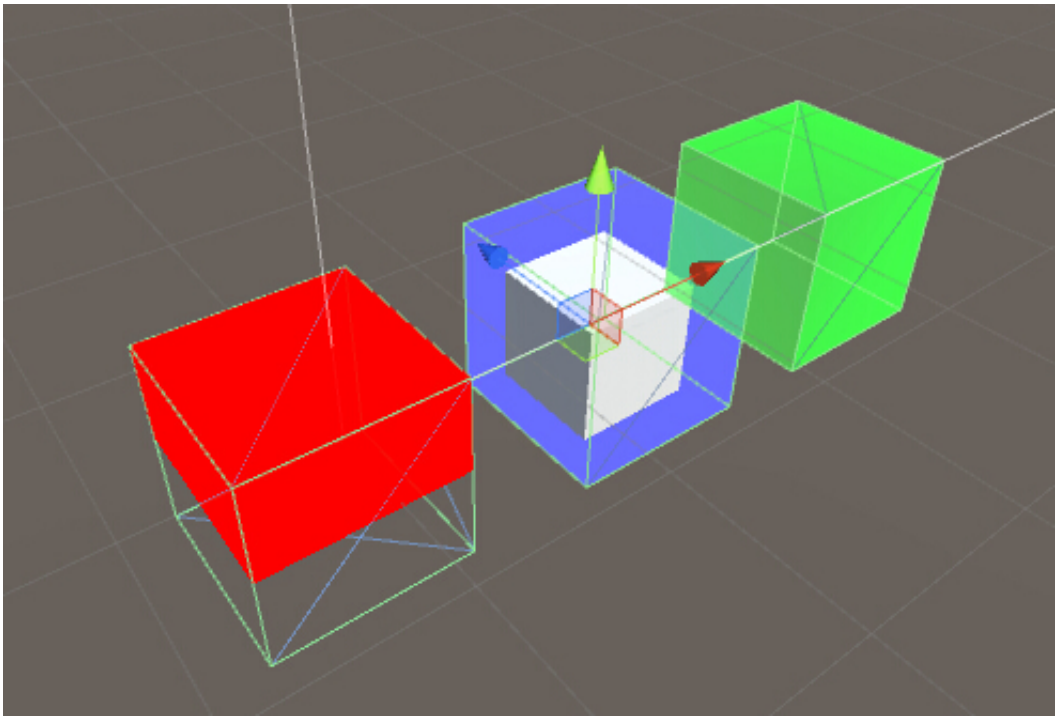
UnityShader快速上手指南（三） - 玄雨

简介

这一篇还是一些基本的shader操作：裁剪、透明和法向量的应用
（纠结了很久写不写这些，因为代码很简单，主要是些概念上的东西）
先来看下大概的效果图：（从左到右依次是裁剪，透明，加了法向量的透明）
(好奇怪，为啥我字那么多，提示我少于150字)



裁剪

代码

```
Shader "LT/Lesson3_Cull"
{
    Properties
    {
        _Color ("Color", Color) = (1, 1, 1, 1)
    }
    SubShader
    {
        Pass
        {
            Cull Off
            CGPROGRAM
```

```
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            uniform float4 _Color;

            appdata_base vert ( appdata_base input)
            {
                input.texcoord = input.vertex ;
                input.vertex  = mul(UNITY_MATRIX_MVP, input.vertex );
                return input;
            }

            fixed4 frag (appdata_base input) : COLOR
            {
                if(input.texcoord.y < 0) {
                    discard;
                }
                return _Color;
            }
            ENDCG
        }
    }
}
```

代码的基本格式我就不介绍了，不熟的请看教程一

这里引入了一个新的东西: Cull Off

裁剪简单的可以分为三种裁剪：擦除裁剪（discard），正面裁剪（Front）和背面裁剪（Back）
这里的Cull Off代表关闭unity自带的背面裁剪（因为如果不写的话默认为Cull Back）

```
Cull Off 关闭裁剪
Cull Back 背面裁剪（默认）
Cull Front 正面裁剪
```

然后关于这个面的正反的界定，是通过法线来完成的，法线向量>0则为正
Cull Back 和 Cull Front Unity替我们完成了，如果需要简单的裁掉正面和反面的话直接使用预制的就行了
（Q：如果我想同时裁正面和反面怎么处理？A:如果你要同时裁掉正面和反面，你的模型就不用显示了好嘛）
这里我想通过模型的坐标位置来裁面，所以关闭了Unity的自动裁剪
然后就是把顶点信息缓存下来:

```
input.texcoord = input.vertex ;
```

为了方便（同时也能提高性能，由于GPU是频繁调用这些函数，所以不管什么空间啊，计算啊都能省就省），我直接将数据存在了input对象中，然后又将input返回给frag使用（这里随便选一个不用的字段用来缓存我们的位置信息就行了，注意vertex是要使用的，所以vertex不能用来缓存这个信息）
然后在frag中判断我们缓存的坐标信息，满足条件就discard掉

```
                if(input.texcoord.y < 0) {
                    discard;
                }
```

这里的discard相当于强行中断该次渲染，也可以说成取消渲染，这样就完成了我们的裁剪功能了
顺带一提，discard很消耗性能的，所以能不用还是就不用了（何况这里还多了一步if判断语句），需要指定裁剪还是改模型来的高效，当然如果需要动态裁剪就需要这样的代码了

透明

还是先来代码

代码

```
Shader "LT/Lesson3_Transparent"
{
    Properties
    {
         _Color ("Color", Color) = (1, 1, 1, 0.5)
    }
    SubShader
    {
        Tags { "Queue" = "Transparent" }
        Pass
        {
            ZWrite Off
            Blend One One
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            uniform float4 _Color;

            appdata_base vert ( appdata_base input)
```

```
            {
                input.vertex  = mul(UNITY_MATRIX_MVP, input.vertex );
                return input;
            }

            fixed4 frag (appdata_base input) : COLOR
            {
                return _Color;
            }
            ENDCG
        }
    }
}
```

恩...这里呢代码的内容比较少，就是直接把外面设置的颜色应用到物体上就完了
但是多了

```
Tags { "Queue" = "Transparent" }
ZWrite Off
Blend One One
```

这三个东西
Tags { "Queue" = "Transparent" }
先来第一个Tags { "Queue" = "Transparent" }
大家可能看到有个Transparent会想要写透明是不是必须要这句话，其实不然，这行代码只是指定一个渲染顺序而已，一般情况下是具体情况具体分析，比如你希望这个shader是渲染背景物体的时候，你可以设置为
"Queue" = "Background"，需要时物体的时候设置为“Geometry”,其实对应下来的就是一个数值，这个数值越小，越先渲染，然后从我们的渲染顺序可以理解，先渲染的自然在背后了
下面列举下常用的值对应的名字

```
    "Background"。值为1000。比如用于天空盒。
    "Geometry"。值为2000。大部分物体在这个队列。不透明的物体也在这里。这个队列内部的物体的渲染顺序会有进一步的优化（应该是从近到远，early-z test可以剔除不需经过FS处理的片元）。其他队列的物体都是按空间位置的从远到近进行渲染。
    "AlphaTest"。值为2450。已进行AlphaTest的物体在这个队列。
    "Transparent"。值为3000。透明物体。
    "Overlay"。值为4000。比如镜头光晕。
```

还有就是，用户可以定义任意值，比如"Queue"="Geometry+10"

ZWrite Off
再来ZWriter Off，这句命令表示不写入深度缓存

呃~~~，然后我们继续来科普概念吧（由于语文不好，后面一大段话是从别的网站复制的）

（1）什么是深度？
深度其实就是该像素点在3d世界中距离摄象机的距离，深度值（Z值）越大，则离摄像机越远。

（2）什么是深度缓存？
深度缓存中存储着每个像素点（绘制在屏幕上的）的深度值！如果启用了深度缓冲区，在绘制每个像素之前，OpenGL会把它的深度值和已经存储在这个像素的深度值进行比较。新像素深度值<原先像素深度值，则新像素值会取代原先的；反之，新像素值被遮挡，其颜色值和深度将被丢弃，最终屏幕显示的就是深度缓存中深度对应的像素点的颜色！(深度主要起的是比较的作用)

（3）什么是深度测试？
在深度测试中，默认情况是将要绘制的新像素的z值与深度缓冲区中对应位置的z值进行比较，如果比深度缓存中的值小，那么用新像素的颜色值更新深度缓存中对应像素的颜色值。

（4）为什么需要深度？
在不使用深度测试的时候，如果我们先绘制一个距离较近的物体，再绘制距离较远的物体，则距离远的物体因为后绘制，会把距离近的物体覆盖掉，这样的效果并不是我们所希望的。而有了深度缓冲以后，绘制物体的顺序就不那么重要了，都能按照远近（Z值）正常显示，这很关键。(越后绘制的东西，距离相机就越近)

那么，在unity中，如果知道了渲染队列，ZWrite，ZTest，如何确定哪个物体先显示呢？
首先，unity先将渲染队列中较前的进行渲染，然后再执行ZWrite，ZTest
ZWrite可以取的值为：On/Off，默认值为On，代表是否要将像素的深度写入深度缓存中
ZTest可以取的值为：Greater/GEqual/Less/LEqual/Equal/NotEqual/Always/Never/Off，默认值为LEqual，代表如何将像素的颜色写入深度缓存中，例如当取默认值的情况下，如果将要绘制的新像素的z值小于等于深度缓存中的值，则将用新像素的颜色值更新深度缓存中对应像素的颜色值。需要注意的是，当ZTest取值为Off时，表示的是关闭深度测试，等价于取值为Always，而不是Never！Always指的是直接将当前像素颜色(不是深度)写进颜色缓冲区中；而Never指的是不要将当前像素颜色写进颜色缓冲区中，相当于消失。

因为ZWrite默认值为On，ZTest默认值为LEqual，所以这很好地解释了为什么在unity中，距离相机近的东西会阻挡住距离相机远的东西。如果我们先绘制一个距离较近的物体，再绘制距离较远的物体，则距离远的物体因为后绘制，会把距离近的物体覆盖掉，这时我们可以通过修改ZWrite和ZTest来改变物体的遮挡关系！

恩，好，介绍完了深度这个玩意儿咯， 那来解释下我们为啥要关掉了（其实不关掉也可以，但是关掉GPU可以少做一步操作啊，提高性能性能性能性能性能性能，所以能关就关吧），因为我们要写的是透明啊，不管先后顺序的，透明都可以看得到 - -

Blend One One
最后是Blend One One，这是个大概念，这个命令是写透明shader所必须的，因为它定义了透明的模式
然而，这玩意儿很简单的，命令是

```
Blend SrcFactor DstFactor
```

然后这个Factor 支持:

```
One
值为1，使用此设置来让源或是目标颜色完全的通过。
Zero
值为0，使用此设置来删除源或目标值。
SrcColor
此阶段的值是乘以源颜色的值。
SrcAlpha
此阶段的值是乘以源alpha的值。
DstColor
此阶段的值是乘以帧缓冲区源颜色的值。
DstAlpha
此阶段的值是乘以帧缓冲区源alpha的值。
OneMinusSrcColor
此阶段的值是乘以(1 - source color)
OneMinusSrcAlpha
此阶段的值是乘以(1 - source alpha)
OneMinusDstColor
此阶段的值是乘以(1 - destination color)
OneMinusDstAlpha
此阶段的值是乘以(1 - destination alpha)
```

然后下面是常用的搭配:

```
Blend SrcAlpha OneMinusSrcAlpha // Alpha blending alpha混合
Blend One One // Additive 相加混合
Blend One OneMinusDstColor // Soft Additive 柔和相加混合
Blend DstColor Zero // Multiplicative 相乘混合
Blend DstColor SrcColor // 2x Multiplicative 2倍相乘混合
```

具体效果，大家可以自己改代码看结果，这里就不多说了

带法向量计算的透明

代码

```
Shader "LT/Lesson3_Silhouette"
{
    Properties
```

```
    {
         _Color ("Color", Color) = (1, 1, 1, 0.5)
    }
    SubShader
    {
        Pass
        {
            ZWrite Off
            Blend SrcAlpha OneMinusSrcAlpha
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            uniform float4 _Color;

            appdata_base vert ( appdata_base input)
            {
                fixed3 tempNormal = normalize(mul(fixed4(input.normal, 0.0),
_World2Object).xyz);
                fixed3 tempViewDir = normalize(_WorldSpaceCameraPos - mul(_Object2World,
input.vertex).xyz);
                input.vertex  = mul(UNITY_MATRIX_MVP, input.vertex );
                input.normal.x = min(1.0, _Color.a / abs(dot(tempNormal, tempViewDir)));
                return input;
            }

            fixed4 frag (appdata_base input) : COLOR
            {
                return float4(float3(_Color.x,_Color.y,_Color.z), input.normal.x);
            }
            ENDCG
        }
    }
}
```

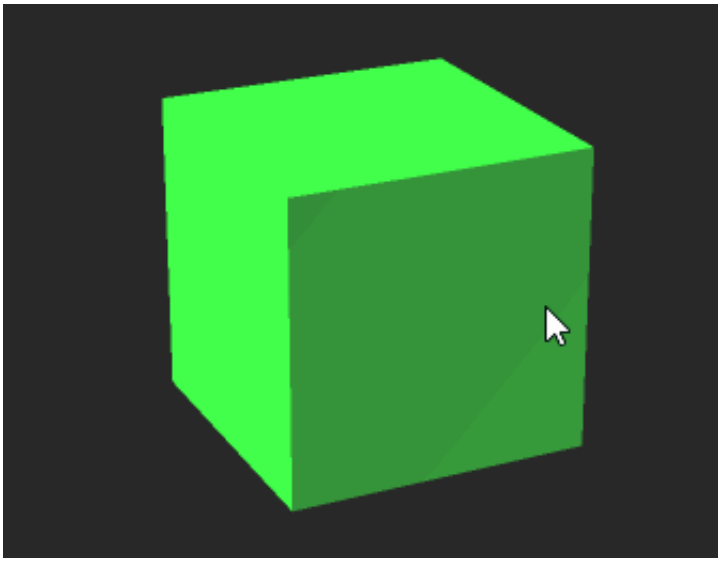
这里没啥新东西，但是代码上把法向量用起来了（原理主要是通过法向量和摄像机朝向算出一个新的透明度用来替换而已）
解释一下吧:

```
    fixed3 tempNormal = normalize(mul(fixed4(input.normal, 0.0), _World2Object).xyz);
    // 计算unity坐标系下的法向量
    fixed3 tempViewDir = normalize(_WorldSpaceCameraPos - mul(_Object2World,
input.vertex).xyz);
```

```
    // 计算unity坐标系啊的摄像机的向量
    input.normal.x = min(1.0, _Color.a / abs(dot(tempNormal, tempViewDir)));
    // 将两个向量点乘然后换算给设定颜色的alpha通道，并缓存起来
    // 至于为啥要点乘，然后用Color.a来除，这个是算法问题啦:
    // a = min(1, a/ |V·N|)，公式是书上来的，数学问题了，不做赘述

    return float4(float3(_Color.x,_Color.y,_Color.z), input.normal.x);
    // 使用设置的颜色的rgb值和算出来的新的alpha值
```

由于个人很喜欢这个效果，所以再来次效果展示



这样是不是有一点简单的立体效果咯捏（这不是光照，真的光照的反射啥的下一篇教程讲）

总结

本篇教程的东西都很基础，但是在以后写shader的过程中会用的很平凡，所以还是单独拿出来讲了一下，与我们的快速上手指南其实有点背道而驰 o(╯□╰)o ，但是为了后面不会看不懂，大家还是多自己写写熟悉一下，比如使用discard和Cull配合两个Pass写一个正反面渲染不同颜色的shader，利用前面讲的SinTime和法向量啥的写一个动态变化效果
还是那句话，又不懂得，QQ:821580467