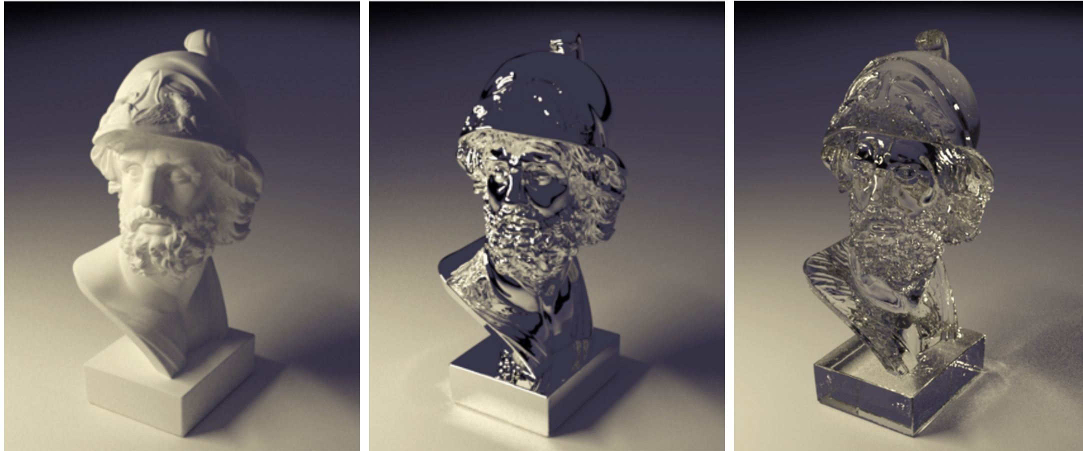


Assignment 4: Materials and Appearance

Deadline: 2020-07-02 23:59

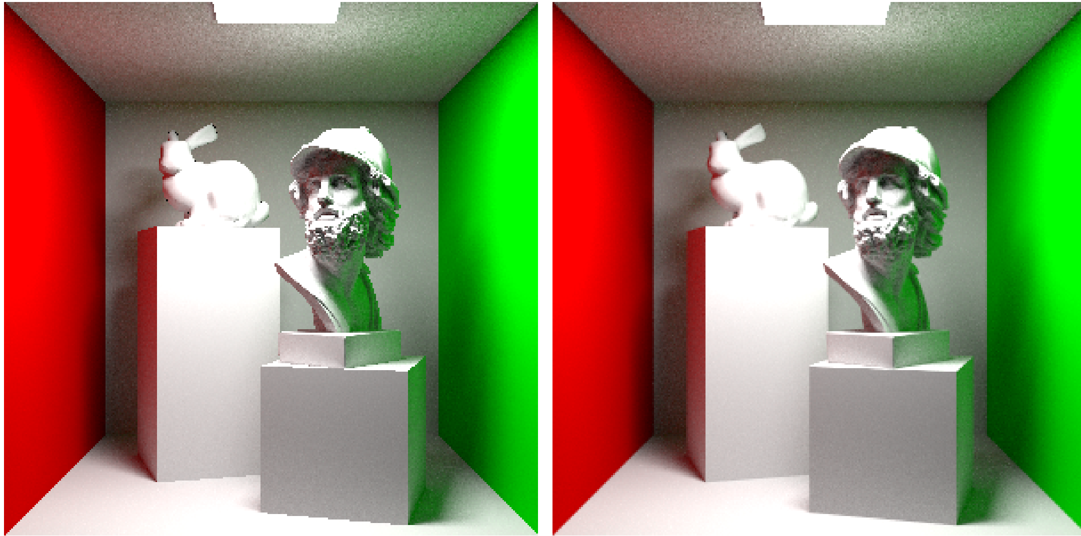
In this assignment, you will introduce more complex materials than the completely diffuse ones that we handled up until now. You should extend your path tracing solution to support multiple BSDFs, such as mirrors and dielectrics. Finally, you should try your hand at low-discrepancy sampling and applying different filters to benefit from powerful image reconstruction filters.

In this final assignment, you are also encouraged to participate in the last competition for the best scenes: show us what great test scenes you can come up with, given the tools provided in the course of the assignments (indirect illumination, mirrors, see-through materials) or any of the features you implemented in addition!

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

```
git checkout master
git pull
git merge submission3       # 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
```

Antialiasing (2 Points)

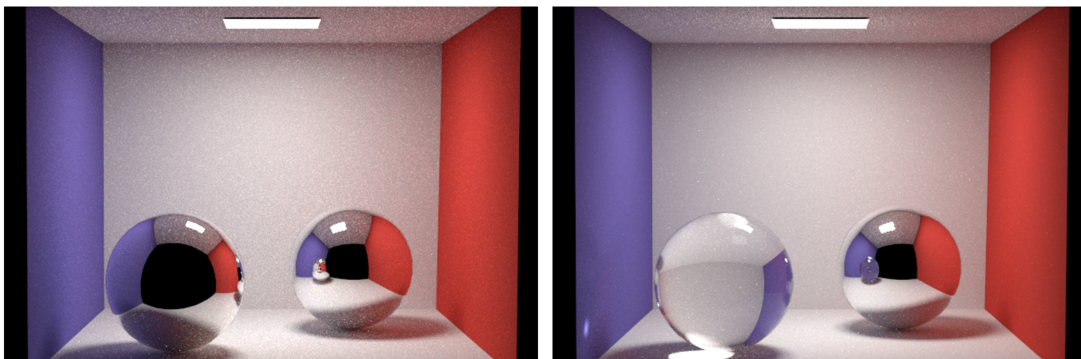


(a) Sampling pixel center only

(b) Sampling over entire pixel

Before we get down to business, let's first get rid of aliasing in our renderings. Until now, we have only ever shot our rays through the center of the pixels. If you have a lower-resolution display or zoomed in on your renderings, you probably saw that they are somewhat jaggy because of this (look at sharp edges, like the bottom of the front box)! We can quickly fix that by running minimalistic antialiasing for the whole pixel: in `main.cpp`, instead of shooting rays always through the pixel center, make it so that the rays can sample the full pixel width and height! Also make sure that your changes are stored in `pixelSample` and then passed to `block.put`. This will be important later.

1 Materials (15 Points + 15 Bonus Points)



(a) Rendering with mirror materials

(b) Rendering with mirrors and dielectrics

1.1 The Dielectric BSDF (10 Points)

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 source 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 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 before continuing with the next bounce in the direction of the surface normal. But, if you actually **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.

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 (see slides for details). 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.

1.2 Next Event Estimation and the Mirror BSDF (5 Points)

In Nori, you will find a perfect mirror material, which needs special care: When you are on a surface and compute an outgoing sample, the direction is fixed by the reflection law. As mentioned during the lecture, this is because the mirror direction distribution is a Dirac delta 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 mirror materials. Only the `sample` function can be guaranteed to pick the unique and correct reflection direction

for the next bounce. The `Mirror` class accounts for that by always returning 0 in the `pdf` function and 0 as the colour when calling `eval`, and 1 in the `sample` function if the surface is reflects the view ray.

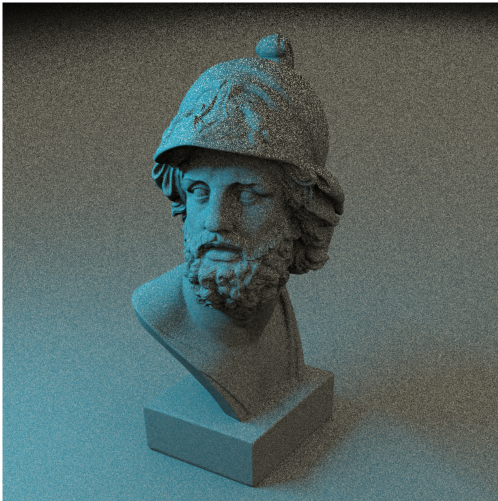
If you want mirror materials to play nice with NEE, you need to take special care: for any direction that is not explicitly the reflection vector, the sampling probability is 0, so you simply can't do light source sampling on mirrors. But if you just ignore direct light on mirror materials, the light sources will be missing in mirror reflections! You can achieve 5 points if you make mirror materials work with NEE.

1.3 Adding a Microfact BSDF (10 Bonus Points)

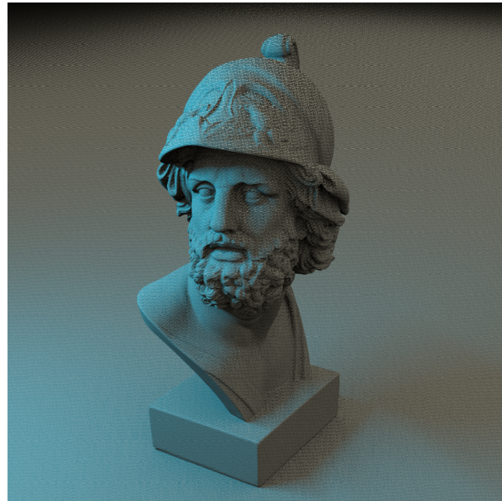
For some bonus points, you can implement a more complex Microfacet material model, according to the steps outlined in Assignment 5, Part 1 found on the Nori webpage. This BSDF should give you a linear blend between a diffuse and a Torrance-Sparrow-based specular model. Note that some of the notes on the webpage do not apply: first, there is no default fresnel implementation in our framework; adding it is part of the assignment for implementing dielectrics. Second, the microfacet BRDF and its distribution will not be tested automatically on the server.

2 Sampling and Appearance (10 Points + 15 Bonus Points)

2.1 Low-Discrepancy Sampling (9 Points + 10 Bonus Points)



(a) Direct lighting with independent sampler



(b) Direct lighting with Halton sampler

Add an additional Halton-based sample generator named `halton` to your solution. The sampler should be able to produce 2D and 1D sequences, based on Halton low-discrepancy sequences that use the radical inverse. For 2D samples, use a combined base-11,13 Halton sequence and for 1D, use a base-7 Halton sequence. To minimize repeating patterns, you should initialize your Halton sampler states (use three separate state variables) to random values (`rand()`). Try your implementation on the light surface sampling scene. Usually, such simple samplers should only be used for individual effects (e.g., picking subpixel coordinates for rays), not the full rendering procedure, but direct lighting is simple enough, so it actually works out ok. If you want to break it, you can try it on full path tracing scenes or change 2D sampling to use base-2,3 instead and see what happens!

For 10 bonus points, implement a sophisticated Halton-based sampling strategy that actually can replace the independent sampler completely! Hints and suggestions for making it work are described in the course book. Make sure that your renderings with Halton converge to the same result as with the independent sampler!

2.2 Support for Filtering (1 Point)

When you fixed aliasing and computed output colors by integrating values over the whole pixel, you basically used a pixel-sized box filter. This is easy to implement, but really not a good choice for filtering: the box filter is sometimes jokingly referred to as the worst filter available. To get support for a few different filters, you need to implement the corresponding suppport in Nori. Once done, you should experiment with different filters and sample counts, to see what a difference they can make.

Apart from the theory behind it, which is not too complex, the **implementation** for supporting separable filters in a tiled renderer is not trivial (it's not that hard either), so we provide the missing code here:

```
void ImageBlock::put(const Point2f &_pos, const Color3f &value) {
if (!value.isValid()) {
/* If this happens, go fix your code instead of removing this warning ;) */
cerr << "Integrator: computed an invalid radiance value: "
<< value.toString() << endl;
return;
}

/* Convert to pixel coordinates within the image block */
Point2f pos(
_pos.x() - 0.5f - (m_offset.x() - m_borderSize),
_pos.y() - 0.5f - (m_offset.y() - m_borderSize));
```

```
/* Compute the rectangle of pixels that will need to be updated */
BoundingBox2i bbox(
Point2i((int)  std::ceil(pos.x() - m_filterRadius),
(int)  std::ceil(pos.y() - m_filterRadius)),
Point2i((int) std::floor(pos.x() + m_filterRadius),
(int) std::floor(pos.y() + m_filterRadius)));
bbox.clip(BoundingBox2i(Point2i(0, 0),
Point2i((int) cols() - 1,
(int) rows() - 1)));

/* Lookup values from the pre-rasterized filter */
for (int x=bbox.min.x(), idx = 0; x<=bbox.max.x(); ++x)
m_weightsX[idx++] = m_filter[(int) (std::abs(x-pos.x()) * m_lookupFactor)];
for (int y=bbox.min.y(), idx = 0; y<=bbox.max.y(); ++y)
m_weightsY[idx++] = m_filter[(int) (std::abs(y-pos.y()) * m_lookupFactor)];

/* Add the colour valuel after filtering to the current estimate.
* Color4f extends the Color3f value by appending a 1. Therefore,
* in the 4th component we are automatically accumulating the filter
* weight. */
for (int y=bbox.min.y(), yr=0; y<=bbox.max.y(); ++y, ++yr)
for (int x=bbox.min.x(), xr=0; x<=bbox.max.x(); ++x, ++xr)
coeffRef(y, x) += Color4f(value) * m_weightsX[xr] * m_weightsY[yr];
}
```

2.3 Tone Mapping (5 Bonus Points)

There is already a basic type of tone mapping in Nori (computations are in float, output is in 8bit integer). Identify that code and extend it with the Reinhard operator for tone mapping, or use something similarly effective.

3 Create your own Scene (5 Easy Points + 30 Bonus Points)

We would ask you to support us and future participants by preparing comprehensive test scenes that we can use in the next year, for features that you particularly liked. Perhaps you just want to get a little variation in (some of you may not want to see any more bunnies...). Preparing a useful scene will earn you 5 points (awarded only once per person). Scenes should only require a reasonable amount of processing power to be useful for students during development.

If, however, you also want to go beyond, to the realm where samples don't matter, feel free to get artistic! We will be holding a competition for who can come up with the most impressive scene: you can prepare scenes by combining individual models (please make sure they are not heavily copyrighted) and features that you implemented (mandatory or bonus) in custom Nori test scenes. Aim to get the most impressive renderings that you can manage! We will pick a winner, whose work will earn her/him 30 bonus points, as well as the honor of being exhibited on the course homepage. If you want to participate, this should be an extra scene in addition to the "useful" scene for future students.

Unless you want to keep them a secret, we encourage you to show off your scenes in TUWEL while you design and refine them to get some feedback!

4 I'm all done, now what?

There is a copious amount of additional bonus points up for grabs this time, check the following sections for details. In addition, you can also find papers or effects on your own and implement them. We will be generous with bonus points (although you won't really need them anymore if you do that). If you would like to go for something really ambitious but need an incentive, talk to us, we can give you more time and another 3 ECTS.

Another thing to keep in mind: if you stick out of the crowd, it is likely we would recommend you for a PhD position either here or at one of the more specialised labs.

5 Bonus Tasks (Loads of Points)

5.1 More Materials

The topic of materials does not stop at the microfacet model. There is a wide range of more complex aspects of objects' physical appearance, and the resulting rendering solutions can become very sophisticated. Background: Physics and Math of Shading by Naty Hoffman is a nice didactic introduction and contains a lot of in-depth information. It is a good read even if you don't want to implement anything! Naty also has course presentations hosted on Youtube for you to access. Our course book also has a lot of information on materials and even some code (but be aware that the notation and conventions might be different to what Nori uses).

Ideas for bonus tasks with materials:

- Implement the (not physically-based) Phong BSDF for glossy materials (see Appendix, 10 Points)
- Support NEE + MIS with dielectric materials by implementing manifold next event estimation (50 Points)
- A physically correct gold BSDF (all effects described by Naty Hoffman, 20 Points)
- Fourier Basis BSDFs (from the book, 10 Points)
- Subsurface scattering (SSS, your own research, 50 Points)

5.2 More Sampling

- Support MIS with mirror materials (10 Points)
- Adaptive sampling and reconstruction to get more out of the samples you take (e.g. MDAS, AWR. 250 Points)
- Path guiding (2 pass unbiased algorithm, first cast photons into the scene and record important 'incoming directions'. Then, use them for importance sampling. 300 Points)
- A basic path tracer on the GPU (CUDA, GLSL, HLSL... should be unbiased and support diffuse and specular BRDFs at least. 300 Points)

5.3 Other Ideas

- Textures (UVs are already loaded, you'd need to extend the xml, 40 Points)
- Light tracing (start the path from the light and connect it to the camera via NEE. You'd have to change stuff in the camera, scene, and main loop. The most important thing would be to hold an additional image block for the whole image plane, because the the ray can hit any pixel. We could give some more hints in TUWEL, just ask! 80 Points)
- Bidirectional Path Tracing (Light tracing is a good start, we can help in TUWEL. 160 Points)
- Participating Media (You would have to do your own research. 320 Points)

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 fourth assignment, push to submission4.

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.
- 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. The original Phong was not even energy conserving, therefore we will implement 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 distill everything important into a summary.

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

$$\begin{aligned} \mathrm{f_r}(x,v,\omega) &= \mathrm{f_{r,d}}(x,v,\omega)+\mathrm{f_{r,s}}(x,v,\omega) && (1) \\ &= k_d p_d \frac{1}{\pi} + k_s(1-p_d)\frac{n+2}{2\pi}\max(0,\cos^n \alpha), && (2) \end{aligned}$$

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

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

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.

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 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.

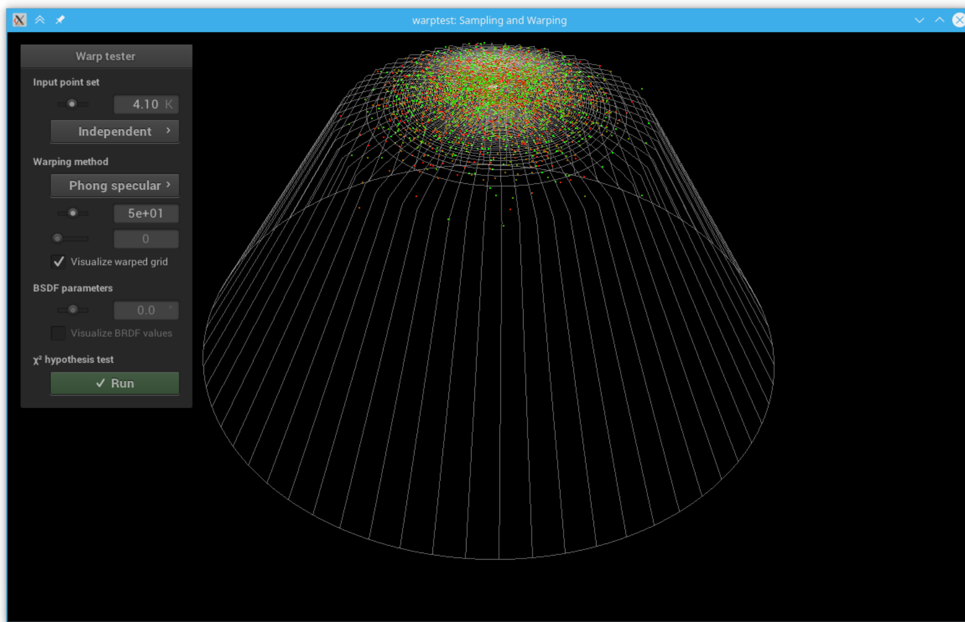


Figure 4: Phong specular sampling

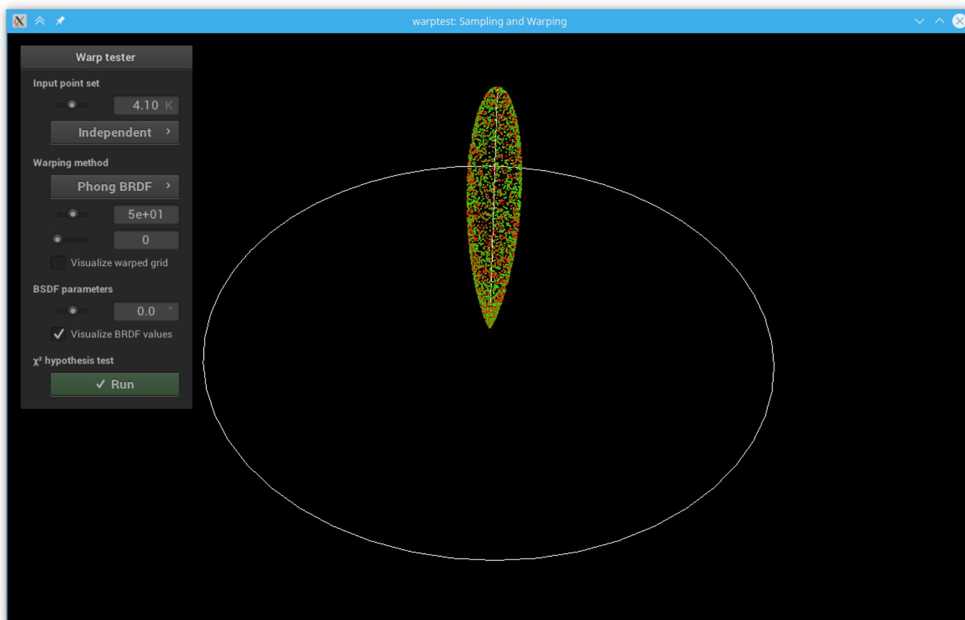


Figure 5: Phong BSDF without rotation

Implementation Details

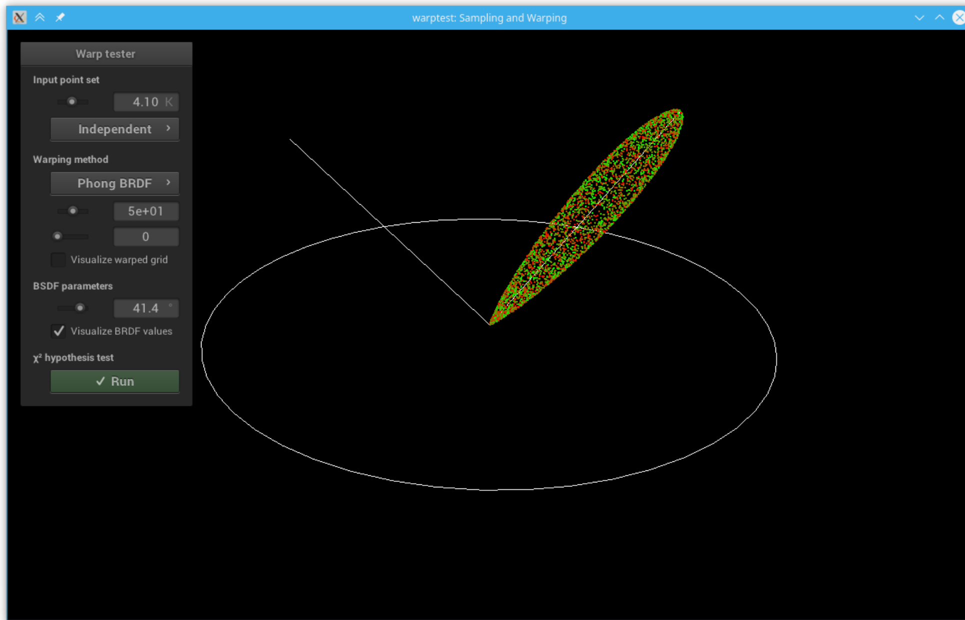


Figure 6: Phong Specular BSDF with rotation

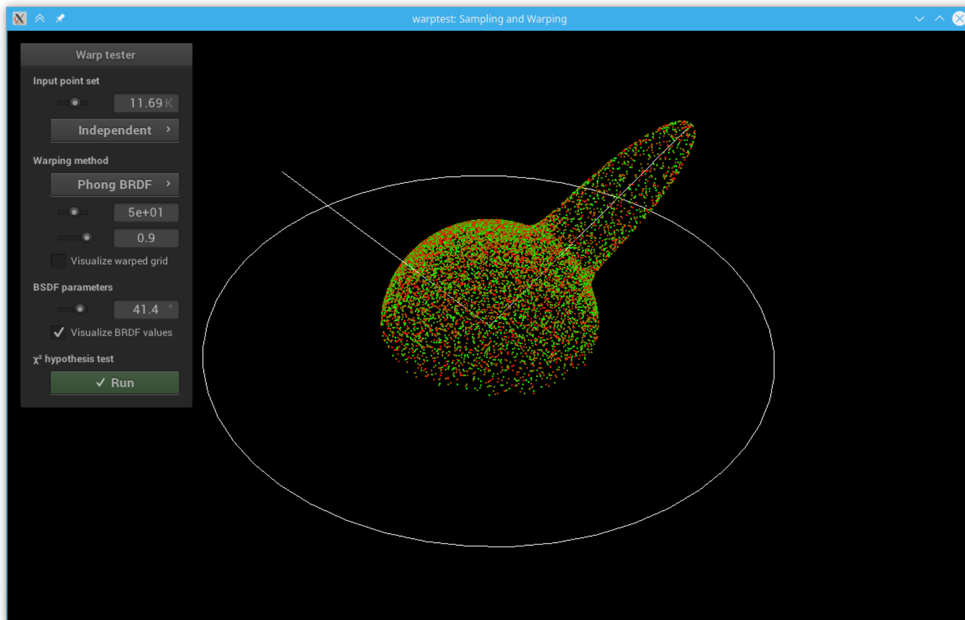


Figure 7: Phong BSDF with rotation and diffuse part

- 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))`".