# Blender Cycles レンダラーでレイマーチングしてみよう

OSL スクリプトを使った レイマーチング・第 2 版

Q@スタジオぽぷり
@popqjp

作者 Twitter アカウント
Q@スタジオぽぷり

# 目次

レイマーチング法 … 3

OSLによるレイマーチング法のスクリプト作成 … 5

Cycles で光線を飛ばして厚みを調べる … 12

レイマーチング法による影の計算 … 14

距離関数の編集 … 15

擬似的なアンビエントオクルージョン(AO)とフォグ … 23

位置情報の利用 … 24

シーン内のオブジェクトしてレイマーチング法を活用する … 26

フラクタル図形とレイマーチング法 … 35

不正確な距離関数 … 38

ポリゴンの裏面に表示する … 43

そのほかのテクニックなど … 46

終わりに … 49

付録：基本形状 … 51

付録２：サンプルコード … 54

# Blender Cycles レンダラーでレイマーチングしてみよう

### OSL スクリプトを使った レイマーチング・第２版

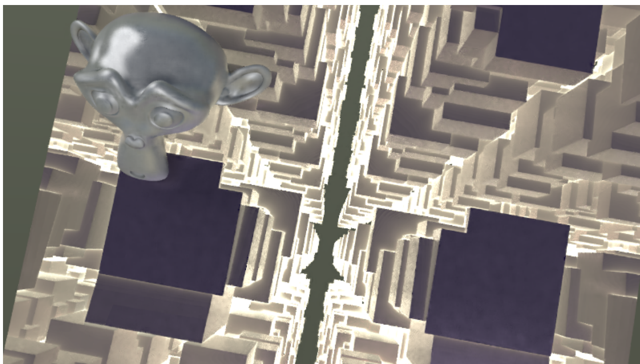
作者 Twitter アカウント
Q@スタジオぽぷり

2018.10.13 ver 1.0
2018.11.17 ver 2.0 (trace命令関係を追加)

## レイマーチング法

グラフィック表示のデモなどで、つやつやと美しく輝く幾何学構造が無限に繰り返すような映像を見たことがありますか？
また、細密なフラクタル構造の中を飛行するような映像を見たことがあるでしょうか。
こうした映像の多くは、レイマーチング法 (Ray Marching) というポリゴンモデルを使わないレンダリング手法で作成されています。

CGでポリゴンモデルを表示する典型的な方法のひとつは、有名なレイトレーシング法です。レイトレーシング法では（現実世界の光線の進む向きとは逆向きですが）カメラから仮想の光線を飛ばして、光線がポリゴンの表面にぶつかる位置や、その位置でのポリゴンの法線を計算します。
一方、レイマーチング法では図形を数式として表現します。光線が図形にぶつかる位置やその位置での法線を、単純な計算を何度も繰り返し計算することで求めることができます。
この「単純な計算を何度も繰り返し計算する」という作業は、特に GPU が大得意とするものです。CPUによるレンダリング以上に高速に、それこそリアルタイムで表示をすることも可能ということで、レイマーチング法は、そうしたデモ映像や一部のゲームなどで導入されています。



ここでは、Blender 上 Cycles レンダラー上で、OSL (Open Shading Language) という言語を使ってレイマーチング法を行う方法を解説します。ただし、この計算は CPU 上で行うために、GPUで行ったレイマーチング法ほどの速度は出ません。

その代わり、Cycles 上で行ったレイマーチングは、周囲の物体への反射や影の表現といったレイマーチング法単独では難易度の高い効果も有効になります。これは、Cycles が基本的にレイトレーシング法によるレンダリングを行い、その一部にレイマーチング法が組み込まれる形になるためです。
また、OSL で作成したレイマーチング用のスクリプトは、ちょっとした改造などで OpenGLなどを用いたリアルタイム表示用のシェーダーにすることも比較的容易だと思います。

本解説の主な内容としては、OSL スクリプトの作成とノード編集になります。OSL はプログラミング言語としては C 言語に似ているものですので、C 言語の流れを汲むプログラミング言語をある程度習得していることが必要になります。また、Blender でのノード編集の初歩的な知識も必要でしょう。
また、高校生レベルの線形代数、幾何学の知識は必要になるかと思います。

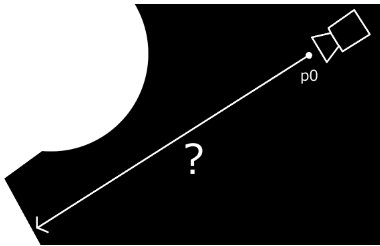
※）ここで解説している方法は、厳密にはレイマーチング法そのものではなく、広い意味でのレイマーチング法の中の１つの方法 (Sphere tracing) になるようです。しかし、多くの場合、この Sphere tracing 法をレイマーチング法として言及されるようですので、ここでもその流れに従ってレイマーチング法と呼ぶようにします。

### レイマーチング法の基本

レイマーチング法は、レイトレーシング法と考え方は似ています。 レンダリングをしたときに最終的に得たいものは画像です。画面上のピクセルごとに何色を表示するべきかの情報が必要になります。
あるピクセルの表示するべき色を調べるために、そのピクセルに対応する方向にむけて、カメラから仮想の光線を飛ばします。その光線が、例えば緑の物体にぶつかるのであればそのピクセルは緑に、赤い物体にぶつかるのであればピクセルは赤く表示すれば良いことになります。この処理を、

すべてのピクセルに対して繰り返し計算をすれば、画像が得られるわけです。

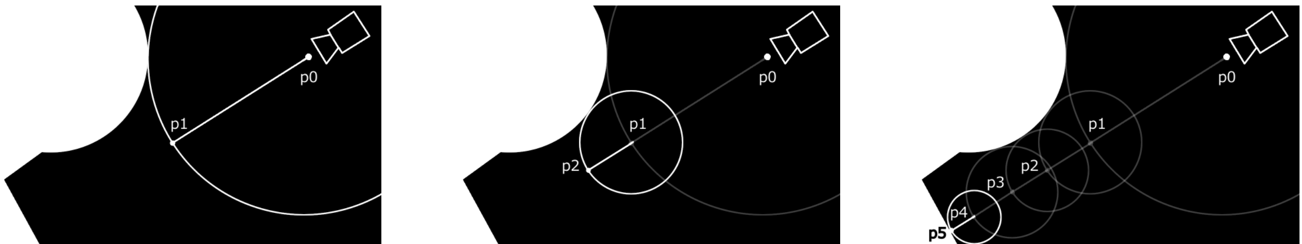
カメラから飛ばした光線がp0から出発した場合に、図形とぶつかる位置を計算します。このとき、光線は前にだけ飛ぶことができるとします。




このとき、ポリゴンとの交点を計算するのであれば、理論上すべてのポリゴンと交差するかどうかをチェックして、交点を求めます。実際には、交差するはずのないポリゴンとの計算を省いて高速化するための工夫をすることになり、そこが技術の見せ所の一つになるわけです。

交点が分かれば、さらにその位置での法線の方向や、光源の方向との関係、マテリアルの特性などから表示するべき色を計算して表示をすることになります。 もし、影や反射、半透明や屈折の効果を入れたい場合には、その交点からさらに光線を飛ばします。例えば光源方向に光線を飛ばしたときに、途中で何かに遮られればそこは光の届かない影の領域です。光源まで光線が到達すれば、逆にそこは光源から光があたる領域というわけですから、光源の強さに応じて明るく見えるように色を調整すれば良いわけです。

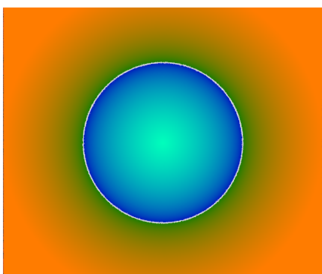
レイマーチング法では、光線と図形との交点を計算したい時、図形の形状そのものの情報は無いけれども、なぜか魔法のように「ある点から図形までの最短距離を計算する式 F(p)」がわかっているとします。p0 からの最短距離F(p0)が分かるのであれば、少なくとも p1 まで進んでも図形の中にめり込んだりしないことがわかります。
p1 まで進んだとして、また p1 からの最短距離F(p1)が分かれば、図形の中にめり込まないで p2 まで進めます。この処理を繰り返すことで、図形とぶつかるぎりぎりまで光線が接近できます。




このやり方は、光線が無限に図形の表面に近づいていきますが、そのままでは図形の表面まで到達しません。ある閾値 C を決めて F(p) < C となった時に図形の表面まで到達したと判断することとします。また、図形に衝突せずにそのまま反対側に抜けていくことも考えられます。繰り返し回数には上限を設定し、上限まで達したら光線と図形は交錯しないと判定します。

この方法のよいところは、「F(p)さえ分かれば」単純な計算の繰り返しで光線と図形が交わる点がわかることです。
しかし、そのような都合のよい式 F(p) は存在するのでしょうか？
すべての形状を表現するような F(p) は存在しませんが、たとえば中心位置が q で半径 r の球のような図形であれば、簡単な式で表現することができます。

F(p) = length(p - q) - r

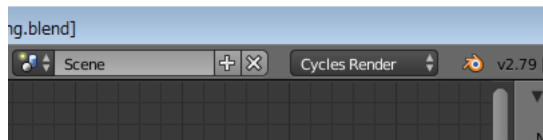


球に対して距離関数を、正の領域を暖色の、負の領域を寒色のグラデーションで表現した図です。
こうした単純な距離関数を組み合わせ、座標変換などのテクニックを駆使することで、ポリゴンで表現することの難しい複雑な形状もあらわすことができます。（逆に、ポリゴンで表現されたキャラクターの距離関数などを求めるのは絶望的であることが分かるでしょう）
レイマーチング法は、幾何的な模様や、後の述べるフラクタル形状をレンダリングするのに向いていて、レイトレーシング法とは得意な図形の形が違います。レイトレーシングを行う Cycles 上での レイマーチングは、上手く使えばお互いの得意な部分を補完することができるのです。

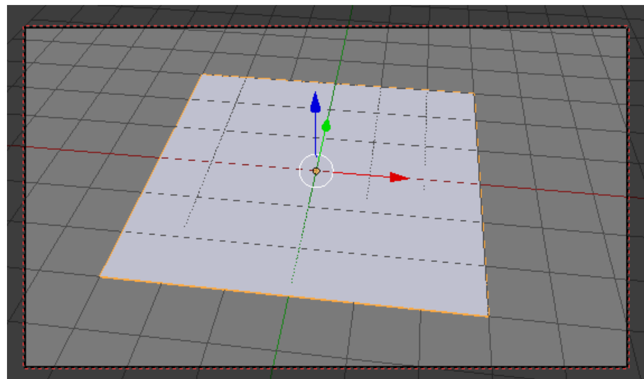
では、まず距離関数を用いて球を表現することを目標に、OSL スクリプトを組んでみましょう。

## オブジェクトとマテリアルを作成する

最初にレンダラーとしてCycles を設定しておきます。



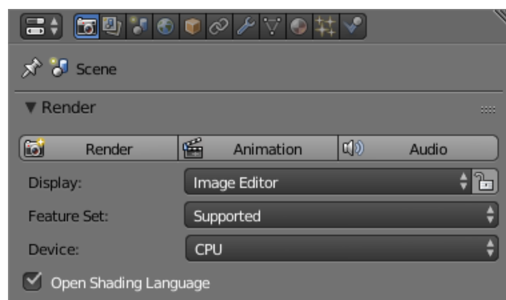

OSL スクリプトはオブジェクトが描画されるときに、そのマテリアルを計算する時に実行されます。描画用のキャンバスとしてシーン内に一枚の板を配置しておきます。 レイマーチング法でレンダリングされた結果は、この板の上に描画されることになります。



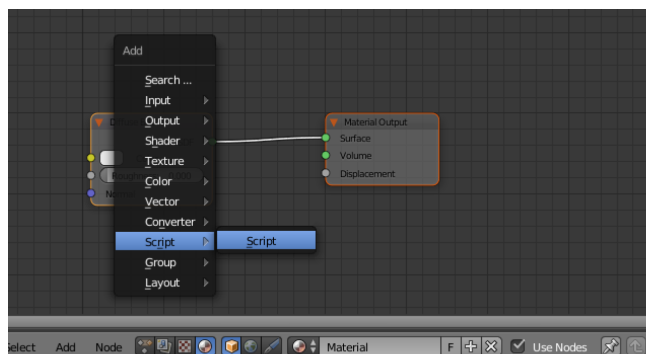
### OSLスクリプトを有効にする

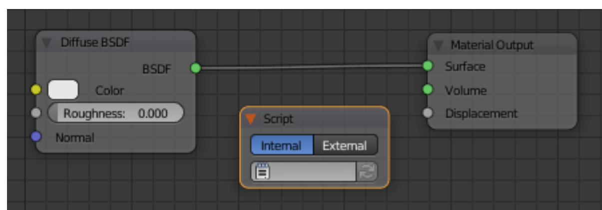
Cycles 使用時には、Open Shading Language (OSL) を使ってユーザーが自由に設定したマテリアルノードを作成することができます。（ただしCPUレンダリング時のみ有効です。）

blender 2.79 において OSL を有効にするにはレンダリングのモードを Cycles に設定します。また、レンダリング設定で Open Shading Language を有効にします。




マテリアルのノード編集画面で、スクリプトノードを追加します。実行するスクリプトを、内部テキスト(Internal)または外部のファイル(External)から選択することで、OSL を実行することができます。

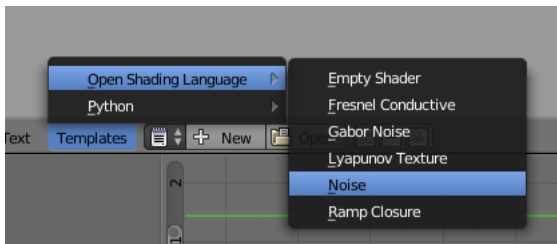



blender 2.80 においては、OSL は常に有効になっています。しかし、2.80 Alpha 版の時点では Eevee における OSL は実装されておらず、Cycles の CPU レンダリング時のみ有効です。

## OSL によるレイマーチング法のスクリプト作成

Blender 標準のテンプレートとして、OSL スクリプトのサンプルが用意されています。まずは ノイズを作成するテンプレート(noise.osl)を呼び出して中身を見てみましょう。



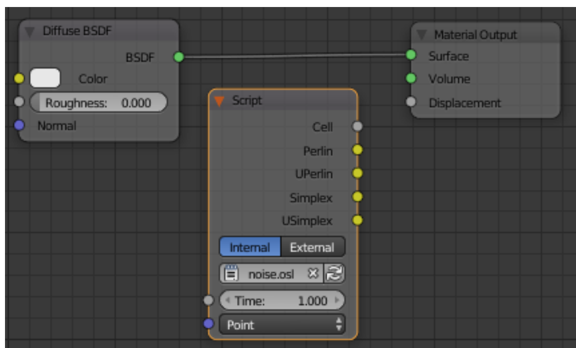

noise.osl

```
shader noise(
        float Time = 1.0,
        point Point = P,
        output float Cell = 0.0,
        output color Perlin = 0.8,
        output color UPerlin = 0.8,
        output color Simplex = 0.8,
        output color USimplex = 0.8)
{
        /* Cell Noise */
        Cell = noise("cell", Point);

        /* Perlin 4D Noise */
        Perlin = noise("perlin", Point, Time);

        /* UPerlin 4D Noise */
        UPerlin = noise("uperlin", Point, Time);

        /* Simplex 4D Noise */
        Simplex = noise("simplex", Point, Time);

        /* USimplex 4D Noise */
        USimplex = noise("usimplex", Point, Time);
}
```

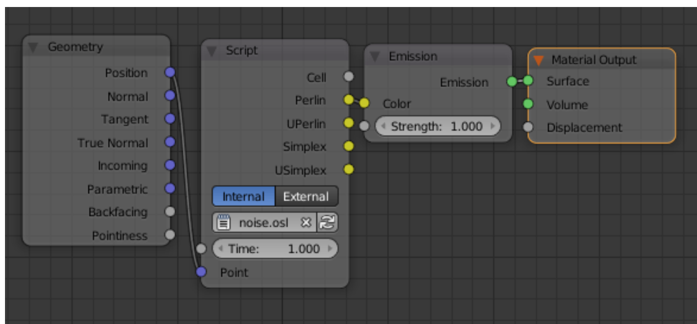
noise.osl の中身を見ると OSLスクリプト の基本形が、

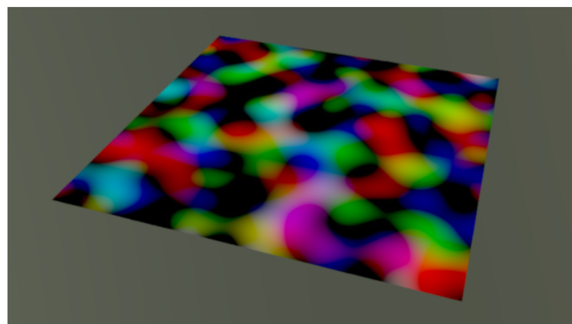
shader noise(...入出力設定...){...スクリプトの中身...}

という形になっていることが分かります。
マテリアルのノード編集画面で、スクリプトノードをnoise.osl に設定します。




ノードに入力と出力ソケットが追加されました。これらのソケット間をつないでマテリアルの設定を行うことができます。位置情報から Perlin ノイズを使って色を設定するマテリアルになるようにノードを組み立ててみます。



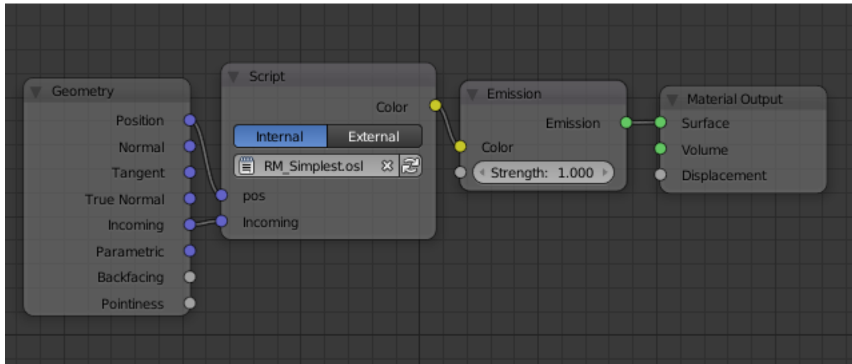



OSL スクリプトで設定したノードも、通常のノードと同じように使用できことが分かります。
OSL スクリプトは、Cycles レンダリングを CPU 上で行ったときのみ有効です。(2.79, 2.80 アルファ 時点)
機能しない場合などは設定の確認をして見ましょう。

## 非常に単純なレイマーチング法のスクリプト

それでは、いよいよ実際にレイマーチング法を実行するためのスクリプトを作成します。
Cycles のレイトレーシングで行われる処理では、まずカメラから放射された光線が、板オブジェクトに到達します。
その板オブジェクト上の位置から出発し、入射した方向にそのまままっすぐ進む光線を、レイマーチング法で計算すればよいことになります。

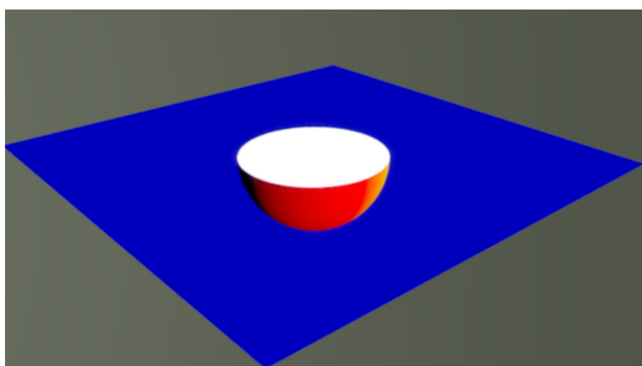
入力には板オブジェクト上の位置(pos)と光線の入射ベクトル(Incoming)を設定し、出力にはレイマーチングの計算結果を出力するための色(Color)を設定します。
ノードの接続は次のように設定し、レイマーチング用のスクリプト RM_Simplest.osl を作成します。




RM_Simplest.osl

```
#define Iterations 50
#define Crit 0.001

float Sphere(point pos, float radius){
    return length(pos) - radius;
}

float DE(point posIn){
    return Sphere(posIn, 1);
}

point RayMarch(int count){
    point pos = P;
    for (count = 0; count < Iterations ; count++) {
        float dist = DE(pos);
        if (dist < Crit) break;
        pos -= dist * I;
    }
    return pos;
}

shader RaymarchingBasic(
    point pos = P,
    vector Incoming = I,
    output color Color = color(1,1,1))
{
    int count = 0;

    if (DE(pos) > Crit){
        point surf = RayMarch(count);
        Color = surf;
    }

    if (count == Iterations) Color = color(0, 0, 0.5);
}
```

このマテリアルを用いてレンダリング実行すると、結果は次のような画像になるはずです。表示用の板は z=0 平面に存在しているため、原点にある球は真っ二つに切られた状態です。色には球の表面の座標情報を用いました。



スクリプトの中身を確認します。

```
shader RaymarchingBasic(
    point pos = P,
    vector Incoming = I,
    output color Color = color(1,1,1))
{...}
```

入力ソケットに、板オブジェクト上の位置 pos と光線の入射ベクトル Incoming (カメラ方向側を向いています)があります。
初期値として P, I がありますが、これはデフォルトで用意されているグローバル変数で、位置ベクトルと 光線の入射ベクトルを表しています。ですから、デフォルト値を使うのであれば本当はソケットは必要ないのですが、デフォルト以外の値も後で使用するので、あえてソケットをつないでいます。 出力としては、色情報を出力するようにしています。初期値は 白です。

メインの関数の中で変数 count を定義しています。これはレイマーチングの繰り返し回数を記録するための変数です。
（C言語と違い、引数は参照渡しになります。レイマーチを実行する関数RayMarch内で count を増やすと、メイン関数での変数も変化します。）

```
float Sphere(point pos, float radius){
    return length(pos) - radius;
}

float DE(point posIn){
    return Sphere(posIn, 1);
}
```

これら２つの関数は、距離関数を返す関数です。ここでは球の距離関数を使って定義しています。

メイン関数の中からレイマーチングのルーチンを呼び出します。

```
    if (DE(pos) > Crit){
        point surf = RayMarch(count);
        Color = surf;
    }
```

if文を使って、初期の距離がすでに閾値以下の場合は、計算のスタート地点がすでに球の内部にあるということでレイマーチの実行をせず、出力する色が初期値の白のままになるようにしてあります。

Raymarch() がレイマーチを行う本体部分ですが、戻り値として光線と球が交差した場所を返すようにしてあり、この座標を色として出力するように 変数 Color に代入しています。 また、レイマーチで繰り返した回数も変数 count で得られるようにしてあります。

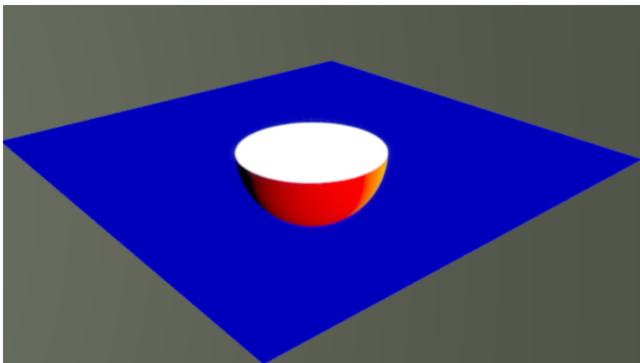
```
point RayMarch(int count){
    point pos = P;
    for (count = 0; count < Iterations ; count++) {
        float dist = DE(pos);
        if (dist < Crit) break;
        pos -= dist * I;
    }
    return pos;
}
```

レイマーチ部分の本体を見てみましょう。
位置 pos を初期位置 P にあわせて、計算を開始しています。
for ループをまわして、距離 dist が閾値以下になるまで dist * I (距離×入射ベクトル)を繰り返し足して（引いて）います。
（入射ベクトルはカメラの方向を向いているので、符号を逆にしています）

距離 dist が閾値以下になれば、光線と球の交点が求まったということですから、その位置を返します。

```
if (count == Iterations) Color = color(0, 0, 0.5);
```

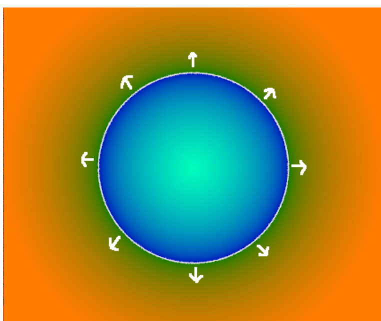
最後に、メイン関数内で count を調べています。球にぶつからずに繰り返し回数の上限に達したということは、光線は球の範囲外をすり抜けていったということです。今回は色を青にすることにしました。



以上のような処理の結果としてこのような図が得られたことになります。

### 法線方向を計算する

光の当たり具合などから、見えるべき色を計算するためには図形表面の法線が分からなければなりません。
実は、距離関数の勾配を計算するだけで法線がもとまります。

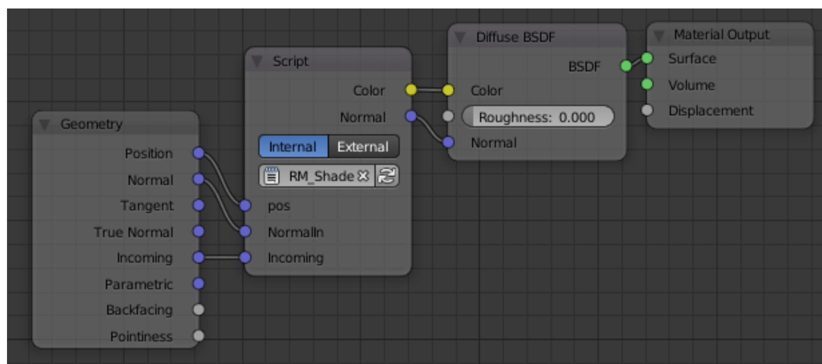


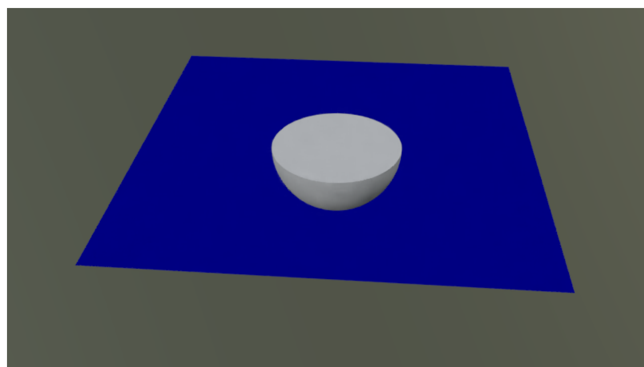
距離関数を色で見てみると分かりやすいので、先ほどの球の距離関数を見てみましょう。 勾配というのはもっとも変化の大きい方向を示しているわけですから、物体表面 F(x) = 0 の等値面、色の変わらない方向、にたいして垂直になります。すなわち、法線方向ですね。
そこで、法線方向を計算する関数 GetNormal(point) を導入して、レイマーチ法で得られた法線方向を出力するスクリプトに変更してみます。

RM_ShadeOnly.osl

```
#define Iterations 300
#define Crit 0.001
#define dP 0.001

float Sphere(point pos, float radius){
    return length(pos) - radius;
}

float DE(point posIn){
    return Sphere(posIn, 1);
}

point RayMarch(int count){
    point pos = P;
    for (count = 0; count < Iterations ; count++) {
        float dist = DE(pos);
        if (dist < Crit) break;
        pos -= dist * I;
    }
    return pos;
}

normal GetNormal(point pos){
    float DEBase = DE(pos);
    normal nv = normal(
        DE(pos + point(dP, 0, 0)) - DEBase,
        DE(pos + point(0, dP, 0)) - DEBase,
        DE(pos + point(0, 0, dP)) - DEBase);
    return normalize(nv);
}

shader RaymarchingBasic(
    point pos = P,
    normal NormalIn = N,
    vector Incoming = I,
    output color Color = color(1,1,1),
    output normal Normal = NormalIn
    )
{
    int count = 0;

    if (DE(pos) > Crit){
        point surf = RayMarch(count);
        if (count < Iterations)
            Normal = GetNormal(surf);
    }

    if (count == Iterations) Color = color(0, 0, 0.5);
}
```

ポリゴンの法線を入力するソケットを追加し、球の内部や、光線が球に交差しない場合は入力された法線がそのまま出力されるようにしてあります。 法線に関するソケットを追加されたので、Diffuse ノードを接続してマテリアルを設定します。

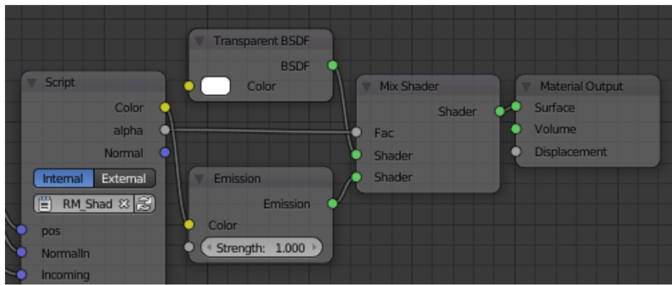


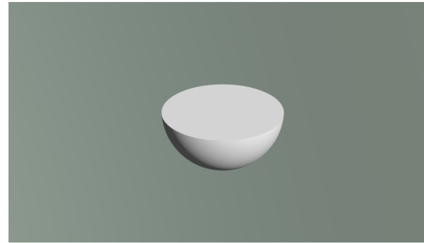



また、光線が球と交差しない場合に、青の色を置く代わりに透明にすることも可能です。
出力する変数に output float alpha = 1.0 を追加して、count が上限に達したときに色に青を代入する代わりに alpha に0をセットします。

```
if (count == Iterations) alpha = 0;
```

あとは、ノード編集でアルファ値に応じて透明シェーダと Mix することで、透明を表現できます。



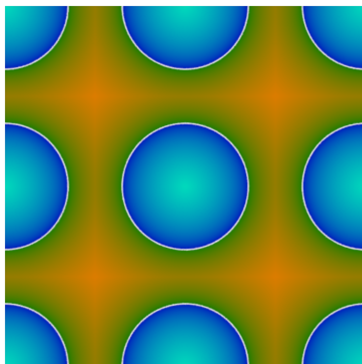



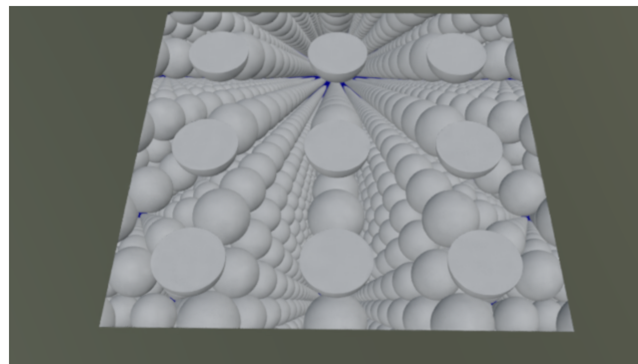
## 距離関数の編集・繰り返し

では、ここでより複雑な距離関数 F(p) を作成する方法を見てみます。割り算の余りを計算する関数 mod(x, s) を利用してみます。この関数は x が 0 から増やしていくときにある値 s を超えるとまた 0 に戻るわけですから、繰り返しをあらわしているともいえます。
ただし、どちらかといえば [0,s]間での繰り返しよりも、[-s/2, +s/2]間での繰り返しのほうが使い勝手が良いわけですから、少し工夫をして次のような point を繰り返す関数 ModPoint を定義してみます。
これを使って、球が無限に配置された距離関数が実現できます。

```
point ModPoint(point pos, point size){
    return mod(pos+size/2, size) - size/2;
}

float DE(point posIn){
    return Sphere(ModPoint(posIn, point(2,2,2)), 0.5);
}
```

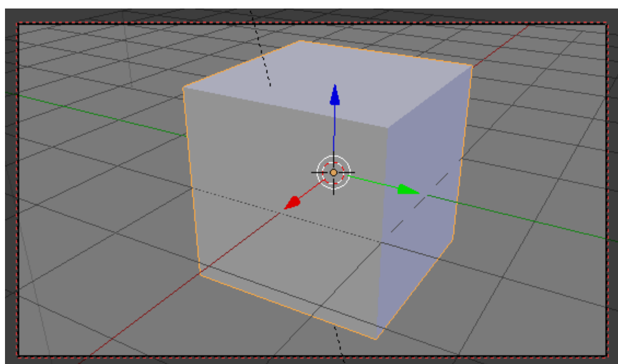
繰り返しを使った距離関数と、そのレンダリング結果は次のようになります。無限に連なる球が表現できました。

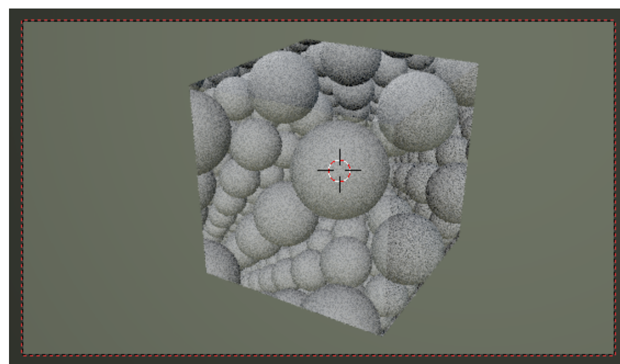




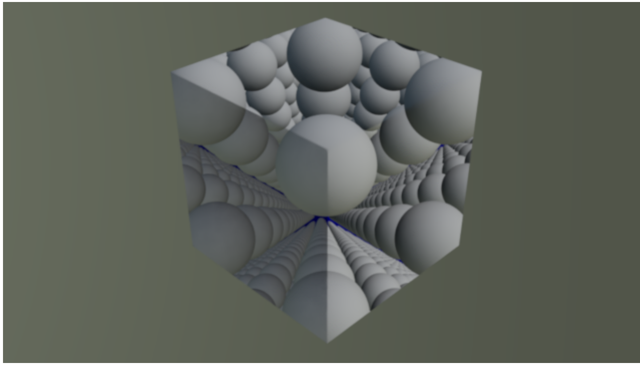
## 平面以外へのマッピング

ではここで、レイマーチング法で球を描くマテリアルを、板ではなく立方体の上に設定します。すると、板ではなく立方体の中に球が配置されることになります。
このまま視線を動かすと、まるで球で埋め尽くされた別の世界への窓が、立方体を通して開いているかのように見えます。





ところが、ここで Diffuse シェーダーを使っているときに光源の反対側に回り込んでみると、元の立方体の形状の陰影が浮かびあがって見えてしまいます。



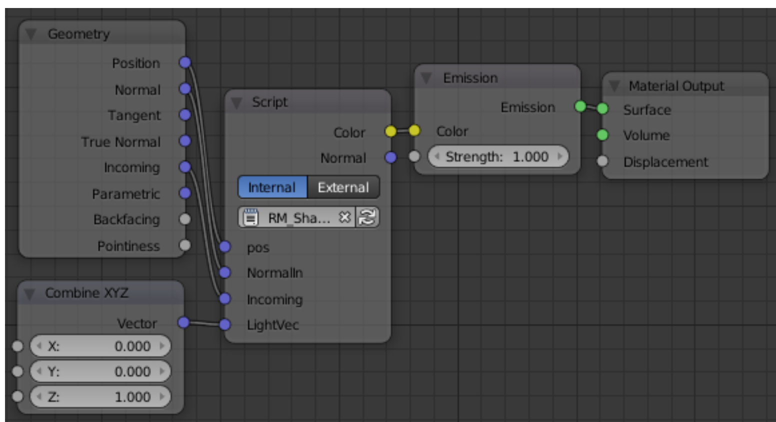
これは、Diffuse シェーダーでは、ポリゴンに到達する光の量にしたがって明るさが決まるために起きています。
レイマーチ法によって得られた色などの情報はポリゴン表面の色として反映されますが、光源の反対側のポリゴンに届く光の量が少ないので暗くなっているのです。
これは、レイマーチ法による結果がポリゴンの上に色として描かれるという性質上、避けにくい問題点の１つです。

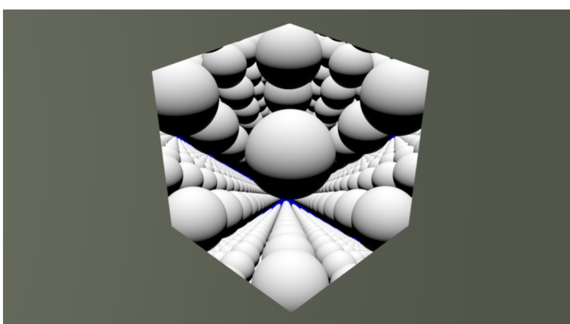
回避方法としては

- あまり目立たない用途（表面のちょっとした立体感のある模様など）にレイマーチング法を使う
- レイマーチング法で描きたい形状にあわせて、近い形状のポリゴンを使う。
- 陰影はレイマーチング法の中で処理して、ポリゴンに届く光の量に影響されない Emission シェーダーを使う

などが考えられます。 ここでは、３番目の方法を試してみます。

```
shader RaymarchingBasic(
    point pos = P,
    normal NormalIn = N,
    vector Incoming = I,
    vector LightVec = vector(0, 0, 1),
    output color Color = color(1,1,1),
    output normal Normal = NormalIn
    )
{
    int count = 0;

    if (DE(pos) > Crit){
        point surf = RayMarch(count);
        if (count < Iterations)
            Normal = GetNormal(surf);
    }

    float difFactor = max(0, dot(LightVec, Normal));
    Color *= difFactor;

    if (count == Iterations) Color = color(0, 0, 0.5);
}
```

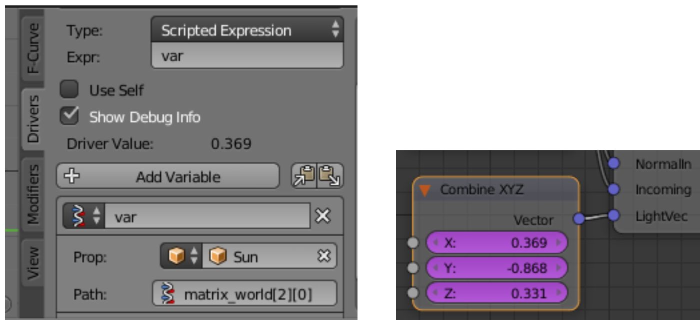


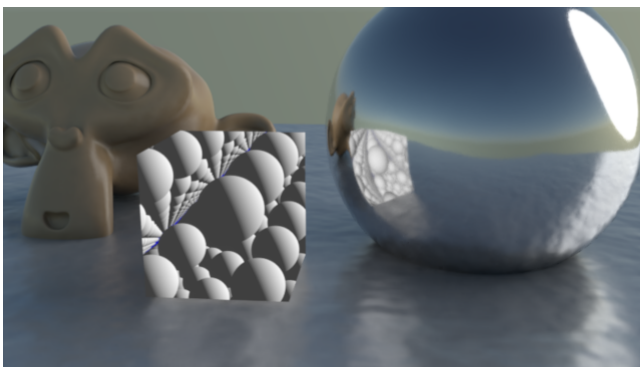
入力ソケットに光の入射ベクトルを設定しています。光の向きと法線の方向の内積をとり、単純な diffuse のファクター(difFactor)を計算して出力

する色を変化させています。
この色を Emission シェーダーに接続することにより、ポリゴンにあたる光の量とは無関係にレイマーチング法で計算した空間を表示することができます。

しかし、ここで入射ベクトルを手入力して設定するのは大変です。シーンに実際に配置した光源の向きとあわせたいところです。ドライバーを使って実現するのが良いでしょう。




知っていると便利な豆知識として、平行光線の光源(Sun)の光線の向きのベクトルは(matrix_world[2][0], matrix_world[2][1], matrix_world[2][2]) と同じになるので、ドライバーなどで呼び出して使うのに便利です。



外部の光源の向きとあわせてレイマーチング法で球の連なりを描画しています。レイトレーシングの一部としてレイマーチング法が行われているため、右の球体の反射にレイマーチング法による球が写りこんでいます。

## Cyclesでの光線を飛ばして、オブジェクトの厚みを調べる

前セクションで表示したシーンでは、球の連なりが無限遠まで続いています。これはこれで面白い効果ですが、ある程度光線が進んだら、ポリゴンの外に飛び出したとして判定を打ち切って透明にしたいところです。
そのためには、カメラから見た方向についてのオブジェクトの「厚み」を知らなければなりませんが、OSLスクリプトに入力する情報は、オブジェクトの形状には無関係なものなので、それだけでは「厚み」が分かりません。OSLスクリプトの内部で、オブジェクトの「厚み」を得る方法を見てみます。

### オブジェクトの「厚み」を得る

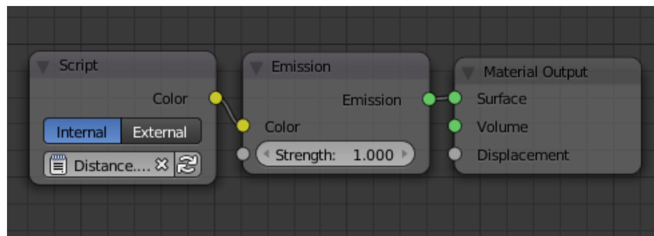
OSL スクリプトの実行中に、Cycles としての光線を飛ばして衝突判定をするという機能 trace があります。それを利用することでオブジェクトの「厚み」を得ることができます。
この時、Cycles として飛ばしている光線と、レイマーチングとして飛ばしている光線を混同しないように気をつけてください。

```
int trace (point pos, vector dir, ...) //Cyclesの光線を飛ばして衝突の判定をする
```

trace命令は位置 pos から 方向 dir に向かって光線を飛ばします。ポリゴン表面にぶつかるならば 1 を、ぶつからなければ 0 を返します。
その後、直前に実行した trace 命令によって衝突したオブジェクトの名前、衝突までに飛んだ距離などの情報を getmessage という命令によって得ることができます。
getmessage 命令は、引数に与える文字列によってどのような情報を得るかを指定します。今回用いる典型的な情報としては以下のようになります。

```
float distance;
getmessage("trace", "hitdist", distance); //距離を得る
string name;
getmessage("trace", "geom:name", name); //衝突したオブジェクトの名前を得る
```

この機能を使用することで飛ばす光線の計算は、Cycles がレンダリングのために自前で飛ばしている通常の光線の計算に比べると速度は大きく低下するそうです。
そのため大量に光線を飛ばすような処理には向いていないですが、幾つかの光線を飛ばして、周囲の状況を調べ、それに従ったシェーダーを作成するような場合に活躍します。
まず、例としてオブジェクトの「厚み」を色として見るシェーダーを作成してみます。




```
shader GetDistance(
    output color Color = color(1,1,1),
    )
{
    float objectDepth = 0;
    if (backfacing() == 0){
        if (trace(P, -I) == 1)
            getmessage("trace", "hitdist", objectDepth);
    }

    Color = objectDepth * 0.1;
}
```

OSLの中身を見てみます。まず、float objectDepth という変数を用意して、これに厚み情報を代入する用意をしています。次に表向きのポリゴンだけで評価をするために、backfacing() を使ってポリゴンの表裏を判定しています。
その後、ポリゴンの表面位置(P) から、カメラから放射された光線の方向(-I)に向けて trace 命令で光線を飛ばして衝突判定をします。
もし衝突が起きたならば、getmessage 命令でその情報を取り出します。
この場合は、光線が飛んだ距離がカメラから見たオブジェクトの「厚み」に相当するわけです。最後に出力する色に、得た値を代入しています。

立方体に対してこのシェーダーを適用した場合と、中にスザンヌを置いて適用した場合を見てみます。

ところで、上の例文でわざわざ backfacing() を使ってポリゴンの表と裏を判定したのはなぜでしょうか？
光線が、箱の内側から外向きにポリゴンにぶつかった場合、そこでスクリプトが実行されて、trace 命令によってそこから外向きに光線を飛ばして距離の測定をしても、箱からさらに外にある外部のオブジェクトとの距離を測定することになってしまいます。それでは箱の厚みを計算したことにはなりません。
そのような場合には、後述するように、場合分けをしてコードの一部を変えて厚さを評価する必要があります。
このように表と裏とで別の処理をしなくてはならない場合があるために、backfacing()でポリゴンの表裏を確認しておくことがしばしば重要になります。

しかし、カメラから飛ばした光線が、箱の内側から外に向かってポリゴンにぶつかることはあるのでしょうか？
カメラがオブジェクトの中に入り込んでいるとすると、カメラから出た光線は内側から外向きにポリゴンにぶつかります。
また、カメラがオブジェクトの外にある場合でも、レイマーチで表示したいオブジェクトに空隙がある場合には、透明になる場所が発生します。すると、カメラから出た光線がオブジェクトの内部に入り込み、今度は内側から反対側のポリゴンにぶつかって、そこで再度レイマーチの計算が始まることになります。
trace 命令を使ってオブジェクトの形状を調べる場合には、ポリゴンをどちらから見ているかを意識することが必要な場合が多くなります。

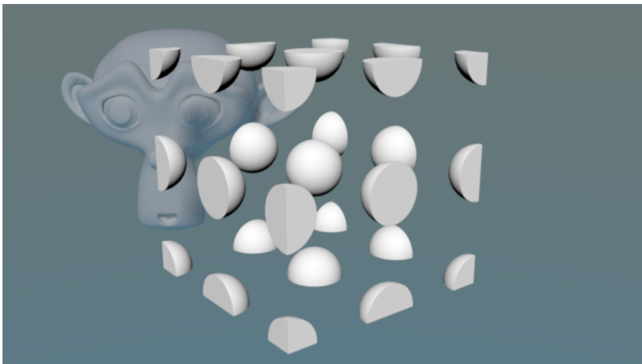
## オブジェクトの厚みを考慮したレイマーチング

trace命令を使って得た厚み情報を使ってレイマーチングを行うスクリプトは以下のようになります。

RM_Depth.osl

```
#define Iterations 300
#define Crit 0.001
#define dP 0.001

float Sphere(point pos, float radius){
    return length(pos) - radius;
}

point ModPoint(point pos, point size){
    return mod(pos+size/2, size) - size/2;
}

float DE(point posIn){
    return Sphere(ModPoint(posIn, point(2,2,2)), 0.5);
}

point RayMarch(point posIn, int count, float objectDepth){
    point pos = posIn;
    float depth = 0;
    for (count = 0; count < Iterations ; count++) {
        float dist = DE(pos);
        depth += dist;
        if (depth > objectDepth) count = Iterations - 1;
        if (dist < Crit) break;
        pos -= dist * I;
    }
    return pos;
}

normal GetNormal(point pos){
    float DEBase = DE(pos);
    normal nv = normal(
        DE(pos + point(dP, 0, 0)) - DEBase,
        DE(pos + point(0, dP, 0)) - DEBase,
        DE(pos + point(0, 0, dP)) - DEBase);
    return normalize(nv);
}
```

```
shader RaymarchingDepth(
    point pos = P,
    vector Incoming = I,
    vector LightVec = vector(0, 0, 1),
    float rayLength = 1,
    output color Color = color(1,1,1),
    output normal Normal = N,
    output float alpha = 1,
    )
{
    int count = 0;
    float objectDepth = 0;
    if (backfacing() == 0){
        if (trace(pos, -I) == 1)
            getmessage("trace", "hitdist", objectDepth);
    }
    if (objectDepth == 0){ alpha = 0; return; }

    if (DE(pos) > Crit){
        point surf = RayMarch(pos, count, objectDepth);
        if (count < Iterations)
            Normal = GetNormal(surf);
    }

    float difFactor = max(0, dot(LightVec, Normal));
    Color *= difFactor;

    if (count == Iterations) alpha = 0;
}
```



これで、レイマーチングを表示しているキャンバス（この場合は箱）の形に合わせてレイマーチングの表示をすることができました。
ここでは背景にスザンヌを置いています。スザンヌに影が落ちていることも確認できます。

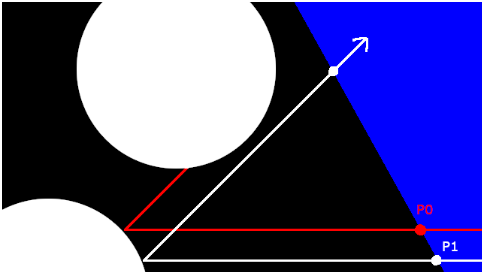
以下、簡単にスクリプトの流れを説明します。
メインの関数では最初に、キャンバス用のオブジェクトの「厚み」を計算して float objectDepth という変数に代入しています。裏面の場合には厚さ0として扱って、その場で計算を打ち切って透明としてあります。

ついで、RayMarch 関数の中でこの objectDepth を計算打ち切り用の判定値として利用しています。
RayMarch の中では、float depth という変数を用意して、レイマーチとして光線が進んだ累積距離を記憶しています。これが objectDepth を超えた場合に count 数を Iterations - 1 まで増やして計算を打ち切っています。（ループの最後に +1 されるので最終的に count の値は Iterations になります）
count 値が Iterations になっているときは、レイマーチの光線が何にも衝突せずに終了したことを示しているので、alpha = 0 として透明にしています。

さて、 backfacing() でポリゴンの裏表の確認を最初にしています。
この確認をしないと、最初の計算で透明であった部分では Cycles で飛ばしている光線が手前側のポリゴンを透過します。次に反対側のポリゴンに裏からぶつかり、そこで計算を開始してポリゴンの裏面に球を表示しようとしまいます。

## レイマーチング法における影の計算

光源の方向を指定すれば、レイマーチング法においても影の計算をすることができます。
図形の表面の座標が決定したら、そこを基点にして光源の方向にもう一度レイマーチを実行し、再度別の表面にぶつかったら(赤いライン）そこは影の領域であり、光源まで到達したら（平行光線の場合は、ポリゴンの外側まで出たら）そこは光の当たっている領域である（白いライン）と考えることができます。




ただし、実際にスクリプトを組むには若干の工夫がいります。

- 図形の表面から影判定の光線を同じ条件でスタートすると、最初から閾値以下になっているので影判定になってしまう。
- 表面にぶつからずに、どこまで光線が到達したら影では無いと判定すれば良いのか。

前者を防ぐために、例えば法線方向にわずかにずらした位置からスタートするなどの調整が必要になります。
また、後者を解決するためには、今度は形状の表面位置からやはり trace 関数によってCycles上での光線を飛ばして、どれだけ光線が飛んだらポリゴンの外にでるのか（青い部分に到達するのか）を計算しておく必要があります。

その後レイマーチング上で影の判定用の光線を飛ばして、「光線がポリゴンの外に出るまでに表面に衝突するかどうか」の判定をします。

この考え方で影判定をするためのレイマーチング法のルーチンは以下のようになります。

```
int IsShadowed(point surf, vector lightVec, float shadowDepth){
    int shadow = 0;
    float depth = 0;
    point pos = surf;
    for (int i = 0; i < Iterations * 3; i++) {
        float dist = DE(pos);
        depth += dist;
        if (depth > shadowDepth) break;
        if (dist < Crit){ shadow = 1; break; }
        pos += dist * lightVec;
    }
    return shadow;
}
```

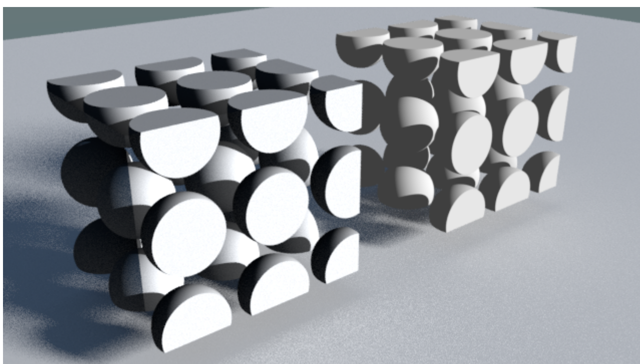
影判定用の光線の計算を開始する位置は、図形の表面(surf)になります。 引数には、光線の進む向きとして光源方向を向いたベクトル(lightVec)、光線がポリゴンの外に出たことを判定するための shadowDepth が追加されています。
レイマーチの上限回数に達するか、光線がポリゴンの外に出る前に図形内で再衝突をすれば、戻り値に１を返します。
図形と光源、ポリゴンの向きによっては、より多い回数レイマーチを繰り返さなければ影にいるかどうか判定できない場合もあるので、上限回数を３倍しています

良く見ればこの関数は、最初に作ったレイマーチ法のルーチンとほとんど同一の構造で、変数名と戻り値が少し違う程度です。今回は変数名など区別したいので別の関数にしていますが、工夫すれば同一の関数を使って処理することも可能でしょう。

上のルーチンを使って影かどうかを色に反映させたメイン関数は以下のようになります。

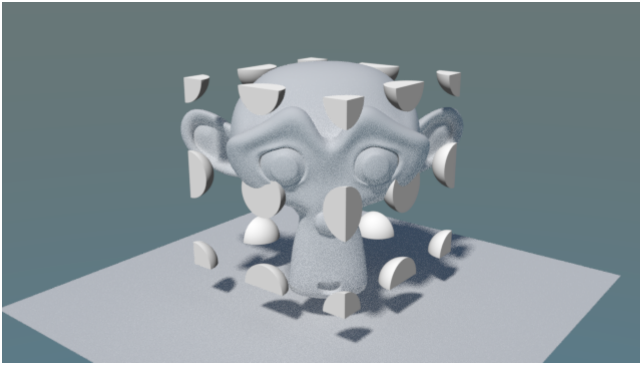
```
shader RaymarchingShadow(
    point pos = P,
    vector Incoming = I,
    vector LightVec = vector(0, 0, 1),
    float rayLength = 1,
    output color Color = color(1,1,1),
    output normal Normal = N,
    output float alpha = 1,
    output int shadow = 0,
    )
{
    int count = 0;

    float objectDepth = 0;
    if (backfacing() == 0){
        if (trace(pos, -I) == 1)
            getmessage("trace", "hitdist", objectDepth);
    }
    if (objectDepth == 0){ alpha = 0; return; }

    point surf;
    if (DE(pos) > Crit){
        surf = RayMarch(pos, count, objectDepth);
        if (count < Iterations)
            Normal = GetNormal(surf);
    }

    if (dot(LightVec, Normal) < 0) shadow = 1; //光源の反対側を向いている面は、計算するまでもなく影の中
    if (count > 0 && shadow == 0){
        float shadowDepth = 0;
        if (trace(surf, LightVec) == 1)
            getmessage("trace", "hitdist", shadowDepth);
        shadow = IsShadowed(surf + Normal*Crit*3, LightVec, shadowDepth);
    }

    float difFactor = max(0, dot(LightVec, Normal)) * (1-shadow);
    Color *= (0.05 + difFactor * 0.95);

    if (count == Iterations) alpha = 0;
}
```

影かどうかの判定をする前に、表面位置 surf から表面までの距離 shadowDepth を trace 機能を使って計算をしてます。
この影判定を入れてレンダリングした画像は以下のようになりました。左が Diffuse シェーダを使ったもので、右が Emission シェーダーを使ったものです。



Diffuse シェーダを使ったものは、光源の向きによってはキャンバスになる四角形の形状が見えてしまいます。

次に、下の図のようにスザンヌをレイマーチングで表示しているキャンバスの中に配置してみます。



trace 命令によってオブジェクトの「厚み」を測っているときには、Cycles での光線はスザンヌにも反応しますので、キャンバスになるポリゴンの表面から中にあるスザンヌまでの「厚み」が分かります。
そのため、スザンヌの手前までの範囲でレイマーチングが実行されて、スザンヌとの前後関係も問題なく描画されます。
ところが、下の板には影は落ちているのですが、スザンヌに影は落ちていません。これは、スザンヌの表面で Cycles が影かどうかの判定をするために光線を飛ばすとき、出発点がキャンバスになる立方体の中にいるので、内側からキャンバスのポリゴンにぶつかり、透明だと判定されて影が落ちていないのです。
同じように、カメラがオブジェクトの中にあっても、内側からポリゴンを見ることになり透明になってしまいます。
物体やカメラがレイマーチングの中にあり、内側からキャンバスを見る時には、もう少し複雑な処理をしなければいけないのですが、それに関してはもうすこし後のセクションで解説をします。

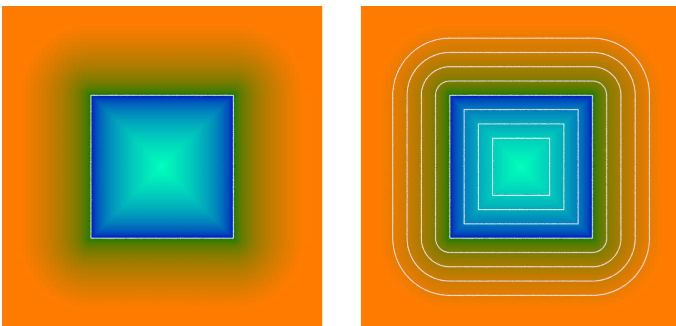
## 距離関数の編集

### 基本的な距離関数・直方体と円筒

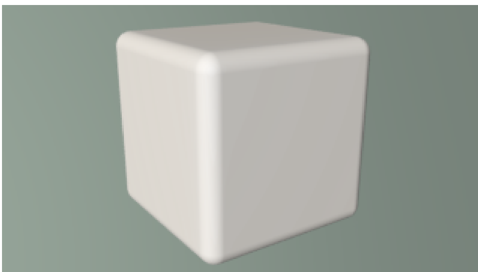
球以外の基本的な距離関数として、直方体の距離関数が簡単な数式で書けることが知られています。

```
float Box(point pos, point b){
    point d = abs(pos) - b;
    return min(max(d[0],max(d[1],d[2])),0.0) + length(max(d,0.0));
}
```

ここで、距離関数F(p)をある定数だけずらすということを考えてみます。
F(p)が例えば１になる場所、というのはF(p)=0になる物体の表面から１だけ離れた場所ですから、全体的にもとの図形から１だけ膨らんだ形になります。F(p)=2の場所はこのF(p)=1の場所からもう１だけ離れた場所にあることになります。
このようにして考えると、距離関数はある定数だけずらしてもやはり距離関数として機能することが分かります。
四角の距離関数を色で描画すると下図左のようになります。等値面（２次元なので等値線ですが）を描いてみると（下図右）、全体的に膨らんで、外側の角が丸くなった四角になるなることが分かります。



つまり、角の丸い四角の距離関数は、通常の四角の距離関数をある定数だけずらしたものと考えることができるのです。



```
float BoxRound(point pos, point b, float radius){
    point d = abs(pos) - b;
    return min(max(d[0],max(d[1],d[2])),0.0) + length(max(d,0.0)) - radius;
}
```

このほか、無限の長さを持つ円柱は、球の次元を１つ落としたものと考えてよいので、以下の式のような形になることは直感的に理解できます。

```
float CylinderX(point pos, float radius){
    return length(point(pos[1], pos[2], 0)) - radius;
}

float CylinderY(point pos, float radius){
```