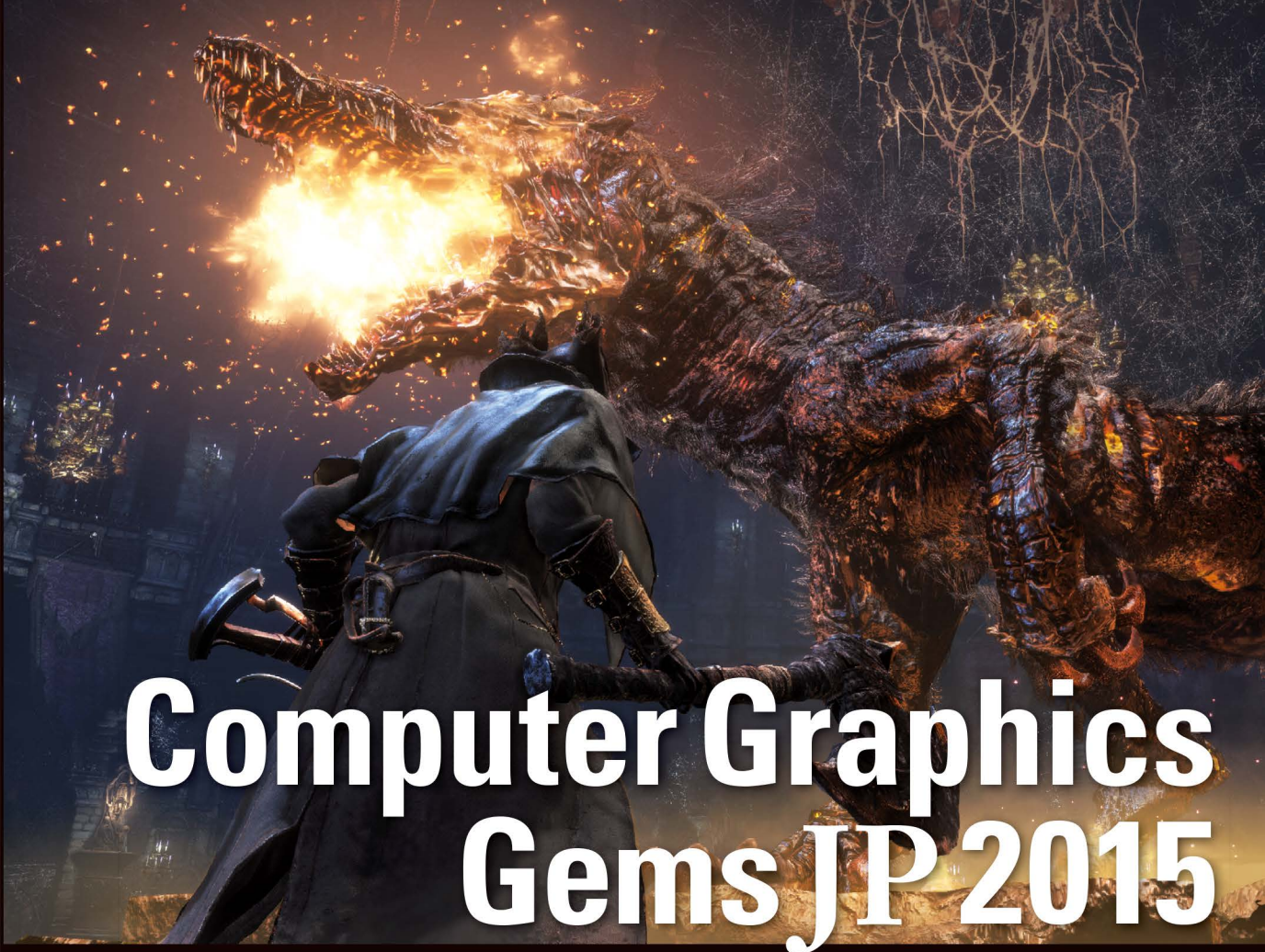

Computer Graphics Gems JP 2015

コンピュータグラフィックス技術の最前線

著者 山本 醍田、鈴木 健太郎、小口 貴弘、德吉 雄介、白鳥 貴亮、向井 智彦、
五十嵐 悠紀、岡部 誠、森本 有紀、上瀧 剛、坂東 洋介

編集 加藤 諒

Born Digital, Inc.

付属データについて

本書の付属データは、ボーンデジタルのWebサイト（http://www.borndigital.co.jp/）から、「書籍 ＞ 書籍サポート」と順に進み、「Computer Graphics Gems JP 2015」のページからダウンロードすることができます。

Contents
目 次

Chapter 2
GPU最適化入門

ソニー・コンピュータエンタテインメント 小口 貴弘　koguchi@rd.scei.sony.co.jp

Chapter 3
球面ガウス関数を用いた動的間接照明の高速近似

株式会社スクウェア・エニックス 徳吉 雄介

Chapter 4
鏡面反射ローブを考慮したバイラテラルフィルター

株式会社スクウェア・エニックス 徳吉 雄介

SECTION 2
アニメーション

Chapter 5
モーションキャプチャ概要

白鳥 貴亮

Chapter 6
画像処理におけるDecision Treeを使った Multiclass Classification

山本 醍田

Chapter 7
スキニング分解

向井 智彦 tmki@acm.org

SECTION 3
イメージマニュピュレーション

SECTION 4
コンピュテーショナルフォトグラフィ

Chapter 11
特徴点照合によるパターンマッチングと姿勢推定

上瀧 剛

Chapter 12
コンピュテーショナルフォトグラフィ：カメラのハックを伴う画像処理

坂東 洋介

Preface
序 文

ようこそComputer Graphics/Visionの世界へようこそ！

2012年から始まったCompute Graphics Gems JPシリーズも、本書で3冊目になりました。

この本はComputer Graphicsって何が出来るの!?　といったよくある質問に対して、とりあえず思いつく限りの厳選メニューを並べてみました。この中で興味がある品はありませんか？　という本です。まずはぱらぱらと本の中身を眺めてみるといいと思います。

本書は、世の中によくあるOpenGL/OpenCV/3D Studio Max/Unityの入門本とは一線を画した内容になっています。前途のような便利道具が次々と世に舞い降りる中、その背景になる理論は次々とブラックボックス化しています。本書はその裏側で動く仕組みを解説し、理解出来るようにすることを目的としています。

ちなみに、2015年という年は日本の学術界にとって躍進の年でした。Siggraph、CVPRといったComputer GraphicsとComputer Visionの世界最高峰の学会で、それぞれ7本/11本の日本人論文が採択されるという大きな成果をあげられました。そんな世界に誇るべき分野なのですが、勉強のとっかかりを掴むのが中々難しい分野でもあるため、本書がそのお手伝いが出来ればなによりです。ちなみにCG ARTSさんらも内容が古くなっていたCG入門書を大幅に改訂し、最新の技術をも広くカバーするようにしました(広告)。

今流行の3D プリンタ/ディープラーニング/バーチャルリアリティヘッドマウントディスプレイといった新しい分野にもComputer Graphicsの技術は頻繁に使われており、覚えておいて損はない分野ですので、興味があれば是非勉強してみることをおすすめします。

── 山本　醍田

Author
著者紹介

山本 醍田

山本 醍田は、組み込み系の仕事に従事するシステムエンジニアである。

ひょんな意気投合から始まった本書もようやく三冊目。2020年ぐらいまではこのシリーズの取り纏めをやる気があるらしい。オーサーをやりたい人募集中なので、やる気ある人はボーンデジタルさんに直訴メール投げてください。最近は色々とネタを思いつきあるのでCGの研究を久々に再開させたところ。その一方で、元々はGlobal Illuminationの人だったのだが、最近はどっぷりとComputer Visionにハマりつつある。歴史小説のオマージュ作品をウェブ連載で始めたところでもある。興味があったらぐぐって読んでみてね。

鈴木 健太郎

ksuzuki@polyphony.co.jp

2012年東京大学理学部卒、2014年同大学大学院情報理工学系研究科修了、同年株式会社ポリフォニー・デジタル入社。

グラフィックスプログラマとしてゲーム開発に従事。リアルタイムレンダリングやオフラインレイトレーシングをはじめとするコンピュータグラフィックス技術全般に興味を持つ。

小口 貴弘

GPUエンジニア　koguchi@rd.scei.sony.co.jp/@koguchit

ソニー・コンピュータエンタテインメント 研究開発本部 LSI 開発部に所属。

2002年に入社以来、プレイステーションを中心に、GPU開発、CG・画像処理についての研究と応用、SDK開発、ゲーム開発支援、熱設計評価、量産試験等に携わる。

そのほかにも、テニス、格闘技・プロレス観戦、子育て、晩酌などに取り組み中。

徳吉 雄介

徳吉 雄介は株式会社スクウェア・エニックスのシニアリサーチャー。

平成19年3月信州大学大学院工学系研究科システム開発工学専攻博士後期課程修了。博士（工学）。同年4月株式会社日立製作所システム開発研究所入社（研究員）。最適化コンパイラの研究開発に従事。ハイパフォーマンスコンピューティング及び組込みプロセッサ向けコンパイラの開発に関わる。平成22年7月より現職。グローバルイルミネーションを中心にレンダリング技術に興味を持つ。

白鳥 貴亮

2007年3月、東京大学大学院情報理工学系研究科電子情報学専攻にて博士号を取得。2007年から2012年までカーネギーメロン大学ロボット工学研究所、およびDisney Researchにて博士研究員として、また2012年から2015年までMicrosoft Researchにて研究員として勤務し、現在Facebook社のOculus Researchにて研究を続けている。モーションキャプチャからアニメーションのデザイン、インタラクションなどキャラクタアニメーション向けのインタフェースについて幅広く研究を行っている。

向井 智彦

東海大学情報通信学部情報メディア学科専任講師。博士（工学）。2006年に豊橋技術科学大学大学院博士後期課程電子・情報工学専攻を修了後は、同学にて助教を務め、2009年より株式会社スクウェアエニックスの研究開発部門におけるシニアリサーチャーを経て、2014年より現職。コンピュータグラフィックスやヒューマノイドアニメーションに関する教育研究に幅広く興味を持つ。

五十嵐 悠紀

2005年お茶の水女子大学理学部情報科学科卒業。2007年東京大学大学院情報理工学系研究科修了、2010年東京大学大学院工学系研究科にて博士（工学）取得。

2010年より日本学術振興会特別研究員PD, RPDとして筑波大学システム情報系に所属。2015年4月より明治大学総合数理学部先端メディアサイエンス学科専任講師、現在に至る。

コンピュータグラフィックスやユーザインタフェースに関する研究に従事。特にファブリケーションに興味を持ち研究を行っている。

岡部 誠

1979年、兵庫県加古川市生まれ。2008年3月、東京大学 大学院情報理工学系研究科 コンピュータ科学専攻にて博士号を取得。2008年4月、マックスプランク研究所のコンピュータグラフィクスグループにポストドクターとして就職。2010年2月、電気通信大学 大学院情報理工学研究科 総合情報学専攻に助教として就職。現在は動画データの分析と映像製作ツールの開発に興味があり、研究を行っている。

森本 有紀

九州大学で博士(芸術工学)を取得後、東大や理研などでの研究員を経て、現在、東京電機大学の講師。元はデザインとかアートに興味があったはずなのだが、なぜかガチのCG分野で戦うはめに。でもCG研究はそれっぽいこともできて面白いので、まあいいや、と思っている。技術的には、関連技術全般の中でも特に今は画像処理やインタラクション技術に興味がある。

上瀧 剛

1980年福岡県生まれ。2007年熊本大学で博士取得後、日立製作所生産技術研究所に勤め、2010年より熊本大学助教。中学生の頃、MSX FANにゲームを投稿して初リジェクト。

画像処理が専門だが、CGや機械学習に興味があり、2006年にスズメレンダラー、2012年にクマ将棋(世界コンピュータ将棋選手権1次予選突破)を作っていた。

坂東 洋介

2001年東京大学理学部卒、2003年同大学大学院情報理工学系研究科修了、同年株式会社東芝入社。コンピュータグラフィクス、コンピュータビジョン、画像処理、無線通信、ストレージシステムの研究開発に従事。並行して2006年東京大学大学院情報理工学系研究科に社会人博士課程入学しコンピュテーショナルフォトグラフィの研究を開始。2010年博士号取得。2011–2013年MIT Media Lab客員研究員。

SECTION

1 レンダリング

Chapter 1
パストレーシングで始める オフライン大域照明レンダリング入門

鈴木 健太郎

概要

コンピュータグラフィックスにおける目標の一つは、より写実的でもっともらしい絵をコンピュータによって生成することです。そのためのレンダリング手法がオフライン、リアルタイム問わず様々に研究されてきました。パストレーシングはそのような手法の基礎とも言うべき手法で、実装は単純ながらも大域照明を実現することが出来ます。発展的なオフラインレンダリングを行うにせよ、リアルタイムレンダリングを追及するにせよ、グラフィックスの諸原理の理解のためには重要な手法です。

本稿では、光学の基礎やレンダリング方程式の解説を行い、それを踏まえたうえでモンテカルロ積分によって具体的な絵を得るためのアルゴリズム、パストレーシングを紹介します。

光学の基礎として光の物体表面での挙動の説明やBRDFの導入を行い、さらに実際に画像をレンダリングするためのモデルとしてレンダリング方程式を説明します。また、高次の積分方程式となるレンダリング方程式を解くためにモンテカルロ積分の初歩についても解説します。さらに、インポータンスサンプリングやロシアンルーレットといった重要な手法の基礎についても触れていきます。完全に動作する具体的なレンダラのソースコードを参考にしながら実装についても説明します。

1.1 はじめに

モデリングやシミュレーション、アニメーションや画像処理など、コンピュータグラフィックスは様々な分野に広がっています[5]。その中でも中心的な存在なのがレンダリングでしょう。レンダリングは大きく分けて二種類の方面で進化を続けてきました。一つは**リアルタイムレンダリング**で、もう一つは**オフラインレンダリング**です。前者は主にゲームやビジュアライゼーションにおいて用いられ、一枚のCGを十数msで計算・表示しなければならないため非常に高速に処理を行わなければなりません。後者は主に映画や静止画のCGにおいて用いられ、一枚のCGの処理に何秒、何時間、時には何日も費やすことで美しさや写実性を追求します(図1.1a)。

従来、リアルタイムレンダリングは計算に使うことのできる時間の差により、オフラインレンダリングに対して品質的に劣るとされてきました。しかし、近年のハードウェアの進化やアルゴリズムの発展に伴いリアルタイムレンダリングにおいてもオフラインレンダリングにも対抗できうる高い品質のCGが得られるようになってきています(図1.1b)。

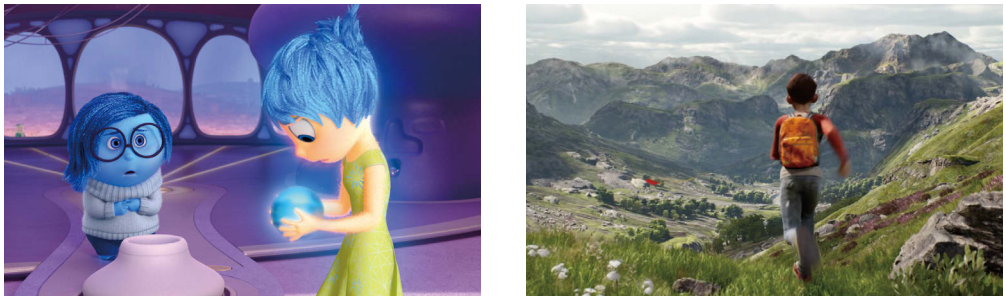

図1.1 (a) インサイド・ヘッド©2015 Disney/Pixar. All Rights Reserved.全国大ヒット上映中！ (b) Unreal Engine 4によるリアルタイムレンダリングのデモ [1] Unreal® Engine, Copyright 1998–2015, Epic Games, Inc. All rights reserved.

それではレンダリングとは具体的に何を指すのでしょうか？ ディスプレイの上に、計算機の内部に構築された仮想的なオブジェクトを表示するには、何を行えばいいのでしょうか？ ヒントは我々が普段モノを見るときの仕組みにあります。人間がモノを見る、というとき、それは光を知覚しているということです。太陽や、蛍光灯や、電灯のような光源から発射された光が周囲の物体に衝突・反射し、その光が眼に飛び込み網膜の上の視細胞が光を受容しその電気信号が脳に伝わることで、人間は光を知覚し、モノを見るのです。この一連の流れは、カメラで写真を撮る時もほとんど同じように行われます。レンダリングをするということは、このような光の振る舞いを計算機によってシミュレーションすることに他なりません。このシミュレーションを通じて、あたかも現実世界の光景のようなCGを得ることが出来るのです。

本稿では光のシミュレーションを物理に基づいて行うことで写実的なコンピュータグラフィックスを計算する方法を紹介していきます。今回紹介する手法の名前を**パストレーシング (Path tracing)**と言い、**レンダリング方程式 (The rendering equation)**と呼ばれる光の支配方程式を直接的に解くことで導出される手法になっています [8]。オフラインレンダリングに分類される手法ですが、実装が分かりやすく、それでいて重要な要素がたくさん入った手法です。この手法は計算を物理に基づくことで、現実世界における光の振る舞いをうまく近似でき、結果としてもっともらしい結果を得ることが出来ます(1.2節)。光が周囲の物体によって何度も反射することで生まれる間接照明の効果も、大域照明(global illumination)を考慮したモデル(レンダリング方程式)を用いることで実現できます(1.3節)。また、これらのモデルはモンテカルロ積分(Monte Carlo integration)と呼ばれる数値的な手法によって解かれます(1.4節)。具体的なアルゴリズムとしてのパストレーシングを1.5節で解説し、さらに細かい実装についても説明します(1.6節)。さらに、物体表面の光の反射についての物理的なモデルを導入することで金属やガラスの様な物体のレンダリングを可能にします(1.7節)。最後に、

パストレーシング以外の様々なオフラインレンダリング手法や発展的な話題について簡単に説明します(1.8節)。

1.2 光に関する物理

1.2.1 物理モデル

今回取り上げるレンダリングアルゴリズムは、光の物理モデルに基づいた物理ベースレンダリングです。物理モデルとは現実世界における様々な物理現象を数式で表現することで、近似的に計算可能にしたものです。物理に基づいたレンダリングを行うことで、大域照明を初めとした物理現象が考慮され、写実的なCGが得られます。

光に関する物理モデルは様々なものが提案されています。これは**光学**という分野で研究されており、幾何光学・波動光学・量子光学などが有名です。さて、現代のコンピュータグラフィックスにおいては基本的に幾何光学を用いることが多いです。カメラや人間の眼に映し出される像は幾何光学によって十分再現可能だからです。他のモデルと比べても、比較的単純なモデルなので計算も効率的に行うことができます。一方、映画「インターステラー」におけるブラックホールなどのレンダリングには幾何光学を超えたモデルが使用されました[6]。幾何光学で表現できない物理現象に対しては、別のモデルを採用することもあり得るわけです。

しかし、大部分のCGは幾何光学で十分なので、ここから先は全て幾何光学に基づいたレンダリングアルゴリズムを紹介していきます。また、大気の光への影響は考慮せず光は真空中を直進する、と仮定します。これは計算や理論を単純化するためです。もし、関与媒質(Participating media)と呼ばれる、煙や炎、雲といった対象をレンダリングするときには空間の媒質密度の変化と光への影響を考慮することになります[11]。

幾何光学に基づいたシミュレーションではレイ(Ray)が大きな役割を果たします。光源から発射されたレイは空間を直進し、物体に衝突すれば反射し、最終的にカメラのセンサーや網膜に衝突することで知覚される、と考えます。ですから、レンダリングも何らかの形でレイを飛ばすことで行われます。この一連のシミュレーションを**レイトレーシング(Ray tracing)**と呼びます。レイトレーシングには狭義の意味と広義の意味があります。狭義の意味でのレイトレーシングはWhittedによる初期の研究を指すことが多いようです[15]。一方、広義の意味でのレイトレーシングは、光線としてのレイを光源やカメラから追跡するアルゴリズムそのものを指します。

光の挙動をレイトレーシングによってシミュレーションする際には、**光の物理量**が重要になってきます。光にまつわる物理シミュレーションを行うわけですから、各種の物理量が必要になるわけです。そこで、幾つかの重要な物理量を紹介します。なお、光の物理量に関する研究は**放射分析学(Radiometry)**と**測光学(Photometry)**という二つの分野が主流で、物理量の単位もそれぞれの分野で異なるものが使われています。グラフィックスの文脈では放射分析学を使うことが多いので、こちらで使われる単位を使用します。

1.2.2 放射束

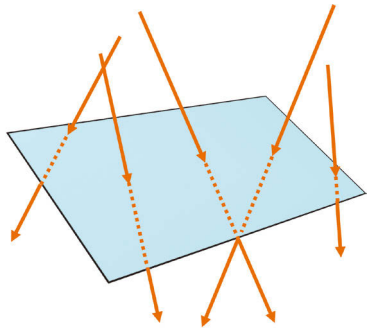

図1.2 放射束のイメージ図。ある領域を単位時間あたりに通過するフォトン数(エネルギー)

最初は**放射束(Flux)**です。これは、単位時間あたりにある領域を通過するフォトン数で、フォトン数はそのままエネルギーと考えることが出来るので、単位時間あたりにある領域を通過するエネルギーであると言えます(図1.2)。個々のフォトンの持つエネルギーの量はスペクトル等によって決まりますが、本稿では深く解説しません。単位は$\frac{\mathrm{J}}{\mathrm{s}}=\mathrm{W}$です。今回は記号$\Phi$を使って表すことにします。

放射束は光の物理量のなかでも基本的な量ですが、単位時間あたりになっていることに気付いたと思います。一般に、CGが対象とするのは静的な光源下で光が光源から発射されて空間で何度も反射した後、もう変化しなくなった(= 平衡状態に至った)状態です。ですからΦは時間の変化に寄らない値になります。世の中には光が時間的に変化する物理現象がありますが(燐光や蛍光)、そういった現象は今回は考慮しません。

1.2.3 放射照度

続いて**放射照度(Irradiance)**です。これは、単位面積あたりの放射束となります(図1.3)。よって、単位は$\frac{\mathrm{W}}{\mathrm{m}^2}$です。記号$E(\vec{x})$を使って表すことにします。放射束との関係式は以下のようになります。

$$E(\vec{x}) = \frac{d\Phi}{dA} \qquad (式1.1)$$

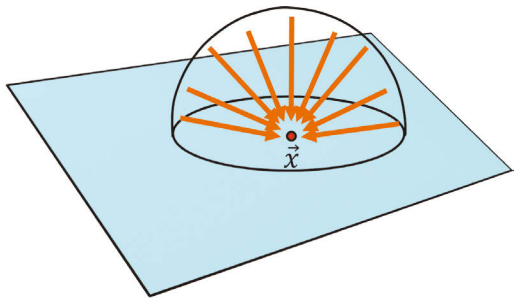

図1.3 放射照度のイメージ図。単位面積当たりの放射束。位置の関数になる

放射束を面積で微分すると放射照度となります。微小面積あたりの放射束が放射照度ということです。放射照度とは、ある領域上のある一点を単位時間あたりに通過するフォトン数である、と考えることが出来ます。また、物体表面上のある一点に周囲から到達する光の総量である、と言うことも出来ます。確かに、その場合の単位は単位面積あたりの放射束となるはずです。放射束の面積密度である、とも言えます。ある一点における量なので、放射照度は位置の関数になります。この位置はレンダリング対象の物体表面上であることが多いですが、空間上の任意の位置においても法線方向さえ決めてやれば、物体表面上の場合と同様にその方向から通過するフォトン数として放射照度を定義出来ます。

1.2.4 立体角

光の物理量を考える上で重要なのが**立体角(Solid angle)**です。これは、二次元平面における角を空間における角に拡張したものです。まず、二次元の平面における角を平面角と呼びます。平面角は、二つの半直線の間の領域のことを指し、半直線間の弧の長さと半径の比を角(radian)として定量化します。例えば、円は2π(rad)の平面角を持ちます。(半径rの円の弧長＝円周＝$2\pi r$なので)

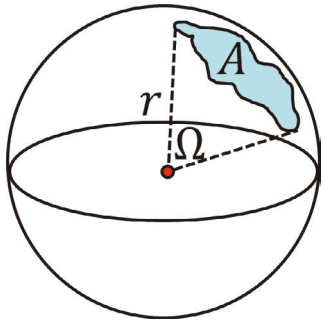

図1.4 立体角

立体角は、平面角と考え方は似ており、ある半直線が空間を動いたときに囲む領域のことを指します。これはある半径の球上の領域となります。そして、領域の面積と半径の二乗の比を立体角(steradian)とします。単位として表記するときはsrです。図1.4のような関係の時、立体角は以下のようになります。

$$\Omega = \frac{A}{r^2} \qquad (式1.2)$$

ただし、Ωを立体角として、Aは領域の面積、rは球の半径です。例えば、球は4π(sr)の立体角を持ちます。(半径rの球の表面積＝$4\pi r^2$なので)立体角は定義より、どの位置を球の中心にするかが問題になります。ある物体の立体角を調べたい場合、その物体をどの位置から見るかによって値は変わってくるということです。位置が分かったら、その位置を中心とした球に物体を投影し、球上で占める面積を計算することで対象の物体をその位置から見たときの立体角を得ることが出来ます。

1.2.5 放射輝度

最後に**放射輝度(Radiance)**です。これは特別重要な量です。なぜなら、物理ベースのレイトレーシングにおいて、一本一本のレイが運ぶ量がまさにこの放射輝度であることが多いからです。放射輝度は、単位立体角あたりの放射照度として定義され、放射照度の立体角密度です。よって、単位は$\frac{\mathrm{W}}{\mathrm{m}^2 \cdot \mathrm{sr}}$となり、以下の式で表現されます。

$$L(\vec{x}, \vec{\omega}) = \frac{dE_{\vec{\omega}}(\vec{x})}{d\sigma(\vec{\omega})} \tag{式1.3}$$

ただし、$L(\vec{x}, \vec{\omega})$は位置$\vec{x}$における方向$\vec{\omega}$からの放射輝度とします。また、$dE_{\vec{\omega}}(\vec{x})$は方向$\vec{\omega}$に垂直な領域における放射照度とします(図1.5)。$d\sigma(\vec{\omega})$は方向$\vec{\omega}$における微小立体角です。放射輝度とは、ある領域上のある一点をある方向に単位時間あたりに通過するフォトン数である、と考えることも出来ます。放射照度はあくまである一点における通過フォトン量であり、方向とは無関係の値でした。一方、放射輝度は方向にも依存した値になります。

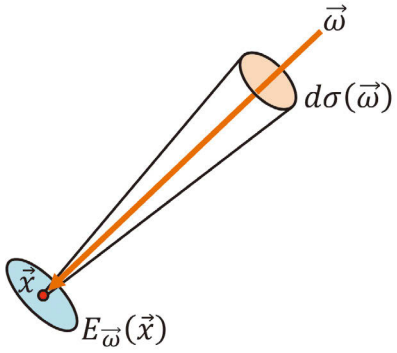

図1.5 放射輝度

さて、方向というのは球上の一点と同じです。ある方向$\vec{\omega}$を考えると、対応する一点が球上に存在します。球上におけるこの一点の周囲の領域はそのまま立体角になります。この立体角$\Delta\sigma(\vec{\omega})$だけを通過するフォトンによる放射照度を$\Delta E_{\vec{\omega}}(\vec{x})$とします。放射輝度は単位立体角あたりの放射照度でしたので、$\frac{\Delta E_{\vec{\omega}}(\vec{x})}{\Delta\sigma(\vec{\omega})}$となります。ここで、領域をどんどん小さくしていけば微小立体角あたりの放射照度となり式1.3が得られます。

放射輝度は基本的にこれで良いのですが、実用上は方向ごとに垂直な領域を考えると手間になることが多いので、位置$\vec{x}$における法線と垂直な領域を考え、その領域における放射照度に関する値とするのが普通です。Lambertのコサイン則より$E_{\vec{\omega}}(\vec{x})\cos\theta = E(\vec{x})$となるので以下の式になります。

$$L(\vec{x}, \vec{\omega}) = \frac{dE(\vec{x})}{d\sigma(\vec{\omega})\cos\theta} = \frac{d^2\Phi(\vec{x})}{dAd\sigma(\vec{\omega})\cos\theta} \tag{式1.4}$$

ここで、式1.1を使いました。

1.2.6 Lambertのコサイン則

最後に、補足としてLambertのコサイン則について簡単に説明します。ある、面積Am^2の領域をΦWのフォトンが通過しているとしたとき、領域Aの放射照度は$E = \frac{\Phi}{A}$となります。また、同じ位置

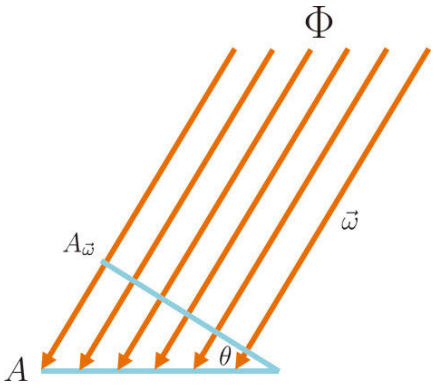

図1.6 Lambertのコサイン則

で、同じ量のフォトンが通過する方向$\vec{\omega}$と垂直な領域を考えると、この領域の面積$A_{\vec{\omega}}$は$A\cos\theta$と等しくなります。(図1.6)によって、この領域における放射照度は以下の式で表現されます。

$$E_{\vec{\omega}} = \frac{\Phi}{A_{\vec{\omega}}} = \frac{\Phi}{A\cos\theta} \qquad (式1.5)$$

よって、一般に$E_{\vec{\omega}}\cos\theta = E$が成り立ちます。

1.3 レンダリング方程式

1.3.1 積分方程式

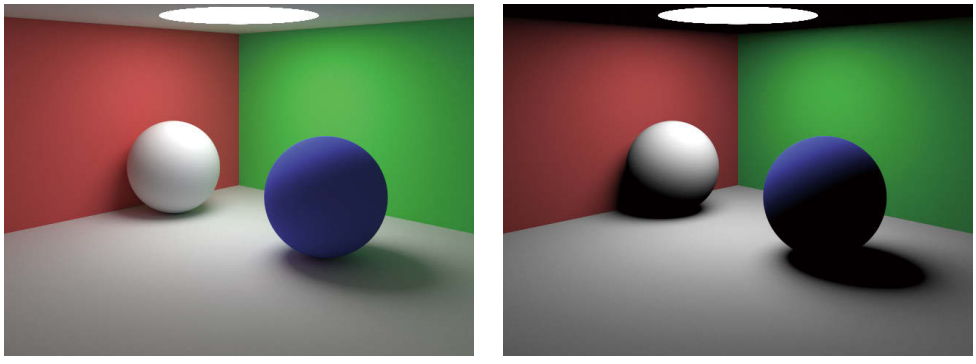

図1.7 左は大域照明あり。右は大域照明なし

それではいよいよレンダリング方程式を説明していきます。レンダリング方程式は、ある空間における光の輸送・伝達を記述した方程式で、真空中での光の挙動のモデルの一つです[8]。このモデルは光の相互反射も考慮された形になっており、即ち**大域照明（Global illumination）**が考慮されたものになっています。大域照明はもっともらしいコンピュータグラフィックスには欠かせません。大域照明のあるなしを比較したのが図1.7です。大域照明を考慮することで、光源から直接光が当たらない影の部分もぼんやりと明るくなっていることが分かります。周囲の壁によって反射された光が到達しているためです。

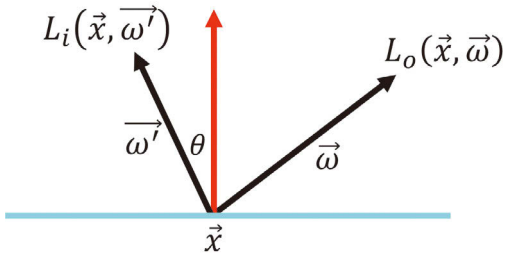

図1.8　レンダリング方程式

レンダリング方程式は図1.8の関係のとき、以下の式で表されます。

$$L_o(\vec{x}, \vec{\omega}) = L_e(\vec{x}, \vec{\omega}) + \int_{\Omega_{\vec{x}}} f_r\left(\vec{x}, \vec{\omega}, \overrightarrow{\omega'}\right) L_i\left(\vec{x}, \overrightarrow{\omega'}\right) \cos\theta d\sigma\left(\overrightarrow{\omega'}\right) \qquad \text{（式1.6）}$$

ここで、$L_o(\vec{x}, \vec{\omega})$は位置$\vec{x}$における方向$\vec{\omega}$への放射輝度、$L_e(\vec{x}, \vec{\omega})$は位置$\vec{x}$における方向$\vec{\omega}$への発光放射輝度、$L_i\left(\vec{x}, \overrightarrow{\omega'}\right)$は位置$\vec{x}$における方向$\overrightarrow{\omega'}$からの放射輝度とします。この場合、光は$-\overrightarrow{\omega'}$方向に進み、位置$\vec{x}$で反射され、$\vec{\omega}$の方向に進むとします。（この辺の表記は文献によって扱いが微妙に異なるため注意が必要です）また、$f_r\left(\vec{x}, \vec{\omega}, \overrightarrow{\omega'}\right)$は**双方向反射率分布関数（Bidirectional reflectance distribution function）**と呼ばれる関数で、物体表面における光の反射の度合いを表します。双方向反射率分布関数は頭文字をとってBRDFと呼ばれることが多く、物体表面の材質を表している項であると言えます。$\Omega_{\vec{x}}$は位置$\vec{x}$における法線方向の半球を表し、レンダリング方程式はこの半球上で積分を行うことで、周囲からやってくる光が$\vec{\omega}$方向にどれくらい反射するかを表現する方程式ということです。$\cos\theta$はコサイン項と呼ばれ、Lambertのコサイン則にでてくる$\cos\theta$と本質的には同じものです。θは方向$\overrightarrow{\omega'}$と$\vec{x}$における法線のなす角です。

さて、最終的に計算したいのは与えられたシーンのCGです。このためには、カメラならばセンサー、眼ならば網膜に入ってくる光の量を計算しなければなりません。詳しくはピンホールカメラの1.5.1項で説明しますが、センサーが受ける光の量を計算するためにはセンサーに入射するレイの放射輝度を得る必要があります。このセンサーに入射するレイとはシーンの物体表面上の点から出射したレイに他なりません。そして、この出射レイの持つ放射輝度こそレンダリング方程式で言うところの$L_o(\vec{x}, \vec{\omega})$です。よって、レンダリングをするとはレンダリング方程式を解き$L_o(\vec{x}, \vec{\omega})$を求める、

ということになります。

それではレンダリング方程式に出てくる各項について詳しく見ていきます。まずは積分についてです。レンダリング方程式の積分域は半球で、積分変数は方向ベクトルになります。半球上の各微小領域(= 微小立体角)を通る放射輝度値(= レイ)を全て集める、というイメージです。集められたそれぞれの放射輝度値はBRDFによって反射方向への値へと変換されます。さて、$\vec{x}$ から $\vec{\omega}$ 方向へのレイとシーンの物体表面との交差点を $\vec{r}(\vec{x}, \vec{\omega})$ とすると以下の関係が成り立ちます。

$$L_i\left(\vec{x}, \vec{\omega'}\right) = L_o\left(\vec{r}\left(\vec{x}, \vec{\omega'}\right), -\vec{\omega'}\right) \tag{式1.7}$$

この関係式を使うことで、式1.6の積分の内部が左辺と同じ関数になり、再帰的な表現になります。このような再帰的な積分方程式は、モンテカルロ積分(1.4節)を用いて数値的に解くことが出来ます。また、$\vec{r}(\vec{x}, \vec{\omega})$はレイトレーシングによって計算できます。レンダリング方程式が再帰的な形をとるということは、レンダリング方程式がシーン内における光の無数の反射を表現しているということを意味します。

続いて、BRDFについて説明します。

1.3.2 双方向反射率分布関数(BRDF)

既に述べたように、BRDFは物体表面における反射の度合いを表現する関数で、入射した光がどれくらいの割合で各方向へ反射されるかを示します。BRDFの定義は以下のようになります[7]。

$$f_r\left(\vec{x}, \vec{\omega}, \vec{\omega'}\right) = \frac{dL_r(\vec{x}, \vec{\omega})}{L_i\left(\vec{x}, \vec{\omega'}\right)\cos\theta d\sigma\left(\vec{\omega'}\right)} = \frac{dL_r(\vec{x}, \vec{\omega})}{dE\left(\vec{x}, \vec{\omega'}\right)} \tag{式1.8}$$

ここで、$L_r(\vec{x}, \vec{\omega})$は反射の放射輝度です。反射の度合いといいつつ、単純な入射する放射輝度値と反射の放射輝度値の比になっていません。これは、レンダリング方程式を式1.6のような単純な形(BRDFと入射放射輝度値とコサイン項の積を積分する形)にするためです。上記の定義を式1.6に代入して積分すると、確かに反射される放射輝度値が得られます。

BRDFの重要な性質が二つあります。それは、ヘルムホルツ(Helmholtz)の相反性とエネルギー保存則です。ヘルムホルツの相反性とは、以下の式で表される性質のことです。

$$f_r\left(\vec{x}, \vec{\omega}, \vec{\omega'}\right) = f_r\left(\vec{x}, \vec{\omega'}, \vec{\omega}\right) \tag{式1.9}$$

これは、ある点における放射輝度の反射は、入射と反射の方向を逆にしても入射するエネルギーが同じなら同じ結果になるということを意味します。カメラからレイトレーシングするタイプのアルゴリズムは大抵何らかの形でこの性質を利用しています(7.1節)。

エネルギー保存則は、以下の式で表される性質のことです。

$$\int_{\Omega_{\vec{x}}} f_r\left(\vec{x}, \vec{\omega}, \vec{\omega'}\right)\cos\theta d\sigma\left(\vec{\omega'}\right) \leq 1, \forall\, \vec{\omega} \tag{式1.10}$$

ここで、θは法線と入射方向$\vec{\omega'}$のなす角です。これは、任意の反射方向について入射する全ての光のエネルギー総和以上に反射することは出来ないということを意味します。

1.3.3 完全拡散面のBRDF

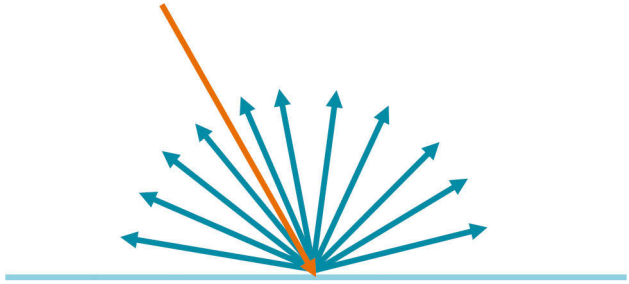

図1.9 完全拡散面において放射輝度は全ての方向に等しく反射される

それでは、具体的な例を挙げてみます。最初は最も単純な、**完全拡散面（ランバート面）**におけるBRDFです。他の例は1.7節で取り上げます。Diffuse BRDFなどと呼ぶこともあります。これは以下の式で表されます。

$$f_d\left(\vec{x}, \vec{\omega}, \vec{\omega'}\right) = \frac{\rho(\vec{x})}{\pi} \qquad (式1.11)$$

ここで、$\rho(\vec{x})$は物体表面の反射率で0から1の範囲の値です。完全拡散面は、どの方向から光が入射しても全ての方向に同じ量の放射輝度を反射する面です。よって、BRDFは定数になります。また、BRDFは式1.10を満たす必要があるため、分母にπが入っています。f_dを式1.10に代入して積分すれば答えはρになるので確かに式を満たします。

1.3.4 半球上の積分

補足として、半球上の積分の方法について説明します。微小立体角$d\sigma(\vec{\omega})$は球面上の微小領域の面積なので、球面座標を用いると$\sin\theta d\theta d\varphi$となります。例えば式1.10は以下のようになります。

$$\int_{\Omega_{\vec{x}}} f_r\left(\vec{x}, \vec{\omega}, \vec{\omega'}\right)\cos\theta d\sigma\left(\vec{\omega'}\right) = \int_0^{2\pi}\int_0^{\frac{\pi}{2}} f_r\left(\vec{x}, \vec{\omega}, \vec{\omega'}\right)\cos\theta\sin\theta d\theta d\phi \qquad (式1.12)$$

1.4 モンテカルロ積分

1.4.1 確率的な数値積分

前節で、レンダリングを行うためには再帰的な積分方程式を解けばよい、というところまで説明しました。それでは、そのような積分はどのように解けばよいのでしょうか。計算機で何らかの積分を解くには数値積分と呼ばれる各種の手法を使うことが多いです。しかし、今回解こうとしている積分の比積分関数は再帰的で高次元、かつ非連続な非常に複雑な形です。このような積分を現実的に解く手法として**モンテカルロ積分（Monte Carlo integration）**があります。モンテカルロ積分には「被積分関

数の値を何らかの形で計算できれば良い」、「高次元な積分についても誤差の収束速度が次元数に影響されず、次元の呪いを受けない」といった特徴があり、まさに今回のような問題を解くのに適しています。

それでは一次元の関数の積分を例にモンテカルロ積分を説明します。例えば以下のような積分を解きたいとします。

$$I = \int_a^b f(x)dx \tag{式1.13}$$

このとき、モンテカルロ積分による推定器は以下のようになります。

$$\hat{I} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{式1.14}$$

ここで、Nはサンプル数、X_iはサンプリングされた個々の確率変数、$p(x)$はX_iをサンプリングする際に使用する**確率密度関数（Probability distribution function）**です（$p(x)$の定義域は$[a, b]$）。モンテカルロ積分はN個のサンプルを$p(x)$に従って発生させ、そのサンプルを使って被積分関数を評価、サンプル数で割ることによって積分の結果を推定します。例えば$p(x) = \frac{1}{b-a}$を使うとすると、これは一様分布なのでa以上b以下の値をランダムかつ等しい確率で一つサンプリングしたものがX_iになります。モンテカルロ積分は、確率的なアルゴリズムなので実行するたびに結果は異なりますが、Nを大きくしていくことで推定値は真値に収束します。このとき、真値に対する誤差（標準偏差）は$O(\frac{1}{\sqrt{N}})$の速度で減少します。これは、モンテカルロ積分においては誤差を半分にするにはサンプル数Nを四倍にしないといけないことを意味します。実は、モンテカルロ積分は誤差収束の観点からするとあまり良い手法ではなく、他の数値積分手法が使える場合はそちらを使ったほうが良いでしょう。

さて、では$p(x)$はどのように決定すれば良いのかについてです。一般に$p(x)$は被積分関数$f(x)$の形状に近いほど良いとされています。極端な話、$p(x) = \frac{f(x)}{c}$とすればモンテカルロ積分の式は以下のようになり推定値の誤差は0になります。

$$\hat{I} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{\frac{f(X_i)}{c}} = \frac{1}{N}\sum_{i=1}^{N}c = c \tag{式1.15}$$

現実的には被積分関数は複雑な形状をしており、それに従ったサンプルを生成するのは難しいですが[*1]、被積分関数についての知識を活用してなるべく似た形状の確率密度を設計することが広く行われています。これを**インポータンスサンプリング（Importance sampling）**と呼びます。例えばレンダリング方程式の例なら、被積分関数にBRDF項が含まれているので、BRDFの形状に基づいたインポータンスサンプリングを行いサンプル（レンダリング方程式の場合は次のレイの方向）を生成する、ということがしばしば行われます。

＊1　メトロポリス光輸送法と呼ばれるより進んだ手法は、被積分関数を確率密度関数としてサンプリングを行う優れたレンダリング手法です。

以上の議論は一次元関数を対象にしていましたが、被積分関数が多次元になっても同様の議論が出来ます。その際、確率密度関数も多次元になります。たとえば、レンダリング方程式なら、積分変数は方向ベクトルでしたので確率密度関数も方向についての関数になります。サンプリングも、半球上から一点(= 方向)をサンプリングする、ということになります。モンテカルロ積分は確率密度関数の測度と積分の測度を揃える必要がありますが、この周辺の話題については本稿では深く説明しません。気になる人は関連文献[13]を調べてみてください。

1.5 パストレーシング

1.5.1 ピンホールカメラモデル

いよいよ実際のレンダリングアルゴリズムであるパストレーシングの解説に移ります。そのために、最終的な画像を得るためのモデルとしてピンホールカメラモデルを導入します。箱に小さい穴を開けると、外の景色が箱の中に写りこみます。これがピンホールカメラです。ピンホールカメラモデルでも、まず極小の穴としての点を考えます。これがカメラの位置に相当します。そして、レンダリングしたいシーンが写りこむ平面がシーン内に存在すると考えます。これが仮想的なカメラのイメージセンサーとなり、レンダリング画像そのものになります。

光源から出た光はシーン内を反射してピンホール、すなわちカメラ位置を通ってイメージセンサーへと至ると考えます。しかし、光源側からレイトレーシングを行ってシミュレーションを行うのは一般にあまり良い方法ではありません。というのも、シーン全体からするとイメージセンサーの大きさはわずかなので、なかなかイメージセンサーへと到達しないからです。さらに、ピンホールカメラモデルではそもそも点であるカメラ位置を通らなければならず、これは不可能です。

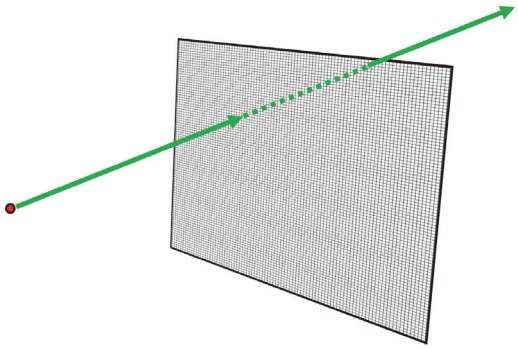

図1.10 ピンホールカメラモデル。左側の赤い点(カメラ位置でありピンホール位置)から右側のイメージセンサー上の個々のピクセル領域に向かってレイを飛ばす

そこで、カメラ側からレイトレーシングを行うという発想になります。また、ピンホールカメラそのままだと像が上下逆に写りこんでしまうため、イメージセンサーの位置もピンホールの前方に配

置します。そして、ピンホールの位置からイメージセンサー上の各ピクセルに向かってレイを飛ばし[*2]、交差点におけるセンサー方向への放射輝度値を計算することになります(図1.11)。交差した位置を$\vec{x}$とすれば、式1.6を使ってカメラへと向かう放射輝度が計算できます。それでは、レンダリング方程式を解く方法について次の節で説明していきます。

1.5.2 モンテカルロ積分による導出

レンダリング方程式とは式1.6で表される積分方程式でした。そこで、積分項をモンテカルロ積分を使って推定します。これは以下の式になります。

$$\hat{L}_o(\vec{x}, \vec{\omega}) = L_e(\vec{x}, \vec{\omega}) + \frac{1}{N}\sum_{i=1}^{N}\frac{f_r(\vec{x}, \vec{\omega}, \vec{\omega_i'})L_i(\vec{x}, \vec{\omega_i'})\cos\theta}{p(\vec{\omega_i'})} \quad \text{(式1.16)}$$

ここで式1.7を使い、右辺も推定値にすると以下の式になります。

$$\hat{L}_o(\vec{x}, \vec{\omega}) = L_e(\vec{x}, \vec{\omega}) + \frac{1}{N}\sum_{i=1}^{N}\frac{f_r(\vec{x}, \vec{\omega}, \vec{\omega_i'})\hat{L}_o(\vec{r}(\vec{x}, \vec{\omega_i'}), -\vec{\omega_i'})\cos\theta}{p(\vec{\omega_i'})} \quad \text{(式1.17)}$$

右辺の$\hat{L}_o$を左辺と同様にモンテカルロ積分で推定することにして、以上の式を再帰的に評価すれば任意の位置、任意の方向への放射輝度値を数値的に計算することが出来、レンダリングを実行できます。この場合$\vec{r}(\vec{x}, \vec{\omega'})$を評価する必要があるため、具体的なアルゴリズムとしては、まず確率密度関数$p(\vec{\omega_i'})$に基づいてN個の方向をサンプリングし、その方向にレイトレーシングすることで$\vec{r}$を評価する、ということを再帰的に繰り返します。つまり、一回のモンテカルロ積分が一回のレイの反射に相当します。このようにカメラ側からレイトレーシングと反射を繰り返していくと、どんどん光の進行方向とは逆方向に計算が進行します。最終的に光源にレイがヒットすることでL_eが何らかの

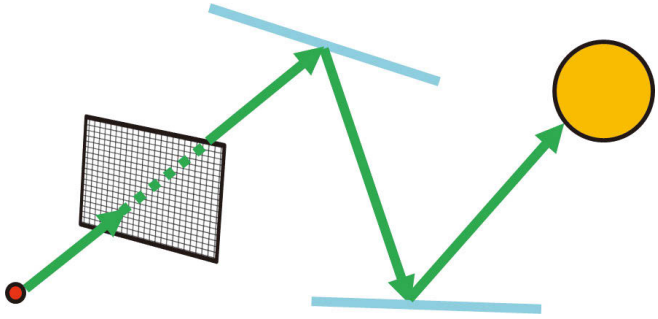

図1.11 パストレーシング。カメラ側からシーンにレイを飛ばし、シーン内で反射を繰り返し、最終的に光源(オレンジの球)に至る

＊2 各ピクセルはイメージセンサー上で一定の領域を占めるため、領域内を細かく区切り、それぞれの点の方向にピンホールからレイを飛ばします。これによりレンダリングされる画像はアンチエイリアスされたものになります。これは、実の所各ピクセルの占める領域に対応する放射束を求めていることに他ならず、ある種のモンテカルロ積分を解いていると言えます。今回の例として実装したパストレーシングでは単純に一様に区切ってその平均をとっていますが、ピクセル上で一様の応答を示すセンサーであると仮定していることに他なりません[10]。

値を持ち、モンテカルロ積分の値を得ることができ、イメージセンサー上の画像の色となります。光源からさらに先に反射していくことも考えられますが、今回の例の実装では光源の反射率を0としてそこで処理を終えます。

しかし、ここで一つ問題が生まれます。一回のモンテカルロ積分の評価のたびにN個のサンプルを生成していると、それぞれのサンプル(= レイ)の先でまたN個のサンプルを生成してN回レイトレーシングを行う必要があり、これを繰り返していくと指数的にモンテカルロ積分の評価回数が増えていってしまいます。一回のモンテカルロ積分の評価は一回分のレイの反射に相当しますが、これでは数回の反射ですらうまく取り扱うことができなくなってしまいます。そこで、パストレーシングではN=1とすることで一回のサンプリングの回数を1回のみに制限します。こうすると指数の底が1になるため評価回数が指数的に増加することがなくなり、現実的に問題を解くことが可能になるのです。図1.12はパストレーシングのイメージ図です。パストレーシングでは各ピクセル領域に何度もレイを飛ばし、都度対応するL_oをモンテカルロ積分によって計算し、推定値の平均をとることで最終的な結果とします[3]。

1.5.3 ロシアンルーレット

モンテカルロ積分を使って再帰的にL_oを推定していくわけですが、このままだと無限回再帰することになってしまいます。これではプログラムとして実装できないため、途中で再帰を打ち切るためにロシアンルーレットと呼ばれる手法が使われることがあります。

ある関数$f(x)$を評価するとき適当な確率Pを決めてやって、その確率に従って$f(x)$の評価を行うか行わないかを決めるのがロシアンルーレットです。

$$\hat{f}(x) = \begin{cases} \frac{f(x)}{P} & (\xi \leq P) \\ 0 & (otherwise) \end{cases} \qquad (式1.18)$$

ただし、ξは$[0, 1]$で一様に分布する確率変数です。このようにして計算される$\hat{f}(x)$はあくまで$f(x)$の推定値になり、Pが$f(x)$の形状に近ければ近いほど誤差が小さくなります。

この手法をモンテカルロ積分に応用すると、ある適当な確率Pに基づいてモンテカルロ積分を実行するかしないかを決め、実行した場合はその推定値をPで割ってやれば良いという事になります。このようにすると再帰の回数が無闇に大きくならず途中で打ち切られるようになり、現実的に計算することが可能になります。

しかし、再帰回数が確率的に変動するためやや扱いにくい側面もあります。今回例として取り上げるパストレーシング実装ではロシアンルーレットによって再帰を打ち切ることはせず、事前に決めた回数だけ反射を計算したら強制的に再帰を打ち切っています。このようにすることで再帰回数が固定化され、トータルの計算コストの見積もりがしやすくなる一方、光源から事前に決めた回数よりも多い回数反射してカメラに到達するような経路については最終的な画像への影響が無視されてしまい、その分がバイアスとして誤差になってしまうという欠点もあります。

1.6 具体的な実装

1.6.1 シーン定義

それでは具体的な実装の解説をしていきます。例として取り上げるレンダラのソースコード全体は書籍のウェブサイトなどからダウンロードできると思いますので参考にしてください。またソースコード全体は *https://github.com/githole/gemspt* でも公開しています。

今回のレンダリング対象は図1.7のようなシーンとします。このシーンは全て球のみで構成されています。レイと球の交差判定は直線と球の交差判定になりますが、これは単純な二次方程式に帰着するので扱いやすいためです。

1.6.2 カメラからのレイの発射

最初に、カメラからシーンに向かってレイを飛ばさなければなりません。今回は1.5.1項で説明したようにピンホールカメラモデルを使うのでカメラ位置から、個々のピクセルに対応するイメージセンサー上の座標に向かってレイを飛ばします。

```
for (int y = 0; y < height; ++y) {
  std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%   \r";
#pragma omp parallel for schedule(static) // OpenMP
  for (int x = 0; x < width; ++x) {
    Random random(y * width + x + 1);

    const int image_index = (height - y - 1) * width + x;
    // num_subpixel x num_subpixel のスーパーサンプリング。
    for (int sy = 0; sy < num_subpixel; ++sy) {
      for (int sx = 0; sx < num_subpixel; ++sx) {
        Color accumulated_radiance = Color();
        // 一つのサブピクセルあたりsamples回サンプリングする。
        for (int s = 0; s < num_sample_per_subpixel; s ++) {
          const double rate = (1.0 / num_subpixel);
          const double r1 = sx * rate + rate / 2.0;
          const double r2 = sy * rate + rate / 2.0;
          // イメージセンサー上の位置。
          const Vec position_on_sensor =
            sensor_center +
            sensor_x_vec * ((r1 + x) / width - 0.5) +
            sensor_y_vec * ((r2 + y) / height- 0.5);
          // レイを飛ばす方向。
          const Vec dir = normalize(position_on_sensor - camera_position);

          accumulated_radiance = accumulated_radiance +
            radiance(Ray(camera_position, dir), random, 0)
            / (double)num_sample_per_subpixel / (double)(num_subpixel * num_subpixel);
        }
        image[image_index] = image[image_index] + accumulated_radiance;
      }
    }
```