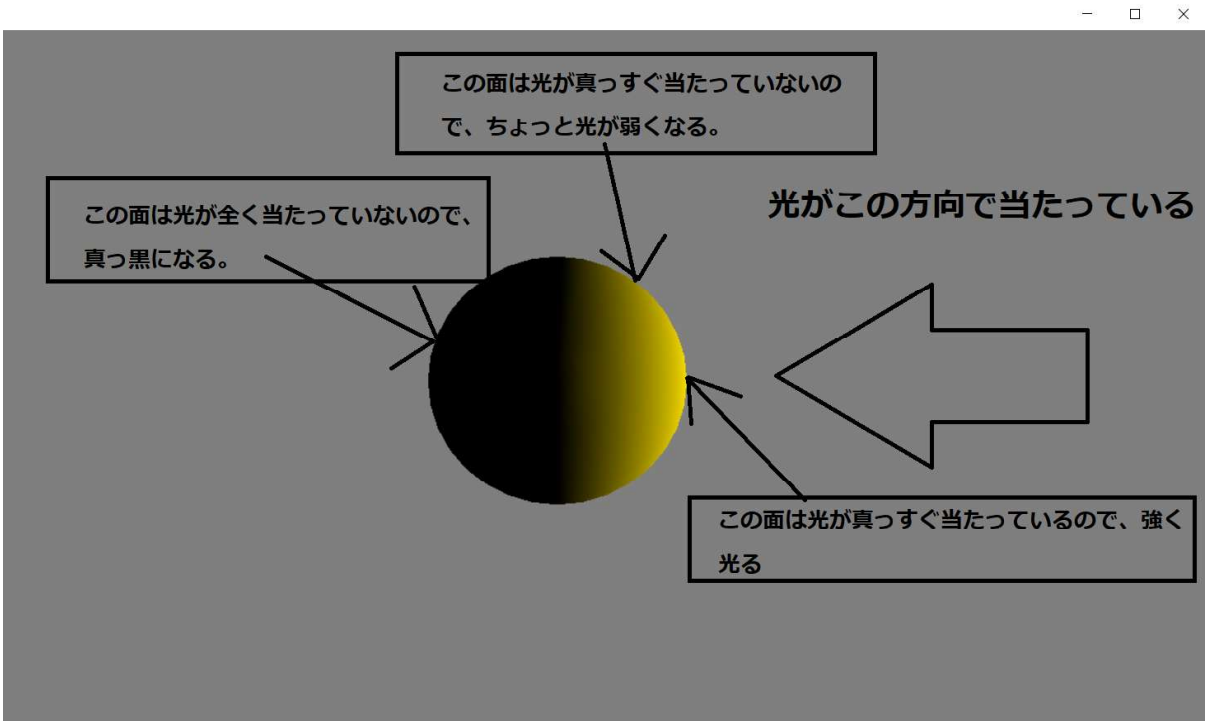
4.3 Phongの反射モデル

4.3ではハンズオンを通して、ライトによる陰影をつけるための反射モデルの一つのPhongの反射モデルについて見ていきます。今回は最もシンプルなライトのディレクションライトを使って実装していきます。Phongの反射モデルは、拡散反射光、鏡面反射光、環境光の３つ反射光を合成したものとなります。今回は、拡散反射光はランバート拡散反射モデル、鏡面反射はフォン鏡面反射モデル、環境光は大胆に近似したモデルで計算していきます。では、次の節からこの３つの反射モデルを詳しく見ていきましょう。

4.3.1 拡散反射光

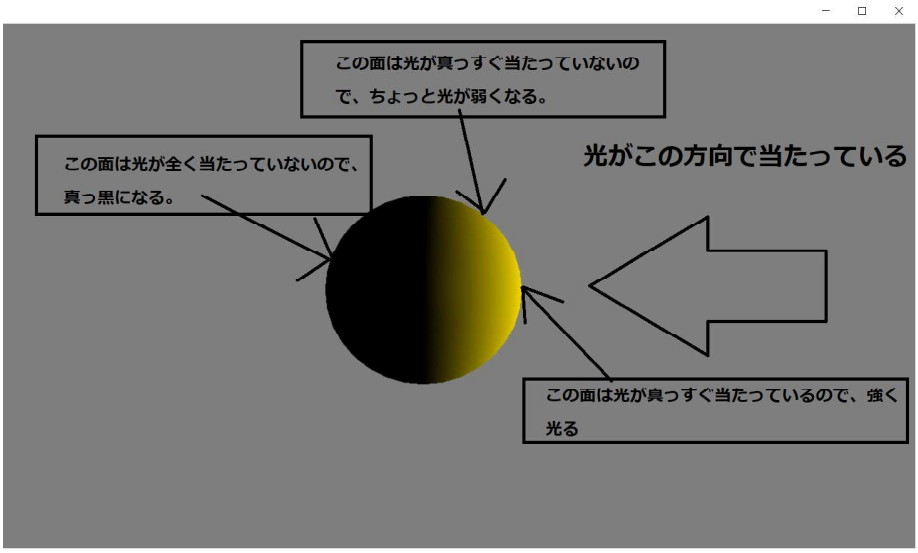
拡散反射光はランバート拡散反射モデルを使用して計算していきます。ランバート拡散反射を簡潔に説明すると「光が強く当たっているサーフェイスは明るく、光があまり当たっていないサーフェイスは暗くなっていく」という反射モデルです(図4.7)。

図4.7



直感的に分かりやすい反射モデルではないかと思います。光が真っすぐ当たっている所は強く光り、真っすぐ当たっていない所は弱く光り、全く当たっていない面は真っ暗になるという反射モデルです。この反射モデルではサーフェースに当たっている光の強さの計算に法線というデータを使用します。法線というのは、サーフェイスの向きを表す３次元のベクトルデータです(図4.8)。

図4.8

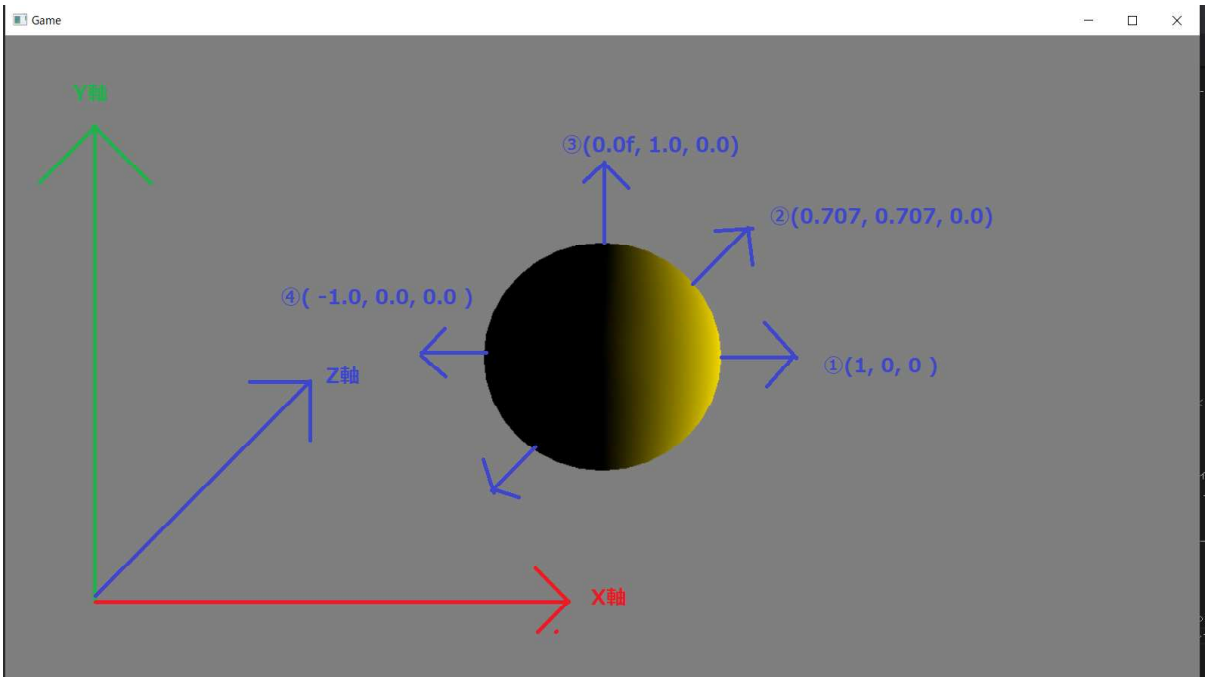


　ランバート拡散反射では、この法線とライトの方向の内積というものを計算することで、ライトの強さを計算します。

内積の性質

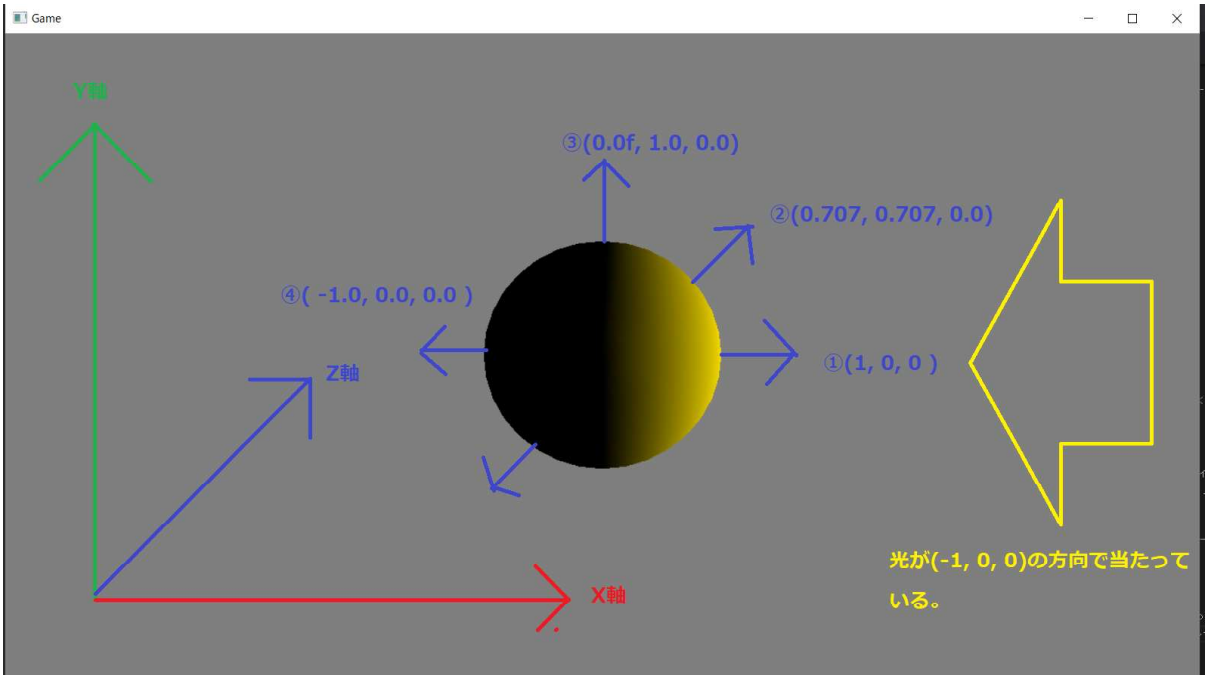
　では、少しだけ内積というものについて見ていきましょう。内積は二つのベクトルを用いて計算されるもので、ゲームで非常に使えるものです。内積の公式は次のようになります。
「二つの3次元ベクトル、v0とv1の内積はv0.x × v1.x + v0.y × v1.y + v0.z × v1.zとなる」
ここでは内積の定理の証明だとか、理屈については説明しません。次に示す内積の重要な性質だけ覚えてください。
「内積の結果は同じ向きの単位ベクトルで計算すると1になり、向きが異なっていくと数値が小さくなっていき、全くの逆向きになると-1になる」
これをまず覚えましょう。
　単位ベクトルというのは大きさ１のベクトルのことです。この性質は単位ベクトル同士で計算したときの性質なので注意してください。大抵の数学ライブラリには、ベクトルを単位ベクトルに変換するための関数が用意されています。　図4.9は各サーフェイスの法線を単位ベクトルで記したものです。

図4.9



この球体のモデルに図4.10のように右側からライトが当たっている場合を考えてみましょう。

図4.10



このとき、各サーフェイス１～４に当たるライトの強さを、ライトのベクトルと法線を利用して求めます。では、各サーフェイスの内積の結果を求めてみましょう。表4.1を見てください。

表4.1

サーフェイスの番号	法線	ライトベクトル	計算式	内積の結果
1	1, 0, 0	-1, 0, 0	1 × -1 + 0 × 0 + 0 × 0	-1
2	0.707, 0.707, 0	-1, 0, 0	0.707 × -1 + 0.707 × 0 + 0 × 0	-0.707
3	0, 1, 0	-1, 0, 0	0 × -1 + 1 × 0 + 0 × 0	0
4	-1, 0, 0	-1, 0, 0	-1 × -1 + 0 × 0 + 0 × 0	1

さて、注目してほしいのは内積の結果の列です。ではサーフェイスの番号と内積の結果だけを抽出した表4.2を見てみましょう。

表4.2

サーフェイスの番号	内積の結果
1	-1
2	-0.707
3	0
4	1

この内積の結果に-1を掛け算すると表4.3のようになります。

表4.3

サーフェイスの番号	内積の結果 × -1
1	**1**
2	**0.707**
3	**0**
4	**-1**

表4.3の内積の結果 × -1の列を見てみると、ライトが強く当たっているサーフェイス１が1になっており、ライトが弱く当たっているサーフェイス２が0.707となっていて、サーフェイス１より小さな数値になっています。ライトが当たっていない、３番と４番のサーフェイスに関しては０以下の数値になっています。この数値をライトの強さとして扱うのがランバート拡散反射です。ではライトの強さの求め方についてまとめます。

1. ライトの方向とサーフェイスの法線とで内積を計算する。
2. 1で求めた結果に-1を乗算してライトの強さを求める。
3. 2で求めたライトの強さを使ってライティングを行う。

4.3.2【ハンズオン】ランバート拡散反射を実装

では、ランバート拡散反射を実装していきましょう。Sample_04_02を立ち上げてください。

step-1 ディレクションライト用の構造体を定義する。

まずは、ディレクションライト用の構造体を定義します。main.cppを開いてリスト4.5のプログラムを入力してください。

[リスト4.5 main.cpp]

```
//step-1 ディレクションライト用の構造体を定義する。
struct DirectionLight {
    Vector3 ligDirection;  //ライトの方向。
    //HLSL側の定数バッファのfloat3型の変数は16の倍数のアドレスに配置されるため、C++側にはパディングを埋めておく。
```

```
    float pad;
    Vector3 ligColor;       //ライトのカラー。
};
```

ここでpadという変数に対して長めのコメントが記載されているのですが、これについてはstep-5で詳しく説明します。

step-2 ディレクションライトのデータを作成する。

まずはディレクションライトのデータを定義しましょう。main.cppを開いてリスト4.6のプログラムを入力してください。

[リスト4.6 main.cpp]

```
//step-2 ディレクションライトのデータを作成する。
DirectionLight directionLig;
//ライトは斜め上から当たっている。
directionLig.ligDirection.x =1.0f;
directionLig.ligDirection.y =-1.0f;
directionLig.ligDirection.z =-1.0f;
//正規化する。
directionLig.ligDirection.Normalize();
//ライトのカラーは灰色。
directionLig.ligColor.x =0.5f;
directionLig.ligColor.y =0.5f;
directionLig.ligColor.z =0.5f;
```

step-3 モデルを初期化。

今回はティーポットのモデルを表示します。リスト4.7のプログラムを入力してください。
[リスト4.7 main.cpp]

```
//step-3 モデルを初期化する。
//モデルを初期化するための情報を構築する。
ModelInitData modelInitData;
modelInitData.m_tkmFilePath ="Assets/modelData/teapot.tkm";
//使用するシェーダーファイルパスを設定する。
modelInitData.m_fxFilePath ="Assets/shader/sample.fx";
//ディレクションライトの情報を定数バッファとしてディスクリプタヒープに登録するために
//モデルの初期化情報として渡す。
modelInitData.m_expandConstantBuffer = &directionLig;
modelInitData.m_expandConstantBufferSize =sizeof(directionLig);
//初期化情報を使ってモデルを初期化する。
Model model;
model.Init(modelInitData);
```

modelInitData.m_expandConstantBufferにディレクションライトのデータのアドレスを渡していることに注目してみてください。ディレクションライトのデータはGPUで参照するため、定数バッファなどの仕組みを使って、メインメモリからグラフィックメモリに転送する必要があります。これはモデル表示でも同じです。modelInitData.m_expandConstantBufferに送りたいデータのアドレスを指定することで、model.Init関数の中で定数バッファが作成されて、ディスクリプタヒープに登録されるようになっています。興味がある方は、是非内部の処理を追いかけてみてください。

step-4 モデルをドロー。

では、これでC++側は最後です。初期化されたモデルをドローしましょう。リスト4.8のプログラムを入力してください。
[リスト4.8 main.cpp]

```
//step-4 モデルをドロー。
model.Draw(renderContext);
```

ここまで入力出来たら、一旦実行してみてください。図4.11のような陰影のないティーポットが表示されるはずです。
図4.11

step-5 ディレクションライトのデータを受け取るための定数バッファを用意する。

続いて、シェーダー側です。まずはディレクションライトのデータを受け取るための定数バッファを用意します。Assets/shader/sample.fxにリスト4.9のプログラムを入力してください。
[リスト4.9 sample.fx]

```
//step-5 ディレクションライトのデータを受け取るための定数バッファを用意する。
cbuffer DirectionLightCb : register( b1 )
{
    float3 ligDirection;    //ライトの方向。
```

```
    float3 ligColor;        //ライトのカラー。
};
```

さて、ここからは少々難しいメモリのお話をしていきます。もし、メモリの話が難しいのであれば、ここは飛ばしてもらってstep-6に進んでください。

この定数バッファですが、step-1で入力した構造体と同じ構成になっている必要があります。

```
//step-1 ディレクションライト用の構造体を定義する。
struct DirectionLight {
    Vector3 ligDirection;   //ライトの方向。
    //HLSL側の定数バッファのfloat3型の変数は16の倍数のアドレスに配置されるため、C++側にはパディングを埋めておく。
    float pad;
    Vector3 ligColor;       //ライトのカラー。
};
```

ですが、比較してみるとcpp側の構造体にはpadという無駄なデータが入っています。これはHLSL側の定数バッファはfloat3などのベクトルデータは16の倍数のアドレスに配置されるためです(図4.12)。

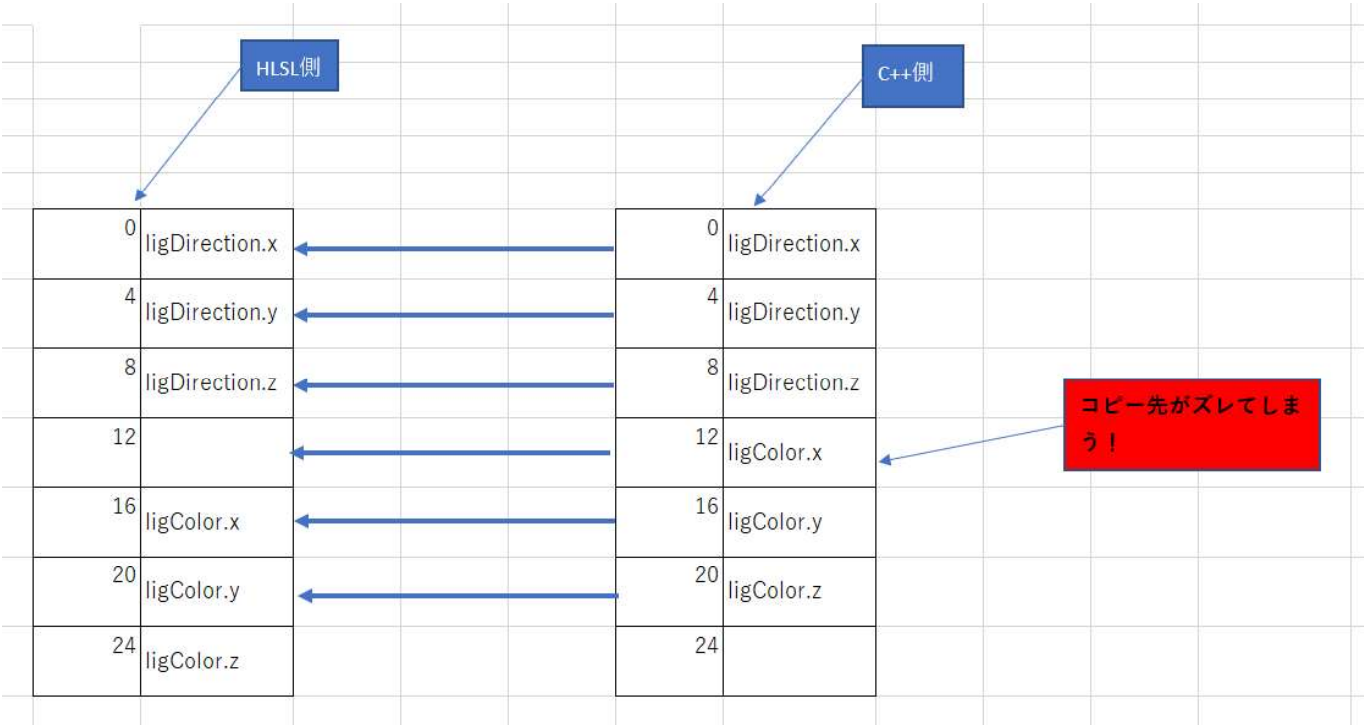
図4.12

一方、cpp側は4の倍数のアドレスに配置されるため、もしパディングがない場合は図4.13のようにデータが配置されます。

図4.13

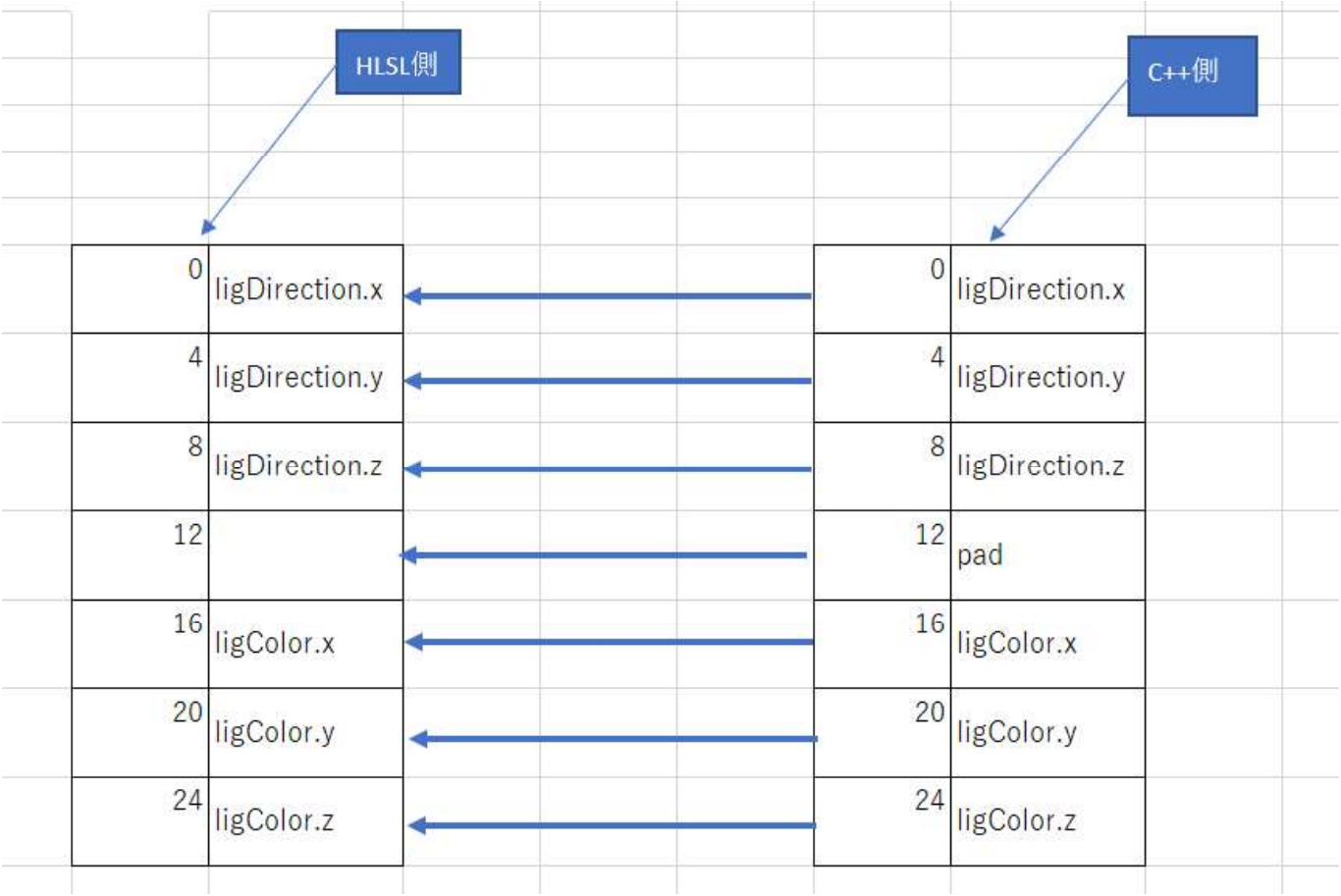
そのため、パディングなしでデータをコピーした場合、図4.14のように、データの転送先がズレてしまうことになります。

図4.14



この問題を解決するために、C++側では明示的に隙間を表すデータ(パディング)を追加しています。パディングを追加することで、図4.15のように、期待した通りのデータ転送を行うことができるようになります。

図4.15



step-6 頂点法線をピクセルシェーダーに渡す。

　続いて頂点シェーダーです。今回のモデルデータには頂点の向きを表す法線のデータが含まれています。このデータを使ってピクセルシェーダーでライティングを行うので、このデータをピクセルシェーダーに渡しましょう。リスト4.10のプログラムを入力してください。
[リスト4.10 sample.fx]

```
//step-6 頂点法線をピクセルシェーダーに渡す。
psIn.normal = mul(mWorld, vsIn.normal);//法線を回転させる。
```

　法線はモデルの回転に合わせて、回す必要があるので、モデルのワールド行列を利用して回転させています。

step-7 ピクセルの法線とライトの方向の内積を計算する。

　では次はピクセルシェーダーです。まず、ピクセルの法線とライトの方向とで内積を計算して、ライトがどれくらいあたっているか計算しましょう。ピクセルの法線がここまで説明していたサーフェイスの法線に相当します。リスト4.11のプログラムを入力してください。
[リスト4.11 sample.fx]

```
//step-7 ピクセルの法線とライトの方向の内積を計算する。
float t = dot( psIn.normal, ligDirection );
```

```
//内積の結果に-1を乗算する。
t *= -1.0f;
```

step-8 内積の結果が0以下なら0にする

ライトの強さにマイナスの値は必要ないので、内積の結果がマイナスになっているようなら、値を補正するプログラムを追加します。リスト4.12のプログラムを入力してください。
[リスト4.12 sample.fx]

```
//step-8 内積の結果が0以下なら0にする。
if( t < 0.0f){
    t = 0.0f;
}
```

step-9 ピクセルが受けているライトの光を求める。

step-8でライトの影響度を求めることができました。次は、ライトのカラー × ライトの影響度を計算して、最終的にピクセルが受けている光を求めます。リスト4.13のプログラムを入力してください。
[リスト4.13 sample.fx]

```
//step-9 ピクセルが受けている光を求める。
float3 diffuseLig = ligColor * t;
```

step-10 最終出力カラーに光を乗算する。

では、これで最後です。最終出力カラーにstep-9で求めた光を乗算しましょう。リスト4.14のプログラムを入力してください。
[リスト 4.14 sample.fx]

```
//step-10 最終出力カラーに光を乗算する。
finalColor.xyz *= diffuseLig;
```

もし、光が真っ黒(RGB＝０、０、０)であれば、最終出力カラーは黒になりますし、光が真っ白(RGB ＝ 1、１、１)であれば、最終出力カラーはテクスチャのカラーがそのまま表示されます。ここまで入力できたら、実行してみてください。図4.16のように、球体に陰影が生まれていれば完成です。

図4.16

評価テスト-7

次の評価テストを実施しなさい。
評価テストへジャンプ

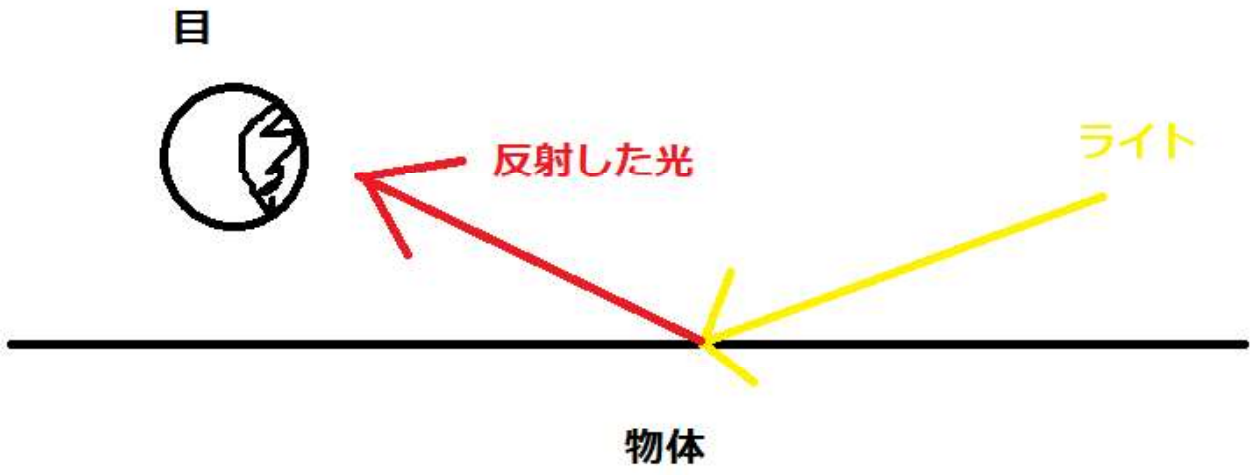
4.3.3 鏡面反射光

　鏡面反射光はフォン鏡面反射モデルを使って計算していきます。フォン鏡面反射を簡潔に説明すると「金属のような反射を表現することができる」というものです。4.3.2のランバート拡散反射にフォン鏡面反射の効果を付与すると下記のようになります(図4.17)。

図4.17

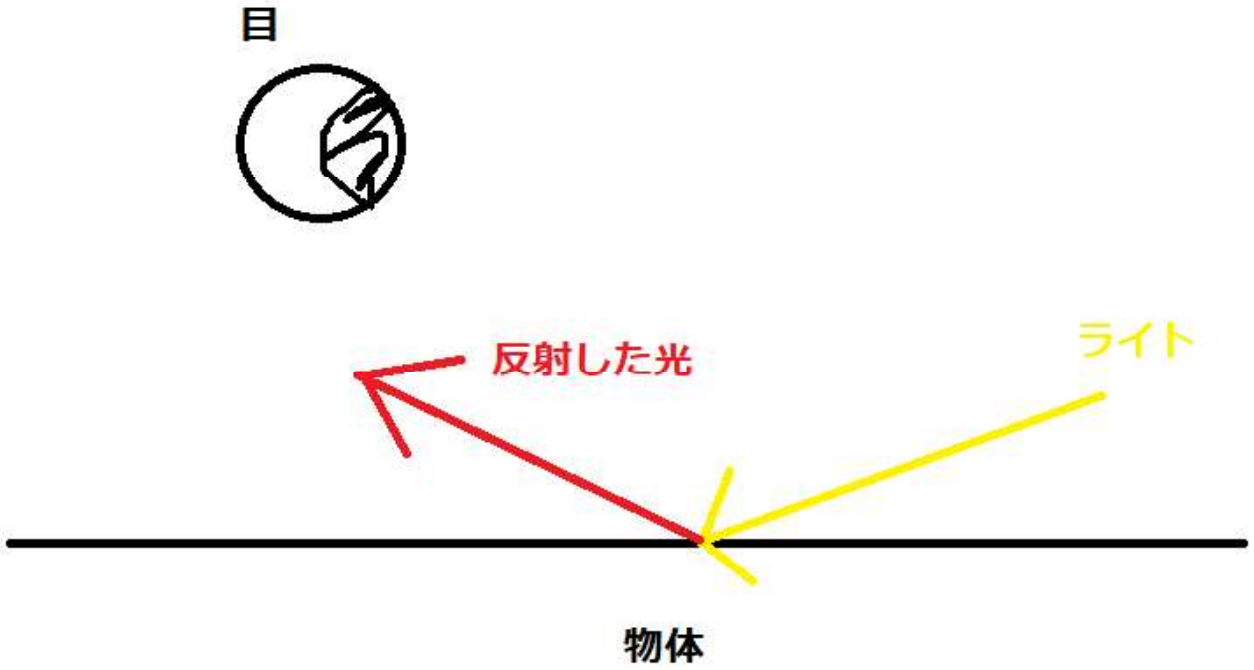
　フォン鏡面反射の強さは反射した光がどれだけ目に飛び込んでいるか？を計算すると求めることができます。図4.18のように反射した光が目に飛び込んでいる場合は鏡面反射が強くなります。

図4.18



一方、図4.19のように反射した光が目に飛び込んでこない場合は、鏡面反射は弱くなります。

図4.19



鏡面反射の強さを求めるためには、下記の処理を行う必要があります。

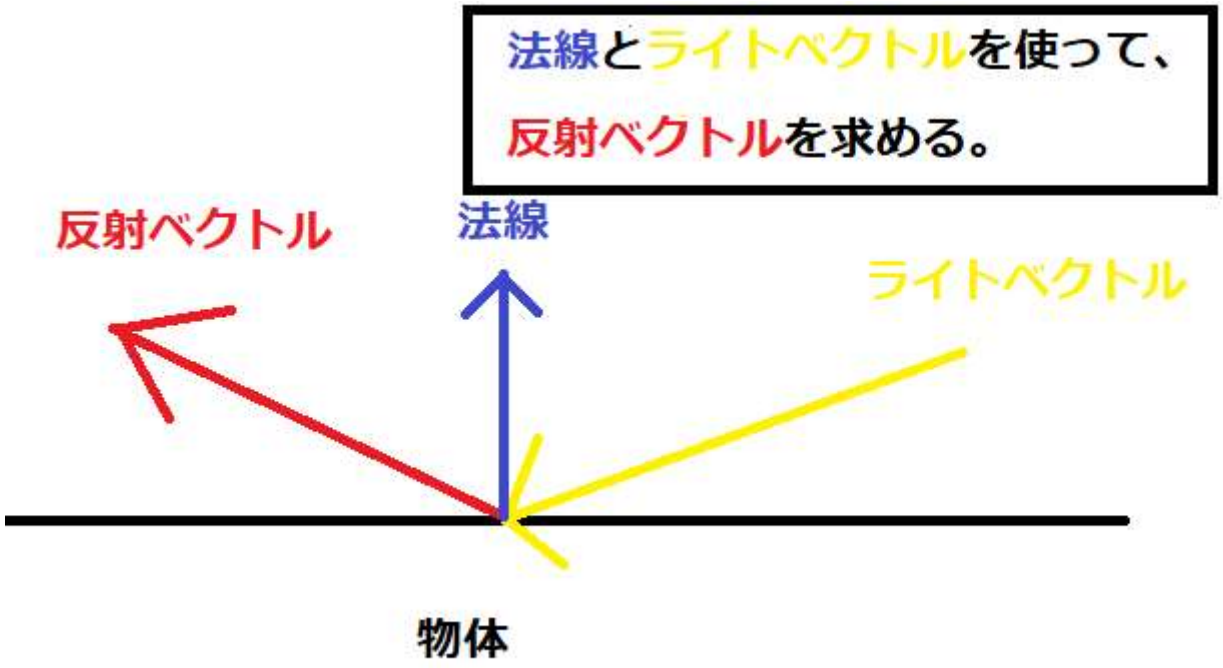
1. ライトがサーフェイスに入射して反射したベクトルを求める。
2. 光が入射したサーフェイスから視点に向かって伸びるベクトルを求める。
3. 1と2で求めたベクトルの内積を使って鏡面反射の強さ(オリジナル)を求める。
4. 3で求めた鏡面反射の強さを絞って、最終的な鏡面反射の強さを求める。

では、各ステップ詳細に見ていきましょう。

1. ライトがサーフェイスに入射して反射したベクトルを求める。

反射ベクトルを計算するためには、入射したサーフェイスの法線を使用します。

図4.20



ライトベクトルをL、法線をNとしたときに、反射ベクトルRは次の計算で求めることができます。

R=L+2×(-N・L)×N

ですが、ここではこの公式の詳しい説明は行いません。HLSLには反射ベクトルを求めるためのreflectという関数があります。今回はこの関数を利用しましょう。下記のようなコードで反射ベクトルを求めることができます。

```
float3 R = reflect(L, N);
```

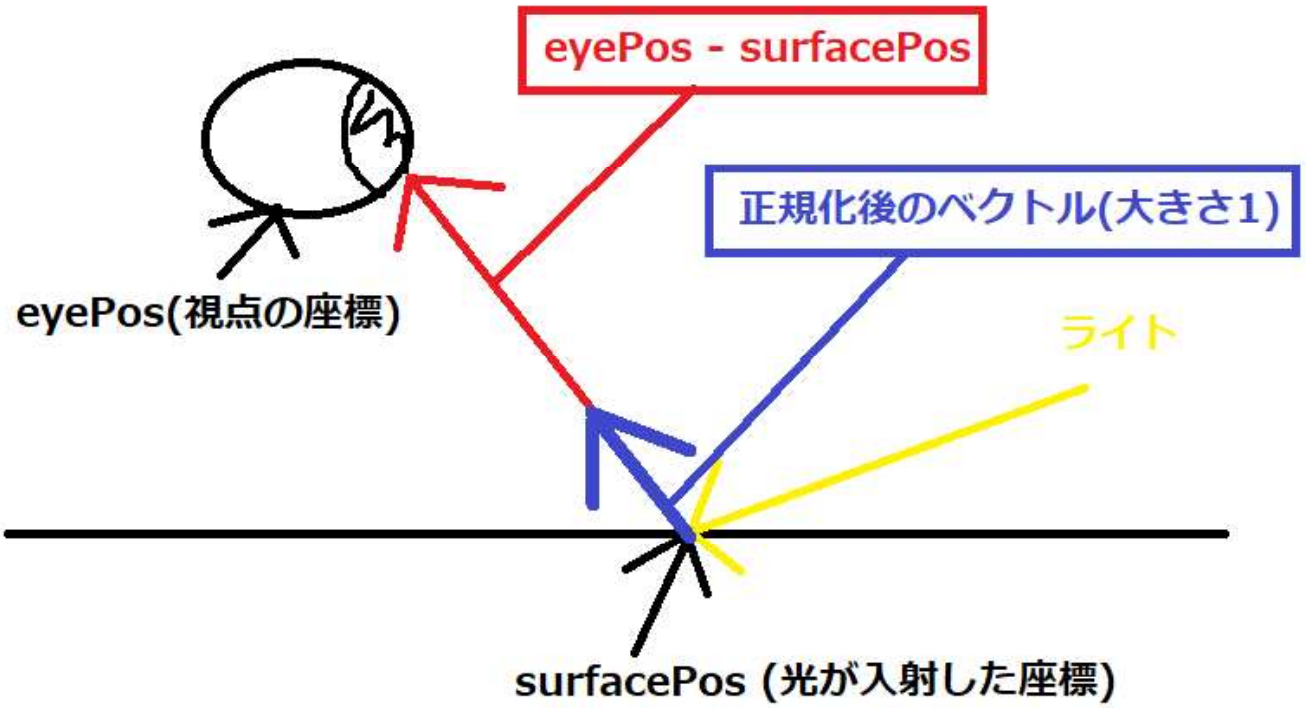
2. 光が入射したサーフェイスから視点に向かって伸びるベクトルを求める。

サーフェイスから視点に向かって伸びるベクトルは簡単なベクトルの引き算で求めることができます。サーフェイスのワールド座標をsurfacePos、視点の座標をeyePosとしたとき、視点に向かって伸びるベクトルtoEyeはeyePos - surfacePosで求めることができます。プログラムでは下記のように記述します。

```
//光が入射したサーフェイスから視点向かって伸びるベクトルを求める。
float3 toEye = eyePos - surfacePos;
//この後の処理で必要になるのは、正規化されたベクトルになるので
//正規化する。
toEye = normalize( toEye );
```

プログラムに記述されているように、後の計算で大きさ１のベクトルが必要になるので、正規化を行っています。

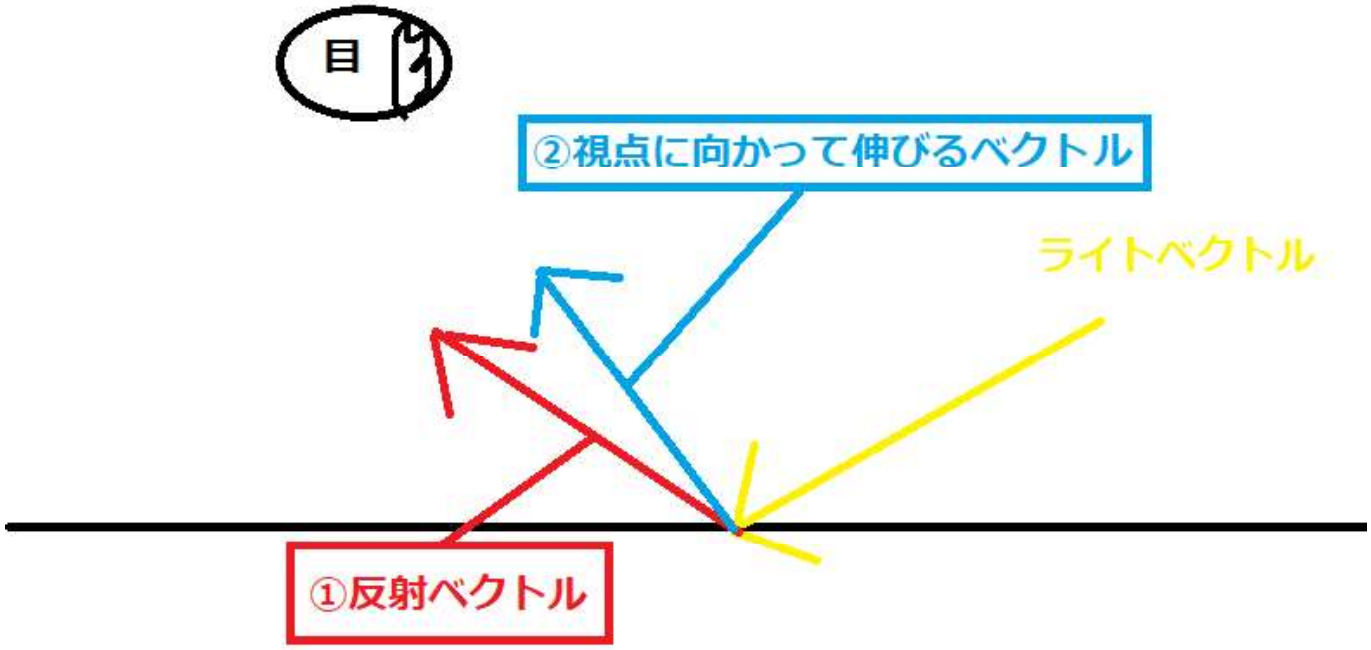
図4.21



3. 1と2で求めたベクトルの内積を使って鏡面反射の強さ(オリジナル)を求める。

　ここまでの計算で図4.22のように反射ベクトルと視点に向かって伸びるベクトルの二つのベクトルを求めることができました。

図4.22



　この二つのベクトルの向きが近ければ近いほど、反射した光が目にたくさん飛び込んでくることとなりま

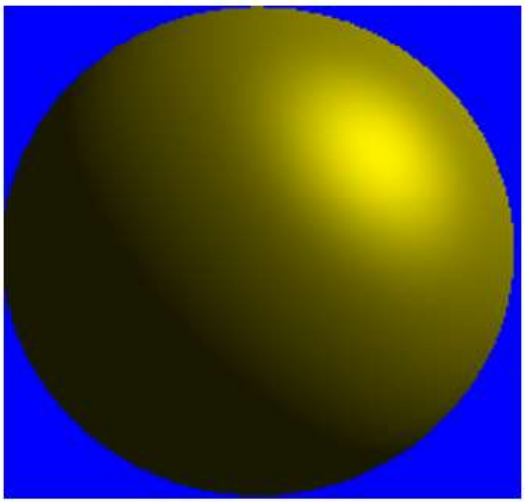
す。ここでまた内積を利用します。内積はベクトルの向きが近ければ１になり、向きが離れていくとどんどん小さな数値になっていきます。この内積の結果を鏡面反射の強さとして扱います。

4. 3で求めた鏡面反射の強さを絞って、最終的な鏡面反射の強さを求める。

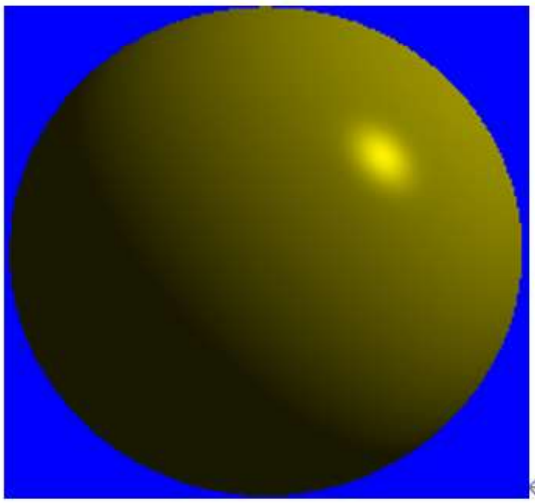
　では、最後に鏡面反射の強さの絞りについて見てみましょう。反射の強さを絞ることで、ハイライトの大きさを調整することができます(図4.23)。この後でハンズオンで実装していくプログラムでは、3で求めた鏡面反射を累乗することで絞りを行っています。

図4.23

絞り 5



絞り 50



↵

3で求めた強さに対して累乗を行うと下記の表のようになります。

表15.1

鏡面反射の強さ(オリジナル)	累乗数	最終的な鏡面反射の強さ
0.8	1	0.8
0.8	2	0.64
0.8	3	0.512
1	1	1
1	2	1
1	3	1

　3で求めた強さを累乗していくと、1はどれだけ累乗しても1のままですが、1より小さい数値はどんどん小さな数値になっていきます。つまり、累乗すればするほど目に飛び込んでくる光は弱くなってくるが、綺麗に反射して目に直接飛び込んでくる光の強さは変化しない、という効果が生まれます。

4.3.4【ハンズオン】フォン鏡面反射を実装しよう

　では、フォン鏡面反射を実装していきましょう。Sample_04_03を立ち上げてください。

step-1 構造体のメンバに視点の位置を追加する。

カメラの視点の情報をGPUに送るために、ディレクションライト構造体のメンバに視点のデータを追加しましょう。リスト4.15のプログラムをmain.cppに入力してください。
[リスト4.15 main.cpp]

```
//step-1 構造体に視点の位置を追加する。
Vector3 eyePos;          //視点の位置。
```

step-2 視点の位置を設定する。

構造体のメンバに視点の情報を追加することができたら、実際にGPUに送るデータにカメラの視点の位置を設定しましょう。main.cppにリスト4.16のプログラムを入力してください。
[リスト4.16 main.cpp]

```
//step-2 視点の位置を設定する。
directionLig.eyePos = g_camera3D->GetPosition();
```

step-3 視点のデータにアクセスするための変数を定数バッファに追加する。

続いてシェーダー側です。C++側から設定された視点のデータにアクセスするための変数を定数バッファに追加しましょう。Assets/shader/sample.fxにリスト4.17のプログラムを入力してください。
[リスト4.17 sample.fx]

```
//step-3 視点のデータにアクセスするための変数を定数バッファに追加する。
float3 eyePos;           //視点の位置。
```

step-4 反射ベクトルを求める。

では、いよいよ本題のピクセルシェーダーです。まずはサーフェイス(ピクセル)に入射した光の反射ベクトルを求めてみましょう。リスト4.18のプログラムを入力してください。
[リスト4.18 sample.fx]

```
//step-4 反射ベクトルを求める。
float3 refVec = reflect( ligDirection, psIn.normal);
```

入射してくる光の方向は定数バッファとして渡されている、ligDirection、サーフェイス(ピクセル)の法線は頂点シェーダーから渡されているpsIn.normalを使います。この二つのベクトルを使ってreflect関数を利用すると反射ベクトルを求めることができます。

step-5 光が当たったサーフェイスから視点に伸びるベクトルを求める。

続いて、光が当たったサーフェイスから視点に伸びるベクトルを計算します。リスト4.19のプログラムを入力してください。
[リスト4.19 sample.fx]

```
//step-5 光が当たったサーフェイスから視点に伸びるベクトルを求める。
float3 toEye = eyePos - psIn.worldPos;
//正規化する。
toEye = normalize( toEye );
```

視点は定数バッファとして渡されているeyePos、サーフェイスの座標は頂点シェーダーから渡されているpsIn.worldPosを使います。

step-6 鏡面反射の強さを求める。

反射ベクトルと視点に向かって伸びるベクトルを求めることができたら、反射ベクトルがどれくらい目に飛び込んでいるかを計算します。この計算には内積を使います。リスト4.20のプログラムを入力してください。
[リスト4.20 sample.fx]

```
//step-6 鏡面反射の強さを求める。
//dot関数を利用してrefVecとtoEyeの内積を求める。
t = dot( refVec, toEye );
//内積の結果はマイナスになるので、マイナスの場合は0にする。
if( t < 0.0f){
    t = 0.0f;
}
```

step-7 鏡面反射の強さを絞る。

鏡面反射の強さを求めることができたら、累乗関数(pow関数)を利用して、鏡面反射の強さを絞りましょう。リスト4.21のプログラムを入力して下さい。
[リスト4.21 sample.fx]

```
//step-7 鏡面反射の強さを絞る。
t = pow( t, 5.0f);
```

step-8 鏡面反射光を求める。

step-7で鏡面反射の強さを求めることができたので、鏡面反射光を求めましょう。リスト4.22のプログラムを入力してください。
[リスト4.22 sample.fx]

```
//step-8 鏡面反射光を求める。
float3 specularLig = ligColor * t;
```

step-9 拡散反射光と鏡面反射光を足し算して、最終的な光を求める。

　鏡面反射光が求まったので、拡散反射光と足し算して、最終的な光を求めましょう。リスト4.23のプログラムを入力してください。
[リスト4.23 sample.fx]

```
//step-9 拡散反射光と鏡面反射光を足し算して、最終的な光を求める。
float3 lig = diffuseLig + specularLig;
```

step-10 テクスチャカラーに求めた光を乗算して最終出力カラーを求める。

　では、いよいよ最後です。求めた光をテクスチャカラーに乗算して、最終カラーとして出力しましょう。リスト4.24のプログラムを入力してください。
[リスト4.24 sample.fx]

```
//step-10 テクスチャカラーに求めた光を乗算して最終出力カラーを求める。
finalColor.xyz *= lig;
```

　入力出来たらビルドして実行してみてください。うまく実装できたら図4.24のようなプログラムが実行できます。

図4.24

評価テスト-8

次の評価テストを行いなさい。
評価テストへジャンプ