# 纹理

使用 纹理映射 技术，我们可以把一张图 粘在 模型表面，逐纹素的控制模型的颜色。

美术人员 会利用纹理展开技术 把 纹理映射坐标 存储在每个顶点上。纹理映射坐标定义了该顶点在纹理中对应的2D坐标。

Unity使用的纹理控件是符合OpenGL的传统的，原点位于左下角。

我们通常会使用一张纹理来代替物体的漫反射颜色。

Texture Type 纹理的类型

Texture Shape 纹理的形状

Alpha Source 透明通道的值来自于哪里

Alpha Is Transparency 是否是透明的

Non Power of 2 如果尺寸不是2的幂数，才用那种方式近似

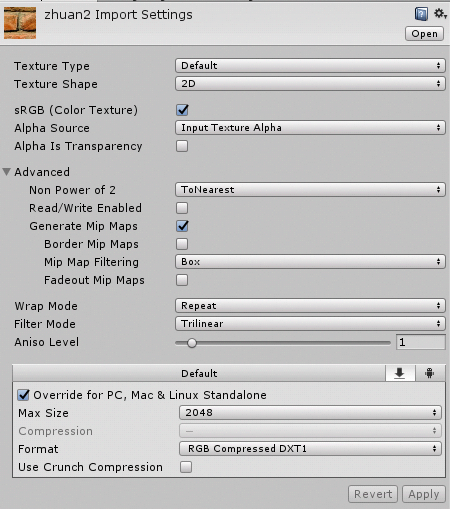
Read/Write Enable 是否开启 读取/写入

Generate Mip Maps 是否开启多级渐进纹理，多占用33% 的内存。

FilterMode 过滤模式 Point 最近邻滤波；Bilinear 是线性滤波，对于每个目标像素找4个临近像素，进行插值；Trilinear 与 Bilinear一样，但同时添加了在多级渐远纹理之间进行混合。

WrapMode 平铺模式，Repeat重复，Clamp截断模式

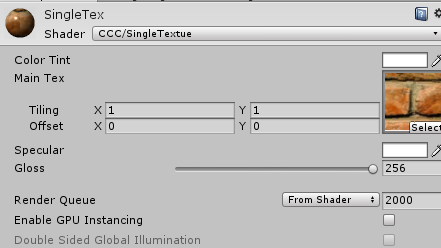
MaxSize 支持的最大的尺寸

Format 纹理的格式，

### 总结

1 纹理 首先要有个类型，该类型决定是什么样的纹理，比如普通纹理，法线纹理，等等。选定纹理的类型，是为了让Unity知道我们的意图，为UnityShader传递正确的纹理，并在一些情况下进行优化

2 纹理还有个形状，主要是图片是立方体。



3 Material中纹理存在Tiling和Offset属性。

Tiling是平铺属性在X方向和Y方向平铺几倍的纹理大小

Offset是偏移属性

4 贴图小了，即当纹理坐标超过了[0,1]的区间时，WrapModel 决定了才用那种方法来绘制多余的部分，repeat 进行重复绘制，clamp小于0的部分用0的颜色，大于1 的部分用1的颜色

5 贴图过大，原纹理中多个纹素对应一个目标像素，需要处理抗锯齿问题。最常用的方法是开启 多级渐远纹理，不同的远近使用不同的纹理，但这样会增加33%的内存占用量。

6 有了贴图之后，需要对贴图采样，采样模式有三种：

Point：选择最近邻；

Bilinear：找到周围的4个像素，进行混合插值

Trilinear：以上的方法 + 多级减员纹理的插值。

7 最后 关于纹理的 最大尺寸和存储格式。

# 凹凸映射

使用一张纹理来修改模型表面的法线。

### 高度纹理

使用一张高度纹理 来模拟 表面位移，然后得到一个修改后的法线值， ----高度映射

高度图中存放的是强度值，表示模型局部的海拔高度。越浅越高，越深越凹。

缺点：实时计算时，不能直接得到表面法线，需要由灰度值计算得到。

### 法线纹理

使用一张法线纹理来直接存储表面法线，----法线映射

法线纹理中 存储的就是表面的法线方向，法线分量是[-1,1],而像素分量是[0,1]。所以需要映射：

Shader中我们有一张法线纹理，采样后获取的是一个像素值，还需要转为 法线值。要对结果进行反映射

那么问题：获取的法线是存在哪个坐标系下的？

方式一：对于模型顶点自带的法线，他们是定义在模型空间中的，一种想法是把修改后的模型空间中的表面法线存储在一张纹理中，这种纹理称为 模型空间的法线纹理。

方式二：实际中，才用的是切线空间来存储法线。每个顶点都有一个切线空间，原点为顶点本身，Z是法线方向，X为切线方向，Y是法线和切线的叉积。 称为切线空间纹理。

#### 我们使用的纹理，首先通过 tex2D(tex, uv) 获取纹理的颜色，对于法线纹理，这个位于[0，1]的颜色并不是颜色，而是法线向量，因为法线向量是[-1,1]，所以需要通过 color\*2-1；

获取法线向量。

#### 这个法线向量可能有两种情况，1 这个向量的坐标系是 模型空间，向量的方向会是四面八方，所以对应的颜色也是五颜六色。2 对应的坐标系是 切线空间，法线向量仅仅是原来法线的扰动，如果没有发生扰动，则法线就是原来的z轴，即（0,0,1）；对应的颜色是（normal+1）/2, 即(0.5, 0.5, 1)即为蓝色。

### 模型空间存储法线的优点：

1 实现简单，更加直观

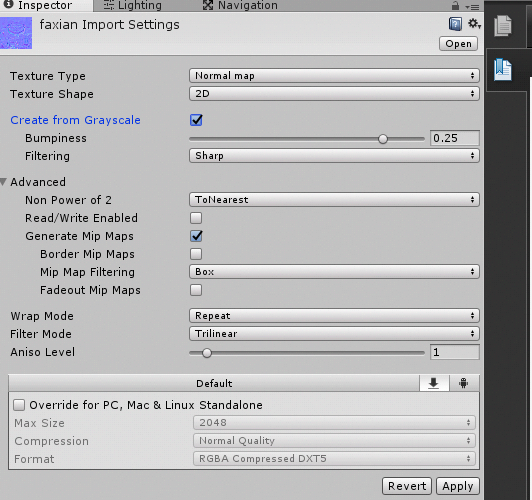
2 可以提供平滑的边界。

### 切线空间存储法线的优点：

1 自由度很高。模型空间的法线只能用于本模型，切线空间的法线可用于任何模型。

2 可进行UV动画。

3 可以重用法线纹理。

4 可压缩，切线空间的法线知道x，y即可求出z的方向。

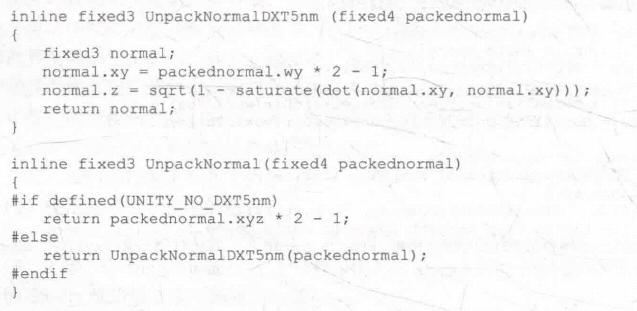
高度图本身保存的是相对高度。

Create from Grayscale 是从高度图中生成法线纹理。就可以把它和切线空间下的法线纹理同等对待了。

有些平台使用了 DXT5nm 格式进行压缩法线纹理：

这种格式的法线纹理中 纹素的w->x分量；g->y 分量。Z分量可以通过xy求出来。

使用这种压缩，可以减少内存占用。



# 总结：

当 纹理 被设置成 normal map 类型的时候，其实根据平台不同可能进行了DXT5nm压缩，这种压缩 只保存了两个变量 w y 分别对应x，y。z方向通过xy求出。

## 透明效果

在实时渲染中要实现透明效果，通常会在渲染模型时控制了他的透明通道。

当开启透明混合后，当一个物体被渲染到屏幕上时，每个片元除了颜色值和深度值之外，还有一个属性——透明度。1 不透明，0 完全透明

实现透明效果的两种方法： 透明度测试 和 透明度混合

原来 我们通过 深度缓冲，来判断渲染的遮挡问题。但，当我们使用透明混合时，我们关闭了深度写入。

#### 透明度测试

设置一个阈值，如果透明度不满足该阈值，则会被舍弃；否则，进行绘制。不用关闭深度写入。效果只有两种，要么完全透明，要么完全不透明。

#### 透明度混合

使用当前片元的透明度作为混合因子，与已经存储在颜色缓冲中的颜色进行混合，得到新的颜色。

需要关闭 深度写入，但是不能关闭 深度测试。

渲染的常用方法：

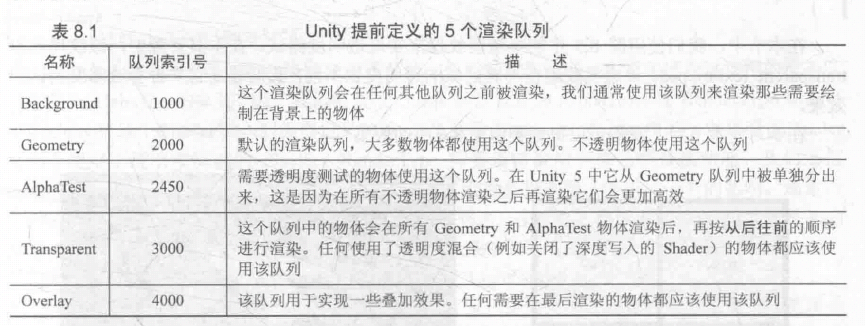
1 先渲染所有不透明的物体，并开启它们的深度测试和深度写入

2 把半透明物体按它们距离摄像机的远近进行排序，然后按照从后往前的顺序渲染这些透明物体，并开启它们的深度测试，但关闭深度写入。

3 但是如果是这种情况，我们采取分割网格，将一个复杂的模型拆分成可以独立排序的多个子模型。如果不分割网格，可以试着让透明通道更加柔和，是穿插不明显。还可以使用开启了深度写入的半透明效果来近似模拟物体的半透明。

## UnityShader的渲染顺序

渲染队列：为了解决渲染顺序的问题而生。Queue 标签设置顺序，越小越先绘制。



如果使用 透明度测试，使用 AlphaTest 标签；

Tags {“Queue”=”AlphaTest”}

如果使用 透明度融合，使用 Transparent 标签。

Tags {“Queue”=”Transparent”}

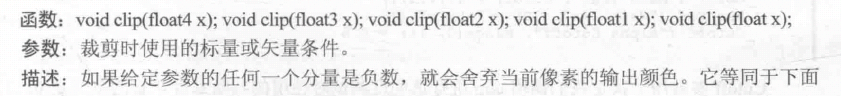
Pass{

ZWrite Off

}

## 透明度测试

原理：只要一个片元的透明度不满足条件，通常是小于某个阈值，那么他对应的片元就会被舍弃，被舍弃的片元不会进行任何处理，也不会对颜色缓冲产生任何影响；否则，就会按照普通的不透明物体的处理方式处理它。

CG函数 clip() 进行透明度测试

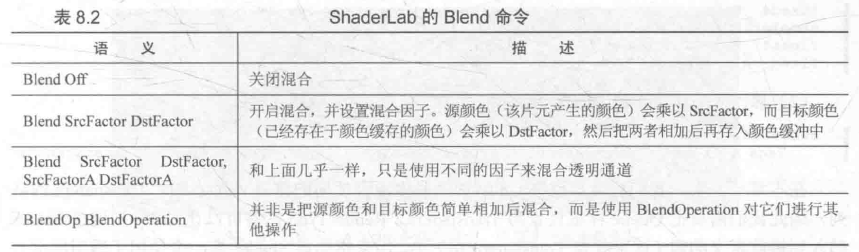
Tags{ "Queue" = "AlphaTest" "IgnoreProjector" = "True" "RenderType" = "TransparentCutout" }

clip(texColor.a - \_Cutoff);

## 透明度混合

原理：使用当前片元的透明度作为混合因子，与已经存储在颜色缓冲中的颜色值进行混合，得到新的颜色。

注意：因为关闭了深度写入，所以我们要注意渲染顺序



注意：设置混合因子的同时，也开启了混合模式。

#### 小结

其实一共有四个因子，一个操作方法。

四个因子 分别是 源颜色的因子，目标颜色的因子，源颜色Alpha的因子，目标颜色Alpha的因子。

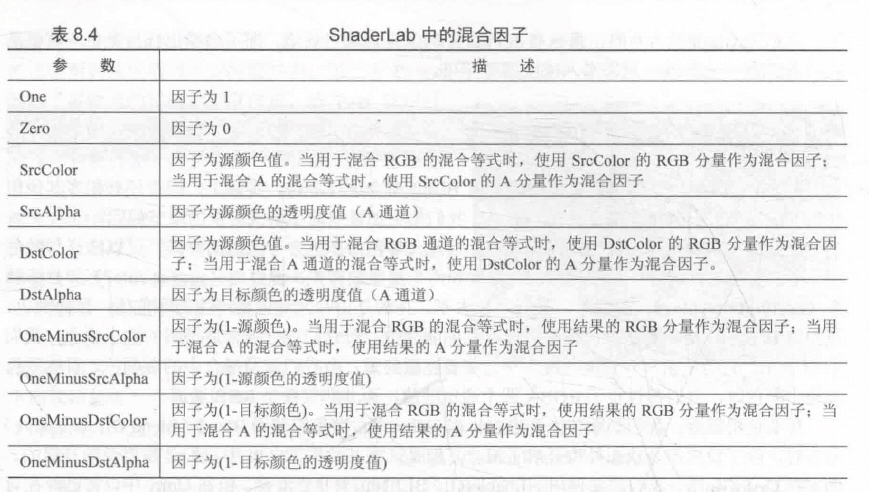
一个操作方法 指的是这些颜色 与 因子如何组合起来。

Blend SrcFactor DstFactor 设置了 颜色的因子，因为没有设置 透明度的因子，因此透明度也是用的这些因子。

Blend SrcFactor DstFactor, SrcFactorA DstFactorA 设置了四个因子。

BlendOp BlendOperation 混合的操作方法，默认是相加

Unity中的混合命令:

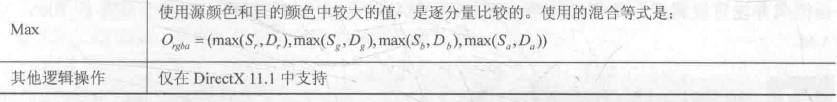
可用因子：

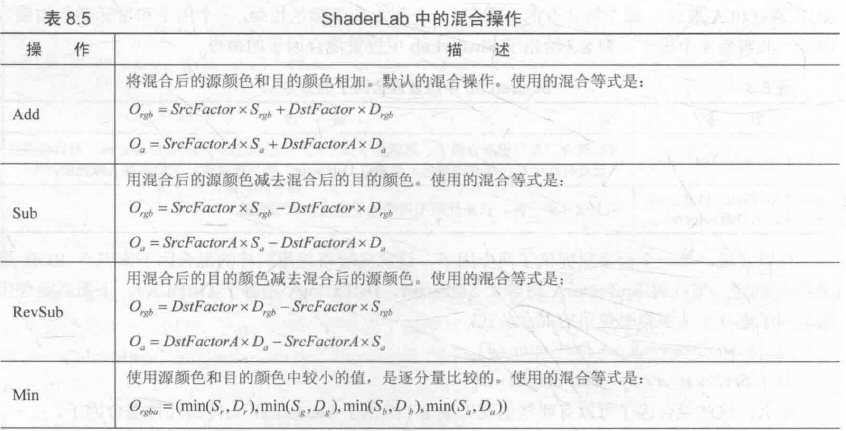
例如：Blend SrcAlpha OneMinusSrcAlpha, One Zero

的意思是： 最终颜色 = 源颜色 \* srcAlpha + (1-srcAlpha) \*目标颜色

最终透明度 = 源颜色的透明度 \* 1 + 0 \* 目标颜色的透明度

这里默认是加操作

可用的混合方法：



注意：默认是Add操作，Min，Max与混合因子没有关系。

Blend SrcAlpha OneMinusSrcAlpha // 正常的混合方式，透明度混合 CS \* A + CT\*(1-A)

Blend OneMinusDstColor One // 柔和相加 CS\*(1-CT) + CT

Blend DstColor Zero // 正片叠底 就是相乘 CS\*CT + CT\*0 = CS\*CT

Blend DstColor SrcColor // 两倍相乘 CS\*CT + CT\*CS = 2CS\*CT

BlendOp Min // 变暗 因为取得最小值

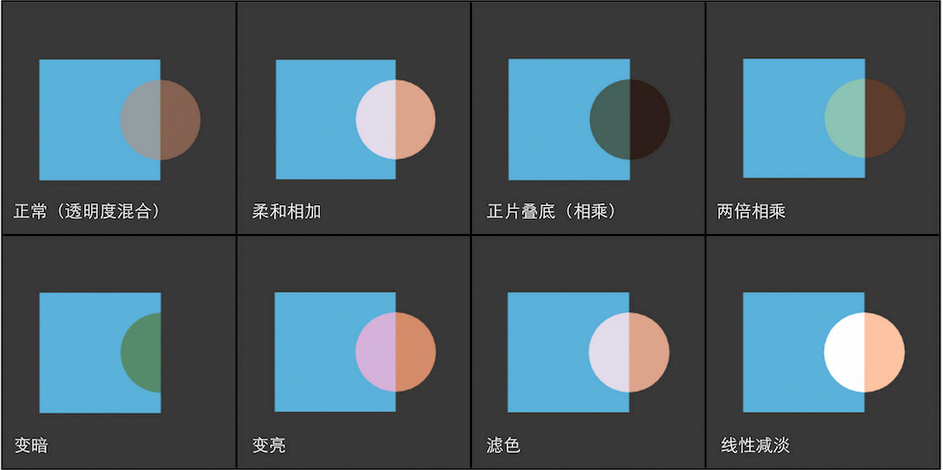
Blend One One

BlendOp max // 变亮 因为取得最大值

Blend One One

Blend OneMinusDstColor One // 滤色 CS\*(1-CT) + CT

Blend One OneMinusSrcColor //滤色 CS + (1-CS)\*CT

Blend One One // 线性减淡 CS+CT

## 混合透视

Tags{ "Queue" = "Transparent" "IgnoreProjector" = "True" "RenderType" = "Transparent" }

ZWrite Off

fixed4 frag(v2f i) : SV\_Target

{

fixed3 worldNormal = normalize(i.worldNormal);

fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));

fixed4 texColor = tex2D(\_MainTex, i.uv);

fixed3 albedo = texColor.rgb \* \_Color.rgb;

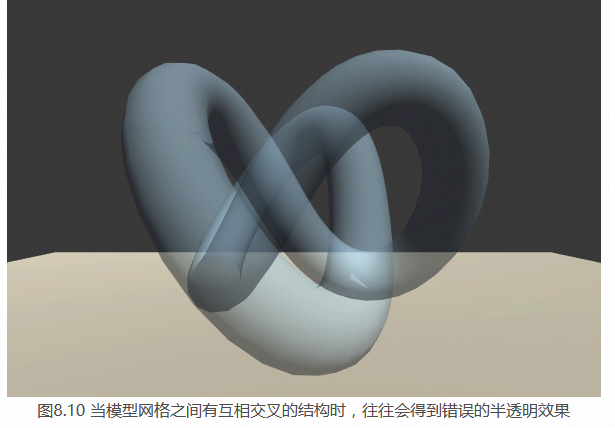
fixed3 ambient = UNITY\_LIGHTMODEL\_AMBIENT.xyz \* albedo;

fixed3 diffuse = \_LightColor0.rgb \* albedo \* saturate(dot(worldLightDir, worldNormal));

return fixed4(ambient + diffuse, texColor.a \* \_AlphaScale);

}

#### 问题1：自身互相遮挡的模型

因为关闭了 深度写入，对于同一模型 无法判断渲染顺序

解决方法：需要添加一个Pass，该Pass中不计算颜色，只进行深度写入

因为渲染 按顺序执行Pass所以要放在前边。多一个Pass会对性能造成一定损耗。

Pass{

ZWrite on

ColorMask 0

}

ColorMask 用于设置颜色通道的 “写掩码”,他的语义如下:

ColorMask RGB |A|0|R|G|B

ColorMask 设为0时，意味着Pass不写入任何颜色

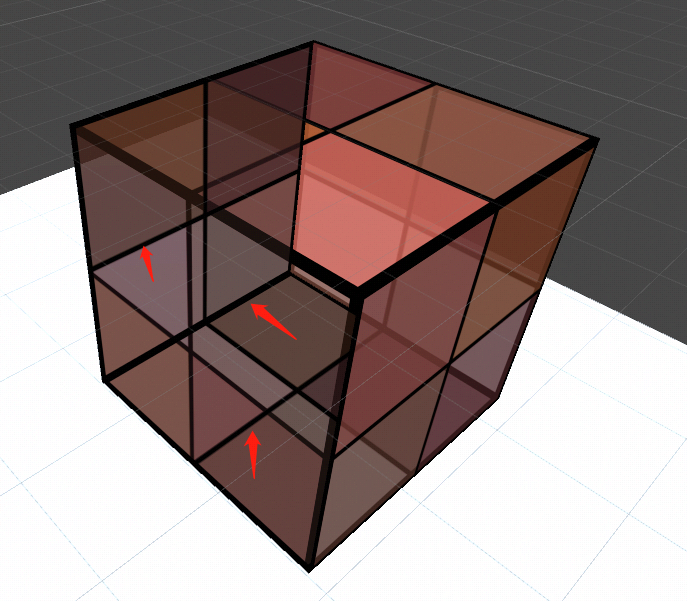
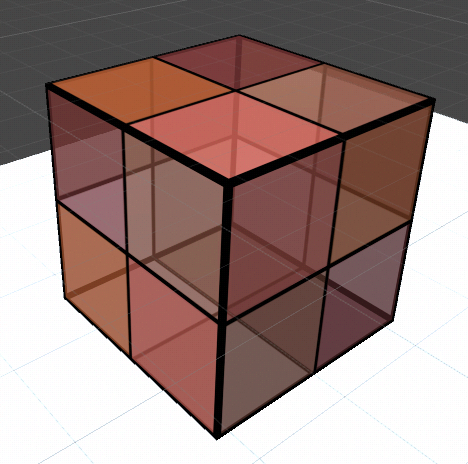
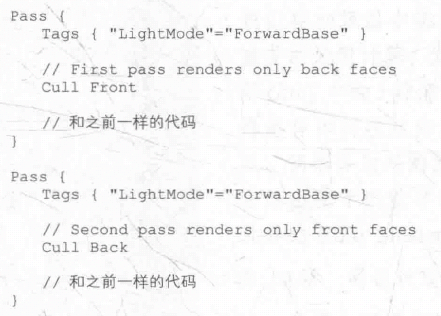
#### 问题2：双面渲染的透明效果

使用 Cull Off|Front|Back 可以进行选择是：关闭剔除，剔除前面，剔除背面 操作

因为透视可以看到内部，所以我们要看到背面。

如果关闭剔除：

后边的黑色框，与前边的黑色框 一样颜色，不能区分前后。这还是因为没有深度写入造成的。

解决方法：先绘制后面的，在绘制前面的。