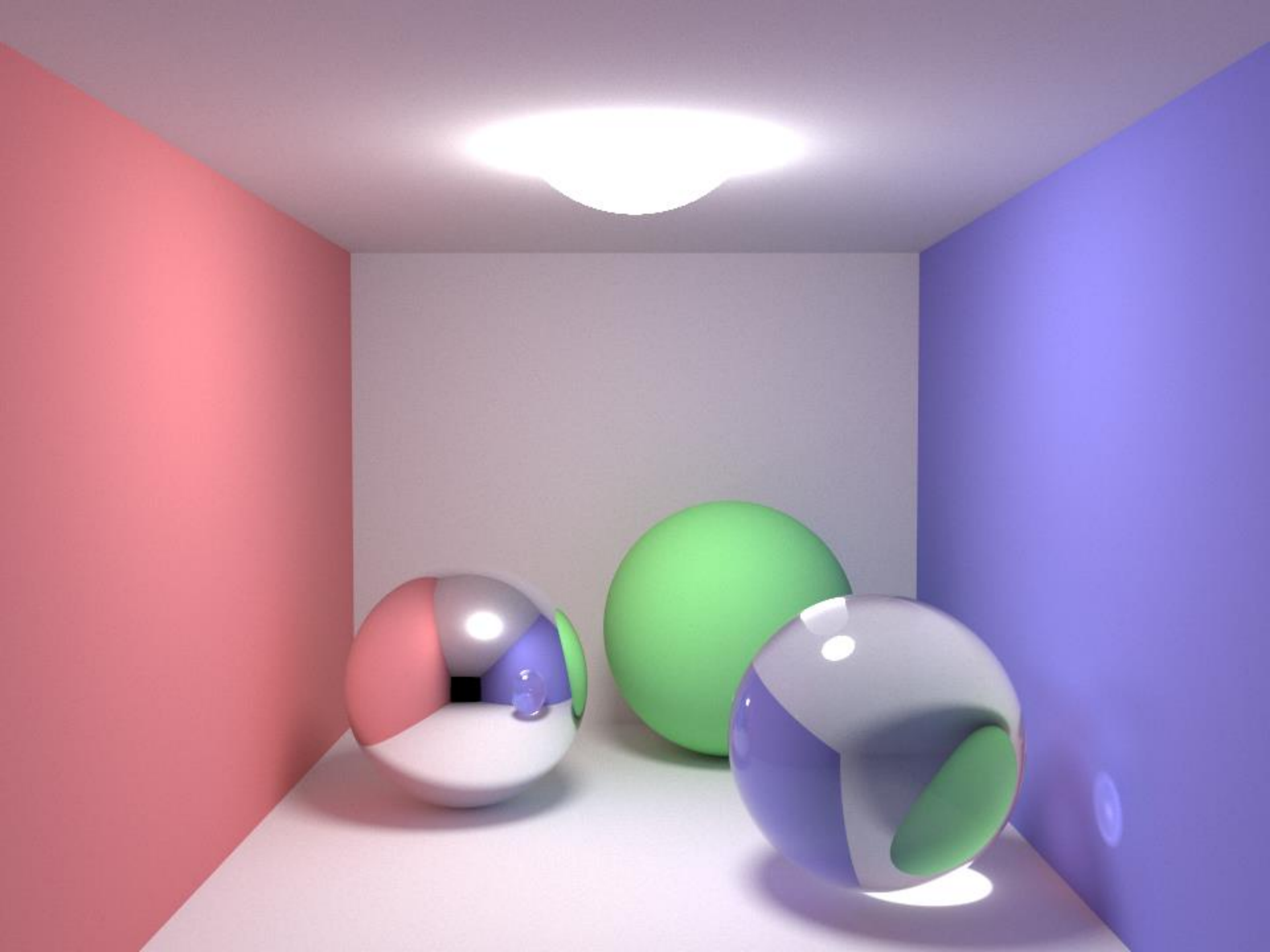
物理ベースレンダラ
edupt解説

2014/06/13 Ver.1.03

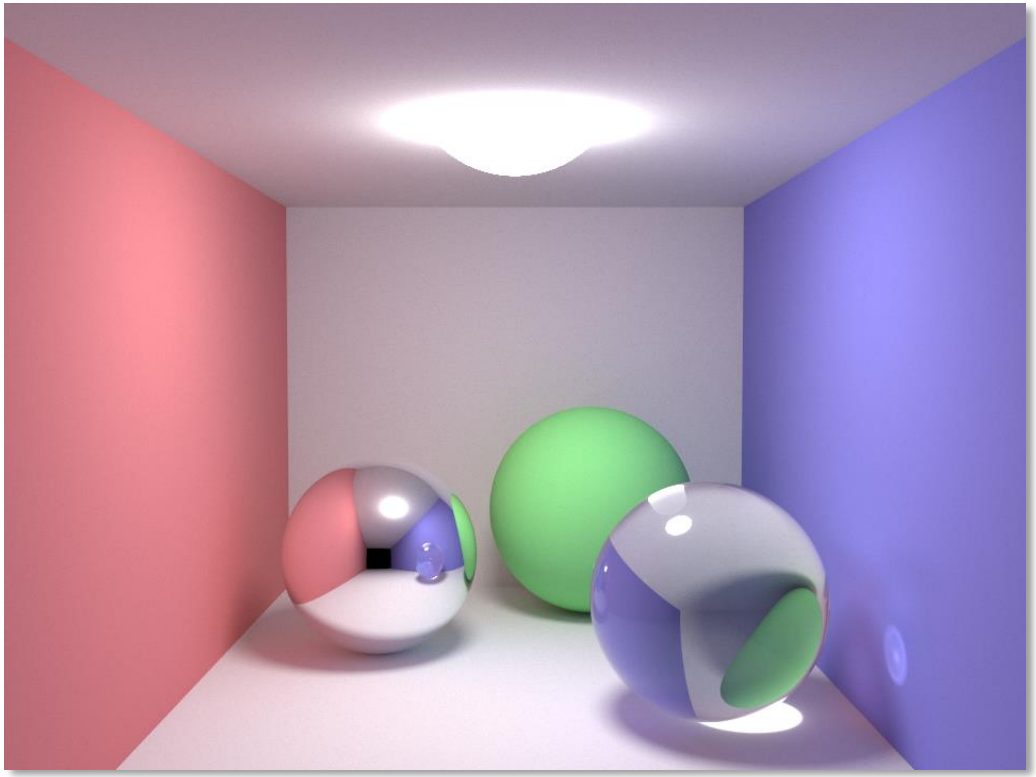
hole (@h013)

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



eduptとは

シンプルでコンパクトな物理ベースレンダラ



edupとは

smallpt をベースに修正・拡張

C++で記述

日本語によるコメント付き

教育的

smallpt (http://www.kevinbeason.com/smallpt/) はKevin Beason氏によってC++で書かれた99行のパストレーシングによるレンダラ

eduptとは

Githubでコード公開

https://github.com/githole/edupt

このスライドは

eduptのコード解説

＋

基礎的な**物理ベースCG**の解説

＋

基礎的な**パストレーシング**の解説

目次

1.物理ベースコンピュータグラフィックス

物理ベースコンピュータグラフィックス

- **まえおき**

  eduptはパストレーシングと呼ばれるアルゴリズムによる物理ベースレンダラですが、そもそも物理ベースCGとは何なのか？何が嬉しいのか？何ができるのか？について

物理ベースコンピュータグラフィックス

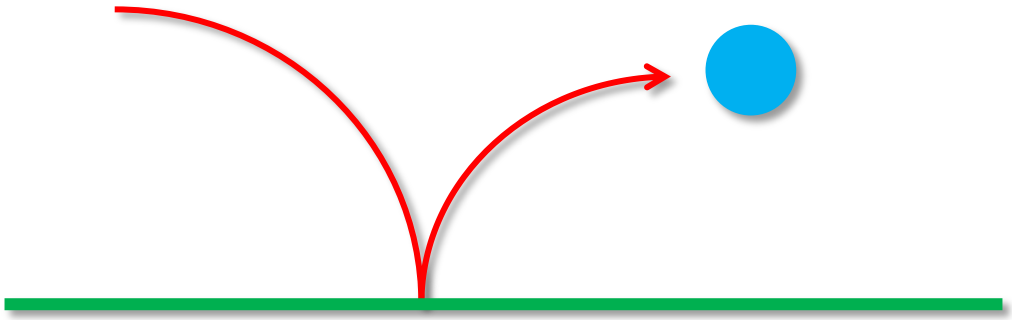
＝

何らかの物理モデルに基づいて計算された
コンピュータグラフィックス

物理モデル

- 現実世界における各種物理現象を**数式で表現**したもの
  - 物理モデルによって各種物理現象が**近似的に計算可能**になる。

$$m\vec{a} = \vec{F}$$



ニュートン力学も物理モデルの一種

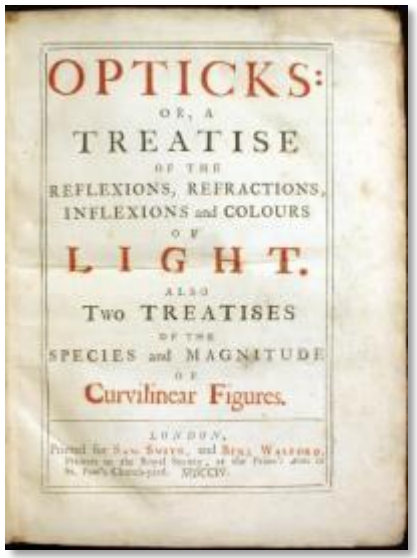
物理ベースじゃないかもしれないコンピュータグラフィックス

- 何らかの物理モデルに基づいていても、**十分にそのモデルに従っていない**計算によって得られたCG
  - 光の相互反射を考慮していないようなCG
- **そもそも何の物理モデルに基づいていない**計算によって得られたCG
  - エネルギー保存則に従っておらず、光がどんどん増幅していくようなCG

物理ベースの嬉しいところ

- 現実の近似としての物理モデルを使う＝**現実らしいCG**が得られる！（＝フォトリアル）
- 各種パラメータ（材質や光源の設定等）について、現実世界の値を**そのまま**使える！
- 既に存在する物理モデルを基礎においているため、**理論的な安心感**がある！
- などなど

コーネルボックス



光学

どんな物理モデルを使う？

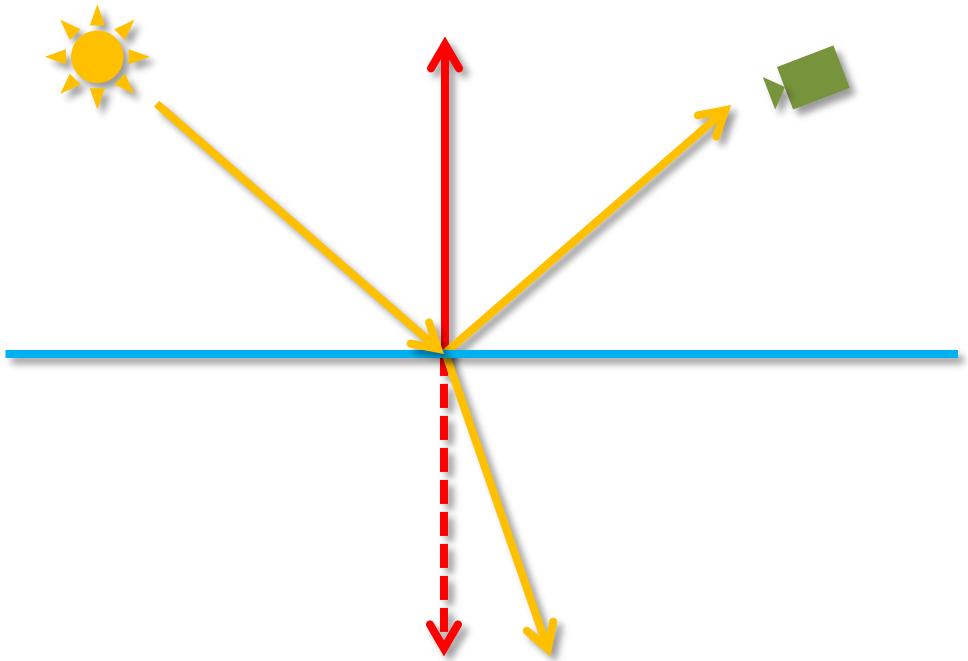
- CGにおいて考慮すべき物理モデルは光に関する物理学、光学についてのモデル
- 様々な光学モデル
  - **幾何光学**
  - 波動光学
  - 量子光学
  - etc

現代のCGでは、基本的に幾何光学をモデルとして用いる。これは、目に映る風景や写真によって描き出される像が幾何光学によって十分再現できるからである。

※もちろん、幾何光学では再現できない物理現象もあり、そういった現象を再現したい場合は別のモデルを採用する。

幾何光学

- 光の伝播を**光線**として記述
  - 光線＝Ray
  - いわゆるレイトレーシングのレイとはこの光線のこと。



CG的には何ができるのか？

- 幾何光学で表現できることは何でもできる
  - **グローバルイルミネーション**
  - 被写界深度
  - モーションブラー
  - etc

eduptでは特に光の相互反射による影響を考慮したグローバルイルミネーションの実現を重視している。幾何光学でグローバルイルミネーションを実現するための物理モデルはこのスライドの後々取り扱う。

2.eduptコード解説
（ユーティリティ編）

2.eduptコード解説（ユーティリティ編）

- **まえおき**

  eduptにおける、各種ユーティリティ関数についての解説。

eduptファイル概要

- **main.cpp**
  edupt::render()を呼び出すだけ。
- **constant.h**
  各種定数。
- **intersection.h**
  交差判定の結果を格納する構造体IntersectionとHitpoint。
- **material.h**
  Color型と物体材質について記述するReflectionTypeと屈折率の設定。
- **ppm.h**
  レンダリング結果をppm画像として書き出すための関数save_ppm_file()。
- **radiance.h**
  ある方向からの放射輝度を得る関数radiance()。パストレーシングのメイン処理。
- **random.h**
  乱数生成クラスXorShift。
- **ray.h**
  一つ一つの光線、レイを表現する構造体Ray。
- **render.h**
  レンダリング画像サイズやサンプル数を受け取り、radiance()を使って各ピクセルの具体的な値を決定する関数render()。
- **scene.h**
  レンダリングするシーンのデータspheres[]とそのシーンに対する交差判定を行う関数intersect_scene()。
- **sphere.h**
  基本的な形状としての球を表現するSphere構造体。
- **vec.h**
  ベクトルを表現する構造体Vec。

eduptファイル概要

- **main.cpp**
  edupt::render()を呼び出すだけ。
- **constant.h**
  各種定数。
- **intersection.h**
  交差判定の結果を格納する構造体IntersectionとHitpoint。
- **material.h**
  Color型と物体材質について記述するReflectionTypeと屈折率の設定。
- **ppm.h**
  レンダリング結果をppm画像として書き出すための関数save_ppm_file()。
- **radiance.h**
  ある方向からの放射輝度を得る関数radiance()。パストレーシングのメイン処理。
- **random.h**
  乱数生成クラスXorShift。
- **ray.h**
  一つ一つの光線、レイを表現する構造体Ray。
- **render.h**
  レンダリング画像サイズやサンプル数を受け取り、radiance()を使って各ピクセルの具体的な値を決定する関数render()。
- **scene.h**
  レンダリングするシーンのデータspheres[]とそのシーンに対する交差判定を行う関数intersect_scene()。
- **sphere.h**
  基本的な形状としての球を表現するSphere構造体。
- **vec.h**
  ベクトルを表現する構造体Vec。

2.1 各種定数

constant.h

```
#ifndef _CONSTANT_H_
#define _CONSTANT_H_

namespace edupt {

const double kPI = 3.14159265358979323846;
const double kINF = 1e128;
const double kEPS = 1e-6;

};

#endif
```

- 各種定数を記録したヘッダ
  - kPI
    - 円周率π。
  - kINF
    - 非常に大きい定数。
  - kEPS
    - 0.0に非常に近い定数。交差判定等で使う。（球の交差判定のところで解説）

2.2 画像出力

ppm.h

```
#ifndef _PPM_H_
#define _PPM_H_

#include <string>
#include <cstdlib>
#include <cstdio>
#include <vector>

#include "material.h"

namespace edupt {

inline double clamp(double x){
    if (x < 0.0)
        return 0.0;
    if (x > 1.0)
        return 1.0;
    return x;
}

inline int to_int(double x){
    return int(pow(clamp(x), 1/2.2) * 255 + 0.5);
}

void save_ppm_file(const std::string &filename, const Color *image, const int width, const int height) {
    FILE *f = fopen(filename.c_str(), "wb");
    fprintf(f, "P3\n%d %d\n%d\n", width, height, 255);
    for (int i = 0; i < width * height; i++)
        fprintf(f,"%d %d %d ", to_int(image[i].x), to_int(image[i].y), to_int(image[i].z));
    fclose(f);
}


};

#endif
```

save_ppm_file()

- **レンダリングした結果を保存するための関数**
  - Color型の配列を受け取る。
  - Color型はRGBデータをそれぞれx,y,zメンバ変数で保存している。

```
void save_ppm_file(const std::string &filename, const Color *image, const int width, const int height) {
    FILE *f = fopen(filename.c_str(), "wb");
    fprintf(f, "P3\n%d %d\n%d\n", width, height, 255);
    for (int i = 0; i < width * height; i++)
        fprintf(f,"%d %d %d ", to_int(image[i].x), to_int(image[i].y), to_int(image[i].z));
    fclose(f);
}
```

save_ppm_file()

- **edupはppm形式で保存**
  - http://netpbm.sourceforge.net/doc/ppm.html
  - ppmはテキスト形式で画像を記録できるフォーマット。
  - ヘッダに”P3 (改行) 幅 高さ (改行) 255”と記録することで、指定したサイズの**RGB**画像（RGBの階調は256段階）を記録する、という指定になる。
  - ヘッダに続けて”R値 G値 B値“を画素数の分だけ並べる。

```
void save_ppm_file(const std::string &filename, const Color *image, const int width, const int height) {
    FILE *f = fopen(filename.c_str(), "wb");
    fprintf(f, "P3\n%d %d\n%d\n", width, height, 255);
    for (int i = 0; i < width * height; i++)
        fprintf(f,"%d %d %d ", to_int(image[i].x), to_int(image[i].y), to_int(image[i].z));
    fclose(f);
}
```

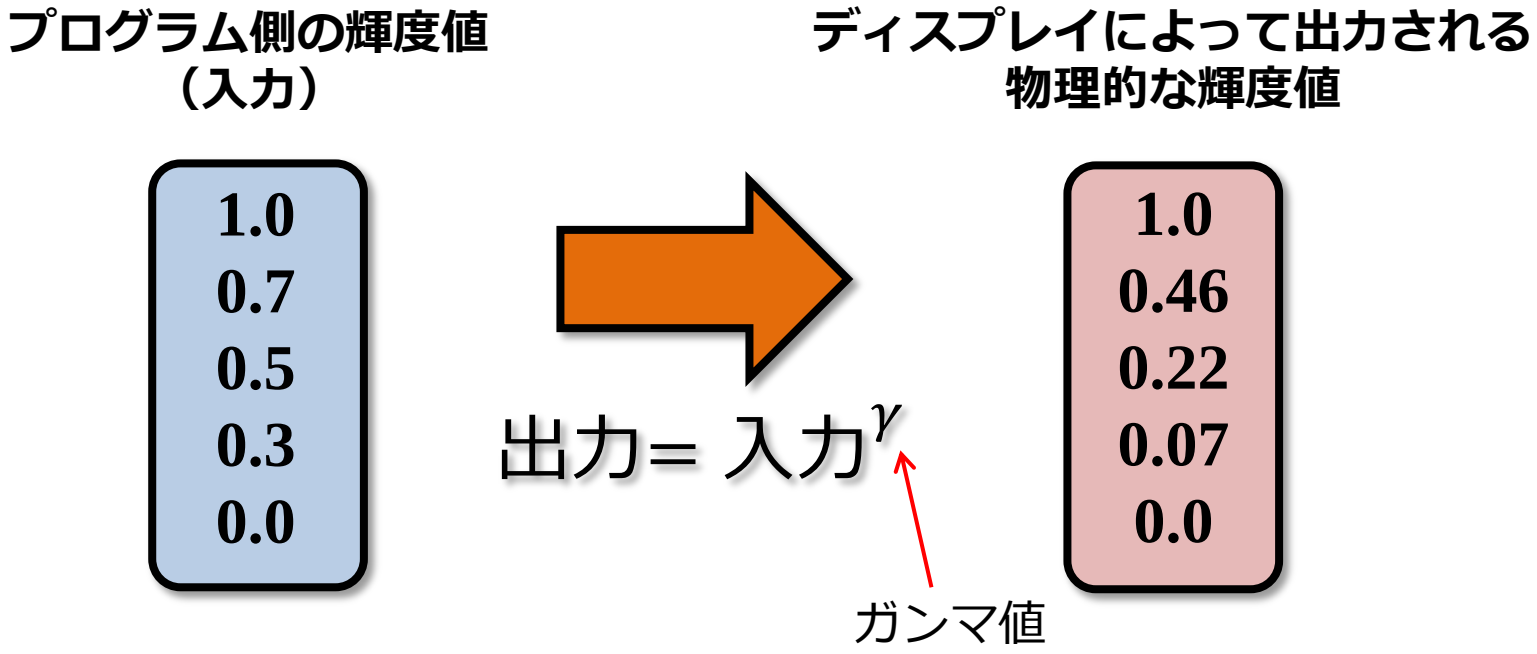
clamp()とto_int()

- **レンダリング結果の配列（Color型）の各値はto_int関数で0から255の256階調の値に変換されて記録される**
  1. 各値xはclamp()によって0から1の範囲に詰められる
  2. $\frac{1}{2.2}$乗する（ガンマ補正）
  3. 255倍する（0から255の範囲になる）
  4. 0.5を足してintにキャストする（小数第一位で四捨五入）

```
inline double clamp(double x){
    if (x < 0.0)
        return 0.0;
    if (x > 1.0)
        return 1.0;
    return x;
}

inline int to_int(double x){
    return int(pow(clamp(x), 1/2.2) * 255 + 0.5);
}
```

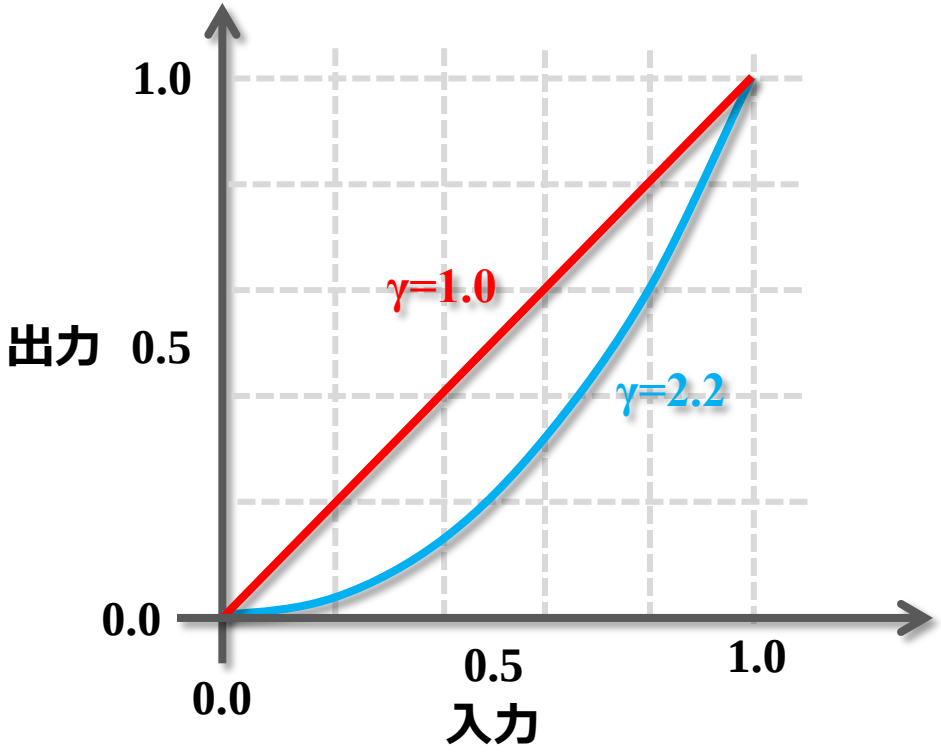
ガンマ

- **ディスプレイのガンマ値**
  - 一般的なディスプレイは入力値と出力値の間に非線形な関係が成り立つ。



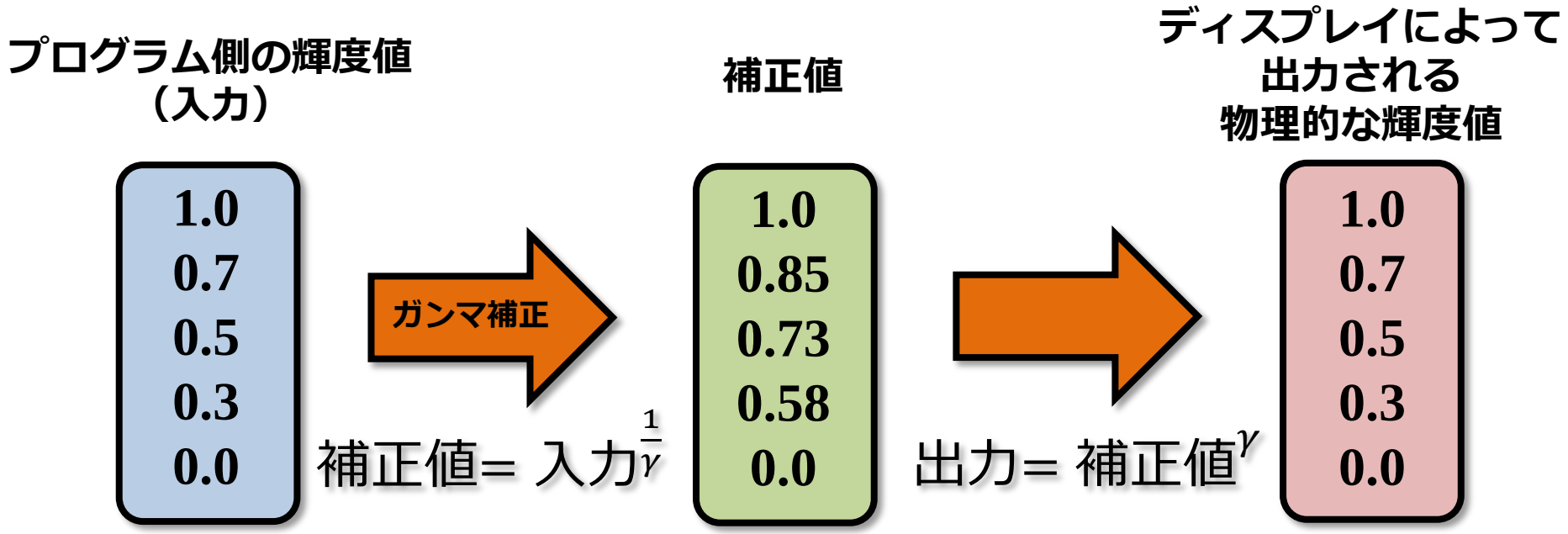
ガンマ

- **γの値**
  - 基本的にディスプレイ、OSに依存
  - sRGB、AdobeRGBのような広く業務にも使われてるカラースペースでは**γ=2.2**が採用されている。

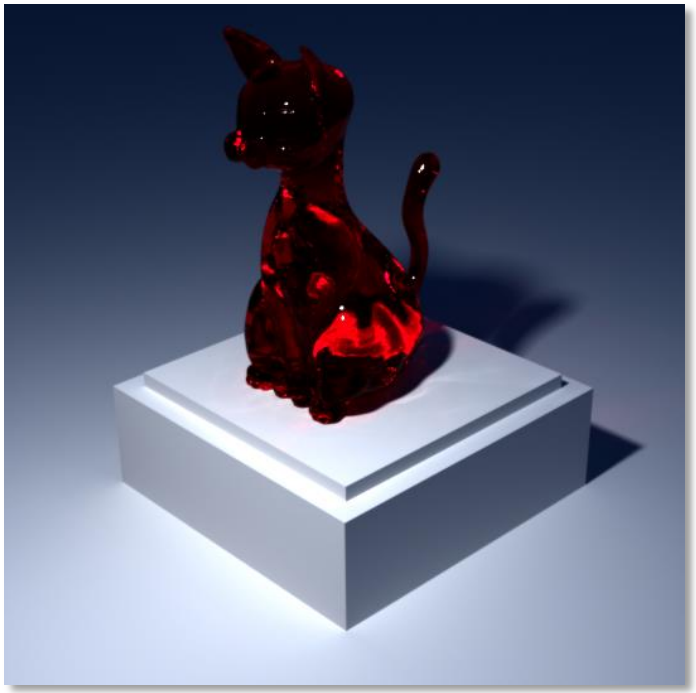


ガンマ補正

- **物理ベースレンダリングの結果はリニアスペースなので ディスプレイの出力輝度値を画像の値に比例させたい**
  - ガンマ補正
  - γ値の逆数のべき乗をディスプレイに対する入力にする。
  - euptではγ=2.2と仮定して画像の保存の際値を $\frac{1}{2.2}$乗している。



ガンマ補正



ガンマ補正前

ガンマ補正後

γ=2.2と仮定

2.3 乱数

random.h

```
#ifndef _RANDOM_H_
#define _RANDOM_H_

#include <climits>

namespace edupt {

// Xor-Shiftによる乱数ジェネレータ
class XorShift {
    unsigned int seed_[4];
public:
    unsigned int next(void) {
        const unsigned int t = seed_[0] ^ (seed_[0] << 11);
        seed_[0] = seed_[1];
        seed_[1] = seed_[2];
        seed_[2] = seed_[3];
        return seed_[3] = (seed_[3] ^ (seed_[3] >> 19)) ^ (t ^ (t >> 8));
    }

    double next01(void) {
        return (double)next() / UINT_MAX;
    }

    XorShift(const unsigned int initial_seed) {
        unsigned int s = initial_seed;
        for (int i = 1; i <= 4; i++){
            seed_[i-1] = s = 1812433253U * (s^(s>>30)) + i;
        }
    }
};

typedef XorShift Random;

};

#endif
```

乱数

- **パストレーシングは確率的なアルゴリズム**
  - さまざまな局面で乱数が必要になる。
  - 質の高い乱数が求められる。
- **擬似乱数アルゴリズム**
  - 線形合同法
  - メルセンヌツイスター
  - **Xorshift**
  - Random123
  - etc

eduptでは実装の単純さと速度、質などのバランスからXorshiftを乱数生成機として使用している

パストレーシングがモンテカルロ積分をその基礎においているため、サンプルを得るために乱数が必要になる。準モンテカルロ法の様な、より進んだアルゴリズムを使うなら、乱数の変わりに低食い違い量列というものを使うこともある。

Xorshift

- **eduptでの実装**
  - WikipediaのXorshiftの項目
    - http://ja.wikipedia.org/wiki/Xorshift
  - Xorshiftのseed初期化
    - http://meme.biology.tohoku.ac.jp/klabo-wiki/index.php?%B7%D7%BB%BB%B5%A1%2FC%2B%2B
  - を参考にしました。
  - next01()によって[0.0, 1.0]の範囲で乱数を得る。
- **Random型にtypedefして他の場所で使用**

3.eduptコード解説（幾何編）

3.edupt コード解説（幾何編）

- **まえおき**

  edupt における、幾何学関係のデータ構造についての解説。
  ベクトル構造体や、基本的な形状である球とレイの交差判定など。

edupt ファイル概要

- **main.cpp**
  edupt::render()を呼び出すだけ。
- **constant.h**
  各種定数。
- **intersection.h**
  交差判定の結果を格納する構造体IntersectionとHitpoint。
- **material.h**
  Color型と物体材質について記述するReflectionTypeと屈折率の設定。
- **ppm.h**
  レンダリング結果をppm画像として書き出すための関数save_ppm_file()。
- **radiance.h**
  ある方向からの放射輝度を得る関数radiance()。パストレーシングのメイン処理。
- **random.h**
  乱数生成クラスXorShift。
- **ray.h**
  一つ一つの光線、レイを表現する構造体Ray。
- **render.h**
  レンダリング画像サイズやサンプル数を受け取り、radiance()を使って各ピクセルの具体的な値を決定する関数render()。
- **scene.h**
  レンダリングするシーンのデータspheres[]とそのシーンに対する交差判定を行う関数intersect_scene()。
- **sphere.h**
  基本的な形状としての球を表現するSphere構造体。
- **vec.h**
  ベクトルを表現する構造体Vec。

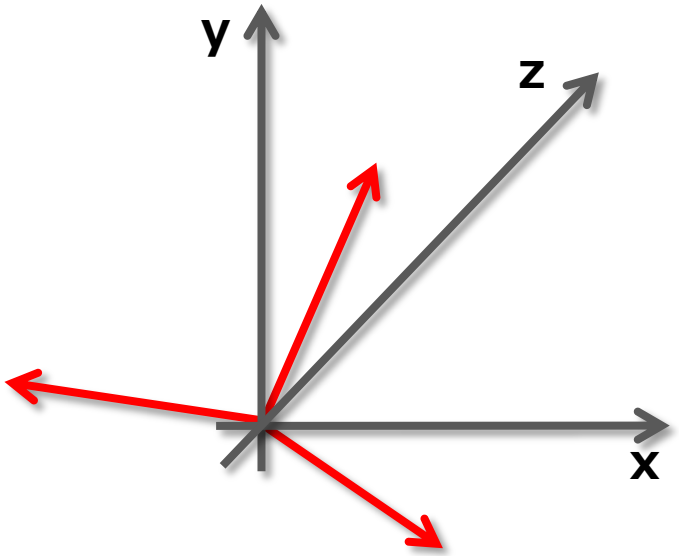
3.1 ベクトル

vec.h

```
#ifndef _VEC_H_
#define _VEC_H_

#include <cmath>

namespace edupt {

struct Vec {
    double x, y, z;
         Vec(const double x = 0, const double y = 0, const double z = 0) : x(x), y(y), z(z) {}
    inline Vec operator+(const Vec &b) const {
        return Vec(x + b.x, y + b.y, z + b.z);
    }
    inline Vec operator-(const Vec &b) const {
        return Vec(x - b.x, y - b.y, z - b.z);
    }
    inline Vec operator*(const double b) const {
        return Vec(x * b, y * b, z * b);
    }
    inline Vec operator/(const double b) const {
        return Vec(x / b, y / b, z / b);
    }
    inline const double length_squared() const {
        return x*x + y*y + z*z;
    }
    inline const double length() const {
        return sqrt(length_squared());
    }
};
inline Vec operator*(double f, const Vec &v) {
    return v * f;
}
inline Vec normalize(const Vec &v) {
    return v * (1.0 / v.length());
}
inline const Vec multiply(const Vec &v1, const Vec &v2) {
    return Vec(v1.x * v2.x, v1.y * v2.y, v1.z * v2.z);
}
inline const double dot(const Vec &v1, const Vec &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
inline const Vec cross(const Vec &v1, const Vec &v2) {
    return Vec(
        (v1.y * v2.z) - (v1.z * v2.y),
        (v1.z * v2.x) - (v1.x * v2.z),
        (v1.x * v2.y) - (v1.y * v2.x));
}

};

#endif
```

Vec構造体

- **三組の実数による構造体**
  - 線形代数におけるベクトルとほぼ同じ。
  - eduptでは点、ベクトル、色等を表現するために使われる。



三次元空間におけるベクトル

Vec構造体

- ベクトル同士の加算
- ベクトル同士の減算
- ベクトルとスカラ量の乗算
- ベクトルとスカラ量の除算
- 二乗長さ、長さを求める関数length_squared()とlength()
  - 各要素の二乗の和の平方根がベクトルの長さ（三平方の定理）

```
inline Vec operator+(const Vec &b) const {
    return Vec(x + b.x, y + b.y, z + b.z);
}
inline Vec operator-(const Vec &b) const {
    return Vec(x - b.x, y - b.y, z - b.z);
}
inline Vec operator*(const double b) const {
    return Vec(x * b, y * b, z * b);
}
inline Vec operator/(const double b) const {
    return Vec(x / b, y / b, z / b);
}
inline const double length_squared() const {
    return x*x + y*y + z*z;
}
inline const double length() const {
    return sqrt(length_squared());
}
```

normalize()

- **ベクトルの正規化関数**
  - ベクトルの各要素を自身の長さで割る操作。
  - 正規化後のベクトルは長さが１になる。

$$\frac{\vec{v}}{\|\vec{v}\|} = \frac{\vec{v}}{\sqrt{v_x^2 + v_y^2 + v_z^2}}$$

```
inline Vec normalize(const Vec &v) {
    return v * (1.0 / v.length());
}
```

multiply()

- 二つのベクトルの要素ごとの積

```
inline const Vec multiply(const Vec &v1, const Vec &v2) {
    return Vec(v1.x * v2.x, v1.y * v2.y, v1.z * v2.z);
}
```

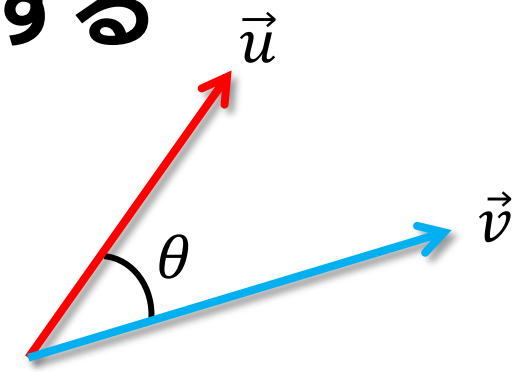
dot()

- 二つのベクトルの内積を計算する

内積

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \|\vec{v}\| \cos\theta$$

$$\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y + u_z v_z$$



特に長さ１のベクトル同士の内積は二つのベクトルの成す角のcosとなる。
二つのベクトルの成す角を得たいときは内積を取るのが定石。

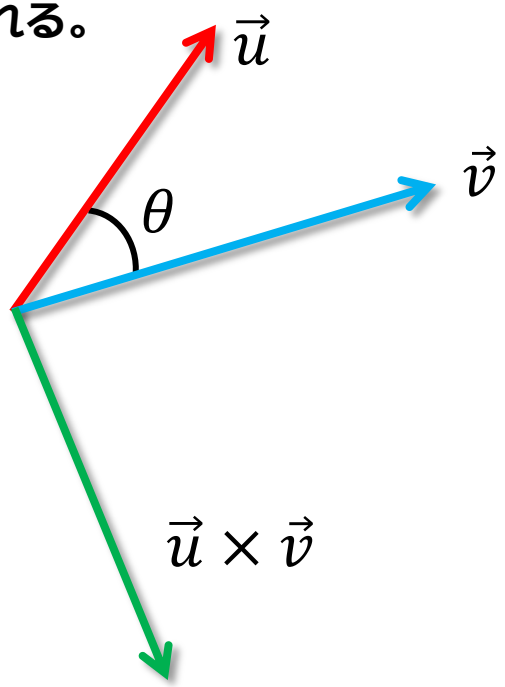
```
inline const double dot(const Vec &v1, const Vec &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

cross()

●二つのベクトルの外積を計算する

外積　$\vec{u} \times \vec{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$

外積によって、二つのベクトルのどちらとも直交するベクトルが得られる。
その向きは右ネジの進む方向になる。

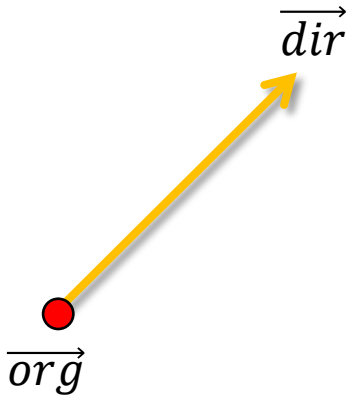


外積の長さ　$\|\vec{u} \times \vec{v}\| = \|\vec{u}\|\|\vec{v}\| \sin\theta$

外積の長さは二つのベクトルの長さと成す角のsinとの積になる。
また、二つのベクトルによって作られる平行四辺形の面積に等しい。

```
inline const Vec cross(const Vec &v1, const Vec &v2) {
    return Vec(
        (v1.y * v2.z) - (v1.z * v2.y),
        (v1.z * v2.x) - (v1.x * v2.z),
        (v1.x * v2.y) - (v1.y * v2.x));
}
```

ray.h



- **レイ（光線）を表現する構造体**
  - 始点orgと向きdirを持つ。
  - 始点は位置ベクトルとしてVecで表現する。
  - eduptでは基本的にdirの長さは1になるようにしている。
    - 何か所かこの仮定を用いている処理がある。
    - もちろんこの仮定をなくしてもよい。（ただしこの仮定を用いている処理は修正しないといけない）

```
#ifndef _RAY_H_
#define _RAY_H_

#include "vec.h"

namespace edupt {

struct Ray {
    Vec org, dir;
    Ray(const Vec &org, const Vec &dir) : org(org), dir(dir) {}
};

};

#endif
```

3.2 球

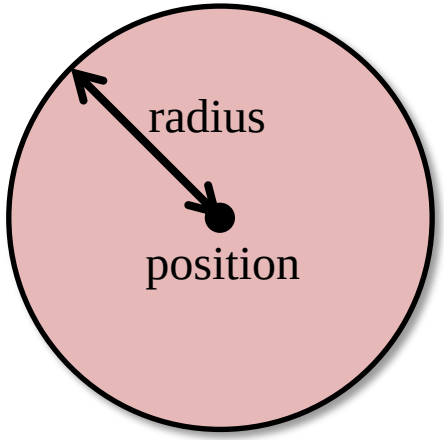
sphere.h

```
#ifndef _SPHERE_H_
#define _SPHERE_H_

#include <cmath>

#include "vec.h"
#include "ray.h"
#include "material.h"
#include "constant.h"
#include "intersection.h"

namespace edupt {

struct Sphere {
    double radius;
    Vec position;
    Color emission;
    Color color;
    ReflectionType reflection_type;

    Sphere(const double radius, const Vec &position, const Color &emission, const Color &color, const ReflectionType reflection_type) :
      radius(radius), position(position), emission(emission), color(color), reflection_type(reflection_type) {}

    // 入力のrayに対する交差点までの距離を返す。交差しなかったら0を返す。
    // rayとの交差判定を行う。交差したらtrue,さもなくばfalseを返す。
    bool intersect(const Ray &ray, Hitpoint *hitpoint) const {
        const Vec p_o = position - ray.org;
        const double b = dot(p_o, ray.dir);
        const double D4 = b * b - dot(p_o, p_o) + radius * radius;

        if (D4 < 0.0)
            return false;

        const double sqrt_D4 = sqrt(D4);
        const double t1 = b - sqrt_D4, t2 = b + sqrt_D4;

        if (t1 < kEPS && t2 < kEPS)
            return false;

        if (t1 > kEPS) {
            hitpoint->distance = t1;
        } else {
            hitpoint->distance = t2;
        }

        hitpoint->position = ray.org + hitpoint->distance * ray.dir;
        hitpoint->normal   = normalize(hitpoint->position - position);
        return true;
    }
};

};

#endif
```

Sphere構造体

- **eduptにおける唯一の幾何形状**
  - 中心座標 positionと半径 radius
  - 発光色 emission
  - 反射率 color
  - 材質（表面における反射の種類）reflection_type

```
double radius;
Vec position;
Color emission;
Color color;
ReflectionType reflection_type;
```



material.h

- **Vec構造体を使ってColorを定義**
- **材質ReflectionTypeと屈折率**
  - 完全拡散面、完全鏡面、ガラス面の三種類
  - ガラスに対する屈折率がkIor
  - 詳しくはレンダリング編

```
#ifndef _MATERIAL_H_
#define _MATERIAL_H_

#include "vec.h"

namespace edupt {

typedef Vec Color;

enum ReflectionType {
    REFLECTION_TYPE_DIFFUSE,    // 完全拡散面。いわゆるLambertian面。
    REFLECTION_TYPE_SPECULAR,   // 理想的な鏡面。
    REFLECTION_TYPE_REFRACTION, // 理想的なガラス的物質。
};

const double kIor = 1.5; // 屈折率(Index of refraction)

};

#endif
```

球との交差判定

- **球の方程式**
  - $\vec{p}$は球の中心でrは球の半径。
  - 以下を満たすような$\vec{x}$は球の表面上の点。

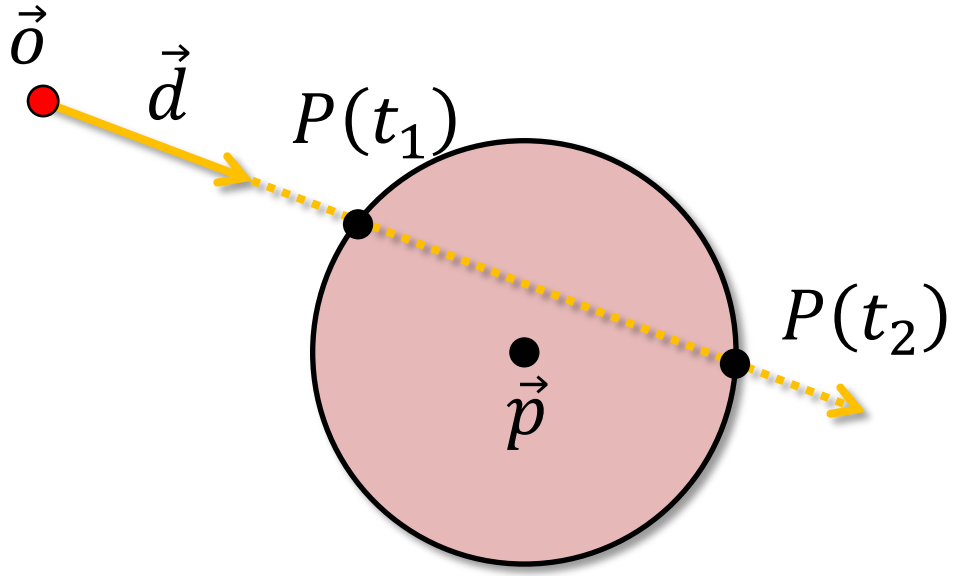
$$\|\vec{p} - \vec{x}\| = r$$

- **あるレイとこの球との交差点を計算したいとする**
  - $\vec{o}$はレイの始点で$\vec{d}$はレイの方向。
  - $P(t)$は始点から$\vec{d}$の方向に距離t進んだ位置にある点。

$$P(t) = \vec{o} + t\vec{d}$$

球との交差判定

- **球の方程式にレイ上の点の式を代入する**
  - 球とレイが交差する、ということはレイ上の点が球の方程式を満たし、球の表面上に存在するということ。



$$\|\vec{p} - P(t)\| = r$$

$$\left(\vec{p} - P(t)\right)^2 = r^2$$

$$\left(\vec{p} - \vec{o} - t\vec{d}\right)^2 = r^2$$

球との交差判定

- **tについての二次方程式になる**
  - この式の解がレイと球の交差点に対応する。

$$(\vec{d}\cdot\vec{d})t^2 - 2\vec{d}\cdot(\vec{p}-\vec{o})t + (\vec{p}-\vec{o})\cdot(\vec{p}-\vec{o}) - r^2 = 0$$

$$A = \vec{d}\cdot\vec{d}$$

$$B = -2\vec{d}\cdot(\vec{p}-\vec{o})$$

$$C = (\vec{p}-\vec{o})\cdot(\vec{p}-\vec{o}) - r^2$$

intersect()

- **二次方程式の判別式を用いて解の存在を判定**
  - さっきの式から判別式を作れる。

判別式 $$\frac{D}{4} = \frac{B^2 - 4AC}{4} = \left(\vec{d} \cdot (\vec{p} - \vec{o})\right)^2 - \left(\vec{d} \cdot \vec{d}\right)\left((\vec{p} - \vec{o}) \cdot (\vec{p} - \vec{o}) - r^2\right)$$

ただし、Ray構造体の向きの長さは1にするという約束があったので $\vec{d} \cdot \vec{d} = 1$ **になる。**

判別式が負で、解がない場合は交差しないことを意味する

```
bool intersect(const Ray &ray, Hitpoint *hitpoint) const {
    const Vec p_o = position - ray.org;
    const double b = dot(p_o, ray.dir);
    const double D4 = b * b - dot(p_o, p_o) + radius * radius;

    if (D4 < 0.0)
        return false;

    const double sqrt_D4 = sqrt(D4);
    const double t1 = b - sqrt_D4, t2 = b + sqrt_D4;

    if (t1 < kEPS && t2 < kEPS)
        return false;

    if (t1 > kEPS) {
        hitpoint->distance = t1;
    } else {
        hitpoint->distance = t2;
    }

    hitpoint->position = ray.org + hitpoint->distance * ray.dir;
    hitpoint->normal   = normalize(hitpoint->position - position);
    return true;
}
```

intersect()

- **判別式を使って解を計算する**
  - 二次方程式の解の公式より

$$t = \frac{-B \pm \sqrt{D}}{2A} = \frac{-\frac{B}{2} \pm \sqrt{\frac{D}{4}}}{A} = \frac{b \pm \sqrt{\frac{D}{4}}}{A} = b \pm \sqrt{\frac{D}{4}}$$

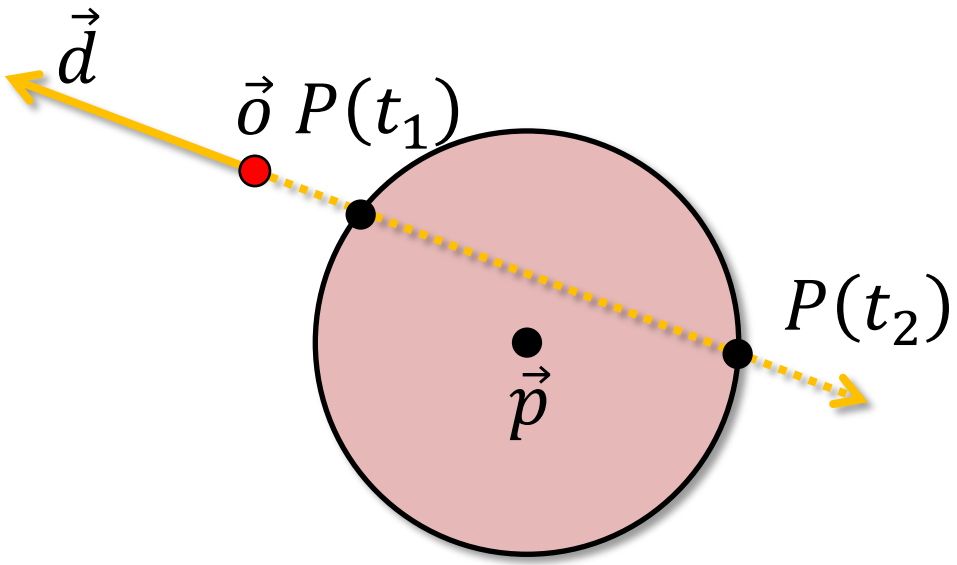
$A = \vec{d} \cdot \vec{d} = 1$**という約束**

$$\frac{B}{2} = -\vec{d} \cdot (\vec{p} - \vec{o}) = -b$$

```
bool intersect(const Ray &ray, Hitpoint *hitpoint) const {
    const Vec p_o = position - ray.org;
    const double b = dot(p_o, ray.dir);
    const double D4 = b * b - dot(p_o, p_o) + radius * radius;

    if (D4 < 0.0)
        return false;

    const double sqrt_D4 = sqrt(D4);
    const double t1 = b - sqrt_D4, t2 = b + sqrt_D4;

    if (t1 < kEPS && t2 < kEPS)
        return false;

    if (t1 > kEPS) {
        hitpoint->distance = t1;
    } else {
        hitpoint->distance = t2;
    }

    hitpoint->position = ray.org + hitpoint->distance * ray.dir;
    hitpoint->normal   = normalize(hitpoint->position - position);
    return true;
}
```

intersect()

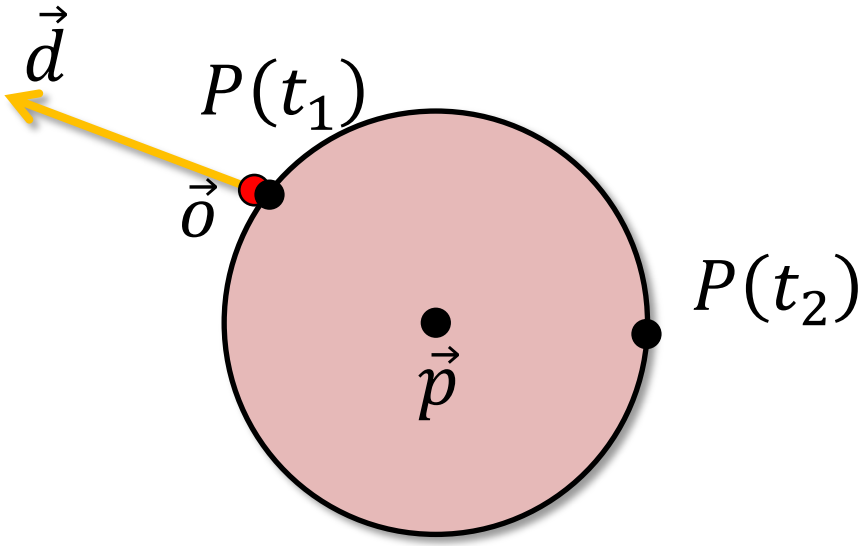
- 得られた解について、0.0以下ならレイの向いている側で交差していない



解が負のケース

intersect()

- **得られた解について、kEPS未満なら**
  - レイの始点が、相手の球の表面と非常に近い場所に存在している。
  - この場合は球の表面から離れていく方向にレイが発射されていると考え、この交差を無視する。
  - **レイトレではレイと物体の交差点を始点にして次のレイを追跡する、ということを行うためこのようなことが起きる。**



解が0.0に近いケース

intersect()

- **得られた二つの解について**
  - 両方ともkEPS未満なら交差点は一つもない。

```
bool intersect(const Ray &ray, Hitpoint *hitpoint) const {
    const Vec p_o = position - ray.org;
    const double b = dot(p_o, ray.dir);
    const double D4 = b * b - dot(p_o, p_o) + radius * radius;

    if (D4 < 0.0)
        return false;

    const double sqrt_D4 = sqrt(D4);
    const double t1 = b - sqrt_D4, t2 = b + sqrt_D4;

    if (t1 < kEPS && t2 < kEPS)
        return false;

    if (t1 > kEPS) {
        hitpoint->distance = t1;
    } else {
        hitpoint->distance = t2;
    }

    hitpoint->position = ray.org + hitpoint->distance * ray.dir;
    hitpoint->normal   = normalize(hitpoint->position - position);
    return true;
}
```

intersect()

- 得られた二つの解について
  - t_1がkEPSより大きいとき
    - $t_1 < t_2$なのでt_2もkEPSより大。
    - レイと球の交差点は幾何的には二点ある。
    - 物理的には始点から発射されたレイは最初にぶつかった点で反射するので、始点に近いほうの交差点、すなわちt_1が交差点になる。
  - t_1がkEPSより小さいとき
    - 1つ前のif文を通過しているのでt_2はkEPSより大きい。
    - よってt_2が交差点。

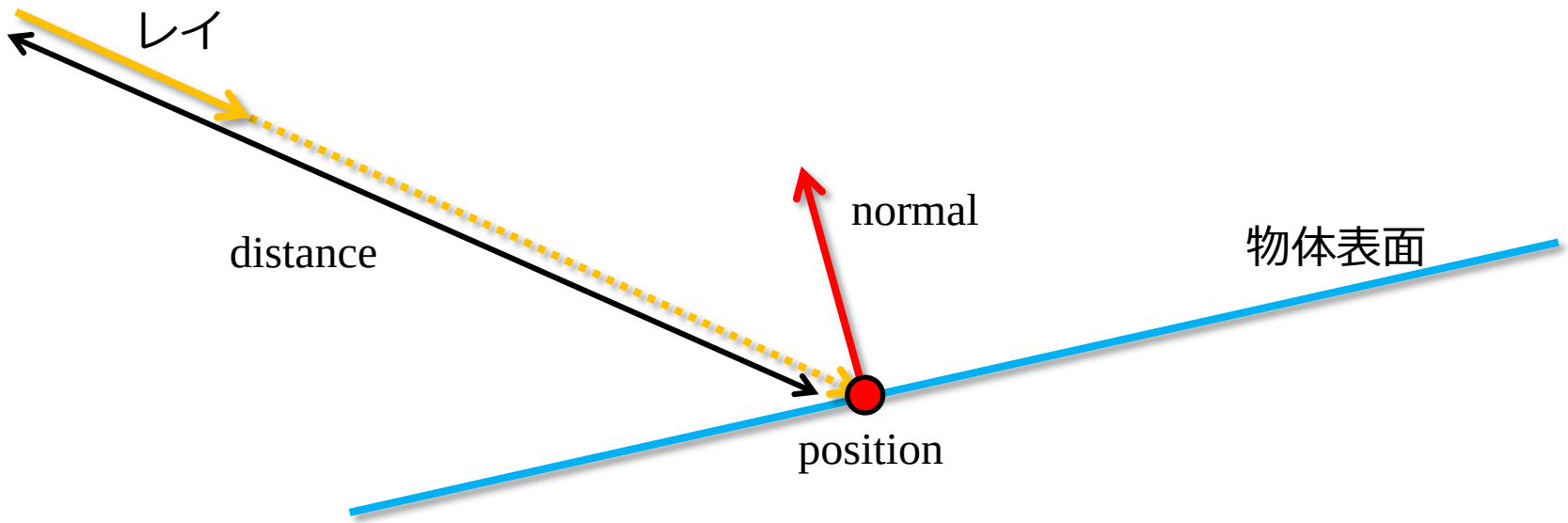
```
bool intersect(const Ray &ray, Hitpoint *hitpoint) const {
    const Vec p_o = position - ray.org;
    const double b = dot(p_o, ray.dir);
    const double D4 = b * b - dot(p_o, p_o) + radius * radius;

    if (D4 < 0.0)
        return false;

    const double sqrt_D4 = sqrt(D4);
    const double t1 = b - sqrt_D4, t2 = b + sqrt_D4;

    if (t1 < kEPS && t2 < kEPS)
        return false;

    if (t1 > kEPS) {
        hitpoint->distance = t1;
    } else {
        hitpoint->distance = t2;
    }

    hitpoint->position = ray.org + hitpoint->distance * ray.dir;
    hitpoint->normal   = normalize(hitpoint->position - position);
    return true;
}
```

intersection.h

```
#ifndef _INTERSECTION_H_
#define _INTERSECTION_H_

#include "vec.h"
#include "constant.h"

namespace edupt {

struct Hitpoint {
    double distance;
    Vec normal;
    Vec position;

    Hitpoint() : distance(kINF), normal(), position() {}
};

struct Intersection {
    Hitpoint hitpoint;
    int object_id;

    Intersection() : object_id(-1) {}
};

};

#endif
```

Hitpoint構造体

- **レイと物体の交差点の幾何学的情報を格納する**
  - レイの始点から交差点までの距離 distance
  - 交差点における物体法線 normal
  - 交差点のワールド座標系での位置 position



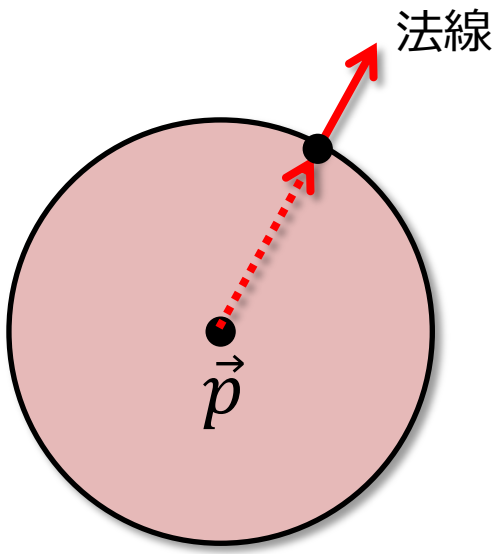
Intersection構造体

- 幾何学的情報（Hitpoint構造体）に加え、物体のIDも格納する

```
struct Intersection {
    Hitpoint hitpoint;
    int object_id;

    Intersection() : object_id(-1) {}
};
```

intersect()

- **交差点までの距離や交差点の位置、交差点における法線をHitpoint構造体に格納する**
  - 球の表面において、交差点における法線は球の中心から交差点へのベクトルを正規化したものになる。




```
bool intersect(const Ray &ray, Hitpoint *hitpoint) const {
    const Vec p_o = position - ray.org;
    const double b = dot(p_o, ray.dir);
    const double D4 = b * b - dot(p_o, p_o) + radius * radius;

    if (D4 < 0.0)
        return false;

    const double sqrt_D4 = sqrt(D4);
    const double t1 = b - sqrt_D4, t2 = b + sqrt_D4;

    if (t1 < kEPS && t2 < kEPS)
        return false;

    if (t1 > kEPS) {
        hitpoint->distance = t1;
    } else {
        hitpoint->distance = t2;
    }

    hitpoint->position = ray.org + hitpoint->distance * ray.dir;
    hitpoint->normal   = normalize(hitpoint->position - position);
    return true;
}
```

3.3 シーンデータ

scene.h

```
#ifndef _SCENE_H_
#define _SCENE_H_

#include "constant.h"
#include "sphere.h"
#include "intersection.h"

namespace edupt {

// レンダリングするシーンデータ
const Sphere spheres[] = {
    Sphere(1e5, Vec( 1e5+1, 40.8, 81.6), Color(),      Color(0.75, 0.25, 0.25), REFLECTION_TYPE_DIFFUSE), // 左
    Sphere(1e5, Vec(-1e5+99, 40.8, 81.6),Color(),      Color(0.25, 0.25, 0.75), REFLECTION_TYPE_DIFFUSE), // 右
    Sphere(1e5, Vec(50, 40.8, 1e5),      Color(),      Color(0.75, 0.75, 0.75), REFLECTION_TYPE_DIFFUSE), // 奥
    Sphere(1e5, Vec(50, 40.8, -1e5+250), Color(),      Color(),                 REFLECTION_TYPE_DIFFUSE), // 手前
    Sphere(1e5, Vec(50, 1e5, 81.6),      Color(),      Color(0.75, 0.75, 0.75), REFLECTION_TYPE_DIFFUSE), // 床
    Sphere(1e5, Vec(50, -1e5+81.6, 81.6),Color(),      Color(0.75, 0.75, 0.75), REFLECTION_TYPE_DIFFUSE), // 天井
    Sphere(20,Vec(65, 20, 20),           Color(),      Color(0.25, 0.75, 0.25), REFLECTION_TYPE_DIFFUSE), // 緑球
    Sphere(16.5,Vec(27, 16.5, 47),       Color(),      Color(0.99, 0.99, 0.99), REFLECTION_TYPE_SPECULAR), // 鏡
    Sphere(16.5,Vec(77, 16.5, 78),       Color(),      Color(0.99, 0.99, 0.99), REFLECTION_TYPE_REFRACTION), //ガラス
    Sphere(15.0,Vec(50.0, 90.0, 81.6),   Color(36,36,36), Color(),              REFLECTION_TYPE_DIFFUSE), //照明
};

// シーンとの交差判定関数
inline bool intersect_scene(const Ray &ray, Intersection *intersection) {
    const double n = sizeof(spheres) / sizeof(Sphere);

    // 初期化
    intersection->hitpoint.distance = kINF;
    intersection->object_id  = -1;

    // 線形探索
    for (int i = 0; i < int(n); i ++) {
        Hitpoint hitpoint;
        if (spheres[i].intersect(ray, &hitpoint)) {
            if (hitpoint.distance < intersection->hitpoint.distance) {
                intersection->hitpoint = hitpoint;
                intersection->object_id = i;
            }
        }
    }

    return (intersection->object_id != -1);
}

};

#endif
```

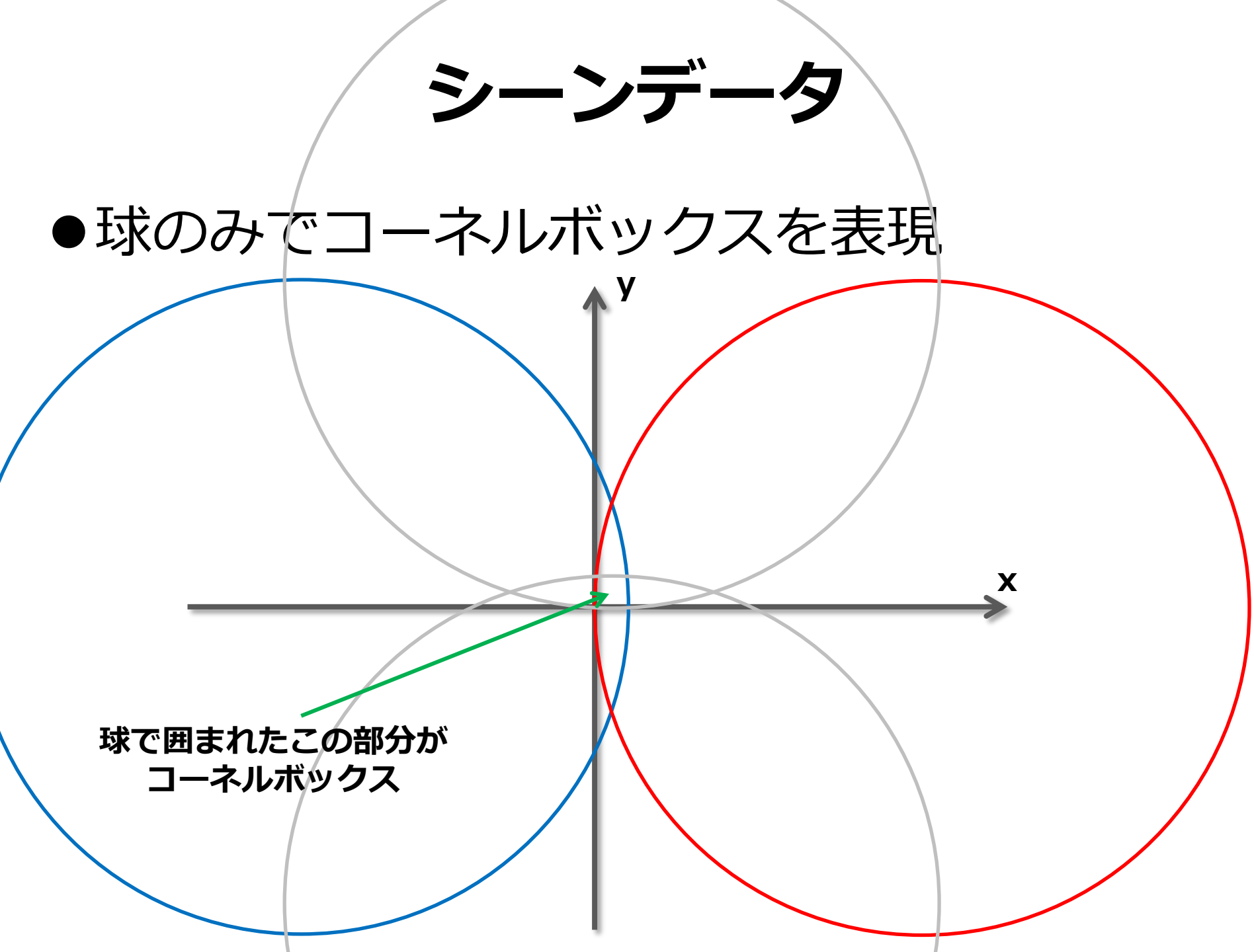
シーンデータ

- **レンダリングするシーンのデータ**
  - Sphere構造体の配列
  - ヘッダに直接記述

```
// レンダリングするシーンデータ
const Sphere spheres[] = {
    Sphere(1e5, Vec( 1e5+1, 40.8, 81.6), Color(),      Color(0.75, 0.25, 0.25), REFLECTION_TYPE_DIFFUSE), // 左
    Sphere(1e5, Vec(-1e5+99, 40.8, 81.6),Color(),      Color(0.25, 0.25, 0.75), REFLECTION_TYPE_DIFFUSE), // 右
    Sphere(1e5, Vec(50, 40.8, 1e5),      Color(),      Color(0.75, 0.75, 0.75), REFLECTION_TYPE_DIFFUSE), // 奥
    Sphere(1e5, Vec(50, 40.8, -1e5+250), Color(),      Color(),                 REFLECTION_TYPE_DIFFUSE), // 手前
    Sphere(1e5, Vec(50, 1e5, 81.6),      Color(),      Color(0.75, 0.75, 0.75), REFLECTION_TYPE_DIFFUSE), // 床
    Sphere(1e5, Vec(50, -1e5+81.6, 81.6),Color(),      Color(0.75, 0.75, 0.75), REFLECTION_TYPE_DIFFUSE), // 天井
    Sphere(20,Vec(65, 20, 20),           Color(),      Color(0.25, 0.75, 0.25), REFLECTION_TYPE_DIFFUSE), // 緑球
    Sphere(16.5,Vec(27, 16.5, 47),       Color(),      Color(0.99, 0.99, 0.99), REFLECTION_TYPE_SPECULAR), // 鏡
    Sphere(16.5,Vec(77, 16.5, 78),       Color(),      Color(0.99, 0.99, 0.99), REFLECTION_TYPE_REFRACTION), //ガラス
    Sphere(15.0,Vec(50.0, 90.0, 81.6),   Color(36,36,36), Color(),              REFLECTION_TYPE_DIFFUSE), //照明
};
```

シーンデータ

- 球のみでコーネルボックスを表現



intersect_scene()

- **シーンに存在する各物体とレイとの交差判定関数**
  - 各交差点の内、一番レイの始点に近いものを求める。（一番近い点で反射するから）
  - 交差判定はすべての物体に対して行う。（全探索）
    - 計算量はO(N) (Nは物体の数)

レンダリングにおいて最も重い部分の一つはレイトレによる交差判定、すなわちこの関数である。
eduptのシーンデータはかなり単純なので全探索しても問題になりにくいが、交差判定をする対象が何百万個ものポリゴンになったりすると線形の全探索ではもちろんどうにもならない。
こういった場合はkd-tree、BVH、Octreeなどの空間分割木を使って計算量をO(logN)にするのが定石。

```
// シーンとの交差判定関数
inline bool intersect_scene(const Ray &ray, Intersection *intersection) {
    const double n = sizeof(spheres) / sizeof(Sphere);

    // 初期化
    intersection->hitpoint.distance = kINF;
    intersection->object_id  = -1;

    // 線形探索
    for (int i = 0; i < int(n); i ++) {
        Hitpoint hitpoint;
        if (spheres[i].intersect(ray, &hitpoint)) {
            if (hitpoint.distance < intersection->hitpoint.distance) {
                intersection->hitpoint = hitpoint;
                intersection->object_id = i;
            }
        }
    }

    return (intersection->object_id != -1);
}
```

4.パストレーシング

- **まえおき**
  - eduptのメインのレンダリング部分の説明の前に光の物理量、レンダリング方程式、そしてレンダリング方程式を解くアルゴリズムの一つパストレーシングについてより一般的な説明をします。

パストレーシング

- **レンダリング方程式に基づいて光の挙動をシミュレーションするアルゴリズムの一つ**
  - レンダリング方程式は光の伝達のモデル。
  - パストレーシングを使うと写実的なCGを作れる。
    - グローバルイルミネーション（光の相互反射）
- **光伝達の物理モデルであるレンダリング方程式**
  - 光の物理量について知らなければならない。

4.1 光の物理量

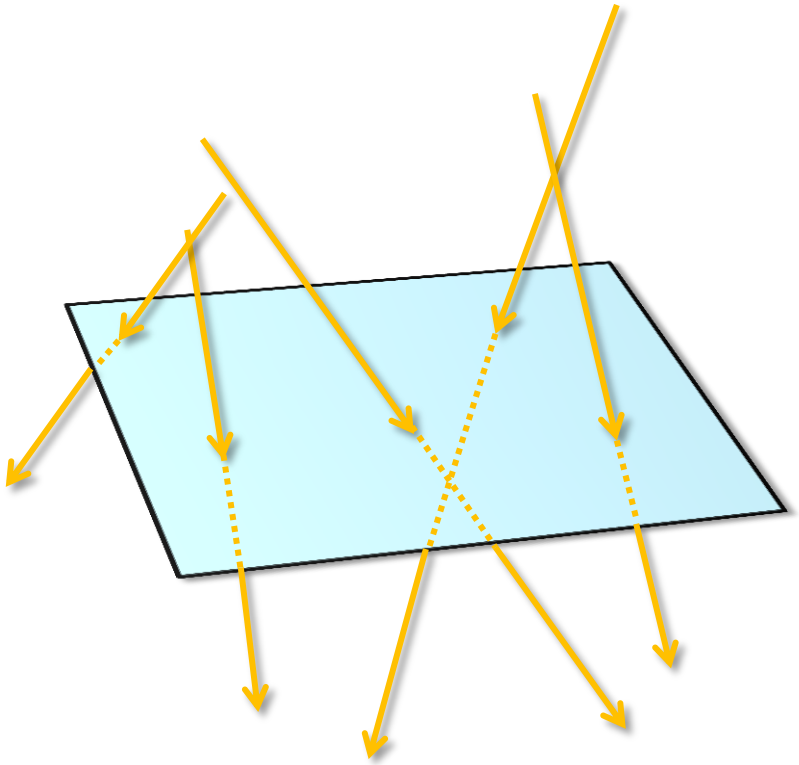
光の物理量

- **物理モデルに従った光の挙動を計算**
  - 光の物理量を把握する必要がある。
- **放射分析学（Radiometry）**
  - 測光学（Photometry）

放射束(Flux)

- 単位時間あたり、ある領域を通過するフォトン数（エネルギー）

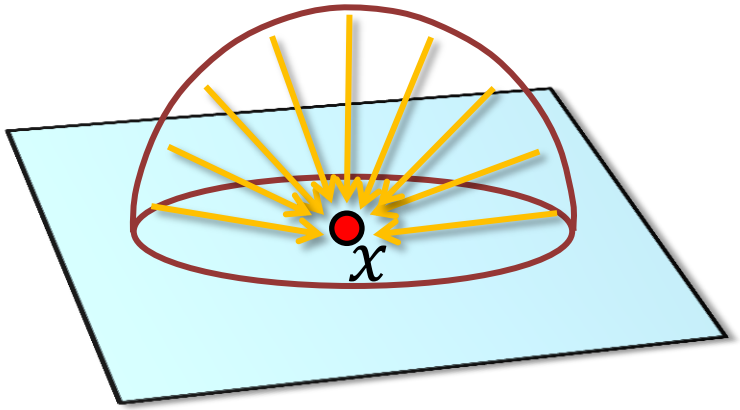
$$\Phi\left[\frac{\mathrm{J}}{\mathrm{s}}\right] = \Phi[\mathrm{W}]$$



放射照度(Irradiance)

- 放射束の面積密度（単位面積あたりの放射束）

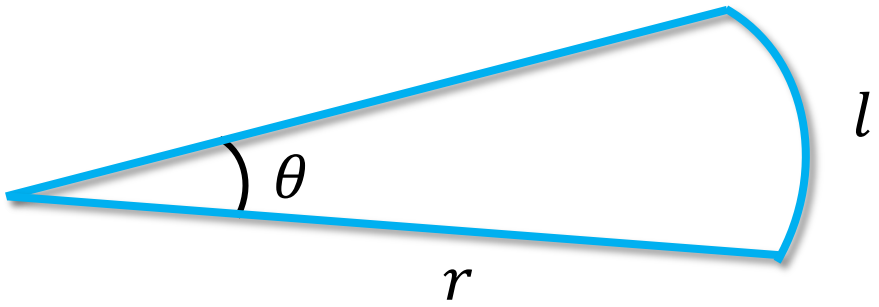
$$\mathrm{B}(x)\left[\frac{\mathrm{W}}{\mathrm{m}^2}\right] = \frac{d\Phi}{dA}$$



平面角

- **平面角（平面における角）**
  - 二つの半直線の間の領域。
  - 弧の長さと半径の比が角度(radian)。
  - 円は2π(rad)の平面角を持つ。

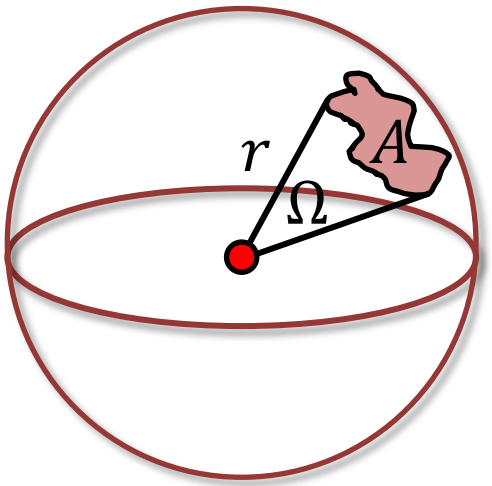
$$\theta = \frac{l}{r}$$



立体角

- **立体角（空間における角）**
  - 球上の領域。
  - 球上の領域の面積と半径の二乗比が立体角(steradians)。
  - 球は4π(sr)の立体角を持つ。

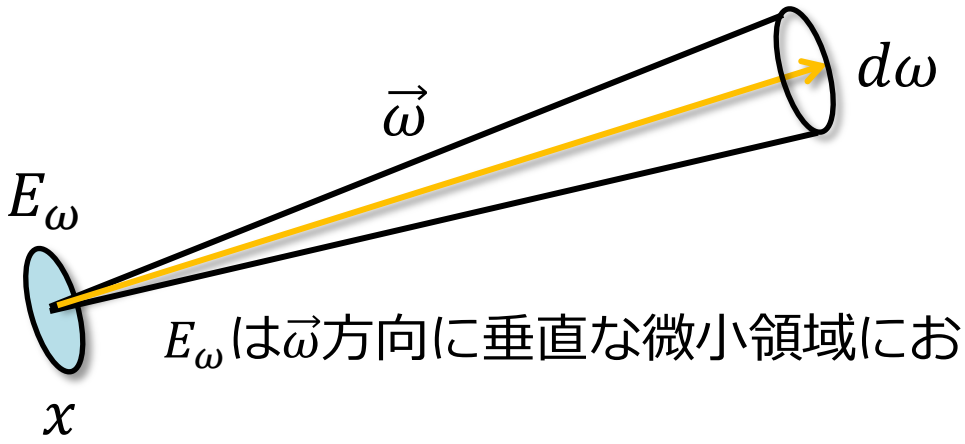
$$\Omega = \frac{A}{r^2}$$



放射輝度(Radiance)

- 放射輝度は放射照度の立体角密度（単位立体角あたりの放射照度）

$$L(x, \vec{\omega}) = \frac{dE_\omega(x)}{d\omega} \left[\frac{\mathrm{W}}{\mathrm{m}^2 \cdot \mathrm{sr}}\right]$$



ある点においてある方向に通過するフォトンの量が放射輝度と考えることができる。パストレーシングにおいて一つ一つのレイが運ぶ物理量は放射輝度

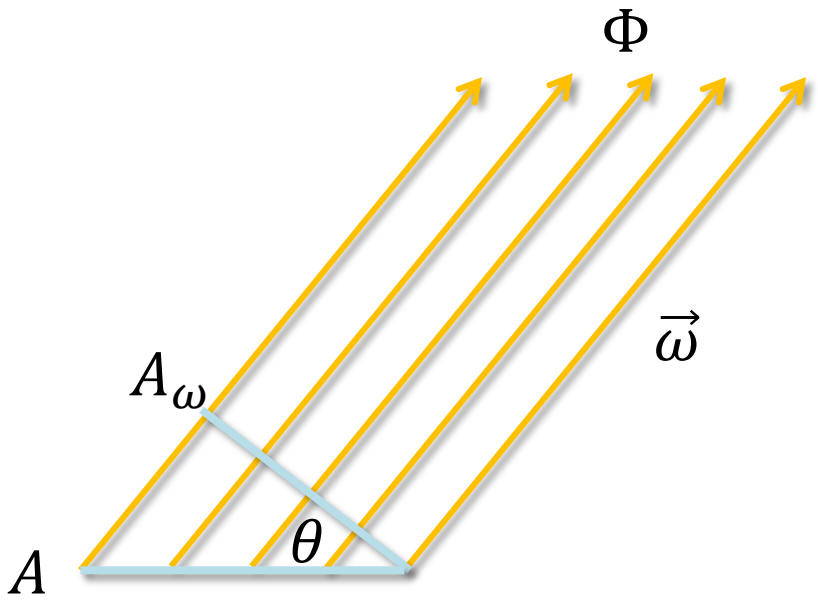
放射輝度(Radiance)

- Lambertのコサイン則より$dE_\omega \cos\theta = dE$

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

Lambertのコサイン則

- 領域AをΦ[W]の光が通過しているとすると、領域Aの放射照度は$E = \frac{\Phi}{A}$で、領域$A_\omega = A\cos\theta$の放射照度は$E_\omega = \frac{\Phi}{A_\omega} = \frac{\Phi}{A\cos\theta}$
- よって、一般に$E_\omega \cos\theta = E$が成り立つ。



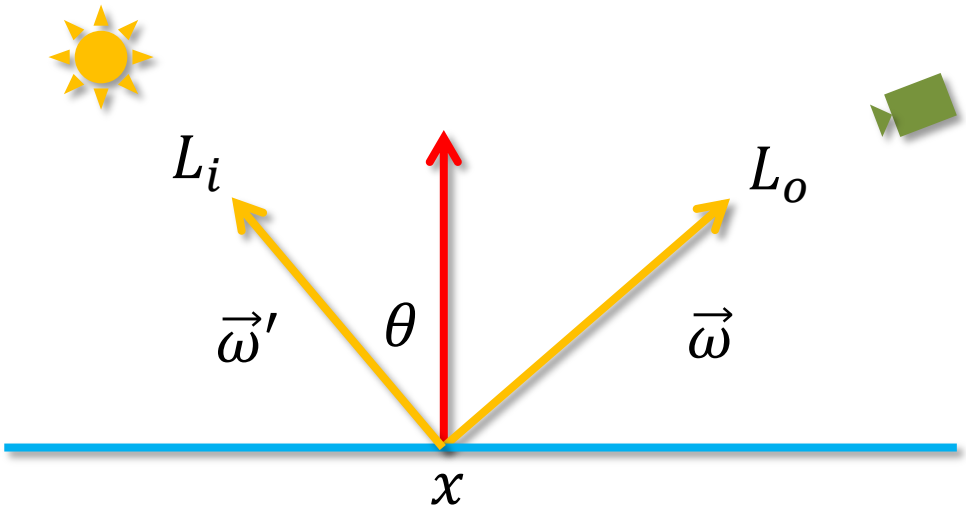
4.2 レンダリング方程式

レンダリング方程式

- **ある空間における光の伝達を記述した方程式**
  - 光の相互反射、グローバルイルミネーションが考慮された式

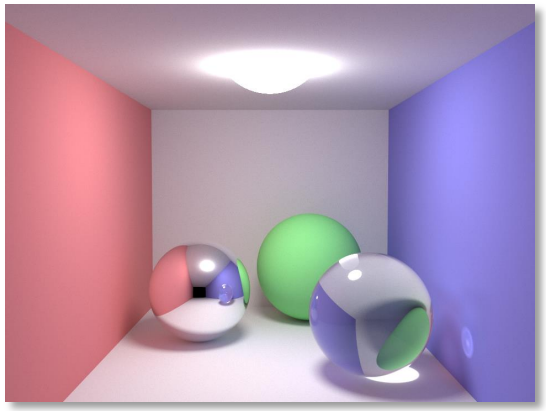
$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega_x} f_r(x,\vec{\omega},\vec{\omega}')L_i(x,\vec{\omega}')\cos\theta\, d\sigma(\vec{\omega}')$$

※真空を仮定

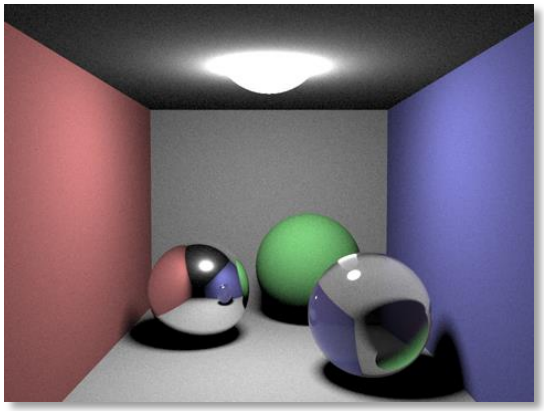


グローバルイルミネーション

- **光源から直接届く光の影響だけでなく、他の物体に一回以上反射して間接的に届く光の影響も計算に含めることでグローバルイルミネーションを考慮できる**
  - レンダリング方程式は間接的に届く光も考慮した式になっているため、これをきちんと解けばグローバルイルミネーションも考慮される。



間接光あり



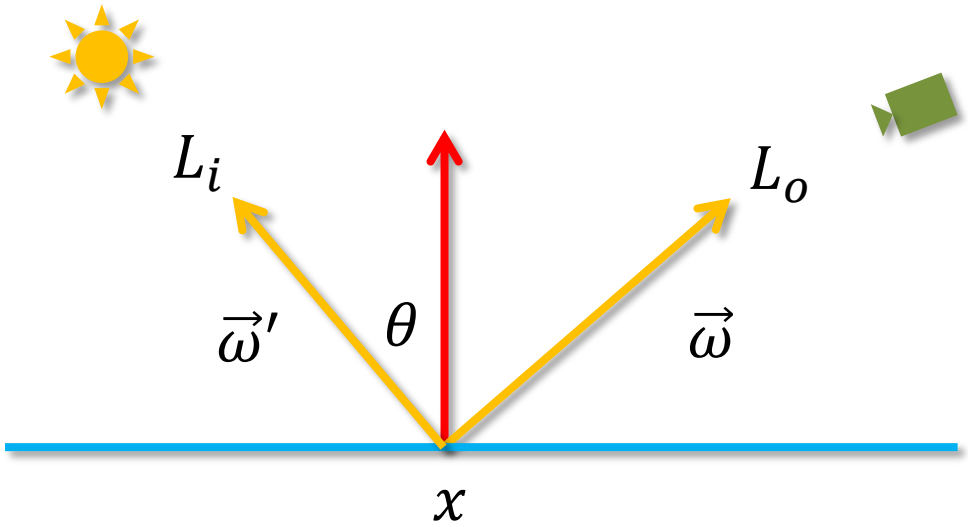
直接光のみ

レンダリング方程式

- ある空間における光の伝達を記述した方程式

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega_x} f_r(x,\vec{\omega},\vec{\omega}')L_i(x,\vec{\omega}')\cos\theta\, d\sigma(\vec{\omega}')$$

xから$\vec{\omega}$方向への放射輝度
画像の各ピクセルについてカメラからレイを飛ばしシーンとの交差点をxとすると、このL_oを求めることでカメラが受け取る放射輝度が求まる。＝シーンのレンダリングができる。

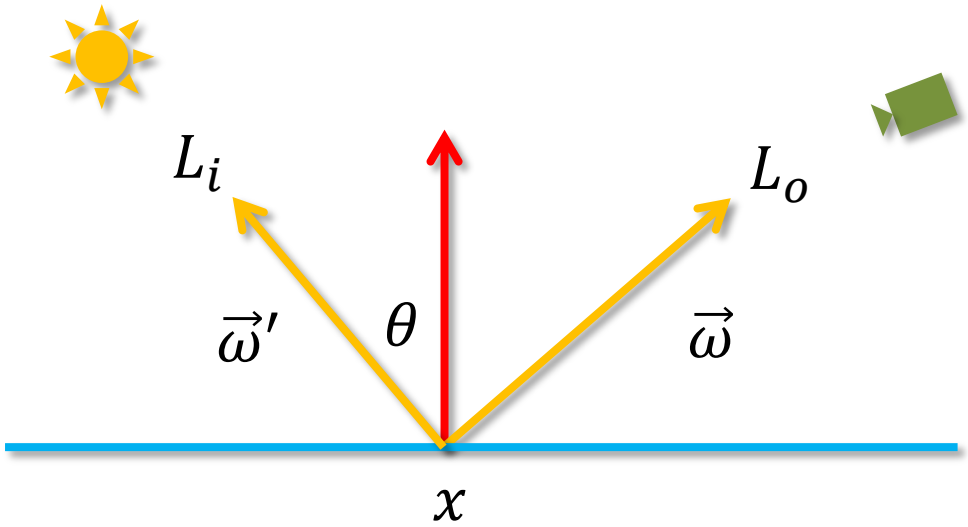


レンダリング方程式

- ある空間における光の伝達を記述した方程式

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega_x} f_r(x,\vec{\omega},\vec{\omega}')L_i(x,\vec{\omega}')\cos\theta\, d\sigma(\vec{\omega}')$$

xから$\vec{\omega}$方向への自己発光による放射輝度
位置xが光源上にあった場合のみ、この値は０じゃなくなる。Sphere構造体でいうところのemissionがこれ。

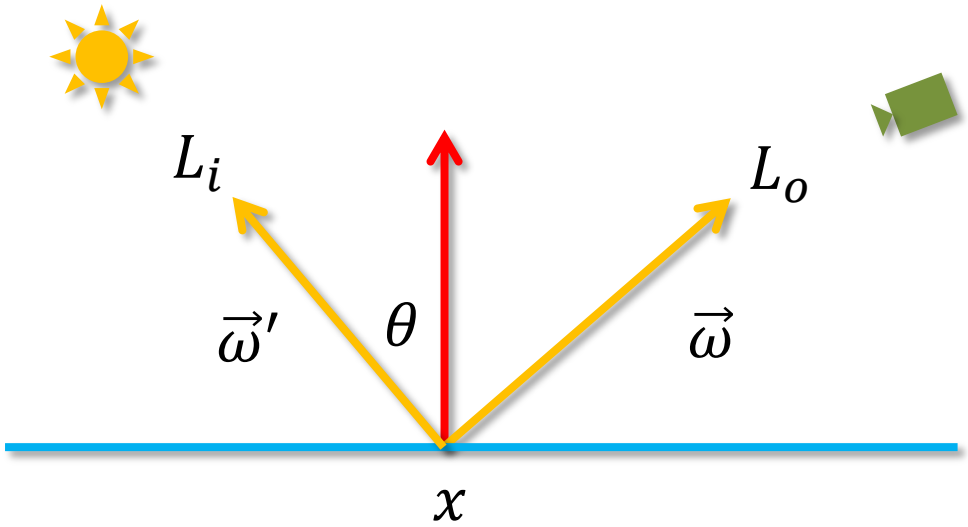


レンダリング方程式

- ある空間における光の伝達を記述した方程式

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega_x} f_r(x,\vec{\omega},\vec{\omega}')L_i(x,\vec{\omega}')\cos\theta\, d\sigma(\vec{\omega}')$$

xにおけるBRDF
$\vec{\omega}'$方向から届き位置xで$\vec{\omega}$方向に反射される光について、反射される程度を示す。放射輝度を放射照度で微分した量として定義される。

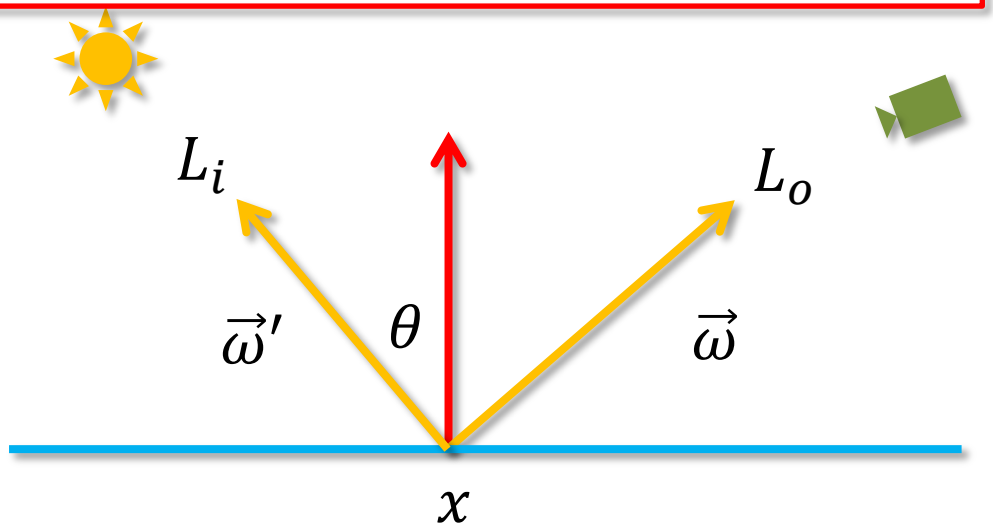


レンダリング方程式

- ある空間における光の伝達を記述した方程式

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega_x} f_r(x,\vec{\omega},\vec{\omega}')L_i(x,\vec{\omega}')\cos\theta\, d\sigma(\vec{\omega}')$$

$\vec{\omega}'$方向からxへの放射輝度
周囲からxへ降り注ぐ光。この入射光について位置xを中心とした半球上で積分することで、位置xに降り注ぐ光の総量を計算できる。それにBRDFをかけることで位置xに降り注ぎ$\vec{\omega}'$方向へと反射していく光L_oが求まる。

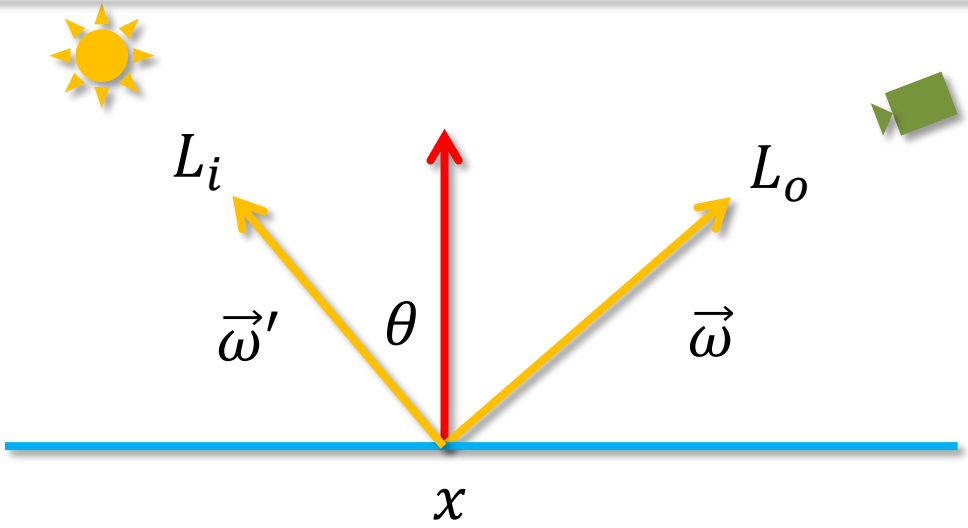


レンダリング方程式

- ある空間における光の伝達を記述した方程式

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega_x} f_r(x,\vec{\omega},\vec{\omega}')L_i(x,\vec{\omega}')\cos\theta\, d\sigma(\vec{\omega}')$$

コサイン項
面に垂直に入射する光と面に横から入射する光では、同じ放射輝度値でも前者の方が位置xにおける影響が強い。（極端な話、真横から入射した光は影響が無い）放射照度と放射輝度の関係式に出てくるcos項と本質的には同じ。

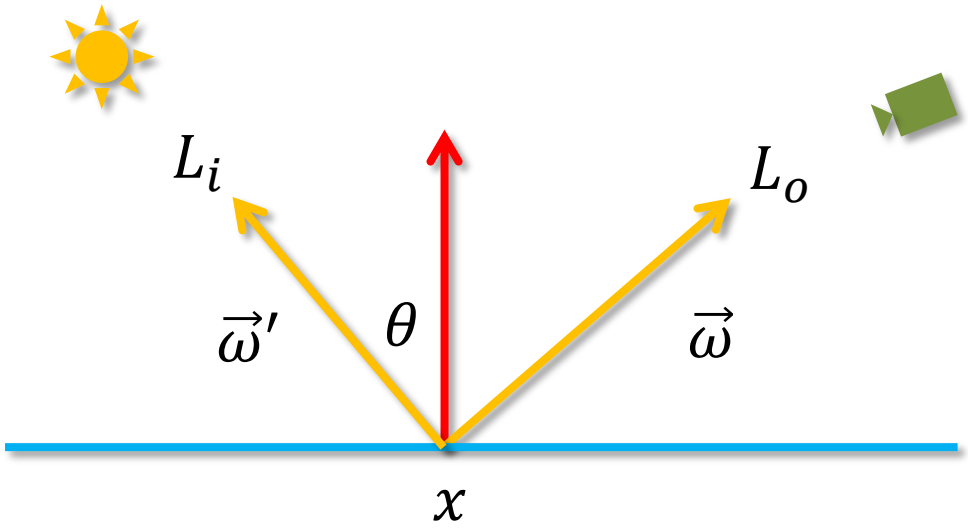


レンダリング方程式

- ある空間における光の伝達を記述した方程式

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega_x} f_r(x,\vec{\omega},\vec{\omega}')L_i(x,\vec{\omega}')\cos\theta\, d\sigma(\vec{\omega}')$$

xから$\vec{\omega}$方向へのレイとシーンの交差点を$r(x,\vec{\omega})$とすると
$L_i(x,\vec{\omega}') = L_o(r(x,\vec{\omega}'),-\vec{\omega}')$という関係が成り立ち
右の積分は再帰的な形になる。



4.3 モンテカルロ積分

モンテカルロ積分

- **$L_o(x, \vec{\omega})$を計算するには複雑な形の積分を解く必要がある**
  - 再帰的
  - 非連続
  - 高次元
- **モンテカルロ積分**
  - 被積分関数の値を何らかの形で計算できさえすれば解ける。
    - 再帰的に評価
  - 高次元な積分についても誤差の収束速度が次元数の影響を受けない。
    - 次元の呪いを受けない

モンテカルロ積分

- **積分したい式**
  - $f(x)$はxが与えられれば計算できるがこの積分を解くこと自体は難しい→モンテカルロ積分

$$I = \int_D f(x) d\mu(x)$$

- **モンテカルロ積分による推定器**

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

サンプル数N
X_iは確率変数で確率密度関数pdfに従ってサンプリングされる
（もちろん$X_i \in D$）

モンテカルロ積分

- **Nを増加させていくと推定値$\hat{I}$の真値Iに対する誤差は減少していく**
  - $O(\frac{1}{\sqrt{N}})$の速度で減少

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

Consistentな推定器（アルゴリズム）
Nを無限大に大きくしたとき$\hat{I}$が真値Iに収束する。
Unbiasedな推定器（アルゴリズム）
$\hat{I}$の期待値が真値Iと等しい。

レンダリング方程式に対するモンテカルロ積分

- 積分したい式

$$I = \int_{\Omega_x} f_r(x, \vec{\omega}, \vec{\omega}') L_i(x, \vec{\omega}') \cos\theta \, d\sigma(\vec{\omega}')$$

- モンテカルロ積分による推定器

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{f_r(x, \vec{\omega}, \vec{\omega}_i') L_i(x, \vec{\omega}_i') \cos\theta}{pdf(\vec{\omega}_i')}$$

サンプル数N
$\vec{\omega}_i'$は確率変数で確率密度関数pdfに従ってサンプリングされる。
このケースでは、$\vec{\omega}_i'$は半球上の方向を示すので、半球上で確率密度関数pdfに従ってN個の方向をサンプリングすることになる。

レンダリング方程式に対するモンテカルロ積分

●合体・展開

$$\widehat{L_o}(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \frac{1}{N}\sum_{i=1}^{N}\frac{f_r(x,\vec{\omega},\vec{\omega_i'})L_i(x,\vec{\omega_i'})\cos\theta}{pdf(\vec{\omega_i'})}$$

$L_i(x,\vec{\omega'}) = L_o(r(x,\vec{\omega'}),-\vec{\omega'})$より
$r(x,\vec{\omega'}) = x_1$とすると再びL_oについてのモンテカルロ積分になり

$$\widehat{L_o}(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \frac{1}{N}\sum_{i=1}^{N}\frac{f_r(x,\vec{\omega},\vec{\omega_i'})\cos\theta}{pdf(\vec{\omega_i'})}\left(L_e(x_1,-\vec{\omega_i'}) + \frac{1}{N}\sum_{j=1}^{N}\frac{f_r(x_1,-\vec{\omega_i'},\vec{\omega_j'})L_i(x_1,\vec{\omega_j'})\cos\theta}{pdf(\vec{\omega_j'})}\right)$$

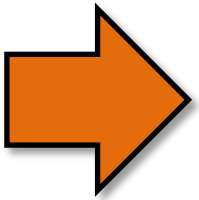
同様に

$$\widehat{L_o}(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \frac{1}{N}\sum_{i=1}^{N}\frac{f_r(x,\vec{\omega},\vec{\omega_i'})\cos\theta}{pdf(\vec{\omega_i'})}$$
$$\left(L_e(x_1,-\vec{\omega_i'}) + \frac{1}{N}\sum_{j=1}^{N}\frac{f_r(x_1,-\vec{\omega_i'},\vec{\omega_j'})\cos\theta}{pdf(\vec{\omega_j'})}\right.$$
$$\left.\left(L_e(x_2,-\vec{\omega_i'}) + \frac{1}{N}\sum_{k=1}^{N}\frac{f_r(x_2,-\vec{\omega_j'},\vec{\omega_k'})L_i(x_2,\vec{\omega_k'})\cos\theta}{pdf(\vec{\omega_k'})}\right)\right)$$

4.4 パストレーシング

パストレーシング

- **レンダリング方程式に対するモンテカルロ積分**
  - 半球上で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。
  - レイとシーンの交差点一つ一つについて、また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、を繰り返す。



反射のたびにレイがN倍になる
=
指数的にレイの数が増える！（爆発）

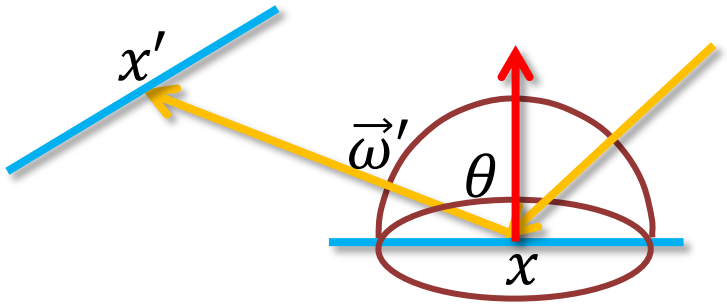
パストレーシング

- **反射のたびにサンプリングする方向の個数を1個に制限**
  - 指数の底が１になるのでレイの数が指数的に増加することがなくなる。
- **パストレーシングによるモンテカルロ積分**
  - 以下をそのままプログラムにしてやればよい。
  - $\widehat{L_o}$は確率変数になるので、何回も評価して（パストレーシングして）その平均をとることで真値L_oに収束する。

$$\widehat{L_o}(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \frac{f_r(x,\vec{\omega},\vec{\omega}_i')L_o(r(x,\vec{\omega}'),-\vec{\omega}')\cos\theta}{pdf(\vec{\omega}_i')}$$

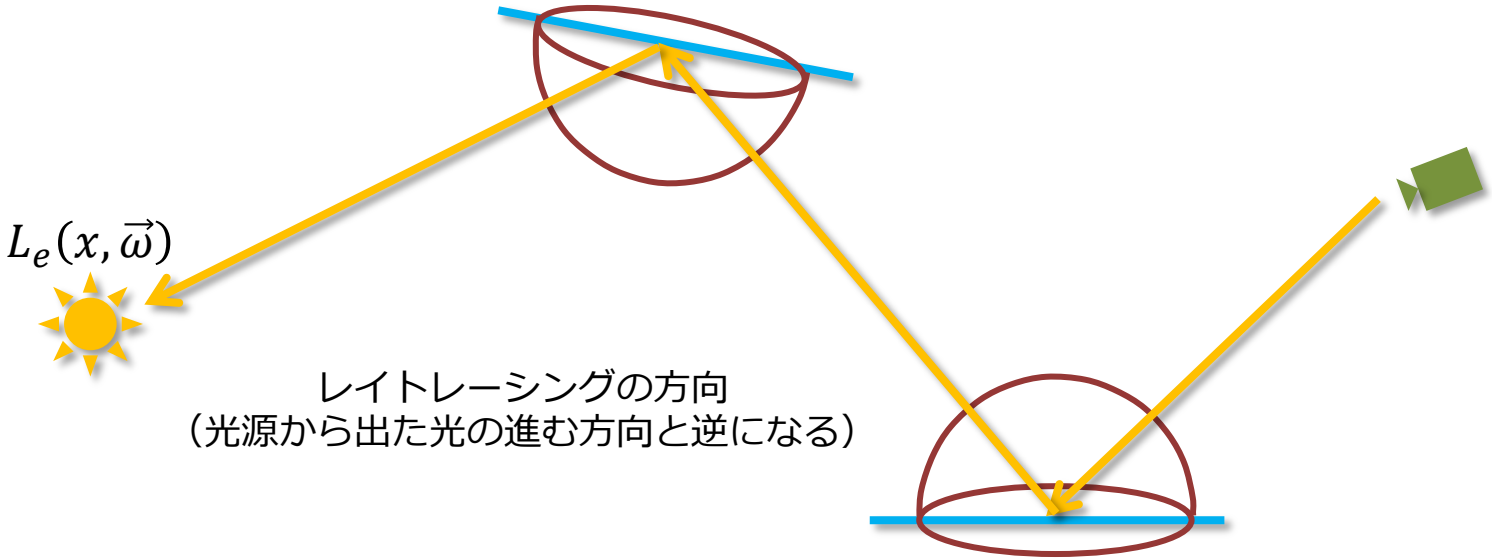
パストレーシング

- **擬似コード**
- $\widehat{L_o}(x,\vec{\omega})$**：位置**$x$**から**$\vec{\omega}$**方向への放射輝度**
  1. 位置xを中心とした半球上で1方向サンプリング。（サンプルは確率密度関数pdfに従うとする）
  2. 得られたサンプルを$\vec{\omega}'$とする
  3. 位置xから$\vec{\omega}'$へレイトレーシング。交差点をx'とする。
  4. θ=位置xにおける法線と$\vec{\omega}'$方向の成す角
  5. return $L_e(x,\vec{\omega}) + \dfrac{f_r(x,\vec{\omega},\vec{\omega}')\widehat{L_o}(x',-\vec{\omega}')\cos\theta}{pdf(\vec{\omega}')}$



パストレーシング

- **直感的には、光源から出た光がカメラに到達する経路をカメラ側から逆方向にレイトレーシングによって追跡して求めている、と考えることができる**
  - レイが物体に衝突するたびに半球上で次の方向をサンプリング
  - 光源に衝突すると、光源からカメラに至る経路の１つが得られた、ということになる
  - 衝突回数（＝反射回数）は何回相互反射したか、に相当



4.5 ロシアンルーレット

ロシアンルーレット

- **再帰の終了条件が無い！（＝無限ループ）**
  - 適当な回数再帰したら終了
    - 統計的Biasが入る。
      - 極端な話、再帰回数を一回に限定すると相互反射が一切考慮されない画像がレンダリングされる。真の画像（相互反射が考慮されたもの）と考慮されない画像の差がこの場合のBias。
  - ロシアンルーレット
    - 統計的にUnbiasedなアルゴリズムになる。

ロシアンルーレット

- **再帰を続けるか否かをある確率qに基づいて決める**
  - uを$[0, 1]$の乱数とし、 $0 < q \leq 1$とする。
    - $u < q$なら普通に再帰
      - ただし再帰によって得られた結果をqで割る
    - さもなければ処理を打ち切る
  - 期待値を計算すると、ロシアンルーレット導入以前と同じ値になる。
- **qの値は任意にとれるが、再帰によって得られる結果に比例させるように決めると分散が小さくなり誤差が小さくなりやすい**
  - eduptでは物体反射率に基づいて決めている。

パストレーシング

- **擬似コード（ロシアンルーレット）**
- **$\widehat{L_o}(x,\vec{\omega})$：位置$x$から$\vec{\omega}$方向への放射輝度**
  1. **0から1の範囲の乱数を得てqとの大小比較**
     1. **q以上ならreturn $L_e(x,\vec{\omega})$として再帰終了**
     2. **さもなければそのまま処理を続行**
  2. 位置xを中心とした半球上で1方向サンプリング。（サンプルは確率密度関数pdfに従うとする）
  3. 得られたサンプルを$\vec{\omega}'$とする
  4. 位置xから$\vec{\omega}'$へレイトレーシング。交差点をx'とする。
  5. θ=位置xにおける法線と$\vec{\omega}'$方向の成す角
  6. $$\text{return } L_e(x,\vec{\omega}) + \frac{f_r(x,\vec{\omega},\vec{\omega}')\widehat{L_o}(x',-\vec{\omega}')\cos\theta}{pdf(\vec{\omega}')}\cdot\frac{\mathbf{1}}{\boldsymbol{q}}$$

ロシアンルーレットの注意

- 無限ループに陥る確率は極めて低くなるとはいえ、ロシアンルーレットをたまたまうまく通過してしまった場合、再帰の深さが深くなりすぎてスタックオーバーフローすることがある
  - ある一定以上の深さになったらロシアンルーレットの通過確率を極端に下げる。
    - 深くなりすぎる確率は激減するが、それでも0ではない。
    - Unbiasedなレンダラにこだわる限り、0にすることはできない。
    - eduptはこの方法。
  - 非再帰パストレーシングにする。
    - うまく工夫すると非再帰版に書き直すことが出来る。
  - Unbiasedなレンダラを諦める
    - 再帰の回数に制限を設ければスタックオーバーフローを防ぐことができる（が、Biasが入る）

5.eduptコード解説
（レンダリング編）

5.eduptコード解説（レンダリング編）

- **まえおき**

  eduptにおけるメインのレンダリング部分の解説。パストレーシングをベースに、光のシーン内での挙動をシミュレーション。

edupt ファイル概要

- **main.cpp**
  edupt::render()を呼び出すだけ。
- **constant.h**
  各種定数。
- **intersection.h**
  交差判定の結果を格納する構造体IntersectionとHitpoint。
- **material.h**
  Color型と物体材質について記述するReflectionTypeと屈折率の設定。
- **ppm.h**
  レンダリング結果をppm画像として書き出すための関数save_ppm_file()。
- **radiance.h**
  ある方向からの放射輝度を得る関数radiance()。パストレーシングのメイン処理。
- **random.h**
  乱数生成クラスXorShift。
- **ray.h**
  一つ一つの光線、レイを表現する構造体Ray。
- **render.h**
  レンダリング画像サイズやサンプル数を受け取り、radiance()を使って各ピクセルの具体的な値を決定する関数render()。
- **scene.h**
  レンダリングするシーンのデータspheres[]とそのシーンに対する交差判定を行う関数intersect_scene()。
- **sphere.h**
  基本的な形状としての球を表現するSphere構造体。
- **vec.h**
  ベクトルを表現する構造体Vec。

main.cpp

- **edupt::render()を呼んでいるだけ**
  - レンダリング解像度
  - スーパーサンプリング解像度
  - サブピクセルあたりのサンプリング回数
    - を指定する

```
#include <iostream>

#include "render.h"

int main(int argc, char **argv) {
    std::cout << "Path tracing renderer: edupt" << std::endl << std::endl;

    // 640x480の画像、(2x2) * 4 sample / pixel
    edupt::render(640, 480, 4, 2);
}
```

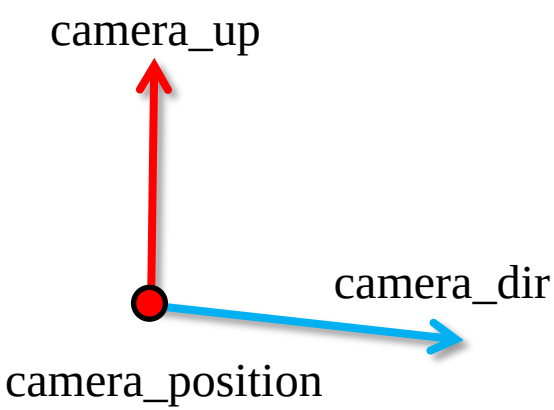
render.h

```
#ifndef _RENDER_H_
#define _RENDER_H_

#include <iostream>

#include "radiance.h"
#include "ppm.h"
#include "random.h"

namespace edupt {

int render(const int width, const int height, const int samples, const int supersamples) {
    // カメラ位置
    const Vec camera_position = Vec(50.0, 52.0, 220.0);
    const Vec camera_dir      = normalize(Vec(0.0, -0.04, -1.0));
    const Vec camera_up       = Vec(0.0, 1.0, 0.0);

    // ワールド座標系でのスクリーンの大きさ
    const double screen_width = 30.0 * width / height;
    const double screen_height= 30.0;
    // スクリーンまでの距離
    const double screen_dist  = 40.0;
    // スクリーンを張るベクトル
    const Vec screen_x = normalize(cross(camera_dir, camera_up)) * screen_width;
    const Vec screen_y = normalize(cross(screen_x, camera_dir)) * screen_height;
    const Vec screen_center = camera_position + camera_dir * screen_dist;

    Color *image = new Color[width * height];

    std::cout << width << "x" << height << " " << samples * (supersamples * supersamples) << " spp" << std::endl;

    // OpenMP
// #pragma omp parallel for schedule(dynamic, 1) num_threads(4)
    for (int y = 0; y < height; y ++) {
        std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%" << std::endl;

        Random rnd(y + 1);
        for (int x = 0; x < width; x ++) {
            const int image_index = (height - y - 1) * width + x;
            // supersamples x supersamples のスーパーサンプリング
            for (int sy = 0; sy < supersamples; sy ++) {
                for (int sx = 0; sx < supersamples; 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 < samples; s ++) {
                        const double rate = (1.0 / supersamples);
                        const double r1 = sx * rate + rate / 2.0;
                        const double r2 = sy * rate + rate / 2.0;
                        // スクリーン上の位置
                        const Vec screen_position =
                            screen_center +
                            screen_x * ((r1 + x) / width - 0.5) +
                            screen_y * ((r2 + y) / height- 0.5);
                        // レイを飛ばす方向
                        const Vec dir = normalize(screen_position - camera_position);

                        accumulated_radiance = accumulated_radiance +
                            radiance(Ray(camera_position, dir), &rnd, 0) / samples / (supersamples * supersamples);
                    }
                    image[image_index] = image[image_index] + accumulated_radiance;
                }
            }
        }
    }

    // 出力
    save_ppm_file(std::string("image.ppm"), image, width, height);
    return 0;
}


};

#endif
```

5.1 カメラ設定

カメラの設定

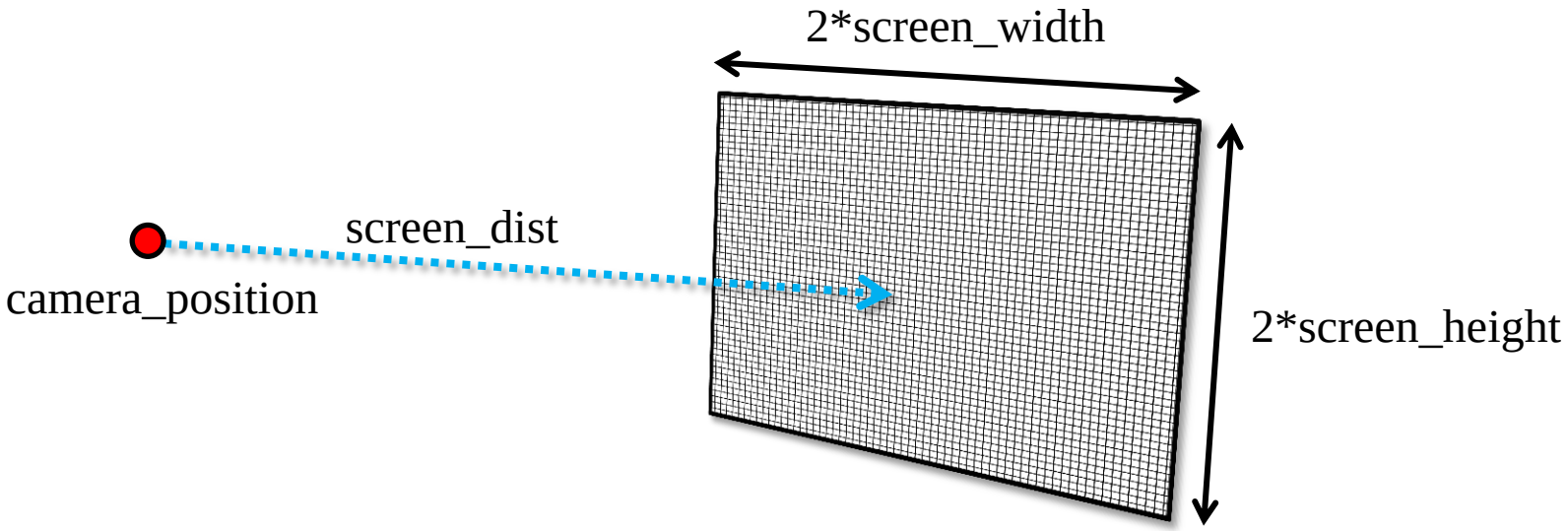
●カメラ姿勢の設定




```
int render(const int width, const int height, const int samples, const int supersamples) {
    // カメラ位置
    const Vec camera_position = Vec(50.0, 52.0, 220.0);
    const Vec camera_dir      = normalize(Vec(0.0, -0.04, -1.0));
    const Vec camera_up       = Vec(0.0, 1.0, 0.0);

    // ワールド座標系でのスクリーンの大きさ
    const double screen_width = 30.0 * width / height;
    const double screen_height= 30.0;
    // スクリーンまでの距離
    const double screen_dist  = 40.0;
    // スクリーンを張るベクトル
    const Vec screen_x = normalize(cross(camera_dir, camera_up)) * screen_width;
    const Vec screen_y = normalize(cross(screen_x, camera_dir)) * screen_height;
    const Vec screen_center = camera_position + camera_dir * screen_dist;
```

カメラの設定

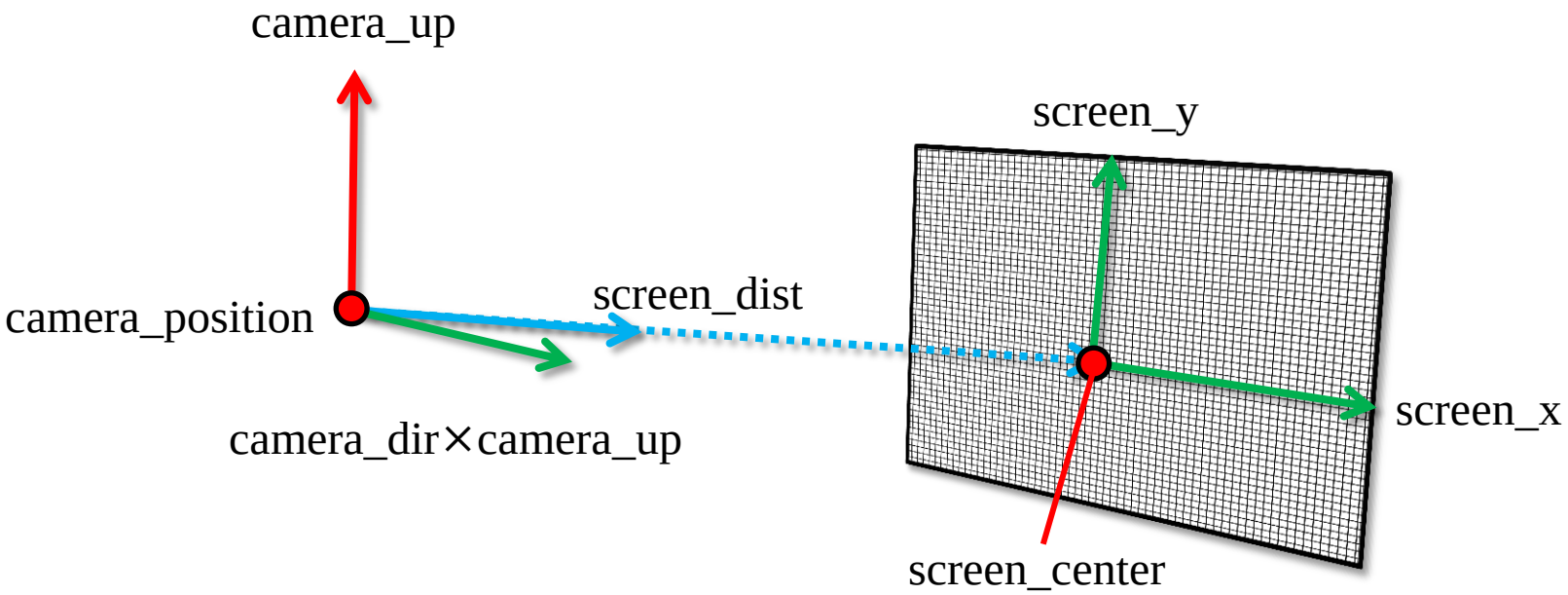
●スクリーンの設定




```
int render(const int width, const int height, const int samples, const int supersamples) {
    // カメラ位置
    const Vec camera_position = Vec(50.0, 52.0, 220.0);
    const Vec camera_dir      = normalize(Vec(0.0, -0.04, -1.0));
    const Vec camera_up       = Vec(0.0, 1.0, 0.0);

    // ワールド座標系でのスクリーンの大きさ
    const double screen_width = 30.0 * width / height;
    const double screen_height= 30.0;
    // スクリーンまでの距離
    const double screen_dist  = 40.0;
    // スクリーンを張るベクトル
    const Vec screen_x = normalize(cross(camera_dir, camera_up)) * screen_width;
    const Vec screen_y = normalize(cross(screen_x, camera_dir)) * screen_height;
    const Vec screen_center = camera_position + camera_dir * screen_dist;
```

カメラの設定

●スクリーンを張るベクトルの設定




```
int render(const int width, const int height, const int samples, const int supersamples) {
    // カメラ位置
    const Vec camera_position = Vec(50.0, 52.0, 220.0);
    const Vec camera_dir      = normalize(Vec(0.0, -0.04, -1.0));
    const Vec camera_up       = Vec(0.0, 1.0, 0.0);

    // ワールド座標系でのスクリーンの大きさ
    const double screen_width = 30.0 * width / height;
    const double screen_height= 30.0;
    // スクリーンまでの距離
    const double screen_dist  = 40.0;
    // スクリーンを張るベクトル
    const Vec screen_x = normalize(cross(camera_dir, camera_up)) * screen_width;
    const Vec screen_y = normalize(cross(screen_x, camera_dir)) * screen_height;
    const Vec screen_center = camera_position + camera_dir * screen_dist;
```

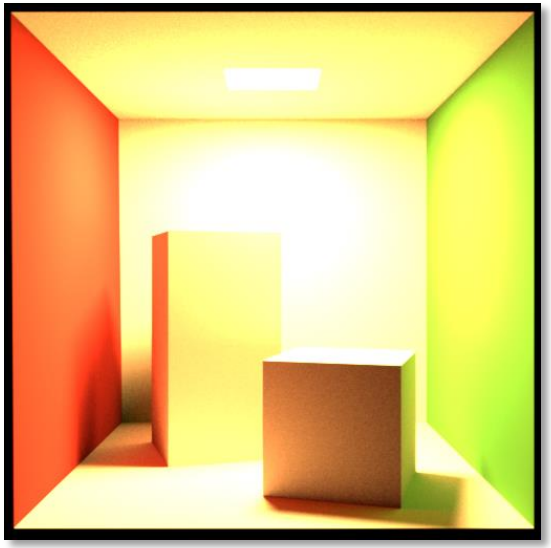
画像バッファの作成

- **レンダリング結果はColor型の配列に格納**
  - 最終的な画像として出力するときは256階調LDRイメージだが、計算途中ではHDRイメージとしてバッファに保存する。

```
Color *image = new Color[width * height];

std::cout << width << "x" << height << " " << samples * (supersamples * supersamples) << " spp" << std::endl;
```

LDRイメージ

- **普通のbmpやpng等の画像はLow Dynamic Range画像**
  - 表現できる輝度の幅が狭い。
  - RGBそれぞれ8ビットの階調が一般的。
  - 高輝度部分が白飛び。
  - 低輝度部分が黒くつぶれる。



白飛び

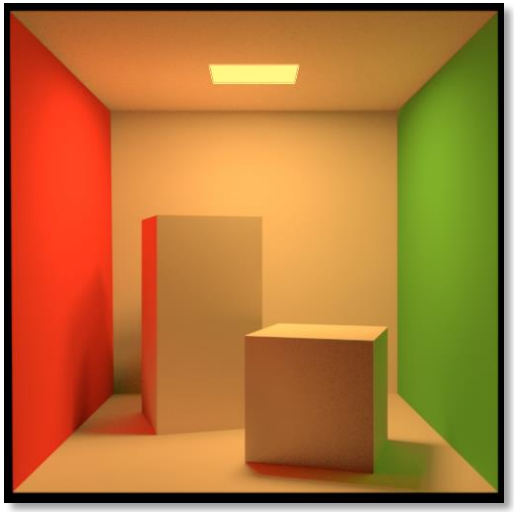
黒つぶれ

HDRイメージ

- **物理ベースレンダリングではより表現できる輝度の幅が大きいHigh Dynamic Range画像がしばしば使われる**
  - 物理モデルによる画像は自然界同様非常にダイナミックレンジが広いため。
  - RGBそれぞれ8ビットより大きい階調・幅（32ビット等）。
  - ディスプレイはLDRによる表示しか対応していないことがほとんどなので、HDR画像を表示するにはなんらかの方法でLDR画像に変換する必要がある　→　**トーンマッピング**
- **代表的なフォーマット**
  - Radiance (.hdr)
  - OpenEXR (.exr)
    - http://www.openexr.com/
  - Floating point TTF

トーンマッピング

- **eduptではトーンマッピングは深く扱わない**
  - 計算途中ではHDRとして記録しておくも、最終的な保存時には高輝度部分を1.0に切り捨て、輝度値が0.0から1.0の範囲に収まるようにする。



トーンマッピングの例

5.2 画像生成

画像生成

```
    // OpenMP
// #pragma omp parallel for schedule(dynamic, 1) num_threads(4)
    for (int y = 0; y < height; y ++) {
        std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%" << std::endl;

        Random rnd(y + 1);
        for (int x = 0; x < width; x ++) {
            const int image_index = (height - y - 1) * width + x;
            // supersamples x supersamples のスーパーサンプリング
            for (int sy = 0; sy < supersamples; sy ++) {
                for (int sx = 0; sx < supersamples; 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 < samples; s ++) {
                        const double rate = (1.0 / supersamples);
                        const double r1 = sx * rate + rate / 2.0;
                        const double r2 = sy * rate + rate / 2.0;
                        // スクリーン上の位置
                        const Vec screen_position =
                            screen_center +
                            screen_x * ((r1 + x) / width - 0.5) +
                            screen_y * ((r2 + y) / height- 0.5);
                        // レイを飛ばす方向
                        const Vec dir = normalize(screen_position - camera_position);

                        accumulated_radiance = accumulated_radiance +
                            radiance(Ray(camera_position, dir), &rnd, 0) / samples / (supersamples * supersamples);
                    }
                    image[image_index] = image[image_index] + accumulated_radiance;
                }
            }
        }
    }
```

画像生成

- **OpenMP**
  - ディレクティブを指定することで簡単に並列化。
  - eduptでは画像の行ごとに別々のスレッドで処理させている。
- **並列化の処理単位は奥深い問題**
  - なるべく全てのコアを常に100%使用したい。

CGのレンダリングはピクセル間の依存が少なかったり無かったりすることが多いため、非常に並列化しやすい問題として知られている。そのため、このように単純なディレクティブを一行指定するだけでもうまく並列化され、高速化が期待できる。

```
    // OpenMP
// #pragma omp parallel for schedule(dynamic, 1) num_threads(4)
    for (int y = 0; y < height; y ++) {
        std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%" << std::endl;

        Random rnd(y + 1);
        for (int x = 0; x < width; x ++) {
            const int image_index = (height - y - 1) * width + x;
            // supersamples x supersamples のスーパーサンプリング
            for (int sy = 0; sy < supersamples; sy ++) {
                for (int sx = 0; sx < supersamples; 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 < samples; s ++) {
                        const double rate = (1.0 / supersamples);
                        const double r1 = sx * rate + rate / 2.0;
                        const double r2 = sy * rate + rate / 2.0;
                        // スクリーン上の位置
                        const Vec screen_position =
                            screen_center +
                            screen_x * ((r1 + x) / width - 0.5) +
                            screen_y * ((r2 + y) / height- 0.5);
                        // レイを飛ばす方向
                        const Vec dir = normalize(screen_position - camera_position);

                        accumulated_radiance = accumulated_radiance +
                            radiance(Ray(camera_position, dir), &rnd, 0) / samples / (supersamples * supersamples);
                    }
                    image[image_index] = image[image_index] + accumulated_radiance;
                }
            }
        }
    }
```

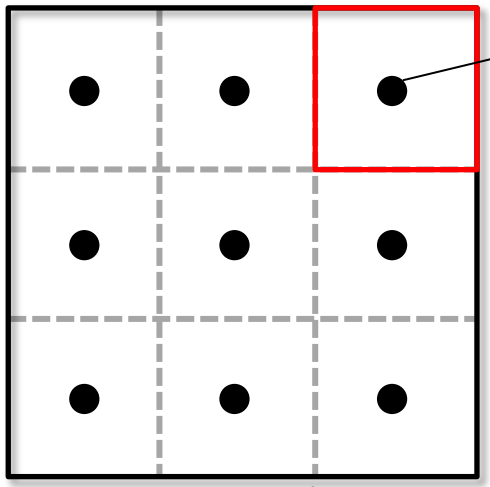
画像生成

- **画像の各ピクセルについて、走査**
  - 画像の行ごとに並列化される可能性があるため、行ごとに乱数生成機を初期化して使用する。
  - 画像の行をまたいで乱数生成機を共有すると競合が発生して結果がおかしくなりうる。
- **image_indexがColor配列に対するインデックス**

```
    // OpenMP
// #pragma omp parallel for schedule(dynamic, 1) num_threads(4)
    for (int y = 0; y < height; y ++) {
        std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%" << std::endl;

        Random rnd(y + 1);
        for (int x = 0; x < width; x ++) {
            const int image_index = (height - y - 1) * width + x;
            // supersamples x supersamples のスーパーサンプリング
            for (int sy = 0; sy < supersamples; sy ++) {
                for (int sx = 0; sx < supersamples; 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 < samples; s ++) {
                        const double rate = (1.0 / supersamples);
                        const double r1 = sx * rate + rate / 2.0;
                        const double r2 = sy * rate + rate / 2.0;
                        // スクリーン上の位置
                        const Vec screen_position =
                            screen_center +
                            screen_x * ((r1 + x) / width - 0.5) +
                            screen_y * ((r2 + y) / height- 0.5);
                        // レイを飛ばす方向
                        const Vec dir = normalize(screen_position - camera_position);

                        accumulated_radiance = accumulated_radiance +
                            radiance(Ray(camera_position, dir), &rnd, 0) / samples / (supersamples * supersamples);
                    }
                    image[image_index] = image[image_index] + accumulated_radiance;
                }
            }
        }
    }
```

画像生成

- **スーパーサンプリング**
  - 画像の各ピクセルについて、ピクセルをさらに小領域に分割し、その各領域について輝度値を求める。
  - 最終的な各ピクセルの輝度値は小領域全ての平均とする。



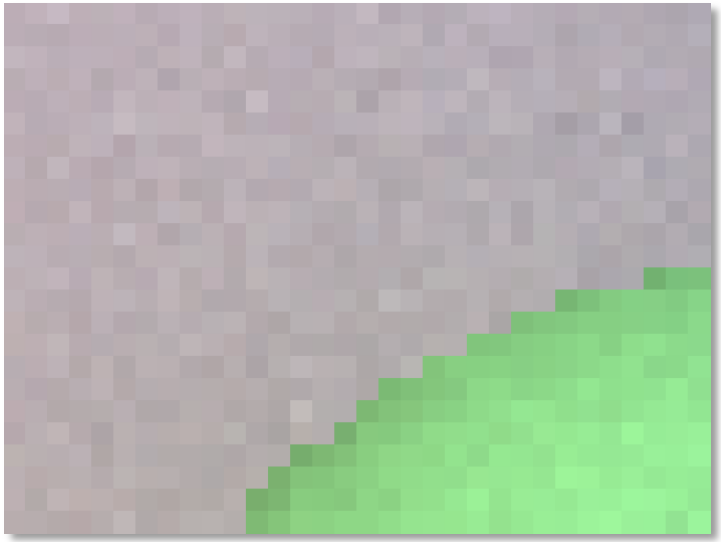
画像上のあるピクセルを
3x3スーパーサンプリングする

サンプリング点（この方向にレイを飛ばして得られた輝度値をこのサブピクセルの色とする）

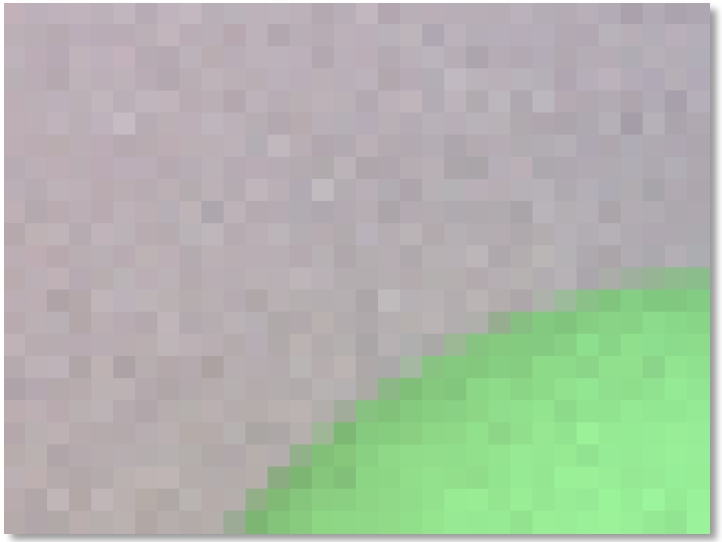
```
   // OpenMP
// #pragma omp parallel for schedule(dynamic, 1) num_threads(4)
    for (int y = 0; y < height; y ++) {
        std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%" << std::endl;

        Random rnd(y + 1);
        for (int x = 0; x < width; x ++) {
            const int image_index = (height - y - 1) * width + x;
            // supersamples x supersamples のスーパーサンプリング
            for (int sy = 0; sy < supersamples; sy ++) {
                for (int sx = 0; sx < supersamples; 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 < samples; s ++) {
                        const double rate = (1.0 / supersamples);
                        const double r1 = sx * rate + rate / 2.0;
                        const double r2 = sy * rate + rate / 2.0;
                        // スクリーン上の位置
                        const Vec screen_position =
                            screen_center +
                            screen_x * ((r1 + x) / width - 0.5) +
                            screen_y * ((r2 + y) / height- 0.5);
                        // レイを飛ばす方向
                        const Vec dir = normalize(screen_position - camera_position);

                        accumulated_radiance = accumulated_radiance +
                            radiance(Ray(camera_position, dir), &rnd, 0) / samples / (supersamples * supersamples);
                    }
                    image[image_index] = image[image_index] + accumulated_radiance;
                }
            }
        }
    }
```

スーパーサンプリング

- アンチエイリアスが目的



スーパーサンプリングなし

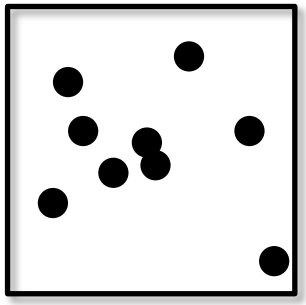


16x16スーパーサンプリング

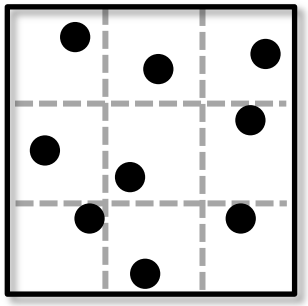
※ピクセルあたりの総サンプリング回数は同じ

アンチエイリアス

- ランダムサンプリング



- 層化サンプリング（ジッターサンプリング）



- 準モンテカルロ法(QMC)
- 一様ジッターサンプリング
- etc

画像生成

- **スクリーン上での各サンプリング点の位置**
  - ピクセルを縦横supersamples個に分割。
  - 分割した各小領域＝サブピクセルの中心をサンプリング点としてその位置を求める。
    - その位置からレイを発射、そのピクセルの輝度値を計算することになる。

```
    // OpenMP
// #pragma omp parallel for schedule(dynamic, 1) num_threads(4)
    for (int y = 0; y < height; y ++) {
        std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%" << std::endl;

        Random rnd(y + 1);
        for (int x = 0; x < width; x ++) {
            const int image_index = (height - y - 1) * width + x;
            // supersamples x supersamples のスーパーサンプリング
            for (int sy = 0; sy < supersamples; sy ++) {
                for (int sx = 0; sx < supersamples; 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 < samples; s ++) {
                        const double rate = (1.0 / supersamples);
                        const double r1 = sx * rate + rate / 2.0;
                        const double r2 = sy * rate + rate / 2.0;
                        // スクリーン上の位置
                        const Vec screen_position =
                            screen_center +
                            screen_x * ((r1 + x) / width - 0.5) +
                            screen_y * ((r2 + y) / height- 0.5);
                        // レイを飛ばす方向
                        const Vec dir = normalize(screen_position - camera_position);

                        accumulated_radiance = accumulated_radiance +
                            radiance(Ray(camera_position, dir), &rnd, 0) / samples / (supersamples * supersamples);
                    }
                    image[image_index] = image[image_index] + accumulated_radiance;
                }
            }
        }
    }
```

画像生成

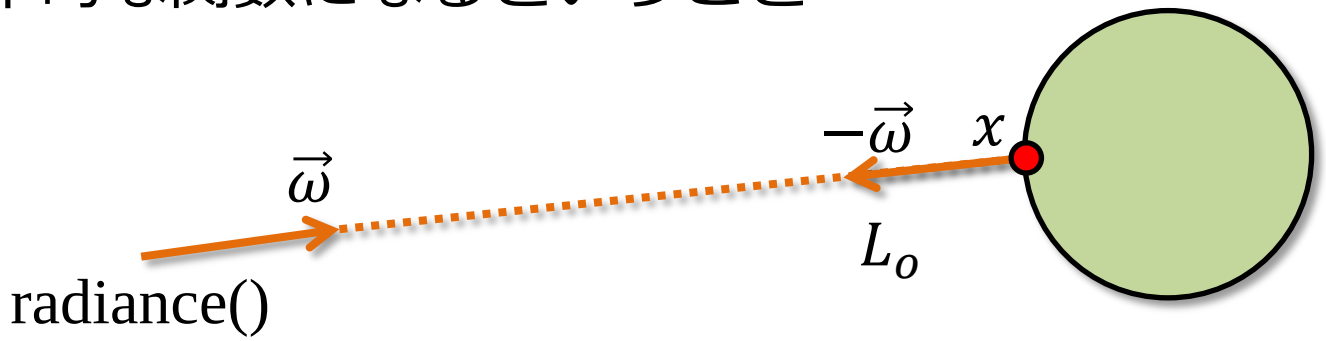
- **サンプリングする点が決まったらその方向にレイを飛ばし、その方向からの放射輝度をradiance()によって得る**
  - radiance()は確率的な関数なので誤差を含む。
  - 同じ方向についてsamples回radiance()を実行し、その平均をその方向からの最終的な放射輝度値とする。

```
    // OpenMP
// #pragma omp parallel for schedule(dynamic, 1) num_threads(4)
    for (int y = 0; y < height; y ++) {
        std::cerr << "Rendering (y = " << y << ") " << (100.0 * y / (height - 1)) << "%" << std::endl;

        Random rnd(y + 1);
        for (int x = 0; x < width; x ++) {
            const int image_index = (height - y - 1) * width + x;
            // supersamples x supersamples のスーパーサンプリング
            for (int sy = 0; sy < supersamples; sy ++) {
                for (int sx = 0; sx < supersamples; 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 < samples; s ++) {
                        const double rate = (1.0 / supersamples);
                        const double r1 = sx * rate + rate / 2.0;
                        const double r2 = sy * rate + rate / 2.0;
                        // スクリーン上の位置
                        const Vec screen_position =
                            screen_center +
                            screen_x * ((r1 + x) / width - 0.5) +
                            screen_y * ((r2 + y) / height- 0.5);
                        // レイを飛ばす方向
                        const Vec dir = normalize(screen_position - camera_position);

                        accumulated_radiance = accumulated_radiance +
                            radiance(Ray(camera_position, dir), &rnd, 0) / samples / (supersamples * supersamples);
                    }
                    image[image_index] = image[image_index] + accumulated_radiance;
                }
            }
        }
    }
```

5.3 radiance()

radiance()

- **radiance()はレイの方向からの放射輝度値を計算する関数**
  - レイとシーンの交差をxとすると$L_o(x, -\vec{\omega})$の値を求めればよいことがわかる
  - L_oはモンテカルロ積分によって求まるのでradiance()も確率的な関数になるということ



$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_{\Omega_x} f_r(x, \vec{\omega}, \vec{\omega}') L_i(x, \vec{\omega}') \cos\theta \, d\sigma(\vec{\omega}')$$

レンダリング方程式

radiance()のアルゴリズム

- **擬似コード（パストレーシングによるL_oの推定を基にしている）**
- **radiance$(x, \vec{\omega})$：位置xに$\vec{\omega}$方向からくる放射輝度**
  1. xから$\vec{\omega}$方向にレイトレーシング。交差点をx'とする
  2. 0から1の範囲の乱数を得てqとの大小比較
     1. q以上ならreturn $L_e(x', \vec{\omega})$として再帰終了
     2. さもなければそのまま処理を続行
  3. 位置x'を中心とした半球上で1方向サンプリング。（サンプルは確率密度関数pdfに従うとする）
  4. 得られたサンプルを$\vec{\omega}'$とする
  5. θ=位置x'における法線と$\vec{\omega}'$方向の成す角
  6. return $L_e(x', \vec{\omega}) + \frac{f_r(x', \vec{\omega}, \vec{\omega}')radiance(x', \vec{\omega}') \cos\theta}{pdf(\vec{\omega}')} \cdot \frac{1}{q}$

radiance.h

```cpp
#ifndef _RADIANCE_H_
#define _RADIANCE_H_

#include <algorithm>

#include "ray.h"
#include "scene.h"
#include "sphere.h"
#include "intersection.h"
#include "random.h"

namespace edupt {

const Color kBackgroundColor = Color(0.0, 0.0, 0.0);
const int kDepth = 5; // ロシアンルーレットで打ち切らない最大深度
const int kDpethLimit = 64;

// ray方向からの放射輝度を求める
Color radiance(const Ray &ray, Random *rnd, const int depth) {
    Intersection intersection;
    // シーンと交差判定
    if (!intersect_scene(ray, &intersection))
        return kBackgroundColor;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , 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）
    // 色の反射率最大のものを得る。ロシアンルーレットで使う。
    // ロシアンルーレットの閾値は任意だが色の反射率等を使うとより良い。
    double russian_roulette_probability = std::max(now_object.color.x, std::max(now_object.color.y, now_object.color.z));

    // 反射回数が一定以上になったらロシアンルーレットの確率を急上昇させる。（スタックオーバーフロー対策）
    if (depth > kDpethLimit)
        russian_roulette_probability *= pow(0.5, depth - kDpethLimit);

    // ロシアンルーレットを実行し追跡を打ち切るかどうかを判断する。
    // ただしDepth回の追跡は保障する。
    if (depth > kDepth) {
        if (rnd->next01() >= russian_roulette_probability)
            return now_object.emission;
    } else
        russian_roulette_probability = 1.0; // ロシアンルーレット実行しなかった

    Color incoming_radiance;
    Color weight = 1.0;

    switch (now_object.reflection_type) {
    // 完全拡散面
    case REFLECTION_TYPE_DIFFUSE: {
        // orienting_normalの方向を基準とした正規直交基底(w, u, v)を作る。この基底に対する半球内で次のレイを飛ばす。
        Vec w, u, v;
        w = orienting_normal;
        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)));

        incoming_radiance = radiance(Ray(hitpoint.position, dir), rnd, depth+1);
        // レンダリング方程式に対するモンテカルロ積分を考えると、outgoing_radiance = weight * incoming_radiance。
        // ここで、weight = (ρ/π) * cosθ / pdf(ω) / R になる。
        // ρ/πは完全拡散面のBRDFでρは反射率、cosθはレンダリング方程式におけるコサイン項、pdf(ω)はサンプリング方向についての確率密度関数。
        // Rはロシアンルーレットの確率。
        //今、コサイン項に比例した確率密度関数によるサンプリングを行っているため、pdf(ω) = cosθ/π
        // よって、weight = ρ/ R。
        weight = now_object.color / russian_roulette_probability;
    } break;

    // 完全鏡面
    case REFLECTION_TYPE_SPECULAR: {
        // 完全鏡面なのでレイの反射方向は決定的。
        // ロシアンルーレットの確率で除算するのは上と同じ。
        incoming_radiance = radiance(Ray(hitpoint.position, ray.dir - hitpoint.normal * 2.0 * dot(hitpoint.normal, ray.dir)), rnd, depth+1);
        weight = now_object.color / russian_roulette_probability;
    } break;

    // 屈折率kIorのガラス
    case REFLECTION_TYPE_REFRACTION: {
        const Ray reflection_ray = Ray(hitpoint.position, ray.dir - hitpoint.normal * 2.0 * dot(hitpoint.normal, ray.dir));
        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

        // Snellの法則
        const double nc = 1.0; // 真空の屈折率
        const double nt = kIor; // オブジェクトの屈折率
        const double nnt = into ? nc / nt : nt / nc;
        const double ddn = dot(ray.dir, orienting_normal);
        const double cos2t = 1.0 - nnt * nnt * (1.0 - ddn * ddn);

        if (cos2t < 0.0) { // 全反射
            incoming_radiance = radiance(reflection_ray, rnd, depth+1);
            weight = now_object.color / russian_roulette_probability;
            break;
        }
        // 屈折の方向
        const Ray refraction_ray = Ray(hitpoint.position,
            normalize(ray.dir * nnt - hitpoint.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 = nt - nc, b = nt + nc;
        const double R0 = (a * a) / (b * b);

        const double c = 1.0 - (into ? -ddn : dot(refraction_ray.dir, -1.0 * orienting_normal));
        const double Re = 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。
        const double nnt2 = pow(into ? nc / nt : nt / nc, 2.0); // 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 Tr = (1.0 - Re) * nnt2; // 屈折方向の光が屈折してray.dirの方向に運ぶ割合

        // 一定以上レイを追跡したら屈折と反射のどちらか一方を追跡する。（さもないと指数的にレイが増える）
        // ロシアンルーレットで決定する。
        const double probability  = 0.25 + 0.5 * Re;
        if (depth > 2) {
            if (rnd->next01() < probability) { // 反射
                incoming_radiance = radiance(reflection_ray, rnd, depth+1) * Re;
                weight = now_object.color / (probability * russian_roulette_probability);
            } else { // 屈折
                incoming_radiance = radiance(refraction_ray, rnd, depth+1) * Tr;
                weight = now_object.color / ((1.0 - probability) * russian_roulette_probability);
            }
        } else { // 屈折と反射の両方を追跡
            incoming_radiance =
                radiance(reflection_ray, rnd, depth+1) * Re +
                radiance(refraction_ray, rnd, depth+1) * Tr;
            weight = now_object.color / (russian_roulette_probability);
        }
    } break;

    }

    return now_object.emission + multiply(weight, incoming_radiance);
}

};

#endif
```

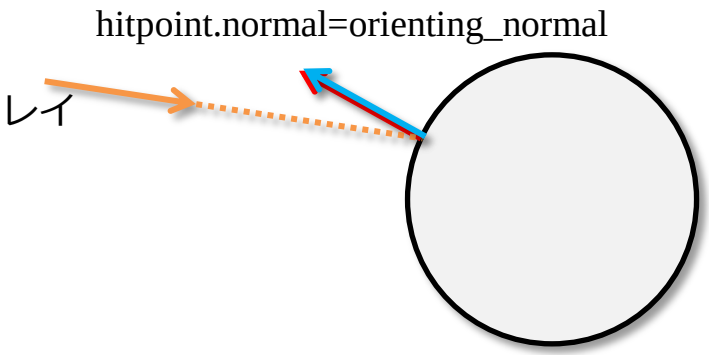
radiance()

- **ray方向にレイトレーシングして交差点計算**
  - 交差しなかったら背景と交差したとして Backgroundcolorを返して終了。

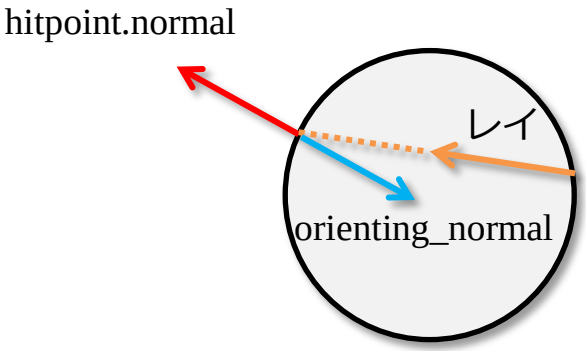
```
// ray方向からの放射輝度を求める
Color radiance(const Ray &ray, Random *rnd, const int depth) {
    Intersection intersection;
    // シーンと交差判定
    if (!intersect_scene(ray, &intersection))
        return kBackgroundColor;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , 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）
    // 色の反射率最大のものを得る。ロシアンルーレットで使う。
    // ロシアンルーレットの閾値は任意だが色の反射率等を使うとより良い。
    double russian_roulette_probability = std::max(now_object.color.x, std::max(now_object.color.y, now_object.color.z));

    // 反射回数が一定以上になったらロシアンルーレットの確率を急上昇させる。（スタックオーバーフロー対策）
    if (depth > kDpethLimit)
        russian_roulette_probability *= pow(0.5, depth - kDpethLimit);

    // ロシアンルーレットを実行し追跡を打ち切るかどうかを判断する。
    // ただしDepth回の追跡は保障する。
    if (depth > kDepth) {
        if (rnd->next01() >= russian_roulette_probability)
            return now_object.emission;
    } else
        russian_roulette_probability = 1.0; // ロシアンルーレット実行しなかった

    Color incoming_radiance;
    Color weight = 1.0;
```

radiance()

- **orienting_normal**
  - レイの向きと物体法線の内積を計算することで二つのベクトルの成す角のcosθが計算できる。この正負によってレイが物体に外から衝突するのか中から衝突するのかを判定でき、レイに対応する法線orienting_normalを得られる。



外から衝突するケース



中から衝突するケース

```
// ray方向からの放射輝度を求める
Color radiance(const Ray &ray, Random *rnd, const int depth) {
    Intersection intersection;
    // シーンと交差判定
    if (!intersect_scene(ray, &intersection))
        return kBackgroundColor;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , 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）
    // 色の反射率最大のものを得る。ロシアンルーレットで使う。
    // ロシアンルーレットの閾値は任意だが色の反射率等を使うとより良い。
    double russian_roulette_probability = std::max(now_object.color.x, std::max(now_object.color.y, now_object.color.z));

    // 反射回数が一定以上になったらロシアンルーレットの確率を急上昇させる。（スタックオーバーフロー対策）
    if (depth > kDpethLimit)
        russian_roulette_probability *= pow(0.5, depth - kDpethLimit);

    // ロシアンルーレットを実行し追跡を打ち切るかどうかを判断する。
    // ただしDepth回の追跡は保障する。
    if (depth > kDepth) {
        if (rnd->next01() >= russian_roulette_probability)
            return now_object.emission;
    } else
        russian_roulette_probability = 1.0; // ロシアンルーレット実行しなかった

    Color incoming_radiance;
    Color weight = 1.0;
```

radiance()

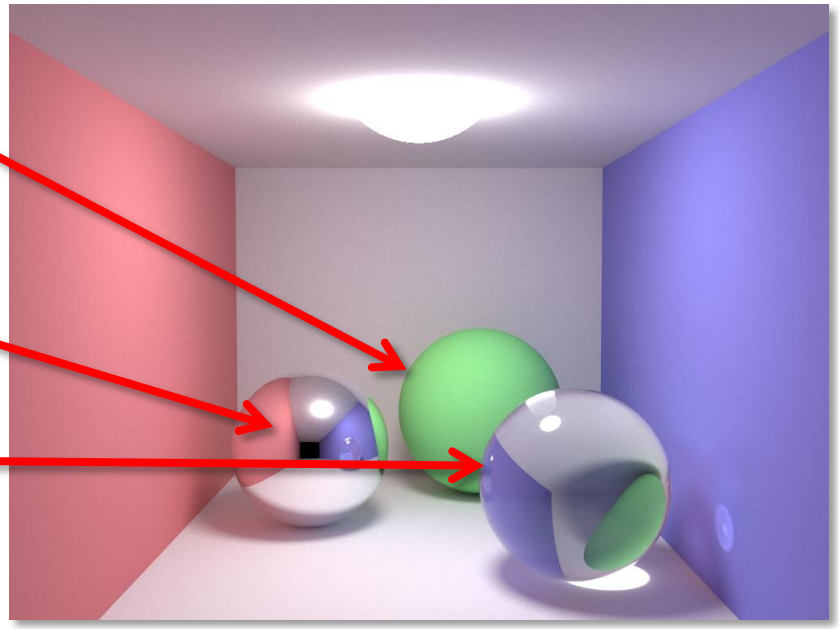
- ロシアンルーレット
  - 無限再帰になるのを防ぐためのロシアンルーレット処理
  - レイが衝突したオブジェクトの反射率の内最大のものをロシアンルーレットの確率にする。
    - 反射率が大きければその後の再帰の重要性も高いだろうから処理を打ち切る確率を下げ、反射率が小さければその後の再帰の重要性も低いだろうから処理を打ち切る確率を上げる。
  - スタックオーバーフロー対策でDepthLimit回以上再帰したら打ち切る確率を上げる。
  - いずれにしても、Depth回の再帰は保障する。

```
// ray方向からの放射輝度を求める
Color radiance(const Ray &ray, Random *rnd, const int depth) {
    Intersection intersection;
    // シーンと交差判定
    if (!intersect_scene(ray, &intersection))
        return kBackgroundColor;

    const Sphere &now_object = spheres[intersection.object_id];
    const Hitpoint &hitpoint = intersection.hitpoint;
    const Vec orienting_normal = dot(hitpoint.normal , 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）
    // 色の反射率最大のものを得る。ロシアンルーレットで使う。
    // ロシアンルーレットの閾値は任意だが色の反射率等を使うとより良い。
    double russian_roulette_probability = std::max(now_object.color.x, std::max(now_object.color.y, now_object.color.z));

    // 反射回数が一定以上になったらロシアンルーレットの確率を急上昇させる。（スタックオーバーフロー対策）
    if (depth > kDpethLimit)
        russian_roulette_probability *= pow(0.5, depth - kDpethLimit);

    // ロシアンルーレットを実行し追跡を打ち切るかどうかを判断する。
    // ただしDepth回の追跡は保障する。
    if (depth > kDepth) {
        if (rnd->next01() >= russian_roulette_probability)
            return now_object.emission;
    } else
        russian_roulette_probability = 1.0; // ロシアンルーレット実行しなかった

    Color incoming_radiance;
    Color weight = 1.0;
```

物体表面の挙動

- **レイが衝突した物体に応じてその後のレイの反射方向を計算する処理が分岐**
  - eduptでは三種類の材質に対応

1. **完全拡散面**
   - Lambertian
2. **完全鏡面**
   - いわゆる鏡
3. **ガラス面**
   - 屈折等

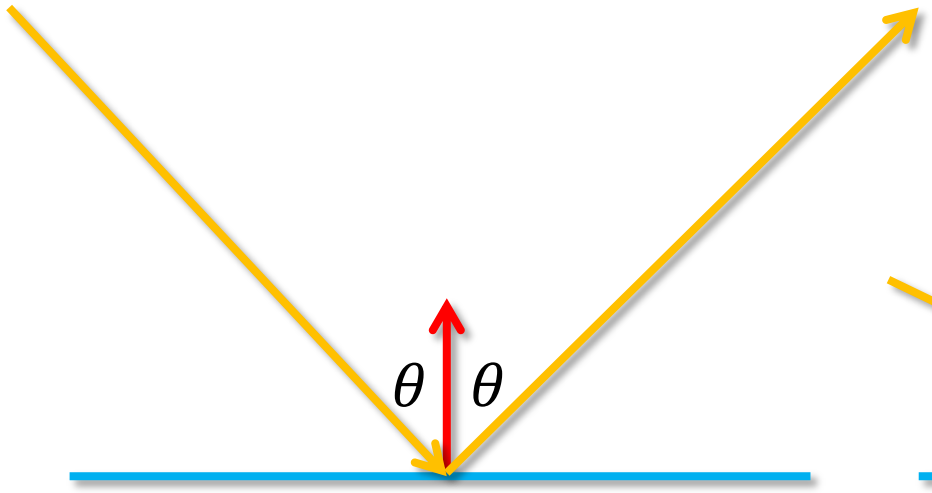


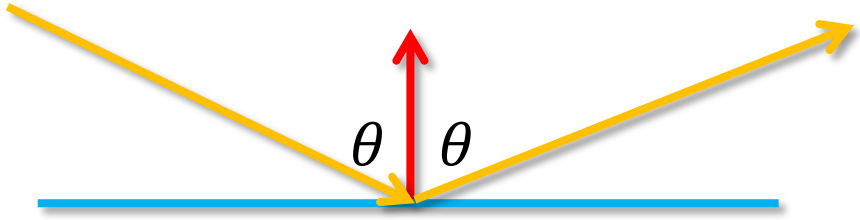
完全拡散面

- **全方向に等しく反射する面**
  - 入射方向に依らない。
  - 拡散反射するときRGBそれぞれ反射する量が違うため物体ごとに異なる色の見え方になる。

完全鏡面

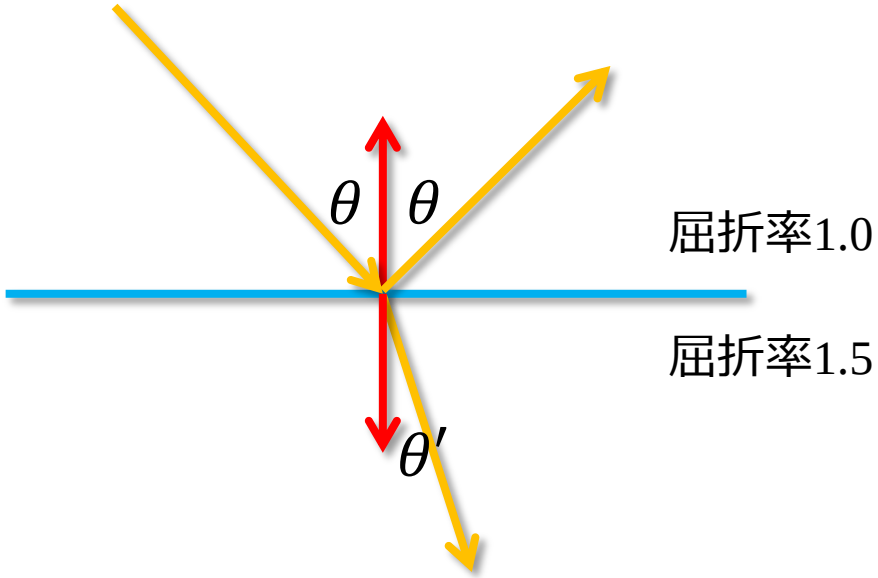
- 入射角と等しい出射角で反射する面





ガラス面

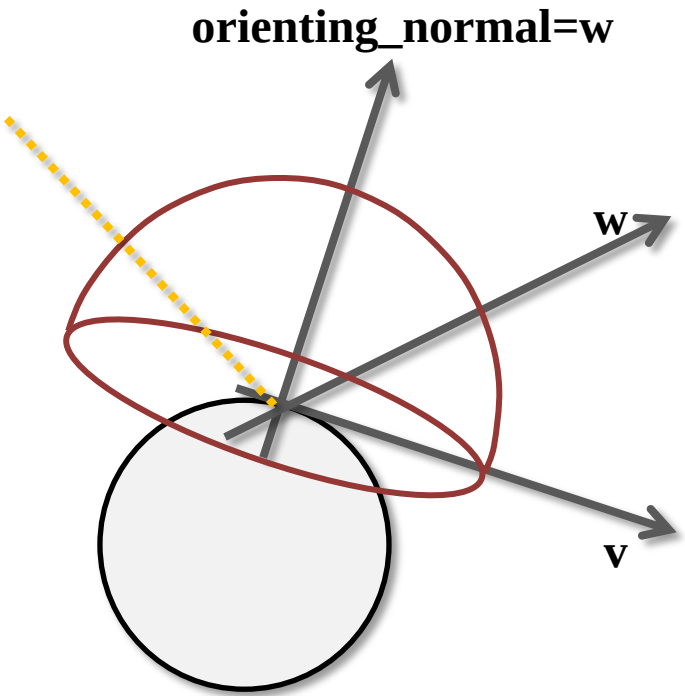
- **物体の屈折率の違いによって光が屈折して物体内部に入り込む面**
  - 物体内部に光が入り込むような面では鏡面反射も起きる。
  - 屈折方向はSnellの法則で求まる。
  - 反射する光の量と屈折する光の量の割合はFresnelの式で求まる。



5.4 完全拡散面

radiance()

- **REFLECTION_TYPE_DIFFUSE**
  - 完全拡散面
  - 次の反射方向をサンプリングする必要がある。
  - orienting_normalの方向を基準として正規直交基底を作り、対応する半球内でサンプリングする。




```
Color incoming_radiance;
Color weight = 1.0;

switch (now_object.reflection_type) {
// 完全拡散面
case REFLECTION_TYPE_DIFFUSE: {
    // orienting_normalの方向を基準とした正規直交基底(w, u, v)を作る。この基底に対する半球内で次のレイを飛ばす。
    Vec w, u, v;
    w = orienting_normal;
    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)));

    incoming_radiance = radiance(Ray(hitpoint.position, dir), rnd, depth+1);
    // レンダリング方程式に対するモンテカルロ積分を考えると、outgoing_radiance = weight * incoming_radiance。
    // ここで、weight = (ρ/π) * cosθ / pdf(ω) / R になる。
    // ρ/πは完全拡散面のBRDFでρは反射率、cosθはレンダリング方程式におけるコサイン項、pdf(ω)はサンプリング方向についての確率密度関数。
    // Rはロシアンルーレットの確率。
    // 今、コサイン項に比例した確率密度関数によるサンプリングを行っているため、pdf(ω) = cosθ/π
    // よって、weight = ρ/ R。
    weight = now_object.color / russian_roulette_probability;
} break;
```

半球内のサンプリング

- **半球内である方向をサンプリングする**
  - サンプリングに使う確率密度関数は何でも良い。
  - 何でも良いが、モンテカルロ積分における被積分関数の形に近い方が分散が小さくなり効率が良くなる。
    - **インポータンスサンプリング**

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

モンテカルロ積分

$$f_r(x', \vec{\omega}, \vec{\omega}')radiance(x', \vec{\omega}')\cos\theta$$

被積分関数

完全拡散面におけるインポータンスサンプリング

- **完全拡散面において**
  - $f_r(x', \vec{\omega}, \vec{\omega}')$、すなわちBRDFは$\frac{\sigma}{\pi}$になる。（$\sigma$は反射率）
  - radiance()は未知。
  - 確率密度関数を$\frac{\cos\theta}{\pi}$として、その確率分布に従って方向をサンプリングすることにする。

完全拡散面におけるインポータンスサンプリング（ただの計算）

- 確率密度関数 $pdf(\theta,\phi)=\frac{\cos\theta}{\pi}$
- 累積分布関数 $F(\theta,\phi)=\frac{1}{\pi}\int\cos\theta\,d\omega$

$$
\begin{aligned}
F(\theta,\phi) &= \frac{1}{\pi}\int_0^{\phi}\int_0^{\theta}\cos\theta'\sin\theta'\,d\theta'\,d\phi' \\
&= \frac{1}{\pi}\int_0^{\phi}d\phi'\int_0^{\theta}\cos\theta'\sin\theta'\,d\theta' \\
&= \frac{\phi}{\pi}\left[-\frac{\cos^2\theta'}{2}\right]_0^{\theta} \\
&= \frac{\phi}{2\pi}(1-\cos^2\theta)
\end{aligned}
$$

$$
F(\phi)=\frac{\phi}{2\pi},F(\theta)=1-\cos^2\theta
$$

完全拡散面におけるインポータンスサンプリング

- 確率密度関数 $pdf(\theta, \phi) = \frac{\cos\theta}{\pi}$
- 上のpdfに従って方向をサンプリングする方法
  - 一つまえのスライドの累積密度関数Fの逆関数を考えると

$$\phi_i = 2\pi u_1$$

$$\theta_i = \cos^{-1}\sqrt{u_2}$$

ただしu_1, u_2は0から1の範囲の乱数

- 上の式に従って乱数を使ってサンプリング、極座標を使って反射方向dirを計算する。

```
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)));
```

完全拡散面におけるインポータンスサンプリング

- 確率密度関数が$\frac{\cos\theta}{\pi}$でBRDFが$\frac{\sigma}{\pi}$なので最終的にradiance()が返すのは

$$L_e(x', \vec{\omega}) + \sigma \cdot radiance(x', \vec{\omega}') \cdot \frac{1}{q}$$

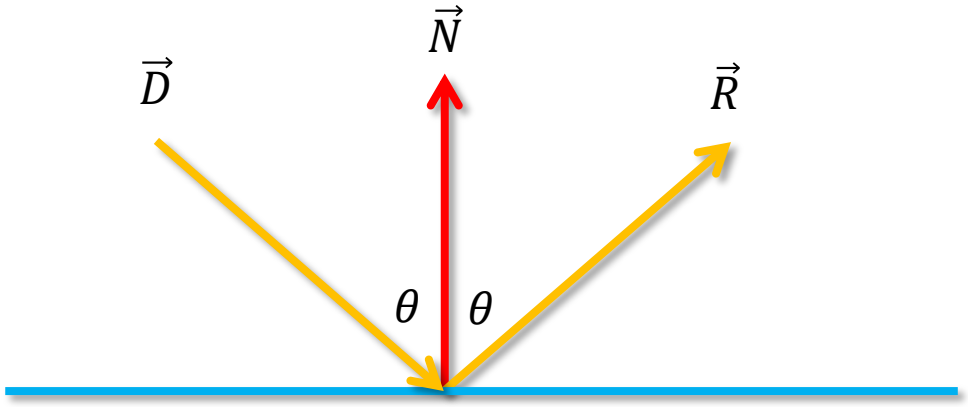
コードだと

```
incoming_radiance = radiance(Ray(hitpoint.position, dir), rnd, depth+1);

        weight = now_object.color / russian_roulette_probability;

    return now_object.emission + multiply(weight, incoming_radiance);
```

5.5 完全鏡面・ガラス面

radiance()

- **REFLECTION_TYPE_SPECULAR**
  - 完全鏡面
  - 次の反射方向は単純に入射角と等しい出射角の方向



$$\vec{R} = \vec{D} - 2(\vec{N} \cdot \vec{D})\vec{N}$$

```
// 完全鏡面
case REFLECTION_TYPE_SPECULAR: {
    // 完全鏡面なのでレイの反射方向は決定的。
    // ロシアンルーレットの確率で除算するのは上と同じ。
    incoming_radiance = radiance(Ray(hitpoint.position, ray.dir - hitpoint.normal * 2.0 * dot(hitpoint.normal, ray.dir)), rnd, depth+1);
    weight = now_object.color / russian_roulette_probability;
} break;
```

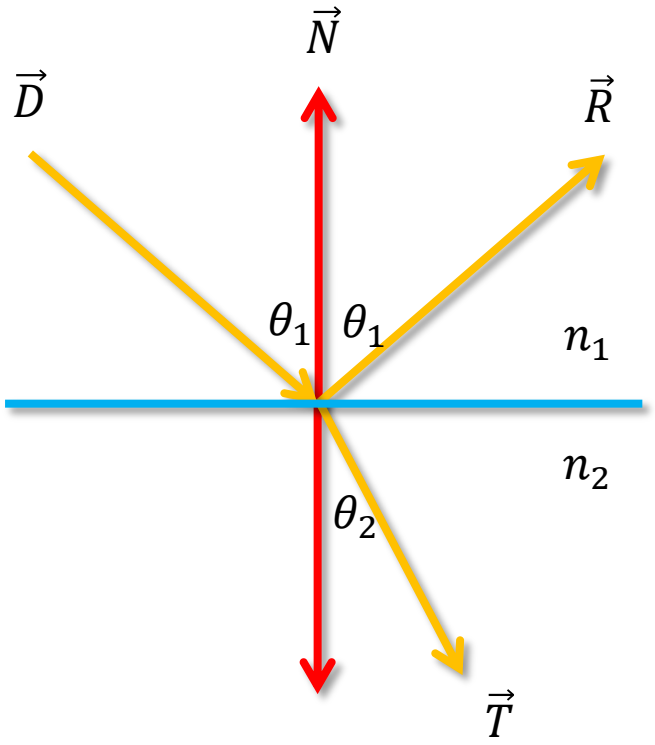
radiance()

- **REFLECTION_TYPE_REFRACTION**
  - ガラス面
  - 屈折率は material.hのkIor
- **まずSnellの法則によって屈折する方向を求める**
  - 屈折する方向によっては屈折が起こらず、入射した光が全て反射される全反射が起こる。（その場合は完全鏡面と同じ処理になる）

```
// 屈折率kIorのガラス
case REFLECTION_TYPE_REFRACTION: {
    const Ray reflection_ray = Ray(hitpoint.position, ray.dir - hitpoint.normal * 2.0 * dot(hitpoint.normal, ray.dir));
    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

    // Snellの法則
    const double nc = 1.0; // 真空の屈折率
    const double nt = kIor; // オブジェクトの屈折率
    const double nnt = into ? nc / nt : nt / nc;
    const double ddn = dot(ray.dir, orienting_normal);
    const double cos2t = 1.0 - nnt * nnt * (1.0 - ddn * ddn);

    if (cos2t < 0.0) { // 全反射
        incoming_radiance = radiance(reflection_ray, rnd, depth+1);
        weight = now_object.color / russian_roulette_probability;
        break;
    }
    // 屈折の方向
    const Ray refraction_ray = Ray(hitpoint.position,
        normalize(ray.dir * nnt - hitpoint.normal * (into ? 1.0 : -1.0) * (ddn * nnt + sqrt(cos2t))));
```

Snellの法則

- 入射前の物体の屈折率をn_1、入射後の物体の屈折率をn_2とする。



$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \quad \text{(Snellの法則)}$$

$$\cos^2\theta_2 = 1 - \left(\frac{n_1}{n_2}\right)^2 \left(1 - \left(-\vec{D}\cdot\vec{N}\right)^2\right)$$

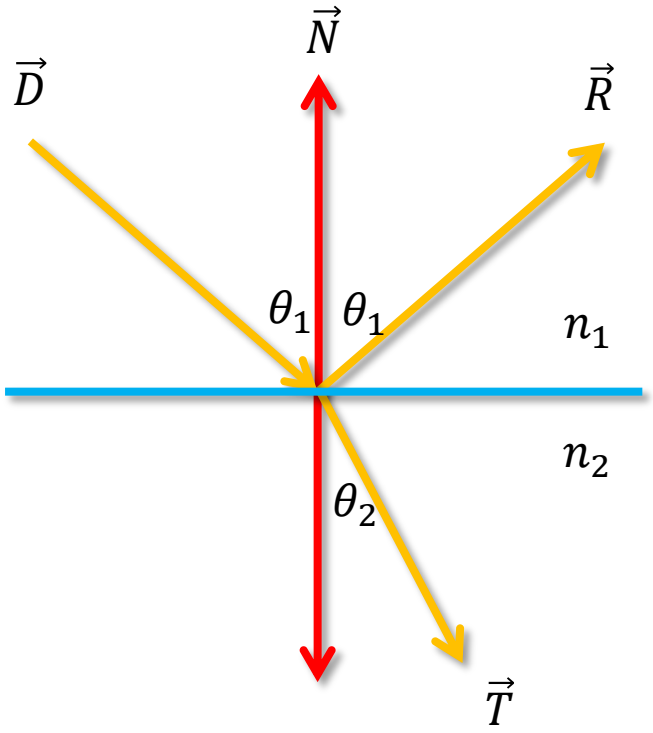
右辺が0未満になったとき、θ_2が90度を超えたということで、全反射。

```
// Snellの法則
const double nc = 1.0; // 真空の屈折率
const double nt = kIor; // オブジェクトの屈折率
const double nnt = into ? nc / nt : nt / nc;
const double ddn = dot(ray.dir, orienting_normal);
const double cos2t = 1.0 - nnt * nnt * (1.0 - ddn * ddn);

if (cos2t < 0.0) { // 全反射
    incoming_radiance = radiance(reflection_ray, rnd, depth+1);
    weight = now_object.color / russian_roulette_probability;
    break;
}
```

Snellの法則

- 屈折の方向

```
// 屈折の方向
const Ray refraction_ray = Ray(hitpoint.position,
    normalize(ray.dir * nnt - hitpoint.normal * (into ? 1.0 : -1.0) * (ddn * nnt + sqrt(cos2t))));
```



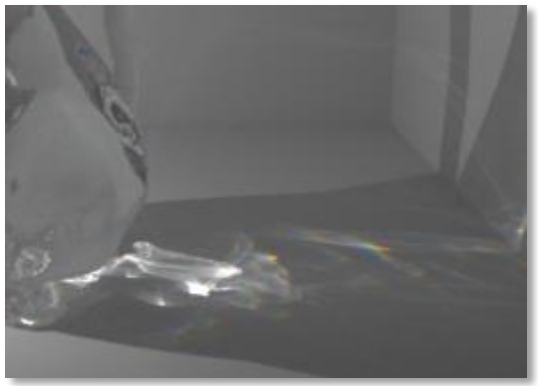
$$\vec{T} = \frac{n_1}{n_2}\vec{D} - \left(\frac{n_1}{n_2}\left(\vec{D}\cdot\vec{N}\right) + \sqrt{\cos^2\theta_2}\right)\vec{N}$$

屈折率

- **真空中の光の速度を物質中の光の速度で割った値**
  - 速度が変化することで光は屈折する。
  - http://refractiveindex.info/ にたくさん。

水：1.33
水晶：1.54
光学ガラス：1.43-2.14
ダイヤモンド：2.42
サファイア：1.77

- **波長によって屈折率は違う**
  - 光の分散

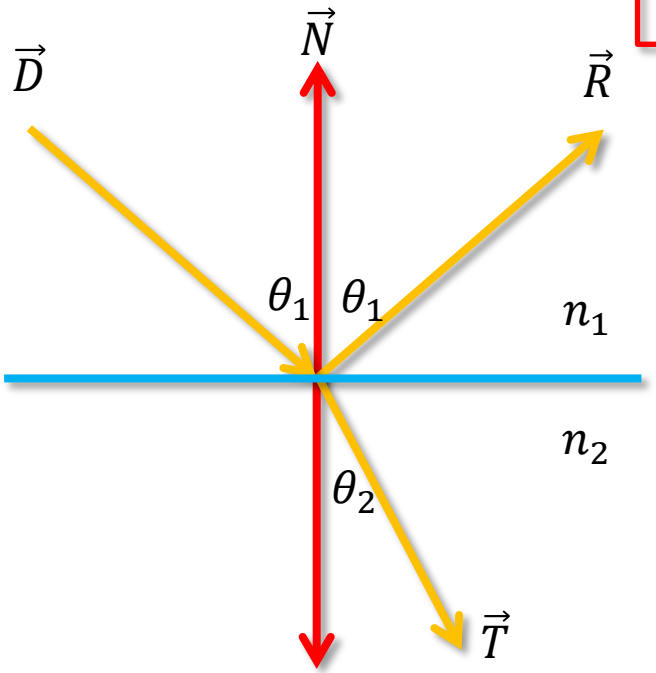


Fresnelの式

- 反射光の運ぶ光の割合はFresnelの式に従う
- Schlickの近似が良く使われる
  - 入射光に対する反射光の運ぶ光の割合$F_r(\theta)$について入射角度の関数として近似。

$$F_r(\theta_1) = F_0 + (1 - F_0)(1 - \cos\theta_1)^5$$

$n_1 > n_2$のときは$F_r(\theta_1) = F_0 + (1 - F_0)(1 - \cos\theta_2)^5$



F_0は垂直入射における反射の量で

$$F_0 = \frac{(n_1 - n_2)^2}{(n_1 + n_2)^2}$$

Fresnelの式

- 反射光の運ぶ光の割合

$$F_r(\theta_1) = F_0 + (1 - F_0)(1 - \cos\theta_1)^5$$

$n_1 > n_2$のときは$F_r(\theta_1) = F_0 + (1 - F_0)(1 - \cos\theta_2)^5$

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

const double c = 1.0 - (into ? -ddn : dot(refraction_ray.dir, -1.0 * orienting_normal));
const double Re = 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。
const double nnt2 = pow(into ? nc / nt : nt / nc, 2.0); // 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 Tr = (1.0 - Re) * nnt2; // 屈折方向の光が屈折してray.dirの方向に運ぶ割合
```

Fresnelの式

- **屈折光の運ぶ光の割合**
  - 基本的には$1 - F_r(\theta_1)$
- **レイが運ぶのは放射輝度**
  - 放射輝度は屈折の前後で値が変化する。
  - 放射輝度が単位立体角あたりの値なのが理由。

$$\text{屈折直後の放射輝度} = \left(\frac{n_2}{n_1}\right)^2 \text{屈折直前の放射輝度}$$

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

const double c = 1.0 - (into ? -ddn : dot(refraction_ray.dir, -1.0 * orienting_normal));
const double Re = 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。
const double nnt2 = pow(into ? nc / nt : nt / nc, 2.0); // 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 Tr = (1.0 - Re) * nnt2; // 屈折方向の光が屈折してray.dirの方向に運ぶ割合
```

ロシアンルーレット

- **反射方向と屈折方向の両方から来る放射輝度を計算する必要がある**
  - 両方に対してradiance()を実行するとレイが指数的に増える→再帰深度が一定以上ならロシアンルーレットを使って増加を抑制。
  - ロシアンルーレットの確率は反射光の運ぶ光の割合に基づいて適当に決める。

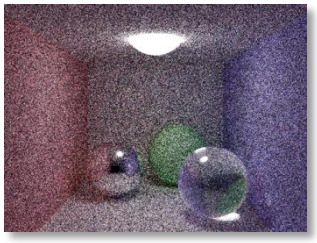
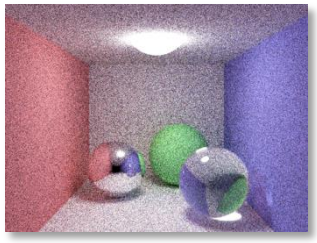
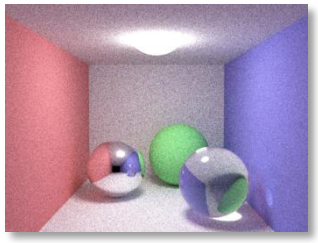
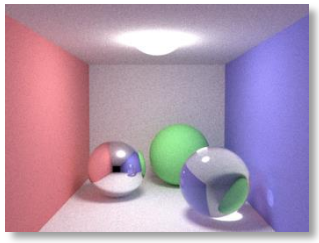
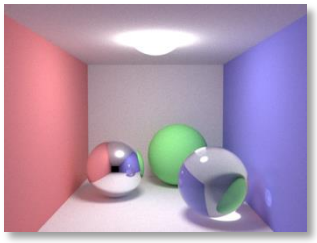
```
// 一定以上レイを追跡したら屈折と反射のどちらか一方を追跡する。（さもないと指数的にレイが増える）
// ロシアンルーレットで決定する。
const double probability  = 0.25 + 0.5 * Re;
if (depth > 2) {
    if (rnd->next01() < probability) { // 反射
        incoming_radiance = radiance(reflection_ray, rnd, depth+1) * Re;
        weight = now_object.color / (probability * russian_roulette_probability);
    } else { // 屈折
        incoming_radiance = radiance(refraction_ray, rnd, depth+1) * Tr;
        weight = now_object.color / ((1.0 - probability) * russian_roulette_probability);
    }
} else { // 屈折と反射の両方を追跡
    incoming_radiance =
        radiance(reflection_ray, rnd, depth+1) * Re +
        radiance(refraction_ray, rnd, depth+1) * Tr;
    weight = now_object.color / (russian_roulette_probability);
}
```

通常時

- **深度が一定数以下なら普通にレイを追跡**
  - 反射方向、屈折方向のそれぞれから来る放射輝度を計算するためにそれぞれの方向にradiance()を実行。
  - Fresnelの式よりすでに計算済みの反射光の運ぶ光の割合と屈折光の運ぶ光の割合をそれぞれに乗算する。

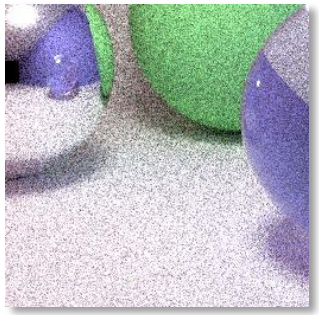
```
// 一定以上レイを追跡したら屈折と反射のどちらか一方を追跡する。（さもないと指数的にレイが増える）
// ロシアンルーレットで決定する。
const double probability  = 0.25 + 0.5 * Re;
if (depth > 2) {
    if (rnd->next01() < probability) { // 反射
        incoming_radiance = radiance(reflection_ray, rnd, depth+1) * Re;
        weight = now_object.color / (probability * russian_roulette_probability);
    } else { // 屈折
        incoming_radiance = radiance(refraction_ray, rnd, depth+1) * Tr;
        weight = now_object.color / ((1.0 - probability) * russian_roulette_probability);
    }
} else { // 屈折と反射の両方を追跡
    incoming_radiance =
        radiance(reflection_ray, rnd, depth+1) * Re +
        radiance(refraction_ray, rnd, depth+1) * Tr;
    weight = now_object.color / (russian_roulette_probability);
}
```

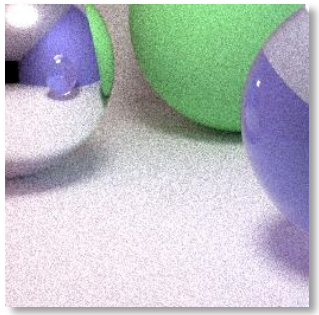
収束について

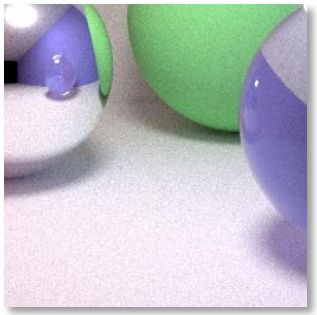







16 sample/pixel
13秒


64 sample/pixel
44秒


256 sample/pixel
174秒


1024 sample/pixel
690秒


4096 sample/pixel
2764秒

Intel Core i7 980 (3.33GHz) 10スレッド

参考文献

- Kevin Beason “smallpt http://www.kevinbeason.com/smallpt/”
- David Cline “smallpt presentation http://www.kevinbeason.com/smallpt/#moreinfo”
- Henrik Wann Jensen (著), 苗村 健 (翻訳) “フォトンマッピング―実写に迫るコンピュータグラフィックス“
- 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”
- Matt Pharr “Image Synthesis (Stanford cs348b) http://candela.stanford.edu/”