Assignment 3: Materials and Importance Sampling

Deadline: 2022-05-24 23:59

In this assignment, you will extend the Monte Carlo rendering system from the previous assignment with basic materials, importance sampling of various functions and next event estimation. In the above image, we see what these methods can do: the left scene is rendered with uniform hemisphere sampling. In the center, we use cosine-weighted hemisphere sampling (importance sampling). On the right, we use next event estimation to perform surface sampling in a recursive path tracer. Images were rendered with the same number of samples per pixel (32).

This assignment is quite long (more than 50 points in total), but don't feel pressed to implement everything. You should read the sheet carefully and then choose whatever is most interesting to you, but also note that some steps build on others (e.g., you can't implement next event estimation without solving surface sampling first).

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

```
git checkout master
git pull
git merge submission2       # 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.
git push
```

We also provide a reference implementation for assignment 2, you can download it from TUWEL.

1 Sample Warping (3 easy points, 7 bonus points)

Random numbers are often generated uniformly in the range between 0 and 1. We can combine multiple such random numbers to sample cartesian domains uniformly, but different distributions are needed, e.g., to get uniform distribution in a non-cartesian domain (for recursive rendering, we need to sample the hemisphere for instance), or for importance sampling techniques.

The process of *changing* one distribution to another is called warping. In this assignment, you will start with easily obtainable, canonic random inputs, and convert them to new, useful distributions. The input to all warping functions are two uniformly distributed ([0, 1)) random numbers, and the output are samples on the target domain. The input is always a 2D vector with values of two canonical random variables ξ_1, ξ_2.

In many cases, we may have an existing sample and need to obtain its PDF value for a given sampling strategy, thus a method to produce the PDF from input samples is also required. The input is always a sample x, 2D or 3D, for which a PDF value $p(x)$ value should be computed.

To visualize and check your implementations, we will be using the `warptest` executable, which is part of the Nori framework. You should complete several of the warping functions that it tests. For an introduction on how to use `warptest` and what each distribution is supposed to do, please refer to the Assignment 3, Part 1 from the Nori home page (`https://wjakob.github.io/nori/`). **Note that our scoring system is different, please find it below**. This task can be fully solved in `warp.cpp`. `SquareToUniformHemisphere` is already there, some of you were already cleverly using it in the first assignment to do uniform hemisphere sampling.

squareToTent 1 point, test your basic Monte Carlo sampling knowledge, **bonus**

squareToUniformDisk 1 points, **required**

> **Sampling:** Use the input canonic variables to generate samples (r, θ) in polar coordinates where $r \in [0, 1)$ and $\theta \in [0, 2\pi)$, such that they are uniformly distributed on a disk when transformed to Cartesian coordinates (x, y). Return the current sample (x, y) at the end of the function body.
>
> **PDF:** The input is a 2D vector with a sample location in Cartesian coordinates

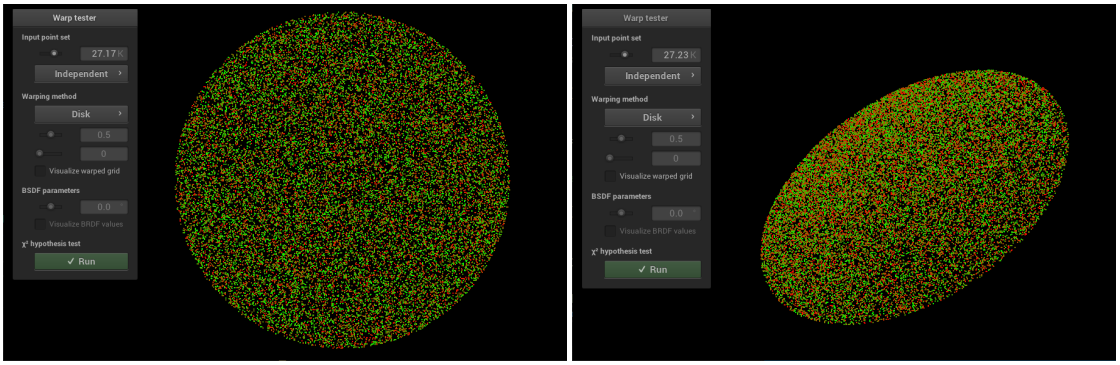


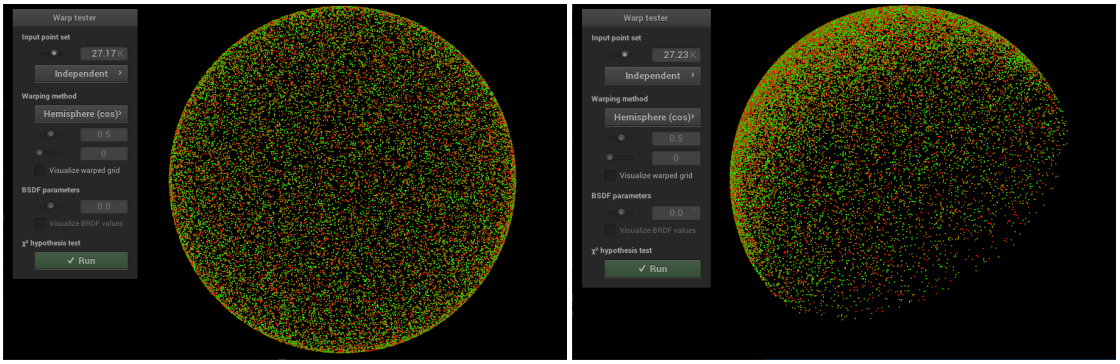
Figure 1: Reference solutions for uniformly distributed samples on the unit disk

(x, y) on a square, with $x, y \in [-1, 1)$. Return the proper value for the corresponding result from the uniform distribution PDF $p(x, y)$ on the disk. Note: For a uniform distribution, the PDF is constant. Just make sure that the sample location is valid!

squareToUniformSphere 1 point, can use it to implement spherical lights, **bonus**

squareToCosineHemisphere 2 point, **required**

Sampling: The input is a 2D vector `sample` that holds values of two canonical random variables ξ_1, ξ_2. Use them to generate samples (θ, ϕ) on the unit hemisphere such that they have a distribution proportional to $\cos(\theta)$ (i.e., more samples the closer we get to the pole of the hemisphere) and convert them to ω with the transformation for spherical coordinates. Return the sample (x, y, z) at the end of the function body.



PDF: Input is a 3D vector with a sample location ω on the unit **sphere** in Cartesian coordinates (x, y, z) with all values in range $[0, 1)$ and $\sqrt{x^2 + y^2 + z^2} = 1$ (z is up). Return the appropriate result for the PDF value $p(\omega)$. Compute and return the appropriate result for a PDF with distribution $p(\omega) \propto \cos(\theta)$.

squareToBeckmann 5 points, used for some *microfacet* materials, **bonus**

2 Light surface sampling (3-6 points)

Extend your direct lighting integrator to support emitter surface sampling. This is a prerequisite for next event estimation and multiple importance sampling.

Light surface sampling is important for performant path tracers. 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, i.e., you switch from a hemisphere to an area integral.

For this, you will need to sample area light surface areas, hence 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)** Works for any light source that can be described by a parallelogram. Parallelograms are 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 limit possible 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 triangles in a mesh, either uniformly or according to their 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`).

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

Task 1 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 2 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 3 Implement the direct lighting integrator for light source sampling. Pick a light source, 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). The PDF here should reflect each choice you made in the current sample, to compensate for all the choices you did not consider: the selection of one light source from the available ones, for mesh lights the choice of one triangle from all available ones, and the choice of a single sample point on the surface area. Add a boolean property `surface_sampling` (default: `false`) which should be parsed from the scene files to allow switching between hemisphere sampling and surface sampling.

3 Materials (15 Points)

3.1 Mirror BSDF (3 easy Points)

The mirror BSDF reflects the incoming ray about the normal. All light (and importance) is reflected in exactly this (and only this) mirror direction. This has several implications:

- A beam of light reflected off a mirror surface will retain all of its radiance. Technically this means, that the BRDF of a mirror is actually $1/\cos\Theta$. In Nori, however, the cosine term is computed in the `BSDF::sample` function for all materials. We can therefore omit the computation for the cosine in `BSDF::sample` and just return 1.

- The PDF is a Dirac delta function (spike in a singular location with infinite height, which integrates to one). When querying for the `pdf` or the `eval` function, we hence just return 0 (in theory it is almost surely impossible to generate such the direction where the spike is located by chance, in practice it's super unlikely).

- We specify `bRec.measure = EDiscrete` to indicate to our integrators that this BSDF did not really leave us a choice regarding the direction in which the ray continues. This is needed for handling special cases (see next event estimation).

Implementing the mirror gives you **3 points**, but enables you to gather more points for MIS and next event estimation.

3.2 The Dielectric BSDF (9 normal Points and 3 hard ones)

A dielectric BSDF can be used to model transparent objects like glass, diamonds or water. Implement it according to the lecture on materials or perhaps the course book PBRT. Use the BSDF in `dielectric.cpp` to make your solution accessible from scene files. Note that

different textbooks use different conventions for the directions and indices of refraction that they reference. You can use any convention you like, but the setup of Nori prefers that `bRec.wi` should be the negative view ray direction. The dielectric BSDF cannot give you the medium of the volume the view ray is coming from and the one it goes to, you should figure this out yourself. It only provides the index of refraction on the exterior and the interior of the object with the given material.

One important note: before, we offset our rays along the surface normal when continuing with the next bounce from a reflective surface to avoid self intersections. But, if you actually want to **enter** an object, this is not a good idea! Instead, offset your rays along the *negative* surface normal. Also, if you want your dielectrics to work with next event estimation, you basically have to treat a hit with them like a hit with a mirror material, because it only reflects / refracts in a single direction.

Implementing until here gives you **9 points**.

While working on dielectrics, you might wonder what the `BSDFQueryRecord::eta` is for. ~~This is only really necessary when you perform Russian Roulette with throughput. When light switches media (e.g. vacuum → glass), the radiance it carries changes. This change of density should be included in the BSDF weight that you return from the BSDF `sample` method. But, if you use Russian Roulette with throughput, then this may erroneously affect your decision to stop, since the throughput is now no longer strictly going down with every bounce, but may in- or decrease somewhat randomly as you switch between media. We can counter this by keeping track of the relative eta in addition to the throughput. After each sampling / evaluation of the BSDF, we can update `eta *= bRec.eta`, and use it to modify the Russian Roulette survival probability to remove the influence on the estimated throughput from switching between media. For this to remain stable in all scenes, make sure that the other supported materials (diffuse, mirror) set a proper `bRec.eta = 1` to avoid unexpected behavior. Implementing this `bRec.eta` business including RR gives you **3 hard points**~~. You wouldn't be alone, we also wonder. It came from upstream, and we couldn't get it to make any difference. If you manage to demonstrate an improvement and explain to us why, you can get **30 hard and risky extra points**. You may have to create your own scene.

4 Importance Sampling (1 easy points, 9 normal points)

4.1 Cosine-weighted sampling of diffuse BSDFs (1 easy points)

Use the cosine-weighted hemisphere sampling method, as described in the lecture. First make sure that your direct lighting and path tracing integrators use the diffuse BSDF class appropriately, then extend the diffuse BSDF with cosine-weighted hemisphere sampling.

Ideally, you can reuse your warping solutions from the first part of this assignment! The BSDF should switch between using cosine-weighted and uniform hemisphere sampling, depending on the value of the `use_cosine` flag provided by each object's material (default: `false`). Note that this affects both the sampling and PDF computation! Confirm for yourself that cosine-weighted hemisphere sampling can reduce the noise in your scenes. To test this, compare the output of the test scenes that end in `uniform` with the ones that end in `cosine`. The latter use cosine-weighted hemisphere sampling and should give slightly cleaner results.

4.2 Next Event Estimation with diffuse materials (4 points)

Implement next event estimation (NEE) for your diffuse path tracer using the 0/1 strategy, i.e., no mixing of sampling strategies. This requires support for light surface sampling.

It should be active depending on a boolean `nee` in the test file (default: `false`). On every bounce, you create one light surface sample, make a ray intersection test from the current position, and compute the contribution. Another ray is then sent out to retrieve indirect light in the next bounce. Ideally, you'll importance sample the BSDF for indirect light to benefit from cosine-weighted hemisphere sampling. Be careful not to erroneously count the emittance twice (i.e., first when doing the light surface sampling and then when hitting a light source randomly). To get a correct image, hit emitters should only be considered on the first intersection. For all other light, the illumination is computed via direct lighting, i.e., one bounce in the future (hence, "next event"). For further details, please see the lecture slides. Just as a heads-up: implementing NEE will dramatically improve the quality of your renderings! In combination with spatial acceleration structures, you should now be able to render impressive scenes fast! To test this, compare the output of test scenes that end in `uniform` or `cosine` with ones that end in `nee`. The latter use next event estimation and should give significantly cleaner results.

4.3 Next Event Estimation with discrete PDF BSDFs (5 points)

Mirrors and dielectrics have a Dirac delta-like BSDF (and associated probability function, Nori calls them **discrete** PDFs or measures).

The key conjecture is that no random hemisphere sampling method could ever hit the singular reflection vector for an incoming view ray by accident, so BSDF sampling is imperative for these materials. If you connected a surface sample to the hit point on such a surface, the BSDF would just say "Nah, no light going through these directions". This is the same as trying to hit exactly 0.5 with a random number between 0 and 1, the probability is 0 (in a computer that would in fact eventually work, but let's not be pedantic).

If you want mirror materials to play nice with NEE, you need to take special care: for any direction that is not explicitly the reflection vector, the sampling probability is 0, so you simply can't do light source sampling on mirrors. But if you just ignore direct light on mirror materials, the light sources will be missing in mirror reflections! Hence, you need to treat this as a special case:

1. Do not perform NEE when on such a surface.

2. If the previously hit surface had a discrete BSDF PDF, then do add the emittance of the current surface. `BSDFQueryRecord::measure` and `EMeasure::EDiscrete` were made for this purpose.

You can achieve 5 points if you make mirror materials work with NEE.

5 Multiple Importance Sampling (MIS, 5 normal points 10 hard points)

MIS is a bit hard to wrap your head around it, but once you do that, you can get quite a light bulb moment. We will try to go slow about it, and divide the implementation into several parts. Again, you will need support for light surface sampling to do this, as well as cosine-weighted hemisphere sampling.

5.1 MIS for Direct Lighting (5 points)

Implement MIS between hemisphere sampling and light surface sampling using the balance heuristic in your direct lighting integrator. Whether or not MIS is used should be parameterizable via boolean `mis_sampling` (default: `false`) in the test files. Choose between the two sampling strategies with equal probability, generate the sample using the chosen method and compute the sample's probability with both methods. You should use the surface's BSDF to generate the hemisphere samples to benefit from cosine-weighted hemisphere sampling if it is enabled by a material. Then use the equations from the lecture to compute the proper MIS weight. Return the contribution that you would get with the chosen method, multiplied by the MIS weight and a straight forward compensation term for the random choice of picking one method over the other. You should use the balance heuristic, simpler heuristics will count but not give full points.

You can test MIS on the `ajax-2lights_dl*.xml` scenes, where you should be able to observe the following: the small area light is better suited for surface sampling, while the larger

one is better with cosine-weighted hemisphere sampling, but MIS can give you the best of both worlds.

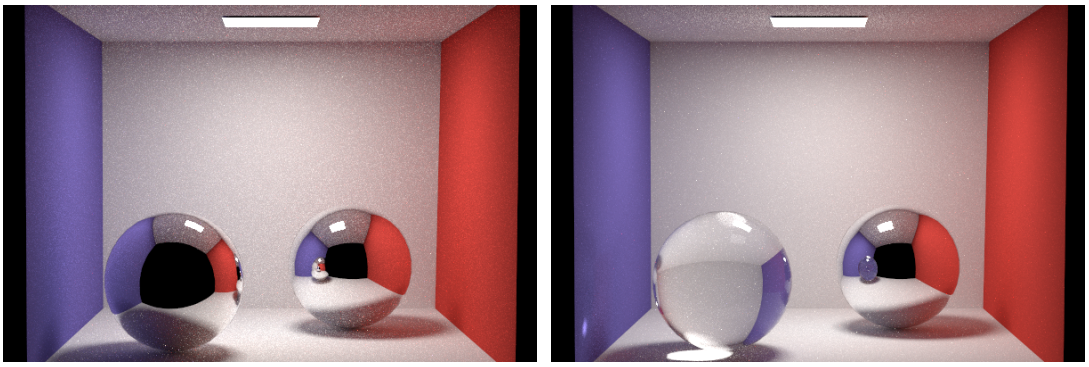
5.2 MIS for Path Tracing with diffuse materials (5 hard points)

If you implemented next event estimation (NEE) earlier (Section 4), the simple 0/1 implementation of NEE is to always choose one strategy for a particular bounce and path. This is already a valid MIS strategy, but it's often not the best one. Implement the balance heuristic for NEE!

This is not a lot of code, but it is quite tricky and intricate. Instead of randomly sampling the strategy, like before, we always choose both strategies. Accordingly, we do not have a correction factor for randomly choosing a strategy. It is also split into two places: 1. When you add the contribution of NEE, and 2. when you add the contribution of a randomly hit emitter. But in principle, it is the same thing: You have to compute both sampling probabilities (of the strategy used, and the strategy not used), and use the balance heuristic equation for computing w_i, then multiply with your contribution $f(x)/p_i(x)$. It's important to use the same *probability space* for both probabilities. You can't put the area probability of surface sampling into one MIS weight together with the solid angle probability of BSDF sampling. Instead, transform the area probability into BSDF probability (change of variables, we showed how to do that).

5.3 MIS for Discrete BSDF PDFs (5 hard points)

NEE required some special treatment for mirrors and dielectrics. This is, because those materials have a Dirac delta probability functions: all rays are directed into a finite set of directions (1 direction for mirrors, 2 for dielectrics). MIS requires a similar special treatment. Implement it!



(a) Rendering with mirror materials

(b) Rendering with mirrors and dielectrics

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 the first 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

- If you are having trouble with performance, consider changing the resolution and/or number of samples for your test cases.
- The warp tests only check if the samples you generate match the corresponding PDFs you define. Best start with the PDFs and then try to match them with sampling.
- Hemisphere sampling, next event estimation and MIS are all methods for integrating the same integral. Given enough samples, they all should converge to the same result.
- If you have questions, please use TUWEL, but refrain from posting critical code sections.
- You are encouraged to write your own 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`.

Appendix: The Phong BSDF

The Phong reflection model is one of the oldest ones, but not physically plausible. Hence we banished it to this appendix (used to be extra points, but not anymore, information only for interested readers). The original Phong was not even energy conserving, therefore we will present the modified Phong (Lafortune and Willems, 1994). That report might be a bit hard to read (but doable, and there are some additional variance reducing improvements), so we will distil everything important into a summary.

Phong is a glossy BSDF, consisting of a diffuse and specular part. The BSDF equation is:

$$\mathrm{f_r}(x,v,\omega) = \mathrm{f_{r,d}}(x,v,\omega) + \mathrm{f_{r,s}}(x,v,\omega) \tag{1}$$

$$= k_d p_d \frac{1}{\pi} + k_s(1-p_d)\frac{n+2}{2\pi}\max(0,\cos^n\alpha), \tag{2}$$

where α is the angle between the perfect specular reflection r_v and ω, k_d, and k_s are diffuse and specular albedo, p_d is the percentage of diffuse reflection (as opposed to specular) and n is the shininess (specular exponent).

The modified phong is not realistic throughout all possible parameter ranges, but it is simple and relatively easy to implement. As with the diffuse BSDF, we need an evaluation, a sampling, and a pdf function. It should be straight-forward to write the evaluation function, sampling is a bit harder.

Because we want to be efficient, we will again try to importance sample this BSDF. At the beginning, we stochastically choose between sampling diffuse and specular part based on p_d. The diffuse part is sampled the same way as with the diffuse BSDF (cosine weighted hemisphere sampling). The specular (or rather "glossy") part has the following steps:

- Implement a sample warper for the phong specular lobe.
- Rotate that lobe, so that $z+$ points into the direction of the perfect reflection vector.
- Reject all samples that would go below the surface, into the object (careful, see implementation details below).

Specular Warping and Pdf We can generate samples with the Pdf

$$\text{pdf}(\omega) = \frac{n+1}{2\pi}\cos^n(\alpha) \tag{3}$$

by using the following warping

$$(x,y,z) = \left(\sqrt{1-\xi_1^{\frac{2}{n+1}}}\cos(2\pi\xi_2), \sqrt{1-\xi_1^{\frac{2}{n+1}}}\sin(2\pi\xi_2), \sqrt{1-x^2-y^2}\right) \tag{4}$$

You should be able to type that directly into the newly added functions in `warp.cpp`. Note that this PDF (and the corresponding samples) do not exactly match the specular part of the Phong BSDF. However, sampling it exactly is difficult (perhaps even impossible), so we use a good-enough approximation that mimics the overall function shape.

Rotation We need to rotate our samples to r_v now, but first we need the direction of perfect reflection r_v. You can copy the steps to compute it from `mirror.cpp`. Now, you could construct a rotation matrix based on the angles of the reflection direction. But that would require expensive calls to arccos etc. It is much easier and faster to create an orthonormal basis from the reflection vector. We even have that already in Nori in the `Frame` class, usually used to map between world and local shading frame.

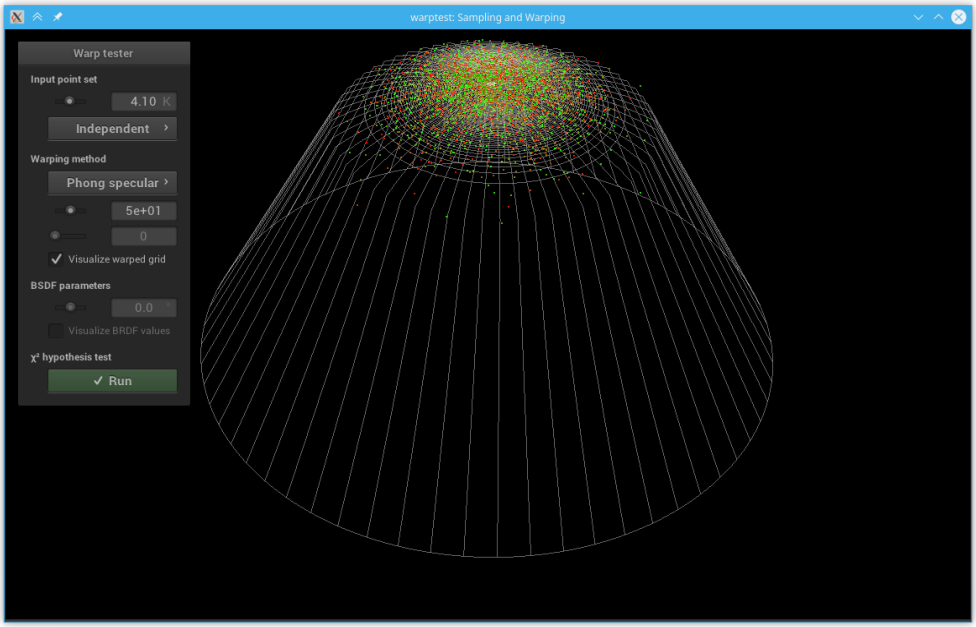


Figure 3: Phong specular sampling

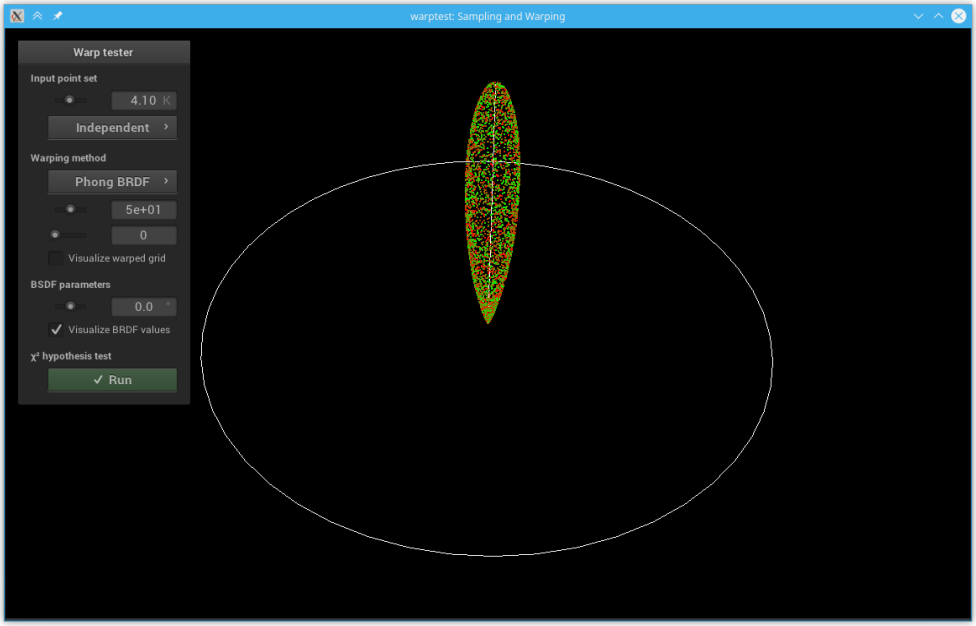


Figure 4: Phong BSDF without rotation

Let's say, the reflection direction is the normal of a reflection frame. Then our warped sample is also in the reflection frame, and we call *toWorld()* to rotate it into our shading

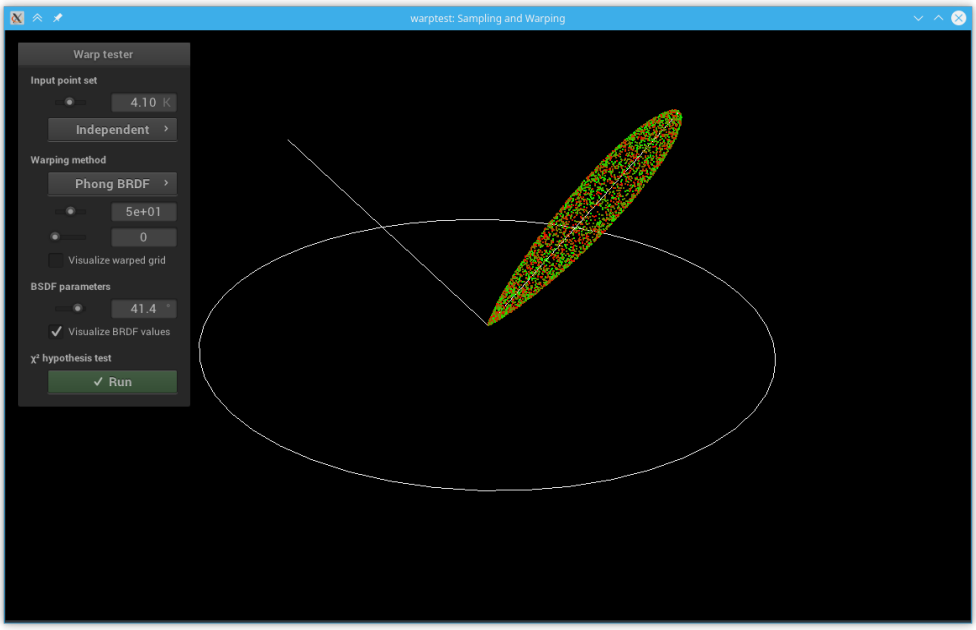


Figure 5: Phong Specular BSDF with rotation

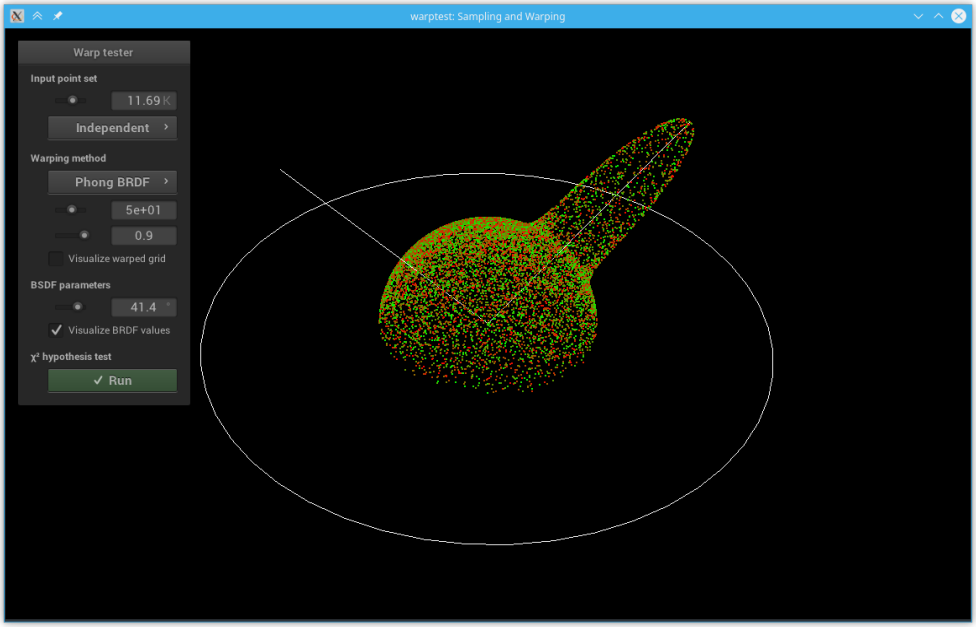


Figure 6: Phong BSDF with rotation and diffuse part

frame.

Constructing that frame requires the reflection vector (goes in through the 3rd parameter) and 2 vectors orthogonal to it. The cross product between the reflection vector and a non-parallel vector gives one orthogonal vector, and another cross product the other. The reflection vector will be one axis of the frame. Find a stable way to construct the remaining two axes for an orthonormal basis around the reflection vector.

Implementation Details

- You need to implement things in `warp.cpp` and `phong.cpp`. Use warptest for testing, there is not only a test for the warp, but for the whole BRDF as well.

- $\cos^n \alpha$ becomes unstable for large exponents, but using our importance sampling method, it appears in the pdf as well, so it cancels out. Use that in your *sample* function. Do not try to do MIS between diffuse/specular part (I tried, it doesn't work).

- Rejection sampling: Do not create a new sample if the one you got is below the surface after rotation. Instead, clamp the contribution to zero, you can easily do that via the $\cos\theta$ term, which belongs to the BSDF now. The reason is, that it wouldn't be possible to compute a correct pdf value if you did "`do x = sample(); while(is_bad(x))`".