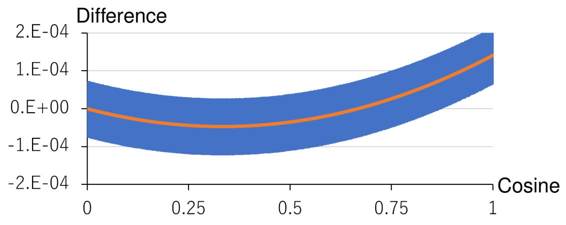
# Accurate Diffuse Lighting from Spherical Gaussian Lights (Supplementary Document)

Yusuke Tokuyoshi
Advanced Micro Devices, Inc.

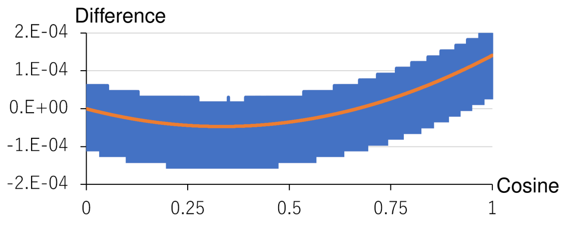
## 1 Accuracy of Our Approximation

### 1.1 Clamped Cosine

Fig. 1 shows the error of our spherical Gaussian (SG) approximation for clamped cosine. Our approximation balances the approximation error and numerical error to reduce the sum of them. Even without using Kahan's precise computation [Kahan(2004)] for Eq. 3 in the main document, the maximum numerical error is still almost the same as the maximum approximation error.



(a) with Kahan's algorithm



(b) w/o Kahan's algorithm

Figure 1: Plots of the difference between our SG approximation and exact clamped cosine (orange: approximation error, blue: numerical error).

### 1.2 Steepness for the Normalized Hemispherical Integral

Fig. 2 shows the plot of steepness $t(\kappa)$ for the normalized hemispherical integral (Eq. 5 in the main document). The reference is numerically computed optimal values. For $\kappa \to \infty$, this optimal steepness asymptotically approaches to $\sqrt{\kappa/2}$ that is the steepness of the cumulative distribution function of a Gaussian on a plane. Our approximation (Eq. 6 in the main document) is close to the reference, and it is also asymptotically approaches to $\sqrt{\kappa/2}$ for $\kappa \to \infty$.

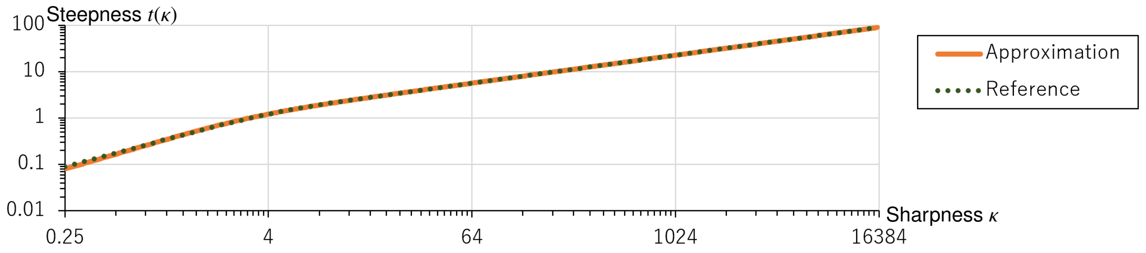


Figure 2: Plots of the steepness $t(\kappa)$. Our approximation (orange) is close to the reference (dots).

## 2 Implementation Details

We implement our method in real-time indirect illumination using virtual SG lights [Tokuyoshi(2015)] whose source code is available in https://github.com/yusuketokuyoshi/VSGL. Listing 1 shows HLSL code for the product integral of an SG and clamped cosine using our approximation. In this code, the *SGProduct* function computes the product of two SGs, and the *HSGIntegral* function computes the hemispherical integral of an SG. The implementations of these functions are shown in Listings 2 and 3. Since a straightforward implementation for them is numerically unstable, we use some numerically stable algorithms in our implementation.

Listing 1: Product integral of an SG and clamped cosine (HLSL). Our improrovement for the clamped-cosine approximation is written in red.

```
struct SGLobe {
 float3 axis;
 float sharpness;
 float logAmplitude;
};

float HSGCosineProductIntegral(SGLobe sg, float3 normal) {
 float LAMBDA = 0.00084560872241480124;
 float ALPHA = 1182.2467339678153;
 SGLobe prodLobe = SGProduct(sg.axis, sg.sharpness, normal, LAMBDA);
 float p = HSGIntegral(dot(prodLobe.axis, normal), prodLobe.sharpness) * exp(LAMBDA + prodLobe.logAmplitude);
 float q = HSGIntegral(dot(sg.axis, normal), sg.sharpness);

 return exp(sg.logAmplitude) * max(ALPHA * p - ALPHA * q, 0.0);
}
```

### 2.1 SG Product

The product of two SGs is given by

$$G(\boldsymbol{\omega}; \mathbf{v}_1, \kappa_1) G(\boldsymbol{\omega}; \mathbf{v}_2, \kappa_2) = \mathrm{e}^{\kappa_3 - \kappa_1 - \kappa_2} G\left(\boldsymbol{\omega}; \frac{\kappa_1 \mathbf{v}_1 + \kappa_2 \mathbf{v}_2}{\kappa_3}, \kappa_3\right), \tag{1}$$

where $\kappa_3 = \|\kappa_1 \mathbf{v}_1 + \kappa_2 \mathbf{v}_2\|$. However, when sharpness $\kappa_1$ or $\kappa_2$ is large, $\kappa_3 - \kappa_1 - \kappa_2$ can produce a catastrophic cancellation. To improve the numerical stability for large $\kappa_1$ or $\kappa_2$, we use the following form:

$$\kappa_3 - \kappa_1 - \kappa_2 = \frac{2\kappa_{\min}((\mathbf{v}_1 \cdot \mathbf{v}_2) - 1)}{1 + \frac{\kappa_{\min}}{\kappa_{\max}} + \sqrt{2\frac{\kappa_{\min}}{\kappa_{\max}}(\mathbf{v}_1 \cdot \mathbf{v}_2) + \left(\frac{\kappa_{\min}}{\kappa_{\max}}\right)^2 + 1}}, \tag{2}$$

where $\kappa_{\min} = \min(\kappa_1, \kappa_2)$ and $\kappa_{\max} = \max(\kappa_1, \kappa_2)$. The HLSL code for this SG product is shown in Listing 2.

Listing 2: Numerically stable SG product (HLSL).

```
SGLobe SGProduct(float3 axis1, float sharpness1, float3 axis2, float sharpness2) {
 float3 axis = axis1 * sharpness1 + axis2 * sharpness2;
 float sharpness = length(axis);
 float cosine = clamp(dot(axis1, axis2), -1.0, 1.0);
 float sharpnessMin = min(sharpness1, sharpness2);
 float sharpnessRatio = sharpnessMin / max(sharpness1, sharpness2);
 float logAmplitude = 2.0 * sharpnessMin * (cosine - 1.0) / (1.0 + sharpnessRatio + sqrt(2.0 * sharpnessRatio * cosine
      + sharpnessRatio * sharpnessRatio + 1.0));
 SGLobe result = { axis / max(sharpness, FLT_MIN), sharpness, logAmplitude };
 return result;
}
```

### 2.2 Hemispherical Integral

The hemispherical integral of an SG is given by an interpolation between upper hemispherical integral $A(\kappa) = 2\pi(1 - \mathrm{e}^{-\kappa})/\kappa$ and lower hemispherical integral $B(\kappa) = 2\pi\mathrm{e}^{-\kappa}(1 - \mathrm{e}^{-\kappa})/\kappa$. However, the calculation of $(1 - \mathrm{e}^{-\kappa})/\kappa$ can produce a large numerical error if sharpness $\kappa$ is small. Therefore, we use a numerically stable algorithm [Higham(2002)] to compute $(e^x - 1)/x$ where $x = -\kappa$ (see Listing 4 for HLSL implementation). Listing 3 shows our hemispherical integral using this algorithm. For the interpolation factor (i.e., normalized hemispherical integral) in Listing 3, our approximation uses the error function (erf). Although erf is available in some computer languages (such as C++), HLSL does not have a built-in erf. Therefore, we use Listing 5 for HLSL.

Listing 3: Numerically stable hemispherical integral of an SG (HLSL). Our improvement for the normalized hemispherical integral is written in red.

```
float HSGIntegral(float cosine, float sharpness) {
 // Our fitted steepness (Eq. 6 in the main document).
 float steepness = sharpness * sqrt((0.5 * sharpness + 0.65173288269070562) / ((sharpness + 1.3418280033141288) *
      sharpness + 7.2216687798956709));

 // Our approximation for the normalized hemispherical integral (Eq. 5 in the main document).
 float s = 0.5 + 0.5 * (erf(steepness * clamp(cosine, -1.0, 1.0)) / erf(steepness));

 // Interpolation between upper and lower hemispherical integrals.
 return 2.0 * M_PI * lerp(exp(-sharpness), 1.0, s) * expm1_over_x(-sharpness);
}
```

Listing 4: $(e^x - 1)/x$ with cancellation of rounding errors [Higham(2002)] (HLSL).

```
float expm1_over_x(float x) {
 float u = exp(x);
 if (u == 1.0) {
  return 1.0;
 }
 float y = u - 1.0;
 if (abs(x) < 1.0) {
  return y / log(u);
 }
 return y / x;
}
```

Listing 5: Error function (HLSL).

```
float erf(float x) {
 // Early return for large |x|.
 if (abs(x) >= 4.0) {
  return asfloat((asuint(x) & 0x80000000) ^ asuint(1.0));
 }

 // Polynomial approximation based on https://forums.developer.nvidia.com/t/optimized-version-of-single-precision-error
      -function-erff/40977
 if (abs(x) > 1.0) {
  float A1 = 1.628459513;
  float A2 = 9.15674746e-1;
  float A3 = 1.54329389e-1;
  float A4 = -3.51759829e-2;
  float A5 = 5.66795561e-3;
  float A6 = -5.64874616e-4;
  float A7 = 2.58907676e-5;
  float a = abs(x);
  float y = 1.0 - exp2(-(((((((A7 * a + A6) * a + A5) * a + A4) * a + A3) * a + A2) * a + A1) * a));
  return asfloat((asuint(x) & 0x80000000) ^ asuint(y));
 } else {
  float A1 = 1.128379121;
  float A2 = -3.76123011e-1;
  float A3 = 1.12799220e-1;
  float A4 = -2.67030653e-2;
  float A5 = 4.90735564e-3;
  float A6 = -5.58853149e-4;
  float x2 = x * x;
  return (((((A6 * x2 + A5) * x2 + A4) * x2 + A3) * x2 + A2) * x2 + A1) * x;
 }
}
```

## References

[Higham(2002)] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics.

[Kahan(2004)] William Kahan. 2004. On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic. https://people.eecs.berkeley.edu/~wkahan/Qdrtcs.pdf

[Tokuyoshi(2015)] Yusuke Tokuyoshi. 2015. Fast Indirect Illumination Using Two Virtual Spherical Gaussian Lights. In *SIGGRAPH Asia '15 Posters*. 12:1–12:1.