レイトレーシング入門

mebiusbox

2022 年 12 月 25 日

目次

第 1 章 光線の基本と反射 3

1 はじめに . . . 3
2 ソースコード . . . 4
3 画像出力 . . . 4
4 レイトレーシングの原理 . . . 6
5 基本クラス . . . 8
6 球の追加 . . . 12
7 シーンクラス . . . 15
8 複数の物体への対応 . . . 18
9 アンチエイリアシング . . . 22
10 拡散反射 . . . 24
11 材質 . . . 28
12 鏡面反射 . . . 31
13 屈折 . . . 35
14 沢山表示してみる . . . 39

第 2 章 テクスチャとコーネルボックス 41

1 テクスチャ . . . 41
2 画像テクスチャ . . . 44
3 発光 . . . 47
4 四角形 . . . 48
5 コーネルボックス . . . 52

第 3 章 モンテカルロレイトレーシング 60

1 光学 . . . 60
2 光の物理量 . . . 60
4 モンテカルロレイトレーシング . . . 60
5 光の散乱 . . . 61
5 重点的サンプリング (Importance sampling) . . . 62
6 この次はどうするか . . . 84
7 参考となる資料 . . . 85

第 1 章

光線の基本と反射

1. はじめに

コンピュータグラフィックをレンダリングするとき, リアルタイムレンダリングかどうかでレンダリングする手法は基本的に変わります.非リアルタイムレンダリングではレイトレーシングが現在でも主流で, 様々な手法が開発されたり改良されています.レイトレーシング法の中でもモンテカルロレイトレーシングはランダムにレイを反射させて光伝達を追跡し, それ以外の光の物理現象を正確に表現することができます.ただし, サンプリング数が多くないと計算結果の精度が良くないという問題を抱えています.それについても様々な改良方法が研究されています.また, リアルタイムにおいてもレイマーチングという手法でレンダリングが行われることも活発になってきており, ゲームなどのリアルタイムレンダリングでも活用する機会が増えてきています.

本記事では, モンテカルロレイトレーシングについて解説します.内容としては Peter Shirley 著の3冊

- Ray Tracing in One Weekend
- Ray Tracing: the Next Week
- Ray Tracing: The Rest Of Your Life

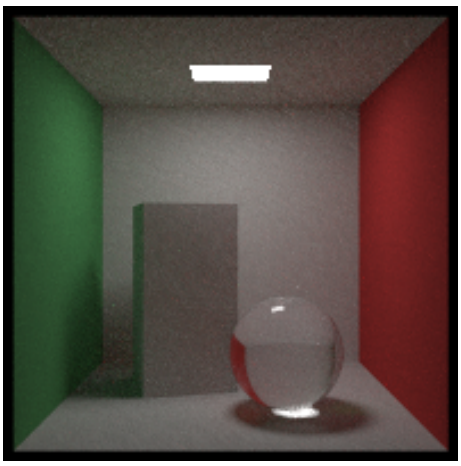
を基本としています.ただし, 全ての内容は含まれておらず, またコードの設計などには若干手を入れています.関与媒質, BVH, ノイズ, モーションブラー, 被写界深度といった内容はここでは扱いませんので, 興味があれば読んでみてください.なんと現在は無料で公開されています.

github/petershirley

すでに多くのレイトレーシングレンダラーのソースコードが公開されていたり, 資料が公開されています.最後にいくつか紹介しますので, そちらも興味があれば参照してみてください.

コードは C++ で書いています. STL や c++11 (スマートポインタなど)など使いますので, ある程度言語について知っていることが前提となります.といってもそんなに多くはないので, わからない機能がありましたら, ネットで検索すれば情報は出てくるはずです.名前空間は特別なものを覗いて明示的に記載します.

今回の目標は次のような画像をレンダリングすることです.



2. ソースコード

本記事のソースコードは以下の場所にあります.

rayt (https://github.com/mebiusbox/rayt)

Windows で, VisualStudio 2015 Community を使用しています.また, プログラミング言語は C++ です.

`rayt.cpp` ファイルがメインファイルになっています.ソースコードには `rayt101.cpp` という様に連番がついたファイルがあります.これはその内容を全て `rayt.cpp` に上書きするようになっていますので, 試すときはそのようにしてください.レンダリング画像を掲示している場所には, その画像を生成したソースコードファイル名を一緒に明記するようにしています.

3. 画像出力

まずは, レンダリングした画像をファイルに保存できるような機能を作ります.この機能がないと何も始まりません.

コードは `rayt.h` と `rayt.cpp` に書いていきます.通常はクラスごとにファイルを分割した方がよいですが, 今回はこの2ファイルに集約しました.ユーティリティ関数や一度実装した後に変更のないクラスは `rayt.h` に記載するようにします.本文中でコードのところに `// rayt.h` と記載があれば `rayt.h` に実装し, 記載がない場合は `rayt.cpp` に実装することを表しています.あと, 基本的に名前空間 `rayt` 内に定義するようにしています.

まずは `Image` クラスを作成します.これはスクリーンのカラー値を格納する領域を管理し, そこに書き込む機能を持っています.

```
// rayt.h
#include <memory>
#include <iostream>

namespace rayt {

    class Image {
    public:
        struct rgb {
            unsigned char r;
            unsigned char g;
            unsigned char b;
        };

        Image() : m_pixels(nullptr) { }
```

```
        Image(int w, int h) {
            m_width = w;
            m_height = h;
            m_pixels.reset(new rgb[m_width*m_height]);
        }

        int width() const { return m_width; }
        int height() const { return m_height; }
        void* pixels() const { return m_pixels.get(); }

        void write(int x, int y, float r, float g, float b) {
            int index = m_width*y + x;
            m_pixels[index].r = static_cast<unsigned char>(r*255.99f);
            m_pixels[index].g = static_cast<unsigned char>(g*255.99f);
            m_pixels[index].b = static_cast<unsigned char>(b*255.99f);
        }

    private:
        int m_width;
        int m_height;
        std::unique_ptr<rgb[]> m_pixels;
    };
} // namespace rayt
```

スマートポインタ `std::unique_ptr` を使用するためには `memory` ヘッダーファイルをインクルードする必要があります.また,コンソール出力を使いますので, `iostream` もインクルードします.

次にメイン関数を実装します.今回は適当にグラデーションの画像を作成します. `Image` クラスのインスタンスを作成して,書き込みます.

```
#include "rayt.h"

int main()
{
    int nx = 200;
    int ny = 100;
    std::unique_ptr<rayt::Image> image(std::make_unique<rayt::Image>(nx, ny));

    for (int j = 0; j<ny; ++j) {
        std::cerr << "Rendering (y = " << j << ") " << (100.0 * j / (ny - 1)) << "%" << std::endl;
        for (int i = 0; i<nx; ++i) {
            float r = float(i) / float(nx);
            float g = float(j) / float(ny);
            float b = 0.5f;
            image->write(i, j, r, g, b);
        }
    }
```

```
    return 0;
}
```

次に，`Image` クラスに書き込んだカラー値を画像に出力します.出力には stb ライブラリを使用します. stb ライブラリはパブリックドメインライブラリです.とても手軽に扱えて便利です.

http://nothings.org/stb/

この中の `stb_image.h` と `stb_image_write.h` を使います.

```
// rayt.h
#define STB_IMAGE_IMPLEMENTATION
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image.h"
#include "stb_image_write.h"
```

`STB_IMAGE_IMPLEMENTATION` と `STB_IMAGE_WRITE_IMPLEMENTATION` は `stb_image.h` と `stb_image_write.h` の前で定義する必要があります.

実際にファイルに出力するには以下のコードを使います.

```
stbi_write_bmp("render.bmp", nx, ny, sizeof(rayt::Image::rgb), image->pixels());
```

これはビットマップ形式で `render.bmp` というファイル名で出力します.これを実行すると以下の画像が得られます. (rayt101.cpp, rayt101.h)

4. レイトレーシングの原理

レイトレーシングはまずカメラから光線を飛ばします.飛んでいる光線が何かに当たったら光線は反射します.反射する方向は当たった物体の表面の材質によって変わります.光線は何度か反射を繰り返し, 最終的に光源に辿り着きます.この光源から放射された光をカメラまで輸送するシミュレーションを行い, 人間の眼で正しく認識できるように変換して画像ファイルとして出力します.

これから実装するために, いくつか定義やユーティリティ関数, ベクトルクラス, 基本クラスなど追加していきます.

4.1 Vectormath

ベクトル計算に Vectormath ライブラリを使用します.

Vectormath

Vectormath ライブラリは Sony が開発したライブラリでソースが公開され, 物理エンジン Bullet で使われています. 3D グラフィックス用のベクトル計算ライブラリということもあって, ベクトルや行列, クォータニオンを扱う十分な機能が含まれています.

名前空間 `Vectormath` に定義されていて長いので，`using` マクロで宣言し，よく使うベクトルクラスは `typedef` で短い名前で再定義しています．

インクルードするディレクトリに `vectormath/include` を追加する必要があります．

```
// rayt.h
#include <vectormath/scalar/cpp/vectormath_aos.h>
using namespace Vectormath::Aos;
typedef Vector3 vec3;
typedef Vector3 col3;
```

C++ のクラスなので，四則演算等は演算子が定義されているので直感的に操作できます．ただし，内積や外積などはメンバ関数ではなく，グローバルな関数として用意されています．よく使う関数を解説します．

関数名	説明
vec3 dot(a,b)	内積を求めます
vec3 cross(a,b)	外積を求めます
float length(v)	ベクトルの長さを求めます
float lengthSqr(v)	ベクトルの長さの二乗を求めます
vec3 normalize(v)	ベクトルを正規化します
vec3 mulPerElem(a,b)	ベクトルの各要素ごとに乗算します
vec3 maxPerElem(a,b)	2つのベクトルから各要素ごとに最大であるベクトルを返します
vec3 minPerElem(a,b)	2つのベクトルから各要素ごとに最小であるベクトルを返します
vec3 lerp(t,a,b)	2つのベクトルをパラメータ t で線形補間したベクトルを返します

4.2 数学の定数

数学の定数をいくつか定義しています．

```
// rayt.h
#include <float.h>  // FLT_MIN, FLT_MAX
#define PI 3.14159265359f
#define PI2 6.28318530718f
#define RECIP_PI 0.31830988618f
#define RECIP_PI2 0.15915494f
#define LOG2 1.442695f
#define EPSILON 1e-6f
#define GAMMA_FACTOR 2.2f
```

4.3 ユーティリティ関数

また，ユーティリティ関数をいくつか定義しています．

```
// rayt.h
#include <random>
inline float drand48() {
    return float(((double)(rand()) / (RAND_MAX))); /* RAND_MAX = 32767 */
}

inline float pow2(float x) { return x*x; }
inline float pow3(float x) { return x*x*x; }
inline float pow4(float x) { return x*x*x*x; }
inline float pow5(float x) { return x*x*x*x*x; }
inline float clamp(float x, float a, float b) { return x < a ? a : x > b ? b : x; }
inline float saturate(float x) { return x < 0.f ? 0.f : x > 1.f ? 1.f : x; }
inline float recip(float x) { return 1.f / x; }
inline float mix(float a, float b, float t) { return a*(1.f - t) + b*t; /* return a + (b-a) * t; */ }
inline float step(float edge, float x) { return (x < edge) ? 0.f : 1.f; }
inline float smoothstep(float a, float b, float t) { if (a >= b) return 0.f; float x = saturate((t - a) / (b - a));
 return x*x*(3.f - 2.f * t); }
inline float radians(float deg) { return (deg / 180.f)*PI; }
inline float degrees(float rad) { return (rad / PI) * 180.f; }
```

以下にリファレンスを記載します.

関数名	説明
float drand48()	[0,1] の一様乱数を返します
float pow2(x)	x の自乗を求めます
float pow3(x)	x の3乗を求めます
float pow4(x)	x の4乗を求めます
float pow5(x)	x の5乗を求めます
float clamp(x,a,b)	$a \leq x \leq b$ を満たす x を返します
float saturate(x)	$0 \leq x \leq 1$ を満たす x を返します
float recip(x)	$1/x$ を返します
float mix(a,b,t)	a と b をパラメータ t で線形補間した値を返します
float step(edge,x)	edge 以下のとき 0 を, それ以外のときは 1 を返します
float smoothstep(a,b,t)	a と b をパラメータ t でスプライン補間した値を返します
float radians(deg)	度数からラジアンに変換します
float degrees(rad)	ラジアンから度数に変換します

5. 基本クラス

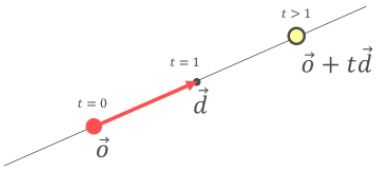
5.1 光線

光線は始点と向きを持っています.コードは次の通りです.

```
// rayt.h
class Ray {
    public:
        Ray() {}
        Ray(const vec3& o, const vec3& dir)
            : m_origin(o)
            , m_direction(dir) {
        }

        const vec3& origin() const { return m_origin; }
        const vec3& direction() const { return m_direction; }
        vec3 at(float t) const { return m_origin + t*m_direction; }

    private:
        vec3 m_origin;    // 始点
        vec3 m_direction; // 方向（非正規化）
    };
```

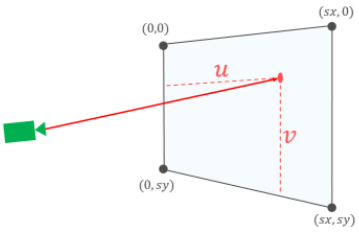
`at` 関数は，始点 $\vec{o}$ から方向 $\vec{d}$ に向かう直線上の任意の点 $\vec{p}$ を，パラメータ t を指定して求めます．$t=0$ のときは始点，$t=1$ のときは始点から方向ベクトル $\vec{d}$ 進んだ位置，$t>1$ なら方向ベクトルより先の位置，$t<0$ なら，始点から反対方向に向かう直線上の位置を取得できます．式で表すと

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



5.2 カメラ

レイトレーシングでは投影するスクリーン上のピクセルごとに光線を飛ばします．光線はカメラの位置から発生し，カメラの向きに飛んでいきます．投影するスクリーンの大きさを sx, sy とし，各ピクセルへの光線はスクリーン上の位置 u, v から算出します．



カメラは投影するスクリーンの X 軸と Y 軸，そしてカメラの向きを Z 軸とする直交基底ベクトルを持っています．この基底ベクトルを使って u, v を基底変換します．

光線を飛ばすピクセルの位置が si, sj とすると,

$$u = \frac{si}{sx}, \quad v = \frac{sj}{sy} \qquad (0 \leq u \leq 1, 0 \leq v \leq 1).$$

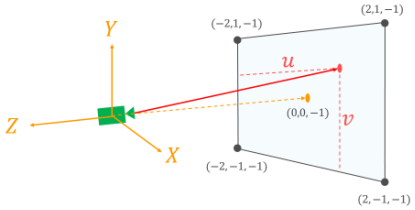
基底ベクトル $\vec{u}, \vec{v}, \vec{w}$ を使ってこれを基底変換すると, 投影するスクリーン上の位置 $\vec{p}$ は

$$\vec{p} = \vec{u} \cdot u + \vec{v} \cdot v + \vec{w}.$$

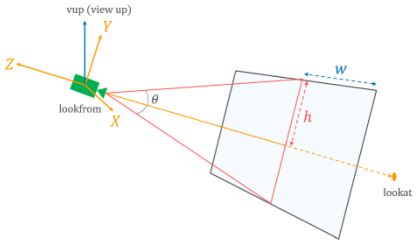
カメラの位置を原点に置き, -z 方向を向いているとします.スクリーンの右上が $(2, 1, -1)$ となるような基底ベクトルは次のようになります.

$$\vec{u} = (4, 0, 0), \quad \vec{v} = (0, 2, 0), \quad \vec{w} = (-2, -1, -1).$$

下図はこの関係を表したものです.



カメラの指定には便利な LookAt 方式があります.カメラの位置 lookfrom と視線対象の位置 lookat, カメラの上方向を表す vup から基底ベクトルを計算することができます.



まず, カメラにおける正規直交基底ベクトル XYZ を計算します. Z は lookat-lookfrom です.次に X は上方向を表す vup と求めた Z の外積です.そして, Y は X と Z の外積となります.

この正規直交基底ベクトルから, 求めたい基底ベクトルを導出します.まず, スクリーン上の高さ h はカメラからスクリーンまでの距離を 1 とすると $h = \tan(\theta/2)$ です.幅 w はスクリーンのアスペクト比$(aspect = sx/sh)$を使って $w = aspect \cdot h$ です.

基底ベクトル $\vec{u}$ と $\vec{v}$ はそれぞれ正規直交ベクトルの XY に対応しているので

$$\vec{u} = 2wX, \quad \vec{v} = 2hY.$$

次に $\vec{w}$ です.スクリーン上の位置 $\vec{p}$ を基底ベクトルから求めるときは以下の式を使いました.

$$\vec{p} = \vec{u} \cdot u + \vec{v} \cdot v + \vec{w}.$$

この式から $\vec{w}$ を求めると

$$\vec{w} = \vec{p} - \vec{u} \cdot u - \vec{v} \cdot v.$$

今，$\vec{p} = \vec{o} - Z$ ($\vec{o}$ はカメラの位置)なので

$$\vec{w} = \vec{o} - wX - hY - Z.$$

`getRay` 関数では，u, v から光線を生成します.基底変換で得られた位置からカメラの位置との差が方向ベクトルとなります.

以下がカメラクラスのコードです.コンストラクタは基底ベクトルを直接指定するのと，LookAt 方式で指定するものがあります.

```
// rayt.h
class Camera {
public:
    Camera() {}
    Camera(const vec3& u, const vec3& v, const vec3& w) {
        m_origin = vec3(0);
        m_uvw[0] = u;
        m_uvw[1] = v;
        m_uvw[2] = w;
    }
    Camera(const vec3& lookfrom, const vec3& lookat, const vec3& vup, float vfov, float aspect) {
        vec3 u, v, w;
        float halfH = tanf(radians(vfov)/2.0f);
        float halfW = aspect * halfH;
        m_origin = lookfrom;
        w = normalize(lookfrom - lookat);
        u = normalize(cross(vup, w));
        v = cross(w, u);
        m_uvw[2] = m_origin - halfW*u - halfH*v - w;
        m_uvw[0] = 2.0f * halfW * u;
        m_uvw[1] = 2.0f * halfH * v;
    }

    Ray getRay(float u, float v) const {
        return Ray(m_origin, m_uvw[2] + m_uvw[0]*u + m_uvw[1]*v - m_origin);
    }

private:
    vec3 m_origin;  // 位置
    vec3 m_uvw[3];  // 直交基底ベクトル
};
```

このカメラを使ってみます.カメラの基底ベクトルを指定し，各ピクセルに向かう光線を作成します.その光線の方向ベクトルを正規化して，そのベクトルの y を使って2色のグラデーションを描きます.

```
vec3 color(const rayt::Ray& r) {
    vec3 d = normalize(r.direction());
    float t = 0.5f*(r.direction().getY() + 1.0f);
    return lerp(t, vec3(0.5f, 0.7f, 1.0f), vec3(1));
```

```
}

int main()
{
    int nx = 200;
    int ny = 100;
    std::unique_ptr<rayt::Image> image(std::make_unique<rayt::Image>(nx, ny));

    vec3 x(4.0f, 0.0f, 0.0f);
    vec3 y(0.0f, 2.0f, 0.0f);
    vec3 z(-2.0f, -1.0f, -1.0f);
    std::unique_ptr<rayt::Camera> camera(std::make_unique<rayt::Camera>(x, y, z));

    for (int j = 0; j<ny; ++j) {
        std::cerr << "Rendering (y = " << j << ") " << (100.0 * j / (ny - 1)) << "%" << std::endl;
        for (int i = 0; i<nx; ++i) {
            float u = float(i) / float(nx);
            float v = float(j) / float(ny);
            rayt::Ray r = camera->getRay(u, v);
            vec3 c = color(r);
            image->write(i, j, c.getX(), c.getY(), c.getZ());
        }
    }

    stbi_write_bmp("render.bmp", nx, ny, sizeof(rayt::Image::rgb), image->pixels());
    return 0;
}
```

実行すると次のような画像になります. (rayt102.cpp, rayt102.h)

6. 球の追加

これでは寂しいので物体を追加しましょう.ここでは「球」を追加します.球はとても扱いやすい形状なので, いろんなところで使われます.

球は中心を原点とすると半径 r を使って

$$x^2 + y^2 + z^2 = r^2,$$

という式で表されます.中心位置を cx, cy, cz とすると

$$(x - cx)^2 + (y - cy)^2 + (z - cz)^2 = r^2.$$

ベクトルを使うと, 中心位置が $\vec{c}$ で位置が $\vec{p}$ とすると

$$(\vec{p}-\vec{c})\cdot(\vec{p}-\vec{c})=r^2.$$

レイトレーシングでは光線と物体の衝突した位置が知りたいので, 光線の方程式

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

を代入すると

$$(\vec{p}(t)-\vec{c})\cdot(\vec{p}(t)-\vec{c})=r^2.$$

つまり

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

ここで $\vec{oc}=\vec{o}-\vec{c}$ とすると

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

展開すると

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

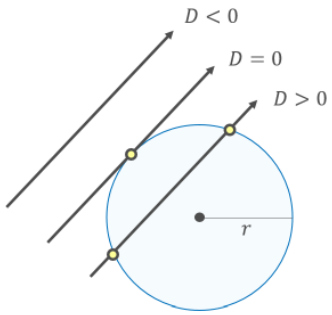
ここで「2次方程式の解の公式」を使います.2次方程式 $ax^2+bx+c=0$ の解は

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}.$$

$\vec{oc}$ を展開して, $ax^2+bx+c=0$ に当てはめてみると

$$\begin{aligned}a&=(\vec{d}\cdot\vec{d})\\b&=2(\vec{d}\cdot(\vec{o}-\vec{c}))\\c&=((\vec{o}-\vec{c})\cdot(\vec{o}-\vec{c}))-r^2,\end{aligned}$$

となります.ここで, $D=b^2-4ac$ というのは判別式といい, 解がいくつあるかがわかります. $D>0$ なら, 2つの解が存在します. $D=0$ なら 1つの解が存在し, $x=-\frac{b}{2a}$ で与えられます.残りの $D<0$ では解は存在しません.
これは図で表すと



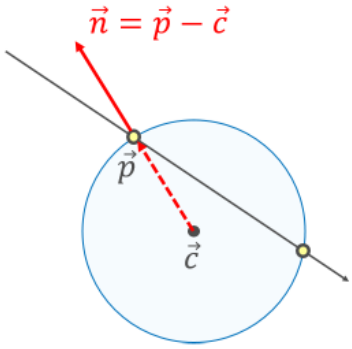
光線と球が衝突したかどうかは判別式のみで十分です.それでは球をレンダリングしてみましょう. `hit_sphere` 関数を追加し,球との衝突処理を実装しています.そして `color` 関数に球との衝突コードを追加しています.

```
bool hit_sphere(const vec3& center, float radius, const rayt::Ray& r) {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = 2.0f * dot(r.direction(), oc);
    float c = dot(oc,oc) - pow2(radius);
    float D = b*b-4*a*c;
    return (D > 0);
}

vec3 color(const rayt::Ray& r) {
    if (hit_sphere(vec3(0,0,-1), 0.5f, r)) {
        return vec3(1.0f, 0.0f, 0.0f);
    }
    vec3 d = normalize(r.direction());
    float t = 0.5f*(r.direction().getY() + 1.0f);
    return lerp(t, vec3(0.5f, 0.7f, 1.0f), vec3(1));
}
```

これは次のような画像になります. (rayt103.cpp)

6.1 球体の法線

光を反射させるには衝突した位置の法線が必要です.法線は衝突した位置と球体の中心位置の差で求まります.

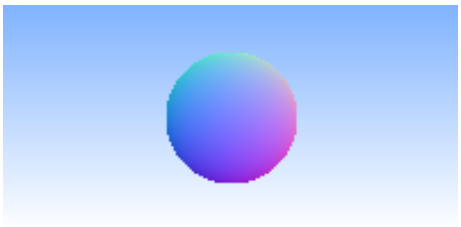


衝突したら,位置を算出する必要があるので解を求めます.正規化した法線は各要素が [-1,1] の範囲になるため, RGB の範囲 [0-1] に変換する必要があります.それは 1 を足した後に 0.5 を掛けます.

`hit_sphere` と `color` を次のように書き換えます.

```
float hit_sphere(const vec3& center, float radius, const rayt::Ray& r) {
    vec3 oc = r.origin() - center;
    float a = dot(r.direction(), r.direction());
    float b = 2.0f * dot(r.direction(), oc);
    float c = dot(oc, oc) - pow2(radius);
    float D = b*b - 4 * a*c;
    if (D < 0) {
        return -1.0f;
    }
    else {
        return (-b - sqrtf(D)) / (2.0f*a);
    }
}

vec3 color(const rayt::Ray& r) {
    vec3 c(0,0,-1);
    float t = hit_sphere(c, 0.5f, r);
    if (t > 0.0f) {
        vec3 N = normalize(r.at(t) - c);
        return 0.5f*(N+vec3(1.0f));
    }
    vec3 d = normalize(r.direction());
    t = 0.5f*(r.direction().getY() + 1.0f);
    return lerp(t, vec3(1), vec3(0.5f, 0.7f, 1.0f));
}
```

これは次のような画像になります. (rayt104.cpp)



7. シーンクラス

ここで, 少しコードを整理するためにシーンクラスを追加します.シーンはカメラや画像出力, 球などの物体を管理します.これまで作成した `hit_sphere` や `color` 関数, カメラや画像クラスのインスタンスをシーンクラスにまとめます.また, 背景色としてグラデーションのものは空っぽいので `backgroundSky` 関数, 単色用に `background` 関数を追加し, `color` 関数から呼び出しています.好きな方を使ってください. `render` 関数はレンダリングを行う関数で, `build` 関数はレンダリングするときに一度呼ばれる関数でカメラや物体の生成などの初期化を行います.

```
#include "rayt.h"
```

```
namespace rayt {

    //------------------------------------------------------------------------
    class Scene {
    public:
        Scene(int width, int height)
            : m_image(std::make_unique<Image>(width, height))
            , m_backColor(0.2f) {
        }

        void build() {

            // Camera

            vec3 w(-2.0f, -1.0f, -1.0f);
            vec3 u(4.0f, 0.0f, 0.0f);
            vec3 v(0.0f, 2.0f, 0.0f);
            m_camera = std::make_unique<Camera>(u, v, w);
        }

        float hit_sphere(const vec3& center, float radius, const rayt::Ray& r) const {
            vec3 oc = r.origin() - center;
            float a = dot(r.direction(), r.direction());
            float b = 2.0f * dot(r.direction(), oc);
            float c = dot(oc, oc) - pow2(radius);
            float D = b*b - 4 * a*c;
            if (D < 0) {
                return -1.0f;
            }
            else {
                return (-b - sqrtf(D)) / (2.0f*a);
            }
        }

        vec3 color(const rayt::Ray& r) {
            vec3 c(0, 0, -1);
            float t = hit_sphere(c, 0.5f, r);
            if (t > 0.0f) {
                vec3 N = normalize(r.at(t) - c);
                return 0.5f*(N + vec3(1.0f));
            }
            return backgroundSky(r.direction());
        }

        vec3 background(const vec3& d) const {
            return m_backColor;
        }
```

```
        vec3 backgroundSky(const vec3& d) const {
            vec3 v = normalize(d);
            float t = 0.5f * (v.getY() + 1.0f);
            return lerp(t, vec3(1), vec3(0.5f, 0.7f, 1.0f));
        }

        void render() {

            build();

            int nx = m_image->width();
            int ny = m_image->height();
            for (int j = 0; j<ny; ++j) {
                std::cerr << "Rendering (y = " << j << ") " << (100.0 * j / (ny - 1)) << "%" << std::endl;
                for (int i = 0; i<nx; ++i) {

                    float u = float(i + drand48()) / float(nx);
                    float v = float(j + drand48()) / float(ny);
                    Ray r = m_camera->getRay(u, v);
                    vec3 c = color(r);
                    m_image->write(i, (ny - j - 1), c.getX(), c.getY(), c.getZ());
                }
            }

            stbi_write_bmp("render.bmp", nx, ny, sizeof(Image::rgb), m_image->pixels());
        }

    private:
        std::unique_ptr<Camera> m_camera;
        std::unique_ptr<Image> m_image;
        vec3 m_backColor;
    };

} // namespace rayt
```

`main` 関数はシーンクラスのインスタンスを作成し，`render` 関数を呼びます. (rayt104.cpp)

```
int main()
{
    int nx = 200;
    int ny = 100;
    std::unique_ptr<rayt::Scene> scene(std::make_unique<rayt::Scene>(nx, ny));
    scene->render();

    return 0;
}
```

7.1 並列処理 - OpenMP

レイトレーシングは並列処理に向いています.今回の実装では各光線ごとに計算を並列で処理させることが可能です. Visual Studio には並列処理のフレームワークである OpenMP が使えるようになっています. OpenMP はコード中にディレクティブを挿入することで簡単に並列処理を実現することができます. `render` 関数に以下のディレクティブを入れます.

```
#pragma omp parallel for schedule(dynamic, 1) num_threads(NUM_THREAD)
```

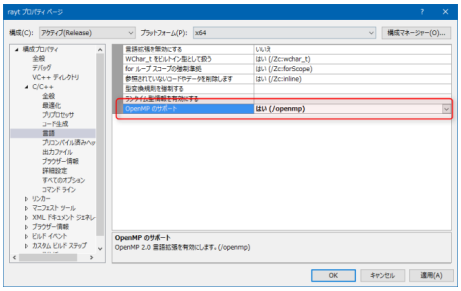
実際のコードは以下のようになります.

```
// rayt.cpp に記載
class Scene {
    ...

    void render() {

        build();

        int nx = m_image->width();
        int ny = m_image->height();
#pragma omp parallel for schedule(dynamic, 1) num_threads(NUM_THREAD)
        for (int j = 0; j<ny; ++j) {
            std::cerr << "Rendering (y = " << j << ") " << (100.0 * j / (ny - 1)) << "%" << std::endl;
            for (int i = 0; i<nx; ++i) {

                float u = float(i + drand48()) / float(nx);
                float v = float(j + drand48()) / float(ny);
                Ray r = m_camera->getRay(u, v);
                vec3 c = color(r);
                m_image->write(i, (ny - j - 1), c.getX(), c.getY(), c.getZ());
            }
        }

        stbi_write_bmp("render.bmp", nx, ny, sizeof(Image::rgb), m_image->pixels());
    }
}
```

ディレクティブについての詳しい内容はネットで検索してみてください. `NUM_THREAD` は使用するスレッドの数です.お使いの PC に合わせたスレッド数を指定します.

```
// rayt.h に記載
#define NUM_THREAD 8 // 例
```

このままでは，OpenMP のディレクティブは有効になっていません．Visual Studio のプロジェクト設定で OpenMP を有効にする必要があります．



あとはビルドして実行するだけで並列処理されます.また，OpenMP は無効になっていたらディレクティブは無視されるのでコードを変更する必要がありません．

並列処理を有効にしてレンダリングすると場合によっては画像がおかしくなる可能性があります.その場合は一度並列処理を疑います.例えば `schedule(dynamic,1)` を消してみたり，並列処理そのものを無効にするなどで確認してみてください．

(rayt105.cpp, rayt105.h)

8. 複数の物体への対応

今までは球が 1 つでしたが，複数の物体にも対応できるように拡張します．

8.1 衝突情報

まず，物体に衝突したときの情報を格納するクラス `HitRec` を用意します．

```
class HitRec {
public:
    float t;
    vec3 p;
    vec3 n;
};
```

`t` は光線のパラメータ，`p` は衝突した位置，`n` は衝突した位置の法線です．

8.2 物体の抽象クラス

次に，物体の抽象クラス `Shape` を追加します.これは衝突関数が仮想関数で定義されています．

```
class Shape {
public:
    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const = 0;
};
```

`t0` と `t1` は光線の衝突範囲です．

8.3 球クラス

そして,球クラス `Sphere` を追加します.

```
class Sphere : public Shape {
public:
    Sphere() {}
    Sphere(const vec3& c, float r)
        : m_center(c)
        , m_radius(r) {
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
        vec3 oc = r.origin() - m_center;
        float a = dot(r.direction(), r.direction());
        float b = 2.0f*dot(oc, r.direction());
        float c = dot(oc, oc) - pow2(m_radius);
        float D = b*b - 4*a*c;
        if (D > 0) {
            float root = sqrtf(D);
            float temp = (-b - root) / (2.0f*a);
            if (temp < t1 && temp > t0) {
                hrec.t = temp;
                hrec.p = r.at(hrec.t);
                hrec.n = (hrec.p - m_center) / m_radius;
                return true;
            }
            temp = (-b + root) / (2.0f*a);
            if (temp < t1 && temp > t0) {
                hrec.t = temp;
                hrec.p = r.at(hrec.t);
                hrec.n = (hrec.p - m_center) / m_radius;
                return true;
            }
        }

        return false;
    }

private:
    vec3 m_center;
    float m_radius;
};
```

`override` は仮想関数をオーバーライドすることをコンパイラに知らせるキーワードです.引数の間違いや誤字などでオーバーライドが出来ていないことを防げます.

`hit` 関数での衝突判定は少し変更しています.解が2つあるとき, $t = \frac{-b-\sqrt{D}}{2a}$ の方が始点に近いため, 近いほうを先に判定しています.

8.4 物体リスト

複数の物体を管理するクラスを追加します.これは `Shape` を継承することで, 物体リストも単体の物体として動作するようになります.

```
class ShapeList : public Shape {
public:
    ShapeList() {}

    void add(const ShapePtr& shape) {
        m_list.push_back(shape);
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
        HitRec temp_rec;
        bool hit_anything = false;
        float closest_so_far = t1;
        for (auto& p : m_list) {
            if (p->hit(r, t0, closest_so_far, temp_rec)) {
                hit_anything = true;
                closest_so_far = temp_rec.t;
                hrec = temp_rec;
            }
        }
        return hit_anything;
    }

private:
    std::vector<ShapePtr> m_list;
};
```

`add` 関数で物体を追加できます.物体は `std::shared_ptr<Shape>` で渡します.これを打つのは面倒なので typedef で定義しておきます.

```
class Shape;
typedef std::shared_ptr<Shape> ShapePtr;
```

また, STL のコンテナを使っているので必要なヘッダをインクルードする必要があります.

```
// rayt.h
#include <vector>
```

それでは球を2つ表示してみます.そのためには `Scene` クラスをいくつか修正する必要があります.まず, 物体リストのインスタンスを追加します.

```
std::unique_ptr<Shape> m_world;
```

次に `build` 関数で物体を生成します.

```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(vec3(0, 0, -1), 0.5f));
world->add(std::make_shared<Sphere>(vec3(0, -100.5, -1), 100));
m_world.reset(world);
```

そして, `color` 関数に物体を渡し, `Shape::hit` 関数を呼ぶようにします.

```
vec3 color(const rayt::Ray& r, const Shape* world) const {
    HitRec hrec;
    if (world->hit(r, 0, FLT_MAX, hrec)) {
        return 0.5f*(hrec.n + vec3(1.0f));
    }
    return backgroundSky(r.direction());
}
```

`render` 関数では `color` 関数に物体を渡します.

```
vec3 c = color(r, m_world.get());
```

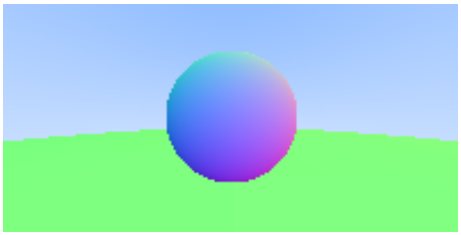
最終的に `Scene` クラスは次のようになります.

```
namespace rayt {
    ...

    class Scene {
    public:
        Scene(int width, int height)
            : m_image(std::make_unique<Image>(width, height))
            , m_backColor(0.2f) {
        }

        void build() {

            // Camera

            vec3 w(-2.0f, -1.0f, -1.0f);
            vec3 u(4.0f, 0.0f, 0.0f);
            vec3 v(0.0f, 2.0f, 0.0f);
            m_camera = std::make_unique<Camera>(u, v, w);
```

```
            // Shapes

            ShapeList* world = new ShapeList();
            world->add(std::make_shared<Sphere>(
                vec3(0, 0, -1), 0.5f));
            world->add(std::make_shared<Sphere>(
                vec3(0, -100.5, -1), 100));
            m_world.reset(world);
        }

        vec3 color(const rayt::Ray& r, const Shape* world) const {
            HitRec hrec;
            if (world->hit(r, 0, FLT_MAX, hrec)) {
                return 0.5f*(hrec.n + vec3(1.0f));
            }
            return backgroundSky(r.direction());
        }

        vec3 background(const vec3& d) const {
            return m_backColor;
        }

        vec3 backgroundSky(const vec3& d) const {
            vec3 v = normalize(d);
            float t = 0.5f * (v.getY() + 1.0f);
            return lerp(t, vec3(1), vec3(0.5f, 0.7f, 1.0f));
        }

        void render() {

            build();

            int nx = m_image->width();
            int ny = m_image->height();
#pragma omp parallel for schedule(dynamic, 1) num_threads(NUM_THREAD)
            for (int j = 0; j<ny; ++j) {
                std::cerr << "Rendering (y = " << j << ") " << (100.0 * j / (ny - 1)) << "%" << std::endl;
                for (int i = 0; i<nx; ++i) {

                    float u = float(i) / float(nx);
                    float v = float(j) / float(ny);
                    Ray r = m_camera->getRay(u, v);
                    vec3 c = color(r, m_world.get());
                    m_image->write(i, (ny - j - 1), c.getX(), c.getY(), c.getZ());
                }
            }
```

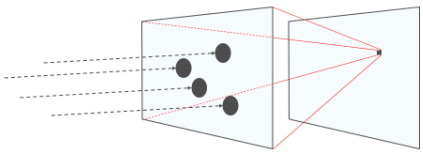
```
            stbi_write_bmp("render.bmp", nx, ny, sizeof(Image::rgb), m_image->pixels());
        }

    private:
        std::unique_ptr<Camera> m_camera;
        std::unique_ptr<Image> m_image;
        std::unique_ptr<Shape> m_world;
        vec3 m_backColor;
    };

} // namespace rayt
```

実行すると次のような画像になります. (rayt106.cpp, rayt106.h)



9. アンチエイリアシング

これまでレンダリングした画像を見てみると球体の周りでジャギーが目立っています.これを取り除くために, 各ピクセルごとにいくつも光線を飛ばして(このとき乱数でずらします), その平均を使用します.図で表すと次のようになります.



この手法をスーパーサンプリングといいます.まずサンプリング数を指定できるようにするために `Scene` のコンストラクタを変更します.

```
Scene(int width, int height, int samples)
    : m_image(std::make_unique<Image>(width, height))
    , m_backColor(0.2f)
    , m_samples(samples) {
}
```

メンバに `m_samples` を追加します.

```
int m_samples;
```

そして, 光線を飛ばしているところを次のように書き換えます.

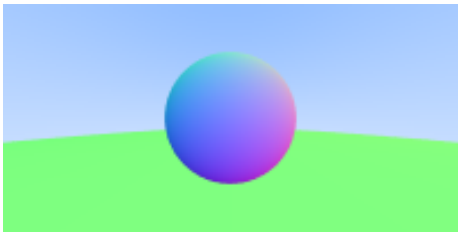
```
void render() {
```

```
    ...

    int nx = m_image->width();
    int ny = m_image->height();
#pragma omp parallel for schedule(dynamic, 1) num_threads(NUM_THREAD)
    for (int j = 0; j<ny; ++j) {
        std::cerr << "Rendering (y = " << j << ") " << (100.0 * j / (ny - 1)) << "%" << std::endl;
        for (int i = 0; i<nx; ++i) {
            vec3 c(0);
            for (int s = 0; s<m_samples; ++s) {
                float u = (float(i) + drand48()) / float(nx);
                float v = (float(j) + drand48()) / float(ny);
                Ray r = m_camera->getRay(u, v);
                c += color(r, m_world.get());
            }
            c /= m_samples;
            m_image->write(i, (ny - j - 1), c.getX(), c.getY(), c.getZ());
        }
    }

    ...
}
```

`main` 関数でサンプリング数を指定します.

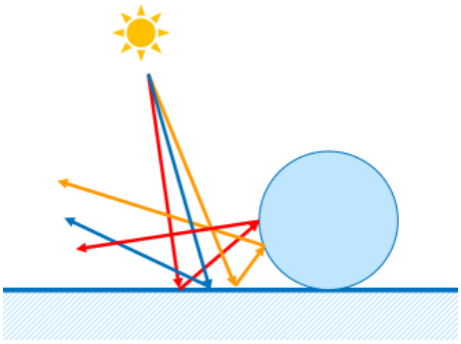
```
// rayt.cpp に記載
int main()
{
    int nx = 200;
    int ny = 100;
    int ns = 100;
    std::unique_ptr<rayt::Scene> scene(std::make_unique<rayt::Scene>(nx, ny, ns));
    scene->render();
    return 0;
}
```

実行すると次のようになります. (rayt107.cpp)

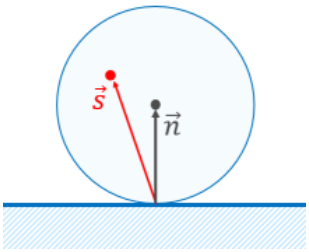


10. 拡散反射

光は物体の表面に当たると，反射するか，透過するか，もしくは両方の現象が起きます.それは物体の表面の材質によって変わります.ほとんどの物体の表面は凹凸があり，眼で確認できるものもあれば，顕微鏡などで見ないとわからないとても小さい凹凸があります.このような表面に光が当たると様々な方向に反射されます.そのような現象を乱反射といい，コンピュータグラフィックでは拡散反射としてシミュレーションします.



この現象をシミュレーションするために，光線が物体に当たったとき，ランダムな方向に反射させればよいことになります.
それでは，ランダムに反射する方向を決めるにはどうすればよいでしょうか.一様乱数を生成し，単位球内にある点を取り出して使うとよさそうです.図にすると $\vec{s}$ の位置に向かって反射させます.



区間 [0,1] の乱数を [-1,1] の区間に変換して，半径 1 の球内であればそれを使用します.

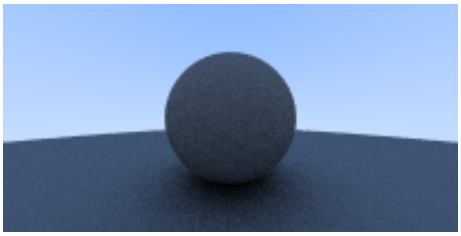
```
// rayt.h
inline vec3 random_vector() {
    return vec3(drand48(), drand48(), drand48());
}

inline vec3 random_in_unit_sphere() {
    vec3 p;
    do {
        p = 2.f * random_vector() - vec3(1.f);
    } while (lengthSqr(p) >= 1.f);
    return p;
}
```

`random_vector` は一様乱数を使ってベクトルを作成します. `random_in_unit_sphere` は単位球の中の任意の点を生成します.この関数では，ランダムに生成したベクトルの半径が 1 以下になるまで繰り返すようになっています.このように条件を満たすまで一様乱数を何度も

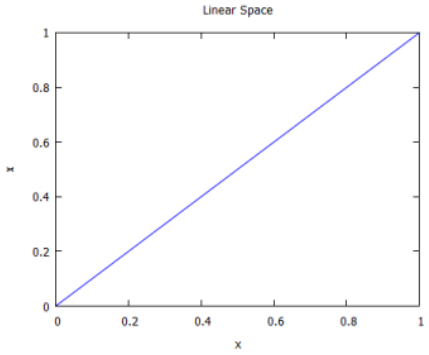
生成して繰り返すことを棄却法といいます.これを使って拡散反射を実装してみます. `color` 関数を次のように書き換えます.

```
vec3 color(const rayt::Ray& r, const Shape* world) const {
    HitRec hrec;
    if (world->hit(r, 0, FLT_MAX, hrec)) {
        vec3 target = hrec.p + hrec.n + random_in_unit_sphere();
        return 0.5f * color(Ray(hrec.p, target - hrec.p), world);
    }
    return backgroundSky(r.direction());
}
```

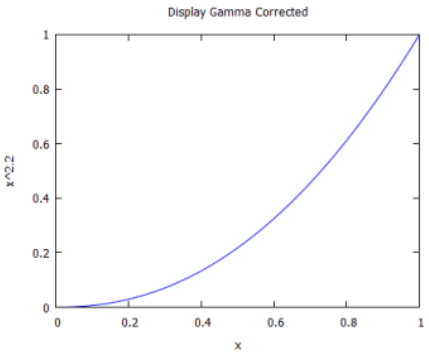
ここでは反射率を 50% にしました.次のような画像になります. (rayt108.cpp, rayt108.h)



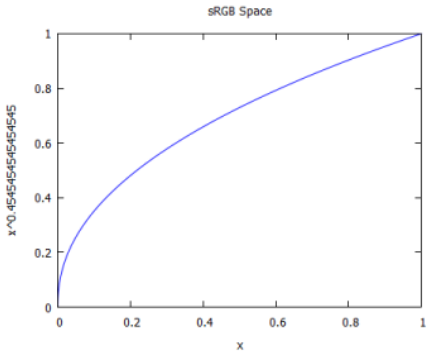
この画像暗い感じがしますね.これはディスプレイのガンマ補正の影響です.プログラム上では色をリニア空間で扱っています.例えば区間 [0,1] の値はリニア空間つまり線形なので, 次の図のように直線になります.



ディスプレイは入力した信号を基本的にガンマ補正するようになっています.特に設定を変更していなければガンマ係数は 2.2 になっています.この場合は下に歪んでいます.



つまり, 全体的に暗くなってしまいます.ではどうすればいいかというと, ガンマ補正がかかることを想定して, ガンマ補正されたらリニア空間になるように画像データの方を調整します.具体的にはリニア空間のデータを以下のようなカラー空間に変換します.



このようなカラー空間を **sRGB 空間**といいます.リニア空間と sRGB 空間との変換式は次のようになっています.

$$C_{srgb} = C_{linear}^{1/2.2}, \quad C_{linear} = C_{srgb}^{2.2}$$

これは近似式で他に厳密な式が存在します.ではこの処理を入れてみます.最初にリニア空間と sRGB 空間との変換関数を追加します.

```
// rayt.h
inline vec3 linear_to_gamma(const vec3& v, float gammaFactor) {
    float recipGammaFactor = recip(gammaFactor);
    return vec3(
        powf(v.getX(), recipGammaFactor),
        powf(v.getY(), recipGammaFactor),
        powf(v.getZ(), recipGammaFactor));
}

inline vec3 gamma_to_linear(const vec3& v, float gammaFactor) {
    return vec3(
        powf(v.getX(), gammaFactor),
        powf(v.getY(), gammaFactor),
        powf(v.getZ(), gammaFactor));
}
```

`linear_to_gamma` がリニア空間から sRGB 空間へ，`gamma_to_linear` が sRGB 空間からリニア空間に変換します.

`Image` クラスの `write` で色を書き込んでいますので,そこでフィルター処理するようにします.また,新しいフィルターを簡単に追加できるような仕組みにします.まず,フィルターの抽象クラスを定義します.

```
// rayt.h
class ImageFilter {
public:
    virtual vec3 filter(const vec3& c) const = 0;
};
```

`filter` 関数は色を受け取ってフィルター処理した色を返します.このフィルタークラスを派生して,ガンマ補正をかけるフィルタークラスを作成します.

```
// rayt.h
class GammaFilter : public ImageFilter {
public:
    GammaFilter(float factor) : m_factor(factor) {}
    virtual vec3 filter(const vec3& c) const override {
        return linear_to_gamma(c, m_factor);
    }
private:
    float m_factor;
};
```

ガンマ係数を指定できるようにしていますが, 通常は 2.2 になります.次に `Image` クラスで, フィルター処理を追加します. `ImageFilter` クラスを複数持てるようにリストでメンバに追加します.

```
std::vector< std::unique_ptr<ImageFilter> > m_filters;
```

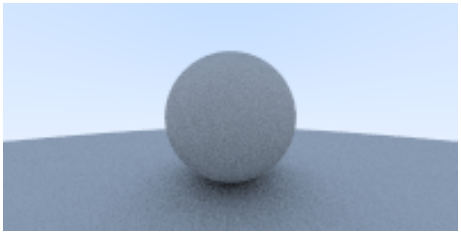
コンストラクタでフィルターを生成します.

```
Image(int w, int h) {
    m_width = w;
    m_height = h;
    m_pixels.reset(new rgb[m_width * m_height]);
    m_filters.push_back(std::make_unique<GammaFilter>(GAMMA_FACTOR));
}
```

そして, `write` 関数内でフィルター処理を行います.

```
void write(int x, int y, float r, float g, float b) {
    vec3 c(r, g, b);
    for (auto& f : m_filters) {
        c = f->filter(c);
    }
    int index = m_width*y + x;
    m_pixels[index].r = static_cast<unsigned char>(c.getX() * 255.99f);
    m_pixels[index].g = static_cast<unsigned char>(c.getY() * 255.99f);
    m_pixels[index].b = static_cast<unsigned char>(c.getZ() * 255.99f);
}
```

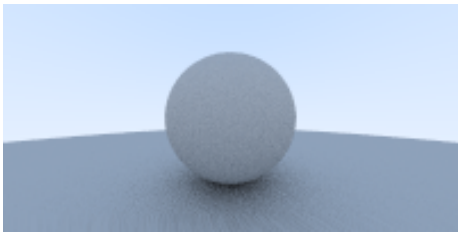
この結果, 次のようになります. (rayt109.h)



この画像，まだおかしなところがあります.このシーンでは光線が最終的に空に向かうのですが，球体の上部や，床に暗いところが多くあります.何か明るいところと暗いところが混じって斑模様のように見えませんか？ これは計算誤差によるもので，反射した位置から次に衝突する位置を判定するときに，t が 0 に近い位置で当たったと判定されていて，反射方向がおかしくなっているようです.そのため，当たり判定のときに，t の区間を 0 からではなく例えば 0.001 のようにずらします.

```
if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
  ...
}
```

この結果は次のようになります. (rayt110.cpp)



先ほどの画像より暗い部分が減っていることがわかると思います.斑模様みたいなものはシャドウアクネと呼ばれることがあります.

11. 材質

物体の反射の特性は表面の材質によって決まります.今は拡散反射だけですが，これから鏡面反射などを追加するためにも，材質をクラスにしましょう.材質に必要な機能は，その材質の表面に光が当たったときにどの方向に光が反射するかということと，入射した光に対してどれくらい反射するかを表す反射率です.光が物体表面に当たったときに反射する，つまり光の向きを変える現象を散乱といいます.また，物体の表面は光を吸収することがあり，吸収されなかった光が散乱して放出されます.反射率は入射した光が吸収されない光の割合とも言えます.この反射率をアルベドまたはリフレクタンスといいます.

材質クラスは抽象クラスで次のように定義します.

```
class Material {
public:
    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const = 0;
};
```

`scatter` 関数は散乱をシミュレーションします.散乱後の光の向きと，反射率を `ScatterRec` クラスに格納します. `ScatterRec` は次のようになっています.

```
class ScatterRec {
public:
    Ray ray;
    vec3 albedo;
};
```

`ray` には散乱後の新しい光線，`albedo` は反射率です.前に実装した拡散反射のような材質をランバートとよぶことがあります.材質クラスとして次のように書き直します.

```
class Lambertian : public Material {
public:
    Lambertian(const vec3& c)
        : m_albedo(c) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        vec3 target = hrec.p + hrec.n + random_in_unit_sphere();
        srec.ray = Ray(hrec.p, target - hrec.p);
        srec.albedo = m_albedo;
        return true;
    };

private:
    vec3 m_albedo;
};
```

このランバート材質はアルベドを指定するようになっています.次にこの材質を処理するようにいくつか変更します. `HitRec` に材質を追加します.

```
class HitRec {
public:
    float t;
    vec3 p;
    vec3 n;
    MaterialPtr mat;
};
```

`MaterialPtr` は typedef で定義しています.

```
class Material;
typedef std::shared_ptr<Material> MaterialPtr;
```

次に `Sphere` に材質を指定できるようにし，`hit` 関数で当たったとき， `HitRec` の材質に設定するようにします.

```
class Sphere : public Shape {
public:
    Sphere() {}
    Sphere(const vec3& c, float r, const MaterialPtr& mat)
        : m_center(c)
        , m_radius(r)
        , m_material(mat) {
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
```

```
        vec3 oc = r.origin() - m_center;
        float a = dot(r.direction(), r.direction());
        float b = 2.0f*dot(oc, r.direction());
        float c = dot(oc, oc) - pow2(m_radius);
        float D = b*b - 4*a*c;
        if (D > 0) {
            float root = sqrtf(D);
            float temp = (-b - root) / (2.0f*a);
            if (temp < t1 && temp > t0) {
                hrec.t = temp;
                hrec.p = r.at(hrec.t);
                hrec.n = (hrec.p - m_center) / m_radius;
                hrec.mat = m_material;
                return true;
            }
            temp = (-b + root) / (2.0f*a);
            if (temp < t1 && temp > t0) {
                hrec.t = temp;
                hrec.p = r.at(hrec.t);
                hrec.n = (hrec.p - m_center) / m_radius;
                hrec.mat = m_material;
                return true;
            }
        }

        return false;
    }

private:
    vec3 m_center;
    float m_radius;
    MaterialPtr m_material;
};
```

そして，`color` 関数で材質を処理します．

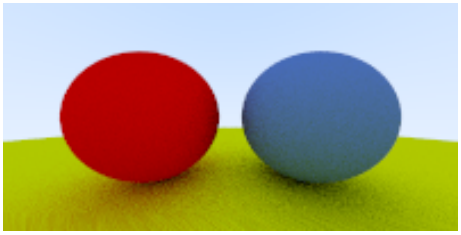
```
vec3 color(const rayt::Ray& r, const Shape* world) const {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        ScatterRec srec;
        if (hrec.mat->scatter(r, hrec, srec)) {
            return mulPerElem(srec.albedo, color(srec.ray, world));
        }
        else {
            return vec3(0);
        }
    }
```

```
    return backgroundSky(r.direction());
}
```

散乱させて, 次の方向から受け取った色と反射率を乗算しています.あとは作成する球体を追加し, 材質を設定します.

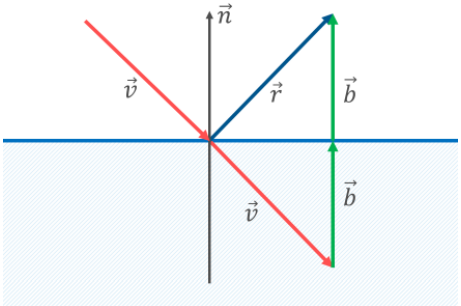
```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(vec3(0.1f, 0.2f, 0.5f))));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(vec3(0.8f, 0.0f, 0.0f))));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(vec3(0.8f, 0.8f, 0.0f))));
m_world.reset(world);
```

この結果は次のようになります. (rayt112.cpp)



12. 鏡面反射

金属のような材質のものは, 拡散反射はせずに鏡面反射します.物体の表面が完全に平坦なとき, 入射角と反射角は同じになります.このような反射ベクトルを求めるには次のように考えます.



$\vec{r}$ が求める反射ベクトルで, $\vec{r} = \vec{v} + 2\vec{b}$ です. $\vec{v}$ は入射ベクトルで, $\vec{b}$ は $\vec{v}$ を $\vec{n}$ に投影したベクトルを反転したものです. $-(\vec{v} \cdot \vec{n}) \cdot \vec{n}$ となります. $\vec{n}$ は単位ベクトルで, $\vec{v}$ は任意の長さのベクトルです.これをコードにすると

```
inline vec3 reflect(const vec3& v, const vec3& n) {
    return v - 2.f * dot(v, n)*n;
}
```

これを使って鏡面反射する材質を作成します.この材質は金属と呼びます.

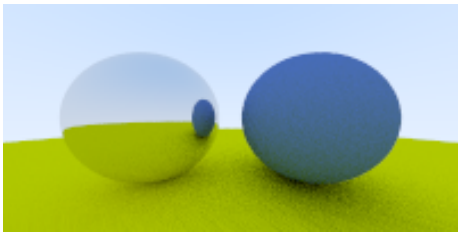
```
class Metal : public Material {
public:
    Metal(const vec3& c, float fuzz)
        : m_albedo(c) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        vec3 reflected = reflect(normalize(r.direction()), hrec.n);
        srec.ray = Ray(hrec.p, reflected);
        srec.albedo = m_albedo;
        return dot(srec.ray.direction(), hrec.n) > 0;
    }

private:
    vec3 m_albedo;
};
```

`scatter` の戻り値は反射ベクトルと法線ベクトルとのなす角度が 0 より大きいときに真とします.

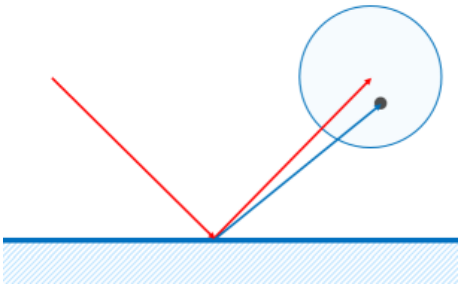
シーンの球体 1 つを金属材質に変えてみます.

```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(vec3(0.1f, 0.2f, 0.5f))));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), 0.5f,
    std::make_shared<Metal>(vec3(0.8f, 0.8f, 0.8f))));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(vec3(0.8f, 0.8f, 0.0f))));
m_world.reset(world);
```

次のような画像になります. (rayt113.cpp, rayt113.h)



今は完全に平坦な表面を考えていましたが, 表面は多少凹凸しています.凹凸のある表面に光線が当たると反射ベクトルは理想よりずれます.この現象をシミュレーションするためには, 高度なものだと微小面モデルとかあるのですが, ここでは簡易な方法で行います.理想的な反射ベクトルの先を中心とした単位球内の点を無作為に選んでそこまでのベクトルを反射ベクトルとします.図にすると以下のようになります.



赤いベクトルは理想的な反射ベクトルです.それに対して青いベクトルはずらした反射ベクトルです.ずらし具合をパラメータで調整できるようにします.このパラメータを `fuzz` とします.これを実装します.

```
class Metal : public Material {
public:
    Metal(const vec3& c, float fuzz)
        : m_albedo(c)
        , m_fuzz(fuzz) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        vec3 reflected = reflect(normalize(r.direction()), hrec.n);
        reflected += m_fuzz*random_in_unit_sphere();
        srec.ray = Ray(hrec.p, reflected);
        srec.albedo = m_albedo;
        return dot(srec.ray.direction(), hrec.n) > 0;
    }

private:
    vec3 m_albedo;
    float m_fuzz;
};
```

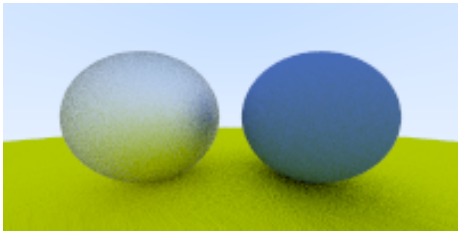
通常通り反射ベクトルを計算したあとに,単位球から無作為に点を生成して,それに `fuzz` を乗算して計算した反射ベクトルに加算します.

```
reflected += m_fuzz*random_in_unit_sphere();
```

球体の金属材質を調整します.

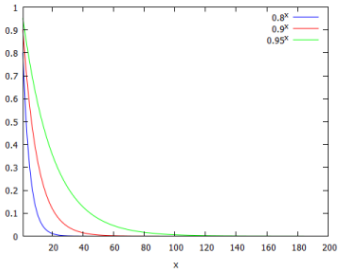
```
std::make_shared<Metal>(vec3(0.8f, 0.8f, 0.8f), 1.f)));
```

この結果は次のようになります. (rayt114.cpp)



ぼやけているように見えます.

さて, 拡散反射や鏡面反射において, ランダムな方向に反射させるようになったので, 反射が複雑になってきています. `color` 関数は反射される度に呼び出されていて再帰的です.どれくらい再帰的に呼び出されるか調査してみると 1 つの光線で最高 180~250 回呼ばれていました.光は反射される度に反射率で減衰していきます.今は反射率 0.8 とか設定しているのですが, その反射率で例えば 100 回反射したら最終的に 0.8^{100} となりかなり小さい値になり結局黒くなります.反射率と反射回数との関係をグラフにしてみると次のようになります.



反射率 0.8 なら反射回数 40 回ぐらいでほぼ 0 になります.先ほどのシーンでは, ランバート材質の物体にも当たることがあり, 反射率は 0.5 とかさらに小さいので, これよりもっと早く 0 に収束していきます.このことから, ある程度の反射回数以上は計算の無駄なので, どこかで打ち切ります.この打ち切る条件を決めるために, ロシアンルーレットと呼ばれる手法がよく使われるのですが, 今回は特定の反射回数で打ち切ります.

`color` 関数を次のように書き換えます.

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            return mulPerElem(srec.albedo, color(srec.ray, world, depth+1));
        }
        else {
            return vec3(0);
        }
    }
    return backgroundSky(r.direction());
}
```

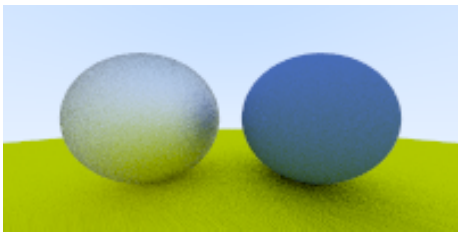
`depth` は再起呼び出しの深さで, `MAX_DEPTH` 以上になったら打ち切ります. `MAX_DEPTH` は適当に決めます.とりあえず 50 回にしておきます.

```
// rayt.h
#define MAX_DEPTH 50
```

`color` 関数を呼び出しているところで，`depth` に 0 を渡します．

```
c += color(r, m_world.get(), 0);
```

これでレンダリングすると次のようになります．結果はほとんど変わりません．(rayt115.cpp,rayt115.h)



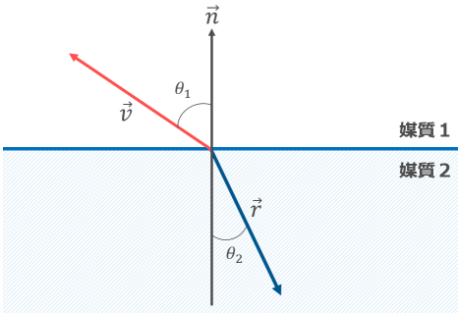
13. 屈折

水やガラス，ダイアモンドなどは見てみると，その先が歪んで見えたり，反射した方向の景色が映りこんで見えます．このような材質に光が当たると，光は鏡面反射する光と物体内部に透過する光に分かれます．光は電磁波の一種で，媒質の中を移動しています．空気も媒質の一つです．例えば光が空気中から別の媒質に入ると光の速度が変化して屈折します．どれくらい屈折するかは入る前の媒質の屈折率，侵入する媒質の屈折率，侵入するときの入射角によって求めることができます．この関係を表したのがスネルの法則です．それによれば，媒質 A の屈折率を η_1，入射角を θ_1，媒質 B の屈折率を η_2，出射角を θ_2 とすると以下の式が成り立ちます．

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

空気の屈折率は 1，ガラスは 1.3–1.7，ダイヤモンドは 2.4 です．他の屈折率はネットで検索すれば調べることが色々と出てくると思います．

屈折ベクトルはスネルの法則から導くことができます．図のように $\vec{r}$ は屈折ベクトル，$\vec{v}$ は方向ベクトル，$\vec{n}$ は表面の法線です．



屈折ベクトル $\vec{r}$ は

$$\vec{r} = -\frac{\eta_1}{\eta_2}(\vec{v} - (\vec{v}\cdot\vec{n})\vec{n}) - \vec{n}\sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - (\vec{v}\cdot\vec{n}^2)}$$

この式の導出についてはスネルの法則を参照してください．

これをコードにすると次のようになります．

```
inline bool refract(const vec3& v, const vec3& n, float ni_over_nt, vec3& refracted) {
```

```
    vec3 uv = normalize(v);
    float dt = dot(uv, n);
    float D = 1.f - pow2(ni_over_nt) * (1.f - pow2(dt));
    if (D > 0.f) {
        refracted = -ni_over_nt * (uv - n*dt) - n*sqrt(D);
        return true;
    }
    else {
        return false;
    }
}
```

`D` は判別式です.一体何を判別しているのでしょうか.ここでスネルの法則を次のように変形します.

$$\sin\theta_2 = \frac{\eta_1}{\eta_2}\sin\theta_1$$

θ_1 は $[0,90]$ の範囲なので, $\sin\theta_1$ の取る範囲は $0 \leq \sin\theta_1 \leq 1$ です. $\eta_1 < \eta_2$ ならば $\frac{\eta_1}{\eta_2} < 1$ となるので, 上記の式から $0 \leq \sin\theta_2 \leq 1$ となることがわかります.しかし, $\eta_1 > \eta_2$ の場合は $\frac{\eta_1}{\eta_2} > 1$ なので, $\sin\theta_1$ が大きいと, $\sin\theta_2 > 1$ となって, 解がありません.これは, 屈折光がなくなり, 反射光のみになります.この現象を全反射といいます.

屈折をする材質を誘導体といいます.これを実装すると

```
class Dielectric : public Material {
public:
    Dielectric(float ri)
        : m_ri(ri) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {

        vec3 outward_normal;
        vec3 reflected = reflect(r.direction(), hrec.n);
        float ni_over_nt;
        if (dot(r.direction(), hrec.n) > 0) {
            outward_normal = -hrec.n;
            ni_over_nt = m_ri;
        }
        else {
            outward_normal = hrec.n;
            ni_over_nt = recip(m_ri);
        }

        srec.albedo = vec3(1);

        vec3 refracted;
        if (refract(-r.direction(), outward_normal, ni_over_nt, refracted)) {
            srec.ray = Ray(hrec.p, refracted);
```

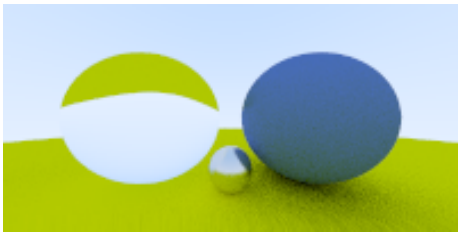
```
        }
        else {
            srec.ray = Ray(hrec.p, reflected);
            return false;
        }

        return true;
    }

private:
    float m_ri;
};
```

`m_ri` は屈折率です.屈折ベクトルは物体の内部に入射するときと, 内部から外部に出射するときとでは屈折率が反転するので, 内積を計算して判定しています.

この誘導体材質の球を追加します.

```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(vec3(0.1f, 0.2f, 0.5f))));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), 0.5f,
    std::make_shared<Dielectric>(1.5f)));
world->add(std::make_shared<Sphere>(
    vec3(0, -0.35, -0.8f), 0.15f,
    std::make_shared<Metal>(vec3(0.8f, 0.8f, 0.8f), 0.2f)));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(vec3(0.8f, 0.8f, 0.0f))));
m_world.reset(world);
```

次のような画像になります. (rayt116.cpp, rayt116.h)



表面の法線に対して光線の向きが表面に近い角度をグレージング角度といいます.金属や誘導体はグレージング角度に近づくと屈折光が少なくなり, 材質によっては全反射します.このような材質が光が入射したときに, どれくらいの比率で反射光と屈折光に分配されるかはフレネルの方程式で求めることができます.フレネルの方程式は偏光されていないのであれば, Schlick の近似式がよく使われます.

$$F_r(\theta) \approx F_0 + (1 - F_0)(1 - \cos\theta)^5$$

このとき, $F_r(\theta)$ はフレネル反射係数で, F_0 は, 表面に対して垂直に光が入射したときのフレネル反射係数です. $F_r(\theta)$ が, 入射した光に対する鏡面反射率を表しています.

F_0 は屈折率を使って求められます.

$$F_0 = \frac{(\eta_1 - \eta_2)^2}{(\eta_1 + \eta_2)^2} = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2$$

これをコードにすると次のようになります.

```
inline float schlick(float cosine, float ri) {
    float r0 = pow2((1.f - ri) / (1.f + ri));
    return r0 + (1.f - r0) * pow5(1.f - cosine);
}
```

基本的に大気中からある媒質に入ることを想定するので, 片方の媒質は空気で屈折率はほぼ 1 です.これを誘導体クラスに組み込みます. 次のように書き換えます.

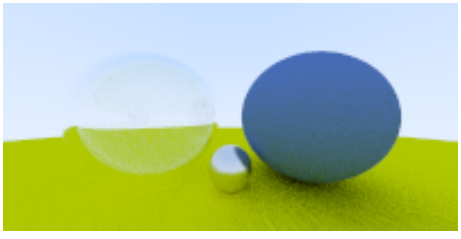
```
class Dielectric : public Material {
public:
    Dielectric(float ri)
        : m_ri(ri) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        vec3 outward_normal;
        vec3 reflected = reflect(r.direction(), hrec.n);
        float ni_over_nt;
        float reflect_prob;
        float cosine;
        if (dot(r.direction(), hrec.n) > 0) {
            outward_normal = -hrec.n;
            ni_over_nt = m_ri;
            cosine = m_ri * dot(r.direction(), hrec.n) / length(r.direction());
        }
        else {
            outward_normal = hrec.n;
            ni_over_nt = recip(m_ri);
            cosine = -dot(r.direction(), hrec.n) / length(r.direction());
        }

        srec.albedo = vec3(1);

        vec3 refracted;
        if (refract(-r.direction(), outward_normal, ni_over_nt, refracted)) {
            reflect_prob = schlick(cosine, m_ri);
        }
        else {
```

```
            reflect_prob = 1;
        }

        if (drand48() < reflect_prob) {
            srec.ray = Ray(hrec.p, reflected);
        }
        else {
            srec.ray = Ray(hrec.p, refracted);
        }

        return true;
    }

private:
    float m_ri;
};
```

`cosine` は入射角です.全反射しなければ,近似式を使ってフレネル反射率を求めます.それを使って,反射するか屈折するかを決定します.半径を負にして少し小さくした球体を追加し,すでにある誘導体材質の球体に重ねると,水泡のような表現ができます.

```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(vec3(0.1f, 0.2f, 0.5f))));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), 0.5f,
    std::make_shared<Dielectric>(1.5f)));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), -0.45f,
    std::make_shared<Dielectric>(1.5f)));
world->add(std::make_shared<Sphere>(
    vec3(0, -0.35, -0.8f), 0.15f,
    std::make_shared<Metal>(vec3(0.8f, 0.8f, 0.8f), 0.2f)));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(vec3(0.8f, 0.8f, 0.0f))));
m_world.reset(world);
```

この結果は次のようになります. (rayt117.cpp, rayt117.h)

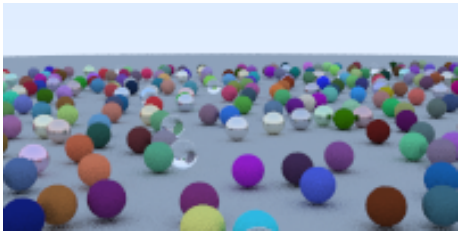


14. 沢山表示してみる

物体を多くおいてみます.また, カメラを LookAt 方式で指定します.

```
// Camera

vec3 lookfrom(13,2,3);
vec3 lookat(0,0,0);
vec3 vup(0,1,0);
float aspect = float(m_image->width()) / float(m_image->height());
m_camera = std::make_unique<Camera>(lookfrom, lookat, vup, 20, aspect);

// Shapes

ShapeList* world = new ShapeList();

int N = 11;
for (int i=-N; i<N; ++i) {
    for (int j=-N; j<N; ++j) {
        float choose_mat = drand48();
        vec3 center(i+0.9f*drand48(), 0.2f, j+0.9f*drand48());
        if (length(center-vec3(4,0.2,0)) > 0.9f) {
            if (choose_mat < 0.8f) {
                world->add(std::make_shared<Sphere>(
                    center, 0.2f,
                    std::make_shared<Lambertian>(mulPerElem(random_vector(),random_vector()))));
            }
            else if (choose_mat < 0.95f) {
                world->add(std::make_shared<Sphere>(
                    center, 0.2f,
                    std::make_shared<Metal>(0.5f * (random_vector()+vec3(1)), 0.5f*drand48())));
            }
            else {
                world->add(std::make_shared<Sphere>(
                    center, 0.2f,
                    std::make_shared<Dielectric>(1.5f)));
            }
        }
    }
}

world->add(std::make_shared<Sphere>(
    vec3(0, -1000, -1), 1000,
    std::make_shared<Lambertian>(vec3(0.5f, 0.5f, 0.5f))));
m_world.reset(world);
```

結果は次のようになります. (rayt118.cpp)



第 2 章

テクスチャとコーネルボックス

1. テクスチャ

テクスチャは物体表面の模様だったり, 反射率などのパラメータなどが格納されている画像データです.物体にテクスチャをマッピングすれば, 画像を使って様々な制御が行えるようになります.マッピング情報からテクスチャ上のカラーを参照することをサンプリングとかルックアップなどといいます.なので, テクスチャのことをルックアップテーブルなんて呼ばれることもあります.

テクスチャの用途は実に様々ですが, ここでは反射率(アルベド)を格納した画像データとして扱います.テクスチャは主に手続き型テクスチャと画像テクスチャに分かれます.それぞれプロシージャルテクスチャ, イメージテクスチャとも呼びます.今回は両方のテクスチャを作成します.まずは基本となるテクスチャの抽象クラスを定義します.

```
// rayt.h
class Texture {
public:
    virtual vec3 value(float u, float v, const vec3& p) const = 0;
};
```

`u` , `v` はテクスチャ座標です. `p` は対象ピクセルの位置情報です.物体に当たった位置のテクスチャ座標が必要なので, `HitRec` に追加します.

```
class HitRec {
public:
    float t;
    float u;
    float v;
    vec3 p;
    vec3 n;
    MaterialPtr mat;
};
```

1.1 カラーテクスチャ

最もシンプルな手続き型テクスチャで単色のテクスチャです.カラー(反射率)を持っています.

```
// rayt.h
class ColorTexture : public Texture {
public:
    ColorTexture(const vec3& c)
        : m_color(c) {
    }

    virtual vec3 value(float u, float v, const vec3& p) const override {
        return m_color;
    }
private:
    vec3 m_color;
};
```

`value` 関数では単にカラー(反射率)を返しています.このポインタを `typedef` で定義しておきます.

```
// rayt.h
class Texture;
typedef std::shared_ptr<Texture> TexturePtr;
```

次に材質にテクスチャを設定できるようにし,反射率に反映させるようにします. `Lambertian` , `Metal` のアルベドをテクスチャに変更します.

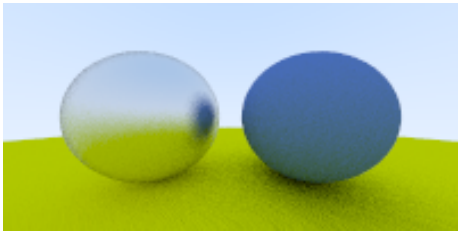
```
class Lambertian : public Material {
public:
    Lambertian(const TexturePtr& a)
        : m_albedo(a) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        vec3 target = hrec.p + hrec.n + random_in_unit_sphere();
        srec.ray = Ray(hrec.p, target - hrec.p);
        srec.albedo = m_albedo->value(hrec.u, hrec.v, hrec.p);
        return true;
    };

private:
    TexturePtr m_albedo;
};

class Metal : public Material {
public:
    Metal(const TexturePtr& a, float fuzz)
        : m_albedo(a)
        , m_fuzz(fuzz) {
    }
```

```
    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        vec3 reflected = reflect(normalize(r.direction()), hrec.n);
        reflected += m_fuzz*random_in_unit_sphere();
        srec.ray = Ray(hrec.p, reflected);
        srec.albedo = m_albedo->value(hrec.u, hrec.v, hrec.p);
        return dot(srec.ray.direction(), hrec.n) > 0;
    }

private:
    TexturePtr m_albedo;
    float m_fuzz;
};
```

球体の生成のところを書き換えます.

```
// Camera

vec3 w(-2.0f, -1.0f, -1.0f);
vec3 u(4.0f, 0.0f, 0.0f);
vec3 v(0.0f, 2.0f, 0.0f);
m_camera = std::make_unique<Camera>(u, v, w);

// Shapes

ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.1f, 0.2f, 0.5f)))));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), 0.5f,
    std::make_shared<Metal>(
        std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.8f)), 0.4f)));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.0f)))));
m_world.reset(world);
```

これを実行すると次のようになります. (rayt201.cpp, rayt201.h)



1.2 格子模様テクスチャ

これも手続き型テクスチャです.格子状の模様を生成します.

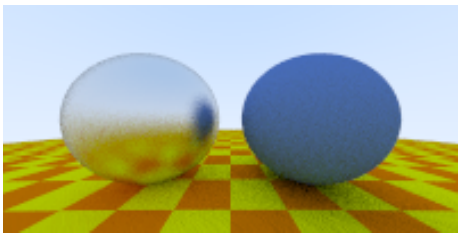
```
// rayt.h
class CheckerTexture : public Texture {
public:
    CheckerTexture(const TexturePtr& t0, const TexturePtr& t1, float freq)
        : m_odd(t0)
        , m_even(t1)
        , m_freq(freq) {
    }

    virtual vec3 value(float u, float v, const vec3& p) const override {
        float sines = sinf(m_freq*p.getX()) * sinf(m_freq*p.getY()) * sinf(m_freq*p.getZ());
        if (sines < 0) {
            return m_odd->value(u, v, p);
        }
        else {
            return m_even->value(u, v, p);
        }
    }

private:
    TexturePtr m_odd;
    TexturePtr m_even;
    float m_freq;
};
```

`ColorTexture` を2つ設定し，sin 関数を使って交互に描きます．`m_freq` は周波数で，縞模様の間隔を調整できます.このテクスチャを床に設定してみます.

```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.1f, 0.2f, 0.5f)))));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), 0.5f,
    std::make_shared<Metal>(
```

```
        std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.8f)), 0.4f)));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(
        std::make_shared<CheckerTexture>(
            std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.0f)),
            std::make_shared<ColorTexture>(vec3(0.8f, 0.2f, 0.0f)), 10.f))));
m_world.reset(world);
```

次のようになります. (rayt202.cpp)



2. 画像テクスチャ

画像テクスチャはビットマップファイルなどの画像ファイルから読み込んで, それをテクスチャとして扱います.画像テクスチャを使用するには, 当たった位置のテクスチャ座標が必要です.球体のテクスチャ座標は球状マッピングで生成します.

$$u = \frac{\phi}{2\pi}, \quad v = \frac{\theta}{\pi}.$$

方位角 ϕ, 極角 θ を計算するには位置から三角関数で求まります.

$$\begin{aligned} x &= \cos(\phi)\cos(\theta) \\ y &= \cos(\phi)\sin(\theta) \\ z &= \sin(\theta) \end{aligned}$$

そうすると

$$\phi = \tan^{-1}(y, x), \quad \theta = \sin^{-1}(z).$$

`atan2` は $[-\pi, \pi]$ の範囲を返し, `sin` は $[-\pi/2, \pi/2]$ の範囲を返します.それを $[0,1]$ にマッピングします.

$$u = 1 - \frac{(\phi + \pi)}{2\pi}, \quad v = \frac{\theta + \pi/2}{\pi}.$$

位置から球状マッピングしたテクスチャ座標を取得するコードは次のようになります.

```
// rayt.h
inline void get_sphere_uv(const vec3& p, float& u, float& v) {
    float phi = atan2(p.getZ(), p.getX());
    float theta = asin(p.getY());
```

```
    u = 1.f - (phi + PI) / (2.f * PI);
    v = (theta + PI / 2.f) / PI;
}
```

球体に当たった時にこの関数を使ってテクスチャ座標を設定します.

```
virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
    vec3 oc = r.origin() - m_center;
    float a = dot(r.direction(), r.direction());
    float b = 2.0f*dot(oc, r.direction());
    float c = dot(oc, oc) - pow2(m_radius);
    float D = b*b - 4 * a*c;
    if (D > 0) {
        float root = sqrtf(D);
        float temp = (-b - root) / (2.0f*a);
        if (temp < t1 && temp > t0) {
            hrec.t = temp;
            hrec.p = r.at(hrec.t);
            hrec.n = (hrec.p - m_center) / m_radius;
            hrec.mat = m_material;
            get_sphere_uv(hrec.n, hrec.u, hrec.v);
            return true;
        }
        temp = (-b + root) / (2.0f*a);
        if (temp < t1 && temp > t0) {
            hrec.t = temp;
            hrec.p = r.at(hrec.t);
            hrec.n = (hrec.p - m_center) / m_radius;
            hrec.mat = m_material;
            get_sphere_uv(hrec.n, hrec.u, hrec.v);
            return true;
        }
    }

    return false;
}
```

次は画像テクスチャを実装します.画像ファイルの読み込みは `stb_image.h` を使います.

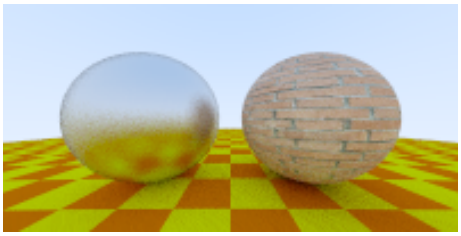
```
// rayt.h
class ImageTexture : public Texture {
    public:
        ImageTexture(const char* name) {
            int nn;
            m_texels = stbi_load(name, &m_width, &m_height, &nn, 0);
        }
```

```
        virtual ~ImageTexture() {
            stbi_image_free(m_texels);
        }

        virtual vec3 value(float u, float v, const vec3& p) const override {
            int i = (u) * m_width;
            int j = (1 - v) * m_height - 0.001;
            return sample(i,j);
        }

        vec3 sample(int u, int v) const
        {
            u = u<0 ? 0 : u >= m_width ? m_width - 1 : u;
            v = v<0 ? 0 : v >= m_height ? m_height - 1 : v;
            return vec3(
                int(m_texels[3 * u + 3 * m_width * v]) / 255.0,
                int(m_texels[3 * u + 3 * m_width * v + 1]) / 255.0,
                int(m_texels[3 * u + 3 * m_width * v + 2]) / 255.0);
        }

    private:
        int m_width;
        int m_height;
        unsigned char* m_texels;
    };
```

`stbi_load` で画像データをファイルから読み込みます. `stbi_image_free` で画像データを読み込んだデータのメモリを解放します.テクスチャ座標からカラーをサンプリングするのは `sample` 関数です.画像ファイルを用意して,画像テクスチャを使用してみます.

```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0.6, 0, -1), 0.5f,
    std::make_shared<Lambertian>(
        std::make_shared<ImageTexture>("./assets/brick_diffuse.jpg"))));
world->add(std::make_shared<Sphere>(
    vec3(-0.6, 0, -1), 0.5f,
    std::make_shared<Metal>(
        std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.8f)), 0.4f)));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(
        std::make_shared<CheckerTexture>(
            std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.0f)),
            std::make_shared<ColorTexture>(vec3(0.8f, 0.2f, 0.0f)), 10.f))));
m_world.reset(world);
```

レンダリングすると次のような画像になります. (rayt203.cpp, rayt203.h)



3. 発光

物体から光を放出するような材質を作ります.この材質の物体は照明と考えることができます.材質クラスに発光色を返す仮想関数を追加して, 発光する材質はオーバーライドします.

```
class Material {
public:
    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const = 0;
    virtual vec3 emitted(const Ray& r, const HitRec& hrec) const { return vec3(0); }
};
```

既存の材質クラスには変更を加えないように純粋仮想関数ではなくて黒を返すようにします.発光材質を `DiffuseLight` クラスとします. 発光色はテクスチャで設定します.

```
class DiffuseLight : public Material {
public:
    DiffuseLight(const TexturePtr& emit)
        : m_emit(emit) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        return false;
    }

    virtual vec3 emitted(const Ray& r, const HitRec& hrec) const override {
        return m_emit->value(hrec.u, hrec.v, hrec.p);
    }

private:
    TexturePtr m_emit;
};
```

散乱はしないので, `scatter` 関数は何もせずに `false` を返します.あとは反射した光に発光を加えます.なので, `color` を書き換えます.

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
```

```
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            return emitted + mulPerElem(srec.albedo, color(srec.ray, world, depth + 1));
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

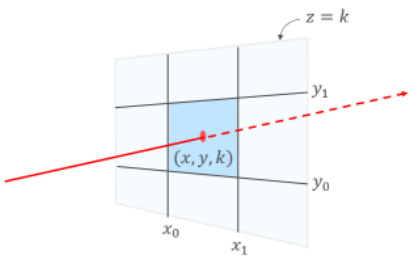
物体を発光させてみましょう.次のようなコードになります.

```
ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0, 0, -1), 0.5f,
    std::make_shared<DiffuseLight>(
        std::make_shared<ColorTexture>(vec3(1)))));
world->add(std::make_shared<Sphere>(
    vec3(0, -100.5, -1), 100,
    std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.8f)))));
m_world.reset(world);
```

実行結果は次のようになります. (rayt204.cpp)

4. 四角形

新しい物体四角形を追加します.これは軸に平行な四角形とします.なので, XY 軸, XZ 軸, YZ 軸に平行な四角形です.例えば, XY 軸の平面上にある四角形は図のようなものです.



光線を四角形に飛ばしたときに，当たるかどうかの判定を考えます.それにはまず光線と平面との交点を求めます.光線と平面が平行でない限り，光線の直線上のどこかで平面と当たります.光線の方程式は以下のとおりです.

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

各ベクトルの Z 要素で表すと

$$\vec{p}_z(t) = \vec{o}_z + t\vec{d}_z.$$

$z = k$ なので

$$t = \frac{k - \vec{o}_z}{\vec{d}_z}.$$

上の式で t が求まるので，光線の方程式から平面上の x と y を求められます.

$$x = \vec{o}_x + t\vec{d}_x, \quad y = \vec{o}_y + t\vec{d}_y.$$

もし $x_0 < x < x_1$ かつ $y_0 < y < y_1$ なら当たっていることになります.これは XY 軸の四角形ですが，XZ 軸，YZ 軸も同様に考えることができます.これをもとに `Rect` クラスを実装します.

```
class Rect : public Shape {
public:
    enum AxisType {
        kXY = 0,
        kXZ,
        kYZ
    };
    Rect() {}
    Rect(float x0, float x1, float y0, float y1, float k, AxisType axis, const MaterialPtr& m)
        : m_x0(x0)
        , m_x1(x1)
        , m_y0(y0)
        , m_y1(y1)
        , m_k(k)
        , m_axis(axis)
        , m_material(m) {
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {

        int xi, yi, zi;
        vec3 axis;
        switch (m_axis) {
        case kXY: xi = 0; yi = 1; zi = 2; axis = vec3::zAxis(); break;
        case kXZ: xi = 0; yi = 2; zi = 1; axis = vec3::yAxis(); break;
        case kYZ: xi = 1; yi = 2; zi = 0; axis = vec3::xAxis(); break;
        }
```

```
        float t = (m_k - r.origin()[zi]) / r.direction()[zi];
        if (t < t0 || t > t1) {
            return false;
        }

        float x = r.origin()[xi] + t*r.direction()[xi];
        float y = r.origin()[yi] + t*r.direction()[yi];
        if (x < m_x0 || x > m_x1 || y < m_y0 || y > m_y1) {
            return false;
        }

        hrec.u = (x - m_x0) / (m_x1 - m_x0);
        hrec.v = (y - m_y0) / (m_y1 - m_y0);
        hrec.t = t;
        hrec.mat = m_material;
        hrec.p = r.at(t);
        hrec.n = axis;
        return true;
    }

private:
    float m_x0, m_x1, m_y0, m_y1, m_k;
    AxisType m_axis;
    MaterialPtr m_material;
};
```

XY 軸, XZ 軸, YZ 軸は指定できるようにしました. `hit` 関数では平行な軸によって参照するベクトルの要素を切り替えています.テクスチャ座標は次のような計算式で求めています.

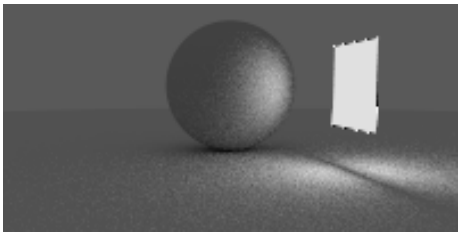
$$u = \frac{x - x_0}{x_1 - x_0}, \quad v = \frac{y - y_0}{y_1 - y_0}.$$

それでは四角形をレンダリングしてみます.

```
// Camera

vec3 lookfrom(13,2,3);
vec3 lookat(0, 1, 0);
vec3 vup(0, 1, 0);
float aspect = float(m_image->width()) / float(m_image->height());
m_camera = std::make_unique<Camera>(lookfrom, lookat, vup, 30, aspect);

// Shapes

ShapeList* world = new ShapeList();
world->add(std::make_shared<Sphere>(
    vec3(0, 2, 0), 2,
```

```
    std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.5f, 0.5f, 0.5f))));
world->add(std::make_shared<Sphere>(
    vec3(0, -1000, 0), 1000,
    std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.8f, 0.8f, 0.8f))));
world->add(std::make_shared<Rect>(
    3,5,1,3,-2,Rect::kXY,
    std::make_shared<DiffuseLight>(
        std::make_shared<ColorTexture>(vec3(4)))));
m_world.reset(world);
```

あと, サンプリング数も 500 に設定しています.これは次のようになります. (rayt205.cpp)



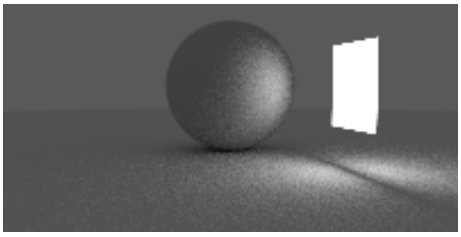
四角形のまわりに白いつぶつぶのようなものがあります.また, 四角形自身も若干暗く見えませんか? 発光の強さを 1 以上にしたのでオーバーフローを起こしているようです.内部では float(32bit) で値を計算していますが, unsigned char(8bit, 範囲は [0,255]) に変換しているときに起きているようです.コンピュータグラフィックスでは, このように [0,255] = [0.0,1.0] の範囲のことを **LDR (Low Dynamic Range)**といい, それ以上の範囲のことを **HDR (High Dynamic Range)**といいます.内部では HDR で計算していることになります. HDR から LDR に変換することトーンマッピングと呼ぶことがあります.今回は, LDR に変換するときに単純にクランプ処理([0,1] に切り詰める)をします. `ImageFilter` クラスを継承して, `TonemapFilter` として実装します.

```
// rayt.h
class TonemapFilter : public ImageFilter {
public:
    TonemapFilter() {}
    virtual vec3 filter(const vec3& c) const override {
        return minPerElem(maxPerElem(c, Vector3(0.f)), Vector3(1.f));;
    }
};
```

`Image` クラスで `TonemapFilter` をフィルターリストに追加します.

```
Image(int w, int h) {
    m_width = w;
    m_height = h;
    m_pixels.reset(new rgb[m_width*m_height]);
    m_filters.push_back(std::make_unique<GammaFilter>(GAMMA_FACTOR));
    m_filters.push_back(std::make_unique<TonemapFilter>());
}
```

もう一度レンダリングすると次のようになります. (rayt206.h)



つぶつぶが無くなっていて, 明るくなっているのがわかると思います.

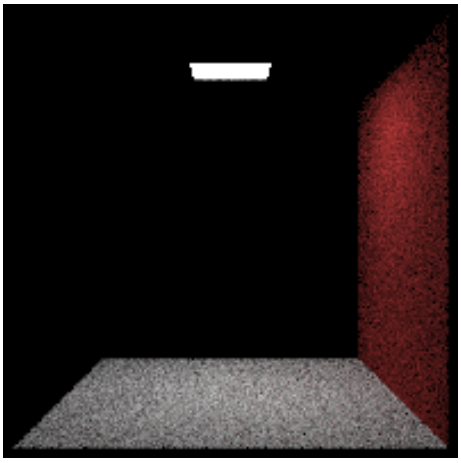
5. コーネルボックス

発光材質と四角形を追加したので, 有名なコーネルボックスを作ってみます.

```
void build() {

    m_backColor = vec3(0);

    // Camera

    vec3 lookfrom(278,278,-800);
    vec3 lookat(278, 278, 0);
    vec3 vup(0, 1, 0);
    float aspect = float(m_image->width()) / float(m_image->height());
    m_camera = std::make_unique<Camera>(lookfrom, lookat, vup, 40, aspect);

    // Shapes

    MaterialPtr red = std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.65f, 0.05f, 0.05f)));
    MaterialPtr white = std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.73f)));
    MaterialPtr green = std::make_shared<Lambertian>(
        std::make_shared<ColorTexture>(vec3(0.12f, 0.45f, 0.15f)));
    MaterialPtr light = std::make_shared<DiffuseLight>(
        std::make_shared<ColorTexture>(vec3(15.0f)));

    ShapeList* world = new ShapeList();
    world->add(
        std::make_shared<Rect>(
        0,555,0,555,555,Rect::kYZ,green));
    world->add(
        std::make_shared<Rect>(
        0,555,0,555,0,Rect::kYZ,red));
    world->add(
```

```
        std::make_shared<Rect>(
        213,343,227,332,554,Rect::kXZ,light));
    world->add(
        std::make_shared<Rect>(
        0, 555, 0, 555, 555, Rect::kXZ, white));
    world->add(
        std::make_shared<Rect>(
        0,555,0,555,0,Rect::kXZ,white));
    world->add(
        std::make_shared<Rect>(
        0,555,0,555,555,Rect::kXY,white));
    m_world.reset(world);
}
```

この結果は次のようになります. (rayt207.cpp)



ノイズのようなものが目立ちますが, これは光源が小さいので, 光線が光源に到達しないところが出てきているからです.また, 壁がいくつか見えません.これは壁の法線が反対側を向いているからです.そのため光線が当たったら, 法線を反対方向に向かせるようにします. `Shape` クラスを継承した `FlipNormals` というクラスを追加します.これは別の `Shape` クラスインスタンスを持っており, `hit` 関数では, そのインスタンスの `hit` 関数を呼んで法線を反転させます.

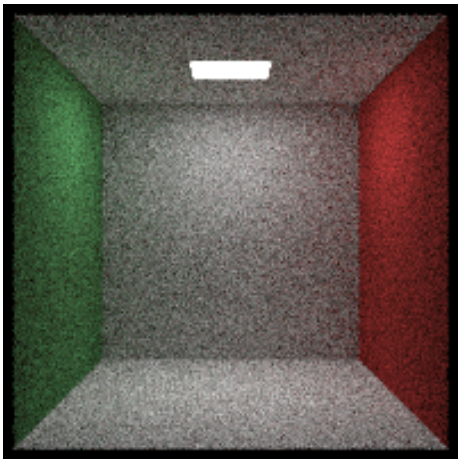
```
class FlipNormals : public Shape {
public:
    FlipNormals(const ShapePtr& shape)
        : m_shape(shape) {
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
        if (m_shape->hit(r, t0, t1, hrec)) {
            hrec.n = -hrec.n;
            return true;
        }
        else {
            return false;
```

```
        }
    }

private:
    ShapePtr m_shape;
};
```

このような仕組みをデザインパターンでは **Decorator** パターンといいます.これを使って, いくつかの四角形の法線を反転させます.

```
ShapeList* world = new ShapeList();
world->add(
    std::make_shared<FlipNormals>(
        std::make_shared<Rect>(
        0,555,0,555,555,Rect::kYZ,green)));
world->add(
    std::make_shared<Rect>(
    0,555,0,555,0,Rect::kYZ,red));
world->add(
    std::make_shared<Rect>(
    213,343,227,332,554,Rect::kXZ,light));
world->add(
    std::make_shared<FlipNormals>(
        std::make_shared<Rect>(
        0, 555, 0, 555, 555, Rect::kXZ, white)));
world->add(
    std::make_shared<Rect>(
    0,555,0,555,0,Rect::kXZ,white));
world->add(
    std::make_shared<FlipNormals>(
        std::make_shared<Rect>(
        0,555,0,555,555,Rect::kXY,white)));
m_world.reset(world);
```

結果は次のようになります. (rayt208.cpp)



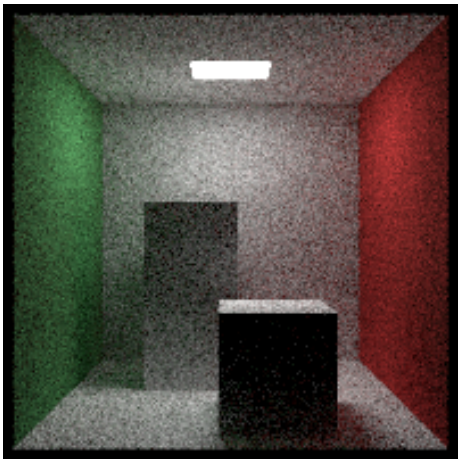
全部の壁が見えるようになったので，箱を追加しましょう．箱は新しい物体として追加します．これは四角形の組み合わせで実装できます．

```
class Box : public Shape {
public:
    Box() {}
    Box(const vec3& p0, const vec3& p1, const MaterialPtr& m)
        : m_p0(p0)
        , m_p1(p1)
        , m_list(std::make_unique<ShapeList>()) {

        ShapeList* l = new ShapeList();
        l->add(std::make_shared<Rect>(
            p0.getX(), p1.getX(), p0.getY(), p1.getY(), p1.getZ(), Rect::kXY, m));
        l->add(std::make_shared<FlipNormals>(std::make_shared<Rect>(
            p0.getX(), p1.getX(), p0.getY(), p1.getY(), p0.getZ(), Rect::kXY, m)));
        l->add(std::make_shared<Rect>(
            p0.getX(), p1.getX(), p0.getZ(), p1.getZ(), p1.getY(), Rect::kXZ, m));
        l->add(std::make_shared<FlipNormals>(std::make_shared<Rect>(
            p0.getX(), p1.getX(), p0.getZ(), p1.getZ(), p0.getY(), Rect::kXZ, m)));
        l->add(std::make_shared<Rect>(
            p0.getY(), p1.getY(), p0.getZ(), p1.getZ(), p1.getX(), Rect::kYZ, m));
        l->add(std::make_shared<FlipNormals>(std::make_shared<Rect>(
            p0.getY(), p1.getY(), p0.getZ(), p1.getZ(), p0.getX(), Rect::kYZ, m)));
        m_list.reset(l);
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
        return m_list->hit(r, t0, t1, hrec);
    }

private:
    vec3 m_p0, m_p1;
    std::unique_ptr<ShapeList> m_list;
};
```

シーンに箱を2つ追加します．

```
world->add(
    std::make_shared<Box>(vec3(130, 0, 65), vec3(295,165,230), white));
world->add(
    std::make_shared<Box>(vec3(265, 0, 295), vec3(430, 330, 460), white));
```

結果は次のようになります．(rayt209.cpp)



法線の反転と同じ方法で,移動と回転の機能を追加します.それぞれ `Translate` と `Rotate` クラスとします.

```
class Translate : public Shape {
public:
    Translate(const ShapePtr& sp, const vec3& displacement)
        : m_shape(sp)
        , m_offset(displacement) {
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
        Ray move_r(r.origin() - m_offset, r.direction());
        if (m_shape->hit(move_r, t0, t1, hrec)) {
            hrec.p += m_offset;
            return true;
        }
        else {
            return false;
        }
    }

private:
    ShapePtr m_shape;
    vec3 m_offset;
};

class Rotate : public Shape {
public:
    Rotate(const ShapePtr& sp, const vec3& axis, float angle)
        : m_shape(sp)
        , m_quat(Quat::rotation(radians(angle), axis)) {
    }

    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const override {
        Quat revq = conj(m_quat);
        vec3 origin = rotate(revq, r.origin());
```

```
        vec3 direction = rotate(revq, r.direction());
        Ray rot_r(origin, direction);
        if (m_shape->hit(rot_r, t0, t1, hrec)) {
            hrec.p = rotate(m_quat, hrec.p);
            hrec.n = rotate(m_quat, hrec.n);
            return true;
        }
        else {
            return false;
        }
    }

private:
    ShapePtr m_shape;
    Quat m_quat;
};
```

このような移動や回転をさせるようなことをアフィン変換といいます.基本は変換する前の状態で当たり判定を行い,結果に対してアフィン変換することで実装することができます.回転はクォータニオンを用いて処理しています.

Decorator パターンは強力ですが,階層が多くなってしまい,コードが見づらくなってしまいます.例えば,法線反転・移動・回転をする四角形の場合,次のようになります.

```
world->add(
    std::make_shared<Translate>(
        std::make_shared<Rotate>(
            std::make_shared<FlipNormals>(
                std::make_shared<Rect>(213, 343, 227, 332, 500, Rect::kXZ, red)),
            vec3(0,1,0), 45),
        vec3(0,0,0)));
```

個人の好みなので,もしかすると上記のコードで十分かもしれません.個人的に好きではないので,少し改善してみます.`std::make_shared` を書くのも面倒ですし,この書き方だと適用される順番が逆順でわかりづらい印象があります.そこで `ShapeBuilder` クラスを導入します.このメンバ関数はすべて自己参照を返すので,メンバ関数を連続で呼び出すことができます.これを使って,物体と法線反転などの追加を行えるようにします.

```
class ShapeBuilder {
public:
    ShapeBuilder() {}
    ShapeBuilder(const ShapePtr& sp)
        : m_ptr(sp) {
    }

    ShapeBuilder& reset(const ShapePtr& sp) {
        m_ptr = sp;
        return *this;
```

```
    }

    ShapeBuilder& sphere(const vec3& c, float r, const MaterialPtr& m) {
        m_ptr = std::make_shared<Sphere>(c, r, m);
        return *this;
    }

    ShapeBuilder& rect(float x0, float x1, float y0, float y1, float k, Rect::AxisType axis, const MaterialPtr& m)
    {
        m_ptr = std::make_shared<Rect>(x0, x1, y0, y1, k, axis, m);
        return *this;
    }
    ShapeBuilder& rectXY(float x0, float x1, float y0, float y1, float k, const MaterialPtr& m) {
        m_ptr = std::make_shared<Rect>(x0, x1, y0, y1, k, Rect::kXY, m);
        return *this;
    }
    ShapeBuilder& rectXZ(float x0, float x1, float y0, float y1, float k, const MaterialPtr& m) {
        m_ptr = std::make_shared<Rect>(x0, x1, y0, y1, k, Rect::kXZ, m);
        return *this;
    }
    ShapeBuilder& rectYZ(float x0, float x1, float y0, float y1, float k, const MaterialPtr& m) {
        m_ptr = std::make_shared<Rect>(x0, x1, y0, y1, k, Rect::kYZ, m);
        return *this;
    }

    ShapeBuilder& rect(const vec3& p0, const vec3& p1, float k, Rect::AxisType axis, const MaterialPtr& m) {
        switch (axis) {
        case Rect::kXY:
            m_ptr = std::make_shared<Rect>(
                p0.getX(), p1.getX(), p0.getY(), p1.getY(), k, axis, m);
            break;
        case Rect::kXZ:
            m_ptr = std::make_shared<Rect>(
                p0.getX(), p1.getX(), p0.getZ(), p1.getZ(), k, axis, m);
            break;
        case Rect::kYZ:
            m_ptr = std::make_shared<Rect>(
                p0.getY(), p1.getY(), p0.getZ(), p1.getZ(), k, axis, m);
            break;
        }
        return *this;
    }
    ShapeBuilder& rectXY(const vec3& p0, const vec3& p1, float k, const MaterialPtr& m) {
        return rect(p0, p1, k, Rect::kXY, m);
    }
    ShapeBuilder& rectXZ(const vec3& p0, const vec3& p1, float k, const MaterialPtr& m) {
        return rect(p0, p1, k, Rect::kXZ, m);
```

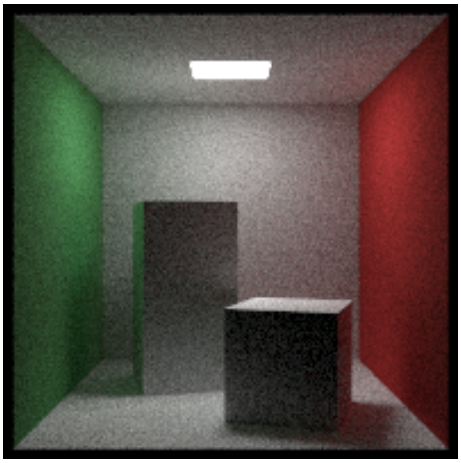
```
    }
    ShapeBuilder& rectYZ(const vec3& p0, const vec3& p1, float k, const MaterialPtr& m) {
        return rect(p0, p1, k, Rect::kYZ, m);
    }

    ShapeBuilder& box(const vec3& p0, const vec3& p1, const MaterialPtr& m) {
        m_ptr = std::make_shared<Box>(p0, p1, m);
        return *this;
    }

    ShapeBuilder& flip() {
        m_ptr = std::make_shared<FlipNormals>(m_ptr);
        return *this;
    }

    ShapeBuilder& translate(const vec3& t) {
        m_ptr = std::make_shared<Translate>(m_ptr, t);
        return *this;
    }

    ShapeBuilder& rotate(const vec3& axis, float angle) {
        m_ptr = std::make_shared<Rotate>(m_ptr, axis, angle);
        return *this;
    }

    const ShapePtr& get() const { return m_ptr; }

private:
    ShapePtr m_ptr;
};
```

これを使ってコーネルボックスを作成するようにすると

```
ShapeList* world = new ShapeList();
ShapeBuilder builder;
world->add(builder.rectYZ(0, 555, 0, 555, 555, green).flip().get());
world->add(builder.rectYZ(0, 555, 0, 555, 0, red).get());
world->add(builder.rectXZ(213, 343, 227, 332, 554, light).get());
world->add(builder.rectXZ(0, 555, 0, 555, 555, white).flip().get());
world->add(builder.rectXZ(0, 555, 0, 555, 0, white).get());
world->add(builder.rectXY(0, 555, 0, 555, 555, white).flip().get());
world->add(builder.box(vec3(0), vec3(165), white)
    .rotate(vec3::yAxis(), -18)
    .translate(vec3(130, 0, 65))
    .get());
world->add(builder.box(vec3(0), vec3(165, 330, 165), white)
    .rotate(vec3::yAxis(), 15)
```

```
    .translate(vec3(265, 0, 295))
    .get());
m_world.reset(world);
```

どうでしょうか？ こっちの方がスッキリしている感じがします.これをレンダリングすると次のようになります.下の画像はサンプル数は 2000 です.(rayt210.cpp)



第 3 章

モンテカルロレイトレーシング

1. 光学

ここからはモンテカルロ法を使ってレイトレーシングを行う手法を説明します.そのためにまずは少し光学について触れておきます.これまで行ってきた光線を飛ばして光伝達をシミュレーションするような方法をレイトレーシングといいました.これは幾何光学という分野になります.なので,レイトレーシングは幾何光学で説明できる物理現象(例えば反射や屈折)を実現することが出来ます.それとは別に,光は電磁波の一種なので,波動の特性を持っています.光の波動による物理現象(例えば回折や干渉)は波動光学の分野になります.もちろん電磁波なので電磁学にも関連しています.例えば偏光がこの分野になります.光の物理現象をシミュレーションするには,こういった分野がすべて必要になってくるのですが,実際は計算が複雑になったり,よくわからん(切実)ので,基本的に幾何光学だけを考えます.フレネルの方程式のときに偏光が出てきましたが,無視して考えていたのはそういうことです.そのため,写実的なものをレンダリングするためには,まず幾何光学について知っておく必要があります.

2. 光の物理量

コンピュータグラフィックスにおいて,光の最も小さい単位は光子です.これをフォトン(**photon**)といいます.光のエネルギーというのはこの光子が集まったもので,放射エネルギーといいます.細かいことをいうと波長が関係してくるのですが,ここでは割愛します.

この放射エネルギーの時間的な変化を表したものを放射束といいます.これは放射エネルギーの仕事量のことでもあるので単位はワット(**W**)です.電球とかで 60W とかありますよね? あれです.

これまで計算してきた数値は光学においてどのような値(量)なのでしょうか.それは放射束です.光伝達というのは放射束を輸送することを指します.

余談ですが,人間の眼が知覚する光の量を測光値といい,この分野を測光学といいます.測光学における放射束を光束といいます.

4. モンテカルロレイトレーシング

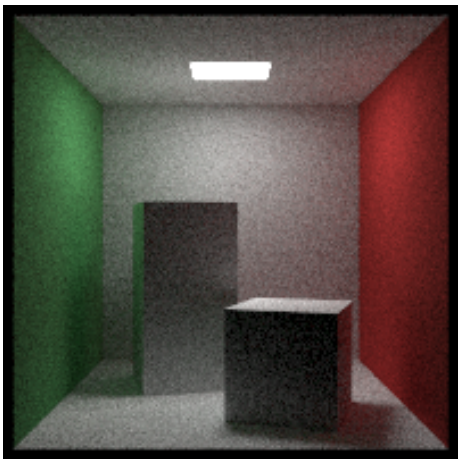
古典的なレイトレーシングは,カメラから光線を飛ばして鏡面反射させていました.その後,鏡面反射の方向の周りに向かって複数の光線を飛ばして計算するように改善されました.これを分散レイトレーシングといいます.これから実装していくモンテカルロレイトレーシングはこの分散レイトレーシングをさらに改善させたものです.

光源から放射された放射束は反射を繰り返して物体表面に入射します.物体の表面は拡散反射や鏡面反射などを行ってあらゆる方向に放射束を反射します.そして,物体表面のある点を考えると,半球上のあらゆるところから放射束が入ってくることになります.つまり,半球上から入ってくるすべての放射束を積分する必要があります.これをシミュレーションしようとするのは困難なので,統計的にサンプリングを行って,結果を推定します.そこでモンテカルロ法を使います.積分範囲内で無作為に光線を発生させてサンプリングし,平均値を求めて積分の推定

値とします.このように表面のある点に入ってくる放射束をモンテカルロ法で求めて, 反射や屈折などを計算する方法をモンテカルロレイトレーシングといいます.

4.1 モンテカルロ法について

モンテカルロ法, またそれに関する確率と統計については CG のための確率・統計入門を参照してください.また, これから立体角といった新しい用語が出てきます.それらについては基礎からはじめる物理ベースレンダリングで説明していますので, 分からない場合は参照してください.これからはモンテカルロ法に基づいて, 各種の物理現象をシミュレーションしていきます.前回の最後でコーネルボックスをレンダリングしました.そのコーネルボックスを使ってモンテカルロレイトレーシングを実装していきます.



5. 光の散乱

材質のところで散乱について説明しました.ここでもう一度定義すると, 散乱とは物体表面に光線が当たったときに, 光線の向きが変わる現象のことです.また, 物体表面で吸収されずに, 反射される光の割合を反射率(アルベド)といいます.ここで, 光とは放射束のことです.

散乱したときの方向分布は, 立体角の確率密度関数(以降, pdf と記述します)によって表され, これを $s(direction)$ と表します.この pdf は入射角に依存していて, グレージング角に近づくほど小さくなっていきます.

ある表面上の点に入射してくる全放射束は

$$color = \int_{\Omega} albedo \cdot s(direction) \cdot color(direction).$$

ここで Ω は半球を表します.上記のように半球上の全ての方向から入射した放射束を積分したものを放射照度(**Irradiance**)といいます.これをモンテカルロ法で解くと

$$color = \sum_{i=1}^{N} \frac{albedo \cdot s(direction) \cdot color(direction)}{p(direction)}.$$

$color(direction)$ は $direction$ 方向から入射してくる放射束で, これを放射輝度(**Radiance**)といいます. $p(direction)$ は無作為に生成した方向の pdf です.

ランバート面では $s(direction)$ は $\cos(\theta)$ に比例します. pdf は積分すると 1 になるので, $\cos(\theta) < 1$ を満たします.また, 角度が大きくなる(グレージング角に近づく)と値は減少していき, 90° のときは $s(direction) = 0$ となります. $\cos(\theta)$ を半球積分すると π になりますので,

$$s(direction) = \frac{\cos(\theta)}{\pi}.$$

$s(direction)$ と $p(direction)$ が同じなら

$$color = albedo \cdot color(direction),$$

となり, 今までの計算と変わらないことになります.

余談ですが, 物理ベースレンダリングとかでよくでてくる双方向反射率分布関数(**Bidirectional reflectance distribution function: BRDF**)との関係は

$$BRDF = \frac{albedo \cdot s(direction)}{\cos\theta},$$

であり, ランバート面の $s(direction) = \frac{\cos\theta}{\pi}$ を上の式に代入すると

$$BRDF = \frac{albedo}{\pi}.$$

これはランバートの BRDF と一致します.また, 関与媒質において $s(direction)$ は位相関数(phase function)に相当します.

5. 重点的サンプリング (Importance sampling)

5.1 散乱 pdf と重み

材質クラスに散乱 pdf を返す関数 `scattering_pdf` を追加します.デフォルトで 0 を返します.

```
virtual float scattering_pdf(const Ray& r, const HitRec& hrec) const { return 0; }
```

ランバート材質の `scattering_pdf` は $\cos(\theta)/\pi$ を返します.また, 裏面に当たったときは散乱しないよう(つまり 0)にしています.

```
virtual float scattering_pdf(const Ray& r, const HitRec& hrec) const override {
    return std::max(dot(hrec.n, normalize(r.direction())), 0.0f) / PI;
}
```

また, モンテカルロ法で積分するために $p(direction)$ が必要です.この $p(direction)$ は確率の重みを表しているともいえます. `ScatterRec` に pdf_value を追加し, `scatter` 関数で値を計算します.

```
class ScatterRec {
public:
    Ray ray;
    vec3 albedo;
    float pdf_value;
};
```

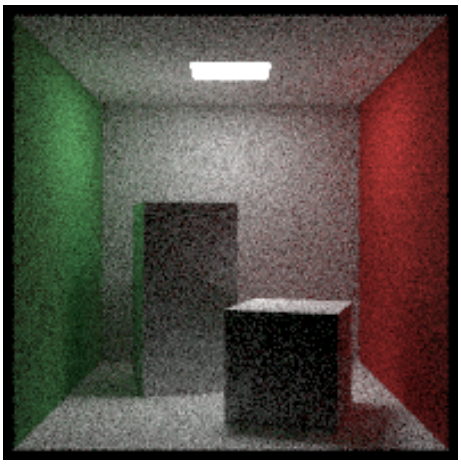
```
virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
    vec3 target = hrec.p + hrec.n + random_in_unit_sphere();
    srec.ray = Ray(hrec.p, normalize(target - hrec.p));
    srec.albedo = m_albedo->value(hrec.u, hrec.v, hrec.p);
```

```
    srec.pdf_value = dot(hrec.n, srec.ray.direction()) / PI;
    return true;
};
```

`color` 関数で, 散乱 pdf の値とアルベド, 放射輝度 (`color` 関数の返り値)の積を求めて, それに `ScatterRec.pdf_value` で除算します.

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec) && srec.pdf_value > 0) {
            float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
            vec3 albedo = srec.albedo * spdf_value;
            return emitted + mulPerElem(albedo, color(srec.ray, world, depth + 1)) / srec.pdf_value;
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

これを実行すると次のようになります.(rayt301.cpp)



ノイズが多くありますね.これからこのノイズを無くしていきます.

5.2 半球上の無作為な方向の生成

重点的サンプリングを行うには表面の法線を中心に半球上の無作為な方向を生成する必要があります.まずは z 軸が法線方向, θ が法線方向からの角度を表しているとします.方向の作成には方位角 ϕ の確率変数 r_1 と, 極角の確率変数 r_2 が必要です.最初にどちらも一様分布に従う確率変数を求めて, それから確率分布が cos になるような極角の確率変数 r_2 を求めます.

方位角 ϕ の区間は $[0, 2\pi]$ なので一様分布に従う pdf は

$$p(\phi) = \frac{1}{2\pi}.$$

確率変数 r_1 は

$$r_1 = \int_0^{\phi} \frac{1}{2\pi} = \frac{\phi}{2\pi}.$$

ϕ について解くと

$$\phi = 2\pi r_1$$

極角 θ の pdf は $p(direction) = f(\theta)$, 立体角 $\sin\theta d\theta d\phi$ なので, 確率変数 r_2 は

$$\begin{aligned} r_2 &= \int_0^{2\pi} \int_0^{\theta} f(\theta) \sin\theta d\theta d\phi \\ &= 2\pi \int_0^{\theta} f(\theta) \sin\theta d\theta \end{aligned}$$

単位球の面積は 4π なので, $f(\theta)$ は $\frac{1}{4\pi}$ となります.

$$\begin{aligned} r_2 &= 2\pi \int_0^{\theta} \frac{1}{4\pi} \sin(t) dt \\ &= \frac{1}{2} \int_0^{\theta} \sin(t) dt \\ &= \frac{1}{2} [-\cos]_0^{\theta} \\ &= -\frac{\cos(\theta)}{2} - \left(-\frac{\cos(0)}{2}\right) \\ &= -\frac{\cos(\theta)}{2} + \frac{1}{2} \\ &= \frac{(1 - \cos(\theta))}{2}. \end{aligned}$$

$\cos(\theta)$ について解くと

$$\cos(\theta) = 1 - 2r_2.$$

極座標(ϕ, θ)からデカルト座標(x, y, z)に変換するときは次の式を使います.

$$\begin{aligned} x &= \cos(\phi)\sin(\theta) \\ y &= \sin(\phi)\sin(\theta) \\ z &= \cos(\theta). \end{aligned}$$

$\cos^2\theta + \sin^2\theta + 1$ を使って変形し, 代入すると

$$
\begin{aligned}
x &= \cos(2\pi r_1)\sqrt{1-(1-2r_2)^2} \\
y &= \sin(2\pi r_2)\sqrt{1-(1-2r_2)^2} \\
z &= 1-2r_2.
\end{aligned}
$$

平方根の項を整理すると

$$
\begin{aligned}
\sqrt{1-(1-2r_2)^2} &= \sqrt{1-(1-4r_2+4_r^2)} \\
&= \sqrt{4r_2-4r_2^2} \\
&= \sqrt{4(r_2-r_2^2)} \\
&= 2\sqrt{r_2-r_2^2} \\
&= 2\sqrt{r_2(1-r_2)}.
\end{aligned}
$$

なので,

$$
\begin{aligned}
x &= \cos(2\pi r_1)2\sqrt{r_2(1-r_2)} \\
y &= \sin(2\pi r_1)2\sqrt{r_2(1-r_2)} \\
z &= 1-2r_2.
\end{aligned}
$$

これは単位球上の無作為な方向なので, 半球上に制限します.半球上で一様分布な pdf は $p(direction) = \frac{1}{2\pi}$ なので,

$$
\begin{aligned}
r_2 &= 2\pi \int_0^\theta \frac{1}{2\pi}\sin(t)dt \\
&= \int_0^\theta \sin(t)dt \\
&= [-cos(t)]_0^\theta \\
&= -\cos(\theta) - (-\cos(0)) \\
&= 1-\cos(\theta).
\end{aligned}
$$

$\cos(\theta)$ について解くと

$$
\cos(\theta) = 1-r_2.
$$

デカルト座標に変換すると

$$
\begin{aligned}
x &= \cos(2\pi r_1)\sqrt{r_2(2-r_2)} \\
y &= \sin(2\pi r_1)\sqrt{r_2(2-r_2)} \\
z &= 1-r_2.
\end{aligned}
$$

次に pdf が $p(direction) = \frac{\cos(\theta)}{\pi}$ の方向を生成します.

$$
\begin{aligned}
r_2 &= 2\pi \int_0^{\theta} \left(\frac{\cos(t)}{\pi} \right) \sin(t) dt \\
&= 2 \int_0^{\theta} \cos(t) \sin(t) dt
\end{aligned}
$$

$$
t = \cos(\theta), \frac{dt}{d\theta} = -\sin(\theta), dt = -\sin\theta d\theta
$$

で置換積分します.

$$
\begin{aligned}
r_2 &= -2 \int_0^{\cos\theta} t dt \\
&= \left[\frac{1}{2} t^2 \right]_0^{\cos\theta} \\
&= -2 \left\{ \left(\frac{1}{2} \cos^2(\theta) \right) - \left(\frac{1}{2} \cos^2(0) \right) \right\} \\
&= -(\cos^2(\theta) - 1) \\
&= 1 - \cos^2(\theta)
\end{aligned}
$$

よって

$$
\cos(\theta) = \sqrt{1 - r_2}.
$$

デカルト座標で表すと

$$
\begin{aligned}
z &= \cos(\theta) = \sqrt{1 - r_2} \\
x &= \cos(\phi)\sin(\theta) = \cos(2\pi r_1)\sqrt{1 - z^2} = \cos(2\pi r_1)\sqrt{r_2} \\
y &= \sin(\phi)\sin(\theta) = \cos(2\pi r_1)\sqrt{1 - z^2} = \sin(2\pi r_1)\sqrt{r_2}.
\end{aligned}
$$

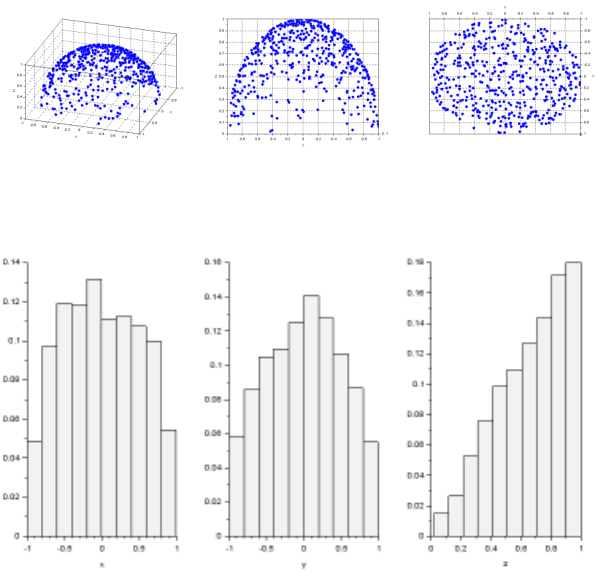
これをコードに書き起こしたものが以下になります.

```
// rayt.h
inline vec3 random_cosine_direction() {
    float r1 = drand48();
    float r2 = drand48();
    float z = sqrt(1.f - r2);
    float phi = PI2*r1;
    float x = cos(phi) * sqrt(r2);
    float y = sin(phi) * sqrt(r2);
    return vec3(x, y, z);
}
```

この関数がどのようなベクトルを作成するのか統計的に調べてみます. Scilab というソフトウェアで上記のベクトルを適当な個数作成してプロットしてみました.コマンドスクリプトは次のようになっています.

```
t=rand(2,500);
x=cos(2*%pi*t(1,:)) .* sqrt(t(2,:));
y=sin(2*%pi*t(1,:)) .* sqrt(t(2,:));
z=sqrt(1-t(2,:));
scatter3(x,y,z,"fill");
scf();
subplot(1,3,1);
    histplot(10,x,1);
    xlabel("x");
subplot(1,3,2);
    histplot(10,y);
    xlabel("y");
subplot(1,3,3);
    histplot(10,z);
    xlabel("z");
```

結果は次の通りです.



一段目の一番左を見ると半球上に点が生成されているのがわかります.一段目の中央, XZ のプロットを見てみると Z 値が高くなると点の数も増えているように見えませんか? ちょっとわかりづらいかもしれません.そんなときは2段目のヒストグラムを見てみます.一番右が Z 値ですが, 段階的になっていることがわかります.つまり, Z 値が高くなるほど多くなっています. Z と比べて XY はどうでしょうか. l 段目の一番右を見てみると円の中で一様分布しているように見えます.先ほどの関数はこのような分布をするベクトルを無作為に生成します.

5.3 正規直交基底 (Orhto-normal bases)

z 方向を中心とした方向を無作為に作成できるようになったので, 任意の方向(例えば表面の法線)を中心にできるようにします.そのため, 互いに直交する3つの単位ベクトルを決定し, 基底変換を行います.基底変換はカメラのときにもやったので, ここでは詳しく説明しません.今回は法線から正規直交基底を作成します.カメラのときには `vup` を指定していましたが, ここでは基本的に Y 方向のベクトルを使いますが, 法線と平行の場合に正しく計算できないので, その時は X 方向のベクトルを使います.正規直交なので, 基底ベクトルは正規化します.

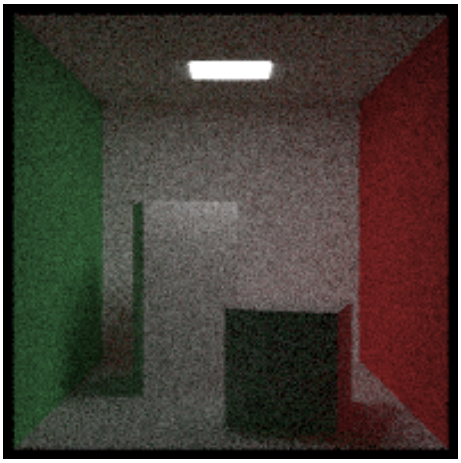
```
// rayt.h
class ONB {
public:
    ONB() {}
    inline vec3& operator[](int i) {
        return m_axis[i];
    }
    inline const vec3& operator[](int i) const {
        return m_axis[i];
    }
    const vec3& u() const {
        return m_axis[0];
    }
    const vec3& v() const {
        return m_axis[1];
    }
    const vec3& w() const {
        return m_axis[2];
    }
    vec3 local(float a, float b, float c) const {
        return a*u() + b*v() + c*w();
    }
    vec3 local(const vec3& a) const {
        return a.getX()*u() + a.getY()*v() + a.getZ()*w();
    }
    void build_from_w(const vec3& n) {
        m_axis[2] = normalize(n);
        vec3 a;
        if (fabs(w().getX()) > 0.9) {
            a = vec3(0, 1, 0);
        }
        else {
            a = vec3(1, 0, 0);
        }
        m_axis[1] = normalize(cross(w(), a));
        m_axis[0] = cross(w(), v());
    }

private:
    vec3 m_axis[3];
};
```

これを使って無作為に作成した方向ベクトルを，物体表面の法線方向を中心として基底変換します．

```
virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
    ONB onb; onb.build_from_w(hrec.n);
```

```
    vec3 direction = onb.local(random_cosine_direction());
    srec.ray = Ray(hrec.p, normalize(direction));
    srec.albedo = m_albedo->value(hrec.u, hrec.v, hrec.p);
    srec.pdf_value = dot(onb.w(), srec.ray.direction()) / PI;
    return true;
};
```

レンダリングすると次のようになります. (rayt302.cpp, rayt302.h)

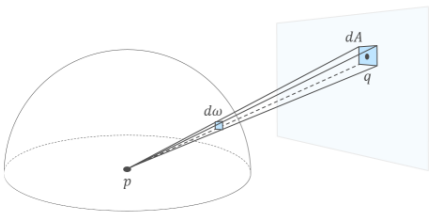


5.4 光源のサンプリング

レイトレーシングでは, カメラから光線を飛ばして, 最終的に光源まで辿りつかないと入射する放射束の精度が大きく下がります.光線の反射は基本的にランダムな方向なのですが, 意図的に光源方向に光線を飛ばしてサンプリングすれば光源にたどり着く可能性が上がります(光線の開始点と光源との間に物体があると, 光源に辿り着かないことがあります)

コーネルボックスでは光源は四角形です.この領域上をサンプリングするために適切な確率密度関数を決定する必要があります.

表面上のある点 p から光源上の点 q の関係は次の図のようになっています.



点 q の面積は dA です.この領域の確率はどれくらいでしょうか.四角形の面積は A なので, 一様分布ならば $\frac{dA}{A}$ です. $d\omega$ は立体角で球の半径を r, 立体角当たりの面積を da とすると $d\omega = \frac{da}{r^2}$ という関係があります.光源上の微小面積 dA の投影面積は $dA\cos(\theta)$ なので $da = dA\cos(\theta)$, なので以下の関係式が成り立ちます.

$$d\omega = \frac{dA\cos(\theta)}{distance(\vec{p},\vec{q})^2}.$$

球上の点をサンプリングする確率は $p(direction)$ なので, 確率変数は $p(direction)\cdot d\omega$ です.今, 球上の $d\omega$ と光源上の dA の確率を一致させたいので,

$$p(direction)\cdot d\omega = p(direction)\cdot\frac{dA\cos(\theta)}{distance(\vec{p},\vec{q})^2} = \frac{dA}{A}.$$

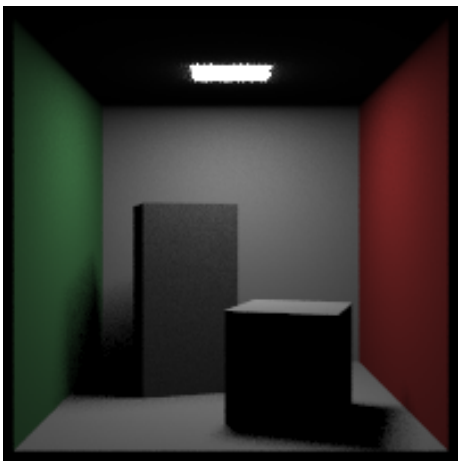
上の式から $p(direction)$ を求めると

$$p(direction) = \frac{distance(\vec{p}, \vec{q})^2}{\cos(\theta) \cdot A}.$$

この式を使って，`color` 関数の中で光源を直接サンプリングするようにしてみます．

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {

            vec3 on_light = vec3(213 + drand48()*(343 - 213), 554, 227 + drand48()*(332 - 227));
            vec3 to_light = on_light - hrec.p;
            float distance_squared = lengthSqr(to_light);
            to_light = normalize(to_light);
            if (dot(to_light, hrec.n) < 0) {
                return emitted;
            }
            float light_area = (343 - 213)*(332 - 227);
            float light_cosine = fabs(to_light.getY());
            if (light_cosine < 0.000001) {
                return emitted;
            }

            srec.pdf_value = distance_squared / (light_cosine * light_area);
            srec.ray = Ray(hrec.p, to_light);

            float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
            vec3 albedo = srec.albedo * spdf_value;
            return emitted + mulPerElem(albedo, color(srec.ray, world, depth + 1)) / srec.pdf_value;
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

これを実行すると次のようになります. (rayt303.cpp)



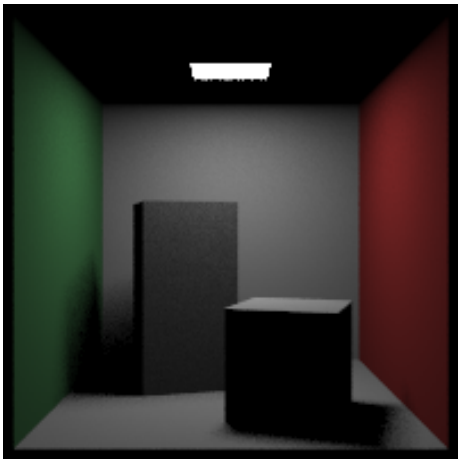
光源の周りにつぶつぶがありますね.これは光源と天井の間に隙間があるためです.今の光源は両面から当たっても発光するので, 片面のみにします.そのため, `DiffuseLight::emitted` を変更します.

```
virtual vec3 emitted(const Ray& r, const HitRec& hrec) const override {
    if (dot(hrec.n, r.direction()) < 0) {
        return m_emit->value(hrec.u, hrec.v, hrec.p);
    }
    else {
        return vec3(0);
    }
}
```

また, コーネルボックスの光源を反転させます.

```
world->add(builder.rectXZ(213, 343, 227, 332, 554, light).flip().get());
```

レンダリングすると次のようになります. (rayt304.cpp)



改善されているのがわかると思います.

5.5 合成 PDF

これまでに, 光源をサンプリングするための pdf と, $\cos(\theta)$ に従った 余弦 pdf を作成しました.これらは合成することができます. pdf は区間 [0,1] であればいいので, 複数の pdf を重み加算します.例えば, 光源 pdf と 反射 pdf をそれぞれ 0.5 の重みとすると次のような合成 pdf が作れます.

$$\text{mix\_pdf(direction)} = \frac{1}{2}\text{light\_pdf(direction)} + \frac{1}{2}\text{reflection\_pdf(direction)}.$$

合成 PDF を作るために, pdf をクラスにします.

```
class Pdf {
public:
    virtual float value(const HitRec& hrec, const vec3& direction) const = 0;
    virtual vec3 generate(const HitRec& hrec) const = 0;
};
```

方向ベクトルから確率密度を求める `value` 関数と, 確率分布に従った方向ベクトルを生成する `generate` 関数があります. `hrec` は生成に必要な情報が含まれていることがあるので渡しています.

まずは 余弦 pdf を実装すると

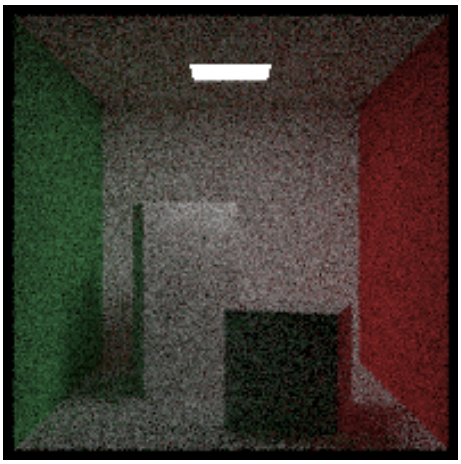
```
class CosinePdf : public Pdf {
public:
    CosinePdf() { }

    virtual float value(const HitRec& hrec, const vec3& direction) const override {
        float cosine = dot(normalize(direction), hrec.n);
        if (cosine > 0) {
            return cosine / PI;
        }
        else {
            return 0;
        }
    }

    virtual vec3 generate(const HitRec& hrec) const override {
        ONB uvw; uvw.build_from_w(hrec.n);
        vec3 v = uvw.local(random_cosine_direction());
        return v;
    }
};
```

この余弦 pdf を試してみます. `color` 関数を次のようにしてみます.

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
```

```
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            CosinePdf cpdf;
            srec.ray = Ray(hrec.p, cpdf.generate(hrec));
            srec.pdf_value = cpdf.value(hrec, srec.ray.direction());
            if (srec.pdf_value > 0) {
                float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
                vec3 albedo = srec.albedo * spdf_value;
                return emitted + mulPerElem(albedo, color(srec.ray, world, depth + 1)) / srec.pdf_value;
            }
            else {
                return emitted;
            }
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

この結果は次のようになります. (rayt305.cpp)



次に物体そのものを pdf として扱えるようにします.これで, 四角形を光源としてサンプリングできるようになります.最初に `shape` クラスに pdf 関連の関数を追加します.

```
class Shape {
public:
    virtual bool hit(const Ray& r, float t0, float t1, HitRec& hrec) const = 0;
    virtual float pdf_value(const vec3& o, const vec3& v) const { return 0; }
    virtual vec3 random(const vec3& o) const { return vec3(1, 0, 0); }
};
```

`Rect` クラスでオーバーライドします.

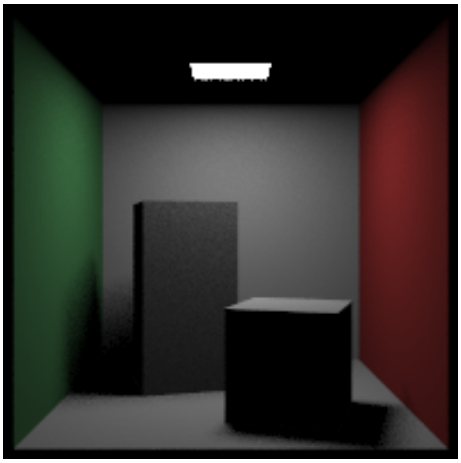
```
virtual float pdf_value(const vec3& o, const vec3& v) const override {
    if (m_axis != kXZ) return 0;
    HitRec hrec;
    if (this->hit(Ray(o, v), 0.001f, FLT_MAX, hrec)) {
        float area = (m_x1 - m_x0) * (m_y1 - m_y0);
        float distance_squared = pow2(hrec.t) * lengthSqr(v);
        float cosine = fabs(dot(v, hrec.n)) / length(v);
        return distance_squared / (cosine * area);
    }
    else {
        return 0;
    }
}

virtual vec3 random(const vec3& o) const override {
    if (m_axis != kXZ) return vec3(1, 0, 0);
    float x = m_x0 + drand48()*(m_x1 - m_x0);
    float y = m_y0 + drand48()*(m_y1 - m_y0);
    vec3 random_point;
    switch (m_axis) {
    case kXY:
        random_point = vec3(x, y, m_k);
        break;
    case kXZ:
        random_point = vec3(x, m_k, y);
        break;
    case kYZ:
        random_point = vec3(m_k, x, y);
        break;
    }
    vec3 v = random_point - o;
    return v;
}
```

`color` 関数も書き換えます.

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            ShapePtr p(std::make_shared<Rect>(
                213, 343, 227, 332, 554, Rect::kXZ, MaterialPtr()));
            ShapePdf cpdf(p, hrec.p);
            srec.ray = Ray(hrec.p, cpdf.generate(hrec));
```

```
            srec.pdf_value = cpdf.value(hrec, srec.ray.direction());

            float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
            vec3 albedo = srec.albedo * spdf_value;
            return emitted + mulPerElem(albedo, color(srec.ray, world, depth + 1)) / srec.pdf_value;
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

この結果は次のようになります. (rayt306.cpp)



それでは合成 pdf を実装します.今回は2つの pdf を合成するクラス `MixturePdf` とします.重みは 0.5 です.無作為なベクトルはそれぞれ 50% の確率でどちらかの pdf を使用して生成するようにしています.

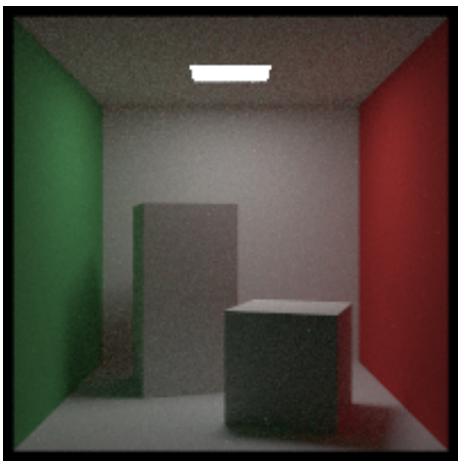
```
class MixturePdf : public Pdf {
public:
    MixturePdf(const Pdf* p0, const Pdf* p1) { m_pdfs[0] = p0; m_pdfs[1] = p1; }

    virtual float value(const HitRec& hrec, const vec3& direction) const override {
        float pdf0_value = m_pdfs[0]->value(hrec, direction);
        float pdf1_value = m_pdfs[1]->value(hrec, direction);
        return 0.5f*pdf0_value + 0.5f*pdf1_value;
    }

    virtual vec3 generate(const HitRec& hrec) const override {
        if (drand48()<0.5f) {
            return m_pdfs[0]->generate(hrec);
        }
        else {
            return m_pdfs[1]->generate(hrec);
        }
```

```
    }

private:
    const Pdf* m_pdfs[2];
};
```

これを使って, 光源 pdf と余弦 pdf を合成してみます.

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            ShapePtr p(std::make_shared<Rect>(
                213, 343, 227, 332, 554, Rect::kXZ, MaterialPtr()));
            ShapePdf shapePdf(p, hrec.p);
            CosinePdf cosPdf;
            MixturePdf mixPdf(&shapePdf, &cosPdf);
            srec.ray = Ray(hrec.p, mixPdf.generate(hrec));
            srec.pdf_value = mixPdf.value(hrec, srec.ray.direction());
            if (srec.pdf_value > 0) {
                float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
                vec3 albedo = srec.albedo * spdf_value;
                return emitted + mulPerElem(albedo, color(srec.ray, world, depth + 1)) / srec.pdf_value;
            }
            else {
                return emitted;
            }
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

この結果は次のようになります. (rayt307.cpp)



これまでは `color` 関数で pdf を作成していましたが, 本来は材質によって pdf は変わります.この仕組みを入れましょう.まずは, `ScatterRec` の `pdf_value` を Pdf のポインタに変更します.

```
class ScatterRec {
public:
    Ray ray;
    vec3 albedo;
    const Pdf* pdf;
};
```

ランバート材質の pdf を設定します.

```
class Lambertian : public Material {
public:
    Lambertian(const TexturePtr& a)
        : m_albedo(a) {
    }

    virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
        srec.albedo = m_albedo->value(hrec.u, hrec.v, hrec.p);
        srec.pdf = &m_pdf;
        return true;
    };

    virtual float scattering_pdf(const Ray& r, const HitRec& hrec) const override {
        return std::max(dot(hrec.n, normalize(r.direction())), 0.0f) / PI;
    }

private:
    TexturePtr m_albedo;
    CosinePdf  m_pdf;
};
```

次に `color` 関数を書き換えます.

```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            ShapePtr p(std::make_shared<Rect>(
                213, 343, 227, 332, 554, Rect::kXZ, MaterialPtr()));
            ShapePdf shapePdf(p, hrec.p);
            MixturePdf mixPdf(&shapePdf, srec.pdf);
            srec.ray = Ray(hrec.p, mixPdf.generate(hrec));
            float pdf_value = mixPdf.value(hrec, srec.ray.direction());
            if (pdf_value > 0) {
                float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
                vec3 albedo = srec.albedo * spdf_value;
                return emitted + mulPerElem(albedo, color(srec.ray, world, depth + 1)) / pdf_value;
            }
            else {
                return emitted;
            }
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

この結果は前回と変わらないはずです. (rayt308.cpp)

5.6 金属の復活

コーネルボックスの材質はランバートのみでしたが, 金属材質も現状に合わせてみます.今回の鏡面反射では反射方向が決まっているので, pdf は必要ないと考えます.そのため, 材質が金属の場合は単純に発光と反射方向からの放射束のみとします.それには, 鏡面反射かどうか知る必要があるので, `ScatterRec` にその情報を追加します.

```
class ScatterRec {
public:
    Ray ray;
    vec3 albedo;
    const Pdf* pdf;
    bool is_specular;
};
```

金属材質の `scatter` 関数を書き換えます.また, `dielectric` の方にも同様に追加します.

```
virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
    vec3 reflected = reflect(normalize(r.direction()), hrec.n);
    reflected += m_fuzz*random_in_unit_sphere();
    srec.ray = Ray(hrec.p, reflected);
    srec.albedo = m_albedo->value(hrec.u, hrec.v, hrec.p);
    srec.pdf = nullptr;
    srec.is_specular = true;
    return dot(srec.ray.direction(), hrec.n) > 0;
}
```

ランバート材質の `scatter` 関数で `is_specular` を `false` に設定します.

```
virtual bool scatter(const Ray& r, const HitRec& hrec, ScatterRec& srec) const override {
    srec.albedo = m_albedo->value(hrec.u, hrec.v, hrec.p);
    srec.pdf = &m_pdf;
    srec.is_specular = false;
    return true;
};
```

そして, `color` 関数で, 鏡面反射のときの処理を追加します.

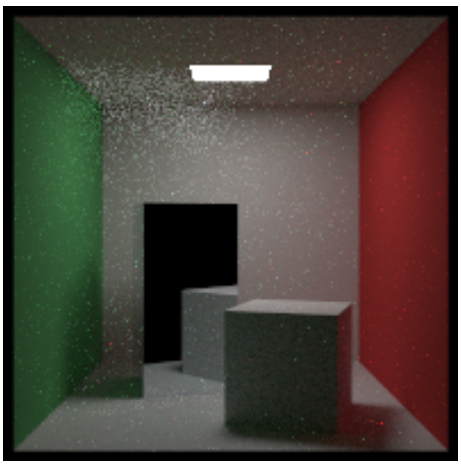
```
vec3 color(const rayt::Ray& r, const Shape* world, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            if (srec.is_specular) {
                return emitted + mulPerElem(srec.albedo, color(srec.ray, world, depth + 1));
            }
            else {
                ShapePtr p(std::make_shared<Rect>(
                    213, 343, 227, 332, 554, Rect::kXZ, MaterialPtr()));
                ShapePdf shapePdf(p, hrec.p);
                MixturePdf mixPdf(&shapePdf, srec.pdf);
                srec.ray = Ray(hrec.p, mixPdf.generate(hrec));
                float pdf_value = mixPdf.value(hrec, srec.ray.direction());
                if (pdf_value > 0) {
                    float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
                    vec3 albedo = srec.albedo * spdf_value;
                    return emitted + mulPerElem(albedo, color(srec.ray, world, depth + 1)) / pdf_value;
                }
                else {
                    return emitted;
                }
```

```
            }
        }
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

あとはコーネルボックスの箱を 1 つをアルミニウム(金属)にしてみます.

```
MaterialPtr aluminum = std::make_shared<Metal>(
    std::make_shared<ColorTexture>(vec3(0.8f, 0.85f, 0.88f)), 0.0f);

ShapeList* world = new ShapeList();
ShapeBuilder builder;
world->add(builder.rectYZ(0, 555, 0, 555, 555, green).flip().get());
world->add(builder.rectYZ(0, 555, 0, 555, 0, red).get());
world->add(builder.rectXZ(213, 343, 227, 332, 554, light).flip().get());
world->add(builder.rectXZ(0, 555, 0, 555, 555, white).flip().get());
world->add(builder.rectXZ(0, 555, 0, 555, 0, white).get());
world->add(builder.rectXY(0, 555, 0, 555, 555, white).flip().get());
world->add(builder.box(vec3(0), vec3(165), white)
    .rotate(vec3::yAxis(), -18)
    .translate(vec3(130, 0, 65))
    .get());
world->add(builder.box(vec3(0), vec3(165, 330, 165), aluminum)
    .rotate(vec3::yAxis(), 15)
    .translate(vec3(265, 0, 295))
    .get());
m_world.reset(world);
```

これを実行すると次のような画像になります. (rayt309.cpp)



左上のところにノイズが強く発生してしまっています.これはアルミニウム箱へのサンプリングが減ってしまったためです.そのため,その箱に

重点的サンプリングすると改善できるはずです.ただし, 今回は箱ではなく球にして重点的サンプリングします.その理由は箱より簡単だからです.

ある点から球に対して重点的サンプリングする場合, 球の立体角を一様分布でサンプリングすればいいと考えられます.一様分布に従う球のベクトルについてはすでにやりました.それによると,

$$r_2 = 2\pi \int_0^{\theta} f(t)\sin(t)dt.$$

今, $f(t)$ が未知なので, $C = f(t)$ とすると

$$r_2 = 2\pi \int_0^{\theta} C\sin(t)dt.$$

この C を求めると

$$\begin{aligned} r_2 &= 2\pi \int_0^{\theta} C\sin(t)dt \\ &= 2\pi C\left[-cos(t)\right]_0^{\theta} \\ &= 2\pi C\{(-\cos(\theta)) - (-\cos(\theta))\} \\ &= 2\pi C(-\cos(\theta) + 1) \\ &= 2\pi C(1 - \cos(\theta)). \end{aligned}$$

また,

$$\cos(\theta) = 1 - \frac{r_2}{2\pi C}.$$

$r_2 = 1$ のとき, θ_{max} が得られるとすると, C は

$$\begin{aligned} 2\pi C(1 - \cos(\theta)) &= r_2 \\ 2\pi C(1 - \cos(\theta_{max})) &= 1 \\ C(1 - \cos(\theta_{max})) &= \frac{1}{2\pi} \\ C &= \frac{1}{2\pi(1 - \cos(\theta_{max}))}. \end{aligned}$$

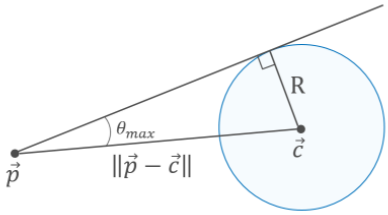
これを先ほどの式に代入すると

$$\begin{aligned} \cos(\theta) &= 1 - \frac{r_2}{2\pi C} \\ &= 1 - \frac{r_2}{2\pi} \cdot 2\pi(1 - \cos(\theta_{max})) \\ &= 1 - r_2(1 - \cos(\theta_{max})). \end{aligned}$$

これを使ってベクトルを生成できます. ϕ は前と同じです.デカルト座標で表すと

$$\begin{aligned} z &= \cos(\theta) = 1 - r_2(\cos(1 - \theta_{max})) \\ x &= \cos(\phi)sin(\theta) = \cos(2\pi r_1)\sqrt{1 - z^2} \\ y &= \sin(\phi)sin(\theta) = \sin(2\pi r_1)\sqrt{1 - z^2}. \end{aligned}$$

ここで，θ_{max} は図のような関係になっています．



この図から θ_{max} を求めると

$$\sin(\theta_{max}) = \frac{R}{distance(\vec{c}-\vec{p})}.$$

$sin^2\theta + cos^2\theta = 1$ を使って

$$\cos(\theta_{max}) = \sqrt{1 - \frac{R^2}{distance(\vec{c}-\vec{p})^2}}.$$

次に方向分布を表す pdf を算出する必要があります.球の全立体角は ω で，ある点から球に向かう方向の微小立体角は $\frac{1}{\omega}$ です.今は単位球上の領域を考えているので

$$\omega = \int_0^{2\pi}\int_0^{\theta_{max}} \sin(\theta)d\theta d\phi = 2\pi(1-\cos(\theta_{max})).$$

これらを使って `sphere` クラスの `pdf_value` 関数と `random` 関数を実装します．

```
virtual float pdf_value(const vec3& o, const vec3& v) const override {
    HitRec hrec;
    if (this->hit(Ray(o, v), 0.001f, FLT_MAX, hrec)) {
        float dd = lengthSqr(m_center - o);
        float rr = std::min(pow2(m_radius), dd);
        float cos_theta_max = sqrtf(1.0f - rr*recip(dd));
        float solid_angle = PI2*(1.0f - cos_theta_max);
        return recip(solid_angle);
    }
    else {
        return 0;
    }
}

virtual vec3 random(const vec3& o) const override {
    vec3 direction = m_center - o;
    float distance_squared = lengthSqr(direction);
    ONB uvw; uvw.build_from_w(direction);
    vec3 v = uvw.local(random_to_sphere(m_radius, distance_squared));
    return v;
}
```

ここで，`random_to_sphere` はある点から球方向に向かう無作為なベクトルを生成します．

```
inline vec3 random_to_sphere(float radius, float distance_squared) {
    float r1 = drand48();
    float r2 = drand48();
    float rr = std::min(pow2(radius), distance_squared);
    float cos_theta_max = sqrtf(1.f - rr * recip(distance_squared));
    float z = 1.0f - r2*(1.0f - cos_theta_max);
    float sqrtz = sqrtf(1.f - pow2(z));
    float phi = PI2*r1;
    float x = cosf(phi) * sqrtz;
    float y = sinf(phi) * sqrtz;
    return vec3(x, y, z);
}
```

これで，球をサンプリングできるようになります．ここで，少し `Scene` クラスを変更して，重点的サンプリング用の物体を渡せるようにします．新たに `m_light` メンバを追加して，`build` 関数で生成します．

```
std::unique_ptr<Shape> m_light;
```

`color` 関数の引数に追加して，処理を行います．

```
vec3 color(const rayt::Ray& r, const Shape* world, const Shape* light, int depth) {
    HitRec hrec;
    if (world->hit(r, 0.001f, FLT_MAX, hrec)) {
        vec3 emitted = hrec.mat->emitted(r, hrec);
        ScatterRec srec;
        if (depth < MAX_DEPTH && hrec.mat->scatter(r, hrec, srec)) {
            if (srec.is_specular) {
                return emitted + mulPerElem(srec.albedo, color(srec.ray, world, light, depth + 1));
            }
            else {
                ShapePdf shapePdf(light, hrec.p);
                MixturePdf mixPdf(&shapePdf, srec.pdf);
                srec.ray = Ray(hrec.p, mixPdf.generate(hrec));
                float pdf_value = mixPdf.value(hrec, srec.ray.direction());
                if (pdf_value > 0) {
                    float spdf_value = hrec.mat->scattering_pdf(srec.ray, hrec);
                    vec3 albedo = srec.albedo * spdf_value;
                    return emitted + mulPerElem(albedo, color(srec.ray, world, light, depth + 1)) / pdf_value;
                }
                else {
                    return emitted;
                }
            }
        }
```

```
        else {
            return emitted;
        }
    }
    return background(r.direction());
}
```

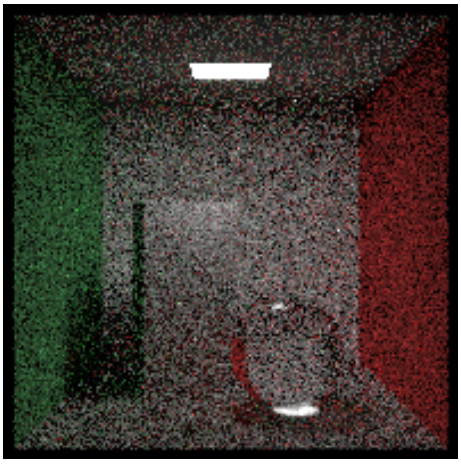
`render` 関数の中で `color` 関数を呼んでいるところも変更します.

```
c += color(r, m_world.get(), m_light.get(), 0);
```

あとは,コーネルボックスの箱を 1 つ球に変更して,材質も `Metal` から `Dielectric` に変えます.

```
MaterialPtr metal = std::make_shared<Dielectric>(1.5f);

ShapeList* world = new ShapeList();
ShapeBuilder builder;
world->add(builder.rectYZ(0, 555, 0, 555, 555, green).flip().get());
world->add(builder.rectYZ(0, 555, 0, 555, 0, red).get());
world->add(builder.rectXZ(213, 343, 227, 332, 554, light).flip().get());
world->add(builder.rectXZ(0, 555, 0, 555, 555, white).flip().get());
world->add(builder.rectXZ(0, 555, 0, 555, 0, white).get());
world->add(builder.rectXY(0, 555, 0, 555, 555, white).flip().get());
world->add(builder.sphere(vec3(190, 90, 190), 90, metal).get());
world->add(builder.box(vec3(0), vec3(165, 330, 165), white)
    .rotate(vec3::yAxis(), 15)
    .translate(vec3(265, 0, 295))
    .get());
m_world.reset(world);

m_light.reset(new Sphere(
    vec3(190, 90, 190), 90, MaterialPtr()));
```

レンダリングすると次のようになります. (rayt310.cpp)



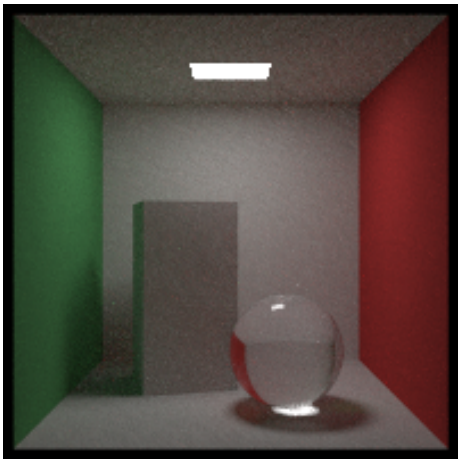
ノイズがひどいですね.これは重点的サンプリング用の物体が球だけだからです. `ShapeList` に pdf の関数を追加して, 複数の物体に対して重点的サンプリング出来るように対応します.重みは均等とし, ベクトルの生成もリストの中から無作為に選んだ物体を使用します.

```
virtual float pdf_value(const vec3& o, const vec3& v) const override {
    float weight = 1.0f / m_list.size();
    float sum = 0;
    for (auto& p : m_list) {
        sum += weight * p->pdf_value(o, v);
    }
    return sum;
}

virtual vec3 random(const vec3& o) const override {
    size_t n = m_list.size();
    size_t index = size_t(drand48() * n);
    if (n > 0 && index >= n) {
        index = n - 1;
    }
    return m_list[index]->random(o);
}
```

これを使って, 重点的サンプリング用の物体が光源(四角形)と誘電体(球)で構成されるようにします.

```
ShapeList* l = new ShapeList();
l->add(builder.rectXZ(213,343,227,332,554,MaterialPtr()).get());
l->add(builder.sphere(vec3(190,90,190),90,MaterialPtr()).get());
m_light.reset(l);
```

これをレンダリングすると次のようになります. (rayt311.cpp)



これが目標の画像になります.

さて, とうとう最後になりましたが, どこかの計算でゼロ除算があったり, `sqrt` 関数に負の値を入れたりとか, 初期化していない変数にアクセスしたりとかで, 値がおかしくなる場合があるかもしれません.計算結果がそのような値になったら黒になるようにフィルターを追加しておきましょう.今さらな感じがしますけどね.

```
class DenanFilter : public ImageFilter {
public:
    DenanFilter() {}
    virtual vec3 filter(const vec3& c) const override {
        return vec3(
            !(c[0] == c[0]) ? 0 : c[0],
            !(c[1] == c[1]) ? 0 : c[1],
            !(c[2] == c[2]) ? 0 : c[2]);
    }
};
```

これをフィルターリストに追加します. (rayt312.h)

```
Image(int w, int h) {
    m_width = w;
    m_height = h;
    m_pixels.reset(new rgb[m_width*m_height]);
    m_filters.push_back(std::make_unique<DenanFilter>());
    m_filters.push_back(std::make_unique<GammaFilter>(GAMMA_FACTOR));
    m_filters.push_back(std::make_unique<TonemapFilter>());
}
```

黒くしていますが, 赤とかにしておくと NaN かどうかがわかりやすくなると思います.

以上でこの入門は終了です.お疲れ様でした!

6. この次はどうするか

6.1 ロシアンルーレット

レイトレーシングでは光線が物体の表面に当たるたびに, 反射した光線をさらに追跡するといった再帰的な処理になっています.この再帰処理はどこかで打ち切らなければなりません.今回は特定の回数に達したときに打ち切るようになっていますが, その光線に適した再帰回数を決められるのが理想です.そこでロシアンルーレットと呼ばれる手法があります.これは重点的サンプリングのように, どれくらい再帰処理をすればよいかを確率によって決定します.

6.2 パストレーシング

コンピュータグラフィックスでは物理ベースレンダリングが主流になっています.この根源となる数式がレンダリング方程式とよばれるもので, モンテカルロレイトレーシングを使ってレンダリング方程式を計算することをパストレーシングといいます.

6.3 関与媒質

埃や煙, 雲といったものを関与媒質ということがあります.これは微粒子が集まっているものと考えられ, 光がそれを通過するときに散乱や吸収が行われます.

6.4 フォトンマップ

光の最も小さい単位は光子(フォトン)です.このフォトンが光源からどのようにばらまかれているかを表した分布データ(フォトンマップ)を作成し,それを追跡することがレンダリングを行う手法です.

7. 参考となる資料

7.1 Peter Shirley 著の「Ray Tracing」

今回の内容のベースとなっている Peter Shirley 著の3冊をお勧めします.

- amazon: Ray Tracing in One Weekend
- amazon: Ray Tracing: the Next Week
- amazon: Ray Tracing: The Rest Of Your Life

今回取り扱っていない関与媒質, BVH, ノイズ, モーションブラー, 被写界深度について書かれています.少し名前を変えていますが, ほとんどが同じ名前になっているので, わかりやすくなっているのではないかと思います.冒頭でもお伝えした通り, この本の内容は現在インターネットから無料で手に入ります.
github/petershirley

7.2 smallpt

smallpt
とても小さいコードのパストレーシングです.次にパストレーシングに挑戦するなら参考になると思います.
SmallPT の解説もありますのでこちらも参考にしてみてください.
Presentation slides about smallpt by David Cline

7.3 edupt

edupt
上記の smallpt を c++ で書き直し, ソースコードには日本語のコメントが沢山付いている物理ベースレンダラです.とりあえず, 次のことに迷ったらこちらのソースコードを見ることをお勧めします.また, edupt の解説もありますので参考にしてみてください.(上記リンク先から参照できます)
また, 作者の方は「Computer Graphics Gems JP 2015 コンピュータグラフィックス技術の最前線」という本で, 「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」を執筆されているので, そちらもお勧めします.

7.4 memoRANDOM

memoRANDOM
レイトレーシングに関する情報が豊富にある日本語のサイトです.とても重宝しています.

7.5「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」

amazon: 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

フォトンマッピングについて書かれている翻訳書です.フォトンマッピングを実装するときや, レイトレーシングをするうえでも必読の本だと言えます.この本を読んでわからないところがあればそれを調べていくというやり方でも実力は付いていくと思います.

7.6「CG Magic」

amazon: CG Magic

レンダリングに関する技術的な解説をしている邦書です.とりあえず, どんな技術があるのか確認したい場合にお勧めします.

7.7 Robust Monte Carlo Methods for Light Transport Simulation

Robust Monte Carlo Methods for Light Transport Simulation

モンテカルロ法について詳しく書かれている文書です. 400 ページ以上あるのに, 無料で手に入るので英語ができるならぜひ読んでみてください.あわせて, Global Illumination compendium もおすすめします.

7.8「Physically Based Rendering」

https://www.pbrt.org/

1000 ページ以上あるトンデモナイ本ですが, 内容はとても充実しているそうです.個人的にはさらっと見ただけで, ちゃんと読んではいません.洋書なので, 知識と根気があるなら読んでみてはいかがでしょうか.現在ではこの本無料で読むことできるようになりました.詳しくは上記のリンク先を参照してください.

7.9 レイトレ合宿

レイトレ合宿 6

レイトレーシングの作品を出して技術を競い合っている人たちがいます.また, 技術力を高めるためにソースコードや資料の公開をしてくれています.大変勉強になりますので, 興味のある方は見てみてください.

最後に

最近勉強したので，そのまとめとして書きました．説明不足や理解不足があるかもしれません．少しでも参考になれば幸いです．

参考文献

[1] Peter Shirley,「Ray Tracing in One Weekend」, 2016

[2] Peter Shirley,「Ray Tracing: the Next Week」, 2016

[3] Peter Shirley,「Ray Tracing: The Rest Of Your Life」, 2016

[4] Eric Veach,「Robust Monte Carlo Methods for Light Transport Simulation」, 1997

[5] Henrik Wann Jenson 著, 苗村健訳「フォトンマッピング ー 実写に迫るコンピュータグラフィックス」オーム社, 2002

[6] 倉地紀子,「CG Magic:レンダリング」オーム社, 2007

[7] Eric Lengyel 著, 狩野智英訳「ゲームプログラミングのための3 D グラフィックス数学」ボーンデジタル社, 2006

[8] 鈴木健太郎,「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」『Computer Graphics Gems JP 2015 コンピュータグラフィックス技術の最前線』ボーンデジタル, 2015, p. 3