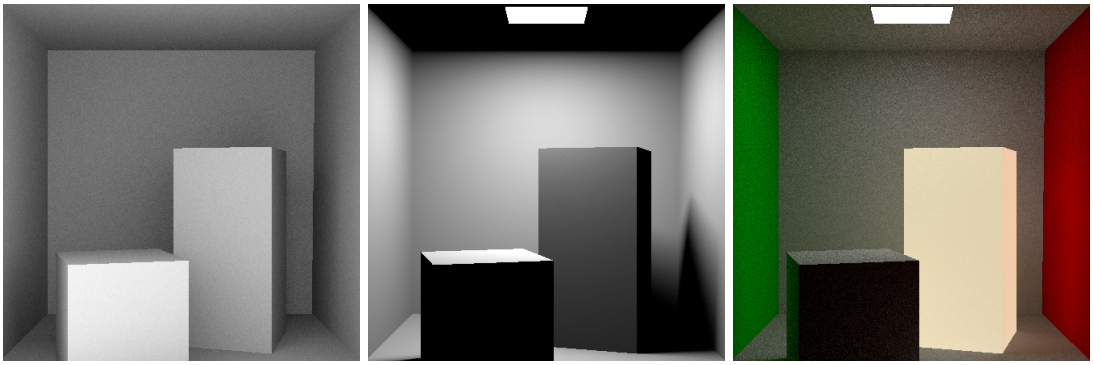

Assignment 1: Monte Carlo Integration and Path Tracing

Deadline: 2022-04-17 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 the first few 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-2022/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 (1 Point)

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 in practice 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 so they work together to perform Monte Carlo integration (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, apart from the main loop and the summing/averaging that happens there, you do not need additional sample/integrate loops inside your integrator functions. 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 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 reflected radiance, 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 directions in the hemisphere 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 f(x), divided by p(x). The proper value for p(x) for uniform sampling a hemisphere is $\frac{1}{2\pi}$. In addition, you will need a function that can generate uniform samples for directions on the hemisphere. This is not trivial, so Nori takes something that is easy to get (a uniformly random 2D value between 0 and 1) and turns it into a uniform hemisphere direction ω. This transformation is called warping. You can draw the 2D random values from `sampler`, and then use `Vector3f Warp::squareToUniformHemisphere(const Point2f &sample)` inside `warp.cpp` to generate ω. Make sure to bring this ω in local space to world space before tracing along it, by using `.shFrame.toWorld`.

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 (4 Points)

Check the slides about light and the recaps in Monte Carlo integration and Path Tracing for the correct integrals. Solve direct lighting by uniformly sampling the hemisphere around the first hit point a ray into your scene intersects with. Hemisphere sampling works nicely for very large lights (sky), but not so well for smaller lights (takes a long time to give smooth results). Don't worry about this yet, but make sure you can confirm it for yourself, and figure out why. You should base your code on `integrator_ao.cpp` and implement it in `integrator_direct_lighting.cpp`.

Task 1 Implement the emitter interfaces. The test cases you receive have two types of emitters: `parallelogram_emitter` and `area`. Some objects in your input scenes will be assigned these types and corresponding parameters. `parallelogram_emitter` are emitters that should only be tied to meshes in the shape of a parallelogram, `area` may turn any mesh into a (complex) light source. However, in the first assignment, there is no real difference between the two. Both types of emitters need to read their brightness (radiance) and colour from the scene file, and store it. The emitter interface has multiple parts to it, but for now, all you need to do for both types is make sure that you can read their radiance from the scene file and access it during rendering via the `eval`. You can use a dummy implementation for `Emitter::pdf()` and `Emitter::sample()` for now. You will complete these in a later task, and they will be different for `parallelogram_emitter` and `area`.

Task 2 Implement the direct lighting integrator. First, 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 for this task it is fine). If you hit a regular, non-emitting surface instead, cast a new, random ray according to uniform hemisphere sampling, similar to ambient occlusion (but no maximum ray length this time!). If the closest intersected object with this new ray is an emitter, use your emitter implementation from Task 1 to compute its contribution using the rendering equation, otherwise return zero (black). We assume that our light sources are all lambertian, therefore the radiance L_i coming from a light in the rendering equation is just its radiance. This should only require a small edit from the `ao` integrator.

4 Simple Path Tracing (15 Points + 17 Bonus)

This will be the first version of your path tracer. Based on the rendering equation, you will get your first images with indirect lighting, shadows and multiple light sources.

4.1 Implement the recursive path tracing algorithm (10 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 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, Implement and use the Diffuse BRDF / BSDF (2 Points) Encapsulate uniform hemisphere sampling for diffuse materials in the BSDF implementation `diffuse.cpp`. The test scenes already apply this material to the objects in the scene, so you can read and use the `albedo` member to render in colour!

Task 3, Russian Roulette (1 easy and 2 regular Points) Implement Russian Roulette, with a minimum guaranteed depth of 4. Whether or not Russian Roulette is used must be parameterisasble via a boolean parameter `rr` from the scene file. If it is not used, fall back to fixed number of recursions. You can start with a version that uses a fixed continuation probability in each bounce (1 Point). The generated test outputs you get in your reports will actually be using a fixed value of 0.7 continuation probability. 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. Note that if you do this, your solution will look slightly different to the report reference. This is fine!

Feel free to also explore ideas that we didn't describe here (rays that miss are black by default, but you could use a sky colour or an environment map). These things do not go unseen :)

4.2 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.

4.3 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**.

4.4 Be patient (2 Bonus Points)

The path-traced images you get with the provided test scene configurations are very noisy. How long does it take on your machine to compute them? How much longer do you think it would take until you get a quality that you are happy with? Experiment with the number of samples and report if the development matches your expectations. Given that our scenes are extremely simple, do you think that with this kind of performance it is feasible to render entire *movies*?

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`.