UNITY

SHADER PROGRAMMING
UNOFFICIAL FANBOOK
01 : RENDERING PIPELINE

Unity Shader Programming Vol.01

XJINE　著

2020-04-10 版　　XJINE 発行

はじめに

この資料は Unity の基本的な操作方法については知識があり、シェーダプログラミングを経験したことがない方を読者に想定しています。あるいは、シェーダプログラミングの学習を進める上での補助的な資料として役立てることを想定しています。

この資料のよい点として (1) もっとも小さいサンプルからステップバイステップ形式で内容が整理されていること、(2) それぞれのステップでサンプルが用意されていることが挙げられます。

ほとんどすべての技術的な解説については Unity 公式の資料やその他の資料から確認することができますが、多くの場合に、それらは学習の順序などを考慮していません。教材形式の資料を用いて学習する一番のメリットは、さまざまな資料を集め、それらを取捨選択しながら順序立てる負担を軽減できることにあるでしょう。

一方で、この資料では高度な技術やその詳細、応用の類は解説されていません。特に、シェーダプログラミング以外の CG に関する詳細な解説や数学的な内容は、その多くを割愛しています。これらは「シェーダプログラミングの入門」に当たっては不要であり、より詳細な情報については、それらを専門に扱う資料を参照した方がよいためです。したがって、数学的な内容からきちんと順序立てて CG を学習したい方、高度な技術や CG による表現技法について学習したい方には、この資料は不向きであるといえます。

それでは、目を通して頂ける皆様にとって、この資料が少しでも役に立つことを祈り、冒頭のご挨拶とさせて頂きます。

資料の内容

この資料では Unity におけるシェーダの基本的なあつかい方と、オブジェクトが描画される過程、**レンダリングパイプライン**について、**Unlit Shader** を例に解説しています。

Unlit Shader とは「Un Lighting Shader」の略称で、陰影や輝きを表現する「ライティング」や「シェーディング」といった処理を含まないシェーダのことを指します。言い換えれば、オブジェクトに設定された色やテクスチャがそのまま描画される、もっともシンプルなシェーダです。

複雑な処理は含まれませんから、シェーダがどのように働いてオブジェクトを描画するのか、Unity ではどのように実装されているのかを学習しやすく、入門にあたって、もっとも適切な題材となっています。この資料の目標は、異なる他の資料の解説やソースコードを読み進めるだけの最低限の知識を共有することです。

サンプル

サンプルは「https://github.com/XJINE/UnityShaderProgramming_01_Sample」から取得することができます。継続的な更新のため、ブランチ名と書籍のバージョンを合わせて管理するようにしています。書籍のバージョンは巻末に記載されています。

この資料は有料にて提供されることがありますが、サンプルとなるリソースをオープンソースとして公開していることをご了承ください。学習資料として提供する上でもっとも適切な形と判断していますし、維持管理にかかるコストを集合知によって回避する目的があります。あるいは、購入前のサンプルとしての役割のためでもあります。

目次

第 1 章

シェーダとレンダリングパイプライン

シェーダのもっとも基本的な役割は、オブジェクトの描画方法を決定することです。まずはオブジェクトが描画される手順について確認していきましょう。オブジェクトは、概ね次の順序で描画されます。

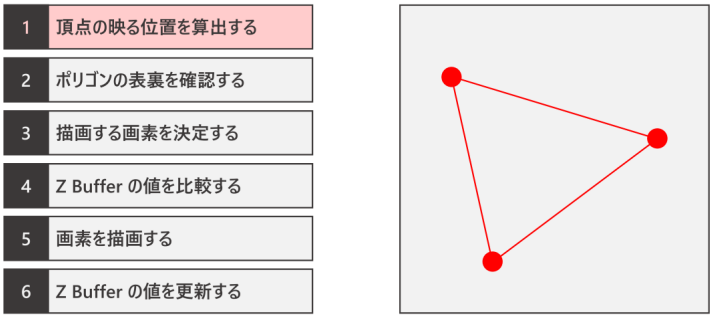
1. 頂点の映る位置を算出する
2. ポリゴンの表裏を確認する
3. 描画する画素を決定する
4. Z Buffer の値を比較する
5. 画素を描画する
6. Z Buffer の値を更新する

これらの一連の処理を実行する仕組みを「レンダリングパイプライン」と呼びます。また、これらの処理はすべて GPU 上で実行されます。

1.1 レンダリングパイプライン

レンダリングパイプラインの各処理について、詳しく確認していきましょう。

1.1.1 1. 頂点の映る位置を算出する

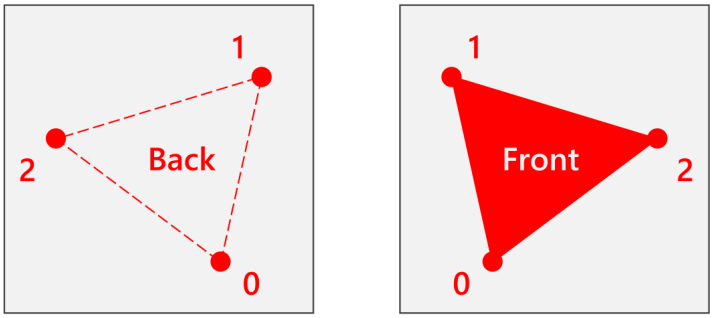


▲図 1.1　画面上に映された頂点

あるオブジェクトは、その形状を定義する複数のメッシュから構成されます。さらに、メッシュは複数のポリゴンから構成され、ふつう、ポリゴンは 3 つの頂点から構成されます。簡単に考えるために、ここではポリゴンを 1 つだけ描画することを考えましょう。

レンダリングパイプラインは、はじめに、画面上のどの位置にポリゴンの頂点が映し出されるかを算出します。1 つのポリゴンは 3 つの頂点から構成されますから、ここでは 3 回分の算出が実行されます。

1.1.2 2. ポリゴンの表裏を確認する



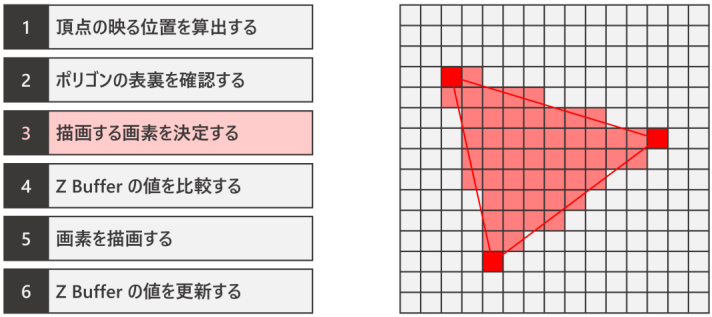
▲図 1.2 (左) 頂点が反時計回りに映る (右) 頂点が時計回りに映る

ポリゴンには表裏が設定されています。画面上に映し出されたポリゴンは、もしかしたら裏面を向いているかもしれません。ふつう、描画負荷を下げるために、裏面を向いているポリゴンは描画されません。

裏面を向いているポリゴンを描画しないようにする処理を「カリング (Cull・ing)」と呼びます。また、シェーダにおけるカリングを特に「Backface Culling」と呼びます。

なお、ポリゴンの表裏はその頂点が定義される順序によって決定されます。Unity では、ポリゴンを構成する頂点が時計回りに見える方向を表向きとして定義しています。

1.1.3 3. 描画する画素を決定する

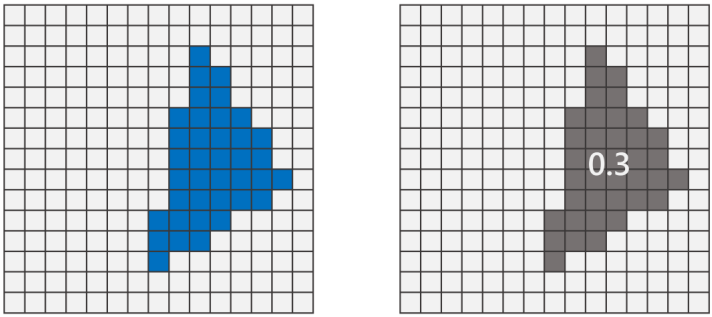


▲図 1.3　ポリゴンがどの画素に相当するかを決定する

表側を向いたポリゴンの頂点が映される位置が算出できました。次は、そのポリゴンが画面上のどの画素に描画されるかを決定します。この処理を 「ラスタライズ (Rasterize)」と呼びます。

ポリゴンを画素の集合によって表現する、といったイメージでもよいです。

1.1.4 4. Z Buffer の値を比較する

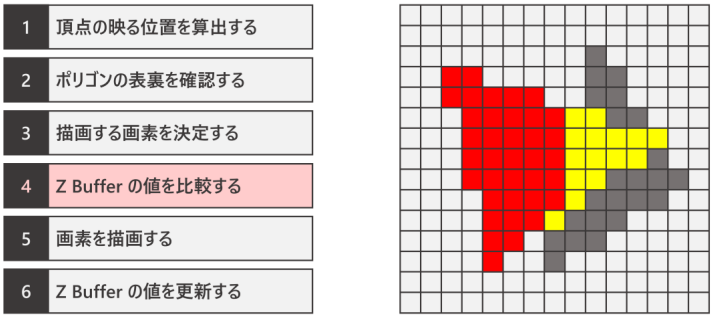


▲図 1.4 (左) 描画済みのポリゴン (右) 既存の Z Buffer

描画する画素を決定したら、その画素に描画済みのポリゴンがないかを確認します。描画済みのポリゴンが存在するかどうかを表す情報は、「Z Buffer」に書き込まれています。Z Buffer は、描画する画面と同じ解像度を持った異なる画面です。

Z Buffer は Depth Buffer や深度バッファとも呼ばれます。いずれの呼び方も一般的ですが、ここでは Z Buffer とします。

仮に、異なるポリゴンが 1 つだけ先に描画されているとしましょう。図の左がポリゴンを描画する画面で、右が Z Buffer です。Z Buffer の画素には、カメラからポリゴンまでの距離を表す値が書き込まれています。すなわち、深度の値 Z です。ここではカメラから 0.3 離れていたとします。

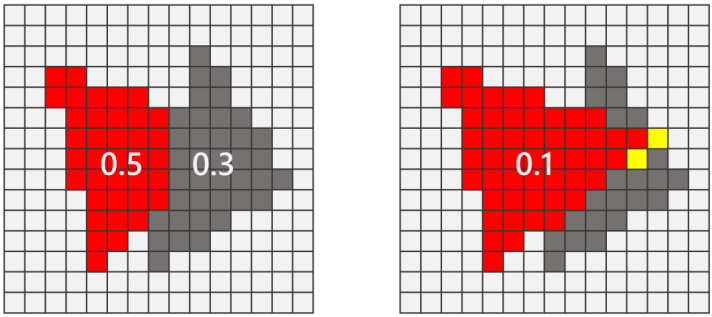


▲図 1.5　既存のポリゴンが描画された画素に重複する様子

現在描画しようとしているポリゴンの話に戻ります。現在描画しようとしているポリゴンは、既存のポリゴンが描画されている画素に部分的に重なります。

このとき、描画しようとしているポリゴンの Z の値が、すでに Z Buffer に書き込まれている Z の値よりも小さい場合は、描画しようとしているポリゴンを上書きして描画することができます。つまり、描画しようとしているポリゴンが、既存のポリゴンよりも手前にあることになります。

一方で、描画しようとしているポリゴンの Z の値が、すでに Z Buffer に書き込まれている Z の値よりも大きい場合は、描画しようとしているポリゴンを上書きすることができません。つまり、描画しようとしているポリゴンは、既存のポリゴンよりも奥にあることになります。

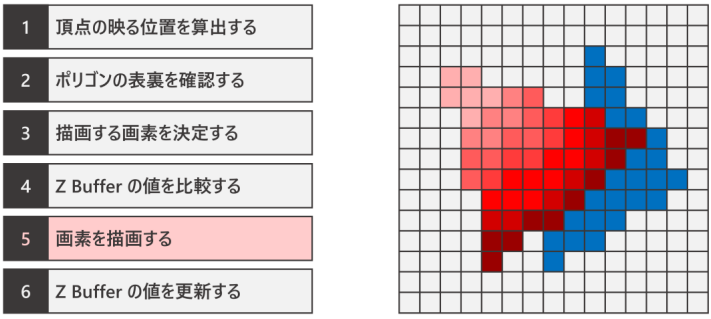


▲図 1.6　(左) Z がすべて大きい様子 (右) Z が部分的に大きい様子

描画しようとしているポリゴンの一部が既存のポリゴンの手前にあり、他の部分は奥にあることもあります。なぜなら、ポリゴンを構成する頂点は三次元の座標で表されますから、その面が、奥行き方向に傾くことがあります。

Z 値を比較する処理を「Z Test」と呼びます。あるいは Depth Test とも呼ばれます。また、Z 値を比較して新しいポリゴンが描画できる状態を「Z Test に成功する」などと言います。

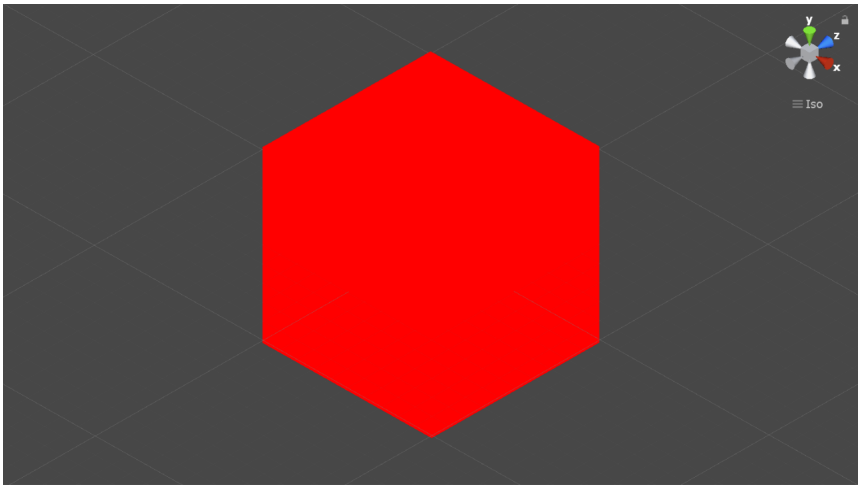
1.1.5　5. 画素を描画する



▲図 1.7　陰影をつけたり単に塗りつぶして画素を描画する

第 2 章

シェーダの基本的な構造と役割



▲図 2.1　シンプルに赤色で描画するシェーダ

レンダリングパイプラインの仕組みを確認したら、もっとも小さいシェーダプログラムを読み解いていきましょう。このシェーダはオブジェクトを赤く塗りつぶして描画するだけのシェーダです。光源の影響を考慮しないシェーダは、「Unlit シェーダ」と呼ばれます。そのまま Un + Lighting です。

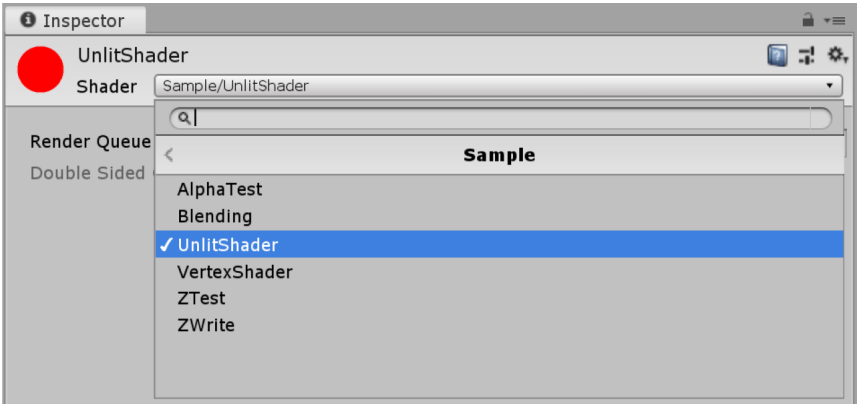
このサンプルは「02_UnlitShader」フォルダに含まれています。最初のサンプルですから、シェーダプログラムのソースコード全体をここに掲載しておきます。

▼ UnlitShader.shader

```
Shader "Sample/UnlitShader"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
            };

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 color = fixed4(1, 0, 0, 1);
                return color;
            }

            ENDCG
        }
    }
}
```

もっとも小さいシェーダプログラムとはいえ、すべて最初から解説しますから、情報量が多くなっています。したがって、一度に理解することは難しいでしょう。もしも理解が進まないときは、一度すべて読み進めてしまって、全体を見ながら繰り返し読み解くとよいと思います。

2.1 Shader 構文



▲図 2.2　UnityEditor 上に反映される階層とシェーダ名

Shader 構文は、シェーダの名前を定義するものです。UnityEditor 上から参照する名前にもなります。一般的なファイルシステムと同様に "/" によって階層を定義して整理することができます。ほとんどの場合に "種類/名前" のように定義されます。

ここでは Sample という階層に UnlitShader というシェーダ名を定義しています。

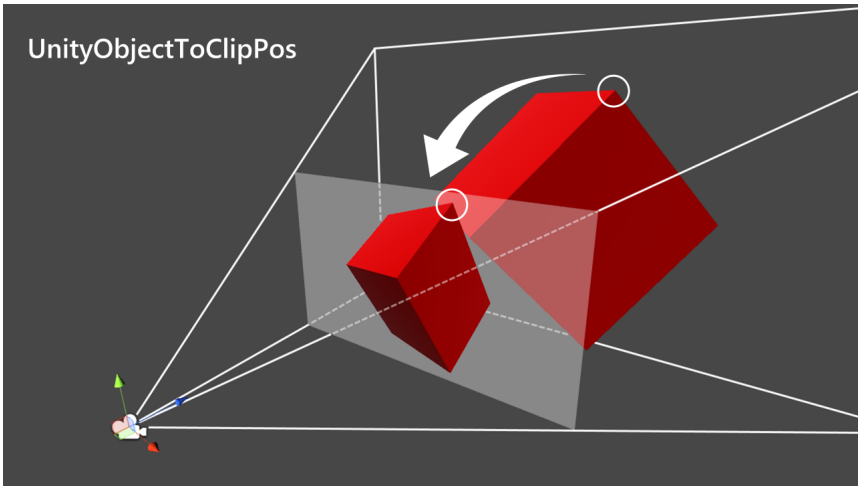
2.2 SubShader / Pass 構文

SubShader と Pass については、必要なときに改めて解説します。今の時点では、定型文であるという認識でよいです。

簡単に解説しておくと、1 つのシェーダプログラムの中には、あるオブジェクトを描画するための方法を複数定義することができ、その 1 つの方法を SubShader として定義します。ここではもっとも簡単なシェーダについて解説していますから、1 つの方法しか定義されていません。

また、あるオブジェクトを描画するために複数回の工程を要する場合があります。その工程の 1 つを Pass として定義します。ここではもっとも簡単なシェーダについて解説していますから、1 つの工程しか定義されていません。

2.6.3 座標空間の変換



▲図 2.3 UnityObjectToClipPos による頂点の処理のイメージ

ここまでの解説で、vert 関数は appdata に定義された vertex 変数から、オブジェクト空間にある頂点の座標情報を受け取れることが確認できました。

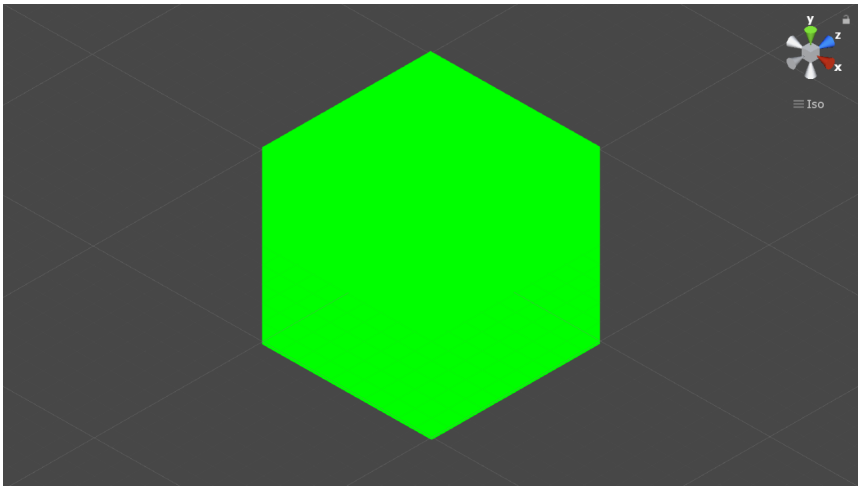
Vertex シェーダの役割を振り返りましょう。受け取った頂点データが、画面上のどの位置に映し出されるかを算出する必要があります。本来なら複雑な説明と処理が必要なのですが、Unity なら簡単です。Unity ではこの算出を UnityObjectToClipPos 関数で完了することができます。#include で参照した UnityCG.cginc ファイルにこの関数が含まれています。

なぜ UnityObjectToClipPos 関数で期待する結果を得られるのかをすべて解説すると、シェーダプログラミングの解説からやや外れるので、ここでは簡単に解説します。

オブジェクト空間にある 3 次元の座標が、あるカメラが映す 2 次元空間のどの位置に映し出されるかを算出するためには、射影 (投影) 変換します。カメラが映す 2 次元の空間を「ビュー空間」と言います。また、射影変換は行列を用いた計算です。

UnityObjectToClipPos 関数に与えたオブジェクト空間の座標は、クリップ (クリッピング) 空間の座標に変換され、ビュー空間の座標に (射影) 変換されます。"クリップ (クリッピング) する" とは、簡単にいえばカメラが映し出す範囲を決定することです。つまりクリップ空間とは、カメラが映し出す空間です。

2.8 オブジェクトの色を変える



▲図 2.4 オブジェクトの色を緑に変更した結果

用意したサンプルでは、オブジェクトを赤に塗るような Fragment シェーダを実装しています。これを赤以外の色にしてみましょう。もっとも簡単な方法は、`fixed4` を初期化するときに別の値を設定することです。

```
fixed4 color = fixed4(0, 1, 0, 1);
```

これだけで別の色に変えることができます。このときの色は、R = 0, G = 1, B = 0, A = 1 なので緑色です。

他の方法では、`color` の R, G, B 成分をそれぞれ変更します。次の例を見てください。出力される色は、R = 0.5, G = 0.5, B = 0, A = 1 で暗い黄になります。

```
fixed4 color = fixed4(0, 0, 0, 1);
color.r = 0.5;
color.g = 0.5;
```

`fixed4` や `float4` のような、複数の値から成り立つ構造体の各成分は、それぞれ、`.r`, `.g`, `.b`, `.a` のようにアクセスすることができます。あるいは、`.x`, `.y`, `.z`, `.w` と

してもアクセスすることができます。色情報としても、ベクトル情報としてもみなすことができる、ということです。

次のコードでは、出力される色は青になります。

```
fixed4 color = fixed4(0, 0, 0, 1);
color.z = 1;
```

一度に複数の成分にアクセスする方法もあります。次のコードでは、出力される色は水色になります。

```
fixed4 color = fixed4(0, 0, 0, 1);
color.gb = 1;
```

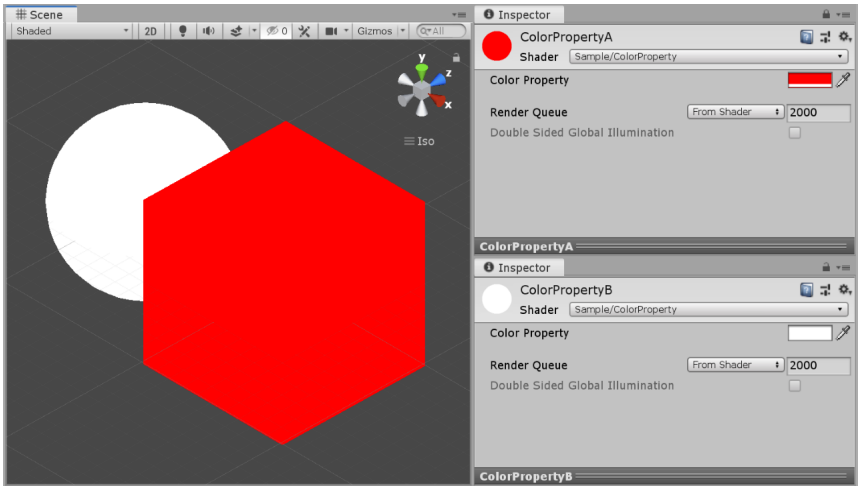
ここで透明度を表す A の値に 0.5 を設定して、結果を確認してみましょう。

```
fixed4 color = fixed4(1, 0, 0, 0.5);
```

半透明になるように思えますが、結果は半透明にならなかったと思います。これは、シェーダが半透明を描画するように設定されていないためです。言い換えれば、レンダリングパイプラインが、半透明な描画のための処理を実行していません。半透明を描画するときは、特別な設定が必要になります。これについては、別のサンプルで解説します。

第 3 章

Inspector からシェーダの値を変更する



▲図 3.1　Inspector から色を変更した結果

先の解説ではシェーダのソースコードを書き換えて色を変更しました。ところが、少しのパラメータを修正する度にソースコードを書き換えるのは非効率です。たとえば赤と緑のオブジェクトを描画するために、ほとんど変わらないソースコードを 2 つ用意することはよくありません。

この問題を解決するにはマテリアルを使います。ふつう、シェーダはマテリアルに設定して扱いますが、そのマテリアルにはシェーダに与えるパラメータを設定するこ

とができます。複数のマテリアルにそれぞれ異なるパラメータを設定すれば、それを参照する 1 つのシェーダから複数の異なる結果が得られますね。

ここではシェーダにパラメータを設定できるようにし、マテリアルの Inspector からそのパラメータを変更できるようにしてみましょう。

このサンプルは「03_Properties」フォルダに含まれています。サンプルでは異なる 2 つのマテリアルに同じシェーダを設定し、異なる色のパラメータを設定しました。それぞれのマテリアルが設定されたオブジェクトは、それぞれ異なる色で描画されていますが、マテリアルが異なるだけで、同じシェーダを使っていることになります。

3.1 Properties 構文

マテリアルの Inspector 上にパラメータを表示するときは `Properties` 構文を使います。先に解説したとおり、`Properties` は `CGPROGRAM` 構文の外にあるので、Unity 固有の言語 ShaderLab に定義される構文です。

ここでは Inspector からマテリアルの色を変更したいので、次のように実装しています。

▼ ColorProperty.shader

```
Properties
{
    _Color("Color Property", Color) = (1, 0, 0, 1)
}
SubShader
{
    Pass
    {
        CGPROGRAM
        …
        struct v2f
        {
            float4 vertex : SV_POSITION;
        };

        fixed4 _Color;
        …
```

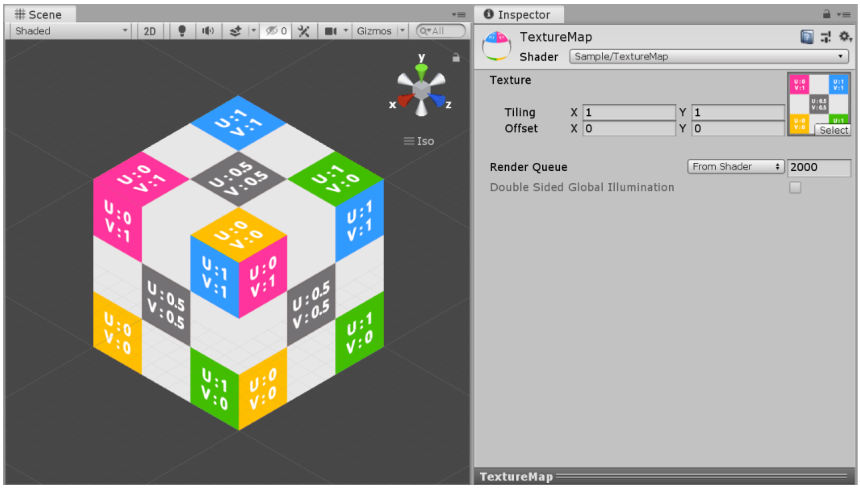
変数名("Inspector上の表記", 値の型) = 値の初期値 がこの構文の書式です。ここでは、変数名が `_Color`, Inspector 上の表記が `"Color Property"`, 値の型が `Color` で、初期値が `(1, 0, 0, 1)` です。

さらに、`CGPROGRAM` 内にも、`Properties` と同じ `_Color` が定義されていることに注目してください。`Properties` に定義され、Inspector 上から操作された値は、`CGPROGRAM` 内の対応する変数の値に代入されます。

このとき `CGPROGRAM` 内では型が `fixed4` であることにも注意が必要です。これは

第 4 章

テクスチャを設定して描画する



▲図 4.1　テクスチャを張り付けた結果

先の解説では Properties を使ってマテリアルにパラメータを設定できるようにしました。ここでは、同じようにしてマテリアルにテクスチャを設定できるようにしましょう。さらに、テクスチャを使ってオブジェクトを描画できるようにします。

このサンプルは「04_TextureMap」フォルダに含まれています。

4.1 Properties にテクスチャの項目を追加する

まずは、マテリアルの Inspector からテクスチャを設定できるようにします。Properties にテクスチャの項目を追加するときは、2D 型を使います。他にも、応用的なレンダリングのための異なる型がありますが、ここでは解説しません。

▼ TextureMap.shader

```
Properties
{
    _MainTex("Texture", 2D) = "white" {}
}
SubShader
{
    Pass
    {
        CGPROGRAM
        …
        sampler2D _MainTex;
        …
```

初期値にはいくつかの規定値が使用できます。初期値は、テクスチャが設定されないとき代わりに得られる色です。たとえば white が設定されるとき、もしテクスチャが設定されなければ、そのテクスチャは常に (1, 1, 1, 1) の色を示します。

white 以外には、black, gray, bump, red が使えます。あるいは単に "" として値を与えないことができます。値を与えないとき、gray に等しくなります。

後に続く {} については、このように記述するものとして覚えておきましょう。執筆時時点では Unity で活用されていません。

変数名の _MainTex は Unity で標準的に定義される変数名です。変数名は任意に変更することができますが、先に解説した他の Properties の定義と同様に、CGPROGRAM に対応する変数が定義されていることを確認してください。このとき CGPROGRAM に定義する変数の型は sampler2D です。

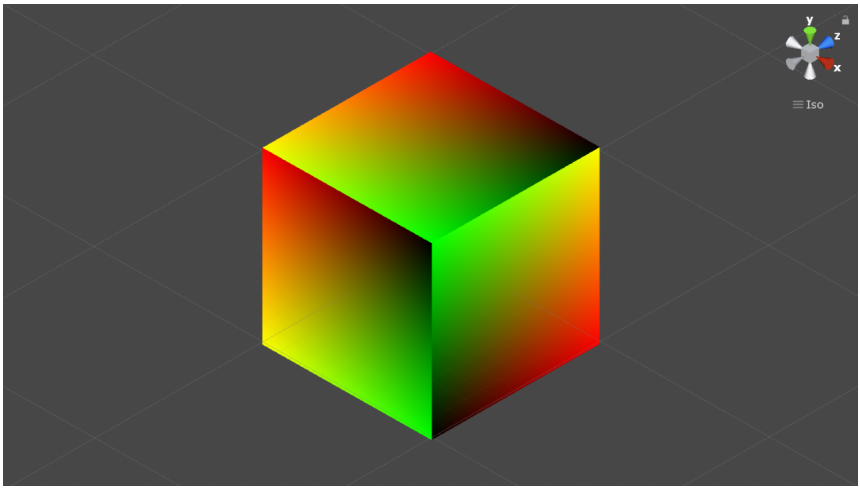
4.1.1 sampler2D 型

sampler2D 型はテクスチャへの参照そのものが代入される型です。Unity のスクリプト側のライブラリに定義されるように、型名が Texture や Texture2D ではありませんが、テクスチャそのものを表す型と覚えてもよいでしょう。

ふつう、sampler2D 型は単体では使わず、tex2D 関数と合わせて使います。tex2D 関数はテクスチャの特定の一部分を参照するための関数です。使い方は、後に続いて解説します。

第 5 章

Vertex, Fragment シェーダの再確認



▲図 5.1　Fragment シェーダで可視化した UV 座標

Vertex シェーダと Fragment シェーダについて、ここで改めて確認しておきましょう。

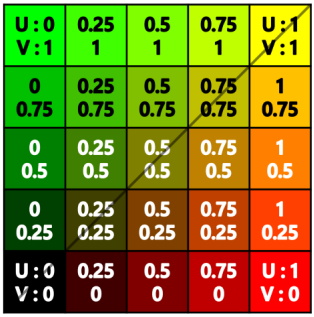
1. 頂点の映る位置を算出する (Vertex Shader)
2. ポリゴンの表裏を確認する (Culling)
3. 描画する画素を決定する (Rasterize)

4. Z Buffer の値を比較する (Z Test)
5. 画素を描画する (Fragment Shader)
6. Z Buffer の値を更新する (Z Write)

Vertex シェーダは、入力されたポリゴンの頂点毎に呼び出され、それが画面上に映る位置を算出します。画面上に映されたポリゴンは、ラスタライズによって複数の画素で表されます。Fragment シェーダはその画素毎に呼び出されて、そこに描画する色を算出します。

このサンプルは「05_VertexFragment」フォルダに含まれています。サンプルでは、Vertex シェーダが頂点毎に呼び出されていること、Fragment シェーダが画素毎に呼び出されていることを確認します。

5.1 Fragment シェーダによる UV 座標の可視化



▲図 5.2　UV 座標の補間と色の相関

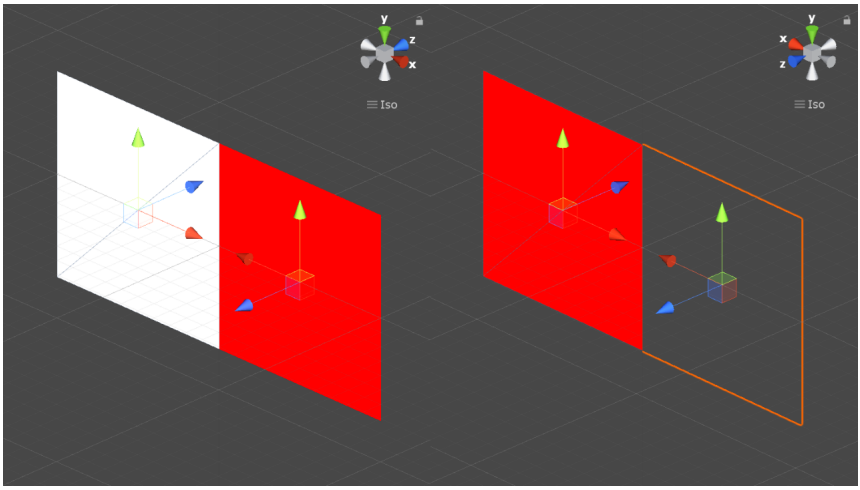
まずは、Fragment シェーダを使って、UV 座標の値を色情報として可視化してみます。ラスタライズ処理は、頂点に与えられた値を線形補間し、Fragment シェーダは画素毎に呼び出されますから、UV 座標を可視化すれば、グラデーションした結果が得られるはずです。

▼ FragmentShader.shader

```
fixed4 frag(v2f i) : SV_Target
{
    fixed4 color = fixed4(i.uv.x, i.uv.y, 0, 1);
    return color;
```

第 6 章

カリングを制御して背面を描画する



▲図 6.1　(赤) 表裏のどちらも描画されている (白) 背面だけが描画されている

ここでは「カリング (Cull・ing)」の設定について解説します。まずはレンダリングパイプラインを振り返って確認しましょう。

1. 頂点の映る位置を算出する (Vertex Shader)
2. ポリゴンの表裏を確認する (Culling)
3. 描画する画素を決定する (Rasterize)

4. Z Buffer の値を比較する (Z Test)
5. 画素を描画する (Fragment Shader)
6. Z Buffer の値を更新する (Z Write)

カリングは Vertex シェーダの後に実行されます。画面上に映されたポリゴンの表裏を確認して、ふつう、ポリゴンが裏側を向くときには描画しないようにする処理です。

たとえばキューブのように閉じられたオブジェクトを描画するとき、カメラに見えていない裏側を向いたポリゴンを描画することは非効率ですね。

ただし、カリングの設定を変更することによって、ポリゴンの裏面だけを描画させたり、あるいは、表裏の両方を描画させることもできます。

ポリゴンの表裏が描画できると、プレーンメッシュで十分に表現可能なオブジェクト、たとえば葉や草、髪や紙などを表現しやすくなります。あるいは、たとえばキューブの内側（裏面）だけを描画するように設定して、その中にカメラを置けば、部屋の中を映すような表現ができます。

このサンプルは「06_Culling」フォルダに含まれています。サンプルには板状のクアッドを用意しました。標準的な設定のままではその裏面が描画されませんが、カリングの設定を変更し、裏面や両面を描画できるようにします。

6.1 Cull 構文

カリングはの設定は `Cull` 構文によって制御します。

`Cull Back` ならポリゴンの裏面を、`Cull Front` ならポリゴンの表面を描画しません（カリングします）。また、`Cull Off` はポリゴンの両面を描画します（カリングしません）。既定値は `Cull Back` で、ふつう省略されます。

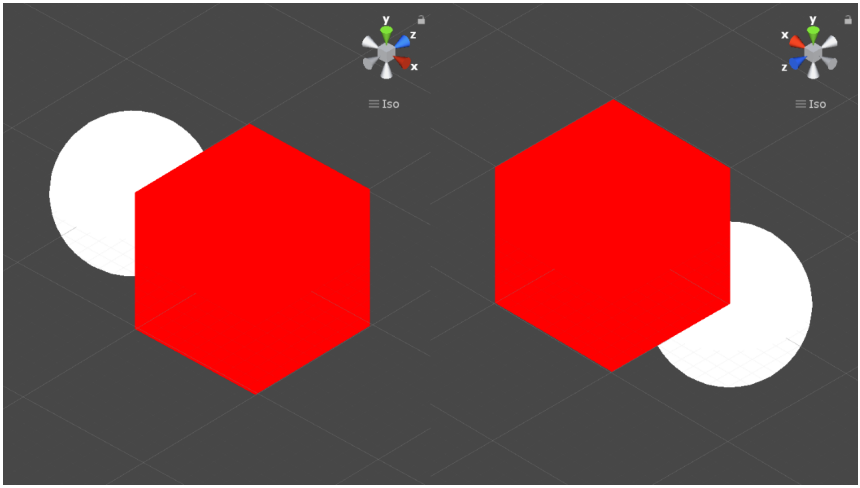
▼ Culling.shader

```
SubShader
{
    Cull Off

    Pass
    {
        CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag
    …
```

`Cull` の値を切り替えながら、描画結果がどのように変化するかを確認してみましょう。

第 7 章

Z Test と Offset によって描画順を制御する



▲図 7.1　回り込んでも常にキューブが手前に描画される

ここでは「Z Test」の設定について解説します。まずはレンダリングパイプラインを振り返って確認しましょう。

1. 頂点の映る位置を算出する (Vertex Shader)
2. ポリゴンの表裏を確認する (Culling)
3. 描画する画素を決定する (Rasterize)
4. Z Buffer の値を比較する (Z Test)

5. 画素を描画する (Fragment Shader)
6. Z Buffer の値を更新する (Z Write)

Z Test はラスタライズされた画素毎に実行されます。Z Buffer の既存の Z 値を参照して、現在の Z 値の方が小さいときだけ成功します。Z Test が成功するとき、Fragment シェーダはその画素を描画します。

このとき、Z Test の設定を変更することによって、Z 値の方が大きいときだけ成功させたり、あるいは、常に Z Test に成功させて、絶対に描画処理を実行させることなどができます。

このサンプルは「07_ZTest」フォルダに含まれています。

7.1 ZTest 構文

Z Test の設定は、`ZTest` 構文によって制御します。既定値は `ZTest LEqual` で、ふつう省略されます。

サンプルでは `ZTest Always` を設定して、常に Z Test に成功するようにしています。

つまり、このシェーダによってオブジェクト (のポリゴン) が描画されるときは、既存のオブジェクトの描画を必ず上書きして描画することになります。サンプルではキューブです。

▼ ZTest.shader

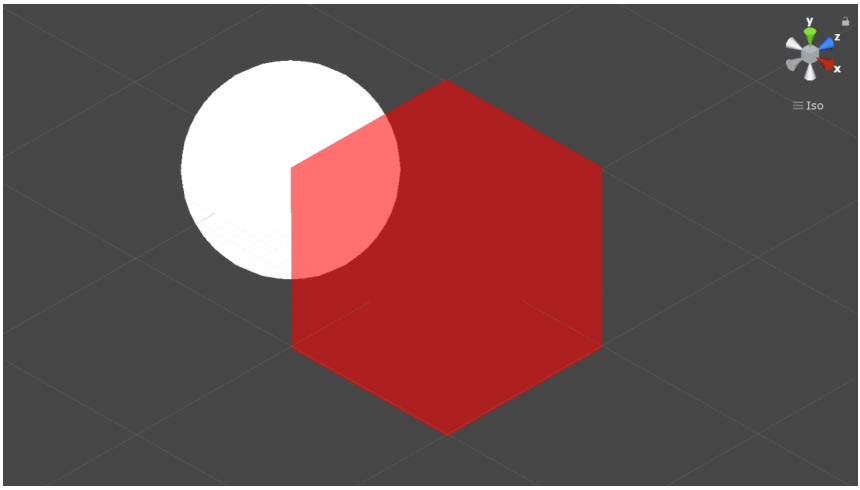
```
SubShader
{
    ZTest Always

    Pass
    {
        CGPROGRAM

        #pragma vertex vert
        #pragma fragment frag
...
```

他に `ZTest` に設定可能な値と効果は次のとおりです。

実際のところ、Z Test を操作する機会は極めて限定的なように思います。絶対的に手前に描画したい UI のために使うことなども考えられますが、UI などを描画するにしても、描画順を制御するか、あるいは配置する位置で制御されるべきです。

第 8 章

Blending と描画順を制御して半透明を表現する



▲図 8.1　半透明のキューブと不透明なスフィア

半透明なオブジェクトを描画する方法について解説します。先の解説では Fragment シェーダが出力する色の A の値を小さくしても、半透明にはなりませんでした。半透明なオブジェクトを描画するためには、(1) オブジェクトの描画順を制御し、(2) 半透明に見えるように色を合成する必要があります。

このサンプルは「08_Blending」フォルダに含まれています。

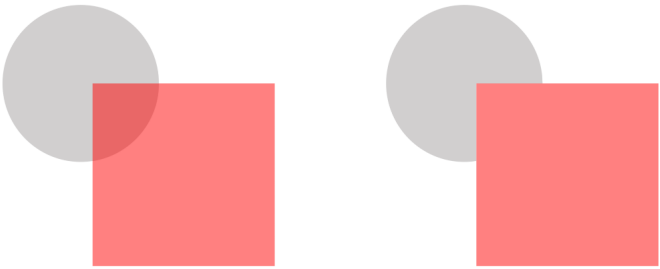
8.1 レンダーキューによる描画順の制御

半透明なオブジェクトを表現するためには、その向こう側に、別のオブジェクトや背景が描画されている必要があります。レンダリングパイプラインを振り返ってみましょう。

1. 頂点の映る位置を算出する (Vertex Shader)
2. ポリゴンの表裏を確認する (Culling)
3. 描画する画素を決定する (Rasterize)
4. Z Buffer の値を比較する (Z Test)
5. 画素を描画する (Fragment Shader)
6. Z Buffer の値を更新する (Z Write)

レンダリングパイプラインがオブジェクトを描画するとき、Z Test によって、すでに描かれているオブジェクトよりも手前にあるオブジェクトだけが上書きして描画されます。

つまり、半透明なオブジェクトを描画するためには、それよりも後ろにあるオブジェクトをあらかじめ描画しておく必要がある、ということです。言い換えれば、描画順の制御が必要になります。



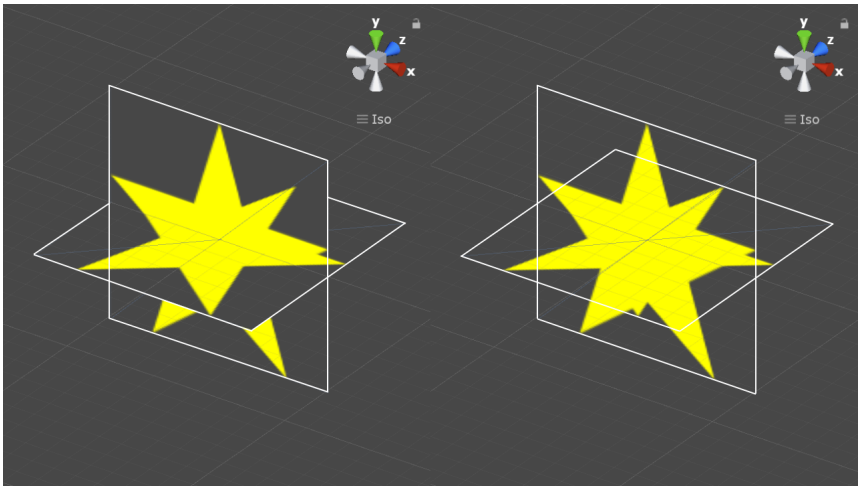
▲図 8.2　(左) 適切な順に描画した例 (右) 順を誤って描画した例

もし描画順を制御しないと、どうなるでしょう。図は、キューブが手前、スフィアが奥にある状態で、キューブを半透明とします。このとき、左はスフィアから描画した場合、右はキューブから描画した場合です。

もし図の右のように、半透明なキューブを先に描画すると、透過した先にあるスフィアがまだ描画されていませんから、背景の色を透過して映すことになります。続

第 9 章

Z Write を制御して部分的な透過を表現する



▲図 9.1　(左) Z Write On (右) Z Write Off

先の解説では半透明なオブジェクトの描画方法について解説しました。それは部分的に透明なオブジェクトはどうでしょうか。

図は 2 つの直交するクアッドに対し、部分的に透過するテクスチャを貼った様子です。図の左に注目してください。水平のクアッドの、透過されるべき部分が透過して見えません。レンダーキューによる描画順の設定もブレンディングの設定も適切に行っているのに、なぜでしょうか。

この問題は描画順に起因します。水平のクアッドの Z 値が書き込まれると、それに遮蔽される垂直のクアッドの一部は描画されません。Z Test に失敗するためです。では、レンダーキューによる描画順の設定が正しく処理されていないのでしょうか。そうではありません。この問題は、垂直のクアッドを先に描画しても同じように起こります。

もしも上手く理解できないときは、レンダリングパイプラインを振り返って確認しましょう。

1. 頂点の映る位置を算出する (Vertex Shader)
2. ポリゴンの表裏を確認する (Culling)
3. 描画する画素を決定する (Rasterize)
4. Z Buffer の値を比較する (Z Test)
5. 画素を描画する (Fragment Shader)
6. 描画する色を合成する (Blending)
7. Z Buffer の値を更新する (Z Write)

問題を解決する方法はいくつかありますが、ここでは Z Write の処理を設定によって無効化し、Z Buffer を更新しないことで解決します。つまり、同じ画素に対して後から描画されるオブジェクトは、Z Test に必ず成功します。

このサンプルは「09_ZWrite」フォルダに含まれています。

9.1 ZWrite 構文

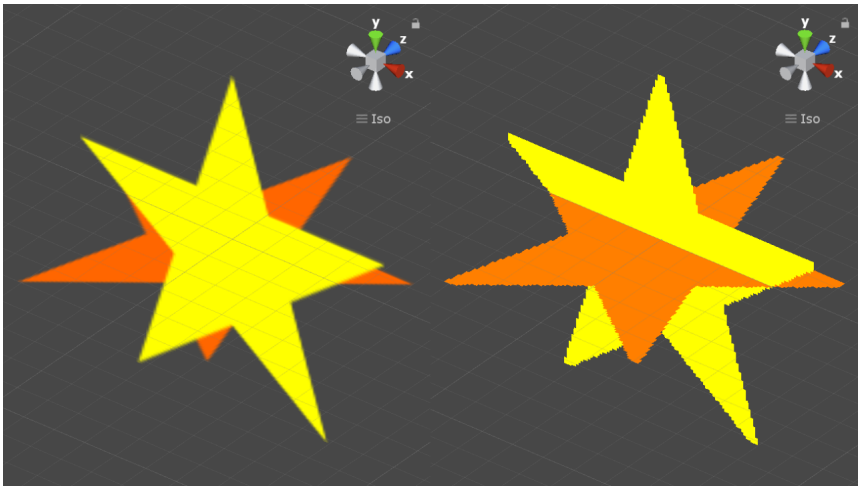
Z Write の設定は `ZWrite` 構文によって制御します。Z 値の書き込みを有効にするときは `ZWrite On`、無効にするときは `ZWrite Off` とします。既定値は `ZWrite On` で、ふつう省略されます。

▼ ZWrite.shader

```
SubShader
{
    Tags
    {
        "Queue" = "Transparent"
        "RenderType" = "Transparent"
    }

    Blend SrcAlpha OneMinusSrcAlpha
    Cull Off
    ZWrite Off
…
```

第 10 章

Alpha Test によって部分的な透過を表現する



▲図 10.1　(左) Z Write による部分透過 (右) Alpha Test による部分透過

先の解説では Z Write を制御することによって部分的に透明なオブジェクトを描画しました。ところが Z Write を無効化することによって実現したため、メッシュ間の前後関係が破綻する問題も同時に起こりました。

この問題を解決するには、Z Write を有効にしつつ、Fragment シェーダでは透明な部分を描画しない、といった処理が必要です。

透明な部分を描画しないようにするためには「Alpha Test」と呼ばれる処理が必要

になります。Alpha Test は、Fragment シェーダから算出された色が透過である場合に、塗りつぶし処理を破棄します。

1. 頂点の映る位置を算出する (Vertex Shader)
2. ポリゴンの表裏を確認する (Culling)
3. 描画する画素を決定する (Rasterize)
4. Z Buffer の値を比較する (Z Test)
5. 画素を描画する (Fragment Shader)
6. 透過なとき処理を破棄する（AlphaTest）
7. 描画する色を合成する (Blending)
8. Z Buffer の値を更新する (Z Write)

塗りつぶし処理を破棄する、塗りつぶさないということは、そこには何も存在しなかったことになりますから、Z Buffer はそのままであるべきです。したがって、Z Write による Z Buffer の更新も行われません。同じように、Blending も不要です。

Alpha Test は、かつては Z Test と同じように扱われていました。つまり、レンダリングパイプラインが提供する処理の 1 つであり、その制御は用意されたいくつかの方法から選択することのみができました。ところが、昨今では Fragment シェーダが Alpha Test の処理までを包括するのが主流です。

ShaderLab には Alpha Test を制御するための `AlphaTest` 構文が残されていますが、ここでは `AlphaTest` 構文を使わず、Fragment Shader で Alpha Test を行います。

このサンプルは「10_AlphaTest」フォルダに含まれています。

10.1 Alpha Test 用の Render Queue

部分的に透過するオブジェクトが存在するということは、その先にあるオブジェクトが見えている必要があります。半透明なオブジェクトの描画を実現したときと同様に、できるかぎり他のオブジェクトを先に描画しておく必要があります。描画順の制御は先に解説したとおり `Queue` タグによって行います。

ShaderLab は標準で `AlphaTest` タグを用意しています。`AlphaTest` タグの値は 2,450 です。一般的なオブジェクトを描画するための `Geometry` 2,000 よりも大きな値ですから、部分的に透過しているオブジェクトは、それらよりも後に描画されるようになります。

おわりに

要望、感想、誤字脱字などのご指摘、その他のお問い合わせについては専用フォームでお伺いします。折り返しのご連絡等については必ずのお約束をしかねる体制であることご了承ください。

【問い合わせフォーム】
https://goo.gl/forms/mOiWo6JuPz848deh1

著者紹介

Twitter : @XJINE
https://twitter.com/XJINE

著作権情報

Unity Shader Programming Vol.01

2018 年 7 月 11 日　v0.9.9
2018 年 7 月 17 日　v1.0.0
2018 年 8 月 18 日　v1.1.1
2019 年 5 月 20 日　v1.1.2
2019 年 9 月 21 日　v2.0.0
2020 年 4 月 10 日　v2.2.1
著　者　XJINE
編　集　XJINE
発行所　XJINE