

Bidirectional Path Tracing with MIS and Light Sources

Lingheng Tony Tao
linghent@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

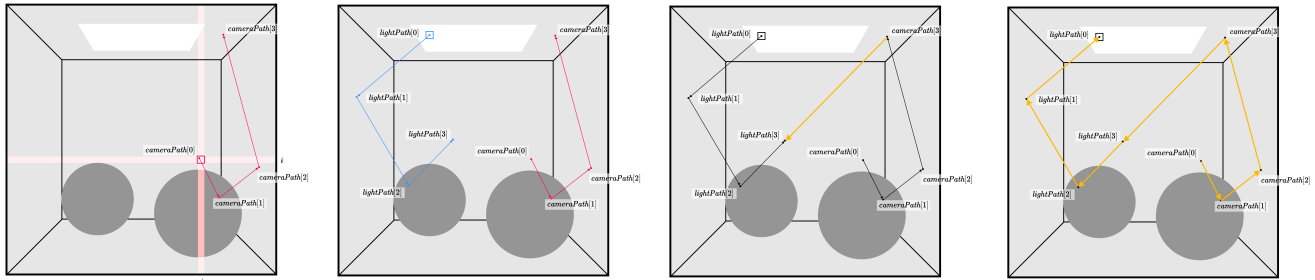


Figure 1: Bidirectional Path Tracer 4 Steps.

ABSTRACT

This report presents a comprehensive analysis of bidirectional path tracing (BDPT) integrated with multiple importance sampling (MIS) techniques, aimed at enhancing the efficiency and accuracy of rendering in computer graphics. Bidirectional path tracing is a robust light transport simulation method that combines two tracing processes: one starting from the camera and the other from the light sources. However, the integration of MIS within BDPT is critical. We discuss the theoretical foundations of MIS in the context of BDPT, highlighting how it mitigates issues related to redundant path sampling and the challenges in weighting path contributions correctly. Experimental results, based on a series of complex virtual scenes, demonstrate that the combined BDPT-MIS approach significantly outperforms standard path tracing methods in terms of both speed and image quality. The paper concludes with a discussion of practical implementation strategies and potential areas for future research in the optimization of light transport algorithms.

CCS CONCEPTS

• Computing methodologies → Ray tracing.

KEYWORDS

Ray Tracing, Bidirectional, Path Tracing, Delta Light, Multiple Importance Sampling, Rendering Equation, Global Illumination

Permission to make digital or hard copies of all or part of this work for personal or professional use is granted by ACM Publishing Department. This work is distributed as an **Unpublished working draft. Not for distribution.**

Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://www.acm.org/permissions).

CMU PBR Final Project, April 30th, 2024, Pittsburgh, PA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN CMU 15-668 Physically Based Rendering

<https://doi.org/this-is-a-placeholder>

2024-04-29 05:48. Page 1 of 1–13.

ACM Reference Format:

Lingheng Tony Tao. 2018. Bidirectional Path Tracing with MIS and Light Sources. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (CMU PBR Final Project)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/this-is-a-placeholder>

1 INTRODUCTION

Rendering realistic images through light transport simulation is pivotal in computer graphics, but it often faces challenges like high variance and slow convergence, especially in complex lighting scenarios. For example, if the area of illumination region is rather small, it can be difficult for path tracers to find a path touching the light sources. *Bidirectional path tracing* (BDPT) enhances traditional path tracing by initiating rays from both the camera and light sources, improving efficiency in difficult lighting conditions. However, BDPT can still struggle with scene complexity.

Integrating *Multiple Importance Sampling* (MIS) with BDPT optimizes the rendering process by combining different sampling strategies to reduce variance. This paper explores the theoretical and practical aspects of combining BDPT with MIS, demonstrating through experiments that this approach outperforms traditional methods in both speed and image quality. We aim to provide insights into their application and potential improvements in real-world settings.

2 LIGHT SOURCES

In order for objects in the scene to be visible, there must be a source of illumination so that some light is reflected from them to the camera sensor. In the past assignments we have implemented the light through materials. Namely, the material can be emissive and thus provide illumination. We have implemented a diffuse area light called `DiffuseLight`, inherited from the base class `Material`. It has been a good light source for providing area light, and we have been using it throughout the whole semester on the quad light in the scenes. However, the ideal *delta* light sources cannot be

59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116

implemented using `ouDiffuseLight`, technically because their fundamental difference in sampling behaviors. Therefore, before starting on the BDPT algorithms, we may want our ray tracer to provide complete support to the delta light sources as well.

2.1 DeltaPoint Class

A delta light source refers to an idealized point light source that emits light from a single, infinitesimally small point. These types of light sources are characterized by having a highly directional output, emitting light in a specific direction with a distribution that approximates a Dirac delta function. Due to their infinitesimal size, delta light sources do not have a surface area from which light can be emitted in different directions, which makes them distinct from area light sources.

In previous assignments, area lights were treated as inherently emissive materials, with materials being applied to surfaces that carry these properties. We have developed the `Surface` class to represent surface geometries. In our code, emitters have also been considered surfaces. To minimize the need for extensive code restructuring, it is logical to introduce a class representing point geometries, utilizing `SurfaceBase` as the foundation. This approach should be adopted even though a point geometry differs significantly from other geometries like quads, triangles, spheres, or meshes.

2.1.1 Declaration and Definition This section describes the declaration of the `DeltaPoint` class. It is declared in `include/dirt/quad.h` and defined in `src/quad.cpp`.

```
// quad.h
class DeltaPoint : public Surface {
public:
    Constructors
    Sample and Probability Functions
    Other Overriding Functions

protected:
    Delta Point Protected Data
};
```

There is not much to elaborate on regarding the constructors, so we will skip this section. The crucial and distinctive aspects lie in the sampling and probability density functions. These elements fundamentally differentiate point geometries from other types of geometries.

2.1.2 Sampling In the semantics of our code base, the `sample` function for a surface `A` returns a vector that points from a query point `o` to `A`. On the other hand, we would also like to get a random point on the query surface. Therefore, we should add another sampling function called `sampleOn()` in the base `SurfaceBase` class.

In `include/dirt/surface.h`, revise the code of `SurfaceBase`

```
// surface.h
SurfaceBaseClass Code Base
virtual Vec3f sampleOn(const Vec2f &sample) const
{ return Vec3f(0.f); }
virtual float pdfOn(const Vec3f &v) const
{ return 0.f; }
```

The `sampleOn()` function samples a random point on the surface, and `pdfOn()` returns the probability density of the sample generated by `sampleOn()`.

Adding these functions to the `SurfaceBase` class, we will require all the derived classes to implement their own corresponding overridden functions. This can be done by simply changing the original `sample()` function, since originally we also need to sample a point on the surface; we are basically ignoring the input query origin. In fact, we can simply return `sample(Vec3f(0.f))`.

Back to the `DeltaPoint` class. The point is representing a space that occupying zero volume. Therefore, if we want to sample a point on the point, we can only return itself.

```
// quad.h
Sample and Probability Functions
Vec3f sampleOn(const Vec2f &sample) const {
    return m_xform.point(Vec3f(0.f));
}
```

Similarly, if we want to sample a direction from a point to the delta point, we simply return the vector that points from the delta point.

```
// quad.h
Sample and Probability Functions
Vec3f sample(const Vec3f &o, const Vec2f &sample)
const {
    return m_xform.point(Vec3f(0.f)) - o;
}
```

Now, the function `pdfOn()` describes the probability density of sampling the point returned by `sampleOn()`. This should always return 1 for a delta point, as long as the input point is returned from `DeltaPoint::sampleOn()`.

```
// quad.h
Sample and Probability Functions
float pdfOn(const Vec3f &v) const {
    return 1.0f;
}
```

Finally, the function `pdf()` returns the probability density of the direction generated by `sample()`. It is clear that if a ray shooting from the query origin with direction `v` does not go through the point, it is considered a miss, therefore return 0 probability density. However, if it indeed hit the point (by checking it actually goes through the point), we return 1 in our structure. This is because this direction can only be found if we use our `sample()` function to generate this direction, considering the point is infinitesimally small.

```
// quad.h
Sample and Probability Functions
float pdfOn(const Vec3f &v) const {
    auto ray = Ray3f(o, v);
    HitInfo hit;
    if(intersect(ray, hit)) return 1.0f;
    return 0.f;
}
```

Here we used the `intersect()` function which we will cover soon.

2.1.3 Intersection The `intersect()` function trivially checks if the ray pass through the point, but since we can only save the position of our point using finite-accurate float number, we will have rounding error. We need to address this error by using a small

233 valuen when checking zero, instead of compare it directly with zero.
 234 In the semantics of our code base, we also require it to intersect()
 235 function to set up the hit information.

```

236 // quad.cpp
237 #Other Overriding Functions
238 bool DeltaPoint::intersect(const Ray3f &ray, HitInfo
239 &hit) const {
240     INCREMENT_INTERSECTION_TESTS;
241
242     // compute ray intersection (and ray parameter).
243     // continue if not hit
244     auto tray = m_xform.inverse().ray(ray);
245     float t = -tray.o.z / tray.d.z;
246     auto p = tray(t);
247     if(abs(p.x) >= Epsilon || abs(p.y) >= Epsilon) {
248         return false;
249     }
250     Vec3f gn =
251         normalize(m_xform.normal(-normalize(ray.d)));
252
253     Vec2f uv = Vec2f(0.f);
254
255     hit = HitInfo(t, m_xform.point(p), gn, gn,
256                 uv, m_material.get(),
257                 m_medium_interface.get(), this);
258
259     return true;
260 }
    
```

261 2.1.4 Bounding Box Maintaining a bounding box for a point might
 262 appear awkward, but as previously mentioned, it's necessary to
 263 account for rounding errors. Therefore, instead of defining a bound-
 264 ing box with strictly zero volume, we assign it a slightly looser
 265 boundary.

```

262 // quad.cpp
263 #Other Overriding Functions
264 Box3f DeltaPoint::localIBBox() const {
265     return Box3f(-Vec3f(Epsilon), Vec3f(Epsilon));
266 }
    
```

267 2.1.5 Private Data We will keep the similar information as what
 268 other Surfaces do. The only difference is that the size of the point
 269 should be strictly zero.

```

270 // quad.h
271 #Delta Point Protected Data
272 Vec2f m_size = Vec2f(0.f);
273 shared_ptr<const Material> m_material;
274 shared_ptr<const MediumInterface> m_medium_interface;
    
```

275 And these complete the DeltaPoint class.

276 2.2 Light Class

277 Initially, our code base did not feature a distinct class for lights; in-
 278 stead, lighting effects were achieved by applying Materials, speci-
 279 cally DiffuseLight, to surfaces. Although delta lights are typically
 280 differentiated from emissive materials in most game engines and
 281 commercial renderers, our existing DeltaPoint class treats points
 282 as a subset of Surface. Consequently, if we decide to introduce a
 283 dedicated class for lights, it would be logical to derive a Light
 284 class from Material, aligning with our current architecture and
 285 simplifying integration.

286 In include/dirt/light.h and src/light.cpp, we have the
 287 code for implementing the lights.

```

291 // light.h
292 class Light : public Material {
293 public:
294     virtual Color3f emitted(const Ray3f &ray, const
295 HitInfo &hit) const
296     { return Color3f(0.f); };
297     bool isEmissive() const override
298     { return true; };
299     virtual bool isDelta() const
300     { return false; };
301
302     Color3f emit;
303 };
    
```

304 2.3 Point Light

305 Now we have the correct geometry to represent the behavior of a
 306 point, and we can start implementing the point light source using a
 307 delta point. Following the convention, PointLights are positioned
 308 at the origin in the light space, but for the sake of convenience, we
 309 may also maintain a world space position of the point lights using
 310 a Vec3f.

311 2.3.1 Mathematical Representation For point lights, which are
 312 light sources that distribute energy uniformly across a sphere sur-
 313 rounding the light, let ϕ represent the radiant flux of the point light.
 314 Assuming uniform energy distribution across the surface of the
 315 sphere, the irradiance at any point on the sphere's surface, with
 316 radius A , is calculated by dividing the total flux by the surface area:

$$317 \quad E = \frac{\phi}{4\pi A^2} \quad (1) \quad 318$$

319 Radiance L is defined as the flux per unit solid angle per unit pro-
 320 jected area. For a point light, the projected area is effectively the
 321 same in any direction because there is no cosine term that typi-
 322 cally appears with extended surfaces. Thus, the radiance can be
 323 expressed simply as:

$$324 \quad L = \frac{\phi}{4\pi} \quad (2) \quad 325$$

326 Notice that this is irrelevant with respect to the distance r .
 327

328 2.3.2 Implementation Now we can start implementing PointLight.
 329 Obviously, PointLight is a derived class of Light, so we basically
 330 should overwrite the emitted() function. The emitted() function
 331 takes an input Ray3f ray, as well as a hit information HitInfo
 332 hit. However, according to the definition of a light with Dirac delta
 333 distribution, we know that it only emits the direction where the
 334 ray actually hits the light. Thus, the function becomes trivial.

335 In include/dirt/light.h and src/light.cpp, we have the
 336 code for implementing the PointLight.

```

337 // light.cpp
338 Color3f PointLight::emitted(const Ray3f &ray, const
339 HitInfo &hit) const {
340     if (abs(hit.p.x-position.x)<Epsilon
341         && abs(hit.p.y-position.y)<Epsilon
342         && abs(hit.p.z-position.z)<Epsilon)
343         return emit;
344     else
345         return Color3f(0.f);
346 }
    
```

2.3.3 Results Using the above implementation, we can successfully output a correct rendering of `PointLight`. For now we are using the `PathTracerMISIntegrator`, simply because we know that naive stochastic path tracer will always fail in finding a light path that goes through a delta light.

Figure 2: Point Light Cornell Box Scene (128 spp / 64 max depth) using `PathTracerMIS`

2.4 Spotlight

Spotlights serve as a practical adaptation of point lights; instead of dispersing light in all directions, they concentrate their beam within a directional cone. For ease of definition within the light coordinate system, we will place the spotlight consistently at the position $(0, 0, 0)$ and direct it along a direction vector d .

2.4.1 Mathematical Representation Spotlights are also delta lights, which means the position for a single spotlight is just the position for the illuminating point. In Figure 3, it shows the important data describing a spotlight.

Other than the intensity and position, the following quantities also identify a spotlight:

Inner angle θ_{inner} . It identifies the angular region where radiance does not falloff with respect to the smaller angle between the (reversed) incoming ray direction and the spotlight direction d .

Outer angle θ_{outer} . It identifies the angular region where it is still covered by the spotlight with respect to the smaller angle between the (reversed) incoming ray direction and the spotlight direction d , but radiance starts to falloff with respect to the cosine value of

Falloff. It describes the attenuation where $\cos\theta$ falls between $\cos\theta_{inner}$ and $\cos\theta_{outer}$.

2.4.2 Implementation Similar to the `PointLight`s, `SpotLight`s are also derived from `Light`. In the `SpotLight` class, we should also keep track of the information we mentioned in the last section.

Figure 3: Spotlight.

In `include/dirt/light.h` and `src/light.cpp`, we have the code for implementing the `PointLight`.

```
// light.h
hSpotlight Public Data
    Vec3f position;
    Vec3f direction;
    float cosInnerAngle;
    float cosOuterAngle;
    float falloff = 0.5f;

Similarly as what we have done for the PointLight, the SpotLight needs its emitted() function.

// light.h
Color3f SpotLight::emitted(const Ray3f &ray, const HitInfo &hit) const {
    if (abs(hit.p.x-position.x)<Epsilon
        && abs(hit.p.y-position.y)<Epsilon
        && abs(hit.p.z-position.z)<Epsilon) {
        Vec3f lightDir = -normalize(ray.d);
        float cosTheta = dot(lightDir, normalize(direction));

        hProcess Angular Attenuation
    }
    // return black otherwise
    return Color3f(0.f);
}
```

It is clear to see that, given the angle between the reversed ray direction and the spotlight direction, it should fall on one of the following three cases:

- Case 0 ($\cos\theta \leq \cos\theta_{inner}$). In this range, we have no angular attenuation.
- Case 02 ($\cos\theta \in [\cos\theta_{outer}, \cos\theta_{inner}]$). In this range, the ray is not covered by the spotlight.
- Case 03 (otherwise). In this range, we should have angular attenuation depending on where $\cos\theta$ locates at in the interval of $[\cos\theta_{outer}, \cos\theta_{inner}]$.

With these in mind, we should be able to finish the remaining part in `SpotLight::emitted()`.

```

465 // light.cpp
466 Process Angular Attenuation
467     float angleAttenuation;
468     if(cosTheta > cosInnerAngle) {
469         angleAttenuation = 1.0f;
470     } else if(cosTheta > cosOuterAngle) {
471         float t =
472             (cosTheta - cosOuterAngle)
473             / (cosInnerAngle - cosOuterAngle);
474         angleAttenuation = pow(t, falloff);
475     } else {
476         angleAttenuation = 0.0f;
477     }
478     return emit * angleAttenuation;
    
```

2.4.3 Result Using the above implementation, we can successfully output a correct rendering of SpotLight. Similarly we are using the PathTracerMIS integrator.

Figure 4: Spotlight Cornell Box Scene (1024 spp / 128 max depth) using PathTracerMIS

3 BIDIRECTIONAL METHOD: MATHEMATICAL REPRESENTATION

We have concluded our exploration of light sources and now have a comprehensive representation of various lighting elements within the scene. This section details the mathematical formulations underlying the basic rendering equation, path tracing, and the use of the Monte Carlo method for solving integral equations. Additionally, it introduces a bidirectional estimator.

3.1 Light Transport Equation

The Light Transport Equation (LTE), or Rendering Equation, can be used to describe outgoing radiance on any surface point. The amount of outgoing radiance $L(x, \omega)$ from point x in direction ω can be computed as the sum of emitted radiance and reflected radiance. [3]

$$L(x, \omega) = L_e(x, \omega) + \int_{\Omega} f_r(x, \omega, \omega') L(x, \omega') d\omega' \quad (3)$$

Emitted radiance $L_e(x, \omega)$ from point x in direction ω is defined only in light sources, otherwise it is zero.

$$L_e(x, \omega) = \begin{cases} L_s(x, \omega) & \text{if } x \in \text{light source} \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Reflected radiance $L_r(x, \omega)$ is computed as all incoming light in the point x , reflected in the direction ω where $L_i(x, \omega')$ represents all incoming radiance from the direction ω' . To find out how much radiance is actually reflected, it is multiplied by the bidirectional reflectance distribution function (BRDF) $f_r(x, \omega, \omega')$. Finally, it is multiplied with the dot product between the normal vector in the point x and the direction ω .

3.2 Importance Transport Equation

The problem that a global illumination algorithm must solve is to compute the light energy that is visible at every pixel in an image. Each pixel functions as a sensor with some notion of how it responds to the light energy that falls on the sensor. The response function captures this notion of the response of the sensor to the incident light energy. This response function is also called the potential function or importance by different authors. [1]

The Importance Transport Equation (ITE) is similar in form to the Light Transport Equation:

$$L(x, \omega) = L_e(x, \omega) + \int_{\Omega} f_r(x, \omega, \omega') I(x, \omega') L(x, \omega') d\omega' \quad (5)$$

Emitted importance $I_e(x, \omega)$ will capture the extent to which the surface is important to the image.

$$I(x, \omega) = \begin{cases} I_s(x, \omega) & \text{if } x \in \text{light source} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Reflected importance $I_r(x, \omega)$ is computed as all incoming importance in the point x , reflected in the direction ω . Notice the similarity to the Equation(4).

Importance flows in the opposite direction as radiance. An informal intuition for the form of the Importance Transport Equation can be obtained by considering two surfaces A and B . If surface A is visible to the eye in a particular image, then I_A will capture the extent to which the surface is important to the image (some measure of the projected area of the surface on the image). If surface B is also visible in an image and surface A reflects light to surface B , due to the importance of B , A will indirectly be even more important. Thus, while energy flows from A to B , importance flows from B to A . [1]

3.3 The Measurement Equation

The LTE formulates the steady-state distribution of light energy in the scene. The ITE formulates the relative importance of surfaces to the image. The Measurement Equation formulates the problem that a global illumination algorithm must solve. [1] This equation brings the two fundamental quantities, importance and radiance, together as follows.

For each pixel p in an image, g_p represents the measurement of radiance through that pixel. The Measurement Function h is [5]:

$$g_p = \int_{A} L(x, \omega) \cos \theta_x dA \int_{\Omega} f_r(x, \omega, \omega') L(x, \omega') d\omega' \quad (7)$$

3.4 Stochastic Path Tracing

Stochastic Path Tracing solves the Light Transport Equation (3) by using Monte Carlo integration. Instead of integrating over the whole hemisphere, this algorithm samples the hemisphere to get the single direction ω . The radiance reflected from ω is then divided by a probability density function (PDF) of the sampling $p(\omega)$, using the distribution that samples the direction.

Using Monte Carlo estimation, the Light Transport Equation is the following:

$$L(x, \omega) = L_e(x, \omega) + \int_{\Omega} p(\omega') \frac{L(x, \omega') f_r(x, \omega, \omega')}{p(\omega)} d\omega' \quad (8)$$

where $p(\omega)$ is the probability density of the direction ω being sampled. Equation (8) can be obviously evaluated in a recursive manner. A ray can be traced from the camera into the nearest hit point x , then sample a new direction ω above x , and repeat these steps until a defined maximal depth, or bounces, has been reached, or some kind of termination condition, such as Russian Roulette, has been triggered, described in Veach thesis [7].

The significant problem of the algorithm mentioned above is that many paths will never hit a light source, therefore their contribution will be zero. This problem becomes more severe when delta lights exist in the scene, because the probability of hitting the delta lights will always be zero. This can be solved by applying strategies such as Next Event Estimation (NEE) which separates the direct and indirect component from (4), such as Shirley et al. [6]. It gives us:

$$L(x, \omega) = L_{direct} + L_{indirect} \quad (9)$$

The first part of the right side of (9) represents the directional lighting. This can be computed as:

$$L_{direct} = \int_A L_e(x, \omega) \frac{f_r(x, \omega, \omega')}{p(\omega)} d\omega' \quad (10)$$

Here, $L_e(x, \omega)$ is the emitted light from some point x , ω is the normalized direction vector pointing from x to y and $f_r(x, \omega, \omega')$ is the geometric coupling term shown as the following diagram.

Figure 5: Geometric Coupling Term.

It is defined as:

$$f_r(x, \omega, \omega') = V(x, y) \frac{\cos \theta_x \cos \theta_y}{|\mathbf{j}_x| |\mathbf{j}_y|} \quad (11)$$

where $V(x, y)$ is the visibility test function, which is equal to 1 if a ray can be shot directly from x to y without hitting anything else, otherwise it is equal to 0.

3.5 Applying Path Integral

The Light Transport Equation (3), which is integrated over all directions, can be transformed with the path integral formation of the light, as presented in Veach thesis [7] into a finite equation which is integrated over surface area. So the overall problem can be rewritten as:

$$L_P = \int_{\Omega} g^1 P d\omega \quad (12)$$

In the above equation, L_P represents the radiance that flow through a pixel, and Ω are all the possible light paths in the scene. P represents a single light path, and g^1 is a measurement function of light contribution. Intuitively, this integral is describing that by taking integral over all the possible path contributions from a pixel, we will be able to evaluate the radiance passing through that pixel.

Also, Equation (12) can be approximated using Monte Carlo integration by taking the average of randomly sampled paths:

$$L_P \approx \frac{1}{N} \sum_{g=0}^N g^1 P_g \quad (13)$$

where P_g is the PDF of sampling path P_g .

As written in Veach thesis [7], the PDF is usually given in respect to the solid angle Ω , for example, when sampling a new direction using the BRDF. We need to convert into a PDF with respect to surface area A , using the Jacobian term to account for the ratio between the differential area and differential solid angle:

$$P(x, \omega) = P(\omega) \frac{|\mathbf{j}_{x,y}|}{|\mathbf{j}_x| |\mathbf{j}_y|} \quad (14)$$

Similarly, if we apply Monte Carlo integration on a concrete measurement (9) with conversion to the surface area, then we get:

$$L_P = \int_{\Omega} L_P d\omega \quad (15)$$

where L_P is a radiance measurement of pixel and P_g is the radiance of a single sample. After applying the path integral P_g we get:

$$L_P = \int_{C=0}^C C \quad (16)$$

where C is the contribution of a single path of length C and $\#$ is the maximum path length.

Now apply Monte Carlo integration on the contribution C and this will give us:

$$C = \int_{\Omega} L_e(x, \omega) \frac{f_r(x, \omega, \omega')}{p(\omega)} d\omega' \quad (17)$$

where y is a point sampled on the light source. The first part of the Equation (17) corresponds to the direct lighting from x_C where $L_e(x, \omega)$ is the amount of radiance from point x going in the direction towards x_C . Then it is multiplied with the geometric coupling term, defined in Equation (11) and also with the BRDF. The rest of the equation belongs to indirect lighting. It is computed for each vertex on the path as the product of PDF and cosine term

calculated using the dot product of the normal and unit direction vector, divided by the PDF relative to the BRDF.

3.6 Bidirectional Approach

This approach integrates the strategies of gathering and shooting rays. Gathering rays from a point on a surface is referred to as path tracing, which is defined in the preceding Equation (17). The principle of shooting rays is known as light tracing, which must be defined to complete the BDPT formulation.

As demonstrated in Veach's thesis [1], each measurement can be expressed in the form of Equation (12) using the path integral framework. Light tracing, which represents a light path from a point on the surface of a randomly selected emitter in the scene, can be formulated as:

$$I_P = \sum_{B=0}^{\Theta} \frac{I_{pixel} I_{camera}}{\cos^2 \theta_{pixel}} \quad (18)$$

where the sum represents the path from point y_0 (the first point on the light path, which is randomly sampled from one of the emitters in the scene) to the maximum path length Θ , B is the single path contribution, and the rest of the equation is the conversion from ux to radiance.

The contribution B can be computed as:

$$B = \frac{I_{y_0} \frac{1}{2C} \frac{1}{\text{light}} \prod_{i=1}^{\Theta} \frac{1}{\cos^2 \theta_i} \prod_{j=1}^{\Theta} \frac{1}{\cos^2 \theta_j}}{\prod_{i=1}^{\Theta} \frac{1}{\cos^2 \theta_i} \prod_{j=1}^{\Theta} \frac{1}{\cos^2 \theta_j}} \quad (19)$$

The first part is the emitted light I_{y_0} from light source at point y_0 . The PDF $\frac{1}{2C} \frac{1}{\text{light}}$ is probability of selecting a ray in the direction $y_0 y_1$. Considering a light that has an uniform distribution of the emitted light, then PDF is:

$$\frac{1}{2C} \frac{1}{\text{light}} = \frac{1}{2C} \quad (20)$$

where $\frac{1}{\text{light}}$ arises from uniformly choosing a point on the emitter, and $\frac{1}{2C}$ originates from uniformly selecting a direction over the hemisphere above the point y_0 . $\prod_{i=1}^{\Theta} \frac{1}{\cos^2 \theta_i}$ represents the importance of the light ray traveling directly from the light source to the pixel. This equals 1 when the ray is in front of the camera and when using a pinhole camera, where only one direction exists for each point x . Furthermore, when using a pinhole camera $\frac{1}{\cos^2 \theta_i}$ is also equal to 1, as only y_0 can be chosen as a point.

The next part $\prod_{i=1}^{\Theta} \frac{1}{\cos^2 \theta_i} \prod_{j=1}^{\Theta} \frac{1}{\cos^2 \theta_j}$ comes from a direct ray from y_B to the camera x . The remaining part of the equation is the product of the BRDF, multiplied with the cosine term and normalized by the PDF.

3.7 BDPT Estimator

We have already defined both strategies of ray evaluation: one comes from the light, the other comes from the camera. Bidirectional path tracing can then be defined as

$$H_P = \sum_{B=0}^{\Theta} \sum_{C=0}^{\Theta} F_{B \cdot C} \quad (21)$$

where B is the unweighted contribution of a path with B vertices on the light path and C vertices on the camera path. A visibility test between B^h vertex on the light path and C^h vertex on the camera path is a part of the geometric coupling term in contribution function. And finally, each path should be assigned to a weight function value $F_{B \cdot C}$.

The evaluation of $F_{B \cdot C}$ depends on the camera and light path.

$$F_{B \cdot C} = \begin{cases} 1 & \text{if } B = 0 \text{ } C = 0 \\ B^0 & \text{if } B_i \text{ } C = 0 \\ C^0 & \text{if } B = 0 \text{ } C_i \text{ } 0 \\ \dots & \text{Otherwise.} \end{cases} \quad (22)$$

We have four important cases to address for [4]:

Case 01 ($B = 0 \text{ } C = 0$): light is directly visible from the camera. The evaluation can be done with direct path between the point x and y_0 .

$$F_{00} = I_{y_0} \frac{1}{2C} \frac{1}{\text{light}}$$

Case 02 ($B_i \text{ } C = 0$): this means we have B vertices on light path and zero vertices on camera path, which represents the classic light tracing algorithm. The contribution can be evaluated using Equation (19).

$$B^0 = B$$

Case 03 ($B = 0 \text{ } C_i \text{ } 0$): this means we have zero vertices on light path and C vertices on camera path, which represents the classic path tracing algorithm. The contribution can be evaluated using Equation (17).

$$C^0 = C$$

Case 04 ($B_i \text{ } C_j \text{ } 0$): the final case is the evaluation of radiance, from camera vertices connected with light vertices of the light path, reaching the pixel.

$$F_{B \cdot C} = \frac{I_{y_0} \frac{1}{2C} \frac{1}{\text{light}} \prod_{i=1}^{\Theta} \frac{1}{\cos^2 \theta_i} \prod_{j=1}^{\Theta} \frac{1}{\cos^2 \theta_j} \prod_{k=1}^{\Theta} \frac{1}{\cos^2 \theta_k} \prod_{l=1}^{\Theta} \frac{1}{\cos^2 \theta_l}}{\prod_{i=1}^{\Theta} \frac{1}{\cos^2 \theta_i} \prod_{j=1}^{\Theta} \frac{1}{\cos^2 \theta_j} \prod_{k=1}^{\Theta} \frac{1}{\cos^2 \theta_k} \prod_{l=1}^{\Theta} \frac{1}{\cos^2 \theta_l}} \quad (23)$$

3.8 Weight Function

Many ways to create a weight function $F_{B \cdot C}$ are possible. One approach could be to define a weight function that strictly uses only camera/light paths; however, this method is quite wasteful since it discards many already sampled paths. A better approach is to employ multiple importance sampling (MIS).

It is obvious that each path with B, C vertices can be sampled in $B, C - 1$ different ways. Each path should have such weight, that together all weights sum to 1. In Veach's work [1] it is described, that the most effective way to use MIS, in order to create a weight

function, is with the power heuristic:

$$F_{B \cdot C} = \int_{g=0}^{?_B^V} \frac{?_B^V}{?_B^V} \quad (24)$$

where V is recommended to be 2, according to Veach. Taking $V = 2$, Equation (24) reduces to:

$$F_{B \cdot C} = \int_{g=0}^{?_B^2} \frac{?_B^2}{?_B^2} = \int_{g=0}^1 \frac{1}{?_B^2} \quad (25)$$

where $?_g$ is defined as the probability density for generating path $P_{B \cdot C}$ using g sub light path vertices and B, C camera sub path vertices.

$$?_g = ?_{g \cdot B \cdot C} \cdot ?_{B \cdot C} \quad (26)$$

and $?_B$ is the actual probability with which the path was generated. According to Veach [7], the current value can be set $?_B = 1$ and the values of the other $?_g$ relative to $?_B$ can be computed using ratio:

$$\frac{?_{g \cdot 1}}{?_g} = \frac{?_{g \cdot 1}^G}{?_{g \cdot 1}^C} \quad (27)$$

where $?_{g \cdot 1}^G$ is the probability density of the same path being generated from the reversed direction, thus called reversed PDF and $?_{g \cdot 1}^C$ being forward PDF accordingly.

According to Gerogiev [2], the power heuristic equation (25) can be split into two parts, determining camera and light weight independently.

$$F_{B \cdot C} = \int_{g=0}^{?_B^2} \frac{1}{?_{g \cdot B \cdot C} \cdot ?_{B \cdot C}^2} \cdot \int_{g=0}^{?_B^2} \frac{1}{?_{g \cdot B \cdot C} \cdot ?_{B \cdot C}^2} \quad (28)$$

$$= \frac{1}{F_{light \cdot B} \cdot 1 \cdot 1 \cdot F_{camera \cdot C} \cdot 1}$$

where

$$F_{camera \cdot C} = \frac{?_{g \cdot 1}^G}{?_{g \cdot 1}^C} \cdot F_{camera \cdot B} \quad (29)$$

$$F_{light \cdot B} = \frac{?_{g \cdot 1}^G}{?_{g \cdot 1}^C} \cdot F_{light \cdot C}$$

These weights can all be evaluated progressively during ray tracing algorithm.

4 BIDIRECTIONAL METHOD: IMPLEMENTATION

The implementation primarily concentrates on generating light and eye paths, computing direct illumination, and establishing vertex connections. Additionally, it includes a rudimentary though still imperfect implementation of multiple importance sampling.

4.1 The BDPTIntegrator Class and Vertex Structure

First, we present a high-level overview of the main algorithm before delving into the details, which form the critical core of the implementation.

We will follow the convention of adding the new integrator into `include/dirt/integrator.h`, and for convenience, the definitions are provided in `src/dbpt.cpp`.

```
// integrator.h
class BDPTIntegrator : public Integrator {
public:
    hConstructor()
    hRender Function()

private:
    hBDPT Helper Functions
    hBDPT Private Data
};
```

We will omit the constructor here. The Render Function should follow the same interface as before, so it will be same as other integrators.

```
// integrator.h
hRender Function
Color3f Li(const Scene &scene, Sampler &sampler,
           const Ray3f &ray) const override;
```

In order to calculate the radiance carried by the input ray, we would follow the BDPT strategy. Namely we will do this in 3 steps: trace a camera path, trace a light path, and then connect them. Before adding the helper functions, we need a data structure for describing the vertices on the light paths. Therefore, we will create a Vertex structure.

```
// integrator.h
struct Vertex {
    Vec3f wi;
    Vec3f wo;
    HitInfo hit;
    Color3f emitted;
    Color3f throughput;
    float pdfFwd;
    float pdfRev;

    Vertex() = default;
};
```

Now we can simply describe a path as an array of vertices.

```
// integrator.h
hBDPT Helper Function
vector<Vertex> traceCameraPath
(const Scene &scene, Sampler &sampler,
 const Ray3f &ray) const;
vector<Vertex> traceLightPath
(const Scene &scene, Sampler &sampler) const;
Color3f connect(const Scene &scene,
                const vector<Vertex> &camPath,
                const vector<Vertex> &lightPath,
                size_t s, size_t t) const;
```

Since we want to calculate the MIS as well, we will add the helper for computing the MIS.

```
// integrator.h
hBDPT Helper Function
float calculateMIS(const vector<Vertex> &camPath,
                  const vector<Vertex> &lightPath,
                  size_t s, size_t t) const;
```

The BDPTIntegrator, similar as other integrators, will maintain a private data for the max bounce depth.

```
// integrator.h
hBDPT Private Data
int m_maxBounces = 64;
```

4.2 Tracing A Camera Path

The camera path starts from a point on the image plane. In fact, the camera samples a ray on the pixel, and asks the integrator for the radiance carried by the ray. Therefore, we will have a deterministic starting point, which is the origin of the input ray from the camera.

The high level algorithm can be described as the following:

```

929 // bdpt.cpp
930 vector<Vertex> BDPTIntegrator::traceCameraPath
931 (const Scene &scene, Sampler &sampler,
932 const Ray3f &ray) const {
933     vector<Vertex> path;
934     Ray3f currentRay = ray;
935     Color3f throughput(1.0f);
936     Color3f result(0.0f);
937
938     float pdfFwd = 1.0f;
939     float pdfRev = 1.0f;
940
941     for(int bounce = 0; bounce < m_maxBounces; ++bounce)
942     {
943         hScene Intersection Test
944         hSample Surface
945         hModify Path Throughput
946         hRussian Roulette
947     }
948 };

```

You can find the details of each step in `src/bdpt.cpp`.

4.3 Tracing A Light Path

The light path follows almost the same steps as the camera path, except for the first vertex. The first vertex of the camera path does not fall on the lens; however, the first vertex of the light path falls on the emitter. This indicates that we need to specially process the first vertex.

The high level algorithm can be described as the following:

```

956 // bdpt.cpp
957 vector<Vertex> BDPTIntegrator::traceLightPath
958 (const Scene &scene, Sampler &sampler) const {
959     vector<Vertex> path;
960     Ray3f currentRay = ray;
961     Color3f throughput(1.0f);
962     Color3f result(0.0f);
963
964     float pdfFwd = 1.0f;
965     float pdfRev = 1.0f;
966
967     hInitialize the First Vertex in Light Path
968     for(int bounce = 0; bounce < m_maxBounces; ++bounce)
969     {
970         hScene Intersection Test
971         hSample Surface
972         hModify Path Throughput
973         hRussian Roulette
974     }
975 };

```

You can find the details of each step in `src/bdpt.cpp`. We will elaborate on the process of generating the first light vertex here. Evidently, the first vertex differs from other edges on the path in many aspects. For instance, all the other vertices on the path are sampled on the material surface, while the initial vertex is sampled directly on the light source. This necessitates a separate process for

calculating the PDF and requires a function that can sample a point on the light source and return the PDF of sampling such a point.

In the current project, we are assuming that all the emitters in the scene are diffuse area light, so we will use the PDF of sampling a direction on the hemisphere above the hit point. However, since we already have `DeltaPoint` class in hand, we can also quickly adapt the following part for delta light sources.

```

987 // bdpt.cpp
988 hInitialize the First Vertex in Light Path
989 float lightPDF;
990 Vec2f lightSample = sampler.next2D();
991 Vec3f lightPos = scene.sampleEmitters(lightSample,
992 lightPDF);
993 Vec3f tmp = randomOnUnitHemisphere(sampler.next2D());
994 Vec3f lightDir(tmp.x, -tmp.z, tmp.y);
995 float hemispherePDF = 1.0f / (2.0f * M_PI);
996
997 HitInfo initHit;
998 Ray3f currentRay(lightPos, lightDir);
999 bool lightHit = scene.intersect(currentRay, initHit);
1000
1001 if(!lightHit)
1002     // not hitting, break directly
1003     return path;
1004
1005 Ray3f revertRay(initHit.p, -lightDir,
1006 Epsilon, INFINITY);
1007 lightHit = scene.intersect(revertRay, initHit);
1008
1009 // should return true but do this for safety
1010 if(!lightHit) return path;
1011
1012 Vertex initVertex;
1013 initVertex.hit = initHit;
1014 initVertex.hit.sn = Vec3f(0.f, -1.f, 0.f);
1015 initVertex.pdfFwd = hemispherePDF * lightPDF;
1016 initVertex.pdfRev = 1.0f;
1017 initVertex.wi = Vec3f(0.f);
1018 initVertex.wo = lightDir;
1019 initVertex.throughput = throughput;
1020 initVertex.nextThroughput = throughput;
1021 Color3f initEmit =
1022     initHit.mat->emitted(revertRay, initHit);
1023 initVertex.emitted = initEmit;
1024 result = initEmit;
1025
1026 path.push_back(initVertex);
1027
1028
1029

```

4.4 Vertex Connection

The most crucial and intricate part of the BDPT algorithm is connecting the camera subpath and light subpath. Once two subpaths are successfully connected, we can evaluate the radiance carried by the entire ray using the `connect()` function. We will perform the vertex selection in the `Li()` function, which provides a nested for-loop that iterates over all the possible combinations of vertices: specifically, $B - 1$ vertices on the camera path and $L - 1$ vertices on the light path.

4.4.1 `Li()` Implementation We will start by implementing the `Li()` which provides the color to the ray tracer program.

```

1030 // bdpt.cpp
1031 Color3f BDPTIntegrator::Li
1032 (const Scene &scene, Sampler &sampler,
1033 const Ray3f &ray) const {
1034
1035
1036
1037
1038

```

```

1045 Color3f L(0.f);
1046
1047 vector<Vertex> lightPath =
1048     traceLightPath(scene, sampler);
1049 vector<Vertex> cameraPath =
1050     traceCameraPath(scene, sampler, ray);
1051
1052 for(int s = 1; s <= cameraPath.size(); ++s) {
1053     for(int t = 1; t <= lightPath.size(); ++t) {
1054         int depth = t+s-2;
1055         if((s==1 && t==1) || depth < 0
1056            || depth > m_maxBounces)
1057             continue;
1058
1059         Color3f contribution =
1060             connect(scene, cameraPath, lightPath,
1061                 s-1, t-1);
1062         float misWeight =
1063             calculateMIS(cameraPath, lightPath, s, t);
1064
1065         L += contribution * misWeight;
1066     }
1067 }
1068
1069 return L;
1070 }
    
```

4.4.2 connect() Implementation. The only remaining work here is to actually connect two subpaths. The function is src/bdpt.cpp .

```

1070 // bdpt.cpp
1071 Color3f BDPTIntegrator::connect
1072     (const Scene &scene,
1073      const vector<Vertex> &camPath,
1074      const vector<Vertex> &lightPath,
1075      size_t s, size_t t) const {
1076     Vertex cameraVertex = camPath[s];
1077     Vertex lightVertex = lightPath[t];
1078     Vec3f connectDir = lightVertex.hit.p - cameraVertex.
1079         hit.p;
1080     Ray3f ray(cameraVertex.hit.p, connectDir);
1081     HitInfo hit;
1082
1083     hVisibility Test
1084
1085     hSpecial Case Handling for (s,t) = (0,0)
1086
1087     Color3f fCamera = cameraVertex.emitted;
1088     Color3f fLight = lightVertex.emitted;
1089
1090     hSpecial Case Handling for (s,t) = {s,0}
1091     hGeneral Case Handling
1092 }
    
```

4.4.3 Visibility Test. The visibility test is required for computing the geometric coupling term in all of the cases, and is quite straightforward using a ray casting method.

If we are testing the visibility from the point x to y , we can shoot a ray from x towards y . Namely, shoot a ray of form

$$r(t) = x + t(y - x)$$

The necessity and sufficiency is that the ray should only intersect with the scene at $t = 1$, since $x + t(y - x) = y$. If an intersection is found at $t < 1$, we know that the ray sees something before seeing y , which indicates y is occluded by that object viewing from x .

```

// bdpt.cpp
hVisibility Test,
bool hitScene = scene.intersect(ray, hit);
if(!hitScene) return Color3f(0.f);
if(hit.t < 1.0f - Epsilon) return Color3f(0.f);
    
```

4.4.4 Geometric Coupling Term. Following the definition of geometric coupling term, we will add a helper function to compute it.

```

// bdpt.cpp
hBDPT Helper Function
float geometryTerm(const Vec3f &nx, const Vec3f &x,
                  const Vec3f &ny, const Vec3f &y) {
    Vec3f connectDir = y - x;
    Vec3f norm = normalize(y-x);
    float cosTheta_i =
        abs(dot(normalize(nx), norm));
    float cosTheta_o =
        abs(dot(normalize(ny), norm));

    return
        cosTheta_i * cosTheta_o
        / (length2(connectDir));
}
    
```

4.4.5 Special Case Handling. The special cases follow the exactly process as what we have discussed in Section 3.7.

```

// bdpt.cpp
hSpecial Case Handling for (s,t) = (0,0)
if(s == 0 && t == 0){
    Color3f Le = initLight.emitted;
    float g =
        geometryTerm(initLight.hit.sn,
                    initLight.hit.p,
                    camPath[0].hit.sn,
                    camPath[0].hit.p);

    return Le * g;
}

hSpecial Case Handling for (s,t) = {s,0}
if (t == 0) {
    Vec3f direction =
        initLight.hit.p - cameraVertex.hit.p;
    Ray3f shadowRay(cameraVertex.hit.p, direction,
                    Epsilon, INFINITY);
    HitInfo shadowHit;

    bool isHit =
        scene.intersect(shadowRay, shadowHit);
    Color3f Le =
        initLight.hit.mat ->
        emitted(shadowRay, shadowHit)
        / initLight.pdfFwd;

    float hemispherePDF = 1.0f / (2.0f * M_PI);
    Le *= hemispherePDF;

    Color3f bsdf =
        cameraVertex.hit.mat->
        eval(-direction,
            normalize(-cameraVertex.wi),
            cameraVertex.hit);

    float g = geometryTerm(initLight.hit.sn,
                          initLight.hit.p,
                          camPath[s].hit.sn,
                          camPath[s].hit.p);
}
    
```

```

1161     Color3f load = cameraVertex.throughput;
1162     return Le * bsdf * g * load;
1163 }

```

4.4.6 General Case and naturally the general case. Still, following what we have discussed in Section 3.7, we will be able to implement this part as well.

```

1168 // bdpt.cpp
1169 hGeneral Case Handling
1170     Color3f bsdf;
1171     if(cameraVertex.hit.mat->isEmissive())
1172         return fCamera;
1173
1174     bsdf = cameraVertex.hit.mat->
1175         eval(cameraVertex.wi,
1176             normalize(connectDir),
1177             cameraVertex.hit);
1178
1179     Color3f bsdf2 = lightVertex.hit.mat->
1180         eval(lightVertex.wo,
1181             normalize(connectDir),
1182             lightVertex.hit);
1183
1184     Color3f result = lightVertex.throughput
1185         * cameraVertex.throughput;
1186     result *= bsdf * bsdf2;
1187     result *= initLight.emitted
1188         / initLight.pdfFwd;
1189     result *= geometryTerm(cameraVertex.hit.sn,
1190         cameraVertex.hit.p,
1191         lightVertex.hit.sn,
1192         lightVertex.hit.p);
1193
1194     return result;

```

4.5 Multiple Importance Sampling

Obviously, the current $L_i()$ function is waiting for the weighting function to provide the exact weight for each path. Otherwise, simply averaging the path results or adding them will not reduce the variance. As what we shown in Section 3.8, since we have $pdfRev$ and $pdfFwd$ already computed during the path construction, the weighting function will be conceptually easy.

```

1199 // bdpt.cpp
1200 hBDPT Helper Function
1201     float BDPTIntegrator::calculateMIS
1202     (const vector<Vertex> &camPath,
1203     const vector<Vertex> &lightPath,
1204     size_t s, size_t t) const {
1205
1206         if (s == 1 && t == 1)
1207             return 1.f;
1208
1209         if (s > 1 && t == 1) {
1210             float wCamera = 1.0f;
1211             for (int i = 1; i <= s; i++) {
1212                 const Vertex& vertex = camPath[i];
1213                 wCamera =
1214                     (remap0(vertex.pdfRev)
1215                     / remap0(vertex.pdfFwd))
1216                     * (wCamera + 1.0f);
1217             }
1218             return 1.0f / (wCamera + 1.0f);
1219         }
1220
1221         if (s == 1 && t > 1) {

```

```

1219         float wLight = 1.0f;
1220         for (int i = 1; i <= t; ++i) {
1221             const Vertex& vertex = lightPath[i];
1222             wLight = (remap0(vertex.pdfRev)
1223                 / remap0(vertex.pdfFwd))
1224                 * (wLight + 1.0f);
1225         }
1226         return 1.0f / (1.0f + wLight);
1227     }
1228
1229     float sum_wlight = 1.f;
1230     float sum_wcamera = 1.f;
1231
1232     for(int i = 1; i <=s; i++) {
1233         sum_wcamera = (remap0(camPath[i].pdfRev)
1234             / remap0(camPath[i].pdfFwd))
1235             * (sum_wcamera + 1.0f);
1236     }
1237
1238     for(int i = 1; i <= t; i++) {
1239         sum_wlight =
1240             (remap0(lightPath[i].pdfRev)
1241             / remap0(lightPath[i].pdfFwd))
1242             * (sum_wlight + 1.0f);
1243     }
1244
1245     return
1246         1.f / (sum_wlight + 1 + sum_wcamera);
1247 }

```

5 BIDIRECTIONAL METHOD: RESULTS

Employing the aforementioned implementation, we can successfully generate a rendering using the BDPTIntegrator.

Figure 6: Arithmetically Averaged BDPT Cornell Box Scene (128 spp / 64 max depth) using BDPTIntegrator.

Although there are still some inaccuracies in the current implementation, such as incorrect shadow shapes and imprecise weights for path construction, and we lack the support for automatically returning the PDF for light sources, these issues can be addressed

1277 with an acceptable amount of error in future developments. De-
 1278 spite these limitations, the current bidirectional path tracer already
 1279 demonstrates some advantages over the traditional stochastic path
 1280 tracer.

1281
 1282
 1283
 1284
 1285
 1286
 1287
 1288
 1289
 1290
 1291
 1292
 1293
 1294
 1295
 1296
 1297
 1298
 1299

1300 Figure 7: Weighted BDPT Cornell Box Scene (128 spp / 64
 1301 max depth) using BDPTIntegrator.

1302
 1303 Notice the white noises and incorrect shadow positions com-
 1304 pared to the below reference image generated by PathTracerMIS
 1305
 1306
 1307
 1308
 1309
 1310
 1311
 1312
 1313
 1314
 1315
 1316
 1317
 1318
 1319
 1320
 1321
 1322
 1323
 1324
 1325

1326 Figure 8: Reference Cornell Box Scene (128 spp / 64 max
 1327 depth) using PathTracerMIS

1328
 1329
 1330 **6 CONCLUSION**

1331 In conclusion, the implementation of Bidirectional Path Tracing
 1332 (BDPT) with Multiple Importance Sampling (MIS) and the inclu-
 1333 sion of delta light sources, such as point lights and spotlights, have
 1334

1335 significantly enhanced the rendering capabilities of the system.
 1336 BDPT+MIS allows for efficient exploration of the path space by
 1337 combining the strengths of both camera and light paths, reduc-
 1338 ing variance and improving convergence rates. The incorporation
 1339 of delta light sources enables the accurate simulation of common
 1340 lighting scenarios found in real-world environments. Point lights
 1341 provide a simple yet effective way to represent omnidirectional
 1342 light sources, while spotlights offer directional control and the abil-
 1343 ity to create focused illumination. The successful integration of
 1344 these techniques has resulted in a robust and versatile rendering
 1345 framework capable of producing high-quality images with realis-
 1346 tic lighting effects. The implementation of BDPT+MIS and delta
 1347 light sources demonstrates the potential for further advancements
 1348 in physically-based rendering and opens up new possibilities for
 1349 creating visually compelling and accurate simulations of complex
 1350 lighting scenarios.
 1351

1352 **A RENDERING COMPETITION**

1353 For the rendering competition, I rendered a breakfast scene from
 1354 the author Wig42. The scene objects are available to be found
 1355 in report/final/Scenes and Jsons . I have rendered using the
 1356 Path Tracer with MIS as well as the Bidirectional Path Tracer with
 1357 MIS, to show that quality and potential incorrectness in my current
 1358 implementation.

1359 The following picture is rendered using BDPTIntegrator with
 1360 MIS.

1361
 1362
 1363
 1364
 1365
 1366
 1367
 1368
 1369
 1370
 1371
 1372
 1373
 1374
 1375
 1376
 1377
 1378
 1379
 1380
 1381
 1382
 1383
 1384

1385 Figure 9: Breakfast Scene (1024 spp / 256 max depth) using
 1386 BDPTIntegrator.

1387
 1388 Since the light sampling and PDF is not yet supported by the
 1389 current implementation of BDPT, the spotlight in the scene has
 1390 been removed from the scene. The following picture is rendered
 1391

