

## 【翻译】第六章节：透明度（关于混合）

2014-11-27 09:26:00   2484 人阅读   [Unity3D](#)   [cg](#)   [透明度](#)

A<sup>-</sup>   A<sup>+</sup>

文章内容	例子源码	网友评论	最后编辑：2014-12-23 08:03:48
本文永久地址： <a href="http://www.omuying.com/article/94.aspx">http://www.omuying.com/article/94.aspx</a> ，【 <a href="#">文章转载请注明出处！</a> 】			

原文链接：[http://en.wikibooks.org/wiki/Cg\\_Programming/Unity/Transparency](http://en.wikibooks.org/wiki/Cg_Programming/Unity/Transparency)

本教程介绍着色器的片段 blending。本篇教程假定你熟悉《[剖面模型](#)》章节。

具体的说，本篇文章关于渲染透明对象，例如：透明的玻璃、塑料、织物等等（实际上是半透明的对象，他们不一定完全透明）。透过（半）透明的对象我们可以看到他们身后的颜色，其时是它们的颜色与它们身后的颜色进行了混合。

### 混合

在《[Programmable Graphics Pipeline](#)》章节中提到，片段着色器为每个片段（除非片段被擦除）计算 RGB 颜色值（在片段输出参数使用语义词 COLOR，包括红、绿、蓝、透明分量），片段的处理细节可以查看《[Per-Fragment Operations](#)》章节，其中一个操作是混合阶段，主要用来合并片段的颜色（由片段输出参数指定），这被称为“source color”，与之相应的是已存在帧缓冲区的像素颜色，这被成为“destination color”（因为帧缓冲区的颜色是颜色混合结果）。

混合是一个固定阶段，你可以通过混合公式对它进行配置但你无法对它编程，你可以通过公式像下面那样来混合 RGBA 颜色值：

```
1 float4 result = SrcFactor * fragment_output + DstFactor * pixel_color;
```

其中 fragment\_output 是通过片段着色器计算的 RGBA 颜色，pixel\_color 是当前帧缓冲区的颜色，result 是混合结果（混合输出阶段），SrcFactor 和 DstFactor 是可配置的 RGBA 颜色（类型是 float4），并且片段颜色和帧颜色的各个分量分别相乘，SrcFactor 和 DstFactor 的值在 Unity ShaderLab 中可以用下面语法指定：

```
1 Blend {code for SrcFactor} {code for DstFactor}
```

比较常用的混合因子（factor）可以查看下表，更详细的混合因子（factor）可以查看《[Unity's ShaderLab reference about blending](#)》：

Code	Resulting Factor (SrcFactor or DstFactor)
One	float4(1.0, 1.0, 1.0, 1.0)
Zero	float4(0.0, 0.0, 0.0, 0.0)
SrcColor	fragment_output
SrcAlpha	fragment_output.aaaa
DstColor	pixel_color
DstAlpha	pixel_color.aaaa
OneMinusSrcColor	float4(1.0, 1.0, 1.0, 1.0) - fragment_output
OneMinusSrcAlpha	float4(1.0, 1.0, 1.0, 1.0) - fragment_output.aaaa
OneMinusDstColor	float4(1.0, 1.0, 1.0, 1.0) - pixel_color
OneMinusDstAlpha	float4(1.0, 1.0, 1.0, 1.0) - pixel_color.aaaa



游戏开发学习



微型摄像机



比基尼除毛



伪装微型摄像机



无线监控摄像机



unity3d摄像机



室内设计师




unity3d移动

广告

### 最新文章

暂无图片

【原创】C# 基础之 Lambda表达式 - 907 次阅读

暂无图片


【原创】C#基础之 IEnumerable和 IEnumerator - 792 次阅读

暂无图片

【原创】C#基础之事件 - 886 次阅读

暂无图片

【原创】C#基础之委托 - 912 次阅读

暂无图片

【原创】C#基础之委托的使用 - 856 次阅读



游戏开发学习



伪装微型摄像机



unity3d移动



比基尼除毛



微型摄像机



无线监控摄像机



室内设计师



超高速摄像机

广告

### 随机阅读

正如在《[Vector and Matrix Operations](#)》章节中见到的那样，pixel\_color.aaaa 是 float4(pixel\_color.a, pixel\_color.a, pixel\_color.a, pixel\_color.a) 的简短写法，此外，请注意，所有颜色分量与混合因子 ( factor ) 值的范围都应限制在 0 至 1。

### alpha 混合

在 Unity 中，“alpha blending” 的混合方式可以像下面那样指定：

```
1 | Blend SrcAlpha OneMinusSrcAlpha
```

这对应于：

```
1 | float4 result = fragment_output.aaaa * fragment_output + (float4(1.0, 1.0, 1.0, 1.0) - fragment_output.aaaa) * pixel_color;
```

这儿使用 fragment\_output 的 alpha 分量表示不透明，片段输出颜色的不透明度越大，那么混合的片段输出颜色越多并且在帧缓冲区中的像素颜色越少，如果片段输出的颜色完全不透明，那么它将完全取代像素的颜色。

上面的混合公式有时候被称为 “over”，即 “fragment\_output over pixel\_color”，因为就像是把一层不透明的片段输出颜色放在像素颜色上（可以想像有一些不透明的颜色覆盖在彩色玻璃或者彩色的半透明塑料上）。

由于 alpha 混合的流行，即使没有采用 alpha 混合，颜色的 alpha 分量也通常被称为不透明。此外，请注意计算机图形学中透明度的常见计算公式为 1 - 不透明度。

### 预乘 alpha 混合

alpha 混合还有一个非常重要的公式，有时候片段输出颜色已经预乘过颜色分量的 alpha 分量，在这种情况下，alpha 不应该被再次相乘，正确的混合是：

```
1 | Blend One OneMinusSrcAlpha
```

这对应于：

```
1 | float4 result = float4(1.0, 1.0, 1.0, 1.0) * fragment_output + (float4(1.0, 1.0, 1.0, 1.0) - fragment_output.aaaa) * pixel_color;
```

### 附加 ( Additive ) 混合

下面是另一种混合方式：

```
1 | Blend One One
```

这对应于：

```
1 | float4 result = float4(1.0, 1.0, 1.0, 1.0) * fragment_output + float4(1.0, 1.0, 1.0, 1.0) * pixel_color;
```

这只是给帧缓冲区的颜色添加片段输出颜色，注意这儿 alpha 并没有被使用，尽管如此，这个混合方式在处理一些透明效果时非常有用，比如，它经常被用在粒子系统中处理火或者透明物体发出的光。

### 着色器代码

下面这个着色器例子中使用不透明度为 0.3 的绿色来呈现 alpha 混合：

```
01 | Shader "Cg shader using blending"
02 | {
03 |     SubShader
04 |     {
05 |         Tags { "Queue" = "Transparent" }
06 |         // draw after all opaque geometry has been drawn
07 |         Pass
08 |         {
```

 暂无图片	【翻译】第九章：漫反射（关于每顶点漫反射和多光源漫反射） - 1946 次阅读
 暂无图片	【原创】Shader 内置 Shader 之 Bumped Diffuse 学习 - 1707 次阅读
 暂无图片	【原创】Shader 内置 Shader 之 Normal-Diffuse 学习 - 1479 次阅读
 暂无图片	【翻译】第二十四章：表面反射（关于反射贴图） - 1441 次阅读
 暂无图片	【翻译】第二十五章：弧形玻璃（关于折射贴图） - 1722 次阅读



```
09     ZWrite Off // don't write to depth buffer
10     // in order not to occlude other objects
11     Blend SrcAlpha OneMinusSrcAlpha // use alpha blending
12
13     CGPROGRAM
14
15     #pragma vertex vert
16     #pragma fragment frag
17
18     float4 vert(float4 vertexPos : POSITION) : SV_POSITION
19     {
20         return mul(UNITY_MATRIX_MVP, vertexPos);
21     }
22
23     float4 frag(void) : COLOR
24     {
25         return float4(0.0, 1.0, 0.0, 0.3);
26         // the fourth component (alpha) is important:
27         // this is semitransparent green
28     }
29     ENDCG
30 }
31 }
32 }
```

把这个着色器应用到一个球体上，然后我们在球的背后添加一个立方体对象，我们可以通过球体看到后面的立方体对象，如图：



除了使用混合方式，我们还需要设置 Tags { "Queue" = "Transparent" } 和 ZWrite Off。

ZWrite Off 的作用是停用深度缓冲区写入，如《[Per-Fragment Operations](#)》章节所述，深度缓冲区保持最近深度的片段并且丢弃较大深度的片段，然而这在透明的情况下这并不是我们期望的，因为透明的片段不会阻塞其他片段，所以深度缓冲需要被停用。另请参阅《[Unity's ShaderLab reference about culling and depth testing](#)》

{ "Queue" = "Transparent" } 标签指定 subshader 在不透明网格之后再呈现透明网格，部分原因是我们停用了深度缓冲区的写入，这后果就是不透明的片段可以遮挡透明的片段，即使不透明的片段比较远，为了解决这个问题，我们首先在绘制透明网格之前先绘制不透明网格，通过为 subshader 指定的 { "Queue" = "Transparent" } 标签来为网格指定是否开启透明还是不透明。更多的 Unity ShaderLab 标签可以查看《[Unity's ShaderLab reference about subshader tags](#)》。

需要提及的是，使用渲染透明网格与停用深度缓冲区的策略并不一定就能解决所有的问题，如果片段混合的顺序不重要，那这种策略就比较完美，例如，如果仅仅是在帧缓冲区给像素颜色添加片段颜色，这儿片段的混合顺序并不重要，详情可以查看《[Order-Independent Transparency](#)》章节。然而对于其他的混合方式，例如 alpha 混合，结果是不同的，因为这取决于片段被混合的顺序（例如你从几乎不透明的绿色玻璃看几乎不透明的红色玻璃，你主要看到绿色，反过来，如果你从几乎不透明的红色玻璃看几乎不透明的绿色玻璃，你主要看到的是红色，同样的，混合几乎不透明的绿色与红色与混合几乎不透明的红色与绿色的效果不同）。为了避免伪影（artifacts），因此最好使用 additive blending 或者（premultiplied）alpha blending 来与小不透明体（small opacities）混合。（在这种情况下，DstFactor 因子接近 1，因此 alpha blending 接近 additive blending）。

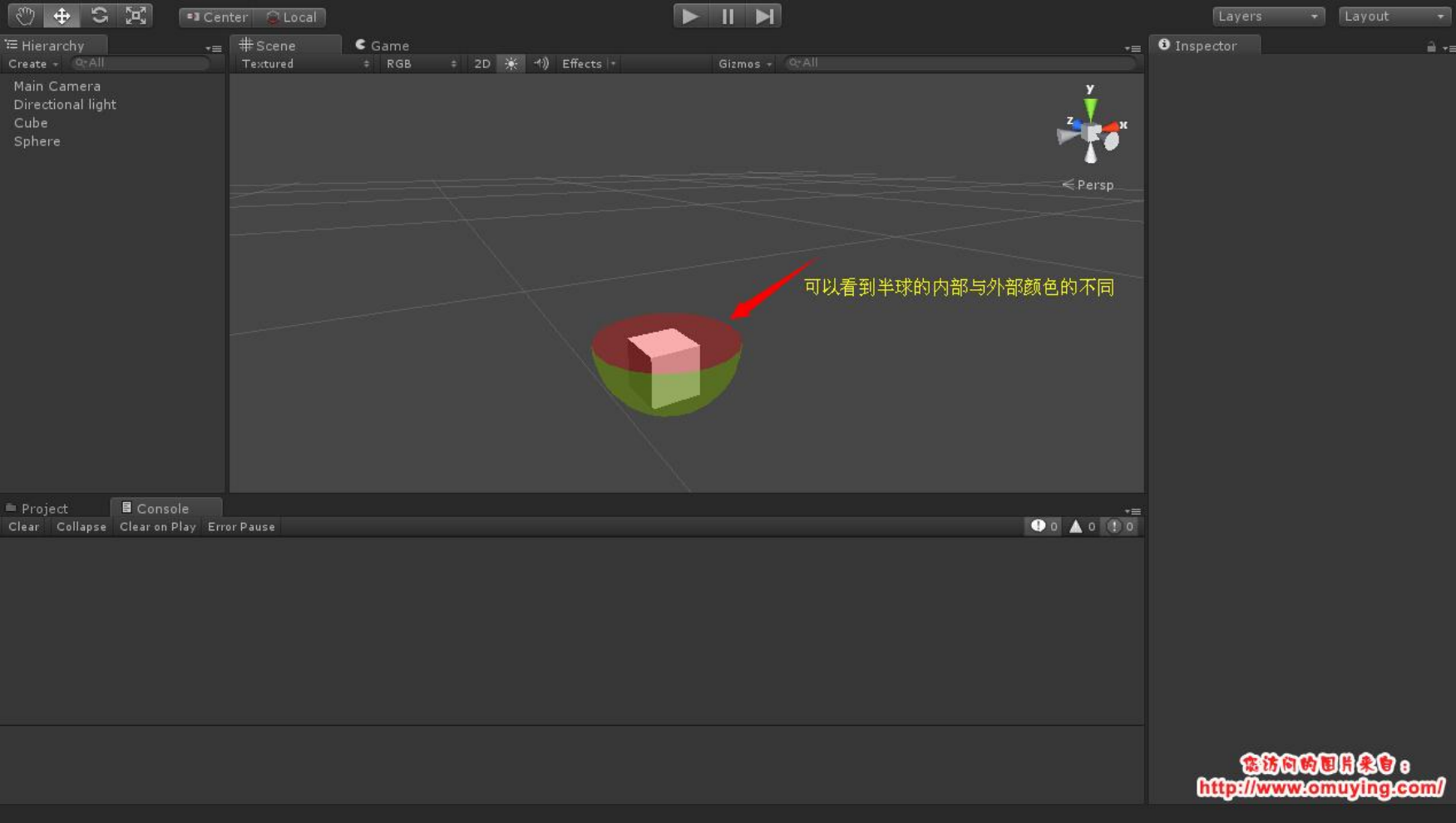
包括后脸

以前的着色器能适用于其他对象，但是它实际上并不会渲染对象的内部，然而，如果我们能看穿一个对象的外部时，我们也应该可以看到对象的内部，如上一章节《剖面模型》中的那样，我们可以使用 Cull Off 停用剔除，然而，如果我们只是停用剔除，这可能会遇到麻烦，如上面所述，因为透明片段可能经常不会被剔除，并且片段有渲染顺序的问题，这可能会导致对象内部与外部重叠的三角形被随机渲染，这可能会导致令人讨厌的结果，因此我们要确保内部在外部之前呈现，在 Unity 的 ShaderLab 中，可以指定两个 Pass 来为同一个网格定义顺序（这儿为了查看测试效果，添加了片段剔除，可以查看原文链接中的代码）：

```
01 Shader "Cg shader using blending"
02 {
03     SubShader
04     {
05         Tags { "Queue" = "Transparent" }
06         // draw after all opaque geometry has been drawn
07         Pass
08         {
09             Cull Front // first pass renders only back faces
10             // (the "inside")
11             ZWrite Off // don't write to depth buffer
12             // in order not to occlude other objects
13             Blend SrcAlpha OneMinusSrcAlpha // use alpha blending
14
15             CGPROGRAM
16
17             #pragma vertex vert
18             #pragma fragment frag
19
20             struct vertexOutput {
21                 float4 pos : SV_POSITION;
22                 float4 posInObjectCoords : TEXCOORD0; //测试使用，可以删除
23             };
24
25             vertexOutput vert(float4 vertexPos : POSITION) : SV_POSITION
26             {
27                 vertexOutput output;
28                 output.pos = mul(UNITY_MATRIX_MVP, vertexPos);
29                 output.posInObjectCoords = vertexPos; //测试使用，可以删除
30                 return output;
31             }
32
33             float4 frag(vertexOutput input) : COLOR
34             {
35                 if (input.posInObjectCoords.y > 0.0) //测试使用，可以删除
36                 {
37                     discard; // drop the fragment if y coordinate > 0
38                 }
39                 return float4(1.0, 0.0, 0.0, 0.3);
40                 // the fourth component (alpha) is important:
41                 // this is semitransparent red
42             }
43             ENDCG
44         }
45
46         Pass
47         {
48             Cull Back // second pass renders only front faces
49             // (the "outside")
50             ZWrite Off // don't write to depth buffer
51             // in order not to occlude other objects
52             Blend SrcAlpha OneMinusSrcAlpha // use alpha blending
53
54             CGPROGRAM
55
56             #pragma vertex vert
57             #pragma fragment frag
58
59             struct vertexOutput {
60                 float4 pos : SV_POSITION;
61                 float4 posInObjectCoords : TEXCOORD0; //测试使用，可以删除
62             };
63
64             vertexOutput vert(float4 vertexPos : POSITION) : SV_POSITION
65             {
66                 vertexOutput output;
67                 output.pos = mul(UNITY_MATRIX_MVP, vertexPos);
68                 output.posInObjectCoords = vertexPos; //测试使用，可以删除
69                 return output;
70             }
71
72             float4 frag(vertexOutput input) : COLOR
73             {
74                 if (input.posInObjectCoords.y > 0.0) //测试使用，可以删除
75                 {
76                     discard; // drop the fragment if y coordinate > 0
77                 }
78                 return float4(0.0, 1.0, 0.0, 0.3);
```

```
79 // the fourth component (alpha) is important:
80 // this is semitransparent green
81 }
82 ENDCG
83 }
84 }
85 }
```

我们可以把这个着色器应用到一个球体对象上，可以看到半球的内外颜色会不相同，但依然是透明的，查看效果如图：



在这个着色器中，第一个 pass 使用前脸剔除来渲染背面（内部），第二个 pass 使用后脸剔除来渲染前面（外部），这个着色器适合凸起的网格（封闭网无凹痕，例如球体或者立方体）或者近似的网格。

恭喜你，在本章节中你应该了解：

- 1、Unity 指定了哪些混合。
- 2、在场景中渲染透明和不透明对象以及怎样归类 Unity 的透明和不透明对象。
- 3、使用两个 pass 来渲染透明对象的内部和外部。

资源下载地址：[点击下载](#)，共下载 22 次。

前一篇：[第五章：剖面模型（关于片段擦除和三角形面剔除）](#)

后一篇：[第七章：顺序无关的透明度（关于顺序无关的混合）](#)



赞

8 人



打酱油

0 人



呵呵

0 人



鄙视

0 人



正能量

1 人



0

1 条评论

最新 最早 最热




965


后面那段代码有报错，返回类型报错！25和64行改为：vertexOutput vert(float4 vertexPos:POSITION)

8月18日 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»



说点什么吧...



发布

最终幻想正在使用多说

