FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Vojtěch Tázlar

A Methodical Approach to the Evaluation of Light Transport Computations

Department of Software and Computer Science Education

Supervisor of the master thesis:	doc. Alexander Wilkie, Dr.
Study programme:	Computer Science
Study branch:	Computer Graphics and Game Development

Prague 2020

In Prague date 30. 07. 2020 Vojtěch Tázlar

I would like to thank my supervisor, doc. Alexander Wilkie, Dr., for all the provided valuable information, advice, and feedback. I would also like to send my thanks to doc. Ing. Jaroslav Křivánek, Ph.D., who introduced the world of computer graphics to me.

Title: A Methodical Approach to the Evaluation of Light Transport Computations

Author: Bc. Vojtěch Tázlar

Department: Department of Software and Computer Science Education

Supervisor: doc. Alexander Wilkie, Dr., department

Abstract: Photorealistic rendering has a wide variety of applications, and so there are many rendering algorithms and their variations tailored for specific use cases. Even though practically all of them do physically-based simulations of light transport, their results on the same scene are often different - sometimes because of the nature of a given algorithm or in a worse case because of bugs in their implementation. It is difficult to compare these algorithms, especially across different rendering frameworks, because there is not any standardized testing software or dataset available. Therefore, the only way to get an unbiased comparison of algorithms is to create and use your dataset or reimplement the algorithms in one rendering framework of choice, but both solutions can be difficult and time-consuming. We address these problems with our test suite based on a rigorously defined methodology of evaluation of light transport algorithms. We present a scripting framework for automated testing and fast comparison of rendering results and provide a documented set of non-volumetric test scenes for most popular research-oriented rendering frameworks. Our test suite is easily extensible to support additional renderers and scenes.

Keywords: computer graphics, rendering, light transport simulation, global illumination, test scenes, image quality metrics

Contents

Introduction

Rendering is a technique of obtaining 2D images from a virtual representation of the world, called a scene. There is a realtime rendering where tens or even hundreds of images need to be produced every second, which is only possible with many simplifications and shortcuts. On the other hand, photorealistic rendering tries to achieve as realistic images as possible and may need several hours of rendering time to produce a single image of the required quality.

Photorealistic rendering has a wide variety of applications - to name a few: architecture, product and design visualization, movies, and video games. It then does not come as a surprise that there are many rendering frameworks (renderers) and their rendering algorithms tailored to specific use cases - they may be optimized for a specific purpose at the expense of others, or do fast approximations. And so, although practically all of them are based on physically-based simulations of light transport (usually some form of path tracing), their results on the same scene can be very different. Human perception also has its limitations, and we may even accept images that are not physically correct as more photorealistic than authentic photographs [1]. Some algorithms may knowingly exploit that, but others may have hidden bugs, which went unnoticed because their results seemed to be correct.

If we want to choose the best rendering algorithm for a specific task, we need to compare them with each other. That is usually done by comparison of rendered images of the same scene. But it is way harder to replicate the same scene in different rendering frameworks than it may look. Rendering frameworks usually use incompatible scene description formats, so it may be straight-up impossible to even define some scenes for a given renderer. On top of that, there is not any standardized testing of renderers available.

In the end, the only way to compare multiple rendering algorithms is to reimplement them in one specific rendering framework and test them on the same scenes or make a set of scenes that are perfectly reproducible across the tested renderers. Often neither of these solutions is feasible because we lack understanding of the algorithms, do not have the required knowledge of the rendering frameworks to create matching scenes, or are not ready to invest necessary time because we are only interested in fast comparison of the algorithms and nothing else.

Both of these approaches are even more problematic in the realm of proprietary renderers, which often use tricks and simplifications to fasten rendering calculations or to allow artistic (non-physically-based) control. With them, we know nothing of the code that should be reimplemented, and even if we recreate a scene, its interpretation by the renderer may be different than expected. This last issue applies to all renderers, but with open-sourced ones, we can find and document reasons for the resulting differences and work around them. But to do that, we need a testing environment and a set of matching scenes that would supplement our endeavor.

Goals

Rendering frameworks are complex pieces of software made of many interacting subsystems. To get a matching rendered image across different renderers, their subsystems must be able to make the same intermediate results, although their approaches and implementations may be different. And so, even if we want to compare one of these subsystems, we inadvertently end up comparing the system as a whole.

In this thesis, we address issues of comparison and evaluation of different renderers with a focus on physically-based rendering, which has only one single correct solution. We specifically target a subsystem of the renderer, which facilitates simulation of light transport in the scene - light transport algorithms.

We present an extensible test suite of light transport algorithms provided with support for the two currently most popular rendering frameworks used for rendering research: Mitsuba [2] and PBRT [3]. We also include a documented set of canonical test scenes for non-volumetric rendering. With this thesis, we hope to provide a foundation for standardized testing of light transport algorithms across different rendering frameworks, which is currently missing in the field of photorealistic rendering.

Thesis Structure

In the first two chapters, we introduce the basics of physically-based rendering (1) and light transport theory (2). Then we continue with the specification of methodology for evaluation of light transport computations (3) and present our evaluation framework (4). Finally, we present our set of test scenes and discuss their development (5).

1. Physically-Based Rendering

In this chapter, we present the absolute basics of rendering: we explain what is visible light (1.1) and color (1.2). Then we introduce the idea of rendering (1.3) and finally discuss how does the rendering input look like (1.4).

1.1 Light

Visible light is electromagnetic radiation, which can be perceived by the human visual system. That is radiation with wavelengths roughly between 360nm and 760nm [4] (exact range differs between individuals). Light is not usually radiation of a single wavelength but consists of many waves of different wavelengths. Distribution of their intensities describes the light and is called a spectrum (figure 1.1).

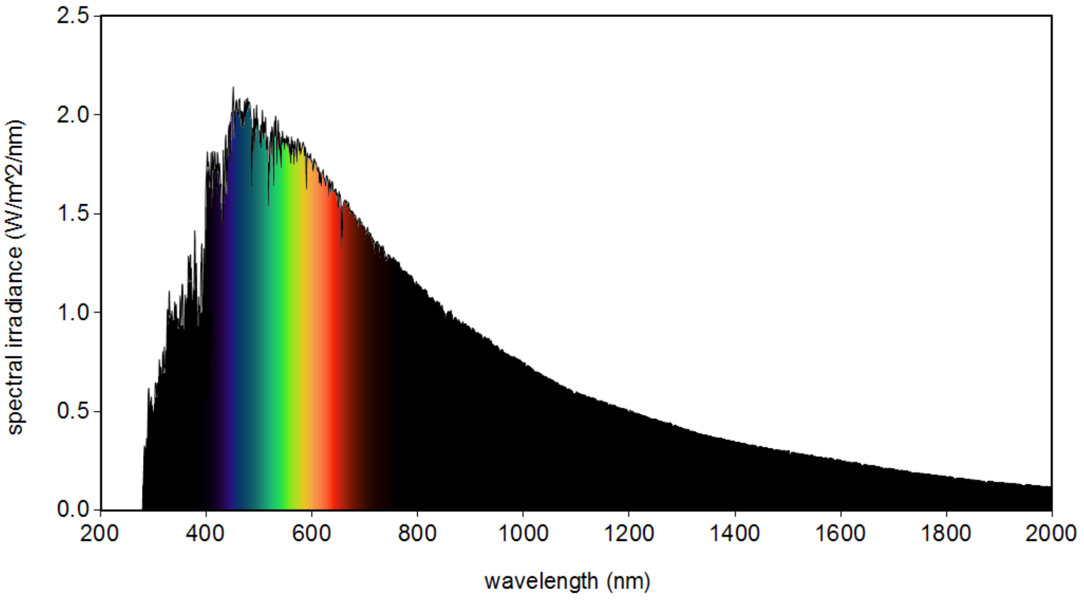


Figure 1.1: Sunlight spectrum. (Source: Ye 2018 [5])
Visible wavelengths are colored by corresponding perceived colors.

1.2 Color

The observed color is a response of our brain to signals sent from light-sensitive cells in retinas of our eyes - rods and cones. Rods are highly sensitive to the intensity of light but can not perceive color. Three types of cones, with varying sensitivities to light of different wavelengths, react to incoming light and send signals to the brain. These signals can be understood as red, green, and blue colors, and their subsequent processing by the brain causes the final color sensation. In the case of monochromatic light we can assign perceived color to a given wavelength (figure 1.2), but generally speaking one specific color can come up from an endless number of different spectra.

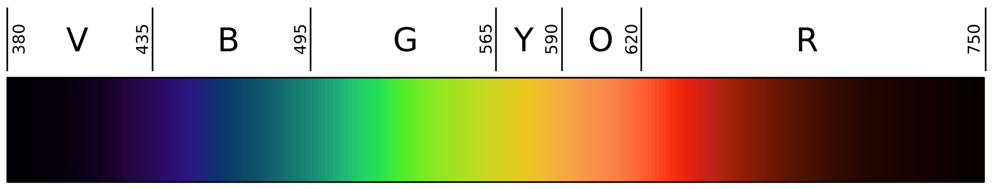


Figure 1.2: Detailed spectrum of visible light.[1]
Numbers represent wavelengths.

Different objects around us have different colors - so what happens during the travel of light from, for example, sun, to our eyes? The light must be reflected from the object to our eye, but not necessarily all of it - some may be refracted or absorbed, so light does not necessarily have the same spectrum after it bounces from the object. And because there is an endless amount of different surfaces with varying properties, we perceive the world around us as so colorful.

1.3 Rendering

To render a photorealistic image of a virtual world, we need to simulate how the light interacts with objects in the scene before it finally gets to the camera (virtual eye) - figure 1.3. For the image to be physically correct, all parts of the rendering system must be mathematically and physically sound - from the description of light sources, individual objects, and materials they are made from, to the light transport simulation itself. Only then we get images that are practically indistinguishable from reality (figure 1.4).

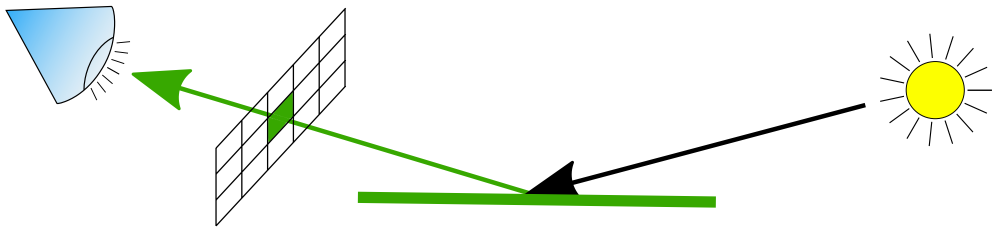


Figure 1.3: Light interactions in the scene.
Light from the sun bounces off the surface into the virtual eye. Only the green part of the light spectrum is reflected. The corresponding pixel of the rendered image registers incoming green light, but to calculate the correct color it needs to combine contributions from as many light-carrying paths in the scene as possible.

Light transport calculations can be done while simulating interactions on the spectrum of visible light (spectral rendering), but more often the light representation is simplified to RGB values (RGB rendering). Some phenomena like dispersion, polarisation, or fluorescence are impossible to simulate without calculations done on the spectral representation of light, yet, they are not necessary for most use cases or would not make a big enough difference, which could warrant the increased computation demands.

[1]Public domain, https://commons.wikimedia.org/wiki/File:Linear_visible_spectrum.svg [Online; accessed 2020-07-20], modified.

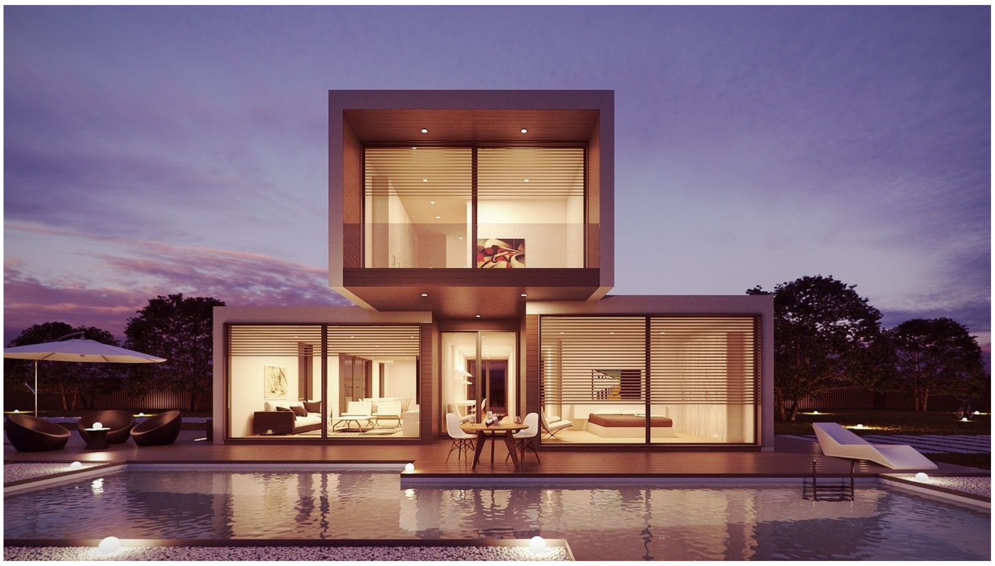


Figure 1.4: Architectural rendering. [2]

In this thesis, we focus on renderers which primarily do their calculations in the RGB space, because most of the current research is done on them and because there is only a small number of spectral renderers. Nonetheless, our work can be directly applied to spectral renderers too.

1.4 Rendering Input

For a renderer to produce an image, it needs to know what to draw and how to draw it. The former is supplied by the scene description and the latter by renderer settings (figure 1.5).

1.4.1 Scene Description

The scene is a virtual representation of the world. It is a file in a file format supported by a renderer which describes:

- Light sources (position, shape, properties)
- The geometrical properties of the objects
- Definition of materials and their assignment to the objects

Even parts of the scene that are not visible in the final image are important - for example, light sources that are not visible still emit light and objects cast shadows, which may be seen in the resulting image. Light transport without exception happens everywhere in the scene.

[2]Public domain, image by Giovanni Gargiulo, https://pixabay.com/photos/architecture-house-3d-design-1477041/ [Online; accessed 2020-07-20].

```
# Camera settings and image resolution
LookAt 3 3 3.5 0 0 0 0 0 1
Camera "perspective" "float fov" [45]
Film "image"
  "integer xresolution" [512]
  "integer yresolution" [512]

# Rendering settings
Integrator "path" "integer maxdepth" [9]
Sampler "halton" "integer pixelsamples" [256]

WorldBegin # Scene definition

# Light sources
LightSource "infinite" "rgb L" [.4 .45 .5]
LightSource "distant" "point from" [-30 40 100]
  "blackbody L" [3000 1.5]

# Objects and their materials
Material "matte" "rgb Kd" [.449, .245, .682]
Shape "sphere" "float radius" 1

Translate 0 0 -1
Material "matte" "rgb Kd" [.8, .8, .8]
Shape "trianglemesh"
  "integer indices" [0 1 2 0 2 3]
  "point P" [-20 -20 0 20 -20 0 20 20 0 -20 20 0]
WorldEnd
```

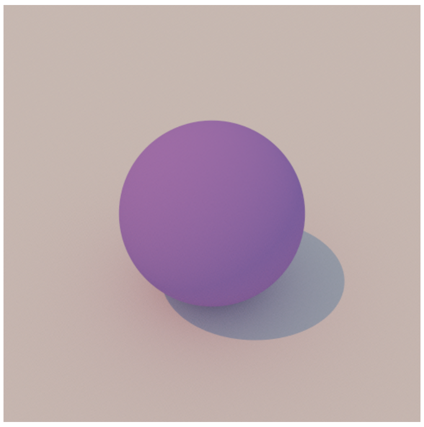


Figure 1.5: PBRT input file and corresponding rendered image.

1.4.2 Renderer Settings

Renderer settings tell the renderer how to produce the final image. They are usually passed to the renderer together with the scene or as a part of the file describing the scene. They allow to specify:

- Selection light transport algorithm or at least specification of its properties
- Target quality of the image (e.g. rendering time)
- Post-processing options (e.g. denoising settings)

1.4.3 Camera

Camera controls position, orientation, and properties of the virtual eye from which the resulting image is rendered. The handling of the camera differs between renderers. On the one hand, it can be seen as a physical object and be a part of the scene description, as it is usually done in digital content creation tools like Autodesk 3ds Max [6] or Blender [7]. But on the other hand, it can be regarded just as a set of settings that dictates how the virtual scene should be viewed (i.e. not a part of the scene itself), as it is done in the PBRT or Mitsuba renderers. Because the camera is closely tied to a specific scene (position and orientation in the scene) it is in both cases shipped together with the scene. We will, in the following text, also think of a camera as a part of the scene.

2. Light Transport Theory

This chapter introduces the basics of the light transport theory, which we use throughout this work. First, we present the rendering equation and methods used for its computation (2.1), then we discuss how different kinds of materials are modeled (2.2) and finally introduce algorithms that are used to resolve light transport and discuss their difficulties (2.3). Note that the presented light transport theory does not address participating media.

For most topics in this chapter, we also recommend Veach's thesis [8] or PBRT book [3].

2.1 Rendering Equation and Monte Carlo Integration

In this section, we describe the fundamental equation of light transport (2.1.1) and discuss Monte Carlo integration methods, which are often used in its computations (2.1.2). Furthermore, we introduce importance sampling and multiple importance sampling techniques (2.1.3), which further improve the efficiency of the Monte Carlo integration.

2.1.1 Rendering Equation

All of the current research and algorithms are based on a rendering equation, first introduced by Kajiya [9]. It assumes that light travels in the scene instantly, and so we can observe a constant distribution of light over a fixed time (equilibrium). For rendering, it means that we can make an image of some constant state of the scene. Here a slightly modified version:

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_{S^2} L(x, \omega_i) f_r(x, \omega_i, \omega_o) cos\theta_i d\omega_i.$$

Where:

x	is a point in the space.
ω_o	is the direction of outgoing light from the point x.
ω_i	is a reversed direction of the incoming light.
L	is the equilibrium radiance.
L_e	is the emitted radiance.
S^2	is a sphere centered at the point x.
$f_r(x, \omega_i, \omega_o)$	is the Bidirectional Scattering Distribution Function (BSDF) describing how much of the incoming light from direction ω_i gets scattered into the outgoing direction ω_o at the point x (see 2.2.1).
$cos\theta_i$	is the cosine of the angle between the w_i and normal on the same side of the surface (e.g. it is a reversed surface normal for an incoming refracted light from the other side of a glass object).

Radiance is a radiometric quantity defined by:

$$L(x,\omega) = \frac{d^2\Phi}{cos\theta dA d\omega} \qquad [Wm^{-2}sr^{-1}].$$

It is a radiant flux density (Φ) per unit area perpendicular to the ray ($cos\theta dA$), per unit solid angle ($d\omega$), in the direction of the ray ω. For all of our purposes, it is enough to understand radiance as an amount of light which is traveling from a point in a specific direction.

Rendering equation above can be loosely interpreted as: outgoing light from the point x in the direction ω_o is equal to the light emitted from that point in the same direction plus all of the light that is scattered from any incoming (ω_i) to the outgoing direction.

Rendering equation is an integral equation with unknown quantity L not just inside of the integral, but also on the left side of the equation. Thus to compute the value of $L(x,\omega_o)$ we need to know values of $L(x,\omega_i)$ which are $L(x_{\omega_i},\omega_o)$ - the same kind of value that we are trying to compute but at different points and with different outgoing orientations. In this way, we could recursively unroll the integral (to the infinity), which is an idea applied in many rendering techniques (for example see: 2.3.1)). It is also a reason why the rendering equation can not be solved analytically, and because of the infinite integral, many of the numerical methods are not feasible either.

2.1.2 Monte Carlo Integration

Monte Carlo integration is a stochastic numerical method of integration. Because it allows us to efficiently compute even complex integrals, it is used in practically all light transport algorithms based on the evaluation of the rendering equation.

For a function $f(x)$ and integral over its domain:

$$I = \int f(x)dx,$$

secondary Monte Carlo estimator of the I is:

$$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \quad X_i \propto p(x).$$

The estimator is a weighted average of N independent samples X_i of a random variable with probability density function (PDF) $p(x)$. It can be proven that the expected value of the estimator is equal to the I:

$$E[F_N] = E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] = \frac{1}{N}\sum_{i=1}^{N}E\left[\frac{f(X_i)}{p(X_i)}\right] \overset{*}{=} \frac{1}{N}\sum_{i=1}^{N}\int\frac{f(x)}{p(x)}p(x)dx$$
$$= \frac{1}{N}\sum_{i=1}^{N}\int f(x)dx = \int f(x)dx = I,$$

where $\overset{*}{=}$ from $F = \frac{f(X)}{p(X)}$ and $E[F] = \int \frac{f(x)}{p(x)}p(x)dx = I$, in other words expected value of a single sample (primary Monte Carlo estimator) is equal to the I.

There are two basic properties of estimators:

- We say that the estimator F is *unbiased* if its expected value $E[F]$ is equal to the estimated quantity Q. In our case, the Monte Carlo estimator is *unbiased* because $E[F_N] = I$. If $E[F] \neq Q$, we call the estimator *biased*, and $B[F] = Q - E[F]$ is the amount of its bias.

- We say that the estimator F is *consistent* if its bias converges to zero, with the number of samples approaching infinity - i.e. if $\lim_{N \to \inf} B[F] = 0$.

This terminology is often extended to light transport algorithms as a whole. We can think of them as estimators of the rendering equation.

Note that the only requirement for use of the Monte Carlo integration is the ability to evaluate integrand $f(x)$ at any point in its domain. Additionally, the number of samples does not depend on the dimension of the integral (compared to other numerical methods like numerical quadrature), which is important for the estimation of infinite-dimensional integrals likes of the rendering equation.

Until now, we have ignored the variance of the estimator and its change with an increasing number of samples. It can be proven (similarly as in the derivation of the expected value above) that error of the Monte Carlo estimator decreases at a rate of $O(\sqrt{N})$ [8]. So to half an error, we need four times the amount of samples. This can be observed during the rendering when the rendered image clears up relatively quickly at the start, but additional improvements get gradually smaller.

2.1.3 Multiple Importance Sampling

Variance can be further reduced by multiple methods. We will discuss the two most commonly used ones in rendering practice: *importance sampling* and *multiple importance sampling.*

Importance sampling is based on the idea that samples from parts of the integration domain with higher values of the integrand have a higher impact on the result. It can be observed that if we had a PDF proportional to the function of the estimated integral ($p(x) = cf(x)$), our samples would have zero variance:

$$\frac{f(X_i)}{p(X_i)} = \frac{1}{c} \overset{*}{=} \frac{1}{\frac{1}{\int f(x)dx}} = \int f(x)dx,$$

where $\overset{*}{=}$ from c, which is a normalization factor of $p(x)$. Integral of PDF $p(x)$ must be 1: $\int p(x)dx = 1$ [3].

Because all the samples are constant ($1/c$), the resulting variance is zero. Nonetheless, if we had a PDF that is proportional to the estimated integral, we would not have to use Monte Carlo integration at all. In practice, this is a motivation for *importance sampling.* Importance sampling tries to preferentially take samples from areas of high contribution by using PDF as similar to the shape of the integrand as possible. So with an increasingly similar shape of the PDF, the variance of the Monte Carlo integration decreases. It is important to keep in mind that this also works the other way around, so with poorly chosen PDF importance sampling may increase variance (figure 2.1).

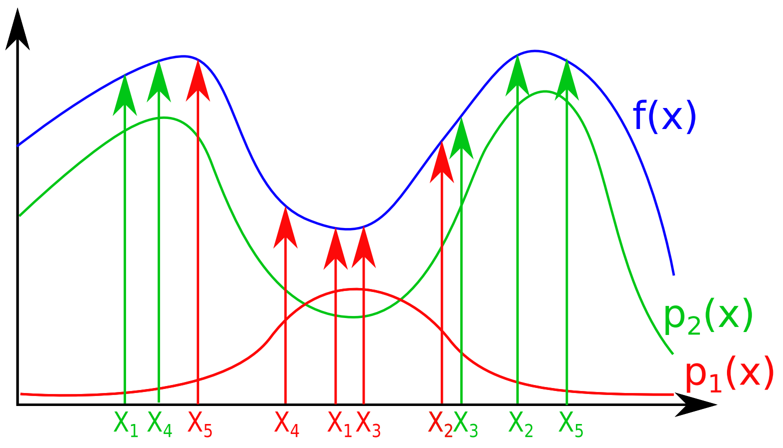


Figure 2.1: Importance Sampling.
If we are estimating integral of the function $f(x)$ with Monte Carlo integration and we had choose to take samples from one of the two available PDFs: $p_1(x)$ and $p_2(x)$, we would select $p_2(x)$. Its shape is similar to the shape of the integrand, therefore samples drawn from it will have a smaller variance compared to samples from the PDF $p_1(x)$, which would even increase variance in comparison to a simple uniform sampling of the integral domain.

Even with importance sampling, we rarely have a PDF that would mimic integrand on its entire domain. Instead, we can have multiple different PDFs where each is a good match of the integrand in part of its domain (figure 2.2). This is especially the case when we are trying to estimate integral, which is a product of multiple functions $\int f(x)g(x)$ (also rendering equation integral). Here if we have two PDFs similar in shape to the $f(x)$ and $g(x)$ respectively, which one should we choose? Ideally, we would want to combine the best of both of them. Multiple importance sampling (MIS) is a technique introduced by Veach [8] that addresses this issue.

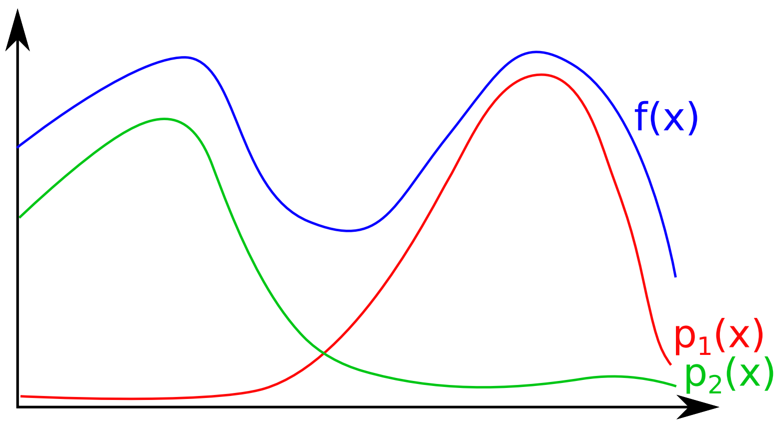


Figure 2.2: Monte Carlo integration and multiple PDFs.
If we are estimating integral of the function $f(x)$ and have available PDFs $p_1(x)$ and $p_2(x)$, we would want to use $p_2(x)$ as the sampling distribution of the left part of the integrand domain and $p_1(x)$ for its right part to minimize the variance of the estimator.

The idea of the MIS is to draw samples from multiple available sampling distributions (PDFs) and weight them together in a single combined estimator:

$$F = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})},$$

where i is a sampling technique providing n_i samples $X_{i,j}$ with PDF $p_i(X_{i,j})$, and w_i is a combination weight of the technique. This equation is similar to the secondary Monte Carlo estimator, with the addition of different sampling techniques and MIS weights. It can be proven that the combined estimator is unbiased as long as $\forall x : \sum_{i=1}^{n} w_i(x) = 1$.

The next step is to define combination weights in a way that minimizes the variance of the combined estimator. Naive averaging does not work, because variance is additive in this case [3]. Veach [8] introduced a weighting function which provably reduces variance called *balance heuristic*:

$$w_i(x) = \frac{n_i p_i(x)}{\sum_k n_k p_k(x)},$$

where k are individual sampling techniques. If we plug these weights into the combined estimator we get:

$$F = \sum_{i=1}^{n} \sum_{j=1}^{n_i} \frac{f(X_{i,j})}{\sum_{k=1}^{n} n_k p_k(x)}.$$

As we can observe, the contribution of a sample no longer depends on which technique it came from - it is not weighted by its PDF but by a weighted average of all the individual PDFs. *Balance heuristic* is by no means optimal function for weights, and so, a lot of current research is focused on finding better weighting functions (see [10] or [11]).

2.2 Modeling Materials

This section introduces functions used for the representation of surface reflectance (2.2.1), discusses laws of geometric optics for reflection and refraction (2.2.2), and finally presents an approach that is often used to model roughness of the surfaces (2.2.3).

2.2.1 BRDF

Reflective properties of materials determine the appearance of the object, whether it is a color or perceived roughness and glossiness. These properties are described by the *Bidirectional reflectance distribution function* (BRDF):

$$f_r(x, \omega_i, \omega_o) = \frac{dL_o(x, \omega_o)}{dE(x, \omega_i)} = \frac{dL_o(x, \omega_o)}{L_i(x, \omega_i) cos\theta_i d\omega_i} \qquad [sr^-1].$$

Where:

x	is a point on the surface.
ω_o	is the direction of outgoing light from the point x.
ω_i	is a reversed direction of the incoming light.
$L_i(\omega), L_o(\omega)$	is the incoming and reflected radiance in the direction ω, respectively.
$cos\theta_i$	is the cosine of the angle between the ω_i and normal of the surface
$dE(x, \omega_i)$	is the differential irradiance at point x (amount of incoming light from a differential cone around ω_i).

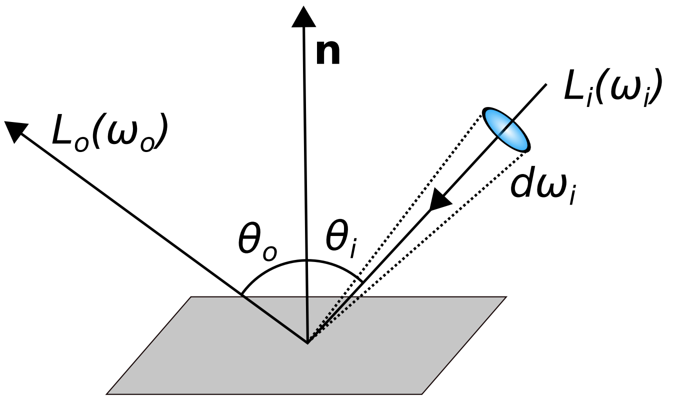


Figure 2.3: Bidirectional Reflectance Distribution Function. Note the differential cone around ω_i.

BRDF describes how much of the incident light from the direction ω_i is reflected to the direction ω_o (figure 2.3). Range of the BRDF values is: [0,∞). After a normalization we can also view BRDF as a probability density over all possible ω_o on the hemisphere around the surface normal.

Physically plausible BRDFs must follow two properties:

1. *Helmholz reciprocity*: $\forall\ \omega_i, \omega_o : f_r(x, \omega_i, \omega_o) = f_r(x, \omega_o, \omega_i)$.
2. *Energy conservation*: $\forall\ \omega_o : \int_{H^2(n)} f_r(x, \omega_i, \omega_o) cos\theta_i d\omega_i \leq 1$ [3]. Surface can not reflect more light that it receives.

Depending on the distribution of the reflected light from the surface we can define basic types of surface reflections (figure 2.4):

- *Ideal diffuse* surface reflects light in all directions equally, also called the Lambertian surface. It is a mathematical model that does not exist in nature. In the following text, we will refer to them as diffuse or Lambertian surfaces.

- *Glossy* (glossy specular) surface reflects light predominately in a set of directions.

- *Ideal mirror* (mirror, or perfect specular) surface reflects incident light in a single outgoing direction based on the *law of reflection* (2.2.2).

- *Retro-reflective* surface reflects light primarily back towards the incident direction.

Most of the surfaces in nature have reflectance properties, which would come up from a combination of these basic types of reflection.

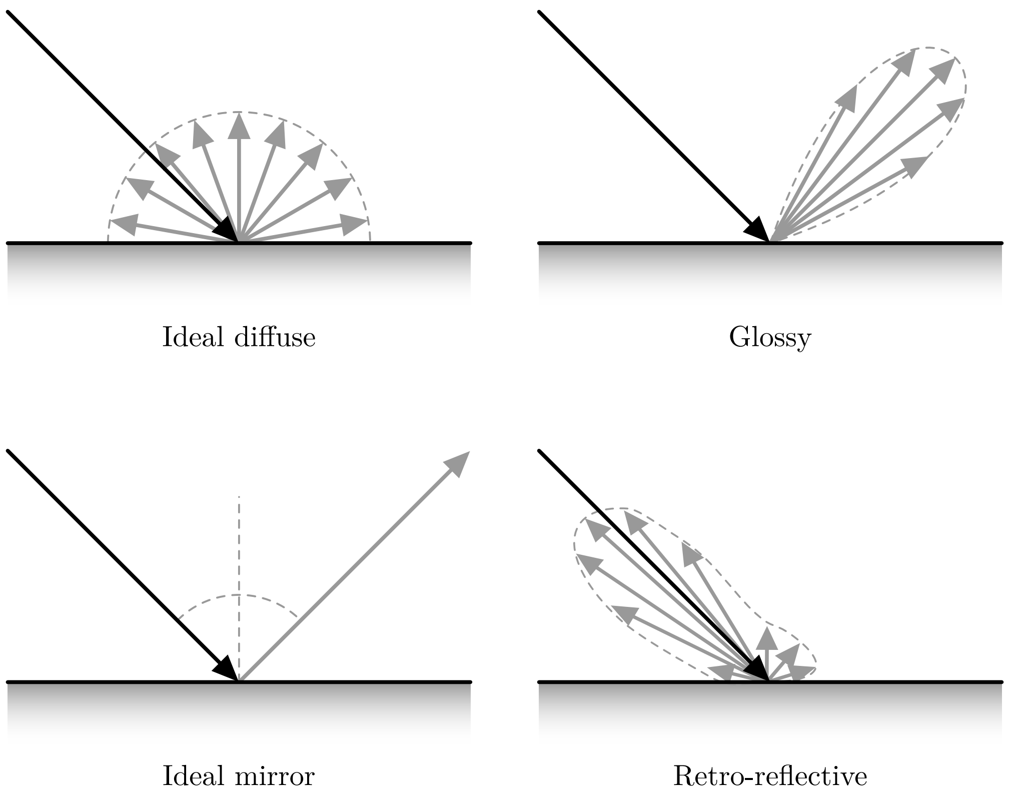


Figure 2.4: Basic types of reflective surfaces. [1]

[1] Public domain, https://commons.wikimedia.org/wiki/File:BRDF_diffuse.svg (also: File:BRDF_glossy.svg and File:BRDF_mirror.svg) [Online; accessed 2020-07-20]

BRDF describes light reflectance at a point on the surface, but there are other interactions of light with a surface, which are expressed by various distribution functions:

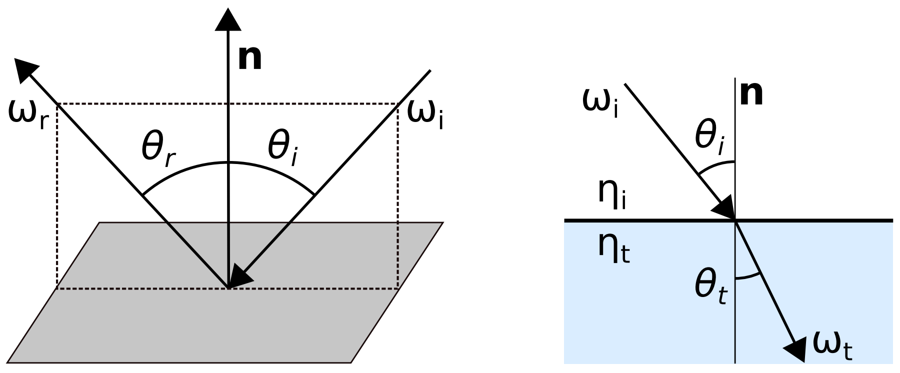
- *Bidirectional transmittance distribution* (BTDF) function describes light transmission (ω_i and ω_o are in opposite hemispheres around x - refraction of the light)
- *Bidirectional scattering distribution functions* (BSDF) is a combination of BRDF and BTDF into a single distribution function.
- *Bidirectional surface scattering reflectance distribution function* (BSSRDF) takes into account light transport under the surface, so the surface point of the incident light may be different than of the outgoing light.

Note that we can also define spatially varying distribution functions, with different distribution at each point of the surface.

2.2.2 Reflection and Refraction

In the previous section, we have introduced functions describing reflective and refractive properties of surfaces. For these functions to describe physically plausible surfaces, they have to follow laws of geometric optics: the *law of reflection*, *Snell's law*, and *Fresnel equations*.

The *law of reflection* describes ideal mirror reflection and states that direction of the reflected light ω_r makes the same angle as the incident direction ω_i on the opposite side of the surface normal n in the plane formed by ω_i and ω_r (figure 2.5).



Law of reflection Snell's law

Figure 2.5: Reflection and refraction laws.

Snell's law describes direction of refracted light on a boundary between two media (figure 2.5):

$$\eta_i sin\theta_i = \eta_t sin\theta_t,$$

where η refers to the index of refraction (IOR) which describes how fast light travels through the medium ($n = c/v$).

Finally, with *Fresnel equations*, we can compute how much of the incident light is reflected and how much of it is refracted. Their result depends on the direction of the incident light and IORs of the media.

It is important to note that there are a few types of materials with distinctive refractive and transmissive properties:

1. *Dielectrics* are materials that do not conduct electricity. These materials transmit part of the incoming light. Examples are air, water, glass, and most of the gemstones.

2. *Conductors* are materials that conduct electricity, such as metals. Compared to dielectrics, these materials reflect most of the incoming light and appear to be opaque. They also transmit part of the incoming light, but it is quickly absorbed inside of the conductor.

3. *Semiconductors* like silicon or germanium are not usually considered in rendering practice.

2.2.3 Microfacet Models

To account for varying roughness of materials, microfacet-based BRDF models are often used. Microfacet models were introduced to graphics by Cook and Torrance [12], and are a staple in the modern rendering practice. These models assume that the macrostructure of the surface consists of small randomly oriented microfacets, where each microfacet behaves as an ideal mirror.

The resulting BRDF models are based on stochastical modeling of light interactions on a patch of microfacets, where aggregate behavior determines the resulting scattering of light (figure 2.6).

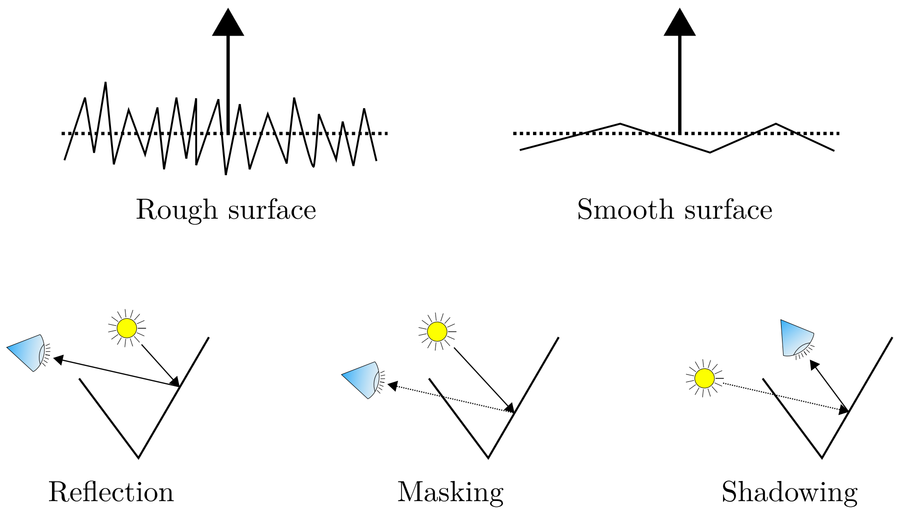


Figure 2.6: Microfacet model.
Three main light interactions on the microfacets are considered (bottom row). During *reflection*, light bounces between microfacets before it gets to the viewer, with *masking* microfacet is occluded by another one and during *shadowing* microfacet is not reached by the light. Distribution of the microfacets directly influences the roughness of the surface (top row).

There are many different distributions of microfacets used. Two important and widely used physically-based ones are Beckmann distribution [13] and GGX [14] (also Trowbridge and Reitz [15]).

2.3 Light Transport Algorithms

In this section, we introduce the path tracing algorithm (2.3.1), discuss what light transport situations are difficult to evaluate (2.3.2) and finally make an overview of the currently popular light transport algorithms (2.3.3).

2.3.1 Path Tracing

Path tracing is one of many light transport algorithms whose objective is to resolve the rendering equation. Because of its relative simplicity, and in most cases, reasonable efficiency, it is the most used rendering algorithm in practice. Path tracing is an *unbiased* Monte Carlo algorithm introduced by Kajiya [9]. Its basic idea is to generate light carrying paths via tracing of rays from the camera through interactions in the scene and ending at light sources.

Let's denote x_i as an ith point on the path from the camera (camera being at x_0) and $r(x_i, \omega_i)$ and as a ray with an origin x and direction ω_i. Then path construction of the path tracer can be described in the following steps:

1. Generate ray from the camera $r(x_0, \omega_0)$, also called a *primary ray*. The direction ω_0 is sampled according to the sampling technique and camera properties. Primary ray will intersect a pixel of the rendered image to which the completed light carrying path will contribute.

2. Cast the primary ray (determine first surface point hit by the ray) $r(x_0, \omega_o)$ to find a point x_1. If the point is on a light source surface, the path is completed. Otherwise, new direction ω_1 needs to be sampled and ray $r(x_1, \omega_1)$ cast. This is repeated for x_i until ray intersects a light source surface.

The sampling of the direction ω_i at a surface point x_i is usually based on the importance sampling of BRDF - recall that BRDF can be normalized to a probability density function (2.2.1). If it is not possible, at least sampling proportional to the cosine term of the rendering equation can be used.

The total contribution of the light carrying path is based on the radiance emitted by the light source and all of its interactions along the path. These are described by the BRDF ($f_r(x, \omega_{inc}, \omega_{out})$) and cosine terms ($cos\theta_{inc}$) at each point of the path between the camera and the light source (see the rendering equation). Also, PDF corresponding to the importance sampling of ω_i is included in the calculation ($p(\omega_i)$):

$$L_{inc}(x_0, R(\omega_0)) = L_e(x_n, R(\omega_{n-1})) \prod_{i=1}^{n-1} \frac{f_r(x_i, \omega_i, R(\omega_{i-1})) cos\theta_i}{p(\omega_i)},$$

where $R(\omega)$ is the reversed orientation of the ω, $L_e(x_n, R(\omega_{n-1}))$ is the emitted radiance by the point on the light source, and $L_{inc}(x_0, R(\omega_0))$ is the incident radiance from the path at the camera. Depending on the type of the camera, PDFs of x_0 and ω_0 sampling may also need to be included.

At this point, it should be apparent that path tracing is in its approach very similar to the rendering equation. We can understand path tracing of a path starting from the ray $r(x_i, \omega_i)$, as an estimation of the incoming light from the direction ω_i to the point x_i. Every generated path is a sample of the Monte

Carlo integration, where each bounce of the path unwinds the infinite-dimensional integral of the rendering equation by an additional step.

With each bounce, the throughput (the relative amount of light which is transported by the path - product in the equation above) usually decreases (except for specular bounces). So the longer the paths are, the less they contribute to the final image. For this reason, it makes sense to terminate inefficient paths preemptively, but methods like limiting the maximal number of bounces or setting a minimum acceptable throughput introduce bias. This issue is addressed by the Russian roulette technique: at each bounce, we decide if the path should continue with some probability p, and if it does throughput is multiplied by $1/p$. It is straightforward to prove that this approach maintains unbiasedness of the resulting estimator:

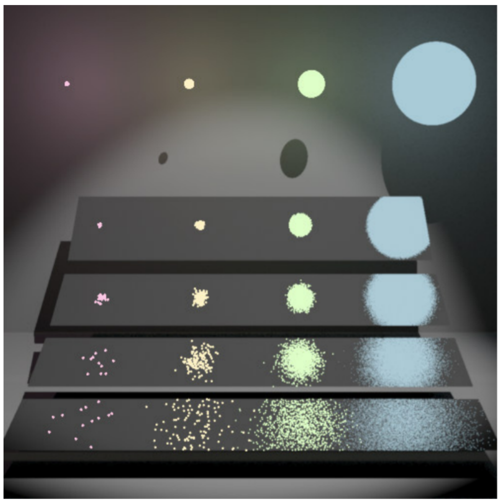
$$E[E_n] = \frac{E[E_o]}{p}p + 0(p-1) = E[E_o],$$

where the expected value of the new estimator E_n is the same as the expected value of the original estimator E_o.

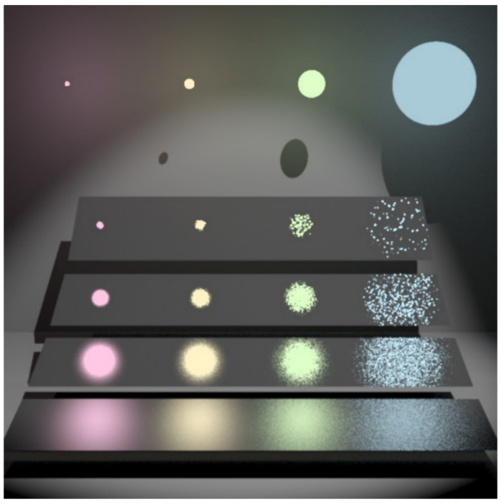
In the current formulation, the path would not be completed until the ray gets to the light source. That is very difficult in cases where the light source is small or even impossible if it is a point light (often used in practice). This problem can be addressed by an explicit sampling of light sources - by a sampling of a point on the light source and connecting it to the currently traced path if it is not occluded by other objects.

Both BRDF sampling and explicit light source sampling have their strong and weak points, but because both of them are estimators of direct illumination at a point, they can be combined with multiple importance sampling into a single robust estimator (figure 2.7, for details see [8] or [3]).

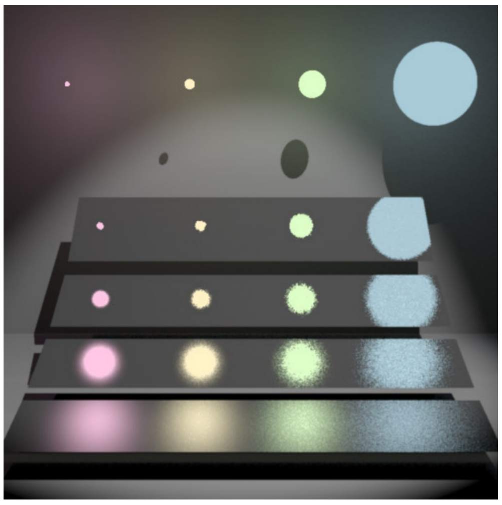
Path tracing in practice usually does the direct illumination estimation at each bounce of the path until the path is terminated because of Russian roulette or other termination criteriums (figure 2.8). The total contribution of the traced path is effectively the sum of contributions from paths of different lengths, where all the paths share the same primary ray $r(x_0, \omega_0)$ and thus contribute to the same pixel of the rendered image.



(a) BRDF sampling



(b) Light source sampling



(c) Multiple importance sampling

Figure 2.7: Direct illumination sampling techniques. (Source: Veach 1997[8]) These images compare direct illumination sampling techniques on glossy surfaces of varying roughness with four spherical light sources of varying sizes but the same total emitted power.
a) BRDF sampling is more efficient on smooth glossy surfaces with a sharp specular highlight (top row) and bigger light sources. Smooth surfaces reflect most of the light predominately in a set of directions, and BRDF sampling samples them efficiently. With a bigger light source, there is a better chance that sampled directions will lead towards it.
b) Light source sampling is a better strategy for rougher surfaces (bottom row) and smaller light sources. Rough surfaces reflect light more uniformly, and BRDF sampling has issues finding reflection directions towards smaller light sources. An explicit sampling of light sources finds light carrying paths easily, but they may be inefficient because the surface does not reflect too much light in the sampled direction (top left).
c) Multiple importance sampling of both sampling strategies. The advantages of both techniques were retained while the disadvantages were avoided.

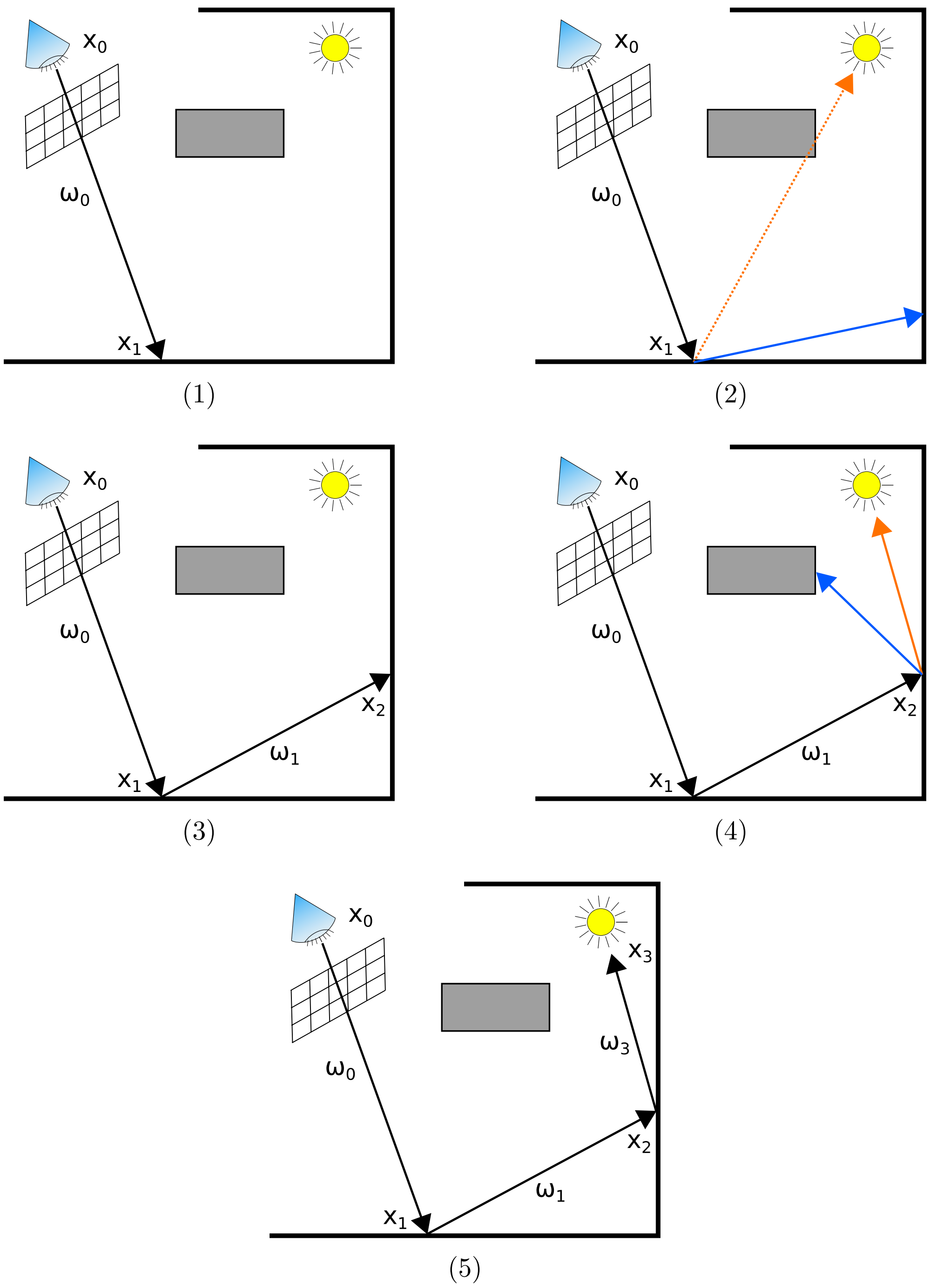


Figure 2.8: Path tracing visualization.
1) Ray $r(x_0, \omega_0)$ is generated from the camera and traced to find point x_1.
2) Direct illumination sampling at x_1. Light source sample (orange) is occluded, and the BRDF sample (blue) has not intersected light source. Zero contribution.
3) ω_1 is sampled from the BRDF at the point x_1. Ray $r(x_1, \omega_1)$ is traced to find x_2.
4) Direct illumination sampling at x_2. The light source sample is not occluded, and the BRDF sample has not intersected light source. Contribution to the path will be added.
5) ω_2 is sampled from the BRDF at the point x_2. Ray $r(x_2, \omega_2)$ is traced to find x_3. Light source was hit, terminating the path. No contribution added, direct illumination estimation at x_2 has already estimated contribution from this path.

2.3.2 Difficult Light Paths

If we want to discuss what kinds of light transport paths are difficult to compute, we need a system for their classification. Heckbert [16] introduced a notation for path classification, and we are going to discuss Veach's extension of it [8]. In this notation, paths are described as regular expressions like:

$$LDD \quad D(SD)^+(S|D)^* \quad SDE$$

Where:

X^+	denotes at least one occurrence of X.
X^*	denotes any number of occurrences of X.
$()$	parentheses express grouping.
$X\|Y$	denotes choice between X and Y.
L	together with the two following letters describes the light source .
E	together with the two preceding letters describes the eye (camera).
S	represents a perfectly specular event like ideal mirror reflection or ideal refraction (BSDF is infinite).
D	represents any other type of scattering from the surface (BSDF is finite).

Although S and D letters in the definition of the light source and camera are not real scattering events like the rest of them, they can be interpreted as such. Out of all the possible variations of great importance to us are especially: SDE, representing pinhole camera (the most often used type of the camera in practice), LDD representing area light source, and LDS representing a point light.

Veach also proved [8] that any algorithm based on a local path sampling, where the continuation of the path is determined at its every point based on the sampling of possible directions (like in the path tracing), is unable to sample a path if it does not contain the substring DD.

These kinds of paths are not rare at all, for example, a simple scene with a pinhole camera, mirror, and a point light:

$$LDS \quad S \quad SDE,$$

contributions of these paths will be missing in an image rendered with a path tracer because no ray from the camera will hit the point light after reflection from the mirror (similar case also in the figure 2.10).

It is possible to avoid this issue by not using perfectly specular surfaces at all, but they are a staple in the rendering practice. Another approach is to introduce DD substring in the camera or the light sources. Algorithms based on the local path sampling will then produce unbiased results, but it will not make previously impossible situations easy. If we switch the point light from the example above for an area light, then the smaller the area light is, the closer (in the limit) we get to the original situation. Therefore even if the algorithm is unbiased, we will observe extremely high variance, which will make the algorithm unusable for the rendering of such a scene in practice.

But this is not the only problematic case. In truth, any path with a substring SD is, to a varying degree, hard to evaluate. These kinds of paths represent *caustics* that can be observed all around us: light coming through the window, light

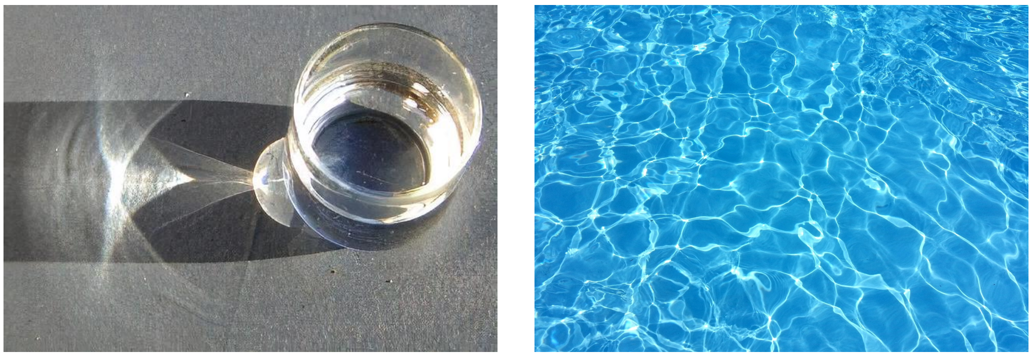


Figure 2.9: Examples of caustics.
Glass with water (Source: Otterstedt 2006 [17]) and pool viewed from top. [2].

bulb illuminating the room, light focused through the glass or the characteristic shimmering of the pool or the ocean (figure 2.9).

The reason why these paths are problematic is that if we trace a path from the camera, we do not know what direction to choose at the diffuse surface to hit the light source (or any other particularly interesting part of the scene) after the subsequent perfectly specular refraction (figure 2.10)). We could also try to trace the paths from the light towards the camera, but the same issue would come up with paths containing substring DS and if the path contains substring SDS neither of the approaches or even their combination is no longer able to efficiently resolve it.

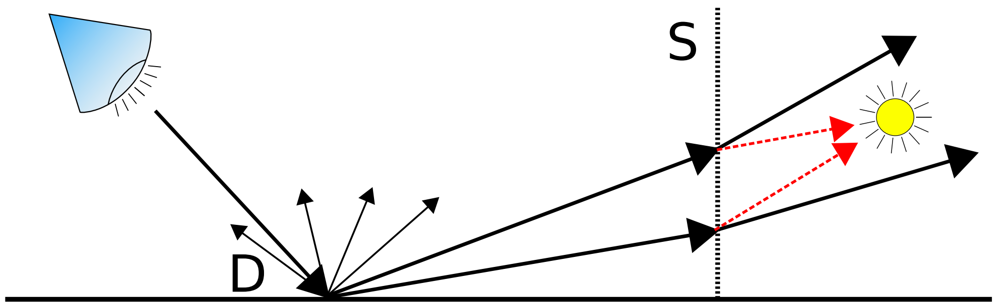


Figure 2.10: Path tracing of DS path.
Path tracer samples continuation of the path at the diffuse surface but it does not take into account the specular boundary in front of the light source that will refract the rays, therefore it can not efficiently generate paths that will lead to the light source (red connections are impossible to use because of the ideal refraction). In the case of the point light, it is unable to find any light carrying paths: $LDS\ SD\ SDE$ does not contain DD substring.

[2]Public domain, https://www.publicdomainpictures.net/en/view-image.php?image=126102&picture=pool-water [Online; accessed: 2020-07-20]

2.3.3 Overview of Light Transport Algorithms

Algorithms presented in this section are not all light transport algorithms ever made - we only cover the most researched and often used ones currently. Also, keep in mind that we introduce them in their original, not state of the art versions.

- *Path tracing* (see section 2.3.1) traces the paths from the camera to recursively solve the rendering equation. It is one of the simplest and most used unbiased algorithms.
- *Bidirectional path tracing* (introduced by Lafortune and Willems [18]) traces paths from both the camera and light sources and connects them into complete light-transmitting paths. It is an unbiased algorithm that can handle paths containing substring DS, making it a good option for resolving many different caustics. Paths containing substring SDS are still problematic.
- *Metropolis light transport* (introduced by Veach and Guibas [19]) is an unbiased algorithm that takes a unique approach to the exploration of the path space. It generates a path in the scene by mutation of the previous one in a way that ensures that the distribution of the sampled paths is proportional to the contributions they have. MLT can perform local exploration of path space: when it finds a path with a high contribution, it can find other similar paths, which allows MLT to compute many caustics efficiently.
- *Photon mapping* (introduced by Jensen, see: [20]) methods are based on the idea of tracing of photons from the light sources, saving them on the scene surfaces and then using them to approximate incident illumination during subsequent path tracing. These methods are able to handle even paths without the substring DD, and resolve indirectly viewed caustics (SDS subpaths) well. But because there is a limited amount of photons traced, the approximation of illumination causes bias.

3. Evaluation of Light Transport Computations

In this chapter, we introduce a methodology for the comparison of light transport algorithms (3.1). Based on it, we specify the needs of the supporting evaluation framework (3.2) and its test dataset (3.3). We also discuss which rendering frameworks we decided to support first (3.4).

3.1 Methodology

It should not come as a surprise that evaluation of renderers, which are complex systems, is difficult. That is especially the case with regards to the light transport algorithms, which are the part of the renderer that ties everything together. It is impossible to test and evaluate them in the vacuum - indeed if there is no scene in which the light travels, there is no light transport to be simulated. And if there is a scene there are also light sources, individual objects with their different materials, and many other pieces that are necessary for the rendering. Therefore, if there are issues with the resulting image, it may be hard to tell what part of the system is responsible. Additionally, if we want to compare as many light transport algorithms as possible, we need to do that across different rendering frameworks, which only brings extra difficulties. So in the first place, it is necessary to discuss what can we evaluate, if there are good reasons to attempt that, and only then think about how it can be done.

3.1.1 Motivation

Multiple properties of light transport algorithms could be evaluated - correctness and efficiency, rendering speed, but also things like level of supportable artistic control (for example, disabling shadows of an object). To discuss them in turn:

- Correctness and efficiency
  - There is only one physically correct result of the rendering of a given scene
  - Algorithms have varying efficiency, and when confronted with more intricate light transport situations, they may even be unable to properly render some scenes
  - Scenes, in general, are not fully reproducible across different rendering frameworks
- Rendering speed
  - Depends on the performance of the hardware and the level of implementation optimization
  - The same algorithm across various rendering frameworks can have different rendering speed

    - Directly usable only when comparing algorithms in the same rendering framework

- Artistic control
    - Non-physically-based features used to fulfill an artistic vision, not to simulate proper light transport
    - Impossible to define fair rating system - the importance of supportable feature depends on the application
    - Important in proprietary renderers used in production (product visualization, movies)

Out of all three discussed properties, only the correctness of the result and to some degree efficiency of the algorithm on difficult scenes can be directly compared between light transport algorithms implemented in different rendering frameworks. These are also the most important properties of a physically-based renderer, and so it would be our choice of what to evaluate.

The questions, whenever an algorithm solves light transport situations correctly or if it can even solve them at all, are fundamental to the physically-based rendering. There are numerous rendering frameworks developed independently, and if we could efficiently compare them, we would be able to spot any differences in their results and ask questions about why they happen, and ultimately, which result is correct and which one is not. This would enhance the development of all evaluated renderers, spotting any of their hidden flaws, and help with their growth towards a common goal - correct and efficient physically-based rendering.

It is important to point out that no implementation of light transport algorithms can produce a physically-correct rendering. Although the theory presented in chapter 2 is mathematically sound, and it is correctly applied in many light transport algorithms, there are still practical issues that can not be resolved. For example, it is impossible to trace light paths of infinite length, even though, based on the recursive nature of the rendering equation, they exist and contribute to the illumination in the scene. Thus, if we want to compare two light transport algorithms, we must make sure that the same set of limitations is applied to both of them, so they resolve the same subset of the global illumination of the scene.

3.1.2 Approach

We can tell whenever two renderings of the same scene converge to the same result with their direct pixel-by-pixel comparison. Specifically, we ask if, and by how much are pixels of an image brighter or dimmer in comparison to the other image (usually absolute or relative differences between pixels are used). These differences are usually visualized as difference image (figure 3.1).

We can say that two renderings of the same scene converge to the same result if their difference image shows only random noise caused by the randomness of sampling of the Monte Carlo integration (figure 3.1).

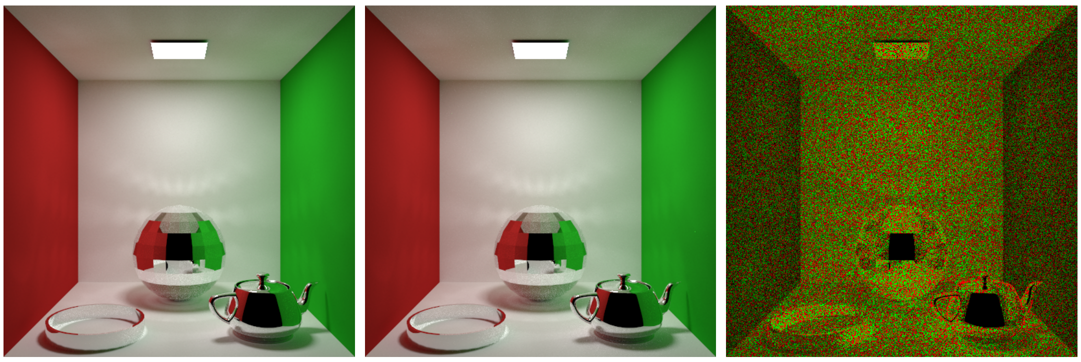


Figure 3.1: Difference image - convergent.
From left to right, these images showcase Mitsuba and PBRT renderings of the same scene and their difference image. Because we can observe red-green noise in the majority of the difference image, we can with confidence declare that both renderers converge to the same result. The only difference mainly at the edges of the teapot could be explained by differences in ray casting, or we would alternatively have to look more into the behavior of light transport at grazing angles.

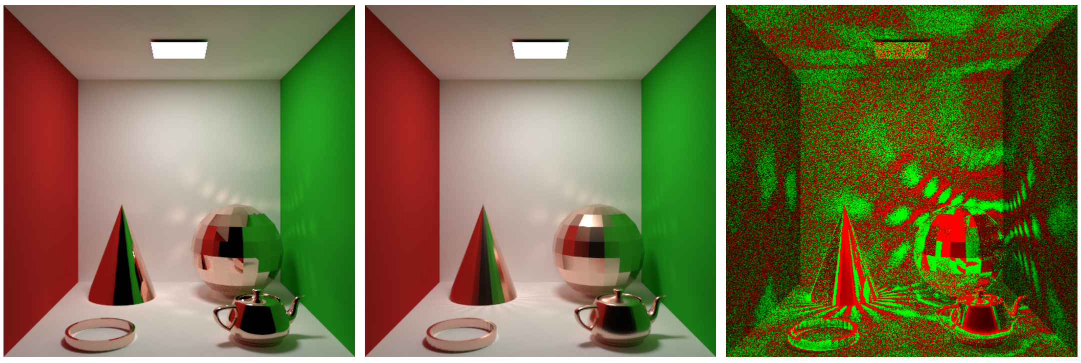


Figure 3.2: Difference image - divergent.
From left to right, these images showcase Mitsuba and PBRT renderings of a similar scene and their difference image. Even when observed by eye, we can spot many differences. If we would try to look for them on the walls, we would be a bit more hard-pressed to find some of them, but we can not miss them on the difference image. The cause of the differences is twofold: material of the objects in the Mitsuba scene is more specular and objects have smooth surfaces in comparison to PBRT with faceted objects and rougher surfaces.

So if we have a physically-based and highly converged rendering of a scene (reference image) and an algorithm that we want to test, all we need to do is render the same scene with it and evaluate the difference image made from comparison with the reference. If some usually continuous part of the difference image has the same color (i.e. part of the image is too bright or too dim in comparison to the reference), we can immediately spot it and get to know that the algorithm gives incorrect results for these pixels (figure 3.2). Usually, we can then proceed with an educated guess about what is causing the issue.

To render an image of the same scene with light transport algorithms from different rendering frameworks, we can choose two approaches:

- Reimplement algorithms in the rendering framework of choice and render the scene with them
- Reproduce the scene in all rendering frameworks and render it with the native implementations of the algorithms

Reimplementation of the algorithms is not feasible, because we can not assume that source code for the algorithms will be available. Additionally, it is not a sustainable long-term solution because each existing and newly introduced algorithm would have to be reimplemented.

Scenes, on the other hand, have to be created only once (for each rendering framework). Although it is practically impossible to exactly reproduce an arbitrary scene in different rendering frameworks, as long as we have scenes that are made to minimize any potential issues of their translation, we should achieve very good or even perfect reproduction of them in many cases.

Even though matching scenes would be possible to use on their own, if we want to efficiently evaluate many algorithms with varying parameters, it will quickly become necessary to have an evaluation framework that would hide as many details of the individual renderers as possible, handle interaction with them and save and present the results. We can not expect potential users to master all the different rendering frameworks only to get a basic evaluation of the rendering algorithms of their interest.

In the following two sections, we summarize properties that the evaluation framework and scenes should have.

3.2 Evaluation Framework

For an evaluation framework, to properly and efficiently support testing of light transport algorithms across different rendering framework, it must:

- *Be automated.* The framework should be able to handle multiple renderers using different light transport algorithms (with varying settings) all at once with minimal user involvement. We want the user to need to interact with the evaluation framework only once - during its launch.
- *Be extensible.* New scenes, light transport algorithms, and even whole renderers are created continuously. If they could not be easily supported by the evaluation system, it would defeat its purpose.

- *Be easy to use.* If the entry-level for the use of the framework is too high or its learning curve too steep, some potential users won't even try to use it. Frequent use cases must be readily available without the need to adjust the inner workings of the evaluation framework. The only exception to this may be extensions to supported renderers.
- *Be transparent.* The inner workings of the evaluation system should be clear to the users who will try to delve into them. The code of the framework will be freely available and as straightforward as possible.
- *Present evaluation results.* As we have discussed, the results of the evaluation will be difference images that come from the comparison of rendered images with each other or with reference images. We want to facilitate a straightforward way to quickly view them without a need for any additional software.

3.3 Evaluation Scenes

An evaluation framework with no test data would not be complete at all. Therefore, we need to provide a set of scenes that can be used for the evaluation of light transport algorithms across different renderers. Specifically, we want the resulting set of scenes to:

- *Cover as many different light transport situations as possible.* If a light transport situation is not covered by the current set of scenes yet, we want to include it with an additional scene.
- *Avoid unnecessary complexity.* Our scenes will be provided for testing and comparison of different algorithms - not to look nice. Additions that do not fundamentally enhance or change the light transport situation in the scene should be omitted. Their cost would be an increased rendering time and detraction from the testing purpose of the scene.
- *Be as small as possible.* We do not want to limit the number of scenes, but we should avoid several scenes testing the same light transport situation as long as there is not a valid reason to include them all.
- *Be exactly documented.* Both what is tested and how is the scene designed to facilitate that, must be documented. It is important for organization, development, and presentation of the scenes to the users.
- *Be compatible with as many potential rendering frameworks as possible.* We want to avoid any features native to a specific renderer, and build our scenes from as simple building blocks as possible.
- *Provide reference images for all the supported renderers.*

3.4 Supported Renderers

Although our evaluation framework will be extensible, we can only provide support for a limited number of renderers from the get-go. To support a renderer we need:

- Infrastructure for the handling of renderer settings and its scene files (code)
- Scenes in a file format supported by the renderer (data)
  - Made so the renderer will produce comparable results to other already supported renderers (ideally renderers should produce the same result on all scenes).
- Reference images of the scenes (especially in a case when different renderers do not generate same resulting image)

As a first step, we want to support the two most popular research-oriented open-source renderers: Mitsuba and PBRT. These renderers have similar scene description formats and/or capabilities like a lot of other open-sourced renderers (e.g. LuxCoreRenderer[1] or Tungsten Renderer[2]), so they can also work as an example implementation in our evaluation framework for a lot of additional renderers. We do not plan to provide support for any proprietary renderers in the scope of this thesis. As we have discussed in the introduction, it is almost impossible to replicate the same rendering results between them, because of their focus on rendering speed and user's efficiency and not so much on the physical correctness of their solutions.

3.4.1 Mitsuba 0.5

Mitsuba [2] is the most heavily used rendering framework in computer graphics research. It is highly modular - each piece of functionality is implemented as an independent module, so new algorithms are simple to add. It offers support for large-scale parallelization (rendering on a server) and even its single-threaded execution is quite optimized (SIMD). We have picked version 0.5, which is available on the main web page. There is also version 0.6, which must be downloaded from the repository[3] and manually compiled. Additionally, there is version 2.X [21], which is a completely new modernized rendering framework meant to replace older versions of Mitsuba.

The decision to use (and stick) with version 0.5 was based on multiple factors:

- Scenes showcased in recent research papers often come up from a set of scenes originally provided by Benedikt Bitterli [22]. Mitsuba scenes in this set are of version 0.5.
- We are not aware of any difference between scene description formats of versions 0.5 and 0.6. All scenes which we were working on were possible to render with version 0.6.

[1]https://luxcorerender.org/ [Online; accessed 2020-07-20]
[2]https://github.com/tunabrain/tungsten [Online; accessed 2020-07-20]
[3]https://github.com/mitsuba-renderer/mitsuba [Online; accessed 2020-07-20]

- We have tested version 0.6, and the rendering results of our scenes were the same as with version 0.5.
- Mitsuba 2.X is backward compatible - able to take older scene description versions as an input.
- Mitsuba 2.X was a newly available software at the time of work on this thesis. It may have had some unknown and unresolved bugs, and many of the light transport algorithms of older versions were not yet supported by.

3.4.2 PBRT 3

PBRT [3] is both an open-source renderer and accompanying book explaining both its implementation and theory behind it. With its focus on the educational aspect, it is not as well optimized, but it makes up for that with its straightforward, easy to understand code and good documentation. We have used its latest version: PBRT 3. It is not backward compatible, but nowadays, there is no reason to use older versions.

4. Evaluation Framework

In this chapter, we present a framework for the evaluation and comparison of light transport algorithms across different rendering frameworks. We discuss its purpose (4.1) and explain its requirements (4.2), installation process (4.3), and structure (4.4). Following that, we present the main features of the evaluation framework contained in three of its scripts: *lteval.py* (4.5), *ltevalwebgen.py* (4.6), *ltevalwebdisplay.py* (4.7). Then we at length explain configuration files, which are used as an input of the evaluation framework (4.8). Finally, we describe some of its implementation details, including methods of its extension (4.9).

4.1 Purpose

The purpose of the (light transport) evaluation framework is to ease up evaluation of light transport algorithms based on the comparison of rendered and reference images (for full discussion refer to 3). For this reason, it needs to be possible to select scenes to be rendered and specify light transport algorithms and their parameters (*test cases*) that should be used for the rendering. All of that, while supporting different rendering frameworks, having no knowledge of specific light transport algorithms and their parameters, maintaining reasonable ease of use and allowing the addition of new scenes and rendering frameworks. Additionally, the evaluation framework can generate a simple website for a fast evaluation of the rendered images.

4.2 Requirements

The light transport evaluation framework is operating system agnostic, but few requirements must be fulfilled for its use:

- *Python 3 environment of version 3.7.4.* The framework was developed and tested with this version, but there is nothing of our knowledge that could cause issues when using higher versions.
- *Python packages: Dominate*[1] *and lxml*[2]. Both packages can be installed with PIP, or with the help of a Python environment provider (e.g., Anaconda). Other necessary packages should be installed as part of the Python distribution.

Python editor and basic knowledge of the Python language are recommended, but not necessary. Some input files must be written in Python, but the framework is easy to use even with no prior experience of the language.

For practical use of the framework except for the generation of supplementary website (see 4.6), renderer executables are necessary. Evaluation framework currently supports Mitsuba 0.5, and PBRT 3 renderers (3.4) whose executables for Windows are provided together with it. To use additional renderers, the framework must be extended (refer to 4.9.4).

[1]https://pypi.org/project/dominate/ [Online; accessed 2020-07-20]
[2]https://pypi.org/project/lxml/ [Online; accessed 2020-07-20]

4.3 Installation

The light transport evaluation framework, including the evaluation scenes (5), does not need any specific installation steps. It can be extracted from the attached zip file provided together with this document, or its current (updated) version can be downloaded from a GitHub repository[3].

4.4 Structure

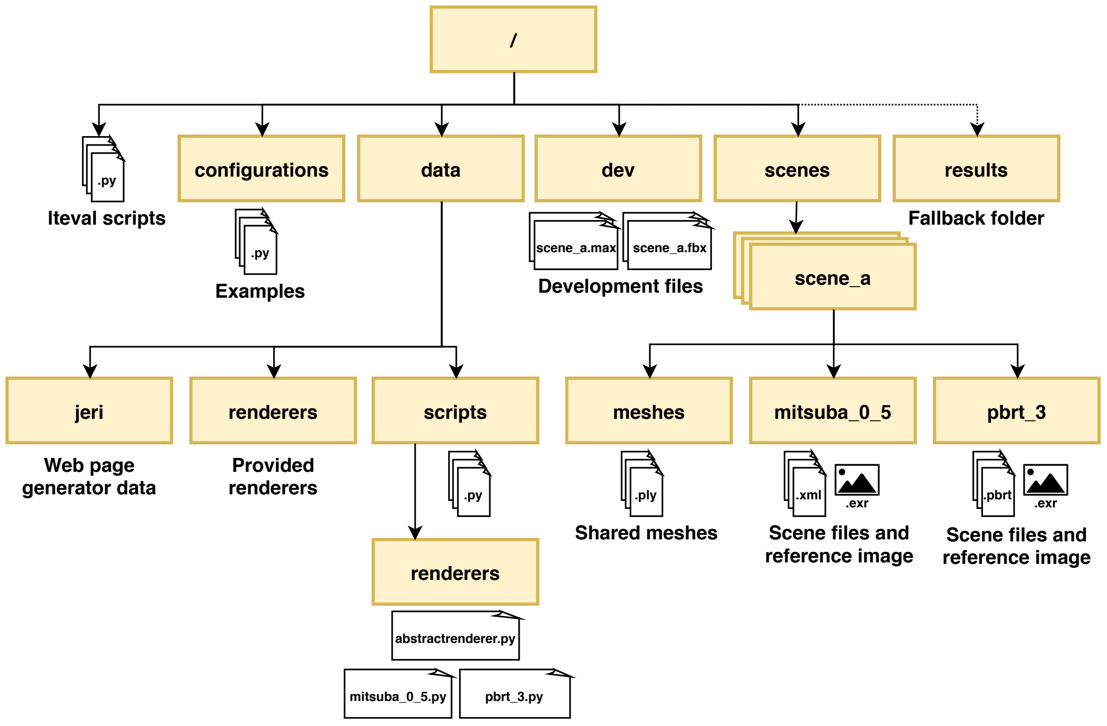


Figure 4.1: Evaluation framework folder structure.

Figure 4.1 presents the folder structure of the evaluation framework. Its understanding is not essential for the use of the framework, but it is necessary for its modifications, including its extensions.

- The main features of the evaluation framework are available in the form of three python scripts in its root directory: *lteval.py*, *ltevalwebgen.py*, and *ltevalwebdisplay.py*.
- *configuration*: Examples of configurations files that are taken as a mandatory parameter of the main *lteval.py* script.
- *data*: Other parts of the framework.
  - *jeri*: Data necessary for the generation of supplementary website.
  - *renderers*: Executables of Mitsuba 0.5 and PBRT for Windows, provided together with the framework.

[3]https://github.com/tazlarv/lteval

  - *scripts*: Additional Python scripts necessary for the run of the framework.
  - *scripts/renderers*: Python scripts providing support for specific rendering frameworks - support for Mitsuba 0.5 and PBRT 3 is provided.

- *dev*: Autodesk 3ds Max scenes created during the development of the evaluation scenes. FBX version added for use in other modeling software.
- *scenes*: Evaluation scenes of all supported renderers. In a folder named after the scene, we can find subfolders for each supported rendering framework (containing scene files and a reference image of the scene) and shared 3D geometry files (meshes).
- *results*: Used for saving of results (rendered images, generated website) if no result folder is specified.

4.5 lteval.py

The *lteval.py* script is the main script of the evaluation framework. Its job is to render evaluation scenes based on settings in the configuration file (the only mandatory parameter of the script). Configuration file describes what scenes should be rendered and what light transport algorithms and parameters should be used for it (see 4.8).

Execution of the lteval.py consists of the following steps:

1. Load a configuration file.
2. Create an output folder.
3. Load scripts for the handling of necessary rendering frameworks.
4. Load specified evaluation scenes.
5. Prepare individual test cases specified in the configuration file (parameters of light transport algorithms to be used).
6. For every scene and every test case:
   (a) Generate a new, modified scene file based on the test case. This file is saved in the same folder as the evaluation scene (e.g., scenes/scene_a/mitsuba_0_5 - see 4.4) named __lteval_{test case name}.
   (b) Render the modified scene.
   (c) Copy the resulting image and reference image of the scene to the output folder (figure 4.2).
7. Potentially generate a website showcasing the rendered images.
8. Potentially open up the generated website.

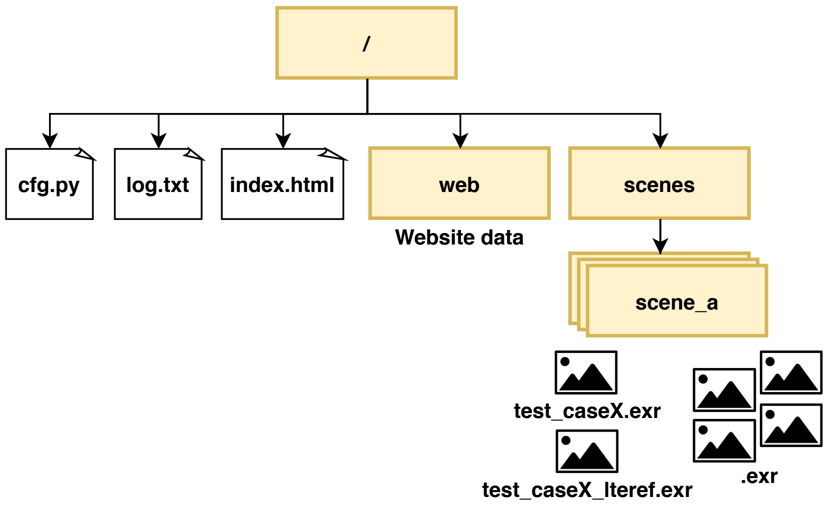


Figure 4.2: Output folder structure.

cfg.py: Copy of the configuration file used to run the lteval.py script that generated this output folder.

log.txt: Console output of the evaluation framework. Includes console output from the renderers.

index.html: Index page of the website generated by ltevalwebgen.py, web folder contains additional data.

scenes: All rendered scenes. Subfolders, named after individual scenes, contain images rendered by lteval.py and corresponding reference images. Images are named after the individual test cases, while references have an additional _lteref suffix. Additional images are handled as if they were produced by the lteval.py script.

Other than the configuration file, which is a mandatory parameter, *lteval.py* script offers few optional arguments:

- -c, --c n, y, fy: Clear - specifies what to do with the generated scene files from step 6a. They may be left alone (n), deleted when the rendering successfully finishes (y - default), or always deleted even if the rendering fails (fy). We recommend the default because invalid scene files generated from incorrectly set up configuration files are often the cause of failed rendering.
- -fc, --fc: Force clear - deletes all generated scene files (from previous runs). If set, the configuration file is ignored, therefore can be specified as an empty string: `python lteval.py "" -fc`.
- --eof (default), --no-eof: End on failure - whenever execution of the script should be stopped when rendering of scene fails.

4.6 ltevalwebgen.py

The *ltevalwebgen.py* script generates a website that allows immediate evaluation of the *lteval.py* renderings. It has one mandatory parameter - path to the output folder of one of the *lteval.py* runs. After a search of the output folder for rendered images is completed, *ltevalwebgen.py* builds up the website, including not just the expected renderings (ones specified by the configuration file of this output folder) but everything it has found. Therefore, if the output folder has an expected structure (figure 4.2), *ltevalwebgen.py* can generate the website, making it usable even outside of the evaluation framework. Currently, only .exr images are supported, but this is not a big limitation if we consider that most of the renderers can render images in this format.

There is also one optional parameter available:

- -d, --d: Display - displays the generated website with *ltevalwebdisplay.py* (default parameters).

4.7 ltevalwebdisplay.py

The generated website can not be displayed directly because of the HDR (high dynamic range) images produced by rendering - they need special treatment. Therefore, the *ltevalwebdisplay.py* script is necessary to handle the displaying of the website. It starts a web server on the local machine and opens up the generated website in the browser. It takes one mandatory parameter, which is a path to the output folder (same as *ltevalwebgen.py*) or the *index.html* file inside of it.

One optional parameter is available:

- -p, --p: Port - port of the local web server (default 8000).

The main web page (*index.html*) of the displayed website presents a list of test cases with their descriptions and a list of scenes with links to additional web pages for viewing and evaluation of their renderings (figure 4.3).

It is highly recommended to have only a single *ltevalwebdisplay.py* running on a port, otherwise, the behavior of multiple webservers on the same port becomes unpredictable.

Execution of the script can be stopped by closing the console or sending an interrupt signal (SIGINT) with `Ctrl+C`. If the script does not react to the interrupt signal immediately, we recommend forced refresh of the web page in the browser (usually `Ctrl+F5`) - so the preferred order is to try to interrupt the script before the web page is closed in the browser. The same recommendation applies whenever multiple *ltevalwebdisplay.py* scripts were run on the same port. Forced refresh clears the cached files of the browser before it reloads the web page. These issues happen because of caching on the side of the browsers and interactions of browsers with the webserver, where we do not have control over what happens.

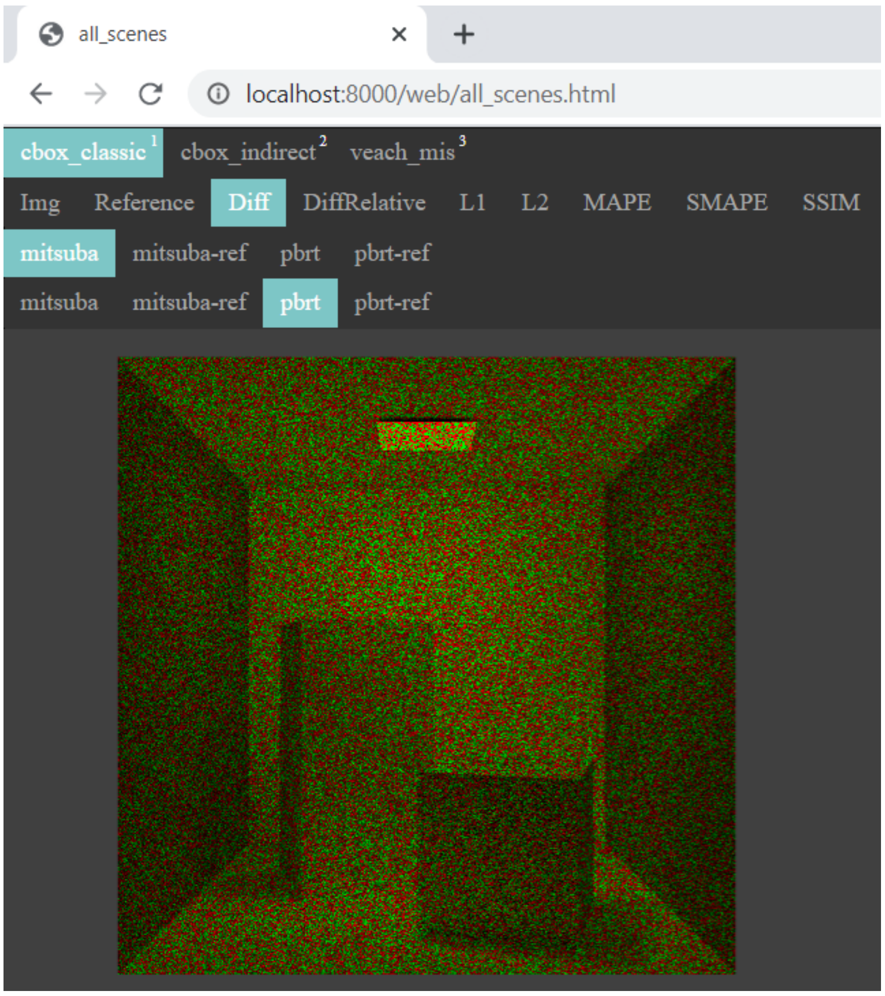


Figure 4.3: Evaluation framework website - image viewer.
One of the available web pages on the website presenting all rendered scenes in a single image viewer. We can observe from the URL that the web page is hosted on the current computer on port 8000.

We can select the displayed image in the menus:

First (top) row - scene selection.
Second row - image type selection. Following rows depend on the selected type.

For *Img* and *References*, a third row will allow selection from rendered test cases of the scene and their references, respectively.

The rest of the available options offer various ways of image comparison. If any of them is selected, two additional rows are displayed to select images to be compared. In the figure, we can observe a difference image of *mitsuba* and *pbrt* test cases in the cbox_classic scene.

It is possible to move the image around (mouse) or zoom in and out (scroll wheel). Additional options (e.g., increasing or decreasing of the image exposure) are described in the help menu opened with **?** shortcut. Note that all the shortcuts presented in the help menu are case sensitive.

4.8 Configuration File

Configuration files are the centerpiece of the evaluation framework. They, as the only mandatory parameter of the *lteval.py* script, specify what light transport algorithms with their settings are used to render selected evaluation scenes.

Because the configuration file (figure 4.4) is a Python file, proper Python syntax is required, and all strings and identifiers are case sensitive. When it is loaded by *lteval.py* it is executed as a Python script, so it is possible to use any valid Python code inside of it. At the end of the execution of the configuration file, the four following variables must be defined and have required content: *configuration*, *scenes*, *renderers*, and *test_cases*. That can be easily done even without knowledge of Python because baring a few details syntax is the same as in JSON - popular and simple file format.

Note that we can specify any path in the configuration file as absolute or relative. Relative paths are relative to the folder of the *lteval.py* script that executes the configuration file. Also, we recommend using forward slashes `/` instead of backslashes, which need to be escaped `\\`.

In the following sections, we discuss all parts of the configuration file from figure 4.4. This file and other additional commented examples of configuration files can be found in the *configurations* folder of the evaluation framework (see 4.4).

4.8.1 configuration

This dictionary allows us to control some of the decisions made by the *lteval.py* script:

- `"name": string` - Name of the configuration and the output folder. If not defined, the name of the configuration file is used instead.
- `"description": string` - Description of the configuration, shown on the index page of the generated website.
- `"output_dir": path` - Path to the folder inside of which the output folder of the configuration will be created.
- `"output_dir_date": boolean` - Whether the current date and time should be appended to the name of the output folder. This allows us to repeatedly run the same configuration without overwriting results from its previous runs.
- `"webpage_generate": boolean` - Whether the *ltevalwebgen.py* website should be generated in the output folder.
- `"webpage_display": boolean` - Whether the generated website should also be displayed with *ltevalwebdisplay.py*.

All of these parameters are optional, but because the *configuration* variable itself is mandatory, at least `configuration = {}` must be defined.

```
# MANDATORY - Dictionary of settings
configuration = {
    "name": "cfg example",  # OPTIONAL, default: name of the configuration file
    "description": "Configuration file basic example",  # OPTIONAL
    "output_dir": "results",  # OPTIONAL, default: results
    "output_dir_date": True,  # OPTIONAL, default: True
    "webpage_generate": True,  # OPTIONAL, default: False
    "webpage_display": True,  # OPTIONAL, default : False
}

# MANDATORY - List of scenes
scenes = ["cbox_classic", "cbox_indirect"]

# MANDATORY - Dictionary of renderers
renderers = {
    "mitsubaRenderer_0_5": {
        "type": "mitsuba_0_5",  # MANDATORY
        "path": "data/renderers/mitsuba_0_5/mitsuba.exe",  # MANDATORY
        "options": "",  # OPTIONAL
    },
}

# OPTIONAL - Dictionary of re-used sets of parameters
parameter_sets = {
    "mitsubaLQ": {
        "integrator": [
            ["type", "", "bdpt"],
            ["maxDepth", "integer", 10]
        ],
        "sampler": [
            ["type", "", "independent"],
            ["sampleCount", "integer", 2],
        ],
        "rfilter": [["type", "", "box"]],
    },
    "mitsubaHQ": {
        "base": ["mitsubaLQ"],
        "sampler": [["sampleCount", "integer", 8]],
    },
}

# MANDATORY - List of test cases
test_cases = [
    {
        "name": "LQ_test_case",  # MANDATORY
        "description": "Fast preview rendering",  # OPTIONAL
        "renderer": "mitsubaRenderer_0_5",  # MANDATORY
        "params": {"base": ["mitsubaLQ"]},  # OPTIONAL
    },
    {
        "name": "HQ_test_case",
        "description": "High quality rendering",
        "renderer": "mitsubaRenderer_0_5",
        "params": {"base": ["mitsubaHQ"]},
    },
]
```

Figure 4.4: Configuration file example.

4.8.2 scenes

List of scenes to be rendered. Names of selected scenes must match the names of folders in the *scenes* directory (see 4.4). Scenes described in chapter 5 are provided as part of the evaluation framework. We discuss how to add a new scene to the framework in section 4.9.4.

As a mandatory variable, the minimal definition is: `scenes = []`. But then no scene is selected, thus nothing is rendered.

4.8.3 renderers

Dictionary of renderers that may be used for rendering of *scenes* with settings defined in *test_cases*.

Every renderer in the dictionary is identified by its key (e.g., mitsubaRenderer_0_5 in figure 4.4), and defined by the dictionary value, which is another dictionary with the following parameters:

- `"type": string` - Type of input (scene) files of the renderer. Type must be the same as one of the scripts in the *data/scripts/renderers* folder (see 4.4). These scripts are used to prepare input files for the renderer based on the *test_cases*.
- `"path": string` - Path to the executable of the renderer.
- `"options": string` - Command-line parameters that are passed to the renderer when rendering starts.

Note that renderer is defined just as a named executable with a specified input file format. This loose definition is important - it allows us to define multiple renderers with different executables and same scene format (testing of different algorithms across derivations of the same rendering framework), or to use a new version of the renderer which is backward compatible with older input format.

Same as with *scenes*, we need to define *renderers* variable (at least `renderers = {}`), but if there is no renderer, we will be unable to define *test_cases*, hence unable to render anything.

4.8.4 parameter_sets

Parameter sets represent named groups of parameters, which are, in the end, translated to formats understandable by specific rendering frameworks. On their own, they are an abstract structured grouping of values whose meaning depends on their application. Their working is explained in figure 4.5.

```
parameter_sets = {
    # Parameter set name
    "mitsubaLQ": {
        # Dictionary of statements
        "integrator": [
            # List of parameters
            ["type", "", "bdpt"],
            ["maxDepth", "integer", 10]
        ],
        "sampler": [
            ["type", "", "independent"],
            ["sampleCount", "integer", 2],
        ],
        "rfilter": [["type", "", "box"]],
    },
    "mitsubaHQ": {
        # Take base parameter set...
        "base": ["mitsubaLQ"],

        # ...and add/modify parameters of "sampler"
        "sampler": [["sampleCount", "integer", 8]],
    },
}
```

Figure 4.5: Parameter sets.

There are two named *parameter sets*: `mitsubaLQ` and `mitsubaHQ`. Parameter set itself is a dictionary of *statements* with their lists of parameters (*parameter lists*).

A *parameter list* is a list of lists of length three (it is a frequent mistake to forget this when specifying a list with single parameter). We can interpret *parameter* as: `[name - string, type - string, value - anything]`, where both name and type together identify the parameter (so multiple parameters of the same name and different type can be defined). If there are multiple parameters with the same name and type, the value of the last one in the *parameter list* is applied (allows redefinition of parameters). *Statement identifiers* then identify individual *parameter lists* and *parameter sets* group together *statements*.

Every *parameter set* may have one `"base"` *statement*, which is interpreted uniquely (see mitsubaHQ). After the *base identifier* follows a list (not a list of lists) of names of previously defined *parameter sets*.

Content of these sets is merged in the order of their declaration in the *base list* in such a way that *parameter sets* under the same *statement identifier* are concatenated, applying the last definition of the parameter from all the base *parameter sets* (recall the rule that the last parameter of the same name and type applies).

Now, the merged *base* is a current definition of the *parameter set*, and we are free to define any *statements*. Furthermore, their *parameter sets* will take precedence over the *base* ones because they are defined later.

Therefore, we can define completely new *statements* or add new parameters to already existing *parameter sets* or modify values of parameters in the already existing *parameter sets*.

In the `mitsubaHQ` set, we use `mitsubaLQ` as a *base* and only redefine the value of one of the sampler's parameters.

4.8.5 test_cases

List of *test cases* - rendering configurations used for rendering of *scenes*. Each test case is used for rendering of all scenes and is defined with the following parameters:

- `"name": string` - Name of the test case. Test case names must be unique, if they are not, an error message is printed, and execution of the *lteval.py* script stopped.
- `"description": string` - Description of the test case, shown on the index page of the generated website.
- `"renderer": string` - Identifier of renderer defined in the *renderers* dictionary (dictionary keys are the identifiers).
- `"params": parameter set` - *Parameter set* whose parameters will be passed to the renderer. Scenes will be rendered with default settings if the parameter set is empty.

Test case ties together renderer and parameters. Altogether we have a rendering executable, know its input file format and parameters which we want to pass to it.

Because *parameter sets* are abstract groups of parameters, their interpretation may differ between rendering frameworks. We provide support for Mitsuba and PBRT (scripts in *data/scripts/renderers* folder - see 4.4), and as such, we have specified the interpretation of *parameter sets* for them - see figure 4.6.

Note that every *parameter set* needs to be created with a renderer in mind because, among other things, names of parameters vary between different rendering frameworks. Therefore, basic knowledge of the renderers in question is necessary[4].

As a mandatory variable, *test_cases* must be at least defined as an empty list: `test_cases = []`.

[4]Mitsuba: https://www.mitsuba-renderer.org/releases/current/documentation_lowres.pdf, PBRT: https://www.pbrt.org/fileformat-v3.html [Online; accessed 2020-07-20]

```
# Parameter set to be interpreted as Mitsuba settings
"params": {
    "integrator": [["type", "", "bdpt"], ["maxDepth", "integer", 10]],
    "sampler": [["type", "", "independent"],["sampleCount", "integer", 2],],
    "rfilter": [["type", "", "box"]],
},

# Parameter set to be interpreted as equivalent PBRT settings
"params": {
    "Integrator": [["type", "", "bdpt"], ["maxdepth", "integer", 9]],
    "Sampler": [["type", "", "random"], ["pixelsamples", "integer", 2]],
    "PixelFilter": [["type", "", "box"]],
},

<!-- Mitsuba interpretation -->
<integrator type="bdpt">
  <integer name="maxDepth" value="10"/>
</integrator>

<sampler type="independent">
  <integer name="sampleCount" value="2"/>
</sampler>

<rfilter type="box"/>


# PBRT interpretation
Integrator "bdpt" "integer maxdepth" [9]
Sampler "random" "integer pixelsamples" [2]
PixelFilter "box"
```

Figure 4.6: Parameter set interpretation in Mitsuba and PBRT.

Rendering settings of both Mitsuba and PBRT are defined inside of their input scene files (see 1.4.1). Therefore, if we redefine some settings in the *parameter set* (i.e., define *statements* with the same *identifier*), we may need to take the existing evaluation scene files and modify them.

We have not wanted to allow mixing of parameters inside of the scene files with the ones defined in the *parameter lists* of individual *statements* (to avoid unexpected outcomes because of an unknown content of scene files). Thus, any original settings in the scene files are overwritten by the newly defined ones in the *parameter set*. Consequently, any *statement* in the *parameter set* must always define a valid setting of the renderer.

Mitsuba and PBRT file formats are similar in structure, although one is an XML file, and the other is a native text format of the renderer. Each *statement* (setting) has an *identifier* (name) and type, so `rfilter` (Mitsuba) and `PixelFilter` (PBRT) above are minimal valid *statements*. Type is defined with parameter `["type", "", "definition_of_type"]`, and any additional parameters are understood as `["parameter_name", "parameter_type", parameter_value]`. For the exact mapping of *parameter sets* to the rendering settings, refer to the figure above.

Three settings (integrator, sampler, and filter) of this example are the only ones of interest when it comes to comparison of the light transport algorithm of Mitsuba and PBRT on the set of evaluation scenes. Nonetheless, it is possible to specify any other setting in the same way. Also, not just integer, but other kinds of values are supported: float, string, boolean, and lists - refer to examples in the configurations folder (4.4).

4.9 Implementation

The development of the evaluation framework presented in this chapter was guided by the principles discussed in section 3.2. Nevertheless, a few specific questions had to be answered:

- How to display rendered images on a website - 4.9.1.
- How should the configuration file look like - 4.9.2.
- How to pass settings from the configuration file to the renderer - 4.9.3.
- How to extend the framework to support new renderer, renderer file format, or a new scene - 4.9.4.

Also, relationships between different scripts that make up the framework are briefly discussed in section 4.9.5.

4.9.1 Third-Party Software

To display high dynamic range images directly in the website we use following third-party software:

- Javascript Extended-Range Image Viewer (JERI) [5]
  - Modified Apache 2.0 license.
  - Allows displaying of high dynamic range images (EXR) inside a website.
  - Provides the image viewer of the generated website.
  - Extended to include difference and relative difference comparisons.

We have wanted to generate difference images on the fly because there can be any number of test cases, and the number of possible comparisons between all of the resulting images (and reference images) increases with the square of the number of test cases.

JERI allowed to display rendered high dynamic range images and their comparisons (generated on the fly) inside of the browser, and also provided an image viewer, which could be adjusted to our needs.

4.9.2 Configuration File Design

There are many different ways how to design a configuration file for the evaluation framework. Its current presented form tries to achieve the following properties, which were derived from the evaluation framework requirements (3.2):

[5]https://jeri.io/index.html [Online; accessed 2020-07-20]

- *Be simple and efficient.* Writing and adjustment of configuration files is the most time-consuming part of the work with the framework. Therefore configuration files must be as simple as possible. To do so, we tried to minimize the number of mandatory components while retaining other discussed properties. Additionally, we introduced the re-use of named *parameter sets* (`"base"` *statement*), which greatly cuts down the length of the *test_cases.*

- *Handle multiple renderers.* It must be possible to use different renderers or multiple versions of the same renderer to allow direct comparison of their algorithms in the framework. All of that is possible with the current definition of renderer as a named executable with a specified input type (see 4.8.3).

- *Handle unknown settings and parameters.* We do not know what algorithm will be used and what are its valid parameters. Thus, we need to be able to take any set of parameters and pass them to the renderer. This is achieved by the translation of *parameter sets* into a renderer input with a fixed and straightforward approach that ignores the purpose of the parameters (figure 4.6).

- *Avoid ambiguos parameters.* All rendering settings defined in the configuration file are applied to evaluation scenes, which have their default rendering settings. Therefore, we could theoretically use the default settings as an implicit `"base"` *parameter set*, but we refrained from doing so. Currently, any setting defined in the configuration file fully overwrites the default one. So, for every setting, there is a full or no definition of it in the configuration file. This helps to avoid unexpected rendering results caused by the application of default parameters, which are not directly accessible from the configuration file.

4.9.3 Scene Files and Passing of Parameters

Passing of settings from test cases to renderers and handling of scenes files are closely tied together. Recall that some renderers have their settings specified inside of scene files (1.4.1), which is exactly the case of both Mitsuba and PBRT. Therefore, if we want to render a scene with modified settings, we need to create new scene files based on both the original scene files and settings defined in the test case.

Before we describe how are these new scene files created, it is necessary to explain how are Mitsuba and PBRT scenes stored in the evaluation framework. Take *scene_a* from figure 4.1, there is a folder for 3D objects of the scene (meshes) but also a folder with a version of the scene for each of the supported rendering frameworks. Content of these folders is the same for Mitsuba and PBRT:

- *description.suffix* - Scene description (geometry, light sources, materials).

- *settings.suffix* - Renderer settings, including camera (everything that is not part of the scene description).

- *scene.suffix* - Main scene file, includes *settings.suffix* and *description.suffix.*

- *reference.exr* - Reference rendering (image) of the scene.

Where *.suffix* depends on the renderer (.xml or .pbrt). The division of the scene into *description* and *settings* files reflects the structure of Mitsuba and PBRT scene formats. For particulars of this division see any of the provided scenes. The *scene* file is not necessary for the rendering of the scene with the evaluation framework, but it allows to render the scene outside of the framework.

When the evaluation framework needs to render a scene, it:

1. Reads the *settings* file.
2. Overwrites any settings which are defined in the *test case*.
3. Appends any remaining settings from the *test case*.
4. Appends directive to include the *description* file.
5. Outputs the modified version of *settings* file next to it as *__lteval_{test_case_name}.suffix* file. It is used as an input of the renderer.
6. After the rendering finishes, this file is deleted (can be controlled - see 4.5).

It would be possible to do this even without the division of the scene into multiple files. But that may not be the case for other renderers (e.g., scene file passed to the renderer together with settings as command-line arguments), moreover division to settings and scene description is quite natural and increases performance because the description does not have to be loaded.

4.9.4 Extending the Framework

For all types of extensions, we first and foremost recommend having a look at the presented evaluation framework. Its support of Mitsuba 0.5 and PBRT 3 is implemented as would any other extension, and the same applies to the provided evaluation scenes.

New Scene

To add a new scene, a corresponding folder must be created in the *scenes* folder of the evaluation framework with subfolders containing the required files for each of the supported scene formats (figure 4.1). Although requirements may differ, they are very similar for both currently supported scene formats (Mitsuba 0.5 and PBRT 3), refer to 4.9.3.

New Renderer

To add a new renderer that uses already supported scene format, nothing needs to be done - the renderer can be defined in the configuration file whenever it is needed (4.8.3). If the scene format of the renderer is not supported, it must be added to the evaluation framework first - refer to the following section.

New Scene Format

To support a new scene format, a corresponding script in *data/scripts/renderers* must be created (figure 4.1). The name of this script must be the same as the name of the scene format, and it must contain exactly one class derived from the abstract base class *AbstractRenderer*, which is declared inside of *data/scripts/renderers/abstractrenderer.py*. For implementation examples refer to the *mitsuba_0_5.py* and *pbrt_3.py* scripts.

4.9.5 Architecture

The architecture of the evaluation framework is presented in figure 4.7. This figure showcases the import relationship of individual Python scripts that make up the framework:

- Public (front-end) scripts
  - *lteval.py* - main script of the evaluation framework.
  - *ltevalwebgen.py* - generation of website presenting images rendered by the evaluation framework.
  - *ltevalwebdisplay.py* - displaying of the generated website.
- Implementation of main features
  - *lteutils.py* - implementation details of *lteval.py*.
  - *webgen.py* - implementation of features provided by *ltevalwebgen.py*.
  - *webdisplay.py* - implementation of features provided by *ltevalwebdisplay.py*.
- Helper scripts
  - *futils.py* - methods for importing of modules from files. Used for on-demand loading of configuration files and scripts providing support of rendering frameworks.
  - *outputconsts.py* - constants, which specify names of subfolders and files in the output directory.
- Support of rendering frameworks
  - *abstractrenderer.py* - abstract base class of classes providing support of rendering frameworks.
  - *mitsuba_0_5.py* - support of the Mitsuba 0.5 rendering framework.
  - *pbrt_3* - support of the PBRT 3 rendering framework.
- Handling of basic objects.
  - *scene.py* - loading and representation of scenes in the framework.
  - *tcase.py* - loading and representation of test cases in the framework.

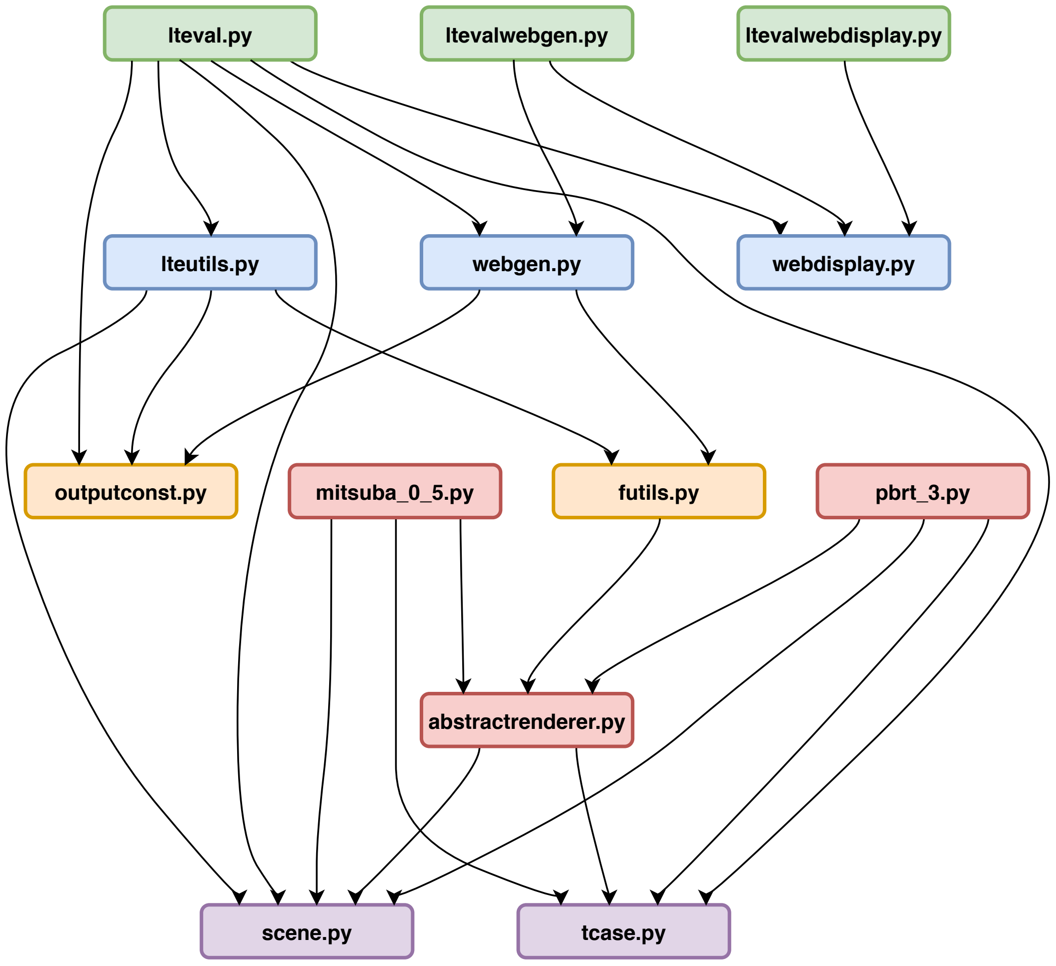


Figure 4.7: Evaluation framework import graph.
Green - public (front-end) scripts. Blue - implementation of the main features. Yellow - helper scripts. Red - support of rendering frameworks. Purple - handling of basic objects.

5. Evaluation Scenes

In this chapter, we present a set of test scenes for evaluation of light transport algorithms. We explain the general decision process used to determine whenever a new scene should be added to the set and its application and consequences (5.1). Then we discuss the development of these scenes, issues that had to be resolved, and limitations imposed by trying to match scenes across different rendering frameworks (5.2). Finally, we showcase the resulting documented set of scenes (5.3).

5.1 Introducing New Scene

Whenever an addition of a new scene is considered, we need to keep in mind properties we want the evaluation set to have (introduced in 3.3). Especially important is to try to cover as many different light transport situations while avoiding multiple scenes for evaluation of the same case of light transport.

To achieve that, we have followed these steps for any potential new scene:

1. In a few words or short sentences describe (brief description), what light transport situation is tested by the scene. The longer the description needs to be, the bigger chance that the scene is too complicated and should be simplified.

2. Describe the light transport situation of the scene in detail (complete description).

3. Go over the current scenes in the set and find any that test similar light transport situations as the new scene. It should be possible to filter out most of the scenes with just a comparison of the brief descriptions, and for the remaining ones, complete descriptions need to be analyzed. This step has 3 possible outcomes:

    (a) No scene was found - we are safe to add the new scene to the set.

    (b) Some similar scenes were found - if we can reasonably argue that the new scene introduces different light transport situation(s), we can add it to the set.

    (c) Some highly similar scenes were found - we may swap the new scene with one of them. There should not be a case when we would want to swap multiple scenes for a single one because that would mean that there were redundant scenes already or the new scene covers a wider variety of light transport situations which are singled out by the found scenes.

For the cases a) and b), we may be interested in differences between the similar scenes and reasons why was a scene originally included despite their existence. Therefore, we want to have easily available information about what scenes are similar and reasoning what makes the scene special in comparison to them. This information will ease up navigation through the set and management of related scenes.

If we put it all together, our documentation of the scene should contain the following parts:

- Brief description of the tested light transport situation in the scene
- A complete and in-depth explanation of the light transport in the scene
- List of similar scenes with regards to light transport situations (if any exist)
- Discussion of what makes the scene unique compared to the other similar scenes (if any exist)

We have gone through the explained procedure of new scene addition for each scene that we present in this chapter. Documentation of all scenes contains at least the four discussed main points.

5.2 Development

To be able to compare light transport algorithms across different rendering frameworks on the same scene, we needed the scene to be as easily and well reproducible in as many renderers as possible. For this reason, we had to use scene description features that are as widespread as possible.

Nonetheless, even the simplest features often differ across the various renderers: be it their names, available parameters, default, or accepted values. And even if everything seems to be the same, the renderer may interpret them in unexpected ways. But we had to start somewhere, and so we have picked Mitsuba 0.5 and PBRT renderers as the ones to begin our endeavor to develop reproducible scenes (for reasoning why these renderers were picked see 3.4).

Our approach was to plan the scenes with as simple features as possible (limitations that we have imposed on the development ourselves) and if even then, there were differences between the resulting renderings from the Mitsuba and PBRT we tried to find out what caused them and if it was possible to adjust the scenes to resolve them (limitations imposed by the individual renderers).

In the following sections, we discuss what limitations had to be accepted (5.2.1) and present our full scene creation workflow (5.2.2).

5.2.1 Limitations

General

- *Coordinate system handedness.* Cameras in Mitsuba are right-handed, and PBRT uses a left-handed system. We have decided to follow the right-handed system (because it is also the system of the most 3D modeling tools - e.g., Autodesk 3ds Max or Blender). For this reason, we reverse the X-axis of PBRT by applying: $Scale - 111$ before the camera is defined.

- *Near and far plane distances.* Cameras in the PBRT have a hardcoded near and far plane distances, at values 0.01, and 1000 respectively[1], anything closer or further away from the camera is not visible in the rendered image. In Mitsuba, it is possible to adjust these values, but PBRT does not even document them in its scene description specification. Our scenes are in sizes of up to 500 units.

Geometry

- *No shape primitives.* We have avoided any shape primitives like Sphere, Cone, or Cube. We can not assume that they are available in other rendering frameworks (especially some more complex ones like hyperboloid in PBRT, which is not even available in Mitsuba).
- *Everything as a triangle mesh.* Triangle mesh is the most simple and widely supported representation of 3D geometry. Even if different renderers need various 3D file formats (e.g., OBJ, PLY), triangle meshes can be translated to practically all of them.
- *Explicit vertex normals.* Although vertex normals of triangle meshes can be calculated from individual triangles (Mitsuba does it properly), we have found out that PBRT was unable to do so. Specifically, while Mitsuba can take into account that one single vertex is shared by multiple triangles (smoothing), PBRT always interpreted such models as faceted (it is what happens in the figure 3.2 where same 3D models are loaded).
- *3D models already in their places.* We want models that are loaded by the renderer to be already at their intended places to avoid declarations like translation, rotation, and scaling for each of them. This should ease up the translation of the scenes between different rendering frameworks. An exception to this rule are models that are sizable and are in the scene in multiple copies.
- *Simple models.* Complex models are unnecessary as long as they don't change light transport in some intended way. Some algorithms may face difficulties depending on the geometry complexity (e.g., finding of nearby photons of the photon mapping algorithm), but from the perspective of the light transport, the situation is often the same no matter the complexity of the models. This leads to simple scenes where light transport is the main focus and not their look.

Light Sources

- *All light sources as area lights (emissive geometry).* An area light is both the most basic type of light source (supported by every renderer) and one of the more realistic ones (compared to often used point lights or spotlights). It introduces the DD substring to all paths, which allows a wider variety of light transport algorithms to evaluate some of the harder scenes - see 2.3.2.

[1]Code fragment: $PerspectiveCameraMethodDefinitions$ at http://www.pbr-book.org/3ed-2018/Camera_Models/Projective_Camera_Models.html [Online; accessed 2020-07-20]

Which is is why we have used area lights in all the presented scenes, and we can express all the possible light paths as $LDD\ (D|S)^*\ SDE$.

- *Only faceted area lights.* This tricky limitation comes from the PBRT, which is unable to use smooth models for area lights. Specifically, no matter what normals the model has if it is used as an emissive geometry, it is interpreted as if it was a faceted model. Luckily in most cases, we can get by with just squares or rectangles, and in the worst-case, a highly subdivided faceted model can be used because it will be indistinguishable from a smooth one.

Materials

- *Basic materials.* Materials are the biggest problem when it comes to the reproducibility of scenes across different renderers. Lambertian diffuse surfaces and perfect mirrors are almost the only materials matching across the rendering frameworks. Additionally, we also need dielectrics to model refractions, and conductors are the simplest solution to the modeling of glossy materials. Therefore, only these four materials are currently used: (Lambertian) diffuse, perfect mirror, dielectric (glass), and conductor (metal). Layered materials should be avoided because they do not introduce new light transport situations, which could not be modeled by the basic materials. On top of that, they also add unnecessary complexity to the reproduction of the scenes between renderers.

- *Minimized number of material types in the scene.* We can not assume that the basic materials will be reproducible across different renderers. Therefore, the fewer materials in the scene are used, the better chance it will be reproducible.

- *GGX for the microfacet distribution function.* PBRT uses GGX in its materials (can not be changed), while Mitsuba allows selecting GGX. Luckily GGX is quite popular and widespread, so it is a reasonable choice of microfacet distribution.

- *Conductor roughness.* The roughness of conductor in PBRT (metal material), can not be less than 0.01. For this reason, it is impossible to model perfectly smooth conductors, so if they are truly needed, it should be possible to use mirror material instead.

5.2.2 Workflow

When we decided to include a scene, our workflow of scene creation followed:

1. Model the scene in Autodesk 3ds Max [6] and use Corona Renderer[23] to quickly iterate over various setups of light sources and materials until the target look of the scene is achieved.

2. Export geometry of the scene and translate it to file formats supported by Mitsuba and PBRT.

3. Create a similar scene in Mitsuba (materials may slightly differ, but the rest of the scene is reproducible).
4. Create a matching scene (to Mitsuba) in PBRT. Iterate until all issues are resolved (as a consequence, Mitsuba scene may be modified if necessary) - document the unresolvable ones. In this step, difference images are used extensively.
5. Do final tweaks (e.g., small changes to now matching materials).
6. Render reference images of both versions of the scene.

5.3 Scenes

In this section, we present the created test scenes based on the methodology for the evaluation of light transport algorithms introduced in 3 and previous sections of this chapter.

Matching versions of these scenes are available for Mitsuba and PBRT renderers as a part of the evaluation framework. If needed, their development files from Autodesk 3ds Max are also provided in .max and .fbx formats. For both refer to 3.4. We also include unbiased reference images for Mitsuba and PBRT, for information about specific settings which were used for their rendering see figure 5.2.

Presentation of every scene includes (for organization of images see figure 5.1):

- Big reference image from Mitsuba renderer.
- Small (corresponding) reference image from PBRT renderer.
- Difference and relative difference images between Mitsuba and PBRT (green and red represent brighter and dimmer pixels of the Mitsuba reference compared to the PBRT reference, respectively). Difference images have exposure increased by 5 stops (equals to 32 times the brightness) to highlight the differences.
- Documentation of the scene (see 5.1).
- Additional discussion about the scene, its development, and differences between Mitsuba and PBRT versions.

For other (non-matching) scenes for Mitsuba and PBRT renderers, we recommend Bitterli's rendering resources [22]. PBRT scene repository is also quite sizable[2].

[2]Public domain, https://www.pbrt.org/scenes-v3.html [Online; accessed 2020-07-20]

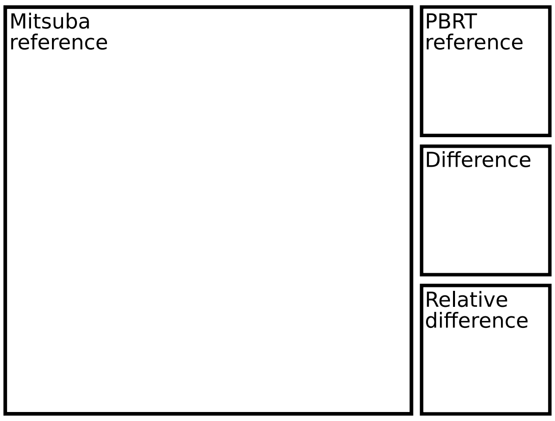


Figure 5.1: Organization of scene images.

Name	Maximum depth	Samples per pixel
cbox_classic	10	20000
cbox_indirect	10	20000
cboxm_diffuse_lsame	10	50000
cbomx_diffuse_lvar	10	50000
cboxm_glossy_lsame	20	50000
cboxm_glossy_lvar	20	50000
veach_mis	2	20000
pool_simple	10	20000
pool_classic	10	200000
ring	10	100000

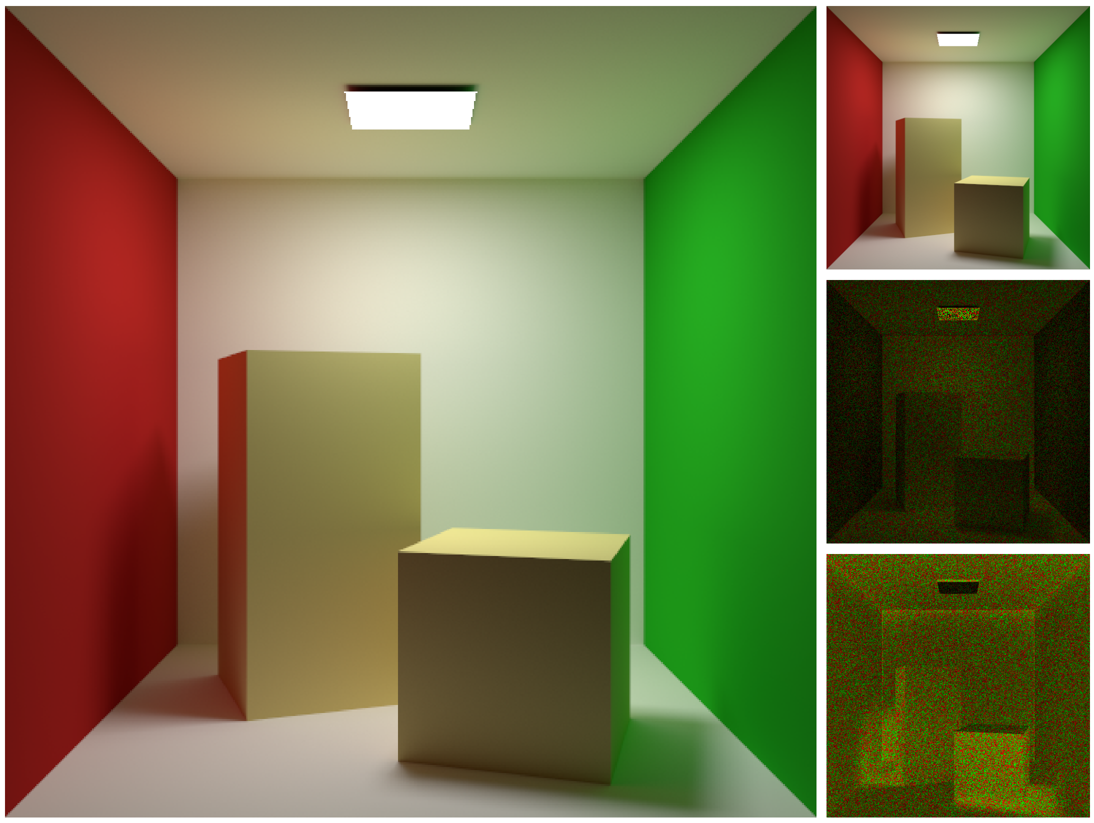
Figure 5.2: Reference images properties.

All of the reference images were rendered using bidirectional path tracing, box reconstruction filter, and random sampling. The table above describes the only settings which varied between the individual scenes.

Maximum depth specifies the longest path in the generated reference image - where the path is measured in a number of segments, not bounces (i.e., for a number of bounces on the path subtract 1).

The number of samples per pixel varied between the scenes, depending on the approximate rate of convergence. For the *pool_classic* scene, even 200 000 samples were not enough to get fully converged images (around 3 days of rendering time on a single server machine).

5.3.1 cbox_classic



Brief Description: Simple global illumination in Lambertian Cornell box.

Full Description: Recreation of the classical Cornell box[3] with Lambertian diffuse surfaces, single area light source, and simple geometry. The focus of this scene is not on some specific light transport situation and how well it is resolved (light transport in the scene is almost as simple as it gets), but whenever an algorithm works at all and produces unbiased results. If this scene is not rendered correctly, some fundamental issues need to be resolved before the use of any other evaluation scene can be considered.

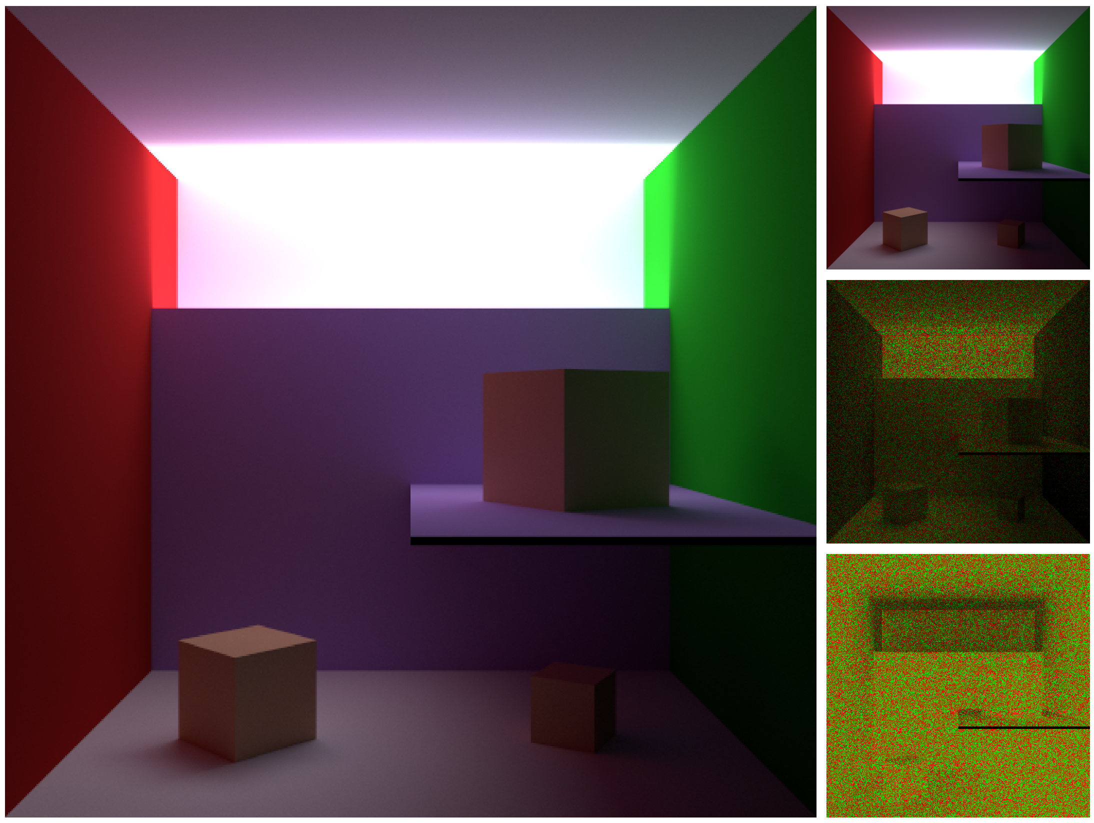
Similar Scenes: cbox_indirect

Included Because: Compared to other similar scenes, cbox_classic on purpose presents a global illumination situation that can be resolved with minimal effort.

Additional Comments: We can observe that both Mitsuba and PBRT converge to the same result, which is an indication that area lights and diffuse materials are evaluated in the same way and that any other scene using them will also be highly reproducible.

[3]http://www.graphics.cornell.edu/online/box/ [Online; accessed 2020-07-20]

5.3.2 cbox_indirect



Brief Description: Indirectly illuminated Lambertian scene.

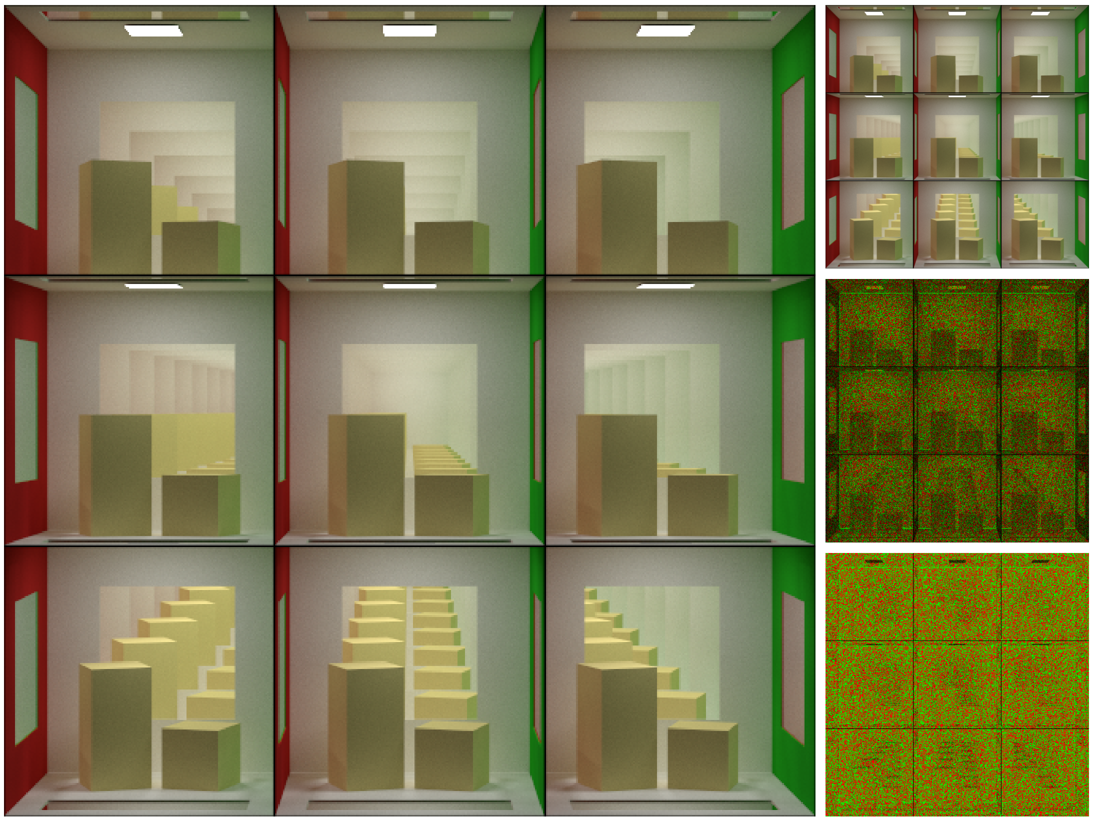
Full Description: The area light on the floor of this scene is hidden behind a wall, so most of the scene is illuminated indirectly by the light scattered from the back of the scene. Because of Lambertian surfaces, additional bounces on the light paths greatly diminish resulting contributions, therefore algorithms that can find and exploit shorter paths achieve way faster convergence.

Similar Scenes: cbox_classic, cboxm_diffuse_lsame

Included Because: This scene simulates varying degrees of indirect illumination from a single light source.

Additional Comments: Both Mitsuba and PBRT converge to the same image, except for the corner between the ceiling and back wall. This is surprising if we consider that these walls are the same geometry as in the *cbox_classic* scene. Because this difference has not leaked into the rest of the global illumination in the scene, we guess that it is connected (or at least should be reproducible) with just a direct illumination.

5.3.3 cboxm_diffuse_lsame



Brief Description: Indirect illumination by many light sources of the same intensity in the Lambertian scene.

Full Description: This scene made up of interconnected boxes showcases a global illumination situation where many light sources greatly contribute to indirect illumination. Because of Lambertian surfaces and light sources of equal intensity, the only property which influences the contribution of the light path is a number of bounces on it. This makes all the light sources almost equally important.

Similar Scenes: cbox_indirect, cboxm_diffuse_lvar, cboxm_glossy_lsame, cboxm_glossy_lvar

Included Because: Like *cbox_indirect*, this scene simulates varying degrees of indirect illumination but from many light sources where the majority of them is always occluded, therefore adding another dimension to the indirect illumination evaluation. Compared to other *cboxm* scenes, all light sources have the same intensity, and contributions of paths depend only on the number of bounces.

Additional Comments: Differences in brightness at the corners between walls of the boxes can be observed again (see *cbox_indirect*).

This scene is one of the *cboxm* scenes targeting global illumination calculations with many light sources in different conditions. These scenes share the same design:

- 9x9x9 grid of interconnected boxes. Boxes on the sides of the grid are sealed (i.e., the only way how the light can get outside of the grid is through its front side).
- Boxes are instanced, so any adjustments to the content of boxes are simple to do.
- 3x3 boxes at the front side of the grid are observed. The upper left box of the rendered image is the center box of the front side (to introduce more variance between the observed boxes, in situations where light sources have the same intensity) - figure 5.6a.
- Changes to the shape or number of objects inside of the individual boxes would not fundamentally change the properties of light transport inside of the scene, it would only change what specific paths light needs to take to get to the camera.
- Changes to the size and/or the number of holes connecting boxes could introduce some interesting light transport situations, like brightly lit boxes connected to the surrounding with small holes (similar to *cbox_indirect* but on a bigger scale with more light sources). Nonetheless, with the presented setup of connecting holes, similar cases are already happening (figure 5.6b).
- Changes to the intensity of light sources would make different paths of the same kind bring varying contributions. We explore them by *lvar* variations of *cboxm* scenes.
- Changes to the materials, influence what kinds of paths are the most contributing. We explore them by *glossy* variations of *cboxm* scenes.

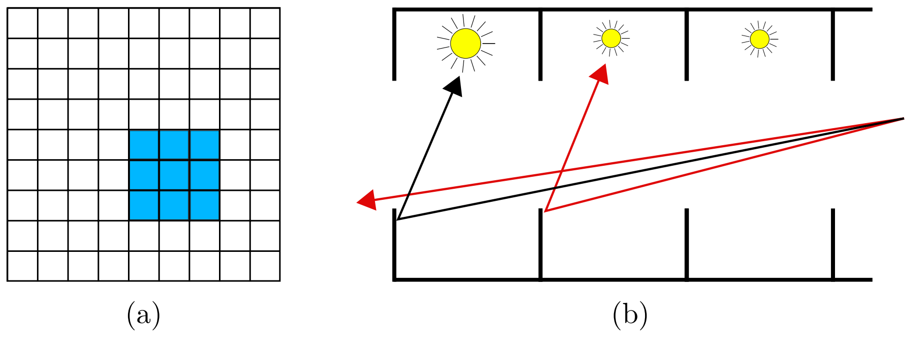
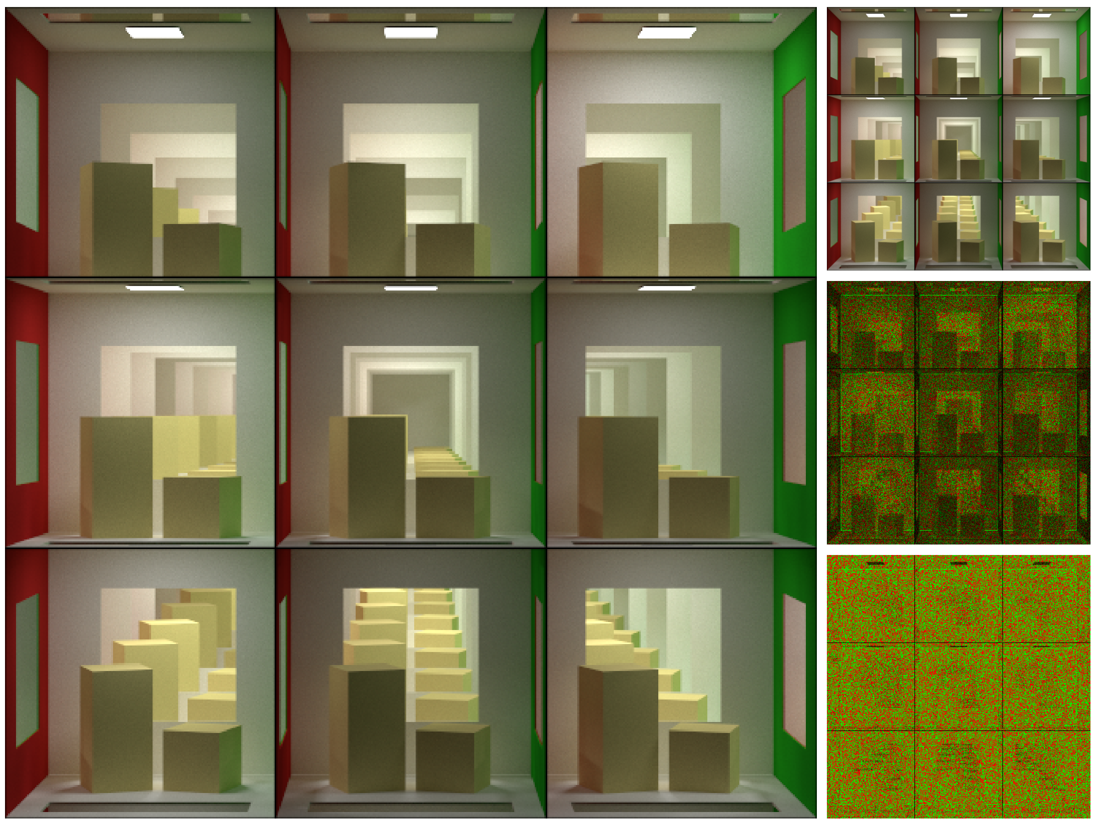


Figure 5.6: cboxm diagrams.
a) Camera focus.
b) The further the box is, the harder it is to sample direction that will lead to it and not to other boxes. If the light sources have varying intensities, some light paths will behave the same way as in the case of the brightly lit box with small holes.

5.3.4 cboxm_diffuse_lvar



Brief Description: Indirect illumination by many light sources of varying intensity in the Lambertian scene.

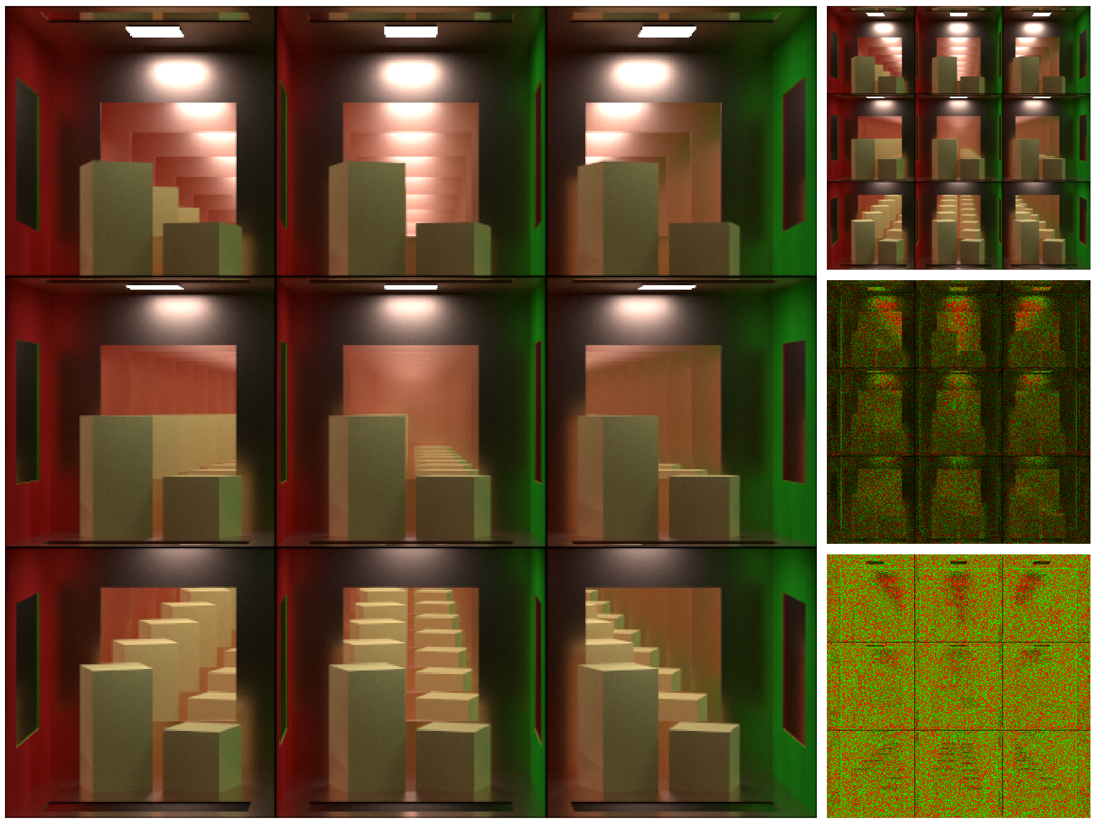
Full Description: Variation of the *cboxm_diffuse_lsame* scene with varying intensities of light sources. Contributions of paths with the same number of bounces differ, in extreme cases making even a longer path more efficient than a shorter one. Algorithms that exploit higher contribution paths and minimize time spent following low contribution paths will have an advantage in this scene.

Similar Scenes: cboxm_diffuse_lsame, cboxm_glossy_lsame, cboxm_glossy _lvar

Included Because: Introduces an additional level of complexity to the *cboxm_ diffuse_lsame* scene, by making some light sources more preferable than others.

Additional Comments : Same as *cboxm_diffuse_lsame* applies.

5.3.5 cboxm_glossy_lsame



Brief Description: Indirect illumination by many light sources of the same intensity in a scene with glossy surfaces.

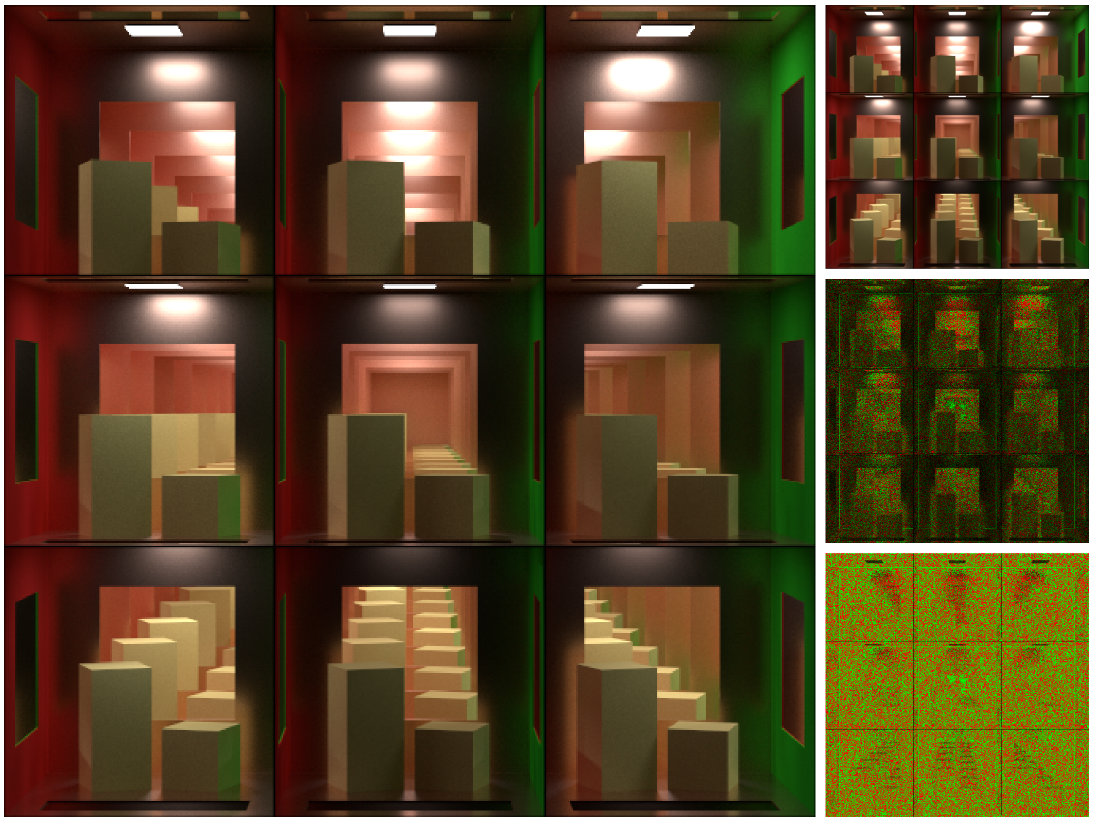
Full Description: Variation of the *cboxm_diffuse_lsame* scene with smooth glossy (conductor) materials. Glossy specular reflections reflect most of the incoming light in roughly the same direction. That makes multiple glossy bounces more efficient than a single bounce from the Lambertian surface (in terms of transmitted radiance). Therefore, it can no longer be said that shorter paths are always preferable.

Similar Scenes: cboxm_diffuse_lsame, cboxm_diffuse_lvar, cboxm_glossy_lvar

Included Because: Introduces different level of complexity than *cboxm_diffuse_lvar*, this time making a specific type of light paths more preferable.

Additional Comments: Same as *cboxm_diffuse_lsame* applies. Glossy highlights of different brightness between Mitsuba and PBRT rendering are most likely caused by differences in handling of conductor materials (the same issue was observed in the *veach_mis* scene).

5.3.6 cboxm_glossy_lvar



Brief Description: Indirect illumination by many light sources of varying intensity in a scene with glossy surfaces.

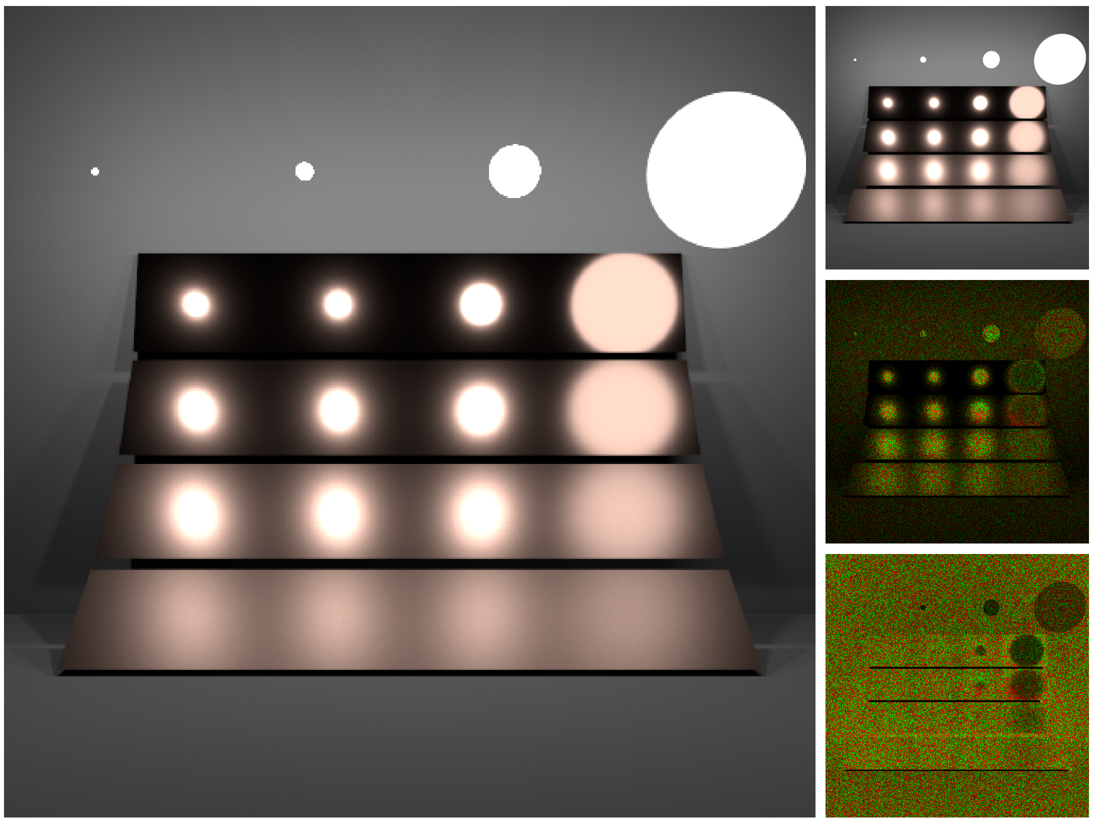
Full Description: Combination of difficulties from the *cboxm_glossy_lsame* and *cboxm_diffuse_lvar* scenes. Paths containing glossy specular reflections are in a vacuum more efficient, but depending on the light source they connect to, many of them are now almost worthless.

Similar Scenes: cboxm_diffuse_lsame, cboxm_diffuse_lvar, cboxm_glossy_lsame

Included Because: Combination of complexities from other *cboxm* scenes. They enhance each other and create a scene where the contribution of a light path is unpredictable until it is connected to a light source.

Additional Comments: For discussion about the difference images, refer to other *cboxm* scenes.

5.3.7 veach_mis

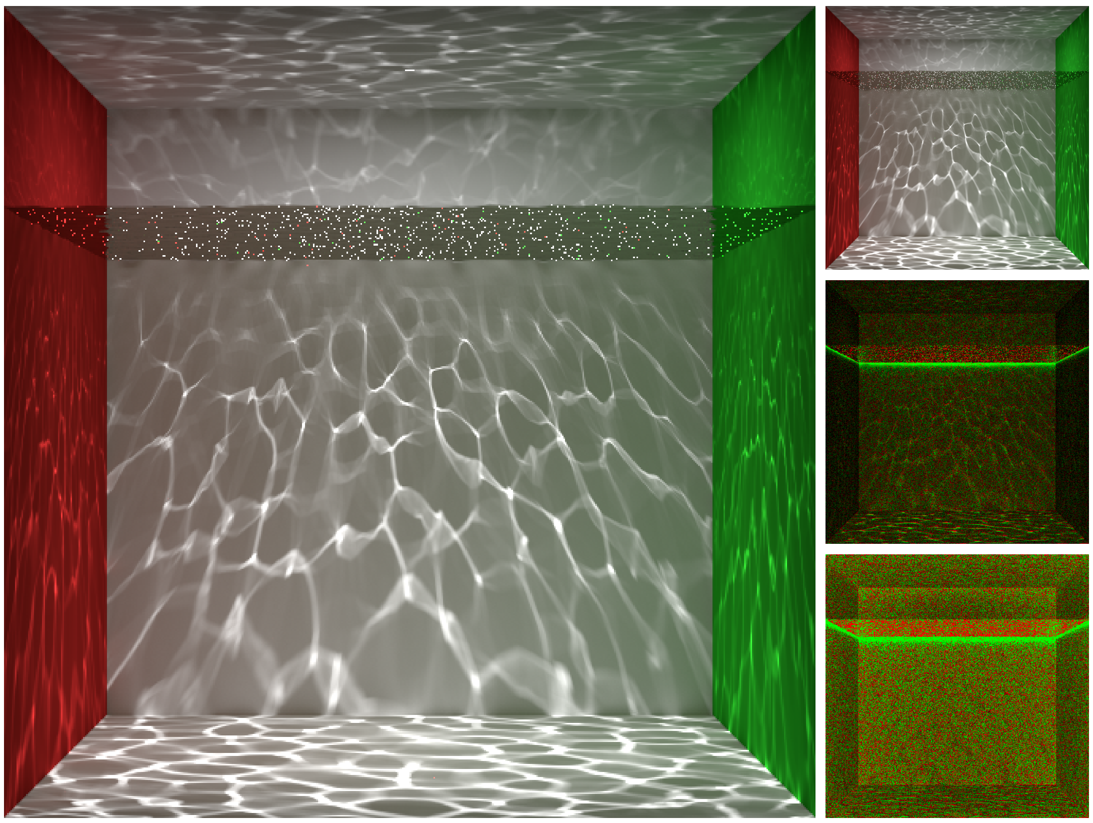


Brief Description: Multiple importance sampling of direct illumination.

Full Description: Recreation of the well known Veach's MIS scene that is often used for evaluation of the efficiency of multiple importance sampling weights for direct illumination estimation. We discussed the basics of MIS in 2.1.3 and commented on this scene at length in figure 2.7.

Additional Comments: Mitsuba and PBRT converge to a similar result, but specular highlights are not the same. The most likely cause is a difference in the evaluation of the metallic material because sampling of area lights has introduced no issues in the *cbox_classic* scene, and we can also observe the same issue in the *cboxm_glossy_lsame* scene.

5.3.8 pool_simple



Brief Description: Directly viewed SD caustics.

Full Description: This scene represents a pool viewed from the underwater. We can easily observe classical pool caustics, caused by refractions from the uneven water surface (in the scene modeled as a dielectric - glass). Small area light allows even a path tracer to render the scene, albeit rather slowly. We can express light paths of interest as $LDD\ SD\ SDE$, refer to 2.3.2.

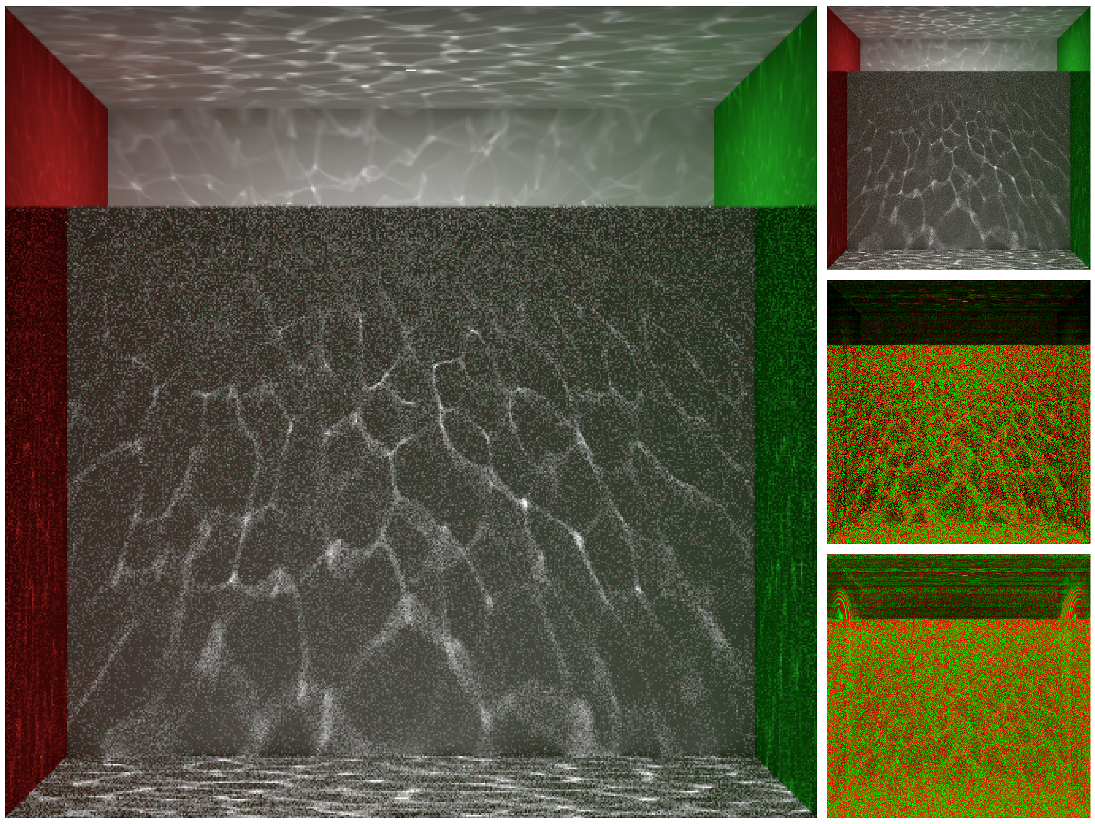
Similar Scenes: ring

Included Because: Caustics are caused by refraction on a dielectric boundary.

Additional Comments: Fireflies (extremely bright pixels which are hard to get rid of, by, for example, increasing the number of samples) can be observed on the glass boundary in both Mitsuba and PBRT. It is an interesting challenge to avoid them while maintaining the unbiasedness of many light transport algorithms.

Difference images showcase that dielectrics behave differently in Mitsuba and PBRT. These differences are not big enough to be observed when comparing reference images by eye, although they are more apparent on the difference images than in the case of conductors (see the *veach_mis* scene).

5.3.9 pool_classic

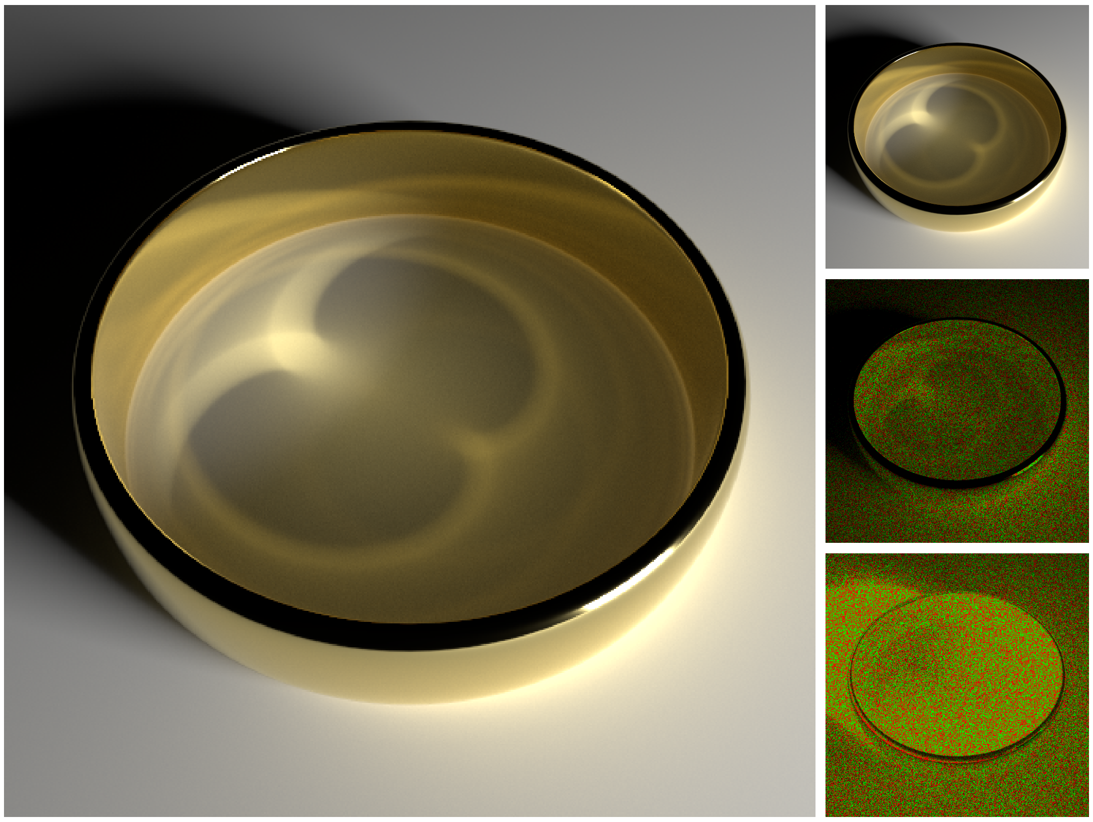


Brief Description: Indirectly viewed *SD* caustics - *SDS* subpaths.

Full Description: This scene represents a pool viewed from outside of the water. Like in the *pool_simple* scene, we can observe pool caustic but this time indirectly through a glass wall placed at the side of the water body - making the scene comparable to an aquarium. Light paths of interest can be expressed as $LDD\ SDS\ SDE$, refer to 2.3.2.

Additional Comments: Like in the *pool_simple* scene, we can observe differences caused by dielectric materials used in the scene.

5.3.10 ring



Brief Description: Directly viewed S^+D caustics.

Full Description: This scene simulates ring caustics. The curvature of the ring focuses light towards the center, forming the observed caustic. Light reflected towards the other (shadowed) side of the ring is focused similarly, creating additional caustic in the shadow of the ring. Light paths of interest are $LDDS^+DSDE$, refer to 2.3.2.

Similar Scenes: pool_simple

Included Because: Caustics are caused by reflections from a highly smooth conductor.

Additional Comments: Both Mitsuba and PBRT converge to the same result (baring the negligible difference on the circumference of the ring in the relative difference image).

Conclusion

In this thesis, we focused on the problems of comparison and evaluation of physically-based renderers, in particular of their light transport algorithms. Our goals were threefold. To define an approach that, without exceptions, can be used for evaluation of light transport algorithms across different rendering frameworks. Based on the approach, develop a curated test dataset for the evaluation (in our case scenes). And finally, provide an automated framework that will facilitate the evaluation. The goal of the thesis was not to evaluate different light transport algorithms but to provide a means for it.

We have determined that the only objective approach to the comparison of two algorithms that can be used across different rendering frameworks is their correctness. As the most fundamental property of physically-based light transport algorithms, it is indeed worthy of evaluation, but we had to address the incompatibilities of different renderers and scene description formats they use. Luckily there is only one correct solution to light transport in any scene, so if we can create scenes in matching variations across different rendering frameworks, we can also directly compare their renderings.

Therefore, we have developed a curated set of test scenes for the evaluation of different properties of light transport algorithms. This set presents scenes with hard global illumination situations like indirect illumination, many light sources, or caustics. We provide this set of scenes in formats of the two most popular research-oriented renderers Mitsuba and PBRT, together with their corresponding reference images. Renderings of our scenes in these two renderers are comparable, but as we have observed, even these two popular and well-debugged renderers give different results. Why is that so, and whichever one of these two renderers is wrong, are questions for their developers, but these are also exactly the kind of questions that we have wanted to introduce.

Finally, to facilitate the evaluation of light transport algorithms with these scenes, we have developed an evaluation framework that allows us to define various test cases (light transport algorithm, its properties, and even other rendering settings), which are then used to render selected scenes. It can also generate a simple website providing an immediate comparison of the rendered images. With our framework, different renderers can be handled in one place, and furthermore, it can be easily extended to support additional renderers or scenes.

Future Work

The result of our work is a self-contained evaluation framework with its test dataset. Therefore potential future work is focused on its improvements, extensions, and practical applications, some concrete examples are:

- Support of additional renderers.
  - LuxCoreRender and Cycles are the next obvious choices.
  - Proprietary renderers are not usually physically-correct, but by supporting some of them, we would bring the research and practical side of the rendering closer together.
- Automated import of new scenes.
  - Scenes of some formats would be possible to import to the evaluation framework automatically.
  - This has its limitations because we need to be able to handle arbitrary content of scene files without knowledge of its purpose (e.g., never seen before setting).
- Website working without a webserver.
- Additional test scenes
  - Scenes with participating media. The focus of the presented set of scenes is non-volumetric light transport. Although light transport inside of participating media is not fundamentally different, there are many algorithms specifically made to resolve it.
  - Other additions. Scenes may be added or replaced over time, thus we expect that future development will bring many changes to the scene set.
- Higher quality reference images. Provided reference images are mostly good enough, but some of the more difficult scenes would benefit from longer rendering time.
- Use of current results. Reasons for differences between renderings of Mitsuba and PBRT could be tracked down, and corresponding bugs of these renderers could be fixed.

Bibliography

[1] Paul Rademacher, Jed Lengyel, Edward Cutrell, and Turner Whitted. Measuring the perception of visual realism in images. *12th Eurographics Workshop on Rendering*, 2001.

[2] Wenzel Jakob. Mitsuba renderer, July 2010. http://www.mitsuba-renderer.org.

[3] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: from theory to implementation.* Morgan Kaufmann, third edition edition, 2017.

[4] D H Sliney. What is light? the visible spectrum and beyond. *Eye*, 30(2):222–229, January 2016.

[5] Gaoyang Ye. *Thermodynamic and structural investigations on the interactions between actinides and phosphonate-based ligands.* Theses, Université Paris-Saclay, September 2018.

[6] 3ds Max | 3D Modeling, Animation & Rendering Software | Autodesk. https://www.autodesk.com/products/3ds-max/overview [Online; accessed: 2020-07-20].

[7] Blender Foundation. blender.org - Home of the Blender project - Free and Open 3D Creation Software. https://www.blender.org/ [Online; accessed: 2020-07-20].

[8] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford University, 1997.

[9] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150. ACM, 1986.

[10] Ivo Kondapaneni, Petr Vévoda, Pascal Grittmann, Tomáš Skřivan, Philipp Slusallek, and Jaroslav Křivánek. Optimal multiple importance sampling. *ACM Trans. Graph.*, 38(4), 2019.

[11] Pascal Grittmann, Iliyan Georgiev, Philipp Slusallek, and Jaroslav Křivánek. Variance-aware multiple importance sampling. *ACM Trans. Graph. (SIGGRAPH Asia 2019)*, 38(6), 2019.

[12] Robert Cook and Kenneth Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1:7–24, 1982.

[13] Petr. Beckmann and Andre. Spizzichino. *The scattering of electromagnetic waves from rough surfaces, by Petr Beckmann and Andre Spizzichino.* Pergamon Press, 1963.

[14] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, pages 195–206. Eurographics Association, 2007.

[15] S. T. Trowbridge and P. K. Reitz. Average irregularity representation of a rough surface for ray reflection. *Journal of The Optical Society of America*, 1975.

[16] Paul Heckbert. Adaptive radiosity textures for bidirectional ray tracing. volume 24, pages 145–154, 1990.

[17] Heiner Otterstedt. Caustics on a water glass, 2006. http://www.otterstedt.de/wiki/index.php/Bild:Kaustik.jpg [Online; accessed 2020-07-20], GNU FDL - image was modified.

[18] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, 1993.

[19] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*. ACM Press, 1997.

[20] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., 2001.

[21] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), December 2019.

[22] Benedikt Bitterli. Rendering resources, 2016. https://benedikt-bitterli.me/resources/.

[23] Corona Renderer. https://corona-renderer.com [Online; accessed: 2020-07-20].

List of Figures

Attachment 1
Electronic attachment contents

The contents of the accompanying electronic attachment are:

- Evaluation framework, including the evaluation scenes, see 4.4 for more details.
- README.txt - Basic information about the content of the electronic attachment and URL of the public GitHub repository with an up-to-date version of the framework.
- A Methodical Approach to the Evaluation of Light Transport Computations.pdf - PDF version of this document.