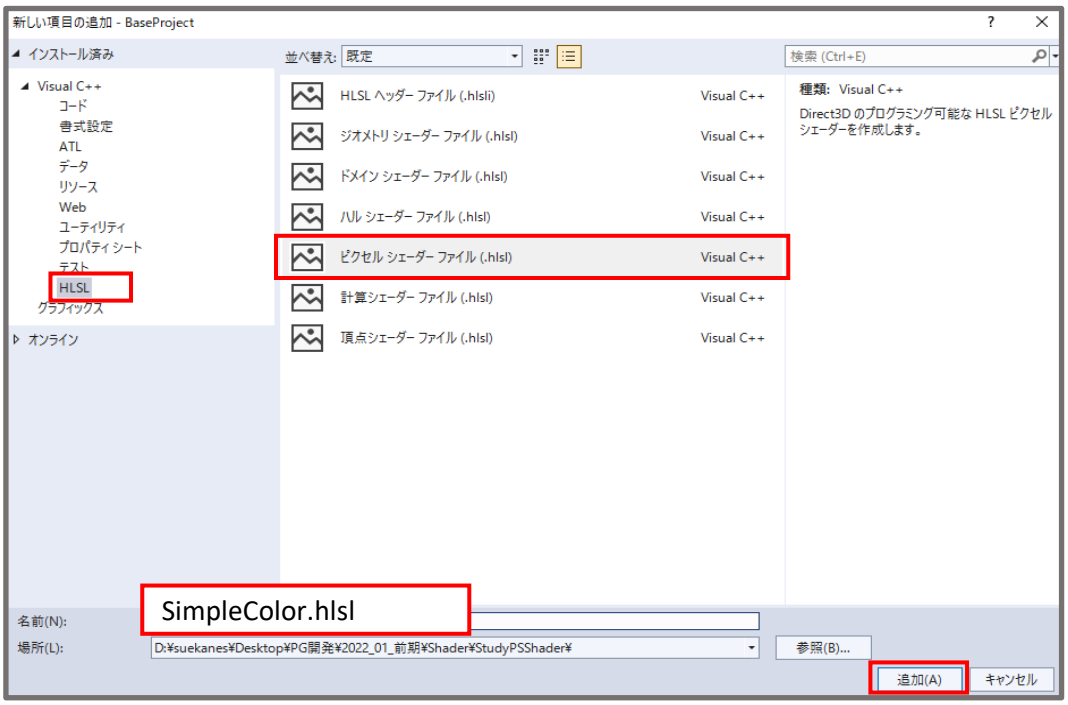
DxLibでオリジナルシェーダを動かす準備

DxLibでは、標準でDirectX11が動作していますので、
DirectX11用の準備を行っていきます。
まずはイメージし易いピクセルシェーダを作成して、動かしていきましょう。

Step① ピクセルシェーダファイルを作る

[追加]->[新しい項目]
[HLSL]->[ピクセルシェーダファイル]



こんな感じの中身になっていると思います。

```
float4 main() : SV_TARGET
{
    return float4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

main関数は、シェーダの開始地点、
オール1.0＝不透明な白色をピクセルの色情報として返します、
という構文です。

Step② １回ビルドしてシェーダをコンパイル

コンパイル先のフォルダにcsoファイルが作成されていることを確認する

"x64/Debug/SimpleColor.cso"

※ 本来なら、コンパイルされたcsoファイルを
きちんとしたプロジェクトフォルダに配置した方が良いが、
コンパイルの度にファイル更新するのが手間なので、
授業ではこのまま使用します

Step③ ピクセルシェーダファイルのロード／リリース

int LoadPixelShader(char *FileName) ;
ピクセルシェーダーバイナリを読み込みシェーダーハンドルを作成する

SimpleColor.csoをロードして、シェーダーハンドルを
メンバ変数に格納しておきましょう。

不要になったら削除しましょう。

int DeleteShader(int ShaderHandle) ;
シェーダーハンドルを削除する

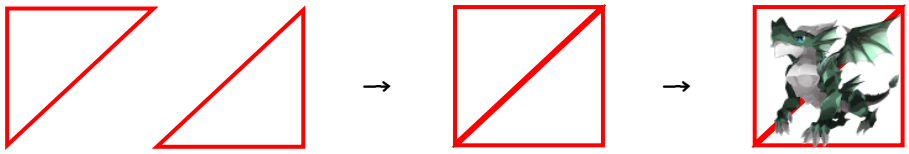
Step④ シェーダを使用して２Ｄポリゴンを描画

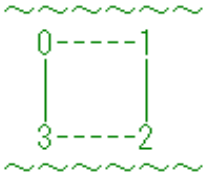
```
int DrawPolygonIndexed2DToShader(
    VERTEX2DSHADER *Vertex, int VertexNum,
    unsigned short *Indices, int PolygonNum ) ;
```

シェーダーを使って２Ｄポリゴンを描画する(インデックスを使用)

VERTEX2DSHADER *Vertex ： ポリゴンを構成する頂点配列の先頭アドレス
int VertexNum ： 頂点の数(Vertex で渡す配列の長さ)
unsigned short *Indices ： 頂点番号配列の先頭アドレス
int PolygonNum ： 描画するポリゴンの数

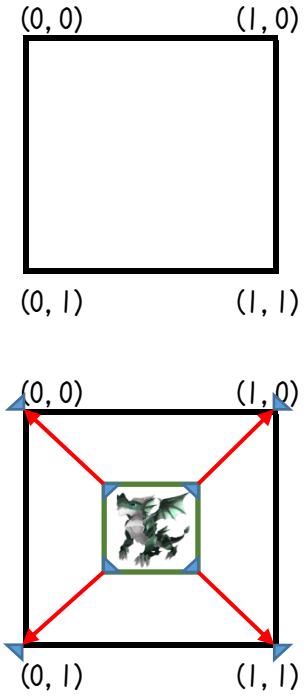
２Ｄ画像でシェーダを使用するためには、２つの三角ポリゴンを用意して、
３Ｄポリゴンにテクスチャを貼り付ける形で実装します。





左上
```
mVertex[0].pos = {0.0f, 0.0f, 0.0f};
mVertex[0].u = 0.0f;
mVertex[0].v = 0.0f;
```

UV座標(u, v)



右上
```
mVertex[1].pos = {128.0f, 0.0f, 0.0f};
mVertex[1].u = 1.0f;
mVertex[1].v = 0.0f;
```

右下
```
mVertex[2].pos = {128.0f, 128.0f, 0.0f};
mVertex[2].u = 1.0f;
mVertex[2].v = 1.0f;
```

左下
```
mVertex[3].pos = {0.0f, 128.0f, 0.0f};
mVertex[3].u = 0.0f;
mVertex[3].v = 1.0f;
```

描画手順

```
// シェーダー設定
SetUsePixelShader(psSimpleColor);

// ポリゴン生成
MakeSquereVertex();

// 描画
DrawPolygonIndexed2DToShader(mVertex, 4, mIndex, 2);
```

実行結果

描画対象となっている全ピクセルの色が白で塗りつぶされました。

```
float4 main() : SV_TARGET
{
    return float4(1.0f, 1.0f, 1.0f, 1.0f);
}
```

今回は、画像サイズが128×128ですので、
16,384回、main関数が実行されて、全てのピクセル箇所で、
白色が決定された、という流れになります。

試しにシェーダーを変更して、色を変えてみましょう。

ＯＫ！

シェーダープログラムの読み方

mainの返り値 float4、return float4() は、
これまでの関数プログラムと同じ要領なので、イメージしやすいかと思います。
SV_TARGETは、シェーダー用語でセマンティクスと言います。

セマンティクスとは、変数(パラメータ)の使用意図を明確にするための文字列です。
今回は、SV_TARGETが使用されていますので、
main関数の返り値は、グラフィックス出力先の色として扱ってください、
という意味になります。

ピクセルシェーダーの役割的に、指定された座標の色を決める必要がありますので、
SV_TARGETが指定されていないとエラーになります。

pixel shader must minimally write all four components of SV_Target0
⇒ 少なくとも、SV_Target0に何らかの値を書き込む必要があります！

メッセージに出てきている"SV_Target0"ですが、SV_Target1、SV_Target2・・・など、
複数の出力先に同時に色を書き込むことができます。
(マルチレンダーターゲット)
SV_TARGETは、ゼロ番目(メインレンダーターゲット)を意味します。

複数のセマンティクスを返り値として渡したい場合は、構造体を使用します。

```
// 構造体宣言
struct PS_OUTPUT
{
    float4 color0 : SV_TARGET0;
    float4 color1 : SV_TARGET1;
};

PS_OUTPUT main()
{
    PS_OUTPUT ret;
    ret.color0 = float4(0.7f, 0.4f, 0.4f, 1.0f);
    ret.color1 = float4(0.7f, 0.4f, 0.4f, 1.0f);
    return ret;
}
```

セマンティクスの種類・説明

https://docs.microsoft.com/ja-jp/windows/win32/direct3dhlsl/dx-graphics-hlsl-semantics

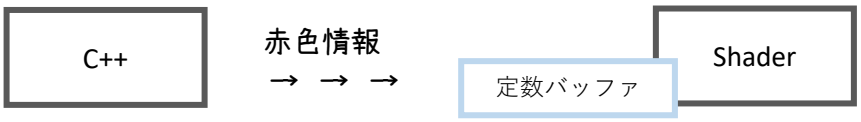
ピクセルシェーダーにおいて、SV_TARGETは最重要セマンティクスです。
上記サイトにも記載がありますが、SV_TARGETはDX10以降の名称になります。
DX9以前は、COLORという名称になっていますので、参考サイトによっては、
読み替え、置き換えが必要になります。

HLSLの簡易リファレンス

https://hlslref.wiki.fc2.com/

C++側からシェーダー側にパラメータを渡す(定数バッファDX11)

色情報をシェーダーに直書きするのは、柔軟性に欠けますので、
C++側からシェーダーにパラメータを渡して、色を変化させていきます。



定数バッファ自体は、DirectX11からの仕様です。
GPU上のメモリを事前に確保して、データを格納することができます。
これを利用して、C++ ⇔ シェーダー間のデータ渡しを行います。

定数バッファ概要
https://docs.microsoft.com/en-us/windows/win32/api/d3d11/ns-d3d11-d3d11_buffer_desc#remarks

DxLibで定数バッファを準備するには、

```
// シェーダー用定数バッファハンドルを初期化する
int CreateShaderConstantBuffer(int BufferSize) ;
```

返り値　　：　int　…　定数バッファハンドル
引数　　：　int　…　定数バッファサイズ
　　メモリに確保するByte容量
　　DX11の仕様になりますが、16の倍数を指定する必要があります。
　　16の倍数を指定する必要があります。
　　よく使用するfloat型が4バイトになりますので、
　　最低でもfloatを4つ分指定します。

If the bind flag is D3D11_BIND_CONSTANT_BUFFER, you must set the **ByteWidth** value in multiples of 16, and less than or equal to **D3D11_REQ_CONSTANT_BUFFER_ELEMENT_COUNT**.

```
void Init(void)
{
    // ピクセルシェーダのロード
    psCustomColor = LoadPixelShader((PATH_SHADER + "CustomColor.cso").c_str());
    // ピクセルシェーダー用の定数バッファを作成
    psCustomColorConstBuf = CreateShaderConstantBuffer(sizeof(float) * 4);
}                                                          ↑
```

16だと意味を持たせにくいので、せめてものsizeof

```
void DrawCustomColor(void)
{
    // シェーダー設定
    SetUsePixelShader(psCustomColor);

    // 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(psCustomColorConstBuf);
    cbBuf->r = 0.5f;
    cbBuf->g = 0.5f;
    cbBuf->b = 0.5f;
    cbBuf->a = 1.0f;

    // ピクセルシェーダー用の定数バッファを更新して書き込んだ内容を反映する
    UpdateShaderConstantBuffer(psCustomColorConstBuf);

    // ピクセルシェーダー用の定数バッファを定数バッファレジスタにセット
    SetShaderConstantBuffer(psCustomColorConstBuf, DX_SHADERTYPE_PIXEL, 0);

    // ポリゴン生成
    MakeSquereVertex();

    // 描画
    DrawPolygonIndexed2DToShader(mVertex, 4, mIndex, 2);
}
```

バッファへの書き込みにCOLOR_Fを使用していますが、

```
// 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(1つずつfloat設定)
float* cbBufFloat = (float*)GetBufferShaderConstantBuffer(psCustomColorConstBuf);
*cbBufFloat = 0.5f;
cbBufFloat++;
*cbBufFloat = 0.5f;
cbBufFloat++;
*cbBufFloat = 0.5f;
cbBufFloat++;
*cbBufFloat = 1.0f;
```

こんな感じで、1つずつ設定しても良いのですが、
なるべく意味を持たせたいので、COLOR_F(float4つの構造体 rgba)がおすすめ。

そして、ピクセルシェーダーを以下のように作成します。

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

float4 main() : SV_TARGET
{
    //return float4(1.0f, 1.0f, 1.0f, 1.0f);
    return g_color;
}
```

register(b0)　は、0番目スロットで、
下記コードの第3引数とリンクしています。

```
SetShaderConstantBuffer(psCustomColorConstBuf, DX_SHADERTYPE_PIXEL, 0);
```

定数バッファを増やしたい場合は、0,1,2,3,4・・・としていきます。

注意！！

DXLibでは,定数バッファのb0、b1、b2が使用されているので、
b3あたりから使い始めた方が無難。

```
// 頂点シェーダー・ピクセルシェーダー共通パラメータ
cbuffer cbD3D11_CONST_BUFFER_COMMON                    : register( b0 )
{
    DX_D3D11_CONST_BUFFER_COMMON                       g_Common ;
} ;

// 基本パラメータ
cbuffer cbD3D11_CONST_BUFFER_PS_BASE                   : register( b1 )
{
    DX_D3D11_PS_CONST_BUFFER_BASE                      g_Base ;
} ;

// シャドウマップパラメータ
cbuffer cbD3D11_CONST_BUFFER_PS_SHADOWMAP              : register( b2 )
{
    DX_D3D11_PS_CONST_BUFFER_SHADOWMAP                 g_ShadowMap ;
} ;
```

そんなこんなで、
C++側からシェーダーで使用する色を設定することができました。