半透明化の工夫

半透明処理は、非常に重たい処理です。
特にスマートフォンは透明処理に弱く、処理負荷を軽減するために、
様々工夫がなされています。(コスト比較 不透明 く 透明 く 半透明)
色の乗算による半透明化と、負荷軽減ための工夫された手法と、
それぞれ実演していきましょう。

新しいシェーダ、アルファブレンドを作成します。

```
SetDrawBlendMode(DX_BLENDMODE_ALPHA, 128);
DrawPolygonIndexed2DToShader(mVertex, 4, mIndex, 2);
SetDrawBlendMode(DX_BLENDMODE_NOBLEND, 0);
```

半透明が効きません。(但し、輪郭線周りのジャギ×2はなくなっている)
おそらく、SetDrawBlendModeで設定された値が、
シェーダに届いてないためだと思われます。
現在の自作シェーダでも半透明かできるよう、定数バッファのα値を
変更します。
※状況によっては、アルファ専用の定数バッファを持っても
　良いかもしれません

C++側

```
COLOR_F* cbBuf = (COLOR_F*)GetBufferShaderConstantBuffer(buf);
cbBuf->r = 1.0f;
cbBuf->g = 1.0f;
cbBuf->b = 1.0f;
cbBuf->a = 0.5f;
```

Shader側

```
return srcCol * g_color;
```

乗算が効いて、半透明化されます。

ただ、先にも上げた通り、半透明は重たい処理になりますので、
できるだけ処理負荷が軽減されるよう、
完全な透明はそもそも描画しないようにします。

- SetDrawBlendModeをコメント化
- 定数バッファのα値を1.0に戻す
- 完全透明は描画しないように、Shaderに下記を追加する

```
if (srcCol.a < 0.01f)
{
    // 描画しない
    discard;
}
```

アウトラインが汚いのは、出力マージャーのブレンドの影響だと思われます。
Shaderでは、並列処理が崩れるため、if文は極力書かない方が良いのですが、
今回は、トータルコスト的に、discardの処理負荷軽減の方が、有効だと
思われるので、アルファブレンドを行うにしても、discardは行っておいた
方が良いでしょう。これをアルファテストと呼びます。

次は、某モンスター狩り系のスマホタイトルでも使用されていた、
負荷軽減された半透明化処理を紹介します。

新しいシェーダ、ディザリングを作成します。

今回は、定数バッファを多めに確保します。

```
psDitheringConstBuf = CreateShaderConstantBuffer(sizeof(float) * 8);
```

定数バッファに分割数(最大数は描画サイズ)を設定する

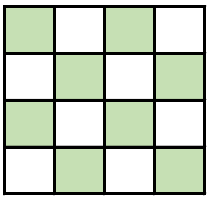
```
COLOR_F* cbBuf = (COLOR_F*)GetBufferShaderConstantBuffer(buf);
cbBuf->r = 1.0f;
cbBuf->g = 1.0f;
cbBuf->b = 1.0f;
cbBuf->a = 1.0f;
cbBuf++;
cbBuf->r = 128;
```

元がfloat4のCOLOR_Fを使用しているので、
ポインタ加算して、5番目のfloatを、
cbBuf->r として使用していますが、
同一の定数バッファで、意味が異なる場合は、
FLOAT4で統一した方が良いかもしれません。

Shader側で以下の文を追加する

```
cbuffer cbColor : register(b0)
{
    float4 g_color;
    float g_division;
}


float4 main(PS_INPUT PSInput) : SV_TARGET
{
    float x = floor(PSInput.TexCoords0.x * g_division);
    float y = floor(PSInput.TexCoords0.y * g_division);
    float s = fmod(x + y, 2.0);

    if (int(s) == 0)
    {
        discard;
    }
```

uv座標を128分割して、交互に描画しないようにしています。



分割というと、mod or fmod or % を使用したくなるのですが、
uv座標が0.0～1.0の間になりますので、単純にはいきません。

そこで、一旦、掛け算(今回は128)をしてあげると、
0.0～128.0の値になり、切り捨て or 四捨五入 を行うと、
0～128 になりますので、modが使えるようになります。

小数点除算の例。今までのイメージとちょっと違いますよね。

UV座標		除算		結果
0.012217	%	128	=	0.012217
0.034225	%	128	=	0.034225
0.812981	%	0.2	=	0.012981

UV座標		小数点除算		結果
0.012217	%	0.2	=	0.012217
0.034225	%	0.2	=	0.034225
0.098113	%	0.2	=	0.098113
0.218931	%	0.2	=	0.018931
0.345711	%	0.2	=	0.145711
0.512943	%	0.2	=	0.112943
0.812981	%	0.2	=	0.012981

floor(UV座標*128)		除算		結果
1	%	2	=	1
4	%	2	=	0
12	%	2	=	0
28	%	2	=	0
44	%	2	=	0
65	%	2	=	1
104	%	2	=	0

このロジックでは、分割数の最大が、スクリーン上での
画像サイズになりますので、最大透明度が0.5となります。

0.5までだと表現も狭くなりますので、もっと汎用的に
このドット抜きを行うアルゴリズムとして、ベイヤー配列というものが
ありますので、そちらを紹介したいと思います。

新しいシェーダ、ベイヤーディザを作成します。

ベイヤー配列とは？

元々は映像機用のカラーフィルタで使用されていました。

0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

このように0, 1, 2, 3, 4・・と、
順番に数字を追っていくと、たすき掛けのような
位置関係になっているのがわかるかと思います。

このたすき掛け配列を半透明度レベルとして、色を抜く(描画しない)と。。

ベイヤー1

	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

↓

ベイヤー6

	8		10
12		14	6
	11		9
15	7	13	

↓

ベイヤー11

12		14	
	11		
15		13	

↓

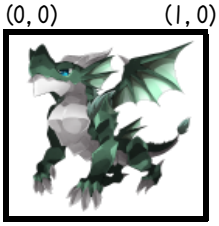
このようになります。
簡易ディザのように半透明チックになりそうです。

```
// 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_level;             // ベイヤーレベル
    float  g_img_sizeX;         // 画像サイズX
    float  g_img_sizeY;         // 画像サイズY
}

// ベイヤー配列
static const int BayerPattern[16] = {
  0,  8,  2, 10,
 12,  4, 14,  6,
  3, 11,  1,  9,
 15,  7, 13,  5
};

float4 main(PS_INPUT PSInput) : SV_TARGET
{
    // UV基準
    int x = round(PSInput.TexCoords0.x * g_img_sizeX);          ・・・①
    int y = round(PSInput.TexCoords0.y * g_img_sizeY);          ・・・①
    int dither = (x % 4) + (y % 4 * 4);                         ・・・②
    if (int(g_level) > BayerPattern[dither])                    ・・・③
    {
        discard;
    }

    // UV座標とテクスチャを参照して、最適な色を取得する
    float4 srcCol = g_SrcTexture.Sample(g_SrcSampler, PSInput.TexCoords0);
    if (srcCol.a < 0.01f)
    {
        // 描画しない
        discard;
    }

    return srcCol * g_color;
}
```

① UV座標から、スクリーン座標上の画像座標に変換
（ 2Dバージョン ）

TexCoords(UV座標、テクスチャ座標)は、以前からも説明させて頂いた通り、
0.0～1.0の座標になります。

(0,0)　　(1,0)



(0,1)　　(1,1)

ベイヤー配列で、ドット抜きをしたい関係上、
ベイヤー配列の4x4に合わせて画像を分割する必要があります。
分割する際は、modを使用したいですが、小数点以下の数字なので使いづらい。
先ほどと同じですね。

今回は、画像サイズ(128*128)情報を用意して、それぞれuv値にかけて、
0.0～1.0 を 0～128 の数字に変えて、分割しやすいようにします。

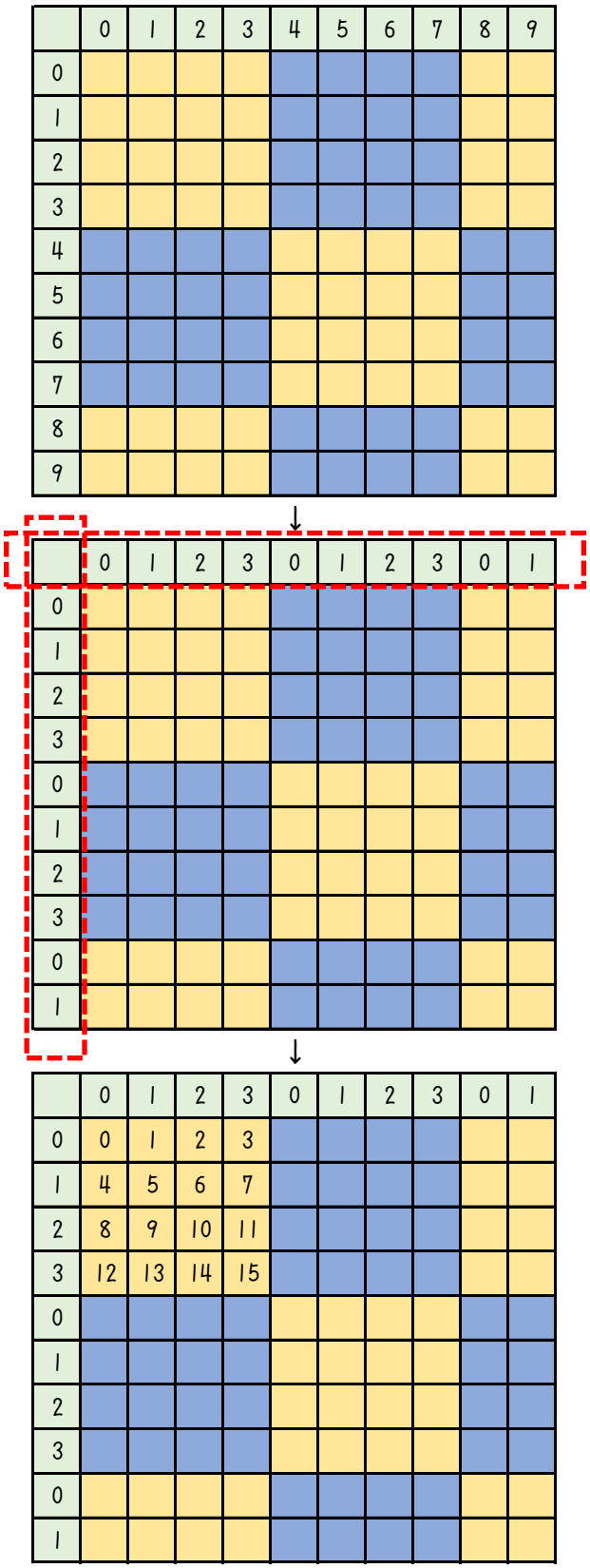
	0.0	0.1	0.2	…	1.0
0.0					
0.1					
0.2					
…					
1.0					

→
→

	0	13	26	…	128
0					
13					
26					
…					
128					

小数点の数字を、除算しやすいように整数化する儀式だと
思って頂いて大丈夫かと思います。

② u と v をそれぞれ 4 で除算して、4×4のベイヤーグループを作る



4×4のベイヤーグループ。
これを作るには、まず、
整数化されたuvをそれぞれ、
4で除算する。(x % 4)

そして、u と v の値を調整して、
ベイヤー配列の添え字0～15の値に
なるようにする。

u(横)は、0～3 そのまま。
v(縦)は、0～3 に 4 をかけて、
0～12に値にする。

0～3 と 0～12の値を足すと、
0～15 の添え字となる。
(左図)

③ ドットを抜くか抜かないか

定数バッファの g_level がゼロの場合は、
ゼロ未満の値は、ベイヤー配列には存在しないので、フル描画。

定数バッファの g_level が１の場合は、
１未満の値は、一番左上のゼロしかないので、
各ベイヤーグループ(4x4)の左上がdiscard(描画しない)となる。

0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

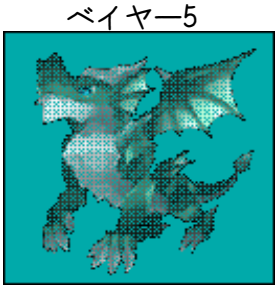
定数バッファの g_level が５の場合は、
５未満の値が抜けるので、以下のように、結構抜けます。

0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

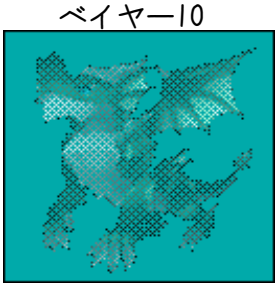
ベイヤー0

ベイヤー3

ベイヤー5

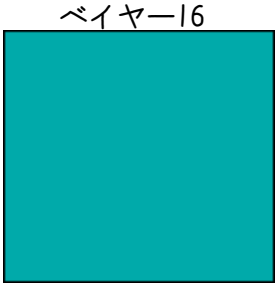


ベイヤー10



ベイヤー14

ベイヤー16



ディザリングを採用する必要はありませんが、
ここで使用した数字の作り方は、今後も使っていきますので、
数字のイメージはできるようにしておいてください。