# Unity的渲染路径

渲染路径 决定了光照是如何应用到UnityShader中的。

在pass中设置了 渲染路径，Unity才会把光源和处理后的光照信息都放在某个数据中。

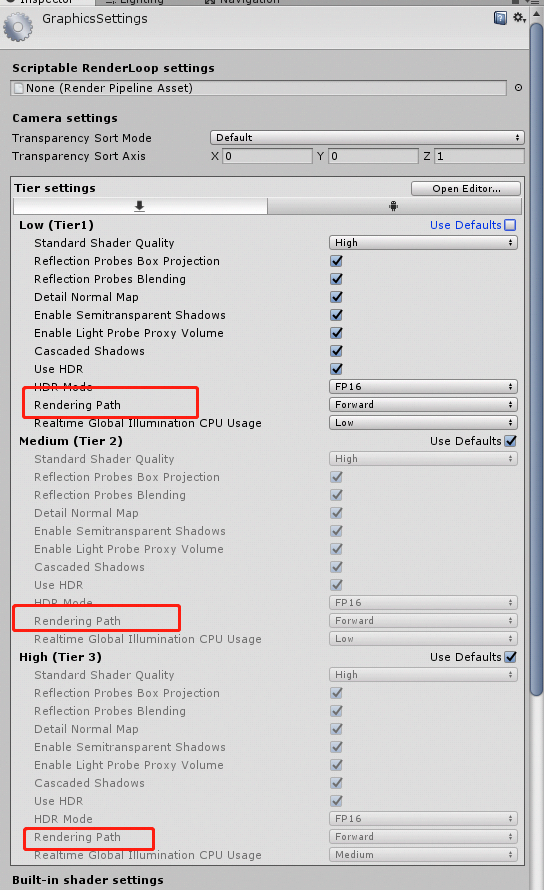
渲染路径：

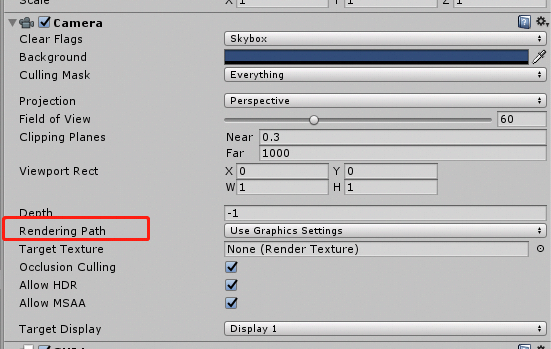
1 前向渲染路径(Forward)

2 延迟渲染路径(Deferred) ==> 更新为 新的延迟渲染路径，但兼容

3 顶点照明渲染路径(Vertex Lit) ==> 弃用，但兼容

设置渲染路径的位置：

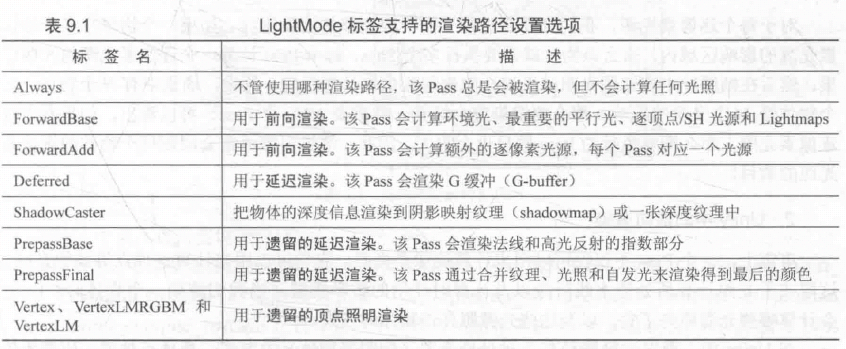
1 统一默认的设置

2 摄像机的单独设置

Pass的标签

Pass{

Tags {“LightMode”=”Forwardbase”}

}

**设置路径的过程，其实是告诉Unity要准备哪些数据，一会这些数据将会用在Pass中。**

**Shader的渲染位置主要在Pass中，pass中有的处理光照，有的不处理光照，有的需要两个处理光照。**

# 前向渲染路径

原理：每进行一次完整的前向渲染，我们需要渲染该对象的渲染图元，并计算两个缓冲区的信息(颜色缓冲区，深度缓冲区)。我们利用深度缓冲来决定一个片元是否可见，如果可见更新颜色缓冲区中的颜色值。

\*\*\* 对于 逐像素光源，我们都需要进行上面一次完整的渲染流程。如果有多个 逐像素光源，则进行多个Pass，每计算一个 逐像素光源的光照结果，就在帧缓冲中把这些光照结果混合起来。

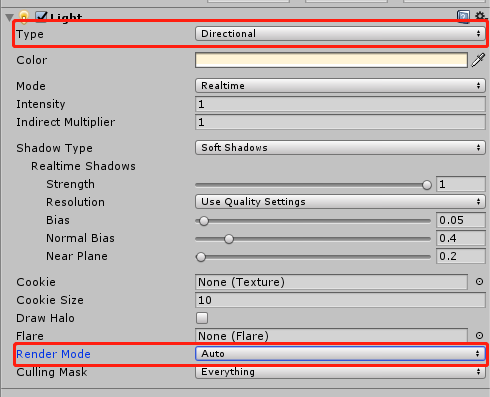
例如，如果有N个物体，M个逐像素光源，需要进行N\*M个Pass，数目很大，需要限制物体的逐像素光照的数目。

前向渲染路径有三种光照方式，即照亮物体的方式。

逐顶点处理、逐像素处理、球谐函数(SH)

光照计算所处流水线阶段 和 计算时使用的数学模型 决定了 Pass 计算那种光照方式

当我们渲染一个物体时， Unity会计算 有哪些光，以及这些光照亮物体的方式。

决定一个光源 的 光照模式 是 光源的类型 和 渲染模式。光源类型：平行光，点光等；渲染模式：Auto, Important, Not Important。

例如，如果设置为 Important，则采用 逐像素光源处理。

**光源排序**

前向渲染中，Unity会对光源排序，排序参考 光源的设置以及对物体的影响程度：

1 场景中 最亮的平行光 逐像素处理

2 如果是 Important 的光源，逐像素处理

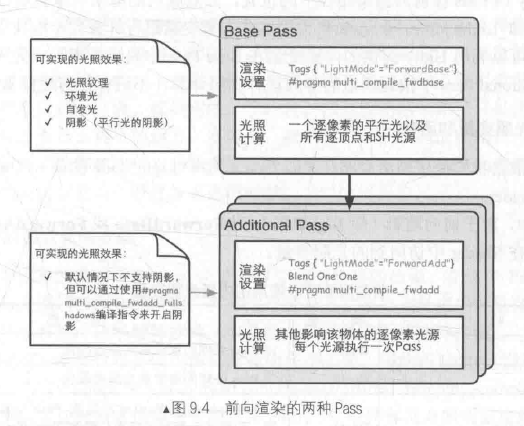
3 如果是 Not Important 的光源，逐顶点 或者 SH处理

4 逐像素的光源数量 不能小于 QualitySetting中的PixelLightCount，否则，会有更多的光源以逐像素的方式渲染。

**光源排序后分类**

排序完成后，一部分光源 按照逐像素处理，不超过4个光源 按照逐顶点处理，剩下的按照SH处理

**光源的处理位置** BasePass 和 AddPass

**注意：**

1 前向渲染，一个Shader通常会定义一个BasePass和一个AddPass，**BasePass一般只会执行一次**，除非进行双面渲染。**AddPass会根据其他逐像素光源的数目多次调用**。

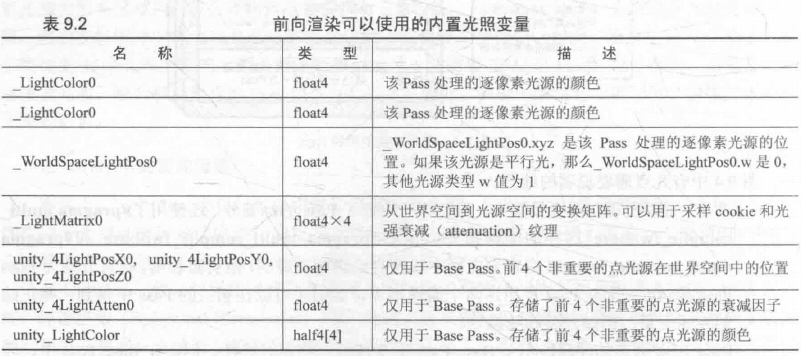
2 **AddPass 需要开启 混合**，因为希望得到剩余的光的总渲染效果。

3 B**asePass 计算 纹理，环境光，自发光，和阴影**。

AddPass默认不计算阴影。除非把

#pragma multi\_compile\_fwdadd 换为 #pragma\_multi\_compile\_fwdadd\_fullshadows

#### 内置的光照变量和函数



**\_LightColor0 该Pass处理的逐像素光源的颜色**

**\_WorldSpaceLightPos0 该Pass处理逐像素光源的位置。如果是平行光w为0，其他为1**

**\_LightMatrix0 类型是float4x4，从世界空间到光源空间的变换**

前四个非重要的**点光源**，只能在**前向渲染**。

**unity\_4LightPosX0** float4 //.

**unity\_4LightPosY0** float4 //

**unity\_4LightPosZ0** float4 //前四个非重要的点光源的世界空间中的位置

**unity\_LightColor** half4[4] //前四个非重要的点光源的颜色。

**unity\_4LightAtten0**  float4 //前四个非重要的点光源的衰减因子。

只能在前向渲染的函数:

float3 **WorldSpaceLightDir**(float4 v);

float3 **UnityWorldSpaceLightDir**(float4 v);

float3 **ObjSpaceLightDir**(float4 v);

float3 **Shade4PointLights**(...); 计算四个点光源的光照

# 顶点照明渲染路径

顶点照明渲染路径 是对 硬件配置要求最少，运算性能最高，效果最差

不支持 通过逐像素才能得到的效果，例如不支持 阴影，法线映射，高精度的高光反射等

它只是 前向渲染的子集，它完成的功能在 前向渲染中都能完成。

它只是 使用了逐顶点的方式来计算光照。

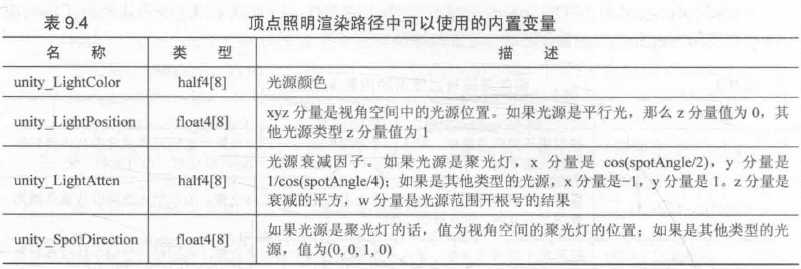
如果使用顶点照明渲染，unity只填充哪些逐顶点相关的光源信息，不可使用逐像素的变量和方法。

#### 照明渲染

通常在一个Pass中就可以完成对物体的渲染，在这个Pass中，我们会计算我们关心的所有光源对该物体的照明，并且这个计算是按逐顶点处理的。

#### 内置函数和变量

在一个顶点照明的Pass中 最多访问8个逐顶点光源。

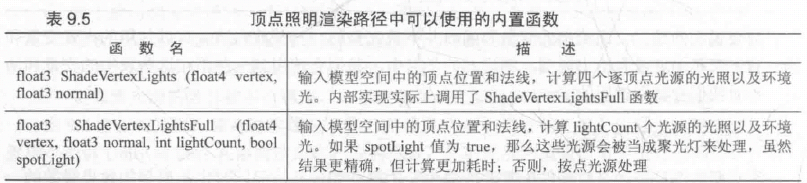
如果光源数量小于8，则其他的光源为黑色。

**unity\_LightColor half4[8] 光源颜色**

**unity\_LightPosition float4[8] 光源位置， 如果是平行光 z为0，其他光为z为1**

**unity\_LightAtten half4[8] 光照衰减，**

**unity\_SpotDirection float4[8] 如果是聚光灯，值为视角空间的聚光灯的位置；否则为（0,0,1,0）**



# 延迟渲染路径

利用G缓冲，存储了我们所关心的表面的其他信息，例如表面的法线，位置，材质属性等。

#### 原理

**利用两个Pass：**

**1 不进行光照，仅仅计算哪些片元可见，这主要是通过深度缓冲实现，把可见的片元的相关信息存储到G缓冲区。**

**2 利用G缓冲区的各个片元信息，进行光照计算。**

**注意：延迟渲染的Pass只有两个，与光源数目没有关系，与屏幕空间的大小有关。**

#### 适用范围：

**场景中光源数目很多，如果使用前向渲染会造成性能瓶颈的情况下。**

**延迟渲染每个光源都可以按照 逐像素方式 处理。**

#### 缺点：

**不支持真正的抗锯齿**

**不能处理半透明物体**

**对显卡有一定要求，显卡必须支持MRT、ShaderMode3.0的深度渲染纹理以及双面模板缓冲**

#### 过程：

Pass1 用于渲染G缓冲，把物体的漫反射颜色，高光反射颜色，平滑度，法线，自发光，深度等信息渲染到屏幕空间的G缓冲区。一个物体，只会执行一次。

Pass2 计算光照模型。

默认的G缓冲区包含几个渲染纹理：

RT0: 格式是ARGB32， RGB漫反射颜色；A没有被使用

RT1：格式是ARGB32， RGB 高光反射颜色；A高光反射的指数部分

RT2：格式是ARGB2101010 RGB 存储法线，A没有被使用

RT3：格式是ARGB32， 用于存储自发光+lightmap+反射探针；

深度缓冲和模板缓冲

Pass2 中默认使用的Unity内置的Standard光照模型。如果要使用其他光照模型，需要替换掉 Internal-DeferredShading.shader

#### 可访问的变量和函数

## 渲染路径比较

|  | **延迟** | **前向** | **旧版延迟** | **顶点光照** |
| --- | --- | --- | --- | --- |
| **功能** |  |  |  |  |
| 每像素光照（法线贴图、光照剪影） | 是 | 是 | 是 | - |
| 实时阴影 | 是 | 带有警告 | 是 | - |
| 反射探针 | 是 | 是 | - | - |
| 深度和法线缓冲区 | 是 | 其他渲染pass | 是 | - |
| 软粒子 | 是 | - | 是 | - |
| 半透明对象 | - | 是 | - | 是 |
| 抗锯齿 | - | 是 | - | 是 |
| 光照剔除遮罩 | 受限 | 是 | 受限 | 是 |
| 光照保真度 | 全部每像素 | 部分每像素 | 全部每像素 | 全部每顶点 |
| **性能** |  |  |  |  |
| 每像素光照的成本 | 照射像素数量 | 像素数量 \* 照射对象数量 | 照射像素数量 | - |
| 正常渲染对象的次数 | 1 | 每像素光照的数量 | 2 | 1 |
| 简单场景的开销 | 高 | 无 | 中 | 无 |
| **平台支持** |  |  |  |  |
| PC (Windows/Mac) | Shader Model 3.0+ 和 MRT | 所有 | Shader Model 3.0+ | 所有 |
| 移动端 (iOS/Android) | OpenGL ES 3.0 和 MRT、Metal（在搭载 A8 或更高版本 SoC 的设备上） | 所有 | OpenGL ES 2.0 | 所有 |
| 游戏主机 | XB1、PS4 | 所有 | XB1、PS4、360 | - |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 前向渲染路径 | **\_LightColor0** | float4 | 该Pass处理的逐像素光源颜色 | **float3 WorldSpaceLightDir(float4 v);** | 获取世界光照的方向，使用局部坐标 |
| **\_WorldSpaceLightPos0** | float4 | 该Pass处理的逐像素光源位置 | **float3 UnityWorldSpaceLightDir(float4 v);** | 获取世界光照的方向，使用世界坐标 |
| **\_LightMatrix0** | float4x4 | 世界坐标到该Pass处理的逐像素光源的矩阵 | **float3 ObjSpaceLightDir(float4 v);** | 获取局部光照的方向，使用局部坐标 |
| **unity\_4LightPosX0** | float4 | 前4个非重要点光源的位置X | **float3 Shade4PointLights(...);** | 获取4个点光源的光照 |
| **unity\_4LightPosY0** | float4 | 前4个非重要点光源的位置Y |  |  |
| **unity\_4LightPosZ0** | float4 | 前4个非重要点光源的位置Z |  |  |
| **unity\_LightColor** | half4[4] | 前4个非重要点光源的颜色 |  |  |
| **unity\_4LightAtten0** | float4 | 前4个非重要点光源的衰减 |  |  |
|  |  |  |  |  |  |
| 顶点照明渲染路径 | **unity\_LightColor** | **half4[8]** | **光源颜色** | float3 ShadeVertexLights(float4 vertex, float3 normal) | 获取四个逐顶点光源的光照和环境光 |
| **unity\_LightPosition** | **float4[8]** | **光源位置， 如果是平行光 z为0，其他光为z为1** |  |  |
| **unity\_LightAtten** | **half4[8]** | **光照衰减，** |  |  |
| **unity\_SpotDirection** | **float4[8]** | **如果是聚光灯，值为视角空间的聚光灯的位置；否则为（0,0,1,0）** |  |  |
|  |  |  |  |  |  |
| 延迟渲染路径 | **\_LightColor** | **float4** | **光源颜色** |  |  |
| **\_LightMatrix0** | **float4x4** | **从世界空间到光源空间的变换矩阵** |  |  |

# Unity的光源类型

四种光源：平行光、点光源、聚光灯、面光源（仅在烘焙时使用）

##### 光源对Shader的影响

平行光：方向

点光源：位置，衰减，方向，范围

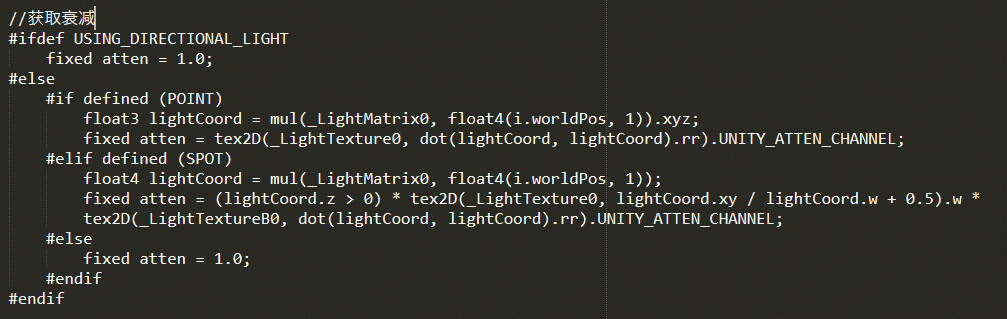
聚光灯：同点光源，但更复杂

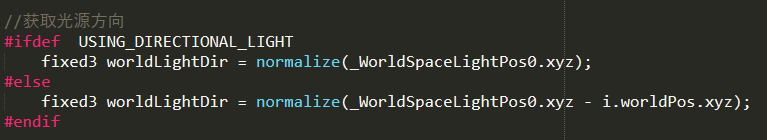
## 前向渲染中 处理不同类型的光源

光源的5个属性：位置，方向，颜色，强度以及衰减。

颜色和强度： \_LightColor0

位置：\_WorldSpaceLightPos0



**### 注意引入头文件 ：#include "AutoLight.cginc"**

**### 最后返回颜色的时候**

**return fixed4(ambient + (diffuse + specular) \* atten, 1.0);**

# Unity的光照衰减

平行光的衰减是1.0

点光源和聚光灯的衰减是查表，大部分能得到良好的效果。

缺点：1 纹理大小会影响精度

2 不直观，无法用其他数学公式计算衰减

#### 用于光照衰减的纹理

光照衰减纹理：**\_LightTexture0**

**如果光源使用了cookie，则纹理是 \_LightTextureB0**

**步骤：**

**1 先获得 定点 在光源坐标系中的位置，**

**float3 lightCoord = mul(\_LightMatrix0, worldPos).xyz;**

**2 对纹理进行采样**

**fixed atten = tex2D(\_LightTexture0, dot(lightCoord, lightCoord).rr).UNITY\_ATTEN\_CHANNEL;**

**为了避免开方，我们用距离的平方 dot(lightCoord, lightCoord).rr，然后通过UNITY\_ATTEN\_CHANNEL 获取衰减值所在的分量。**

#### 计算线性衰减

# Unity 的 阴影

#### 阴影的实现

ShadowMap技术：将摄像机与光源重合，场景中光源的阴影区域就是那些摄像机看不到的地方。

在前向渲染中，如果最重要的平行光开启了阴影，Unity会为该光源计算它的阴影映射纹理，shaowmap，它本质上是一张深度图。记录了从该光源的位置出发，能看到的场景中距离它最近的表面位置(深度信息)。

为了得到深度图，我们可以在BasePass和AdditionalPass中来更新深度信息，但是这样很费，所以，我们用一个单独的Pass来计算深度，即LightMode=”ShadowCaster”。

传统的阴影映射纹理的实现：

正常渲染的Pass中把顶点位置变换到光源空间下，用xy分量对阴影映射纹理进行采样，得到深度信息。如果该深度值 小于 顶点的深度值z，则说明该点位于阴影中。

新的阴影采样技术：**屏幕空间的阴影映射技术**

首先，通过LightMode为ShadowCaster的Pass来得到可投射阴影的光源的 阴影映射纹理 以及摄像机的深度纹理。然后，根据两者 得到 屏幕空间的阴影图。如果摄像机的深度图的表面深度 大于 阴影纹理中的深度值，则说明 该表面虽然可见，但在阴影中。

这样 阴影图 就包含了屏幕空间所有有阴影的区域。如果我们想要一个物体接收来自其他物体的阴影，只需要在Shader中对阴影图进行采样。由于阴影图是屏幕空间下的，因此，我们首先需要把表面坐标从模型空间变换到屏幕空间，然后对这个坐标对阴影图进行采样即可。

阴影图 是通过 光源的阴影映射纹理 和 摄像机的深度纹理 获得的。

阴影图 是所有的阴影区域，通过对阴影图的采集 获取 阴影值，在\*（漫反射+高光反射）。

**总结：**

**1 接受阴影： 如果想要一个物体接收来自其他物体的阴影，就必须在Shader中对阴影映射纹理进行采样，把采样结果和最后的光照结果相乘来产生阴影效果。**

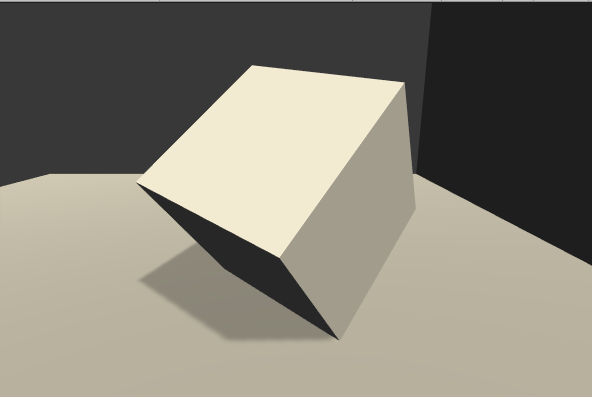
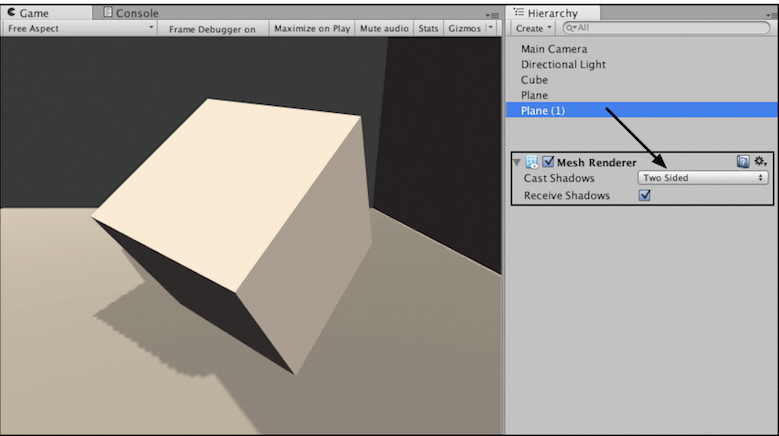
**2 产生阴影： 如果想要一个物体向其他物体投射阴影，就必须把该物体加入到光源和阴影映射纹理中计算，从而让其他物体在对阴影映射纹理采样时，可以得到该物体的相关信息。这个过程是 LightMode=ShadowCaster得到的。**

要想接受 阴影，需要 对阴影映射纹理进行采样，或者对屏幕空间的阴影图 进行采样，采样值\***最后的光照结果 获得结果**

想要产生阴影，需要添加进入 阴影纹理的计算中

#### 产生阴影 使用 LightMode=”ShadowCaster”的Pass

#### 默认情况下，计算光源的阴影映射纹理时会剔除掉物体的背面。但是对于内置平面，他只有一个面，所以它的背光面不会产生阴影。可以选择Two Sided 来允许对物体的所有面都计算阴影信息。



右侧墙没有阴影。 开启双面渲染后，有了阴影。

\*\*\* 底板可以接受阴影，因为标准shader中有处理接受阴影的操作。

\*\*\* 立方体 没有接受阴影，因为没有处理

#### 接收阴影

struct v2f

{

float4 pos : SV\_POSITION;

float3 worldNormal : TEXCOORD0;

float3 worldPos : TEXCOORD1;

SHADOW\_COORDS(2) // 阴影计算1

};

1创建 阴影映射纹理坐标

v2f vert(appdata\_base v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.worldNormal = UnityObjectToWorldNormal(v.normal);

o.worldPos = mul(unity\_ObjectToWorld, v.vertex).xyz;

TRANSFER\_SHADOW(o); // 阴影计算2

return o;

}

2 计算纹理

fixed4 frag(v2f i) : SV\_Target

{

float3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));

float3 worldViewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));

float3 worldNormal = normalize(i.worldNormal);

fixed3 ambient = UNITY\_LIGHTMODEL\_AMBIENT.xyz;

float atten = 1.0;

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

float3 halfDir = normalize(worldLightDir + worldViewDir);

fixed3 specular = \_LightColor0.rgb \* \_Specular.rgb \* pow(max(0, dot(halfDir, worldNormal)), \_Gloss);

fixed shadow = SHADOW\_ATTENUATION(i); // 阴影计算3

return fixed4(ambient + (diffuse + specular)\*atten\*shadow, 1.0);

}

3 获取阴影值，与颜色做乘法

##### 整合阴影 和 光衰

1,2前两步同上

3 获取 两者的乘积

fixed4 frag(v2f i) :SV\_Target

{

float3 worldViewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));

float3 worldNormal = normalize(i.worldNormal);

//获取光源方向

#ifdef USING\_DIRECTIONAL\_LIGHT

fixed3 worldLightDir = normalize(\_WorldSpaceLightPos0.xyz);

#else

fixed3 worldLightDir = normalize(\_WorldSpaceLightPos0.xyz - i.worldPos.xyz);

#endif

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

float3 halfDir = normalize(worldLightDir + worldViewDir);

fixed3 specular = \_LightColor0.rgb \* \_Specular.rgb \* pow(max(0, dot(halfDir, worldNormal)), \_Gloss);

UNITY\_LIGHT\_ATTENUATION(atten, i, i.worldPos);

return fixed4((diffuse + specular)\*atten, 1.0);

}

##### 透明度物体的阴影

**透明度测试的处理：**

**因为在片元着色器中 舍弃了某些片元，而VertexLit的阴影投射纹理并没有这样的操作，所以使用VertexLit是不行的。**

**但是可以使用Unity自带的 Transparent/Cutout/VertexLit 作为Fallback，注意：这里面使用了\_Cutoff 属性来进行透明度测试**

**但是默认情况下，物体渲染到深度图和阴影映射纹理中仅考虑了物体的正面，并没有考虑背面，所以背对光源的面的深度信息没有加入到阴影映射纹理的计算中。所以需要设置为 TwoSided**

#### 透明度融合的阴影

过于复杂，因此半透明的Shader是不会产生任何阴影效果的。