Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Petr Vévoda

Robust light transport simulation in participating media

Department of Software and Computer Science Education

Supervisor of the master thesis: Jaroslav Křivánek
Study programme: Informatics
Specialization: Computer graphics

Prague 2014

Here I would like to thank Jaroslav Křivánek for supervising my thesis, Martin Šik for cooperation on the implementation of the UPBP algorithm and Iliyan Georgiev, Toshiya Hachisuka, Derek Nowrouzezahrai, and Wojciech Jarosz for advices and suggestions on improving the implementation. I would also like to thank Iliyan Georgiev and Tomáš Davidovič for providing support for the SmallVCM project, and Chaos Group, Toshiya Hachisuka and Ondřej Karlík for creating and sharing the test scenes.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Robust light transport simulation in participating media

Autor: Petr Vévoda

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: doc. Ing. Jaroslav Křivánek, Ph.D., Kabinet software a výuky informatiky

Abstrakt: Simulace přenosu světla je využívána v realistické syntéze obrazu k vytváření fyzikálně věrných obrazů virtuálních scén. Důležitou součástí scén bývají opticky aktivní média (např. vzduch, voda, kůže). Efektivní výpočet přenosu světla v médiích robustní vůči jejich velké rozmanitosti je dosud otevřený problém. Naimplementovali jsme algoritmus UPBP nově vyvinutý Křivánkem et al. Ten řeší tento problém kombinací několika vzájemně se doplňujících předešlých metod s pomocí vzorkování podle násobné důležitosti a vyniká v zobrazování scén, v nichž tyto metody samotné selhávaly. Tato implementace je dostupná online, soustředili jsme se na její důkladný popis, abychom usnadnili a podpořili další výzkum v této oblasti.

Klíčová slova: simulace přenosu světla, syntéza obrazu, opticky aktivní média, vzorkování podle násobné důležitosti, kombinování estimátorů

Title: Robust light transport simulation in participating media

Author: Petr Vévoda

Department: Department of Software and Computer Science Education

Supervisor: doc. Ing. Jaroslav Křivánek, Ph.D., Department of Software and Computer Science Education

Abstract: Light transport simulation is used in realistic image synthesis to create physically plausible images of virtual scenes. Important components of the scenes are participating media (e.g. air, water, skin etc.). Efficient computation of light transport in participating media robust to their large diversity is still an open problem. We implemented the UPBP algorithm recently developed by Křivánek et al. It addresses the problem by combining several complementary previous methods using multiple importance sampling, and excels at rendering scenes where the previous methods alone fail. The implementation is available online, we focused on its thorough description to facilitate and support further research in this field.

Keywords: light transport simulation, image synthesis, participating media, multiple importance sampling, combining estimators

Contents

Introduction

One of the significant areas of computer graphics is realistic image synthesis. Its goal is creating an image of a virtual scene indistinguishable from a photography. To achieve physically plausible results it simulates light transport in the scene, i.e. how light propagates through the space after emitting from a light source till reaching a human eye or a camera sensor. The process of synthesizing images by means of computer programs is often called rendering.

Modern realistic image synthesis aims to reproduce a wide range of lighting effects, including the interaction of light with participating media, e.g. light scattering in fog, smoke, wax, skin or liquids (see Figure 1). However, faithfully simulating light transport in media can incur a large computational cost as variations in media density (e.g., haze vs. skin), scattering albedo (wine vs. milk), and scattering anisotropy (air vs. dust) result in significantly different light behaviour. As such, designing a single light transport simulation algorithm that is *robust* to these variations remains an open problem, which is important not only in computer graphics but also across many other diverse fields, such as medical imaging or nuclear physics.

Two classes of widely adopted approaches excel at rendering complex volumetric shading effects: those based on Monte Carlo (MC) estimation of the path integral [5] and those based on photon density estimation [10]. None of them alone is perfect though. Several different photon density estimators focus on handling complex effects such as indirect caustics, where bidirectional path-tracing (BPT) [15, 27], the main representative of the former group, performs poorly. On the other hand, BPT is unbiased, general and better captures e.g. direct illumination in media far away from lights.

Křivánek et al. [14] sought to combine the strengths of volumetric photon density estimators with the versatility of BPT in a principled way. The resulting algorithm called *unified points, beams, and paths (UPBP)* excels at rendering scenes with different kinds of media, where previous techniques each fail in complementary ways. It also naturally incorporates a combination of BPT and surface photon density estimator (surface photon mapping) [12], which was previously described by Georgiev et al. [4] and Hachisuka et al. [7].

We collaborated with Křivánek on the UPBP algorithm and created its implementation, which provided evidence for the qualities of the algorithm in practice and was used for generating results in the UPBP paper [14] presented at the SIGGRAPH 2014 conference [23]. The result of our work, a renderer called SmallUPBP, is released online [24] and can be used and modified freely.

This thesis presents the implementation. It explains how we extended the code of the SmallVCM project [25], describes the infrastructure we had to build to add media support, what methods and data structures we used for volumetric photon density estimators and how we dealt with the key aspect of the implementation - computation of multiple importance sampling (MIS) weights.

The thesis has four chapters, the first two are theoretical, the remaining two describe implementation. We begin with a review of the theory related to BPT and photon density estimators in Chapter 1 and introduce our approach to their combination in Chapter 2. Chapter 3, the main chapter of the thesis, then focuses

on our work, i.e. the implementation of the UPBP algorithm. Finally, capabilities of our program are demonstrated in Chapter 4.

The implementation can be found on the attached DVD, its contents are listed in Attachment 3. Basic information about compilation, running and controlling the program are provided in Attachment 1.

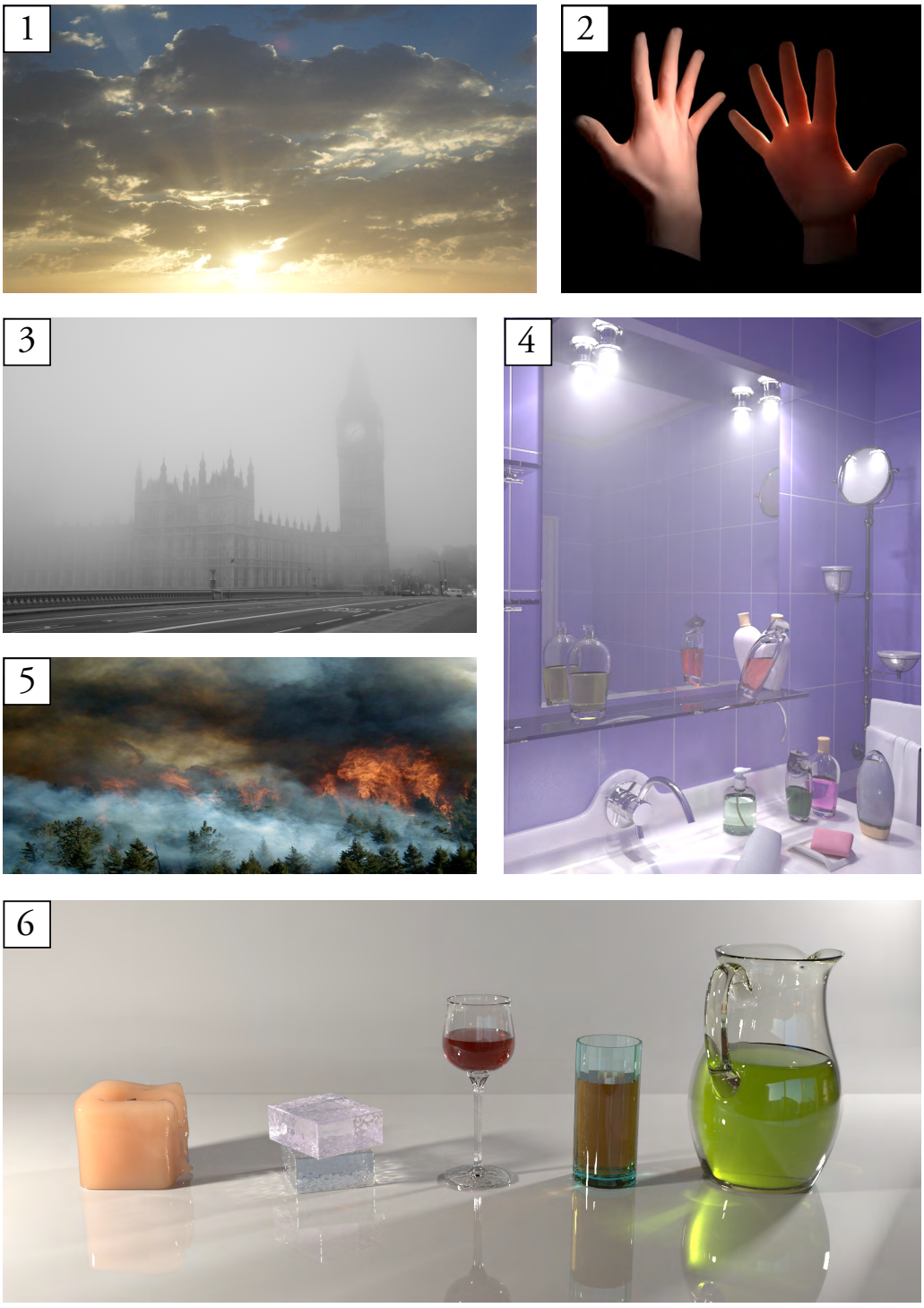


Figure 1: A few examples of participating media. Images 1, 3, and 5 are real photos of clouds, fog and fire with smoke, respectively. Images 2, 4 and 6 are computer generated. Image 2 shows subsurface scattering in skin. Image 4 captures a steamy bathroom with several flasks filled with different media. Image 6 shows from left to right: wax candle, glycerin soap bar on top of a block of a back-scattering medium, diluted wine, apple juice, and olive oil. Images 4 and 6 were rendered using the UPBP algorithm [14]. Source: Image 1: `http://commons.wikimedia.org/wiki/File:Sky_Riyadh.jpg`, Image 2: `http://www.mrbluesummers.com/3510`, Image 3: `http://www.panoramio.com/photo/14455719`, Image 4: [14], Image 5: `http://thewmpa.org/resources/forest-fire-info`, Image 6: [14].

1. Background

A necessary theoretical background is presented in this chapter. We define our notation and review the theory of both estimator families (the path integral formulation of light transport for MC estimators and the volumetric photon density estimation). This chapter contains modified text of Section 4 in [14].

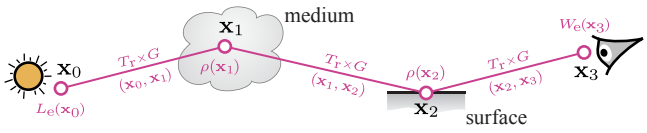
1.1 Path integral formulation

Intuitively, a sensed image of a scene is made by light emitted from a light source that travelled through the scene, got scattered on surfaces and/or in media and finally hit our eyes. Light can follow many different paths from a light source to the eye and if we sum up contributions of all such paths, we get a complete image of the scene. Formally, the path integral framework [26, 20] expresses image pixel intensity I as an integral over the space Ω of light transport paths: $I = \int_\Omega f(\overline{\mathbf{x}})\,\mathrm{d}\mu(\overline{\mathbf{x}})$. A length-$k$ path $\overline{\mathbf{x}} = \mathbf{x}_0 \ldots \mathbf{x}_k \in \Omega$ has $k \geq 1$ segments and $k+1$ vertices, with its first vertex $\mathbf{x}_0$ on a light source, its last vertex $\mathbf{x}_k$ on the eye/camera[1] sensor, and the $\mathbf{x}_1 \ldots \mathbf{x}_{k-1}$ inner scattering vertices on surfaces and/or in media. The differential path measure $\mathrm{d}\mu(\overline{\mathbf{x}})$ is a product measure corresponding to area and volume integration for surface and medium vertices, respectively. The measurement contribution function $f(\overline{\mathbf{x}})$ measures contribution of the path $\overline{\mathbf{x}}$ to the image and it is the product of emitted radiance $L_\mathrm{e}(\mathbf{x}_0) = L_\mathrm{e}(\mathbf{x}_0 \to \mathbf{x}_1)$[2], path throughput $T(\overline{\mathbf{x}})$, and sensor sensitivity $W_\mathrm{e}(\mathbf{x}_k) = W_\mathrm{e}(\mathbf{x}_{k-1} \to \mathbf{x}_k)$:

$$f(\overline{\mathbf{x}}) = L_\mathrm{e}(\mathbf{x}_0)\, T(\overline{\mathbf{x}})\, W_\mathrm{e}(\mathbf{x}_k). \tag{1.1}$$

The path throughput $T(\overline{\mathbf{x}})$ determines how much of the emitted light reaches the sensor. It is the product of the geometry and transmittance terms for path segments, and scattering function for the inner path vertices, as expressed and illustrated below:

$$T(\overline{\mathbf{x}}) = \left[\prod_{i=0}^{k-1} G(\mathbf{x}_i, \mathbf{x}_{i+1}) T_\mathrm{r}(\mathbf{x}_i, \mathbf{x}_{i+1})\right]\left[\prod_{i=1}^{k-1} \rho(\mathbf{x}_i)\right]. \tag{1.2}$$



The geometry term for a path segment $\mathbf{xy}$ is given by

$$G(\mathbf{x}, \mathbf{y}) = V(\mathbf{x}, \mathbf{y}) \frac{D(\mathbf{x} \to \mathbf{y}) D(\mathbf{y} \to \mathbf{x})}{\|\mathbf{x} - \mathbf{y}\|^2}, \tag{1.3}$$

[1] The words "camera" and "eye" are interchangeable in this context. We use mainly the first one since the implementation uses it. Křivánek et al. [14] and our images use "eye".

[2] Light transport quantities are often directional dependent. We use a common notation with the arrow sign "$\to$" to clearly and simply identify the intended direction.

where $D(\mathbf{x} \to \mathbf{y}) = |n_\mathbf{x} \cdot \omega_\mathbf{xy}|$ if $\mathbf{x}$ is on a surface, and $D(\mathbf{x} \to \mathbf{y}) = 1$ if $\mathbf{x}$ is in a medium, and likewise for $D(\mathbf{y} \to \mathbf{x})$. Here $n_\mathbf{x}$ is the surface normal at $\mathbf{x}$ and $\omega_\mathbf{xy}$ is a unit-length vector from $\mathbf{x}$ to $\mathbf{y}$. $V(\mathbf{x}, \mathbf{y})$ is the visibility indicator function, $V(\mathbf{x}, \mathbf{y}) = 1$, if $\mathbf{y}$ is directly visible from $\mathbf{x}$ (i.e. no geometry blocks a ray connecting these two vertices), otherwise $V(\mathbf{x}, \mathbf{y}) = 0$. The transmittance of segment $\mathbf{xy}$ captures attenuation by intersected media and is given by

$$T_\mathrm{r}(\mathbf{x}, \mathbf{y}) = \exp\Big(-\int_0^{\|\mathbf{x}-\mathbf{y}\|} \sigma_\mathrm{t}(\mathbf{x} + t\omega_\mathbf{xy})\,\mathrm{d}t\Big). \tag{1.4}$$

We will often write only $T_\mathrm{r}(t)$ where $t = \|\mathbf{x} - \mathbf{y}\|$, if $\mathbf{x}, \mathbf{y}$ are obvious or not important. Finally, the scattering function describes effects of a surface or medium at a path vertex. We define it as

$$\rho(\mathbf{x}_i) = \begin{cases} \rho_\mathrm{s}(\mathbf{x}_{i-1} \to \mathbf{x}_i \to \mathbf{x}_{i+1}) & \text{if } \mathbf{x}_i \text{ on surface} \\ \rho_\mathrm{p}(\mathbf{x}_{i-1} \to \mathbf{x}_i \to \mathbf{x}_{i+1})\sigma_\mathrm{s}(\mathbf{x}_i) & \text{if } \mathbf{x}_i \text{ in medium,} \end{cases} \tag{1.5}$$

where ρ_s and ρ_p are the bidirectional scattering distribution function (BSDF) and phase function, respectively. σ_s and σ_t denote the scattering and attenuation (extinction) coefficients.

1.1.1 MC estimators

The path integral can be evaluated with an unbiased MC estimator $\langle I \rangle = \frac{1}{m}\sum_{j=1}^{m} f(\overline{\mathbf{x}}_j)/p(\overline{\mathbf{x}}_j)$ that averages estimates from m random paths $\overline{\mathbf{x}}_j$ sampled using a *path sampling technique* with probability distribution $p(\overline{\mathbf{x}})\,\mathrm{d}\mu(\overline{\mathbf{x}})$. The path pdf $p(\overline{\mathbf{x}})$ is given by the joint density of the individual path vertices, i.e., $p(\overline{\mathbf{x}}) = p(\mathbf{x}_0, \ldots, \mathbf{x}_k)$, and it is determined by the path sampling technique employed to generate the path. We use bidirectional path tracing (BPT) which generates paths by independently sampling one subpath from a light and another from the camera, optionally connecting them with an edge. The different path sampling techniques in BPT for generating a given path correspond to the different lengths of the light and camera subpaths. The full path pdf is then given by the product of the pdfs for the two subpaths, $p(\overline{\mathbf{x}}) = p(\mathbf{x}_0 \ldots \mathbf{x}_s)p(\mathbf{x}_t \ldots \mathbf{x}_k)$. The subpath pdf reflects the local sampling techniques used to generate the individual subpath vertices, and can be written as a product of vertex pdfs $p(\mathbf{x}_i|\text{vertices sampled before } \mathbf{x}_i)$.

In our notation, we express directional pdfs $p(\omega)$ w.r.t. the projected solid angle measure, $\hat{p}(\omega)$ w.r.t. the (non-projected) solid angle measure, distance pdfs $p(t)$ w.r.t. the Euclidean length on $\mathbb{R}^1$, and volume vertex pdfs $p(\mathbf{x})$ w.r.t. the Euclidean volume on $\mathbb{R}^3$. In participating media, converting from the projected solid angle $\times$ length product measure to the volume measure involves multiplication by the geometry term $G(\mathbf{x}, \mathbf{y})$. Converting from the solid angle $\times$ length product measure to the volume measure lacks factor $D(\mathbf{x} \to \mathbf{y})$ (since $\hat{p}(\omega_\mathbf{xy}) = p(\omega_\mathbf{xy})D(\mathbf{x} \to \mathbf{y})$).

We define the *subpath contribution*, or *weight*, for light and camera subpaths as the partial evaluation of a path integral estimator:

$$C_\mathrm{l}(\mathbf{x}_0 \ldots \mathbf{x}_i) = L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \ldots \mathbf{x}_i)}{p(\mathbf{x}_0 \ldots \mathbf{x}_i)} \tag{1.6}$$

$$C_\mathrm{c}(\mathbf{x}_j \ldots \mathbf{x}_k) = \frac{T(\mathbf{x}_j \ldots \mathbf{x}_k)}{p(\mathbf{x}_j \ldots \mathbf{x}_k)}W_\mathrm{e}(\mathbf{x}_k). \tag{1.7}$$

1.2 Volumetric photon density estimation

Jarosz et al. [10] introduced several distinct volumetric photon density estimators that differ in the representation of equilibrium radiance they employ (point samples or "photons"/trajectory samples or "photon beams"), the radiometric quantities they estimate (in-scattered radiance at a point/integrated radiance along a ray), and the dimension of blur employed when reconstructing the desired quantity from the samples (1D/2D/3D). If we should describe this second approach in a nutshell, it is based on distribution of light in a form of photons and beams all over a scene and then averaging contributions of photons and beams found around a place of interest. In contrast to the aforementioned unbiased path integral estimators, Jarosz et al.'s volumetric photon density estimators are *not* given as general path sampling techniques, i.e. they have no notion of path contribution function and path pdf. Such a formulation is provided by Křivánek et al. [14] and used in our implementation.

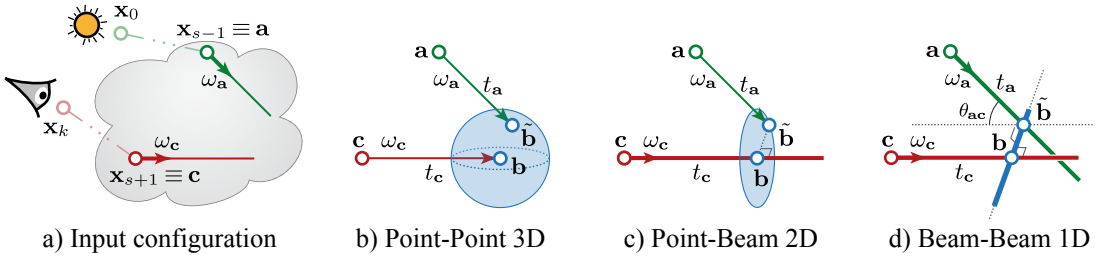


Figure 1.1: Illustration of the used volumetric radiance estimators.

Fig. 1.1a shows the shared geometric setup for the Křivánek et al.'s formulation of the volumetric photon density estimators. A light subpath extends up to a vertex $\mathbf{x}_{s-1}$, hereafter denoted as $\mathbf{a}$ for brevity, and a direction $\omega_\mathbf{a}$ is sampled from $\mathbf{a}$. The ray $(\mathbf{a}, \omega_\mathbf{a})$ defines a *photon beam*, whose energy is given by the light subpath weight including scattering at $\mathbf{a}$ (evaluating the scattering function and dividing by the probability of sampling the ray direction):

$$C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{a}] = C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{a}) \frac{\rho(\mathbf{a})}{p(\omega_\mathbf{a})}. \tag{1.8}$$

Similarly, a camera subpath extends up to a vertex $\mathbf{x}_{s+1}$, denoted $\mathbf{c}$, and a direction $\omega_\mathbf{c}$ is sampled from $\mathbf{c}$. The ray $(\mathbf{c}, \omega_\mathbf{c})$ defines a *query beam* with weight w.r.t. the pixel estimate given by

$$C_\mathrm{c}[\mathbf{c} \dots \mathbf{x}_k) = \frac{\rho(\mathbf{c})}{p(\omega_\mathbf{c})} C_\mathrm{c}(\mathbf{c} \dots \mathbf{x}_k). \tag{1.9}$$

By sampling a distance $t_\mathbf{a}$ along the ray $(\mathbf{a}, \omega_\mathbf{a})$, one could create a *photon* at position $\tilde{\mathbf{b}} \equiv \tilde{\mathbf{x}}_s$ with weight $C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{a}] \frac{T_\mathrm{r}(t_\mathbf{a})}{p(t_\mathbf{a})}$ (it is the photon beam energy attenuated by transmittance along the beam and amplified by probability of sampling its length). Similarly a *query point* at $\mathbf{b} \equiv \mathbf{x}_s$ could be created by sampling a distance $t_\mathbf{c}$ along the ray $(\mathbf{c}, \omega_\mathbf{c})$. Instead, Křivánek et al. [14] treated the photon beam $(\mathbf{a}, \omega_\mathbf{a})$ and the query beam $(\mathbf{c}, \omega_\mathbf{c})$ as the common input to all the estimators, and included the terms involved with the calculation of the photon or query point weights into the estimator expressions themselves. With

this convention in place, every estimator operates on the same shared input, shown in Fig. 1.1a.

One remark about the notation. If meaning of $\omega_{\mathbf{x}}$ would not be clear from the context, we use $\omega_{\mathbf{xy}}$ to denote a direction from $\mathbf{x}$ to $\mathbf{y}$. If the second vertex is not yet known, $\omega_{\mathbf{x}\rightarrow}$ denotes a direction sampled from light vertex $\mathbf{x}$ (i.e. "towards the camera") and $\omega_{\leftarrow\mathbf{y}}$ a direction sampled from camera vertex $\mathbf{y}$ (i.e. "towards the light "). Similarly, we add some notion of direction to sampled distance t. We use $t_{\mathbf{xy}}$ to denote a distance sampled along the ray $(\mathbf{x}, \omega_{\mathbf{xy}})$. If the second vertex is not yet known, $t_{\mathbf{x}\rightarrow}$ denotes a a distance sampled along the ray $(\mathbf{x}, \omega_{\mathbf{x}\rightarrow})$ and $t_{\leftarrow\mathbf{y}}$ a distance sampled along the ray $(\mathbf{y}, \omega_{\leftarrow\mathbf{y}})$. This way we distinguish later in this text between pdfs/probabilities of sampling a distance of the same path segment as if it was sampled on the way from the light and as if it was sampled on the way from the camera.

$$\langle I \rangle_{\text{P-P3D}} = \underbrace{\frac{T_{\text{r}}(t_{\mathbf{a}})}{p(t_{\mathbf{a}})}}_{\text{photon sampling}} \rho(\tilde{\mathbf{b}}, \mathbf{b})\, K_3(\tilde{\mathbf{b}}, \mathbf{b}) \underbrace{\frac{T_{\text{r}}(t_{\mathbf{c}})}{p(t_{\mathbf{c}})}}_{\text{query point sampling}} \tag{1.10}$$

$$\langle I \rangle_{\text{P-B2D}} = \underbrace{\frac{T_{\text{r}}(t_{\mathbf{a}})}{p(t_{\mathbf{a}})}}_{\text{photon sampling}} \rho(\tilde{\mathbf{b}}, \mathbf{b})\, K_2(\tilde{\mathbf{b}}, \mathbf{b})\, T_{\text{r}}(t_{\mathbf{c}}) \tag{1.11}$$

$$\langle I \rangle_{\text{B-B1D}} = T_{\text{r}}(t_{\mathbf{a}})\, \rho(\tilde{\mathbf{b}}, \mathbf{b})\, \frac{K_1(\tilde{\mathbf{b}}, \mathbf{b})}{\sin\theta_{\mathbf{ac}}}\, T_{\text{r}}(t_{\mathbf{c}}) \tag{1.12}$$

Equations 1.10–1.12 give the resulting expressions of the estimators, which are also illustrated in Figs. 1.1b–d. We list only three out of the 9 estimators reformulated by Křivánek et al. [14]. As shown in the same paper, not all of the estimators have complementary advantages that the combined algorithm could benefit from. For instance, P-B2D and P-B3D both have very similar pdfs and only differ by the amount of bias. For this reason, we choose to only use the minimum-blur volumetric estimators, i.e., P-P3D, P-B2D and B-B1D, as they introduce less bias (though the B-P2D estimator has this property too we leave it out as it cannot be implemented as efficiently as P-B2D). Note that these three estimators are also known under other names, P-P3D as volumetric photon mapping [13], P-B2D as beam radiance estimate (BRE) [9] and B-B1D as photon beams [10].

All three estimators share the same prefix $C_{\text{l}}(\mathbf{x}_0 \dots \mathbf{a}]$ and postfix $C_{\text{c}}[\mathbf{c} \dots \mathbf{x}_k)$ which are purposefully omitted for notational brevity. K_d denotes a normalized d-dimensional kernel. The scattering function ρ at a query location $\mathbf{x}_j$ is evaluated with the direction of the photon beam or photon, which may not pass through this location. To describe this behaviour, definition of ρ is amended as

$$\rho(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} \rho_{\text{s}}(\mathbf{x}_{i-1} \rightarrow \mathbf{x}_i, \mathbf{x}_j \rightarrow \mathbf{x}_{j+1}) & \text{both } \mathbf{x}_i, \mathbf{x}_j \text{ on surface} \\ \rho_{\text{p}}(\mathbf{x}_{i-1} \rightarrow \mathbf{x}_i, \mathbf{x}_j \rightarrow \mathbf{x}_{j+1}) \sigma_{\text{s}}(\mathbf{x}_j) & \text{both } \mathbf{x}_i, \mathbf{x}_j \text{ in medium} \\ 0 & \text{otherwise.} \end{cases}$$

If one vertex is on a surface while the other in a medium the scattering function ρ is zero (it does not make sense to merge them).

The estimator abbreviations provide information about the radiance data (photon **P**oints or photon **B**eams), query type (**P**oint or **B**eam) and the kernel dimension (1D, 2D or 3D), in exactly this order (e.g., P-B2D refers to point data $\times$ beam query, 2D kernel).

1.2.1 "Long" and "short" beams.

Jarosz et al. [10] derived photon beams assuming a beam extends until the closest surface, with the transmittance along the beam appearing as a part of the estimator. We follow Křivánek et al. [14] and refer to this as "long" beams (Fig. 1.2a). Jarosz et al. [11] also proposed an unbiased approximation of transmittance by several step functions, and coined this approach "progressive deep shadow maps". Approximating transmittance by a single step function yields beams of finite extent where the transmittance vanishes (it is replaced by a constant 1). Křivánek refers to this as "short" beams (Fig. 1.2b). The same idea can be applied to query beams. To distinguish between these options, we will use $\mathrm{B_s}$ and $\mathrm{B_l}$ to denote short and long beams, respectively, either of which can be used in place of any B in the estimators P-B2D and B-B1D. The original names P-B2D and B-B1D without beam type specification will denote generally all estimators P-$\mathrm{B_l}$2D, P-$\mathrm{B_s}$2D and $\mathrm{B_l}$-$\mathrm{B_l}$1D, $\mathrm{B_l}$-$\mathrm{B_s}$1D, $\mathrm{B_s}$-$\mathrm{B_l}$1D, $\mathrm{B_s}$-$\mathrm{B_s}$1D.

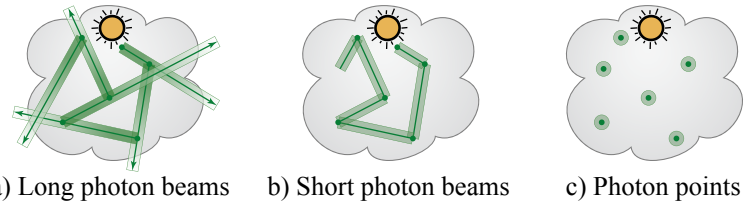


Figure 1.2: "Long" and "short" photon beams, and photon points.

Equations 1.11–1.12 hold for long photon and query beams. To derive the impact of short beams, Křivánek et al. [14] used a new interpretation of short beams as a Russian roulette (RR) decision on the outcome of the long-beam estimator.

Consider a pdf $p(t) = \sigma_\mathrm{t}(t)T_\mathrm{r}(t)$ used to sample the length of a short beam. The probability that the beam contributes at some distance t_0 from its origin is the probability that the beam length l is at least t_0:

$$\Pr\{l > t_0\} = \int_{t_0}^{\infty} p(t')\,\mathrm{d}t' = T_\mathrm{r}(t_0)\ . \tag{1.13}$$

Any long beam estimator can be converted to a corresponding short beam estimator by making a zero contribution whenever the beam is too short to make the considered contribution, else by the original estimator divided by the RR probability (1.13). The short beam variants of the estimators Equations 1.11–1.12

then read:

$$\langle I\rangle_{\text{P-B}_\text{s}\text{2D}} = \langle I\rangle_{\text{P-B}_\text{l}\text{2D}} \frac{H(l_\mathbf{c} - t_\mathbf{c})}{\Pr\{l_\mathbf{c} > t_\mathbf{c}\}} \tag{1.14}$$

$$\langle I\rangle_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}} = \langle I\rangle_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}} \frac{H(l_\mathbf{c} - t_\mathbf{c})}{\Pr\{l_\mathbf{c} > t_\mathbf{c}\}} \tag{1.15}$$

$$\langle I\rangle_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}} = \langle I\rangle_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}} \frac{H(l_\mathbf{a} - t_\mathbf{a})}{\Pr\{l_\mathbf{a} > t_\mathbf{a}\}} \tag{1.16}$$

$$\langle I\rangle_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}} = \langle I\rangle_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}} \frac{H(l_\mathbf{a} - t_\mathbf{a})}{\Pr\{l_\mathbf{a} > t_\mathbf{a}\}} \frac{H(l_\mathbf{c} - t_\mathbf{c})}{\Pr\{l_\mathbf{c} > t_\mathbf{c}\}} \tag{1.17}$$

,

where $l_\mathbf{a}$ is the length of the photon beam, $l_\mathbf{c}$ is the length of the query beam, and the Heaviside step function H indicates whether or not the beam is long enough to make a contribution at $t_\mathbf{a}$ and $t_\mathbf{c}$, respectively. We purposefully do not expand these equations and let transmittances cancel out since that would not be correct if transmittance was a vector (as in our implementation).

2. Combining estimators

In the previous chapter we introduced two estimator families, now we would like to combine them. Why? As Křivánek et al. proved by a variance analysis of the volumetric photon density estimators (see Section 5 in [14]), no single estimator is superior to all others in all circumstances. Points can be better than beams and vice versa, depending on the relative size of the blur kernel to the mean free path (MFP, a mean distance between two subsequent scattering events in a medium). Dense media are better handled by points and thin media by beams. This suggests it would be beneficial to combine the volumetric photon density estimators. And such combination would be even more robust and also capable of handling light transport outside participating media, if BPT could be plugged in.

Multiple importance sampling (MIS) [28] is one approach for combining estimators. Consider an integral $I = \int_{\mathcal{D}_x} f(x)\,\mathrm{d}x$ and its n estimators

$$\langle I \rangle_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \frac{f(X_{i,j})}{p_i(X_{i,j})}, i = 1, \ldots, n. \tag{2.1}$$

The i-th estimator is constructed by taking n_i random variables $X_{i,j}$, $j = 1, \ldots, n_i$ from a sampling technique with the pdf $p_i(x)$. Multiple importance sampling combines (weighted instances of) these estimators:

$$\langle I \rangle_{\mathrm{MIS}} = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})}, \tag{2.2}$$

where $w_i(x)$ are heuristic weighting functions that must sum up to one for any x. A provably good choice for $w_i(x)$, in terms of minimizing the variance of $\langle I \rangle_{\mathrm{MIS}}$, is the *balance heuristic*:

$$\hat{w}_i(x) = \frac{n_i p_i(x)}{\sum_{k=1}^{n} n_k p_k(x)}. \tag{2.3}$$

In this original formulation MIS is designed to combine unbiased estimators of the same integral. However, this is not the case in our setup. We would like to combine unbiased path integration techniques from BPT with the volumetric photon density estimators. These estimators, in fact, converge to different results, and, in addition, operate on space of different dimension (the volumetric photon density estimators have the additional integration that corresponds to blurring by the kernel; no such thing exists in BPT).

Similar incompatibility has already been encountered when trying to combine BPT with surface photon density estimators and previous work proposed number of solutions to resolve it. Georgiev et al. [4] derived the Vertex Connection and Merging framework (VCM), where the extra integration that corresponds to blurring is interpreted as a Russian roulette decision, with an immediate impact on the path pdf. The UPS framework by Hachisuka et al. [7] defined an extended path space by introducing a "virtual perturbation" of one of the path vertices in BPT. While it is not impossible to extend VCM or UPS to handle media, Křivánek et al. [14] has chosen a more rigorous approach. He derived an extended

version of MIS that is able to combine estimators of integral over spaces of different dimensions. While this extended MIS can be readily used to combine all of the aforementioned estimators, the theory is somewhat unweildy. For this reason, we describe a much more direct approach for deriving the MIS weights for combining the estimators. We derive a "pdf" for each of the estimators, which can then be directly plugged into the balance heuristic (2.3).

2.1 Intuitive pdf derivation

To easily derive path pdfs for the estimators, we use a somewhat artificial, yet reasonable, separation of the terms that appear in the (known) estimator formulas into the path contribution function and the "pdf" itself. Because we know that estimators are generally constructed as $\langle I \rangle = f(\overline{\mathbf{x}})/p(\overline{\mathbf{x}})$, we can write the pdf $p(\overline{\mathbf{x}})$ with which each considered technique creates path $\overline{\mathbf{x}}$ as:

$$p(\overline{\mathbf{x}}) = \frac{f(\overline{\mathbf{x}})}{\langle I \rangle}. \tag{2.4}$$

We arbitrarily choose the path contribution function to be:

$$f(\overline{\mathbf{x}}) = L_\mathrm{e}(\mathbf{x}_0) T(\mathbf{x}_0 \dots \tilde{\mathbf{b}}) \rho(\tilde{\mathbf{b}}, \mathbf{b}) T(\mathbf{b} \dots \mathbf{x}_k) W_\mathrm{e}(\mathbf{x}_k), \tag{2.5}$$

and dividing it by the known estimator formulas we get the pdf for each of the estimators.

2.1.1 Point-Point 3D

We start with the P-P3D estimator. The contribution of a photon at $\tilde{\mathbf{b}}$ to a pixel value through a P-P3D estimate performed at vertex $\mathbf{b}$ is (after completing and expanding Equation 1.10):

$$\begin{aligned} \langle I \rangle_\text{P-P3D} \;=\; & L_\mathrm{e}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{a})}{p(\mathbf{x}_0 \dots \mathbf{a})} \frac{\rho(\mathbf{a})}{p(\omega_\mathbf{a})} \frac{G(\mathbf{a}, \tilde{\mathbf{b}})}{G(\mathbf{a}, \tilde{\mathbf{b}})} \\ & \frac{T_\mathrm{r}(t_\mathbf{a})}{p(t_\mathbf{a})} \rho(\tilde{\mathbf{b}}, \mathbf{b})\, K_3(\tilde{\mathbf{b}}, \mathbf{b}) \frac{T_\mathrm{r}(t_\mathbf{c})}{p(t_\mathbf{c})} \\ & \frac{G(\mathbf{b}, \mathbf{c})}{G(\mathbf{b}, \mathbf{c})} \frac{\rho(\mathbf{c})}{p(\omega_\mathbf{c})} \frac{T(\mathbf{c} \dots \mathbf{x}_k)}{p(\mathbf{c} \dots \mathbf{x}_k)} W_\mathrm{e}(\mathbf{x}_k). \end{aligned} \tag{2.6}$$

Notice that $\rho(\mathbf{a}) G(\mathbf{a}, \tilde{\mathbf{b}}) T_\mathrm{r}(t_\mathbf{a})$ is simply the throughput between $\mathbf{a}$ and $\tilde{\mathbf{b}}$ including scattering at $\mathbf{a}$. Multiplying $T(\mathbf{x}_0 \dots \mathbf{a})$ by this factor therefore yields $T(\mathbf{x}_0 \dots \tilde{\mathbf{b}})$. Similarly $T_\mathrm{r}(t_\mathbf{c}) G(\mathbf{b}, \mathbf{c}) \rho(\mathbf{c}) T(\mathbf{c} \dots \mathbf{x}_k) = T(\mathbf{b} \dots \mathbf{x}_k)$. Furthermore, $p(\omega_\mathbf{a}) p(t_\mathbf{a})$ is the probability density of sampling $\tilde{\mathbf{b}}$ from $\mathbf{a}$ in the projected solid angle $\times$ length product measure (first we sample the direction $\omega_\mathbf{a}$ of the photon beam, then a length on it). Multiplication by the geometry term $G(\mathbf{a}, \tilde{\mathbf{b}})$ then converts it to the volume measure, that is $p(\tilde{\mathbf{b}} \,|\, \mathbf{a})$, and clearly $p(\tilde{\mathbf{b}} \,|\, \mathbf{a}) p(\mathbf{x}_0 \dots \mathbf{a}) = p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})$. Analogously for the camera subpath we get $p(t_\mathbf{c}) G(\mathbf{b}, \mathbf{c}) p(\omega_\mathbf{c}) p(\mathbf{c} \dots \mathbf{x}_k) = p(\mathbf{b} \,|\, \mathbf{c}) p(\mathbf{c} \dots \mathbf{x}_k) = p(\mathbf{b} \dots \mathbf{x}_k)$. We apply these equalities on Equation 2.6 and get a more compact

formula:

$$\begin{aligned}\langle I\rangle_{\text{P-P3D}} &= L_{\text{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\dots\tilde{\mathbf{b}})}{p(\mathbf{x}_0\dots\tilde{\mathbf{b}})}\\ &\quad \rho(\tilde{\mathbf{b}},\mathbf{b})\,K_3(\tilde{\mathbf{b}},\mathbf{b})\\ &\quad \frac{T(\mathbf{b}\dots\mathbf{x}_k)}{p(\mathbf{b}\dots\mathbf{x}_k)}W_{\text{e}}(\mathbf{x}_k).\end{aligned} \tag{2.7}$$

Dividing the contribution function (2.5) by the expression above yields the "reverse-engineered" path pdf:

$$\boxed{p_{\text{P-P3D}}(\overline{\mathbf{x}}) = p(\mathbf{x}_0\dots\tilde{\mathbf{b}})\frac{1}{K_3(\tilde{\mathbf{b}},\mathbf{b})}p(\mathbf{b}\dots\mathbf{x}_k).} \tag{2.8}$$

2.1.2 Point-Beam 2D

Following similar steps as in the P-P3D case we rewrite the contribution of the P-B$_{\text{l}}$2D estimator as:

$$\begin{aligned}\langle I\rangle_{\text{P-B}_{\text{l}}\text{2D}} &= L_{\text{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\dots\tilde{\mathbf{b}})}{p(\mathbf{x}_0\dots\tilde{\mathbf{b}})}\\ &\quad \rho(\tilde{\mathbf{b}},\mathbf{b})\,K_2(\tilde{\mathbf{b}},\mathbf{b})\\ &\quad \frac{T(\mathbf{b}\dots\mathbf{x}_k)}{G(\mathbf{b},\mathbf{c})p(\omega_{\mathbf{c}})p(\mathbf{c}\dots\mathbf{x}_k)}W_{\text{e}}(\mathbf{x}_k).\end{aligned} \tag{2.9}$$

Again, by dividing the contribution function (2.5) by the estimator we obtain the path pdf:

$$\boxed{p_{\text{P-B}_{\text{l}}\text{2D}}(\overline{\mathbf{x}}) = p(\mathbf{x}_0\dots\tilde{\mathbf{b}})\frac{G(\mathbf{b},\mathbf{c})p(\omega_{\mathbf{c}})}{K_2(\tilde{\mathbf{b}},\mathbf{b})}p(\mathbf{c}\dots\mathbf{x}_k).} \tag{2.10}$$

When considering a short query beam, the estimator is according to Equation 1.14 divided by probability $\Pr\{l_{\mathbf{c}} > t_{\mathbf{c}}\}$ that the query beam of length $l_{\mathbf{c}}$ was sampled long enough to make a contribution at $t_{\mathbf{c}}$. So the resulting path pdf is:

$$\boxed{p_{\text{P-B}_{\text{s}}\text{2D}}(\overline{\mathbf{x}}) = \Pr\{l_{\mathbf{c}} > t_{\mathbf{c}}\}\,p_{\text{P-B}_{\text{l}}\text{2D}}(\overline{\mathbf{x}}).} \tag{2.11}$$

2.1.3 Beam-Beam 1D

Finally, we rewrite the contribution of the B$_{\text{l}}$-B$_{\text{l}}$1D estimator:

$$\begin{aligned}\langle I\rangle_{\text{B}_{\text{l}}\text{-B}_{\text{l}}\text{1D}} &= L_{\text{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\dots\tilde{\mathbf{b}})}{p(\mathbf{x}_0\dots\mathbf{a})p(\omega_{\mathbf{a}})G(\mathbf{a},\tilde{\mathbf{b}})}\\ &\quad \rho(\tilde{\mathbf{b}},\mathbf{b})\,\frac{K_1(\tilde{\mathbf{b}},\mathbf{b})}{\sin\theta_{\mathbf{ac}}}\\ &\quad \frac{T(\mathbf{b}\dots\mathbf{x}_k)}{G(\mathbf{b},\mathbf{c})p(\omega_{\mathbf{c}})p(\mathbf{c}\dots\mathbf{x}_k)}W_{\text{e}}(\mathbf{x}_k).\end{aligned} \tag{2.12}$$

As before, we divide the path contribution function (2.5) by the estimator to obtain the path pdf expression:

$$\boxed{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) = p(\mathbf{x}_0 \dots \mathbf{a}) \frac{p(\omega_\mathbf{a}) G(\mathbf{a}, \tilde{\mathbf{b}}) \sin\theta_\mathbf{ac} G(\mathbf{b}, \mathbf{c}) p(\omega_\mathbf{c})}{K_1(\tilde{\mathbf{b}}, \mathbf{b})} p(\mathbf{c} \dots \mathbf{x}_k).} \tag{2.13}$$

Getting pdfs for the short beam variants is straightforward ($l_\mathbf{a}$ is the length of the photon beam, $l_\mathbf{c}$ is the length of the query beam):

$$\boxed{\begin{aligned} p_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(\overline{\mathbf{x}}) &= \Pr\{l_\mathbf{c} > t_\mathbf{c}\}\, p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) && (2.14)\\ p_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) &= \Pr\{l_\mathbf{a} > t_\mathbf{a}\}\, p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) && (2.15)\\ p_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(\overline{\mathbf{x}}) &= \Pr\{l_\mathbf{a} > t_\mathbf{a}\} \Pr\{l_\mathbf{c} > t_\mathbf{c}\}\, p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}). && (2.16) \end{aligned}}$$

2.1.4 Comparison to Křivánek's work

We will now compare the pdf formulas we derived above with those presented by Křivánek et al. [14]. Let's assume using a constant kernel in the form $K_i(\mathbf{x}, \mathbf{y}) = |S_i(\mathbf{x})|^{-1}$, where $S_i(\mathbf{x})$ is the support of $K_i(\mathbf{x}, \mathbf{y})$ for a given $\mathbf{x}$. Then we have

$$p_{\text{P-P3D}}(\overline{\mathbf{x}}) = |S_3(\tilde{\mathbf{b}})| p(\mathbf{x}_0 \dots \tilde{\mathbf{b}}) p(\mathbf{b} \dots \mathbf{x}_k) \tag{2.17}$$

$$p_{\text{P-B}_\text{l}\text{2D}}(\overline{\mathbf{x}}) = |S_2(\tilde{\mathbf{b}})| p(\mathbf{x}_0 \dots \tilde{\mathbf{b}}) G(\mathbf{b}, \mathbf{c}) p(\omega_\mathbf{c}) p(\mathbf{c} \dots \mathbf{x}_k) \tag{2.18}$$

$$p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) = |S_1(\tilde{\mathbf{b}})| p(\mathbf{x}_0 \dots \mathbf{a}) p(\omega_\mathbf{a}) G(\mathbf{a}, \tilde{\mathbf{b}}) \sin\theta_\mathbf{ac} G(\mathbf{b}, \mathbf{c}) p(\omega_\mathbf{c}) p(\mathbf{c} \dots \mathbf{x}_k). \tag{2.19}$$

These expressions are exactly the same as those Křivánek et al. derived under the constant kernel assumption for the MIS weights computation (see Section 7 in [14]).

2.1.5 Bidirectional path tracing

As already mentioned, our goal is to combine the volumetric photon density estimators with unbiased path integration techniques from BPT. Assuming the same input configuration as shown in the Fig. 1.1a, there are two such techniques that produce path of the same length as the volumetric photon density estimators do. Either we can sample the light subpath up to the vertex $\tilde{\mathbf{b}}$ and then explicitly connect it to the camera subpath endpoint $\mathbf{c}$, or we can generate the camera subpath till the vertex $\mathbf{b}$ and connect it to the last vertex on the light subpath $\mathbf{a}$. We label the first one as $\text{BPT}_{\tilde{\mathbf{b}}\mathbf{c}}$ and the second one as $\text{BPT}_{\mathbf{a}\mathbf{b}}$ (the subscript denotes the explicitly connected subpath endpoints). Deriving their formulas is easy, their are products of subpath contributions and the throughput along the connecting edge:

$$\langle I \rangle_{\text{BPT}_{\tilde{\mathbf{b}}\mathbf{c}}} = L_\text{e}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \tilde{\mathbf{b}})}{p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})} \rho(\tilde{\mathbf{b}}) T_\text{r}(\tilde{\mathbf{b}}, \mathbf{c}) G(\tilde{\mathbf{b}}, \mathbf{c}) \rho(\mathbf{c}) \frac{T(\mathbf{c} \dots \mathbf{x}_k)}{p(\mathbf{c} \dots \mathbf{x}_k)} W_\text{e}(\mathbf{x}_k) \tag{2.20}$$

$$\langle I \rangle_{\text{BPT}_{\mathbf{a}\mathbf{b}}} = L_\text{e}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{a})}{p(\mathbf{x}_0 \dots \mathbf{a})} \rho(\mathbf{a}) T_\text{r}(\mathbf{a}, \mathbf{b}) G(\mathbf{a}, \mathbf{b}) \rho(\mathbf{b}) \frac{T(\mathbf{b} \dots \mathbf{x}_k)}{p(\mathbf{b} \dots \mathbf{x}_k)} W_\text{e}(\mathbf{x}_k). \tag{2.21}$$

Product of throughput of subpaths $\mathbf{x}_0 \dots \tilde{\mathbf{b}}$, $\mathbf{c} \dots \mathbf{x}_k$, and the connecting edge $\tilde{\mathbf{b}}\mathbf{c}$ is the throughput of the whole path $\mathbf{x}_0 \dots \mathbf{a}\tilde{\mathbf{b}}\mathbf{c} \dots \mathbf{x}_k$. That means:

$$T(\mathbf{x}_0 \dots \tilde{\mathbf{b}})T(\mathbf{c} \dots \mathbf{x}_k)\rho(\tilde{\mathbf{b}})T_{\mathrm{r}}(\tilde{\mathbf{b}}, \mathbf{c})G(\tilde{\mathbf{b}}, \mathbf{c})\rho(\mathbf{c}) = T(x_0 \dots \mathbf{a}\tilde{\mathbf{b}}\mathbf{c} \dots x_k) \tag{2.22}$$

and Equation 2.20 becomes

$$\langle I \rangle_{\mathrm{BPT}_{\tilde{\mathbf{b}}\mathbf{c}}} = L_{\mathrm{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \dots \mathbf{a}\tilde{\mathbf{b}}\mathbf{c} \dots \mathbf{x}_k)}{p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})p(\mathbf{c} \dots \mathbf{x}_k)}W_{\mathrm{e}}(\mathbf{x}_k). \tag{2.23}$$

Similarly, we simplify the second formula (2.21) and get:

$$\langle I \rangle_{\mathrm{BPT}_{\mathbf{ab}}} = L_{\mathrm{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \dots \mathbf{abc} \dots \mathbf{x}_k)}{p(\mathbf{x}_0 \dots \mathbf{a})p(\mathbf{b} \dots \mathbf{x}_k)}W_{\mathrm{e}}(\mathbf{x}_k). \tag{2.24}$$

Since there are not both vertices $\tilde{\mathbf{b}}, \mathbf{b}$ on the path as with the volumetric photon density estimators but only one depending on the connecting edge, the contribution function (2.5) turns into:

$$f_{\mathrm{BPT}_{\tilde{\mathbf{b}}\mathbf{c}}}(\overline{\mathbf{x}}) = L_{\mathrm{e}}(\mathbf{x}_0)T(\mathbf{x}_0 \dots \mathbf{a}\tilde{\mathbf{b}}\mathbf{c} \dots \mathbf{x}_k)W_{\mathrm{e}}(\mathbf{x}_k) \tag{2.25}$$

$$f_{\mathrm{BPT}_{\mathbf{ab}}}(\overline{\mathbf{x}}) = L_{\mathrm{e}}(\mathbf{x}_0)T(\mathbf{x}_0 \dots \mathbf{abc} \dots \mathbf{x}_k)W_{\mathrm{e}}(\mathbf{x}_k). \tag{2.26}$$

Following the same procedure as before, we divide these formulas by the expressions 2.23 and 2.24, respectively, which yields the (expected) path pdfs:

$$\boxed{p_{\mathrm{BPT}_{\tilde{\mathbf{b}}\mathbf{c}}}(\overline{\mathbf{x}}) = p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})p(\mathbf{c} \dots \mathbf{x}_k)} \tag{2.27}$$

$$\boxed{p_{\mathrm{BPT}_{\mathbf{ab}}}(\overline{\mathbf{x}}) = p(\mathbf{x}_0 \dots \mathbf{a})p(\mathbf{b} \dots \mathbf{x}_k).} \tag{2.28}$$

Note that the estimator and pdf equations for BPT hold not only in media but also on surfaces.

Apart from the input configuration shown in the Fig. 1.1a there are two more situations when this estimator can be applied: if the whole light transport path is sampled from a light and if it is entirely sampled from the camera. These situations are possible only if the camera and the light, respectively, can be hit by a ray and can give out importance and radiance, respectively, in its direction (i.e. it is not a point or direction light or a pinhole camera). Since there is no explicit connection of subpath ends, all formulas are simple and the same for both cases - the estimator contribution:

$$\langle I \rangle_{\mathrm{BPT}_{\mathrm{direct}}} = L_{\mathrm{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \dots \mathbf{x}_k)}{p(\mathbf{x}_0 \dots \mathbf{x}_k)}W_{\mathrm{e}}(\mathbf{x}_k), \tag{2.29}$$

the contribution function:

$$f_{\mathrm{BPT}_{\mathrm{direct}}}(\overline{\mathbf{x}}) = L_{\mathrm{e}}(\mathbf{x}_0)T(\mathbf{x}_0 \dots \mathbf{x}_k)W_{\mathrm{e}}(\mathbf{x}_k), \tag{2.30}$$

and the resulting path pdf:

$$\boxed{p_{\mathrm{BPT}_{\mathrm{direct}}}(\overline{\mathbf{x}}) = p(\mathbf{x}_0 \dots \mathbf{x}_k).} \tag{2.31}$$

2.1.6 Surface photon density estimator

There is one more estimator included in the UPBP implementation, the surface photon density estimator (denoted P-P2D in [14], we use SURF). Assume the same input configuration as shown in the Fig. 1.1a and that the rays $(\mathbf{a}, \omega_\mathbf{a}), (\mathbf{c}, \omega_\mathbf{c})$ got through any media (if present) and hit surface at the point $\tilde{\mathbf{b}}$ and $\mathbf{b}$, respectively. The estimator formula has the following form

$$\begin{aligned}\langle I \rangle_{\text{SURF}} \;=\; & L_\text{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \dots \mathbf{a})}{p(\mathbf{x}_0 \dots \mathbf{a})}\frac{\rho(\mathbf{a})}{p(\omega_\mathbf{a})}\frac{G(\mathbf{a}, \tilde{\mathbf{b}})}{G(\mathbf{a}, \tilde{\mathbf{b}})} \\ & \frac{T_\text{r}(t_\mathbf{a})}{p(t_\mathbf{a})}\rho(\tilde{\mathbf{b}}, \mathbf{b})\, K_2(\tilde{\mathbf{b}}, \mathbf{b})\frac{T_\text{r}(t_\mathbf{c})}{p(t_\mathbf{c})} \\ & \frac{G(\mathbf{b}, \mathbf{c})}{G(\mathbf{b}, \mathbf{c})}\frac{\rho(\mathbf{c})}{p(\omega_\mathbf{c})}\frac{T(\mathbf{c} \dots \mathbf{x}_k)}{p(\mathbf{c} \dots \mathbf{x}_k)}W_\text{e}(\mathbf{x}_k),\end{aligned} \tag{2.32}$$

where $t_\mathbf{a}, t_\mathbf{c}$ denotes lengths of the rays. Actually, it is almost identical to the P-P3D estimator (2.6) with two differences. Only a two-dimensional kernel is used (since the vertices are located on a surface) and the terms $p(t_\mathbf{a}), p(t_\mathbf{c})$ have a different meaning. While they expressed the probability density of sampling distances $t_\mathbf{a}, t_\mathbf{c}$ along the rays in the P-P3D case, here they express the probability *mass* (as opposed to the probability density) that the rays got through media and reached a surface at these distances (we explain this issue in Section 3.2.5). We can follow exactly same steps as in the P-P3D case and arrive at a very similar path pdf expression:

$$\boxed{p_\text{SURF}(\overline{\mathbf{x}}) = p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})\frac{1}{K_2(\tilde{\mathbf{b}}, \mathbf{b})}p(\mathbf{b} \dots \mathbf{x}_k).} \tag{2.33}$$

2.1.7 List of pdfs

We conclude this subsection with a list of all the derived path pdfs:

$$\begin{aligned}
p_\text{P-P3D}(\overline{\mathbf{x}}) &= p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})\frac{1}{K_3(\tilde{\mathbf{b}}, \mathbf{b})}p(\mathbf{b} \dots \mathbf{x}_k) \\
p_{\text{P-B}_\text{l}\text{2D}}(\overline{\mathbf{x}}) &= p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})\frac{G(\mathbf{b}, \mathbf{c})p(\omega_\mathbf{c})}{K_2(\tilde{\mathbf{b}}, \mathbf{b})}p(\mathbf{c} \dots \mathbf{x}_k) \\
p_{\text{P-B}_\text{s}\text{2D}}(\overline{\mathbf{x}}) &= \Pr\{l_\mathbf{c} > t_\mathbf{c}\}\, p_{\text{P-B}_\text{l}\text{2D}}(\overline{\mathbf{x}}) \\
p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) &= p(\mathbf{x}_0 \dots \mathbf{a})\frac{p(\omega_\mathbf{a})G(\mathbf{a}, \tilde{\mathbf{b}})\sin\theta_\mathbf{ac}G(\mathbf{b}, \mathbf{c})p(\omega_\mathbf{c})}{K_1(\tilde{\mathbf{b}}, \mathbf{b})}p(\mathbf{c} \dots \mathbf{x}_k) \\
p_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(\overline{\mathbf{x}}) &= \Pr\{l_\mathbf{c} > t_\mathbf{c}\}\, p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) \\
p_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) &= \Pr\{l_\mathbf{a} > t_\mathbf{a}\}\, p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) \\
p_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(\overline{\mathbf{x}}) &= \Pr\{l_\mathbf{a} > t_\mathbf{a}\}\Pr\{l_\mathbf{c} > t_\mathbf{c}\}\, p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(\overline{\mathbf{x}}) \\
p_{\text{BPT}_{\tilde{\mathbf{b}}\mathbf{c}}}(\overline{\mathbf{x}}) &= p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})p(\mathbf{c} \dots \mathbf{x}_k) \\
p_{\text{BPT}_\mathbf{ab}}(\overline{\mathbf{x}}) &= p(\mathbf{x}_0 \dots \mathbf{a})p(\mathbf{b} \dots \mathbf{x}_k) \\
p_{\text{BPT}_\text{direct}}(\overline{\mathbf{x}}) &= p(\mathbf{x}_0 \dots \mathbf{x}_k) \\
p_\text{SURF}(\overline{\mathbf{x}}) &= p(\mathbf{x}_0 \dots \tilde{\mathbf{b}})\frac{1}{K_2(\tilde{\mathbf{b}}, \mathbf{b})}p(\mathbf{b} \dots \mathbf{x}_k).
\end{aligned}$$

2.2 Estimators along a path

All the estimators and their equations stated above were derived for the vertex $\mathbf{x}_s$ on a length-k path $\overline{\mathbf{x}} = \mathbf{x}_0 \dots \mathbf{x}_k \in \Omega$ with $k \geq 1$ segments and $k+1$ vertices, with its first vertex $\mathbf{x}_0$ on a light source, its last vertex $\mathbf{x}_k$ on the camera sensor, and the $\mathbf{x}_1 \dots \mathbf{x}_{k-1}$ scattering vertices on surfaces and/or in media. We would like to point out that this is a general derivation and each of the estimators can be actually applied on different vertices along the path, i.e. on the vertex $\mathbf{x}_s$ for different s, yielding a whole family of estimators.

Let there be m out of the $k-1$ scattering vertices located in media and the remaining $n = k-1-m$ vertices on (not purely specular) surfaces. Then there are $7m$ volumetric photon density estimators (P-P3D, P-B_l2D, P-B_s2D, B_l-B_l1D, B_l-B_s1D, B_s-B_l1D, B_s-B_s1D for each of the m vertices in media), n surface photon density estimators (SURF for each of the n vertices on surfaces) and $k+2$ path integral estimators (BPT for each connecting edge, i.e. for each of the k segments, $+1$ for a path sampled completely from a light, $+1$ for a path sampled completely from the camera). If any of the n vertices is located on a purely specular surface (its BRDF is a delta function) then the surface photon density estimator for such vertex as well as both path integral estimators for incident edges do not exist (there is zero probability of having sampled exactly the one and only acceptable direction). Figure 2.1 shows an example how the volumetric photon density estimators can be applied on different vertices along a path, Figure 2.2 illustrates all estimators along a fixed path.

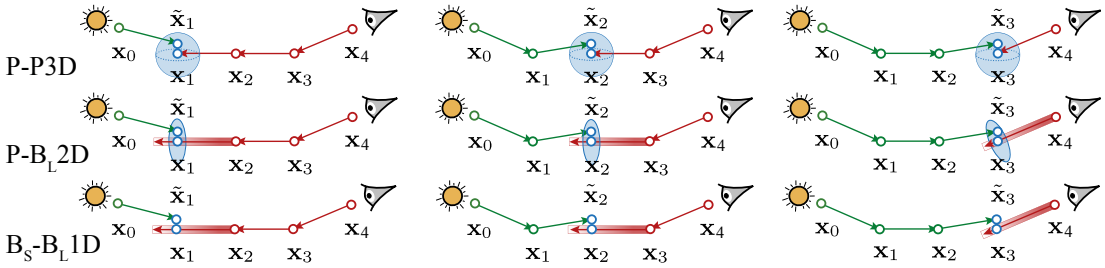


Figure 2.1: An example how the volumetric photon density estimators (shown P-P2D, P-B_l2D, B_s-B_l1D) can be applied on different vertices along a path (the path is located entirely in a medium).

2.3 Summary

At the beginning of this chapter, we explained why combination of volumetric photon density estimators and BPT seems promising. Then we show how combination of estimators using MIS works and that it depends on path pdfs. The rest of the chapter described derivation of path pdfs for all necessary estimators. We finished here the theoretical part of the thesis, description of the implementation follows.

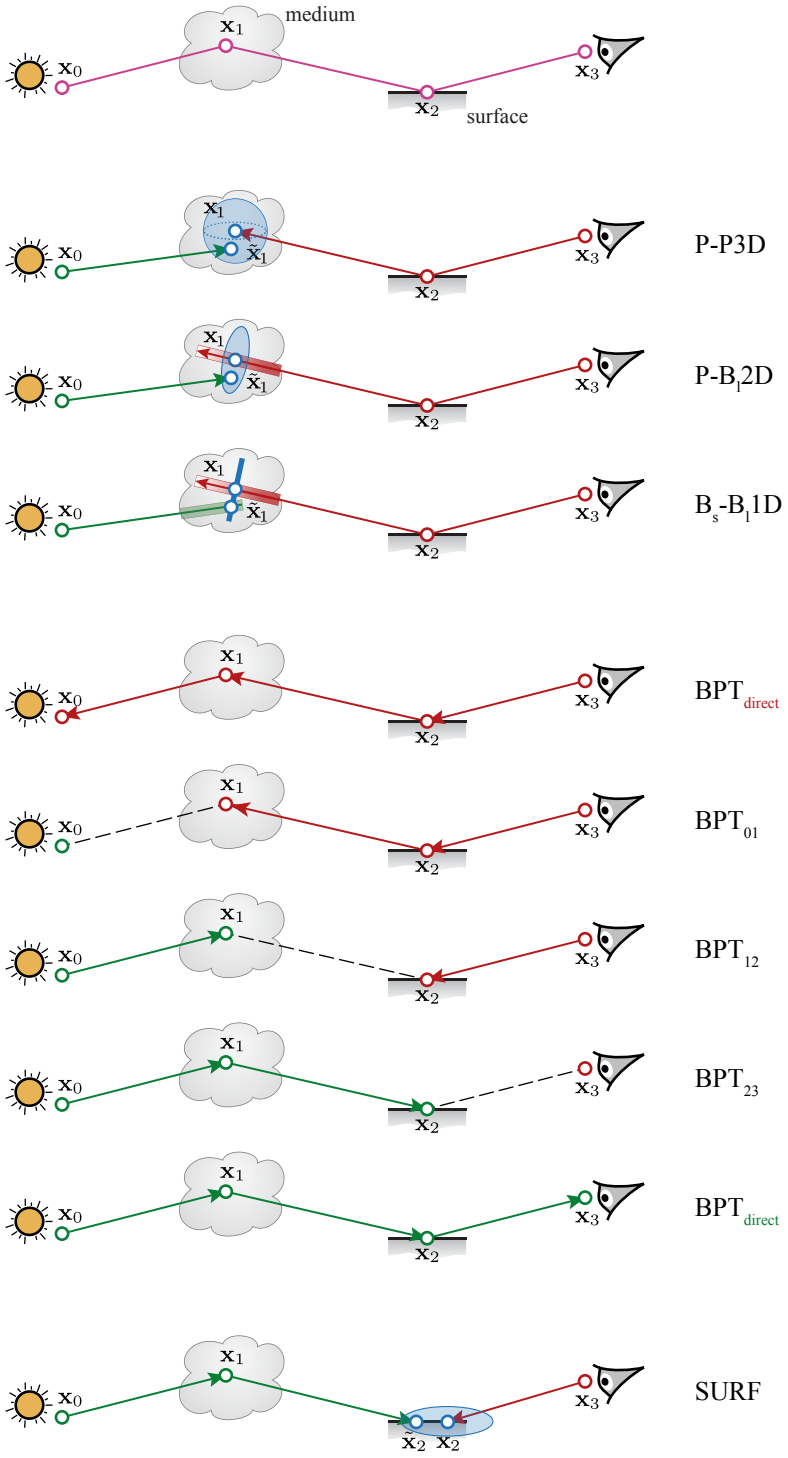


Figure 2.2: An example of *all* estimators along a fixed path. The path has 4 vertices (i.e. its length is 3), one of the two inner vertices is located in a medium, the other on a surface. Therefore, there are 7 volumetric density estimators, 5 path integral estimators and one surface photon density estimator. Only 3 volumetric photon density estimators are actually shown as the others differ only in a beam type and their image would be very similar.

3. Our work

Our work was to transform the theory described in the previous chapters into practice, i.e. to implement the UPBP algorithm. We decided not to start from scratch but to build on SmallVCM [25], an open-source implementation of the VCM [4] algorithm. It was a reasonable choice since UPBP "includes" VCM (VCM simulates a subset of light transport UPBP can handle and they are the same in scenes without media). However, SmallVCM had no support for media so it was still a challenging task. For example we had to add a representation of media and a phase function, completely change the ray-scene intersection routine to be able to tell in what medium a traced ray is located, add data structures for photon beams, and, most importantly, implement a new computation of MIS weights. The resulting program called SmallUBPB was used to generate images in the UPBP paper [14] (e.g. those in Figure 1) and its source code can be downloaded from [24]. This chapter describes its main features.

3.1 Code introduction

3.1.1 Structure

We begin with a brief overview of the code. It is written in C++ and divided into two solutions: `OpenEXR` and `SmallUPBP`. The former one contains OpenEXR library [19] we use to read and save images in the OpenEXR format. The latter one is further divided into three projects: `embree`, `sys` and `SmallUPBP`. The first two contain Embree [3] which we employed for acceleration of ray tracing and photon lookup. Along with OpenEXR they are almost unmodified third-party code. We rather focus on the last one as it contains the UPBP algorithm itself. Its (mostly header) files are separated into these folders:

Beams Code in this folder relates to the B-B1D estimator. It implements its evaluation (`PhBeams.hxx`, `PhBeams.cxx`, `PhotonBeam.hxx`) and data structures needed for beam lookup (`Grid.hxx`, `PhGrid.hxx`). There are also files adding some debugging features (`BeamDensity.hxx`, `GridStats.hxx`) and alternative data structures (`PhBrute.hxx`, `PhEmbree.hxx`).

Bre Similarly to the B-B1D estimator, P-B2D has also its own folder. It contains code for evaluation of this estimator (`Bre.hxx`, `Bre.cxx`) and incorporation of Embree for photon lookup along a beam (`EmbreeAcc.hxx`).

Misc Files that do not fit in other folders are located here. They serve for debugging (`DebugImages.hxx`, `Timer.hxx`), configuration of a scene and its rendering (`Config.hxx`), scene import (`ObjReader.hxx`, `ObjReader.cxx`), random number generation (`Rng.hxx`) or SSE support (`Sse.hxx`). Also basic definitions (`Defs.hxx`), auxiliary (mathematical) functions (`Utils.hxx`, `Utils2.hxx`) and a few data structures (`Framebuffer.hxx`, `HashGrid.hxx`, `KdTmpl.hxx`) can be found in this folder.

Path All code related to a light transport path was put here. It comprises description of a ray, its segments and intersections (`Ray.hxx`), data structures

for keeping record of intersected media boundaries (`BoundaryStack.hxx`, `PriorityStack.hxx`, `StaticArray.hxx`), evaluation of scattering function (`PhaseFunction.hxx`, `Bsdf.hxx`) and definition of a coordinate system (`Frame.hxx`) or path vertices (`UPBPLightVertex.hxx`, `VltPathVertex.hxx`). The very important MIS weights computation functions (`PathWeight.hxx`) are also in this folder.

PrecompiledLibs Contains mainly third-party code for SSE acceleration.

Renderers All renderers (i.e. classes implementing different light transport algorithms) can be found here. There is one abstract renderer (`Renderer.hxx`) and seven derived, three of them with no media support (`EyeLight.hxx`, `PathTracer.hxx`, `VertexCM.hxx`) and four volumetric (`VolBidirPT.hxx`, `VolLightTracer.hxx`, `VolPathTracer.hxx`, `UPBP.hxx`).

Scene Code for description of a scene is placed in this folder. That includes a camera (`Camera.hxx`), lights (`Distribution.hxx`, `EnvMap.hxx`, `Lights.hxx`), surfaces (`Materials.hxx`), media (`Media.hxx`), geometry (`Geometry.hxx`, `Geometry.cxx`) and overall scene setup along with ray-scene intersection functions (`Scene.hxx`).

Structs Folder containing implementation of simple basic data structures describing vectors (`Vector.hxx`), matrices (`Mat4f.hxx`), directions and positions (`Vector3.hxx`), color values (`Pixel.hxx`, `Rgb.hxx`) and bounding boxes (`BoundingBox.hxx`). Code duplication is mainly for performance reasons.

Outside the folders lies the `SmallUPBP.cxx` file which contains the `main` function.

3.1.2 Highest level

Once we roughly know how the code is organized, we can proceed to explanation of its operation. This section is an introduction and describes only the highest level of the program, i.e. where it all starts. The following sections then go to depth. They put emphasis on our contribution and more closely describe main features we had to add in order to get a usable implementation of the UPBP algorithm.

To make the description as clear as possible we use mostly pseudocode with references to the files listed above. The pseudocode is actually quite close to the real code, only simplified. Most of the fields and methods exist in the real code and those which do not (methods defined for pseudocode simplification) are marked with a green asterisk. Bold style is used for pieces of code that are important and will be described more closely. Comments are printed in green and keywords and data types in blue. A name surrounded by three dots marks a place where a piece of code was left out to improve clarity, the code will be supplied afterwards labelled with this name.

This introduction is also a good place to clarify one aspect of the implementation. It produces RGB images. It means that the following quantities introduced in previous chapters are actually RGB triplets: $f, L_\mathrm{e}, W_\mathrm{e}, T,\ T_\mathrm{r}$, $\rho, \sigma_\mathrm{t}, \sigma_\mathrm{s}, \rho_\mathrm{s}, \rho_\mathrm{p}, C_\mathrm{l}(\dots), C_\mathrm{c}(\dots), C_\mathrm{l}(\dots], C_\mathrm{c}[\dots)$. All the equations still hold, computations are simply preformed component-wise. The only difference is in sampling

short query beams described in Section 1.2.1. Since a pdf is a scalar quantity while attenuation coefficient an RGB triplet we use only the minimum positive component of the coefficient. The transmittance in the sampling pdf as well as in the probability of sampling the short query beam long enough is then computed only for this one component of the coefficient.

Now we get to the promised highest implementation level. The program starts in the `main` method:

Listing 3.1: `main` (method, `SmallUPBP.cxx`)

```
main()
{
    // A configuration = a set of values specifying what and how to render.
    Config config;

    // Set up the configuration according to the command line arguments.
    ParseCommandline(config);

    // Create a framebuffer representing the resulting image.
    Framebuffer fbuffer;
    config.mFramebuffer = &fbuffer;

    // Render the image to the framebuffer according to the configuration.
    render(config);

    // Save the framebuffer to a file.
    fbuffer.Save(config.mOutputName);
}
```

Note that the configuration setup besides reading options specified on the command line also covers loading a scene.

The `m` prefix in front of some variable names denotes a class (or struct) property. We also use `a` to prefix input method arguments, `o` for output method arguments (`ao` for both input and output arguments) and `k` for enumeration constants.

Let's take a closer look at the `render` method:

Listing 3.2: `render` (method, `SmallUPBP.cxx`)

```
render(Config &aConfig)
{
    int usedThreads = aConfig.mNumThreads;

    // Create 1 renderer per thread. The CreateRenderer function is a simple
    // switch where a constructor of a renderer specified by the configuration
    // is called.
    AbstractRenderer** renderers = new AbstractRenderer*[usedThreads];
    for (int i=0; i < usedThreads; i++) renderers[i] = CreateRenderer(aConfig);

    // Rendering loop. Run each iteration on a currently available renderer.
    for (int iter=0; iter < aConfig.mIterations; iter++)
    {
        int threadId = GetAvailableThreadId*();
        renderers[threadId]->RunIteration(iter);
    }

    // Accumulate framebuffers of all used renderers (not all created renderers
    // had to have been used).
    int usedRenderers = 0;
    for (int i=0; i < usedThreads; i++)
    {
            ..AccumUsed..
    }

    // Scale the framebuffer by the number of used renderers.
    aConfig.mFramebuffer->Scale(1.f / usedRenderers);
}
```

Listing 3.3: `AccumUsed` (part of the `render` method, `SmallUPBP.cxx`)

```
if (!renderers[i]->WasUsed()) continue;

// The first one is taken directly as the result (the framebuffer
// returned by the GetFramebuffer method in the output parameter
// is already divided by the number of iterations the renderer
// has carried out).
if (usedRenderers == 0)
    renderers[i]->GetFramebuffer(*aConfig.mFramebuffer);
else
{
    // The others are added to the first one.
    Framebuffer tmp;
    renderers[i]->GetFramebuffer(tmp);
    aConfig.mFramebuffer->Add(tmp);
}

usedRenderers++;
```

That is all about the highest level of the program. In a nutshell it reads command line arguments, creates renderers according to them, calls their `RunIteration` method, accumulates the result (averages framebuffers over iterations and renderers) and saves it to an image. The following text in this chapter describes key parts of the implementation necessary to understand the `RunIteration` method.

3.2 Media support

In this section we present changes we made to the original SmallVCM implementation and infrastructure we newly built in order to add support for participating media. We begin with an overview of the original representation of a scene. Then we explain how we add representation of media. Furthermore, we show how to track what media a ray intersects. And finally, we describe sampling and evaluating of the scattering function in media.

3.2.1 Representing scenes

A scene is represented by the `Scene` class (in `Scene.hxx`). Its geometry is stored as a list of geometric primitives – triangles and spheres. They are implemented as classes `Triangle` and `Sphere` derived from a common ancestor `AbstractGeometry` (all in `Geometry.hxx`) and offer mainly the `Intersect` method capable of intersecting the primitive with a given ray.

Besides geometry there is a list of so called materials (`mMaterials`). A material defines reflective and refractive properties of a surface. Phong reflection model is used along with Fresnel terms for mixing mirror reflection and refraction. A material is represented by the `Material` class (in `Materials.hxx`) which contains parameters of the Phong model (diffuse reflectance, phong reflectance and exponent), mirror reflectance and the index of refraction (IOR). Surfaces with a material with zero diffuse and phong reflectance are called purely specular. Evaluation of the model and its sampling will be described later in this section.

There is one more list – that of light sources (`mLights`). There are several types of lights:

Area An area light source with cosine light distribution, located in the scene. Can be hit by a traced ray.

Point An ideal point light source with uniform light distribution, located in the scene. Cannot be hit by a traced ray (it does not have an area).

Directional An ideal directional light source with delta light distribution, located in infinity. Cannot be hit by a traced ray (actually it could be, but since it has delta light distribution, its contribution would be zero anyway).

Background A background light source with light distribution uniform or controlled by an environment map, located in infinity. Can be hit by a traced ray.

Each type is implemented as an individual class derived from a common ancestor `AbstractLight` (in `Lights.hxx`) which prescribes methods for sampling and evaluation of contribution of the light. There can be an arbitrary number of lights of the first three types in the list, the background light, however, is either missing or there is only one and stored outside the list (as `mBackground`). The code of light sources is almost completely taken from the original SmallVCM implementation and we won't describe it closer.

The last thing a description of a scene includes is a camera (stored in `mCamera`). An ideal pinhole camera is used and it is represented by the `Camera` class (in `Camera.hxx`) which stores parameters of the camera and provides methods for points transformation (between world and image plane coordinates) and ray generation.

Note that the definition of factor $D(\mathbf{x}\to\mathbf{y})$ for geometry term $G(\mathbf{x},\mathbf{y})$ (1.3) in Section 1.1 specifies its value neither for a vertex on the camera nor a vertex on light sources. We therefore amend this definition as follows. $D(\mathbf{x}\to\mathbf{y}) = |n_\mathbf{x}\cdot\omega_\mathbf{xy}|$ if $\mathbf{x}$ is an inner path vertex located on a surface or the first path vertex located on an area light source. $D(\mathbf{x}\to\mathbf{y}) = 1$ otherwise (if $\mathbf{x}$ is an inner path vertex in a medium or the last vertex on the camera or the first vertex on other than an area light source). Furthermore, we introduce notation for situations when only a direction was sampled and the distant vertex is not yet known. Then $D(\mathbf{x}\to) = |n_\mathbf{x}\cdot\omega_{\mathbf{x}\to}|$ if $\mathbf{x}$ is an inner path vertex located on a surface or the first path vertex located on an area light source and $D(\mathbf{x}\to) = 1$ otherwise. Similarly, $D(\leftarrow\mathbf{y}) = |n_\mathbf{y}\cdot\omega_{\leftarrow\mathbf{y}}|$ if $\mathbf{y}$ is an inner path vertex located on a surface and $D(\leftarrow\mathbf{y}) = 1$ otherwise

Each geometry primitive in a scene must be associated with a material, i.e. must contain an index of a material in the `mMaterials` list, and those that serves as area light sources must be also associated with the corresponding light, i.e. must contain an index of an area light source in the `mLights` list. All the aforementioned data as well as these links between them are obtained during loading of a scene.

3.2.2 Representing media

We implemented media in a similar way as materials. A scene now contains also a list of media (`mMedia`) and every geometry primitive that forms a boundary of a medium must be associated with it, i.e. must contain its index in the list. Crossing a geometry primitive with an associated medium then means either entering or leaving the medium depending on whether the primitive was hit from its front or back face, i.e. whether a cosine of a ray direction and a normal of the

primitive at the intersection is negative or positive, respectively. This suggest that if we want to have a bounded medium in a scene, we have to enclose it in geometry associated with it.

In order not to be limited to media enclosed by solid surfaces a scene can also contain one global medium (its index is stored as `mGlobalMediumID`) and geometry with so called "imaginary materials". An imaginary material is also an object of the `Material` class and is distinguished from normal materials only by a flag (`mGeometryType`). Geometry associated with an imaginary material then acts only as a container for a medium and does not interact with light in any way. When tracing a light transport path crossing such geometry, only the current medium can be affected and the geometry is ignored, meaning that there is no reflection or refraction on its surface and no path vertex is created on it. We call geometry with imaginary materials "imaginary geometry", the rest is "real geometry" and we store it separately (as `mRealGeometry` and `mImaginaryGeometry`). Normal materials are then called "real materials". Similarly, an intersection with real geometry is called "a real intersection" and an intersection with imaginary geometry "an imaginary intersection".

If a light source or the camera are located in other than the global medium, they have to be associated with this medium and also with a material of geometry that encloses it. A ray leaving such light or camera can be viewed as immediately entering the geometry enclosing the medium (without any interaction with its surface).

Now let's take a look at what exactly a medium is. It is represented by a class derived from a common ancestor `AbstractMedium` (in `Media.hxx`). We implemented homogeneous medium in the `HomogeneousMedium` class (also in `Media.hxx`). Its most important parts are shown in the following pseudocode (actual code is more complicated since the `AbstractMedium` class is general and supports heterogeneous media too):

Listing 3.4: `HomogeneousMedium` (class, `Media.hxx`)

```
class HomogeneousMedium
{
   Rgb   mAbsorptionCoef;
   Rgb   mEmissionCoef;
   Rgb   mScatteringCoef;
   Rgb   mAttenuationCoef; // = mAbsorptionCoef + mScatteringCoef
   float mMeanCosine;
   float mContinuationProb;

   // Evaluates emission that is accumulated by a ray
   // travelling in this media over the given distance.
   Rgb EvalEmission(const float aDistanceAlongRay) const
   {
      return mEmissionCoef * aDistanceAlongRay;
   }

   // Evaluates attenuation that is accumulated by a ray
   // travelling in this media over the given distance.
   Rgb EvalAttenuation(const float aDistanceAlongRay) const
   {
      return Rgb::exp(-mAttenuationCoef * aDistanceAlongRay);
   }

   ..HomogeneousMediumPart2..
}
```

Listing 3.5: `HomogeneousMediumPart2` (part of the `HomogeneousMedium` class, `Media.hxx`)

```
// Evaluates attenuation that is accumulated by a ray
// travelling in this media over the given distance
// in one color channel only, i.e. using only the given
// component of the attenuation coefficient.
float EvalAttenuationInOneDim(
   const float aAttenuationCoefComp,
   const float aDistanceAlongRay) const
{
   return std::exp(-aAttenuationCoefComp * aDistanceAlongRay);
}

// Samples the distance a ray will travel in this medium before
// a collision (absorption or scattering). If it is lesser than or
// equal to the given distance to a boundary of the medium,
// the sampled distance is returned along with a pdf of sampling
// this distance. Otherwise, the distance to a boundary is returned
// with a probability of sampling distance greater than the distance
// to the boundary.
float SampleRay(
   const float aDistToBoundary,
   const float aRandom,
   float       *oPdf,
   const uint  aRaySamplingFlags = 0,
   float       *oRevPdf = NULL) const
{
   ..SampleRay..
}

// Gets a pdf/probability that a ray will travel the given distance
// in this medium. The given flags specify whether the distance
// is within the medium or the ray left it. Returns the pdf/probability
// the SampleRay method would return with the given distance.
float RaySamplePdf(
   const float aSampledDist,
   const uint  aRaySamplingFlags = 0,
   float       *oRevPdf = NULL) const
{
   float oPdf = 1.0f;
   if (oRevPdf) *oRevPdf = 1.0f;

   // Same condition as in the SampleRay method.
   if (mMinPositiveAttenuationCoefComp() && HasScattering())
   {
      // Compute attenuation for the distance.
      float att = EvalAttenuationInOneDim(
         mMinPositiveAttenuationCoefComp(), aSampledDist);

      // Compute a pdf/probability of the sampling.
      if (aRaySamplingFlags & kEndInMedium)
         // Pdf of a sample before the boundary.
         oPdf = mMinPositiveAttenuationCoefComp() * att;
      else
         // Probability of sampling beyond the boundary.
         oPdf = att;

      // Compute a pdf/probability of the sampling as if it was done
      // in a reverse direction.
      if (oRevPdf)
      {
         if (aRaySamplingFlags & kOriginInMedium)
            // Pdf of a sample before the boundary.
            *oRevPdf = mMinPositiveAttenuationCoefComp() * att;
         else
            // Probability of sampling beyond the boundary.
            *oRevPdf = att;
      }
   }

   return oPdf;
}
```

Listing 3.6: `SampleRay` (part of the `SampleRay` method, `Media.hxx`)

```
// We sample only if the attenuation and scattering coefficients
// of the medium have positive components. The first condition
// is needed since we sample according to the minimum positive
// component of the attenuation coefficient. The second one
// because it is useless to create a path vertex in a purely
// absorbing medium.
if (mMinPositiveAttenuationCoefComp() > 0 && HasScattering())
{
   // Sample the distance.
   float d = -std::log(1-aRandom) / mMinPositiveAttenuationCoefComp();
   if (d <= aDistToBoundary) // The sample is before the boundary.
   {
      // Compute attenuation for the sampled distance
      // (needed for pdfs only).
      float att = EvalAttenuationInOneDim(
         mMinPositiveAttenuationCoefComp(), d);

      // Compute a pdf of a sample before the boundary.
      if (oPdf) *oPdf = mMinPositiveAttenuationCoefComp() * att;

      // Compute a pdf/probability of the sampling as if it was done
      // in a reverse direction.
      if (oRevPdf)
      {
         if (aRaySamplingFlags & kOriginInMedium)
            // Pdf of a sample before the boundary.
            *oRevPdf = *oPdf;
         else
            // Probability of sampling beyond the boundary.
            *oRevPdf = att;
      }

      // Return the sampled distance.
      return d;
   }
   else // The sample is beyond the boundary.
   {
      // Compute attenuation for the distance to the boundary
      // (needed for pdfs only).
      float att = EvalAttenuationInOneDim(
         mMinPositiveAttenuationCoefComp(), aDistToBoundary);

      // Compute the probability of sampling beyond the boundary.
      if (oPdf) *oPdf = att;

      // Compute a pdf/probability of the sampling as if it was done
      // in a reverse direction.
      if (oRevPdf)
      {
         if (aRaySamplingFlags & kOriginInMedium)
            // Pdf of a sample before the boundary.
            *oRevPdf = mMinPositiveAttenuationCoefComp() * att;
         else
            // Probability of sampling beyond the boundary.
            *oRevPdf = att;
      }

      // Return the distance to the boundary.
      return aDistToBoundary;
   }
}
else // We cannot sample a distance.
{
   if (oPdf) *oPdf = 1.0f;
   if (oRevPdf) *oRevPdf = 1.0f;
   return aDistToBoundary;
}
```

The code is quite long but not difficult. In Listing 3.4 we can see that a homogeneous medium is described with a several coefficients: the absorption coefficient

σ_a (`mAbsorptionCoef`), the emission coefficient ε (`mEmissionCoef`), the scattering coefficient σ_s (`mScatteringCoef`) and the attenuation (extinction) coefficient $\sigma_\mathrm{t} = \sigma_\mathrm{a} + \sigma_\mathrm{s}$ (`mAttenuationCoef`). We use the Henyey-Greenstein phase function so there is also the mean cosine g (`mMeanCosine`) that determines the (an)isotropy of scattering. And finally there is the continuation probability p_C used for the Russian roulette decision whether a traced subpath will scatter out and continue or end (`mContinuationProb`). For the sake of simplicity, this probability is set by the user in our implementation.

Since the medium is homogeneous, the way it attenuates a ray travelling through it depends only on the distance the ray travels in it, the ray location and direction are irrelevant. The emission L_e accumulated along a ray over the distance d then equals

$$L_e = \varepsilon d \tag{3.1}$$

and the attenuation T_r from Equation 1.4 becomes

$$T_\mathrm{r}(\mathbf{x}, \mathbf{y}) = \exp\Big(-\int_0^{\|\mathbf{x}-\mathbf{y}\|} \sigma_\mathrm{t}(\mathbf{x} + t\omega_{\mathbf{xy}})\,\mathrm{d}t\Big) \tag{3.2}$$

$$= \exp\Big(-\int_0^{d} \sigma_\mathrm{t}\,\mathrm{d}t\Big) \tag{3.3}$$

$$= \exp(-\sigma_\mathrm{t} d) \equiv T'_\mathrm{r}(d). \tag{3.4}$$

These formulas are implemented in the `EvalEmission` and `EvalAttenuation` methods listed above.

The `SampleRay` method (Listing 3.6) samples a distance a ray travels in the medium before a collision, i.e. before absorption or scattering. In other words it samples the ray in the medium for a next path vertex. Evaluating estimators at the vertex and continuing the subpath from it represent scattering, failing to continue because of the Russian roulette decision can be viewed as absorption (but note that the probability of such a decision $1 - p_C$ is not necessarily equal to the probability of absorption since p_C is set by the user). The `SampleRay` method performs the sampling only if the scattering coefficient of the medium is not zero. It is because no estimators can be evaluated in a purely absorbing medium (it would imply scattering) and therefore it is useless to create a path vertex in such a medium. Absorption is handled there only via attenuating the throughput of the ray.

We want to sample proportionally to attenuation of the medium, i.e. we want to importance sample the ray with a pdf proportional to $T'_\mathrm{r}(d)$. Since $T'_\mathrm{r}(d)$ is an RGB triplet and pdf a scalar quantity, we define

$$T'_{\mathrm{r,m}}(d) = \exp(-\sigma_{\mathrm{t,m}} d), \tag{3.5}$$

where $\sigma_{\mathrm{t,m}}$ is the minimum positive component of σ_t. Then we seek a pdf proportional to $T'_{\mathrm{r,m}}(d)$. Normalization yields a pdf $\bar{p}(d) = \sigma_{\mathrm{t,m}} T'_{\mathrm{r,m}}(d)$. We use the standard inversion method for drawing samples from a given distribution. By integrating p from 0 to d we get the cumulative distribution function (cdf) $P(d) = 1 - T'_{\mathrm{r,m}}(d)$ and from its inversion we obtain a formula for sampling the distance d:

$$d = -\ln(1 - r)/\sigma_{\mathrm{t,m}} \tag{3.6}$$

where $r \in [0, 1)$ is a randomly generated number. Let d_{max} denote a distance along the ray from a point where it enters the medium (or from the ray origin if it is already in the medium) to a point where it leaves. If the sampled distance d is lesser than or equal to d_{max}, the method returns d along with the desired pdf $\bar{p}(d) = \sigma_{\mathrm{t,m}} T'_{\mathrm{r,m}}(d)$. If it is longer, d_{max} is returned with a probability $\Pr\{d > d_{max}\} = 1 - P(d_{max}) = T'_{\mathrm{r,m}}(d_{max})$.

Besides the `HomogeneousMedium` class there is also the `ClearMedium` class. It is a special case of `HomogeneousMedium` with all coefficients and the mean cosine zero and the continuation probability equal to 1. Implementation of other types of media, e.g. heterogeneous media, is left for future work. The UPBP algorithm is not limited to a homogeneous case.

This section revealed how media are represented and sampled. We could see that they are objects of quite a simple class and that these objects are associated with geometry. But how do we know in what medium we are?

3.2.3 Intersecting media

In order to provide larger scene variability and also because of numerical stability our implementation allows geometry and consequently media to overlap. However, we had to solve how to correctly track the current medium along a traced light transport path as well as which geometry can interact with light and which should be ignored. Our goal was to ensure that if two objects intersect, only one of them is present in the overlap region. This decision should apply consistently to both geometry and media, and must be independent of the direction in which the path is propagated. It is necessary since we do not support media mixing. Possible results for two overlapping spheres are shown in Figure 3.1.

Firstly, if there is more than one object at one place, we need to decide which one to use. Such a decision is made based on priorities (as proposed by Schmidt et al. [22]). Each material is given a priority (the `mPriority` property of the `Material` class), objects are then judged according to the priority of a material associated with their geometry. The object with the highest priority is used. If there are more objects with the same priority, the one entered by the traced path as the last is used.

Secondly, we need to recognize that the path is entering an overlap region of some objects and we need to figure out in what object the path will stay after leaving the region. For this reason we employed a priority stack of crossed object boundaries. We call it "boundary stack" and it is implemented in the `BoundaryStack.hxx` and `PriorityStack.hxx` files. A boundary is represented by a triplet consisting of the material, medium and priority associated with geometry of the boundary. The stack keeps these triplets sorted according to the priority with the triplet with the highest priority on its top.

When a new path is about to be traced, the stack is cleared and initialized with a global medium (there is always a global medium, either specified by the scene or clear). A triplet $(-1, \texttt{mGlobalMediumID}, \texttt{GLOBAL\_MEDIUM\_PRIORITY})$ is put on the bottom of the stack. This triplet is never removed from the stack and always stays at its bottom. After that, if the light source or the camera that the path is started from is associated with some medium (i.e. when it is located in other than the global medium), a corresponding triplet is put on the stack too.

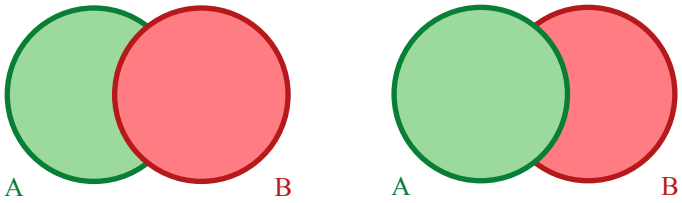


Figure 3.1: Two possible results of rendering two overlapping spheres. Only a planar cut through the spheres is shown, the thick line on their circumference represents their geometry, the less saturated color inside them represents media. Notice that either geometry and medium of sphere B are used in the overlap region with sphere A completely ignored there (the left image) or vice versa (the right image).

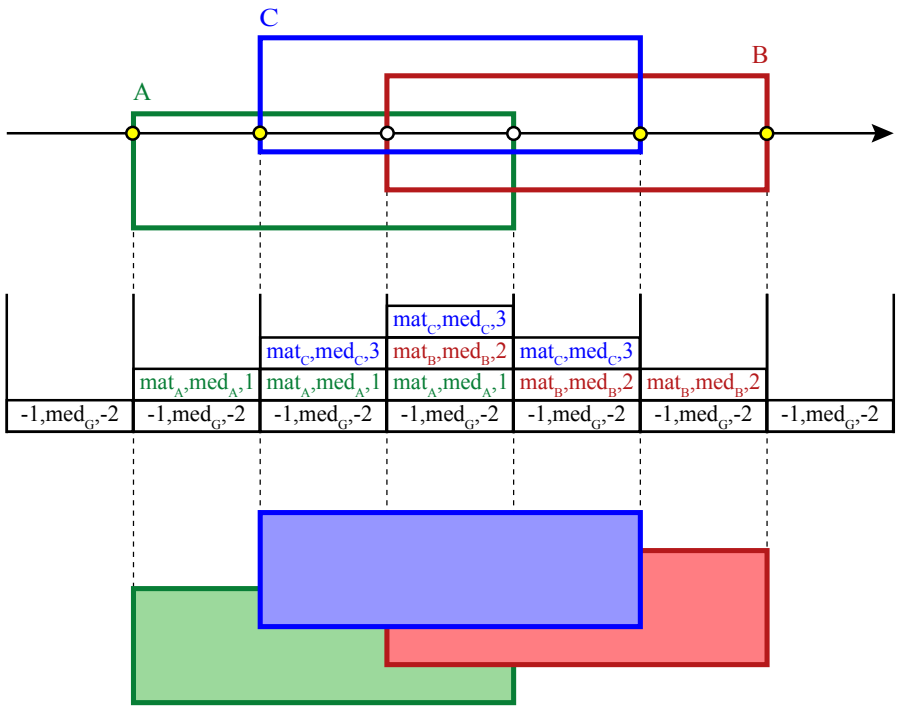


Figure 3.2: An example of how the boundary stack works. The upper image shows three overlapping rectangles A, B, C. The green rectangle A has material mat_A, medium med_A and priority 1, the red rectangle B has material mat_B, medium med_B and priority 2 and the blue rectangle C has material mat_C, medium med_C and priority 3. Outside the rectangles there is a global medium med_G with priority -2. The rectangles are intersected by a ray from left to right, active intersections are yellow, passive are white. The middle row shows how the stack changes. Each column represents the state of the stack between the indicated intersections. The bottom image shows the result. At each state it corresponds to the top of the stack.

During tracing every time the path enters an object, i.e. crosses geometry from its front face, a triplet consisting of the associated material, medium and priority is created. The triplet is put on the top of the stack and then moved towards the bottom until there is no triplet with a higher priority below it. When the path leaves an object, i.e. crosses geometry from its back face, a triplet consisting of the associated material, medium and priority is looked up in the stack and removed. The medium on the top of the stack is the current medium, the geometry that added or removed the top triplet can affect the result (real geometry can directly interact with light, imaginary geometry can at least change the current medium). We call intersections with such geometry "active" and the rest "passive". Figure 3.2 shows an example.

3.2.3.1 Intersection method

After we explained what to do when crossing geometry in order not to lose track of the current medium, we can proceed to the ray-scene intersection method. We implemented a completely new one – the `Intersect` method of the `Scene` class. The rest of this section is focused on its implementation.

What should be an output of such method? What the traced ray hit and what it went through on the way.

Intersection. When a ray is traced through a scene, it either leaves the scene or a scattering event occurs. That means the ray gets scattered in a medium or hits a surface. We speak about both these situations as finding an "intersection" with the scene. The task of the `Intersect` method is to find the first "intersection" of the ray with the scene while respecting priorities of scene objects. Firstly, the method tries to find the first active real intersection of the ray with the scene, i.e. the first intersection with the real geometry (as opposed to imaginary, see Section 3.2.2) not ignored because of priorities. After that, it samples media along the ray from its origin to the first active real intersection (or to infinity if there is none) for a possible medium scattering point. Data of the "intersection" are returned in a common structure `Isect`:

Listing 3.7: `Isect` (struct, `Ray.hxx`)

```
1  struct Isect
2  {
3      float mDist;     // Distance to the found medium scattering point
4                       // or the real intersection.
5      int   mMatID;    // Index of the material of the hit geometry for the real
6                       // intersection, -1 for the medium scattering point.
7      int   mMedID;    // Index of the interacting medium for the medium scattering
8                       // point, index of the medium enclosed by the hit geometry
9                       // for the real intersection.
10     int   mLightID;  // Index of the hit light, -1 means none.
11     Dir   mNormal;   // Normal at the real intersection.
12     bool  mEnter;    // Whether the ray enters geometry at the real intersection
13                      // (cosine of its direction and the normal is negative).
14 }
```

Volume segments. No matter whether the "intersection" is found or not, the method returns parts of the ray going through any medium as so called "volume segments" of the ray. There are two types – `VolumeSegment` and `LiteVolumeSegment`.

The former one is defined as follows:

Listing 3.8: `VolumeSegment` (struct, `Ray.hxx`)

```
struct VolumeSegment
{
    float mDistMin;          // Distance of the segment beginning from
                             // the ray origin.
    float mDistMax;          // Distance of the segment end from the ray origin.
    int   mMediumID;         // ID of the medium in this segment.
    float mRaySamplePdf;     // If scattering occurred in this medium: pdf of
                             // having sampled a distance within the medium;
                             // otherwise: probability of passing through
                             // the entire medium.
    float mRaySampleRevPdf; // Similar to mRaySamplePdf but in the reverse
                             // direction of the ray.
    Rgb   mAttenuation;      // Attenuation caused by this segment (not divided
                             // by the pdf).
}
```

The `LiteVolumeSegment` structure is a subset of `VolumeSegment`, it contains only the first three fields (the distances and the medium index). It is used with photon beams because of lower memory consumption. The `Intersect` method returns all parts of the ray going through any medium from the origin of the ray up to its first active real intersection (or to infinity if there is none) as `LiteVolumeSegment` structures. All parts are also returned as `VolumeSegment` structures except the case of scattering in a medium. Then only those parts located before the scattering point are returned as `VolumeSegment` structures (Figure 3.3 illustrates this difference). This is because we do not need the additional data for segments behind the scattering point and therefore outside the traced light subpath but we need to know those segments for construction of *long* photon beams.

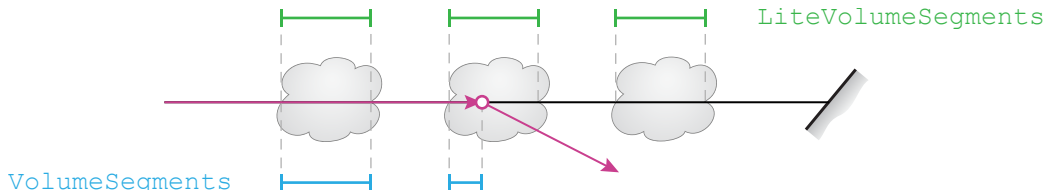


Figure 3.3: Illustration of the difference between `LiteVolumeSegment` and `VolumeSegment` structures returned by the `Intersect` method in case of scattering in a medium.

Assumptions. However, there are several assumptions that have to hold for our method to work correctly:

1. A scene is completely filled with media (places where no medium is specified are filled with the clear medium). It is quite natural and lowers the number of special cases to handle.

2. There is always one global medium (possibly clear). No material is assigned a lower priority than the `GLOBAL_MEDIUM_PRIORITY` (the priority at the bottom of the stack). It ensures that the global medium triplet is always at the bottom of the stack and the stack is never emptied.

3. All other media are associated with geometry. All geometric primitives enclosing one instance of a medium (e.g. all triangles forming a surface of one wax candle) are associated with the same medium and material. It is necessary in order to correctly identify when a path enters and exits one object, i.e. to correctly match the triplet of the exit point with the triplet of the entry point on the boundary stack. Infinite media, i.e. media with open boundary geometry, are allowed but should be used with caution. Figure 3.4 shows an example.

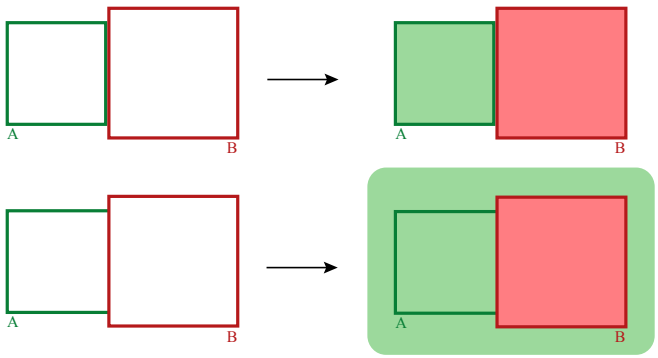


Figure 3.4: An example of a "medium leak". The left two images show two empty touching rectangles A and B, rectangle B having a higher priority, the right two images show what happens if geometry of the rectangles is associated with media. The thick line on their circumference represents their geometry, the less saturated color inside them represents media. If rectangle A is complete as in the upper two images, media stay inside the rectangles. However, if rectangle A lacks the side adjacent to rectangle B as in the lower two images, its medium fills the entire scene except for objects with an equal or higher priority, i.e. rectangle B does not block it. This "medium leak" is not global, it applies only to paths that enter rectangle A from the outside and then leave it through the missing side (the medium of rectangle A is not removed from the stack). Other paths produce the same result as in the upper right image.

4. All imaginary materials have lower priority than any real material. It is also quite natural and allows the method to find an intersection with real geometry first and than handle intersections with imaginary geometry separately.

5. No imaginary geometry intersect real geometry. But imaginary geometry can intersect imaginary geometry, real geometry can intersect real geometry and imaginary geometry can surround real geometry (without intersection). This assumption means that if a path is behind real geometry (if there is a real material on the top of the boundary stack), then the method does not have to test imaginary geometry for intersections.

6. Intersecting geometry is assigned different priorities. Otherwise, the result would be directionally dependent. Imagine two overlapping spheres. If they have the same priority, then the one entered by a traced path as the last is used in the overlap region. However, that depends on the direction of the path, i.e. from which side of the region the path comes (e.g. for a path coming in the opposite direction the other sphere is used).

Algorithm. Let's take a closer look at how the `Intersect` method works. It is given a ray and a current boundary stack and performs the following steps:

1. Tries to find the first active real intersection with the scene. That means:
   (a) Creates copies of the given ray and boundary stack. In order to find the intersection the method needs to modify a ray and a boundary stack starting in the original states, i.e. the states of the given ray and boundary stack. However, the original states will be still needed lately. Therefore, the method uses copies.
   (b) Casts the ray copy in the scene and tries to find its first intersection with the real geometry (using the `mRealGeometry->Intersect` method).
   (c) If it is unsuccessful, goes to 2.
   (d) If it is successful and the priority of the intersection (i.e. of the material associated with the intersected geometry) is equal to or higher than the priority on the top of the stack copy, the first active real intersection is found and the method goes to 2. Although only the real geometry was tested, we know the intersection is active since the imaginary geometry has always lower priority (according to the assumptions).
   (e) If it is successful and the priority of the intersection is lower than the priority on the top of the stack copy, the intersection is passive. The method stores the intersection, updates the stack copy for crossing the geometry, takes the intersection as a new origin of the ray copy and goes to 1b.
2. Finds all volume segments along the given ray in a range from its origin to the found first active real intersection (or to infinity if there is none) and stores the `LiteVolumeSegment` structure for each of them.
   (a) If there is a real material on the top of the given stack, then the origin of the given ray is located behind the real geometry. Therefore, no geometry with a real or imaginary material with a priority equal to or higher than the one at top is crossed in the range since the found active real intersection is the first and there are no imaginary materials in the range at all (according to the assumptions). It means that there is exactly one volume segment covering the entire range (if no medium is specified by the scene there, the clear medium is used). The method stores the `LiteVolumeSegment` structure for it and goes to 3.
   (b) Otherwise, there might be volume segments separated by imaginary geometry in the range (not by real geometry since the found active real intersection is the first). So the method finds all intersections of the given ray with the imaginary geometry of the scene in the range (the `mImaginaryGeometry->IntersectAll` method is used). The result is a list of the intersections sorted in ascending order according to their distance from the origin of the ray.
   (c) Creates a new copy of the given boundary stack. In order to correctly handle overlap of imaginary geometry the method needs to modify a boundary stack starting in the original state. However, the original state will be still needed lately. Therefore, the method uses a copy.

(d) Goes through the list of intersections and for each of them performs:

i. Checks if the intersection is on the front face of geometry and its priority is equal to or higher than the top of the stack copy. It would mean that the intersection is active, a volume segment is being closed by beginning of a new one enclosed in geometry with an equal or higher priority and needs to be stored.

ii. Checks if the intersection is on the back face and the corresponding triplet is the same as on the top of the stack copy. It would mean that the intersection is active, a volume segment is being closed by its own enclosing geometry and needs to be stored.

iii. If any of the two conditions is satisfied, creates and stores the `LiteVolumeSegment` structure which extends from the end of a previously stored segment (or from the range beginning, if it is the first segment) to the intersection and is filled with the current medium (the medium currently on the top of the stack copy). In other cases, i.e. when entering a medium with a lower priority or leaving other than the current medium, the intersection is passive, the current medium is not affected and therefore no segment is created.

iv. Uses the intersection for updating the stack copy.

(e) Stores a segment for the remaining part of the range filled with the current medium.

3. Samples the volume segments found in 2 for a possible medium scattering point and creates `VolumeSegment` structures from them, i.e. goes through the stored `LiteVolumeSegment` structures and for each of them performs:

(a) Samples the given ray within the segment using the `SampleRay` method (Listing 3.4) of the medium in the segment. The distance passed to the `SampleRay` method is the length of the segment.

(b) Creates the `VolumeSegment` structure. Its beginning and medium is taken from the `LiteVolumeSegment` structure, pdfs were returned by the `SampleRay` method, attenuation and emission is computed using the `EvalAttenuation` and `EvalEmission` methods (Listing 3.4) of the medium. If scattering occurred, i.e. if the `SampleRay` method returned distance within the segment, the end of the `VolumeSegment` is set in this distance; otherwise, the end of the `LiteVolumeSegment` is used (as shown in Figure 3.3).

(c) Updates the given boundary stack by imaginary intersections found in 2b, which lie within the range of the created `VolumeSegment` (excluding the beginning, including the end). This is the last time the original state of the given boundary stack is needed, therefore no stack copy is necessary.

(d) If scattering occurred, stops creating `VolumeSegment` structures (as shown in Figure 3.3) and goes to 4.

4. Updates the given boundary stack by real intersections found in 1e. If scattering occurred, only those that lie before the scattering point are used.

Together with updates in 3c this step ensures that the given stack at the end of the method corresponds to the position of the returned "intersection". Note that the updates by imaginary geometry and real geometry can be done separately only because no imaginary geometry can intersect real geometry and imaginary materials have always lower priority than real materials (according to the assumptions).

5. Returns true, if scattering occurred or if the ray got through media but an active real intersection with geometry was found; otherwise, false (the ray continued to the infinity). Furthermore, returns the `LiteVolumeSegment` and `VolumeSegment` structures, the updated boundary stack and data of the scattering point/real intersection in the `Isect` structure.

We described a typical run of the `Intersect` method. However, the method is overloaded and depending on the overloading and given flags some steps can be skipped. For example, volume segments may not sampled, the `VolumeSegment` structures may not be created or media might be completely ignored. The method for testing whether a connection between two vertices is occluded by geometry or not – the `Occluded` method – is also only a call to the `Intersect` method with the right arguments. It then looks for an active intersection only along the connection and only if none exists, it finds volume segments without sampling them (only computes the probabilities of getting through them using the `RaySamplePdf` method).

The code of the `Intersect` method is quite complex, therefore we omit it here. It can be found in the `Scene.hxx` file. We at least went through its outline since it is essential part of the renderer.

Numerical issues. Unfortunately, we did not manage to solve all numerical problems. When a ray intersects geometry, there is an epsilon distance the ray must travel before a next intersection can be detected. This is because of numerical issues when one intersection with a geometric primitive could be detected twice. On the other hand, if two geometric primitives are within the epsilon distance from each other, then an intersection with one of them can be ignored yielding an error. This error is negligible for areas around an intersection of two objects, but can be bigger if a region between two objects has large area but is narrower than the epsilon, especially if the objects contain media. Therefore, objects touching each other in the reality should be modelled as objects that overlap by more than the epsilon distance. The epsilon value can be found in the `Defs.hxx` file (`EPS_RAY`).

3.2.4 Evaluating media

After solving how to represent media and how to keep track of what medium a ray is located in, evaluation and sampling of the phase function at path vertices is quite straightforward. When the `Intersect` method (Section 3.2.3.1) finds an intersection of a ray with the scene (either a medium scattering point or a real intersection), an object of the `BSDF` class (from the `Bsdf.hxx` file) is created based on the ray and the returned `Isect` structure (Listing 3.7). The constructor mainly stores some data describing the material or medium at the intersection

and computes the local coordinate frame. If the intersection is on a surface, its normal is taken as the z-axis of the frame; otherwise, the ray direction is used. Furthermore, the ray direction is stored as the fixed incoming direction but with the opposite sign to point away from the intersection. If the intersection is on a surface, the constructor also computes probabilities of sampling different components of the Phong BSDF (diffuse reflection, Phong reflection, mirror reflection, mirror refraction) and the continuation probability. This is carried out by the `GetComponentProbabilities` method in the `Bsdf.hxx` file. Probability of sampling a component is computed as a ratio of the corresponding albedo and the total albedo. The continuation probability is computed as the maximum component of the summed material reflectance coefficients. If the intersection is in a medium, there is only one component and the continuation probability is a constant (the `mContinuationProb` property of the `HomogeneousMedium` class, see Listing 3.4).

There are three main methods the `BSDF` class offers: `Evaluate`, `Pdf` and `Sample`. They handle the case of a surface intersection directly, for an intersection in a medium they call static methods of the same name from the `PhaseFunction` class which implements the Henyey-Greenstein phase function (in the `PhaseFunction.hxx` file). We briefly describe the methods:

Evaluate. Evaluates the scattering function for the stored fixed incoming direction and the given generated outgoing direction (`aWorldDirGen`), both pointing away from the intersection. Returns the scattering function factor, a cosine between the frame z-axis and the outgoing direction (`oCosThetaGen`) and pdfs w.r.t. the solid angle measure of having sampled the outgoing direction given the incoming (`oDirectPdfW`) and vice versa (`oReversePdfW`). If the intersection is in a medium, a sine between the frame z-axis and the outgoing direction is also returned (`oSinTheta`). If the intersection is on a surface, the returned factor is a sum of factors of the two non-specular components (diffuse and Phong, others have zero probability) and the returned pdfs are sums of pdfs of sampling the directions from the two components weighted by component probabilities. Special cases:

- `oCosThetaGen` is always 1 if the intersection is in a medium.
- `oSinTheta` is always 0 if the intersection is on a surface.
- all returned values are always 0 if the intersection is in a medium without scattering.

Listing 3.9: `Evaluate` (method, `Bsdf.hxx`)

```
Rgb Evaluate(
    const Dir &aWorldDirGen,
    float     &oCosThetaGen,
    float     *oDirectPdfW = NULL,
    float     *oReversePdfW = NULL,
    float     *oSinTheta = NULL)
{ ... }
```

Pdf. When `aPdfDir == kReverse`, this method returns a pdf w.r.t. the solid angle measure with which would be the given outgoing direction (`aWorldDirGen`) generated from the stored fixed incoming direction. When `aPdfDir == kReverse`, it provides a pdf for the reverse direction. If the intersection is on a surface, the returned pdf is a sum of pdfs of sampling the direction from the two non-specular components (diffuse and Phong, others have zero probability) weighted by component probabilities.

Listing 3.10: `Pdf` (method, `Bsdf.hxx`)

```
float Pdf(
   const Dir    &aWorldDirGen,
   const PdfDir aPdfDir = kForward)
{ ... }
```

Sample. Samples an outgoing direction for the stored fixed incoming direction. Returns the scattering function factor, the generated direction (`oWorldDirGen`), a pdf of sampling w.r.t. the solid angle measure (`oPdfW`), the cosine between the frame z-axis and the generated direction (`oCosThetaGen`) and a flag indicating the sampled event (`oSampledEvent`). The events are the four BSDF components plus scattering in a medium. If the intersection is in a medium, sine between the frame z-axis and the generated direction is also returned (`oSinTheta`). If the intersection is on a surface, firstly a component is randomly picked according to the component probabilities, then the component is sampled. If a non-specular component is chosen (diffuse, Phong), the other non-specular component is also evaluated (the factor and pdf are then sums again). Special cases:

- `oCosThetaGen` is always 1 if the intersection is in a medium.
- `oSinTheta` is always 0 if the intersection is on a surface.
- all returned values are always 0 if the intersection is in a medium without scattering.

Listing 3.11: `Sample` (method, `Bsdf.hxx`)

```
Rgb Sample(
   const Dir &aRndTriplet,
   Dir       &oWorldDirGen,
   float     &oPdfW,
   float     &oCosThetaGen,
   uint      *oSampledEvent = NULL,
   float     *oSinTheta = NULL)
{ ... }
```

Note that besides different interface there is one important distinction between using the `Evaluate` and `Sample` methods of the `Bsdf` class and using static methods of the same name from the `PhaseFunction` class. While the former return a complete value of the scattering function (1.5), the latter returns only value of the phase function, i.e. it has to be multiplied by the scattering coefficient to produce the same result.

The part of the code of the three methods above that handles the case of a surface intersection is taken almost without modifications from the original

SmallVCM implementation. Therefore, we won't describe it closer. However, the case that handles medium scattering is completely new and so, for the sake of completeness, we would like to present here the formulas the methods implement. Let $\cos\theta_o$ denote the cosine between z-axis of the local coordinate frame and the outgoing direction ω_o. The frame z-axis is the ray direction, i.e. $-\omega_i$, a direction opposite to the stored fixed incoming direction ω_i. Then the phase function factor ρ_p as well as the pdf of sampling the outgoing direction given the incoming direction $\hat{p}(\omega_o)$ (w.r.t. the solid angle measure) and the pdf of sampling the incoming direction given the outgoing direction $\hat{p}(\omega_i)$ (w.r.t. the solid angle measure) equal

$$\rho_\mathrm{p}(\omega_i,\omega_o) = \hat{p}(\omega_i) = \hat{p}(\omega_o) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g\cos\theta_o)^{\frac{3}{2}}},$$

where g is the mean cosine. In the `Evaluate` and `Pdf` methods both directions ω_i and ω_o are known so $\cos\theta_o$ can be easily computed. In the `Sample` method, only ω_i is known. Firstly, $\cos\theta_o$ is generated:

$$\cos\theta_o = \begin{cases} \frac{1}{2g}\left(1+g^2-\left(\frac{1-g^2}{1-g+2gr_1}\right)^2\right) & \text{if } g \neq 0, \\ 1-2r_1 & \text{if } g = 0, \end{cases}$$

then the coordinates of ω_o in the local coordinate frame are computed as

$$\begin{aligned} x &= \cos(2\pi r_2)\sin\theta_o, \\ y &= \sin(2\pi r_2)\sin\theta_o, \\ z &= \cos\theta_o, \end{aligned}$$

where $r_1, r_2 \in [0,1)$ are randomly generated numbers and $\sin\theta_o = \sqrt{1-\cos^2\theta_o}$.

3.2.5 Multiple media along a ray

The theory derived in Chapters 1 and 2 assumed that path segments always completely lie in a single medium. However, such assumption does not always hold as there can be more than one volume segment a traced ray got through before interacting with a surface or a medium. For this reason we have to amend the previous definitions. Assume two vertices $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ on a subpath and n volume segments $s_1,\ldots,s_n$ along a ray between them. Let d_j denote a length of s_j, d'_j a length that was sampled when the ray entered s_j, t_i a distance between $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$, and l_i length of a beam shot from vertex $\mathbf{x}_i$. Then we redefine:

$$p(t_i) = \begin{cases} \prod_{k=1}^{n}\Pr\{d'_k > d_k\} = \prod_{k=1}^{n} T'_\mathrm{r,m}(d_k) & \text{if } \mathbf{x}_{i+1} \text{ is on a surface,} \\ \left(\prod_{k=1}^{n-1}\Pr\{d'_k > d_k\}\right)\bar{p}(d_n) = & \\ \left(\prod_{k=1}^{n-1} T'_\mathrm{r,m}(d_k)\right)\sigma_\mathrm{t,m}T'_\mathrm{r,m}(d_n) & \text{if } \mathbf{x}_{i+1} \text{ is in a medium} \end{cases}$$

$$\Pr\{l_i > t_i\} = \prod_{k=1}^{n}\Pr\{d'_k > d_k\} = \prod_{k=1}^{n} T'_\mathrm{r,m}(d_k) \equiv T_\mathrm{r,m}(t_i)$$

$$T_\mathrm{r}(\mathbf{x}_i,\mathbf{x}_{i+1}) = T_\mathrm{r}(t_i) = \prod_{k=1}^{n} T'_\mathrm{r}(d_k).$$

Expressed by words, attenuation along the path segment is simply a product of attenuation by all volume segments along it. Similarly, probability of getting over the path segment is a product of probabilities of overcoming the volume segments. If $\mathbf{x}_{i+1}$ is on a surface, also the pdf has the same value. Otherwise, the last volume segment contributes with a pdf of sampling inside it instead of probability of getting through.

3.2.6 Summary

We explained how we cope with media from their representation over tracking in a scene to evaluation of their phase function. The environment is therefore described and we can move to the rendering itself.

3.3 Renderers

The UPBP algorithm is implemented as a so called "renderer". A renderer is a class derived from the `AbstractRenderer` class (located in `Renderer.hxx`) which prescribes implementing the `RunIteration` method - a key method where all the rendering takes place.

We followed the convention of SmallVCM and kept renders for different estimators and their combinations separate. It made the development and debugging simpler and may also help understanding the code.

There are seven renderers (each written in its own file of the same name). Three of them come from the original SmallVCM implementation and have no support for media. They are `EyeLight` (simple shading based on a visualization of the dot product of a surface normal and an incoming ray direction), `PathTracer` (traditional path tracing with next event estimation [21]) and `VertexCM` (VCM [4]). During the development we gradually added four new renderers capable of handling light transport in media. They are `VolPathTracer` (a volumetric version of `PathTracer`), `VolLightTracer` (volumetric light tracing, BRE [9] and photon beams [11]), `VolBidirPT` (volumetric BPT) and finally `UPBP`.

Obviously, we will further focus mainly on the `UPBP` renderer. It can handle absorption and scattering in any homogeneous[1] participating media. It is the most complete and consequently also the most complex renderer. Therefore we describe here only its very core and leave out many less important features of the actual implementation (mostly auxiliary and debugging).

In this section we describe mainly skeleton of the algorithm with evaluation of the BPT estimator. Evaluation of photon density estimators and computation of MIS weights are presented later in separate sections.

3.3.1 `UPBP` initialization

We begin with a few notes about the initialization of the `UPBP` renderer. As expected, most of it takes place in a constructor of the renderer and takes the configuration (the `Config` instance) as its input.

[1] As we mentioned in Section 3.2.2, the UPBP algorithm is not limited to the homogeneous case. An implementation of heterogeneous media is possible but we left it for the future work.

For example:

- a framebuffer of the renderer (`mFramebuffer`) is set up for the desired resolution
- structures for storing and evaluating contribution of photons and beams (`mPB2DEmbreeBre`, `mBB1DPhotonBeams`) are initialized (see Section 3.4.2 and 3.4.3 for more information about them)
- initial and reduction factor values for kernel radii are set (the fields have names in a form `m<est>RadiusInitial`, `m<est>RadiusAlpha`, where `<est>` stands for `Surf`, `PP3D`, `PB2D` and `BB1D`)
- a random number generator (`mRng`) is initialized
- photon and query beam types (`mPhotonBeamType`, `mQueryBeamType`) are set
- a number of carried out iterations (`mIterations`) is set to zero
- a reference to a scene to render (`mScene`) is set
- a maximum path length (`mMaxPathLegth`) is set.

The `UPBP` renderer (as well as some of the others) is configurable to render images using only a subset of all estimators it otherwise combines. For this purpose two more variables are set in the constructor based on its input:

`mAlgorithm` Determines the basic algorithm. Has (exactly) one of these values:

- `kLT` light tracing
- `kPTdir` the simplest path tracing, no light sampling, waits for a direct hit of a light source
- `kPTls` path tracing with explicit light sampling only, does not accumulate emission of directly hit light sources
- `kPTmis` path tracing with contributions of light sampling and directly hit light sources combined by MIS
- `kBPT` bidirectional path tracing (BPT)
- `kPPM` progressive (surface) photon mapping
- `kBPM` bidirectional (surface) photon mapping
- `kVCM` VCM (vertex connection in media, merging on surfaces only)
- `kCustom` custom subset of estimators from UPBP

`mEstimatorTechniques` Specifies which estimators from UPBP are used when `mAlgorithm == kCustom`. Contains any subset of these: `kBPT`, `kSURF`, `kPP3D`, `kPB2D` and `kBB1D`.

When `mAlgorithm != kCustom`, `mEstimatorTechniques` are set to contain `kBPT` if `mAlgorithm` $\in$ {`kBPT`, `kVCM`}, and to contain `kSURF` if `mAlgorithm` $\in$ {`kPPM`, `kBPM`, `kVCM`}.

According to these two variables a group of boolean flags are set in order to simplify switching conditions in code:

`mTraceLightPaths` whether or not to trace paths from lights

`mTraceCameraPaths` whether or not to trace paths from the camera

`mConnectToCamera` if paths from lights can be explicitly connected to the camera

`mConnectToLightSource` if paths from the camera can be explicitly connected to lights

`mConnectToLightVertices` if paths from the camera can be explicitly connected to vertices on paths from lights

`mMergeWithLightVerticesSurf` whether or not to apply the SURF estimator

`mMergeWithLightVerticesPP3D` whether or not to apply the P-P3D estimator

`mMergeWithLightVerticesPB2D` whether or not to apply the P-B2D estimator

`mMergeWithLightVerticesBB1D` whether or not to apply the B-B1D estimator

Relations among all these values are shown in Table 3.1.

3.3.1.1 Rendering modes

While the aforementioned values control what algorithms are used to find light transport paths there is also a way to limit what light transport paths are found. We incorporated this feature to be able to compare the UPBP algorithm with the previous work [13, 9, 10] since the volumetric photon density estimators are originally designed to capture only medium transport. There are two more values `mEstimatorTechniques` can contain - `PREVIOUS` and `COMPATIBLE` (and if any of them is present another flag, `mConnectToCameraFromSurf`, is set to false). These values enable the so called *previous* and *compatible* rendering modes. In both modes the renderer will simulate only a subset of light transport paths that can be described by the regular expression $L(S|D|M)^*MS^*C$, where L denotes a light source, C the camera, S a purely specular surface interaction, D a diffuse/glossy surface interaction and M a medium interaction. That means no non-specular surface interaction on a path from the camera before the first medium interaction. This is the only condition put on the compatible mode. The previous mode has one more – a path from the camera does not continue after the first medium interaction. So in the compatible mode the type of light transport paths is limited to that original volumetric photon density estimators can sample but the extended possibilities how to sample those paths are kept. In the previous mode all limitations of the original volumetric photon density estimators are simulated. Figure 3.5 shows an example.

	mAlgorithm			
	kLT	kPTdir	kPTls	kPTmis
mTraceLightPaths	1	0	0	0
mTraceCameraPaths	0	1	1	1
mConnectToCamera	1	0	0	0
mConnectToLightSource	0	0	1	1
mConnectToLightVertices	0	0	0	0
mMergeWithLightVerticesSurf	0	0	0	0
mMergeWithLightVerticesPP3D	0	0	0	0
mMergeWithLightVerticesPB2D	0	0	0	0
mMergeWithLightVerticesBB1D	0	0	0	0

	mAlgorithm			
	kBPT	kPPM	kBPM	kVCM
mTraceLightPaths	1	1	1	1
mTraceCameraPaths	1	1	1	1
mConnectToCamera	1	0	0	1
mConnectToLightSource	1	0	0	1
mConnectToLightVertices	1	0	0	1
mMergeWithLightVerticesSurf	0	1	1	1
mMergeWithLightVerticesPP3D	0	0	0	0
mMergeWithLightVerticesPB2D	0	0	0	0
mMergeWithLightVerticesBB1D	0	0	0	0

	mAlgorithm
	kCustom
mTraceLightPaths	1 if mEstimatorTechniques $\neq \emptyset$
mTraceCameraPaths	1 if mEstimatorTechniques $\neq \emptyset$
mConnectToCamera	1 if kBPT $\in$ mEstimatorTechniques
mConnectToLightSource	1 if kBPT $\in$ mEstimatorTechniques
mConnectToLightVertices	1 if kBPT $\in$ mEstimatorTechniques
mMergeWithLightVerticesSurf	1 if kSURF $\in$ mEstimatorTechniques
mMergeWithLightVerticesPP3D	1 if kPP3D $\in$ mEstimatorTechniques
mMergeWithLightVerticesPB2D	1 if kPB2D $\in$ mEstimatorTechniques
mMergeWithLightVerticesBB1D	1 if kBB1D $\in$ mEstimatorTechniques

Table 3.1: Relations among values that control functionality of the UPBP renderer.

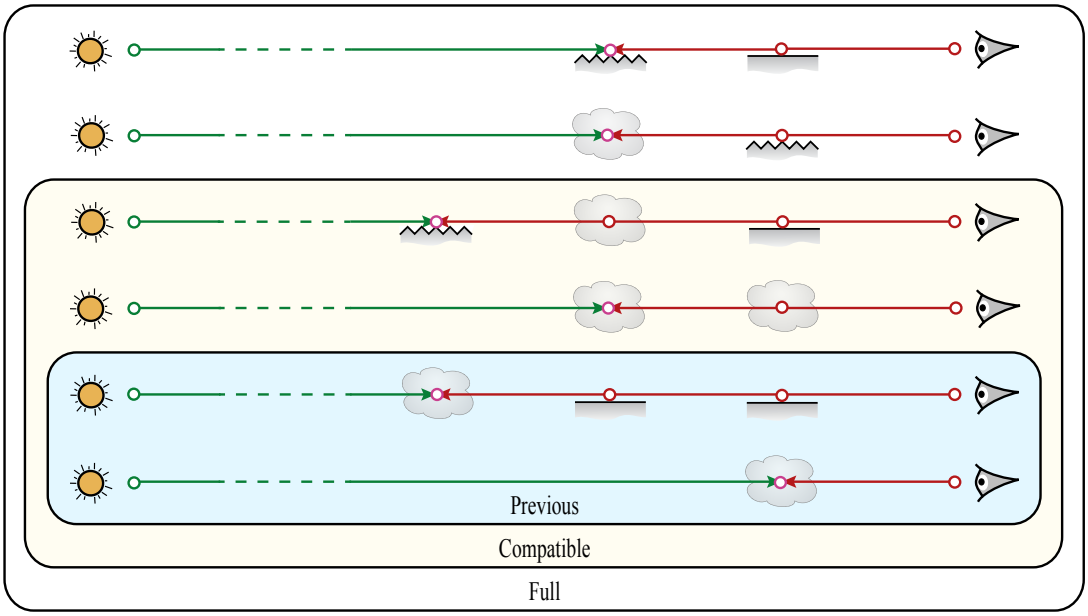


Figure 3.5: An example of different light transport paths simulated in "previous" and "compatible" rendering modes and while rendering full light transport. A cloud represents a medium, a straight line a purely specular surface and a zigzag line diffuse/glossy surface. Only the camera subpath (red, right) is important, the light subpath (green, left) can be arbitrary. The bottom two paths illustrate that "previous" mode allows only camera subpaths ending at the first interaction with a medium possibly preceded by an arbitrary number of interactions with purely specular surfaces. "Compatible" mode contains all paths from "previous" mode but additionally allows camera subpaths to continue after the first medium interaction. While only purely specular surfaces are allowed before the first medium interaction (same as in "previous" mode), camera subpaths can continue without restrictions after it, e.g. with interactions with another medium or a diffuse surface as shown by the middle two paths. While rendering full light transport no restrictions are placed at all. Besides paths from "compatible" mode all other paths are included, the top two paths serve as examples.

3.3.2 `UPBP` render iteration

After initialization we can proceed to the rendering algorithm itself - the `RunIteration` method. It runs in two stages in order to maximize path reuse and consequent brute-force variance reduction of photon density estimators.

In the first stage, we trace a number of light subpaths, connect their vertices to the camera (corresponds to light tracing), and then store the vertices (in `mLightVertices`) and parts of segments located in media (as photon beams in `mPhotonBeamsArray`). We build separate hashed grids over the surface and medium vertices (`mSurfHashGrid`, `mPP3DHashGrid`), which are later used for the SURF and P-P3D estimators. We also build an additional bounding volume hierarchy (BVH) over the medium vertices (`mPB2DEmbreeBre`) for the P-B2D estimator. Photon beams are organized in a uniform grid (`mBB1DPhotonBeams`). See Sections 3.4.1, 3.4.2 and 3.4.3 for more information about these structures.

In the second stage, we trace one camera subpath per pixel and construct a number of estimates as follows. Each vertex, surface or medium, is connected to a

light source and to the vertices of a light subpath in order to evaluate the different unbiased estimators from BPT. Furthermore, at each camera subpath vertex we evaluate the SURF (on surface) or the P-P3D (in medium) estimator by looking up the photons from the corresponding grid. For each camera subpath segment passing through a medium, we evaluate the P-B2D and B-B1D estimators.

This way the photon density estimators evaluated in the second stage can benefit from reuse of all light subpaths traced in the first stage and amortize effort put in their sampling.

Since a simple average of independent iterations would not be consistent as it would contain bias in the form of blur inherited from the photon density estimators, progressive reduction of kernel radii is implemented. At the beginning of each iteration radii of the photon density estimator kernels are reduced according to a scheme

$$r_i = r_1 \sqrt[d]{i^{\alpha-1}}, \tag{3.7}$$

where r_1 is an initial radius (`m<est>RadiusInitial`, where `<est>` stands for `Surf`, `PP3D`, `PB2D` and `BB1D`), i an iteration number (`aIteration`+1), α a reduction factor (`m<est>RadiusAlpha`) and d a dimension of the kernel. This scheme is proved by Georgiev et al. [4] to ensure consistency of the SURF estimator, consistency of the other estimators is the subject of our future work. Note that the results in Chapter 4 and in the UPBP paper [14] *do not use any radius reduction*. The only exception are the reference images where full light transport is simulated including evaluation of the SURF estimator. There the radius for the SURF estimator is reduced according to Equation 3.7 with $\alpha = 0.75$ (as recommended in [4]).

That was an outline of the `RunIteration` method, here is its more detailed pseudocode:

Listing 3.12: `RunIteration` (method, `UPBP.hxx`)

```
virtual void RunIteration(int aIteration)
{
    // Init ////////////////////////////////////////////////////////////////

    const int resX = int(mScene.mCamera.mResolution.get(0));
    const int resY = int(mScene.mCamera.mResolution.get(1));

    // Get path count, one path for each pixel.
    mPathCount = resX * resY;

    // Compute reduced kernel radii.
    float radiusSurf, radiusPP3D, radiusPB2D, radiusBB1D;
    ComputeRadii*(radiusSurf, radiusPP3D, radiusPB2D, radiusBB1D);

    // Compute weight and normalization factors for photon density estimators.
    ...ComputeFactors...

    // mPathEnds is an array of indices to mLightVertices. For each light
    // subpath stores where it ends. Here it is cleared, nothing ends anywhere.
    mPathEnds.resize(mPathCount);
    memset(&mPathEnds[0], 0, mPathEnds.size() * sizeof(int));

    // Remove all light vertices and reserve space for some.
    mLightVertices.clear();
    mLightVertices.reserve(LightVertsUpperBound*());

    // Remove all photon beams and reserve space for some.
    mPhotonBeamsArray.clear();
    mPhotonBeamsArray.reserve(BeamsUpperBound*());

    ..RunIterationPart2..
}
```

Listing 3.13: `RunIterationPart2` (part of the `RunIteration` method, `UPBP.hxx`)

```
////////////////////////////////////////////////////////////////////////////
// Stage 1: Generate light paths
////////////////////////////////////////////////////////////////////////////

// If pure path tracing is used, there are no lights or only one path
// segment is allowed, light tracing step is skipped.
if (mTraceLightPaths && mScene.GetLightCount() > 0 && mMaxPathLength > 1)
{
    ... TraceLightPaths ...
}

////////////////////////////////////////////////////////////////////////////
// Build accerelation structures
////////////////////////////////////////////////////////////////////////////

if (!mLightVertices.empty())
{
    // For SURF.
    if (mMergeWithLightVerticesSurf)
    {
        mSurfHashGrid.Reserve(mPathCount);
        mSurfHashGrid.Build(mLightVertices, radiusSurf, SURF);
    }

    // For PP3D.
    if (mMergeWithLightVerticesPP3D)
    {
        mPP3DHashGrid.Reserve(mPathCount);
        mPP3DHashGrid.Build(mLightVertices, radiusPP3D, PP3D);
    }

    // For PB2D.
    if (mMergeWithLightVerticesPB2D)
    {
        mPB2DEmbreeBre.build(&mLightVertices[0],
            (int)mLightVertices.size(), radiusPB2D);
    }

    // For BB1D.
    if (mMergeWithLightVerticesBB1D && !mPhotonBeamsArray.empty())
    {
        mBB1DPhotonBeams.build(mPhotonBeamsArray, radiusBB1D);
    }
}

////////////////////////////////////////////////////////////////////////////
// Stage 2: Generate camera paths
////////////////////////////////////////////////////////////////////////////

// Unless rendering with traditional light tracing.
if (mTraceCameraPaths)
{
    ... TraceCameraPaths ...
}

////////////////////////////////////////////////////////////////////////////
// Final steps
////////////////////////////////////////////////////////////////////////////

// Increment the number of iterations made.
mIterations++;

// Delete stored photons.
if (mMergeWithLightVerticesPB2D && !mLightVertices.empty())
    mPB2DEmbreeBre.destroy();

// Delete stored photon beams.
if (mMergeWithLightVerticesBB1D && !mPhotonBeamsArray.empty())
    mBB1DPhotonBeams.destroy();

// (The other two structures do not have to be explicitly destroyed.)
```

3.3.2.1 Tracing light subpaths

Let's take a closer look at tracing light subpaths. Tracing a single light subpath begins with sampling a position on a light source and a direction from it (an origin and direction of the first ray on the subpath). Then in a loop a ray is cast and its intersection with a surface or a scattering point in a medium is found together with segments on the ray passing through media. These segments are used for storing beams and attenuating the throughput of the subpath. A new vertex is created, stored and taken as a new ray origin. An attempt to connect it to the camera is made and finally a new ray direction is sampled, closing the loop. The number of traced light subpaths n_{paths} (`mPathCount`) is equal to the number of pixels. Detailed pseudocode follows:

Listing 3.14: `TraceLightPaths` (part of the `RunIteration` method, `UPBP.hxx`)

```
for (int pathIdx = 0; pathIdx < mPathCount; pathIdx++)
{
   // Generate an origin and a direction of the first segment
   // of a light subpath.
   SubPathState lightState;
   GenerateLightSample(pathIdx, lightState);

   // In attenuating media the ray can never travel from infinity.
   if (!lightState.mIsFiniteLight
       && mScene.GetGlobalMediumPtr()->HasAttenuation())
   {
      mPathEnds[pathIdx] = (int)mLightVertices.size();
      continue;
   }

   // We assume that the light is (on) a surface.
   bool originInMedium = false;

   // Trace the subpath.
   for (;; ++lightState.mPathLength)
   {
      // Prepare a ray.
      Ray ray(lightState.mOrigin, lightState.mDirection);

      // Cast the ray.
      Isect isect;
      mVolumeSegments.clear();
      mLiteVolumeSegments.clear();
      bool intersected = mScene.Intersect(
         ray, originInMedium ? AbstractMedium::kOriginInMedium : 0,
         mRng, isect, lightState.mBoundaryStack,
         mVolumeSegments, mLiteVolumeSegments);

      // Store beams if required.
      if (mMergeWithLightVerticesBB1D)
         AddBeams(
            ray, lightState.mThroughput, &mLightVertices.back(),
            originInMedium ? AbstractMedium::kOriginInMedium : 0,
            lightState.mLastPdfWInv);

      // Stop tracing if the ray left the scene.
      if (!intersected) break;

      ..TraceLightPath..
   }

   // Remember the end of this path.
   mPathEnds[pathIdx] = (int)mLightVertices.size();
}
```

Listing 3.15: `TraceLightPath` (part of the `RunIteration` method, `UPBP.hxx`)

```
// Attenuate by intersected media (if any).
.. AttenuateLightSubpath ...

// Stop tracing if the path throughput is no longer positive.
if (lightState.mThroughput.isBlackOrNegative()) break;

// Prepare scattering function at the hitpoint (BSDF/phase depending
// on whether the hitpoint is at surface or in media, the isect knows).
BSDF bsdf(
    ray, isect, mScene, BSDF::kFromLight,
    mScene.RelativeIOR(isect, lightState.mBoundaryStack));

// Terminate if the scattering function is invalid (e.g. when hitting
// surface too parallel with tangent plane.
if (!bsdf.IsValid())
    break;

// Compute the hitpoint.
const Pos hitPoint = ray.origin + ray.direction * isect.mDist;

// Current vertex will be the next origin.
originInMedium = isect.IsInMedium();

// Create and store a new vertex.
... StoreVertex ...

// Connect to the camera, unless omitted in the algorithm,
// the scattering function is purely specular or we are not
// allowed to connect from a surface (we are in the previous
// or compatible mode).
if (mConnectToCamera && !bsdf.IsDelta()
   && (bsdf.IsInMedium() || mConnectToCameraFromSurf))
    ConnectToCamera(
    pathIdx, lightState, hitPoint, bsdf,
    mLightVertices.back().mMisData.mRaySamplePdfsRatio);

// Terminate if the path would become too long after scattering.
if (lightState.mPathLength + 2 > mMaxPathLength)
    break;

// Continue random walk.
if (!SampleScattering(
    bsdf, hitPoint, isect, lightState, mLightVertices.back().mMisData,
    mLightVertices.at(mLightVertices.size() - 2).mMisData))
    break;
```

The current state of the light subpath is kept in the `SubPathState` structure:

Listing 3.16: `SubPathState` (struct, `UPBP.hxx`)

```
struct SubPathState
{
    Pos    mOrigin;         // Origin of the next path segment.
    Dir    mDirection;      // Direction of the next path segment.
    Rgb    mThroughput;     // Path contribution.
    uint   mPathLength;     // Number of path segments, including the next one.
    uint   mIsFiniteLight; // Whether the path was just generated
                            // by a finite light.
    uint   mSpecularPath;  // Whether all scattering events so far were specular.
    bool   mLastSpecular;  // Whether the last sampled event was specular.
    float  mLastPdfWInv;   // Inverse of pdf of the last sampled direction w.r.t.
                            // to the solid angle measure at the last vertex.

    BoundaryStack mBoundaryStack; // Stack of crossed boundaries.
}
```

Note that the `mThroughput` variable, despite its name, actually stores the whole subpath contribution (Equations 1.6, 1.7), not the throughput T alone. The

`mBoundaryStack` field stores the boundary stack described in Section 3.2.3 needed to correctly track the current medium along the subpath.

Naming convention. On the occasion of the first encounter of a pdf variable (`mLastPdfWInv` in Listing 3.16) we would like to say a few words about our naming convention. Names of pdf variables are typically in the form:

`[m|a|o]<name>[Dir|Rev]Pdf[A|W][Inv]`

Name of the pdf `<name>` can be prefixed with letters `m`, `a` or `o` which, as described previously, denote a class (or struct) property or an input and output method argument, respectively. After the name an identifier of the "direction of the pdf" can come. It is the direction in which the related sampling is performed. No identifier or `Dir` mean a direction congruent with the direction of currently sampled subpath (i.e. from a light or from the camera), `Rev` means the reverse direction. Then there is the word `Pdf` possibly followed by letters `A` or `W` denoting the pdf is computed w.r.t. the surface area measure ("A" as an area) or the solid angle measure ("W" as an ASCII substitution of ω). Finally, the name can end with `Inv` if the variable actually stores an inverse value of the pdf.

Initialization. Initialization of the `SubPathState` structure is carried out in the `GenerateLightSample` method after sampling lights and creating the first vertex of the subpath:

Listing 3.17: `GenerateLightSample` (method, `UPBP.hxx`)

```
1  void GenerateLightSample(int aPathIdx, SubPathState &oLightState)
2  {
3     // Choose a light uniformly.
4     const int           lightCount    = mScene.GetLightCount();
5     const float         lightPickProb = 1.f / lightCount;
6     const int           lightID       = int(mRng.GetFloat() * lightCount);
7     const AbstractLight *light        = mScene.GetLightPtr(lightID);
8
9     // The chosen light may need these random numbers for sampling
10    // a position on it and a direction from it.
11    const Vec2f rndDirSamples = mRng.GetVec2f();
12    const Vec2f rndPosSamples = mRng.GetVec2f();
13
14    // Sample emission of the chosen light.
15    float emissionPdfW, directPdfA, cosLight;
16    oLightState.mThroughput = light->Emit(
17       mScene.mSceneSphere, rndDirSamples, rndPosSamples, oLightState.mOrigin,
18       oLightState.mDirection, emissionPdfW, &directPdfA, &cosLight);
19
20    // Complete the probabilities.
21    emissionPdfW *= lightPickProb;
22    directPdfA *= lightPickProb;
23
24    // Create and store the first vertex on the subpath.
25    ...StoreFirstVertex...
26
27    // Complete light subpath state initialization.
28    oLightState.mThroughput /= emissionPdfW;
29    oLightState.mPathLength = 1;
30    oLightState.mIsFiniteLight = light->IsFinite() ? 1 : 0;
31    oLightState.mLastSpecular = false;
32    oLightState.mLastPdfWInv = directPdfA / emissionPdfW;
33
34    // Init the boundary stack with the global medium and a medium
35    // of the light (if defined).
36    InitBoundaryStackForLight*(oLightState.mBoundaryStack);
37 }
```

The `light->Emit` method samples emission of the chosen light, i.e. its surface (if it has any) and outgoing direction. It returns the resulting point and direction along with the throughput (emission in the direction) and probabilities of such sampling (`emissionPdfW`) and of getting the same point and direction via explicit light sampling (direct illumination sampling) when tracing a subpath from the camera (`directPdfA`). For example for an area light source `emissionPdfW` is probability of sampling a point on it and a direction from it, while `directPdfA` is only probability of sampling the point.

The `GenerateLightSample` method also creates and stores the first vertex of the light subpath. Vertices of light subpaths are of type `UPBPLightVertex`:

Listing 3.18: UPBPLightVertex (struct, UPBPLightVertex.hxx)

```
struct UPBPLightVertex
{
   Pos  mHitpoint;    // Position of the vertex.
   Rgb  mThroughput; // Path throughput.
   int  mPathIdx;     // Path index.
   uint mPathLength; // Number of segments between a source and the vertex.
   bool mInMedium;    // Whether the vertex is in a participating medium (not on
                      // a surface).
   bool mConnectable;// Whether the vertex can be used for explicit vertex
                      // connection in BPT.
   bool mIsFinite;    // Whether the vertex is not on an infinite light source.
   BSDF mBSDF;        // Stores all required local information, including
                      // the incoming direction.
   MisData mMisData; // Data needed for MIS weights computation.
}
```

Again, the `mThroughput` variable stores the whole light subpath contribution $C_1(\mathbf{x}_0 \dots \mathbf{x}_j)$ (as proved at the end of this part as Theorem 1). The `mConnectable` field is necessary since we store all light vertices to correctly compute MIS weighs but not all of them can be used in BPT for explicit vertex connection. There is no point in connecting to vertices on ideally specular surfaces as the probability of connection in the only acceptable direction is zero. Similar to Georgiev et al. [4] we also do not connect to vertices on light sources in order to reduce correlation, we rather connect to a randomly sampled new point on a light source. Furthermore, each light vertex stores the whole `BSDF` object representing the scattering function at the vertex (see Section 3.2.4). The `mMisData` field is described in Section 3.5.

The `SubPathState` structure is similar to the `UPBPLightVertex` structure. However, while the former one describes a current state of a traced subpath (both light and camera) and there is always only one at a time, the latter one describes a light vertex and is stored with each of them.

Here is the pseudocode creating and storing the first light vertex:

Listing 3.19: `StoreFirstVertex` (part of the `GenerateLightSample` method, `UPBP.hxx`)

```
UPBPLightVertex lightVertex;                    // Create
lightVertex.mHitpoint = oLightState.mOrigin;
lightVertex.mThroughput = Rgb(1.0f);
lightVertex.mPathIdx = aPathIdx;
lightVertex.mPathLength = 0;
lightVertex.mInMedium = false;
lightVertex.mConnectable = false;
lightVertex.mIsFinite = light->IsFinite();
...SetFirstLightVertexMisData...                // Set up lightVertex.mMisData
mLightVertices.push_back(lightVertex);          // Store
```

Section 3.5 describes how `mMisData` of the first light vertex are set.

The last step performed in the `GenerateLightSample` method is initializing the boundary stack, i.e. the stack is cleared and triplets of the global medium and the light source are put in it (see Section 3.2.3).

Tracing. We now have the first light vertex and a direction of the first segment and can start tracing the subpath. The first ray is shot in the scene and traced. This is done by the `mScene.Intersect` method. As described in Section 3.2.3.1 the method tries to find an "intersection" of the ray with the scene (a medium scattering point or a real intersection) and also returns volume segments along the ray.

What are the found volume segments used for? Firstly, each segment is converted to a beam and stored in `mPhotonBeamsArray`. This is implemented in the `AddBeams` method. It cycles through the segments (`VolumeSegment` if using short photon beams, `LiteVolumeSegment` otherwise), for each segment creates a corresponding beam and adds it to `mPhotonBeamsArray`. Beams are described in Section 3.4.3. Secondly, throughput of the subpath is attenuated according to the segments:

Listing 3.20: `AttenuateLightSubpath` (part of the `RunIteration` method, `UPBP.hxx`)

```
float raySamplePdf(1.0f);
float raySampleRevPdf(1.0f);
if (!mVolumeSegments.empty())
{
    // Pdf of sampling through the segments.
    raySamplePdf = VolumeSegment::AccumulatePdf(mVolumeSegments);

    // Pdf of sampling through the segments in the reverse direction.
    // Needed for setting MIS data of a stored vertex only.
    raySampleRevPdf = VolumeSegment::AccumulateRevPdf(mVolumeSegments);

    // Attenuation by the segments.
    lightState.mThroughput *=
        VolumeSegment::AccumulateAttenuationWithoutPdf(mVolumeSegments)
        / raySamplePdf;
}
```

The static methods `VolumeSegment::Accumulate...` simply multiply appropriate values of the individual segments together. Attenuation of the throughput of the subpath is then computed according to Equation 1.6.

New vertex. At this point we have everything we need to create and store a new vertex of the subpath:

Listing 3.21: `StoreVertex` (part of the `RunIteration` method, `UPBP.hxx`)

```
UPBPLightVertex lightVertex;                        // Create
lightVertex.mHitpoint = hitPoint;
lightVertex.mThroughput = lightState.mThroughput;
lightVertex.mPathIdx = pathIdx;
lightVertex.mPathLength = lightState.mPathLength;
lightVertex.mInMedium = originInMedium;
lightVertex.mConnectable = !bsdf.IsDelta();
lightVertex.mIsFinite = true;
lightVertex.mBSDF = bsdf;
...SetLightVertexMisData...                         // Set up lightVertex.mMisData
mLightVertices.push_back(lightVertex);              // Store
```

Section 3.5 describes how `mMisData` of the new light vertex are set.

Connection to the camera. Before we continue tracing the subpath we try to explicitly connect the subpath at the new vertex to the camera. It comprises evaluation of the scattering function at the vertex for the incoming direction and the direction to the camera, casting a ray to make sure the connection is not occluded by geometry, attenuating by media the ray intersects and proper MIS weighting. It is implemented in the `ConnectToCamera` method:

Listing 3.22: `ConnectToCamera` (method, `UPBP.hxx`)

```
void ConnectToCamera(
    const int           aLightPathIdx,
    const SubPathState &aLightState,
    const Pos          &aHitpoint,
    const BSDF         &aLightBSDF,
    const float         aRaySampleRevPdfsRatio)
{
    // Get the camera and a direction to it.
    const Camera &camera = mScene.mCamera;
    Dir directionToCamera = camera.mOrigin - aHitpoint;

    // Check whether the vertex is in front of camera.
    if (dot(camera.mDirection, -directionToCamera) <= 0.f) return;

    // Check whether it projects to the screen (and where).
    const Vec2f imagePos = camera.WorldToRaster(aHitpoint);
    if (!camera.CheckRaster(imagePos)) return;

    // Compute distance and normalize the direction to the camera.
    const float distEye2 = directionToCamera.square();
    const float distance = std::sqrt(distEye2);
    directionToCamera /= distance;

    // Evaluate the scattering function at the vertex.
    float cosToCamera, bsdfDirPdfW, bsdfRevPdfW, sinTheta;
    Rgb bsdfFactor = aLightBSDF.Evaluate(
        directionToCamera, cosToCamera, &bsdfDirPdfW, &bsdfRevPdfW, &sinTheta);

    if (bsdfFactor.isBlackOrNegative()) return;

    // There is a Russian roulette decision at each vertex whether the subpath
    // will continue. Its pdf has to be included. bsdfDirPdfW is not updated
    // since it is not used.
    bsdfRevPdfW *= aLightBSDF.ContinuationProb();

    // Compute a pdf conversion factor from image plane area to surface area
    const float cosAtCamera = dot(camera.mDirection, -directionToCamera);
    const float imagePointToCameraDist = camera.mImagePlaneDist / cosAtCamera;
    const float imageToSolidAngleFactor = Utils::sqr(imagePointToCameraDist)
        / cosAtCamera;
    const float imageToSurfaceFactor = imageToSolidAngleFactor
        * std::abs(cosToCamera) / Utils::sqr(distance);

    // We put the virtual image plane at such a distance from the camera origin
    // that the pixel area is one and thus the image plane sampling pdf is 1. The
    // area pdf of aHitpoint as sampled from the camera is then equal to the
    // conversion factor from image plane area density to surface area density.
    const float cameraPdfA = imageToSurfaceFactor;
    const float surfaceToImageFactor = 1.f / imageToSurfaceFactor;

    // Test whether the connection is not occluded by geometry and find volume
    // segments along it.
    mVolumeSegments.clear();
    if (!mScene.Occluded(
        aHitpoint, directionToCamera, distance, aLightState.mBoundaryStack,
        aLightBSDF.IsInMedium() ? AbstractMedium::kOriginInMedium : 0,
        mVolumeSegments))
    {
        ..ConnectToCamContrib..
    }
}
```

After a direction to the camera is computed, the scattering function is evaluated in the `aLightBSDF.Evaluate` method (see Section 3.2.4). Note that the returned pdfs of sampling the scattering function in the direct (i.e. from the light) and reverse direction (`bsdfDirPdfW` and `bsdfRevPdfW`, respectively) are needed only for MIS weights computation (we are making an explicit connection, there is no sampling). The former one is actually not needed at all since there is no chance of hitting our ideal pinhole camera by accident.

Before the resulting contribution is computed the `mScene.Occluded` method casts a ray from the vertex to the camera in order to test that there are no obstacles, i.e. the ray does not intersect any geometry before reaching the camera. The method is more closely described in Section 3.2.3.1. If the connection is not occluded, the contribution is computed:

Listing 3.23: `ConnectToCamContrib` (part of the `ConnectToCamera` method, `UPBP.hxx`)

```
// Get attenuation from intersected media (if any).
float raySampleRevPdf(1.0f);
Rgb attenuation(1.0f);
if (!mVolumeSegments.empty())
{
    // Pdf of sampling through the segments in the reverse direction.
    // Needed for MIS weighting only, the direct one is not needed
    // at all (we cannot hit the camera).
    raySampleRevPdf = VolumeSegment::AccumulateRevPdf(mVolumeSegments);

    // Attenuation by the segments (without pdf since we made
    // an explicit connection and did not sample media).
    attenuation =
        VolumeSegment::AccumulateAttenuationWithoutPdf(mVolumeSegments);

    if (!attenuation.isPositive())
        return;
}

// Compute MIS weight (if not doing simple light tracing).
float misWeight = 1.f;
if (mAlgorithm != kLT)
{
    ...ConnectToCameraMis...
}

// We divide the contribution by surfaceToImageFactor to convert
// the (already divided) pdf from surface area to image plane area,
// w.r.t. which the pixel integral is actually defined. We also
// divide by the number of samples this technique makes, which is
// equal to the number of light sub-paths.
Rgb contrib = aLightState.mThroughput * bsdfFactor * attenuation
    / (mPathCount * surfaceToImageFactor);

// Weight the contribution.
contrib *= misWeight;

// Update the framebuffer.
mFramebuffer.AddColor(imagePos, contrib);
```

First, the attenuation caused by the volume segments found along the ray is accumulated. Then the MIS weight is computed (as described in detail in Section 3.5). Finally, the unweighted contribution is computed according to Equation 2.23 (with $\mathbf{c} = \mathbf{x}_k$).

We have

$$\begin{aligned}
\texttt{aLightState.mThroughput} &= C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_{k-1}),\\
\texttt{bsdfFactor} &= \rho(\mathbf{x}_{k-1}),\\
\texttt{attenuation} &= T_{\mathrm{r}}(\mathbf{x}_{k-1}, \mathbf{x}_k),\\
\texttt{misWeight} &= \hat{w}_{BPT_{\mathbf{x}_{k-1},\mathbf{x}_k}},\\
p(\mathbf{x}_k) &= 1,
\end{aligned}$$

and use

$$\begin{aligned}
W_{\mathrm{e}}(\mathbf{x}_k) &= p(\omega_{\mathbf{x}_k})\\
\hat{p}(\omega_{\mathbf{x}_k}) &= \frac{\texttt{imageToSolidAngleFactor}}{\texttt{mPathCount}}.
\end{aligned}$$

Then

$$\begin{aligned}
\frac{\texttt{surfaceToImageFactor}^{-1}}{\texttt{mPathCount}} &= \frac{\texttt{imageToSurfaceFactor}}{\texttt{mPathCount}}\\
&= \frac{\texttt{imageToSolidAngleFactor}}{\texttt{mPathCount}} \frac{D(\mathbf{x}_{k-1} \to \mathbf{x}_k)}{\|\mathbf{x}_k - \mathbf{x}_{k-1}\|^2}\\
&= \hat{p}(\omega_{\mathbf{x}_k}) \frac{D(\mathbf{x}_{k-1} \to \mathbf{x}_k)}{\|\mathbf{x}_k - \mathbf{x}_{k-1}\|^2}\\
&= W_{\mathrm{e}}(\mathbf{x}_k) G(\mathbf{x}_{k-1}, \mathbf{x}_k),
\end{aligned}$$

and the final contribution of the connection is:

$$\begin{aligned}
\texttt{contrib}=&\texttt{misWeight} * \frac{\texttt{aLightState.mThroughput*bsdfFactor*attenuation}}{\texttt{mPathCount*surfaceToImageFactor}}\\
=&\hat{w}_{BPT_{\mathbf{x}_{k-1},\mathbf{x}_k}} C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_{k-1}) \frac{T_{\mathrm{r}}(\mathbf{x}_{k-1}, \mathbf{x}_k) G(\mathbf{x}_{k-1}, \mathbf{x}_k) \rho(\mathbf{x}_{k-1})}{p(\mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k)\\
=&\hat{w}_{BPT_{\mathbf{x}_{k-1},\mathbf{x}_k}} L_{\mathrm{e}}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{x}_k)}{p(\mathbf{x}_0 \dots \mathbf{x}_{k-1}) p(\mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k).
\end{aligned}$$

Sampling scattering. Everything for the light subpath of the current length is done, now we can try adding another segment. We make a Russian roulette decision whether to continue sampling the subpath or not, sample the scattering function for a direction of the new segment and update the state of the subpath. The `SampleScattering` method is responsible for this.

Listing 3.24: `SampleScattering` (method, `UPBP.hxx`)

```
bool SampleScattering(
   const BSDF     &aBSDF,
   const Pos      &aHitPoint,
   const Isect    &aIsect,
   SubPathState   &aoState,
   MisData        &aoCurrentMisData,
   MisData        &aoPreviousMisData)
{
   // Make a Russian roulette decision.
   const float contProb = aBSDF.ContinuationProb();
   if (contProb == 0 || (contProb < 1.0f && mRng.GetFloat() > contProb))
      return false;

   // Sample the scattering function. If the vertex is located
   // in a medium, the returned cosThetaOut is 1.
   Dir   rndTriplet = mRng.GetVec3f();
   float bsdfDirPdfW, cosThetaOut, sinTheta;
   uint  sampledEvent;
   Rgb bsdfFactor = aBSDF.Sample(rndTriplet, aoState.mDirection,
      bsdfDirPdfW, cosThetaOut, &sampledEvent, &sinTheta);

   if (bsdfFactor.isBlackOrNegative())
      return false;

   bool specular = (sampledEvent & BSDF::kSpecular) != 0;

   // If we sampled a specular event, then the reverse probability
   // cannot be evaluated, but we know it is exactly the same as
   // the forward probability, so just set it. If non-specular event
   // happened, we evaluate the pdf. We need for MIS weighting.
   float bsdfRevPdfW = bsdfDirPdfW;
   if (!specular)
      bsdfRevPdfW = aBSDF.Pdf(aoState.mDirection, BSDF::kReverse);

   // Do not forget the Russian roulette probability.
   bsdfDirPdfW *= contProb;
   bsdfRevPdfW *= contProb;

   const float bsdfDirPdfWInv = 1.0f / bsdfDirPdfW;

   // Update subpath state. aoState.mDirection set in aBSDF.Sample.
   aoState.mOrigin = aHitPoint;
   aoState.mThroughput *= bsdfFactor * cosThetaOut * bsdfDirPdfWInv;
   aoState.mSpecularPath &= specular ? 1 : 0;
   aoState.mLastPdfWInv = bsdfDirPdfWInv;
   aoState.mLastSpecular = specular;

   // Switch medium on refraction.
   if ((sampledEvent & BSDF::kRefract) != 0)
      mScene.UpdateBoundaryStackOnRefract(aIsect, aoState.mBoundaryStack);

   // Update affected MIS data.
   ...SampleScatteringMis...

   return true;
}
```

The Russian roulette decision is made with probability dependent on material or medium properties. How this probability is computed before returning by the `aBSDF.ContinuationProb` method is described in Section 3.2.4. The same Section also offers details about the `aBSDF.Sample` method which implements sampling of the scattering function. It mainly returns a sampled direction (`aoState.mDirection`), a scattering function factor for it (`bsdfFactor`) and also a pdf of the sampling (in a direct direction, i.e. from the light – `bsdfDirPdfW`). When we have the new direction, the state of the subpath is updated as well as some of the MIS data (see Section 3.5).

Closing the loop. At this point the inner loop of tracing light subpaths starts a new run, a new ray is shot from the last vertex in the new direction and tracing continues. It stops when the subpath throughput drops to zero, the Russian roulette decision fails or the subpath reaches its maximum length. Then a new light subpath is traced. This way n_{paths} (`mPathCount`) light subpaths are traced.

To conclude our description of tracing light subpaths we would like to prove one important statement:

Theorem 1. *Assume any light subpath traced by the described algorithm and its i-th stored vertex (counting from zero). Then its* ***`mThroughput`*** *field satisfies:*

$$\texttt{mThroughput} = \begin{cases} \texttt{Rgb(1.0f)} & \text{if } i = 0, \\ C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_i) & \text{if } i > 0, \end{cases} \tag{3.8}$$

where $C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_i)$ is the subpath contribution (1.6).

Proof. Let $i = 0$. The first vertex of a subpath is created and stored in the `GenerateLightSample` method and its `mThroughput` is not modified anywhere else. Since the `mThroughput` field is assigned `Rgb(1.0f)` there (Listing 3.19, line 3) the theorem holds for $i = 0$.

Let $i > 0$. The subsequent vertices are created and stored in the inner loop of tracing light subpaths and their `mThroughput` field is assigned the value of `lightState.mThroughput` there (Listing 3.21, line 3). These values are not modified anywhere else.

$i = 1$ From the beginning of tracing of a new light subpath till reaching storing of the second vertex ($\mathbf{x}_1$) the `lightState.mThroughput` value is modified twice. Firstly in the `GenerateLightSample` method (Listing 3.17, lines 16 and 28) where it is initialized to

$$\texttt{mThroughput} = L_{\mathrm{e}}(\mathbf{x}_0)\frac{D(\mathbf{x}_0 \to \mathbf{x}_1)}{p(\mathbf{x}_0)\hat{p}(\omega_{\mathbf{x}_0})}.$$

$L_{\mathrm{e}}(\mathbf{x}_0)D(\mathbf{x}_0 \to \mathbf{x}_1)$ is emission of the sampled light source returned by the `light->Emit` method, $p(\mathbf{x}_0)$ is the probability density of sampling $\mathbf{x}_0$ (picking a light source and potentially sampling its surface) and $\hat{p}(\omega_{\mathbf{x}_0})$ is the probability density of sampling the direction from the light w.r.t. the (non-projected) solid angle measure. Note $p(\mathbf{x}_0)\hat{p}(\omega_{\mathbf{x}_0}) =$ `emissionPdfW`.

The second modification takes place after shooting the first ray when the throughput is attenuated by the intersected volume segments (Listing 3.20, line 13). The `lightState.mThroughput` value becomes

$$\texttt{mThroughput} = L_{\mathrm{e}}(\mathbf{x}_0)\frac{D(\mathbf{x}_0 \to \mathbf{x}_1)}{p(\mathbf{x}_0)\hat{p}(\omega_{\mathbf{x}_0})}\frac{T_{\mathrm{r}}(t_{\mathbf{x}_0})}{p(t_{\mathbf{x}_0})}.$$

$T_{\mathrm{r}}(t_{\mathbf{x}_0})$ is the accumulated attenuation, $p(t_{\mathbf{x}_0})$ is `raySamplePdf`, i.e. pdf of sampling the distance $t_{\mathbf{x}_0}$ through the segments w.r.t. the Euclidean length on $\mathbb{R}^1$. Since we know the vertex $\mathbf{x}_1$ was created and using definitions of the

throughput (1.2) and geometry term (1.3) we get

$$\begin{aligned}
\texttt{mThroughput} &= L_\mathrm{e}(\mathbf{x}_0)\frac{D(\mathbf{x}_0 \to \mathbf{x}_1)}{p(\mathbf{x}_0)\hat{p}(\omega_{\mathbf{x}_0})}\frac{T_\mathrm{r}(t_{\mathbf{x}_0})}{p(t_{\mathbf{x}_0})} \\
&= L_\mathrm{e}(\mathbf{x}_0)\frac{D(\mathbf{x}_0 \to \mathbf{x}_1)}{p(\mathbf{x}_0)\hat{p}(\omega_{\mathbf{x}_0})}\frac{T_\mathrm{r}(t_{\mathbf{x}_0})}{p(t_{\mathbf{x}_0})}\frac{D(\mathbf{x}_1 \to \mathbf{x}_0)\|\mathbf{x}_1 - \mathbf{x}_0\|^2}{D(\mathbf{x}_1 \to \mathbf{x}_0)\|\mathbf{x}_1 - \mathbf{x}_0\|^2} \\
&= L_\mathrm{e}(\mathbf{x}_0)\frac{T_\mathrm{r}(t_{\mathbf{x}_0})\frac{D(\mathbf{x}_0 \to \mathbf{x}_1)D(\mathbf{x}_1 \to \mathbf{x}_0)}{\|\mathbf{x}_1 - \mathbf{x}_0\|^2}}{p(\mathbf{x}_0)\left(\hat{p}(\omega_{\mathbf{x}_0})p(t_{\mathbf{x}_0})\frac{D(\mathbf{x}_1 \to \mathbf{x}_0)}{\|\mathbf{x}_1 - \mathbf{x}_0\|^2}\right)} \\
&= L_\mathrm{e}(\mathbf{x}_0)\frac{T_\mathrm{r}(t_{\mathbf{x}_0})G(\mathbf{x}_0, \mathbf{x}_1)}{p(\mathbf{x}_0)p(\mathbf{x}_1 \,|\, \mathbf{x}_0)} \\
&= L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \dots \mathbf{x}_1)}{p(\mathbf{x}_0 \dots \mathbf{x}_1)} = C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{x}_1).
\end{aligned}$$

We also used

$$\hat{p}(\omega_{\mathbf{x}_0})p(t_{\mathbf{x}_0})\frac{D(\mathbf{x}_1 \to \mathbf{x}_0)}{\|\mathbf{x}_1 - \mathbf{x}_0\|^2} = p(\mathbf{x}_1 \,|\, \mathbf{x}_0).$$

It is a conversion from the solid angle × length product measure to the volume measure. $\hat{p}(\omega_{\mathbf{x}_0})$ is w.r.t. the solid angle measure, not projected solid angle measure, so it is not converted by the whole geometric factor.

The theorem holds for $i = 1$.

$i > 1$ Assume the theorem holds for $j - 1, j > 1$, i.e. `mThroughput` of the vertex $\mathbf{x}_{j-1}$ equals $C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{x}_{j-1})$. Between storing vertices $\mathbf{x}_{j-1}$ and $\mathbf{x}_j$ the value of `lightState.mThroughput` is again modified twice. First while sampling scattering at $\mathbf{x}_{j-1}$ (Listing 3.24, line 43):

$$\texttt{mThroughput} = C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{x}_{j-1})\frac{\rho(\mathbf{x}_{j-1})D(\mathbf{x}_{j-1} \to \mathbf{x}_j)}{p_{RR}(\mathbf{x}_{j-1})\hat{p}(\omega_{\mathbf{x}_{j-1}})}.$$

$\rho(\mathbf{x}_{j-1})$ is `bsdfFactor` returned by the `aBSDF.Sample` method, $D(\mathbf{x}_{j-1} \to \mathbf{x}_j) =$ `cosThetaOut`, $p_{RR}(\mathbf{x}_{j-1})$ is the probability of the Russian roulette decision and $\hat{p}(\omega_{\mathbf{x}_{j-1}})$ the probability density of sampling the scattering function w.r.t. to the (non-projected) solid angle measure. Note $p_{RR}(\mathbf{x}_{j-1})\hat{p}(\omega_{\mathbf{x}_{j-1}}) =$ `bsdfDirPdfW`.

The second modification is the already described attenuation by intersected volume segments. Repeating the same steps we get

$$\begin{aligned}
\texttt{mThroughput} &= C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{x}_{j-1})\frac{\rho(\mathbf{x}_{j-1})D(\mathbf{x}_{j-1} \to \mathbf{x}_j)}{p_{RR}(\mathbf{x}_{j-1})\hat{p}(\omega_{\mathbf{x}_{j-1}})}\frac{T_\mathrm{r}(t_{\mathbf{x}_{j-1}})}{p(t_{\mathbf{x}_{j-1}})} \\
&= L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \dots \mathbf{x}_{j-1})}{p(\mathbf{x}_0 \dots \mathbf{x}_{j-1})}\frac{\rho(\mathbf{x}_{j-1})G(\mathbf{x}_{j-1}, \mathbf{x}_j)T_\mathrm{r}(t_{\mathbf{x}_{j-1}})}{p_{RR}(\mathbf{x}_{j-1})p(\mathbf{x}_j \,|\, \mathbf{x}_{j-1})} \\
&= L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0 \dots \mathbf{x}_j)}{p(\mathbf{x}_0 \dots \mathbf{x}_j)} = C_\mathrm{l}(\mathbf{x}_0 \dots \mathbf{x}_j).
\end{aligned}$$

The theorem holds for j and therefore, by mathematical induction, it holds for all $i > 1$. □

3.3.2.2 Tracing camera subpaths

Similar to the way we described tracing light subpaths we now show how tracing subpaths from the camera works. One camera subpath is traced for each pixel of the resulting image. Its tracing begins with sampling a point within the area of the corresponding pixel in the image plane. The camera origin and a direction from it to the sampled point determines the origin and direction of the first ray on the subpath. Then in a loop a ray is cast and we look for its "intersection" with the scene (an intersection with a surface or a scattering point in a medium) together with volume segments along the ray. The segments are used for evaluation of the P-B2D and B-B1D estimators and attenuation of the subpath throughput. Then, if the ray leaves the scene without any intersection, contribution of a background light is added and the subpath is terminated. Otherwise, we get a new camera vertex. In contrast to tracing light subpaths no vertices are stored, only data needed for MIS weight computation are kept for vertices of the currently traced camera subpath. If the new vertex is on a light source, its contribution is added and the subpath terminated (our renderer makes a somewhat unrealistic assumption that light sources do not reflect any light). Otherwise an attempt to connect the vertex to a randomly sampled light source and to the stored light vertices is made. Finally, the SURF and P-P3D estimators are evaluated, a new ray direction is sampled and the tracing continues. Detailed pseudocode follows:

Listing 3.25: `TraceCameraPaths` (part of the `RunIteration` method, `UPBP.hxx`)

```
for (int pathIdx = 0; pathIdx < mPathCount; ++pathIdx)
{
   // Generate an origin and a direction of the first segment
   // of a camera subpath.
   SubPathState cameraState;
   const Vec2f screenSample =
      GenerateCameraSample(pathIdx, cameraState);

   // For accumulating the result.
   Rgb color(0);

   // We assume that the camera is (on) a surface.
   bool originInMedium = false;

   // Whether only purely specular surfaces are allowed on the subpath
   // or glossy and diffuse too. False for the ''previous'' and ''compatible''
   // mode, for the ''compatible'' mode it is later reset to true after
   // the first scattering in a medium.
   bool onlySpecSurf =
      (mEstimatorTechniques & (PREVIOUS | COMPATIBLE)) != 0;

   // Trace the subpath.
   for (;; ++cameraState.mPathLength)
   {
      // Prepare a ray.
      Ray ray(cameraState.mOrigin, cameraState.mDirection);

      ..TraceCameraPathPart1..
      ..TraceCameraPathPart2..
      ..TraceCameraPathPart3..
      ..TraceCameraPathPart4..
   }

   // Store the result in the framebuffer.
   mFramebuffer.AddColor(screenSample, color);
}
```

Listing 3.26: `TraceCameraPathPart1` (part of the `RunIteration` method, `UPBP.hxx`)

```
// Trace the ray.
Isect isect(1e36f);
mVolumeSegments.clear();
mLiteVolumeSegments.clear();
if (!mScene.Intersect(
    ray, originInMedium ? AbstractMedium::kOriginInMedium : 0,
    mRng, isect, cameraState.mBoundaryStack,
    mVolumeSegments, mLiteVolumeSegments))
{
    // No scattering event occurred, the ray left the scene.

    // Evaluate PB2D (unless omitted in the algorithm).
    if (mMergeWithLightVerticesPB2D && !mLightVertices.empty())
    {
        ...EvaluatePB2DIfLeft...
    }

    // Evaluate BB1D (unless omitted in the algorithm).
    if (mMergeWithLightVerticesBB1D && !mPhotonBeamsArray.empty())
    {
        ...EvaluateBB1DIfLeft...
    }

    // Get the background light (if there is any).
    const BackgroundLight* background = mScene.GetBackground();
    if (!background)
        break;

    // Stop if there is a global attenuating medium since contribution
    // of the background light coming from infinity is always attenuated
    // to zero.
    if (mScene.GetGlobalMediumPtr()->HasAttenuation())
        break;

    // Stop if we are doing path tracing with explicit light sampling
    // and could have sampled this light last time in the next event
    // estimation (if this is not the first segment and the last sampled
    // vertex is not on a purely specular surface where no light sampling
    // is performed).
    if (mAlgorithm == kPTls && cameraState.mPathLength > 1
       && !cameraState.mLastSpecular)
        break;

    // Attenuate by intersected media (if any).
    ...AttenuateCameraSubpath...

    // Stop tracing if the path throughput is no longer positive.
    if (cameraState.mThroughput.isBlackOrNegative())
        break;

    // Update affected MIS data.
    ...SetCameraVertexMisDataIfLeft...

    // Accumulate contribution of the background light.
    color += cameraState.mThroughput *
        GetLightRadiance(mScene.GetBackground(), cameraState, Pos(0));

    // Stop tracing this subpath.
    break;
}
```

Listing 3.27: `TraceCameraPathPart2` (part of the `RunIteration` method, `UPBP.hxx`)

```
// A scattering event occurred.

// Evaluate PB2D (unless omitted in the algorithm).
if (mMergeWithLightVerticesPB2D && !mLightVertices.empty())
{
    ...EvaluatePB2D...
}

// Evaluate BB1D (unless omitted in the algorithm).
if (mMergeWithLightVerticesBB1D && !mPhotonBeamsArray.empty())
{
    ...EvaluateBB1D...
}

// Attenuate by intersected media (if any).
...AttenuateCameraSubpath...

// Stop tracing if the path throughput is no longer positive.
if (cameraState.mThroughput.isBlackOrNegative())
    break;

// Prepare scattering function at the hitpoint (BSDF/phase depending
// on whether the hitpoint is at surface or in media, the isect knows).
BSDF bsdf(
    ray, isect, mScene, BSDF::kFromCamera,
    mScene.RelativeIOR(isect, cameraState.mBoundaryStack));

// Terminate if the scattering function is invalid (e.g. when hitting
// surface too parallel with tangent plane.
if (!bsdf.IsValid())
    break;

// Compute the hitpoint
Pos hitPoint = ray.origin + ray.direction * isect.mDist;

// Current vertex will be the next origin.
originInMedium = isect.IsInMedium();

// Update affected MIS data.
...SetCameraVertexMisData...

// A light source has been hit.
if (isect.mLightID >= 0)
{
    // Stop if we are doing path tracing with explicit light sampling
    // and could have sampled this light last time in the next event
    // estimation (if this is not the first segment and the last sampled
    // vertex is not on a purely specular surface where no light sampling
    // is performed).
    if (mAlgorithm == kPTls && cameraState.mPathLength > 1
       && !cameraState.mLastSpecular)
        break;

    // Get the hit light.
    const AbstractLight *light = mScene.GetLightPtr(isect.mLightID);

    // Add its contribution.
    color += cameraState.mThroughput *
        GetLightRadiance(light, cameraState, hitPoint);

    // Terminate the subpath since our light sources do not have
    // reflective properties.
    break;
}

// Terminate if the subpath is already too long.
if (cameraState.mPathLength >= mMaxPathLength)
    break;
```

Listing 3.28: `TraceCameraPathPart3` (part of the `RunIteration` method, `UPBP.hxx`)

```
// Connect to a light source, unless omitted in the algorithm,
// the scattering function is purely specular, there are no lights
// or we are not allowed to connect from a surface (we are in
// the previous or compatible mode).
if (mConnectToLightSource && !bsdf.IsDelta() && mScene.GetLightCount() > 0
   && (bsdf.IsInMedium() || !onlySpecSurf))
{
   color += cameraState.mThroughput *
      DirectIllumination(cameraState, hitPoint, bsdf);
}

// Connect to light vertices, unless omitted in the algorithm,
// the scattering function is purely specular, there are no light
// vertices stored or we are not allowed to connect from a surface
// (we are in the previous or compatible mode).
if (mConnectToLightVertices && !bsdf.IsDelta() && !mLightVertices.empty()
   && (bsdf.IsInMedium() || !onlySpecSurf))
{
   // Get indices of vertices of a light subpath with the same subpath
   // index as this camera subpath (for vertex connection each light
   // subpath is assigned to a particular camera subpath, as in
   // traditional BPT).
   const Vec2i range(
      (pathIdx == 0) ? 0 : mPathEnds[pathIdx - 1],
      mPathEnds[pathIdx]);

   // Cycle through the indices.
   for (int i = range[0]; i < range[1]; i++)
   {
      // Get the corresponding stored light vertex.
      const UPBPLightVertex &lightVertex = mLightVertices[i];

      // Light vertices are stored in increasing path length
      // order. Once we go above the maximum path length, we can
      // skip the rest.
      if ((lightVertex.mPathLength + 1 + cameraState.mPathLength)
         > mMaxPathLength)
         break;

      // We store all light vertices in order to compute MIS weights
      // but not all can be used for connection.
      if (!lightVertex.mConnectable)
         continue;

      // Add contribution.
      color += cameraState.mThroughput * lightVertex.mThroughput
         * ConnectVertices(lightVertex, bsdf, hitPoint, cameraState);
   }
}

// Evaluate SURF, unless omitted in the algorithm, the scattering
// function is purely specular, there are no light vertices stored,
// we are in a medium or we are not allowed to merge at a surface
// (we are in the previous or compatible mode).
if (mMergeWithLightVerticesSurf && !bsdf.IsDelta()
   && !mLightVertices.empty() && bsdf.IsOnSurface() && !onlySpecSurf)
{
   ...EvaluateSURF...

   // PPM merges only at the first non-specular surface
   // from the camera.
   if (mAlgorithm == kPPM) break;
}
```

Listing 3.29: `TraceCameraPathPart4` (part of the `RunIteration` method, `UPBP.hxx`)

```
// Evaluate PP3D, unless omitted in the algorithm, the scattering
// function is purely specular, there are no light vertices stored
// or we are on a surface.
if (mMergeWithLightVerticesPP3D && !bsdf.IsDelta()
   && !mLightVertices.empty() && bsdf.IsInMedium())
{
   ...EvaluatePP3D...
}

// Continue random walk.
if (!SampleScattering(
   bsdf, hitPoint, isect, cameraState,
   mCameraVerticesMisData[cameraState.mPathLength],
   mCameraVerticesMisData[cameraState.mPathLength - 1]))
   break;

// Ensure functionality of the previous and compatible modes.
if (bsdf.IsOnSurface())
{
   // Terminate if we hit a surface and sampled non-specular component
   // of its bsdf while in the previous or compatible mode.
   if (onlySpecSurf && !cameraState.mLastSpecular)
      break;
}
else
{
   // Terminate if we sampled scattering in a medium while in the previous
   // mode. If in the compatible mode continue unlimited from now on.
   if (onlySpecSurf)
   {
      if (mEstimatorTechniques & COMPATIBLE)
         onlySpecSurf = false;
      else
         break;
   }
}
```

Initialization. The current state of the camera subpath is kept in the same structure as in the case of light subpaths – the `SubPathState` structure. Again its `mThroughput` field stores the whole subpath contribution $C_c(\mathbf{x}_j \dots \mathbf{x}_k)$ (as proved at the end of this section as Theorem 2). This time the state is initialized in the `GenerateCameraSample` method:

Listing 3.30: `GenerateCameraSample` (method, `UPBP.hxx`)

```
Vec2f GenerateCameraSample(
   const int     aPixelIndex,
   SubPathState &oCameraState)
{
   const Camera &camera = mScene.mCamera;

   // Get the camera resolution.
   const int resX = int(camera.mResolution.get(0));
   const int resY = int(camera.mResolution.get(1));

   // Based on the resolution and the given pixel (=subpath) index determine
   // the pixel coordinates.
   const int x = aPixelIndex % resX;
   const int y = aPixelIndex / resX;

   // Sample a position within the pixel (jitter the coordinates).
   const Vec2f sample = Vec2f(float(x), float(y)) + mRng.GetVec2f();

   ..GenerateCameraSamplePart2..
}
```

Listing 3.31: `GenerateCameraSamplePart2` (part of the `GenerateCameraSample` method, `UPBP.hxx`)

```
// Generate a ray coming from the camera origin and intersecting the image
// plane in the sampled position.
const Ray primaryRay = camera.GenerateRay(sample);

// Compute a PDF conversion factor from the area on the image plane
// to the solid angle on the ray.
const float cosAtCamera = dot(camera.mDirection, primaryRay.direction);
const float imagePointToCameraDist = camera.mImagePlaneDist / cosAtCamera;
const float imageToSolidAngleFactor =
    Utils::sqr(imagePointToCameraDist) / cosAtCamera;

// We put the virtual image plane at such a distance from the camera origin
// that the pixel area is one and thus the image plane sampling pdf is 1.
// The solid angle ray pdf is then equal to the conversion factor from
// image plane area density to ray solid angle density.
const float cameraPdfW = imageToSolidAngleFactor;

// Set up the camera subpath state.
oCameraState.mOrigin = primaryRay.origin;
oCameraState.mDirection = primaryRay.direction;
oCameraState.mThroughput = Rgb(1);
oCameraState.mPathLength = 1;
oCameraState.mSpecularPath = 1;
oCameraState.mLastSpecular = true;
oCameraState.mLastPdfWInv = mPathCount / cameraPdfW;

// Init the boundary stack with the global medium and a medium
// of the camera (if defined).
InitBoundaryStackForCamera*(oCameraState.mBoundaryStack);

// Return the sampled position.
return sample;
```

Note that we use $W_e(\mathbf{x}_k) = p(\omega_{\mathbf{x}_k})$ and $\hat{p}(\omega_{\mathbf{x}_k}) = \texttt{cameraPdfW}/\texttt{mPathCount}$. Therefore, $\texttt{oCameraState.mThroughput} = \texttt{Rgb(1)}\frac{W_e(\mathbf{x}_k)}{p(\omega_{\mathbf{x}_k})} = \texttt{Rgb(1)}$.

Tracing. After generating the origin and direction of the first ray we can start tracing the camera subpath. This works in a similar way as tracing a light subpath. In the `mScene.Intersect` method (see Section 3.2.3.1) the first ray is shot in the scene. It either leaves the scene or an "intersection" with the scene is found. In both cases its volume segments are returned and used for evaluation of the P-B2D and B-B1D estimators. The segments are interpreted as query beams and nearby photons and photon beams stored during tracing light subpaths are looked up and possible contribution of the estimators is computed for them. See Sections 3.4.2, 3.4.3 for more information. Then the subpath throughput is attenuated:

Listing 3.32: `AttenuateCameraSubpath` (part of the `RunIteration` method, `UPBP.hxx`)

```
float raySamplePdf(1.0f); float raySampleRevPdf(1.0f);
if (!mVolumeSegments.empty())
{
    // Pdf of sampling through the segments (needed for attenuation
    // and MIS weighting).
    raySamplePdf = VolumeSegment::AccumulatePdf(mVolumeSegments);
    // Pdf of sampling through the segments in the reverse direction
    // (needed for MIS weighting).
    raySampleRevPdf = VolumeSegment::AccumulateRevPdf(mVolumeSegments);
    // Attenuation by the segments.
    cameraState.mThroughput *=
        VolumeSegment::AccumulateAttenuationWithoutPdf(mVolumeSegments)
        / raySamplePdf;
}
```

Light hit. If the ray intersected the scene, the `BSDF` object (see Section 3.2.4) is created for the new vertex, data needed for MIS weight computation are updated (see Section 3.5) and contribution of possibly directly hit light source is evaluated in the `GetLightRadiance` method. If the ray left the scene, MIS weights data are also updated and contribution of a background light (if present) is computed in the same `GetLightRadiance` method. Here it is:

Listing 3.33: `GetLightRadiance` (method, `UPBP.hxx`)

```
Rgb GetLightRadiance(
    const AbstractLight *aLight,
    const SubPathState  &aCameraState,
    const Pos           &aHitpoint) const
{
    // aLight = the hit light = either a finite light intersected
    // by the ray or a background light if the ray left the scene.

    // Get probability of sampling the hit light in case of explicit
    // light sampling (needed for MIS weighting).
    const int   lightCount = mScene.GetLightCount();
    const float lightPickProb = 1.f / lightCount;

    // Get radiance emitted by the hit light in the incident direction.
    float directPdfA, emissionPdfW;
    const Rgb radiance = aLight->GetRadiance(mScene.mSceneSphere,
        aCameraState.mDirection, aHitpoint, &directPdfA, &emissionPdfW);

    if (radiance.isBlackOrNegative())
        return Rgb(0);

    // If the light was hit directly from the camera, no weighting is required.
    if (aCameraState.mPathLength == 1)
        return radiance;

    // When evaluating only photon density estimators, only purely specular
    // camera subpaths are allowed to give radiance when hitting a light
    // (we cannot get it otherwise and the rest is handled by the estimators
    // at non-specular vertices).
    if (mEstimatorTechniques && !(mEstimatorTechniques & BPT))
        return aCameraState.mSpecularPath ? radiance : Rgb(0);

    // Complete the probabilities.
    directPdfA *= lightPickProb;
    emissionPdfW *= lightPickProb;

    // Compute correct MIS weight.
    float misWeight = 1.f;
    ...DirectlyHitLightMis...

    // Return weighted result.
    return misWeight * radiance;
}
```

Radiance emitted by the hit light is computed in the `aLight->GetRadiance` method. The method also returns the probability density of the ray hitting the light in the given point and direction as if it was a result of explicit light sampling (`directPdfA`) or of sampling emission of the light (`emissionPdfW`). The meaning of these two pdfs is the same as in the `GenerateLightSample` method (Listing 3.17) and they are needed for MIS weight computation. Before returning the radiance it needs to be properly weighted. Section 3.5 describes how to accomplish this. The returned radiance is then added multiplied by subpath throughput according to Equation 2.29.

Since

$$\begin{aligned}\texttt{cameraState.mThroughput} &= C_{\mathrm{c}}(\mathbf{x}_0 \dots \mathbf{x}_k),\\ \texttt{misWeight} &= \hat{w}_{\mathrm{BPT}_{\mathrm{direct}}},\\ \texttt{radiance} &= L_{\mathrm{e}}(\mathbf{x}_0)\end{aligned}$$

the final contribution of the connection (as added on Line 55 in Listing 3.26 and 58 in Listing 3.27) is:

$$\begin{aligned}\texttt{color+=cameraState.mThroughput*contrib}&\\ =&\hat{w}_{\mathrm{BPT}_{\mathrm{direct}}} L_{\mathrm{e}}(\mathbf{x}_0) C_{\mathrm{c}}(\mathbf{x}_0 \dots \mathbf{x}_k)\\ =&\hat{w}_{\mathrm{BPT}_{\mathrm{direct}}} L_{\mathrm{e}}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{x}_k)}{p(\mathbf{x}_0 \dots \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k).\end{aligned}$$

Light sampling. If the ray left the scene the subpath is terminated. Otherwise, explicit light sampling is performed, i.e. the new camera vertex (at the found intersection) is connected to randomly chosen point on a light source. It actually means sampling direct illumination for the camera vertex, therefore a method that implements it bears the name `DirectIllumination`:

Listing 3.34: `DirectIllumination` (method, `UPBP.hxx`)

```
1  Rgb DirectIllumination(
2     const SubPathState  &aCameraState,
3     const Pos           &aHitpoint,
4     const BSDF          &aCameraBSDF)
5  {
6     // Choose a light uniformly.
7     const int            lightCount    = mScene.GetLightCount();
8     const float          lightPickProb = 1.f / lightCount;
9     const int            lightID       = int(mRng.GetFloat() * lightCount);
10    const AbstractLight *light         = mScene.GetLightPtr(lightID);
11
12    // The chosen light may need these random numbers for sampling
13    // a position on it.
14    const Vec2f rndPosSamples = mRng.GetVec2f();
15
16    // Light in infinity in attenuating homogeneous global medium is always
17    // reduced to zero.
18    if (!light->IsFinite() && mScene.GetGlobalMediumPtr()->HasAttenuation())
19       return Rgb(0);
20
21    // Get radiance coming from the chosen light.
22    Dir directionToLight;
23    float distance;
24    float directPdfW, emissionPdfW, cosAtLight;
25    const Rgb radiance = light->Illuminate(mScene.mSceneSphere, aHitpoint,
26       rndPosSamples, directionToLight, distance, directPdfW,
27       &emissionPdfW, &cosAtLight);
28
29    if (radiance.isBlackOrNegative())
30       return Rgb(0);
31
32    // Evaluate the scattering function at the camera vertex.
33    float bsdfDirPdfW, bsdfRevPdfW, cosToLight, sinTheta;
34    Rgb bsdfFactor = aCameraBSDF.Evaluate(
35       directionToLight, cosToLight, &bsdfDirPdfW, &bsdfRevPdfW, &sinTheta);
36
37    if (bsdfFactor.isBlackOrNegative())
38       return Rgb(0);
39
40    ..DirectIlluminationPart2..
41 }
```

Listing 3.35: `DirectIlluminationPart2` (part of the `DirectIllumination` method, `UPBP.hxx`)

```
// Needed for MIS weighting only, we are making an explicit connection,
// there is no Russian roulette decision.
const float continuationProbability = aCameraBSDF.ContinuationProb();

// If the light is delta light, we can never hit it
// by BSDF sampling, so the probability of this path is 0.
bsdfDirPdfW *= light->IsDelta() ? 0.f : continuationProbability;
bsdfRevPdfW *= continuationProbability;

Rgb contrib(0);

// Test whether the connection is not occluded by geometry and find volume
// segments along it.
mVolumeSegments.clear();
if (!mScene.Occluded(
    aHitpoint, directionToLight, distance, aCameraState.mBoundaryStack,
    aCameraBSDF.IsInMedium() ? AbstractMedium::kOriginInMedium : 0,
    mVolumeSegments))
    {
        // Get attenuation from intersected media (if any).
        float raySamplePdf(1.0f);
        float raySampleRevPdf(1.0f);
        Rgb attenuation(1.0f);
        if (!mVolumeSegments.empty())
        {
            // Pdf of sampling through the segments.
            raySamplePdf = VolumeSegment::AccumulatePdf(mVolumeSegments);

            // Pdf of sampling through the segments in the reverse direction.
            raySampleRevPdf =
                VolumeSegment::AccumulateRevPdf(mVolumeSegments);

            // Attenuation by the segments (without pdf since we made
            // an explicit connection and did not sample media).
            attenuation =
                VolumeSegment::AccumulateAttenuationWithoutPdf(mVolumeSegments);

            if (!nextAttenuation.isPositive())
                return Rgb(0);
        }

        // Compute correct MIS weight.
        float misWeight = 1.f;
        ...ConnectToLightMis...

        // Compute the result.
        contrib = misWeight * (cosToLight / (lightPickProb * directPdfW))
            * (radiance * attenuation * bsdfFactor);
    }

// Return the result.
return contrib;
```

The method starts with uniform sampling of light sources in the scene. One light source is randomly chosen and its `light->Illuminate` method is called. It samples the light (if necessary, e.g. for area lights) and returns the resulting direction (from the camera vertex to the sampled light point), radiance coming in this direction from the light and pdfs of the sampling (`directPdfW`) and of sampling emission of the light (`emissionPdfW`). Note that while `emissionPdfW` is the same as in the `GenerateLightSample` and `GetLightRadiance` methods (Listing 3.17 and 3.33), `directPdfW` is different. In contrast to `directPdfA` in the other two methods, `directPdfW` is expressed w.r.t. the solid angle measure at the camera vertex and is equal to `directPdfA`$\|\mathbf{x}_1 - \mathbf{x}_0\|^2 / D(\mathbf{x}_0 \to \mathbf{x}_1)$. Then the `aCameraBSDF.Evaluate` method computes the scattering function factor at the camera vertex for a direction

of the last camera subpath segment and the computed direction. It also returns pdfs, namely pdfs of sampling the scattering function in the direct (i.e. from the camera) and reverse direction (`bsdfDirPdfW` and `bsdfRevPdfW`, respectively). The next step is to test whether the connection between the camera vertex and the light is not occluded by geometry. This is done by ray casting in the `mScene.Occluded` method, which also finds volume segments intersected by the ray. These segments are then used for accumulating attenuation by media along the connection. For more information about the two aforementioned methods see Section 3.2. Together with the attenuation pdfs of sampling through the segments in a direct (i.e. from the camera) and reverse direction (`raySamplePdf` and `raySampleRevPdf`, respectively) are computed. Note that we are making an explicit connection, there is no sampling of media, scattering function or light emission, therefore the pdfs `raySamplePdf` and `raySampleRevPdf` as well as `bsdfDirPdfW`, `bsdfRevPdfW` and `emissionPdfW` are needed only for MIS weight computation. It comes right after the attenuation part and is closely described in Section 3.5. Finally, the resulting contribution is computed according to Equation 2.24 (with $\mathbf{a} = \mathbf{x}_0$). Since

$$\begin{aligned}
\texttt{cameraState.mThroughput} &= C_{\mathrm{c}}(\mathbf{x}_1 \dots \mathbf{x}_k), \\
\texttt{misWeight} &= \hat{w}_{BPT_{\mathbf{x}_0,\mathbf{x}_1}}, \\
\frac{\texttt{cosToLight}}{\texttt{lightPickProb*directPdfW}} &= \frac{G(\mathbf{x}_0, \mathbf{x}_1)}{p(\mathbf{x}_0)}, \\
\texttt{radiance} &= L_{\mathrm{e}}(\mathbf{x}_0), \\
\texttt{attenuation} &= T_{\mathrm{r}}(\mathbf{x}_0, \mathbf{x}_1), \\
\texttt{bsdfFactor} &= \rho(\mathbf{x}_1),
\end{aligned}$$

the final contribution of the connection (as added on Line 8 in Listing 3.28) is:

$$\begin{aligned}
&\texttt{color+=cameraState.mThroughput*contrib} \\
&= \hat{w}_{BPT_{\mathbf{x}_0,\mathbf{x}_1}} L_{\mathrm{e}}(\mathbf{x}_0) \frac{T_{\mathrm{r}}(\mathbf{x}_0, \mathbf{x}_1) G(\mathbf{x}_0, \mathbf{x}_1) \rho(\mathbf{x}_1)}{p(\mathbf{x}_0)} C_{\mathrm{c}}(\mathbf{x}_1 \dots \mathbf{x}_k) \\
&= \hat{w}_{BPT_{\mathbf{x}_0,\mathbf{x}_1}} L_{\mathrm{e}}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{x}_k)}{p(\mathbf{x}_0) p(\mathbf{x}_1 \dots \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k).
\end{aligned}$$

Vertex connection. After we tried connecting the last camera vertex to a light source, connecting to the stored light vertices (i.e. evaluation of the rest of the BPT techniques) is an obvious next step. We take all stored light vertices of the corresponding light subpath (the one which has the same index as the currently traced camera subpath) and try to connect the camera vertex to each of them. Evaluation of one such connection is implemented in the `ConnectVertices` method.

Listing 3.36: `ConnectVertices` (method, `UPBP.hxx`)

```
Rgb ConnectVertices(
    const UPBPLightVertex &aLightVertex,
    const BSDF            &aCameraBSDF,
    const Pos             &aCameraHitpoint,
    const SubPathState    &aCameraState)
{
    // Get a direction and distance of the connection.
    Dir direction     = aLightVertex.mHitpoint - aCameraHitpoint;
    const float dist2 = direction.square();
    float  distance   = std::sqrt(dist2);
    direction        /= distance;

    // Evaluate the scattering function at the camera vertex.
    float cosCamera, cameraBsdfDirPdfW, cameraBsdfRevPdfW, sinThetaCamera;
    Rgb cameraBsdfFactor = aCameraBSDF.Evaluate(
        direction, cosCamera, &cameraBsdfDirPdfW,
        &cameraBsdfRevPdfW, &sinThetaCamera);

    if (cameraBsdfFactor.isBlackOrNegative())
        return Rgb(0);

    // Camera continuation probability (needed for MIS weighting only).
    const float cameraCont = aCameraBSDF.ContinuationProb();
    cameraBsdfDirPdfW *= cameraCont;
    cameraBsdfRevPdfW *= cameraCont;

    // Evaluate the scattering function at the light vertex.
    float cosLight, lightBsdfDirPdfW, lightBsdfRevPdfW, sinThetaLight;
    const Rgb lightBsdfFactor = aLightVertex.mBSDF.Evaluate(
        -direction, cosLight, &lightBsdfDirPdfW,
        &lightBsdfRevPdfW, &sinThetaLight);

    if (lightBsdfFactor.isBlackOrNegative())
        return Rgb(0);

    // Light continuation probability (needed for MIS weighting only).
    const float lightCont = aLightVertex.mBSDF.ContinuationProb();
    lightBsdfDirPdfW *= lightCont;
    lightBsdfRevPdfW *= lightCont;

    // Compute a geometry term between the two vertices.
    const float geometryTerm = cosLight * cosCamera / dist2;
    if (geometryTerm < 0)
        return Rgb(0);

    // Convert solid angle PDFs to area PDFs.
    const float cameraBsdfDirPdfA =
        PdfWtoA(cameraBsdfDirPdfW, distance, cosLight);
    const float lightBsdfDirPdfA =
        PdfWtoA(lightBsdfDirPdfW, distance, cosCamera);

    // Prepare flags for the occlusion testing method. It needs to know if
    // the origin (the camera vertex) and end (the light vertex) of
    // the connection are in a medium or not to correctly compute
    // sampling pdfs.
    uint raySamplingFlags = 0;
    if (aCameraBSDF.IsInMedium())
        raySamplingFlags |= AbstractMedium::kOriginInMedium;
    if (aLightVertex.mInMedium)
        raySamplingFlags |= AbstractMedium::kEndInMedium;

    // Test whether the connection is not occluded by geometry and find volume
    // segments along it.
    mVolumeSegments.clear();
    if (mScene.Occluded(
        aCameraHitpoint, direction, distance, aCameraState.mBoundaryStack,
        raySamplingFlags, mVolumeSegments))
        return Rgb(0);

    ..ConnectVerticesPart2..
}
```

Listing 3.37: `ConnectVerticesPart2` (part of the `ConnectVertices` method, `UPBP.hxx`)

```
// Get attenuation from intersected media (if any).
float raySamplePdf(1.0f);
float raySampleRevPdf(1.0f);
Rgb attenuation(1.0f);
if (!mVolumeSegments.empty())
{
   // Pdf of sampling through the segments.
   raySamplePdf = VolumeSegment::AccumulatePdf(mVolumeSegments);

   // Pdf of sampling through the segments in the reverse direction.
   raySampleRevPdf = VolumeSegment::AccumulateRevPdf(mVolumeSegments);

   // Attenuation by the segments (without pdf since we made
   // an explicit connection and did not sample media).
   attenuation =
      VolumeSegment::AccumulateAttenuationWithoutPdf(mVolumeSegments);

   if (!mediaAttenuation.isPositive())
   return Rgb(0);
}

// Compute correct MIS weight.
float misWeight = 1.f;
...ConnectToVertexMis...

// Compute the result.
Rgb contrib = misWeight * cameraBsdfFactor * lightBsdfFactor
   * attenuation * geometryTerm;

// Return the result.
return contrib;
```

To compute the contribution of the connection we first need to evaluate the scattering function at both ends, i.e. at the camera vertex and at the light vertex. This is done by the `aCameraBSDF.Evaluate` and `aLightVertex.mBSDF.Evaluate` method, respectively. Except for the factors these methods again return pdfs of sampling the scattering function in a direct direction (`cameraBsdfDirPdfW` from the camera, `lightBsdfDirPdfW` from the light) and a reverse direction (`cameraBsdfRevPdfW` from the light, `lightBsdfRevPdfW` from the camera). Since we are making an explicit connection these pdfs are needed for MIS weighting only. The methods also return cosines at the connecting edge which are subsequently used for computing a geometry term. Then the `mScene.Occluded` method tests whether the connection is not occluded by geometry and finds volume segments intersected by the connecting edge. Evaluation of the scattering function as well as the occlusion testing is more closely described in Section 3.2. The volume segments are used for accumulating attenuation caused by media along the connection and computing pdfs of sampling through the media in a direct (i.e. from the camera) and reverse direction (`raySamplePdf` and `raySampleRevPdf`, respectively) needed for MIS weighting. The MIS weight is computed right after that (see Section 3.5) and used for weighting a contribution of the connection computed according to Equation 2.24.

Since

$$\begin{aligned}
\texttt{cameraState.mThroughput} &= C_{\mathrm{c}}(\mathbf{x}_s \dots \mathbf{x}_k),\\
\texttt{lightVertex.mThroughput} &= C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_{s-1}),\\
\texttt{misWeight} &= \hat{w}_{BPT_{\mathbf{x}_{s-1},\mathbf{x}_s}},\\
\texttt{cameraBsdfFactor} &= \rho(\mathbf{x}_s),\\
\texttt{lightBsdfFactor} &= \rho(\mathbf{x}_{s-1}),\\
\texttt{attenuation} &= T_{\mathrm{r}}(\mathbf{x}_{s-1}, \mathbf{x}_s),\\
\texttt{geometryTerm} &= G(\mathbf{x}_{s-1}, \mathbf{x}_s),
\end{aligned}$$

the final contribution of the connection (as added on Line 46 in Listing 3.28) is:

$$\begin{aligned}
&\texttt{color+=cameraState.mThroughput*lightVertex.mThroughput*contrib}\\
&\quad = \hat{w}_{BPT_{\mathbf{x}_{s-1},\mathbf{x}_s}} C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_{s-1}) T_{\mathrm{r}}(\mathbf{x}_{s-1}, \mathbf{x}_s) G(\mathbf{x}_{s-1}, \mathbf{x}_s)\\
&\qquad \rho(\mathbf{x}_s)\rho(\mathbf{x}_{s-1}) C_{\mathrm{c}}(\mathbf{x}_s \dots \mathbf{x}_k)\\
&\quad = \hat{w}_{BPT_{\mathbf{x}_{s-1},\mathbf{x}_s}} L_{\mathrm{e}}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{x}_k)}{p(\mathbf{x}_0 \dots \mathbf{x}_{s-1}) p(\mathbf{x}_s \dots \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k).
\end{aligned}$$

Last steps. That is all about the `ConnectVertices` method, let's continue with description of tracing the camera subpath. Evaluation of BPT techniques is just finished, beam-based photon density estimators were evaluated right after casting the ray so the only thing remaining are the point-based photon density estimators, i.e. SURF and P-P3D. Their evaluation is performed now and is described in Section 3.4.1. Everything on the camera subpath of the current length is then done, the last step – adding another segment – is the same as when tracing light subpaths. It is also implemented in the same method `SampleScattering`, see Listing 3.24 for its description.

Closing the loop. At this point the inner loop of tracing camera subpaths starts a new run, a new ray is shot from the last vertex in a direction sampled in the `SampleScattering` method and tracing continues. It stops when the subpath throughput drops to zero, the Russian roulette decision fails or the subpath reaches its maximum length. Then a new camera subpath is traced. This way n_{paths} (`mPathCount`) camera subpaths are traced.

To conclude our description of tracing camera subpaths we prove similar statement as for light subpaths:

Theorem 2. *Assume any camera subpath traced by the described algorithm, number $k \geq 1$ and the* `cameraState` *structure at vertex $\mathbf{x}_{k-i}, i \in [1, k]$, i.e. after attenuation by media along the i-th segment but before sampling the scattering function (at lines 46-59 in Listing 3.26 and from 17 in Listing 3.27 to 9 in Listing 3.29). Then its* `mThroughput` *field satisfies:*

$$\texttt{mThroughput} = C_{\mathrm{c}}(\mathbf{x}_{k-i} \dots \mathbf{x}_k), \tag{3.9}$$

where $C_{\mathrm{c}}(\mathbf{x}_{k-i} \dots \mathbf{x}_k)$ is the subpath contribution (1.7)*.*

Proof. Let $i = 1$. The `cameraState` is initialized in the `GenerateCameraSample` method and its `mThroughput` field is assigned `Rgb(1.0f)` there (Listing 3.31, line 21). That equals $\texttt{Rgb(1)}\frac{W_{\mathrm{e}}(\mathbf{x}_k)}{p(\omega_{\mathbf{x}_k})}$. Then the `cameraState.mThroughput` value is modified with attenuation by media along the first segment (Listing 3.32, line 11). It becomes

$$\texttt{mThroughput} = \frac{T_{\mathrm{r}}(t_{\mathbf{x}_k})}{p(\mathbf{x}_k)p(\omega_{\mathbf{x}_k})p(t_{\mathbf{x}_k})} W_{\mathrm{e}}(\mathbf{x}_k).$$

$T_{\mathrm{r}}(t_{\mathbf{x}_k})$ is the accumulated attenuation, $p(\mathbf{x}_k) = 1$, $p(t_{\mathbf{x}_k})$ is `raySamplePdf`, i.e. pdf of sampling the distance $t_{\mathbf{x}_0}$ through the media w.r.t. the Euclidean length on $\mathbb{R}^1$. Following the same steps as when deriving Equation 2.7 and we get

$$\begin{aligned}
\texttt{mThroughput} &= \frac{T_{\mathrm{r}}(t_{\mathbf{x}_k})}{p(\mathbf{x}_k)p(\omega_{\mathbf{x}_k})p(t_{\mathbf{x}_k})} W_{\mathrm{e}}(\mathbf{x}_k) \\
&= \frac{T_{\mathrm{r}}(t_{\mathbf{x}_k})}{p(\mathbf{x}_k)p(\omega_{\mathbf{x}_k})p(t_{\mathbf{x}_k})} \frac{G(\mathbf{x}_{k-1}, \mathbf{x}_k)}{G(\mathbf{x}_{k-1}, \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k) \\
&= \frac{T_{\mathrm{r}}(t_{\mathbf{x}_k}) G(\mathbf{x}_{k-1}, \mathbf{x}_k)}{p(\mathbf{x}_k)p(\omega_{\mathbf{x}_k})p(t_{\mathbf{x}_k}) G(\mathbf{x}_{k-1}, \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k) \\
&= \frac{T(\mathbf{x}_{k-1} \dots \mathbf{x}_k)}{p(\mathbf{x}_{k-1} \dots \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k) \\
&= C_{\mathrm{c}}(\mathbf{x}_{k-1} \dots \mathbf{x}_k).
\end{aligned}$$

The `cameraState.mThroughput` value is then not modified until sampling the scattering function in the `SampleScattering` method. The theorem holds for $i = 1$.

Let $i > 1$. Assume the theorem holds for $j - 1, j \in (1, k]$, i.e. `mThroughput` at vertex $\mathbf{x}_{k-j+1}$ equals $C_{\mathrm{c}}(\mathbf{x}_{k-j+1} \dots \mathbf{x}_k)$. Between vertices $\mathbf{x}_{k-j+1}$ and $\mathbf{x}_{k-j}$ the value of `cameraState.mThroughput` is modified twice. First while sampling the scattering function (Listing 3.24, line 43):

$$\texttt{mThroughput} = C_{\mathrm{c}}(\mathbf{x}_{k-j+1} \dots \mathbf{x}_k) \frac{\rho(\mathbf{x}_{k-j+1}) D(\mathbf{x}_{k-j+1} \to \mathbf{x}_{k-j})}{p_{RR}(\mathbf{x}_{k-j+1}) \hat{p}(\omega_{\mathbf{x}_{k-j+1}})}.$$

$\rho(\mathbf{x}_{k-j+1})$ is `bsdfFactor` returned by the `aBSDF.Sample` method, $D(\mathbf{x}_{k-j+1} \to \mathbf{x}_{k-j})$ = `cosThetaOut`, $p_{RR}(\mathbf{x}_{k-j+1})$ is the probability of the Russian roulette decision and $\hat{p}(\omega_{\mathbf{x}_{k-j+1}})$ the probability density of sampling the scattering function w.r.t. to the solid angle measure. Note $p_{RR}(\mathbf{x}_{k-j+1})\hat{p}(\omega_{\mathbf{x}_{k-j+1}}) =$ `bsdfDirPdfW`.

The second modification is the already described attenuation by intersected media. Repeating the same steps we get

$$\begin{aligned}\texttt{mThroughput} &= C_{\mathrm{c}}(\mathbf{x}_{k-j+1}\dots\mathbf{x}_k)\frac{\rho(\mathbf{x}_{k-j+1})D(\mathbf{x}_{k-j+1}\to\mathbf{x}_{k-j})}{p_{RR}(\mathbf{x}_{k-j+1})\hat{p}(\omega_{\mathbf{x}_{k-j+1}})}\frac{T_{\mathrm{r}}(t_{\mathbf{x}_{k-j+1}})}{p(t_{\mathbf{x}_{k-j+1}})}\\ &= \frac{T(\mathbf{x}_{k-j+1}\dots\mathbf{x}_k)}{p(\mathbf{x}_{k-j+1}\dots\mathbf{x}_k)}\frac{\rho(\mathbf{x}_{k-j+1})G(\mathbf{x}_{k-j+1},\mathbf{x}_{k-j})T_{\mathrm{r}}(t_{\mathbf{x}_{k-j+1}})}{p_{RR}(\mathbf{x}_{k-j+1})p(\mathbf{x}_{k-j}\,|\,\mathbf{x}_{k-j+1})}\\ &= \frac{T(\mathbf{x}_{k-j}\dots\mathbf{x}_k)}{p(\mathbf{x}_{k-j}\dots\mathbf{x}_k)} = C_{\mathrm{c}}(\mathbf{x}_{k-j}\dots\mathbf{x}_k).\end{aligned}$$

for $W_{\mathrm{e}}(\mathbf{x}_k) = 1$. The theorem holds for j and therefore, by mathematical induction, it holds for all $i > 1$. □

As a consequence of Theorem 2, the `cameraState.mThroughput` value on lines 58 in Listing 3.27, 8 and 46 in Listing 3.28 equals the subpath contribution C_e.

We finished our description of the `UPBP` renderer. We showed its main structure and then thoroughly went through tracing of subpaths. We conclude this section with a few notes about other renderers, the following sections will then focus on other important parts of the code the renderers use (photon density estimators implementation and MIS weights computation).

3.3.3 Other renderers

As mentioned at the beginning of this section there are three other renderers we newly implemented besides `UPBP` – `VolPathTracer`, `VolLightTracer` and `VolBidirPT`.

`VolPathTracer.` Implements traditional path tracing with next event estimation. It can be run in three modes, either it simulates the simplest path tracing without light sampling, which waits for a direct hit of a light source, or path tracing with explicit light sampling only, which does not accumulate emission of directly hit light sources. The third mode is path tracing with contributions of light sampling and directly hit light sources combined by MIS. Its output almost equals to running the `UPBP` renderer with `mAlgorithm` set to `kPTdir`, `kPTls` or `kPTmis`, respectively. The only difference is that the `VolPathTracer` renderer, as the only one of the renderers, supports light emission by participating media. It is easy with path tracing as it only means adding emission of the intersected volume segments right before using them for attenuating the path throughput. However, it becomes difficult when tracing light subpaths as it would mean storing emission of the segments and then going through them in the opposite direction and adding them multiplied by throughput accumulated also in the opposite direction. That is why we decided not to support media emission in other renderers.

`VolLightTracer.` Simulates ordinary light tracing and can also evaluate the P-B2D and B-B1D estimators. Its output almost equals to running the `UPBP` renderer with `mAlgorithm` set to `kLT` or to `kCustom` with `mEstimatorTechniques` containing `kPB2D` or `kBB1D`, respectively. However, in contrast to `UPBP` the photon density estimators are evaluated only for primary camera rays as there is no camera subpath tracing.

`VolBidirPT`. Implements bidirectional path tracing. It can also run in three unidirectional path tracing modes, the same as `VolPathTracer`. Its output exactly equals to running the `UPBP` renderer with `mAlgorithm` set to `kBPT`, `kPTdir`, `kPTls` or `kPTmis`, respectively.

Original renderers. The rest of the renderers, `EyeLight`, `PathTracer` and `VertexCM`, stay almost the same as in the original SmallVCM implementation. However, they are modified to work with our new ray-scene intersection method used by the newly added renderers. Since the original renderers are not capable of handling scattering in media the method is given a flag meaning that all media should be ignored. The original functionality is retained.

3.4 Photon density estimators implementation

For the sake of clarity we left out a description of several code parts in the previous section. We return to them in the following text. Here we begin with implementation of the photon density estimators.

3.4.1 SURF and P-P3D

First, there are the SURF and P-P3D estimators. They are very similar, they differ only in the kernel dimension (2D vs. 3D) and location of vertices they can be evaluated on (on a surface vs. in a medium). Therefore, they share almost the same code.

These two estimators evaluate the contribution of stored photon points distributed from light sources over a scene at query points distributed from the camera.

3.4.1.1 Photon points

The photon points are the light vertices stored during tracing of light subpaths. The SURF estimator uses those located on surfaces, the P-P3D estimator those in media. Note that we do not reuse these vertices among iterations, i.e. every iteration uses only light vertices created in that iteration. For faster photon point lookup we build hashed grids over them, one for each of the two estimators. A hashed grid is implemented in the `HashGrid` class (in the `HashGrid.hxx` file). The build is handled by its `Build` method and takes place right after all light subpaths have been traced and before tracing camera subpaths (on lines 18-30 in Listing 3.13). It takes light vertices with the right location (on a surface/in a medium depending on the estimator), divides their bounding box into cube cells with a side two kernel radii long and hashes content of these cells into a small number of bins (`mPathCount` according to our heuristic).

3.4.1.2 Query points

The query points are the camera vertices created during tracing of a camera subpath. The evaluation of the two estimators takes place at each camera vertex before

sampling scattering and continuing the subpath (on lines 50in Listing 3.28 till 9 in Listing 3.29). Here is the left out code:

Listing 3.38: `EvaluateSURF` (part of the `RunIteration` method, `UPBP.hxx`)

```
RangeQuery query(*this, hitPoint, bsdf, cameraState);
mSurfHashGrid.Process(mLightVertices, query);
color += cameraState.mThroughput * mSurfNormalization * query.GetContrib();
```

Listing 3.39: `EvaluatePP3D` (part of the `RunIteration` method, `UPBP.hxx`)

```
RangeQuery query(*this, hitPoint, bsdf, cameraState);
mPP3DHashGrid.Process(mLightVertices, query);
color += cameraState.mThroughput * mPP3DNormalization * query.GetContrib();
```

The code is very similar for both estimators. Firstly, an object of the `RangeQuery` class (from the `UPBP.hxx` file) is created, it stores all data necessary for evaluation of the estimators. After that, it is passed to the `Process` method of a corresponding hashed grid. This method searches 8 nearest cells around the query point (`hitPoint`) and finds all photon points there that lie within kernel radius from the query point (no point from other cells can be within the radius because of the cell dimension). For each of them it then calls the `Process` method of the given `RangeQuery` object to evaluate the corresponding estimator:

Listing 3.40: `Process` (method, `UPBP.hxx`)

```
void Process(const UPBPLightVertex& aLightVertex)
{
    // We store all light vertices but not all can be used for merging
    // (delta and light).
    if (!aLightVertex.mConnectable) return;

    // Reject if the full path length would be above the maximum path length.
    if (aLightVertex.mPathLength + mCameraState.mPathLength >
        mUPBP.mMaxPathLength)
        return;

    // Retrieve light incoming direction in the world coordinates.
    const Dir lightDirection = aLightVertex.mBSDF.WorldDirFix();

    // Evaluate the scattering function at the camera vertex.
    float cosCamera, cameraBsdfDirPdfW, cameraBsdfRevPdfW, sinTheta;
    const Rgb cameraBsdfFactor = mCameraBsdf.Evaluate(
        lightDirection, cosCamera, &cameraBsdfDirPdfW,
        &cameraBsdfRevPdfW, &sinTheta);

    if (cameraBsdfFactor.isBlackOrNegative()) return;

    // Complete the forward path pdf.
    cameraBsdfDirPdfW *= mCameraBsdf.ContinuationProb();

    // Complete the reverse path pdf. Even though this is pdf from the
    // camera BSDF, the continuation probability must come from the light
    // BSDF, because that would govern it if the light path actually continued.
    cameraBsdfRevPdfW *= aLightVertex.mBSDF.ContinuationProb();

    // Compute correct MIS weight.
    float misWeight = 1.0f;
    if (mUPBP.mAlgorithm != kPPM)
    {
        ...SurfPP3DMis...
    }

    // Accumulate contribution.
    mContrib += misWeight * cameraBsdfFactor * aLightVertex.mThroughput;
}
```

At first, the scattering function is evaluated using the `mCameraBsdf.Evaluate` method (described in Section 3.2.4). The scattering function at the camera vertex is used, i.e. the evaluation is carried out for the material/medium at the camera vertex, the incoming ray direction at the camera vertex (as incoming) and the incoming direction at the light vertex (as outgoing). The returned pdfs of sampling the scattering function in the direct (`cameraBsdfDirPdfW`) and reverse direction (`cameraBsdfRevPdfW`) are needed only for computation of the MIS weight, which comes right after that and is described in Section 3.5. Finally, weighted contribution is accumulated.

Result. This way the weighted contribution of all photon points stored within the kernel radius from the query point is accumulated inside the `RangeQuery` object. After returning from the `Process` method this contribution is added multiplied by the camera subpath throughput and a corresponding normalization factor including a kernel according to Equations 2.32 and 2.7. Let's consider one of the included photon points $\tilde{\mathbf{x}}_s$. Then

$$\begin{aligned}
\texttt{cameraState.mThroughput} &= C_{\mathrm{c}}(\mathbf{x}_s \dots \mathbf{x}_k), \\
\texttt{mSurfNormalization} &= \frac{K_2(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{n_{paths}}, \\
\texttt{mPP3DNormalization} &= \frac{K_3(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{n_{paths}}, \\
\texttt{cameraBsdfFactor} &= \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s), \\
\texttt{aLightVertex.mThroughput} &= C_{\mathrm{l}}(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s).
\end{aligned}$$

If the SURF estimator is being evaluated, then $\texttt{misWeight} = \hat{w}_{\mathrm{SURF}_s}$ and an individual contribution of photon point $\tilde{\mathbf{x}}_s$ (as added on line 3 in Listing 3.38) is:

```
color+=cameraState.mThroughput * mSurfNormalization * misWeight
       * cameraBsdfFactor * aLightVertex.mThroughput
```

$$\begin{aligned}
&= \frac{1}{n_{paths}} \hat{w}_{\mathrm{SURF}_s} C_{\mathrm{l}}(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s) \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) K_2(\tilde{\mathbf{x}}_s, \mathbf{x}_s) C_{\mathrm{c}}(\mathbf{x}_s \dots \mathbf{x}_k) \\
&= \frac{1}{n_{paths}} \hat{w}_{\mathrm{SURF}_s} L_{\mathrm{e}}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s)} \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) K_2(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{T(\mathbf{x}_s \dots \mathbf{x}_k)}{p(\mathbf{x}_s \dots \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k).
\end{aligned}$$

If the P-P3D estimator is being evaluated, then $\texttt{misWeight} = \hat{w}_{\mathrm{P\text{-}P3D}_s}$ and an individual contribution of photon point $\tilde{\mathbf{x}}_s$ (as added on line 3 in Listing 3.39) is:

```
color+=cameraState.mThroughput * mPP3DNormalization * misWeight
       * cameraBsdfFactor * aLightVertex.mThroughput
```

$$\begin{aligned}
&= \frac{1}{n_{paths}} \hat{w}_{\mathrm{P\text{-}P3D}_s} C_{\mathrm{l}}(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s) \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) K_3(\tilde{\mathbf{x}}_s, \mathbf{x}_s) C_{\mathrm{c}}(\mathbf{x}_s \dots \mathbf{x}_k) \\
&= \frac{1}{n_{paths}} \hat{w}_{\mathrm{P\text{-}P3D}_s} L_{\mathrm{e}}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s)} \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) K_3(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{T(\mathbf{x}_s \dots \mathbf{x}_k)}{p(\mathbf{x}_s \dots \mathbf{x}_k)} W_{\mathrm{e}}(\mathbf{x}_k).
\end{aligned}$$

Kernel. For these two estimators we use constant kernels in the form $K_i(\mathbf{x}, \mathbf{y}) = |S_i(\mathbf{x})|^{-1}$, where $S_i(\mathbf{x})$ is the support of $K_i(\mathbf{x}, \mathbf{y})$ for a given $\mathbf{x}$. In our implementation that means

$$K_2(\tilde{\mathbf{x}}_s, \mathbf{x}_s) = \left(\pi r_{\text{SURF}}^2\right)^{-1}$$
$$K_3(\tilde{\mathbf{x}}_s, \mathbf{x}_s) = \left(\frac{4}{3}\pi r_{\text{P-P3D}}^3\right)^{-1},$$

where r_{SURF} and $r_{\text{P-P3D}}$ are the current radii for the SURF and P-P3D estimators, respectively. These are the `radiusSurf` and `radiusPP3D` values computed at the beginning of each render iteration (on line 13 in Listing 3.12) according to the radius reduction scheme described in Section 3.3.2.

Factors. At the beginning of each iteration also the normalization factors `mSurfNormalization` and `mPP3DNormalization` are computed. We left out the code responsible for it from the initial description of the `RunIteration` method in Section 3.3.2 (on line 16 in Listing 3.12). Here it is:

Listing 3.41: `ComputeFactors` (part of the `RunIteration` method, `UPBP.hxx`)

```
1  const float etaSurf = mPathCount * PI_F * radiusSurf * radiusSurf;
2  const float etaPP3D = mPathCount * (4.0f / 3.0f) * PI_F
3                       * radiusPP3D * radiusPP3D * radiusPP3D;
4  const float etaPB2D = mPathCount * PI_F * radiusPB2D * radiusPB2D;
5  const float etaBB1D = mPathCount * 0.5f * radiusBB1D;
6
7  mSurfNormalization = 1.f / etaSurf;
8  mPP3DNormalization = 1.f / etaPP3D;
9  mPB2DNormalization = 1.f / mPathCount;
10 mBB1DNormalization = 1.f / mPathCount;
11
12 mSurfMisWeightFactor = etaSurf;
13 mPP3DMisWeightFactor = etaPP3D;
14 mPB2DMisWeightFactor = etaPB2D;
15 mBB1DMisWeightFactor = etaBB1D;
```

Besides the normalization factors there are MIS weight factors. While the former ones are used when evaluating the corresponding estimator, i.e. when creating a path with it, the latter ones are needed when computing MIS weights for estimators that could also create the path (see Section 3.5 for details).

3.4.2 P-B2D

We continue with the P-B2D estimator (also called the beam radiance estimate, BRE, as introduced by Jarosz et al. [9]). It evaluates the contribution of stored photon points distributed from light sources over a scene along query beams distributed from the camera.

3.4.2.1 Photon points

Same as with the P-P3D estimator, the photon points are the light vertices stored in media during tracing of light subpaths in the current iteration. For faster photon point lookup along a beam we do not use the hashed grid built for the P-P3D estimator, we instead build an additional bounding volume hierarchy (BVH) over the light vertices. It is implemented in the `EmbreeBre` class (in the

`Bre.hxx` and `Bre.cxx` files) and internally uses Embree [3] for acceleration of the building and traversal of the BVH. The `EmbreeBre` class also encapsulates evaluation of the P-B2D estimator and is represented by the `mPB2DEmbreeBre` object. The build is handled by its `build` method and takes place right after all light subpaths have been traced and before tracing camera subpaths (on line 35 in Listing 3.13). It takes light vertices located in a medium, creates a sphere with radius equal to the kernel radius at each of them (an object of the `EmbreePhoton` class from the `Bre.cxx` file) and lets Embree build its BVH over these spheres.

3.4.2.2 Query beams

The query beams are parts of a traced camera subpath going through a medium, i.e. the volume segments (described in Section 3.2.3.1) found during tracing of the subpath. The evaluation of the estimator takes place at each camera vertex after tracing an outgoing ray (a "query ray"), either when the ray left the scene or when a surface intersection or medium scattering point was found (on lines 15 in Listing 3.26 and 6 in Listing 3.27). Here is the left out code:

Listing 3.42: `EvaluatePB2DIfLeft` (part of the `RunIteration` method, `UPBP.hxx`)

```
1 AdditionalRayDataForMis data(...);
2 const Rgb contrib = mPB2DEmbreeBre.evalBre(
3     mQueryBeamType, ray, mVolumeSegments, mEstimatorTechniques,
4     originInMedium ? AbstractMedium::kOriginInMedium : 0, &data);
5 color += cameraState.mThroughput * mPB2DNormalization * contrib;
```

Listing 3.43: `EvaluatePB2D` (part of the `RunIteration` method, `UPBP.hxx`)

```
1  AdditionalRayDataForMis data(...);
2  Rgb contrib(0);
3  if (isect.IsOnSurface() || mQueryBeamType == SHORT_BEAM)
4      contrib = mPB2DEmbreeBre.evalBre(
5          mQueryBeamType, ray, mVolumeSegments, mEstimatorTechniques,
6          originInMedium ? AbstractMedium::kOriginInMedium : 0, &data);
7  else
8      contrib = mPB2DEmbreeBre.evalBre(
9          mQueryBeamType, ray, mLiteVolumeSegments, mEstimatorTechniques,
10         originInMedium ? AbstractMedium::kOriginInMedium : 0, &data);
11 color += cameraState.mThroughput * mPB2DNormalization * contrib;
```

The code is almost the same in both cases. Firstly, data needed for MIS weight computation are stored in an object of the `AdditionalRayDataForMis` class. This object is then passed to the `evalBre` method of the `mPB2DEmbreeBre` object, which is responsible for evaluation of the estimator.

Long and short query beams. As we explained in Section 1.2.1, there are two types of beams – long and short. However, combining the long- and short-beam variants of the same estimator would not be useful because the long-beam variant always has less variance (as shown by Křivánek et al. [14]). On the other hand, evaluating the long-beam estimators is obviously more costly, so a judicious choice needs to be made. Therefore, our implementation allows users to set the type of query beams which will be then used during the whole rendering. The user choice is stored in the `mQueryBeamType` field (set during construction of the `UPBP` renderer, see Section 3.3.1).

Segments. If a scattering point in a medium was sampled and long query beams are used, then the evaluation is carried out for the `LiteVolumeSegment` structures; otherwise, for the `VolumeSegment` structures. In other words, if no scattering point in a medium was sampled, then the `VolumeSegment` and `LiteVolumeSegment` structures, short and long query beams all denote the same parts of the query ray. Otherwise, short query beams denote the parts of the query ray stored as the `VolumeSegment` structures (they end at a scattering point in a medium if sampled) and long query beams denote the parts stored as the `LiteVolumeSegment` structures. This way the Heaviside step function used in Equation 1.14 is implemented.

Although designation "(volume) segment" and "(query/photon) beam" both refer to the same place in the space, we use the former one in the sense "a piece of a medium" and the latter one in a sense "a (possibly) limited ray".

Evaluation. Here is the aforementioned `evalBre` method:

Listing 3.44: `evalBre` (method, `Bre.cxx`)

```
Rgb evalBre(
    BeamType beamType,
    const Ray& queryRay,
    const VolumeSegments& segments,
    const uint estimatorTechniques,
    const uint raySamplingFlags,
    AdditionalRayDataForMis* additionalRayDataForMis)
{
    Rgb result(0);
    Rgb attenuation(1);
    float raySamplePdf = 1.0f;
    float raySampleRevPdf = 1.0f;

    // For each volume segment do ...
    for (VolumeSegments::const_iterator it = segments.begin();
        it != segments.end(); ++it)
    {
        // Get a medium of the volume segment.
        const AbstractMedium * medium = scene.mMedia[it->mMediumID];

        // Compute contribution along a query beam corresponding
        // to the volume segment.
        Rgb segmentResult(0);
        if (medium->HasScattering())
        {
            ..evalBreContrib..
        }

        // Update the attenuation.
        attenuation *= beamType == SHORT_BEAM ?
            it->mAttenuation / it->mRaySamplePdf
            :
            it->mAttenuation;

        if (!attenuation.isPositive())
            return result;

        // Update pdfs.
        raySamplePdf *= it->mRaySamplePdf;
        raySampleRevPdf *= it->mRaySampleRevPdf;
    }

    return result;
}
```

Listing 3.45: `evalBreContrib` (part of the `evalBre` method, `Bre.cxx`)

```
1  if (additionalRayDataForMis)
2  {
3      additionalRayDataForMis->mRaySamplePdf = raySamplePdf;
4      additionalRayDataForMis->mRaySampleRevPdf = raySampleRevPdf;
5
6      // These flags are used when evaluating the estimator and relate
7      // to the part of the query beam from its beginning to the point of
8      // evaluation. Therefore, the end is always in a medium. The flag
9      // for the beginning is set only for the first query beam based on
10     // the given flags.
11     additionalRayDataForMis->mRaySamplingFlags =
12         AbstractMedium::kEndInMedium;
13     if (it == segments.begin())
14         additionalRayDataForMis->mRaySamplingFlags |= raySamplingFlags;
15 }
16
17 // Create a query beam for the volume segment.
18 embree::Ray queryBeam(
19     toEmbreeV3f(queryRay.origin), toEmbreeV3f(queryRay.direction),
20     it->mDistMin, it->mDistMax);
21
22 // Send Embree data needed for evaluation of the estimator.
23 queryBeam.setAdditionalData(
24     medium, &segmentResult, beamType | estimatorTechniques,
25     &queryRay, additionalRayDataForMis);
26
27 // Intersect the stored photon point spheres by the created
28 // query beam and compute their contribution.
29 embreeIntersector->intersect(queryBeam);
30
31 // Attenuate the contribution computed for the volume segment
32 // and add it to the total result.
33 result += attenuation * segmentResult;
```

The method iterates over the given volume segments along the query ray. For each segment in a medium with scattering it updates `additionalRayDataForMis`, creates a query beam, pass it with other necessary data to Embree and use Embree to find intersections of the beam with the stored photon point spheres and compute their contribution (see below). Contribution of all photon points in the current segment is accumulated in the `segmentResult` field, which is then attenuated by previous segments along the query ray and added to the total result.

Attenuation of photon point contribution consists of attenuation by a medium inside the segment of evaluation and of attenuation by media inside the previous segments along the query ray. The attenuation inside the segment of evaluation is handled when computing the contribution (see below), the attenuation by the previous segments is kept in the `attenuation` field. The `attenuation` field is initialized with 1 (Listing 3.44, line 10) and then updated after each segment (Listing 3.44, line 30). Let there be k segments $s_1, \ldots, s_k$ along the query ray, let d_i denote the length of segment s_i and d'_i the length sampled when tracing the query ray through a medium of segment s_i (tracing and sampling a ray is described in Section 3.2). If long query beams are used, contribution of all photon points along the query ray up to the first active real intersection (or to infinity if there is none) is computed no matter whether any scattering point in a medium was sampled or not, i.e. no ray sampling pdf is considered. Therefore, after segment s_j the `attenuation` field is multiplied simply by $T'_{\mathrm{r}}(d_j)$ becoming $\prod_{i=1}^{j} T'_{\mathrm{r}}(d_i)$. However, if short query beams are used, contribution of a photon point is computed only if the scattering point in a medium was sampled far enough (or was not sampled at all), i.e. a ray sampling pdf must be considered

(as explained in Section 1.2.1). In this case, after segment s_j the `attenuation` field is multiplied by $T'_{\mathrm{r}}(d_j)/\Pr\{d'_j > d_j\}$ becoming $\prod_{i=1}^{j} T'_{\mathrm{r}}(d_i)/\Pr\{d'_i > d_i\}$.

When the `evalBre` method is called with segments as the `VolumeSegment` structures, attenuation and pdfs/probabilities are already precomputed and stored in their `mAttenuation`, `mRaySamplePdf` and `mRaySampleRevPdf` properties. The `mAttenuation` property stores $T'_{\mathrm{r}}(d_j)$ for all segments s_j. The `mRaySamplePdf` property stores probability $\Pr\{d'_j > d_j\}$ for all segments s_j, $j < k$ (we know that a scattering point in a medium was *not* sampled there, since there would not be further segments otherwise) and $\bar{p}(d_k)$ or $\Pr\{d'_k > d_k\}$ for the last segment depending on whether a scattering point was sampled there or not (that may cause different value of the `attenuation` field than described above, but it is not needed after the last segment). The `mRaySampleRevPdf` property stores similar pdfs/probabilities but for a query ray going in the opposite direction. Therefore, it equals $\Pr\{d'_j > d_j\}$ for all segments s_j, $j > 1$ (we know that beginnings of these segments are on media boundary, which corresponds to the opposite query ray leaving them without sampling a scattering point in any of them) and $\bar{p}(d_1)$ or $\Pr\{d'_1 > d_1\}$ for the first segment depending on whether it begins inside a medium or on its boundary (beginning inside a medium corresponds to sampling a scattering point in it while tracing the opposite query ray). When the `evalBre` method is called with segments as the `LiteVolumeSegment` structures, neither the attenuation nor the pdfs/probabilities are stored in them and have to be computed using the `EvalAttenuation` and `RaySamplePdf` methods (shown in Listing 3.4).

Computing contribution. When Embree is called to intersect the stored photon point spheres with a given query beam (on line 29 in Listing 3.45), it traverses its BVH over the spheres and calls the `breIntersectFuncHomogeneous2` method of the `EmbreePhoton` class on encountered spheres. This method tests whether the photon point is close enough to the query beam and if so, evaluates the P-B2D estimator for them:

Listing 3.46: `breIntersectFuncHomogeneous2` (method, `Bre.cxx`)

```
1  void breIntersectFuncHomogeneous2(embree::Intersector1* This, embree::Ray& ray)
2  {
3      const EmbreePhoton* thisPhoton = (const EmbreePhoton*)This;
4      const UPBPLightVertex* lightVertex = thisPhoton->lightVertex;
5
6      // Data needed for the computation. Contains some global renderer variables
7      // and camera subpath state properties.
8      const embree::AdditionalRayDataForMis* data = ray.additionalRayDataForMis;
9
10     float photonIsectDist, isectRadSqr;
11
12     // Test whether the query beam intersects photon disc of this photon point.
13     if (testIntersectionBre(
14         photonIsectDist, isectRadSqr, *ray.origRay, ray.tnear, ray.tfar,
15         lightVertex->mHitpoint, thisPhoton->radiusSqr))
16     {
17         // Reject if the full path length would be above the maximum path length.
18         if (lightVertex->mPathLength + data->mCameraPathLength >
19             data->mMaxPathLength)
20             return;
21
22         ..breIntersectPart1..
23         ..breIntersectPart2..
24     }
25 }
```

Listing 3.47: `breIntersectPart1` (part of the `breIntersectFuncHomogeneous2` method, `Bre.cxx`)

```
// Compute the intersection.
const Pos isectPt = ray.origRay->origin
   + photonIsectDist * ray.origRay->direction;

// Compute attenuation along the beam from its origin to the intersection.
Rgb attenuation =
   ray.medium->EvalAttenuation(photonIsectDist - ray.tnear);
const float pdf =
   attenuation[ray.medium->mMinPositiveAttenuationCoefCompIndex()];
if (ray.flags & SHORT_BEAM)
   attenuation /= pdf;

if (!attenuation.isPositive())
   return;

// Compute ray sampling pdfs in the segment. The direct one
// does not have to test ray sampling flags since we know
// the end is in a medium.
float raySamplePdf =
   ray.medium->mMinPositiveAttenuationCoefComp() * pdf;
float raySampleRevPdf =
   (data->mRaySamplingFlags & AbstractMedium::kOriginInMedium) ?
   raySamplePdf : pdf;

// Multiply pdfs of the current segment with pdfs of previous
// segments to get overall pdfs.
raySamplePdf *= data->mRaySamplePdf;
raySampleRevPdf *= data->mRaySampleRevPdf;

// Ratio of a probability that the beam is sampled long enough
// for intersection and a pdf of sampling a scattering point at
// the intersection. Needed for MIS weight computation.
float raySamplePdfsRatio =
1.0f / ray.medium->mMinPositiveAttenuationCoefComp();

// Retrieve light incoming direction in the world coordinates.
const Dir lightDirection = lightVertex->mBSDF.WorldDirFix();

// Get the scattering coefficient.
const Rgb& scatteringCoeff = ray.medium->GetScatteringCoef();

// Evaluate the scattering function.
float cameraBsdfDirPdfW, cameraBsdfRevPdfW, sinTheta;
const Rgb cameraBsdfFactor = scatteringCoeff * PhaseFunction::Evaluate(
   -ray.dir, lightDirection, ray.medium->MeanCosine(),
   &cameraBsdfDirPdfW, &cameraBsdfRevPdfW, &sinTheta);

if (cameraBsdfFactor.isBlackOrNegative())
   return;

// Complete the direct probability.
cameraBsdfDirPdfW *= ray.medium->ContinuationProb();

// Complete the reverse probability. Even though this is pdf from
// the camera BSDF, the continuation probability must come from
// the light BSDF, because that would govern it if the light path
// actually continued.
cameraBsdfRevPdfW *= lightVertex->mBSDF.ContinuationProb();

// Compute the Epanechnikov kernel.
const float kernel = (1 - isectRadSqr / thisPhoton->radiusSqr)
   / (thisPhoton->radiusSqr * PI_F * 0.5f);

if (!Float::isPositive(kernel))
   return;
```

Listing 3.48: `breIntersectPart2` (part of the `breIntersectFuncHomogeneous2` method, `Bre.cxx`)

```
// Compute an unweighted result.
const Rgb unweightedResult = lightVertex->mThroughput *
    attenuation *
    cameraBsdfFactor *
    kernel;

if (unweightedResult.isBlackOrNegative())
    return;

// Compute a correct MIS weight.
float misWeight = 1.0f;
...PB2DMis...

// Weight and accumulate the result (on the beam).
*static_cast<Rgb*>(ray.accumResult) +=
    misWeight *
    unweightedResult;
```

At first, the `testIntersectionBre` method tests whether the photon point is close enough to the query beam, i.e. whether the query ray the beam lies on intersects a "photon disc" of the photon point sphere (planar cut of the sphere going through its centre perpendicular to the ray) within bounds of the beam. If it does, the method returns a distance of this intersection from the beam origin (`photonIsectDist`) and a squared distance of this intersection from the centre of the sphere (`isectRadSqr`). The evaluation then continues.

Attenuation by a medium along the beam (from its origin to the intersection) is computed, its minimum positive component equals to the probability that the beam is sampled long enough to intersect the photon disc (as shown in Section 3.2.2). As explained above, if using short query beams, the attenuation has to be divided by this probability. After computing ray sampling pdfs/probabilities (according to formulas in Section 3.2.2), the phase function is evaluated using the static `PhaseFunction::Evaluate` method (described in Section 3.2.4). The phase function along the beam is used, i.e. the evaluation is carried out for the medium along the beam, the query ray direction (as incoming) and the incoming direction at the light vertex (as outgoing). The returned pdfs of sampling the phase function in the direct (`cameraBsdfDirPdfW`) and reverse direction (`cameraBsdfRevPdfW`) are needed only for computation of the MIS weight, which comes at the end of the method and is described in Section 3.5. But before that, we evaluate the kernel. Instead of a constant kernel as in the case of the SURF and P-P3D estimators we use the Epanechnikov kernel. Then an unweighted result is computed, it is weighted and accumulated.

Result. After returning from the `evalBre` method the computed contribution is added multiplied by the camera subpath throughput and a normalization factor according to Equation 2.9. Let's consider a photon point at light vertex $\tilde{\mathbf{x}}_s$ contributing in the j-th segment along a query ray of length l_{s+1} going from camera vertex $\mathbf{x}_{s+1}$ creating vertex $\mathbf{x}_s$. Let further $begin_j$ denote a point on the

query ray where the j-th segment begins. Then

$$\begin{aligned}
\texttt{cameraState.mThroughput} &= C_{\mathrm{c}}[\mathbf{x}_{s+1}\dots\mathbf{x}_k),\\
\texttt{mPB2DNormalization} &= \frac{1}{n_{paths}},\\
\texttt{lightVertex->mThroughput} &= C_{\mathrm{l}}(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s),\\
\texttt{cameraBsdfFactor} &= \rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s),\\
\texttt{kernel} &= K_2(\tilde{\mathbf{x}}_s,\mathbf{x}_s).
\end{aligned}$$

If long query beams are used, then

$$\begin{aligned}
\texttt{misWeight} &= \hat{w}_{\text{P-B}_{\mathrm{l}}\text{2D}_s},\\
\texttt{attenuation} &= \prod_{i=1}^{j-1}(T'_{\mathrm{r}}(d_i))T'_{\mathrm{r}}(|begin_j-\mathbf{x}_s|) = T_{\mathrm{r}}(t_{s+1}),
\end{aligned}$$

where we used a definition of T_{r} from Section 3.2.5. An individual contribution of photon point $\tilde{\mathbf{x}}_s$ (as added on line 5 in Listing 3.42 or on line 11 in Listing 3.43) is:

```
color+=cameraState.mThroughput * mPB2DNormalization * misWeight
       * lightVertex->mThroughput * attenuation
       * cameraBsdfFactor * kernel
```

$$\begin{aligned}
&=\frac{1}{n_{paths}}\hat{w}_{\text{P-B}_{\mathrm{l}}\text{2D}_s}C_{\mathrm{l}}(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)K_2(\tilde{\mathbf{x}}_s,\mathbf{x}_s)T_{\mathrm{r}}(t_{s+1})C_{\mathrm{c}}[\mathbf{x}_{s+1}\dots\mathbf{x}_k)\\
&=\frac{1}{n_{paths}}\hat{w}_{\text{P-B}_{\mathrm{l}}\text{2D}_s}L_{\mathrm{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)K_2(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\quad T_{\mathrm{r}}(t_{s+1})\frac{\rho(\mathbf{x}_{s+1})}{p(\omega_{\mathbf{x}_{s+1}})}\frac{T(\mathbf{x}_{s+1}\dots\mathbf{x}_k)}{p(\mathbf{x}_{s+1}\dots\mathbf{x}_k)}W_{\mathrm{e}}(\mathbf{x}_k)\\
&=\frac{1}{n_{paths}}\hat{w}_{\text{P-B}_{\mathrm{l}}\text{2D}_s}L_{\mathrm{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)K_2(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\quad\frac{T(\mathbf{x}_s\dots\mathbf{x}_k)}{G(\mathbf{x}_s,\mathbf{x}_{s+1})p(\omega_{\mathbf{x}_{s+1}})p(\mathbf{x}_{s+1}\dots\mathbf{x}_k)}W_{\mathrm{e}}(\mathbf{x}_k).
\end{aligned}$$

If short query beams are used, then

$$\begin{aligned}
\texttt{misWeight} &= \hat{w}_{\text{P-B}_{\mathrm{s}}\text{2D}_s},\\
\texttt{attenuation} &= \prod_{i=1}^{j-1}\left(\frac{T'_{\mathrm{r}}(d_i)}{\Pr\{d'_i>d_i\}}\right)\frac{T'_{\mathrm{r}}(|begin_j-\mathbf{x}_s|)}{\Pr\{d'_j>|begin_j-\mathbf{x}_s|\}} = \frac{T_{\mathrm{r}}(t_{s+1})}{\Pr\{l_{s+1}>t_{s+1}\}},
\end{aligned}$$

where we used a definition of $\Pr\{l_{s+1}>t_{s+1}\}$ from Section 3.2.5. An individual contribution of photon point $\tilde{\mathbf{x}}_s$ (as added on line 5 in Listing 3.42 or on line 11

in Listing 3.43) is:

$$
\begin{aligned}
&\texttt{color+=cameraState.mThroughput * mPB2DNormalization * misWeight}\\
&\qquad\texttt{* lightVertex->mThroughput * attenuation}\\
&\qquad\texttt{* cameraBsdfFactor * kernel}\\
&=\frac{1}{n_{paths}}\hat{w}_{\text{P-B}_s\text{2D}_s}C_{\text{l}}(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)K_2(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\quad\frac{T_{\text{r}}(t_{s+1})}{\Pr\{l_{s+1}>t_{s+1}\}}C_{\text{c}}[\mathbf{x}_{s+1}\dots\mathbf{x}_k)\\
&=\frac{1}{n_{paths}}\hat{w}_{\text{P-B}_s\text{2D}_s}L_{\text{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)K_2(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\quad\frac{T_{\text{r}}(t_{s+1})}{\Pr\{l_{s+1}>t_{s+1}\}}\frac{\rho(\mathbf{x}_{s+1})}{p(\omega_{\mathbf{x}_{s+1}})}\frac{T(\mathbf{x}_{s+1}\dots\mathbf{x}_k)}{p(\mathbf{x}_{s+1}\dots\mathbf{x}_k)}W_{\text{e}}(\mathbf{x}_k)\\
&=\frac{1}{n_{paths}}\hat{w}_{\text{P-B}_s\text{2D}_s}L_{\text{e}}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0\dots\tilde{\mathbf{x}}_s)}\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)K_2(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\quad\frac{T(\mathbf{x}_s\dots\mathbf{x}_k)}{G(\mathbf{x}_s,\mathbf{x}_{s+1})\Pr\{l_{s+1}>t_{s+1}\}p(\omega_{\mathbf{x}_{s+1}})p(\mathbf{x}_{s+1}\dots\mathbf{x}_k)}W_{\text{e}}(\mathbf{x}_k),
\end{aligned}
$$

which is in line with Equation 1.14.

Kernel and factors. Note that while we used a constant kernel for evaluation of the SURF and P-P3D estimators and so it could be included in normalization factors `mSurfNormalization` and `mPP3DNormalization`, we use the Epanechnikov kernel with the P-B2D estimator, which is dependent on a mutual position of a query beam and a photon point. Therefore, it is computed during evaluation of the estimator and cannot be included in normalization factor `mPB2DNormalization` (see Listing 3.41). However, now we speak about a kernel used when evaluating the P-B2D estimator, i.e. when creating a path with it. When we compute MIS weights for estimators along the path, then the P-B2D estimator is assumed to use a constant kernel too, since we do not have data to compute the Epanechnikov kernel there (we have only a vertex there, no beam, no distance from it). That is why `mPB2DMisWeightFactor` (in Listing 3.41) includes a constant kernel.

3.4.3 B-B1D

Finally, we describe implementation of the B-B1D estimator. It evaluates the contribution of stored photon beams distributed from light sources over a scene along query beams distributed from the camera.

3.4.3.1 Photon beams

The photon beams are parts of a traced light subpath going through a medium, i.e. the volume segments (described in Section 3.2.3.1) found during tracing of the subpath. For each of these segments a photon beam is created and stored in a list using the `AddBeams` method (of the `UPBP` class) which is called every time a new ray of a light subpath (a "photon ray") is traced (on line 36 in Listing 3.14). Note

that similarly to photon points, photon beams are not reused among iterations, i.e. every iteration uses only photon beams created in that iteration.

Long and short photon beams. The distinction between short and long beams introduced in Section 1.2.1 applies not only to query beams but to photon beams as well. And for the same reason as in the case of query beams, our implementation never combines the long and short beams using MIS. Instead, it uses only one type of photon beams during whole rendering based on user settings, i.e. based on a value of the `mPhotonBeamType` field (set in a constructor of the `UPBP` renderer, see Section 3.3.1).

Adding beams. Here is the aforementioned `AddBeams` method creating and storing photon beams:

Listing 3.49: `AddBeams` (method, `UPBP.hxx`)

```
void AddBeams(
    const Ray &aRay,
    const Rgb &aThroughput,
    UPBPLightVertex *aLightVertex,
    const uint aRaySamplingFlags,
    const float aLastPdfWInv)
{
    Rgb throughput = aThroughput;
    float raySamplePdf = 1.0f;
    float raySampleRevPdf = 1.0f;

    if (mPhotonBeamType == SHORT_BEAM)
    {
        // Short photon beams are created from VolumeSegment structures.
        for (VolumeSegments::const_iterator it = mVolumeSegments.cbegin();
            it != mVolumeSegments.cend(); ++it)
        {
            ..AddShortBeam..

            // Update throughput.
            throughput *= it->mAttenuation / it->mRaySamplePdf;

            // Update pdfs.
            raySamplePdf *= it->mRaySamplePdf;
            raySampleRevPdf *= it->mRaySampleRevPdf;
        }
    }
    else
    {
        // Long photon beams are created from LiteVolumeSegment structures.
        for (LiteVolumeSegments::const_iterator it = mLiteVolumeSegments.cbegin();
            it != mLiteVolumeSegments.cend(); ++it)
        {
            ..AddLongBeam..

            // Update throughput.
            throughput *= medium->EvalAttenuation(it->mDistMax - it->mDistMin);

            // Update pdfs.
            float segmentRaySampleRevPdf;
            float segmentRaySamplePdf = beam.mMedium->RaySamplePdf(
                aRay, it->mDistMin, it->mDistMax, it == mLiteVolumeSegments.cbegin()
                ? aRaySamplingFlags : 0, &segmentRaySampleRevPdf);
            raySamplePdf *= segmentRaySamplePdf;
            raySampleRevPdf *= segmentRaySampleRevPdf;
        }
    }
}
```

Listing 3.50: `AddShortBeam` (part of the `AddBeams` method, `UPBP.hxx`)

```
PhotonBeam beam;
beam.mMedium = mScene.mMedia[it->mMediumID];

// Only beams in media with scattering are stored.
if (beam.mMedium->HasScattering())
{
    // Set the beam direction (the direction of the ray the segment lies
    // on), origin (the point on the ray where the segment begins)
    // and length (the length of the segment).
    beam.mRay = Ray(
        aRay.origin + aRay.direction * it->mDistMin, aRay.direction);
        beam.mLength = it->mDistMax - it->mDistMin;

    // Set other beam properties.
    beam.mFlags = SHORT_BEAM;
    beam.mRaySamplePdf = raySamplePdf;
    beam.mRaySampleRevPdf = raySampleRevPdf;
    beam.mLastPdfWInv = aLastPdfWInv;
    beam.mThroughputAtOrigin = throughput;
    beam.mLightVertex = aLightVertex;

    // These flags are used when evaluating the estimator and relate
    // to the part of the beam from its beginning to the point of
    // evaluation. Therefore, the end is always in a medium. The flag
    // for the beginning is set only for the first beam based on
    // the given flags.
    beam.mRaySamplingFlags = AbstractMedium::kEndInMedium;
    if (it == mVolumeSegments.cbegin())
        beam.mRaySamplingFlags |= aRaySamplingFlags;

    // Store the created beam.
    mPhotonBeamsArray.push_back(beam);
}
```

Listing 3.51: `AddLongBeam` (part of the `AddBeams` method, `UPBP.hxx`)

```
PhotonBeam beam;
beam.mMedium = mScene.mMedia[it->mMediumID];

// Only beams in media with scattering are stored.
if (beam.mMedium->HasScattering())
{
    // Set the beam direction (the direction of the ray the segment lies
    // on), origin (the point on the ray where the segment begins)
    // and length (the length of the segment).
    beam.mRay = Ray(
        aRay.origin + aRay.direction * it->mDistMin, aRay.direction);
        beam.mLength = it->mDistMax - it->mDistMin;

    // Set other beam properties.
    beam.mFlags = LONG_BEAM;
    beam.mRaySamplePdf = raySamplePdf;
    beam.mRaySampleRevPdf = raySampleRevPdf;
    beam.mLastPdfWInv = aLastPdfWInv;
    beam.mThroughputAtOrigin = throughput;
    beam.mLightVertex = aLightVertex;

    // These flags are used when evaluating the estimator and relate
    // to the part of the beam from its beginning to the point of
    // evaluation. Therefore, the end is always in a medium. The flag
    // for the beginning is set only for the first beam based on
    // the given flags.
    beam.mRaySamplingFlags = AbstractMedium::kEndInMedium;
    if (it == mLiteVolumeSegments.cbegin())
        beam.mRaySamplingFlags |= aRaySamplingFlags;

    // Store the created beam.
    mPhotonBeamsArray.push_back(beam);
}
```

The method iterates over segments found when tracing the last photon ray. It uses segments of the `VolumeSegment` type for short query beams and of the `LiteVolumeSegment` type for long query beams. For each of the segments which lie in a medium with scattering, the method creates an object of the `PhotonBeam` class (in the `PhotonBeam.hxx` file), sets its properties and stores it in a simple list (`mPhotonBeamsArray`). Between the segments it gradually computes throughput and ray sampling pdfs in a very similar way as the `evalBre` method does (see Listing 3.44). The only difference is that while the `evalBre` method computes only attenuation caused by the processed segments, the `AddBeams` method has to multiply it with throughput of the traced light subpath and store it with beams, since the value is not kept anywhere else.

Acceleration structure. For a faster photon beams lookup along a query beam we build a uniform grid over the photon beams. Building of the grid (as well as evaluation of the estimator) is provided by the `mBB1DPhotonBeams` object of the `PhotonBeamsEvaluator` class (from the `PhBeams.hxx` and `PhBeams.cxx` files). The build is handled by its `build` method and takes place right after all light subpaths have been traced and before tracing camera subpaths (on line 42 in Listing 3.13). It takes the stored photon beams (`mPhotonBeamsArray`), makes a cylinder with radius equal to the kernel radius from each of them and builds a uniform grid over them. The grid itself is implemented in the `Grid` class (in the `Grid.hxx` file) and is composed of cube shaped cells. When it is given the beams, it divides their axis aligned bounding box (AABB) in a dimension of its maximum extent according to user defined resolution. This gives the size of the cells. Resolution in other dimensions is then set to cover the AABB. A pointer to a beam is stored in each cell the cylinder of the beam intersects.

3.4.3.2 Query beams

Description of query beams is almost identical as in the case of the P-B2D estimator. The query beams are parts of a traced camera subpath going through a medium, i.e. the volume segments (described in Section 3.2.3.1) found during tracing of the subpath. The evaluation of the estimator takes place at each camera vertex after tracing an outgoing ray (a "query ray"), either when the ray left the scene or when an "intersection" was found (on lines 12 in Listing 3.27 and 21 in Listing 3.26). Here is the left out code:

Listing 3.52: `EvaluateBB1D` (part of the `RunIteration` method, `UPBP.hxx`)

```
1  AdditionalRayDataForMis data(...);
2  Rgb contrib(0);
3  if (isect.IsOnSurface() || mQueryBeamType == SHORT_BEAM)
4     contrib = mBB1DPhotonBeams.evalBeamBeamEstimate(
5        mQueryBeamType, ray, mVolumeSegments, mEstimatorTechniques,
6        originInMedium ? AbstractMedium::kOriginInMedium : 0, &data);
7  else
8     contrib = mBB1DPhotonBeams.evalBeamBeamEstimate(
9        mQueryBeamType, ray, mLiteVolumeSegments, mEstimatorTechniques,
10       originInMedium ? AbstractMedium::kOriginInMedium : 0, &data);
11 color += cameraState.mThroughput * mBB1DNormalization * contrib;
```

Listing 3.53: `EvaluateBB1DIfLeft` (part of the `RunIteration` method, `UPBP.hxx`)

```
AdditionalRayDataForMis data(...);
const Rgb contrib = mBB1DPhotonBeams.evalBeamBeamEstimate(
    mQueryBeamType, ray, mVolumeSegments, mEstimatorTechniques,
    originInMedium ? AbstractMedium::kOriginInMedium : 0, &data);
color += cameraState.mThroughput * mBB1DNormalization * contrib;
```

The code is almost the same as in Listing 3.43 and 3.42. It differs only by using the `mBB1DNormalization` factor and the `evalBeamBeamEstimate` method, the rest is identical.

Evaluation. The `evalBeamBeamEstimate` method goes through the given segments and lets the grid find intersection of corresponding query beams with the stored photon beams and compute their contribution:

Listing 3.54: `evalBeamBeamEstimate` (method, `PhBeams.cxx`)

```
Rgb evalBeamBeamEstimate(
    BeamType beamType,
    const Ray& queryRay,
    const VolumeSegments& segments,
    const uint estimatorTechniques,
    const uint raySamplingFlags,
    embree::AdditionalRayDataForMis* additionalRayDataForMis)
{
    Rgb result(0);
    Rgb attenuation(1);
    float raySamplePdf = 1.0f;
    float raySampleRevPdf = 1.0f;

    // For each volume segment do ...
    for (VolumeSegments::const_iterator it = segments.begin();
        it != segments.end(); ++it)
    {
        // Get a medium of the volume segment.
        const AbstractMedium * medium = scene.mMedia[it->mMediumID];

        // Compute contribution along a query beam corresponding
        // to the volume segment.
        Rgb segmentResult(0);
        if (medium->HasScattering())
        {
            ..evalBeamBeamContrib..

            // Attenuate the contribution computed for the volume segment
            // and add it to the total result.
            result += attenuation * segmentResult;
        }

        // Update the attenuation.
        attenuation *= beamType == SHORT_BEAM ?
            it->mAttenuation / it->mRaySamplePdf
            :
            it->mAttenuation;

        if (!attenuation.isPositive())
            return result;

        // Update pdfs.
        raySamplePdf *= it->mRaySamplePdf;
        raySampleRevPdf *= it->mRaySampleRevPdf;
    }

    return result;
}
```

Listing 3.55: `evalBeamBeamContrib` (part of the `evalBeamBeamEstimate` method, `PhBeams.cxx`)

```
if (additionalRayDataForMis)
{
   additionalRayDataForMis->mRaySamplePdf = raySamplePdf;
   additionalRayDataForMis->mRaySampleRevPdf = raySampleRevPdf;

   // These flags are used when evaluating the estimator and relate
   // to the part of the query beam from its beginning to the point of
   // evaluation. Therefore, the end is always in a medium. The flag
   // for the beginning is set only for the first query beam based on
   // the given flags.
   additionalRayDataForMis->mRaySamplingFlags =
      AbstractMedium::kEndInMedium;
   if (it == segments.begin())
      additionalRayDataForMis->mRaySamplingFlags |= raySamplingFlags;
}
// Intersect the stored photon beams by a query beam corresponding
// to the volume segment and compute their contribution.
segmentResult = accelStruct->evalBeamBeamEstimate(
   queryRay, beamType | estimatorTechniques, medium,
   it->mDistMin, it->mDistMax, additionalRayDataForMis);
```

This method is also very similar to the corresponding one evaluating the P-B2D estimator (see Listing 3.44 and its description). Instead of Embree the method uses the grid for finding intersections of a query beam with the stored photon beams and compute their contribution. The grid traces the query beam through its cells and calls the `accumulate2` method of each photon beam found there. The query beam for the method is specified via the query ray (`ray`) and distances on it from its origin to the beginning (`mint`) and the end (`maxt`) of the query beam. The photon beam is besides its ray (`mRay`) determined only by its length (`mLength`) since the ray origin equals the photon beam origin (this ray of the photon beam should not be confused with the photon ray, which originates at light vertex and all photon beams lie on it similarly as query beams lie on the query ray).

Computing contribution. The `accumulate2` method tests whether the query beam intersects the photon beam and if so, evaluates the B-B1D estimator for them:

Listing 3.56: `accumulate2` (method, `PhotonBeam.hxx`)

```
void accumulate2(
   const Ray &ray,
   const float mint,
   const float maxt,
   const float isectmint,
   const float isectmaxt,
   Rgb & accumResult,
   uint rayFlags,
   const HomogeneousMedium * medium,
   const AdditionalRayDataForMis* additionalDataForMis = NULL)
{
   float beamBeamDistance, sinTheta, queryIsectDist, beamIsectDist;

   if (mMedium == medium && testIntersectionBeamBeam(
      ray.origin, ray.direction, isectmint, isectmaxt, mRay.origin,
      mRay.direction, 0, mLength, mMaxRadiusSqr, beamBeamDistance,
      sinTheta, queryIsectDist, beamIsectDist)) {
      ..accumulatePart1..
      ..accumulatePart2..
   }
}
```

Listing 3.57: `accumulatePart1` (part of the `accumulate2` method, `PhotonBeam.hxx`)

```
// Reject if the full path length would be above the maximum path length.
if (mLightVertex->mPathLength + 1
   + additionalDataForMis->mCameraPathLength
   > additionalDataForMis->mMaxPathLength)
   return;

// Compute attenuation along the query beam.
Rgb attenuation = medium->EvalAttenuation(queryIsectDist - mint);
const float pfdQuery =
   attenuation[medium->mMinPositiveAttenuationCoefCompIndex()];
if (rayFlags & SHORT_BEAM)
   attenuation /= pfdQuery;

// Compute attenuation along the photon beam.
const Rgb beamAtt = medium->EvalAttenuation(beamIsectDist);
attenuation *= beamAtt;
const float pfdBeam =
   beamAtt[medium->mMinPositiveAttenuationCoefCompIndex()];
if (mFlags & SHORT_BEAM)
   attenuation /= pfdBeam;

if (!attenuation.isPositive())
   return;

// Compute ray sampling pdfs for the query and photon beam.
// The direct ones do not have to test ray sampling flags
// since we know the end is in a medium.
float raySamplePdfQuery =
   medium->mMinPositiveAttenuationCoefComp() * pfdQuery;
float raySampleRevPdfQuery =
   (additionalDataForMis->mRaySamplingFlags &
   AbstractMedium::kOriginInMedium) ? raySamplePdfQuery : pfdQuery;
float raySamplePdfBeam =
   medium->mMinPositiveAttenuationCoefComp() * pfdBeam;
float raySampleRevPdfBeam = (mRaySamplingFlags &
   AbstractMedium::kOriginInMedium) ? raySamplePdfBeam : pfdBeam;

// Multiply pdfs of the query and photon beam with pdfs
// of previous segments along the query a photon ray
// to get overall pdfs.
raySamplePdfQuery *= additionalDataForMis->mRaySamplePdf;
raySampleRevPdfQuery *= additionalDataForMis->mRaySampleRevPdf;
raySamplePdfBeam *= mRaySamplePdf;
raySampleRevPdfBeam *= mRaySampleRevPdf;

// Ratio of a probability that the query/photon beam is sampled
// long enough for intersection and a pdf of sampling a scattering
// point at the intersection. Needed for MIS weight computation.
float raySamplePdfsRatioQuery =
   1.0f / medium->mMinPositiveAttenuationCoefComp();
float raySamplePdfsRatioBeam =
   1.0f / medium->mMinPositiveAttenuationCoefComp();

// Get the scattering coefficient.
const Rgb& scatteringCoeff = medium->GetScatteringCoef();

// Evaluate the scattering function.
float bsdfDirPdfW, bsdfRevPdfW;
const Rgb bsdfFactor = scatteringCoeff * PhaseFunction::Evaluate(
   -ray.direction, -mRay.direction, medium->MeanCosine(),
   &bsdfDirPdfW, &bsdfRevPdfW);

if (bsdfFactor.isBlackOrNegative())
   return;

// Complete the probabilities.
bsdfDirPdfW *= medium->ContinuationProb();
bsdfRevPdfW *= medium->ContinuationProb();
```

Listing 3.58: `accumulatePart2` (part of the `accumulate2` method, `PhotonBeam.hxx`)

```
// Compute the Epanechnikov kernel.
const float kernel =
    (1 - beamBeamDistance * beamBeamDistance / (mRadius * mRadius))
    * 3 / (4 * mRadius*sinTheta);

// Compute an unweighted result.
const Rgb unweightedResult =
    mThroughputAtOrigin *
    attenuation *
    bsdfFactor *
    kernel;

if (unweightedResult.isBlackOrNegative())
    return;

// Compute a correct MIS weight.
float misWeight = 1.0f;
...BB1DMis...

// Weight and accumulate the result.
accumResult +=
    misWeight *
    unweightedResult;
```

At first, the `testIntersectionBeamBeam` method tests whether the query beam intersects the cylinder of the photon beam. Let l_q denote a line the query beam lies on, l_p a line the photon beam lies on, P_q a point on l_q which is closest to l_p and P_p a point on l_p which is closest l_q. The query beam intersects the cylinder if distance $|P_q - P_p|$ is not greater than the kernel radius, point P_p lies on the photon beam (i.e. from 0 to `mLength` on the ray of the photon beam) and point P_q lies on a part of the query beam inside a grid cell where the photon beam was found (i.e. from `isectmint` to `isectmaxt` on the query ray). If all the conditions are met, the method returns distance $|P_q - P_p|$ (`beamBeamDistance`), a distance along the query ray (i.e. from a camera vertex) to point P_q (`queryIsectDist`) and a distance along the photon beam (i.e. from the beam origin) to point P_p (`beamIsectDist`). The evaluation then continues.

Attenuation by a medium along the query beam (from its origin to point P_q) is computed, its minimum positive component equals to the probability the query beam is sampled long enough to intersect the cylinder (as shown in Section 3.2.2). As explained above, if using short query beams, the attenuation has to be divided by this probability. Similarly, attenuation along the photon beam (from its origin to point P_p) is computed and divided by its minimum positive component if short photon beams are used. The attenuation along the query beam and the photon beam are stored multiplied together (in the `attenuation` field). After computing ray sampling pdfs/probabilities for both the query beam and photon beam (according to formulas in Section 3.2.2), the phase function is evaluated using the static `PhaseFunction::Evaluate` method (described in Section 3.2.4). The evaluation is carried out for the medium along the query beam, the query beam direction (as incoming) and the photon beam direction (as outgoing). The returned pdfs of sampling the phase function in the direct (`bsdfDirPdfW`) and reverse direction (`bsdfRevPdfW`) are needed only for computation of the MIS weight, which comes at the end of the method and is described in Section 3.5. But before that, we evaluate the kernel. As in the case of the P-B2D estimator we use the Epanechnikov kernel. Then an unweighted result is computed, it is weighted and accumulated.

Result. After returning from the `testIntersectionBeamBeam` method the computed contribution is added multiplied by the camera subpath throughput and a normalization factor according to Equation 2.12. Let's consider a query ray of length l_{s+1} going from camera vertex $\mathbf{x}_{s+1}$ and a photon ray of length l_{s-1} going from light vertex $\mathbf{x}_{s-1}$. Let the j-th query beam on the query ray intersect cylinder of the k-th photon beam on the photon ray creating vertices $\mathbf{x}_s$ on the query beam and $\tilde{\mathbf{x}}_s$ on the photon beam. Let further $begin_{q,j}$ denote a point on the query ray where the j-th query beam begins and $begin_{p,k}$ a point on the photon ray where the k-th photon beam begins. Then

$$\begin{aligned}
\texttt{cameraState.mThroughput} &= C_{\mathrm{c}}[\mathbf{x}_{s+1} \dots \mathbf{x}_k), \\
\texttt{mBB1DNormalization} &= \frac{1}{n_{paths}}, \\
\texttt{mThroughputAtOrigin} &= C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_{s-1}], \\
\texttt{bsdfFactor} &= \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s), \\
\texttt{kernel} &= \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}}.
\end{aligned}$$

If long query and photon beams are used, then

$$\begin{aligned}
\texttt{misWeight} &= \hat{w}_{\mathrm{B_l\text{-}B_l1D}_s}, \\
\texttt{attenuation} &= \prod_{i=1}^{j-1} (T'_{\mathrm{r}}(d_i)) T'_{\mathrm{r}}(|begin_{q,j} - \mathbf{x}_s|) \\
&\quad \prod_{i=1}^{k-1} (T'_{\mathrm{r}}(d_i)) T'_{\mathrm{r}}(|begin_{p,k} - \tilde{\mathbf{x}}_s|) \\
&= T_{\mathrm{r}}(t_{s-1}) T_{\mathrm{r}}(t_{s+1}),
\end{aligned}$$

where we used a definition of T_{r} from Section 3.2.5. An individual contribution of this pair of query beam and photon beam (as added on line 5 in Listing 3.53 or on line 11 in Listing 3.52) is:

$$\begin{aligned}
&\texttt{color+=cameraState.mThroughput * mBB1DNormalization * misWeight} \\
&\qquad \texttt{* mThroughputAtOrigin * attenuation} \\
&\qquad \texttt{* bsdfFactor * kernel} \\
&= \frac{1}{n_{paths}} \hat{w}_{\mathrm{B_l\text{-}B_l1D}_s} C_{\mathrm{l}}(\mathbf{x}_0 \dots \mathbf{x}_{s-1}] T_{\mathrm{r}}(t_{s-1}) \\
&\quad \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}} T_{\mathrm{r}}(t_{s+1}) C_{\mathrm{c}}[\mathbf{x}_{s+1} \dots \mathbf{x}_k).
\end{aligned}$$

After expanding we get

$$
\begin{aligned}
=&\frac{1}{n_{paths}}\hat{w}_{\mathrm{B_l\text{-}B_l1D}_s}L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\ldots\mathbf{x}_{s-1})}{p(\mathbf{x}_0\ldots\mathbf{x}_{s-1})}\frac{\rho(\mathbf{x}_{s-1})}{p(\omega_{\mathbf{x}_{s-1}})}T_\mathrm{r}(t_{s-1})\\
&\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\frac{K_1(\tilde{\mathbf{x}}_s,\mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}}T_\mathrm{r}(t_{s+1})\frac{\rho(\mathbf{x}_{s+1})}{p(\omega_{\mathbf{x}_{s+1}})}\frac{T(\mathbf{x}_{s+1}\ldots\mathbf{x}_k)}{p(\mathbf{x}_{s+1}\ldots\mathbf{x}_k)}W_\mathrm{e}(\mathbf{x}_k)\\
=&\frac{1}{n_{paths}}\hat{w}_{\mathrm{B_l\text{-}B_l1D}_s}L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\ldots\tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0\ldots\mathbf{x}_{s-1})p(\omega_{\mathbf{x}_{s-1}})G(\mathbf{x}_{s-1},\tilde{\mathbf{x}}_s)}\\
&\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\frac{K_1(\tilde{\mathbf{x}}_s,\mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}}\frac{T(\mathbf{x}_s\ldots\mathbf{x}_k)}{G(\mathbf{x}_s,\mathbf{x}_{s+1})p(\omega_{\mathbf{x}_{s+1}})p(\mathbf{x}_{s+1}\ldots\mathbf{x}_k)}W_\mathrm{e}(\mathbf{x}_k).
\end{aligned}
$$

If short query beams and long photon beams are used, then

$$
\begin{aligned}
\texttt{misWeight} &= \hat{w}_{\mathrm{B_l\text{-}B_s1D}_s},\\
\texttt{attenuation} &= \prod_{i=1}^{j-1}\left(\frac{T'_\mathrm{r}(d_i)}{\Pr\{d'_i>d_i\}}\right)\frac{T'_\mathrm{r}(|begin_{q,j}-\mathbf{x}_s|)}{\Pr\{d'_j>|begin_{q,j}-\mathbf{x}_s|\}}\\
&\quad\prod_{i=1}^{k-1}(T'_\mathrm{r}(d_i))T'_\mathrm{r}(|begin_{p,k}-\tilde{\mathbf{x}}_s|)\\
&= \frac{T_\mathrm{r}(t_{s-1})T_\mathrm{r}(t_{s+1})}{\Pr\{l_{s+1}>t_{s+1}\}},
\end{aligned}
$$

where we used a definition of $\Pr\{l_{s+1}>t_{s+1}\}$ from Section 3.2.5. The individual contribution is:

$$
\begin{aligned}
&\texttt{color+=cameraState.mThroughput * mBB1DNormalization * misWeight}\\
&\qquad\texttt{* mThroughputAtOrigin * attenuation}\\
&\qquad\texttt{* bsdfFactor * kernel}\\
=&\frac{1}{n_{paths}}\hat{w}_{\mathrm{B_l\text{-}B_s1D}_s}C_\mathrm{l}(\mathbf{x}_0\ldots\mathbf{x}_{s-1}]T_\mathrm{r}(t_{s-1})\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\frac{K_1(\tilde{\mathbf{x}}_s,\mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}}\frac{T_\mathrm{r}(t_{s+1})}{\Pr\{l_{s+1}>t_{s+1}\}}C_\mathrm{c}[\mathbf{x}_{s+1}\ldots\mathbf{x}_k)\\
=&\frac{1}{n_{paths}}\hat{w}_{\mathrm{B_l\text{-}B_s1D}_s}L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\ldots\mathbf{x}_{s-1})}{p(\mathbf{x}_0\ldots\mathbf{x}_{s-1})}\frac{\rho(\mathbf{x}_{s-1})}{p(\omega_{\mathbf{x}_{s-1}})}T_\mathrm{r}(t_{s-1})\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\frac{K_1(\tilde{\mathbf{x}}_s,\mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}}\frac{T_\mathrm{r}(t_{s+1})}{\Pr\{l_{s+1}>t_{s+1}\}}\frac{\rho(\mathbf{x}_{s+1})}{p(\omega_{\mathbf{x}_{s+1}})}\frac{T(\mathbf{x}_{s+1}\ldots\mathbf{x}_k)}{p(\mathbf{x}_{s+1}\ldots\mathbf{x}_k)}W_\mathrm{e}(\mathbf{x}_k)\\
=&\frac{1}{n_{paths}}\hat{w}_{\mathrm{B_l\text{-}B_s1D}_s}L_\mathrm{e}(\mathbf{x}_0)\frac{T(\mathbf{x}_0\ldots\tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0\ldots\mathbf{x}_{s-1})p(\omega_{\mathbf{x}_{s-1}})G(\mathbf{x}_{s-1},\tilde{\mathbf{x}}_s)}\rho(\tilde{\mathbf{x}}_s,\mathbf{x}_s)\\
&\frac{K_1(\tilde{\mathbf{x}}_s,\mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}}\frac{T(\mathbf{x}_s\ldots\mathbf{x}_k)}{G(\mathbf{x}_s,\mathbf{x}_{s+1})\Pr\{l_{s+1}>t_{s+1}\}p(\omega_{\mathbf{x}_{s+1}})p(\mathbf{x}_{s+1}\ldots\mathbf{x}_k)}W_\mathrm{e}(\mathbf{x}_k).
\end{aligned}
$$

If long query beams and short photon beams are used, then

$$\begin{aligned}
\texttt{misWeight} &= \hat{w}_{\text{B}_\text{s}\text{-B}_\text{l}1\text{D}_s}, \\
\texttt{attenuation} &= \prod_{i=1}^{j-1} (T'_\text{r}(d_i)) T'_\text{r}(|begin_{q,j} - \mathbf{x}_s|) \\
&\quad \prod_{i=1}^{k-1} \left(\frac{T'_\text{r}(d_i)}{\Pr\{d'_i > d_i\}} \right) \frac{T'_\text{r}(|begin_{p,k} - \tilde{\mathbf{x}}_s|)}{\Pr\{d'_k > begin_{p,k} - \tilde{\mathbf{x}}_s|\}} \\
&= \frac{T_\text{r}(t_{s-1}) T_\text{r}(t_{s+1})}{\Pr\{l_{s-1} > t_{s-1}\}}
\end{aligned}$$

where we used a definition of $\Pr\{l_{s-1} > t_{s-1}\}$ from Section 3.2.5. The individual contribution is:

$$\begin{aligned}
&\texttt{color+=cameraState.mThroughput * mBB1DNormalization * misWeight} \\
&\quad \texttt{* mThroughputAtOrigin * attenuation} \\
&\quad \texttt{* bsdfFactor * kernel} \\
&= \frac{1}{n_{paths}} \hat{w}_{\text{B}_\text{s}\text{-B}_\text{l}1\text{D}_s} C_\text{l}(\mathbf{x}_0 \dots \mathbf{x}_{s-1}] \frac{T_\text{r}(t_{s-1})}{\Pr\{l_{s-1} > t_{s-1}\}} \\
&\quad \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}} T_\text{r}(t_{s+1}) C_\text{c}[\mathbf{x}_{s+1} \dots \mathbf{x}_k) \\
&= \frac{1}{n_{paths}} \hat{w}_{\text{B}_\text{s}\text{-B}_\text{l}1\text{D}_s} L_\text{e}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{x}_{s-1})}{p(\mathbf{x}_0 \dots \mathbf{x}_{s-1})} \frac{\rho(\mathbf{x}_{s-1})}{p(\omega_{\mathbf{x}_{s-1}})} \frac{T_\text{r}(t_{s-1})}{\Pr\{l_{s-1} > t_{s-1}\}} \\
&\quad \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}} T_\text{r}(t_{s+1}) \frac{\rho(\mathbf{x}_{s+1})}{p(\omega_{\mathbf{x}_{s+1}})} \frac{T(\mathbf{x}_{s+1} \dots \mathbf{x}_k)}{p(\mathbf{x}_{s+1} \dots \mathbf{x}_k)} W_\text{e}(\mathbf{x}_k) \\
&= \frac{1}{n_{paths}} \hat{w}_{\text{B}_\text{s}\text{-B}_\text{l}1\text{D}_s} L_\text{e}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0 \dots \mathbf{x}_{s-1}) p(\omega_{\mathbf{x}_{s-1}}) \Pr\{l_{s-1} > t_{s-1}\} G(\mathbf{x}_{s-1}, \tilde{\mathbf{x}}_s)} \\
&\quad \rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}} \frac{T(\mathbf{x}_s \dots \mathbf{x}_k)}{G(\mathbf{x}_s, \mathbf{x}_{s+1}) p(\omega_{\mathbf{x}_{s+1}}) p(\mathbf{x}_{s+1} \dots \mathbf{x}_k)} W_\text{e}(\mathbf{x}_k).
\end{aligned}$$

If short query and photon beams are used, then

$$\begin{aligned}
\texttt{misWeight} &= \hat{w}_{\text{B}_\text{s}\text{-B}_\text{s}1\text{D}_s}, \\
\texttt{attenuation} &= \prod_{i=1}^{j-1} \left(\frac{T'_\text{r}(d_i)}{\Pr\{d'_i > d_i\}} \right) \frac{T'_\text{r}(|begin_{q,j} - \mathbf{x}_s|)}{\Pr\{d'_j > |begin_{q,j} - \mathbf{x}_s|\}} \\
&\quad \prod_{i=1}^{k-1} \left(\frac{T'_\text{r}(d_i)}{\Pr\{d'_i > d_i\}} \right) \frac{T'_\text{r}(|begin_{p,k} - \tilde{\mathbf{x}}_s|)}{\Pr\{d'_k > begin_{p,k} - \tilde{\mathbf{x}}_s|\}} \\
&= \frac{T_\text{r}(t_{s-1}) T_\text{r}(t_{s+1})}{\Pr\{l_{s-1} > t_{s-1}\} \Pr\{l_{s+1} > t_{s+1}\}}
\end{aligned}$$

and the individual contribution is:

```
color+=cameraState.mThroughput * mBB1DNormalization * misWeight
        * mThroughputAtOrigin * attenuation
        * bsdfFactor * kernel
```

$$
\begin{aligned}
=&\frac{1}{n_{paths}}\hat{w}_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}_s} C_\text{l}(\mathbf{x}_0 \dots \mathbf{x}_{s-1}] \frac{T_\text{r}(t_{s-1})}{\Pr\{l_{s-1} > t_{s-1}\}} \\
&\rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}} \frac{T_\text{r}(t_{s+1})}{\Pr\{l_{s+1} > t_{s+1}\}} C_\text{c}[\mathbf{x}_{s+1} \dots \mathbf{x}_k) \\
=&\frac{1}{n_{paths}}\hat{w}_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}_s} L_\text{e}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \mathbf{x}_{s-1})}{p(\mathbf{x}_0 \dots \mathbf{x}_{s-1})} \frac{\rho(\mathbf{x}_{s-1})}{p(\omega_{\mathbf{x}_{s-1}})} \frac{T_\text{r}(t_{s-1})}{\Pr\{l_{s-1} > t_{s-1}\}} \\
&\rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}} \frac{T_\text{r}(t_{s+1})}{\Pr\{l_{s+1} > t_{s+1}\}} \frac{\rho(\mathbf{x}_{s+1})}{p(\omega_{\mathbf{x}_{s+1}})} \frac{T(\mathbf{x}_{s+1} \dots \mathbf{x}_k)}{p(\mathbf{x}_{s+1} \dots \mathbf{x}_k)} W_\text{e}(\mathbf{x}_k) \\
=&\frac{1}{n_{paths}}\hat{w}_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}_s} \\
&L_\text{e}(\mathbf{x}_0) \frac{T(\mathbf{x}_0 \dots \tilde{\mathbf{x}}_s)}{p(\mathbf{x}_0 \dots \mathbf{x}_{s-1}) p(\omega_{\mathbf{x}_{s-1}}) \Pr\{l_{s-1} > t_{s-1}\} G(\mathbf{x}_{s-1}, \tilde{\mathbf{x}}_s)} \\
&\rho(\tilde{\mathbf{x}}_s, \mathbf{x}_s) \frac{K_1(\tilde{\mathbf{x}}_s, \mathbf{x}_s)}{\sin\theta_{\mathbf{x}_{s-1}\mathbf{x}_{s+1}}} \\
&\frac{T(\mathbf{x}_s \dots \mathbf{x}_k)}{G(\mathbf{x}_s, \mathbf{x}_{s+1}) \Pr\{l_{s+1} > t_{s+1}\} p(\omega_{\mathbf{x}_{s+1}}) p(\mathbf{x}_{s+1} \dots \mathbf{x}_k)} W_\text{e}(\mathbf{x}_k).
\end{aligned}
$$

These results are in line with Equations 1.15-1.17.

Kernel and factors. Similar to the P-B2D estimator we use the Epanechnikov kernel with the P-B2D estimator which is dependent on a mutual position of query and photon beams. Therefore, it is computed during evaluation of the estimator and cannot be included in normalization factor `mBB1DNormalization` as in the case of the SURF and P-P3D estimators (see Listing 3.41). Again this holds for a kernel used when evaluating the B-B1D estimator, i.e. when creating a path with it. When we compute MIS weights for estimators along the path, then the B-B1D estimator is assumed to use a constant kernel too, since we do not have data to compute the Epanechnikov kernel there (we have only a vertex there, no beams, no distance between them). That is why `mBB1DMisWeightFactor` (in Listing 3.41) includes a constant kernel.

3.4.4 Data structures

We conclude this section with a brief discussion of used acceleration data structures. For the SURF and P-P3D estimators we needed to quickly find all light vertices stored within a fixed radius from a given camera vertex. A simple grid proved to be sufficient for this purpose, we used the more compact hashed variant already implemented in SmallVCM for the SURF estimator.

For the P-B2D estimator we also needed to query stored light vertices but this time along a query beam. Although the hashed grid could be used for line queries

too we rather kept it simple for point queries and chose a different structure for the P-B2D estimator. Since the search for light vertices within a fixed radius from a query beam can be viewed as intersecting spheres by a ray, we decided to take advantage of the Embree ray tracing kernel. This way we can benefit from its optimized BVH building and ray tracing and only handle intersections with encountered spheres. The efficiency of this implementation is what motivated us to include only the P-B2D estimator and not B-P2D for which this approach is not possible.

Effective implementation of the B-B1D estimator was more problematic as it requires fast look up of photon beams, i.e. lines instead of points. We tried using Embree, implementing different types of kd-tree (with or without chopping beams) and finally ended up with a uniform grid with a photon beam referenced in all cells it intersects. Although the grid performance was the best, it is still far from optimal and consequently the B-B1D estimator is much slower than the others (as shown in Chapter 4). We tried several optimizations, e.g. limiting the maximum number of photon beams in one cell tested for an intersection, but they did not bring any significant improvement (at least not for a usable number of photon beams). Solving this issue is left for future work.

Note that the aforementioned acceleration data structures were implemented mainly by Iliyan Georgiev (the hashed grid) and Martin Šik (the utilization of Embree and the uniform grid).

3.4.5 Summary

The previous lines introduced implementation of the volumetric photon density estimators and described its most important parts in detail. They explained where the estimators are computed and how and what data structures make it possible. Now we have a complete picture of evaluation of estimators. The last thing remaining is to compute their MIS weights.

3.5 MIS weights computation

This section describes one of the key aspects of our work – an algorithm for the computation of MIS weights in the UPBP algorithm. As stated in Chapter 2 we use the balance heuristic (2.3). The pdfs this formula requires are derived in Section 2.1, so we have everything necessary for its evaluation. However, implementing the formula directly, i.e. computing each pdf separately from scratch and then performing the addition and division, would be very inefficient and also prone to arithmetic underflow. We therefore use a different algorithm which overcomes these issues, an extension of the algorithm proposed by Veach [26] for computation of MIS weights in bidirectional path tracing.

3.5.1 Algorithm

We derive the algorithm on an example. Assume a light transport path $\overline{\mathbf{x}}_7 = \mathbf{x}_0 \dots \mathbf{x}_7$ with its first vertex $\mathbf{x}_0$ on a light source, its last vertex $\mathbf{x}_7$ on the camera, and the $\mathbf{x}_1, \dots, \mathbf{x}_6$ scattering vertices on surfaces or in media. Further assume that it was created by the BPT technique connecting vertices $\mathbf{x}_3$ and $\mathbf{x}_4$

and we want to compute the corresponding MIS weight $\hat{w}_{BPT_{3,4}}(\overline{\mathbf{x}}_7)$. The balance heuristic tells us that we should take the pdf of the technique used to create (sample) the path and divide it by the sum of pdfs of all techniques that could possibly create the path. The pdf of the used technique is $p_{\mathrm{BPT}_{3,4}}$ but what are the other techniques that could create the path? As shown in Section 2.2, each of the estimators combined in the UPBP algorithm can be applied on different vertices along a path. It means that path $\overline{\mathbf{x}}_7$ could be created by applying any of estimators P-P3D, P-$\mathrm{B_l}$2D, P-$\mathrm{B_s}$2D, $\mathrm{B_l}$-$\mathrm{B_l}$1D, $\mathrm{B_l}$-$\mathrm{B_s}$1D, $\mathrm{B_s}$-$\mathrm{B_l}$1D, $\mathrm{B_s}$-$\mathrm{B_s}$1D, SURF on vertices $\mathbf{x}_1, \ldots, \mathbf{x}_6$ (we do not make any assumption about the location of these vertices, so we consider both scattering on a surface and in a medium for each of them), BPT on edges $(\mathbf{x}_i, \mathbf{x}_{i+1})$, $i \in \{0, \ldots, 6\}$ or by sampling entirely from the camera (sampling the entire path from the light is not possible since we use an ideal pinhole camera). We get

$$\hat{w}_{BPT_{3,4}} = \frac{p_{\mathrm{BPT}_{3,4}}}{d}, \tag{3.10}$$

where

$$\begin{aligned} d = &\sum_{j=1}^{6} \left(p_{\text{P-P3D}_j} + p_{\text{P-B}_\text{l}\text{2D}_j} + p_{\text{P-B}_\text{s}\text{2D}_j} + p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_j} + p_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}_j} + p_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}_j} + p_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}_j} \right) \\ &+ \sum_{j=1}^{6} p_{\mathrm{SURF}_j} + \sum_{i=0}^{6} p_{\mathrm{BPT}_{i,i+1}} + p_{\mathrm{BPT}_{\mathrm{direct}}}. \end{aligned}$$

The subscript denotes what estimator is applied and on what vertices. For notation brevity we omit the path argument.

Equation 3.10 is a result of direct application of the balance heuristic formula on our example path $\overline{\mathbf{x}}_7$. As we explained, it is not suitable for direct implementation. We rather express the inverted value of the weight and get

$$\frac{1}{\hat{w}_{BPT_{3,4}}} = \frac{d}{p_{\mathrm{BPT}_{3,4}}} = w^L + 1 + w^C, \tag{3.11}$$

where w^L contains terms for vertices along the light subpath (from $\mathbf{x}_0$ to $\mathbf{x}_3$):

$$\begin{aligned} w^L = &\sum_{j=1}^{3} \left(\frac{p_{\text{P-P3D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{P-B}_\text{l}\text{2D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{P-B}_\text{s}\text{2D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} \right. \\ &\left. + \frac{p_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} \right) + \sum_{j=1}^{3} \frac{p_{\mathrm{SURF}_j}}{p_{\mathrm{BPT}_{3,4}}} + \sum_{i=0}^{2} \frac{p_{\mathrm{BPT}_{i,i+1}}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\mathrm{BPT}_{\mathrm{direct}}}}{p_{\mathrm{BPT}_{3,4}}}, \end{aligned}$$

the term 1 is for $\frac{p_{\mathrm{BPT}_{3,4}}}{p_{\mathrm{BPT}_{3,4}}}$ and w^C contains terms for vertices along the camera subpath (from $\mathbf{x}_4$ to $\mathbf{x}_7$):

$$\begin{aligned} w^C = &\sum_{j=4}^{6} \left(\frac{p_{\text{P-P3D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{P-B}_\text{l}\text{2D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{P-B}_\text{s}\text{2D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} + \frac{p_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} \right. \\ &\left. + \frac{p_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}_j}}{p_{\mathrm{BPT}_{3,4}}} \right) + \sum_{j=4}^{6} \frac{p_{\mathrm{SURF}_j}}{p_{\mathrm{BPT}_{3,4}}} + \sum_{i=4}^{6} \frac{p_{\mathrm{BPT}_{i,i+1}}}{p_{\mathrm{BPT}_{3,4}}}. \end{aligned}$$

These formulas are much more useful since a lot of factors will actually vanish and the rest will share a common factor that will gradually change between vertices of the subpaths. We show it by substituting the pdf expressions from Section 2.1.7. Again for notation brevity we use a few simplifications. Let $i \in \{0, \dots, 7\}$, $j \in \{1, \dots, 7\}$, $k \in \{0, \dots, 6\}$. For vertices sampled in the direction from the light we define

$$\begin{aligned}\vec{p}_i &= p(\mathbf{x}_0, \dots, \mathbf{x}_i)\\ \overset{\mapsto}{p}_j &= p(\mathbf{x}_j \,|\, \mathbf{x}_{j-1}),\ \overset{\mapsto}{p}_0 = \vec{p}_0.\end{aligned}$$

Similarly for vertices sampled in the direction from the camera we define

$$\begin{aligned}\overleftarrow{p}_i &= p(\mathbf{x}_i, \dots, \mathbf{x}_7)\\ \overset{\leftarrow}{p}_k &= p(\mathbf{x}_k \,|\, \mathbf{x}_{k+1}),\ \overleftarrow{p}_7 = \overleftarrow{p}_7.\end{aligned}$$

Note that the following equations hold:

$$\begin{aligned}\vec{p}_i &= \prod_{n=0}^{i} \overset{\mapsto}{p}_n\\ \overleftarrow{p}_i &= \prod_{n=i}^{7} \overset{\leftarrow}{p}_n\\ \overset{\mapsto}{p}_j &= p(\omega_{\mathbf{x}_{j-1}}) p(t_{\mathbf{x}_{j-1}}) G(\mathbf{x}_{j-1}, \mathbf{x}_j)\\ \overset{\leftarrow}{p}_k &= p(\omega_{\mathbf{x}_{k+1}}) p(t_{\mathbf{x}_{k+1}}) G(\mathbf{x}_k, \mathbf{x}_{k+1}),\end{aligned}$$

so for example:

$$\frac{\vec{p}_2}{\vec{p}_4} = \frac{\vec{p}_2}{\vec{p}_2 \overset{\mapsto}{p}_3 \overset{\mapsto}{p}_4} = \frac{1}{\overset{\mapsto}{p}_3 \overset{\mapsto}{p}_4}, \qquad \frac{G(\mathbf{x}_4, \mathbf{x}_5) p(\omega_{\mathbf{x}_5})}{\overset{\leftarrow}{p}_4} = \frac{1}{p(t_{\mathbf{x}_5})}.$$

3.5.1.1 Light subpath

We begin with derivations for terms from w^L. For the P-P3D estimator we get:

$$\frac{p_{\text{P-P3D}_3}}{p_{\text{BPT}_{3,4}}} = \frac{\vec{p}_3 \frac{1}{K_3(\mathbf{x}_3, \mathbf{x}_3)} \overleftarrow{p}_3}{\vec{p}_3 \overleftarrow{p}_4} = \frac{1}{K_3(\mathbf{x}_3, \mathbf{x}_3)} \overset{\leftarrow}{p}_3$$

$$\frac{p_{\text{P-P3D}_2}}{p_{\text{BPT}_{3,4}}} = \frac{\vec{p}_2 \frac{1}{K_3(\mathbf{x}_2, \mathbf{x}_2)} \overleftarrow{p}_2}{\vec{p}_3 \overleftarrow{p}_4} = \frac{1}{K_3(\mathbf{x}_2, \mathbf{x}_2)} \frac{\overset{\leftarrow}{p}_2 \overset{\leftarrow}{p}_3}{\overset{\mapsto}{p}_3}$$

$$\frac{p_{\text{P-P3D}_1}}{p_{\text{BPT}_{3,4}}} = \frac{\vec{p}_1 \frac{1}{K_3(\mathbf{x}_1, \mathbf{x}_1)} \overleftarrow{p}_1}{\vec{p}_3 \overleftarrow{p}_4} = \frac{1}{K_3(\mathbf{x}_1, \mathbf{x}_1)} \frac{\overset{\leftarrow}{p}_1 \overset{\leftarrow}{p}_2 \overset{\leftarrow}{p}_3}{\overset{\mapsto}{p}_2 \overset{\mapsto}{p}_3}.$$

For the P-B$_l$2D estimator we get:

$$\frac{p_{\text{P-B}_l\text{2D}_3}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_3}\frac{G(\mathbf{x}_3,\mathbf{x}_4)p(\omega_{\mathbf{x}_4})}{K_2(\mathbf{x}_3,\mathbf{x}_3)}\overleftarrow{p_4}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{G(\mathbf{x}_3,\mathbf{x}_4)p(\omega_{\mathbf{x}_4})}{K_2(\mathbf{x}_3,\mathbf{x}_3)}$$
$$= \frac{1}{p(t_{\mathbf{x}_4})K_2(\mathbf{x}_3,\mathbf{x}_3)}\overleftrightarrow{p_3}$$

$$\frac{p_{\text{P-B}_l\text{2D}_2}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_2}\frac{G(\mathbf{x}_2,\mathbf{x}_3)p(\omega_{\mathbf{x}_3})}{K_2(\mathbf{x}_2,\mathbf{x}_2)}\overleftarrow{p_3}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{G(\mathbf{x}_2,\mathbf{x}_3)p(\omega_{\mathbf{x}_3})}{K_2(\mathbf{x}_2,\mathbf{x}_2)}\frac{\overleftrightarrow{p_3}}{\overrightarrow{p_3}}$$
$$= \frac{1}{p(t_{\mathbf{x}_3})K_2(\mathbf{x}_2,\mathbf{x}_2)}\frac{\overleftrightarrow{p_2}\,\overleftrightarrow{p_3}}{\overrightarrow{p_3}}$$

$$\frac{p_{\text{P-B}_l\text{2D}_1}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_1}\frac{G(\mathbf{x}_1,\mathbf{x}_2)p(\omega_{\mathbf{x}_2})}{K_2(\mathbf{x}_1,\mathbf{x}_1)}\overleftarrow{p_2}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{G(\mathbf{x}_1,\mathbf{x}_2)p(\omega_{\mathbf{x}_2})}{K_2(\mathbf{x}_1,\mathbf{x}_1)}\frac{\overleftrightarrow{p_2}\,\overleftrightarrow{p_3}}{\overrightarrow{p_2}\,\overrightarrow{p_3}}$$
$$= \frac{1}{p(t_{\mathbf{x}_2})K_2(\mathbf{x}_1,\mathbf{x}_1)}\frac{\overleftrightarrow{p_1}\,\overleftrightarrow{p_2}\,\overleftrightarrow{p_3}}{\overrightarrow{p_2}\,\overrightarrow{p_3}}.$$

The short beam version for $i \in \{1,\ldots,3\}$ satisfies

$$\frac{p_{\text{P-B}_s\text{2D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i+1}} > t_{\mathbf{x}_{i+1}}\}\frac{p_{\text{P-B}_l\text{2D}_i}}{p_{\text{BPT}_{3,4}}}.$$

For the B$_l$-B$_l$1D estimator we get:

$$\frac{p_{\text{B}_l\text{-B}_l\text{1D}_3}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_2}\frac{p(\omega_{\mathbf{x}_2})G(\mathbf{x}_2,\mathbf{x}_3)\sin\theta_{\mathbf{x}_2\mathbf{x}_4}G(\mathbf{x}_3,\mathbf{x}_4)p(\omega_{\mathbf{x}_4})}{K_1(\mathbf{x}_3,\mathbf{x}_3)}\overleftarrow{p_4}}{\overrightarrow{p_3}\,\overleftarrow{p_4}}$$
$$= \frac{p(\omega_{\mathbf{x}_2})G(\mathbf{x}_2,\mathbf{x}_3)\sin\theta_{\mathbf{x}_2\mathbf{x}_4}G(\mathbf{x}_3,\mathbf{x}_4)p(\omega_{\mathbf{x}_4})}{K_1(\mathbf{x}_3,\mathbf{x}_3)\overrightarrow{p_3}}$$
$$= \frac{\sin\theta_{\mathbf{x}_2\mathbf{x}_4}}{p(t_{\mathbf{x}_2})p(t_{\mathbf{x}_4})K_1(\mathbf{x}_3,\mathbf{x}_3)}\overleftrightarrow{p_3}$$

$$\frac{p_{\text{B}_l\text{-B}_l\text{1D}_2}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_1}\frac{p(\omega_{\mathbf{x}_1})G(\mathbf{x}_1,\mathbf{x}_2)\sin\theta_{\mathbf{x}_1\mathbf{x}_3}G(\mathbf{x}_2,\mathbf{x}_3)p(\omega_{\mathbf{x}_3})}{K_1(\mathbf{x}_2,\mathbf{x}_2)}\overleftarrow{p_3}}{\overrightarrow{p_3}\,\overleftarrow{p_4}}$$
$$= \frac{p(\omega_{\mathbf{x}_1})G(\mathbf{x}_1,\mathbf{x}_2)\sin\theta_{\mathbf{x}_1\mathbf{x}_3}G(\mathbf{x}_2,\mathbf{x}_3)p(\omega_{\mathbf{x}_3})}{K_1(\mathbf{x}_2,\mathbf{x}_2)\overrightarrow{p_2}}\frac{\overleftrightarrow{p_3}}{\overrightarrow{p_3}}$$
$$= \frac{\sin\theta_{\mathbf{x}_1\mathbf{x}_3}}{p(t_{\mathbf{x}_1})p(t_{\mathbf{x}_3})K_1(\mathbf{x}_2,\mathbf{x}_2)}\frac{\overleftrightarrow{p_2}\,\overleftrightarrow{p_3}}{\overrightarrow{p_3}}$$

$$\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_1}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_0}\frac{p(\omega_{\mathbf{x}_0})G(\mathbf{x}_0,\mathbf{x}_1)\sin\theta_{\mathbf{x}_0\mathbf{x}_2}G(\mathbf{x}_1,\mathbf{x}_2)p(\omega_{\mathbf{x}_2})}{K_1(\mathbf{x}_1,\mathbf{x}_1)}\overleftarrow{p_2}}{\overrightarrow{p_3}\,\overleftarrow{p_4}}$$

$$= \frac{p(\omega_{\mathbf{x}_0})G(\mathbf{x}_0,\mathbf{x}_1)\sin\theta_{\mathbf{x}_0\mathbf{x}_2}G(\mathbf{x}_1,\mathbf{x}_2)p(\omega_{\mathbf{x}_2})}{K_1(\mathbf{x}_1,\mathbf{x}_1)\overrightarrow{p_1}}\frac{\overleftarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_2}\,\overrightarrow{p_3}}$$

$$= \frac{\sin\theta_{\mathbf{x}_0\mathbf{x}_2}}{p(t_{\mathbf{x}_0})p(t_{\mathbf{x}_2})K_1(\mathbf{x}_1,\mathbf{x}_1)}\frac{\overleftarrow{p_1}\,\overleftarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_2}\,\overrightarrow{p_3}}.$$

The short beam versions for $i \in \{1,\ldots,3\}$ satisfy:

$$\frac{p_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i+1}} > t_{\mathbf{x}_{i+1}}\}\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_i}}{p_{\text{BPT}_{3,4}}}$$

$$\frac{p_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i-1}} > t_{\mathbf{x}_{i-1}}\}\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_i}}{p_{\text{BPT}_{3,4}}}$$

$$\frac{p_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i-1}} > t_{\mathbf{x}_{i-1}}\}\Pr\{l_{\mathbf{x}_{i+1}} > t_{\mathbf{x}_{i+1}}\}\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_i}}{p_{\text{BPT}_{3,4}}}.$$

For the SURF estimator we get:

$$\frac{p_{\text{SURF}_3}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_3}\frac{1}{K_2(\mathbf{x}_3,\mathbf{x}_3)}\overleftarrow{p_3}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_2(\mathbf{x}_3,\mathbf{x}_3)}\overleftarrow{p_3}$$

$$\frac{p_{\text{SURF}_2}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_2}\frac{1}{K_2(\mathbf{x}_2,\mathbf{x}_2)}\overleftarrow{p_2}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_2(\mathbf{x}_2,\mathbf{x}_2)}\frac{\overleftarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_3}}$$

$$\frac{p_{\text{SURF}_1}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_1}\frac{1}{K_2(\mathbf{x}_1,\mathbf{x}_1)}\overleftarrow{p_1}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_2(\mathbf{x}_1,\mathbf{x}_1)}\frac{\overleftarrow{p_1}\,\overleftarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_2}\,\overrightarrow{p_3}}.$$

And finally for the BPT estimator we get:

$$\frac{p_{\text{BPT}_{2,3}}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{\overrightarrow{p_3}}\overleftarrow{p_3}$$

$$\frac{p_{\text{BPT}_{1,2}}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_1}\,\overleftarrow{p_2}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{\overrightarrow{p_2}}\frac{\overleftarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_3}}$$

$$\frac{p_{\text{BPT}_{0,1}}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_0}\,\overleftarrow{p_1}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{\overrightarrow{p_1}}\frac{\overleftarrow{p_1}\,\overleftarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_2}\,\overrightarrow{p_3}}$$

$$\frac{p_{\text{BPT}_{\text{direct}}}}{p_{\text{BPT}_{3,4}}} = \frac{\overleftarrow{p_0}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{\overrightarrow{p_0}}\frac{\overleftarrow{p_0}\,\overleftarrow{p_1}\,\overleftarrow{p_2}\,\overleftarrow{p_3}}{\overrightarrow{p_1}\,\overrightarrow{p_2}\,\overrightarrow{p_3}}.$$

3.5.1.2 Camera subpath

Now we go through very similar derivations for terms from w^C. For the P-P3D estimator we get:

$$\frac{p_{\text{P-P3D}_4}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_4}\frac{1}{K_3(\mathbf{x}_4,\mathbf{x}_4)}\overleftarrow{p_4}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_3(\mathbf{x}_4,\mathbf{x}_4)}\overmapsto{p_4}$$

$$\frac{p_{\text{P-P3D}_5}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_5}\frac{1}{K_3(\mathbf{x}_5,\mathbf{x}_5)}\overleftarrow{p_5}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_3(\mathbf{x}_5,\mathbf{x}_5)}\frac{\overmapsto{p_4}\overmapsto{p_5}}{\overleftrightarrow{p_4}}$$

$$\frac{p_{\text{P-P3D}_6}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_6}\frac{1}{K_3(\mathbf{x}_6,\mathbf{x}_6)}\overleftarrow{p_6}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_3(\mathbf{x}_6,\mathbf{x}_6)}\frac{\overmapsto{p_4}\overmapsto{p_5}\overmapsto{p_6}}{\overleftrightarrow{p_4}\,\overleftrightarrow{p_5}}.$$

For the P-B$_\text{l}$2D estimator we get:

$$\begin{aligned}\frac{p_{\text{P-B}_\text{l}\text{2D}_4}}{p_{\text{BPT}_{3,4}}} &= \frac{\overrightarrow{p_4}\frac{G(\mathbf{x}_4,\mathbf{x}_5)p(\omega_{\mathbf{x}_5})}{K_2(\mathbf{x}_4,\mathbf{x}_4)}\overleftarrow{p_5}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{G(\mathbf{x}_4,\mathbf{x}_5)p(\omega_{\mathbf{x}_5})}{K_2(\mathbf{x}_4,\mathbf{x}_4)}\frac{\overmapsto{p_4}}{\overleftrightarrow{p_4}}\\ &= \frac{1}{p(t_{\mathbf{x}_5})K_2(\mathbf{x}_4,\mathbf{x}_4)}\overmapsto{p_4}\end{aligned}$$

$$\begin{aligned}\frac{p_{\text{P-B}_\text{l}\text{2D}_5}}{p_{\text{BPT}_{3,4}}} &= \frac{\overrightarrow{p_5}\frac{G(\mathbf{x}_5,\mathbf{x}_6)p(\omega_{\mathbf{x}_6})}{K_2(\mathbf{x}_5,\mathbf{x}_5)}\overleftarrow{p_6}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{G(\mathbf{x}_5,\mathbf{x}_6)p(\omega_{\mathbf{x}_6})}{K_2(\mathbf{x}_5,\mathbf{x}_5)}\frac{\overmapsto{p_4}\overmapsto{p_5}}{\overleftrightarrow{p_4}\,\overleftrightarrow{p_5}}\\ &= \frac{1}{p(t_{\mathbf{x}_6})K_2(\mathbf{x}_5,\mathbf{x}_5)}\frac{\overmapsto{p_4}\overmapsto{p_5}}{\overleftrightarrow{p_4}}\end{aligned}$$

$$\begin{aligned}\frac{p_{\text{P-B}_\text{l}\text{2D}_6}}{p_{\text{BPT}_{3,4}}} &= \frac{\overrightarrow{p_6}\frac{G(\mathbf{x}_6,\mathbf{x}_7)p(\omega_{\mathbf{x}_7})}{K_2(\mathbf{x}_6,\mathbf{x}_6)}\overleftarrow{p_7}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{G(\mathbf{x}_6,\mathbf{x}_7)p(\omega_{\mathbf{x}_7})}{K_2(\mathbf{x}_6,\mathbf{x}_6)}\frac{\overmapsto{p_4}\overmapsto{p_5}\overmapsto{p_6}}{\overleftrightarrow{p_4}\,\overleftrightarrow{p_5}\,\overleftrightarrow{p_6}}\\ &= \frac{1}{p(t_{\mathbf{x}_7})K_2(\mathbf{x}_6,\mathbf{x}_6)}\frac{\overmapsto{p_4}\overmapsto{p_5}\overmapsto{p_6}}{\overleftrightarrow{p_4}\,\overleftrightarrow{p_5}}.\end{aligned}$$

The short beam version for $i \in \{4,\ldots,6\}$ satisfies

$$\frac{p_{\text{P-B}_\text{s}\text{2D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i+1}} > t_{\mathbf{x}_{i+1}}\}\frac{p_{\text{P-B}_\text{l}\text{2D}_i}}{p_{\text{BPT}_{3,4}}}.$$

For the B$_\text{l}$-B$_\text{l}$1D estimator we get:

$$\begin{aligned}\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}_4}}{p_{\text{BPT}_{3,4}}} &= \frac{\overrightarrow{p_3}\frac{p(\omega_{\mathbf{x}_3})G(\mathbf{x}_3,\mathbf{x}_4)\sin\theta_{\mathbf{x}_3\mathbf{x}_5}G(\mathbf{x}_4,\mathbf{x}_5)p(\omega_{\mathbf{x}_5})}{K_1(\mathbf{x}_4,\mathbf{x}_4)}\overleftarrow{p_5}}{\overrightarrow{p_3}\,\overleftarrow{p_4}}\\ &= \frac{p(\omega_{\mathbf{x}_3})G(\mathbf{x}_3,\mathbf{x}_4)\sin\theta_{\mathbf{x}_3\mathbf{x}_5}G(\mathbf{x}_4,\mathbf{x}_5)p(\omega_{\mathbf{x}_5})}{K_1(\mathbf{x}_4,\mathbf{x}_4)\overleftrightarrow{p_4}}\\ &= \frac{\sin\theta_{\mathbf{x}_3\mathbf{x}_5}}{p(t_{\mathbf{x}_3})p(t_{\mathbf{x}_5})K_1(\mathbf{x}_4,\mathbf{x}_4)}\overmapsto{p_4}\end{aligned}$$

$$\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}1\text{D}_5}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_4}\frac{p(\omega_{\mathbf{x}_4})G(\mathbf{x}_4,\mathbf{x}_5)\sin\theta_{\mathbf{x}_4\mathbf{x}_6}G(\mathbf{x}_5,\mathbf{x}_6)p(\omega_{\mathbf{x}_6})}{K_1(\mathbf{x}_5,\mathbf{x}_5)}\overleftarrow{p_6}}{\overrightarrow{p_3}\,\overleftarrow{p_4}}$$

$$= \frac{p(\omega_{\mathbf{x}_4})G(\mathbf{x}_4,\mathbf{x}_5)\sin\theta_{\mathbf{x}_4\mathbf{x}_6}G(\mathbf{x}_5,\mathbf{x}_6)p(\omega_{\mathbf{x}_6})}{K_1(\mathbf{x}_5,\mathbf{x}_5)\overleftarrow{p_5}}\frac{\overmapsto{p_4}}{\overleftarrow{p_4}}$$

$$= \frac{\sin\theta_{\mathbf{x}_4\mathbf{x}_6}}{p(t_{\mathbf{x}_4})p(t_{\mathbf{x}_6})K_1(\mathbf{x}_5,\mathbf{x}_5)}\frac{\overmapsto{p_4}\,\overmapsto{p_5}}{\overleftarrow{p_4}}$$

$$\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}1\text{D}_6}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_5}\frac{p(\omega_{\mathbf{x}_5})G(\mathbf{x}_5,\mathbf{x}_6)\sin\theta_{\mathbf{x}_5\mathbf{x}_7}G(\mathbf{x}_6,\mathbf{x}_7)p(\omega_{\mathbf{x}_7})}{K_1(\mathbf{x}_6,\mathbf{x}_6)}\overleftarrow{p_7}}{\overrightarrow{p_3}\,\overleftarrow{p_4}}$$

$$= \frac{p(\omega_{\mathbf{x}_5})G(\mathbf{x}_5,\mathbf{x}_6)\sin\theta_{\mathbf{x}_5\mathbf{x}_7}G(\mathbf{x}_6,\mathbf{x}_7)p(\omega_{\mathbf{x}_7})}{K_1(\mathbf{x}_6,\mathbf{x}_6)\overleftarrow{p_6}}\frac{\overmapsto{p_4}\,\overmapsto{p_5}}{\overleftarrow{p_4}\,\overleftarrow{p_5}}$$

$$= \frac{\sin\theta_{\mathbf{x}_5\mathbf{x}_7}}{p(t_{\mathbf{x}_5})p(t_{\mathbf{x}_7})K_1(\mathbf{x}_6,\mathbf{x}_6)}\frac{\overmapsto{p_4}\,\overmapsto{p_5}\,\overmapsto{p_6}}{\overleftarrow{p_4}\,\overleftarrow{p_5}}.$$

The short beam versions for $i \in \{4,\ldots,6\}$ satisfy:

$$\frac{p_{\text{B}_\text{l}\text{-B}_\text{s}1\text{D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i+1}} > t_{\mathbf{x}_{i+1}}\}\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}1\text{D}_i}}{p_{\text{BPT}_{3,4}}}$$

$$\frac{p_{\text{B}_\text{s}\text{-B}_\text{l}1\text{D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i-1}} > t_{\mathbf{x}_{i-1}}\}\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}1\text{D}_i}}{p_{\text{BPT}_{3,4}}}$$

$$\frac{p_{\text{B}_\text{s}\text{-B}_\text{s}1\text{D}_i}}{p_{\text{BPT}_{3,4}}} = \Pr\{l_{\mathbf{x}_{i-1}} > t_{\mathbf{x}_{i-1}}\}\Pr\{l_{\mathbf{x}_{i+1}} > t_{\mathbf{x}_{i+1}}\}\frac{p_{\text{B}_\text{l}\text{-B}_\text{l}1\text{D}_i}}{p_{\text{BPT}_{3,4}}}.$$

For the SURF estimator we get:

$$\frac{p_{\text{SURF}_4}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_4}\frac{1}{K_2(\mathbf{x}_4,\mathbf{x}_4)}\overleftarrow{p_4}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_2(\mathbf{x}_4,\mathbf{x}_4)}\overmapsto{p_4}$$

$$\frac{p_{\text{SURF}_5}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_5}\frac{1}{K_2(\mathbf{x}_5,\mathbf{x}_5)}\overleftarrow{p_5}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_2(\mathbf{x}_5,\mathbf{x}_5)}\frac{\overmapsto{p_4}\,\overmapsto{p_5}}{\overleftarrow{p_4}}$$

$$\frac{p_{\text{SURF}_6}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_6}\frac{1}{K_2(\mathbf{x}_6,\mathbf{x}_6)}\overleftarrow{p_6}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{K_2(\mathbf{x}_6,\mathbf{x}_6)}\frac{\overmapsto{p_4}\,\overmapsto{p_5}\,\overmapsto{p_6}}{\overleftarrow{p_4}\,\overleftarrow{p_5}}.$$

And finally for the BPT estimator we get:

$$\frac{p_{\text{BPT}_{4,5}}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_4}\,\overleftarrow{p_5}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{\overleftarrow{p_4}}\overmapsto{p_4}$$

$$\frac{p_{\text{BPT}_{5,6}}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_5}\,\overleftarrow{p_6}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{\overleftarrow{p_5}}\frac{\overmapsto{p_4}\,\overmapsto{p_5}}{\overleftarrow{p_4}}$$

$$\frac{p_{\text{BPT}_{6,7}}}{p_{\text{BPT}_{3,4}}} = \frac{\overrightarrow{p_6}\,\overleftarrow{p_7}}{\overrightarrow{p_3}\,\overleftarrow{p_4}} = \frac{1}{\overleftarrow{p_6}}\frac{\overmapsto{p_4}\,\overmapsto{p_5}\,\overmapsto{p_6}}{\overleftarrow{p_4}\,\overleftarrow{p_5}}.$$

3.5.1.3 Formalization

We can see that many factors really vanished and the rest show a common structure consisting of an estimator specific factor and a common factor shared among estimators on the same vertex. Also note that the common factor changes gradually as we move between neighbouring vertices on the same subpath. We now formalize this fact, it let us derive a preliminary version of the algorithm for computing the MIS weights.

Let $k \in \mathbb{N}$, $k \geq 2$, $r, s \in \mathbb{R}$, $a, j \in \{1, \ldots, k-1\}$ and $b, i \in \{0, \ldots, k-1\}$ (the bounds will be discussed later). We define

$$R^L_{r,a}(i) = \begin{cases} 0 & \text{if } i > a \\ \overleftarrow{p_i}\, r & \text{if } i = a \\ \frac{\overleftarrow{p_i}}{\overrightarrow{p_{i+1}}} R^L_{r,a}(i+1) & \text{if } i < a \end{cases}$$

$$R^C_{s,b}(i) = \begin{cases} 0 & \text{if } i < b \\ \overrightarrow{p_i}\, s & \text{if } i = b \\ \frac{\overrightarrow{p_i}}{\overleftarrow{p_{i-1}}} R^C_{s,b}(i-1) & \text{if } i > b, \end{cases}$$

i.e. R denotes the common factor shared among estimators on the same vertex. $R^L_{r,a}(i)$ is the common factor for vertex $\mathbf{x}_i$ on the light subpath ending at vertex $\mathbf{x}_a$, $R^C_{s,b}(i)$ is the common factor for vertex $\mathbf{x}_i$ on the camera subpath beginning at vertex $\mathbf{x}_b$, numbers r, s serve for initialization and we explain them later. Figure 3.6 illustrates the definition of the R factor.

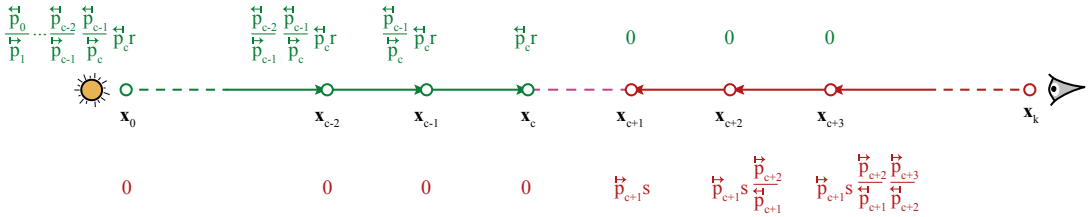


Figure 3.6: Illustration of the R factor definition. It shows a length-k light transport path $\overline{\mathbf{x}} = \mathbf{x}_0 \ldots \mathbf{x}_k$ in the middle. Vertices $\mathbf{x}_0, \ldots, \mathbf{x}_c$ were traced from a light source, i.e. form the light subpath (left, green), vertices $\mathbf{x}_{c+1}, \ldots, \mathbf{x}_k$ were traced from the camera, i.e. form the camera subpath (right, red). Above each vertex corresponding value of $R^L_{r,a}$ for $a = c$ is shown, value of $R^C_{s,b}$ for $b = c+1$ is stated below. Note that neither $R^L_{r,a}$ nor $R^C_{s,b}$ is defined for the $\mathbf{x}_k$ vertex.

The estimator specific factors are labelled as follows:

$$F_{\text{P-P3D}}(j) = \frac{1}{K_3(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F_{\text{P-B}_\text{l}\text{2D}}(j) = \frac{1}{p(t_{\mathbf{x}_{j+1}})K_2(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F_{\text{P-B}_\text{s}\text{2D}}(j) = \frac{\Pr\{l_{\mathbf{x}_{j+1}} > t_{\mathbf{x}_{j+1}}\}}{p(t_{\mathbf{x}_{j+1}})K_2(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(j) = \frac{\sin\theta_{\mathbf{x}_{j-1}\mathbf{x}_{j+1}}}{p(t_{\mathbf{x}_{j-1}})p(t_{\mathbf{x}_{j+1}})K_1(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(j) = \frac{\Pr\{l_{\mathbf{x}_{j+1}} > t_{\mathbf{x}_{j+1}}\}\sin\theta_{\mathbf{x}_{j-1}\mathbf{x}_{j+1}}}{p(t_{\mathbf{x}_{j-1}})p(t_{\mathbf{x}_{j+1}})K_1(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(j) = \frac{\Pr\{l_{\mathbf{x}_{j-1}} > t_{\mathbf{x}_{j-1}}\}\sin\theta_{\mathbf{x}_{j-1}\mathbf{x}_{j+1}}}{p(t_{\mathbf{x}_{j-1}})p(t_{\mathbf{x}_{j+1}})K_1(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(j) = \frac{\Pr\{l_{\mathbf{x}_{j-1}} > t_{\mathbf{x}_{j-1}}\}\Pr\{l_{\mathbf{x}_{j+1}} > t_{\mathbf{x}_{j+1}}\}\sin\theta_{\mathbf{x}_{j-1}\mathbf{x}_{j+1}}}{p(t_{\mathbf{x}_{j-1}})p(t_{\mathbf{x}_{j+1}})K_1(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F_{\text{SURF}}(j) = \frac{1}{K_2(\mathbf{x}_j, \mathbf{x}_j)},$$

$$F^L_{\text{BPT}}(j) = \frac{1}{\overset{\mapsto}{p}_j},$$

$$F^C_{\text{BPT}}(j) = \frac{1}{\overset{\leftarrow}{p}_j}.$$

The whole w^L term can be then rewritten as

$$\begin{aligned} w^L = \sum_{m=3}^{1} R^L_{1,3}(m)\Big(& F_{\text{P-P3D}}(m) + F_{\text{P-B}_\text{l}\text{2D}}(m) + F_{\text{P-B}_\text{s}\text{2D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(m) \\ & + F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{SURF}}(m) \\ & + F^L_{\text{BPT}}(m)\Big) + F^L_{\text{BPT}}(0)R^L_{1,3}(0), \end{aligned} \tag{3.12}$$

and similarly the w^C term as

$$\begin{aligned} w^C = \sum_{m=4}^{6} R^C_{1,4}(m)\Big(& F_{\text{P-P3D}}(m) + F_{\text{P-B}_\text{l}\text{2D}}(m) + F_{\text{P-B}_\text{s}\text{2D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(m) \\ & + F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{SURF}}(m) \\ & + F^C_{\text{BPT}}(m)\Big). \end{aligned} \tag{3.13}$$

These formulas already give us a basic idea how the algorithm for computation of MIS weights can work. It simply makes one pass through light subpath vertices and camera subpath vertices and at each vertex computes the common factor R from the previous one and adds terms for the estimators. The following algorithm computes the $\hat{w}_{BPT_{3,4}}(\overline{\mathbf{x}}_7)$ weight:

Listing 3.59: Algorithm computing the $\hat{w}_{BPT_{3,4}}(\overline{\mathbf{x}}_7)$ weight

$w^L := 0$
$R^L := 1$
for $m := 3$ **downto** 0 **do**
    $R^L := R^L \cdot \overleftarrow{p_m}$
    if $m \neq 0$ **then**
        $w^L := w^L + R^L \cdot (F_{\text{P-P3D}}(m) + F_{\text{P-B}_\text{l}\text{2D}}(m) + F_{\text{P-B}_\text{s}\text{2D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m)$
            $+ F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{SURF}}(m))$
    endif
    $w^L := w^L + R^L \cdot F^L_{\text{BPT}}(m)$
    $R^L := R^L / \overrightarrow{p_m}$
endfor

$w^C := 0$
$R^C := 1$
for $m := 4$ **to** 6 **do**
    $R^C := R^C \cdot \overrightarrow{p_m}$
    $w^C := w^C + R^C \cdot (F_{\text{P-P3D}}(m) + F_{\text{P-B}_\text{l}\text{2D}}(m) + F_{\text{P-B}_\text{s}\text{2D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m)$
        $+ F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{SURF}}(m) + F^C_{\text{BPT}}(m))$
    $R^C := R^C / \overleftarrow{p_m}$
endfor

$\hat{w}_{BPT_{3,4}}(\overline{\mathbf{x}}_7) := 1/(w^L + 1 + w^C)$

3.5.1.4 Generalization

Now we have formulas and an efficient algorithm for computing the MIS weight for paths of length 7 which were created by the BPT technique connecting the third and fourth vertex. We need to generalize them for computing an arbitrary MIS weight. We begin with the w^C term:

- We want to support paths of an arbitrary length $k \in \mathbb{N}$, $k \geq 2$. Note that $k = 0$ is meaningless and for $k = 1$ the path can be created only by a single technique – sampling entirely from the camera, so there is no MIS weighting. Therefore we define k to be at least 2. All we need is to change the upper bound of the sum from 6 to $k - 1$. The sum does not go to k since we can never hit the camera while sampling the path from a light, we cannot evaluate the photon density estimators on it and the BPT connection to the following vertex would not be defined there (there is no vertex $\mathbf{x}_{k+1}$).

- We want to support paths created by other techniques applied on other vertices. So we need to allow beginning of the camera subpath also at other than the fourth vertex. This is accomplished by replacing number 4 with $b \in \{0, \ldots, k - 1\}$. The path could be sampled entirely from the camera, that is why b is defined to be at least 0. The reason for b being at most $k - 1$ is the upper bound of the sum. The situation when the path was created by connecting to the camera and therefore b should equal k is solved independently.

- Not all estimators can be actually applied on every vertex. The volumetric photon density estimators can be applied only on vertices in media, the surface photon density estimator on vertices on surfaces (not purely specular) and the path integral estimators can connect only vertices which are not on purely specular (delta) surfaces. We express this using indicator functions $I_M(b), I_S(b), I_{!\delta}(b, b+1) \in \{0, 1\}$ satisfying:

  $I_M(b) = 1 \Leftrightarrow \mathbf{x}_b$ is in a medium,
  $I_S(b) = 1 \Leftrightarrow \mathbf{x}_b$ is on a surface (not purely specular),
  $I_{!\delta}(b, b+1) = 1 \Leftrightarrow$ neither $\mathbf{x}_b$ nor $\mathbf{x}_{b+1}$ are on a purely specular surface.

  Note that while BPT can connect a vertex on the camera or a light source, the photon density estimators can be applied only on inner path vertices.

The generalized w^C term has the following form:

$$w^C_{k,s,b} = \sum_{m=b}^{k-1} R^C_{s,b}(m) \Bigg(I_M(m) \Big[F_{\text{P-P3D}}(m) + F_{\text{P-B}_\text{l}\text{2D}}(m) + F_{\text{P-B}_\text{s}\text{2D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) \Big] + I_S(m) F_{\text{SURF}}(m) + I_{!\delta}(m, m+1) F^C_{\text{BPT}}(m) \Bigg), \tag{3.14}$$

and can be computed by the following algorithm:

Listing 3.60: Algorithm computing the $w^C_{k,s,b}$ term

```
1  ComputeWC(k, s, b)
2  {
3      w := 0
4      R := s
5      for m := b to k − 1 do
6          R := R · p_m^→
7          w := w+R·I_M(m)·(F_P-P3D(m)+F_P-B_l2D(m)+F_P-B_s2D(m)+F_B_l-B_l1D(m)+F_B_l-B_s1D(m)+F_B_s-B_l1D(m)
8                +F_B_s-B_l1D(m))
9          w := w + R · I_S(m) · F_SURF(m)
10         w := w + R · I_!δ(m, m + 1) · F^C_BPT(m)
11         R := R/p_m^←
12     endfor
13     return w
14 }
```

Now we generalize the w^L formula. We perform similar steps as with the camera subpath:

- To support also other ends of the light subpath than the third vertex we replace number 3 with $a \in \{1, \ldots, k-1\}$. The sum then runs from a down to 1, therefore $a \geq 1$. It does not go to 0, since the photon density estimators cannot be applied on a light source, BPT connection would not be defined there and the case of sampling the entire path from the camera is expressed by the last term outside the sum. The situation when the path was created by connecting the light and therefore a should equal 0 is solved independently. The reason for a being at most $k-1$ is that we can never hit the camera while sampling the path from a light.

- The same indicator functions as in the w^C formula are used but there are two more:

  $$I_{!\delta L}(b) = 1 \Leftrightarrow \mathbf{x}_b \text{ is on a light source that can be hit (is not delta),}$$
  $$I_V(b) = 1 \Leftrightarrow \mathbf{x}_b \text{ is not the last vertex of the light subpath or the path was created by path integral estimators.}$$

  The first indicator is used to eliminate contribution of sampling the entire path from the camera for lights that cannot be hit. The second indicator is necessary to avoid computing contribution of photon density estimators on the last light subpath vertex twice. That could otherwise happen because in the case of creating the path by applying one of the photon density estimators, the last light subpath vertex is also the first camera subpath vertex and the contribution of photon density estimators on this vertex is already computed in $w^C_{k,s,b}$.

The generalized w^L term has the following form:

$$\begin{aligned} w^L_{k,r,a} = \sum_{m=a}^{1} R^L_{r,a}(m) \Bigg(I_V(m) \Big\{ I_M(m) \Big[& F_{\text{P-P3D}}(m) + F_{\text{P-B}_\text{l}\text{2D}}(m) + F_{\text{P-B}_\text{s}\text{2D}}(m) \\ & + F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) + F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(m) \Big] \\ & + I_S(m) F_{\text{SURF}}(m) \Big\} + I_{!\delta}(m-1, m) F^L_{\text{BPT}}(m) \Bigg) \\ & + I_{!\delta L}(0) F^L_{\text{BPT}}(0) R^L_{r,a}(0). \end{aligned} \tag{3.15}$$

and can be computed by the following algorithm:

Listing 3.61: Algorithm computing the $w^L_{k,r,a}$ term

```
ComputeWL(k, r, a)
{
    w := 0
    R := r
    for m := a downto 0 do
        R := R · p_m^↦
        if m ≠ 0 then
            if I_V(m) = 1 then
                w := w + R · I_M(m) · (F_P-P3D(m) + F_P-B_l2D(m) + F_P-B_s2D(m) + F_B_l-B_l1D(m) + F_B_l-B_s1D(m)
                    +F_B_s-B_l1D(m) + F_B_s-B_l1D(m))
                w := w + R · I_S(m) · F_SURF(m)
            endif
            w := w + R · I_!δ(m, m + 1) · F^L_BPT(m)
        endif
        else
            w := w + R · I_!δL(m) · F^L_BPT(m)
        endelse
        R := R/p_m^↤
    endfor
    return w
}
```

Having the generalized terms we now have to figure out how to use them for different techniques of creating path $\overline{\mathbf{x}}_k$ of length k. For the BPT technique connecting vertices $\mathbf{x}_c$, $\mathbf{x}_{c+1}$, $c \in \{1, \ldots, k-2\}$ (case handled by the `ConnectVertices` method 3.36) we already know this. The corresponding weight satisfies

$$\boxed{\frac{1}{\hat{w}_{BPT_{c,c+1}}} = w^L_{k,1,c} + 1 + w^C_{k,1,c+1}.} \tag{3.16}$$

If connecting vertices $\mathbf{x}_0$, $\mathbf{x}_1$ (the `DirectIllumination` method 3.34), the light subpath contains only the $\mathbf{x}_0$ vertex and there is only one technique associated with it – sampling the entire path from the camera. Instead of the w^L term we therefore have only

$$\frac{p_{\mathrm{BPT}_{\mathrm{direct}}}}{p_{\mathrm{BPT}_{0,1}}} = \frac{\overleftarrow{p_0}}{\overrightarrow{p_0}\,\overleftarrow{p_1}} = \frac{\overleftrightarrow{p_0}}{\overrightarrow{\overleftrightarrow{p_0}}}$$

and the corresponding weight satisfies

$$\boxed{\frac{1}{\hat{w}_{BPT_{0,1}}} = I_{!\delta L}(0) F^L_{\mathrm{BPT}}(0) R^L_{1,0}(0) + 1 + w^C_{k,1,1}.} \tag{3.17}$$

On the other hand, if connecting vertices $\mathbf{x}_{k-1}$, $\mathbf{x}_k$ (the `ConnectToCamera` method 3.22), the camera subpath contains only the $\mathbf{x}_k$ vertex and there is no technique associated with it (we cannot sample the entire path from the light since we cannot hit our ideal pinhole camera). Therefore there is no w^C term and the corresponding weight satisfies

$$\boxed{\frac{1}{\hat{w}_{BPT_{k-1,k}}} = w^L_{k,1,k-1} + 1.} \tag{3.18}$$

If the entire path was sampled from the camera (the `GetLightRadiance` method 3.33), then there is simply no w^L term and the corresponding weight satisfies

$$\boxed{\frac{1}{\hat{w}_{\mathrm{BPT}_{\mathrm{direct}}}} = 1 + w^C_{k,1,0}.} \tag{3.19}$$

Finally, let the path be created by applying any of the photon density estimators on vertex $\mathbf{x}_e$, $e \in \{1, \ldots, k-1\}$. In contrast to the path integral estimators, the light subpath and camera subpath now have a common vertex – the $\mathbf{x}_e$ vertex. We have to ensure that the terms of the photon density estimators on this vertex are not computed twice. As already mentioned, we accomplish this using the I_V indicator in the w^L term. Note that it does not apply to the terms of the path integral estimator as they are computed for edges which are not shared. If we went through the same derivation process for each of the photon density estimators as we did for the path integral estimator, we would see that estimator specific factors are still the same, as well as the way the common factor changes. The only difference is how the common factor is initialized. It is initialized exactly the way needed to produce 1 on the $\mathbf{x}_e$ vertex when computing the term of the photon density estimator which created the path (on both subpaths no matter whether or not it is actually computed). Let *pde* denote the photon density estimator which created the path, i.e. $pde \in \{\text{P-P3D}, \text{P-B}_\text{l}\text{2D}, \text{P-B}_\text{s}\text{2D}, \text{B}_\text{l}\text{-B}_\text{l}\text{1D}, \text{B}_\text{l}\text{-B}_\text{s}\text{1D}, \text{B}_\text{s}\text{-B}_\text{l}\text{1D}, \text{B}_\text{s}\text{-B}_\text{s}\text{1D}, \text{SURF}\}$, and define initialization factors:

$$IF^L_{pde}(e) = \frac{1}{\overleftarrow{p_e}\, F_{pde}(e)}$$
$$IF^C_{pde}(e) = \frac{1}{\overrightarrow{p_e}\, F_{pde}(e)}.$$

Then the resulting weight has the form:

$$\boxed{\frac{1}{\hat{w}_{pde_e}} = w^L_{k,r,e} + w^C_{k,s,e}, \qquad r = IF^L_{pde}(e),\ s = IF^C_{pde}(e).} \tag{3.20}$$

Let's check if it works. Indeed, for a given *pde* the corresponding terms in the sums in $w^L_{k,r,e}$ and $w^C_{k,s,e}$ satisfy:

$$R^L_{r,e}(e)F_{pde}(e) = \overleftarrow{p}_e\, rF_{pde}(e) = \overleftarrow{p}_e\, IF^L_{pde}(e)F_{pde}(e) = \overleftarrow{p}_e \frac{1}{\overleftarrow{p}_e\, F_{pde}(e)} F_{pde}(e) = 1$$
$$R^C_{s,e}(e)F_{pde}(e) = \overrightarrow{p}_e\, sF_{pde}(e) = \overrightarrow{p}_e\, IF^C_{pde}(e)F_{pde}(e) = \overrightarrow{p}_e \frac{1}{\overrightarrow{p}_e\, F_{pde}(e)} F_{pde}(e) = 1.$$

Summary. The final algorithm is then simple: based on the estimator that created the path, pick one of Equations 3.16-3.20 and evaluate it using the algorithms 3.60 and 3.61. This algorithm is very efficient, it performs only one pass over path vertices and only a constant time operation for each vertex, i.e. it runs in $O(k)$ for a k-length path. Furthermore, since it computes with quotients of pdfs, it does not suffer from the arithmetic underflow.

Previous work. As we mentioned at the beginning, our algorithm is an extension of the algorithm proposed by Veach [26] for computation of MIS weights in bidirectional path tracing. His version is limited to scenes without media and paths created only by BPT techniques, but the basic algorithm is the same. We extended it with pdfs of sampling through media and terms for photon density estimators. See Section 10.2 in [26], mainly Equation 10.9.

3.5.2 Implementation

Now we describe how the algorithm for computation of MIS weights derived above is implemented. We begin with data the algorithm requires. Most of the data are stored in the `MisData` structures:

Listing 3.62: `MisData` (struct, `UPBPLightVertex.hxx`)

```
1  struct MisData
2  {
3      float mPdfInv;
4      float mRevPdf;
5      float mRevPdfWithoutBsdf;
6      float mRaySamplePdfInv;
7      float mRaySampleRevPdfInv;
8      float mRaySamplePdfsRatio;
9      float mRaySampleRevPdfsRatio;
10     float mSinTheta;
11     float mCosThetaOut;
12     float mSurfMisWeightFactor;
13     float mPP3DMisWeightFactor;
14     float mPB2DMisWeightFactor;
15     float mBB1DMisWeightFactor;
16     bool  mIsDelta;
17     bool  mIsOnLightSource;
18     bool  mIsSpecular;
19 }
```

In order to compute the MIS weight of a path, we need these data for its every vertex. We therefore store one `MisData` structure with every light vertex of every light subpath (inside its `UPBPLightVertex` object, see Listing 3.18) and one for every camera vertex on a currently traced camera subpath (inside the static `mCameraVerticesMisData` array). This asymmetry is caused by the fact that while almost any light vertex can be used for estimator evaluation, camera vertices of previously traced camera subpaths are never needed again.

3.5.2.1 Light vertices MIS data

Consider light subpath $\mathbf{x}_0 \dots \mathbf{x}_s$ and one of its vertices $\mathbf{x}_i$. Fields of the `MisData` structure stored with $\mathbf{x}_i$ have the following values:

`mPdfInv` Equals $\frac{1}{\overrightarrow{p_i}}$.

`mRevPdf` If $i = 0$, it equals:

0	for point and directional light sources
$D(\mathbf{x}_i \rightarrow)^2$	for area and background light sources if $s = i$
$D(\mathbf{x}_i \rightarrow)\frac{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})^3}{\lvert\mathbf{x}_i - \mathbf{x}_{i+1}\rvert^2}$	for area and background light sources if $s = i + 1$ and the scattering function of vertex $\mathbf{x}_{i+1}$ was not sampled
$\overleftarrow{p_i}$	otherwise.

If $i > 0$, it equals:

1	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was not sampled
$D(\mathbf{x}_i \rightarrow)$	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was sampled
$D(\mathbf{x}_i \rightarrow)\frac{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})}{\lvert\mathbf{x}_i - \mathbf{x}_{i+1}\rvert^2}$	if $s = i + 1$ and the scattering function of vertex $\mathbf{x}_{i+1}$ was not sampled
$\overleftarrow{p_i}$	otherwise.

`mRevPdfWithoutBsdf` Almost the same as the `mRevPdf` field but lacks probability of sampling the scattering function. That means, if $i = 0$, it equals:

0	for point and directional light sources
$D(\mathbf{x}_i \rightarrow)$	for area and background light sources if $s = i$
$D(\mathbf{x}_i \rightarrow)\frac{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})}{\lvert\mathbf{x}_i - \mathbf{x}_{i+1}\rvert^2}$	for area and background light sources if $s > i$.

If $i > 0$, it equals:

1	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was not sampled
$D(\mathbf{x}_i \rightarrow)$	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was sampled
$D(\mathbf{x}_i \rightarrow)\frac{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})}{\lvert\mathbf{x}_i - \mathbf{x}_{i+1}\rvert^2}$	otherwise.

`mRaySamplePdfInv` Equals $\frac{1}{p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})}$ if $i > 0$. Otherwise, it is zero.

`mRaySampleRevPdfInv` Equals $\frac{1}{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})}$ if $i < s$. Otherwise, it is one.

`mRaySamplePdfsRatio` Equals $\frac{\Pr\{t>t_{\mathbf{x}_{i-1}\mathbf{x}_i}\}}{p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})}$ if $i > 0$ and $\mathbf{x}_i$ is in a medium. Otherwise, it is zero.

`mRaySampleRevPdfsRatio` Equals $\frac{\Pr\{t>t_{\mathbf{x}_{i+1}\mathbf{x}_i}\}}{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})}$ if $i > 0$ and $\mathbf{x}_i$ is in a medium (condition $i < s$ is not required to be satisfied since the ratio can be computed anyway). Otherwise, it is zero.

`mSinTheta` Equals $\sin\theta_{\mathbf{x}_i}$ if $i > 0$, the vertex is located in a medium and its phase function was sampled. Otherwise, it is zero.

`mCosThetaOut` If $i > 0$ and the scattering function of the vertex was sampled or if $i = 0$, then it equals $D(\mathbf{x}_i \rightarrow)$. Otherwise, it is zero.

`mSurfMisWeightFactor` Equals $\frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if $i > 0$ and the vertex is located on a surface. Otherwise, it is zero.

`mPP3DMisWeightFactor` Equals $\frac{n_{paths}}{K_3(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if $i > 0$ and the vertex is located in a medium. Otherwise, it is zero.

`mPB2DMisWeightFactor` Equals $\frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if $i > 0$ and the vertex is located in a medium. Otherwise, it is zero.

`mBB1DMisWeightFactor` Equals $\frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if $i > 0$ and the vertex is located in a medium. Otherwise, it is zero.

`mIsDelta` Equals `true` if $i = 0$ and the light source that generated the vertex is point or directional. It is also `true` if $i > 0$ and the vertex is located on a purely specular surface. Otherwise, it is `false`.

`mIsOnLightSource` Equals `true` if $i = 0$. Otherwise, it is `false`.

`mIsSpecular` Equals `true` if $i > 0$, the vertex is located on a surface and a specular component of its material BSDF was sampled. Otherwise, it is `false`.

These values are gathered and computed during tracing the light subpath on several places. Firstly, when the first light vertex is created (i.e. $\mathbf{x}_0$, $i = s = 0$) in the `GenerateLightSample` method (on line 9 in Listing 3.19). The code is presented in the following Listing 3.63. Note that `directPdfA` is a pdf of sampling a light source, i.e. exactly $p(\mathbf{x}_0) = \overset{\mapsto}{p}_0$ as needed, and `cosLight` equals $D(\mathbf{x}_i \rightarrow)$ (these values are computed earlier in the `GenerateLightSample` method, see Listing 3.17). The `IsDelta` method returns true only for point and directional light sources, the `IsFinite` method returns true only for point and area light sources.

[2] $D(\mathbf{x}_i \rightarrow) = |n_{\mathbf{x}_i} \cdot \omega_{\mathbf{x}_i \rightarrow}|$, see Section 3.2.1

[3] $t_{\mathbf{x}_{i+1}\mathbf{x}_i}$ is a distance between the vertices $\mathbf{x}_{i+1}$ and $\mathbf{x}_i$, considered as sampled in the direction from the camera, see Section 1.2

Listing 3.63: `SetFirstLightVertexMisData` (part of the `GenerateLightSample` method, `UPBP.hxx`)

```
// Set pdfs.
lightVertex.mMisData.mPdfInv = 1.0f / directPdfA;
lightVertex.mMisData.mRevPdf = light->IsDelta() ?
   0.0f : (light->IsFinite() ? cosLight : 1.f);
lightVertex.mMisData.mRevPdfWithoutBsdf = lightVertex.mMisData.mRevPdf;
lightVertex.mMisData.mRaySamplePdfInv = 0.0f;
lightVertex.mMisData.mRaySampleRevPdfInv = 1.0f;
lightVertex.mMisData.mRaySamplePdfsRatio = 0.0f;
lightVertex.mMisData.mRaySampleRevPdfsRatio = 0.0f;
// Set angles.
lightVertex.mMisData.mSinTheta = 0.0f;
lightVertex.mMisData.mCosThetaOut = (!light->IsDelta() && light->IsFinite()) ?
   cosLight : 1.f;
// Set MIS weight factors.
lightVertex.mMisData.mSurfMisWeightFactor = 0.0f;
lightVertex.mMisData.mPP3DMisWeightFactor = 0.0f;
lightVertex.mMisData.mPB2DMisWeightFactor = 0.0f;
lightVertex.mMisData.mBB1DMisWeightFactor = 0.0f;
// Set flags.
lightVertex.mMisData.mIsDelta = light->IsDelta();
lightVertex.mMisData.mIsOnLightSource = true;
lightVertex.mMisData.mIsSpecular = false;
```

Listing 3.64: `SetLightVertexMisData` (part of the `RunIteration` method, `UPBP.hxx`)

```
// Infinite lights use MIS handled via solid angle integration, so we do not
// divide by the distance for such lights.
const float distSq =
   (lightState.mPathLength > 1 || lightState.mIsFiniteLight == 1) ?
   Utils::sqr(isect.mDist) : 1.0f;
const float raySamplePdfInv = 1.0f / raySamplePdf;
// Set pdfs.
lightVertex.mMisData.mPdfInv =
   lightState.mLastPdfWInv * distSq * raySamplePdfInv
   / std::abs(bsdf.CosThetaFix());
lightVertex.mMisData.mRevPdf = 1.0f;
lightVertex.mMisData.mRevPdfWithoutBsdf = lightVertex.mMisData.mRevPdf;
lightVertex.mMisData.mRaySamplePdfInv = raySamplePdfInv;
lightVertex.mMisData.mRaySampleRevPdfInv = 1.0f;
lightVertex.mMisData.mRaySamplePdfsRatio = bsdf.IsInMedium() ? 1.0f /
   bsdf.GetMedium()->mMinPositiveAttenuationCoefComp() : 0.0f;
lightVertex.mMisData.mRaySampleRevPdfsRatio =
   lightVertex.mMisData.mRaySamplePdfsRatio;
// Set angles.
lightVertex.mMisData.mSinTheta = 0.0f;
lightVertex.mMisData.mCosThetaOut = 0.0f;
// Set MIS weight factors.
lightVertex.mMisData.mSurfMisWeightFactor =
   bsdf.IsOnSurface() ? mSurfMisWeightFactor : 0;
lightVertex.mMisData.mPP3DMisWeightFactor =
   bsdf.IsOnSurface() ? 0 : mPP3DMisWeightFactor;
lightVertex.mMisData.mPB2DMisWeightFactor =
   bsdf.IsOnSurface() ? 0 : mPB2DMisWeightFactor;
lightVertex.mMisData.mBB1DMisWeightFactor =
   bsdf.IsOnSurface() ? 0 : mBB1DMisWeightFactor;
// Set flags.
lightVertex.mMisData.mIsDelta = bsdf.IsDelta();
lightVertex.mMisData.mIsOnLightSource = false;
lightVertex.mMisData.mIsSpecular = false;
// Update reverse pdfs of the previous vertex.
mLightVertices.back().mMisData.mRevPdf *= raySampleRevPdf / distSq;
mLightVertices.back().mMisData.mRevPdfWithoutBsdf =
   mLightVertices.back().mMisData.mRevPdf;
mLightVertices.back().mMisData.mRaySampleRevPdfInv =
   1.0f / raySampleRevPdf;
```

`MisData` are also computed when a new light vertex of the subpath is created (i.e. $\mathbf{x}_i$, $i = s > 0$) after tracing a ray in the `RunIteration` method (on line 10 in Listing 3.21). The code is presented in the Listing 3.64 above. Again we briefly describe it. The `raySamplePdf` field contains $p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})$ and is computed earlier in the `RunIteration` method when attenuating the subpath throughput by intersected media (see Listing 3.20). The absolute value of `bsdf.CosThetaFix()` equals $D(\mathbf{x}_i \to \mathbf{x}_{i-1})$ and `lightState.mLastPdfWInv` equals $1/\hat{p}(\omega_{\mathbf{x}_{i-1}\mathbf{x}_i})$ (it is computed at the previous vertex either in method `GenerateLightSample` or `SampleScattering`, see Listing 3.17 or Listing 3.24, respectively). Therefore:

$$\begin{aligned}
\texttt{mPdfInv} =& \texttt{lightState.mLastPdfWInv * distSq * raySamplePdfInv} \\
& \texttt{/ std::abs(bsdf.CosThetaFix())} \\
=& \frac{|\mathbf{x}_{i-1} - \mathbf{x}_i|^2}{\hat{p}(\omega_{\mathbf{x}_{i-1}\mathbf{x}_i}) p(t_{\mathbf{x}_{i-1}\mathbf{x}_i}) D(\mathbf{x}_i \to \mathbf{x}_{i-1})} \\
=& \frac{|\mathbf{x}_{i-1} - \mathbf{x}_i|^2}{p(\omega_{\mathbf{x}_{i-1}\mathbf{x}_i}) p(t_{\mathbf{x}_{i-1}\mathbf{x}_i}) D(\mathbf{x}_i \to \mathbf{x}_{i-1}) D(\mathbf{x}_{i-1} \to \mathbf{x}_i)} \\
=& \frac{1}{p(\mathbf{x}_i \,|\, \mathbf{x}_{i-1})} = \frac{1}{\overset{\mapsto}{p}_i}.
\end{aligned}$$

Pdfs in the reverse direction as well as $\sin\theta_{\mathbf{x}_i}$ and $D(\mathbf{x}_i \to)$ cannot be computed yet since the subpath does not continue yet. The `mRaySamplePdfsRatio` value is computed using the formula

$$\begin{aligned}
\frac{\Pr\{t > t_{\mathbf{x}_{i-1}\mathbf{x}_i}\}}{p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})} =& \frac{\prod_{j=1}^{n} \Pr\{d'_j > d_j\}}{\left(\prod_{j=1}^{n-1} \Pr\{d'_j > d_j\}\right) \bar{p}(d_n)} = \frac{\Pr\{d'_n > d_n\}}{\bar{p}(d_n)} \\
=& \frac{T'_{\mathrm{r,m}}(d_n)}{\sigma_{\mathrm{t,m}} T'_{\mathrm{r,m}}(d_n)} = \frac{1}{\sigma_{\mathrm{t,m}}},
\end{aligned}$$

where we used definitions from Section 3.2.5. Since the result is always a constant independent on a direction, `mRaySampleRevPdfsRatio` has the same value.

The MIS weight factors are used when we evaluate an estimator, i.e. create a path, and need to compute MIS weights for estimators that could also create it. Then we assume that all estimators use a constant kernel because it has to be computed only using vertices along the path. The MIS weight factors are consequently constant and can be computed at the beginning of each iteration (see Listing 3.41).

Once we have the new light vertex we can update reverse pdfs of the previous one with a length and ray sampling pdf of the connecting edge between them (the last lines in Listing 3.64).

Finally, the data are updated when sampling the scattering function at the vertex in the `SampleScattering` method (on line 53 in Listing 3.24):

Listing 3.65: `SampleScatteringMis` (part of the `SampleScattering` method, `UPBP.hxx`)

```
1 aoCurrentMisData.mRevPdf *= cosThetaOut;
2 aoCurrentMisData.mRevPdfWithoutBsdf = aoCurrentMisData.mRevPdfA;
3 aoCurrentMisData.mIsSpecular = specular;
4 aoCurrentMisData.mSinTheta = sinTheta;
5 aoCurrentMisData.mCosThetaOut = cosThetaOut;
6 aoPreviousMisData.mRevPdf *= bsdfRevPdfW;
```

Data of the current vertex are updated by the newly obtained angles, `sinTheta` equals $\sin\theta_{\mathbf{x}_i}$ for vertices in media, 0 otherwise, `cosThetaOut` equals $D(\mathbf{x}_i \to)$. The reverse pdf of the previous vertex is completed with $\hat{p}(\omega_{\mathbf{x}_i\mathbf{x}_{i-1}})$.

3.5.2.2 Camera vertices MIS data

Data for camera vertices are analogous. Assume camera subpath $\mathbf{x}_s \dots \mathbf{x}_k$ and one of its vertices $\mathbf{x}_i$, $i < k$ (although the data are stored for the camera vertex too, they are never used). Fields of the `MisData` structure stored for $\mathbf{x}_i$ have the following values:

`mPdfInv` Equals $\frac{1}{\overleftarrow{p_i}}$.

`mRevPdf` Equals:

1	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was not sampled
$D(\leftarrow\mathbf{x}_i)$[4]	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was sampled
$D(\leftarrow\mathbf{x}_i)\frac{p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})^5}{\lvert\mathbf{x}_i-\mathbf{x}_{i-1}\rvert^2}$	if $s = i-1$ and the scattering function of vertex $\mathbf{x}_{i-1}$ was not sampled
$\overrightarrow{p_i}$	otherwise.

`mRevPdfWithoutBsdf` Almost the same as the `mRevPdf` field but lacks probability of sampling the scattering function. That means, it equals:

1	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was not sampled
$D(\leftarrow\mathbf{x}_i)$	if $s = i$ and the scattering function of vertex $\mathbf{x}_i$ was sampled
$D(\leftarrow\mathbf{x}_i)\frac{p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})}{\lvert\mathbf{x}_i-\mathbf{x}_{i-1}\rvert^2}$	otherwise.

`mRaySamplePdfInv` Equals $\frac{1}{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})}$.

`mRaySampleRevPdfInv` Equals $\frac{1}{p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})}$ if $i > s$. Otherwise, it is one.

`mRaySamplePdfsRatio` Equals $\frac{\Pr\{t>t_{\mathbf{x}_{i+1}\mathbf{x}_i}\}}{p(t_{\mathbf{x}_{i+1}\mathbf{x}_i})}$ if $\mathbf{x}_i$ is in a medium. Otherwise, it is zero.

`mRaySampleRevPdfsRatio` Equals $\frac{\Pr\{t>t_{\mathbf{x}_{i-1}\mathbf{x}_i}\}}{p(t_{\mathbf{x}_{i-1}\mathbf{x}_i})}$ if $\mathbf{x}_i$ is in a medium (condition $i > s$ is not required to be satisfied since the ratio can be computed anyway). Otherwise, it is zero.

`mSinTheta` Equals $\sin\theta_{\mathbf{x}_i}$ if $i > 0$, the vertex is located in a medium and its phase function was sampled. Otherwise, it is zero.

`mCosThetaOut` If the scattering function of the vertex was sampled, then it equals $D(\leftarrow\mathbf{x}_i)$. Otherwise, it is zero.

[4] $D(\leftarrow\mathbf{x}_i) = |n_{\mathbf{x}_i} \cdot \omega_{\leftarrow\mathbf{x}_i}|$, see Section 3.2.1

[5] $t_{\mathbf{x}_{i-1}\mathbf{x}_i}$ is a distance between the vertices $\mathbf{x}_{i-1}$ and $\mathbf{x}_i$, considered as sampled in the direction from the light, see Section 1.2

`mSurfMisWeightFactor` Equals $\frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if the vertex is located on a surface but not on a light source. Otherwise, it is zero.

`mPP3DMisWeightFactor` Equals $\frac{n_{paths}}{K_3(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if the vertex is located in a medium. Otherwise, it is zero.

`mPB2DMisWeightFactor` Equals $\frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if the vertex is located in a medium. Otherwise, it is zero.

`mBB1DMisWeightFactor` Equals $\frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_i,\mathbf{x}_i)}$ if the vertex is located in a medium. Otherwise, it is zero.

`mIsDelta` Equals `true` if the vertex is located on a purely specular surface but not on a light source. Otherwise, it is `false`.

`mIsOnLightSource` Equals `true` if the vertex is located on a light source. Otherwise, it is `false`.

`mIsSpecular` Equals `true` if the vertex is located on a surface and a specular component of its material BSDF was sampled. Otherwise, it is `false`.

As in the case of light subpaths, we now describe where the values are gathered and computed during tracing the camera subpath. In contrast to the first light vertex, data for the first camera vertex are never used and therefore are not computed. Update of data after sampling the scattering function is the same as for light vertices (performed by the same code from Listing 3.65) but for the opposite direction. So the only code missing is that handling MIS data of newly created camera vertex (i.e. $\mathbf{x}_i$, $i = s < k$) after tracing a ray in the `RunIteration` method (on line 40 in Listing 3.26):

Listing 3.66: `SetCameraVertexMisData` (part of the `RunIteration` method, `UPBP.hxx`)

```
1  const float distSq = Utils::sqr(isect.mDist);
2  const float raySamplePdfInv = 1.0f / raySamplePdf;
3  MisData& cameraVertexMis = mCameraVerticesMisData[cameraState.mPathLength];
4
5  // Set pdfs.
6  cameraVertexMis.mPdfInv = cameraState.mLastPdfWInv * distSq * raySamplePdfInv /
7     std::abs(bsdf.CosThetaFix());
8  cameraVertexMis.mRevPdf = 1.0f;
9  cameraVertexMis.mRevPdfWithoutBsdf = cameraVertexMis.mRevPdf;
10 cameraVertexMis.mRaySamplePdfInv = raySamplePdfInv;
11 cameraVertexMis.mRaySampleRevPdfInv = 1.0f;
12 cameraVertexMis.mRaySamplePdfsRatio = bsdf.IsInMedium() ?
13    1.0f / bsdf.GetMedium()->mMinPositiveAttenuationCoefComp() : 0.0f;
14 cameraVertexMis.mRaySampleRevPdfsRatio = cameraVertexMis.mRaySamplePdfsRatio;
15
16 // Set angles.
17 cameraVertexMis.mSinTheta = 0.0f;
18 cameraVertexMis.mCosThetaOut = 0.0f;
19
20 // Set MIS weight factors.
21 cameraVertexMis.mSurfMisWeightFactor = bsdf.IsOnSurface() ?
22    (isect.mLightID >= 0 ? 0.0f : mSurfMisWeightFactor) : 0.0f;
23 cameraVertexMis.mPP3DMisWeightFactor = bsdf.IsOnSurface() ?
24    0.0f : mPP3DMisWeightFactor;
25 cameraVertexMis.mPB2DMisWeightFactor = bsdf.IsOnSurface() ?
26    0.0f : mPB2DMisWeightFactor;
27 cameraVertexMis.mBB1DMisWeightFactor = bsdf.IsOnSurface() ?
28    0.0f : mBB1DMisWeightFactor;
29
30 ..SetCameraVertexMisDataPart2..
```

Listing 3.67: `SetCameraVertexMisDataPart2` (part of the `RunIteration` method, `UPBP.hxx`)

```
1  // Set flags.
2  cameraVertexMis.mIsDelta = isect.mLightID >= 0 ? false : bsdf.IsDelta();
3  cameraVertexMis.mIsOnLightSource = isect.mLightID >= 0;
4  cameraVertexMis.mIsSpecular = false;
5
6  // Update reverse pdfs of the previous vertex.
7  mCameraVerticesMisData[cameraState.mPathLength - 1].mRevPdf *=
8      raySampleRevPdf / distSq;
9  mCameraVerticesMisData[cameraState.mPathLength - 1].mRevPdfWithoutBsdf =
10     mCameraVerticesMisData[cameraState.mPathLength - 1].mRevPdf
11 mCameraVerticesMisData[cameraState.mPathLength - 1].mRaySampleRevPdfInv =
12     1.0f / raySampleRevPdf;
```

The code performs the same operations as in the case of light vertices, but for the opposite direction. It is therefore not necessary to repeat its description. However, new camera vertices are not created only when an "intersection" with a scene is found, but also in case when a ray leaves the scene. Such a vertex represents a point on the background light at infinity. Its MIS data are computed the following way (on line 52 in Listing 3.26):

Listing 3.68: `SetCameraVertexMisDataIfLeft` (part of the `RunIteration` method, `UPBP.hxx`)

```
1  const float raySamplePdfInv = 1.0f / raySamplePdf
2  MisData& cameraVertexMis = mCameraVerticesMisData[cameraState.mPathLength];
3
4  // Set pdfs.
5  cameraVertexMis.mPdfInv = cameraState.mLastPdfWInv * raySamplePdfInv;
6  cameraVertexMis.mRevPdf = 1.0f;
7  cameraVertexMis.mRevPdfWithoutBsdf = cameraVertexMis.mRevPdf;
8  cameraVertexMis.mRaySamplePdfInv = raySamplePdfInv;
9  cameraVertexMis.mRaySampleRevPdfInv = 1.0f;
10 cameraVertexMis.mRaySamplePdfsRatio = 0.0f;
11 cameraVertexMis.mRaySampleRevPdfsRatio = cameraVertexMis.mRaySamplePdfsRatio;
12
13 // Set angles.
14 cameraVertexMis.mSinTheta = 0.0f;
15 cameraVertexMis.mCosThetaOut = 0.0f;
16
17 // Set MIS weight factors.
18 cameraVertexMis.mSurfMisWeightFactor = 0.0f;
19 cameraVertexMis.mPP3DMisWeightFactor = 0.0f;
20 cameraVertexMis.mPB2DMisWeightFactor = 0.0f;
21 cameraVertexMis.mBB1DMisWeightFactor = 0.0f;
22
23 // Set flags.
24 cameraVertexMis.mIsDelta = false;
25 cameraVertexMis.mIsOnLightSource = true;
26 cameraVertexMis.mIsSpecular = false;
27
28 // Update reverse pdfs of the previous vertex.
29 mCameraVerticesMisData[cameraState.mPathLength - 1].mRevPdf *=
30     raySampleRevPdf;
31 mCameraVerticesMisData[cameraState.mPathLength - 1].mRevPdfWithoutBsdf =
32     mCameraVerticesMisData[cameraState.mPathLength - 1].mRevPdf
33 mCameraVerticesMisData[cameraState.mPathLength - 1].mRaySampleRevPdfInv =
34     1.0f / raySampleRevPdf;
```

This computation is simpler. Since the background light source is in infinity, the distance and cosine vanish.

3.5.2.3 Camera subpath MIS weight

Once we have all necessary data ready we can proceed to the computation of MIS weights. The crucial task is to evaluate the $w^C_{k,s,b}$ (3.14) and $w^L_{k,r,a}$ (3.15) terms. In Section 3.5.1.4 we theoretically derived effective algorithms 3.60 and 3.61 for this purpose. Now we show how they are actually implemented.

We begin with algorithm 3.60 for computing the $w^C_{k,s,b}$ term. Recall that b is an index of the last camera vertex on the complete path of length k, s is an initialization factor for the algorithm (see Section 3.5.1.4). Note that b is counted in direction from a light, but we speak about the camera vertices in the order they were sampled, i.e. we call vertex $\mathbf{x}_b$ the last vertex of the camera subpath and $\mathbf{x}_k$ the first vertex of the camera subpath. The algorithm is implemented in the `AccumulateCameraPathWeight` method:

Listing 3.69: `AccumulateCameraPathWeight` (method, `PathWeight.hxx`)

```
static float AccumulateCameraPathWeight(
   const int      aPathLength,
   const float    aLastRevPdf,
   const float    aLastSinTheta,
   const float    aLastRaySampleRevPdfInv,
   const float    aLastRaySampleRevPdfsRatio,
   const float    aNextToLastPartialRevPdfW,
   const uint     aQueryBeamType,
   const uint     aPhotonBeamType,
   const uint     aEstimatorTechniques,
   const MisData *aCameraVerticesMisData)
{
   // The resulting weight.
   float weight = 0;

   // The common factor.
   float product = 1.0f;

   // An index counting vertices in the order in which they are processed
   // (opposite to the sampling order).
   int index = 0;

   // The first camera vertex is ignored. Therefore, the cycle stops before
   // index = PathLength.
   while (index < aPathLength)
   {
      // Get data for the vertex to process.
      const MisData& current = aCameraVerticesMisData[aPathLength - index];

      // Get specularity of the next vertex to process. The first camera vertex
      // is never specular.
      bool nextIsSpecular = index < aPathLength - 1 ?
         aCameraVerticesMisData[aPathLength - index - 1].mIsSpecular : false;

      // Get valid reverse data.
      float rev = current.mRevPdf;
      float sinTheta = current.mSinTheta;
      float rayRev = current.mRaySampleRevPdfInv;
      float rayRevRatio = current.mRaySampleRevPdfsRatio;
      if (index == 0) {
         rev = aLastRevPdf;
         sinTheta = aLastSinTheta;
         rayRev = aLastRaySampleRevPdfInv;
         rayRevRatio = aLastRaySampleRevPdfsRatio;
      }
      else if (index == 1) rev *= aNextToLastPartialRevPdfW;

      ..AccumCamPart2..
      }
   return weight;
}
```

Listing 3.70: `AccumCamPart2` (part of the `AccumulateCameraPathWeight` method, `PathWeight.hxx`)

```
// Update the common factor.
product *= rev;

// Ignore specularity of sampled event for the last camera vertex.
// The last camera vertex is never delta here (the method would not
// be called) and sampling a specular event is a thing of path
// continuation not relevant for the last camera vertex.
bool currentUsable = index == 0 || !current.mIsSpecular;

// Add weight contribution of the SURF estimator if used
// and the vertex was not sampled specular.
if ((aEstimatorTechniques & SURF) && currentUsable)
   weight += product * current.mSurfMisWeightFactor;

// Add weight contribution of the PP3D estimator if used.
if (aEstimatorTechniques & PP3D)
   weight += product * current.mPP3DMisWeightFactor;

// Add weight contribution of the PB2D estimator if used.
if (aEstimatorTechniques & PB2D)
{
   if (aQueryBeamType & LONG_BEAM)
      weight += product * current.mPB2DMisWeightFactor
         * current.mRaySamplePdfInv;
   else
      weight += product * current.mPB2DMisWeightFactor
         * current.mRaySamplePdfsRatio;
}

// Add weight contribution of the BB1D estimator if used.
if (aEstimatorTechniques & BB1D)
{
   if (aQueryBeamType & LONG_BEAM)
   {
      if (aPhotonBeamType & LONG_BEAM)
         weight += product * current.mBB1DMisWeightFactor
            * current.mRaySamplePdfInv * sinTheta * rayRev;
      else
         weight += product * current.mBB1DMisWeightFactor
            * current.mRaySamplePdfInv * sinTheta * rayRevRatio;
   }
   else
   {
      if (aPhotonBeamType & LONG_BEAM)
         weight += product * current.mBB1DMisWeightFactor
            * current.mRaySamplePdfsRatio * sinTheta * rayRev;
      else
         weight += product * current.mBB1DMisWeightFactor
            * current.mRaySamplePdfsRatio * sinTheta * rayRevRatio;
   }
}

// Get inverse of the forward pdf.
float fwdInv = current.mPdfInv;

// Update the common factor.
product *= fwdInv;

// Add weight contribution of the BPT estimator if used and none
//  of the two vertices were sampled specular.
if ((aEstimatorTechniques & BPT) && currentUsable && !nextIsSpecular)
   weight += product;

++index;
```

Method arguments. First, we explain arguments of the method:

`aPathLength` A length of the camera subpath, i.e. $k - b$.

`aLastRevPdf` Equals $\overrightarrow{sp}_b$.

`aLastSinTheta` Equals $\sin\theta_{\mathbf{x}_b}$ if $b > 0$ and $\mathbf{x}_b$ is in a medium. Otherwise, it is 0.

`aLastRaySampleRevPdfInv` Equals $\frac{1}{p(t_{\mathbf{x}_{b-1}\mathbf{x}_b})}$ if $b > 0$. Otherwise, it is 0.

`aLastRaySampleRevPdfsRatio` Equals $\frac{\Pr\{t>t_{\mathbf{x}_{b-1}\mathbf{x}_b}\}}{p(t_{\mathbf{x}_{b-1}\mathbf{x}_b})}$ if $b > 0$ and $\mathbf{x}_b$ is in a medium. Otherwise, it is 0.

`aNextToLastPartialRevPdfW` Equals $\hat{p}(\omega_{\mathbf{x}_b\mathbf{x}_{b+1}})$.

`aQueryBeamType` A type of query beams.

`aPhotonBeamType` A type of photon beams.

`aEstimatorTechniques` Flags of used estimator techniques.

`*aCameraVerticesMisData` `MisData` structures for the camera subpath.

The necessary `MisData` structures with data for camera vertices $\mathbf{x}_b, \mathbf{x}_{b+1}, \ldots, \mathbf{x}_k$ are stored in the given `aCameraVerticesMisData` array at indices `aPathLength`, `aPathLength` $- 1, \ldots, 0$ (in this order). They however cannot provide all data for the last and next-to-last vertex on the camera subpath (i.e. for $\mathbf{x}_b$ and $\mathbf{x}_{b+1}$) since the reverse pdfs and angle θ at $\mathbf{x}_b$ depend on a connecting edge with $\mathbf{x}_{b-1}$, i.e. on the light subpath. They are therefore supplied separately via arguments (`aLastRevPdf` - `aNextToLastPartialRevPdfW`).

Method body. Now we go through the body of the method. Let $\mathbf{x}_m$ denote a vertex to be processed, $m \in \{b, \ldots, k-1\}$ (we do not process $\mathbf{x}_k$ as explained in Section 3.5.1.4). A partially computed term $w^C_{k,s,b}$ is kept in the `weight` field (corresponds to w in algorithm 3.60), the `product` field stores common factor $R^C_{s,b}(m)$ (corresponds to R in algorithm 3.60). The method cycles over the camera vertices from $\mathbf{x}_b$ to $\mathbf{x}_{k-1}$ (the `index` field corresponds to $m - b$) and for each of them performs:

1. Gets the `MisData` structure for vertex $\mathbf{x}_m$ and stores it in the `current` field. It will be needed for evaluation of all the estimators.

2. Gets valid "reverse" data for vertex $\mathbf{x}_m$:

   `rev` If $m = b$, equals $\overrightarrow{sp}_m$. Otherwise, $\overrightarrow{p}_m$.

   `sinTheta` Equals $\sin\theta_{\mathbf{x}_m}$ if $m > 0$ and $\mathbf{x}_m$ is in a medium. Otherwise, it is 0.

   `rayRev` Equals $\frac{1}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}$ if $m > 0$. Otherwise, it is 0.

   `rayRevRatio` Equals $\frac{\Pr\{t>t_{\mathbf{x}_{m-1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}$ if $m > 0$ and $\mathbf{x}_m$ is in a medium. Otherwise, it is 0.

For $m = b$ these values have to be taken from the method arguments, for $m > b$ `current` data are used. For $m = b + 1$ `current.mRevPdf` lacks the pdf of sampling the scattering function at vertex $\mathbf{x}_b$ (see its definition in Section 3.5.2.2) and is therefore completed using method argument `aNextToLastPartialRevPdfW`.

3. Updates the `product` common factor by `rev` so it equals $R^C_{s,b}(m)$ (as on line 6 in algorithm 3.60). Note that initialization factor s is for $m = b$ included in `rev`.

4. Computes weight contribution of the SURF estimator (as on line 9 in algorithm 3.60):

$$\begin{aligned}\texttt{weight+=}&\texttt{product * current.mSurfMisWeightFactor}\\ &= R^C_{s,b}(m) I_S(m) \frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_m, \mathbf{x}_m)}\\ &= R^C_{s,b}(m) I_S(m) F_{\mathrm{SURF}}(m).\end{aligned}$$

   The computation is carried out only when vertex $\mathbf{x}_m$ was not sampled specular. This condition together with the fact `mSurfMisWeightFactor` is zero for all but vertices on surfaces realizes the $I_S(m)$ indicator function.

5. Computes weight contribution of the P-P3D estimator (as on line 7 in algorithm 3.60):

$$\begin{aligned}\texttt{weight+=}&\texttt{product * current.mPP3DMisWeightFactor}\\ &= R^C_{s,b}(m) I_M(m) \frac{n_{paths}}{K_3(\tilde{\mathbf{x}}_m, \mathbf{x}_m)}\\ &= R^C_{s,b}(m) I_M(m) F_{\text{P-P3D}}(m).\end{aligned}$$

   The fact `mPP3DMisWeightFactor` is zero for all but vertices in media realizes the $I_M(m)$ indicator function.

6. Computes weight contribution of the P-B$_\mathrm{l}$2D estimator (as on line 7 in algorithm 3.60):

$$\begin{aligned}\texttt{weight+=}&\texttt{product * current.mPB2DMisWeightFactor}\\ &\texttt{\quad * current.mRaySamplePdfInv}\\ &= R^C_{s,b}(m) I_M(m) \frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{1}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})}\\ &= R^C_{s,b}(m) I_M(m) F_{\text{P-B}_\mathrm{l}\text{2D}}(m)\end{aligned}$$

   or of the P-B$_\mathrm{s}$2D estimator:

$$\begin{aligned}\texttt{weight+=}&\texttt{product * current.mPB2DMisWeightFactor}\\ &\texttt{\quad * current.mRaySamplePdfsRatio}\\ &= R^C_{s,b}(m) I_M(m) \frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{\Pr\{t > t_{\mathbf{x}_{m+1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})}\\ &= R^C_{s,b}(m) I_M(m) F_{\text{P-B}_\mathrm{s}\text{2D}}(m).\end{aligned}$$

   The fact `mPB2DMisWeightFactor` is zero for all but vertices in media realizes the $I_M(m)$ indicator function.

7. Computes weight contribution of the B_l-B_l1D estimator (as on line 7 in algorithm 3.60):

$$\begin{aligned}\texttt{weight+=product * current.mBB1DMisWeightFactor}&\\ \texttt{* current.mRaySamplePdfInv * sinTheta * rayRev}&\\ = R^C_{s,b}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{1}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m} \frac{1}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}&\\ = R^C_{s,b}(m) I_M(m) F_{\text{B}_l\text{-B}_l\text{1D}}(m)&\end{aligned}$$

or of the B_s-B_l1D estimator:

$$\begin{aligned}\texttt{weight+=product * current.mBB1DMisWeightFactor}&\\ \texttt{* current.mRaySamplePdfInv * sinTheta * rayRevRatio}&\\ = R^C_{s,b}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{1}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m} \frac{\Pr\{t > t_{\mathbf{x}_{m-1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}&\\ = R^C_{s,b}(m) I_M(m) F_{\text{B}_s\text{-B}_l\text{1D}}(m)&\end{aligned}$$

or of the B_l-B_s1D estimator:

$$\begin{aligned}\texttt{weight+=product * current.mBB1DMisWeightFactor}&\\ \texttt{* current.mRaySamplePdfsRatio * sinTheta * rayRev}&\\ = R^C_{s,b}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{\Pr\{t > t_{\mathbf{x}_{m+1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m} \frac{1}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}&\\ = R^C_{s,b}(m) I_M(m) F_{\text{B}_l\text{-B}_s\text{1D}}(m)&\end{aligned}$$

or of the B_s-B_s1D estimator:

$$\begin{aligned}\texttt{weight+=product * current.mBB1DMisWeightFactor}&\\ \texttt{* current.mRaySamplePdfsRatio * sinTheta}&\\ \texttt{* rayRevRatio}&\\ = R^C_{s,b}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{\Pr\{t > t_{\mathbf{x}_{m+1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m}&\\ \frac{\Pr\{t > t_{\mathbf{x}_{m-1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}&\\ = R^C_{s,b}(m) I_M(m) F_{\text{B}_s\text{-B}_s\text{1D}}(m).&\end{aligned}$$

The fact `mBB1DMisWeightFactor` is zero for all but vertices in media realizes the $I_M(m)$ indicator function.

8. Updates the `product` common factor by `fwdInv` $= \frac{1}{\overleftarrow{p_m}}$ (as on line 11 in algorithm 3.60).

9. Computes weight contribution of the BPT estimator (as on line 10 in algorithm 3.60):

$$\begin{aligned}\texttt{weight+=product}&\\ = R^C_{s,b}(m) I_{!\delta}(m, m+1) \frac{1}{\overleftarrow{p_m}}&\\ = R^C_{s,b}(m) I_{!\delta}(m, m+1) F_{\text{BPT}}(m).&\end{aligned}$$

The computation is carried out only when neither vertex $\mathbf{x}_m$ nor $\mathbf{x}_{m+1}$ were sampled specular which realizes the $I_{l\delta}(m, m+1)$ indicator function.

This way the MIS weight for the camera subpath is computed. As we can see, if method `AccumulateCameraPathWeight` is given the arguments it expects, it computes the correct $w^C_{k,s,b}$ term.

3.5.2.4 Light subpath MIS weight

After the camera subpath we similarly describe implementation of algorithm 3.61 computing MIS weight of the light subpath, i.e. the $w^L_{k,r,a}$ term. Recall that a is an index of the last light vertex on the complete path of length k, r is an initialization factor for the algorithm (see Section 3.5.1.4). Note that in contrast to the camera subpath, a is counted in direction from a light, the same direction in which we speak about the light vertices, i.e. we call vertex $\mathbf{x}_0$ the first vertex of the light subpath and $\mathbf{x}_a$ the last vertex of the camera subpath. The algorithm is implemented in the `AccumulateLightPathWeight` method:

Listing 3.71: `AccumulateLightPathWeight` (method, `PathWeight.hxx`)

```
static float AccumulateLightPathWeight(
    const int       aPathIndex,
    const int       aPathLength,
    const float     aLastRevPdf,
    const float     aLastSinTheta,
    const float     aLastRaySampleRevPdfInv,
    const float     aLastRaySampleRevPdfsRatio,
    const float     aNextToLastPartialRevPdfW,
    const uint      aCurrentlyEvaluatedTechnique,
    const uint      aQueryBeamType,
    const uint      aPhotonBeamType,
    const uint      aEstimatorTechniques,
    const std::vector<int>              *aPathEnds,
    const std::vector<UPBPLightVertex> *aLightVertices,
    const MisData *aBeamLightVertexMisData = NULL)
{
    // The resulting weight.
    float weight = 0;

    // The common factor.
    float product = 1.0f;

    // An index of the last light vertex.
    int lastIndex = (aPathIndex == 0) ?
        aPathLength : aPathEnds->at(aPathIndex - 1) + aPathLength;

    // An index counting vertices in the order in which they are processed.
    int index = 0;

    // We process the first light vertex too.
    while (index <= aPathLength)
    {
        // Get data for the vertex to process.
        const MisData& current =
            (aCurrentlyEvaluatedTechnique == BB1D && index == 0) ?
            *aBeamLightVertexMisData
            :
            aLightVertices->at(lastIndex - index).mMisData;

        ..AccumLightPart2..
    }

    return weight;
}
```

Listing 3.72: `AccumLightPart2` (part of the `AccumulateLightPathWeight` method, `PathWeight.hxx`)

```
// Get specularity of the next vertex to process. The first light vertex
// is never specular.
bool nextIsSpecular = index < aPathLength ?
   aLightVertices->at(lastIndex - index - 1).mMisData.mIsSpecular : false;

// Get valid reverse data.
float rev = current.mRevPdf;
float sinTheta = current.mSinTheta;
float rayRev = current.mRaySampleRevPdfInv;
float rayRevRatio = current.mRaySampleRevPdfsRatio;
if (index == 0)
{
   rev = aLastRevPdf;
   sinTheta = aLastSinTheta;
   rayRev = aLastRaySampleRevPdfInv;
   rayRevRatio = aLastRaySampleRevPdfsRatio;
}
else if (index == 1)
{
   if (aCurrentlyEvaluatedTechnique == BB1D)
      rev = aNextToLastPartialRevPdfW;
   else
      rev = current.mRevPdfAWithoutBsdf * aNextToLastPartialRevPdfW;
}

// Reverse probability is never zero unless we are on delta light
// (such vertex is never the last since there is no merging,
// connection or camera sampling for on-light vertices
// and light sampling does not call this method).
if (rev == 0) break;

// Update the common factor.
product *= rev;

// Ignore specularity of sampled event for the last light vertex.
// The last light vertex is never delta here (the method would not
// be called) and sampling a specular event is a thing of path
// continuation not relevant for the last light vertex.
bool currentUsable = index == 0 || !current.mIsSpecular;

// For photon density estimation techniques (SURF, PP3D, PB2D, BB1D)
// weight contribution of the last vertex is computed on the camera
// subpath.
if (index != 0 || aCurrentlyEvaluatedTechnique == BPT)
{
   ..AccumLightPart3..
}

// Get inverse of the forward pdf.
float fwdInv = current.mPdfInv;

// Update the common factor.
product *= fwdInv;

// Add weight contribution of the BPT estimator if used and none
// of the two vertices were sampled specular.
if ((aEstimatorTechniques & BPT) && currentUsable && !nextIsSpecular)
   weight += product;

++index;

// If in previous mode, a path can be created only by applying BPT
// or any of the volumetric photon density estimators at the first
// vertex from camera in a medium. Weight contribution of the volumetric
// photon density estimators is computed on the camera subpath,
// weight contribution of BPT (connection to a light source or a light
// vertex) is computed here in the first iteration and the method
// then ends.
if (aEstimatorTechniques & PREVIOUS) break;
```

Listing 3.73: `AccumLightPart3` (part of the `AccumulateLightPathWeight` method, `PathWeight.hxx`)

```
// Add weight contribution of the SURF estimator if used
// and the vertex was not sampled specular.
if ((aEstimatorTechniques & SURF) && currentUsable)
   weight += product * current.mSurfMisWeightFactor;

// Add weight contribution of the PP3D estimator if used.
if (aEstimatorTechniques & PP3D)
   weight += product * current.mPP3DMisWeightFactor;

// Add weight contribution of the PB2D estimator if used.
if (aEstimatorTechniques & PB2D)
{
   if (aQueryBeamType & LONG_BEAM)
      weight += product * current.mPB2DMisWeightFactor
         * rayRev;
   else
      weight += product * current.mPB2DMisWeightFactor
         * rayRevRatio;
}
// Add weight contribution of the BB1D estimator if used.
if (aEstimatorTechniques & BB1D)
{
   if (aQueryBeamType & LONG_BEAM)
   {
      if (aPhotonBeamType & LONG_BEAM)
         weight += product * current.mBB1DMisWeightFactor
            * rayRev * sinTheta * current.mRaySamplePdfInv;
      else
         weight += product * current.mBB1DMisWeightFactor
            * rayRev * sinTheta * current.mRaySamplePdfsRatio;
   }
   else
   {
      if (aPhotonBeamType & LONG_BEAM)
         weight += product * current.mBB1DMisWeightFactor
            * rayRevRatio * sinTheta * current.mRaySamplePdfInv;
      else
         weight += product * current.mBB1DMisWeightFactor
            * rayRevRatio * sinTheta * current.mRaySamplePdfsRatio;
   }
}
```

Method arguments. First, we explain arguments of the method:

`aPathIndex` An index of a stored light subpath with vertices to process. This light subpath can be longer than the one actually evaluated (e.g. if the camera subpath was connected to other than its last vertex).

`aPathLength` A length of the light subpath, i.e. a. It is a length of the actually evaluated part of the corresponding stored light subpath.

`aLastRevPdf` Equals $r\overleftarrow{p}_a$.

`aLastSinTheta` If the complete path was created by evaluating any of the photon density estimators, the value of `aLastSinTheta` is not used, since it is needed only for computing weight contribution of the B-B1D estimator at the last vertex and that is handled on the camera subpath (a zero is typically passed then). Otherwise, this method is run only when the path was created by BPT techniques connecting camera and light subpaths or connecting a light subpath to the camera. Then `aLastSinTheta` equals $\sin\theta_{\mathbf{x}_a}$ for vertices in media and 0 for vertices on surfaces.

`aLastRaySampleRevPdfInv` Similarly to `aLastSinTheta` it equals 0 if the complete path was created by evaluating any of the photon density estimators. Otherwise, it equals $\frac{1}{p(t_{\mathbf{x}_{a+1}\mathbf{x}_a})}$.

`aLastRaySampleRevPdfsRatio` Similarly to `aLastSinTheta` it equals 0 if the complete path was created by evaluating any of the photon density estimators. Otherwise, it equals $\frac{\Pr\{t>t_{\mathbf{x}_{a+1}\mathbf{x}_a}\}}{p(t_{\mathbf{x}_{a+1}\mathbf{x}_a})}$ for vertices in media and 0 for vertices on surfaces.

`aNextToLastPartialRevPdfW` Equals $p^{\leftarrow}_{a-1}$ if the complete path was created by the B-B1D estimator. Otherwise, it equals $\hat{p}(\omega_{\mathbf{x}_a\mathbf{x}_{a-1}})$.

`aCurrentlyEvaluatedTechnique` An identifier of the estimator technique which created the complete path. It is needed because for the photon density estimators only BPT weight contribution must be evaluated for the last light vertex, the rest is handled on the camera subpath.

`aQueryBeamType` A type of query beams.

`aPhotonBeamType` A type of photon beams.

`aEstimatorTechniques` Flags of used estimator techniques.

`aPathEnds` Indices of path ends in the `aLightVertices` array (`aPathEnds[i]` points on a vertex right after the last vertex of the i-th traced light subpath).

`aLightVertices` All stored light vertices (stored in the order they were created).

`aBeamLightVertexMisData` Points to the `MisData` structure for the last light vertex if the B-B1D estimator created the complete path. Otherwise, it is `NULL`.

While there is always only one camera subpath stored and it ends exactly at the vertex where estimators are currently evaluated, all light subpaths traced in the current rendering iteration are stored and typically only a subpath of one of them is the light subpath the method is called to compute MIS weight contribution for. The method therefore needs more arguments to properly identify the necessary data than `AccumulateCameraPathWeight`. And there is one more related difference. When evaluating estimators with beams (P-B2D or B-B1D) the vertex emerging on a query/photon beam is not actually created but its data are needed anyway. While they can be set in the static `mCameraVerticesMisData` array freely since they will be overwritten when the subpath continues, we can neither add to the `aLightVertices` list nor modify data of any of the stored vertices, since they can be still needed. `MisData` structure for the vertex emerging on a photon beam when evaluating the B-B1D estimator is therefore supplied separately in the `aBeamLightVertexMisData` method argument.

Method body. Now we go through the body of the method. Let $\mathbf{x}_m$ denote a vertex to be processed, $m \in \{0, \ldots, a\}$. A partially computed term $w^L_{k,r,a}$ is kept in the `weight` field (corresponds to w in algorithm 3.61), the `product` field stores common factor $R^L_{r,a}(m)$ (corresponds to R in algorithm 3.61). The method cycles

over the light vertices from $\mathbf{x}_a$ to $\mathbf{x}_0$ (the `index` field corresponds to $a-m$) and for each of them performs:

1. Gets the `MisData` structure for vertex $\mathbf{x}_m$ and stores it in the `current` field. It will be needed for evaluation of all the estimators. If the complete path was created by the B-B1D estimator and $m = a$, then the separately passed `aBeamLightVertexMisData` structure is used.

2. Gets valid "reverse" data for vertex $\mathbf{x}_m$:

   `rev` If $m = a$, equals $r\overleftarrow{p}_m$. Otherwise, $\overleftarrow{p}_m$.

   `sinTheta` Equals 0 if $m = a$ and the complete path was not created by BPT. Otherwise, 0 for vertices on surfaces and $\sin\theta_{\mathbf{x}_m}$ for vertices in media.

   `rayRev` Equals 0 if $m = a$ and the complete path was not created by BPT. Otherwise, $\frac{1}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})}$.

   `rayRevRatio` Equals 0 if $m = a$ and the complete path was not created by BPT. Otherwise, $\frac{\Pr\{t>t_{\mathbf{x}_{m+1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})}$ for vertices in media and 0 for vertices on surfaces..

   For $m = a$ these values have to be taken from the method arguments, for $m < a$ `current` data are used. For $m = a-1$ `current.mRevPdf` may contain a pdf dependent on the outgoing direction the stored light subpath originally continued in from $\mathbf{x}_a$ (this could not happen on the camera subpath). For other than the B-B1D estimator the `current.mRevPdfWithoutBsdf` value is therefore taken instead and completed using method argument `aNextToLastPartialRevPdfW`. For the B-B1D estimator complete $\overleftarrow{p}_{a-1}$ is already passed in `aNextToLastPartialRevPdfW`.

3. Updates the `product` common factor by `rev` so it equals $R^L_{r,a}(m)$ (as on line 6 in algorithm 3.61). Note that initialization factor r is for $m = a$ included in `rev`.

4. If $m < a$ or the complete path was created by the BPT estimator, evaluates weight contribution of photon density estimators (this condition realizes the $I_V(m)$ indicator function):

   (a) Computes weight contribution of the SURF estimator (as on line 11 in algorithm 3.61):

   $$\begin{aligned}\texttt{weight}+=&\ \texttt{product * current.mSurfMisWeightFactor}\\ =&\ R^L_{r,a}(m)I_S(m)\frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_m,\mathbf{x}_m)}\\ =&\ R^L_{r,a}(m)I_S(m)F_{\mathrm{SURF}}(m).\end{aligned}$$

   The computation is carried out only when vertex $\mathbf{x}_m$ was not sampled specular. This condition together with the fact `mSurfMisWeightFactor` is zero for all but vertices on surfaces realizes the $I_S(m)$ indicator function.

(b) Computes weight contribution of the P-P3D estimator (as on line 9 in algorithm 3.61):

```
weight+=product * current.mPP3DMisWeightFactor
```

$$= R^L_{r,a}(m) I_M(m) \frac{n_{paths}}{K_3(\tilde{\mathbf{x}}_m, \mathbf{x}_m)}$$
$$= R^L_{r,a}(m) I_M(m) F_{\text{P-P3D}}(m).$$

The fact `mPP3DMisWeightFactor` is zero for all but vertices in media realizes the $I_M(m)$ indicator function.

(c) Computes weight contribution of the P-B_l2D estimator (as on line 9 in algorithm 3.61):

```
weight+=product * current.mPB2DMisWeightFactor
        * rayRev
```

$$= R^L_{r,a}(m) I_M(m) \frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{1}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})}$$
$$= R^L_{r,a}(m) I_M(m) F_{\text{P-B}_l\text{2D}}(m)$$

or of the P-B_s2D estimator:

```
weight+=product * current.mPB2DMisWeightFactor
        * rayRevRatio
```

$$= R^L_{r,a}(m) I_M(m) \frac{n_{paths}}{K_2(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{\Pr\{t > t_{\mathbf{x}_{m+1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})}$$
$$= R^L_{r,a}(m) I_M(m) F_{\text{P-B}_s\text{2D}}(m).$$

The fact `mPB2DMisWeightFactor` is zero for all but vertices in media realizes the $I_M(m)$ indicator function. Note that in contrast to camera subpaths reverse pdfs are used here instead of direct ones.

(d) Computes weight contribution of the B_l-B_l1D estimator (as on line 9 in algorithm 3.61):

```
weight+=product * current.mBB1DMisWeightFactor
        * rayRev * sinTheta * current.mRaySamplePdfInv
```

$$= R^L_{r,a}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{1}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m} \frac{1}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}$$
$$= R^L_{r,a}(m) I_M(m) F_{\text{B}_l\text{-B}_l\text{1D}}(m)$$

or of the B_s-B_l1D estimator:

```
weight+=product * current.mBB1DMisWeightFactor
        * rayRev * sinTheta
        * current.mRaySamplePdfsRatio
```

$$= R^L_{r,a}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{1}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m} \frac{\Pr\{t > t_{\mathbf{x}_{m-1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}$$
$$= R^L_{r,a}(m) I_M(m) F_{\text{B}_s\text{-B}_l\text{1D}}(m)$$

or of the B_l-B_s1D estimator:

$$\begin{aligned}\texttt{weight+=}&\texttt{product * current.mBB1DMisWeightFactor}\\ &\texttt{* rayRevRatio * sinTheta}\\ &\texttt{* current.mRaySamplePdfInv}\\ &= R^L_{r,a}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{\Pr\{t > t_{\mathbf{x}_{m+1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m}\\ &\quad \frac{1}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}\\ &= R^L_{r,a}(m) I_M(m) F_{\mathrm{B_l\text{-}B_s1D}}(m)\end{aligned}$$

or of the B_s-B_s1D estimator:

$$\begin{aligned}\texttt{weight+=}&\texttt{product * current.mBB1DMisWeightFactor}\\ &\texttt{* rayRevRatio * sinTheta}\\ &\texttt{* current.mRaySamplePdfsRatio}\\ &= R^L_{r,a}(m) I_M(m) \frac{n_{paths}}{K_1(\tilde{\mathbf{x}}_m, \mathbf{x}_m)} \frac{\Pr\{t > t_{\mathbf{x}_{m+1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m+1}\mathbf{x}_m})} \sin\theta_{\mathbf{x}_m}\\ &\quad \frac{\Pr\{t > t_{\mathbf{x}_{m-1}\mathbf{x}_m}\}}{p(t_{\mathbf{x}_{m-1}\mathbf{x}_m})}\\ &= R^L_{r,a}(m) I_M(m) F_{\mathrm{B_s\text{-}B_s1D}}(m).\end{aligned}$$

The fact `mBB1DMisWeightFactor` is zero for all but vertices in media realizes the $I_M(m)$ indicator function. Note that in contrast to camera subpaths the usage of reverse and forward pdfs is swapped here.

5. Updates the `product` common factor by $\texttt{fwdInv} = \frac{1}{\overset{\mapsto}{p}_m}$ (as on line 18 in algorithm 3.61).

6. Computes weight contribution of the BPT estimator (as on line 13 in algorithm 3.61):

$$\begin{aligned}\texttt{weight+=}&\texttt{product}\\ &= R^L_{r,a}(m) I_{!\delta}(m-1, m) \frac{1}{\overset{\mapsto}{p}_m}\\ &= R^L_{r,a}(m) I_{!\delta}(m-1, m) F_{\mathrm{BPT}}(m).\end{aligned}$$

The computation is carried out only when neither vertex $\mathbf{x}_m$ nor $\mathbf{x}_{m-1}$ were sampled specular which realizes the $I_{!\delta}(m-1, m)$ indicator function. For $m = 0$ the $I_{!\delta L}(m)$ indicator function is realized by terminating the method on line 30 if the reverse probability is zero (happens only for delta light sources). If it is not terminated only BPT can contribute to the weight since the MIS weight factors of photon density estimators are all zero on light sources.

This way the MIS weight for the light subpath is computed. As we can see, if method `AccumulateLightPathWeight` is given the arguments it expects, it computes the correct $w^L_{k,r,a}$ term.

3.5.2.5 Complete path MIS weight

In the two previous sections we show how to implement computation of MIS weights for light and camera subpaths. Now it is time to put these two parts together and compute the MIS weight of a complete path. Section 3.5.1.4 lists formulas how to accomplish this for each of the estimators, here we present their programming code. In the following text k denotes a length of the complete path, $k \in \mathbb{N}$, $k \geq 2$.

To improve readability we place description of implementation of each estimator on a new page.

BPT: connecting vertices. Recall that the MIS weight for the BPT technique connecting vertices $\mathbf{x}_c$, $\mathbf{x}_{c+1}$, $c \in \{1, \ldots, k-2\}$ satisfies Equation 3.16:

$$\frac{1}{\hat{w}_{BPT_{c,c+1}}} = w^L_{k,1,c} + 1 + w^C_{k,1,c+1}.$$

This equation is implemented in the `ConnectVertices` method (on line 24 in Listing 3.37):

Listing 3.74: `ConnectToVertexMis` (part of the `ConnectVertices` method, `UPBP.hxx`)

```
1  // Camera part.
2  const float wCamera = AccumulateCameraPathWeight(
3      aCameraState.mPathLength,
4      raySampleRevPdf * lightBsdfDirPdfA,
5      sinThetaCamera,
6      1.0f / raySampleRevPdf,
7      mCameraVerticesMisData[aCameraState.mPathLength].mRaySamplePdfsRatio,
8      cameraBsdfRevPdfW,
9      mQueryBeamType,
10     mPhotonBeamType,
11     mEstimatorTechniques,
12     mCameraVerticesMisData);
13
14 // Light part.
15 const float wLight = AccumulateLightPathWeight(
16     aLightVertex.mPathIdx,
17     aLightVertex.mPathLength,
18     raySamplePdf * cameraBsdfDirPdfA,
19     sinThetaLight,
20     1.0f / raySamplePdf,
21     aLightVertex.mMisData.mRaySamplePdfsRatio,
22     lightBsdfRevPdfW,
23     BPT,
24     mQueryBeamType,
25     mPhotonBeamType,
26     mEstimatorTechniques,
27     &mPathEnds,
28     &mLightVertices,
29     NULL);
30
31 // Complete weight.
32 const float misWeight = 1.0f / (wLight + 1.0f + wCamera);
```

We begin with the MIS weight of the camera subpath. Most of the arguments passed to the `AccumulateCameraPathWeight` are obvious, we discuss only these:

2nd (`aLastRevPdf`) It is given `raySampleRevPdf * lightBsdfDirPdfA`. The factors are computed earlier in the method and their product equals:

$$p(t_{\mathbf{x}_c\mathbf{x}_{c+1}})\hat{p}(\omega_{\mathbf{x}_c\mathbf{x}_{c+1}})\frac{D(\mathbf{x}_{c+1} \to \mathbf{x}_c)}{|\mathbf{x}_c - \mathbf{x}_{c+1}|^2} = \overset{\mapsto}{p}_{c+1}$$

i.e. the required value of the `aLastRevPdf` argument for $s = 1$.

3rd (`aLastSinTheta`) It is given the `sinThetaCamera` value computed earlier in the method. It equals 0 if $\mathbf{x}_{c+1}$ is on a surface and $\sin\theta_{\mathbf{x}_{c+1}}$ if it is in a medium, i.e. it equals the required value of the `aLastSinTheta` argument.

4th (`aLastRaySampleRevPdfInv`) It is given `1.0f / raySampleRevPdf`. The denominator is computed earlier in the method and equals $p(t_{\mathbf{x}_c\mathbf{x}_{c+1}})$, i.e. the fraction equals the required value of the `aLastRaySampleRevPdfInv` argument.

5th (`aLastRaySampleRevPdfsRatio`) It is given the `mRaySamplePdfsRatio` property of $\mathbf{x}_{c+1}$ since it depends neither on a direction nor on $\mathbf{x}_c$. It equals 0 if $\mathbf{x}_{c+1}$ is on a surface and $\frac{\Pr\{t>t_{\mathbf{x}_c\mathbf{x}_{c+1}}\}}{p(t_{\mathbf{x}_c\mathbf{x}_{c+1}})}$ if it is in a medium, i.e. it equals the required value of the `aLastRaySampleRevPdfsRatio` argument.

6th (`aNextToLastPartialRevPdfW`) It is given the `cameraBsdfRevPdfW` value computed earlier in the method. It equals $\hat{p}(\omega_{\mathbf{x}_{c+1}\mathbf{x}_{c+2}})$, i.e. the required value of the `aNextToLastPartialRevPdfW` argument.

Computation of the MIS weight of the light subpath is very similar. Arguments of the `AccumulateLightPathWeight` method are analogous but for the $\mathbf{x}_c$ vertex and the opposite direction.

We can see that both methods are given the right arguments, so they return the correct $w^C_{k,1,c+1}$ and $w^L_{k,1,c}$ terms and we get

$$\begin{aligned}\texttt{misWeight} &= \texttt{1.0f / (wLight + 1.0f + wCamera)}\\ &= \frac{1}{w^L_{k,1,c} + 1 + w^C_{k,1,c+1}} = \hat{w}_{BPT_{c,c+1}}.\end{aligned}$$

BPT: connecting to a light. For for the BPT technique connecting vertices $\mathbf{x}_0$ and $\mathbf{x}_1$ we have Equation 3.17:

$$\frac{1}{\hat{w}_{BPT_{0,1}}} = I_{!\delta L}(0) F^L_{\mathrm{BPT}}(0) R^L_{1,0}(0) + 1 + w^C_{k,1,1}.$$

This equation is implemented in the `DirectIllumination` method (on line 44 in Listing 3.35):

Listing 3.75: `ConnectToLightMis` (part of the `DirectIllumination` method, `UPBP.hxx`)

```
1  float misWeight = 1.f;
2  // Full BPT.
3  if (mConnectToLightVertices)
4  {
5     // Camera path.
6     const float wCamera = AccumulateCameraPathWeight(
7        aCameraState.mPathLength,
8        raySampleRevPdf * emissionPdfW * cosToLight / (directPdfW * cosAtLight),
9        sinTheta,
10       1.0f / raySampleRevPdf,
11       mCameraVerticesMisData[aCameraState.mPathLength].mRaySamplePdfsRatio,
12       bsdfRevPdfW
13       mQueryBeamType,
14       mPhotonBeamType,
15       mEstimatorTechniques,
16       mCameraVerticesMisData);
17
18    // Light part.
19    const float wLight = light->IsDelta() ?
20       0 : (raySamplePdf * bsdfDirPdfW) / (directPdfW * lightPickProb);
21
22    // Complete weight.
23    misWeight = 1.0f / (wLight + 1.0f + wCamera);
24 }
25 // Only PT with explicit light sampling and accumulation of emission
26 // of directly hit light sources
27 else if (mAlgorithm != kPTls && !light->IsDelta())
28    misWeight = 1.0f /
29       ((raySamplePdf * bsdfDirPdfW) / (directPdfW * lightPickProb) + 1.0f);
```

This time only the `AccumulateCameraPathWeight` method is used. Some of its arguments are again worth noting:

2nd (`aLastRevPdf`) It is given `raySampleRevPdf*emissionPdfW*cosToLight / (directPdfW*cosAtLight)`. The factors are computed earlier in the method and their product equals:

$$\frac{p(t_{\mathbf{x}_0\mathbf{x}_1}) p(\mathbf{x}_0) \hat{p}(\omega_{\mathbf{x}_0\mathbf{x}_1}) D(\mathbf{x}_1 \to \mathbf{x}_0)}{p(\mathbf{x}_0) \frac{|\mathbf{x}_0 - \mathbf{x}_1|^2}{D(\mathbf{x}_0 \to \mathbf{x}_1)} D(\mathbf{x}_0 \to \mathbf{x}_1)} = p(t_{\mathbf{x}_0\mathbf{x}_1}) \hat{p}(\omega_{\mathbf{x}_0\mathbf{x}_1}) \frac{D(\mathbf{x}_1 \to \mathbf{x}_0)}{|\mathbf{x}_0 - \mathbf{x}_1|^2} = \overset{\mapsto}{p}_1,$$

i.e. the required value of the `aLastRevPdf` argument for $s = 1$.

3rd (`aLastSinTheta`) It is given the `sinTheta` value computed earlier in the method. It equals 0 if $\mathbf{x}_1$ is on a surface and $\sin\theta_{\mathbf{x}_1}$ if it is in a medium, i.e. it equals the required value of the `aLastSinTheta` argument.

4th (`aLastRaySampleRevPdfInv`) It is given `1.0f / raySampleRevPdf`. The denominator is computed earlier in the method and equals $p(t_{\mathbf{x}_0\mathbf{x}_1})$, i.e. the fraction equals the required value of the `aLastRaySampleRevPdfInv` argument.

5th (`aLastRaySampleRevPdfsRatio`) It is given the `mRaySamplePdfsRatio` property of $\mathbf{x}_1$ since it depends neither on a direction nor on $\mathbf{x}_0$. It equals 0 if $\mathbf{x}_1$ is on a surface and $\frac{\Pr\{t>t_{\mathbf{x}_0\mathbf{x}_1}\}}{p(t_{\mathbf{x}_0\mathbf{x}_1})}$ if it is in a medium, i.e. it equals the required value of the `aLastRaySampleRevPdfsRatio` argument.

6th (`aNextToLastPartialRevPdfW`) It is given the `bsdfRevPdfW` value computed earlier in the method. It equals $\hat{p}(\omega_{\mathbf{x}_1\mathbf{x}_2})$, i.e. the required value of the `aNextToLastPartialRevPdfW` argument.

Receiving these arguments the method returns the correct $w^C_{k,1,1}$ term.

The MIS weight of the light subpath is computed directly as `(raySamplePdf * bsdfDirPdfW) / (directPdfW * lightPickProb)`. The factors are computed earlier in the method and their product equals:

$$\frac{p(t_{\mathbf{x}_1\mathbf{x}_0})\hat{p}(\omega_{\mathbf{x}_1\mathbf{x}_0})}{p(\mathbf{x}_0)\frac{|\mathbf{x}_0-\mathbf{x}_1|^2}{D(\mathbf{x}_0\to\mathbf{x}_1)}} = \frac{p(t_{\mathbf{x}_0\mathbf{x}_1})\hat{p}(\omega_{\mathbf{x}_0\mathbf{x}_1})\frac{D(\mathbf{x}_1\to\mathbf{x}_0)}{|\mathbf{x}_0-\mathbf{x}_1|^2}}{p(\mathbf{x}_0)} = \frac{1}{\overset{\leftarrow}{p}_0} = F^L_{\mathrm{BPT}}(0)R^L_{1,0}(0).$$

The $I_{!\delta L}(0)$ indicator function is realized by performing this computation only if $\mathbf{x}_0$ is not on a delta light source.

Using the computed results for the subpaths we get the MIS weight of the complete path:

$$\begin{aligned}\texttt{misWeight} &= \texttt{1.0f / (wLight + 1.0f + wCamera)}\\ &= \frac{1}{I_{!\delta L}(0)F^L_{\mathrm{BPT}}(0)R^L_{1,0}(0) + 1 + w^C_{k,1,1}} = \hat{w}_{BPT_{0,1}}.\end{aligned}$$

Note that the `DirectIllumination` method is called in three distinct cases depending on estimators the renderer is configured to combine. If it uses full BPT (i.e. if `mConnectToLightVertices` is set, see Section 3.3.1), then the MIS weight is computed as presented. However, if it is allowed to use only path tracing with explicit light sampling and accumulation of emission of directly hit light sources (`mAlgorithm` is `kPTmis`) and the light source is not delta, then there is only one technique besides the evaluated one that could create the path – sampling the entire path from the camera. It is the same technique as considered when computing the `wLight` term, i.e. the resulting weight is also the same up to the missing `mCamera` term. Finally, if the renderer cannot accumulate emission of directly hit light sources (`mAlgorithm` is `kPTmis`) or if the light source is delta, then the path could not be created by any other technique and weighting is not needed (the weight is 1).

BPT: connecting to the camera. If connecting vertices $\mathbf{x}_{k-1}$, $\mathbf{x}_k$, there is no w^C term and the corresponding weight satisfies Equation 3.18:

$$\frac{1}{\hat{w}_{BPT_{k-1,k}}} = w^L_{k,1,k-1} + 1.$$

This equation is implemented in the `ConnectToCamera` method (on line 24 in Listing 3.23):

Listing 3.76: `ConnectToCameraMis` (part of the `ConnectToCamera` method, `UPBP.hxx`)

```
1  // Light part.
2  const float wLight = AccumulateLightPathWeight(
3      aLightPathIdx,
4      aLightState.mPathLength,
5      raySampleRevPdf * cameraPdfA,
6      sinTheta,
7      1.0f / raySampleRevPdf,
8      aRaySampleRevPdfsRatio,
9      bsdfRevPdfW,
10     BPT,
11     mQueryBeamType,
12     mPhotonBeamType,
13     mEstimatorTechniques,
14     aCameraConnection,
15     &mPathEnds,
16     &mLightVertices,
17     NULL)
18     / mPathCount;
19
20 // Complete weight.
21 misWeight = 1.0f / (wLight + 1.0f);
```

We again point out only a few of arguments of the `AccumulateCameraPathWeight` called to compute the MIS weight of the light subpath:

2nd (`aLastRevPdf`) It is given `raySampleRevPdf * cameraPdfA`. The factors are computed earlier in the method and their product equals:

$$p(t_{\mathbf{x}_k\mathbf{x}_{k-1}})\hat{p}(\omega_{\mathbf{x}_k\mathbf{x}_{k-1}})\frac{D(\mathbf{x}_{k-1}\to\mathbf{x}_k)}{|\mathbf{x}_{k-1}-\mathbf{x}_k|^2} = \overleftarrow{p_{k-1}}$$

i.e. the required value of the `aLastRevPdf` argument for $s = 1$.

3rd (`aLastSinTheta`) It is given the `sinTheta` value computed earlier in the method. It equals 0 if $\mathbf{x}_{k-1}$ is on a surface and $\sin\theta_{\mathbf{x}_{k-1}}$ if it is in a medium, i.e. it equals the required value of the `aLastSinTheta` argument.

4th (`aLastRaySampleRevPdfInv`) It is given `1.0f / raySampleRevPdf`. The denominator is computed earlier in the method and equals $p(t_{\mathbf{x}_k\mathbf{x}_{k-1}})$, i.e. the fraction equals the required value of the `aLastRaySampleRevPdfInv` argument.

5th (`aLastRaySampleRevPdfsRatio`) It is given the `mRaySamplePdfsRatio` property of $\mathbf{x}_{k-1}$ (passed via `aRaySampleRevPdfsRatio` argument of method `ConnectToCamera`) since it depends neither on a direction nor on $\mathbf{x}_k$. It equals 0 if $\mathbf{x}_{k-1}$ is on a surface and $\frac{\Pr\{t>t_{\mathbf{x}_k\mathbf{x}_{k-1}}\}}{p(t_{\mathbf{x}_k\mathbf{x}_{k-1}})}$ if it is in a medium, i.e. it equals the required value of the `aLastRaySampleRevPdfsRatio` argument.

6th (`aNextToLastPartialRevPdfW`) It is given the `bsdfRevPdfW` value computed earlier in the method. It equals $\hat{p}(\omega_{\mathbf{x}_{k-1}\mathbf{x}_{k-2}})$, i.e. the required value of the `aNextToLastPartialRevPdfW` argument.

We can see that the method is given the right arguments, so it returns the correct $w^L_{k,1,k-1}$ term and we get

$$\begin{aligned}\texttt{misWeight} &= \texttt{1.0f / (wLight + 1.0f)} \\ &= \frac{1}{w^L_{k,1,k-1} + 1} = \hat{w}_{BPT_{k-1,k}}.\end{aligned}$$

BPT: hitting a light. On the other hand, if the entire path is sampled from the camera, then there is no w^L term and the corresponding weight satisfies Equation 3.19:

$$\frac{1}{\hat{w}_{\mathrm{BPT}_{\mathrm{direct}}}} = 1 + w^C_{k,1,0}.$$

This equation is implemented in the `GetLightRadiance` method (on line 39 in Listing 3.33):

Listing 3.77: `DirectlyHitLightMis` (part of the `GetLightRadiance` method, `UPBP.hxx`)

```
1  float misWeight = 1.f;
2  if (mConnectToLightVertices)
3  {
4     const float wCamera = AccumulateCameraPathWeight(
5        aCameraState.mPathLength,
6        directPdfA,
7        0,
8        0,
9        0,
10       emissionPdfW / directPdfA
11       mQueryBeamType,
12       mPhotonBeamType,
13       mEstimatorTechniques,
14       mCameraVerticesMisData);
15
16       misWeight = 1.0f / (1.0f + wCamera);
17 }
18 else if (mAlgorithm == kPTmis && !aCameraState.mLastSpecular)
19 {
20    const float wCamera = directPdfA *
21       mCameraVerticesMisData[aCameraState.mPathLength].mPdfAInv;
22
23    misWeight = 1.0f / (1.0f + wCamera);
24 }
```

As before, the `AccumulateCameraPathWeight` method is used to compute the MIS weight of the camera subpath. Its interesting arguments are:

2nd (`aLastRevPdf`) It is given the `directPdfA` value computed earlier in the method. It equals $\overrightarrow{p_0}$, i.e. the required value of the `aLastRevPdf` argument for $s = 1$.

3rd - 5th The last vertex of the camera subpath is on a light source, therefore there is no sine or reverse ray sampling pdfs.

6th (`aNextToLastPartialRevPdfW`) It is given `emissionPdfW / directPdfA`. The factors are computed earlier in the method and their product equals:

$$\frac{p(\mathbf{x}_0)\hat{p}(\omega_{\mathbf{x}_0\mathbf{x}_1})}{p(\mathbf{x}_0)} = \hat{p}(\omega_{\mathbf{x}_0\mathbf{x}_1}),$$

i.e. the required value of the `aNextToLastPartialRevPdfW` argument.

The method is given the right arguments, so it returns the correct $w^C_{k,1,0}$ term and we get

$$\begin{aligned}\texttt{misWeight} &= \texttt{1.0f / (1.0f + wCamera)} \\ &= \frac{1}{1 + w^C_{k,1,0}} = \hat{w}_{\mathrm{BPT}_{\mathrm{direct}}}.\end{aligned}$$

Note that the `GetLightRadiance` method is also called in three distinct cases depending on estimators the renderer is configured to combine. If it uses full BPT (i.e. if `mConnectToLightVertices` is set, see Section 3.3.1), then the MIS weight is computed as presented. However, if it is allowed to use only path tracing with explicit light sampling and accumulation of emission of directly hit light sources (`mAlgorithm` is `kPTmis`) and $\mathbf{x}_1$ was not sampled specular, then there is only one technique besides the evaluated one that could create the path – connecting to the light. It is a complementary case to what the `DirectIllumination` method handles. To compute the corresponding MIS weight, we need to compute the ratio of pdfs of creating the path by connecting to the light and by sampling it entirely from the camera. It satisfies

$$\frac{\overrightarrow{p_0}\overleftarrow{p_1}}{\overleftarrow{p_0}} = \frac{\overmapsto{p_0}}{\overleftarrow{\mapsto}\,p_0}$$

and is implemented on line 20. Finally, if the renderer cannot sample light sources (`mAlgorithm` is `kPTdir`) or if $\mathbf{x}_1$ was sampled specular and therefore does not allow connecting to the light, then the path could not be created by any other technique and weighting is not needed (the weight is 1).

SURF and P-P3D. Let the path be created by applying the SURF or P-P3D estimator on vertex $\mathbf{x}_e$, $e \in \{1, \ldots, k-1\}$. Then the resulting MIS weight satisfies Equation 3.20:

$$\frac{1}{\hat{w}_{pde_e}} = w^L_{k,r,e} + w^C_{k,s,e}, \qquad r = IF^L_{pde}(e),\ s = IF^C_{pde}(e),$$

where $pde \in \{\text{SURF}, \text{P-P3D}\}$. The initialization factors equals (according to Section 3.5.1.4):

$$\begin{aligned}
IF^L_{\text{SURF}}(e) &= \frac{1}{\overleftarrow{p_e}\, F_{\text{SURF}}(e)} = \frac{K_2(\mathbf{x}_e, \mathbf{x}_e)}{\overleftarrow{p_e}}, \\
IF^C_{\text{SURF}}(e) &= \frac{1}{\overrightarrow{p_e}\, F_{\text{SURF}}(e)} = \frac{K_2(\mathbf{x}_e, \mathbf{x}_e)}{\overrightarrow{p_e}}, \\
IF^L_{\text{P-P3D}}(e) &= \frac{1}{\overleftarrow{p_e}\, F_{\text{P-P3D}}(e)} = \frac{K_3(\mathbf{x}_e, \mathbf{x}_e)}{\overleftarrow{p_e}}, \\
IF^C_{\text{P-P3D}}(e) &= \frac{1}{\overrightarrow{p_e}\, F_{\text{P-P3D}}(e)} = \frac{K_3(\mathbf{x}_e, \mathbf{x}_e)}{\overrightarrow{p_e}}.
\end{aligned}$$

These equations are implemented in the `Process` method (on line 35 in Listing 3.40):

Listing 3.78: `SurfPP3DMis` (part of the `Process` method, `UPBP.hxx`)

```
1  // (Part of) the initialization factor.
2  const float misWeightFactorInv = 1.0f /
3  (aLightVertex.mInMedium ?
4  aLightVertex.mMisData.mPP3DMisWeightFactor
5  :
6  aLightVertex.mMisData.mSurfMisWeightFactor);
7
8  // Camera part.
9  const float wCamera = mUPBP.AccumulateCameraPathWeight(
10     mCameraState.mPathLength,
11     misWeightFactorInv,
12     sinTheta,
13     aLightVertex.mMisData.mRaySamplePdfInv,
14     aLightVertex.mMisData.mRaySamplePdfsRatio,
15     cameraBsdfRevPdfW
16     mQueryBeamType,
17     mPhotonBeamType,
18     mEstimatorTechniques,
19     mCameraVerticesMisData);
20
21 // Light part.
22 const float wLight = mUPBP.AccumulateLightPathWeight(
23     aLightVertex.mPathIdx,
24     aLightVertex.mPathLength,
25     misWeightFactorInv,
26     0,
27     0,
28     0,
29     cameraBsdfDirPdfW,
30     aLightVertex.mInMedium ? PP3D : SURF
31     mQueryBeamType,
32     mPhotonBeamType,
33     mEstimatorTechniques,
34     &mPathEnds,
35     &mLightVertices,
36     NULL);
37
38 // Complete weight.
39 misWeight = 1.f / (wLight + wCamera);
```

We begin with the MIS weight of the camera subpath. The non-trivial arguments passed to the `AccumulateCameraPathWeight` method are:

2nd (`aLastRevPdf`) It is given the `misWeightFactorInv` value. For the SURF estimator it equals:

$$K_2(\mathbf{x}_e, \mathbf{x}_e) = \frac{K_2(\mathbf{x}_e, \mathbf{x}_e)\overset{\mapsto}{p_e}}{\overset{\mapsto}{p_e}} = IF^C_{\mathrm{SURF}}(e)\overset{\mapsto}{p_e},$$

i.e. the required value of the `aLastRevPdf` argument for $s = IF^C_{\mathrm{SURF}}(e)$. The case of the P-P3D estimator differs only in the kernel dimension.

3rd (`aLastSinTheta`) It is given the `sinTheta` value computed earlier in the method. It equals $\sin\theta_{\mathbf{x}_e}$, i.e. the required value of the `aLastSinTheta` argument.

4th (`aLastRaySampleRevPdfInv`) It is given the `mRaySamplePdfInv` property of the $\tilde{\mathbf{x}}_e$ light vertex. It equals $\frac{1}{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})}$, i.e. the required value of the `aLastRaySampleRevPdfInv` argument.

5th (`aLastRaySampleRevPdfsRatio`) It is given the `mRaySamplePdfsRatio` property of the $\tilde{\mathbf{x}}_e$ light vertex. It equals $\frac{\Pr\{t>t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e}\}}{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})}$, i.e. the required value of the `aLastRaySampleRevPdfsRatio` argument.

6th (`aNextToLastPartialRevPdfW`) It is given the `cameraBsdfRevPdfW` value computed earlier in the method. It equals $\hat{p}(\omega_{\mathbf{x}_e\mathbf{x}_{e+1}})$, i.e. the required value of the `aNextToLastPartialRevPdfW` argument.

Computation of the MIS weight of the light subpath is very similar. It uses the same value for the `aLastRevPdf` argument since $IF^L_{pde}(e)\overset{\leftarrow}{p_e} = IF^C_{pde}(e)\overset{\mapsto}{p_e}$ for $pde \in \{\mathrm{SURF}, \mathrm{P\text{-}P3D}\}$. The other "last" arguments are zero, as they are used only for evaluation of MIS weight contribution of photon density estimators on the last light vertex which is handled on the camera subpath.

We can see that both methods are given the right arguments, so they return the correct $w^C_{k,s,e}$ and $w^L_{k,r,e}$ terms and we get

$$\begin{aligned}\texttt{misWeight} &= \texttt{1.0f / (wLight + wCamera)}\\ &= \frac{1}{w^L_{k,r,e} + w^C_{k,s,e}} = \hat{w}_{pde_e}\end{aligned}$$

for $r = IF^L_{pde}(e), s = IF^C_{pde}(e)$ and $pde \in \{\mathrm{SURF}, \mathrm{P\text{-}P3D}\}$.

P-B2D. If the path is created by applying the P-B2D estimator on vertex $\mathbf{x}_e$, $e \in \{1, \ldots, k-1\}$, the resulting MIS weight satisfies the same Equation 3.20 but for different initialization factors

$$r = IF^L_{\text{P-B}_\text{l}\text{2D}}(e) = \frac{1}{\overleftarrow{p}_e F_{\text{P-B}_\text{l}\text{2D}}(e)} = \frac{p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_2(\mathbf{x}_e, \mathbf{x}_e)}{\overleftarrow{p}_e},$$
$$s = IF^C_{\text{P-B}_\text{l}\text{2D}}(e) = \frac{1}{\overrightarrow{p}_e F_{\text{P-B}_\text{l}\text{2D}}(e)} = \frac{p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_2(\mathbf{x}_e, \mathbf{x}_e)}{\overrightarrow{p}_e},$$

for long query beams and

$$r = IF^L_{\text{P-B}_\text{s}\text{2D}}(e) = \frac{1}{\overleftarrow{p}_e F_{\text{P-B}_\text{s}\text{2D}}(e)} = \frac{p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_2(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\overleftarrow{p}_e},$$
$$s = IF^C_{\text{P-B}_\text{s}\text{2D}}(e) = \frac{1}{\overrightarrow{p}_e F_{\text{P-B}_\text{s}\text{2D}}(e)} = \frac{p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_2(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\overrightarrow{p}_e},$$

for short query beams. An implementation of these equations is contained in the `breIntersectFuncHomogeneous2` method (on line 12 in Listing 3.48):

Listing 3.79: `PB2DMis` (part of the `breIntersectFuncHomogeneous2` method, `Bre.cxx`)

```
1  // Create temporary camera vertex on the query beam.
2  const float distSq = Utils::sqr(photonIsectDist);
3  const float raySamplePdfInv = 1.0f / raySamplePdf;
4
5  MisData* cameraVerticesMisData =
6      static_cast<MisData*>(data->mCameraVerticesMisData);
7  MisData& cameraVertexMisData =
8      cameraVerticesMisData[data->mCameraPathLength].
9
10 cameraVertexMisData.mPdfAInv = data->mLastPdfWInv * distSq * raySamplePdfInv;
11 cameraVertexMisData.mRaySamplePdfInv = raySamplePdfInv;
12 cameraVertexMisData.mRaySamplePdfsRatio = raySamplePdfsRatio;
13 cameraVertexMisData.mSurfMisWeightFactor = 0;
14 cameraVertexMisData.mPP3DMisWeightFactor = data->mPP3DMisWeightFactor;
15 cameraVertexMisData.mPB2DMisWeightFactor = data->mPB2DMisWeightFactor;
16 cameraVertexMisData.mBB1DMisWeightFactor = data->mBB1DMisWeightFactor;
17 cameraVertexMisData.mIsDelta = false;
18 cameraVertexMisData.mIsOnLightSource = false;
19 cameraVertexMisData.mIsSpecular = false;
20
21 cameraVerticesMisData[data->mCameraPathLength - 1].mRaySampleRevPdfInv =
22     1.0f / raySampleRevPdf;
23
24 // (Part of) the initialization factor.
25 const float last = (ray.flags & SHORT_BEAM) ?
26     1.0 / (raySamplePdfsRatio * cameraVertexMisData.mPB2DMisWeightFactor) :
27     raySamplePdf / cameraVertexMisData.mPB2DMisWeightFactor;
28
29 // Camera part.
30 const float wCamera = AccumulateCameraPathWeight(
31     data->mCameraPathLength,
32     last,
33     sinTheta,
34     lightVertex->mMisData.mRaySamplePdfInv,
35     lightVertex->mMisData.mRaySamplePdfsRatio,
36     cameraBsdfRevPdfW * raySampleRevPdf / distSq,
37     data->mQueryBeamType,
38     data->mPhotonBeamType,
39     ray.flags,
40     cameraVerticesMis);
41
42 ..PB2DMisPart2..
```

Listing 3.80: `PB2DMisPart2` (part of the `breIntersectFuncHomogeneous2` method, `Bre.cxx`)

```
// Light part.
const float wLight = AccumulateLightPathWeight(
    lightVertex->mPathIdx,
    lightVertex->mPathLength,
    last,
    0,
    0,
    0,
    cameraBsdfDirPdfW,
    PB2D,
    data->mQueryBeamType,
    data->mPhotonBeamType,
    ray.flags,
    static_cast<std::vector<int>*>(data->mPathEnds),
    static_cast<std::vector<UPBPLightVertex>*>(data->mLightVertices)
    NULL);

// Complete weight.
const float misWeight = 1.f / (wLight + wCamera);
```

To compute the MIS weight of the camera subpath we need `MisData` for the $\mathbf{x}_e$ camera vertex. However, we do not have these data as the last camera vertex created during tracing of the camera subpath is $\mathbf{x}_{e+1}$. We therefore compute them and set them temporarily in the `mCameraVerticesMisData` array. After this method is finished, they are overwritten by every subsequent P-B2D (and B-B1D) estimator evaluation along the same query ray and then, finally, when the camera subpath continues. This applies to the modification of the reverse pdf of $\mathbf{x}_{e+1}$ too.

Properties of the `MisData` structure for $\mathbf{x}_e$ are computed in a very similar way to creating a new camera vertex during tracing the camera subpath, see Listing 3.66. Setting of some of them is missing but those are passed separately via arguments of the `AccumulateCameraPathWeight` method. These are:

2nd (`aLastRevPdf`) It is given the `last` value. For long query beams it equals:

$$\begin{aligned}&\texttt{raySamplePdf / cameraVertexMisData.mPB2DMisWeightFactor}\\&= p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_2(\mathbf{x}_e,\mathbf{x}_e) = \frac{p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_2(\mathbf{x}_e,\mathbf{x}_e)}{\overset{\mapsto}{p}_e}\overset{\mapsto}{p}_e = IF^C_{\text{P-B}_\text{l}\text{2D}}(e)\overset{\mapsto}{p}_e,\end{aligned}$$

i.e. the required value of the `aLastRevPdf` argument for $s = IF^C_{\text{P-B}_\text{l}\text{2D}}(e)$. For short query beams it equals:

$$\begin{aligned}&\texttt{raySamplePdfsRatio / cameraVertexMisData.mPB2DMisWeightFactor}\\&= \frac{p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})}{\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}}K_2(\mathbf{x}_e,\mathbf{x}_e) = \frac{p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_2(\mathbf{x}_e,\mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\overset{\mapsto}{p}_e}\overset{\mapsto}{p}_e = IF^C_{\text{P-B}_\text{l}\text{2D}}(e)\overset{\mapsto}{p}_e,\end{aligned}$$

i.e. the required value of the `aLastRevPdf` argument for $s = IF^C_{\text{P-B}_\text{s}\text{2D}}(e)$.

3rd (`aLastSinTheta`) It is given the `sinTheta` value computed earlier in the method. It equals $\sin\theta_{\mathbf{x}_e}$, i.e. the required value of the `aLastSinTheta` argument.

4th (`aLastRaySampleRevPdfInv`) It is given the `mRaySamplePdfInv` property of the $\tilde{\mathbf{x}}_e$ light vertex. It equals $\frac{1}{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})}$, i.e. the required value of the `aLastRaySampleRevPdfInv` argument.

5th (`aLastRaySampleRevPdfsRatio`) It is given the `mRaySamplePdfsRatio` property of the $\tilde{\mathbf{x}}_e$ light vertex. It equals $\frac{\Pr\{t>t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e}\}}{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})}$, i.e. the required value of the `aLastRaySampleRevPdfsRatio` argument.

6th (`aNextToLastPartialRevPdfW`) It is given

$$\begin{aligned}&\texttt{cameraBsdfRevPdfW * raySampleRevPdf / distSq}\\&= \hat{p}(\omega_{\mathbf{x}_e\mathbf{x}_{e+1}})\frac{p(t_{\mathbf{x}_e\mathbf{x}_{e+1}})}{|\mathbf{x}_e-\mathbf{x}_{e+1}|^2}.\end{aligned}$$

In case of the previous estimators the `aNextToLastPartialRevPdfW` argument is given only the pdf of sampling the scattering function. However, the $\mathbf{x}_{e+1}$ vertex there has already included factor $p(t_{\mathbf{x}_e\mathbf{x}_{e+1}})/|\mathbf{x}_e-\mathbf{x}_{e+1}|^2$ in its `mRevPdf` property, since the $\mathbf{x}_e$ vertex is not temporary. Here we therefore have to include it in the method argument.

Computation of the MIS weight of the light subpath is very similar. It uses the same value for the `aLastRevPdf` argument since $IF^L_{pde}(e)\overleftarrow{p_e} = IF^C_{pde}(e)\overrightarrow{p_e}$ for $pde \in \{\text{P-B}_\text{l}\text{2D}, \text{P-B}_\text{s}\text{2D}\}$. The other "last" arguments are zero, as they are used only for evaluation of MIS weight contribution of photon density estimators on the last light vertex which is handled on the camera subpath.

We can see that both methods are given the right arguments, so they return the correct $w^C_{k,s,e}$ and $w^L_{k,r,e}$ terms and we get

$$\begin{aligned}\texttt{misWeight} &= \texttt{1.0f / (wLight + wCamera)}\\&= \frac{1}{w^L_{k,r,e}+w^C_{k,s,e}} = \hat{w}_{pde_e}\end{aligned}$$

for $r = IF^L_{pde}(e), s = IF^C_{pde}(e)$ and $pde \in \{\text{P-B}_\text{l}\text{2D}, \text{P-B}_\text{s}\text{2D}\}$.

B-B1D. And finally, if the path is created by applying the B-B1D estimator on vertex $\mathbf{x}_e$, $e \in \{1, \ldots, k-1\}$, the resulting MIS weight satisfies the Equation 3.20 with initialization factors

$$IF^L_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(e) = \frac{1}{\overleftarrow{p}_e F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overleftarrow{p}_e},$$

$$IF^C_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(e) = \frac{1}{\overrightarrow{p}_e F_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overrightarrow{p}_e},$$

$$IF^L_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(e) = \frac{1}{\overleftarrow{p}_e F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overleftarrow{p}_e},$$

$$IF^C_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(e) = \frac{1}{\overrightarrow{p}_e F_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overrightarrow{p}_e},$$

$$IF^L_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(e) = \frac{1}{\overleftarrow{p}_e F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overleftarrow{p}_e},$$

$$IF^C_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(e) = \frac{1}{\overrightarrow{p}_e F_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overrightarrow{p}_e},$$

$$IF^L_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(e) = \frac{1}{\overleftarrow{p}_e F_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e}\}\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overleftarrow{p}_e},$$

$$IF^C_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(e) = \frac{1}{\overrightarrow{p}_e F_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(e)} = \frac{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})p(t_{\mathbf{x}_{e+1}\mathbf{x}_e})K_1(\mathbf{x}_e, \mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e}\}\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}\overrightarrow{p}_e}.$$

An implementation of these equations is contained in the `accumulate2` method (on line 18 in Listing 3.58):

Listing 3.81: `BB1DMis` (part of the `accumulate2` method, `PhotonBeam.hxx`)

```
1  // Create temporary camera vertex on the query beam.
2
3  const float distSqQuery = Utils::sqr(queryIsectDist);
4  const float raySamplePdfInvQuery = 1.0f / raySamplePdfQuery;
5  MisData* cameraVerticesMisData =
6      static_cast<MisData*>(additionalDataForMis->mCameraVerticesMisData);
7  MisData& cameraVertexMisData =
8      cameraVerticesMisData[additionalDataForMis->mCameraPathLength];
9  const AdditionalRayDataForMis* data = additionalDataForMis;
10
11 cameraVertexMisData.mPdfAInv =
12     data->mLastPdfWInv * distSqQuery * raySamplePdfInvQuery;
13 cameraVertexMisData.mRaySamplePdfInv = raySamplePdfInvQuery;
14 cameraVertexMisData.mRaySamplePdfsRatio = raySamplePdfsRatioQuery;
15 cameraVertexMisData.mSurfMisWeightFactor = 0;
16 cameraVertexMisData.mPP3DMisWeightFactor = data->mPP3DMisWeightFactor;
17 cameraVertexMisData.mPB2DMisWeightFactor =   data->mPB2DMisWeightFactor;
18 cameraVertexMisData.mBB1DMisWeightFactor =   data->mBB1DMisWeightFactor;
19 cameraVertexMisData.mIsDelta = false;
20 cameraVertexMisData.mIsOnLightSource = false;
21 cameraVertexMisData.mIsSpecular = false;
22
23 cameraVerticesMisData[data->mCameraPathLength - 1].mRaySampleRevPdfInv =
24     1.0f / raySampleRevPdfQuery;
25
26 ..BB1DMisPart2..
27 ..BB1DMisPart3..
```

Listing 3.82: `BB1DMisPart2` (part of the `accumulate2` method, `PhotonBeam.hxx`)

```
// Create temporary light vertex on the photon beam.

const float distSqBeam = mLightVertex->mIsFinite ?
    Utils::sqr(beamIsectDist) : 1.0f;
const float raySamplePdfInvBeam = 1.0f / raySamplePdfBeam;
MisData beamLightVertexMisData;

beamLightVertexMisData.mPdfAInv =
    mLastPdfWInv * distSqBeam * raySamplePdfInvBeam;
beamLightVertexMisData.mRaySamplePdfInv = raySamplePdfInvBeam;
beamLightVertexMisData.mRaySamplePdfsRatio = raySamplePdfsRatioBeam;
beamLightVertexMisData.mSurfMisWeightFactor = 0;
beamLightVertexMisData.mPP3DMisWeightFactor = data->mPP3DMisWeightFactor;
beamLightVertexMisData.mPB2DMisWeightFactor = data->mPB2DMisWeightFactor;
beamLightVertexMisData.mBB1DMisWeightFactor = data->mBB1DMisWeightFactor;
beamLightVertexMisData.mIsDelta = false;
beamLightVertexMisData.mIsOnLightSource = false;
beamLightVertexMisData.mIsSpecular = false;

// We need to modify value of the previous light vertex pdf but
// it may be still needed. We therefore back it up, modify,
// use and then, at the end, restore from the backup.
const float originRaySampleRevPdfInvBackup =
    mLightVertex->mMisData.mRaySampleRevPdfInv;
mLightVertex->mMisData.mRaySampleRevPdfInv = 1.0f / raySampleRevPdfBeam;

// (Part of) the initialization factor.
const float last = (rayFlags & SHORT_BEAM) ?
    (
        (mFlags & SHORT_BEAM) ?
        1.0 / (raySamplePdfsRatioQuery * raySamplePdfsRatioBeam
        * cameraVertexMisData.mBB1DMisWeightFactor * sinTheta)
        :
        raySamplePdfBeam / (raySamplePdfsRatioQuery
        * cameraVertexMisData.mBB1DMisWeightFactor * sinTheta)
    )
    :
    (
        (mFlags & SHORT_BEAM) ?
        raySamplePdfQuery / (raySamplePdfsRatioBeam
        * cameraVertexMisData.mBB1DMisWeightFactor * sinTheta)
        :
        raySamplePdfQuery * raySamplePdfBeam
        / (cameraVertexMisData.mBB1DMisWeightFactor * sinTheta)
    );

// Camera part.
const float wCamera = AccumulateCameraPathWeight(
    data->mCameraPathLength,
    last,
    sinTheta,
    raySamplePdfInvBeam,
    raySamplePdfsRatioBeam,
    bsdfRevPdfW * raySampleRevPdfQuery / distSqQuery,
    data->mQueryBeamType,
    data->mPhotonBeamType,
    rayFlags,
    cameraVerticesMisData);
```

Listing 3.83: `BB1DMisPart3` (part of the `accumulate2` method, `PhotonBeam.hxx`)

```
// Light part.
const float wLight = AccumulateLightPathWeight(
    mLightVertex->mPathIdx,
    mLightVertex->mPathLength + 1,
    last,
    0,
    0,
    0,
    (mLightVertex->mMisData.mIsOnLightSource && mLightVertex->mMisData.mIsDelta)
        ? 0 :
        bsdfDirPdfW * raySampleRevPdfBeam
        * std::abs(mLightVertex->mMisData.mCosThetaOut) / distSqBeam,
    BB1D,
    data->mQueryBeamType,
    data->mPhotonBeamType,
    rayFlags,
    static_cast<std::vector<int>*>(data->mPathEnds),
    static_cast<std::vector<UPBPLightVertex>*>(data->mLightVertices),
    &beamLightVertexMisData);

// Complete weight.
const float misWeight = 1.f / (wLight + wCamera);

// Restore the modified value of the previous light vertex pdf.
mLightVertex->mMisData.mRaySampleRevPdfInv = originRaySampleRevPdfInvBackup;
```

To compute the MIS weight of the camera subpath we need `MisData` for the $\mathbf{x}_e$ camera vertex. However, we do not have these data as the last camera vertex created during tracing of the camera subpath is $\mathbf{x}_{e+1}$. We therefore compute them and set them temporarily in the `mCameraVerticesMisData` array. After this method is finished, they are overwritten by every subsequent B-B1D estimator evaluation along the same query ray and then, finally, when the camera subpath continues. This applies to the modification of the reverse pdf of $\mathbf{x}_{e+1}$ too.

Properties of the `MisData` structure for $\mathbf{x}_e$ are computed in a very similar way to creating a new camera vertex during tracing the camera subpath, see Listing 3.66. Setting of some of them is missing but those are passed separately via arguments of the `AccumulateCameraPathWeight` method. These are:

2nd (`aLastRevPdf`) It is given the `last` value. Depending on the query and photon beam types it equals:

$$\text{LB-LB: } \frac{p(t_{\mathbf{x}_{e-1}\mathbf{x}_e})p(t_{\mathbf{x}_{e+1}\tilde{\mathbf{x}}_e})K_1(\mathbf{x}_e,\mathbf{x}_e)}{\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}} = IF^C_{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}}(e)\overrightarrow{p}_e,$$

$$\text{LB-SB: } \frac{p(t_{\mathbf{x}_{e-1}\mathbf{x}_e})p(t_{\mathbf{x}_{e+1}\tilde{\mathbf{x}}_e})K_1(\mathbf{x}_e,\mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}} = IF^C_{\text{B}_\text{l}\text{-B}_\text{s}\text{1D}}(e)\overrightarrow{p}_e,$$

$$\text{SB-LB: } \frac{p(t_{\mathbf{x}_{e-1}\mathbf{x}_e})p(t_{\mathbf{x}_{e+1}\tilde{\mathbf{x}}_e})K_1(\mathbf{x}_e,\mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e-1}\mathbf{x}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}} = IF^C_{\text{B}_\text{s}\text{-B}_\text{l}\text{1D}}(e)\overrightarrow{p}_e,$$

$$\text{SB-SB: } \frac{p(t_{\mathbf{x}_{e-1}\mathbf{x}_e})p(t_{\mathbf{x}_{e+1}\tilde{\mathbf{x}}_e})K_1(\mathbf{x}_e,\mathbf{x}_e)}{\Pr\{t > t_{\mathbf{x}_{e-1}\mathbf{x}_e}\}\Pr\{t > t_{\mathbf{x}_{e+1}\mathbf{x}_e}\}\sin\theta_{\mathbf{x}_{e-1}\mathbf{x}_{e+1}}} = IF^C_{\text{B}_\text{s}\text{-B}_\text{s}\text{1D}}(e)\overrightarrow{p}_e,$$

i.e. the required value of the `aLastRevPdf` argument for $s = IF^C_{pde}(e)$ where $pde \in \{\text{B}_\text{l}\text{-B}_\text{l}\text{1D}, \text{B}_\text{l}\text{-B}_\text{s}\text{1D}, \text{B}_\text{s}\text{-B}_\text{l}\text{1D}, \text{B}_\text{s}\text{-B}_\text{s}\text{1D}\}$.

3rd (`aLastSinTheta`) It is given the `sinTheta` value computed earlier in the method. It equals $\sin\theta_{\mathbf{x}_e}$, i.e. the required value of the `aLastSinTheta` argument.

4th (`aLastRaySampleRevPdfInv`) It is given the `raySamplePdfInvBeam` value computed earlier in the method. It equals $\frac{1}{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})}$, i.e. the required value of the `aLastRaySampleRevPdfInv` argument.

5th (`aLastRaySampleRevPdfsRatio`) It is given the `raySamplePdfsRatioBeam` value computed earlier in the method. It equals $\frac{\Pr\{t>t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e}\}}{p(t_{\mathbf{x}_{e-1}\tilde{\mathbf{x}}_e})}$, i.e. the required value of the `aLastRaySampleRevPdfsRatio` argument.

6th (`aNextToLastPartialRevPdfW`) It is given

$$\begin{aligned}&\texttt{bsdfRevPdfW * raySampleRevPdfQuery / distSqQuery}\\&= \hat{p}(\omega_{\mathbf{x}_e\mathbf{x}_{e+1}})\frac{p(t_{\mathbf{x}_e\mathbf{x}_{e+1}})}{|\mathbf{x}_e-\mathbf{x}_{e+1}|^2}.\end{aligned}$$

In case of the previous estimators the `aNextToLastPartialRevPdfW` argument is given only the pdf of sampling the scattering function. However, the $\mathbf{x}_{e+1}$ vertex there has already included factor $p(t_{\mathbf{x}_e\mathbf{x}_{e+1}})/|\mathbf{x}_e-\mathbf{x}_{e+1}|^2$ in its `mRevPdf` property, since the $\mathbf{x}_e$ vertex is not temporary. Here we therefore have to include it in the method argument.

Computation of the MIS weight of the light subpath is very similar. It uses the same value for the `aLastRevPdf` argument since $IF^L_{pde}(e)\overleftarrow{p}_e = IF^C_{pde}(e)\overrightarrow{p}_e$ for $pde \in \{\mathrm{B_l\text{-}B_l1D}, \mathrm{B_l\text{-}B_s1D}, \mathrm{B_s\text{-}B_l1D}, \mathrm{B_s\text{-}B_s1D}\}$. The other "last" arguments are zero, as they are used only for evaluation of MIS weight contribution of photon density estimators on the last light vertex which is handled on the camera subpath. We also have to create a temporary `MisData` structure for the $\tilde{\mathbf{x}}_e$ light vertex (`beamLightVertexMisData`). Because we cannot modify data of stored light vertices, the structure is sent as a separate argument. And there is one more special value expected by the `AccumulateLightPathWeight` method. The `aNextToLastPartialRevPdfW` argument is given the whole $\overleftarrow{p}_{e-1}$ pdf (zero of course, if $e = 1$ and $\mathbf{x}_{e-1}$ is on a delta light source, since there is no chance hitting it).

We can see that both methods are given the right arguments, so they return the correct $w^C_{k,s,e}$ and $w^L_{k,r,e}$ terms and we get

$$\begin{aligned}\texttt{misWeight} &= \texttt{1.0f / (wLight + wCamera)}\\&= \frac{1}{w^L_{k,r,e}+w^C_{k,s,e}} = \hat{w}_{pde_e}\end{aligned}$$

for $r = IF^L_{pde}(e), s = IF^C_{pde}(e)$, $pde \in \{\mathrm{B_l\text{-}B_l1D}, \mathrm{B_l\text{-}B_s1D}, \mathrm{B_s\text{-}B_l1D}, \mathrm{B_s\text{-}B_s1D}\}$.

3.5.3 Summary

The aim of the Section 3.5 was to carefully explain all aspects of the computation of MIS weights. It began with theoretical derivation of the algorithm and then proceeded to its implementation. First, necessary data were introduced, what we need to store and how it is obtained. Then the methods realizing the algorithm itself were presented closely. Finally, the section listed how these methods are called to compute MIS weights for each estimator. We made this part more detailed than the others as we believe that the computation of MIS weights is crucial.

4. Results

This chapter presents results our implementation of the UPBP algorithm produces. It mainly contains modified and extended text of Section 9 in [14].

Since the main motivation for development of the UPBP algorithm was simulation of light transport in participating media, we test performance of the UPBP algorithm by comparing it to the individual volumetric photon density estimators (P-P3D, P-B2D, B-B1D) and to bidirectional path tracing (BPT). The SURF estimator is not included in the comparison (neither individually nor as a part of UPBP) because its contribution in our tests targeted on media would be negligible.

4.1 Comparison setup

We begin with an overview of our comparison setup. We explain what query and photon beams were used, which rendering modes were compared or how we set the number of subpaths for generating photon beams.

Long and short beams. As we already mentioned in Section 3.4, combining the long- and short-beam variants of the same estimators (i.e. all six variants of P-B2D and B-B1D as listed in Section 1.2.1) would not be useful because the long-beam variant always has less variance (as shown by Křivánek et al. [14]). On the other hand, evaluating the long-beam estimators is obviously more costly, so a judicious choice needs to be made. In our tests, the use of short photon beams and long query beams provided the best performance for a given time budget. Therefore, we set the UPBP algorithm to use only long query beams and short photon beams and compare it only to the P-B_l2D and B_s-B_l1D estimator variants. Opting for short photon beams is important for the overall performance, because long photon beams have much higher cost of queries and construction of an acceleration data structure, and this overhead is rarely compensated by a corresponding variance reduction.

Compatible UPBP. Because the volumetric photon density estimators are designed to capture only medium transport, we perform the comparison for the UPBP algorithm set in "compatible" rendering mode (described in Section 3.3.1.1). In this mode it simulates only a subset of light transport paths corresponding to what these estimators are able to capture while preserving extended possibilities of sampling the paths.

The individual estimators. To produce results of the individual volumetric photon density estimators and BPT for the comparison we did not use any separate implementation of these estimators. We instead utilized the flexibility of our implementation and set it to evaluate only a desired estimator. Moreover, BPT is run in "compatible" mode to calculate only medium transport (for the same reason as UPBP) and the volumetric photon density estimators are run in "previous" mode (described in Section 3.3.1.1) to emulate the previous work [13, 9, 10]. Note that this emulation affects only traced camera subpaths. Our P-P3D estimator

does *not* use ray marching as suggested by Jensen and Christensen [13], so that the implementation is consistent with the derived theory. Ray marching would correspond to evaluating more P-P3D than P-B_l2D and B_s-B_l1D estimators (one evaluation per ray-marching step), which would complicate the interpretation of the results.

Photon beams count. We sample fewer light subpaths for generating photon beams than for generating photon points (see Table 4.1). We set the number of subpaths such that the total rendering time spent on evaluating the B_s-B_l1D estimator is about the same as the time spent on the other estimators. While this simple heuristic works well in our scenes, a more systematic analysis of estimator efficiency is an important avenue for future work.

Reference images. Besides running the UPBP algorithm in "compatible" mode we also rendered reference images capturing full light transport. The purpose of these images is to present the employed scenes and demonstrate capabilities of the algorithm. They should not be compared with other images neither by their content nor rendering time (their rendering ran until convergence). Apart from rendering mode and time there is one more difference in rendering configuration of the reference images. They are the only case in which evaluation of the SURF estimator is included and as such also the only case when radius reduction is applied. The radius for the SURF estimator is reduced according to Equation 3.7 with $\alpha = 0.75$ (as recommended in [4]), other radii are constant.

PC configuration. All the tests were run on a Windows 7 PC with a 4-core Intel i7-2600K CPU and 16GB of RAM using 8 threads.

4.2 Scenes

To demonstrate the robustness of the UPBP method, we render three scenes containing media with a wide range of parameters, featuring complex specular-medium-specular transport. The scenes are called Still life, Mirror balls and Bathroom, the rendering settings are shown in Table 4.1, parameters of the media in these scenes are listed in Table 4.2. For each scene we present one reference image capturing full light transport and a series of images comparing contributions of the individual volumetric estimators and BPT to our method.

Still life. The Still life scene in Figure 4.1 features different kinds of media (from left to right: wax candle, glycerin soap bar on top of a block of a back-scattering medium, diluted wine, apple juice, and olive oil). Figure 4.1 compares the result of our UPBP algorithm to the previous methods, implemented as described above. These results are in line with the observation Křivánek et al. [14] made in his canonical variance analysis, that beams (B_s-B_l1D) are not necessarily more efficient than the point-based estimators P-P3D and P-B_l2D. For instance, P-B_l2D renders a less noisy image than B_s-B_l1D for the dense wax, while the opposite is true for the thinner glycerin soap. The unbiased BPT techniques efficiently capture the light transport in the thin diluted wine and the apple juice. No single technique is

able to efficiently render all the media in this scene, while our combined algorithm performs well.

Mirror balls. The Mirror balls scene in Figure 4.2 shows the good performance of photon beams at rendering (reflected) caustics in thin media. The number of photon beams per iteration is only 0.63% of the number of photons. Despite this significant difference in the numbers of samples, the variance of B_s-B_l1D in the thin medium that fills the space is very low. Although BPT is efficient at rendering volumetric caustics, their reflections are more efficiently captured by B_s-B_l1D. P-B_l2D and P-P3D produce good results in the two spheres with a dense medium. However, the variance of P-P3D in the thin medium is enormous, while P-B_l2D performs nearly as well as B_s-B_l1D. Our combined algorithm is able to take the best of all these estimators to produce a superior result.

Bathroom. The Bathroom scene in Figure 4.3 has similar settings to the Mirror balls scene and the various estimators show similar performance. The B_s-B_l1D estimator excels at capturing the focused light around the complex light fixture, while having high variance in the thick media of the flask contents and the washbasin. Our combined algorithm still has an edge over P-B_l2D, the best-performing previous method, though the advantage of the combined estimator is nearly offset by the overhead of evaluating many techniques.

	Still life	**Mirror balls**	**Bathroom**
Image resolution	1600 × 700	800 × 800	672 × 896
# light subpaths	1,120,000	640,000	602,112
# photon beam subpaths	23,000 (2.05%)	4,000 (0.63%)	3,000 (0.50%)
Maximum path length	80	12	20
Rendering time	25 min	60 min	60 min
# iterations UPBP	752	665	497
# iterations BPT	1315	1962	973
# iterations P-P3D	2319	2367	2493
# iterations B-P2D	2096	1931	2018
# iterations B-B1D	2218	1331	1372

Table 4.1: Rendering settings and statistics.

Medium	**Scattering coefficient σ_s**			**Absorption coefficient σ_a**		
	R	**G**	**B**	**R**	**G**	**B**
Wax candle	1.5000	1.5000	1.5000	0.0300	0.1000	0.2000
Glycerin soap (top)	0.0201	0.0202	0.0221	0.0020	0.0040	0.0002
Block (bottom)	0.0100	0.0100	0.0100	0.0100	0.0100	0.0100
Diluted wine	0.0150	0.0130	0.0111	0.1220	0.3510	0.4020
Apple juice	0.0201	0.0243	0.0323	0.1014	0.1862	0.4084
Olive oil	0.0410	0.0390	0.0120	0.0620	0.0470	0.3530
Global medium (fog)	0.0002	0.0002	0.0001	0.0001	0.0001	0.0001
Green glass sphere	0.0004	0.0001	0.0004	0.0200	0.0010	0.0200
Orange sphere	0.1000	0.1000	0.1000	0.0050	0.0600	0.2600
Dark amethyst	0.0600	0.1000	0.1000	0.0001	0.0100	0.0100

Medium	**Single-scattering albedo**			**Mean free path**	g
	R	**G**	**B**		
Wax candle	0.9803	0.9615	0.7500	0.032	0.8
Glycerin soap (top)	0.9090	0.8347	0.9910	2.300	0.6
Block (bottom)	0.5000	0.5000	0.5000	2.500	-0.9
Diluted wine	0.1094	0.0357	0.0268	0.365	0.9
Apple juice	0.1654	0.1154	0.0732	0.412	0.9
Olive oil	0.0042	0.4535	0.0995	0.581	0.9
Global medium (fog)	0.6667	0.600	0.5000	5.000	0.5
Green glass sphere	0.0196	0.0909	0.0196	12.62	0.0
Orange sphere	0.6250	0.9523	0.2777	0.043	-0.9
Dark amethyst	0.5964	0.0909	0.0909	0.100	-0.3

Table 4.2: Medium parameters for the Still life (top parts) and the Mirror balls scenes (bottom parts). Media are listed in the order in which they appear in the respective scenes from left to right. The mean free path value of the media is given here as a multiple of the largest side of the enclosing object's bounding box. The last column gives the mean cosine of the Henyey-Greenstein phase function.

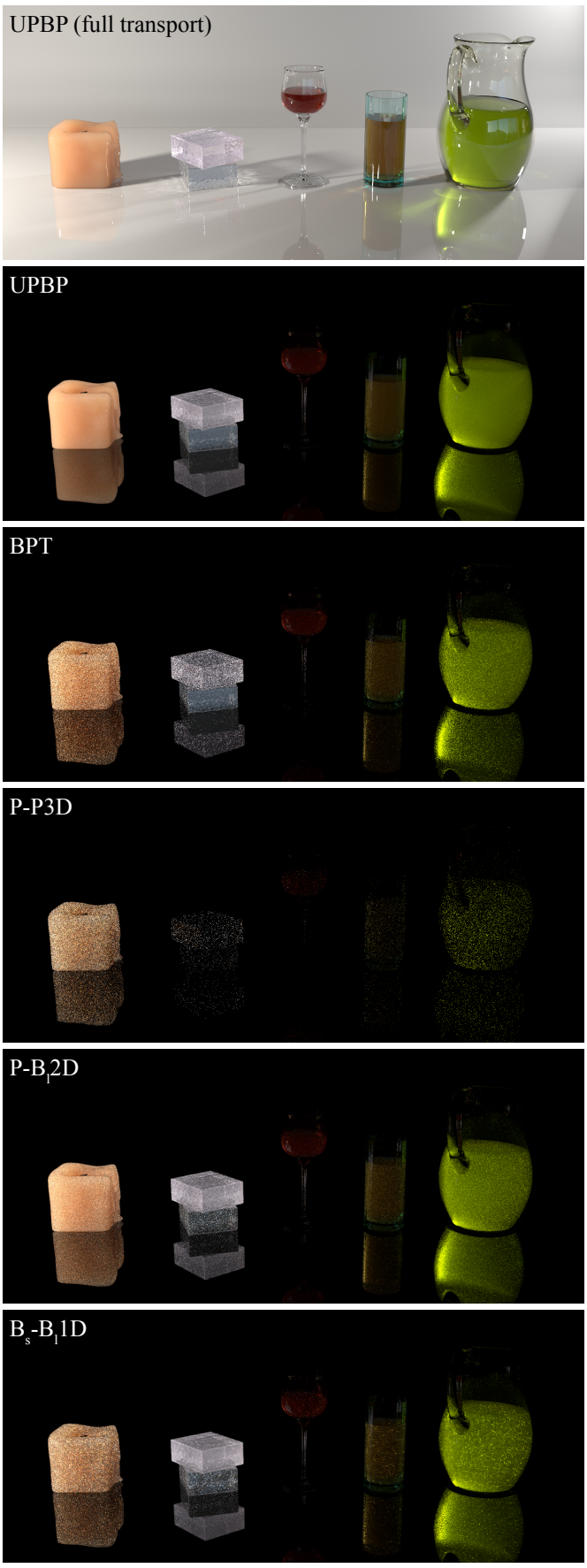


Figure 4.1: Results for the Still life scene.

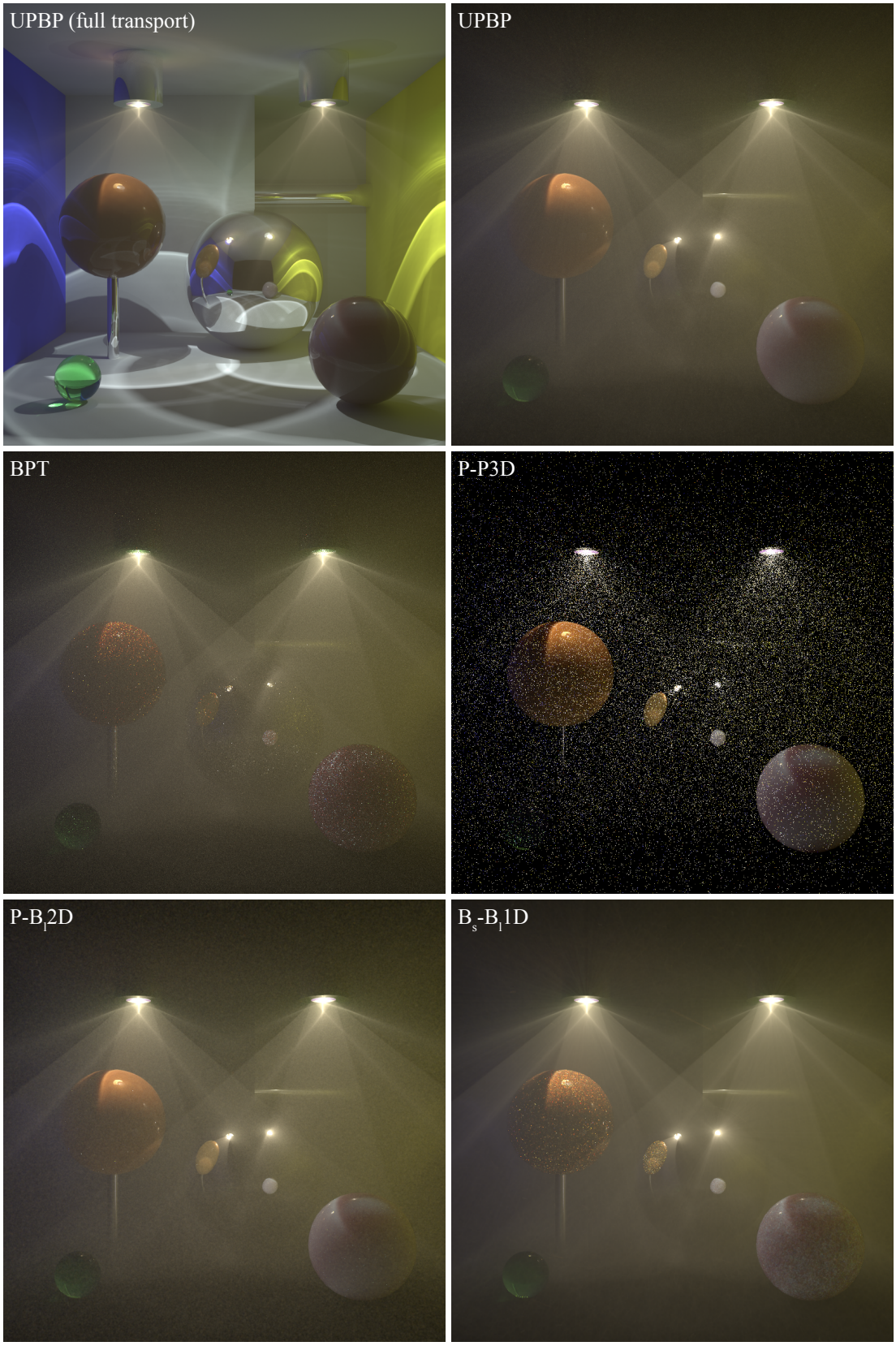


Figure 4.2: Results for the Mirror balls scene.

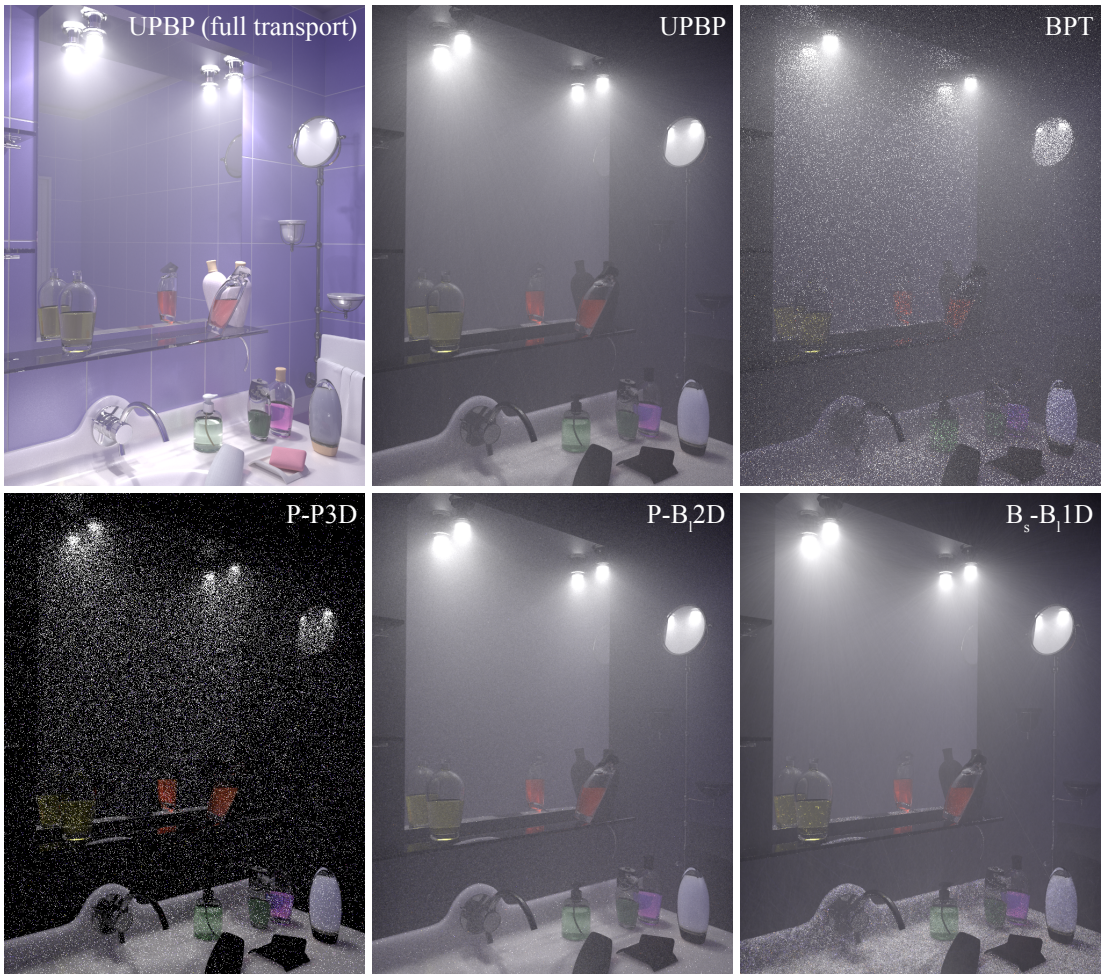


Figure 4.3: Results for the Bathroom scene.

Conclusion

Our goal was to implement the UPBP algorithm. Although we did not have to start from scratch, it was still a challenging task. We implemented media representation and phase function evaluation, rewrote the ray-scene intersection function and figured out how to track what the current medium is. We incorporated query and photon beams handling and, above all, provided a fast and stable algorithm for the computation of MIS weights.

The resulting renderer empirically proved that the UPBP algorithm is working. We could see that it automatically exploits the complementary benefits of each of the included techniques to provide a combination more robust to scene variations and lighting scenarios than any single previous technique alone. We decided to release it for public so as to everybody could take advantage of it. Mainly computer graphics researchers are expected to work with it but commercial usage is allowed as well.

We have several suggestions for future work. Media support could be extended for heterogeneous and emissive media to broaden the area of application, photon beams data structure should be revised to improve performance of the B-B1D estimator. Except for the SURF estimator, a constant kernel radius is now used. A comprehensive asymptotic analysis of the trade-off between variance and bias would yield the appropriate radius reduction scheme and make the combined algorithm consistent.

We were given a chance to collaborate on development of the UPBP algorithm, a new state of the art rendering algorithm in realistic image synthesis. We managed to create its implementation but that is only a beginning. There is a plenty of room for further research, improvements and extensions and anybody can experiment with it. This thesis provides knowledge needed to start.

Bibliography

[1] *Apache 2.0 license* [online]. [cit. 27. 11. 2014]. Dostupné z: `http://www.apache.org/licenses/LICENSE-2.0`.

[2] *Modified BSD license* [online]. [cit. 27. 11. 2014]. Dostupné z: `http://www.openexr.com/license.html`.

[3] *Embree* [online]. [cit. 3. 7. 2014]. Dostupné z: `http://embree.github.io`.

[4] GEORGIEV, I. et al. Light transport simulation with vertex connection and merging. *ACM Trans. Graph. (Proc. of SIGGRAPH Asia)*. 2012, 31, 6. ISSN 0730-0301. doi: 10.1145/2366145.2366211.

[5] GEORGIEV, I. et al. Joint Importance Sampling of Low-Order Volumetric Scattering. *ACM Trans. Graph. (Proc. of SIGGRAPH Asia)*. Nov. 2013, 32, 6.

[6] HACHISUKA, T.; OGAKI, S.; JENSEN, H. W. Progressive Photon Mapping. *ACM Trans. Graph. (Proc. of SIGGRAPH Asia)*. 2008, 27, 5.

[7] HACHISUKA, T.; PANTALEONI, J.; JENSEN, H. W. A path space extension for robust light transport simulation. *ACM Trans. Graph. (Proc. of SIGGRAPH Asia)*. November 2012, 31, 6. ISSN 0730-0301. doi: 10.1145/2366145.2366210.

[8] *PCG HDR Image Tools* [online]. [cit. 21. 5. 2014]. Dostupné z: `http://bitbucket.org/edgarv/hdritools`.

[9] JAROSZ, W.; ZWICKER, M.; JENSEN, H. W. The Beam Radiance Estimate for Volumetric Photon Mapping. *Computer Graphics Forum (Proc. of Eurographics)*. 2008, 27, 2.

[10] JAROSZ, W. et al. A Comprehensive Theory of Volumetric Radiance Estimation Using Photon Points and Beams. *ACM Trans. Graph.* 2011, 30, 1, s. 5:1–5:19.

[11] JAROSZ, W. et al. Progressive Photon Beams. *ACM Trans. Graph. (Proc. of SIGGRAPH Asia)*. 2011, 30, 6.

[12] JENSEN, H. W. *Realistic Image Synthesis Using Photon Mapping*. Natick, MA, USA : A. K. Peters, Ltd., 2001. ISBN 1-56881-147-0.

[13] JENSEN, H. W.; CHRISTENSEN, P. H. Efficient Simulation of Light Transport in Scenes With Participating Media Using Photon Maps. In *Proc. of SIGGRAPH '98*, 1998.

[14] KŘIVÁNEK, J. et al. Unifying points, beams, and paths in volumetric light transport simulation. *ACM Trans. Graph.* August 2014, 33, 4, s. 1–13. ISSN 0730-0301.

[15] LAFORTUNE, E. P.; WILLEMS, Y. D. Bi-Directional Path Tracing. In *Compugraphics '93*, 1993.

[16] *MIT license* [online]. [cit. 27. 11. 2014]. Dostupné z: `http://en.wikipedia.org/wiki/MIT_License`.

[17] *Wavefront MTL file format specification* [online]. [cit. 27. 11. 2014]. Dostupné z: `http://paulbourke.net/dataformats/mtl/`.

[18] *Wavefront OBJ file format specification* [online]. [cit. 27. 11. 2014]. Dostupné z: `http://www.martinreddy.net/gfx/3d/OBJ.spec`.

[19] *OpenEXR* [online]. [cit. 27. 11. 2014]. Dostupné z: `http://www.openexr.com/`.

[20] PAULY, M.; KOLLIG, T.; KELLER, A. Metropolis Light Transport for Participating media. In *Rendering Techniques (Proc. of Eurographics Workshop on Rendering)*, 2000.

[21] PHARR, M.; HUMPHREYS, G. *Physically Based Rendering: From Theory to Implementation*. San Francisco, USA : Morgan Kaufmann, 2nd edition, 2010.

[22] SCHMIDT, C. M.; BUDGE, B. Simple Nested Dielectrics in Ray Traced Images. *Journal of Graphics Tools*. 2002, 7, 2, s. 1–8.

[23] *SIGGRAPH* [online]. [cit. 21. 5. 2014]. Dostupné z: `http://www.siggraph.org/`.

[24] *SmallUPBP* [online]. [cit. 15. 6. 2014]. Dostupné z: `http://www.smallupbp.com`.

[25] *SmallVCM* [online]. [cit. 15. 6. 2014]. Dostupné z: `http://www.smallvcm.com`.

[26] VEACH, E. *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University, Stanford, CA, USA, December 1997.

[27] VEACH, E.; GUIBAS, L. Bidirectional Estimators for Light Transport. In *Proc. of Eurographics Rendering Workshop*, 1994.

[28] VEACH, E.; GUIBAS, L. J. Optimally combining sampling techniques for Monte Carlo rendering. In *Proc. of SIGGRAPH '95*, 1995. doi: 10.1145/218380.218498. ISBN 0-89791-701-4.

[29] VORBA, J. Bidirectional photon mapping. In *Proc. of the Central European Seminar on Computer Graphics (CESCG '11)*, 2011.

List of Figures

List of Tables

List of Abbreviations

AABB	axis aligned bounding box
BPT	bidirectional path tracing
BPM	bidirectional photon mapping
LT	light tracing
MC	Monte Carlo
MFP	mean free path
MIS	multiple importance sampling
PPM	progressive photon mapping
PT	path tracing
RR	Russian roulette
UPBP	unified points, beams, and paths
UPS	unified path sampling
VCM	vertex connection and merging

List of Notation

I	image pixel intensity
Ω	space of light transport paths
$\overline{\mathbf{x}}$	light transport path
$f(\overline{\mathbf{x}})$	measurement contribution function of path $\overline{\mathbf{x}}$
$\mathrm{d}\mu(\overline{\mathbf{x}})$	differential path measure
$\mathbf{x}_i$	path vertex
$\tilde{\mathbf{x}}_i$	light vertex merged with camera vertex $\mathbf{x}_i$ by a photon density estimator
$L_\mathrm{e}(\mathbf{x}_0)$	radiance emitted from vertex $\mathbf{x}_0$
$T(\overline{\mathbf{x}})$	throughput of path $\overline{\mathbf{x}}$
$W_\mathrm{e}(\mathbf{x}_k)$	sensitivity of a sensor at vertex $\mathbf{x}_k$
$G(\mathbf{x}_i, \mathbf{x}_{i+1})$	geometry term for a path segment $\mathbf{x}_i\mathbf{x}_{i+1}$
$T_\mathrm{r}(\mathbf{x}_i, \mathbf{x}_{i+1})$	transmittance (attenuation) along a path segment $\mathbf{x}_i\mathbf{x}_{i+1}$
$T_\mathrm{r}(t)$	transmittance (attenuation) along a path segment segment of length t
T'_r	transmittance (attenuation) in homogeneous media
$T'_\mathrm{r,m}$	transmittance (attenuation) in homogeneous media in a color channel corresponding to the minimum positive component of σ_t
ρ	scattering function
$V(\mathbf{x}_i, \mathbf{x}_{i+1})$	indicator function of visibility between vertices $\mathbf{x}_i, \mathbf{x}_{i+1}$
$n_\mathbf{x}$	normal at vertex $\mathbf{x}$
$\omega_\mathbf{x}$	direction sampled from $\mathbf{x}$ (orientation is determined by context)
$\omega_{\mathbf{x}\rightarrow}$	direction sampled from light vertex $\mathbf{x}$ ("towards the camera")
$\omega_{\leftarrow\mathbf{x}}$	direction sampled from camera vertex $\mathbf{x}$ ("towards the light")
$\omega_\mathbf{xy}$	direction from $\mathbf{x}$ to $\mathbf{y}$
$t_\mathbf{x}$	distance sampled from $\mathbf{x}$ (orientation is determined by context)
$t_\mathbf{xy}$	distance sampled along the ray $(\mathbf{x}, \omega_\mathbf{xy})$
$t_{\mathbf{x}\rightarrow}$	distance sampled along the ray $(\mathbf{x}, \omega_{\mathbf{x}\rightarrow})$
$t_{\leftarrow\mathbf{x}}$	distance sampled along the ray $(\mathbf{x}, \omega_{\leftarrow\mathbf{x}})$
$l_\mathbf{x}$	length of a query/photon beam with origin at vertex $\mathbf{x}$
$D(\mathbf{x}\rightarrow\mathbf{y})$	equals $\lvert n_\mathbf{x} \cdot \omega_\mathbf{xy}\rvert$ if $\mathbf{x}$ is an inner path vertex located on a surface or the first path vertex located on an area light source, $D(\mathbf{x}\rightarrow\mathbf{y}) = 1$ otherwise
$D(\mathbf{x}\rightarrow)$	equals $\lvert n_\mathbf{x} \cdot \omega_{\mathbf{x}\rightarrow}\rvert$ if $\mathbf{x}$ is an inner path vertex located on a surface or the first path vertex located on an area light source, $D(\mathbf{x}\rightarrow) = 1$ otherwise
$D(\leftarrow\mathbf{x})$	equals $\lvert n_\mathbf{x} \cdot \omega_{\leftarrow\mathbf{x}}\rvert$ if $\mathbf{x}$ is an inner path vertex located on a surface, $D(\leftarrow\mathbf{y}) = 1$ otherwise
ρ_s	bidirectional scattering distribution function
ρ_p	phase function

σ_t	extinction coefficient
$\sigma_{\mathrm{t,m}}$	minimum positive component of σ_t
σ_s	scattering coefficient
σ_a	absorption coefficient
ε	emission coefficient
$\langle I \rangle$	unbiased MC estimator of the path integral
$p(\overline{\mathbf{x}})$	path pdf, equals $p(\mathbf{x}_0, \ldots, \mathbf{x}_k)$
$p(\omega)$	directional pdfs expressed w.r.t. the projected solid angle measure
$\hat{p}(\omega)$	directional pdfs expressed w.r.t. the solid angle measure
$p(t)$	distance pdfs expressed w.r.t. the Euclidean length on $\mathbb{R}^1$
$p(\mathbf{x})$	volume pdfs expressed w.r.t. the Euclidean volume on $\mathbb{R}^3$
$\bar{p}(d)$	pdf of sampling distance d within one volume segment
$p(t_{\mathbf{x}_i\mathbf{x}_{i+1}})$	a pdf of sampling through all volume segments between vertices $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$, $\mathbf{x}_{i+1}$ is in a medium
Pr	probability
$\Pr\{t > t_{\mathbf{x}_i\mathbf{x}_{i+1}}\}$	probability that distance t is sampled through all volume segments between vertices $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ long enough for vertex $\mathbf{x}_{i+1}$ to contribute
$C_\mathrm{l}(\mathbf{x}_0 \ldots \mathbf{x}_i)$	light subpath contribution
$C_\mathrm{l}(\mathbf{x}_0 \ldots \mathbf{x}_i]$	light subpath contribution including scattering at $\mathbf{x}_i$
$C_\mathrm{c}(\mathbf{x}_j \ldots \mathbf{x}_k)$	camera subpath contribution
$C_\mathrm{c}[\mathbf{x}_j \ldots \mathbf{x}_k)$	camera subpath contribution including scattering at $\mathbf{x}_j$
P-P3D	volumetric photon mapping estimator
P-B2D	generally denotes the P-B$_\mathrm{l}$2D and P-B$_\mathrm{s}$2D estimators
P-B$_\mathrm{l}$2D	beam radiance estimator using long query beams
P-B$_\mathrm{s}$2D	beam radiance estimator using short query beams
B-B1D	generally denotes the B$_\mathrm{l}$-B$_\mathrm{l}$1D, B$_\mathrm{l}$-B$_\mathrm{s}$1D, B$_\mathrm{s}$-B$_\mathrm{l}$1D and B$_\mathrm{s}$-B$_\mathrm{s}$1D estimators
B$_\mathrm{l}$-B$_\mathrm{l}$1D	photon beams estimator using long photon and query beams
B$_\mathrm{l}$-B$_\mathrm{s}$1D	photon beams estimator using long photon beams and short query beams
B$_\mathrm{s}$-B$_\mathrm{l}$1D	photon beams estimator using short photon beams and long query beams
B$_\mathrm{s}$-B$_\mathrm{s}$1D	photon beams estimator using short photon beams and short query beams
$\mathrm{BPT}_{\mathbf{x}_i\mathbf{x}_{i+1}}$	bidirectional path tracing estimator connecting vertices $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$
$\mathrm{BPT}_\mathrm{direct}$	bidirectional path tracing estimator sampling the whole light transport path entirely from the camera
SURF	surface photon mapping estimator
$\langle I \rangle_\mathrm{e}$	estimator e expression
K_3	normalized 3D kernel
K_2	normalized 2D kernel
K_1	normalized 1D kernel
$\sin\theta_{\mathbf{x}_{i-1}\mathbf{x}_{i+1}}$	sine of angle between $\omega_{\mathbf{x}_{i-1}\rightarrow}$ and $\omega_{\leftarrow\mathbf{x}_{i+1}}$

H	Heaviside step function
$\hat{w}_i$	MIS weighting function for technique i using balance heuristic
g	mean cosine
p_C	continuation probability
n_{paths}	number of light/camera subpaths traced
$\overrightarrow{p_i}$	denotes $p(\mathbf{x}_0, \ldots, \mathbf{x}_i)$
$\overleftarrow{p_i}$	denotes $p(\mathbf{x}_i, \ldots, \mathbf{x}_k)$
$\overrightarrow{p_j}$	denotes $p(\mathbf{x}_j \mid \mathbf{x}_{j-1})$, $\overrightarrow{p_0} = \overrightarrow{p_0}$
$\overleftarrow{p_k}$	denotes $p(\mathbf{x}_k \mid \mathbf{x}_{k+1})$, $\overleftarrow{p_7} = \overleftarrow{p_7}$
w^L	part of inverted MIS weight corresponding to a light subpath
$w^L_{k,r,a}$	part of inverted MIS weight corresponding to a light subpath of complete path of length k ending at vertex $\mathbf{x}_b$ initialized with r
w^C	part of inverted MIS weight corresponding to a camera subpath
$w^C_{k,s,b}$	part of inverted MIS weight corresponding to a camera subpath of complete path of length k starting at vertex $\mathbf{x}_b$ initialized with s
$R^L_{r,a}(i)$	common factor for vertex $\mathbf{x}_i$ on the light subpath ending at vertex $\mathbf{x}_a$ initialized with r
$R^C_{s,b}(i)$	common factor for vertex $\mathbf{x}_i$ on the camera subpath beginning at vertex $\mathbf{x}_b$ initialized with s
F_e	factor specific for estimator e
IF^L_e	initialization factor for estimator e and light subpath
IF^C_e	initialization factor for estimator e and camera subpath
$I_M(b)$	indicator $I_M(b) = 1 \Leftrightarrow \mathbf{x}_b$ is in a medium
$I_S(b)$	indicator $I_S(b) = 1 \Leftrightarrow \mathbf{x}_b$ is on a surface (not purely specular)
$I_{!\delta}(b, b+1)$	indicator $I_{!\delta}(b, b+1) = 1 \Leftrightarrow$ neither $\mathbf{x}_b$ nor $\mathbf{x}_{b+1}$ are on a purely specular surface
$I_{!\delta L}(b)$	indicator $I_{!\delta L}(b) = 1 \Leftrightarrow \mathbf{x}_b$ is on a light source that can be hit (is not delta)
$I_V(b)$	indicator $I_V(b) = 1 \Leftrightarrow \mathbf{x}_b$ is not the last vertex of the light subpath or the path was created by path integral estimators

Attachments

1 User documentation

While the previous chapters focus on the UPBP algorithm, its theoretical background and implementation, the main subject of this chapter is the resulting SmallUPBP computer program and its description from the user's point of view.

The program can be found on the attached DVD both as an executable file and as a source code. The latest version of the source code can be also downloaded from the SmallUPBP project site [24].

We begin with an explanation of how the program can be run and controlled, then we add a few notes related to its modification.

1.1 Running the program

1.1.1 Requirements

Partly because it is characteristic for all rendering software and partly because the program is experimental and its optimization was not our priority, the program is very hardware demanding. Since it is CPU-oriented, its performance is highly dependent on CPU power, number of threads and available operating memory. Results presented in Chapter 4 were all computed on a PC with a 4-core Intel i7-2600K CPU and 16GB of RAM using 8 threads and rendering a single image took from minutes (for the "compatible" and "previous" images) to days (for the full transport reference images). Although a worse configuration can be used, it may increase rendering times and limit the maximum image resolution or a number of traced light paths and stored photon beams.

In terms of software requirements, we targeted the program on 64-bit Microsoft Windows platform (it may work on others but we have not tested it). The supplied executable file has to be run under Microsoft Windows 7 operating system or newer with either Microsoft Visual Studio 2013 or Visual C++ 2013 Redistributable installed, all in 64-bit versions. In order to run the program in a different environment, it has to be newly compiled from the source code (see Section 1.2.3).

1.1.2 Installation

If the requirements are met, the supplied executable file can be run directly without any installation. It is the `SmallUPBP.exe` file located on the attached DVD in the folder `Implementation\Executable`. It is independent of any other files and can be moved and renamed freely (however, the example batch files mentioned in Section 1.1.8 expect it in the original location and with the original name). If executed directly, it should (after a short computation) output the same image as the one shown in Section 1.1.8. If there is neither Microsoft Visual Studio 2013 nor Visual C++ 2013 Redistributable installed on the computer, the program will not run and Windows will probably display an error message about a missing dynamic library. To solve this, one of the two mentioned software must be installed. Visual C++ 2013 Redistributable x64 is supplied on the attached

DVD in the folder `Implementation\Install`. If the program still does not work, compiling from the source code can help, see Section 1.2.3 for instructions.

1.1.3 Run

The program has no graphical user interface, it is a command line program. Its configuration, i.e. what will be rendered and how, is set via arguments the program is run with. By using the `-h` or `-hf` arguments, the program always displays (no matter whether any other arguments are used or not) a short or full help, respectively. The short help contains only the most important arguments whereas the full version lists all arguments the program knows.

1.1.4 Scenes

What can the program render? There are 41 predefined scenes available. To select one, argument `-s <sceneID>` is used, where `<sceneID>` represents the number of the desired scene. The predefined scenes are listed and briefly described in the full help (`-hf`). More detailed description is given in Attachment 2.

Moreover, a user can specify his or her own scene to render. The rest of this section describes its format. A user scene has to be defined in three files sharing the same `<name>`:

`<name>.obj` A file with geometry of the scene in the OBJ file format [18]. Can be obtained via export from 3ds max or other 3D modelling software. The program recognizes records `v` for vertices, `vn` for normals, `vt` for texture coordinates and `f` for faces (in formats `v`, `v/vt`, `v/vt/vn`, `v//vn`). Materials of (group of) faces are specified using `usemtl` followed by a name of a material in the `<name>.mtl` file referenced using `mtllib`. All faces have to have materials, exactly one `mtllib` reference is expected (to the `<name>.mtl` file).

`<name>.mtl` A file with materials of the scene geometry in the MTL file format [17]. Can be obtained via export from 3ds max or other 3D modelling software (while exporting to OBJ). Must contain all materials used in the `<name>.obj` file. The program recognizes records `Kd` for diffuse reflectance, `Ke` for emission (for area light sources), `Ks` for phong reflectance and `Ns` for phong exponent (see Section 3.2 for description of materials and light sources). Other material properties can be specified in the `<name>.obj.aux` file.

`<name>.obj.aux` A file with definition of the camera, media, additional material properties and lights. It is a manually created ASCII file in our own format.

Camera definition is exactly in a format which can be obtained via export from 3ds max (while exporting to ASCII). The program recognizes records `TM_ROW1` for the camera roll, `TM_ROW2` for the camera (backward) direction, `TM_ROW3` for the camera position, `CAMERA_FOV` for the camera horizontal field of view and `CAMERA_TDIST` for the camera focal distance. In addition to the 3ds max export, there can be one more record. If the camera is not in the global medium (note that there is always a global medium, clear by default), it is enclosed by a geometry with a material. This material must be explicitly specified by `CAMERA_MATERIAL`.

A medium is defined by typing `medium <mediumID>`, where `<mediumID>` represents an identifier of the medium. The definition is then followed by properties of the medium, each property record on a new line. The program recognizes records `absorption`, `emission`, `scattering`, `g` and `continuation_probability` (they correspond to the `mAbsorptionCoef`, `mEmissionCoef`, `mScatteringCoef`, `mMeanCosine` and `mContinuationProb` properties of the `HomogeneousMedium` class described in Section 3.2.2). To place the medium in the scene, it is either made the global medium by typing `globalMediumID <mediumID>` anywhere bellow its definition or associated with a material.

Additional properties of a material are introduced by typing `material <materialID>`, where `<materialID>` represents an identifier of the material in the `<name>.mtl` file. Then its properties can be set, each property record on a new line. The program recognizes records `geometryType` for the material type (real vs. imaginary), `ior` for the index of refraction, `Ke` for emission (overrides the emission in the `<name>.mtl` file if specified), `mirror` for mirror reflectance and `priority` for the material priority (see Section 3.2 for description of materials).

Furthermore, a medium can be associated with a material (recall that media are associated with geometry not materials, so it is internally associated with all geometry with this material). This is accomplished by adding `mediumID <mediumID>` to properties of the material, where `<mediumID>` represents an identifier of the medium to be associated with. Crossing boundary of such a material may then mean entering or leaving the specified medium (see Section 3.2.2).

Finally, the `<name>.obj.aux` file can define lights. An area light source is defined simply by setting emission of materials associated with its geometry. Similarly to the camera, if it is not in the global medium, it is enclosed by a geometry with a material and this material must be explicitly specified by `enclosingMatId`. Light sources of other types (point, direction, background) are defined by their own records: `<light_point>`, `<light_directional>`, `<light_background_constant>` and `<light_background_em>`. The last one defines a background light source with its emission controlled by an environment map. It needs a path (absolute or relative to `<name>.obj`) to an OpenEXR file containing the environment map. The map is assumed to be designed for the latitude-longitude mapping (i.e. its height maps to latitude, width to longitude). The record also contains a factor (positive) for scaling colors of the map (uniformly) and a factor (non-negative multiple of the map width) for rotating the map about the vertical axis (clockwise).

An example of the `<name>.obj.aux` file is given in Listing 4.1. For further details see the supplied scene files (described in Section 1.1.8) and the `ObjReader.hxx` and `ObjReader.cxx` source files. Note that the user scenes also have to comply with assumptions made in Section 3.2.3.1.

That is all about the format of user scenes. To select a user scene, argument `-s -1 <path>` is used, where `<path>` represents a path to the `<name>.obj` file. If no scene is selected at all (neither predefined, nor user), argument `-s 0` is assumed.

Listing 4.1: An example of the `sample_scene.obj.aux` scene file.

```
# CAMERA

TM_ROW0 0.6291    0.7773      0.0091
TM_ROW1 -0.0088      -0.0046       1.0000
TM_ROW2 0.7773    -0.6292       0.0040
TM_ROW3 105.5772   -163.5308     151.8657
CAMERA_FOV 0.6981
CAMERA_TDIST 1.0000
CAMERA_MATERIAL MaterialWine

# MEDIA

medium MediumWine
absorption 0.122 0.351 0.402
emission 0.0 0.0 0.0
scattering 0.015 0.013 0.011
g 0.9

medium MediumGlass
absorption 0.05 0.05 0.05
emission 0.0 0.0 0.0
scattering 0.0 0.0 0.0
g 0.0

medium MediumGlobal
absorption 0.0001 0.0001 0.0001
emission 0.0 0.0 0.0
scattering 0.0016 0.0016 0.0016
g 0.4
continuation_probability 0.8

globalMediumID MediumGlobal

# MATERIALS

material MaterialWine
mirror 1.0 1.0 1.0
ior 1.333
mediumId MediumWine
priority 1

material MaterialGlass
mirror 1.0 1.0 1.0
ior 1.520
mediumId MediumGlass
priority 5
geometryType real

# LIGHTS

material MaterialLight
Ke 800 790 780
enclosingMatId MaterialWine

#light_point <posX> <posY> <posZ> <emissionR> <emissionG> <emissionB>
#light_directional <dirX> <dirY> <dirZ> <emissionR> <emissionG> <emissionB>
#light_background_constant <emissionR> <emissionG> <emissionB>

light_background_em 0.6 0.96 room.exr
```

1.1.5 Algorithms

Now we know what the program can render. The next question is what algorithms it can use for the rendering. As we stated in Section 3.3, there are seven renderers to choose from and some of them are further configurable to render images using only a subset of all estimators they otherwise combine. To

specify which renderer and in what configuration should render an image, argument `-a <algorithmID>` is used, where `<algorithmID>` represents the identifier (acronym) of the desired combination. The full help lists all algorithms the program offers, here we give a more detailed description of them:

ID	Renderer	Algorithm
`el`	`EyeLight`	A visualization of the dot product of a surface normal and the camera ray direction. Ignores media.
`pt`	`PathTracer`	Path tracing with contributions of light sampling and directly hit light sources combined by MIS [21]. Ignores media.
`ppm`	`VertexCM`	Progressive photon mapping [6]. Ignores media.
`bpm`	`VertexCM`	Bidirectional photon mapping [29]. Ignores media.
`lt`	`VertexCM`	Light tracing. Ignores media.
`bpt`	`VertexCM`	Bidirectional path tracing [26]. Ignores media.
`vcm`	`VertexCM`	Vertex connection and merging [4, 7]. Ignores media.
`vptd`	`VolPathTracer`	The simplest volumetric path tracing, no light sampling, waits for a direct hit of a light source. Handles media. Supports light emission by media.
`vpts`	`VolPathTracer`	Path tracing of purely specular paths only, i.e. no light sampling, waits for a direct hit of a light source, terminates a path on the first vertex in a medium or with the first sampled non-specular surface interaction. Handles media (attenuates a path by them but places no vertices in them). Supports light emission by media.
`vptls`	`VolPathTracer`	Path tracing with explicit light sampling only, does not accumulate emission of directly hit light sources. Handles media. Supports light emission by media (but does not sample them as light sources).
`vptmis`	`VolPathTracer`	Path tracing with contributions of light sampling and directly hit light sources combined by MIS. Handles media. Supports light emission by media (but does not sample them as light sources).
`vlt`	`VolLightTracer`	Light tracing. Handles media.

pb2d	VolLightTracer	P-B2D (or the beam radiance estimate [9]) for primary camera rays (there is no camera subpath tracing). Primary camera rays ignore a background medium if present. Handles media (obviously).
bb1d	VolLightTracer	B-B1D (or the photon beams [11]) for primary camera rays (there is no camera subpath tracing). Primary camera rays ignore a background medium if present. Handles media (obviously).
vbpt_lt	VolBidirPT	Light tracing. Handles media.
vbpt_ptd	VolBidirPT	The simplest path tracing, no light sampling, waits for a direct hit of a light source. Handles media.
vbpt_ptls	VolBidirPT	Path tracing with explicit light sampling only, does not accumulate emission of directly hit light sources. Handles media.
vbpt_ptmis	VolBidirPT	Path tracing with contributions of light sampling and directly hit light sources combined by MIS. Handles media.
vbpt	VolBidirPT	Bidirectional path tracing. Handles media.
upbp_lt	UPBP	Light tracing. Handles media.
upbp_ptd	UPBP	The simplest path tracing, no light sampling, waits for a direct hit of a light source. Handles media.
upbp_ptls	UPBP	Path tracing with explicit light sampling only, does not accumulate emission of directly hit light sources. Handles media.
upbp_ptmis	UPBP	Path tracing with contributions of light sampling and directly hit light sources combined by MIS. Handles media.
upbp_bpt	UPBP	Bidirectional path tracing. Handles media.
upbp_ppm	UPBP	Progressive photon mapping. Handles media. Corresponds to evaluating SURF on the first non-specular surface from camera and terminating the traced path.
upbp_bpm	UPBP	Bidirectional photon mapping. Handles media. Actually SURF.
upbp_vcm	UPBP	Vertex connection and merging. Handles media. Actually a combination of BPT and SURF.
upbp_all	UPBP	Complete UPBP algorithm. Handles media (obviously).

Table 4.5: List of algorithms offered by SmallUPBP.

Besides the listed algorithms a user is allowed to pick his own combination of techniques from UPBP. The argument is in a form `-a upbp_<tech>[+<tech>]`* where `<tech>`$\in$ {`bpt`, `surf`, `pp3d`, `pb2d`, `bb1d`}. For example:

- `-a upbp_pp3d` evaluates P-P3D only,
- `-a upbp_pb2d+bb1d` evaluates a combination of P-B2D and B-B1D,
- `-a upbp_bpt+surf+pp3d+pb2d+bb1d` evaluates complete UPBP.

If no algorithm is selected, argument `-a upbp_all` is assumed.

1.1.6 Other options

The way a selected scene is rendered is not determined solely by the selected algorithm. There are many other options that influence the result. The full help (`-hf`) lists all of them, here we give a more detailed description:

Basic options

`-l <length>`

Sets:	Maximum length of light transport paths (and so the maximum length of traced subpaths). Setting `-l n` means that no complete light transport path has more than `n` path segments. For `-l 1` only area light sources directly visible from the camera are rendered. For `-l 2` only direct illumination is computed.
Value:	`<length>` $\in \mathbb{N}$
Default:	`-l 10`

`-t <sec>`

Sets:	Target number of seconds the rendering should run for. If specified together with the target number of iterations, the time takes precedence. Typing `-t 0` has no effect (as if the argument was not used at all).
Value:	`<sec>` $\in \mathbb{R}$, `<sec>` ≥ 0
Default:	None, the argument is not effective if not explicitly specified (target number of iterations is used).

`-i <iter>`

Sets:	Target number of rendering iterations. If specified together with the target number of seconds (`-t`, see above), the time takes precedence.
Value:	`<iter>` $\in \mathbb{N}$
Default:	`-i 1`

`-o <name>`

Sets:	Name of the resulting image file.
Value:	`<name>` is a valid file name, with extension .bmp or .exr. If specified without an extension, .exr is assumed. The name can be prefixed with relative or absolute path but the path must exist.

Default: None, the argument is not effective if not explicitly specified (name of the resulting image file is constructed from the rendering configuration).

`-r <res>`

Sets: Resolution of the rendered image.
Value: `<res>` in format `<width>x<height>`, where `<width>`, `<height>` $\in \mathbb{N}$
Default: `-r 256x256`

`-s <seed>`

Sets: Base seed for the random number generator.
Value: `<seed>` $\in \mathbb{N} \cup \{0\}$
Default: `-s 1234`

Performance options

`-th <threads>`

Sets: Number of threads used for rendering. Typing `-th 0` is equivalent to typing `-th <threads>` with `<threads>` equal to the number of cores of the installed CPU.
Value: `<threads>` $\in \mathbb{N} \cup \{0\}$
Default: `-th 0`

`-maxMemPerThread <memory>`

Sets: Maximum MB of memory allowed for light vertex array per each thread. If exceeded, the program terminates. Works only for algorithms from the `UPBP` renderer. Memory consumption can be reduced by lowering resolution (`-r`), tracing shorter paths (`-l`) or tracing fewer light subpaths (`-pcpi`).
Value: `<memory>` $\in \mathbb{N}$
Default: `-maxMemPerThread 500`

Radius options

`-r_alpha <alpha>`

Sets: Same radius reduction parameter for techniques SURF, P-P3D, P-B2D and B-B1D. If both `r_alpha` and `r_alpha_<tech>` are specified, `r_alpha_<tech>` takes precedence (but for techniques other than `<tech>` argument `r_alpha` applies). Works for all relevant renderers.
Value: `<alpha>` $\in \mathbb{R}$, `<alpha>` > 0
Default: None, the argument is not effective if not explicitly specified (radius reduction parameter has for different techniques different defaults).

`-r_alpha_<tech> <alpha>`

Sets: Radius reduction parameter for one of techniques SURF, P-P3D, P-B2D and B-B1D. If both `r_alpha` and `r_alpha_<tech>` are specified, `r_alpha_<tech>` takes precedence (but for techniques other than `<tech>` argument `r_alpha` applies). Works for all relevant renderers.

Value: `<tech>` $\in$ { `surf`, `pp3d`, `pb2d`, `bb1d` }, `<alpha>` $\in \mathbb{R}$, `<alpha>` > 0

Default: `-r_alpha_surf 0.75`, `-r_alpha_pp3d 1.0`, `-r_alpha_pb2d 1.0`, `-r_alpha_bb1d 1.0`

`-r_initial <init>`

Sets: Same initial radius for techniques SURF, P-P3D, P-B2D and B-B1D. If `<init>` is positive, it denotes absolute radius value. If `<init>` is negative, it denotes a value relative to the scene size (the actual radius is computed as its opposite value multiplied by the scene size). If both `r_initial` and `r_initial_<tech>` are specified, `r_initial_<tech>` takes precedence (but for techniques other than `<tech>` argument `r_initial` applies). Works for all relevant renderers.

Value: `<init>` $\in \mathbb{R}$, `<init>` $\neq 0$

Default: None, the argument is not effective if not explicitly specified (initial radius has for different techniques different defaults).

`-r_initial_<tech> <init>`

Sets: Initial radius for one of techniques SURF, P-P3D, P-B2D and B-B1D. If `<init>` is positive, it denotes absolute radius value. If `<init>` is negative, it denotes a value relative to the scene size (the actual radius is computed as its opposite value multiplied by the scene size). If both `r_initial` and `r_initial_<tech>` are specified, `r_initial_<tech>` takes precedence (but for techniques other than `<tech>` argument `r_initial` applies). Works for all relevant renderers.

Value: `<tech>` $\in$ { `surf`, `pp3d`, `pb2d`, `bb1d` }, `<init>` $\in \mathbb{R}$, `<init>` $\neq 0$

Default: `-r_initial_surf -0.0015`, `-r_initial_pp3d -0.001`, `-r_initial_pb2d -0.001`, `-r_initial_bb1d -0.001`

`-r_initial_pb2d_knn <const> <knn>`

Sets: Radius of a photon to the distance to its `<knn>`th nearest neighbour photon. The radius is multiplied by `<const>`. Undergoes no radius reduction. Takes precedence over `r_initial` and `r_initial_pb2d` if those arguments are also specified. Works for all relevant renderers.

Value: `<const>` $\in \mathbb{R}$, `<const>` > 0, `<knn>` $\in \mathbb{N}$

Default: None, the argument is not effective if not explicitly specified (the photon radius is determined by the initial radius and radius reduction parameter specified by the corresponding arguments).

`-r_initial_bb1d_knn <const> <knn>`

Sets:	Radius at each end of a photon beam to the distance between the beam end and the `<knn>`th closest beam vertex multiplied by `<const>`. Makes photon beams conic. Undergoes no radius reduction. Takes precedence if `r_initial` or `r_initial_bb1d` are also specified. Works for all relevant renderers.
Value:	`<const>` $\in \mathbb{R}$, `<const>` > 0, `<knn>` $\in \mathbb{N}$
Default:	None, the argument is not effective if not explicitly specified (the beam radius is constant along its length and determined by the initial radius and radius reduction parameter specified by the corresponding arguments).

Light transport options

`-previous`

Sets:	"Previous" rendering mode (see Section 3.3.1.1). Works only for algorithms from the `UPBP` renderer.
Value:	-
Default:	None, the argument is not effective if not explicitly specified.

`-previous_bb1d`

Sets:	"Previous" rendering mode (see Section 3.3.1.1), but for the B-B1D estimator only, other estimators are evaluated along all light transport paths. Works only for algorithms from the `UPBP` renderer.
Value:	-
Default:	None, the argument is not effective if not explicitly specified.

`-compatible`

Sets:	"Compatible" rendering mode (see Section 3.3.1.1). Works only for algorithms from the `UPBP` renderer.
Value:	-
Default:	None, the argument is not effective if not explicitly specified.

`-speconly`

Sets:	Only purely specular paths to be traced. Works only for algorithms from the `UPBP` renderer.
Value:	-
Default:	None, the argument is not effective if not explicitly specified.

`-ignorespec <option>`

Sets:	Whether purely specular subpaths from the camera will be ignored (`-ignorespec 1`) or not (`-ignorespec 0`). Works only for algorithms from the `UPBP` renderer.
Value:	`<option>` $\in \{0, 1\}$
Default:	`-ignorespec 0`

Beams options

`-gridres <res>`

Sets:	Resolution of the photon beams grid in the dimension of the maximum extent of the grid's bounding box. Resolution in other dimensions is set to give cube shaped grid cells. Works for all relevant renderers.
Value:	`<res>` $\in \mathbb{N}$
Default:	`-gridres 256`

`-gridmax <max>`

Sets:	Maximum number of photon beams tested for an intersection in one grid cell (0 means unlimited). The way this reduction is achieved is specified by the `gridred` argument. Works only if `-a bb1d` is used (i.e. for the `VolLightTracer` renderer, not `UPBP`). Implemented to increase performance of the B-B1D estimator, but it did not bring any significant improvement in our tests.
Value:	`<max>` $\in \mathbb{N} \cup \{0\}$
Default:	`-gridmax 0`

`-gridred <red>`

Sets:	Type of reduction of number of photon beams tested for intersection in one grid cell. Applied when actual number of beams n_a in a given grid cell exceeds maximum number n_m set by the `gridmax` argument. There are four types: • `<red>` $= 0$ (presample): all stored beams are randomly shuffled, first n_m beams are tested • `<red>` $= 1$ (offset): all stored beams are randomly shuffled, n_m beams are tested starting from a random index (and continuing from the first if necessary) • `<red>` $= 2$ (fixed resampling): n_m randomly chosen beams (not necessarily distinct) are tested • `<red>` $= 3$ (resampling): for each beam decides with probability n_m/n_a whether to test it or skip Works only if `-a bb1d` is used (i.e. for the `VolLightTracer` renderer, not `UPBP`). Implemented to increase performance of the B-B1D estimator, but it did not bring any significant improvement in our tests.
Value:	`<res>` $\in \{0, 1, 2, 3\}$
Default:	`-gridred 0`

`-beamdens <type> <max>`

Sets: Whether to output image(s) of statistics of type `<type>` of intersected photon beams grid cells and photon beams tested for intersection normalized to the given `<max>` or not. There are 6 types:

- `<type>` = 0: outputs nothing
- `<type>` = 1: outputs a false color image of a number of photon beams tested for an intersection with a primary ray relative to `<max>`
- `<type>` = 2: outputs a false color image of an average number of photon beams tested for an intersection with a primary ray relative to `<max>`
- `<type>` = 3: outputs a false color image of a number of photon beams grid cells intersected by a primary ray relative to `<max>`
- `<type>` = 4: outputs a false color image of a number of photon beams grid cells with more stored beams than set by the `gridmax` argument relative to `<max>`
- `<type>` = 5: outputs all four images

If `<max>` $= -1$, then the maximum in data is used; otherwise, the `<max>` value itself. Works only if `-a bb1d` is used (i.e. for the `VolLightTracer` renderer, not `UPBP`). Implemented for debugging of the grid.

Value: `<type>` $\in \{0, 1, 2, 3, 4, 5\}$, `<max>` $\in \mathbb{N} \cup \{-1\}$

Default: `-beamdens 0 -1`

`-beamstore <factor>`

Sets: Multiple of photon beams radius used for decision whether to store photon beams or not. If `<factor>` = 0, all beams are stored. Otherwise, beams are stored only in media with mean free path greater than 0.5π `<factor>radiusBB1D`, where `radiusBB1D` is the radius of photon beams (undergoes reduction). Works only if `-a bb1d` is used (i.e. for the `VolLightTracer` renderer, not `UPBP`). Implemented to increase performance of the B-B1D estimator (not using beams in media where P-B2D is better), but it did not bring any significant improvement in our tests.

Value: `<factor>` $\in \mathbb{R}$, `<factor>` ≥ 0

Default: `-beamstore 0`

`-qbt <type>`

Sets: Type of query beams. `-qbt S` sets short query beams, `-qbt L` sets long query beams. Works for all relevant renderers.

Value: `<type>` $\in \{\mathtt{S}, \mathtt{L}\}$

Default: `-qbt L`

`-pbt <type>`

Sets:	Type of photon beams. `-pbt S` sets short photon beams, `-pbt L` sets long photon beams. Works for all relevant renderers.
Value:	`<type>` $\in \{\texttt{S}, \texttt{L}\}$
Default:	`-pbt L`

`-pbc <count>`

Sets:	Number of traced light subpaths that will generate photon beams. If `<count>` is positive, it denotes an absolute number of light subpaths used for beam generation (the closest integer value is used). If `<count>` is negative, it denotes a number relative to the number of pixels (the actual number is computed as the closest integer value less than or equal to the absolute `<count>` value multiplied by the number of pixels). This argument influences only if a light subpath generates photon beams or not, the path is always traced. Works for all relevant renderers.
Value:	`<count>` $\in \mathbb{R}$, `<count>` $\neq 0$
Default:	`-pbc -1`

`-nosin`

Sets:	$\sin\theta$ to use only in computation of B-B1D contribution not MIS weights. Works for all relevant renderers. This was used during the development to test the importance of including the sin theta term in the MIS weight calculation. This should not be used in practice, as excluding the sin theta term increases variance.
Value:	-
Default:	None, the argument is not effective if not explicitly specified.

Debug options

`-debugimg_option <option>`

Sets: Whether or not to a output debug image. Possibilities are:

- `option=none`: outputs nothing
- `option=simple_pyramid`: outputs images of contributions for all possible lengths of light and camera subpaths (they form a pyramid, see the `Readme.txt` file on the attached DVD in the folder `Implementation\Tools`)
- `option=per_technique`: outputs images of individual contribution of each of combined techniques
- `option=pyramid_per_technique`: outputs images of individual contribution of each of combined techniques for all possible lengths of light and camera subpaths

Works only for algorithms from the `UPBP` renderer.

Value: <option> ∈ {none, simple_pyramid, per_technique, pyramid_per_technique}
Default: -debugimg_option none

-debugimg_multbyweight <option>

Sets: Whether or not the output debug images should contain contributions multiplied by MIS weight or not. Possibilities are:

- option=no: the contributions are unweighted
- option=yes: the contributions are weighted
- option=output_both: images for both weighted and unweighted contributions are generated

Works only for algorithms from the UPBP renderer.
Value: <option> ∈ {no, yes, output_both}
Default: -debugimg_multbyweight yes

-debugimg_output_weights <option>

Sets: Whether or not to output images of MIS weights for each of combined techniques. Works only for algorithms from the UPBP renderer.
Value: <option> ∈ {no, yes}
Default: -debugimg_output_weights no

Other options

-continuous_output <iter_count>

Sets: Whether or not an image accumulated so far should be output continuously every <iter_count> iterations. If <iter_count> = 0, then no images before finishing rendering are output.
Value: <iter_count> ∈ $\mathbb{N} \cup \{0\}$
Default: -continuous_output 0

-em <filepath>

Sets: Environment map in scenes with a background light source. If the rendered scene has not a background light source, this argument has no effect. If it has a background light, it is set to use the given environment map regardless of what it originally used.
Value: <filepath> is an absolute path an OpenEXR file containing the environment map. The map is assumed to be designed for the latitude-longitude mapping (i.e. its height maps to latitude, width to longitude).
Default: None, the argument is not effective if not explicitly specified.

`-min_dist2med <distance>`

Sets:	Minimum distance from the camera at which a contribution of a medium can be computed (no path with the last vertex before the camera in a medium closer to the camera than `<distance>` contributes to the result). If `<distance>` is positive, it denotes absolute distance value. If it is negative, it denotes a value relative to the scene size (the actual distance is computed as its absolute value multiplied by the scene size). If `<distance>` is zero, all paths contribute without restriction. Works only for algorithms from the `UPBP` renderer.
Value:	`<distance>` $\in \mathbb{R}$
Default:	`-min_dist2med 0`

`-rpcpi <path_count>`

Sets:	Reference light subpath count traced per iteration. If `<path_count>` is positive, it denotes an absolute reference number of light subpaths traced per iteration (the closest integer value is used). If it is negative, it denotes a number relative to the number of pixels (the actual reference number is computed as the closest integer value less than or equal to the absolute `<path_count>` value multiplied by the number of pixels).
Value:	`<path_count>` $\in \mathbb{R}$, `<path_count>` $\neq 0$
Default:	`-rpcpi -1`

`-pcpi <path_count>`

Sets:	Light subpath count traced per iteration. If `<path_count>` is positive, it denotes an absolute number of light subpaths traced per iteration (the closest integer value is used). If it is negative, it denotes a number relative to the number of pixels (the actual number is computed as the closest integer value less than or equal to the absolute `<path_count>` value multiplied by the number of pixels).
Value:	`<path_count>` $\in \mathbb{R}$, `<path_count>` $\neq 0$
Default:	`-pcpi -1`

`-sn <option>`

Sets:	Whether to use shading (i.e. interpolated) normals (`-sn 1`) or not (`-sn 0`). When `-sn 0` is specified, all shading calculations use the geometric normals.
Value:	`<option>` $\in \{0, 1\}$
Default:	`-sn 1`

`-time`

Sets:	The program to append the rendering time to the name of the output image file.
Value:	-
Default:	None, the argument is not effective if not explicitly specified.

Table 4.11: List of all arguments the SmallUPBP program recognizes.

1.1.7 Output

When the rendering is finished, an image with the accumulated result is output. The image is either LDR stored in the BMP format or, by default, HDR stored in OpenEXR format [19].

OpenEXR images can be converted to PNG using batch file `topng.bat` provided on the attached DVD in the folder `Implementation\Tools` (if PCG HDR Image Tools [8] are installed on the computer). The same folder also contains two scripts creating convenient html pages from debug images output when using the `debugimg` arguments. See the supplied `Readme.txt` file for details.

1.1.8 Examples

Let's take a look at a few examples of running the program. We assume that the executable file the program is run from (either the given one or compiled from the source code) is called `SmallUPBP.exe`. The we can execute

- `SmallUPBP.exe`

  If no arguments are given to the program, a default configuration is rendered. It corresponds to running the program with all arguments set to their default values. The default values are listed in Table 4.11. Figure 4.4 shows the result. Comparing with it is a fast and simple way how to check whether the program is correctly installed/compiled.

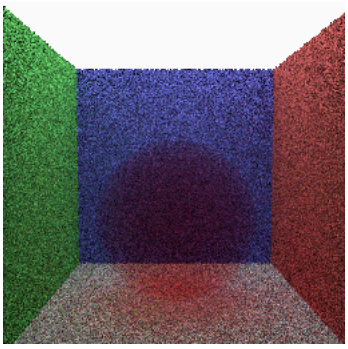


  Figure 4.4: An image produced by the SmallUPBP program in its default configuration.

- `SmallUPBP.exe -s 3 -a vptmis -i 10000 -r 100x100`

  The program renders the predefined scene number 3 (`-s 3`) using the volumetric path tracer with MIS (`-a vptmis`) for 10000 iterations (`-i 10000`) at a resolution of 100x100 pixels (`-r 100x100`).

  This is how we produced the image for scene 3 in the list of predefined scenes in Attachment 2. Images for scenes 4, 9 and 10 were also obtained by running the program with the same arguments only varying the scene number. Arguments for the rest of scenes differs only in the algorithm, `-a upbp_all` is used.

- `"..\..\..\Executable\SmallUPBP.exe" -s -1 "..\mirrorballs.obj" -a upbp_all -l 12 -i 1000000 -r 800x600 -pbc 4000 -maxMemPerThread 2000 -continuous_output 100`

  The program renders the given user scene (`-s -1 "..\mirrorballs.obj"`) using the complete UPBP algorithm (`-a upbp_all`) at a resolution of 800x600 pixels (`-r 800x600`) with maximum path length limited to 12 (`-l 12`). Only 4000 light subpaths (out of $800.600 = 480000$) is allowed to generate photon beams (`-pbc 4000`) and memory consumption must not exceeds 2000 MB per thread (`-maxMemPerThread 2000`). We want a converged result but we do not know how many iterations it will take. Therefore we set the number of iterations to a random sufficiently large number (`-i 1000000`), let the program output every 100 iterations image accumulated so far (`-continuous_output 100`) and terminate the program manually when the difference between two successive images is small enough.

  This is how we can reproduce the full transport reference image for the Mirror balls scene as shown in Figure 4.2 or in the UPBP paper [14]. We supplied scene files (.obj, .mtl, .obj.aux, see Section 1.1.4) for the Still life, Mirror balls, Bathroom and Candle scenes on the attached DVD in the folder `Implementation\scenes` as well as batch files needed to render all images presented in the UPBP paper and consequently in Chapter 4 (they can be also downloaded from the SmallUPBP project site [24]). The argument list above comes from one of them, namely `run_reference.bat` from the `Implementation\scenes\mirrorballs\batches` folder. It assumes the `SmallUPBP.exe` file in the folder `Implementation\Executable` and files of the Mirrorballs scene in the folder `Implementation\scenes\mirrorballs` sharing name `mirrorballs`. If any of these assumptions does not hold, the paths in the batch file have to updated. For example, if an executable file obtained by compiling the source code should be used (from its original location), then the path to the executable file has to be changed to `..\..\..\Build\SmallUPBP\x64\Release\SmallUPBP.exe`. If the paths are set correctly, then rendering the image is started simply by executing the batch file. The same applies to the other batch files. We refer to them and the supplied `Readme.txt` files for more information about how the other images are rendered.

We finished description of the program from the outside, now we proceed to its source code.

1.2 Modifying the program

The implementation is intended mainly for research purposes and as such it is expected to be often modified. The following text presents basic information a programmer needs about structure of the source code, licensing and compilation.

1.2.1 Structure

The implementation can be found on the attached DVD in the `Implementation` folder, which has the following content.

`embree-2.0`	Folder with source code of the Embree library.
`Executable`	Folder with a precompiled program executable.
`Install`	Folder with Visual C++ 2013 Redistributable x64.
`OpenEXR`	Folder with source code of the OpenEXR library.
`scenes`	Folder with scene and batch files used to render images shown in Chapter 4 and in the UPBP paper.
`SmallUPBP`	Folder with source code of the SmallUPBP program.
`Tools`	Folder with a few scripts for displaying and comparing rendered images.

If the implementation is downloaded from the SmallUPBP project site [24], it has exactly the same folder structure, only the `Executable` folder is missing.

1.2.2 License

SmallUPBP is released under the following license:

- most of the supplied code, scenes and associated files are subject to the MIT license [16]
- Embree is distributed under the Apache 2.0 license [1]
- OpenEXR uses the modified BSD licence [2]

1.2.3 Compilation

The source code is divided into two Microsoft Visual Studio 2013 solutions. The first one, `OpenEXR.sln` in the `Implementation\OpenEXR\src` folder, contains single project of the same name with the OpenEXR library. The second one, `SmallUPBP.sln` in the `Implementation\SmallUPBP` folder, contains three projects: `embree`, `sys` and `SmallUPBP`. The first two are parts of Embree and need to be compiled with the `OpenEXR` project as static libraries and linked with the main `SmallUPBP` project. The easiest way to do this is to use the provided solutions. The `OpenEXR.sln` solution is built first. It creates the static library `OpenEXR.lib` (or `OpenEXR-dbg.lib` depending on the selected configuration) in `Implementation\OpenEXR`. Then the `SmallUPBP.sln` is built. It creates the remaining `embree.lib` and `sys.lib` static libraries as well as the executable `SmallUPBP.exe` file in `Implementation\Build\SmallUPBP\x64\Release` (or in `Implementation\Build\SmallUPBP\x64\Debug` depending on the selected configuration). Note that the build configurations of both solutions must match. The `SmallUPBP.exe` file is completely independent of any other files and can be moved and run freely.

1.2.4 Code description

By describing compilation we got to the code itself. For more information about the third-party code of OpenEXR and Embree we refer to [19] and [3], respectively. Explanation of the UPBP implementation, how classes and methods cooperate, what data structures are used, which algorithms are employed – that is the goal of this thesis, mainly Chapter 3.

2 Predefined scenes

In this attachment we list all scenes predefined in the SmallUPBP program. But before we do so, we first describe basic components the scenes are composed of.

2.1 Materials

We begin with materials. There are 9 materials with different parameters used in the scenes. Default parameters of a material are:

diffuse reflectance = (0, 0, 0),
phong reflectance = (0, 0, 0),
phong exponent = 1,
mirror reflectance = (0, 0, 0),
IOR = -1,
priority = -1,
type = real.

We list the 9 materials (sorted ascending by the resulting priority) with parameters in which they differ from the defaults:

Name	Parameters
`Water`	mirror reflectance = (0.7, 0.7, 0.7), IOR = 1.33
`Ice`	mirror reflectance = (0.5, 0.5, 0.5), IOR = 1.31
`Glass`	mirror reflectance = (1.0, 1.0, 1.0), IOR = 1.6
`DiffuseRed`	diffuse reflectance = (0.803922, 0.152941, 0.152941)
`DiffuseGreen`	diffuse reflectance = (0.156863, 0.803922, 0.172549)
`DiffuseBlue`	diffuse reflectance = (0.156863, 0.172549, 0.803922)
`DiffuseWhite`	diffuse reflectance = (0.803922, 0.803922, 0.803922)
`GlossyWhite`	diffuse reflectance = (0.1, 0.1, 0.1), phong reflectance = (0.7, 0.7, 0.7), phong exponent = 90.0
`Mirror`	mirror reflectance = (1.0, 1.0, 1.0)

2.2 Media

Then there are 13 media used in the scenes. First, there is the `Clear` medium:

absorption = (0.0, 0.0, 0.0),
emission = (0.0, 0.0, 0.0),
scattering = (0.0, 0.0, 0.0),
continuation probability = 1,
mean cosine = 0.

The other media all have continuation probability set to 0.8. We list them (sorted ascending by the resulting priority) with the rest of parameters in which they differ from `Clear`:

Name	Parameters
`Water`	absorption = (0.7, 0.6, 0.0)
`WhiteIsoScat`	scattering = (0.9, 0.9, 0.9)
`WhiteAnisoScat`	scattering = (0.9, 0.9, 0.9), mean cosine = 0.6
`WeakWhiteIsoScat`	scattering = (0.1, 0.1, 0.1)
`WeakYellowIsoScat`	scattering = (0.1, 0.1, 0.0)
`WeakWhiteAnisoScat`	scattering = (0.1, 0.1, 0.1), mean cosine = 0.6
`RedAbs`	absorption = (0.0, 1.0, 1.0)
`YellowEmit`	emission = (0.7, 0.7, 0.0)
`YellowGreen`	absorption = (0.5, 0.0, 0.5), emission = (1.0, 1.0, 0.0), scattering = (0.1, 0.1, 0.0)
`BlueAbsEmit`	absorption = (0.1, 0.1, 0.0), emission = (0.0, 0.0, 1.0)
`AbsAnisoScat`	absorption = (0.0, 0.2, 0.0), scattering = (0.002, 0.002, 0.0), mean cosine = 0.6
`RedAbsAnisoScat`	absorption = (0.02, 2.0, 2.0), scattering = (12.0, 20.0, 20.0), mean cosine = -0.3

2.3 Background

A large part of the scenes is situated inside the (empty) Cornell box. We use three versions which differ in materials of their walls:

	`BoxDiffuseFloor`	`BoxGlossyFloor`	`BoxWhiteBack`
Ceiling	`DiffuseWhite`	`DiffuseWhite`	`DiffuseWhite`
Left wall	`DiffuseGreen`	`DiffuseGreen`	`DiffuseGreen`
Right wall	`DiffuseRed`	`DiffuseRed`	`DiffuseRed`
Back wall	`DiffuseBlue`	`DiffuseBlue`	`DiffuseWhite`
Floor	`DiffuseWhite`	`GlossyWhite`	`DiffuseWhite`

Note that the walls have no media associated.

2.4 Foreground

There are 8 pieces of geometry that can be used in the scenes besides the Cornell box. We say that they form the scene foreground while the box forms the scene background. We describe their positions in relation to the box, but these are fixed no matter whether the box is actually used or not. The foreground can be formed by:

`LargeSphereMiddle` A large sphere placed on the box floor in its centre.

`SmallSphereLeft` A small sphere placed on the box floor on the left. Fits in the box together with `SmallSphereRight` without overlapping.

`SmallSphereRight` A small sphere placed on the box floor on the right. Fits in the box together with `SmallSphereLeft` without overlapping.

`SmallSphereBottomLeft` A small sphere floating in the box slightly below and to the left from its centre.

`SmallSphereBottomRight` A small sphere floating in the box slightly below and to the right from its centre.

`SmallSphereTop` A small sphere floating in the box slightly above its centre.

`VeryLargeSphere` A sphere floating in the centre of the box large enough to encompass all three spheres `SmallSphereBottomLeft`, `SmallSphereBottomRight` and `SmallSphereTop` without any intersection.

`VeryLargeBox` A box of the same size and position as the Cornell box (but with all six faces).

Each piece can be associated with any of the aforementioned materials and any of the aforementioned media. We denote the associated material and medium by writing them in parenthesis after the name of the geometry. For example we have:

- `SmallSphereLeft(Glass, RedAbs)` is the small sphere made of the `Glass` material and filled with the `RedAbs` medium.
- `SmallSphereLeft(Glass, -)` is the small sphere made of the `Glass` material and filled with the `Clear` medium (recall that there is always some medium, at least clear).
- `SmallSphereLeft(-, RedAbs)` is the small sphere made of an imaginary material and filled with the `RedAbs` medium.

Case `SmallSphereLeft(-, -)` has no sense and is therefore not allowed.

Upon creation, priorities of materials associated with geometry are recalculated so as to meet the following:

1. If geometry GA and GB are associated with different materials $MatA$ and $MatB$, then geometry GA has a lower priority than GB if and only if material $MatA$ is listed in Section 2.1 sooner than $MatB$.
2. If geometry GA is associated with an imaginary material while GB with a real one, then geometry GA has a lower priority.
3. If geometry GA and GB are both associated with the same material (real or imaginary), then the result depends on media. If geometry GA and GB are associated with different media $MedA$ and $MedB$, then geometry GA has a lower priority than GB if and only if medium $MedA$ is listed in Section 2.2 sooner than $MedB$.

2.5 Light sources

Each scene has at most one light source. There are 8 different light sources available. We again describe their positions in relation to the Cornell box forming the scene background. However, same as in the case of the foreground, their positions are fixed no matter whether the box is actually used or not. The available light sources are:

`CeilingAreaBig` A white area light source covering the whole ceiling of the box.

`CeilingAreaSmall` A white area light source placed in the centre of the box ceiling covering only a small part of it.

`CeilingAreaSmallDistant` Same as `CeilingAreaSmall` but placed along the y axis slightly before the box.

`CeilingPoint` A white point light source placed in the centre of the box ceiling.

`FacingAreaSmall` Same as `CeilingAreaSmall` but floating in the centre of the box facing the camera.

`FacingPoint` A point light source placed in the centre of the box.

`Sun` An orange directional light source in infinity.

`Background` A constant light blue background light source.

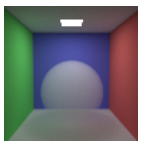
2.6 Scenes

Now we can proceed to the scene list. There are 41 scenes, which differ in their background (one of the three boxes or none), global medium, light source and foreground (various configurations of spheres and/or boxes):

Scene 0

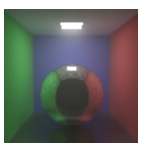
Background	`BoxDiffuseFloor`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`LargeSphereMiddle(-, RedAbs)`

Scene 1



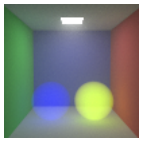
Background	`BoxDiffuseFloor`
Global medium	`Clear`
Light	`CeilingAreaSmall`
Foreground	`LargeSphereMiddle(-, WhiteIsoScat)`

Scene 2



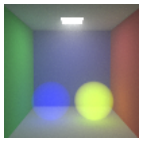
Background	`BoxGlossyFloor`
Global medium	`WeakWhiteIsoScat`
Light	`CeilingAreaSmall`
Foreground	`LargeSphereMiddle(Mirror, -)`

Scene 3



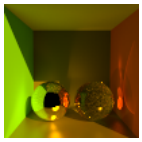
Background	`BoxDiffuseFloor`
Global medium	`WeakWhiteIsoScat`
Light	`CeilingAreaSmall`
Foreground	`SmallSphereLeft(-, BlueAbsEmit)`, `SmallSphereRight(-, YellowGreen)`

Scene 4



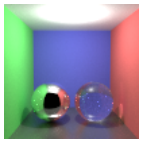
Background	`BoxDiffuseFloor`
Global medium	`WeakWhiteAnisoScat`
Light	`CeilingAreaSmall`
Foreground	`SmallSphereLeft(-, BlueAbsEmit)`, `SmallSphereRight(-, YellowGreen)`

Scene 5



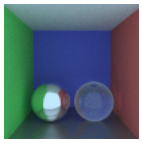
Background	`BoxGlossyFloor`
Global medium	`Clear`
Light	`Sun`
Foreground	`SmallSphereLeft(Mirror, -)`, `SmallSphereRight(Glass, -)`

Scene 6



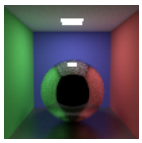
Background	`BoxGlossyFloor`
Global medium	`Clear`
Light	`CeilingPoint`
Foreground	`SmallSphereLeft(Mirror, -)`, `SmallSphereRight(Glass, -)`

Scene 7



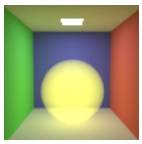
Background	`BoxGlossyFloor`
Global medium	`Clear`
Light	`Background`
Foreground	`SmallSphereLeft(Mirror, -)`, `SmallSphereRight(Glass, -)`

Scene 8



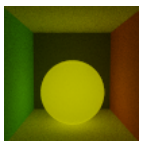
Background	`BoxGlossyFloor`
Global medium	`Clear`
Light	`CeilingAreaSmall`
Foreground	`LargeSphereMiddle(Mirror, -)`

Scene 9



Background	`BoxDiffuseFloor`
Global medium	`Clear`
Light	`CeilingAreaSmall`
Foreground	`LargeSphereMiddle(-, YellowEmit)`

Scene 10



Background	`BoxDiffuseFloor`
Global medium	`Clear`
Light	none
Foreground	`LargeSphereMiddle(Glass, YellowEmit)`

Scene 11

Background	none
Global medium	`WhiteIsoScat`
Light	`CeilingAreaBig`
Foreground	none

Scene 12

Background	none
Global medium	`WhiteIsoScat`
Light	`CeilingAreaSmall`
Foreground	none

Scene 13

Background	none
Global medium	`WhiteIsoScat`
Light	`CeilingPoint`
Foreground	none

Scene 14

Background	none
Global medium	`WhiteAnisoScat`
Light	`CeilingAreaBig`
Foreground	none

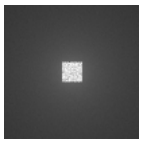
Scene 15

Background	none
Global medium	`WhiteAnisoScat`
Light	`CeilingAreaSmall`
Foreground	none

Scene 16

Background	none
Global medium	`WhiteAnisoScat`
Light	`CeilingPoint`
Foreground	none

Scene 17

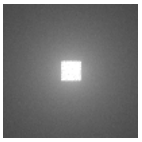


Background	none
Global medium	`WhiteIsoScat`
Light	`FacingAreaSmall`
Foreground	none

Scene 18

Background	none
Global medium	`WhiteIsoScat`
Light	`FacingPoint`
Foreground	none

Scene 19

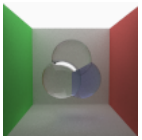


Background	none
Global medium	`WhiteAnisoScat`
Light	`FacingAreaSmall`
Foreground	none

Scene 20

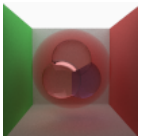
Background	none
Global medium	`WhiteAnisoScat`
Light	`FacingPoint`
Foreground	none

Scene 21



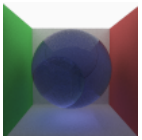
Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`SmallSphereBottomLeft(Glass, -)`, `SmallSphereBottomRight(Water, Water)`, `SmallSphereTop(Ice, -)`

Scene 22



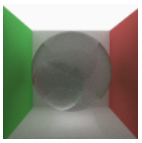
Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`SmallSphereBottomLeft(Glass, -)`, `SmallSphereBottomRight(Water, Water)`, `SmallSphereTop(Ice, -)`, `VeryLargeSphere(-, RedAbs)`

Scene 23



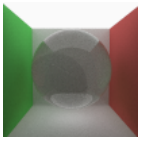
Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`SmallSphereBottomLeft(Glass, -)`, `SmallSphereBottomRight(Water, Water)`, `SmallSphereTop(Ice, -)`, `VeryLargeSphere(Water, Water)`

Scene 24



Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`SmallSphereBottomLeft(Glass, -)`, `SmallSphereBottomRight(Water, Water)`, `SmallSphereTop(Ice, -)`, `VeryLargeSphere(Ice, -)`

Scene 25

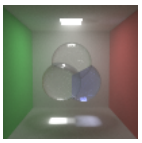


Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`SmallSphereBottomLeft(Glass, -)`, `SmallSphereBottomRight(Water, Water)`, `SmallSphereTop(Ice, -)`, `VeryLargeSphere(Glass, -)`

Scene 26

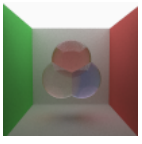
Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`SmallSphereBottomLeft(Glass, -)`, `SmallSphereBottomRight(Water, Water)`, `SmallSphereTop(Ice, -)`, `VeryLargeSphere(Mirror, -)`

Scene 27



Background	`BoxWhiteBack`
Global medium	`WeakWhiteAnisoScat`
Light	`CeilingAreaSmall`
Foreground	`SmallSphereBottomLeft(Glass, -)`, `SmallSphereBottomRight(Water, Water)`, `SmallSphereTop(Ice, -)`

Scene 28

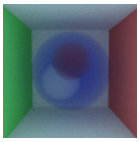


Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`SmallSphereBottomLeft(Glass, WhiteIsoScat)`, `SmallSphereBottomRight(Glass, Water)`, `SmallSphereTop(Glass, RedAbs)`

Scene 29

Background	none
Global medium	`WeakWhiteIsoScat`
Light	`CeilingAreaBig`,
Foreground	`SmallSphereBottomLeft(-, WhiteIsoScat)`, `SmallSphereBottomRight(-, Water)`, `SmallSphereTop(-, RedAbs)`

Scene 30

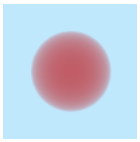


Background	`BoxWhiteBack`
Global medium	`Clear`
Light	`Background`
Foreground	`SmallSphereBottomLeft(-, WhiteIsoScat)`, `SmallSphereBottomRight(-, Water)`, `SmallSphereTop(-, RedAbs)`, `VeryLargeSphere(Water, Water)`

Scene 31

Background	none
Global medium	`AbsAnisoScat`
Light	`Background`
Foreground	none

Scene 32

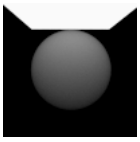


Background	none
Global medium	`Clear`
Light	`Background`
Foreground	`VeryLargeSphere(-, RedAbs)`

Scene 33

Background	none
Global medium	`Clear`
Light	`Background`
Foreground	`VeryLargeSphere(-, WhiteIsoScat)`

Scene 34



Background	none
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`VeryLargeSphere(-, WhiteIsoScat)`

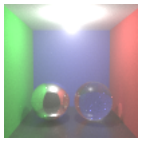
Scene 35

Background	none
Global medium	`Clear`
Light	`CeilingAreaSmallDistant`
Foreground	`VeryLargeBox(-, WeakYellowIsoScat)`

Scene 36

Background	`BoxGlossyFloor`
Global medium	`WeakWhiteIsoScat`
Light	`Sun`
Foreground	`SmallSphereLeft(Mirror, -)`, `SmallSphereRight(Glass, -)`

Scene 37

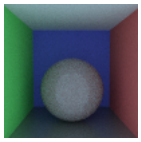


Background	`BoxGlossyFloor`
Global medium	`WeakWhiteIsoScat`
Light	`CeilingPoint`
Foreground	`SmallSphereLeft(Mirror, -)`, `SmallSphereRight(Glass, -)`

Scene 38

Background	`BoxGlossyFloor`
Global medium	`WeakWhiteIsoScat`
Light	`Background`
Foreground	`SmallSphereLeft(Mirror, -)`, `SmallSphereRight(Glass, -)`

Scene 39



Background	`BoxGlossyFloor`
Global medium	`Clear`
Light	`Background`
Foreground	`LargeSphereMiddle(Glass, RedAbsAnisoScat)`

Scene 40

Background	`BoxGlossyFloor`
Global medium	`Clear`
Light	`CeilingAreaBig`
Foreground	`LargeSphereMiddle(Glass, RedAbsAnisoScat)`

Most of the images of the scenes presented above were rendered using the complete UPBP algorithm. However, images for scenes 3, 4, 9, 10 were rendered by path tracing from the `VolPathTracer` renderer, because they include emissive media, which only the `VolPathTracer` renderer can handle. Note, that images for scenes 31, 36, 38 are black because they have a light source in infinity and a global attenuating medium. There is therefore no light that could get from a light source to the camera.

2.7 Modification

To modify the predefined scenes, the code of the SmallUPBP program has to be changed. Materials, media, geometry and lights are all created in the `LoadCornellBox` method of the `Scene` class from the `Scene.hxx` file. There their parameters can be modified. Assembling of the Cornell boxes and scenes is done in the global `initSceneConfigs` method from the `Config.hxx` file. There the scenes can be removed, added or modified (in terms of what the scenes include and where).

3 DVD contents

The attached DVD has the following contents:

`Implementation\embree-2.0`
: Folder with source code of the Embree library.

`Implementation\Executable`
: Folder with a precompiled program executable.

`Implementation\Install`
: Folder with Visual C++ 2013 Redistributable x64.

`Implementation\OpenEXR`
: Folder with source code of the OpenEXR library.

`Implementation\scenes`
: Folder with scene and batch files used to render images shown in Chapter 4 and in the UPBP paper.

`Implementation\SmallUPBP`
: Folder with source code of the SmallUPBP program.

`Implementation\Tools`
: Folder with a few scripts for displaying and comparing rendered images.

`Thesis`
: This thesis.