Nikolai Hofmann, Marc Stamminger *Erlangen, 06.05.2025*

Global Illumination Tutorials (exercise sheet 1)

Welcome to the global illumination (GI) tutorials. During the upcoming assignments, you will be asked to implement parts of a C++ GI framework. Some of the topics covered during the exercises will be: random sampling, BRDFs, importance sampling, environment lighting, path tracing, bidirectional methods and volumetric rendering. We'll start with random sampling and area light shading now.

Organizational. There will be 5 exercise sheets, each yielding 10 points. To pass the tutorials, you will need as least **50%** of the points *total*, i.e. 25 points. To submit, register your group of two on StudOn and submit **one** `.zip` file for your group, containing all the files you've worked on, and potentially a `.pdf` or rendered images. In order to be graded, your group needs to upload your solution before the submission deadline **and** check in with us during the grading period. Grading will take place on Thursdays after each submission deadline, where you will be asked to explain your solution and possibly answer some additional questions. There is a link to a collaborative Stuve-Pad on StudOn where you can schedule your group for a timeslot during the grading period to organize things a bit. Please let us know (nikolai.hofmann@fau.de) should you not be able to attend during a grading period, so we can arrange an alternative.

Lecture: on Tuesdays from 12:15 - 13:45
Tutorials: on Thursdays from 09:00-12:00
Submission deadline: Wednesday 21.05.2025, 23:55
Grading period: Thursday 22.05.2025, 09:00-12:00

Build. We use CMake as build system and support the CIP pools out of the box. The exercises are guaranteed to run in the CIP pools, if you want to run it on your own machine, good luck. :) See the provided `README.md` for specific build instructions. There are no restrictive hardware requirements, anything with semi-recent CPU should be able to run the framework. Should you encounter any problems, ask away in the StudOn forum, so other students may help you as well, or drop us a mail. Note that you can use `/proj/ciptmp/<your-IDM-ID>/` to avoid quota problems in the CIP.

Code. We will use the same codebase in different states for each of the five assignments, so spending some time upfront to read the code is probably not a bad idea. The framework is roughly split into three parts: the driver (`src/driver`), the GI core (`src/gi`) and GI algorithms (`src/algorithms`). You can basically ignore the driver part, as we will only be working with the core and algorithms. Note that, if no display is available (e.g. when connected to the CIP pools via SSH), you can still render using the command line, albeit without using the live preview. Additionally, the current state of the scene may be always imported and exported via simple JSON files (see `src/configs`). These can be loaded via drag-and-drop, or the command line. You can move the camera using the `WASDRF` keys and rotate via clicking and dragging the mouse. A lot of state, such as camera parameters, materials, light sources, etc., may also be changed at runtime via the GUI in the top bar.

Browser Version. For following the lecture and comparing your solution against a reference, we provide a web-based version of the framework: `https://lehre.lgdv.tf.fau.de/gi`. Note that all scene data must be downloaded upfront, which is roughly 500MB in size.

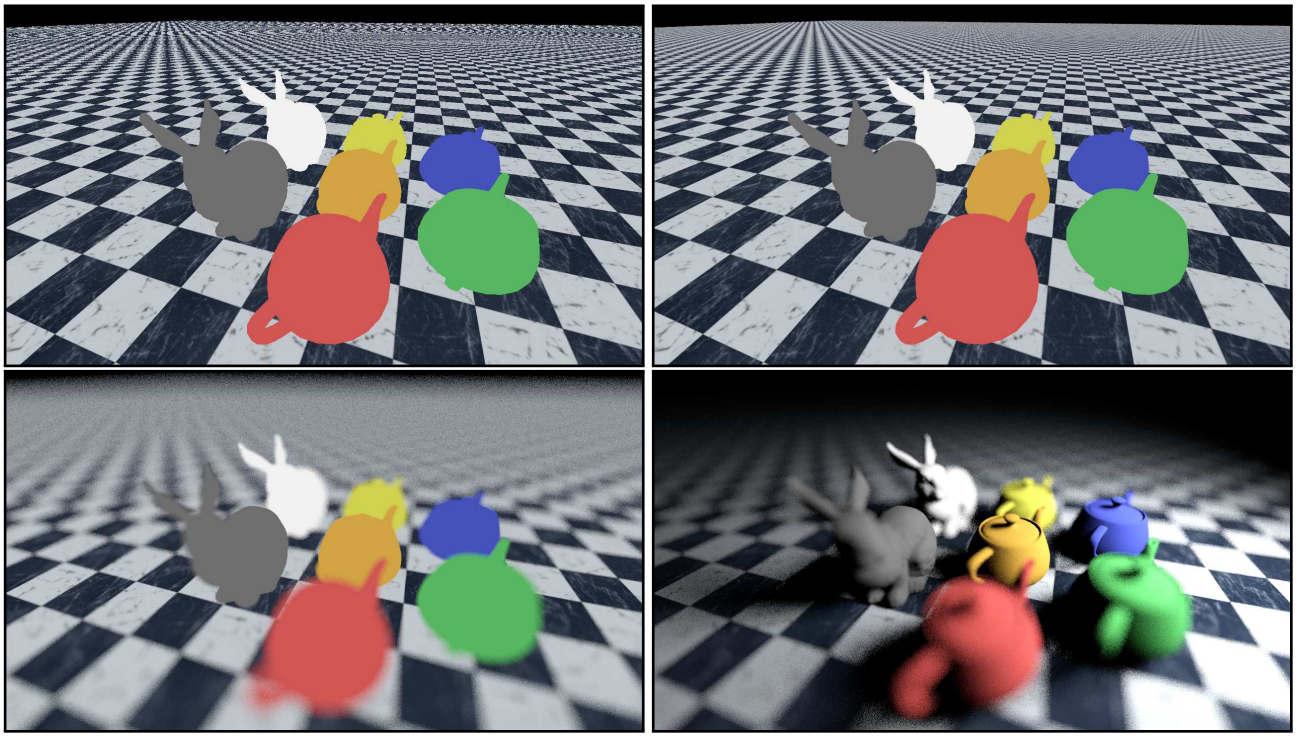

Figure 1: Reference images for various stages of this exercise: start (top left), with anti-aliasing (top right), with depth of field (bottom left) and the final result, with area light shading (bottom right).

Assignment 0 (Download, build and run the framework)

First things first, download `dependencies.zip` and `data.zip` from StudOn. Extract both archives in either the root folder of your assignment source tree (where the directories `src` and `configs` are located), or one directory above to avoid copying them for each assignment. Configure CMake and build the framework (see `README.md` for more specific instructions). Execute the resulting `gi` binary, which is also located in the root folder, and provide the config file `configs/a01.json` as argument. You can also drag and drop config files into the preview window, if available. Have a look at the resulting image (Figure 1 top left) and the code you just ran (`src/algorithms/simple.cpp`) and go from there.

Assignment 1 [2 Points] (Random sampling)
[Files: `src/gi/random.h`]

You may have noticed severe aliasing artifacts along the checkerboard pattern in the background. To improve upon this, we will now add anti-aliasing by simple supersampling of the image plane, i.e. shoot multiple (jittered) rays through each pixel and average their results. In order to do so, however, we first need to implement sub-sampling of a pixel, i.e. generate samples from some two dimensional distribution.

You are now asked to implement different sampling strategies in `src/gi/random.h`. The simplest form of random sampling is just drawing samples from an uniform distribution (i.e. "truly random"). See the first image of Figure 2 to compare your outputs to. Another method, called *stratified sampling*, is to divide the region into several equally sized subregions, called *strata*, and place one sample in each. You are asked to implement stratified sampling in 1D and 2D, by using a uniform sample to jitter a sample inside its strata. For reference, see the second image from the left in Figure 2.

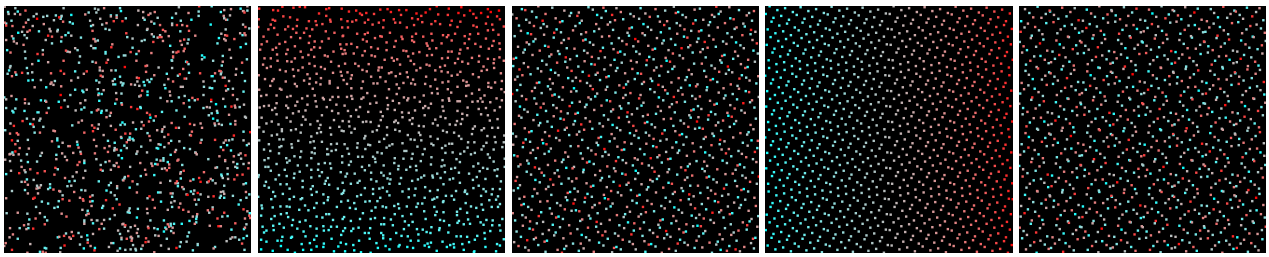

Figure 2: Different 2D sampling strategies from left to right: Uniform, Stratified, Halton, Hammersley, Low-Discrepancy. The first sample drawn is shaded in cyan, the last one red and linear in between.

Another method to get evenly spread samples, are *pseudorandom* sampling sequences. Probably the most common one is the Halton sequence, which `OpenGL` uses internally for their implementation of `MSAA` for instance. Further examples are the Hammersley- and 02 low-discrepancy sequences. Implement those techniques in the `HaltonSampler2D`, `HammersleySampler2D` and `LDSampler2D`. You won't need to implement the mathematical foundations, though, as they are already provided to you via the `halton`, `hammersley` and `sample02` functions, respectively. You can compare the automatically generated output from your implementation (see command line) against Figure 2.

Assignment 2 [3 Points] (Anti-aliasing and depth of field)
[Files: `src/algorithms/simple.cpp`, `src/gi/camera.cpp`]

To actually put the previously implemented sampling methods to use during rendering, you'll need to adjust the function `Camera::perspective_view_ray` in `src/gi/camera.cpp`. Apply the given random sample (parameter `pixel_sample`) instead of shooting each ray through the pixel center. Now, turn back to the algorithm module in `src/algorithms/simple.cpp`. The function `sample_pixel` is called for each pixel, with the desired amount of samples per pixel as argument. Initialize a `Sampler` of your choice with the required amount of samples and configure the renderer to shoot `#samples` view rays instead of just one, while passing a unique random sample to `Camera::view_ray`. When called multiple times, the framebuffer will automatically average the per pixel colors in `Framebuffer::add_sample`, thus resulting in an anti-aliased image. You can now compare your results with the top-right image in Figure 1.

When setting up the view ray, you may also pass an optional second random sample for the depth of field (*DOF*) effect. Thus, create a second `Sampler` for depth of field and have a look at the `Camera::apply_DOF` function, where you are asked to implement the thin lens approximation[1]. To this end, you need to jitter the ray origin on a disk of `lens_radius` size, while updating the ray direction accordingly to converge at the plane of focus. This concept is illustrated in Figure 3.

Assignment 3 [3 Points] (Direct illumination and area light sampling)
[Files: `src/gi/light.cpp`, `src/algorithms/simple.cpp`]

We now want to add direct illumination to our renderer via simple numerical integration of an area light source. To this end, we will, for now, use a simplified formulation of the rendering equation:

$$L_o(x, \omega_o) = \int_\Omega k_a V(x, \omega_i) L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i \ , \tag{1}$$

[1] `https://www.pbr-book.org/3ed-2018/Camera_Models/Projective_Camera_Models#TheThinLensModelandDepthofField`

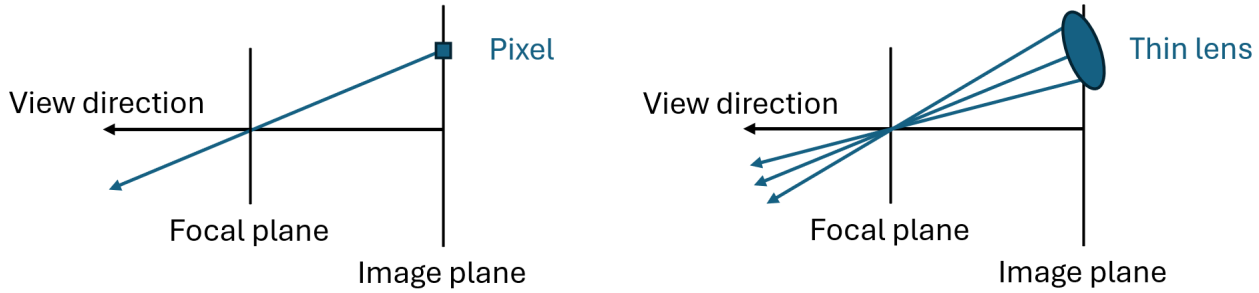

Figure 3: Simplified illustration of a simple camera model without DOF (left) and with the thin lens approximation (right). In a ray tracer, we solely require the aperture size (lens radius) and distance to the plane of focus (focal depth) to implement this model. To this end, we jitter the view ray's origin on a circle (thin lens) oriented around it's direction using random sampling. Subsequently, we re-adjust the ray's direction in order to ensure that it intersects the original ray exactly at the plane of focus.

where L_o and L_i are the outgoing and incoming radiance (or radiance exitance and irradiance more specifically), k_a the surface albedo, $V(x, \omega_i)$ a binary visibility function, ω_i the direction to the light source, x the point to be shaded and n the shading normal. First of all, compute the irradiance (L_i) in `AreaLight::sample_Li`, which is computed via solid angle integration:

$$L_i(x, \omega_i) = k_e \frac{A(-\omega_i \cdot n_l)}{r^2} , \tag{2}$$

where k_e is the emitted light of the surface, A the surface area of the emitting light source, n_l the surface normal of the light source and r the distance to the emitter. Initialize and return the shadow ray for later visibility testing here as well.

Now, back in the algorithm module, setup two additional `Samplers`, one for selecting a light source and one for sampling the light source. Use `Scene::sample_light` to select a light source and your previously implemented `Light::sample_Li` to sample the selected light source (and setting up a shadow ray). Finally, use `Scene::occluded` to test for visibility along the shadow ray. You may simply ignore the returned `pdf` variables for now. Compute L_o via the above equation and add the result to the framebuffer. You should now see smooth shading and soft shadows, comparable to the bottom right image in Figure 1. Take care to handle edge cases, such as regions behind the light source, as well.

Assignment 4 [2 Points] (Evaluation of sampling techniques)
[Files: `src/gi/random.h`, `src/algorithms/simple.cpp`]

If you haven't done so already, have a look at the output from all of your implemented samplers and verify the results against Figures 1 and 2. Note that Figure 1 was rendered with uniform samplers only. What are some obvious advantages and disadvantages of each of these sampling techniques? Try to render images while using only uniform, only stratified and only Halton samplers and compare the results by flipping between the images. Are the outputs identical and if not, why? Are all drawn samples statistically independent or are they correlated, and how does that (positively or negatively) impact image quality? Please prepare answers to these questions for grading and prepare some renderings to support your assertions.

Happy Hacking! :)