動きのある表現

① UVスクロール

画像を動きをつけるための基礎テクニックです。


```
float4 main(PS_INPUT PSInput) : SV_TARGET
{
    // uvスクロール
    float2 uv = PSInput.TexCoords0;
    uv.x += g_time * g_speed;
    //uv.y += g_time * g_speed;

    // 変更されたuv座標を元に色を取得する
    float4 srcCol = g_SrcTexture.Sample(g_SrcSampler, uv);
    return srcCol *= g_color;
}
```

横にスクロールするはずなのですが、このままでは上手くいかず、
画像が移動して、消えてしまいます。
UV値が 0.0 ～ 1.0 の範囲を超えているためです。
HLSLには、UV値が 0.0 ～ 1.0 の範囲外を超えたらどうするか決定する、
アドレッシングモードの設定ができます。

```
SamplerState g_SrcSampler
{
    AddressU = WRAP;
    AddressV = WRAP;
};
```

↑独自のサンプラーを定義する。(WRAPは繰り返し)
しかし、DxLibでは、このサンプラー記述が受け付けて貰えないようで、

描画前に、

```
SetTextureAddressModeUV(DX_TEXADDRESS_WRAP, DX_TEXADDRESS_WRAP);
```

を実行する必要があります。
対象モデルに対して、設定する場合は、

```
MV1SetTextureAddressMode
```

を使用します。

但し、アドレッシングモードに頼らず、
0.0 ～ 1.0 の数字を作るテクニックはどんどん高めて頂きたいので、

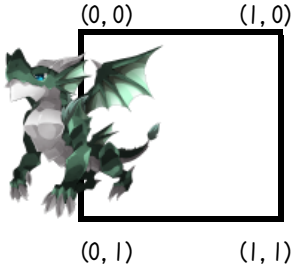
```
// uvスクロール
float2 uv = PSInput.TexCoords0;
uv.x += g_time * g_speed;
uv.x = frac(uv.x);
```

こういったやり方も知っておいて貰えたらと思います。

注意！

プラスにしているので、右に画像が移動しそうですが、
スクロールになりますので、左に画像が移動します。

左上の頂点に設定されているUV座標は、
Uが、0.0f になりますので、
画像の左上を表示します。



左上の頂点に設定されているUV座標は、
Uが、0.0f なのですが、値が加算されます。
例えば、0.5f加算すると、
0.5f は、画像の中心になりますので、
左上が中心を表示する形となります。

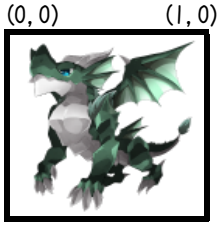
② 回転

```
// 回転ベクトル
float rotCos = cos(g_time * g_speed);
float rotSin = sin(g_time * g_speed);

// ２次元の回転行列
float2x2 mat = { rotCos, -rotSin, rotSin, rotCos };

// uvスクロール
float2 uv = PSInput.TexCoords0;
uv = mul(uv, mat);
```

時間経過で回転するベクトルを作成し、回転行列を使って、
UV値を回転させます。

但し、このままだと、左上の(0, 0)が中心になってしまいますので、

(0,0)　　(1,0)



(0,1)　　(1,1)

中心をUVの真ん中に調整してあげます。

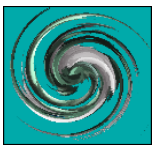
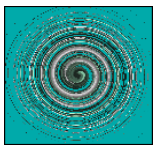
```
// uvスクロール
float2 uv = PSInput.TexCoords0 - 0.5f;
uv = mul(uv, mat) + 0.5f;
```

マイナスになることで、
Xの負の方向ベクトルが生まれます。
（Yも）

③ 渦巻

一定の渦巻量で回転


時間経過で渦巻量を増やす

```
float4 main(PS_INPUT PSInput) : SV_TARGET
{
    // uv調整
    float2 uv = PSInput.TexCoords0 - 0.5f;

    // ベクトルの長さ
    float len = length(uv);

    // 一定の渦巻量で回転をし続ける
    float pow = 3.5f;
    float rad = len * pow + fmod(g_time, 3.14159f * 2.0f);

    // 時間に応じて渦巻量をどんどん増やす
    //float rad = len * g_time * g_speed;

    // 回転ベクトル
    float rotCos = cos(rad);
    float rotSin = sin(rad);

    // ２次元の回転行列
    float2x2 mat = { rotCos, -rotSin, rotSin, rotCos };

    // uv調整
    uv = mul(uv, mat) + 0.5f;

    // 変更されたuv座標を元に色を取得する
    float4 srcCol = g_SrcTexture.Sample(g_SrcSampler, uv);
    return srcCol *= g_color;
}
```

UV値をベクトルとして扱えることを、回転で体感して頂いたかと思います。

画像の内側は回転量を少なくし、外側は回転量を大きくしていくと、
渦巻のような表現に近づきます。

画像の内側、外側は、UVをベクトルと見立てると、中心座標からの、
ベクトルの長さで判別できそうです。

```
// ベクトルの長さ
float len = length(uv);
```

lengthは、ベクトルの長さを返す関数です。

```
float rad = len * pow;
```

これで、外側は回転量が大きくなる、という式が作れました。
回転を続けたいのであれば、別に回転量を作る必要がありますので、
時間を代用して、

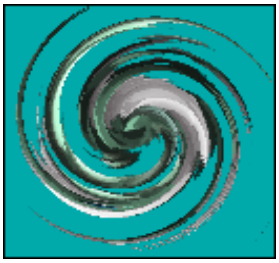
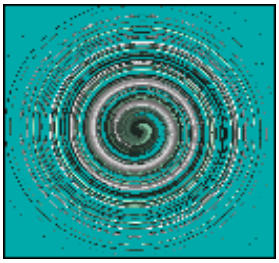
```
fmod(g_time, 3.14159f * 2.0f)
```

0 ～ 360度を足して上げます。
(増加する回転量はコストを考えると別で作った方が良いです
  あくまで代用として、fmod(時間)を使用しています)

空間にどんどん吸い込まれるような表現をするためには、
渦巻量をどんどん増加させる必要がありますので、

```
float rad = len * g_time * g_speed;
```

増加(蓄積)する値として、時間を使用して、
どんどん渦巻量を増やして上げます。

⑭ 走査線

スキャンしている雰囲気。

横の縞々模様を下地として作り、
特定範囲を明るくする。

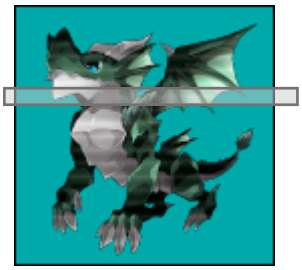
a. 下地を作る

← 下地作りの目標。

縦に一定間隔で暗くするため、
色の減算を使用する。

```
srcCol.rgb -= abs(sin(uv.y * 60.0f)) * 0.10f;
```

縦の間隔を作るため、uvのvを使用する。
ある程度のまとまりを作るため、sin関数内で乗算する。
乗算値を大きくすると、間隔が小さくなる。
最後の乗算は、減算の色量調整。
値を大きくすると、その分暗くなる。

スキャンしている感を
下地にも出すため、
縦にUVスクロールする。

```
srcCol.rgb -= abs(sin(uv.y *  60.0f + g_time * 1.0f)) * 0.10f;
```

縞々感やランダム感をもっと出すために、
色量や間隔、速度をずらした線を、もう１本増やす。
しかも、スクロール方向を逆にする。

```
srcCol.rgb -= abs(sin(uv.y * 100.0f - g_time * 2.0f)) * 0.15f;
```

例：間隔を細かく、速度を倍に、もっと暗く

---

b. 特定範囲を光らせる

下地と同様に間隔を作る

```
float area = sin(uv.y * 2.0f - g_time * 0.5f);
```

sinカーブを均一化された特定範囲とするために、
step関数を使用して、値を1 or 0 にする。

```
float isArea = step(0.996f, area * area);
```

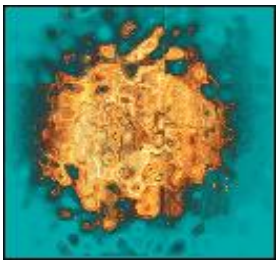
特定範囲だけ、色加算を行い明るくする。

```
srcCol.rgb += isArea * 0.3f;
```

このままだと、
特定範囲にも下地の縞模様が入ってしまうので、
消すようにする。

```
srcCol.rgb -= (1.0f - isArea) * abs(sin(uv.y *  60.0f + g_time * 1.0f)) * 0.05f;
srcCol.rgb -= (1.0f - isArea) * abs(sin(uv.y * 100.0f - g_time * 2.0f)) * 0.15f;
```

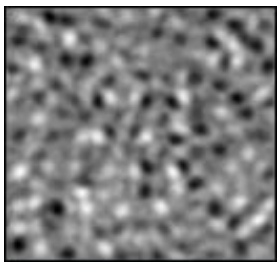
⑤ 歪み

空間を歪ませて、燃え上がっている感じを作る。

a.歪みを作る

自然な歪みを作るためのテクニックとして、
パーリンノイズが良く使われます。



シェーダコード内で生成する手法もありますが、
今回は、予め用意した画像を使用します。
このランダムなRGBを元に、
色やUV値を制御していきます。

C++側で、ノイズテクスチャをシェーダに渡します。

```
SetUseTextureToShader(1, texNoize);
```

シェーダ側でテクスチャを受け取り、uv値を元に色を取得します。
色は、RGBAの float4 が型となり、値は、それぞれ、0.0～1.0となります。
この数字を使用して、uv値を変えていくのですが、
正方向(+)よりは、正負方向(+-)の方が、ノイズ感が高まりますので、
数字を -1.0～1.0 に変換して上げます。

```
float4 noiseUv = g_NoiseTexture.Sample(g_SrcSampler, puv) * 2.0f - 1.0f;
```

ノイズの大きさ(係数)を決めて、uv値を変える。

```
uv.y += noiseUv * 0.2f;
```

b.揺らぎを作る

縦は上方向にスクロールすると、炎が動いている感じが作れそうです。

```
puv.y += frac(g_time * 0.5f);
```

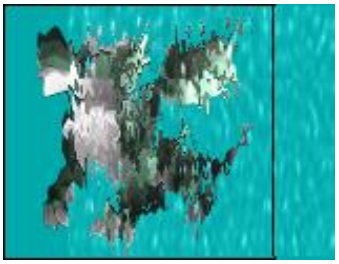
frac関数は小数部を返す関数になりますので、0.000～0.999…を返します。

横は一方向ではなく、往復する動きの方が、炎っぽい動きになるでしょう。

```
puv.x += sin(g_time * 0.3f);
```

uvが 0.0 ～ 1.0 範囲外にならないよう、調整するものアリですが、
アドレッシングモードで、繰り返し設定を行いましょう。

⑥ 滝



歪みを応用して、滝を作ってみましょう。
背景効果として使用されるケースが多いかと思いますので、画像を普段通りに描画した後、
少しずらした範囲をGetDrawScreenGraph関数で
取得して、その範囲に滝シェーダーをかけます。

パーリンノイズを使用した歪みは、色々な表現を行うことができます。
滝ができたら、風や、空、雲、煙など、チャレンジしてみましょう。

⑦ 波紋

中心から外側の方向に正弦波を作る。
正弦波を元に、UVスクロール、着色を行う。