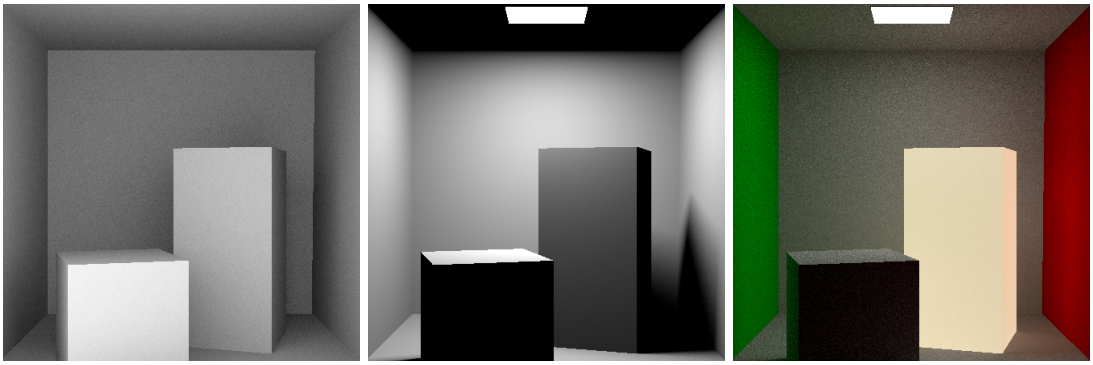

Assignment 1: Monte Carlo Integration and Path Tracing

Deadline: 2021-04-18 23:59

In this assignment you will implement all of the crucial parts to get a Monte Carlo based rendering system. The result will be 1. an ambient occlusion integrator, 2. a direct light renderer, and 3. a simple path tracer. The assignments build up upon each other, be sure to test everything before continuing. For most points in this assignment you can ignore the material BRDF and just assume white diffuse materials ($\rho = \{1, 1, 1\}$).

We have updated the `assignments` repository. Please merge all upstream changes before starting to work.

```
git checkout master
git pull
git merge submission1       # just to be sure
git push                    # just in case something fails, make a backup
git remote add upstream git@submission.cg.tuwien.ac.at:rendering-2020/assignments.git
git pull upstream master
# resolve any merge conflict, or just confirm the merge.
git push
```

Important: As you have seen in assignment 0, you have to register a name for your integrators (and any other additions) with Nori framework. Our test system expects pre-defined names and attributes when invoking Nori via your solution. Please study the given scene xml files and choose the correct names for registration. It is recommended that you run the test files for yourself before submission.

1 Completing Nori's MC Intestines

Nori is an almost complete Monte Carlo integrator. But we have left out some crucial parts for you to complete. By doing so, you'll get a short tour of the main MC machinery.

The main loop structure of our renderer looks something like this:

```
/* For each pixel and pixel sample */
for (y=0; y<height; ++y) {
   for (x=0; x<width; ++x) {
      for (i=0; i<N; ++i) { // N = Target sample count per pixel
         ray = compute_random_camera_ray_for_pixel(x, y)
         value = Li(ray, other, stuff)
         pixel[y][x] += value
      }
      pixel[y][x] /= N
   }
}
```

Obviously, the code will be slightly different longer in practise due to parallelisation, filtering (something we will learn later) and general architectural design. Look into the code, try to understand how things are done and complete the following functions (all changes are a single line):

main.cpp, renderBlock() Iterate over all required samples (target count stored in `sampler`)

block.cpp, ImageBlock::put(Point2f, Color3f) Accumulate samples and sample count

block.cpp, ImageBlock::toBitmap() Divide RGB color by accumulated sample count (look at Color4f, if the count is in member `.w`, there is a function you can use)

For the normals integrator from last time, these changes shouldn't make a difference. However, for the techniques that you will implement in this assignment, they provide the basis for proper MC integration to resolve the noise in your images. Beyond implementing them, make sure that you understand how they interconnect and how Nori converts ray samples into output pixel colors.

As mentioned during the lecture, beyond the main loop you do not need another sample generating loop inside the integrators. If you were to do that in a path tracer, there would be the problem of an ever-exploding number of samples (curse of dimensionality).

2 Ambient occlusion (3 easy points)

Implement ambient occlusion! Its rendering equation is

$$L_i(x) = \int_\Omega \frac{1}{\pi} \mathrm{V}(x, x + \alpha\omega) \cos(\theta) \, \mathrm{d}\omega, \tag{1}$$

where L_i is the brightness, x a position on the surface, V the visibility function, α a constant, and θ the angle between ω and the surface normal at x. The visibility function is 1 or 0, depending on whether the ray from x to $x + \alpha\omega$ reaches its destination without interference. This is also commonly referred to as a shadow ray. α should be configurable via XML and default to `scene->getBoundingBox().getExtents().norm()` if no value is provided (experiment with it!). $\frac{1}{\pi}$ represents a simple white diffuse BRDF, as we explained in the lecture about light when we talked about the furnace test.

For integration, you should sample the hemisphere surface around point x uniformly. Since Nori's main loop already takes care of computing the mean for MC integration, the function should return one sample of the integrand, divided $\mathrm{p}(x)$. The proper value for $\mathrm{p}(x)$ for uniform sampling was discussed in the lecture. In addition, you will need a function that can turn a uniformly random 2D value between 0 and 1 into a uniform hemisphere sample ω. This transformation is called warping. You can draw the 2D random values from the `sampler`. Apply the formulas from the lecture or look at `Vector3f Warp::squareToUniformHemisphere(const Point2f &sample)` inside `warp.cpp` and `warp.h` to generate ω. Make sure to bring ω to world space before tracing (`.shFrame.toWorld`).

Pay attention to the individual mathematical factors (including those inside $\mathrm{p}(x)$), some of them cancel out and don't need to be computed at all!

Altogether, this should be about 20 lines in a new `integrator_ao.cpp` file (not counting boiler plate code). Compare results with different sample counts (16, 64, 256...), do you see an improvement? If not, go back to Completing Nori's MC Intestines!

3 Direct lighting (up to 9 Points)

Check the slides about light and the recaps in Monte Carlo integration and Path Tracing for the correct integrals. There are two possibilities on how to implement direct lighting: hemisphere sampling and light source sampling. Hemisphere sampling works well only for very very large lights (sky), while light source sampling works especially well with small lights. To make sure that both methods can be used, our scenes will contain area lights. If we had point or directional lights, hemisphere sampling would not work and we could only use light source sampling (can you guess why?). All these sampling methods can be combined using MIS (you will learn about that later).

You should start with uniform hemisphere sampling (it's very similar to ambient occlusion in terms of code structure). Once hemisphere sampling works, you can continue with light source sampling and check whether the two methods converge to the same image when using a high number of samples. If they don't, you have a bug, since both rendering methods are based on the same physical concepts and should eventually produce the same image (although one might be noisier than the other with low sample counts). You may also try our provided unit tests locally (maybe you have to edit the python script to correct the scene file lookup path).

3.1 Hemisphere sampling (3 easy points)

You should base your code on `integrator_ao.cpp` and implement it in `integrator_direct_lighting.cpp`.

Task 1 Implement the emitter interface (create either a `parallelogram_emitter` or `mesh_emitter` class) and the supporting machinery. Emitters need to read their brightness (radiance) and colour from the scene file and store it (minimum requirements for an emitter). A name and debug info might also be good. If you don't plan to implement light source sampling, you can use a dummy implementation for `Emitter::pdf()` and `Emitter::sample()`.

Task 2 Implement the integrator. First, you need to check whether the camera ray directly hits a light source (emitter). If so, return its colour and be done. This is not completely correct, but you can ignore direct illumination of light sources for now. If you hit a regular surface instead, cast a random ray according to uniform hemisphere sampling, similar to ambient occlusion (no maximum ray length this time!). If the closest intersected object is a light, compute its contribution using the equations from the lecture, otherwise return zero (black). This should only require a small edit from the `ao` integrator.

3.2 Light surface sampling (up to 6 points)

Light surface sampling is important for performant path tracers (it's referenced as "next event estimation" or "direct light sampling" there). In contrast to hemisphere sampling, you are not simply shooting rays around the hemisphere and hope to find light. Instead, you try to connect hit points directly to light sources and check if that connection is possible. If you implement it, you should see improvements immediately. You will need to sample area light surfaces, i.e., you need a function to pick uniformly random points on the surface of each light. There are 2 options, of which you should choose **one** for your implementation:

1. **Parallelogram lights (3 points)** Parallelograms are very easy to sample uniformly, just use a linear combination $k_1 a + k_2 b$ of its side vectors a, b with coefficients k_1, k_2 where $0 \leq k_1, k_2 < 1$. Obviously, this option will restrict you to using rather basic light source shapes in your scene.

2. **Triangle mesh lights (6 points)** This can give very cool results, i.e., imagine a glowing mesh. Mesh sampling is not that hard either: Select the triangle according to its surface area (larger triangles are more often selected). The implementation in `nori/dpdf.h` will be useful here. Once you have selected a triangle, sample a point on it (`http://mathworld.wolfram.com/TrianglePointPicking.html`).

   Be careful when you reuse random numbers! Example: 2 triangles, `s = rand(0, 1) < 0.5` would give you the first triangle. If you want to reuse `s` for sampling the position (after using it for discretely sampling the triangle), clearly you will only ever sample the first half of the first and the second half of the second triangle. In order to avoid artefacts, `s` needs to be shifted and scaled! `DiscretePDF::sampleReuse` is precisely for that. Later on, you could use it for sampling the light as well (it's enough to query one random light per sample if you normalise properly). But if you are uncertain, you can always just draw additional fresh random numbers from `sampler`.

You can get 3 points for parallelogram or 6 points for triangle mesh lights, **but not both**.

Task 3 Implement sampling. The parallelogram, mesh, or emitter classes would be good places (your choice). You need to implement something like `samplePosition` (taking random numbers, returning a position and its surface normal) and `pdf` (taking a position and returning the sample probability density).

Task 4 To pick one of the available light sources for sampling, you will need a list of emitters in the scene. Hook into `Scene::addChild`. In our assignments, surface emitters are always children of meshes. The switch emitter case is for point lights or other emitters without physical surface, you can ignore it for now. Additionally, the emitter object needs a reference to the geometry (mesh or parallelogram, otherwise the sampling code has no data). Don't be afraid to add stuff to headers or create new ones, it's your design now.

Task 5 Implement the direct lighting integrator for light source sampling. Pick a light, either uniformly or according to the emitted light (importance sampling), and then sample a point on its surface. Once you have a point, cast a shadow ray and compute the contribution, if any (f(x) divided by joint pdf). If there are multiple lights, make sure to compensate for the fact that you chose a particular one! Add a boolean property to allow switching between hemisphere sampling and surface sampling.

4 Simple Path Tracing (15 Points + 15 Bonus)

4.1 Implement the recursive path tracing algorithm (8 points)

Create a new integrator and call it `path_tracer_recursive(.cpp)`. Start with a copy of the direct lighting integrator. It might pay off to keep your code clean so you can easily make small adjustments when we improve it in future assignments.

Task 1, Start (5 easy points) Start with the pseudocode from the path tracing lecture slides. Since Nori's main loop has no `depth` parameter, let `Li` be a stub that calls an additional, recursive function that can keep track of the current depth. For the first task, you only have to implement a fixed depth recursion. You can choose to use a constant in code, or a parameter in the scene files, but the default if no parameters are given must be a depth of 3. During development, you should experiment with this number and can observe how the image becomes more realistic as you increase the depth.

Task 2, Russian Roulette (1 easy and 2 normal points) Implement Russian Roulette, with a minimum guaranteed depth of 4. Whether or not Russian roulette is used must be parameterisasble via boolean `rr` from the scene file. You can start with a version that uses a fixed continuation probability in each bounce (1 Point). Check the slides for details.

However, the proper way to do it is to keep track of the *throughput*. With every bounce, the importance emitted from the camera is attenuated, and the probability for continuation should become lower. You should keep track of this throughput in a Color3f vector, and use its largest coefficient for Russian Roulette (2 Points). Check the slides for details.

4.2 Implement and use the Diffuse BRDF / BSDF (2 points)

Encapsulate uniform hemisphere sampling of diffuse materials in `diffuse.cpp`. The test cases already use it, so you can store and use its albedo to generate colour! These 2 points are only valid in conjunction with a working path tracer. Check slides for details.

4.3 Implement path tracing in a loop (5 points)

Every recursive algorithm can be written in a loop as well. Sometimes a stack is needed, but in the path tracer that is not necessary. The loop form is much friendlier to the processor, and you can avoid stack overflows (which could happen with very deep recursions).

The code should be pretty similar. You already keep track of the throughput, if you implemented Russian roulette. Now you should get roughly something like this:

```
Li(Scene scene, Ray ray, int depth) {
    Color value = 0;
    Color throughput = 1;
    // .. some other stuff

    while (true) {
        // stuff
        throughput *= "something <= 1"

        // stuff
        value += throughput * something

        if (something)
            break;
    }
    return value;
}
```

You might *break*, or add things to *value* in more than one place, or in a different order. This is just the basic idea.

4.4 Implement a higher-dimensional path tracing effect (15 bonus points)

Implement either motion blur or depth-of-field effects. For motion blur, you will need to give something in your scene the ability to move (scene objects, camera). For each path, you will need an additional uniformly random time variable `t` and consider it when you perform intersection with your scene. To implement depth-of-field, you will need two additional uniformly random `u,v` variables for each path and consider them in the setup of your camera ray. You can gain 15 bonus points for either effect, **but not for both**.

Submission format

Put a short PDF or text file called `submission<X>` into your git root directory and state all the points that you think you should get. This does not need to be long. Also mention the code files, where you implemented something if it is not obvious.

To store or submit your code, please use our own, institute-hosted submission Gitlab `https://submission.cg.tuwien.ac.at`. You will receive a mail with your account and assignment repository as soon as they are ready. The `master` branch is for development only. You should push there while you are experimenting with the assignment and don't want to lose your work. Once your solution works and you believe it is ready to be graded, please use the branch `submission<X>` where `<X>` is the assignment number. E.g., in order to submit your solution for this assignment, push to `submission1`.

If you push to a `submission` branch, the server will trigger automatic compilation and some testing for your code. You can track the state of new submissions being processed on the GitLab page for your repository under "CI/CD > Pipelines". If a stage fails, click on it to receive additional output and system information from the executing server. If everything worked, you will shortly find a report with your test results in the "CI/CD" pipeline section, when checking the artifacts of the "report" stage. You can submit multiple times until the deadline, but don't clog the system by, e.g., using the submission server for debugging. The last submission that was pushed before the deadline counts, regardless of the results from automatic testing. They are only meant for your convenience and to provide some automated feedback.

Please make sure to NOT add unnecessary files (project folders, temporary compiler results), as your application will be created from your code and CMake setup only. Examples of files that are usually relevant:

- **changed or added** CMakeLists.txt files
- **changed or added** code files (.h, .cpp)
- **changed or added** test cases if you want to show off advanced solutions

Make sure to keep the directory structure in your submitted archive the same as in the framework.

Words of wisdom

- Remember that you don't need all points to get the best grade. The workload of 3 ECTS counts on taking the exam, which gives a lot of points.
- Nori provides you with a `Sampler` that is passed in to the functions that produce the integrator input. Use this class to draw values from a canonic random variable.
- Be careful of so-called "self-intersections". These happen when you immediately hit the same surface that you started your ray from, due to inaccuracies in floating point

computations. You can avoid these by offsetting rays in the normal direction of the surface with a small ϵ. Use `Epsilon` defined in `nori/common.h`.

- Hemisphere sampling and light source sampling are two methods to compute the same integral. Therefore, given enough samples, they both should converge to the same result.

- The framework is using Eigen under the hood for vectors and matrices etc. Be careful when using `auto` in your code (Read here why).

- Please use TUWEL for questions, but refrain from posting critical code sections.

- You are encouraged to write new test cases to experiment with challenging scenarios.

- Tracing rays is expensive. You don't want to render high resolution images or complex scenes for testing. You may also want to avoid the `Debug` mode if you don't actually need it (use a release with debug info build!).

- To reduce the waiting time, Nori runs multi-threaded by default. To make debugging easier, you will want to set the number of threads to 1. To do so, simply execute Nori with the additional arguments `-t 1`.