Bidirectional Path Tracing with MIS and Light Sources

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

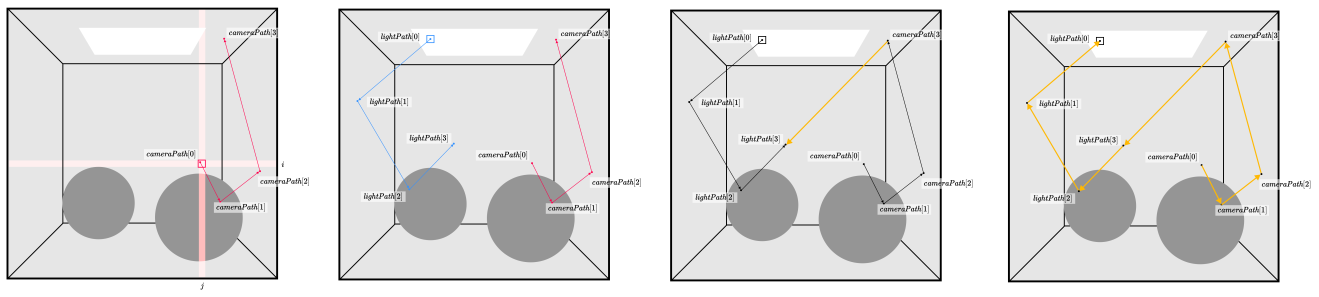

Figure 1: Bidirectional Path Tracer 4 Steps.

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

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
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

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 emai (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 implemented using our `DiffuseLight`, technically because their foundamental 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 $\mathcal{A}$ returns a vector that points from a query point o to $\mathcal{A}$. 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
⟨SurfaceBase Class 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 **o** to the delta point, we simply return the vector that points from **o** to 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 **o** 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 Query. The `intersect()` function trivially checks if the ray pass through the point, but since we can only save the position of our point using finitely-accurate float number, we will have rounding error. We need to address this error by using a small value ϵ when checking zero, instead of compare it directly with zero. In the semantics of our code base, we also require the `intersect()` function to set up the hit information.

```
// quad.cpp
⟨Other Overriding Functions⟩+ ≡
    bool DeltaPoint::intersect(const Ray3f &ray, HitInfo
     &hit) const {
        INCREMENT_INTERSECTION_TESTS;

        // compute ray intersection (and ray parameter).
        // continue if not hit
        auto tray = m_xform.inverse().ray(ray);
        float t = -tray.o.z / tray.d.z;
        auto p = tray(t);
        if(abs(p.x) >= Epsilon || abs(p.y) >= Epsilon) {
            return false;
        }
        Vec3f gn =
            normalize(m_xform.normal(-normalize(ray.d)));

        Vec2f uv = Vec2f(0.f);

        hit = HitInfo(t, m_xform.point(p), gn, gn,
                      uv, m_material.get(),
                      m_medium_interface.get(), this);
        return true;
    }
```

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

```
// quad.cpp
⟨Other Overriding Functions⟩+ ≡
    Box3f DeltaPoint::localBBox() const {
        return Box3f(-Vec3f(Epsilon), Vec3f(Epsilon));
    }
```

2.1.5 Private Data. We will keep the similar information as what other `Surface`s do. The only difference is that the size of the point should be strictly zero.

```
// quad.h
⟨Delta Point Protected Data⟩+ ≡
    Vec2f m_size = Vec2f(0.f);
    shared_ptr<const Material> m_material;
    shared_ptr<const MediumInterface> m_medium_interface;
```

And these complete the `DeltaPoint` class.

2.2 Light Class

Initially, our code base did not feature a distinct class for lights; instead, lighting effects were achieved by applying `Material`s, specifically `DiffuseLight`, to surfaces. Although delta lights are typically differentiated from emissive materials in most game engines and commercial renderers, our existing `DeltaPoint` class treats points as a subset of `Surface`. Consequently, if we decide to introduce a dedicated class for lights, it would be logical to derive this `Light` class from `Material`, aligning with our current architecture and simplifying integration.

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

```
// light.h
class Light : public Material {
public:
    virtual Color3f emitted(const Ray3f &ray, const
     HitInfo &hit) const
        { return Color3f(0.f); };
    bool isEmissive() const override
        { return true; }
    virtual bool isDelta() const
        { return false; }

    Color3f emit;
};
```

2.3 Point Light

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

2.3.1 Mathematical Representation. For point lights, which are light sources that distribute energy uniformly across a sphere surrounding the light, let Φ represent the radiant flux of the point light. Assuming uniform energy distribution across the surface of the sphere, the irradiance E at any point on the sphere's surface, with radius r, is calculated by dividing the total flux by the surface area:

$$E = \frac{\Phi}{4\pi r^2}. \tag{1}$$

Radiance L is defined as the flux per unit solid angle per unit projected area. For a point light, the projected area is effectively the same in any direction because there is no cosine term that typically appears with extended surfaces. Thus, the radiance can be expressed simply as:

$$L = \frac{\Phi}{4\pi}. \tag{2}$$

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

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

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

```
// light.cpp
Color3f PointLight::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)
        return emit;
    else
        return Color3f(0.f);
}
```

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

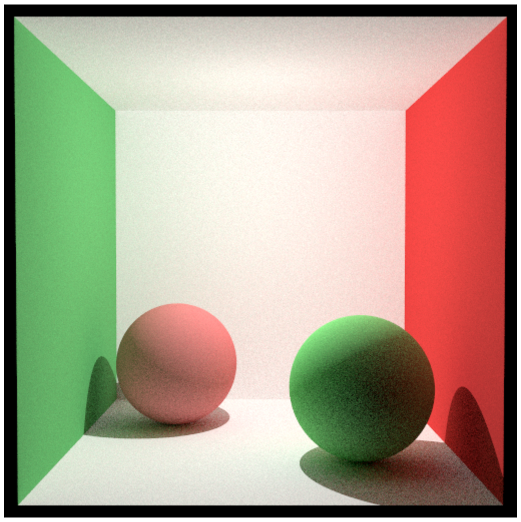

Figure 2: Point Light Cornell Box Scene (128 spp / 64 max depth) using `PathTracerMIS`.

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 $\mathbf{d}$.

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 identifies 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 $\mathbf{d}$.
- **Outer angle** θ_{outer}. It identifies the angular region where it is still covered by the spotlight with respect to the smaller angle θ between the (reversed) incoming ray direction and the spotlight direction $\mathbf{d}$, but radiance starts to falloff with respect to the cosine value of θ.
- **Falloff.** It describes the attenuation where $\cos\theta$ falls between $\cos\theta_{\text{inner}}$ and $\cos\theta_{\text{outer}}$.

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.

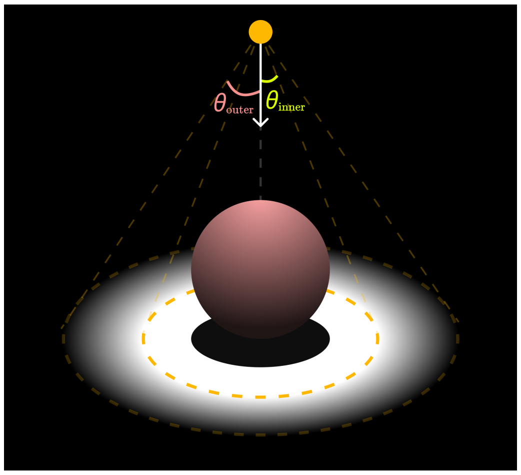

Figure 3: Spotlight.

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

```
// light.h
⟨Spotlight 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 own `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));

        ⟨Process Angular Attenuation⟩
    }
    // return black otherwise
    return Color3f(0.f);
}
```

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

- **Case 01**($\cos\theta > \cos\theta_{\text{inner}}$). In this range, we have no angular attenuation.
- **Case 02**($\cos\theta < \cos\theta_{\text{outer}}$). 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 $[\cos\theta_{\text{outer}}, \cos\theta_{\text{inner}}]$.

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

```
// light.cpp
⟨Process Angular Attenuation⟩+ ≡
   float angleAttenuation;
    if(cosTheta > cosInnerAngle) {
        angleAttenuation = 1.0f;
    } else if(cosTheta > cosOuterAngle) {
        float t =
            (cosTheta - cosOuterAngle)
          / (cosInnerAngle - cosOuterAngle);

        angleAttenuation = pow(t, falloff);
    } else {
        angleAttenuation = 0.0f;
    }

    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.

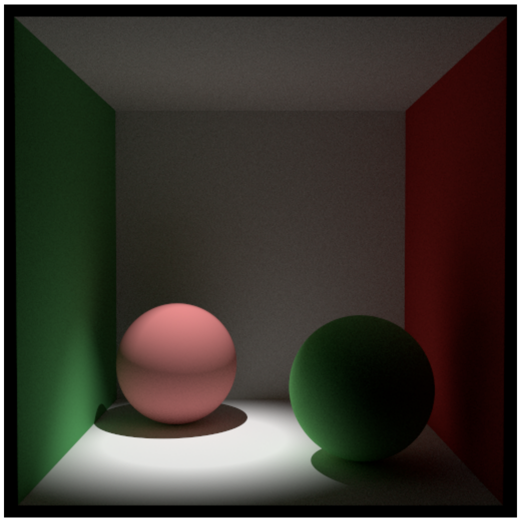

Figure 4: Spotlight Cornell Box Scene (1024 spp / 128 max depth) using `PathTracerMIS`.

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 $L(\mathbf{x}, \boldsymbol{\omega}_o)$ from point $\mathbf{x}$ in direction $\boldsymbol{\omega}_o$ can be computed as the sum of emitted radiance and reflected radiance. [3]

$$L(\mathbf{x}, \boldsymbol{\omega}_o) = L_e(\mathbf{x}, \boldsymbol{\omega}_o) + L_r(\mathbf{x}, \boldsymbol{\omega}_o) \tag{3}$$

Emitted radiance $L_e(\mathbf{x}, \boldsymbol{\omega}_o)$ from point $\mathbf{x}$in direction $\boldsymbol{\omega}_o$ is defined only in light sources, otherwise it is zero.

$$L_r(\mathbf{x}, \boldsymbol{\omega}_o) = \int_\Omega L(\mathbf{x}', -\boldsymbol{\omega}_i) f_r(\mathbf{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) |\mathbf{n}_\mathbf{x} \cdot \boldsymbol{\omega}_i| \mathrm{d}\boldsymbol{\omega}_i \tag{4}$$

Reflected radiance $L_r(\mathbf{x}, \boldsymbol{\omega}_o)$ is computed as all incoming light in the point $\mathbf{x}$, reflected in the direction $\boldsymbol{\omega}_o$ where $L(\mathbf{x}', -\boldsymbol{\omega}_i)$ represents all incoming radiance from the direction $\boldsymbol{\omega}_i$. To find out how much radiance is actually r eflected, it is multiplied by the *bidirectional reflectance distribution function* (BRDF) $f_r(\mathbf{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o)$. Finally, it is multiplied with the dot product between the normal vector in the point $\mathbf{x}$and the direction $\boldsymbol{\omega}_i$.

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:

$$W(\mathbf{x}, \boldsymbol{\omega}_o) = W_e(\mathbf{x}, \boldsymbol{\omega}_o) + W_r(\mathbf{x}, \boldsymbol{\omega}_o) \tag{5}$$

Emitted importance $W_e(\mathbf{x}, \boldsymbol{\omega}_o)$ will capture the extent to which the surface is important to the image.

$$W_r(\mathbf{x}, \boldsymbol{\omega}_o) = \int_\Omega W(\mathbf{x}', -\boldsymbol{\omega}_i) f_r(\mathbf{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) |\mathbf{n} \cdot \boldsymbol{\omega}_i)| \mathrm{d}\boldsymbol{\omega}_i \tag{6}$$

Reflected importance $W_r(\mathbf{x}, \boldsymbol{\omega}_o)$ is computed as all incoming importance in the point $\mathbf{x}$, reflected in the direction $\boldsymbol{\omega}_o$. 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 $\mathcal{A}$ and $\mathcal{B}$. If surface $\mathcal{A}$ is visible to the eye in a particular image, then $W_e(\mathcal{A})$ 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 $\mathcal{B}$ is also visible in an image and surface $\mathcal{A}$ reflects light to surface $\mathcal{B}$, due to the importance of $\mathcal{B}$, $\mathcal{A}$ will indirectly be even more important. Thus, while energy flows from $\mathcal{A}$ to $\mathcal{B}$, importance flows from $\mathcal{B}$ to $\mathcal{A}$. [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 j in an image, M_j represents the measurement of radiance through that pixel j. The Measurement Function M is [5]:

$$M_j = \int_{\mathcal{A}_\text{film}} \int_\Omega W_e^{(j)}(\mathbf{x}_\text{film}, \boldsymbol{\omega}) L_i(\mathbf{x}_\text{film}, \boldsymbol{\omega}) |\mathbf{n}_{\mathbf{x}_\text{film}} \cdot \boldsymbol{\omega}| \mathrm{d}\boldsymbol{\omega} \mathrm{d}\mathcal{A}_\text{film} \tag{7}$$

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 $\boldsymbol{\omega}_i$. The radiance reflected from $\boldsymbol{\omega}_i$ is then divided by a probability density function (PDF) of the sampling $\boldsymbol{\omega}_i$, using the distribution that samples the direction.

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

$$L(\mathbf{x}, \boldsymbol{\omega}_o) = L_e(\mathbf{x}, \boldsymbol{\omega}_o) + \frac{L(\mathbf{x}', -\boldsymbol{\omega}_i) f_r(\mathbf{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) |\mathbf{n}_\mathbf{x} \cdot \boldsymbol{\omega}_i|}{p(\boldsymbol{\omega}_i)} \tag{8}$$

where $p(\boldsymbol{\omega}_i)$ is the probability density of the direction $\boldsymbol{\omega}_i$ 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 $\mathbf{x}$, then sample a new direction $\boldsymbol{\omega}_i$ above $\mathbf{x}$, 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:

$$L_r(\mathbf{x}, \boldsymbol{\omega}_o) = L_{\text{direct}} + L_{\text{indirect}} \tag{9}$$

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

$$L_{\text{direct}} = \int_{\mathcal{A}} L_e(\mathbf{y} \to \mathbf{x}) f_r(\mathbf{x}, \overrightarrow{\mathbf{xy}}) G(\mathbf{x}, \mathbf{y}) d\mathcal{A}_i \tag{10}$$

Here, $L_e(\mathbf{y} \to \mathbf{x})$ is the emitted light from some point $\mathbf{y}$, $\overrightarrow{\mathbf{xy}}$ is the normalized direction vector pointing from $\mathbf{x}$ to $\mathbf{y}$ and $G(\mathbf{x}, \mathbf{y})$ is the *geometric coupling term*, shown as the following diagram.

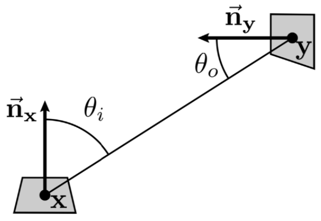

Figure 5: Geometric Coupling Term.

It is defined as:

$$G(\mathbf{x}, \mathbf{y}) = V(\mathbf{x}, \mathbf{y}) \frac{|\mathbf{n}_\mathbf{x} \cdot \overrightarrow{\mathbf{xy}}||\mathbf{n}_\mathbf{y} \cdot \overrightarrow{\mathbf{yx}}|}{||\mathbf{x} - \mathbf{y}||^2} \tag{11}$$

where $V(\mathbf{x}, \mathbf{y})$ is the visibility test function, which is equal to 1 if a ray can be shot directly from $\mathbf{x}$ to $\mathbf{y}$ 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:

$$L_\mathcal{P} = \int_D M_j(\mathcal{P}) \mathrm{d}D \tag{12}$$

In the above equation, L_q represents the radiance that flow through a pixel, and D are all the possible light paths in the scene. $\mathcal{P}$ presents a single light path, and M_j 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 N randomly sampled paths:

$$\langle L_\mathcal{P} \rangle = \frac{1}{N} \sum_{i=0}^{N} \frac{M_j(\mathcal{P}_i)}{p(\mathcal{P}_i)} \tag{13}$$

where $p(\mathcal{P}_i)$ is the PDF of sampling path $\mathcal{P}_i$.

As written in Veach thesis[7], the PDF is usually given in respect to the solid angle $p(\boldsymbol{\omega})$, for example, when sampling a new direction using the BRDF. We need to convert into a PDF with respect to surface area $p(\mathbf{x})$, using the Jacobian term to account for the ratio between the differential area and differential solid angle:

$$p(\mathbf{x}) = p(\boldsymbol{\omega}) \left(\frac{|\mathbf{n}_\mathbf{x} \cdot \overrightarrow{\mathbf{xy}}|}{||\mathbf{x} - \mathbf{y}||^2} \right) \tag{14}$$

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

$$L_\mathcal{P} = \frac{1}{N} \sum_{i=0}^{N} L_{\mathcal{P}_i} \tag{15}$$

where $L_\mathcal{P}$ is a radiance measurement of pixel and $L_{\mathcal{P}_i}$ is the radiance of a single sample. After applying the path integral on $L_{\mathcal{P}_i}$ we get:

$$L_{\mathcal{P}_i} = \sum_{t=0}^{N_E} C_t \tag{16}$$

where C_t is the contribution of a single path of length t and N_E is the maximum path length.

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

$$\begin{aligned} C_t = {} & \frac{L_e(\mathbf{y}, \overrightarrow{\mathbf{yx}_t})}{p(\mathbf{y})} f_r(\mathbf{x}_t, \overrightarrow{\mathbf{yx}_t}, \overrightarrow{\mathbf{x}_t\mathbf{x}_{t-1}}) G(\mathbf{x}_t, \mathbf{y}) \\ & \left(\prod_{i=1}^{t-1} \frac{f_r(\mathbf{x}_i, \overrightarrow{\mathbf{x}_{i+1}\mathbf{x}_i}, \overrightarrow{\mathbf{x}_i\mathbf{x}_{i-1}}) |\mathbf{n}_{\mathbf{x}_i} \cdot \overrightarrow{\mathbf{x}_i\mathbf{x}_{i+1}}|}{p(\overrightarrow{\mathbf{x}_{i+1}\mathbf{x}_i})} \right) \end{aligned} \tag{17}$$

where $\mathbf{y}$ is a point sampled on the light source. The first part of the Equation (17) corresponds to the direct lighting from $\mathbf{y}$ to $\mathbf{x}_t$, where $L_e(\mathbf{y}, \overrightarrow{\mathbf{yx}_t})$ is the amount of radiance from point $\mathbf{y}$ going in the direction towards $\mathbf{x}_t$. 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 [7], 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:

$$L_{\mathcal{P}} = \sum_{s=0}^{N_L} C_s \frac{||P_{\text{pixel}} - P_{\text{camera}}||^2}{\cos^2 \theta A_{\text{pixel}}} \tag{18}$$

where the sum represents the path from point $\mathbf{y}_0$ (the first point on the light path, which is randomly sampled from one of the emitters in the scene) to the maximum path length N_L, C_s is the single path contribution, and the rest of the equation is the conversion from flux to radiance.

The contribution C_s can be computed as:

$$C_s = \frac{L_e(\mathbf{y}_0, \overrightarrow{\mathbf{y}_0\mathbf{y}_1})}{p(\mathbf{y}_0, \overrightarrow{\mathbf{y}_0\mathbf{y}_1})} f_r(\mathbf{y}_s, \overrightarrow{\mathbf{y}_{s-1}\mathbf{y}_s}, \overrightarrow{\mathbf{y}_s\mathbf{y}_{s+1}}) G(\mathbf{y}_s, \mathbf{x}) \\ \frac{W_e(\mathbf{x}, \overrightarrow{\mathbf{y}_s\mathbf{x}})}{p(\mathbf{x})} \left(\prod_{i=1}^{s-1} \frac{f_r(\mathbf{y}_i, \overrightarrow{\mathbf{y}_{i+1}\mathbf{y}}, \overrightarrow{\mathbf{y}_i\mathbf{y}_{i-1}}) |\mathbf{n}_{\mathbf{y}_i} \cdot \overrightarrow{\mathbf{y}_i\mathbf{y}_{i+1}}|}{p(\overrightarrow{\mathbf{y}_{i+1}\mathbf{y}_i})} \right) \tag{19}$$

The first part is the emitted light $L_e(\mathbf{y}, \overrightarrow{\mathbf{y}_0\mathbf{y}_1})$ from light source at point $\mathbf{y}_0$. The PDF $p(\mathbf{y}, \overrightarrow{\mathbf{y}_0\mathbf{y}_1})$ is probability of selecting a ray in the direction $\overrightarrow{\mathbf{y}_0\mathbf{y}_1}$. Considering a light that has an uniform distribution of the emitted light, then PDF is:

$$p(\mathbf{y}, \overrightarrow{\mathbf{y}_0\mathbf{y}_1}) = \frac{1}{2\pi} \cdot \frac{1}{A_{\text{light}}} \tag{20}$$

where $\frac{1}{A_{\text{light}}}$ arises from uniformly choosing a point on the emitter, and $\frac{1}{2\pi}$ originates from uniformly selecting a direction over the hemisphere above the point $\mathbf{y}_0$. $W_e(\mathbf{x}, \overrightarrow{\mathbf{y}_s\mathbf{x}})$ 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 $\mathbf{x}$. Furthermore, when using a pinhole camera, $p(\mathbf{x})$ is also equal to 1, as only $\mathbf{x}$ can be chosen as a point.

The next part $f_r(\mathbf{y}_s, \overrightarrow{\mathbf{y}_{s-1}\mathbf{y}_s}, \overrightarrow{\mathbf{y}_s\mathbf{y}_{s+1}}) G(\mathbf{y}_s, \mathbf{x})$ comes from a direct ray from $\mathbf{y}_s$ to the camera $\mathbf{x}$. 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

$$\langle L_{\mathcal{P}} \rangle = \sum_{s=0}^{N_L} \sum_{t=0}^{N_E} w_{s,t} \cdot C_{s,t} \tag{21}$$

where $C_{s,t}$ is the unweighted contribution of a path with s vertices on the light path and t vertices on the camera path. A visibility test between s^{th} vertex on the light path and t^{th} 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 $w_{s,t}$.

The evaluation of $C_{s,t}$ depends on the camera and light path.

$$C_{s,t} = \begin{cases} C_{0,0} & \text{if } s = 0, t = 0 \\ C_{s,0} & \text{if } s > 0, t = 0 \\ C_{0,t} & \text{if } s = 0, t > 0 \\ C & \text{Otherwise.} \end{cases} \tag{22}$$

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

- **Case 01** ($s = 0, t = 0$): light is directly visible from the camera. The evaluation can be done with direct path between the point $\mathbf{x}$ and $\mathbf{y}_0$.

$$C_{0,0} = L_e(\mathbf{y}_0, \overrightarrow{\mathbf{y}_0\mathbf{x}}) G(\mathbf{y}_0, \mathbf{x}).$$

- **Case 02** ($s > 0, t = 0$): this means we have s 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).

$$C_{s,0} = C_s$$

- **Case 03** ($s = 0, t > 0$): this means we have zero vertices on light path and t vertices on camera path, which represents the classic path tracing algorithm. The contribution can be evaluated using Equation (17).

$$C_{0,t} = C_t$$

- **Case 04** ($s > 0, t > 0$): the final case is the evaluation of radiance, from t camera vertices connected with s light vertices of the light path, reaching the pixel.

$$C = \frac{L_e(\mathbf{y}_0, \overrightarrow{\mathbf{y}_0\mathbf{y}_1})}{p(\mathbf{y}_0, \overrightarrow{\mathbf{y}_0\mathbf{y}_1})} G(\mathbf{y}_s, \mathbf{x}_t) \\ f_r(\mathbf{x}_t, \overrightarrow{\mathbf{y}_s\mathbf{x}_t}, \overrightarrow{\mathbf{x}_t\mathbf{x}_{t-1}}) f_r(\mathbf{y}_s, \overrightarrow{\mathbf{y}_s\mathbf{y}_{s+1}}, \overrightarrow{\mathbf{y}_s\mathbf{x}_t}) \\ \left(\prod_{i=1}^{t-1} \frac{f_r(\mathbf{x}_i, \overrightarrow{\mathbf{x}_{i+1}\mathbf{x}_i}, \overrightarrow{\mathbf{x}_i\mathbf{x}_{i-1}}) |\mathbf{n}_{\mathbf{x}_i} \cdot \overrightarrow{\mathbf{x}_i\mathbf{x}_{i+1}}|}{p(\overrightarrow{\mathbf{x}_{i+1}\mathbf{x}_i})} \right) \\ \left(\prod_{i=1}^{s-1} \frac{f_r(\mathbf{y}_i, \overrightarrow{\mathbf{y}_{i+1}\mathbf{y}}, \overrightarrow{\mathbf{y}_i\mathbf{y}_{i-1}}) |\mathbf{n}_{\mathbf{y}_i} \cdot \overrightarrow{\mathbf{y}_i\mathbf{y}_{i+1}}|}{p(\overrightarrow{\mathbf{y}_{i+1}\mathbf{y}_i})} \right) \tag{23}$$

3.8 Weight Function

Many ways to create a weight function $w_{s,t}$ 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 $s + t$ vertices can be sampled in $s + t - 1$ different ways. Each path should have such weight, that together all weights sum to 1. In Veach's work [7] it is described, that the most effective way to use MIS, in order to create a weight function, is with the power heuristic:

$$w_{s,t} = \frac{p_s^\beta}{\sum_{i=0}^{s+t-1} p_i^\beta} \tag{24}$$

where β is recommended to be 2, according to Veach[7]. Taking $\beta = 2$, Equation (24) reduces to:

$$w_{s,t} = \frac{p_s^2}{\sum_{i=0}^{s+t-1} p_i^2} = \frac{1}{\sum_{i=0}^{s+t-1}(p_i/p_s)^2} \tag{25}$$

where p_i is defined as the probability density for generating path $\mathcal{P}_{s,t}$ using i sub light path vertices and $s + t - i$ camera sub path vertices.

$$p_i = p_{i,s+t-i}(\mathcal{P}_{s,t}) \tag{26}$$

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

$$\frac{p_{i+1}}{p_i} = \frac{\overleftarrow{p_i}(x)}{\overrightarrow{p_i}(x)} \tag{27}$$

where $\overleftarrow{p_i}(x)$ is the probability density of the same path being generated from the reversed direction, thus called *reversed PDF*, and $\overrightarrow{p_i}(x)$ 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.

$$\begin{aligned} w_{s,t} &= \frac{1}{\sum_{i=0}^{s-1}(p_{i,s+t-i}/p_{s,t})^2 + 1 + \sum_{i=s+1}^{s+t}(p_{i,s+t-1}/p_{s,t})^2} \\ &= \frac{1}{w_{\text{light},s-1} + 1 + w_{\text{camera},t-1}} \end{aligned} \tag{28}$$

where

$$\begin{aligned} w_{\text{camera},i} &= \frac{\overleftarrow{p_i}(x)}{\overrightarrow{p_i}(x)}(w_{\text{camera},i-1} + 1) \\ w_{\text{light},i} &= \frac{\overleftarrow{p_i}(x)}{\overrightarrow{p_i}(x)}(w_{\text{light},i-1} + 1) \end{aligned} \tag{29}$$

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/bdpt.cpp`.

```
// integrator.h
class BDPTIntegrator : public Integrator {
  public:
    ⟨Constructors⟩
    ⟨Render Function⟩

  private:
    ⟨BDPT Helper Functions⟩
    ⟨BDPT 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
⟨Render 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 `Vertex`s.

```
// integrator.h
⟨BDPT 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
⟨BDPT 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
⟨BDPT 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:

```
// bdpt.cpp
vector<Vertex> BDPTIntegrator::traceCameraPath
    (const Scene &scene, Sampler &sampler,
     const Ray3f &ray) const {
    vector<Vertex> path;
    Ray3f currentRay = ray;
    Color3f throughput(1.0f);
    Color3f result(0.0f);

    float pdfFwd = 1.0f;
    float pdfRev = 1.0f;

    for(int bounce = 0; bounce < m_maxBounces; ++bounce)
     {
        ⟨Scene Intersection Test⟩
        ⟨Sample Surface⟩
        ⟨Modify Path Throughput⟩
        ⟨Russian Roulette⟩
    }
};
```

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:

```
// bdpt.cpp
vector<Vertex> BDPTIntegrator::traceLightPath
    (const Scene &scene, Sampler &sampler) const {
    vector<Vertex> path;
    Ray3f currentRay = ray;
    Color3f throughput(1.0f);
    Color3f result(0.0f);

    float pdfFwd = 1.0f;
    float pdfRev = 1.0f;

    ⟨Initialize the First Vertex in Light Path⟩
    for(int bounce = 0; bounce < m_maxBounces; ++bounce)
     {
        ⟨Scene Intersection Test⟩
        ⟨Sample Surface⟩
        ⟨Modify Path Throughput⟩
        ⟨Russian Roulette⟩
    }
};
```

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.

```
// bdpt.cpp
⟨Initialize the First Vertex in Light Path⟩+ ≡
    float lightPDF;
    Vec2f lightSample = sampler.next2D();
    Vec3f lightPos = scene.sampleEmitters(lightSample,
     lightPDF);
    Vec3f tmp = randomOnUnitHemisphere(sampler.next2D());
    Vec3f lightDir(tmp.x, -tmp.z, tmp.y);
    float hemispherePDF = 1.0f / (2.0f * M_PI);

    HitInfo initHit;
    Ray3f currentRay(lightPos, lightDir);
    bool lightHit = scene.intersect(currentRay, initHit);

     if(!lightHit)
        // not hiting, break directly
        return path;

    Ray3f revertRay(initHit.p, -lightDir,
                    Epsilon, INFINITY);
    lightHit = scene.intersect(revertRay, initHit);

    // should return true but do this for safety
    if(!lightHit) return path;

    Vertex initVertex;
    initVertex.hit = initHit;
    initVertex.hit.sn = Vec3f(0.f,-1.f,0.f);
    initVertex.pdfFwd = hemispherePDF * lightPDF;
    initVertex.pdfRev = 1.0f;
    initVertex.wi = Vec3f(0.f);
    initVertex.wo = lightDir;
    initVertex.throughput = throughput;
    initVertex.nextThroughput = throughput;
    Color3f initEmit =
        initHit.mat->emitted(revertRay, initHit);
    initVertex.emitted = initEmit;
    result = initEmit;

    path.push_back(initVertex);
```

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 for (s, t)-paths: specifically, $s - 1$ vertices on the camera path and $t - 1$ 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.

```
// bdpt.cpp
Color3f BDPTIntegrator::Li
    (const Scene &scene, Sampler &sampler,
     const Ray3f &ray) const {
```

```
    Color3f L(0.f);

    vector<Vertex> lightPath =
        traceLightPath(scene, sampler);
    vector<Vertex> cameraPath =
        traceCameraPath(scene, sampler, ray);

    for(int s = 1; s <= cameraPath.size(); ++s) {
        for(int t = 1; t <= lightPath.size(); ++t) {
            int depth = t+s-2;
            if((s==1 && t==1) || depth < 0
             || depth > m_maxBounces)
                continue;

            Color3f contribution =
                connect(scene, cameraPath, lightPath,
                        s-1, t-1);
            float misWeight =
                calculateMIS(cameraPath,lightPath,s,t);

            L += contribution * misWeight;
        }
    }

    return L;
}
```

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

```
// bdpt.cpp
Color3f BDPTIntegrator::connect
    (const Scene &scene,
     const vector<Vertex> &camPath,
     const vector<Vertex> &lightPath,
     size_t s, size_t t) const {

    Vertex cameraVertex = camPath[s];
    Vertex lightVertex = lightPath[t];
    Vertex initLight = lightPath[0];
    Vec3f connectDir = lightVertex.hit.p - cameraVertex.
     hit.p;
    Ray3f ray(cameraVertex.hit.p, connectDir);
    HitInfo hit;

     ⟨Visibility Test⟩

     ⟨Special Case Handling for (s,t) = (0,0)⟩

    Color3f fCamera = cameraVertex.emitted;
    Color3f fLight = lightVertex.emitted;

     ⟨Special Case Handling for (s,t) = (s,0)⟩
     ⟨General Case Handling⟩
}
```

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

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

$$\mathbf{r}(t) = \mathbf{x} + t\overrightarrow{\mathbf{xy}}$$

The necessity and sufficiency is that the ray should only intersect with the scene, at $t = 1$, since $\mathbf{x} + 1 \cdot \overrightarrow{\mathbf{xy}} = \mathbf{y}$. If a intersection is found at $t < 1$, we know that the ray sees something before seeing $\mathbf{y}$, which indicates $\mathbf{y}$is occluded by that object viewing from $\mathbf{x}$.

```
// bdpt.cpp
⟨Visibility 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
⟨BDPT 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 Cases. The special cases follow the exactly process as what we have discussed in Section 3.7.

```
// bdpt.cpp
⟨Special 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;
    }

⟨Special 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);
        Color3f load = cameraVertex.throughput;
        return Le * bsdf * g * load;
    }
```

4.4.6 *General Case.*

And finally the general case. Still, following what we have discussed in Section 3.7, we will be able to implement this part as well.

```
// bdpt.cpp
⟨General Case Handling⟩+ ≡
    Color3f bsdf;
    if(cameraVertex.hit.mat->isEmissive())
        return fCamera;

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

    Color3f bsdf2 = lightVertex.hit.mat->
        eval(lightVertex.wo,
             normalize(connectDir),
             lightVertex.hit);

    Color3f result = lightVertex.throughput
                   * cameraVertex.throughput;
    result *= bsdf * bsdf2;
    result *= initLight.emitted
            / initLight.pdfFwd;
    result *= geometryTerm(cameraVertex.hit.sn,
                           cameraVertex.hit.p,
                           lightVertex.hit.sn,
                           lightVertex.hit.p);

    return result;
```

4.5 Multiple Importance Sampling

Obviously, the current `Li()` 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.

```
// bdpt.cpp
⟨BDPT Helper Function⟩+ ≡
    float BDPTIntegrator::calculateMIS
        (const vector<Vertex> &camPath,
         const vector<Vertex> &lightPath,
         size_t s, size_t t) const {

        if (s == 1 && t == 1)
            return 1.f;

        if (s > 1 && t == 1) {
            float wCamera = 1.0f;
            for (int i = 1; i <= s; i++) {
                const Vertex& vertex = camPath[i];
                wCamera =
                    (remap0(vertex.pdfRev)
                   / remap0(vertex.pdfFwd))
                   * (wCamera + 1.0f);
            }
            return 1.0f / (wCamera + 1.0f);
        }

        if (s == 1 && t > 1) {
            float wLight = 1.0f;
            for (int i = 1; i <= t; ++i) {
                const Vertex& vertex = lightPath[i];
                wLight = (remap0(vertex.pdfRev)
                        / remap0(vertex.pdfFwd))
                        * (wLight + 1.0f);
            }
            return 1.0f / (1.0f + wLight);
        }

        float sum_wlight = 1.f;
        float sum_wcamera = 1.f;

        for(int i = 1; i <=s; i++) {
            sum_wcamera = (remap0(camPath[i].pdfRev)
                        / remap0(camPath[i].pdfFwd))
                        * (sum_wcamera + 1.0f);
        }

        for(int i = 1; i <= t; i++) {
            sum_wlight =
                (remap0(lightPath[i].pdfRev)
               / remap0(lightPath[i].pdfFwd))
               * (sum_wlight + 1.0f);
        }

        return
            1.f / (sum_wlight + 1 + sum_wcamera);

    }
```

5 BIDIRECTIONAL METHOD: RESULTS

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

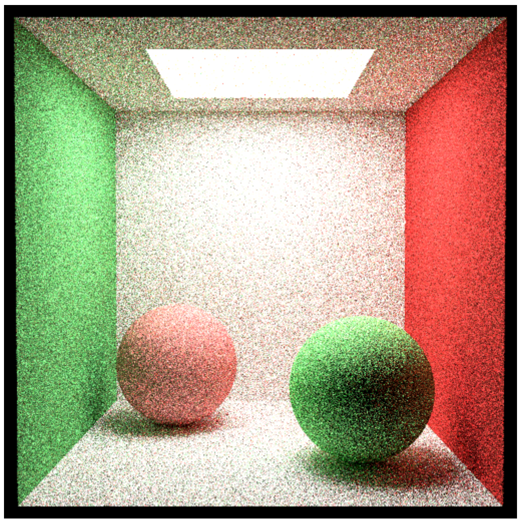

Figure 6: Arithmatically Averaged BDPT Cornell Box Scene (128 spp / 64 max depth) using `BDPTIntegrator`.

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 with an acceptable amount of effort in future developments. Despite these limitations, the current bidirectional path tracer already demonstrates some advantages over the traditional stochastic path tracer.

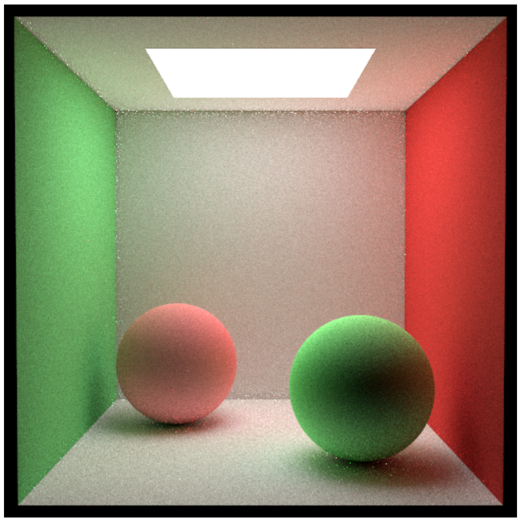

Figure 7: Weighted BDPT Cornell Box Scene (128 spp / 64 max depth) using `BDPTIntegrator`.

Notice the white noises and incorrect shadow positions compared to the below referece image generated by `PathTracerMIS`.

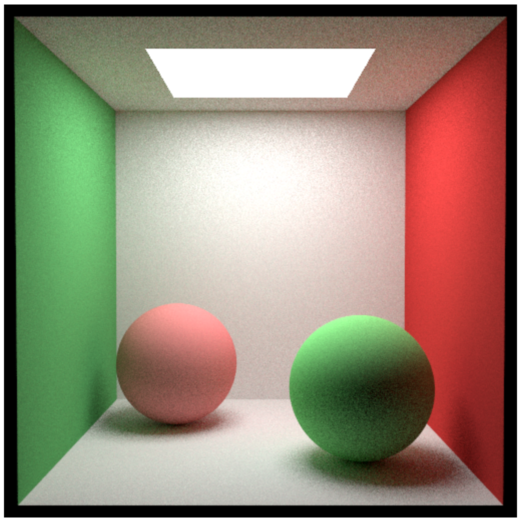

Figure 8: Reference Cornell Box Scene (128 spp / 64 max depth) using `PathTracerMIS`.

6 CONCLUSION

In conclusion, the implementation of *Bidirectional Path Tracing* (BDPT) with *Multiple Importance Sampling* (MIS) and the inclusion of delta light sources, such as point lights and spotlights, have significantly enhanced the rendering capabilities of the system. BDPT+MIS allows for efficient exploration of the path space by combining the strengths of both camera and light paths, reducing variance and improving convergence rates. The incorporation of delta light sources enables the accurate simulation of common lighting scenarios found in real-world environments. Point lights provide a simple yet effective way to represent omnidirectional light sources, while spotlights offer directional control and the ability to create focused illumination. The successful integration of these techniques has resulted in a robust and versatile rendering framework capable of producing high-quality images with realistic lighting effects. The implementation of BDPT+MIS and delta light sources demonstrates the potential for further advancements in physically-based rendering and opens up new possibilities for creating visually compelling and accurate simulations of complex lighting scenarios.

A RENDERING COMPETITION

For the rendering competition, I rendered a breakfast scene from the author Wig42. The scene objects and (`.json`) file can be found in `report/final/Scenes and Jsons`. I have rendered using the Path Tracer with MIS as well as the Bidirectional Path Tracer with MIS, to show that quality and potential incorrectness in my current implementation.

The following picture is rendered using `BDPTIntegrator` with MIS.

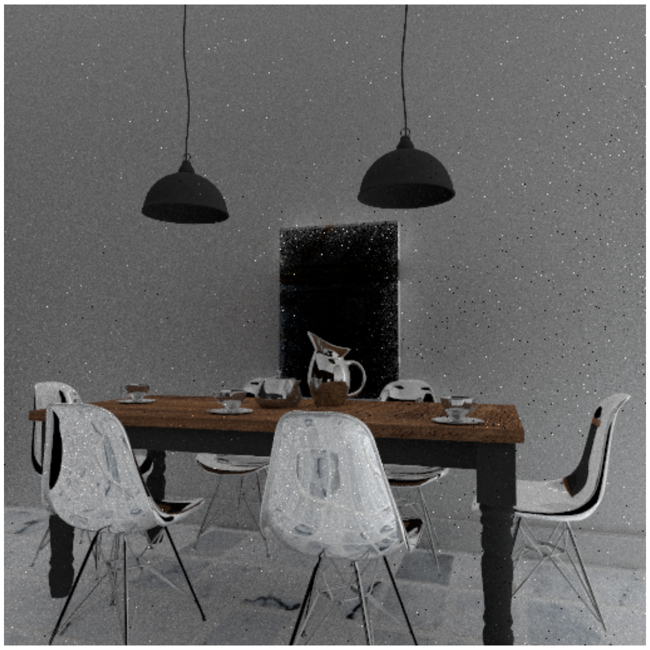

Figure 9: Breakfast Scene (1024 spp / 256 max depth) using `BDPTIntegrator`.

Since the light sampling and PDF is not yet supported by the current implementation of BDPT, the spotlight in the scene has been removed from the scene. The following picture is rendered using `PathTracerMIS`, with the spotlight above the scene. Notice they are under the same environmental lighting, but the BDPT still produces a darker rendering.

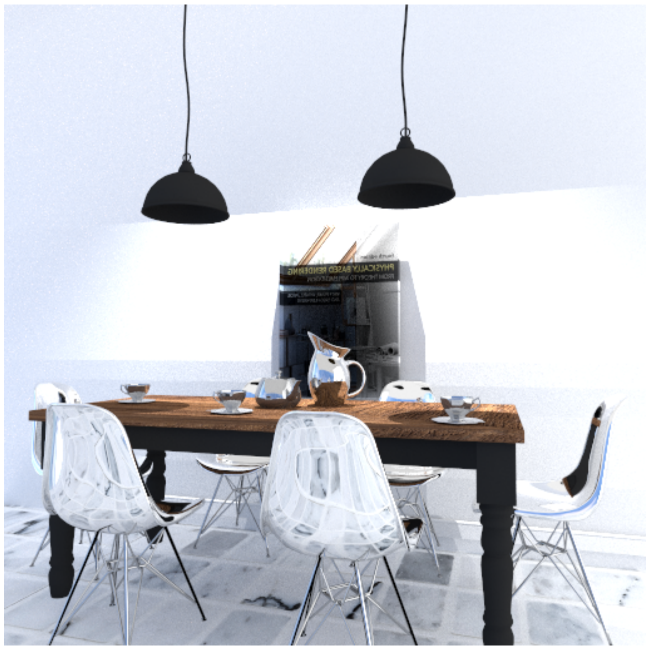

Figure 10: Breakfast Scene (1024 spp / 256 max depth) using `PathTracerMIS`.

ACKNOWLEDGMENTS

Special thanks to Professor Ioannis Gkioulekas, for proffering the fantastic Physically Based Rendering (15-468/668/868) course at Carnegie Mellon University, and Teaching Assistant Jeff Tan and Alan Lee, for the help throughout the final project and the entire semester.

REFERENCES

[1] Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. 2006. *Advanced Global Illumination.* AK Peters Ltd.

[2] Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidovič, and Philipp Slusallek. 2012. Light transport simulation with vertex connection and merging. *ACM Trans. Graph.* 31, 6, Article 192 (nov 2012), 10 pages. https://doi.org/10.1145/2366145.2366211

[3] James T. Kajiya. 1986. The rendering equation. *SIGGRAPH Comput. Graph.* 20, 4 (aug 1986), 143–150. https://doi.org/10.1145/15886.15902

[4] Eric Lafortune and Yves Willems. 1998. Bi-Directional Path Tracing. *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics' 93* (01 1998).

[5] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[6] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. 1996. Monte Carlo techniques for direct lighting calculations. *ACM Trans. Graph.* 15, 1 (jan 1996), 1–36. https://doi.org/10.1145/226150.226151

[7] Eric Veach. 1998. *Robust monte carlo methods for light transport simulation.* Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) Guibas, Leonidas J. AAI9837162.

Received 30 April 2024; revised 30 April 2024; accepted 30 April 2024