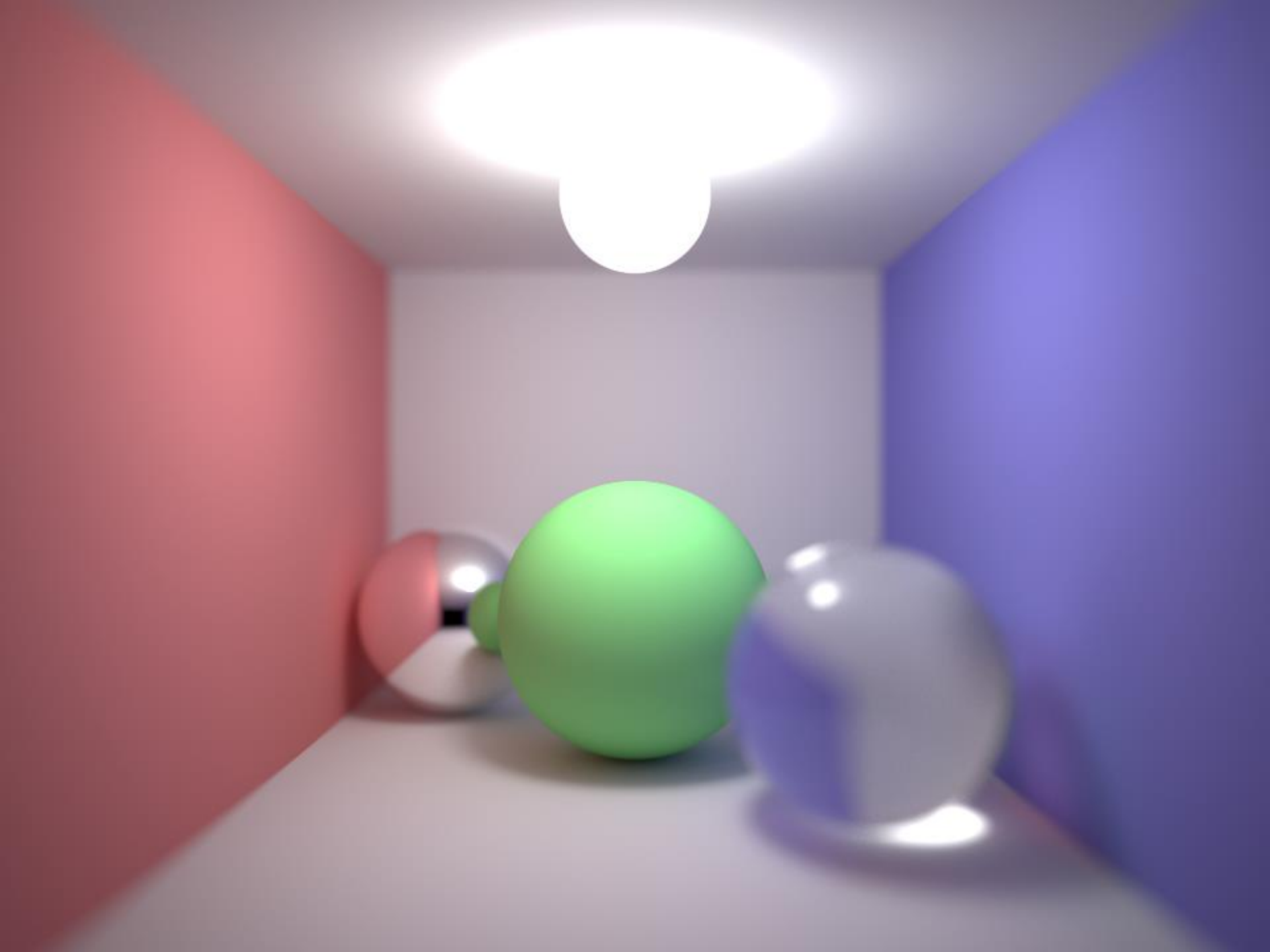
双方向パストレーシングレンダラ edubpt解説

2013/11/19 Ver.1.00

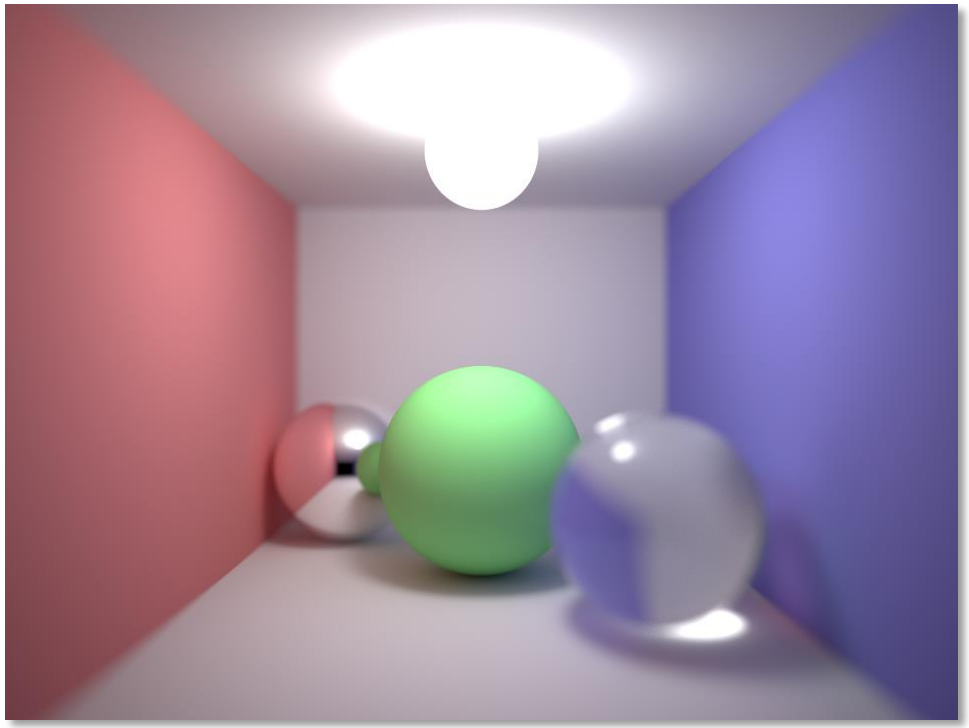
hole(@h013)

http://kagamin.net/hole/edubpt/index.htm



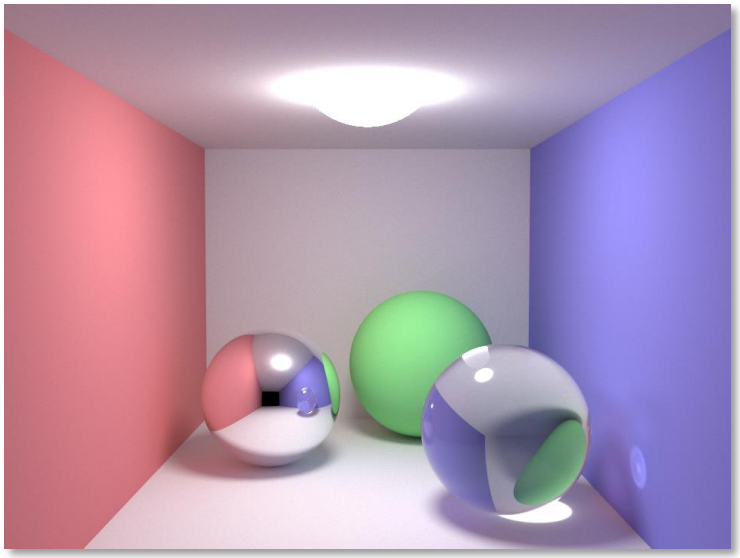
edubptとは

シンプルでコンパクトな
双方向パストレーシングレンダラ



edubptとは

物理ベース・パストレーシングレンダラ
eduptの続編



http://kagamin.net/hole/edupt/index.htm
https://github.com/githole/edupt

edubptとは

主な特徴

edupt をベースに双方向化

C++で記述

日本語によるコメント付き

教育的

edubptとは

Githubでコード公開

https://github.com/githole/edubpt

edubptとは

細かい仕様

双方向パストレーシングによる物理ベースレンダラ
物理カメラモデルの採用による被写界深度の再現
リファレンス用のパストレーシングも内蔵（eduptとほぼ同じ）
パストレーシング・ライトトレーシングも内蔵（双方向パストレ用）
OpenMPによる並列化
標準ライブラリ以外の依存ライブラリは（基本的に）無し

このスライドの内容

edubptのコード解説
＋
双方向パストレーシングの解説
＋
その他、物理ベースレンダリングの**発展的解説**

※前回の「物理ベースレンダラedupt解説」の内容をある程度理解している前提の内容
※割と数学の話が多め

目次

1. **レンダリングに関する基礎知識**
    1. ロシアンルーレット
    2. レンダリングのための確率論
    3. 確率密度の変換
    4. ジオメトリファクタ
    5. マルチプルインポータンスサンプリング
2. **レンダリングの定式化**
3. **パストレーシング**
4. **双方向パストレーシング・理論編**
    1. 双方向パストレが解く式
    2. レンダリング方程式
5. **双方向パストレーシング・実装編その１**
    1. パストレーシングによるパスのサンプリング
    2. ライトトレーシングによるパスのサンプリング
6. **双方向パストレーシング・実装編その２**
7. **レンダリング結果**

1. レンダリングに関する基礎知識

レンダリングに関する基礎知識

- レンダリングという問題を解いていく際に、必要となる各種の基礎的な理論・技法について説明していく。
  - この章で説明した事柄は、このスライド全体にわたって使う。

1.1 ロシアンルーレット

ロシアンルーレット

- 以下のような無限級数を計算したいとする。
  - （レンダリングは結局無限級数の計算に帰着される）

$$C = C_0 + C_1 + C_2 + \cdots$$

- 計算機は無限級数を扱うことが出来ない。
  - 無限ループ
- そこで、確率的に処理を打ち切ることで統計的な期待値は真値になるようにしつつ、現実的な計算量で推定値を得ることを考える。

$$C \approx \hat{C} = \text{推定値}$$

$$\text{ただし、} E\left[\hat{C}\right] = C$$

ロシアンルーレット

- 次のような項を考えると、

$$D_i = C_i + C_{i+1} + C_{i+2} + \cdots$$

- Cを以下のように書き換えることが出来る。

$$\begin{aligned} C &= D_0 \\ &= C_0 + D_1 \\ &= C_0 + C_1 + D_2 \\ &= \cdots \end{aligned}$$

ロシアンルーレット

- 以下のように処理を分岐させてCを推定する。
  - r_0は $[0,1)$の範囲の乱数
  - p_0は $[0,1]$の範囲の定数
  - $\hat{C}$はCの推定器で、確率変数

$$\hat{C} = \begin{cases} \dfrac{D_0}{p_0}, & r_0 < p_0 \\ 0, & r_0 \geq p_0 \end{cases}$$

- すると、 $E\left[\hat{C}\right] = D_0 = C$となる
  - 実際の実装では、乱数r_0を何らかの方法で生成し、 p_0未満ならD_0の評価を行い、p_0以上なら処理を打ち切って$\hat{C}$の評価値を0とする。
  - p_0は基本的に任意の値で良いが、 D_0の値の大小に比例させたほうが分散$V[\hat{C}]$は小さくなる。

ロシアンルーレット

- D_0の評価を行うことになったとする。
- D_0も$\hat{C}$と同じく無限級数なので、再び確率的に処理を分岐させて推定する。

$$\widehat{D_0} = C_0 + \begin{cases} \dfrac{D_1}{p_1}, & r_1 < p_1 \\ 0, & r_1 \geq p_1 \end{cases}$$

- さっきの式と合わせると、推定のための式全体は以下のようになる。
  - それぞれの条件分岐のたびに乱数生成を行い、処理を打ち切るか、次の項を評価するかを決める。
  - **以下、D_iの評価を行うたびに、それ以降の項D_{i+1}を評価するかどうかを確率的に決める。**
  - $E\left[\hat{\hat{C}}\right] = E\left[\widehat{D_0}\right] = C_0 + E[D_1] = C$となる。（以降も同じで、常に推定式全体の期待値は$C$ になる）
  - さっきと同様、p_iは基本的に任意の値で良いが、 D_iの値の大小に比例させたほうが分散は小さくなる。

$$\hat{\hat{C}} = \begin{cases} \dfrac{\boxed{\widehat{D_0}}}{p_0}, & r_0 < p_0 \\ 0, & r_0 \geq p_0 \end{cases} = \begin{cases} \dfrac{\boxed{C_0 + \begin{cases} \dfrac{D_1}{p_1}, & r_1 < p_1 \\ 0, & r_1 \geq p_1 \end{cases}}}{p_0}, & r_0 < p_0 \\ 0, & r_0 \geq p_0 \end{cases}$$

ロシアンルーレット

- まとめると、推定は以下のように行われる。

$$C \approx \hat{C} = \begin{cases} \dfrac{\widehat{D_0}}{p_0}, & r_0 < p_0 \\ 0, & r_0 \geq p_0 \end{cases}$$

$$\widehat{D_i} = C_i + \begin{cases} \dfrac{\widehat{D_{i+1}}}{p_{i+1}}, & r_{i+1} < p_{i+1} \\ 0, & r_{i+1} \geq p_{i+1} \end{cases}$$

ロシアンルーレット

- Cを推定するための疑似コード

```
1. Ĉ ← 0, i ← 0, p_t ← 1
2. while(true)
3.     p_t ← p_t · p_i
4.     r_i ← random([0,1))
5.     if r_i < p_i then
6.         Ĉ ← Ĉ + c_i / p_t
7.         i ← i + 1
8.     else break
9. end
10. return Ĉ
```

ロシアンルーレット

- **以上のように、確率的に処理を打ち切るかどうかを選択することで、トータルの計算量を現実的な量に抑えつつ、推定値$\hat{C}$の期待値は真値と等しくする（Unbiased）ことが出来る。**
  - 例えば、$p_i = \frac{1}{2}$とすると、100回連続で処理が行われる（打ち切られない）確率は$\left(\frac{1}{2}\right)^{100}$となる。確率的には、100兆回、100京回といった、現実的でない回数連続で処理が行われてしまう確率も0ではないが、極端に低い確率になるため、そのようなことは起こらないとしてよい。
- **レンダリングの文脈においては、シーンをレイトレーシングしてパス（光路）を追跡していく際、反射回数が無限に増えてしまうのを防ぐためにロシアンルーレットによって処理を打ち切る。**
  - また、フレネル反射において反射と屈折のどちらを追跡するかを決定するのもロシアンルーレット。

ロシアンルーレット

- 今回の例では全ての項の計算についてロシアンルーレットを行ったが、最初の数項についてはp_iを1に設定することで打ち切らないようにする、といったことを行うと最初の数項については、ロシアンルーレットによる分散が無くなり、トータルの分散を小さく抑えることも出来る。eduptでは最初の数回のトレースではロシアンルーレットを行っていない。
- ただし、edubptにおいては頂点ごとに完全に独立にp_iを設定したいので、このようなことは行っていない。

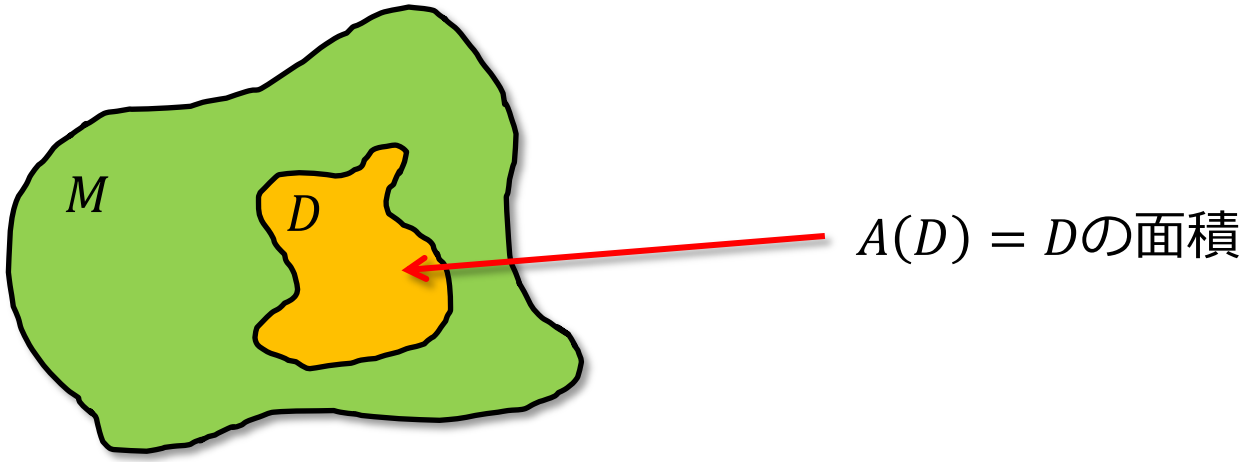
1.2 レンダリングのための確率論

レンダリングのための確率論

- **確率分布**（あるいは確率測度）について考える。
- 確率分布Pは以下のような測度関数として定義される。
- $P(D) = Probability\{X \in \mathrm{D}\}$（ここで$P$は$\Omega$上に定義される測度で、右辺は確率変数$X$が集合$D$に含まれる確率）
- 対応する確率密度関数pはΩ上の測度μを使って*Radon-Nikodym*微分を行うと$p(x) = \frac{dP}{d\mu}(x)$となり、これは$P(D) = \int_D p(x)d\mu(x)\ (= \int_D \frac{dP}{d\mu}(x)d\mu(x))$を満たすものである。
- **一般に、測度μに関する確率密度関数$p(x)$を$P_\mu(x)$と表記することにする。**

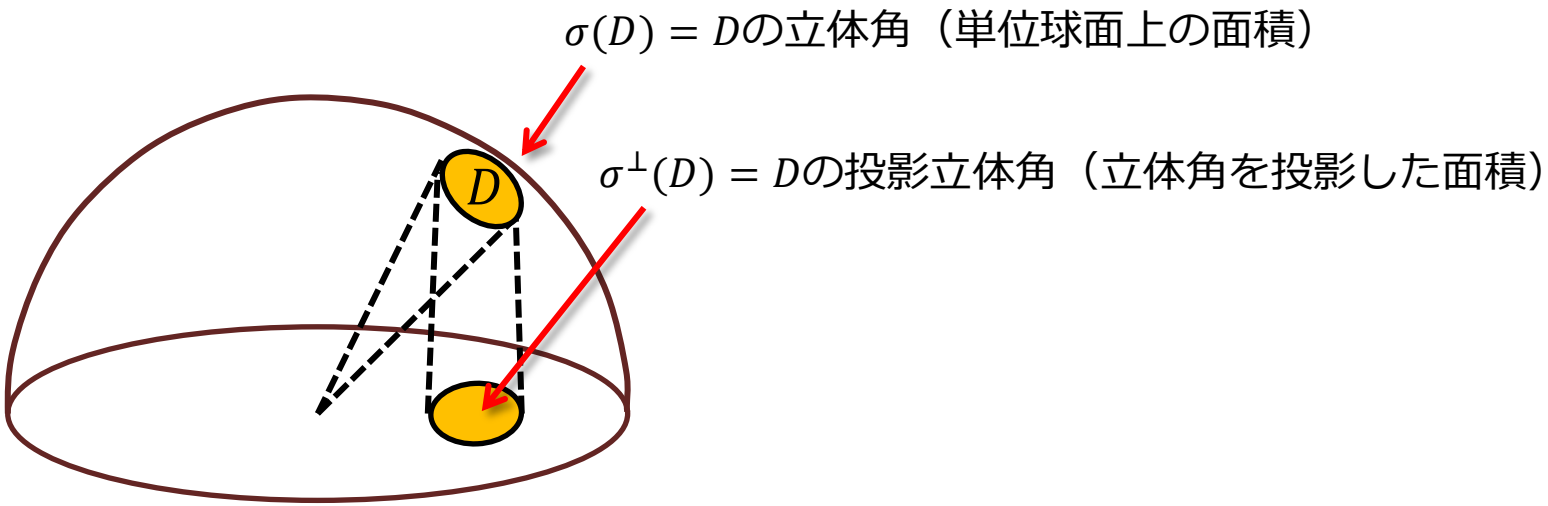
具体例

- シーンのジオメトリ表面全体をMとする。
- M上の確率分布Pを考える。
- M上の面積測度Aに関する確率密度関数を$P_A(x)$と書く。
  - 面積測度とは、ある領域を与えると（いわゆる普通の）面積を返す関数
  - $P_A(x)$はM上のある一点xについての面積測度に関する確率密度関数となる。
  - 今後もしばしば出てくる。
- 確率分布と測度に応じて様々な確率密度関数が考えられる。
  - 同じ面積測度Aを使っても、別の確率分布を与えれば、別の確率密度関数になる。
  - $P_A(x)$と$Q_A(x)$といったように、別の確率密度関数が得られる。
  - 同じ確率分布を使っても、別の測度を与えれば、別の確率密度関数になる。



具体例2

- 単位球面全体をS^2とする。
- S^2上の確率分布Pを考える。
- 立体角測度σに関する確率密度関数を$P_\sigma(\vec{\omega})$と書く。
  - 立体角測度とは、単位球面上のある領域を与えると、その領域の立体角を返す関数
  - $\sigma(D)$は立体角なので単位はステラジアン。
  - **$P_\sigma(\vec{\omega})$はS^2上のある一点についての確率密度を表すが、単位球面上の一点というのはすなわちある一方向のことなので、$P_\sigma(\vec{\omega})$はある一方向についての確率密度関数になる。**
- 確率密度関数を考えるとき、使う測度は（S^2上の適切な測度なら）なんでもいい。
  - 普通の立体角測度σや、投影立体角測度$\sigma^\perp$を使うことができる。
  - このスライドでは主に立体角測度を使うことにする。



具体例2

- 全球から一方向を一様にサンプリングするとき（よくあるシチュエーション）
- 一様にサンプリングということは確率密度が全方向で定数なので

$$P_\sigma(\vec{\omega}) = C$$

- 確率分布の定義より$P(S^2) = 1$となるため

$$\int_{S^2} P_\sigma(\vec{\omega})\, d\sigma(\vec{\omega}) = \int_{S^2} C\, d\sigma(\vec{\omega}) = C \int_{S^2} d\sigma(\vec{\omega}) = 4\pi C = 1$$

- **より、 $\boldsymbol{C = \frac{1}{4\pi}}$となり、 $\boldsymbol{P_\sigma(\vec{\omega}) = \frac{1}{4\pi}}$ となる（単位球面の面積の逆数になる）。**
  - ここで、微小立体角$d\sigma(\vec{\omega})$が球面上の微小領域$\sin\theta d\theta d\phi$と等しいことを利用して変数変換して上の式を解いた。

$$\int_{S^2} d\sigma(\vec{\omega}) = \int_0^{2\pi} \int_0^{\pi} \sin\theta\, d\theta d\phi$$

具体例2

- 全球から一方向を一様にサンプリングするとき（よくあるシチュエーション）
- **同じ分布について、別の測度で確率密度関数を定義してみる。**
- 投影立体角測度$\sigma^{\perp}$を使う。
  - 微小投影立体角$d\sigma^{\perp}$と微小立体角$d\sigma$の関係式、 $\cos\theta d\sigma = d\sigma^{\perp}$より

$$P_{\sigma}(\vec{\omega}) = \cos\theta\, P_{\sigma^{\perp}}(\vec{\omega})$$

- が得られる。$P_{\sigma}(\vec{\omega}) = \frac{1}{4\pi}$であったため上式に代入すると、

$$P_{\sigma^{\perp}}(\vec{\omega}) = \frac{1}{4\pi\cos\theta}$$

- となる。同じ一様分布だが、測度によって異なる確率密度が得られるということがわかる。

モンテカルロ積分

$$I = \int_{\Omega} f(x) d\mu(x)$$

- Ωを積分範囲、 μをΩ上に定義される測度とすると、 μに関する確率密度関数P_{μ}を使って以下のモンテカルロ積分$\hat{I}$によって上の積分を近似出来る
  - X_iは関数に対するサンプルで、 P_{μ}に従ってサンプリングされる。

$$I \approx \hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{P_{\mu}(X_i)}$$

証明

$$
\begin{aligned}
E\left[\hat{I}\right] = E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{P_\mu(X_i)}\right] &= \frac{1}{N}\sum_{i=1}^{N}\int_\Omega \frac{f(x)}{P_\mu(x)}P_\mu(x)d\mu(x) \\
&= \int_\Omega \frac{f(x)}{P_\mu(x)}P_\mu(x)d\mu(x) \\
&= \int_\Omega f(x)d\mu(x) \\
&= I
\end{aligned}
$$

注意

- $I = \int_\Omega f(x) d\mu(x)$という積分をモンテカルロ積分で解くためにはサンプルを測度μに関する確率密度関数P_μに基づいてサンプリングする必要がある
  - 別の測度μ'に関する確率密度関数$P_{\mu'}$に基づいてサンプリングした場合、得られる結果はあくまで$\int_\Omega f(x) d\mu'(x)$の近似になり、$\int_\Omega f(x) d\mu(x)$の近似にならない。

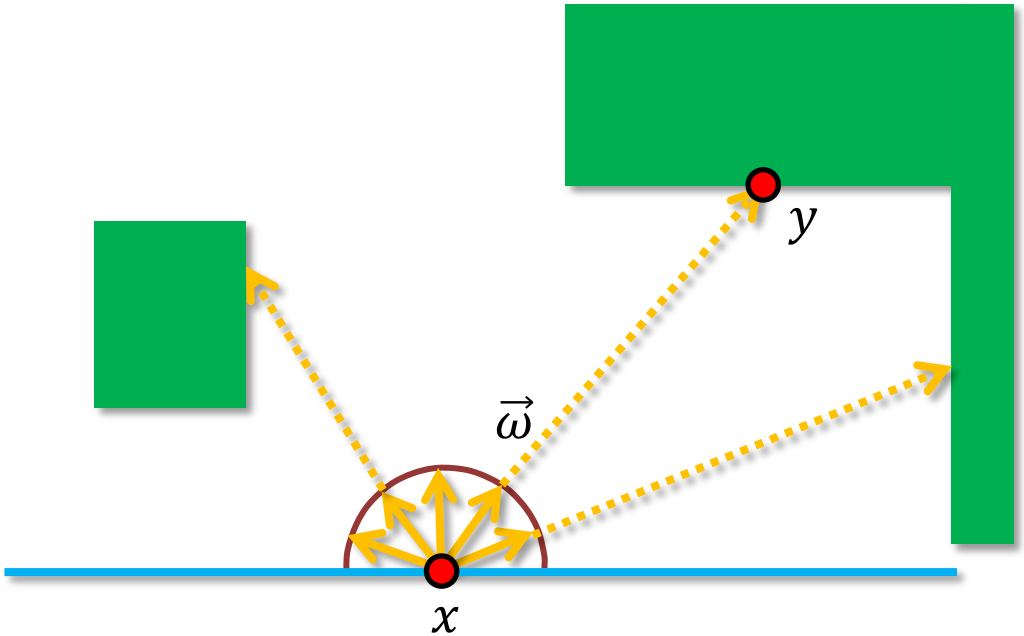
具体例

- 単位半球面Ω上に定義されたある関数$f(\vec{\omega})$を半球面上で微小立体角に関して積分したいとする。
- $\int_{\Omega} f(\vec{\omega}) d\sigma(\vec{\omega})$を解く。
- モンテカルロ積分を使うなら、何らかの確率密度関数P_{σ}に基づいてサンプルを生成する。
- このP_{σ}は、$\int_{\Omega} P_{\sigma}(\vec{\omega}) d\sigma(\vec{\omega}) = 1$を満たしさえすれば、任意の関数を与えることができる（$f(\vec{\omega})$の形状に近い方が誤差の収束が速い→Importance Sampling）
- たとえば、$\frac{\cos\theta}{\pi}$や$\frac{1}{2\pi}$などが、条件を満たす確率密度関数となる。

1.3 確率密度の変換

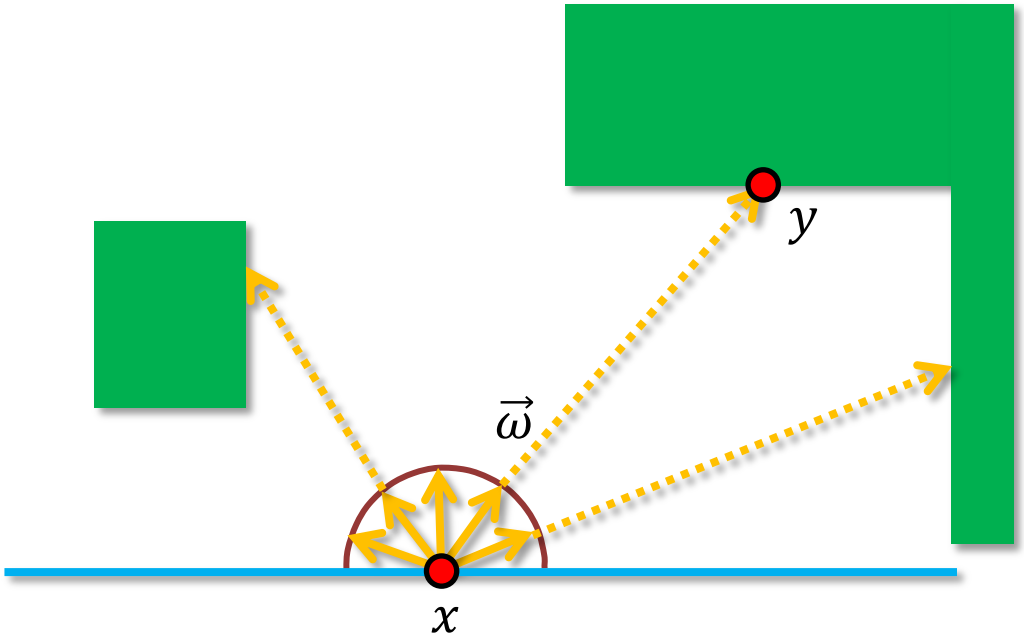
確率密度の変換

- **位置xから次の頂点をサンプリングするとする。**
- レイを飛ばす方向をサンプリングし、その方向にレイトレーシングして、シーンとの交点が次の頂点y
- 方向のサンプリングなので、立体角測度に関する確率密度関数に基づいてサンプリングを行う。
  - 例えば、$P_\omega(\vec{\omega}) = \frac{\cos\theta}{\pi}$のような確率密度関数に基づいてサンプリングすれば、$\cos\theta$項を考慮したインポータンスサンプリングということになる。（いつものやつ）



確率密度の変換

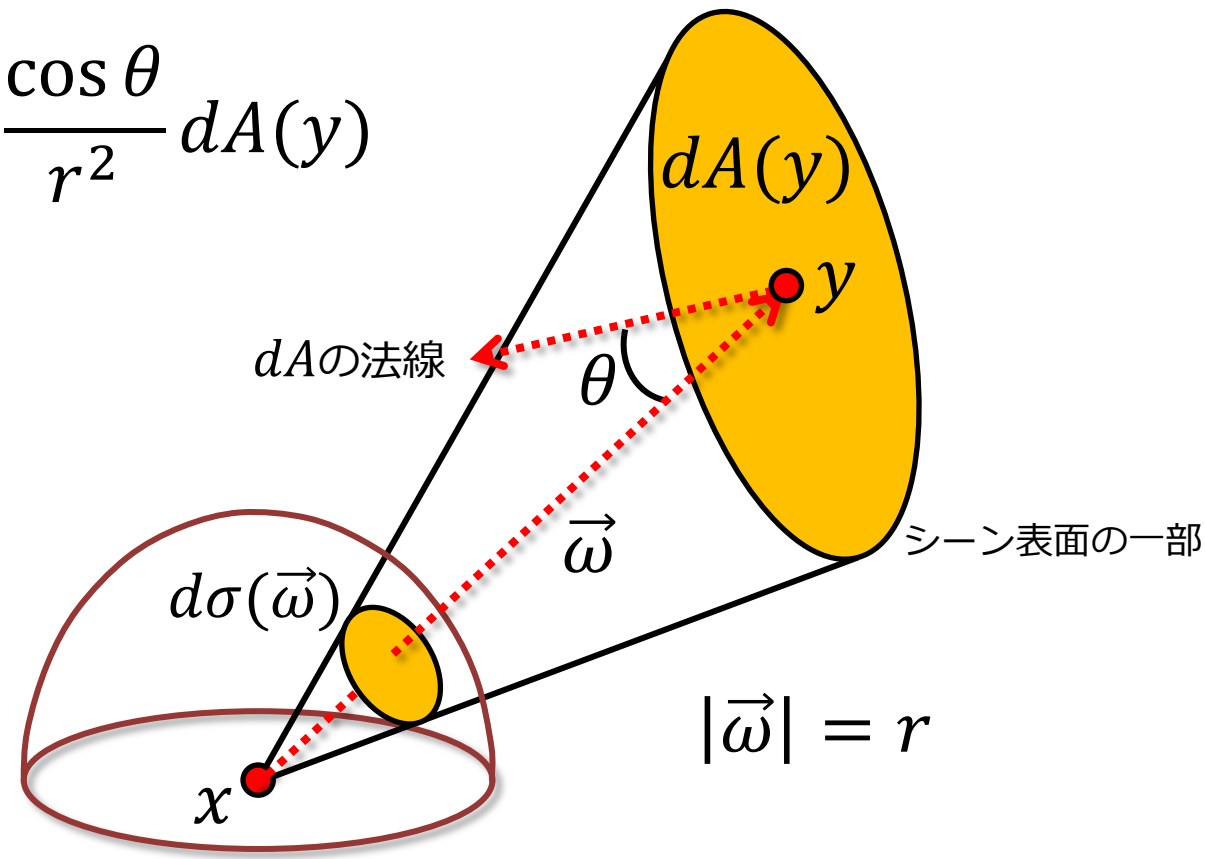
- 今回、最終的に解く積分の積分範囲がシーン表面になり、変数（＝頂点）はシーン表面を動くことになる。（後の章で説明。双方向パストレーシングをすっきり解くためにそういう式に変形する）
- そのような積分をモンテカルロ積分する際、各頂点は面積測度に関する確率密度関数でサンプリングされなければならないが、実際の実装では立体角測度に関する確率密度関数によってサンプリングされる。
  - 頂点を直接、面積測度に関する確率密度関数でサンプリングした場合、頂点間が別の物体でさえぎられて有効なサンプルにならない確率が非常に高くなるため、実際の実装ではレイトレに基づく立体角測度によるサンプリングを行ったほうが効率が良いため。
- よって、**立体角測度に関する確率密度**を**面積測度に関する確率密度**に変換する必要が出てくる。



確率密度の変換

- 位置xにおける$\vec{\omega}$方向の微小立体角と、対応するシーン内の微小面積との関係式を立てる。
  - 立体角に対応する面積とは、立体角をシーンに投影したときの、シーン内の領域
  - 図のような幾何的関係が成り立つので、以下の式が得られる。ここで、立体角は単位球面上の面積に等しいことを利用した。

$$d\sigma(\vec{\omega}) = \frac{\cos\theta}{r^2} dA(y)$$



確率密度の変換

- 微小立体角と微小面積の関係式より、立体角測度に関する確率密度と面積測度に関する確率密度の関係式が得られる。以下の式により、確率密度の測度を変換することができる。

$$\boxed{P_A(y) = \frac{\cos\theta}{r^2} P_\sigma(\vec{\omega})}$$

導出

$$\int_M P_A(y)dA(y) = 1, \int_\Omega P_\sigma(\vec{\omega})d\sigma(\vec{\omega}) = 1$$ （確率密度関数は全範囲で積分すると1）

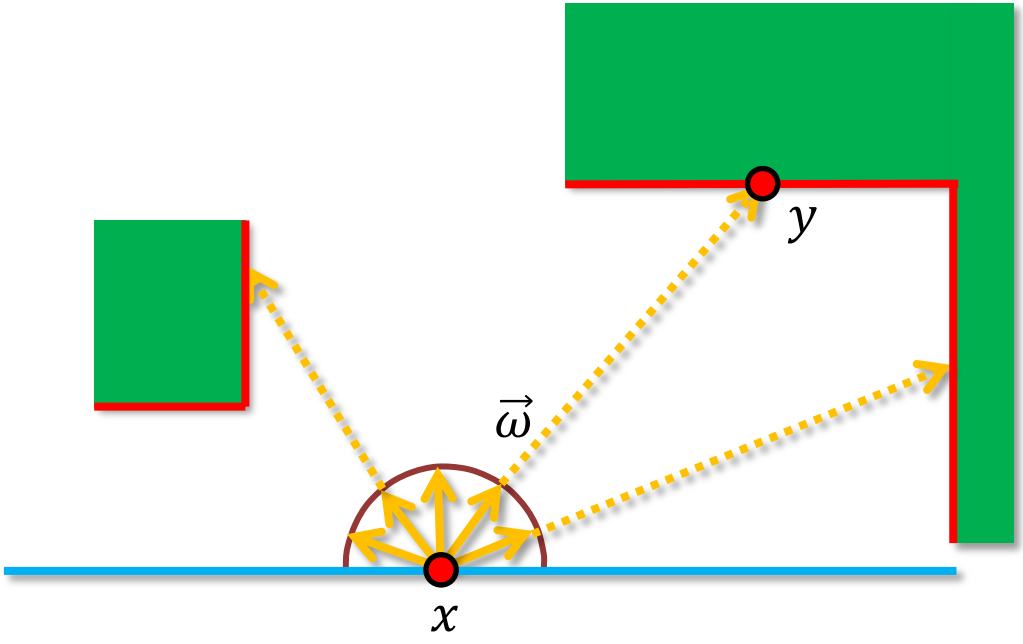
$$d\sigma(\vec{\omega}) = \frac{\cos\theta}{r^2} dA(y)$$

より

$$\int_\Omega P_\sigma(\vec{\omega})d\sigma(\vec{\omega}) = \int_M P_\sigma(\vec{\omega})\frac{\cos\theta}{r^2}dA(y) = \int_M P_A(y)dA(y)$$

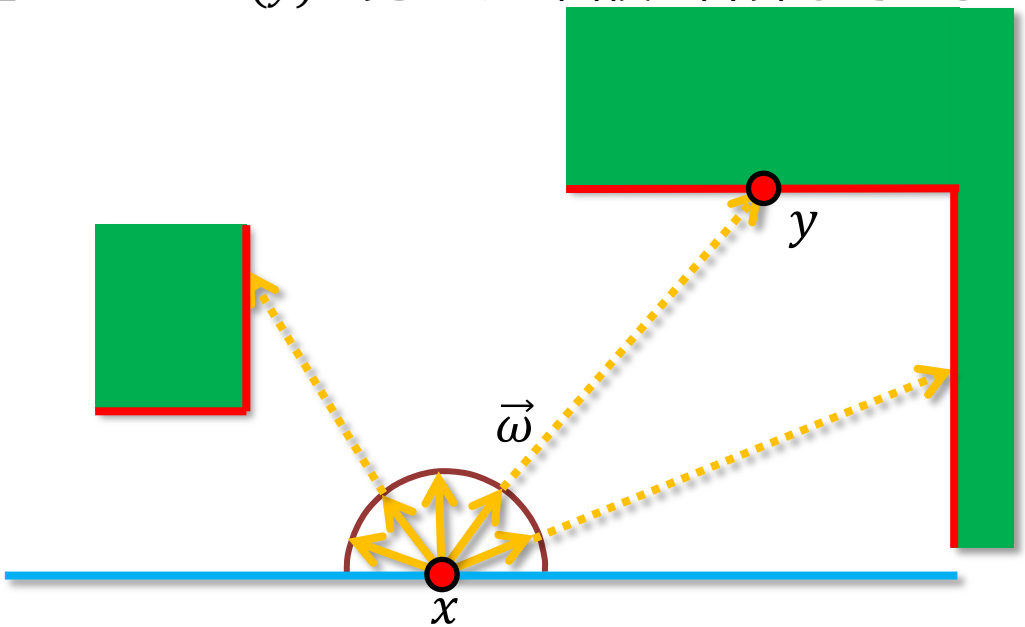
確率密度の変換（まとめ）

- 位置xから半球上に方向をサンプリングしてレイトレーシング、その交点をサンプリングするということは、位置xから見える全ての範囲（下図の赤い領域）の中から一点をサンプリングするということと同じ。
- 前者は立体角測度に関する確率密度に基づいてyをサンプリングしており、後者は面積測度に関する確率密度に基づいてyをサンプリングしている、ということになる。
- 両者を相互に変換するのが先の式。



確率密度の変換（まとめ）

- $\boldsymbol{P_A(y) = \frac{\cos\theta}{r^2} P_\sigma(\vec{\omega})}$
- 位置xから半球上に方向をサンプリングするとき、 xから近いシーン内の領域はrが小さく、その領域からyがサンプリングされる確率（= xから発射したレイがヒットする確率）も高くなる。逆に、 xから遠いシーン内の領域はrが大きく、その領域からyがサンプリングされる確率は低くなる。
- また、対象の領域が傾いていればいるほど、レイがヒットする確率は下がる。（＝$\cos\theta$が小さくなる）
- という関係を上の式は示している。
  - $\frac{\cos\theta}{r^2}$は、 位置xからの$dA(y)$の見かけの面積を計算している。



具体例

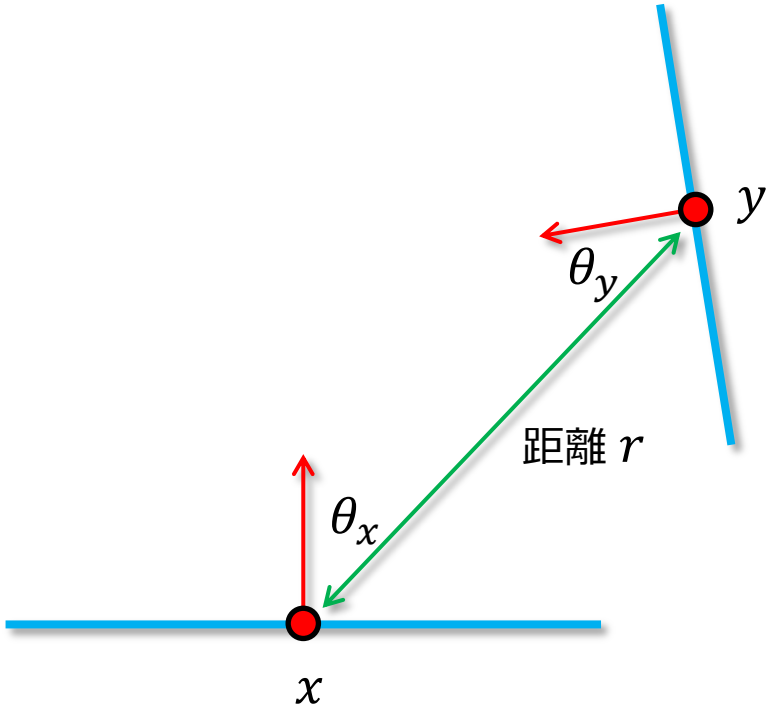
- $\int_{\Omega} f(\vec{\omega}) d\sigma(\vec{\omega})$を解く際、変数変換して積分範囲や積分変数を変換することを考える。
- シーン内における微小立体角と微小面積の関係式$d\sigma(\vec{\omega}) = d\sigma(x \to x') = \frac{\cos\theta}{r^2} dA(x')$より
- $\int_{\Omega} f(\vec{\omega}) d\sigma(\vec{\omega}) = \int_{\Omega} f(x \to x') d\sigma(x \to x') = \int_{\mathrm{M}} f(x \to x') \frac{\cos\theta}{r^2} dA(x')$
- ただしMはxから一方向で到達可能な全てのシーンジオメトリ表面になる。
- これをモンテカルロ積分するとなると、 M上の確率密度関数$\mathrm{P_A}$に基づいてサンプルを生成することになる
- 条件を満たしさえすれば任意の関数が与えられる
  - 面積測度に関する確率密度関数を使ったモンテカルロ積分に帰着することは多いが、実際のレンダリングの文脈ではシーンジオメトリ上の確率密度関数を与えて、それに基づいてサンプリングを行うことは少なく、結局立体角測に関する確率密度関数によってサンプリングを行ってから確率密度関数を面積測度関するものに変換することが多い。（例外：光源上のサンプリング、レンズ上のサンプリング）

1.4 ジオメトリファクタ

ジオメトリファクタ

- シーン内の二頂点間の幾何的関係を示す項として、ジオメトリファクタ（以下、Ｇ項）と呼ばれるものがある。
- 今後、頻繁に出てくるので注意。

$$G(x \leftrightarrow y) = \frac{\cos\theta_x \cos\theta_y}{r^2}$$



1.5 マルチプルインポータンスサンプリング

マルチプルインポータンスサンプリング

- ある積分をモンテカルロ積分で解くとする

$$I = \int_{\Omega} f(x) d\mu(x)$$

- 普通のモンテカルロ積分は以下のようになる
  - このとき、サンプルX_iは確率密度関数$p(x)$に従ってサンプリングされている

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

マルチプルインポータンスサンプリング

- **モンテカルロ積分の効率（収束の速さ）はサンプリングのための確率密度関数$p(x)$の形状によって決まる。**
  - 積分対象の関数$f(x)$に近ければ近いほど良い。
  - 極論、 $f(x)$に比例していれば$\frac{f(X_i)}{p(X_i)}$が定数になるため誤差は常に 0 になる。が、そもそも$f(x)$の形状が非常に複雑だからモンテカルロ積分を試みているため、これは不可能。
  - それでも$f(x)$についての知識を利用してうまい確率密度関数を考えることは出来る。→インポータンスサンプリング（Importance Sampling）
  - 例えば、拡散面の上においてレンダリング方程式に出現する$\cos\theta$に着目して$\cos\theta$に比例して次のレイの方向を決める、といったことが行われる。

$$\hat{I} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$$

マルチプルインポータンスサンプリング

- 複数の確率密度関数を用意して、それぞれのモンテカルロ積分の結果を組み合わせるという発想。
- 個々の確率密度関数は最適なものではないが、組み合わせれば良い結果が得られえる　→　**マルチプルインポータンスサンプリング（Multiple Importance Sampling）**
  - 例えば、 ある材質上で次のレイの方向をサンプリングするとき、**(1)材質のBRDFに基づいて確率密度関数を決めてサンプリング (2)光源の方向に基づいて確率密度関数を決めてサンプリング (3)何も考えずランダムにサンプリング ...** といったように、複数のサンプリング戦略が考えられる。このとき、唯一つの確率密度関数を上手に作成するのも良いが、そうではなく、別々に用意した確率密度関数を使って独立にサンプリングしてそれぞれのモンテカルロ積分の結果を組み合わせる、といった戦略もあり得る。ということ。
- 以下のように定式化される。
  - Sは異なるサンプリング戦略（確率密度関数）の数。
  - w_sは各サンプリング戦略による各サンプルごとの重み。

$$\hat{I} = \sum_{s=1}^{S} \left(\frac{1}{N_s} \sum_{i=1}^{N_s} w_s(X_{s,i}) \frac{f(X_{s,i})}{p_s(X_{s,i})} \right)$$

マルチプルインポータンスサンプリング

- **モンテカルロ積分がUnbiasedであるための重みw_sの条件その１**

$$w_s(x) \neq 0 \Longrightarrow p_s(x) \neq 0$$

あるいは $p_s(x) = 0 \Longrightarrow w_s(x) = 0$

- **モンテカルロ積分がUnbiasedであるための重みw_sの条件その２**

$$f(x) \neq 0 \Longrightarrow \sum_{s=1}^{S} w_s(x) = 1$$

以上は、任意のサンプルxについて、 $f(x)$が0でないならそのサンプルをサンプリングできる確率密度関数が少なくとも一つは存在しなければならず、さらにそのときの重みの和は1でなければならない、ということを意味する。

マルチプルインポータンスサンプリング

- **条件を満たす重みの例**
  - 全てのサンプリング戦略（確率密度関数）に対して等しい重みを与える。（ただし、任意のサンプルについて、全ての確率密度関数がサンプル出来なければならない）
  - あまり良くない。

$$w_s(x) = \frac{1}{S}$$

マルチプルインポータンスサンプリング

- **条件を満たす重みの例**
  - **バランスヒューリスティック**
  - 他の情報が無ければ最良の重み付け戦略となる。

$$\boxed{w_s(x) = \frac{p_s(x)}{\sum_{j=1}^{S} p_j(x)}}$$

- これは、あるサンプルxがサンプリング戦略sによってサンプリングされたときの重み。
  - xが**仮に**他のサンプリング戦略によってサンプリングされたとして、その確率密度の和に対する、 sによる確率密度の割合が上式。

具体例

- 二つのサンプリング戦略$p_1(x)$と$p_2(x)$があるとする。今、確率密度関数の範囲を$\mathrm{D} = \{X, Y\}$とすると、これらは以下の値をとるものとする。
  - $p_1(X) = 0.2, p_1(Y) = 0.8$
  - $p_2(X) = 0.6, p_2(Y) = 0.4$

- 今、 $p_1(x)$を使ってサンプリングした所、 Yが得られたとする。このときの重み$w_1(Y)$は
  - ここで、 Yが仮にp_2によってサンプリングされたとして、 $p_2(Y)$を使っていることに注意。

$$w_1(Y) = \frac{p_1(Y)}{p_1(Y) + p_2(Y)} = \frac{0.8}{0.8 + 0.4} = 0.666 \ldots$$

具体例

- 同様にして、全ての重みは以下のようになる。
  - これは先ほどの条件を満たしている。

$$w_1(X) = 0.25, w_1(Y) = 0.666 \ldots$$

$$w_2(X) = 0.75, w_2(Y) = 0.333 \ldots$$

- このことから、バランスヒューリスティックによる重み付けは、そのサンプルをサンプリングする確率密度が高い戦略に、より大きい重みが付けられるということが分かる。
- もし、一般のモンテカルロ積分のようにp_1のみ使ってサンプリングした場合、Xがあまりサンプリングされない。もし$f(X)$の値が大きい場合、これは大きな誤差・分散を生む。また、p_2のみの場合も逆のことが言える。そこで、二つの確率密度関数を使い、それぞれの得意なサンプルについての重みを大きくすることで、ノイズの大きくなるかもしれない結果の重みを小さくしつつ、ノイズが小さくなることが見込まれる結果の重みを大きくしている。これがバランスヒューリスティックの考え方。

他の重みの例

- **条件を満たす重みの例**
  - **パワーヒューリスティック**
  - バランスヒューリスティックの一般化
  - edubptではβ=2として使用。

$$\boxed{w_s(x) = \frac{p_s(x)^\beta}{\sum_{j=1}^{S} p_j(x)^\beta}}$$

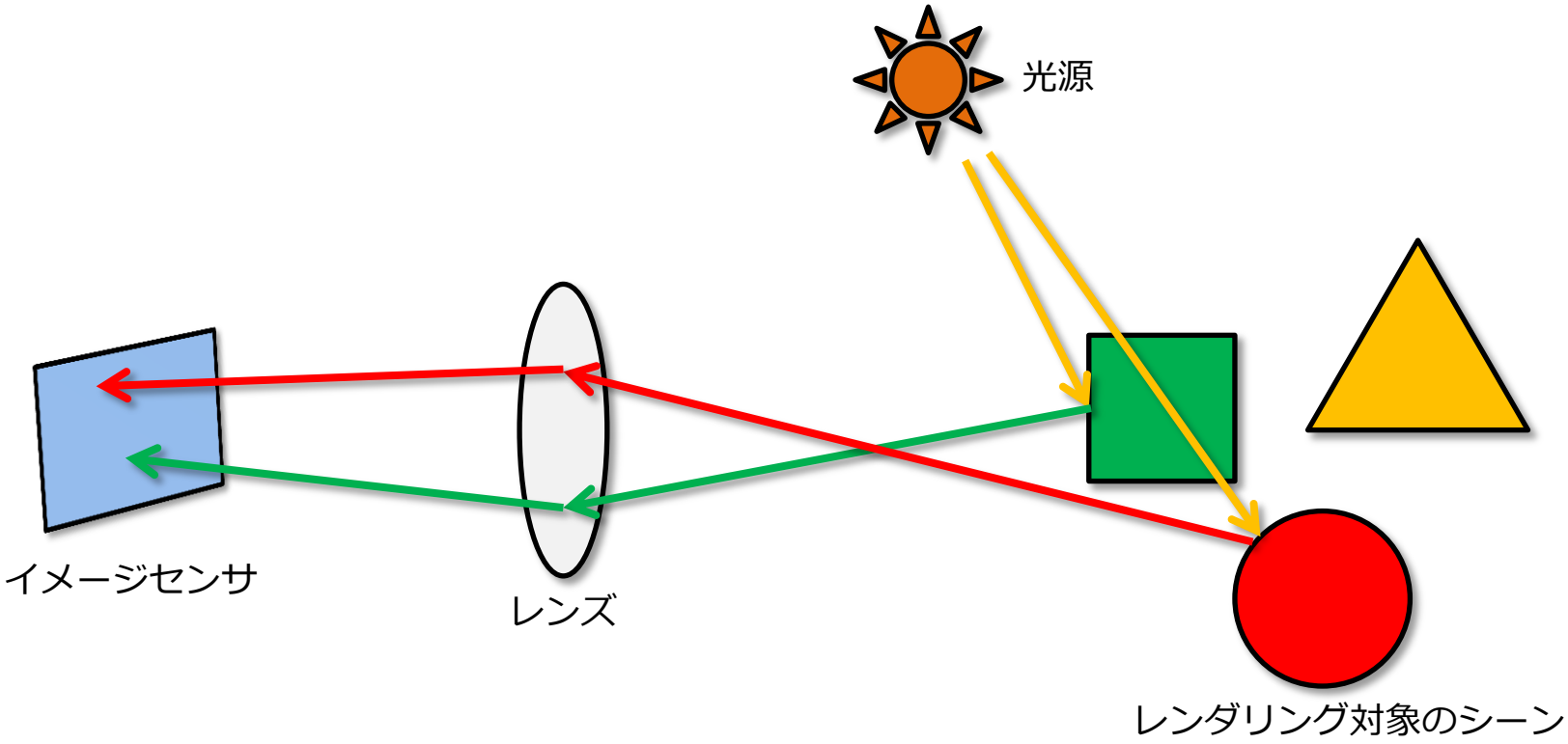
2. レンダリングの定式化

レンダリングの定式化

- レンダリングという問題を解くために、まずはどのような式でレンダリングという問題を表現できるかをはっきりさせる。
- そのために、カメラモデルと画素の受ける光について定式化する。

カメラモデル

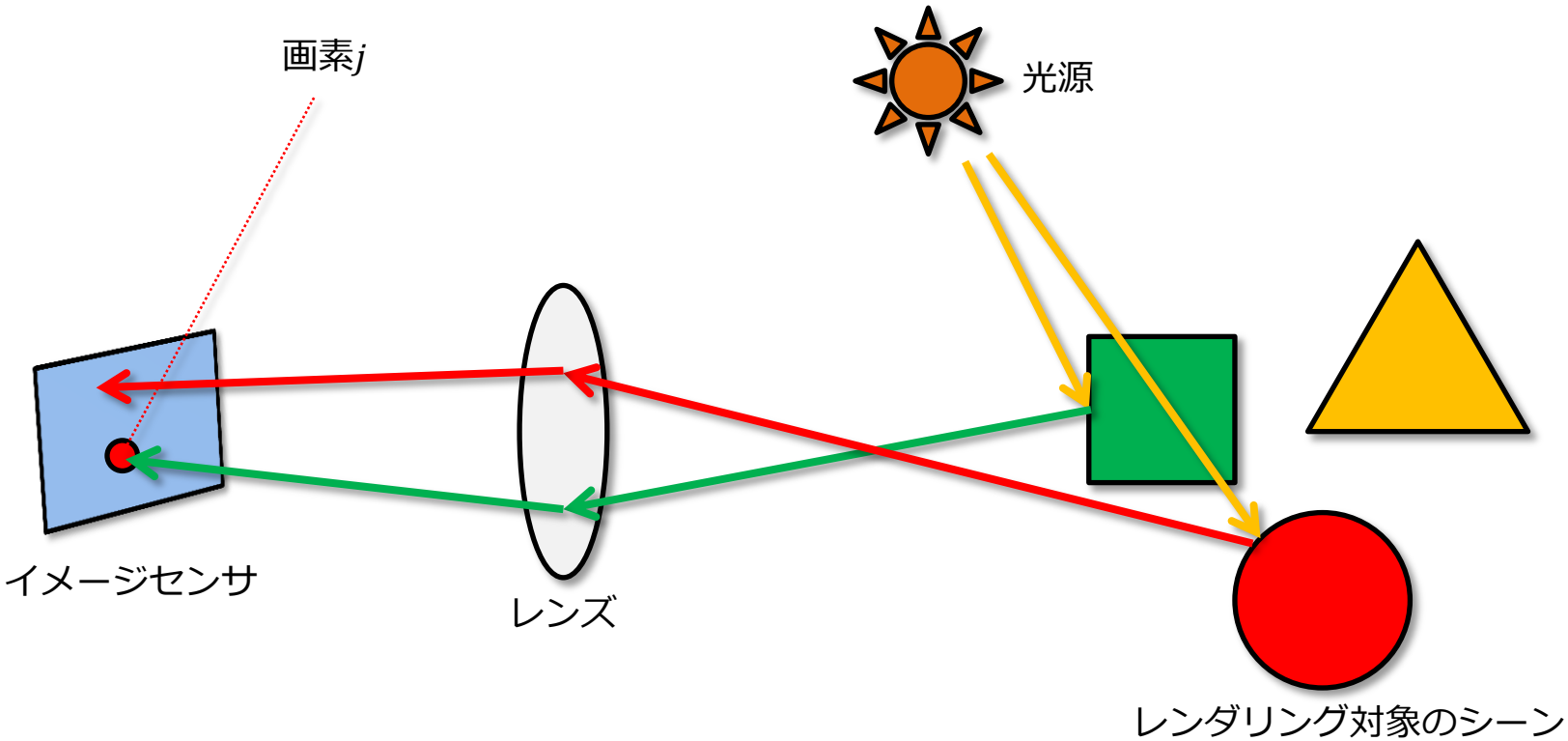
- イメージセンサに入射する光を物理的に計算し、それを各画素値に変換することで最終的な画像を得る。**→ レンダリング**
- 一般的なカメラの物理構造を模倣し、画像の計算モデルとして使う。



カメラモデル

カメラモデル

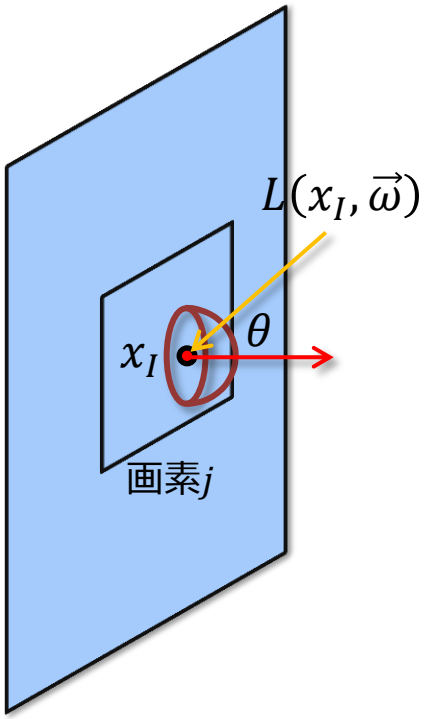
- イメージセンサ上のj番目の画素の値について考える



カメラモデル

画素の受ける光

- 画素jの最終的な値をI_jとする。
- この値は、画素jにおける**放射束**（単位時間あたりに画素を通過するフォトン数・エネルギー）と画素の**センシティビティ**（感応度）によって決まるとする。
  - 今回は露光時間については考慮せず、単位時間あたりのエネルギーについて考える。



x_Iに$\vec{\omega}$方向から入射する放射輝度

$$I_j = \int_{I_j} \int_{\Omega} \underline{W(x_I, \vec{\omega})} \underline{L(x_I, \vec{\omega}) \cos\theta} \, d\sigma(\vec{\omega}) \, dA_{I_j}(x_I)$$

センシティビティ

x_Iにおける入射放射輝度×コサイン項

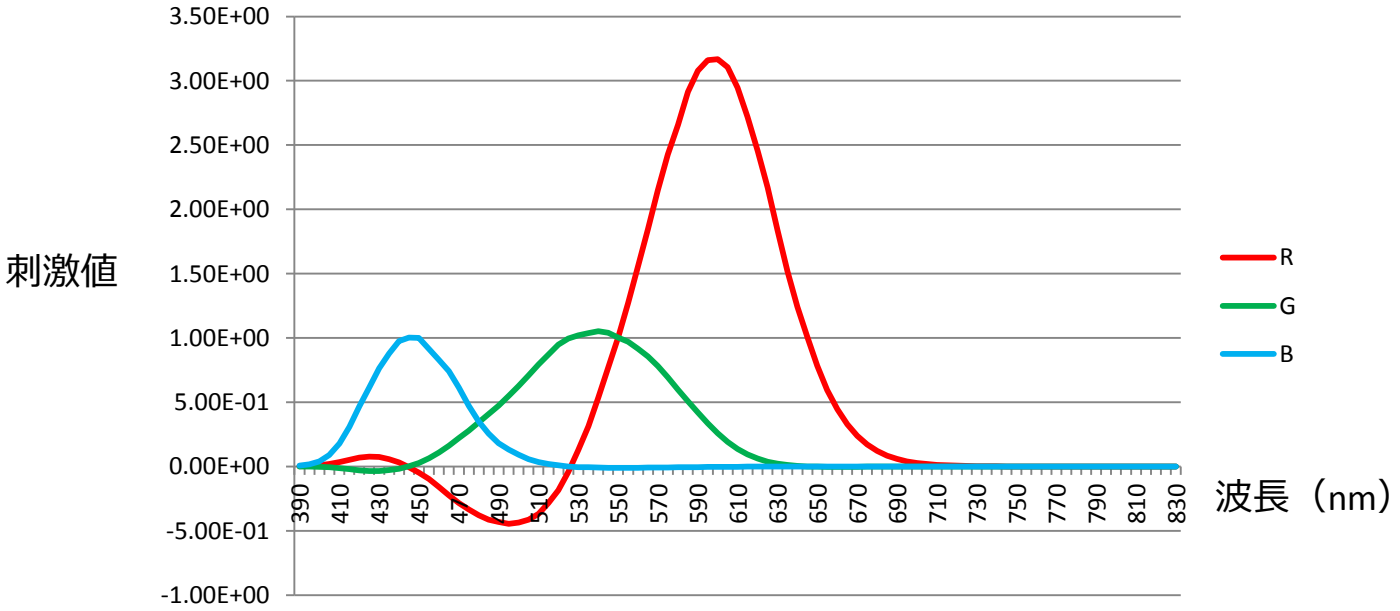
イラディアンスを面積分しているので得られるのは放射束になる。（ただしセンシティビティが入る）

光の波長

- 光は物理量としては波長ごとの放射輝度や放射照度として表現される。一方、画像に実際に記録されるのはRGB値である。これは、RGBの三種類だけで人間の認識できる色を表現できるからである。
- 人間の網膜には色を知覚する錐体細胞がある。この錐体細胞の種類は三つあることが知られており、その三つそれぞれが波長ごとに異なった刺激を受ける。つまり、人間の色の知覚は全てこの三種類の細胞の刺激の組み合わせで表現できる。
- そこで、三つの異なる波長の光をRGB原刺激として定め、これを様々な強度で組み合わせることによって任意の色を表現することが行われている。
  - たとえば、R刺激として波長700nm、G刺激として546.1nm、B刺激として435.8nmの光を考え、この三種類の刺激を組み合わせることで任意の色を表現する（CIE 1931）。

光の波長

- 波長ごとの色の知覚と、対応するRGB原刺激値は以下のようになる。
- 以下のグラフは実験的に定めたものなので、実験した年や環境によっても変化することに注意。



波長ごとのRGB刺激値（Stiles & Burch (1959) 10-deg, RGB CMF）

光の波長

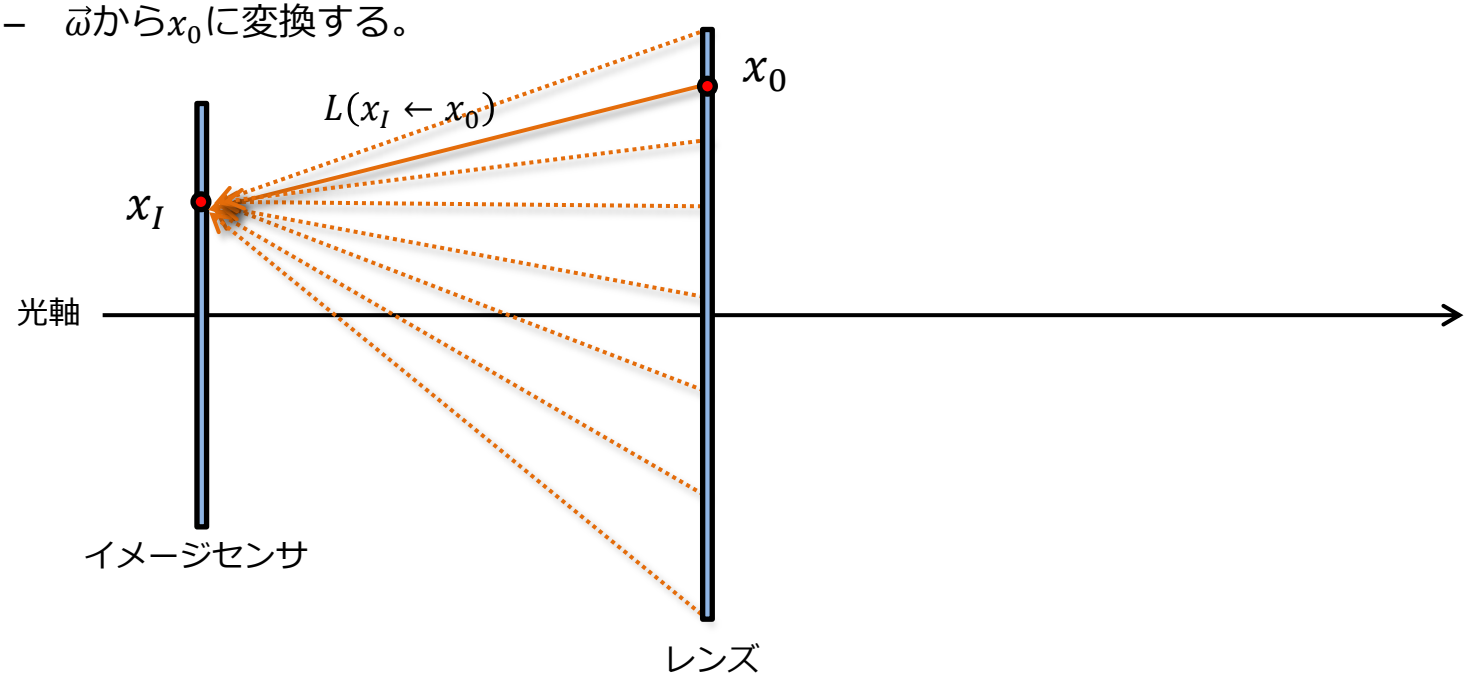
- 人間が知覚できる色を表現するのはRGB値で十分であった。そこで、レンダリングするときも全波長分の光の挙動を考慮するのではなく、RGB値に対応する波長のみ考慮する。Color構造体が三要素なのも、そのため。
- また、屈折率は波長ごとに変化するが、これも一種類とする。
- この簡易化により、光の波長の連続的な変化に由来する現象は再現できなくなる。（光の分散等）

センシティビティ

- RGBの三波長分のみ考慮してレンダリングするのだが、イメージセンサが受ける光の量というのはあくまで物理量である。
- 人間の網膜上では、入射した光は最終的に「刺激」という心理的な量になる。デジタルカメラのイメージセンサも、受け取る光は物理的実体だが、最終的に画像に保存するときには何らかのスケーリングが行われRGBについて相対的な量に変換される。この変換の係数がセンシティビティ。
- 今回のレンダラでは、センシティビティは画素の物理的大きさに反比例させている。このようにすることで、イメージセンサ全体の大きさを固定して、解像度だけ変えても、各画素の値が変化しないようにしている。
  - もしこうしないと解像度が変化すると各画素の物理的大きさが変化し、受け取る光の量も変わり、画素値も変化するため不便。

画素の受ける光

- 画素j内の点x_Iにおいて半球積分するわけだが、実際は**レンズを通ってきた光のみ**が画素jに寄与する。
- 半球積分する代わりに、レンズ上の点x_0 からx_Iへの放射輝度$L(x_I \leftarrow x_0)$を積分することでx_Iにおけるイラディアンスを計算、最終的な画素値を求めることを考える。
  - x_0をレンズ上を動く積分変数にする。
- これは、積分変数を変数変換すればよい。
  - $\vec{\omega}$からx_0に変換する。

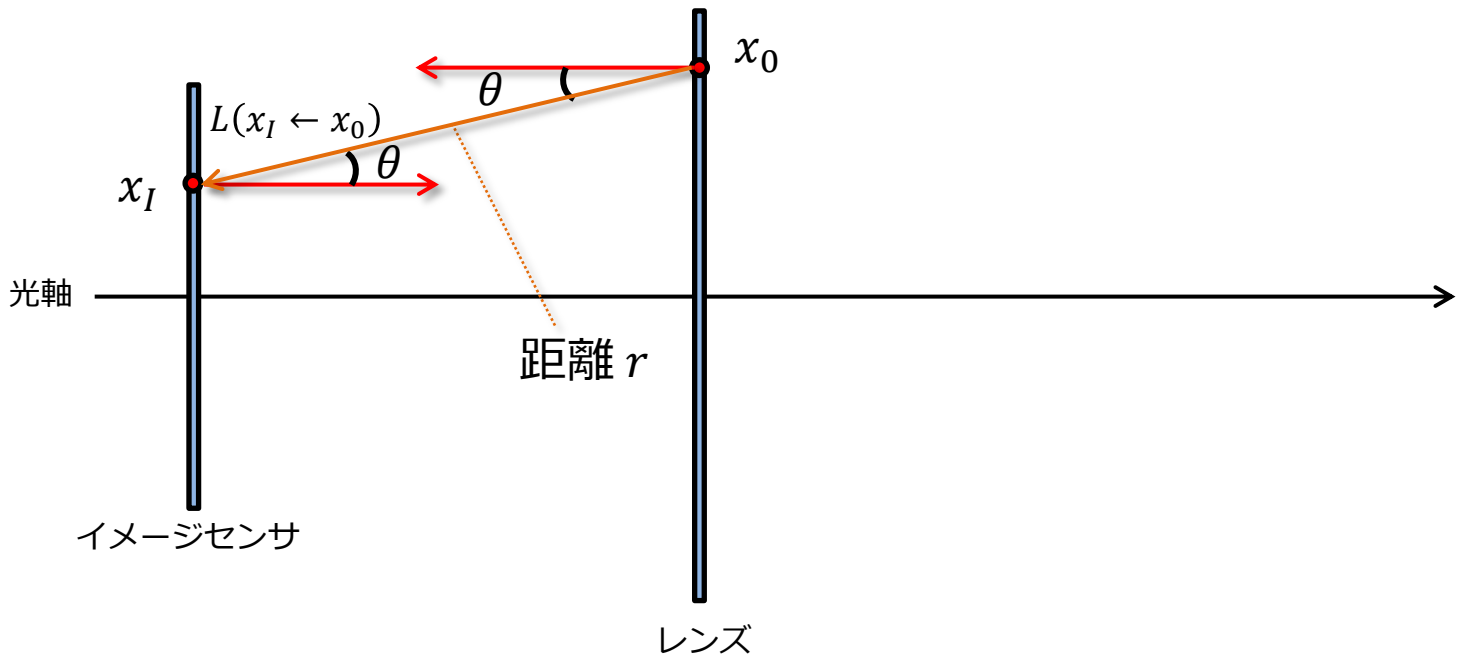


レンズモデルとして、厚さのない平面でレンズを近似する薄レンズモデルを使う。

画素の受ける光

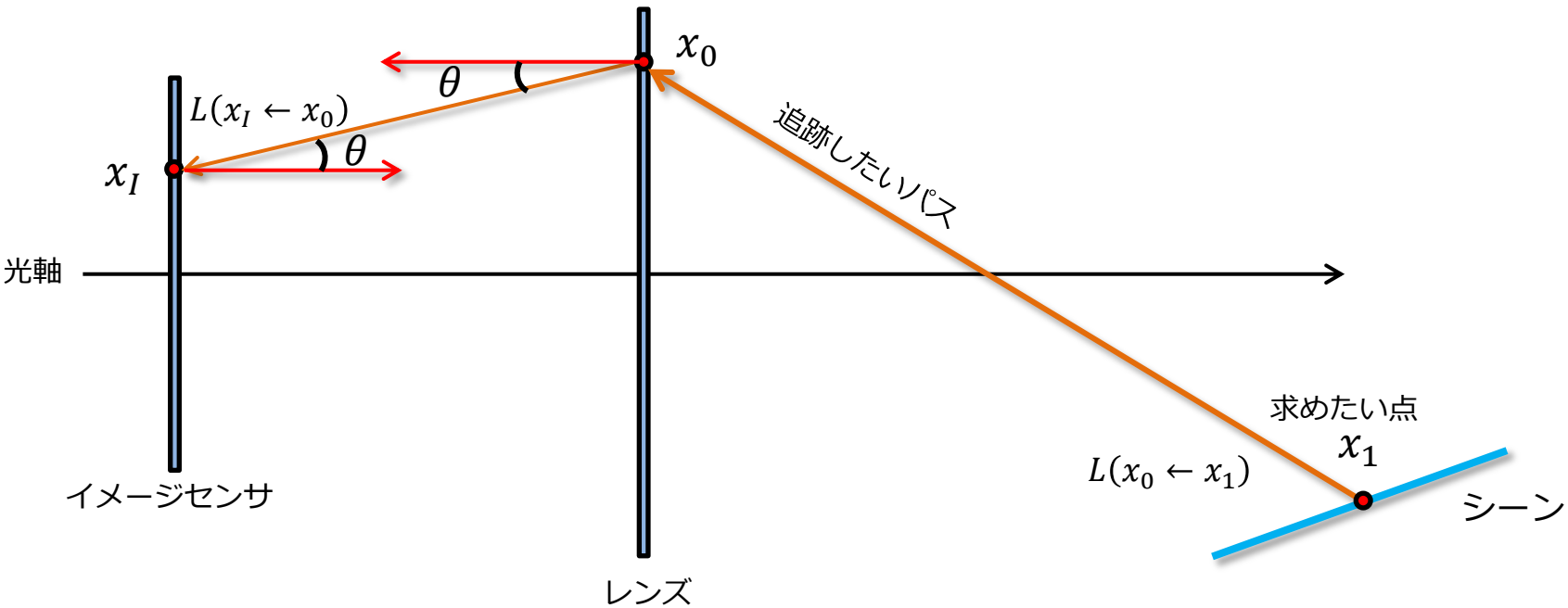
- 微小立体角と微小面積の関係式$d\sigma(\vec{\omega}) = \frac{\cos\theta}{r^2} dA(x_0)$を使うことで積分変数と積分範囲を変換できる。
  - 今回は、レンズをシーン中のほかのオブジェクトと同じ物理的実体とみなすため、レンズ上の微小面積はシーン表面に対する測度と同じ面積測度を用いて得ることが出来ると考える。

$$
\begin{aligned}
I_j &= \int_{I_j} \int_{\Omega} W(x_I, \vec{\omega}) L(x_I, \vec{\omega}) \cos\theta \, d\sigma(\vec{\omega}) \, dA_{I_j}(x_I) \\
&= \int_{I_j} \int_{Lens} W(x_I \leftarrow x_0) L(x_I \leftarrow x_0) \frac{\cos^2\theta}{r^2} dA(x_0) \, dA_{I_j}(x_I)
\end{aligned}
$$



画素の受ける光

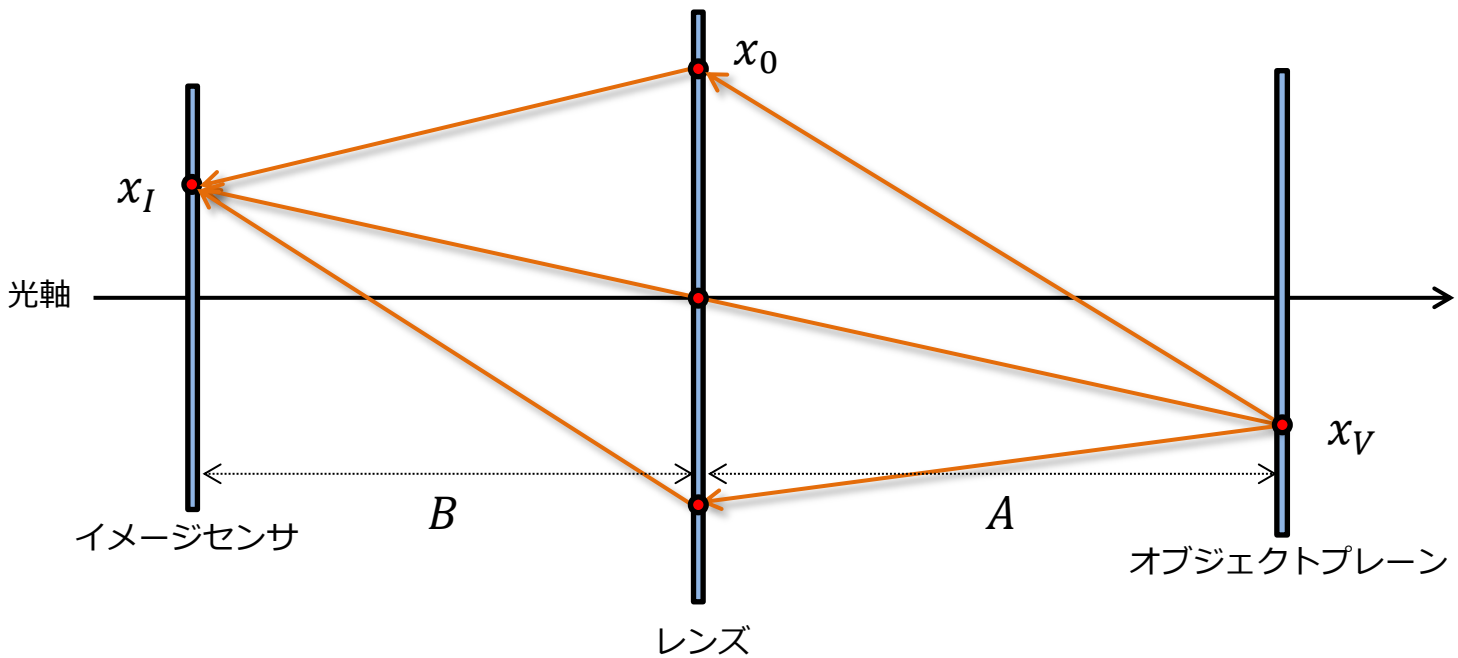
- $L(x_I \leftarrow x_0)$を計算するために、シーン内の点x_1からx_0に到達し、レンズを通過してx_Iへと至るパスを計算する必要がある。
  - このとき、$L(x_I \leftarrow x_0) = L(x_0 \leftarrow x_1)$となる。
  - **x_Iとx_1はそれぞれイメージセンサ上の点、シーン上の点で、別なので注意。**
- このような点x_1は、x_Iとx_0が定まると自然に決まる。
  - レンズの公式・結像公式



レンズの公式

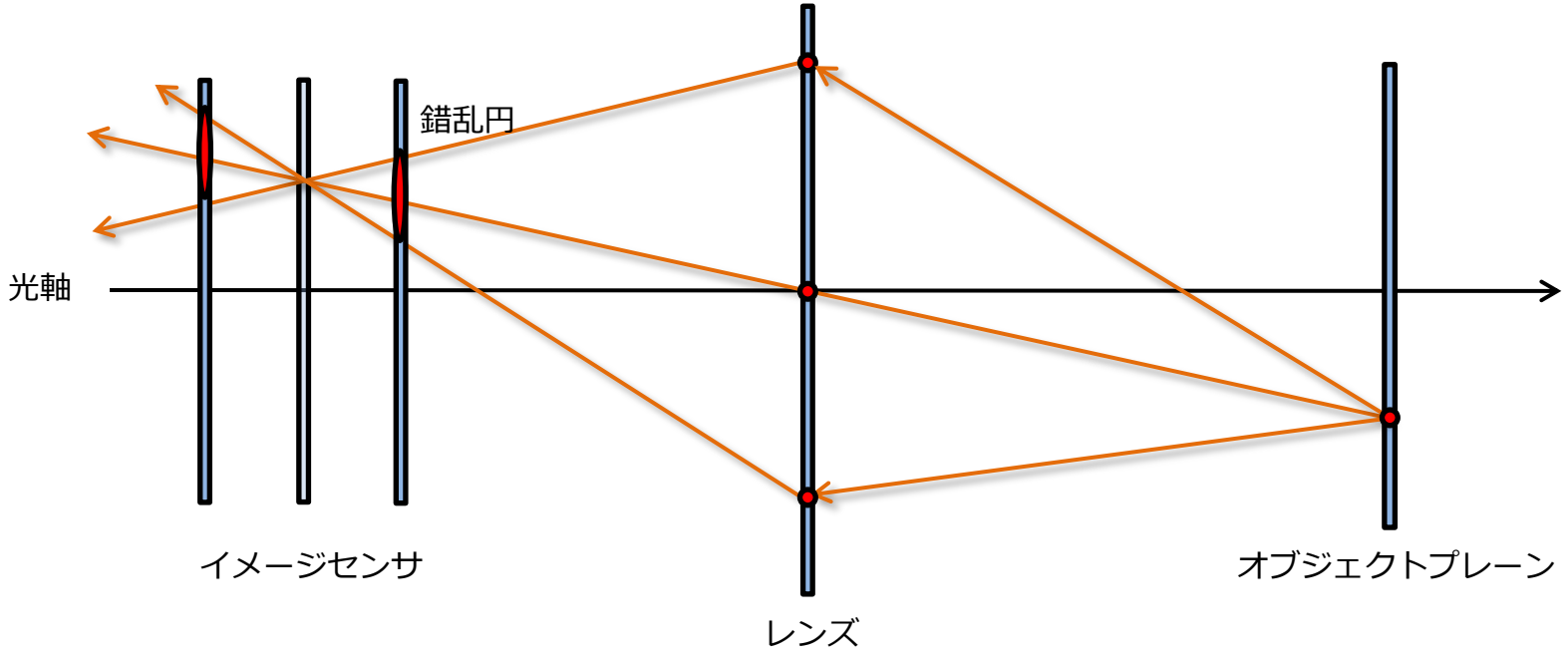
- 幾何光学における、レンズと焦点の関係式。
  - イメージセンサとレンズの距離をB、レンズとオブジェクトプレーンの距離をA、レンズの焦点距離をFとすると、以下の関係式が成り立つ。
  - ここでオブジェクトプレーンとは、イメージセンサ上の一点から放射された光が再び一点で収束する点が作る平面のことをいう。また、逆も成り立つ。

$$\frac{1}{A}+\frac{1}{B}=\frac{1}{F}$$



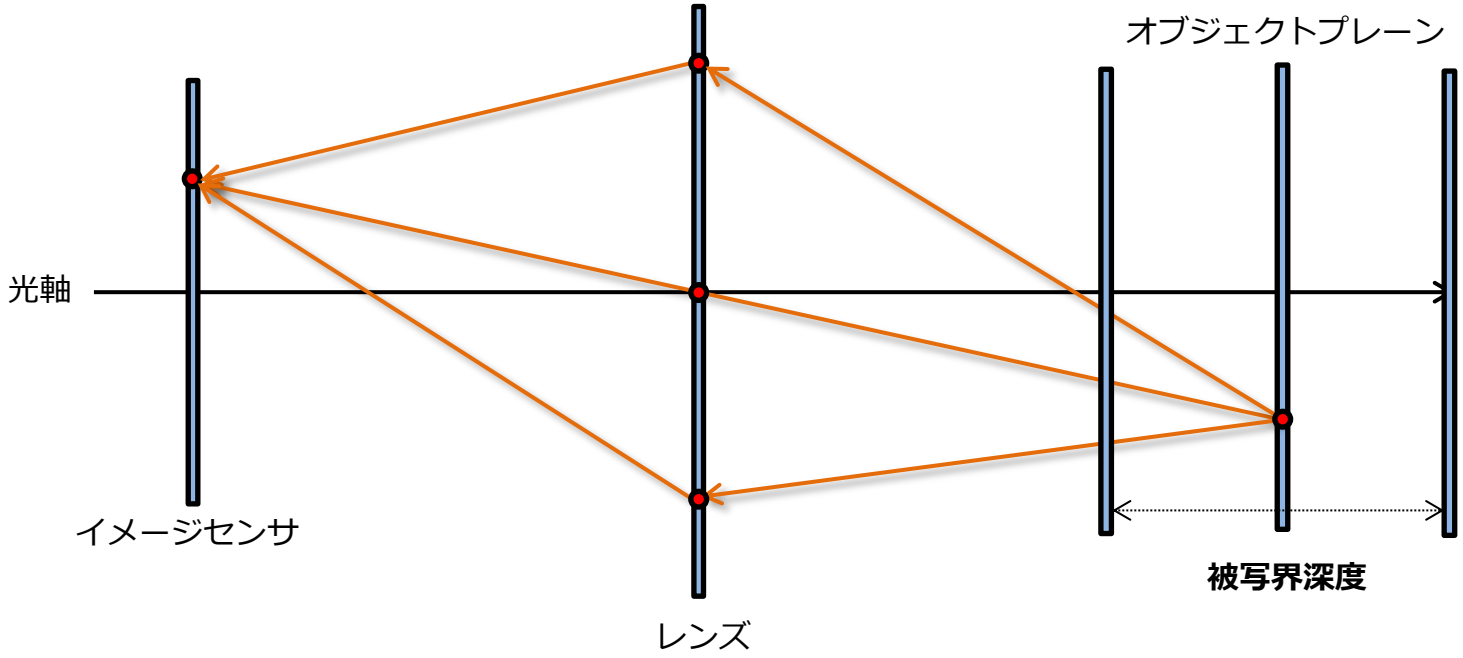
許容錯乱円

- イメージセンサとオブジェクトプレーンの位置関係によっては、オブジェクトプレーン上で点であった物体がイメージプレーン上では円になる。（錯乱円）
  - この円が、イメージセンサ上での画素（受光素子）よりも大きくなると、周囲の画素に影響が波及し、いわゆるピンボケが起こる。
  - この限界の円を許容錯乱円という。



被写界深度

- 錯乱円の大きさが許容錯乱円以下になる（＝焦点が合う）、オブジェクトプレーンの範囲を被写界深度と呼ぶ。



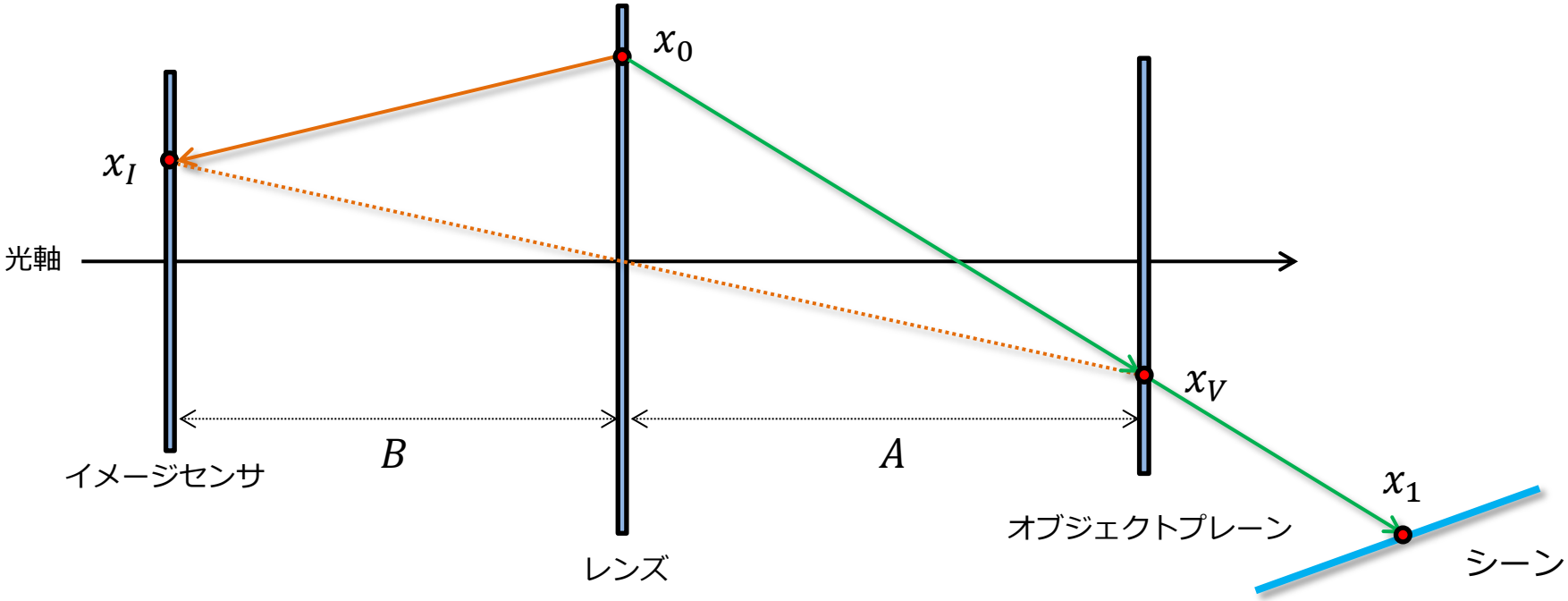
ボケ（Bokeh）

- 今回はレンズ形状を円形にしているため、錯乱円の形状も円形。
- 物理的なカメラは絞りと呼ばれる機構によって入射する光の量を制御している。
- 絞りの形状によって見かけのレンズ形状が変化するため、錯乱円の形状が円形以外（六角形、五角形等）を取ることがある。
  - 今回は行っていないが、レンズ上でx_0をサンプリングする際、絞り形状の中からサンプリングするようにすれば、任意の形状のボケを得ることが出来る。

今回はカメラモデルとして有限の大きさを持つレンズの存在を考慮しているため、これらのボケ効果や被写界深度によるピンボケなどは全て再現される。

画素の受ける光

- 以上より、イメージセンサとレンズの距離、レンズの焦点距離が決まるとレンズとオブジェクトプレーンの距離・位置も決まる。
- よって、 x_Iとレンズの中心点を結ぶ直線とオブジェクトプレーンの交点を求めることが出来る。これをx_Vとする。
- x_0からx_Vに向かってレイトレーシングしてシーンとの交点を求めれば、それがx_1となる。

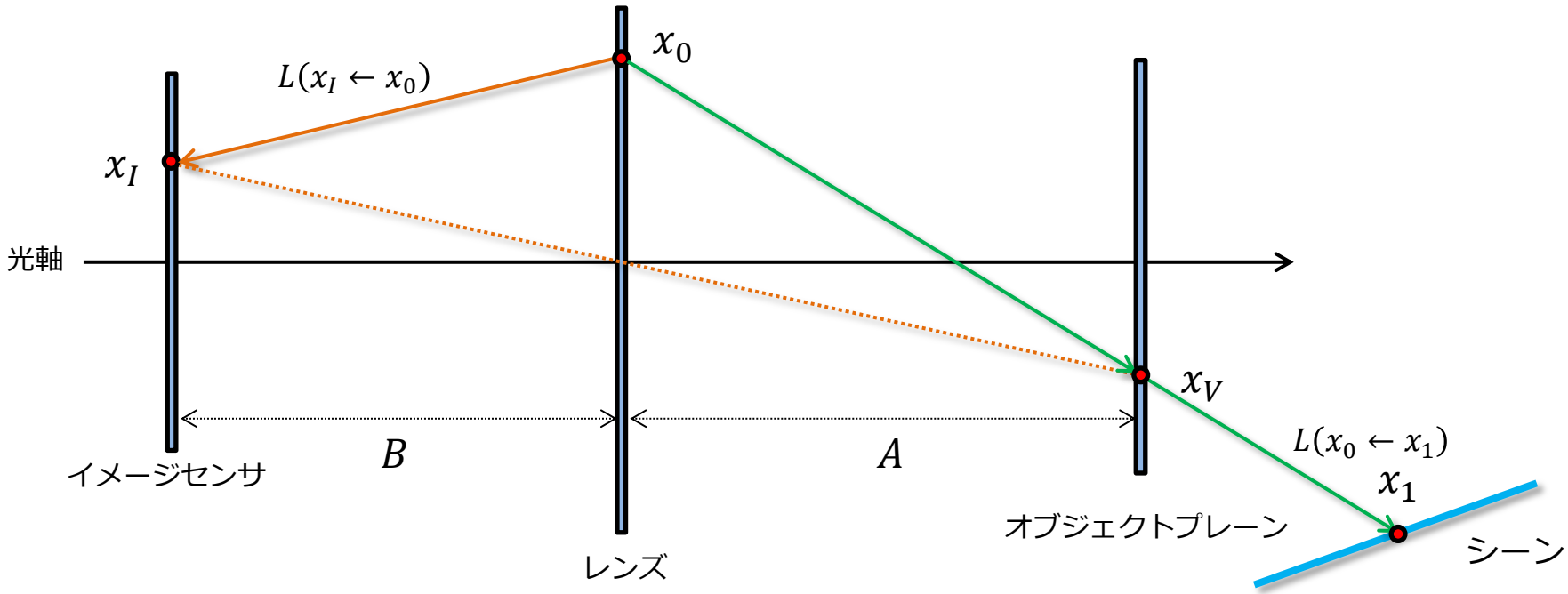


画素の受ける光

- 以上より、画素jの値は

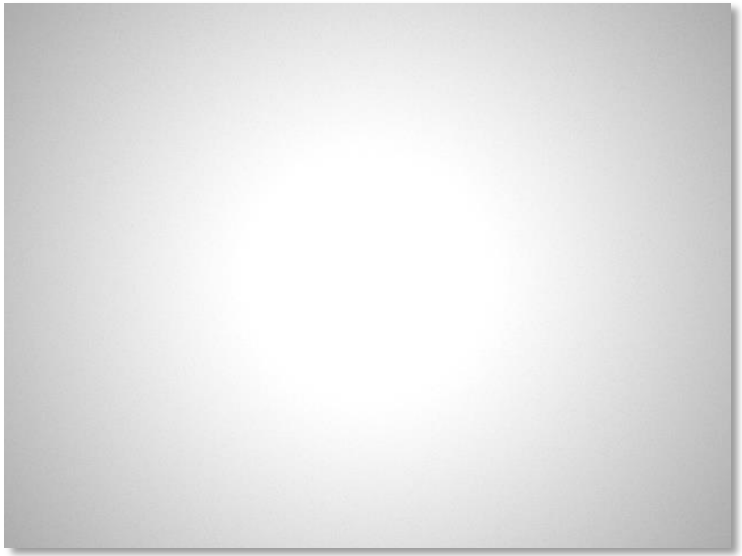
$$I_j = \int_{I_j}\int_{Lens} W(x_I \leftarrow x_0) L(x_0 \leftarrow x_1) \frac{\cos^2\theta}{r^2} dA(x_0)\, dA_{I_j}(x_I)$$

- $L(x_0 \leftarrow x_1)$はレンダリング方程式を使えば計算可能。
  - **パストレーシング**



ビネッティング

- 以下は、白い光源だけをレンダリングした結果。画像周辺部分が暗くなっている。
- これは、現実のカメラにおけるビネッティング（周辺光量低下）である。特に、ナチュラルビネッティングと呼ぶ。
- ひとつ前のスライドの積分中、$\frac{\cos^2\theta}{r^2}$について、$r = \frac{B}{\cos\theta}$で置き換えると、$\frac{\cos^4\theta}{B^2}$となる。所謂コサイン四乗則である。
- イメージセンサの端に行くほど、θの最大値も大きくなり、コサイン四乗則による減衰も大きくなるため、画像周辺部が暗くなる。



3. パストレーシング

パストレーシング

- 前章で解くべき式が得られた。

$$I_j = \int_{I_j} \int_{Lens} W(x_I \leftarrow x_0) L(x_0 \leftarrow x_1) \frac{\cos^2 \theta}{r^2} dA(x_0)\, dA_{I_j}(x_I)$$

- 双方向パストレについて考える前に、この式を普通のパストレーシングで解いてみる。
  - reference_pathtracing.hがこれに対応している。
  - カメラモデル以外は、eduptのパストレーシングと大きく変わらない。

レンダリング方程式

- イメージセンサの式とレンダリング方程式の二つがあれば問題を解くことができる。
  - 特に、下のレンダリング方程式についてはeduptにおけるやり方と全く同様にして解くことができる。

$$I_j = \int_{I_j} \int_{Lens} W(x_I \leftarrow x_0) L(x_0 \leftarrow x_1) \frac{\cos^2 \theta}{r^2} dA(x_0)\, dA_{I_j}(x_I)$$

$$L(x_0 \leftarrow x_1) = L_e(x_0 \leftarrow x_1) + \int_{\Omega} L(x_1 \leftarrow x_2) \underbrace{f_r(x_0 \leftarrow x_1 \leftarrow x_2)}_{\text{BRDF}} \cos\theta\, d\sigma(x_1 \leftarrow x_2)$$

レンダリング方程式

対応するソースコード

- camera.hのstruct Cameraに、カメラパラメータを設定。
- render.hのrender_by_refernce_pathtracing()が画像の各画素値を計算する関数で、I_jを解いている。
  - 最初の点、 x_0とx_IをCameraを使ってサンプリング。
- render_by_refernce_pathtracing()の中で呼んでいるradiance_no_recursion がレンダリング方程式を解く関数で、パストレーシング部分。

render.h

- render_by_reference_pathtracing()
  - この関数が画素値についてのモンテカルロ積分を行い、実際にレンダリングを担当。

```
int render_by_refernce_pathtracing(const Camera &camera, const int num_threads) {
    const int width = camera.image_width_px;
    const int height = camera.image_height_px;
    const int spp = camera.samples_per_pixel;

    std::vector<Color> image_buffer(width * height);
#ifdef USE_OPENMP
    omp_set_num_threads(num_threads);
#endif

    for (int sample = 0; sample < spp; ++sample) {
    std::cerr << "Rendering (" << (sample + 1) << " samples/pixel) " << (100.0 * (sample + 1) / spp) << "%" << std::endl;

#pragma omp parallel for schedule(dynamic, 1)
        for (int y = 0; y < height; ++y) {
            Random random(y * spp + sample);
            for (int x = 0; x < width; ++x) {
                Vec position_on_imagesensor, position_on_objectplane, position_on_lens;
                double P_Image, P_lens;
                camera.sample_points(x, y, &random, &position_on_imagesensor, &position_on_objectplane, &position_on_lens, &P_Image, &P_lens);

                // 方向
                const Vec dir = normalize(position_on_objectplane - position_on_lens);
                // L(x0 <- x1) (= L(xI <- x0) = L(x0 <- xV))
                const Color L = radiance_no_recursion(Ray(position_on_lens, dir), &random);
                if (!is_valid_value(L)) // 異常値は除外する
                    continue;

                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const double coefficient = pow(dot(normalize(-1.0 * camera.imagesensor_dir), normalize(x0_xI)), 2.0) / x0_xI.length_squared(); // センサ上の点の半球積分を、レンズ上の点の積分に変数変換した時に導入される係数

                image_buffer[y * width + x] = image_buffer[y * width + x] +
                    camera.W * L * coefficient / (P_Image * P_lens);
            }
        }
    }

    // サンプル数で割る + 左右反転
    std::vector<Color> finale_buffer(width * height);
    for (int y = 0; y < height; ++y)
        for (int x = 0; x < width; ++x)
            finale_buffer[y * width + x] = image_buffer[y * width + (width - x - 1)] / spp;
    // 出力
    save_ppm_file("image_ref_pt.ppm", &finale_buffer[0], width, height);

    return 0;
}
```

render.h

- 画素値についてのモンテカルロ積分
  - レンズ上の点x_0とイメージセンサ上の点x_Iをサンプリング。
  - 各点のサンプリング確率密度も得る。

解いている式 $$I_j = \int_{I_j} \int_{Lens} W(x_I \leftarrow x_0) L(x_0 \leftarrow x_1) \frac{\cos^2 \theta}{r^2} dA(x_0)\, dA_{I\,j}(x_I)$$

```
int render_by_refernce_pathtracing(const Camera &camera, const int num_threads) {
    const int width = camera.image_width_px;
    const int height = camera.image_height_px;
    const int spp = camera.samples_per_pixel;

    std::vector<Color> image_buffer(width * height);
#ifdef USE_OPENMP
    omp_set_num_threads(num_threads);
#endif

    for (int sample = 0; sample < spp; ++sample) {
    std::cerr << "Rendering (" << (sample + 1) << " samples/pixel) " << (100.0 * (sample + 1) / spp) << "%" << std::endl;

#pragma omp parallel for schedule(dynamic, 1)
        for (int y = 0; y < height; ++y) {
            Random random(y * spp + sample);
            for (int x = 0; x < width; ++x) {
                Vec position_on_imagesensor, position_on_objectplane, position_on_lens;
                double P_Image, P_lens;
                camera.sample_points(x, y, &random, &position_on_imagesensor, &position_on_objectplane, &position_on_lens, &P_Image, &P_lens);

                // 方向
                const Vec dir = normalize(position_on_objectplane - position_on_lens);
                // L(x0 <- x1) (= L(xI <- x0) = L(x0 <- xV))
                const Color L = radiance_no_recursion(Ray(position_on_lens, dir), &random);
                if (!is_valid_value(L)) // 異常値は除外する
                    continue;

                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const double coefficient = pow(dot(normalize(-1.0 * camera.imagesensor_dir), normalize(x0_xI)), 2.0) / x0_xI.length_squared(); // センサ上の点の半球積分を、レンズ上の点の積分に変数変換した時に導入される係数

                image_buffer[y * width + x] = image_buffer[y * width + x] +
                    camera.W * L * coefficient / (P_Image * P_lens);
            }
        }
    }

    // サンプル数で割る + 左右反転
    std::vector<Color> finale_buffer(width * height);
    for (int y = 0; y < height; ++y)
        for (int x = 0; x < width; ++x)
            finale_buffer[y * width + x] = image_buffer[y * width + (width - x - 1)] / spp;
    // 出力
    save_ppm_file("image_ref_pt.ppm", &finale_buffer[0], width, height);

    return 0;
}
```

render.h

- 画素値についてのモンテカルロ積分
  - 続いて、$L(x_0 \leftarrow x_1)$を計算する。これはradiance_no_recursionが担当しており、単純なパストレーシング。

解いている式 $$I_j = \int_{I_j} \int_{Lens} W(x_I \leftarrow x_0) L(x_0 \leftarrow x_1) \frac{\cos^2 \theta}{r^2} dA(x_0)\, dA_{I_j}(x_I)$$

```
int render_by_refernce_pathtracing(const Camera &camera, const int num_threads) {
    const int width = camera.image_width_px;
    const int height = camera.image_height_px;
    const int spp = camera.samples_per_pixel;

    std::vector<Color> image_buffer(width * height);
#ifdef USE_OPENMP
    omp_set_num_threads(num_threads);
#endif

    for (int sample = 0; sample < spp; ++sample) {
    std::cerr << "Rendering (" << (sample + 1) << " samples/pixel) " << (100.0 * (sample + 1) / spp) << "%" << std::endl;

#pragma omp parallel for schedule(dynamic, 1)
        for (int y = 0; y < height; ++y) {
            Random random(y * spp + sample);
            for (int x = 0; x < width; ++x) {
                Vec position_on_imagesensor, position_on_objectplane, position_on_lens;
                double P_Image, P_lens;
                camera.sample_points(x, y, &random, &position_on_imagesensor, &position_on_objectplane, &position_on_lens, &P_Image, &P_lens);

                // 方向
                const Vec dir = normalize(position_on_objectplane - position_on_lens);
                // L(x0 <- x1) (= L(xI <- x0) = L(x0 <- xV))
                const Color L = radiance_no_recursion(Ray(position_on_lens, dir), &random);
                if (!is_valid_value(L)) // 異常値は除外する
                    continue;

                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const double coefficient = pow(dot(normalize(-1.0 * camera.imagesensor_dir), normalize(x0_xI)), 2.0) / x0_xI.length_squared(); // センサ上の点の半球積分を、レンズ上の点の積分に変数変換した時に導入される係数

                image_buffer[y * width + x] = image_buffer[y * width + x] +
                    camera.W * L * coefficient / (P_Image * P_lens);
            }
        }
    }

    // サンプル数で割る + 左右反転
    std::vector<Color> finale_buffer(width * height);
    for (int y = 0; y < height; ++y)
        for (int x = 0; x < width; ++x)
            finale_buffer[y * width + x] = image_buffer[y * width + (width - x - 1)] / spp;
    // 出力
    save_ppm_file("image_ref_pt.ppm", &finale_buffer[0], width, height);

    return 0;
}
```

render.h

- 画素値についてのモンテカルロ積分
  - $\frac{\cos^2\theta}{r^2}$を計算。
  - 最後に$\frac{W(x_I \leftarrow x_0)L(x_0 \leftarrow x_1)\frac{\cos^2\theta}{r^2}}{P_{I_j}(x_I)P_A(x_0)}$を計算して画素値に加算する（モンテカルロ積分）。

解いている式 $$I_j = \int_{I_j}\int_{Lens} W(x_I \leftarrow x_0)L(x_0 \leftarrow x_1)\frac{\cos^2\theta}{r^2}\,dA(x_0)\,dA_{I_j}(x_I)$$

```
int render_by_refernce_pathtracing(const Camera &camera, const int num_threads) {
    const int width = camera.image_width_px;
    const int height = camera.image_height_px;
    const int spp = camera.samples_per_pixel;

    std::vector<Color> image_buffer(width * height);
#ifdef USE_OPENMP
    omp_set_num_threads(num_threads);
#endif

    for (int sample = 0; sample < spp; ++sample) {
    std::cerr << "Rendering (" << (sample + 1) << " samples/pixel) " << (100.0 * (sample + 1) / spp) << "%" << std::endl;

#pragma omp parallel for schedule(dynamic, 1)
        for (int y = 0; y < height; ++y) {
            Random random(y * spp + sample);
            for (int x = 0; x < width; ++x) {
                Vec position_on_imagesensor, position_on_objectplane, position_on_lens;
                double P_Image, P_lens;
                camera.sample_points(x, y, &random, &position_on_imagesensor, &position_on_objectplane, &position_on_lens, &P_Image, &P_lens);

                // 方向
                const Vec dir = normalize(position_on_objectplane - position_on_lens);
                // L(x0 <- x1) (= L(xI <- x0) = L(x0 <- xV))
                const Color L = radiance_no_recursion(Ray(position_on_lens, dir), &random);
                if (!is_valid_value(L)) // 異常値は除外する
                    continue;

                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const double coefficient = pow(dot(normalize(-1.0 * camera.imagesensor_dir), normalize(x0_xI)), 2.0) / x0_xI.length_squared(); // センサ上の点の半球積分を、レンズ上の点の積分に変数変換した時に導入される係数

                image_buffer[y * width + x] = image_buffer[y * width + x] +
                    camera.W * L * coefficient / (P_Image * P_lens);
            }
        }
    }

    // サンプル数で割る + 左右反転
    std::vector<Color> finale_buffer(width * height);
    for (int y = 0; y < height; ++y)
        for (int x = 0; x < width; ++x)
            finale_buffer[y * width + x] = image_buffer[y * width + (width - x - 1)] / spp;
    // 出力
    save_ppm_file("image_ref_pt.ppm", &finale_buffer[0], width, height);

    return 0;
}
```

render.h

- 画素値についてのモンテカルロ積分
  - 画像の出力時に、サンプリング数で割る。

解いている式 $$I_j = \int_{I_j} \int_{Lens} W(x_I \leftarrow x_0) L(x_0 \leftarrow x_1) \frac{\cos^2 \theta}{r^2} dA(x_0)\, dA_{I_j}(x_I)$$

```
int render_by_refernce_pathtracing(const Camera &camera, const int num_threads) {
    const int width = camera.image_width_px;
    const int height = camera.image_height_px;
    const int spp = camera.samples_per_pixel;

    std::vector<Color> image_buffer(width * height);
#ifdef USE_OPENMP
    omp_set_num_threads(num_threads);
#endif

    for (int sample = 0; sample < spp; ++sample) {
    std::cerr << "Rendering (" << (sample + 1) << " samples/pixel) " << (100.0 * (sample + 1) / spp) << "%" << std::endl;

#pragma omp parallel for schedule(dynamic, 1)
        for (int y = 0; y < height; ++y) {
            Random random(y * spp + sample);
            for (int x = 0; x < width; ++x) {
                Vec position_on_imagesensor, position_on_objectplane, position_on_lens;
                double P_Image, P_lens;
                camera.sample_points(x, y, &random, &position_on_imagesensor, &position_on_objectplane, &position_on_lens, &P_Image, &P_lens);

                // 方向
                const Vec dir = normalize(position_on_objectplane - position_on_lens);
                // L(x0 <- x1) (= L(xI <- x0) = L(x0 <- xV))
                const Color L = radiance_no_recursion(Ray(position_on_lens, dir), &random);
                if (!is_valid_value(L)) // 異常値は除外する
                    continue;

                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const double coefficient = pow(dot(normalize(-1.0 * camera.imagesensor_dir), normalize(x0_xI)), 2.0) / x0_xI.length_squared(); // センサ上の点の半球積分を、レンズ上の点の積分に変数変換した時に導入される係数

                image_buffer[y * width + x] = image_buffer[y * width + x] +
                    camera.W * L * coefficient / (P_Image * P_lens);
            }
        }
    }

    // サンプル数で割る + 左右反転
    std::vector<Color> finale_buffer(width * height);
    for (int y = 0; y < height; ++y)
        for (int x = 0; x < width; ++x)
            finale_buffer[y * width + x] = image_buffer[y * width + (width - x - 1)] / spp;
    // 出力
    save_ppm_file("image_ref_pt.ppm", &finale_buffer[0], width, height);

    return 0;
}
```

camera.h

- **sample_points()**
- **レンズ上の点x_0とイメージセンサ上の点x_Iをサンプリング。**
  - まずイメージセンサ上で普通にサンプリング。
  - その後、対応するオブジェクトプレーン上の位置を計算。
    - これは前章で説明したとおり、レンズの公式を使って求まる。
  - 最後に、レンズ上でサンプリング（円内にサンプリング点を生成）。

```
// イメージセンサ上とレンズ上にサンプルを生成する
void sample_points(
    const int imagebuffer_x, const int imagebuffer_y, Random *random,
    Vec *position_on_imagesensor, Vec *position_on_objectplane, Vec *position_on_lens,
    double *P_Image, double *P_lens) const {
    // イメージセンサー上でサンプル生成
    // 生成するのはimagebuffer_x, yのピクセル内

    // ピクセル内の座標、[0, 1]の範囲
    const double u_on_pixel = random->next01();
    const double v_on_pixel = random->next01();
    // イメージセンサ上の座標、[-0,5, 0.5]の範囲 (0,0)がイメージセンサ中央を示す。
    const double u_on_imagesensor = ((imagebuffer_x + u_on_pixel) / image_width_px - 0.5);
    const double v_on_imagesensor = ((imagebuffer_y + v_on_pixel) / image_height_px - 0.5);
    *position_on_imagesensor =
        imagesensor_center +
        u_on_imagesensor * imagesensor_u +
        v_on_imagesensor * imagesensor_v; // xI

    // オブジェクトプレーン上の座標計算
    const double ratio = focal_length / imagesensor_to_lens_distance;
    const double u_on_objectplane = -ratio * u_on_imagesensor;
    const double v_on_objectplane = -ratio * v_on_imagesensor;
    *position_on_objectplane =
        objectplane_center +
        u_on_objectplane * objectplane_u +
        v_on_objectplane * objectplane_v; // xV

    // レンズ上でサンプル生成 [-1, 1]
    const double r0 = sqrt(random->next01());
    const double r1 = random->next01() * 2.0 * kPI;
    const double u_on_lens = r0 * cos(r1);
    const double v_on_lens = r0 * sin(r1);
    *position_on_lens =
        lens_center +
        u_on_lens * lens_u +
        v_on_lens * lens_v; // x0

    *P_Image = 1.0 / (pixel_width * pixel_height); // ピクセル内の一点をサンプリングする確率密度関数（面積測度）
    *P_lens = sampling_pdf_on_lens(); // レンズ上の一点をサンプリングする確率密度関数（面積測度）
}
```

reference_pathtracing.h

- radiance_no_recursion()によって$L(x_0 \leftarrow x_1)$を計算
  - 再帰を使っていないだけでeduptのradiance関数とほぼ同じ。
  - パストレーシングによってシーン内の光の経路を追跡する。

```
#ifndef _REFERENCE_PATHTRACING_H_
#define _REFERENCE_PATHTRACING_H_

#include "random.h"
#include "scene.h"
#include "util.h"
#include "camera.h"
#include "specular.h"
#include "sampling.h"
#include "camera.h"

namespace edubpt {

// L(xi <- xi+1) を計算する
Color radiance_no_recursion(const Ray &ray, Random *rnd) {
    Color L;
    Ray now_ray = ray;
    Color weight(1, 1, 1);
    double pdf = 1.0;

    for (;;) {
        Intersection intersection;
        if (!intersect_scene(now_ray, &intersection)) {
            L = L + multiply(weight, BackgroundColor) / pdf;
            break;
        }
        // ロシアンルーレット
        const Sphere &now_object = spheres[intersection.object_id];
        const double russian_roulette_probability = russian_roulette(now_object);
        if (rnd->next01() >= russian_roulette_probability) {
            break;
        }
        pdf *= russian_roulette_probability;

        // 放射輝度加算
        L = L + multiply(weight, now_object.emission) / pdf;

        const Hitpoint &hitpoint = intersection.hitpoint;
         // 交差位置の法線（物体からのレイの入出を考慮）
        const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal);

        switch (now_object.reflection_type) {
        // 完全拡散面
        case REFLECTION_TYPE_DIFFUSE: {
            double pdf_omega;
            const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &pdf_omega);
            now_ray = Ray(hitpoint.position, dir);

            const double cos_term = dot(orienting_normal, dir);
            weight = multiply(weight, (now_object.color / kPI) * cos_term); // LambertianのBRDF = ρ/π と cos項
            pdf    *= pdf_omega;
        } break;
        // 完全鏡面
        case REFLECTION_TYPE_MIRROR: {
            // 完全鏡面なのでレイの反射方向は決定的。
            now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));
            weight = multiply(weight, now_object.color);
            pdf    *= 1.0;
        } break;
        // 屈折率Iorのガラス
        case REFLECTION_TYPE_GLASS: {
            const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか。

            Vec reflection_dir, refraction_dir;
            double fresnel_reflectance, fresnel_transmittance;
            if (check_refraction(
                into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
                &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                // 屈折 + 反射
                if (rnd->next01() < reflection_probability) { // 反射
                    now_ray = Ray(hitpoint.position, reflection_dir);
                    weight  = fresnel_reflectance * multiply(weight, now_object.color);
                    pdf    *= reflection_probability;
                } else { // 屈折
                    // レイの運ぶ放射輝度は屈折率の異なる物体間を移動するとき、屈折率の比の二乗の分だけ変化する。
                    const double nnt2 =
                        pow(into ?
                        refractive_index_of_vaccum / refractive_index_of_object :
                        refractive_index_of_object / refractive_index_of_vaccum, 2.0);
                    now_ray = Ray(hitpoint.position, refraction_dir);
                    weight  = nnt2 * fresnel_transmittance * multiply(weight, now_object.color);
                    pdf    *= 1.0 - reflection_probability;
                }
            } else {
                // 全反射
                now_ray = Ray(hitpoint.position, reflection_dir);
                weight  = multiply(weight, now_object.color);
                pdf    *= 1.0;
                break;
            }
        } break;
        }
    }
    return L;
}

};

#endif
```

sampling.h

- **各種サンプリングアルゴリズムを提供**
  - インポータンスサンプリング用の$\frac{\cos\theta}{\pi}$に従う半球サンプリング用関数等

```
#ifndef _SAMPLING_H_
#define _SAMPLING_H_

namespace edubpt {

// cosθ/πに比例
// 引数は両方正規化しておく
inline double sample_hemisphere_pdf_omega(const Vec &normal, const Vec &dir) {
    return std::max(dot(normal, dir), 0.0) / kPI;
}
Vec sample_hemisphere_cos_term(const Vec &normal, Random *rnd, double *pdf_omega) {
    Vec w = normal, u, v;
    if (fabs(w.x) > kEPS) // ベクトルwと直交するベクトルを作る。w.xが0に近い場合とそうでない場合とで使うベクトルを変える。
        u = normalize(cross(Vec(0.0, 1.0, 0.0), w));
    else
        u = normalize(cross(Vec(1.0, 0.0, 0.0), w));
    v = cross(w, u);
    // コサイン項を使った重点的サンプリング
    const double r1 = 2 * kPI * rnd->next01();
    const double r2 = rnd->next01(), r2s = sqrt(r2);
    Vec dir = normalize((
        u * cos(r1) * r2s +
        v * sin(r1) * r2s +
        w * sqrt(1.0 - r2)));

    *pdf_omega = sample_hemisphere_pdf_omega(normal, dir);

    return dir;
}

// 半径Rの球面上から一点をサンプリング
inline double sample_sphere_pdf_A(const double R) {
    return 1.0 / (4.0 * kPI * R * R);
}
Vec sample_sphere(const double R, Random *rnd, double *pdf_A) {
    const double z = rnd->next01() * 2.0 - 1.0;
    const double sz = sqrt(1 - z*z);
    const double phi = rnd->next01() * 2.0 * kPI;

    *pdf_A = sample_sphere_pdf_A(R);

    return R * Vec(sz * cos(phi), sz * sin(phi), z);
}

// ロシアンルーレットの確率計算関数
// 局所的な情報に基づいて決定する
double russian_roulette(const Sphere &sphere) {
    return sphere.emission.length_squared() > 0 ? 1.0 : std::max(sphere.color.x, std::max(sphere.color.y, sphere.color.z));
}

// ガラス面で、反射する確率 (1 - reflection_probabilityが屈折する確率)
const double reflection_probability = 0.5;

};

#endif
```

specular.h

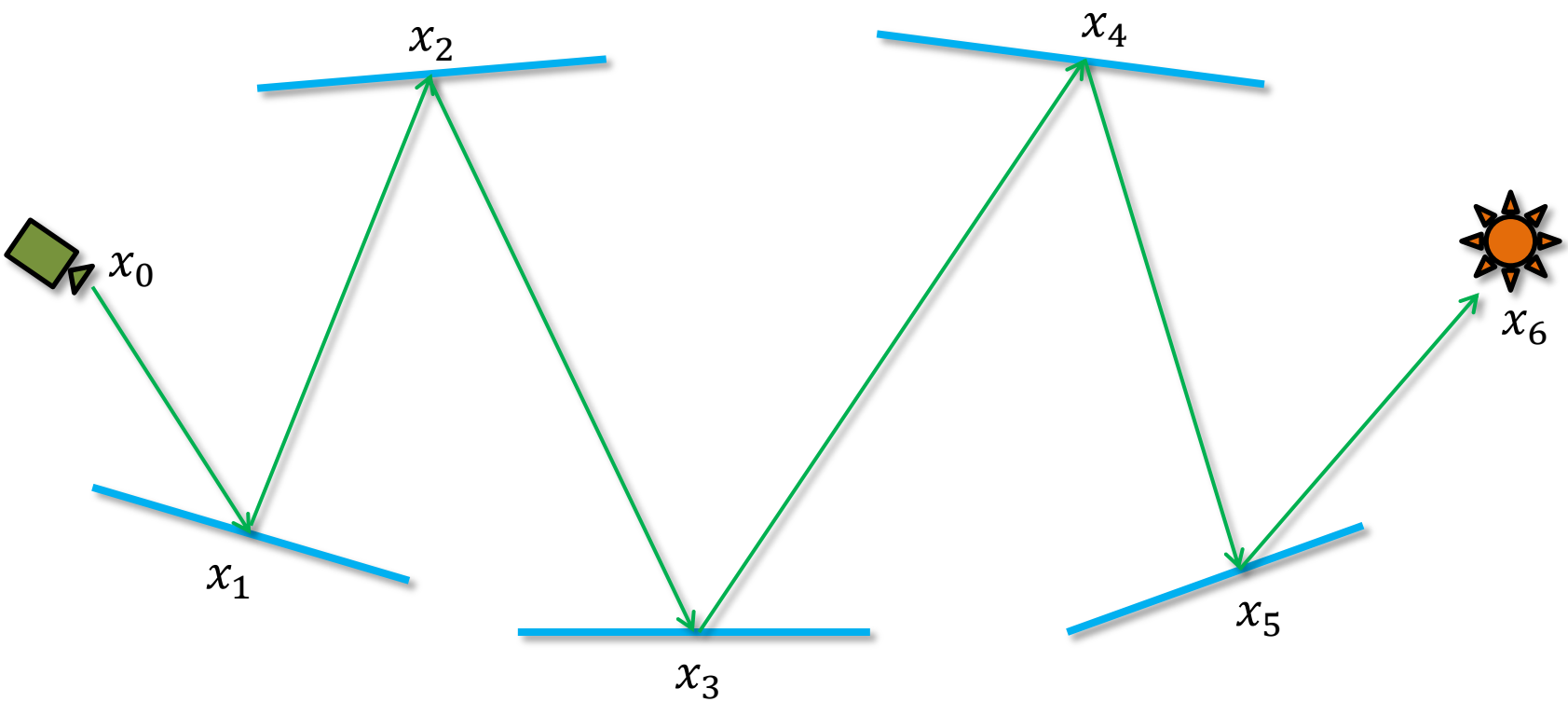
- 反射ベクトルの方向を得る関数や、屈折方向を得る関数を提供

```
#ifndef _SPECULAR_H_
#define _SPECULAR_H_

#include "material.h"
#include "vec.h"

namespace edubpt {

// 界面に向かう入射ベクトルがin
// 入射点における法線がnormal
// 返り値は入射点から出射する、反射ベクトル
Vec reflection_vector(const Vec &in, const Vec &normal) {
    return normalize(in - normal * 2.0 * dot(normal, in));
}

// falseなら全反射
// trueなら反射+屈折
bool check_refraction(
    const bool into,
    const Vec& position,
    const Vec &in,
    const Vec &normal,
    const Vec &orienting_normal,
    Vec *reflection_dir,
    Vec *refraction_dir,
    double *fresnel_reflectance,
    double *fresnel_transmittance) {
    *reflection_dir = reflection_vector(in, normal);

    // Snellの法則
    const double nnt = into ? refractive_index_of_vaccum / refractive_index_of_object : refractive_index_of_object / refractive_index_of_vaccum;
    const double ddn = dot(in, orienting_normal);
    const double cos2t = 1.0 - nnt * nnt * (1.0 - ddn * ddn);

    if (cos2t < 0.0) { // 全反射
        *refraction_dir = Vec();
        *fresnel_reflectance = 1.0;
        *fresnel_transmittance = 0.0;
        return false;
    }
    // 屈折の方向
    *refraction_dir =
        normalize(in * nnt - normal * (into ? 1.0 : -1.0) * (ddn * nnt + sqrt(cos2t)));

    // SchlickによるFresnelの反射係数の近似を使う
    const double a = refractive_index_of_object - refractive_index_of_vaccum;
    const double b = refractive_index_of_object + refractive_index_of_vaccum;
    const double R0 = (a * a) / (b * b);

    const double c = 1.0 - (into ? -ddn : dot(*refraction_dir, -1.0 * orienting_normal));
    *fresnel_reflectance = R0 + (1.0 - R0) * pow(c, 5.0); // 反射方向の光が反射してray.dirの方向に運ぶ割合。同時に屈折方向の光が反射する方向に運ぶ割合。
    *fresnel_transmittance = 1.0 - *fresnel_reflectance; // 屈折方向の光が屈折してray.dirの方向に運ぶ割合

    return true;
}

};

#endif
```

4. 双方向パストレーシング・理論編

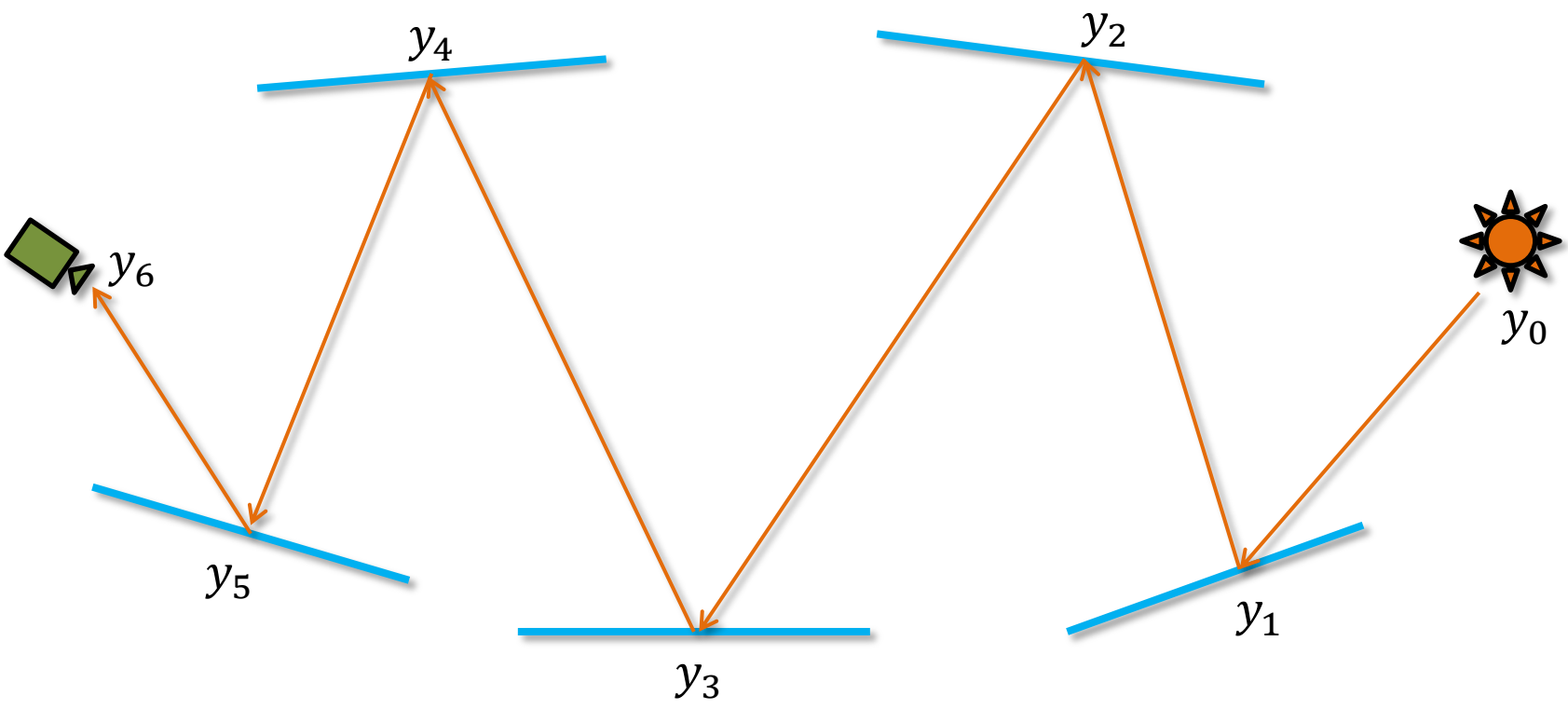
パストレーシング

- カメラからレイを飛ばし、光源に当たるまで反射・追跡を繰り返す。これによって、光源からカメラまでのパスを逆方向から得る。パスが得られれば、既に立式した積分に基づいて最終的な画像が計算できる。



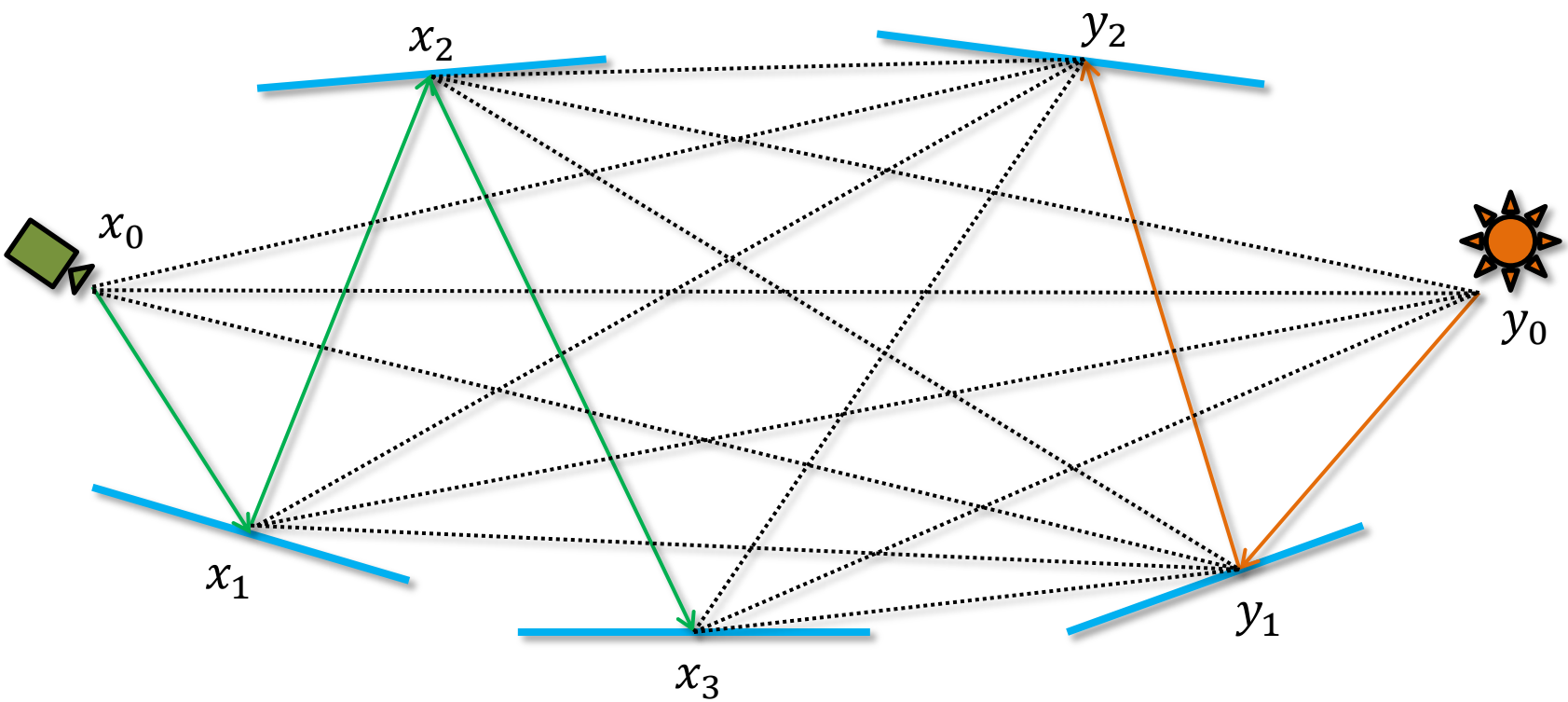
ライトトレーシング

- 光源からレイを飛ばし、カメラ（今回のカメラモデルではレンズ）に当たるまで反射・追跡を繰り返す。これによって、光源からカメラまでのパスを得る。パスが得られれば、既に立式した積分に基づいて最終的な画像が計算できる。



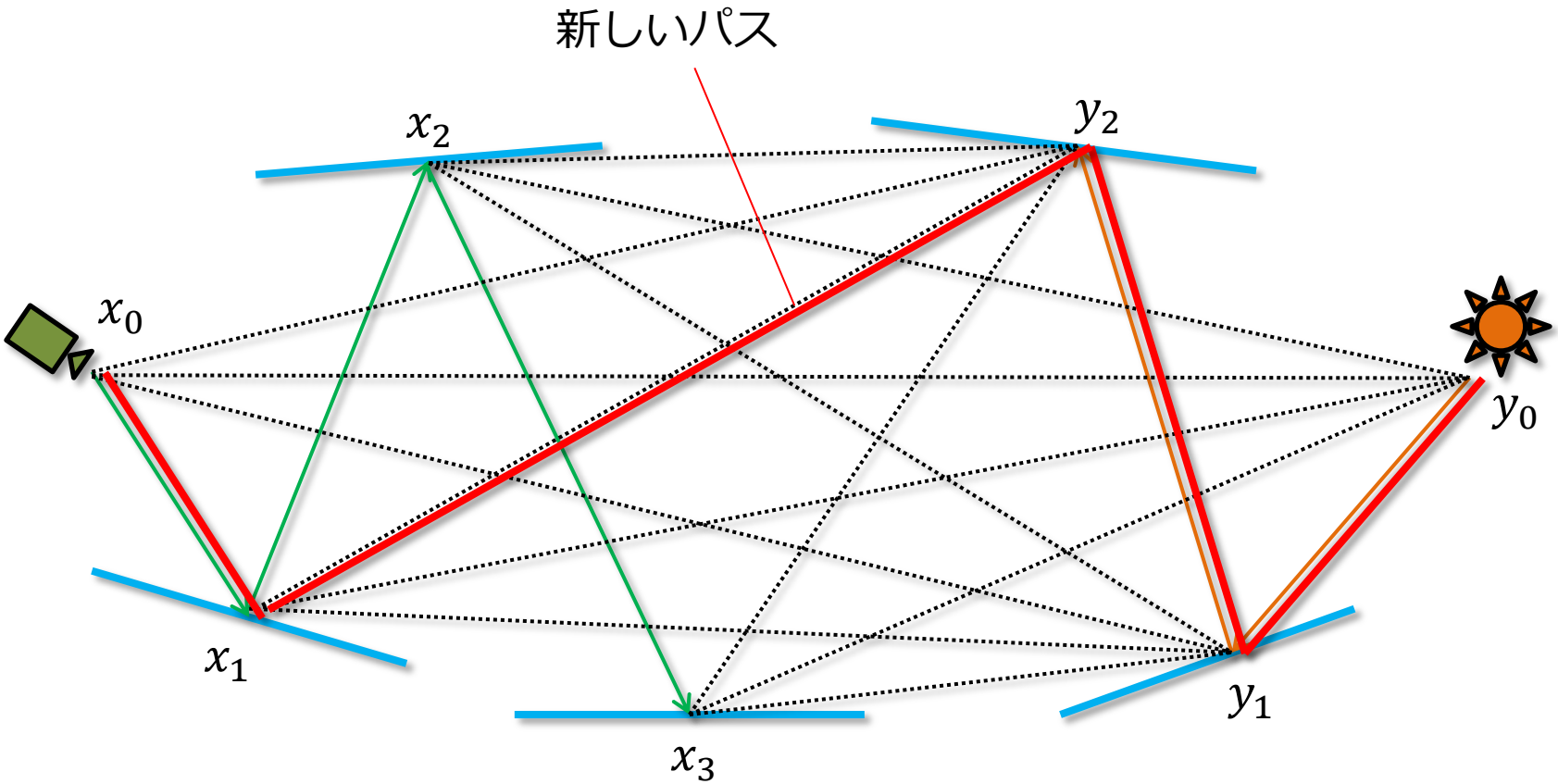
双方向パストレーシング

- カメラ側と光源側の両方からパスを追跡し、両者の結果を合わせることで片側から追跡しただけではサンプリングしにくいパスを得る。



双方向パストレーシング

- カメラ側と光源側の両方からパスを追跡し、両者の結果を合わせることで片側から追跡しただけではサンプリングしにくいパスを得る。



4.1 双方向パストレが解く式

双方向パストレが解く式

- 画素jの値について

$$I_j = \int_{I_j} \int_{Lens} W(x_I \leftarrow x_0) L(x_0 \leftarrow x_1) \frac{\cos^2 \theta}{r^2} dA(x_0)\, dA_{I_j}(x_I)$$

- であったが、双方向パストレーシングにおいてはこの式をさらに変形したものを使う。（ライトトレーシングによって得られたパスを使う際、都合が良くなるようにする）
- 具体的には、**変数x_Iの積分を変数x_1の積分に変数変換する。**
  - **これによって、シーン上の点を変数とした積分になる。**

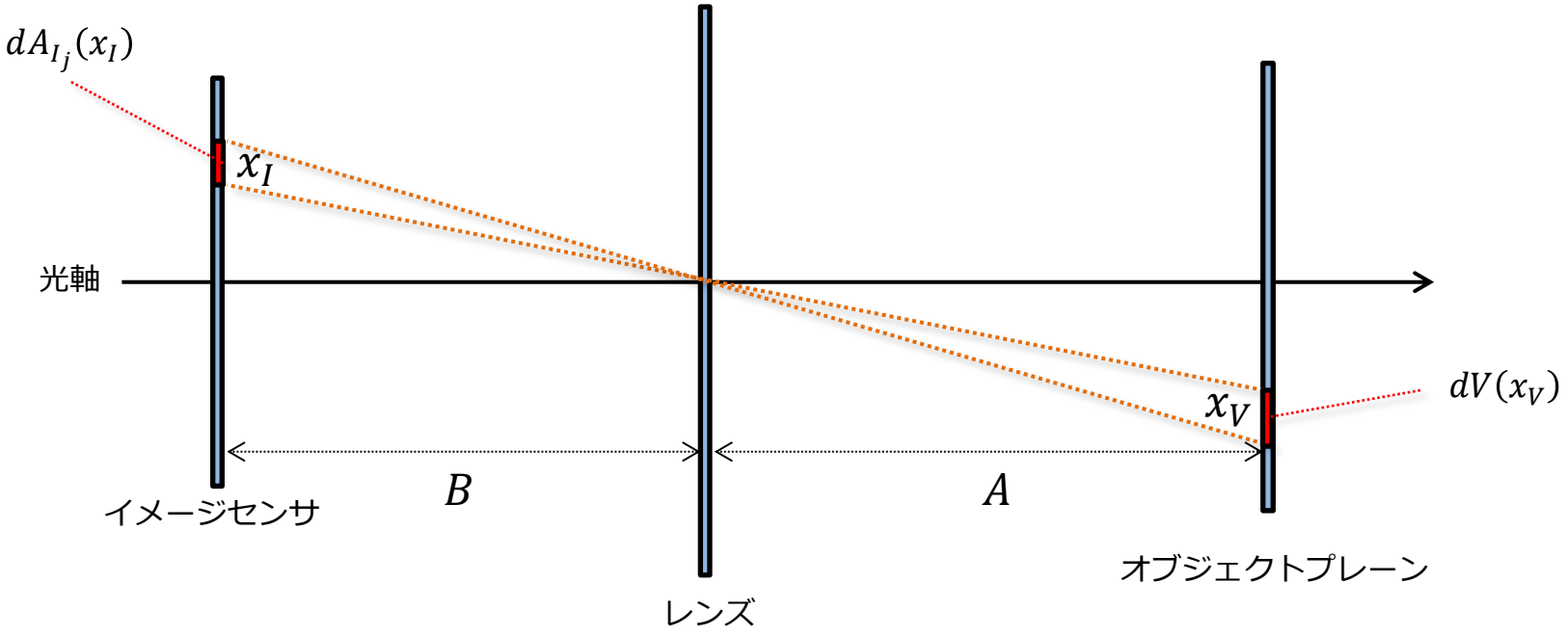
双方向パストレが解く式

- **イメージセンサ上の微小面積と、オブジェクトプレーン上の微小面積の関係式**
  - 相似関係が成り立つので、面積比は距離の二乗の比になる。

$$dA_{I_j}(x_I) : dV(x_V) = B^2 : A^2$$

  - よって、以下のようになる。

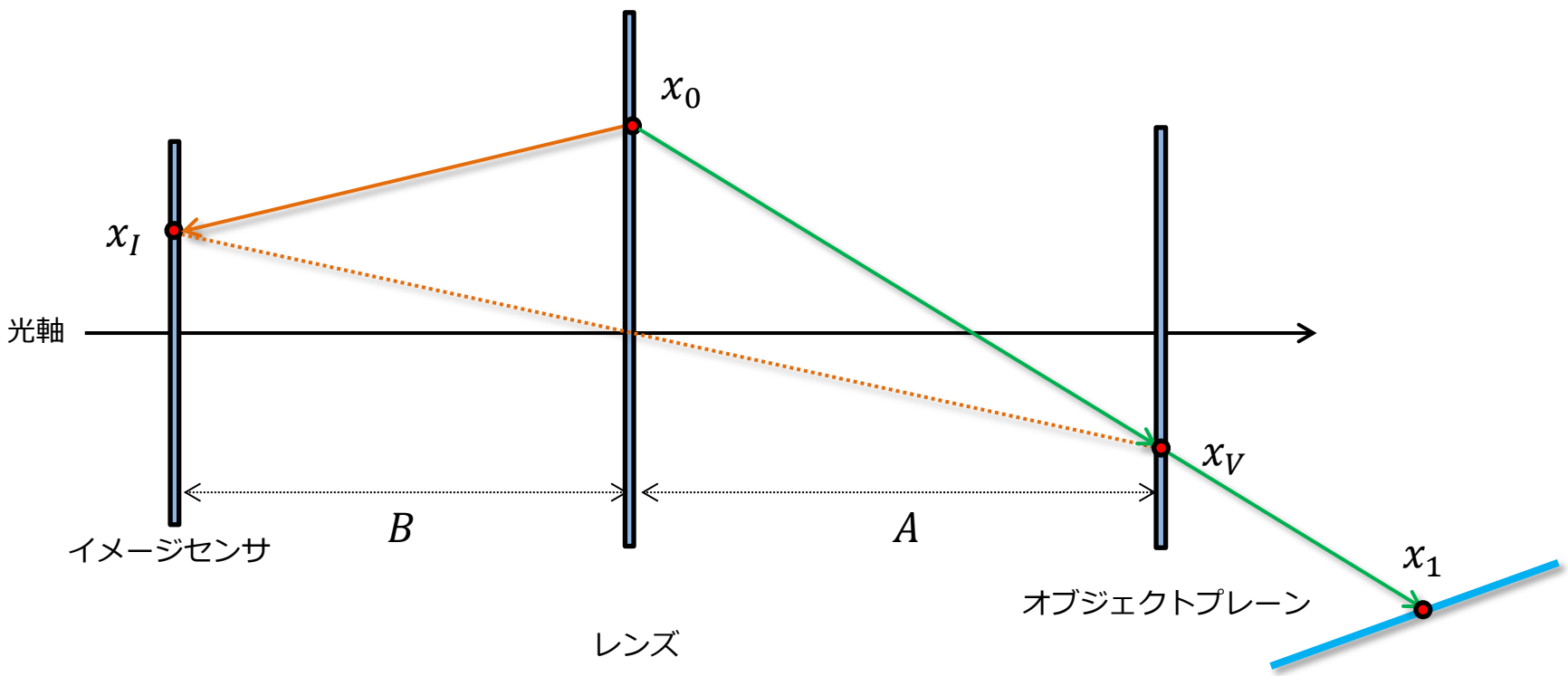
$$\boxed{dA_{I_j}(x_I) = \left(\frac{B}{A}\right)^2 dV(x_V)}$$



双方向パストレが解く式

- まず、イメージセンサ上の微小面積と、オブジェクトプレーン上の微小面積の関係式 $dA_{I_j}(x_I) = \left(\frac{B}{A}\right)^2 dV(x_V)$ を使い、変数x_Iの積分をx_Vの積分に変数変換する。
  - このとき、$W(x_I \leftarrow x_0) = W(x_0 \leftarrow x_V)$、$L(x_I \leftarrow x_0) = L(x_0 \leftarrow x_V)$という関係式を使う。
- 変換後の式

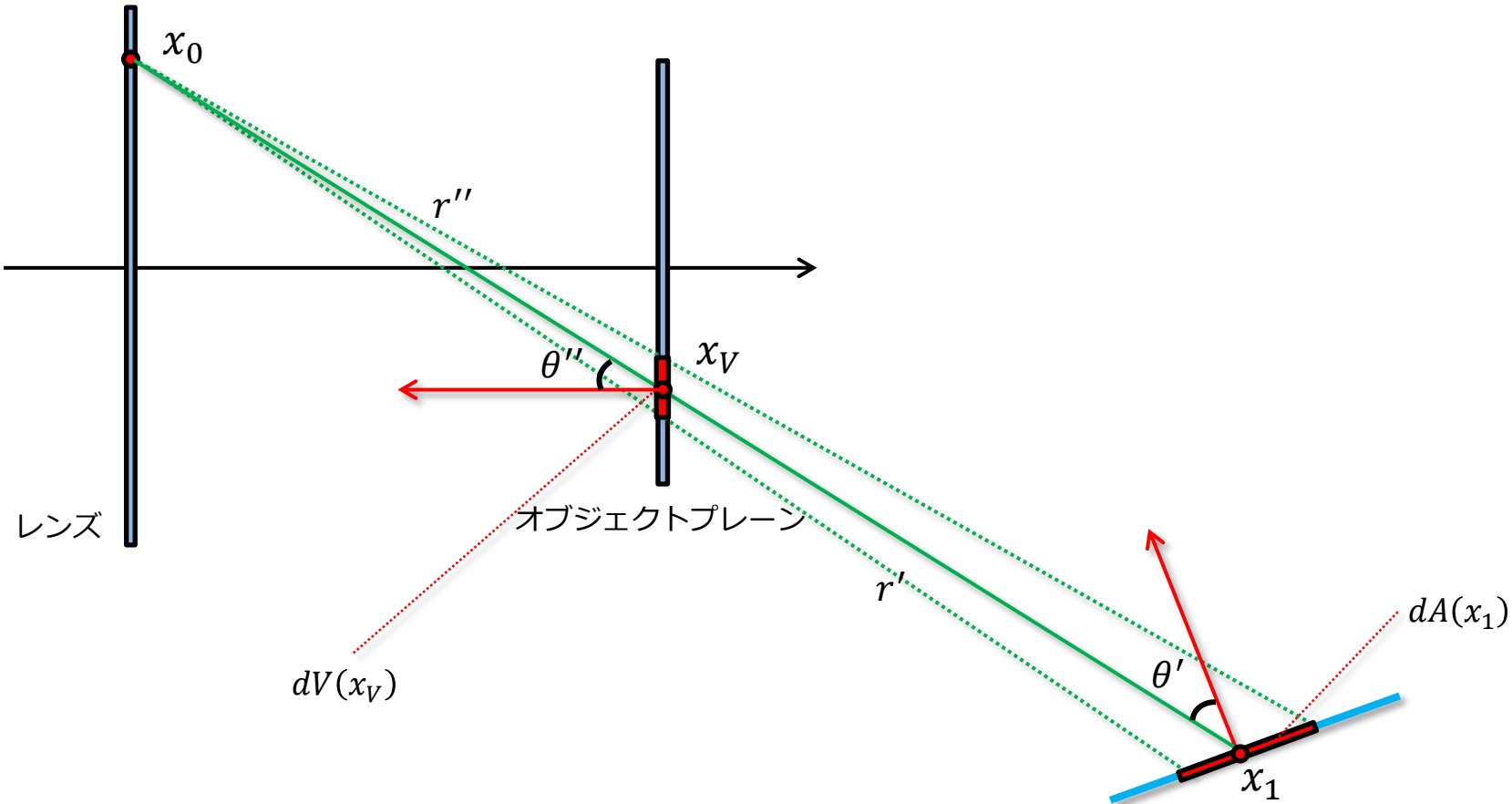
$$I_j = \int_V \int_{Lens} W(x_0 \leftarrow x_V) L(x_0 \leftarrow x_V) \frac{\cos^2\theta}{r^2} dA(x_0) \left(\frac{B}{A}\right)^2 dV\,(x_V)$$



双方向パストレが解く式

- **オブジェクトプレーン上の微小面積と、シーン上の微小面積の関係式**
  - x_0とx_Vの距離をr''、 x_0とx_1の距離をr'とする。
  - $dV(x_V)$を$\cos\theta''$で投影した面積と$dA(x_1)$を$\cos\theta'$で投影した面積の比が距離の二乗比になる。（相似関係）

$$dV(x_V)\cos\theta'' : dA(x_1)\cos\theta' = r''^2 : r'^2$$



双方向パストレが解く式

- **オブジェクトプレーン上の微小面積と、シーン上の微小面積の関係式**
  - 先の式より、

$$\boxed{dV(x_V) = \left(\frac{r''}{r'}\right)^2 \frac{\cos\theta'}{\cos\theta''} dA(x_1)}$$

- この関係式を使って変数x_Vの積分をx_1の積分に変数変換する。
  - このとき、$W(x_0 \leftarrow x_V) = W(x_0 \leftarrow x_1)$、$L(x_0 \leftarrow x_V) = L(x_0 \leftarrow x_1)$という関係式を使う。

$$I_j = \int_M \int_{Lens} W(x_0 \leftarrow x_1) L(x_0 \leftarrow x_1) \frac{\cos^2\theta}{r^2} dA(x_0) \left(\frac{B}{A}\right)^2 \left(\frac{r''}{r'}\right)^2 \frac{\cos\theta'}{\cos\theta''} dA(x_1)$$

- この式をジオメトリファクタ（以下、G項）を使って書きなおす。

$$G(x_0 \leftrightarrow x_1) = \frac{\cos\theta' \cos\theta''}{r'^2}$$

双方向パストレが解く式

- すると以下のようになる。

$$I_j = \int_M \int_{Lens} W(x_0 \leftarrow x_1) G(x_0 \leftrightarrow x_1) \left(\frac{Br'' \cos\theta}{Ar \cos\theta''} \right)^2 L(x_0 \leftarrow x_1) dA(x_0)\, dA(x_1)$$

- $W'(x_0 \leftarrow x_1) = W(x_0 \leftarrow x_1) \left(\frac{Br'' \cos\theta}{Ar \cos\theta''} \right)^2$ とすると以下のようになる。
  - シーンの面積に関する積分になった。
  - この式を双方向パストレーシングで解く。

$$\boxed{I_j = \int_M \int_{Lens} W'(x_0 \leftarrow x_1) G(x_0 \leftrightarrow x_1) L(x_0 \leftarrow x_1) dA(x_0)\, dA(x_1)}$$

式変形のまとめ

元の式

$$I_j = \int_{I_j}\int_{Lens} W(x_I \leftarrow x_0)L(x_0 \leftarrow x_1)\frac{\cos^2\theta}{r^2}dA(x_0)\,dA_{I_j}(x_I)$$

$$\boxed{dA_{I_j}(x_I) = \left(\frac{B}{A}\right)^2 dV(x_V)}$$

$$I_j = \int_V\int_{Lens} W(x_0 \leftarrow x_V)L(x_0 \leftarrow x_V)\frac{\cos^2\theta}{r^2}dA(x_0)\left(\frac{B}{A}\right)^2 dV\,(x_V)$$

$$\boxed{dV(x_V) = \left(\frac{r''}{r'}\right)^2 \frac{\cos\theta'}{\cos\theta''}dA(x_1)}$$

$$I_j = \int_M\int_{Lens} W(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)\left(\frac{Br''\cos\theta}{Ar\cos\theta''}\right)^2 L(x_0 \leftarrow x_1)dA(x_0)\,dA(x_1)$$

$$\boxed{W'(x_0 \leftarrow x_1) = W(x_0 \leftarrow x_1)\left(\frac{Br''\cos\theta}{Ar\cos\theta''}\right)^2}$$

最終的な式

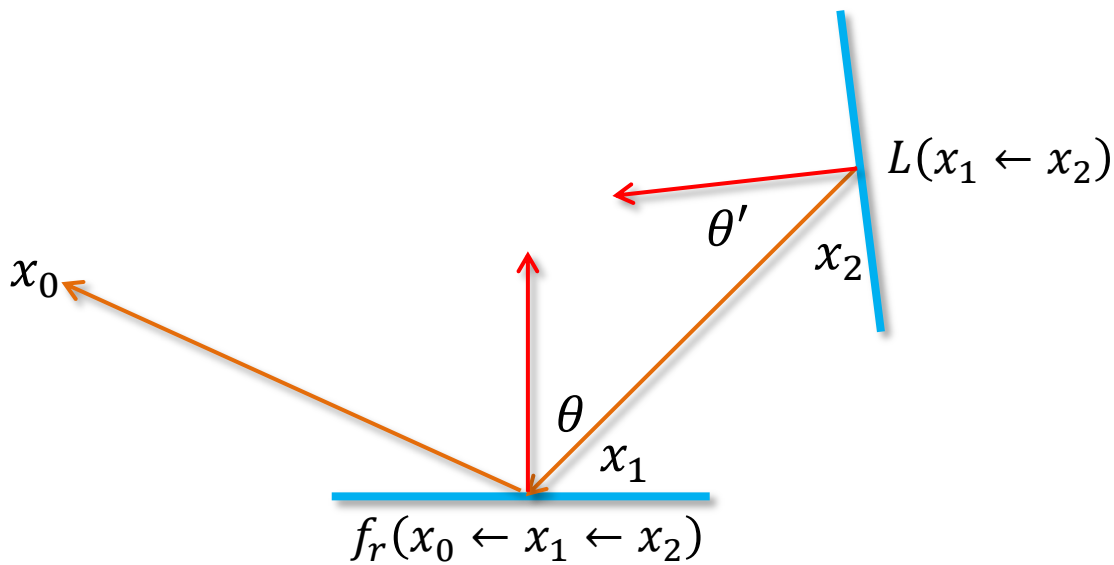
$$\boxed{I_j = \int_M\int_{Lens} W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)L(x_0 \leftarrow x_1)dA(x_0)\,dA(x_1)}$$

4.2 レンダリング方程式

レンダリング方程式

- $L(x_0 \leftarrow x_1)$はレンダリング方程式を使って再帰的に定義される。

$$L(x_0 \leftarrow x_1) = L_e(x_0 \leftarrow x_1) + \int_\Omega L(x_1 \leftarrow x_2) \underbrace{f_r(x_0 \leftarrow x_1 \leftarrow x_2)}_{\text{BRDF}} \cos\theta \, d\sigma(x_1 \leftarrow x_2)$$

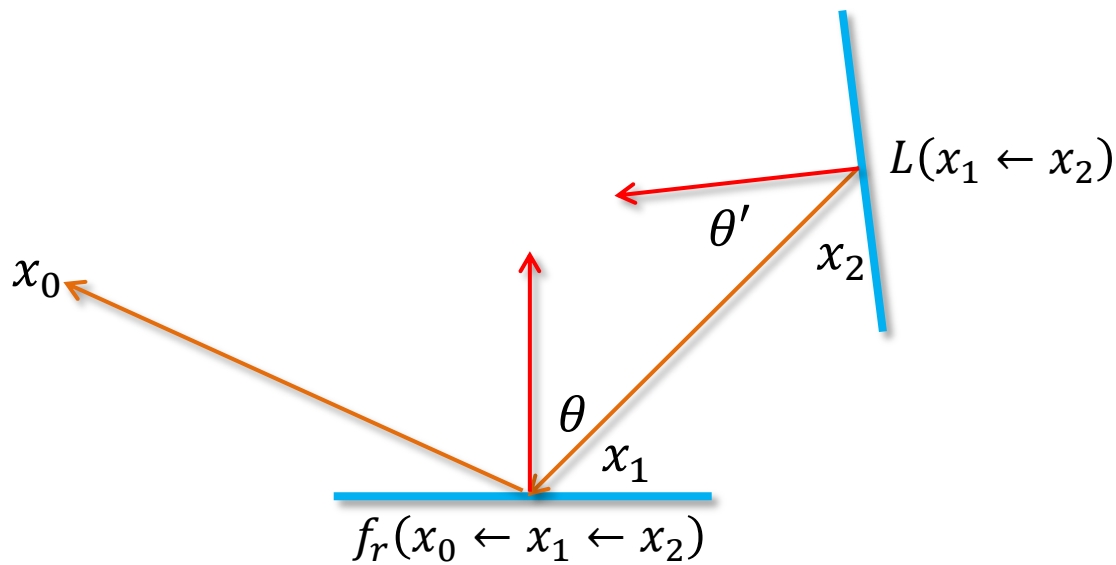


レンダリング方程式

- これは立体角に関する半球積分だが、シーン上の点を変数とする、面積に関する積分に変換する。（積分範囲はシーン表面M）
  - I_jの積分を変数変換してシーン上の点を変数とする面積分にしたのと同じ理由。
  - 例によって微小立体角と微小面積の関係式$d\sigma(x_1 \leftarrow x_2) = \frac{\cos\theta'}{r^2} dA(x_2)$を使う。
  - さらに$G(x_1 \leftrightarrow x_2) = \frac{\cos\theta \cos\theta'}{r^2}$も使う。

$$L(x_0 \leftarrow x_1) = L_e(x_0 \leftarrow x_1) + \int_\Omega L(x_1 \leftarrow x_2) f_r(x_0 \leftarrow x_1 \leftarrow x_2) \cos\theta \, d\sigma(x_1 \leftarrow x_2)$$

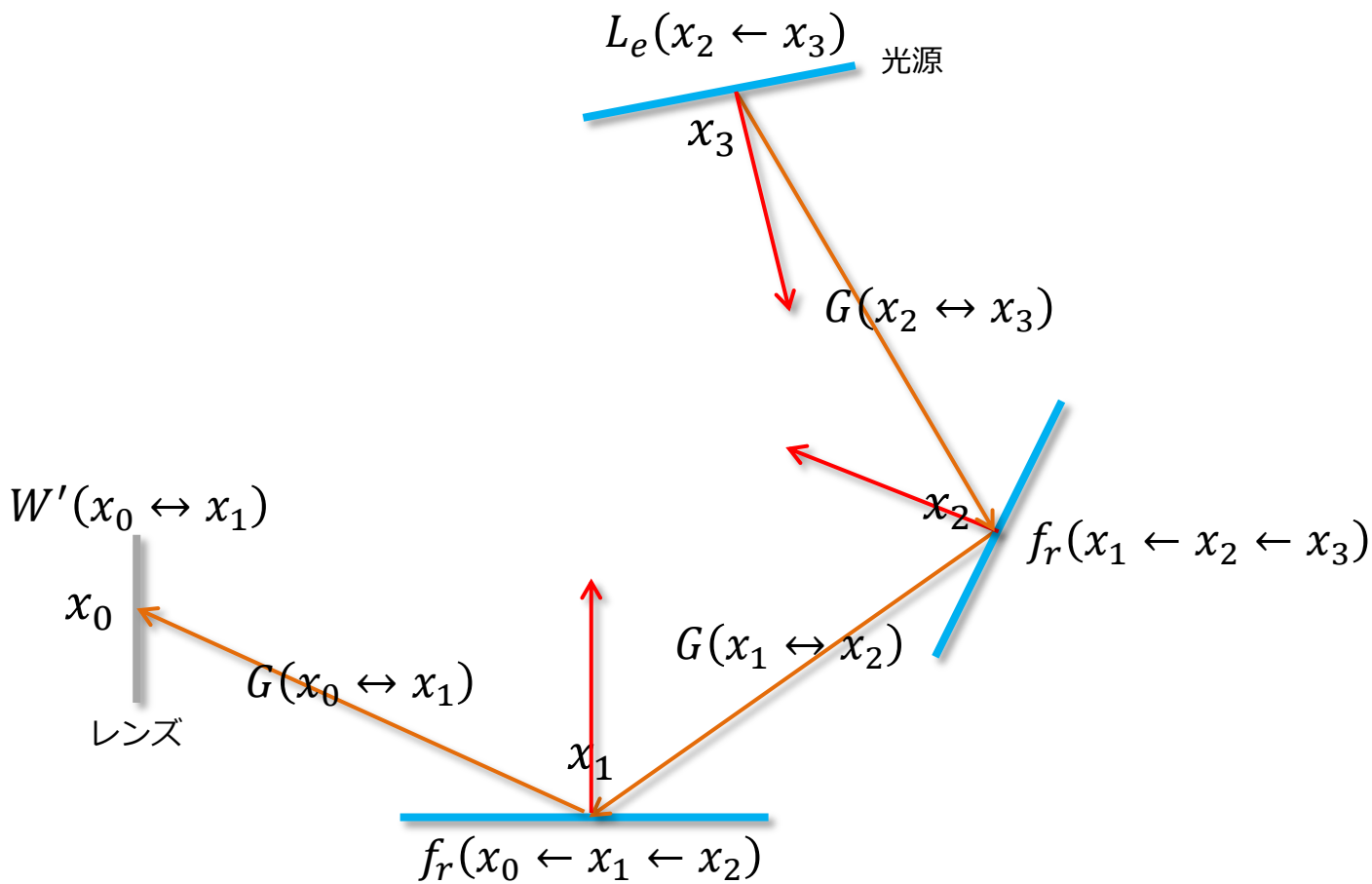
$$= L_e(x_0 \leftarrow x_1) + \int_M L(x_1 \leftarrow x_2) f_r(x_0 \leftarrow x_1 \leftarrow x_2) G(x_1 \leftrightarrow x_2) dA(x_2)$$



レンダリング方程式

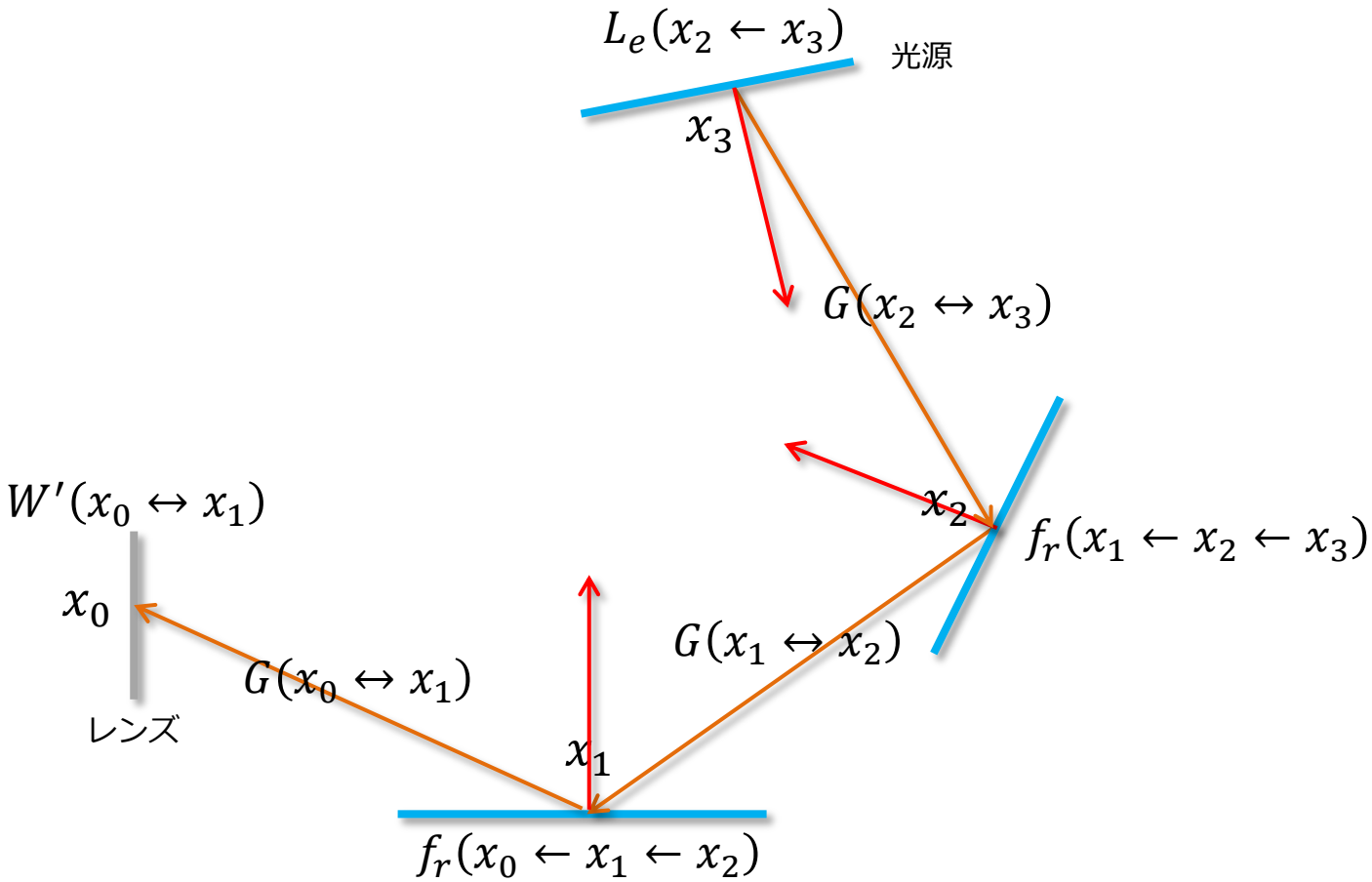
- すると、解く対象の積分は以下のように展開される。

$$
\begin{aligned}
I_j = &\int_M \int_{Lens} W'(x_0 \leftarrow x_1) G(x_0 \leftrightarrow x_1) L_e(x_0 \leftarrow x_1) dA(x_0)\, dA(x_1) \\
&+ \int_M \int_M \int_{Lens} W'(x_0 \leftarrow x_1) G(x_0 \leftrightarrow x_1) f_r(x_0 \leftarrow x_1 \leftarrow x_2) G(x_1 \leftrightarrow x_2) L_e(x_1 \leftarrow x_2) dA(x_0)\, dA(x_1)\, dA(x_2) \\
&+ \int_M \int_M \int_M \int_{Lens} W'(x_0 \leftarrow x_1) G(x_0 \leftrightarrow x_1) f_r(x_0 \leftarrow x_1 \leftarrow x_2) G(x_1 \leftrightarrow x_2) f_r(x_1 \leftarrow x_2 \leftarrow x_3) G(x_2 \leftrightarrow x_3) L_e(x_2 \leftarrow x_3) dA(x_0)\, dA(x_1)\, dA(x_2)\, dA(x_3) \\
&+ \cdots
\end{aligned}
$$



レンダリング方程式

- つまり、光源からレンズまでのパスの頂点の数の分だけ、独立した重積分が得られ、その和が最終的な画素値I_jになる、ということ。



モンテカルロ積分

- そのような重積分一つ一つを、モンテカルロ積分で解くことが出来る。
  - シーン上の頂点によって構成されるパスが、モンテカルロ積分に対するサンプルとして必要になる。
  - そのようなパスは、例えばパストレーシングなどで得ることが出来る。
  - 双方向パストレーシングにおいては、**カメラ側からのトレーシングと光源側からのトレーシングによって得られた頂点集合を組み合わせることで様々な頂点数の複数のパスを生成し、それらのパスを各モンテカルロ積分の計算に使用する。**

$$\widehat{I_j} = \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)L_e(x_0 \leftarrow x_1)}{P_A(x_0)P_A(x_1)} + \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)L_e(x_1 \leftarrow x_2)}{P_A(x_0)P_A(x_1)P_A(x_2)} + \cdots$$

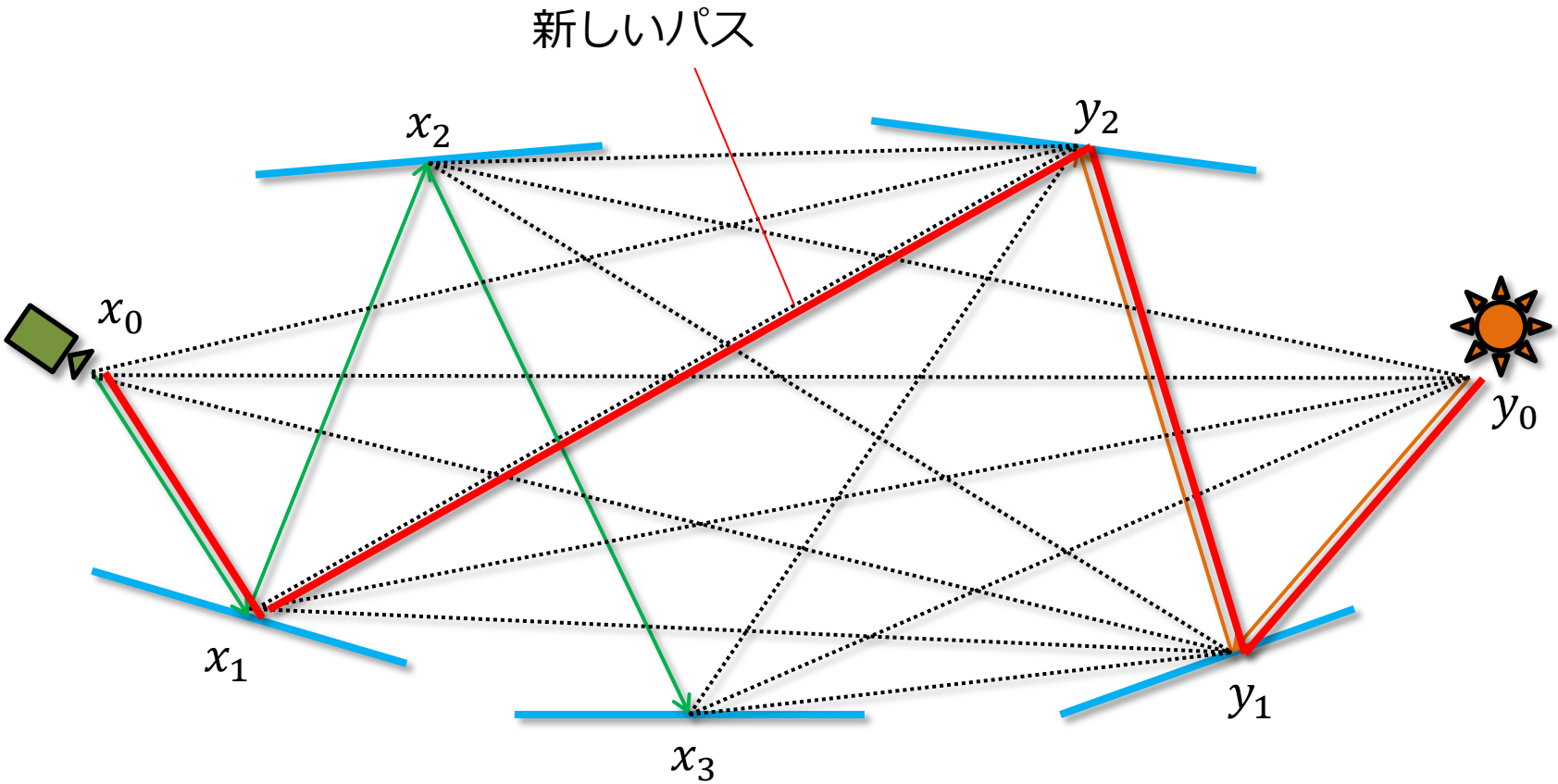
$$\Updownarrow$$

$\widehat{I_j}$ = 頂点数２の積分＋頂点数３の積分＋頂点数４の積分＋ …

5. 双方向パストレーシング・実装編その１

双方向パストレーシング・実装編

- **カメラ側からパストレーシングしてサンプリングされた頂点集合と、光源側からライトトレーシングしてサンプリングされた頂点集合をマージして一つのパスにする。**
  - そのやり方について、実装を並べつつ解説。



最終的に必要なもの

- 最終的に以下のようなモンテカルロ積分の和を計算しなければならない

$$
\begin{aligned}
\widehat{I_j} = & \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)L_e(x_0 \leftarrow x_1)}{P_A(x_0)P_A(x_1)} \\
& + \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)L_e(x_1 \leftarrow x_2)}{P_A(x_0)P_A(x_1)P_A(x_2)} \\
& + \cdots
\end{aligned}
$$

- そこで、各頂点ごとの面積測度に関する確率密度（分母）とモンテカルロスループット（分子）を求める必要がある。
  - カメラ側からのパス、光源側からのパス、両方についてスループットと確率密度を求める。
  - カメラ側のパスと光源側のパスをつなげるときは、それぞれの確率密度とスループットを乗算していくことになる。
- この結果が、上の式のモンテカルロ積分の和を計算するための一つ一つの項の推定値になる。（上の式は頂点数ごとに別々のモンテカルロ積分の和になっており、その各項）

最終的に必要なもの

頂点x_1とx_2を新しくつないだので、改めてG項とBRDFを計算

$$\widehat{I_j} = \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)L_e(x_0 \leftarrow x_1)}{P_A(x_0)P_A(x_1)}$$
$$+ \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)L_e(x_1 \leftarrow x_2)}{P_A(x_0)P_A(x_1)P_A(x_2)}$$
$$+ \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)f_r(x_1 \leftarrow x_2 \leftarrow x_3)G(x_2 \leftrightarrow x_3)L_e(x_2 \leftarrow x_3)}{P_A(x_0)P_A(x_1)P_A(x_2)P_A(x_3)}$$
$$+ \cdots$$

パストレーシングによってサンプリングされたパスx_0x_1のスループットと確率密度

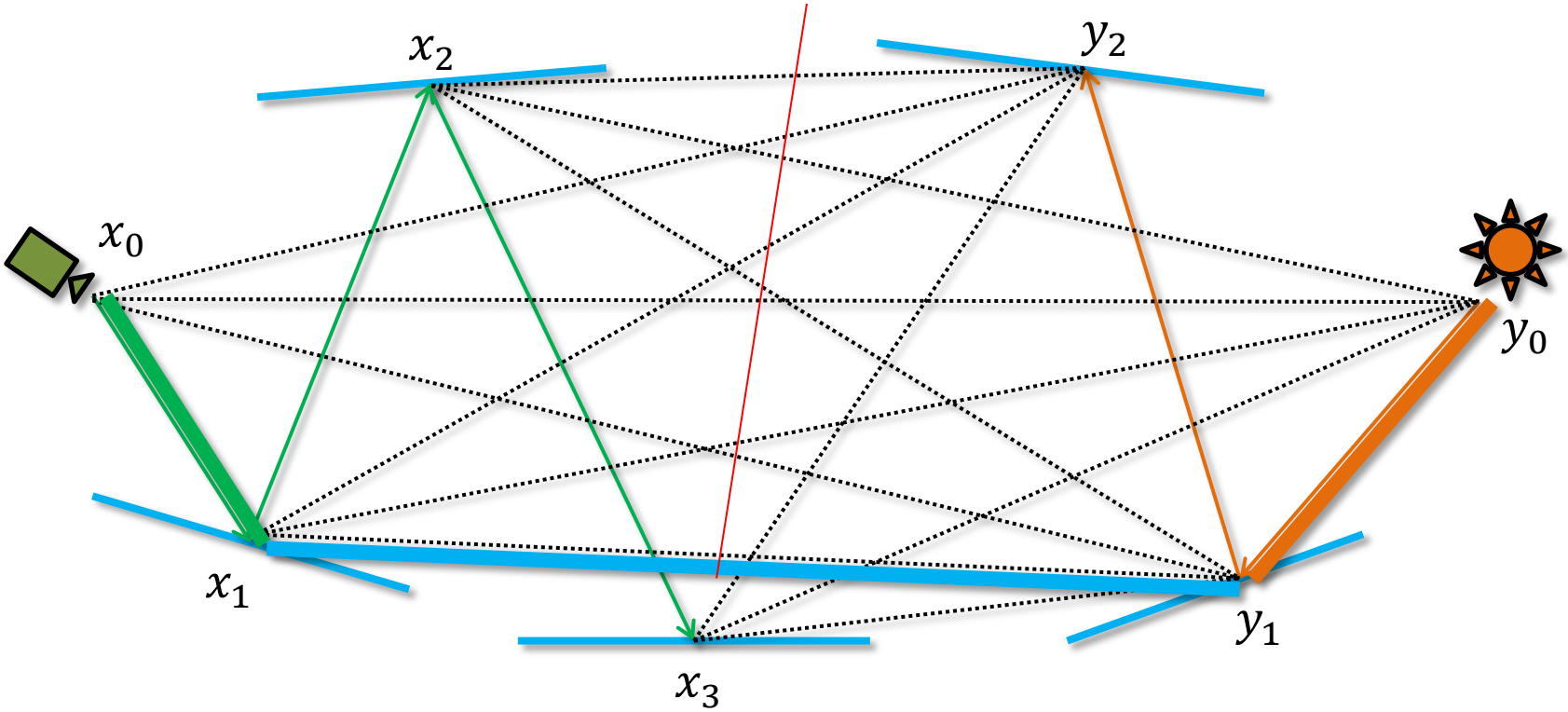
ライトトレーシングによってサンプリングされたパスx_2x_3のスループットと確率密度

- 以上のような計算を行うために、前半部分をパストレーシング、後半部分をライトトレーシングによって得るわけである。

最終的に必要なもの

$$
\begin{aligned}
\widehat{I_j} = & \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)L_e(x_0 \leftarrow x_1)}{P_A(x_0)P_A(x_1)} \\
& + \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)L_e(x_1 \leftarrow x_2)}{P_A(x_0)P_A(x_1)P_A(x_2)} \\
& + \frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)f_r(x_1 \leftarrow x_2 \leftarrow x_3)G(x_2 \leftrightarrow x_3)L_e(x_2 \leftarrow x_3)}{P_A(x_0)P_A(x_1)P_A(x_2)P_A(x_3)} \\
& + \cdots
\end{aligned}
$$

上の色を付けた項と対応するパス



5.1 パストレーシングによるパスのサンプリング

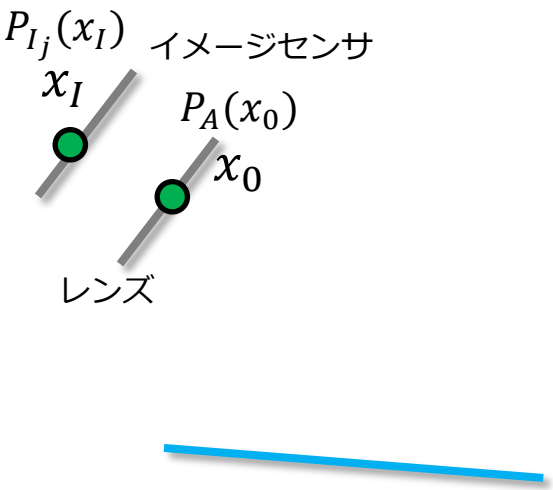
pathtracing.h

- パストレーシングによる頂点のサンプリングを行う
  - generate_vertices_by_pathtracing ()

```
#ifndef _PATHTRACING_H_
#define _PATHTRACING_H_

#include <vector>

#include "random.h"
#include "scene.h"
#include "vector.h"
#include "camera.h"
#include "specular.h"
#include "sampling.h"
#include "camera.h"
#include "vertex.h"

namespace edubpt {


struct PathtracingResult {
    Color value;
    int imagebuffer_x, imagebuffer_y;
    bool is_light_hit;

    Path(PathtracingResult const &value, const int image_x, const int image_y, const bool is_light_hit) :
    value(value), imagebuffer_x(imagebuffer_x), imagebuffer_y(imagebuffer_y), is_light_hit(is_light_hit) {}
};

PathtracingResult generate_vertices_by_pathtracing(const Camera &camera, const int imagebuffer_x, const int imagebuffer_y, Random *rnd, std::vector<Vertex> &vertices) {
    Vec position_on_imagesensor, position_on_objectplane, position_on_lens;
    // イメージセンサとレンズから一様ランダムに頂点をサンプリング。
    double P_Image, P_lens;
    camera.sample_points(imagebuffer_x, imagebuffer_y, rnd, &position_on_imagesensor, &position_on_objectplane, &position_on_lens, &P_Image, &P_lens);

    double total_pdf_A = P_lens;
    Color MC_throughput(1.0, 1.0, 1.0); // PathTracingの場合、これは常に1.0で固定となり、最後に光源からの寄与と掛算することでイメージセンサへの寄与を計算することになる。

    // レンズ上の頂点（LE）を頂点リストに追加。
    vertices.push_back(Vertex(position_on_lens, camera.normal_on_lens, camera.normal_on_lens, -1, Vertex::OBJECT_TYPE_LENS, total_pdf_A, MC_throughput));

    Ray new_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
    double now_sampled_pdf_omega = 1.0;
    Vec previous_normal = camera.normal_on_lens;

    for (int now_vertex_index = 1; ++now_vertex_index) {
        Intersection intersection;
        if (!intersect_scene(new_ray, &intersection))
            break;

        const Sphere &now_object = spheres[intersection.object_id];
        const HitPoint &hitpoint = intersection.hitpoint;
        // 交差位置の法線（物体表面からのベクトルを考慮）
        const Vec orienting_normal = dot(hitpoint.normal, -now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal);
        const double russian_roulette_probability = russian_roulette(now_object);

        // ロシアンルーレットによって、新しい頂点を（実際に）サンプリングし、生成するのかどうかを決定する。
        if (rnd->next01() >= russian_roulette_probability) {
            break;
        }
        // 新しい頂点がサンプリングされたので、トータルの幾何学項を掛算する
        total_pdf_A *= russian_roulette_probability;

        const Vec to_next_vertex = now_ray.org - hitpoint.position;
        if (now_vertex_index == 1)
        {
            // eからサンプリング幾何学項をイメージセンサとのサンプリング幾何学項を変換することで求める。
            const Vec wo_x2 = position_on_imagesensor - position_on_lens;
            const Vec wi_x1 = position_on_objectplane - position_on_lens;
            const Vec wo_x1 = hitpoint.position - position_on_lens;
            const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, wo_x1, wo_x2, orienting_normal);
            total_pdf_A *= PA_x1;

            // そうして、MC_throughputは計算の他と同じだが、たった今1つの頂点。
            MC_throughput = camera.W_dash(wo_x1, wo_x2, wi_x1) * MC_throughput;
        } else {
            // 新しい頂点をサンプリングするための幾何学項変換はこの角角度に変するものであったため、これを幾何学項に変する幾何学項変換に変換する。
            const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(to_next_vertex), orienting_normal) / to_next_vertex.length_squared());
            // 全ての頂点をサンプリングする幾何学項の総計を加算。
            total_pdf_A *= now_sampled_pdf_A;
        }
        // ジオメトリターム（G項）
        const double G = dot(normalize(to_next_vertex), orienting_normal) * dot(normalize(-1.0 * to_next_vertex), previous_normal) / to_next_vertex.length_squared();
        MC_throughput = G * MC_throughput;

        // 光源にたどりたどるまで追跡終了。（光源の放射率が1で固定であることを前提としており、これ以上の追跡はMC_throughputが0になるため、光源にも放射率があるならこの限りではない）
        if (now_object.emission.length_squared() > 0) {
            vertices.push_back(Vertex(hitpoint.position, orienting_normal, hitpoint.normal, intersection.object_id, Vertex::OBJECT_TYPE_LIGHT, total_pdf_A, MC_throughput))
            const Color result = multiply(MC_throughput, now_object.emission) / total_pdf_A;
            return PathtracingResult(result, imagebuffer_x, imagebuffer_y, true);
        }

        // 新しい頂点を頂点リストに追加する。
        vertices.push_back(Vertex(hitpoint.position, orienting_normal, hitpoint.normal, intersection.object_id,
                now_object.reflection_type == REFLECTION_TYPE_DIFFUSE ? Vertex::OBJECT_TYPE_DIFFUSE :Vertex::OBJECT_TYPE_SPECULAR,
                total_pdf_A, MC_throughput));



        // 反射による処理
        // 次の頂点をサンプリングするために、新しいレイの方向を決める。
        switch (now_object.reflection_type) {
        // 完全拡散面
        case REFLECTION_TYPE_DIFFUSE: {
            const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &now_sampled_pdf_omega);
            now_ray = Ray(hitpoint.position, dir);
            MC_throughput = multiply(now_object.color / NPI, MC_throughput);
        } break;
        // 完全鏡面
        case REFLECTION_TYPE_MIRROR: {
            // 完全鏡面なのでレイの反射方向は決定的。
            // ジャコビアン含むとサンプリング幾何学項はdω/dωの確率の所有となるが、分母とキャンセルされるため、値のみ乗ける。
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));

            // スループット調整
            // sample_pdf_omegaのdω/dωの確率とキャンセルするため、値のみ乗ける。 ここのあたりはLightTracingと同様。
            // BRDFの場合は乗算
            MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        } break;
        // ガラス
        case REFLECTION_TYPE_GLASS: {
            const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか。

            Vec reflection_dir, refraction_dir;
            double fresnel_reflectance, fresnel_transmittance;
            if (calc_fresnel(
                    into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance) {
                // 屈折 + 反射
                if (rnd->next01() < reflection_probability) { // 反射
                    now_sampled_pdf_omega = 1.0;
                    now_ray = Ray(hitpoint.position, reflection_dir);
                    MC_throughput = fresnel_reflectance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
                    total_pdf_A *= reflection_probability;
                } else {
                    const double nnt2 =
                        pow(into ?
                            refractive_index_of_vacuum / refractive_index_of_object :
                            refractive_index_of_object / refractive_index_of_vacuum, 2.0);
                    now_sampled_pdf_omega = 1.0;
                    now_ray = Ray(hitpoint.position, refraction_dir);
                    MC_throughput = nnt2 * fresnel_transmittance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
                    total_pdf_A *= 1.0 - reflection_probability;
                }
            } else {
                // 全反射
                now_sampled_pdf_omega = 1.0;
                now_ray = Ray(hitpoint.position, reflection_dir);
                MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
                break;
            }
        } break;
        }

        previous_normal = orienting_normal;
    }

    return PathtracingResult(Color(), 0, 0, false);
}

};

#endif
```

パストレーシングによるサンプリングの例

- まず、レンズ上x_0とイメージセンサ上x_Iにサンプルを生成する。この時、各頂点のサンプリング確率密度も一緒に求める。
  - これが$P_A(x_0)$になる。ソースコード中ではP_lensである。
- シーンをトレースし、次々に頂点をサンプリングしていき、各頂点をリストに追加していく。
  - 頂点x_Nを追加するとき、Nまでの確率密度の総計（$\prod_{i=0}^{N} P_A(x_i)$）とモンテカルロスループットの総計（G項とBRDFの積）が必要になるので、順次計算しておく。
  - それぞれソースコード中ではtotal_pdf_AとMC_throughputという変数に格納しておく。



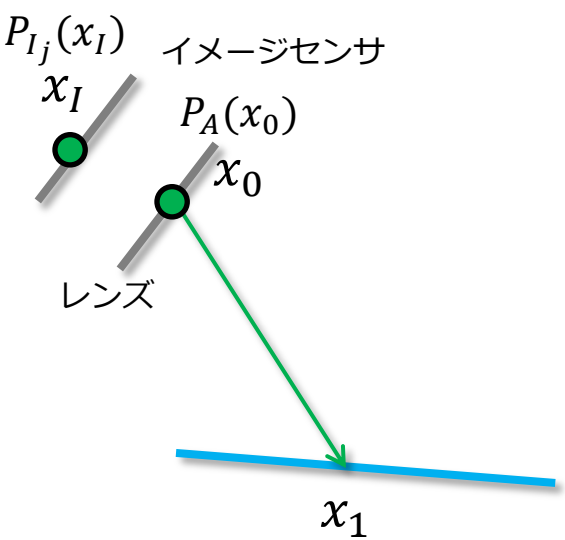

```
PathtracingResult generate_vertices_by_pathtracing(const Camera &camera, const int imagebuffer_x, const int imagebuffer_y, Random *rnd, std::vector<Vertex> &vertices) {
    Vec position_on_imagesensor, position_on_objectplane, position_on_lens;
    // イメージセンサとレンズ上から最初の点をサンプリング
    double P_Image, P_lens;
    camera.sample_points(imagebuffer_x, imagebuffer_y, rnd, &position_on_imagesensor, &position_on_objectplane, &position_on_lens, &P_Image, &P_lens);

    double total_pdf_A = P_lens;
    Color MC_throughput(1, 1, 1); // PTの場合、これはインポータンス値（brdfとかのGとかの積）

    // レンズ上の頂点（x0）を頂点リストに追加
    vertices.push_back(Vertex(position_on_lens, camera.normal_on_lens, camera.normal_on_lens, -1, Vertex::OBJECT_TYPE_LENS, total_pdf_A, MC_throughput));

    Ray now_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
    double now_sampled_pdf_omega = 1.0;
    Vec previous_normal = camera.normal_on_lens;
```

パストレーシングによるサンプリングの例

- 既に見たように、 x_0とx_Iが決まると、シーンをレイトレーシングすることで最初のシーン上の頂点x_1をサンプリングできる。
  - レンズ上の点から、対応するオブジェクトプレーン上の点の方向にレイを飛ばす。
- $P_A(x_1)$を計算する必要があるが、これには以下の関係式を使う。

$$dA_{I_j}(x_I) = \left(\frac{B}{A}\right)^2 dV(x_V)$$

$$dV(x_V) = \left(\frac{r''}{r'}\right)^2 \frac{\cos\theta'}{\cos\theta''} dA(x_1)$$

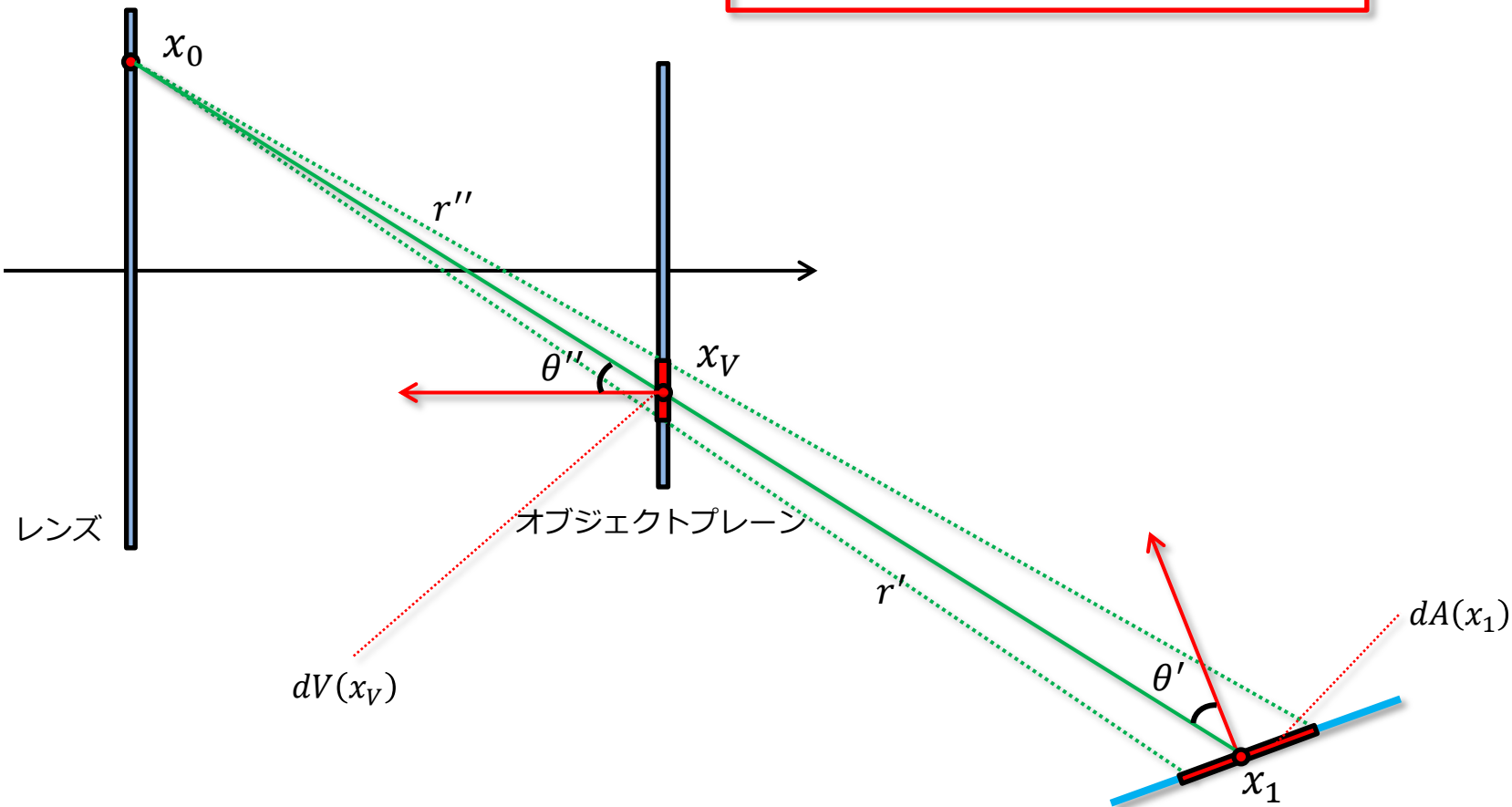



```
Ray now_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
double now_sampled_pdf_omega = 1.0;
Vec previous_normal = camera.normal_on_lens;

for (int now_vertex_id = 1;; ++now_vertex_id) {
    Intersection intersection;
    if (!intersect_scene(now_ray, &intersection))
        break;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal); // 交差位置の法線（物体からのレイの入出を考慮）
    const double russian_roulette_probability = russian_roulette(now_object);

    // ロシアンルーレット
    if (rnd->next01() >= russian_roulette_probability) {
        break;
    }
    // 新しい頂点がサンプリングされた
    total_pdf_A *= russian_roulette_probability;

    const Vec between = now_ray.org - hitpoint.position;
    if (now_vertex_id == 1)
    {
        // x1だけは確率密度の計算が特殊になる
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = hitpoint.position - position_on_lens;
        const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, orienting_normal);
        total_pdf_A *= PA_x1;

        // さらに、MC_throughputの計算も若干特殊になる（カメラ周りの処理）
        MC_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput;
    } else {
        // 新しい頂点を得たので、今までの頂点をサンプリングする確率の総計を出す
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(between), orienting_normal) / between.length_squared());
        total_pdf_A *= now_sampled_pdf_A;
    }
```

パストレーシングによるサンプリングの例

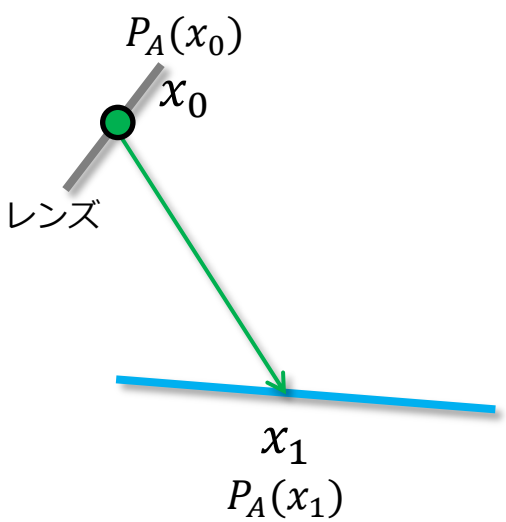
- $dA(a) = CdB(b) \Leftrightarrow CP_A(a) = P_B(b)$　という関係より、以下の式が成り立つ。この式を使ってイメージセンサ上の確率密度$P_{I_j}(x_I)$をシーンの面積測度に関する確率密度$P_A(x_1)$に変換する。

$$\boxed{P_A(x_1) = \left(\frac{B}{A}\right)^2 \left(\frac{r''}{r'}\right)^2 \frac{\cos\theta'}{\cos\theta''} P_{I_j}(x_I)}$$



パストレーシングによるサンプリングの例

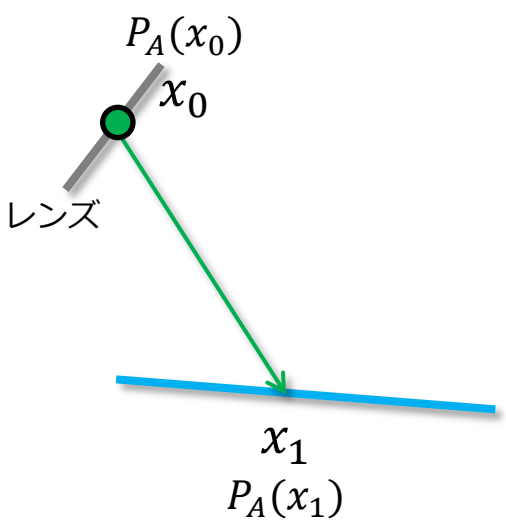
- camera.P_image_to_PA_x1()が実際の変換式を実装したものになっている。




```
Ray now_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
double now_sampled_pdf_omega = 1.0;
Vec previous_normal = camera.normal_on_lens;

for (int now_vertex_id = 1;; ++now_vertex_id) {
    Intersection intersection;
    if (!intersect_scene(now_ray, &intersection))
        break;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal); // 交差位置の法線（物体からのレイの入出を考慮）
    const double russian_roulette_probability = russian_roulette(now_object);

    // ロシアンルーレット
    if (rnd->next01() >= russian_roulette_probability) {
        break;
    }
    // 新しい頂点がサンプリングされた
    total_pdf_A *= russian_roulette_probability;

    const Vec between = now_ray.org - hitpoint.position;
    if (now_vertex_id == 1)
    {
        // x1だけは確率密度の計算が特殊になる
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = hitpoint.position - position_on_lens;
        const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, orienting_normal);
        total_pdf_A *= PA_x1;

        // さらに、MC_throughputの計算も若干特殊になる（カメラ周りの処理）
        MC_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput;
    } else {
        // 新しい頂点を得たので、今までの頂点をサンプリングする確率の総計を出す
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(between), orienting_normal) / between.length_squared());
        total_pdf_A *= now_sampled_pdf_A;
    }
```

パストレーシングによるサンプリングの例

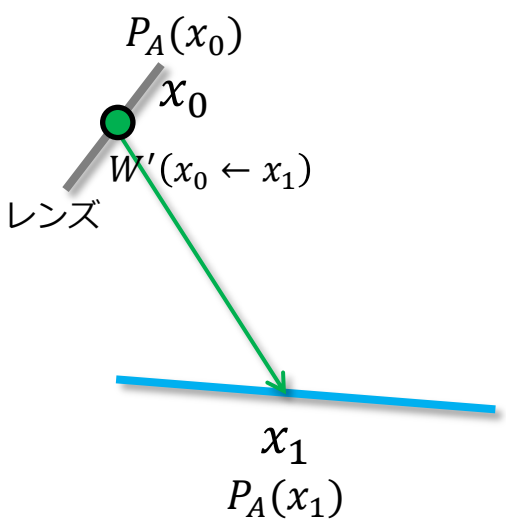
- なお、各頂点ごとに生成するかしないかをロシアンルーレットで判定することで無限に頂点が生成されるのを防いでいる。
- 最終的な頂点のサンプリング確率密度$P_A(x_i)$はロシアンルーレットの確率を含んだものになる。
  - ロシアンルーレットの確率は頂点ごとに独立に計算する。




```
Ray now_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
double now_sampled_pdf_omega = 1.0;
Vec previous_normal = camera.normal_on_lens;

for (int now_vertex_id = 1;; ++now_vertex_id) {
    Intersection intersection;
    if (!intersect_scene(now_ray, &intersection))
        break;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal); // 交差位置の法線（物体からのレイの入出を考慮）
    const double russian_roulette_probability = russian_roulette(now_object);

    // ロシアンルーレット
    if (rnd->next01() >= russian_roulette_probability) {
        break;
    }
    // 新しい頂点がサンプリングされた
    total_pdf_A *= russian_roulette_probability;

    const Vec between = now_ray.org - hitpoint.position;
    if (now_vertex_id == 1)
    {
        // x1だけは確率密度の計算が特殊になる
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = hitpoint.position - position_on_lens;
        const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, orienting_normal);
        total_pdf_A *= PA_x1;

        // さらに、MC_throughputの計算も若干特殊になる（カメラ周りの処理）
        MC_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput;
    } else {
        // 新しい頂点を得たので、今までの頂点をサンプリングする確率の総計を出す
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(between), orienting_normal) / between.length_squared());
        total_pdf_A *= now_sampled_pdf_A;
    }
```

パストレーシングによるサンプリングの例

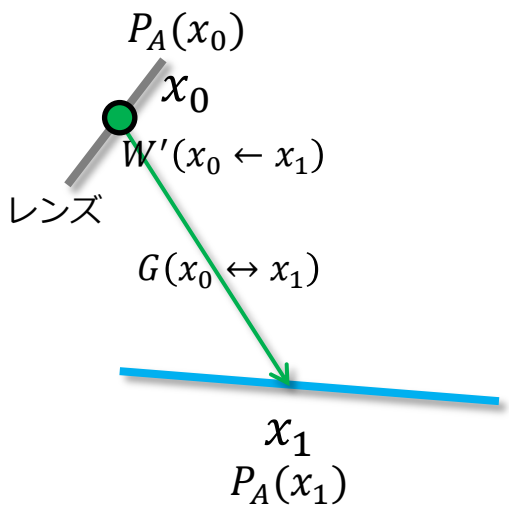
- モンテカルロスループットは通常はG項とBRDFを乗算するだけだが、x_1に関しては$W'(x_0 \leftarrow x_1)$の分を余分に考慮する必要がある。
  - センサのセンシティビティや、レンズに光が入射することによる幾何的な係数




```
Ray now_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
double now_sampled_pdf_omega = 1.0;
Vec previous_normal = camera.normal_on_lens;

for (int now_vertex_id = 1;; ++now_vertex_id) {
    Intersection intersection;
    if (!intersect_scene(now_ray, &intersection))
        break;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal); // 交差位置の法線（物体からのレイの入出を考慮）
    const double russian_roulette_probability = russian_roulette(now_object);

    // ロシアンルーレット
    if (rnd->next01() >= russian_roulette_probability) {
        break;
    }
    // 新しい頂点がサンプリングされた
    total_pdf_A *= russian_roulette_probability;

    const Vec between = now_ray.org - hitpoint.position;
    if (now_vertex_id == 1)
    {
        // x1だけは確率密度の計算が特殊になる
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = hitpoint.position - position_on_lens;
        const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, orienting_normal);
        total_pdf_A *= PA_x1;

        // さらに、MC_throughputの計算も若干特殊になる（カメラ周りの処理）
        MC_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput;
    } else {
        // 新しい頂点を得たので、今までの頂点をサンプリングする確率の総計を出す
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(between), orienting_normal) / between.length_squared());
        total_pdf_A *= now_sampled_pdf_A;
    }
```

パストレーシングによるサンプリングの例

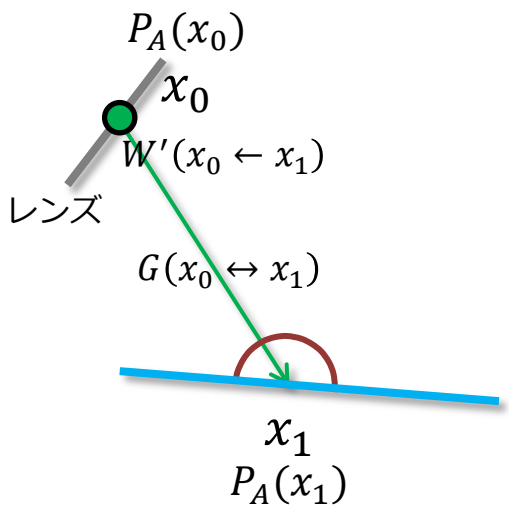
- その後、G項を計算し、新しい頂点を頂点リストに追加する。
  - その頂点までの確率密度の総計と、モンテカルロスループットの総計も追加する。
- ただし、新しい頂点が光源上にあった場合、そこで追跡を打ち切る。
  - 今回光源は全て反射率0と仮定しており、これ以上の追跡は無駄であるため。もし反射率0という仮定を入れないなら更なる追跡が必要になる。
  - この段階のトータルのコントリビューション（モンテカルロスループット/パスの確率密度）も計算しておく。




```
} else {
    // 新しい頂点を得たので、今までの頂点をサンプリングする確率の総計を出す
    const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(between), orienting_normal) / between.length_squared());
    total_pdf_A *= now_sampled_pdf_A;
}
// ジオメトリターム
const double G =  dot(normalize(between), orienting_normal) * dot(normalize(-1.0 * between), previous_normal) / between.length_squared();
MC_throughput = G * MC_throughput;

// 光源にヒットしたらそこで追跡終了（光源の反射率がゼロであることを仮定しており、これ以上の追跡は全てMC_throughputがゼロになるため。光源にも反射率があるならこの限りではない）
if (now_object.emission.length_squared() > 0) {
    vertices.push_back(Vertex(hitpoint.position, orienting_normal, hitpoint.normal, intersection.object_id, Vertex::OBJECT_TYPE_LIGHT, total_pdf_A, MC_throughput));
    const Color result = multiply(MC_throughput, now_object.emission) / total_pdf_A;
    return PathtracingResult(result, imagebuffer_x, imagebuffer_y, true);
}

// 新しい頂点を頂点リストに追加する
vertices.push_back(Vertex(hitpoint.position, orienting_normal, hitpoint.normal, intersection.object_id,
        now_object.reflection_type == REFLECTION_TYPE_DIFFUSE ? Vertex::OBJECT_TYPE_DIFFUSE :Vertex::OBJECT_TYPE_SPECULAR,
        total_pdf_A, MC_throughput));
```

パストレーシングによるサンプリングの例

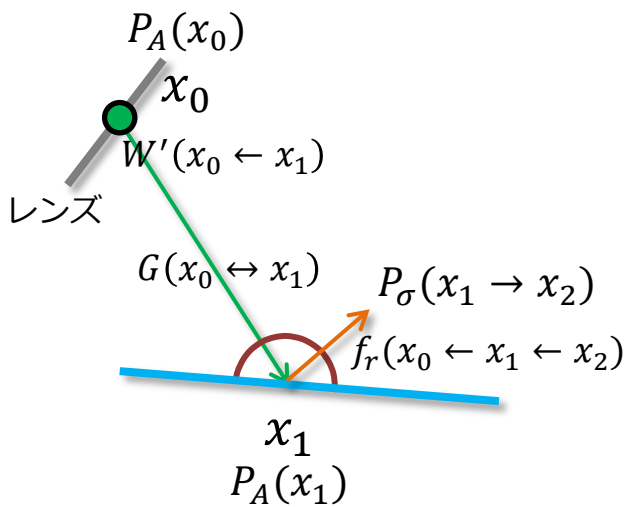
- 次の頂点をサンプリングするために、今の頂点を始点としてレイトレーシングを行う。これは立体角測度に関する確率密度によるサンプリングとなる。
- 使用する確率密度は現在の頂点の材質によって決める。
  - edupt同様、今回は完全拡散面、完全鏡面、完全ガラス面の三種類を考える。




```
// 材質ごとの処理
// 次の頂点をサンプリングするために、新しいレイの方向を決める。
switch (now_object.reflection_type) {
// 完全拡散面
case REFLECTION_TYPE_DIFFUSE: {
    const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &now_sampled_pdf_omega);
    now_ray = Ray(hitpoint.position, dir);
    MC_throughput = multiply(now_object.color / kPI, MC_throughput);
} break;
// 完全鏡面
case REFLECTION_TYPE_MIRROR: {
    // 完全鏡面なのでレイの反射方向は決定的。
    // スペキュラ面におけるサンプリング確率密度はDiracのδ関数の形をとるが、分母とキャンセルされるため、係数のみ与える。
    now_sampled_pdf_omega = 1.0;
    now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));

    // スループット調整
    // sample_pdf_omegaのDiracのδ関数とキャンセルするため、係数のみ与える。（このあたりはLighttracingと同様）
    // ガラスの場合も同様
    MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
} break;
// ガラス
case REFLECTION_TYPE_GLASS: {
    const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか。

    Vec reflection_dir, refraction_dir;
    double fresnel_reflectance, fresnel_transmittance;
    if (check_refraction(
        into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
        &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
        // 屈折 + 反射
        if (rnd->next01() < reflection_probability) { // 反射
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, reflection_dir);
            MC_throughput = fresnel_reflectance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= reflection_probability;
        } else { // 屈折
            const double nnt2 =
                pow(into ?
                refractive_index_of_vaccum / refractive_index_of_object :
                refractive_index_of_object / refractive_index_of_vaccum, 2.0);
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, refraction_dir);
            MC_throughput = nnt2 * fresnel_transmittance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= 1.0 - reflection_probability;
        }
    } else {
        // 全反射
        now_sampled_pdf_omega = 1.0;
        now_ray = Ray(hitpoint.position, reflection_dir);
        MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        break;
    }
} break;
}
```

完全拡散面の場合

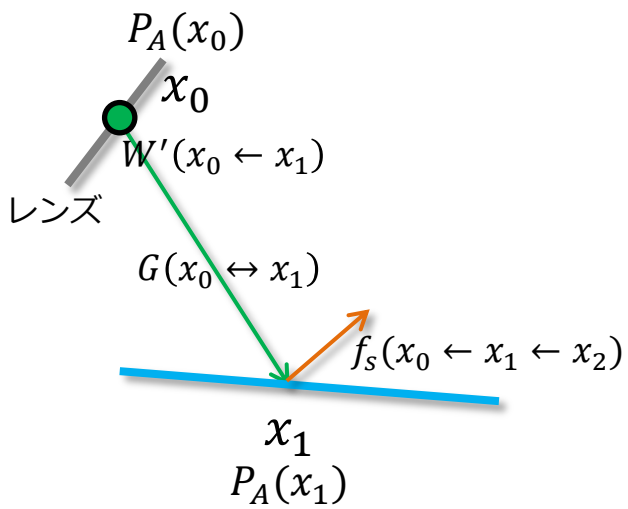
- 普通に半球上から次の方向をサンプリングする。 $\cos\theta$ を考慮しているため、確率密度は$\frac{\cos\theta}{\pi}$となる。この確率密度は次の頂点x_2の分になる。
- さらに、スループットにBRDFの分を乗算する。完全拡散面なので、 $\frac{\rho}{\pi}$となる。（ ρは反射率）このスループットは次の頂点x_2の分になる。




```
// 材質ごとの処理
// 次の頂点をサンプリングするために、新しいレイの方向を決める。
switch (now_object.reflection_type) {
// 完全拡散面
case REFLECTION_TYPE_DIFFUSE: {
    const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &now_sampled_pdf_omega);
    now_ray = Ray(hitpoint.position, dir);
    MC_throughput = multiply(now_object.color / kPI, MC_throughput);
} break;
// 完全鏡面
case REFLECTION_TYPE_MIRROR: {
    // 完全鏡面なのでレイの反射方向は決定的。
    // スペキュラ面におけるサンプリング確率密度はDiracのδ関数の形をとるが、分母とキャンセルされるため、係数のみ与える。
    now_sampled_pdf_omega = 1.0;
    now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));

    // スループット調整
    // sample_pdf_omegaのDiracのδ関数とキャンセルするため、係数のみ与える。（このあたりはLighttracingと同様）
    // ガラスの場合も同様
    MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
} break;
// ガラス
case REFLECTION_TYPE_GLASS: {
    const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか。

    Vec reflection_dir, refraction_dir;
    double fresnel_reflectance, fresnel_transmittance;
    if (check_refraction(
        into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
        &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
        // 屈折 + 反射
        if (rnd->next01() < reflection_probability) { // 反射
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, reflection_dir);
            MC_throughput = fresnel_reflectance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= reflection_probability;
        } else { // 屈折
            const double nnt2 =
                pow(into ?
                refractive_index_of_vaccum / refractive_index_of_object :
                refractive_index_of_object / refractive_index_of_vaccum, 2.0);
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, refraction_dir);
            MC_throughput = nnt2 * fresnel_transmittance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= 1.0 - reflection_probability;
        }
    } else {
        // 全反射
        now_sampled_pdf_omega = 1.0;
        now_ray = Ray(hitpoint.position, reflection_dir);
        MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        break;
    }
} break;
}
```

完全鏡面の場合

- 完全鏡面の場合、次のレイの方向は一方向に決まる。
- ただし、その場合でも立体角測度に関する確率密度関数とBRDFを求める必要がある。
  - モンテカルロ積分に投入する必要があるため。
- そこで、Diracのδ関数を導入する。



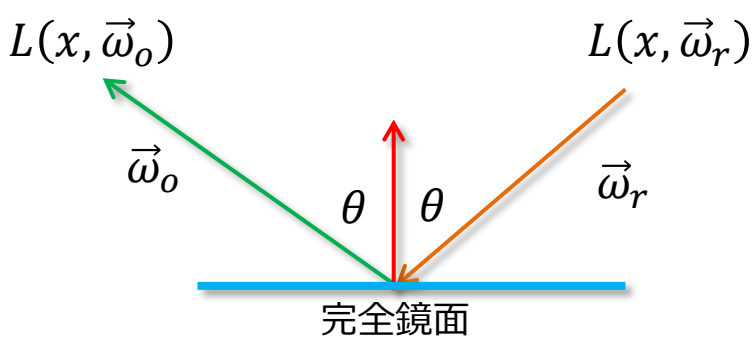

```
// 材質ごとの処理
// 次の頂点をサンプリングするために、新しいレイの方向を決める。
switch (now_object.reflection_type) {
// 完全拡散面
case REFLECTION_TYPE_DIFFUSE: {
    const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &now_sampled_pdf_omega);
    now_ray = Ray(hitpoint.position, dir);
    MC_throughput = multiply(now_object.color / kPI, MC_throughput);
} break;
// 完全鏡面
case REFLECTION_TYPE_MIRROR: {
    // 完全鏡面なのでレイの反射方向は決定的。
    // スペキュラ面におけるサンプリング確率密度はDiracのδ関数の形をとるが、分母とキャンセルされるため、係数のみ与える。
    now_sampled_pdf_omega = 1.0;
    now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));

    // スループット調整
    // sample_pdf_omegaのDiracのδ関数とキャンセルするため、係数のみ与える。（このあたりはLighttracingと同様）
    // ガラスの場合も同様
    MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
} break;
// ガラス
case REFLECTION_TYPE_GLASS: {
    const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか。

    Vec reflection_dir, refraction_dir;
    double fresnel_reflectance, fresnel_transmittance;
    if (check_refraction(
        into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
        &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
        // 屈折 + 反射
        if (rnd->next01() < reflection_probability) { // 反射
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, reflection_dir);
            MC_throughput = fresnel_reflectance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= reflection_probability;
        } else { // 屈折
            const double nnt2 =
                pow(into ?
                refractive_index_of_vaccum / refractive_index_of_object :
                refractive_index_of_object / refractive_index_of_vaccum, 2.0);
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, refraction_dir);
            MC_throughput = nnt2 * fresnel_transmittance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= 1.0 - reflection_probability;
        }
    } else {
        // 全反射
        now_sampled_pdf_omega = 1.0;
        now_ray = Ray(hitpoint.position, reflection_dir);
        MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        break;
    }
} break;
}
```

Diracのδ関数

- 完全鏡面のBRDFとして望ましい性質。
  - 光が完全にある一方向のみに反射されること。つまり以下のような性質。

$$L(x,\vec{\omega}_o) = \int_{S^2} L(x,\vec{\omega}_i) f_s(\vec{\omega}_i \to \vec{\omega}_o) \cos\theta \, d\sigma(\vec{\omega}_i) = L(x,\vec{\omega}_r)$$

- そのようなBRDF、f_sとしてDiracのδ関数を採用する。
  - この関数を使うことで、完全鏡面やガラス面も普通の完全拡散面に近い取扱いを行える。
- レンダリングの文脈では以下のような関数δを考える。
  - $x \neq 0 \Longrightarrow \delta(x) = 0$
  - $\int_{\mathbb{R}} \delta(x) dx = 1$



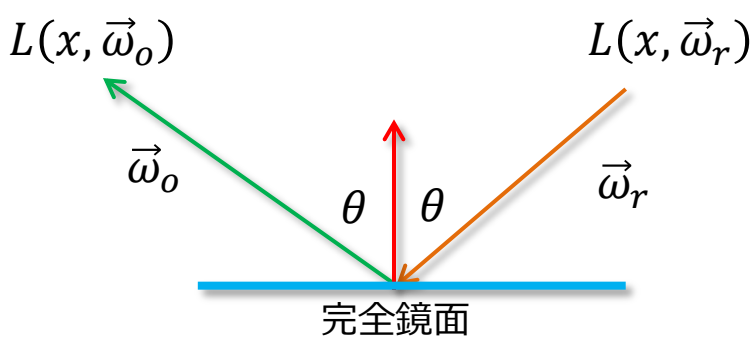
Diracのδ関数

- δ関数を使ってf_sを以下のように書くことが出来る。

$$f_s(\vec{\omega}_i \to \vec{\omega}_o) = \frac{\delta_\sigma(\vec{\omega}_i - \vec{\omega}_r)}{\cos\theta}$$

- これは確かに、以下の式を満たす。

$$L(x, \vec{\omega}_o) = \int_{S^2} L(x, \vec{\omega}_i) f_s(\vec{\omega}_i \to \vec{\omega}_o) \cos\theta \, d\sigma(\vec{\omega}_i) = L(x, \vec{\omega}_r)$$



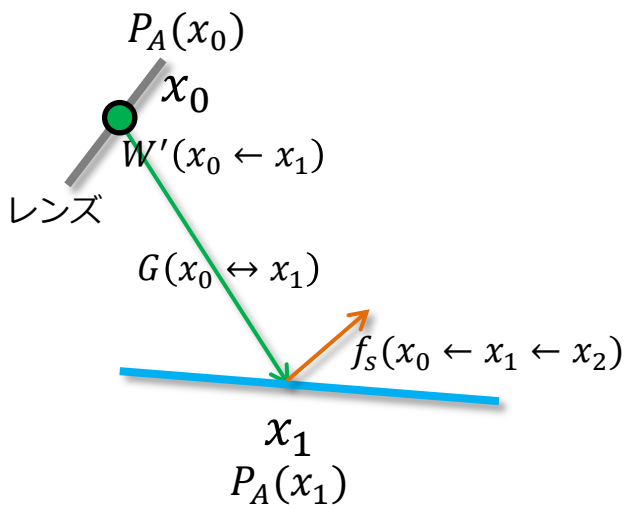
Diracのδ関数

- この時、立体角測度に関する確率密度関数$P_\sigma(\vec{\omega}_i)$は以下のようになる。

$$P_\sigma(\vec{\omega}_i) = \delta_\sigma(\vec{\omega}_i - \vec{\omega}_r)$$

完全鏡面の場合

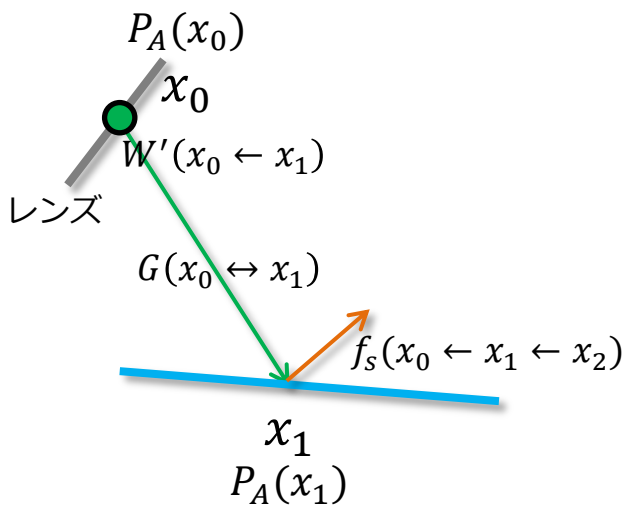
- 以上より、now_sampled_pdf_omegaは$\delta_\sigma(\vec{\omega}_i - \vec{\omega}_r)$となり、スループットにはBRDFである$\frac{\delta_\sigma(\vec{\omega}_i - \vec{\omega}_r)}{\cos\theta}$を乗算すればよいということになる。しかし、$\delta_\sigma$を数値的に表現することはできない。
- 記号的にδ_σの存在を記録してもよいが、最終的なモンテカルロ積分においてBRDFが分子に、確率密度に分母が来て、一対一対応がとれることから、δ_σは最終的にお互い打ち消しあうという事実を利用する。
- つまり、δ_σを抜いた分をnow_sampled_pdf_omegaとすればよいし、モンテカルロスループットも抜いた分で計算する。




```
// 材質ごとの処理
// 次の頂点をサンプリングするために、新しいレイの方向を決める。
switch (now_object.reflection_type) {
// 完全拡散面
case REFLECTION_TYPE_DIFFUSE: {
    const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &now_sampled_pdf_omega);
    now_ray = Ray(hitpoint.position, dir);
    MC_throughput = multiply(now_object.color / kPI, MC_throughput);
} break;
// 完全鏡面
case REFLECTION_TYPE_MIRROR: {
    // 完全鏡面なのでレイの反射方向は決定的。
    // スペキュラ面におけるサンプリング確率密度はDiracのδ関数の形をとるが、分母とキャンセルされるため、係数のみ与える。
    now_sampled_pdf_omega = 1.0;
    now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));

    // スループット調整
    // sample_pdf_omegaのDiracのδ関数とキャンセルするため、係数のみ与える。（このあたりはLighttracingと同様）
    // ガラスの場合も同様
    MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
} break;
// ガラス
case REFLECTION_TYPE_GLASS: {
    const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか。

    Vec reflection_dir, refraction_dir;
    double fresnel_reflectance, fresnel_transmittance;
    if (check_refraction(
        into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
        &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
        // 屈折 + 反射
        if (rnd->next01() < reflection_probability) { // 反射
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, reflection_dir);
            MC_throughput = fresnel_reflectance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= reflection_probability;
        } else { // 屈折
            const double nnt2 =
                pow(into ?
                refractive_index_of_vaccum / refractive_index_of_object :
                refractive_index_of_object / refractive_index_of_vaccum, 2.0);
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, refraction_dir);
            MC_throughput = nnt2 * fresnel_transmittance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= 1.0 - reflection_probability;
        }
    } else {
        // 全反射
        now_sampled_pdf_omega = 1.0;
        now_ray = Ray(hitpoint.position, reflection_dir);
        MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        break;
    }
} break;
}
```

完全ガラス面の場合

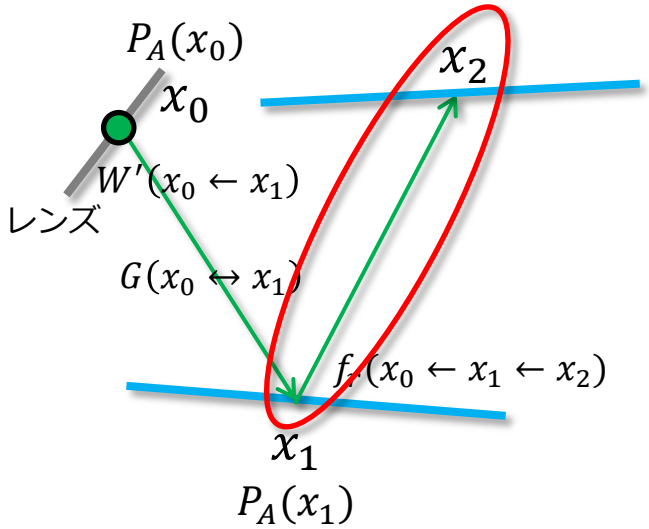
- 基本的に完全鏡面の場合と同じになる。全反射の場合は全く同じだし、反射＋屈折が起こる際も、ロシアンルーレットで反射が選ばれればやはりまったく同じになる。
  - もちろん、ロシアンルーレットの確率をtotal_pdf_Aに乗算する必要がある。
- 屈折が起こる場合も、大きくは変わらないが、屈折前と屈折後で立体角が変化することによる放射輝度値の変化（**nnt2**）をスループットの計算に入れる。




```
// 材質ごとの処理
// 次の頂点をサンプリングするために、新しいレイの方向を決める。
switch (now_object.reflection_type) {
// 完全拡散面
case REFLECTION_TYPE_DIFFUSE: {
    const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &now_sampled_pdf_omega);
    now_ray = Ray(hitpoint.position, dir);
    MC_throughput = multiply(now_object.color / kPI, MC_throughput);
} break;
// 完全鏡面
case REFLECTION_TYPE_MIRROR: {
    // 完全鏡面なのでレイの反射方向は決定的。
    // スペキュラ面におけるサンプリング確率密度はDiracのδ関数の形をとるが、分母とキャンセルされるため、係数のみ与える。
    now_sampled_pdf_omega = 1.0;
    now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));

    // スループット調整
    // sample_pdf_omegaのDiracのδ関数とキャンセルするため、係数のみ与える。（このあたりはLighttracingと同様）
    // ガラスの場合も同様
    MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
} break;
// ガラス
case REFLECTION_TYPE_GLASS: {
    const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか。

    Vec reflection_dir, refraction_dir;
    double fresnel_reflectance, fresnel_transmittance;
    if (check_refraction(
        into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
        &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
        // 屈折 + 反射
        if (rnd->next01() < reflection_probability) { // 反射
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, reflection_dir);
            MC_throughput = fresnel_reflectance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= reflection_probability;
        } else { // 屈折
            const double nnt2 =
                pow(into ?
                refractive_index_of_vaccum / refractive_index_of_object :
                refractive_index_of_object / refractive_index_of_vaccum, 2.0);
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, refraction_dir);
            MC_throughput = nnt2 * fresnel_transmittance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= 1.0 - reflection_probability;
        }
    } else {
        // 全反射
        now_sampled_pdf_omega = 1.0;
        now_ray = Ray(hitpoint.position, reflection_dir);
        MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        break;
    }
} break;
}
```

パストレーシングによるサンプリングの例

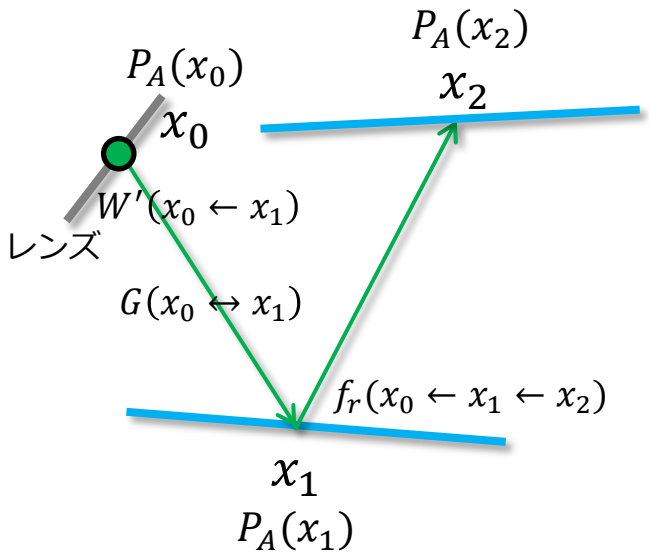
- 次のループの最初で、新しい頂点がシーン上からサンプリングされる。




```
Ray now_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
double now_sampled_pdf_omega = 1.0;
Vec previous_normal = camera.normal_on_lens;

for (int now_vertex_id = 1;; ++now_vertex_id) {
    Intersection intersection;
    if (!intersect_scene(now_ray, &intersection))
        break;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal); // 交差位置の法線（物体からのレイの入出を考慮）
    const double russian_roulette_probability = russian_roulette(now_object);

    // ロシアンルーレット
    if (rnd->next01() >= russian_roulette_probability) {
        break;
    }
    // 新しい頂点がサンプリングされた
    total_pdf_A *= russian_roulette_probability;

    const Vec between = now_ray.org - hitpoint.position;
    if (now_vertex_id == 1)
    {
        // x1だけは確率密度の計算が特殊になる
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = hitpoint.position - position_on_lens;
        const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, orienting_normal);
        total_pdf_A *= PA_x1;

        // さらに、MC_throughputの計算も若干特殊になる（カメラ周りの処理）
        MC_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput;
    } else {
        // 新しい頂点を得たので、今までの頂点をサンプリングする確率の総計を出す
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(between), orienting_normal) / between.length_squared());
        total_pdf_A *= now_sampled_pdf_A;
    }
```

パストレーシングによるサンプリングの例

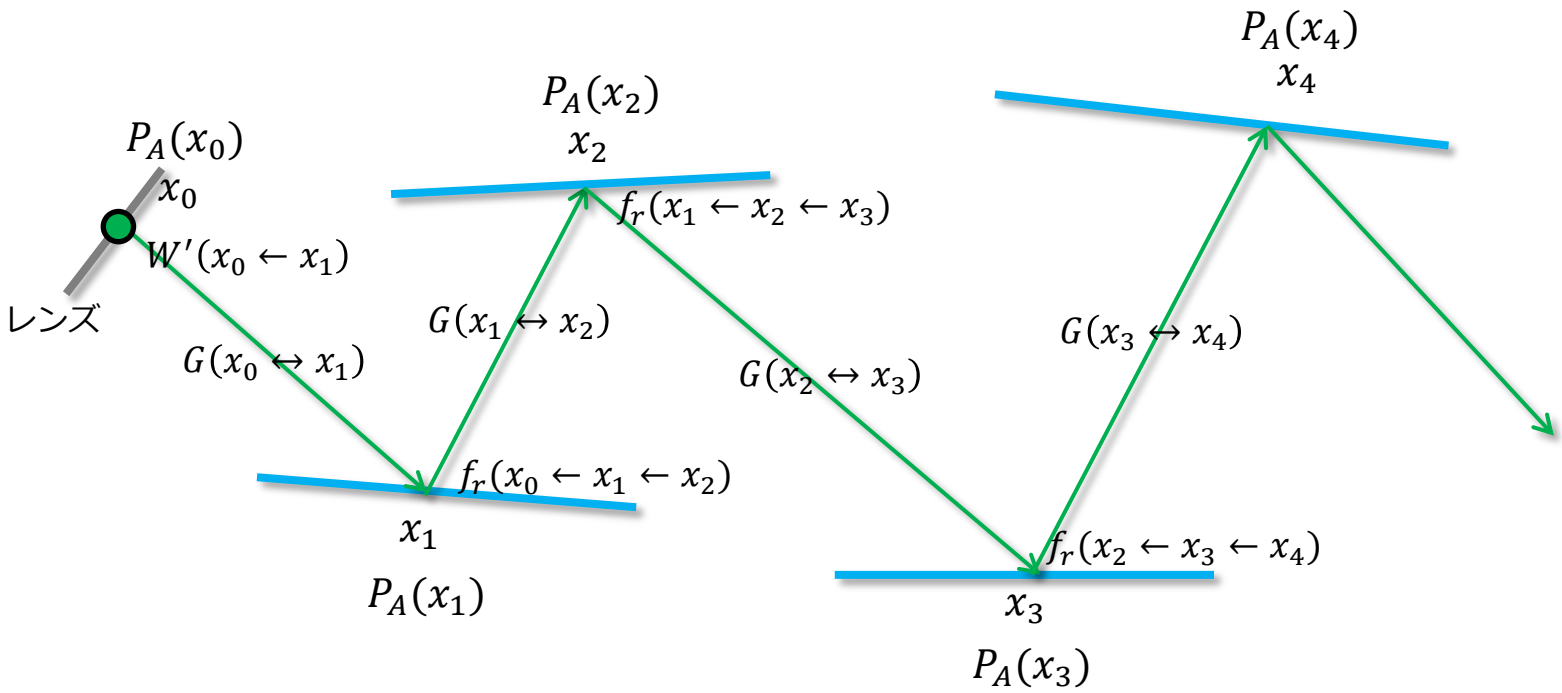
- 最初の頂点x_1 以外は、面積測度に関する確率密度$P_A(x_i)$を、その頂点をサンプリングするための立体角測度に関する確率密度$P_\sigma(x_{i-1} \to x_i)$から変換して求める。
  - 変換には、関係式$P_A(x_i) = \frac{\cos\theta}{r^2} P_\sigma(x_{i-1} \to x_i)$を使う。




```
Ray now_ray(position_on_lens, normalize(position_on_objectplane - position_on_lens));
double now_sampled_pdf_omega = 1.0;
Vec previous_normal = camera.normal_on_lens;

for (int now_vertex_id = 1;; ++now_vertex_id) {
    Intersection intersection;
    if (!intersect_scene(now_ray, &intersection))
        break;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal); // 交差位置の法線（物体からのレイの入出を考慮）
    const double russian_roulette_probability = russian_roulette(now_object);

    // ロシアンルーレット
    if (rnd->next01() >= russian_roulette_probability) {
        break;
    }
    // 新しい頂点がサンプリングされた
    total_pdf_A *= russian_roulette_probability;

    const Vec between = now_ray.org - hitpoint.position;
    if (now_vertex_id == 1)
    {
        // x1だけは確率密度の計算が特殊になる
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = hitpoint.position - position_on_lens;
        const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, orienting_normal);
        total_pdf_A *= PA_x1;

        // さらに、MC_throughputの計算も若干特殊になる（カメラ周りの処理）
        MC_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput;
    } else {
        // 新しい頂点を得たので、今までの頂点をサンプリングする確率の総計を出す
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(between), orienting_normal) / between.length_squared());
        total_pdf_A *= now_sampled_pdf_A;
    }
```

パストレーシングによるサンプリングの例

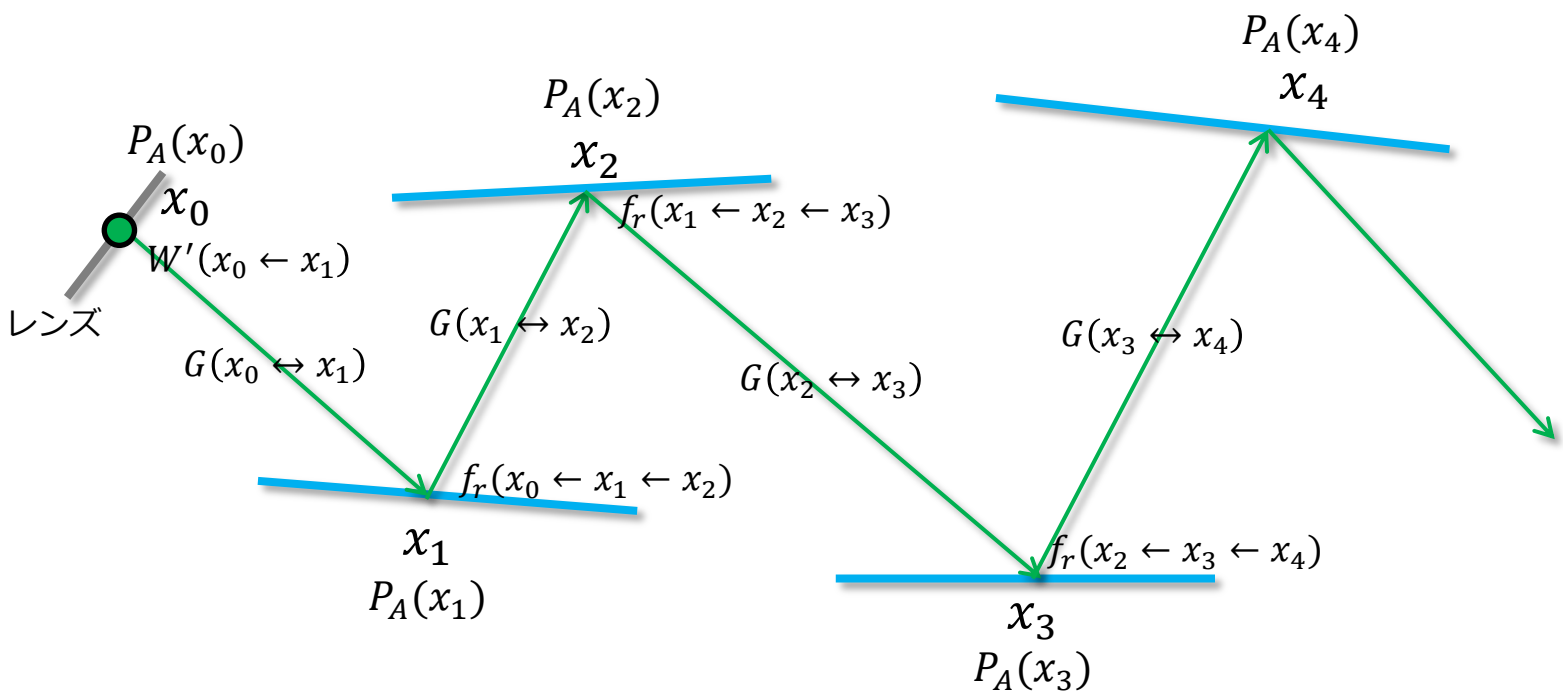
- 以上の処理を（ロシアンルーレットで打ち切られるか光源にヒットするまで）繰り返すことで頂点を生成していく。
- 同時に、モンテカルロ積分に使うためのスループットと確率密度も計算していく。



パストレーシングによるサンプリングの例

- このようにして生成されたパスはモンテカルロ積分に使うことが出来る。

$$
\begin{aligned}
\widehat{I_j} = & \frac{W'(x_0 \leftarrow x_1) G(x_0 \leftrightarrow x_1) L_e(x_0 \leftarrow x_1)}{P_A(x_0) P_A(x_1)} \\
& + \frac{W'(x_0 \leftarrow x_1) G(x_0 \leftrightarrow x_1) f_r(x_0 \leftarrow x_1 \leftarrow x_2) G(x_1 \leftrightarrow x_2) L_e(x_1 \leftarrow x_2)}{P_A(x_0) P_A(x_1) P_A(x_2)} \\
& + \cdots
\end{aligned}
$$



5.2 ライトトレーシングによるパスのサンプリング

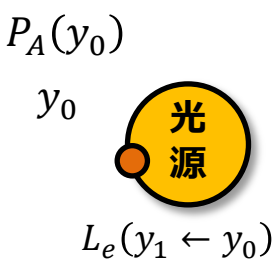
lighttracing.h

- ライトトレーシングによる頂点のサンプリングを行う
  - generate_vertices_by_lighttracing()

```
#pragma once
#ifndef _LIGHTTRACING_H_
#define _LIGHTTRACING_H_

#include "random.h"
#include "vector.h"
#include "color.h"
#include "camera.h"
#include "material.h"
#include "sphere.h"
#include "camera.h"
#include "vertex.h"

namespace edubpt {

struct LightTracingResult {
  Color color;
  int imageBuffer_x, imageBuffer_y;
  bool is_lens_hit;

  LightTracingResult(const Color &color, const int &imageBuffer_x, const int &imageBuffer_y, const bool &is_lens_hit) :
    color(color), imageBuffer_x(imageBuffer_x), imageBuffer_y(imageBuffer_y), is_lens_hit(is_lens_hit) {}
};
LightTracingResult generate_vertices_by_lighttracing(const Camera &camera, Random *rnd, std::vector<Vertex> &vertices) {
  const int max_depth = 10;
  double pdf_A_on_light;
  const Vec position_on_light = sphereLight[0].position + sample_spherepoint(sphereLight[0].radius, rnd, &pdf_A_on_light);
  const double cos_term = dot(normalize(-position_on_light), sphereLight[0].emission);
  double total_pdf_A = pdf_A_on_light; // 最初の頂点の確率密度
  // 光源上に最初の1つ目の光を表わすパスを追加
  vertices.push_back(Vertex(position_on_light, normal_on_light, normal_on_light, Light(0, Vertex::OBJECT_TYPE_LIGHT, total_pdf_A, Color(0, 0, 0)));

  // 現在の頂点情報 (レンダリング画像のピクセル)
  // 今生成された頂点 (1つ目の光源)から2つ目の頂点の位置を決めるのに使う。
  // 光源の放射輝度 (emission)を保存しておく。1つ目の頂点なので、sphereLight[0].emission になる。
  Color MC_throughput = sphereLight[0].emission;

  // 現在の頂点情報を保存しておくので、BRDFsの重みにおけるサンプリング方法と同じものを使って次の方向を決める。
  double new_sampled_pdf_omega;
  const Vec next_dir = sample_hemisphere_cos_term(normal_on_light, rnd, &new_sampled_pdf_omega);
  Ray new_ray(position_on_light, next_dir);
  Vec previous_normal = normal_on_light;

  for (;;) {
    Intersection intersection;
    const bool scene_hit = intersect_scene(new_ray, &intersection);

    // レンズとの交差判定
    const double &lens_z = camera.position_on_objectplane, &position_on_imagesensor, &u_on_imagesensor;
    const double &lens_r = camera.intersect_lens(new_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &u_on_imagesensor);
    if (DEPS < lens_r && lens_z < intersection.hitpoint.distance) {
      const double &image_x, &image_y;
      const Vec v0_v1 = position_on_imagesensor - position_on_lens;
      const Vec v0_v9 = position_on_objectplane - position_on_lens;
      const Vec v0_v1 = new_ray.org - position_on_lens;

      // イメージセンサ (ピクセル) との位置 = 光線方向によって要等変が変わる。
      // (v0 - hit)はw方向に一定なので必要な変数はxのみでよい!
      int x = (int)in_imagesensor.x;
      int y = (int)in_imagesensor.y;
      clamp(x, 0, camera.image_width_pel);
      clamp(y, 0, camera.image_height_pel);

      // レンズ上の点の方向でピクセルの値を加算。
      const double new_sampled_pdf_A = new_sampled_pdf_omega * (dot(normalize(v0_v1), normalize(camera.imagesensor_dir)) / v0_v1.length_squared());
      total_pdf_A *= new_sampled_pdf_A;

      // ジオメトリターム
      const double G = dot(normalize(v0_v1), normalize(camera.imagesensor_dir)) * dot(normalize(-1.0 * v0_v1), previous_normal) / v0_v1.length_squared();
      MC_throughput *= G * MC_throughput;

      // レンズ上に各頂点のパス長情報パスリストに追加 (最終点の前から)
      vertices.push_back(Vertex(position_on_lens, normalize(camera.imagesensor_dir), normalize(camera.imagesensor_dir), -1, Vertex::OBJECT_TYPE_LENS, total_pdf_A, MC_throughput));

      // 最終的な結果計算 = センサーセンスポイント全体に、ピクセルから届いた最終的な結果を足す。
      const Color result = Camera.m_kpixel_dir, v0_v1, v0_v1) * MC_throughput) / total_pdf_A;
      return LightTracingResult(result, x, y, true);
    }
    if (!scene_hit)
      break;

    const Sphere &one_object = spherelist[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    // 交差位置の法線 (視点からみたレイの方向考慮)
    const Vec &normal = hitpoint.normal, orienting_normal = (dot(ray.dir, hitpoint.normal) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal));
    const double russian_roulette_probability = max(one_object.color);
    // ロシアンルーレット
    if (rnd->next01() >= russian_roulette_probability) {
      break;
    }

    // 新しい頂点がサンプリングされたので、パーセントの確率密度で乗算する。
    total_pdf_A *= russian_roulette_probability;

    const Vec to_next_vertex = ray.org - hitpoint.position;
    // 次の頂点までの方向ベクトルとレンズの法線方向の内積を使って幾何項を計算する。
    const double new_sampled_pdf_A = new_sampled_pdf_omega * (dot(normalize(to_next_vertex), orienting_normal)) / to_next_vertex.length_squared());
    // 最初の頂点をサンプリングする確率密度の値が変わる。
    total_pdf_A *= new_sampled_pdf_A;

    // ジオメトリターム (G項)
    const double G = dot(normalize(to_next_vertex), orienting_normal) * dot(normalize(-1.0 * to_next_vertex), previous_normal) / to_next_vertex.length_squared();
    MC_throughput *= G * MC_throughput;

    // 新しい頂点を頂点リストに追加する。
    vertices.push_back(Vertex(hitpoint.position, orienting_normal, hitpoint.normal, intersection.object_id,
        new_object.reflection_type == REFLECTION_TYPE_DIFFUSE ? Vertex::OBJECT_TYPE_DIFFUSE : Vertex::OBJECT_TYPE_SPECULAR,
        total_pdf_A, MC_throughput));


    // 反射してこの方向
    // 次の頂点をサンプリングするために、新しいレイの方向を決める。
    switch (new_object.reflection_type) {
    case DIFFUSE: {
    case REFLECTION_TYPE_DIFFUSE: {
      // 次の方向をサンプリング
      const Vec &next_dir = sample_hemisphere_cos_term(orienting_normal, rnd, &new_sampled_pdf_omega);
      new_ray = Ray(hitpoint.position, dir);
      // スループット調整 (dir <- hitpoint.position <- 一致時 の反射がかかっているのでBRDF調整する)
      MC_throughput = multiply(new_object.color / M_PI, MC_throughput);
    } break;

    // 完全鏡面
    case REFLECTION_TYPE_REFLECTION: {
      // 次の方向をサンプリングするために、新しいレイの方向を決める。
      // 完全鏡面なのでサンプリング確率密度はdiracの確率の所をとるが、分母とナンセンスなものとなる。結果のみ与える。
      new_sampled_pdf_omega = 1.0;
      new_ray = Ray(hitpoint.position, reflection_vector(new_ray.dir, hitpoint.normal));

      // スループット調整
      // (ray.dir, hitpoint.normal)がなる。 (法線角度に依存する関数)
      // (ray.dir, hitpoint.normal)がキャンセルされるので、(よって、 cos項) (分母)(分子)して、new_sampled_pdf_omega = 1.0にしたのち今のとおり
      // 次の pdf_omegaからなる。 (法線角度に依存する確率密度)
      // BRDFsはBRDFで分母掛けすると、pdf_omegaとの正負が相殺されるようになることからこのようになる。
      MC_throughput = multiply(new_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
    } break;

    // ガラス
    case REFLECTION_TYPE_GLASS: {
      const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトからはあるのか、入るのか

      Vec reflection_dir, refraction_dir;
      double Fresnel_reflectance, Fresnel_transmittance;
      if (check_refraction
          into, hitpoint.position, new_ray.dir, hitpoint.normal, orienting_normal,
          &refraction_dir, &refraction_dir, &Fresnel_reflectance, &Fresnel_transmittance) {
        // 屈折
        if (rnd->next01() < reflection_probability) { // 反射
          new_sampled_pdf_omega = 1.0;
          new_ray = Ray(hitpoint.position, reflection_dir);
          MC_throughput = Fresnel_reflectance * multiply(new_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
          total_pdf_A *= reflection_probability;
        } else { // 屈折
          const double nnt = (
              into ? (
                  refractive_index_of_object / refractive_index_of_vacuum :
                  refractive_index_of_vacuum / refractive_index_of_object, 2.0);
          new_sampled_pdf_omega = 1.0;
          new_ray = Ray(hitpoint.position, refraction_dir);
          MC_throughput = nnt2 * Fresnel_transmittance * multiply(new_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
          total_pdf_A *= 1.0 - reflection_probability;
        }
      } else {
        // 全反射
        new_sampled_pdf_omega = 1.0;
        new_ray = Ray(hitpoint.position, reflection_dir);
        MC_throughput = multiply(new_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        break;
      }
    } break;

    }

    previous_normal = orienting_normal;
  }

  return LightTracingResult(Color(), 0, 0, false);
}

}

#endif
```

ライトトレーシングによるサンプリングの例

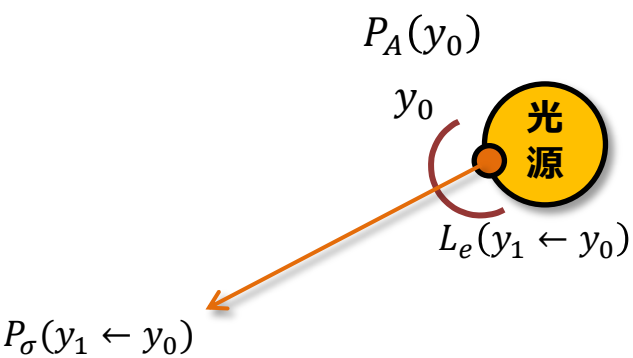
- まず、光源上y_0にサンプルを生成する。頂点のサンプリング確率密度も求める。
  - これが$P_A(y_0)$になる。ソースコード中ではpdf_A_on_lightである。
  - 半径Rの球上から一様にサンプリングしているため、 $P_A(y_0) = \frac{1}{4\pi R^2}$になる。
- パストレーシングの場合と同様に、シーンをトレースし、次々に頂点をサンプリングしていく。
  - 頂点y_Nを追加するとき、 Nまでの確率密度の総計（$\prod_{i=0}^{N} P_A(y_i)$）とモンテカルロスループットの総計（G項とBRDFの積）が必要になるので、順次計算しておく。
  - それぞれソースコード中ではtotal_pdf_AとMC_throughputという変数に格納しておく。
  - この辺もパストレーシングの場合と同様。
- なお、完全拡散光源なので$L_e(y_1 \leftarrow y_0)$は次の頂点y_1をサンプリングしなくても決まる。




```
LighttracingResult generate_vertices_by_lighttracing(const Camera &camera, Random *rnd, std::vector<Vertex> &vertices) {
    // 光源上にサンプル点生成（y0）
    double pdf_A_on_light;
    const Vec position_on_light = spheres[LightID].position + sample_sphere(spheres[LightID].radius, rnd, &pdf_A_on_light);
    const Vec normal_on_light = normalize(position_on_light - spheres[LightID].position);
    double total_pdf_A = pdf_A_on_light; // 確率密度の積を保持（面積測度に関する確率密度）

     // 光源上に生成された頂点を頂点リストに追加
    vertices.push_back(Vertex(position_on_light, normal_on_light, normal_on_light, LightID, Vertex::OBJECT_TYPE_LIGHT, total_pdf_A, Color(0, 0, 0)));

    // 現在の放射輝度（モンテカルロ積分のスループット）
    // 本当は次の頂点（y1）が決まらないと光源からその方向への放射輝度値は決まらないが、今回は
    // 完全拡散光源を仮定しているので、方向に依らずに一定の値（spheres[LightID].emission）になる。
    Color MC_throughput = spheres[LightID].emission;

    // 完全拡散光源を仮定しているので、Diffuse面におけるサンプリング方法と同じものをつかって次の方向を決める
    double now_sampled_pdf_omega;
    const Vec next_dir = sample_hemisphere_cos_term(normal_on_light, rnd, &now_sampled_pdf_omega);
    Ray now_ray(position_on_light, next_dir);
    Vec previous_normal = normal_on_light;
```

ライトトレーシングによるサンプリングの例

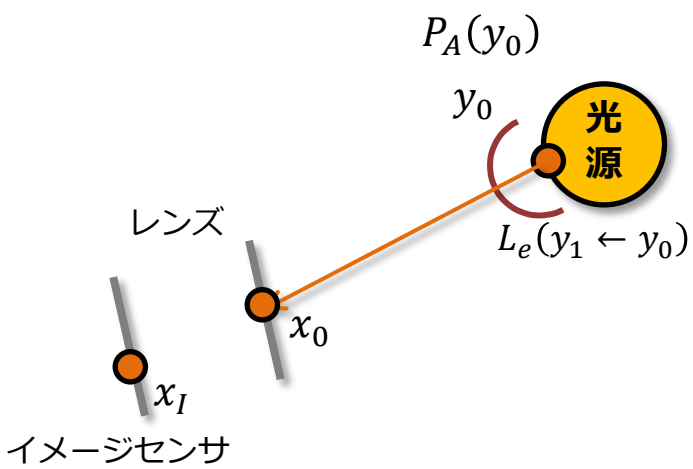
- 光源の次の頂点を生成するために、レイの方向をサンプリングする。これは、完全拡散面上における方向のサンプリングと同じようにして行う。




```
LighttracingResult generate_vertices_by_lighttracing(const Camera &camera, Random *rnd, std::vector<Vertex> &vertices) {
    // 光源上にサンプル点生成（y0）
    double pdf_A_on_light;
    const Vec position_on_light = spheres[LightID].position + sample_sphere(spheres[LightID].radius, rnd, &pdf_A_on_light);
    const Vec normal_on_light = normalize(position_on_light - spheres[LightID].position);
    double total_pdf_A = pdf_A_on_light; // 確率密度の積を保持（面積測度に関する確率密度）

     // 光源上に生成された頂点を頂点リストに追加
    vertices.push_back(Vertex(position_on_light, normal_on_light, normal_on_light, LightID, Vertex::OBJECT_TYPE_LIGHT, total_pdf_A, Color(0, 0, 0)));

    // 現在の放射輝度（モンテカルロ積分のスループット）
    // 本当は次の頂点（y1）が決まらないと光源からその方向への放射輝度値は決まらないが、今回は
    // 完全拡散光源を仮定しているので、方向に依らずに一定の値（spheres[LightID].emission）になる。
    Color MC_throughput = spheres[LightID].emission;

    // 完全拡散光源を仮定しているので、Diffuse面におけるサンプリング方法と同じものをつかって次の方向を決める
    double now_sampled_pdf_omega;
    const Vec next_dir = sample_hemisphere_cos_term(normal_on_light, rnd, &now_sampled_pdf_omega);
    Ray now_ray(position_on_light, next_dir);
    Vec previous_normal = normal_on_light;
```

ライトトレーシングによるサンプリングの例

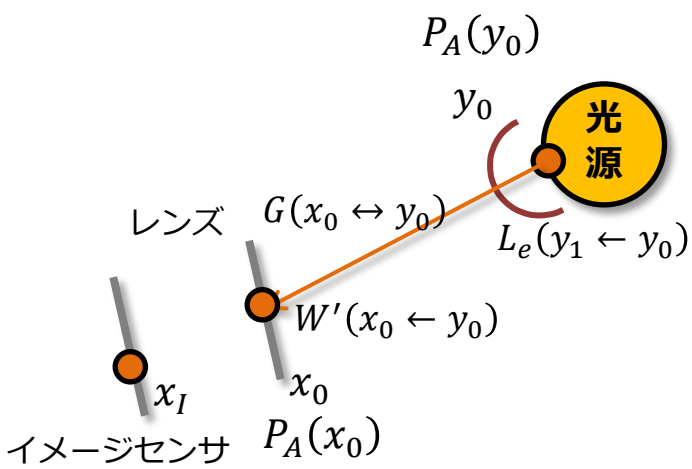
- レイの方向に向かってレイトレーシングして次の頂点を得る。
- この時、レンズとの交差判定も行う。
  - レンズにヒットした場合、そこで追跡を打ち切る。
  - 幾何的な計算により、レンズ上の点と、対応するイメージセンサ上の点が求まる。




```
for (;;) {
    Intersection intersection;
    const bool scene_hit = intersect_scene(now_ray, &intersection);

    // レンズと交差判定
    Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
    const double lens_t = camera.intersect_lens(now_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
    if (kEPS < lens_t && lens_t < intersection.hitpoint.distance) {
        // レイがレンズにヒット&イメージセンサにヒット
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = now_ray.org - position_on_lens;

        // イメージセンサ（ピクセル上）の位置、入射方向によって寄与率が変わる
        //  （が、今回はWを定数にしているのでは寄与率は変わらない）
        int x = (int)uv_on_imagesensor.x;
        int y = (int)uv_on_imagesensor.y;
        clamp(x, 0, camera.image_width_px);
        clamp(y, 0, camera.image_height_px);

        // レンズの上の点のサンプリング確率を計算
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(x0_x1), normalize(camera.imagesensor_dir)) / x0_x1.length_squared());
        total_pdf_A *= now_sampled_pdf_A;

        // ジオメトリターム
        const double G =  dot(normalize(x0_x1), normalize(camera.imagesensor_dir)) * dot(normalize(-1.0 * x0_x1), previous_normal) / x0_x1.length_squared();
        MC_throughput = G * MC_throughput;

        // レンズ上に生成された点を頂点リストに追加（基本的に使わない）
        vertices.push_back(Vertex(position_on_lens, normalize(camera.imagesensor_dir), normalize(camera.imagesensor_dir), -1, Vertex::OBJECT_TYPE_LENS, total_pdf_A, MC_throughput));

        // 幾何的な係数計算 + センサーセンシティビティを計算してモンテカルロ積分して最終的な結果を得る
        const Color result = (camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput) / total_pdf_A;
        return LighttracingResult(result, x, y, true);
    }
    if (!scene_hit)
        break;
```

ライトトレーシングによるサンプリングの例

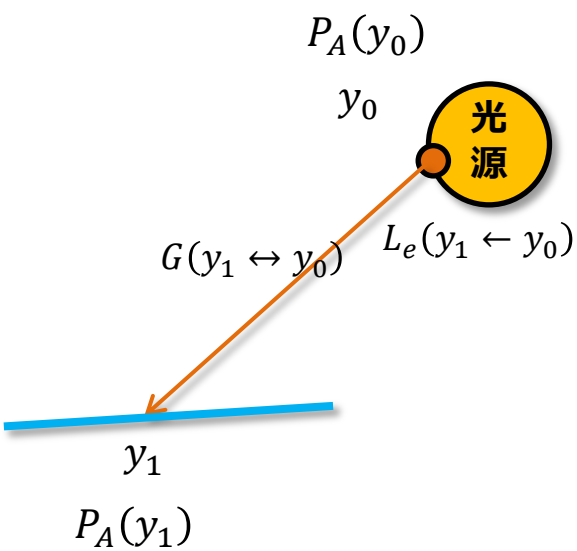
- レンズ上の点x_0の面積測度に関する確率密度を計算する。これは、$P_\sigma(y_1 \leftarrow y_0)$の測度を面積測度に変換することで行う。
  - パストレーシングの時と同様にして測度を変換する。
- また、G項も計算する。
- さらに、イメージセンサのセンシティビティ諸々を含めた項、W'も計算する。
  - この段階のトータルのコントリビューション（モンテカルロスループット/パスの確率密度）も計算しておく。




```
for (;;) {
    Intersection intersection;
    const bool scene_hit = intersect_scene(now_ray, &intersection);

    // レンズと交差判定
    Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
    const double lens_t = camera.intersect_lens(now_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
    if (kEPS < lens_t && lens_t < intersection.hitpoint.distance) {
        // レイがレンズにヒット&イメージセンサにヒット
        const Vec x0_xI = position_on_imagesensor - position_on_lens;
        const Vec x0_xV = position_on_objectplane - position_on_lens;
        const Vec x0_x1 = now_ray.org - position_on_lens;

        // イメージセンサ（ピクセル上）の位置、入射方向によって寄与率が変わる
        //  （が、今回はWを定数にしているのでは寄与率は変わらない）
        int x = (int)uv_on_imagesensor.x;
        int y = (int)uv_on_imagesensor.y;
        clamp(x, 0, camera.image_width_px);
        clamp(y, 0, camera.image_height_px);

        // レンズの上の点のサンプリング確率を計算
        const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(x0_x1), normalize(camera.imagesensor_dir)) / x0_x1.length_squared());
        total_pdf_A *= now_sampled_pdf_A;

        // ジオメトリターム
        const double G =  dot(normalize(x0_x1), normalize(camera.imagesensor_dir)) * dot(normalize(-1.0 * x0_x1), previous_normal) / x0_x1.length_squared();
        MC_throughput = G * MC_throughput;

        // レンズ上に生成された点を頂点リストに追加（基本的に使わない）
        vertices.push_back(Vertex(position_on_lens, normalize(camera.imagesensor_dir), normalize(camera.imagesensor_dir), -1, Vertex::OBJECT_TYPE_LENS, total_pdf_A, MC_throughput));

        // 幾何的な係数計算 + センサーセンシティビティを計算してモンテカルロ積分して最終的な結果を得る
        const Color result = (camera.W_dash(x0_xV, x0_xI, x0_x1) * MC_throughput) / total_pdf_A;
        return LighttracingResult(result, x, y, true);
    }
    if (!scene_hit)
        break;
```

ライトトレーシングによるサンプリングの例

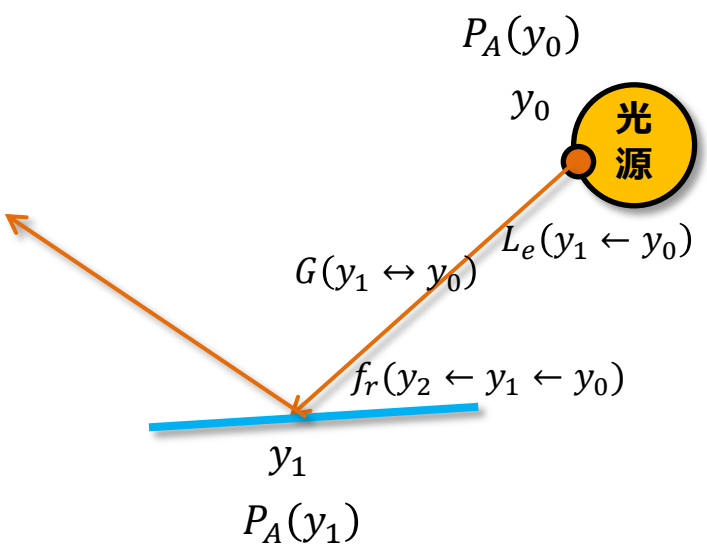
- 飛ばしたレイがレンズとヒットしなかった場合、シーンとの交点を新しい頂点とする。
- ロシアンルーレットも行う。
- さらに、新しい頂点の面積測度に関する確率密度$P_A(y_1)$を、立体角測度に関する確率密度$P_\sigma(y_1 \leftarrow y_0)$から求める。
- G項も求め、スループットに乗算する。




```
if (!scene_hit)
    break;

const Sphere &now_object = spheres[intersection.object_id];
const Hitpoint &hitpoint = intersection.hitpoint;
const Vec orienting_normal = dot(hitpoint.normal , now_ray.dir) < 0.0 ? hitpoint.normal: (-1.0 * hitpoint.normal); // 交差位置の法線（物体からのレイの入出を考慮）
const double russian_roulette_probability = russian_roulette(now_object);
// ロシアンルーレット
if (rnd->next01() >= russian_roulette_probability) {
    break;
}

// 新しい頂点がサンプリングされたので、トータルの確率密度に乗算する
total_pdf_A *= russian_roulette_probability;

const Vec to_next_vertex = now_ray.org - hitpoint.position;
// 新しい頂点をサンプリングするための確率密度関数は立体角測度に関するものであったため、これを面積測度に関する確率密度関数に変換する
const double now_sampled_pdf_A = now_sampled_pdf_omega * (dot(normalize(to_next_vertex), orienting_normal) / to_next_vertex.length_squared());
// 全ての頂点をサンプリングする確率密度の総計を出す
total_pdf_A *= now_sampled_pdf_A;

// ジオメトリターム（G項）
const double G =  dot(normalize(to_next_vertex), orienting_normal) * dot(normalize(-1.0 * to_next_vertex), previous_normal) / to_next_vertex.length_squared();
MC_throughput = G * MC_throughput;

// 新しい頂点を頂点リストに追加する
vertices.push_back(Vertex(hitpoint.position, orienting_normal, hitpoint.normal, intersection.object_id,
        now_object.reflection_type == REFLECTION_TYPE_DIFFUSE ? Vertex::OBJECT_TYPE_DIFFUSE :Vertex::OBJECT_TYPE_SPECULAR,
        total_pdf_A, MC_throughput));
```

ライトトレーシングによるサンプリングの例

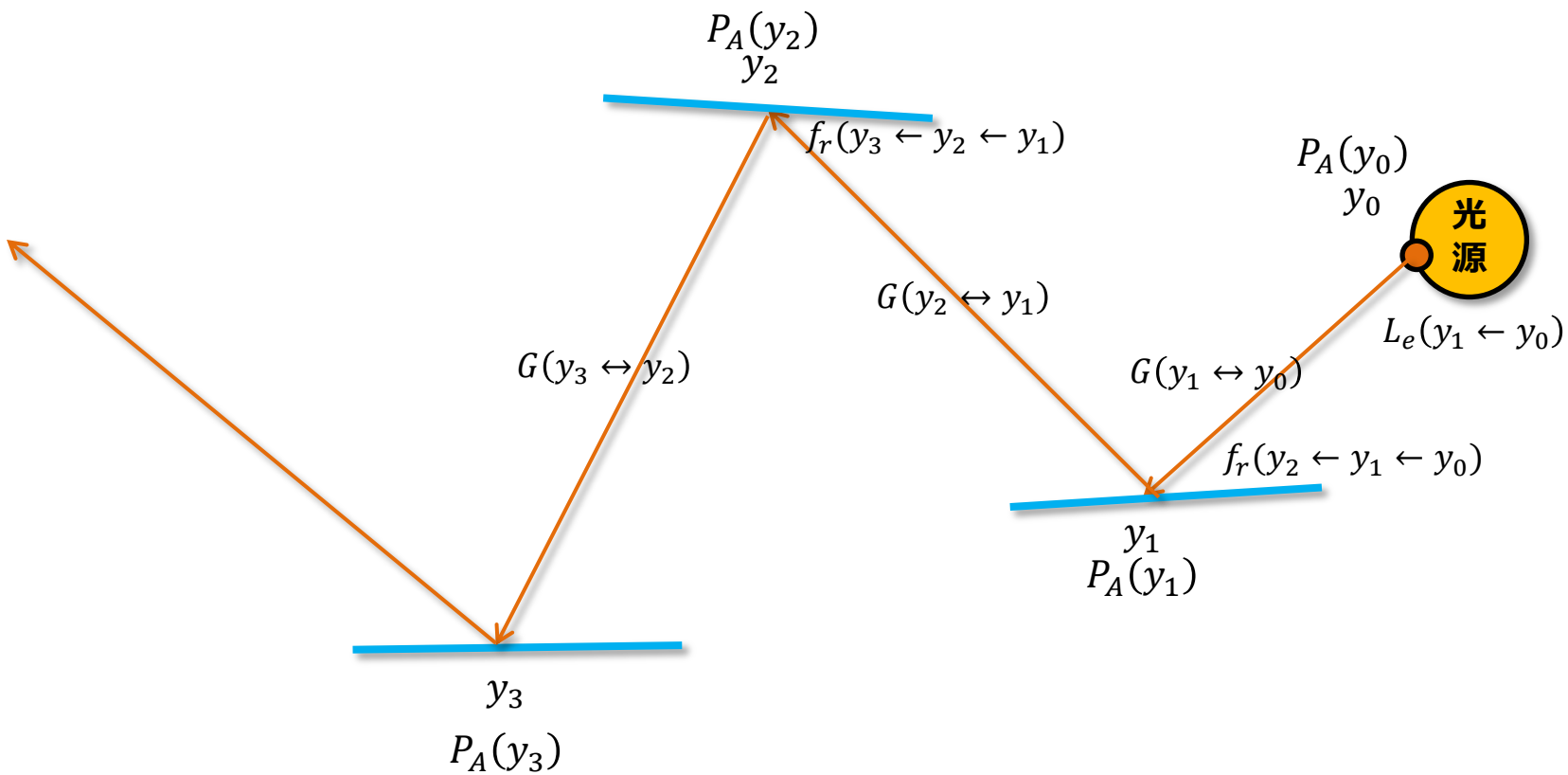
- 現在の頂点の材質ごとに、次の頂点をサンプリングするための方向のサンプリング方法も変わる。
- これは、パストレーシングの時と基本的に同じ処理になる。
  - パストレーシングのときと同様、スループットにBRDFを乗算していくことになる。
  - ただし、nnt2については光の進行方向の違いにより、若干処理が変わる。




```
// 材質ごとの処理
// 次の頂点をサンプリングするために、新しいレイの方向を決める。
switch (now_object.reflection_type) {
// 完全拡散面
case REFLECTION_TYPE_DIFFUSE: {
    // 次の方向をサンプリング
    const Vec dir = sample_hemisphere_cos_term(orienting_normal, rnd, &now_sampled_pdf_omega);
    now_ray = Ray(hitpoint.position, dir);
    // スループット調整 (dir <- hitpoint.position <- 一個前 の情報がそろっているのでBRDF評価できる)
    MC_throughput = multiply(now_object.color / kPI, MC_throughput);
} break;

// 完全鏡面
case REFLECTION_TYPE_MIRROR: {
    // 完全鏡面なのでレイの反射方向は決定的。
    // スペキュラ面におけるサンプリング確率密度はDiracのδ関数の形をとるが、分母とキャンセルされるため、係数のみ与える。
    now_sampled_pdf_omega = 1.0;
    now_ray = Ray(hitpoint.position, reflection_vector(now_ray.dir, hitpoint.normal));

    // スループット調整
    // スペキュラBRDFはδ/cosθになる。（立体角測度に関する関数）
    // また、このδはpdf_omegaのδとキャンセルされる。（よって、δの扱いは無視してよい。now_sampled_pdf_omega = 1.0にしたのもそのため）
    // なお、pdf_omegaはδになる。（立体角測度に関する確率密度）
    // BRDFはcosθをかけて半球積分すると1、pdf_omegaはそのまま半球積分すると1になることからこのようになる。
    MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
} break;

// ガラス
case REFLECTION_TYPE_GLASS: {
    const bool into = dot(hitpoint.normal, orienting_normal) > 0.0; // レイがオブジェクトから出るのか、入るのか

    Vec reflection_dir, refraction_dir;
    double fresnel_reflectance, fresnel_transmittance;
    if (check_refraction(
        into, hitpoint.position, now_ray.dir, hitpoint.normal, orienting_normal,
        &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
        // 屈折 + 反射
        if (rnd->next01() < reflection_probability) { // 反射
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, reflection_dir);
            MC_throughput = fresnel_reflectance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= reflection_probability;
        } else { // 屈折
            const double nnt2 =
                pow(into ?
                refractive_index_of_object / refractive_index_of_vaccum :
                refractive_index_of_vaccum / refractive_index_of_object, 2.0);
            now_sampled_pdf_omega = 1.0;
            now_ray = Ray(hitpoint.position, refraction_dir);
            MC_throughput = nnt2 * fresnel_transmittance * multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
            total_pdf_A *= 1.0 - reflection_probability;
        }
    } else {
        // 全反射
        now_sampled_pdf_omega = 1.0;
        now_ray = Ray(hitpoint.position, reflection_dir);
        MC_throughput = multiply(now_object.color / dot(normalize(to_next_vertex), orienting_normal), MC_throughput);
        break;
    }
} break;

}
```

ライトトレーシングによるサンプリングの例

- 以上の処理を（ロシアンルーレットで打ち切られるかレンズにヒットするまで）繰り返すことで頂点を生成していく。
- 同時に、モンテカルロ積分に使うためのスループットと確率密度も計算していく。最初の部分以外とレンズ判定以外はパストレーシングとほぼ同じになる。



まとめ

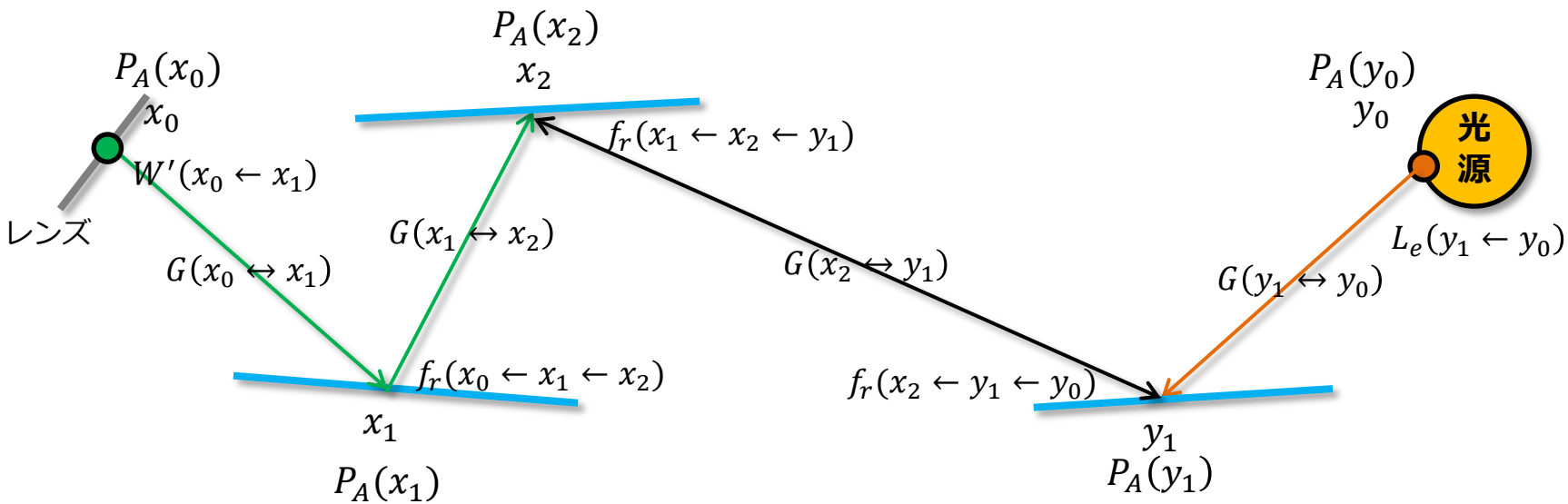
- 解きたいモンテカルロ積分について、
  - 分子（モンテカルロスループット）
    - **光源における放射輝度×それ以降の頂点間のBRDFの積×それ以降の頂点間のG項の積×センサのセンシティビティ等の項（W’）**
  - 分母（確率密度）
    - 各頂点をサンプリングするための**面積測度に関する確率密度の積**
- が必要なので、パストレーシング or ライトトレーシングによってこれを頂点ごとに逐次求めていく。
  - この結果を組み合わせて双方向パストレーシングを実行する。

6. 双方向パストレーシング・実装編その２

パスの接続の例

- パストレーシングによってサンプリングされたパス$x_0x_1x_2$とライトトレーシングによってサンプリングされたパスy_1y_0を接続すると、以下のような五頂点分のモンテカルロ積分を解くことに相当する。
  - total_pdf_Aや、MC_throughputは、以下を計算するためのものだった。
  - 下の式の緑の部分の分母と分子が、generate_vertices_by_pathtracing ()によって得られる頂点リストの要素であるVertex構造体のメンバ、total_pdf_A とthroughputに対応する。
    - この場合は、x_2の分までなので、3番目の要素のメンバになる。
  - 同様にオレンジの部分はgenerate_vertices_by_lighttracing ()によって得られる。
  - 黒い部分は頂点の接続によって新たに計算される。

$$\frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)f_r(x_1 \leftarrow x_2 \leftarrow y_1)G(x_2 \leftrightarrow y_1)f_r(x_2 \leftarrow y_1 \leftarrow y_0)G(y_1 \leftrightarrow y_0)L_e(y_1 \leftarrow y_0)}{P_A(x_0)P_A(x_1)P_A(x_2)P_A(y_0)P_A(y_1)}$$



パスの接続とモンテカルロ積分

- 以下、具体的な例。
- **頂点数が3**のカメラ側のパスと**頂点数が2**のライト側のパスを接続した場合、新たに**頂点数5**のパスが得られる。
- **頂点数が2**のカメラ側のパスと**頂点数が3**のライト側のパスを接続した場合も、新たに**頂点数5**のパスが得られる。
- この二つのパスは、**頂点数5の積分に対するモンテカルロ積分を異なるサンプリング戦略でサンプリングして得られたサンプル（パス）である**、と考える。
- よって、それぞれのパスによるモンテカルロ積分を独立に行い、その結果をマルチプルインポータンスサンプリングによって重み付けして統合する必要がある。
  - 同じ積分に対するモンテカルロ積分を、異なるサンプリング戦略で独立に解いた場合、各々の結果をマルチプルインポータンスサンプリングによって統合するのだった。

$$\int_M \int_M \int_M \int_M \int_{Lens} W'(x_0 \leftarrow x_1) \prod_{i=0}^{3} G(x_i \leftrightarrow x_{i+1}) \prod_{i=0}^{2} f_r(x_i \leftarrow x_{i+1} \leftarrow x_{i+2})\, L_e(x_3 \leftarrow x_4) dA(x_0)\, dA(x_1)\, dA(x_2)\, dA(x_3)\, dA(x_4)$$

頂点数5の積分

同じ頂点数の同じ積分を解いているが、頂点の組み合わせの数だけ別のモンテカルロ積分が存在し、それらの結果をMISによって統合する。

カメラ側のパス：**頂点数0**
光源側のパス：**頂点数5**

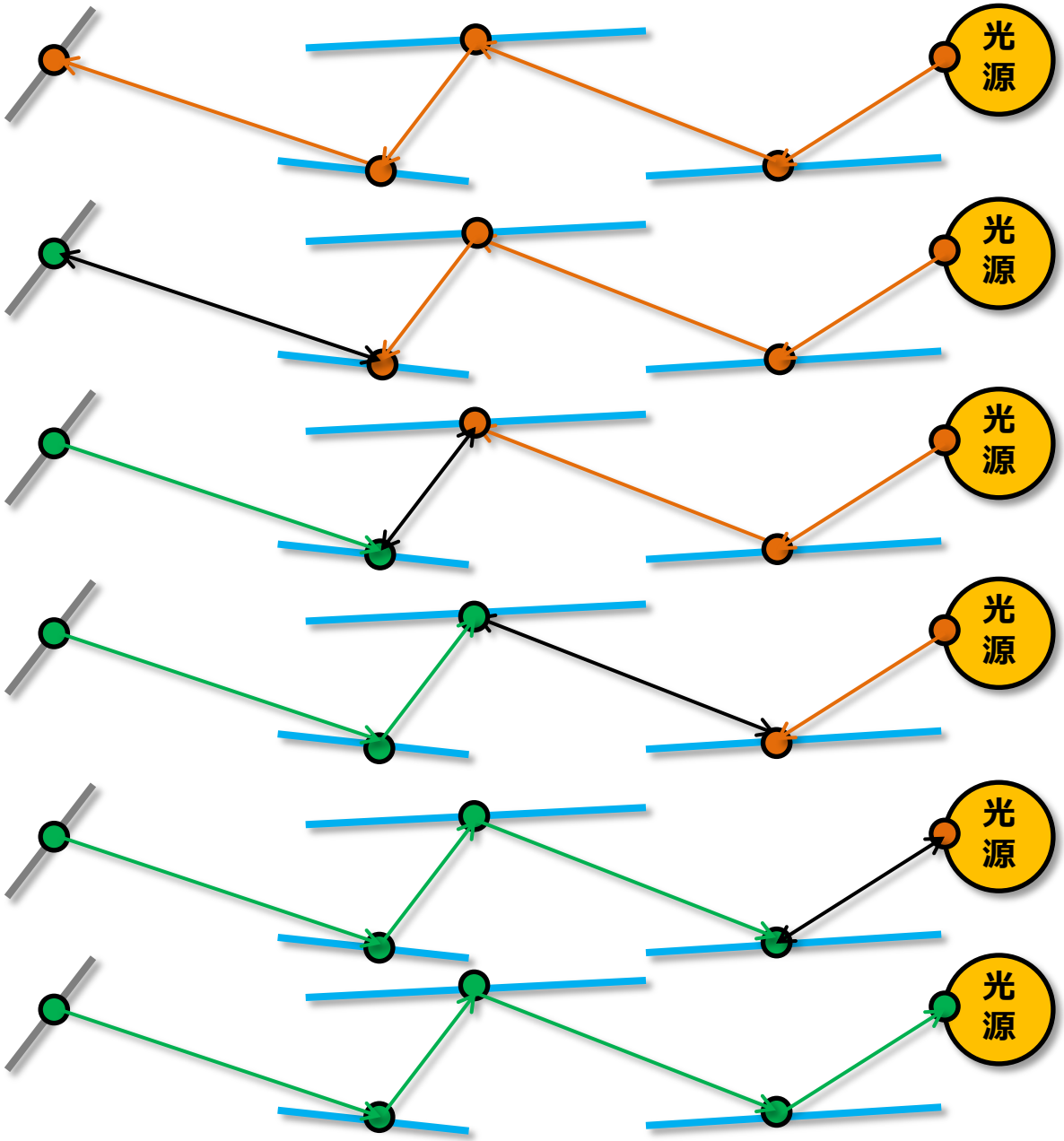
カメラ側のパス：**頂点数1**
光源側のパス：**頂点数4**

カメラ側のパス：**頂点数2**
光源側のパス：**頂点数3**

カメラ側のパス：**頂点数3**
光源側のパス：**頂点数2**

カメラ側のパス：**頂点数4**
光源側のパス：**頂点数1**

カメラ側のパス：**頂点数5**
光源側のパス：**頂点数0**



bidirectional_pathtracing.h

- 頂点同士を接続し、最終的な画像を得るための実装を説明していく。

```cpp
#ifndef _BIDIRECTIONAL_PATHTRACING_H_
#define _BIDIRECTIONAL_PATHTRACING_H_

#include "random.h"
#include "scene.h"
#include "util.h"
#include "camera.h"
#include "specular.h"
#include "sampling.h"
#include "camera.h"
#include "vertex.h"

#include "pathtracing.h"
#include "lighttracing.h"

#include <assert.h>
#include <iostream>

namespace edubpt {

// 頂点from から頂点next をサンプリングしたとすると、面積測度に関するサンプリング確率密度を計算する。
// 今回はLighttracing もPathtracing も同じ実装でサンプリングを行うためこの関数で良い。
// サンプリング範囲が異なるなら、それぞれのための関数を用意する必要がある。
double calc_pdf_A(const Camera &camera, const std::vector<const Vertex*> &vs, const int prev_from_idx, const int from_idx, const int next_idx) {
    const Vertex &from_vertex = *vs[from_idx];
    const Vertex &next_vertex = *vs[next_idx];
    Vertex const *prev_from_vertex = NULL;
    if (0 == prev_from_idx && prev_from_idx < vs.size())
        prev_from_vertex = vs[prev_from_idx];

    const Vec to = next_vertex.position - from_vertex.position;
    const Vec normalized_to = normalize(to);
    double pdf = 0.0;

    // 頂点from のオブジェクトの種類によって、その点の頂点のサンプリング確率密度の計算方法が変わる。
    switch (from_vertex.type) {
    case Vertex::OBJECT_TYPE_LIGHT:
    case Vertex::OBJECT_TYPE_DIFFUSE:
        pdf = sample_hemisphere_pdf_omega(from_vertex.orienting_normal, normalized_to);
        break;
    case Vertex::OBJECT_TYPE_LENS:
        {
            // レンズとの点からシーンとの点をサンプリングするときの面積測度に関する確率密度を計算。
            // イメージセンサとの点のサンプリング確率密度を光に変換する。
            const Ray test_ray(next_vertex.position, -1.0 * normalized_to);
            Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
            const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
            if (kEPS < lens_t) {
                const Vec x0_x1 = position_on_imagesensor - position_on_lens;
                const Vec x0_xv = position_on_objectplane - position_on_lens;
                const Vec x0_x1 = test_ray.org - position_on_lens;

                const double P_Image = 1.0 / (camera.imagesensor_width * camera.imagesensor_height);
                const double PA_x1 = camera.P_Image_to_PA_x1P_Image, x0_xV, x0_x1, next_vertex.orienting_normal);
                return PA_x1;
            }
            return 0.0;
        }
        break;
    case Vertex::OBJECT_TYPE_SPECULAR:
        if (spheres[from_vertex.objectID].reflection_type == REFLECTION_TYPE_GLASS) {
            // 始点がガラスに存在した場合、どのときに屈折によってサンプリングされたのか、反射によってサンプリングされたかによって
            // その頂点のサンプリング確率が変わるので、その判定を行う。
            if (prev_from_vertex != NULL) {
                // from頂点のひとつ前の頂点に基づいて、物体に入りはしているのか、それとも出て行くのかを判定する。
                const Vec into_from_vertex_dir = normalize(from_vertex.position - prev_from_vertex->position);
                const bool into = dot(into_from_vertex_dir, from_vertex.object_normal) < 0.0; // prev_from_vertexからfrom_vertexへのベクトルが、スペキュラ物体に入るのか、それとも出るのか
                const Vec from_new_orienting_normal = into ? from_vertex.object_normal : -1.0 * from_vertex.object_normal;

                // 全反射しているのか、していないのか判定
                Vec reflection_dir, refraction_dir;
                double fresnel_reflectance, fresnel_transmittance;
                if (check_refraction(
                    into, from_vertex.position, into_from_vertex_dir, from_vertex.object_normal, from_new_orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                    // 反射 or 屈折
                    pdf = dot(from_new_orienting_normal, normalized_to) > 0.0 ? reflection_probability : 1.0 - reflection_probability;
                } else {
                    // 全反射
                    pdf = 1.0;
                }
            }
        } else {
            // 鏡面
            pdf = 1.0;
        }
        break;
    }

    // 次の頂点の法線を、現在の頂点からの方向ベクトルに基づいて求める
    const Vec next_new_orienting_normal = dot(to, next_vertex.object_normal) < 0.0 ? next_vertex.object_normal : -1.0 * next_vertex.object_normal;
    // 立体角測度に関する確率密度を面積測度に関する確率密度へ変換
    return pdf * (dot(-1.0 * normalized_to, next_new_orienting_normal) / to.length_squared());
}

double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_p(num_eye_vertex + num_light_vertex);
    const double PA_x0 = sample_sphere_pdf_A(spheres[light[0].radius]); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, ... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を掛けずに入れる。
    pi1_p[0] = PA_x0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_p[i] = a / b;
    }
    pi1_p[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; 1 <= k; ++i) {
        p[i + 1] = p[i] * pi1_p[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_p[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
```

```cpp
struct BidirectionalPathtracingResult {
    struct Sample {
        int imagebuffer_x, imagebuffer_y;
        Color value;
        bool start_from_pixel;

        Sample(const int imagebuffer_x, const int imagebuffer_y, const Color &value, const bool start_from_pixel) :
        imagebuffer_x(imagebuffer_x), imagebuffer_y(imagebuffer_y), value(value), start_from_pixel(start_from_pixel) {}
    };
    std::vector<Sample> samples;
};
BidirectionalPathtracingResult bidirectional_pathtracing(const Camera &camera, const int imagebuffer_x, const int imagebuffer_y, Random *rnd) {
    BidirectionalPathtracingResult bpt_result;

    std::vector<Vertex> eye_vs, light_vs;
    const PathtracingResult pt_result = generate_vertices_by_pathtracing(camera, imagebuffer_x, imagebuffer_y, rnd, eye_vs);
    const LighttracingResult lt_result = generate_vertices_by_lighttracing(camera, rnd, light_vs);

    // num_light_vertex == 0のとき、カメラ側からのパスが光源にヒットしていれば、それをそのまま使用する。
    if (pt_result.is_light_hit) {
        const double mis_weight = calc_mis_weight(camera, eye_vs[eye_vs.size()-1].total_pdf_A, eye_vs, light_vs, (const int)eye_vs.size(), 0);
        const Color result = mis_weight * pt_result.value;
        bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(imagebuffer_x, imagebuffer_y, result, true));
    }
    // num_eye_vertex == 0のとき、光源側からのパスがレンズにヒットしていれば、それをそのまま使用する。
    if (lt_result.is_lens_hit) {
        const double mis_weight = calc_mis_weight(camera, light_vs[light_vs.size()-1].total_pdf_A, eye_vs, light_vs, 0, (const int)light_vs.size());
        const int lx = lt_result.imagebuffer_x, ly = lt_result.imagebuffer_y;
        const Color result = mis_weight * lt_result.value;
        bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(lx, ly, result, false));
    }

    // 各頂点を接続する。
    // eyeサブパス:カメラ側のパスの頂点の一部を使って作られるパス
    // lightサブパス:光源側のパスの頂点の一部を使って作られるパス
    for (int num_eye_vertex = 1; num_eye_vertex <= eye_vs.size(); ++num_eye_vertex) {
        for (int num_light_vertex = 1; num_light_vertex <= light_vs.size(); ++num_light_vertex) {
            int target_x = imagebuffer_x, target_y = imagebuffer_y;
            const Vertex &eye_end = eye_vs[num_eye_vertex - 1];
            const Vertex &light_end = light_vs[num_light_vertex - 1];

            // トータルの確率密度計算
            const double total_pdf_A = eye_end.total_pdf_A * light_end.total_pdf_A;
            if (total_pdf_A == 0)
                continue;

            // MCスループット
            Color eye_throughput = eye_end.throughput;
            Color light_throughput = light_end.throughput;
            // 頂点を結ぶことで新しく導入される項(幾何項のBRDFとG項)
            Color connection_throughput(1, 1, 1);

            // num_light_vertex == 1のとき、非完全拡散光源の場合は相手の頂点の位置が乗ってMCスループットが変化するのでその分ここで光源からの放射輝度値を計算する。
            // 相当は完全拡散光源なので無視してemissionの値を入れる。
            if (num_light_vertex == 1) {
                light_throughput = spheres[light_vs[0].objectID].emission;
            }

            // 陰影線でレイトレーシングしておく。
            Intersection intersection;
            const Vec light_end_to_eye_end = eye_end.position - light_end.position;
            const Ray test_ray = Ray(light_end.position, normalize(light_end_to_eye_end));
            intersect_scene(test_ray, &intersection);

            // eyeサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
            // eyeサブパスの場合の計算の種類によって変わる。
            switch (eye_end.type) {
            case Vertex::OBJECT_TYPE_DIFFUSE:
                connection_throughput = multiply(connection_throughput, spheres[eye_end.objectID].color / kPI);
                // 陰影線と衝突の時間で遮蔽されるかどうかを判定する。遮蔽されていると処理終わり。
                if (!(intersection.hitpoint.position - eye_end.position).length() == kEPS) {
                    continue;
                }
                break;
            case Vertex::OBJECT_TYPE_LENS:
                // ここにくるのはnum_eye_vertex == 1のときだけ
                {
                    // lightサブパスを屈折レンズにつなげるパターン
                    Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
                    const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
                    if (kEPS < lens_t && lens_t < intersection.hitpoint.distance) {
                        // レイがレンズにヒットしてイメージセンサにヒット
                        const Vec x0_x1 = position_on_imagesensor - position_on_lens;
                        const Vec x0_xv = position_on_objectplane - position_on_lens;
                        const Vec x0_x1 = test_ray.org - position_on_lens;

                        target_x = (int)uv_on_imagesensor.x;
                        target_y = (int)uv_on_imagesensor.y;
                        clamp(target_x, 0, camera.image_width_px);
                        clamp(target_y, 0, camera.image_height_px);

                        connection_throughput = camera.W_dash(x0_xV, x0_x1, x0_x1) * connection_throughput;
                    } else {
                        // lightサブパスを屈折レンズにつなげようとしたが、遮蔽されたりイメージセンサにヒットしなかった場合、終わり。
                        continue;
                    }
                }
                break;
            // eyeサブパスの場合がスペキュラ光源（反射等）だった場合は重みが忙にならパス全体の寄与も忙になるので、処理終わり。
            case Vertex::OBJECT_TYPE_LIGHT:
            case Vertex::OBJECT_TYPE_SPECULAR:
                continue;
            }

            // lightサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
            // lightサブパスの場合の計算の種類によって変わる。
            switch (light_end.type) {
            case Vertex::OBJECT_TYPE_DIFFUSE:
                connection_throughput = multiply(connection_throughput, spheres[light_end.objectID].color / kPI);
                break;
            case Vertex::OBJECT_TYPE_LIGHT:
                // 光源が完全拡散と仮定しているため、ライトトレーシングの終 最初の頂点以外は光源上に頂点を生成されず、
                // num_light_vertex == 1以外でここに入ってくることは無い。
                break;
                // lightサブパスの場合がスペキュラやレンズだった場合は重みが忙になりパス全体の寄与も忙になるので、処理終わり。
            case Vertex::OBJECT_TYPE_LENS:
            case Vertex::OBJECT_TYPE_SPECULAR:
                continue;
            }

            // 陰影線のジオメトリターム
            const double G = {
                std::max(dot(normalize(-1.0*light_end_to_eye_end), eye_end.orienting_normal), 0.0) *
                std::max(dot(normalize(light_end_to_eye_end), light_end.orienting_normal), 0.0)) /
                light_end_to_eye_end.length_squared();
            connection_throughput = G * connection_throughput;

            // MISの重み計算する。
            const double mis_weight = calc_mis_weight(camera, total_pdf_A, eye_vs, light_vs, num_eye_vertex, num_light_vertex);

            if (isnan(mis_weight)) {
                continue;
            }

            // 最終的なモンテカルロコントリビューション =
            // MIS重み * 頂点を接続することで新しく導入される項 * eyeサブパススループット * lightサブパススループット / パスのサンプリング確率密度の総計
            const Color result = mis_weight * multiply(multiply(connection_throughput, eye_throughput), light_throughput) / total_pdf_A;
            bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(target_x, target_y, result, num_eye_vertex <= 1 ? false : true));
        }
    }

    return bpt_result;
}
};
#endif
```

双方向パストレーシング

- bidirectional_pathtracing()によって、カメラパラメータ、現在のピクセル座標（I_jのjに対応する(x, y)）を受け取り、双方向パストレーシングによって画素値を計算する。
- まず、パストレーシングとライトトレーシングによって、カメラ側、光源側より頂点をサンプリングしてパスを生成する。

```
BidirectionalPathtracingResult bidirectional_pathtracing(const Camera &camera, const int imagebuffer_x, const int imagebuffer_y, Random *rnd) {
    BidirectionalPathtracingResult bpt_result;

    std::vector<Vertex> eye_vs, light_vs;
    const PathtracingResult  pt_result = generate_vertices_by_pathtracing(camera, imagebuffer_x, imagebuffer_y, rnd, eye_vs);
    const LighttracingResult lt_result = generate_vertices_by_lighttracing(camera, rnd, light_vs);

    // num_light_vertex == 0のとき、カメラ側からのパスが光源にヒットしていれば、それをそのまま使用する。
    if (pt_result.is_light_hit) {
        const double mis_weight = calc_mis_weight(camera, eye_vs[eye_vs.size()-1].total_pdf_A, eye_vs, light_vs, (const int)eye_vs.size(), 0);
        const Color result = mis_weight * pt_result.value;
        bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(imagebuffer_x, imagebuffer_y, result, true));
    }
    // num_eye_vertex == 0のとき、光源側からのパスがレンズにヒットしていれば、それをそのまま使用する。
    if (lt_result.is_lens_hit) {
        const double mis_weight = calc_mis_weight(camera, light_vs[light_vs.size()-1].total_pdf_A, eye_vs, light_vs, 0, (const int)light_vs.size());
        const int lx = lt_result.imagebuffer_x, ly = lt_result.imagebuffer_y;
        const Color result =  mis_weight * lt_result.value;
        bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(lx, ly, result, false));
    }
```

双方向パストレーシング

- パストレーシングによるパスが直接光源にヒットしていた場合、その結果をそのまま画素値の計算に使う。（この時、ライトトレーシングによる頂点数は0）
- また、ライトトレーシングによるパスが直接レンズにヒットしていた場合、その結果をそのまま対応する画素値の計算に使う。（この時、パストレーシングによる頂点数は0）
- 既に説明したように、それぞれのパスによるモンテカルロ積分に対するMISの重みを計算する。後で詳しく解説。

```
BidirectionalPathtracingResult bidirectional_pathtracing(const Camera &camera, const int imagebuffer_x, const int imagebuffer_y, Random *rnd) {
    BidirectionalPathtracingResult bpt_result;

    std::vector<Vertex> eye_vs, light_vs;
    const PathtracingResult  pt_result = generate_vertices_by_pathtracing(camera, imagebuffer_x, imagebuffer_y, rnd, eye_vs);
    const LighttracingResult lt_result = generate_vertices_by_lighttracing(camera, rnd, light_vs);

    // num_light_vertex == 0のとき、カメラ側からのパスが光源にヒットしていれば、それをそのまま使用する。
    if (pt_result.is_light_hit) {
        const double mis_weight = calc_mis_weight(camera, eye_vs[eye_vs.size()-1].total_pdf_A, eye_vs, light_vs, (const int)eye_vs.size(), 0);
        const Color result = mis_weight * pt_result.value;
        bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(imagebuffer_x, imagebuffer_y, result, true));
    }
    // num_eye_vertex == 0のとき、光源側からのパスがレンズにヒットしていれば、それをそのまま使用する。
    if (lt_result.is_lens_hit) {
        const double mis_weight = calc_mis_weight(camera, light_vs[light_vs.size()-1].total_pdf_A, eye_vs, light_vs, 0, (const int)light_vs.size());
        const int lx = lt_result.imagebuffer_x, ly = lt_result.imagebuffer_y;
        const Color result =  mis_weight * lt_result.value;
        bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(lx, ly, result, false));
    }
```

頂点間の接続

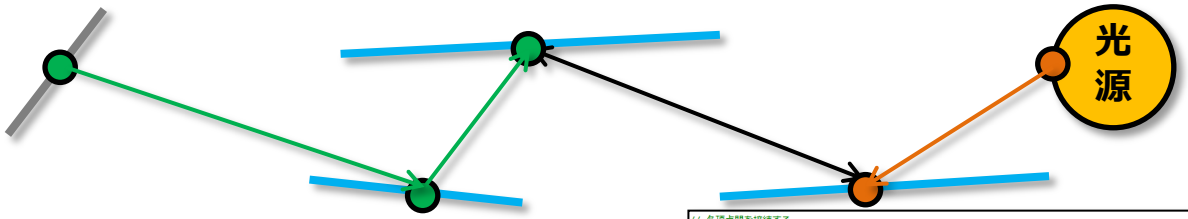
- 以後、カメラ側のパスの一部を使って作られるパスをeyeサブパス、光源側のパスの一部を使って作られるパスをlightサブパスと呼ぶことにする。
- eyeサブパスとlightサブパスを接続するのが今後の処理になる。全ての接続の組み合わせを試す。
  - これによってさまざまな頂点数のパスが得られ、さまざまな頂点数の積分に対するモンテカルロ積分の結果を得られる。
- 例えば、カメラ側のパスの**頂点数が7**、光源側のパスの**頂点数が5**だとすると、eyeサブパスは**頂点数1から7の7種類**、lightサブパスは**頂点数1から5の5種類**が考えられる。この時、接続の組み合わせは**7×5=35種類**となる。
  - ひとつ前のスライドの処理は、eyeサブパスやlightサブパスの頂点数が0の場合の処理であった。

```
// 各頂点間を接続する。
// eyeサブパス:カメラ側のパスの頂点の一部を使って作られるパス
// lightサブパス:光源側のパスの頂点の一部を使って作られるパス
for (int num_eye_vertex = 1; num_eye_vertex <= eye_vs.size(); ++num_eye_vertex) {
    for (int num_light_vertex = 1; num_light_vertex <= light_vs.size(); ++num_light_vertex) {
        int target_x = imagebuffer_x, target_y = imagebuffer_y;
        const Vertex &eye_end = eye_vs[num_eye_vertex - 1];
        const Vertex &light_end = light_vs[num_light_vertex - 1];

        // トータルの確率密度計算
        const double total_pdf_A = eye_end.total_pdf_A * light_end.total_pdf_A;
        if (total_pdf_A == 0)
            continue;

        // MCスループット
        Color eye_throughput = eye_end.throughput;
        Color light_throughput = light_end.throughput;
        // 頂点を接続することで新しく導入される項（基本的にBRDFとG項）
        Color connection_throughput(1, 1, 1);

        // num_light_vertex == 1のとき、非完全拡散光源の場合は相手の頂点の位置次第でMCスループットが変化するため改めて光源からの放射輝度値を計算する。
        // 今回は完全拡散光源なので単純にemissionの値を入れる。
        if (num_light_vertex == 1) {
            light_throughput = spheres[light_vs[0].objectID].emission;
        }

        // 端点間でレイトレーシングしておく。
        Intersection intersection;
        const Vec light_end_to_eye_end = eye_end.position - light_end.position;
        const Ray test_ray = Ray(light_end.position, normalize(light_end_to_eye_end));
        intersect_scene(test_ray, &intersection);
```

頂点間の接続

- eyeサブパスの頂点数とlightサブパスの頂点数が決まったら、いよいよサブパスを接続する。
- まず、接続したパスのトータルのサンプリング確率密度を計算する。これは、各サブパスのサンプリング確率密度を乗算すればよい。
  - 接続したパスのサンプリング確率密度とは、パスを構成する頂点の面積測度に関する確率密度の積であった。そして、各サブパスのサンプリング確率密度は、サブパスを構成する頂点の確率密度の積であった。よって、サブパスの確率密度を乗算すればよいことになる。
  - 例えば、eyeサブパスの**頂点数3**、lightサブパスの**頂点数2**なら、以下のような式の分母を計算していることに相当する。

$$\frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)f_r(x_1 \leftarrow x_2 \leftarrow y_1)G(x_2 \leftrightarrow y_1)f_r(x_2 \leftarrow y_1 \leftarrow y_0)G(y_1 \leftrightarrow y_0)L_e(y_1 \leftarrow y_0)}{P_A(x_0)P_A(x_1)P_A(x_2)P_A(y_0)P_A(y_1)}$$

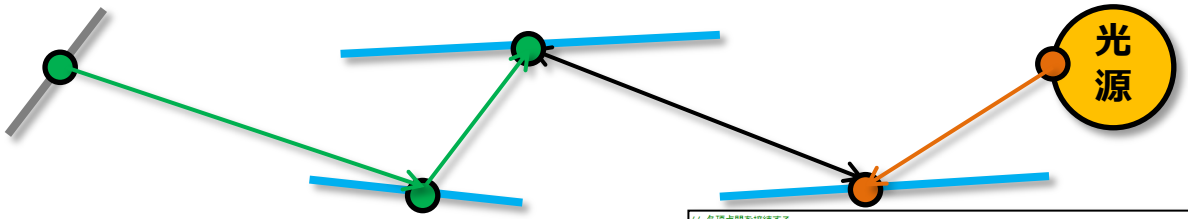



```
// 各頂点間を接続する。
// eyeサブパス:カメラ側のパスの頂点の一部を使って作られるパス
// lightサブパス:光源側のパスの頂点の一部を使って作られるパス
for (int num_eye_vertex = 1; num_eye_vertex <= eye_vs.size(); ++num_eye_vertex) {
    for (int num_light_vertex = 1; num_light_vertex <= light_vs.size(); ++num_light_vertex) {
        int target_x = imagebuffer_x, target_y = imagebuffer_y;
        const Vertex &eye_end = eye_vs[num_eye_vertex - 1];
        const Vertex &light_end = light_vs[num_light_vertex - 1];

        // トータルの確率密度計算
        const double total_pdf_A = eye_end.total_pdf_A * light_end.total_pdf_A;
        if (total_pdf_A == 0)
            continue;

        // MCスループット
        Color eye_throughput = eye_end.throughput;
        Color light_throughput = light_end.throughput;
        // 頂点を接続することで新しく導入される項 (基本的にBRDFとG項)
        Color connection_throughput(1, 1, 1);

        // num_light_vertex == 1のとき、非完全拡散光源の場合は相手の頂点の位置次第でMCスループットが変化するため改めて光源からの放射輝度値を計算する。
        // 今回は完全拡散光源なので単純にemissionの値を入れる。
        if (num_light_vertex == 1) {
            light_throughput = spheres[light_vs[0].objectID].emission;
        }

        // 端点間でレイトレーシングしておく。
        Intersection intersection;
        const Vec light_end_to_eye_end = eye_end.position - light_end.position;
        const Ray test_ray = Ray(light_end.position, normalize(light_end_to_eye_end));
        intersect_scene(test_ray, &intersection);
```

頂点間の接続

- 続いて、モンテカルロ積分を解くためのスループットを求める。各サブパスのスループットは既に求まっているため、サブパスの接続によるスループットを新たに計算すればよい。
  - 例えば、eyeサブパスの**頂点数3**、lightサブパスの**頂点数2**なら、以下のような式の分子の黒い部分を新たに計算する必要がある。今回はconnection_throughputが対応する。
  - 基本的に、eyeサブパス側のBRDFと、lightサブパス側のBRDF、さらに端点間のG項で構成される。
- ただし、lightサブパスの**頂点数が1**の時、非完全拡散光源の場合は相手の頂点の位置次第でスループットが変化するため改めて光源からの放射輝度値を計算する。
  - 今回は完全拡散光源なので単純にemissionの値を入れる。

$$\frac{W'(x_0 \leftarrow x_1)G(x_0 \leftrightarrow x_1)f_r(x_0 \leftarrow x_1 \leftarrow x_2)G(x_1 \leftrightarrow x_2)f_r(x_1 \leftarrow x_2 \leftarrow y_1)G(x_2 \leftrightarrow y_1)f_r(x_2 \leftarrow y_1 \leftarrow y_0)G(y_1 \leftrightarrow y_0)L_e(y_1 \leftarrow y_0)}{P_A(x_0)P_A(x_1)P_A(x_2)P_A(y_0)P_A(y_1)}$$




```
// 各頂点間を接続する。
// eyeサブパス:カメラ側のパスの頂点の一部を使って作られるパス
// lightサブパス:光源側のパスの頂点の一部を使って作られるパス
for (int num_eye_vertex = 1; num_eye_vertex <= eye_vs.size(); ++num_eye_vertex) {
    for (int num_light_vertex = 1; num_light_vertex <= light_vs.size(); ++num_light_vertex) {
        int target_x = imagebuffer_x, target_y = imagebuffer_y;
        const Vertex &eye_end = eye_vs[num_eye_vertex - 1];
        const Vertex &light_end = light_vs[num_light_vertex - 1];

        // トータルの確率密度計算
        const double total_pdf_A = eye_end.total_pdf_A * light_end.total_pdf_A;
        if (total_pdf_A == 0)
            continue;

        // MCスループット
        Color eye_throughput = eye_end.throughput;
        Color light_throughput = light_end.throughput;
        // 頂点を接続することで新しく導入される項 (基本的にBRDFとG項)
        Color connection_throughput(1, 1, 1);

        // num_light_vertex == 1のとき、非完全拡散光源の場合は相手の頂点の位置次第でMCスループットが変化するため改めて光源からの放射輝度値を計算する。
        // 今回は完全拡散光源なので単純にemissionの値を入れる。
        if (num_light_vertex == 1) {
            light_throughput = spheres[light_vs[0].objectID].emission;
        }

        // 端点間でレイトレーシングしておく。
        Intersection intersection;
        const Vec light_end_to_eye_end = eye_end.position - light_end.position;
        const Ray test_ray = Ray(light_end.position, normalize(light_end_to_eye_end));
        intersect_scene(test_ray, &intersection);
```

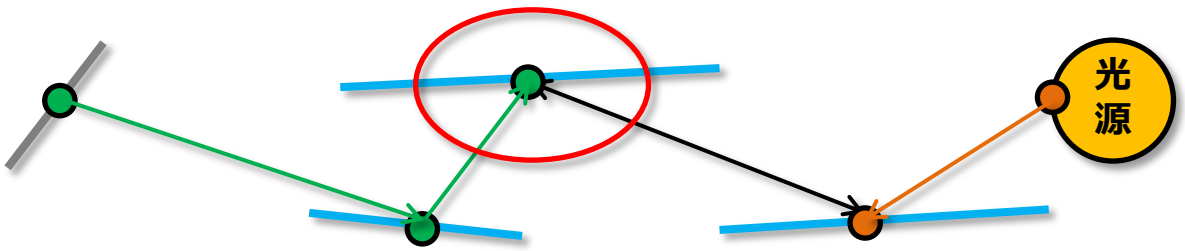
頂点間の接続

- 接続する際、サブパスの端点間に別の物体があれば当然そのパスのスループットは 0 になるので、端点間をレイトレーシングしておく。
- ただし、lightサブパスを直接レンズにつなげるパターンというのがあるため、実際に処理を打ち切るかどうかについてはもう少し後で判定する。

```
// 各頂点間を接続する。
// eyeサブパス:カメラ側のパスの頂点の一部を使って作られるパス
// lightサブパス:光源側のパスの頂点の一部を使って作られるパス
for (int num_eye_vertex = 1; num_eye_vertex <= eye_vs.size(); ++num_eye_vertex) {
    for (int num_light_vertex = 1; num_light_vertex <= light_vs.size(); ++num_light_vertex) {
        int target_x = imagebuffer_x, target_y = imagebuffer_y;
        const Vertex &eye_end = eye_vs[num_eye_vertex - 1];
        const Vertex &light_end = light_vs[num_light_vertex - 1];

        // トータルの確率密度計算
        const double total_pdf_A = eye_end.total_pdf_A * light_end.total_pdf_A;
        if (total_pdf_A == 0)
            continue;

        // MCスループット
        Color eye_throughput = eye_end.throughput;
        Color light_throughput = light_end.throughput;
        // 頂点を接続することで新しく導入される項 (基本的にBRDFとG項)
        Color connection_throughput(1, 1, 1);

        // num_light_vertex == 1のとき、非完全拡散光源の場合は相手の頂点の位置次第でMCスループットが変化するため改めて光源からの放射輝度値を計算する。
        // 今回は完全拡散光源なので単純にemissionの値を入れる。
        if (num_light_vertex == 1) {
            light_throughput = spheres[light_vs[0].objectID].emission;
        }

        // 端点間でレイトレーシングしておく。
        Intersection intersection;
        const Vec light_end_to_eye_end = eye_end.position - light_end.position;
        const Ray test_ray = Ray(light_end.position, normalize(light_end_to_eye_end));
        intersect_scene(test_ray, &intersection);
```

頂点間の接続

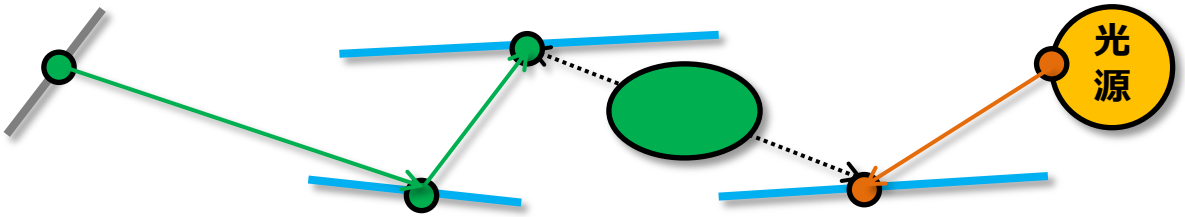
- connection_throughput の計算を行っていく。まず、eyeサブパス側。
- eyeサブパスの端点の材質によって処理が変わる。
- まず、完全拡散面だった場合、 connection_throughputにBRDFを乗算する。




```
// eyeサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
// eyeサブパスの端点の材質の種類によって変わる。
switch (eye_end.type) {
case Vertex::OBJECT_TYPE_DIFFUSE:
    connection_throughput = multiply(connection_throughput, spheres[eye_end.objectID].color / kPI);
    // 端点同士が別の物体で遮蔽されるかどうかを判定する。遮蔽されていたら処理終わり。
    if ((intersection.hitpoint.position - eye_end.position).length() >= kEPS) {
        continue;
    }
    break;
case Vertex::OBJECT_TYPE_LENS:
    // ここに入るのはnum_eye_vertex == 1のときだけ
    {
        // lightサブパスを直接レンズにつなげるパターン
        Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
        const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
        if (kEPS < lens_t && lens_t < intersection.hitpoint.distance) {
            // レイがレンズにヒット&イメージセンサにヒット
            const Vec x0_xI = position_on_imagesensor - position_on_lens;
            const Vec x0_xV = position_on_objectplane - position_on_lens;
            const Vec x0_x1 = test_ray.org - position_on_lens;

            target_x = (int)uv_on_imagesensor.x;
            target_y = (int)uv_on_imagesensor.y;
            clamp(target_x, 0, camera.image_width_px);
            clamp(target_y, 0, camera.image_height_px);

            connection_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * connection_throughput;
        } else {
            // lightサブパスを直接レンズにつなげようとしたが、遮蔽されたりイメージセンサにヒットしなかった場合、終わり。
            continue;
        }
    }
    break;
    // eyeサブパスの端点がスペキュラや光源（反射率0）だった場合は重みがゼロになりパス全体の寄与もゼロになるので、処理終わり。
case Vertex::OBJECT_TYPE_LIGHT:
case Vertex::OBJECT_TYPE_SPECULAR:
    continue;
}
```

頂点間の接続

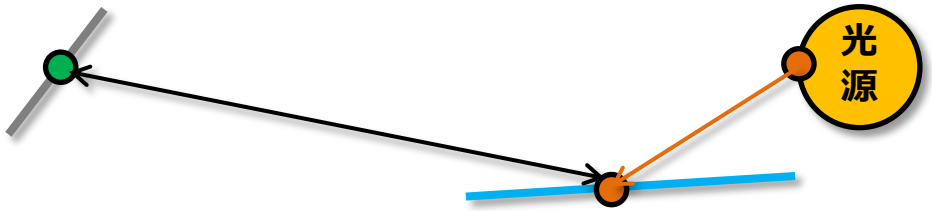
- なお、端点が完全拡散面上だった場合かつ端点間に別の物体が存在すれば、そこで処理を打ち切れる。




```
// eyeサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
// eyeサブパスの端点の材質の種類によって変わる。
switch (eye_end.type) {
case Vertex::OBJECT_TYPE_DIFFUSE:
    connection_throughput = multiply(connection_throughput, spheres[eye_end.objectID].color / kPI);
    // 端点同士が別の物体で遮蔽されるかどうかを判定する。遮蔽されていたら処理終わり。
    if ((intersection.hitpoint.position - eye_end.position).length() >= kEPS) {
        continue;
    }
    break;
case Vertex::OBJECT_TYPE_LENS:
    // ここに入るのはnum_eye_vertex == 1のときだけ
    {
        // lightサブパスを直接レンズにつなげるパターン
        Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
        const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
        if (kEPS < lens_t && lens_t < intersection.hitpoint.distance) {
            // レイがレンズにヒット&イメージセンサにヒット
            const Vec x0_xI = position_on_imagesensor - position_on_lens;
            const Vec x0_xV = position_on_objectplane - position_on_lens;
            const Vec x0_x1 = test_ray.org - position_on_lens;

            target_x = (int)uv_on_imagesensor.x;
            target_y = (int)uv_on_imagesensor.y;
            clamp(target_x, 0, camera.image_width_px);
            clamp(target_y, 0, camera.image_height_px);

            connection_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * connection_throughput;
        } else {
            // lightサブパスを直接レンズにつなげようとしたが、遮蔽されたりイメージセンサにヒットしなかった場合、終わり。
            continue;
        }
    }
    break;
    // eyeサブパスの端点がスペキュラや光源（反射率0）だった場合は重みがゼロになりパス全体の寄与もゼロになるので、処理終わり。
case Vertex::OBJECT_TYPE_LIGHT:
case Vertex::OBJECT_TYPE_SPECULAR:
    continue;
}
```

頂点間の接続

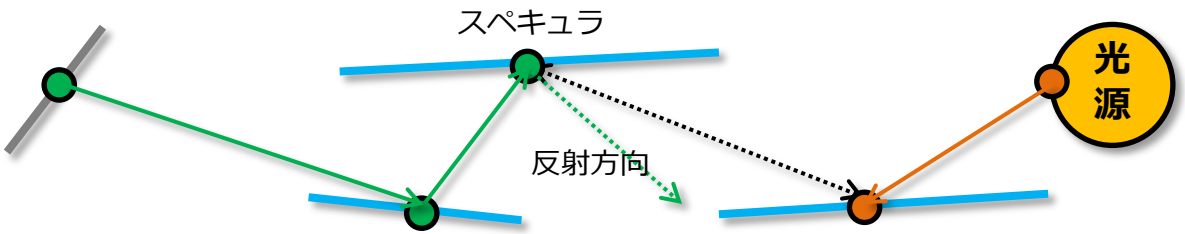
- レンズ上だった場合、レンズへの入射方向によって最終的なイメージセンサ上の位置が決まるので、その計算を行う。また、 connection_throughput にw'を乗算する。




```
// eyeサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
// eyeサブパスの端点の材質の種類によって変わる。
switch (eye_end.type) {
case Vertex::OBJECT_TYPE_DIFFUSE:
    connection_throughput = multiply(connection_throughput, spheres[eye_end.objectID].color / kPI);
    // 端点同士が別の物体で遮蔽されるかどうかを判定する。遮蔽されていたら処理終わり。
    if ((intersection.hitpoint.position - eye_end.position).length() >= kEPS) {
        continue;
    }
    break;
case Vertex::OBJECT_TYPE_LENS:
    // ここに入るのはnum_eye_vertex == 1のときだけ
    {
        // lightサブパスを直接レンズにつなげるパターン
        Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
        const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
        if (kEPS < lens_t && lens_t < intersection.hitpoint.distance) {
            // レイがレンズにヒット&イメージセンサにヒット
            const Vec x0_xI = position_on_imagesensor - position_on_lens;
            const Vec x0_xV = position_on_objectplane - position_on_lens;
            const Vec x0_x1 = test_ray.org - position_on_lens;

            target_x = (int)uv_on_imagesensor.x;
            target_y = (int)uv_on_imagesensor.y;
            clamp(target_x, 0, camera.image_width_px);
            clamp(target_y, 0, camera.image_height_px);

            connection_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * connection_throughput;
        } else {
            // lightサブパスを直接レンズにつなげようとしたが、遮蔽されたりイメージセンサにヒットしなかった場合、終わり。
            continue;
        }
    }
    break;
    // eyeサブパスの端点がスペキュラや光源（反射率0）だった場合は重みがゼロになりパス全体の寄与もゼロになるので、処理終わり。
case Vertex::OBJECT_TYPE_LIGHT:
case Vertex::OBJECT_TYPE_SPECULAR:
    continue;
}
```

頂点間の接続

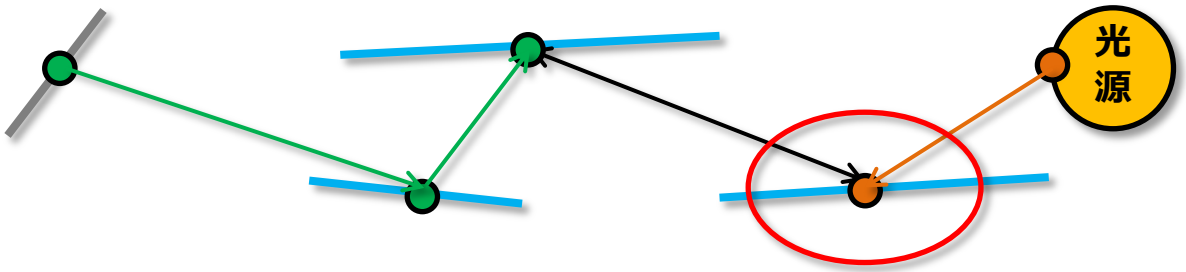
- スペキュラ上において反射方向や屈折方向は決定的に決まる。この方向と、lightサブパスの端点への方向が一致する確率は0なので、eyeサブパスの端点がスペキュラ上に存在した場合、スループットは必ず0になる。（BRDFがδであり、0になるため）よって、そこで処理を打ち切る。
- また、端点が光源上にいても打ち切る。（反射率0を仮定しているため）




```
// eyeサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
// eyeサブパスの端点の材質の種類によって変わる。
switch (eye_end.type) {
case Vertex::OBJECT_TYPE_DIFFUSE:
    connection_throughput = multiply(connection_throughput, spheres[eye_end.objectID].color / kPI);
    // 端点同士が別の物体で遮蔽されるかどうかを判定する。遮蔽されていたら処理終わり。
    if ((intersection.hitpoint.position - eye_end.position).length() >= kEPS) {
        continue;
    }
    break;
case Vertex::OBJECT_TYPE_LENS:
    // ここに入るのはnum_eye_vertex == 1のときだけ
    {
        // lightサブパスを直接レンズにつなげるパターン
        Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
        const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
        if (kEPS < lens_t && lens_t < intersection.hitpoint.distance) {
            // レイがレンズにヒット&イメージセンサにヒット
            const Vec x0_xI = position_on_imagesensor - position_on_lens;
            const Vec x0_xV = position_on_objectplane - position_on_lens;
            const Vec x0_x1 = test_ray.org - position_on_lens;

            target_x = (int)uv_on_imagesensor.x;
            target_y = (int)uv_on_imagesensor.y;
            clamp(target_x, 0, camera.image_width_px);
            clamp(target_y, 0, camera.image_height_px);

            connection_throughput = camera.W_dash(x0_xV, x0_xI, x0_x1) * connection_throughput;
        } else {
            // lightサブパスを直接レンズにつなげようとしたが、遮蔽されたりイメージセンサにヒットしなかった場合、終わり。
            continue;
        }
    }
    break;
    // eyeサブパスの端点がスペキュラや光源（反射率0）だった場合は重みがゼロになりパス全体の寄与もゼロになるので、処理終わり。
case Vertex::OBJECT_TYPE_LIGHT:
case Vertex::OBJECT_TYPE_SPECULAR:
    continue;
```

頂点間の接続

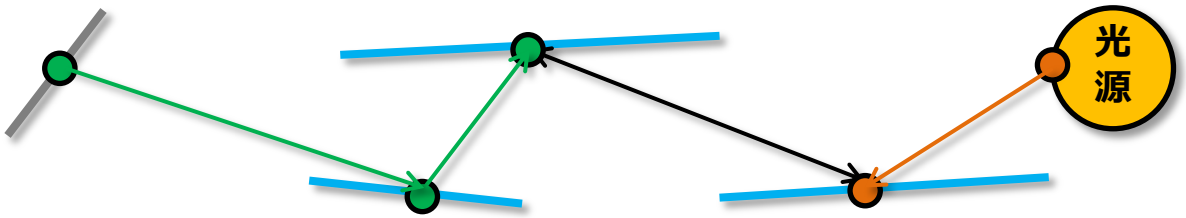
- 次に、lightサブパス側について計算を行う。
- eyeサブパス同様、lightサブパスの端点の材質によって処理が変わる。
- まず、完全拡散面だった場合、 connection_throughputにBRDFを乗算する。




```
// lightサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
// lightサブパスの端点の材質の種類によって変わる。
switch (light_end.type) {
case Vertex::OBJECT_TYPE_DIFFUSE:
    connection_throughput = multiply(connection_throughput, spheres[light_end.objectID].color / kPI);
    break;
case Vertex::OBJECT_TYPE_LIGHT:
    // 光源の反射率0を仮定しているため、ライトトレーシングの時、最初の頂点以外は光源上に頂点生成されず、
    // num_light_vertex == 1以外でここに入ってくることは無い。
    break;
    // lightサブパスの端点がスペキュラやレンズだった場合は重みがゼロになりパス全体の寄与もゼロになるので、処理終わり。
case Vertex::OBJECT_TYPE_LENS:
case Vertex::OBJECT_TYPE_SPECULAR:
    continue;
}
```

頂点間の接続

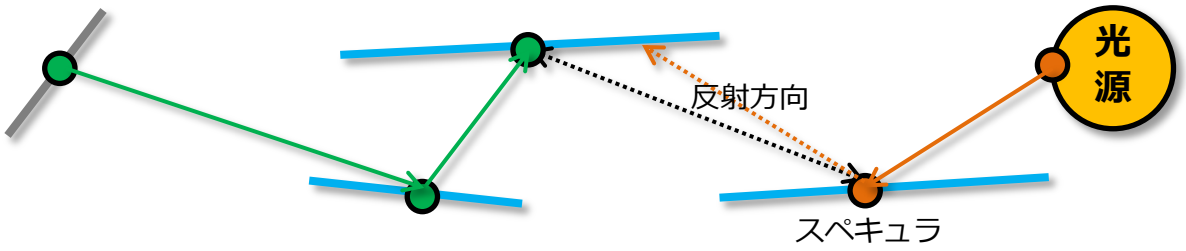
- 光源の反射率0を仮定しているため、ライトトレーシングの時、最初の頂点以外は光源上に頂点生成されず、 lightサブパスの頂点数が1になる時以外で、以下の赤枠部分に入ることは無い。




```
// lightサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
// lightサブパスの端点の材質の種類によって変わる。
switch (light_end.type) {
case Vertex::OBJECT_TYPE_DIFFUSE:
    connection_throughput = multiply(connection_throughput, spheres[light_end.objectID].color / kPI);
    break;
case Vertex::OBJECT_TYPE_LIGHT:
    // 光源の反射率0を仮定しているため、ライトトレーシングの時、最初の頂点以外は光源上に頂点生成されず、
    // num_light_vertex == 1以外でここに入ってくることは無い。
    break;
    // lightサブパスの端点がスペキュラやレンズだった場合は重みがゼロになりパス全体の寄与もゼロになるので、処理終わり。
case Vertex::OBJECT_TYPE_LENS:
case Vertex::OBJECT_TYPE_SPECULAR:
    continue;
}
```

頂点間の接続

- eyeサブパスの場合と同様に、サブパスの端点がスペキュラ上に存在した場合、そこで処理を打ち切る。またレンズ上に存在した場合もスペキュラとみなすため、打ち切る。




```
// lightサブパスのスループット調整のための重み計算（主にBRDFの考慮のため）
// lightサブパスの端点の材質の種類によって変わる。
switch (light_end.type) {
case Vertex::OBJECT_TYPE_DIFFUSE:
    connection_throughput = multiply(connection_throughput, spheres[light_end.objectID].color / kPI);
    break;
case Vertex::OBJECT_TYPE_LIGHT:
    // 光源の反射率0を仮定しているため、ライトトレーシングの時、最初の頂点以外は光源上に頂点生成されず、
    // num_light_vertex == 1以外でここに入ってくることは無い。
    break;
    // lightサブパスの端点がスペキュラやレンズだった場合は重みがゼロになりパス全体の寄与もゼロになるので、処理終わり。
case Vertex::OBJECT_TYPE_LENS:
case Vertex::OBJECT_TYPE_SPECULAR:
    continue;
}
```

モンテカルロ積分

- 最後に、端点間のG項を計算し、MISの重みを計算し、モンテカルロ積分を行って終わり。
- モンテカルロ積分は、今まで計算したスループットと端点間のスループットの積をパスのサンプリング確率密度で割ることによって行う。もちろんMISの重みも乗算する。
- これにより、eyeサブパスの頂点数+lightサブパスの頂点数分の頂点数の積分に対応したモンテカルロ積分が解かれたことになる。

```
// 端点間のジオメトリターム
const double G = (
    std::max(dot(normalize(-1.0*light_end_to_eye_end), eye_end.orienting_normal), 0.0) *
    std::max(dot(normalize(light_end_to_eye_end), light_end.orienting_normal), 0.0)) /
    light_end_to_eye_end.length_squared();
connection_throughput = G * connection_throughput;

// MISの重みを計算する。
const double mis_weight = calc_mis_weight(camera, total_pdf_A, eye_vs, light_vs, num_eye_vertex, num_light_vertex);

if (_isnan(mis_weight)) {
    continue;
}

// 最終的なモンテカルロコントリビューション =
// MIS重み * 頂点を接続することで新しく導入される項 * eyeサブパススループット * lightサブパススループット / パスのサンプリング確率密度の総計
const Color result = mis_weight * multiply(multiply(connection_throughput, eye_throughput), light_throughput) / total_pdf_A;
bpt_result.samples.push_back(BidirectionalPathtracingResult::Sample(target_x, target_y, result, num_eye_vertex <= 1 ? false : true));
```

MISの重み計算

- あるeyeサブパスとlightサブパスから成るパスに対応するMISの重みを計算することについて考える。
  - lightサブパスの頂点数をs、eyeサブパスの頂点数をtとする。
  - $k = s + t - 1$とする。
  - eyeサブパスを$x_0 x_1 \dots x_{t-1}$、lightサブパスを$y_{s-1} \dots y_1 y_0$とする。
  - 接続後のパスを$\bar{x} = x_0 x_1 \dots x_{t-1}\, y_{s-1} \dots y_1 y_0$とする。
- パワーヒューリスティックを使うと、重みは以下のようになる。
  - ただし、$p_{s,t}(\bar{x})$は頂点数$s + t$のモンテカルロ積分に対するサンプリング戦略の一つで、eyeサブパスの頂点数t、lightサブパスの頂点数sの場合の$\bar{x}$のサンプリング確率密度。
  - 既に説明したとおり、eyeサブパスの頂点数t 、lightサブパスの頂点数sのモンテカルロ積分は、頂点数$s + t$の積分に対するモンテカルロ積分の一つである（サンプリング戦略の一つ）。この時、同じ頂点数$s + t$のモンテカルロ積分に対して、全部で$s + t + 1$種類のサンプリング戦略が考えられるので、以下のような式になる。
  - $s + t + 1$種類のサンプリング戦略とは、すなわちeyeサブパス、lightサブパスの頂点数の組み合わせの数である。

$$w_{s,t}(\bar{x}) = \frac{p_{s,t}(\bar{x})^{\beta}}{\sum_{j=0}^{s+t} p_{j,(s+t)-j}(\bar{x})^{\beta}}$$

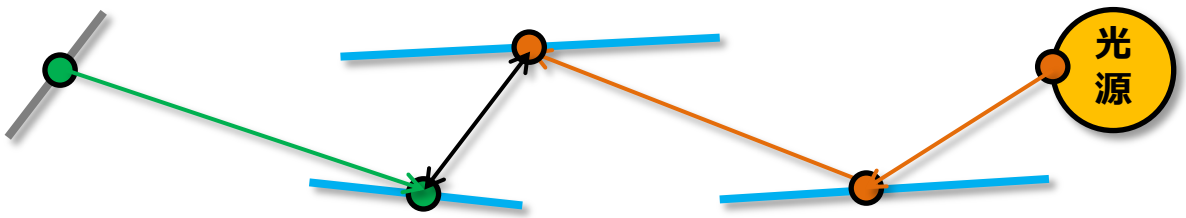
MISの重み計算の例

- あるeyeサブパスとlightサブパスから成るパスに対応するMISの重みを計算することについて考える。
  - eyeサブパスの**頂点数を2**、lightサブパスの**頂点数を3**とする。
- パワーヒューリスティックを使うと、重みは以下のようになる。
  - $\bar{x}$を「仮に」eyeサブパスの**頂点数5**、lightサブパスの**頂点数0**の様なサンプリングによって得られたとしたときの、確率密度が$p_{0,5}(\bar{x})$。$p_{1,4}(\bar{x})$や$p_{3,2}(\bar{x})$等も同様。

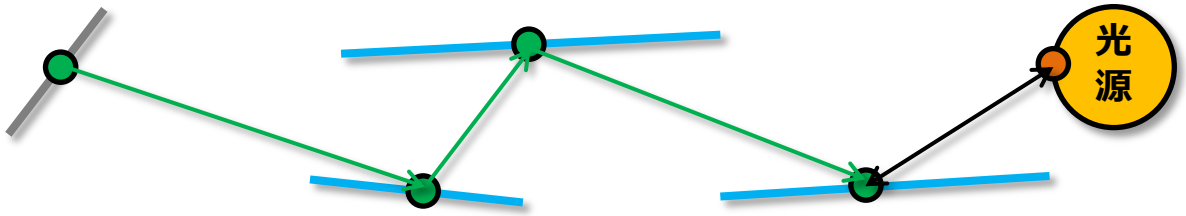
$$w_{3,2}(\bar{x}) = \frac{p_{3,2}(\bar{x})^{\beta}}{p_{0,5}(\bar{x})^{\beta} + p_{1,4}(\bar{x})^{\beta} + p_{2,3}(\bar{x})^{\beta} + p_{3,2}(\bar{x})^{\beta} + p_{4,1}(\bar{x})^{\beta} + p_{5,0}(\bar{x})^{\beta}}$$

MISの重み計算の例

- 実際に得られたパスを、もし別のサンプリング戦略（頂点数の組み合わせ）でサンプリングされたとしたら、どのような確率密度になるのかを計算する必要があるということ。



実際のパス（eyeサブパス頂点数2、lightサブパス頂点数3）



同じパスを別の戦略（eyeサブパス頂点数4、lightサブパス頂点数1）でサンプリングした場合

calc_mis_weight()

- 以上の話に基づき、与えられたeyeサブパス、lightサブパスのMIS重みを計算する。

```
double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, ... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
}
```

calc_mis_weight()

- まず、eyeサブパスとlightサブパスの始点である、光源上のサンプリング確率密度とレンズ上のサンプリング確率密度を計算しておく。
- さらに、後続の処理をやりやすくするためにeyeサブパスとlightサブパスを一列に並べて置く。
  - 先頭から順番に、lightサブパスの頂点を並べ、並べ終わったらeyeサブパスの頂点を逆順で並べることで全体のパスとする。
  - $\bar{x} = x_0 x_1 \dots x_{t-1}\, y_{s-1} \dots y_1 y_0 = \boldsymbol{z_{s+t-1}} \dots \boldsymbol{z_1 z_0}$
  - z_iとvs[i]が対応している。

```
double calc_mis_weight(const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, ... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
}
```

MISの重みの効率的計算

- 頂点数の合計が一定であることから、$p_{s,t}(\bar{x})$を以下のように書き直す。
  - iはあるサンプリング戦略における、lightサブパスの頂点数を示す。

$$p_i(\bar{x}) = p_{i,(s+t)-i}(\bar{x})$$

- $p_i(\bar{x})$を$i = 0$から$s + t$の範囲で計算すれば重みは計算できるが、単純にこれを行うと効率が悪い。そこで、まず$p_i(\bar{x})$と$p_{i+1}(\bar{x})$の比、$\frac{p_{i+1}(\bar{x})}{p_i(\bar{x})}$を計算して、その結果に基づいて$p_i(\bar{x})$を計算することにする。
- ソースコード中ではこの比はpi1_piという配列が対応する。

MISの重みの効率的計算

- 比の計算は以下のようにして行う。
- $i = 0$の時、

$$\frac{p_1(\bar{x})}{p_0(\bar{x})} = \frac{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-1})P_A^L(y_0)}{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-1})P_A^E(x_k)} = \frac{P_A^L(y_0)}{P_A^E(x_k)} = \frac{P_A^L(z_0)}{P_A^E(z_0)}$$

- ただし、$P_A^E(x)$は、パストレーシングによって頂点xがサンプリングされたときの、面積測度に関する確率密度で、$P_A^L(x)$は、ライトトレーシングによって頂点xがサンプリングされたときの、面積測度に関する確率密度。
- $P_A^L(y_0) = P_A^L(z_0)$は頂点が光源上からサンプリングされる確率密度で、これは計算済み。
- $P_A^E(x_k) = P_A^E(z_0)$を計算するわけだが、実際の分布は$P_\sigma^E(z_1 \to z_0)$に基づいているため、ひとつ前の頂点の情報も必要になる。さらに、z_1がスペキュラ上に存在した場合、さらにその一つ前の頂点z_2の情報に必要になるので、都合三つ分の頂点を使って$P_A^E(z_0)$を計算することになる。
  - この計算はcalc_pdf_A()が行う。
  - なお、頂点をlightサブパスの始点が最初に来るように一列に並べていたので、x_kは頂点配列上ではz_0に対応し、y_0もz_0に対応している。

MISの重みの効率的計算

- なお、各頂点のサンプリング確率密度の計算に際してはロシアンルーレットによる生成判定の確率も考慮する。
  - ここで以下のように分母分子を打ち消しあうことが容易になるようにするために、ロシアンルーレットの確率を頂点ごとに独立に計算していた。

$$\frac{p_1(\bar{x})}{p_0(\bar{x})} = \frac{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-1})P_A^L(y_0)}{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-1})P_A^E(x_k)} = \frac{P_A^L(y_0)}{P_A^E(x_k)} = \frac{P_A^L(z_0)}{P_A^E(z_0)}$$

```
double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, .... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));

        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
}
```

MISの重みの効率的計算

- $0 < i < k$の時、

$$\frac{p_{i+1}(\bar{x})}{p_i(\bar{x})} = \frac{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-(i+1)})P_A^L(y_i)P_A^L(y_{i-1})\ldots P_A^L(y_1)P_A^L(y_0)}{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-(i+1)})P_A^E(x_{k-i})P_A^L(y_{i-1})\ldots P_A^L(y_1)P_A^L(y_0)} = \frac{P_A^L(y_i)}{P_A^E(x_{k-i})} = \frac{P_A^L(z_i)}{P_A^E(z_i)}$$

- $p_i(\bar{x})$はlightサブパス頂点数iのサンプリング戦略なので、同じパスについて、y_0からy_{i-1}はP_A^Lでサンプリングし、 x_{k-i}からx_0はP_A^Eでサンプリングすることになる。
- 先ほどと同様に、対応する頂点のサンプリング確率密度を、その前の頂点の情報を使って計算していく。
- 一列に並べた頂点配列上で、y_iに対応するインデックスはiで、その前の頂点は（ライトトレーシングなので）$i-1, i-2$になる。つまりz_{i-2}, z_{i-1}, z_i。
- x_{k-i}に対応するインデックスもiで、その前の頂点は（パストレーシングなので）$i+1, i+2$になる。つまりz_{i+2}, z_{i+1}, z_i。

```
double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, .... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
}
```

MISの重みの効率的計算

- $i = k$の時、

$$\frac{p_{k+1}(\bar{x})}{p_k(\bar{x})} = \frac{P_A^L(y_k)P_A^L(y_{k-1})\ldots P_A^L(y_1)P_A^L(y_0)}{P_A^E(x_0)P_A^L(y_{k-1})\ldots P_A^L(y_1)P_A^L(y_0)} = \frac{P_A^L(y_k)}{P_A^E(x_0)} = \frac{P_A^L(z_k)}{P_A^E(z_k)}$$

- $P_A^E(x_0)$ は頂点がレンズ上からサンプリングされる確率密度で、これは計算済み。

```
double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, ... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
}
```

MISの重みの効率的計算

- 次に、比を使って$p_i(\bar{x})$を計算する。まず、初期値として$p_s(\bar{x})$を与える。これは、「実際に」サンプリングされたパスの確率密度なので、すでに求まっている。
- あとは、$\frac{p_{i+1}(\bar{x})}{p_i(\bar{x})}$が全て分かっているため、$p_s(\bar{x})$から初めてすべての$p_i(\bar{x})$を計算することが出来る。

```
double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, ... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
}
```

calc_pdf_A()

- $P_A^E(x)$と$P_A^L(y)$を計算する関数。今回はパストレーシングもライトトレーシングも頂点のサンプリング自体はほとんど同じなので、共通の関数でよい。
- 基本的に、始点と次の頂点を与えられると次の頂点の面積測度に関するサンプリング確率密度を返す。スペキュラの考慮のため、始点のひとつ前の頂点も与える。

```
// 頂点fromから頂点nextをサンプリングしたとするとき、面積測度に関するサンプリング確率密度を計算する。
// 今回はlighttracingもPathtracingも同じ戦略でサンプリングを行うため一つの関数で良い。
// サンプリング戦略が異なるなら、それぞれのための関数を用意する必要がある。
double calc_pdf_A(const Camera &camera, const std::vector<const Vertex*> &vs, const int prev_from_idx, const int from_idx, const int next_idx) {
    const Vertex &from_vertex = *vs[from_idx];
    const Vertex &next_vertex = *vs[next_idx];
    Vertex const *prev_from_vertex = NULL;
    if (0 <= prev_from_idx && prev_from_idx < vs.size())
        prev_from_vertex = vs[prev_from_idx];

    const Vec to = next_vertex.position - from_vertex.position;
    const Vec normalized_to = normalize(to);
    double pdf = 0.0;

    // 頂点fromのオブジェクトの種類によって、その次の頂点のサンプリング確率密度の計算方法が変わる。
    switch (from_vertex.type) {
    case Vertex::OBJECT_TYPE_LIGHT:
    case Vertex::OBJECT_TYPE_DIFFUSE:
        pdf = sample_hemisphere_pdf_omega(from_vertex.orienting_normal, normalized_to);
        break;
    case Vertex::OBJECT_TYPE_LENS:
        {
            // レンズ上の点からシーン上の点をサンプリングするときの面積測度に関する確率密度を計算。
            // イメージセンサ上の点のサンプリング確率密度を元に変換する。
            const Ray test_ray(next_vertex.position, -1.0 * normalized_to);
            Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
            const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
            if (kEPS < lens_t) {
                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const Vec x0_xV = position_on_objectplane - position_on_lens;
                const Vec x0_x1 = test_ray.org - position_on_lens;

                const double P_Image = 1.0 / (camera.imagesensor_width * camera.imagesensor_height);
                const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, next_vertex.orienting_normal);
                return PA_x1;
            }
            return 0.0;
        }
        break;
    case Vertex::OBJECT_TYPE_SPECULAR:
        if (spheres[from_vertex.objectID].reflection_type == REFLECTION_TYPE_GLASS) {
            // 始点がガラス上に存在した場合、次の点が屈折によってサンプリングされたのか、反射によってサンプリングされたかによって
            // 次の頂点のサンプリング確率が変わるので、その判定を行う。
            if (prev_from_vertex != NULL) {
                // from頂点のひとつ前の頂点に基づいて、物体に入り込んでいるのか、それとも出て行くのかを判定する。
                const Vec into_from_vertex_dir = normalize(from_vertex.position - prev_from_vertex->position);
                const bool into = dot(into_from_vertex_dir, from_vertex.object_normal) < 0.0; // prev_from_vertexからfrom_vertexへのベクトルが、スペキュラ物体に入るのか、それとも出るのか
                const Vec from_new_orienting_normal = into ? from_vertex.object_normal : -1.0 * from_vertex.object_normal;

                // 全反射しているのか、していないのかを判定
                Vec reflection_dir, refraction_dir;
                double fresnel_reflectance, fresnel_transmittance;
                if (check_refraction(
                    into, from_vertex.position, into_from_vertex_dir, from_vertex.object_normal, from_new_orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                    // 反射 or 屈折
                    pdf = dot(from_new_orienting_normal, normalized_to) > 0.0 ? reflection_probability : 1.0 - reflection_probability;
                } else {
                    // 全反射
                    pdf = 1.0;
                }
            }
        } else {
            // 鏡面
            pdf = 1.0;
        }
        break;
    }

    // 次の頂点の法線を、現在の頂点からの方向ベクトルに基づいて改めて求める
    const Vec next_new_orienting_normal = dot(to, next_vertex.object_normal) < 0.0 ? next_vertex.object_normal : -1.0 * next_vertex.object_normal;
    // 立体角測度に関する確率密度を面積測度に関する確率密度に変換
    return pdf * (dot(-1.0 * normalized_to, next_new_orienting_normal) / to.length_squared());
}
```

calc_pdf_A()

- 始点の材質ごとに、次の頂点のサンプリング方法は変わる。始点が完全拡散面や光源上に存在すれば、次の頂点はコサイン項によるインポータンスサンプリングによって行われる。
- ここでの結果は、立体角測度に関するものなので、最後に面積測度に関する確率密度に変換する。

```
// 頂点fromから頂点nextをサンプリングしたとするとき、面積測度に関するサンプリング確率密度を計算する。
// 今回はLighttracingもPathtracingも同じ戦略でサンプリングを行うため一つの関数で良い。
// サンプリング戦略が異なるなら、それぞれのための関数を用意するの要がある。
double calc_pdf_A(const Camera &camera, const std::vector<const Vertex*> &vs, const int prev_from_idx, const int from_idx, const int next_idx) {
    const Vertex &from_vertex = *vs[from_idx];
    const Vertex &next_vertex = *vs[next_idx];
    Vertex const *prev_from_vertex = NULL;
    if (0 <= prev_from_idx && prev_from_idx < vs.size())
        prev_from_vertex = vs[prev_from_idx];

    const Vec to = next_vertex.position - from_vertex.position;
    const Vec normalized_to = normalize(to);
    double pdf = 0.0;

    // 頂点fromのオブジェクトの種類によって、その次の頂点のサンプリング確率密度の計算方法が変わる。
    switch (from_vertex.type) {
    case Vertex::OBJECT_TYPE_LIGHT:
    case Vertex::OBJECT_TYPE_DIFFUSE:
        pdf = sample_hemisphere_pdf_omega(from_vertex.orienting_normal, normalized_to);
        break;
    case Vertex::OBJECT_TYPE_LENS:
        {
            // レンズ上の点からシーン上の点をサンプリングするときの面積測度に関する確率密度を計算。
            // イメージセンサ上の点のサンプリング確率密度を元に変換する。
            const Ray test_ray(next_vertex.position, -1.0 * normalized_to);
            Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
            const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
            if (kEPS < lens_t) {
                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const Vec x0_xV = position_on_objectplane - position_on_lens;
                const Vec x0_x1 = test_ray.org - position_on_lens;

                const double P_Image = 1.0 / (camera.imagesensor_width * camera.imagesensor_height);
                const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, next_vertex.orienting_normal);
                return PA_x1;
            }
            return 0.0;
        }
        break;
    case Vertex::OBJECT_TYPE_SPECULAR:
        if (spheres[from_vertex.objectID].reflection_type == REFLECTION_TYPE_GLASS) {
            // 始点がガラスと仮定した場合、次の点が屈折によってサンプリングされたのか、反射によってサンプリングされたかによって
            // 次の頂点のサンプリング確率が変わるので、その判定を行う。
            if (prev_from_vertex != NULL) {
                // from頂点のひとつ前の頂点に基づいて、物体に入り込んでいるのか、それとも出て行くのかを判定する。
                const Vec into_from_vertex_dir = normalize(from_vertex.position - prev_from_vertex->position);
                const bool into = dot(into_from_vertex_dir, from_vertex.object_normal) < 0.0; // prev_from_vertexからfrom_vertexへのベクトルが、スペキュラ物体に入るのか、それとも出るのか
                const Vec from_new_orienting_normal = into ? from_vertex.object_normal : -1.0 * from_vertex.object_normal;

                // 全反射しているのか、していないのかを判定
                Vec reflection_dir, refraction_dir;
                double fresnel_reflectance, fresnel_transmittance;
                if (check_refraction(
                    into, from_vertex.position, into_from_vertex_dir, from_vertex.object_normal, from_new_orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                    // 反射 or 屈折
                    pdf = dot(from_new_orienting_normal, normalized_to) > 0.0 ? reflection_probability : 1.0 - reflection_probability;
                } else {
                    // 全反射
                    pdf = 1.0;
                }
            }
        } else {
            // 鏡面
            pdf = 1.0;
        }
        break;
    }

    // 次の頂点の法線を、現在の頂点からの方向ベクトルに基づいて改めて求める
    const Vec next_new_orienting_normal = dot(to, next_vertex.object_normal) < 0.0 ? next_vertex.object_normal : -1.0 * next_vertex.object_normal;
    // 立体角測度に関する確率密度を面積測度に関する確率密度に変換
    return pdf * (dot(-1.0 * normalized_to, next_new_orienting_normal) / to.length_squared());
}
```

calc_pdf_A()

- 始点がレンズ上に存在した場合、すでに見てきたようなやり方で次の頂点のサンプリング確率密度を計算する。

```
// 頂点fromから頂点nextをサンプリングしたとすると、面積測度に関するサンプリング確率密度を計算する。
// 今回はLighttracingもPathtracingも同じ戦略でサンプリングを行うため一つの関数で良い。
// サンプリング戦略が異なるなら、それぞれのための関数を用意する必要がある。
double calc_pdf_A(const Camera &camera, const std::vector<const Vertex*> &vs, const int prev_from_idx, const int from_idx, const int next_idx) {
    const Vertex &from_vertex = *vs[from_idx];
    const Vertex &next_vertex = *vs[next_idx];
    Vertex const *prev_from_vertex = NULL;
    if (0 <= prev_from_idx && prev_from_idx < vs.size())
        prev_from_vertex = vs[prev_from_idx];

    const Vec to = next_vertex.position - from_vertex.position;
    const Vec normalized_to = normalize(to);
    double pdf = 0.0;

    // 頂点fromのオブジェクトの種類によって、その次の頂点のサンプリング確率密度の計算方法が変わる。
    switch (from_vertex.type) {
    case Vertex::OBJECT_TYPE_LIGHT:
    case Vertex::OBJECT_TYPE_DIFFUSE:
        pdf = sample_hemisphere_pdf_omega(from_vertex.orienting_normal, normalized_to);

    case Vertex::OBJECT_TYPE_LENS:
        {
            // レンズ上の点からシーン上の点をサンプリングするときの面積測度に関する確率密度を計算。
            // イメージセンサ上の点のサンプリング確率密度を元に変換する。
            const Ray test_ray(next_vertex.position, -1.0 * normalized_to);
            Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
            const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
            if (kEPS < lens_t) {
                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const Vec x0_xV = position_on_objectplane - position_on_lens;
                const Vec x0_x1 = test_ray.org - position_on_lens;

                const double P_Image = 1.0 / (camera.imagesensor_width * camera.imagesensor_height);
                const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, next_vertex.orienting_normal);
                return PA_x1;
            }
            return 0.0;
        }
        break;
    case Vertex::OBJECT_TYPE_ACCUCAM:
        if (spheres[from_vertex.objectID].reflection_type == REFLECTION_TYPE_GLASS) {
            // 始点がガラス上に存在した場合、次の点が屈折によってサンプリングされたのか、反射によってサンプリングされたかによって
            // 次の頂点のサンプリング確率が変わるので、その判定を行う。
            if (prev_from_vertex != NULL) {
                // from頂点のひとつ前の頂点に基づいて、物体に入り込んでいるのか、それとも出て行くのかを判定する。
                const Vec into_from_vertex_dir = normalize(from_vertex.position - prev_from_vertex->position);
                const bool into = dot(into_from_vertex_dir, from_vertex.object_normal) < 0.0; // prev_from_vertexからfrom_vertexへのベクトルが、スペキュラ物体に入るのか、それとも出るのか
                const Vec from_new_orienting_normal = into ? from_vertex.object_normal : -1.0 * from_vertex.object_normal;

                // 全反射しているのか、していないのかを判定
                Vec reflection_dir, refraction_dir;
                double fresnel_reflectance, fresnel_transmittance;
                if (check_refraction(
                    into, from_vertex.position, into_from_vertex_dir, from_vertex.object_normal, from_new_orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                    // 反射 or 屈折
                    pdf = dot(from_new_orienting_normal, normalized_to) > 0.0 ? reflection_probability : 1.0 - reflection_probability;
                } else {
                    // 全反射
                    pdf = 1.0;
                }
            }
        } else {
            // 拡散
            pdf = 1.0;
        }
        break;
    }

    // 次の頂点の法線を、現在の頂点からの方向ベクトルに基づいて改めて求める
    const Vec next_new_orienting_normal = dot(to, next_vertex.object_normal) < 0.0 ? next_vertex.object_normal : -1.0 * next_vertex.object_normal;
    // 立体角測度に関する確率密度を面積測度に関する確率密度に変換
    return pdf * (dot(-1.0 * normalized_to, next_new_orienting_normal) / to.length_squared());
}
```

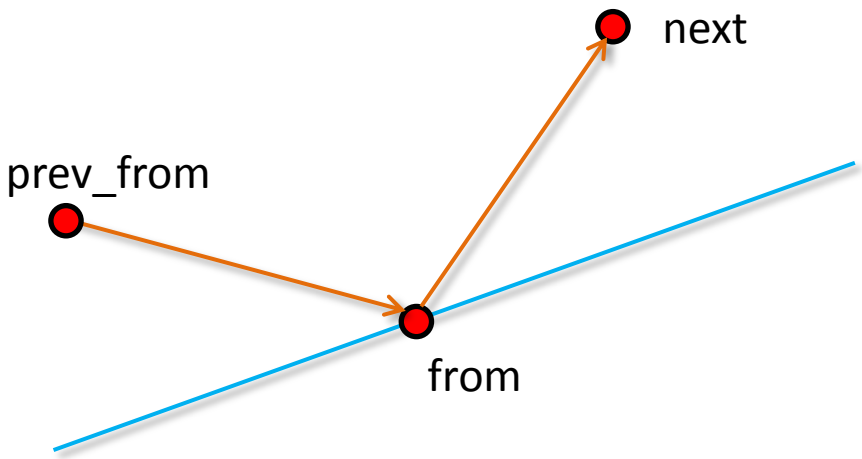
calc_pdf_A()

- 始点がスペキュラ上に存在した場合、処理が分岐する。
- まず、始点が完全鏡面上にいた場合は、次の頂点のサンプリング確率は$\delta_\sigma(\vec{\omega}_i - \vec{\omega}_r)$になる。始点と次の頂点の組み合わせは、常に「実際に」サンプリングされたパスの一部であるため、関数の値が0になることはない。
  - 唯一、始点としてスペキュラ上のサブパスの端点が選ばれた場合のみ、次の頂点のサンプリング確率密度は0になるはずだが、この場合はそもそもとっくに処理が打ち切られているはずなので考えなくて良い。
- しかし、例によって数値的に処理しにくいので、δ_σはあとでまとめて処理することにして、ひとまず係数だけ考慮する。完全鏡面なら1になる。

```
// 頂点fromから頂点nextをサンプリングしたとするとき、面積測度に関するサンプリング確率密度を計算する。
// 今回はLighttracingもPathtracingも同じ戦略でサンプリングを行うため一つの関数で良い。
// サンプリング戦略が異なるなら、それぞれのための関数を用意するる要がある。
double calc_pdf_A(const Camera &camera, const std::vector<const Vertex*> &vs, const int prev_from_idx, const int from_idx, const int next_idx) {
    const Vertex &from_vertex = *vs[from_idx];
    const Vertex &next_vertex = *vs[next_idx];
    Vertex const *prev_from_vertex = NULL;
    if (0 <= prev_from_idx && prev_from_idx < vs.size())
        prev_from_vertex = vs[prev_from_idx];

    const Vec to = next_vertex.position - from_vertex.position;
    const Vec normalized_to = normalize(to);
    double pdf = 0.0;

    // 頂点fromのオブジェクトの種類によって、その次の頂点のサンプリング確率密度の計算方法が変わる。
    switch (from_vertex.type) {
    case Vertex::OBJECT_TYPE_LIGHT:
    case Vertex::OBJECT_TYPE_DIFFUSE:
        pdf = sample_hemisphere_pdf_omega(from_vertex.orienting_normal, normalized_to);
        break;
    case Vertex::OBJECT_TYPE_LENS:
        {
            // レンズ上の点からシーン上の点をサンプリングするときの面積測度に関する確率密度を計算。
            // イメージセンサ上の点のサンプリング確率密度を元に変換する。
            const Ray test_ray(next_vertex.position, -1.0 * normalized_to);
            Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
            const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
            if (kEPS < lens_t) {
                const Vec x0_xI = position_on_imagesensor - position_on_lens;
                const Vec x0_xV = position_on_objectplane - position_on_lens;
                const Vec x0_x1 = test_ray.org - position_on_lens;

                const double P_Image = 1.0 / (camera.imagesensor_width * camera.imagesensor_height);
                const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, next_vertex.orienting_normal);
                return PA_x1;
            }
            return 0.0;
        }
        break;
    case Vertex::OBJECT_TYPE_SPECULAR:
        if (spheres[from_vertex.objectID].reflection_type == REFLECTION_TYPE_GLASS) {
            // 始点がガラス上に存在した場合、次の点が屈折によってサンプリングされたのか、反射によってサンプリングされたかによって
            // 次の頂点のサンプリング確率が変わるので、その判定を行う。
            if (prev_from_vertex != NULL) {
                // from頂点のひとつ前の頂点に基づいて、物体に入り込んでいるのか、それとも出て行くのかを判定する。
                const Vec into_from_vertex_dir = normalize(from_vertex.position - prev_from_vertex->position);
                const bool into = dot(into_from_vertex_dir, from_vertex.object_normal) < 0.0; // prev_from_vertexからfrom_vertexへのベクトルが、スペキュラ物体に入るのか、それとも出るのか
                const Vec from_new_orienting_normal = into ? from_vertex.object_normal : -1.0 * from_vertex.object_normal;

                // 全反射しているのか、していないのかを判定
                Vec reflection_dir, refraction_dir;
                double fresnel_reflectance, fresnel_transmittance;
                if (check_refraction(
                    into, from_vertex.position, into_from_vertex_dir, from_vertex.object_normal, from_new_orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                    // 反射 or 屈折
                    pdf = dot(from_new_orienting_normal, normalized_to) > 0.0 ? reflection_probability : 1.0 - reflection_probability;
                } else {
                    // 全反射
                    pdf = 1.0;
                }
            }
        } else {
            // 鏡面
            pdf = 1.0;
        }
        break;
    }

    // 次の頂点の法線を、現在の頂点からの方向ベクトルに基づいて改めて求める
    const Vec next_new_orienting_normal = dot(to, next_vertex.object_normal) < 0.0 ? next_vertex.object_normal : -1.0 * next_vertex.object_normal;
    // 立体角測度に関する確率密度を面積測度に関する確率密度に変換
    return pdf * (dot(-1.0 * normalized_to, next_new_orienting_normal) / to.length_squared());
}
```

calc_pdf_A()

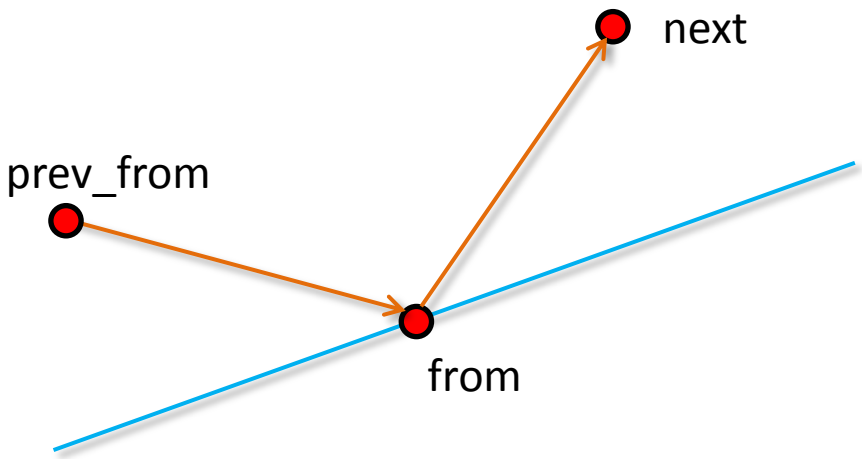
- 始点がガラス上にいた場合、始点のひとつ前の頂点から始点へ向かうベクトルを考え、そのベクトルがガラスに進入しようとしているのか、出て行こうとしているかを判定する。
- これらの結果を使うことで、始点から次の頂点へのベクトルが反射によるものなのか、屈折によるものなのかを判定することが出来る。そして、それぞれのイベントに応じてロシアンルーレットの確率を計算し、δ_σの係数とする。
- また、全反射が起こる状況もある。この場合はδ_σの係数は鏡面同様1になる。




```
// 頂点fromから頂点nextをサンプリングしたとするとき、面積測度に関するサンプリング確率密度を計算する。
// 今回はLighttracingもPathtracingも同じ戦略でサンプリングを行うため一つの関数で良い。
// サンプリング戦略が異なるなら、それぞれのための関数を用意するる要がある。
double calc_pdf_A(const Camera &camera, const std::vector<const Vertex*> &vs, const int prev_from_idx, const int from_idx, const int next_idx) {
    const Vertex &from_vertex = *vs[from_idx];
    const Vertex &next_vertex = *vs[next_idx];
    Vertex const *prev_from_vertex = NULL;
    if (0 <= prev_from_idx && prev_from_idx < vs.size())
        prev_from_vertex = vs[prev_from_idx];

    const Vec to = next_vertex.position - from_vertex.position;
    const Vec normalized_to = normalize(to);
    double pdf = 0.0;

    // 頂点fromのオブジェクトの種類によって、その次の頂点のサンプリング確率密度の計算方法が変わる。
    switch (from_vertex.type) {
    case Vertex::OBJECT_TYPE_LIGHT:
    case Vertex::OBJECT_TYPE_DIFFUSE:
        pdf = sample_hemisphere_pdf_omega(from_vertex.orienting_normal, normalized_to);
        break;
    case Vertex::OBJECT_TYPE_LENS:
        {
            // レンズ上の点からシーン上の点をサンプリングするときの面積測度に関する確率密度を計算。
            // イメージセンサ上の点のサンプリング確率密度を元に変換する。
            const Ray test_ray(next_vertex.position, -1.0 * normalized_to);
            Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
            const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
            if (kEPS < lens_t) {
                const Vec x0_x1 = position_on_imagesensor - position_on_lens;
                const Vec x0_xV = position_on_objectplane - position_on_lens;
                const Vec x0_x1 = test_ray.org - position_on_lens;

                const double P_Image = 1.0 / (camera.imagesensor_width * camera.imagesensor_height);
                const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, next_vertex.orienting_normal);
                return PA_x1;
            }
            return 0.0;
        }
        break;
    case Vertex::OBJECT_TYPE_SPECULAR:
        if (spheres[from_vertex.objectID].reflection_type == REFLECTION_TYPE_GLASS) {
            // 始点がガラス上に存在した場合、次の頂点はどちらによってサンプリングされたのか、反射によってサンプリングされたのかによって
            // 次の頂点のサンプリング確率が変わるので、その判定を行う。
            if (prev_from_vertex != NULL) {
                // from頂点のひとつ前の頂点に基づいて、物体に入り込んでいるのか、それとも出て行くのかを判定する。
                const Vec into_from_vertex_dir = normalize(from_vertex.position - prev_from_vertex->position);
                const bool into = dot(into_from_vertex_dir, from_vertex.object_normal) < 0.0; // prev_from_vertexからfrom_vertexへのベクトルが、スペキュラ物体に入るのか、それとも出るのか
                const Vec from_new_orienting_normal = into ? from_vertex.object_normal : -1.0 * from_vertex.object_normal;

                // 全反射しているのか、していないのかを判定
                Vec reflection_dir, refraction_dir;
                double fresnel_reflectance, fresnel_transmittance;
                if (check_refraction(
                    into, from_vertex.position, into_from_vertex_dir, from_vertex.object_normal, from_new_orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                    // 反射 or 屈折
                    pdf = dot(from_new_orienting_normal, normalized_to) > 0.0 ? reflection_probability : 1.0 - reflection_probability;
                } else {
                    // 全反射
                    pdf = 1.0;
                }
            }
            else {
                // 鏡面
                pdf = 1.0;
            }
        }
        break;
    }

    // 次の頂点の法線を、現在の頂点からの方向ベクトルに基づいて改めて求める
    const Vec next_new_orienting_normal = dot(to, next_vertex.object_normal) < 0.0 ? next_vertex.object_normal : -1.0 * next_vertex.object_normal;
    // 立体角測度に関する確率密度を面積測度に関する確率密度に変換
    return pdf * (dot(-1.0 * normalized_to, next_new_orienting_normal) / to.length_squared());
}
```

calc_pdf_A()

- なお、ここの判定は、三頂点の位置関係から幾何的に計算しているが、パストレーシング・ライトトレーシングの際に各頂点に屈折・反射・全反射といった反射の種類に関するメタデータを付加しておけば、ここで改めて幾何的計算を行う必要はない。
- 今回は、頂点に付加するメタデータが増えるのを嫌ってこのような実装にしたが、これ以外の実装も当然考えられることに注意。




```
// 頂点fromから頂点nextをサンプリングしたとするとき、面積測度に関するサンプリング確率密度を計算する。
// 今回はLighttracingもPathtracingも同じ戦略でサンプリングを行うため一つの関数で良い。
// サンプリング戦略が異なるなら、それぞれのための関数を用意するる要がある。
double calc_pdf_A(const Camera &camera, const std::vector<const Vertex*> &vs, const int prev_from_idx, const int from_idx, const int next_idx) {
    const Vertex &from_vertex = *vs[from_idx];
    const Vertex &next_vertex = *vs[next_idx];
    Vertex const *prev_from_vertex = NULL;
    if (0 <= prev_from_idx && prev_from_idx < vs.size())
        prev_from_vertex = vs[prev_from_idx];

    const Vec to = next_vertex.position - from_vertex.position;
    const Vec normalized_to = normalize(to);
    double pdf = 0.0;

    // 頂点fromのオブジェクトの種類によって、その次の頂点のサンプリング確率密度の計算方法が変わる。
    switch (from_vertex.type) {
    case Vertex::OBJECT_TYPE_LIGHT:
    case Vertex::OBJECT_TYPE_DIFFUSE:
        pdf = sample_hemisphere_pdf_omega(from_vertex.orienting_normal, normalized_to);
        break;
    case Vertex::OBJECT_TYPE_LENS:
        {
            // レンズ上の点からシーン上の点をサンプリングするときの面積測度に関する確率密度を計算。
            // イメージセンサ上の点のサンプリング確率密度を元に変換する。
            const Ray test_ray(next_vertex.position, -1.0 * normalized_to);
            Vec position_on_lens, position_on_objectplane, position_on_imagesensor, uv_on_imagesensor;
            const double lens_t = camera.intersect_lens(test_ray, &position_on_lens, &position_on_objectplane, &position_on_imagesensor, &uv_on_imagesensor);
            if (kEPS < lens_t) {
                const Vec x0_x1 = position_on_imagesensor - position_on_lens;
                const Vec x0_xV = position_on_objectplane - position_on_lens;
                const Vec x0_x1 = test_ray.org - position_on_lens;

                const double P_Image = 1.0 / (camera.imagesensor_width * camera.imagesensor_height);
                const double PA_x1 = camera.P_Image_to_PA_x1(P_Image, x0_xV, x0_x1, next_vertex.orienting_normal);
                return PA_x1;
            }
            return 0.0;
        }
        break;
    case Vertex::OBJECT_TYPE_SPECULAR:
        if (spheres[from_vertex.objectID].reflection_type == REFLECTION_TYPE_GLASS) {
            // 現在のガラス上に存在した場合、次の点の出方によってサンプリングされたのか、反射によってサンプリングされたかによって
            // 次の頂点のサンプリング確率が変わるので、その判定を行う。
            if (prev_from_vertex != NULL) {
                // from頂点のひとつ前の頂点に基づいて、物体に入り込んでいるのか、それとも出て行くのかを判定する。
                const Vec into_from_vertex_dir = normalize(from_vertex.position - prev_from_vertex->position);
                const bool into = dot(into_from_vertex_dir, from_vertex.object_normal) < 0.0; // prev_from_vertexからfrom_vertexへのベクトルが、スペキュラ物体に入るのか、それとも出るのか
                const Vec from_new_orienting_normal = into ? from_vertex.object_normal : -1.0 * from_vertex.object_normal;

                // 全反射しているのか、していないのかを判定
                Vec reflection_dir, refraction_dir;
                double fresnel_reflectance, fresnel_transmittance;
                if (check_refraction(
                    into, from_vertex.position, into_from_vertex_dir, from_vertex.object_normal, from_new_orienting_normal,
                    &reflection_dir, &refraction_dir, &fresnel_reflectance, &fresnel_transmittance)) {
                    // 反射 or 屈折
                    pdf = dot(from_new_orienting_normal, normalized_to) > 0.0 ? reflection_probability : 1.0 - reflection_probability;
                } else {
                    // 全反射
                    pdf = 1.0;
                }
            }
        } else {
            // 鏡面
            pdf = 1.0;
        }
        break;
    }

    // 次の頂点の法線を、現在の頂点からの方向ベクトルに基づいて改めて求める
    const Vec next_new_orienting_normal = dot(to, next_vertex.object_normal) < 0.0 ? next_vertex.object_normal : -1.0 * next_vertex.object_normal;
    // 立体角測度に関する確率密度を面積測度に関する確率密度に変換
    return pdf * (dot(-1.0 * normalized_to, next_new_orienting_normal) / to.length_squared());
}
```

スペキュラ処理

- 最後にスペキュラ処理をする。calc_pdf_A()の結果は、 δ_σに関してはあくまで係数を保持しただけのものであった。ここで、一般に以下の式が成り立っていた。

$$\frac{p_{i+1}(\bar{x})}{p_i(\bar{x})} = \frac{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-(i+1)})P_A^L(y_i)P_A^L(y_{i-1})\ldots P_A^L(y_1)P_A^L(y_0)}{P_A^E(x_0)P_A^E(x_1)\ldots P_A^E(x_{k-(i+1)})P_A^E(x_{k-i})P_A^L(y_{i-1})\ldots P_A^L(y_1)P_A^L(y_0)} = \frac{P_A^L(y_i)}{P_A^E(x_{k-i})} = \frac{P_A^L(z_i)}{P_A^E(z_i)}$$

```
double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, .... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
}
```

スペキュラ処理

- 今、lightサブパスの端点z_{s-1}とeyeサブパスの端点z_sはスペキュラ上に存在しない。このとき、各p_iについて以下のような式が成り立つ。
  - このとき、$p_s(\bar{x})$は実際にサンプリングされたパスの確率密度なのですでに求まっている。
  - $C(z_{s-1} \to z_s)$は立体角測度を面積測度に変換するための例の係数（$\frac{\cos\theta}{r^2}$）

$$p_{s+1}(\bar{x}) = \frac{P_A^L(z_s)}{P_A^E(z_s)} p_s(\bar{x}) = \frac{P_\sigma^L(z_{s-1} \to z_s) C(z_{s-1} \to z_s)}{P_\sigma^E(z_{s+1} \to z_s) C(z_{s+1} \to z_s)} p_s(\bar{x})$$

$$p_{s+2}(\bar{x}) = \frac{P_\sigma^L(z_s \to z_{s+1}) C(z_s \to z_{s+1})}{P_\sigma^E(z_{s+2} \to z_{s+1}) C(z_{s+2} \to z_{s+1})} \frac{P_\sigma^L(z_{s-1} \to z_s) C(z_{s-1} \to z_s)}{P_\sigma^E(z_{s+1} \to z_s) C(z_{s+1} \to z_s)} p_s(\bar{x})$$

$$p_{s+3}(\bar{x}) = \frac{P_\sigma^L(z_{s+1} \to z_{s+2}) C(z_{s+1} \to z_{s+2})}{P_\sigma^E(z_{s+3} \to z_{s+2}) C(z_{s+3} \to z_{s+2})} \frac{P_\sigma^L(z_s \to z_{s+1}) C(z_s \to z_{s+1})}{P_\sigma^E(z_{s+2} \to z_{s+1}) C(z_{s+2} \to z_{s+1})} \frac{P_\sigma^L(z_{s-1} \to z_s) C(z_{s-1} \to z_s)}{P_\sigma^E(z_{s+1} \to z_s) C(z_{s+1} \to z_s)} p_s(\bar{x})$$

スペキュラ処理

- 仮にz_{s+1}がスペキュラ上に存在したとすると、$P_\sigma^E(z_{s+1} \to z_s)$が$\delta_\sigma$を含むことになる。分子に$\delta_\sigma$が存在しなければ、キャンセルできないので、分母に$\delta_\sigma$が残る$p_i$は0になる。
- 以下の場合、p_{s+1}とp_{s+2}が0になる。また、p_{s+3}は分母と分子にδ_σが現れるのでキャンセルされ、0とは限らない。

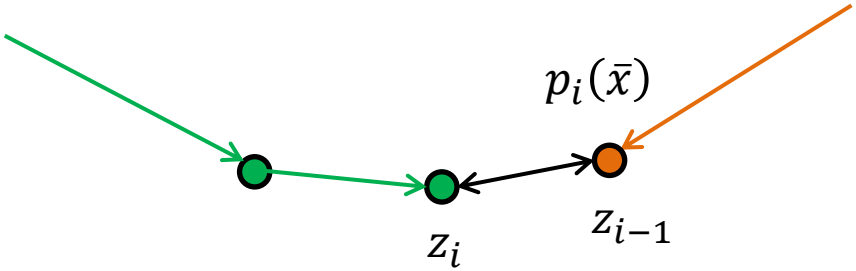
$$p_{s+1}(\bar{x}) = \frac{P_A^L(z_s)}{P_A^E(z_s)} p_s(\bar{x}) = \frac{P_\sigma^L(z_{s-1} \to z_s) C(z_{s-1} \to z_s)}{\boxed{P_\sigma^E(z_{s+1} \to z_s)} C(z_{s+1} \to z_s)} p_s(\bar{x})$$

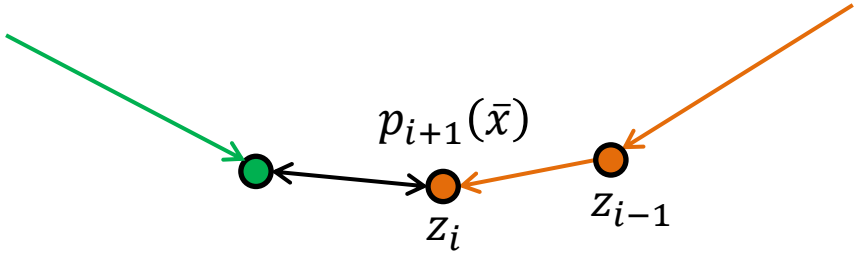
$$p_{s+2}(\bar{x}) = \frac{P_\sigma^L(z_s \to z_{s+1}) C(z_s \to z_{s+1})}{P_\sigma^E(z_{s+2} \to z_{s+1}) C(z_{s+2} \to z_{s+1})} \frac{P_\sigma^L(z_{s-1} \to z_s) C(z_{s-1} \to z_s)}{\boxed{P_\sigma^E(z_{s+1} \to z_s)} C(z_{s+1} \to z_s)} p_s(\bar{x})$$

$$p_{s+3}(\bar{x}) = \frac{\boxed{P_\sigma^L(z_{s+1} \to z_{s+2})} C(z_{s+1} \to z_{s+2})}{P_\sigma^E(z_{s+3} \to z_{s+2}) C(z_{s+3} \to z_{s+2})} \frac{P_\sigma^L(z_s \to z_{s+1}) C(z_s \to z_{s+1})}{P_\sigma^E(z_{s+2} \to z_{s+1}) C(z_{s+2} \to z_{s+1})} \frac{P_\sigma^L(z_{s-1} \to z_s) C(z_{s-1} \to z_s)}{\boxed{P_\sigma^E(z_{s+1} \to z_s)} C(z_{s+1} \to z_s)} p_s(\bar{x})$$

スペキュラ処理

- 今の話は$p_{s+1}(\bar{x}), p_{s+2}(\bar{x}), \ldots$についてだったが、$p_{s-1}(\bar{x}), p_{s-2}(\bar{x}), \ldots$についても同様。
- まとめると、 頂点z_iがスペキュラ上に存在する場合、 $p_i(\bar{x}), p_{i+1}(\bar{x})$は0になるということ。これによって、 δ_σを適切に処理したことになる。
- また、直感的には、 z_iがスペキュラ上に存在すれば、その頂点を端点とする接続によるパスのサンプリング確率は0になる、ということ。





いずれの場合も、 z_iがスペキュラ上に存在するなら、端点間を接続する確率は0になる。

パワーヒューリスティック

- 最後に、以上の結果に基づいてパワーヒューリスティックによってMIS重みを計算する。

```
double calc_mis_weight(const Camera &camera, const double total_pdf_A, const std::vector<Vertex> &eye_vs, const std::vector<Vertex> &light_vs, const int num_eye_vertex, const int num_light_vertex) {
    std::vector<const Vertex*> vs(num_light_vertex + num_eye_vertex);
    std::vector<double> pi1_pi(num_eye_vertex + num_light_vertex);
    const double PA_y0 = sample_sphere_pdf_A(spheres[LightID].radius); // 光源上のサンプリング確率
    const double PA_x0 = camera.sampling_pdf_on_lens(); // レンズ上のサンプリング確率

    // 頂点を一列に並べる。
    // vs[0] = y0, vs[1] = y1, ... vs[k-1] = x1, vs[k] = x0
    const int k = num_eye_vertex + num_light_vertex - 1;
    for (int i = 0; i < num_light_vertex; ++i)
        vs[i] = &light_vs[i];
    for (int i = num_eye_vertex - 1; i >= 0; --i)
        vs[num_light_vertex + num_eye_vertex - 1 - i] = &eye_vs[i];

    // ロシアンルーレットの確率を忘れずに入れる。
    pi1_pi[0] = PA_y0 / (calc_pdf_A(camera, vs, 2, 1, 0) * russian_roulette(spheres[vs[0]->objectID]));
    for (int i = 1; i < k; ++i) {
        const double a = calc_pdf_A(camera, vs, i - 2, i - 1, i);
        const double b = calc_pdf_A(camera, vs, i + 2, i + 1, i);
        pi1_pi[i] = a / b;
    }
    pi1_pi[k] = (calc_pdf_A(camera, vs, k - 2, k - 1, k) * russian_roulette(spheres[vs[k]->objectID])) / PA_x0;

    // pを求める
    std::vector<double> p(num_eye_vertex + num_light_vertex + 1);
    p[num_light_vertex] = total_pdf_A;
    for (int i = num_light_vertex; i <= k; ++i) {
        p[i + 1] = p[i] * pi1_pi[i];
    }
    for (int i = num_light_vertex - 1; i >= 0; --i) {
        p[i] = p[i + 1] / pi1_pi[i];
    }

    // Specular処理
    for (int i = 0; i < vs.size(); ++i) {
        if (vs[i]->type == Vertex::OBJECT_TYPE_SPECULAR) {
            p[i] = 0.0;
            p[i + 1] = 0.0;
        }
    }

    // Power-heuristic
    double mis_weight = 0.0;
    for (int i = 0; i < p.size(); ++i) {
        const double v = p[i] / p[num_light_vertex];
        mis_weight += v * v; // beta = 2
    }
    return 1.0 / mis_weight;
```

render.h

- render_by_bidirectional_pathtracing()
- 双方向パストレーシングによるモンテカルロの結果を、具体的な画像に画素値として与える。

```
void render_by_bidirectional_pathtracing(const Camera &camera, const int num_threads) {
    const int width = camera.image_width_px;
    const int height = camera.image_height_px;
    const int spp = camera.samples_per_pixel;

    std::vector<Color> image_buffer(width * height * num_threads);
#ifdef USE_OPENMP
    omp_set_num_threads(num_threads);
#endif

    const long long iteration_per_thread = spp / num_threads;
    std::cerr << "Average bidirectional sampling/pixel: " << (iteration_per_thread * num_threads) << std::endl;
    int thread_id = 0;
#pragma omp parallel private(thread_id)
    {
#ifdef USE_OPENMP
        thread_id = omp_get_thread_num();
#endif
        Random random(thread_id * 32);
        for (int iteration = 0; iteration < iteration_per_thread; ++iteration) {
            if (thread_id == 0)
                std::cerr << "Rendering (" << ((double)(num_threads * (iteration + 1))) << " bidirectional sampling/pixel) "
                << (100.0 * (double)((iteration + 1)) / iteration_per_thread) << "%" << std::endl;

            for (int y = 0; y < height; ++y) {
                for (int x = 0; x < width; ++x) {
                    BidirectionalPathtracingResult bpt_result = bidirectional_pathtracing(camera, x, y, &random);

                    for (int i = 0; i < bpt_result.samples.size(); ++i) {
                        const int ix = bpt_result.samples[i].imagebuffer_x;
                        const int iy = bpt_result.samples[i].imagebuffer_y;
                        const int idx = (thread_id * width * height) + (iy * width + ix);
                        if (is_valid_value(bpt_result.samples[i].value)) {
                            // 得られたサンプルについて、サンプルが現在の画素 (x,y)から発射されたeyeサブパスを含むものだった場合、
                            // Ixy のモンテカルロ推定値はsamples[i].valueそのものなので、そのまま足す。その後、下の画像出力時に発射された回数の総計 (iteration_per_thread * num_threads)で割る。
                            //
                            // 得られたサンプルについて、現在の画素から発射されたeyeサブパスを含むものではなかった場合 (lightサブパスが別の画素(x',y')に到達した場合) は
                            // Ix'y' のモンテカルロ推定値を新しく得たわけだが、この場合、画像全体に対して光源側からサンプルを生成し、たまたまx'y'にヒットしたと考えるため、
                            // このようなサンプルについては最終的に光源から発射した回数の総計 (width * height * iteration_per_thread * num_threads)で割って、画素への寄与とする必要がある。
                            // iteration_per_thread * num_threadsの分は上と共通なので、width * heightで割ってからimage_bufferに足すことで、最終的な画像出力時に帳尻があり、正確な結果になる。
                            if (bpt_result.samples[i].start_from_pixel)
                                image_buffer[idx] = image_buffer[idx] + bpt_result.samples[i].value;
                            else
                                image_buffer[idx] = image_buffer[idx] + bpt_result.samples[i].value / ((double)width * height);
                        }
                    }
                }
            }
        }
    }

    // サンプル数で割る + 左右反転
    std::vector<Color> final_buffer(width * height);
    for (int t = 0; t < num_threads; ++t)
        for (int y = 0; y < height; ++y)
            for (int x = 0; x < width; ++x) {
                final_buffer[y * width + x] = final_buffer[y * width + x] + image_buffer[(t * width * height) + y * width + (width - x - 1)] /
                    ((double)iteration_per_thread * num_threads);
            }
    // 出力
    save_ppm_file("image_bpt.ppm", &final_buffer[0], width, height);
}


};

#endif
```

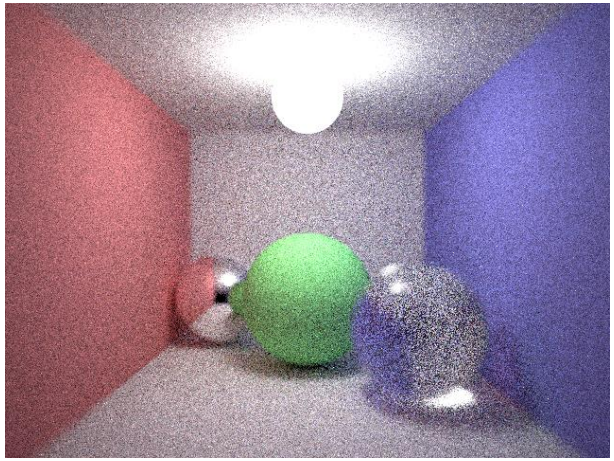
render.h

- 得られたサンプルについて、サンプルが現在の画素(x, y)から発射されたeyeサブパスを含むものだった場合、 $I_{x,y}$のモンテカルロ推定値はsamples[i].valueそのものなので、そのまま足す。その後、下の画像出力時に発射された回数の総計(iteration_per_thread * num_threads)で割る。
- 得られたサンプルについて、現在の画素から発射されたeyeサブパスを含むものではなかった場合（lightサブパスが別の画素(x', y')に到達した場合）は$I_{x',y'}$ のモンテカルロ推定値を新しく得たわけだが、この場合、画像全体に対して光源側からサンプルを生成し、たまたま(x', y') 'にヒットしたと考えるため、このようなサンプルについては最終的に光源から発射した回数の総計(width * height * iteration_per_thread * num_threads)で割って、画素への寄与とする必要がある。
- iteration_per_thread * num_threadsの分は上と共通なので、width * heightで割ってからimage_bufferに足すことで、最終的な画像出力時に帳尻があり、正確な結果になる。

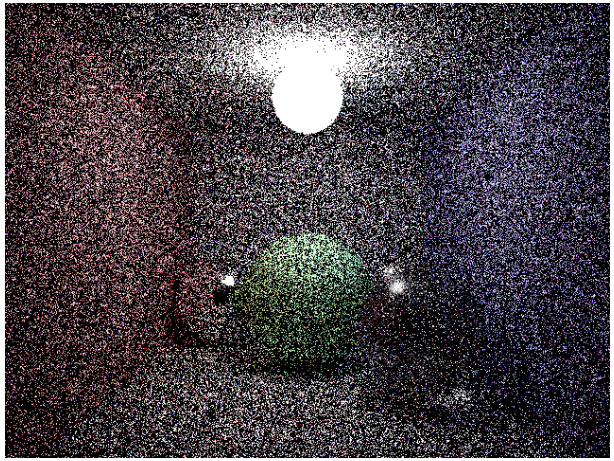
7. レンダリング結果

比較

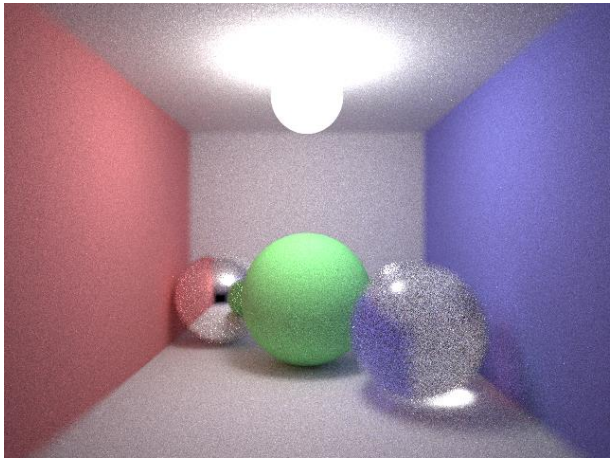
コーネルボックス



パストレーシング
59秒



ライトトレーシング
62秒

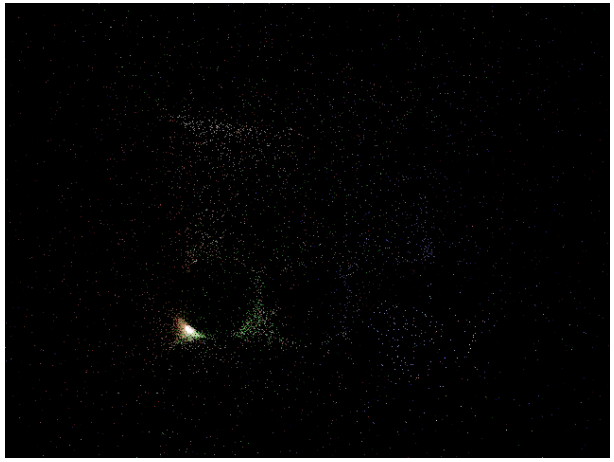


双方向パストレーシング
60秒

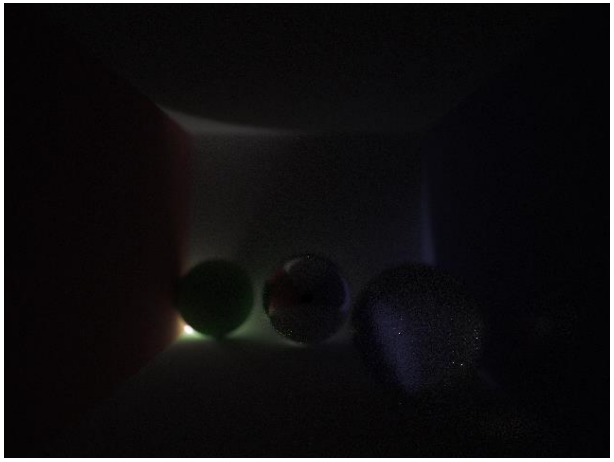
比較

小光源

パストレーシング
61秒



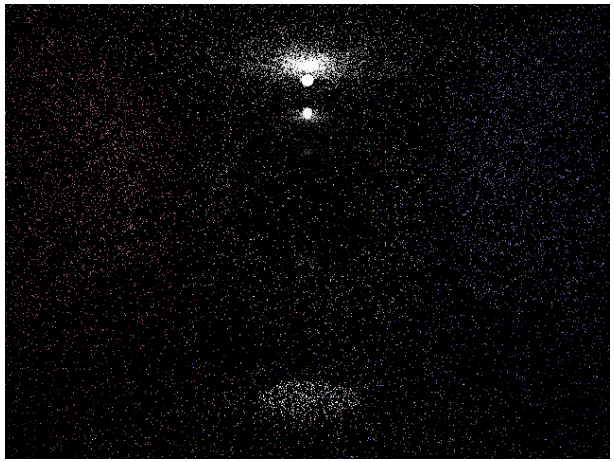
ライトトレーシング
61秒



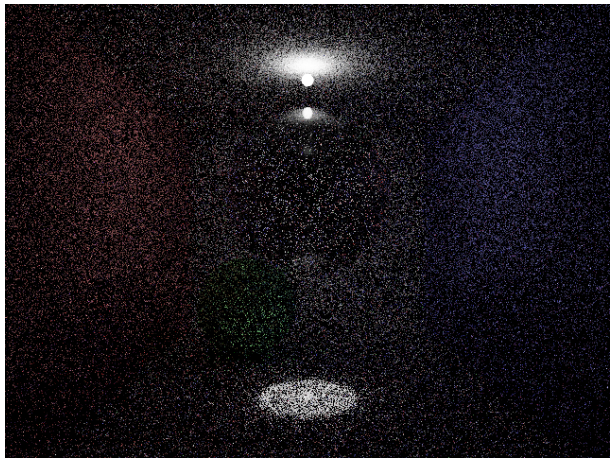
双方向パストレーシング
63秒

比較

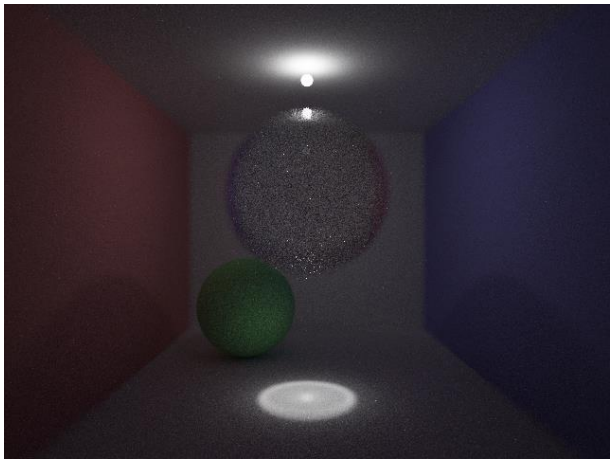
ガラス球



パストレーシング
61秒



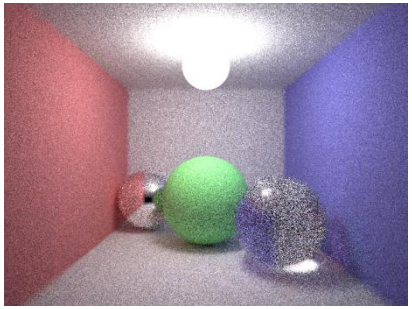
ライトトレーシング
66秒



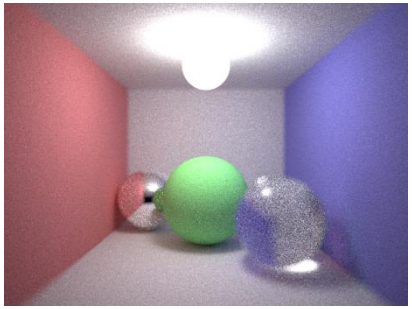
双方向パストレーシング
67秒

収束

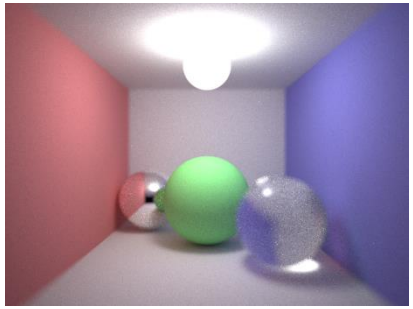
コーネルボックス
双方向パストレーシングの実行回数/pixel



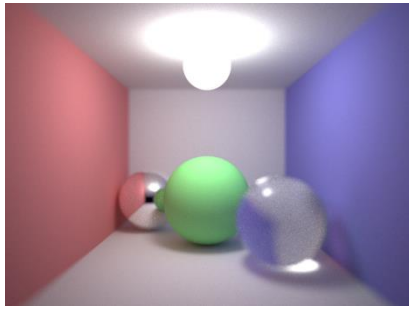
10



50



250



1000

謝辞

- アドバイスや感想を下さった以下の方々に感謝いたします。
  - Yuki OZAWA（@qatnonoil）
  - お餅（@omochi64）

参考文献

- **Eric Veach “Robust Monte Carlo Methods for Light Transport Simulation “**
- Dietger van Antwerpen “A Survey of Importance Sampling Applications in Unbiased Physically Based Rendering”
- Philip Dutre, Philippe Bekaert, Kavita Bala “Advanced Global Illumination, Second Edition”
- Matt Pharr, Greg Humphreys “Physically Based Rendering, Second Edition: From Theory To Implementation”
- Kevin Suffern “Ray Tracing from the Ground Up”
- Computer Graphics “http://www35.atpages.jp/shocker/memoRANDOM/CG/CG.php”