# 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２回　エミッシブ　○

## エミッシブ

・エミッシブは、モデル自体が発する光の色
・発光しているため、ライトがなくても色を見ることができる
・光源ではないので、ほかのモデルを照らすことはできない

### 概要

バーテックスシェーダーでは、座標変換のほかに、頂点色の計算も行います。頂点色は、ライトとマテリアルから計算しますが、マテリアルにはエミッシブという、ライトの影響を受けず、自らが発光している属性があります。色の計算にライトの色や向きを考慮する必要が一切ないので、すべてを自作しなければならないバーテックスシェーダーでも、簡単に実装することができます。

### 頂点色

頂点色は、頂点ごとにあらかじめ設定されている場合と、マテリアルから計算する場合の２とおりあります。頂点に設定済みの場合は計算が不要のため高速ですが、その分メモリを消費します。とはいっても、赤、緑、青、透明度それぞれ４バイトなので、頂点あたり４×４＝１６バイト増になるだけです。頂点が１万あったとしても、１６０ＫＢ程度なのでたいした負担にはなりません。それよりも、頂点すべてに色を設定する、という作業自体が不可能です。そのため、ほとんどの場合は、部品ごとに色をまとめてマテリアルを設定し、それをもとに計算します。Direct3Dでの色値は通常0.0～1.0の範囲で、以下の式を用いてマテリアルから頂点色が計算されています。

頂点色 ＝ エミッシブ ＋ アンビエント ＋ ディフューズ ＋ スペキュラー

各属性ごとにさまざまな計算式が考案されています。単純で高速なものから複雑な計算式になるもののよりリアルなものといったように、状況に応じた計算式を選び、用いることができるのがシェーダーの利点のひとつです。T&L固定機能では、あらかじめ用意されたものから、レンダリングステートで切り換えることしかできませんでした。

なお、実際に画面に表示される色は、ラスター化されて頂点間が補間された色値にテクスチャの色値が乗算されたものが出力されています。

### エミッシブ(発光色)

エミッシブは、モデル(頂点)そのものが自ら発光しているというものです。ライトに影響されずモデルの基本色を設定したいときに用います。

頂点の向き(法線)、ライトの色と向き、すべてを無視し、頂点に色を設定するのがエミッシブです。よって計算式は、

エミッシブ ＝ 色

となります。指定された色をそのままエミッシブとして用いれば、頂点の向きとライトを考慮しないで色付けをしたことになります(光源からの角度を考慮するようにすれば、陰をつけることができます)。

プログラムでは、シェーダーのグローバル変数としてエミッシブ色を格納する変数を宣言し、モデル描画前にプログラム側から色の渡してもらいます。ライトの影響を一切受けないため、この色をそのまま頂点色として出力するだけです(今回は、アンビエントやディフューズがないため、そのまま出力できます。ほかの属性がある場合は、加算して合成します)。

# 課　　題

座標変換とエミッシブ演算を行う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"Emissive.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。

－Emissive.fx－

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

//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
float4x4    g_WVP;
float4      g_ModelEmissive;

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

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

    Out.Pos   = mul(Pos, g_WVP);

    // Vertex Color
    Out.Color = g_ModelEmissive;

    return Out;
}

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

前回の座標変換に加え、バーテックスシェーダー出力構造体に"float4 Color : COLOR0;"が加わり、座標だけではなく色の出力も行えるようになっています(下線部)。

まず、座標変換の計算式どおり、頂点座標に変換行列を乗算し射影座標にします。前回はこの座標変換までででしたが、さらにエミッシブ色の設定も"Out.Color = g_ModelEmissive;"で行っています。エミッシブはライトの影響を受けず、今回はディフューズやアンビエントもないので、そのまま頂点色として設定できます。最後に、その結果を出力(return Out)して終了です。

(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。

シェーダーのプログラムが正しくても、このままでは色がつきません。シェーダーのグローバル変数にエミッシブ色を設定する必要があります。

ESライブラリでは、エミッシブなどのマテリアルは、エフェクトクラスにシェーダーのグローバル変数名を登録しておけば、モデル描画時に自動的に適用(登録したグローバル変数名へモデルのマテリアルの転送)が行われるようになっています。エフェクトクラスへのグローバル変数名の登録は、RetisterMaterialByName関数で行います。

以下のプログラムを(２)の後に追加しましょう。

```
// マテリアルに対応するグローバル変数名を登録する
effect->RetisterMaterialByName(NULL, NULL, NULL, "g_ModelEmissive", NULL, NULL);
```

RetisterMaterialByName関数の引数は、左からディフューズ、アンビエント、スペキュラー、エミッシブ、スペキュラーの強さ、テクスチャの順になっています。使用しない属性はNULLを指定します。

(４)モデルのエミッシブ色を設定します。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.Emissive = Color(0.7f, 0.7f, 0.7f);
plyMdl->SetMaterial(mtrl);

// 隕石
mtrl.Emissive = 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。座標変換が行われ、色がつけば成功です。

(６)プログラムの最適化を行います。

隕石のモデルは、すべて同じものを使用しています。また、マテリアルも同じです。(４)では、隕石モデルの描画のたびに、シェーダーへマテリアルの設定を行っています。これを隕石モデル描画前に１度だけ行うように変更します(ただし、モデルごとに色を変える場合は不要です)。

隕石のモデル描画部分を以下のように変更しましょう。

```
// 隕石モデル描画前に、エフェクトへマテリアルを設定する
effect->SetMaterial(meteoMdl->GetMaterial(), NULL);

// 隕石モデル描画
const UINT   pass = effect->Begin();
for(int i = 0; i < METEO_MAX; i++) {
    meteoMdl->SetPosition(meteoPos[i]);
    meteoMdl->SetScale(meteoSize[i]);
    effect->SetFloat4x4("g_WVP", meteoMdl->GetWorldMatrix() * CameraMat);

    for(UINT ps = 0; ps < pass; ps++) {
        effect->BeginPass(ps);
        meteoMdl->DrawPure();
        effect->EndPass();
    }
}
effect->End();
```

変更点は、隕石モデル描画前にSetMaterial関数でシェーダー(のグローバル変数)にマテリアルの設定を行っていることと、隕石モデルの描画にDrawPure関数を使っていることです。DrawPure関数は、エフェクトとパスのBeginやEndを行わず、またマテリアルやテクスチャの変更も行いません。純粋にモデルの描画のみを行うので、設定を使い回して描画するときなどに使用できます。