

# POV-Ray 実習マニュアル

第四版

2008 年 1 月 30 日 (水)

Copyright © 2000–2008 Daikoku Manabu

This tutorial is licensed under a Creative Commons Attribution 2.1 Japan License.

## 目次

<b>第 1 章 POV-Ray の基礎</b>	<b>4</b>
1.1 POV-Ray の基礎の基礎	4
1.1.1 3次元グラフィックスとは何か	4
1.1.2 POV-Ray とは何か	4
1.1.3 POV-Team	4
1.1.4 シーン	4
1.2 POV-Ray の使い方	4
1.2.1 シーンの入力	4
1.2.2 レンダリング	5
1.2.3 画像のサイズとアンチエイリアシングの設定	5
1.3 式	5
1.3.1 式の基礎	5
1.3.2 リテラル	5
1.3.3 演算子	6
1.3.4 ベクトル	6
1.3.5 ベクトルの加算	6
1.3.6 ベクトルの乗算	7
1.3.7 基本ベクトル	7
1.4 座標系	7
1.4.1 座標系の基礎	7
1.4.2 座標	7
1.4.3 右手系と左手系	8
1.5 色	8
1.5.1 光の三原色	8
1.5.2 色の記述	8
1.6 シーンの基礎	8
1.6.1 シーンの構成要素	8
1.6.2 ホワイトスペース	8
1.6.3 注釈	9
1.7 物体	9
1.7.1 物体の基礎	9
1.7.2 プリミティブ	9
1.7.3 直方体	10
1.7.4 ピグメント	11
1.8 光源	11
1.8.1 光源の基礎	11
1.8.2 光源の明るさと色	12
1.9 カメラ	12
1.9.1 カメラの基礎	12
1.9.2 カメラの位置	12
1.9.3 カメラの方向	13
1.9.4 カメラの視野の角度	13
1.10 宣言	14
1.10.1 識別子	14

1.10.2	識別子の作り方	14
1.10.3	宣言の書き方	14
1.10.4	識別子の使い方	14
1.10.5	宣言の末尾のセミコロン	15
1.11	変形	16
1.11.1	変形の基礎	16
1.11.2	移動	16
1.11.3	拡大	17
1.11.4	回転	18
1.11.5	回転の順序	19
1.11.6	プリミティブの変形	19
1.12	CSG	20
1.12.1	CSG の基礎	20
1.12.2	合併	20
1.12.3	共通部分	21
1.12.4	減算	21
1.12.5	CSG の組み合わせ	22
1.13	インクルードファイル	23
1.13.1	インクルードファイルの基礎	23
1.13.2	ファイルをインクルードする記述	23
1.13.3	標準インクルードファイル	23
1.13.4	インクルードファイルの書き方	24
1.14	繰り返し	25
1.14.1	関係演算子	25
1.14.2	論理演算子	25
1.14.3	繰り返しの記述	26
1.14.4	繰り返しの入れ子	26
1.14.5	変形の繰り返し	27
<b>第 2 章</b>	<b>テクスチャー</b>	<b>28</b>
2.1	テクスチャーの基礎	28
2.1.1	この章について	28
2.1.2	テクスチャーの要素	28
2.1.3	テクスチャーの記述	28
2.2	ピグメントの基礎	28
2.2.1	ピグメントについての復習	28
2.2.2	複数の色から構成されるピグメント	29
2.2.3	カラーリスト	29
2.2.4	ピグメントに対する変形	30
2.3	カラーマップ	30
2.3.1	カラーマップとは何か	30
2.3.2	カラーマップの作り方	31
2.3.3	パターンタイプ	31
2.3.4	縞模様	31
2.4	光の透過	32
2.4.1	フィルターとトランスミット	32
2.4.2	フィルターによる色の記述	32
2.4.3	屈折率	33
2.4.4	形状の内部の境界面	34
2.4.5	計算の複雑さ	35
2.5	空	36
2.5.1	空にピグメントを与える方法	36
2.5.2	現実的な空	36
2.6	フィニッシュ	37
2.6.1	フィニッシュの基礎	37

目次	3
2.6.2 フィニッシュの記述	37
2.7 ノーマル	39
2.7.1 ノーマルの基礎	39
2.7.2 ノーマルの記述	39
2.7.3 ノーマルでよく使われるパターンタイプ	39
<b>第3章 プリミティブ</b>	<b>40</b>
3.1 円形のプリミティブ	41
3.1.1 この章について	41
3.1.2 円柱	41
3.1.3 円錐台	41
3.1.4 トーラス	42
3.2 平面	42
3.2.1 平面とは何か	42
3.2.2 平面の作り方	43
3.2.3 CSGの素材としての平面	43
3.3 テキスト	44
3.3.1 テキスト	44
3.3.2 テキストの作り方	44
3.3.3 文字間隔	44
3.3.4 同梱されているフォント	45
参考文献	45
索引	47

## 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次元的方法と3次元的方法があります。2次元的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りますが、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 Ray-Tracer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<sup>1</sup>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)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。

<sup>1</sup>POV-Ray 公式サイト 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く、球が空中に浮かんでいる画像が生成されるはずですが、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をするかどうかを設定することができます。リストボックスの項目の中に書かれている AA というのはアンチエイリアシングをするという意味で、No AA というのはそれをしないという意味です。

ちなみに、「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。

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まなわれない小さなサイズでレンダリングするといいいでしょう。

## 1.3 式

### 1.3.1 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。

### 1.3.2 リテラル

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。たとえば、583、-240、0.017、というような 10 進数は、値として特定の数値が得られるリテラルです。



というベクトルが値として得られます。

### 1.3.6 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、

$$\langle 5, 2, 3 \rangle * 40$$

という式を評価すると、

$$\langle 200, 80, 120 \rangle$$

というベクトルが値として得られます。

### 1.3.7 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、

$$x = \langle 1, 0, 0 \rangle$$

$$y = \langle 0, 1, 0 \rangle$$

$$z = \langle 0, 0, 1 \rangle$$

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。

## 1.4 座標系

### 1.4.1 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。

### 1.4.2 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、 $\langle 7, 3, -6 \rangle$ 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点で交わる場所のことです。

### 1.4.3 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異なりますので、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。

## 1.5 色

### 1.5.1 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。

### 1.5.2 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。たとえば、

```
color rgb <1, 0.5, 0>
```

はオレンジ色をあらわしていて、

```
color rgb <0.5, 0, 1>
```

は紫色をあらわしています。

## 1.6 シーンの基礎

### 1.6.1 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節で入力したシーンも、やはり、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。

### 1.6.2 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、単語または式が連続する場合は、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、単語や式の前後などには、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。

### 1.6.3 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してくれ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 contain line feed. */
```

という記述を、注釈とみなして無視します。

スタイルシート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。

## 1.7 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節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節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」のところの書き方は、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。ですから、

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

という記述は、原点を中心の位置とする半径が1の球、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 { <-1, 0.8, 0>, 0.4 }
  pigment { color rgb 1 }
}

object {
  sphere { <1, -0.3, 3>, 2 }
  pigment { color rgb 1 }
}
```

---

### 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。すると、頂点<sub>1</sub> と頂点<sub>2</sub> をつなぐ直線を対角線とする直方体ができます。たとえば、

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

という記述を書くことによって、<-1, -1, -1> と <1, 1, 1> 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* と等しくなります。

次のシーンは、異なる位置と大きさを持つ二つの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 { <3.5, -2, 4>, <4, 3, 12> }
  pigment { color rgb 1 }
}

object {
  box { <-4, -2.5, 4>, <3, -3, 12> }
  pigment { color rgb 1 }
}
```

---

#### 1.7.4 ピグメント

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、物体のテクスチャーを記述したものです。しかし、テクスチャーを構成するすべての要素が、これによって記述されているわけではありません。これは、「ピグメント」(pigment)と呼ばれる、テクスチャーを構成するひとつの要素だけを記述したものです。

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。ちなみに、

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

という記述の中にある1というのは、<1, 1, 1>という意味ですから、この場合は物体に対して白色のピグメントを与えていることとなります。

次のシーンは、異なるピグメントを持つ二つの球を作っています。

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 { <-0.9, 0, 0>, 0.8 }
  pigment { color rgb <0, 1, 1> }
}

object {
  sphere { <0.9, 0, 0>, 0.8 }
  pigment { color rgb <1, 1, 0> }
}
```

---

テクスチャーについては、第2章で、さらに詳しく説明したいと思います。

## 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となります(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。

### 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、白色の球の左側と右側のそれぞれに、異なる色を持つ光源を置いています。

シーンの例 light.pov

---

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

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

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

```

---

## 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。

### 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、 $\langle 0, 0, 0 \rangle$  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、カメラの方向を直方体の頂点のひとつに向けています。

シーンの例 `lookat.pov`

---

```
camera {
  location <-0.7, 1.7, -0.7>
  look_at <0, 1, 0>
  angle 70
}

light_source { <-4, 7, -5> color rgb 1.6 }

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

---

### 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、先ほどのシーンと同じ直方体を撮影しているのですが、カメラと物体とのあいだの距離を大きくして、カメラの視野の角度を狭くしています。

シーンの例 `angle.pov`

---

```
camera {
  location <-50, 51, -50>
  look_at <0, 1, 0>
  angle 1.8
}

light_source { <-4, 7, -5> color rgb 1.6 }

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

---

## 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、その識別子をシーンの中を書くとき、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、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はいけません。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。
- キーワード(camera、object、sphereなど)と一致するものを識別子として使うことはできない。

たとえば、namako、Kurage、Uni874、two\_wordsなどは正しい識別子の例で、umi@u.shiや396ikuraなどは正しくない識別子の例です。

POV-Rayでは、識別子を作るとき、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などのキーワード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、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 Pigment = pigment { color rgb <0, 1, 1> }

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

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 `semico.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 SphereColor = color rgb <0.5, 1, 0>;

camera {
    location CameraLocation
    look_at CameraLookAt
```

```

    angle    CameraAngle
}

light_source { LightLocation LightColor }

object {
    sphere { SphereLocation, SphereRadius }
    pigment { SphereColor }
}

```

---

## 1.11 変形

### 1.11.1 変形の基礎

POV-Rayでは、物体に対して、「変形」(transformation)と呼ばれる操作を加えることができます。変形には、移動、拡大、回転という3種類の操作があります。

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、識別子を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。

### 1.11.2 移動

物体を移動させたいときは、物体を作る記述の中に、

```
translate 3次元ベクトル
```

という記述を書きます。この中の3次元ベクトルは、 $x$ 、 $y$ 、 $z$ 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、ということを示しています。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の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、ひとつの形状から、位置の異なる三つの物体を作っています。

シーンの例 `transla.pov`

---

```

#declare Sphere = sphere { 0, 1 }
#declare Red = color rgb <1, 0, 0>;
#declare Green = color rgb <0, 1, 0>;
#declare Blue = color rgb <0, 0, 1>;

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

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

object {
    Sphere
    pigment { Red }
    translate <-1, 1, 1>
}

```

```

object {
  Sphere
  pigment { Green }
}

object {
  Sphere
  pigment { Blue }
  translate <1, -1, -1>
}

```

---

### 1.11.3 拡大

物体を拡大したいときは、物体を作る記述の中に、

```
scale 3次元ベクトル
```

という記述を書きます。この中の3次元ベクトルは、 $x$ 、 $y$ 、 $z$ のそれぞれの座標軸の方向の拡大率をあらわしています（拡大率が1よりも小さい場合は、縮小されることになります）。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、本来の大きさと作られたのち、 $x$ 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、ひとつの形状から、位置と大きさの異なる三つの物体を作っています。

シーンの例 `scale.pov`

---

```

#declare Sphere = sphere { 0, 1 }
#declare Red = color rgb <1, 0, 0>;
#declare Green = color rgb <0, 1, 0>;
#declare Blue = color rgb <0, 0, 1>;

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

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

object {
  Sphere
  pigment { Red }
  translate <-2, 0, -2>
  scale 0.5
}

object {
  Sphere
  pigment { Green }
}

object {
  Sphere
  pigment { Blue }
  translate <1, 0, 1>
  scale 2
}

```

}

いくつかの変形を組み合わせる場合は、それらをどのような順序で組み合わせるのかということに注意を払う必要があります。なぜなら、順序が変われば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実行されます。たとえば、

```
translate <1, 0, 1>
scale 2
```

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

#### 1.11.4 回転

物体を回転させたいときは、物体を作る記述の中に、

```
rotate 3次元ベクトル
```

という記述を書きます。この中の3次元ベクトルは、 $x$ 、 $y$ 、 $z$ のそれぞれの座標軸を回転軸とする回転の角度をあらわしています（回転は、 $x$ 軸、 $y$ 軸、 $z$ 軸という順序で実行されます）。たとえば、

```
object {
  Katachi
  pigment { color Iro }
  rotate <20, -50, 30>
}
```

という記述で物体を作ったとすると、この物体は、本来の方向で作られたのち、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、次に $z$ 軸を回転軸として30度だけ回転することになります。

回転角度がプラスの場合、その回転の向きは、左手で、親指がプラスの方向を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。

次のシーンは、ひとつの形状から、方向の異なる三つの物体を作っています。

シーンの例 rotate.pov

```
#declare Box = box { <-1, -0.1, -2>, <1, 0.1, 2> }
#declare Red = color rgb <1, 0, 0>;
#declare Green = color rgb <0, 1, 0>;
#declare Blue = color rgb <0, 0, 1>;
```

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

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

```
object {
  Box
  pigment { Red }
  rotate <0, 0, 30>
  translate <-2, 0, 0>
}
```

```
object {
  Box
  pigment { Green }
}
```

```
object {
  Box
  pigment { Blue }
  rotate <0, 0, -30>
  translate <2, 0, 0>
}
```

## 1.11.5 回転の順序

rotate の記述を 1 個だけ書いた場合、 $x$ 、 $y$ 、 $z$  のそれぞれの軸を回転軸とする回転は、 $x$ 、 $y$ 、 $z$  という順序で実行されます。これ以外の順序で物体を回転させたい場合は、rotate の記述を 2 個または 3 個、書く必要があります。たとえば、まず最初に  $y$  軸で 30 度だけ回転させて、その次に  $x$  軸で  $-45$  度だけ回転させたい、という場合は、

```
rotate <0, 30, 0>
rotate <-45, 0, 0>
```

というように、2 個の rotate を書くことが必要になります。

次のシーンで作られている物体のうちで、赤色のものは、 $z$  軸、 $y$  軸という順序で回転させたもので、青色のものは、 $y$  軸、 $z$  軸という順序で回転させたものです。

## シーンの例 rotate2.pov

---

```
#declare Box = box { <-1, -0.1, -2>, <1, 0.1, 2> }
#declare Red = color rgb <1, 0, 0>;
#declare Green = color rgb <0, 1, 0>;
#declare Blue = color rgb <0, 0, 1>;

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

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

object {
    Box
    pigment { Red }
    rotate <0, 0, 30>
    rotate <0, 45, 0>
    translate <-2, 0, 0>
}

object {
    Box
    pigment { Green }
}

object {
    Box
    pigment { Blue }
    rotate <0, 45, 30>
    translate <2, 0, 0>
}

```

---

## 1.11.6 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 { <-5, 5, -5> color rgb 1.6 }

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

---

## 1.12 CSG

### 1.12.1 CSGの基礎

POV-Rayは、いくつかの形状を組み合わせるによって新しい形状を作り出すことができるという機能を持っています。形状を組み合わせるによって新しい形状を作るという操作は、「CSG」(constructive solid geometry)と呼ばれます。CSGに属する操作というのは、集合に対する演算と同じもので、合併(和)、共通部分(積)、減算(差)などがあります。

CSG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CSGは、unionという名前によって識別されます。

CSGを使って形状を作る記述は、

```
CSGの名前 { 物体の記述 ... }
```

と書きます。「物体の記述」ののところには、物体の記述またはプリミティブの記述を書きます。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CSGを実行することによって得られる形状をあらわすこととなります。

### 1.12.2 合併

いくつかの形状を合体させることによって新しい形状を作り出すというCSGは、「合併」(union)または「和」(summation)と呼ばれます。合併は、unionという名前によって識別されます。

いくつかの形状を合併させた結果を求めたいときは、

```
union { 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となります。

次のシーンは、球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

```
#declare Sphere = sphere { 0, 1 }

#declare Box = box {
  <0, -1, 0>, <2, 1, 2>
  rotate <0, 45, 0>
}

#declare Union = union {
  object { Sphere }
  object { Box }
}

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

light_source { <0, 6, -5> color rgb 1.6 }

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

---

### 1.12.3 共通部分

いくつかの形状が与えられたとき、それら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くという CSG は、「共通部分」(intersection) または「積」(product) と呼ばれます。共通部分は、intersection という名前によって識別されます。

いくつかの形状の共通部分を求めたいときは、

```
intersection { 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、球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 `inter.pov`

---

```
#declare Sphere = sphere { 0, 1 }

#declare Box = box {
  <0, -1, 0>, <2, 1, 2>
  rotate <0, 45, 0>
}

#declare Intersection = intersection {
  object { Sphere }
  object { Box }
}

camera {
  location <1, 2, -2>
  look_at <0.4, 0, 0>
  angle 70
}

light_source { <0, 6, -5> color rgb 1.6 }

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

---

### 1.12.4 減算

いくつかの形状が与えられたとき、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くことによって、残った部分を求めるという CSG は、「減算」(subtraction) または「差」(difference) と呼ばれます。減算は、difference という名前によって識別されます。

形状から形状を減算したいときは、

```
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、球から直方体を減算した形状を持つ物体と、直方体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 `diffe.pov`

---

```
#declare Sphere = sphere { 0, 1 }

#declare Box = box {
  <0, -1, 0>, <2, 1, 2>
  rotate <0, 45, 0>
}

#declare Difference1 = difference {
  object { Sphere }
  object { Box }
}
```

---

```

#declare Difference2 = difference {
    object { Box }
    object { Sphere }
}

#declare Aqua = color rgb <0, 1, 1>;
#declare Yellow = color rgb <1, 1, 0>;

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

light_source { <0, 6, -5> color rgb 1.6 }

object {
    Difference1
    pigment { Aqua }
    translate <-0.4, 0, 0>
}

object {
    Difference2
    pigment { Yellow }
    translate <0.4, 0, 0>
}

```

---

#### 1.12.5 CSGの組み合わせ

CSGの記述というのは、それ自体がひとつの物体の記述になります。ですから、CSGの記述の中にCSGの記述を書く、ということも可能です。つまり、CSGの記述は、

```

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

```

というように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、CSGによって作られた形状に対してさらにCSGを実行する、ということができます。

次のシーンは、二つの球を合併させた形状から直方体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

---

```

#declare Sphere1 = sphere { 0, 1 }
#declare Sphere2 = sphere { <1.2, 0, 0>, 0.4 }
#declare Box = box { <-3, 0, -2>, <3, 2, 0> }

#declare DifferenceUnion = difference {
    union {
        object { Sphere1 }
        object { Sphere2 }
    }
    object { Box }
}

camera {
    location <2, 1, -2>
    look_at <0.4, 0, 0>
    angle 70
}

light_source { <0, 8, -10> color rgb 1.6 }

```

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
stones.inc	石のテクスチャー
metals.inc	金属のテクスチャー
golds.inc	金のテクスチャー
glass.inc	ガラスのテクスチャー
skies.inc	空のテクスチャーと雲の形状
stars.inc	星空のテクスチャー
textures.inc	ピグメント、フィニッシュ、カラーマップなど

表 1.1: 主要な標準インクルードファイル

```
object {
  DifferenceUnion
  pigment { color rgb <0.3, 0.6, 1> }
}
```

## 1.13 インクルードファイル

### 1.13.1 インクルードファイルの基礎

POV-Ray では、数値、ベクトル、色、形状、テクスチャーなど、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のに対して、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。

### 1.13.2 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、この記述の場所に埋め込むことができます。

### 1.13.3 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、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 1.1 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、その中で識別子が与えられているものを示しています。

標準インクルードファイルは、POV-Ray がインストールされているフォルダの下にある、include というフォルダの下に格納されています。しかし、標準インクルードファイルは、シーンのファイルと同じフォルダの中に置かれていなくても、ファイルの名前を書くだけで指定することができます。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とするならば、

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

という記述を書けばいいわけです。

次のシーンは、shapes.inc と colors.inc という二つの標準インクルードファイルを使って書かれています。

シーンの例 include.pov

---

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

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

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

object {
    Cylinder_X
    pigment { Orange }
}
```

---

Cylinder\_X というのは、shapes.inc という標準インクルードファイルの中で宣言されている識別子のひとつで、 $x$  軸の方向に無限に長い円柱という形状に与えられています。

Orange というのは、colors.inc という標準インクルードファイルの中で宣言されている識別子のひとつで、オレンジ色という色に与えられています。

#### 1.13.4 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には、宣言以外のもの（たとえば、カメラやライトや物体を実際に作る記述など）を書いてはいけません。
- (2) インクルードファイルの先頭と末尾には、同一のファイルが重複してインクルードされることを防ぐためと、POV-Ray のバージョンを退避させて復元するための記述を書かないといけません。

重複してインクルードされることを防ぐためと、バージョンを退避させて復元するための記述というのは、ファイルの先頭に書く、

```
#ifndef ( 独自の識別子 )
#declare 独自の識別子 = version;
```

という記述と、ファイルの末尾に書く、

```
#version 独自の識別子 ;
#end
```

という記述のことです。この中に出てくる「独自の識別子」のところには、ほかのインクルードファイルとは重複しないような識別子を書きます（三か所とも、同じ識別子でないといけません）。

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。

式	意味
$A == B$	$A$ と $B$ とは等しい。
$A != B$	$A$ と $B$ とは等しくない。
$A > B$	$A$ のほうが $B$ よりも大きい。
$A < B$	$A$ のほうが $B$ よりも小さい。
$A >= B$	$A$ のほうが $B$ よりも大きいか、または $A$ と $B$ とは等しい。
$A <= B$	$A$ のほうが $B$ よりも小さいか、または $A$ と $B$ とは等しい。

表 1.2: 関係演算子

インクルードファイルの例 `frame.inc`


---

```
#ifndef (Frame_Inc_Temp)
#declare Frame_Inc_Temp = version;

#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` をインクルードすることによって、`Frame` という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, 1, -2.5>
    look_at 0
    angle 70
}

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

object {
    Frame
    pigment { color rgb <0.6, 1, 0> }
}
```

---

## 1.14 繰り返し

## 1.14.1 関係演算子

成り立っているか成り立っていないかという判断の対象となるものは、「条件」(condition) と呼ばれます。

POV-Ray では、条件をあらわす式を書くことができるようにするために、表 1.2 に示されている、「関係演算子」(relational operator) と呼ばれる演算子を使うことができるようになっています。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。

## 1.14.2 論理演算子

POV-Ray では、関係演算子に加えて、表 1.3 に示されている、「論理演算子」(logical operator) と呼ばれる演算子も使うことができます。

論理演算子を使うことによって、条件を否定する条件を記述したり、条件と条件とを組み合わ

式	意味
$! A$	$A$ ではない。
$A \&\& B$	$A$ かつ $B$ である。
$A \ \  B$	$A$ または $B$ である。

表 1.3: 論理演算子

せた条件を記述することができます。たとえば、

```
X < 100 && Y == 0
```

という式を書くことによって、「 $X$ は100よりも小さくて、かつ、 $Y$ は0と等しい」という条件を記述することができます。

#### 1.14.3 繰り返しの記述

同じような形状が何個も並んでできているような形状は、同じような記述を何個も並べて書くという方法で作ることもできますが、そのような形状は、「繰り返し」(iteration)と呼ばれる処理を使うことによって、簡潔に記述することができます。繰り返しというのは、その言葉のとおり、何らかの処理を繰り返すという処理のことです。

繰り返しをあらわす記述は、

```
#while (式)
    繰り返しの対象となる記述
#end
```

と書きます。この記述は、「繰り返しの対象となる記述」のところに書かれたものを繰り返して処理するという意味です。

繰り返しの記述の先頭には、丸括弧で囲んで1個の式を書く必要があります。この式は、繰り返しの条件を指定するものです。繰り返しの対象となる記述は、繰り返しの条件が成り立っているあいだけ繰り返されます。

次のシーンは、直線に沿って10個の球を並べた形状を作って、その形状を持つ物体を作っています。

#### シーンの例 iterate.pov

---

```
#declare SphereSequence = union {
    #declare X = 0;
    #while (X < 20)
        object { sphere { <X, 0, 0>, 1 } }
        #declare X = X + 2;
    #end
}

camera {
    location <9, 0, -18>
    look_at <9, 0, 0>
    angle 70
}

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

object {
    SphereSequence
    pigment { color rgb <0.3, 0.8, 1> }
}

```

---

#### 1.14.4 繰り返しの入れ子

繰り返しは、入れ子にすることができます。すなわち、繰り返しを含む処理をさらに繰り返すということができます。ですから、形状を直線的に並べる記述だけではなくて、平面的に並べる記述や立体的に並べる記述も、繰り返しを入れ子にすることによって書くことができます。

次のシーンは、1000 個の球を立体的に並べた形状を作って、その形状を持つ物体を作っています。

シーンの例 `nesting.pov`

---

```
#declare SphereBox = union {
  #declare Z = 0;
  #while (Z < 20)
    #declare Y = 0;
    #while (Y < 20)
      #declare X = 0;
      #while (X < 20)
        object { sphere { <X, Y, Z>, 1 } }
        #declare X = X + 2;
      #end
      #declare Y = Y + 2;
    #end
    #declare Z = Z + 2;
  #end
}

camera {
  location <-8, 24, -18>
  look_at <4, 12, 0>
  angle 70
}

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

object {
  SphereBox
  pigment { color rgb <0.3, 1, 0.6> }
}
```

---

#### 1.14.5 変形の繰り返し

繰り返しの対象となる処理の中で形状を変形させる、ということも可能です。

繰り返しの中で形状を変形させることによって、さまざまな興味深い形状を作ることができます。たとえば、形状を回転させながら繰り返すことによって、円周に沿って形状を並べた形状を作ることができます。

次のシーンは、円周に沿って 36 個の球を並べた形状を作って、その形状を持つ物体を作っています。

シーンの例 `circle.pov`

---

```
#declare Circle = union {
  #declare Theta = 0;
  #while (Theta < 360)
    object {
      sphere { <10, 0, 0>, 1 }
      rotate <0, Theta, 0>
    }
    #declare Theta = Theta + 10;
  #end
}

camera {
  location <0, 6, -20>
  look_at <0, -2, 0>
  angle 70
}

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

object {
  Circle
  pigment { color rgb <1, 0.3, 0.6> }
}
```

## 第2章 テクスチャー

### 2.1 テクスチャーの基礎

#### 2.1.1 この章について

第1.7節で簡単に説明したように、物体の記述の中には、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」と呼ばれる一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。

#### 2.1.2 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。

#### 2.1.3 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。

### 2.2 ピグメントの基礎

#### 2.2.1 ピグメントについての復習

第1.7節で説明したように、物体の素材が持っている色のことを「ピグメント」(pigment)と言います。

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。

### 2.2.2 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。

### 2.2.3 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。たとえば、

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

という記述を書くことによって、立方体のパターンで赤と青を組み合わせたカラーリストピグメントを作ることができます。

シーンの例 colist.pov

---

```
#declare White = color rgb 1;
#declare SkyBlue = color rgb <0.3, 0.6, 1>;
#declare NavyBlue = color rgb <0, 0, 0.6>;
#declare Box = box { <-5, -5, -5>, <5, 5, 5> }

camera {
    location <-14, 14, -14>
    look_at <-3, 0, 0>
    angle 70
}

light_source { <-40, 70, -60> color rgb 1.6 }

object {
    Box
    pigment { checker color White, color SkyBlue }
    translate <-15, 0, 0>
}

object {
    Box
    pigment {
        hexagon color White, color SkyBlue, color NavyBlue
    }
}

object {
```

```

Box
pigment { brick color White, color SkyBlue }
translate <15, 0, 0>
}

```

---

#### 2.2.4 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。

シーンの例 tranpig.pov

---

```

#declare RedWhiteChecker = pigment {
    checker color rgb <1,0,0>, color rgb <1,1,1>
}

#declare Box = box { <-2, -2, -2>, <2, 2, 2> }

camera {
    location <18, 6, -11>
    look_at <12, 0, 0>
    angle 70
}

light_source { <40, 60, -70> color rgb 1.6 }

object {
    Box
    pigment { RedWhiteChecker }
}

object {
    Box
    pigment {
        RedWhiteChecker
        translate <0.7, 0.6, 0.5>
    }
    translate <6, 0, 0>
}

object {
    Box
    pigment {
        RedWhiteChecker
        scale <1.3, 0.8, 1.6>
    }
    translate <12, 0, 0>
}

object {
    Box
    pigment {
        RedWhiteChecker
        rotate <0, 0, 30>
    }
    translate <18, 0, 0>
}

```

---

## 2.3 カラーマップ

### 2.3.1 カラーマップとは何か

前の節で説明したように、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、それを書くことによって、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。

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。

### 2.3.2 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。

### 2.3.3 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す。パターンタイプというのはパターンの種類のことで、

### 2.3.4 縞模様

パターンタイプにはさまざまなものがありますが、ここでは、gradient というパターンタイプを紹介したいと思います。これを使うことによって、縞模様のピグメントを作ることができます。gradient を使ったパターンの記述は、

```
gradient [3次元ベクトル]
```

と書きます。この中に書く 3 次元ベクトルは、縞模様の方向を決めるためのもので、ベクトルが向いている方向に沿って色が変化していきます。たとえば、

```
gradient x
```

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

シーンの例 colmap.pov

```
#declare Red = color rgb <1, 0, 0>;
#declare Yellow = color rgb <1, 1, 0>;

#declare RedYellow = pigment {
  gradient x
  color_map {
    [0.0 color Red]
    [0.4 color Yellow]
    [0.6 color Yellow]
    [1.0 color Red]
  }
}
```

```

    }
}

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

camera {
    location <-2, 1.4, -2.6>
    look_at <0, -0.4, 0>
    angle 70
}

light_source { <-40, 70, -60> color rgb 1.6 }

object {
    Box
    pigment { RedYellow }
}

```

---

## 2.4 光の透過

### 2.4.1 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、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ことにあります。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。

### 2.4.2 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。

シーンの例 filter.pov

---

```

#declare White = color rgb 1;
#declare Blue = color rgb <0, 0, 1>;
#declare RedFilter = color rgbf <1, 0, 0, 1>;
#declare Board = box { <-1e10, -1e10, 0>, <1e10, 1e10, 1> }
#declare Sphere = sphere { 0, 2 }

#declare Lens = intersection {
    object { Sphere translate <0, 0, -1.6> }
    object { Sphere translate <0, 0, 1.6> }
    scale <1, 1, 0.4>
}

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

```

```
light_source { <0, 0, -20> color rgb 1.6 }

object {
  Board
  pigment {
    checker color White, color Blue
    scale 0.4
  }
}

object {
  Lens
  pigment { color RedFilter }
  translate <0, 0, -4>
}
```

---

### 2.4.3 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。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。

シーンの例 ior.pov

---

```
#declare White = color rgb 1;
#declare Blue = color rgb <0, 0, 1>;
#declare RedFilter = color rgbf <1, 0, 0, 1>;
#declare Board = box { <-1e10, -1e10, 0>, <1e10, 1e10, 1> }
#declare Sphere = sphere { 0, 2 }

#declare Lens = intersection {
  object { Sphere translate <0, 0, -1.6> }
  object { Sphere translate <0, 0, 1.6> }
  scale <1, 1, 0.4>
}

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

light_source { <0, 0, -20> color rgb 1.6 }

object {
  Board
  pigment {
    checker color White, color Blue
    scale 0.4
  }
}

object {
  Lens
  pigment { color RedFilter }
  interior { ior 1.5 }
  translate <0, 0, -4>
}
```

---

#### 2.4.4 形状の内部の境界面

第 1.12 節で説明したように、union という CSG を使うことによって、いくつかの形状を合併させた形状を作ることができます。

ただし、union を使って形状を合併させた場合は、形状の内部に埋もれた境界面がそのまま保存されます。不透明な物体を作る場合は、それで何の問題もないのですが、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。

透明な物体を作る場合に、その内部の境界面を消してしまいたいときは、union ではなく、merge という CSG を使います。merge というのは、形状の内部に埋もれた境界面を消すというところ以外、union とまったく同じ動作をする CSG です。

シーンの例 merge.pov

---

```
#declare White = color rgb 1;
#declare Blue = color rgb <0, 0, 1>;
#declare RedFilter = color rgbf <1, 0, 0, 1>;
#declare Board = box { <-1e10, -1e10, 0>, <1e10, 1e10, 1> }
#declare Sphere = sphere { 0, 2 }

#declare Lens = intersection {
    object { Sphere translate <0, 0, -1.6> }
    object { Sphere translate <0, 0, 1.6> }
    scale <1, 1, 0.4>
}

#declare UnionLens = union {
    object { Lens translate <0, -0.6, 0> }
    object { Lens translate <0, 0.6, 0> }
}

#declare MergeLens = merge {
    object { Lens translate <0, -0.6, 0> }
    object { Lens translate <0, 0.6, 0> }
}

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

light_source { <0, 0, -20> color rgb 1.6 }

object {
    Board
    pigment {
        checker color White, color Blue
        scale 0.4
    }
}

object {
    UnionLens
    pigment { color RedFilter }
    interior { ior 1.5 }
    translate <-1.6, 0, -3>
}

object {
    MergeLens
    pigment { color RedFilter }
    interior { ior 1.5 }
    translate <1.6, 0, -3>
}
```

---

## 2.4.5 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、レンダリングの結果として得られる画像の一部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 数値 }
```

という形の記述を書きます。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。

シーンの例 `maxtra.pov`

---

```
global_settings { max_trace_level 7 }

#declare White = color rgb 1;
#declare Blue = color rgb <0, 0, 1>;
#declare Filter = color rgbf <1, 1, 1, 1>;
#declare Board = box { <-1e10, -1e10, 0>, <1e10, 1e10, 1> }
#declare Sphere = sphere { 0, 2 }

#declare Lens = intersection {
    object { Sphere translate <0, 0, -1.6> }
    object { Sphere translate <0, 0, 1.6> }
    scale <1, 1, 0.4>
}

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

light_source { <0, 0, -20> color rgb 1.6 }

object {
    Board
    pigment {
        checker color White, color Blue
        scale 0.4
    }
}

object {
    Lens
    pigment { color Filter }
    interior { ior 1.5 }
    translate <-0.7, -0.7, -3.5>
}

object {
    Lens
    pigment { color Filter }
    interior { ior 1.5 }
    translate <0.7, -0.2, -4>
}

object {
    Lens
    pigment { color Filter }
    interior { ior 1.5 }
}
```

```

    translate <0, 0.3, -4.5>
}

```

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を設定していますが、デフォルトの5にすると、生成される画像の中に黒い部分ができてしまいます。

## 2.5 空

### 2.5.1 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。

#### シーンの例 sky.pov

```

#declare Green = color rgb <0, 0.5, 0>;
#declare Brown = color rgb <0.4, 0.2, 0.1>;
#declare SkyColor = color rgb <0, 0.8, 1>;
#declare Radius = 6.4e6;
#declare Earth = sphere { <0, -Radius, 0>, 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 SkyColor } }

object {
    Earth
    pigment { checker color Green, color Brown }
}

```

### 2.5.2 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、天頂に近づ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、カラーマップを書くことによって作ることができます。

#### シーンの例 realsky.pov

```

#declare Green = color rgb <0, 0.5, 0>;
#declare Brown = color rgb <0.4, 0.2, 0.1>;
#declare SkyColor1 = color rgb <0.7, 0.9, 1>;
#declare SkyColor2 = color rgb <0, 0.5, 1>;
#declare Radius = 6.4e6;
#declare Earth = sphere { <0, -Radius, 0>, Radius }

#declare SkyPigment = pigment {
    gradient y
    color_map {
        [0 color SkyColor1]
        [0.5 color SkyColor2]
    }
}

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 { SkyPigment } }

object {
    Earth
    pigment { checker color Green, color Brown }
}

```

---

## 2.6 フィニッシュ

### 2.6.1 フィニッシュの基礎

第2.1節で説明したように、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。

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、次のような要素から構成されます。

拡散反射 (diffuse reflection)	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。
鏡面反射 (specular reflection)	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。周囲の風景が表面に映り込む。
ハイライト (highlight)	光沢のある物体に光を当てたときにできる、まわりよりも強く光っている部分。

### 2.6.2 フィニッシュの記述

フィニッシュの記述は、基本的には、

```

finish {
    diffuse      拡散反射の比率 (0 から 1 まで)
    reflection   鏡面反射の比率 (0 から 1 まで)
    specular     ハイライトの明るさ (0 から 1 まで)
    roughness    ハイライトの大きさ (0 から 1 まで)
}

```

というように書きます。

シーンの例 finish.pov

---

```

#declare White = color rgb 1;
#declare Green = color rgb <0, 0.5, 0>;
#declare Blue = color rgb <0, 0, 1>;
#declare SkyColor = color rgb <0, 0.8, 1>;
#declare Radius = 6.4e6;
#declare Earth = sphere { <0, -Radius, 0>, Radius }
#declare Sphere = sphere { 0, 1 }

#declare BlueTexture1 = texture {
    pigment { color Blue }
    finish {
        diffuse 0.8
        reflection 0
        specular 0.6
        roughness 0.06
    }
}

#declare BlueTexture2 = texture {

```

```
pigment { color Blue }
finish {
    diffuse    0.8
    reflection 0
    specular   0.8
    roughness  0.006
}
}

#declare BlueTexture3 = texture {
    pigment { color Blue }
    finish {
        diffuse    0.4
        reflection 0.2
        specular   0.8
        roughness  0.006
    }
}

#declare BlueTexture4 = texture {
    pigment { color Blue }
    finish {
        diffuse    0.4
        reflection 0.8
        specular   0.8
        roughness  0.006
    }
}

camera {
    location <0, 1, -7>
    look_at  <0, 3, 0>
    angle    70
}

light_source { <0, 1e10, 0> color rgb 1.6 }
light_source { <0, 1, -5> color rgb 1.2 }

sky_sphere { pigment { color SkyColor } }

object {
    Earth
    pigment { checker color White, color Green }
}

object {
    Sphere
    texture { BlueTexture1 }
    translate <-3.4, 3, 0>
}

object {
    Sphere
    texture { BlueTexture2 }
    translate <-1.1, 3, 0>
}

object {
    Sphere
    texture { BlueTexture3 }
    translate <1.1, 3, 0>
}

object {
    Sphere
    texture { BlueTexture4 }
    translate <3.4, 3, 0>
}
```

}

## 2.7 ノーマル

### 2.7.1 ノーマルの基礎

第2.1節で説明したように、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。

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2.3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。

### 2.7.2 ノーマルの記述

ノーマルの記述は、基本的には、

```
normal { パターンタイプ名 凹凸の深さ }
```

というように書きます。「凹凸の深さ」というところには、0 から 1 までのあいだの数値を書きません。そうすると、名前で指定されたパターンタイプを使って、数値で指定された深さの凹凸を持つノーマルが作られます。

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。

### 2.7.3 ノーマルでよく使われるパターンタイプ

パターンタイプのうちで、ノーマルでよく使われるものとしては、次のようなものがあります。

bumps	オレンジの皮の表面のような凹凸。
dents	ハンマーで叩いたときにできるくぼみのような凹凸。
wrinkles	セロファンやアルミ箔をくしゃくしゃにしたような凹凸。
ripples	水面にできる波紋のような凹凸。
waves	波紋のような凹凸。ripples よりも、丸みを帯びていて幅が広い。

#### シーンの例 normal.pov

```
#declare White = color rgb 1;
#declare Green = color rgb <0, 0.5, 0>;
#declare Blue = color rgb <0, 0, 1>;
#declare SkyColor = color rgb <0, 0.8, 1>;
#declare Radius = 6.4e6;
#declare Earth = sphere { <0, -Radius, 0>, Radius }
#declare Sphere = sphere { 0, 1 }

#declare Finish1 = finish {
    diffuse    0.8
    reflection 0
    specular   0.4
    roughness  0.08
}

#declare Finish2 = finish {
    diffuse    0.2
    reflection 0.8
    specular   0.8
    roughness  0.006
}

#declare BlueTexture1 = texture {
    pigment { color Blue }
    finish { Finish1 }
    normal { bumps 0.2 scale 0.04 }
}
```

```
#declare BlueTexture2 = texture {
    pigment { color Blue }
    finish { Finish1 }
    normal { bumps 0.8 scale 0.04 }
}

#declare BlueTexture3 = texture {
    pigment { color Blue }
    finish { Finish1 }
    normal { bumps 0.2 scale 0.08 }
}

#declare BlueTexture4 = texture {
    pigment { color Blue }
    finish { Finish2 }
    normal { bumps 0.2 scale 0.08 }
}

camera {
    location <0, 1, -7>
    look_at <0, 3, 0>
    angle 70
}

light_source { <0, 1e10, 0> color rgb 1.6 }
light_source { <0, 1, -5> color rgb 1.6 }

sky_sphere { pigment { color SkyColor } }

object {
    Earth
    pigment { checker color White, color Green }
}

object {
    Sphere
    texture { BlueTexture1 }
    translate <-3.4, 3, 0>
}

object {
    Sphere
    texture { BlueTexture2 }
    translate <-1.1, 3, 0>
}

object {
    Sphere
    texture { BlueTexture3 }
    translate <1.1, 3, 0>
}

object {
    Sphere
    texture { BlueTexture4 }
    translate <3.4, 3, 0>
}
```

---

## 3.1 円形のプリミティブ

### 3.1.1 この章について

第 1.7 節で説明したように、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、それら以外にもまだまだたくさんの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重要なものをいくつか紹介していきたいと思います。

### 3.1.2 円柱

この節では、丸い形を作る三つのプリミティブを紹介したいと思います。

まず最初は cylinder です。これは、円柱を作るプリミティブです。

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<sub>1</sub>」と「中心<sub>2</sub>」のところには、円柱の両端にあるそれぞれの底面について、その中心の座標を書きます。そして、「半径」のところには、それらの底面の半径を書きます。

次のシーンは、cylinder を使って二つの円柱を作っています。

シーンの例 cylinder.pov

---

```
#declare Cylinder1 = cylinder { <0, -2, 0>, <0, 2, 0>, 1 }
#declare Cylinder2 = cylinder { <0, 0, -1>, <0, 0, 1>, 2 }

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

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

object {
    Cylinder1
    pigment { color rgb <1, 0.3, 0.6> }
    translate <-2.3, 0, 0>
}

object {
    Cylinder2
    pigment { color rgb <0.3, 1, 0.6> }
    translate <1.7, 0, 0>
}

```

---

### 3.1.3 円錐台

次に紹介するのは、cone というプリミティブです。これを使うことによって、円錐台を作ることができます。

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<sub>1</sub>」と「半径<sub>1</sub>」のところには、円錐台の両端にある底面のうちのひとつについて、その中心の座標と半径を書きます。そして、「中心<sub>2</sub>」と「半径<sub>2</sub>」のところには、もうひとつの底面について、その中心の座標と半径を書きます。

ちなみに、「半径<sub>1</sub>」または「半径<sub>2</sub>」のどちらか一方を 0 にすることによって、円錐を作ることができます。

次のシーンは、cone を使って円錐台と円錐を作っています。

シーンの例 cone.pov

---

```
#declare Cone1 = cone { <0, -1, 0>, 2, <0, 1, 0>, 1 }
#declare Cone2 = cone { <-1, 0, 0>, 0, <1, 0, 0>, 2 }

```

---

```

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

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

object {
    Cone1
    pigment { color rgb <0.3, 1, 0.6> }
    translate <-1.5, 0, 0>
}

object {
    Cone2
    pigment { color rgb <0.6, 1, 0.3> }
    translate <1.2, 0, 0>
}

```

---

### 3.1.4 トーラス

次は、torusです。これは、「トーラス」と呼ばれる形状を作るプリミティブです。トーラスというのは、ドーナツのような形のことです。

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<sub>1</sub>」のところにはトーラス全体の半径を書いて、「半径<sub>2</sub>」のところには、トーラスを構成している輪になった円管の断面の半径を書きます。

トーラスは、原点を中心にして、 $x$  軸と  $z$  軸に対して平行に作られます。

次のシーンは、torusを使って二つのトーラスを作っています。

シーンの例 torus.pov

```

#declare Torus1 = torus { 10, 0.2 }
#declare Torus2 = torus { 4, 2 }

camera {
    location <-8, 2, -20>
    look_at 0
    angle 70
}

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

object {
    Torus1
    pigment { color rgb <0.3, 0.6, 1> }
}

object {
    Torus2
    pigment { color rgb <0.3, 1, 0.8> }
    rotate <90, 0, 0>
    translate <10, 0, 0>
}

```

---

## 3.2 平面

### 3.2.1 平面とは何か

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。

### 3.2.2 平面の作り方

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、物質のある側とない側との境界面に対して垂直で、物質のない方向を示すベクトルを書きます。そして、「距離」のところには、境界面から原点までの距離を書きます。

「法線ベクトル」のところに  $y$  (つまり  $\langle 0, 1, 0 \rangle$ ) 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) を作成することができます。

次のシーンは、plane を使って平面とトーラスを作っています。

シーンの例 plane.pov

---

```
#declare Floor = plane { y, 0 }
#declare Torus = torus { 10, 1 }

camera {
    location <-2, 1, -14>
    look_at <5, 2, 0>
    angle 70
}

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

object {
    Floor
    pigment { color rgb <1, 0.6, 0.3> }
}

object {
    Torus
    pigment { color rgb <0.3, 0.6, 1> }
    rotate <0, 0, 40>
}

```

---

### 3.2.3 CSG の素材としての平面

CSG を使って形状を作るときに、何らかの形状の一部を平面ですばっと切り落としたい、ということがしばしばあります。そのためのもっとも簡単な方法は、その形状から平面を減算することです。

次のシーンは、土星のような形状を半分だけにした形状を作って、その形状を持つ物体を作っています。

シーンの例 csgplane.pov

---

```
#declare HalfSaturn = difference {
    union {
        sphere { 0, 1 }
        torus { 2, 0.4 }
    }
    plane { x, 0 }
}

camera {
    location <-20, 4, -40>
    look_at <0.7, 0, 0>
    angle 5
}

```

---

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

object {
    HalfSaturn
    pigment { color rgb <0.4, 0.7, 1> }
}
```

---

### 3.3 テキスト

#### 3.3.1 テキスト

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、テキストという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。

#### 3.3.2 テキストの作り方

text を使ってテキストを作る記述は、

```
text { ttf "パス名", "テキスト", 厚さ, 文字間隔 }
```

と書きます。「パス名」のところには、TrueType フォントが格納されているファイルのパス名を書きます。「テキスト」のところには、形状にしたいテキスト（これは本来の意味）を書きます。「厚さ」のところには、2次元の形状に対して加える厚さを書きます。そして、「文字間隔」のところには、文字の間隔を指定するベクトルを書きます。ここに0と書くと、普通の間隔になります。

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、テキストを形状とする物体を作っています。

シーンの例 text.pov

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

camera {
    location <0, 1, -2>
    look_at <1.1, 0.3, 0>
    angle 70
}

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

object {
    Text
    pigment { color rgb <0.3, 1, 0.6> }
}
```

---

#### 3.3.3 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、異なる文字間隔を持つ二つのテキストを作って、それぞれを形状とする物体を作っています。

シーンの例 offset.pov

```
#declare Text1 = text { ttf "timrom.ttf", "tako", 1, 0.4*x }
#declare Text2 = text { ttf "timrom.ttf", "kani", 1, 0.4*y }
```

```
camera {
  location <-0.8, 2.6, -2.4>
  look_at <0.9, 1.4, 0>
  angle 70
}

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

object {
  Text1
  pigment { color rgb <1, 0.3, 0.6> }
  translate <0, 2, 0>
}

object {
  Text2
  pigment { color rgb <0.8, 0.3, 1> }
}
```

---

### 3.3.4 同梱されているフォント

POV-Ray の公式サイトで配布されているアーカイブには、

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

という三つの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 のそれぞれによって作られたテキストを形状とする物体を作っています。

シーンの例 font.pov

---

```
#declare Text1 = text { ttf "timrom.ttf", "timrom", 1, 0 }
#declare Text2 = text { ttf "cyrvetic.ttf", "cyrvetic", 1, 0 }
#declare Text3 = text { ttf "crystal.ttf", "crystal", 1, 0 }

camera {
  location <-0.2, 0, -2.8>
  look_at <1.2, 1.2, 0>
  angle 70
}

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

object {
  Text1
  pigment { color rgb <0.7, 0.7, 1> }
  translate <0, 2, 0>
}

object {
  Text2
  pigment { color rgb <0.3, 0.8, 1> }
  translate <0, 1, 0>
}

object {
  Text3
  pigment { color rgb <0.8, 0.3, 1> }
}
```

---

## 参考文献

[Introduction,2004] POV-Team, “Introduction to POV-Ray”, 2004.

[Reference,2004] POV-Team, “POV-Ray Reference”, 2004.

[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。  
<http://www.arch.oita-u.ac.jp/a-kse/povjp/index.htm>
- [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。
- [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。

## 索引

- !, 26
- !=, 25
- #declare, 14
- #end, 24, 26
- #ifndef, 24
- #include, 23
- #version, 24
- #while, 26
- &&, 26
- \*, 6, 7
- \*/, 9
- +, 6
- , 6
- .inc, 23
- .pov, 5
- /, 6
- /\*, 9
- //, 9
- <, 25
- <=, 25
- ==, 25
- >, 25
- >=, 25
- ||, 26
- 2次元グラフィックス, 4
- 3次元グラフィックス, 4
- angle, 13
- box, 10
- brick, 29
- bumps (パターンタイプ), 39
- camera, 8, 12
- checker, 29
- color, 8, 32
- colors.inc, 23
- cone, 41
- crystal.ttf, 45
- CSG, 20
  - の組み合わせ, 22
  - の素材としての平面, 43
- cylinder, 41
- cyrvetic.ttf, 45
- dents (パターンタイプ), 39
- difference, 21
- diffuse, 37
- global\_settings, 35
- glass.inc, 23
- gradient (パターンタイプ), 31
- hexagon, 29
- interior, 33
- intersection, 21
- ior, 33
- light\_source, 8, 12
- location, 12
- look\_at, 13
- max\_trace\_level, 35
- merge, 34
- metals.inc, 23
- n*次元ベクトル, 6
- object, 8, 9
- pigment, 11
- plane, 42
- POV-Ray, 4
  - の使い方, 4
- POV-Team, 4
- reflection, 37
- RGB, 8, 32
- rgb, 8
- rgbf, 32
- ripples (パターンタイプ), 39
- rotate, 18
- roughness, 37
- scale, 17
- shapes.inc, 23
- shapes2.inc, 23
- shapesq.inc, 23
- skies.inc, 23
- sky\_sphere, 36
- specular, 37
- sphere, 10
- stars.inc, 23
- stones.inc, 23
- text, 44
- textures.inc, 23

- timrom.ttf, 45
- torus, 42
- translate, 16
- TrueType フォント, 44
- union, 20, 34
- version, 24
- waves (パターンタイプ), 39
- woods.inc, 23
- wrinkles (パターンタイプ), 39
- x, 7
- $x$  座標, 7
- $x$  軸, 7
- y, 7
- $y$  座標, 7
- $y$  軸, 7
- z, 7
- $z$  座標, 7
- $z$  軸, 7
- 明るさ
  - 光源の——, 12
- 値, 5
- アンダースコア, 14
- アンチエイリアシング, 5
- 位置
  - カメラの——, 12
- 移動, 16
- イメージマップ, 29
- 入れ子
  - 繰り返しの——, 26
- 色, 8
  - の記述, 8
  - 光源の——, 12
  - 透明な——, 32
- インクルード, 23
- インクルードファイル
  - の書き方, 24
- インクルードファイル, 23
- 英字, 14
- 演算, 6
- 演算子, 6
- 円錐, 41
- 円錐台, 41
- 円柱, 41
- 凹凸, 39
- 改行, 8
- 回転, 18
  - の順序, 19
- 回転楕円体, 19
- 書き方
  - インクルードファイルの——, 24
- 拡散反射, 37
- 拡大, 17
- 加算, 6
  - ベクトルの——, 6
- 合併, 20, 34
- カメラ, 8, 12
  - の位置, 12
  - の視野の角度, 13
  - の方向, 13
- カラーマップ, 29, 30
- カラーリスト, 29
- カラーリストピグメント, 29
- 環境光, 11
- 関係演算子, 25
- 記述
  - 色の——, 8
  - 光源の——, 12
  - 物体の——, 9
- 基本ベクトル, 7
- 球, 10
- 共通部分, 21
- 鏡面反射, 37
- 空白, 8
- 屈折率, 33
- 組み合わせ
  - CSG の——, 22
- 組み込み識別子, 7
- 繰り返し, 26
  - の入れ子, 26
  - 変形の——, 27
- 計算
  - の複雑さ, 35
- 形状, 9
- 減算, 6, 21
- 現実的な
  - 空, 36
- 原点, 7
- 光源, 8, 11
  - の明るさと色, 12
  - の記述, 12
- 構成要素
  - シーンの——, 8
- コメントアウト, 9
- 差, 21
- 座標, 7
- 座標系, 7

- シーン, 4
  - の構成要素, 8
- シーン記述言語, 4
- 式, 5
- 識別子, 14
  - の使い方, 14
  - の作り方, 14
- 軸, 7
- 次元, 6
- 縞模様, 31
- 視野の角度
  - カメラの——, 13
- 順序
  - 回転の——, 19
  - 変形の——, 18
- 条件, 25
- 乗算, 6
  - ベクトルの——, 7
- 焦点距離, 13
- 除算, 6
- 数字, 14
- スカラー, 6
- 成分, 6
- 積, 7, 21
- セミコロン, 15
- 宣言, 14
- 宣言する, 14
- 空, 36
  - 現実的な——, 36
- タブ, 8
- 注釈, 9
- 直方体, 10
- 使い方
  - POV-Ray の——, 4
  - 識別子の——, 14
- 作り方
  - 識別子の——, 14
- テキスト, 44
- テクスチャー, 9, 28
  - の要素, 28
- 透明な
  - 色, 32
- トーラス, 42
- トランスミット, 32
- ノーマル, 28, 39
- ハイライト, 37
- パターン
  - 立方体の——, 29
  - 煉瓦の——, 29
  - 六角柱の——, 29
- パターンタイプ, 31, 39
- パターンタイプ名, 31
- 光の三原色, 8
- ピグメント, 11, 28
  - に対する変形, 30
- ピグメントマップ, 29
- 左手系, 8
- 評価, 5
- 標準インクルードファイル, 23
- フィニッシュ, 28, 37
- フィルター, 32
- フォント, 45
- 複雑さ
  - 計算の——, 35
- 物体, 8, 9
  - の記述, 9
- プリミティブ, 9, 41
- 平面, 42
  - CSG の素材としての——, 43
- ベクトル, 6
  - の加算, 6
  - の乗算, 7
- 変形, 16
  - の繰り返し, 27
  - の順序, 18
  - ピグメントに対する——, 30
- 方向
  - カメラの——, 13
- ホワイトスペース, 8
- 右手系, 8
- 文字間隔, 44
- モデリング, 4
- 優先順位, 6
- 要素
  - テクスチャーの——, 28
- 立方体
  - のパターン, 29
- リテラル, 5
- 煉瓦
  - のパターン, 29
- レンダラー, 4
- レンダリング, 4, 5, 12
- 六角柱

——のパターン, 29  
論理演算子, 25  
和, 6, 20