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 2.3.6 — 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.

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

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

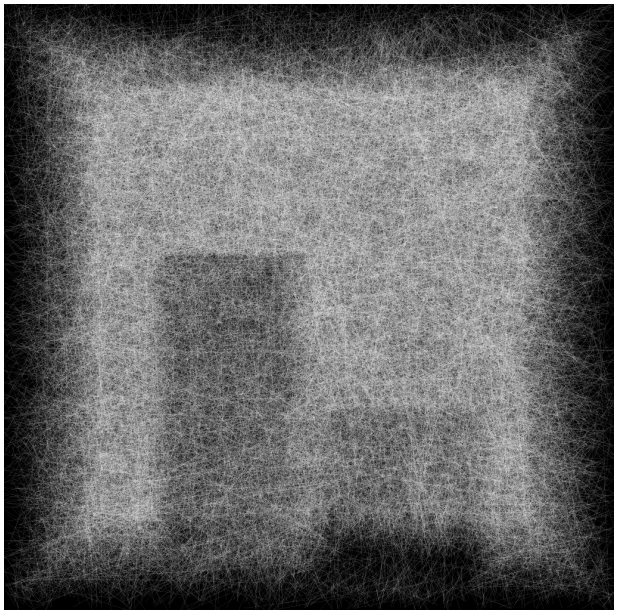
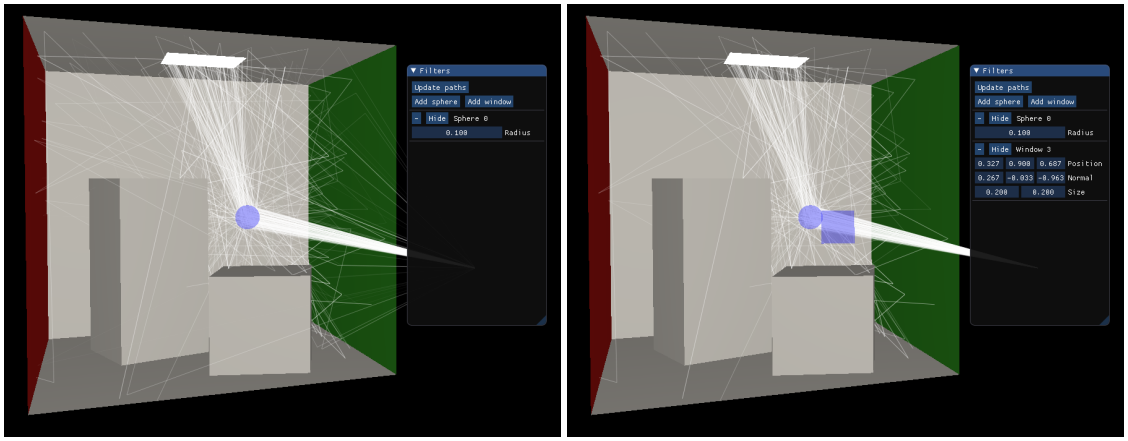


Figure 2.2: Visual clutter in an early render of all the paths used to path trace a 64×64, 1 spp image of the Cornell Box scene. The actual scene is not rendered to improve readability.

2.3 Visualization client

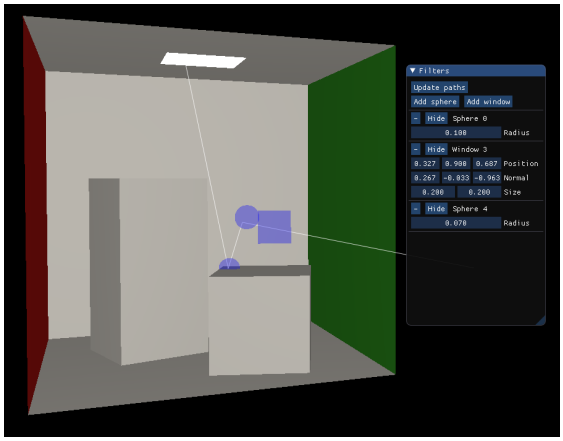
2.3.1 Path filters

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:

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

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 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:

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

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. 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'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 $S_{1\ldots n}$ 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 $|S_{1\ldots n}|$ slots each

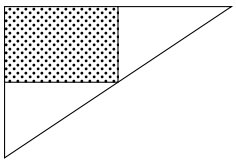


Figure 2.4: The triangle encompassing a window filter (dotted). It is just a right triangle of which right angle lies on a window vertex and of which catheti are as long as twice the window sides.

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 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. To save resources and keep the UI snappy even when the viewport requires most GPU resources, the

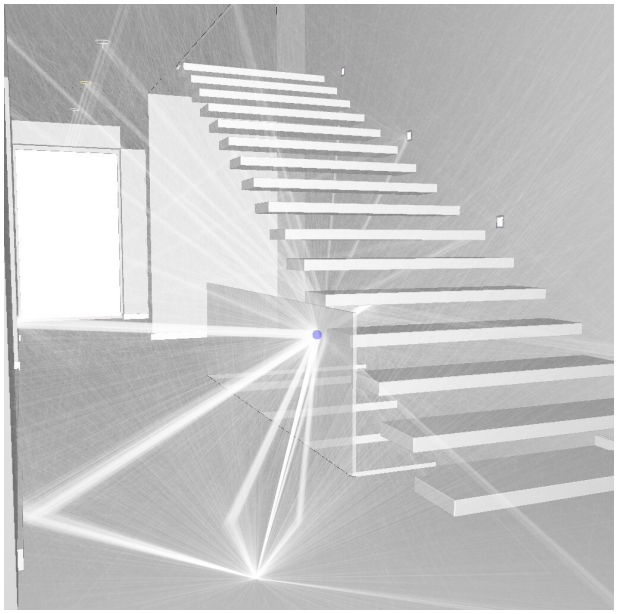
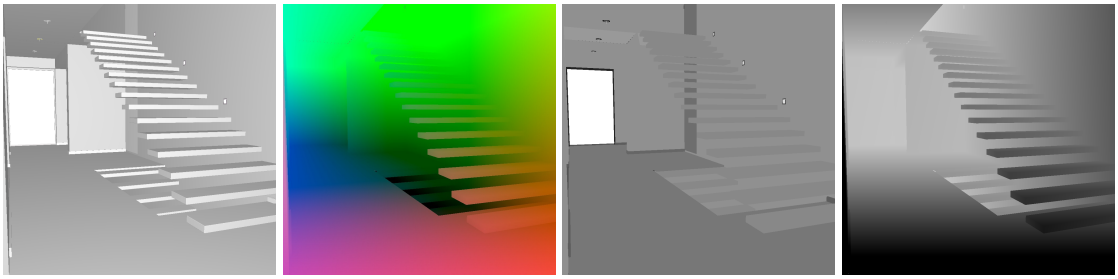


Figure 2.5: An example of a final frame rendered by the viewport.

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.

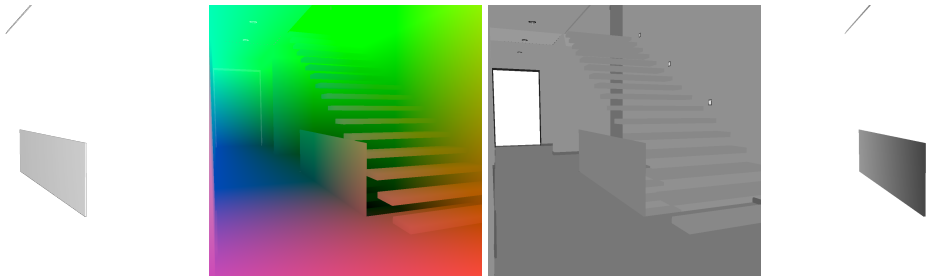
The scene is rendered in two passes, one for the opaque geometries and the other for translucent ones. The opaque geometries pass is done first and outputs a *beauty* texture, the *world positions* texture, the *geometry ID* texture, and a *depth* texture. The beauty texture (fig. 2.6a) is a color render 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.

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.



(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.

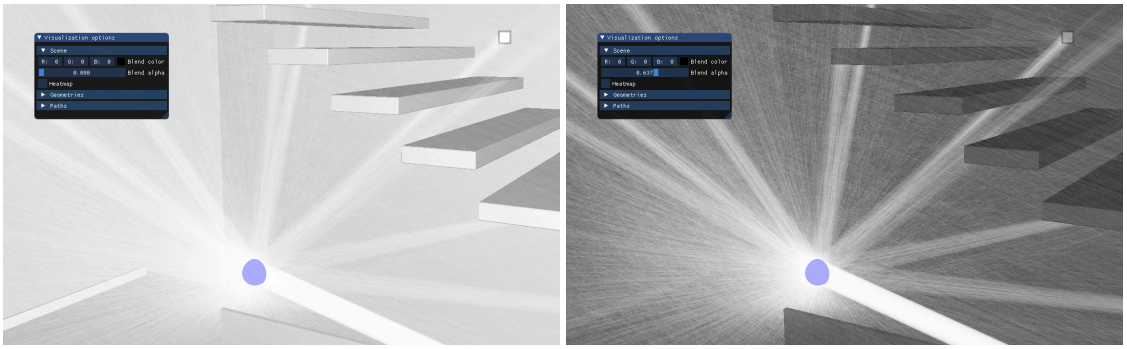
To facilitate scene navigation and readability, some visualization options are provided under the **"Scene"** section of the **"Visualization options"** panel:

"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 2.8.

"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 2.3.4.

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. 2.9a) and a *back* (fig. 2.9b) 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



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

Figure 2.8: Use of the blend color visualization option to improve path visibility in a bright scene.

pass through a single pixel, the OpenGL built-in alpha blending is enabled. By default, all paths are rasterized with an alpha value directly controlled by the **"Paths alpha"** slider on the **"Paths"** section of the **"Visualization options"** panel. In that section there are also three checkboxes controlling other visual properties of the paths:

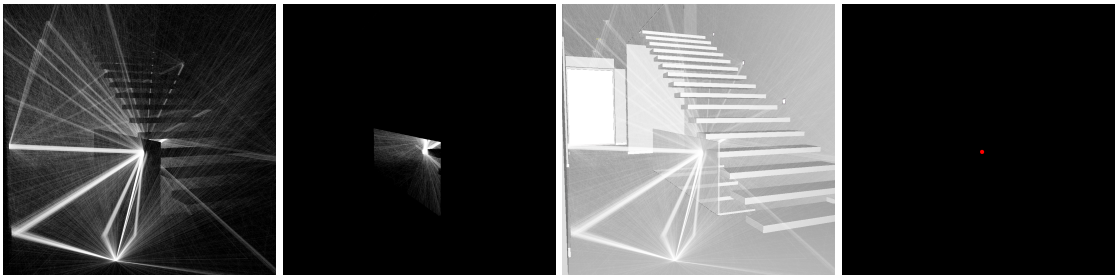
"Render" Toggles the rendering of the paths all together. It might be helpful in situations where the user selected too many paths and moving the viewport camera gets difficult.

"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 useful while determining the origin of fireflies and such rendering artifacts. From the implementation point of view, this is achieved by loading to the GPU a shader storage buffer — `sso` 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 `sso` 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. 2.9b), 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. 2.9a) are super imposed. This leads to the result shown in figure 2.9c. As probably already noticed by the reader, this render pipeline provides a rather simplistic way to solve the so often called "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.

To make filters appear over the paths, they are rendered last. They are first rendered as a mask (fig. 2.9d), 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).



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

Figure 2.9: Textures written by the paths pass with the composited result of the scene renders plus the paths and the filter mask texture.

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.

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.

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[2] (fig. 2.10). 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

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

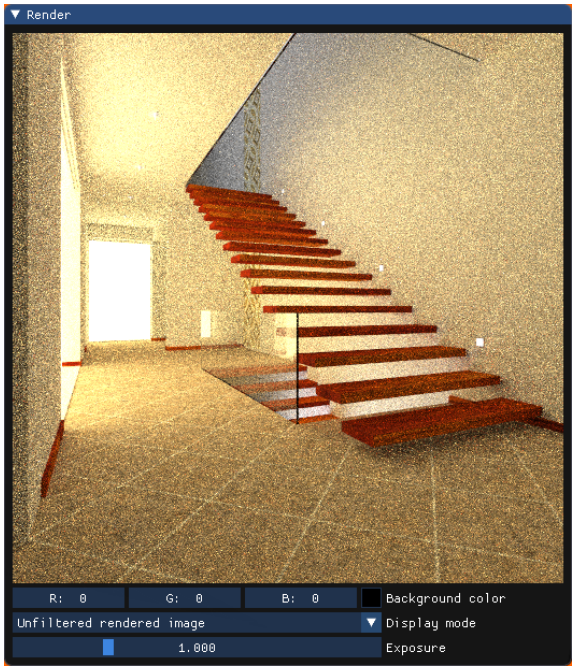
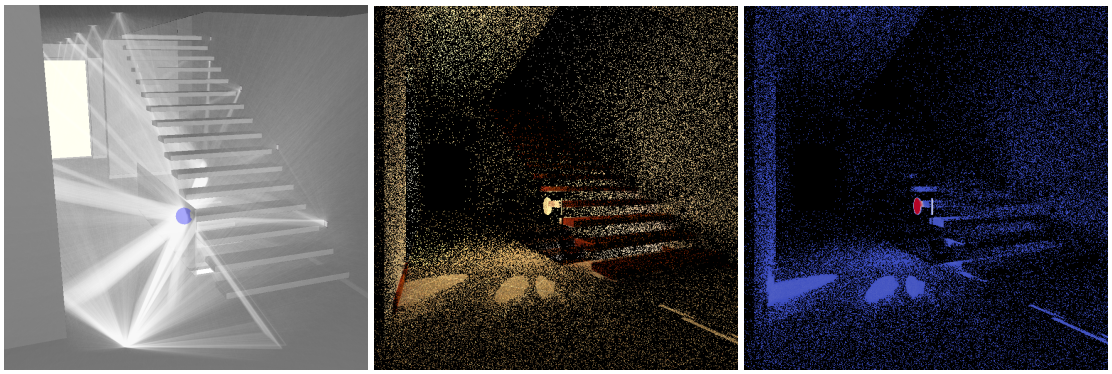


Figure 2.10: "Render" panel in its default state, showing an unaltered render of the *Modern Hall* [Bit16] scene.



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

Figure 2.11: 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).

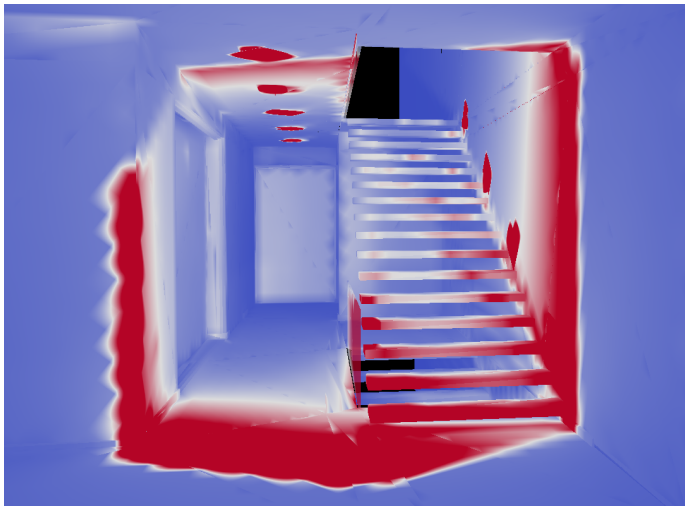


Figure 2.12: Viewport render of the *Modern Hall* [Bit16] scene with the bounce density heatmap relative to a dataset generated by Yocto/GL [PNC19] on top of it.

on it through a fragment shader. The tone mapping described on the OpenEXR documentation[3] 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. 2.11b) 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. 2.11c) 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[4]. 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

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

[4] 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.

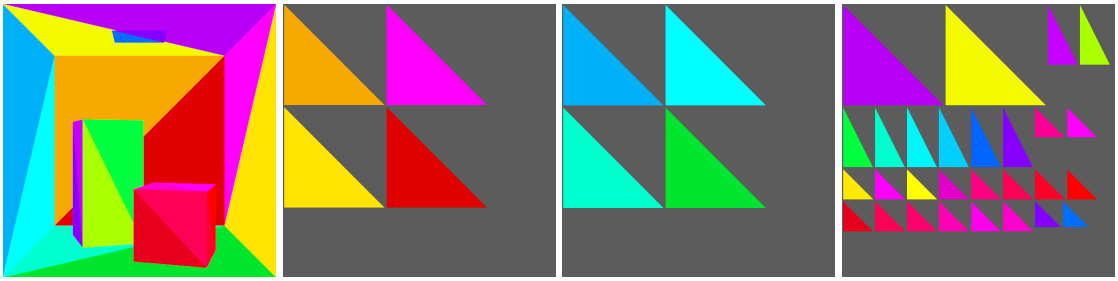
only way in the tool to render paths by actually rasterizing their rays over a scene render; this, as already mentioned before and shown in figure 2.2, gets quickly unmanageable and rather useless due to heavy visual cluttering. This is where the *bounce density heatmap* enters the game. As its name partially suggests, it is a heatmap that shows the density of path bounces on the surfaces of the scene. As shown in figure 2.3.4, 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 an unexpected crowding — or a lacking — of paths on a surface area.

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 to all effect a texture and as such it needs a UV mapping to be correctly applied to the surfaces, a UV mapping generated on runtime upon scene loading. In practical terms, this means displacing all the triangles in the scene on a 2-dimensional plane so that they do not overlap with each other. This is a 2-dimensional knapsack problem and finding an optimal solution for it is no trivial task but, 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.

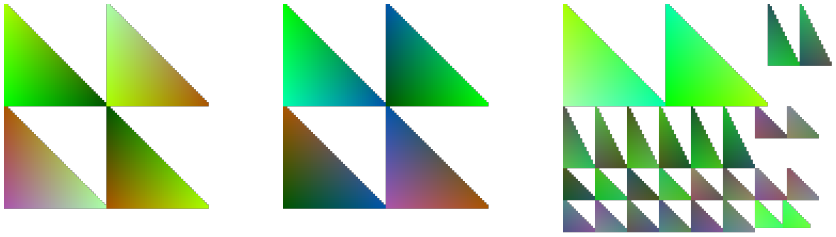
At the end of this operation, the output is equivalent to the one visualized in figure 2.13: a 2-dimensional UV coordinate spanning across several texture squares is assigned to each scene vertex. These coordinates are appended to the vertex buffers on the GPU used for scene rendering.

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 2.14, 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



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

Figure 2.13: 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 2.14: The world positions of the surface points of the *Cornell Box* scene rendered on the UV space shown in figure 2.13. Black point has been set to −1 and the white one to +2 to improve readability.

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. 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 dumps 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 status 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 to 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.

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.

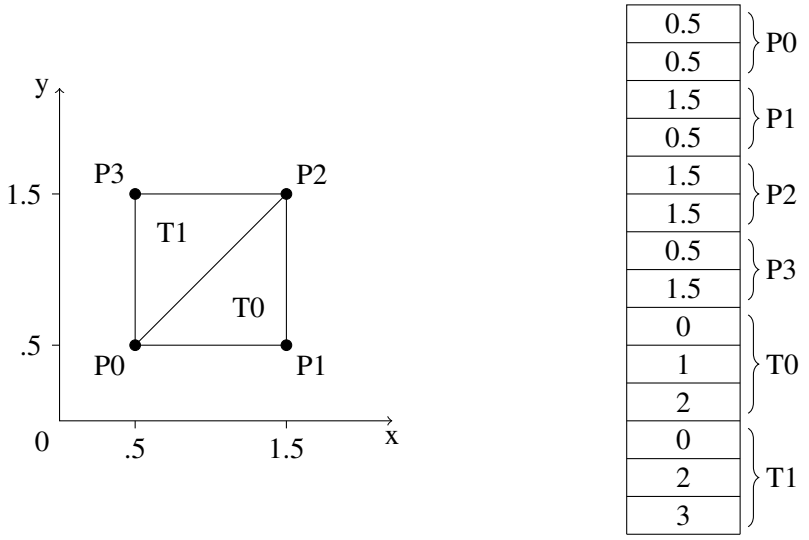


Figure 2.15: Geometry example. To simplify, a 2-dimensional case is presented. On the right a simple triangulated mesh is shown, on the left there is a valid binary file. A corresponding JSON file is shown in Listing 2.3.

Listing 2.3 JSON scene file example relative to Figure 2.15. 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"
        }
      ]
    }
  ]
}
```

3 Use cases

Looking also at the general picture usually helps, for example, looking at figure 2.3.4, it is clear that way more paths ended up hitting the lights on the ceiling and on the staircase rather than the two light portals down the hall

Bibliography

[Bit16] B. Bitterli. *Rendering resources*. https://benedikt-bitterli.me/resources/. 2016 (cit. on pp. 10, 21, 22, 25).

[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. 25).

[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 pp. 10, 22).

[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. 31).

All links were last followed on December 18, 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