# ESライブラリ＆＆
# ゲームプログラミング

## ○　バーテックスシェーダー編 - 第４回　ディフューズ　○

## ディフューズ

・ディフューズは、モデルの陰付けを行う
・ランバートの余弦則は、ディフューズ色を簡単に求めることができる
・ランバートの余弦則では、ライトの角度と法線の向きから反射の強さを求める
・最終的な色は、ライトの色、マテリアルのディフューズ反射、上記反射の強さを掛けて決定する

### 概要

ディフューズは、光を一定の方向に反射し、モデルに陰を与えます。ディフューズ色を求めるもっとも基本的な方法は、ランバートの余弦則を用いるものです。ライトの角度と頂点の法線の向き(光を反射する方向)で反射の強さを決め、法線がライトに向いているほど明るくなるというものです。反射の向きによって色が変わるので、モデルに陰を与えることができます。

### ランバートの余弦則

ランバートの余弦則は、視線を考慮せず、ライトの入射方向と頂点の法線だけで算出する方法です。この法則は、「物体の表面で反射する光の強さは、光の入射ベクトルＬと表面に対する法線ベクトルＮとが成す角度の余弦(cos)に比例する」というものです。

ランバートの余弦則を数式で表すと以下のようになります。

$$I = I_d k_d \cos\alpha$$
$$= I_d k_d (N \cdot L) \quad ※'\cdot'は２つのベクトルの内積$$

Ｉは頂点のディフーズ色、$I_d$はライトのディフューズ色、$k_d$はマテリアルのディフューズ反射です。$\cos\alpha$の$\alpha$には、ライトと法線からもとめる角度(＝成す角、なす角)が必要です。２つのベクトルから角度を求め、さらにcosを計算するか、２つのベクトルを正規化(長さを±1.0にすること)して内積を求めます。正規化された２つのベクトルの内積＝２つのベクトルのなす角のcos(ベクトルa，bの内積＝$|a||b|\cos\alpha$)なので、どちらも同じ結果になります。

上記の式を書き換えると、

頂点のディフューズ色

＝ ライトのディフューズ色 ＊ マテリアルのディフューズ反射 ＊
ライトの入射方向(ライトのベクトル)と頂点の法線方向(法線のベクトル)のなす角のコサイン

＝ ライトのディフューズ色 ＊ マテリアルのディフューズ反射 ＊
入射ベクトルと法線ベクトルの内積

となります。下線部だけをみると、アンビエントの計算式とまったく同じです。向きよる色の強弱をつけるため、cosもしくは内積を式に加えているだけです。

アンビエントでは、光と反射の向きを考慮せず常に一定の色にしていたため、陰影がつきませんでした。ディフューズは、cosもしくは内積に色値を比例させています。
「cosもしくは内積」は、法線がライトに向いているほど1.0に近くなり、垂直から反対方向になると0から-1.0になります。色値で0は黒になるので、色がつかず陰になります(負の場合は、正しい色にならないので0にします)。1.0に近ければ色が強く出るようになり、0に近くなれば陰(黒)になるという仕組みです。
よって、「cosもしくは内積」を加えただけで、色に強弱がつき、陰影を与えることができるというわけです。なお、内積はHLSLではdot関数で求めることができるので、内積の計算方法や意味がわからなくても、計算式をそのままコード化することができます。

# 課　題

ランバート余弦則を使ってディフューズ色をつけるバーテックスシェーダーを作成しましょう。

(１)以下のシェーダープログラムを"Diffuse.fx"として作成し、Meteoプロジェクトの"Shader"フォルダに保存しましょう。

－Diffuse.fx－

```
//-----------------------------------------------------------------------------
// File: Diffuse.fx
//
// The effect file for the Diffuse HLSL sample.
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
float4x4    g_WVP;

float4      g_LightDiffuse;
float3      g_LightDirection;

float4      g_ModelDiffuse;

//-----------------------------------------------------------------------------
// VertexShader Output Structure
//-----------------------------------------------------------------------------
struct VS_OUTPUT
{
    float4 Pos      : POSITION;
    float4 Color    : COLOR0;
};

//-----------------------------------------------------------------------------
// pass0 VertexShader Main Function
//-----------------------------------------------------------------------------
VS_OUTPUT VS_P0_Main(float4 Pos    : POSITION,
                     float3 Normal : NORMAL)
{
    VS_OUTPUT   Out = (VS_OUTPUT)0;

    Out.Pos = mul(Pos, g_WVP);

    // Vertex Color
    Out.Color = g_LightDiffuse * g_ModelDiffuse * max(dot(Normal, -g_LightDirection), 0);

    return Out;
}

//-----------------------------------------------------------------------------
// Techniques
//-----------------------------------------------------------------------------
technique Diffuse
{
   pass P0
   {
      VertexShader = compile vs_1_1 VS_P0_Main();
      PixelShader  = NULL;
   }
}
```

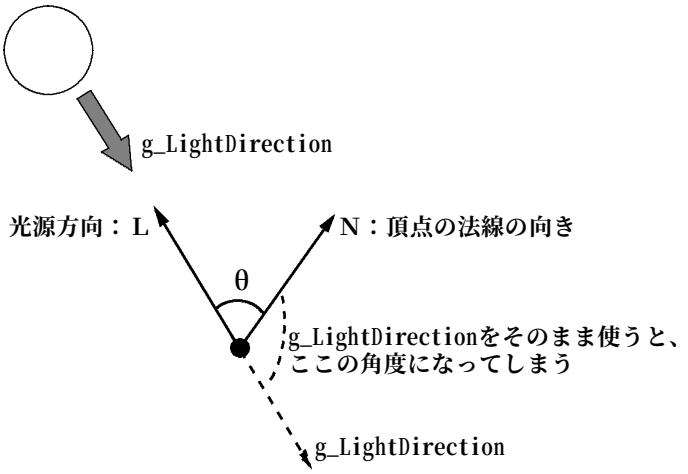
グローバル変数としてライトのディフューズ色と向き、マテリアルのディフューズ反射を宣言しています。(「float4 g_LightDiffuse」と「float3 g_LightDirection」、「float4 g_ModelDiffuse」)
また、バーテックスシェーダーの引数に"float3 Normal : NORMAL"が加わり、法線ベクトルも受け取るようにしています。

ディフューズ色は、計算式どおり

```
Out.Color = g_LightDiffuse * g_ModelDiffuse * dot(Normal, g_LightDirection);
```

と、そのままコード化されていそうですが、そうはなっていません。まず、ライトのベクトルにマイナスが付き、反対方向が指定されています(下線部)。

これは、頂点から見た法線の向きとライトへの向き(光源方向)で計算しなければならないためです。ライトの向き"g_LightDirection"は光が進む方向なので、頂点から見た光源への方向とは正反対です。そのまま内積を求めると、下図の点線との内積になり、反対方向の角度になってしまいます。




「頂点から見た光源方向」にするため、ライトの方向をマイナスにし、反対に向けています。

次にmax関数を使い、内積と0のうち大きい方が採用されるようになっています。内積は、負の値になる場合があります。色値に負を設定すると正しい色づけが行われないので、結果が負の場合はmax関数により、0を用いるようにしています。よって、

Out.Color = ライトのディフューズ色 ＊ マテリアルのディフューズ反射 ＊ 内積
※ただし、内積の結果が負になる場合は0となる(max関数)　(=cosα)

となり、ランバートの余弦則がほぼそのままコード化されたことになります。

(2)(1)で作成したエフェクトファイルを読み込むように、プログラムを変更しましょう。

(3)シェーダーのグローバル変数に値を設定し、エフェクトを使って描画しましょう。

ライトとマテリアルの設定が必要です。以下のプログラムを(2)の後に追加しましょう。

```
// エフェクトへライトのディフューズ色と方向を設定
effect->SetFloat4("g_LightDiffuse",   Color(1.0f, 1.0f, 1.0f));
effect->SetFloat3("g_LightDirection", Vector3_Down);

// エフェクトへマテリアルに対応するグローバル変数名を通知する
effect->※ここは各自考えましょう※;
```

(4)モデルのアンビエント色を設定します。以下のプログラムを適切な場所に追加しましょう。

```
// マテリアル設定
Material   mtrl;

// プレイヤー
mtrl.Diffuse = Color(0.5f, 0.5f, 0.5f);
plyMdl->SetMaterial(mtrl);

// 隕石
mtrl.Diffuse  = Color(0.4f, 0.4f, 0.4f);
meteoMdl->SetMaterial(mtrl);
```

描画などの部分は、前回と変更ありません。

(５)プログラムを実行し、動作を確認しましょう。モデルに陰影がつけば成功です。

(６)ライトの色を変え、描画色が変更されるか確認しましょう。ライトの色によって、描画色に影響が出れば成功です。

応用問題１：ディフューズだけでなく､アンビエントも考慮されるようにプログラムを変更しましょう。
応用問題２：さらに、エミッシブも考慮されるようにプログラムを変更しましょう。

※今回の計算式は、一見正しいようですが「モデルの回転・拡大縮小を考慮していない」という問題があります。そのため、モデルを回転させると陰影も回転してしまいます。
バーテックスシェーダーに送られてくる情報はモデル座標系です。頂点は座標変換しているものの、法線はそのまま(つまりモデル座標系のまま)用いています。
本来法線は、モデルの角度やスケーリングに影響を受け、その向きが変わります。法線をワールド座標系に変換するなり、回転と拡大縮小を考慮するようにしなければなりません。
ただし、厳密に計算すると計算量が増え、動作に影響が出る場合もあります。Meteoのような平行移動しかしないような場合はほとんど問題がないので、あえて今回の「動作が軽い」計算式を用いる場合もあります。また、モデルの種類や視点からの距離によって使い分けることもできます。