Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Visual Exploration Of Light Transport In Path Tracing

Giulio Martella

Course of Study:	Computer Science
Examiner:	Jun.-Prof. Dr. Michael Sedlmair
Supervisors:	Dr. Guido Reina Michael Becher, M.Sc. Patrik Gralka, M.Sc.
Commenced:	July 1, 2020
Completed:	January 4, 2021

Abstract

We are presenting a visualization tool capable of showing how the paths shot by a path tracer interact in a scene. The tool let the user explore all, not a subset, the paths that contribute to a final render. It comes with a data gathering C++ library that can be plugged into the code base of existing path tracers and can gather the entirety of the needed data exposing only two methods. The tool provides several utilities to filter paths with intuitive spatial queries and to visualize information on the filtered data. As detailed in this work, performance and memory consumption expediences make the tool usable on consumer-grade calculators. Examples of usage are provided to show how the provided features of the tool combine to help users exploring, debugging and analyzing datasets gathered from a third party path tracer.

Contents

1 Introduction

Rendering is generally the process of generating a two-dimensional image, called *render*, from the mathematical description of a three-dimensional scene. Several rendering algorithms have been developed over the years and many of them have been designed to produce images as *photorealistic* and lifelike as possible. The most recent ones of these kind are all based upon simulating how light interacts with the scene, closely imitating its natural behavior. A particularly successful rendering algorithm following this philosophy is *Path Tracing* [Kaj86] since its derivations and itself are widespread across animation, visual effects and video games. In simple words, path tracing is based upon the idea of shooting rays out of an artificial camera towards the scene and make each bounce around the scene until it reaches a light source; then, the luminous energy carried by the ray and its bounces, called *path*, can be computed — or *traced* back. Without diving further into its details — which are left for section 1.1 —, it is clear how complex path tracing can get. This makes it difficult to grasp by people approaching it and difficult to debug.

We are presenting a tool capable of showing the user an overview of the inner workings of a path tracer by providing interactive visualizations of the very core of a path tracer: paths and how they interact with the scene. We strived to create something that does not tell but — literally — shows the swarm of paths shot by a path tracer to help understand what is actually going on during the rendering process.

The idea of providing interactive renderings of the data generated by a path tracer — or a ray tracer — is not original. Let us mention a few that inspired our work:

- The *Ray tracing visualization toolkit* (*rtVTK*) by C. Gribble et Al. [GFE+12] is a nice starting point but it focuses on ray tracers [CPC84; Whi79] more than on path tracers. Furthermore, it is designed to render only a single ray tree at once, making it difficult to picture a global overview on the data that concurred in the generation of the final render; after all, a good mantra of presenting visual data is, quoting Ben Shneiderman, "Overview first, zoom and filter, then details-on-demand" [Shn03]. rtVTK practically presents only the details.
- The work presented in *A Framework for Visual Dynamic Analysis of Ray Tracing Algorithms* by H. Lesev and A. Penev [LP14] has extensive filtering and data gathering features which inspired us, but it works only on ray tracers, as much as rtVTK.
- Another great inspiration for us was the framework presented in *Applying Visual Analytics to Physically Based Rendering* by G. Simons et Al. [SHP+19], especially the visualization parts. We however drift off it because, while they reduce the data they gather from a path tracer by selecting only about 1/6000th of the total paths, our goal is to keep the whole dataset and let the user choose what to visualize from all the samples.

- Another tightly related work is *EMCA*, which stands for *Explorer of Monte-Carlo based Algorithms*, by C. Kreisl [Kre19]. It focuses on path tracing and visualizations but its client-server architecture with per-pixel path analysis lacks of the global view we are trying to achieve.

1.1 Background

We will now provide a quick overview on path tracing to establish a common ground of terminology between us and the reader. This is not meant to be in any way an exhaustive explanation, please refer to the cited literature to have a better insight.

1.1.1 The rendering equation

As we introduced before, *photorealistic rendering* is the process that given a mathematical description of a three-dimensional scene outputs an as life-like as possible image, know as *render*. The process simulates how light travels from light sources to an abstract camera sensor while interacting with the scene. An important equation that describes how light behaves is the *Rendering Equation* introduced by J. Kajiya in 1986 [Kaj86] here presented with different notations:

$$L_o(\vec{x}, \vec{\omega}_o) = L_e(\vec{x}, \vec{\omega}_o) + L_r(\vec{x}, \vec{\omega}_o) \tag{1.1}$$

It says that, considering a surface, the amount of light that leaves the surface point $\vec{x}$ in direction $\vec{\omega}_o$, called *outgoing radiance* (L_o), is determined by the emitted radiance (L_e), which is the one emanated by the surface itself, plus the reflected radiance (L_r) which is given by:

$$L_r(\vec{x}, \vec{\omega}_o) = \int_{\mathcal{H}^2} f(\vec{\omega}_o, \vec{x}, \vec{\omega}_i) L_i(\vec{x}, \vec{\omega}_i) d\vec{\omega}_i \tag{1.2}$$

This means it consists in the sum of the radiance incoming (L_i) from all possible directions ($\vec{\omega}_i \in \mathcal{H}^2$) hitting point $\vec{x}$, weighted by the reflective and refractive properties of the material ($f(\cdots)$).

Now, image sensors, as well as all animal visual organs, are made of several small photo sensitive units that measure the incoming radiance hitting their surfaces over a short period of time. By putting together the values read by the units an image is produced. To compute the incoming radiance on each of these photo sensitive units called *pixels*, the incoming radiance must be integrated over the pixel surface and scaled by its area to make the result independent of the sensor's size:

$$L_{pixel} = \frac{1}{A_{pixel}} \int_{\vec{q} \in pixel} L_i(\vec{q}, \vec{\omega}_r) dA_q \tag{1.3}$$

Where $\vec{\omega}_r$ depends on the camera mathematical model, that can range from the simplest pinhole camera to a complex simulation of real camera lenses groups.

At this point is clear that rendering is all about computing the radiance hitting a sensor but the rendering equation presented above (eq. 1.1) describes only outgoing radiance. Fortunately, thanks to the fact that radiance is constant along a ray[1], it is possible to write:

$$L_i(\vec{x}, \vec{\omega}_i) = L_o(\vec{y}, \vec{\omega}_o) \tag{1.4}$$

Where $\vec{y}$ is the first point on a surface hit by the ray shot from $\vec{x}$ in direction $\vec{\omega}_i$ and $\vec{\omega}_o$ is the direction pointing from $\vec{y}$ to $\vec{x}$. Introducing the Raytracing RT operator:

$$\vec{y} = RT(\vec{x}, \vec{\omega}_i) \tag{1.5}$$

And rewriting $\vec{\omega}_o$ in function of $\vec{\omega}_i$:

$$\vec{\omega}_o = -\vec{\omega}_i \tag{1.6}$$

We have that:

$$L_i(\vec{x}, \vec{\omega}_i) = L_o(RT(\vec{x}, \vec{\omega}_i), -\vec{\omega}_i) \tag{1.7}$$

The first thing that can be taken from this is that to compute the incoming radiance on a sensor it is sufficient to shoot a ray and compute the radiance on the scene point hit by the ray outgoing in the inverse ray direction. Secondly, plugging everything that has been said until now produces a more complete rendering equation:

$$L_o(\vec{x}, \vec{\omega}_o) = L_e(\vec{x}, \vec{\omega}_o) + \int_{\mathcal{H}^2} f(\vec{\omega}_o, \vec{x}, \vec{\omega}_i) L_o(RT(\vec{x}, \vec{\omega}_i), -\vec{\omega}_i) d\vec{\omega}_i \tag{1.8}$$

1.1.2 Monte Carlo integration

Given all the equations above, all it is needed to render a convincing image is solving them. The first problem that surfaces is that they contain integrals. Integrals are infinitesimal sums and as such they do not conform too well with the discrete nature of electronic calculators. In other words, integrals cannot be numerically solved on computers. Their solution, though, can be approximated. Without lingering on the reasons, a particularly suitable approximation method for our case is the Monte Carlo integration [KW09].

It randomly picks N samples ($x_i, i \in [0, N)$) from the integration interval (Ω), evaluates the integrand ($f(x)$) with each sample and does an average of the evaluated values weighting them by the probability density function ($pdf(x)$) of picking the corresponding sample from the interval. This gives:

$$\int_\Omega f(x)dx \approx \frac{1}{N} \sum_{i=0}^{N} \frac{f(x_i)}{pdf(x_i)} \tag{1.9}$$

The value computed by Monte Carlo gets closer to the correct result increasing the number of random samples N. It has been demonstrated that if the samples are picked from the interval using a uniform distribution, the relative error is directly proportional to $\frac{1}{\sqrt{N}}$. From this it can be deducted

[1] Radiance is constant along a ray only in perfect vacuum. All non-volumetric path tracers — which we exclusively focus on — assume vacuum between surfaces.

that Monte Carlo will never output the correct solution but it will asymptotically tend to it as samples increase; that is why literature usually talk about *convergence* and sometimes about *convergence speed*, which is an informal value expressing how many samples are needed to approximate an acceptable result, where what is considered acceptable varies case by case.

The $\frac{1}{\sqrt{N}}$ factor can be improved by a technique called *importance sampling* that is based upon the idea of drawing samples from a distribution as close as possible to the integrand in order to reduce the variance. The ideal case would be using the integrand itself as distribution but often, due to its complexity, is impossible to draw samples directly from it. Alternative distributions that roughly approximate the integrand are used instead. This can even be brought a step forward by stochastically combining two approximate distributions together using the so-called *multiple importance sampling* (MIS) [VG95].

1.1.3 Path tracing

Now that we have a tool to numerically solve integrals, we can apply it to the integrals presented in section 1.1.1. The total incoming radiance on a pixel (eq. 1.3) can then be calculated by randomly picking sample points on its surface and then applying Monte Carlo (eq. 1.9):

$$L_{pixel} \approx \frac{1}{NA_{pixel}} \sum_{i=0}^{N} \frac{L_i(\vec{q}_i, \vec{\omega}_{ri})}{p(\vec{q}_i)} \tag{1.10}$$

The number of samples picked here N is usually constant for each pixel. Due to its direct relation between final render quality — more samples means a solution closer to the real one — and render time — each additional sample adds computational burden — it is a rather important parameter in all path tracers and it is called *samples per pixel* or *spp* for short.

Now, it is a matter of computing L_i for each pixel sample. Recalling the logical results given by the conservation of radiance along rays presented in equation 1.7 the problem shifts on how to compute the outgoing radiance of the point of the scene hit by the ray shot from the sample position ($\vec{q}_i$) in its associated direction ($\vec{\omega}_{ri}$). This value, as presented by the complete version of the rendering equation (eq. 1.8), depends on another integral having all possible directions as its integration interval. Furthermore, inside the integral there is again an outgoing radiance, but this time computed with other parameters ($L_o(RT(\vec{x}, \vec{\omega}_i), -\vec{\omega}_i)$). By substitution, it begins to become clear that evaluating one outgoing radiance theoretically leads to solving a recursively infinite amount of integrals. This, through the power of Monte Carlo integration, becomes manageable and a path tracer computes outgoing radiances by sampling all the possible directions just once. The rendering equation can then be written as:

$$L_o(\vec{x}, \vec{\omega}_o) \approx L_e(\vec{x}, \vec{\omega}_o) + \frac{1}{pdf(\vec{\omega}_i)} f(\vec{\omega}_o, \vec{x}, \vec{\omega}_i) L_o(RT(\vec{x}, \vec{\omega}_i), -\vec{\omega}_i) \tag{1.11}$$

With this approximation the calculations consist in shooting a ray, find an intersection with the scene, compute the properties of the material on the intersection, pick a random new direction, shoot a ray in that direction from the intersection point and then keep doing the same recursively, forming a path. In practice, to not make a path bounce indefinitely, a maximum number of bounces, called *maximum path length*, is arbitrarily decided before rendering, as much as the number of samples per pixel is.

Path tracers that behave as described until now are informally called naïve path tracers. Other more complex tracers employ techniques to speed up the Monte Carlo convergence. As already mentioned, a common technique is importance sampling, as it can be applied to path tracing by sampling new bounces' directions from probability distributions that naturally carry more radiance. This can happen in two ways: either paths are forced to follow the properties of the material they are bouncing on, or they are directed towards a light source. A usually even more efficient approach is to combine these two sampling strategies through multiple importance sampling.

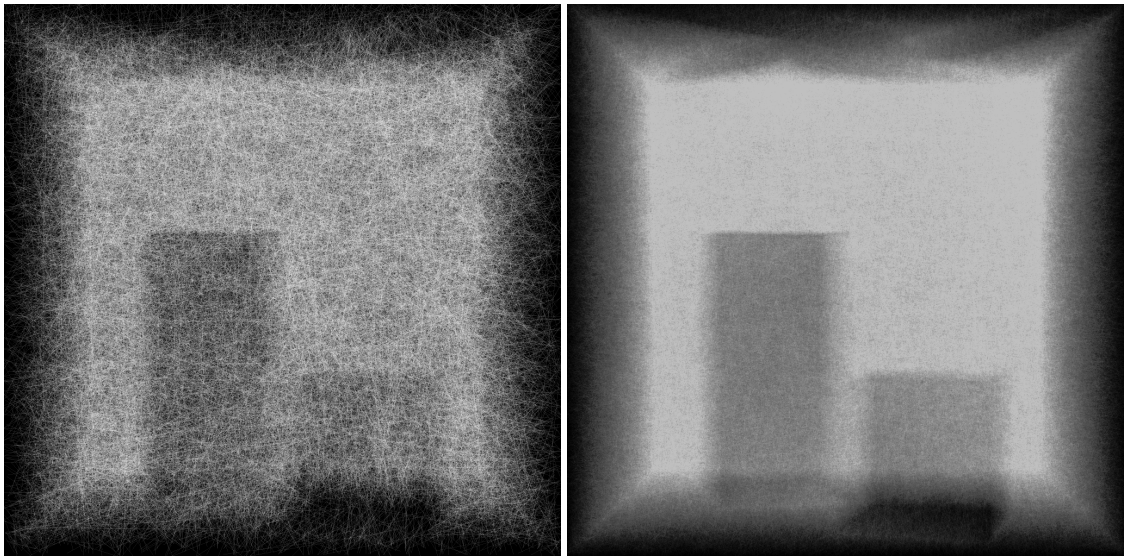
1.2 Motivation

As introduced above, path tracing is not easy to picture. As many research groups before us thought, having a tool showing what a tracer is doing by visualizing its interactions with the 3D scene would help the professional developers lost in the debugging cycle as well as the students trying to get a hang of path tracing's inner workings. The challenge this vision has to face is the sheer amount of these interactions; talking about path tracing, to have a decent render[2], more than two hundreds paths have to be shot for each pixel: to produce an image, more than fifty million paths have to be shot. Postponing data size considerations for later (sec. 2.2.1), leaves us with potential datasets that without proper data reduction or filtering are of no use to anyone. Related works usually perform substantial reductions of the datasets [Kre19; SHP+19], but we believe that having all the paths available for visualization would improve the experience.

With these premises, a more precise idea for a visualization tool came to us. The user should have the power to visually and interactively explore all the paths previously generated by a path tracer during rendering. The paths have to be immersed in an interactive rendering of the scene: their most important property is their geometry or, in other words, where they bounce inside the scene. In the beginning, out of curiosity and naivety, we tried visualizing all the paths of some datasets and even if those were rather small, we achieved compelling yet cluttered results (fig. 1.1); this made visualizing the millions of paths of a proper dataset out of question from the earliest stages. To navigate this ocean of paths, the user should be provided with a global summary overview of the dataset in its entirety, and they should also able to filter them using spatial queries. For the global overview we envisioned a heatmap applied as a texture to the 3D view of the scene that shows areas where there is more path activity, while for the spatial queries we thought the user might want to select path bouncing on certain noteworthy surfaces, such as lights or mirrors.

Later in the development, it became clear that the possibility of comparing different datasets would come in handy to users. It could be useful in many cases such as comparing two progressive versions of the same in-development path tracer to understand what changed or, more in an educational context, such as showing a ground truth next to a purposely wrong dataset to understand why one is wrong on a deeper level.

[2] We considered as a decent render a 512×512 pixels and 256 spp image which is a good halfway point between production quality and being way too far from convergence.



(a) 64×64, 1 spp **(b)** 64×64, 64 spp

Figure 1.1: Visual clutter in an early version of the tool where the scene rendering has not been implemented yet. Both datasets have been generated on the *Cornell Box* scene.

Of course, before even talking about visualizing data, data has to be gathered from a path tracer. We envisioned a software, parallel to the visualization client, able to plug to any path tracer and gather with little effort the required data during the rendering process. Then the idea of a gathering library with a very simple interface came about; a library that can be plugged into any path tracer the user is currently working on by just calling the few required functions.

To summarize, our vision consisted in a tool divided into a visualization client and a data gathering library which can be used both for debugging and teaching purposes. The library should have an easy to interact with interface, while the client would let the user explore different datasets simultaneously and in their entirety using the filtering and visualization options provided to them.

2 The tool

The tool has two components: a *data gatherer* and a *visualization client*.

The data gatherer is a C++ header-only class of which methods have to be called within the main loop of any unidirectional path tracer. It stores to disk data generated — and usually discarded — by the path tracer during rendering. For each path shoot by the renderer the tool stores its transported radiance, its camera sample, and the world positions of its bounces.

Once the rendering is done and the data is collected, the visualization client can be used to explore the gathered information. To allow interactive 3D exploration it also needs a scene description; this is not generated by the gatherer and it has to be provided separately by the user. The scene format used is novel, based on triangle meshes and it shares many similarities with glTF [RAPC14]. This peculiar choice has been done mainly to not do any assumptions on the kind of geometry representation the path tracer uses. In this way even path tracers that use exotic geometries can be analyzed with the tool — at least until the user is able to convert whatever they are using to triangles.

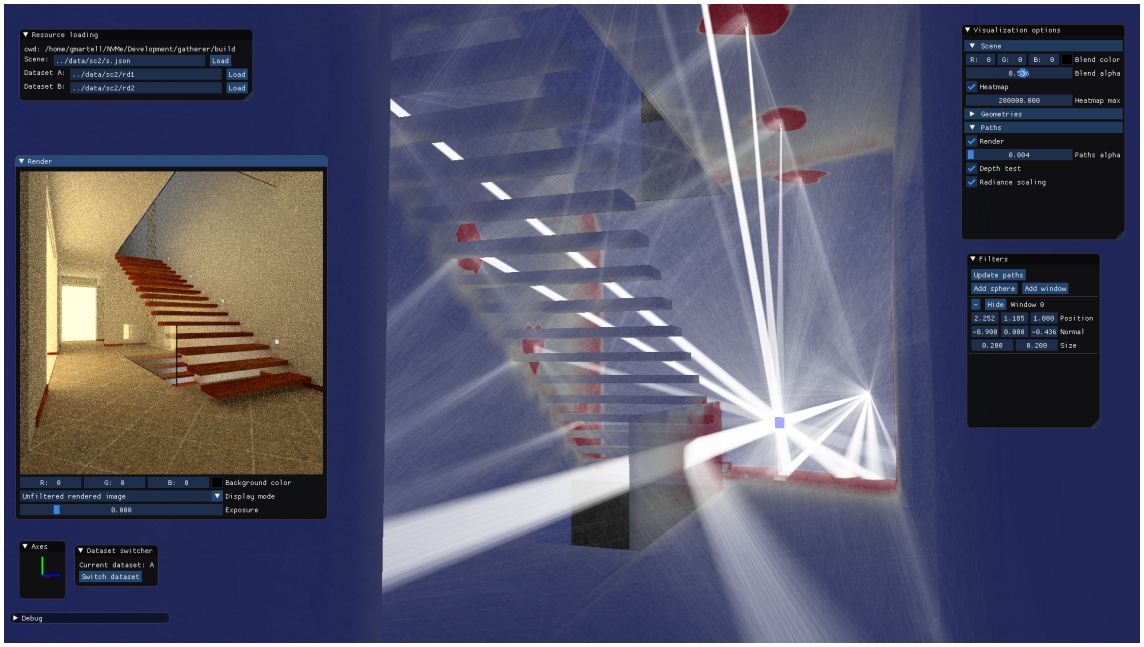


Figure 2.1: A full screenshot of the visualization client. A dataset generated by Yocto/GL [PNC19] upon the *Modern Hall* scene [Bit16] is loaded.

2.1 Technology

Before talking about the actual implementations, here is presented a summary of the technologies used to create the tool.

The data gatherer has been developed as a header-only C++17[1] library. The choice was motivated by the belief — not backed by any data — that most path tracers, included the ones written by students, are written in C++. In hindsight, probably writing it in pure C would have made it potentially compatible with more path tracers; after all it uses very few features of C++.

Deciding the technologies to use for the visualization client took more careful thinking, since it was clear it would be the biggest software piece in the context of this project. The final combination consists in:

C++ Due to the available expertise, the other plausible languages were Python and — for a very brief moment — Julia, but they were discarded due to the necessity of managing memory and threads directly. C has been considered too, but STL and OOP made the scale tip over C++. As for the data gatherer, C++17 is used.

"Raw" OpenGL Rendering hundreds of thousands of paths in an interactive and fully customizable fashion was perceived since the beginning as no easy task and it made clear the necessity to be able to control and program every possible stage of the rendering pipeline. Due to this, visualization frameworks such as VTK [SML06] had been ruled out immediately in favor of just GLEW and GLFW3 to provide an OpenGL 4.6 Core Profile context.

Dear ImGui Since all rendering customization needs came from the interactive 3D rendering of paths and scenes, using just OpenGL to build a GUI from scratch would have been out of the scope and so a GUI framework had to be chosen. The final candidates were Qt5 and Dear ImGui[2] but since ImGui was not completely familiar to the author, it has been chosen due to educational purposes.

Boost Log and Filesystem It seemed like a good idea to have a complete logger and a filesystem library with a STL-like style and it led to the adoption of the Boost Log[3] and Boost Filesystem[4] libraries. Since it is not a crucial dependency, it might be removed in a future development.

Both the gatherer and the visualization client use a header-only math library called *math.hpp*[5] to handle vector and matrix types and operations. It includes the *half* library[6] which is an implementation of the 16 bits floating point number representation, commonly called half precision float or just `half`, as described by the IEEE 754 standard[7].

[1] C++17 is the informal name of the ISO/IEC 14882:2017 standard. https://www.iso.org/standard/68564.html
[2] https://github.com/ocornut/imgui
[3] https://www.boost.org/doc/libs/1_75_0/libs/log/doc/html/index.html
[4] https://www.boost.org/doc/libs/1_39_0/libs/filesystem/doc/index.htm
[5] https://github.com/giuliom95/math
[6] http://half.sourceforge.net/
[7] https://ieeexplore.ieee.org/document/8766229

Figure 2.2: *Modern Hall* scene [Bit16] rendered by Yocto/GL [PNC19]. It is a 512x512, 512spp image; maximum bounces per path are set to 10.

2.2 Data gatherer

2.2.1 The data to gather

Until now, it has been said that the gatherer has to be able to store "the paths with some useful data", but how does one store a path and what is useful data have not been defined yet. In the beginning, the focus was on just visualizing paths in an interactive scene rendering so that the user could see how paths interact with the scene. To achieve that it was clear the actual geometry of all paths had to be stored. This meant storing for each path a camera sample and all of its bounces. Several ways of storing path bounces had been considered, such as storing each bounce as a direction and a distance, but we opted for the most naïve solution: each bounce is stored as a three-dimensional point in world space.

This immediately raised some concerns about how much memory would be required to store all these points: let us calculate how many bytes it would take to just store the bounces of all paths needed to produce image 2.2. It is 512 pixels wide and 512 pixels tall, it has been rendered with 512 samples per pixel — spp — each with a depth of maximum 10 — this means that each of the 512 paths shot every pixel bounced in the scene at most 10 times. Storing the points as a triplet of single precision float numbers — 4 bytes, the C++ built-in `float` type — we would have:

$$(512 \times 512)pixels \times 512spp \times 10bounces \times 3dimensions \times 4bytes \approx 15GB \tag{2.1}$$

As said just above, the paths in this case bounce at most 10 times: most paths are terminated way before either when they bounce out of the scene or — in some path tracers — they hit a light. This leads to the need of an additional buffer storing the amount of bounces each path has. Put with computer graphics terminology we might say the points positions buffer represents the *geometry* of

the paths while this "path lengths" buffer is the *topology*. This new buffer needs to contain integers and, considering that the standard C++ `int` built-in type on x86_64 machines is 4 bytes long, we would have that the size of the buffer in the context of figure 2.2 would approximately be:

$$(512 \times 512)pixels \times 512spp \times 4bytes \approx 500MB \tag{2.2}$$

Now this is not an unmanageable amount of data, but since the tool has to be usable on consumer grade machines, reducing those number has been one of the main priorities during the early stages of development. The most effective change that has been made is to simply use smaller primitive data types. Single precision floats had been almost immediately replaced with half-precision floating point numbers, effectively halving the size of the world positions buffers. Using the same assumptions made for equation 2.1, we would have 7.5GB instead of 15GB. Halving precision, beside a memory footprint reduction, comes with a reduction of actual precision and potential noticeable rounding errors; but since these numbers will mostly be directly used for visualization purposes with very little preprocessing, precision errors should not negatively impact the user experience. Considering now the paths lengths buffer, in most reasonable cases, 1 byte per path would suffice: 255 bounces are more than enough for most cases. Examining equation 2.2, after the reduction from 4 bytes to 1 that buffer occupies approximately 128MB. Similar reasoning has been applied to all other buffers: every floating point number is half precision and every integer is made as small as is reasonable.

To complete the spatial representation of the paths, an origin point, which is semantically not a bounce, was needed. For the sake of simplicity, an additional fake bounce lying on the render camera eye was added to each path. Due to its wasteful nature — it adds 6 bytes per path which would accumulate to $512 \times 512 \times 512 \times 6 \approx 750MB$ in the case of figure 2.2 —, this solution was hastily set aside; in the final version of the tool, the origin point of the paths is not stored in the dataset all together. The camera eye point that comes with the scene description — see subsection 2.3.6 — is implicitly considered the origin of all paths.

To fuel some components of the visualization client that have been developed over time, two other buffers are built by `gatherer`. One contains the camera samples and the other the radiance carried by each path. A camera sample is the correct position of a path on the virtual image sensor. The radiance of each path is just stored as a triplet of floats indicating the radiance carried by a path for the red, green and blue color components.

Focusing on implementation details it is clear that four binary data buffers have to be stored on disk. They could all be cramped into a single file but to make the loading code simpler, it has been decided to store each buffer in its own separate file. This means that a full dataset has to be stored into a folder rather than a file. The folder, usually, but not necessarily, called `renderdata` contains two subfolders, `bounces` and `paths`. This was made to logically separate buffers that contains data proper to entire paths — like the lengths and the radiances — to the ones relative to bounces — the bounces positions. It may seem a little overzealous but it was made to support an easy and organized implementation of new buffers.

At the end of the day, there are 4 binary files:

`bounces/positions.bin` The world positions of all bounces of each path, stored as triplets of `half`.

`paths/lengths.bin` The number of bounces of each path — almost always called *path length* in the code —, each stored as a single `uint8_t`.

`paths/radiance.bin` The radiance carried by each path, stored as triplets of `half`.

`paths/camerasamples.bin` The sample position on the film plane of each path. Each sample is stored in a `CameraSample` data structure[8], which contains:

- A couple of `uint16_t` indicating which pixel of the render image the sample belongs to.
- A couple of `half` $\in [0, 1]$ representing the position of the sample relative to the pixel, where $(0, 0)$ is the upper left corner of the pixel and $(1, 1)$ is the lower right one.

2.2.2 `Gatherer` class

All data gathering is managed by a header-only library that exposes one single class called `Gatherer`. To correctly gather data from a path tracer the user must initialize an instance of `Gatherer` at any point before the main loop and then call its methods where appropriate while tracing paths. Signatures of the constructor and the methods of `Gatherer` is presented in Listing 2.1. The constructor needs a folder path where the data will be stored and the number of threads — `nthreads` — the path tracer will run on. The `addbounce` method has to be called every time a path bounces and the position of the bounce is final. It has that position in world coordinates as an argument alongside a *thread id*, an integer that goes from zero to `nthreads` and unequivocally identifies a thread. When all the computations on a path are done and the radiance carried by it is known, the user has to call the `finalizepath` method passing the thread id, the radiance and a `CameraSample` struct — see Listing 2.2 for its layout.

As hinted by the `nthreads` and `tid` arguments of `Gatherer` methods, the class has been designed to be able to gather data from path tracers that evaluate paths on multiple threads parallelly. This extra effort has been made to not force users to use only single-threaded path tracers. It has to be noted though that the `Gatherer` class assumes each path is entirely resolved on just one thread: splitting the computations of a single path on multiple threads is not currently supported.

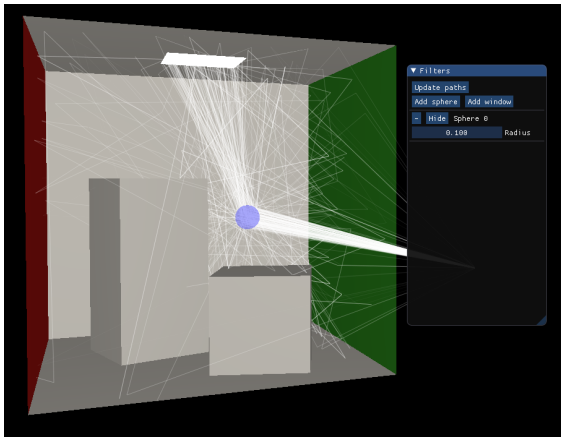
Datasets gathered from the rendering of high quality scenes can get big fast. To avoid any memory saturation problems and to leave as many resources as possible to the path tracing computations, `Gatherer` allocates a fixed amount of memory per thread and will use only that; it will store data on disk when any of these memory chunks are about to fill completely. In order to write to disk without introducing any software barrier in the multi thread code, each thread writes on a set of files written exclusively by it. The destructor of `Gatherer` takes care of stitching all these file sets into one, being extra careful to keep the indices consistent. This is a fully disk-bound operation and it may take a while, especially on rather big dataset.

Listing 2.1 Simplified `Gatherer` class definition with everything a user needs to gather data successfully.

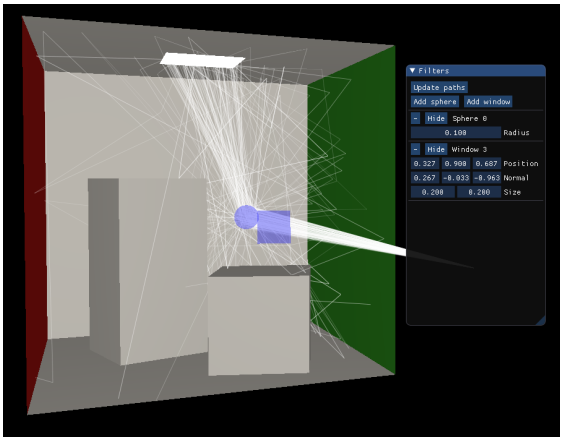
```
class Gatherer
{
public:
  Gatherer
  (
    unsigned nthreads,
    const std::filesystem::path& folder
  );

  void addbounce
  (
    unsigned tid,
    Vec3h pos
  );

  void finalizepath
  (
    unsigned tid,
    Vec3h radiance,
    CameraSample sample
  );
}
```

Listing 2.2 Data structures used inside the `Gatherer` class. The `half` type contains a 16bit precision floating point number complying to the IEEE 754 standard.

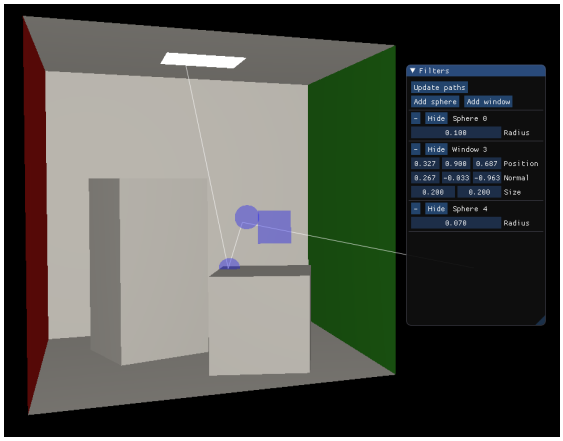
```
using Vec3h = std::array<half, 3>;

class CameraSample
{
public:
  uint16_t i, j;
  half u, v;
};
```



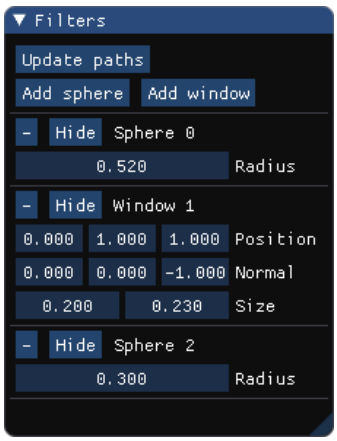
(a) In the beginning, one sphere is placed on the back wall.



(b) Then a window is placed to select only the paths that go from the camera to the sphere on the back wall.



(c) Finally, a second sphere is placed on top of the short box to select only a single path.



(d) "Filters" floating panel

Figure 2.3: (**a**, **b**, **c**) Example of using more path filters together in a small dataset generated on the Cornell Box scene of 64×64 pixels and 32 spp.
(**d**) Screenshot of the **"Filters"** floating panel with two sphere filters and a window filter.

2.3 Visualization client

2.3.1 Path filters

As said already, visualizing all paths at once is both useless and impossible in most cases. User can filter paths so they can focus only on the scene portions they are interested in. Two filters types are provided: the *sphere filter* and the *window filter*. The former is a sphere that has to be placed on a scene surface and selects the paths bouncing inside its radius, the latter is a rectangular window that selects the paths passing through it.

As shown in figure 2.3, combination of filters of both kinds can be used together. Considering all paths as a mathematical set of distinct paths $\mathcal{S}$, a filter F creates a subset $F(\mathcal{S}) \subseteq \mathcal{S}$, and combining the filters $F_1, F_2, ..., F_n$ gives the intersection of their relative subsets:

$$F_1(\mathcal{S}) \cap F_2(\mathcal{S}) \cap ... \cap F_n(\mathcal{S}) \subseteq \mathcal{S} \tag{2.3}$$

Due to this strong affinity to mathematical sets, the initial code handling the filters' combination was mostly based on the `std::set<T>` data structure of C++ standard library. It was set aside when it turned out to be significantly slower than just storing everything into `std::vector<T>` structures and performing set intersections with the `std::set_intersection` function from the `<algorithm>` STL library. To further speed up the filtering, the filters are computed in order so they all need to just test the paths selected by the previous filter; in the implementation, equation 2.3 is practically evaluated as if it was:

$$F_n(...F_2(F_1(\mathcal{S}))) \subseteq \mathcal{S} \tag{2.4}$$

The tool uses a `std::vector<unsigned>` called `selectedpaths` to keep the indexes of the currently selected paths. Each filtering step is practically run over the `selectedpaths` output by the last stage. Before any filter, `selectedpaths` is filled with all path indexes. Forcing this into mathematical notation gives that the ith filter step out of m filters is:

$$\texttt{selectedpaths}_i = F_i(\texttt{selectedpaths}_{i-1}),\ i \in [1, m] \tag{2.5}$$

where $\texttt{selectedpaths}_i$ is the `selectedpaths` vector after filtering through the ith filter and $\texttt{selectedpaths}_0 = \mathcal{S}$.

Users can add, modify and delete filters from the **"Filters"** floating panel (fig. 2.3d). On the top of it there are three push buttons:

"Update paths" Since path filtering is an operation that takes a noticeable time to complete, the user must explicitly express the will to apply the current filter set to the paths.

"Add sphere" After the user clicks this, they have to click on a surface of the scene. A filter sphere will appear on the clicked surface.

"Add window" Same as the previous button but it adds a window filter on the clicked scene surface instead.

[8] See listing 2.2 for the actual C++ definition.

Filters currently present in the scene will appear right below these buttons. Each filter has an entry and all of these will have a delete button — marked with a minus **"-"** sign —, a **"Hide"/"Show"** button to toggle the filter visibility in the viewport, and a label with the filter name. Then, according to their filter type, there will be sliders controlling miscellaneous attributes.

Sphere filter

This was the first filter added. It has a spherical shape and simply selects all the paths having at least one bounce inside its volume. Users can place a sphere filter on clicking on any scene surface on the viewport after having clicked the **"Add Sphere"** button at the top of the **"Filters"** panel. Once a sphere filter has been added, its radius can be controlled by the slider in its slot on the panel. The default radius is one tenth of the scene's largest dimension. One possible future improvement would be giving the possibility to translate the sphere after it has been initially created; might this be achieved through 3D gizmos or just by dragging, every step in that direction should help the usability.

The actual path filtering is plainly done by computing the squared distance between each bounce of each path and the sphere center. For each bounce with a computed distance less than the sphere radius, its entire path gets flagged as selected. This allows for a path-wise early termination: if a path has been flagged, it is useless to check the remaining bounces of the same path. Early termination or not, this computation is rather heavy since a render dataset rarely goes under the one hundred million bounces mark. To mitigate the otherwise lengthy waiting times, the computation is done on multiple threads; The set of paths that has to be filtered is divided in n subsets $S_{1...n}$ of roughly the same cardinality, where n is the number of maximum thread concurrency available on the machine. Each subset is coupled with a `std::vector<unsigned>` that has to hold the indexes of the paths of the subset that satisfy the filter; to avoid memory allocations, these vectors reserve $|S_{1...n}|$ slots each before any computation. Then, n threads start to apply the filter to their assigned subset. Once all of them are done, the content of the vectors — which are the indexes of the selected paths by each thread — are moved into `selectedpaths`, the STL vector keeping the index of all selected paths.

On the viewport the spheres are rendered without passing any vertex buffer to the GPU: first, using hard-coded coordinates and the GLSL `gl_VertexID` built-in variable, a cube of 2 units long side centered on the origin is generated by the vertex shader; then through tessellation new vertices are generated — every patch level has been set to 3 — and then displaced on the surface of a unit sphere. Finally scaled and transformed at the end of the tessellation evaluation shader, the resulting triangles are ready to rasterize.

Window filter

The window filter is a rectangle embedded in 3D space that selects all the paths passing through it. After clicking on **"Add Window"** button, user can spawn a window filter by clicking on a surface; a window facing the camera with its central point lying on the clicked surface will be created. The filter can then be translated, rotated and scaled through the many sliders available in the filter slot on the **"Filters"** panel. The current interface is rather difficult to use and many things can be done to improve it, such as 3D gizmos, as already suggested for the sphere filter.

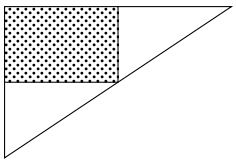


Figure 2.4: The triangle encompassing a window filter (dotted). It is just a right triangle of which right angle lies on a window vertex and of which catheti are as long as twice the window sides.

The filtering is performed by iterating over the paths and for each test if any of its rays intersect the window. Even this filter supports the same kind of path-wise early termination presented for the sphere filter. To perform the ray-window intersection test, a slightly modified version of the Möller-Trumbore algorithm [MT97] has been employed: each ray, instead of being tested against the two co-planar triangles that make up the window, is actually tested against a single large triangle encompassing the window, like shown in figure 2.4; then, rays intersecting the triangle on points with any of the barycentric coordinates relative to the hypotenuse's vertices greater than 0.5 are discarded.

2.3.2 Viewport

The three-dimensional viewport is the central component of the visualization tool. It takes care of interactively rendering three entities: the scene geometry, the paths' rays and the filters shapes. Each presented their own challenges that culminated in the implementation of their cross interactions. Having these many elements to show brought the necessity of employing multi-pass rendering, compositing techniques throughout the render pipeline and consequent multiple frame buffers. To save resources and keep the UI snappy even when the viewport requires most GPU resources, the viewport renders only when a change of any visualization parameter or scene/dataset properties happens. For this purpose, a boolean variable called `mustrenderviewport` is used and has to be set true to require a viewport redraw. The render pipeline and the interface controls provided to the user to tweak its stages will now be explained analyzing the steps made to render figure 2.5.

The scene is rendered in two passes, one for the opaque geometries and the other for translucent ones. The opaque geometries pass is done first and outputs a *beauty* texture, the *world positions* texture, the *geometry ID* texture, and a *depth* texture. The beauty texture (fig. 2.6a) is a color render of the scene lighted by a point light placed on the camera. Every surface is flat shaded with a plain Lambertian model [Lam60]. As detailed in section 2.3.6, the scene format does not store the vertex normals, instead the surface normals are computed on the fragment shader using the `dFdx` and `dFdy` GLSL functions on the world position of the fragments. As their names suggest, the world position, geometry ID and depth textures (fig. 2.6b, 2.6c and 2.6d) respectively store the world position, geometry ID and the depth — which is distance from the camera $\in [0, 1]$, where 0 is on the near clipping plane and 1 is on the far clipping plane — of each fragment. Having depth data might seem redundant when the world positions are available, but to take advantage of the built-in depth testing functionalities of OpenGL it has been preferred to keep both textures. The world positions and geometry IDs are used to get information on the points clicked by the user; these are then used for example, when placing filters or selecting scene objects to hide.

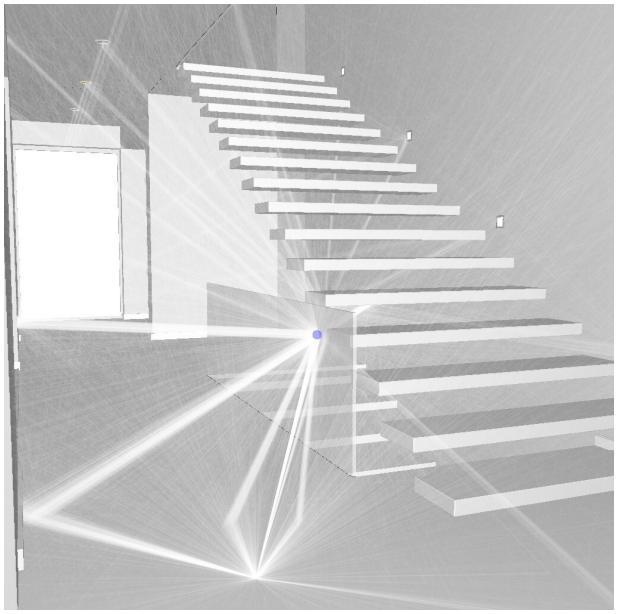
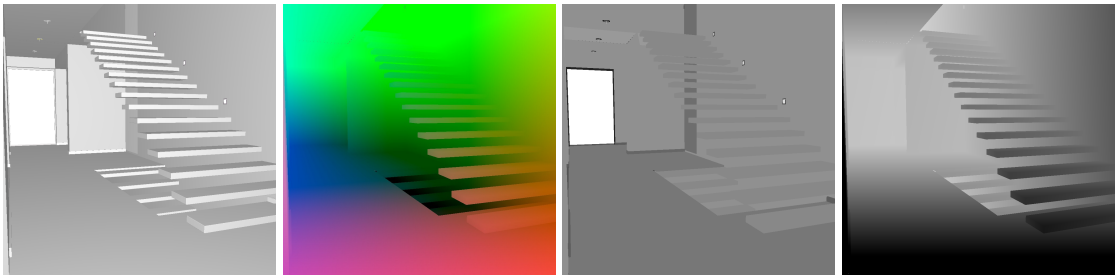


Figure 2.5: An example of a final frame rendered by the viewport.



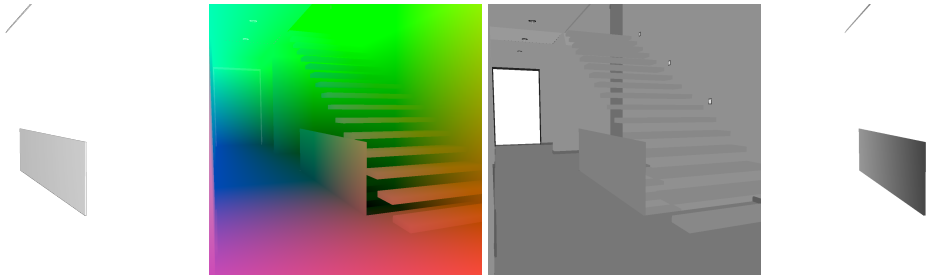
(a) Beauty **(b)** World positions **(c)** Geometry ID **(d)** Depth

Figure 2.6: Textures written by the opaque scene pass. The white and black levels of (b), (c) and (d) have been altered to improve readability.

The pass for translucent objects outputs beauty (fig. 2.7a), world positions (fig. 2.7b), geometry IDs (fig. 2.7c), and depth (fig. 2.7d) as much as the previous pass does. Here though, while beauty and depth are written on new textures, the world positions and the geometries IDs are written upon the ones that came from the opaque pass (fig. 2.6b and 2.6c). Depth testing is enabled for this pass and it is performed against the depth texture of the opaque pass (fig. 2.6d); translucent objects behind opaque ones are simply not visible.

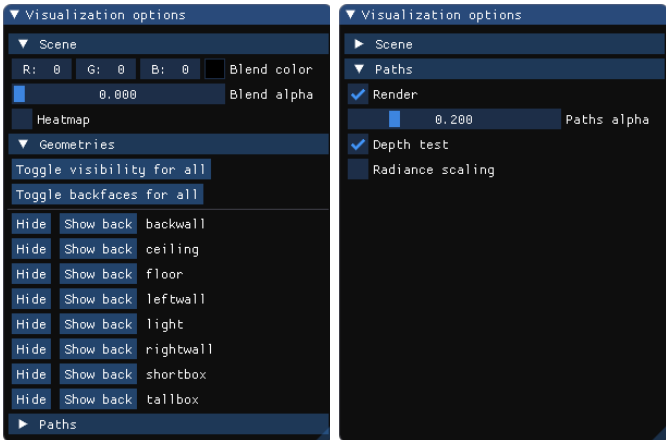
To facilitate scene navigation and readability, some visualization options are provided under the **"Scene"** section of the **"Visualization options"** panel (fig. 2.8a):

"Blend color"/"Blend alpha" They perform alpha blending of the fragments of the scene with the user specified color; it comes in handy when a scene has many bright-colored surfaces which can make paths difficult to see. A visual example is provided in figure 2.9.



(a) Beauty **(b)** World positions **(c)** Geometry ID **(d)** Depth

Figure 2.7: Textures written by the translucent scene pass. The white and black levels of (b), (c) and (d) have been altered to improve readability.



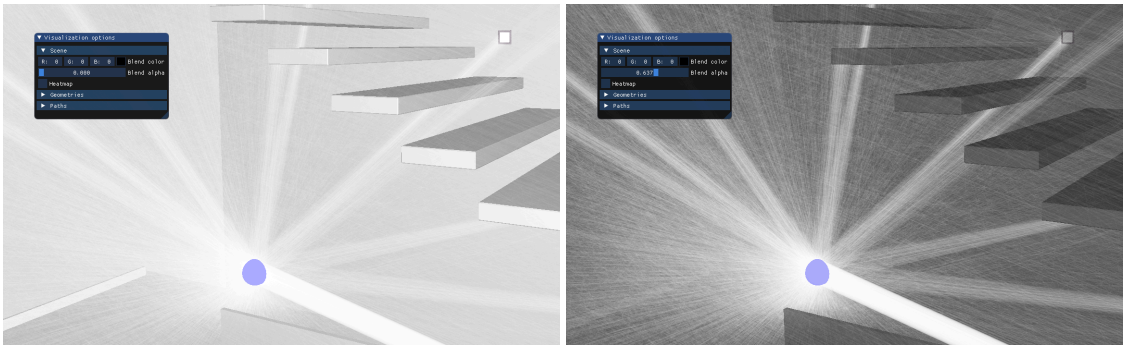
(a) "Scene" section **(b) "Paths"** section

Figure 2.8: Screenshots of the **"Visualization options"** panel showing its two sections while the *Cornell box* scene is loaded.

"Geometries" This collapsible section lets the user toggle visibility and back face culling for either single geometries or all of them together. Users can click on a geometry from the viewport to select it. A selected geometry will be rendered in yellow by the viewport and its name on the **"Geometries"** list will be highlighted. Geometries can be selected from the list too.

"Heatmap" It enables the bounce density heatmap rendering, feature described extensively in section 2.13.

Right after the scene passes, the paths are rendered. Every time there is a change of the set of the selected paths, the new set is loaded on the GPU. The rendering is done using the `glMultiDrawArrays` OpenGL render function with the `GL_LINE_STRIP` render mode. This has been proven to be visibly faster than calling `glDrawArrays` several times inside a `for` loop. Through the use of the two depth textures generated by the scene render passes (fig. 2.6d and 2.7d), the path fragments are rendered on two textures: a *front* (fig. 2.10a) and a *back* (fig. 2.10b) one. All the fragments that are occluded by neither opaque nor translucent scene geometry end up in the front texture, the ones occluded only by a translucent geometry fill the back texture. Since usually hundreds of thousands of paths pass through a single pixel, the OpenGL built-in alpha blending is enabled. By default, all paths are



(a) Alpha = 0 **(b)** Alpha = 0.637

Figure 2.9: Use of the blend color visualization option to improve path visibility in a bright scene.

rasterized with an alpha value directly controlled by the **"Paths alpha"** slider on the **"Paths"** section of the **"Visualization options"** panel (fig. 2.8b). In that section there are also three checkboxes controlling other visual properties of the paths:

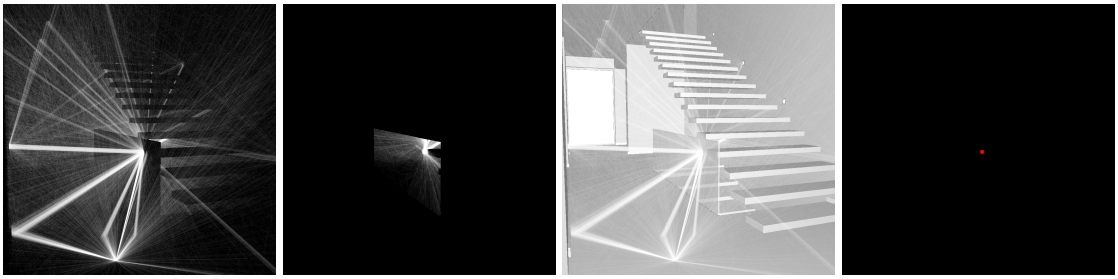
"Render" Toggles the rendering of the paths all together. It might be helpful in situations where the user selected too many paths and moving the viewport camera gets difficult.

"Depth test" Toggles the depth test during paths' rendering. It can be handy with cluttered scenes.

"Radiance scaling" When checked, the paths' alpha gets scaled by their transported radiance. In other words, it gives a visual method to pick the rays that carry more light energy than others. This can be used to detect paths carrying an abnormal amount of radiance which could be the source of many rendering artifacts such as fireflies[9]. From the implementation point of view, this is achieved by loading to the GPU a shader storage buffer — `ssbo` for short — containing the summed radiance of the red, green and blue channels of each path; the shaders then, thanks to the `gl_DrawID` variable — which is available since the `glMultiDrawArrays` render function is used —, can access the `ssbo` correctly and shade the paths by multiplying the radiance with the user defined path alpha.

Reached this point, what the viewport rendered until now is composited together. Starting from the opaque beauty texture (fig. 2.6a), the *back* paths are added (fig. 2.10b), then the translucent beauty texture (fig. 2.7a) is alpha blended over with a default value of 0.7 and finally the *front* paths (fig. 2.10a) are super imposed. This leads to the result shown in figure 2.10c. As probably already noticed by the reader, this render pipeline provides a rather simplistic way to solve the so often called "order independent transparency" problem that breaks when there are paths that has to be rendered between two translucent objects: they will render as they were behind both objects. Due to the lesser severity of the resulting artifacts, a proper and technically more complex solution has never been even planned.

[9] In Monte Carlo based rendering, a firefly is a pixel with an abnormally high total radiance. They usually happen on caustics effects or near light sources placed too close to surfaces.



(a) *Front* paths **(b)** *Back* paths **(c)** Scene + paths **(d)** Filter mask

Figure 2.10: Textures written by the paths pass with the composited result of the scene renders plus the paths and the filter mask texture.

To make filters appear over the paths, they are rendered last. They are first rendered as a mask (fig. 2.10d), taking in account only the opaque depth texture (fig. 2.6d) so that filters appear occluded by opaque object scenes. The mask is then used to composite the *scene* + *paths* with a semi-transparent dark blue of RGBA value of $(0, 0, 0.33, 0.33)$, outputting the final frame (fig. 2.5).

Camera controls

Users can navigate the scene using camera controls common to many commercial animation suite packages:

Alt key + Left mouse button drag Tumbles the camera around a focus point in front of the camera.

Alt key + Right mouse button drag Dollies the camera.

Alt key + Middle mouse button drag Tracks the camera.

At this moment, camera controls are rather coarse and could use some polishing; for example, their sensitivity does not adapt to changes in scene size and this leads to difficulties while navigating small or big scenes. Even the clipping planes ignore the scene size and the default values often create artifacts. Still taking inspiration from commercial packages, camera controls can also be combined with the mechanic of selecting geometries from the viewport to make the camera orbit around a geometry on a key press.

Axes widget

Imitating most 3D applications, a small floating panel showing the current camera orientation by a render of the three coordinate axes is provided. The x-axis is colored in red, the y-axis in green, and the z-axis in blue, following the well established convention in computer graphics. The rendering of it is done without any vertex buffer since the vertices are positioned by the vertex shader using the `gl_VertexID` built-in GLSL variable and some bit-wise operations.

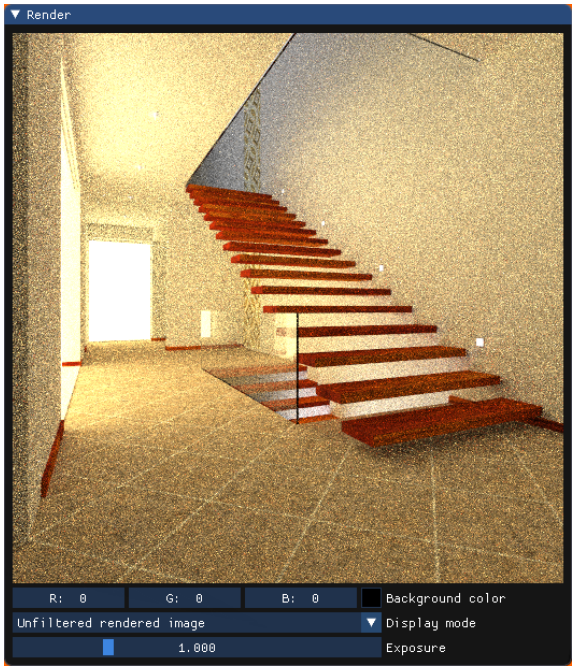
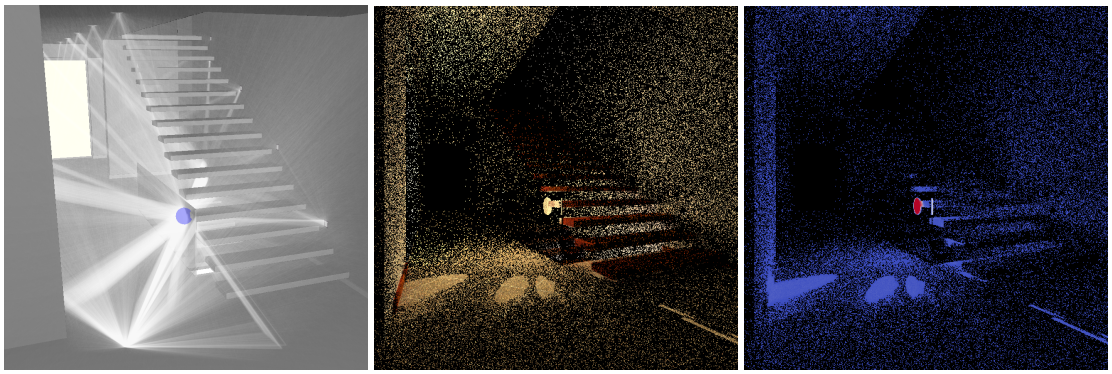


Figure 2.11: "Render" panel in its default state, showing an unaltered render of the *Modern Hall* [Bit16] scene.



(a) Viewport render **(b)** Radiance view **(c)** Bounces view

Figure 2.12: An example of the **"Final radiance"** (b) and **"Paths per pixel"** (c) views of the **"Render"** panel relative to the selected paths shown in a viewport capture (a).

2.3.3 Render image widget

The render image is a vital part of a dataset, after all it is the end result of a path tracer. A panel, simply called **"Render"**, is dedicated to showing the render image and some useful views related to it. By default, as soon as a dataset is loaded from disk, a pristine render image built upon the radiance and camera samples buffers of the dataset[10] (fig. 2.11). Due to its nature, this image has a high dynamic range and it has a linear color space, so a tone mapping operation has been applied on it through a fragment shader. The tone mapping described on the OpenEXR documentation[11] is used here and its *exposure* parameter has been directly linked to the **"Exposure"** slider; users can tweak it to explore the dynamic range of the render.

When some path filters have been placed into the scene, users can access the **"Final radiance"** and the **"Paths per pixel"** views using the **"Display mode"** drop-down list. They both show information about the pixels affected by the currently selected paths and the pixels not touched by the path selection are rendered of the color specified by the user through the **"Background color"** color picker. The **"Final radiance"** view (fig. 2.12b) simply displays the pixels of the final image; it was thought of actually displaying the radiance carried only by the selected paths, but as per today it is still to implement. The **"Paths per pixel"** view (fig. 2.12c) instead shows for each pixel the number of paths originated on it, fitted to $[0, 1]$ using the maximum and minimum as extremes and then color mapped with the *coolwarm* scheme[12]. In other words, it shows the density of paths for each pixel.

The only controls the user has on this widget are the already mentioned **"Exposure"** slider and the **"Background color"** color picker. As a future development, panning and zooming controls for the image plus some other controls on the tone mapping have been proposed.

2.3.4 Bounce density heatmap

In some cases, especially when exploring datasets generated upon scenes the user is not familiar with, it is difficult to place filters in a meaningful way. To be able to do that, approximate insight on how all the paths interact with the scene is needed. Up to what has been described until now, the only way in the tool to render paths is by actually rasterizing their rays over a scene render; this, as already mentioned before and shown in figure 1.1, gets quickly unmanageable and rather useless due to heavy visual cluttering. This is where the *bounce density heatmap* enters the game. As its name partially suggests, it is a heatmap that shows the density of path bounces on the surfaces of the scene. As shown in figure 2.13, it is applied as a texture on the scene surfaces and it is color mapped to the *coolwarm* scheme, the same described in section 2.3.3; in other words, the user can differentiate with ease the surfaces where there are more bounces due to their bright red color from the ones seldom reached by paths, colored in blue. This might, for example, help discover anomalies is the distributions of the paths such as unexpected crowding — or lacking — of paths on a surface area.

[10] Refer to section 2.2.1 for information about the buffers of the dataset

[11] https://www.openexr.com/using.html

[12] The *coolwarm* scheme is a $\mathbb{R} \to \mathbb{R}^3$ mapping that maps a scalar $\in [0, 1]$ to an RGB color. 0 gets mapped to $(0.230, 0.299, 0.754)^\intercal$, 0.5 to $(0.865, 0.865, 0.865)^\intercal$, 1 to $(0.706, 0.016, 0.150)^\intercal$, and all scalars in between are linearly interpolated.

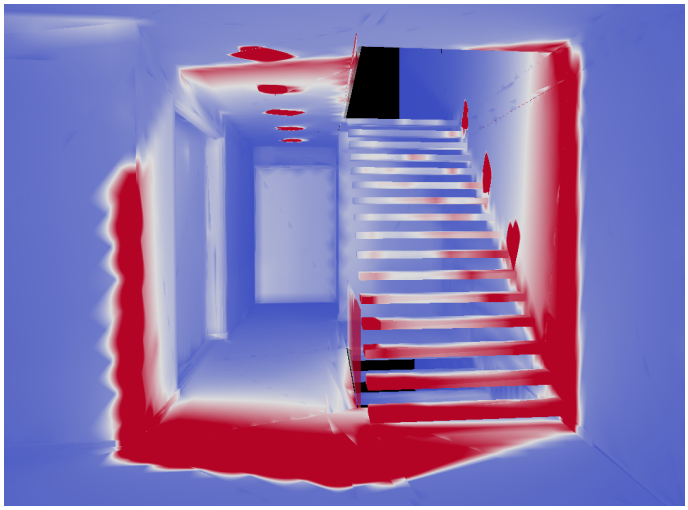
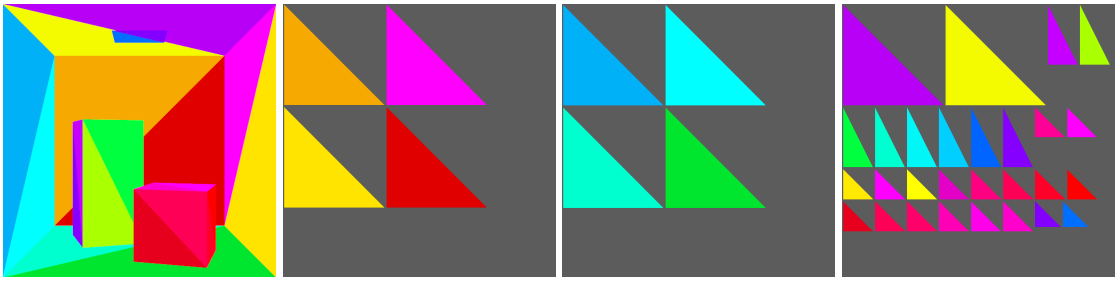


Figure 2.13: Viewport render of the *Modern Hall* [Bit16] scene with the bounce density heatmap relative to a dataset generated by Yocto/GL [PNC19] on top of it.

As already mentioned before and as will be seen in detail in section 2.3.6, the scene format tinkered for this tool strictly contains only geometry: there is no information about either normals or, more importantly in this context, UVs. Since the heatmap is effectively a texture, it needs a UV mapping to be correctly applied to the surfaces. We decided to generate such UV mapping upon scene loading. In practical terms, this means displacing all the triangles in the scene on a two-dimensional plane so that they do not overlap with each other. This is a two-dimensional knapsack problem and finding an optimal solution for it is no trivial task and therefore, since optimizing it would reduce only the GPU memory footprint which is not a critical resource in this context, the very first solution that worked has been adopted without attempting any optimization. The following summarily explained algorithm has been used:

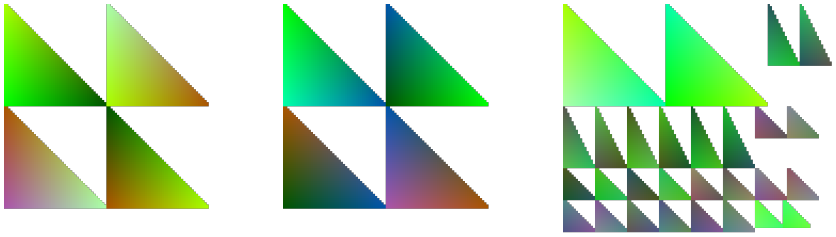
1. Determine a texture size. While testing, 128×128 proved to be a good compromise between resolution and performance.
2. Rotate every triangle in the scene so that it lays flat on the xz-plane with one of its side parallel to the z-axis.
3. Sort the triangles by their height, which is their length along the z-axis.
4. Arbitrarily determine the size in scene units of a single texel and scale all the triangles by that size. After this, all the lengths of the triangles will be expressed in texel size.
5. Start placing the sorted triangles on the first row of a texture, ensuring all vertices of the triangles end up at the center of a texel (shape deformation is allowed). When a row is filled (when a triangle ends up having a vertex with $x > 128$), start a new one. When a texture is filled (when a triangle ends up having a vertex with $z > 128$) create a new texture and start filling that.

At the end of this operation, the output is equivalent to the one visualized in figure 2.14: a two-dimensional UV coordinate spanning across several texture squares is assigned to each scene vertex. These coordinates are appended to the vertex buffers on the GPU used for scene rendering.



(a) Scene **(b)** Map n.0 **(c)** Map n.1 **(d)** Map n.2

Figure 2.14: Triangles of the *Cornell Box* scene rendered on the UV space. A unique color has been applied to every triangle so that it is easier to find matches between the 3D render (a) and the maps (b, c, d).



(a) Map n.0 **(b)** Map n.1 **(c)** Map n.2

Figure 2.15: The world positions of the surface points of the *Cornell Box* scene rendered on the UV space shown in figure 2.14. Black point has been set to −1 and the white one to +2 to improve readability.

During development, it has been briefly considered assigning to every triangle its own texture. Both solutions would have brought us similar results, but the road to reach those results would have been completely different: while solving the knapsack problem was a matter of coming up with an algorithm, storing a texture per triangle while maintaining useful properties such as a constant ratio between texel size and world space units, would have been a matter of cleverly bending OpenGL features and limitations. We opted the first path due to personal preferences.

Once the triangles have been arranged in the UV space, the actual heatmap has to be generated. Since our goal is to provide a heatmap showing the density of path bounces on the scene surfaces, points in world coordinates have to be compared with triangular meshes surfaces. To enable this comparison, the world coordinates of the scene triangles are rasterized onto the UV space using an ad hoc OpenGL render pipeline. The result of this operation, shown in figure 2.15, are multiple `RGB32F` — so that negative values are valid — textures where each fragment holds the world coordinates of the corresponding area patch. In other words, every colored pixel of these textures corresponds to a world space point. Making these maps as precise as they can be had been proven crucial to reduce artifacts in the final heatmap renderings and that lead to extra care while designing this particular rasterization pipeline: for example, to make sure triangles narrower or even smaller than a pixel are correctly rasterized, individual triangles are rendered first as `GL_POINTS`, then as `GL_TRIANGLES` with the `GL_CONSERVATIVE_RASTERIZATION_NV` option activated, and finally as `GL_TRIANGLES` with `GL_FRONT_AND_BACK` polygonMode set to `GL_LINE` — so as wireframe — to

also get the triangles smaller than a pixel that escaped the conservative rasterization pass. These multiple steps may seem — and they are — rather slow but since this render pipeline is invoked only once on scene load, it results reasonable.

Having these *UV → world mappings* and the texel size in world units at hand, computing the actual heatmap can be done by counting for each pixel the path bounces that lie within a sphere centered on the world point stored in it and having radius equal to the texel size. Although conceptually easy, this algorithm requires a remarkable amount of time to complete; after all, for each pixel of those textures, it has to iterate over all path bounces of the dataset. Once considered that a reasonable dataset has about half a billion bounces, it is clear that this operation takes time. This is further worsened by the impossibility of performing any kind of early termination since knowing that a bounce of a path lies within a sphere does not ensure the other bounces of the same path will not and vice versa. To speed up the computation the obvious step of making it multi-threaded has been taken almost immediately and due to the *embarrassingly parallel* nature of the algorithm, time improvements are linearly proportional to the number of available CPU cores. Even with multi-threading and further measures taken to reduce memory allocation and cache trashing, the performances are still disastrous. To give an idea, to generate a heatmap of a 512×512 pixels render with 256 spp of the *Modern Hall* [Bit16] scene on an Intel Core i7-6820HQ (4 physical cores, 8 concurrent threads), it takes about five hours. Possible improvements were thought, most of them regarding embedding the bounces in a spatial acceleration structure, but they have never been implemented and left for future developments. At the beginning it was planned that the tool would compute heatmaps on the fly and everything was designed to accommodate that but, as soon as it was clear the times to generate one are unbearably long, the system has been coded to quietly dump a heatmap on disk as soon as it is done computing and to load that copy the next time the same dataset is loaded. It was also thought of making the heatmap generation optional and asynchronous — now it is blocking — but even this never saw the light due to time constraints.

2.3.5 Double dataset handling

In the middle of development it became progressively clearer that having a way to compare two different datasets with ease would have been a crucial feature to perform debugging on datasets. To satisfy this necessity, the possibility to load two datasets at once was implemented. From the user perspective, when a second dataset is loaded, nothing changes but for the appearance of the **"Dataset switcher"** panel: every other UI component stays the same and it has the same functionalities as before but now the user can either click on the **"Switch dataset"** button or press the **X** keyboard key to switch back and forth between the two datasets. In the current state of things, when a switch happens all visualization changes but the parameters, settings, and filters chosen by the user stay the same. Extra care has been taken to make switching as fast as an ordinary frame render to allow a delay free experience of this rather critical interaction. This came at the expense of memory: both datasets have to be loaded on primary memory at all times and the same goes with the currently selected paths on the GPU memory.

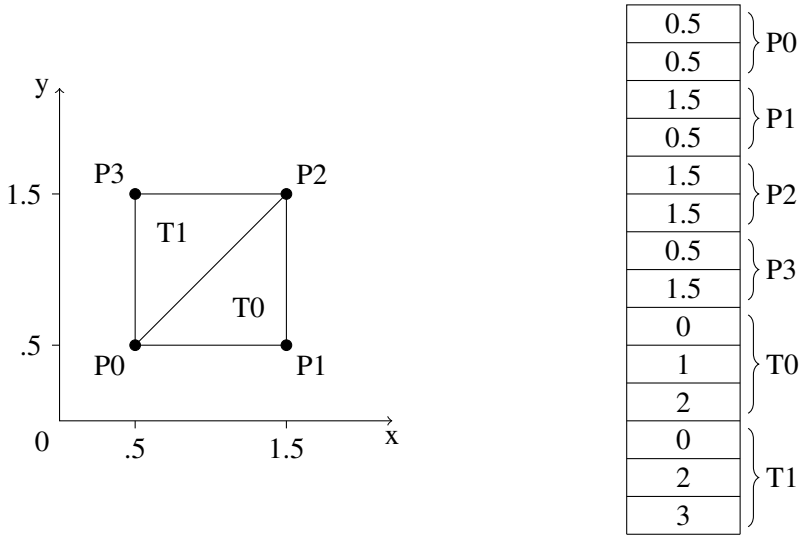


Figure 2.16: Geometry example. To simplify, a two-dimensional case is presented. On the right a simple triangulated mesh is shown, on the left there is a valid binary file. A corresponding JSON file is shown in Listing 2.3.

2.3.6 Scene format

As mentioned before, this tool uses an ad hoc scene description format. This format supports only triangle meshes and represents each with one vertex and one index buffer. A whole scene is contained in a JSON [Bra17] and a binary file. They must have the same name and respectively have the `.json` and `.bin` extension.

The binary file contains all the mesh buffers as a contiguous sequence of little-endian 32bits numbers. While vertex buffers are stored as arrays of single precision floats, index buffers are arrays of unsigned integers.

The JSON file, beside containing information about the camera and the render image, lists the scene geometries. For each it stores a color, a flag determining if the object is in any way translucent, a name, and buffer descriptions for both the vertex and the index buffer. These have a field holding their type — which is either `vertices` or `indices` —, an `offset` field determining the distance in bytes from the beginning of the binary file to the first byte of the buffer, and a `size` field storing the size of the buffer in bytes. For detailed information on the JSON structure a schema is provided in Appendix A.

Listing 2.3 JSON scene file example relevant to Figure 2.16. Information not relative to the geometry shape has been omitted. Please also remember this is a two-dimensional example, hence the vertex buffer size is only 2 dimensions * 4 points * 4 bytes = 32 bytes instead of being 48 bytes like it would be in the ordinary three-dimensional case.

```
{
  "camera": {...},
  "render": {...},
  "geometries": [
    {
      "name":     "quad",
      "material": {...},
      "buffers": [
        {
          "offset": 0,
          "size":   32,
          "type":   "vertices"
        },
        {
          "offset": 32,
          "size":   24,
          "type":   "indices"
        }
      ]
    }
  ]
}
```

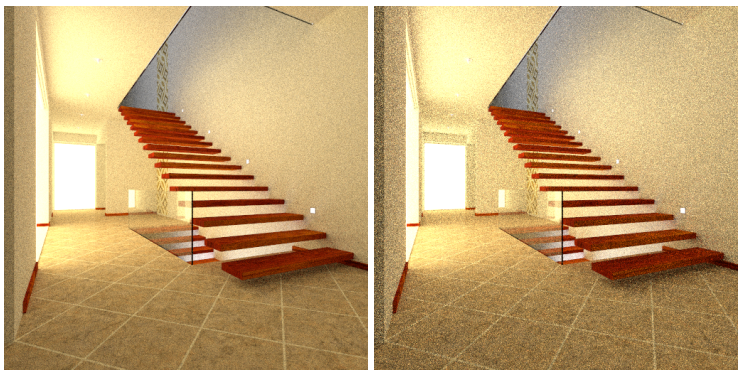
3 Results

Assessing the validity of a tool without performing user tests is often pointless but, due to the complexity of the tool, there was no time to perform any. In their absence, the analysis of two couples of significant datasets are presented.

3.1 First dataset couple

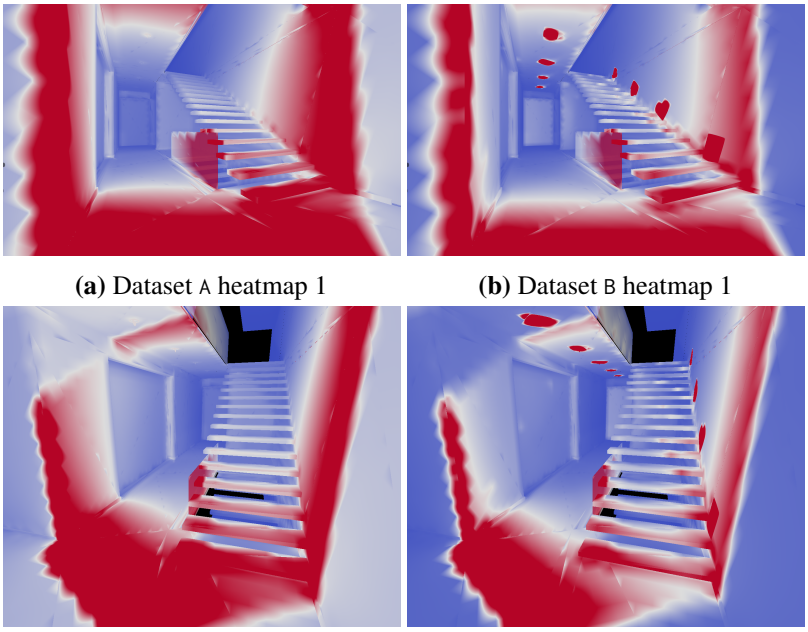
The first example we are analyzing consists in comparing two datasets generated by Yocto/GL [PNC19] on the *Modern Hall* scene [Bit16] with same resolution and same number of samples per pixel — 512×512 and 256 spp — but they are using different next event estimation techniques. The algorithm behind dataset `A` is using importance sampling based on the surface materials only, while the one behind dataset `B` is using multiple importance sampling on both surface materials and scene light sources. Given this, by intuition, the final render of dataset `B` should be less noisy than the one of dataset `A` but, as pictured in figure 3.1, it is actually the opposite. Exploiting the features of the presented tool, we will now illustrate the reason behind this.

Bounce density heatmap visualization (fig. 3.2) clearly shows the effects of importance sampling: please notice how the most bounce-dense areas in dataset `B` are localized on the lights on the ceiling and near the stairs and how the glass panel tends to be redder than its surroundings in both datasets. While the former shows how the paths of dataset `B` tends to go towards the light in the scene, the latter is a direct manifestation of the total internal reflection phenomenon; paths that enter the glass pane from its narrower side — the one closer to the camera — gets trapped inside the glass and keep bouncing on the internal glass surface (fig. 3.3) as similarly as it happens inside an optic fiber cable. This would be possible only when performing next event estimation based on the surface material



(a) Render `A` **(b)** Render `B`

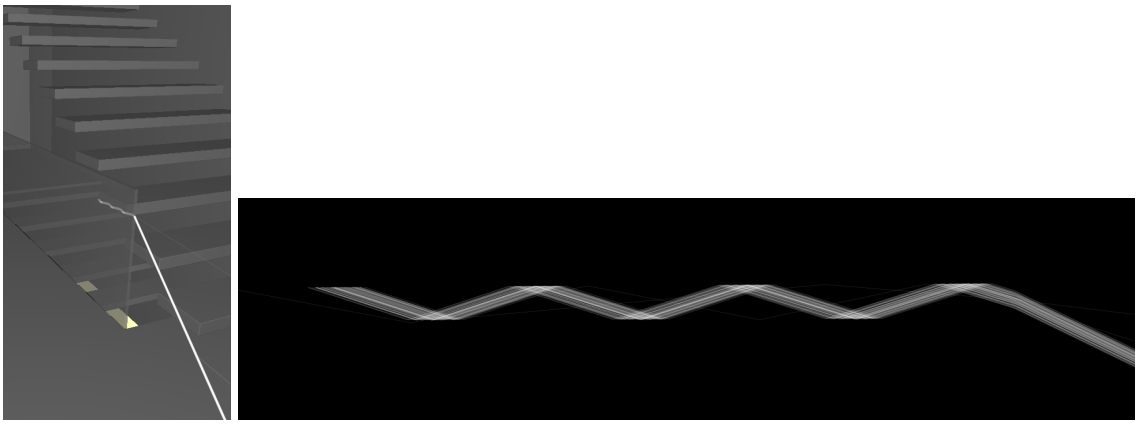
Figure 3.1: Render images of the two datasets.



(a) Dataset A heatmap 1 **(b)** Dataset B heatmap 1

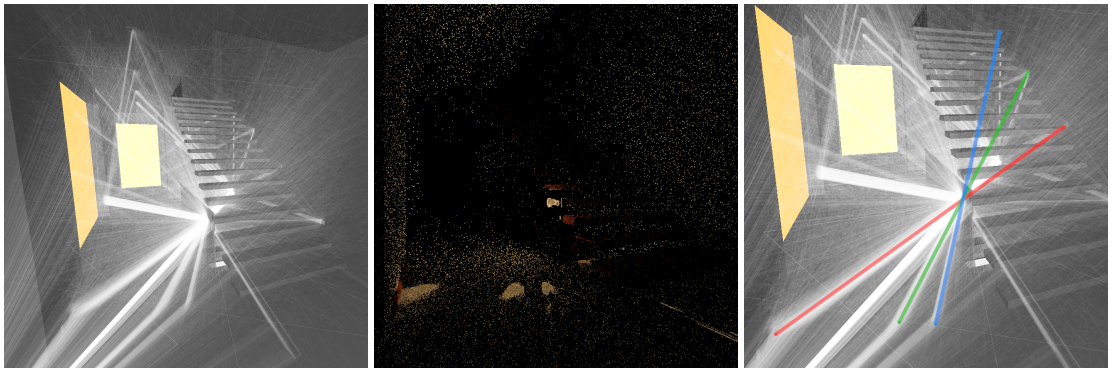
(c) Dataset A heatmap 2 **(d)** Dataset B heatmap 2

Figure 3.2: Heatmap renderings for both datasets from two different cameras. **"Heatmap max"** parameter has been set to 200, 000 for both.



(a) **(b)**

Figure 3.3: Viewport renders of a hundred of paths bouncing inside the glass panel, demonstrating the total internal reflection phenomenon. (**b**) is taken from above with all geometries hidden. Please note also how the paths bend upon entering the glass surface on the extreme right of (**b**).



(a) Viewport **(b) "Final radiance"** viz **(c)** Bundles detail

Figure 3.4: Visualizations of the paths selected by a sphere filter placed on the glass panel for dataset `B`. (**a**) shows a viewport render, (**b**) is the view available in the **"Image"** panel with the **"Final radiance"** visualization mode selected (sec. 2.3.3), and (**c**) shows highlighted in red, green, and blue three paths bundles and how they were directed to three light sources but have been occluded by the glass panel.

properties: without accounting the index of refraction of glass while generating new directions after the bounces, a path tracer would not be able to reproduce this phenomenon. Even though this is an interesting result, it does not entirely explain why a render clearly done using multiple importance sampling has more noise that the one without this feature. A hint is there though: as already noted, a copious amount of paths of dataset `B` bounced on the tiny lights on the ceiling and next the stairs, even if this is normal for this kind of algorithm, it gets worrying when noticed that the scene is almost entirely illuminated by the two warm light portals on the corridor and the two cold ones down and up the stairs; the tiny lights where all those paths ended up to do not contribute at all at the global illumination of the scene. This means that every path that ends up on those lights will carry very little radiance compared to the ones that reach the portal lights, introducing noise in the final render. But let us analyze this further.

By placing a sphere filter on a point on the glass panel and looking at how dataset `B` reacts, more information starts showing (fig. 3.4). Many path bundles connecting various areas in the scene appear and it seems that, while most of these connect the camera, lights, and the filter between each other, there are some that apparently are not directly connected to any relevant point. But, after further inspection, it is clear that they are bundles of paths that try to reach a light source but can not since there are some other scene geometries sitting between the bundle and the light. Please look, for example, at the three path bundles connecting the camera to the sphere filter while bouncing on the floor highlighted by figure 3.4c and how they do not seem to be connected to any light source directly; they are made of paths that have been directed towards the lights on the stairs but could not due to the glass panel standing in their way. We can coincidentally see them since our sphere filter sits on the portion of the glass panel in the way of those paths.

Looking at figure 3.4a with more attention, more curious path bundles can be spotted: almost every light on the wall by the stairs has a bundle entering it while being almost parallel to the wall. A detail of the first stair light is presented in figure 3.5 and it can be seen that the bundle seems to hit the dark gray frame surrounding the light more that the light itself. Hiding the frame geometry in the viewport confirms the suspect: all those paths which have been shot in direction of the stair

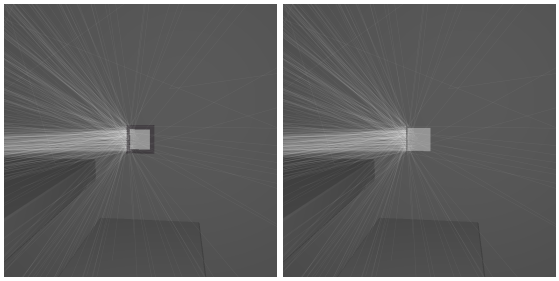


Figure 3.5: Detail of the first stair light with the surrounding geometry shown on left and hidden on the right.

light are completely wasted since they are occluded by a frame. Furthermore, the frame seems to occlude a significant portion of the light surface making it unreachable by any path while still being a valid candidate for light sampling; in other words, the next event estimation will keep trying shooting paths on those surfaces hidden under the frame but will never reach them in any possible case, wasting energy and time.

To summarize, the render with multiple importance sampling over both the materials and the lights — dataset `B` — has more noise than the one sampling only against the materials — dataset `A` — because:

1. The renderer, Yocto/GL, while sampling the light areas, gives the same weight to lights with a huge difference in intensity. This leads to having many paths that carry very little radiance even if they hit a light. These, when averaged with the ones hitting the brighter lights, make the convergence slower which equates to more noise in the final render.
2. The scene appears to be somewhat flawed: there is superfluous geometry occluding a significant portion of some lights. This hinders a correct next event estimation since those occluded portions can still be targeted by paths seeking a light.

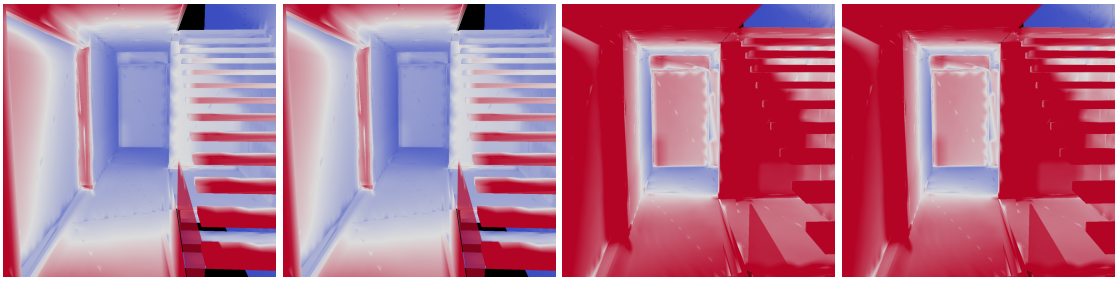
In this example we presented a possible use of our tool geared towards debug and educational purposes alike. Understanding the source of noise in the render can surely help while debugging but while doing that we exposed many compelling features and side effects of a path tracer that might help explain the inner workings of global illumination to pupils and such in a visual way.

3.2 Second dataset couple

To provide a more challenging test, the datasets of which renders are shown in figure 3.6, both generated by Yocto/GL [PNC19] without multiple importance sampling, have been tinkered to present a very subtle difference. Dataset `A` is correct while dataset `B` has been generated after introducing an error in the sampling of microfacet distributions: the direction vector generated according to the distribution has the y and z components swapped before being transformed from surface space to world space. Logically, this should have generated way more noticeable visual artifacts in the render but, as highlighted by figure 3.6c, it looks like the wrong render does not account for the Fresnel effect correctly. By just looking at the renders, the user might be led into

(a) Render A **(b)** Render B **(c)** A − B

Figure 3.6: Render images of the two datasets (**a**, **b**) and their mathematical delta (**c**) to show their differences. (**c**) has been generated outside the tool and presented for clarity.



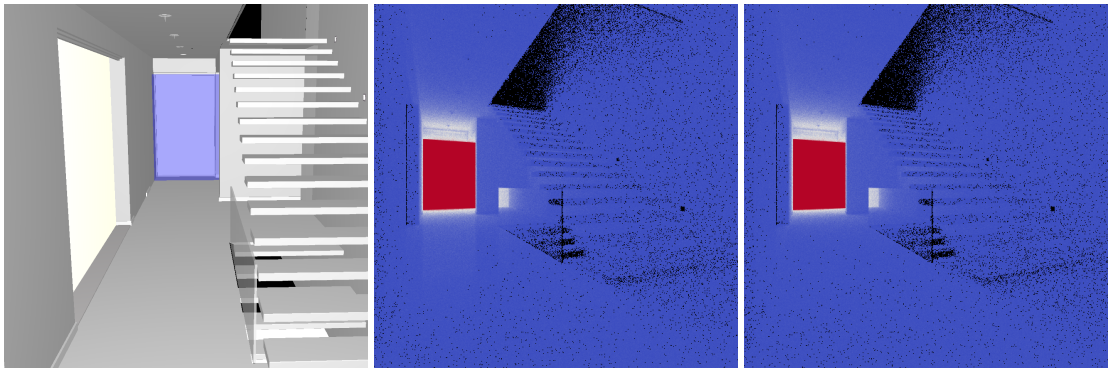
(a) Dataset A heatmap 1 **(b)** Dataset B heatmap 1 **(c)** Dataset A heatmap 2 **(d)** Dataset B heatmap 2

Figure 3.7: Heatmap renderings for both datasets from two different cameras and with two different parameters for the **"Heatmap max"** parameter (for (**a**) and (**b**) is set to 100, 000 while for (**c**) and (**d**) is set to 40, 000).

thinking that there is an error in the Fresnel term calculation or in the surface material in general but here, the situation is different: the direction of the rays are wrong not the calculations of the energy they carry.

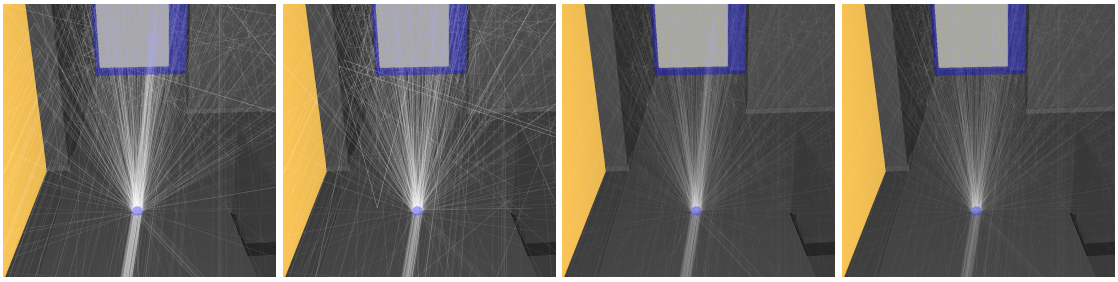
The first instinct was to check the heatmap on the areas where the renders are most different — that is to say the floor right in front of the light portal at the end of the corridor — but only minimal differences can be spotted. In the screenshots in figure 3.7 vague changes can be seen with a lot of effort, and they just achieve to communicate that the correct dataset — dataset A — has a slightly higher bounces density than the wrong dataset. No conclusions can be deducted from this little information.

Next step has been to place a window filter covering the entirety of the light at the end of the corridor. Due to the high number of paths selected, the viewport was too visually cluttered to be of any utility, so the focus went on the **"Render"** panel in the **"Paths per pixel"** render mode. This, pictured in figures 3.8b and 3.8c, showed how many more paths in dataset A having their first bounce of the floor ended up hitting that light. It seemed like a further confirmation that something was wrong with the Fresnel term computation.



(a) Filter position **(b)** Dataset A **(c)** Dataset B

Figure 3.8: Paths per pixel visualization for the window filter shown in (**a**).



(a) A, radiance scaled **(b)** B, radiance scaled **(c)** A, no radiance scaling **(d)** B, no radiance scaling

Figure 3.9: Viewport renderings of paths bouncing in the sphere filter placed on the floor and through the window filter on the light in the back. (**a**) and (**b**) have the **"Radiance scaling"** visualization option activated while the others do not. Scene rendering has been darkened using the tool's visualization options.

It all seemed even further confirmed by placing a sphere filter on the floor where the Fresnel effect shows and analyzing the geometry of paths through there. As shown in figure 3.9, a clear bundle of rays going from the sphere to the window filter appears for dataset A, but not for dataset B. When only looking at those paths having the **"Radiance scaling"** visualization option activated (fig. 3.9a and 3.9b), the theory about the error in the Fresnel term computation might still stand strong; after all, due to the high quantity of paths displayed, it is difficult to understand if that bundle of rays is there in both datasets but it carries way less radiance in dataset B due to miscalculations while evaluating the material.

All doubts are dispelled by disabling the **"Radiance scaling"**. As soon as it is off, it is clear that that bundle of rays is there, no matter the energy it carries. It shows that the problem is in the picking of a new ray direction that, even if strictly related to surface material evaluation, it is still not in the Fresnel term evaluation.

This conclusion, even if correct, has been drawn by taking several steps in a direction probably dictated by our previous knowledge of the bug. We do not know and cannot make estimates on how any different user would approach the debugging process having this tool available. Nevertheless, this example would fit way more in an educational context as much as the previous did: it could be

used to show in a visual way the crucial importance of correct next event estimation since it does not only affect the convergence times — so how much noise ends up being in the final image — but also the *correctness* of the render process.

4 Conclusions

We presented a visual debugging and exploration tool capable of showing how paths shot by a path tracer interact with the scene. We showed how the tool is structured in a data gathering part and a visualization client one; the gatherer can be plugged into the code of pre-existing path tracers as a library and dumps the needed data on disk during rendering; the client lets users explore the gathered data. Due to the sheer amount of paths a tracer has to shoot, we showed how the provided filtering tools, combined with the visualization widgets and options help users in exploring the datasets with relative ease. We dived into the implementation details of the bits that make our tool unique and into the performance exploits to speed up operations that would not be possible otherwise. Such as, to name one, the multi-pass rendering pipeline that lets user visually explore hundreds of thousands of paths on a personal computer. We also detailed two couples of datasets that show the capabilities of the previously introduced features combined, partially demonstrating how the tool can be used as both a valid debug framework for path tracers and as an educational tool to explain with practical examples how a tracer works under the hood.

4.1 Future developments

The tool we presented is far from being defined complete. Several improvements can be done all over and many have been noted and described throughout this document. Many others, though, since not inherent to any particular feature are gathered here.

More filters Several path selection filters have been imagined during development, such as one excluding paths carrying less than a threshold radiance or another selecting paths only shot from a set of pixels selected via the **"Image"** panel. Interesting was the idea of extending the sphere filter adding a sort of *sub-filters* such as one selecting the paths both bouncing in the sphere and bouncing in a user-controlled cone of directions; that would have been for example helpful for focusing on paths going toward a light after bouncing on an interesting surface patch. Another useful sphere filter sub-filter would have been one splitting between the bounces that have their direction sampled from the surface material and the ones that sampled scene lights; it would have been helpful in both examples of chapter 3 but implementing it would have meant almost doubling the memory footprint of each bounce.

Save and load status The visualization client has several parameters of which settings are lost every time the application is closed. Useful with no doubts to every user would be a save and load feature for the parameter settings. Since most parameters are controlled by ImGui widgets, maybe it would be clever to use the state preservation feature of the library.

Disk streaming Every dataset we tested hardly went over 1 billion bounces, but production renders tend to have way more bounces: a Full HD (1920×1080 pixels) render with 1024 spp has roughly double the bounces, which means a memory footprint twice as big. So much

data will not entirely fit on the primary memory of most personal computers and it makes data streaming from disk an essential feature for those cases. Even with the most careful implementation, streaming would impact performances but having something working slow is better than something not working at all.

Spatial acceleration structures Talking about performances brings up a somehow obvious possible improvement: spatial acceleration structures. Embedding every bounce in a data structure such as a *k-d tree* [Ben75] and then storing the paths' topology — so which bounces make each path — in a separate data structure, would improve performances of many bits of the visualization tool. Consider how the current filtering options are all based upon spatial queries and how these could be immensely faster if bounces were in a acceleration structure. As already mentioned in section 2.13, the same goes for the bounce density heatmap generation, which is currently the slowest process on our tool.

User testing Assessing the usability and usefulness of a software piece is no easy task and is canonically delegated to user testing. Gathering people orbiting around path tracing and perform usability testings with them would surely give great insight on what features can be improved, added or discarded all together.

Bibliography

[Ben75] J. L. Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517 (cit. on p. 44).

[Bit16] B. Bitterli. *Rendering resources*. 2016. URL: https://benedikt-bitterli.me/resources/ (cit. on pp. 13, 15, 27, 29, 31, 35).

[Bra17] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://rfc-editor.org/rfc/rfc8259.txt (cit. on p. 32).

[CPC84] R. L. Cook, T. Porter, L. Carpenter. "Distributed ray tracing". In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 1984, pp. 137–145 (cit. on p. 7).

[GFE+12] C. Gribble, J. Fisher, D. Eby, E. Quigley, G. Ludwig. "Ray tracing visualization toolkit". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2012, pp. 71–78 (cit. on p. 7).

[Kaj86] J. T. Kajiya. "The rendering equation". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, pp. 143–150 (cit. on pp. 7, 8).

[Kre19] C. Kreisl. "EMCA - Explorer of Monte-Carlo based Algorithms". MA thesis. 2019. URL: https://github.com/ckreisl/emca (cit. on pp. 8, 11).

[KW09] M. H. Kalos, P. A. Whitlock. *Monte carlo methods*. John Wiley & Sons, 2009 (cit. on p. 9).

[Lam60] J. H. Lambert. *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. Klett, 1760 (cit. on p. 22).

[LP14] H. Lesev, A. Penev. "A framework for visual dynamic analysis of ray tracing algorithms". In: *Cybernetics and Information Technologies* 14.2 (2014), pp. 38–49 (cit. on p. 7).

[MT97] T. Möller, B. Trumbore. "Fast, minimum storage ray-triangle intersection". In: *Journal of graphics tools* 2.1 (1997), pp. 21–28 (cit. on p. 22).

[PNC19] F. Pellacini, G. Nazzaro, E. Carra. "Yocto/GL: A Data-Oriented Library For Physically-Based Graphics". In: (2019) (cit. on pp. 13, 15, 29, 35, 38).

[RAPC14] F. Robinet, R. Arnaud, T. Parisi, P. Cozzi. "glTF: Designing an open-standard runtime asset format". In: *GPU Pro* 5 (2014), pp. 375–392 (cit. on p. 13).

[Shn03] B. Shneiderman. "The eyes have it: A task by data type taxonomy for information visualizations". In: *The craft of information visualization*. Elsevier, 2003, pp. 364–371 (cit. on p. 7).

[SHP+19] G. Simons, S. Herholz, V. Petitjean, T. Rapp, M. Ament, H. Lensch, C. Dachsbacher, M. Eisemann, E. Eisemann. “Applying visual analytics to physically based rendering”. In: *Computer Graphics Forum*. Vol. 38. 1. Wiley Online Library. 2019, pp. 197–208 (cit. on pp. 7, 11).

[SML06] W. Schroeder, K. Martin, B. Lorensen. *The Visualization Toolkit (4th ed.)* Kitware, 2006. ISBN: 978-1-930934-19-1 (cit. on p. 14).

[VG95] E. Veach, L. J. Guibas. “Optimally combining sampling techniques for Monte Carlo rendering”. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 1995, pp. 419–428 (cit. on p. 10).

[WAHD19] A. Wright, H. Andrews, B. Hutton, G. Dennis. *JSON Schema: A Media Type for Describing JSON Documents*. Internet-Draft draft-handrews-json-schema-02. Work in Progress. Internet Engineering Task Force, Sept. 2019. 75 pp. URL: https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-02 (cit. on p. 47).

[Whi79] T. Whitted. “An improved illumination model for shaded display”. In: *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*. 1979, p. 14 (cit. on p. 7).

All links were last followed on December 31, 2020.

A Scene format JSON schema

Is hereby presented a JSON schema[WAHD19] on which the JSON file of the scene description format must be built upon. Refer to Subsection 2.3.6 for more information on the scene description format.

```
{
  "type": "object",
  "properties": {
    "camera": {
      "type": "object",
      "description": "Camera properties"
      "properties": {
        "eye": {
          "type": "array",
          "description": "World position of camera eye"
          "minItems": 3,
          "maxItems": 3,
          "items": {
            "type": "number"
          }
        },
        "look": {
          "type": "array",
          "description":
            "World normalized vector of camera look direction"
          "minItems": 3,
          "maxItems": 3,
          "items": {
            "type": "number"
          }
        }
      }
    },
    "render": {
      "type": "object",
      "description": "Rendered image properties",
      "properties": {
        "width": {
          "type": "integer",
          "description": "Width (in pixels) of the rendered image"
        },
        "height": {
          "type": "integer",
          "description": "Width (in pixels) of the rendered image"
        },
```

```
      "spp": {
        "type": "integer",
        "description": "Samples Per Pixel of the rendered image"
      }
    }
  },
  "geometries":{
    "type": "array"
    "items": {
      "type": "object",
      "description": "A single geometry",
      "properties": {
        "name": {
          "type": "string"
        },
        "buffers": {
          "type": "array",
          "items": {
            "type": "object",
            "description": "Buffer description",
            "items" {
              "type": {
                "type": "string",
                "description": "Type of buffer",
                "enum": ["indices", "vertices"],
              },
              "offset": {
                "type": "integer",
                "description":
                  "Offset (in bytes) relative to the binary file"
              },
              "size": {
                "type": "integer",
                "description": "Size (in bytes)"
              }
            }
          }
        },
        "material": {
          "type": "object",
          "description": "Basic material description",
          "items" {
            "color": {
              "type": "array",
              "minItems": 3,
              "maxItems": 3,
              "items": {
                "type": "number"
              }
            }
            "translucent": {
              "type": "boolean"
            }
```

```
            }
          }
        }
      }
    }
  }
}
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature