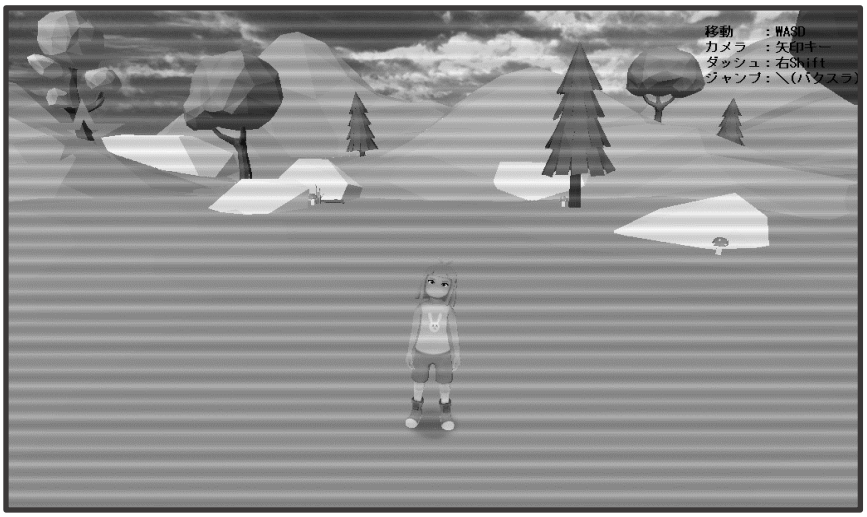
ポストエフェクト

ポストエフェクトとは、通常の描画が終わったあとに、
画面全体にかける演出効果のことです。

通常描画

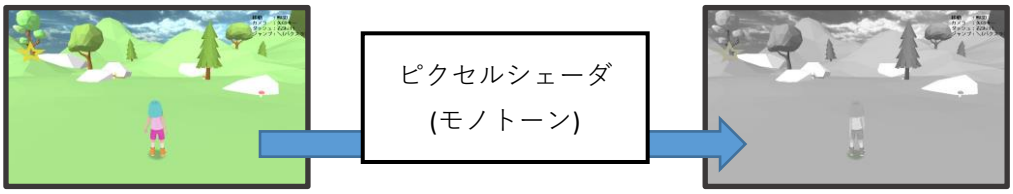


通常描画
＋
モノトーン
＋
走査線

通常描画
＋
モノトーン
＋
走査線
＋
レンズ歪み

通常描画
＋
モノトーン
＋
走査線
＋
レンズ歪み
＋
ビネット
↓
ホラー演出

ピクセルシェーダを使用して、通常描画した画面スクリーンを
１つのテクスチャとして、画像加工していく流れになります。

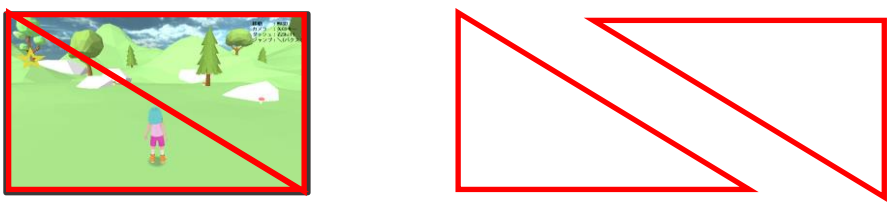


また、今回は、３Ｄ空間上のオブジェクトをシェーダを使用して
描画するのではなく、画面全体(スクリーン全体)、２Ｄ空間上に描画を
行う形になりますので、DxLibの関数としては、

DrawPolygonIndexed2DToShader

を使用します。

２Ｄとはいえ、ポリゴンは作る必要がありますが、
座標はスクリーン座標(２Ｄ座標)で良いため、スクリーンサイズを
使用して、画面全体を覆う２つのポリゴンを作れば良いです。



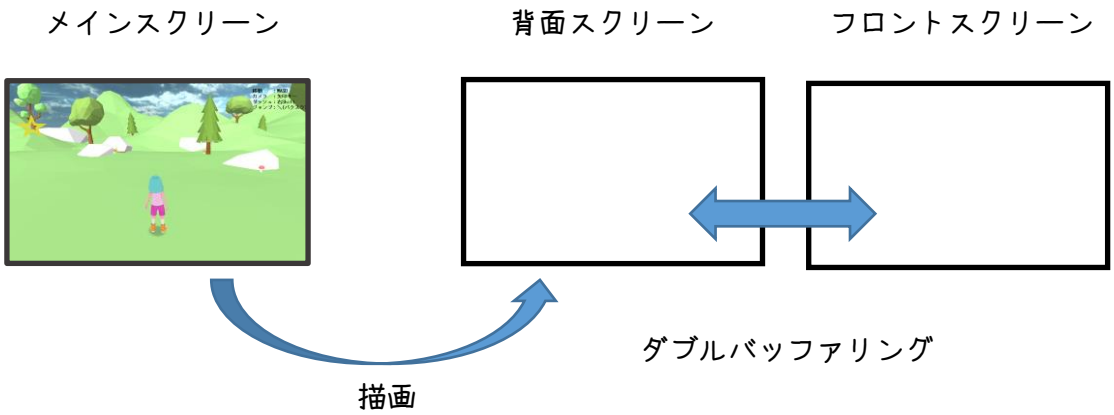
ポリゴンの作成例

```
// 頂点
VERTEX2DSHADER vertexs_[NUM_VERTEX];

// 頂点インデックス
WORD indexes_[NUM_VERTEX_IDX];

float sX = static_cast<float>(0.0f);
float sY = static_cast<float>(0.0f);
float eX = static_cast<float>(1024.0f);
float eY = static_cast<float>(640.0f);

// 4頂点の初期化
for (int i = 0; i < 4; i++)
{
    vertexs_[i].rhw = 1.0f;
    vertexs_[i].dif = GetColorU8(255, 255, 255, 255);
    vertexs_[i].spc = GetColorU8(255, 255, 255, 255);
    vertexs_[i].su = 0.0f;
    vertexs_[i].sv = 0.0f;
}

// 左上
vertexs_[cnt].pos = VGet(sX, sY, 0.0f);
vertexs_[cnt].u = 0.0f;
vertexs_[cnt].v = 0.0f;

// 右上
vertexs_[cnt].pos = VGet(eX, sY, 0.0f);
vertexs_[cnt].u = 1.0f;
vertexs_[cnt].v = 0.0f;
```

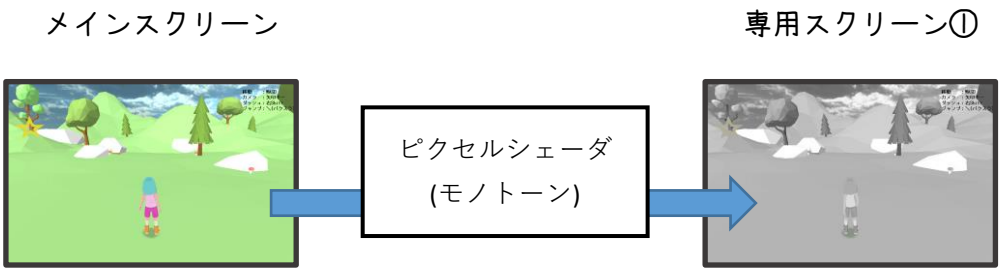
描画制御

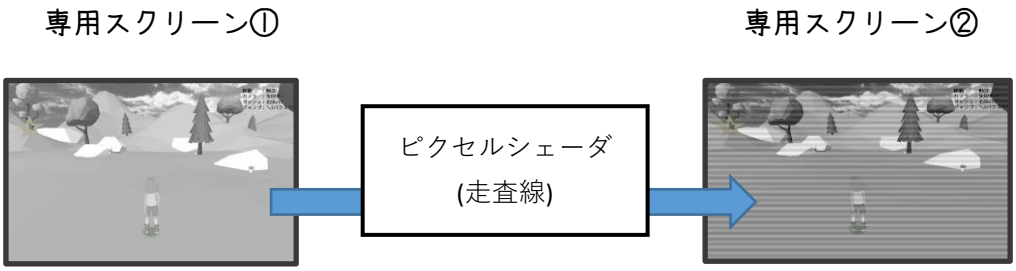
アーキテクチャの授業教材では、背面スクリーンに直接描画していますが、
背面スクリーンは、テクスチャとしてピクセルシェーダに使用できないため、
通常描画は、別のスクリーンを作成してそちらに描画するようにします。
そして、描画フェーズの最後に、背面スクリーンに描画します。

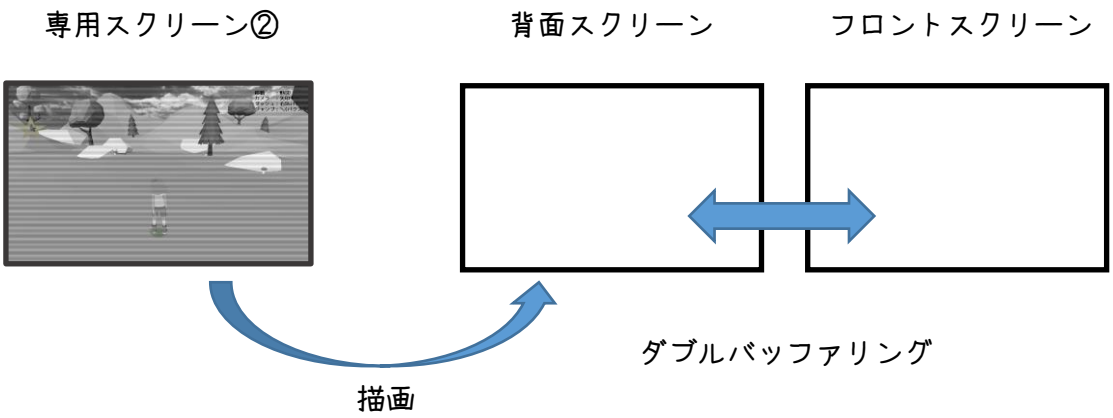
```
void SceneManager::Draw(void)
{

    // 描画先グラフィック領域の指定
    //（３Ｄ描画で使用するカメラの設定などがリセットされる）
    SetDrawScreen(mainScreen_);

    ～ 省略 ～

    // 背面スクリーンにメインスクリーンを描画
    SetDrawScreen(DX_SCREEN_BACK);
    DrawGraph(0, 0, mainScreen_, true);

}
```



ポストエフェクトを行う場合、専用スクリーンを作成して、
画像フィルタをかけるようにします。







専用スクリーンを作成する度に、使用メモリも増えますし、
描画スクリーンの切替や、描画の回数が増えるたびに、
処理時間も長くなりますので、負荷が高くなります。

できるだけ、スクリーンや描画回数は減らした方が良いですが、
まずは、実装することを優先していきましょう。

モノトーンや走査線ではなくて良いですので、
２種類以上のポストエフェクトを掛け合わせて実装しましょう。

クラス設計

２種類のポストエフェクトを実装して貰いました。
GameSceneで直書きしている場合、

- コードが長い
- 同じようなコードを複数箇所書いている
- 可読性が低い

上記のような感想を持てた方は、しっかりプログラミングに慣れている
状態かと思います。
このままだと、３つ目のポストエフェクトは書きたくなくなってきますし、
後々、追加や修正に苦労するのが目に見えています。

とはいえ、どのようにクラス設計・実装を行うのが正解なのか、
すぐに具体的なイメージが沸くわけではありませんし、
設計に関しては、実装の経験量や、幅広い知識量が必要になってきます。

手詰まりになりやすい作業ですし、なかなか自分のモノにするのが
難しい領域ですが、まずは設計イメージを可視化することから始めましょう。

手順①　必要な情報を洗い出す

ポストエフェクト用のスクリーンハンドル
シェーダハンドル
定数バッファハンドル
定数バッファの内容
シェーダに渡すテクスチャ
描画するポリゴン(頂点)情報
描画するポリゴンのインデックス情報

⇒これらがメンバ変数になる

手順②　必要な機能を洗い出す

シェーダのロード
定数バッファの作成
シェーダのメモリ解放
定数バッファのメモリ解放

ポリゴンの生成機能
ポリゴンの位置調整機能

シェーダを使用した描画機能
- ・シェーダの設定
- ・定数バッファの設定
- ・テクスチャの設定

⇒これらがメンバ関数になる

手順③　どういうクラス構成だったら
効率的か？
わかりやすいか？
使いやすいか？
汎用的か？
処理速度が早いか？
様々な点を考慮しながら考える。
※但し、キリは無いし、絶対的な正解も無いので、
最初は、重要視したい点を絞る
“わかりやすさ”がオススメ

わかりやすさ や 使いやすさ を考えた場合、

可能であれば、

```
postEffect->Draw();
```

この関数実行１発で、ポストエフェクトが実現されるのが理想。
とても、わかりやすいし使いやすい。
完全ではないにしろ、ここに近づけるようにする。

Draw関数一発で済ますためには、事前準備が必要になる。
シェーダを使用するにあたり、事前準備が手間で面倒なのは、
体感して貰っていると思いますので、これを簡略化したい。

また、描画を行う際の設定も煩雑。これも簡略化したい。

『 事前準備 』 と 『 描画設定 』 を簡略化して、
Draw 1発で描画されるように使いやすくしたい。

このように、情報のグルーピングを行うと、
わかりやすさも増していきます。

手順④　発明はしなくていい。真似よう。

ある意味、皆さんの可能性を潰してしまう話なので、躊躇はありますが、
少なくとも、私は、これまでずっと真似をし続けてきました。

IT、ゲーム限らず、会社の先輩、上司、本、インターネット、、、
私のこれまでの制作物の中で、完全なオリジナル技術は1つもなく、
既存技術の理解、応用、組み合わせで仕事を行ってきました。

基盤系、技術研究などの職種でしたら、オリジナル技術の発明が
必要とされることもあるかもしれませんが、
一般のプログラマー、フロント系は、発明なんて必要ありません。

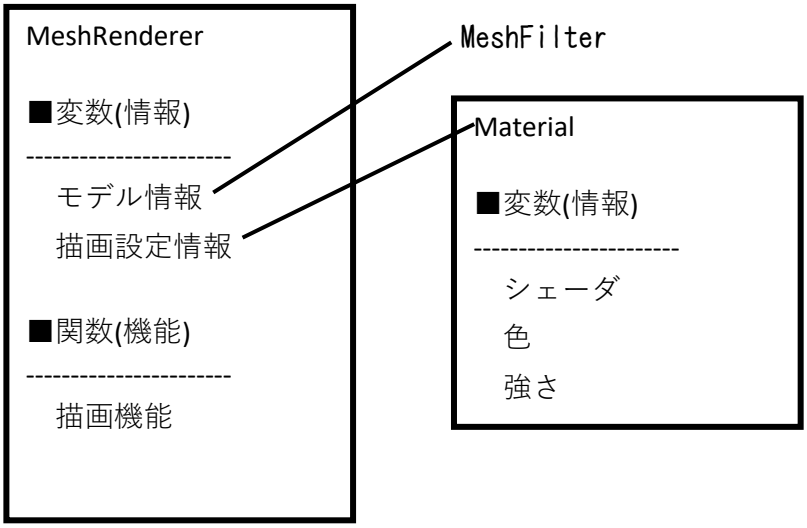
クラス設計も同じで、真似れそうなら、真似ましょう。
※その上で、理解する努力は行うこと

私は、Unity出身なのと、学校の授業でもUnity科目がありますので、
Unityから真似ることが多いです。

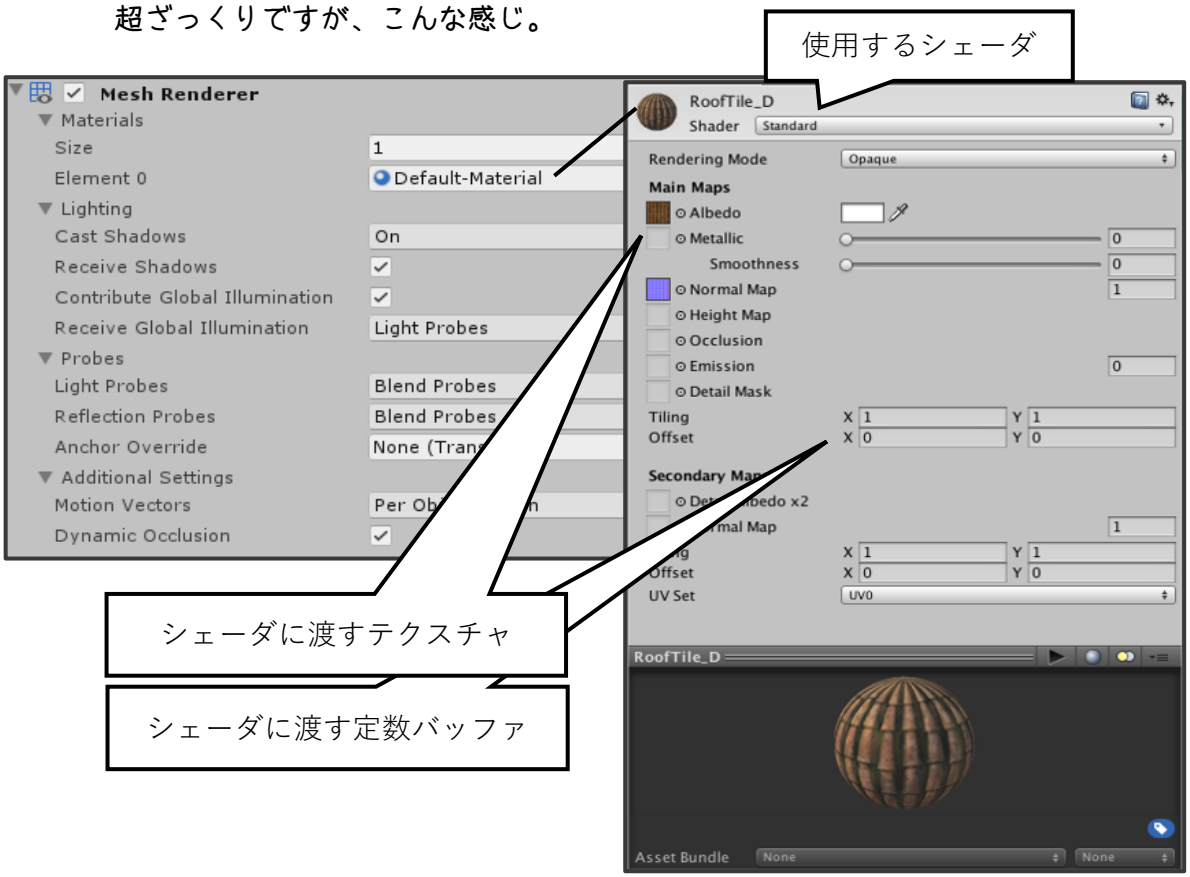
今回も、Unityから真似ます。

Unityでは、３Ｄモデルを描画する時は、
MeshRenderer という３Ｄモデルの描画機能を使用して、
３Ｄモデルが画面に描画されます。

そして、MeshRenderer には、Material という描画に必要な設定が、
必ず引っ付くようになっています。



超ざっくりですが、こんな感じ。



手順⑤　ある程度、方向性が決まったらクラスを書く

一発で納得できるクラス設計・実装はできません。
何度も何度も、修正することになります。
なので、いつまでも考えずに、ある程度イメージができたら、
イメージを詳細に固めるためにも、上手くいくかテストするためにも、
早くコードを書きましょう。

Materialクラス例

```
// シェーダハンドル
int shader_;

// 定数バッファの確保サイズ(FLOAT4をいくつ作るか)
int constBufFloat4Size_;

// 定数バッファハンドル
int constBuf_;

// テクスチャアドレス
TEXADDRESS texAddress_;

// 定数バッファ
std::vector<FLOAT4> constBufs_;

// 画像
std::vector<int> textures_;

/// <summary>
/// コンストラクタ
/// </summary>
/// <param name="shaderFileName"></param>
/// <param name="constBufFloat4Size"></param>
PixelMaterial(std::string shaderFileName, int constBufFloat4Size);
```

基本的には情報設定クラス。
他言語であれば、構造体に近い。

Rendererクラス例

```
// 座標
Vector2 pos_;

// 描画サイズ
Vector2 size_;

// 頂点
VERTEX2DSHADER vertexs_[NUM_VERTEX];

// 頂点インデックス
WORD indexes_[NUM_VERTEX_IDX];

// ピクセルマテリアル
PixelMaterial& pixelMaterial_;        ※マテリアルの実体生成前設定を
                                        強制する目的で参照

// コンストラクタ
PixelRenderer(PixelMaterial& pixelMaterial);

// 描画矩形の生成
void MakeSquereVertex(Vector2 pos, Vector2 size);
void MakeSquereVertex(void);

// 描画
void Draw(void);
void Draw(int x, int y);
```

全く一緒でなくて大丈夫です。
可能であれば、自分で設計してみましょう。