シンプルで定番な表現

いくつか、定番パターンを実装してみましょう。

① グレースケール

３色の平均

ITU-R Rec BT.601

この画像では、余り変わらないですね。
CIE XYZ の Y や 逆ガンマ補正など、色々試してみましょう。

```
// ①３色の平均値を取る
float gray = (dstCol.r + dstCol.g + dstCol.b) / 3.0f;

// ② ITU-R Rec BT.601(内積なので、色乗算して足している)
float gray = dot(dstCol.rgb, float3(0.299, 0.587, 0.114));
```

② セピア

１回グレースケールした後に、セピア調の色を加える

```
// ①シンプルセピア
dstCol.rgb *= float3(1.07f, 0.74f, 0.43f);

// ②調整付きセピア
float3 sepia = dstCol.rgb;
sepia.r = dot(dstCol.rgb, float3(0.393f, 0.769f, 0.189f));
sepia.g = dot(dstCol.rgb, float3(0.349f, 0.686f, 0.168f));
sepia.b = dot(dstCol.rgb, float3(0.272f, 0.534f, 0.131f));
dstCol.rgb = lerp(dstCol.rbg, sepia, g_sepia_pow);
```

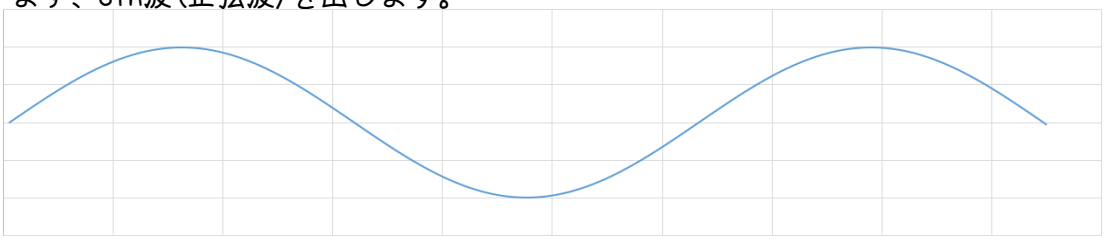
③ ノイズ

まず、HLSL内でランダムな数字を作る必要があります。
一般的な言語のように random関数があれば良いのですが、
HLSL言語にはありませんので(UnityShaderには搭載されている)
自作する必要があります。有名な疑似乱数式として、

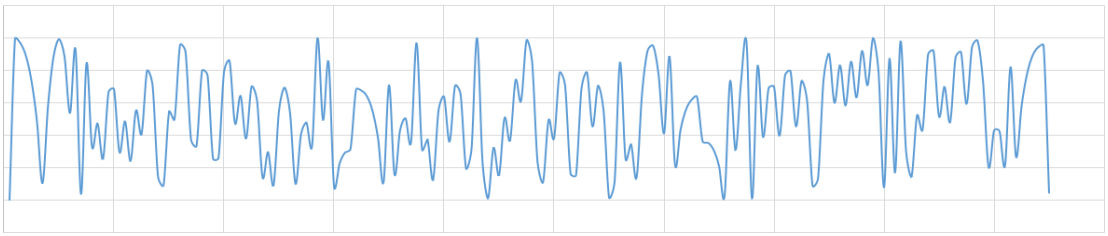
```
float noise =
    frac(sin(dot(UV座標, float2(12.9898f, 78.233f))) * 43758.5453f);
```

上記があります。

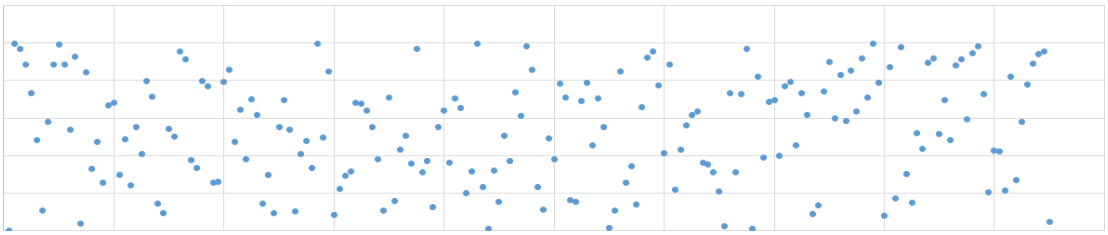
やっていることの基本と致しましては、
まず、Sin波(正弦波)を出します。



この値 -1.0 ～ 1.0 の値を大きくします。(例えば200倍で絶対値を取る)
その上で小数部のみを取り出します。(frac関数) すると、



上図のような波形になります。
波形のままだとわかりづらいので、グラフをプロット(点)にします。

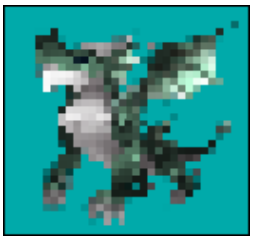
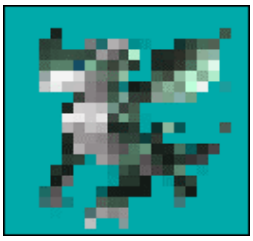


ランダムな値に見えますよね？これが、frac - sin です。
これを uv で変化させるようにする必要がありますし、
最終的には、色(0.0～1.0)にする必要もありますので、
それを成り立たせる式が冒頭の有名な疑似乱数式となります。

あとは表現したい内容に合わせてカスタマイズしていきます。

```
float4 main(PS_INPUT PSInput) : SV_TARGET
{
    float4 srcCol = g_SrcTexture.Sample(g_SrcSampler, PSInput.TexCoords0);
    //if (srcCol.a < 0.01f)
    //{
    //  discard;
    //}
    // ノイズ計算
    float noise = frac(sin(
        dot(PSInput.TexCoords0 * g_time, float2(12.9898f, 78.233f))) * 43758.5453f) - 0.5f;
    if (srcCol.a == 0.0f && noise > 0.0f) {
        // 完全透明でノイズ有りは、ノイズカラー使用
        srcCol.rgb = float3(noise, noise, noise);
        // 更に不透明にする
        srcCol.a = 1.0f;
    }
    else
    {
        // ノイズカラーを加算する
        srcCol.rgb += float3(noise, noise, noise);
    }
    return srcCol;
}
```

④ モザイク


```
// 定数バッファ：スロット0番目(b0と書く)
cbuffer cbColor : register(b0)
{
    float4 g_color;
    float  g_sizeX;          // 画像サイズX
    float  g_sizeY;          // 画像サイズY
    float  g_scale;          // モザイクスケール
}

float4 main(PS_INPUT PSInput) : SV_TARGET
{
    // uv座標を変える
    float2 uv = PSInput.TexCoords0;
    float scaleX = g_sizeX / g_scale;
    float scaleY = g_sizeY / g_scale;
    uv.x = floor(uv.x * scaleX) / scaleX;
    uv.y = floor(uv.y * scaleY) / scaleY;

    // 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;
}
```

今回は、モザイクスケールを 1 ～ 10 の値として、
モザイクの粗さを調節するものとする。(1 だとモザイク無し)

最初に、モザイクスケールが 1 だった場合の計算を解説します。
画像サイズは今回 128 になりますので、
scaleX と scaleY はそれぞれ 128 になります。
(128.0f / 1.0f)

scaleX が 128の時のuv.xの計算をシミュレートしてみると、

		scX	fX		
uv.x	scaleX	uv.x*scaleX	floor(scX)	fx / scaleX	元のuvとの差
0.00000	128	0.00000	0	0.00000	0.00000
0.05000	128	6.40000	6	0.04688	-0.00313
0.10000	128	12.80000	12	0.09375	-0.00625
0.15000	128	19.20000	19	0.14844	-0.00156
0.20000	128	25.60000	25	0.19531	-0.00469
0.25000	128	32.00000	32	0.25000	0.00000
0.30000	128	38.40000	38	0.29688	-0.00312
0.35000	128	44.80000	44	0.34375	-0.00625
0.40000	128	51.20000	51	0.39844	-0.00156
0.45000	128	57.60000	57	0.44531	-0.00469
0.50000	128	64.00000	64	0.50000	0.00000

元のuvよりも若干の差はありますが、大きな差はありません。

次は、モザイクスケールが 4 、
scaleX が 32の時のuv.xの計算をシミュレートしてみると、

		scX	fX		
uv.x	scaleX	uv.x*scaleX	floor(scX)	fx / scaleX	元のuvとの差
0.00000	32	0.00000	0	0.00000	0.00000
0.05000	32	1.60000	1	0.03125	-0.01875
0.10000	32	3.20000	3	0.09375	-0.00625
0.15000	32	4.80000	4	0.12500	-0.02500
0.20000	32	6.40000	6	0.18750	-0.01250
0.25000	32	8.00000	8	0.25000	0.00000
0.30000	32	9.60000	9	0.28125	-0.01875
0.35000	32	11.20000	11	0.34375	-0.00625
0.40000	32	12.80000	12	0.37500	-0.02500
0.45000	32	14.40000	14	0.43750	-0.01250
0.50000	32	16.00000	16	0.50000	0.00000

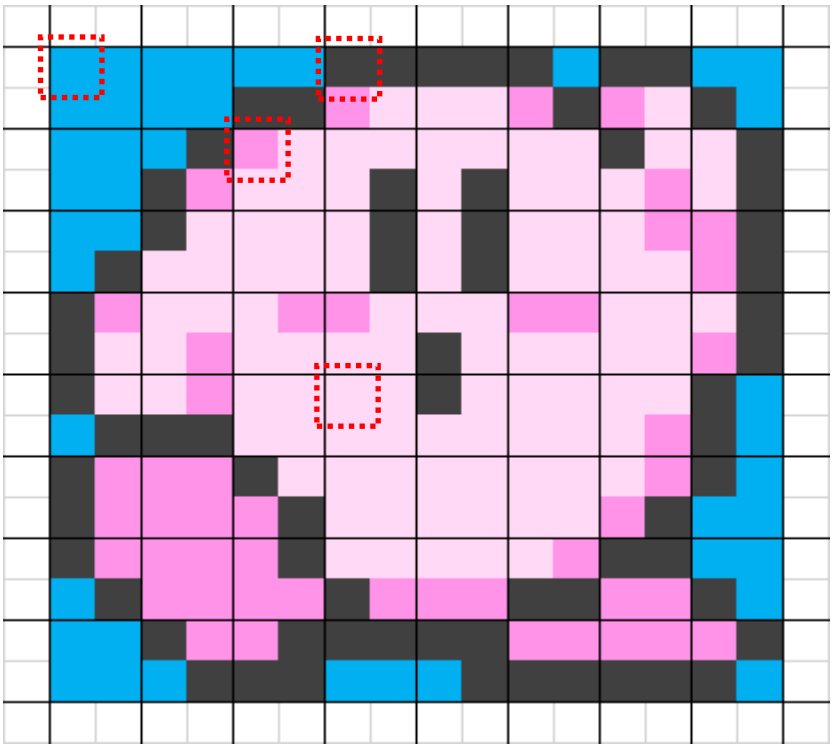
元のuv値との差が広がってきました。

scXの小数点以下の切り捨てで、捨てられる小数部の数字の大きさは、
変化がありませんが、scaleXの掛け算、割り算により、
母数からの数字比率が大きくなっているため、差が大きくなります。

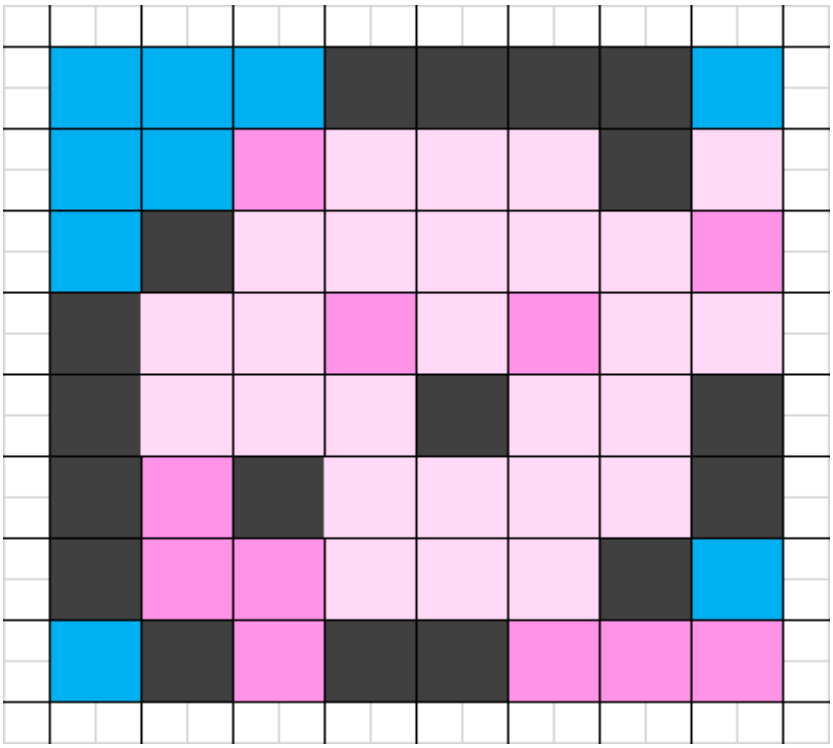
x も y もマイナスに uv座標がずれていますので、

ずれが発生するuv座標は、左上に寄っていきます。
寄っていくといっても、
全体的に寄っていくわけではなく、
モダイクスケールで設定された、
各ブロックごとに寄っていきます。

仮に 16 * 16 サイズを、モザイクスケール 2 で割った場合、



上図のような割り方になります。
各ブロックごとに、一番左上の色に寄っていく形になりますので、
その色で各ブロックを塗りつぶすと。。。



上図のように、色パターンが圧縮されます。

⑤ クレヨン

周囲の色をランダムに取得する。

```
// ランダム生成関数
float rand(float2 co) {
    // -0.50～0.49
    float a = frac(dot(co, float2(2.067390879775102f, 12.45114686629082 49f))) - 0.5f;
    float s = a * (6.182785114200511f + a * a *
        (-38.026512460676566f + a * a * 53.392573080032137f));
    // -0.99～0.99
    float t = frac(s * 43758.5453f);
    return t;
}
```