ESライブラリ＆＆
ゲームプログラミング

○　ピクセルシェーダー編 - 第３回　ラスターエフェクト　○

ラスターエフェクト

・１ラインごとにスクロールを行うことをラスタースクロールと呼ぶ
・ファミコンの「Ｆ１レース」やドラゴンクエストの「旅の扉」などが有名
・3D環境では、ハードウェア割り込みではなくテクスチャの変形などで擬似的に行っている

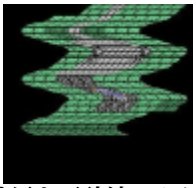
概要

ラスタースクロールは、１ラインごとに描画位置を変え、ラインごとにスクロールを実現するものです。画面下部１／３を固定してステータスを表示し、上２／３をスクロールさせるゲームや、疑似３Ｄのレースゲーム、画面効果として「うねうね」させたものなど、さまざまなゲームで応用されてきました。

▲基本画面

▷

▲ラインごとに描画位置を変えてコーナーを再現



▲描画位置を正弦波のように変更し、まどろみを表現している「旅の扉」

本来のラスタースクロールは、ディスプレイが各ラインを表示しようとするときに、表示始点をずらすことによって実現しています。表示開始位置がラインごとに左右にずれるため、ラインごとに独立してスクロールしているかのように見えます(中央や右の画像)。ハードウェアの水平帰線割り込みを利用し、各走査線ごとに表示位置を変えることで実現していましたが、近年は画像そのものを変形させることによって再現する擬似ラスタースクロールが主流です。

ピクセルシェーダーによるラスターエフェクト

ピクセルシェーダーでも、ポストエフェクトとしてラスターエフェクトが行えます。通常、ピクセルシェーダーでは、ピクセルの移動ができません。たとえば、(31，3)の位置のピクセルを(56，55)に移動して描画、ということはできません。しかし、ラスターエフェクトではピクセルの移動が必須です。

ピクセルシェーダーでは「ピクセルの移動」はできませんが、「ピクセルの内容の変更」と「ＵＶ座標の操作」は自由にできます。これを組み合わせ、本来描画すべきピクセルの色値ではなく、別の位置にあるピクセルの色値を出力すれば、ピクセルが移動しているように見えます。

ピクセルシェーダーに渡されてくる情報は、おもにそのピクセルに対応する「頂点色」と「テクスチャのＵＶ座標」です。通常は、渡されてくるＵＶ座標に対応するテクスチャのピクセル値をtex2D関数で取得し、頂点色と乗算して出力します。ここで、ＵＶ座標を増減させ、本来取得すべきピクセルとは別の場所のピクセル値を取得し、出力してみます。すると、その分だけずれた画像が描画されます。ＵＶ座標を操作すると、描画されるテクスチャ画像もその分だけ移動しているようにみえます。出力するピクセルそのものの移動はできませんが、画像の内容は自由に変更できます。本来出力すべきテクスチャ画像でないものも問題なく出力できます。これをラスターエフェクトに応用します。

まず、ディスプレイではなく「作業領域にレンダリング」します。この画像をディスプレイに転送する際の、ポストエフェクトのピクセルシェーダーでＵＶ座標を操作します。すると、ラスタースクロールと同等の効果を得ることができます。このとき、cos関数やsin関数で位置をずらすと、ドラゴンクエストの「旅の扉」のようにもできます。

課　　題

ピクセルシェーダーを使って画面にラスターエフェクトを適用してみましょう。

(１)ラスターエフェクトのひな形を作成します。次のプログラムを"Raster.fx"として作成し、プロジェクトの"Shader"フォルダに保存しましょう。

－Raster.fx－

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

//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
sampler tex0 : register(s0) = sampler_state {
    MipFilter = LINEAR;
    MinFilter = LINEAR;
    MagFilter = LINEAR;
    AddressU  = WRAP;
    AddressV  = WRAP;
};

float   amplitude = 0.0f;
float   division  = 1.0f;

//-----------------------------------------------------------------------------
// pass0 PixelShader Main Function
//-----------------------------------------------------------------------------
float4 PS_P0_Main(float2 UV : TEXCOORD0) : COLOR0
{
    return tex2D(tex0, UV);
}

//-----------------------------------------------------------------------------
// Techniques
//-----------------------------------------------------------------------------
technique Raster
{
    pass P0
    {
        VertexShader = NULL;
        PixelShader  = compile ps_2_0 PS_P0_Main();
    }
}
```

(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)レンダリング用の作業領域を生成します。まず、変数の宣言が必要です。ヘッダーファイルに以下のプログラムを追加しましょう。

```
RENDERTARGET    offscreen;
```

(４)作業領域を生成します。以下のプログラムをLoadContent関数の適切な場所に追加しましょう。

```
// オフスクリーンターゲット生成
offscreen = GraphicsDevice.CreateRenderTarget(1280, 720, SurfaceFormat_RGBA8888,
                                              DepthFormat_Unknown);
```

(５)「作業領域へのレンダリング→ピクセルシェーダーを用いて画面へ転送」を行うようにプログラムを変更します。CGameMain::Draw関数を以下のように変更しましょう。

```
void CGameMain::Draw()
{
    GraphicsDevice.Clear(Color_Black);

    // TODO: Add your drawing code here
    GraphicsDevice.BeginScene();

    GraphicsDevice.SetRenderTarget(offscreen);      // 作業領域をレンダリング出力先に設定
    GraphicsDevice.Clear(Color_Black);              // 作業領域のクリア

    // 2D描画
    SpriteBatch.Begin();
    SpriteBatch.Draw(bgSpr, Vector3(0.0f, 0.0f, 1.0f), 0.25f);
    SpriteBatch.End();

    // 3D描画
    GraphicsDevice.SetRenderState(CullMode, CullMode_None);

    // プレイヤー
    m_pPlyModel->SetScale(PLY_SCALE);
    m_pPlyModel->SetRotation(plyRot);
    m_pPlyModel->SetPosition(plyPos);
    m_pPlyModel->Draw();

    // 隕石
    for(int i = 0; i < METEO_MAX; i++) {
        m_pMeteoModel->SetPosition(meteoPos[i]);
        m_pMeteoModel->SetScale(meteoSize[i]);
        m_pMeteoModel->Draw();
    }

    // オフスクリーンターゲット→スクリーン
    GraphicsDevice.RenderTargetToBackBuffer(offscreen, effect);

    GraphicsDevice.EndScene();
}
```

(６)出力画像を全体的に画面半分左にずらして見ます。以下のプログラムをピクセルシェーダーの「return tex2D(tex0, UV);」の前に追加しましょう。

```
    UV.x += 0.5f;
```

本来のＵＶ座標より、左に0.5fずれたピクセルを取得します。ＵＶ座標は、画面端が1.0fなので、ちょうど画面半分ずれたピクセルが出力されます。これにより、画像が右に移動したように見えます。

(７)ピクセルシェーダーを変更し、右に0.25fずれたピクセルを取得するようにしましょう。

(８)画面の下に行けば行くほどより左にずれるようにしてみます。(６)または(７)の部分を以下のように変更しましょう。

```
    UV.x -= UV.y;
```

UV.yには、ＵＶのＶ座標が入ります。画面上が0.0f、画面下が1.0fなので、これをそのままUV.xに加算すれば、画面下に行けば行くほど左右のずれが大きくなるという仕組みです。

(９)左右の移動量の計算にsin関数を使って見ましょう。(８)の部分を以下のように変更しましょう。

```
UV.x += sin(UV.y * 3.141952) * 5.0f;
```

(10)「揺らぎ」の量をプログラム側で制御できるようにします。(９)の部分を以下のように変更しましょう。

```
UV.x += sin(UV.y * 3.141952 * division) * amplitude;
```

「division」は分割量、「amplitude」は揺らぎの大きさです。これらは、シェーダーのグローバル変数として宣言済みです。プログラム側で数値を徐々に増減させることにより、「旅の扉」のようにすることができます。