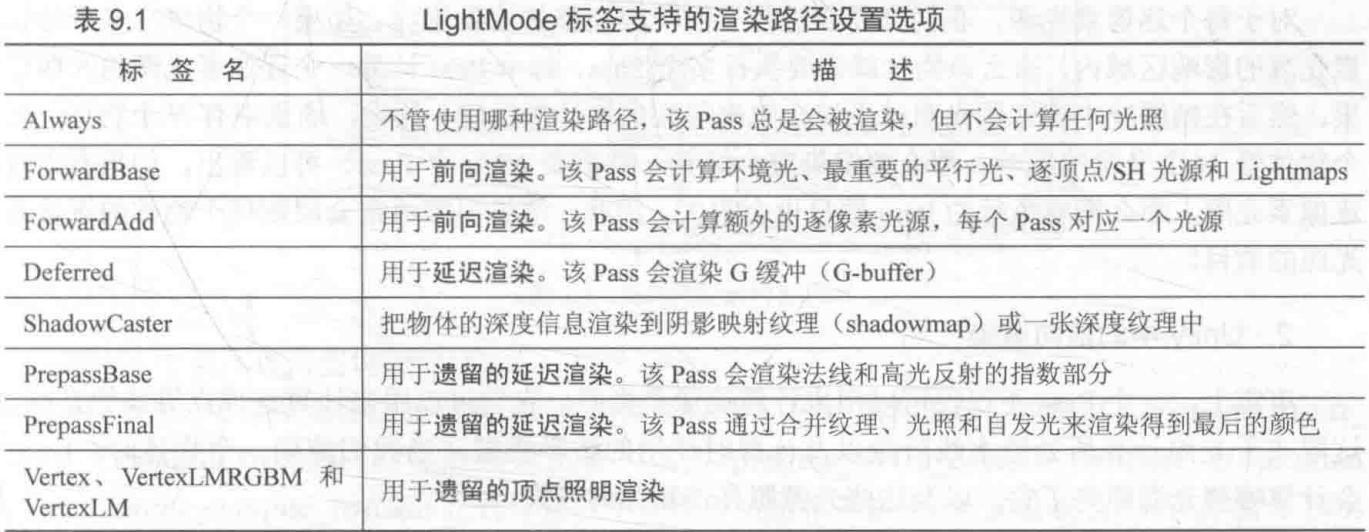
**Unity的渲染路径**

在Unity中，渲染路径决定了光照是如何应用到Unity Shader中的。因此，如果要和光源打交道，我们需要为每个Pass指定它使用的渲染路径，只有这样才能让Unity知道，“哦，原来这个程序员想要这种渲染路径，那么好的，我把光源和处理后的光照信息都放在这些数据里，你可以访问啦”，也就是说，我们只有为Shader正确的选择和设置了需要的渲染路径，该Shader的光照计算才能被正确执行。

Unity支持多种类型的渲染路径。在Unity 5.0版本以前，主要有3种：前向渲染路径（Forward Rendering Path）、延迟渲染路径（Deferred Rendering Path）和顶点照明渲染路径（Vertex Lit Rendering Path）。但在Unity 5.0以后，Unity做了很多更改，主要有两个变化：S后弦，顶点照明渲染路径已经被Unity抛弃（但目前仍然可以对之前使用了顶点照明渲染路径的Unity Shader兼容）；其次，新的延迟渲染路径代替了原来的延迟渲染路径（同样，目前也提供了对旧版本的兼容）

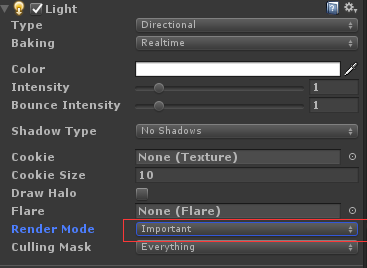
**需要注意的是，如果当前的显卡并不支持所渲染的渲染路径，Unity会自动使用更低一级的渲染路径。例如，如果一个GPU不支持延迟渲染，那么Unity就会使用前向渲染。在Unity 5.x中如果使用了前向渲染又没有为Pass指定任何前向渲染适合的标签，就会被当成一个和顶点照明渲染路径等同的Pass，那么一些光照变量很可能不会被正确赋值，我们计算出的效果也很可能有错误**



**Unity中的前向渲染**

事实上，一个Pass不仅仅可以用来计算逐像素光照，它也可以用来计算逐顶点等其他光照。这取决于光照计算所处流水线阶段以及计算时使用的数学模型。当我们渲染一个物体时，Unity会计算那些光照照亮了它，以及这些关照照亮该物体的方式。

在Unity中，前向渲染路径有3种处理光照的方式：逐顶点处理、逐像素处理、球谐喊函数处理。而决定一个光源使用哪种处理模式取决于它的类型和渲染模式。光源类型指的是该光源是平行光还是其他类型的光源，而光源的渲染模式指的是该光源是否是重要的。如果我们把一个光照的模式设置为Important，意味着我们告诉Unity，“嘿老兄，这个光源很重要，我希望你可以认真对待它，把它当成一个逐像素光源来处理”，我们可以在光源的Light组件中设置这些属性：



在前向渲染中，当我们渲染一个物体时，Unity会根据场景中各个光源的设置以及这些光源对物体的影响程度（例如，距离该物体的远近、光源强度等）对这些光源进行一个重要度排序。其中，一定数量的光源会按逐像素的方式处理，然后由最多4个光源按逐顶点的方式处理，剩下的光源可以按照SH方式处理。Unity使用的判断规则如下：

1）场景中最亮的平行光总是按逐像素处理的

2）渲染模式被设置成Not Important的光源会按照逐顶点或者SH处理

3）渲染模式被设置成Important的光源，会按逐像素处理

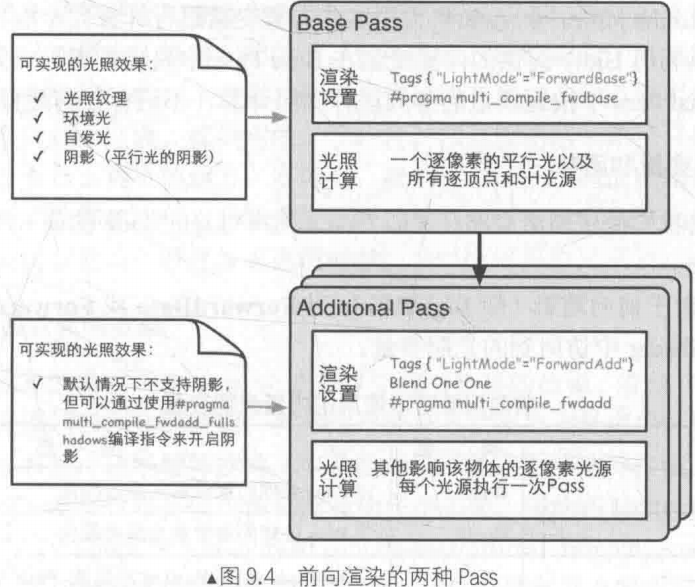
4）如果根据以上规则得到的逐像素光源数量小于Quality Setting中的逐像素光源数量（Pixel Light Count），会有更多的光源以逐像素的方式进行渲染

对于Base Pass来说，它处理的逐像素光源类型一定是平行光，如果场景中没有任何的平行光，那么Base Pass会当成全黑的光源处理，如果有多个平行光，Unity会选择最亮的平行光传递给Base Pass进行逐像素处理，其他的平行光会按照逐顶点或者在Additional Pass中按逐像素处理

在Quality Setting中有：



那么，在哪里进行光照计算？当然是在pass里，前面提到过，前向渲染有两种Pass：Base Pass和Additional Pass。通常来说，这两种Pass进行的标签和渲染设置以及常规光照计算如下图所示：



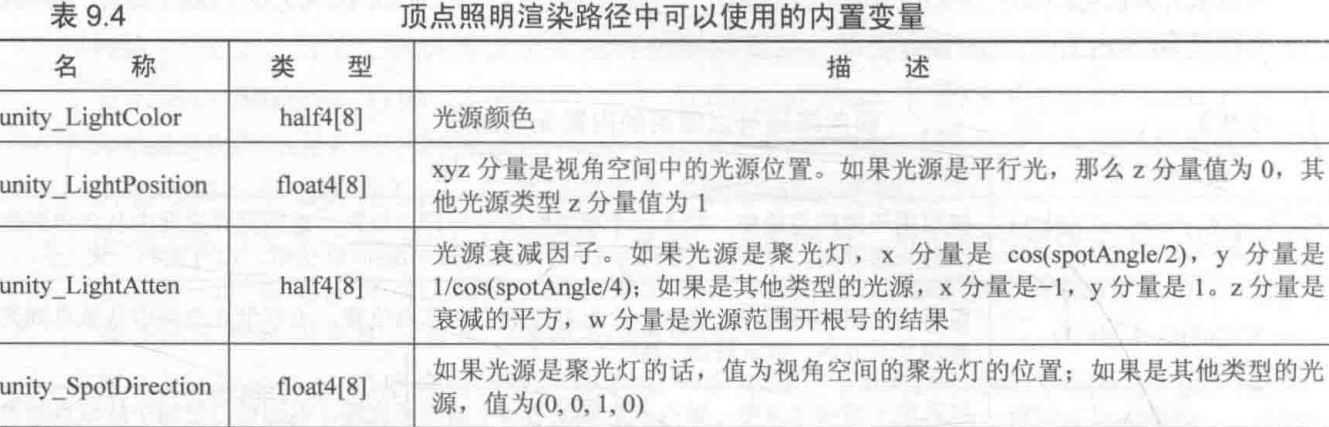
对于上图有几点需要说明的地方：

1. 首先可以发现在渲染设置中，我们除了设置了Pass的标签外，还使用了#pragma multi\_compile\_fwdbase这样的编译命令。虽然#pragma multi\_compile\_fwdbase和#pragma multi\_compile\_fwdadd在官方文档中还没有给出相关说明，但实验表明，只有分别为Base Pass和Additional Pass使用这两个编译指令，我们才可以在相关的Pass中得到一些正确的光照变量，例如光照衰减值
2. Base Pass旁边的注释给出了Base Pass中支持的一些光照特性。例如，在Base Pass中，我们可以访问光照纹理。
3. Base Pass中渲染的平行光默认是支持阴影的（如果开启了光源的阴影功能），而Additional Pass中渲染的光源在默认情况下是没有阴影效果的，即便我们在它的Light组件在宏设置了有阴影的Shadow Type。但我们可以在Additional Pass中使用#pragma multi\_compile\_fwdadd\_fullshadows代替#pragma multi\_compile\_fwdadd编译指令，为点光源和聚光灯开启阴影效果，但这需要Unity在内部使用更多的Shader变种
4. 环境光和自发光也是在Base Pass中计算的。这是因为，对于一个物体来说，环境光和自发光我们只希望计算一次即可，而如果我们在Additional Pass中计算着两种光照，就会造成叠加多次环境光和自发光，这不是我们想要的
5. 在Additional Pass的渲染设置中，我们还开启和设置了混合模式。这是因为，我们希望每个Additional Pass可以与上一次的光照结果在帧缓冲中进行叠加，从而得到最终的有多个光照的渲染效果。如果我们没有开启和设置混合模式，那么Additional Pass的渲染结果会覆盖掉之前的渲染结果，看起来就好像改物体只受到该光源的影响。通常情况下我们选择的混合模式是Blend One One
6. 对于前向渲染来说，一个Unity Shader通常会定义一个Base Pass（Base Pass也可以定义多次，例如需要双面渲染等情况）以及一个Additional Pass。一个Base Pass仅会执行一次（定义了多个Base Pass的情况除外），而一个Additional Pass会根据影响该物体的其它逐像素光源的数目被多次调用，即每个逐像素光源会执行一次Additional Pass。

**Unity中的顶点照明渲染路径**

顶点照明渲染路径是对硬件配置要求最少、运算性能最高，但同时也是得到的效果最差的一种类型，它不支持哪些逐像素才能得到的效果，例如阴影、法线映射、高精度的高光反射等。实际上，它仅仅是前向渲染路径的一个子集，也就是说，所有可以在顶点照明渲染路径中实现的功能都可以在前向渲染路径中完成。就如它的名字一样，顶点照明渲染路径只是使用了逐顶点的方式来计算光照，并没有什么神奇的地方。实际上，我们在上面的前向渲染路径中也可以计算一些逐顶点的光源。但如果选择使用顶点照明渲染路径，那么Unity只会填充那些逐顶点相关的光源变量，意味着我们不可以使用一些逐像素光照变量。

在Unity中，我们可以在一个顶点照明Pass中最多访问到8个逐顶点光源，如果我们只需要渲染其中两个光源对物体的照明，可以仅仅使用下表内置光照数据的前两个。如果影响该物体的光源数量小于8，那么数组中剩下的光源颜色会设置成黑色。



**Unity中的延迟渲染路径**

除了前向渲染中使用的颜色缓冲和深度缓冲外，延迟渲染还会利用额外的缓冲区，这些缓冲区被统称为G缓冲，G是应为geometry的缩写。G缓冲区存储了我们所关心的表面（通常指的是离摄像机最近的表面）的其他信息，例如该表面的法线、位置、用于光照计算的材质属性等。

延迟渲染主要包含了两个Pass。在第一个Pass中，我们不进行任何光照计算，而是仅仅计算那些片元是可见的，这主要是通过深度缓冲技术来实现，当法线一个片元是可见的，我们就把它的相关信息存储到G缓冲区中。然后，在第二个Pass中，我们利用G缓冲区的各个片元信息，例如表面法线、视角方向、满发射系数等，仅仅真正的光照计算。我们可以看出，延迟渲染使用的Pass数目通常就是两个，这跟场景中包含的光源数目是没有关系的。换句话说，延迟渲染的效率不依赖于场景的复杂度，而是和我们使用的屏幕空间的大小有关。这是因为，我们需要的信息都存储在缓冲区中，而这些缓冲区可以理解成是一张张2D图像，我们的计算实际上就是在这些图像空间中进行的。

延迟渲染有一些缺点：

1. 不支持真正的抗锯齿功能
2. 不能处理半透明物体
3. 对显卡有一定要求。如要要使用延迟渲染的话，显卡必须支持MRT（Multiple Render Targets）、Shader Mode 3.0以上、深度渲染纹理以及双面的模板缓冲

默认的G缓冲区（注意，不同Unity版本的渲染纹理存储内容会有所不同）包含了一下几个渲染纹理（Render Texture，RT）

1. RT0：格式是ARGB32，RGB通道用于存储漫反射颜色，A通道没有被使用
2. RT1：格式是ARGB32，RGB通道用于存储高光反射颜色，A通道用于存储高光发射的指数部分
3. RT2：格式是ARGB2101010，RGB通道用于存储法线，A通道没有被使用
4. RT3：格式是ARGB32（非HDR）或ARGBHalf（HDR），用于存储自发光+lightmao+反射探针（reflection probes）
5. 深度缓冲和模板缓冲

测试一下前向渲染：

Shader "Unlit/ForwardRendering"

{

Properties

{

\_Gloss ("Gloss", Range(9.0, 256.0)) = 20.0

}

SubShader

{

pass

{

Tags { "LightMode" = "ForwardBase" }

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_fwdbase

#include "UnityCG.cginc"

#include "Lighting.cginc"

float \_Gloss;

struct a2v

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct v2f

{

float4 position : SV\_POSITION;

float3 worldNormal : TEXCOORD0;

float4 worldPosition : TEXCOORD1;

};

v2f vert(a2v v)

{

v2f o;

o.position = mul(UNITY\_MATRIX\_MVP, v.vertex);

o.worldNormal = UnityObjectToWorldNormal(v.normal);

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

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

// ambient

float3 ambient = UNITY\_LIGHTMODEL\_AMBIENT.rgb;

// diffuse

fixed3 worldNormal = normalize(o.worldNormal);

fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(o.worldPosition));

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

// specular

fixed3 worldViewDir = normalize(UnityWorldSpaceViewDir(o.worldPosition));

fixed3 halfViewDir = normalize(worldLightDir + worldViewDir);

fixed3 specular = \_LightColor0.rgb \* pow(saturate(dot(halfViewDir, worldNormal)), \_Gloss);

return fixed4(diffuse + specular, 1.0);

//return fixed4(0.0, 1.0, 0.0, 1.0);

}

ENDCG

}

pass

{

Tags { "LightMode" = "ForwardAdd" }

Blend One One

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_fwdadd

#include "UnityCG.cginc"

#include "Lighting.cginc"

#include "AutoLight.cginc"

float \_Gloss;

struct a2v

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct v2f

{

float4 position : SV\_POSITION;

float3 worldNormal : TEXCOORD0;

float4 worldPosition : TEXCOORD1;

};

v2f vert(a2v v)

{

v2f o;

o.position = mul(UNITY\_MATRIX\_MVP, v.vertex);

o.worldNormal = UnityObjectToWorldNormal(v.normal);

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

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

// diffuse

fixed3 worldNormal = normalize(o.worldNormal);

fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(o.worldPosition));

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

// specular

fixed3 worldViewDir = normalize(UnityWorldSpaceViewDir(o.worldPosition));

fixed3 halfViewDir = normalize(worldLightDir + worldViewDir);

fixed3 specular = \_LightColor0.rgb \* pow(saturate(dot(halfViewDir, worldNormal)), \_Gloss);

fixed atten = 1.0;

#ifdef USING\_DIRECTIONAL\_LIGHT

atten = 1.0;

#else

float3 lightCoord = mul(unity\_WorldToLight, o.worldPosition).xyz;

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

#endif

//return fixed4((diffuse + specular) \* 1, 1.0);

return fixed4(1.0, 0.0, 0.0, 1.0);

}

ENDCG

}

}

FallBack "Diffuse"

}

在上面的例子中，ForwardBase渲染路径得到了正常的光照颜色，在ForwardAdd渲染路径中始终返回红色

我们设置了#pragma multi\_compile\_fwdbase与#pragma multi\_compile\_fwdadd，这两个指令可以保证我们在Base Pass与Additional Pass中访问到正确的光照变量。与Base Pass不同的是，我们还使用Blend命令开启和设置了混合模式，如果没有Blend命令的话，Additional Pass会直接覆盖掉之前的光照结果，因为Base Pass只执行一次，而Additional Pass会执行剩下的光源，需要与颜色缓冲区的已有颜色进行混合，同时，由于Additional Pass处理的光源类型可能是平行光、点光源或是聚光灯，因此区分了光源类型。如果是平行光的话衰减值为1.0，对于非平行光会有强度衰减，我们可以使用数学表达式来计算给定点相对于点光源和聚光灯的衰减，但这些计算往往涉及到开根号、除法等计算量较大的操作，因此Unity选择了使用一张纹理作为查找表（Lookup Table，LUT），以在片元着色器中得到光源的衰减。我们首先得到光源空间下的坐标，然后使用该坐标对衰减纹理进行采样得到衰减值。

在之前的表中，我们可以看到

ForwardBase用于前向渲染，该Pass会计算环境光、最重要的平行光、逐顶点/SH光源和Lightmaps

ForwardAdd用于前向渲染，该Pass会计算额外的逐像素光源，每个Pass对应一个光源

对于ForwardBase Pass来说，它处理的逐像素光源类型一定是平行光

也就是说场景中的Pixel光源数为n，则ForwardAdd Pass会被执行n-1次

Light组件的Render Mode决定该光源是顶点光还是像素光

1）在没有任何光照时



2）没有平行光只有一个点光源时

说明此时点光源为ForwardAdd

3）没有平行光只有一个点光源时，且点光源为Not Important

说明ForwardAdd处理的是逐像素光源，逐顶点光源没有效果

4）只有一个绿色平行光时

也就是说此时执行的是ForwardBase

5）只有一个绿色平行光，并且设置该平行光的Render Mode为Not Important时

必须是处理逐像素的平行光？

6）有两个平行光，增加了一个白色的平行光

如果两个的Render Mode都是Not Important



如果两个都设置为Auto或者Important

我们设置绿色的光照强度为1.4，白色的光照强度为0.65



我们设置绿色的光照强度为1.4，白色的光照强度为1



也就是说这两个平行光一个执行了ForwardBase，一个执行了ForwardAdd

7）在添加几个点光源

发现都是执行的ForwardAdd，并且如果将点光源设置为Not Important时点光源都不会有效果，这是因为我们没有再Base Pass中计算逐顶点和SH光源

8）将相机的Rendering Path改为Vertex Lit

即使我们将ForwardBase的像素着色器修改为返回固定的绿色，仍然是这样的显示

也就是说着两类Pass不能在Camera为VertexLit时被渲染，但可以被渲染路径为Forward和Deffered的Camera渲染

可以注意到，Unity处理这些点光源的顺序是按照他们的重要度排序的，但是Unity官方文档中并没有给出光源强度、颜色和距离物体的远近是如何具体影响光源的重要度排序的，我们仅知道排序结果和这三者都有关系。同时，对于场景中的一个物体，如果它不在一个光源的光照范围内，Unity是不会为这个物体调用Pass来处理这个光源的。可以通过Frame Debugger的渲染事件查看

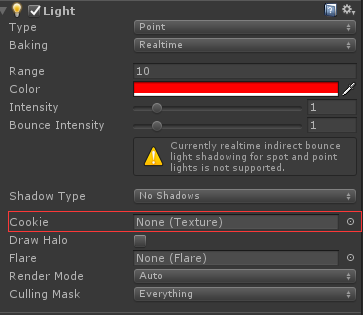
**Unity的光照衰减**

在上面的例子中，我们提到Unity使用一张纹理作为查看表来在片元着色器中计算逐像素光照的衰减。这样做的好处是，计算衰减不依赖于数学公式的复杂性，我们只要使用一个参数值去纹理中采样即可。但使用纹理查看来计算衰减也有一些弊端：

1. 需要预处理得到采样纹理，而且纹理的大小也会影响衰减的精度
2. 不直观，同时也不方便，因此一旦把数据存储到查找表中，我们就无法使用其他数学公式来计算衰减

但由于这种方法可以在一定程度上提升性能，而且得到的效果在大部分情况下都是良好的，因此，Unity默认就是使用这种纹理查找的方式来计算逐像素的点光源和聚光灯的衰减的。Unity在内部使用一张名为\_LightTexture0的纹理来计算光源衰减。需要注意的是，如果我么对该光源使用了cookie，那么衰减查找纹理是\_LightTextureB0。我们通常指关心\_LightTexture0对角线上的纹理颜色值，这些值表明了在光源空间中不同位置的点的衰减值。例如（0,0）点表明了与光源位置重合的点的衰减值，而（1，1）点表明了在光源空间中所关心的距离最远的点的衰减。

为了对\_LightTexture0纹理采样得到给定到该光源的衰减值，我们首先需要得到该点在光源空间中的位置，这是通过\_LightMatrix0变换矩阵得到的，它可以将顶点从世界坐标变换到光源坐标。然后我们可以使用这个坐标的模的平方对衰减纹理进行采样，得到衰减值。



**Unity中的阴影**

阴影的实现原理：

在实时渲染中，我们最常用的是一种名为Shadow Map的技术。这种技术理解起来非常简单，它会首先把摄像机的位置放在与光源重合的位置，那么场景中该光源的阴影区域就是那些摄像机看不到的地方。而Unity就是使用的这种技术。

在前向渲染路径中，如果场景中最重要的平行光开启了阴影，Unity就会为该光源计算它的阴影映射纹理（shadowmap）。这张阴影映射纹理本质上也是一张深度图，它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）。

阴影采样：

1. 传统方法：

正常渲染Pass，计算顶点的光源空间，用xy分量对纹理采样，如果顶点值大于该深度值，就说明在阴影区域

1. Unity5及以后

屏幕空间的阴影映射技术（screenspace shadow map），是在延迟渲染中产生阴影的方法，不过需要显卡支持MRT。根据

阴影映射纹理和深度纹理得到屏幕空间的阴影图。阴影图包含了屏幕空间所有阴影区域

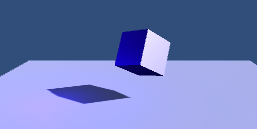
当使用了屏幕空间的阴影映射技术时，

**不透明的物体阴影**

1）让物体投射阴影

将平行光的阴影设置为soft shadows，创建一个Plane，默认该平面会采用Standard的shader，同时Reveive Shadows也是开启的，将上面的ForwardRenderingMat赋值给正方形



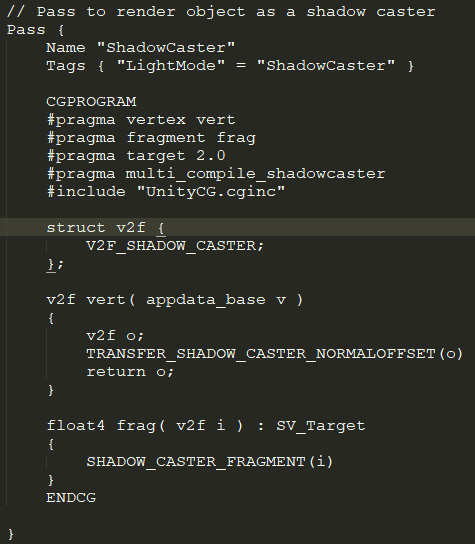


Cast Shadows可以被设置为开启（On）或关闭（Off）。如果开启了Cast Shadow属性，那么Unity就会把该物体加入到光源的阴影映射纹理的计算中，从而让其他物体在对阴影映射纹理采样时可以得到该物体的相关信息。正如之前所说，这个过程是通过为该物体执行LightMode为ShadowCast的Pass来实现的。Reveive Shadows则可以选择是否让物体接受来自其他物体的阴影。如果没有开始Receive Shadows，那么当我们调用Unity的内置宏和变量计算阴影时，这些宏通过判断该物体没有开启接受阴影的功能，就不会在内部为我们计算阴影。

从上面的例子中我们发现尽管我们使用ForwardRendering的Shader没有LightMode=ShadowCast标志，但是正方形仍然可以向下面的平面投射阴影，这是为什么？其秘密就在于FallBack “Specular”,Specular对应于Normal-Glossy.shader，其目录对应于Legacy Shaders/Specular，虽然Specular本身也没有包含这样一个Pass，但是由于它的FallBack调用了VertexLit，它会继续回调，并最终回调到内置的VertexLit。



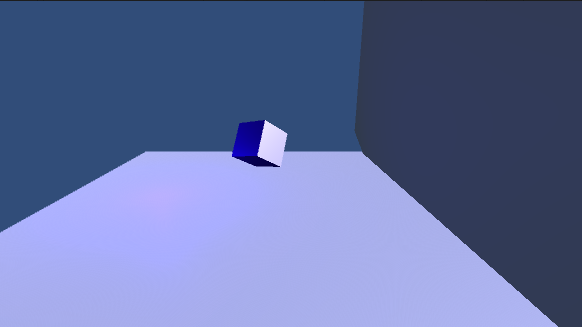
这个shader对应于Normal-VertexLit.shader，我们在这个shader中看到了传说中的Pass标记为ShadowCaster



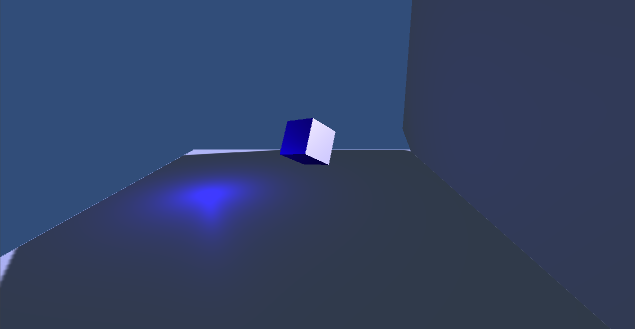
实际上我们改为FallBack “Diffuse”也仍然会有阴影，Diffuse对应Normal-Diffuse.shader文件，Diffuse的Fallback也会对应到Legacy-Shaders/VertexLit

我们看到，上面的代码非常简短，尽管有一些宏和指令使我们之前没有遇到过的，但它们的用处实际上就是为了把深度信息写入渲染目标中。在Unity 5中，这个Pass的渲染目标可以是光源的阴影映射纹理，或是摄像机的深度纹理。如果我们将ForwardRendering.shader中的FallBack去掉就发现没有阴影了（已验证）。当然，我们可以不依赖FallBack，而自行在SubShader中定义自己的LightMode为ShadowCaster的Pass。这种自定义的Pass可以让我们更加灵活地控制阴影的产生。但由于这个Pass的功能通常是可以在多个Unity Shader间通用的，英雌直接FallBack是一个更加方便的用法。

我们在添加一个Plane，会发现新添加右边的Plane并没有在底部的Plane中产生阴影，即使我们给该新的Plane设置Cast Shadows开启：



在默认情况下，我们在计算光源的阴影映射纹理时会剔除掉物体的背面，但对于内置的平面来说，它只有有一个面，因此在本例中当计算阴影映射纹理时，由于右侧的平面的光源空间下么有任何正面，因此就不会添加到阴影映射纹理中。我们可以将Cast Shadows设置为Two Sided来允许对物体的所有面都计算阴影信息。

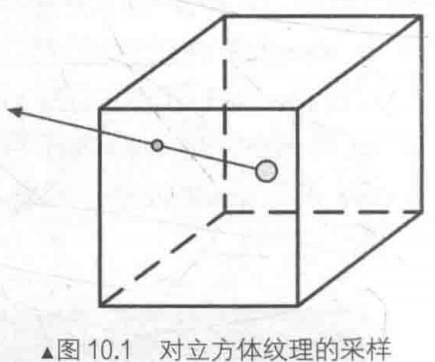


在本例中，最下面的平面之所有可以接受阴影是因为它使用了内置的Standard Shader，而这个内置的Shader进行了接受阴影的相关操作。但由于正方体使用的ForwardRendering并没有对阴影进行任何处理，因此它不会显示出右侧平面投射来的阴影。

2）让物体接受阴影

**立方体纹理**

和之前见到的纹理不同，立方体纹理 一共包含6张图像。立方体的每个面表示沿着世界空间下的轴向（上、下、左、右、前、后）观察所得到的图像。对立方体纹理采样我们需要提供一个三维的纹理坐标，这个三维纹理坐标表示了我们在世界空间下的一个3D方向。这个方向矢量从立方体的中心出发，当它向外部延伸时就会和立方体的6个纹理之一发生相交，而采样得到的结果就是该交点计算而来的，下图展示了采样过程：



立方体纹理在实时渲染中有很多应用，最常见的是用于天空盒子以及环境映射

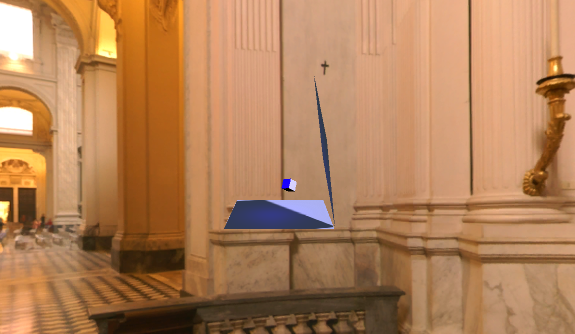
添加天空盒我们有两种方式：

1. 创建一个天空盒材质，将改材质赋值给Window-Lighting的Skybox选项，为了让摄像机正常显示天空盒，我们还需要保证场景的摄像机Camera组件中的Clear Flags被设置为Skybox，这种方式设置的天空盒会应用于该场景中的所有摄像机
2. 如果我们希望某些摄像机可以使用不同的天空盒，可以通过向该摄像机添加Skybox组件来覆盖掉之前的设置

在Unity中，天空盒是在所有不透明物体之后渲染的，而其背后使用的网格是一个立方体或一个细分后的球体

在Camera中添加Skybox组件





我们可以通过这个Skybox修改旋转角度：

void Update()

{

Skybox skybox = gameObject.GetComponent<Skybox>();

float rotation = skybox.material.GetFloat("\_Rotation");

if (skybox.material.HasProperty("\_Rotation"))

{

skybox.material.SetFloat("\_Rotation", rotation + 0.1f);

}

}

**环境映射的立方体纹理**

除了天空盒，立方体纹理最常见的用处是用于环境映射。通过这总方法，我们可以模拟出金属质感的材质。在unity 5中，创建用于环境映射的立方体纹理的方法有三种：第一种方法是直接由一些特殊布局的纹理创建；第二种方法是手动创建一个Cubemap资源，再把6张图赋给它；第三种方法是由脚本生成

如果使用第一种方法，我们需要提供一张具有特殊布局的纹理，例如类似立方体展开图的交叉布局、全景布局等。然后我们只需要把该纹理的Texture Type设置为Cubemap即可，unity会为我们做好剩下的事情。在基于物理的渲染中，我们通常会使用一张HDR图像来生成高质量的Cubemap。

第二种方法是unity 5之前的版本中使用的方法。我们首先需要在项目资源中创建一个Cubemap，然后把6张纹理拖曳到它的面板中。在Unity 5中，官方推荐使用第一种方法创建立方体纹理，这是因为第一种方法可以对纹理数据进行压缩，而且可以支持边缘修正、光滑反射和HDR等功能。

Shader "Unlit/Refection"

{

Properties

{

\_Color ("Color Tint", Color) = (1.0, 1.0, 1.0, 1.0)

\_ReflectColor ("Reflection Color", Color) = (1.0, 1.0, 1.0, 1.0)

\_ReflectAmount ("Reflect Amount", Range(0, 1)) = 1

\_Cubemap ("Reflection Cubemap", Cube) = "\_Skybox" {}

}

SubShader

{

Pass

{

Tags { "LightMode" = "ForwardBase" }

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

#include "Lighting.cginc"

struct a2v

{

float4 vertex : POSITION;

float4 texcoord : TEXCOORD0;

float3 normal : NORMAL;

};

struct v2f

{

float4 pos : SV\_POSITION;

float3 worldNormal : TEXCOORD0;

float4 worldPosition : TEXCOORD1;

float3 worldViewDir : TEXCOORD2;

float3 worldRef1 : TEXCOORD3;

};

fixed4 \_Color;

fixed4 \_ReflectColor;

float \_ReflectAmount;

samplerCUBE \_Cubemap;

v2f vert(a2v v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.worldNormal = UnityObjectToWorldNormal(v.normal);

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

o.worldViewDir = UnityWorldSpaceViewDir(o.worldPosition);

o.worldRef1 = reflect(-o.worldViewDir, o.worldNormal);

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

fixed3 worldNormal = normalize(o.worldNormal);

fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(o.worldPosition));

fixed3 worldViewDir = normalize(o.worldViewDir);

fixed3 ambient = UNITY\_LIGHTMODEL\_AMBIENT.xyz;

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

fixed3 reflection = texCUBE(\_Cubemap, o.worldRef1).rgb \* \_ReflectColor.rgb;

return fixed4(ambient + lerp(diffuse, reflection, \_ReflectAmount), 1.0);

}

ENDCG

}

}

FallBack "Diffuse"

}



对于立方体纹理的采样需要使用CG的texCUBE函数。注意到，在上面的计算中，我们在采样时并没有与oworldRef1进行归一化处理。这是因为，用于采样的参数仅仅是作为方向变量传递给texCUBE函数的，因此我们没有必要进行一次归一化操作。然后我们使用\_ReflectAmount来混合漫反射颜色与反射颜色，并和环境光照相加都返回。