Nikolai Hofmann, Marc Stamminger *Erlangen, 03/07/2025*

Global Illumination Tutorials (exercise sheet 4)

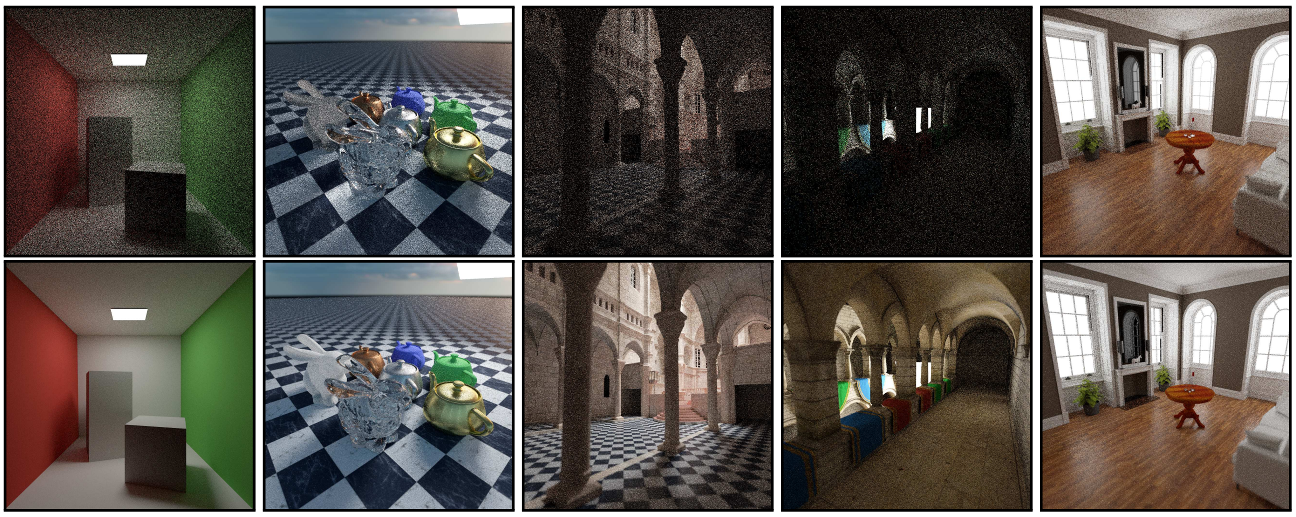

Figure 1: Example renderings using "naive" pathtracing (top row) and path tracing with next event estimation (bottom row). The respective config files from left to right are: `a04_box.json`, `a04_pots.json`, `a04_sibenik.json`, `a04_spz.json` and `a04_fireplace.json` (and their `*_naive.json` counterparts). You may use these to verify your rendered output images against.

In this assignment you are asked to implement a pathtracer, where we combine all of the techniques from previous exercises. We will then extend the algorithm with russian roulette and (optionally) multiple importance sampling. Thereby efficiently solving the rendering equation and producing realistic looking results. We will implement a new algorithm module for each variant, namely in `src/algorithms/naive_pathtracer.cpp` and `src/algorithms/pathtracer.cpp`, while relying on previously implemented functionality. Obviously, you can peek at the code from previous assignments for code snippets.

Lecture: on Tuesdays from 12:15 - 13:45
Tutorials: on Thursdays from 09:00-12:00
Submission deadline: Wednesday 16.07.2025, 23:55
Grading period: Thursday 17.07.2025, 09:00-12:00

We will look into two approaches to solve the recursive formulation of the rendering equation:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega f_r(x,\omega_o,\omega_i) L_o(ray(x,\omega_i), -\omega_i)(\omega_i \circ n) d\omega_i \,. \tag{1}$$

The first ("naive") approach is to bounce the ray through the scene until we accidentally hit a light source. New bounces are importance sampled over a certain distribution (e.g. cosine hemisphere, GGX, ...) and radiance is accumulated when a light source was hit. The second approach is called *next event estimation*, where we force a connection between each vertex and a light source, if visibility permits. The preceding path may additionally be reused in the next iteration of the algorithm for further efficiency. This approach shows drastically increased convergence in most cases due to more light transporting paths contributing to the final image.

Note that, if you want to produce converged and polished renderings for bragging rights, you can switch the "Settings → Beauty render" option via the GUI and wait. It might take a while with the naive approach, however.

Assignment 1 [4 Points] (Naive Pathtracing)
[Files: `src/algorithms/naive_pathtracer.cpp`]

Our goal is to solve the recursive rendering equation, as in Equation 1. Since the recursion depth is infinite, we need to limit the amount of bounces traced. To this end, we simply limit the recursion depth to `context.MAX_CAM_PATH_LENGTH`. In order to find light transporting paths, we will trace "reversed" paths from the camera to the light source. Due to the camera sensor being very small compared to the scene and most light sources, this approach is more sensible than the other way around. We will record the attenuation factor (i.e. "throughput") along the camera path, so when we find a light source, we can compute the amount of radiance transported from the light source back to the camera.

Thus in order to implement this scheme, setup a ray from the camera, bounce through the scene (importance sampled according to the BRDF) and record the throughput until you found a light source, i.e. `hit.is_light()` is true. When a light source was found, compute the transported radiance via your throughput and `hit.Le()` or `Light::Le(...)`, depending on the type of light source, i.e. `hit.valid ? AreaLight : SkyLight`. Terminate the path when either a light source was found, or the maximum recursion depth was reached. Make sure to avoid negative color values and divide-by-zero errors.

Note that our light sources are front-faced only (along the light's normal vector) and we don't need to shoot shadow rays with this approach, since a light source can only be found from viable points in the first place.

Assignment 2 [4 Points] (Next Event Estimation)
[Files: `src/algorithms/pathtracer.cpp`]

You might notice high amounts of noise remaining in the renderings, even after tracing many samples per pixel. To improve upon this and converge to a solution faster, we want to employ next event estimation, i.e. connect each path vertex with a light source and switch to the closed form of the rendering equation (area formulation) when possible. Thus, we will find more contributing paths and converge to a noise-free solution faster.

We now want to enforce a connection between each vertex along a path and a light source. To this end, we rely on the previously implemented light source sampling (`Scene::sample_light_source`) scheme and closed-form integration of direct lighting (`Light::sample_Li`) to compute the incident radiance at a path vertex. After adding the path's contribution, we then "forget" the connection between the vertex and to the light source and continue to bounce the path through the scene, while adapting the throughput, i.e. reusing the previously traced path. Don't forget to trace shadow rays between the current path vertex and the sampled light source in order to only transport light where the light source is actually visible.

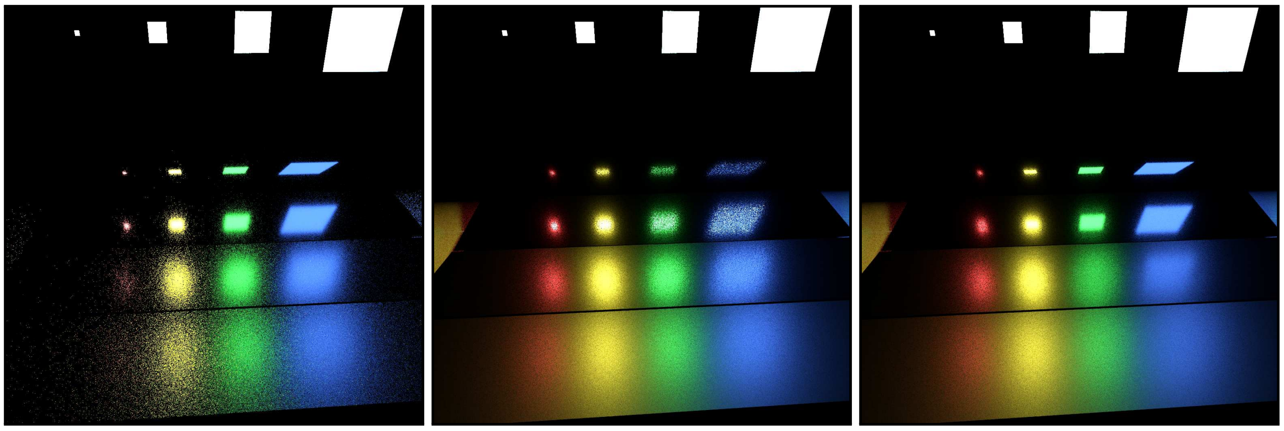

Figure 2: Glossy specular reflection with varying roughness using BRDF sampling, or "naive" path tracing, only (left), next event estimation only (middle) and both strategies combined using multiple importance sampling (right). Scene config file: `a04_mis.json`.

Consequently, if you now happen to directly hit a light source, you may simply terminate the path. Since we're connecting every path vertex with a light source anyways, we may ignore the L_e term in the rendering equation to avoid double counting of emission. Take further care to handle corner cases, such as direct light source or environment light hits, and again make sure to avoid negative color values and divide-by-zero errors.

Assignment 3 [2 Points] (Russian Roulette)

[Files: `src/algorithms/naive_pathtracer.cpp`, `src/algorithms/pathtracer.cpp`]

We obviously want to concentrate computational power on paths that contribute most to the final image. Yet, a high number of paths traced still contain little to no useful information, especially when the throughput is low. Thus, to increase performance, we want a technique to select and cull paths with a low throughput value, while still achieving correct results on average. Simply terminating paths with a low throughput value would yield skewed results, and therefore we apply a small stochastic "trick", called *russian roulette*.

We select some termination probability q, where the integrand is not evaluated and is simply set to a constant value (for example zero). The integrand is still evaluated with probability $1 - q$ and, when weighted by $\frac{1}{1-q}$, which accounts for all skipped samples, it will still yield the correct result on average. Thus, the expected value of the integrator remains unchanged.

Implement the russian roulette operator:

$$F' = \begin{cases} \frac{F-qc}{1-q} & \xi > q \\ c & \text{otherwise} \end{cases} \tag{2}$$

where F is the throughput, q is the termination probability and $c = 0$. Break out of the integration when $F' = 0$. Apply russian roulette after `context.RR_MIN_PATH_LENGTH` bounces and when the throughput is below `context.RR_THRESHOLD` so you are able to tweak its application via the GUI. Note that this optimization applies to both "naive" pathtracing and next event estimation, so you only need to complete one of the above assignments in order to be able to work on this task.

When asked, you should be able to provide answers to the following questions: Does this technique add bias to the renderings and why? Does this technique increase or decrease variance and why? Explain your choice of termination probability and record the speedup you were able to achieve.

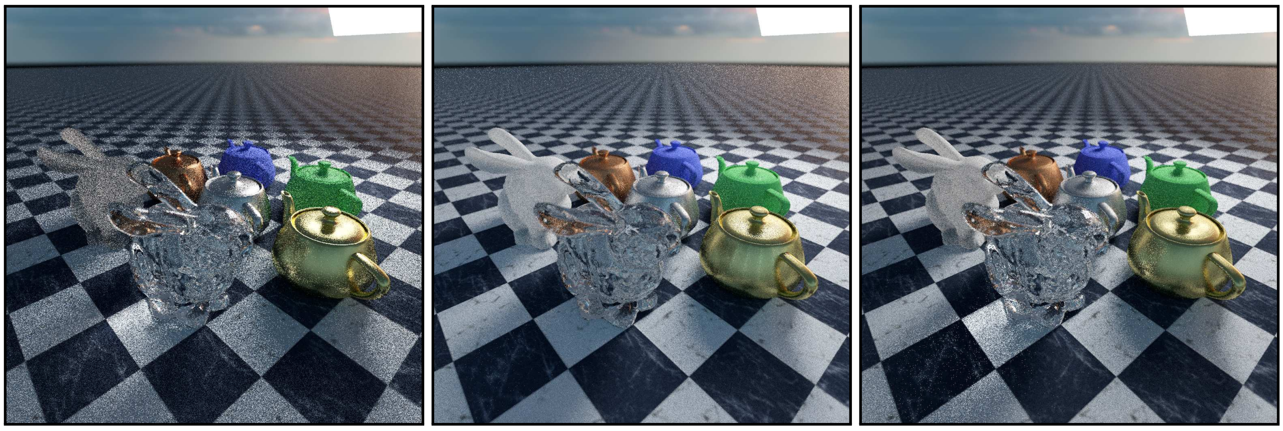

Figure 3: The `a04_pots.json` scene rendered with "naive" path tracing (left), next event estimation (middle) and multiple importance sampling (right). Note how MIS is able to incorporate the strengths of both approaches into one, especially visible via the translucent white bunny (diffuse transmission), the caustics behind the glass bunny, the glossy reflection of the gold teapot and background noise.

Assignment 4 [5 Points] (Bonus: Multiple Importance Sampling)
[Files: `src/algorithms/pathtracer.cpp`, `src/gi/light.cpp`]

With next event estimation, we have previously simply ignored the fact that we're still able to hit a light source directly accidentally. Run the `a04_mis.json` configuration with both your pathtracers and compare to the left and middle images in Figure 2. Note the respective strengths and weaknesses of the two approaches.

In order to efficiently combine both previous approaches, i.e. "naive" path tracing and next event estimation, we can apply multiple importance sampling (MIS). MIS is a technique to combine two sampling strategies via weighting their respective contribution appropriately. These weights are computed via the PDFs of both strategies and the balance or power heuristic (see `src/gi/sampling.h`).

Your task is to add the contribution from direct light source hits to your pathtracer and weight samples from direct light hits and next event estimation appropriately. In order to apply MIS, we additionally need to be able to compute the PDF of both BRDF samples and light source samples independently. For BRDF samples, this is already implemented via `SurfaceInteraction::pdf`. For light samples, you will need to compute the PDF (in terms of solid angle) for both light source types in `AreaLight::pdf_Li` and `SkyLight::pdf_Li`. When incorporated into the pathtracer, you should be able to get similar results to Figure 2 (right) and Figure 3 (right).

Hints: When `SurfaceInteraction::is_light()` is true, use `hit.light` to get a pointer to the light source. The type of the respective light source is: `hit.valid ? AreaLight : SkyLight`. The PDF of selecting a light source can be queried via `Scene::light_source_pdf`. To get an environment light's contribution, use: `hit.light->Le(ray.dir)`. Note that there are some edge cases, such as Dirac delta distributions (i.e. perfectly specular bounces), which can be queried via `hit.is_type(BRDF_SPECULAR)`, or direct light source hits, which require special handling. For such cases, we can simply disable MIS, or set the sample's weight to one.

Happy Hacking! :)