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１回　バーテックスシェーダー　○

バーテックスシェーダー

・バーテックスシェーダーは、頂点の座標変換と頂点色(とテクスチャ座標)の計算を行う
・座標変換は、T&L固定機能の「ワールド変換」「ビュー変換」「プロジェクション変換」に相当
・頂点色の計算は、ライトの向き・色と法線・マテリアルで行う

概要

バーテックスシェーダーは、モデルの頂点の数だけ呼び出されます。役割は、モデルに定義されている頂点の座標変換と、頂点色を計算するというものです。各モデルの原点中心の相対座標(モデル座標系)をカメラからのぞき込んだ座標(射影座標系)に変換し、さらに、ライトとマテリアル情報から頂点色の計算(照明演算)まで行う必要があります。

座標変換といっても、T&L固定機能と計算方法は同じです。モデル座標にワールド、ビュー、プロジェクションの順に変換行列を掛けるだけです。照明演算は、どのようなライトを使うかによって計算方法は変わりますが、最も基本的な平行光源であれば、ライトの向きと頂点の向きと色(ライトの方向と頂点の法線、ライトの色と頂点のマテリアル)から計算することができます。

シェーダーのグローバル変数

HLSLで記述するシェーダーは、もととなっているＣ言語と同じように、グローバル変数を定義することができます。このグローバル変数は、シェーダー内のどこからでも参照できるのに加え、シェーダーを呼び出すプログラム側からも読み書きすることができます。これにより、デフォルトではシェーダーに渡されない情報を渡したり、計算結果をプログラム側から読み取ったりすることができます。

頂点に関するもの(座標、法線、接線、従法線、頂点色、テクスチャ座標、頂点ブレンド比率)以外はシェーダーには渡されないため、変換行列やマテリアルといった情報(これらは頂点ではなくモデルの情報になります)は、グローバル変数で受け渡しを行います。受け渡しは簡単にできるようになっています。エフェクトはDirect3DXのID3DXEffectインタフェースで管理しますが、グローバル変数名と値を指定するだけで渡すことができるSetDataメソッドを始めとするいくつかのメソッドが用意されています。ESライブラリでも、エフェクトクラスに同名の関数に加え、さらにいくつかの関数を用意しています。これらの関数を用いれば、プログラムとシェーダー間の情報の受け渡しができます。

座標変換

バーテックスシェーダーで、まず最初に行わなくてはならないのが座標変換です。バーテックスシェーダーに送られてくる頂点座標に各変換行列を掛け、射影座標に変換します(射影後のビューポート変換は、Direct3Dによって自動で行われます)。
射影座標は、

頂点座標 × ワールド行列 × ビュー行列 × 射影行列

で求めますが、頂点座標以外は、頂点に関する情報ではない(頂点の座標変換には必要ですが、頂点そのものの情報ではない)ため、バーテックスシェーダーに渡されません。よって、プログラム側からグローバル変数でシェーダーに渡し、参照できるようにします。

課　　題

座標変換のみ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"WVP.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。

－WVP.fx－

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

//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
float4x4    g_World;
float4x4    g_View;
float4x4    g_Proj;

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

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

    Out.Pos = Pos;
    Out.Pos = mul(Out.Pos, g_World);
    Out.Pos = mul(Out.Pos, g_View );
    Out.Pos = mul(Out.Pos, g_Proj );

    return Out;
}

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

バーテックスシェーダーのみ用いています。バーテックスシェーダー"VS_P0_Main関数"では、Direct3Dより送られてくるさまざまな頂点情報のうち、"float4 Pos : POSITION"によって頂点座標だけを受け取っています。

その後、座標変換の計算式どおり、ワールド変換行列"g_World"、ビュー変換行列"g_View"、プロジェクション変換行列"g_Proj"の順に掛けて(mul関数)、その結果を出力(return Out)しています。出力された情報をもとに、ピクセル化以降の処理がDirect3Dによって行われ、画面に出力されます。

このように、シェーダーのプログラムは、基本的には計算式をそのままコードにするだけで、高度な技術が適用できるようになっています。

(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。

シェーダーのプログラムが正しくても、このままでは何も描画されません。シェーダーのグローバル変数に正しい行列を設定することによって、シェーダーが正しく動作できます。

プレイヤーのモデル描画部分を以下のように変更しましょう。

```
// プレイヤーモデル描画
plyModel->SetScale(PLY_SCALE);
plyModel->SetRotation(plyRot);
plyModel->SetPosition(plyPos);

effect->SetFloat4x4("g_World", plyMdl->GetWorldMatrix());          // ワールド変換行列
effect->SetFloat4x4("g_View" , Camera->GetViewMatrix());           // ビュー変換行列
effect->SetFloat4x4("g_Proj" , Camera->GetProjectionMatrix());     // 射影変換行列

plyMdl->Draw(effect);       // グローバル変数の設定ができたらエフェクトで描画
```

このように、シェーダーのグローバル変数に値を渡すには、グローバル変数名と値を指定するだけで行えます。なお、モデルクラスとカメラクラスのGet～Matrix関数は、指定されたワールド、ビュー、プロジェクション行列を返す関数です。

(４)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
//  effect->SetFloat4x4("g_View", Camera->GetViewMatrix());
//  effect->SetFloat4x4("g_Proj", Camera->GetProjectionMatrix());
    for(int i = 0; i < METEO_MAX; i++) {
        meteoMdl->SetPosition(meteoPos[i]);
        meteoMdl->SetScale(meteoSize[i]);

        effect->SetFloat4x4("g_World", meteoMdl->GetWorldMatrix());

        meteoMdl->Draw(effect);
    }
```

描画時にはカメラは変更しないので、プレイヤー→隕石の順に描画している場合は、プレイヤー描画時に"g_View", "g_Proj"は設定済みなので、改めて設定する必要はありません。ワールド変換行列は、各キャラクターごとに異なるので、描画の都度、シェーダーに渡す必要があります(下線部)。

(５)プログラムを実行し、動作を確認しましょう。正しく座標変換が行われれば、色はつきませんが動作自体はこれまでと変わりません。

(６)プログラムの最適化を行います。

２ヵ所ほど最適化を行える場所があります。１ヵ所目は座標変換の部分です。頂点の数だけバーテックスシェーダーが呼び出されますが、そのたびに「ワールド行列 × ビュー行列 × 射影行列」の計算が行われています。ワールド変換行列は、頂点毎ではなく、各キャラクター(描画モデル)ごとにしか変更のない部分です。よって、

モデル × ワールド行列 × ビュー行列 × 射影行列

は、まとめて

モデル × 座標変換行列

と書き換えても問題がないことになります。座標変換行列は、ワールド、ビュー、射影を掛けた行列です。こうすると、頂点の数だけ行われていた「ワールド行列 × ビュー行列 × 射影行列」の計算がなくなり、さらには頂点座標から一気に射影座標への変換ができるようになります。「座標変換行列」の計算は、モデルとカメラを直接参照できるプログラム側で行い、その結果をシェーダーに渡すようにします。

シェーダーのグローバル変数を以下のように変更しましょう。

```
//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
float4x4    g_WVP;  // World * View * Projection Matrix
```

(７)バーテックスシェーダーを以下のように変更しましょう。

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

    Out.Pos = mul(Pos, g_WVP);

    return Out;
}
```

送られてくる頂点座標に座標変換行列を掛けるだけの非常にシンプルな構成になります。GPUの負担がかなり減ります。

(８)シェーダーのグローバル変数"g_WVP"に値を設定して描画するようにプログラムを変更しましょう。

プログラム側でモデルとカメラから各変換行列を取得し、掛け合わせてシェーダーに渡します。プレイヤーのモデル描画部分を以下のように変更しましょう。

```
    // カメラ行列取得
    Matrix   CameraMat = Camera->GetViewProjectionMatrix();     // ビュー行列 × 射影行列

    // プレイヤーモデル描画
    plyMdl->SetScale(PLY_SCALE);
    plyMdl->SetRotation(plyRot);
    plyMdl->SetPosition(plyPos);

    effect->SetFloat4x4("g_WVP", plyMdl->GetWorldMatrix() * CameraMat);

    plyMdl->Draw(effect);
```

(９)(８)と同じように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
    for(int i = 0; i < METEO_MAX; i++) {
        meteoMdl->SetPosition(meteoPos[i]);
        meteoMdl->SetScale(meteoSize[i]);

        effect->SetFloat4x4("g_WVP", meteoMdl->GetWorldMatrix() * CameraMat);

        meteoMdl->Draw(effect);
    }
```

この変更により、頂点の数だけGPUで行われていた「ワールド行列 × ビュー行列 × 射影行列」の計算は、描画モデルの数だけCPUで行われるようになります。通常は、頂点数≫モデル数なので、CPUとGPUの性能にもよりますが、高速化に貢献できます。

(10)エフェクトのBeginとEnd関数の呼び出し回数を少なくしましょう。

エフェクトのBeginとEnd関数の呼び出しを少なくすると描画が速くなります。同じエフェクトを使って描画するモデルは、モデル毎にBeginとEndを呼び出すよりも、１回のBeginとEnd内ですべてのモデルの描画を行った方が高速です。

隕石のモデル描画部分を以下のように変更しましょう。

```
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++) {
        meteoMdl->DrawPass(effect, ps);
    }
}
effect->End();
```

（９）のモデルのDraw関数では、呼び出されるたびにエフェクトのBeginとEnd関数を呼び出しています。このようにすると、１回のBegin～End内ですべての隕石モデルの描画が行えます。モデル描画１回ごとにBegin～End関数を呼び出すよりも速く描画できます。