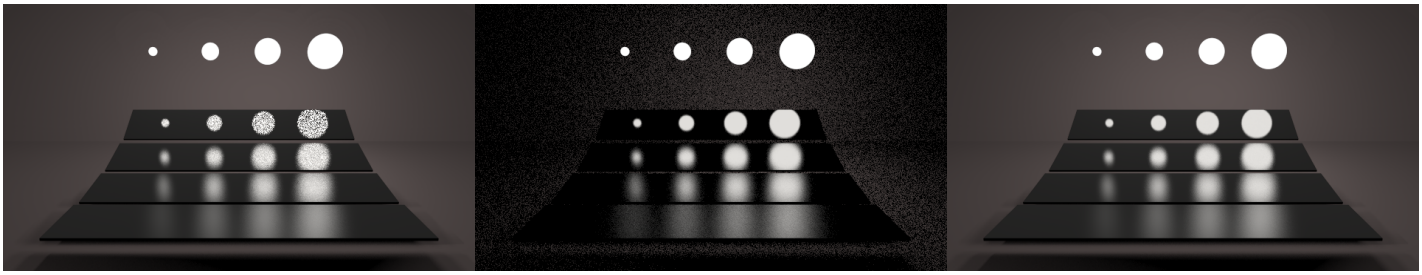
ECSE 446/546: Realistic/Advanced Image SynthesisAssignment 4: Advanced Monte Carlo

Due Monday, November 5th, 2018 at 11:59pm EST on myCourses
18% (ECSE 446 / COMP 598) *or* **13%** (ECSE 546)



In this assignment, you'll implement a direct illumination integrator using importance sampling schemes for BRDFs and for spherical lights. You will also combine both BRDF and emitter importance sampling using multiple importance sampling (MIS). This assignment contains **offline rendering tasks only.**

Contents

Before you begin, we recommend that you review the course material, especially slides on surface area and subtended solid angle sampling, Monte Carlo estimators and MIS.

Afterwards, start by copying your rendering loops as well as any code needed from previous assignments and **ensure that you initialize your random sampler** object with your student ID, *e.g.,* `Sampler sampler(123456789)`.

Area Lights in TinyRender

In Assignment 2 you implemented a point light source. While this emission profile is convenient in certain contexts, it is not physically-realizable. In the real world, emitters have form and geometry (*e.g.,* light bulbs, neon lights, etc.), in these cases, the direct illumination equation does not reduce from an integral to a deterministic evaluation.

Emitter Structure

TinyRender has an `Emitter` structure in `src/core.h` that encapsulates the notion of an area light source. Its definition is as follows:

```
struct Emitter {
   size_t shapeID;
   float area;
   v3f radiance;
   Distribution1D faceAreaDistribution;
   v3f getRadiance() const { return radiance; }
   v3f getPower() const { return area * M_PI * radiance; }
   bool operator==(const Emitter& other) const { return shapeID == other.shapeID; }
};
```

Field	Description
`shapeID`	ID of the shape the emitter is associated with
`area`	Total surface area of the area light
`radiance`	Emitted radiance (uniform distribution)
`faceAreaDistribution`	Discrete probability density function over the mesh triangles

When the scene description file is parsed, a vector of emitters is automatically created in `Scene`. This means that emitters are also indexed: you can have a mesh with shape ID 3 but emitter ID 0 (*e.g.,* if there is only one light source in your scene). To determine if an intersection `i` lies on a surface emitter, you can retrieve its emission with `getEmission(i)` and check if it is nonzero. Other functions you might want to use are described below; all are available from the scene.

Method	Description
`getEmitterIDByShapeID()`	Maps the shape ID to the corresponding emitter ID, if it exists
`getEmitterByID()`	Returns a reference to an `Emitter` struct with this ID
`selectEmitter()`	Uniformly chooses one emitter in the list of emitters (independent and identically distributed) and returns its emitter ID

For instance, to select an emitter to sample, you can do:

```
float emPdf;
size_t id = selectEmitter(sampler.next(), emPdf);
const Emitter& em = getEmitterByID(id);
```

where `emPdf` is the PDF over all emitters in the scene. In our case, this is just $1/$ for light sources. Later, you will need to multiply the light source PDF (*e.g.*, 1/Area) by the probability of *choosing* that emitter (`emPdf`) before evaluating the emitted radiance. This is because *selecting* a light to sample is also part of the stochastic process.

You will be using the `Emitter` interface throughout this assignment. You are encouraged to look at the function signatures related to emitters to understand how they fit into the rendering framework.

Handling Spherical Lights

In TinyRender, there are no analytic sphere shapes, only triangle meshes. In this assignment, however, **we will assume that all lights are spherical,** so how does TinyRender handle this? To avoid creating additional structures for analytic shapes, it is possible to *convert* tesselated meshes to analytic spheres. If `em` is known to be an emissive surface, you can retrieve its center and radius as follows:

```
v3f emCenter = scene.getShapeCenter(em.shapeID);
float emRadius = scene.getShapeRadius(em.shapeID);
```

Once you have these quantities, you can essentially treat the mesh as an analytic sphere to specialize your sampling strategies (*e.g.,* solid angle or surface area). This will come in handy in Part 2.2 of the assignment. Note that your intersection test will still rely on the tesselated mesh. Before we move to these sampling schemes, let's have a look at a much simpler MC estimator for direct lighting.

1 A Simple Direct Illumination MC Estimator (*10 pts*)

In this first task, you will implement a direct illumination integrator that uses area lights: as discussed in class, we may draw samples according to directions or points. Recall the reflection equation, which expresses the reflected radiance distribution as an integral over, i.e., the unit hemisphere centered at $\mathbf{x}$ of the product of these terms: the BRDF, the cosine foreshortening term and the incident radiance:

$$L_r(\mathbf{x}, \omega_r) = \int_{\Omega^2} L_i(\mathbf{x}, \omega_i)\, f_r(\mathbf{x}, \omega_i, \omega_r)\, \cos\theta_i \, \mathrm{d}\omega_i. \tag{1}$$

You will approximate this integral using Monte Carlo estimation, considering only *direct* illumination; this means that $L_i(\mathbf{x}, \omega_i)$ is only non-zero for rays $\mathbf{r}(t) = \mathbf{x} + t\omega_i$ that happen to hit a light source.

A correct but naive way of evaluating the reflection equation is to sample directions uniformly over the hemisphere. Your first estimator will instead implement **cosine-weighted hemispherical importance sampling**. The function `DirectIntegrator::renderCosineHemisphere` (`src/integrators/direct.h`) will draw samples according to the appropriate strategy, and return an MC estimate which will once again rely on: sampling directions, tracing shadow rays from your shading point in order to evaluate the visibility in these directions, evaluating the remaining terms of the integrand at these directions, and dividing by the appropriate sampling PDF.

Implement this Monte Carlo estimator and render the test scenes `sphere_cosine_X.toml` for $\in \{4, 64\}$ in `data/a4/sphere/tinyrender`. Make sure to compare with the references below before moving onto the next part.

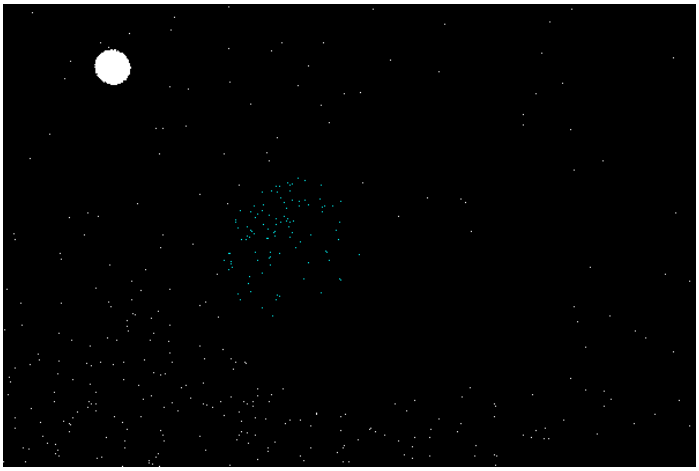


Figure 1: *Cosine-hemispherical sampling at 1 spp with 4 emitter samples*

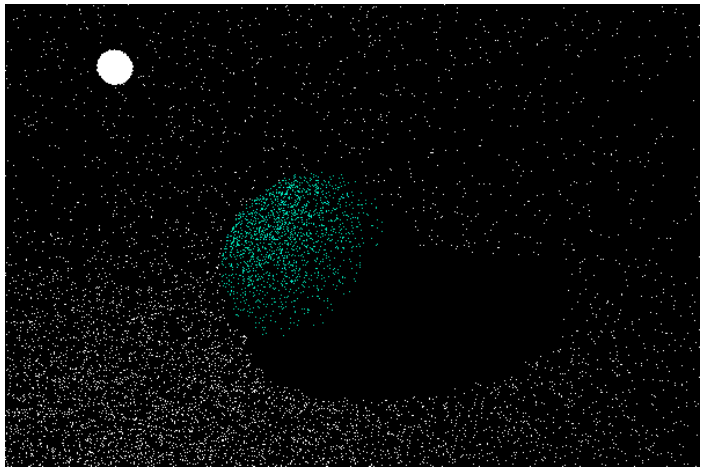


Figure 2: *Cosine-hemispherical sampling at 1 spp with 64 emitter samples*

Note that this hemispherical estimator can be extremely inefficient: light sources may only subtend a small solid angle, and so many hemispherical samples can be wasted, causing the algorithm to produce noisy images at low (and even moderate) sample counts. We will soon see how to mitigate this problem with better sampling routines.

2 Direct Illumination with Importance Sampling (*50 pts*)

2.1 Sampling BRDFs (*20 pts*)

Next, you will implement methods to directly sample the two material models we have seen so far: diffuse and Phong BRDFs. In A3, you implemented functions to sample different distributions and evaluate their associated probability density functions. In this assignment, you will write methods to sample a direction in a BRDF lobe and evaluate the corresponding PDF in that direction. These two functions are part of the `BSDF` structure, which both `DiffuseBSDF` and `PhongBSDF` inherit from.

The function `BSDF::sample()` takes as input an intersection point (in local coordinates), a 2D uniform canonical random variables, and a pointer to PDF value it will populate as output. Your goal is to sample an appropriate direction $\omega_i \sim f_s(\mathbf{x}, \omega_i, \omega_o)$ and set the density $p(\omega_i)$, before returning the evaluation of the BRDF. The `BSDF::pdf()` method needs to be implemented too, as you might sometimes need to evaluate the PDF in a direction, separately.

2.1.1 Diffuse Reflectance BRDF (*5 pts*)

Start by implementing `DiffuseBSDF::sample()` and `DiffuseBSDF::pdf()` in `src/bsdfs/diffuse.h`. These functions both return zero for now; you have to sample a direction according to **cosine-weighted hemispherical distribution** and evaluate the BRDF in this direction. Use your previous warping functions from A3 in `src/core/math.h`.

2.1.2 Phong Reflectance BRDF (*5 pts*)

Next implement `PhongBSDF::sample()` and `PhongBSDF::pdf()` in `src/bsdfs/phong.h`. Once again, you will use the functions you implemented in A3 to warp 2D samples to 3D directions. Since you need to return the Phong evaluation divided by the PDF, you can call `Phong::eval()` and `Phong::pdf()` inside `Phong::sample()`. Make sure that this division is well-defined, otherwise simply return zero.

2.1.3 Direct Illumination with BRDF Importance Sampling (*10 pts*)

To test your BRDF sampling routines, implement `DirectIntegrator::renderBSDF()` in `src/integrators/direct.h`. Your implementation should be similar to your MC estimator from Part 1, except directions will now be sampled according to the BRDF lobes. When you're done, render the two test scenes `sphere_bsdf_diffuse.toml` and `sphere_bsdf_phong.toml` in `data/a4/sphere/tinyrender`. Your results for the diffuse BRDF scene should match your renders from Part 1. Your outputs for the Phong BRDF should be similar to the ones below:

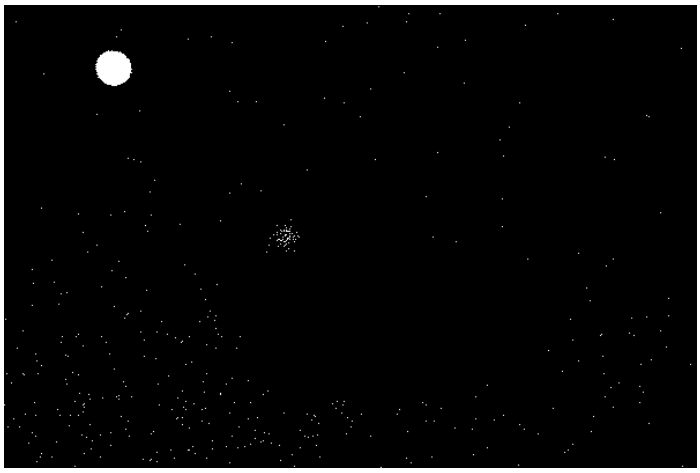


Figure 3: *Phong BSDF sampling at 1 spp with 4 BSDF samples*

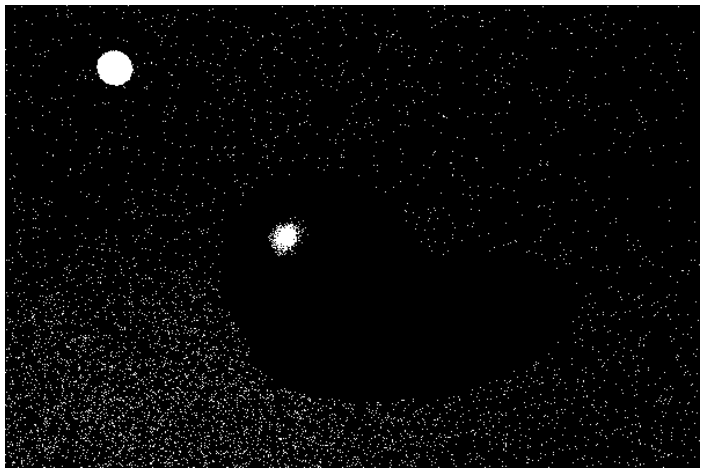


Figure 4: *Phong BSDF sampling at 1 spp with 64 BSDF samples*

2.2 Sampling Spherical Lights (*30 pts*)

We will consider ways of sampling spherical light sources.

2.2.1 Uniform Surface Area Sampling (*15 pts*)

Instead of sampling directions on the hemisphere, we can sample *surface points* directly on the light source. Conceptually, this means that we can express our integral using the surface area form of the reflection equation, integrating over the light source surfaces $_e$ instead of over the hemisphere of incident lighting directions ^{2}:

$$L_r(\mathbf{x}, \omega_r) = \int_e L_e(\mathbf{y}, \mathbf{y} \to \mathbf{x})\, f_r(\mathbf{x}, \mathbf{x} \to \mathbf{y}, \omega_r)\, G(\mathbf{x} \leftrightarrow \mathbf{y})\, \mathrm{d}\mathbf{y}, \tag{2}$$

where G is the geometry term discussed in class. Here $\mathbf{x} \to \mathbf{y}$ refers to the normalized direction from $\mathbf{x}$ to $\mathbf{y}$, and $L_e(\mathbf{y}, \mathbf{y} \to \mathbf{x})$ is the amount of emitted radiance in the direction $\mathbf{y} \to \mathbf{x}$. Implement the area sampling scheme in `sampleSphericalEmitter()` by sampling a position $\mathbf{y}$ on the sphere and evaluating the PDF at that point. You will need to set the direction $\omega_i = \mathbf{x} \to \mathbf{y}$, the normal at $\mathbf{y}$, and the corresponding PDF. Then, implement the MC estimator for this strategy in your `DirectIntegrator`.

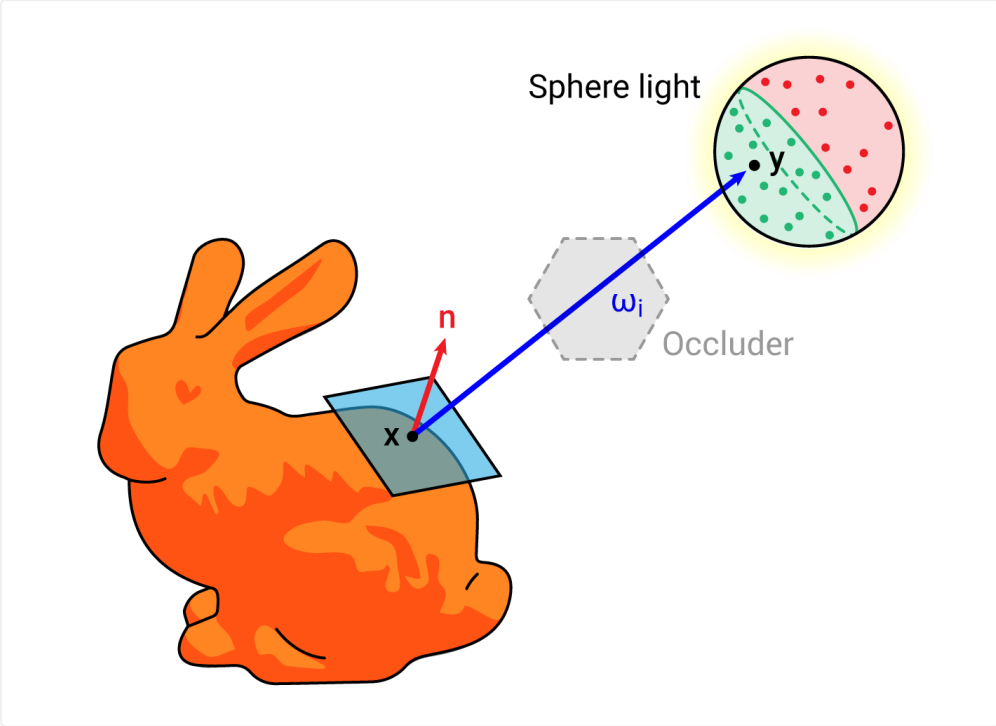


Figure 5: *Surface area sampling. Green points can return nonzero radiance depending on the visibility, whereas red points are not visible from the shading point and thus contribute zero (due to G).*

Implement `DirectIntegrator::renderArea()` in `src/integrators/direct.h` using this sampling scheme. You will need to implement `sampleSphereByArea()` to draw samples uniformly on the sphere light.

This surface area sampling strategy is typically preferable to uniform hemispherical sampling, but roughly half of the surface samples end up being wasted (*i.e.*, falling on the backside of the sphere light, from the point of view of a shading point). Indeed, any sample on the emitter that is not directly visible from the shading point $\mathbf{x}$ do not contribute to the integral (red points in the above figure). There is a third, more efficient option for sphere light sampling.

2.2.2 Subtended Solid Angle Sampling (*15 pts*)

To avoid generating samples that are guaranteed to not contribute to our estimator, you will implement subtended solid angle sampling in `DirectIntegrator::renderSolidAngle()`. Sample direction towards the subtended spherical cap using `DirectIntegrator::sampleSphereBySolidAngle()` and set the appropriate fields. As before, PBRTe3 Chapter 14.2 will be useful here.

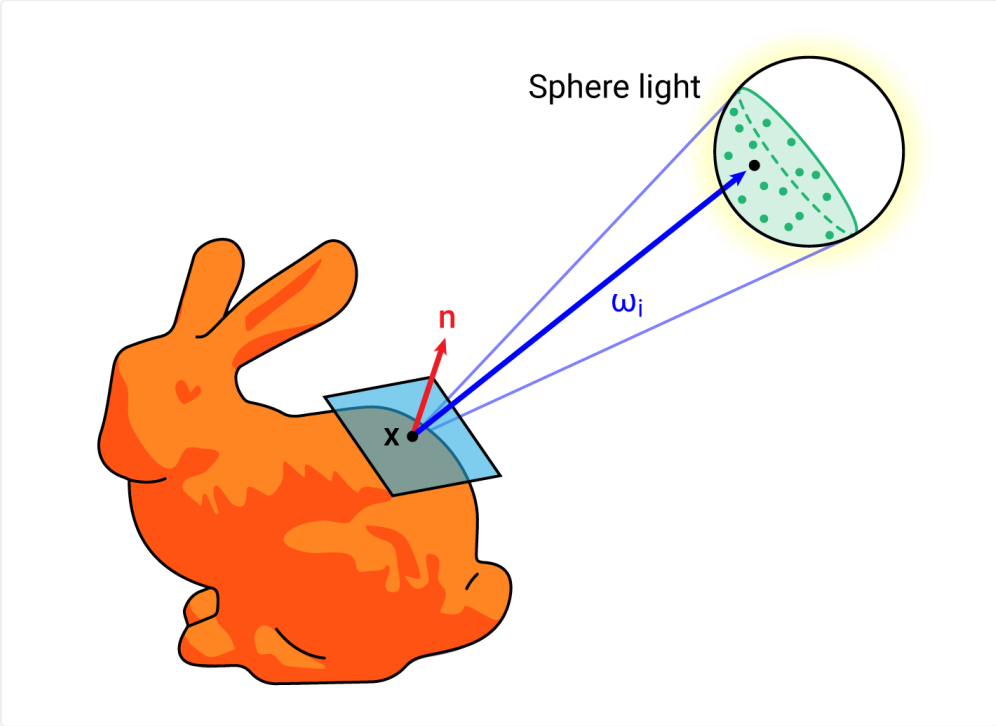


Figure 6: *Subtended solid angle sampling. Every direction sampled will result in a point on the emitter that is on the correct side.*

Comparing Sampling Strategies

Make sure to compare your results with the reference images below. What's important to notice is that, in the context of spherical lights, surface area sampling will always result it more variance (noise) than the subtended solid angle sampling for the same amount of samples.

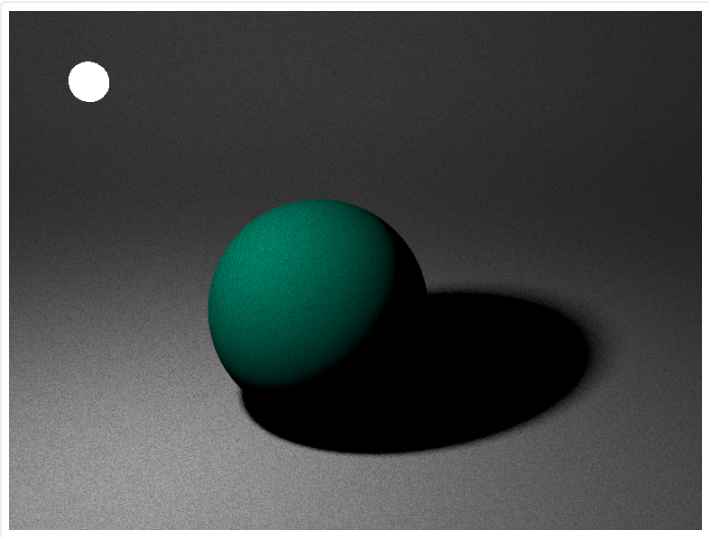


Figure 7: *Surface area sampling at 64 spp with 1 emitter sample*

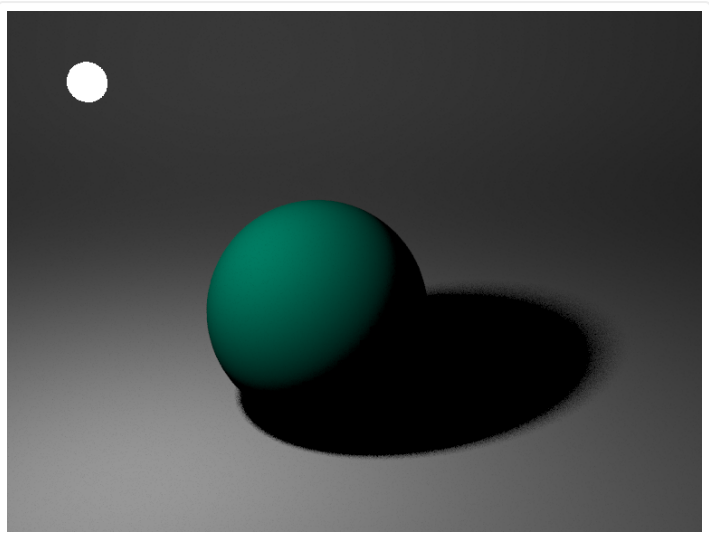


Figure 8: *Subtended solid angle sampling at 64 spp with 1 emitter sample*

3 Direct Illumination with MIS (*40 pts*)

Multiple Importance Sampling

In the final part of this assignment, you will implement multiple importance sampling (MIS) as introduced by Veach & Guibas (1995). Specifically, you will implement an integrator that combines BRDF and emitter importance sampling to render an image.

Recall that if two sampling distributions p_f and p_g can be used in an estimator for the integral $\int h(x)\,\mathrm{d}x$, a single Monte Carlo estimator that uses both distributions simultaneously is given by the MIS formulation as

$$h(x)\,\mathrm{d}x \approx \frac{1}{n_f}\sum_{i=1}^{n_f}\frac{h(\quad_i)w(\quad_i)}{p_f(\quad_i)} + \frac{1}{n_g}\sum_{j=1}^{n_g}\frac{h(\quad_j)w(\quad_j)}{p_g(\quad_j)}, \tag{3}$$

where n_f is the number of samples drawn proportional to p_f, n_g the number of samples drawn proportional to p_g, and w_f & w_g are special normalization weighting functions chosen so as to not bias the value of the estimator.

Many such weighting functions exist, and a provably good choice is based on a *balance heuristic*, given by the following equation, where $s \in \langle f, g\rangle = I$:

$$w_s(x) = \frac{n_s\, p_s(x)}{\sum_{i\in I} n_i\, p_i(x)}, \tag{4}$$

Your task is to implement MIS with this heuristic. The balance heuristic is already implemented as part of the `DirectIntegrator` structure. Implementation-wise, your `DirectIntegrator::renderMIS` function should have two Monte Carlo loops: one over `m_emitterSamples` and one over `m_bsdfSamples`. The former should be reminiscent of the first part of this assignment, except you will need to adapt it to take the weighting function into account. Use the function `DirectIntegrator::balanceHeuristic()` for this purpose. Use subtended solid angle emitter sampling for this part. The BRDF sampling part isn't much different: once you find an intersection `i`, you'll want to call `BSDF::sample()` instead of `BSDF::eval()` to sample a direction according to the shading point's BRDF.

Rendering the Veach Scene

Implement the MIS integrator in `DirectIntegrator::renderMIS()` and test your code by rendering the test scenes in `data/a4/veach` containing the popularized scenes from Eric Veach's thesis. Below are the reference images that were rendered at 64 spp. Note that a total of 5 lights are present in the scene: there is one ambient light that is not visible in the images.

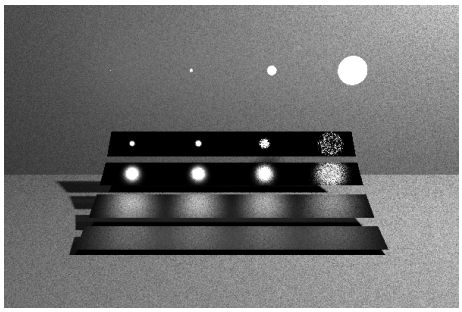


Figure 9: *Emitter sampling with 1 emitter sample*

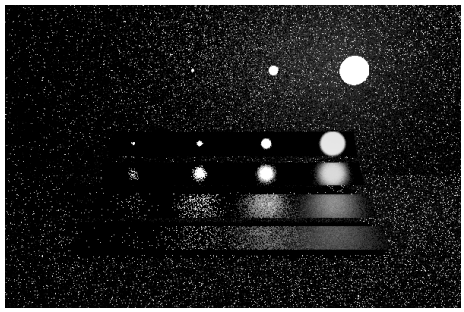


Figure 10: *BSDF sampling with 1 BSDF sample*

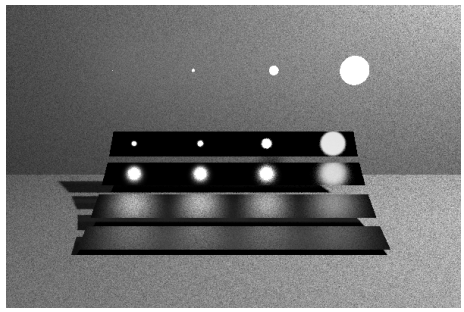


Figure 11: *MIS with 1/1 emitter/BSDF sample*

The final render for this part of the assignment is the same scene, except the lights have different colors.

Testing Your Implementation

Implementing MIS can be tricky and you might find many subtle bugs in your previous implementations that didn't surface until now. Here are a few tips to assist you in debugging your code:

- Test with diffuse-only materials to begin with. Only test MIS with Phong once you're convinced your integrator works properly with diffuse surfaces.
- Test with *emitter samples only* by setting the number of BSDF samples to zero, and make sure the output image is the same as Part 2.1.
- Then test with *BSDF samples only* by setting the number of emitter samples to zero in the TOML file. The output image should obviously have more noise, but should eventually converge with enough samples.
- Finally, *combine the two* and compare the rendered image to the provided test scene.
- When in doubt, recall that a naive uniform sampling MC estimator will (eventually) generate the correct answer: this is one way to isolate bugs in your specialized PDF sampling and evaluation code, since uniform sampling does not rely on much more than evaluating the integrand terms (as opposed to drawing samples according to them).

What to submit

Render all the scenes in `data/a4/teapot` and `data/a4/veach_final`. Also edit your `config.json` to include your credentials. You should only be modifying the **lines 2–4** of your configuration file.

Submit this file with all your code in a `.zip` or `.tar` archive file. Include your raw `.exr` files in separated folders (see structure below).

```
a4_first_last.zip
   config.json
   src
   offline/
      teapot_cosine.exr
      teapot_bsdf.exr
      teapot_area.exr
      teapot_solid_angle.exr
      veach_emitter.exr
      veach_bsdf.exr
      veach_mis.exr
```

Make sure your code compiles and runs before submitting! **You will obtain a score of zero if your submission does not follow this exact structure.** You can use the `tree` command to verify it.

formatted by Markdeep 1.04