Importance Sampling Microfacet-Based BSDFs with the Distribution of Visible Normals

Supplemental Material 2/2

Eric Heitz[1] and Eugene d'Eon[2]

[1]INRIA ; CNRS ; Univ. Grenoble Alpes
[2]The Jig Lab

Abstract

This document provides additional results and details concerning the implementation of the algorithms presented in the associated paper. In Section 1 we provide an exhaustive set of convergence comparisons with the previous method. More results of rough refraction are provided in Section 2. We include in Section 3 the importance sampling schemes for the anisotropic Beckmann and GGX distributions, which can be used in the paper's Algorithm 3. In Section 4 we show how to importance sample non-centered normal distributions that occur with level of detail representations. Finally, in Section 5 presents a C++ implementation of the paper's Algorithm 4 and of the associated precomputation steps.

Contents

1 Convergence Comparisons

In this section we investigate in detail the convergence of the importance sampling scheme. To avoid perturbation due to the environment sampling, we use a perfectly white environment with $L(\omega_o) = 1$ (this is known as the White Furnace Test). We use importance sampling to solve the equation:

$$I = \int_{\Omega} f(\omega_i, \omega_o) \, |\omega_o \cdot \omega_g| \, d\omega_o,$$

whose solution is the average value of the shadowing term (≤ 1).

Table 1 and 2 compares the previous importance sampling technique with ours. Each plot shows 8 instances of the convergence process for this integral for both methods. We observe several properties:

- At normal incidence ($\theta_i = 0$) the techniques have equivalent convergence. In this case $\omega_i = \omega_g$ and thus $D_{\omega_i}(\omega_m) = (\omega_m \cdot \omega_g) D(\omega_m)$, and the techniques are equivalent.
- Our importance sampling is more efficient with incident angles $\theta_i > 1.0$ because this is where the effect of view dependence is important.
- These results shows also that the more the roughness α is important, the better our technique is. This is because view-dependent effects are the most important on bent micronormals, whose presence depends on the roughness.

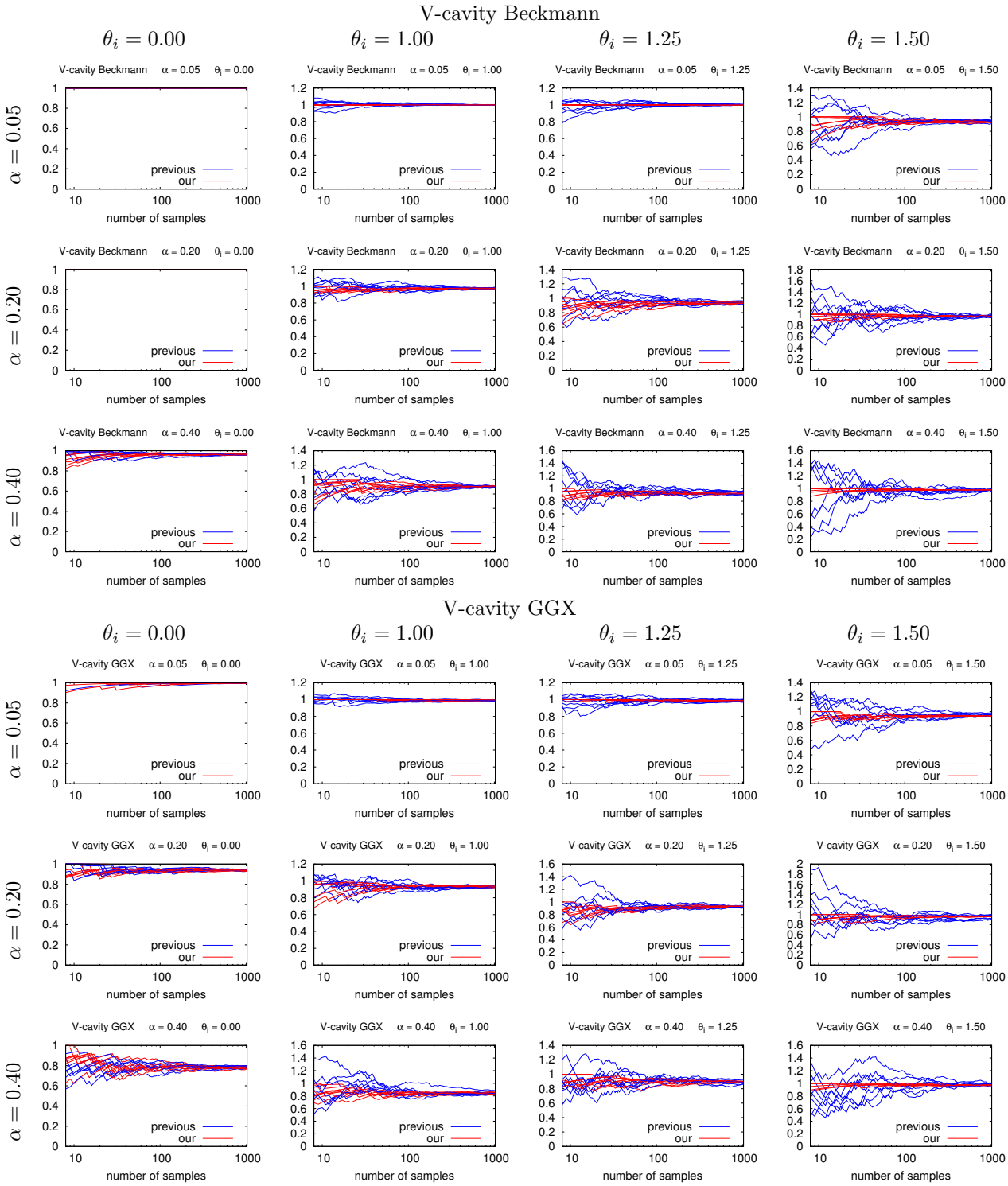

Figure 1: Importance sampling with the V-cavity model.

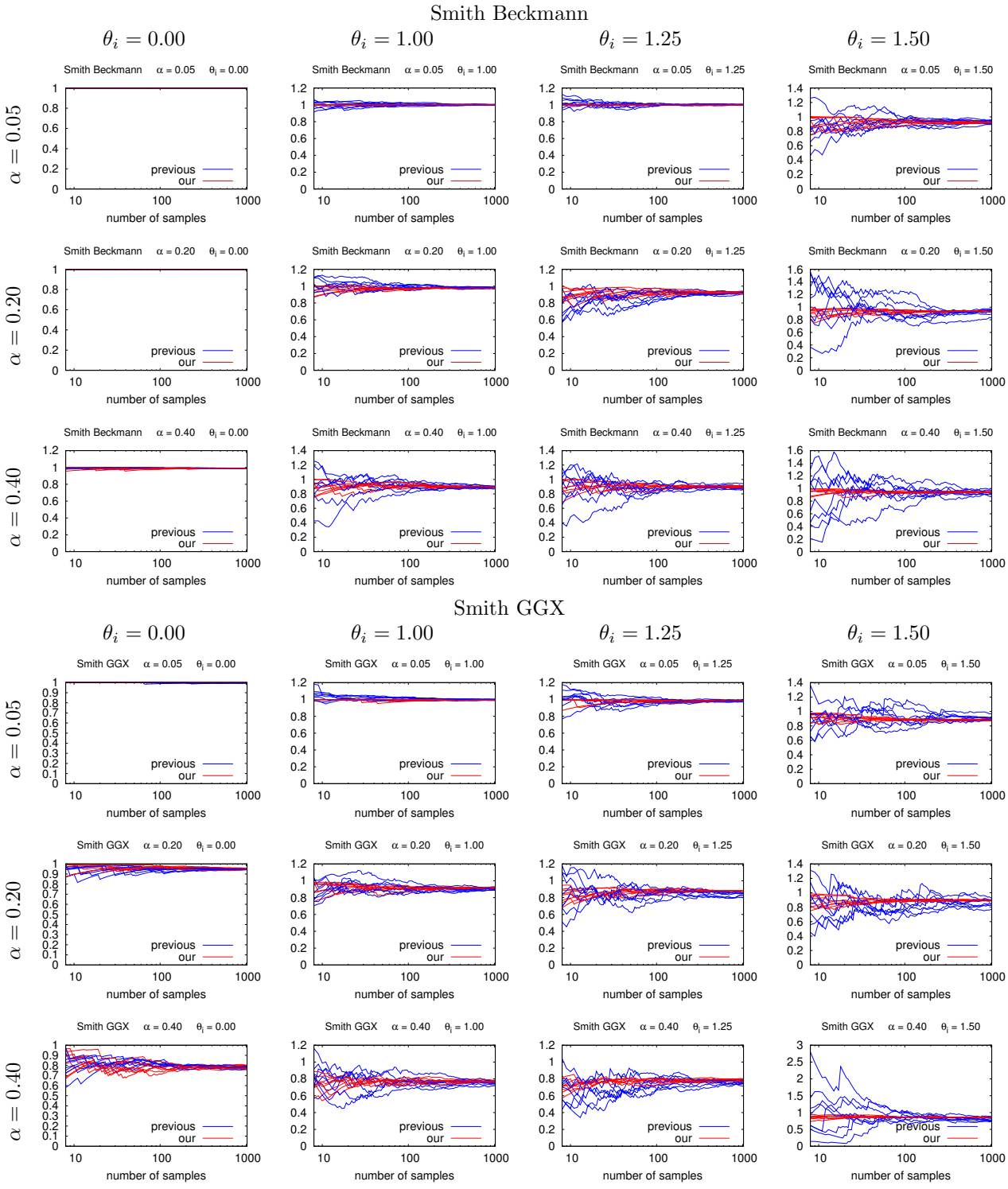

Figure 2: Importance sampling with the Smith model.

2 Results

2.1 Isotropic Beckmann

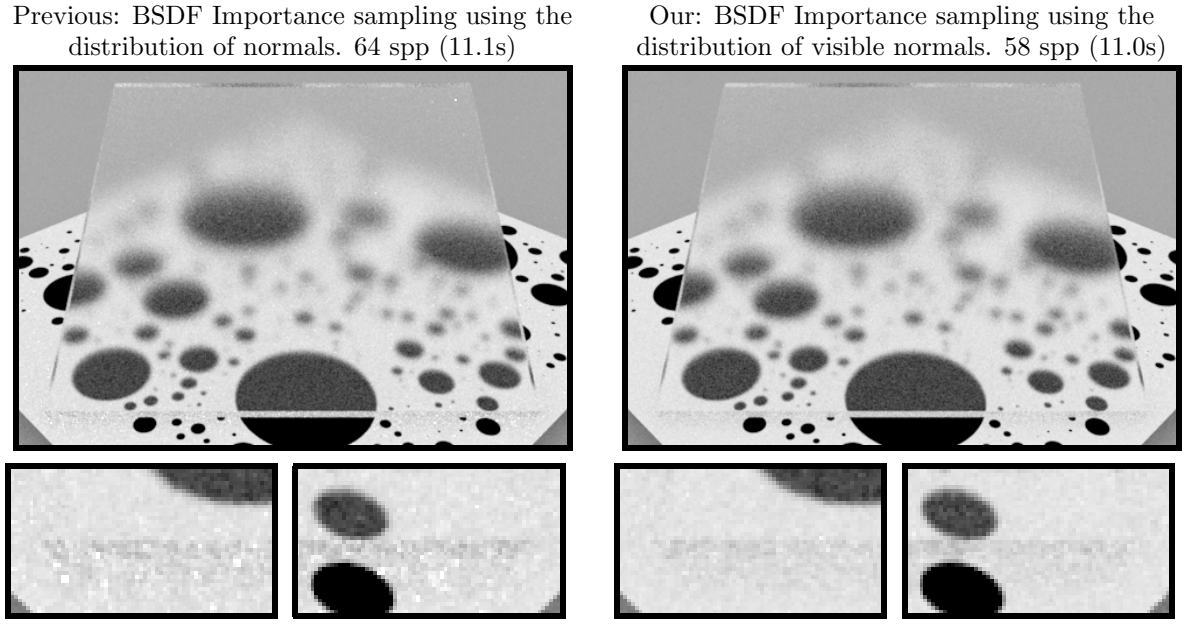

Figure 3: A dielectric glass plate ($n = 1.5$) with isotropic Beckmann roughness ($\alpha = 0.1$) on all faces (with the Smith masking function).

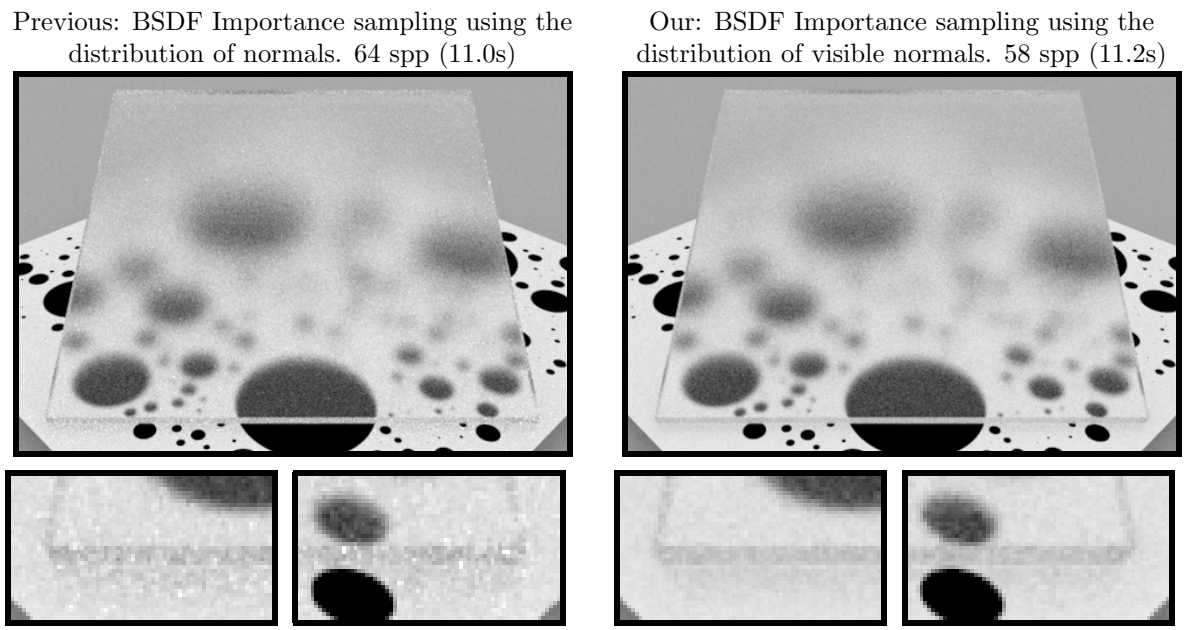

Figure 4: A dielectric glass plate ($n = 1.5$) with isotropic Beckmann roughness ($\alpha = 0.2$) on all faces (with the Smith masking function).

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (10.7s)

Our: BSDF Importance sampling using the distribution of visible normals. 52 spp (10.5s)

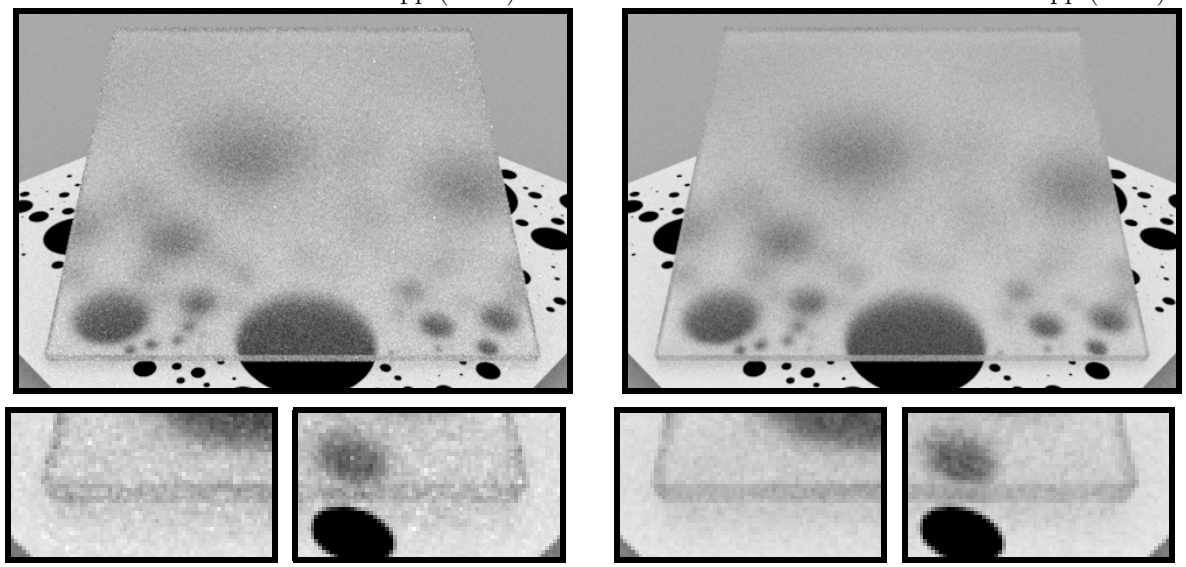

Figure 5: A dielectric glass plate ($n = 1.5$) with isotropic Beckmann roughness ($\alpha = 0.4$) on all faces (with the Smith masking function).

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (9.0s)

Our: BSDF Importance sampling using the distribution of visible normals. 50 spp (8.9s)

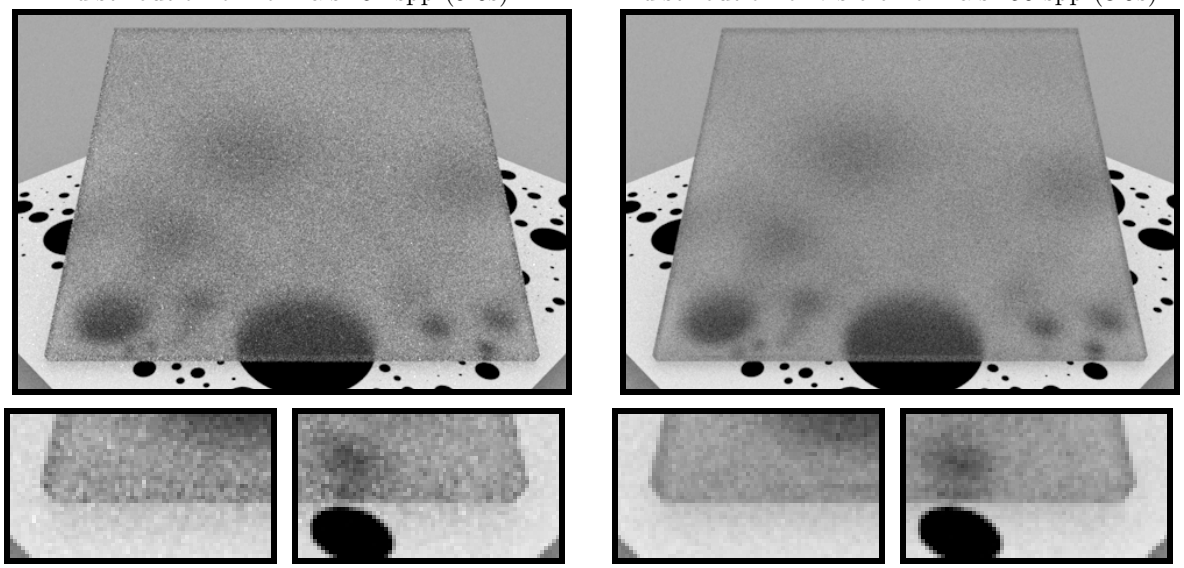

Figure 6: A dielectric glass plate ($n = 1.5$) with isotropic Beckmann roughness ($\alpha = 0.8$) on all faces (with the Smith masking function).

2.2 Anisotropic Beckmann

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (11.2s)

Our: BSDF Importance sampling using the distribution of visible normals. 58 spp (11.3s)

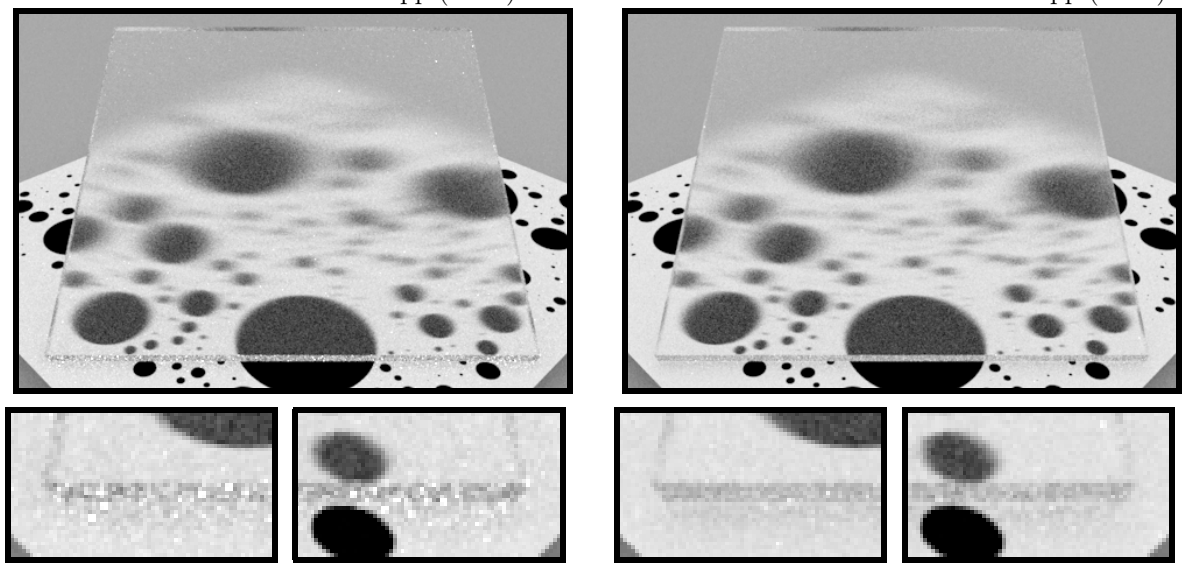

Figure 7: A dielectric glass plate ($n = 1.5$) with anisotropic Beckmann roughness ($\alpha_x = 0.03$, $\alpha_y = 0.24$) on all faces (with the Smith masking function).

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (10.3s)

Our: BSDF Importance sampling using the distribution of visible normals. 54 spp (10.3s)

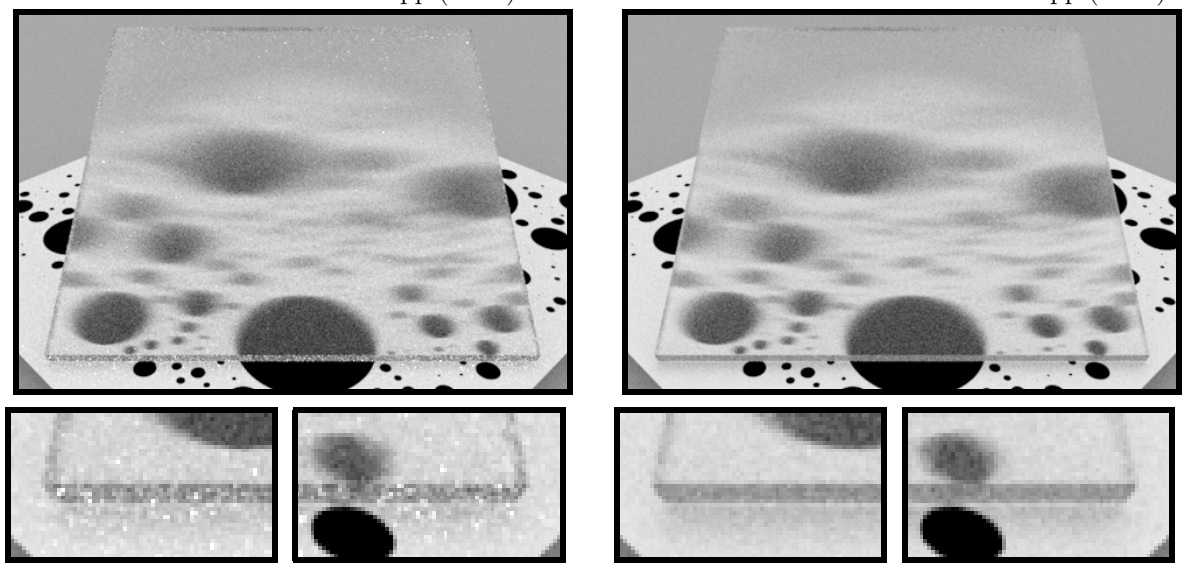

Figure 8: A dielectric glass plate ($n = 1.5$) with anisotropic Beckmann roughness ($\alpha_x = 0.03$, $\alpha_y = 0.48$) on all faces (with the Smith masking function).

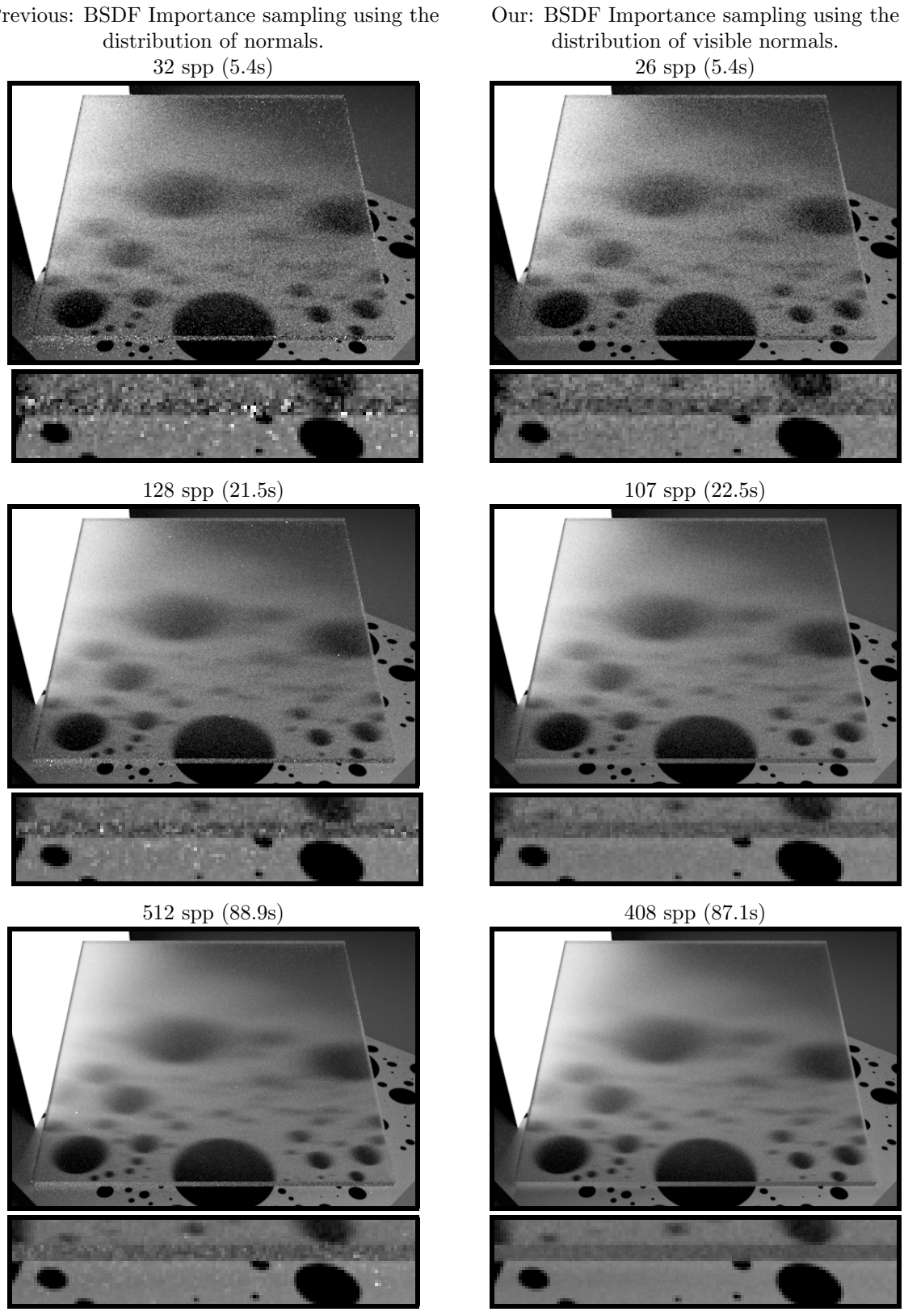

Figure 9: A dielectric glass plate ($n = 1.5$) with anisotropic Beckmann roughness ($\alpha_x = 0.05$, $\alpha_y = 0.40$) on all faces (with the Smith masking function).

2.3 Isotropic GGX

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (10.4s)

Our: BSDF Importance sampling using the distribution of visible normals. 58 spp (10.5s)

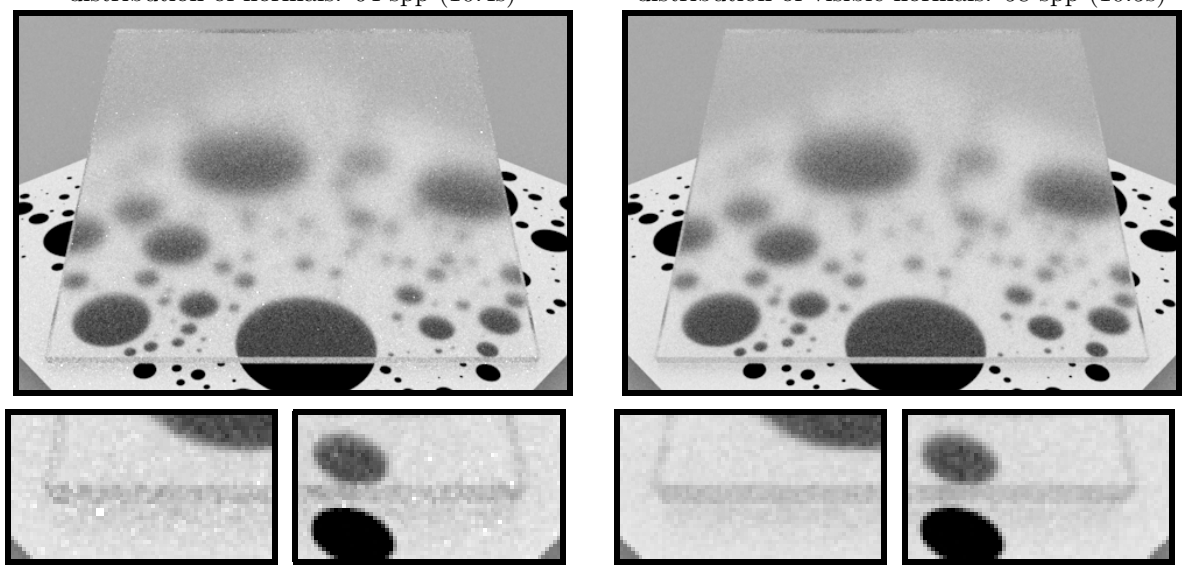

Figure 10: A dielectric glass plate ($n = 1.5$) with isotropic GGX roughness ($\alpha = 0.1$) on all faces (with the Smith masking function).

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (10.6s)

Our: BSDF Importance sampling using the distribution of visible normals. 58 spp (10.6s)

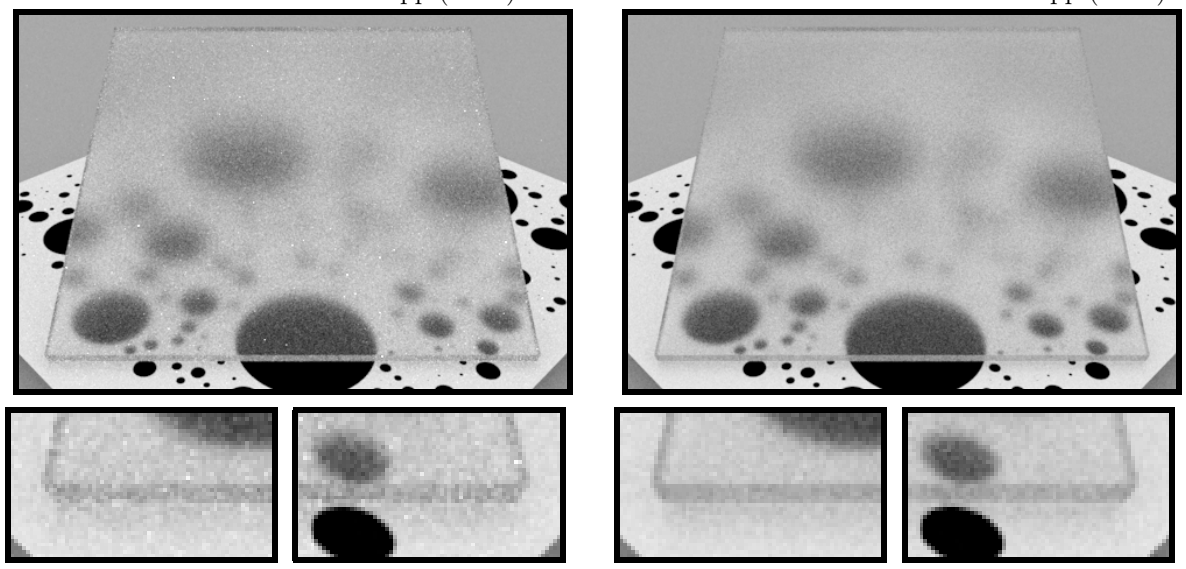

Figure 11: A dielectric glass plate ($n = 1.5$) with isotropic GGX roughness ($\alpha = 0.2$) on all faces (with the Smith masking function).

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (9.5s)

Our: BSDF Importance sampling using the distribution of visible normals. 56 spp (9.7s)

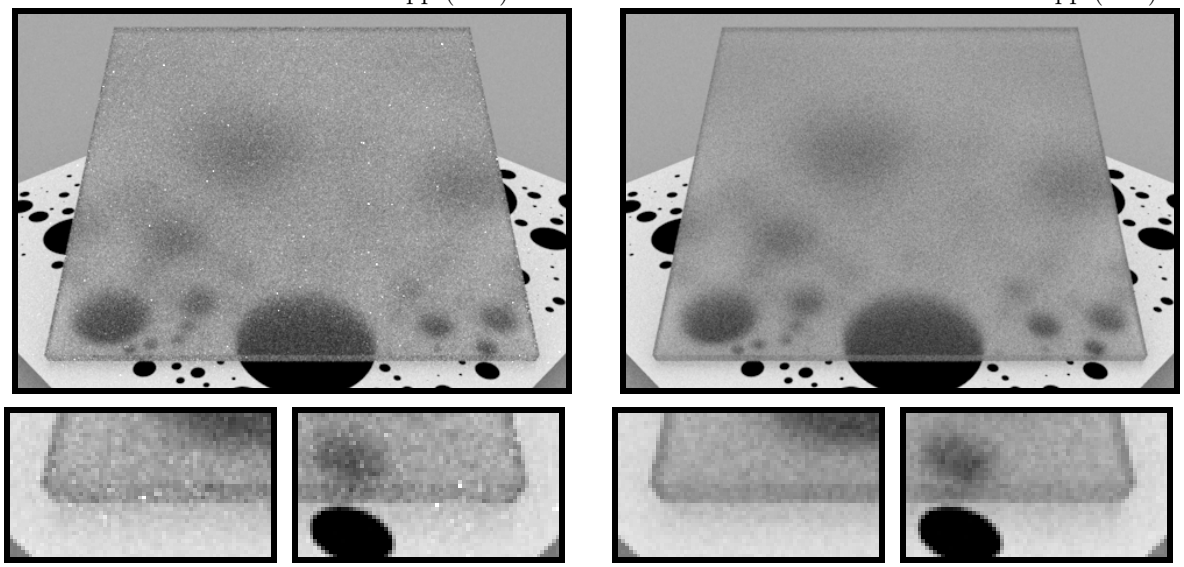

Figure 12: A dielectric glass plate ($n = 1.5$) with isotropic GGX roughness ($\alpha = 0.4$) on all faces (with the Smith masking function).

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (7.8s)

Our: BSDF Importance sampling using the distribution of visible normals. 52 spp (7.8s)

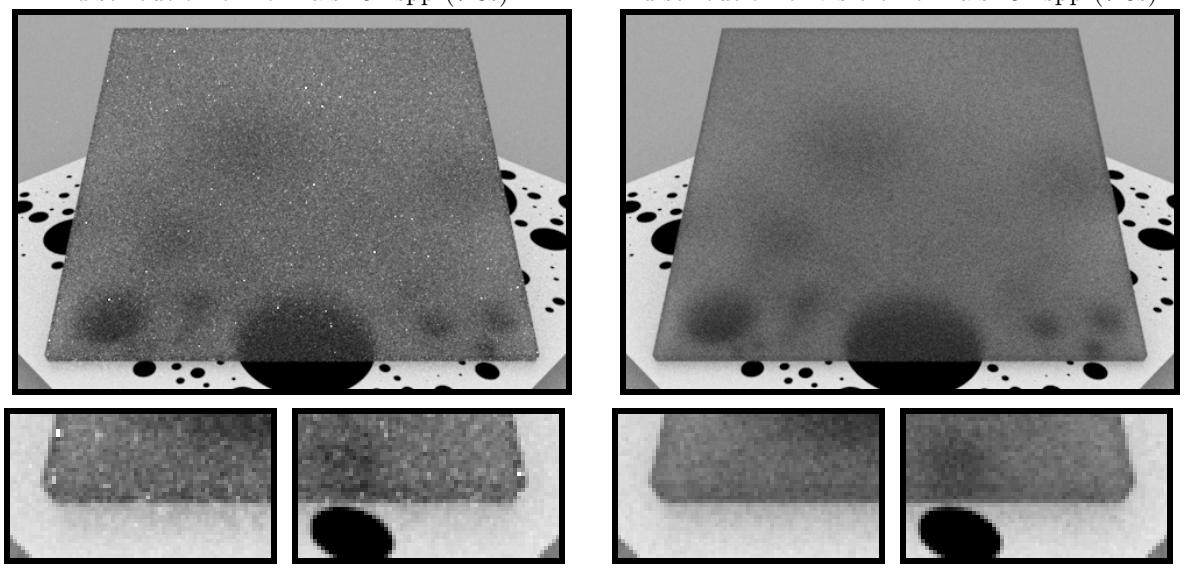

Figure 13: A dielectric glass plate ($n = 1.5$) with isotropic GGX roughness ($\alpha = 0.8$) on all faces (with the Smith masking function).

2.4 Anisotropic GGX

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (11.0s)

Our: BSDF Importance sampling using the distribution of visible normals. 58 spp (10.6s)

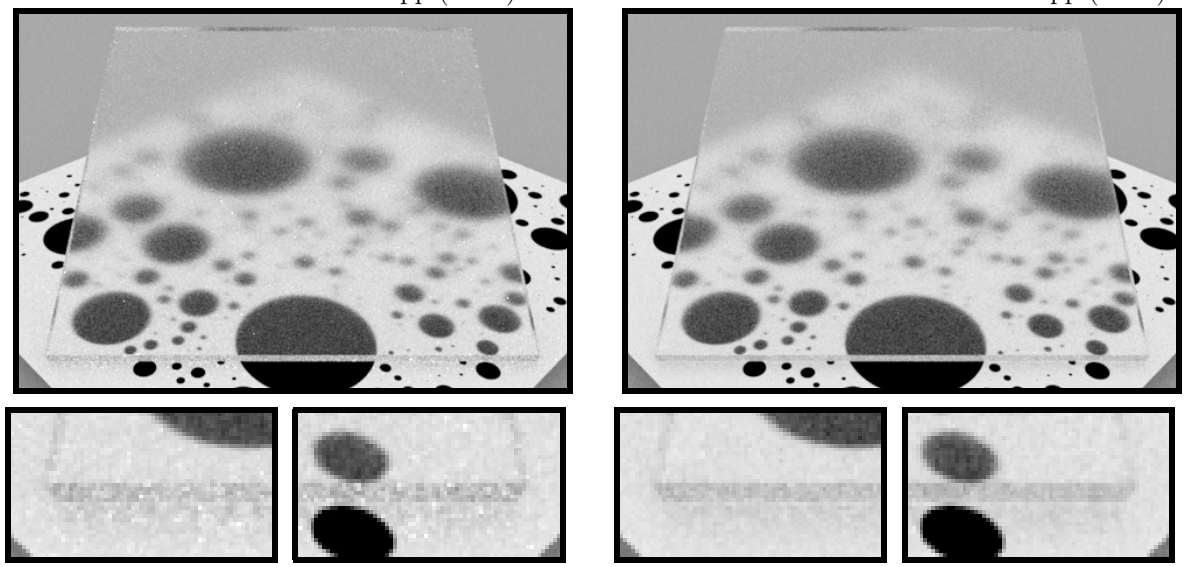

Figure 14: A dielectric glass plate ($n = 1.5$) with anisotropic GGX roughness ($\alpha_x = 0.05$, $\alpha_y = 0.10$) on all faces (with the Smith masking function).

Previous: BSDF Importance sampling using the distribution of normals. 64 spp (10.2s)

Our: BSDF Importance sampling using the distribution of visible normals. 52 spp (10.0s)

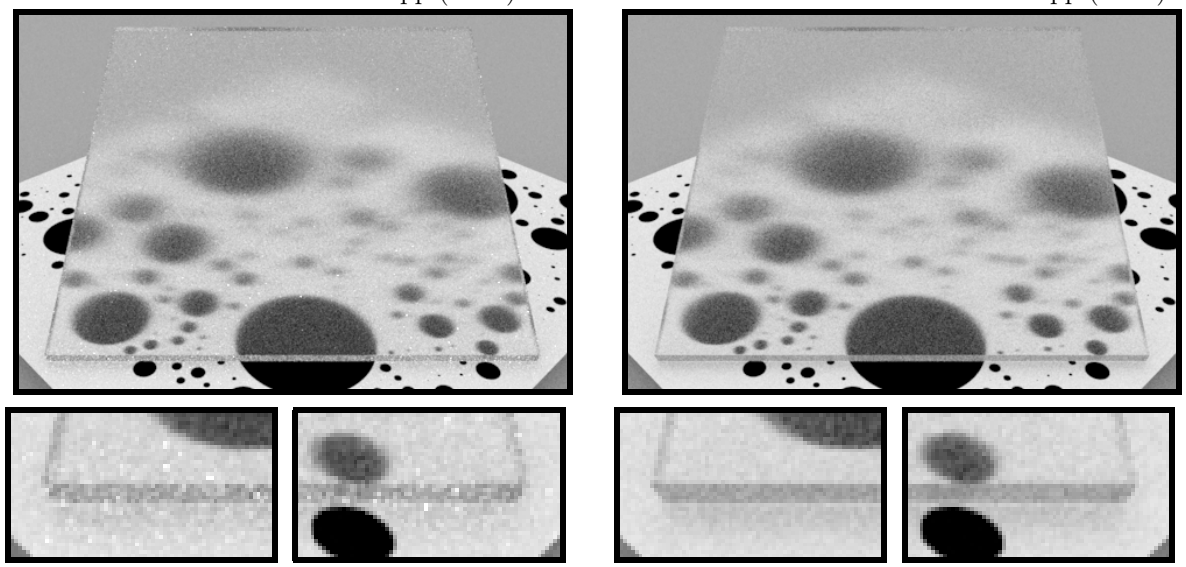

Figure 15: A dielectric glass plate ($n = 1.5$) with anisotropic GGX roughness ($\alpha_x = 0.05$, $\alpha_y = 0.20$) on all faces (with the Smith masking function).

3 Sample $(\omega_m \cdot \omega_g)\, D(\omega_m)$

Algorithm 3 of the paper shows how to sample PDF D_{ω_i} with the V-cavity model. The first step is to sample ω_m with PDF $(\omega_m \cdot \omega_g)D(\omega_m)$ before randomly swapping ω_m with ${\omega_m}'$. Walter et al. provide the sampling procedure for isotropic Beckmann and GGX distributions [WMLT07]. In this section, we recall how to generate micronormals samples with their anisotropic extensions [Hei14].

3.1 Generic Sampling Algorithm

Here, we provide the generic algorithm that can be used to sample micronormal PDF built from a slope distribution, i.e.:

$$(\omega_m \cdot \omega_g)D(\omega_m) = \frac{P^{22}(\tilde{m} = (x_{\tilde{m}}, y_{\tilde{m}}))}{(\omega_m \cdot \omega_g)^3},$$

where P^{22} is the slope PDF and the factor $\left\| \frac{\partial \tilde{m}}{\partial \omega_m} \right\| = \frac{1}{(\omega_m \cdot \omega_g)^3}$ is the Jacobian of the slope to normal transformation. Thus, if we sample $(x_{\tilde{m}}, y_{\tilde{m}})$ with PDF P^{22} and transform the slopes into a normal ω_m, we sample ω_m with PDF $(\omega_m \cdot \omega_g)\, D(\omega_m)$.

Algorithm 1 Sample $(\omega_m \cdot \omega_g)\, D(\omega_m)$ (generic)

$\begin{pmatrix} x_{\tilde{m}} \\ y_{\tilde{m}} \end{pmatrix} \leftarrow$ Sample $P^{22}(x_{\tilde{m}}, y_{\tilde{m}})$ ▷ sample slope PDF

$\omega_m \leftarrow \frac{(-x_{\tilde{m}}, -y_{\tilde{m}}, 1)}{\sqrt{x_{\tilde{m}}^2 + y_{\tilde{m}}^2 + 1}}$ ▷ compute normal

3.2 Sample Anisotropic Beckmann Distribution

The Beckmann distribution is built from the 2D Gaussian distribution of slopes

$$P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, \alpha_x, \alpha_y) = \frac{1}{\pi \alpha_x \alpha_y} \exp\left(- \left(\frac{x_{\tilde{m}}}{\alpha_x} \right)^2 - \left(\frac{y_{\tilde{m}}}{\alpha_y} \right)^2 \right),$$

which can be sampled with a Box-Muller transform:

$$x_{\tilde{m}} = \alpha_x \sqrt{-\log(\mathcal{U}_1)}\, \cos(2\pi \mathcal{U}_2),$$
$$y_{\tilde{m}} = \alpha_y \sqrt{-\log(\mathcal{U}_1)}\, \sin(2\pi \mathcal{U}_2).$$

By using this in Algorithm 1, we get Algorithm 2.

Algorithm 2 Sample $(\omega_m \cdot \omega_g)\, D(\omega_m)$ with for Beckmann distributions

$x_{\tilde{m}} = \alpha_x \sqrt{-\log(\mathcal{U}_1)}\, \cos(2\pi \mathcal{U}_2)$

$y_{\tilde{m}} = \alpha_y \sqrt{-\log(\mathcal{U}_1)}\, \sin(2\pi \mathcal{U}_2)$

$\omega_m \leftarrow \frac{(-x_{\tilde{m}}, -y_{\tilde{m}}, 1)}{\sqrt{x_{\tilde{m}}^2 + y_{\tilde{m}}^2 + 1}}$ ▷ compute normal

3.3 Sample Anisotropic GGX Distribution

The GGX distribution is built from the 2D distribution of slopes

$$P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, \alpha_x, \alpha_y) = \frac{1}{\pi \alpha_x \alpha_y \left(1 + \left(\frac{x_{\tilde{m}}}{\alpha_x} \right)^2 + \left(\frac{y_{\tilde{m}}}{\alpha_y} \right)^2 \right)^2},$$

which can be sampled by:

$$x_{\tilde{m}} = \alpha_x \frac{\sqrt{\mathcal{U}_1}}{\sqrt{1-\mathcal{U}_1}} \cos(2\pi\,\mathcal{U}_2),$$
$$y_{\tilde{m}} = \alpha_y \frac{\sqrt{\mathcal{U}_1}}{\sqrt{1-\mathcal{U}_1}} \sin(2\pi\,\mathcal{U}_2).$$

By using this in Algorithm 1, we get Algorithm 3.

Algorithm 3 Sample $(\omega_m \cdot \omega_g)\, D(\omega_m)$ for GGX distributions

$x_{\tilde{m}} = \alpha_x \frac{\sqrt{\mathcal{U}_1}}{\sqrt{1-\mathcal{U}_1}} \cos(2\pi\,\mathcal{U}_2)$
$y_{\tilde{m}} = \alpha_y \frac{\sqrt{\mathcal{U}_1}}{\sqrt{1-\mathcal{U}_1}} \sin(2\pi\,\mathcal{U}_2)$
$\omega_m \leftarrow \frac{(-x_{\tilde{m}}, -y_{\tilde{m}}, 1)}{\sqrt{x_{\tilde{m}}^2 + y_{\tilde{m}}^2 + 1}}$ ▷ compute normal

4 Importance Sampling Non-centered Distributions

In most cases, the normal distribution is *centered*, i.e. the average direction of the normals ω_m is the geometric normal ω_g. However, rendering with *non-centered distributions* is required when levels of detail are used. This is the case in LEADR mapping [DHI+13], where microfacet theory is extended to handle non-centered distributions. In this section, we show that the importance sampling strategies developed in this paper can be used for non-centered distributions and would thus perfectly fit in the LEADR mapping framework.

4.1 Definition

A non-centered distribution is a centered distribution shifted in slope space by a vector $(\bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}})$ and is noted:

$$P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y) = P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y),$$

where the centered distribution is:

$$P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y) = P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, \alpha_x, \alpha_y).$$

The vector $(\bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}})$ is the average slope of the distribution and is the slope associated to the *mesonormal*. For instance, the non-centered Gaussian slope distribution used in LEADR mapping is:

$$P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y) = \frac{1}{\pi\, \alpha_x\, \alpha_y} \exp\left(- \left(\frac{x_{\tilde{m}} - \bar{x}_{\tilde{m}}}{\alpha_x} \right)^2 - \left(\frac{y_{\tilde{m}} - \bar{y}_{\tilde{m}}}{\alpha_y} \right)^2 \right),$$

and with the transformation to normal space:

$$D(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y) = \frac{P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y)}{(\omega_m \cdot \omega_g)^4}.$$

4.2 Importance Sampling Non-centered Distributions

Algorithm 1 from this supplemental material can be extended to non-centered distributions by shifting the generated slopes before the transformation into normal. This is illustrated in Algorithm 4 where the additional shift operation is highlighted in red.

Algorithm 4 Sample $(\omega_m \cdot \omega_g) D(\omega_m)$ (generic)

$\begin{pmatrix} x_{\tilde{m}} \\ y_{\tilde{m}} \end{pmatrix} \leftarrow$ Sample $P^{22}(x_{\tilde{m}}, y_{\tilde{m}})$ ▷ sample slope PDF

$\begin{pmatrix} x_{\tilde{m}} \\ y_{\tilde{m}} \end{pmatrix} \leftarrow \begin{pmatrix} x_{\tilde{m}} + \bar{x}_{\tilde{m}} \\ y_{\tilde{m}} + \bar{y}_{\tilde{m}} \end{pmatrix}$ ▷ shift slope

$\omega_m \leftarrow \frac{(-x_{\tilde{m}}, -y_{\tilde{m}}, 1)}{\sqrt{x_{\tilde{m}}^2 + y_{\tilde{m}}^2 + 1}}$ ▷ compute normal

4.3 Importance Sampling Visible Non-centered Distributions with the Smith Model

With the Smith model, we have seen that the stretching invariance of the masking function Heitz [Hei14] holds also for the visible slope distribution, and we use it to handle varying roughness and anisotropy in our importance sampling scheme. In the same way, Heitz also shows that the masking function is shearing

invariant, and we show that this is also the case for the visible slope distribution and we use it to handle non-centered distributions.

The shearing operation, illustrated in Figure 16, is defined by:

$$\mathcal{T}^{\bar{x}_{\tilde{m}},\bar{y}_{\tilde{m}}}(\omega_i) = \frac{(x_i, y_i, z_i - \bar{x}_{\tilde{m}} x_i - \bar{y}_{\tilde{m}} y_i)}{\sqrt{x_i^2 + y_i^2 + (z_i - \bar{x}_{\tilde{m}} x_i - \bar{y}_{\tilde{m}} y_i)^2}},$$

and the shearing invariance of the visible slope distribution is written:

$$P^{22}_{\omega_i}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y) = P^{22}_{\mathcal{T}^{\bar{x}_{\tilde{m}},\bar{y}_{\tilde{m}}}(\omega_i)}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{x}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y). \tag{1}$$

As a result, we can use the same precomputed data for non-centered anisotropic configurations:

$$\begin{aligned} P^{22}_{\omega_i}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y) &= P^{22}_{\mathcal{T}^{\bar{x}_{\tilde{m}},\bar{y}_{\tilde{m}}}(\omega_i)}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{x}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y) \\ &= \frac{1}{\alpha_x \alpha_y} P^{22}_{\mathcal{S}^{\alpha_x,\alpha_y}(\mathcal{T}^{\bar{x}_{\tilde{m}},\bar{y}_{\tilde{m}}}(\omega_i))}\left(\frac{x_{\tilde{m}} - \bar{x}_{\tilde{m}}}{\alpha_x}, \frac{y_{\tilde{m}} - \bar{y}_{\tilde{m}}}{\alpha_y}, 0, 0, 1, 1\right). \end{aligned}$$

This formula shows that $P^{22}_{\omega_i}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y)$ can be sampled if a sampling algorithm is available for $P^{22}_{\omega_i}(x_{\tilde{m}}, y_{\tilde{m}}, 0, 0, 1, 1)$. By using this, we transform Algorithm 4 from the paper into Algorithm 5 from this supplemental where the changes are highlighted in red.

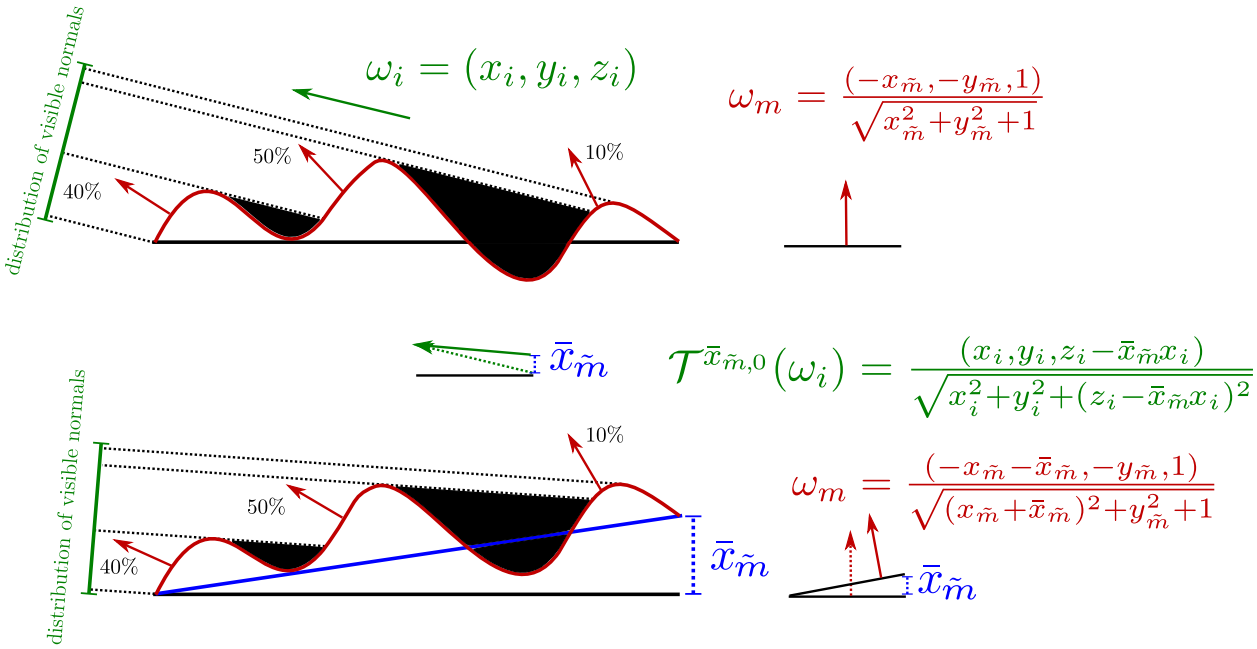

Figure 16: Shearing the configuration.

Algorithm 5 Sample $D_{\omega_i}(\omega_m)$ (Smith profile)

$\omega_i' \leftarrow \mathcal{S}^{\alpha_x,\alpha_y}\left(\mathcal{T}^{\bar{x}_{\tilde{m}},\bar{y}_{\tilde{m}}}\left(\omega_i\right)\right)$ ▷ 1. shear and stretch

$x_{\tilde{m}} \leftarrow T_{x_{\tilde{m}}}[\theta_i', \mathcal{U}_1]$ ▷ 2.a. Sample $P^{22}_{\omega_i}(x_{\tilde{m}}, 1, 1)$

$y_{\tilde{m}} \leftarrow T_{y_{\tilde{m}}}[x_{\tilde{m}}, \mathcal{U}_2]$ ▷ 2.b. Sample $P^{22}_{\omega_i}(y_{\tilde{m}}|x_{\tilde{m}}, 1, 1)$

$\begin{pmatrix} x_{\tilde{m}} \\ y_{\tilde{m}} \end{pmatrix} \leftarrow \begin{pmatrix} \cos\phi_i' & -\sin\phi_i' \\ \sin\phi_i' & \cos\phi_i' \end{pmatrix} \begin{pmatrix} x_{\tilde{m}} \\ y_{\tilde{m}} \end{pmatrix}$ ▷ 3. rotate

$\begin{pmatrix} x_{\tilde{m}} \\ y_{\tilde{m}} \end{pmatrix} \leftarrow \begin{pmatrix} \alpha_x x_{\tilde{m}} \\ \alpha_y y_{\tilde{m}} \end{pmatrix}$ ▷ 4.a unstretch

$\begin{pmatrix} x_{\tilde{m}} \\ y_{\tilde{m}} \end{pmatrix} \leftarrow \begin{pmatrix} x_{\tilde{m}} + \bar{x}_{\tilde{m}} \\ y_{\tilde{m}} + \bar{y}_{\tilde{m}} \end{pmatrix}$ ▷ 4.b unshear

$\omega_m \leftarrow \frac{(-x_{\tilde{m}}, -y_{\tilde{m}}, 1)}{\sqrt{x_{\tilde{m}}^2 + y_{\tilde{m}}^2 + 1}}$ ▷ 5. compute normal

Proof of Equation (1) (shearing invariance):

$$
\begin{aligned}
&P^{22}_{\omega_i}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y) \\
&= \frac{\chi^+(-x_i x_{\tilde{m}} - y_i y_{\tilde{m}} + z_i)\,(-x_i x_{\tilde{m}} - y_i y_{\tilde{m}} + z_i)\,P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y)}{\int\int \chi^+(-x_i x_{\tilde{m}} - y_i y_{\tilde{m}} + z_i)\,(-x_i x_{\tilde{m}} - y_i y_{\tilde{m}} + z_i)\,P^{22}(x_{\tilde{m}}, y_{\tilde{m}}, \bar{x}_{\tilde{m}}, \bar{y}_{\tilde{m}}, \alpha_x, \alpha_y)\,dx_{\tilde{m}}dy_{\tilde{m}}} \\
&= \frac{\chi^+(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}} + \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}} + \bar{y}_{\tilde{m}}) + z_i)\,(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}} + \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}} + \bar{y}_{\tilde{m}}) + z_i)\,P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y)}{\int\int \chi^+(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}} + \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}} + \bar{y}_{\tilde{m}}) + z_i)\,(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}} + \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}} + \bar{y}_{\tilde{m}}) + z_i)\,P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y)\,dx_{\tilde{m}}dy_{\tilde{m}}} \\
&= \frac{\chi^+(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y)}{\int\int \chi^+(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y)\,dx_{\tilde{m}}dy_{\tilde{m}}} \\
&= \frac{\sqrt{x_i^2 + y_i^2 + (z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})^2}}{\sqrt{x_i^2 + y_i^2 + (z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})^2}} \times \\
&\quad \frac{\chi^+(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y)}{\int\int \chi^+(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,(-x_i(x_{\tilde{m}} - \bar{x}_{\tilde{m}}) - y_i(y_{\tilde{m}} - \bar{y}_{\tilde{m}}) + z_i - x_i\bar{x}_{\tilde{m}} - y_i\bar{y}_{\tilde{m}})\,P^{22}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{y}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y)\,dx_{\tilde{m}}dy_{\tilde{m}}} \\
&= P^{22}_{\mathcal{T}^{\bar{x}_{\tilde{m}},\bar{y}_{\tilde{m}}}(\omega_i)}(x_{\tilde{m}} - \bar{x}_{\tilde{m}}, y_{\tilde{m}} - \bar{x}_{\tilde{m}}, 0, 0, \alpha_x, \alpha_y)
\end{aligned}
$$

5 Sample $D_{\omega_i}(\omega_m)$ with the Smith Model (C++ Implementation with Precomputed Data)

In this section, we describe a C++ implementation of the paper's Algorithm 4 and of the associated precomputations steps. We implement a virtual class **VNDFSampler** (Visible Normal Distribution Function Sampler). The API contains two public methods:

- method **init**() is called after construction and do the precomputation of tables $T_{x_{\tilde{m}}}$ and $T_{y_{\tilde{m}}}$. It calls methods **init_Tx()** which precomputes $T_{x_{\tilde{m}}}$, and method **init_Ty()**, which precomputes $T_{y_{\tilde{m}}}$.
- method **sample**() takes as arguments the incident direction ω_i, anisotropic roughness α_x and α_y, and two uniform random numbers $\mathcal{U}_1$ and $\mathcal{U}_2$. It generates a microfacet normal with PDF D_{ω_i}.

```
class VNDFSampler
{
public:
        // precompute Tx and Ty
        void init();

        // generate a view dependent normal
        void sample(
                        // input
                        const double omega_i[3], // incident direction
                        const double alpha_x, const double alpha_y, // anisotropic roughness
                        const double U1, const double U2, // random numbers
                        // output
                        double omega_m[3] // normal
                        );

        VNDFSampler();
        ~VNDFSampler();

protected:
// methods implemented by child class
        // 2D slope distribution (alpha = 1.0)
        virtual double P22(const double slope_x, const double slope_y) = 0;
        // maximal slope amplitude (range that contains 99.99% of the distribution)
        virtual double slope_max() = 0;

private:
// inverse slope_x CDF for a given view angle theta
        double Tx(const double theta, const double U1);
// inverse slope_y CDF for a given slope_x
        double Ty(const double slope_x, const double U2);

// data
        void init_Tx();
        void init_Ty();
        double * data_Tx;
        double * data_Ty;
};


VNDFSampler::VNDFSampler()
{
        data_Tx = 0;
        data_Ty = 0;
}

VNDFSampler::~VNDFSampler()
{
        if(data_Tx) delete [] data_Tx;
        if(data_Ty) delete [] data_Ty;
}

void VNDFSampler::init()
{
        init_Tx();
        init_Ty();
}
```

A child class must implement the protected virtual methods:

- **P22()**, the isotropic slope distribution for $\alpha = 1$. This method is called in methods **init_Tx()** and **init_Ty()**, where the normalization factor of $P^{22}_{\omega_i}$ is computed numerically. Thus, method **P22()** does not need to return the value of the normalized distribution in our implementation.

- **slope_max()**, the maximal slope amplitude of $P^{22}_{\omega_i}$ (wider than the maximal slope amplitude of P^{22} since the view dependence tends to shift the distribution).

```
class VNDFSamplerBeckmann : public VNDFSampler
{
protected:
	// 2D slope distribution (alpha = 1.0)
	virtual double P22(const double slope_x, const double slope_y)
	{
		return exp( - (slope_x*slope_x + slope_y*slope_y) );
	}

	// maximal slope amplitude (range that contains 99.99% of the distribution)
	virtual double slope_max()
	{
		return 6.0;
	}
};

class VNDFSamplerGGX : public VNDFSampler
{
protected:
	// 2D slope distribution (alpha = 1.0)
	virtual double P22(const double slope_x, const double slope_y)
	{
		double tmp = 1.0 + slope_x*slope_x + slope_y*slope_y;
		return 1.0 / (tmp*tmp);
	}

	// maximal slope amplitude (range that contains 99.99% of the distribution)
	virtual double slope_max()
	{
		return 16.0;
	}
};
```

Method **sample()** implements Algorithm 4 from the paper.

```
void VNDFSampler::sample(
        // input
        const double omega_i[3], // incident direction
        const double alpha_x, const double alpha_y, // anisotropic roughness
        const double U1, const double U2, // random numbers
        // output
        double omega_m[3]) // normal
{
// 1. stretch omega_i
        double omega_i_[3];
        omega_i_[0] = alpha_x * omega_i[0];
        omega_i_[1] = alpha_y * omega_i[1];
        omega_i_[2] = omega_i[2];
        // normalize
        double inv_omega_i = 1.0 / sqrt(omega_i_[0]*omega_i_[0] + omega_i_[1]*omega_i_[1] + omega_i_[2]*omega_i_[2]);
        omega_i_[0] *= inv_omega_i;
        omega_i_[1] *= inv_omega_i;
        omega_i_[2] *= inv_omega_i;
        // get polar coordinates of omega_i_
        double theta_ = 0.0;
        double phi_ = 0.0;
        if (omega_i_[2] < 0.99999)
        {
                theta_ = acos(omega_i_[2]);
                phi_ = atan2(omega_i_[1], omega_i_[0]);
        }

// 2. sample P22_{omega_i}(x_slope, y_slope, 1, 1)
        double slope_x = Tx(theta_, U1);
        double slope_y = Ty(slope_x, U2);

// 3. rotate
        double tmp = cos(phi_)*slope_x - sin(phi_)*slope_y;
        slope_y = sin(phi_)*slope_x + cos(phi_)*slope_y;
        slope_x = tmp;

// 4. unstretch
        slope_x = alpha_x * slope_x;
        slope_y = alpha_y * slope_y;

// 5. compute normal
        double inv_omega_m = sqrt(slope_x*slope_x + slope_y*slope_y + 1.0);
        omega_m[0] = -slope_x/inv_omega_m;
        omega_m[1] = -slope_y/inv_omega_m;
        omega_m[2] = 1.0/inv_omega_m;
}
```

Constant **DATA_T_SIZE** specifies the width of the precomputed 2D tables $T_{x\tilde{m}}$ and $T_{y\tilde{m}}$. Constant **DATA_TMP_SIZE** specifies the size of the 1D temporary tables.

```
static const int DATA_T_SIZE = 1024;
static const int DATA_TMP_SIZE = 2048;

void VNDFSampler::init_Tx()
{
	// allocate Tx
	data_Tx = new double[DATA_T_SIZE*DATA_T_SIZE];

	// allocate temporary data
	double * slope_x   = new double[DATA_TMP_SIZE];
	double * CDF_P22_omega_i = new double[DATA_TMP_SIZE];

	// loop over incident directions
	for(int index_theta=0 ; index_theta < DATA_T_SIZE ; ++index_theta)
	{
		// incident vector
		const double theta = 0.5 * M_PI * index_theta / (DATA_T_SIZE-1.0);
		const double sin_theta = sin(theta); // sin theta_i = x_i
		const double cos_theta = cos(theta); // cos theta_i = z_i

		// for a given incident vector
		// integrate P22_{omega_i}(x_slope, 1, 1), Eq. (10)
		slope_x[0] = -slope_max();
		CDF_P22_omega_i[0] = 0;
		for(int index_slope_x=1 ; index_slope_x<DATA_TMP_SIZE ; ++index_slope_x)
		{
			// slope_x
			slope_x[index_slope_x] = -slope_max() + 2.0 * slope_max() * index_slope_x/(DATA_TMP_SIZE-1.0);

			// dot product with incident vector
			double dot_product = max(0.0, -slope_x[index_slope_x]*sin_theta + cos_theta);

			// marginalize P22_{omega_i}(x_slope, 1, 1), Eq. (10)
			double P22_omega_i = 0.0;
			for(int j=0 ; j<100 ; ++j)
			{
				double slope_y = -slope_max() + 2.0 * slope_max() * j/99.0;
				P22_omega_i += dot_product * P22(slope_x[index_slope_x], slope_y);
			}

			// CDF of P22_{omega_i}(x_slope, 1, 1), Eq. (10)
			CDF_P22_omega_i[index_slope_x] = CDF_P22_omega_i[index_slope_x-1] + P22_omega_i;
		}

		// renormalize CDF_P22_omega_i
		for(int index_slope_x=1 ; index_slope_x<DATA_TMP_SIZE ; ++index_slope_x)
			CDF_P22_omega_i[index_slope_x] /= CDF_P22_omega_i[DATA_TMP_SIZE-1];

		// loop over random number U1
		int index_slope_x=0;
		for(int index_U=0 ; index_U < DATA_T_SIZE ; ++index_U)
		{
			const double U = 0.0000001 + 0.9999998 * index_U / (double)(DATA_T_SIZE-1);

			// inverse CDF_P22_omega_i, solve Eq.(11)
			while( CDF_P22_omega_i[index_slope_x] <= U )
				++index_slope_x;

			const double interp =
				(CDF_P22_omega_i[index_slope_x]-U) /
				(CDF_P22_omega_i[index_slope_x] - CDF_P22_omega_i[index_slope_x-1]);

			// store value
			data_Tx[index_U + index_theta*DATA_T_SIZE] =
				interp * slope_x[index_slope_x-1]
				+ (1.0f-interp) * slope_x[index_slope_x];
		}
	}

	// free temporary data
	delete [] slope_x;
	delete [] CDF_P22_omega_i;
}
```

```
void VNDFSampler::init_Ty()
{
	// allocate Ty
	data_Ty = new double[DATA_T_SIZE*DATA_T_SIZE];

	// allocate temporary data
	double * slope_y  = new double[DATA_TMP_SIZE];
	double * CDF_P22y = new double[DATA_TMP_SIZE];

	// loop over slope_x
	for(int index_slope_x=0 ; index_slope_x < DATA_T_SIZE ; ++index_slope_x)
	{
		// slope_x
		const double slope_x = -slope_max() + 2.0 * slope_max() * index_slope_x / (double)(DATA_T_SIZE-1);

		// CDF of P22y, Eq.(13)
		slope_y[0] = -slope_max();
		CDF_P22y[0] = 0;
		for(int index_slope_y=1 ; index_slope_y<DATA_TMP_SIZE ; ++index_slope_y)
		{
			slope_y[index_slope_y] = -slope_max() + 2.0 * slope_max() * index_slope_y/(DATA_TMP_SIZE-1.0);
			CDF_P22y[index_slope_y] = CDF_P22y[index_slope_y-1] + P22(slope_x, slope_y[index_slope_y]);
		}

		// renormalize CDF_P22y
		for(int index_slope_y=1 ; index_slope_y<DATA_TMP_SIZE ; ++index_slope_y)
		{
			CDF_P22y[index_slope_y] /= CDF_P22y[DATA_TMP_SIZE-1];
		}

		// loop over random number U2
		int index_slope_y=0;
		for(int index_U=0 ; index_U < DATA_T_SIZE ; ++index_U)
		{
			double U = 0.0000001 + 0.9999998 * index_U / (double)(DATA_T_SIZE-1);

			// inverse CDF_P22y, solve Eq.(13)
			while( CDF_P22y[index_slope_y] <= U )
				++index_slope_y;

			const double interp =
				(CDF_P22y[index_slope_y]-U) /
				(CDF_P22y[index_slope_y] - CDF_P22y[index_slope_y-1]);

			// store value
			data_Ty[index_U + index_slope_x*DATA_T_SIZE] =
				interp * slope_y[index_slope_y-1] +
				(1.0f-interp) * slope_y[index_slope_y];
		}
	}

	// free temporary data
	delete [] slope_y;
	delete [] CDF_P22y;
}
```

We use linear interpolation to access the precomputed data.

```
double VNDFSampler::Tx(const double theta, const double U)
{
        // indices and interpolation weight in dimension 1 (theta)
        const double t = theta/(0.5*M_PI) * (double)(DATA_T_SIZE-1);
        const int index_theta1 = max(0, (int) floor(t));
        const int index_theta2 = min(index_theta1+1, DATA_T_SIZE-1);
        const double interp_theta = t - (double)index_theta1;

        // indices and interpolation weight in dimension 2 (U1)
        const double u = U * (double)(DATA_T_SIZE-1);
        const int index_u1 = max(0, (int) floor(u));
        const int index_u2 = min(index_u1+1, DATA_T_SIZE-1);
        const double interp_u = u - (double)index_u1;

        const double slope_x = (1.0f-interp_theta) * (1.0f-interp_u) * data_Tx[index_u1 + index_theta1*DATA_T_SIZE]
                             + (1.0f-interp_theta) * interp_u * data_Tx[index_u2 + index_theta1*DATA_T_SIZE]
                             + interp_theta * (1.0f-interp_u) * data_Tx[index_u1 + index_theta2*DATA_T_SIZE]
                             + interp_theta * interp_u * data_Tx[index_u2 + index_theta2*DATA_T_SIZE];

        return slope_x;
}

double VNDFSampler::Ty(const double slope_x, const double U)
{
        // indices and interpolation weight in dimension 1 (slope_x)
        const double x = (slope_x+slope_max())/(2.0*slope_max()) * (double)(DATA_T_SIZE-1);
        const int index_x1 = max(0, (int) floor(x));
        const int index_x2 = min(index_x1+1, DATA_T_SIZE-1);
        const double interp_x = x - (double)index_x1;

        // indices and interpolation weight in dimension 2 (U2)
        const double u = U * (double)(DATA_T_SIZE-1);
        const int index_u1 = max(0, (int) floor(u));
        const int index_u2 = min(index_u1+1, DATA_T_SIZE-1);
        const double interp_u = u - (double)index_u1;

        const double slope_y = (1.0f-interp_x) * (1.0f-interp_u) * data_Ty[index_u1 + index_x1*DATA_T_SIZE]
                             + (1.0f-interp_x) * interp_u * data_Ty[index_u2 + index_x1*DATA_T_SIZE]
                             + interp_x * (1.0f-interp_u) * data_Ty[index_u1 + index_x2*DATA_T_SIZE]
                             + interp_x * interp_u * data_Ty[index_u2 + index_x2*DATA_T_SIZE];

        return slope_y;
}
```

References

[DHI+13] Jonathan Dupuy, Eric Heitz, Jean-Claude Iehl, Pierre Poulin, Fabrice Neyret, and Victor Ostromoukhov. Linear efficient antialiased displacement and reflectance mapping. *ACM Trans. Graph.*, 32(6):211:1–211:11, November 2013.

[Hei14] E. Heitz. Understanding the masking-shadowing function in microfacet-based brdfs. *Journal of Computer Graphics Techniques (JCGT)*, 3(2), 2014. To appear.

[WMLT07] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance. Microfacet models for refraction through rough surfaces. In *Proc. Eurographics Symposium on Rendering*, EGSR'07, pages 195–206, 2007.