POV-Ray 実習マニュアル

第五版 revision01

大黒学

POV-Ray 実習マニュアル・第五版 revision01
著者——大黒学

2000 年 10 月 13 日（金）　第零版発行
2001 年　8 月　9 日（木）　第一版発行
2002 年　1 月 24 日（木）　第二版発行
2004 年　3 月 19 日（金）　第三版発行
2008 年　1 月 30 日（水）　第四版発行
2013 年 12 月　4 日（水）　第五版発行
2015 年　2 月 10 日（火）　第五版 revision01 発行

目次

第 1 章　POV-Ray の基礎

1.1　POV-Ray の基礎の基礎

1.1.1　3 次元グラフィックスとは何か

コンピュータを使って画像を作成する方法には、2 次元的な方法と 3 次元的な方法があります。2 次元的な方法を使って作られた画像は「2 次元グラフィックス」(two-dimensional graphics) と呼ばれ、3 次元的な方法を使って作られた画像は「3 次元グラフィックス」(three-dimensional graphics) と呼ばれます。

画像を作るための 2 次元的な方法というのは、平面に対して人間が直接に画像を描画するという方法のことです。それに対して、3 次元的な方法というのは、物体や光源やカメラなどを記述したデータを人間が作って、そのデータに対する計算によって画像を生成するという方法のことです。

物体や光源やカメラなどを記述したデータから画像を生成する計算は、「レンダリング」(rendering) と呼ばれます。

また、3 次元グラフィックスを作るために必要となる、物体の形状を作るという作業は、「モデリング」(modeling) と呼ばれます。

1.1.2　POV-Ray とは何か

レンダリングを実行するプログラム（コンピュータのソフト）は、「レンダラー」(renderer) と呼ばれます。

レンダラーにはさまざまなものがあるのですが、そのうちのひとつに、POV-Ray（読み方は「ポブレイ」）と呼ばれるものがあります。POV-Ray は、Persistence of Vision Raytracer というのが正式名称で、

- 無料で配布されている。
- 物体や光源やカメラなどのデータがテキストである。

という特徴を持っています。

1.1.3　POV-Team

POV-Ray は、POV-Team と呼ばれるボランティアの人々によって開発されていて、無料で配布されているソフトです。ですから、それをダウンロードしたり、パソコンにインストールしたり、使用したりする上で、誰かにお金を払う必要は、まったくありません。

POV-Ray は、その公式サイト[1]に置かれていて、そこから自由にダウンロードすることができます。

1.1.4　シーン

POV-Ray によって処理される、物体や光源やカメラなどが記述されたデータは、「シーン」(scene) と呼ばれます。

シーンは、テキストデータ、つまり文字で書かれているデータです。ですから、テキストエディターを使うことによって、それを入力したり修正したりすることができます。

シーンは、「シーン記述言語」(scene description language, SDL) と呼ばれる言語を使って記述されます。

ちなみに、この「POV-Ray 実習マニュアル」という文章は、POV-Ray のシーン記述言語について解説することを目的とするチュートリアルです。

1.2　POV-Ray の使い方

1.2.1　シーンの入力

この節では、POV-Ray の使い方について説明したいと思います。

POV-Ray を使って画像を生成するためには、まず最初に、レンダリングの対象となるシーンをコンピュータに入力する必要があります。シーンというのはテキストデータですので、テキストエディターを使うことによってシーンを入力することができます。

[1] POV-Ray 公式サイトの URL は、`http://www.povray.org/` です。

Windows 版の POV-Ray はテキストエディターを内蔵していますので、それを使うことによってシーンを入力することができます。[New] というボタンをクリックすると、[Untitled] というタブの付いた入力領域が開いて、そこにシーンを入力することができる状態になります。そして、シーンを保存したいときは、[Save] というボタンをクリックします。

それでは、実際にシーンを入力して見ましょう。次のシーンを入力して、`sphere.pov` というファイルに保存してください。

シーンの例 `sphere.pov`

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
}
```

この例のように、POV-Ray のシーンを格納するファイルには、`.pov` という拡張子を付けることになっています。

1.2.2 レンダリング

Windows 版の POV-Ray の場合、レンダリングは、[Run] というボタンを押すことによって開始させることができます。レンダリングの実行中は [Run] のボタンが [Stop] に変わりますので、レンダリングを途中で止めたい場合は、その [Stop] を押してください。

それでは、先ほど入力したシーンを POV-Ray にレンダリングさせてみましょう。シーンの中にエラーが何もなければ、球が空中に浮かんでいる画像が生成されるはずです。エラーがあった場合はエラーメッセージが表示されますので、それを読んで、エラーを修正してください。

1.2.3 画像のサイズとアンチエイリアシングの設定

Windows 版の POV-Ray の場合、ツールバーの下にあるリストボックスを使うことによって、生成される画像のサイズと、アンチエイリアシングに関する設定することができます。「アンチエイリアシング」(antialiasing) というのは、画像の中の斜めの線がギザギザに見えないように処理することです。

リストボックスのそれぞれの項目の中に書かれている、512×384 というような数字の組は、画像の横の長さと縦の長さをピクセルの個数で示したものです。

ピクセルの個数の右側に書かれている No AA とか AA 0.3 というのは、アンチエイリアシングに関する指定です。No AA というのはアンチエイリアシングをしないという意味で、AA 0.3 というのは、0.3 をしきい値（閾値）とするアンチエイリアシングをするという意味です。

レンダリングに要する時間は、画像のサイズに比例して長くなります。また、アンチエイリアシングを実行すると、それをしない場合よりもレンダリングに要する時間が長くなります。ですから、シーンがまだ完成していない段階で、レンダリングの結果を確認したいときは、アンチエイリアシングをともなわない小さなサイズでレンダリングするといいでしょう。

1.3 シーンの基礎

1.3.1 シーンの構成要素

POV-Ray のシーンは、カメラ (camera)、光源 (light source)、物体 (object) という三つの基本的な要素から構成されます。

三つの基本的な要素は、それぞれ、次のような形で書きます。

カメラ `camera { ... }`

光源 `light_source { ... }`

物体 `object { ... }`

第 1.2.1 項で入力したシーンも、これらの三つの要素を含んでいますので、確認してみてください。

1.3.2　ホワイトスペース

空白 (space)、タブ (tab)、改行 (line feed) のような、文字と文字を引き離すために使われる文字は、総称して「ホワイトスペース」(white space) と呼ばれます。

POV-Ray のシーンの中で、二つの単語が連続する場合は、1 個以上のホワイトスペースでそれらを区切る必要があります。

また、単語やコンマや括弧などの前後には、任意の個数のホワイトスペースを挿入することができます。この場合のホワイトスペースは、記述の意味を変化させません。たとえば、

```
light_source { <-5, 5, -5> color rgb 1.6 }
```

という光源の記述は、

```
light_source {
    <-5, 5, -5>
    color rgb 1.6
}
```

と書いたとしても、同じ意味だと解釈されます。

1.3.3　注釈

シーンの中には、シーンを読む人間に読んでもらうことを目的とするテキストを書くこともできます。そのようなテキストは、「注釈」(comment) と呼ばれます。注釈を書くときは、POV-Ray がシーンを解釈するときに注釈を無視するように書かないといけません。

シーンの中の注釈を POV-Ray に無視してもらう方法は二つあります。ひとつはスラッシュスラッシュ(`//`) を使う方法です。シーンの中に `//` を書くと、その直後から最初の改行までが無視されます。たとえば、POV-Ray は、

```
// I am a comment.
```

という記述を、注釈とみなして無視します。

注釈を無視してもらう方法の二つ目は、スラッシュアスタリスク (`/*`) とアスタリスクスラッシュ (`*/`) でそれを囲むという方法です。改行を含んでいる注釈、つまり 2 行以上の注釈も、その全体を `/*` と `*/` で囲むことによって、無視してもらうことができます。たとえば、POV-Ray は、

```
/* I am a comment
which contains a line feed. */
```

という記述を、注釈とみなして無視します。

シーンを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合、無効にしたい部分を削除してしまうと、復元するのに手間がかかります。そのような場合には、通常、その部分を削除するのではなく、注釈にします。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。

1.4　式

1.4.1　式の基礎

POV-Ray のシーンの中では、長さ、角度、位置、方向などのさまざまな量を指定するために、「式」(expression) と呼ばれる記述が使われます。

式に対して実行される動作は、「評価」(evaluation) と呼ばれます。

式を評価すると、その結果として、何らかのデータが得られます。式を評価することによって得られたデータは、その式の「値」(value) と呼ばれます。

1.4.2　リテラル

評価すると、その値として特定のデータが得られる式は、「リテラル」(literal) と呼ばれます。そして、得られるデータが数値であるようなリテラルは、「数値リテラル」(numeric literal) と呼ばれます。

583、-240、0.017、というような 10 進数は、数値リテラルとして評価することができて、その値として、それがあらわしている数値が得られます。

巨大な数値や微小な数値が値として得られる数値リテラルは、「10 の何乗」を意味する e という文字を使って記述することも可能です。たとえば、1e30 という数値リテラルの値は、

```
1000000000000000000000000000000
```

という数値リテラルの値と同じです。同じように、1e-30 という数値リテラルの値は、

```
0.000000000000000000000000000001
```

という数値リテラルの値と同じです。

1.4.3 演算子

「演算子」(operator) と呼ばれるものを使って式を組み合わせたものも、式の一種です。演算子というのは、「演算」(operation) と呼ばれる動作に与えられた名前のことです。

演算子を使って式を組み合わせた式は、

式 演算子 式

と書きます。この形の式を評価すると、演算子の左右の式の値に対して、演算子があらわしている演算が実行されて、その結果が式全体の値になります。

1.4.4 算術演算子

数値に対する計算をあらわしている演算子は、「算術演算子」(arithmetic operator) と呼ばれます。

POV-Ray には、次の四つの算術演算子があります。

A + *B*　*A* と *B* とを足し算（加算）する。

A - *B*　*A* から *B* を引き算（減算）する。

A * *B*　*A* と *B* とを掛け算（乗算）する。

A / *B*　*A* を *B* で割り算（除算）する。整数の範囲で割り切れない場合は小数点以下も求める。

たとえば、530+42 という式を評価すると、530 と 42 との加算が実行されて、値として 572 が得られます。

* と / は、+ と - よりも優先順位が高くて、* と / は同じ優先順位、+ と - も同じ優先順位です。たとえば、7*3+4 という式は、7 と 3 とを乗算した結果と 4 とを加算するという意味だと解釈されますので、その値は 25 になります。

式の一部分を丸括弧で囲むと、演算子の優先順位にかかわらず、囲まれた部分はひとつの式であると解釈されます。たとえば、7*(3+4) という式は、7 と、3 と 4 とを加算した結果とを乗算するという意味だと解釈されますので、その値は 49 になります。

1.4.5 ベクトル

式を評価することによって得られる値は、必ずしも単独の数値とは限りません。「ベクトル」(vector) と呼ばれるものが値として得られる式もあります。ベクトルというのは、何個かの数値を一列に並べたもののことです。

ベクトルを構成するそれぞれの数値は、そのベクトルの「成分」(component) と呼ばれます。

ベクトルの成分の個数は、そのベクトルの「次元」(dimension) と呼ばれます。次元が n のベクトルは、「n 次元ベクトル」(n-dimensional vector) と呼ばれます。1 次元ベクトル、つまり 1 個の数値は、「スカラー」(scalar) と呼ばれます。

成分のそれぞれを求める式を記述することによってベクトルを求めたいときは、

<式, 式, ⋯, 式>

という形の式を書きます。たとえば、

```
<75, 34, 81>
```

という式を評価すると、その値として、75 と 34 と 81 をこの順序で並べることによってできるベクトルが得られます。

ベクトルを必要とする場所に、値としてスカラーが得られる式を書くと、得られたスカラーは、それを必要な個数だけ並べることによってできるベクトルに変換されます。たとえば、3 次元ベクトルを必要とする場所に 20 という式を書いたとすると、その式の値は、

```
<20, 20, 20>
```

というベクトルに変換されます。

1.4.6　ベクトルの加算

2 個のベクトルが与えられたときに、それぞれのベクトルを構成している成分のそれぞれについて、同じ順番のものどうしを加算して、その結果として得られた数値を同じ順番で並べることによって 1 個のベクトルを求める、という演算は、「ベクトルの加算」と呼ばれます。そして、ベクトルの加算によって得られたベクトルは、与えられた 2 個のベクトルの「和」(sum) と呼ばれます。

+ という演算子を使うことによって、ベクトルとベクトルの和を求めることができます。たとえば、

```
<5, 3, 4> + <2, 7, 8>
```

という式を評価すると、

```
<7, 10, 12>
```

というベクトルが値として得られます。

1.4.7　ベクトルの乗算

1 個のベクトルと 1 個のスカラーが与えられたときに、ベクトルを構成している成分のそれぞれとスカラーを乗算して、その結果として得られた数値を同じ順番で並べることによって 1 個のベクトルを求める、という演算は、「ベクトルの乗算」と呼ばれます。そして、ベクトルの乗算によって得られたベクトルは、与えられたベクトルとスカラーの「積」(product) と呼ばれます。

* という演算子を使うことによって、ベクトルとスカラーの積を求めることができます。たとえば、

```
<5, 2, 3> * 40
```

という式を評価すると、

```
<200, 80, 120>
```

というベクトルが値として得られます。

1.5　座標系

1.5.1　座標系の基礎

空間の中にある点の位置を指定するための仕組みは、「座標系」(coordinate system) と呼ばれます。

座標系は、「軸」(axis) と呼ばれる、方向を持つ直線を使って、点の位置を指定します。軸の本数は、空間の次元と一致します。ですから、1 次元空間は 1 本の軸、2 次元空間は 2 本の軸、3 次元空間は 3 本の軸を使います。

3 次元空間の座標系で使われる 3 本の軸のそれぞれは、「x 軸」(x-axis)、「y 軸」(y-axis)、「z 軸」(z-axis) と呼ばれます。これらの軸は、空間の中の 1 点で互いに直角に交わっていて、その点は「原点」(origin) と呼ばれます。

x 軸、y 軸、z 軸のそれぞれの上の位置は、それを原点から見たときに、軸が向いている方向（プラスの方向）にある場合は、原点からの距離であらわされ、それとは逆の方向（マイナスの方向）にある場合は、原点からの距離をマイナスにしたものによってあらわされます。

1.5.2　座標

3 次元空間の中にある点の位置は、「座標」(coordinates) と呼ばれる 3 次元ベクトルによって指定することができます。座標を構成している成分は、先頭から順番に、「x 座標」(x-coordinate)、「y 座標」(y-coordinate)、「z 座標」(z-coordinate) と呼ばれます。

3 次元空間の中にある点の位置を指定する座標は、その点を含む平面が x 軸と垂直に交わる位置を x 座標、その点を含む平面が y 軸と垂直に交わる位置を y 座標、その点を含む平面が z 軸と垂直に交わる位置を z 座標とする 3 次元ベクトルです。

たとえば、`<7, 3, -6>` という座標によって指定される位置というのは、7 の位置で x 軸と垂直に交わる平面と、3 の位置で y 軸と垂直に交わる平面と、-6 の位置で z 軸と垂直に交わる平面とが 1 点で交わるところのことです。

1.5.3 右手系と左手系

3 次元空間の座標系には、2 種類のものがあります。ひとつは「右手系」(right-handed system) と呼ばれ、もうひとつは「左手系」(left-handed system) と呼ばれます。右手系と左手系の相違点は、x 軸、y 軸、z 軸のそれぞれがどちらの方向を向いているか、という点にあります。

右手系の座標系は、右手の親指を x 軸、人差し指を y 軸、中指を z 軸だとみなしたときに、それぞれの指がプラスの方向を向くような座標系です。それに対して、左手について同じことが成り立つような座標系が左手系です。

3 次元空間を扱うソフトがどちらの座標系を採用しているかというのは、それぞれのソフトによって違いますので、注意が必要です。ちなみに、POV-Ray が採用しているのは左手系の座標系です。

1.6 色

1.6.1 光の三原色

色は、光の三原色のそれぞれを混ぜ合わせる比率によって記述することができます。光の三原色というのは、赤 (red)、緑 (green)、青 (blue) という三つの色のことです。光の三原色のそれぞれを混ぜ合わせる比率は、「RGB」と呼ばれます。

1.6.2 色の記述

POV-Ray のシーンの中では、RGB は、

```
color rgb 式
```

という形の記述によってあらわされます。この中の「式」のところには、3 次元ベクトルを値とする式を書きます。その 3 次元ベクトルは、赤、緑、青という順番で原色の比率を並べたものだと解釈されます。原色の比率は、基本的には、0 から 1 までのあいだの数値で指定します。次の表は、どのような 3 次元ベクトルがどのような色をあらわしているかということを示しています。

`<0, 0, 0>`	黒	`<1, 1, 0>`	黄色
`<1, 1, 1>`	白	`<0, 1, 1>`	水色
`<1, 0, 0>`	赤	`<1, 0, 1>`	赤紫
`<0, 1, 0>`	ライム	`<0, 0.5, 0>`	緑
`<0, 0, 1>`	青	`<1, 0.5, 0>`	オレンジ色
`<0.5, 0.5, 0.5>`	グレー	`<0, 0.5, 1>`	空色

ちなみに、上の表から分かるとおり、原色の緑は「ライム」(lime) と呼ばれる色で、普通に「緑」と呼ばれるのは、原色の緑よりも暗い色です。

1.7 物体

1.7.1 物体の基礎

POV-Ray のシーンの中では、個々の物体は、

```
object { ... }
```

という形のものを書くことによって記述することができます。

物体の記述の中には、最低限、二つのことを書く必要があります。ひとつは物体の形状で、もうひとつは物体のテクスチャー（材質感）です。

第 1.2.1 項で入力したシーンの中には、

```
    object {
        sphere { 0, 1 }
        pigment { color rgb 1 }
    }
```

という物体の記述が書かれていました。この中にある、

```
    sphere { 0, 1 }
```

という部分が、物体の形状を記述した部分で、

```
    pigment { color rgb 1 }
```

という部分が、物体のテクスチャーを記述した部分です。

1.7.2　プリミティブ

POV-Ray の内部では、「プリミティブ」(primitive) と呼ばれるいくつかの基本的な形状が、最初から定義されています。物体の形状は、それらのプリミティブを使うことによって記述することができます。

第 1.2.1 項で入力したシーンの中にある、

```
    sphere { 0, 1 }
```

という記述は、プリミティブを使って球 (sphere) という形状を書きあらわしたものです。

プリミティブは、それに与えられている名前によって識別されます。たとえば、球を作るプリミティブは、`sphere` という名前によって識別されます。

プリミティブを使って形状を作る記述は、

プリミティブ名 { 属性の記述 }

と書きます。「属性の記述」のところにどう書けばいいのかということは、プリミティブごとに違います。たとえば、`sphere` というプリミティブを使って球を作る場合、「属性の記述」のところには、

中心の座標 , 半径

と書くことになっています。「中心の座標」のところには 3 次元ベクトルを値とする式を書いて、「半径」のところにはスカラーを値とする式を書きます。そうすると、「中心の座標」のところに書かれた式の値を座標とする点に中心があって、「半径」のところに書かれた式の値が半径であるような球が作られます。たとえば、

```
    sphere { <3, 2, 7>, 5 }
```

という記述は、`<3, 2, 7>` を中心の座標とする半径が 5 の球を作る、という意味になります。

第 1.4.5 項で説明したように、ベクトルを必要とする場所に、値としてスカラーが得られる式を書くと、得られたスカラーは、それを必要な個数だけ並べることによってできるベクトルに変換されます。ですから、第 1.2.1 項で入力したシーンの中にある、

```
    sphere { 0, 1 }
```

という記述は、その中の `0` が `<0, 0, 0>` という 3 次元ベクトルに変換されますので、原点を中心の位置とする半径が 1 の球を作る、という意味になります。

次のシーンは、異なる位置と半径を持つ二つの球を作っています。

シーンの例　`sphere2.pov`

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { <0, 0, 0>, 0.2 }
    pigment { color rgb 1 }
}
```

```
object {
    sphere { <0, -101, 0>, 100 }
    pigment { color rgb 1 }
}
```

球の位置と半径をいろいろと変更してレンダリングしてみましょう。

1.7.3 直方体

プリミティブについては、第 3 章でさまざまなものを紹介する予定なのですが、ここで、もうひとつだけプリミティブを紹介しておきたいと思います。それは、直方体 (box) を作る、box というプリミティブです。

box を使って直方体を作る記述は、

box { 頂点$_1$, 頂点$_2$ }

と書きます。「頂点$_1$」と「頂点$_2$」のそれぞれのところには、3 次元ベクトルを値とする式を書きます。そうすると、「頂点$_1$」と「頂点$_2$」のそれぞれのところに書かれた式の値を座標とする点をつなぐ直線を対角線とする直方体が作られます。たとえば、

```
box { <0, 1, 2>, <3, 4, 5> }
```

という記述を書くことによって、<0, 1, 2> と <3, 4, 5> とをつなぐ直線を対角線とする直方体を作ることができます。

box によって作られる直方体は、それぞれの辺が、x 軸、y 軸、z 軸と等しくなります。

次のシーンは、<1, 1, 1> と <2, 2, 2> とをつなぐ直線を対角線とする直方体を作っています。

シーンの例 box.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    box { <1, 1, 1>, <2, 2, 2> }
    pigment { color rgb 1 }
}
```

直方体の頂点の位置をいろいろと変更してレンダリングしてみましょう。

1.7.4 ピグメント

テクスチャーは、さまざまな要素から構成されるのですが、この項では、それらのうちの「ピグメント」(pigment) と呼ばれる要素だけについて説明します。テクスチャーのその他の要素については、第 5 章で詳しく説明することにしたいと思います。

ピグメントというのは、物体の素材が持っている色のことです。ピグメントは、物体の記述の中に、

pigment { 色の記述 }

という形のものを書くことによって指定することができます。

これまでに紹介したシーンの中に書かれている、

```
pigment { color rgb 1 }
```

という部分は、物体のピグメントを記述したものです。この記述の中にある 1 は、<1, 1, 1> という 3 次元ベクトルに変換されますので、この記述は物体に対して白色のピグメントを与えていることになります。

次のシーンは、黄緑色のピグメントを持つ球を作っています。

シーンの例 pigment.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { <0, 0, 0>, 1 }
    pigment { color rgb <0.5, 1, 0> }
}
```

球のピグメントをいろいろと変更してレンダリングしてみましょう。

1.7.5　物体の記述としてのプリミティブの記述

POV-Ray では、プリミティブの記述というのは、それ自体もまた物体の記述です。ピグメントの記述は、プリミティブの記述の中に書くことも可能です。ということは、

```
object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
}
```

という物体の記述は、

```
sphere {
    0, 1
    pigment { color rgb 1 }
}
```

と書いてもかまわないということです。

1.8　光源

1.8.1　光源の基礎

人間の目は、光がなければ物体を視覚によって認識することができません。POV-Ray の場合も同じです。光を発するもの、つまり光源をシーンの中にまったく記述しなかったとすると、ほとんど真っ黒な画像が生成されることになります（明示的に記述された光源が存在していない場合でも、「環境光」(ambient light) と呼ばれる光が存在しているため、完全に真っ黒になるわけではありません）。

POV-Ray のシーンの中では、個々の光源は、

```
light_source { ... }
```

という形のものを書くことによって記述することができます。

光源の記述の中には、最低限、二つのことを書く必要があります。ひとつは光源の位置で、もうひとつは光源の明るさと色です。

これまでに紹介してきたシーンの中には、

```
light_source { <-5, 5, -5> color rgb 1.6 }
```

という光源の記述が書かれていました。この中にある、

```
<-5, 5, -5>
```

という部分が、光源の位置を指定する座標で、

```
color rgb 1.6
```

という部分が、光源の明るさと色の記述です。

次のシーンは、<5, 0, 0> と <-5, 0, 0> に置かれた二つの光源で、原点に置かれた球を照らしています。

シーンの例　lightlocation.pov

```
camera {
    location <0, 0, -3>
```

```
    look_at 0
    angle 70
}

light_source { <5, 0, 0> color rgb 1.6 }
light_source { <-5, 0, 0> color rgb 1.6 }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
}
```

光源の位置をいろいろと変更してレンダリングしてみましょう。

1.8.2 光源の明るさと色

色は、通常、三原色のそれぞれの明るさを、0 から 1 までの数値で書きあらわすことによって記述されます。しかし、光源の場合は、明るさの記述と色の記述とが一体になっていますので、適度な明るさにするために 1 よりも大きな数値を指定することが必要になる場合もあります。

次のシーンは、オレンジ色の光源で白色の球を照らしたものです。

シーンの例 lightcolor.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb <1.6, 0.8, 0> }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
}
```

光源の色をいろいろと変更してレンダリングしてみましょう。

1.9 カメラ

1.9.1 カメラの基礎

レンダリングというのは、3 次元空間のどこかにカメラを置いて、そのカメラが撮影した映像を計算によって求めるという処理のことだと考えることができます。ですから、POV-Ray のシーンの中には、必ず、レンダリングに使われるカメラについての記述を書かないといけません。

POV-Ray のシーンの中では、カメラは、

```
camera { ... }
```

という形のものを書くことによって記述することができます。

カメラの記述の中には、通常、カメラの位置、方向、視野の角度についての記述を書きます。

1.9.2 カメラの位置

カメラの位置を指定する記述は、

```
location 位置
```

と書きます。「位置」のところに 3 次元ベクトルを値とする式を書くと、それを座標とする位置にカメラが置かれます。

これまでに紹介したシーンの中には、カメラの位置を指定する記述として、

```
location <0, 0, -3>
```

というものが書かれていました。これは、z 軸のマイナスの方向へ原点から 3 だけ移動した位置にカメラがあるということを意味しています。

次のシーンは、x 軸、y 軸、x 軸のそれぞれに沿って置かれた 3 個の直方体を、<10，5，-10> という位置に置かれたカメラで撮影しています。

シーンの例　location.pov

```
camera {
    location <10, 5, -10>
    look_at 0
    angle 70
}

light_source { <20, 30, -20> color rgb 1.6 }

object {
    box { <-10, -1, -1>, <10, 1, 1> }
    pigment { color rgb <1, 1, 0> }
}

object {
    box { <-1, -10, -1>, <1, 10, 1> }
    pigment { color rgb <1, 0, 1> }
}

object {
    box { <-1, -1, -10>, <1, 1, 10> }
    pigment { color rgb <0, 1, 1> }
}
```

カメラの位置をいろいろと変えてレンダリングしてみてください。

1.9.3　カメラの方向

カメラを向ける方向を指定する記述は、

```
look_at 注視点
```

と書きます。「注視点」のところに 3 次元ベクトルを値とする式を書くと、それを座標とする点の方向へカメラが向けられます。

これまでに紹介したシーンの中には、カメラの方向を指定する記述として、

```
look_at 0
```

というものが書かれていました。この記述の中の 0 は、<0，0，0> という意味ですので、この場合、カメラは原点の方向を向くことになります。

次のシーンは、第 1.9.2 項のシーンと、直方体の位置と大きさも、カメラの位置も同じです。しかし、カメラの方向を <-10，0，0> に向けています。

シーンの例　lookat.pov

```
camera {
    location <10, 5, -10>
    look_at <-10, 0, 0>
    angle 70
}

light_source { <20, 30, -20> color rgb 1.6 }

object {
    box { <-10, -1, -1>, <10, 1, 1> }
    pigment { color rgb <1, 1, 0> }
}

object {
    box { <-1, -10, -1>, <1, 10, 1> }
    pigment { color rgb <1, 0, 1> }
}

object {
    box { <-1, -1, -10>, <1, 1, 10> }
```

```
    pigment { color rgb <0, 1, 1> }
}
```

カメラの方向をいろいろと変えてレンダリングしてみてください。

1.9.4 カメラの視野の角度

カメラは、焦点距離を短くしたり長くしたりすることによって、視野の角度を広くしたり狭くしたりすることができます。それと同じように、POV-Ray のカメラも、視野の角度を指定することができるようになっています。

カメラの視野の角度を指定する記述は、

```
angle 角度
```

と書きます。「角度」のところにスカラーを値とする式を書くと、それがカメラの視野の角度になります（単位は度で、範囲は 0 から 180 までです）。

これまでに紹介したシーンの中には、カメラの視野の角度を指定する記述として、

```
angle 70
```

というものが書かれていました。これは、視野の角度が 70 度だということを意味しています。

次のシーンは、第 1.9.2 項のシーンと、直方体の位置と大きさも、カメラの位置と方向も同じです。しかし、カメラの視野の角度を 70 度から 30 度に狭くしています。

シーンの例 angle.pov

```
camera {
    location <10, 5, -10>
    look_at 0
    angle 30
}

light_source { <20, 30, -20> color rgb 1.6 }

object {
    box { <-10, -1, -1>, <10, 1, 1> }
    pigment { color rgb <1, 1, 0> }
}

object {
    box { <-1, -10, -1>, <1, 10, 1> }
    pigment { color rgb <1, 0, 1> }
}

object {
    box { <-1, -1, -10>, <1, 1, 10> }
    pigment { color rgb <0, 1, 1> }
}
```

カメラの視野の角度をいろいろと変更してレンダリングしてみましょう。

1.10 宣言

1.10.1 識別子

POV-Ray のシーンの中では、「識別子」(identifier) と呼ばれる名前をさまざまなものに与えることができます。

何かに識別子が与えられているとするとき、その識別子をシーンの中に書くと、その識別子が与えられている何かの記述をそこに書いたのと同じ意味になります。たとえば、58.3 という数値に Yokohaba という識別子が与えられているとするとき、Yokohaba という識別子を書くと、58.3 というリテラルをそこに書いたのと同じ意味になります。

識別子を使う目的は二つあります。目的のひとつは、意味の分かりやすい識別子を使うことによって、シーンを読みやすくすることです。

そしてもうひとつの目的は、同じものの記述を一箇所にまとめることです。同じものの記述がシーンの中に分散しているとすると、それらを修正する必要が生じた場合、それらの記述をひと

つひとつ修正していかないといけません。しかし、識別子を使ってそれらを書いておけば、ものに識別子を与える記述だけを修正すれば、自動的にすべてが修正されることになります。

1.10.2　識別子の作り方

POV-Ray で使うことのできる識別子を作るときは、次のような規則にしたがう必要があります。

- 使うことのできる文字は、英字、数字、またはアンダースコア (_)。
- 文字数は、1 文字から 40 文字まで。
- 先頭の文字は英字でなければならない。
- 予約語と同じものは識別子としては使えない。「予約語」(reserved word) というのは、用途があらかじめ予約されている単語のこと。たとえば、 `camera`、 `object`、 `sphere` など。

識別子として使うことのできるものの例としては、次のようなものがあります。

```
a  A  a8  namako  back_to_the_future
```

英字の大文字と小文字は区別されますので、たとえば、 `a` と `A` は異なる識別子だと認識されます。

最後の例のように、アンダースコアは、複数の単語から構成される識別子を作るときに、空白の代わりとして使うことができます。

識別子として使うことのできないものの例としては、次のようなものがあります。

`nam@ko`　使うことのできない文字を含んでいる。

`8a`　先頭の文字が数字。

`camera`　同じ予約語が存在する。

POV-Ray では、識別子を作るとき、予約語と重なってしまうことを防ぐため、先頭の文字を大文字にすることが推奨されています。

1.10.3　宣言の書き方

何かに識別子を与えることを、識別子を「宣言する」(declare) と言います。識別子を宣言したいときは、「宣言」(declaration) と呼ばれるものを書きます。

宣言というのは、

`#declare` 識別子 `=` 記述

という形の記述のことです。この形のものを書くことによって、「記述」のところに書かれた記述によってあらわされるものに対して、「識別子」のところに書かれた識別子を与えることができます。たとえば、

```
#declare Hako = object {
    box { <0, 0, 0>, <1, 1, 1> }
    pigment { color rgb 1 }
}
```

という宣言を書くことによって、 `Hako` という識別子を白い直方体に与えることができます。

1.10.4　識別子の使い方

シーンの中に識別子を書けば、基本的には、その識別子が与えられているものの記述をそこに書いたのと同じ意味になるわけですが、 `object` や `pigment` などの予約語を繰り返すことが必要になる場合もあります。

たとえば、 `Midori` という識別子がピグメントに与えられているとするとき、その識別子を使って物体にピグメントを与えるためには、

```
pigment { Midori }
```

という記述を書く必要があります。

次のシーンは、カメラ、光源、形状、ピグメント、物体に与えられた識別子を使って書かれています。

シーンの例　`declare.pov`

```
#declare Camera = camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

#declare Light = light_source { <-5, 5, -5> color rgb 1.6 }
#declare Sphere = sphere { 0, 1 }
#declare SkyBlue = pigment { color rgb <0.3, 0.6, 1> }

#declare Object = object {
    Sphere
    pigment { SkyBlue }
}

camera { Camera }
Light
Object
```

1.10.5 宣言の末尾のセミコロン

数値やベクトルに識別子を与えたいときは、

```
#declare 識別子 = 式;
```

という形の宣言を書きます。そうすると、「式」のところに書かれた式の値に対して、「識別子」のところに書かれた識別子が与えられます。この場合、宣言の末尾にはかならずセミコロン (;) を書かないといけませんので、注意が必要です。

たとえば、

```
#declare Takasa = 80;
```

という宣言を書くことによって、Takasa という識別子を 80 という数値に与えることができます。同じように、

```
#declare Houkou = <3, 2, 5>;
```

という宣言を書くことによって、Houkou という識別子を <3, 2, 5> というベクトルに与えることができます。

色に識別子を与えたいときは、

```
#declare 識別子 = color rgb 3次元ベクトル;
```

という形の宣言を書きます。この場合も、宣言の末尾にはセミコロンが必要です。

たとえば、

```
#declare Orange = color rgb <1, 0.5, 0>;
```

という宣言を書くことによって、Orange という識別子をオレンジ色に与えることができます。

次のシーンは、数値とベクトルと色に与えられた識別子を使って書かれています。

シーンの例 semicolon.pov

```
#declare CameraLocation = <0, 0, -3>;
#declare CameraLookAt = <0, 0, 0>;
#declare CameraAngle = 70;
#declare LightLocation = <-5, 5, -5>;
#declare LightColor = color rgb <1.6, 1.6, 1.6>;
#declare SphereLocation = <0, 0, 0>;
#declare SphereRadius = 1;
#declare Crimson = color rgb <0.86, 0.07, 0.23>;

camera {
    location CameraLocation
    look_at CameraLookAt
    angle CameraAngle
}

light_source { LightLocation LightColor }
```

```
object {
    sphere { SphereLocation, SphereRadius }
    pigment { Crimson }
}
```

1.10.6　基本ベクトル

成分として 1 を 1 個だけ含んでいて、残りの成分がすべて 0 であるようなベクトルは、「基本ベクトル」(elementary vector) と呼ばれます。

3 次元の基本ベクトルに対しては、POV-Ray の内部で、

```
x = <1, 0, 0>
y = <0, 1, 0>
z = <0, 0, 1>
```

というように、x、y、z という識別子が与えられています。

ちなみに、x、y、z のような、POV-Ray の内部で定義されている、特定のものに与えられた識別子は、「組み込み識別子」(built-in identifier) と呼ばれます。

第 2 章　変形

2.1　変形の基礎

2.1.1　変形とは何か

POV-Ray では、物体に対して、次のような操作をすることができます。

移動　位置の変更。

拡大　大きさの変更。

回転　方向の変更。

これらの操作は、総称して「変形」(transformation) と呼ばれます。

2.1.2　変形の必要性

同一の形状を持つ物体を何個も作る場合には、その形状に識別子を与えておくと、とても便利です。しかし、形状というのは、位置と大きさと方向についての情報も含んでいますので、そのままだと、物体を何個作っても、それらはすべて同じ位置、同じ大きさ、同じ方向を持つことになります。

ですから、識別子が与えられたひとつの形状から複数の物体を作る場合、それらの移動、拡大、回転という操作が必要になります。

また、プリミティブの中には、位置や大きさが固定されているものもあります。そのようなプリミティブを使って任意の位置に任意の大きさで物体を作るためには、その物体に対する変形が必要になります。

2.2　移動

2.2.1　移動の基礎

物体の位置を変更するという操作は、「移動」(translation) と呼ばれます。

物体を移動させるためには、それをどこへ移動させるのかという移動先を指定する必要があります。移動先は、x、y、z のそれぞれの座標軸のプラスの方向にどれだけ物体を移動させるか、ということをあらわす 3 次元ベクトルによって指定されます。

2.2.2　移動の記述

物体を移動させたいときは、物体を作る記述の中に、

```
translate 移動先
```

という記述を書きます。「移動先」のところに書くのは、3 次元ベクトルを値とする式です。そうすると、その式の値を移動先とする移動が実行されます。たとえば、

```
object {
    Katachi
    pigment { Iro }
    translate <20, 10, -30>
}
```

という記述で物体を作ったとすると、この物体は、本来の位置に作られたのち、x 軸の方向に 20、y 軸の方向に 10、z 軸の方向に -30 だけ移動することになります。

物体を作るときは、このように、本来の位置から移動させるのが普通ですので、形状を作って識別子を与えるときは、どのように移動させればいいかということが分かりやすくなるように、なるべく、原点を基準とする位置にその形状を作るといいでしょう。

次のシーンは、二つの直方体を作っています。ひとつは原点を中心とする位置に置かれていて、もうひとつは、その位置から <3, 4, 5> へ移動した位置に置かれています。

シーンの例 translate.pov

```
#declare Box = box { -1, 1 }

camera {
    location <10, 5, -10>
    look_at 0
    angle 70
}

light_source { <20, 30, -20> color rgb 1.6 }

object {
    Box
    pigment { color rgb 1 }
}

object {
    Box
    pigment { color rgb <0, 1, 1> }
    translate <3, 4, 5>
}
```

移動先をいろいろと変更してレンダリングしてみましょう。

2.3 拡大

2.3.1 拡大の基礎

物体の大きさを変更するという操作は、「拡大」(scaling) と呼ばれます。

物体を拡大するためには、それをどの方向へどれだけ拡大するのかという拡大率を指定する必要があります。拡大率は、x、y、z のそれぞれの座標軸の方向にどれだけ物体を拡大するか、ということをあらわす 3 次元ベクトルによって指定されます。拡大率を構成する数値を 1 よりも小さくすると、物体は、その軸の方向に縮小されることになります。

2.3.2 拡大の記述

物体を拡大したいときは、物体を作る記述の中に、

```
scale 拡大率
```

という記述を書きます。「拡大率」のところに書くのは、3 次元ベクトルを値とする式です。そうすると、その式の値を拡大率とする拡大が実行されます。たとえば、

```
object {
    Katachi
    pigment { color Iro }
    scale <2, 0.5, 3>
}
```

という記述で物体を作ったとすると、この物体は、本来の大きさで作られたのち、x 軸の方向に 2 倍、y 軸の方向に 0.5 倍、z 軸の方向に 3 倍だけ拡大されることになります。

拡大は、常に原点を中心にして実行されます。ですから、原点から離れた位置にある物体を拡大すると、その物体の大きさだけではなくて位置も変化する、という点に注意する必要があります。

次のシーンは、二つの直方体を作っています。ひとつは本来の大きさのままで、もうひとつは、x 軸の方向に 5 倍、y 軸の方向に 0.2 倍、z 軸の方向に 0.5 倍だけ拡大しています。

シーンの例　scale.pov

```
#declare Box = box { -1, 1 }

camera {
    location <10, 5, -10>
    look_at 0
    angle 70
}

light_source { <20, 30, -20> color rgb 1.6 }

object {
    Box
    pigment { color rgb 1 }
}

object {
    Box
    pigment { color rgb <0, 1, 1> }
    scale <5, 0.2, 0.5>
}
```

拡大率をいろいろと変更してレンダリングしてみましょう。

2.3.3　変形の順序

いくつかの変形を組み合わせる場合は、それらの変形をどのような順序で実行するかということに注意を払う必要があります。なぜなら、変形を実行する順序が変わると、その結果も変わるからです。

変形は、記述が書かれている順序のとおりに実行されます。たとえば、

```
scale 2
translate 2
```

と書いたとすると、これらの変形は、拡大してから移動させるという順序で実行されます。

次のシーンは、3 個の直方体を作っています。拡大も移動もしていないもの、拡大してから移動させたもの、移動させてから拡大したものです。

シーンの例　order.pov

```
#declare Box = box { -1, 1 }

camera {
    location <10, 5, -10>
    look_at 0
    angle 70
}

light_source { <20, 30, -20> color rgb 1.6 }

object {
    Box
    pigment { color rgb 1 }
}

object {
    Box
    pigment { color rgb <0, 1, 1> }
    scale 2
```

```
    translate 2
}

object {
    Box
    pigment { color rgb <1, 1, 0> }
    translate 2
    scale 2
}
```

2.3.4 プリミティブの変形

変形の記述は、物体を作る記述の中だけではなくて、プリミティブの記述の中にも書くことができます。

次のシーンは、プリミティブの記述の中に scale を書くことによって、回転楕円体 (spheroid) を作っています。

シーンの例 rugby.pov

```
#declare RugbyBall = sphere {
    0, 1
    scale <1.8, 1, 1>
}

camera {
    location <0, 0, -4>
    look_at 0
    angle 70
}

light_source { <-10, 5, -10> color rgb 1.6 }

object {
    RugbyBall
    pigment { color rgb <0.6, 0.2, 0.1> }
}
```

2.4 回転

2.4.1 回転の基礎

物体の大きさを変更するという操作は、「回転」(rotation) と呼ばれます。

物体を回転させるためには、どの軸でどれだけそれを回転させるのかという回転角を指定する必要があります。回転角は、x 軸、y 軸、z 軸のそれぞれを回転軸にしてどれだけ物体を回転させるか、ということをあらわす 3 次元ベクトルによって指定されます。

たとえば、<20，-50，30> という回転角は、まず x 軸を回転軸として 20 度、次に y 軸を回転軸として -50 度、最後に z 軸を回転軸として 30 度だけ回転させる、という意味になります。

回転角を構成する数値は、度 (degree) を単位とする角度です。角度は、プラスの場合とマイナスの場合とでは回転の方向が逆になります。プラスの回転方向は、左手で、回転軸のプラスの方向を親指が向くようにして回転軸をつかんだときに、親指以外の指が向く方向と一致します。

特定の軸のみを回転軸として回転させる場合は、回転角を、角度と基本ベクトルとの積の形で書くことも可能です。たとえば、<30，0，0> という回転角は、30*x と書くこともできます。

2.4.2 回転の記述

物体を回転させたいときは、物体を作る記述の中に、

rotate 回転角

という記述を書きます。「回転角」というところに書くのは、3 次元ベクトルを値とする式です。そうすると、その式の値を回転角とする回転が実行されます。たとえば、

```
    object {
        Katachi
```

```
        pigment { color Iro }
        rotate 120*z
    }
```

という記述で物体を作ったとすると、この物体は、本来の方向で作られたのち、z 軸を回転軸として 120 度だけ回転することになります。

次のシーンは、二つの直方体を作っています。ひとつは本来の向きのままで、もうひとつは、z 軸を回転軸として 120 度だけ回転させています。

シーンの例　rotate.pov

```
#declare Box = box { 0, <1, 10, 1> }

camera {
    location <0, 0, -26>
    look_at 0
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

object {
    Box
    pigment { color rgb 1 }
}

object {
    Box
    pigment { color rgb <0, 1, 1> }
    rotate 120*z
}
```

回転角をいろいろと変更してレンダリングしてみましょう。

2.4.3　回転の順序

二つ以上の異なる軸を回転軸にして物体を回転させる場合、回転させる軸の順序が異なると、異なる結果が得られます。たとえば、x 軸で 30 度回転させてから y 軸で 60 度回転させた場合と、y 軸で 60 度回転させてから x 軸で 30 度回転させた場合とでは、得られる結果は異なります。

次のシーンは、3 個の直方体を作っています。本来の向きのままのもの、y 軸で 90 度回転させてから z 軸で 90 度回転させたもの、z 軸で 90 度回転させてから y 軸で 90 度回転させたものです。

シーンの例　rotate2.pov

```
#declare Box = box { 0, <1, 10, 5> }

camera {
    location <10, 15, -10>
    look_at <0, 5, 0>
    angle 70
}

light_source { <20, 30, -20> color rgb 1.6 }

object {
    Box
    pigment { color rgb 1 }
}

object {
    Box
    pigment { color rgb <0, 1, 1> }
    rotate 90*y
    rotate 90*z
}

object {
```

```
    Box
    pigment { color rgb <1, 1, 0> }
    rotate 90*z
    rotate 90*y
}
```

第 3 章　基本的なプリミティブ

3.1　円柱

3.1.1　この章について

第 1.7.2 項で説明したように、POV-Ray の内部では、「プリミティブ」(primitive) と呼ばれるいくつかの基本的な形状が、最初から定義されています。

プリミティブとしては、これまでに sphere と box を紹介しましたが、それら以外にもさまざまなプリミティブがあります。そこで、この章では、プリミティブのうちで基本的なものをいくつか紹介したいと思います。

3.1.2　円柱の基礎

円を、その円が含まれる平面に対して垂直な直線に沿って移動させることによってできる形状、たとえばジュースやコーヒーの缶のような形状は、「円柱」(cylinder) と呼ばれます。

円柱の両端にあるそれぞれの円は、その円柱の「底面」(base) と呼ばれます。

POV-Ray では、cylinder というプリミティブを使うことによって、円柱を作ることができます。

3.1.3　円柱の記述

cylinder を使って円柱を作る記述は、

```
cylinder { 中心1 , 中心2 , 半径 }
```

と書きます。「中心$_1$」と「中心$_2$」のそれぞれのところには 3 次元ベクトルを値とする式を書いて、「半径」のところにはスカラーを値とする式を書きます。そうすると、「中心$_1$」と「中心$_2$」のそれぞれのところに書かれた式の値を座標とする点に中心があって、「半径」のところに書かれた式の値が半径であるような円を底面とする円柱が作られます。たとえば、

```
    cylinder { <2, 0, 3>, <2, 5, 3>, 4 }
```

という記述は、<2, 0, 3> と <2, 5, 3> のそれぞれを中心の座標とする半径が 4 の円を底面とする円柱を作る、という意味になります。

次のシーンは、cylinder を使って円柱を作っています。

シーンの例　cylinder.pov

```
camera {
    location <0, 4, -6>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    cylinder { <0, -2, 0>, <0, 2, 0>, 1 }
    pigment { color rgb 1 }
}
```

底面の中心の座標や底面の半径をいろいろと変更してレンダリングしてみましょう。

3.2　円錐

3.2.1　円錐の基礎

円柱というのは、半径を変化させずに円を移動させてできる形状ですが、一定の速度で円の半径を変化させながら移動させると、プリンのような形状ができます。そのような形状は、「円錐台」(frustum) と呼ばれます。

円錐台の両端にある円も、円柱の場合と同じように、「底面」(base) と呼ばれます。

ソフトクリームの容器のような形状、つまり一方の底面の半径が 0 になっている円錐台は、「円錐」(cone) と呼ばれます。

POV-Ray では、cone というプリミティブを使うことによって、円錐台を作ることができます。

3.2.2　円錐の記述

cone を使って円錐台を作る記述は、

cone { 中心$_1$, 半径$_1$, 中心$_2$, 半径$_2$ }

と書きます。「中心$_1$」と「中心$_2$」のそれぞれのところには 3 次元ベクトルを値とする式を書いて、「半径 $_1$」と「半径 $_2$」のそれぞれのところにはスカラーを値とする式を書きます。そうすると、「中心$_1$」のところに書かれた式の値を座標とする点に中心があって、「半径 $_1$」のところに書かれた式の値が半径であるような円と、「中心$_2$」のところに書かれた式の値を座標とする点に中心があって、「半径 $_2$」のところに書かれた式の値が半径であるような円錐台が作られます。たとえば、

```
cone { <2, 0, 3>, 6, <2, 5, 3>, 4 }
```

という記述は、中心の座標が <2, 0, 3> で半径が 6 の円と、中心の座標が <2, 5, 3> で半径が 4 の円を底面とする円柱を作る、という意味になります。

「半径$_1$」または「半径$_2$」のどちらか一方を 0 にすると、その円錐台は円錐になります。

次のシーンは、cone を使って円錐台を作っています。

シーンの例　cone.pov

```
camera {
    location <0, 4, -10>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    cone { <0, -2, 0>, 3, <0, 2, 0>, 2 }
    pigment { color rgb 1 }
}
```

底面の中心の座標や底面の半径をいろいろと変更してレンダリングしてみましょう。

3.3　トーラス

3.3.1　トーラスの基礎

ドーナツや浮き輪のような、円管をリングにした形状は、「トーラス」(torus) と呼ばれます。

POV-Ray では、torus というプリミティブを使うことによって、トーラスを作ることができます。

3.3.2　トーラスの記述

torus を使ってトーラスを作る記述は、

torus { 半径$_1$, 半径$_2$ }

と書きます。「半径 $_1$」と「半径 $_2$」のそれぞれのところにはスカラーを値とする式を書きます。そうすると、「半径 $_1$」のところに書かれた式の値が全体の半径で、「半径$_2$」のところに書かれた

式の値が円管の断面の半径であるようなトーラスが作られます。たとえば、

```
torus { 12, 3 }
```

という記述は、全体の半径が 12 で、円管の断面の半径が 3 であるようなトーラスを作る、という意味になります。

次のシーンは、torus を使ってトーラスを作っています。

シーンの例 torus.pov

```
camera {
    location <0, 4, -8>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    torus { 3, 1 }
    pigment { color rgb 1 }
}
```

全体の背景や断面の半径をいろいろと変更してレンダリングしてみましょう。

3.3.3 トーラスの位置と向き

torus というプリミティブは、それ自体の位置と向きが固定されていますので、位置を変更したり向きを変更したりするためには、変形を使う必要があります。

次のシーンは、3 個のトーラスを作っています。位置と向きがそのままのもの、位置を変更したもの、向きと位置を変更したものです。

シーンの例 torus2.pov

```
#declare Torus = torus { 3, 1 }

camera {
    location <0, 8, -12>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    Torus
    pigment { color rgb 1 }
}

object {
    Torus
    pigment { color rgb <0, 1, 1> }
    translate <0, 0, 16>
}

object {
    Torus
    pigment { color rgb <1, 1, 0> }
    rotate 90*x
    translate <3, 0, 0>
}
```

位置や向きをいろいろと変更してレンダリングしてみましょう。

3.4　平面

3.4.1　平面の基礎

この節では、plane というプリミティブについて説明したいと思います。

plane は、「平面」(plane) と呼ばれる形状を作るプリミティブです。ただし、平面と言っても、数学用語の「平面」とは意味が少し違います。数学では、「平面」という言葉は 2 次元空間という意味で使われますが、plane によって作られる平面は、3 次元の形状です。

plane によって作られるのは、数学用語の意味での平面で 3 次元空間を二つに分断して、その一方の側だけを物質で埋め尽くした形状です。

平面の向きは、法線ベクトルによって指定されます。「法線ベクトル」(normal vector) というのは、物体の表面を構成している平面に対して垂直に立っていて、その向きが物体の外側を示しているもののことです。

平面の位置は、原点からの距離によって指定されます。プラスの距離を指定すると、法線ベクトルとは逆の方向にその距離だけ離れた位置に平面の表面が置かれます。

3.4.2　平面の記述

plane を使って平面を作る記述は、

```
plane { 法線ベクトル , 距離 }
```

と書きます。「法線ベクトル」のところには 3 次元ベクトルを値とする式を書いて、「距離」のところにはスカラーを値とする式を書きます。そうすると、「法線ベクトル」のところに書かれた式の値が法線ベクトルで、「距離」のところに書かれた式の値が原点からの距離であるような平面が作られます。たとえば、

```
plane { <0, 1, 2>, 3 }
```

という記述は、法線ベクトルが <0, 1, 2> で、原点からの距離が 3 であるような平面を作る、という意味になります。

「法線ベクトル」のところに y（つまり <0, 1, 0>）と書くことによって、境界面が水平な平面（つまり地面または床）を作ることができます。

次のシーンは、plane を使って平面を作っています。

シーンの例　plane.pov

```
camera {
    location <0, 1, -1>
    look_at <0, 1, 0>
    angle 70
}

light_source { <0, 50, 0> color rgb 1.6 }

object {
    plane { y, 0 }
    pigment { color rgb 1 }
}
```

法線ベクトルや原点からの距離をいろいろと変更してレンダリングしてみましょう。

3.5　テキスト

3.5.1　テキストの基礎

この節では、text というプリミティブについて説明したいと思います。

text は、「テキスト」(text) と呼ばれる形状を作るプリミティブです。

「テキスト」という言葉は、もともとは文字が並んでできているデータ、つまり文字列という意味ですが、ここで「テキスト」と呼んでいるのは、フォントによって作られた 2 次元の形状に対して厚さを加えることによって、それを 3 次元化した形状のことです。

POV-Ray は、任意の TrueType フォントを使ってテキストを作ることができます。

このチュートリアルでは、「文字列」という言葉を、文字が並んでできているデータという意味で使って、「テキスト」という言葉を、文字列を 3 次元化した形状という意味で使うことにします。

3.5.2 文字列リテラル

得られるデータが文字列であるようなリテラルは、「文字列リテラル」(string literal) と呼ばれます。

文字列リテラルは、二重引用符 (") で文字列を囲んだものです。文字列リテラルを評価すると、二重引用符で囲まれた文字列が値として得られます。たとえば、 "namako" という文字列リテラルを評価すると、 namako という文字列が値として得られます。

二重引用符を含む文字列を値とする文字列リテラルを書くときは、その二重引用符の直前にバックスラッシュ(\) を書く必要があります。たとえば、 ab"cd という文字列を値とする文字列リテラルは、 "ab\"cd" と書く必要があります。

バックスラッシュを含む文字列を値とする文字列リテラルを書くときは、バックスラッシュを二重に書く必要があります。たとえば、 ab\cd という文字列を値とする文字列リテラルは、 "ab\\cd" と書く必要があります。

3.5.3 テキストの記述

text を使ってテキストを作る記述は、

text { ttf パス名 , 文字列 , 厚さ , 文字間隔 }

と書きます。「パス名」と「文字列」のそれぞれのところには文字列を値とする式を書いて、「厚さ」のところにはスカラーを値とする式を書いて、「文字間隔」のところには 3 次元ベクトルを値とする式を書きます。そうすると、「パス名」のところに書かれた式の値をパス名とするファイルに格納されている TrueType フォントを使って、「厚さ」のところに書かれた式の値を厚さとする、「文字列」のところに書かれた式の値を 3 次元化したテキストが作られます。「文字間隔」のところに書かれた式の値は、文字の間隔を指定します。ここに 0 （つまり <0, 0, 0>）と書くと、ノーマルな間隔になります。

たとえば、

```
text { ttf "timrom.ttf", "umiushi", 3, 0 }
```

という記述は、 timrim.ttf というファイルに格納されている TrueType フォントを使って、 umiushi という文字列から、厚さが 3 で文字間隔がノーマルなテキストを作る、という意味になります。

テキストの先頭の文字は、その左下の手前が原点になる位置に置かれます。そして、それ以降の文字は、文字間隔が 0 の場合、x 軸の方向に並べられます。

次のシーンは、 text を使ってテキストを作っています。

シーンの例 text.pov

```
camera {
    location <0, 2, -5>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    text { ttf "timrom.ttf", "namako", 1, 0 }
    pigment { color rgb 1 }
}
```

文字列や厚さをいろいろと変更してレンダリングしてみましょう。

3.5.4 テキストの位置と向きと大きさ

text というプリミティブは、それ自体の位置と向きと大きさが固定されていますので、位置を変更したり向きを変更したり大きさを変更したりするためには、変形を使う必要があります。

次のシーンは、4 個のテキストを作っています。位置と向きと大きさがそのままのもの、位置を変更したもの、向きを変更したもの、大きさと位置を変更したものです。

シーンの例　text2.pov

```
#declare Text = text { ttf "timrom.ttf", "m", 1, 0 }

camera {
    location <0, 2, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    Text
    pigment { color rgb 1 }
}

object {
    Text
    pigment { color rgb <0, 1, 1> }
    translate <1, 0, 0>
}

object {
    Text
    pigment { color rgb <1, 1, 0> }
    rotate -135*z
}

object {
    Text
    pigment { color rgb <1, 0, 1> }
    scale 2
    translate <-2, 0, 0>
}
```

位置や向きや大きさをいろいろと変更してレンダリングしてみましょう。

3.5.5　文字間隔

テキストを作る記述の中には、文字間隔を指定するベクトルを書くわけですが、先ほども説明したとおり、0 と書けば、普通の間隔になります。

0 以外のベクトルを文字間隔として記述することによって、x 軸方向と y 軸方向の間隔を指定することができます。

次のシーンは、二つのテキストを作っています。ひとつは文字間隔が 0.7*x で、もうひとつは文字間隔が 0.3*y です。

シーンの例　offset.pov

```
camera {
    location <3, 3, -5>
    look_at <3, 1, 0>
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    text { ttf "timrom.ttf", "hitode", 1, 0.7*x }
    pigment { color rgb 1 }
}

object {
    text { ttf "timrom.ttf", "kamenote", 1, 0.3*y }
    pigment { color rgb 1 }
```

```
    translate <0, 0, 2>
}
```

3.5.6 同梱されているフォント

POV-Ray の公式サイトからダウンロードした POV-Ray のファイルには、

`timrom.ttf` `cyrvetic.ttf` `crystal.ttf`

という三つの TrueType フォントが同梱されています。

次のシーンは、`timrom.ttf`、`cyrvetic.ttf`、`crystal.ttf` のそれぞれを使って、3 個のテキストを作っています。

シーンの例 `font.pov`

```
camera {
    location <1.7, 0, -3>
    look_at <1.7, 1, 0>
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    text { ttf "timrom.ttf", "timrom", 1, 0 }
    pigment { color rgb 1 }
    translate <0, 2, 0>
}

object {
    text { ttf "cyrvetic.ttf", "cyrvetic", 1, 0 }
    pigment { color rgb 1 }
    translate <0, 1, 0>
}

object {
    text { ttf "crystal.ttf", "crystal", 1, 0 }
    pigment { color rgb 1 }
}
```

3.6 ブロブ

3.6.1 ブロブの基礎

互いに融合しようとしているかのように変形した、いくつかの球または円柱から構成される形状は、「ブロブ」(blob) または「メタボール」(metaball) と呼ばれます。ブロブは、動物の体のような、なめらかで不規則な形状を作りたいときなどに使われる形状です。

ブロブを構成している円または円柱は、「ブロブ部品」(blob component) と呼ばれます。個々のブロブ部品は、本来、ある種の力が作用している力場として存在しています。その力は、ブロブ部品の中心で最も強くなっていて、そこから一定の距離だけ離れた場所で 0 になります。ブロブというのは、力が特定の強さになっている点をつないだ面で空間を内側と外側に分割して、内側に物質を詰め込むことによって実体化した物体のことです。内側と外側の境界面を決定する力の強さは、「閾値」(threshold) と呼ばれます。閾値は、日本語では「しきい値」と呼ばれることもあります。

ブロブ部品には、中心（つまり力が最も強いところ）が点であるものと線分であるものの 2 種類があります。中心が点であるブロブ部品は実体化すると球になって、中心が線分であるブロブ部品は実体化すると円柱になります。

いくつかの山があって、ふもとからの高さが指定されたとするとき、その高さの等高線によって囲まれた 2 次元の形状を作る、ということができます。ブロブというのは、それと同じ操作を 4 次元の山に対して実行することによってできた 3 次元の等高線のことだと考えることもできます。

POV-Ray では、`blob` というプリミティブを使うことによって、ブロブを作ることができます。

3.6.2　ブロブの記述

blob を使ってブロブを作る記述は、

```
blob {
    threshold 閾値
    ブロブ部品
        ⋮
}
```

と書きます。この中の「閾値」というところには、ブロブの内側と外側の境界面を決定する閾値を指定します。そして、「ブロブ部品」というところには、ブロブ部品を作るための sphere または cylinder の記述を書きます。ただし、 sphere と cylinder の書き方は、本来の書き方とは少し違います。 sphere の記述は、

```
sphere { 中心の座標 , 半径 , 強さ }
```

と書きます。同じように、 cylinder の記述は、

```
cylinder { 中心1 , 中心2 , 半径 , 強さ }
```

と書きます。これらの記述の中の「半径」のところには、力の強さが 0 になる、中心からの距離を書きます。そして、「強さ」というところには、ブロブ部品の中心での力の強さを書きます。たとえば、

```
blob {
    threshold 0.5
    sphere { <-1, 0, 0>, 1.5, 1 }
    sphere { <1, 0, 0>, 1.5, 1 }
}
```

という記述は、中心での力の強さが 1 で、中心から 1.5 だけ離れたところで力が 0 になる二つのブロブ部品から構成される、力の強さが 0.5 のところをつないでできる境界面で囲まれたブロブを作ります。

次のシーンは、二つの球から構成されるブロブを作っています。

シーンの例　blob.pov

```
camera {
    location <0, 0, -4>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    blob {
        threshold 0.5
        sphere { <-1, 0, 0>, 1.5, 1 }
        sphere { <1, 0, 0>, 1.5, 1 }
    }
    pigment { color rgb 1 }
}
```

閾値や強さをいろいろと変更してレンダリングしてみましょう。

3.6.3　円柱を含むブロブ

ブロブの記述の中に sphere の記述を書いた場合は、中心が点であるようなブロブ部品ができるわけですが、それに対して、 cylinder の記述を書いた場合は、中心が線分であるようなブロブ部品ができます。

次のシーンは、円柱と球から構成されるブロブを作っています。

シーンの例　blob2.pov

```
camera {
    location <0, 0, -5>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    blob {
        threshold 0.5
        sphere { <-1, 0, 0>, 1.5, 1 }
        cylinder { <1, -1, 0>, <1, 1, 0>, 1.5, 1 }
    }
    pigment { color rgb 1 }
}
```

閾値や強さをいろいろと変更してレンダリングしてみましょう。

3.6.4 ブロブとテクスチャー

ブロブを構成するそれぞれの球や円柱に対しては、異なるテクスチャーを指定することもできます。

次のシーンは、異なる色を持つ球から構成されるブロブを作っています。

シーンの例 blob3.pov

```
camera {
    location <0, 0, -4>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    blob {
        threshold 0.5
        sphere {
            <-1, 0, 0>, 1.5, 1
            pigment { color rgb <1, 1, 0> }
        }
        sphere {
            <1, 0, 0>, 1.5, 1
            pigment { color rgb <0, 1, 1> }
        }
    }
}
```

3.6.5 マイナスの強さを持つブロブ部品

ブロブ部品の中心での力の強さは、マイナスにすることも可能です（閾値は、プラスでないといけません）。中心での力の強さがマイナスであるようなブロブ部品の場合、力は、中心から離れるほど強くなっていって、半径として指定された距離で 0 になります。

マイナスの強さを持つブロブ部品を使うことによって、凹みを持つブロブを作ることができます。

次のシーンは、プラスの強さを持つ球とマイナスの強さを持つ球から構成されるブロブを作っています。

シーンの例 blob4.pov

```
camera {
    location <5, 0, -3>
    look_at 0
    angle 70
}
```

```
light_source { <5, 3, -5> color rgb 1.6 }

object {
    blob {
        threshold 0.5
        sphere { <0, 0, 0>, 4, 1 }
        sphere { <2, 0, 0>, 1.5, -1 }
    }
    pigment { color rgb 1 }
}
```

閾値や強さをいろいろと変更してレンダリングしてみましょう。

3.7　プリズム

3.7.1　プリズムの基礎

閉じた平面図形を、それが含まれる平面に対して垂直な直線に沿って移動させたときに、その軌跡として残る形状は、「プリズム」(prism) と呼ばれます。プリズムは、日本語では、「掃引体」と呼ばれることもあります。

プリズムの両端にあるそれぞれの平面図形は、そのプリズムの「底面」(base) と呼ばれます。

円柱や角柱は、特殊なプリズムです。円柱というのは底面が円であるようなプリズムのことで、角柱というのは底面が多角形であるようなプリズムのことです。

POV-Ray では、`prism` というプリミティブを使うことによって、プリズムを作ることができます。

`prism` というプリミティブは、xz 平面の上に作られた平面図形を y 軸に沿って移動させることによって、プリズムを作ります。平面図形は、何個かの点を直線や曲線でつないでいくことによって作ります。

3.7.2　プリズムの記述

`prism` を使って、直線によって構成される平面図形を移動させることによってプリズムを作る記述は、

```
prism {
    y 座標1 , y 座標2 , 点の個数 ,
    点 1 , 点 2 , ··· , 点 n
}
```

と書きます。この中の「y 座標$_1$」と「y 座標$_2$」のところには、平面図形を移動させる範囲を示す y 座標を書きます。「点の個数」というところには、平面図形を構成する点の個数を書きます。そして、点 $_1$、点 $_2$、……、点 $_n$ というところには、平面図形を作るための点の位置を示す 2 次元ベクトルを書きます。それらの 2 次元ベクトルは、1 個目の成分が x 座標、2 個目の成分が z 座標だと解釈されて、xz 平面の上に平面図形が作られることになります。

点の個数と座標については、注意しないといけないことがあります。それは、最初の点と同じものを最後の点として書くことによって、図形を閉じる必要がある、ということです。したがって、点の個数は、実際の個数よりも 1 だけ大きくなります。たとえば、三角形を底面とするプリズム（つまり三角柱）を作る場合は、点の個数として 4 を指定して、

```
<0, 1>, <1, 0>, <-1, 0>, <0, 1>
```

というように、最初と同じ点を最後に書く必要がある、ということです。

ただし、図形を閉じなかったとしても、

```
Parse Warning: Linear prism not closed. Closing it.
```

という警告が表示されるだけで、レンダリングは実行されます。

次のシーンは、`prism` を使って五角柱を作っています。

シーンの例　`prism.pov`

```
camera {
    location <0, 5, -5>
```

```
    look_at <0, 0.5, 0>
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    prism {
        -2, 2, 6,
        <0, 3>, <2, 1>, <1, -1>, <-1, -1>, <-2, 1>, <0, 3>
    }
    pigment { color rgb 1 }
}
```

3.7.3 穴の開いたプリズム

プリズムを作るための平面図形は、一度閉じたのち、別の点から再び作り始めることも可能です。平面図形の内部に別の平面図形を作ると、プリズムに、内側の平面図形を形とする穴が開くことになります。

次のシーンは、穴の開いた三角柱を作ります。

シーンの例 prismwithhole.pov

```
camera {
    location <0, 5, -10>
    look_at 0
    angle 70
}

light_source { <-2, 5, -5> color rgb 1.6 }

object {
    prism {
        0, 1, 8,
        <0, 8>, <4, -4>, <-4, -4>, <0, 8>,
        <0, 2>, <2, -3>, <-2, -3>, <0, 2>
    }
    pigment { color rgb 1 }
}
```

3.7.4 曲線によるプリズム

プリズムの底面となる平面図形は、曲線を使って作ることもできます。

プリズムの底面を作るときに使うことのできる曲線としては、次の 3 種類のものがあります。

- 2 次スプライン曲線 (quadratic spline curve)
- 3 次スプライン曲線 (cubic spline curve)
- ベジェ曲線 (Bézier curve)

ここでは、ベジェ曲線を使ってプリズムの底面を作る方法について説明することにしたいと思います。

「ベジェ曲線」(Bézier curve) は、4 個の制御点によって決定される曲線です。それぞれの制御点を、制御点 1、制御点 2、制御点 3、制御点 4 と呼ぶことにしましょう。ベジェ曲線は、制御点 1 から制御点 2 に向かって出発して、少しずつ方向を変えながら、制御点 3 と制御点 4 とをつなぐ直線に接する形で、制御点 4 に到達します。

プリズムの記述を、

```
prism {
    bezier_spline
    ...
}
```

と書くと、その中に書かれた個々の 2 次元ベクトルがベジェ曲線を作るための制御点の座標だと解釈されて、ベジェ曲線による平面図形を底面とするプリズムが作られます。

ベジェ曲線を使ってプリズムの底面を作る場合、連続する 2 本のベジェ曲線は、前のベジェ曲線の制御点 4 と後ろのベジェ曲線の制御点 1 とが同じでないといけません。

また、直線を使う場合と同じように、最初の点と同じものを最後の点として書くことによって、図形を閉じないといけません。

次のシーンは、ベジェ曲線を使って、ハート形の底面を持つプリズムを作っています。

シーンの例　bezier.pov

```
camera {
    location <0, 4, -8>
    look_at 0
    angle 70
}

light_source { <-2, 5, -5> color rgb 1.6 }

object {
    prism {
        bezier_spline
        0, 1, 8,
        <0, -4>, <-12, 8>, <-2, 20>, <0, 6>,
        <0, 6>, <2, 20>, <12, 8>, <0, -4>
    }
    pigment { color rgb 1 }
}
```

3.7.5　ベジェ曲線の滑らかな接続

ベジェ曲線とベジェ曲線とを接続したとき、その全体は、必ずしも滑らかな曲線になっているとは限りません。もしも、2 本のベジェ曲線を滑らかに接続したいならば、そうなるように制御点を指定する必要があります。

2 本のベジェ曲線を滑らかに接続したい場合は、1 本目の制御点 3、1 本目の制御点 4（2 本目の制御点 1 と同じ）、2 本目の制御点 2、という 3 個の点が、1 本の直線の上に乗っている必要があります。

次のシーンは、2 本のベジェ曲線を滑らかに接続することによって作られた卵形を底面とするプリズムを作っています。

シーンの例　egg.pov

```
camera {
    location <0, 10, -12>
    look_at 0
    angle 70
}

light_source { <-2, 5, -10> color rgb 1.6 }

object {
    prism {
        bezier_spline
        0, 1, 8,
        <0, 5>, <10, 3>, <10, -3>, <0, -5>,
        <0, -5>, <-10, -7>, <-10, 7>, <0, 5>
    }
    pigment { color rgb 1 }
}
```

3.8　回転体

3.8.1　回転体の基礎

閉じた平面図形を回転させたときに軌跡として残る図形は、「回転体」(lathe) と呼ばれます（lathe という英単語（読み方は「レイズ」）は、旋盤という意味です）。

POV-Ray では、lathe というプリミティブを使うことによって、回転体を作ることができます。

lathe というプリミティブは、xy 平面の上に作られた平面図形を、y 軸を回転軸として 360 度回転させることによって、回転体を作ります。

回転体を作るために回転させる平面図形は、プリズムの場合と同じように、何個かの点を直線や曲線でつないでいくことによって作ります。ただし、その平面図形は、xy 平面のどこにでも作ることができるというわけではなくて、x 座標が 0 またはプラスになる位置にしか作れません。

3.8.2 回転体の記述

lathe を使って、直線によって構成される平面図形を回転させることによってプリズムを作る記述は、

```
lathe {
    点の個数,
    点_1, 点_2, ···, 点_n
}
```

と書きます。この中の「点の個数」というところには、平面図形を構成する点の個数を書きます。そして、点 $_1$、点 $_2$、……、点 $_n$ というところには、平面図形を作るための点の位置を示す 2 次元ベクトルを書きます。それらの 2 次元ベクトルは、1 個目の成分が x 座標、2 個目の成分が y 座標だと解釈されて、xy 平面の上に平面図形が作られることになります。

プリズムの場合と同じように、平面図形を閉じないといけませんので、最初の点と同じものを最後の点として書く必要があります。

次のシーンは、直線によって構成される平面図形を回転させた回転体を作っています。

シーンの例 lathe.pov

```
camera {
    location <0, 6, -16>
    look_at <0, -2, 0>
    angle 70
}

light_source { <-2, 5, -10> color rgb 1.6 }

object {
    lathe {
        6,
        <5, 3>, <7, 3>, <6, 0>, <7, -3>, <5, -3>, <5, 3>
    }
    pigment { color rgb 1 }
}
```

3.8.3 曲線による回転体

回転体を作るために回転させる平面図形は、曲線を使って作ることもできます。

回転体を作るときに使うことのできる曲線は、プリズムの場合と同じで、2 次スプライン曲線、3 次スプライン曲線、ベジェ曲線の 3 種類です。

回転体の記述を、

```
lathe {
    bezier_spline
    ···
}
```

と書くと、その中に書かれた個々の 2 次元ベクトルがベジェ曲線を作るための制御点の座標だと解釈されて、ベジェ曲線による平面図形を回転させた回転体が作られます。

次のシーンは、ベジェ曲線を使って作ったハート形を回転させた回転体を作っています。

シーンの例 bezierlathe.pov

```
camera {
    location <0, 30, -70>
    look_at <0, -2, 0>
    angle 70
```

```
}

light_source { <-20, 40, -50> color rgb 1.6 }

object {
    lathe {
        bezier_spline
        8,
        <30, -4>, <18, 8>, <26, 20>, <30, 10>,
        <30, 10>, <34, 20>, <42, 8>, <30, -4>
    }
    pigment { color rgb 1 }
}
```

3.9　球スイープ

3.9.1　球スイープの基礎

球を移動させたときに軌跡として残る図形は、「球スイープ」(sphere sweep) と呼ばれます。

POV-Ray では、sphere_sweep というプリミティブを使うことによって、球スイープを作ることができます。

3.9.2　球スイープの記述

sphere_sweep を使って、直線に沿って球を移動させた球スイープを作る記述は、

```
sphere_sweep {
    linear_spline
    点の個数,
    点 1, 半径 1, 点 2, 半径 2, ···, 点 n, 半径 n
}
```

と書きます。この中の「点の個数」というところには、球の移動が開始する点、移動の方向が変化する点、そして移動が終了する点の個数を書きます。そして、点 $_1$、半径 $_1$、点 $_2$、半径 $_2$、……、点 $_n$、半径 $_n$ というところには、それらの点の位置を示す 3 次元ベクトルと、その位置での球の半径を書きます。そうすると、球がそれらの点を順番にたどっていくことによって球スイープが作られます。

次のシーンは、球スイープを作っています。

シーンの例　spheresweep.pov

```
camera {
    location <8, 10, -8>
    look_at <0, 2, 0>
    angle 70
}

light_source { <5, 20, -10> color rgb 1.6 }

object {
    sphere_sweep {
        linear_spline
        4,
        <-10, -5, 0>, 1,
        <0, -5, 0>, 1,
        <0, 5, 0>, 1,
        <0, 5, 10>, 1
    }
    pigment { color rgb 1 }
}
```

3.10 超 2 次楕円体

3.10.1 超 2 次楕円体の基礎

POV-Ray のプリミティブの中に、superellipsoid というものがあります。これは、「超 2 次楕円体」(superellipsoid) と呼ばれる形状を作るためのプリミティブです。このプリミティブを使うことによって、角が丸くなった直方体または円柱を作ることができます。

3.10.2 超 2 次楕円体の記述

superellipsoid を使って超 2 次楕円体を作る記述は、

superellipsoid { <数値$_E$, 数値$_N$> }

と書きます。この中の「数値$_E$」と「数値$_N$」のところには、角の丸みを指定する数値を書きます。数値$_E$ は、超 2 次楕円体を z 軸上の位置から見た場合の角の丸みで、数値$_N$ は、x 軸上または y 軸上の位置から見た場合の角の丸みです。

数値$_E$ と 数値$_N$ の範囲は、どちらも、0 から 1 までです。1 だと円で、0 に近づくにつれて正方形に近づいていきます。

superellipsoid は、常に、原点を中心とする位置に超 2 次楕円体を作ります。

次のシーンは、超 2 次楕円体を作っています。

シーンの例 superellipsoid.pov

```
camera {
    location <2, 1.6, -3>
    look_at 0
    angle 70
}

light_source { <4, 6, -5> color rgb 1.6 }

object {
    superellipsoid { <0.3, 0.3> }
    pigment { color rgb 1 }
}
```

数値$_E$ と 数値$_N$ をいろいろと変更してレンダリングしてみましょう。

第 4 章 CSG

4.1 CSG の基礎

4.1.1 CSG とは何か

作りたい形状を作り出すための手法のひとつとして、単純な形状を組み合わせていくことによってその形状を構築していく、というものがあります。形状を作り出すためのこのような手法は、「CSG」(constructive solid geometry) と呼ばれます。

POV-Ray で複雑な形状を作るためには、CSG に習熟する必要があります。

4.1.2 集合演算

CSG では、「集合演算」(set operation) と呼ばれる操作を使って、形状を組み合わせます。集合演算というのは、集合に対する演算のことで、次のようなものがあります。

- 合併（和集合）
- 共通部分（積集合）
- 減算（差集合）

集合演算は、それに与えられている名前によって識別されます。たとえば、形状を合併させる集合演算は、union という名前によって識別されます。

4.1.3 CSG の記述

CSG を使って形状を作る記述は、

集合演算の名前 { 物体の記述 ··· }

と書きます。そうすると、その記述は、「物体の記述」のところに書かれた物体の形状に対して集合演算を実行することによって得られる形状をあらわすことになります。

CSG の記述は、それ自体もまた物体の記述になります。CSG の記述の中には、テクスチャーの記述や変形の記述を書くことも可能です。

4.2 合併

4.2.1 合併の基礎

2 個以上の集合について、それらの少なくともひとつに含まれている要素から構成される集合を作るという集合演算は、「合併」(union) と呼ばれます。合併によって作られた集合は、「和集合」(sum set) と呼ばれます。

POV-Ray では、合併は、union という名前によって識別されます。

CSG で合併を使うことによって、いくつかの形状を合体させた形状を作り出すことができます。

4.2.2 合併の記述

合併の記述は、

union { 物体の記述 ··· }

と書きます。そうすると、その記述は、中括弧の中に記述された物体の形状に対して合併を実行した結果をあらわすことになります。

次のシーンは、円柱と直方体を合併させた形状を持つ物体を作っています。

シーンの例 union.pov

```
camera {
    location <3, 3, -4>
    look_at <-1, -0.5, 0>
    angle 70
}

light_source { <6, 8, -5> color rgb 1.6 }

union {
    cylinder { <-1, -0.8, -1>, <-1, 0.8, -1>, 1.4 }
    box { <-1, -1, -1>, <1, 1, 1> }
    pigment { color rgb 1 }
}
```

4.2.3 ハンマーの形状

二つの球と円柱を合併させることによって、ハンマーの形状を作ることができます。

次のシーンは、ハンマーの形状を持つ物体を作っています。

シーンの例 hammer.pov

```
#declare Hammer = union {
    cylinder { <-2, 3, 0>, <2, 3, 0>, 1 }
    box { <-0.3, 5, -0.2>, <0.3, -5, 0.2> }
}

camera {
    location <8, 0, -8>
    look_at 0
    angle 70
}

light_source { <7, 0, -5> color rgb 1.6 }
```

```
object {
    Hammer
    pigment { color rgb 1 }
}
```

4.2.4 テーブルの形状

天板の形状と足の形状を合併させることによって、テーブルの形状を作ることができます。

次のシーンは、テーブルの形状を持つ物体を作っています。

シーンの例 table.pov

```
#declare Table = union {
    box { <-5, -0.2, -5>, <5, 0.2, 5> }
    cylinder { <-4, 0, -4>, <-4, -7, -4>, 0.3 }
    cylinder { <4, 0, -4>, <4, -7, -4>, 0.3 }
    cylinder { <-4, 0, 4>, <-4, -7, 4>, 0.3 }
    cylinder { <4, 0, 4>, <4, -7, 4>, 0.3 }
}

camera {
    location <8, 2, -14>
    look_at <0, -4, 0>
    angle 70
}

light_source { <10, 10, 0> color rgb 1.3 }
light_source { <0, -5, -10> color rgb 1.2 }

object {
    Table
    pigment { color rgb 1 }
}
```

4.2.5 形状の内部の境界面

union を使って形状を合併させた場合、形状の内部に埋もれた境界面は、そのまま保存されます。不透明な物体を作る場合は、それでも別に問題はないのですが、その形状から透明な物体を作ったとすると、物体の内部に残っている境界面が見えてしまうことになります（透明な物体の作り方については、第 5.4 節で説明します）。

合併を使って作った形状を持つ透明な物体を作りたいけれども、その内部に境界面が残っているのは望ましくない、という場合のために、POV-Ray では、合併を実行する集合演算として、union のほかにもうひとつ、merge というものが準備されています。merge は、形状を合併させるというところは union と同じですが、それに加えて、形状の内部に埋もれた境界面を消すという動作をします。

次のシーンは、union を使って作った形状と、merge を使って作った形状のそれぞれから透明な物体を作っています。

シーンの例 merge.pov

```
camera {
    location <0, 3, -6>
    look_at 0
    angle 70
}

light_source { <6, 8, -5> color rgb 1.6 }

union {
    box { <-1, -1, -1>, <1, 1, 1> }
    box { <-0.5, -0.5, -0.5>, <1.5, 1.5, 1.5> }
    pigment { color rgbf <1, 1, 1, 0.8> }
    translate <-2, 0, 0>
}

merge {
```

```
    box { <-1, -1, -1>, <1, 1, 1> }
    box { <-1.5, -0.5, -0.5>, <0.5, 1.5, 1.5> }
    pigment { color rgbf <1, 1, 1, 0.8> }
    translate <2, 0, 0>
}
```

4.3 共通部分

4.3.1 共通部分の基礎

2 個以上の集合について、それらのすべてに共通して含まれている要素から構成される集合は、「共通部分」(intersection) または「積」(product) と呼ばれます。

POV-Ray では、共通部分を求めるという集合演算は、`intersection` という名前によって識別されます。

CSG で共通部分を求めることによって、いくつかの形状が重なっている部分（つまり同じ位置を占めている部分）だけを残してそれ以外の部分を取り除いた形状を作り出すことができます。

4.3.2 共通部分の記述

共通部分の記述は、

```
intersection { 物体の記述 ··· }
```

と書きます。そうすると、その記述は、中括弧の中に記述された物体の形状に対する共通部分をあらわすことになります。

次のシーンは、円柱と直方体の共通部分による形状を持つ物体を作っています。

シーンの例 `intersection.pov`

```
camera {
    location <2, 2, -3>
    look_at <-1, -0.5, 0>
    angle 70
}

light_source { <6, 8, -5> color rgb 1.6 }

intersection {
    cylinder { <-1, -0.8, -1>, <-1, 0.8, -1>, 1.4 }
    box { <-1, -1, -1>, <1, 1, 1> }
    pigment { color rgb 1 }
}
```

4.3.3 八角柱

多角形を、その多角形が含まれる平面に対して垂直な直線に沿って移動させることによってできる形状は、「角柱」(prism) と呼ばれます。

角柱の両端にあるそれぞれの多角形は、その角柱の「底面」(base) と呼ばれます。

底面が八角形であるような角柱、すなわち八角柱 (octagonal prism) は、二つの直方体の共通部分を求めることによって作ることができます。

次のシーンは、八角柱の物体を作っています。

シーンの例 `octagonal.pov`

```
#declare Octagonal = intersection {
    box { <-1, -1, -1>, <1, 1, 1> }
    box {
        <-1, -2, -1>, <1, 2, 1>
        rotate y*45
    }
}

camera {
    location <0, 3, -3>
    look_at 0
    angle 70
```

```
}

light_source { <-5, 10, -5> color rgb 1.6 }

object {
    Octagonal
    pigment { color rgb 1 }
}
```

4.4 減算

4.4.1 減算の基礎

集合 A と、2 個以上の集合 B_1、B_2、B_3、……があるとするとき、A の要素のうちで、B_1、B_2、B_3、……のうちのどれにも含まれていないものから構成される集合を作るという集合演算は、「減算」(subtraction) と呼ばれます。減算によって作られた集合は、「差集合」(difference set) と呼ばれます。

POV-Ray では、減算は、`difference` という名前によって識別されます。

CSG で減算を使うことによって、ひとつの形状から、それ以外の形状が重なっている部分を取り除いた形状を作り出すことができます。

4.4.2 減算の記述

減算の記述は、

`difference {` 物体の記述 `··· }`

という形の記述を書きます。そうすると、その記述は、中括弧の中の 1 個目に書かれた物体の形状から、2 個目以降に書かれた物体の形状を減算した結果をあらわすことになります。

次のシーンは、円柱から直方体を減算した形状を持つ物体と、直方体から円柱を減算した形状を持つ物体を作っています。

シーンの例 `difference.pov`

```
camera {
    location <2, 4, -4>
    look_at <-1, -0.5, 0>
    angle 70
}

light_source { <2, 6, -5> color rgb 1.6 }

difference {
    cylinder { <-1, -0.8, -1>, <-1, 0.8, -1>, 1.4 }
    box { <-1, -1, -1>, <1, 1, 1> }
    pigment { color rgb 1 }
    translate <-0.5, 0, -0.5>
}

difference {
    box { <-1, -1, -1>, <1, 1, 1> }
    cylinder { <-1, -0.8, -1>, <-1, 0.8, -1>, 1.4 }
    pigment { color rgb 1 }
    translate <0.5, 0, 0.5>
}
```

4.4.3 五円玉の形状

減算という集合演算は、穴を開けるとか、凹みを作るというような、何らかの形状から、その一部分を取り除いた形状を作りたい、というときに使われます。

たとえば、五円玉や五十円玉のような穴あき硬貨の形状は、円柱から円柱を減算することによって作り出すことができます。

次のシーンは、五円玉の形状を持つ物体を作っています。

シーンの例 `goendama.pov`

```
#declare Goendama = difference {
    cylinder { <0, -0.1, 0>, <0, 0.1, 0>, 2 }
    cylinder { <0, -1, 0>, <0, 1, 0>, 0.6 }
}

camera {
    location <0, 2, -4>
    look_at <0, -0.4, 0>
    angle 70
}

light_source { <-5, 7, -5> color rgb 1.6 }

object {
    Goendama
    pigment { color rgb 1 }
}
```

4.4.4 半球

球を、その中心を通る平面で二つに切断したそれぞれの部分は、「半球」(hemisphere) と呼ばれます。半球は、球から平面（第 3.4 節参照）を減算することによって作ることができます。

次のシーンは、半球の物体を作っています。

シーンの例 hemisphere.pov

```
#declare Hemisphere = difference {
    sphere { 0, 1 }
    plane { -y, 0 }
}

camera {
    location <0, 0.5, -2>
    look_at <0, -0.4, 0>
    angle 70
}

light_source { <-5, 4, -5> color rgb 1.6 }

object {
    Hemisphere
    pigment { color rgb 1 }
}
```

このように、形状の一部分を直線的にスパッと切り落とした形状を作りたいというときは、平面を使うと便利です。

4.5 CSG の入れ子

4.5.1 CSG の入れ子の基礎

第 4.1.3 項で説明したように、CSG の記述というのは、それ自体もまた物体の記述です。ということは、CSG の記述の中に CSG の記述を書く、ということも可能だということです。つまり、CSG の記述は、

```
difference {
    union {
        ...
    }
    ...
}
```

というように、いくらでも入れ子にすることができる、ということです。このような記述を書くことによって、集合演算によって作られた形状に対してさらに集合演算を実行する、ということができます。

次のシーンは、円柱と直方体を合併させた形状から球を減算した形状を持つ物体を作っています。

シーンの例 diffuni.pov

```
camera {
    location <0, 2, -4>
    look_at <-0.8, -0.8, 0>
    angle 70
}

light_source { <-6, 8, -5> color rgb 1.6 }

difference {
    union {
        cylinder { <-1, -0.8, -1>, <-1, 0.8, -1>, 1.4 }
        box { <-1, -1, -1>, <1, 1, 1> }
    }
    sphere { <-1, 0.5, -1>, 1.6 }
    pigment { color rgb 1 }
}
```

4.5.2 パックマンの形状

パックマンの形状は、平面と平面との共通部分を球から減算することによって作ることができます。

次のシーンは、パックマンの形状を持つ物体を作っています。

シーンの例 pacman.pov

```
#declare PacMan = difference {
    sphere { 0, 1 }
    intersection {
        plane { <-0.7, 1, 0>, 0 }
        plane { <-0.7, -1, 0>, 0 }
    }
}

camera {
    location <0.8, 0, -3>
    look_at 0
    angle 70
}

light_source { <-2, 2, -3> color rgb 1.6 }
light_source { <4, 1, 0> color rgb 1.2 }

object {
    PacMan
    pigment { color rgb <1, 1, 0> }
}
```

4.5.3 マグカップの形状

マグカップの形状は、円柱から円柱を減算した結果と、トーラスから平面を減算した結果とを合併させることによって作ることができます。

次のシーンは、マグカップの形状を持つ物体を作っています。

シーンの例 magcup.pov

```
#declare MagCup = union {
    difference {
        cylinder { <0, 0, 0>, <0, 2, 0>, 1 }
        cylinder { <0, 0.1, 0>, <0, 3, 0>, 0.9 }
    }
    difference {
        torus { 0.6, 0.1 }
        plane { x, 0 }
        rotate 90*x
```

```
        translate <0.9, 1.2, 0>
    }
}

camera {
    location <2, 3, -2>
    look_at <0, 1, 0>
    angle 70
}

light_source { <4, 6, -2> color rgb 1.6 }

object {
    MagCup
    pigment { color rgb 1 }
}
```

第5章 テクスチャー

5.1 テクスチャーの基礎

5.1.1 この章について

第 1.7.1 項で説明したように、物体の記述の中には、最低限、その物体の形状についての記述と、その物体のテクスチャー（材質感）についての記述を書く必要があります。

テクスチャーというのはいくつかの要素から構成されるのですが、これまでに紹介したシーンで記述されていたテクスチャーは、それらのすべての要素ではなくて、「ピグメント」(pigment) と呼ばれるひとつの要素だけでした。

この章では、テクスチャーを構成するそれぞれの要素について、もう少し詳細に説明していきたいと思います。

5.1.2 テクスチャーの要素

テクスチャーは、次の三つの要素から構成されます。

ピグメント (pigment)　物体の素材が持っている色。

フィニッシュ(finish)　物体の表面が持っている、光の反射に関する性質。

ノーマル (normal)　物体の表面にある細かい凹凸。

5.1.3 テクスチャーの記述

テクスチャーの記述は、基本的には、

```
texture {
    ピグメントの記述
    フィニッシュの記述
    ノーマルの記述
}
```

というように書きます。この記述は、

```
texture {     }
```

という枠組みの部分を省略して、

```
ピグメントの記述
フィニッシュの記述
ノーマルの記述
```

と書いてもかまいません。

物体を作るときには、その記述の中にテクスチャーの記述を書く必要があるわけですが、テクスチャーの三つの要素のうちで、かならず書かないといけないのはピグメントの記述だけです。フィニッシュとノーマルについては、記述を省略すると、デフォルトのフィニッシュやノーマルが指定されたとみなされます。

5.2 ピグメントの基礎

5.2.1 ピグメントについての復習

第 1.7.4 項で説明したように、物体の素材が持っている色は、「ピグメント」(pigment) と呼ばれます。

ピグメントの記述は、基本的には、

```
pigment { 色の記述 }
```

というように書きます。このような記述を書くことによって、この中に書かれた色を物体の素材に対して適用することができます。

5.2.2 複数の色から構成されるピグメント

実は、ピグメントの記述の中に書くことができるものは、単なる色の記述だけではありません。そこには、たとえば次のようなものを書くこともできます。

- カラーリスト (color list)
- カラーマップ (color map)
- ピグメントマップ (pigment map)
- イメージマップ (image map)

これらのうちのいずれかを書くことによって、一つの色だけではなくて、複数の色から構成されるピグメントを作ることができます。

5.2.3 カラーリスト

「カラーリスト」(color list) と呼ばれるものをピグメントの記述の中に書くことによって、一定のパターンで二つまたは三つの色を組み合わせたピグメントを作ることができます。カラーリストを書くことによって作られたピグメントは、「カラーリストピグメント」(color list pigment) と呼ばれます。

カラーリストピグメントとしては、次のようなものがあります。

`checker` 立方体のパターン。
`hexagon` 六角柱のパターン。
`brick` 煉瓦のパターン。

カラーリストというのは、カラーリストピグメントの名前（`checker` など）の後ろに、2 個または 3 個の色の記述を書いたもののことです。色の記述は、`checker` と `brick` の場合は 2 個、`hexagon` の場合は 3 個を、コンマで区切って並べます。

5.2.4 立方体のパターン

`checker` は、二つの色の立方体を交互に積み重ねたパターンを作るカラーリストピグメントです。

`checker` を使うカラーリストは、

```
checker 色の記述1 , 色の記述2
```

と書きます。そうすると、それぞれの色の立方体を交互に積み重ねたパターンで色が塗られることになります。たとえば、

```
checker color rgb <1, 0, 0>, color rgb <0, 0, 1>
```

と書くことによって、赤色と青色の立方体のそれぞれを交互に積み重ねたパターンを作ることができます。

次のシーンは、白色の立方体と緑色の立方体を交互に積み重ねたパターンを立方体に適用しています。

シーンの例 checker.pov

```
camera {
    location <10, 10, -15>
    look_at 0
    angle 70
}

light_source { <10, 20, -20> color rgb 1.6 }

object {
    box { -5, 5 }
    pigment { checker color rgb 1, color rgb <0, 0.6, 0> }
}
```

5.2.5 六角柱のパターン

hexagonは、y軸方向に無限に長い三色の六角柱を組み合わせたパターンを作るカラーリストピグメントです。

hexagonを使うカラーリストは、

hexagon 色の記述$_1$, 色の記述$_2$, 色の記述$_3$

と書きます。そうすると、それぞれの色の六角柱を組み合わせたパターンで色が塗られることになります。たとえば、

```
hexagon color rgb <0, 1, 1>, color rgb <1, 1, 0>, color rgb <1, 0, 1>
```

と書くことによって、水色と黄色と赤紫色の六角柱を組み合わせたパターンを作ることができます。

次のシーンは、白色と空色とネイビーの六角柱を組み合わせたパターンを立方体に適用しています。

シーンの例 hexagon.pov

```
camera {
    location <10, 10, -15>
    look_at 0
    angle 70
}

light_source { <10, 20, -20> color rgb 1.6 }

object {
    box { -5, 5 }
    pigment { hexagon color rgb 1, color rgb <0.3, 0.6, 1>,
                      color rgb <0, 0, 0.5> }
}
```

5.2.6 煉瓦のパターン

brickは、煉瓦を積み重ねたようなパターンを作るカラーリストピグメントです。

brickを使うカラーリストは、

brick 色の記述$_1$, 色の記述$_2$

と書きます。そうすると、「色の記述$_1$」で記述された色のセメントを使って、「色の記述$_2$」で記述された色の煉瓦を積み重ねたパターンで色が塗られることになります。たとえば、

```
brick color rgb <0, 1, 1>, color rgb <1, 1, 0>
```

と書くことによって、水色のセメントで黄色の煉瓦を積み重ねたパターンを作ることができます。

次のシーンは、白色のセメントで茶色の煉瓦を積み重ねたパターンを立方体に適用しています。

シーンの例 brick.pov

```
camera {
    location <10, 10, -15>
    look_at 0
    angle 70
}

light_source { <10, 20, -20> color rgb 1.6 }

object {
    box { -5, 5 }
    pigment { brick color rgb 1, color rgb <0.6, 0, 0> }
}
```

5.2.7 ピグメントに対する変形

ピグメントの記述の中には、変形の記述、つまり移動、拡大、回転の記述を書くこともできます。ピグメントの記述の中に変形の記述を書いた場合には、ピグメントが移動したり拡大したり回転したりすることになります。

次のシーンは、立方体のパターンを、x 軸方向に 5 倍だけ拡大して、z 軸を回転軸にして 30 度回転させて、x 軸方向に 3 だけ移動させたのち、立方体に適用しています。

シーンの例　tranpig.pov

```
camera {
    location <10, 10, -15>
    look_at 0
    angle 70
}

light_source { <10, 20, -20> color rgb 1.6 }

object {
    box { -5, 5 }
    pigment {
        checker color rgb 1, color rgb <0, 0.6, 0>
        scale <5, 1, 1>
        rotate 30*z
        translate <3, 0, 0>
    }
}
```

拡大率や回転角や移動先をいろいろと変更してレンダリングしてみましょう。

5.3 カラーマップ

5.3.1 カラーマップとは何か

第 5.2.2 項で説明したように、ピグメントの記述の中には、「カラーマップ」(color map) と呼ばれる記述を書くことができます。カラーマップを使うと、カラーリストと同じように、複数の色から構成されるパターンのピグメントを作ることができます。

カラーマップというのは、0 から 1 までの線分の上の位置に対して色を割り当てる記述のことです。

5.3.2 カラーマップの作り方

カラーマップは、

```
color_map {
    [位置の記述  色の記述]
              ⋮
}
```

というように書きます。この中の「位置の記述」というところには、線分の上の位置を示す、0 から 1 までのあいだの数値を書きます。たとえば、

```
[0.4 color rgb <1, 0, 0>]
```

という記述は、線分の上の 0.4 という位置に対して赤色を割り当てるという意味になります。このようにして、0 から 1 までのあいだのいくつかの位置に対して色を割り当てていくことによって、ひとつのカラーマップが作られます。なお、色が割り当てられた位置と位置とのあいだは、一方の色から他方の色へ、色が連続的に変化していくことになります。

5.3.3 パターンタイプ

カラーマップからピグメントを作りたいときは、

```
pigment {
    パターンの記述
    カラーマップ
}
```

という形の記述を書きます。この中の「パターンの記述」というところには、基本的には、「パターンタイプ名」と呼ばれる名前を書きます（その名前の後ろに、さらに何かを書かないといけない場合もあります）。

「パターンタイプ名」というのは、「パターンタイプ」(pattern type) と呼ばれるものの名前で、パターンタイプというのはパターンの種類のことです。

パターンタイプにはさまざまなものがあるのですが、この節では、次の八つのパターンタイプを紹介したいと思います。

`gradient`	勾配。
`cylindrical`	円柱。
`spherical`	球。
`wood`	年輪。
`onion`	タマネギ。
`spotted`	斑点。
`cells`	枡目。
`crackle`	ひび。

5.3.4 勾配

`gradient` というパターンタイプは、「勾配」(gradient) と呼ばれるピグメントを作ります。勾配というのは、特定の方向に沿って反復的に色を変化させるというパターンのことです。

`gradient` を使うパターンの記述は、

```
gradient 方向
```

と書きます。この中の「方向」というところには、3 次元ベクトルを値とする式を書きます。その 3 次元ベクトルは、勾配の方向を決めるためのものです。色は、そのベクトルが向いている方向に沿って変化していきます。たとえば、

```
gradient y
```

というパターンの記述を書くことによって、y 軸の方向に沿って色が変化する勾配のピグメントを作ることができます。

次のシーンは、カラーマップを使って作られた、y 軸に沿って色が変化する勾配を、0.4 倍に縮小させたのち、立方体に適用しています。

シーンの例 `gradient.pov`

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }
```

```
object {
    box { -1, 1 }
    pigment {
        gradient y
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 0.4
    }
}
```

5.3.5 円柱

cylindrical というパターンタイプは、「円柱」(cylindrical) と呼ばれるピグメントを作ります。円柱というのは、y 軸を中心にして、そこから、xz 平面に沿って放射状に色が変化していくパターンのことです。中心の色はカラーマップの 1 の色で、y 軸から離れるにつれて 0 の色に近づいていきます。

cylindrical を使うパターンの記述は、ただ単に cylindrical と書くだけです。

カラーマップで作られたピグメントも、移動させたり拡大したり回転させたりすることができます。円柱の中心軸は y 軸に固定されていますので、それ以外の向きを持つ円柱を作りたい場合は、それを回転させる必要があります。

次のシーンは、カラーマップを使って作られた円柱のパターンを、x 軸で 90 度回転させて、x 軸の方向に 1 だけ移動させたのち、立方体に適用しています。

シーンの例 cylindrical.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }

object {
    box { -1, 1 }
    pigment {
        cylindrical
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        rotate 90*x
        translate <1, 0, 0>
    }
}
```

5.3.6 球

spherical というパターンタイプは、「球」(sphere) と呼ばれるピグメントを作ります。球というのは、原点を中心にして、そこから放射状に色が変化していくパターンのことです。ただし、中心の色はカラーマップの 1 の色で、y 軸から離れるにつれて 0 の色に近づいていきます。

spherical を使うパターンの記述は、ただ単に spherical と書くだけです。

次のシーンは、カラーマップを使って作られた球のパターンを、2 倍に拡大して、中心を <1, 1, -1> の位置に移動させたのち、立方体に適用しています。

シーンの例 spherical.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}
```

```
light_source { <5, 5, -5> color rgb 1.6 }

object {
    box { -1, 1 }
    pigment {
        spherical
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 2
        translate <1, 1, -1>
    }
}
```

5.3.7 年輪

wood というパターンタイプは、「年輪」(annual ring) と呼ばれるピグメントを作ります。年輪というのは、z 軸を中心にして、xy 平面に沿って反復的に同心円柱を作っていくパターンのことです。

wood を使うパターンの記述は、ただ単に wood と書くだけです。

次のシーンは、カラーマップを使って作られた年輪のパターンを、0.2 倍に縮小して、x 軸の方向に 1 だけ移動させたのち、立方体に適用しています。

シーンの例 wood.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }

object {
    box { -1, 1 }
    pigment {
        wood
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 0.2
        translate <1, 0, 0>
    }
}
```

5.3.8 タマネギ

onion というパターンタイプは、「タマネギ」(onion) と呼ばれるピグメントを作ります。タマネギというのは、原点を中心にして、反復的に同心球を作っていくパターンのことです。

onion を使うパターンの記述は、ただ単に onion と書くだけです。

次のシーンは、カラーマップを使って作られたタマネギのパターンを、0.2 倍に縮小して、中心を <1, 1, -1> の位置に移動させたのち、立方体に適用しています。

シーンの例 onion.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }

object {
```

```
    box { -1, 1 }
    pigment {
        onion
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 0.2
        translate <1, 1, -1>
    }
}
```

5.3.9 斑点

spotted というパターンタイプは、「斑点」(spot) と呼ばれるピグメントを作ります。斑点というのは、ランダムな形状で色が変化していくパターンのことです。

spotted を使うパターンの記述は、ただ単に spotted と書くだけです。

次のシーンは、カラーマップを使って作られた斑点を、0.3 倍に縮小させたのち、立方体に適用しています。

シーンの例 spotted.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }

object {
    box { -1, 1 }
    pigment {
        spotted
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 0.3
    }
}
```

5.3.10 枡目

cells というパターンタイプは、「枡目」(cell) と呼ばれるピグメントを作ります。枡目というのは、空間を立方体の区画に細分化して、カラーマップの中のランダムな位置の色をそれぞれの区画に割り当てるというパターンのことです。

cells を使うパターンの記述は、ただ単に cells と書くだけです。

次のシーンは、カラーマップを使って作られた枡目を、0.4 倍に縮小させたのち、立方体に適用しています。

シーンの例 cells.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }

object {
    box { -1, 1 }
    pigment {
        cells
        color_map {
```

```
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 0.4
    }
}
```

5.3.11 ひび

crackle というパターンタイプは、「ひび」(crackle) と呼ばれるピグメントを作ります。ひびというのは、空間をランダムな形状の区画に細分化するというパターンのことです。

crackle を使うパターンの記述は、

crackle ひびのタイプの記述

と書きます。この中の「ひびのタイプの記述」というところには、ひびが持っている四つのタイプのうちのどれかの名前を書きます。ひびのタイプによっては、その名前の右側に数値またはベクトルを書くことによって、ひびの形状などを変化させることができます。

ひびが持っている四つのタイプのそれぞれは、form、metric、offset、solid という名前によって識別されます。form の右側には 1 個の 3 次元ベクトル（デフォルトは <-1, 1, 0>）、metric の右側には 1 個の数値（デフォルトは 2）、offset の右側には 1 個の数値（デフォルトは 0）を書くことができます。solid の右側には、何も書くことができません。

次のシーンは、カラーマップを使って作られた metric タイプのひびを、0.4 倍に縮小させたのち、立方体に適用しています。

シーンの例 crackle.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }

object {
    box { -1, 1 }
    pigment {
        crackle metric 1
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 0.4
    }
}
```

solid タイプのひびは、空間をランダムな形状の区画に細分化して、カラーマップの中のランダムな位置の色をそれぞれの区画に割り当てるというパターンです。このタイプのひびを使うことによって、ステンドグラスのようなピグメントを作ることができます。

次のシーンは、カラーマップを使って作られた solid タイプのひびを、0.6 倍に縮小させたのち、立方体に適用しています。

シーンの例 cracklesolid.pov

```
camera {
    location <3, 2, -3>
    look_at 0
    angle 70
}

light_source { <5, 5, -5> color rgb 1.6 }

object {
    box { -1, 1 }
```

```
    pigment {
        crackle solid
        color_map {
            [0 color rgb 1 ]
            [1 color rgb <0, 0, 1> ]
        }
        scale 0.6
    }
}
```

5.4 光の透過

5.4.1 フィルターとトランスミット

POV-Ray では、不透明な色だけではなくて、透明な色というものを記述することもできます。透明な色から作られたピグメントを物体に与えると、それは、透明な物体、つまり光を透過させる物体になります。

透明な色は、RGB に加えて、「フィルター」(filter) または「トランスミット」(transmit) と呼ばれる数値を書くことによって記述することができます。

フィルターとトランスミットというのは、どちらも、光をどれくらい透過させるのかという比率です。完全に不透明というのが 0 で、完全に透明というのが 1 です。

フィルターとトランスミットとの相違点は、透過する光の色が RGB によって制限されるかどうかというところにあります。フィルターの場合は、RGB で記述された色の光だけが透過するのに対して、トランスミットの場合は、RGB で記述された色とは無関係に、すべての色の光が透過することになります。

5.4.2 フィルターによる色の記述

フィルターを指定することによって透明な色を記述したいときは、

```
color rgbf 4次元ベクトル
```

という形のものを書きます。この中の「4 次元ベクトル」というところには、赤の比率、緑の比率、青の比率、そしてフィルターを、この順番で並べた 4 次元ベクトルを書きます。たとえば、

```
color rgbf <1, 0, 0, 1>
```

と記述された色は、赤色の光を完全に透過させることになります。

次のシーンは、緑色の光を完全に透過させる色を適用した球を作っています。

シーンの例 filter.pov

```
camera {
    location <0, 0, -5>
    look_at 0
    angle 70
}

light_source { <0, 0, -10> color rgb 1.6 }

object {
    box { <-4, -4, 0>, <4, 4, 1> }
    pigment {
        checker color rgb 1, color rgb <0, 0, 1>
    }
}

object {
    sphere { <0, 0, -2>, 1 }
    pigment { color rgbf <0, 1, 0, 1> }
}
```

物体の色の記述をいろいろと変更してレンダリングしてみましょう。

5.4.3 屈折率

色の記述の中にフィルターを指定する数値を書くことによって、透明な物体を作ることができるわけですが、しかし、それだけだと、光がその物体の表面を通過するときに屈折が生じませんので、その物体は、まだ、ガラスや水のようには見えません。透明な物体をリアルに見せるためには、その物体の屈折率 (index of refraction) を記述する必要があります。

屈折率は、物体を作る記述の中に、

```
interior { ior 屈折率 }
```

という形の記述を書くことによって指定します。「屈折率」というところには、数値を値とする式を書きます。そうすると、その数値が屈折率として物体に適用されます。ちなみに、水の屈折率は 1.33 で、ガラスの屈折率は（成分によって少し違いますが）だいたい 1.5 前後です。

次のシーンは、すべての光を完全に透過させる色を適用した球を作って、その球に対して 1.33 という屈折率（水と同じ）を与えています。

シーンの例 ior.pov

```
camera {
    location <0, 0, -5>
    look_at 0
    angle 70
}

light_source { <0, 0, -10> color rgb 1.6 }

object {
    box { <-4, -4, 0>, <4, 4, 1> }
    pigment {
        checker color rgb 1, color rgb <0, 0, 1>
    }
}

object {
    sphere { <0, 0, -2>, 1 }
    pigment { color rgbf <1, 1, 1, 1> }
    interior { ior 1.33 }
}
```

屈折率をいろいろと変更してレンダリングしてみましょう。

5.4.4 計算の複雑さ

POV-Ray は、レンダリングに要する時間をなるべく短くするために、計算の複雑さに対して限界を設定して、それよりも複雑な計算をしないようにしています。計算の複雑さの限界は、ひとつの数値であらわされていて、デフォルトではその数値として 5 が設定されています。

透明な物体を含むシーンをレンダリングする場合、計算の複雑さの限界が 5 のままだと、その限界のために計算が中途半端で終わってしまって、レンダリングの結果として得られる画像の一部分が黒くなってしまう、ということがしばしばあります。

計算の複雑さの限界として 5 以外の数値を設定したいときは、シーンの中に、

```
global_settings { max_trace_level 限界 }
```

という形の記述を書きます。「限界」というところには、数値を値とする式を書きます。そうすると、その数値が、計算の複雑さの限界として設定されます。たとえば、

```
global_settings { max_trace_level 8 }
```

という記述を書くことによって、計算の複雑さの限界として 8 を設定することができます。

次のシーンは、レンズ状の透明な物体を、3 個、カメラの位置から見ると重なって見えるように並べています。

シーンの例 maxtracelevel.pov

```
global_settings { max_trace_level 7 }

#declare Lens = object {
```

```
    sphere { 0, 1 }
    pigment { color rgbf <1, 1, 1, 1> }
    interior { ior 1.5 }
    scale <1, 1, 0.05>
}

camera {
    location <0, 0, -5>
    look_at 0
    angle 70
}

light_source { <0, 0, -10> color rgb 1.6 }

object {
    box { <-4, -4, 0>, <4, 4, 1> }
    pigment {
        checker color rgb 1, color rgb <0, 0, 1>
    }
}

object { Lens translate <0.4, -0.3, -1> }
object { Lens translate <-0.4, -0.3, -1.5> }
object { Lens translate <0, 0.3, -2> }
```

このシーンは、計算の複雑さの限界をあらわす数値として 7 を設定しています。デフォルトの 5 にするとどうなるか、試してみてください。

5.5 空

5.5.1 空にピグメントを与える方法

ピグメントは、物体だけではなく、空 (sky) に与えることも可能です。空に対してピグメントを与えたいときは、

```
sky_sphere { ピグメントの記述 }
```

という形のものを書きます。たとえば、

```
sky_sphere { pigment { color rgb <1, 1, 0> } }
```

という記述を書くことによって、空に対して黄色のピグメントを与えることができます。

次のシーンは、空に対して空色のピグメントを与えています。

シーンの例 sky.pov

```
#declare Radius = 6.4e6; // Earth radius

camera {
    location <0, 1, 0>
    look_at <0, 1.2, 1>
    angle 70
}

light_source { <0, 1e10, 0> color rgb 1.6 }

sky_sphere { pigment { color rgb <0.3, 0.6, 1> } }

object {
    sphere { <0, -Radius, 0>, Radius }
    pigment { checker color rgb 1, color rgb <0, 0.4, 0> }
}
```

空の色をいろいろと変更してレンダリングしてみましょう。

5.5.2 現実的な空

空というのは、地平線に近いところは色が薄く、天頂に近付くほど色が濃くなっていくのが普通です。そのような現実的な空は、勾配を使うことによって作ることができます。

次のシーンは、勾配を使って作られたピグメントを空に対して与えています。

シーンの例 realsky.pov

```
#declare Radius = 6.4e6; // Earth radius

camera {
    location <0, 1, 0>
    look_at <0, 1.2, 1>
    angle 70
}

light_source { <0, 1e10, 0> color rgb 1.6 }

sky_sphere {
    pigment {
        gradient y
        color_map {
            [0 color rgb <0.7, 0.9, 1>]
            [0.5 color rgb <0, 0.5, 1>]
        }
    }
}

object {
    sphere { <0, -Radius, 0>, Radius }
    pigment { checker color rgb 1, color rgb <0, 0.4, 0> }
}
```

カラーマップをいろいろと変更してレンダリングしてみましょう。

5.6 フィニッシュ

5.6.1 フィニッシュの基礎

第 5.1.2 項で説明したように、テクスチャーというのは、ピグメント、フィニッシュ、ノーマルという三つの要素から構成されます。この節では、テクスチャーを構成している要素のひとつであるフィニッシュについて説明したいと思います。

「フィニッシュ」(finish) というのは、物体の表面が持っている、光の反射に関する性質のことです。

物体の表面での光の反射には、拡散反射と鏡面反射という 2 種類のものがあります。

「拡散反射」(diffuse reflection) というのは、光が当たる角度とは無関係に反射光がさまざまな角度に出て行く反射のことです。それに対して、「鏡面反射」(specular reflection) というのは、光が当たる角度に応じたひとつの方向だけに反射光が出て行く反射のことです。強く鏡面反射する物体の表面には、周囲の風景が映り込むことになります。

物体に当たった光の強さに対する、物体の表面で反射して出て行く光の強さの比率は、「反射率」(reflectance) と呼ばれます。

フィニッシュの記述は、

finish { 反射に関する記述 }

というように書きます。

5.6.2 拡散反射

拡散反射に関する記述は、

diffuse 反射率

と書きます。「反射率」のところには、0 から 1 までの数値を値とする式を書きます。そうすると、その数値が拡散反射の反射率として物体に適用されます。拡散反射の反射率のデフォルトは

0.6 です。
次のシーンは、拡散反射の反射率が 0.4 の球を作っています。

シーンの例 diffuse.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
    finish { diffuse 0.4 }
}
```

拡散反射の反射率をいろいろと変更してレンダリングしてみましょう。

5.6.3 鏡面反射

鏡面反射に関する記述は、

`reflection` 反射率

と書きます。「反射率」のところには、0 から 1 までの数値を値とする式を書きます。そうすると、その数値が鏡面反射の反射率として物体に適用されます。鏡面反射の反射率のデフォルトは 0 です。
次のシーンは、拡散反射の反射率が 0.2 で、鏡面反射の反射率が 0.4 の球を作っています。

シーンの例 reflection.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    box { <-5, -3, -5>, <5, -2, 5> }
    pigment { checker color rgb 1, color rgb <0, 0.4, 0> }
}

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
    finish { diffuse 0.2 reflection 0.4 }
}
```

拡散反射の反射率や鏡面反射の反射率をいろいろと変更してレンダリングしてみましょう。

5.6.4 ハイライト

光沢のある物体に光を当てたときに、光源が映り込むことによってできる、まわりよりも強く光っている部分は、「ハイライト」(highlight) と呼ばれます。POV-Ray では、鏡面反射の反射率を設定しても、デフォルトではハイライトはできません。
ハイライトができるようにしたいときは、フィニッシュの記述の中に、

`phong` 強さ

`phong_size` 滑らかさ

という形のものを書きます。

「強さ」のところには、0 から 1 までの数値を値とする式を書きます。これは、ハイライトの光の強さを示す数値で、1 に近いほど光が強くなります。デフォルトは 0 です。

「滑らかさ」のところには 1 から 250 までの数値を値とする式を書きます。これは、物体の表面の滑らかさを示す数値です。0 に近いほどざらざらになって、250 に近いほどつるつるになります。デフォルトは、プラスチックが持つ 40 という滑らかさです。

次のシーンは、ハイライトの強さが 0.6 で、物体の表面の滑らかさが 120 の球を作っています。

シーンの例　highlight.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    box { <-5, -3, -5>, <5, -2, 5> }
    pigment { checker color rgb 1, color rgb <0, 0.4, 0> }
}

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
    finish {
        diffuse 0.2
        reflection 0.4
        phong 0.6
        phong_size 120
    }
}
```

ハイライトの強さや物体の表面の滑らかさをいろいろと変更してレンダリングしてみましょう。

5.7　ノーマル

5.7.1　ノーマルの基礎

第 5.1.2 項で説明したように、テクスチャーというのは、ピグメント、フィニッシュ、ノーマルという三つの要素から構成されます。この節では、テクスチャーを構成している要素のひとつであるノーマルについて説明したいと思います。

「ノーマル」(normal) というのは、物体の表面にある細かい凹凸のことです。POV-Ray では、第 5.3 節で説明したパターンタイプを指定することによって、さまざまなパターンの凹凸を、物体の表面に与えることができます。

5.7.2　ノーマルの記述

ノーマルの記述は、

```
normal { パターンの記述 }
```

というように書きます。「パターンの記述」というところには、基本的にはパターンタイプ名を書きます（その後ろに、さらに何かを書かないといけない場合もあります)。

ノーマルの記述の中には、変形の記述、つまり移動、拡大、回転の記述を書くこともできます。そうすることによって、ノーマルを移動させたり拡大したり回転させたりすることができます。

この節では、ノーマルでよく使われるパターンタイプとして、次の二つを紹介したいと思います。

`bumps`　バンプ。

`facets`　切子面。

5.7.3　バンプ

bumps というパターンタイプは、「バンプ」(bump) と呼ばれるノーマルを作ります。バンプというのは、オレンジの皮の表面のような凹凸のことです。

bumps を使うパターンの記述は、

bumps 深さ

と書きます。この中の「深さ」というところには、数値を値とする式を書きます。そうすると、その数値を深さとするバンプが作られます。

次のシーンは、バンプが適用された球を作っています。

シーンの例 bumps.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
    normal { bumps 0.4 scale 0.08 }
}
```

凹凸の深さや拡大率をいろいろと変更してレンダリングしてみましょう。

5.7.4 切子面

facets というパターンタイプは、「切子面」(facet) と呼ばれるノーマルを作ります。切子面というのは、小さな平面が集まってできた曲面のことです。

facets を使うパターンの記述は、ただ単に facets と書くだけです。

次のシーンは、切子面が適用された球を作っています。

シーンの例 facets.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
    normal { facets size 0.2 }
}
```

拡大率をいろいろと変更してレンダリングしてみましょう。

第 6 章　インクルードファイル

6.1　インクルードファイルの基礎

6.1.1　インクルードファイルとは何か

第 1.10 節で説明したように、POV-Ray では、数値、ベクトル、色、物体、光源など、さまざまなものに識別子を与えることができます。

何かに識別子を与える記述、つまり宣言は、基本的には、その識別子を使うシーンの中に書けばいいわけですが、識別子を与える記述と、その識別子を使う記述とを、別々のファイルの中に入れておく、ということも可能です。そのように、宣言をシーンから独立させてひとつのファイ

ファイル名	内容
colors.inc	色
shapes.inc	形状
shapes2.inc	形状（shapes.inc が必要）
shapesq.inc	形状（shapes.inc が必要）
woods.inc	木材のテクスチャー（colors.inc が必要）
stones.inc	石のテクスチャー（colors.inc が必要）
metals.inc	金属のテクスチャー
golds.inc	金のテクスチャー

表 6.1: 主要な標準インクルードファイル

ルに入れておくことによって、過去に書いた宣言を新しいシーンで再利用することが、とても簡単にできるようになります。

別のファイルの中の記述をシーンの中に埋め込むことを、ファイルを「インクルードする」(include) と言います。そして、ファイルにインクルードされるファイル（またはその内容）は、「インクルードファイル」(include file) と呼ばれます。

POV-Ray のシーンを格納するファイルには、.pov という拡張子を付けるわけですが、それに対して、POV-Ray のインクルードファイルには、.inc という拡張子を付けることになっています。

6.1.2　ファイルをインクルードする記述

シーンの中にファイルの内容をインクルードしたいときは、

```
#include "パス名"
```

という記述を書きます。そうすると、「パス名」のところに書かれたパス名を持つファイルの内容が、この記述の場所に埋め込まれることになります。たとえば、

```
#include "namako.inc"
```

という記述を書くことによって、namako.inc というファイルの内容を、その記述の場所に埋め込むことができます。

6.1.3　標準インクルードファイル

POV-Ray の公式サイトで配布されているアーカイブの中には、POV-Ray の本体だけではなくて、さまざまな識別子を宣言しているインクルードファイルも含まれています。そのような、POV-Ray に標準で添付されているインクルードファイルは、「標準インクルードファイル」(standard include file) と呼ばれます。表 6.1 は、標準インクルードファイルのうちの主要なものについて、そのファイル名と、その中に書かれているものを示しています。

POV-Ray を Windows にインストールした場合、標準インクルードファイルは、POV-Ray をインストールしたユーザーのマイドキュメントの中にある POV-Ray のフォルダーの中にある include というフォルダーの中に格納されます。しかし、標準インクルードファイルを指定するパス名の中に、そのフォルダーを指定する長いパス名を書く必要はありません。なぜなら、標準インクルードファイルは、それがたとえシーンのファイルと同じフォルダーの中に置かれていないとしても、ファイルの名前を書くだけで指定することができるからです。たとえば、colors.inc という標準インクルードファイルをインクルードしたい場合は、

```
#include "colors.inc"
```

と書くだけで、それをインクルードすることができます。

6.2　色の標準インクルードファイル

6.2.1　色の標準インクルードファイル

colors.inc という標準インクルードファイルの中には、さまざまな色に対して識別子を与える宣言が書かれています。

6.2.2 colors.inc を使ったシーンの例

次のシーンは、 colors.inc で洗顔されている識別子を使って、三つの立方体に対して色を適用しています。

シーンの例 colors.pov

```
#include "colors.inc"

#declare Box = box { <-1, -1, -1>, <1, 1, 1> }

camera {
    location <6, 3, -4>
    look_at <1.4, 0, 0>
    angle 70
}

light_source { <8, 10, -12> color rgb 1.6 }

object { Box pigment { SkyBlue } translate <-3, 0, 0> }
object { Box pigment { Orange } }
object { Box pigment { SpringGreen } translate <3, 0, 0> }
```

6.3 形状の標準インクルードファイル

6.3.1 形状の標準インクルードファイルの基礎

次の標準インクルードファイルの中には、さまざまな形状に対して識別子を与える宣言が書かれています。

- shapes.inc
- shapes2.inc
- shapesq.inc

6.3.2 shapes.inc

shapes.inc の中に書かれている形状は、立方体、球、楕円体、円柱、円錐、平面、放物面、そして双曲面です。

次のシーンは、 shapes.inc を使って双曲面を作っています。

シーンの例 shapes.pov

```
#include "shapes.inc"

camera {
    location <0, 0, -10>
    look_at 0
    angle 70
}

light_source { <3, 2, -5> color rgb 1.6 }

object {
    Hyperboloid_Y
    pigment { color rgb 1 }
}
```

6.3.3 shapes2.inc

shapes2.inc の中に書かれている形状には、四面体、八面体、十二面体、二十面体、円錐台、斜方六面体、六角柱、ピラミッドなどがあります。

shapes2.inc を使うためには、それに先立って shapes.inc をインクルードしておく必要があ

ります。
　次のシーンは、 shapes2.inc を使って六角柱を作っています。

シーンの例　shapes2.pov

```
#include "shapes.inc"
#include "shapes2.inc"

camera {
    location <3, 1, -2>
    look_at 0
    angle 70
}

light_source { <6, 8, -4> color rgb 1.6 }

object {
    Hexagon
    pigment { color rgb 1 }
}
```

6.3.4　shapesq.inc

　shapesq.inc の中には、さまざまな不思議な形状が書かれています。
　shapesq.inc を使うためには、それに先立って shapes.inc をインクルードしておく必要があります。
　次のシーンは、 shapesq.inc の中で Quartic_Cylinder という識別子が与えられている形状を持つ物体を作っています。

シーンの例　shapesq.pov

```
#include "shapes.inc"
#include "shapesq.inc"

camera {
    location <0, 1, -3>
    look_at 0
    angle 70
}

light_source { <4, 8, -4> color rgb 1.6 }
light_source { <0, -1, -1> color rgb 0.8 }

object {
    Quartic_Cylinder
    pigment { color rgb 1 }
}
```

6.4　テクスチャーの標準インクルードファイル

6.4.1　形状の標準インクルードファイルの基礎

　次の標準インクルードファイルの中には、さまざまなテクスチャーに対して識別子を与える宣言が書かれています。

- woods.inc
- stones.inc
- metals.inc
- golds.inc

6.4.2　woods.inc

　woods.inc の中には、さまざまな木材のテクスチャーが書かれています。
　woods.inc を使うためには、それに先立って colors.inc をインクルードしておく必要があり

ます。

次のシーンは、woods.inc の中で T_Wood10 という識別子が与えられているテクスチャーが適用された物体を作っています。

シーンの例 woods.pov

```
#include "colors.inc"
#include "woods.inc"

camera {
    location <2, 2, -3>
    look_at 0
    angle 70
}

light_source { <6, 8, -5> color rgb 1.6 }

object {
    box { <-1, -1, -1>, <1, 1, 1> }
    texture { T_Wood10 }
}
```

6.4.3 stones.inc

stones.inc の中には、さまざまな石のテクスチャーが書かれています。

stones.inc を使うためには、それに先立って colors.inc をインクルードしておく必要があります。

次のシーンは、stones.inc の中で T_Stone17 という識別子が与えられているテクスチャーが適用された物体を作っています。

シーンの例 stones.pov

```
#include "colors.inc"
#include "stones.inc"

camera {
    location <2, 2, -3>
    look_at 0
    angle 70
}

light_source { <6, 8, -5> color rgb 1.6 }

object {
    box { <-1, -1, -1>, <1, 1, 1> }
    texture { T_Stone17 }
}
```

6.4.4 metals.inc

metals.inc の中には、さまざまな金属のテクスチャーが書かれています。

次のシーンは、metals.inc の中で T_Chrome_2A という識別子が与えられているテクスチャーが適用された物体を作っています。

シーンの例 metals.pov

```
#include "metals.inc"

camera {
    location <0, 1, -3>
    look_at <0, 2, 0>
    angle 70
}

light_source { <10, 10, -10> color rgb 1.6 }

object {
    plane { y, 0 }
```

```
    pigment { checker color rgb 1, color rgb <0, 0.4, 0> }
}

object {
    sphere { <0, 2, 0>, 1 }
    texture { T_Chrome_2A }
}
```

6.4.5 golds.inc

golds.inc の中には、金のテクスチャーが書かれています。

次のシーンは、golds.inc の中で T_Gold_1A という識別子が与えられているテクスチャーが適用された物体を作っています。

シーンの例　golds.pov

```
#include "golds.inc"

camera {
    location <0, 1, -3>
    look_at <0, 2, 0>
    angle 70
}

light_source { <10, 10, -10> color rgb 1.6 }

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 0.4, 0> }
}

object {
    sphere { <0, 2, 0>, 1 }
    texture { T_Gold_1A }
}
```

6.5 オリジナルなインクルードファイル

6.5.1 この節について

インクルードファイルというのは、POV-Ray のユーザーが自分で書くことも可能です。自分がよく使う形状やテクスチャーなどは、シーンの中に書くのではなく、インクルードファイルの中に書いておけば、同じものを何回も書く手間が省けますので、とても便利です。

そこで、この節では、インクルードファイルというのはどのように書けばいいのか、ということについて説明したいと思います。

6.5.2 インクルードファイルの書き方

インクルードファイルの書き方は、シーンの書き方とほとんど同じで、異なっているのは次の 2 点だけです。

(1) インクルードファイルの中には、カメラやライトや物体などに識別子を与える宣言を書くことはできるが、それらを実際に作る記述を書くことはできない。

(2) インクルードファイルの先頭と末尾には、同一のインクルードファイルが重複して処理されることを防ぐためと、POV-Ray のバージョン番号を退避させて、設定して、復元するための記述を書かないといけない。

6.5.3 インクルードファイルの先頭の記述

インクルードファイルの先頭には、まず最初に、同一のインクルードファイルが重複して処理されることを防ぐために、

```
#ifndef ( 独自の識別子 )
```

という記述を書く必要があります。「独自の識別子」というところには、ほかのインクルードファイルでは使われていない識別子を書きます。

インクルードファイルの先頭には、上の記述に続けて、識別子を使って POV-Ray のバージョン番号を退避させておくために、

```
#declare 独自の識別子 = version;
```

という記述を書く必要があります。「独自の識別子」というところには、`#ifndef` に書いたものと同じ識別子を書きます。

インクルードファイルの先頭には、上の記述に続けて、インクルードファイルを処理することのできる POV-Ray のバージョン番号を POV-Ray に設定するために、

```
#version バージョン番号;
```

という記述を書く必要があります。「バージョン番号」というところには、インクルードファイルを処理することのできる POV-Ray のバージョン番号を書きます。

6.5.4 インクルードファイルの末尾の記述

インクルードファイルの末尾には、まず最初に、POV-Ray のバージョン番号を復元するために、

```
#version 独自の識別子;
```

という記述を書く必要があります。「独自の識別子」のところには、インクルードファイルの先頭に書いたものと同じ識別子を書きます。

インクルードファイルの末尾には、上の記述に続けて、

```
#end
```

という記述を書く必要があります。この記述は、`#ifndef` とペアになるもので、インクルードファイルを処理する必要がない場合に、どこまでスキップすればいいかということを示しています。

6.5.5 オリジナルなインクルードファイルの例

それでは、オリジナルなインクルードファイルを実際に書いてみましょう。

次のインクルードファイルは、正方形の枠という形状に対して `Frame` という識別子を与えています。

インクルードファイルの例 `frame.inc`

```
#ifndef (Frame_Inc_Temp)
#declare Frame_Inc_Temp = version;
#version 3.6;

#declare Frame = difference {
    box { <-1, -0.05, -1>, <1, 0.05, 1> }
    box { <-0.9, -0.1, -0.9>, <0.9, 0.1, 0.9> }
}

#version Frame_Inc_Temp;
#end
```

次のシーンは、上の `frame.inc` の中で `Frame` という識別子が与えられている形状を持つ物体を作っています。

シーンの例 `frame.pov`

```
#include "frame.inc"

camera {
    location <0, 2, -2>
    look_at 0
    angle 70
}
```

```
light_source { <0, 4, -3> color rgb 1.6 }

object {
    Frame
    pigment { color rgb 1 }
}
```

第 7 章　繰り返しと選択

7.1　繰り返しと選択の基礎

7.1.1　シーンの実行

シーンは、基本的には、カメラの記述、光源の記述、そして物体の記述から構成されます。カメラの記述というのは、「カメラを作る」という動作の記述です。同じように、光源の記述というのは「光源を作る」という動作の記述で、物体の記述というのは「物体を作る」という動作の記述です。

POV-Ray は、レンダリングに先立って、シーンの中に書かれている記述を上から順番に実行していきます。そして、カメラの記述にしたがってカメラを作り、光源の記述にしたがって光源を作り、物体の記述にしたがって物体を作ります。

POV-Ray によるシーンの処理は、原則的には上から下へ直線的に実行されるのですが、直線的ではない実行を指示することも可能です。直線的ではない実行というのは、繰り返しと選択です。

「繰り返し」(iteration) というのは、何らかの動作を、一回だけ実行するのではなくて、何回も反復して実行することです。そして「選択」(selection) というのは、いくつかの候補の中からひとつの動作を選んで実行することです。

繰り返しや選択を含むシーンを書くことによって、さまざまな興味深い結果を得ることができます。

7.1.2　条件

繰り返しや選択は、何らかの判断にもとづいて実行されます。

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」(condition) と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」(true) であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」(false) であると言われます。

7.1.3　真偽値

真を意味するデータと、偽を意味するデータは、総称して「真偽値」(Boolean) と呼ばれます。

POV-Ray では、真偽値は数値によってあらわされます。真偽値を必要とする文脈では、0 は偽と解釈され、0 以外の数値は真と解釈されます。

7.1.4　識別子の再宣言

第 1.10 節で説明したように、POV-Ray では、「宣言」と呼ばれる記述を書くことによって、「識別子」と呼ばれる名前をさまざまなものに与えることができます。

何かに名前として与える識別子は、すでに使われているものでもかまいません。つまり、すでに何らかのものに名前として与えられている識別子を別のものに与え直す、ということも可能だということです。たとえば、

```
#declare Takasa = 80;
```

という宣言で、80 という数値に `Takasa` という識別子を与えたのち、

```
#declare Takasa = 230;
```

という宣言で、230 という別の数値に `Takasa` という識別子を与えることが可能です。

このように、すでに何らかのものに名前として与えられている識別子を別のものに与え直すことを、識別子を「再宣言する」(redeclare) と言います。

識別子が与えられているものに対する演算の結果に対して、もとの識別子と同じものを与える、ということも可能です。たとえば、

```
#declare Takasa = Takasa + 50;
```

という宣言を書くことによって、Takasa という識別子が与えられている数値を、現在のものから、それよりも 50 だけ大きいものに変更することができます。

7.2 関係演算子

7.2.1 関係演算子の基礎

二つのデータのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる演算子は、「関係演算子」(relational operator) と呼ばれます。

関係演算子の優先順位は、加算や乗算などの演算子よりも低くなっています。

関係演算子を含む式を評価すると、演算子の左右にある式が評価されて、それらの式の値のあいだに関係が成り立っているかどうかという判断が実行されます。そして、関係が成り立っているならば真 (1)、成り立っていないならば偽 (0) が、式全体の値になります。

7.2.2 大小関係

次の関係演算子を使うことによって、数値の大小関係について調べることができます。

A > B　A は B よりも大きい。
A < B　A は B よりも小さい。
A >= B　A は B よりも大きいか、または A と B とは等しい。
A <= B　A は B よりも小さいか、または A と B とは等しい。

たとえば、

```
X < 100
```

という式を書くことによって、「X は 100 よりも小さい」という条件を記述することができます。

7.2.3 等しいかどうか

二つのオブジェクトが等しいかどうかということは、次の関係演算子を使うことによって調べることができます。

A = B　A と B とは等しい。
A != B　A と B とは等しくない。

たとえば、

```
X = 100
```

という式を書くことによって、「X と 100 とは等しい」という条件を記述することができます。

7.3 論理演算子

7.3.1 論理演算子の基礎

処理の対象が真偽値で、処理の結果も真偽値であるような動作をあらわしている演算子は、「論理演算子」(logical operator) と呼ばれます。

POV-Ray には、次の三つの論理演算子があります。

A & B　A かつ B である。
A | B　A または B である。
! A　A ではない。

& と | は、関係演算子よりも低い優先順位を持っています。

7.3.2 論理積演算子

& は、「論理積演算子」(logical AND operator) と呼ばれます。これは、二つの条件が両方とも成り立っているかどうかを判断したいとき、つまり、A かつ B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
1  &  1    ⟶  1
1  &  0    ⟶  0
0  &  1    ⟶  0
0  &  0    ⟶  0
```

たとえば、

```
X > 100 & X < 200
```

という式を書くことによって、「X は 100 よりも大きく、かつ、X は 200 よりも小さい」という条件を記述することができます。

7.3.3　論理和演算子

| は、「論理和演算子」(logical OR operator) と呼ばれます。これは、二つの条件のうちの少なくとも一つが成り立っているかどうかを判断したいとき、つまり、A または B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
1  |  1    ⟶  1
1  |  0    ⟶  1
0  |  1    ⟶  1
0  |  0    ⟶  0
```

たとえば、

```
X == 100 | Y == 100
```

という式を書くことによって、「X と 100 とが等しいか、または、Y と 100 とが等しい」という条件を記述することができます。

7.3.4　論理否定演算子

! は、「論理否定演算子」(logical negation operator) と呼ばれます。これは、真偽値を反転させたいとき、つまり、A ではないという条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
! 1    ⟶  0
! 0    ⟶  1
```

7.4　繰り返しの記述

7.4.1　繰り返しの記述の基礎

POV-Ray では、繰り返しをあらわす記述は、

```
#while ( 式 )
    動作の記述
#end
```

と書きます。この記述の中の「式」のところには、何らかの条件をあらわす式を書きます。そして、「動作の記述」のところには、何らかの動作をあらわす記述を書きます。

繰り返しをあらわす記述は、次の動作を実行します。

(1)　「式」を評価する。その値が偽ならば、繰り返しを終了する。

(2)　「式」の値が真ならば、「動作の記述」を実行する。

(3)　(1) に戻る。

つまり、「式」があらわしている条件が成り立っているあいだだけ、「動作の記述」の実行を繰り返す、ということです。たとえば、

```
#while (X < 10)
    動作の記述
#end
```

という記述を書くと、「動作の記述」は、X < 10 という条件が成り立っているあいだだけ繰り返されることになります。ですから、

```
#declare X = 0;
#while (X < 10)
    動作の記述
    #declare X = X + 1;
#end
```

という記述を書くと、X は、0、1、2、3、4、……と変化していって、X が 10 になったときに X < 10 という条件が成り立たなくなりますので、「動作の記述」は、10 回だけ繰り返されることになります。

7.4.2 物体の列

繰り返しのたびに物体の位置が変化するように物体を作る記述を繰り返すと、物体を並べた列ができます。

次のシーンは、x 軸に沿って 10 個の球を並べた列を作っています。

シーンの例 sequence.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, 0, 0>, 1 }
        pigment { color rgb 1 }
    }
    #declare X = X + 2;
#end
```

7.4.3 色を変化させる繰り返し

物体の列を作る場合、繰り返しの中で物体の色を少しずつ変化させることも可能です。

次のシーンは、色を赤から黄色へ少しずつ変化させながら、10 個の球を作っています。

シーンの例 colorseq.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare G = 0;
#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, 0, 0>, 1 }
        pigment { color rgb <1, G, 0> }
    }
    #declare G = G + 0.1;
    #declare X = X + 2;
#end
```

7.4.4　穴の列

1 個の形状から形状の列を減算することによって、穴の列を持つ形状を作ることができます。

次のシーンは、10 個の円形の穴を持つ板を作っています。

シーンの例　holeseq.pov

```
#declare HoleSequence = difference {
    box { <-2, -0.1, -2>, <20, 0.1, 2> }
    union {
        #declare X = 0;
        #while (X < 20)
            object { cylinder { <X, -1, 0>, <X, 1, 0>, 0.5 } }
            #declare X = X + 2;
        #end
    }
}

camera {
    location <9, 12, -14>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 40, -50> color rgb 1.6 }

object {
    HoleSequence
    pigment { color rgb 1 }
}
```

7.5　繰り返しの入れ子

7.5.1　繰り返しの入れ子の基礎

繰り返しをあらわす記述は、それ自体もまた、動作をあらわしている記述です。したがって、繰り返しをあらわす記述の中に繰り返すをあらわす記述を書くということ、つまり、繰り返しをあらわす記述を入れ子にする、ということも可能です。

7.5.2　物体の列の列

物体の列を作る繰り返しを二重の入れ子にすることによって、物体の列から構成される列を作ることができます。

次のシーンは、x 軸に沿って 10 個の球を並べた列を、y 軸に沿って 10 個だけ並べた列を作っています。

シーンの例　sequsequ.pov

```
camera {
    location <9, 9, -22>
    look_at <9, 9, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare Y = 0;
#while (Y < 20)
    #declare X = 0;
    #while (X < 20)
        object {
            sphere { <X, Y, 0>, 1 }
            pigment { color rgb 1 }
        }
        #declare X = X + 2;
    #end
    #declare Y = Y + 2;
#end
```

7.5.3 物体の列の列の列

物体の列を作る繰り返しを三重の入れ子にすることによって、物体の列から構成される列から構成される列を作ることができます。

次のシーンは、x 軸に沿って 10 個の球を並べた列を、y 軸に沿って 10 個だけ並べた列を、z 軸に沿って 10 個だけ並べた列を作っています。

シーンの例 seqseqseq.pov

```
camera {
    location <-8, 24, -18>
    look_at <4, 12, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare Z = 0;
#while (Z < 20)
    #declare Y = 0;
    #while (Y < 20)
        #declare X = 0;
        #while (X < 20)
            object {
                sphere { <X, Y, Z>, 1 }
                pigment { color rgb 1 }
            }
            #declare X = X + 2;
        #end
        #declare Y = Y + 2;
    #end
    #declare Z = Z + 2;
#end
```

7.6 移動の繰り返し

7.6.1 移動の繰り返しの基礎

球の列を作る場合は、プリミティブの記述を、その中の座標を変化させながら繰り返せばいいわけですが、列を作るためのこの方法は、すべてのプリミティブについて使うことができるわけではありません。なぜなら、トーラスやテキストのような、位置が固定されているプリミティブも存在するからです。

また、CSG を使った複雑な形状を持つ物体の列を、プリミティブを作る記述の中の座標を変化させて作るというのは、かなり困難なことです。

物体の列は、移動の記述を書いて、その移動先の座標を繰り返しのたびに変化させる、という方法でも作ることが可能です。位置が固定されているプリミティブの列を作るためには、この方法を使う必要があります。また、複雑な形状を持つ物体の列は、この方法を使うほうが、プリミティブの座標を変化させるよりも簡単に作ることができます。

7.6.2 トーラスの列

トーラスというのは、原点を中心とする位置に固定されたプリミティブですので、その列を作るためには、移動の記述を書いて、繰り返しのたびに移動先の座標を変化させる、という方法を使う必要があります。

次のシーンは、x 軸に沿って 10 個のトーラスを並べた列を作っています。

シーンの例 torusseq.pov

```
camera {
    location <9, 10, -14>
    look_at <9, 0, 0>
    angle 70
}
```

```
light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 20)
    object {
        torus { 1.6, 0.2 }
        pigment { color rgb 1 }
        translate <X, 0, 0>
    }
    #declare X = X + 2;
#end
```

7.6.3　ハンマーの列

複雑な形状を持つ物体の列を作る場合も、プリミティブの座標を変化させる方法よりも、移動の記述を使う方法のほうが、簡単に作ることができます。

次のシーンは、x 軸に沿って 10 個のハンマーを並べた列を作っています。

シーンの例　hammerseq.pov

```
#declare Hammer = union {
    cylinder { <-2, 3, 0>, <2, 3, 0>, 1 }
    box { <-0.3, 5, -0.2>, <0.3, -5, 0.2> }
}

camera {
    location <27, 0, -50>
    look_at <27, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 60)
    object {
        Hammer
        pigment { color rgb 1 }
        translate <X, 0, 0>
    }
    #declare X = X + 6;
#end
```

7.7　回転の繰り返し

7.7.1　回転の繰り返しの基礎

回転の角度を少しずつ変えながら物体を作る記述を繰り返すことによって、角度が少しずつ異なるいくつかの物体を作ることができます。

7.7.2　円に沿った列

物体は、原点を中心にして回転します。ですから、原点にある物体を回転させると、その物体は、位置を変化させないで回転することになります。

それに対して、原点から離れた位置にある物体を回転させると、その物体は、原点を中心とする円に沿って移動することになります。

次のシーンは、x 軸の方向に原点から 5 だけ離れた位置にある球を、30 度ずつ回転させながら

シーンの例　circle.pov

```
camera {
    location <0, 0, -14>
    look_at 0
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }
```

```
#declare Theta = 0;
#while (Theta < 360)
    object {
        sphere { <5, 0, 0>, 1 }
        pigment { color rgb 1 }
        rotate <0, 0, Theta>
    }
    #declare Theta = Theta + 30;
#end
```

7.7.3 渦巻

蚊取り線香のような、回転するにつれて中心から遠ざかっていく 2 次元の曲線は、「渦巻」(spiral) と呼ばれます。

次のシーンは、渦巻に沿って球を並べています。

シーンの例 spiral.pov

```
camera {
    location <0, 0, -80>
    look_at 0
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#declare Theta = 0;
#while (Theta < 1000)
    object {
        sphere { <X, 0, 0>, 1 }
        pigment { color rgb 1 }
        rotate <0, 0, Theta>
    }
    #declare X = X + 0.4;
    #declare Theta = Theta + 11;
#end
```

7.7.4 螺旋

回転しながら、回転の面に対して垂直な方向へ移動していく 3 次元の曲線は、「螺旋」(helix) と呼ばれます。

次のシーンは、螺旋に沿って球を並べています。

シーンの例 helix.pov

```
camera {
    location <20, 0, -40>
    look_at <20, 0, 0>
    angle 70
}

light_source { <-20, 0, -50> color rgb 1.6 }

#declare X = 0;
#declare Theta = 0;
#while (Theta < 1800)
    object {
        sphere { <X, 10, 0>, 1 }
        pigment { color rgb 1 }
        rotate <Theta, 0, 0>
    }
    #declare X = X + 0.3;
    #declare Theta = Theta + 15;
#end
```

7.8　選択の記述

7.8.1　選択の記述の基礎

POV-Ray では、選択をあらわす記述は、

```
#if ( 式 )
    動作の記述 1
#else
    動作の記述 2
#end
```

と書きます。この記述の中の「式」のところには、何らかの条件をあらわす式を書きます。そして、「動作の記述 $_1$」と「動作の記述 $_2$」のところには、何らかの動作をあらわす記述を書きます。

選択をあらわす記述を動作すると、まず最初に「式」が評価されます。そして、その値が真ならば「動作の記述 $_1$」が実行されて、偽ならば「動作の記述 $_2$」が実行されます。つまり、「式」があらわしている条件が成り立っているかどうかということによって、二つの動作のうちのどちらかが実行される、ということです。たとえば、

```
#if (X < 10)
    ほげほげ
#else
    もげもげ
#end
```

という記述を書くと、`X < 10` という条件が成り立っているならば「ほげほげ」が実行されて、その条件が成り立っていないならば「もげもげ」が実行される、ということになります。

次のシーンは、x 軸に沿って 10 個の球を並べた列を作っているのですが、x 座標が 10 未満のものは色を青色にして、10 以上のものは色を赤色にしています。

シーンの例　`select.pov`

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, 0, 0>, 1 }
        #if (X < 10)
            pigment { color rgb <0, 0, 1> }
        #else
            pigment { color rgb <1, 0, 0> }
        #end
    }
#declare X = X + 2;
#end
```

7.8.2　動作をするかしないかの選択

二つの動作のうちのどちらかを選択するのではなくて、ひとつの動作を実行するかしないかを選択したい、という場合は、

```
#if ( 式 )
    動作の記述
#end
```

という記述を書きます。そうすると、「式」の値が真ならば「動作の記述」が実行されますが、偽ならば何も実行されません。たとえば、

```
#if (X < 10)
    ほげほげ
#end
```

という記述を書くと、X < 10 という条件が成り立っているならば「ほげほげ」が実行されて、その条件が成り立っていないならば何も実行されない、ということになります。

次のシーンは、x 軸に沿って 10 個の球を並べた列を作っているのですが、x 座標が 12 のものだけ、y 軸の方向へ 2 だけ移動させています。

シーンの例 doornotdo.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, 0, 0>, 1 }
        pigment { color rgb 1 }
        #if (X = 12)
            translate <0, 2, 0>
        #end
    }
#declare X = X + 2;
#end
```

7.9 多肢選択

7.9.1 多肢選択の基礎

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。

7.9.2 多肢選択の記述

POV-Ray では、多肢選択をあらわす記述は、

```
#switch ( 式 )
    選択肢
      •
      •
      •
#end
```

と書きます。この記述の中の「式」のところには、何らかの数値を求める式を書きます。そして、「選択肢」のところには、いくつかの選択肢を書きます。

選択肢としては、次の 3 種類の記述を書くことができます。

一致選択肢	式の値が一致したときに選択される。
範囲選択肢	式の値が範囲内にあるときに選択される。
デフォルト選択肢	これ以外のどの選択肢も選択されなかったときに選択される。

7.9.3 一致選択肢

一致選択肢は、

```
#case ( 式 )
    動作の記述
#break
```

と書きます。この記述の中の「式」のところには、何らかの数値を求める式を書きます。「動作の記述」のところには、何らかの動作をあらわす記述を書きます。

一致選択肢は、#switch の右側の式の値と、#case の右側の式の値とが一致したときに選択される選択肢です。

次のシーンは、x 軸に沿って 10 個の球を並べた列を作っているのですが、x 座標が 6 のものは z 軸を回転軸として 90 度だけ回転させて、x 座標が 10 のものは y 軸の方向へ 2 だけ移動させて、x 座標が 14 のものは y 軸の方向に 2 倍だけ拡大しています。

シーンの例　case.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, 0, 0>, 1 }
        pigment { color rgb 1 }
        #switch (X)
            #case (6)
                rotate 90*z
            #break
            #case (10)
                translate <0, 2, 0>
            #break
            #case (14)
                scale <1, 2, 1>
            #break
        #end
    }
#declare X = X + 2;
#end
```

7.9.4　範囲選択肢

範囲選択肢は、

```
#range ( 式1 , 式2 )
    動作の記述
#break
```

と書きます。この記述の中の「式 $_1$」のところには範囲の下限を求める式、「式 $_2$」のところには範囲の上限を求める式を書きます。「動作の記述」のところには、何らかの動作をあらわす記述を書きます。

範囲選択肢は、#switch の右側の式の値が、#case の右側の二つの式によって指定された範囲の中にあるときに選択される選択肢です（下限または上限と一致する場合も選択されます）。

次のシーンは、x 軸に沿って 10 個の球を並べた列を作っているのですが、x 座標が 4 のものは y 軸の方向に 0.5 倍だけ拡大して、x 座標が 6 から 12 までのものは y 軸の方向に 2 倍だけ拡大して、x 座標が 14 のものは y 軸の方向に 0.5 倍だけ拡大しています。

シーンの例　range.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }
```

```
#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, 0, 0>, 1 }
        pigment { color rgb 1 }
        #switch (X)
            #case (4)
                scale <1, 0.5, 1>
            #break
            #range (6, 12)
                scale <1, 2, 1>
            #break
            #case (14)
                scale <1, 0.5, 1>
            #break
        #end
    }
#declare X = X + 2;
#end
```

7.9.5 デフォルト選択肢

デフォルト選択肢は、多肢選択の最後の選択肢として、

```
    #else
        動作の記述
```

と書きます。「動作の記述」のところには、何らかの動作をあらわす記述を書きます。

多肢選択の最後の選択肢としてデフォルト選択肢を書いておくと、これ以外のどの選択肢も選択されなかった場合、この選択肢が選択されて、その中の「動作の記述」が実行されます。

次のシーンは、x 軸に沿って 10 個の球を並べた列を作っているのですが、球の色を、x 座標が 6 のものは赤色、x 座標が 10 のものはオレンジ色、x 座標が 14 のものは黄色、その他のものは水色にしています。

シーンの例 else.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, 0, 0>, 1 }
        #switch (X)
            #case (6)
                pigment { color rgb <1, 0, 0> }
            #break
            #case (10)
                pigment { color rgb <1, 0.5, 0> }
            #break
            #case (14)
                pigment { color rgb <1, 1, 0> }
            #break
            #else
                pigment { color rgb <0, 1, 1> }
        #end
    }
#declare X = X + 2;
#end
```

第 8 章　関数とマクロ

8.1　関数とマクロの基礎

8.1.1　関数とは何か

POV-Ray では、「関数」(function) と呼ばれるものを使うことができます。

関数というのは、式を書くことによって実行させることのできる、何らかの動作をするもののことです。関数を動作させることを、関数を「呼び出す」(call) と言います。

関数は名前によって識別されます。関数に与えられた名前は、「関数名」(function name) と呼ばれます。

8.1.2　マクロとは何か

POV-Ray では、関数によく似た、「マクロ」(macro) と呼ばれるものを使うこともできます。

マクロも、関数と同じように、何らかの動作をします。マクロを動作させることを、マクロを「呼び出す」(call) と言います。

マクロは名前によって識別されます。マクロに与えられた名前は、「マクロ名」(macro name) と呼ばれます。

マクロを呼び出すという動作をあらわす記述は、「マクロ呼び出し」(macro call) と呼ばれます。

8.1.3　引数

関数とマクロは、その動作に先立って、自分を呼び出した側から何個かのデータを受け取って、そのデータを処理するという動作をします。関数とマクロが受け取るデータは、「引数」(argument) と呼ばれます。

8.1.4　戻り値

関数は、引数を処理することによって何らかのデータを求めるという動作をします。そして関数は、処理の結果として得られたデータを、自分を呼び出した側に返します。関数が返すデータは、「戻り値」(return value) と呼ばれます。

マクロは、その動作が式によって記述されている場合は戻り値を返しますが、式ではないものによって動作が記述されている場合は戻り値を返しません。

8.1.5　組み込み関数とユーザー定義関数

関数は、組み込み関数とユーザー定義関数に分類することができます。

「組み込み関数」(built-in function) というのは、POV-Ray の中に最初から組み込まれている関数のことです。POV-Ray のユーザーは、何も定義を書かなくても、組み込み関数を呼び出すことができます。

POV-Ray のユーザーが使うことのできる関数は、組み込み関数だけではありません。ユーザーは、自分が必要とする関数を自由に定義することができます。POV-Ray のユーザーが定義した関数は、「ユーザー定義関数」(user-defined function) と呼ばれます。

マクロを定義するときに、その動作を式によって記述すると、そのマクロはユーザー定義関数になります。

8.1.6　組み込み関数の分類

POV-Ray の組み込み関数は、戻り値がどのようなデータなのかということによって、次の 3 種類に分類することができます。

- 数値関数 (numeric function)
- 文字列関数 (string function)
- ベクトル関数 (vector function)

名前から分かるとおり、数値関数というのは戻り値として数値を返す関数のことで、文字列関数というのは戻り値として文字列を返す関数のことで、ベクトル関数というのは戻り値としてベクトルを返す関数のことです。

数値関数は第 8.2 節で、文字列関数は第 8.3 節で紹介することにしたいと思います。ベクトル関数は、このチュートリアルでは扱いませんので、必要ならば POV-Ray の公式サイトにあるリ

ファレンスマニュアルを参照してください。

8.1.7 関数呼び出し

関数を呼び出すためには、それを呼び出すという動作をあらわす式を書く必要があります。関数を呼び出すという動作をあらわす式は、「関数呼び出し」(function call) と呼ばれます。

関数呼び出しは、

関数名 (式 , ···)

と書きます。この中の「関数名」のところには、関数の名前を書きます。そして、丸括弧の中には、式をコンマ (,) で区切って並べます。

関数呼び出しは式ですから、評価することができます。関数呼び出しを評価すると、関数名で指定された関数が呼び出されて、丸括弧の中の式の値が、その関数に引数として渡されます。そして、関数が戻り値として返したデータが、関数呼び出しの値になります。

例として、pow という数値関数を呼び出す関数呼び出しを書いてみましょう。pow は、A と B という二つの数値を引数として受け取って、A の B 乗を戻り値として返す関数です。たとえば、

```
pow(2, 4)
```

という関数呼び出しを書いたとすると、呼び出された pow は、2 の 4 乗を求めて、その結果を戻り値として返しますので、この関数呼び出しの値は 16 になります。

8.1.8 マクロ呼び出し

マクロを呼び出すためには、それを呼び出すという動作をあらわす記述を書く必要があります。マクロを呼び出すという動作をあらわす記述は、「マクロ呼び出し」(macro call) と呼ばれます。

マクロ呼び出しは、関数呼び出しと同じように、

マクロ名 (記述 , ···)

と書きます。この中の「マクロ名」のところには、マクロの名前を書きます。そして、丸括弧の中には、何らかの記述をコンマ (,) で区切って並べます。

マクロ呼び出しを実行すると、マクロ名で指定されたマクロが呼び出されて、丸括弧の中の記述が、そのマクロに引数として渡されます。たとえば、

```
Namako(27, 61)
```

というマクロ呼び出しは、Namako というマクロを呼び出して、それに対して引数として 27 と 61 を渡します。

戻り値を返すマクロを呼び出すマクロ呼び出しは、関数呼び出しだと考えることができます。戻り値を返すマクロを呼び出した場合、そのマクロが戻り値として返したデータは、そのマクロ呼び出しの値になります。

8.1.9 マクロと関数との相違点

マクロと関数との相違点は、マクロのほうが関数よりも動作の自由度が高いということです。関数は、データを処理して、その結果を戻り値として返す、という動作しかできませんが、マクロは、物体を作ったりするなど、関数よりも広範囲の動作をすることができます。

関数には、POV-Ray の中に最初から組み込まれている、組み込み関数と呼ばれるものがありますが、それに対して、POV-Ray の中に最初から組み込まれているマクロというものは存在しません。POV-Ray では、すべてのマクロは、ユーザーによって定義されたものです。

8.2 数値関数

8.2.1 整数除算

POV-Ray で割り算（除算）をしたいときは、/ という算術演算子を使えばいいわけですが、この演算子は、割り算が整数の範囲で割り切れなかった場合、小数点以下も求めた商を結果として出します。

POV-Ray では、整数の範囲での割り算（整数除算）をしたいという場合のために、次のような組み込み関数が定義されています。

div(A, B)　A を B で整数除算して、その商を返す。

mod(A, B)　A を B で整数除算して、そのあまりを返す。

次のシーンは、x 軸に沿って 25 個の球を並べた列を作っているのですが、x 座標が 6 で割り切れるものは色を黄色にして、割り切れないものは色を赤色にしています。

シーンの例　mod.pov

```
camera {
    location <24, 0, -40>
    look_at <24, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#while (X < 50)
    object {
        sphere { <X, 0, 0>, 1 }
        #if (mod(X, 6) = 0)
            pigment { color rgb <1, 1, 0> }
        #else
            pigment { color rgb <1, 0, 0> }
        #end
    }
#declare X = X + 2;
#end
```

8.2.2　擬似乱数列

ランダムな数値の列は、「乱数列」(random number sequence) と呼ばれます。

コンピュータは、あらかじめ定められたとおりにしか動作できませんので、コンピュータに真の乱数列を生成させることは不可能です。しかし、擬似的な乱数列を生成させることは可能で、コンピュータによって生成された擬似的な乱数列は、「擬似乱数列」(pseudorandom number sequence) と呼ばれます。

POV-Ray は、擬似乱数列を生成する seed という組み込み関数を持っています。この関数は、擬似乱数列の種 (seed) となるプラスの整数を引数として受け取って、擬似乱数列を生成して、それを識別する整数を戻り値として返します。

seed が生成する擬似乱数列は、0 から 1 までの区間のランダムな数値から構成されています。擬似乱数列から個々の数値を取り出したいときは、rand という組み込み関数を使います。この関数は、擬似乱数列を識別する整数（seed の戻り値）を引数として受け取って、その擬似乱数列から 1 個の数値を取り出して、それを戻り値として返します。

次のシーンは、x 軸に沿って 10 個の球を並べた列を作っているのですが、それぞれの球の y 座標は、0 から 2 までのランダムな数値になっています。

シーンの例　rand.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare R = seed(91);
#declare X = 0;
#while (X < 20)
    object {
        sphere { <X, rand(R)*2, 0>, 1 }
        pigment { color rgb 1 }
    }
    #declare X = X + 2;
#end
```

擬似乱数列の種（seed に渡す引数）をいろいろと変更してレンダリングしてみましょう。
次のシーンは、位置と半径と色がランダムに指定された 1000 個の球を作っています。

シーンの例 rand2.pov

```
camera {
    location <50, 50, -50>
    look_at <50, 50, 0>
    angle 70
}

light_source { <20, 20, -100> color rgb 1.6 }

#declare RL = seed(63);
#declare RC = seed(127);
#declare N = 0;
#while (N < 1000)
    object {
        sphere {
                <rand(RL)*100, rand(RL)*100, rand(RL)*100>,
                rand(RL)*6
        }
        pigment { color rgb <rand(RC), rand(RC), rand(RC)> }
    }
    #declare N = N + 1;
#end
```

このシーンについても、擬似乱数列の種をいろいろと変更してレンダリングしてみましょう。RL の種を変更すると球の位置と半径が変化して、RC の種を変更すると球の色が変化します。

8.2.3 その他の主要な数値関数

POV-Ray の主要な数値関数としては、div、mod、seed、rand のほかに、次のようなものがあります。

abs(A)	A の絶対値を返す。
acos(A)	A のアークコサイン（逆余弦）を返す。
acosh(A)	A のアークハイパーボリックコサイン（逆双曲線余弦）を返す。
asc(S)	文字列 S の先頭の文字の文字コード（ASCII コード）を返す。
asin(A)	A のアークサイン（逆正弦）を返す。
asinh(A)	A のアークハイパーボリックサイン（逆双曲線正弦）を返す。
atan2(A, B)	A と B の符合を考慮して、y/x のアークタンジェント（逆正接）を返す。
atanh(A)	A のアークハイパーボリックタンジェント（逆双曲線正接）を返す。
ceil(A)	A よりも小さくない最小の整数を返す。
cos(A)	A のコサイン（余弦）を返す。
cosh(A)	A のハイパーボリックコサイン（双曲線余弦）を返す。
degrees(A)	ラジアンであらわされた角度 A を度に変換した結果を返す。
exp(A)	自然対数の底 (2.71828182846) の A 乗を返す。
floor(A)	A よりも大きくない最大の整数を返す。
int(A)	A の小数点以下の部分を切り捨てた結果を返す。
log(A)	A の自然対数の値を返す。
max(···)	任意個の引数のうちで最大のものを返す。
min(···)	任意個の引数のうちで最小のものを返す。
pow(A, B)	A の B 乗を返す。
radians(A)	度であらわされた角度 A をラジアンに変換した結果を返す。
strcmp(S, T)	文字列 S と T を比較して、辞書式順序で S が後ろで T が前ならばプラスの数値、S が前で T が後ろならばマイナスの数値、S と T が同じものならば 0 を返す。
strlen(S)	文字列 S の長さ（含まれる文字の個数）を返す。

sin(A)　　A のサイン（正弦）を返す。
sinh(A)　　A のハイパーボリックサイン（双曲線正弦）を返す。
sqrt(A)　　A の平方根を返す。
tan(A)　　A のタンジェント（正接）を返す。
tanh(A)　　A のハイパーボリックタンジェント（双曲線正接）を返す。
val(S)　　数値をあらわしている文字列 S を数値に変換した結果を返す。
vlength(V)　　ベクトル V の長さを返す。

三角関数が扱う角度の単位は、ラジアンです。

8.3　文字列関数

8.3.1　数値から文字列への変換

str という POV-Ray の組み込み関数は、数値を文字列に変換した結果を戻り値として返します。

str は、次のような 3 個の引数を受け取ります。

1 個目　文字列に変換される数値。
2 個目　戻り値として返す文字列の最短の長さ。変換の結果がこの長さよりも短い場合は、空白が左側に充填される。マイナスの整数を渡すと、空白ではなく 0 が左側に充填される。
3 個目　文字列に変換する小数点以下の桁数。

次のシーンは、2^0、2^1、2^2、……、2^{16} を物体にしています。

シーンの例　str.pov

```
camera {
    location <-6, 0, -17>
    look_at <0, -6, 0>
    angle 70
}

light_source { <-50, 40, -50> color rgb 1.6 }

#declare P = 1;
#declare Y = 0;
#while (P < 100000)
    object {
        text { ttf "timrom.ttf", str(P, 0, 0), 0.2, 0 }
        pigment { color rgb 1 }
        translate <0, Y, 0>
    }
#declare P = P * 2;
#declare Y = Y - 1;
#end
```

8.3.2　その他の文字列関数

POV-Ray の文字列関数としては、str のほかに、次のようなものがあります。

chr(N)　　文字コードが N の文字から構成される長さが 1 の文字列を返す。
concat(…)　　任意個の引数（文字列）を連結した結果を返す。
strlwr(S)　　S に含まれるすべての大文字を小文字に変換した結果を返す。
strupr(S)　　S に含まれるすべての小文字を大文字に変換した結果を返す。
substr(S, P, L)　　S から部分文字列を取り出して、その結果を返す。P は取り出す部分文字列の先頭の位置を示す整数（先頭の文字を 1 と数える）。L は取り出す部分文字列の長さ。
vstr(N, A, S, L, P)　　ベクトル A を文字列に変換した結果を返す。N はベクトルの次元。S は成分を区切る文字列。L は、個々の成分を変換してできる文字列の最短の長さ。変換の結果がこの長さよりも短い場合は、空白が左側

に充填される。マイナスの整数を渡すと、空白ではなく 0 が左側に充填される。P は小数点以下の桁数。

8.4 マクロ

8.4.1 マクロ宣言

マクロは、「マクロ宣言」(macro declaration) と呼ばれる宣言を書くことによって定義することができます。

マクロ宣言は、

```
#macro マクロ名( 仮引数, ··· )
    動作の記述
#end
```

と書きます。「マクロ名」のところには、マクロに与えたい識別子を書いて、「動作の記述」のところには、マクロの動作となる記述を書きます。「仮引数」のところには、引数に与える識別子を書きます。仮引数は、マクロが実行されているあいだだけ、マクロが受け取った引数に、名前として与えられます。たとえば、

```
#macro Ball(Radius)
    sphere { 0, Radius }
#end
```

というマクロ宣言を書くことによって、受け取った引数を半径とする球を原点に作る、Ball というマクロを定義することができます。このマクロを、

```
Ball(7)
```

というマクロ呼び出しで呼び出すと、Radius という仮引数が 7 という数値に名前として与えられますので、半径が 7 の球が原点に作られることになります。

8.4.2 仮引数と引数との対応

仮引数と引数とは、マクロ宣言の丸括弧の中で仮引数が並んでいる順序と、マクロ呼び出しの丸括弧の中で式が並んでいる順序とが一致するように対応づけられます。たとえば、

```
#macro Hoge(A, B, C)
    動作の記述
#end
```

というマクロを、

```
Hoge(5, 7, 3)
```

というマクロ呼び出しで呼び出したとすると、A は 5 を、B は 7 を、C は 3 を受け取ることになります。

次のシーンは、3 個のトーラスを作ります。

シーンの例 macrotorus.pov

```
#macro Torus(Radius1, Radius2, Rotate, Translate, Color)
    object {
        torus { Radius1, Radius2 }
        pigment { color rgb Color }
        rotate Rotate
        translate Translate
    }
#end

camera {
    location <0, 12, -40>
    look_at 0
    angle 70
}
```

```
light_source { <50, 50, -50> color rgb 1.6 }

Torus(10, 1, 0, <-12, 0, 0>, <1, 1, 0>)
Torus(10, 1, 0, <12, 0, 0>, <0, 1, 1>)
Torus(6, 0.7, 90*x, 0, <0, 1, 0>)
```

このシーンの中で定義されている Torus は、トーラスを作るマクロです。このマクロは、次のような引数を受け取ります。

(1) トーラス全体の半径。
(2) 円管の断面の半径。
(3) 回転角（3 次元ベクトル）。
(4) 中心の位置の座標（3 次元ベクトル）。
(5) 色（3 次元ベクトル）。

8.4.3　マクロ宣言の中での識別子の宣言

マクロ宣言の中には、識別子の宣言を書くことも可能です。ただし、マクロ宣言の中では、識別子の宣言は、通常、

```
#local 識別子 = 記述
```

と書きます。つまり、#declare と書くのではなくて、#local と書くわけです。

#local を使って宣言された識別子は、#declare の場合とは違って、それが名前として何かに与えられるのは、マクロが実行されているあいだだけです。したがって、その識別子は、マクロ宣言の外にはいかなる影響も与えません。もしもマクロ宣言の外で同じ識別子が使われていたとしても、それらの識別子は互いにまったく無関係だとみなされます。ですから、マクロ宣言を書くときには、識別子の宣言に #local を使っている限り、識別子の重複を気にする必要はありません。

マクロの仮引数も、それが引数に名前として与えられるのは、マクロが実行されているあいだだけですので、#local を使って宣言された識別子と同じように、マクロ宣言の外にある識別子との重複は、気にしなくても大丈夫です。

次のシーンは、3 列の球の列を作ります。

シーンの例　macrosequence.pov

```
#macro Sequence(N, Radius, Rotate, Translate, Color)
    #local X = 0;
    #while (X < N * Radius * 2)
        union {
            object {
                sphere { <X, 0, 0>, Radius }
            }
            pigment { color rgb Color }
            rotate Rotate
            translate Translate
        }
        #local X = X + Radius * 2;
    #end
#end

camera {
    location <6, 8, -30>
    look_at <6, 8, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

Sequence(10, 1, 0, 0, <0.5, 1, 0>)
Sequence(20, 0.5, 0, <0, 16, 0>, <0, 0.5, 1>)
Sequence(5, 2, 90*z, <-6, 0, 0>, <1, 0.5, 0>)
```

このシーンの中で定義されている Sequence は、球を並べた列を作るマクロです。このマクロは、次のような引数を受け取ります。

(1) 球の個数。
(2) 球の半径。
(3) 回転角（3 次元ベクトル）。
(4) 先頭の球の座標（3 次元ベクトル）。
(5) 色（3 次元ベクトル）。

8.4.4 ユーザー定義関数

第 8.1.5 項で説明したように、動作が式によって記述されたマクロを定義すれば、そのマクロはユーザー定義関数になります。

次のシーンは、ランダムな角度、ランダムな位置、ランダムな色で、200 個のランダムな英字の大文字を作っています。

シーンの例　alphabet.pov

```
#macro RandAlpha(R)
    chr(int(rand(R)*25)+65)
#end

#macro RandColor(R)
    <rand(R), rand(R), rand(R)>
#end

#macro RandRotate(R)
    <rand(R)*360, rand(R)*360, rand(R)*360>
#end

#macro RandTranslate(R, D)
    <rand(R)*D, rand(R)*D, rand(R)*D>
#end

camera {
    location <50, 50, -40>
    look_at <50, 50, 0>
    angle 70
}

light_source { <20, 20, -100> color rgb 1.6 }

#declare RA = seed(127);
#declare RC = seed(255);
#declare RR = seed(511);
#declare RT = seed(1023);
#declare N = 0;
#while (N < 200)
    object {
        text { ttf "timrom.ttf", RandAlpha(RA), 0.2, 0 }
        pigment { color rgb RandColor(RC) }
        scale 20
        rotate RandRotate(RR)
        translate RandTranslate(RT, 100)
    }
    #declare N = N + 1;
#end
```

このシーンの中では、次のような関数を定義しています。

RandAlpha(R)	擬似乱数列 R を使って生成したランダムな英字の大文字を返す。
RandColor(R)	擬似乱数列 R を使って生成したランダムな色をあらわす 3 次元ベクトルを返す。
RandRotate(R)	擬似乱数列 R を使って生成したランダムな回転角をあらわす 3 次元ベクトルを返す。

RandTranslate(R, D) 擬似乱数列 R を使って生成した、一辺が D の立方体の中にあるランダムな座標を返す。

第 9 章　アニメーション

9.1　アニメーションの基礎

9.1.1　POV-Ray とアニメーション

まず多数の静止画を描いて、そののちそれらを時間軸の上に並べる、という方法によって作られた動画は、「アニメーション」(animation) と呼ばれます。

アニメーションの素材となる個々の静止画は、「フレーム」(frame) と呼ばれます。

POV-Ray は、シーンを少しずつ変化させながら多数のフレームを作るという機能を持っています。この機能を使って作られたフレームを時間軸の上に並べれば、アニメーションができることになります。

9.1.2　フレーム数

POV-Ray のツールバーの下には、「コマンドライン」(command line) と呼ばれる入力欄があります。アニメーションの素材となるフレームを作りたいときは、この入力欄に、

+KFF フレーム数

という形のものを入力します。「フレーム数」のところには、プラスの整数を書きます。その状態で Run のボタンをクリックすると、入力した整数で指定された回数だけレンダリングが実行されて、その枚数の静止画が作られます。

ただし、レンダリングのたびに何らかの変化が生じるようにシーンが書かれていなければ、ただ単に、まったく同じ静止画が何枚も作成されるだけです。

9.1.3　clock

レンダリングのたびに変化が生じるようにするためには、clock という識別子をシーンの中に書いておく必要があります。

clock という識別子は、0 から 1 までの数値に与えられます。そして、その数値は、レンダリングのたびに変化します。1 回目のレンダリングのときの clock は 0 で、最後のレンダリングのときの clock は 1 です。そのあいだのレンダリングでは、clock は、0 と 1 のあいだを等間隔に刻んだ数値に与えられます。

たとえば、フレーム数が 6 枚の場合、clock は、

0、0.2、0.4、0.6、0.8、1

と変化します。同じように、フレーム数が 9 枚の場合、clock は、

0、0.125、0.25、0.375、0.5、0.625、0.75、0.875、1

と変化します。

それでは、コマンドラインに +KFF9 と入力して（つまりフレーム数として 9 フレームを指定して）、次のシーンをレンダリングしてみてください。

シーンの例　clock.pov

```
camera {
    location <-0.2, 0, -1.5>
    look_at <0.8, 0.4, 0>
    angle 70
}

light_source { <-50, 40, -50> color rgb 1.6 }

object {
    text { ttf "timrom.ttf", str(clock, 0, 3), 0.2, 0 }
    pigment { color rgb 1 }
}
```

レンダリングの結果は、clock1.bmp、clock2.bmp、clock3.bmp、……というファイルに保存されます。

それぞれのフレームは、clock が与えられている数値を物体にしたものです。1 フレーム目は 0.000、2 フレーム目は 0.125、3 フレーム目は 0.250、……になっているはずです。

9.1.4 フレームレート

POV-Ray は、動画を作成する機能は持っていません。ですから、POV-Ray が作成したフレームを素材にしてアニメーションを作るためには、静止画を時間軸の上に並べることによって動画を作成する機能を持っているソフトを使う必要があります。

動画を作成するソフトにアニメーションを作らせるときには、フレームを時間軸の上に並べる間隔を指定する必要があります。フレームが同じであっても、それらのを並べる間隔が短かければアニメーションは速く進行することになりますし、間隔が長ければ遅く進行することになります。

フレームを時間軸の上に並べる間隔は、「フレームレート」(frame rate) と呼ばれる数値によって指定されます。

フレームレートというのは、1 秒間に並べるフレーム数のことで、fps(frames per second) という単位で示されます。

アニメーションは、フレームレートが小さいと、カクカクとした不自然な動きに見えますが、フレームレートが十分に大きければ、滑らかに動いているように見えます。通常のアニメーションは、24fps で作られます。

9.1.5 動画の作成

静止画から動画を作成する機能を持つソフトには、さまざまなものがあります。ここでは、Stripe というフリーのソフトを使って動画を作成する方法を紹介したいと思います。

Stripe では、次のような操作をすることによって、静止画（形式は BMP または JPEG）から動画（形式は AVI）を作成することができます。

(1) Stripe を起動すると、ウィンドウの中にいくつかのフレームが表示されるので、そのどれかを右クリックする。
(2) クリックした場所にメニューが表示されるので、「フォルダを開く」を選択する。
(3) 「現在編集中のデータは破棄されますがよろしいですか？」というダイアログボックスが表示されるので、「OK」をクリックする。
(4) 「フォルダの参照」というダイアログボックスが表示されるので、フレームが格納されているフォルダーを選択して、「OK」をクリックする。そうすると、そのフォルダの中にあるすべてのフレームが表示される。
(5) メニューから「ツール」→「設定」を選択する。
(6) 「STRIPE の設定」というダイアログボックスが表示されるので、「Frame Rate」という入力欄にフレームレート、「映像サイズ」という入力欄に動画の大きさ（単位はピクセル）を入力して、「OK」をクリックする。
(7) ツールバーの中にある「全て変換」というボタンをクリックする。
(8) 「名前を付けて保存」というダイアログボックスが表示されるので、動画を保存するファイル名の名前を入力して（拡張子は、自動的に .avi になるので書かなくてもよい）、「保存」をクリックする。

それでは、先ほどの clock.pov を 9 フレームでレンダリングして、生成されたフレームから動画を作成してみてください。フレームレートを 3fps に設定すると、3 秒の動画ができるはずです。

9.2 clock の使い方

9.2.1 clock の使い方の基礎

第 9.1.3 項で説明したように、clock という識別子は、レンダリングのたびに変化する数値に与えられます。1 回目のレンダリングのときは 0 で、最後のレンダリングのときは 1 です。

アニメーションを作るとき、変化させたい数値は、必ずしも 0 から 1 までとは限りません。む

しろ、20 から 30 まで変化させたいとか、600 から 400 まで変化させたい、というように、範囲が 0 から 1 までではない場合のほうが多いでしょう。そこで、この節では、0 から 1 までではない範囲で数値を変化させる方法について説明したいと思います。

9.2.2　指定された範囲で数値を変化させる式

1 回目のレンダリングのときにほしい数値を「開始値」、最後のレンダリングのときにほしい数値を「終了値」と呼ぶことにしましょう。そうすると、開始値から終了値までの範囲で変化する数値を求める式は、

clock* 終了値 − 開始値 + 開始値

と書くことができます。

レンダリングの進行とともに数値が大きくなるようにしたい場合も、それとは逆に数値が小さくなるようにしたい場合も、使う式はこれひとつだけです。

9.2.3　数値が増加するアニメーション

それでは、0 から 1 までではない範囲で数値が増加するアニメーションを作ってみましょう。

次のシーンは、20 から 30 までの範囲で増加する数値を物体にします。

シーンの例　ascending.pov

```
camera {
    location <-0.4, 0, -1>
    look_at <0.3, 0.4, 0>
    angle 70
}

light_source { <-50, 40, -50> color rgb 1.6 }

object {
    text { ttf "timrom.ttf", str(clock*10+20, 0, 0), 0.2, 0 }
    pigment { color rgb 1 }
}
```

このシーンを 11 フレームでレンダリングして、作成されたフレームから 3fps で動画を作成してみてください。

9.2.4　数値が減少するアニメーション

次に、数値が減少するアニメーションを作ってみましょう。使う式は先ほどと同じです。

次のシーンは、30 から 20 までの範囲で減少する数値を物体にします。

シーンの例　descending.pov

```
camera {
    location <-0.4, 0, -1>
    look_at <0.3, 0.4, 0>
    angle 70
}

light_source { <-50, 40, -50> color rgb 1.6 }

object {
    text { ttf "timrom.ttf", str(clock*-10+30, 0, 0), 0.2, 0 }
    pigment { color rgb 1 }
}
```

先ほどと同じように、このシーンを 11 フレームでレンダリングして、作成されたフレームから 3fps で動画を作成してみてください。

9.3　物体が移動するアニメーション

9.3.1　物体が移動するアニメーションの基礎

clock から求めた数値を位置として指定された物体は、レンダリングのたびに少しずつ移動することになります。ですから、そのシーンから動画を作れば、物体が移動するアニメーションができることになります。

9.3.2 球が移動するアニメーション

まず手始めに、球が移動するアニメーションを作ってみましょう。

プリミティブの sphere は、中心の座標を記述することに球の位置を決定することができますので、clock から求めた数値を座標として使えば、時間とともに球を移動させることができます。

次のシーンは、球を左から右へ移動させます。

シーンの例 transphere.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { <clock*4-2, 0, 0>, 1 }
    pigment { color rgb 1 }
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.3.3 テキストが移動するアニメーション

次に、テキストが移動するアニメーションを作ってみましょう。

プリミティブの torus や text は、それ自体の位置が固定されていますので、トーラスやテキストが移動するアニメーションを作るためには、移動の記述を使う必要があります。また、移動の記述を使えば、CSG を使って作られた複雑な形状を持つ物体を移動させるアニメーションを作ることも簡単です。

次のシーンは、「Hello, world!」というテキストを右から左へ移動させます。

シーンの例 transtext.pov

```
camera {
    location <0, 2, -5>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    text { ttf "timrom.ttf", "Hello, world!", 0.2, 0 }
    pigment { color rgb 1 }
    translate <clock*-4.7+2, 0, 0>
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.4 形状が変化するアニメーション

9.4.1 形状が変化するアニメーションの基礎

clock から求めた数値を、形状を決定するための数値として指定された物体は、レンダリングのたびに少しずつ形状が変化することになります。ですから、そのシーンから動画を作れば、形状が変化するアニメーションができることになります。

9.4.2　トーラスの半径が変化するアニメーション

まず手始めに、トーラスの半径が変化するアニメーションを作ってみましょう。

トーラスの半径を決定する数値として、clock から求めた数値を指定すれば、時間とともにトーラスの半径を変化させることができます。

次のシーンは、トーラスの半径を 1 から 6 まで変化させます。

シーンの例　radiustorus.pov

```
camera {
    location <0, 5, -10>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    torus { clock*5+1, 0.2 }
    pigment { color rgb 1 }
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.4.3　円柱が圧縮されるアニメーション

変形のアニメーションは、clock から求めた数値を拡大率として指定した拡大の記述を書くことによって作ることも可能です。

次のシーンは、円柱の z 軸方向の拡大率を 1 から 0.1 まで変化させます。

シーンの例　scalecylinder.pov

```
camera {
    location <2, 3, -2>
    look_at <0, 1, 0>
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    cylinder { 0, <0, 2, 0>, 1 }
    pigment { color rgb 1 }
    scale <1, 1, clock*-0.9+1>
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.4.4　物体が増殖するアニメーション

clock から求めた数値を繰り返しの回数として指定した繰り返しの記述を書くことによって、物体が増殖するアニメーションを作ることができます。

次のシーンは、球を 1 個から 10 個まで増殖させます。

シーンの例　whileclock.pov

```
camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

#declare X = 0;
#declare N = 0;
#while (N < clock*9+1)
```

```
    object {
        sphere { <X, 0, 0>, 1 }
        pigment { color rgb 1 }
    }
    #declare X = X + 2;
    #declare N = N + 1;
#end
```

このシーンを 10 フレームでレンダリングして、作成されたフレームから 3fps で動画を作成してみてください。

9.5 物体が回転するアニメーション

9.5.1 物体が回転するアニメーションの基礎

物体を作る記述の中に回転の記述を書いて、clock から求めた数値を回転角として指定すると、その物体は、レンダリングのたびに少しずつ回転することになります。ですから、そのシーンから動画を作れば、物体が回転するアニメーションができることになります。

9.5.2 直方体が回転するアニメーション

直方体が回転するアニメーションを作ってみましょう。

次のシーンは、y 軸を回転軸にして直方体を 1 回転させます。

シーンの例 rotatebox.pov

```
camera {
    location <0, 0, -4>
    look_at 0
    angle 70
}

light_source { <-2, 0, -5> color rgb 1.6 }

object {
    box { <-1.5, -1, -0.2>, <1.5, 1, 0.2> }
    pigment { color rgb 1 }
    rotate clock*360*y
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.5.3 円運動のアニメーション

固定された場所で物体を回転させるためには、先ほどのシーンのように、回転軸がその物体の内部を通過している必要があります。回転軸が物体から離れたところを通過している場合、その物体は、回転に伴って円運動をすることになります。

次のシーンは、y 軸を回転軸にして球を回転させているのですが、球は y 軸から離れた位置にありますので、y 軸を中心にして円運動をすることになります。

シーンの例 curcularmotion.pov

```
camera {
    location <0, 4, -20>
    look_at 0
    angle 70
}

light_source { <-50, 50, -50> color rgb 1.6 }

object {
    sphere { <0, 0, 10>, 1 }
    pigment { color rgb 1 }
    rotate clock*360*y
}
```

このシーンを 96 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.6　色が変化するアニメーション

9.6.1　色が変化するアニメーションの基礎

clock から求めた数値を、原色の比率として色の記述の中で使うと、その色は、レンダリングのたびに少しずつ変化することになります。ですから、そのシーンから動画を作れば、色が変化するアニメーションができることになります。

9.6.2　球の色が変化するアニメーション

球の色が変化するアニメーションを作ってみましょう。

次のシーンは、球の色を赤色から青色へ変化させます。

シーンの例　redblue.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { 0, 1 }
    pigment { color rgb <1-clock, 0, clock> }
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.6.3　フィルターによるアニメーション

clock から求めた数値を、フィルターとして色の記述の中で使うと、光を透過させる比率が、レンダリングのたびに少しずつ変化することになります。ですから、そのシーンから動画を作れば、物体が透明になったり不透明になったりするアニメーションができることになります。

次のシーンは、球の手前に直方体を置いています。その直方体は、最初は不透明ですが、しだいに透明になっていきます。

シーンの例　filterbox.pov

```
camera {
    location <1, 0, -4>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
}

object {
    box { <-0.5, -0.5, -1.5>, <0.5, 0.5, -1.6> }
    pigment { color rgbf <1, 1, 1, clock> }
    interior { ior 1.5 }
}
```

9.7 カメラによるアニメーション

9.7.1 この節について

これまでは、物体が移動したり変形したり回転したりするアニメーションの作り方について説明してきたわけですが、この節では、物体の側の変化によるアニメーションではなく、カメラの側が変化することによるアニメーションの作り方について説明したいと思います。

9.7.2 カメラが移動するアニメーション

clock から求めた数値を位置として指定されたカメラは、レンダリングのたびに少しずつ移動することになります。ですから、そのシーンから動画を作れば、カメラが移動するアニメーションができることになります。

次のシーンは、原点に向けたカメラの x 座標を -4 から 4 まで変化させています。

シーンの例 transcamera.pov

```
camera {
    location <clock*8-4, 2, -3>
    look_at 0
    angle 70
}

light_source { <-5, 4, -5> color rgb 1.2 }
light_source { <5, 4, -5> color rgb 1.2 }

object {
    box { <-1, -1, -1>, <1, 1, 1> }
    pigment { color rgb 1 }
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.7.3 方向を固定したカメラの移動

先ほどのシーンは、常に原点を向くように方向を変化させながらカメラを移動させたわけですが、場合によっては、方向を固定したままカメラを移動させたい、ということもあります。そのような場合は、カメラの記述の中に移動の記述を書いて、clock から求めた数値を移動先として指定します。

次のシーンは、先ほどと同じように、カメラの x 座標を -4 から 4 まで変化させるのですが、カメラは、方向を固定したまま移動します。

シーンの例 transcamera2.pov

```
camera {
    location <0, 2, -3>
    look_at 0
    angle 70
    translate <clock*8-4, 0, 0>
}

light_source { <-5, 4, -5> color rgb 1.2 }
light_source { <5, 4, -5> color rgb 1.2 }

object {
    box { <-1, -1, -1>, <1, 1, 1> }
    pigment { color rgb 1 }
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.7.4 カメラが回転するアニメーション

物体と同じように、カメラも、その記述の中に回転の記述を書くことによって、回転させることができます。

カメラを作る記述の中に回転の記述を書いて、clock から求めた数値を回転角として指定すると、カメラは、レンダリングのたびに少しずつ回転することになります。ですから、そのシーンから動画を作れば、カメラが回転するアニメーションができることになります。

次のシーンは、部屋の中に置かれたカメラを 360 度回転させます。

シーンの例　rotatecamera.pov

```
camera {
    location <0, -1, 0>
    look_at <0, -1, 1>
    angle 70
    rotate clock*360*y
}

light_source { <0, -3, 0> color rgb 1.6 }

difference {
    sphere { 0, 30 }
    box { <-20, -5, -15>, <20, 5, 15> }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

9.7.5　ズームインとズームアウト

カメラの焦点距離を少しずつ長くすること、言い換えれば、カメラの視野の角度を少しずつ狭くすることを、（被写体に）「ズームインする」(zoom in) と言います。逆に、カメラの焦点距離を少しずつ短くすること、言い換えれば、カメラの視野の角度を少しずつ広くすることを、（被写体から）「ズームアウトする」(zoom out) と言います。

clock から求めた数値を、カメラの視野の角度として指定すると、カメラは、レンダリングのたびに視野の角度が変化することになります。ですから、そのシーンから動画を作れば、被写体にズームインしたり、被写体からズームアウトしたりするアニメーションができることになります。

次のシーンは、カメラの視野の角度を 70 度から 3 度まで変化させます。

シーンの例　zoomin.pov

```
camera {
    location <30, 20, -50>
    look_at 0
    angle clock*-67+70
}

light_source { <3, 4, -5> color rgb 1.6 }

object {
    box { <-1, -1, -1>, <1, 1, 1> }
    pigment { color rgb 1 }
}
```

このシーンを 48 フレームでレンダリングして、作成されたフレームから 24fps で動画を作成してみてください。

第 10 章　光源の奥義

10.1　平行光源

10.1.1　さまざまな光源

POV-Ray では、光源の記述をシーンの中に書かないと、ほとんど真っ黒な画像が生成されることになります。ですから、これまでに紹介してきたすべてのシーンには、光源の記述が書かれています。

これまでのシーンに書かれていた光源の記述は、すべて、「点光源」(point light) と呼ばれる光源を作るものです。点光源というのは、特定の一点に存在していて、すべての方向に同じ強さの光を出す光源のことです。

POV-Ray で扱うことのできる光源は、点光源だけではありません。そのほかにも、次のような光源を扱うことができます。

- 平行光源 (parallel light)
- 面光源 (area light)
- スポットライト (spot light)

10.1.2 平行光源の基礎

四方八方に光を振りまくのではなくて、特定の方向にのみ光を出す光源は、「平行光源」(parallel light) と呼ばれます。

太陽の光を POV-Ray で表現したい場合、点光源を太陽の位置に置いてもかまわないわけですが、太陽は、地球から見て十分に遠いところにありますので、平行光源で代用することも可能です。

10.1.3 平行光源の記述

平行光源は、

```
light_source {
    開始位置
    色の記述
    parallel
    point_at 終了位置
}
```

という記述を書くことによって作ることができます。この中の「開始位置」と「終了位置」のそれぞれには、3 次元ベクトルを書きます。そうすると、開始位置から終了位置に向かって光を出す光源が作られます。たとえば、

```
light_source {
    <0, 10, 0>
    color rgb 1.6
    parallel
    point_at <0, 0, 0>
}
```

という記述を書くことによって、真上から真下に向かって光を出す平行光源を作ることができます。

次のシーンは、真上から真下に向かって光を出す平行光源を作っています。

シーンの例 parallel.pov

```
camera {
    location <0, 10, -5>
    look_at <0, 1, 0>
    angle 70
}

light_source {
    <0, 10, 0>
    color rgb 1.6
    parallel
    point_at <0, 0, 0>
}

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}
```

```
object {
    sphere { <0, 5, 0>, 1 }
    pigment { color rgb 1 }
}
```

終了位置をいろいろと変更してレンダリングしてみましょう。

10.2　面光源

10.2.1　面光源の基礎

一点から光を出すのではなくて、有限の大きさの面から光を出す光源は、「面光源」(area light) と呼ばれます。

点光源を使った場合、物体の影の輪郭はきわめてシャープな線になりますが、面光源を使うと、影の輪郭がぼやけたものになります。照明器具として蛍光灯が使われている部屋のシーンを書く場合には、点光源よりも面光源を使うほうがいいでしょう。

POV-Ray の面光源は、長方形の領域の中に点光源を並べることによって作られた擬似的なものです。

10.2.2　面光源の記述

面光源は、

```
light_source {
    位置
    色の記述
    area_light 対角点1, 対角点2, 縦, 横
}
```

という記述を書くことによって作ることができます。この中の「位置」のところには、面光源の位置を書きます。「対角点 $_1$」と「対角点 $_2$」のそれぞれには、光源となる長方形を決定するための 3 次元ベクトルを書きます。そして、「縦」と「横」のそれぞれには、点光源を縦に並べる個数と横に並べる個数を書きます。たとえば、

```
light_source {
    <0, 10, 0>
    color rgb 1.6
    area_light <2, 0, 0>, <0, 0, 2>, 5, 5
}
```

という記述を書くことによって、位置が <0, 10, 0> で、長方形を決定するベクトルが <2, 0, 0> と <0, 0, 2> で、25 個の点光源を 5×5 で並べた面光源を作ることができます。

次のシーンは、面光源を作っています。

シーンの例　arealight.pov

```
camera {
    location <0, 10, -5>
    look_at <0, 1, 0>
    angle 70
}

light_source {
    <0, 10, 0>
    color rgb 1.6
    area_light <2, 0, 0>, <0, 0, 2>, 5, 5
}

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

object {
```

```
    sphere { <0, 5, 0>, 1 }
    pigment { color rgb 1 }
}
```

光源となる長方形を決定するための 3 次元ベクトルや、点光源を縦と横に並べる個数をいろいろと変更してレンダリングしてみましょう。

10.3 スポットライト

10.3.1 スポットライトの基礎

点光源というのは四方八方に光を出すわけですが、それに対して、特定の方向にのみ光を出す光源は、「スポットライト」(spotlight) と呼ばれます。

スポットライトが出す光は、光源から遠ざかるほど広がっていきますので、その光が当たる範囲は、円錐の形になります。

スポットライトの光が広がっていく円錐の中心を通る直線は、「光軸」(light axis) と呼ばれます。

10.3.2 スポットライトの記述

スポットライトは、

```
light_source {
    位置
    色の記述
    spotlight
    point_at 方向点
    radius 減衰開始角度
    falloff 減衰終了角度
    tightness 減衰の急激さ
}
```

という記述を書くことによって作ることができます。この中の「位置」のところには、スポットライトの位置を書きます。「方向点」のところには、光軸を向ける方向を示す点の座標を書きます。

「減衰開始角度」のところには、光が減衰しない範囲を示す、光軸からの角度を、0 度から 90 度までの範囲内で書きます（デフォルトは 30 度)。「減衰終了角度」のところには、少しでも光が当たる範囲を示す、光軸からの角度を、0 度から 90 度までの範囲内で書きます（デフォルトは 45 度)。そうすると、光は、減衰開始角度から外へ出て行くにしたがって少しずつ減衰していって、減衰終了角度のところから外にはまったく当たらなくなります。

「減衰の急激さ」のところには、どれぐらい急激に光を減衰させるかということを示す、0 から 100 までの範囲の数値を書きます。この数値が大きいほど、光は急激に減衰します。この数値のデフォルトは 0 です。

たとえば、

```
light_source {
    <0, 10, 0>
    color rgb 1.6
    spotlight
    point_at 0
    radius 20
    falloff 40
    tightness 10
}
```

という記述を書くことによって、位置が <0, 10, 0> で、光軸が <0, 0, 0> を通る、減衰開始角度が 20 度、減衰終了角度が 40 度、減衰の急激さが 10 のスポットライトを作ることができます。

次のシーンは、スポットライトを作っています。

シーンの例　spotlight.pov

```
camera {
    location <0, 10, -5>
    look_at <0, 1, 0>
    angle 70
}

light_source {
    <0, 10, 0>
    color rgb 1.6
    spotlight
    point_at 0
    radius 20
    falloff 40
    tightness 10
}

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

object {
    sphere { <0, 5, 0>, 1 }
    pigment { color rgb 1 }
}
```

スポットライトの位置、方向点、減衰開始角度、減衰終了角度、減衰の急激さをいろいろと変更してレンダリングしてみましょう。

10.4　発光する物体

10.4.1　発光する物体の基礎

POV-Ray の光源というのは、物体ではありませんので、それにカメラを向けたとしても、光源が存在する場所に何かが見えるということはありません。

POV-Ray には、発光する物体を作るという機能が備わっています。この機能を使うことによって、たとえば照明器具などが視界の中に存在するシーンを書くことができます。

10.4.2　発光する物体の記述

発光する物体を作りたいときは、光源の記述の中に、その物体の記述を書きます。

物体の記述を伴う光源の記述は、

```
light_source {
    位置
    色の記述
    looks_like 物体の記述
}
```

と書きます。この中の「物体の記述」というところに物体の記述を書くと、それによって記述された物体が発光することになります。

発光する物体の記述を書くときには、二つのことに注意しないといけません。

ひとつは、光源の記述の中に書かれる物体の記述の中では、光源の位置が原点になっている、ということです。

そしてもうひとつは、物体が光っているように見えるようにするためには、フィニッシュの記述の中に、「環境光」と呼ばれるものについての記述を書く必要がある、ということです。

「環境光」(ambient light) というのは、光源がまったく存在していない状態のときに物体に当たっている光のことで、

```
ambient 数値
```

という記述を書くことによって指定することができます。この中の「数値」のところには、0 から 1 までの範囲内で、環境光の強さを示す数値を書きます。数値は、1 に近いほど強い光になります。

発光する物体を作る場合には、環境光の強さとして、もっとも強い 1 を指定する必要があります。つまり、

```
finish { ambient 1 }
```

というフィニッシュを書く必要がある、ということです。

次のシーンは、発光する球を作っています。

シーンの例　looks_like.pov

```
camera {
    location <0, 10, -5>
    look_at <0, 1, 0>
    angle 70
}

light_source {
    <0, 1, 0>
    color rgb 1.6
    looks_like {
        object {
            sphere { 0, 1 }
            pigment { color rgb 1 }
            finish { ambient 1 }
        }
    }
}

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

union {
    cylinder { <-2, 0, -2>, <-2, 2, -2>, 0.2 }
    cylinder { <2, 0, -2>, <2, 2, -2>, 0.2 }
    cylinder { <2, 0, 2>, <2, 2, 2>, 0.2 }
    cylinder { <-2, 0, 2>, <-2, 2, 2>, 0.2 }
    pigment { color rgb 1 }
}
```

第 11 章　メディア

11.1　メディアの基礎

11.1.1　メディアとは何か

POV-Ray では、空間に浮遊している塵や霧のような微粒子のことを、「メディア」(media) と呼びます。

メディアを作る記述を書くことによって、光の軌跡が見える状態になっているシーンや、遠くのものが霞んで見えるシーンなどを作ることができます。

11.1.2　メディアの記述

メディアは、

```
media { ··· }
```

という記述を書くことによって作ることができます。

11.1.3　メディアのタイプ

メディアには、次の三つのタイプがあります。

- 散乱メディア (scattering)
- 吸収メディア (absorption)
- 発光メディア (emission)

散乱メディアについては第 11.2 節で、吸収メディアについては第 11.3 節で、発光メディアについては第 11.4 節で、詳しく説明することにしたいと思います。

11.1.4　大気メディアと物体メディア

メディアは、それが存在する場所によって、次の 2 種類に分類することができます。

- 大気メディア (atomospheric media)
- 物体メディア (object media)

大気メディアというのは、あらゆる空間に存在するメディアのことです。それに対して、物体メディアというのは、特定の物体の内部のみに存在するメディアのことです。

物体の記述の外にメディアの記述を書くと、大気メディアが作られます。それに対して、物体の記述の中にメディアの記述を書くと、物体メディアが作られます。

物体メディアについては、第 11.5 節で、もう少し詳しく説明することにしたいと思います。

11.2　散乱メディア

11.2.1　散乱メディアの基礎

光をさまざまな方向へ反射させるというタイプのメディアは、「散乱メディア」(scattering media) と呼ばれます。

光の軌跡が見える状態になっているシーンや、霧が立ち込めているシーンは、散乱メディアを使うことによって作ることができます。

11.2.2　散乱メディアの記述

散乱メディアを作る記述は、

```
media { scattering { 散乱メディアタイプ, 色の記述 } }
```

と書きます。この中の「散乱メディアタイプ」というところには、散乱メディアのタイプを指定するための番号を書きます。

利用することのできる散乱メディアのタイプは、次の 5 種類です。

1　等方散乱 (isotropic scattering)
2　ミー散乱 (Mie scattering) のヘイズモデル (haze model)
3　ミー散乱 (Mie scattering) のマーキーモデル (murky model)
4　レイリー散乱 (Rayleigh scattering)
5　ヘニエイ・グリーンスタイン散乱 (Henyey-Greenstein scattering)

等方散乱 (isotropic scattering) は、光がどの方向にも等しく散乱する散乱メディアのタイプです。

ミー散乱 (Mie scattering) は、雲や霧を構成している微小な水滴や、大気に飛散した汚染物質のような散乱メディアのタイプです。このタイプの散乱メディアは、等方散乱とは違って指向性があります。光が進む方向と視線の方向とが正反対の場合は散乱がもっとも強く見えて、それらが同じ方向を向いている場合は散乱がもっとも弱く見えます。

ミー散乱には、番号が 2 のヘイズモデル (haze model) と番号が 3 のマーキーモデル (murky model) という二つのモデルがあります。マーキーモデルは、ヘイズモデルよりも強い指向性を持っています。

レイリー散乱 (Rayleigh scattering) は、空気の分子のような、きわめて小さな粒子によって引き起こされる散乱をモデル化した散乱メディアのタイプです。晴れた日の空が青く見えるのは、空気の分子が引き起こすレイリー散乱が原因です。レイリー散乱も、ミー散乱ほどではありませんが、指向性を持っています。光が進む方向と視線の方向とが一致する場合と正反対の場合は散乱がもっとも強く見えて、それらが互いに垂直になっている場合は散乱がもっとも弱く見えます。

ヘニエイ・グリーンスタイン散乱 (Henyey-Greenstein scattering) は、「ヘニエイ・グリーンスタイン位相関数」という関数によって、離心率 (eccentricity) から指向性を示す楕円を求める散乱メディアのタイプです。

散乱メディアを作る記述の中の「色の記述」というところには、散乱する光の色の記述を書きます。この記述は、散乱の強さの記述も兼ねています。あまり強く散乱させると、物体が見えなくなってしまいますので、通常、0.1 よりも小さな数値を指定します。

11.2.3 光の軌跡が見えるシーン

空気中に塵が浮遊している場合、スポットライトで何かを照らしたり、暗い部屋に窓から光が差し込んでいたりすると、その光の軌跡を見ることができます。そのようなシーンは、散乱メディアを使うことによって作ることができます。

次のシーンは、散乱メディアを使うことによって、スポットライトが出す光の軌跡が見える状態を作っています。

シーンの例 scattering.pov

```
camera {
    location <0, 3, -20>
    look_at <0, 6, 0>
    angle 70
}

light_source {
    <0, 20, 0>
    color rgb 1.6
    spotlight
    point_at 0
    radius 18
    falloff 20
    tightness 10
}

media { scattering { 1, color rgb 0.05 } }

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

object {
    sphere { <0, 5, 0>, 1 }
    pigment { color rgb 1 }
}
```

11.2.4 散乱メディアによる霧

霧が立ち込めているシーンも、散乱メディアを使うことによって作ることができます。

次のシーンは、散乱メディアを使うことによって、霧が立ち込めている状態を作っています。

シーンの例 fogbyscattering.pov

```
camera {
    location <0, 15, -80>
    look_at <0, 15, 0>
    angle 70
}

light_source { <-200, 500, -500> color rgb 1.6 }

media { scattering { 2, color rgb 0.02 } }

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}
```

```
#declare Theta = 0;
#while (Theta < 360)
    object {
        cylinder { <-40, 0, 0>, <-40, 40, 0>, 2 }
        pigment { color rgb <0, 1, 0> }
        rotate <0, Theta, 0>
    }
    #declare Theta = Theta + 20;
#end
```

POV-Ray で霧を作る方法は、散乱メディアを使う方法のほかに、もうひとつあります。それは、霧を作るための専用の記述を書くという方法です。この方法については、第 11.7 節で説明することにしたいと思います。

11.3　吸収メディア

11.3.1　吸収メディアの基礎

光を吸収するタイプのメディアは、「吸収メディア」(absorption media) と呼ばれます。

吸収メディアを使うことによって、色の付いたガスが充満したようなシーンを作ることができます。

11.3.2　吸収メディアの記述

吸収メディアを作る記述は、

```
media { absorption 色の記述 }
```

と書きます。この中の「色の記述」というところには、吸収する光の色を記述します。

記述するのは吸収メディアが吸収する光の色ですから、物体には、吸収されなかった光が当たることになります。たとえば、

```
color rgb <0, 1, 1>
```

という色を記述したとすると、その吸収メディアは、緑色の光と青色の光をすべて吸収しますので、物体には赤色の光しか当たらないということになります。

次のシーンは、赤色と緑色の光をすべて吸収する吸収メディアを作っています。光源の色も物体の色も白色ですが、物体は青色に見えるはずです。

シーンの例　absorption.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

media { absorption color rgb <1, 1, 0> }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
}
```

11.4　発光メディア

11.4.1　発光メディアの基礎

光を発光するタイプのメディアは、「発光メディア」(emission media) と呼ばれます。

発光メディアを使うことによって、光で満たされたようなシーンを作ることができます。

11.4.2 発光メディアの記述

発光メディアを作る記述は、

```
media { emission 色の記述 }
```

と書きます。この中の「色の記述」というところには、発光する光の色を記述します。

次のシーンは、黄色の光を出す発光メディアを作っています。光源を作っていないという点に注意してください。

シーンの例 emission.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

media { emission color rgb <0.3, 0.3, 0> }

object {
    sphere { 0, 1 }
    pigment { color rgb 1 }
}
```

11.5 物体メディア

11.5.1 物体メディアの基礎

第 11.1.4 項で説明したように、メディアは、それが存在する場所によって、大気メディア (atomospheric media) と物体メディア (object media) の 2 種類に分類することができます。

これまでの節で紹介したメディアを使ったシーンは、すべて大気メディアを作るものでした。つまり、それらのシーンでは、メディアはあらゆる空間に存在していたわけです。

この節では、物体メディアの作り方、すなわち特定の物体の内部のみにメディアを作る方法について説明したいと思います。

物体メディアを作るためには、物体の記述の中に、次の二つの記述を書く必要があります。

- 物体を中空にする記述。
- インテリアの記述。

11.5.2 物体を中空にする記述

物体メディアを作るためには、内部が中空になった物体を作る必要があります。中空の物体は、物体の記述の中に `hollow` と書くだけで作ることができます。

ただし、`hollow` と書くことによって中空の物体を作ったとしても、色が透明でなければ、内部の様子を見ることができません。ですから、物体メディアを作る場合は、物体の内部を中空にするだけではなくて、その色を透明にする必要があります。

11.5.3 インテリアの記述

POV-Ray では、物体の内部の特性は「インテリア」(interior) と呼ばれます。

インテリアは、

```
interior { ··· }
```

と書くことによって記述することができます。

物体メディアを作るためには、その物体の記述の中にインテリアの記述を書いて、そのインテリアの記述の中にメディアの記述を書く必要があります。たとえば、

```
interior {
    media { scattering { 1, color rgb 0.2 } }
}
```

というインテリアの記述を物体の記述の中に書くことによって、内部に散乱メディアを持つ物体メディアを作ることができます。

11.5.4 散乱の物体メディア

それでは、内部に散乱メディアを持つ物体メディアを作るシーンを書いてみましょう。
次のシーンは、内部に散乱メディアを持つ球の物体メディアを作っています。

シーンの例 objectscattering.pov

```
camera {
    location <0, 6, -20>
    look_at <0, 6, 0>
    angle 70
}

light_source { <0, 20, 0> color rgb 1.6 }

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

object {
    sphere { <0, 14, 0>, 1 }
    pigment { color rgb 1 }
}

object {
    sphere { <0, 6, 0>, 5 }
    hollow
    pigment { color rgbf 1 }
    interior {
        media { scattering { 1, color rgb 0.2 } }
    }
}
```

11.5.5 吸収の物体メディア

次に、内部に吸収メディアを持つ物体メディアを作るシーンを書いてみましょう。
次のシーンは、赤色と緑色の光をすべて吸収する吸収メディアを内部に持つ球の物体メディアを作っています。

シーンの例 objectabsorption.pov

```
camera {
    location <0, 5, -4>
    look_at <0, 2, 0>
    angle 70
}

light_source { <0, 20, 0> color rgb 1.6 }

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

object {
    sphere { <0, 3, 0>, 1 }
    hollow
    pigment { color rgbf 1 }
    interior {
        media { absorption color rgb <1, 1, 0> }
    }
}
```

11.5.6 発光の物体メディア

次に、内部に発光メディアを持つ物体メディアを作るシーンを書いてみましょう。
次のシーンは、黄色の光を出す発光メディアを内部に持つ球の物体メディアを作っています。

シーンの例 objectemission.pov

```
camera {
    location <0, 0, -3>
    look_at 0
    angle 70
}

light_source { <-5, 5, -5> color rgb 1.6 }

object {
    sphere { 0, 1 }
    hollow
    pigment { color rgbf 1 }
    interior {
        media { emission color rgb <1, 1, 0> }
    }
}
```

11.6 密度のパターン

11.6.1 密度のパターンの基礎

単位容積の中にどれぐらいの量の微粒子が浮遊しているかということは、メディアの「密度」(density) と呼ばれます。

メディアは、デフォルトでは、どの位置でも密度が一定のもの、つまり濃淡のないものが作られるのですが、メディアの記述の中に密度のパターンの記述を書くことによって、位置によって濃淡のあるメディアを作ることもできます。

11.6.2 密度のパターンの記述

密度のパターンは、

```
density { パターンタイプ名 }
```

と書くことによって記述することができます。この中の「パターンタイプ名」というところには、パターンタイプを指定する名前を書きます。

次のシーンは、斑点を作る spotted というパターンタイプを使って、位置によって濃淡のある散乱メディアを作っています。

シーンの例 spottedscattering.pov

```
camera {
    location <0, 3, -20>
    look_at <0, 6, 0>
    angle 70
}

light_source {
    <0, 20, 0>
    color rgb 1.6
    spotlight
    point_at 0
    radius 18
    falloff 20
    tightness 10
}

media {
    scattering { 1, color rgb 0.05 }
    density { spotted }
}

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
```

```
}
```

11.7 霧

11.7.1 霧の基礎

霧は、第 11.2.4 項で説明したように、散乱メディアを使うことによって作ることができます。しかし、霧を作る方法はそれだけではありません。POV-Ray には、霧を作るための専用の記述というものがあって、それを書くことによって霧を作ることも可能です。

この節では、霧を作るための専用の記述の書き方について説明したいと思います。

霧を作るための専用の記述を使って作ることのできる霧には、次の 2 種類のものがあります。

- コンスタントフォッグ (constant fog)
- グラウンドフォッグ (ground fog)

11.7.2 コンスタントフォッグ

地面からの高さとは無関係に、一定の密度で立ち込めている霧は、「コンスタントフォッグ」(constant fog) と呼ばれます。

コンスタントフォッグを作る記述は、

```
fog {
    distance 距離
    色の記述
}
```

と書きます。この中の「距離」というところには、霧によって背景の 36.8%が見えなくなる距離を書きます。つまり、この数値が小さければ小さいほど、霧の密度が高くなるということです。そして「色の記述」というところには、霧の色を指定する記述を書きます。

次のシーンは、コンスタントフォッグを作っています。

シーンの例 fog.pov

```
camera {
    location <0, 15, -80>
    look_at <0, 15, 0>
    angle 70
}

light_source { <-200, 500, -500> color rgb 1.6 }

fog {
    distance 60
    color rgb 1
}

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

#declare Theta = 0;
#while (Theta < 360)
    object {
        cylinder { <-40, 0, 0>, <-40, 40, 0>, 2 }
        pigment { color rgb <0, 1, 0> }
        rotate <0, Theta, 0>
    }
    #declare Theta = Theta + 20;
#end
```

11.7.3 グラウンドフォッグ

地面からの高さが低いところだけに立ち込めている霧は、「グラウンドフォッグ」(constant fog) と呼ばれます。

グラウンドフォッグを作る記述は、

```
fog {
    fog_type 2
    distance 距離
    色の記述
    fog_offset オフセット
    fog_alt 高さ
}
```

と書きます。この中の「距離」と「色の記述」のところに書くものは、コンスタントフォッグの場合と同じです。「オフセット」と「高さ」のところには、霧がどれくらいの高さまで立ち込めているかということを示す数値を書きます。グラウンドフォッグは、地面から、「高さ」で指定された高さまでは一定の密度で立ち込めます。その高さからは、高くなるにつれて密度が下がっていって、「高さ」に「オフセット」を加算した高さで、地表での密度の 25%になります。

次のシーンは、グラウンドフォッグを作っています。

シーンの例 groundfog.pov

```
camera {
    location <0, 15, -80>
    look_at <0, 15, 0>
    angle 70
}

light_source { <-200, 500, -500> color rgb 1.6 }

fog {
    fog_type 2
    distance 6
    color rgb 1
    fog_offset 2
    fog_alt 4
}

object {
    plane { y, 0 }
    pigment { checker color rgb 1, color rgb <0, 1, 0> }
}

#declare Theta = 0;
#while (Theta < 360)
    object {
        cylinder { <-40, 0, 0>, <-40, 40, 0>, 2 }
        pigment { color rgb <0, 1, 0> }
        rotate <0, Theta, 0>
    }
    #declare Theta = Theta + 20;
#end
```

参考文献

[Introduction,2004] POV-Team, “Introduction to POV-Ray”, 2004.

[Reference,2004] POV-Team, “POV-Ray Reference”, 2004.

[石本,1998] 石本浩司、『POV-Ray for Windows 入門——No. 1 フリーソフトで作る 3DCG の世界——』、ソシム、1998、ISBN 978-4-88337-081-8。

[工藤,2003] 工藤武信、真鍋正規、『POV-Ray 利用参考マニュアル（3.5V2 版）』、大分大学、2003。

[小室,1999] 小室日出樹、『POV-Ray ではじめるレイトレーシング･改訂二版』、アスキー、1999、ISBN 978-4-7561-3098-3。

[小室,2000] 小室日出樹、『POV-Ray で学ぶ実習コンピュータグラフィックス――CG 検定カリキュラム対応――』、アスキー、2000、ISBN 978-4-7561-3367-0。

[鈴木,2008] 鈴木広隆、倉田和夫、佐藤尚、『POV-Ray による 3 次元 CG 制作：モデリングからアニメーションまで』、CG-ARTS 協会、2008、ISBN 978-4-903474-19-9。

索引