m,m'}(\mathbf{R})$ such that

$$y_l^m \circ \mathbf{R} = \sum_{-l \leq m' \leq l} W_l^{m,m'}(\mathbf{R}) y_l^{m'} \tag{1}$$

These $W_l^{m,m'}$ are called the Wigner coefficients, and their values depend exclusively on the rotation $\mathbf{R}$.

Applying Equation **??** to each zonal harmonic y_l^0, we can express each rotated zonal harmonic z_l^m as a finite sum of at most $2l+1$ band-l spherical harmonics, with a set of rotations $\mathbf{R}_m$:

$$z_l^m(\theta, \varphi) = \sum_{-l \leq m' \leq l} W_l^{0,m'}(\mathbf{R}_m) y_l^{m'}(\theta, \varphi)$$

The Wigner coefficient for general spherical harmonics and any rotation are quite complex. However, in our case (i.e., for a rotated zonal harmonic basis), they have a simple analytical expression thanks to the SH addition theorem:

$$W_l^{0,m'}(\mathbf{R}_m) = y_l^{m'}(\mathbf{z}_m)\,,$$

where $\mathbf{z}_m = \mathbf{R}_m \mathbf{z}$. The SH projection coefficients for a function f can therefore be expressed as a linear combination of its RZH projection coefficients (both of degree l) using a simple matrix product as:

$$C = MZ \quad \text{with} \quad M = |y_l^i(\mathbf{z}_j)|_{ij} \tag{2}$$

Conversely, given that matrix M is invertible, one can convert a spherical harmonics expansion into a RZH expasion with:

$$Z = M^{-1} C$$

Alternatively, one can interpret RZH basis as a non orthogonal basis over the sphere: projection coefficients of a function f onto z_l^m cannot be obtained by computing the inner product with each element of the basis, but by computing the inner product with each element of the *dual basis* $\overline{z}_l^m$ instead. In this case we have:

$$f = \sum_m \lambda_l^m z_l^m \quad \text{with} \quad \lambda_l^m = \int_{S^2} f(\omega)\overline{z}_l^m(\omega)d\omega \ .$$

The $\overline{z}_l^m$ are the dual functions of the RZH basis, defined as a sum of degree-l spherical harmonics given by the rows of matrix M:

$$\overline{z}_l^m = \sum_m M_{mm'} y_l^{m'}$$

2 Stable computation of spherical harmonics

Traditionally, in the computer graphics and computational physics literature, spherical harmonic basis functions are computated as the product of a normalization term N_l^m and the un-normalized spherical harmonic basis functions $\tilde{y}_l^m$. Numerically, this computation is unstable since N_l^m approaches 0 as m grows and $\tilde{y}_l^m$ can yield arbitrarily large values.

More precisely, N_l^m exceeds 32−bit floating point precision beyond $L = 30$, and 64−bit double precision beyond $L = 86$, which renders the classical formula unsuitable for GPU computations with floating point values unless we limit ourselves to low values of L.

However, the product $y_l^m = N_l^m \tilde{y}_l^m$ is always bounded between -1 and 1. As a consequence, we derive a stable recursive formula for computing normalized spherical harmonics, combining the classical formula for N_l^m and the un-normalized y_l^m:

$$
\begin{aligned}
P_0^0 &= \tfrac{1}{\sqrt{4\pi}} && (3)\\
P_l^l &= -\sin\theta\sqrt{\tfrac{(2l+1)}{2l}}P_{l-1}^{l-1} && (4)\\
P_{l+1}^l &= \cos\theta P_l^l\sqrt{2l+3} &&\\
P_l^m &= \sqrt{\tfrac{(2l+1)(2l-1)}{(l+m)(l-m)}}\cos\theta P_{l-1}^m - \sqrt{\tfrac{(l+m-1)(l-m-1)(2l+1)}{(l+m)(l-m)(2l-3)}}P_{l-2}^m && (5)
\end{aligned}
$$

and then

$$
y_m^l(\theta,\phi) = \begin{cases} \sqrt{2}P_l^m\cos(m\phi) & \text{if } m > 0\\ P_l^0 & \text{if } m = 0\\ \sqrt{2}P_l^{-m}\sin(m\phi) & \text{if } m < 0 \end{cases}
$$

With this formula, it is possible to compute spherical harmonic values on the GPU up to degree 100 using 32-bit floats without any numerical instability (for zonal harmonics, one only needs Equations **??** and **??**). We supply the corresponding code below:

```
float Ylm::normalized_sph_Ylm_eval( int _L, int _M, float cos_theta,
                                    float cos_m_phi, float sin_m_phi )
{
   static const float ONE_OVER_SQRT_4_PI = 1.0/sqrt(4.0*M_PI) ;
   static const float          SQRT_2    =     sqrt(2.0     ) ;

   /* Evaluate the associated Legendre function at cos(theta) */
   /* and compute the value of the spherical harmonic.        */

   int M = abs(_M) ;
   int L = _L ;

   float x = cos_theta ;

   float  pmm ;
   float  pll=0.0, pmmp1;
   int     i;
```

```
   float fact = (M != 0)?SQRT_2:1.0 ;

   if(_M < 0)
      fact *= sin_m_phi ;
   else
      fact *= cos_m_phi ;

   pmm = ONE_OVER_SQRT_4_PI ;

   // Go from (0,0) to (M,M)
   //
   float somx2 = sqrt( ( 1.0 - x ) * ( 1.0 + x ) );

   for(i = 1; i <= M; ++i)
      pmm *= - somx2 * sqrt( (i+i+1)/(float)(i+i) ) ;

   if( L == M )
      return pmm * fact ;

   pmmp1 = x * pmm * sqrt(M+M+3) ; // compute (M+1,M)

   if( L == M + 1 )
      return pmmp1 * fact ;

   // compute (L,M) from (L-1,M) and (L-2,M)
   //
   for( i = M + 2; i <= L; i++ )
   {
      pll = (x * sqrt( (i+i+1)*(i+i-1)*(i-M)/(float)(i+M) ) * pmmp1
               - sqrt( (i-1+M)*(i-1-M)*(i-M)/(float)(i+M)
                     * (i+i+1)/(float)(i+i-3) ) * pmm)/(i-M) ;
      pmm = pmmp1;
      pmmp1 = pll;
   }

   return pll*fact ;
}
```

3 RZH Shader (Eq. 22)

The shader below is the one we use for all our real-time captures of RZH shading with isotropic and anisotropic BRDFs in our results and video. We removed the bilinear interpolation between BRDF coefficients, for the sake of clarity. The shader uses lobe direction sharing to compute the preconvolution of the illumination from its SH coefficients and the product with the RZH coefficients of the BRDF lobe at the same time.

The variables accessed by the shader are:

Variable name	Type	Description
`BRDFDirs1DTex`	Texture	The $2L+1$ shared ZH directions
`BRDFCoefs2D_RZH`	Texture	RZH BRDF lobe projection coefficients
`getCubemapCoef(L,M)`	Function	Returns the precomputed cubemap SH coefficients for index (L,M), stored in a texture
`v3NormalWorldPS`	vec3	Object's normal at the shading point in world coordinates
`v3ViewWorld`	vec3	View direction in world coordinates

```
FVec3 shadeRZH(const int Lmax)
{
    vec3 Y_axis = normalize( cross(v3NormalWorldPS, v3ViewWorldPS) );
    vec3 X_axis = normalize( cross(Y_axis, v3NormalWorldPS) );
    mat3x3 TBN_L2G = mat3x3(X_axis, Y_axis, v3NormalWorldPS);// local to global coordinates
    mat3x3 TBN_G2L = transpose(TBN_L2G);

    // local incoming view direction.
    vec3 v3ViewLocal = (TBN_G2L * v3ViewWorldPS) ;
    float thetaViewLocal = acos( clamp(v3ViewLocal.z/ length(v3ViewLocal), 0.0f, 1.0f) );
    int brdf_lookup_index = int( floor(thetaViewLocal/M_PI_2 * nThetas_BRDF) );

    float phiViewLocal = atan( v3ViewLocal.y, v3ViewLocal.x );

    if(phiViewLocal < 0.0f) phiViewLocal += M_2tPI;
    int phiViewLocalIndex = int( floor(phiViewLocal/M_2tPI * nPhis_BRDF) ) ;
    phiViewLocalIndex = min(phiViewLocalIndex, nPhis_BRDF-2) ;

    // This is the index for lookup of \lambda_l^m (Eq.22)

    brdf_lookup_index += phiViewLocalIndex * nThetas_BRDF ;

    FVec3 res = FVec3(0.,0.,0.) ;  // Final result accumulated here

    const FType SQRT_4_PI           = sqrt(4.0*M_PI) ;
    const FType ONE_OVER_SQRT_4_PI  = 1.0/sqrt(4.0*M_PI) ;
    const FType SQRT_2              = sqrt(2.0) ;

    for(int d=-Lmax; d<=Lmax; ++d) // for each direction of the BRDF RZH decomposition
    {
        // Fetch the current RZH direction z_d in local coordinates
        // and compute R_n^T z_d, as it appears in Eq.22

        FVec3 Rnwpk = TBN_L2G * texelFetch(BRDFDirs1DTex, d+Lmax, 0).xyz;
```

```
FType sin_theta = sqrt(1. - Rnwpk.z*Rnwpk.z) ;

FType x       =  Rnwpk.z;              // cos_theta
FType sin_1   =  Rnwpk.y / sin_theta ; // sin_phi;
FType cos_1   =  Rnwpk.x / sin_theta ; // cos_phi;
FType sin_m   =  0.0 ;
FType cos_m   =  1.0 ;
FType radical =  sqrt( 1.0 - x*x );
FType Pmm     =  ONE_OVER_SQRT_4_PI ;
FVec3 lambdaLM; // BRDF RZH coefficient
FVec3 constant;

// Now loop over all SH coeficients of the envmap, and compute y_l^m(z_d),
// and accumulating as in Eq.22

for(int M = 0; M <= Lmax; M++)
{
    FType fact_times_SQRT_4PI = ((M > 0)?SQRT_2:1.0) * SQRT_4_PI ;
    if(abs(d) <= M)
    {
       lambdaLM = texelFetch(BRDFCoefs2D_RZH_Tex,
                     ivec2(lmtoi(M,d),brdf_lookup_index), 0).rgb;

       constant = Pmm * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(M)+1.) ;
       res += cos_m * getCubemapCoef(M,M) * constant;
       if(M!=0)
         res += - sin_m * getCubemapCoef(M,-M) * constant;
    }
    FType prev_P = Pmm ;
    FType curr_P = x * sqrt(FType((M*2)+3)) * Pmm ;
    if(M != Lmax && abs(d) <= M+1)
    {
       lambdaLM  = texelFetch(BRDFCoefs2D_RZH_Tex, ivec2(lmtoi(M+1,d),
                     brdf_lookup_index), 0).rgb;

       constant = curr_P * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(M+1.)+1.) ;
       res += cos_m * getCubemapCoef(M+1,M) * constant;
       if(M!=0)
          res += - sin_m * getCubemapCoef(M+1,-M) * constant;
    }
    for(int L = M+2; L<=Lmax; L++)
    {
        FVec2 c12 = coefs(L,M) ; // this computes coefs in the stable formula (Eq.5) for SH.

        FType temp = x * c12.x * curr_P - c12.y * prev_P ;
        prev_P = curr_P;
        curr_P = temp ;/// ( L - M );
        if(abs(d)<=L)
        {
           lambdaLM = texelFetch(BRDFCoefs2D_RZH_Tex,
                          ivec2(lmtoi(L,d),brdf_lookup_index), 0).rgb;

           constant = curr_P * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(L)+1.) ;
           res += cos_m * getCubemapCoef(L,M) * constant;
           if(M!=0)
              res += - sin_m * getCubemapCoef(L,-M) * constant;
        }
```

```
            }

            // This is a simplified version of the sable SH formula for M=0.

            Pmm *= - radical * sqrt( ((M*2)+3)/FType((M*2)+2) ) ;

            FType temp = cos_1 * cos_m - sin_1 * sin_m;
            sin_m = sin_1 * cos_m + cos_1 * sin_m;
            cos_m = temp;
        }
    }

    return res;
}
```

4 RZH Shader with visibility (Eq. 27 and 30)

This shader is a straightforward extension of the previous shader to handle visibility according to Equations 27 and 30 in the paper. The shader contains 3 blocks. First, coefficients $v \otimes y^0_{l''}(\mathbf{R}^T_\mathbf{n}\mathbf{z}_{m''})$ are computed for all $l'' < L$ and $-l'' \leq m'' \leq l''$. Then the RZH coefficients of the product of the BRDF and the environment map (in world coordinates) are computed using a product with the matrix containing the precomputed $\gamma^{l,m}_{l',m'}$ coefficients. Finally, the shader computes $I(\omega_0)$ by applying the same code as the RZH shading (see previous shader), replacing the BRDF coefficients with the RZH-projected product of the envmap and the BRDF rotated to the local frame, and using the visibility in place of the envmap.

Basically, the first block performs the same computation as *shadeRZH()*, but stores the computed coefficients instead of adding them up. The next block of code converts these coefficients into RZH coefficients for the BRDF times the envmap. The final block uses them (in place of the BRDF in the original shader) to integrate against the visibility, re-using the loops of *shadeRZH()* once again.

We note that no SH rotations are performed and that the SH visibility coefficients are precomputed and stored in a texture containing one line of coefficients per vertex. Since visibility coefficients are all computed in the same coordinate system, we interpolate them directly using the barycentric coordinates of the fragment obtained from the geometry shader.

Variable name	Variable type	Comment
`BRDFDirs1DTex`	Texture	The $2L+1$ shared ZH directions
`BRDFCoefs2D_RZH`	Texture	The RZH projection coefficients of the BRDF lobe
`getCubemapCoef(L,M)`	Function	Returns precomputed SH cubemap coefficients of index (L,M), stored in a texture
`v3NormalWorldPS`	vec3	The object's normal at the shading point in world coordinates
`v3ViewWorld`	vec3	The view direction in world coordinates
`getVisibilityMapCoef(L,M)`	Function	Returns precomputed SH projection coefficients of the visibility at the current vertex
`TripleProductMatrixTex`	Texture	RZH coefficients for the product of each SH basis function and the environment map, in world coordinates

```
FVec3 shadeRZHVisibility(const int Lmax)
{
   const vec3 v3NormalWorldPS = normalize(v3NormalWorld) ;
   const vec3 v3ViewWorldPS = normalize(v3ViewWorld) ;

   vec3 Y_axis = normalize( cross(v3NormalWorldPS, v3ViewWorldPS) );
   vec3 X_axis = normalize( cross(Y_axis, v3NormalWorldPS) );
   mat3x3 TBN_L2G = mat3x3(X_axis, Y_axis, v3NormalWorldPS);// local to global coordinates
   mat3x3 TBN_G2L = transpose(TBN_L2G);

   vec3 v3ViewLocal = (TBN_G2L * v3ViewWorldPS) ; // local incoming view direction.
```

```
   float thetaViewLocal = acos( clamp(v3ViewLocal.z/ length(v3ViewLocal), 0.0f, 1.0f) ); // 0 < theta < pi/2
   int thetaViewLocalIndex = int( floor(thetaViewLocal/M_PI_2 * nThetas_BRDF) );

   thetaViewLocalIndex = min(thetaViewLocalIndex, nThetas_BRDF-2) ;
   thetaViewLocalIndex += phiViewLocalIndex * nThetas_BRDF ;

// The treatment of visibility is the following:
// 1 - Get the local RZH coefficients of the BRDF;
//     Rotate them in the world CS and transform them into SH coefficients
// 2 - Multiply the coefficients by the triple product matrix, to get the RZH coefficients
//     of the BRDF times the Envmap
// 3 - Rotate these RZH coefs into the local CS
//     Use the RZH Shading procedure as before on these coefficients with the precomputed
//     visibility SH coefs.

FVec3 BRDF_global_SH_coeffs[(MAX_L+1)*(MAX_L+1)] ;
FVec3 BRDF_Envmap_RZH_product_coefs[(MAX_L+1)*(MAX_L+1)] ;

for(int L=0;L<=Lmax;++L)
   for(int M=-L;M<=L;++M)
    BRDF_global_SH_coeffs[L*(L+1)+M] = FVec3(0.0,0.0,0.0) ;

// BLOCK 1

   const int Mmax = Lmax ;

   const FType SQRT_4_PI           = sqrt(4.0*M_PI) ;
   const FType ONE_OVER_SQRT_4_PI  = 1.0/sqrt(4.0*M_PI) ;
   const FType SQRT_2              = sqrt(2.0) ;

   for(int d=-Lmax; d<=Lmax; ++d) // for each direction in the BRDF RZH
   {
       FVec3 Rnwpk = TBN_L2G * texelFetch(BRDFDirs1DTex, d+Lmax, 0).xyz;
       FType sin_theta = sqrt(1. - Rnwpk.z*Rnwpk.z) ;

       FType x       =  Rnwpk.z; // cos_theta
       FType sin_1   =  Rnwpk.y / sin_theta ; //sin_phi;
       FType cos_1   =  Rnwpk.x / sin_theta ; //cos_phi;
       FType sin_m   =  0.0 ;
       FType cos_m   =  1.0 ;
       FType radical =  sqrt( 1.0 - x*x );
       FType Pmm     =  ONE_OVER_SQRT_4_PI ;
       FVec3 lambdaLM; // BRDF RZH coefficient
       FVec3 constant;

       for(int M = 0; M <= Mmax; M++)
       {
           FType fact_times_SQRT_4PI = ((M > 0)?SQRT_2:1.0) * SQRT_4_PI ;
           if(abs(d) <= M)
           {
               lambdaLM = texelFetch(BRDFCoefs2D_RZH_Tex, ivec2(lmtoi(M,d),thetaViewLocalIndex), 0).rgb;

               constant = Pmm * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(M)+1.) ;

               BRDF_global_SH_coeffs[M*(M+1)+M] += cos_m * constant; //ylm[lmtoi(M,M)] = Pmm * cos_m * fact;

               if(M!=0)
```

```
                BRDF_global_SH_coeffs[M*(M+1)-M] += - sin_m * constant; //Ylm(M,-M) = - Pmm * sin_m * fact;
            }
            FType prev_P = Pmm ;
            FType curr_P = x * sqrt(FType((M*2)+3)) * Pmm ;
            if(M != Mmax && abs(d) <= M+1)
            {
                lambdaLM  = texelFetch(BRDFCoefs2D_RZH_Tex, ivec2(lmtoi(M+1,d),thetaViewLocalIndex), 0).rgb;

                constant = curr_P * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(M+1.)+1.) ;

                BRDF_global_SH_coeffs[(M+1)*(M+1+1)+M] += cos_m * constant;

                if(M!=0)
                   BRDF_global_SH_coeffs[(M+1)*(M+1+1)-M] += - sin_m * constant;
            }
            for(int L = M+2; L<=Lmax; L++)
            {
                FVec2 c12 = coefs(L,M) ;
                FType temp = x * c12.x * curr_P - c12.y * prev_P ;
                prev_P = curr_P;
                curr_P = temp ;/// ( L - M );
                if(abs(d)<=L)
                {
                   lambdaLM = texelFetch(BRDFCoefs2D_RZH_Tex, ivec2(lmtoi(L,d),thetaViewLocalIndex), 0).rgb;

                   constant = curr_P * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(L)+1.) ;

                   BRDF_global_SH_coeffs[L*(L+1)+M] += cos_m * constant;

                   if(M!=0)
                      BRDF_global_SH_coeffs[L*(L+1)-M] += - sin_m * constant;
                }
            }
            Pmm *= - radical * sqrt( ((M*2)+3)/FType((M*2)+2) ) ;

            FType temp = cos_1 * cos_m - sin_1 * sin_m;
            sin_m = sin_1 * cos_m + cos_1 * sin_m;
            cos_m = temp;
        }

}

FVec3 res ;

// BLOCK 2 - multiply by triple product matrix to obtain the RZH coefs of BRDF x Envmap

int MaxLvis = Lmax ;
int max_lm1 = (MaxLvis+1)*(MaxLvis+1) ;
int max_lm2 = (Lmax+1)*(Lmax+1) ;

for(int LM=0;LM<max_lm1;++LM)
{
   FVec3 val = FVec3(0.,0.,0.) ;

   for(int lm=0;lm<max_lm2;++lm)
      val += texelFetch(TripleProductMatrixTex,ivec2(LM,lm),0).rgb * BRDF_global_SH_coeffs[lm] ;
```

```
   BRDF_Envmap_RZH_product_coefs[LM] = val ;
}

// BLOCK 3 - use these coefficients in the correct local CS to compute the shading.

res = FVec3(0.0,0.0,0.0) ;

const int Mmax2 = MaxLvis ;

for(int d=-MaxLvis; d<=MaxLvis; ++d) // for each direction in the BRDF RZH
{
    FVec3 Rnwpk = texelFetch(BRDFDirs1DTex, d+Lmax, 0).xyz; // the directions are in local coordinates
    FType sin_theta = sqrt(1. - Rnwpk.z*Rnwpk.z) ;

    FType x       =  Rnwpk.z; // cos_theta
    FType sin_1   =  Rnwpk.y / sin_theta ; //sin_phi;
    FType cos_1   =  Rnwpk.x / sin_theta ; //cos_phi;
    FType sin_m   =  0.0 ;
    FType cos_m   =  1.0 ;
    FType radical =  sqrt( 1.0 - x*x );
    FType Pmm     =  ONE_OVER_SQRT_4_PI ;
    FVec3 lambdaLM; // BRDF RZH coefficient
    FVec3 constant;
    for(int M = 0; M <= Mmax2; M++)
    {
        FType fact_times_SQRT_4PI = ((M > 0)?SQRT_2:1.0) * SQRT_4_PI ;
        if(abs(d) <= M)
        {
            lambdaLM = BRDF_Envmap_RZH_product_coefs[M*(M+1)+d] ;

            constant = Pmm * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(M)+1.) ;
            res += cos_m * getVisibilityMapCoef(M,M) * constant;
            if(M!=0)
                res += - sin_m * getVisibilityMapCoef(M,-M) * constant;
        }
        FType prev_P = Pmm ;
        FType curr_P = x * sqrt(FType((M*2)+3)) * Pmm ;
        if(M != Mmax2 && abs(d) <= M+1)
        {
            lambdaLM = BRDF_Envmap_RZH_product_coefs[(M+1)*(M+2)+d] ;

            constant = curr_P * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(M+1.)+1.) ;
            res += cos_m * getVisibilityMapCoef(M+1,M) * constant;
            if(M!=0)
                res += - sin_m * getVisibilityMapCoef(M+1,-M) * constant;
        }
        for(int L = M+2; L<=MaxLvis; L++)
        {
            FVec2 c12 = coefs(L,M) ;
            FType temp = x * c12.x * curr_P - c12.y * prev_P ;
            prev_P = curr_P;
            curr_P = temp ;/// ( L - M );
            if(abs(d)<=L)
            {
                lambdaLM = BRDF_Envmap_RZH_product_coefs[L*(L+1)+d] ;

                constant = curr_P * fact_times_SQRT_4PI * lambdaLM / sqrt(2.*(L)+1.) ;
```

```
                res += cos_m * getVisibilityMapCoef(L,M) * constant;
                if(M!=0)
                    res += - sin_m * getVisibilityMapCoef(L,-M) * constant;
            }
        }
        Pmm *= - radical * sqrt( ((M*2)+3)/FType((M*2)+2) ) ;

        FType temp = cos_1 * cos_m - sin_1 * sin_m;
        sin_m = sin_1 * cos_m + cos_1 * sin_m;
        cos_m = temp;
    }
  }

return res;
```

5 Screen-space shape operator shader

This is the shader for the screen-space shape operator that is used for antialiasing reflexions.

```
vec3 getShapeOperator()
{
   // We estimate the shape operator, which is the matrix
   //        S = [ a b ]
   //            [ c d ]
   // ...that gives the derivative of the normal when moving tangent to the surface
   // in a direction indicated by 2D vector v:
   //
   //          S * v = \frac{ d n }{ d v }
   //
   // Once we have v1,v2,c1 and c2, we know that a symmetric gaussian beam that
   // comes from a pixel, will bounce on the surface and spread according to c1
   // and c2, in directions v1 and v2.  Along v1, for instance, the angular
   // spread of a beam of angular size \delta\theta, after reflexion will be
   //
   //               \delta\theta_1 = \delta\theta d / \cos\theta  * c1
   //
   //  ...where
   //     d                   is the distance between the camera and the objet we hit
   //     \delta\theta        is the angular with of a pixel ( Fov / W, basically )
   //     c1                  curvature along v1
   //     \theta              angle between normal and view direction.
   //
   // Now to compute the shape operator, we need
   //
   //    1 - to compute the 3D vectors X and Y, tangent to the surface, that
   //    project on (1,0,0) and (0,1,0) in the screen (assuming z=(0,0,1) is
   //    the depth direction). We have therefore X.n=0 and Y.n=0.  Assuming P
   //    is the point in world space and C is the camera in world space:
   //
   //            Vector xx = ScreenToWorld( vec3(1,0,0) )
   //            Vector yy = ScreenToWorld( vec3(0,1,0) )
   //            Vector zz = ScreenToWorld( vec3(0,0,1) )
   //
   //            X = normalize(  xx - (n.xx)zz/ (n.zz) )   // you can check, X.n=0
   //            Y = normalize(  yy - (n.yy)zz/ (n.zz) )   // you can check, Y.n=0
   //
   vec2 uv = gl_FragCoord.xy / ScreenSize ;
   vec3 normal00 = normalize( texture(NormalTex, uv).rgb );
   vec3 position00 = normalize( texture(PositionTex, uv).rgb );

   vec3 X = normalize( xx - dot(normal00,xx) * zz / dot(normal00,zz) ) ;
   vec3 Y = normalize( yy - dot(normal00,yy) * zz / dot(normal00,zz) ) ;

   //    2 - compute the values of the shape operator along X and Y using:
   //
   //      vec3 Sx = normals( current_pixel + (1,0)) - normals(current_pixel)
   //      vec3 Sy = normals( current_pixel + (0,1)) - normals(current_pixel)

   vec3 Sx = ( textureOffset(NormalTex, uv, ivec2(1,0)).rgb - normal00 );
   vec3 Sy = ( textureOffset(NormalTex, uv, ivec2(0,1)).rgb - normal00 );
```

```
    //
    //    3 - define a vector B, orthogonal to X and n (Y is not, actually), and
    //      express B as a linear combination of X and Y
    //
    //      vec3 B = n^X                  // with this, we have B= (Y-(X.Y)X) / |Y-(X.Y)X|
    //      float l1 = -X.Y / norm(Y - (X.Y)X)
    //      float l2 =    1 / norm(Y - (X.Y)X)   // we now have B=l1 X + l2 Y

    vec3 B = cross(normal00, X) ;
    float norm = length(Y-dot(X,Y)*X) ;
    float l1 = dot(-X,Y) / norm ;
    float l2 = 1.0 / norm ;

    //
    //    4 - now convert Sx and Sy into the (X,B,n) coordinate system
    //
    //       vec2 shape_op_times_X = ( X.Sx , B.Sx ) // normally B.Sx=0. To be verified.
    //       vec2 shape_op_times_Y = ( X.Sy , B.Sy )

    vec2 shape_op_times_X = vec2( dot(X,Sx) , dot(B,Sx) ) ;
    vec2 shape_op_times_Y = vec2( dot(X,Sy) , dot(B,Sy) ) ;

    //
    //    5 - compute the shape operator, by computing both columns as 2D vectors:
    //
    //       Vec2 shape_op_column1 = shape_op_times_X
    //       Vec2 shape_op_column2 = shape_op_times_X * l1 + shape_op_times_Y * l2

    vec2 shape_op_column1 = shape_op_times_X ;
    vec2 shape_op_column2 = shape_op_times_X * l1 + shape_op_times_Y * l2 ;

    return vec3(shape_op_column1.x, shape_op_column2.y, shape_op_column1.y);
}
```

6 Antialiasing reflexion shader

Below is the isotropic version of our antialiasing shader. The mip-level is used to interpolate the precomputed $\beta_{ll'}$ coefficients of the Gaussian stored in a texture.

```
vec3 antialiasCubemap(int Lmax,vec3 dir,float level)
{
   // 1 - compute the trigonometric coordinates of dir

   float cos_theta = dir[2] ;
   float sin_theta = sqrt(1.0f - cos_theta*cos_theta) ;
   float cos_phi = 1.0 ;
   float sin_phi = 0.0 ;

   if(sin_theta != 0.0)
   {
      cos_phi = dir[0]/sin_theta ;
      sin_phi = dir[1]/sin_theta ;
   }

   const float ONE_OVER_SQRT_4_PI = 1.0/sqrt(4.0*M_PI) ;
   const float SQRT_2 = sqrt(2.0) ;

   vec3 res = vec3(0.,0.,0.) ;

   float x       =  cos_theta;
   float sin_1   =  sin_phi;
   float cos_1   =  cos_phi;
   float sin_m   =  0.0 ;
   float cos_m   =  1.0 ;
   float radical =  sqrt( 1.0 - x*x );
   float Pmm     =  ONE_OVER_SQRT_4_PI ;

   for(int M = 0; M <= Lmax; M++ )
   {
      float fact = (M > 0)?SQRT_2:1.0 ;

      {
         float mllp = get_mllp_coef(level,M) ;// coefficient \beta_{ll'} from Eq.16 & 28
         float gstar = sqrt(4*M_PI/(2*M+1)) ; // comes from the convolution formula

         float ff = gstar * mllp * Pmm * fact ;

         // getCubeMapCoef(L,M) returns the SH coeficient of the cubemap.

         res +=  cos_m * ff * getCubemapCoef(M,M);

         if(M!=0)
            res += - sin_m * ff * getCubemapCoef(M,-M);
      }

      float prev_P = Pmm ;
      float curr_P = x * sqrt(float((M*2.)+3.)) * Pmm ;

      if(M!=Lmax)
      {
         float mllp = get_mllp_coef(level,M+1) ;
```

```
         float gstar = sqrt(4*M_PI/(2*M+3)) ;

         float ff = gstar * mllp * curr_P * fact ;

         res += cos_m * ff * getCubemapCoef(M+1,M);

         if(M!=0)
            res +=  - sin_m * ff * getCubemapCoef(M+1,-M);
      }

      int LL = (M+2)*2 ;

      for(int L = M+2; L<=Lmax; L++,LL+=2 )
      {
         vec2 c12 = coefs(L,M) ; // Coeficients for Eq. 4 of supplemental document

         float temp = x * c12.x * curr_P - c12.y * prev_P ;
         prev_P = curr_P;
         curr_P = temp ;/// ( L - M );

         float mllp = get_mllp_coef(level,L) ;
         float gstar = sqrt(4*M_PI/(2*L+1)) ;
         float ff = gstar * mllp * curr_P * fact ;

         res += cos_m * ff * getCubemapCoef(L,M);

         if(M!=0)
            res += - sin_m * ff * getCubemapCoef(L,-M);
      }

      Pmm   *= - radical * sqrt( ((M*2.)+3.)/float((M*2.)+2.) ) ;

      float temp   = cos_1 * cos_m - sin_1 * sin_m;
      sin_m  = sin_1 * cos_m + cos_1 * sin_m;
      cos_m  = temp;
   }

   return res ;
}
```