Real-Time Approximation of a Normal Distribution Function for Normal-Mapped Surfaces

Han-sung SON†, Nonmember and JungHyun HAN†a), Member

SUMMARY This paper proposes to pre-compute approximate normal distribution functions and store them in textures such that real-time applications can process complex specular surfaces simply by sampling the textures. The proposed method is compatible with the GPU pipeline-based algorithms, and rendering is completed at real time. The experimental results show that the features of complex specular surfaces, such as the glinty appearance of leather and metallic flakes, are successfully reproduced.

1. Introduction

A normal map is a special texture that stores surface normals of an object. It is an indispensable component for realistic rendering. When the textured surface becomes small in the screen space, however, an aliasing artifact may be observed. In order to resolve this problem, Toksvig [1] proposed to determine the variance of Gaussians according to the lengths of the sampled (unnormalized) normal vectors. This method alleviated the aliasing problem but could not express glints on complex surfaces. Olano et al. [2] proposed to approximate the normal distribution function (NDF) with an anisotropic Gaussian, but the approximated NDF was often different from the original NDF. Han et al. [3] and Xu et al. [4] attempted to approximate the NDF with a few Gaussians, but the complex NDF of specular surfaces was not correctly reconstructed.

Given a normal map, we propose to pre-compute approximate NDFs and store them in textures such that real-time applications can process complex specular surfaces simply by sampling the textures. The experimental results prove that rendering is done quite fast and the features of complex specular surfaces are successfully reproduced.

2. Pre-Computation of Approximate NDF

We decompose the input normal map into a regular grid of square tiles. Shown on the left of Fig. 1-(a) is an example normal map of resolution 64×64 (For the sake of presentation, the normal map size is made excessively small). In the current implementation, the resolution of a tile is set to 8×8, and therefore the example normal map is divided into 64 tiles. A discrete NDF is computed per tile using a modified version of the method proposed by Yan et al. [5].

Given a unit vector represented in (s, t, \sqrt{1 - (s^2 + t^2)}), its density in a normal map T is computed as follows:

\[ D(s) = \frac{1}{S(T)} \int_T G_s(n(u) - s) \, du \]  

where \( s \) denotes (s, t), \( S(T) \) is the number of texels in T, \( G_s \) represents the intrinsic roughness Gaussian, \( n(u) \) represents the texture coordinates (u, v) used for accessing the normal map, and \( n(u) \) denotes the first two coordinates of a unit normal obtained by filtering the normal map with \( u \).

Figure 1-(a) visualizes an NDF with horizontal s axis and vertical t axis. Each pixel in the visualized NDF rep-
represents \( D(s) \) computed in Eq. (1). Note that \( D(s) \) is defined inside the circular area only.

The 2D NDF is converted into a 1D array with 4096 (64×64) elements. Shown on the left of Fig. 1-(b) are 64 such 1D arrays, each of which represents a tile’s NDF. Principal component analysis (PCA) is applied to the stack of arrays to produce the eigenvectors. The number of eigenvectors can be determined as needed. Figure 1-(b) shows 16 eigenvectors, \( \{e_0, e_1, \ldots, e_{15}\} \). Each eigenvector is converted back to a 2D array with 64×64 elements. Shown in Fig. 1-(c) are 16 such 2D arrays, which we name eigenvector texture.

Given 64 NDFs, PCA computes the mean of 64 density values for each of 4096 sample positions. The mean values make up another 2D array of resolution 64×64, which we denote by \( M \). Then, an NDF is approximated as follows:

\[
D \approx \sum_{i=0}^{15} w_i e_i + M \tag{2}
\]

where \( w_i \) denotes the weight of \( e_i \). Each NDF is associated with a distinct set of 16 weights, \( \{w_0, w_1, \ldots, w_{15}\} \). The weights of all NDFs are stored in another texture named weight texture. Conceptually, it is a collection of 2D arrays, as shown in Fig. 1-(d). The number of arrays is equal to that of eigenvectors, and the number of elements in each array is equal to the number of tiles. The weights of an NDF are stored in the elements of the same 2D position. For example, the weights for \( D_4 \) are stored at the upper-right corners of 16 2D arrays (with a blue ellipse in Fig. 1-(d)).

3. Runtime Texturing and Weight Mipmapping

Let us present how the eigenvector and weight textures are processed at runtime. Consider a pixel with texture coordinates, \((u, v)\), which we denote by \( u \). Then, the weight texture is filtered with \( u \). On the other hand, suppose a half vector (between the incident and reflected light vectors) \((s, t, \sqrt{1-(s^2+t^2)})\) at the pixel. Let \( s \) denote \((s, t)\). Then, the density of \( s \) at \( u \) is computed as follows:

\[
D(u, s) \approx W(u) \cdot E(s) + M(s) \tag{3}
\]

where \( W(u) \) is a 16D vector obtained from the weight texture, \( E(s) \) is another 16D vector obtained from the eigenvector texture, \( \cdot \) denotes the dot product operation, and \( M(s) \) represents the mean density at \( s \). Equation (3) implies that \( D(u, s) \) is obtained simply by texture sampling.

Figure 2 compares the NDFs reconstructed with different numbers of eigenvectors. As the number of eigenvectors is increased, the reconstructed NDF becomes closer to the original.

![Fig. 2](image)

Fig. 2 NDFs reconstructed with different numbers of eigenvectors. As the number of eigenvectors is increased, the reconstructed NDF becomes closer to the original.

Mipmapping the weight texture. (a) A pixel footprint covers four tiles of the normal map. (b) Levels 0 and 1 of the weight mipmap.

![Fig. 3](image)

Fig. 3 Mipmapping the weight texture. (a) A pixel footprint covers four tiles of the normal map. (b) Levels 0 and 1 of the weight mipmap.
stored in the weight texture. Most mipmap implementations employ a box filter, and \( \frac{W_{x1} + W_{x2} + W_{y1} + W_{y2}}{4} \) in Eq. (4) is the result of box-filtering four texels of the original weight texture. It is stored at level 1 of the mipmap. See Fig. 3-(b).

Suppose that the pixel footprint is smaller than a tile. Then, among 8×8 texels of the tile, those outside the footprint should not be involved in lighting. For this, we resort to the normalized Blinn-Phong distribution function [7], for which we provide a normal obtained by general normal map sampling (normalized mipmapping). Then, the output of the Blinn-Phong function is blended with that of filtering the eigenvector/weight textures through Hermite interpolation: The closer the camera is to the surface, the more the Blinn-Phong function contributes. The proposed scheme is so flexible that the Blinn-Phong distribution function can be replaced by other functions such as Beckmann distribution [8].

4. Experimental Results

The proposed method has been implemented on a PC with Intel i5-2500K 3.30GHz CPU, 12GB RAM, and NVIDIA GeForce GTX560 Ti GPU. Both NDF calculation and texture generation were programmed in C#.

Figure 4 compares our method with general normal mapping. It can be observed that the glinty appearance becomes more noticeable as the number of eigenvectors increases. Simultaneously, the frame rate decreases because the texture sampling count increases.

Using another model, Fig. 5 compares our method with general normal mapping. If the distance between the model and the camera is long, the normal-mapped surface looks like a smooth plastic material. In contrast, our method presents the glinty appearance well independently of the distances between the camera and the surface. Figure 6 shows other objects rendered with our method.

5. Conclusion

This paper proposed to pre-compute approximate NDFs with eigenvector and weight textures, which are then sampled at runtime to realistically render complex specular surfaces. The proposed method is compatible with mipmapping, and therefore traditional GPU programs can benefit from the proposed method. In addition to the original normal map, the method requires texture slots for eigenvector and weight textures, which are fortunately afforded by contemporary GPUs.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (No. NRF-2017M3C4A7066316).

References

[1] M. Toksvig, “Mipmapping normal maps,” journal of graphics tools, vol.10, no.3, pp.65–71, 2005.
[2] M. Olano and D. Baker, “Lean mapping,” Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, pp.181–188, ACM, 2010.
[3] C. Han, B. Sun, R. Ramamoorthi, and E. Grinspun, “Frequency domain normal map filtering,” ACM Transactions on Graphics (TOG), vol.26, no.3, p.28, ACM, 2007.
[4] C. Xu, R. Wang, S. Zhao, and H. Bao, “Real-time linear brdf mip-mapping,” Computer Graphics Forum, vol.36, no.4, pp.27–34, Wiley Online Library, 2017.
[5] L.-Q. Yan, M. Hašan, W. Jakob, J. Lawrence, S. Marschner, and R. Ramamoorthi, “Rendering glints on high-resolution normal-mapped specular surfaces,” ACM Transactions on Graphics (TOG), vol.33, no.4, p.116, 2014.
[6] L. Williams, “Pyramidal parametrics,” ACM Siggraph Computer Graphics, vol. 17, no. 3, pp. 1–11, ACM, 1983.

[7] E.P. Lafortune and Y.D. Willems, “Using the modified Phong reflectance model for physically based rendering,” 1994.

[8] P. Beckmann and A. Spizzichino, The Scattering of Electromagnetic Waves from Rough Surfaces, Radar Library, Artech House, 1987.