テクスチャを使ったシェーダー

キャラクターなどを描画する場合、
既に書き出された画像ファイルを使用していくことが多いかと思いますので、
シェーダーで、画像情報を扱えるようにしていきます。

C++側

```
// シェーダーにテクスチャを転送
SetUseTextureToShader(0, texDragon);
```

Shader側

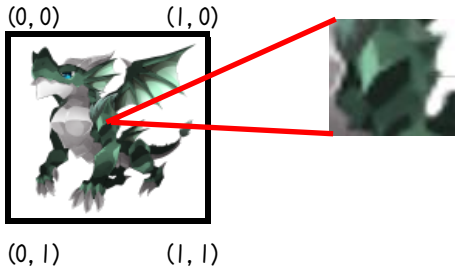
```
// ピクセルシェーダーの入力
struct PS_INPUT
{
    float4 Position        : SV_POSITION;        // 座標( プロジェクション空間 )
    float4 Diffuse         : COLOR0;             // ディフューズカラー
    float2 TexCoords0      : TEXCOORD0;          // テクスチャ座標
};

// 定数バッファ：スロット0番目(b0と書く)
cbuffer cbParam : register(b0)
{
    float4 g_color;
}

// 描画するテクスチャ
Texture2D    g_SrcTexture       : register(t0);
// サンプラー：適切な色を決める処理
SamplerState g_SrcSampler       : register(s0);

float4 main(PS_INPUT PSInput) : SV_TARGET
{
    // 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);

    return srcCol;
}
```

わかりにくいですが、定数バッファと同じように、
SetUseTextureToShader関数の第一引数の値と、
シェーダー側のレジスタ番号 register(t●) が連動しています。

```
SetUseTextureToShader(0, texDragon);
Texture2D    g_SrcTexture       : register(t0);
```

g_SrcTextureに画像情報が入っているのですが、
UV座標の値によって、画像の中のどの位置から取得するか決めないといけません。

UV座標



UV(0.5,0.5)は、ど真ん中になりますので、
■←ほぼ黒っぽい色が取得できます。

UVとピクセル座標に対応した最適な色(※)を取得する機能をサンプラーといいます。

サンプラーを定義して、

```
SamplerState g_SrcSampler       : register(s0);
```

画像情報とUV座標をサンプラーに渡して、色を取得します

```
float4 srcCol = g_SrcTexture.Sample(g_SrcSampler, PSInput.TexCoords0);
```

※ 画像に対して、フィルタをかけ、最適な色を計算します。
ポリゴンサイズによって、拡大・縮小される場合などの色の算出方法。
フィルタの種類はいくつかあり、詳細は下記を参照。

フィルタの種類
https://docs.microsoft.com/en-us/windows/win32/api/d3d11/ne-d3d11-d3d11_filter

フィルタの特徴
https://docs.microsoft.com/ja-jp/visualstudio/debugger/graphics/point-bilinear-trilinear-and-anisotropic-texture-filtering-variants?view=vs-2022

フィルタの違い

フィルタを変更する場合

```
SamplerState TextureSampler
{
    Filter = D3D11_FILTER_MAXIMUM_MIN_MAG_MIP_LINEAR;
};
```

→サンプラーを独自に定義する

```
float4 srcCol = g_SrcTexture.Sample(TextureSampler, PSInput.TexCoords0);
```

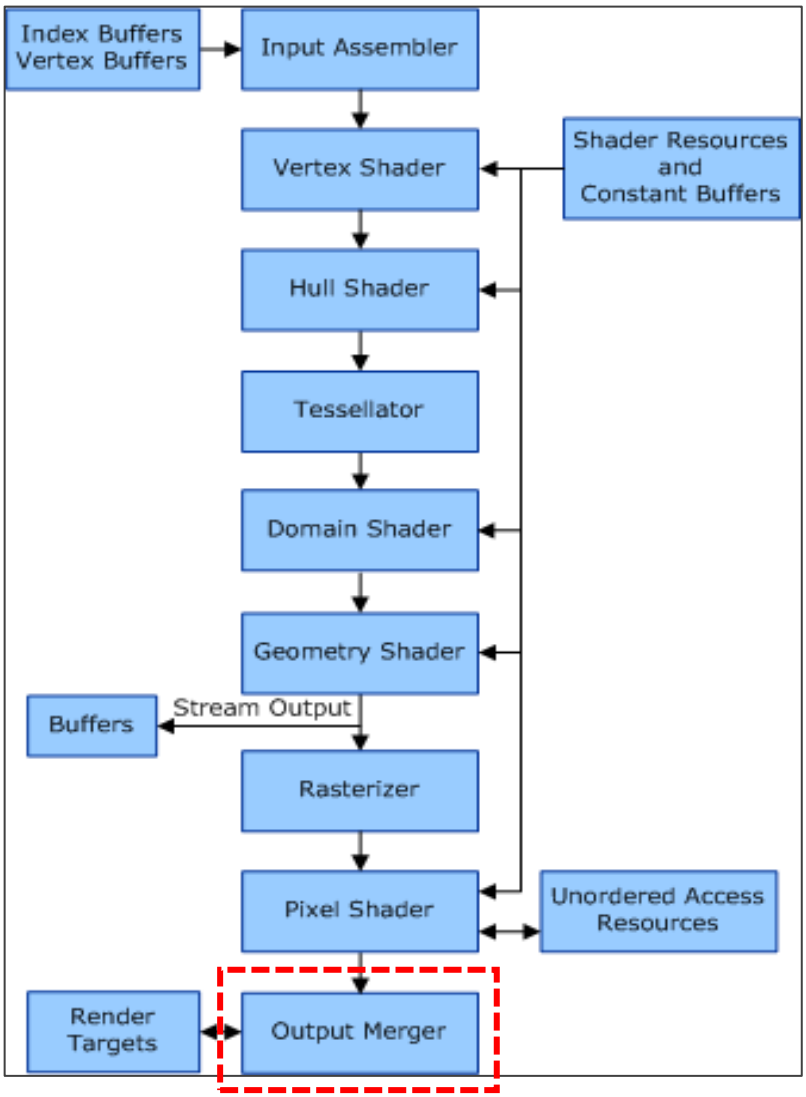
デフォルト
D3D11_FILTER_MIN_MAG_MIP_LINEAR

その他、サンプラー定義

https://docs.microsoft.com/ja-jp/windows/win32/api/d3d11/ns-d3d11-d3d11_sampler_desc

上手くいけば、下図のように描画されますが、透過されていません。

透過するためには、描画先となる色の情報が必要ですが、
基本的にシェーダーは描画先のピクセル情報を知ることができません。
では、どこで透過処理が行われているかというと、



最後のステージ、出力マージャーになります。
出力マージャーでは、描画先に書き込み最終的な色を決めます。

出力マージャー色のブレンディング

アルファブレンド
　描画先と、今から描画する色を透過合成します。

加算ブレンド
　描画先と、今から描画する色を加算合成します。

乗算ブレンド
　描画先と、今から描画する色を乗算合成します。

加算と乗算は以前、色の合成を行いましたので、イメージがつきやすいかと思います。
透過も非常に大切で、α値が1.0f(完全透明)だったら、今から描画する色が何色であれ、完全透明になりますので、描画先の色から変わりません。
α値が0.5fの場合、描画先の色と今から描画する色を半分ずつ取り入れた色にして、半透明を実現します。

DxLibで、このブレンディングを変更したい場合は、
SetDrawBlendMode を使用します。

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

アルファブレンドを行う、きちんと透明化されました。

DrawGraphの透過フラグをtrueにすると透過されるかと思いますが、
おそらく、内部的にアルファブレンドを使用しているのだと思います。

```
DrawGraph(mPosX, mPosY, texDragon, false);
```