15-468 Final Project Report

Jeff Tan (jefftan)

05/12/2022

1 Summary

For my final project, I implemented the following features in DIRT:

- **Bidirectional path tracing** (Advanced, 10pts): BDPT is a generalization of the standard path tracing algorithm that constructs paths that start from the camera on one end and the light source on the other end, connecting them in the middle with a visibility ray. It better handles difficult lighting conditions and reduces the variance of the image. Unlike photon mapping, BDPT is unbiased and does not blur the scene illumination.
- **GGX Microfacet BSDF model** (Simple, 2pts): Accurately modeling transmission through a rough refractive boundary, for example glass and unpolished metals, is difficult in computer graphics due to the need for physically based and verified models. The GGX microfacet BSDF is empirically better able to model transmission through a rough refractive boundary, providing a closer match for certain surfaces compared to the standard Beckmann distribution function which we implemented in Assignment 2.
- **Parallelism on CPUs**: DIRT is single-threaded, which significantly slows down renderings when rendering large images, interacting with complex scene geometry, or when using a large number of image samples. With the wide availability of multicore CPU processors and the abundant parallelism available during rendering as each pixel is sampled independently, implementing CPU-based parallelism can bring significant speedups without much effort.

In each section below, I provide a theoretical summary of one of these features, describe the technical details of my implementation in DIRT, and show some feature-specific results and renderings to showcase the feature's impact.

2 Parallelism on CPUs

2.1 Feature Description

In this section, we aim to parallelize DIRT on a multicore CPU processor to bring significant speedups to DIRT without much effort. Compared to GPU-accelerated raytracing which requires completely rewriting the code into GPU kernels from the ground up, CPU-based parallelism is comparatively simple to implement.

2.2 Implementation Details

Sequential implementation: The standard implementation of DIRT is single-threaded, and the `Integrator` class renders each scene as follows, where `resolutionX` and `resolutionY` denote the image resolution and `numSamples` denotes the number of samples per pixel:

- `for y in range(resolutionY):`
  - `for x in range(resolutionX):`
    - `image(x, y) = Black`
    - `for sample_idx in range(numSamples):`
      - `sample = sampler.generateSample()`
      - `ray = camera.generateRay(sample)`
      - `image(x, y) += Emitted radiance along ray`
    - `image(x, y) /= numSamples`

Here, we loop through each pixel and for each sample, we generate a random sample, generate a random camera ray passing through that pixel, and accumulate the emitted radiance along that ray onto the image

Parallelism strategy: Because the camera and scene geometry are fixed, the outer two for-loops are easy to parallelize as their data can be shared between threads. Only the sampler needs to be duplicated across threads, as it has internal state that would be costly to synchronize between threads. After each thread is done rendering, we also need to aggregate the rendered pixels from each thread to create the output image.

Granularity size: Parallelizing across the image dimensions in theory gives us `resolutionX * resolutionY` units of parallel work, but since we are parallelizing on CPUs, we only have at most 64 or 128 threads available in hardware. Therefore, it is important to choose the granularity size well, as a too-large granularity results in uneven work distribution and idle threads towards the end of execution, while a too-small granularity results in excess overhead needed to create threads, distribute work, and aggregate the results. As a middle ground between these extremes, we divide up the image into 16×16 tiles and treat each tile as an individual unit of work to be distributed across threads.

Parallel implementation: Our parallel implementation of DIRT is summarized as follows:

- `tiles = // Divide image up into 16x16 tiles`
- `mutex = // Create a mutual exclusion lock guarding access to the global image`
- `parallel for tile in tiles:`
  - `// Create and seed a thread-local copy of the sampler`
  - `(x0, x1, y0, y1) = // Compute sample bounds for this tile`
  - `for y in range(y0, y1):`
    - `for x in range(x0, x1):`
      - `tile(x, y) = Black`

    - `for sample_idx in range(numSamples):  tile(x, y) += Sample i`
        - `sample = sampler.generateSample()`
        - `ray = camera.generateRay(sample)`
        - `tile(x, y) += Emitted radiance along ray`
    - `tile(x, y) /= numSamples`
- `// Acquire mutex and merge tile into global image`

Here, we divide up the image into 16x16 tiles and define a mutual exclusion lock to guard access to the global image. In parallel, across the set of image tiles, we generate a unique `Sampler` instance for that tile, compute the sample bounds of the tile, and loop over the pixels in the tile as before, rendering each pixel using independent samples. At the end, we acquire the mutual exclusion lock and accumulate the image tile onto the global image.

Sampler initialization: It is necessary to construct a unique `Sampler` instance per tile because the RNG inside the sampler has internal state which is expensive to synchronize between threads. To ensure that each sampler generates unique random numbers, we seed each tile's sampler with that tile's linear index, computed as `tile.y * nTiles.x + tile.x`. Here, `tile.y` and `tile.x` are the (x, y) coordinates of the tile and `nTiles.x` represents the number of tiles needed to cover the width of the image, which is equal to `ceiling(resolutionX / 16)`.

Parallel-for implementation: In order to implement parallel for, we represent each unit of parallel work using a `ParallelForLoop` class that encapsulates the arguments and function body of a C++ lambda expression. We create a linked list of `ParallelForLoop` objects for each tile in the image, and define a thread pool that continually pulls tiles off the work queue and executes the lambda expression within. This lambda expression is responsible for duplicating the sampler, computing tile bounds, and rendering then merging in a particular image tile.

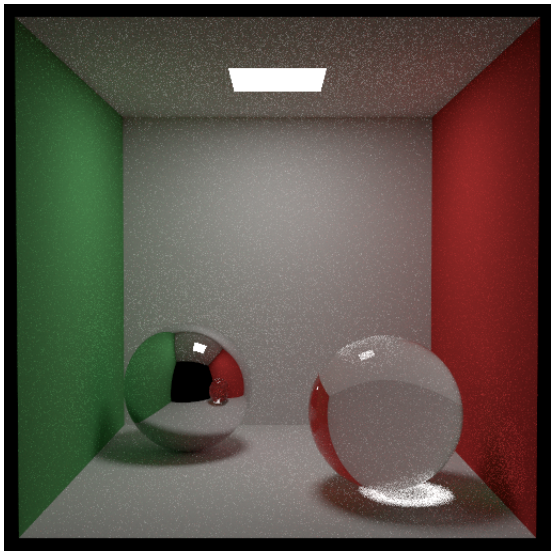
2.3 Results

Our implementation yields significant speedups on a wide range of integrators and samplers. Shown below is a table of runtimes to render various scenes, using an 8-core Intel i7-9700K CPU, 32 spp, path tracing, and independent sampling:

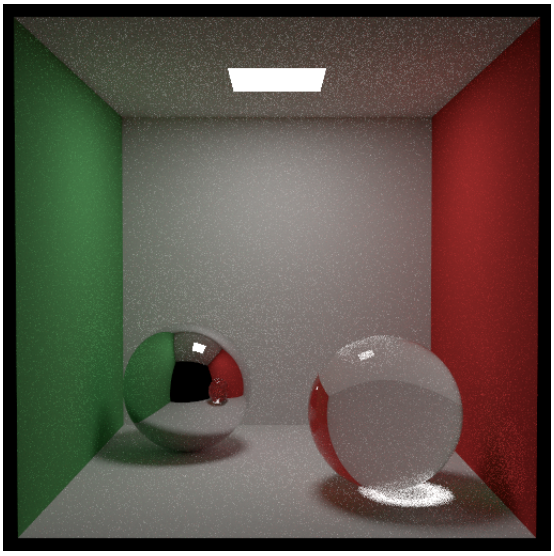
Scene	1 core (s)	2 cores (s)	4 cores (s)	8 cores (s)	Speedup
teapot	80.00	38.50	20.00	10.00	8.00x
cornell-box	21.50	11.00	5.50	3.00	7.16x
veach-mats	13.50	7.00	4.00	2.00	6.75x
earth	3.00	1.75	1.00	0.50	6.00x

Shown below are two examples of cornell-box rendered with and without parallelism, using 128 spp, path tracing with NEE, and independent sampling, to show that the output is the same up to random noise:

With parallelism



Without parallelism



Performance analysis: With more complicated scenes that require a longer runtime, the parallel speedups approaches the theoretical maximum of 8x. The teapot scene includes a detailed triangle mesh that requires large numbers of BVH acceleration structure lookups and triangle-ray intersection tests, which gives each thread more work to do and amortizes the overhead of creating and managing a parallel work queue. On the other hand, scenes with simpler geometries and faster render times such as earth and veach-mats have lower overall speedups.

2.4 Supporting Bidirectional Path Tracing

The approach described above does not fully extend to bidirectional path tracing. In the $t = 1$ case of BDPT, there is the possibility of sampling light paths that connect directly to the camera with a visibility ray. Since these paths have the potential to write to any location on the image, the output is no longer confined to just the image tile that the renderer is currently considering. To deal with this case, we simply acquire the global image lock and accumulate the contribution of $t = 1$ rays directly onto the global image.

Despite the more frequent locking, this does not result in worse performance characteristics. This is likely because the amount of computation time greatly outweighs the amount of time needed to merge the images, so there is not much contention between threads. Shown below is the runtime needed to render cornell-box, using the same hardware, 32 spp, bidirectional path tracing with a max depth of 8, and independent sampling. Since BDPT requires more computation, we are able to get better speedup:

Scene	1 core (s)	2 cores (s)	4 cores (s)	8 cores (s)	Speedup
cornell-box	50.50	26.00	13.00	6.50	7.76x

3 GGX Microfacet BSDF Model

3.1 Feature Description

In this section, we aim to use the GGX Microfacet BSDF model to accurately model transmission through a rough refractive boundary. In Assignment 2, we already implemented the Beckmann distribution function to accomplish this, however the GGX microfacet BSDF is empirically able to provide a closer match for certain surfaces, allowing for high-fidelity and realistic rendering of rough materials at various viewing angles. GGX results in a softer highlight that falls off gradually, with improved importance sampling and better energy preservation.

BSDF models: A BSDF model (Bidirectional Scattering Distribution Function) describes how light scatters from a surface, and it is defined as the ratio of scattered radiance in direction ω_o caused by unit irradiance from direction ω_i, with respect to the local surface normal $\mathbf{n}_s$.

Microfacet models: In microfacet models, the tiny microscopic details of a rough surface are replaced by a simplified macro-level model that matches the scattering distribution of the micro-surface. Since the details of the microfacet model are too small to see directly, only the overall directional scattering pattern matters. Typically, wave effects are ignored and single scattering is assumed.

Parameterizing microfacet models: The microfacet normal distribution $D(\mathbf{m})$, expressed as a density of normals per infinitesimal solid angle, describes the distribution of surface normals $\mathbf{m}$ over the microsurface. The bidirectional shadowing-masking function $G(\omega_i, \omega_o, \mathbf{m})$ describes what fraction of the microsurface with normal $\mathbf{m}$ is visible in directions ω_i and ω_o. Together, these equations parameterize a microfacet model.

3.2 Implementation Details

Beckmann Distribution: The Beckmann distribution, which we implemented in class, is defined as follows, where we use a rational approximation for the G_1 term, and α_b is a width parameter:

$$D(\mathbf{m}) = \chi^+(\mathbf{m}\cdot\mathbf{n})\left(\frac{e^{-\tan^2\theta_m/\alpha_b^2}}{\pi\alpha_b^2\cos^4\theta_m}\right)$$

$$G_1(\mathbf{v},\mathbf{m}) = \chi^+\left(\frac{\mathbf{v}\cdot\mathbf{m}}{\mathbf{v}\cdot\mathbf{n}}\right)\begin{cases}\frac{3.53a+2.181a^2}{1+2.276a+2.577a^2} & \text{if } a<1.6\\ 1 & \text{otherwise}\end{cases}$$

Here, $D(\mathbf{m})|\mathbf{m}\cdot\mathbf{n}|$ are sampled as follows:

$$\theta_m = \arctan\sqrt{-\alpha_b^2\log(1-\xi_1)}$$

$$\phi_m = 2\pi\xi_2$$

GGX Distribution: The GGX distribution, which we implement here, is defined as follows, where α_g is a width parameter:

$$D(\mathbf{m}) = \chi^+(\mathbf{m}\cdot\mathbf{n})\left(\frac{\alpha_g^2}{\pi\cos^4\theta_m(\alpha_g^2+\tan^2\theta_m)^2}\right)$$

$$G_1(\mathbf{v},\mathbf{m}) = \chi^+\left(\frac{\mathbf{v}\cdot\mathbf{m}}{\mathbf{v}\cdot\mathbf{n}}\right)\frac{2}{1+\sqrt{1+\alpha_g^2\tan^2\theta_v}}$$

Here, $D(\mathbf{m})|\mathbf{m} \cdot \mathbf{n}|$ are sampled as follows:

$$\theta_m = \arctan\left(\frac{\alpha_g\sqrt{\xi_1}}{\sqrt{1-\xi_1}}\right)$$

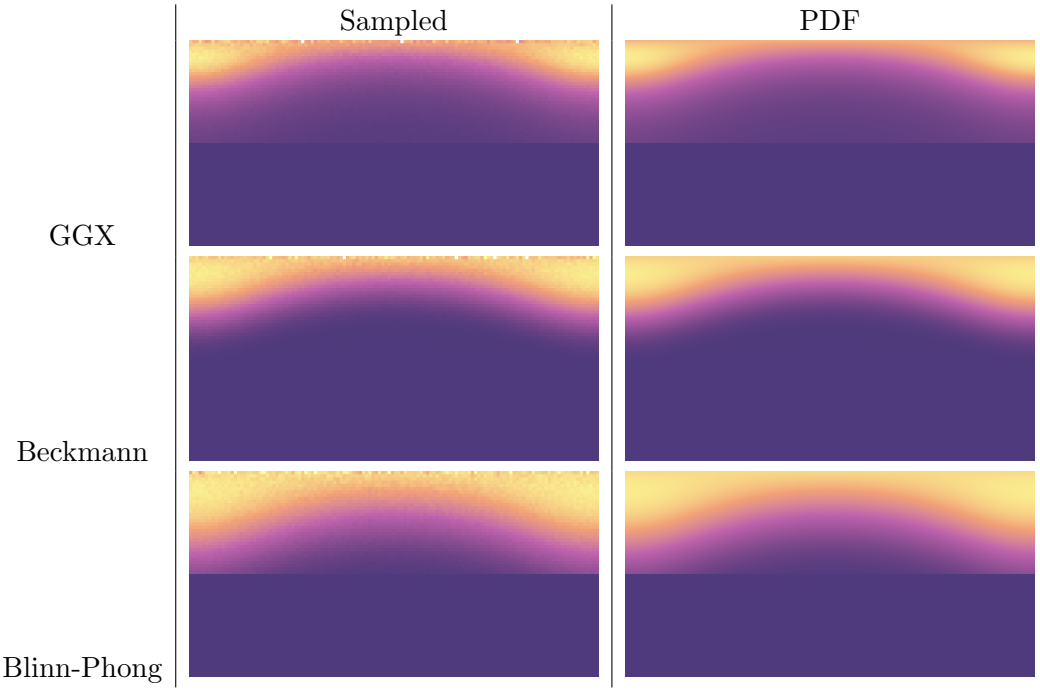
$$\phi_m = 2\pi\xi_2$$

Implementation: Beckmann and GGX have almost identical formulations except for the mathematical definitions of D and G_1, so my implementation directly follows the `Beckmann` class except for the substitutions shown above.

First, we sample normals from the GGX distribution by computing θ and ϕ from a new 2D sample. Since the distribution directly samples the normal and not the scattered direction, we must apply a change of variables. Finally, we implement the eval method by calculating the Fresnel term as in Beckmann, as well as the shadow/masking term following the math above. The χ^+ term is simply to make sure the scattered direction is in the same hemisphere as the surface normal.

3.3 Results

Correctness: To verify the correctness of my microfacet model, I used the script `src/03_sample_test.cpp` from Assignment 2 to visualize the sampled PDF. My results are shown below, in comparison to Beckmann and Blinn-Phong. Here, both Beckmann and Blinn-Phong use parameters $\alpha = 0.3$, $\eta = 0.34$, and extinction coefficient 3.9, while Blinn-Phong uses exponent 10:



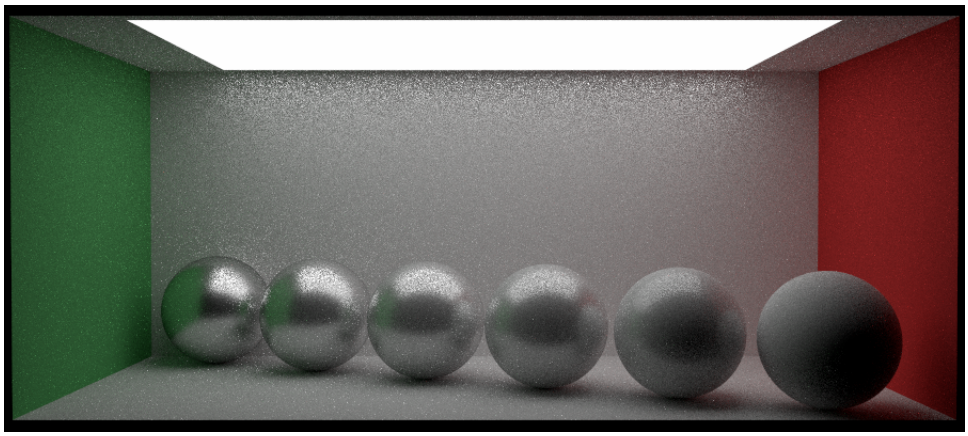
We can see the similarities between GGX and Beckmann. `03_sample_test.cpp` also reports the

following, showing that the PDFs and samples are valid:

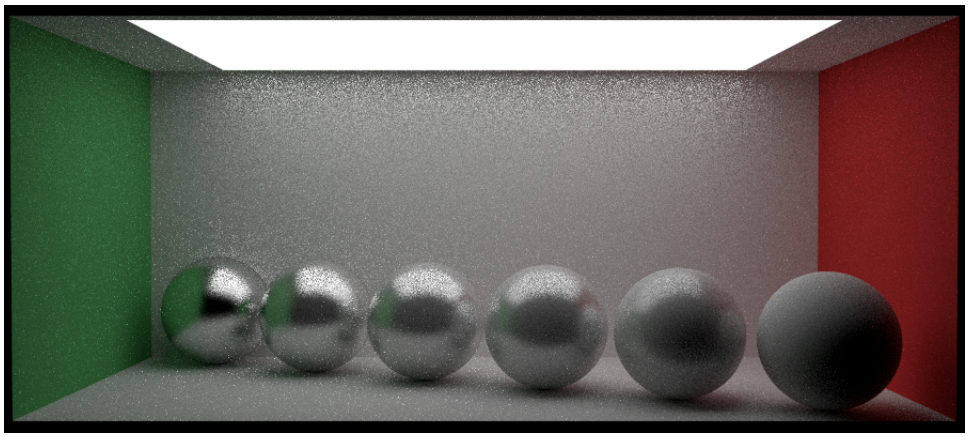
```
Running sample test for "beckmann"
Integral of PDF (should be close to 1): 1.04291
99% of samples were valid (this should be close to 100%)
--------------------------------------------------------------------------------
Running sample test for "ggx"
Integral of PDF (should be close to 1): 0.978916
91% of samples were valid (this should be close to 100%)
--------------------------------------------------------------------------------
Running sample test for "rotated-beckmann"
Integral of PDF (should be close to 1): 1.02772
98% of samples were valid (this should be close to 100%)
--------------------------------------------------------------------------------
Running sample test for "rotated-ggx"
Integral of PDF (should be close to 1): 0.950388
88% of samples were valid (this should be close to 100%)
```

Appearance: In the following images, we smoothly blend between a Lambertian sphere with albedo 0.8 and a Beckmann/GGX sphere with albedo 0.8, α 0.1, η 0.05, and extinction coefficient 3.9. These images are rendered using path tracing NEE at 64 samples per pixel:

GGX



Beckmann



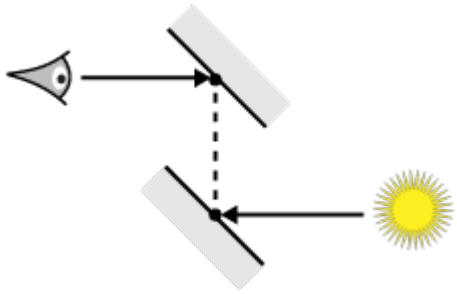
We can see that GGX is better suited for modeling rough surfaces such as unpolished metal, while Beckmann is better suited for modeling glossy and polished surfaces.

4 Bidirectional Path Tracing

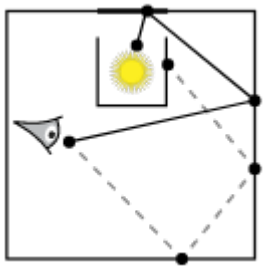
4.1 Feature Description

In this section, we aim to implement bidirectional path tracing as a generalization of the standard path tracing algorithm. BDPT constructs paths that start from the camera on one end and the light source on the other end, connecting them in the middle with a visibility ray. It better handles difficult lighting conditions and reduces the variance of the image. Unlike photon mapping, BDPT is unbiased and does not blur the scene illumination.

Camera and Light Subpaths: The core idea of BDPT is to generate camera and light subpaths by sampling an initial direction from either the camera or light, and shooting rays recursively out from them. Once this is done to form a light subpath of length s and a camera subpath of length t, we can consider all $s \cdot t$ ways to connect the camera subpath to the light subpath using an intermediate visibility ray. If there are no occlusions along the ray, then light can freely flow along the light subpath, across the visibility ray, and along the camera subpath:



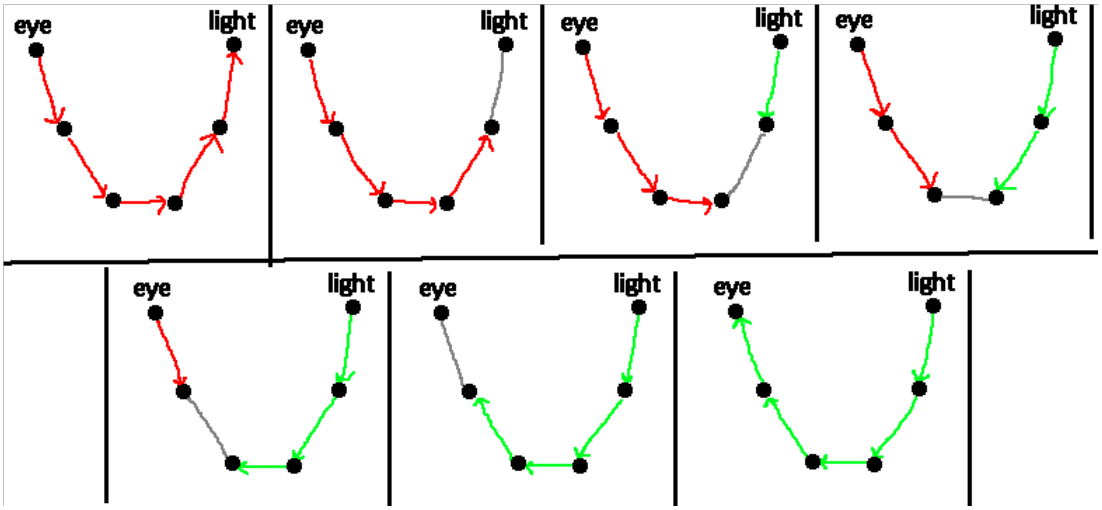
Difficult Paths: BDPT is particularly useful for difficult paths like these, where a randomly sampled ray from the camera has a very small probability of reaching the light source. In these cases, standard path tracing is inefficient as most rays carry zero radiance:



Reusing Paths: BDPT allows us to reuse the camera and light subpaths on each iteration. Once a light path of length s and a camera path of length t are constructed, then a large number of unique paths can be constructed between them, with length ranging from 2 up to $s + t$. This strategy allows us to amortize the cost of recursive raytracing to compute paths, making the most expensive part of BDPT the visibility test required to connect each pair of subpaths.

Multiple Importance Sampling: BDPT also allows us to sample the same path using multiple strategies. For any path containing n vertices, it is possible to sample that same path in $n+1$ ways using BDPT. Here, red represents the camera subpath, green represents the light subpath, and the

grey line represents the visibility ray connecting the two:



4.2 Implementation Details

Integrator Framework: Our implementation of BDPT follows from the general parallel integrator implementation that we presented earlier in this report. We split the image into 16x16 tiles and for each tile in parallel, we construct a new sampler, compute sample bounds, and loop over the pixels in the tile to render them.

Assumptions: Throughout our implementation, we assume that the camera is a pinhole perspective camera, so there is no need to consider $t = 0$ paths where the light path extends directly into the camera. Due to the difficulty of implementing and sampling alternative light sources, we also assume that the only lighting comes from diffuse area lights. We assume that there are no participating media.

Tracing Camera and Light Subpaths: To trace camera and light subpaths, we define two helper functions `generateCameraSubpath` and `generateLightSubpath` to sample an initial ray. For the camera subpath, we simply use the existing `camera.generateRay` to sample an initial ray for the camera subpath and compute the corresponding PDF. We also evaluate the importance emitted along the sampled ray. For the light subpath, we implemented a new function `sample_Le` which samples a uniformly random light source in the scene, samples a random location on that light source, then samples a cosine-weighted outgoing direction with respect to the surface normal at the samled point. The corresponding PDF is the product of the PDFs for each of these three sampling operations. We also evaluate the radiance emitted along the sampled ray.

Vertex Abstraction: To aid the rest of our BDPT implementation, we define a `Vertex` abstraction which stores a point of interaction along some subpath. These points of interaction could include geometry intersections where we need to evaluate BRDFs, camera or light endpoints, or intersections with participating media. Each vertex contains fields for the position `Vec3f p`, geometric normal `Vec3f n`, the current throughput `Color3f beta`, the forward and reverse PDF values `float pdfFwd` and `float pdfRev`, as well as any data associated with surface or other interactions.

Random Walk: After sampling the initial importance/radiance values, generating the initial rays for the camera and light subpaths, and creating initial vertices on both subpaths, both `generateCameraSubpath` and `generateLightSubpath` call a helper function `randomWalk` which iteratively traces rays and generates new vertices on the given path. This function recursively

traces a ray and upon any surface interaction, it initializes a `Vertex` with surface intersection information and samples the BSDF at the current vertex, storing forward and reverse probabilities at the current vertex and scaling the throughput value `beta` accordingly.

Random Walk: Specular Interactions Because our original implementation of DIRT treats specular and non-specular interactions differently, we must include two cases in our code to account for this. In the case that sampling the BSDF at the current vertex results in a specular interaction, we scale `beta` by the specular attenuation and simply set the next ray to the scattered direction. In this case, both the forward and reverse PDFs are 0 to represent that the BSDF is parameterized by a Dirac delta. In the non-specular case, we evaluate the BSDF at the sampled direction and scale the forward and reverse PDFs by the corresponding forward and backwards PDFs. We then scale `beta` by the BRDF value divided by the forward PDF. After all of these steps are complete, we spawn the next ray using either the scattered direction from a specular interaction or the sampled scattering direction from a non-specular interaction.

Connecting Subpaths Into Full Paths: After camera and light subpaths are sampled using the `generateCameraSubpath`, `generateLightSubpath`, and `randomWalk` helper functions, we finally consider all possible combinations of s and t and connect these paths together. This is done in a separate helper function called `connectBDPT`, which given a light subpath, camera subpath, and values of s and t, computes the radiance contribution of that particular path. We consider several special cases in this function:

- **Case 1**: In the $s = 0$ case, the light subpath is empty and we interpret the camera subpath as a complete path that connects directly to the light. This is equivalent to the path tracing setting. Here, if the last endpoint of the camera subpath is a light source, the radiance contribution is simply the amount of emitted radiance scaled by the `beta` value of the last vertex on the camera subpath.
- **Case 2**: In the $t = 1$ case, the camera subpath has one node on it and we interpret the light subpath as a nearly complete path that connects directly to a point sampled on the camera. Here, the radiance contribution is the amount of importance emitted by the camera, divided by the PDF of sampling the given outgoing ray from the camera, times the `beta` value of the last vertex on the light subpath as well as the BRDF value of that vertex with respect to its predecessor on the light subpath and its successor which is the camera node.
- **Case 3**: In the $s = 1$ case, the light subpath has one node on it and we interpret the camera subpath as a nearly complete path that connects directly to a point sampled on the light. This is equivalent to the next-event estimation setting. Here, if the last endpoint of the camera subpath is able to connect to a light source, the radiance contribution is simply the amount of radiance emitted by the light, divided by the PDF of sampling the given outgoing ray from the light, times the `beta` value of the last vertex on the camera subpath as well as the BRDF value of that vertex with respect to its predecessor on the camera subpath and its successor which is the light source.
- **Case 4**: For all other cases, we have at least that $s \geq 2$ and $t \geq 2$. In this case, we connect the two subpaths normally. If the two endpoints of the camera and light subpaths are mutually visible from each other, the radiance contribution is equal to the `beta` value of the last vertex on the camera subpath, times the `beta` value of the last vertex on the light subpath, a BRDF evaluation between the next-to-last vertex on the camera subpath, last vertex on the camera subpath, and last vertex on the light subpath, as well as a BRDF evaluation between the last vertex on the camera subpath, last vertex on the light subpath, and the next-to-last vertex on the light subpath.

After evaluating these four special cases, our `connectBDPT` function computes the MIS weight for the given connection strategies using cached `pdfFwd` and `pdfRev` values in the vertices.

Visibility Testing: In each of the four cases above, we needed to evaluate whether the last vertex on the camera subpath and the last vertex on the light subpath were mutually visible. To make this task easier, we define a helper class `VisibilityTester` that simply takes in two `Vertex` objects representing a point on some subpath. To evaluate the visibility tester, we simply spawn a ray starting at the first vertex with an upper time bound just before the second vertex, to see if there are an occluding objects in between. If so, that particular path should not be considered when computing the final contribution of that s and t combination in `connectBDPT`.

Multiple Importance Sampling: Finally, we perform multiple importance sampling following the structure in PBRT in a function called `MISWeight`. To make the code cleaner, this section makes use of a helper class called `ScopedAssignment` which temporarily performs a variable assignment while the `ScopedAssignment` is in scope, then reverts that variable assignment once the destructor is called. `MISWeight` iterates over all alternative strategies that could have hypothetically generated the same path through the scene, but with an earlier or later crossover point between the camera and light subpaths. Using the balance heuristic, it reweights the path contribution accordingly. Since my structure for this part of the code follows PBRT almost exactly, please refer to PBRT for an extended explanation for how MIS is implemented for BDPT.

4.3 Results

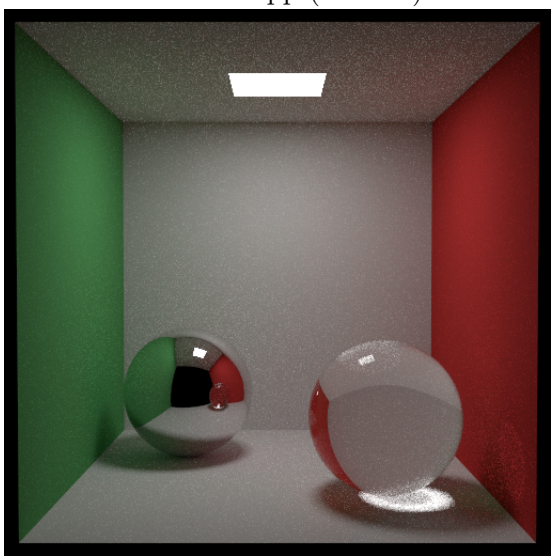
To evaluate the results of my BDPT implementation with MIS, I compare it to implementations from earlier homeworks such as regular path tracing and path tracing with next event estimation. All results in this section use an independent sampler.

NEE vs. BDPT on Cornell Box: Shown on the next page is the cornell-box scene rendered using path tracing with NEE, compared against BDPT with MIS:

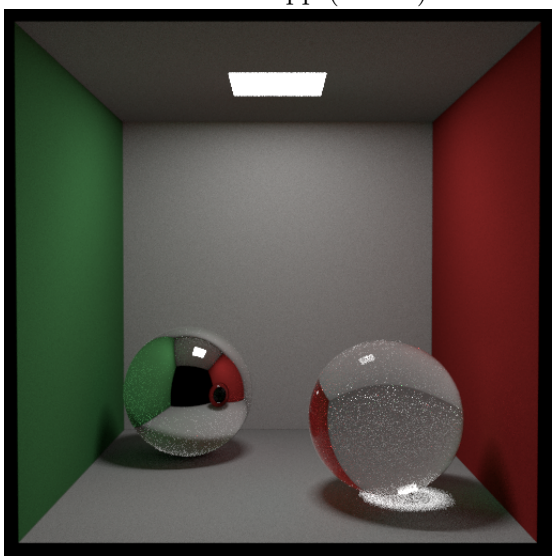
While the results are generally good, we can see that there are some subtle differences between NEE and BDPT, specifically in the handling of specular materials as reflection and refraction do not seem to be well handled. This is partially due to how DIRT handles these separately from other types of surface interactions, making it difficult to case on in the code. The green and red colors within the spherical shadowes of the NEE image are much harder to see in the BDPT image, the back wall and ceiling are slightly darker, and the reflection of the glass ball is black in the BDPT image but red in the NEE image.

During my final presentation to the class, I originally presented some results where the BDPT results looked exactly the same as NEE results, although at a much higher noise level than seen here. Unfortunately, I later found out during some debugging that my code was effectively setting the MIS weights to 0 for all cases where $s \neq 0$, $s \neq 1$, or $t \neq 1$, making it equivalent to NEE. Since those results were buggy and took a longer time to render than NEE alone, I chose not to include them in my final report.

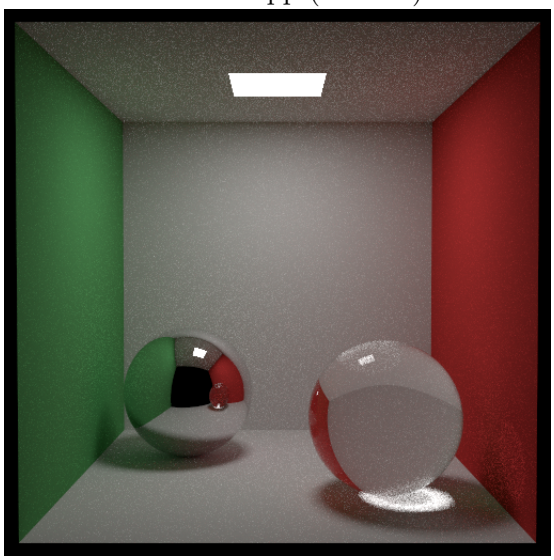
NEE 256 spp (105.00s)



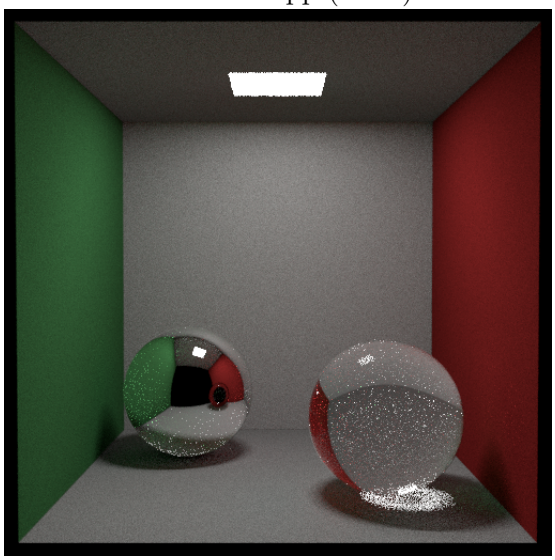
BDPT 128 spp (26.50s)



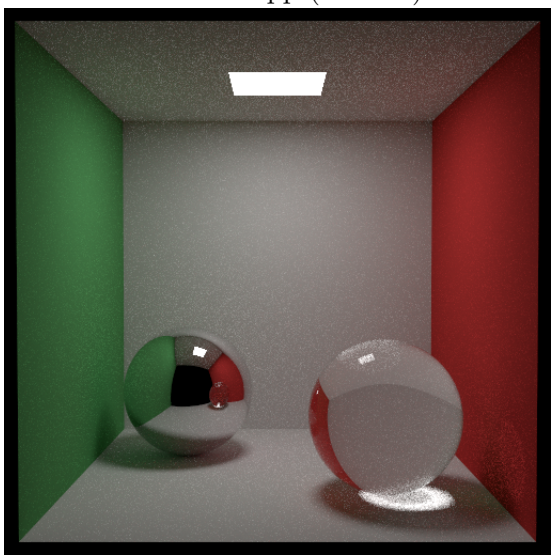
NEE 256 spp (105.00s)



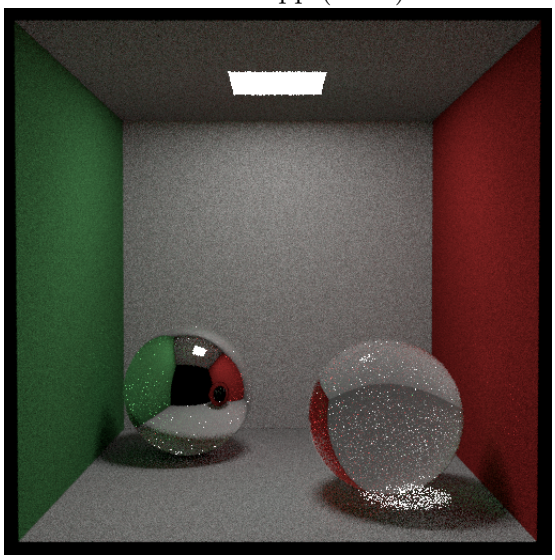
BDPT 32 spp (7.00s)



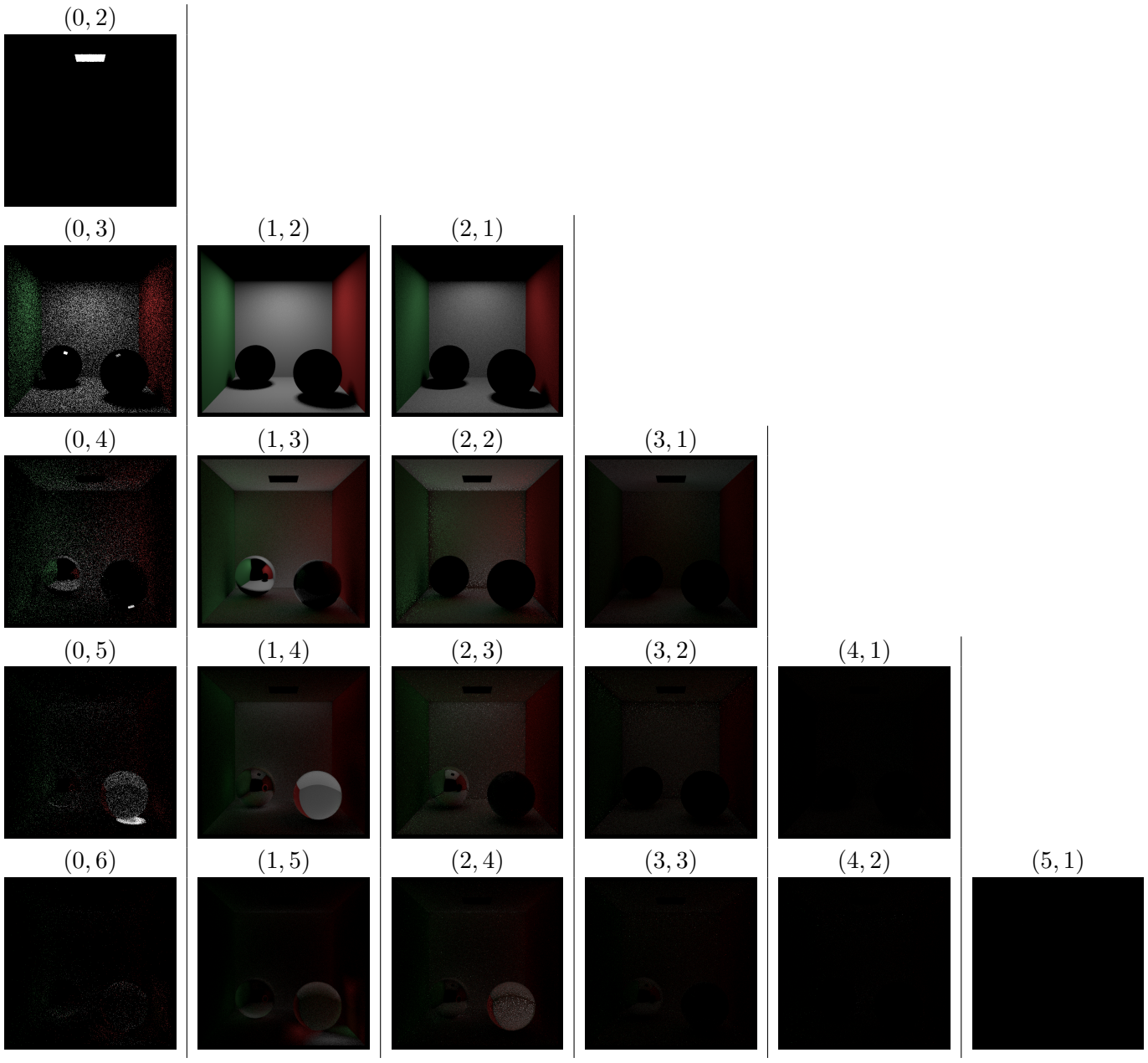
NEE 256 spp (105.00s)



BDPT 8 spp (1.75s)

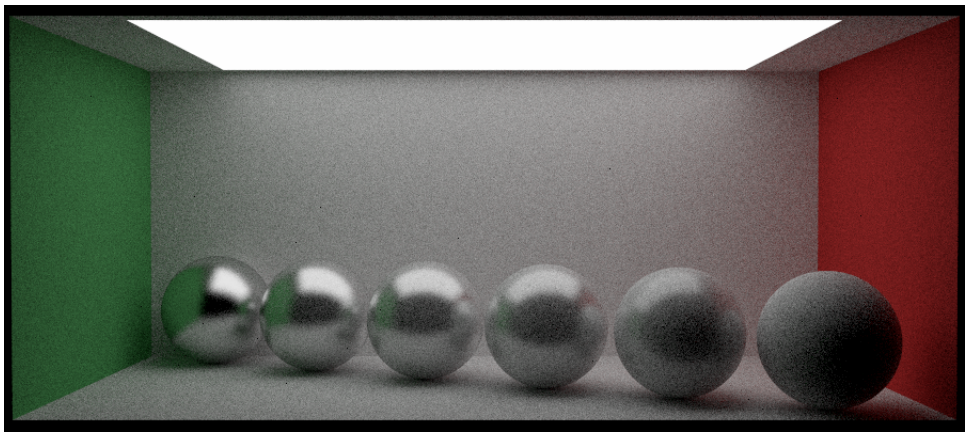


Shown below are the contributions from each (s, t) path type in BDPT, rendered at 32 spp. Here, we can clearly see that different types of paths are best at capturing different effects in the final image:

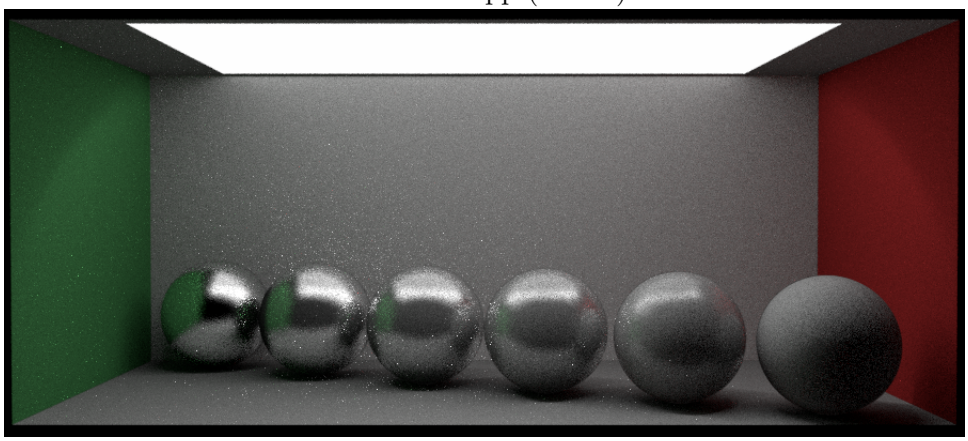


Other Scenes and Materials: Shown below are some results on other scenes and materials in comparison to the reference provided in class. Overall, the results of BDPT are a little darker than the reference and seem to miss some kinds of light interactions. My hypothesis is that due to how DIRT separately handles specular and non-specular interactions, certain classes of interactions are getting missed, causing those light paths to contribute zero radiance to the scene instead of what they are supposed to:

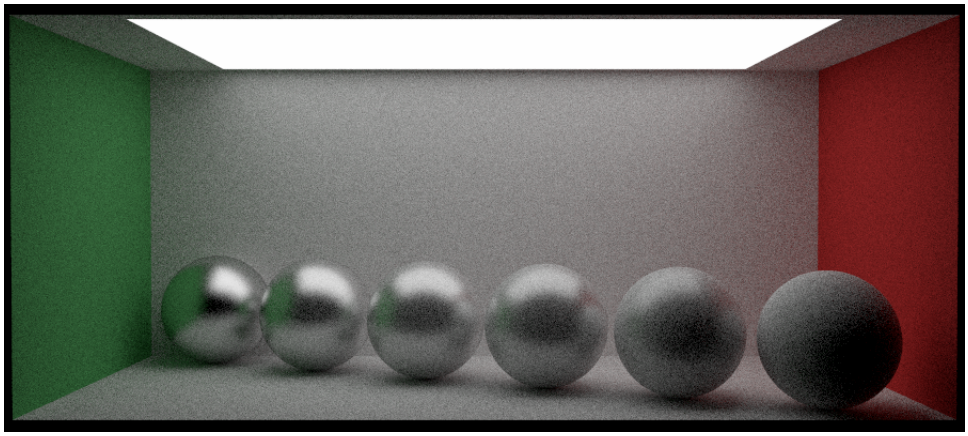
beckmann.json Reference



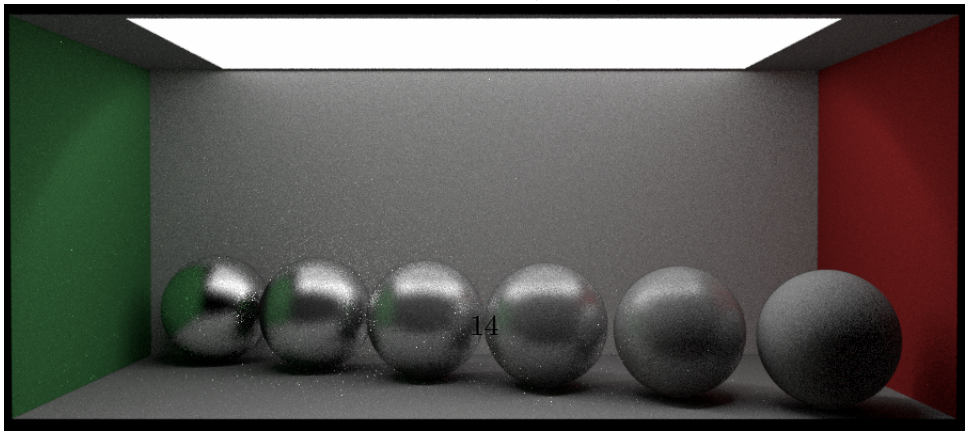
BDPT 32 spp (13.00s)



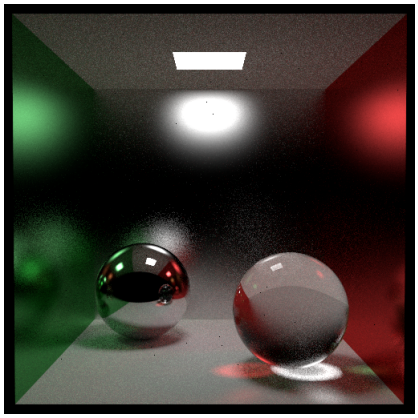
blinn-phong.json Reference



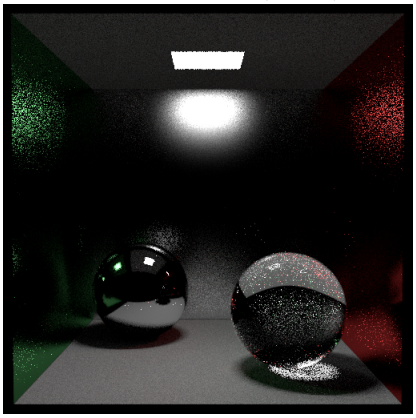
BDPT 32 spp (13.50s)



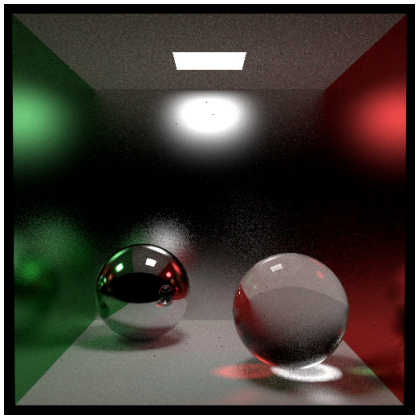
glossy-cornell-box.json Reference



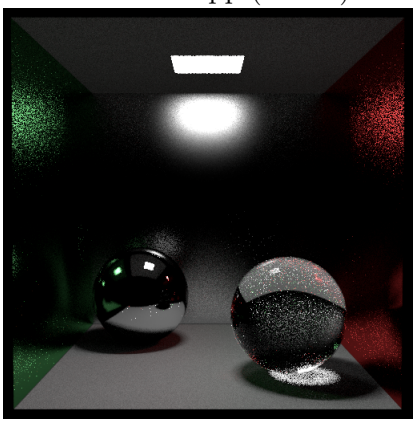
BDPT 25 spp (8.00s)



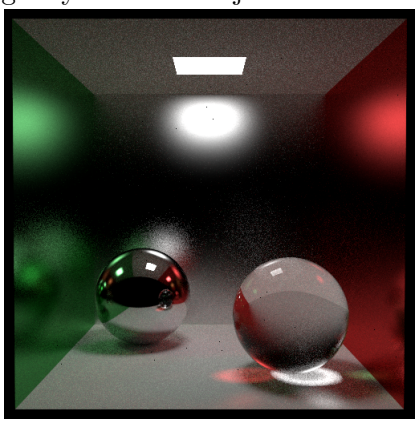
glossy-cornell-box.json Reference



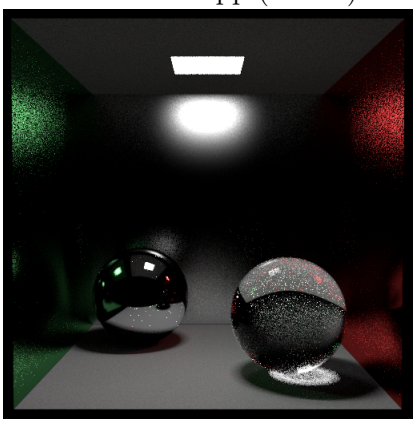
BDPT 50 spp (13.00s)



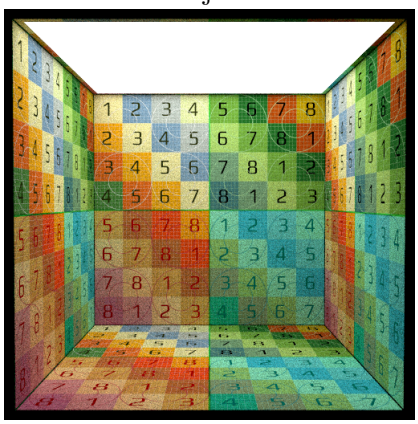
glossy-cornell-box.json Reference



BDPT 100 spp (22.50s)



textured-box.json Reference



BDPT 100 spp (22.00s)