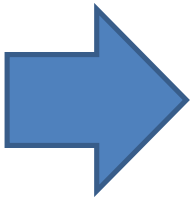

©Kizuna AI




©Kizuna AI


©Kizuna AI

グレースケール

RGBの情報から輝度情報を取り出し
その輝度情報を使用して白黒化する。

色の情報にはRGB形式だけでなく
YUVという形式もある。

輝度信号（Y）
青色成分の差分信号（U）
赤色成分の差分信号（V）の3要素からなるが
ここではYに注目する。

YUVとは？
　明るさ(輝度)と、色の成分とで表す。
人の目が輝度の変化には敏感なのに対して、
色の変化にはそれほど敏感でないことから、
テレビやJPEGなどの画像の圧縮を行う際に用いられる。

RGBからYUVへの変換は、以下のような
変換式によって行える。

グレースケールは画像のピクセルの明度Yを
黒0～1白で表して表示したもの。

$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B$$
$$U = -0.169 \times R - 0.3316 \times G + 0.500 \times B$$
$$V = 0.500 \times R - 0.4186 \times G - 0.0813 \times B$$

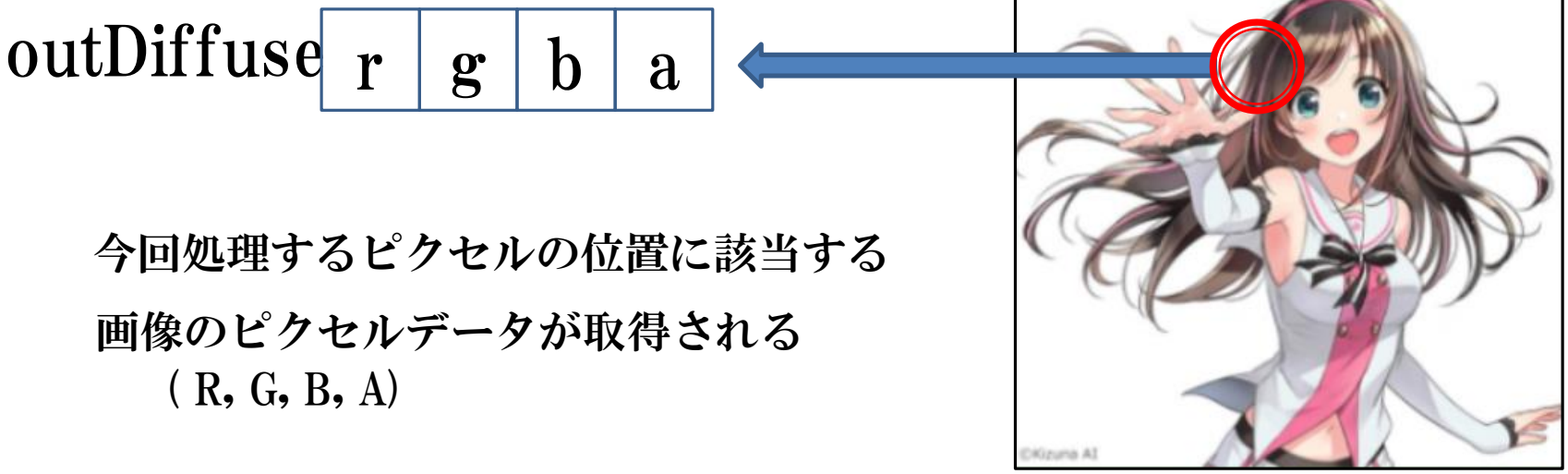
ピクセルシェーダーの例

```
//テクスチャの色を取得
outDiffuse = g_Texture.Sample(g_SamplerState,
                              In.TexCoord);

//Y(輝度変換)
outDiffuse = 0.299*outDiffuse.r +
             0.587*outDiffuse.g +
             0.114*outDiffuse.b;
```

ピクセルシェーダーの例

```
//テクスチャの色を取得
outDiffuse = g_Texture.Sample(g_SamplerState,
                              In.TexCoord);
```

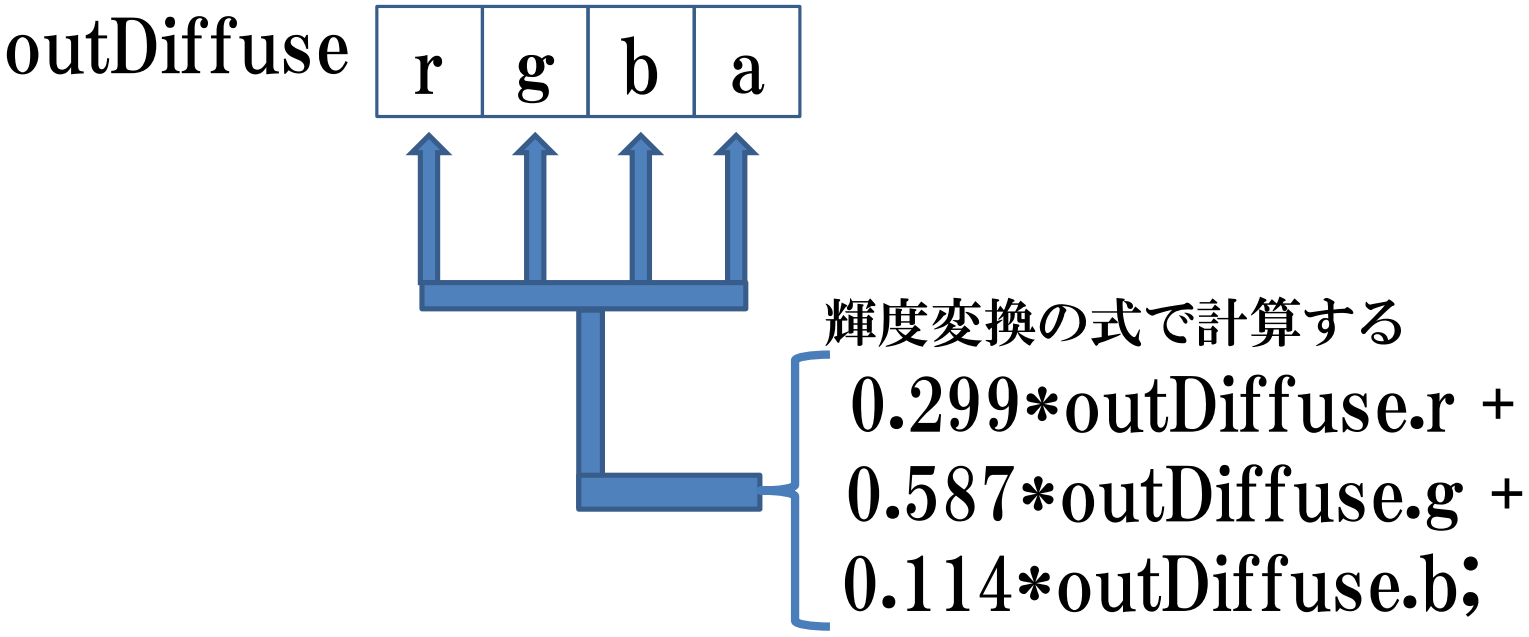


今回処理するピクセルの位置に該当する

画像のピクセルデータが取得される

（R, G, B, A）

```
//Y(輝度変換)
outDiffuse = 0.299*outDiffuse.r +
             0.587*outDiffuse.g +
             0.114*outDiffuse.b;
```



図にあるように、
outDiffuseには要素を指定しないで結果を
入れてしまっているので、全ての要素が
書き換えられ、αの値も壊される。
RGBの全ての要素が０～１の同じ値になるので
画像は白～黒のグレースケールとなる。

※最後にαに1.0fを入れておかないと、どの
ような見た目になるかわからない。

セピア調　変換

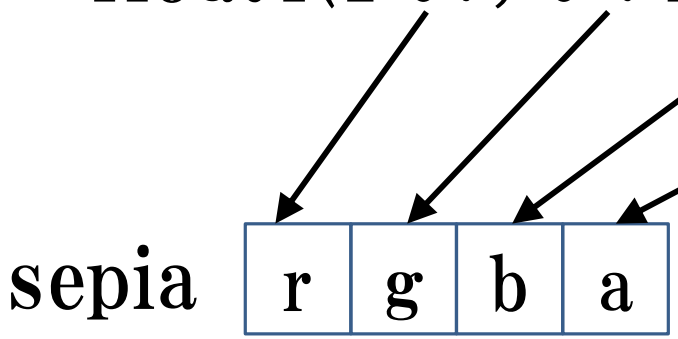
セピア調変換は、処理の途中まではグレースケールと同じ。
輝度を求めて、その輝度の通りに後から色を付ける。
付ける色は基本は茶色系なことが多いが、厳密に決まっているわけではない。

・・・グレースケールに変換してから・・・

```
float4 sepia = float4(1.07, 0.74, 0.43, 1.0f);
outDiffuse *= sepia;
```

ここでは変数に色のデータを持たせてから
計算する手法をとっている。

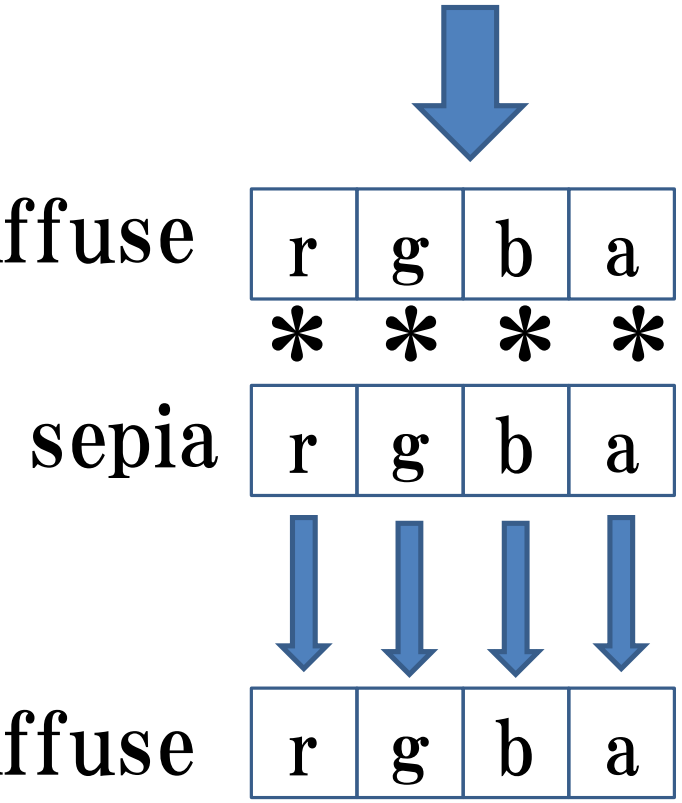
//変数sepiに色データをセット
float4 sepia = float4(1.07, 0.74, 0.43, 1.0f);



画像を何色にしたいか？はこのデータで決まる。

ローカル変数を作って、初期化する場合は
このような表記をすることも多い。

outDiffuse *= sepia;



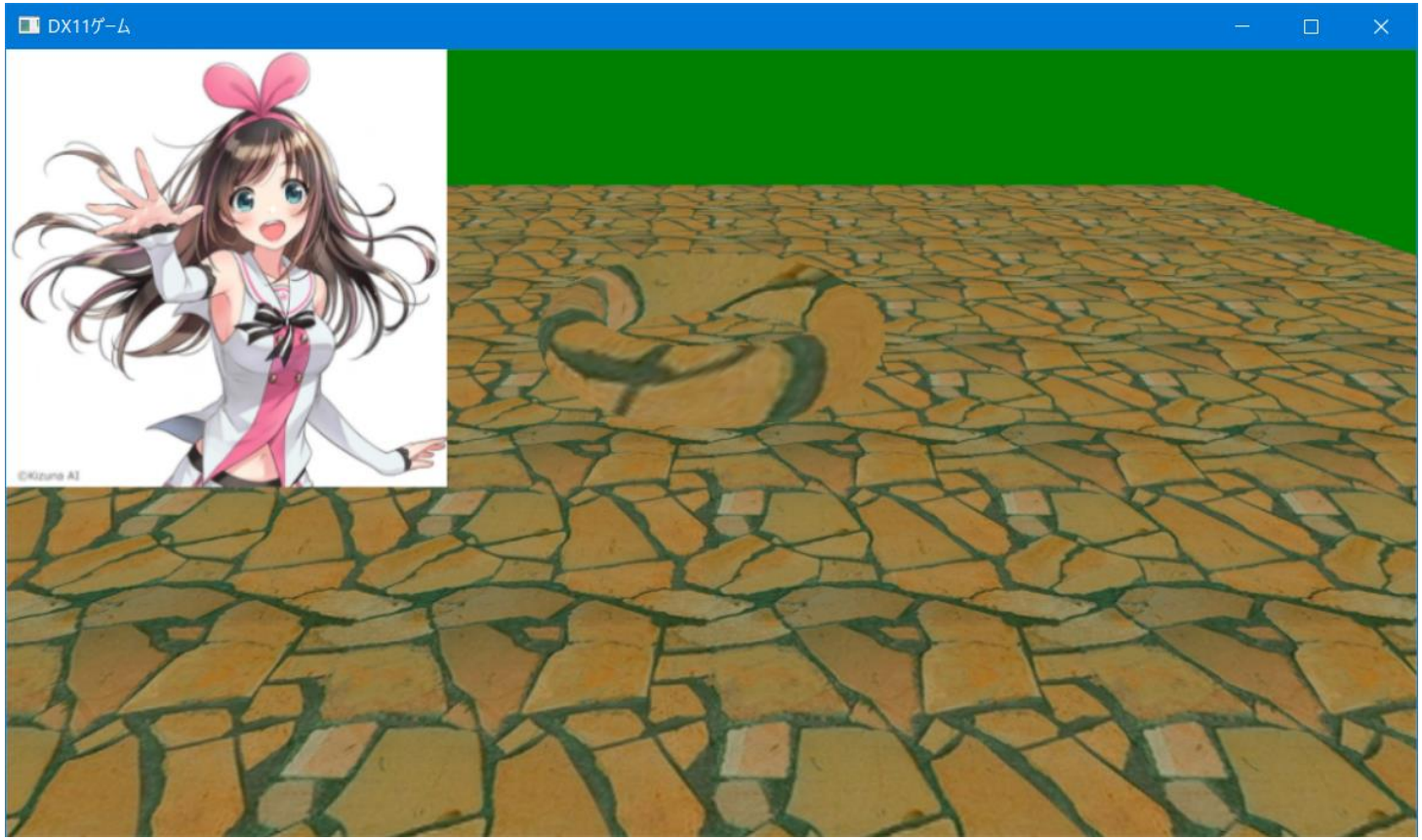
お互いの要素
同士を乗算する。

結果はそのまま
各要素へ入る

なんとなく個人的には、セピア色についてはこのくらいの値の方が落ち着いた感じがする。

```
float4 sepia = float4(240.0f/255.0f,
                      200.0f/255.0f,
                      148.0f/255.0f,
                      1.0f);
```

フィールドとモデルを出そう



とりあえず練習用のフィールドとモデルが
組み込まれているので
きちんと表示できるようにしてみよう。
field.h/field.cppおよびplayer.h/player.cpp

まずは、
以前、field.cppとplayer.cppのDraw()へ
returnを追加して描画を止めたので、
それを削除しておく。

授業の最初でpolygon.hとpolygon.cppを
修正したのと同じやり方で変更する。

ヘッダーファイル→
　　メンバー変数の追加
Init()&Uninit() →
　　シェーダーのロードと解放
Draw() →
　　シェーダーのセット

ヘッダーファイルへ追加した物
polygon.hやpolygo.cppからコピーでよい

```
//頂点シェーダーオブジェクト
ID3D11VertexShader*m_VertexShader;

//ピクセルシェーダーオブジェクト
ID3D11PixelShader*      m_PixelShader;

//頂点の構造体の中身の構成を表すオブジェクト
ID3D11InputLayout*      m_VertexLayout;
```

Init()へ追加した物

```
//バーテックスシェーダーファイルのロード＆オブジェクト作成
CRenderer::CreateVertexShader(&m_VertexShader,
        &m_VertexLayout, "unlitColorVS.cso");


//ピクセルシェーダーファイルのロード＆オブジェクト作成
CRenderer::CreatePixelShader(&m_PixelShader,
         "unlitColorPS.cso");
```

Uninit()へ追加した物

```
m_VertexLayout->Release();
m_VertexShader->Release();
m_PixelShader->Release();
```

Draw()へ追加した物

```
//インプットレイアウトのセット（DirectXへ頂点の構造を教える）
CRenderer::GetDeviceContext()->
        IASetInputLayout(m_VertexLayout);

//バーテックスシェーダーオブジェクトのセット
CRenderer::GetDeviceContext()->
        VSSetShader(m_VertexShader, NULL, 0);

//ピクセルシェーダーオブジェクトのセット
CRenderer::GetDeviceContext()->
        PSSetShader(m_PixelShader, NULL, 0);
```

それぞれ追加したら、実行確認を行う。

どこかしらの追加作業を忘れても、他の設定がそのまま使われて描画ができてしまうこともあるので、注意！！！

慎重にやること。