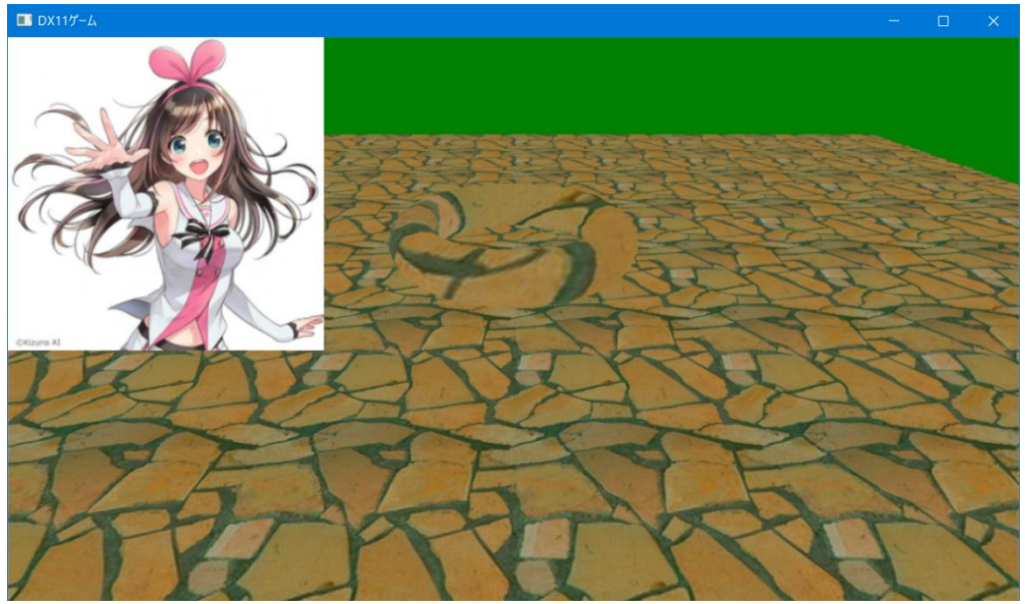
シェーダーで光源計算
（ライティング）

光源計算

ディフューズ光の計算なし



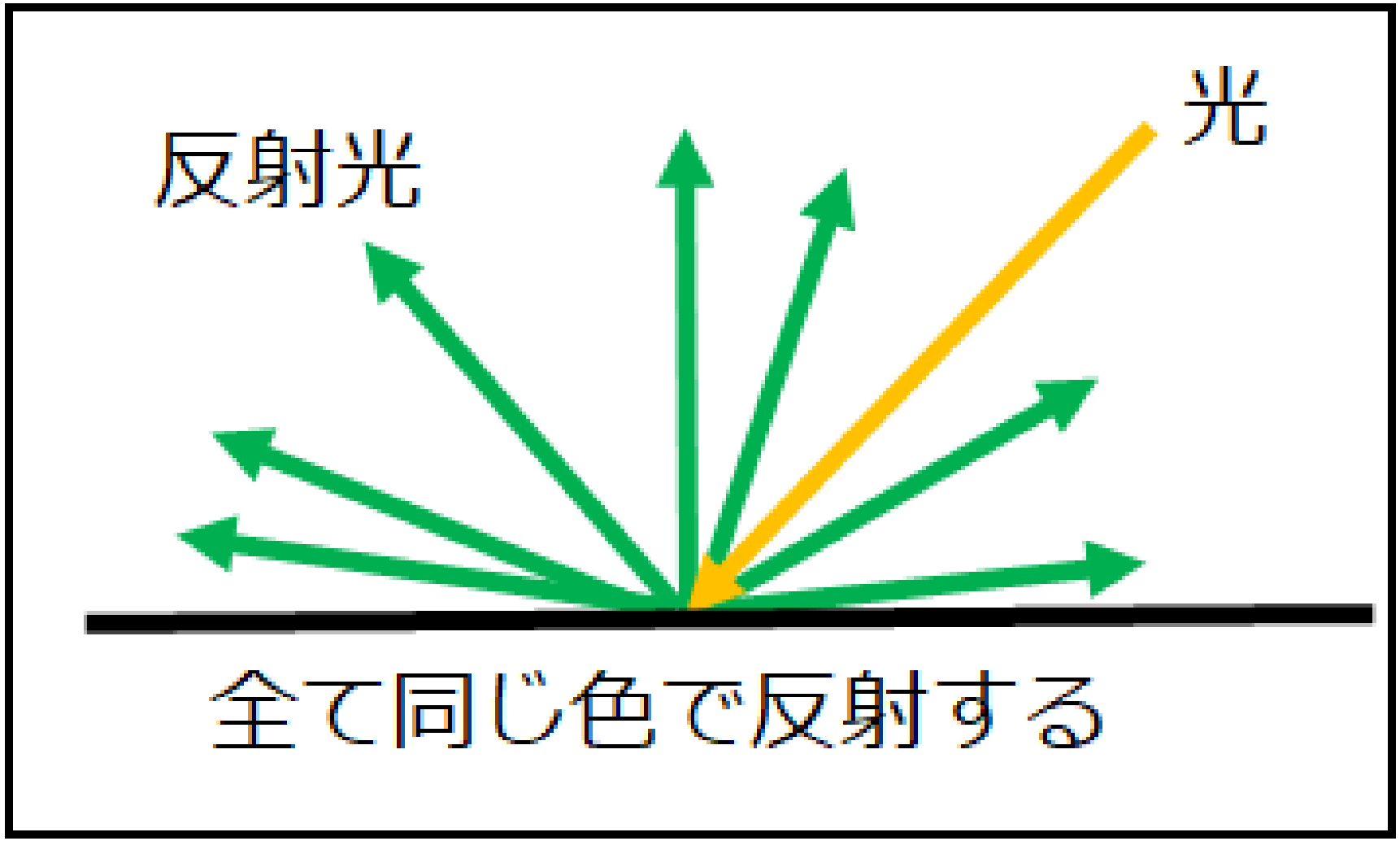
ディフューズ光の計算あり

一般に光源計算は拡散反射（ランバート反射）の処理をいう。

拡散反射（かくさんはんしゃ：diffuse　reflection）

石膏やチョークの表面での反射のように、どの方向からみても物体面の
輝度が一定となる反射である。

世の中いろいろな光源処理があるが、基本になっている部分はこれ。


光
反射光
全て同じ色で反射する

ランバートの余弦則

物体の表面で反射する光の輝度
（明るさ）は光の入射ベクトルと表面の
法線ベクトルのなす角度のコサイン（余弦）に
比例する。

$$I = kd \times Ii \times \cos\alpha$$

$$= kd \times Ii \times (N \cdot L)$$

I＝拡散反射光の強さ（**明るさ**）

Kd＝拡散反射率（**マテリアル値**）

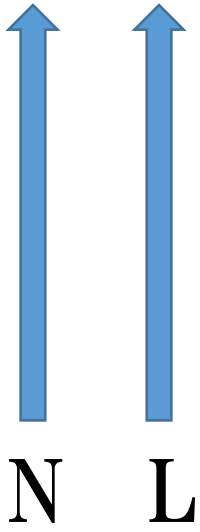
li＝入射光の強さ

N＝法線ベクトル（正規化済み）

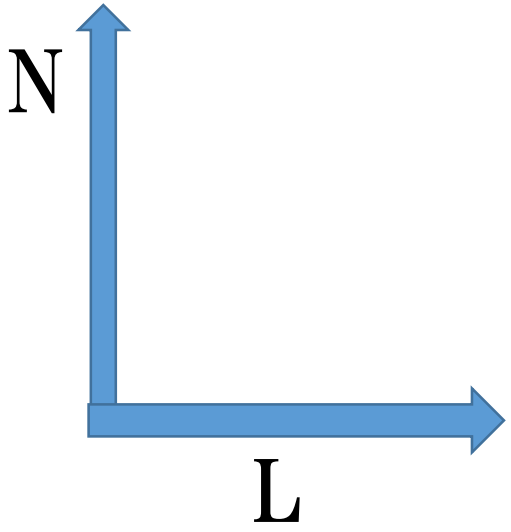
L＝光の方向ベクトル（正規化済み）

内積の特色

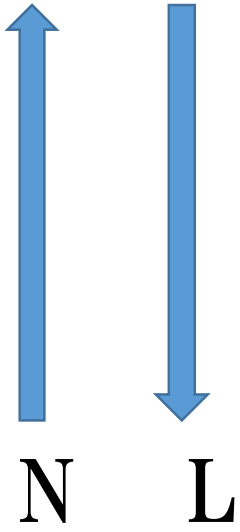
正規化されたベクトルNとLがあるとき



N・L=1.0



N・L=0.0



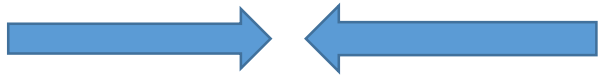
N・L=−1.0

これが今回の光源計算の核となる。

このときNとLは頂点の法線ベクトルNと光のベクトルLと考える。

NとLが同じ方向を向く場合は値が一番大きく(1.0)、一番多く光が当たって明るい。
NとLが直角以上の向きを持つ場合は値が0以下となり、光が当たらなくなり暗い。

ただし計算上は光の方向Lを反転させて考えないと現実的におかしくなるので、
内積の結果の符号を反転させる。

L  N　　←光が面に真正面からあたっている状態といえる

この時の内積は-1.0だが現実的には一番明るい状態なので1.0とする。

頂点に光を当てる

頂点シェーダーを使って、光源計算を行う。
頂点には法線がある、光の情報はC言語側からシェーダーへ
渡す必要がある。

シェーダーファイルとして以下を作成する。
vertexLightingVS.hlsl
vertexLightingPS.hlsl

作成したらソリューションエクスプローラへ追加して、プロパティを
変更しておく。
・オブジェクトファイル名　→　%(Filename).cso
・シェーダーの種類　→　ピクセルシェーダー or　頂点シェーダー

ライト構造体　（LIGHT構造体）

サンプルではrender.h内でLIGHT構造体が定義されている。
これはmanager.cppのDraw()関数内でデータがセットされ、
シェーダーに渡されている。

よってシェーダー側で受け取る準備を作成する。
common.hlslへ構造体を作成する。

common.hlslへ追加

```
//ライトオブジェクト構造体とコンスタントバッファ
struct LIGHT
{
	bool Enable;
	bool3 Dummy;		//4の倍数にすると効率がいいので調整用
	float4 Direction;		実はC言語でも同じだがVS2017がやってくれている。
	float4 Diffuse;
	float4 Ambient;
};

cbuffer LightBuffer : register(b4)//コンスタントバッファ4番とする
{
	LIGHT Light; //ライト構造体
}
```

ピクセルシェーダー

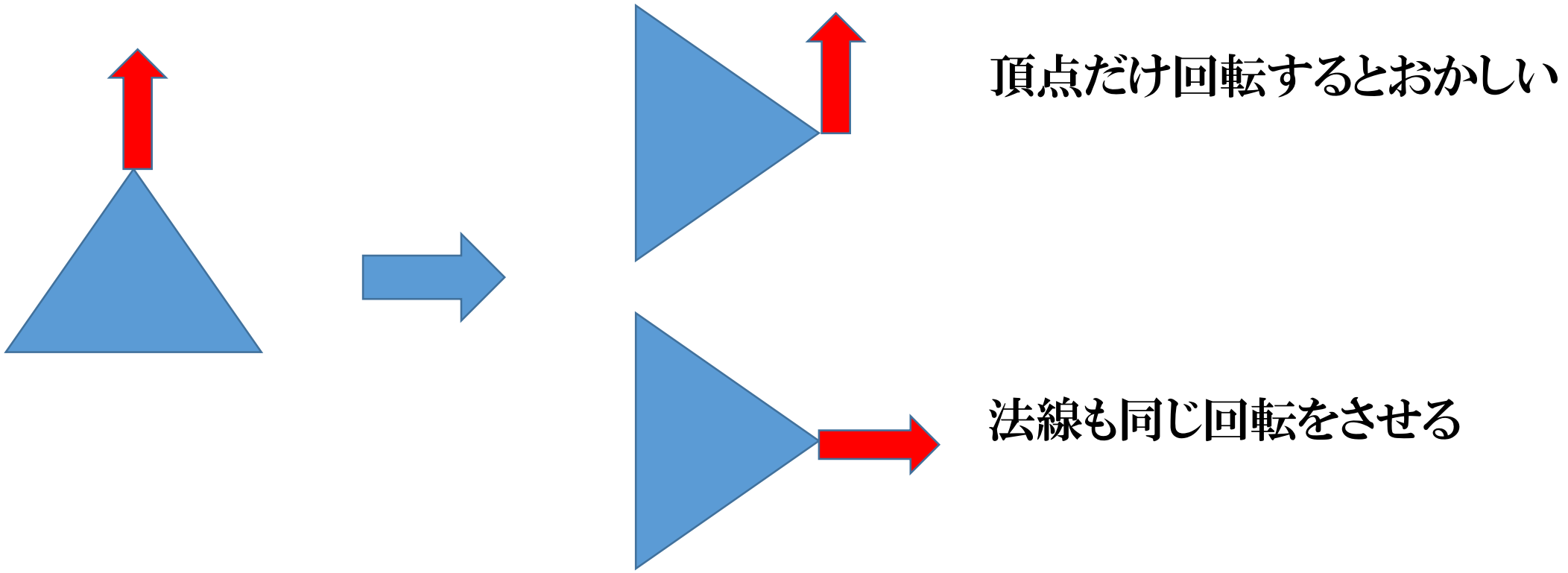
今回は頂点シェーダーが光源計算を行い、ピクセルシェーダーは特に
やることはないので下記のような処理としておく。（テクスチャ表示の状態）

```
#include "common.hlsl"

Texture2D g_Texture : register(t0);                 //テクスチャ0番
SamplerState g_SamplerState : register(s0);    //サンプラー0番

void main(in PS_IN In, out float4 outDiffuse : SV_Target)
{

   outDiffuse = g_Texture.Sample(g_SamplerState, In.TexCoord);  //テクスチャの色を取得
   outDiffuse *= In.Diffuse;                                                          //デフューズ(頂点の明るさ)を合成

}
```

頂点シェーダー

光源計算は、以下の段取りで進む。
これまでの頂点処理も必要なので同じものを作成するが、説明は省略とする。

1　法線は頂点と同じワールド変換をする。（ただし移動はしない）
　　一応正規化も行う。



２　回転させた法線と光のベクトルの内積を計算して符号を反転する。

明るさ　＝　－dot(光のベクトル、回転した法線)　　dotは内積を表す

この時、内積の値は-1.0～1.0となる。
しかし明るさ(色)の値は0.0～1.0の範囲なので、サチュレート(飽和処理)を行い、
0.0～1.0の間に強制的に収める。

３　結果をOut.Diffuseとして出力する。

法線と光のベクトルの関係から「明るさ」が計算できたので、それを頂点の色として
出力する。
これは単に0.0～1.0の明るさ値なのでグレースケールとなる。

以下、頂点シェーダーのコードを考える。

最初はこの状態

```
#include ″common.hlsl″


void main(in VS_IN In,　out PS_IN Out)
{
　　ここへ追加していく


}
```

法線の回転処理を追加する

```
//頂点法線をワールド行列で回転させる(頂点と同じ回転をさせる)
float4　worldNormal,　normal;　　//ローカル変数を作成

normal = float4(                    );　　//法線ベクトルのwを0とする(行列を乗算しても平行移動しない)
worldNormal = mul(                  );　　//法線をワールド行列で回転する
worldNormal = normalize(worldNormal);//回転後の法線を正規化する
```

法線をローカル変数へコピーした後、wを0にする。
ここが1だと行列で変換した際に平行移動の値が0となるため、回転だけが適用される。
法線は向きのみが意味を持つベクトルのため、多くの場合はこのような措置をとる。
（同じことをするD3DXVec3TransformNormal()関数というものがある）

また、ワールド行列に1.0倍以外のスケールが入っていると法線の長さが変化してしまうため正規化を行う。

光源計算の処理を追加する

```
//明るさの計算
float   light = -dot(                                    );  //光ベクトルと法線の内積   XYZ要素のみで計算
light = saturate(light);                                     //明るさを0～1の間で飽和化する
```

シェーダーでの内積計算はdot関数を使用する。
引数には、光のベクトルと回転後の法線を指定するが、ーをつけて符号の反転を忘れないよう注意する。

前述の通り、内積の値を0.0から1.0に収めるためにsaturate関数を使って飽和化する。

計算された明るさの値を出力する

```
Out.Diffuse = light;              //明るさを頂点色として出力
Out.Diffuse.a = In.Diffuse.a;     //念のため
```

これまでのデフォルト処理を追加する

```
matrix wvp;
wvp = mul(World, View);
wvp = mul(wvp, Projection);                  //変換行列作成

Out.Position = mul( In.Position, wvp );  //頂点出力
Out.Normal = worldNormal;                //回転後の法線出力     In.Normalでなく回転後のものを出力
Out.TexCoord = In.TexCoord;              //テクスチャ座標出力
```

終了

いったんビルドしてエラーをなくす。
変数が初期化されていない旨のワーニングは出るかもしれないが、問題ない。

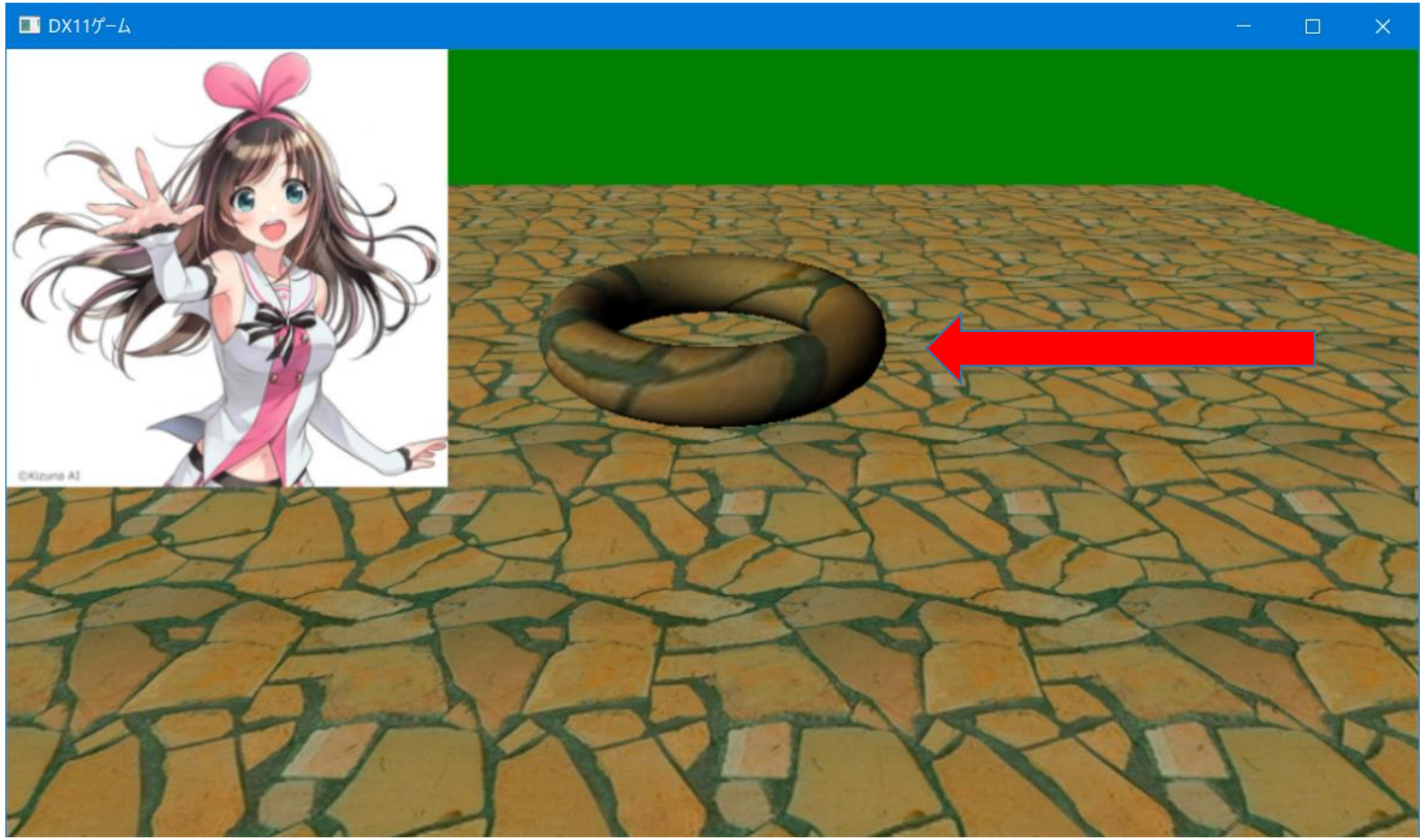
エラーが取れたら

player.cppのInit関数内で読み込むシェーダーのファイル名を

vertexLightingVS.cso
vertexLightingPS.cso

に変更して実行する。
これを忘れると意味がない。

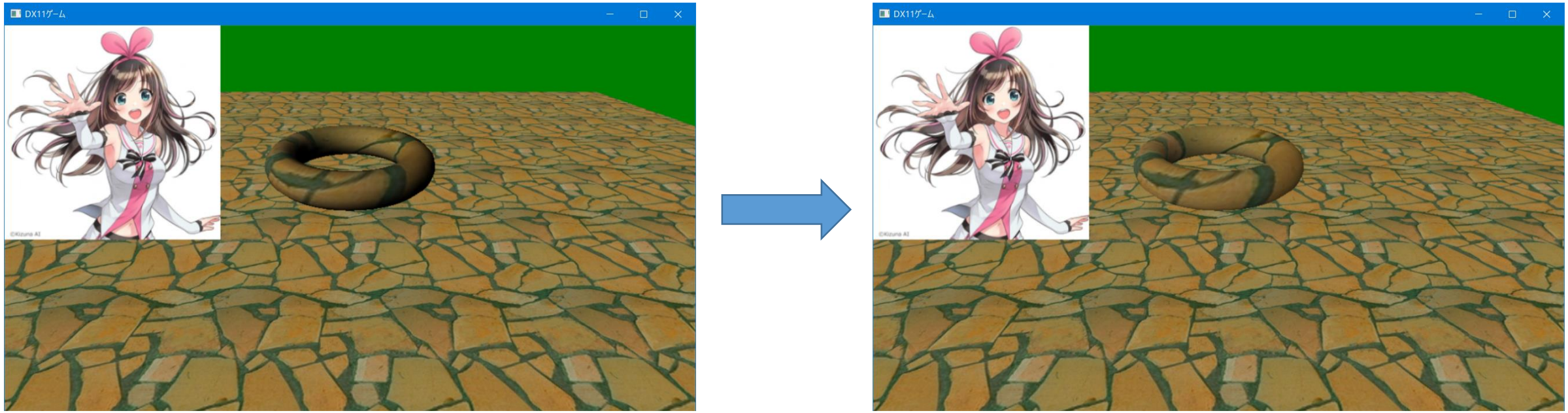
成功すると、下記のように光源処理されたドーナツが表示できる。



このまま処理すると、明暗がきついような感じになることが多い。
そこで、少しアレンジした「ハーフランバート」と呼ばれる手法で全体的に明るさを底上げしてあげることもある。

下記のようにすこしだけ明るさの計算法が異なる。

```
//ハーフランバートによる明るさの計算
float light = 0.5 - 0.5 * dot(Light.Direction.xyz, worldNormal.xyz);
```



頂点色を反映させる。

頂点の色を光源計算に反映させる場合は、明るさの出力時に頂点の色を乗算する。

```
Out.Diffuse = light * In.Diffuse;//明るさを頂点色として出力
```