Multiple-Scattering Microfacet BSDFs with the Smith Model

Supplemental Material: Implementation

Eric Heitz[1] Johannes Hanika[1] Eugene d'Eon[2] Carsten Dachsbacher[1]

[1]Karlsruhe Institute of Technology [2]8i

Abstract

In this document, we describe an implementation of our multiple scattering microfacet BSDF models. We also propose a series of tests that are useful for validating an implementation of the models at intermediate milestones and to debug the implementation.

Remarks

- **The presented implementation is not optimized.** We designed it with an emphasis on modularity and readability instead of performance. We used this implementation to produce the plots presented in the article and the supplemental material "Comparison to Measured data". Note that images in the article and timings have been obtained from an optimized implementation.

- **This implementation is dedicated to non-absorptive materials that are** 100% **energy conserving.** This property is useful for validation and debugging because we can verify that several of the integrals evaluate to exactly 1. More specifically, this means that the conductor material has a constant Fresnel term $F = 1$ and the diffuse material albedo is $a = 1$. Implementing absorptive conductor and diffuse materials can be easily achieved by adding a Fresnel term and an albedo to their associated phase functions.

Contents

1 Height Distributions

1.1 API

A height distribution implements the following functions

- the PDF P^1
- the CDF C^1
- the inverse CDF C^{-1}

```
/* API */
class MicrosurfaceHeight
{
public:
        // height PDF
        virtual float P1(const float h) const=0;
        // height CDF
        virtual float C1(const float h) const=0;
        // inverse of the height CDF
        virtual float invC1(const float U) const=0;
};
```

In the following we propose the uniform or the Gaussian height distribution. Note that the BSDFs resulting from the Smith model are independent of the choice of the height distribution. Hence, choosing one or the other does not impact the BSDF.

1.2 Uniform Height Distribution

Uniform height distribution in $[-1, 1]$.

```
/* Uniform height distribution in [-1, 1] */
class MicrosurfaceHeightUniform : public MicrosurfaceHeight
{
public:
        // height PDF
        virtual float P1(const float h) const;
        // height CDF
        virtual float C1(const float h) const;
        // inverse of the height CDF
        virtual float invC1(const float U) const;
};
```

Height PDF

$$P^1(h) = \begin{cases} \frac{1}{2} & \text{if } h \in [-1, 1] \\ 0 & \text{otherwise .} \end{cases}$$

```
float MicrosurfaceHeightUniform::P1(const float h) const
{
        const float value = (h >= -1.0f && h <= 1.0f) ? 0.5f : 0.0f;
        return value;
}
```

Height CDF

$$C^1(h) = \begin{cases} 0 & \text{if } h < -1, \\ \frac{h+1}{2} & \text{if } h \in [-1, 1], \\ 1 & \text{if } h > 1. \end{cases}$$

```
float MicrosurfaceHeightUniform::C1(const float h) const
{
        const float value = std::min(1.0f, std::max(0.0f, 0.5f*(h+1.0f)));
        return value;
}
```

Height inverse CDF

$$C^{-1}(\mathcal{U}) = 2\,\mathcal{U} - 1$$

```
float MicrosurfaceHeightUniform::invC1(const float U) const
{
        const float h = std::max(-1.0f, std::min(1.0f, 2.0f*U-1.0f));
        return h;
}
```

1.3 Gaussian Height Distribution

Gaussian height distribution $\mathcal{N}(0,1)$.

```
/* Gaussian height distribution N(0,1) */
class MicrosurfaceHeightGaussian : public MicrosurfaceHeight
{
public:
        // height PDF
        virtual float P1(const float h) const;
        // height CDF
        virtual float C1(const float h) const;
        // inverse of the height CDF
        virtual float invC1(const float U) const;
};
```

Height PDF

$$P^1(h) = \frac{1}{\sqrt{2\,\pi}} \exp\left(-\frac{h^2}{2}\right)$$

```
float MicrosurfaceHeightGaussian::P1(const float h) const
{
        const float value = INV_SQRT_2_M_PI * expf(-0.5f * h*h);
        return value;
}
```

Height CDF

$$C^1(h) = \frac{1}{2} + \frac{1}{2}\mathrm{erf}\left(\frac{h}{\sqrt{2}}\right)$$

```
float MicrosurfaceHeightGaussian::C1(const float h) const
{
        const float value = 0.5f + 0.5f * (float)erf(INV_SQRT_2*h);
        return value;
}
```

Height inverse CDF

$$C^{-1}(\mathcal{U}) = \sqrt{2}\ \mathrm{erf}^{-1}(2\,\mathcal{U} - 1)$$

```
float MicrosurfaceHeightGaussian::invC1(const float U) const
{
        const float h = SQRT_2 * erfinv(2.0f*U - 1.0f);
        return h;
}
```

2 Slope Distributions

2.1 API

```
/* API */
class MicrosurfaceSlope
{
public:
        MicrosurfaceSlope(const float alpha_x=1.0f, const float alpha_y=1.0f)
                : m_alpha_x(alpha_x), m_alpha_y(alpha_y)
        {}

public:
        // roughness
        const float m_alpha_x, m_alpha_y;
        // projected roughness in wi
        float alpha_i(const vec3& wi) const;

public:
        // distribution of normals (NDF)
        float D(const vec3& wm) const;
        // distribution of visible normals (VNDF)
        float D_wi(const vec3& wi, const vec3& wm) const;
        // sample the VNDF
        vec3 sampleD_wi(const vec3& wi, const float U1, const float U2) const;

public:
        // distribution of slopes
        virtual float P22(const float slope_x, const float slope_y) const=0;
        // Smith's Lambda function
        virtual float Lambda(const vec3& wi) const=0;
        // projected area towards incident direction
        virtual float projectedArea(const vec3& wi) const=0;
        // sample the distribution of visible slopes with alpha=1.0
        virtual vec2 sampleP22_11(const float theta_i, const float U1, const float U2) const=0;
};
```

Roughness The slope distribution is parameterized by the roughnesses α_x and α_y.

Projected roughness The projected roughness in the incident direction is

$$\alpha_i = \sqrt{\cos^2 \phi_i \, \alpha_x^2 + \sin^2 \phi_i \, \alpha_y^2}.$$

```
float MicrosurfaceSlope::alpha_i(const vec3& wi) const
{
        const float invSinTheta2 = 1.0f / (1.0f - wi.z*wi.z);
        const float cosPhi2 = wi.x*wi.x*invSinTheta2;
        const float sinPhi2 = wi.y*wi.y*invSinTheta2;
        const float alpha_i = sqrtf( cosPhi2*m_alpha_x*m_alpha_x + sinPhi2*m_alpha_y*m_alpha_y );
        return alpha_i;
}
```

The distribution of normals The distribution of normals is

$$D(\boldsymbol{\omega}_m) = \frac{P^{22}(x_{\tilde{m}}, y_{\tilde{m}})}{\cos^4 \theta_m},$$

where the slope distribution P^{22} is implemented by the derived class.

```
float MicrosurfaceSlope::D(const vec3& wm) const {
	if( wm.z <= 0.0f)
		return 0.0f;

	// slope of wm
	const float slope_x = -wm.x/wm.z;
	const float slope_y = -wm.y/wm.z;

	// value
	const float value = P22(slope_x, slope_y) / (wm.z*wm.z*wm.z*wm.z);
	return value;
}
```

The distribution of visible normals The distribution of visible normals is

$$D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) = \frac{\langle \boldsymbol{\omega}_i, \boldsymbol{\omega}_m \rangle \, D(\boldsymbol{\omega}_m)}{\int_\Omega \langle \boldsymbol{\omega}_i, \boldsymbol{\omega}_m \rangle \, D(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m}$$

The projected area of the denominator is implemented by the derived class.

```
float MicrosurfaceSlope::D_wi(const vec3& wi, const vec3& wm) const {
	if( wm.z <= 0.0f)
		return 0.0f;

	// normalization coefficient
	const float projectedarea = projectedArea(wi);
	if(projectedarea == 0)
		return 0;
	const float c = 1.0f / projectedarea;

	// value
	const float value = c * std::max(0.0f, dot(wi, wm)) * D(wm);
	return value;
}
```

Importance Sampling the Distribution of Visible Normals We use the technique of Heitz and d'Eon [HD14]. First, the configuration is stretched to match an isotropic roughness of $\alpha = 1$. Then, we use the method **sampleP22_11** provided by the derived class to sample a slope in this configuration. Finally, the inverse stretch transformation is applied to the slope and it is transformed into a normal.

```
vec3 MicrosurfaceSlope::sampleD_wi(const vec3& wi, const float U1, const float U2) const {

	// stretch to match configuration with alpha=1.0
	const vec3 wi_11 = normalize(vec3(m_alpha_x * wi.x, m_alpha_y * wi.y, wi.z));

	// sample visible slope with alpha=1.0
	vec2 slope_11 = sampleP22_11(acosf(wi_11.z), U1, U2);

	// align with view direction
	const float phi = atan2(wi_11.y, wi_11.x);
	vec2 slope(cosf(phi)*slope_11.x - sinf(phi)*slope_11.y, sinf(phi)*slope_11.x + cos(phi)*slope_11.y);

	// stretch back
	slope.x *= m_alpha_x;
	slope.y *= m_alpha_y;

	// if numerical instability
	if( (slope.x != slope.x) || !IsFiniteNumber(slope.x) )
	{
		if(wi.z > 0) return vec3(0.0f,0.0f,1.0f);
		else return normalize(vec3(wi.x, wi.y, 0.0f));
	}
```

```
        // compute normal
        const vec3 wm = normalize(vec3(-slope.x, -slope.y, 1.0f));
        return wm;
}
```

2.2 Beckmann Slope Distribution

```
/* Beckmann slope distribution */
class MicrosurfaceSlopeBeckmann : public MicrosurfaceSlope
{
public:
        MicrosurfaceSlopeBeckmann(const float alpha_x=1.0f, const float alpha_y=1.0f)
                : MicrosurfaceSlope(alpha_x, alpha_y)
        {}

        // distribution of slopes
        virtual float P22(const float slope_x, const float slope_y) const;
        // Smith's Lambda function
        virtual float Lambda(const vec3& wi) const;
        // projected area towards incident direction
        virtual float projectedArea(const vec3& wi) const;
        // sample the distribution of visible slopes with alpha=1.0
        virtual vec2 sampleP22_11(const float theta_i, const float U1, const float U2) const;
};
```

Slope PDF

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

```
float MicrosurfaceSlopeBeckmann::P22(const float slope_x, const float slope_y) const
{
        const float value = 1.0f / (M_PI * m_alpha_x * m_alpha_y) * expf(-slope_x*slope_x/(m_alpha_x*m_alpha_x) - slope_y*slope_y/(m_alpha_y*m_alpha_y) );
        return value;
}
```

Smith Λ Function

$$\Lambda(\boldsymbol{\omega}_i) = \frac{1}{2}(\mathrm{erf}(a) - 1) + \frac{1}{2\, a\, \sqrt{\pi}}\, \exp\left(-a^2\right)$$

$$a = \frac{1}{\alpha_i\, \tan\theta_i}$$

```
float MicrosurfaceSlopeBeckmann::Lambda(const vec3& wi) const
{
        if(wi.z > 0.9999f)
                return 0.0f;
        if(wi.z < -0.9999f)
                return -1.0f;

        // a
        const float theta_i = acosf(wi.z);
        const float a = 1.0f/tanf(theta_i)/alpha_i(wi);

        // value
        const float value = 0.5f*((float)erf(a) - 1.0f) + INV_2_SQRT_M_PI / a * expf(-a*a);

        return value;
}
```

Projected Area The projected area is

$$\begin{aligned}\int_\Omega \langle \boldsymbol{\omega}_i, \boldsymbol{\omega}_m \rangle \, D(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m &= (1 + \Lambda(\boldsymbol{\omega}_i)) \, \cos\theta_i \\ &= \frac{\cos\theta_i}{2} \left(\operatorname{erf}(a) + 1\right) + \frac{\alpha_i \, \sin\theta_i}{2\sqrt{\pi}} \, \exp\left(-a^2\right) \\ a &= \frac{1}{\alpha_i \, \tan\theta_i}\end{aligned}$$

It can be computed by calling the implementation of Λ. However, some of the terms simplify when the expression is expanded, making the implementation less sensitive to numerical errors.

```
float MicrosurfaceSlopeBeckmann::projectedArea(const vec3& wi) const
{
        if(wi.z > 0.9999f)
                return 1.0f;
        if(wi.z < -0.9999f)
                return 0.0f;

        // a
        const float alphai = alpha_i(wi);
        const float theta_i = acosf(wi.z);
        const float a = 1.0f/tanf(theta_i)/alphai;

        // value
        const float value = 0.5f*((float)erf(a) + 1.0f)*wi.z + INV_2_SQRT_M_PI * alphai * sinf(theta_i) * expf(-a*a);

        return value;
}
```

Importance Sampling the Distribution of Visible Slopes ($\alpha = 1$)

Code from Wenzel Jakob [Jak14] modified to work for $\theta_i \in [0, \pi[$.
Note: we removed the smart initialization of **erf_min**, which was designed assuming $\theta_i > \frac{\pi}{2}$. Another smart initialization can certainly be found.

```
vec2 MicrosurfaceSlopeBeckmann::sampleP22_11(const float theta_i, const float U, const float U_2) const
{
	vec2 slope;

	if(theta_i < 0.0001f)
	{
		const float r = sqrtf(-logf(U));
		const float phi = 6.28318530718f * U_2;
		slope.x = r * cosf(phi);
		slope.y = r * sinf(phi);
		return slope;
	}

	// constant
	const float sin_theta_i = sinf(theta_i);
	const float cos_theta_i = cosf(theta_i);

	// slope associated to theta_i
	const float slope_i = cos_theta_i/sin_theta_i;

	// projected area
	const float a = cos_theta_i/sin_theta_i;
	const float projectedarea = 0.5f*((float)erf(a) + 1.0f)*cos_theta_i + INV_2_SQRT_M_PI * sin_theta_i * expf(-a*a);
	if(projectedarea < 0.0001f || projectedarea!=projectedarea)
		return vec2(0,0);
	// VNDF normalization factor
	const float c = 1.0f / projectedarea;

	// search
	float erf_min = -0.9999f;
	float erf_max = std::max(erf_min, (float)erf(slope_i));
	float erf_current = 0.5f * (erf_min+erf_max);

	while(erf_max-erf_min > 0.00001f)
	{
		if (!(erf_current >= erf_min && erf_current <= erf_max))
			erf_current = 0.5f * (erf_min + erf_max);

		// evaluate slope
		const float slope = erfinv(erf_current);

		// CDF
		const float CDF = (slope>=slope_i) ? 1.0f : c * (INV_2_SQRT_M_PI*sin_theta_i*expf(-slope*slope) + cos_theta_i*(0.5f+0.5f*(float)erf(slope)));
		const float diff = CDF - U;

		// test estimate
		if( abs(diff) < 0.00001f )
			break;

		// update bounds
		if(diff > 0.0f)
		{
			if(erf_max == erf_current)
				break;
			erf_max = erf_current;
		}
		else
		{
			if(erf_min == erf_current)
				break;
			erf_min = erf_current;
		}

		// update estimate
		const float derivative = 0.5f*c*cos_theta_i - 0.5f*c*sin_theta_i * slope;
		erf_current -= diff/derivative;
	}

	slope.x = erfinv(std::min(erf_max, std::max(erf_min, erf_current)));
	slope.y = erfinv(2.0f*U_2-1.0f);
	return slope;
}
```

2.3 GGX Slope Distribution

```
/* GGX slope distribution */
class MicrosurfaceSlopeGGX : public MicrosurfaceSlope
{
public:
        MicrosurfaceSlopeGGX(const float alpha_x=1.0f, const float alpha_y=1.0f)
                : MicrosurfaceSlope(alpha_x, alpha_y)
        {}

        // distribution of slopes
        virtual float P22(const float slope_x, const float slope_y) const;
        // Smith's Lambda function
        virtual float Lambda(const vec3& wi) const;
        // projected area towards incident direction
        virtual float projectedArea(const vec3& wi) const;
        // sample the distribution of visible slopes with alpha=1.0
        virtual vec2 sampleP22_11(const float theta_i, const float U1, const float U2) const;
};
```

Slope PDF

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

```
float MicrosurfaceSlopeGGX::P22(const float slope_x, const float slope_y) const
{
        const float tmp = 1.0f + slope_x*slope_x/(m_alpha_x*m_alpha_x) + slope_y*slope_y/(m_alpha_y*m_alpha_y);
        const float value = 1.0f / (M_PI * m_alpha_x * m_alpha_y) / (tmp * tmp);
        return value;
}
```

Smith Λ Function

$$\Lambda(\boldsymbol{\omega}_i) = \frac{-1 + \text{sign}(a)\sqrt{1 + \frac{1}{a^2}}}{2}$$

$$a = \frac{1}{\alpha_i\, \tan\theta_i}$$

Note that the sign() function is usually omitted in previous work [Hei14] because it is assumed that $\theta_i > \frac{\pi}{2}$. However, in our context $\theta_i \in [0, \pi]$ and it is important to incorporate the sign.

```
float MicrosurfaceSlopeGGX::Lambda(const vec3& wi) const
{
        if(wi.z > 0.9999f)
                return 0.0f;
        if(wi.z < -0.9999f)
                return -1.0f;

        // a
        const float theta_i = acosf(wi.z);
        const float a = 1.0f/tanf(theta_i)/alpha_i(wi);

        // value
        const float value = 0.5f*(-1.0f + sign(a) * sqrtf(1 + 1/(a*a)));

        return value;
}
```

Projected Area The projected area is

$$\begin{aligned}\int_\Omega \langle \boldsymbol{\omega}_i, \boldsymbol{\omega}_m \rangle \, D(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m &= (1 + \Lambda(\boldsymbol{\omega}_i)) \, \cos\theta_i \\ &= \frac{1}{2}\left(\cos\theta_i + \sqrt{\cos\theta_i^2 + \sin\theta_i^2 \, \alpha_i^2}\right)\end{aligned}$$

It can be computed by calling the implementation of Λ. However some of the terms simplify when the expression is expanded, making the implementation less sensitive to numerical errors.

```
float MicrosurfaceSlopeGGX::projectedArea(const vec3& wi) const
{
        if(wi.z > 0.9999f)
                return 1.0f;
        if( wi.z < -0.9999f)
                return 0.0f;

        // a
        const float theta_i = acosf(wi.z);
        const float sin_theta_i = sinf(theta_i);

        const float alphai = alpha_i(wi);

        // value
        const float value = 0.5f * (wi.z + sqrtf(wi.z*wi.z + sin_theta_i*sin_theta_i*alphai*alphai));

        return value;
}
```

Importance Sampling the Distribution of Visible Slopes ($\alpha = 1$)

Code from Heitz and d'Eon [HD14] modified to work for $\theta_i \in [0, \pi[$.

```
vec2 MicrosurfaceSlopeGGX::sampleP22_11(const float theta_i, const float U, const float U_2) const
{
	vec2 slope;

	if(theta_i < 0.0001f)
	{
		const float r = sqrtf(U/(1.0f-U));
		const float phi = 6.28318530718f * U_2;
		slope.x = r * cosf(phi);
		slope.y = r * sinf(phi);
		return slope;
	}

	// constant
	const float sin_theta_i = sinf(theta_i);
	const float cos_theta_i = cosf(theta_i);
	const float tan_theta_i = sin_theta_i/cos_theta_i;

	// slope associated to theta_i
	const float slope_i = cos_theta_i/sin_theta_i;

	// projected area
	const float projectedarea = 0.5f * (cos_theta_i + 1.0f);
	if(projectedarea < 0.0001f || projectedarea!=projectedarea)
		return vec2(0,0);
	// normalization coefficient
	const float c = 1.0f / projectedarea;

	const float A = 2.0f*U/cos_theta_i/c - 1.0f;
	const float B = tan_theta_i;
	const float tmp = 1.0f / (A*A-1.0f);

	const float D = sqrtf(std::max(0.0f, B*B*tmp*tmp - (A*A-B*B)*tmp));
	const float slope_x_1 = B*tmp - D;
	const float slope_x_2 = B*tmp + D;
	slope.x = (A < 0.0f || slope_x_2 > 1.0f/tan_theta_i) ? slope_x_1 : slope_x_2;

	float U2;
	float S;
	if(U_2 > 0.5f)
	{
	S = 1.0f;
	U2 = 2.0f*(U_2-0.5f);
	}
	else
	{
	S = -1.0f;
	U2 = 2.0f*(0.5f-U_2);
	}
	const float z = (U2*(U2*(U2*0.27385f-0.73369f)+0.46341f)) / (U2*(U2*(U2*0.093073f+0.309420f)-1.000000f)+0.597999f);
	slope.y = S * z * sqrtf(1.0f+slope.x*slope.x);

	return slope;
}
```

3 Microsurface

3.1 API

```
/* API */
class Microsurface
{
public:
        // height distribution
        const MicrosurfaceHeight* m_microsurfaceheight;
        // slope distribution
        const MicrosurfaceSlope* m_microsurfaceslope;

public:

        Microsurface(const bool height_uniform, // uniform or Gaussian height distribution
                                const bool slope_beckmann, // Beckmann or GGX slope distribution
                                const float alpha_x,
                                const float alpha_y) :
                m_microsurfaceheight((height_uniform) ?
                  static_cast<MicrosurfaceHeight*>(new MicrosurfaceHeightUniform)
                : static_cast<MicrosurfaceHeight*>(new MicrosurfaceHeightGaussian)),
                m_microsurfaceslope((slope_beckmann) ?
                  static_cast<MicrosurfaceSlope*>(new MicrosurfaceSlopeBeckmann(alpha_x, alpha_y))
                : static_cast<MicrosurfaceSlope*>(new MicrosurfaceSlopeGGX(alpha_x, alpha_y)))
        {}

        ~Microsurface()
        {
                delete m_microsurfaceheight;
                delete m_microsurfaceslope;
        }

        // evaluate BSDF with a random walk (stochastic but unbiased)
        // scatteringOrder=0 --> contribution from all scattering events
        // scatteringOrder=1 --> contribution from 1st bounce only
        // scatteringOrder=2 --> contribution from 2nd bounce only, etc..
        virtual float eval(const vec3& wi, const vec3& wo, const int scatteringOrder=0) const;

        // sample BSDF with a random walk
        // scatteringOrder is set to the number of bounces computed for this sample
        virtual vec3 sample(const vec3& wi, int& scatteringOrder) const;
        vec3 sample(const vec3& wi) const {int scatteringOrder; return sample(wi, scatteringOrder);}

public:
        // masking function
        float G_1(const vec3& wi) const;
        // masking function at height h0
        float G_1(const vec3& wi, const float h0) const;
        // sample height in outgoing direction
        float sampleHeight(const vec3& wo, const float h0, const float U) const;

public:
        // evaluate local phase function
        virtual float evalPhaseFunction(const vec3& wi, const vec3& wo) const=0;
        // sample local phase function
        virtual vec3 samplePhaseFunction(const vec3& wi) const=0;

        // evaluate BSDF limited to single scattering
        // this is in average equivalent to eval(wi, wo, 1);
        virtual float evalSingleScattering(const vec3& wi, const vec3& wo) const=0;
};
```

Masking Function of Height The masking function at height h is

$$G_1^{\text{dist}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m, h) = C^1(h)^{\Lambda(\boldsymbol{\omega}_i)}$$

```
float Microsurface::G_1(const vec3& wi, const float h0) const
{
        if(wi.z > 0.9999f)
                return 1.0f;
        if(wi.z <= 0.0f)
                return 0.0f;

        // height CDF
        const float C1_h0 = m_microsurfaceheight->C1(h0);
        // Lambda
        const float Lambda = m_microsurfaceslope->Lambda(wi);
        // value
        const float value = powf(C1_h0, Lambda);
        return value;
}
```

Masking Function The masking function is the average over the heights:

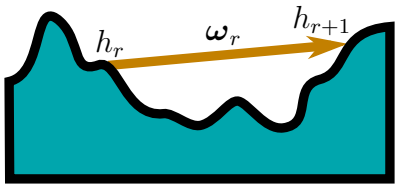
$$\begin{aligned} G_1^{\text{dist}}(\boldsymbol{\omega}_i) &= \int_{-\infty}^{+\infty} P^1(h)\, G_1^{\text{dist}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m, h)\, dh \\ &= \int_{-\infty}^{+\infty} P^1(h)\, C^1(h)^{\Lambda(\boldsymbol{\omega}_i)}\, dh \\ &= \frac{1}{1 + \Lambda(\boldsymbol{\omega}_i)} \end{aligned}$$

```
float Microsurface::G_1(const vec3& wi) const
{
        if(wi.z > 0.9999f)
                return 1.0f;
        if(wi.z <= 0.0f)
                return 0.0f;

        // Lambda
        const float Lambda = m_microsurfaceslope->Lambda(wi);
        // value
        const float value = 1.0f / (1.0f + Lambda);
        return value;
}
```

Height sampling

For a ray starting at height h_r and travelling in direction $\boldsymbol{\omega}_r$ we sample the height h_{r+1} of the next intersection.



Algorithm 1 Sample height $h_{r+1}(\boldsymbol{\omega}_r, h_r, \mathcal{U})$

if $\mathcal{U} \geq 1 - G_1^{\text{dist}}(\boldsymbol{\omega}_r, h_r, \infty)$ **then** ▷ Leave the microsurface

$h_{r+1} = \infty$

else ▷ Intersect the microsurface

$h_{r+1} = C^{-1}\left(\frac{C^1(h_r)}{(1-\mathcal{U})^{1/\Lambda(\boldsymbol{\omega}_r)}}\right)$

end if

return h_{r+1}

In the implementation we consider three special case to avoid numerical instabilities due to $\Lambda = 0$ or $\Lambda = \infty$.

- If $\theta = 0$, the ray goes upward and cannot intersect the microsurface: $h_{r+1} = \infty$.
- If $\theta = \pi$, the ray goes downward and the next height is such that $h_{r+1} \in [C^{-1}(0), h_r]]$, where $C^{-1}(0)$ is the lowest point of the microsurface.
- If $\theta = \frac{\pi}{2}$ the ray goes horizontally and the next height is $h_{r+1} = h_r$.

```
float Microsurface::sampleHeight(const vec3& wr, const float hr, const float U) const
{
        if(wr.z > 0.9999f)
                return FLT_MAX;
        if(wr.z < -0.9999f)
        {
                const float value = m_microsurfaceheight->invC1(U*m_microsurfaceheight->C1(hr));
                return value;
        }
        if(fabsf(wr.z) < 0.0001f)
                return hr;

        // probability of intersection
        const float G_1_ = G_1(wr, hr);

        if (U > 1.0f - G_1_) // leave the microsurface
                return FLT_MAX;

        const float h = m_microsurfaceheight->invC1(
                        m_microsurfaceheight->C1(hr) / powf((1.0f-U),1.0f/m_microsurfaceslope->Lambda(wr))
                        );
        return h;
}
```

Evaluating the BSDF We evaluate the BSDF with a random walk algorithm. Since in this implementation we model non-absorptive materials, there is no need to track energies and the weights of the samples is always 1.

Algorithm 2 Random Walk Evaluation

$L \leftarrow 0$ ▷ intial radiance
$h \leftarrow +\infty$ ▷ initial height
$\boldsymbol{\omega} \leftarrow -\boldsymbol{\omega}_i$ ▷ initial direction
while true **do**
  $h \leftarrow$ sample $(h, \boldsymbol{\omega})$ ▷ next height
  if $h = \infty$ **then** ▷ leave microsurface?
    break
  end if
  $L \leftarrow L + p(-\boldsymbol{\omega}, \boldsymbol{\omega}_o)\, G_1(\boldsymbol{\omega}_o, h)$ ▷ next event estimation
  $\boldsymbol{\omega} \leftarrow$ sample $p(-\boldsymbol{\omega}, .)$ ▷ next direction
end while
return L

```
float Microsurface::eval(const vec3& wi, const vec3& wo, const int scatteringOrder) const
{
	if(wo.z < 0)
		return 0;
	// init
	vec3 wr = -wi;
	float hr = 1.0f + m_microsurfaceheight->invC1(0.999f);

	float sum = 0;

	// random walk
	int current_scatteringOrder = 0;
	while(scatteringOrder==0 || current_scatteringOrder <= scatteringOrder)
	{
		// next height
		float U = generateRandomNumber();
		hr = sampleHeight(wr, hr, U);

		// leave the microsurface?
		if( hr == FLT_MAX )
			break;
		else
			current_scatteringOrder++;

		// next event estimation
		float phasefunction = evalPhaseFunction(-wr, wo);
		float shadowing = G_1(wo, hr);
		float I = phasefunction * shadowing;

		if ( IsFiniteNumber(I) && (scatteringOrder==0 || current_scatteringOrder==scatteringOrder) )
			sum += I;

		// next direction
		wr = samplePhaseFunction(-wr);

		// if NaN (should not happen, just in case)
		if( (hr != hr) || (wr.z != wr.z) )
			return 0.0f;
	}

	return sum;
}
```

Importance Sampling the BSDF We importance sample the BSDF with a random walk algorithm. Since in this implementation we model non-absorptive materials, there is no need to track energies and the weights of the samples is always 1.

Algorithm 3 Random Walk Importance Sampling

$h \leftarrow +\infty$ ▷ initial height
$\boldsymbol{\omega} \leftarrow -\boldsymbol{\omega}_i$ ▷ initial direction
while true **do**
  $h \leftarrow$ sample $(h, \boldsymbol{\omega})$ ▷ next height
  if $h = \infty$ **then** ▷ leave microsurface?
    break
  end if
  $\boldsymbol{\omega} \leftarrow$ sample $p(-\boldsymbol{\omega}, .)$ ▷ next direction
end while
return $\boldsymbol{\omega}$

```
vec3 Microsurface::sample(const vec3& wi, int& scatteringOrder) const
{
	// init
	vec3 wr = -wi;
	float hr = 1.0f + m_microsurfaceheight->invC1(0.999f);

	// random walk
	scatteringOrder = 0;
	while(true)
	{
		// next height
		float U = generateRandomNumber();
		hr = sampleHeight(wr, hr, U);

		// leave the microsurface?
		if( hr == FLT_MAX )
			break;
		else
			scatteringOrder++;

		// next direction
		wr = samplePhaseFunction(-wr);

		// if NaN (should not happen, just in case)
		if( (hr != hr) || (wr.z != wr.z) )
			return vec3(0,0,1);
	}

	return wr;
}
```

3.2 Conductor Microsurface

This class implements the BSDF of a 100% energy conserving rough conductor material.

```
/* Microsurface made of conductor material */
class MicrosurfaceConductor : public Microsurface
{
public:
        MicrosurfaceConductor(const bool height_uniform, // uniform or Gaussian
                              const bool slope_beckmann, // Beckmann or GGX
                              const float alpha_x,
                              const float alpha_y)
                : Microsurface(height_uniform, slope_beckmann, alpha_x, alpha_y)
        {}

public:
        // evaluate local phase function
        virtual float evalPhaseFunction(const vec3& wi, const vec3& wo) const;
        // sample local phase function
        virtual vec3 samplePhaseFunction(const vec3& wi) const;

        // evaluate BSDF limited to single scattering
        // this is in average equivalent to eval(wi, wo, 1);
        virtual float evalSingleScattering(const vec3& wi, const vec3& wo) const;
};
```

Evaluating the Phase Function The conductor phase function is

$$p^{\text{cond}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_h)}{4\,|\boldsymbol{\omega}_i \cdot \boldsymbol{\omega}_h|}.$$

Note that a Fresnel term can be added, but the material will not be 100% energy conserving.

```
float MicrosurfaceConductor::evalPhaseFunction(const vec3& wi, const vec3& wo) const
{
        // half vector
        const vec3 wh = normalize(wi+wo);
        if(wh.z < 0.0f)
                return 0.0f;

        // value
        const float value = 0.25f * m_microsurfaceslope->D_wi(wi, wh) / dot(wi, wh);
        return value;
}
```

Importance Sampling the Phase Function To sample the phase function, we generate a normal $\boldsymbol{\omega}_m$ by sampling $D_{\boldsymbol{\omega}_i}$ and we apply the reflection operator.
The weight of the sample is 1. If a Fresnel term is added, it is represented in the weight of the sample.

Algorithm 4 Sample conductor phase function

$\boldsymbol{\omega}_m \leftarrow$ sample $D_{\boldsymbol{\omega}_i}$
$\boldsymbol{\omega}_o \leftarrow \text{reflect}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m)$ ▷ outgoing direction
$w \leftarrow F(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m)$ ▷ weight

```
vec3 MicrosurfaceConductor::samplePhaseFunction(const vec3& wi) const
{
        const float U1 = generateRandomNumber();
        const float U2 = generateRandomNumber();

        vec3 wm = m_microsurfaceslope->sampleD_wi(wi, U1, U2);

        // reflect
        const vec3 wo = -wi + 2.0f * wm * dot(wi, wm);

        return wo;
}
```

Single Scattering BSDF

The single scattering BSDF can be used to reduce the variance of the random walk algorithm (the contribution of the first bounce can be replaced with the single scattering BSDF, hence removing the variance introduced by the averaged masking-shadowing function). This is not done in this implementation, but we use the single scattering for testing and validation, see Section 4.4.3.

```
float MicrosurfaceConductor::evalSingleScattering(const vec3& wi, const vec3& wo) const
{
        // half-vector
        const vec3 wh = normalize(wi+wo);
        const float D = m_microsurfaceslope->D(wh);

        // masking-shadowing
        const float G2 = 1.0f / (1.0f + m_microsurfaceslope->Lambda(wi) + m_microsurfaceslope->Lambda(wo));

        // BRDF * cos
        const float value = D * G2 / (4.0f * wi.z);

        return value;
}
```

3.3 Diffuse Microsurface

This class implements the BSDF of a 100% energy conserving rough diffuse material.

```
/* Microsurface made of conductor material */
class MicrosurfaceDiffuse : public Microsurface
{
public:
        MicrosurfaceDiffuse(const bool height_uniform, // uniform or Gaussian
                            const bool slope_beckmann, // Beckmann or GGX
                            const float alpha_x,
                            const float alpha_y)
                : Microsurface(height_uniform, slope_beckmann, alpha_x, alpha_y)
        {}

public:
        // evaluate local phase function
        virtual float evalPhaseFunction(const vec3& wi, const vec3& wo) const;
        // sample local phase function
        virtual vec3 samplePhaseFunction(const vec3& wi) const;

        // evaluate BSDF limited to single scattering
        // this is in average equivalent to eval(wi, wo, 1);
        virtual float evalSingleScattering(const vec3& wi, const vec3& wo) const;
};
```

Evaluating the Phase Function The diffuse phase function is

$$p^{\text{diff}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{1}{\pi} \int_\Omega \langle \boldsymbol{\omega}_o, \boldsymbol{\omega}_m \rangle \, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m.$$

We evaluate it stochastically by sampling a normal $\boldsymbol{\omega}_m$ from $D_{\boldsymbol{\omega}_i}$ and evaluating the diffuse BRDF with this normal.

Algorithm 5 Stochastic evaluation of diffuse phase function

$\boldsymbol{\omega}_m \leftarrow$ sample $D_{\boldsymbol{\omega}_i}$
return $\frac{a}{\pi} \langle \boldsymbol{\omega}_o, \boldsymbol{\omega}_m \rangle$

```
float MicrosurfaceDiffuse::evalPhaseFunction(const vec3& wi, const vec3& wo) const
{
        const float U1 = generateRandomNumber();
        const float U2 = generateRandomNumber();
        vec3 wm = m_microsurfaceslope->sampleD_wi(wi, U1, U2);

        // value
        const float value = 1.0f/M_PI * std::max(0.0f, dot(wo, wm));
        return value;
}
```

Importance Sampling the Phase Function To sample the phase function, we generate a normal $\boldsymbol{\omega}_m$ by sampling $D_{\boldsymbol{\omega}_i}$ and we apply the reflection operator.

Algorithm 6 Sample conductor phase function

$\boldsymbol{\omega}_m \leftarrow$ sample $D_{\boldsymbol{\omega}_i}$

$\boldsymbol{\omega}_o \leftarrow \text{diffuse}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m)$ ▷ outgoing direction

$w \leftarrow a$ ▷ weight

```
 // build orthonormal basis (Building an Orthonormal Basis from a 3D Unit Vector Without Normalization, [Frisvad2012])
 void buildOrthonormalBasis(vec3& omega_1, vec3& omega_2, const vec3& omega_3)
{
        if(omega_3.z < -0.9999999f)
        {
           omega_1 = vec3 ( 0.0f , -1.0f , 0.0f );
           omega_2 = vec3 ( -1.0f , 0.0f , 0.0f );
        } else {
           const float a = 1.0f /(1.0f + omega_3.z );
           const float b = -omega_3.x*omega_3 .y*a ;
           omega_1 = vec3 (1.0f - omega_3.x*omega_3. x*a , b , -omega_3.x );
           omega_2 = vec3 (b , 1.0f - omega_3.y*omega_3.y*a , -omega_3.y );
        }
}


vec3 MicrosurfaceDiffuse::samplePhaseFunction(const vec3& wi) const
{
        const float U1 = generateRandomNumber();
        const float U2 = generateRandomNumber();
        const float U3 = generateRandomNumber();
        const float U4 = generateRandomNumber();

        vec3 wm = m_microsurfaceslope->sampleD_wi(wi, U1, U2);

        // sample diffuse reflection
        vec3 w1, w2;
        buildOrthonormalBasis(w1, w2, wm);

        float r1 = 2.0f*U3 - 1.0f;
        float r2 = 2.0f*U4 - 1.0f;

        // concentric map code from
        // http://psgraphics.blogspot.ch/2011/01/improved-code-for-concentric-map.html
        float phi, r;
        if (r1 == 0 && r2 == 0) {
                r = phi = 0;
        } else if (r1*r1 > r2*r2) {
                r = r1;
                phi = (M_PI/4.0f) * (r2/r1);
        } else {
                r = r2;
                phi = (M_PI/2.0f) - (r1/r2) * (M_PI/4.0f);
        }
        float x = r*cosf(phi);
        float y = r*sinf(phi);
        float z = sqrtf(std::max(0.0f, 1.0f - x*x - y*y));
        vec3 wo = x*w1 + y*w2 + z*wm;

        return wo;
}
```

Single Scattering BSDF

The single scattering BSDF can be used to reduce the variance of the random walk algorithm (the contribution of the first bounce can be replaced with the single scattering BSDF, hence removing the variance introduced by the averaged masking-shadowing function). This is not done in this implementation, but we use the single scattering for testing and validation, see Section 4.4.3.

```
// stochastic evaluation
// Heitz and Dupuy 2015
// Implementing a Simple Anisotropic Rough Diffuse Material with Stochastic Evaluation
float MicrosurfaceDiffuse::evalSingleScattering(const vec3& wi, const vec3& wo) const
{
        // sample visible microfacet
        const float U1 = generateRandomNumber();
        const float U2 = generateRandomNumber();
        const vec3 wm = m_microsurfaceslope->sampleD_wi(wi, U1, U2);

        // shadowing given masking
        const float Lambda_i = m_microsurfaceslope->Lambda(wi);
        const float Lambda_o = m_microsurfaceslope->Lambda(wo);
        float G2_given_G1 = (1.0f + Lambda_i) / (1.0f + Lambda_i + Lambda_o);

        // evaluate diffuse and shadowing given masking
        const float value = 1.0f / (float)M_PI * std::max(0.0f, dot(wm, wo)) * G2_given_G1;

        return value;
}
```

3.4 Dieletric Microsurface

This class implements the BSDF of a 100% energy conserving rough dielectric material.

```
/* Microsurface made of conductor material */
class MicrosurfaceDielectric : public Microsurface
{
public:
        const float m_eta;
public:
        MicrosurfaceDielectric(const bool height_uniform, // uniform or Gaussian
                               const bool slope_beckmann, // Beckmann or GGX
                               const float alpha_x,
                               const float alpha_y,
                               const float eta = 1.5f)
                : Microsurface(height_uniform, slope_beckmann, alpha_x, alpha_y),
                m_eta(eta)
        {}

        // evaluate BSDF with a random walk (stochastic but unbiased)
        // scatteringOrder=0 --> contribution from all scattering events
        // scatteringOrder=1 --> contribution from 1st bounce only
        // scatteringOrder=2 --> contribution from 2nd bounce only, etc..
        virtual float eval(const vec3& wi, const vec3& wo, const int scatteringOrder=0) const;

        // sample final BSDF with a random walk
        // scatteringOrder is set to the number of bounces computed for this sample
        virtual vec3 sample(const vec3& wi, int& scatteringOrder) const;

public:
        // evaluate local phase function
        virtual float evalPhaseFunction(const vec3& wi, const vec3& wo) const;
        float evalPhaseFunction(const vec3& wi, const vec3& wo, const bool wi_outside, const bool wo_outside) const;
        // sample local phase function
        virtual vec3 samplePhaseFunction(const vec3& wi) const;
        vec3 samplePhaseFunction(const vec3& wi, const bool wi_outside, bool& wo_outside) const;

        // evaluate BSDF limited to single scattering
        // this is in average equivalent to eval(wi, wo, 1);
        virtual float evalSingleScattering(const vec3& wi, const vec3& wo) const;

protected:
        float Fresnel(const vec3& wi, const vec3& wm, const float eta) const;
        vec3 refract(const vec3 &wi, const vec3 &wm, const float eta) const;
};
```

For dielectric material we need to write dedicated virtual methods **eval** and **sample**: we modify the random walk algorithm because the ray can cross the boundary and scatter inside the material.

The Fresnel Coefficient

```
vec3 MicrosurfaceDielectric::refract(const vec3 &wi, const vec3 &wm, const float eta) const
{
        const float cos_theta_i = dot(wi, wm);
        const float cos_theta_t2 = 1.0f - (1.0f-cos_theta_i*cos_theta_i) / (eta*eta);
        const float cos_theta_t = -sqrtf(std::max(0.0f,cos_theta_t2));

        return wm * (dot(wi, wm) / eta + cos_theta_t) - wi / eta;
}


float MicrosurfaceDielectric::Fresnel(const vec3& wi, const vec3& wm, const float eta) const
{
        const float cos_theta_i = dot(wi, wm);
        const float cos_theta_t2 = 1.0f - (1.0f-cos_theta_i*cos_theta_i) / (eta*eta);

        // total internal reflection
        if (cos_theta_t2 <= 0.0f) return 1.0f;

        const float cos_theta_t = sqrtf(cos_theta_t2);

        const float Rs = (cos_theta_i - eta * cos_theta_t) / (cos_theta_i + eta * cos_theta_t);
        const float Rp = (eta * cos_theta_i - cos_theta_t) / (eta * cos_theta_i + cos_theta_t);

        const float F = 0.5f * (Rs * Rs + Rp * Rp);
        return F;
}
```

Evaluating the Phase Function The phase function is

$$p^{\text{diel}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{F(\boldsymbol{\omega}_i, \boldsymbol{\omega}_{h_r})\, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_{h_r})}{4\,|\boldsymbol{\omega}_i \cdot \boldsymbol{\omega}_{h_r}|} + \langle \boldsymbol{\omega}_o, \boldsymbol{\omega}_m \rangle \frac{\eta_o^2\,(1 - F(\boldsymbol{\omega}_i, \boldsymbol{\omega}_{h_t}))\, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_{h_t})}{(\eta_i(\boldsymbol{\omega}_i \cdot \boldsymbol{\omega}_h) + \eta_o(\boldsymbol{\omega}_o \cdot \boldsymbol{\omega}_h))^2}$$

```
// wrapper (only for the API and testing)
float MicrosurfaceDielectric::evalPhaseFunction(const vec3& wi, const vec3& wo) const
{
        return evalPhaseFunction(wi, wo, true, true) + evalPhaseFunction(wi, wo, true, false);
}

float MicrosurfaceDielectric::evalPhaseFunction(const vec3& wi, const vec3& wo, const bool wi_outside, const bool wo_outside) const
{
        const float eta = wi_outside ? m_eta : 1.0f / m_eta;

        if( wi_outside == wo_outside ) // reflection
        {
                // half vector
                const vec3 wh = normalize(wi+wo);
                // value
                const float value = (wi_outside) ?
                        (0.25f * m_microsurfaceslope->D_wi(wi, wh) / dot(wi, wh) * Fresnel(wi, wh, eta)) :
                        (0.25f * m_microsurfaceslope->D_wi(-wi, -wh) / dot(-wi, -wh) * Fresnel(-wi, -wh, eta)) ;
                return value;
        }
        else // transmission
        {
                vec3 wh = -normalize(wi+wo*eta);
                wh *= (wi_outside) ? (sign(wh.z)) : (-sign(wh.z));

                if(dot(wh, wi) < 0)
                        return 0;

                float value;
                if(wi_outside){
                        value = eta*eta * (1.0f-Fresnel(wi, wh, eta)) *
                                m_microsurfaceslope->D_wi(wi, wh) * std::max(0.0f, -dot(wo, wh)) *
                                1.0f / powf(dot(wi, wh)+eta*dot(wo,wh), 2.0f);
                }
                else
                {
                        value = eta*eta * (1.0f-Fresnel(-wi, -wh, eta)) *
                                m_microsurfaceslope->D_wi(-wi, -wh) * std::max(0.0f, -dot(-wo, -wh)) *
                                1.0f / powf(dot(-wi, -wh)+eta*dot(-wo,-wh), 2.0f);
                }

                return value;
        }
}
```

Importance Sampling the Phase Function To sample the phase function, we generate a normal $\boldsymbol{\omega}_m$ by sampling $D_{\boldsymbol{\omega}_i}$ and we apply the reflection operator.

Algorithm 7 Sample dielectric phase function

$\boldsymbol{\omega}_m \leftarrow$ sample $D_{\boldsymbol{\omega}_i}$
if $\mathcal{U} < F(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m)$ **then**
    $\boldsymbol{\omega}_o \leftarrow \text{reflect}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m)$ ▷ outgoing direction
else
    $\boldsymbol{\omega}_o \leftarrow \text{transmit}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_m)$ ▷ outgoing direction
end if
$w \leftarrow 1$ ▷ weight

```
vec3 MicrosurfaceDielectric::samplePhaseFunction(const vec3& wi) const
{
        bool wo_outside;
        return samplePhaseFunction(wi, true, wo_outside);
}

vec3 MicrosurfaceDielectric::samplePhaseFunction(const vec3& wi, const bool wi_outside, bool& wo_outside) const
{
        const float U1 = generateRandomNumber();
        const float U2 = generateRandomNumber();

        const float eta = wi_outside ? m_eta : 1.0f / m_eta;

        vec3 wm = wi_outside ? (m_microsurfaceslope->sampleD_wi(wi, U1, U2)) :
                                                (-m_microsurfaceslope->sampleD_wi(-wi, U1, U2)) ;

        const float F = Fresnel(wi, wm, eta);

        if( generateRandomNumber() < F )
        {
                const vec3 wo = -wi + 2.0f * wm * dot(wi, wm); // reflect
                return wo;
        }
        else
        {
                wo_outside = !wi_outside;
                const vec3 wo = refract(wi, wm, eta);
                return normalize(wo);
        }
}
```

Evaluating the BSDF We evaluate the BSDF with a random walk algorithm.

This algorithm is similar to the algorithm on page 18 but tracks an additional Boolean variable **outside**. By convention, the outside of the material is always on top of the inside in the z dimension. Each time the ray is transmitted, the value of **outside** changes. If the ray is inside, the height sampling is done by flipping the configuration vertically, i.e. we update $h_r \leftarrow C^{-1}(1 - C^1(h_r))$. If the height distribution is symmetric: $P^1(h) = P^1(-h)$, this equivalent to change the sign: $h_r \leftarrow -h_r$.

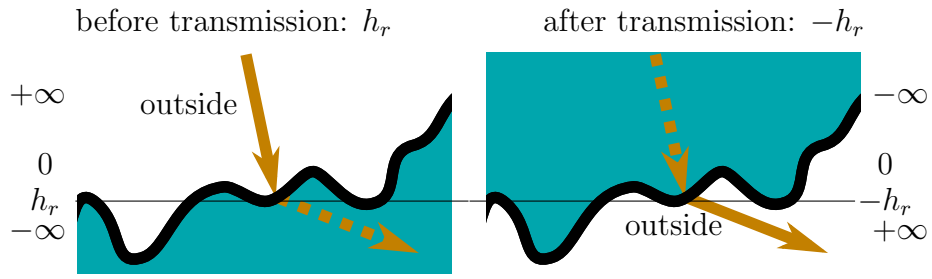


Figure 1: *Vertical flip.* After each transmission event we vertically flip the configuration to track the correct values.

```
float MicrosurfaceDielectric::eval(const vec3& wi, const vec3& wo, const int scatteringOrder) const
{
	// init
	vec3 wr = -wi;
	float hr = 1.0f + m_microsurfaceheight->invC1(0.999f);
	bool outside = true;

	float sum = 0.0f;

	// random walk
	int current_scatteringOrder = 0;
	while(scatteringOrder==0 || current_scatteringOrder <= scatteringOrder)
	{
		// next height
		float U = generateRandomNumber();
		hr = (outside) ? sampleHeight(wr, hr, U) : -sampleHeight(-wr, -hr, U);

		// leave the microsurface?
		if( hr == FLT_MAX || hr == -FLT_MAX)
			break;
		else
			current_scatteringOrder++;

		// next event estimation
		float phasefunction = evalPhaseFunction(-wr, wo, outside, (wo.z>0) );
		float shadowing = (wo.z>0) ? G_1(wo, hr) : G_1(-wo, -hr);
		float I = phasefunction * shadowing;

		if ( IsFiniteNumber(I) && (scatteringOrder==0 || current_scatteringOrder==scatteringOrder) )
			sum += I;

		// next direction
		wr = samplePhaseFunction(-wr, outside, outside);

		// if NaN (should not happen, just in case)
		if( (hr != hr) || (wr.z != wr.z) )
			return 0.0f;
	}

	return sum;
}
```

Importance Sampling the BSDF We importance sample the BSDF with a random walk algorithm.

This algorithm is similar to the algorithm on page 19 but is modified in the same way as the dielectric **eval**().

```
vec3 MicrosurfaceDielectric::sample(const vec3& wi, int& scatteringOrder) const
{
	// init
	vec3 wr = -wi;
	float hr = 1.0f + m_microsurfaceheight->invC1(0.999f);
	bool outside = true;

	// random walk
	scatteringOrder = 0;
	while(true)
	{
		// next height
		float U = generateRandomNumber();
		hr = (outside) ? sampleHeight(wr, hr, U) : -sampleHeight(-wr, -hr, U);

		// leave the microsurface?
		if( hr == FLT_MAX || hr == -FLT_MAX)
			break;
		else
			scatteringOrder++;

		// next direction
		wr = samplePhaseFunction(-wr, outside, outside);

		// if NaN (should not happen, just in case)
		if( (hr != hr) || (wr.z != wr.z) )
			return vec3(0,0,1);
	}

	return wr;
}
```

Single Scattering BSDF

The single scattering BSDF can be used to reduce the variance of the random walk algorithm (the contribution of the first bounce can be replaced with the single scattering BSDF, hence removing the variance introduced by the averaged masking-shadowing function). This is not done in this implementation, but we use the single scattering for testing and validation, see Section 4.4.3.

```
float MicrosurfaceDielectric::evalSingleScattering(const vec3& wi, const vec3& wo) const
{
        bool wi_outside = true;
        bool wo_outside = wo.z > 0;

        const float eta = m_eta;

        if(wo_outside) // reflection
        {
                // D
                const vec3 wh = normalize(vec3(wi+wo));
                const float D = m_microsurfaceslope->D(wh);

                // masking shadowing
                const float Lambda_i = m_microsurfaceslope->Lambda(wi);
                const float Lambda_o = m_microsurfaceslope->Lambda(wo);
                const float G2 = 1.0f / (1.0f + Lambda_i + Lambda_o);

                // BRDF
                const float value = Fresnel(wi, wh, eta) * D * G2 / (4.0f * wi.z);
                return value;
        }
        else // refraction
        {
                // D
                vec3 wh = -normalize(wi+wo*eta);
                if(eta<1.0f)
                        wh = -wh;
                const float D = m_microsurfaceslope->D(wh);

                // G2
                const float Lambda_i = m_microsurfaceslope->Lambda(wi);
                const float Lambda_o = m_microsurfaceslope->Lambda(-wo);
                const float G2 = (float) beta(1.0f+Lambda_i, 1.0f+Lambda_o);

                // BSDF
                const float value = std::max(0.0f, dot(wi, wh)) * std::max(0.0f, -dot(wo, wh)) *
                                                        1.0f / wi.z * eta*eta * (1.0f-Fresnel(wi, wh, eta)) *
                                                        G2 * D / powf(dot(wi, wh)+eta*dot(wo,wh), 2.0f);
                return value;
        }
}
```

4 Tests for Validation and Debug

In this section we propose validation procedures that we use to validate the different milestones of our implementation.

Generating Random Configurations

```
static std::random_device rd;
static std::mt19937 gen(rd);
static std::uniform_real_distribution<> dis(0, 1);

float generateRandomNumber()
{
        return (float)dis(gen);
}

vec3 generateRandomDirection()
{
        const float theta = M_PI * (float)dis(gen);
        const float phi = 2.0f * M_PI * (float)dis(gen);
        const vec3 w(cosf(phi)*sinf(theta), sinf(phi)*sinf(theta), cosf(theta));

        cout << "w = (" << w.x << ", " << w.y << ", " << w.z << ")" << endl;
        cout << endl;

        return w;
}

vec3 generateRandomDirectionUp()
{
        const float theta = 0.5f * M_PI * (float)dis(gen);
        const float phi = 2.0f * M_PI * (float)dis(gen);
        const vec3 w(cosf(phi)*sinf(theta), sinf(phi)*sinf(theta), cosf(theta));

        cout << "w = (" << w.x << ", " << w.y << ", " << w.z << ")" << endl;
        cout << endl;

        return w;
}

Microsurface * generateRandomMicrosurface()
{
        // height
        const bool height_uniform = generateRandomNumber() > 0.5f;

        // roughness
        const float alpha_x = generateRandomNumber() + 0.1f;
        const float alpha_y = generateRandomNumber() + 0.1f;

        // distribution (Beckmann or GGX)
        const bool Beckmann = generateRandomNumber() > 0.5f;

        // material
        Microsurface * m;
        const int material = 2;(int)floor(generateRandomNumber()*3.0);
        switch(material)
        {
                case 0: m = new MicrosurfaceDiffuse(height_uniform, Beckmann, alpha_x, alpha_y);
                                cout << "Material: Diffuse" << endl;
                                break;
                case 1: m = new MicrosurfaceConductor(height_uniform, Beckmann, alpha_x, alpha_y);
                                cout << "Material: Conductor" << endl;
                                break;
                default: m = new MicrosurfaceDielectric(height_uniform, Beckmann, alpha_x, alpha_y);
                                cout << "Material: Dielectric" << endl;
                                break;
        }

        cout << "height distribution: " << ((height_uniform)?"Uniform":"Gaussian") << endl;
        cout << "slope distribution: " << ((Beckmann)?"Beckmann":"GGX") << endl;
        cout << "alpha_x = " << m->m_microsurfaceslope->m_alpha_x << endl;
        cout << "alpha_y = " << m->m_microsurfaceslope->m_alpha_y << endl;
        cout << endl;

        return m;
}
```

4.1 Intersection with the Microsurface

We verify that the intersection probabilities are coherent with the masking-shadowing functions.

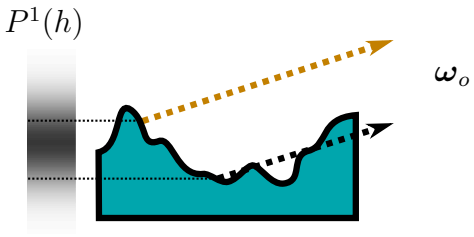
4.1.1 Masking

We generate a random height h and sample the height of the next intersection in direction $\boldsymbol{\omega}_i$. The probability that the ray leaves the microsurface is the masking probability averaged over the heights

$$G_1^{\text{dist}}(\boldsymbol{\omega}_o) = \frac{1}{1+\Lambda(\boldsymbol{\omega}_o)}.$$

Algorithm 8 Test Masking

$V \leftarrow 0$
for $n = 1..N$ **do**
  $h = C^{-1}(\mathcal{U}_1)$
  $h_{\boldsymbol{\omega}_o} = h_{n+1}(\boldsymbol{\omega}_o, h, \mathcal{U}_2)$
  if $h_{\boldsymbol{\omega}_o} = \infty$ **then**
    $V \leftarrow V + \frac{1}{N}$
  end if
end for
return V ▷ $\lim_{N\to\infty} V = \frac{1}{1+\Lambda(\boldsymbol{\omega}_o)}$




```
void test_masking()
{
        const vec3 wo = generateRandomDirectionUp();
        Microsurface * m = generateRandomMicrosurface();

        // analytic
        float G1 = 1.0f / (1.0f + m->m_microsurfaceslope->Lambda(wo));

        // sampling
        const int N = 100000;
        double V = 0;
        for(int n=0 ; n<N ; ++n)
        {
                // generate random height
                const float U1 = generateRandomNumber();
                const float h = m->m_microsurfaceheight->invC1(U1);

                // next height
                const float U2 = generateRandomNumber();
                const float h_wo = m->sampleHeight(wo, h, U2);

                // leave microsurface
                if(h_wo == FLT_MAX)
                        V += 1.0 / (double)N;
        }

        cout << "analytic = \t" << G1 << endl;
        cout << "stochastic = \t" << V << endl;

        delete m;
}
```

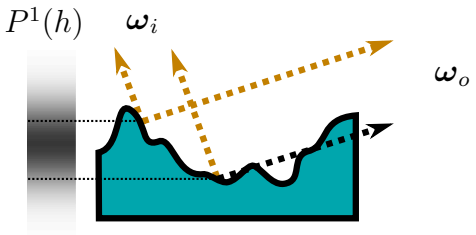
4.1.2 Masking-Shadowing

We generate a random height h and sample the heights of the next intersections in directions $\boldsymbol{\omega}_i$ and $\boldsymbol{\omega}_o$. The probability that the rays leave the microsurface in both directions is the masking-shadowing probability averaged over the heights

$$G_2^{\text{dist}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \frac{1}{1 + \Lambda(\boldsymbol{\omega}_i) + \Lambda(\boldsymbol{\omega}_o)}.$$

Algorithm 9 Test Masking-Shadowing

$V \leftarrow 0$
for $n = 1..N$ **do**
  $h = C^{-1}(\mathcal{U}_1)$
  $h_{\boldsymbol{\omega}_i} = h_{n+1}(\boldsymbol{\omega}_i, h, \mathcal{U}_2)$
  $h_{\boldsymbol{\omega}_o} = h_{n+1}(\boldsymbol{\omega}_o, h, \mathcal{U}_3)$
  if $h_{\boldsymbol{\omega}_i} = \infty$ and $h_{\boldsymbol{\omega}_o} = \infty$ **then**
    $V \leftarrow V + \frac{1}{N}$
  end if
end for
return V ▷ $\lim_{N\to\infty} V = \frac{1}{1+\Lambda(\boldsymbol{\omega}_i)+\Lambda(\boldsymbol{\omega}_o)}$




```
void test_masking_shadowing()
{
        const vec3 wi = generateRandomDirectionUp();
        const vec3 wo = generateRandomDirectionUp();
        Microsurface * m = generateRandomMicrosurface();

        // analytic
        float G2 = 1.0f / (1.0f + m->m_microsurfaceslope->Lambda(wi) + m->m_microsurfaceslope->Lambda(wo));

        // sampling
        const int N = 100000;
        double V = 0;
        for(int n=0 ; n<N ; ++n)
        {
                // generate random height
                const float U1 = generateRandomNumber();
                const float h = m->m_microsurfaceheight->invC1(U1);

                // next heights
                const float U2 = generateRandomNumber();
                const float U3 = generateRandomNumber();
                const float h_wi = m->sampleHeight(wi, h, U2);
                const float h_wo = m->sampleHeight(wo, h, U3);

                // both leave microsurface
                if( (h_wi == FLT_MAX) && (h_wo == FLT_MAX))
                        V += 1.0 / (double)N;
        }

        cout << "analytic = \t" << G2 << endl;
        cout << "stochastic = \t" << V << endl;

        delete m;
}
```

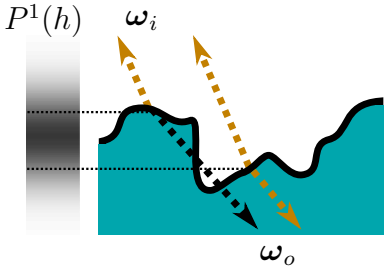
4.1.3 Masking-Shadowing Transmission

Same test as in Section 4.1.2 with transmitted direction $\boldsymbol{\omega}_o$:

$$G_2^{\text{dist}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \mathrm{B}(1 + \Lambda(\boldsymbol{\omega}_i), 1 + \Lambda(-\boldsymbol{\omega}_o)).$$

Algorithm 10 Test Masking-Shadowing Transmission

$V \leftarrow 0$
for $n = 1..N$ **do**
  $h = C^{-1}(\mathcal{U}_1)$
  $h_{\boldsymbol{\omega}_i} = h_{n+1}(\boldsymbol{\omega}_i, h, \mathcal{U}_2)$
  $h_{\boldsymbol{\omega}_o} = h_{n+1}(-\boldsymbol{\omega}_o, -h, \mathcal{U}_3)$
  if $h_{\boldsymbol{\omega}_i} = \infty$ and $h_{\boldsymbol{\omega}_o} = \infty$ **then**
    $V \leftarrow V + \frac{1}{N}$
  end if
end for
return V ▷ $\lim_{N\to\infty} V = \mathrm{B}(1 + \Lambda(\boldsymbol{\omega}_i), 1 + \Lambda(-\boldsymbol{\omega}_o))$




```
void test_masking_shadowing_transmission()
{
        const vec3 wi = generateRandomDirectionUp();
        const vec3 wo = -generateRandomDirectionUp(); // lower hemisphere
        Microsurface * m = generateRandomMicrosurface();

        // analytic
        float G2 = (float) beta(1.0f + m->m_microsurfaceslope->Lambda(wi), 1.0f + m->m_microsurfaceslope->Lambda(-wo));

        // sampling
        const int N = 100000;
        double V = 0;
        for(int n=0 ; n<N ; ++n)
        {
                // generate random height
                const float U1 = generateRandomNumber();
                const float h = m->m_microsurfaceheight->invC1(U1);

                // next heights
                const float U2 = generateRandomNumber();
                const float U3 = generateRandomNumber();
                const float h_wi = m->sampleHeight(wi, h, U2);
                const float h_wo = m->sampleHeight(-wo, -h, U3); // vertical flip

                // both leave microsurface
                if(h_wi == FLT_MAX && h_wo == FLT_MAX)
                        V += 1.0 / (double)N;
        }

        cout << "analytic = \t" << G2 << endl;
        cout << "stochastic = \t" << V << endl;

        delete m;
}
```

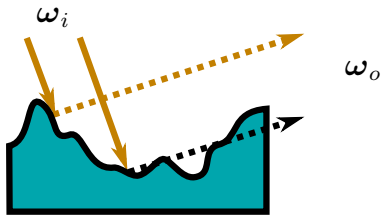
4.1.4 Shadowing Given Masking

We sample a height h by casting a ray towards the microsurface from direction $\boldsymbol{\omega}_i$, i.e. the ray travels towards direction $-\boldsymbol{\omega}_i$. This point of the microsurface is not masked in direction $\boldsymbol{\omega}_i$. Then, we sample the height of the next intersection in direction $\boldsymbol{\omega}_o$. The probability that the ray leaves the microsurface in direction $\boldsymbol{\omega}_o$ is the shadowing in direction $\boldsymbol{\omega}_o$ given that the ray is not masked in direction $\boldsymbol{\omega}_i$

$$\frac{G_2^{\text{dist}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o)}{G_1^{\text{dist}}(\boldsymbol{\omega}_i)} = \frac{1 + \Lambda(\boldsymbol{\omega}_i)}{1 + \Lambda(\boldsymbol{\omega}_i) + \Lambda(\boldsymbol{\omega}_o)}.$$

Algorithm 11 Test Conditional Masking-Shadowing

$V \leftarrow 0$
for $n = 1..N$ **do**
  $h = h_{n+1}(-\boldsymbol{\omega}_i, h, \mathcal{U})$
  $h_{\boldsymbol{\omega}_o} = h_{n+1}(\boldsymbol{\omega}_o, h, \mathcal{U})$
  if $h_{\boldsymbol{\omega}_o} = \infty$ **then**
    $V \leftarrow V + \frac{1}{N}$
  end if
end for
return V ▷ $\lim_{N\to\infty} V = \frac{1+\Lambda(\boldsymbol{\omega}_i)}{1+\Lambda(\boldsymbol{\omega}_i)+\Lambda(\boldsymbol{\omega}_o)}$




```
void test_shadowing_given_masking()
{
        const vec3 wi = generateRandomDirectionUp();
        const vec3 wo = generateRandomDirectionUp();
        Microsurface * m = generateRandomMicrosurface();

        // analytic
        float G2_cond = (1.0f + m->m_microsurfaceslope->Lambda(wi)) /
                (1.0f + m->m_microsurfaceslope->Lambda(wi) + m->m_microsurfaceslope->Lambda(wo));

        // sampling
        const int N = 100000;
        double V = 0;
        for(int n=0 ; n<N ; ++n)
        {
                // intersect the mirocrsurface from wi
                const float U1 = generateRandomNumber();
                const float h = m->sampleHeight(-wi, m->m_microsurfaceheight->invC1(0.99f), U1);

                // next height
                const float U2 = generateRandomNumber();
                const float h_wo = m->sampleHeight(wo, h, U2);

                // leave microsurface
                if(h_wo == FLT_MAX)
                        V += 1.0 / (double)N;
        }

        cout << "analytic = \t" << G2_cond << endl;
        cout << "stochastic = \t" << V << endl;

        delete m;
}
```

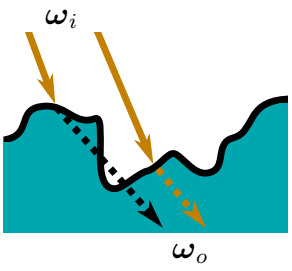
4.1.5 Shadowing Given Masking with Transmission

Same test as in Section 4.1.4 with transmitted direction $\boldsymbol{\omega}_o$:

$$\frac{G_2^{\text{dist}}(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o)}{G_1^{\text{dist}}(\boldsymbol{\omega}_i)} = (1 + \Lambda(\boldsymbol{\omega}_i))\ \mathrm{B}(1 + \Lambda(\boldsymbol{\omega}_i), 1 + \Lambda(\boldsymbol{\omega}_o)).$$

Algorithm 12 Test Conditional Masking-Shadowing

$V \leftarrow 0$
for $n = 1..N$ **do**
  $h = h_{n+1}(-\boldsymbol{\omega}_i, h, \mathcal{U})$
  $h_{\boldsymbol{\omega}_o} = h_{n+1}(\boldsymbol{\omega}_o, h, \mathcal{U})$
  if $h_{\boldsymbol{\omega}_o} = \infty$ **then**
    $V \leftarrow V + \frac{1}{N}$
  end if
end for
return V ▷
$\lim_{N\to\infty} V = (1 + \Lambda(\boldsymbol{\omega}_i))\ \mathrm{B}(1 + \Lambda(\boldsymbol{\omega}_i), 1 + \Lambda(\boldsymbol{\omega}_o))$




```
void test_shadowing_given_masking_transmission()
{
        const vec3 wi = generateRandomDirectionUp();
        const vec3 wo = -generateRandomDirectionUp(); // lower hemisphere
        Microsurface * m = generateRandomMicrosurface();

        // analytic
        float G2_cond = (1.0f + m->m_microsurfaceslope->Lambda(wi)) *
                (float)beta(1.0f + m->m_microsurfaceslope->Lambda(wi), 1.0f + m->m_microsurfaceslope->Lambda(-wo));

        // sampling
        const int N = 100000;
        double V = 0;
        for(int n=0 ; n<N ; ++n)
        {
                // intersect the mirocrsurface from wi
                const float U1 = generateRandomNumber();
                const float h = m->sampleHeight(-wi, m->m_microsurfaceheight->invC1(0.99f), U1);

                // next height
                const float U2 = generateRandomNumber();
                const float h_wo = m->sampleHeight(-wo, -h, U2); // vertical flip

                // leave microsurface
                if(h_wo == FLT_MAX)
                        V += 1.0 / (double)N;
        }

        cout << "analytic = \t" << G2_cond << endl;
        cout << "stochastic = \t" << V << endl;

        delete m;
}
```

4.2 The Distribution of Visible Normals

4.2.1 Normalization

We verify with a quadrature that the VNDF $D_{\boldsymbol{\omega}_i}$ is normalized:

$$\int_{\Omega} D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m)\, d\boldsymbol{\omega}_m = 1.$$

```
// test that \int_\Omega D_wi(wm) dwm = 1
void test_normalization_D_wi()
{
        const vec3 wi = generateRandomDirection();
        Microsurface * m = generateRandomMicrosurface();

        // quadrature \int <wi, wm> D(wm) dwm
        double value_quadrature = 0;
        for(double theta_m=0 ; theta_m < 0.5*M_PI ; theta_m += 0.005)
        for(double phi_m=0 ; phi_m < 2.0*M_PI ; phi_m += 0.005)
        {
                const vec3 wm(cos(phi_m)*sin(theta_m), sin(phi_m)*sin(theta_m), cos(theta_m));
                value_quadrature += 0.005*0.005*abs(sin(theta_m)) * (double)m->m_microsurfaceslope->D_wi(wi, wm);
        }

        // display
        cout << "\\int D_wi(wm) dwm = \t\t" << value_quadrature << endl;

        delete m;
}
```

4.2.2 Importance Sampling

We test the importance sampling procedures of the VNDF $D_{\boldsymbol{\omega}_i}$ by comparing its moments computed with a quadature and importance sampling.

$$\mathbb{E}[x_m] = \int_\Omega x_m \, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m, \qquad \mathbb{E}[y_m^2] = \int_\Omega y_m^2 \, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m$$
$$\mathbb{E}[y_m] = \int_\Omega y_m \, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m, \qquad \mathbb{E}[y_m^2] = \int_\Omega y_m^2 \, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m$$
$$\mathbb{E}[z_m] = \int_\Omega z_m \, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m, \qquad \mathbb{E}[z_m^2] = \int_\Omega z_m^2 \, D_{\boldsymbol{\omega}_i}(\boldsymbol{\omega}_m) \, d\boldsymbol{\omega}_m$$

```
// test the importance sampling of D_wi
void test_sample_D_wi()
{
	const vec3 wi = generateRandomDirection();
	Microsurface * m = generateRandomMicrosurface();

	// quadrature \int <wi, wm> D_wi(wm) f(wm) dwm
	double quadrature_int_x = 0;
	double quadrature_int_y = 0;
	double quadrature_int_z = 0;
	double quadrature_int_x2 = 0;
	double quadrature_int_y2 = 0;
	double quadrature_int_z2 = 0;
	for(double theta_m=0 ; theta_m < 0.5*M_PI ; theta_m += 0.005)
	for(double phi_m=0 ; phi_m < 2.0*M_PI ; phi_m += 0.005)
	{
		const vec3 wm(cos(phi_m)*sin(theta_m), sin(phi_m)*sin(theta_m), cos(theta_m));
		const double d = 0.005*0.005*abs(sin(theta_m)) * (double)m->m_microsurfaceslope->D_wi(wi, wm);
		quadrature_int_x += d * wm.x;
		quadrature_int_y += d * wm.y;
		quadrature_int_z += d * wm.z;
		quadrature_int_x2 += d * wm.x * wm.x;
		quadrature_int_y2 += d * wm.y * wm.y;
		quadrature_int_z2 += d * wm.z * wm.z;
	}

	// sampling \int <wi, wm> D_wi(wm) f(wm) dwm
	double stochastic_int_x = 0;
	double stochastic_int_y = 0;
	double stochastic_int_z = 0;
	double stochastic_int_x2 = 0;
	double stochastic_int_y2 = 0;
	double stochastic_int_z2 = 0;
	for(int n=0 ; n<100000 ; ++n)
	{
		const float U1 = (float)dis(gen);
		const float U2 = (float)dis(gen);
		const vec3 wm = m->m_microsurfaceslope->sampleD_wi(wi, U1, U2);

		stochastic_int_x += wm.x / 100000.0;
		stochastic_int_y += wm.y / 100000.0;
		stochastic_int_z += wm.z / 100000.0;
		stochastic_int_x2 += wm.x * wm.x / 100000.0;
		stochastic_int_y2 += wm.y * wm.y / 100000.0;
		stochastic_int_z2 += wm.z * wm.z / 100000.0;
	}

	// display
	cout << "quadrature \\int D_wi(wm) wm.x dwm = \t\t" << quadrature_int_x << endl;
	cout << "quadrature \\int D_wi(wm) wm.y dwm = \t\t" << quadrature_int_y << endl;
	cout << "quadrature \\int D_wi(wm) wm.z dwm = \t\t" << quadrature_int_z << endl;
	cout << "quadrature \\int D_wi(wm) wm.x^2 dwm = \t\t" << quadrature_int_x2 << endl;
	cout << "quadrature \\int D_wi(wm) wm.x^2 dwm = \t\t" << quadrature_int_y2 << endl;
	cout << "quadrature \\int D_wi(wm) wm.x^2 dwm = \t\t" << quadrature_int_z2 << endl;
	cout << endl;
	cout << "stochastic \\int D_wi(wm) wm.x dwm = \t\t" << stochastic_int_x << endl;
	cout << "stochastic \\int D_wi(wm) wm.y dwm = \t\t" << stochastic_int_y << endl;
	cout << "stochastic \\int D_wi(wm) wm.z dwm = \t\t" << stochastic_int_z << endl;
	cout << "stochastic \\int D_wi(wm) wm.x^2 dwm = \t\t" << stochastic_int_x2 << endl;
	cout << "stochastic \\int D_wi(wm) wm.x^2 dwm = \t\t" << stochastic_int_y2 << endl;
	cout << "stochastic \\int D_wi(wm) wm.x^2 dwm = \t\t" << stochastic_int_z2 << endl;

	delete m;
}
```

4.3 Phase Function of the Microsurface

4.3.1 Energy Conservation

We verify that the phase function is energy conserving (for non-absorptive material):

$$\int_{\Omega} p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o)\, d\boldsymbol{\omega}_o = 1.$$

```
// test that \\int p(wi, wo) dwo = 1
void test_normalization_phasefunction()
{
        const vec3 wi = generateRandomDirection();
        Microsurface * m = generateRandomMicrosurface();

        // quadrature \int p(wi, wo) dwo
        double value_quadrature = 0;
        for(double theta_o=0 ; theta_o < M_PI ; theta_o += 0.005)
        for(double phi_o=0 ; phi_o < 2.0*M_PI ; phi_o += 0.005)
        {
                const vec3 wo(cos(phi_o)*sin(theta_o), sin(phi_o)*sin(theta_o), cos(theta_o));
                value_quadrature += 0.005*0.005*abs(sin(theta_o)) * (double)m->evalPhaseFunction(wi, wo);
        }

        // display
        cout << "\\int p(wi, wo) dwo = \t\t" << value_quadrature << endl;

        delete m;
}
```

4.3.2 Reciprocity

We verify that the phase function satisfies the microflake reciprocity constraint, i.e.

$$\sigma^{\text{microflake}}(\boldsymbol{\omega}_i)\, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = \sigma^{\text{microflake}}(\boldsymbol{\omega}_o)\, p(\boldsymbol{\omega}_o, \boldsymbol{\omega}_i),$$

which in our case becomes

$$\begin{aligned} \sigma^{\text{Smith}}(-\boldsymbol{\omega}_i)\, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) &= \sigma^{\text{Smith}}(-\boldsymbol{\omega}_o)\, p(\boldsymbol{\omega}_o, \boldsymbol{\omega}_i) \\ (1 + \Lambda(\boldsymbol{\omega}_i))\, \cos\theta_i\, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) &= (1 + \Lambda(\boldsymbol{\omega}_o))\, \cos\theta_o\, p(\boldsymbol{\omega}_o, \boldsymbol{\omega}_i) \end{aligned}$$

```
// test that (1+Lambda(wi)) cos(theta_i) f_p(wi, wo) = (1+Lambda(wo)) cos(theta_o) f_p(wo, wi)
void test_reciprocity_phasefunction()
{
        const vec3 wi = generateRandomDirectionUp();
        const vec3 wo = generateRandomDirectionUp();
        Microsurface * m = generateRandomMicrosurface();

        float Li = m->m_microsurfaceslope->Lambda(wi);
        float Lo = m->m_microsurfaceslope->Lambda(wo);
        float fi = m->evalPhaseFunction(wi, wo);
        float fo = m->evalPhaseFunction(wo, wi);

        const float value_wi_wo = (1.0f + m->m_microsurfaceslope->Lambda(wi)) * wi.z * m->evalPhaseFunction(wi, wo);
        const float value_wo_wi = (1.0f + m->m_microsurfaceslope->Lambda(wo)) * wo.z * m->evalPhaseFunction(wo, wi);

        // display
        cout << "Lambda(wi) cos(theta_i) f_p(wi, wo) = \t\t" << value_wi_wo << endl;
        cout << "Lambda(wo) cos(theta_o) f_p(wo, wi) = \t\t" << value_wo_wi << endl;

        delete m;
}
```

4.3.3 Importance Sampling

We test the importance sampling procedures of the phase function f_p by comparing its moments computed with a quadature and importance sampling.

$$\mathbb{E}[x_o] = \int_\Omega x_o \, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, d\boldsymbol{\omega}_o, \qquad \mathbb{E}[y_o^2] = \int_\Omega y_o^2 \, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, d\boldsymbol{\omega}_o$$
$$\mathbb{E}[y_o] = \int_\Omega y_o \, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, d\boldsymbol{\omega}_o, \qquad \mathbb{E}[y_o^2] = \int_\Omega y_o^2 \, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, d\boldsymbol{\omega}_o$$
$$\mathbb{E}[z_o] = \int_\Omega z_o \, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, d\boldsymbol{\omega}_o, \qquad \mathbb{E}[z_o^2] = \int_\Omega z_o^2 \, p(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, d\boldsymbol{\omega}_o$$

```
// test the importance sampling of p
void test_sample_phasefunction()
{
	const vec3 wi = generateRandomDirection();
	Microsurface * m = generateRandomMicrosurface();

	if(m->m_microsurfaceslope->projectedArea(wi) < 0.01f)
	{
		cout << "Warning: the projected area is too small" << endl;
		cout << "the ray cannot intersect the microsurface in this configuration" << endl;
		cout << "and the phase function should not be called." << endl << endl;
	}

	// quadrature with eval p(wi, wo)
	double quadrature_int_x = 0;
	double quadrature_int_y = 0;
	double quadrature_int_z = 0;
	double quadrature_int_x2 = 0;
	double quadrature_int_y2 = 0;
	double quadrature_int_z2 = 0;
	for(double theta_o=0 ; theta_o < M_PI ; theta_o += 0.005)
	for(double phi_o=0 ; phi_o < 2.0*M_PI ; phi_o += 0.005)
	{
		const vec3 wo(cos(phi_o)*sin(theta_o), sin(phi_o)*sin(theta_o), cos(theta_o));
		const double d = 0.005*0.005*abs(sin(theta_o)) * (double)m->evalPhaseFunction(wi, wo);
		quadrature_int_x += d * wo.x;
		quadrature_int_y += d * wo.y;
		quadrature_int_z += d * wo.z;
		quadrature_int_x2 += d * wo.x * wo.x;
		quadrature_int_y2 += d * wo.y * wo.y;
		quadrature_int_z2 += d * wo.z * wo.z;
	}

	// sampling \p(wi, wo)
	const int N = 100000;
	double stochastic_int_x = 0;
	double stochastic_int_y = 0;
	double stochastic_int_z = 0;
	double stochastic_int_x2 = 0;
	double stochastic_int_y2 = 0;
	double stochastic_int_z2 = 0;
	for(int n=0 ; n<N ; ++n)
	{
		const vec3 wo = m->samplePhaseFunction(wi);

		stochastic_int_x += wo.x / (double) N;
		stochastic_int_y += wo.y / (double) N;
		stochastic_int_z += wo.z / (double) N;
		stochastic_int_x2 += wo.x * wo.x / (double) N;
		stochastic_int_y2 += wo.y * wo.y / (double) N;
		stochastic_int_z2 += wo.z * wo.z / (double) N;



	}

	// display
	cout << "quadrature \\int p(wi, wo) wo.x dwo = \t\t" << quadrature_int_x << endl;
	cout << "quadrature \\int p(wi, wo) wo.y dwo = \t\t" << quadrature_int_y << endl;
	cout << "quadrature \\int p(wi, wo) wo.z dwo = \t\t" << quadrature_int_z << endl;
	cout << "quadrature \\int p(wi, wo) wo.x^2 dwo = \t" << quadrature_int_x2 << endl;
	cout << "quadrature \\int p(wi, wo) wo.x^2 dwo = \t" << quadrature_int_y2 << endl;
	cout << "quadrature \\int p(wi, wo) wo.x^2 dwo = \t" << quadrature_int_z2 << endl;
	cout << endl;
	cout << "stochastic \\int p(wi, wo) wo.x dwo = \t\t" << stochastic_int_x << endl;
	cout << "stochastic \\int p(wi, wo) wo.y dwo = \t\t" << stochastic_int_y << endl;
```

```
        cout << "stochastic␣\\int␣p(wi,␣wo)␣wo.z␣dwo␣=␣\t\t" << stochastic_int_z << endl;
        cout << "stochastic␣\\int␣p(wi,␣wo)␣wo.x^2␣dwo␣=␣\t" << stochastic_int_x2 << endl;
        cout << "stochastic␣\\int␣p(wi,␣wo)␣wo.x^2␣dwo␣=␣\t" << stochastic_int_y2 << endl;
        cout << "stochastic␣\\int␣p(wi,␣wo)␣wo.x^2␣dwo␣=␣\t" << stochastic_int_z2 << endl;

        delete m;
}
```

4.4 The Multiple Scattering BSDF

4.4.1 Energy Conservation

We verify with a quadrature that the phase functions are normalized:

$$\int_{\Omega} f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \cos\theta_o \, d\boldsymbol{\omega}_o = 1.$$

```
// test that \int f(wi, wo) dwo = 1
void test_normalization_bsdf()
{
        const vec3 wi = generateRandomDirectionUp();
        Microsurface * m = generateRandomMicrosurface();

        // quadrature \int f_p(wi, wo) dwo
        double value_quadrature = 0;
        for(double theta_o=0 ; theta_o < M_PI ; theta_o += 0.005)
        for(double phi_o=0 ; phi_o < 2.0*M_PI ; phi_o += 0.005)
        {
                const vec3 wo(cos(phi_o)*sin(theta_o), sin(phi_o)*sin(theta_o), cos(theta_o));

                // stochastic evaluation
                const int N = 10;
                double value_current = 0;
                for(int n=0 ; n<N ; ++n)
                {
                        value_current += (double)m->eval(wi, wo) / (double) N;
                }

                value_quadrature += 0.005*0.005*abs(sin(theta_o)) * value_current;
        }

        // display
        cout << "\\int f_p(wi, wo) dwo = \t\t" << value_quadrature << endl;

        delete m;
}
```

Note that the **eval**() procedure returns the cosine weighted BSDF $f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \cos\theta_o$.

4.4.2 Reciprocity

We verify that the multiple scattering BSDF model is reciprocal:

$$f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = f(\boldsymbol{\omega}_o, \boldsymbol{\omega}_i).$$

```
// test that f(wi, wo) = f(wo, wi)
void test_reciprocity_bsdf()
{
        const vec3 wi = generateRandomDirectionUp();
        const vec3 wo = generateRandomDirectionUp();
        Microsurface * m = generateRandomMicrosurface();

        // stochastic evaluation of f(wi, wo)
        const int N = 1000000;
        double f_wi_wo = 0;
        for(int n=0 ; n<N ; ++n)
        {
                f_wi_wo += (double)m->eval(wi, wo) / (double) N;
        }

        // stochastic evaluation of f(wi, wo)
        double f_wo_wi = 0;
        for(int n=0 ; n<N ; ++n)
        {
                f_wo_wi += (double)m->eval(wo, wi) / (double) N;
        }

        // display
        cout << "f(wi, wo) = \t\t" << f_wi_wo / fabsf(wo.z) << endl;
        cout << "f(wo, wi) = \t\t" << f_wo_wi / fabsf(wi.z) << endl;

        delete m;
}
```

Note that the **eval**() procedure returns the cosine weighted BSDF $f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \cos\theta_o$. This is why we divide by wi.x $= \cos\theta_i$ and wo.x $= \cos\theta_o$.

4.4.3 Single Scattering

We verify that the random walk evaluation clamped to first scattering event returns the same value than the classic single scattering BSDF models.

```
void test_single_scattering()
{
	const vec3 wi = vec3(0,0,1);generateRandomDirectionUp();
	const vec3 wo = generateRandomDirectionUp();
	Microsurface * m = generateRandomMicrosurface();

	const int N = 100000;

	// eval truncated random walk (loop because eval is stochastic)
	double V = 0;
	for(int n=0 ; n<N ; ++n)
	{
		V += m->eval(wi, wo, 1) / (double)N;
	}

	// eval single (loop because single of diffuse is also stochastic)
	double V_single = 0;
	for(int n=0 ; n<N ; ++n)
	{
		V_single += m->evalSingleScattering(wi, wo) / (double)N;
	}

	cout << "random walk cut after 1st bounce = \t" << V << endl;
	cout << "single scattering = \t" << V_single << endl;

	delete m;
}
```

4.4.4 Importance Sampling

We test the importance sampling procedures of the BSDF f by comparing its moments computed with a quadature and importance sampling.

$$\mathbb{E}[x_o] = \int_\Omega x_o \, f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, \cos\theta_o \, d\boldsymbol{\omega}_o, \qquad \mathbb{E}[y_o^2] = \int_\Omega y_o^2 \, f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, \cos\theta_o \, d\boldsymbol{\omega}_o$$

$$\mathbb{E}[y_o] = \int_\Omega y_o \, f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, \cos\theta_o \, d\boldsymbol{\omega}_o, \qquad \mathbb{E}[y_o^2] = \int_\Omega y_o^2 \, f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, \cos\theta_o \, d\boldsymbol{\omega}_o$$

$$\mathbb{E}[z_o] = \int_\Omega z_o \, f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, \cos\theta_o \, d\boldsymbol{\omega}_o, \qquad \mathbb{E}[z_o^2] = \int_\Omega z_o^2 \, f(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) \, \cos\theta_o \, d\boldsymbol{\omega}_o$$

```
// test the importance sampling of the BSDF
void test_sample_bsdf()
{
	const vec3 wi = generateRandomDirectionUp();
	Microsurface * m = generateRandomMicrosurface();

	// quadrature \int f(wi, wo) f(wo) dwo
	double quadrature_int_x = 0;
	double quadrature_int_y = 0;
	double quadrature_int_z = 0;
	double quadrature_int_x2 = 0;
	double quadrature_int_y2 = 0;
	double quadrature_int_z2 = 0;
	for(double theta_o=0 ; theta_o < M_PI ; theta_o += 0.005)
	for(double phi_o=0 ; phi_o < 2.0*M_PI ; phi_o += 0.005)
	{
		const vec3 wo(cos(phi_o)*sin(theta_o), sin(phi_o)*sin(theta_o), cos(theta_o));

		// stochastic evaluation
		const int N = 10;
		double value_current = 0;
		for(int n=0 ; n<N ; ++n)
		{
			value_current += (double)m->eval(wi, wo) / (double) N;
		}

		const double d = 0.005*0.005*abs(sin(theta_o)) * value_current;
		quadrature_int_x += d * wo.x;
		quadrature_int_y += d * wo.y;
		quadrature_int_z += d * wo.z;
		quadrature_int_x2 += d * wo.x * wo.x;
		quadrature_int_y2 += d * wo.y * wo.y;
		quadrature_int_z2 += d * wo.z * wo.z;
	}

	// sampling \int f_p(wo) f(wo) dwo
	const int N = 100000;
	double stochastic_int_x = 0;
	double stochastic_int_y = 0;
	double stochastic_int_z = 0;
	double stochastic_int_x2 = 0;
	double stochastic_int_y2 = 0;
	double stochastic_int_z2 = 0;
	for(int n=0 ; n<N ; ++n)
	{
		const float U1 = (float)dis(gen);
		const float U2 = (float)dis(gen);
		const vec3 wo = m->sample(wi);

		stochastic_int_x += wo.x / (double) N;
		stochastic_int_y += wo.y / (double) N;
		stochastic_int_z += wo.z / (double) N;
		stochastic_int_x2 += wo.x * wo.x / (double) N;
		stochastic_int_y2 += wo.y * wo.y / (double) N;
		stochastic_int_z2 += wo.z * wo.z / (double) N;
	}

	// display
	cout << "quadrature \\int f_p(wo) wo.x dwo = \t\t" << quadrature_int_x << endl;
	cout << "quadrature \\int f_p(wo) wo.y dwo = \t\t" << quadrature_int_y << endl;
	cout << "quadrature \\int f_p(wo) wo.z dwo = \t\t" << quadrature_int_z << endl;
	cout << "quadrature \\int f_p(wo) wo.x^2 dwo = \t\t" << quadrature_int_x2 << endl;
	cout << "quadrature \\int f_p(wo) wo.y^2 dwo = \t\t" << quadrature_int_y2 << endl;
	cout << "quadrature \\int f_p(wo) wo.z^2 dwo = \t\t" << quadrature_int_z2 << endl;
	cout << endl;
	cout << "stochastic \\int f_p(wo) wo.x dwo = \t\t" << stochastic_int_x << endl;
```

```
        cout << "stochastic␣\\int␣f_p(wo)␣wo.y␣dwo␣=␣\t\t" << stochastic_int_y << endl;
        cout << "stochastic␣\\int␣f_p(wo)␣wo.z␣dwo␣=␣\t\t" << stochastic_int_z << endl;
        cout << "stochastic␣\\int␣f_p(wo)␣wo.x^2␣dwo␣=␣\t\t" << stochastic_int_x2 << endl;
        cout << "stochastic␣\\int␣f_p(wo)␣wo.y^2␣dwo␣=␣\t\t" << stochastic_int_y2 << endl;
        cout << "stochastic␣\\int␣f_p(wo)␣wo.z^2␣dwo␣=␣\t\t" << stochastic_int_z2 << endl;

        delete m;
}
```

References

[HD14] Eric Heitz and Eugene D'Eon. Importance sampling microfacet-based BSDFs using the distribution of visible normals. In *Proc. Eurographics Symposium on Rendering*, pages 103–112, 2014. 2.1, 2.3

[Hei14] Eric Heitz. Understanding the masking-shadowing function in microfacet-based BRDFs. *Journal of Computer Graphics Techniques*, 3(2):32–91, 2014. 2.3

[Jak14] Wenzel Jakob. An improved visible normal sampling routine for the beckmann distribution. Technical report, ETH Zürich, August 2014. 2.2