Masterarbeit

A light-weight visual debugger for Monte Carlo-based light transport simulation

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Computergrafik
Christoph Kreisl, 2019

Bearbeitungszeitraum: von 01.10.2018 bis 31.03.2019

Abstract

In this work we present a novel light-weight, interactive visualization framework for debugging and analyzing complex physical based rendering systems based on Monte Carlo integration. The framework allows debugging on the fly by directly interacting with the rendered image to analyze occurring artifacts. The memory consumption is minimal as the data is generated on demand. For this purpose, only minimal code changes within the rendering system are required to allow deterministically rendering and visualization. We provide multiple views whose rendering data is linked to each other, visualizing arbitrary render and user data. A three-dimensional view visualization allows to explorer the rendered scene as well as the traced light paths. The framework enhances tasks in development, education, and analysis by enabling the user to interact with various visual representation which are all together linked by the brushing and linking paradigm. We present several use cases on how the framework can be used to identify and analyze various rendering artifacts and occurring path tracing problems.

Acknowledgments

I would like to thank Sebastian Herholz for his huge support and effort he spent on feedback and instructions. Many thanks also to Tobias Rittig for his outstanding help and feedback creating this work.

Contents

1. Introduction

The creation of photo-realistic images from a generated three-dimensional scene is called rendering. This method is used in various areas such as animated films, special effects, product design, architecture and the game industry. Those rendering algorithms are constantly evolving to create more and more realistic images and adapt lighting conditions and object properties to the real world. Computing light within virtual scenes is mostly based on ray tracing which traces rays from surface to surface and accumulate the emitted light along the way. In the recent years path tracing has become the method of choice in the movie industry [FHP+18]. For that path tracing algorithms are becoming more and more complex to handle various lighting conditions and object properties, like Photon mapping [Jen96] or Metropolis light transport [VG97]. These algorithms operate on a large amount of data and need to be physically correct, which has added difficulties in debugging and dynamic analysis of the algorithms in the ray tracing domain. The complexity of the algorithms makes it hard to identify errors in the implementation or sources of problematic light transport, which is hard to solve by the used algorithm. Analyzing or debugging such complex render systems and their path tracing algorithms is quite difficult, since for faster computing most of the calculation runs in parallel. Moreover, to achieve a photo-realistic image billions of rays has to be traced through the scene which takes a very long time. All of this makes it difficult to analyze rendering systems from scratch. Most simple debugging tools, which are provided by the integrated development environment (IDE), have no visualization tools operating on such kind of data structures to provide a deeper look insight the rendering process and what is happening during path tracing.

Visualization and interactive feedback in rendering systems gets more and more important for rendering engineers or visual artists. Most methods work on the visualization of the distribution of light within the scene [RKRD12] to help research scientists to develop, optimize or modify [SNM+13] new approaches e.g. path guiding [MGN17]. However, they are often limited to very specific scenarios or rendering systems. The challenges involve solving several different tasks, such as providing a deterministic process in order to reproduce and allow debugging of traced rays through the scene. Moreover, the problem of recording all data during the rendering process lead to a huge amount of data and memory consumption. Finally, in most cases there is no possibility adding arbitrary custom data to provide debug information which give a deeper look within the rendering process.

In this work we introduce a new interactive, light-weight visualization debugger

framework for Monte Carlo based light transport simulation, allowing interactively analyzing render artifacts directly on the rendered image. The framework supports an interactive visualization based on the brushing and linking paradigm [Sii00] and dynamic analysis of ray-based rendering algorithms in the computer graphic domain. A modular architecture design can be adapted to any rendering system based on Monte Carlo integration. Instead of working on large amounts of stored data, which is collected during the rendering process, the render data is interactively requested, computed and visualized on the fly. In order to allow debugging of complex rendering systems, we make extensive use of the Seedmap concept. Using the Seedmap allows deterministic rendering. Furthermore, it provides a fast and interactive access to specific data with low memory usage. We present an interface around the main rendering process which separates the tasks within a rendering server system and a interactive visualization client system. Besides, the interface provides the functionality to add arbitrary custom data during the rendering process, thus helps developers and students to gain deeper insight in path tracing algorithms. For this purpose we provide different views for visualization and dynamic analysis of the scene and traced paths. Finally, we present various use cases, showing how the framework can be used to debug path tracing artifacts and problems or to gain additional insights into the light transport within a rendered scene.

2. Related Work

In the following we present some of the prior works which are most important to our approach.

Monte Carlo light transport. The light transport in virtual scenes is described by the rendering equation [Kaj86]. It is based on geometric optics. In order to solve the rendering equation one can use a variety of algorithms including path tracing [Kaj86], light tracing [Arv86] or Metropolis light transport [VG97]. These approaches show how difficult and computation intensive solving the rendering equation can be, especially if the results should be physically correct.

Light Path Visualization and Modification. Only a couple of frameworks and tools deal with the visualization of computer graphics algorithms, regarding to light transport algorithms. [RKRD12] introduces a visualization technique for inspecting the global light transport in virtual scenes. The rendering is based on Photon mapping [Jen96] providing interactively visual feedback, helping users to gain a deeper insight in how light interacts and travels within the scene. They implemented the light inspection tool as a plug-in in the 3D modelling and rendering software Maya 2012 from Autodesk, making it difficult to use it for other systems.

Another system by [SNM+13] introduces a visualization tool with the possibility of light manipulation and modification on static and animated scenes. They focus on helping artists to achieve a specific lightning setting by modifying traced light paths. The light modification approach addresses artists and light designers, giving scientists no opportunity to add path or debug information for analysis.

Ray Tracing Visualization. The Framework for Visual Dynamic Analysis of Ray Tracing Algorithms by [HL14] introduces a framework for visualizing ray tracing algorithms. They use a data logging API which collects all render information during the rendering process. Therefor, the data can be stored in an online cloud storage or log file enabling online and offline editing. Information about the traced rays in the scene are visualized within a graphical user interface (GUI). The GUI visualizes the scene geometry in combination with traced paths. Additionally, they provide a filter mechanism to filter paths depending on their attributes. The framework helps developers to verify new implemented rendering approaches by inspecting

the 3D path space. Additionally, it helps students to gain a deeper understanding of how path tracing algorithms are operating. Apart from the data produced for visualization, there is no other path or debug information which the user can add for analyzing or debugging the ray tracing algorithm.

Gribble et al. [GFE+12] introduces a similar approach of ray tracing visualization within a GUI. For that purpose every ray state is recorded, saved and later displayed. The GUI gives control of the entire visualization process. The toolkit provides interactive ray tracing and bounding volume hierarchy visualization of the geometry in the scene. Similar to [HL14] and [GFE+12] our framework uses a GUI as an interactive connection between the user and the rendering system.

Visualization with the Brushing and Linking paradigm. An interactive ray tracing visualization approach with several views, providing various analyzing tools is introduced by [SAH+16]. They use the brushing and linking paradigm [Sii00] to allow an interactive visualization between their views. Therefor, multiple views with different visualization aspects, such as a heat-map, a three-dimensional scene and a parallel coordinate plot [Ins09] are provided. Similarly, our framework provides multiple views for various analyzing of the traced paths. But rather than analyzing all pixels and their corresponding path information at once, which would exceed over billions of rays, we only analyze one specific pixel and its traced paths at a time.

3. Background

This chapter gives an introduction to the most important theory topics needed for further comprehension, especially for readers without a computer graphics background.

3.1. Monte Carlo Integration

Monte Carlo (MC) is a mathematical technique for numerical integration. It describes a method for computing integrals using probabilistic techniques. As an example, think of solving an one-dimensional integral of the form

$$I = \int_{I^S} f(x)dx, \tag{3.1}$$

where I^S represents the integration space. Solving this integral analytically requires the antiderivative, meaning $f(x)$ must be known. Even if $f(x)$ is given does not mean that the analytic integration of $f(x)$ is known. A MC estimator instead only requires the function $f(x)$ to be evaluable at arbitrary points, making it easy to implement and applicable to many problems. It is defined as:

$$I_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}, \tag{3.2}$$

with x_i being a random variable in the domain. N stands for the amount of samples which are drawn from the function $f(x_i)$. The denominator $p(x_i)$ describes a probability density function (PDF) from which each sample $x_i \in \mathcal{S}$ is drawn. The PDF must not be negative, furthermore it must satisfy the conditions $p(x_i) > 0$ where $f(x_i) \neq 0$ and $\int_{x\in\mathcal{S}} p(x_i)dx = 1$.

The result is an approximation of the correct value with respective error bars, and the correct value is likely to be within those error bars. This error describes the variance of an estimator. Depending on the variance one can say how efficient an estimator is. The variance of a MC estimator is given by

$$Var(X) = \sigma^2\left(\frac{f}{p}\right) = \int_{I^s} \frac{f^2(x)}{p(x)}dx - \left(\int_{I^s} f(x)dx\right)^2. \tag{3.3}$$

The variance of an estimator $E(x)$ decreases linearly to the amount of drawn samples N, which is given by

$$\begin{aligned} var(E(x)) &\sim 1/N \\ \sigma &\sim 1/\sqrt{N}. \end{aligned} \tag{3.4}$$

By taking infinitely many samples, the error between the estimate and expected value is statistically zero. This is given by the *law of large numbers* which ensures that

$$prob\left(E(x) = \lim_{N\to\infty} E[I_N]\right) = 1, \tag{3.5}$$

and therefore the expected value of the estimator converges to the expected value of the integral I.

In comparison to other integration methods like *Trapezoidal* or *Gaussian quadrature*, MC integration is the most effective method for high-dimensional integrals. Usually, by increasing the functions dimension much more samples are required for convergence to the expected value. The amount of samples needed to sample each dimension equally can be described with N^d, where d represents the dimension. This phenomena is called the *Curse of dimensionality*. MC integration, however is independent of the integrand's number of dimension. This makes it effective for higher-dimensional integrals. Another aspect is the converge rate of a MC estimator, which decreases with $O(n^{-1/2})$. This points out that four times the sample count are needed to cute the variance of the estimator in halve no matter the dimensionality of the problem.

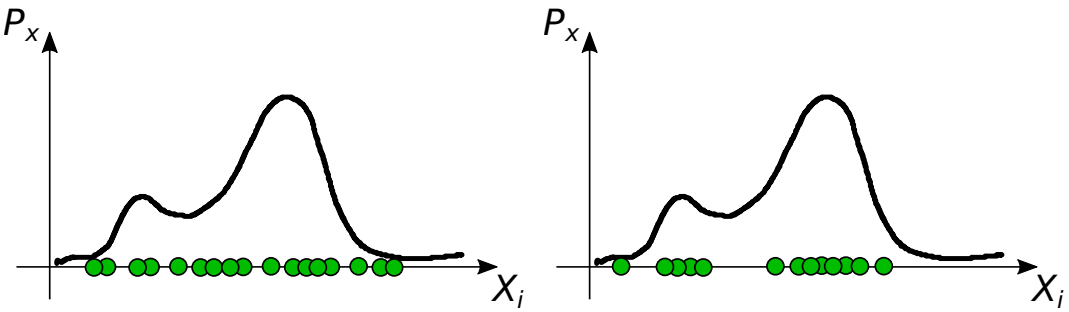
Applying MC to higher-dimensional integrals is straightforward. Only the random variable has to match the dimensionality:

$$\begin{aligned} &\text{Dimension: 1,} \quad I_N = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \\ &\text{Dimension: 2,} \quad I_N = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i, y_i)}{p(x_i, y_i)} \\ &\qquad\qquad \cdots \\ &\text{Dimension: } n, \quad I_N = \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i, y_i, \cdots, n_i)}{p(x_i, y_i, \cdots, n_i)}. \end{aligned} \tag{3.6}$$

3.2. Sampling

The crucial part of MC Integration is generating random samples x_i and evaluate a function $f(x)$ and their PDF in the function's domain. Ideally, one would prefer the samples to be generated according to a given PDF p. This is useful, since MC integration only provides estimates and for a faster convergence rate each estimate should contain as little variance as possible.

3.2.1. Uniform Sampling



(a) Uniform Sampling (b) Importance Sampling

Figure 3.1.: Comparison between the uniform and importance sampling strategy. Both strategies are applied to a function, showing the main differences.

The naive way of sampling an unknown function $f(x)$ is to sample it uniformly as seen in Fig. 3.1a. All samples are evenly distributed over the function. The PDF of uniform sampling is given by $p(x) = 1/I_S$. The problem arising, is that parts of the function are sampled independently of their value. This is a problem regarding the converge rate which in this case is comparably low. Therefore, more samples are needed to reach the expected value. Instead of drawing samples uniformly distributed over a function, we want more samples placed where the functions value is greater and contributes relatively more to the overall estimated integral, which is depicted in Fig. 3.1b.

3.2.2. Importance Sampling

The main goal of importance sampling is to reduce the variance of the MC estimator by drawing samples according to a non-uniform distribution $p(x)$. In order to obtain a good estimate we want f/p to have a low variance as possible. We draw samples $X \sim p(x)$ according to a PDF p. The optimal case to draw samples would be

$$\frac{f(x)}{p(x)} = I, \tag{3.7}$$

which would lead to

$$Var(X) = \sigma^2\left(\frac{f}{p}\right) = 0, \tag{3.8}$$

consequently *zero variance* in total. This shows that the variance depends on the choice of $p(x)$. Obtaining a perfect result with zero variance, the shape of the PDF function has to match the shape of $f(x)$ perfectly. Given that this shape can be rather complex (e.g. $f(x)$ can be a composition or product of several complex functions) or even unknown in most cases. It is not always possible to derive the perfect PDF for a function and therefore achieve zero-variance. In the case that one can sample proportional to multiple components of the function one can use Multiple Importance Sampling (MIS) and the balance heuristic [VG97] to mix both sampling strategies.

Generating samples according to a given PDF p one uses the inversion method, which converts a given random number into a sample. This is done by using the cumulative distribution function (CDF). The CDF of a continuous random variable X can be expressed as integral of its PDF and is defined as

$$P_X(x) = P(X \leq x) = \int_{-\infty}^{x} p_X(t)dt \tag{3.9}$$

and for discrete cases

$$P_X(x) = P(X \leq x) = \sum_{x_i \leq x} P(X = x_i). \tag{3.10}$$

For a uniform random variable the CDF is given by $P_U(x) = \int_{-\infty}^{x} U(t)dt = x$.

To compute the PDF to an according uniform random variable set both CDFs equal and solve for the variable of the PDF. This will give a formula depending on a uniform random variable which gives a distribution according to the density of p.

3.3. Random Number Generators

Since MC integration is based on drawing random generated samples to evaluate a function f, this section gives a short overview of RNGs types. In general a random number generator (RNG) generates a sequence of numbers that can not be reasonably

predicted. In the domain of random number generators a distinction is made between two generators types, true hardware random-number generator (HRNG) and pseudo-random number generator (PRNG). Our goal is to visualize and debug the behaviour of MC estimators. Therefore it is important, that the estimator behaves deterministic and its results are reproducible.

3.3.1. Hardware Random-Number Generators

HRNGs or true random number generator (TRNG) generate genuinely random numbers. In order to generate true random numbers a hardware device is used which generates random numbers based on a physical process [Tka03]. Such devices are often based on microscopic phenomena that generates random noise signals. These signals are in theory not predictable. As a consequence of not being predictable the main characteristic is a lack of repeatability. This method of generating random variables is used in the field of data encryption, for example to create random cryptographic keys to encrypt data. Therefore, since numbers are not predictable or reproducible, HRNGs can make testing algorithms or debugging code difficult, or nearly impossible.

3.3.2. Pseudo-Random Number Generators

A PRNG in comparison to HRNG generates numbers which appear random, but are actually deterministic. It is also known as a deterministic random bit generator (DRBG). The main difference to HRNG is that the generated random number sequence of a PRNG is produced by an algorithm and can be reproduced if the state of the PRNG is known. The sequence of random numbers is completely determined by an initial value, which is called the PRNGs seed. Using the same seed as instantiation will always produce the same sequence of random numbers.

Based on this concept many different algorithms like the Halton Sequence [Hal64], Blue Noise [APC+16] or the Golden Ratio Sequences [SKD12] are introduced to generate random numbers based on a low-discrepancy pattern. Low-Discrepancy sequences are mostly called quasi-random or sub-random sequence, due to their common use as a replacement of uniformly distributed random numbers.

The main characteristic of being reproducible or deterministic by a known initial seed makes it perfect for MC integration and the purpose of debugging.

3.4. Rendering Equation

Light transport is described by the rendering equation [Kaj86]:

$$L_o(x,\vec{\omega}_o) = L_e(x,\vec{\omega}_o) + \int_\Omega f_r(\vec{\omega}_i,x,\vec{\omega}_o)\, L_i(x,\vec{\omega}_i)\, |\cos\theta_i|\ d\vec{\omega}_i, \tag{3.11}$$

where $L_o(x,\vec{\omega}_o)$ describes the spectral radiance leaving the particular position x along the direction $\vec{\omega}_o$. $L_e(x,\vec{\omega}_o)$ delineates the self-emission e.g. of an object and the integral represents the reflected radiance over the upper hemisphere. The factors within the integral are the bidirectional reflectance distribution function (BRDF) $f_r(\vec{\omega}_i,x,\vec{\omega}_o)$ which describes the proportion of light reflected from $\vec{\omega}_i$ to $\vec{\omega}_o$ at the position x. The BRDF tries to represent the surface properties of objects in the world. $L_i(x,\vec{\omega}_i)$ represents the incident radiance at the position x. The cosine $\cos(\theta_i)$ describes the angle between the surface normal $\vec{n}$ and the incoming direction $\vec{\omega}_i$ which reduces the light due to solid angle theory.

Solving the rendering equation for any given scene is the main challenge in realistic rendering. The solution is not trivial since evaluating the incident radiance $L_i(x,\vec{\omega}_i)$ can be itself a reflected radiance quantity $L_o(x,\vec{\omega}_o)$ from another point in the scene. This duality makes the evaluation of L_o a recursive high dimensional problem, which in theory could be of infinite dimension.

3.5. Path Tracing

Path tracing is an attempt to solve the *Rendering Equation* 3.11. Theoretically, light is transmitted and traced from the light sources within the scene to the camera. The path tracing approach instead works vise versa by shooting rays from the camera through the image plane (pixel) into the scene. [Kaj86] proposed a solution by using MC integration to approximate the Rendering Equation. Applying MC integration to the Rendering Equation results in

$$L_o(x,\vec{\omega}_o) \approx L_e(x,\vec{\omega}_o) + \frac{1}{N}\sum_{i=1}^{N}\frac{1}{q_j}\cdot\frac{f_r(\vec{\omega}_j,x,\vec{\omega}_o)\, L_i(x,\vec{\omega}_j)\, |\cos\theta_j|}{p(\vec{\omega}_j)}. \tag{3.12}$$

The first ray is shot from the camera through a pixel in the image plane. If the ray hits a surface point x the integral over the hemisphere is solved stochastically by sampling a direction with respect to solid angle that has PDF $p(\vec{\omega}_j)$. At the intersection point the material property is evaluated by the BRDF f_r computing how much light is reflected into the particular viewing direction. $1/q_j$ represents the path survival probability by *Russian Roulette* 3.5. After a new direction is sampled the survival probability decides, if the path is terminated or not. If no termination is performed, the new direction is again traced and followed within the scene. Repeating this forms a path through the scene which can be seen in Fig. 3.2.

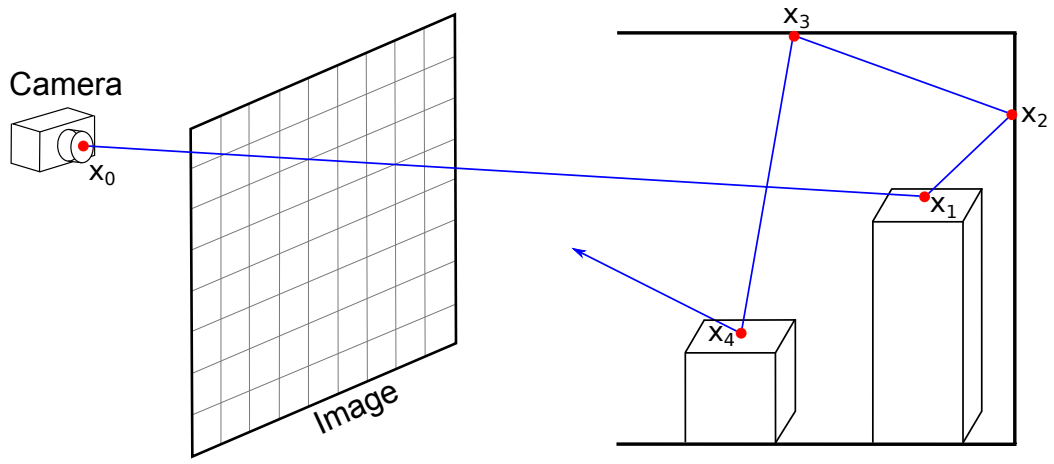


Figure 3.2.: Path tracing process showing how a pixel is computed by tracing paths through a three-dimensional scene. Rays start from the camera they are shot through a pixel in the image plane into the scene to compute the pixel's intensity.

The path starts from the vertex x_0 at the cameras position. At each intersection, the BRDF is sampled to generate a new direction, therefore the next intersection vertex x_{i+1} is found by tracing a ray from x_i to the sampled direction and finding the closest intersection. This results in a path with n bounces, with a light contribution of

$$P(n) = \sum_{N=1}^{M} L_e(x_{N+1} \rightarrow x_N) \cdot \prod_{n=1}^{M} \frac{1}{q_{n_i}} \cdot \frac{f_r(x_n, \vec{\omega}_{n_i}, \vec{\omega}_{n_o}) |\cos \theta_{n_i}|}{p(\vec{\omega}_{n_i})}. \tag{3.13}$$

$n \in [1, +\infty]$ describes the number of bounces and M represents the maximum value of bounces for a path. The recursively called L_i is solved by tracing the path within the scene and evaluating the corresponding BRDF at each intersection point. If a path hits a point that is emitting light e.g. a light source, the radiance is carried through the path back to the start point by multiplying with the path weight i.e. the product of all f_r's along the path. If no surface is hit by the ray, then $L_i(x, \vec{\omega}_j)$ may return the value of an environment map or otherwise zero. This results in a path tracing algorithm based on the rendering equation approximated by MC integration.

Russian Roulette. Russian Roulette is a method to increase the efficiency of the estimator F by terminating potentially low contribution paths with a higher probability. Based on this Russian Roulette addresses the problem of evaluating expensive samples with low or even zero contributions to the final result. Simply said, Russian Roulette skips paths with low contribution by replacing them with new estimators F_j depending on a survival probability q_j. With the probability of

$(1 - q_j)$, the integrand is still evaluated, otherwise the path is terminated. This forms a new estimator of the form

$$F'_j = \begin{cases} \frac{1}{q_j} F_j & \text{with probability } q_j \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

The probability q_j is chosen for each estimate F_j separately, based on some convenient estimate of its contribution e.g. the throughput which represents the contribution along a path. Comparing the expected value of the resulting estimator lead to the same expected value of the original estimator:

$$\begin{aligned} E[F'] &= q_j \cdot \frac{1}{q_j} E[F_j] + (1 - q_j) \cdot 0 \\ &= E[F_j] \end{aligned} \tag{3.15}$$

Russian Roulette does not reduce variance. Rather a problem can be poorly chosen Russian Roulette weights, which can even increase variance. Think of applying Russian Roulette to all traced rays within the scene with a survival probability of 0.01, meaning only 1% of the generated paths are further traced. Each path is weighted by $1/0.01 = 100$. Leading to a final rendered result which would be still 'correct' in strictly mathematical sense, although visually the rendered image would contain mostly black pixels with a few very bright ones with high contribution caused by the Russian Roulette probability.

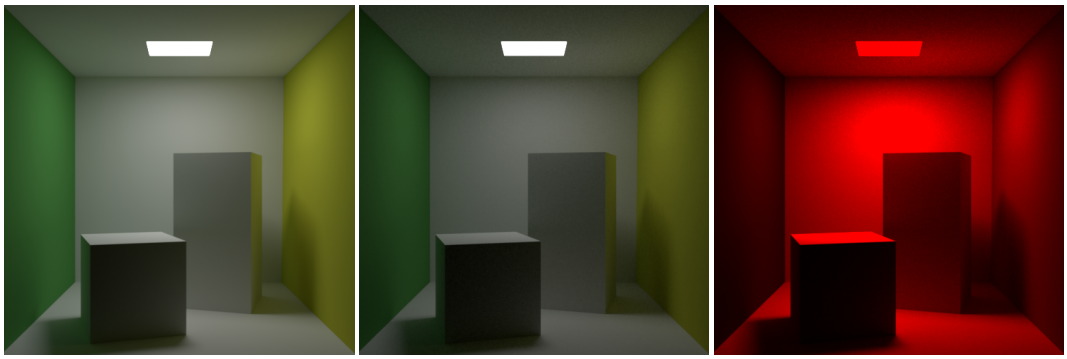
3.6. Potential Problems Of Path Tracing Algorithms

Solving the Rendering Equation (Eq. 3.11) with the MC path tracing approach (Sec. 3.5) by sampling a path through the scene can lead to various problems, like the mentioned Russian Roulette problem which can appear by a bad chosen termination probability q_j. In the following we present more problems which one has to be aware of with path tracing algorithms.

3.6.1. Implementation Artifacts

Most common artifacts or appearing problems can be traced back to wrong implementations, what may be due to a wrong implemented physical expressions of different properties or other internal programming errors. However, such implementation artifacts can be based on a wide range of different tasks in path tracing algorithms e.g. occurring errors within the BRDF, leading to a false contribution, or

occurring errors due to bad sampling strategies e.g. a bad path direction decisions, leading to a slow or bad convergence rate. A well known problem is if an estimator F converges incorrectly, resulting in an wrong expected result. Such errors lead in most cases to too bright or too dark rendered images as shown in Fig. 3.3. The result of the estimator F_n (middle) has less contribution compared to the ground truth image (left), which is shown by the resulting red difference image (right). Another aspect can be problems with the framework or the rendering system handling numerical problems such as computing correct geometry intersections. These calculations can lead to numerical instabilities. Numerical instabilities describe occurring problems of the floating value representation within a computer and the related problems of floating-point arithmetic. To summarize, solving such problems are not trivial, since path tracing refers to a multidimensional problem addressing more than one task.



(a) Ground Truth (b) Result of an estimator F_n (c) Compared Image (Exposure +2)

Figure 3.3.: Comparison of an estimator F_n (middle) with the ground truth data (left), showing that the estimator F_n does not converge correctly, which can be seen by the red color of the compared image (right).

3.6.2. High Variance Samples (Fireflies)

High variance samples of an estimator can be seen as bright pixel noise within an rendered image. Fig. 3.4 shows a caustic from a glass sphere, where the caustic can be seen as bright spot in the middle of the image. The caustic is surrounded by bright pixels called fireflies. They tend to be confined to particular parts of the image, where they are caused by interactions between particular material and lighting settings that only affect certain objects in the scene. The origin of such artifacts can be traced back to Importance Sampling (Sec. 3.2.2). These rendering artifacts are caused from a miss match between the PDF p and the actual contribution from the sampled direction, meaning that directions and therefore regions with high contributions are not sampled properly. The resulting high value relies to the ratio between the function value and the probability of sampling that value is large. Such

artifacts mostly appear, when a low-probability event occurs. These events cause large radiance values and increase the total variance of the result.

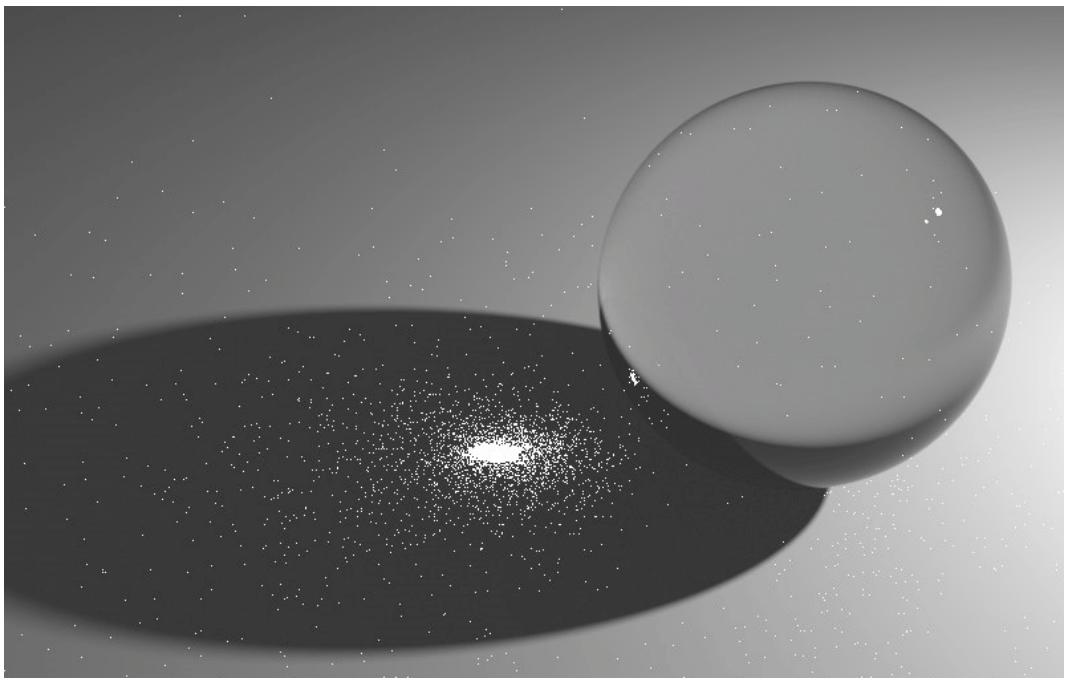


Figure 3.4.: Rendered dielectric sphere with the Blender renderer Cycles leading to fireflies within the scene (Source: `https://blender.stackexchange.com/questions/86635/will-increasing-samples-reduce-fireflies`)

3.6.3. Debugging Artifacts

In general analyzing code or complex implemented data structures can be done using the powerful debugger tool of its IDE. This tool is good for dynamic analysis of widely used algorithms and data structures. Nevertheless, debuggers are designed to work on source code itself which covers a wide area of applications, but in the domain of computer graphics the underlying data has mostly a very specific visual representation. Referring to path tracing algorithms, the corresponding data is represented by three-dimensional rays, scene geometry, energy, etc., which is poorly represented by floating point values in general purpose debugging tools. Besides, rendering algorithms are designed for high performance and are executed in parallel, which makes it in return hard to analyze or debug. In addition, the general complexity of a path tracing algorithm and its amount of data generated during tracing rays through the scene are enormous. The path count can exceed one billion in order to achieve a well converged image. The only visualization feedback

a computer graphics engineer has, in order to know if its implementation is correct, is comparing his results with the ground truth data. Which requires that the ground truth data is on the one hand available and on the other hand valid. In case the result does not match the ground truth data one is left with the IDE debugger for analysis.

Ideal Debugging System. Instead of working with visual feedback by comparing the rendered image with the ground truth data, the goal is to work directly on the 'production' scenes and images, so errors detected in the final result can directly be selected and debugged. However, this requires the complete rendering system to be deterministically structured to make debugging and code-stepping possible. This possibility of debugging would allow to directly select artifacts such as fireflies on the rendered image to receive corresponding path and debug information. Besides path visualization it should be possible to add arbitrary debug data to paths or intersections, which allow for an effective way of visual debugging of path tracing algorithms.

4. Visual Debugger

4.1. Motivation

While debugging complex light transport algorithms which lead to errors or artifacts in the rendered result, one is often limited to comparing images for visual feedback. The comparison can only show if the result is too bright or too dark for the entire or parts of the image. This type of visualization mainly shows the symptoms of the problem but give no additional information about its source. In order to gain deeper insights into the occurring problem, one has to use the IDE debugger which quickly reaches its limitations regarding complex rendering systems due to parallelism and complex data representation. Therefor, visualization is an effective way to gain deeper insights into complex data structures and what is happening during a computational processes. With regard to path tracing it is mostly not exactly clear what is happening at each intersection within the scene. A remaining question is how to debug and visualize such a complex path tracing algorithm traversing the scene.

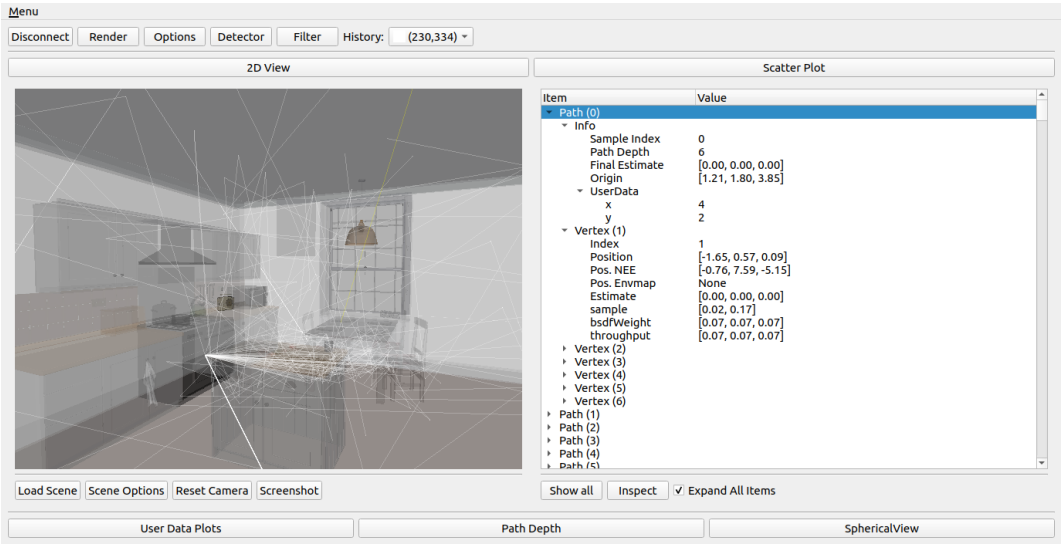


Figure 4.1.: The Visual Debugger showing traced paths through a single pixel in a three-dimensional scene representation (left panel). Additionally, arbitrary information per path and intersection point are available to analyze and debug the path tracing algorithm (right panel).

The ideal way would be an interactive visual representation with corresponding debug information, giving a deeper insight within the path tracing algorithm and its process. In addition the system should be light-weight with low-memory consumption, since saving all generated data would lead to a memory overhead. Another aspect is that most visualization frameworks like [HL14] and [SAH+16] are directly implemented within the used rendering system. Visual implementation related to a specific rendering system makes it harder to encapsulate the visualization framework and apply it to other systems. Therefore, the changes to allow a visualization and data exchange should be kept to a minimum.

We present a light-weight, interactive visualization framework as shown in Fig. 4.1. The framework supports a three-dimensional scene representation, visualizing the scenes geometry and traced paths within it (Fig. 4.1 left panel). Furthermore, a view supports the visualization of arbitrary debug information per path and its intersection points (Fig. 4.1 right panel). Therefor, the framework uses minimal memory consumption to allow a visualized analysis of path tracing algorithms based on MC integration. It allows debugging directly on the rendered image by selecting a pixel of interest. The user can navigate through the paths and inspect the MC-process step by step. In a hierarchical manner the tool starts at first visualizing the outcome of each path then the individual paths itself and at last each separate path vertex. Therefor, only minimal code changes on the rendering system are necessary to allow data exchange and visualization with the framework. Multiple views support various visualizations techniques for analyzing and debugging the path tracing algorithm. All views are connected and share the same data, allowing for fast interactive feedback.

4.2. Overview

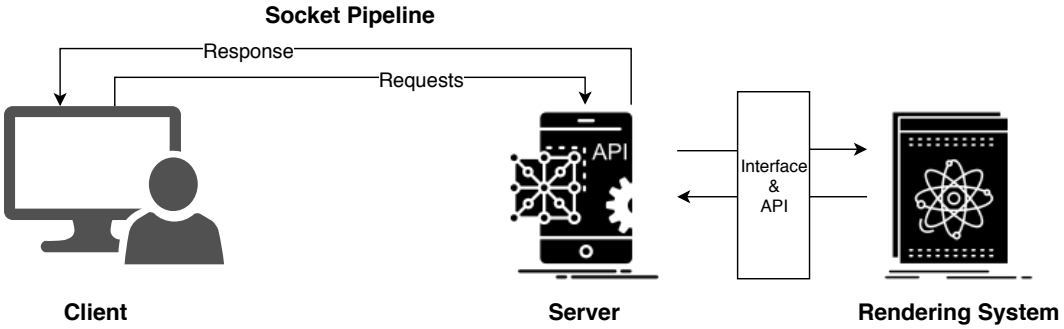


Figure 4.2.: Overview of the Visual Debugger architecture that shows a coarse example of the workflow. The client sends requests to the server which will response with collected data from the rendering system via an API.

The Visual Debugger framework expands and improves the process of development and analysis of unidirectional path tracing algorithms based on MC integration. Introducing a new interactive, light-weight visualization approach, in order to analyze and debug complex path tracing algorithms and their rendering systems. The framework helps to visualize, analyze and debug the rendering process of any MC based rendering system. It is easy to extend and works with an interactively GUI supporting the brushing and linking paradigm [Sii00] (Sec. 4.6.1). To adapt the visual framework to a rendering system only minimal changes need to be made. The framework is separated into two parts, a client and a server. The client provides an interactive GUI, allowing directly analyzing and debugging on the rendered image. The GUI, therefor supports multiple views which visualizes different data. The second part, the server, represents the rendering system which prepares the data for visualization. A light-weight interface makes it possible to easily modify any render system for visualization (Sec. 4.5.1). The interface supports a connection to the client, who can request specific data via a socket pipeline.

The client sends requests to the server asking for data to visualize. A simple workflow overview can be seen in Fig 4.2. As an example, the client sends a request to the server by selecting a pixel of the deterministically rendered image. The server starts a re-rendering step of this pixel. During this process all necessary data from the rendering system is collected via an application programming interface (API). Afterwards, the data is send back to the client which visualizes the received data in an interactive GUI.

4.3. Render Data

Debugging on the rendered image enables the possibility to directly select artifacts within the image. Besides, the user can add arbitrary debug information per path or intersection, which is automatically displayed on the client side. This data is then used for visualization and analysis. Since we want to analyze and debug the *Rendering Equation* and its evaluation using a MC path tracing algorithm (Eq. 3.13), we are interested in the following properties per sampled path:

- *path index*, (sample count index)
- *max path depth* (maximum number of intersections): M,
- *bounce number*: i,
- *final estimate*(contribution of each path): $L_o(x, \vec{\omega_o})$,
- *vertex position*: x_{n_i},
- *PDF*: $p(\vec{\omega}_{n_i})$ (probability of sampling the direction),
- *throughput*: $\prod_{i=1}^{M} \frac{1}{q_{n_i}} \cdot \frac{f_r(\vec{\omega}_{n_i}, x, \vec{\omega}_{n_o}) \, |\cos\theta_{n_i}|}{p(\vec{\omega}_{n_i})}$,
- *custom debug information* (can be added by the user)

The render data is collected during the re-rendering step of the requested pixel via an API (Sec. 4.5.2). After the collection step it is send to the client where it forms the basis of our visualization.

4.4. Seedmap

The main problem of path tracing visualization remains in storing all necessary data. The simplest way to visualize and debug a complex rendering system would be to record all generated information during the main rendering process and save it in a file on the disk. Afterwards, the stored data is used for visualization. This concept is introduced by Gribble et al. [GFE+12]. According to this publication, one disadvantage is the overall memory consumption by saving all necessary data in a file. As an example, think of rendering an image of size 512×512 pixels with a sample count of 1024. For a path visualization all intersection information within the scene are necessary. Under the assumption of this setup the memory consumption M can be computed by

$$M = \underbrace{(512*512)}_{\text{image pixels}} * \underbrace{1024}_{\text{samples per pixel}} * \underbrace{(3*4*40)}_{\text{bytes per intersection}}$$
$$\approx 128.85 \text{ gigabytes}, \tag{4.1}$$

representing only the traced path information. The bytes component results from the three-dimensional vertex representation data structure $p(x, y, z)$. Each component represents a float value with 4 bytes. 40 describes the amount of intersections per path, representing the path depth.

In order to analyze or debug the path tracing algorithm more information is necessary, such as the current contribution f_r, PDF $p(\vec{w}_j)$ of sampling the direction or self added output data. All of this data would further increases the memory consumption.

The difficulty of debugging complex rendering systems can be therefore split into two problems:

1. deterministic rendering state to allow incremental, reproducible debugging
2. save all necessary path tracing information with low-memory consumption

In order to support debugging directly on the rendered image, the entire process must be deterministic. PRNGs (Sec.3.3.2) provide a deterministic state by initializing the generator with a known seed at the beginning. Knowing this seed lets us reproduce the generated sequence of random numbers. Since the crucial part of MC integration is using random numbers to generate samples, the core render process of the rendering system has to be slightly modified to allow deterministic rendering and consequently debugging.

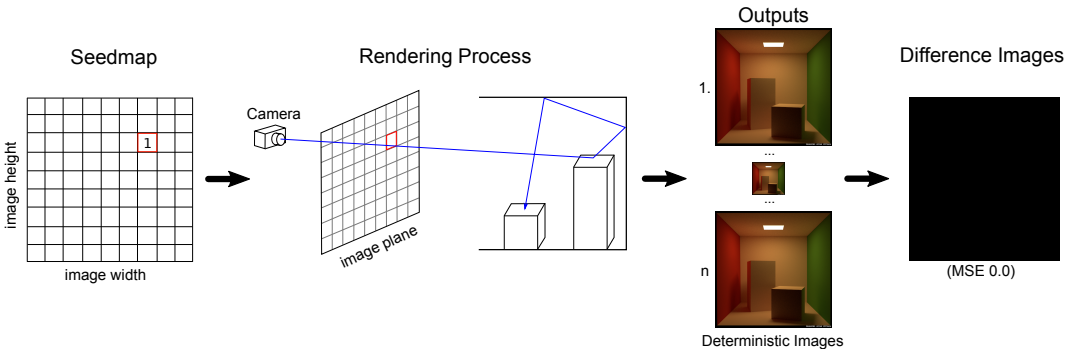


Figure 4.3.: Seedmap pipeline

We use the concept of a *Seedmap* to make rendering deterministic, which allows us to reproduce all important rendering data on the fly, while inspecting an area of interest. The Seedmap holds a seed for every pixel of the rendering image. The size is in that case dependent from the final image size. A deterministic rendering

process is characterized by the fact that after repeated rendering the result is always identical. With regard to path generation and path tracing, this means that the same paths are repeatedly generated conditioned by the seed. Our approach is based on the characteristics that rendering processes build on MC integration use PRNGs to generate samples which are then used to generate the traced paths through the scene. Fig. 4.3 shows the rendering pipeline, using a Seedmap to generate deterministic images. For rendering a pixel (image plane marked red rectangle), the corresponding seed is used to initialize the PRNG. The PRNG then generates samples for the ray tracing process. Each pixel belongs to its own seed and is independent from the other ones. Therefore, the property of running everything in parallel is not affected by this modification. Furthermore, this approach allows for a reproduction of each step during the rendering process which is similar to using a debugger. Comparing the output images with each other using a error function (e.g. MSE), the result is always zero, meaning that there is no difference between each deterministically rendered output image.

4.4.1. Memory Consumption

Interactively computing the current pixel state with the use of a Seedmap decreases the memory consumption significantly. Instead of saving all information during the main rendering process, the Seedmap gives the ability to reproduce each pixel state with just a predefined seed. Comparing to the memory consumption in Equation 4.1 with the same requirements of rendering a 512×512 pixel large image, by assuming the seed is a 4 byte integer, the memory consumption results in M_s given by

$$\begin{aligned} M_s &= \underbrace{(512 * 512)}_{\text{image pixel}} * \underbrace{4}_{\text{seed size}} \\ &\approx 1.05 \text{ megabytes}, \end{aligned} \tag{4.2}$$

representing the memory consumption to store all pixel seeds. Since all paths and their corresponding render data are interactively computed by selecting a pixel in the deterministically rendered image the additional memory consumption of all path information can be neglected in the formula. Initializing the Seedmap is done during the pre-processing step before the actual rendering takes place. Comparing the memory consumption of both, the Seedmap and the tabulation approach, a reduction of 99.9% is achieved.

The Seedmap concept solves both mentioned problems (Sec. 4.4) by making a complex rendering system deterministic to allow debugging and to do so with low memory consumption. Since all information can be re-computed on the fly with the predefined pixel seed.

4.5. Server

The server represents the rendering system with a light-weight interface build around the main rendering process to allow data exchange with the client via a socket pipeline. The server handles several requests from the client, such as rendering the deterministic image, re-computing a requested pixel and sending responses back to the client e.g. render data of the selected pixel, scene geometry, camera information or general scene information.

4.5.1. Interface

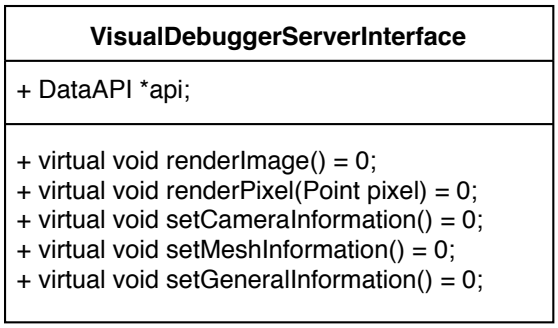


Figure 4.4.: UML class representation of the Visual Debugger Server Interface. In order to create a communication with the client and allow visualized debugging the pure virtual functions have to be implemented.

In order to allow a data exchange between the server and the client application a light-weight interface is provided. The interface interacts with the rendering system and handles incoming requests from the client. As shown in the UML diagram in Fig. 4.4, the user has to implement predefined functions of the interface in order to allow visual debugging. The functions covers the following functionalities:

1. Start rendering the deterministic image with *renderImage*.
2. Providing all scene geometry and camera information with *setMeshInformation* and *setCameraInformation*.
3. A function called *renderPixel* which reproduces the pixel state and its traced paths by using the provided seed from the Seedmap.
4. Setting general information about the scene and sample count within *setGeneralInformation* (can be left out).

The interface just responses to requests from the client by collecting data with the API (Sec. 4.5.2). After collection, the data is send to the client for visualization. The

interface is independent of the used rendering system which enables adapting it to any kind of rendering system that is based on MC integration. The user itself just has to implement the predefined functions to allow the communication between server and client.

4.5.2. Data Structure and API

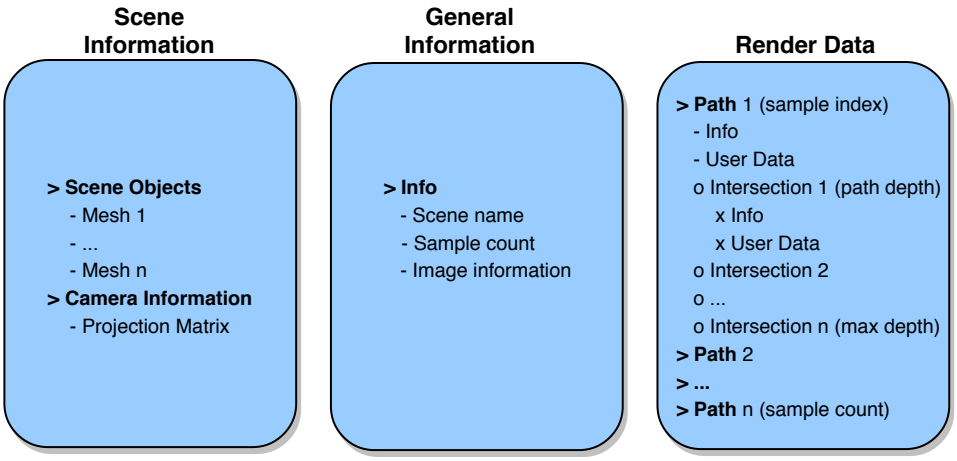


Figure 4.5.: Data packages and intern structure. The data is separated into three packages, which can be independently requested by the client.

The data is interactively requested by the client and can be classified into three packages parts as shown in Fig. 4.5. The scene information (left) consist of the data about the objects in the scene and the camera projection matrix. The general information package (middle) provides common information about the scene, like the scene name and the sample count. The render data package (right) holds specific information about all traced paths and custom user data for debugging. Each data package can be requested independently. These data packages forms the basis of our visualization and interaction technique. The core data including path visualization and debugging information is the render data package. In the following we present how the render data is collected during the pixel re-rendering step by the API. In order to keep the data apart, indices are introduced. These indices are used to identify each traced path and its corresponding intersection points. In our case the sample count references the path-index and each intersection point is referenced by the current bouncing index of the path.

For collecting the render data (Sec. 4.3) we introduce a logging application programming interface (API) based on the Singleton pattern [GHJV95]. The Singleton pattern restricts the instance count of the API class to one. This is useful since only the header file has to be included to access the API. We provide several functions to

add user specific data to the current path or intersection point as debug information. This data is collected during the pixel re-rendering step. The following pseudocode 1 demonstrates the use of the API in an iterative ray tracer.

Algorithm 1 Pseudocode of an iterative ray tracer showing how data is added via the Visual Debugger API. The core functions, separating the packages with indices are marked red. The blue marked functions add data in order to allow a 3D visualization of the traced paths in the scene. Green marked lines show how arbitrary custom data per path or intersection point can be added as debug information.

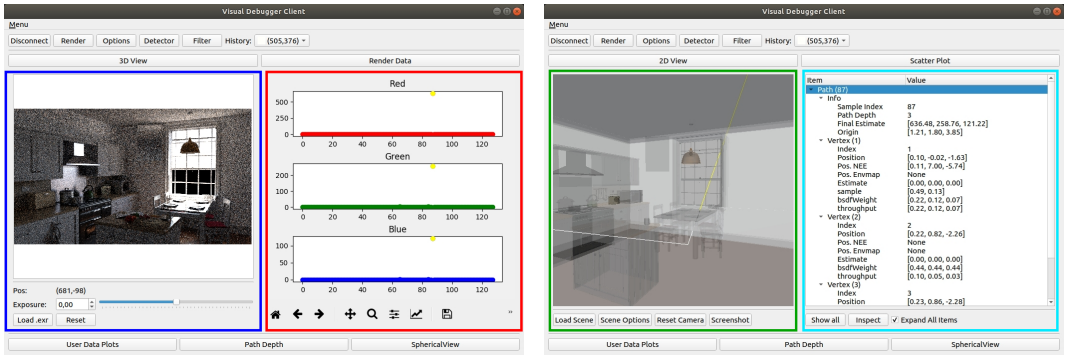
```
import vsddataapi
foreach pixel(x,y) in the image do
    L = Color(0) // Final estimate of MC integration
    for sample=1 to sample count do
        shoot a ray from the eye (camera) through the pixel
        vsddataapi->setPathIndex(sample)
        vsddataapi->setPathOrigin(ray.origin)
        L += trace(ray)
    end
    mcL = L / sample count
    pixel.setPixelColor(mcL)
    vsddataapi->setFinalEstimate(mcL)
end
function trace(ray):
    for bounce=1 to max bounces do
        find nearest intersection its with scene
        vsddataapi->setDepthIndex(bounce)
        vsddataapi->setIntersectionPosition(its.pos)
        evaluate BSDF at intersection point
        vsddataapi->addPathInfo("maxBounces", maxBouncesValue)
        vsddataapi->addVertexInfo("PDF", pdfValue)
        compute and trace new ray direction depending on material properties
    end
    return L // light contribution of traced path
end function
```

The core functions to identify the data are *setPathIndex* and *setDepthIndex* (marked red). They create in-memory data structures to handle all information as a package. To allow three-dimensional path tracing visualization, information about the path origin and each intersection point has to be added via the API functions (marked blue). Additionally, functions to add next event estimations and ray directions which end outside the scene and would intersect an environment map are provided. We support adding custom data (marked green) which is visualized and analyzed on the client side. The API has no additional requirements except for setting the path and

depth indices. If data is missing, the visualization of this part is skipped. This means, if one is interested in the traced paths within the scene all required information are setting the indices, the path origin, and each intersection position in the scene. Since in most cases the path origin is the camera origin it can be omitted. Consequently the first ray which is shot from the camera through the image plane is not visualized.

During the main render process of the deterministic image no data is collected. With use of the Seedmap (Sec. 4.4) we are able to reproduce the pixel state, all traced paths and rendered information with the pre-defined pixel seed afterwards. The data collected by the provided API is gathered interactively if a pixel of the deterministically rendered image is selected on the client side.

4.6. Client



(a) 2D render image (marked blue) and sample contribution view (marked red)

(b) 3D render scene (marked green) and render data view (marked teal)

Figure 4.6.: The Visual Debugger Client GUI displays the deterministically rendered image (marked blue) with a sample contribution plot (marked red). A three-dimensional scene viewer (marked green) providing visualizations about the traced paths and a view which contains all collected render data (marked teal). All views interactively use the brushing and linking paradigm (Sec. 4.6.1).

The client requests data from the server which is visualized in different views. The application is based on a Model-View-Controller (MVC) [EKP88] pattern, supporting an interactive GUI. The GUI is separated into four sub-views as shown in Fig. 4.6. The Render Image View (marked blue, Sec. 4.6.2) displays the deterministically rendered image and enables debugging by selecting a pixel of interest. A three-dimensional scene view (marked green, Sec. 4.6.3) provides visualization of the rendered scene and traced light paths. A scatter plot (marked red, Sec. 4.6.4) visualizes information about the sample contribution of each path. The render data view (marked teal, Sec.

4.6.5) displays the collected render data (Sec. 4.3) per path and its intersection points. The Render Image, Scene view and the sample contribution, render data view can be swapped, in order to compare the displayed data. This allows for a simple layout while still being able to compare the most important combinations of visualizations. A custom tool interface (Sec. 4.6.6) is provided which allows implementing custom functions for data visualization. These custom tools can interact with the rendering system via the socket and request specifically created data.

4.6.1. Brushing and Linking

Brushing and linking describes the connection between two or more view elements sharing the same data, such that a change to the representation in one view automatically affects the representation in the others. This makes it an important technique in interactive, visual analysis to perform viewable exploration and analysis of large and structured data sets. Therefore, all views and their data in the Visual Debugger are connected with the brushing-and-linking paradigm [Sii00]. Linking 2D and 3D data is especially powerful since it connects scientific and information visualization in one application together.

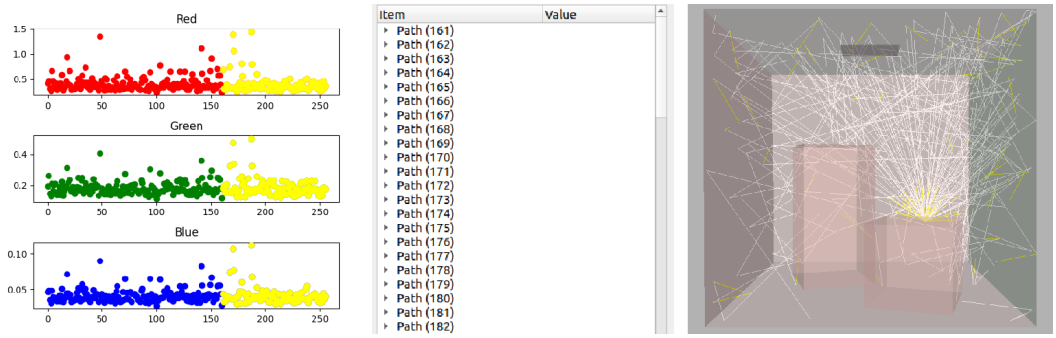


Figure 4.7.: Brushing and linking within the different views of the Visual Debugger. Selected sample contributions (left marked yellow) affect the Render Data view (middle) and the 3D Scene view (right).

Fig. 4.7 shows an example of the brushing and linking paradigm within the Visual Debugger GUI. Selected data in the Sample Contribution View (left, Sec. 4.6.4) automatically affects the Render Data View (middle, Sec. 4.6.5) and 3D Scene View (right, Sec. 4.6.3). Showing the corresponding path information with a three-dimensional visualization. The same applies if a path or vertex item in the Render Data View is selected. This example shows how paths can be analyzed with the brushing and linking paradigm by supporting multiple views which displays the same data in different ways. Interactive visualization in combination with the paradigm applied gives the user a powerful tool for analyzing path tracing algorithms.

4.6.2. Render View

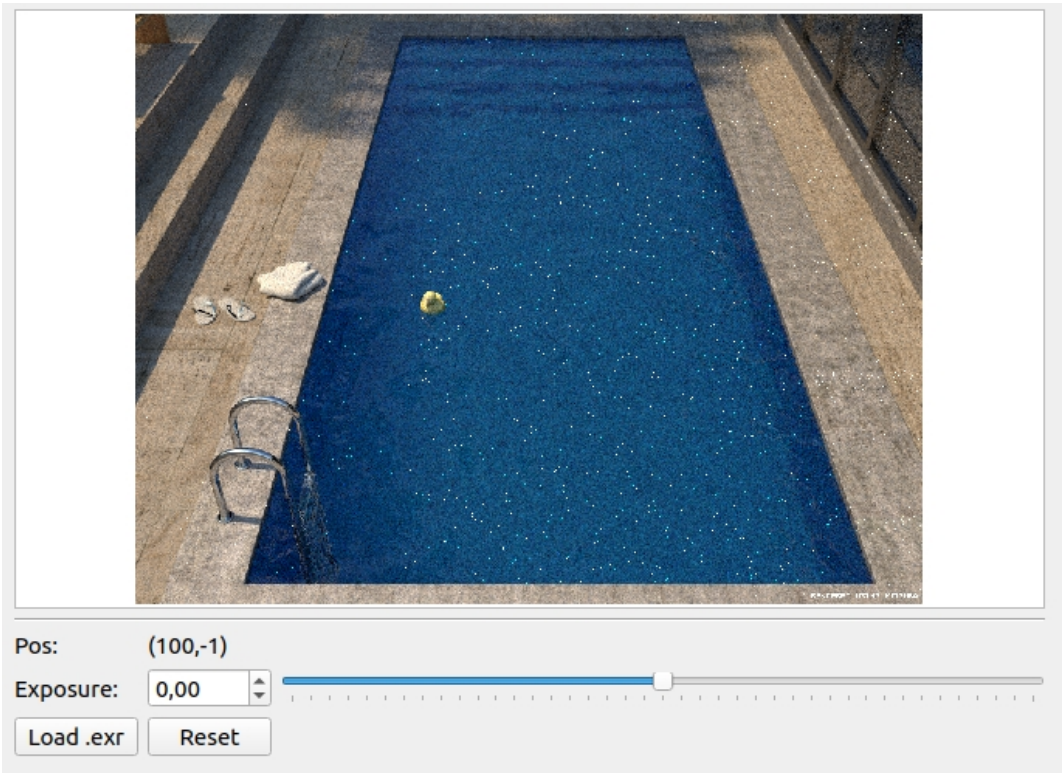


Figure 4.8.: Render View showing a deterministically rendered swimming pool scene image (Mitsuba [Jak10] volpath, 64spp).

The render view Fig. 4.8 displays the deterministic result of the rendering process. The main task of the Render View is to send pixel requests selected by the user to the server which computes and collects all necessary information via the API. At this point the server takes the deposited seed from the Seedmap and reproduces all traced paths through this pixel within the scene interactively. The user sends a request by selecting a pixel of interest e.g. a bright firefly pixel. This allows the user to interactively analyze and debug on the rendered image. A request with the position of the click is send to the server interface via the socket pipeline. On the server side this specific pixel is recomputed, gathering all information with the provided API. A history keeps track of all requested pixels. Previous deterministic rendered images can be loaded via a Drag and Drop function. If no deterministic image is provided a request can be send to the server, starting the normal rendering process. Afterwards, the client receives a response which contains the image information, which automatically loads the image. With the Render View, the user can directly debug on the rendered image by requesting pixel of interest. By the re-rendering step and the following response from the server the user preserves more information

about the requested pixel.

4.6.3. Scene View

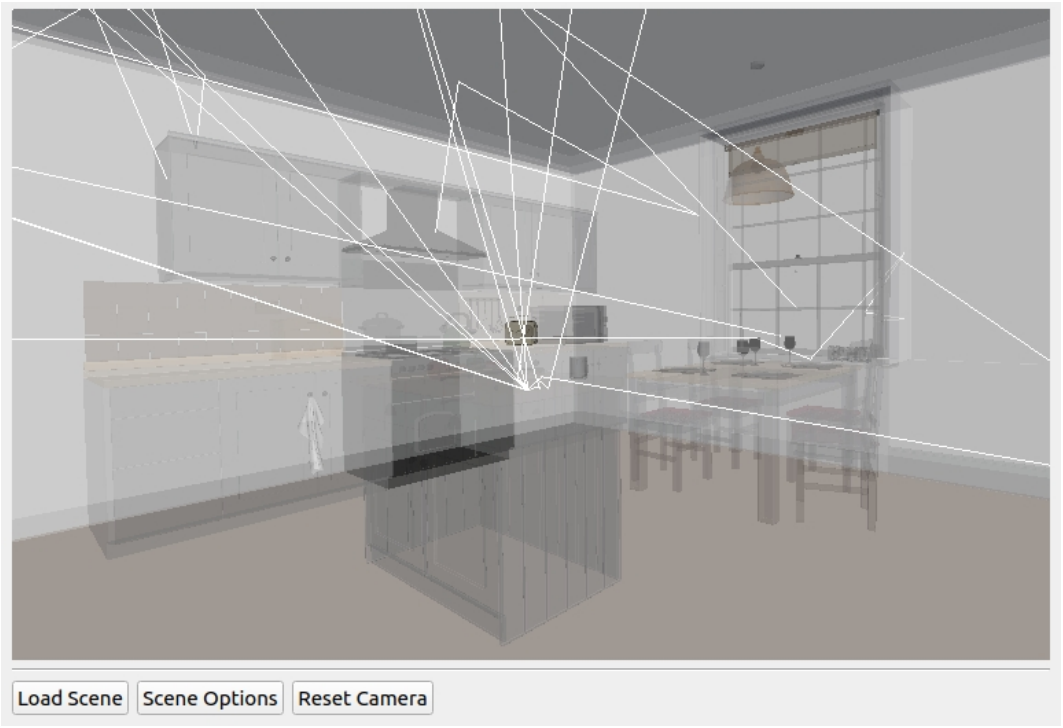


Figure 4.9.: Three dimensional representation of the Country Kitchen scene visualized with the Visual Debugger's 3D viewer. Additionally, paths, that were traced through the scene, are visualized as white lines.

The scene view shown in Fig. 4.9 provides a three-dimensional view, displaying the scene geometry and paths (as white lines) in the rendered scene. The view is used to explore and visualize the render data in 3D object space. Since rendering is based on path tracing which consists of tracing rays within the scene, the path visualization helps to understand the general light transport in the scene. The scene geometry can be loaded independently of the traced paths. All objects are semi-transparent to allow displaying paths within dielectrics or participating media. The transparency can be changed subsequently. Traced paths are visualized as thin white lines. A path consists of multiple line segments between. Each segment connects two intersection points. Selected segments are highlighted green. Each intersection point can visualize their incoming $\vec{\omega_i}$, outgoing $\vec{\omega_o}$ and next event estimation ray state. Next event estimations are visualized blue if not occluded, otherwise they are marked red. Rays ending in an environment map are marked yellow. Intersections are handled

and visualized as vertices within the scene. They are marked orange if selected. Like in other views we provide a three-dimensional rectangle selection tool for selecting specific intersections of interest within the scene. The selector is connected with the brushing and linking paradigm (Sec. 4.6.1) and informs automatically all other views about the new selected vertex. The Scene View provides a three-dimensional scene representation with various visualization options for the traced light paths within, in order to help the user to explorer and analyze them.

Scene Options

For a better interaction with the 3D scene view several options are provided, which directly influence the scene interactively. These options include: comparing paths and intersection points, changing the transparency of the scene and modifying the camera settings. Changing the size and opacity of paths or vertices for a better visualization. Displaying incoming, outgoing or next event estimation of the path or single intersections. These options give the user full control for the three-dimensional scene representation and the traced paths within it.

4.6.4. Sample Contribution View

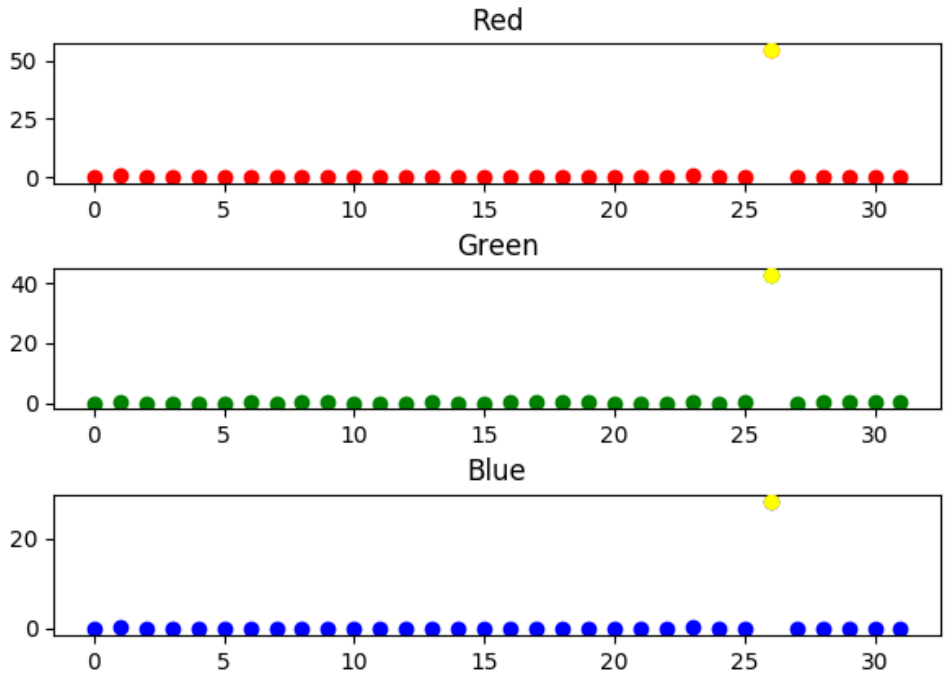


Figure 4.10.: Sample Contribution View showing a pixel sampled 32 times. Only one path lead to a high contribution (marked yellow). This path is automatically detected by the provided high variance detector (Sec. 4.6.4).

The Sample Contribution View shown in Fig. 4.10 provides an overview over each path and its contribution. Displaying the sample contribution for each path can be essential in order to detect outliers such as fireflies or general paths with high or low contribution. The data is presented as a scatter plot, split into the color spectrum of red, green and blue. In case of analyzing special cases an alpha channel is provided. The data is connected with the brushing and linking paradigm (Sec. 4.6.1). The view provides multiple selection tools to mark and add more items of interest. The view helps the user to analyze paths regarding to their contribution.

High Variance Sample Detector

We provide a high variance detector which automatically detects paths with high contribution. The detector works on the provided render data (Sec. 4.3) and detects outliers based on their final estimate. Hence, it can be seen as a firefly detector. Detecting and removing fireflies is one big topic in photo-realistic rendering. There

exists many approaches, like working with half buffers [Jef18], or re-weighting firefly path samples [ZHD18] to remove high variance samples. A naive approach would be to use much more samples. Another method is clamping. In this case values are clamped to a fixed maximum brightness value so that outliers, which are greater than this value are clamped to this threshold. Nevertheless, clamping results in a biased result or other image artifacts. The most common way to remove fireflies is done after the main rendering procedure of the image, in the so called post-processing step. During this step filter algorithms are applied to remove or adapt the firefly to its surrounded pixels [TM98]. Instead of removing fireflies by clamping or during the post-processing step one could analyze the path tracing algorithm which solves the *Rendering Equation*. This would make the post-processing step unnecessary and could save time for e.g. animated films. As described firefly artifacts occur by applying Importance Sampling (Sec. 3.2.2), this is due to the miss match of the shape of the PDF p and the integrant f. The most important information for analysis can thus be limited to the PDF p and the contribution f.

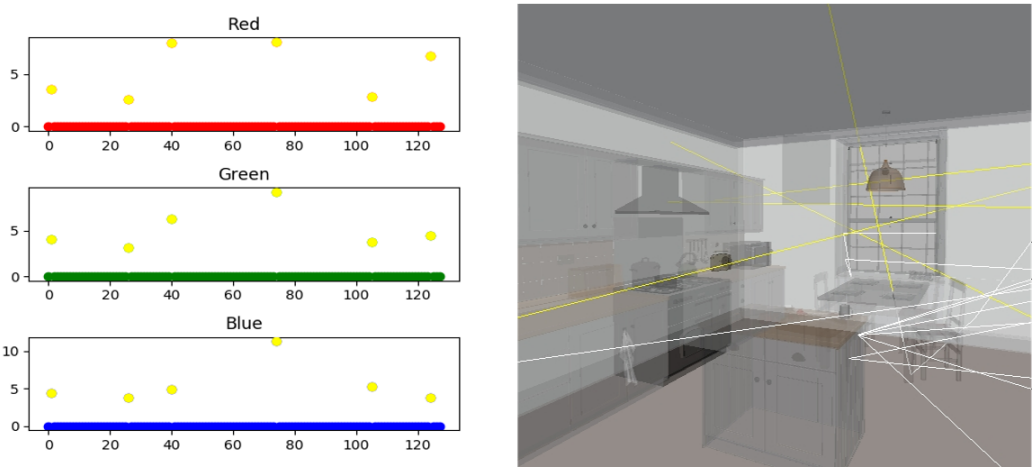


Figure 4.11.: The Sample Contribution View (left, Sec. 4.6.4) and the 3D Scene view (right, Sec. 4.6.3) showing the result of the outliers detection based on the final estimate contribution per path (marked yellow dots).

In order to detect outliers with high contribution two algorithms are provided. The first algorithm is based on the *mean-squared error* and the *standard deviation*. The second one is based on the *Generalized ESD Test for Outliers* [Ros83]. The ESD Test is more robust than the first algorithm which compare values of standard deviation. The detector detects and selects the outliers as shown in Fig. 4.11 (marked yellow). Paths with a high contribution are selected, updating all other views by the brushing and linking paradigm (Sec. 4.6.1). This allows the user to directly visualize paths that lead to a high contribution by selecting a pixel.

4.6.5. Render Data View

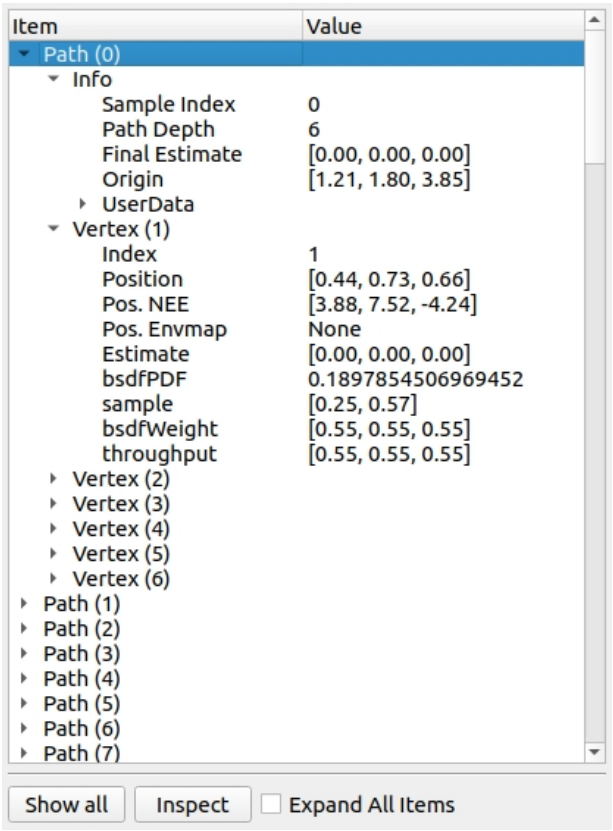


Figure 4.12.: The Render Data View shows arbitrary information about all traced paths in the scene. The data is arranged in a tree-structure, where one root node represents one path. Each path consists of multiple sub-nodes about general and intersection information. Each node displays arbitrary added user data, helping to analyze and debug the traced paths.

Fig. 4.12 shows the Render Data View, providing an overview over all collected data during the pixel re-rendering step. The view can be seen as a debugger view, holding information for each created path and its corresponding steps (intersections). Instead of stepping through source code like a debugger, each step in this case is connected to a path and its intersections. The data is arranged in a tree-structure, where a root node represents one traced path in the scene. Each node holds general path information, arbitrary custom data added by the user via the API and corresponding intersection points as child nodes. The same structure applies for each intersection point. In combination with the 3D Scene View (Sec. 4.6.3) and the brushing and linking paradigm (Sec. 4.6.1) one does not only have the debug information of the

individual paths and their intersections, but also an interactive 3D scene, which is automatically updated if a path or vertex node is selected. The view gives the user the opportunity to directly debug the path tracing algorithm by inspecting each path and its intersection parameters.

Ray Filtering

We provide a filter algorithm which directly works on the collected render data (Sec. 4.3). The ability to filter data by specific criteria provides more flexibility in analyzing. [HL14] introduces a package filtering system based on their collected ray state data. The filtering algorithm is based on a multigraph, where each data package represents one node. Instead of using a graph structure for filtering, we store our data in hash tables. Hash tables provide fast access and good efficiency when it comes to filtering as long as the corresponding hash key represents the search key. Each path consists of multiple hash tables, depending on its intersections. Each hash table can be checked with $O(1)$ if the search key is included.

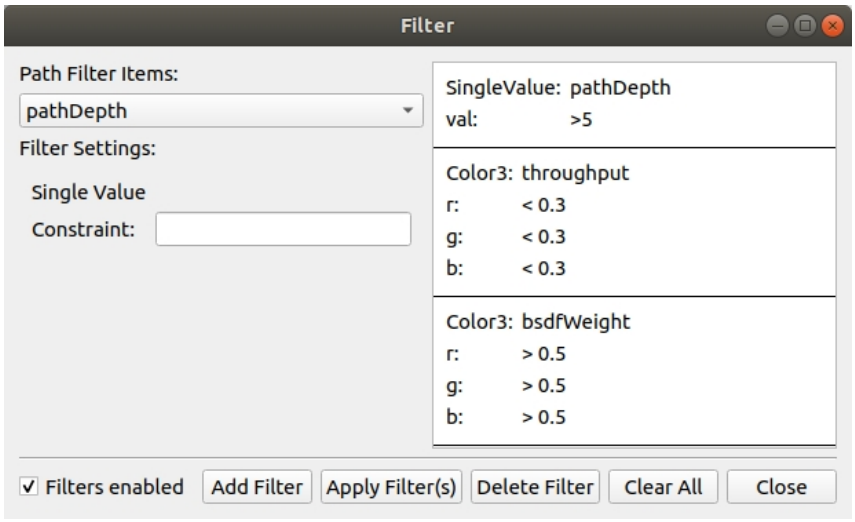


Figure 4.13.: The filter view handles adding and removing filters which are applied to the collected render and custom user data (Sec. 4.3).

Fig. 4.13 shows the filter view which handles adding and removing various filters. In order to allow filtering in several ways, multiple filter criteria can be applied. Therefore, a drop down menu (left side) holds a list of all added custom user data. Depending on the selected item, the filter view is updated to support the specific filter constraints. All added filters can be seen within a list (right side). This list keeps track of the added filter and general information about the filter criteria. If multiple filters are active the render data has to satisfy each added filter. In case of removing a filter from the list, the removed path data is restored. All changes automatically

affects the other views by the brushing and linking paradigm (Sec. 4.6.1). This allows a more detailed filtering for analyzing of the corresponding data, giving the user more control to visualize specific parts of the collected data.

Save And Load State

In order to be more flexible and independent from the server a save and load function is provided. The save function allows to store the current pixel state and its corresponding render data in a XML file. The stored data includes the data packages Scene Information and Render Data (Sec. 4.5.2). This saves all three-dimensional geometry with the camera information and the render data of the current selected pixel. Occurring problems during path tracing can thus be easily saved and restored. This can be especially useful in teaching to give students the ability to visually restore problems encountered during path tracing. Furthermore, since several developers usually work on new path tracing methods, this stored data can be loaded independently for analysis. The developer does not have to modify the Seedmap or start the server interface.

4.6.6. Custom Tool Interface

With the custom tool interface the user can request and display own data within the Visual Debugger framework. In most cases new ray tracing approaches work on new structures, which can not be added via the default functions of the provided API. For those cases the tool interface offers the possibility to extend the client by a custom tool which can handle and display these data structures. A tool can request own generated data from the server interface. It is connected to the provided API. Hence, each tool can be requested via the API with an unique identifier to add or work with data generated during the pixel re-rendering process. In addition, the custom tool interface is connected with the brushing and linking paradigm (Sec. 4.6.1) to allow interactive feedback from all other views. This offers the user a wide range of possibilities to visualize own created data structures. Besides, the option to generate and request own data, the tool interface has access to the render data of the current selected pixel. This gives the user the possibility to create own statistics of the already provided render data.

User Data Plots

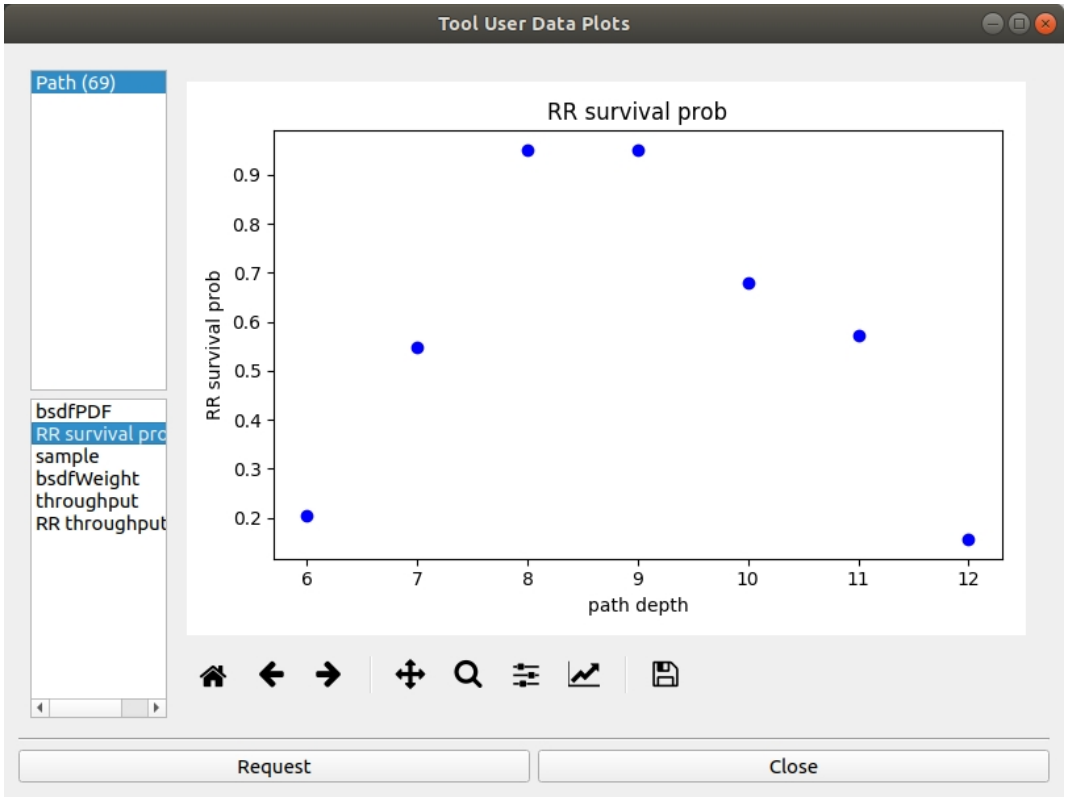


Figure 4.14.: The User Data Tool View displaying the collected Russian Roulette survival probability q_{n_i} (Sec. 3.5) over the path depth of a traced path (sample index 69). All added user data: bsdfPDF, bsdfWeight, throughput, etc. (list bottom left) is automatically analyzed and visualized within plots.

The User Data Tool works on the custom added render data of each intersections point. It visualizes the user added data within a plot diagram over the path depth which can be seen Fig. 4.14. The plot shows the Russian Roulette (Sec. 3.5) survival probability q_{n_i} for each bounce of a traced path. It reveals how the probability at each bounce processes through the path until the path gets terminated. Since the API supports multiple data types the tool provides an interactive plot system, which visualizes the data types in different plots. This includes single value plots e.g. information about the probability of sampling a direction, three-dimensional plots for 2D points and RGB plot to visualize e.g. the current light contribution at one intersection point or the current throughput. All plots are automatically created by the provided render data, giving users a better insight of each path and their corresponding intersection data.

Path Depth

The Path Depth Tool provides a summary of each traced path and its reached depth, visualized within a plot depicted in Fig. 4.15. As there can be seen most traced paths terminate at a path depth of five. Only few paths reach a higher path depth. The maximum can be seen at a depth of twelve. Finding and analyzing paths that end too early can be essential to determine why an image does not converge correctly. By terminating paths through *Russian Roulette* (Sec. 3.5) or other ending rules to save rendering time, essential contribution can be lost. Another aspect may be the path distribution in a scene. The reason why most paths terminate at a depth of five in the shown Fig. 4.15 relies on the fact that Russian Roulette is interfering at this path depth.

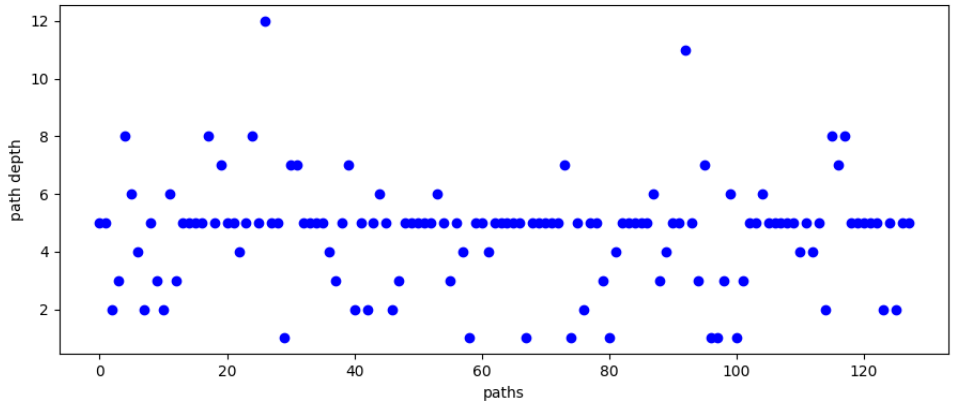


Figure 4.15.: Path Depth Tool visualizing for each traced path the reached path depth. Most paths are terminated at a path depth of 5 caused by Russian Roulette (Sec. 3.5)

Spherical Incident Radiance View

The Spherical Incident Radiance View shows how the custom tool interface can be extended to visualize user specific data types. This can be essential in order to analyze or verify new implemented approaches and the corresponding data structure. To gain a deeper insight of each intersection point and its surrounding view, the Spherical Incident Radiance View shown in Fig. 4.16 provides a 360 degree spherical rendered view for each path intersection point. The three-dimensional scene view shows the Torus scene with a traced path, interacting with the geometry (marked blue and red rectangle). For each intersection point a corresponding Spherical view image is rendered (right side marked blue and red). Visualization and calculation are separated among client and server. The corresponding data is computed by the rendering system on the server side. On the client side the tool provides a view which can interactively request the data from the server. With the Spherical View

Tool one gets deeper insights in what parts of the scene are visible by certain regions. This can be essential to figure out where exactly light comes from. Sec. 5.4, since there it is an essential tool to validate the approximations used for guiding.

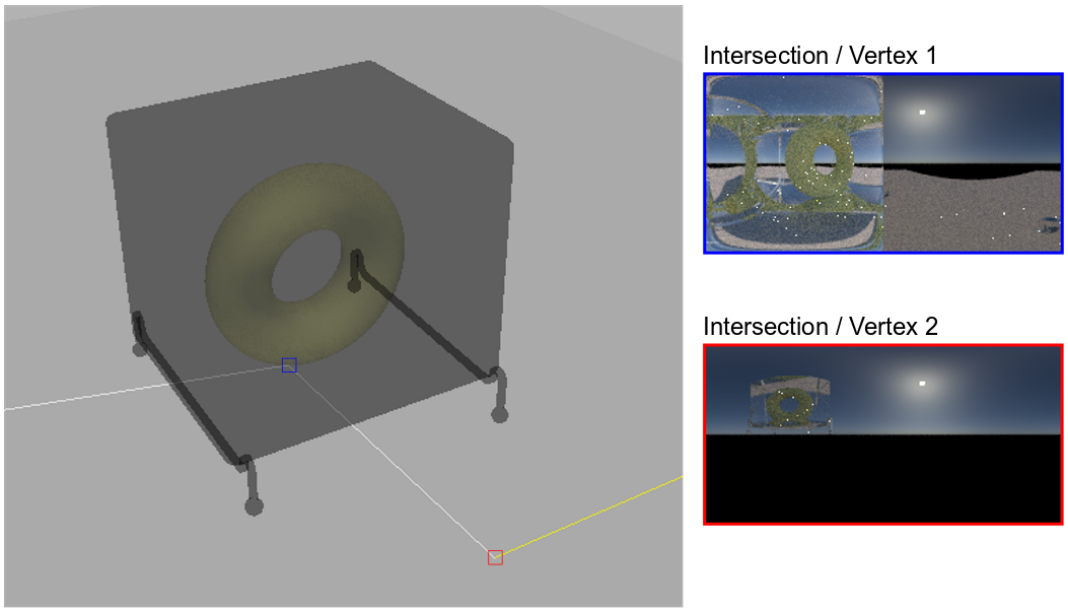


Figure 4.16.: The 3D Scene view displaying the Torus scene with a visualized traced path, which interacts with the scenes geometry. The Spherical Incident Radiance Custom Tool is used to display a spherical rendered view from the first (marked blue) and second (marked red) intersection.

5. Use Cases

This chapter provides an overview of the use cases of the Visual Debugger. We present how framework can be used to identify errors and to show various rendering and ray tracing problems. For this purpose we adapted the Visual Debugger framework to the Mitsuba [Jak10] renderer to allow visual debugging and path tracing analysis. Mitsuba is a research-oriented rendering systems following the architecture of Physically Based Rendering (PBRT) [PJH16]. It is written in C++ and supports unbiased as well as biased rendering techniques.

5.1. Floating Point Accuracy Problem

Computers use floating-point (FP) arithmetic to represent real numbers. This FP representation is an approximation to support a trade-off between range and precision. Rendering systems highly work with FP numbers to compute accurate intersections during ray tracing. In the worst case, this trade-off can lead to an unknown behavior that can lead to artifacts in the rendered image.

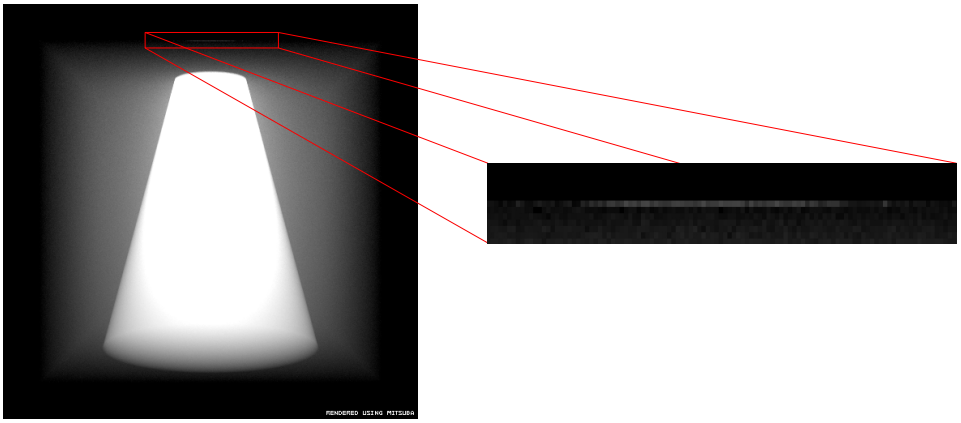
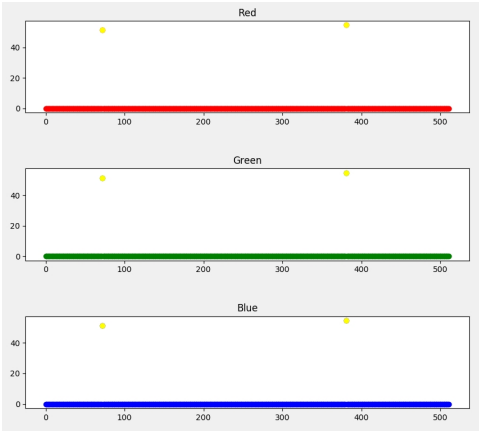


Figure 5.1.: Fireflies in a cube's edge enclosing a participating medium. (Mitsuba volpath, 16k spp)

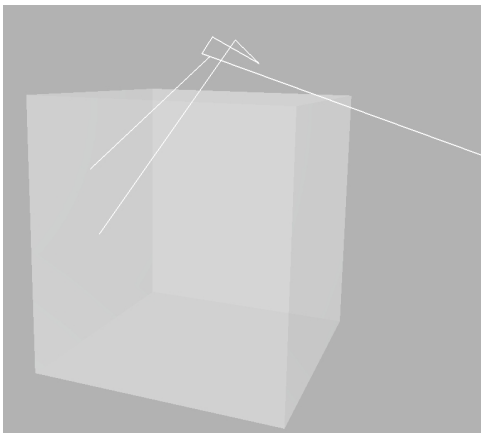
Think of the following scene composition. The camera is facing towards a cube with a spotlight placed over the cube. The light falls from above straight onto the cube. The cube itself represents a participating medium e.g. fog or smoke. Rendering this scene

with Mitsuba's volume path tracing algorithm leads to the output depicted in Fig. 5.1. On the top edge of the cube, firefly artifacts occur in a straight line. In a physically correct rendering these artifacts should not appear. Analyzing the problem with an IDE debugger could not help to solve the problem. It quickly reaches its limitation due to the complex rendering system and its parallelism.

With the Visual Debugger we have the ability to directly select one firefly of interest on the deterministically rendered image, which will visualize all traced paths through this pixel. This helps us to identify the problem that lead to the fireflies on the top edge of the cube. The Path Contribution View (Sec. 4.6.4) shows all contributions per path, in which one can identify two paths, leading to a high contribution as shown in Fig. 5.2a (marked in yellow). The corresponding trajectories are visualized in the 3D Scene View (Sec. 4.6.3) and can be seen in Fig. 5.2b.



(a) Sample Contribution View, showing two paths with a high contribution (marked yellow)



(b) 3D scene view showing the two paths which barely intersecting the cube.

Figure 5.2.: The Visual Debugger Sample Contribution View (Sec. 4.6.4) and 3D Scene View (Sec. 4.6.3) identifying intersection problems of traced paths.

The visualization reveals that scatter events occur outside the medium and contribute a high value. These events are not allowed to occur outside the object that acts as a hard boundary to the medium. However, the check if the ray is still inside the cube medium fails. The ray is traced further, where scatter events occur within the light beam from the spotlight. All other paths evaluate to zero, as they did not intersect with the edge of the cube and no environment map is present. The problem can be traced back to the so called Epsilon ϵ variable used in rendering systems. This value has a fixed definition. It can be described as a very small floating point number, which depends on the accuracy. It is used to prevent floating-point accuracy problems which can occur during path tracing. For example, preventing self intersection during subsequent ray traversal after a surface intersection.

This use case's problem is based on the same problem. A distinction is made between single-precision accuracy and double-precision accuracy. In the Mitsuba renderer ϵ is defined on single-precision with $\epsilon = 10^{-4}$ and on double-precision with $\epsilon = 10^{-7}$. A solution to the floating point accuracy problem during intersection testing is to compile Mitsuba with double point precision to reduce epsilon. Fig. 5.3 shows cutout comparison between the rendered scene with single and double precision. Occurring artifacts in the single precision image (marked blue) do not appear any more in the rendered image with double precision (marked green).

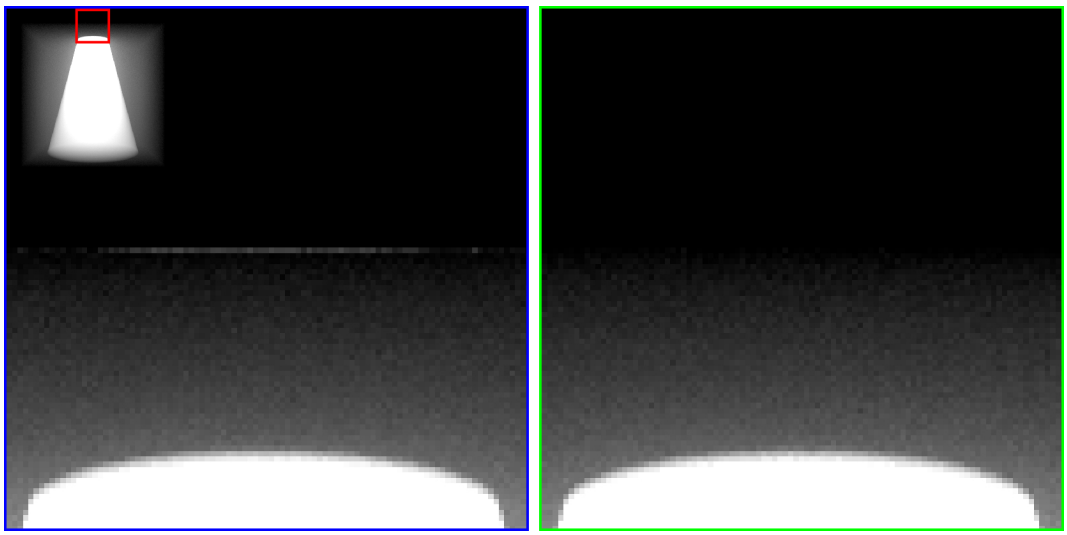


Figure 5.3.: Comparison between single and double precision rendered images. The left image shows the rendered result with single precision, which lead to artifacts along the cube's edge. The right image shows the rendered result with double precision, resulting in a clean image without occurring artifacts (Mitsuba volpath, 16k spp).

5.2. Volumetric Material Distributions For Fabrication

Figure 5.4.: 3D prints by Sumin et al. [SRB+19].

The Visual Debugger has been used to identify and solve a problem in the work of Sumin et al. [SRB+19]. They introduce a new approach for high-fidelity color texture reproduction on 3D prints by effectively compensating for internal light scattering within arbitrarily shaped objects as shown in Fig. 5.4. The problem at hand arises from an appearance simulation of volumetric material distributions for fabrication. Rendering is used to predict the high-fidelity appearance of printouts produced by a polyjet 3D printer. In a study aiming to find the globally optimal solution for material arrangements forming a specific surface appearance, the authors build an optimization system around a renderer in a canonical setup. In this setup, a thin sheet geometry is filled by a heterogeneous medium and rendered with a customized sensor, that has different ray origins per pixel. Each pixel is aligned with texture pixels on both sides of the geometry, that form the target appearance. Despite a symmetrical filling of the voxels, the rendered appearance on both sides of the sheet differed depicted in Fig. 5.5.

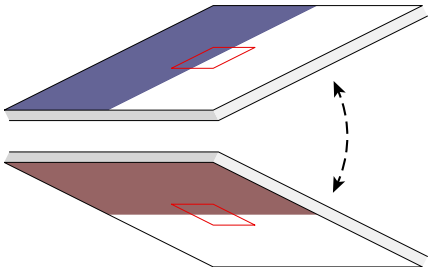


Figure 5.5.: (Thin) Two-Sided target textures are arranged into a centered region of interest (red square) on the front (or back) of a slab of parameterisable thickness. Edge-padding enlarges the volume to eliminate in-scattering of external lateral illumination [SRB+19]

In this case the traced paths through the volumetric material are analyzed. For this purpose, we extend the Visual Debugger framework to support visualization of bounding boxes per intersection point within the 3D scene view. Using this extended ability, we were able to visualize the exact bounding boxes of the underlying medium and its relation to the containing shape. With the Visual Debugger we found a mismatch in the alignment by half a voxel. Additionally, we visualized the voxels, where rays enter the medium. By looking at multiple samples of the same pixel, there a numerical issue could be spotted. As shown in Fig. 5.6, rays entered the voxels at the corner (marked red), which lead to instabilities of a rounding operation. In the corrected version, rays enter in the center of the voxel, avoiding such floating point instabilities altogether.

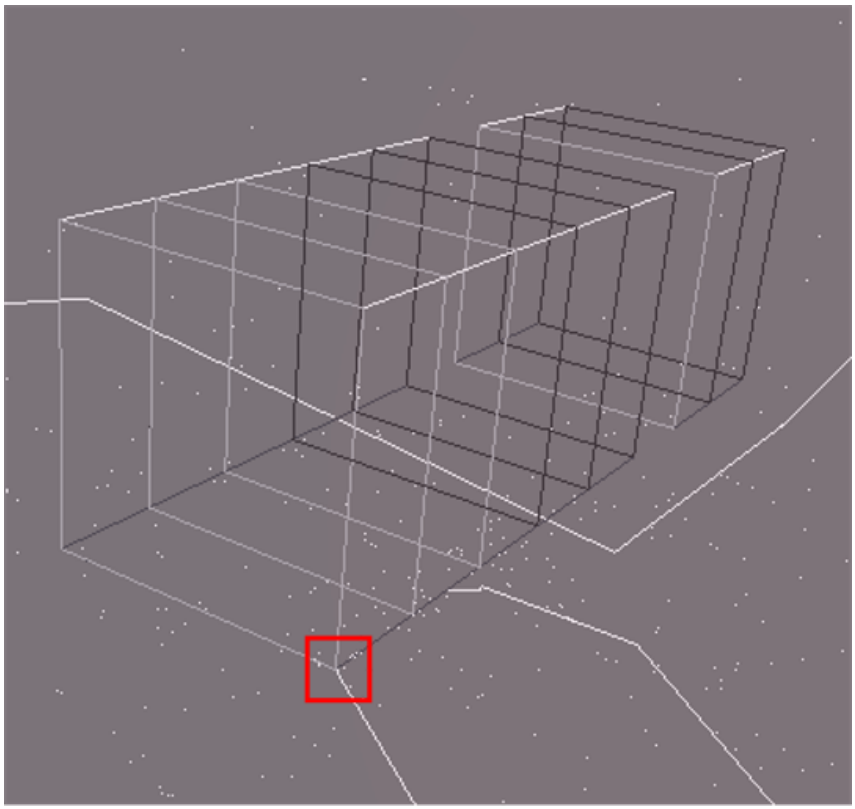


Figure 5.6.: Volume Debugging with Bounding Boxes. A numerical rounding operation lead to instabilities where rays enter the voxel at the corner (marked red).

5.3. Portal Masked Environment Map Sampling

[BNJ15] introduces a new technique to efficiently importance sample direct illumination from an environment map in indoor scenes, where lighting is typically only visible through small openings (e.g. windows). The visibility of those small opening are often addressed manually by placing a portal around each window to allow direct sampling towards these openings. They use a warped environment map to identify and importance sample only the visible parts of the surrounding through a portal.

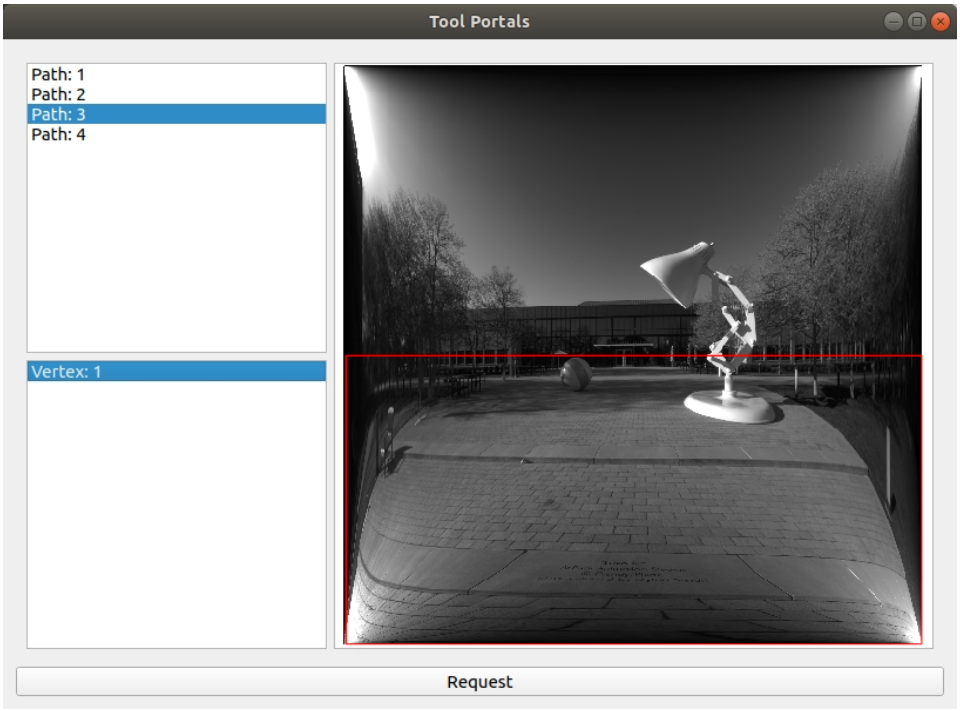
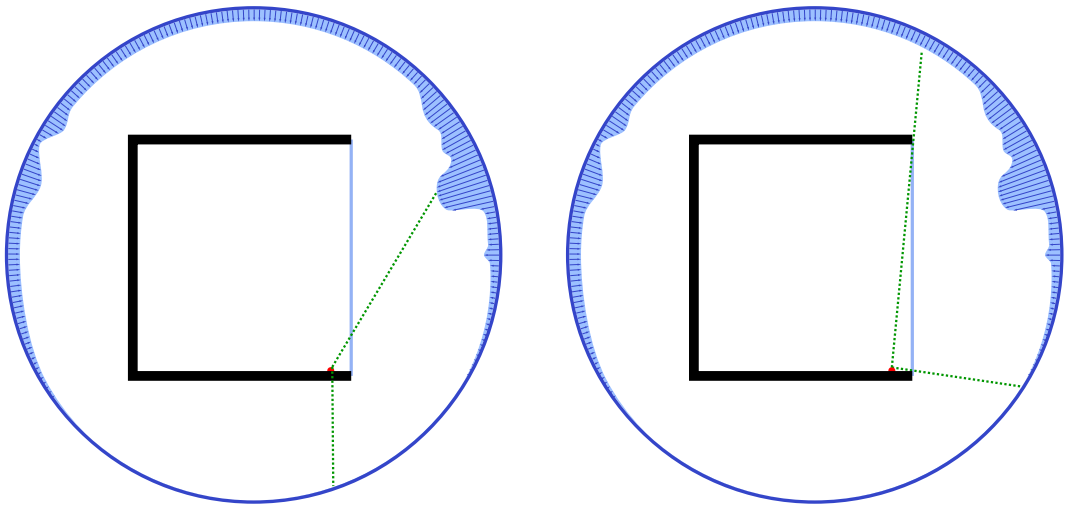


Figure 5.7.: Visual Debugger Tool Portals Extension showing the rectified environment map with the visible region of an intersection point close to the portal in the scene (marked red).

This approach was implemented in the Mitsuba renderer as a student project. The Visual Debugger did not exist yet and most debugging was done by using the provided IDE debugger. However, debugging was quite difficult and all problems could not be resolved. Until now, with the Visual Debugger we are able to visualize and track every intersection point and its visibility through each portal. We extended the tool interface of the Visual Debugger with a new tool to visualize portals and their corresponding rectified environment map. It is now possible to show the warped environment map in its own view. We can show each visible area per intersection point within the environment view, which can be seen in Figure 5.7. The image

shows the warped environment map with a red rectangle which denotes the visible part of an intersection point near the portal plane.

The scene in this case is a Cornell Box whose front opening represents the portal. Analyzing occurring fireflies at the bottom surface and their corresponding visible area through the portal reveals, that the visible part did no match the correct area. We found a miss match in the calculation of the visible area of the intersection points near the portal plane.



(a) Wrong visible area of intersection point (marked red) near the portal plane

(b) Corrected version of visible area

Figure 5.8.: Visualization of the Cornell Box scene with a surrounding environment map and a light portal on the right opening. Green dotted lines visualize the visible area of an intersection point close to the portal (marked red)

Figure 5.8a gives an impression of the wrong computed visible area (green dotted lines). Besides Fig. 5.8b displays the corrected version. The mismatch of the visible region lead to a wrong computed PDF, which in this case has lead to the occurring fireflies.

5.4. Path Guiding

Rendering complex scenes with difficult light transport in photo-realistic scenes can take multiple hours in order to compute a noise-free image. Recent approaches like [VKv+14], [HEV+16] and [MGN17] introduces path guiding algorithms to efficiency sample light path directions to their contribution.

Instead of only using directional samples based on the BRDF

$$\begin{aligned} p_w(\vec{\omega}_i|x,\vec{\omega}_o) &= p_{f_r}(\vec{\omega}_i|x,\vec{\omega}_o) \propto f_r(\vec{\omega}_i,x,\vec{\omega}_o) \text{ or} \\ &= p_{f_r}(\vec{\omega}_i|x,\vec{\omega}_o) \propto f_r(\vec{\omega}_i,x,\vec{\omega}_o)\cos\theta_i, \end{aligned} \tag{5.1}$$

they also include knowledge about the incident radiance

$$p_{L_i}(\vec{\omega}_i|x,\vec{\omega}_o) \propto L_i(x,\vec{\omega}_i). \tag{5.2}$$

They use 50/50 sampling between incident radiance and the BRDF resulting in a sampling probability

$$p_w(\vec{\omega}_i|x,\vec{\omega}_o) = 0.5 \cdot p_{f_r}(\vec{\omega}_i|x,\vec{\omega}_o) + 0.5 \cdot p_{L_i}(\vec{\omega}_i|x,\vec{\omega}_o). \tag{5.3}$$

Except Herholz, who uses the product of the incident radiance and the BRDF. Fig. 5.9 shows a visual comparison between an un-guided and guided path tracing algorithm with equal sample count (16 spp). As there can be seen the light reflections beneath the table (marked green) can not be clearly rendered with the un-guided path tracing algorithm. For an improved converged image much more samples are required. In comparison to guided path tracing the light reflection is efficiently sampled by only using few samples.

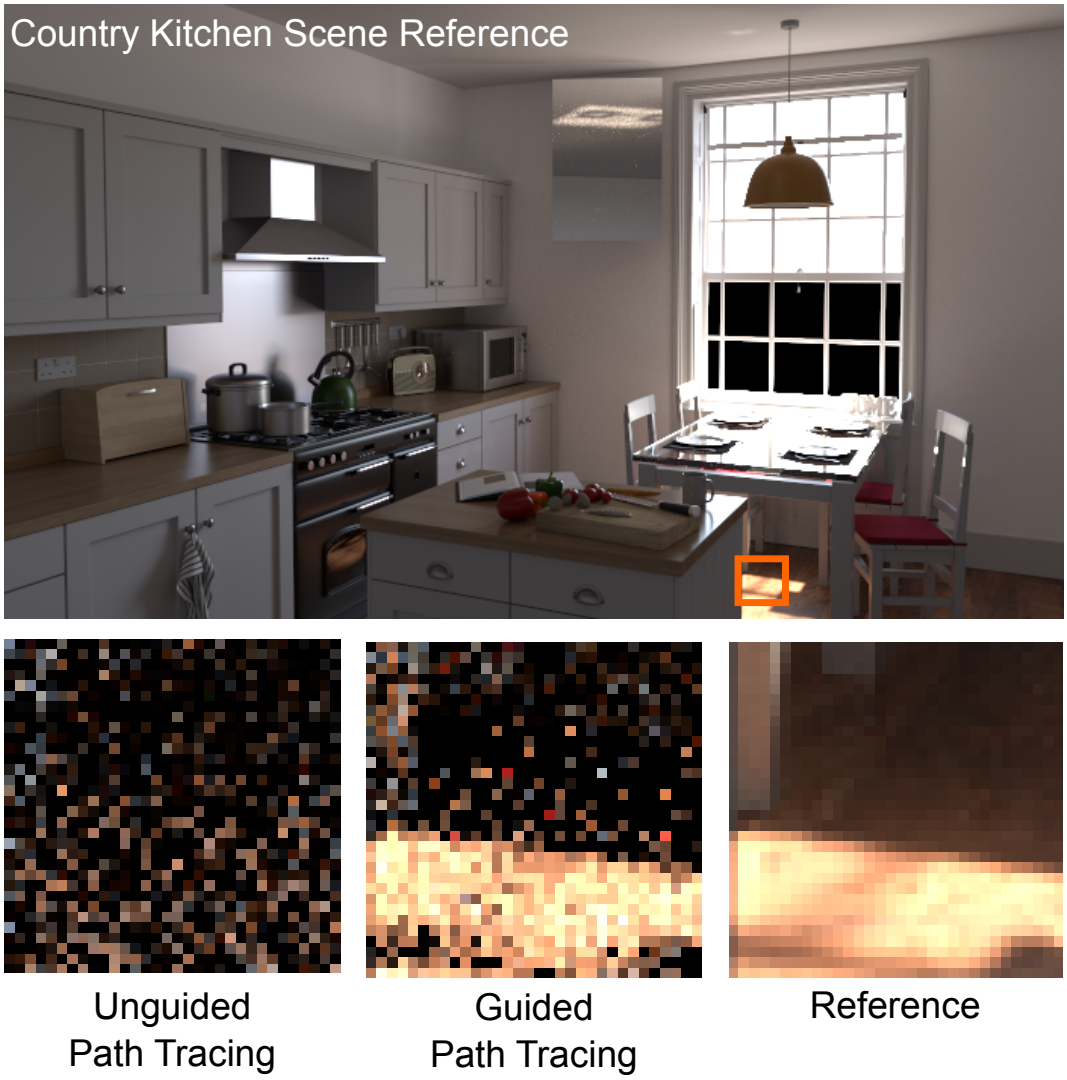
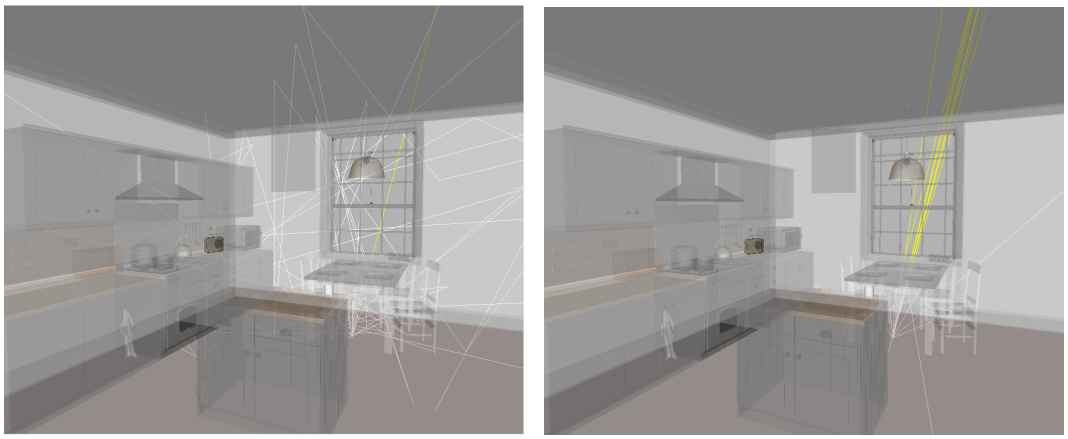


Figure 5.9.: Comparison between un-guided and guided path tracing with equal sample count (16 spp), showing how efficiently light reflections beneath the glass table can be sampled with path guiding.

With the Visual Debugger we are able to visualize two things regarding to path guiding:

1. the effect on guiding in path space

2. the approximations of the incident radiance $p_{L_i}(\vec{\omega}_i|x,\vec{\omega}_o) \propto L_i(x,\vec{\omega}_i)$, all these guiding techniques rely on



(a) unguided path tracing (b) guided path tracing

Figure 5.10.: Comparison between un-guided (left) and guided (right) path tracing. Showing how efficiently path guiding is since paths only receive light contribution by finding a way through the window and hitting the environment map.

The effect on guiding in path space can be shown with the 3D Scene View (Sec. 4.6.3) of the Visual Debugger. For this purpose we selected a pixel beneath the glass table as shown in Fig. 5.9 (marked orange rectangle) to visualize the differences between un-guided and guided traced paths in path space. The scene has only one light source, which is given by an environment map surrounding the scene. Receiving contribution by next event estimation is not possible therefor. In order to obtain light contribution, paths have to find a way through the window. The comparison can be seen in Fig. 5.10. Comparing un-guided and guided traced paths shows how efficient light paths can be sampled in complex scenes to obtain light contribution. In the case of un-guided path tracing, only one path receives contribution by managing it to find a way through the window (marked yellow). All other traced paths bounce within the scene, receiving no contribution. Comparing to a guided path tracing approach more than the half of traced rays manage it to travel through the window and obtain contribution. Consequently less samples are needed for a well converged result. The Visual Debugger provides therefore a great opportunity to show how different path tracing algorithms behave in complex scenes to receive contribution.

Spherical and vMM View. The crucial part of path guiding is to have an accurate approximation of the incident radiance field at each point within the scene in order to determine the direction the path should go to gain light contribution for a faster image convergence. To analyze the approximation we added a custom tool which works similar to the Spherical Incident Radiance View tool (Sec. 4.6.6), but instead displays Mises-Fisher Mixture Models (vMM) [FLE87]. vMM is a probability

distribution which is used to represent incident radiance approximations. This allow us to gain more information about incoming light at an intersection point. The vMM custom tool view sends a request of the current vertex position to the server, which computes the vMM model at the defined intersection point. After calculation, the data is send back to the client, where it is visualized.

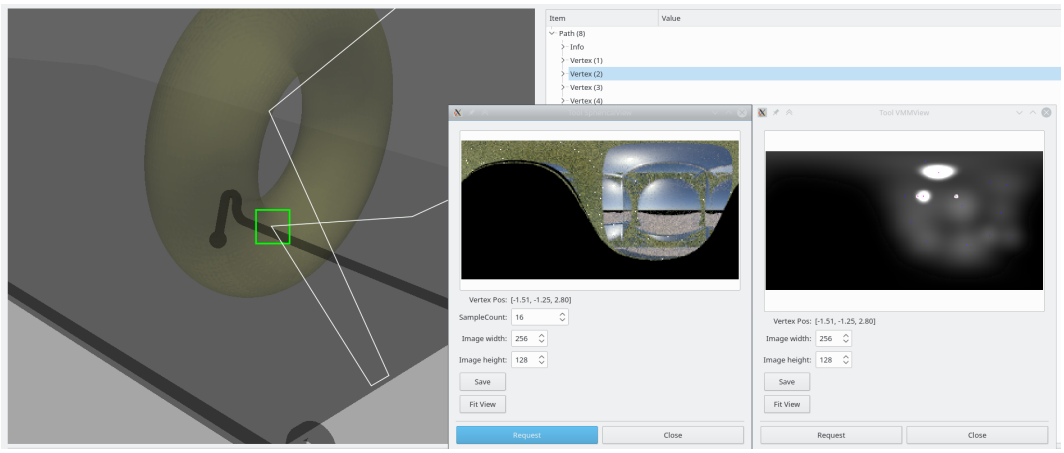


Figure 5.11.: Three dimensional scene view (left) in combination with the spherical (middle) and VMM (right) custom view tools.

The Fig. 5.11 shows the new vMM custom tool view (right) in combination with the Spherical Incident Radiance View (middle) and the 3D Scene View (left). The tool shows the vMM of an intersection point on the torus (marked green). The path intersections can be seen in the 3D Scene View. For each intersection point, Spherical and vMM data can be requested in order to verify the incident radiance approximation. With the combination of the Spherical custom tool and the three-dimensional scene view the user gains a deeper insight from which direction light is emitted. Furthermore, it allows the user to check, if the approximation of the incident radiance matches the optimal one, or if there are still some problem: e.g. bad fits or unsufficient data used for fitting.

5.5. Education

Education is another area that can benefit from the Visual Debugger framework. We adapted the framework to the Nori2 [Jak15] renderer, which is a minimalistic ray tracer. It is widely used at universities to teach students the principles of rendering and the concepts of ray tracing. The ability to analyze and visualize traced paths helps students to gain a deeper understanding of how light-matter interaction is computed. Furthermore, the course instructor also benefit from the interactive visualization by having the opportunity to demonstrate the non intuitive parts of the

rendering algorithms and to address the common pitfalls in their implementation. With the save and load opportunity several problems can be saved and loaded for demonstration and analyzing purpose.

Overall the framework can help both students and developers significantly to understand and analyze certain path tracing algorithms and intern rendering processes.

6. Conclusion and Future Work

With the Visual Debugger we provide a light-weight, interactive visualization framework to analyze physically-based light transport based on Monte Carlo integration. The visualization technique helps to achieve a deeper understanding of the path propagation through rendered scenes. We have shown several use cases which address various path tracing problems and how the Visual Debugger framework can be used to help developers and research scientists to analyze and optimize their approaches. The use of the Seedmap structure shows how the memory consumption can be reduced significantly and how a deterministic rendering state can be achieved in order to allow debugging of such complex systems. The framework proves to be very extensible and ready for analyzing large-scale rendering systems. We have shown how the framework can be extended by the provided custom tool interface to generate and visualize data for the user's needs. With this tool interface new features can be developed and integrated within the existing framework. This extensibility of the framework enables a number of techniques. To summarize, the Visual Debugger framework improves tasks in ray tracing development, education, and analysis by allowing users to visually identify and explore key elements of various rendering algorithms.

We intend to make the code publicly available. This allows extending and improving the framework with new visualization tools. Another further improvement would be to connect the custom tool interface with the three-dimensional render view, allowing a 3D visualization of custom data within the three-dimensional scene space, like we did with the visualization of bounding boxes.

A. Abbreviations

IDE integrated development environment . . . 9
GUI graphical user interface . . . 11
MC Monte Carlo . . . 13
PDF probability density function . . . 13
CDF cumulative distribution function . . . 16
MIS Multiple Importance Sampling . . . 16
RNG random number generator . . . 16
HRNG true hardware random-number generator . . . 17
PRNG pseudo-random number generator . . . 17
TRNG true random number generator . . . 17
DRBG deterministic random bit generator . . . 17
BRDF bidirectional reflectance distribution function . . . 18
API application programming interface . . . 27
MVC Model-View-Controller . . . 34
PBRT Physically Based Rendering . . . 47
FP floating-point . . . 47
vMM Mises-Fisher Mixture Models . . . 56

Bibliography

[APC+16] Abdalla G.M. Ahmed, Hélène Perrier, David Coeurjolly, Victor Ostromoukhov, Jianwei Guo, Dong-Ming Yan, Hui HUANG, and Oliver Deussen. Low-discrepancy blue noise sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 35(6):247:1–247:13, 2016.

[Arv86] James Arvo. Backward ray tracing. In *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, pages 259–263, 1986.

[BNJ15] Benedikt Bitterli, Jan Novák, and Wojciech Jarosz. Portal-masked environment map sampling. *Computer Graphics Forum (Proc. of EGSR)*, 34(4), June 2015.

[EKP88] Glenn E. Krasner and Stephen Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-oriented Programming - JOOP*, 1, 01 1988.

[FHP+18] Luca Fascione, Johannes Hanika, Rob Pieké, Ryusuke Villemin, Christophe Hery, Manuel Gamito, Luke Emrose, and André Mazzone. Path tracing in production. In *ACM SIGGRAPH 2018 Courses*, SIGGRAPH '18, pages 15:1–15:79, New York, NY, USA, 2018. ACM.

[FLE87] N. I. Fisher, T. Lewis, and B. J. J. Embleton. *Statistical analysis of spherical data*. 1987.

[GFE+12] Christiaan Gribble, Jeremy Fisher, Daniel Eby, Ed Quigley, and Gideon Ludwig. Ray tracing visualization toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 71–78, New York, NY, USA, 2012. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Hal64] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, 1964.

[HEV+16] Sebastian Herholz, Oskar Elek, Jiří Vorba, Hendrik Lensch, and Jaroslav Křivánek. Product Importance Sampling for Light Transport Path Guiding. *Computer Graphics Forum*, 2016.

[HL14] Alexander Penev Hristo Lesev. A framework for visual dynamic analysis of ray tracing algorithms. *CYBERNETICS AND INFORMATION TECHNOLOGIES*, 14(2), 2014.

[Ins09] Alfred Inselberg. Parallel coordinates. In *Encyclopedia of Database Systems*, pages 2018–2024. 2009.

[Jak10] Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

[Jak15] Wenzel Jakob. Nori2 minimalistic ray tracer, 2015. https://wjakob.github.io/nori.

[Jef18] Keith Jeffery. Firefly detection with half buffers. In *Proceedings of the 8th Annual Digital Production Symposium*, DigiPro '18, pages 11:1–11:5, New York, NY, USA, 2018. ACM.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[MGN17] Thomas Müller, Markus Gross, and Jan Novák. Practical path guiding for efficient light-transport simulation. *Computer Graphics Forum (Proceedings of EGSR)*, 36(4):91–100, June 2017.

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.

[RKRD12] Tim Reiner, Anton Kaplanyan, Marcel Reinhard, and Carsten Dachsbacher. Selective Inspection and Interactive Visualization of Light Transport in Virtual Scenes. *Computer Graphics Forum*, 2012.

[Ros83] Bernard Rosner. Percentage points for a generalized esd many-outlier procedure. *Technometrics*, 25:165–172, 1983.

[SAH+16] Gerard Simons, Marco Ament, Sebastian Herholz, Carsten Dachsbacher, Martin Eisemann, and Elmar Eisemann. An interactive information visualization approach to physically-based rendering. In *Vision, Modeling & Visualization*. The Eurographics Association, 2016.

[Sii00] Harri Siirtola. Direct manipulation of parallel coordinates. In *CHI '00 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '00, pages 119–120, New York, NY, USA, 2000. ACM.

[SKD12] Colas Schretter, Leif Kobbelt, and Paul-Olivier Dehaye. Golden ratio sequences for low-discrepancy sampling. *Journal of Graphics Tools*, 16, 06 2012.

[SNM+13] Thorsten-Walther Schmidt, Jan Novák, Johannes Meng, Anton S. Kaplanyan, Tim Reiner, Derek Nowrouzezahrai, and Carsten Dachsbacher. Path-space manipulation of physically-based light transport. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2013)*, 32(4):129:1–129:11, July 2013.

[SRB+19] Denis Sumin, Tobias Rittig, Vahid Babaei, Tim Weyrich, Thomas Nindel, Piotr Didyk, Bernd Bickel, Jaroslav Křivánek, Alexander Wilkie, and Karol Myszkowski. Geometry-aware scattering compensation for 3D printing. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 2019. To be published.

[Tka03] Thomas E. Tkacik. A hardware random number generator. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 450–453, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[TM98] C Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. volume 98, pages 839–846, 02 1998.

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[VKv+14] Jiří Vorba, Ondřej Karlík, Martin Šik, Tobias Ritschel, and Jaroslav Křivánek. On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)*, 33(4), aug 2014.

[ZHD18] Tobias Zirr, Johannes Hanika, and Carsten Dachsbacher. Re-weighting firefly samples for improved finite-sample monte carlo estimates. *Comput. Graph. Forum*, 37(6):410–421, 2018.