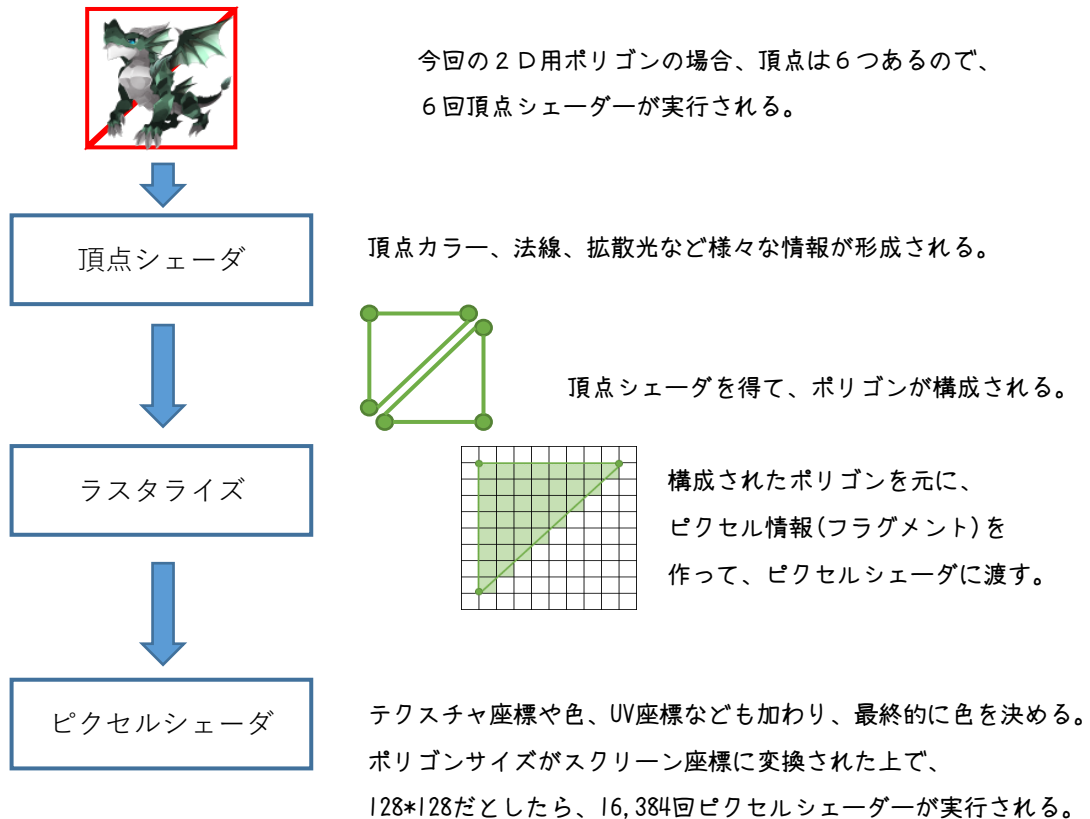


# ピクセルシェーダの基礎

レンダリングパイプラインをもう1度確認しましょう。

超簡易版



定数バッファを使えば、C++側からの情報をピクセルシェーダで使用できることは、既実践していますが、頂点シェーダの情報(3D)の情報も、ピクセルシェーダーで使用することができます。

これは、頂点シェーダ内に記載されている出力項目に依存します。

現在、ピクセルシェーダは自作していますが、頂点シェーダは特に指定していませんので、DxLib標準のものを使用している形となります。

DrawPolygonIndexed2DToShaderで使用されるシェーダは、おそらく、Base\_2D\_VS.hlslか、Base\_3D\_Simple\_VS.hlslだと思います。

それぞれの頂点シェーダ出力を確認すると、以下のように同じ出力内容でした。

## Base\_2D\_VS.hlsl

```
// 頂点シェーダーの出力
struct VS_OUTPUT
{
    float4 Position      : SV_POSITION ;
    float4 Diffuse        : COLOR0 ;
    float2 TexCoords0     : TEXCOORD0 ;
    float2 TexCoords1     : TEXCOORD1 ;
};
```

## Base\_3D\_Simple\_VS.hlsl

```
// 頂点シェーダーの出力
struct VS_OUTPUT
{
    float4 Position      : SV_POSITION ;           // 座標 ( プロジェクション空間 )
    float4 Diffuse        : COLOR0 ;               // ディフューズカラー
    float2 TexCoords0     : TEXCOORD0 ;           // テクスチャ座標 0
    float2 TexCoords1     : TEXCOORD1 ;           // テクスチャ座標 1
};
```

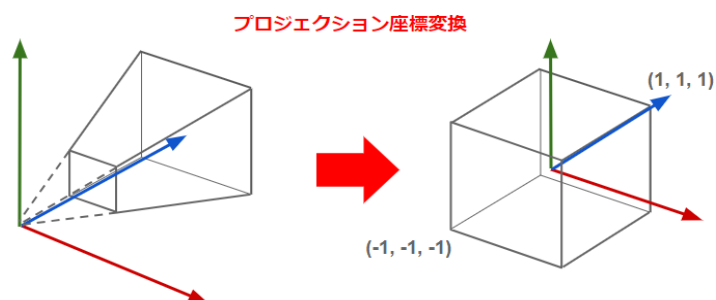
これらをピクセルシェーダで使用していきたいと思います。

その前に各項目の解説ですが、

float4 Position : SV\_POSITION ; ⇒ プロジェクション座標

通常は、プロジェクション座標が格納される。

プロジェクション空間とは、カメラに写った領域(左図)を、xyzがそれぞれ -1 から 1 に収まるような空間に変換された空間のこと。



float4 Diffuse : COLOR0; ⇒ ディフューズカラー

上記の頂点シェーダでは、3Dモデルソフトで設定されたディフューズカラーをそのまま使用していますが、ライトの位置・色等で再計算されたりします。

float2 TexCoords0 : TEXCOORD0; ⇒ テクスチャ座標 (UV座標)



テクスチャを、0.0～1.0の空間とした場合の座標。  
2Dであれば、四方の位置関係が把握できる。  
また、サンプラーにかければ、  
対象テクスチャの色を取得することができる。

float2 TexCoords1 : TEXCOORD1; ⇒ サブテクスチャ座標 (UV座標)

すみません、使ったことないです。。。

それではシェーダーに書いてみましょう。

// ピクセルシェーダーの入力

struct PS\_INPUT

{

float4 Position : SV\_POSITION; // 座標 ( プロジェクション空間 )

float4 Diffuse : COLOR0; // ディフューズカラー

float2 TexCoords0 : TEXCOORD0; // テクスチャ座標

};

// 定数バッファ: スロット0番目 (b0と書く)

cbuffer cbParam : register(b0)

{

float4 g\_color;

}

float4 main(PS\_INPUT PSInput) : SV\_TARGET

{

// UV座標を取得する

float2 uv = PSInput.TexCoords0;

return g\_color;

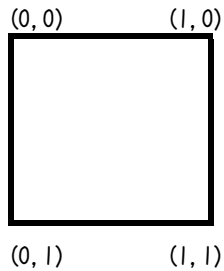
}

順番が非常に大切で、  
頂点シェーダーの出力に記載されている  
順番と合わせる必要があります。  
当然、型もです。(Byteがずれるため)  
TEXCOORD1は今回使用しませんので、  
省略します。

構造体引数として受け取る。




## UV座標を使って、色を変化させてみましょう

UV座標

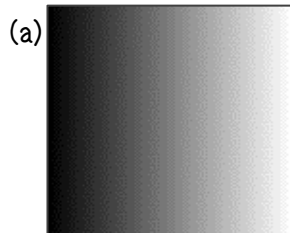


色

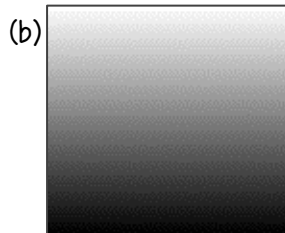
float4(R, G, B, A)

	float4(0.0f, 0.0f, 0.0f, 1.0f)
	float4(0.5f, 0.5f, 0.5f, 1.0f)
	float4(1.0f, 1.0f, 1.0f, 1.0f)

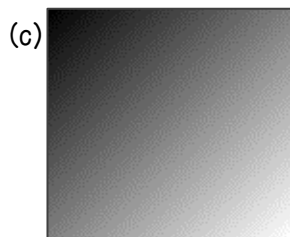
演習① UV座標を使用して、以下の図を完成させましょう



左が黒、右が白



上が白、下が黒



左上が黒、右下が白

色情報は、RGBA、それぞれ0.0f～1.0fの間に値を収める必要がある。

0.0f～1.0fの数字を制御していくことがとても大切になります。

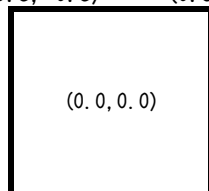
( 1.0fを超えると1.0f扱い。0未満は0。 )

UV座標を素直に使ってきましたが、ここからがシェーダの面白いところです。

```
float x = abs(uv.x - 0.5f);  
float y = abs(uv.y - 0.5f);  
return float4(x + y, x + y, x + y, 1.0f);
```

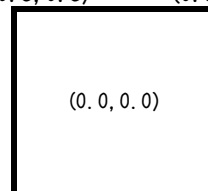
UV値は、0.0～1.0になるので、-0.5してあげると、-0.5～0.5の数字になります。

(-0.5, -0.5) (0.5, -0.5)



絶対値を取ると⇒

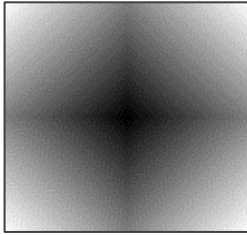
(0.5, 0.5) (0.5, 0.5)



(-0.5, 0.5) (0.5, 0.5)

(0.5, 0.5) (0.5, 0.5)

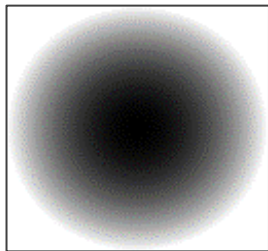
xとyを足すと、0.0~1.0の値になりますので、



中心は黒、外側は白  
こんな色になります。

これを応用して、円を描画してみてください。

演習② UV座標を使用して、以下の図を完成させましょう



上手く円を描画できましたでしょうか？

この円に対して、指定色を反映させていきたいと思います。

```
// ピクセルシェーダー用の定数バッファのアドレスを取得
COLOR_F* cbBuf = (COLOR_F*)GetBufferShaderConstantBuffer(psCircleColorConstBuf);
cbBuf->r = 0.94f;
cbBuf->g = 0.90f;
cbBuf->b = 0.55f;
cbBuf->a = 1.0f;
```

※パステルイエローに近い色

円形と指定色との乗算(黒っぽくなる)

```
g_color * float4(r, g, b, 1.0f)
```

円形と指定色との加算(白っぽくなる)

```
g_color + float4(r, g, b, 1.0f)
```

乗算



加算

