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:

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

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:

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

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 2.3.6 — 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 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.

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

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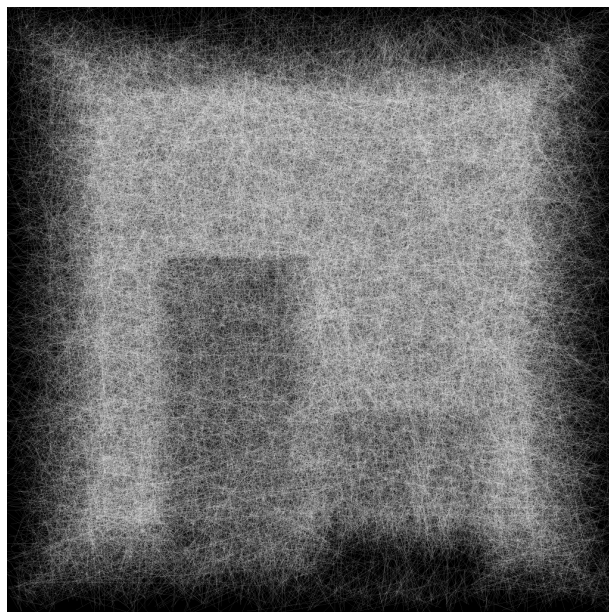

Figure 2.2: Visual clutter in an early render of all the paths used to path trace a 64×64, 1 spp image of the Cornell Box scene. The actual scene is not rendered to improve readability.

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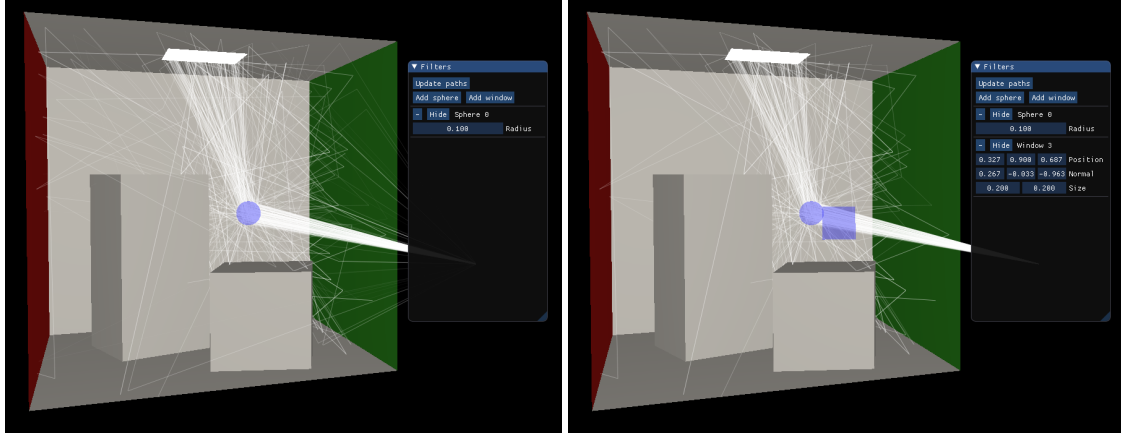
(a) In the beginning, one sphere is placed on the back wall.

(b) Then a window is placed to select only the paths that go from the camera to the sphere on the back wall.
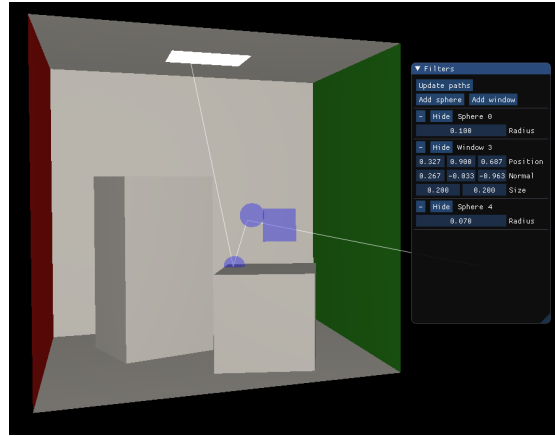
(c) Finally, a second sphere is placed on top of the short box to select only a single path.

Figure 2.3: Example of using more path filters together in a small dataset generated on the Cornell Box scene of 64×64 pixels and 32 spp.

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:

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

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. To further speed the filtering up, 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:

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

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:

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

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. 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.

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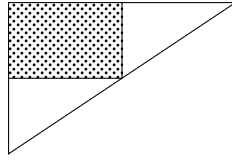
Figure 2.4: The triangle encompassing a window filter (dotted). It is just a right triangle of which right angle lies on a window vertex and of which catheti are as long as twice the window sides.

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 side 2 centered on the origin is generated on 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.

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 Bounce density heatmap

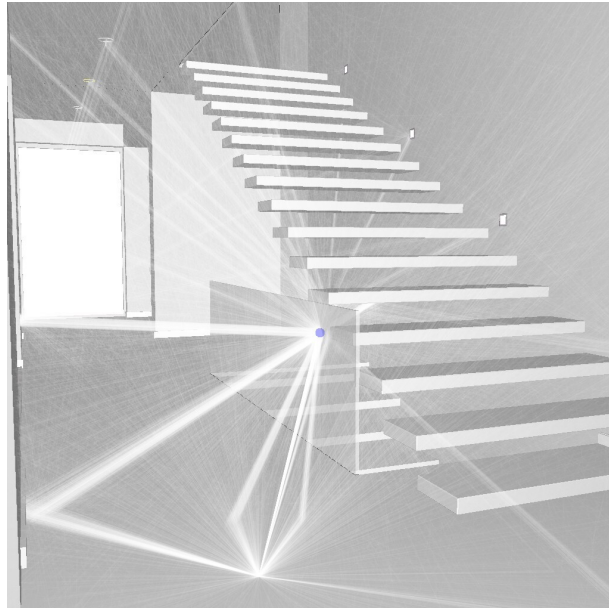It might be extended by letting users draw transfer maps over a histogram.

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

2.3.3 Viewport

The 3-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. The pipeline 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.

The pass for translucent objects is practically equivalent to the previous pass, the only peculiarity is that it writes beauty and depth on new textures while the world positions and the geometries IDs are written upon the ones generated by the opaque pass. Figure 2.7 shows the textures after this pass.
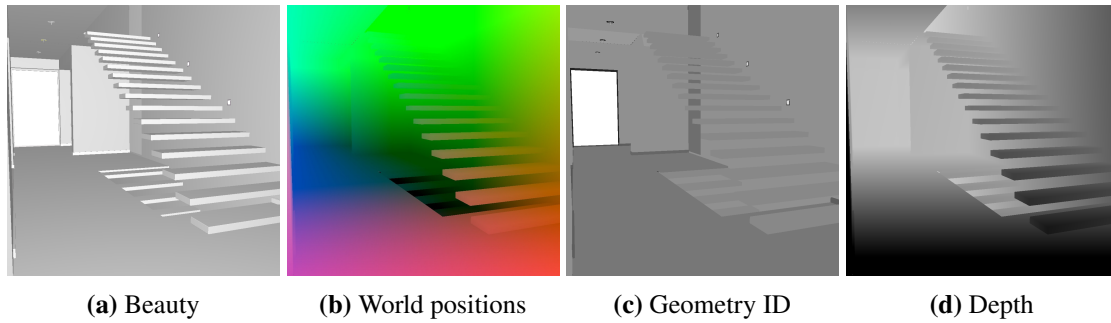
(a) Beauty   **(b)** World positions   **(c)** Geometry ID   **(d)** Depth

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.

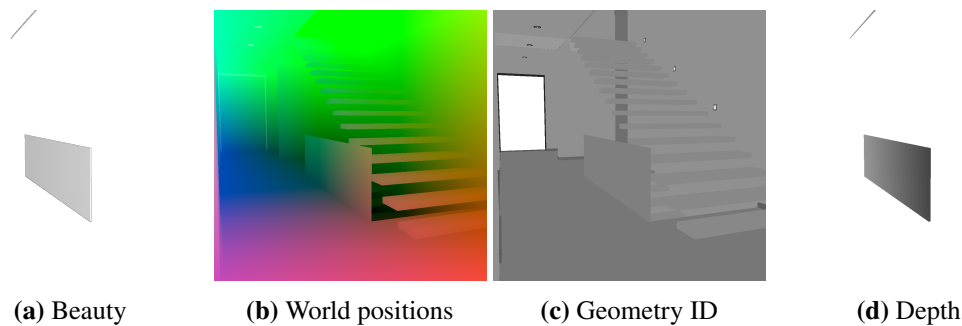**(a)** Beauty   **(b)** World positions   **(c)** Geometry ID   **(d)** Depth

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.

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.

2.3.4 Render image widget

2.3.5 Double dataset handling

- User can load two datasets at once and compare them.
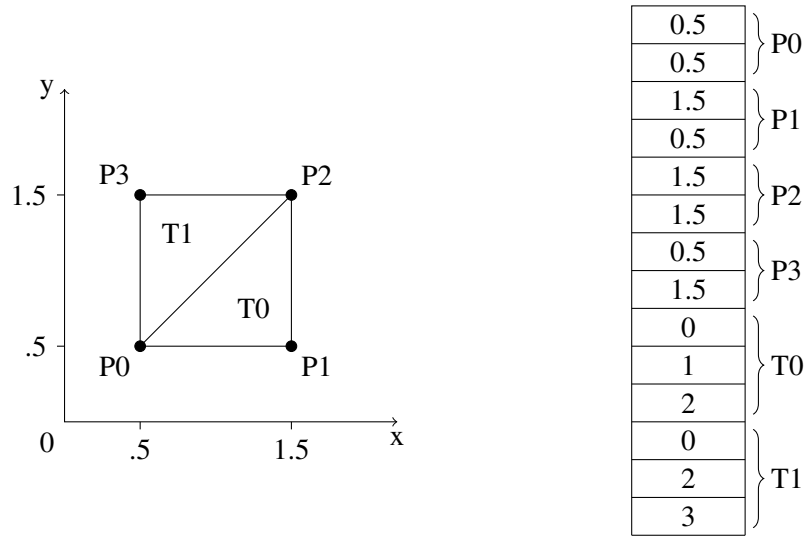- It works by having both on memory

Figure 2.8: 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.

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, 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.

Listing 2.3 JSON scene file example relative to Figure 2.8. 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. 19).

[Lam60]   J. H. Lambert. *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. Klett, 1760 (cit. on p. 17).

[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. 16).

[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. 23).

All links were last followed on December 11, 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