Rust ではじめるレイトレーシング入門

mebiusbox

2021 年 2 月 10 日

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．

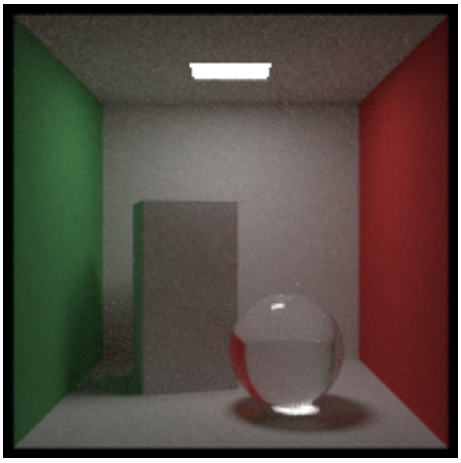
本記事では，モンテカルロレイトレーシングについて解説します．内容としては Peter Shirley 著の 3 冊

- 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．https://raytracing.github.io/

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．

サンプルコードは Rust で書いています．Rust は最近，人気の出ている言語でこれまでの様々な言語を参考にして作られた新しい言語です．本書では最初に Rust 言語について解説した後に，実際に Rust 言語でレイトレーシングのプログラムを作っていきます．本書は以前に作成した「レイトレーシング入門」の Rust 版で，Rust 言語の解説とサンプルコードを書き直したものです．レイトレーシングの内容そのものは変わっていません．Rust 言語を既に習得されているなら，第一章は飛ばしてもらって構いません．本書の目標は次のような画像をレンダリングすることです．



目次

第 1 章

Rust

1． Rust とは

Rust はマルチパラダイムプログラミング言語です．Rust は静的型付け（statically-typed），式ベース（expression-based）であり，手続き型・関数型プログラミングの両方を実装することが出来ます．また，オブジェクト指向を言語としてサポートしている訳ではありませんが，オブジェクト指向プログラミングも制限付きで実装することが出来ます．Rust はパフォーマンス，信頼性，生産性に重点を置き，システムプログラミング言語として適した言語を目指しています．

個人的に Rust は関数型プログラミング言語である Haskell に大きく影響を受けており，ほとんどの部分が Haskell から引き継いでいるように思えます．ただし，純粋関数型でもなく，モナドというものもありません．そこにポインタや参照といった別の言語の概念を取り入れ，さらに所有権といった独自の機能を組み込んだものです．これは，用語や機能が既存のプログラミング言語の概念に似てはいますが，必ずしも一致していないため，混乱しやすいところです．なので，最初は Rust を全く新しい言語として扱ったほうがいいかもしれません．

Rust には公式ドキュメント The Rust Programming Language Book があります．これはかなり丁寧で豊富な内容が含まれているわけですが，説明が冗長であり，量も多いので読むのに時間がかかります．そのため，手っ取り早く Rust を始める場合は Tour of Rust を利用した方がいいです．手軽にコードを動かしながら進められるのでとても解りやすいです．順番としては Tour of Rust をやり終えてから公式ドキュメントまたは他の参考サイトを参照するのが良いと思います．

これから Rust について解説していきます．注意として，わかり易さ・明確さを重視しているため，公式ドキュメントで使用している用語，およびその意味とは異なるところがいくつかあります．また，読者対象には何かしらのプログラミング言語を学んでいる人を想定しています．Rust は簡単な言語ではありません．プログラミング初心者の場合は，まず別の言語から覚えることをオススメします．

2． Hello World!!

以下は，おなじみの Hello World プログラムです．非常に単純で直感的に書けます．Rust Playground というサイトでは Rust プログラムを手軽に試すことができます．試しに下の内容を実行してみても構いません．サイトを開いたら最初から似たようなプログラムが入力されているかもしれません．

```
fn main() {
    print!("hello world!")
}
```

3. 基本

3.1 コメント

Rust のコメントには，一行コメント (`//`) と，ブロックコメント (`/*`) (`*/`) があります． `/*` ブロックコメントは `/*` このようにネストして `*/` 書くことができます． `*/`

3.2 式と文

Rust は式ベースの言語です．ほとんどが**式**（expression）で表されます．ここで式は返り値を評価するものです．簡単に言うと，式は何かしらの値を返します．それに対して，**文**（statement）は処理を実行しますが値を返しません．

3.3 ブロック

式の１つにブロックがあります．これは `{` と `}` で囲んだものです．例えば `{0}` というのは 0 を返す式です．ブロックが返す値は省略することができます．その場合は `()` を返します．この `()` をユニットと言います．つまり， `{}` というのは `{()}` ということになります．ブロックには２つの機能があります．１つはスコープを作成します．もう１つは，文をいくつも記述できることです．

```
{ statement; statement; statement; ...; (expression) }
```

ここでセミコロン (`;`) は式を文に変化させるものです．

3.4 オブジェクト

数値や関数や参照など，型の実体はすべて**オブジェクト**です．つまり，式が返す値もまたオブジェクトになります．例えば， `1` という値も数値オブジェクトであり， `1 == {1}` という関係にあります．

3.5 所有権

オブジェクトには**所有権** (Ownership) が付いています．この所有権には２つの属性があります．

3.6 束縛

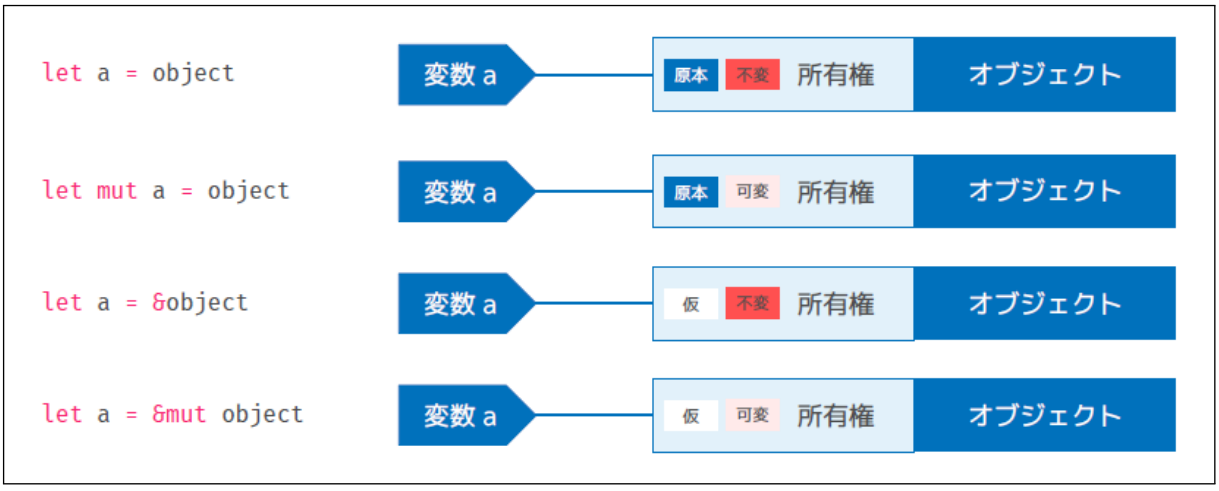
`let` 文を使うことでオブジェクトと変数を**束縛**します．変数はそのスコープから外れたときに束縛していた所有権を放棄します．また，最初に束縛したオブジェクトの所有権は基本的に原本となり，原本および仮の所有権がすべて放棄された時にオブジェクトは破棄されます．

3.7 参照

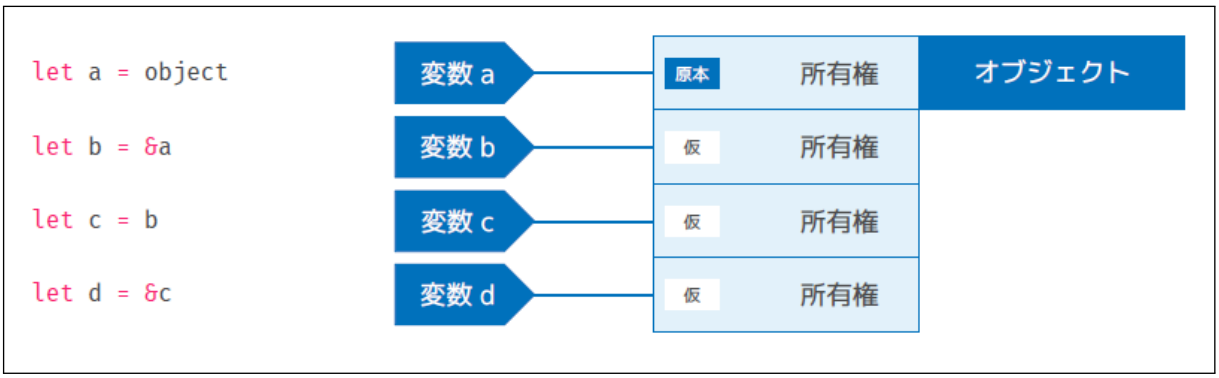
Rust では所有権を使ってオブジェクトを受け渡します．通常は所有権を渡してしまうと束縛が解除されて，受け取った側がそれを束縛します．そこで，仮の所有権を作成して相手に渡すことで，渡す側は束縛を解除されず，仮の所有権を受け取った側はその所有権を使ってオブジェクトを操作することが出来ます．そして，受け取った側の変数がスコープを外れた時に束縛していた仮の所有権が破棄されます．この時，原本または他の仮の所有権があればオブジェクトは破棄されません．仮の所有権を作成する方法の 1 つが**参照**（reference）です．これは `&` 演算子を使います．

3.8 可変性

Rust は標準でオブジェクトを**不変**（immutable）で束縛します．そこで，`let` ではなく `let mut` を使うことで，オブジェクトを**可変**（mutable）で束縛することが出来ます．今までのことをまとめると次のようになります．



原本の所有権から仮の所有権を作成することが出来ます．また，仮の所有権を複製することも出来ますし，仮の所有権からさらに仮の所有権を作れます．ただし，仮の所有権から原本の所有権は作れません．



3.9 借用チェック

同一オブジェクトに対する参照と可変について，いくつか制限があります．

- 不変参照 (`&`) は何個でも同時に存在することが出来る
- 不変参照 (`&`) と可変参照 (`&mut`) は同時に存在することが出来ない
- 可変参照 (`&mut`) は同時に１つしか存在することが出来ない

ここで大事なことは，上記の制限は**関数呼び出し時**（かつコンパイル時）にチェックされるということです（これを**借用チェック**と呼びます）．このチェックが行われる直前の可変参照（必ず１つ）もしくは不変参照（複数可）がその時に存在していることになります．少なくとも可変参照を作成した時には，それまでの不変参照または可変参照がすべて無効となり，存在しないことになります．もちろん，あくまで同一オブジェクトに対する参照に対してです．以下は複数の不変参照が存在しても問題のないコードです．

```
fn main() {
    let a = 10;               // immutable object
    let aref1 = &a;           // reference
    let aref2 = &a;           // reference
    println!("{}, {}, {}", a, aref1, aref2); // borrow check!! - OK
}
```

次は，可変参照をしているところが複数ありますが，借用チェック時に制約を満たしているのでこれも問題ありません．

```
fn main() {
    let mut a = 10;           // mutable object
    let a_ref1 = &a;          // reference
    let a_mut_ref1 = &mut a;  // mutable reference
    let a_mut_ref2 = &mut a;  // mutable refernece
    *a_mut_ref2 = 20;         // assign
    println!("{}", a);        // borrow check!! - OK
}
```

次は，可変参照を複数存在してしまうことになるのでコンパイルエラーになります．

```
fn main() {
    let mut a = 10;           // mutable object
    let a_ref1 = &a;          // reference
    let a_mut_ref1 = &mut a;  // mutable reference
    let a_mut_ref2 = &mut a;  // この時点で a_ref1, a_mut_ref1 は存在しない
    *a_mut_ref1 = 20;         // assign (error)
    println!("{}", a);        // borrow check!! - Error!
}
```

次は，可変参照と不変参照が同時に存在してしまうことになるのでコンパイルエラーになります．

```
fn main() {
    let mut a = 10;           // mutable object
    let a_ref1 = &a;          // reference
    let a_mut_ref1 = &mut a;  // mutable reference
    let a_mut_ref2 = &mut a;  // この時点で a_ref1, a_mut_ref1 は存在しない
    println!("{}", a_ref1);   // borrow check!! - Error!
}
```

不変参照や可変参照を複数束縛しているコードでも借用チェック時に制約を満たしていればコンパイルは通ります．

```
fn main() {
    let mut a = 10;           // mutable object
    let a_ref1 = &a;          // reference
    let a_mut_ref1 = &mut a;  // mutable reference
    let a_mut_ref2 = &mut a;  // この時点で a_ref1, a_mut_ref1 は存在しない
    let a_ref2 = &a;          // この時点で a_mut_ref2 は存在しない
    //println!("{}", a_mut_ref2);        // borrow check!! - Error!
    //println!("{} {}", a_ref1, a_ref2); // borrow check!! - Error!
    println!("{}", a_ref2);   // borrow check!! - OK
}
```

このように関数呼び出しによる借用チェックによって，スコープから抜けていない変数であっても，それが参照なら存在していないことになりうるということです（ここで存在していないと言っていますが，実際には存在できないようにコンパイル時にエラーが出るということ）．参照を束縛した変数をなるべく作らないことが大切です．

3.10 参照外し

参照（仮の所有権）を使ってオブジェクトの操作をする場合は**参照外し**（dereference）が必要です．これは `*` 演算子を使います．

```
fn main() {
    let mut a = 10;            // mutable object
    let a_mut_ref = &mut a;    // mutable reference
    *a_mut_ref = 20;           // dereference and assign
    println!("{}", a_mut_ref); // auto dereference
}
```

参照外しは関数に渡した時や，`.` 演算子によるフィールド操作・メソッド呼び出し時などにおいて自動で行われる場合があります．

4．コピートレイト

Rust には**トレイト**（trait）というデータ型を分類する概念があります．例えば，数値全般を表す `Num` というトレイトがあったとき，それを実装しているデータ型はすべて数値型として分類することができる，というものです．トレイトには特有のメソッドを実装することが出来ます．また，ジェネリクスにおいて，あるトレイトを実装した型であるという制約をかけることが出来ます．これを**トレイト境界**（trait bound）と呼びます．

トレイトは標準でいくつか実装されているものがあり，その１つが**コピートレイト**（Copy Trait）です．束縛したオブジェクトがコピートレイトを実装したデータ型の変数から別の変数に束縛するときは，所有権は移動せず，値をコピーして新しいオブジェクト（そして所有権）を作成します．Rust のプリミティブ型はコピートレイトを実装しています．

```
fn main() {
    let a = 10;            // immutable object
    let b = a;             // copy
    print!("{} {}", a, b); // borrow check!! – OK
}
```

不変参照 (`&`) もコピートレイトを実装しています．

```
fn main() {
    let a = 10;                 // immutable object
    let a_ref = &a;             // reference
    let a_ref_copy = a_ref;     // copy reference
    print!("{} {} {}", a, a_ref, a_ref_copy); // borrow check!! – OK
}
```

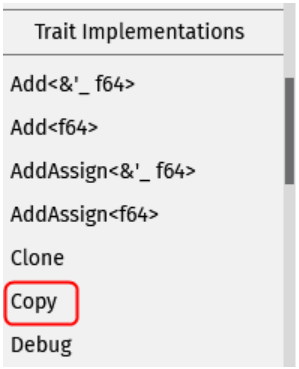
注意なのが，可変参照 (`&mut`) はコピートレイトを実装していません．なぜなら，可変参照は１つしか存在してはいけないからです．

```
fn main() {
    let mut a = 10;                // mutable object
    let a_mut_ref = &mut a;        // mutable reference
    let a_mut_ref_move = a_mut_ref; // move mutable reference
    print!("{}", a_mut_ref);       // borrow check!! – Error!
}
```

次のように可変参照の場合は移動します．

```rust
fn main() {
    let mut a = 10;                 // mutable object
    let a_mut_ref = &mut a;         // mutable reference
    let a_mut_ref_move = a_mut_ref; // move mutable reference
    print!("{}", a_mut_ref_move);   // borrow check!! – OK
}
```

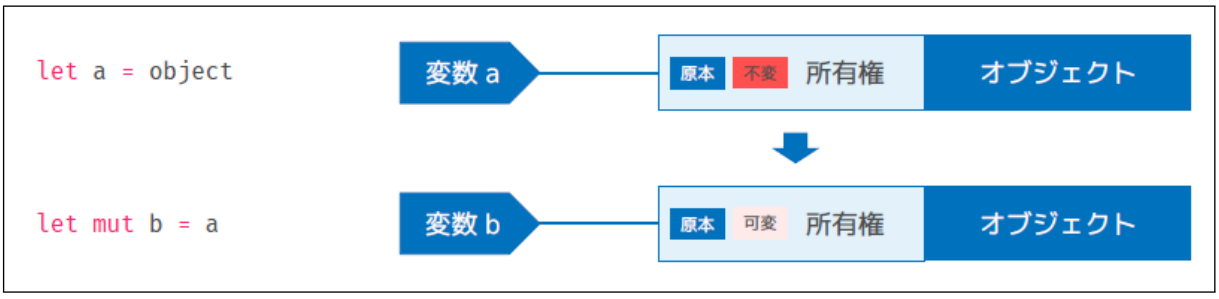
データ型がコピートレイトを実装しているかどうかはドキュメントに記載されています．調べたい型のドキュメントにある `Trait Implementations` の中に `Copy` があるかどうか確認してみてください．



また，下記に示す関数 `copy_trait_check` では，トレイト境界を使って引数の型がコピートレイトを実装していることを強制します．実装されていなかったらコンパイルエラーになります．エラーから分かるように， `String` 型はコピートレイトを実装していません．

```rust
fn copy_trait_check<T: Copy>(_: T) {} // trait bound

fn main() {
    let s = String::from("hello");    // String
    copy_trait_check(s);
                  // ^ the trait ‘Copy‘ is not implemented for ‘String‘
                  // error[E0277]: the trait bound ‘String: Copy‘ is not satisfied
    let a = 10;          // i32
    copy_trait_check(a); // OK
}
```

不変束縛の変数から可変束縛の変数に変えることができます．



変数 `a` から変数 `b` に所有権が移動し，可変に変わります．そして，変数 `a` の束縛は解除されます．もし，オブジェクトがコピートレイトを実装していたらコピーが作成され，変数 `a` はオブジェクトを束縛したままで，変数 `b` には新しい可変のオブジェクトが束縛されます．

5. データ型

Rust の標準にある基本的なデータ型は次のとおりです：

整数型	`i8`, `u8`, `i16`, `u16`, `i32`, `i64`, `u64`, `isize`, `usize`
浮動小数点型	`f32`, `f64`
ブーリアン型	`bool`
文字列型	`char`
タプル（複合）型	`(500, 6.4, true)`. `()` はユニット.
配列型	`[1,2,3,4,5]`, `[3;5]` = `[3,3,3,3,3]`

数値型のリテラルには次のものが使えます：

- `98_222` (10 進数)
- `0xff` (16 進数)
- `0o77` (8 進数)
- `0b1111_0000` (2 進数)
- `b'A'` (バイト)
- `0.` (浮動小数点数)

基本的なデータ型はコピートレイトを実装しています．また，複合・配列型については，含まれている要素がすべてコピートレイトを実装していれば，全体もコピートレイトを実装したことになります．参照も型の１つで，不変参照はコピートレイトを実装していますし，可変参照は実装していません．また，参照のまた参照ということも可能です．

```
fn main() {
    let a = 42;
    let ref_ref_ref_a = &&&a;
    let ref_a = **ref_ref_ref_a;
    let b = *ref_a;
    print!("{} {}", a, b);
}
```

比較するときは基本的に同じ型でなくてはならないので，参照もまた型であるということが以下でわかります．

```
fn main() {
    let a         = 10;     // immutable object
    let a_ref     = &a;     // reference
    let a_ref_ref = &a_ref; // reference to reference
    println!("{}", a == a_ref);
    // error[E0277]: can't compare `{integer}` with `&{integer}`
    println!("{}", a_ref_ref == a_ref);
    // error[E0277]: can't compare `&{integer}` with `{integer}`
}
```

Rust は強い型推論を持っていますが，意図的にデータ型を指定したい場合があります．その場合は変数名の後ろに `:` を付けてデータ型を指定します．

```
fn main() {
    let a: i32 = 10;
    let b: u32 = 20;
    let c: f32 = 0.;
    let d: &i32 = &50;
    print!("{} {} {} {}", a, b, c, d);
}
```

変数は**パターン**を使って，要素を分解して束縛することが出来ます．

```
fn main() {
    let (x,y,z) = (1,2,3);
    let [a,b,c] = [4,5,6];
    let (i,_,_) = (7,8,9);
    println!("xyz= {} {} {}", x, y, z);
    println!("abc= {} {} {}", a, b, c);
    println!("  i= {}", i);
}
```

`_` は**ワイルドカード**と呼ばれるもので，オブジェクトを無視するときに使います．Rust では同じスコープ内で，変数名を使い回すことができます．

```
fn main() {
    let str_len = String::from("hello world!");
    let str_len = str_len.len();
    println!("{}", str_len);
}
```

あるスコープのさらにローカルなスコープにおいても外側にある変数名と同じ名前で新しく束縛できます．このとき，外側の変数はローカルから隠れます．これを**シャドーイング**と言います．

```
fn main() {
    let a = 10;

    { // local scope
        let mut a = 20;
        a += 30;
        println!("{}", a); // 50
    }

    println!("{}", a); // 10
}
```

明示的に数値オブジェクトを型変換して使いたい場合があります．その場合は `as` を使います．

```
fn main() {
    let a = 13u8;
    let b = 7u32;
    let c = a as u32 + b;
    println!("{}", c);

    let t = true;
    println!("{}", t as u8);
}
```

6. 関数

関数を定義するには `fn` を使い，本体は `{}` で囲みます．引数の型は必ず明記しなければなりません． `fn` は文で， `{}` は式です．式は返り値を持つものでしたよね．返り値の型は `->` で指定します．また， `return` で処理を中断して値を返すことが出来ます．

```
fn add(a: i32, b: i32) -> i32 {
    a+b
}

fn main() {
    print!("{}", add(10,20));
}
```

関数の引数に対してパターンによる分解束縛をすることが出来ます．

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("location: ({},{})", x, y);
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

7. 制御構文

7.1 if式

`if` は条件によって処理を分岐するものです． `if` ， `else` ， `else if` が使えます．それぞれに続くものは式 `{}` です．

```
fn main() {
    let number = 6;
    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4 or 2");
    }
}
```

`if` は式なので，値を返せます．ただし，その場合は返す値が同じ型でなければなりません．

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };
    // let number = if condition { 5 } else { "six" }; Error!
    println!("The value of number is: {}", number);
}
```

7.2 loop 式

無限ループするには `loop` を使います．ループから抜ける場合は `break` を使います． `loop` もまた式なので， `break` に返り値を指定することが出来ます．

```
fn main() {
    let mut counter = 0;
    let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {}", result);
}
```

7.3 while 式

条件を満たしている間だけループさせる場合は `while` 式を使います． `while` 式は常に `()` を返します．`break` は使えますが，値を返すことは出来ません．

```
fn main() {
    let mut number = 3;
    while number != 0 {
        println!("{}", number);
        number -= 1;
    };

    println!("LIFTOFF!!!");
}
```

7.4 for 式

イテレータを使って，各要素に対して処理を行いたい場合は `for` を使います． `for` 式は常に `()` を返します．（イテレータとは，連続する一連のデータへのアクセスを提供するオブジェクトのことです）

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a.iter() {
        println!("the value is: {}", element);
    }
}
```

8．スライス

8.1 スライス

スライスは参照の１つで，別のオブジェクト内の連続した要素を指し示すものです．スライスを取得するには，オブジェクトに対して**数列指定**（ `[m..n]` ）します．参照なので，スライスの型は例えば配列だと `&[T]` となります． `T` は任意の型です．

```
fn main() {
    let a = [1,2,3,4,5];
    let a_slice = &a[1..3];
    dbg!(a_slice); // [2,3]
}
```

8.2 数列

数列は `Range` 型と呼ばれるもので `m..n` の形で指定します． `m` と `n` はそれぞれ開始値と終了値で， `m` から `n-1` までの連番を表します． `m..=n` とすることで， `m` から `n` までの連番を表します．数列は `for` 式で利用することができます．

```rust
fn main() {
    let mut sum=0;
    for i in 1..100 {
        sum += i;
    }
    println!("sum={}", sum);
}
```

Rust はインデックスが `0` から始まります．数列指定では開始インデックスと終了インデックスを `..` を使って指定します．例えば `m..n` なら `[m, m+1, m+2, ⋯, n-1]` となります． `m..=n` の場合は `[m, m+1, m+2, ⋯, n]` となります．開始インデックスと終了インデックスは省略することが出来ます．

```rust
let s = String::from("hello");
let slice = &s[0..2];
let slice = &s[0..=2];
let slice = &s[..2];
let slice = &s[3..s.len()];
let slice = &s[3..];
let slice = &s[..];
```

8.3 文字列リテラル

文字列のスライスの型は `&str` です．そして，文字列リテラルは不変の文字列スライス (`&str`) です．

```rust
let s = "Hello, world!";
```

9． 構造体

9.1 構造体

構造体はデータ型の要素を集めたものです． １つ１つの要素を**フィールド**と呼びます．構造体の定義は `struct` を使い，フィールドは名前と型を指定します．

```rust
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

構造体のオブジェクトを作成する場合は，各フィールドを `key:value` という形で束縛します．

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

ここでは「**オブジェクト**」と「**インスタンス**」について説明します．一般的に構造体や列挙型など（オブジェクト指向でのクラス）の実体は「インスタンス」と呼ばれています．しかし，本書ではそれらを「オブジェクト」に統一します．そして，「インスタンス」は，関数型プログラミング言語 Haskell に従って，**データ型**を表します．例えば，コピートレイトを実装した構造体 `Hoge` があるとします．このとき， `Hoge` はコピートレイトの**インスタンス**（**データ型**）です．そして，トレイト境界でコピートレイトを指定した場合，コピートレイトのインスタンスである `Hoge` は制約を満たしていることになります．

可変のオブジェクトを作成するとすべてのフィールドが可変になります．オブジェクトのフィールドは `.` 演算子を使って指定します．

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```

オブジェクト作成時に指定する変数名と構造体のフィールド名が一致している場合，フィールド名を省略することが出来ます．

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

あるオブジェクトのフィールドに束縛したものを使って，新しいオブジェクトを作成するときに便利な構文があります．オブジェクト作成時に，明示的にフィールドを指定しなかったものは `..` の後に渡したオブジェクトのフィールドを束縛します．ただし，コピートレイトを実装している型なら複製され，そうでないなら所有権が移動することに注意が必要です．

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

9.2 タプル構造体

指定した要素で構成されたタプルに名前をつけることが出来ます．このようなタプルを**タプル構造体**といいます．この場合，フィールド名はありません．タプル構造体は同じ構成をしていても別の型として区別されます．タプルは `.0` というように要素のインデックスを指定するか，要素の分解を使います．

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let Point (x,y,z) = Point(0, 0, 0);

    println!("{} {} {}", black.0, black.1, black.2);
    println!("{} {} {}", x, y, z);
}
```

Rust は静的型付け言語です．この特性を利用して既存の型から新しい型を作成することで意図的な意味を付加させて，制約することが出来ます．また，既存の型を利用するので薄いラッパー型と考えることも出来ます．これは **Newtype** パターンと呼ばれるもので，タプル構造体を利用する例の１つです．例えば， `String` 型から `Password` 型を作成し， `{}` で出力したときに伏せ字にする場合は次のようになります．

```
use std::fmt;

struct Password(String);

impl fmt::Display for Password {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.0.chars().map(|_| '*').collect::<String>())
    }
}

fn main() {
    let a = String::from("123456789");
    println!("{}", a); // 123456789

    let a = Password(String::from("123456789"));
    println!("{}", a); // *********
}
```

9.3 ユニット構造体

`()` のことをユニットと言いました．このようなフィールドを何も持たない構造体のことを**ユニット構造体**（Unit-like Structs）と言います．（日本語ドキュメントだとユニット様構造体と翻訳されていますが，ユニット構造体の方が言いやすいし意味も伝わるでしょう）．ユニット構造体はフィールドを持たず，トレイトだけ実装するといった時に使われるようです．

9.4 メソッド

メソッドは関数に似ていますが，構造体と関連していて，`self` を使うことで，そのメソッドを呼び出したオブジェクトを操作することが出来ます．メソッドの第一引数は必ず `self` になります．また，基本的に不変参照（`&self`）か可変参照（`&mut self`）になります．もちろん，可変参照のメソッドは，呼び出し元が可変の所有権を使って呼び出さなければなりません．メソッドは `impl` ブロックの中で，関数と同じく `fn` を使って定義します．

```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50, };

    println!("The area of the rectangle is {} square pixels.", rect1.area());
}
```

`impl` ブロックは複数定義することが出来ます．トレイトごとに実装を分けたりすることが出来ます．

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

重要

メソッドの第一引数が参照ではなく `self` の場合があります．これは呼び出し元のオブジェクトの所有権をメソッドが受け取ります．つまり，このメソッドを呼び出したとき，呼び出し元のオブジェクトが束縛されていたら，それは解除され使用できなくなるということです．これはメソッド呼び出しによってオブジェクトが別のものに変換するといったときに使われるようです．例えば，後に出てくる `Option` や `Result` の `unwrap` というメソッドは `unwrap(self)` です．

9.5 関連関数

`impl` ブロックの中で関数を定義することが出来ます．それは `self` を引数に取りません．このような関数を関連関数と呼びます．これは構造体に関連しているにも関わらず，そのオブジェクトが無くても呼び出すことが出来ます．関連関数は主にその構造体のオブジェクトを生成する関数の定義に使い，そのような関連関数には `new` という名前が使われます．関連関数は構造体の名前に `::` を使って呼び出します．

```
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn new(x: f64, y: f64) -> Self { // Self は実装している型の型エイリアス
        Self { x, y }
    }
}

fn main() {
    let a = Point::new(3., 5.);

    print!("x={}, y={}", a.x, a.y);
}
```

10． 列挙型

列挙型は取りうる様々な値を列挙しておき，そのうちのどれか 1 つだけ値を取るデータ型です．列挙した値のことを**列挙子**（variant）と呼びます．構造体がフィールドの集合に対して `AND` の関係であると考えれば，列挙型は `OR` の関係にあると言えます．列挙型は `enum` を使って定義し，列挙子は `::` で指定します．

```
enum IpAddrKind {
    V4,
    V6,
}
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

列挙子はそれぞれ別々の型にすることが出来ます.

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

列挙型は，構造体のようにメソッドを定義することも出来ますし，トレイトのインスタンスにもなることが出来ます.

11. ジェネリクス

Rust では，例えば数値演算をするときに，左値と右値の型が同じである必要があります．ここで，加算を行う `add` という関数を考えてみます．数値型には整数型や浮動小数点型などあります．型の数だけ `add` 関数を定義してしまうと同じコードが大量に出来てしまいます．そこで，引数の型が変わっても関数本体のコードが変わらない場合は，任意の型を受け取れる関数を定義することでコードの重複を避けることが出来ます．このような仕組みを**ジェネリクス**と言い，任意の型のことを**ジェネリック型**と言います.

関数，構造体，列挙型でジェネリック型を使うには，それぞれの名前の後ろに `<>` でジェネリック型の名前を指定します．名前には慣例的に `T` をよく使います．また，複数であれば， `,` で列挙します.

```
fn add<T>(a: T, b: T) -> T {
    a+b
}

struct Point<T> { x: T, y: T }

enum Result<T,E> {
    Ok(T),
    Err(E),
}
```

メソッドの定義では次のように記述します.

```
struct Point<T> { x: T, y: T }

impl<T> Point<T> {
    fn xy(self) -> (T, T) {
        (self.x, self.y)
    }
}
```

`impl` の後ろに `<T>` を宣言しています．こうすることで `Point<T>` の `T` がジェネリック型であることを明示しています．もし， `impl<T>` でなければ， `Point<T>` の `T` はジェネリック型ではなく `T` という名前の

型を指定することになってしまいます．これとは逆に，ジェネリック型に明示的な型を指定するやり方があります．

```
impl Point<f64> {
    fn distance(&self) -> f64 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

ジェネリック型は任意の型を受け取りますが，静的型付け言語では，コンパイル時に型がわかるので，型に特化したコードが生成されます．このような仕組みを**単相化**（monomorphzation）と言います．

Rust は強い型推論があるので，ジェネリック型に対して適切な型を自動で推論してくれます．しかし，明示的に指定したい場合もあります．この場合は `::<...>` 演算子を使います．この演算子は魚が速く泳いでいるように見えることから **turbofish** と呼ばれています．

```
let point = Point::<f64>{x: 3., y: 5.};
```

12． Option 型

Rust には無効な値を取ることができる便利な列挙型として `Option` があります． `T` はジェネリック型です．

```
enum Option<T> {
    Some(T),
    None,
}
```

`Option` 型に有効な値を束縛するときは `Some` を使います．また，無効な値を束縛するときは `None` を使います．

```
let some_number = Some(5);
let some_string = Some("a string");
let absent_number: Option<i32> = None;
```

`Option` 型のオブジェクトから値を取り出すには，この後に説明するパターンマッチングか， `unwrap` などのメソッドを使います．

13． パターンマッチング

13.1 match 式

パターンマッチングは，式の値がパターンに一致するかしないかを判定する仕組みです． `match` 式は，パターンマッチングによって評価する式を変えるときに使います．

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

`match` 式はパターンと式を `=>` で結合したものを並べたものです．この `パターン => 式` のことを**アーム**（arm）と言います． `match` 式はアームのパターンを順番に処理していき，最初にパターンに一致した式を評価してその結果を返します．そして，パターンに一致したアーム以降は処理されません．このような仕組みを短絡評価またはショートサーキットと呼びます

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

`match` 式はマッチング対象のオブジェクトが取りうる値をすべて網羅しなければなりません．そのため，記述したアーム以外に一致する**ワイルドカード**（ `_` ）を使うことができます．

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

`Option` 型が束縛しているオブジェクトを `match` 式で取り出すことが出来ます．ここで注意なのが， `match` 式でパターンが一致したときに，束縛しているオブジェクトを受け取りますが，そのとき所有権も移動しているということです．

```
fn main() {
    let a: Option<String> = Some(String::from("hello"));
    match a {
        Some(x) => println!("{}", x), // move ownership
        None => ()
    }
    println!("{:?}", a);
    //               ^ value borrowed here after partial move
    // error[E0382]: borrow of partially moved value: ‘a‘
}
```

これは何が起きているかと言うと，変数 `a` が束縛している `Option` 型のオブジェクトが，内部で束縛しているオブジェクトの所有権をアームのパターンによって取り出され所有権が渡されています．これにより，変数 `a` は束縛したままですが，その内部では何も束縛していないことになります．なので，部分移動（partial move）が発生していることになりエラーとなります．この部分移動に対応する方法として次の２つがあります．

１つはアームのパターンでオブジェクトを参照で受け取る方法です．注意なのが，パターンで参照を取得するときは `ref` を使います．可変参照なら `ref mut` です．

```
fn main() {
    let a: Option<String> = Some(String::from("hello"));
    match a {
        Some(ref x) => println!("{}", x), // reference
        None => ()
    }
    println!("{:?}", a); // borrow check!! - OK
}
```

もう１つは，返り値としてオブジェクトを返すことです．

```
fn main() {
    let a: Option<String> = Some(String::from("hello"));
    let a = match a {
        Some(x) => { println!("{}", x); Some(x) }
        None => None,
    };
    println!("{:?}", a); // borrow check!! - OK
}
```

13.2 パターンによる変数束縛

変数に束縛するときにパターンを使って分解出来ることを覚えていますか？ この分解束縛のパターンでも参照を使うことが出来ます．

```
struct Account { name: String, pass: String }

fn main() {
    let a = Account { name: String::from("name"), pass: String::from("pass") };
    let Account { name, pass } = a;    // move ownership
    println!("{} {}", name, pass);     // borrow check!! - OK
    println!("{} {}", a.name, a.pass); // borrow check!! - Error
}
```

上記のコードはパターンによる分解束縛時に所有権も移動してしまい，借用チェックでコンパイルエラーになります．その場合は，パターンに `ref` を使って不変参照で受け取ることで借用チェックに通ることになります．

```
struct Account { name: String, pass: String }

fn main() {
    let a = Account { name: String::from("name"), pass: String::from("pass") };
    let Account { ref name, ref pass } = a; // reference
    println!("{} {}", name, pass);     // borrow check!! - OK
    println!("{} {}", a.name, a.pass); // borrow check!! - OK
}
```

13.3 if let 式

match 式のアーム（ワイルドカード以外）が 1 つのときは `if let` 式を使うと短く記述することが出来ます．例えば次のコードを見てください．

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

`if let` 式を使うと次のように短く記述することが出来ます．

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

`if let` と同様に `while let` も使うことが出来ます．

13.4 マッチガード

マッチガードは `match` 式のアームのパターンに，さらに `if` 条件を加えることができるものです．これにより，より複雑なパターンを扱えます：

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

14. エラー処理

14.1 panic!

基本的に復帰不能なエラーが発生したら，どうしようも出来ません．メッセージを表示してプログラムを終了する手っ取り早い方法が `panic!` です

```
fn main() {
    panic!("crash");
}
```

14.2 Result 型

どこかでエラーが発生したとしても，すぐにプログラムを終了させるわけにはなかなかいきません．ある関数の内部でエラーが発生したら，それを呼び出し元に知らせる必要があります．そこで `Result` 型が使われます．

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

ここではファイル処理を考えてみます．既存のファイルを開いて処理をしたいとします．もし，ファイルが無ければ作成します．その場合，最初にファイルを開こうとしたときにエラーが発生し，そのエラーがファイルが無かったことを表していればファイルを新規に作成するようにします．ファイルを開く処理 `File::open` は `std::io::Result` 型を返します．これは `Result<T, Error>` 型の別名です．

```
use std::{fs::File, io::ErrorKind};

fn main() {
    let f = File::open("hello.txt");
    let f = match f {
        Ok(file) => file,
        Err(ref error) if error.kind() == ErrorKind::NotFound => {
            match File::create("hello.txt") {
                Ok(fc) => fc,
```

```
                Err(e) => panic!("Tried to create file but there was a problem: {:?}", e),
            }
        },
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```

14.3 unwrap, expect

`Option` 型, `Result` 型ともに, 値を取り出す `unwrap` という関数があります. これは, もし値が `None`, `Err` のときに, `panic!` を呼び出します.

```
pub fn unwrap(self) -> T
```

`unwrap` が `panic!` を呼び出すと, 標準のエラーメッセージが表示されますが, `unwrap` の代わりに `expect` を呼ぶと, エラーメッセージに情報を追加することが出来ます.

```
pub fn expect(self, msg: &str) -> T
```

`Option` 型と `Result` 型の `unwrap` は以下のように `match` 式を短くしたものです.

```
option.unwrap()
```

↓

```
match option {
    Some(v) => v,
    None => panic!(...),
}
```

```
result.unwrap()
```

↓

```
match result {
    Ok(v) => v,
    Err(e) => panic!(...),
}
```

14.4 エラー伝搬

`Option` 型, `Result` 型を返す関数の中で, 値が `None` または `Err` のときに, 処理を中断して呼び出し元に値を返す仕組みが用意されています. それは `?` 演算子を使います.

```
fn hoge() -> Option<i32> {
    let a = Some(10);
    let b = a?;
    Some(b)
}
```

```
fn hoge() -> Option<i32> {
    let a = None;
    let b = a?; // return Option<i32>::None
    Some(b)
}
```

14.5 コンビネータ

`Option` 型，`Result` 型も**コンビネータ**です．このコンビネータがエラー処理のコードを大幅に削減してくれます．手続き型であれば，１つ１つの関数呼出しの結果がエラーか無効な値かを確認していきます．これだと，確認コードが大量に出来てしまいます．そこで，まずはエラー伝搬です．関数が `Option` か `Result` を返せば，`?` 演算子を使ってチェーン方式で処理を記述することが出来ます．途中でエラーが発生すれば，処理を打ち切ってエラー伝搬されます．

```
let ret = open()?.read()?.replace()?.write()?.close()?;
```

コンビネータとは簡単に言うと，**高階関数**のことで，高階関数とは関数を引数に取る関数のことです．例えば，関数 $f(x)$ と $g(x)$，これらの合成関数が $(f \circ g)(x) = f(g(x))$ とします．この場合，この $\circ$ がコンビネータで，２つの関数を取っています．先程の `open.read.replace.write.close` で考えてみると `close(write(replace(read(open()))))` の関係に見えないでしょうか．ここで `.` 演算子がコンビネータであり，その役を担っているのが `Option` 型と `Result` 型と考えることが出来ます．

`Option` 型，`Result` 型にはコンビネータとしての便利なメソッドが多く用意されています．基本的なものとして，`map` は値に関数を適用して，その結果をコンビネータに変換します．

```
pub fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Option<U>     // Option
pub fn map<U, F: FnOnce(T) -> U>(self, op: F) -> Result<U, E> // Result
```

`and_then` は関数を適用して，その結果をそのまま返します．つまり，`and_then` に渡す関数はコンビネータを返します．

```
pub fn and_then<U, F: FnOnce(T) -> Option<U>>(self, f: F) -> Option<U>        // Option
pub fn and_then<U, F: FnOnce(T) -> Result<U, E>>(self, op: F) -> Result<U, E> // Result
```

コンビネータは型を合わせる必要があります．`Option` のメソッドに渡す関数は．単純に `T` 型を返す関数や `Option` を返す関数ならよいのですが，`Result` を返す場合にはそのままでは利用できません．そこで，`Option` と `Result` には相互に変換するメソッドがいくつかあります．例えば，`ok_or` は `Option` から `Result` に，`ok` は `Result` から `Option` に変換します．これにより `Option` や `Result` を返す関数を１つのメソッドチェーン内に利用することが出来ます．

コンビネータは `?` 演算子を使っていなければ，途中の処理で `None` になったり，`Err` になった場合，チェーンの最後の型で返ってきます．これによりエラー処理を書く場所が少なくなります．

メソッドのところでも少し触れましたが，コンビネータのメソッドの引数は `self` が多いです．これは，メソッド呼び出しで，`Option<u32>` が `Option<f32>` になったり，`Option<T>` が `Result<T,E>` になったり型の変換を行っているからです．

15. トレイト

15.1 トレイトとは

すでにトレイト，コピートレイト，トレイト境界について触れていますが，ここでさらに詳しく解説します．まずは，おさらいです．**トレイト**（trait）はデータ型を分類する仕組みのことです．また，ジェネリック型に**トレイト境界**を指定することで，その型が特定のトレイトの**インスタンス**であることを強制します．さらに，トレイトには特有のメソッドを実装することが出来，型に対して共通の振る舞いを定義することが出来ます．つまり，あるトレイトのメソッドは，そのインスタンスであれば呼び出せることになります．また，インスタンスによってその振る舞いの実装を変えることも出来ます．

トレイトを定義するには `trait` を使います．トレイト名を指定して，ブロック内に共通のメソッドを定義します．このメソッドはインスタンス側で実装しなければなりませんが，トレイト側で実装することも出来ます．この場合は，インスタンス側はトレイト側の実装をそのまま使うことも出来ますし，その振る舞いを上書き（**オーバーライド**）することも出来ます．

```
pub trait Geometry {
    fn area(&self) -> f64;
    fn name(&self) -> &str { return "Geometry" }
}
```

トレイトの実装は `impl A for B` で指定します．ここで `A` にはトレイト名を，`B` には実装する型を指定します．

```
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width as f64 * self.height as f64
    }
    fn name(&self) -> &str { return "Rectangle" }
}
```

`Geometry` トレイトを定義して，`Rectangle`，`Triangle` というインスタンスを定義する場合は次のようになります．

```
pub trait Geometry {
    fn area(&self) -> f64;
    fn name(&self) -> &str { return "Geometry" }
}

struct Rectangle { width: u32, height: u32 }

impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width as f64 * self.height as f64
    }
    fn name(&self) -> &str { return "Rectangle" }
}
```

```
struct Triangle { bottom: u32, height: u32 }

impl Geometry for Triangle {
    fn area(&self) -> f64 {
        self.bottom as f64 * self.height as f64 * 0.5
    }
    fn name(&self) -> &str { return "Triangle" }
}

fn main() {
    let a = Rectangle { width: 10, height: 20 };
    let b = Triangle  { bottom: 20, height: 5 };
    println!("{} area={}", a.name(), a.area());
    println!("{} area={}", b.name(), b.area());
}
```

15.2 トレイト継承

トレイトは別のトレイトのインスタンスになることが出来ます．これを**継承**と呼ぶこともあります．`trait 継承先 : 継承元` という形で指定します．継承したインスタンスは継承元のトレイトも実装する必要があります．

```
pub trait Geometry {
    ...
}

pub trait Drawable: Geometry {
    ...
}

impl Geometry for Rectangle {
    ...
}

impl Drawable for Rectangle {
    ...
}
```

トレイトのインスタンスを表すには `impl A` のようにします． `A` はトレイト名です．次の関数の引数 `geometry` は `Geometry` のインスタンスでなければなりません．これがトレイト境界です．

```
fn draw(geometry: &impl Geometry) {
    ...
}
```

トレイト境界の指定はより便利な方法があります：

```
fn draw<T: Geometry>(geom1: &T, geom2: &T) {
    ...
}
```

また，トレイトは `+` を使って複数指定することが出来ます：

```
fn draw(geometry: &(impl Geometry + Display))
fn draw<T: Geometry + Display>(geometry: &T)
```

他にも `where` を使って次のように書くことも出来ます：

```
fn draw<T>(geometry: &T)
    where T: Summary + Display
```

ジェネリック型にもトレイト境界を指定することが出来ます．次のコードでは，`Display` と `PartialOrd` のインスタンスの場合なら，`cmd_display` メソッドが実装されます．

```
use std::fmt::Display;

struct Pair<T> { x: T, y: T }

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```

15.3 derive 属性

これまでいくつかのトレイトを見てきました．実際には多くのトレイトがあり，そして，型は多くのトレイトのインスタンスになっています．トレイトのインスタンスにするには `impl` で実装しなければならず，とても面倒です．そこで，規定の実装をしてくれる機能が用意されています．それが `derive` 属性です．実装したいトレイトを型の定義時に `#[derive(trait, ···)]` という形で指定します．

```
#[derive(Debug, Copy, Clone)]
pub struct Vec3 {
```

以下は `derive` 属性でよく使われるものです：

属性	説明
Copy	所有権の移動をせずに，複製を作成するマーカートレイト
Clone	オブジェクトの複製（ディープコピー）を作成できる
Debug	`{:?}` で出力できる
Display	`{}` で出力できる
PartialEq, Eq	`==`，`!=` が使える．Eq はマーカートレイト．
PartialOrd, Ord	`<`，`>`，`<=`，`>=` が使える．Ord は順序付けができる．

15.4 From トレイト

ある型から別の型に変換するときに，便利な `From` トレイトというのがあります． `from` メソッドを対応した型ごとに実装することで，その型から `into` メソッドで変換することが出来ます．

```
#[derive(Debug)]
struct Point { x: f64, y: f64 }

impl From<f64> for Point {
    fn from(input: f64) -> Self {
        Point { x: input, y: input }
    }
}

fn main() {
    let p1 = Point::from(1.0);
    let p2: Point = (1.0).into();
    println!("{:?} {:?}", p1, p2);
}
```

16． RAII

16.1 RAII とは

RAII(**R**esource **A**cquisition **I**s **I**nitialization) とはリソースの確保をオブジェクトの初期化時に行い，リソースの開放をオブジェクトの破棄と同時に行う手法です．これから解説する内容は公式だとスマートポインタと呼ばれていますが，安易にポインタという用語を使うべきでないと思っているし，RAII はポインタに限った話でもないので本書では使いません．

16.2 Deref トレイト

実は，Rust の**参照**は RAII の最も代表となる 1 つです．参照は仮の所有権を保持し，スコープから外れると仮の所有権を破棄します．参照の機能は大きく 2 つあります．参照先のオブジェクトを操作できること，参照先のオブジェ

クトの仮の所有権を破棄することです．このうち，参照先のオブジェクトを操作するには**参照外し**が必要です．これを実現しているのが `Deref` トレイトです．参照外しは参照に対して `*` 演算子を使います．このように，RAII におけるリソースに対して操作をするには `Deref` トレイトを実装し `*` 演算子を使うことです．また，可変参照に対しては `DerefMut` トレイトを実装します．

16.3 Drop トレイト

参照のもう１つの機能が仮の所有権の破棄です．これは `Drop` トレイトで実装します． `Drop` トレイトのインスタンスのオブジェクトは，それが破棄されるときに `drop` メソッドが呼ばれます．このメソッドでリソースの開放処理を行います． `drop` メソッドは可変参照を引数に取ります．

```
impl Drop for Resource {
    fn drop(&mut self) {
        ...
    }
}
```

`drop` メソッドは基本的にオブジェクトが破棄されるときに自動で呼び出されますが，明示的に呼びたい場合があるかもしれません．その場合は， `std::mem::drop` 関数で強制的に呼び出してオブジェクトを破棄することが出来ます．ただし，あまり使うべきではありません．

16.4 メモリ

Rust では次の３つのメモリ領域があります． １つ目は**データメモリ**で，静的データが格納されています．静的データはプログラム実行中に存在するデータのことです． ２つ目は**スタックメモリ**です．これは変数や関数呼び出し時の引数など一時的な格納場所で，コンパイラが最適化しやすく高速にデータ操作が出来ます． ３つ目は**ヒープメモリ**です．ここにはプログラム実行中に利用できるメモリで，スタックメモリよりも大きなサイズを利用することが出来ます．ただし，利用するにはオーバーヘッドがかかります．また，スタックメモリのサイズはヒープメモリに比べてかなり限られているので，ヒープメモリを積極的に利用することになります．

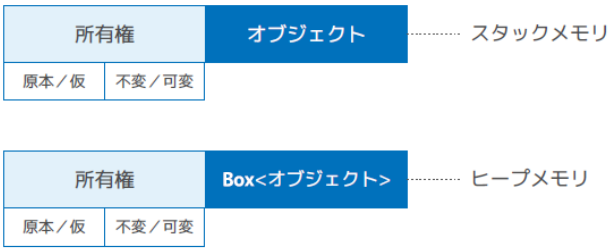
16.5 Box

これまで，メモリを意識してきませんでした．基本的に作成したオブジェクトはスタックメモリに置かれます．また， `static` に指定したオブジェクトや文字列リテラルなどはデータメモリに置かれます．では，ヒープメモリに置くにはどうすればよいでしょうか．それが `Box<T>` です．使い方は簡単で， `Box::new` または `Box::<T>::new` にオブジェクトを渡すだけです．

```
let a = Box::new(10);        // type inference
let a = Box::<i32>::new(20); // explicit type
let a = 30;                  // immutable object
let b = Box::new(a);         // move object from stack memory to heap memory
let c = *b;                  // dereference
```

この図，覚えてますか？

`Box` を表すと…



16.6 Rc

Rc(**R**eference **C**ount) とは**参照カウンタ**のことです．原本の所有権を束縛できるのは１つだけです．参照を使えば，仮の所有権を作成することができます．しかし，参照は制約が強いです．同じオブジェクトを複数から束縛することは出来ないのでしょうか．それを可能にするのが参照カウンタで，`Rc<T>` です．`Rc` は原本または仮の所有権を保持することが出来ます．基本的な機能としては参照と変わらず，仮の所有権を作成します．ただし，`&` 演算子ではなく `clone` というメソッドです．

```
use std::rc::Rc;

let a = Rc::new(10);
let b = a.clone();
```

この `clone` メソッドはディープコピーの `clone` と勘違いしやすいため，関連関数の `clone` を使うほうが良いらしいです．

```
use std::rc::Rc;

let a = Rc::new(10);
let b = Rc::clone(&a);
```

通常は，原本の所有権を束縛した変数が破棄されるときは，仮の所有権を持った変数は存在してはいけません．しかし，`Rc` の場合は，原本の所有権を束縛した変数が破棄されたときに，`clone` で作成した仮の所有権を束縛した変数が存在することができます．このとき，オブジェクトは破棄されず，すべての仮の所有権を束縛した変数が破棄されたときにオブジェクトが破棄されます．
また，この図が出てきました．

`Rc` を表すと…

`Rc` はオブジェクトをヒープメモリに置きます．なので，`Box` の代わりに使うことができます．

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(10);
    let b = a.clone();
    println!("{} {}", a, b);
}
```

17．内部可変性

Rust には制限付きで，不変オブジェクトを安全に可変にする方法が用意されています．この実装は内部可変性パターン（Interior mutability）と呼ばれるものが使われています．ここでは，詳しく説明しませんが，内部的には `unsafe` で実装されています．この機能を使うには `Cell`，`RefCell` というものを使います．興味がある人は調べてみてください．

またまた，この図が出てきました．

`Cell`，`RefCell` を表すと…

18．クロージャ

18.1 クロージャとは

クロージャとは，簡単に言うと，変数に束縛できたり，関数の引数として渡すことのできる名前のない関数（無名関数）のことです．クロージャはその呼び出し元のスコープにある変数を**キャプチャ**することも出来ます．厳密に言うと，無名関数の中で束縛していない変数のことを自由変数と言い，自由変数をまとめた環境を無名関数のスコープ内に閉じこめたものをクロージャと呼びます．

クロージャは `||` で定義します．引数があれば `|param1, param2|` のように `||` の間に入れます．その後に `{}` で本体を記述します．本体が式１つだけなら `{}` を省略することが出来ます．次のコードは関数とそれと同じ振る舞いをするクロージャの例です．

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

クロージャはそれぞれ独自（unique）の型を持っています．クロージャは `Fn`，`FnMut`，`FnOnce` トレイトのどれかのインスタンスです．それぞれ `&self`，`&mut self`，`self` を内部的に引数として受け取っているかどうかの違いがあります．また，`Fn` は `FnMut` を，`FnMut` は `FnOnce` を継承しています．

自由変数をまとめた環境をどのように扱うかで，どのトレイトのインスタンスになるかが決まります．まず，すべてのクロージャは必ず `FnOnce` のインスタンスになります．そして，無名関数の中で環境から所有権を移動することがなければ（可変参照は出来る），`FnMut` のインスタンスになります．さらに，環境を変更しないのであれば，不変参照となるので `Fn` のインスタンスになります．

自由変数が環境にまとめられるとき，自由変数が束縛しているオブジェクトがコピートレイトのインスタンスであれば，コピーが作成されます．もし，自由変数をクロージャの中だけで使用するということが分かっているならば，環境にまとめられるときに，コピーではなく所有権を移動することが出来ます．それを行うには，クロージャの前に `move` を指定します．

18.2 Fn と fn

`Fn` トレイトと `fn` というキーワードは別ものです．`fn` は関数定義で使いますが，`fn` は型でもあります．そして，`fn` のことを**関数ポインタ**と言います．`fn` は `Fn` のインスタンスなので，`FnMut`，`FnOnce` のインスタンスでもあります．

18.3 Sized トレイト

Rust は静的型付け言語です．静的型の特徴の１つとして，型のサイズがコンパイル時に分かることです．しかし，コンパイル時にサイズがわからないこともあります．Rust は型のサイズが分かっているとき，自動でその型を `Sized` トレイトのインスタンスにします．Rust は型が `Sized` トレイトのインスタンスであることを仮定し，それを制約します．つまり，関数の引数などは `Sized` トレイトのインスタンスでなければなりません．ジェネリック型も同じです．それに対して，例えば，`str` 型は実行時にサイズが決まるので，そのような型のことを**動的サイズ型**（Dynamically sized types: DST）といいます．

トレイトは `Sized` トレイトの対象になりません．トレイトはデータ型の分類の仕組みであり，サイズは考慮していないからです．ここで，クロージャに話を戻します．クロージャはトレイトで実装されているので，コンパイル時にサイズがわからないのです．例えば，次のようにクロージャを返す関数はエラーになります．

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
// error[E0277]: the trait bound ‘std::ops::Fn(i32) -> i32 + ’static:
// std::marker::Sized‘ is not satisfied
```

このような動的サイズ型をどうすれば `Sized` トレイトのインスタンスにすることができるかということですが，参照（`&`）にするか，`Box` にするかです．例えば，`Box` を使えばクロージャを次のように返すことが出来ます．

```
fn returns_closure() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

既に述べたように Rust は引数の型が `Sized` であることを制約します．例えば次のようなコードがあります．

```
fn put<T: std::fmt::Debug>(a: &T) {
    println!("{:?}", a);
}

fn main() {
    put("hoge");
}
```

この `T` は文字列スライスである `str` で，関数 `put` の引数は参照で受け取っています．`str` は動的サイズ型なので参照を付けていますが，これはエラーになってしまいます．これは暗黙的に `T: Sized` となっているためで，引数側に参照を付けても `T` が `Sized` でなければなりません．しかし，引数側で参照を付けているので，コードとしては動的サイズ型を指定しても問題ありません．そこで，`T` が動的サイズ型を受け入れられるように `?Sized` を指定するこができます．

```rust
fn put<T: std::fmt::Debug + ?Sized>(a: &T) {
    println!("{:?}", a);
}

fn main() {
    put("hoge");
}
```

これでコンパイルが通ります．

18.4 参照を返す関数

`str` 型は動的サイズ型です．このままでは `Sized` にならないので，文字列スライスは `&str` 型です．クロージャを返すところでは参照を使わずに `Box` を使いました．参照を使っても返すことができるのですが，関数から参照を返すときには**ライフタイム**（lifetime）というものを考慮しなければなりません．個人的にこのライフタイムは余程の理由がない限り扱うべきでないものと思っているので，本書では詳しく扱いません．なので，関数から参照を返すのは可能なかぎり避けましょう．また，構造体にも参照のフィールドを持つことも出来ますが，こちらもライフタイムが必要になってしまいます．

18.5 静的と動的

Rust は静的型付け言語にもかかわらず，動的サイズ型もサポートしているのが強みでもあります．これにより静的ディスパッチおよび動的ディスパッチの両方を実現することが出来ます．動的サイズ型は参照や `Box` を使うことで扱えることがわかりました．クロージャはトレイトであり，動的サイズ型であり，参照や `Box` を使うことでオブジェクトとして扱えるようになります．このようなオブジェクトを**トレイトオブジェクト**といいます．

トレイトオブジェクトは，トレイトのメソッドのみ呼び出せることになります．トレイトオブジェクトは，そのトレイトのインスタンスであればどの型のオブジェクトでも置き換えることができます．このように，トレイトオブジェクトを扱う側は実際のオブジェクトの型を知らなくても，そのメソッドを呼び出せるということ，そしてオブジェクトの型によってメソッドの動作を変えることが出来ることになります．これらの仕組みを**動的ディスパッチ**といいます．

ジェネリック型や `impl Trait` で指定した型はコンパイル時に型が決まりますので静的です．この `Trait` には任意のトレイト名を指定します．トレイトの型は `Sized` ではないので，参照や `Box` で指定する必要があるのですが，静的である `impl Trait` と区別しやすいように，動的であることを明示する `dyn` が導入され，`dyn Trait` という型を使います．よって，`&dyn Trait` や `&mut dyn Trait`，`Box<dyn Trait>` という形で利用します．

ここでクロージャを返す関数を振り返ってみます．動的と静的を区別するために `dyn` を指定するのですが，これは後から導入されたものなので，省略してもコンパイルは通ります．ただし，警告で付けるように促されるので，実際は次のようになります：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

実は `impl` を使うと簡単に静的として扱えます：

```
fn returns_closure() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
```

19. ライフタイム

ライフタイムというのは参照が有効になるスコープのことです．参照は原本の所有権が存在している限り有効なもので，借用チェックによって厳密にチェックされます．元々，関数に渡すために参照で仮の所有権を渡して破棄してもらう仕組みなのに，それを関数の返り値として返すとはおかしな話です．Rust はパフォーマンスを最優先しているので，仕方ないとは思います．返せたほうが便利なときもあるでしょう．Rust の初期版では参照を使っているところは明示的にすべてライフタイムを指定する必要があったようですが，今はかなり緩和されました．

ライフタイムは `&` 演算子の後ろに指定します．慣例的に `a,b,c,…` と指定します．

```
&i32        // a reference
&'a i32     // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

特別なライフタイムの１つに `'static` があります．これはプログラム実行中にずっと存在するライフタイムです．ライフタイムが `'static` なオブジェクトはデータメモリに置かれます．例えば，文字列リテラルは `'static` なライフタイムを持っています．

ライフタイムを指定したコードは本当に見づらいので，なるべくライフタイムを指定するようなコードは書かないほうがいいとは思います．

```
impl<'i, 't, 'a, R, P, E: 'i> RuleListParser<'i, 't, 'a, P>
where
    P: QualifiedRuleParser<'i, QualifiedRule = R, Error = E>
        + AtRuleParser<'i, AtRule = R, Error = E>,
{
    pub fn new_for_stylesheet(input: &'a mut Parser<'i, 't>, parser: P) -> Self {
        // ...
    }
}
```

20. 並列処理

20.1 スレッド

最も基本的な並列処理はスレッドを作成することです．スレッドを作成するには `thread::spawn` を使います．引数にはクロージャを指定します．

```
use std::thread;
thread::spawn(|| {
    // thread code
});
```

`thread::spawn` は `JoinHandle` 型を返します． `join()` メソッドを呼び出すことで終了を待ちます．`join()` は `Result` 型を返します．

```
let handle = thread::spawn(|| {
    // thread code
});
handle.join().unwrap();
```

クロージャの環境をスレッド間で共有することは通常の方法では出来ません．コピーを作成できるなら，環境にコピーされますが，そうでないなら所有権を移動しなければなりません．その場合は `move` を使います．

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

複数のスレッド間で状態を共有するには**排他制御**が必要です．これを行うために `Mutex` があります． `lock` メソッドでリソースをロックします． `lock` メソッドは `LockResult` 型を返します．また， `LockResult` 型は RAII である `MutexGuard` オブジェクトを束縛しているので，自動でロックを解除します．

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }
```

```
    println!("m = {:?}", m);
}
```

排他制御を行う `Mutex` は出来ましたが，このオブジェクトをスレッド間で共有しなければなりません．所有権の共有は `Rc` で出来ますが，これのマルチスレッド版である `Arc` を使います．

```
fn main() {
    let counter = Arc::new(Mutex::new(0));

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        ...
```

おや，またこの図が出てきました．

マルチスレッド版の `Rc` である `Arc`，排他制御を行う `Mutex` の関係を表すと…

20.2 Send + Sync

マルチスレッドではオブジェクトの所有権の操作を考慮する必要があります．オブジェクトの所有権がスレッド間で移動できる場合は，`Send` マーカートレイトのインスタンスになります．**マーカートレイト**とは，メソッドを持たないトレイトのことで，トレイト境界に使うためのものです．`Sized` トレイトもマーカートレイトの１つです．次に，複数のスレッドから安全に参照できる場合は `Sync` マーカートレイトのインスタンスになります．これは参照 `&T` が `Send` ならば，`T` 型は `Sync` であり，参照が別のスレッドに送ることができるという意味になります．

ほとんどのプリミティブ型は `Send+Sync` です．また，`Sync` であるデータ型で構成された型は，それもまた `Sync` です．これは自動的にそれぞれのインスタンスになります．これらを手動で実装するのは安全ではありません．マルチスレッドに対応していない `Rc` は `Send` でもなく，`Sync` でもありません（かわりに `!Send` や `!Sync` の非実装マーカートレイトがつきます）．それに対して，`Arc` は `Send+Sync` です．また，`Arc`

が束縛する型もまた `Send+Sync` である必要があります．もし，`Send+Sync` でないなら，`Mutex` を利用します．

次の図は `Rc` と `Arc` のドキュメントからの抜粋です．`Trait Implementations` にありますが，新規に定義したデータ型などは基本的に `Send` と `Sync` は自動的につくので `Auto Trait Implementations` で確認することが出来ます．

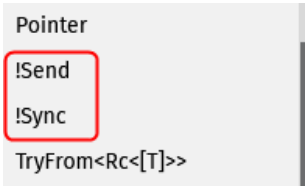


Fig.1.1: std::rc::Rc

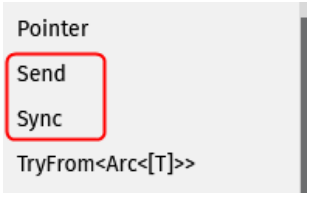


Fig.1.2: std::sync::Arc

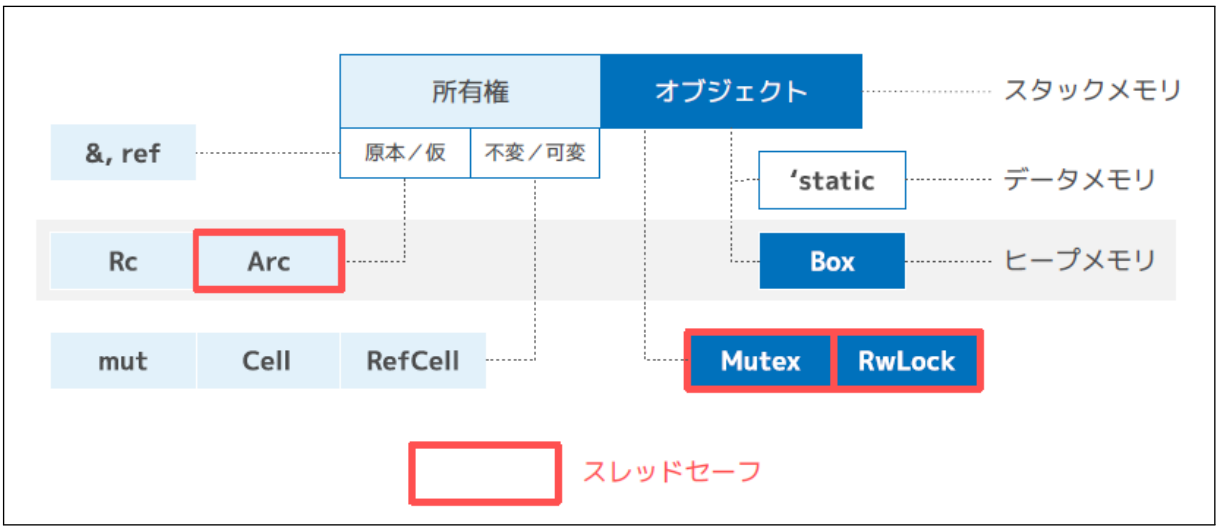
20.3 RwLock

排他制御を行うのは `Mutex` 以外に `RwLock` があります．`Mutex` は常に１つのスレッドがリソースの操作をすることが出来ますが，`RwLock` の場合は，不変参照だけなら複数のスレッドが同時にリソースをロックすることができ，可変参照のときだけ，１つのスレッドに制限するものです．他にアトミック変数というのもあります．これはプリミティブ型のみしか扱えませんが，`Mutex` よりは高速に動作します．処理速度に問題がなければ基本的に `Mutex` を使うのがいいでしょう．`RwLock` やアトミック変数の詳細は公式リファレンスなどを参照してください．

21．所有権のまとめ

さて，この図に戻ってきました．

といっても，今回が最後です．ここまで長かったですね．これまでの内容をまとめてみましょう．



22．マクロ

マクロはメタプログラミングの１つで，コードを展開してくれるものです．特に関数の可変長引数に対応していて，多用します．関数マクロは，関数の最後に `!` 演算子が付いたものです． `print!` ，`println!` は標準出力に文字列を出力するマクロです． `eprint!` ，`eprintln!` は標準エラーに文字列を出力します． `dbg!` は式を評価してデバッグ表示してくれます． `unimplemented!` は未実装を表し， `panic!` を起こします． `todo!` も同じですが，ニュアンスが異なり，「まだ未実装」という意味です．マクロの機能は多いので，詳しくは公式ドキュメントなどを参照してください．

23．イテレータ

イテレータは連続したオブジェクトを順番に取り扱うための機能を提供するオブジェクトです．配列やスライス，後で解説する**コレクション**でよく使います．イテレータは `Iterator` トレイトのインスタンスです：

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;

    // methods with default implementations elided
}
```

ここで， `type Item` は**関連型**と呼ばれるもので，インスタンスはこの `Item` 型を定義しなければなりません．

イテレータには多くの便利なメソッドが定義されています．いくつか紹介します．まずは `zip` です．これは別のイテレータを受け取って合成し，新しいイテレータを返します．要素はタプルになります．このようなイテレータから別のイテレータを作るメソッドは**アダプタ**と呼ばれています．

```
let a1 = [1, 2, 3];
let a2 = [4, 5, 6];
let mut iter = a1.iter().zip(a2.iter());
```

`map` は各要素に関数を適用します：

```
let a = [1, 2, 3];
let mut iter = a.iter().map(|x| 2 * x);
```

`filter` は各要素に対して関数を適用し，`true` を返した要素だけを取り出します：

```
let a = [0i32, 1, 2];
let mut iter = a.iter().filter(|x| x.is_positive());
```

`fold` は状態を持ち，各要素に対して関数を適用して状態を更新し，その状態を返します：

```
let a = [1, 2, 3];
// the sum of all of the elements of the array
let sum = a.iter().fold(0, |acc, x| acc + x);
```

`collect` はイテレータの全要素をコレクションに変換します：

```
let a = [1, 2, 3];
let doubled: Vec<i32> = a.iter()
                         .map(|&x| x * 2)
                         .collect();
```

`enumerate` はインデックスと各要素のペアをタプルにします：

```
let a = ['a', 'b', 'c'];
let mut iter = a.iter().enumerate();
// (0, &'a'), (1, &'b'), (2, &'c')
```

`inspect` はイテレータの各要素を確認するための `map` です．`map` では `println!` などが使えないためです：

```
let a = [1, 4, 2, 3];
let sum = a.iter()
    .cloned()
    .inspect(|x| println!("about to filter: {}", x))
    .fold(0, |sum, i| sum + i);
println!("{}", sum);
```

コレクションなどイテレータを返すメソッドには `iter()`，`iter_mut()`，`into_iter()` があります．それぞれ，引数に `&self`，`&mut self`，`self` を受け取ります．

24. コレクション

24.1 コレクションとは

Rust の標準ライブラリには便利な複数のオブジェクトを管理するデータ構造が用意されています．これらを**コレクション**と呼びます．コレクションはヒープメモリに置かれます．ここでは代表的なコレクションを解説します．

24.2 Vec

ベクタ `Vec<T>` は伸縮可能な配列です．空のベクタを作成するには `Vec::new` を使います：

```
let v: Vec<i32> = Vec::new();
```

初期値を指定してベクタを作成する場合は `vec!` マクロを使います：

```
let v = vec![1, 2, 3];
```

各要素を取り出して処理する一般的な方法は `for` 式を使います：

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

`Vec<T>` のメソッドの一部です：

```
insert(&mut self, index: usize, element: T)
remove(&mut self, index: usize) -> T
push(&mut self, value: T)
pop(&mut self) -> Option<T>
append(&mut self, other: &mut Vec<T>)
clear(&mut self)
len(&self) -> usize
is_empty(&self) -> bool
first(&self) -> Option<&T>
first_mut(&mut self) -> Option<&mut T>
last(&self) -> Option<&T>
last_mut(&mut self) -> Option<&mut T>
```

24.3 String

文字列を表す `String` もコレクションです．内部では **UTF-8** でエンコードされたデータです．文字列型には `OsString`，`OsStr`，`CString`，`CStr`，`String`，`str` などがあります．それぞれ，エンコード方式が違います． `String` と `str` のようにペアになっており， `str` はスライスです．

文字列の生成は `String::new` です：

```
let mut s = String::new();
```

文字列以外の型から，文字列に変換するには `to_string` メソッドが便利です．これは `Display` トレイトのインスタンスなら自動で実装してくれます．

```
let i = 5;
let five = i.to_string();
```

文字列リテラルからの作成は `String::from` を使います：

```
let five = String::from("5");
```

文字列から数値型に変換するには `parse` を使います．これは `Result` 型を返します：

```
let a_string = String::from("5");
let b = a_string.parse::<i32>()?;
```

書式付きで文字列を作成するには `format!` マクロを使います：

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
let s = format!("{}-{}-{}", s1, s2, s3);
```

`String` のメソッドの一部です：

```
push_str(&mut self, string: &str)
push(&mut self, ch: char)
pop(&mut self) -> Option<char>
as_bytes(&self) -> &[u8]
truncate(&mut self, new_len: usize)
insert(&mut self, idx: usize, ch: char)
insert_str(&mut self, idx: usize, string: &str)
remove(&mut self, idx: usize) -> char
len(&self) -> usize
is_empty(&self) -> bool
clar(&mut self)
chars(&self) -> Chars<'_>
bytes(&self) -> Bytes<'_>
starts_with<'a,P>(&'a self, pat: P) -> bool
ends_with<'a,P>(&'a self, pat: P) -> bool
find<'a, P>(&'a self, pat: P) -> Option<usize>
rfind<'a, P>(&'a self, pat: P) -> Option<usize>
trim(&self) -> &str
```

24.4 HashMap

`HashMap<K,V>` は**連想配列**です．オブジェクトにキーを結びつけて管理します．空の連想配列を作成するには `HashMap::new` を使い，新しい要素を挿入する場合は `insert` を使います：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

キーに対応した要素を取得するには `get` を使います：

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

各要素を取り出す場合は `for` 式で，キーとオブジェクトを分解束縛します：

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

キーがまだ存在していないときに挿入する場合は `entry` と `or_insert` を使います：

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);
println!("{:?}", scores);
```

２つのベクタから連想配列を作成するにはイテレータを使って，`zip`，`collect` を使う方法があります：

```rust
use std::collections::HashMap;

let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let mut scores: HashMap<_, _> =
    teams.into_iter().zip(initial_scores.into_iter()).collect();
```

`HashMap<K,V>` のメソッドの一部です：

```rust
keys(&self) -> Keys<'_, K, V>
values(&self) -> Values<'_, K, V>
values_mut(&mut self) -> ValuesMut<'_, K, V>
iter(&self) -> Iter<'_, K, V>
iter_mut(&mut self) -> IterMut<'_, K, V>
len(&self) -> usize
clear(&mut self)
entry(&mut self, key: K) -> Entry<'_, K, V>
contains_key<Q: ?Sized>(&self, k: &Q) -> bool
insert(&mut self, k: K, v: V) -> Option<V>
remove<Q: ?Sized>(&mut self, k: &Q) -> Option<V>
```

25. Cargo

Cargo はパッケージ管理ツールです．**パッケージ**とは１つ以上のクレートを含んだもののことです．そして，**クレート**とは Rust プログラムをビルドしたもので，バイナリクレート（実行可能ファイル）とライブラリクレートの２つがあります．パッケージは複数のバイナリクレートを含めることができますが，ライブラリクレートは１つまでしか含めることが出来ません．**モジュール**はクレートの中でグループ化されたコードのことで，読みやすさと再利用性を高めるためのものです．また，**プライバシー**を設定することができ，内部の実装を利用できなくすることも出来ます．

`init` コマンドを使うと，実行したフォルダ内に，単一のバイナリクレートを含んだパッケージの作成環境が作られます．

```
cargo init
```

実行すると次のような構成になります．

```
(workspace)
├── src
│   └── main.rs
├── cargo.toml
└── .gitignore
```

`cargo.toml` は構成ファイルです． `src/main.rs` が**クレートルート**です．このファイルがモジュール構造の起点となります． `build` コマンドを実行するとビルドが始まります．Cargo は必要な外部パッケージなどを自動でダウンロードしてビルドします．依存する外部パッケージは `cargo.toml` に記述し，実際にダウンロードしたパッケージのバージョンを `cargo.lock` ファイルに書き込みます．

```
cargo build
```

`run` コマンドを実行すると，作成した実行可能ファイルを起動します．ビルドが必要な場合は自動的にビルドしてくれます．

```
cargo run
```

`build` ，`run` コマンドは標準でデバッグ情報付き実行可能ファイルを作成します．オプション `--release` をつけることで，リリース用の実行可能ファイルを作成することができます．また，Rust にはプロファイルというものがあり， `dev` と `release` があります． `--release` オプションは `release` プロファイルを使用することを明示するものです．このプロファイルをカスタマイズすることができます．例えば，最適化オプションなどです．詳しくはドキュメントを参照してください．

```
cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
cargo build --release
    Finished release [optimized] target(s) in 0.0 secs
```

`check` コマンドはコンパイルチェックをします． `build` コマンドとは違ってリンクを行わないので，コンパイルが通るかのチェックはこちらのほうが早いです．

```
cargo check
```

外部パッケージを使用したい場合は， `Cargo.toml` の `dependencies` セクションに使用したいパッケージを指定します．例えば `rand` パッケージを使う場合は次のようにします：

```
[dependencies]
rand = "0.8.0"
```

現在のパッケージ構成を確認したい場合は以下のコマンドを使います：

```
cargo tree
```

バイナリパッケージをインストールすることができます．その場合は `install` コマンドを使います．

```
cargo install <package>
```

26. モジュール

26.1 モジュールを定義

モジュールの定義は `mod` を使います．モジュールの本体は `{}` で囲みます．モジュールの中にモジュールを定義することも出来ます．

```
mod M1 {
    mod M2 {
        mod M3 {
            fn hoge() {}
        }
    }

    mod M4 {
        fn foo() {}
        fn bar() {}
    }
}
```

モジュールは同じモジュール内に対してだけ公開された状態になります．そこで，外部のモジュールに対しても公開するには `pub` を使います． `pub` はモジュール，関数，構造体などに１つずつ設定することが出来ます．

```
pub mod M1 {
    pub mod M2 {
        pub mod M3 {
            pub fn hoge() {}
        }
    }
}
```

26.2 パス

モジュールを利用するには**パス**が必要です．モジュールの起点はクレートルートで，パスは `crate` になります．前のコードが `src/main.rs` に含まれていた場合，それぞれのパスは次のようになります．

```
crate
└── M1
    ├── M2
    │   └── M3
    │       └── hoge
    └── M4
        ├── foo
        └── bar
```

パスの指定には２種類あります．絶対パスと相対パスです．絶対パスはクレートルートを表す `crate` ，または外部パッケージおよび標準ライブラリの場合はパッケージ名から指定することが出来ます．相対パスの場合は `self` ，

または `super` を使います． `self` は現在のモジュールから， `super` は親のモジュールからの指定になります．パスの区切りは `::` を使います．ちなみに `self::` は省略出来ます．

```
crate::M1::M2::M3::hoge
```

26.3 モジュールの利用

モジュールを利用するには `use` を使ってパスを指定します：

```
use crate::M1::M2::M3::hoge;
use std::fmt::Result;
```

`use` で指定したパスに `as` で別名をつけることが出来ます：

```
use std::io::Result as IoResult;
```

`use` で指定するパスにおいて，あるモジュールから別々のパスを指定する場合，それぞれのパスを `use` で指定すると必要な行が増えてしまいます．そこで，パスのリストを記述することが出来ます：

```
use crate::M1::{M2::M3, M4};
```

また，あるモジュール以下をすべて現在のスコープで利用する場合はグロブ（ `*` ）を指定することができます：

```
use crate::M1::*;
```

26.4 モジュールツリー

クレートそのものがモジュールであり，クレートルートは `src/main.rs` ファイルで，パスは `crate` でした．Rust のモジュールシステムはクレートルート以下のフォルダとファイルもまたモジュールと見なします．これによって別々のファイルに実装を分けることが出来るわけです．このフォルダとファイルによるモジュールの作り方が 2 種類あります．先に一般的な方法を説明します．

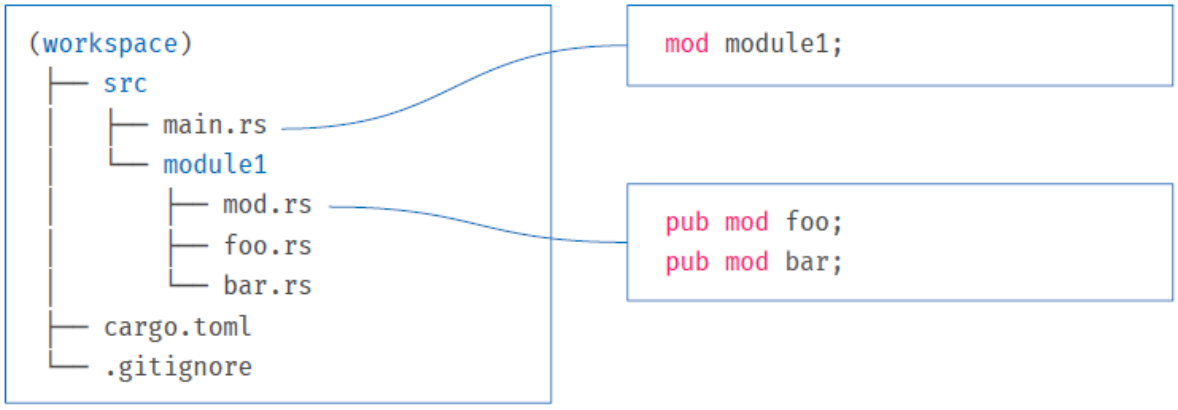
`mod` では `{}` で囲む他に，指定した名前と同じファイルを同じフォルダ内から検索して，その中身を挿入することが出来ます．例えば， `src/hoge.rs` というファイルがあるとします． `src/main.rs` から `mod hoge;` とすれば `src/hoge.rs` の中身を `src/main.rs` ファイルに挿入します．ここで `src/hoge.rs` の中身が次のようになっているとします．

```
pub fn hoge() {}
```

この場合， `src/main.rs` からは `crate::hoge::hoge()` で呼び出すことが出来ます．

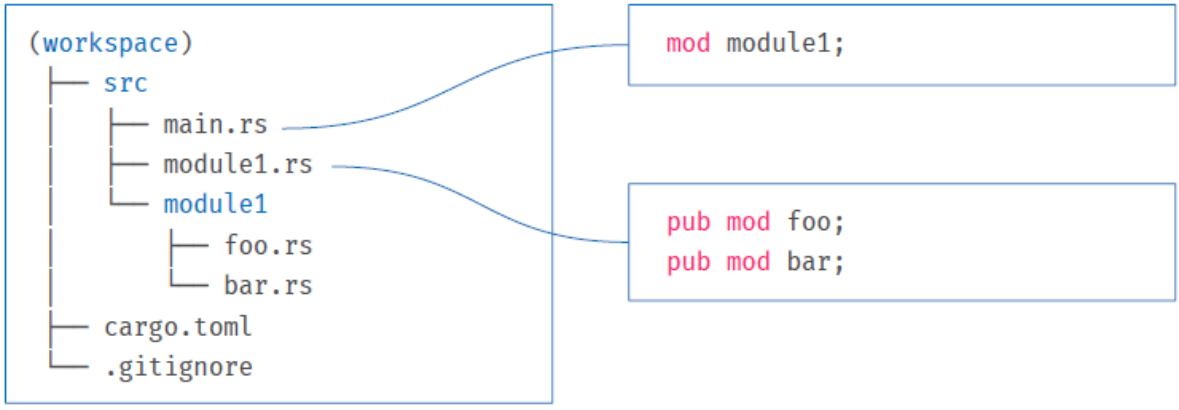
次に右図のようなフォルダ構成を考えます．このような構成にすると，`src/main.rs` から `mod module1;` とすることで，`src/module1/mod.rs` ファイルが読み込まれます．そのファイルの中身は `pub mod foo;` とします．これで，`src/main.rs` から `src/module1/foo.rs` のモジュールを利用することが出来ます．同じように `pub mod bar;` を `mod.rs` に追加すれば `bar.rs` モジュールも利用できるようになります．このようにモジュールの階層を作ってプログラムを構築していきます．これを**モジュールツリー**といいます．

```
(workspace)
├── src
|   ├── main.rs
|   └── module1
|       ├── mod.rs
|       ├── foo.rs
|       └── bar.rs
├── cargo.toml
└── .gitignore
```



もう１つのモジュールツリーの作り方ですが，右図を見てください．今度は `mod.rs` ファイルの代わりに，フォルダと同じファイル `module1.rs` が存在しています．このファイルの中身は `mod.rs` と同じになります．ファイルの構成が違いますが，モジュールツリーは同じなので，コードに変更はありません．

```
(workspace)
├── src
|   ├── main.rs
|   ├── module1.rs
|   └── module1
|       ├── foo.rs
|       └── bar.rs
├── cargo.toml
└── .gitignore
```



26.5 モジュールの再公開

`pub use` を使うことで，外部モジュールからでも，指定したパスで利用できるようになります．これを再公開といいます．

```
pub use crate::module1;
```

27．ユニットテスト

Rust にはテストコードを記述する機能が用意されています．テストを実行するには以下のコマンドを実行します：

```
cargo test
```

実行するユニットテストの関数には `test` 属性を付けます：

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}
```

テスト時のみ（`cargo test`）ビルドするモジュールを作ることが出来ます．それにはモジュールの前に `cfg(test)` 属性を付けます．

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

テストに便利なマクロがいくつか用意されています． `assert!` マクロは引数が `true` かどうかをテストします．また， `==` や `!=` 演算子を使う代わりに `assert_eq!` ， `assert_ne!` マクロが用意されています．

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert!(2 == 2);
        assert_eq!(2 + 2, 4);
        assert_ne!(2 + 2, 5);
    }
}
```

`assert!` マクロや，`unwrap` などは `panic!` を呼び出す場合があります．場合によっては `panic!` が呼び出されることを期待したテストコードを記述したい場合があります．しかし，テストは通常 `panic!` を起こすと失敗になります．そこで，`should_panic` 属性を付けることで，`panic!` を起こすことがテストの目的であることを明示します：

```
#[test]
#[should_panic]
fn it_works() {
    panic!();
}
```

28． Tips

28.1 未使用の変数

プログラムの中に未使用の変数があればコンパイル時に警告が出ます．その場合は変数の前にアンダースコア（`_`）を付けることで，警告を抑制できます．

28.2 分解束縛

分解束縛において，多くのフィールドがあるときに，必要なものだけ束縛して，残りは無視したいことがあるかもしれません．そのときは，`..` を使用します：

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

28.3 ドキュメント

オフラインで Rust のドキュメントを参照するには次のコマンドを使います

```
rustup doc
```

現在のワークスペースに関するドキュメントを参照するには次のコマンドを使います

```
cargo doc [--open]
```

Rust には `rustdoc` というソースコードからドキュメントを作成するツールが付属しています．`cargo doc` はワークスペースにあるソースコードおよび，外部パッケージからドキュメントを生成してくれます．コメントにドキュメントを入れるには `///` や `//!` を使います．`///` は宣言に対して，`//!` はコンテキストに対して，ドキュメントを作成します．詳しくは公式ドキュメントを参照してください．

28.4 デバッグ時やテスト時のみ有効なコード

`cfg!` を使うことで環境に合わせたコードを作成することが出来ます．例えばデバッグ時の場合は `debug_assertions` を指定します．

```
if cfg!(debug_assertions) {
    ...
}
```

否定の時は `not` を使います．

```
if cfg!(not(debug_assertions)) {
    ...
}
```

他には，テスト (`cargo test`) 時に有効になる `test` があります．

```
if cfg!(test) {
    ...
}
```

28.5 属性

属性とは追加情報（メタデータ）のことで，宣言やコンテキストに対して付与することができます．よく使うのは，トレイトの規定実装を示す `#[derive(...)]`，ユニットテストの `#[test]`，`#[should_panic]`，`#[ignore]`，そして，コンパイラに伝える `#[allow(...)]` や `#[cfg(...)]` です．コンテキストはそのモジュール内に全て適用されるので，例えばクレートルート (`src/main.rs` など) の先頭に `#![allow(...)]` を指定すると，すべてのモジュールに適用されます．

28.6 演算子のオーバーロード

Rust は新しい演算子の定義をすることができません．任意の演算子のオーバーロードは `std::ops` にあるものだけ可能です．

```
use std::ops::Add;
#[derive(Debug, PartialEq)]
    struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}
```

28.7 型の別名

既存の型に別名を付けることができます．その場合は `type` を使います．

```
type Kilometers = i32;
```

28.8 Prelude

`Prelude` とは標準で使用可能なモジュールのことです．ここには基本的な型や `Box`，`Vec` といった頻繁に使用するものが含まれています．例えば次のような単純な Hello World!! コードがあります．

```
fn main() {
    println!("Hello world!");
}
```

これを，Rust Playground でコード展開してみると，次のようになります．

```
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std;
fn main() {
    {
        ::std::io::_print(
            ::core::fmt::Arguments::new_v1(
                &["Hello world!\n"],
                &match () {
                    () => [],
```

```
                }
            )
        );
    };
}
```

`use std::prelude::v1::*` があるのがわかります．このように，パッケージ内のライブラリを利用するときに最低限必要なものをまとめたものを `prelude` として用意されている場合があります．

28.9 dbg!

`dbg!` マクロは変数や式の結果を出力するのに便利です．次のように使います．

```
let a = 2;
let b = dbg!(a * 2) + 1;
// ^-- prints: [src/main.rs:2] a * 2 = 4
assert_eq!(b, 5);
```

28.10 Display と Debug トレイト

`Display` と `Debug` トレイトは文字列変換やデバッグ出力に便利なトレイトです．`Debug` トレイトは `derive` 属性で規定実装が可能ですが，`Display` はそれが出来ません．例えば，`Debug` トレイトを `derive` で規定実装し，`Display` トレイトを実装する場合は次のようになります．

```
use std::fmt;

#[derive(Debug)]
struct Point {
    x: f64,
    y: f64,
}

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}, {}", self.x, self.y)
    }
}

fn main() {
    let p = Point { x: 1.0, y: 2.0 };
    println!("{}, {:?}", p, p);
}
```

28.11 rustfmt

Rust にはコード整形ツールがついています．それが `rustfmt` です．次のように使います．

```
rustfmt <filename>
```

`cargo fmt` を使うとワークスペース内のファイルに対して実行します．

```
cargo fmt
```

`--check` オプションを指定すると，整形を行わずに警告を表示してくれます． `cargo fmt` にこのオプションをつけるときは， `cargo` のオプションと区別するために `--` を前に付けます．

```
rustfmt <filename> --check
cargo fmt -- --check
```

28.12 clippy

不適切なコードを指摘してくれる clippy というツールがあります．次のように使います．

```
cargo clippy
```

28.13 API ガイドライン

Rust の関数や変数の名前など，コーディングに関する規約がまとめられています．一読することをオススメします．

Rust API Guidelines

第 2 章

レイトレーシング

1．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_rust (https://github.com/mebiusbox/rayt_rust)

開発環境は Windows で，エディタには Visual Studio Code を使っています．構成は次のようになっています．

```
(workspace)
├── src
|    ├── main.rs
|    ├── code101.rs
|    ├── code102.rs
|    ├── ...
|    └── rayt
|        ├── mod.rs
|        ├── float3.rs
|        └── ...
├── cargo.toml
└── .gitignore
```

`src/main.rs` がクレートルートです．各レンダリングした画像と同じモジュール（ `code101` ，`code102` など）があり，１つのファイルにまとめてあります．また，それぞれに共通のものは `rayt` モジュールにまとめてあります．

サンプルコードはダウンロードしたりクローンしたりしたフォルダで， `cargo run` で実行することが出来ます．また，各レンダリング画像に対応した番号を実行するときは `cargo run <no>` で実行できます．

公開しているサンプルコードの `src/main.rs` は，本書の内容とは若干異なっています．本書の `main` 関数は，各コードモジュールにある `run` 関数になっていますので注意してください．

2．プロジェクトの準備

まずは，プロジェクトの準備をします．以下のコマンドで１つのバイナリクレートを作成する環境を作ります．

```
cargo init
```

この場合，`src/main.rs` がクレートルートになります．`main.rs` は次のようになっています．

```
fn main() {
    println!("Hello world!!");
}
```

ビルドは次のようにします．

```
cargo build
```

実行するときは次のようにします．

```
cargo run
```

本書で記載しているコードには先頭にファイル名がコメントでついてあるものがあり，そのファイルに実装することを表しています．もし，このファイル名が省略されている場合は基本的に `src/main.rs` に実装することを表しています．

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．ここでは，色を表す `Color` 型を定義します．

```
struct Color([f64; 3]);
```

これは `f64` 型の３つの長さの配列を持つタプル構造体です．

```
const IMAGE_WIDTH: u32 = 200;
const IMAGE_HEIGHT: u32 = 100;

fn main() {
    let mut pixels: Vec<Color> = Vec::with_capacity(IMAGE_WIDTH as usize * IMAGE_HEIGHT as usize);
    for j in 0..IMAGE_HEIGHT {
        let par_iter = (0..IMAGE_WIDTH).into_iter().map(|i| {
            Color([
                i as f64 / IMAGE_WIDTH as f64,
                j as f64 / IMAGE_HEIGHT as f64,
                0.5,
            ])
        });
        let mut line_pixels: Vec<_> = par_iter.collect();
        pixels.append(&mut line_pixels);
    }
    save_ppm(String::from("render.ppm"), &pixels).unwrap();
}
```

幅 `IMAGE_WIDTH` ，高さ `IMAGE_HEIGHT` の画像を作成します． `Vec::with_capacity` で画素分の配列を作成して，イテレータを使ってグラデーションを作っています． `save_ppm` は PPM 形式で画像を保存します．

```
use std::{fs::File, io::prelude::*};

fn save_ppm(filename: String, pixels: &[Color]) -> std::io::Result<()> {
    let mut file = File::create(filename)?;
    writeln!(file, "P3")?;
    writeln!(file, "{} {}", IMAGE_WIDTH, IMAGE_HEIGHT)?;
    writeln!(file, "255")?;
    for Color([r, g, b]) in pixels {
        let to255 = |x| (x * 255.99) as u8;
        writeln!(file, "{} {} {}", to255(r), to255(g), to255(b))?;
    }
    file.flush()?;
    Ok(())
}
```

`to255` はクロージャで，浮動小数点数 (`f64`) から整数 (`u8`) に変換します．

Fig.2.1: code101

4． ベクトルモジュール

レイトレーシングを行うために必要なベクトル計算を行うモジュールをまずは作ります．既にベクトル計算を行うクレートが沢山ありますが，今回は新規に作ります．共通のモジュールは `src/rayt` フォルダにまとめていきます． `src/rayt` フォルダを作成し，中に `mod.rs` ファイルを作成します．次にベクトル計算のモジュール用のファイル `float3.rs` を作成します．このモジュールは， `f64` 型の 3 つの要素を持つタプル構造体 `Float3` をベースに，ベクトル計算，カラー計算の機能を実装していきます．また，効率よりも簡潔さを重視したため，引数には参照を使わずにコピーするようにしました．

```
// src/rayt/float3.rs
#[derive(Debug, Copy, Clone, PartialEq)]
pub struct Float3([f64; 3]);
```

`Float3` をベクトル型，カラー型，位置ベクトル型として扱うので，それぞれ `Vec3` ， `Color` ， `Point3` として別名を定義します．

```
// src/rayt/float3.rs
pub type Color = Float3;
pub type Vec3 = Float3;
pub type Point3 = Float3;
```

`mod.rs` には `float3.rs` をインポートします．

```
// src/rayt/mod.rs
mod float3;
pub use self::float3::{Float3, Color, Vec3, Point3};
```

すべてのファイルからは以下のコードを先頭に入れることで，共通モジュールを利用できるようにします．

```
use crate::rayt::*;
```

クレートルート（`src/main.rs` のこと）には `rayt` モジュールをインポートします．

```
// src/main.rs
mod rayt;
```

Rust は標準で未使用の関数の警告を出してくれます．これは便利ですが，基本的に無効にしておきたいので，クレートルートの先頭で指定します．

```
// src/main.rs
#![arrow(dead_code)]
```

それでは `Float3` を実装していきましょう．まずはコンストラクタをいくつか定義します．

```
// src/rayt/float3.rs
pub const fn new(x: f64, y: f64, z: f64) -> Self {
    Self([x, y, z])
}

pub const fn zero() -> Self {
    Self([0.0; 3])
}

pub const fn one() -> Self {
    Self([1.0; 3])
}

pub const fn full(value: f64) -> Self {
    Self([value; 3])
}
```

関数の定義で `const` を使っています．これは定数を返す関数を表します．Rust の算術関数は型のメソッドとして用意されています．以下は，`f64` にあるメソッドの抜粋です．

- **max** (self, other: f64) -> f64
- **min** (self, other: f64) -> f64
- **asin** (self) -> f64
- **acos** (self) -> f64
- **atan** (self) -> f64
- **atan2** (self, other: f64) -> f64
- **sin_cos** (self) -> (f64, f64)
- **to_radians** (self) -> f64
- **to_degrees** (self) -> f64
- **sin** (self) -> f64
- **cos** (self) -> f64
- **tan** (self) -> f64
- **is_nan** (self) -> bool
- **is_infinite** (self) -> bool
- **is_finite** (self) -> bool
- **is_sign_positive** (self) -> bool
- **is_sign_negative** (self) -> bool
- **abs** (self) -> f64
- **recip** (self) -> f64
- **floor** (self) -> f64
- **ceil** (self) -> f64
- **round** (self) -> f64
- **trunc** (self) -> f64
- **fract** (self) -> f64
- **powi** (self, n: i32) -> f64
- **powf** (self, n: f64) -> f64
- **sqrt** (self) -> f64
- **exp** (self) -> f64
- **exp2** (self) -> f64
- **ln** (self) -> f64
- **log** (self, base: f64) -> f64
- **log2** (self) -> f64
- **log10** (self) -> f64

また，定数も定義されています．

- EPSILON
- NAN
- INFINITY
- MIN
- MIN_POSITIVE
- MAX

また，この他の数学定数も `std::f64::consts` に定義されています．

- E
- LN_2
- LN_10
- LOG2_10
- LOG2_E
- LOG10_2
- LOG10_E
- TAU
- FRAC_1_PI
- FRAC_1_SQRT_PI
- FRAC_2_PI
- FRAC_2_SQRT_PI
- FRAC_PI_[2,3,4,6,8]
- PI
- SQRT_2

よく使う `PI` と `FRAC_1_PI` を再公開し，独自に `PI2` と `EPS` を定義します．

```
// src/rayt/mod.rs
pub use std::f64::consts::PI;
pub use std::f64::consts::FRAC_1_PI;
pub const PI2: f64 = PI*2.0;
pub const EPS: f64 = 1e-6;
```

既に実装したコンストラクタ以外に，イテレータから生成できると便利です．そこで，`FromIterator` トレイトを実装します．

```
// src/rayt/float3.rs
impl FromIterator<f64> for Float3 {
    fn from_iter<I: IntoIterator<Item = f64>>(iter: I) -> Self {
        let mut initer = iter.into_iter();
        Float3([
            initer.next().unwrap(),
            initer.next().unwrap(),
            initer.next().unwrap(),
        ])
    }
}
```

算術系のメソッドもいくつか実装します．

```
// src/rayt/float3.rs
pub fn sqrt(&self) -> Self {
    Self::from_iter(self.0.iter().map(|x| x.sqrt()))
}

pub fn near_zero(&self) -> bool {
    self.0.iter().all(|x| x.abs() < EPS)
}

pub fn saturate(&self) -> Self {
    Self::from_iter(self.0.iter().map(|x| x.min(1.0).max(0.0)))
}
```

この `Float3` タプル構造体はフィールドを公開していません．そのため，内部の要素にアクセスする便利な機能として，配列を返すメソッドとイテレータを返すメソッドを用意します．

```
// src/rayt/float3.rs
pub fn to_array(&self) -> [f64; 3] {
    self.0
}

pub fn iter(&self) -> std::slice::Iter<'_, f64> {
    self.0.iter()
}

pub fn iter_mut(&mut self) -> std::slice::IterMut<'_, f64> {
    self.0.iter_mut()
}
```

ちなみに，タプル構造体でフィールドに `pub` をつければ，外部から `.0` で操作できます．

```
pub struct Float3(pub [f64; 3]);
```

これで基本的な算術メソッドは実装したので，ベクトル演算メソッドを実装します．例えば，内積や外積，長さなどを計算します．

```
// src/rayt/float3.rs
pub fn dot(&self, rhs: Self) -> f64 {
    self.0.iter().zip(rhs.0.iter()).fold(0.0, |acc, (l,r)| acc + l*r)
}

pub fn cross(&self, rhs: Self) -> Self {
    Self([
        self.0[1] * rhs.0[2] - self.0[2] * rhs.0[1],
        self.0[2] * rhs.0[0] - self.0[0] * rhs.0[2],
        self.0[0] * rhs.0[1] - self.0[1] * rhs.0[0],
    ])
}

pub fn length(&self) -> f64 {
    self.length_squared().sqrt()
}

pub fn length_squared(&self) -> f64 {
    self.0.iter().fold(0.0, |acc, x| acc + x * x)
}

pub fn normalize(&self) -> Self {
    *self / self.length()
}

pub fn lerp(&self, v: Self, t: f64) -> Self {
    *self + (v - *self) * t
}
```

以下は，各メソッドの簡単な説明です．

関数名	説明
dot	内積を求めます
cross	外積を求めます
length	ベクトルの長さを求めます
length_squared	ベクトルの長さの二乗を求めます
normalize	正規化したベクトルを返します
lerp	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

この他にベクトル型として便利なメソッドをいくつか実装します．

```
// src/rayt/float3.rs
pub fn x(&self) -> f64 { self.0[0] }
pub fn y(&self) -> f64 { self.0[1] }
pub fn z(&self) -> f64 { self.0[2] }
pub const fn xaxis() -> Self { Self::new(1.0, 0.0, 0.0) }
```

```
pub const fn yaxis() -> Self { Self::new(0.0, 1.0, 0.0) }
pub const fn zaxis() -> Self { Self::new(0.0, 0.0, 1.0) }
```

今度はカラー演算のメソッドを実装していきます．16 進数の文字列から生成したり，RGB の画素値に変換したりします．

```
// src/rayt/float3.rs
pub fn from_hex(hex: &[u8; 6]) -> Self {
    if let Ok(hex_str) = std::str::from_utf8(hex) {
        let r = u8::from_str_radix(&hex_str[0..2], 16).unwrap();
        let g = u8::from_str_radix(&hex_str[2..4], 16).unwrap();
        let b = u8::from_str_radix(&hex_str[4..6], 16).unwrap();
        Self::from_rgb(r, g, b)
    } else {
        panic!();
    }
}

pub fn from_rgb(r: u8, g: u8, b: u8) -> Self {
    Self::new(r as f64 / 255.0, g as f64 / 255.0, b as f64 / 255.0)
}

pub fn to_rgb(&self) -> [u8; 3] {
    [self.r(), self.g(), self.b()]
}

pub fn r(&self) -> u8 { (255.99 * self.0[0].min(1.0).max(0.0)) as u8 }
pub fn g(&self) -> u8 { (255.99 * self.0[1].min(1.0).max(0.0)) as u8 }
pub fn b(&self) -> u8 { (255.99 * self.0[2].min(1.0).max(0.0)) as u8 }
```

レイトレーシングではランダムに生成されたベクトルや乱数を多用します．これも `Float3` に機能を集約させます．乱数を扱うために，`rand` パッケージが必要です．`cargo.toml` の `dependencies` セクションに依存するパッケージを指定します．

```
[dependencies]
rand = "0.8.3"
```

`cargo.toml` に追記したら，ビルドすれば必要なパッケージを自動でダウンロードしてくれます．この外部パッケージの管理に便利なバイナリパッケージ `cargo-edit` があります．インストールするには以下を実行します．

```
cargo install cargo-edit
```

これで，`cargo` に `add`，`list`，`rm` サブコマンドが追加されます．`add` を使えばパッケージを簡単に追加することが出来ます．

```
cargo add rand
```

基本的にパッケージはバージョンを指定しますが，`cargo-edit` を使えば，最新のバージョンを自動で指定してくれます．これを実行すると，`cargo.toml` ファイルが更新されます．`rand` パッケージを使うには，`rand::prelude` をインポートします．

```
// src/rayt/float3.rs
use rand::prelude::*;
```

これで乱数が使えるようになるので，いくつか便利なメソッドを実装します．

```
// src/rayt/float3.rs
pub fn random() -> Self {
    Self::new(random::<f64>(), random::<f64>(), random::<f64>())
}

pub fn random_full() -> Self {
    Self::full(random::<f64>())
}

pub fn random_limit(min: f64, max: f64) -> Self {
    Self::from_iter(Self::random().0.iter().map(|x| min + x * (max - min)))
}
```

次に演算子のオーバーロードをします．`std::ops` で定義されている演算子をいくつか実装します．

```
impl std::ops::Neg for Float3 {
    type Output = Self;
    fn neg(self) -> Self {
        Self::from_iter(self.0.iter().map(|x| -x))
    }
}

impl std::ops::AddAssign<Float3> for Float3 {
    fn add_assign(&mut self, rhs: Self) {
        for i in 0..3 { self.0[i] += rhs.0[i] }
    }
}

impl std::ops::Add<Float3> for Float3 {
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Self::from_iter(self.0.iter().zip(rhs.0.iter()).map(|(l, r)| l + r))
    }
}

impl std::ops::SubAssign<Float3> for Float3 {
    fn sub_assign(&mut self, rhs: Self) {
        for i in 0..3 { self.0[i] -= rhs.0[i] }
    }
}

impl std::ops::Sub<Float3> for Float3 {
```

```
    type Output = Self;
    fn sub(self, rhs: Self) -> Self {
        Self::from_iter(self.0.iter().zip(rhs.0.iter()).map(|(l, r)| l - r))
    }
}

impl std::ops::Mul<f64> for Float3 {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self {
        Self::from_iter(self.0.iter().map(|x| x * rhs))
    }
}

impl std::ops::Mul<Float3> for f64 {
    type Output = Float3;
    fn mul(self, rhs: Float3) -> Float3 {
        Float3::from_iter(rhs.0.iter().map(|x| x * self))
    }
}

impl std::ops::MulAssign<f64> for Float3 {
    fn mul_assign(&mut self, rhs: f64) {
        for i in 0..3 { self.0[i] *= rhs }
    }
}

impl std::ops::Mul<Float3> for Float3 {
    type Output = Float3;
    fn mul(self, rhs: Float3) -> Float3 {
        Float3::from_iter(self.0.iter().zip(rhs.0.iter()).map(|(l, r)| l * r))
    }
}

impl std::ops::DivAssign<f64> for Float3 {
    fn div_assign(&mut self, rhs: f64) {
        for i in 0..3 { self.0[i] /= rhs }
    }
}

impl std::ops::Div<f64> for Float3 {
    type Output = Self;
    fn div(self, rhs: f64) -> Self {
        Float3::from_iter(self.0.iter().map(|x| x / rhs))
    }
}
```

5．クォータニオンモジュール

ベクトルモジュール同様に，クォータニオンモジュールも作成します．`src/rayt/quat.rs` ファイルを作成して実装します．クォータニオンもベクトルと同様にタプル構造体にします．

```rust
// src/rayt/quat.rs
pub struct Quat(Vec3, f64);
```

まずはコンストラクタを実装します.

```rust
// src/rayt/quat.rs
pub const fn new(x: f64, y: f64, z: f64, w: f64) -> Self {
    Quat(Vec3::new(x, y, z), w)
}

pub fn from_rot(v: Vec3, rad: f64) -> Self {
    let (s, c) = (rad * 0.5).sin_cos();
    Quat(v * s, c)
}

pub fn from_rot_x(rad: f64) -> Self {
    let (s, c) = (rad * 0.5).sin_cos();
    Quat::new(s, 0.0, 0.0, c)
}

pub fn from_rot_y(rad: f64) -> Self {
    let (s, c) = (rad * 0.5).sin_cos();
    Quat::new(0.0, s, 0.0, c)
}

pub fn from_rot_z(rad: f64) -> Self {
    let (s, c) = (rad * 0.5).sin_cos();
    Quat::new(0.0, 0.0, s, c)
}

pub const fn unit() -> Self {
    Quat::new(0.0, 0.0, 0.0, 1.0)
}

pub const fn zero() -> Self {
    Quat::new(0.0, 0.0, 0.0, 0.0)
}
```

次は演算メソッドです.

```rust
// src/rayt/quat.rs
pub fn conj(&self) -> Self {
    Quat(-self.0, self.1)
}

pub fn dot(&self, rhs: Self) -> f64 {
    self.0.dot(rhs.0) + self.1 * rhs.1
}

pub fn length(&self) -> f64 {
    self.length_squared().sqrt()
```

```
}

pub fn length_squared(&self) -> f64 {
    self.0.length_squared() + self.1.powi(2)
}

pub fn normalize(&self) -> Self {
    let recip = self.length().recip();
    Quat(self.0 * recip, self.1 * recip)
}
```

配列に変換するメソッドを定義します.

```
// src/rayt/quat.rs
pub fn to_array(&self) -> [f64; 4] {
    let [x, y, z] = self.0.to_array();
    [x, y, z, self.1]
}
```

ベクトルを回転させるメソッドを実装します.

```
// src/rayt/quat.rs
pub fn rotate(&self, p: Vec3) -> Vec3 {
    let [x1, y1, z1, w1] = self.to_array();
    let [x2, y2, z2] = p.to_array();
    let x = (w1 * x2 + y1 * z2) - (z1 * y2);
    let y = (w1 * y2 + z1 * x2) - (x1 * z2);
    let z = (w1 * z2 + x1 * y2) - (y1 * x2);
    let w = (x1 * x2 + y1 * y2) - (z1 * z2);
    Vec3::new(
        ((w * x1 + x * w1) - y * z1) + z * y1,
        ((w * y1 + y * w1) - z * x1) + x * z1,
        ((w * z1 + z * w1) - x * y1) + y * x1,
    )
}
```

また，演算子もオーバーロードします.

```
// src/rayt/quat.rs
impl std::ops::Mul<Quat> for Quat {
    type Output = Self;
    fn mul(self, rhs: Quat) -> Self {
        let [x1, y1, z1, w1] = self.to_array();
        let [x2, y2, z2, w2] = rhs.to_array();
        Quat::new(
            w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2,
            w1 * y2 + y2 * w2 + z1 * x2 - x1 * z2,
            w1 * z2 + z1 * w2 + x1 * y2 - y1 * x2,
            w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2,
        )
    }
```

```
}
```

これでクォータニオンモジュールは完成です．最後に忘れずに，`src/rayt/mod.rs` ファイルを編集します．

```
// src/rayt/mod.rs
mod quat;
use self::quat::Quat;
```

6．基本クラス

6.1 光線

光線は始点と向きを持っています．光線は `ray` モジュールとして作成します．コードは次の通りです．

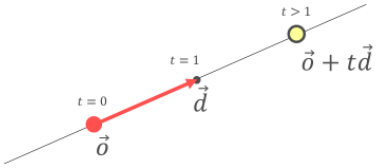
```
// src/rayt/ray.rs
use crate::rayt::*;

#[derive(Debug, Clone, Copy)]
pub struct Ray {
    pub origin: Point3,
    pub direction: Vec3,
}

impl Ray {
    pub fn new(origin: Point3, direction: Vec3) -> Self {
        Self { origin, direction }
    }

    pub fn at(&self, t: f64) -> Point3 {
        self.origin + t * self.direction
    }
}
```

忘れずに `src/rayt/mod.rs` に追加します．

```
// src/rayt/mod.rs
mod ray;
use self::ray::Ray;
```

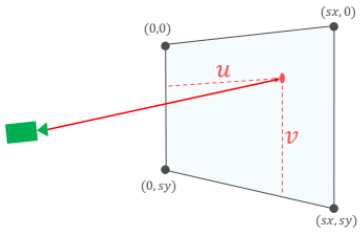
`at` 関数は，始点 $\vec{o}$ から方向 $\vec{d}$ に向かう直線上の任意の点 $\vec{p}$ を，パラメータ t を指定して求めます．$t=0$ のときは始点，$t=1$ のときは始点から方向ベクトル $\vec{d}$ 進んだ位置，$t>1$ なら方向ベクトルより先の位置，$t<0$ なら，始点から反対方向に向かう直線上の位置を取得できます．式で表すと

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



6.2 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，

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

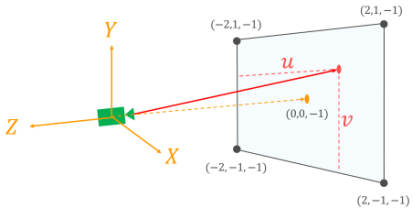
基底ベクトル $\vec{u}, \vec{v}, \vec{w}$ を使ってこれを基底変換すると，投影するスクリーン上の位置 $\vec{p}$ は

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

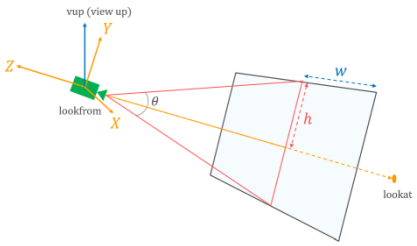
カメラの位置を原点に置き，-z 方向を向いているとします．スクリーンの右上が $(2, 1, -1)$ となるような基底ベクトルは次のようになります．

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

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

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

`ray` 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．これらをカメラモジュール（`src/rayt/camera.rs`）にまとめます．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．

```
// src/rayt/camera.rs
use crate::rayt::*;

#[derive(Debug)]
pub struct Camera {
    pub origin: Point3,
    pub u: Vec3,
    pub v: Vec3,
    pub w: Vec3,
}

impl Camera {
    pub fn new(u: Vec3, v: Vec3, w: Vec3) -> Self {
        Self { origin: Point3::zero(), u, v, w }
    }
```

```
    pub fn from_lookat(origin: Vec3, lookat: Vec3, vup: Vec3, vfov: f64, aspect: f64) -> Self {
        let halfh = (vfov.to_radians() * 0.5).tan();
        let halfw = aspect * halfh;
        let w = (origin - lookat).normalize();
        let u = vup.cross(w).normalize();
        let v = w.cross(u);
        let uw = halfw * u;
        let vh = halfh * v;
        Self {
            origin,
            u: 2.0 * uw,
            v: 2.0 * vh,
            w: origin - uw - vh - w,
        }
    }

    pub fn ray(&self, u: f64, v: f64) -> Ray {
        Ray {
            origin: self.origin,
            direction: self.w + self.u * u + self.v * v - self.origin,
        }
    }
}
```

`src/rayt/mod.rs` に追加します．

```
// src/rayt/mod.rs
mod camera;
use self::camera::Camera;
```

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．

```
fn color(ray: &Ray) -> Color {
    let d = ray.direction.normalize();
    let t = 0.5 * (d.y() + 1.0);
    Color::new(0.5, 0.7, 1.0).lerp(Color::one(), t)
}
```

以前作成した画像生成のコードは PPM 形式でした．もっと一般的な画像形式で保存するために，`image` パッケージを利用します．

```
cargo add image
```

次のコードは `RgbImage` を作って画素値を設定し，PNG 形式で保存します．

```
use crate::rayt::*;
use image::{Rgb, RgbImage};
```

```rust
const IMAGE_WIDTH: u32 = 200;
const IMAGE_HEIGHT: u32 = 100;

fn color(ray: &Ray) -> Color {
    let d = ray.direction.normalize();
    let t = 0.5 * (d.y() + 1.0);
    Color::new(0.5, 0.7, 1.0).lerp(Color::one(), t)
}

fn main() {
    let camera = Camera::new(
        Vec3::new(4.0, 0.0, 0.0),
        Vec3::new(0.0, 2.0, 0.0),
        Vec3::new(-2.0, -1.0, -1.0),
    );
    let mut img = RgbImage::new(IMAGE_WIDTH, IMAGE_HEIGHT);
    img.enumerate_pixels_mut()
        .collect::<Vec<(u32, u32, &mut Rgb<u8>)>>()
        .iter_mut()
        .for_each(|(x, y, pixel)| {
            let u = *x as f64 / (IMAGE_WIDTH - 1) as f64;
            let v = *y as f64 / (IMAGE_HEIGHT - 1) as f64;
            let ray = camera.ray(u, v);
            let rgb = color(&ray).to_rgb();
            pixel[0] = rgb[0];
            pixel[1] = rgb[1];
            pixel[2] = rgb[2];
        });
    img.save(String::from("render.png")).unwrap();
}
```

さらに並列処理を入れます．今回は並列処理パッケージ `rayon` を使います．

```
cargo add rayon
```

`rayon` を使うことで，先程のコードをとても簡単に並列化することができます．それは `iter_mut` を `par_iter_mut` にするだけです．

```rust
use crate::rayt::*;
use image::{Rgb, RgbImage};
use rayon::prelude::*;

const IMAGE_WIDTH: u32 = 200;
const IMAGE_HEIGHT: u32 = 100;

fn color(ray: &Ray) -> Color {
    let d = ray.direction.normalize();
    let t = 0.5 * (d.y() + 1.0);
    Color::new(0.5, 0.7, 1.0).lerp(Color::one(), t)
}
```

```rust
fn main() {
    let camera = Camera::new(
        Vec3::new(4.0, 0.0, 0.0),
        Vec3::new(0.0, 2.0, 0.0),
        Vec3::new(-2.0, -1.0, -1.0),
    );
    let mut img = RgbImage::new(IMAGE_WIDTH, IMAGE_HEIGHT);
    img.enumerate_pixels_mut()
        .collect::<Vec<(u32, u32, &mut Rgb<u8>)>>()
        .par_iter_mut()
        .for_each(|(x, y, pixel)| {
            let u = *x as f64 / (IMAGE_WIDTH - 1) as f64;
            let v = *y as f64 / (IMAGE_HEIGHT - 1) as f64;
            let ray = camera.ray(u, v);
            let rgb = color(&ray).to_rgb();
            pixel[0] = rgb[0];
            pixel[1] = rgb[1];
            pixel[2] = rgb[2];
        });
    img.save(String::from("render.png")).unwrap();
}
```

実行すると次のような画像になります．

Fig.2.2: code102

7．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

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

展開すると

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

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

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

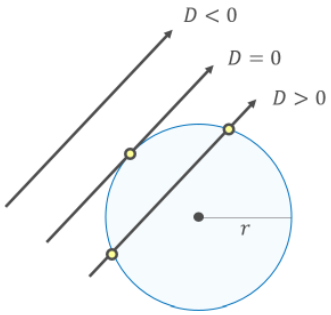
oc を展開して，$ax^2 + bx + c = 0$ に当てはめてみると

$$a = (\vec{d} \cdot \vec{d})$$

$$b = 2(\vec{d} \cdot (\vec{o} - \vec{c}))$$

$$c = ((\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c})) - r^2,$$

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．

```rust
fn hit_sphere(center: Point3, radius: f64, ray: &Ray) -> bool {
    let oc = ray.origin - center;
    let a = ray.direction.dot(ray.direction);
    let b = 2.0 * ray.direction.dot(oc);
    let c = oc.dot(oc) - radius.powi(2);
    let d = b * b - 4.0 * a * c;
    d > 0.0
}

fn color(ray: Ray) -> Color {
    if hit_sphere(Point3::new(0.0, 0.0, -1.0), 0.5, &ray) {
        return Color::new(1.0, 0.0, 0.0);
    }
    let d = ray.direction.normalize();
    let t = 0.5 * (d.y() + 1.0);
    Color::new(0.5, 0.7, 1.0).lerp(Color::one(), t)
}
```

これは次のような画像になります.

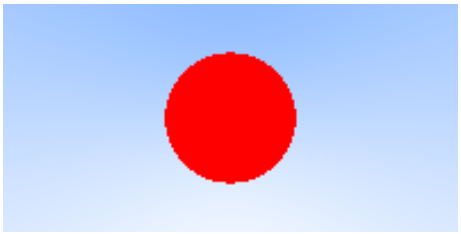
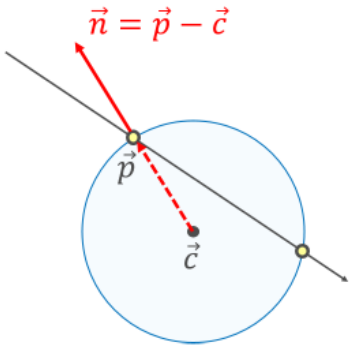


Fig.2.3: code103

7.1 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.

```
fn hit_sphere(center: Point3, radius: f64, ray: &Ray) -> f64 {
    let oc = ray.origin - center;
    let a = ray.direction.dot(ray.direction);
    let b = 2.0 * ray.direction.dot(oc);
    let c = oc.dot(oc) - radius.powi(2);
    let d = b * b - 4.0 * a * c;
    if d < 0.0 {
        -1.0
    } else {
        return (-b - d.sqrt()) / (2.0 * a);
    }
}

fn color(ray: Ray) -> Color {
    let c = Point3::new(0.0, 0.0, -1.0);
    let t = hit_sphere(c, 0.5, &ray);
    if t > 0.0 {
        let n = (ray.at(t) - c).normalize();
        return 0.5 * (n + Vec3::one());
    }
    let d = ray.direction.normalize();
    let t = 0.5 * (d.y() + 1.0);
    Color::one().lerp(Color::new(0.5, 0.7, 1.0), t)
}
```

この計算は前回の画像から縦軸を反転しているので `main` 関数は次のようになります.

```
fn main() {
    let camera = Camera::new(
        Vec3::new(4.0, 0.0, 0.0),
        Vec3::new(0.0, 2.0, 0.0),
        Vec3::new(-2.0, -1.0, -1.0),
    );
    let mut img = RgbImage::new(IMAGE_WIDTH, IMAGE_HEIGHT);
    img.enumerate_pixels_mut()
        .collect::<Vec<(u32, u32, &mut Rgb<u8>)>>()
        .par_iter_mut()
        .for_each(|(x, y, pixel)| {
            let u = *x as f64 / (IMAGE_WIDTH - 1) as f64;
            let v = (IMAGE_HEIGHT - *y - 1) as f64 / (IMAGE_HEIGHT - 1) as f64;
            let ray = camera.ray(u, v);
            let rgb = color(ray).to_rgb();
            pixel[0] = rgb[0];
            pixel[1] = rgb[1];
            pixel[2] = rgb[2];
        });
    img.save(String::from("render.png")).unwrap();
}
```

これは次のような画像になります.

Fig.2.4: code104

8. 確認用ウィンドウモジュール

レンダリングした画像をすぐに確認できる仕組みを作ります．これには `minifb` パッケージを使用します．

```
cargo add minifb
```

まずは，レンダリングした画像を表示するウィンドウを作成する機能を作ります．それを `src/rayt/window.rs` モジュールにまとめます．

```rust
// src/rayt/window.rs
use minifb::{Key, KeyRepeat, Window, WindowOptions};
use image::RgbImage;

pub fn draw_in_window(backup_filename: &str, pixels: RgbImage) -> minifb::Result<()> {
    if cfg!(test) { return Ok(()) }
    let (image_width, image_height) = pixels.dimensions();
    let mut buffer: Vec<u32> = vec![0; (image_width * image_height) as usize];
    let mut window = Window::new(
        "ESC to exit",
        image_width as usize,
        image_height as usize,
        WindowOptions {
            topmost: true,
            ..WindowOptions::default()
        },
    )
    .unwrap_or_else(|e| panic!("{}", e));

    // Limit to max ~30 fps update here
    window.limit_update_rate(Some(std::time::Duration::from_micros(16600 * 2)));

    for (i, (_, _, pixel)) in buffer.iter_mut().zip(pixels.enumerate_pixels()) {
        *i = u32::from_be_bytes([0, pixel[0], pixel[1], pixel[2]]);
    }

    window.update_with_buffer(&buffer, image_width as usize, image_height as usize)?;

    let mut backup_buffer: Option<Vec<u32>> = None;
    let mut show_backup = false;

    while window.is_open() && !window.is_key_down(Key::Escape) {
```

```
        if window.is_key_pressed(Key::D, KeyRepeat::No) {
            if backup_buffer.is_none() {
                if let Ok(img) = image::open(backup_filename) {
                    let backup_image = img.to_rgb8();
                    let (w, h) = backup_image.dimensions();
                    let mut buf = vec![0; (w * h) as usize];
                    for (i, (_, _, pixel)) in buf.iter_mut().zip(backup_image.enumerate_pixels())
                        {
                        *i = u32::from_be_bytes([0, pixel[0], pixel[1], pixel[2]]);
                    }

                    backup_buffer = Some(buf);
                }
            }

            show_backup = !show_backup;
        }

        let mut current_buffer = &buffer;
        if show_backup {
            if let Some(ref x) = backup_buffer {
                current_buffer = x;
            }
        }
        window.update_with_buffer(current_buffer, image_width as usize, image_height as usize)?;
    }

    Ok(())
}
```

`src/rayt/mod.rs` にも追加しておきます．

```
// src/rayt/mod.rs
mod window;
use self::window::*;
```

`minifb` の使い方はここでは詳しく説明しません． `draw_in_window` 関数でやっていることは，引数として渡された `RgbImage` を描画するウィンドウを作成して表示します．追加の機能として，指定のファイルを読み込んで，キーボードの `D` キーを押すと切り替える仕組みが入っています．次に前回のレンダリングした画像があれば，別名にしてバックアップを取る関数を作ります．

```
use std::{fs, path::Path};

const OUTPUT_FILENAME: &str = "render.png";
const BACKUP_FILENAME: &str = "render_bak.png";

fn backup() {
    let output_path = Path::new(OUTPUT_FILENAME);
    if output_path.exists() {
        println!("backup {:?} -> {:?}", OUTPUT_FILENAME, BACKUP_FILENAME);
        // replacing the original file if to already exists
```

```
        fs::rename(OUTPUT_FILENAME, BACKUP_FILENAME).unwrap();
    }
}
```

プログラム実行時にバックアップを取り，レンダリングした画像を確認するには次のようになります．

```
fn main() {
    backup();

    let camera = Camera::new(
        Vec3::new(4.0, 0.0, 0.0),
        Vec3::new(0.0, 2.0, 0.0),
        Vec3::new(-2.0, -1.0, -1.0),
    );
    let mut img = RgbImage::new(IMAGE_WIDTH, IMAGE_HEIGHT);
    img.enumerate_pixels_mut()
        .collect::<Vec<(u32, u32, &mut Rgb<u8>)>>()
        .par_iter_mut()
        .for_each(|(x, y, pixel)| {
            let u = *x as f64 / (IMAGE_WIDTH - 1) as f64;
            let v = (IMAGE_HEIGHT - *y - 1) as f64 / (IMAGE_HEIGHT - 1) as f64;
            let ray = camera.ray(u, v);
            let rgb = color(ray).to_rgb();
            pixel[0] = rgb[0];
            pixel[1] = rgb[1];
            pixel[2] = rgb[2];
        });
    img.save(OUTPUT_FILENAME).unwrap();
    draw_in_window(BACKUP_FILENAME, img).unwrap();
}
```

前回とレンダリング画像は変わりません．

Fig.2.5: code105

9．レンダーモジュール

ここで，少しコードを整理するためにレンダーモジュール（ `src/rayt/render.rs` ）を追加します．シーントレイトはカメラや画像出力，球などの物体を管理します．共通メソッドにいくつか必要なメソッドを定義しておきます．これまで作成した `hit_sphere` や `color` 関数，カメラのオブジェクトをシーンインスタンスのオブジェクトにまとめます．また，背景色を返す関数として `background` 関数を追加し， `color` 関数を `trace` 関数と

名前を変えて，そこから呼び出しています． `backup` 関数もレンダーモジュールに移動します． `render` 関数には `Scene` トレイトのインスタンスであるオブジェクトを渡します．

```
// src/rayt/render.rs
use crate::rayt::*;
use image::{Rgb, RgbImage};
use rayon::prelude::*;

const IMAGE_WIDTH: u32 = 200;
const IMAGE_HEIGHT: u32 = 100;
const OUTPUT_FILENAME: &str = "render.png";
const BACKUP_FILENAME: &str = "render_bak.png";

fn backup() {
    let output_path = Path::new(OUTPUT_FILENAME);
    if output_path.exists() {
        println!("backup {:?} -> {:?}", OUTPUT_FILENAME, BACKUP_FILENAME);
        // replacing the original file if to already exists
        fs::rename(OUTPUT_FILENAME, BACKUP_FILENAME).unwrap();
    }
}

pub trait Scene {
    fn camera(&self) -> Camera;
    fn trace(&self, ray: Ray) -> Color;
    fn width(&self) -> u32 { IMAGE_WIDTH }
    fn height(&self) -> u32 { IMAGE_HEIGHT }
    fn aspect(&self) -> f64 { self.width() as f64 / self.height() as f64 }
}

pub fn render(scene: impl Scene + Sync) {
    backup();

    let camera = scene.camera();
    let mut img = RgbImage::new(scene.width(), scene.height());
    img.enumerate_pixels_mut()
        .collect::<Vec<(u32, u32, &mut Rgb<u8>)>>()
        .par_iter_mut()
        .for_each(|(x, y, pixel)| {
            let u = *x as f64 / (scene.width() - 1) as f64;
            let v = (scene.height() - *y - 1) as f64 / (scene.height() - 1) as f64;
            let ray = camera.ray(u, v);
            let rgb = scene.trace(ray).to_rgb();
            pixel[0] = rgb[0];
            pixel[1] = rgb[1];
            pixel[2] = rgb[2];
        });
    img.save(OUTPUT_FILENAME).unwrap();
    draw_in_window(BACKUP_FILENAME, img).unwrap();
}
```

ここで重要なのは `render` 関数の引数で指定しているトレイト境界 `impl Scene + Sync` の部分です．

`Sync` マーカートレイトを指定しているのは，シーンインスタンスのオブジェクトが複数のスレッドから参照されるためです．レンダーモジュールも `src/rayt/mod.rs` に追加しておきます．

```
// src/rayt/mod.rs
mod render;
use self::render::*;
```

次に，`SimpleScene` というシーンのインスタンスに処理をまとめて実行します．

```
use crate::rayt::*;

struct SimpleScene {}

impl SimpleScene {
    fn hit_sphere(&self, center: Point3, radius: f64, ray: &Ray) -> f64 {
        let oc = ray.origin - center;
        let a = ray.direction.dot(ray.direction);
        let b = 2.0 * ray.direction.dot(oc);
        let c = oc.dot(oc) - radius.powi(2);
        let d = b * b - 4.0 * a * c;
        if d < 0.0 {
            -1.0
        } else {
            return (-b - d.sqrt()) / (2.0 * a);
        }
    }

    fn background(&self, d: Vec3) -> Color {
        let t = 0.5 * (d.normalize().y() + 1.0);
        Color::one().lerp(Color::new(0.5, 0.7, 1.0), t)
    }
}

impl Scene for SimpleScene {
    fn camera(&self) -> Camera {
        Camera::new(
            Vec3::new(4.0, 0.0, 0.0),
            Vec3::new(0.0, 2.0, 0.0),
            Vec3::new(-2.0, -1.0, -1.0),
        )
    }

    fn trace(&self, ray: Ray) -> Color {
        let c = Point3::new(0.0, 0.0, -1.0);
        let t = self.hit_sphere(c, 0.5, &ray);
        if t > 0.0 {
            let n = (ray.at(t) - c).normalize();
            return 0.5 * (n + Vec3::one());
        }
        self.background(ray.direction)
    }
}
```

```
fn main() {
    render(SimpleScene {});
}
```

前回とレンダリング画像は変わりません.

Fig.2.6: code106

10. 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.

10.1 衝突情報

まず，物体に衝突したときの情報を格納する構造体 `HitInfo` を用意します.

```
struct HitInfo {
    t: f64,
    p: Point3,
    n: Vec3,
}

impl HitInfo {
    const fn new(t: f64, p: Point3, n: Vec3) -> Self {
        Self { t, p, 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.

10.2 物体トレイト

次に，物体トレイト `Shape` を追加します．これは衝突関数が共通の振る舞いとして定義されています.

```
trait Shape: Sync {
    fn hit(&self, ray: &Ray, to: f64, t1: f64) -> Option<HitInfo>;
}

struct Sphere {
    center: Point3,
    radius: f64,
```

```
}
```

ここで，`Sync` トレイトを継承しています．トレイト境界を使って，`Shape + Sync` とすることも出来ますが，トレイト継承の方が確実です．また，`hit` メソッドは `Option<HitInfo>` を返します．`t0` と `t1` は光線の衝突範囲です．

10.3 球

そして，`Shape` のインスタンスである球 `Sphere` を追加します．

```
struct Sphere {
    center: Point3,
    radius: f64,
}

impl Sphere {
    const fn new(center: Point3, radius: f64) -> Self {
        Self { center, radius }
    }
}
```

`Shape` トレイトメソッドを実装します．

```
impl Shape for Sphere {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let oc = ray.origin - self.center;
        let a = ray.direction.dot(ray.direction);
        let b = 2.0 * ray.direction.dot(oc);
        let c = oc.dot(oc) - self.radius.powi(2);
        let d = b * b - 4.0 * a * c;
        if d > 0.0 {
            let root = d.sqrt();
            let temp = (-b - root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                return Some(HitInfo::new(temp, p, (p - self.center) / self.radius));
            }
            let temp = (-b + root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                return Some(HitInfo::new(temp, p, (p - self.center) / self.radius));
            }
        }

        None
    }
}
```

`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．

10.4 物体リスト

複数の物体を管理する型を追加します．これも `Shape` トレイトのインスタンスにすることで，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．

```
struct ShapeList {
    pub objects: Vec<Box<dyn Shape>>,
}

impl ShapeList {
    pub fn new() -> Self {
        Self { objects: Vec::new() }
    }

    pub fn push(&mut self, object: Box<dyn Shape>) {
        self.objects.push(object);
    }
}
```

`Box` は束縛する `T` 型に `Sync` マーカートレイトがついていれば， `Box` そのものも `Sync` になります．`Shape` トレイトは `Sync` マーカートレイトを継承しているので，そのインスタンスも `Sync` であることが強制されます． `ShapeList` でも `Shape` のトレイトメソッドを実装します．

```
impl Shape for ShapeList {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let mut hit_info: Option<HitInfo> = None;
        let mut closest_so_far = t1;
        for object in &self.objects {
            if let Some(info) = object.hit(ray, t0, closest_so_far) {
                closest_so_far = info.t;
                hit_info = Some(info);
            }
        }

        hit_info
    }
}
```

それでは球を２つ表示してみます．そのためには `SimpleScene` に物体リスト型のオブジェクトを追加します．

```
struct SimpleScene {
    world: ShapeList,
}
```

次にコンストラクタで物体を生成します．

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(Box::new(Sphere::new(Point3::new(0.0, 0.0, -1.0), 0.5)));
```

```
    world.push(Box::new(Sphere::new(Point3::new(0.0, -100.5, -1.0), 100.0)));
    Self { world }
}
```

そして，`trace` 関数で `Shape::hit` 関数を呼ぶようにします．

```
fn trace(&self, ray: Ray) -> Color {
    let hit_info = self.world.hit(&ray, 0.0, f64::MAX);
    if let Some(hit) = hit_info {
        0.5 * (hit.n + Vec3::one())
    } else {
        self.background(ray.direction)
    }
}
```

最終的にコードは次のようになります．

```
use crate::rayt::*;

struct HitInfo {
    t: f64,
    p: Point3,
    n: Vec3,
}

impl HitInfo {
    const fn new(t: f64, p: Point3, n: Vec3) -> Self {
        Self { t, p, n }
    }
}

trait Shape: Sync {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo>;
}

struct Sphere {
    center: Point3,
    radius: f64,
}

impl Sphere {
    const fn new(center: Point3, radius: f64) -> Self {
        Self { center, radius }
    }
}

impl Shape for Sphere {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let oc = ray.origin - self.center;
        let a = ray.direction.dot(ray.direction);
        let b = 2.0 * ray.direction.dot(oc);
        let c = oc.dot(oc) - self.radius.powi(2);
```

```
        let d = b * b - 4.0 * a * c;
        if d > 0.0 {
            let root = d.sqrt();
            let temp = (-b - root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                return Some(HitInfo::new(temp, p, (p - self.center) / self.radius));
            }
            let temp = (-b + root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                return Some(HitInfo::new(temp, p, (p - self.center) / self.radius));
            }
        }

        None
    }
}

struct ShapeList {
    pub objects: Vec<Box<dyn Shape>>,
}

impl ShapeList {
    pub fn new() -> Self {
        Self { objects: Vec::new() }
    }

    pub fn push(&mut self, object: Box<dyn Shape>) {
        self.objects.push(object);
    }
}

impl Shape for ShapeList {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let mut hit_info: Option<HitInfo> = None;
        let mut closest_so_far = t1;
        for object in &self.objects {
            if let Some(info) = object.hit(ray, t0, closest_so_far) {
                closest_so_far = info.t;
                hit_info = Some(info);
            }
        }

        hit_info
    }
}

struct SimpleScene {
    world: ShapeList,
}

impl SimpleScene {
    fn new() -> Self {
```

```
        let mut world = ShapeList::new();
        world.push(Box::new(Sphere::new(Point3::new(0.0, 0.0, -1.0), 0.5)));
        world.push(Box::new(Sphere::new(Point3::new(0.0, -100.5, -1.0), 100.0)));
        Self { world }
    }

    fn background(&self, d: Vec3) -> Color {
        let t = 0.5 * (d.normalize().y() + 1.0);
        Color::one().lerp(Color::new(0.5, 0.7, 1.0), t)
    }
}

impl Scene for SimpleScene {
    fn camera(&self) -> Camera {
        Camera::new(
            Vec3::new(4.0, 0.0, 0.0),
            Vec3::new(0.0, 2.0, 0.0),
            Vec3::new(-2.0, -1.0, -1.0),
        )
    }

    fn trace(&self, ray: Ray) -> Color {
        let hit_info = self.world.hit(&ray, 0.0, f64::MAX);
        if let Some(hit) = hit_info {
            0.5 * (hit.n + Vec3::one())
        } else {
            self.background(ray.direction)
        }
    }
}

fn main() {
    render(SimpleScene::new());
}
```

実行すると次のような画像になります.

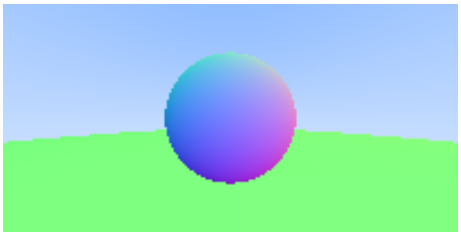
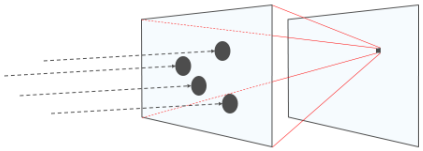


Fig.2.7: code107

11. 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．これをレンダーモジュールに実装しましょう． `render` 関数をコピーして `render_aa` 関数を作成し，光線を飛ばしているところを次のように書き換えます．

```
// src/rayt/render.rs
pub fn render_aa(scene: impl Scene + Sync) {
    backup();

    let camera = scene.camera();
    let mut img = RgbImage::new(scene.width(), scene.height());
    img.enumerate_pixels_mut()
        .collect::<Vec<(u32, u32, &mut Rgb<u8>)>>()
        .par_iter_mut()
        .for_each(|(x, y, pixel)| {
            let mut pixel_color = (0..scene.spp()).into_iter().fold(Color::zero(), |acc, _| {
                let [rx, ry, _] = Float3::random().to_array();
                let u = (*x as f64 + rx) / (scene.width() - 1) as f64;
                let v = ((scene.height() - *y - 1) as f64 + ry) / (scene.height() - 1) as f64;
                let ray = camera.ray(u, v);
                acc + scene.trace(ray)
            });
            pixel_color /= scene.spp() as f64;
            let rgb = pixel_color.to_rgb();
            pixel[0] = rgb[0];
            pixel[1] = rgb[1];
            pixel[2] = rgb[2];
        });
    img.save(OUTPUT_FILENAME).unwrap();
    draw_in_window(BACKUP_FILENAME, img).unwrap();
}
```

サンプリング数は `Scene` トレイトのメソッドで取得できるようにします．

```
// src/rayt/render.rs
const SAMPLES_PER_PIXEL: usize = 8;
pub trait Scene {
    fn camera(&self) -> Camera;
    fn trace(&self, ray: Ray) -> Color;
    fn width(&self) -> u32 { IMAGE_WIDTH }
    fn height(&self) -> u32 { IMAGE_HEIGHT }
    fn spp(&self) -> usize { SAMPLES_PER_PIXEL }
    fn aspect(&self) -> f64 { self.width() as f64 / self.height() as f64 }
}
```

`render` 関数で呼び出していたところを `render_aa` に変更します．

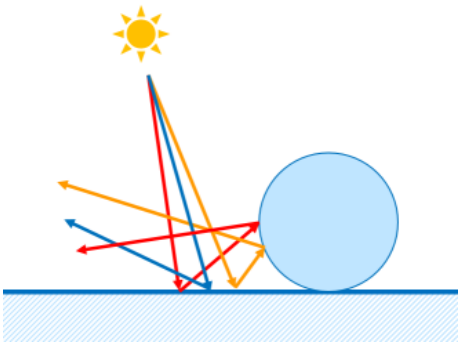
```
fn main() {
    render_aa(SimpleScene::new());
}
```

実行すると次のようになります．

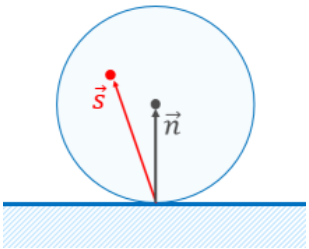
Fig.2.8: code108

12． 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 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．

```
// ray/rayt/float3.rs
pub fn random_in_unit_sphere() -> Self {
    loop {
        let point = Self::random_limit(-1.0, 1.0);
        if point.length_squared() < 1.0 {
            return point;
        }
    }
}
```

`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． `trace` 関数を次のように書き換えます．

```
fn trace(&self, ray: Ray) -> Color {
    let hit_info = self.world.hit(&ray, 0.0, f64::MAX);
    if let Some(hit) = hit_info {
        let target = hit.p + hit.n + Vec3::random_in_unit_sphere();
        0.5 * self.trace(Ray::new(hit.p, target - hit.p))
    } else {
        self.background(&ray.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．

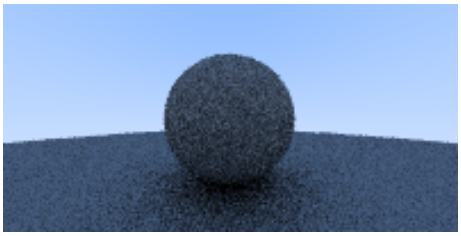
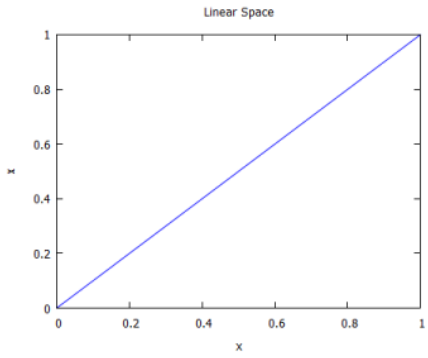
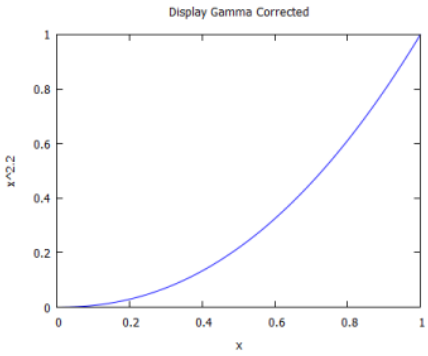


Fig.2.9: code109

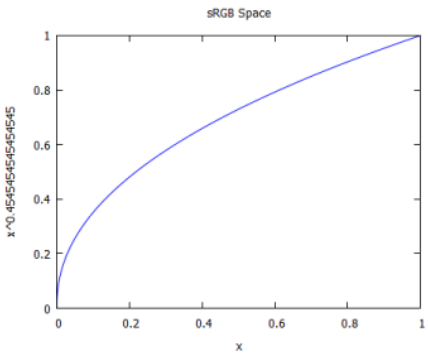
この画像暗い感じがしますね．これはディスプレイのガンマ補正の影響です．プログラム上では色をリニア空間で扱っています．例えば区間 [0,1] の値はリニア空間つまり線形なので，次の図のように直線になります．



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．

```
// ray/rayt/float3.rs
pub fn gamma(&self, factor: f64) -> Self {
    let recip = factor.recip();
    Self::from_iter(self.0.iter().map(|x| x.powf(recip)))
}

pub fn degamma(&self, factor: f64) -> Self {
    Self::from_iter(self.0.iter().map(|x| x.powf(factor)))
}
```

`gamma` がリニア空間から sRGB 空間へ，`degamma` が sRGB 空間からリニア空間に変換します．この処理をレンダーモジュールの関数に適用します．

```
// ray/rayt/render.rs
const GAMMA_FACTOR: f64 = 2.2;
pub fn render_aa(...) {
    ...
    let rgb = pixel_color.gamma(GAMMA_FACTOR).to_rgb();
    ...
}
```

この結果，次のようになります．

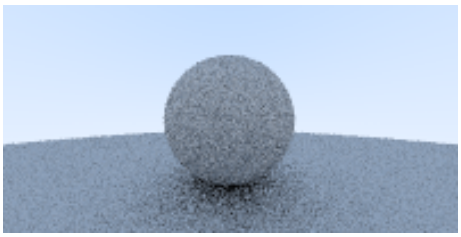


Fig.2.10: code109(gamma-collected)

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．

```
fn trace(&self, ray: Ray) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let target = hit.p + hit.n + Vec3::random_in_unit_sphere();
        0.5 * self.trace(Ray::new(hit.p, target - hit.p))
    } else {
        self.background(ray.direction)
    }
}
```

この結果は次のようになります．

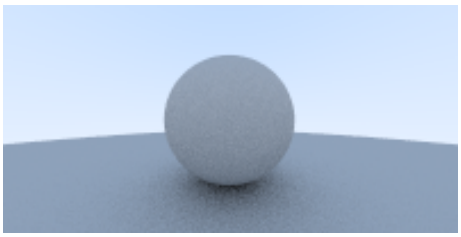


Fig.2.11: code110

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．

13．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，材質を型にしましょう．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．

まずは材質のトレイトを次のように定義します．

```
trait Material: Sync + Send {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo>;
}
```

`Material` トレイトは `Sync` と `Send` を継承しています． `Sync` は `Shape` で説明した通りです． `Send` は，所有権を移動できることを明示します． `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 `ScatterInfo` 構造体に格納します． `ScatterInfo` は次のようになっています．

```
struct ScatterInfo {
    ray: Ray,
    albedo: Color,
}

impl ScatterInfo {
    fn new(ray: Ray, albedo: Color) -> Self {
        Self { ray, 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**と呼ぶことがあります．これを材質トレイトのインスタンスとして実装します．

```
struct Lambertian {
    albedo: Color,
}

impl Lambertian {
    fn new(albedo: Color) -> Self {
        Self { albedo }
    }
}
```

`Material` トレイトメソッドを実装します.

```
impl Material for Lambertian {
    fn scatter(&self, _ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let target = hit.p + hit.n + Vec3::random_in_unit_sphere();
        Some(ScatterInfo::new(Ray::new(hit.p, target - hit.p), self.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. `HitInfo` に材質を追加します.

```
struct HitInfo {
    t: f64,
    p: Point3,
    n: Vec3,
    m: Arc<dyn Material>,
}

impl HitInfo {
    fn new(t: f64, p: Point3, n: Vec3, m: Arc<dyn Material>) -> Self {
        Self { t, p, n, m }
    }
}
```

所有権を共有するので, `Arc` を使っています. `Arc` は頻繁に使いますので, `src/rayt/mod.rs` に `pub use std::sync::Arc;` を追加しておきます. また, スレッド間で所有権を移動（共有）するので, `Material` トレイトには `Send` マーカートレイトを継承させています. 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, `HitInfo` の材質に設定するようにします.

```
struct Sphere {
    center: Point3,
    radius: f64,
    material: Arc<dyn Material>,
}

impl Sphere {
    fn new(center: Point3, radius: f64, material: Arc<dyn Material>) -> Self {
```

```
        Self { center, radius, material }
    }
}

impl Shape for Sphere {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let oc = ray.origin - self.center;
        let a = ray.direction.dot(ray.direction);
        let b = 2.0 * ray.direction.dot(oc);
        let c = oc.dot(oc) - self.radius.powi(2);
        let d = b * b - 4.0 * a * c;
        if d > 0.0 {
            let root = d.sqrt();
            let temp = (-b - root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                return Some(HitInfo::new(
                    temp,
                    p,
                    (p - self.center) / self.radius,
                    Arc::clone(&self.material)
                ));
            }
            let temp = (-b + root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                return Some(HitInfo::new(
                    temp,
                    p,
                    (p - self.center) / self.radius,
                    Arc::clone(&self.material)
                ));
            }
        }

        None
    }
}
```

そして，`trace` 関数で材質を処理します．

```
fn trace(&self, ray: Ray) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let scatter_info = hit.m.scatter(&ray, &hit);
        if let Some(scatter) = scatter_info {
            return scatter.albedo * self.trace(scatter.ray);
        } else {
            return Color::zero();
        }
    } else {
        self.background(ray.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．

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(Box::new(Sphere::new(
        Point3::new(0.6, 0.0, -1.0),
        0.5,
        Arc::new(Lambertian::new(Color::new(0.1, 0.2, 0.5))),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(-0.6, 0.0, -1.0),
        0.5,
        Arc::new(Lambertian::new(Color::new(0.8, 0.0, 0.0))),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(0.0, -100.5, -1.0),
        100.0,
        Arc::new(Lambertian::new(Color::new(0.8, 0.8, 0.0))),
    )));
    Self { world }
}
```

この結果は次のようになります．

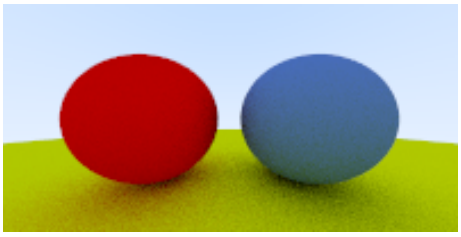
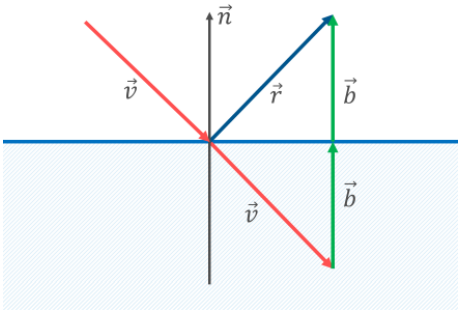


Fig.2.12: code111

14． 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

```
// src/rayt/float3.rs
pub fn reflect(&self, normal: Self) -> Self {
    *self - 2.0 * self.dot(normal) * normal
}
```

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．

```
struct Metal {
    albedo: Color,
}

impl Metal {
    fn new(albedo: Color) -> Self {
        Self { albedo }
    }
}
```

`Material` トレイトメソッドを実装します．

```
impl Material for Metal {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let reflected = ray.direction.normalize().reflect(hit.n);
        if reflected.dot(hit.n) > 0.0 {
            Some(ScatterInfo::new(Ray::new(hit.p, reflected), self.albedo))
        } else {
            None
        }
    }
}
```

`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 `ScatterInfo` を返し，それ以外は `None` を返します．シーンの球体 1 つを金属材質に変えてみます．

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(Box::new(Sphere::new(
        Point3::new(0.6, 0.0, -1.0),
        0.5,
        Arc::new(Lambertian::new(Color::new(0.1, 0.2, 0.5))),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(-0.6, 0.0, -1.0),
        0.5,
        Arc::new(Metal::new(Color::new(0.8, 0.8, 0.8))),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(0.0, -100.5, -1.0),
        100.0,
        Arc::new(Lambertian::new(Color::new(0.8, 0.8, 0.0))),
    )));
    Self { world }
}
```

次のような画像になります．

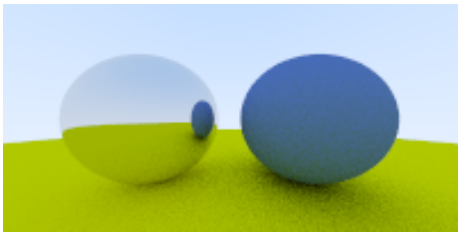
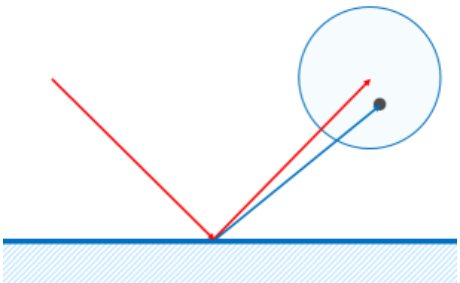


Fig.2.13: code112

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．

```
struct Metal {
    albedo: Color,
    fuzz: f64,
}

impl Metal {
    fn new(albedo: Color, fuzz: f64) -> Self {
        Self { albedo, fuzz }
    }
}

impl Material for Metal {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let mut reflected = ray.direction.normalize().reflect(hit.n);
        reflected = reflected + self.fuzz * Vec3::random_in_unit_sphere();
        if reflected.dot(hit.n) > 0.0 {
            Some(ScatterInfo::new(Ray::new(hit.p, reflected), self.albedo))
        } else {
            None
        }
    }
}
```

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 = reflected + self.fuzz * Vec3::random_in_unit_sphere();
```

球体の金属材質を調整します．

```
world.push(Box::new(Sphere::new(
    Point3::new(-0.6, 0.0, -1.0),
    0.5,
    Arc::new(Metal::new(Color::new(0.8, 0.8, 0.8), 1.0)),
)));
```

この結果は次のようになります．

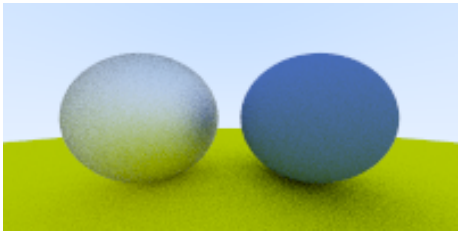
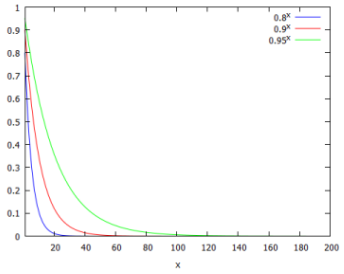


Fig.2.14: code113

ぼやけているように見えます．

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．

`trace` 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． `trace` 関数を次のように書き換えます．

```
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            scatter.albedo * self.trace(scatter.ray, depth - 1)
        } else {
            Color::zero()
        }
    } else {
        self.background(ray.direction)
    }
}
```

`depth` は再起呼び出しの深さで， `0` になったら打ち切ります．これに対応するために，レンダーモジュールを書き換える必要があります．まず，深さを指定できる新しいシーントレイト `SceneWithDepth` を定義します．

```
// src/rayt/render.rs
pub trait SceneWithDepth {
    fn camera(&self) -> Camera;
    fn trace(&self, ray: Ray, depth: usize) -> Color;
    fn width(&self) -> u32 { IMAGE_WIDTH }
    fn height(&self) -> u32 { IMAGE_HEIGHT }
    fn spp(&self) -> usize { SAMPLES_PER_PIXEL }
    fn aspect(&self) -> f64 { self.width() as f64 / self.height() as f64 }
}
```

次に `render_aa` 関数をコピーして， `render_aa_with_depth` 関数を作成し， `trace` 関数を呼び出しているところで， `depth` に `MAX_RAY_BOUNCE_DEPTH` を渡します．

```rust
// src/rayt/render.rs
const MAX_RAY_BOUNCE_DEPTH: usize = 50;
pub fn render_aa_with_depth(scene: impl SceneWithDepth + Sync) {
    backup();

    let camera = scene.camera();
    let mut img = RgbImage::new(scene.width(), scene.height());
    img.enumerate_pixels_mut()
        .collect::<Vec<(u32, u32, &mut Rgb<u8>)>>()
        .par_iter_mut()
        .for_each(|(x, y, pixel)| {
            let mut pixel_color = (0..scene.spp()).into_iter().fold(Color::zero(), |acc, _| {
                let [rx, ry, _] = Float3::random().to_array();
                let u = (*x as f64 + rx) / (scene.width() - 1) as f64;
                let v = ((scene.height() - *y - 1) as f64 + ry) / (scene.height() - 1) as f64;
                let ray = camera.ray(u, v);
                acc + scene.trace(ray, MAX_RAY_BOUNCE_DEPTH)
            });
            pixel_color /= scene.spp() as f64;
            let rgb = pixel_color.gamma(GAMMA_FACTOR).to_rgb();
            pixel[0] = rgb[0];
            pixel[1] = rgb[1];
            pixel[2] = rgb[2];
        });
    img.save(OUTPUT_FILENAME).unwrap();
    draw_in_window(BACKUP_FILENAME, img).unwrap();
}
```

そして，`SimpleScene` で実装するトレイトを `SceneWithDepth` に変更して，`render_aa` を `render_aa_with_depth` に置換えます．

```rust
impl SceneWithDepth for SimpleScene {
    fn camera(&self) -> Camera {
        Camera::new(
            Vec3::new(4.0, 0.0, 0.0),
            Vec3::new(0.0, 2.0, 0.0),
            Vec3::new(-2.0, -1.0, -1.0),
        )
    }

    fn trace(&self, ray: Ray, depth: usize) -> Color {
        let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
        if let Some(hit) = hit_info {
            let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
            if let Some(scatter) = scatter_info {
                scatter.albedo * self.trace(scatter.ray, depth - 1)
            } else {
                Color::zero()
            }
        } else {
            self.background(ray.direction)
        }
```

```
    }
}

fn main() {
    render_aa_with_depth(SimpleScene::new());
}
```

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．

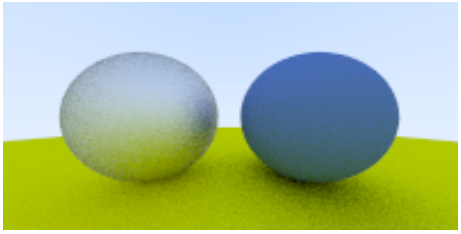


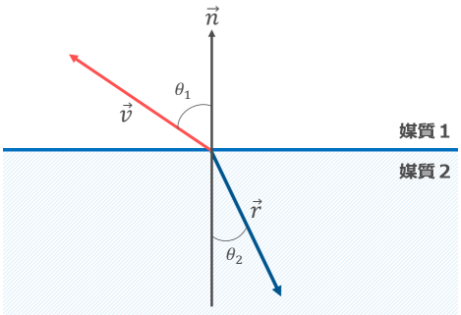
Fig.2.15: code114

15． 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．

```
// src/rayt/float3.rs
pub fn refract(&self, normal: Self, ni_over_nt: f64) -> Option<Float3> {
    let uv = self.normalize();
    let dt = uv.dot(normal);
    let d = 1.0 - ni_over_nt.powi(2) * (1.0 - dt.powi(2));
    if d > 0.0 {
        Some(-ni_over_nt * (uv - normal * dt) - normal * d.sqrt())
    } else {
        None
    }
}
```

`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 $0 \le \sin\theta_1 \le 1$ 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 $0 \le \sin\theta_2 \le 1$ 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

```
struct Dielectric {
    ri: f64,
}

impl Dielectric {
    const fn new(ri: f64) -> Self {
        Self { ri }
    }
}
```

`Material` トレイトメソッドを実装します．

```
impl Material for Dielectric {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let reflected = ray.direction.reflect(hit.n);
        let (outward_normal, ni_over_nt) = {
            if ray.direction.dot(hit.n) > 0.0 {
                (-hit.n, self.ri)
            } else {
                (hit.n, self.ri.recip())
            }
        };
        if let Some(refracted) = (-ray.direction).refract(outward_normal, ni_over_nt) {
            Some(ScatterInfo::new(Ray::new(hit.p, refracted), Color::one()))
        } else {
            Some(ScatterInfo::new(Ray::new(hit.p, reflected), Color::one()))
        }
```

```
    }
}
```

`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．

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(Box::new(Sphere::new(
        Point3::new(0.6, 0.0, -1.0),
        0.5,
        Arc::new(Lambertian::new(Color::new(0.1, 0.2, 0.5))),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(-0.6, 0.0, -1.0),
        0.5,
        Arc::new(Dielectric::new(1.5)),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(-0.0, -0.35, -0.8),
        0.15,
        Arc::new(Metal::new(Color::new(0.8, 0.8, 0.8), 0.2)),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(0.0, -100.5, -1.0),
        100.0,
        Arc::new(Lambertian::new(Color::new(0.8, 0.8, 0.0))),
    )));
    Self { world }
}
```

次のような画像になります．

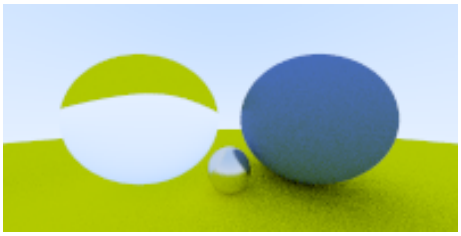


Fig.2.16: code115

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．

```
fn schlick(cosine: f64, ri: f64) -> f64 {
    let r0 = ((1.0 - ri) / (1.0 + ri)).powi(2);
    r0 + (1.0 - r0) * (1.0 - cosine).powi(5)
}
```

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．これを誘導体に組み込みます．次のように書き換えます．

```
impl Material for Dielectric {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let reflected = ray.direction.reflect(hit.n);
        let (outward_normal, ni_over_nt, cosine) = {
            let dot = ray.direction.dot(hit.n);
            if dot > 0.0 {
                (-hit.n, self.ri, self.ri * dot / ray.direction.length())
            } else {
                (hit.n, self.ri.recip(), -dot / ray.direction.length())
            }
        };

        if let Some(refracted) = (-ray.direction).refract(outward_normal, ni_over_nt) {
            if Vec3::random_full().x() > Self::schlick(cosine, self.ri) {
                return Some(ScatterInfo::new(Ray::new(hit.p, refracted), Color::one()));
            }
        }

        Some(ScatterInfo::new(Ray::new(hit.p, reflected), Color::one()))
    }
}
```

`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．

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(Box::new(Sphere::new(
        Point3::new(0.6, 0.0, -1.0),
        0.5,
        Arc::new(Lambertian::new(Color::new(0.1, 0.2, 0.5))),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(-0.6, 0.0, -1.0),
        0.5,
```

```
        Arc::new(Dielectric::new(1.5)),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(-0.6, 0.0, -1.0),
        -0.45,
        Arc::new(Dielectric::new(1.5)),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(-0.0, -0.35, -0.8),
        0.15,
        Arc::new(Metal::new(Color::new(0.8, 0.8, 0.8), 0.2)),
    )));
    world.push(Box::new(Sphere::new(
        Point3::new(0.0, -100.5, -1.0),
        100.0,
        Arc::new(Lambertian::new(Color::new(0.8, 0.8, 0.0))),
    )));
    Self { world }
}
```

この結果は次のようになります.

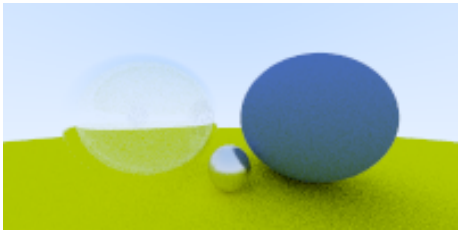


Fig.2.17: code116

16. 沢山表示してみる

物体を多く置いてみる前に，物体を追加するためのコードが複雑になってきたので対応します．Rust はコンストラクタの影響で初期化コードが長くなる傾向があります．そのためによく使われるのが Builder パターンと呼ばれるものです． `ShapeBuilder` を作成し，物体を作るメソッドをチェーンで組めるようにします.

```
struct ShapeBuilder {
    material: Option<Arc<dyn Material>>,
    shape: Option<Box<dyn Shape>>,
}

impl ShapeBuilder {
    fn new() -> Self {
        Self { material: None, shape: None }
    }

    // materials
```

```
    fn lambertian(mut self, albedo: Color) -> Self {
        self.material = Some(Arc::new(Lambertian::new(albedo)));
        self
    }

    fn metal(mut self, albedo: Color, fuzz: f64) -> Self {
        self.material = Some(Arc::new(Metal::new(albedo, fuzz)));
        self
    }

    fn dielectric(mut self, ri: f64) -> Self {
        self.material = Some(Arc::new(Dielectric::new(ri)));
        self
    }

    // shapes

    fn sphere(mut self, center: Point3, radius: f64) -> Self {
        self.shape = Some(Box::new(Sphere::new(center, radius, self.material.unwrap())));
        self.material = None;
        self
    }

    // build

    fn build(self) -> Box<dyn Shape> {
        self.shape.unwrap()
    }
}
```

`ShapeBuilder` はまずマテリアルを作成してから，物体を作成し，最後に `build` で取得します．`ShapeBuilder` を使って，ランダムに生成したシーン `RandomScene` を作成します．

```
struct RandomScene {
    world: ShapeList,
}

impl RandomScene {
    fn new() -> Self {
        let mut world = ShapeList::new();

        world.push(ShapeBuilder::new()
            .lambertian(Color::new(0.5, 0.5, 0.5))
            .sphere(Point3::new(0.0, -1000.0, 0.0), 1000.0)
            .build());

        // Small spheres
        for au in -11..11 {
            let a = au as f64;
            for bu in -11..11 {
                let b = bu as f64;
                let [rx, rz, material_choice] = Float3::random().to_array();
                let center = Point3::new(a + 0.9 * rx, 0.2, b + 0.9 * rz);
```

```
                if (center - Point3::new(4.0, 0.2, 0.0)).length() > 0.9 {
                    world.push({
                        if material_choice < 0.8 {
                            let albedo = Color::random() * Color::random();
                            ShapeBuilder::new()
                                .lambertian(albedo)
                                .sphere(center, 0.2)
                                .build()
                        } else if material_choice < 0.95 {
                            let albedo = Color::random_limit(0.5, 1.0);
                            let fuzz = Float3::random_full().x();
                            ShapeBuilder::new()
                                .metal(albedo, fuzz)
                                .sphere(center, 0.2)
                                .build()
                        } else {
                            ShapeBuilder::new()
                                .dielectric(1.5)
                                .sphere(center, 0.2)
                                .build()
                        }
                    });
                }
            }
        }

        // Big spheres
        world.push(ShapeBuilder::new()
            .dielectric(1.5)
            .sphere(Point3::new(0.0, 1.0, 0.0), 1.0)
            .build());
        world.push(ShapeBuilder::new()
            .lambertian(Color::new(0.4, 0.2, 0.1))
            .sphere(Point3::new(-4.0, 1.0, 0.0), 1.0)
            .build());
        world.push(ShapeBuilder::new()
            .metal(Color::new(0.7, 0.6, 0.5), 0.0)
            .sphere(Point3::new(4.0, 1.0, 0.0), 1.0)
            .build());

        Self { world }
    }

    fn background(&self, d: Vec3) -> Color {
        let t = 0.5 * (d.normalize().y() + 1.0);
        Color::one().lerp(Color::new(0.5, 0.7, 1.0), t)
    }
}
```

そして，カメラを LookAt 方式で指定して実装します

```
impl SceneWithDepth for RandomScene {
    fn camera(&self) -> Camera {
        Camera::from_lookat(
            Point3::new(13.0, 2.0, 3.0),
            Point3::new(0.0, 0.0, 0.0),
            Vec3::yaxis(),
            20.0,
            self.aspect(),
        )
    }

    fn trace(&self, ray: Ray, depth: usize) -> Color {
        let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
        if let Some(hit) = hit_info {
            let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
            if let Some(scatter) = scatter_info {
                scatter.albedo * self.trace(scatter.ray, depth - 1)
            } else {
                Color::zero()
            }
        } else {
            self.background(ray.direction)
        }
    }
}
```

結果は次のようになります.

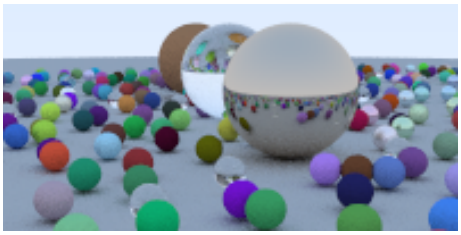


Fig.2.18: code117

第 3 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トレイトを定義します．

```
trait Texture: Sync + Send {
    fn value(&self, u: f64, v: f64, p: Point3) -> Color;
}
```

`Material` インスタンスが `Texture` を保持するので， `Send+Sync` マーカートレイトを継承する必要があります． `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， `HitInfo` に追加します．

```
struct HitInfo {
    t: f64,
    p: Point3,
    n: Vec3,
    m: Arc<dyn Material>,
    u: f64,
    v: f64,
}

impl HitInfo {
    fn new(t: f64, p: Point3, n: Vec3, m: Arc<dyn Material>, u: f64, v: f64) -> Self {
        Self { t, p, n, m, u, v }
    }
}
```

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．

```
struct ColorTexture {
    color: Color,
}

impl ColorTexture {
    const fn new(color: Color) -> Self {
        Self { color }
    }
}

impl Texture for ColorTexture {
    fn value(&self, _u: f64, _v: f64, _p: Point3) -> Color {
        self.color
    }
}
```

`value` 関数では単にカラー（反射率）を返しています．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．まずは，`Lambertian` を対応します．

```
struct Lambertian {
    albedo: Box<dyn Texture>,
}

impl Lambertian {
    fn new(albedo: Box<dyn Texture>) -> Self {
        Self { albedo }
    }
}

impl Material for Lambertian {
    fn scatter(&self, _ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let target = hit.p + hit.n + Vec3::random_in_unit_sphere();
        let albedo = self.albedo.value(hit.u, hit.v, hit.p);
        Some(ScatterInfo::new(Ray::new(hit.p, target - hit.p), albedo))
    }
}
```

次に，`Metal` のアルベドをテクスチャに変更します．

```
struct Metal {
    albedo: Box<dyn Texture>,
    fuzz: f64,
}

impl Metal {
    fn new(albedo: Box<dyn Texture>, fuzz: f64) -> Self {
        Self { albedo, fuzz }
```

```
    }
}

impl Material for Metal {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let mut reflected = ray.direction.normalize().reflect(hit.n);
        reflected = reflected + self.fuzz * Vec3::random_in_unit_sphere();
        if reflected.dot(hit.n) > 0.0 {
            let albedo = self.albedo.value(hit.u, hit.v, hit.p);
            Some(ScatterInfo::new(Ray::new(hit.p, reflected), albedo))
        } else {
            None
        }
    }
}
```

`ShapeBuilder` も対応させます.

```
struct ShapeBuilder {
    texture: Option<Box<dyn Texture>>,
    material: Option<Arc<dyn Material>>,
    shape: Option<Box<dyn Shape>>,
}

impl ShapeBuilder {
    fn new() -> Self {
        Self { texture: None, material: None, shape: None }
    }

    // textures

    fn color_texture(mut self, color: Color) -> Self {
        self.texture = Some(Box::new(ColorTexture::new(color)));
        self
    }

    // materials

    fn lambertian(mut self) -> Self {
        self.material = Some(Arc::new(Lambertian::new(self.texture.unwrap())));
        self.texture = None;
        self
    }

    fn metal(mut self, fuzz: f64) -> Self {
        self.material = Some(Arc::new(Metal::new(self.texture.unwrap(), fuzz)));
        self.texture = None;
        self
    }

    fn dielectric(mut self, ri: f64) -> Self {
        self.material = Some(Arc::new(Dielectric::new(ri)));
        self
```

```
    }

    // shapes

    fn sphere(mut self, center: Point3, radius: f64) -> Self {
        self.shape = Some(Box::new(Sphere::new(center, radius, self.material.unwrap())));
        self.material = None;
        self
    }

    // build

    fn build(self) -> Box<dyn Shape> {
        self.shape.unwrap()
    }
}
```

他には，`HitInfo` のコンストラクタでエラーになるので，今は `0.0` を指定しておきます．そして，球体の生成のところを書き換えます．`RandomScene` ではなく `SimpleScene` に戻っていることに注意してください．

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(ShapeBuilder::new()
        .color_texture(Color::new(0.1, 0.2, 0.5))
        .lambertian()
        .sphere(Point3::new(0.6, 0.0, -1.0), 0.5)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::new(0.8, 0.8, 0.8))
        .metal(0.4)
        .sphere(Point3::new(-0.6, 0.0, -1.0), 0.5)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::new(0.8, 0.8, 0.0))
        .lambertian()
        .sphere(Point3::new(0.0, -100.5, -1.0), 100.0)
        .build());

    Self { world }
}
```

これを実行すると次のようになります．

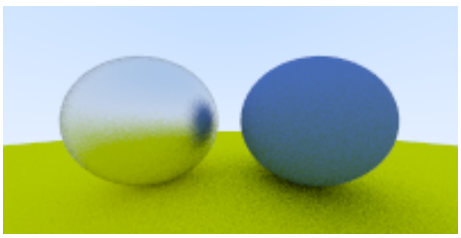


Fig.3.1: code201

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．

```
struct CheckerTexture {
    odd: Box<dyn Texture>,
    even: Box<dyn Texture>,
    freq: f64,
}

impl CheckerTexture {
    fn new(odd: Box<dyn Texture>, even: Box<dyn Texture>, freq: f64) -> Self {
        Self { odd, even, freq }
    }
}

impl Texture for CheckerTexture {
    fn value(&self, u: f64, v: f64, p: Point3) -> Color {
        let sines = p.iter().fold(1.0, |acc, x| acc * (x * self.freq).sin());
        if sines < 0.0 {
            self.odd.value(u, v, p)
        } else {
            self.even.value(u, v, p)
        }
    }
}
```

`ColorTexture` を２つ設定し，sin 関数を使って交互に描きます． `freq` は周波数で，縞模様の間隔を調整できます．これを， `ShapeBuilder` に追加します．

```
fn checker_texture(mut self, odd_color: Color, even_color: Color, freq: f64) -> Self {
    self.texture = Some(Box::new(CheckerTexture::new(
        Box::new(ColorTexture::new(odd_color)),
        Box::new(ColorTexture::new(even_color)),
        freq,
    )));
    self
}
```

このテクスチャを床に設定してみます．

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(ShapeBuilder::new()
        .color_texture(Color::new(0.1, 0.2, 0.5))
        .lambertian()
        .sphere(Point3::new(0.6, 0.0, -1.0), 0.5)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::new(0.8, 0.8, 0.8))
        .metal(0.4)
```

```
        .sphere(Point3::new(-0.6, 0.0, -1.0), 0.5)
        .build());
    world.push(ShapeBuilder::new()
        .checker_texture(
            Color::new(0.8, 0.8, 0.0),
            Color::new(0.8, 0.2, 0.0),
            10.0)
        .lambertian()
        .sphere(Point3::new(0.0, -100.5, -1.0), 100.0)
        .build());

    Self { world }
}
```

次のようになります.

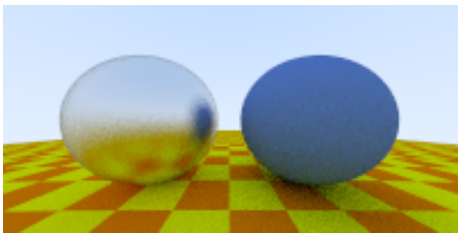


Fig.3.2: code202

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．

$$x = \cos(\phi)\cos(\theta)$$

$$y = \cos(\phi)\sin(\theta)$$

$$z = \sin(\theta)$$

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，`asin` 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.

```
impl Sphere {
    ...

    fn uv(p: Point3) -> (f64, f64) {
        let phi = p.z().atan2(p.x());
        let theta = p.y().asin();
        (1.0 - (phi + PI) / (2.0 * PI), (theta + PI / 2.0) / 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.

```
impl Shape for Sphere {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let oc = ray.origin - self.center;
        let a = ray.direction.dot(ray.direction);
        let b = 2.0 * ray.direction.dot(oc);
        let c = oc.dot(oc) - self.radius.powi(2);
        let d = b * b - 4.0 * a * c;
        if d > 0.0 {
            let root = d.sqrt();
            let temp = (-b - root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                let n = (p - self.center) / self.radius;
                let (u, v) = Self::uv(n);
                return Some(HitInfo::new(temp, p, n, Arc::clone(&self.material), u, v));
            }
            let temp = (-b + root) / (2.0 * a);
            if t0 < temp && temp < t1 {
                let p = ray.at(temp);
                let n = (p - self.center) / self.radius;
                let (u, v) = Self::uv(n);
                return Some(HitInfo::new(temp, p, n, Arc::clone(&self.material), u, v));
            }
        }

        None
    }
}
```

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 `image` パッケージを使います.

```
struct ImageTexture {
    pixels: Vec<Color>,
    width: usize,
    height: usize,
```

```
}

impl ImageTexture {
    fn new(path: &str) -> Self {
        let rgbimg = image::open(path).unwrap().to_rgb8();
        let (w, h) = rgbimg.dimensions();
        let mut image = vec![Color::zero(); (w * h) as usize];
        for (i, (_, _, pixel)) in image.iter_mut().zip(rgbimg.enumerate_pixels()) {
            *i = Color::from_rgb(pixel[0], pixel[1], pixel[2]);
        }
        Self { pixels: image, width: w as usize, height: h as usize }
    }

    fn sample(&self, u: i64, v: i64) -> Color {
        let tu = if u < 0 { 0 } else if u as usize >= self.width { self.width - 1 } else { u as
            usize };
        let tv = if v < 0 { 0 } else if v as usize >= self.height { self.height - 1 } else { v as
            usize };
        self.pixels[tu + self.width * tv]
    }
}

impl Texture for ImageTexture {
    fn value(&self, u: f64, v: f64, _p: Point3) -> Color {
        let x = (u * self.width as f64) as i64;
        let y = ((1.0 - v) * self.height as f64) as i64;
        self.sample(x, y)
    }
}
```

`image::open` で画像データをファイルから読み込みます．読み込んだイメージから `Vec<Color>` 型の `pixels` に格納します．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．これを `ShapeBuilder` に追加します．

```
fn image_texture(mut self, path: &str) -> Self {
    self.texture = Some(Box::new(ImageTexture::new(path)));
    self
}
```

次の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．

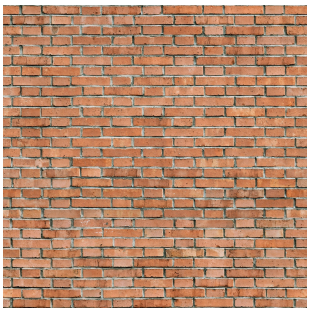


Fig.3.3: brick_diffuse.jpg

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(ShapeBuilder::new()
        .image_texture("resources/brick_diffuse.jpg")
        .lambertian()
        .sphere(Point3::new(0.6, 0.0, -1.0), 0.5)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::new(0.8, 0.8, 0.8))
        .metal(0.4)
        .sphere(Point3::new(-0.6, 0.0, -1.0), 0.5)
        .build());
    world.push(ShapeBuilder::new()
        .checker_texture(
            Color::new(0.8, 0.8, 0.0),
            Color::new(0.8, 0.2, 0.0),
            10.0)
        .lambertian()
        .sphere(Point3::new(0.0, -100.5, -1.0), 100.0)
        .build());

    Self { world }
}
```

レンダリングすると次のような画像になります．

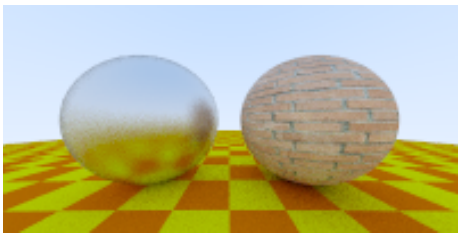


Fig.3.4: code203

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． `Material` トレイトに発光色を返すメソッドを追加して，発光する材質はオーバーライドします．

```
trait Material: Sync + Send {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo>;
    fn emitted(&self, _ray: &Ray, _hit: &HitInfo) -> Color { Color::zero() }
}
```

既存の材質インスタンスには変更を加えないように黒を返すように規定実装します．発光材質を `DiffuseLight` とし，発光色はテクスチャで設定します．

```rust
struct DiffuseLight {
    emit: Box<dyn Texture>,
}

impl DiffuseLight {
    fn new(emit: Box<dyn Texture>) -> Self {
        Self { emit }
    }
}

impl Material for DiffuseLight {
    fn scatter(&self, _ray: &Ray, _hit: &HitInfo) -> Option<ScatterInfo> {
        None
    }

    fn emitted(&self, _ray: &Ray, hit: &HitInfo) -> Color {
        self.emit.value(hit.u, hit.v, hit.p)
    }
}
```

散乱はしないので，`scatter` 関数は何もせずに `None` を返します．あとは反射した光に発光を加えるために，`trace` を書き換えます．

```rust
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            emitted + scatter.albedo * self.trace(scatter.ray, depth - 1)
        } else {
            emitted
        }
    } else {
        self.background(ray.direction)
    }
}
```

`DiffuseLight` を `ShapeBuilder` に追加します．

```rust
fn diffuse_light(mut self) -> Self {
    self.material = Some(Arc::new(DiffuseLight::new(self.texture.unwrap())));
    self.texture = None;
    self
}
```

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．

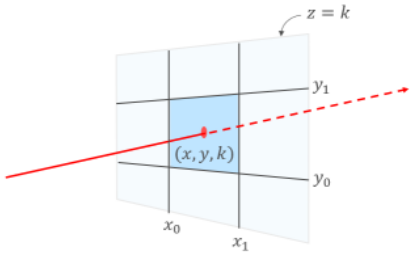
```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(ShapeBuilder::new()
        .color_texture(Color::one())
        .diffuse_light()
        .sphere(Point3::new(0.0, 0.0, -1.0), 0.5)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::full(0.8))
        .lambertian()
        .sphere(Point3::new(0.0, -100.5, -1.0), 100.0)
        .build());

    Self { world }
}

fn background(&self, _: Vec3) -> Color {
    Color::full(0.1)
}
```

実行結果は次のようになります．

Fig.3.5: code204

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．まずは XY, XZ, YZ 軸のどれかを持つ列挙型を定義します．

```
enum RectAxisType {
    XY,
    XZ,
    YZ,
}
```

次に `Rect` を実装します．

```
struct Rect {
    x0: f64,
    x1: f64,
    y0: f64,
    y1: f64,
    k: f64,
    axis: RectAxisType,
    material: Arc<dyn Material>,
}

impl Rect {
    fn new(x0: f64, x1: f64, y0: f64, y1: f64, k: f64, axis: RectAxisType, material: Arc<dyn
        Material>) -> Self {
        Self { x0, x1, y0, y1, k, axis, material }
    }
}

impl Shape for Rect {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let mut origin = ray.origin;
        let mut direction = ray.direction;
        let mut axis = Vec3::zaxis();
        match self.axis {
            RectAxisType::XY => {}
            RectAxisType::XZ => {
                origin = Point3::new(origin.x(), origin.z(), origin.y());
                direction = Vec3::new(direction.x(), direction.z(), direction.y());
                axis = Vec3::yaxis();
```

```
            }
            RectAxisType::YZ => {
                origin = Point3::new(origin.y(), origin.z(), origin.x());
                direction = Vec3::new(direction.y(), direction.z(), direction.x());
                axis = Vec3::xaxis();
            }
        }

        let t = (self.k - origin.z()) / direction.z();
        if t < t0 || t > t1 {
            return None;
        }

        let x = origin.x() + t * direction.x();
        let y = origin.y() + t * direction.y();
        if x < self.x0 || x > self.x1 || y < self.y0 || y > self.y1 {
            return None;
        }

        Some(HitInfo::new(
            t,
            ray.at(t),
            axis,
            Arc::clone(&self.material),
            (x - self.x0) / (self.x1 - self.x0),
            (y - self.y0) / (self.y1 - self.y0),
        ))
    }
}
```

`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．

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

`Rect` を `ShapeBuilder` に追加します．

```
fn rect_xy(mut self, x0: f64, x1: f64, y0: f64, y1: f64, k: f64) -> Self {
    self.shape = Some(Box::new(Rect::new(x0, x1, y0, y1, k, RectAxisType::XY, self.material.unwrap
        ())));
    self.material = None;
    self
}

fn rect_xz(mut self, x0: f64, x1: f64, y0: f64, y1: f64, k: f64) -> Self {
    self.shape = Some(Box::new(Rect::new(x0, x1, y0, y1, k, RectAxisType::XZ, self.material.unwrap
        ())));
    self.material = None;
    self
}

fn rect_yz(mut self, x0: f64, x1: f64, y0: f64, y1: f64, k: f64) -> Self {
```

```
    self.shape = Some(Box::new(Rect::new(x0, x1, y0, y1, k, RectAxisType::YZ, self.material.unwrap
        ())));
    self.material = None;
    self
}
```

それでは四角形をレンダリングしてみます.

```
fn new() -> Self {
    let mut world = ShapeList::new();
    world.push(ShapeBuilder::new()
        .color_texture(Color::full(0.5))
        .lambertian()
        .sphere(Point3::new(0.0, 2.0, 0.0), 2.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::full(4.0))
        .diffuse_light()
        .rect_xy(3.0, 5.0, 1.0, 3.0, -2.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::full(0.8))
        .lambertian()
        .sphere(Point3::new(0.0, -1000.0, 0.0), 1000.0)
        .build());

    Self { world }
}

fn background(&self, _: Vec3) -> Color {
    Color::full(0.1)
}
```

`camera` メソッドで作るカメラも変更します.

```
fn camera(&self) -> Camera {
    Camera::from_lookat(
        Vec3::new(13.0, 2.0, 3.0),
        Vec3::yaxis(),
        Vec3::yaxis(),
        30.0,
        self.aspect(),
    )
}
```

これは次のようになります.

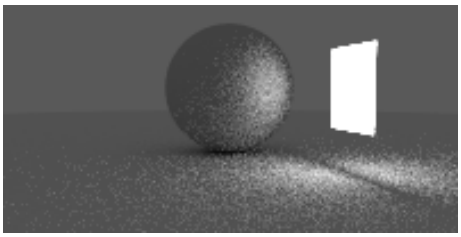


Fig.3.6: code205

C 言語などでは計算によってオーバーフローを起こす可能性があります．Rust では Debug でオーバーフローしたときに `panic!` が呼ばれます．また，Rust `1.45` から，浮動小数点数から整数への型変換時にオーバーフローする場合は，自動的にクランプされます．現在のコードでは，`to_rgb` メソッドで浮動小数点数 (`f64`) を型変換で整数 (`u8`) に変換しています．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．

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．

```
impl CornelBoxScene {
    fn new() -> Self {
        let mut world = ShapeList::new();

        let red = Color::new(0.64, 0.05, 0.05);
        let white = Color::full(0.73);
        let green = Color::new(0.12, 0.45, 0.15);

        world.push(ShapeBuilder::new()
            .color_texture(green)
            .lambertian()
            .rect_yz(0.0, 555.0, 0.0, 555.0, 555.0)
            .build());
        world.push(ShapeBuilder::new()
            .color_texture(red)
            .lambertian()
            .rect_yz(0.0, 555.0, 0.0, 555.0, 0.0)
            .build());
        world.push(ShapeBuilder::new()
            .color_texture(Color::full(15.0))
            .diffuse_light()
            .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
            .build());
        world.push(ShapeBuilder::new()
            .color_texture(white)
            .lambertian()
            .rect_xz(0.0, 555.0, 0.0, 555.0, 555.0)
            .build());
```

```
        world.push(ShapeBuilder::new()
            .color_texture(white)
            .lambertian()
            .rect_xz(0.0, 555.0, 0.0, 555.0, 0.0)
            .build());
        world.push(ShapeBuilder::new()
            .color_texture(white)
            .lambertian()
            .rect_xy(0.0, 555.0, 0.0, 555.0, 555.0)
            .build());

        Self { world }
    }

    fn background(&self, _: Vec3) -> Color {
        Color::full(0.0)
    }
}
```

カメラや画像の高さも変更します.

```
impl SceneWithDepth for CornelBoxScene {
    fn camera(&self) -> Camera {
        Camera::from_lookat(
            Vec3::new(278.0, 278.0, -800.0),
            Vec3::new(278.0, 278.0, 0.0),
            Vec3::yaxis(),
            40.0,
            self.aspect(),
        )
    }

    ...

    fn width(&self) -> u32 { 200 }
    fn height(&self) -> u32 { 200 }
}
```

この結果は次のようになります.

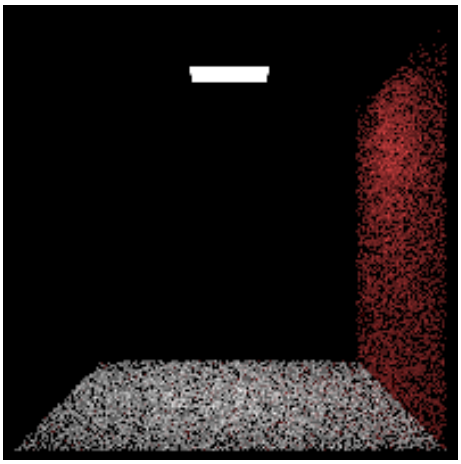


Fig.3.7: code206

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` インスタンスの `FlipFace` を追加します．これは別の `Shape` オブジェクトを束縛していて， `hit` 関数では，そのオブジェクトの `hit` 関数を呼んで法線を反転させます．

```
struct FlipFace {
    shape: Box<dyn Shape>,
}

impl FlipFace {
    fn new(shape: Box<dyn Shape>) -> Self {
        Self { shape }
    }
}

impl Shape for FlipFace {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        if let Some(hit) = self.shape.hit(ray, t0, t1) {
            Some(HitInfo { n: -hit.n, ..hit })
        } else {
            None
        }
    }
}
```

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．まずは， `FlipFace` を `ShapeBuilder` に追加します．

```
// decorators

fn flip_face(mut self) -> Self {
    self.shape = Some(Box::new(FlipFace::new(self.shape.unwrap())));
    self
}
```

```
fn new() -> Self {
    let mut world = ShapeList::new();

    let red = Color::new(0.64, 0.05, 0.05);
    let white = Color::full(0.73);
    let green = Color::new(0.12, 0.45, 0.15);

    world.push(ShapeBuilder::new()
        .color_texture(green)
        .lambertian()
        .rect_yz(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(red)
        .lambertian()
        .rect_yz(0.0, 555.0, 0.0, 555.0, 0.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::full(15.0))
        .diffuse_light()
        .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xz(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xz(0.0, 555.0, 0.0, 555.0, 0.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xy(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());

    Self { world }
}
```

結果は次のようになります.

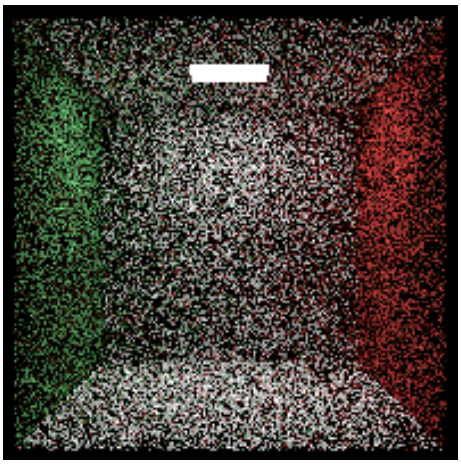


Fig.3.8: code207

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．

```
struct Box3D {
    p0: Point3,
    p1: Point3,
    shapes: ShapeList,
}

impl Box3D {
    fn new(p0: Point3, p1: Point3, material: Arc<dyn Material>) -> Self {
        let mut shapes = ShapeList::new();
        shapes.push(ShapeBuilder::new()
            .material(Arc::clone(&material))
            .rect_xy(p0.x(), p1.x(), p0.y(), p1.y(), p1.z())
            .build());
        shapes.push(ShapeBuilder::new()
            .material(Arc::clone(&material))
            .rect_xy(p0.x(), p1.x(), p0.y(), p1.y(), p0.z())
            .flip_face()
            .build());
        shapes.push(ShapeBuilder::new()
            .material(Arc::clone(&material))
            .rect_xz(p0.x(), p1.x(), p0.z(), p1.z(), p1.y())
            .build());
        shapes.push(ShapeBuilder::new()
            .material(Arc::clone(&material))
            .rect_xz(p0.x(), p1.x(), p0.z(), p1.z(), p0.y())
            .flip_face()
            .build());
        shapes.push(ShapeBuilder::new()
            .material(Arc::clone(&material))
            .rect_yz(p0.y(), p1.y(), p0.z(), p1.z(), p1.x())
            .build());
        shapes.push(ShapeBuilder::new()
            .material(Arc::clone(&material))
            .rect_yz(p0.y(), p1.y(), p0.z(), p1.z(), p0.x())
```

```
            .flip_face()
            .build());

        Self { p0, p1, shapes }
    }
}

impl Shape for Box3D {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        self.shapes.hit(ray, t0, t1)
    }
}
```

`Box` という名前にしたかったのですが，Rust では予約語になっているので，`Box3D` としました．`ShapeBuilder` に追加します．

```
fn box3d(mut self, p0: Point3, p1: Point3) -> Self {
    self.shape = Some(Box::new(Box3D::new(p0, p1, self.material.unwrap())));
    self.material = None;
    self
}
```

シーンに箱を２つ追加します．

```
world.push(ShapeBuilder::new()
    .color_texture(white)
    .lambertian()
    .box3d(Point3::new(130.0, 0.0, 65.0), Point3::new(295.0, 165.0, 230.0))
    .build());
world.push(ShapeBuilder::new()
    .color_texture(white)
    .lambertian()
    .box3d(Point3::new(265.0, 0.0, 295.0), Point3::new(430.0, 330.0, 460.0))
    .build());
```

結果は次のようになります．

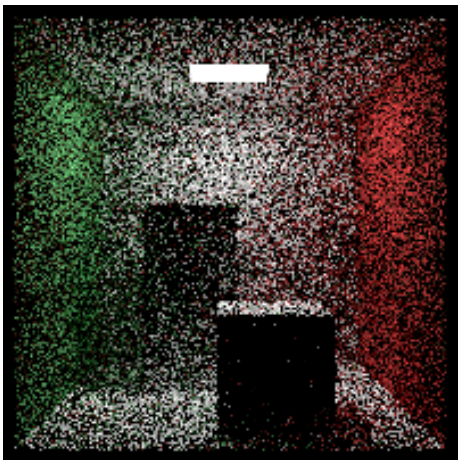


Fig.3.9: code208

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．まずは移動（ `Translate` ）の実装です．

```
struct Translate {
    shape: Box<dyn Shape>,
    offset: Point3,
}

impl Translate {
    fn new(shape: Box<dyn Shape>, offset: Point3) -> Self {
        Self { shape, offset }
    }
}

impl Shape for Translate {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let moved_ray = Ray::new(ray.origin - self.offset, ray.direction);
        if let Some(hit) = self.shape.hit(&moved_ray, t0, t1) {
            Some(HitInfo { p: hit.p + self.offset, ..hit })
        } else {
            None
        }
    }
}
```

次に，回転（ `Rotate` ）の実装です．

```
struct Rotate {
    shape: Box<dyn Shape>,
    quat: Quat,
}

impl Rotate {
    fn new(shape: Box<dyn Shape>, axis: Vec3, angle: f64) -> Self {
        Self { shape, quat: Quat::from_rot(axis, angle.to_radians()) }
    }
}

impl Shape for Rotate {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo> {
        let revq = self.quat.conj();
        let rotated_ray = Ray::new(revq.rotate(ray.origin), revq.rotate(ray.direction));
        if let Some(hit) = self.shape.hit(&rotated_ray, t0, t1) {
            Some(HitInfo { p: self.quat.rotate(hit.p), n: self.quat.rotate(hit.n), ..hit })
        } else {
            None
        }
    }
}
```

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．これらを `ShapeBuilder` に追加します．

```
fn translate(mut self, offset: Point3) -> Self {
    self.shape = Some(Box::new(Translate::new(self.shape.unwrap(), offset)));
    self
}

fn rotate(mut self, axis: Vec3, angle: f64) -> Self {
    self.shape = Some(Box::new(Rotate::new(self.shape.unwrap(), axis, angle)));
    self
}
```

移動と回転を使ってみましょう.

```
world.push(ShapeBuilder::new()
    .color_texture(white)
    .lambertian()
    .box3d(Point3::zero(), Point3::full(165.0))
    .rotate(Vec3::yaxis(), -18.0)
    .translate(Point3::new(130.0, 0.0, 65.0))
    .build());
world.push(ShapeBuilder::new()
    .color_texture(white)
    .lambertian()
    .box3d(Point3::zero(), Point3::new(165.0, 330.0, 165.0))
    .rotate(Vec3::yaxis(), 15.0)
    .translate(Point3::new(265.0, 0.0, 295.0))
    .build());
```

これをレンダリングすると次のようになります.

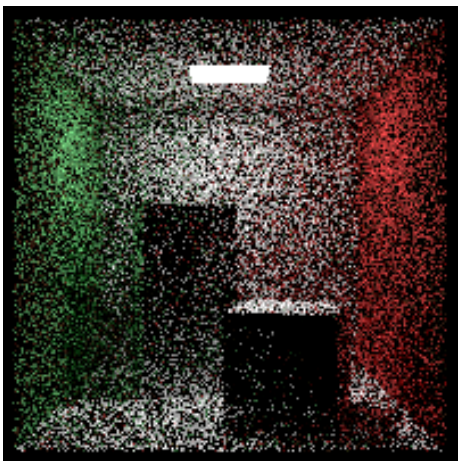


Fig.3.10: code209

第 4 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．

3．モンテカルロレイトレーシング

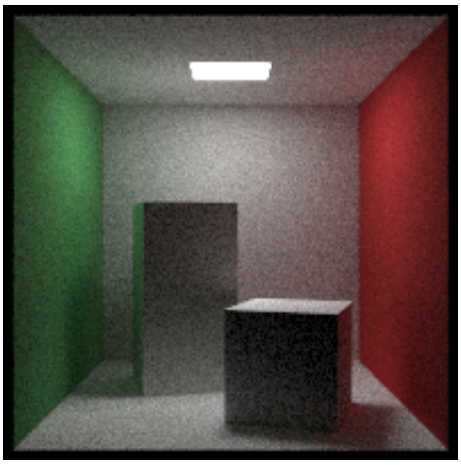
古典的なレイトレーシングは，カメラから光線を飛ばして鏡面反射させていました．その後，鏡面反射の方向の周りに向かって複数の光線を飛ばして計算するように改善されました．これを分散レイトレーシングといいます．これ

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．

3.1 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．



4． 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

`Material` トレイトに散乱 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．

```
trait Material: Sync + Send {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo>;
    fn emitted(&self, _ray: &Ray, _hit: &HitInfo) -> Color { Color::zero() }
    fn scattering_pdf(&self, _ray: &Ray, _hit: &HitInfo) -> f64 { 0.0 }
}
```

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．

```
impl Material for Lambertian {
    ...
    fn scattering_pdf(&self, ray: &Ray, hit: &HitInfo) -> f64 {
        ray.direction.normalize().dot(hit.n).max(0.0) * FRAC_1_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． `ScatterInfo` に `pdf_value` を追加し， `scatter` 関数で値を計算します．

```
struct ScatterInfo {
    ray: Ray,
    albedo: Color,
    pdf_value: f64,
}

impl ScatterInfo {
    fn new(ray: Ray, albedo: Color, pdf_value: f64) -> Self {
        Self { ray, albedo, pdf_value }
    }
}
```

```
impl Material for Lambertian {
    fn scatter(&self, _ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let target = hit.p + hit.n + Vec3::random_in_unit_sphere();
        let new_ray = Ray::new(hit.p, target - hit.p);
        let albedo = self.albedo.value(hit.u, hit.v, hit.p);
        let pdf_value = new_ray.direction.dot(hit.n) * FRAC_1_PI;
        Some(ScatterInfo::new(new_ray, albedo, pdf_value))
    }
    ...
}
```

`Metal` や `Dielectric` では `pdf_value` に `0.0` を指定しておきます． `trace` 関数で，散乱 pdf の値とアルベド，放射輝度 (`trace` 関数の返り値）の積を求めて，それに `ScatterInfo.pdf_value` で除算します．

```
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            let pdf_value = hit.m.scattering_pdf(&scatter.ray, &hit);
            let albedo = scatter.albedo * pdf_value;
            emitted + albedo * self.trace(scatter.ray, depth - 1) / scatter.pdf_value
        } else {
            emitted
        }
    } else {
```

```
            self.background(ray.direction)
        }
    }
```

これを実行すると次のようになります．

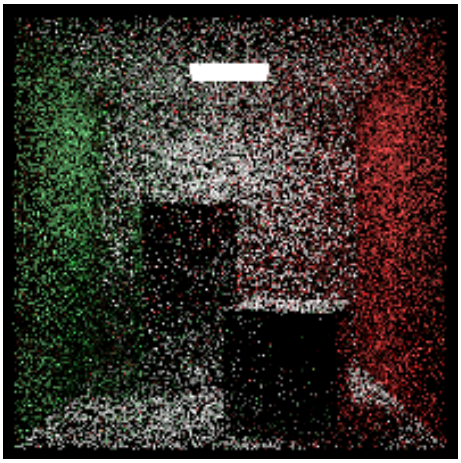


Fig.4.1: code301

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

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

$$= 2\pi \int_0^{\theta} f(\theta)\sin\theta d\theta$$

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.

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

$\cos^2 + \sin^2 + 1$ 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,

$$x = \cos(2\pi r_1)2\sqrt{r_2(1-r_2)}$$

$$y = \sin(2\pi r_1)2\sqrt{r_2(1-r_2)}$$

$$z = 1-2r_2.$$

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

$$x = \cos(2\pi r_1)\sqrt{r_2(2-r_2)}$$

$$y = \sin(2\pi r_1)\sqrt{r_2(2-r_2)}$$

$$z = 1-r_2.$$

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.

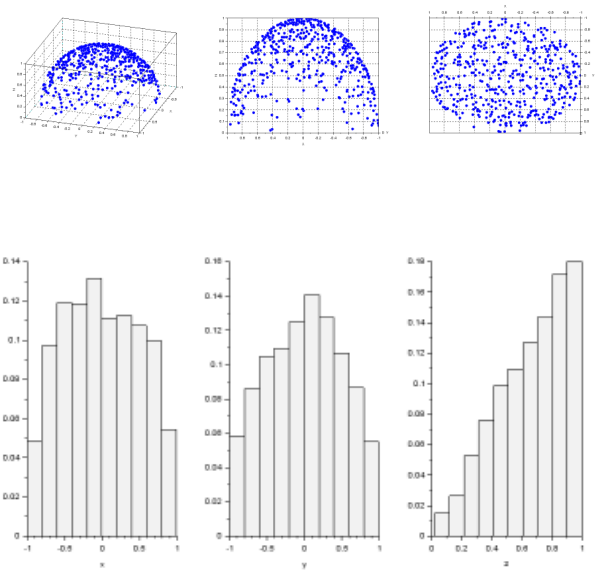
```
// src/rayt/float3.rs
pub fn random_cosine_direction() -> Self {
    let Self([r1, r2, _]) = Self::random();
    let z = (1.0 - r2).sqrt();
    let (x, y) = (PI2 * r1).sin_cos();
    let r2sqrt = r2.sqrt();
    Self::new(x * r2sqrt, y * r2sqrt, z)
}
```

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．1 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．これはモジュール化します．

```
// src/rayt/onb.rs
use crate::rayt::*;

pub struct ONB {
    axis: [Vec3; 3],
}

impl ONB {
    pub fn new(n: Vec3) -> Self {
```

```
        let w = n.normalize();
        let v = if w.x().abs() > 0.9 {
            w.cross(Vec3::yaxis()).normalize()
        } else {
            w.cross(Vec3::xaxis()).normalize()
        };
        let u = w.cross(v);
        Self { axis: [u, v, w] }
    }

    pub fn u(&self) -> Vec3 { self.axis[0] }
    pub fn v(&self) -> Vec3 { self.axis[1] }
    pub fn w(&self) -> Vec3 { self.axis[2] }

    pub fn local(&self, v: Vec3) -> Vec3 {
        self.axis[0] * v.x() + self.axis[1] * v.y() + self.axis[2] * v.z()
    }
}
```

`src/rayt/mod.rs` に追加しておきます.

```
// src/rayt/mod.rs
mod onb;
pub use self::onb::ONB;
```

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.

```
impl Material for Lambertian {
    fn scatter(&self, _ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let direction = ONB::new(hit.n).local(Vec3::random_cosine_direction());
        let new_ray = Ray::new(hit.p, direction.normalize());
        let albedo = self.albedo.value(hit.u, hit.v, hit.p);
        let pdf_value = new_ray.direction.dot(hit.n) * FRAC_1_PI;
        Some(ScatterInfo::new(new_ray, albedo, pdf_value))
    }
    ...
}
```

レンダリングすると次のようになります.

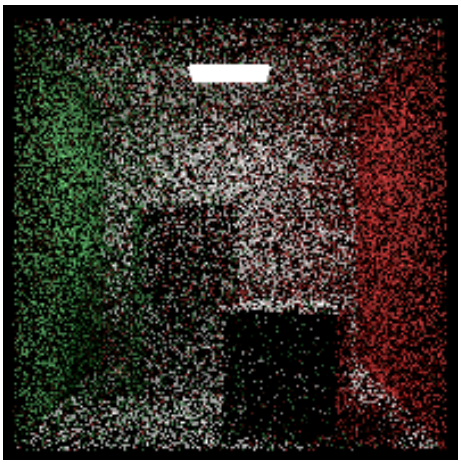
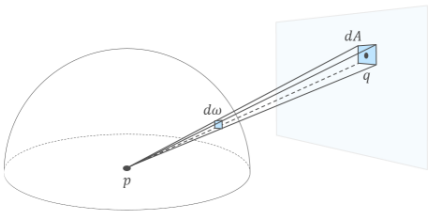


Fig.4.2: code302

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．

```
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            let [rx, rz, _] = Point3::random().to_array();
            let on_light = Point3::new(213.0 + rx * (343.0 - 213.0), 554.0, 227.0 + rz * (332.0 -
                227.0));
            let to_light = on_light - hit.p;
            let distance_squared = to_light.length_squared();
            let to_light = to_light.normalize();
            if to_light.dot(hit.n) < 0.0 {
                return emitted;
            }
            let light_area = (343.0 - 213.0) * (332.0 - 217.0);
            let light_cosine = to_light.y().abs();
            if light_cosine < 0.000001 {
                return emitted;
            }

            let spdf_value = distance_squared / (light_cosine * light_area);
            let new_ray = Ray::new(hit.p, to_light);

            let pdf_value = hit.m.scattering_pdf(&new_ray, &hit);
            let albedo = scatter.albedo * pdf_value;
            emitted + albedo * self.trace(new_ray, depth - 1) / spdf_value
        } else {
            emitted
        }
    } else {
        self.background(ray.direction)
    }
}
```

これを実行すると次のようになります.

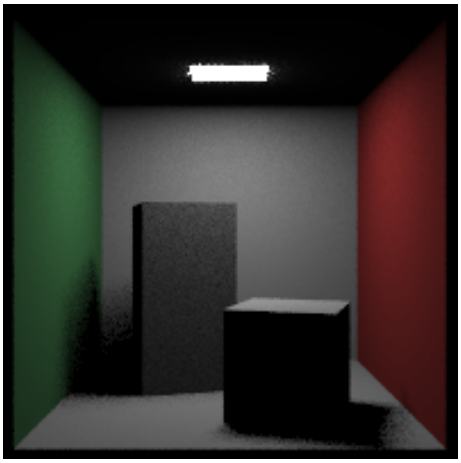


Fig.4.3: code303

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．

```
impl Material for DiffuseLight {
    fn scatter(&self, _ray: &Ray, _hit: &HitInfo) -> Option<ScatterInfo> {
        None
    }

    fn emitted(&self, ray: &Ray, hit: &HitInfo) -> Color {
        if ray.direction.dot(hit.n) < 0.0 {
            self.emit.value(hit.u, hit.v, hit.p)
        } else {
            Color::zero()
        }
    }
}
```

また，コーネルボックスの光源を反転させます．

```
world.push(ShapeBuilder::new()
    .color_texture(Color::full(15.0))
    .diffuse_light()
    .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
    .flip_face()
    .build());
```

レンダリングすると次のようになります．

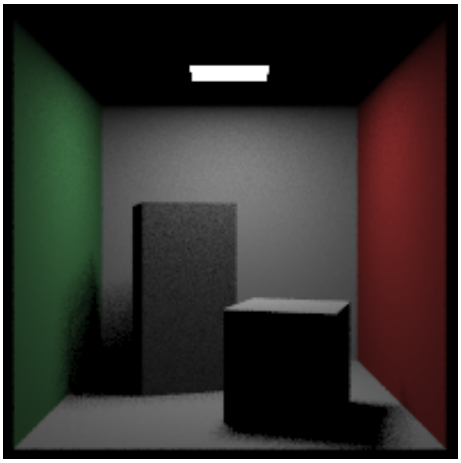


Fig.4.4: code304

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(θ) 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 をトレイトにします．

```
trait Pdf {
    fn value(&self, hit: &HitInfo, direction: Vec3) -> f64;
    fn generate(&self, hit: &HitInfo) -> Vec3;
}
```

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． `hit` 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

```
struct CosinePdf {}
impl CosinePdf {
    const fn new() -> Self {
        Self {}
    }
}
impl Pdf for CosinePdf {
    fn value(&self, hit: &HitInfo, direction: Vec3) -> f64 {
        let cosine = direction.normalize().dot(hit.n);
        if cosine > 0.0 {
            cosine * FRAC_1_PI
        } else {
            0.0
        }
    }

    fn generate(&self, hit: &HitInfo) -> Vec3 {
        ONB::new(hit.n).local(Vec3::random_cosine_direction())
    }
}
```

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

```
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            let pdf = CosinePdf::new();
            let new_ray = Ray::new(hit.p, pdf.generate(&hit));
            let spdf_value = pdf.value(&hit, new_ray.direction);
            if spdf_value > 0.0 {
                let pdf_value = hit.m.scattering_pdf(&new_ray, &hit);
                let albedo = scatter.albedo * pdf_value;
                emitted + albedo * self.trace(new_ray, depth - 1) / spdf_value
            } else {
                emitted
            }
```

```
        } else {
            emitted
        }
    } else {
        self.background(ray.direction)
    }
}
```

この結果は次のようになります.

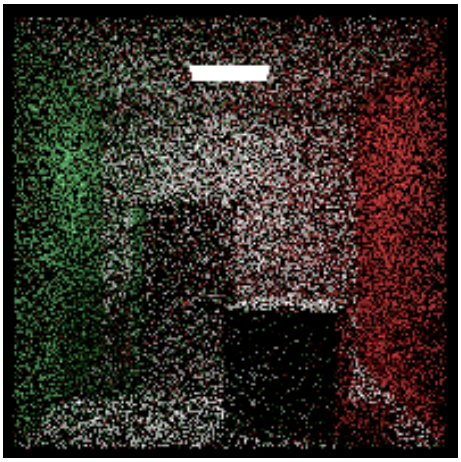


Fig.4.5: code305

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` を束縛する pdf を実装します.

```
struct ShapePdf {
    shape: Box<dyn Shape>,
    origin: Point3,
}
impl ShapePdf {
    fn new(shape: Box<dyn Shape>, origin: Point3) -> Self {
        Self { shape, origin }
    }
}
impl Pdf for ShapePdf {
    fn value(&self, _hit: &HitInfo, direction: Vec3) -> f64 {
        self.shape.pdf_value(self.origin, direction)
    }

    fn generate(&self, _hit: &HitInfo) -> Vec3 {
        self.shape.random(self.origin)
    }
}
```

そして `shape` に pdf 関連の関数を追加します.

```
trait Shape: Sync {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo>;
    fn pdf_value(&self, _o: Vec3, _v: Vec3) -> f64 { 0.0 }
    fn random(&self, _o: Vec3) -> Vec3 { Vec3::xaxis() }
}
```

`Rect` でオーバーライドします.

```
impl Shape for Rect {
    ...
    fn pdf_value(&self, o: Vec3, v: Vec3) -> f64 {
        if let Some(hit) = self.hit(&Ray::new(o, v), 0.001, f64::MAX) {
            let area = (self.x1 - self.x0) * (self.y1 - self.y0);
            let distance_squared = hit.t.powi(2) * v.length_squared();
            let cosine = v.dot(hit.n).abs() / v.length();
            distance_squared / (cosine * area)
        } else {
            0.0
        }
    }

    fn random(&self, o: Vec3) -> Vec3 {
        let [rx, ry, _] = Vec3::random().to_array();
        let x = self.x0 + rx * (self.x1 - self.x0);
        let y = self.y0 + ry * (self.y1 - self.y0);
        match self.axis {
            RectAxisType::XY => Point3::new(x, y, self.k) - o,
            RectAxisType::XZ => Point3::new(x, self.k, y) - o,
            RectAxisType::YZ => Point3::new(self.k, x, y) - o,
        }
    }
}
```

`trace` 関数も書き換えます.

```
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            let pdf = ShapePdf::new(ShapeBuilder::new()
                .color_texture(Color::zero())
                .lambertian()
                .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
                .build(), hit.p);
            let new_ray = Ray::new(hit.p, pdf.generate(&hit));
            let spdf_value = pdf.value(&hit, new_ray.direction);
            if spdf_value > 0.0 {
                let pdf_value = hit.m.scattering_pdf(&new_ray, &hit);
                let albedo = scatter.albedo * pdf_value;
```

```
                emitted + albedo * self.trace(new_ray, depth - 1) / spdf_value
            } else {
                emitted
            }
        } else {
            emitted
        }
    } else {
        self.background(ray.direction)
    }
}
```

この結果は次のようになります.

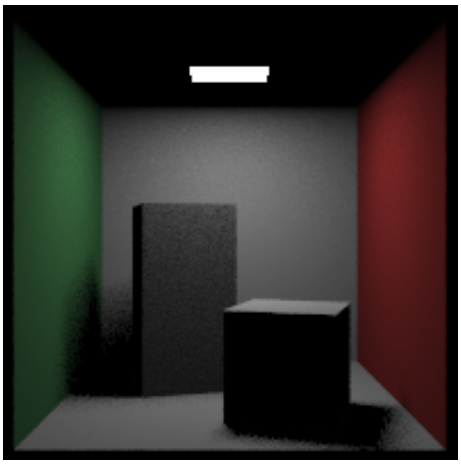


Fig.4.6: code306

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 `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.

```
struct MixturePdf {
    pdfs: [Box<dyn Pdf>; 2],
}

impl MixturePdf {
    fn new(pdf0: Box<dyn Pdf>, pdf1: Box<dyn Pdf>) -> Self {
        Self { pdfs: [pdf0, pdf1] }
    }
}

impl Pdf for MixturePdf {
    fn value(&self, hit: &HitInfo, direction: Vec3) -> f64 {
        let pdf0 = self.pdfs[0].value(&hit, direction);
        let pdf1 = self.pdfs[1].value(&hit, direction);
        0.5 * pdf0 + 0.5 * pdf1
    }
    fn generate(&self, hit: &HitInfo) -> Vec3 {
        if Vec3::random_full().x() < 0.5 {
            self.pdfs[0].generate(hit)
        } else {
```

```
            self.pdfs[1].generate(hit)
        }
    }
}
```

これを使って，光源 pdf と余弦 pdf を合成してみます.

```
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            let pdf = MixturePdf::new(
                Box::new(ShapePdf::new(ShapeBuilder::new()
                .color_texture(Color::zero())
                .lambertian()
                .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
                .build(), hit.p)),
                Box::new(CosinePdf::new()));
            let new_ray = Ray::new(hit.p, pdf.generate(&hit));
            let spdf_value = pdf.value(&hit, new_ray.direction);
            if spdf_value > 0.0 {
                let pdf_value = hit.m.scattering_pdf(&new_ray, &hit);
                let albedo = scatter.albedo * pdf_value;
                emitted + albedo * self.trace(new_ray, depth - 1) / spdf_value
            } else {
                emitted
            }
        } else {
            emitted
        }
    } else {
        self.background(ray.direction)
    }
}
```

この結果は次のようになります.

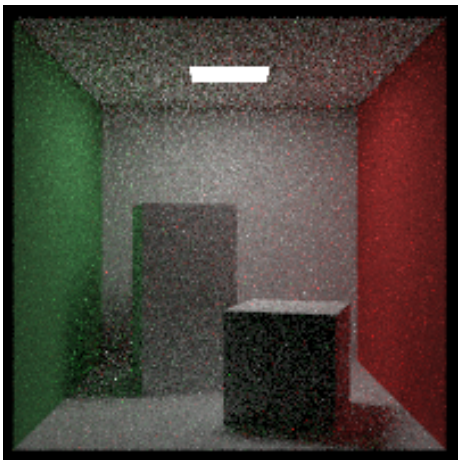


Fig.4.7: code307

これまでは `trace` 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， `ScatterInfo` の `pdf_value` を変更して `Pdf` オブジェクトを束縛できるようにします．

```
struct ScatterInfo {
    ray: Ray,
    albedo: Color,
    pdf: Option<Arc<dyn Pdf>>,
}

impl ScatterInfo {
    fn new(ray: Ray, albedo: Color, pdf: Option<Arc<dyn Pdf>>) -> Self {
        Self { ray, albedo, pdf }
    }
}
```

これによって， `Pdf` インスタンスのオブジェクトの所有権がスレッド間で移動することになるので， `Pdf` に `Send+Sync` マーカートレイトを継承します．

```
trait Pdf: Send + Sync {
    fn value(&self, hit: &HitInfo, direction: Vec3) -> f64;
    fn generate(&self, hit: &HitInfo) -> Vec3;
}
```

また， `Shape` も `ShapePdf` で束縛され，そして `ShapePdf` も `Pdf` のインスタンスなので， `Shape` は `Send+Sync` でなければなりません．

```
trait Shape: Send + Sync {
    fn hit(&self, ray: &Ray, t0: f64, t1: f64) -> Option<HitInfo>;
    fn pdf_value(&self, _o: Vec3, _v: Vec3) -> f64 { 0.0 }
    fn random(&self, _o: Vec3) -> Vec3 { Vec3::xaxis() }
}
```

次に，ランバート材質の pdf を設定します．

```rust
struct Lambertian {
    albedo: Box<dyn Texture>,
    pdf: Arc<dyn Pdf>,
}

impl Lambertian {
    fn new(albedo: Box<dyn Texture>) -> Self {
        Self { albedo, pdf: Arc::new(CosinePdf::new()) }
    }
}

impl Material for Lambertian {
    fn scatter(&self, ray: &Ray, hit: &HitInfo) -> Option<ScatterInfo> {
        let albedo = self.albedo.value(hit.u, hit.v, hit.p);
        Some(ScatterInfo::new(*ray, albedo, Some(Arc::clone(&self.pdf))))
    }

    fn scattering_pdf(&self, ray: &Ray, hit: &HitInfo) -> f64 {
        ray.direction.normalize().dot(hit.n).max(0.0) * FRAC_1_PI
    }
}
```

`Metal` や `Dielectric` には `None` を指定するようにします．次に `trace` 関数を書き換えます．

```rust
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            if let Some(pdf) = scatter.pdf {
                let pdf = MixturePdf::new(
                    Arc::new(ShapePdf::new(ShapeBuilder::new()
                    .color_texture(Color::zero())
                    .lambertian()
                    .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
                    .build(), hit.p)),
                    Arc::clone(&pdf));
                let new_ray = Ray::new(hit.p, pdf.generate(&hit));
                let spdf_value = pdf.value(&hit, new_ray.direction);
                if spdf_value > 0.0 {
                    let pdf_value = hit.m.scattering_pdf(&new_ray, &hit);
                    let albedo = scatter.albedo * pdf_value;
                    emitted + albedo * self.trace(new_ray, depth - 1) / spdf_value
                } else {
                    emitted
                }
            } else {
                emitted + scatter.albedo * self.trace(scatter.ray, depth - 1)
            }
        } else {
            emitted
```

```
        }
    } else {
        self.background(ray.direction)
    }
}
```

この結果は前回と変わらないはずです．

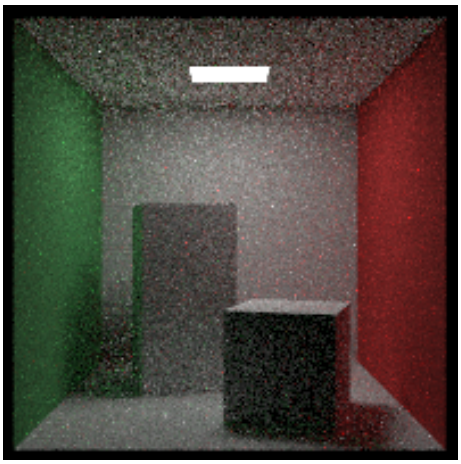


Fig.4.8: code308

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． `trace` 関数で，鏡面反射のときの処理を追加します．

```
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            if let Some(pdf) = scatter.pdf {
                let pdf = MixturePdf::new(
                    Arc::new(ShapePdf::new(ShapeBuilder::new()
                    .color_texture(Color::zero())
                    .lambertian()
                    .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
                    .build(), hit.p)),
                    Arc::clone(&pdf));
                let new_ray = Ray::new(hit.p, pdf.generate(&hit));
                let spdf_value = pdf.value(&hit, new_ray.direction);
                if spdf_value > 0.0 {
                    let pdf_value = hit.m.scattering_pdf(&new_ray, &hit);
                    let albedo = scatter.albedo * pdf_value;
                    emitted + albedo * self.trace(new_ray, depth - 1) / spdf_value
                } else {
                    emitted
```

```
                }
            } else {
                emitted + scatter.albedo * self.trace(scatter.ray, depth - 1)
            }
        } else {
            emitted
        }
    } else {
        self.background(ray.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.

```
fn new() -> Self {
    let mut world = ShapeList::new();

    let red = Color::new(0.64, 0.05, 0.05);
    let white = Color::full(0.73);
    let green = Color::new(0.12, 0.45, 0.15);

    world.push(ShapeBuilder::new()
        .color_texture(green)
        .lambertian()
        .rect_yz(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(red)
        .lambertian()
        .rect_yz(0.0, 555.0, 0.0, 555.0, 0.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::full(15.0))
        .diffuse_light()
        .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xz(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xz(0.0, 555.0, 0.0, 555.0, 0.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
```

```
        .rect_xy(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());

    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .box3d(Point3::zero(), Point3::full(165.0))
        .rotate(Vec3::yaxis(), -18.0)
        .translate(Point3::new(130.0, 0.0, 65.0))
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::new(0.8, 0.85, 0.88))
        .metal(0.0)
        .box3d(Point3::zero(), Point3::new(165.0, 330.0, 165.0))
        .rotate(Vec3::yaxis(), 15.0)
        .translate(Point3::new(265.0, 0.0, 295.0))
        .build());

    Self { world }
}
```

これを実行すると次のような画像になります．

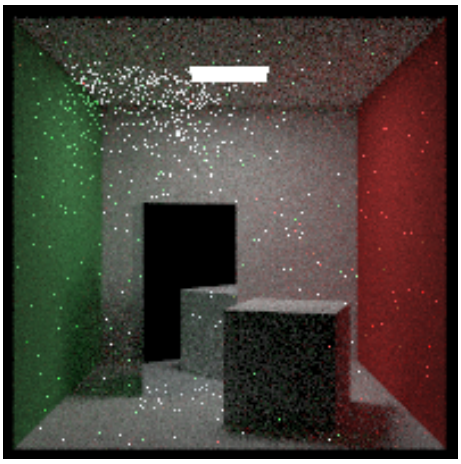


Fig.4.9: code309

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

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

よって,

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

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

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

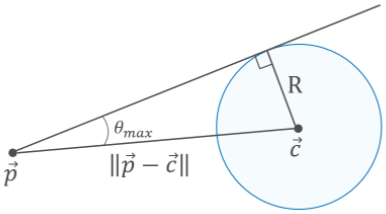
これを先ほどの式に代入すると

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

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` の `pdf_value` 関数と `random` 関数を実装します．

```
impl Shape for Sphere {
    ...
    fn pdf_value(&self, o: Vec3, v: Vec3) -> f64 {
        if let Some(_) = self.hit(&Ray::new(o, v), 0.001, f64::MAX) {
            let dd = (self.center - o).length_squared();
            let rr = self.radius.powi(2).min(dd);
            let cos_theta_max = (1.0 - rr * dd.recip()).sqrt();
            let solid_angle = PI2 * (1.0 - cos_theta_max);
            solid_angle.recip()
        } else {
            0.0
        }
    }

    fn random(&self, o: Vec3) -> Vec3 {
        let direction = self.center - o;
        let distance_squared = direction.length_squared();
        ONB::new(direction).local(Vec3::random_to_sphere(self.radius, distance_squared))
    }
}
```

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．

```rust
// src/rayt/float3.rs
pub fn random_to_sphere(radius: f64, distance_squared: f64) -> Self {
    let Self([rx, ry, _]) = Self::random();
    let rr = radius.powi(2).min(distance_squared);
    let cos_theta_max = (1.0 - rr * distance_squared.recip()).sqrt();
    let z = 1.0 - ry * (1.0 - cos_theta_max);
    let sqrtz = (1.0 - z.powi(2)).sqrt();
    let (x, y) = (PI2 * rx).sin_cos();
    Self::new(x * sqrtz, y * sqrtz, 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，`Scene` に，重点的サンプリング用の物体を束縛させます．新たに `light` フィールドを追加して束縛します．

```rust
struct CornelBoxScene {
    world: ShapeList,
    light: Arc<dyn Shape>,
}
```

`trace` 関数で処理を行います．

```rust
fn trace(&self, ray: Ray, depth: usize) -> Color {
    let hit_info = self.world.hit(&ray, 0.001, f64::MAX);
    if let Some(hit) = hit_info {
        let emitted = hit.m.emitted(&ray, &hit);
        let scatter_info = if depth > 0 { hit.m.scatter(&ray, &hit) } else { None };
        if let Some(scatter) = scatter_info {
            if let Some(pdf) = scatter.pdf {
                let shape_pdf = Arc::new(ShapePdf::new(Arc::clone(&self.light), hit.p));
                let pdf = MixturePdf::new(shape_pdf, Arc::clone(&pdf));
                let new_ray = Ray::new(hit.p, pdf.generate(&hit));
                let spdf_value = pdf.value(&hit, new_ray.direction);
                if spdf_value > 0.0 {
                    let pdf_value = hit.m.scattering_pdf(&new_ray, &hit);
                    let albedo = scatter.albedo * pdf_value;
                    emitted + albedo * self.trace(new_ray, depth - 1) / spdf_value
                } else {
                    emitted

                }
            } else {
                emitted + scatter.albedo * self.trace(scatter.ray, depth - 1)
            }
        } else {
            emitted
        }
    } else {
        self.background(ray.direction)
    }
}
```

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．

```
fn new() -> Self {
    let mut world = ShapeList::new();

    let red = Color::new(0.64, 0.05, 0.05);
    let white = Color::full(0.73);
    let green = Color::new(0.12, 0.45, 0.15);

    world.push(ShapeBuilder::new()
        .color_texture(green)
        .lambertian()
        .rect_yz(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(red)
        .lambertian()
        .rect_yz(0.0, 555.0, 0.0, 555.0, 0.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(Color::full(15.0))
        .diffuse_light()
        .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xz(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xz(0.0, 555.0, 0.0, 555.0, 0.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .rect_xy(0.0, 555.0, 0.0, 555.0, 555.0)
        .flip_face()
        .build());

    world.push(ShapeBuilder::new()
        .dielectric(1.5)
        .sphere(Point3::new(190.0, 90.0, 190.0), 90.0)
        .build());
    world.push(ShapeBuilder::new()
        .color_texture(white)
        .lambertian()
        .box3d(Point3::zero(), Point3::new(165.0, 330.0, 165.0))
        .rotate(Vec3::yaxis(), 15.0)
        .translate(Point3::new(265.0, 0.0, 295.0))
```

```
        .build());

    let mut light = ShapeList::new();
    light.push(ShapeBuilder::new()
        .color_texture(Color::zero())
        .lambertian()
        .sphere(Point3::new(190.0, 90.0, 190.0), 90.0)
        .build());

    Self { world, light: Arc::new(light) }
}
```

レンダリングすると次のようになります.

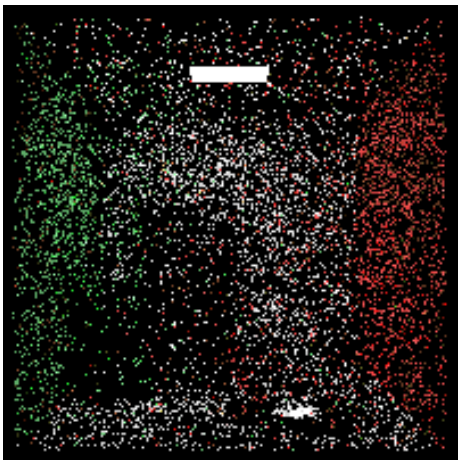


Fig.4.10: code310

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.

```
impl Shape for ShapeList {
    ...
    fn pdf_value(&self, o: Vec3, v: Vec3) -> f64 {
        if self.objects.is_empty() { panic!(); }
        let weight = 1.0 / self.objects.len() as f64;
        self.objects.iter().fold(0.0,|acc, s| acc + weight * s.pdf_value(o, v))
    }

    fn random(&self, o: Vec3) -> Vec3 {
        if self.objects.is_empty() { panic!(); }
        let index = (Vec3::random_full().x() * self.objects.len() as f64).floor() as usize;
        self.objects[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.

```
let mut light = ShapeList::new();
light.push(ShapeBuilder::new()
    .color_texture(Color::zero())
    .lambertian()
    .rect_xz(213.0, 343.0, 227.0, 332.0, 554.0)
    .build());
light.push(ShapeBuilder::new()
    .color_texture(Color::zero())
    .lambertian()
    .sphere(Point3::new(190.0, 90.0, 190.0), 90.0)
    .build());

Self { world, light: Arc::new(light) }
```

これをレンダリングすると次のようになります．

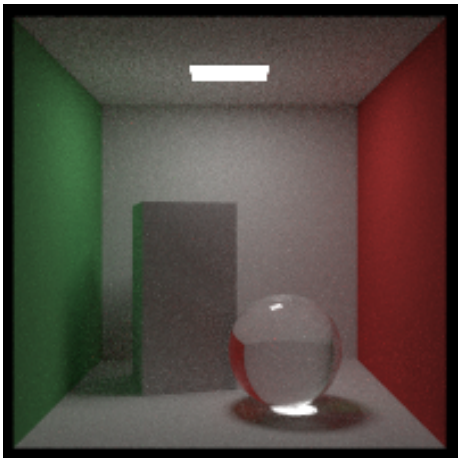


Fig.4.11: code311

これが目標の画像になります．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．

https://raytracing.github.io/

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ー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ー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
- レイトレ合宿 7

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

本書は，流行りの Rust 言語を勉強したので，以前に書いた「レイトレーシング入門」を Rust で書き直したものです．私自身，まだ Rust 言語を使い始めたばかりなので，まだわからないことが多いです．何か間違いなどありましたら，ご連絡していただけると助かります．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