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
Supervisor:	Dr. Guido Reina Michael Becher, M.Sc. Patrik Gralka, M.Sc.
Commenced:	July 1, 2020
Completed:	January 4, 2021

Abstract

<Short summary of the thesis>

Contents

1 Introduction

1.1 Motivation

This project has been based upon the idea of giving the user the power to visually and interactively explore all the paths generated by a path tracer during rendering. In the very first vision, a user should have been able to select a portion of a surface of the 3d scene the tracer has been run upon and see the paths that bounce there with a bunch of useful data. To be able to do that the whole set of paths shoot by a tracer are needed: by the very stochastic nature of a path tracer, it is impossible to determine which paths will end up bouncing where without resolving them all first. That is why it has been decided it was essential to store data about each path during the rendering process. To make the tool usable in most possible use cases, it had to be able to plug into an existing path tracer and this lead to the conception of the tool as a two software pieces suite: a *data gatherer library* called `gatherer` and a *visualization client* called `gathererclient`.

1.2 Technology

C++ does not have a built-in `half` type so an IEEE 754 compatible header only library[1] has been used thoroughly the written software.

[1] http://half.sourceforge.net/

2 The tool

2.1 General description

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 in triangles.

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 the most naïve solution has been opted: each bounce is stored as a 3-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 2.1. 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 shoot 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:

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

Figure 2.1: *Modern Hall* scene [Bit16] rendered by Yocto/GL [PNC19]. It is a 512x512, 512spp image; maximum bounces per path are set to 10.

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 2.1 would approximately be:

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

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 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 2.1 —, 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 2.3.3 — 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 contained 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 and each contain:

`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[1], 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

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

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
  );
}
```

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 though noted 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 that are written only 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 sadly a fully disk-bound operation and it may take a while, especially on rather big dataset.

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;
};
```

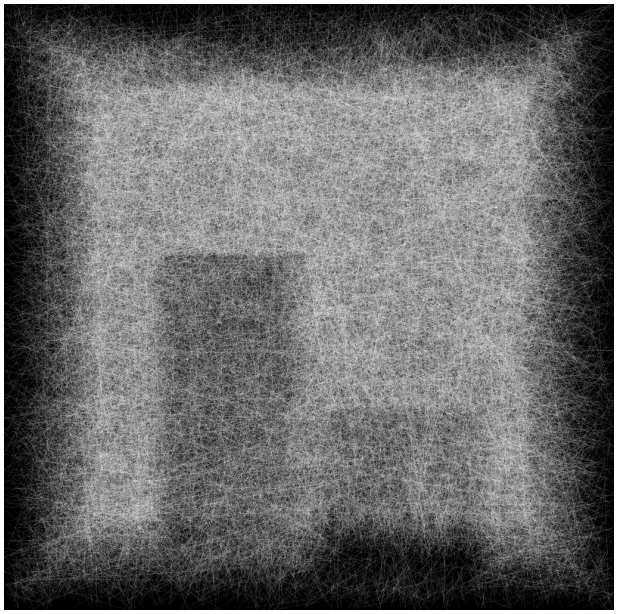

Figure 2.2: Visual clutter in an early render of all the paths used to path trace a 64×64, 1 spp image of the Cornell Box scene. The actual scene is not rendered to improve readability.

2.3 Visualization client

2.3.1 Path filters

Visualizing all paths at once is both useless and impossible in most cases. Useless due to visual clutter — see figure 2.2 — and impossible because most consumer grade GPUs cannot render several gigabytes of paths at once and keep an interactive frame rate. 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.

An unlimited number and combination of filters of any kind 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 $F_1(\mathcal{S}) \cap F_2(\mathcal{S}) \cap ... \cap F_n(\mathcal{S}) \subseteq \mathcal{S}$. 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 put apart 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.

Sphere filter

Window filter

2.3.2 Bounce density heatmap

It might be extended by letting users draw transfer maps over a histogram.

2.3.3 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, it 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.

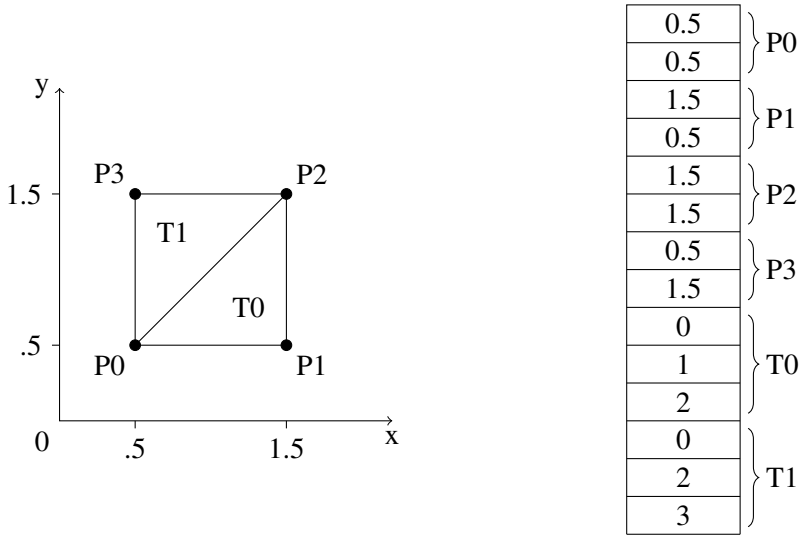

Figure 2.3: Geometry example. To simplify, a 2-dimensional case is presented. On the right a simple triangulated mesh is shown, on the left there is a valid binary file. A corresponding JSON file is shown in Listing 2.3.

Listing 2.3 JSON scene file example relative to Figure 2.3. Information not relative to the geometry shape has been omitted. Please also remember this is a 2-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 3-dimensional case.

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

Bibliography

[Bit16] B. Bitterli. *Rendering resources*. https://benedikt-bitterli.me/resources/. 2016 (cit. on p. 10).

[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. 14).

[PNC19] F. Pellacini, G. Nazzaro, E. Carra. "Yocto/GL: A Data-Oriented Library For Physically-Based Graphics". In: (2019) (cit. on p. 10).

[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. 9).

[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. 19).

All links were last followed on December 8, 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 2.3.3 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