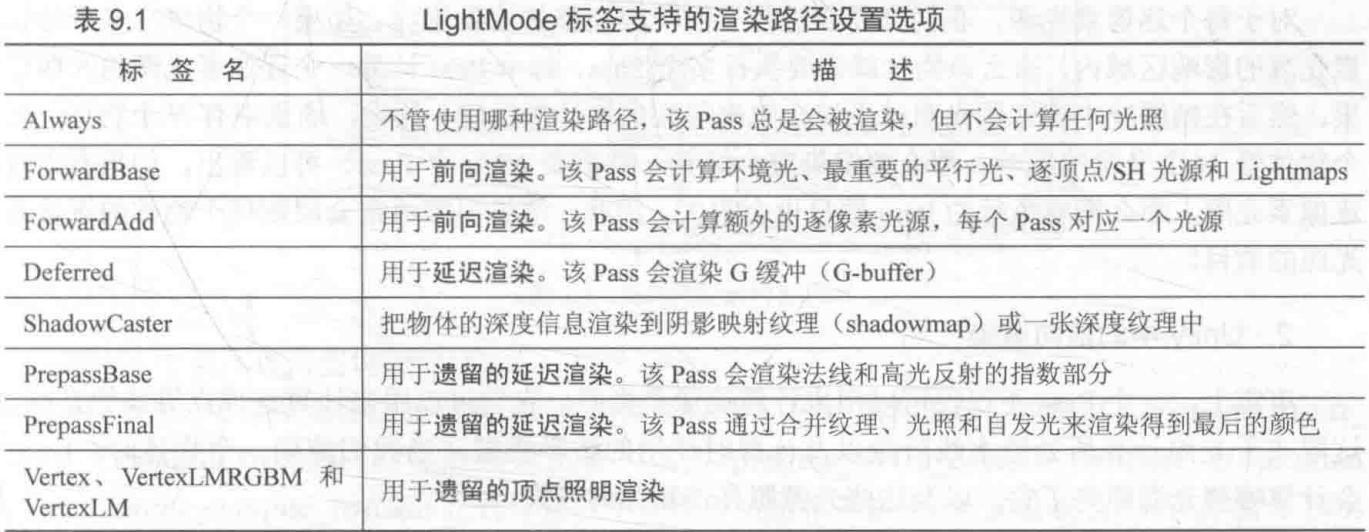
**Unity的渲染路径**

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）

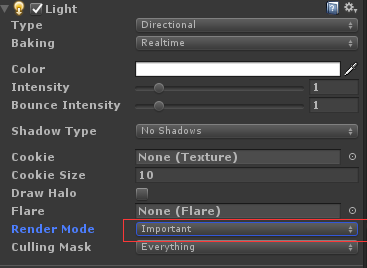
**需要注意的是，如果当前的显卡并不支持所渲染的渲染路径，Unity会自动使用更低一级的渲染路径。例如，如果一个GPU不支持延迟渲染，那么Unity就会使用前向渲染。在Unity 5.x中如果使用了前向渲染又没有为Pass指定任何前向渲染适合的标签，就会被当成一个和顶点照明渲染路径等同的Pass，那么一些光照变量很可能不会被正确赋值，我们计算出的效果也很可能有错误**



**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

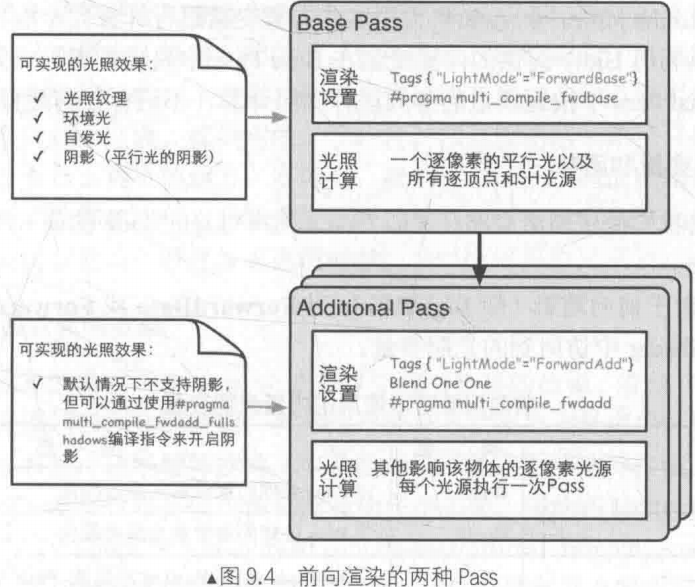
4）如果根据以上规则得到的逐像素光源数量小于Quality Setting中的逐像素光源数量（Pixel Light Count），会有更多的光源以逐像素的方式进行渲染

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：



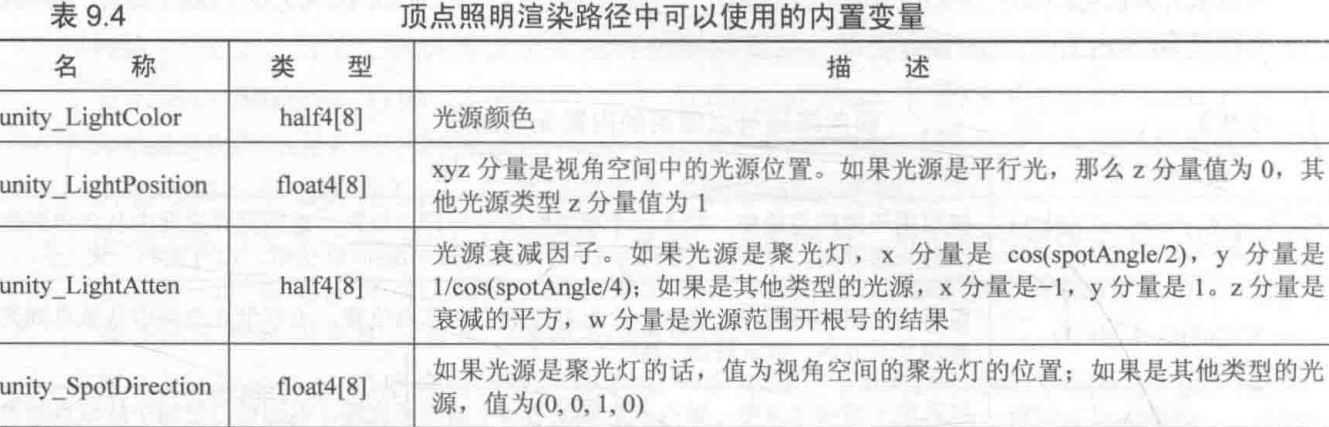
对于上图有几点需要说明的地方：

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，Base Pass总是会执行一次

**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，当发现一个片元是可见的，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

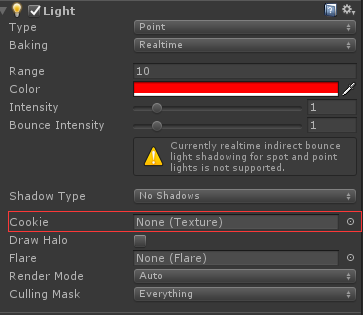
**Unity的光照衰减**

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）。

那么，在计算阴影映射纹理时，我们如何判定距离它最近的表面位置呢？一种方法是，先把摄像机放置到光源位置上，然后按照正常的渲染流程，即调用Base Pass和Additional Pass来更新深度信息，得到阴影映射纹理，但这种方法会对性能造成一定的浪费，因为我们实际上仅仅需要深度信息而已，而Base Pass和Addition Pass往往涉及到很多复杂的光照模型计算。因此，unity选择使用一个额外的Pass来专门更新光源的阴影映射纹理，这个Pass就是LightMode标签被设置为ShadowCaster的Pass，这个Pass的渲染目标不是帧缓存，而是阴影映射纹理（或深度纹理）。Unity首先把摄像机放置到光源的位置上，然后调用该Pass，通过对顶点变换后得到的光源空间下的位置，并据此来输出深度信息到阴影映射纹理中。因此，当开启了光源的阴影效果后，底层渲染引擎首先会在当前渲染物体的Unity Shader中找到LightMode为ShadowCaster的Pass，如果没有，它就会在FallBack指定的Unity Shader中继续寻找，如果仍然没有找到，该物体就无法向其他物体投射阴影（但它仍然可以接受来自其他物体的阴影）。当找到了一个LightMode为ShadowCaster的Pass后，Unity会使用该Pass来更新光源的阴影映射纹理。

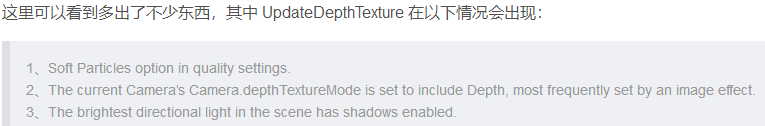
在传统的阴影映射纹理实现中，我们会在正常渲染的Pass中把顶点位置变换到光源空间下，以得到它在光源空间中的三维位置信息，然后我们使用xy分量对阴影映射纹理进行采样，得到阴影映射纹理中该位置的深度信息，如果该深度值小于该顶点的深度值（通常由z分量得到），那么说明该点位于阴影中。但在Unity 5中，Unity使用了不同于这种传统的阴影采样技术，即屏幕空间的阴影映射技术（Screenspace Shadow Map）。屏幕空间的阴影映射原本是延迟渲染中产生阴影的方法，需要注意的是，并不是所有的平台unity都会使用这种技术，这是因为，屏幕空间的阴影映射需要显卡支持MRT，而有些移动平台不支持这种特性。

当使用了屏幕空间的阴影映射技术时，Unity首先会通过调用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，根据光源的阴影映射纹理和摄像机的深度纹理来得到屏幕空间的阴影图。如果摄像机的深度图中记录的表面深度大于转换到阴影映射纹理中的深度值，就说明该表面虽然是可见的，但是却处于该光源的阴影中。通过这种方法，阴影图就包含了屏幕空间中所有阴影的区域，如果我们想要一个物体接受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中，然后使用这个坐标对阴影图进行采样即可。

总结：

如果我们想要一个物体接受来自其他物体的阴影，就必须在Shader中对阴影映射纹理（包括屏幕空间的阴影图）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

如果我们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，在unity中，这个过程就是通过为该物体执行LightMode为ShadowCaster的Pass来实现的。如果使用了屏幕空间的投影技术，Unity还会使用这个Pass产生一张摄像机的深度纹理。



屏幕空间的阴影

**这种技术只适合平行光，对于点光源和聚光灯只是生成了一张阴影贴图**

**UpdateDepthTexture：更新摄像机的深度纹理**

**RenderShadowmap：渲染得到平行光的阴影映射纹理**

**CollectShadows：根据深度纹理和阴影映射纹理得到屏幕空间的阴影图**

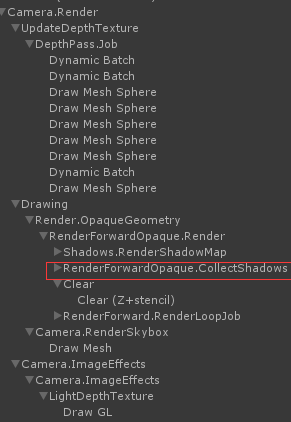
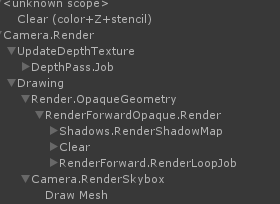
延迟渲染中的光照计算绝大部分都是在屏幕空间里进行的，同样也包括阴影。这种屏幕空间的阴影实现主要有这么几个步骤：

1）首先得到从当前摄像机处观察到的深度纹理。在延迟渲染里这张深度图本来就有，如果是前向渲染的话就需要把场景整个渲染一遍，把深度渲染到深度图中。

2）然后再从光源出发得到从该光源处观察到的深度纹理，也被称为这个光源的ShadowMap。

3）然后在屏幕空间做一次阴影收集计算（Shadows Collector），这次计算会得到一张屏幕空间阴影纹理，也就是说这张图里面需要有阴影的部分已经显示在图上了。这个过程概括来说就是把每一个像素根据它在摄像机深度纹理中的深度值得到世界空间坐标，再把它的坐标从世界空间转换到光源空间中，和光源的ShadowMap里面的深度值对比，如果大于ShadowMap中的深度距离，那么就说明光源无法照到，在阴影内。

4）最后，在正常渲染物体为它计算阴影的时候，只需要按照当前处理的fragment在屏幕空间中的位置对步骤3得到的屏幕空间阴影图采样就可以了。

当项目工程的目标平台是Mobile的时候，就不会使用屏幕空间的阴影映射技术，即使用原始的Shadow Map方法，在代码中会定义内置宏UNITY\_NO\_SCREENSPACE\_SHADOW来控制。下面两张图可以看到是否使用了屏幕空间阴影技术：  


可以发现，在屏幕空间的阴影下会有UpdateDepthTexture生成深度图，光源方向的深度图，以及屏幕阴影图，而不用这种技术只会生成光源方向的深度图

此时LightMode为ShadowCaster的Pass会同时影响阴影投射和阴影接收的效果。具体来说，不管是希望物体能够向其他

物体投射阴影还是自己接收来自其他物体的阴影，都需要定义LightMode为ShadowCaster的Pass。

因为对于投射阴影来说，需要使用LightMode为ShadowCaster的Pass来生成该光源的阴影映射纹理，这样如果它距离光源更近的话就会记录到阴影映射纹理中；对于接收阴影来说，也需要使用这个Pass来生成屏幕空间的深度纹理，从而可以在计算屏幕空间的阴影纹理时，可以据此来判断该点是否在阴影中。简单来说，因为阴影纹理是一张在正常渲染物体前就生成好的，因此\*\*无论是阴影投射还是阴影接收，都需要定义LightMode为ShadowCaster的Pass\*\*。也就是说在屏幕空间阴影技术下，摄像机的深度图与光源的深度图都是通过这个ShadowCaster Pass完成的



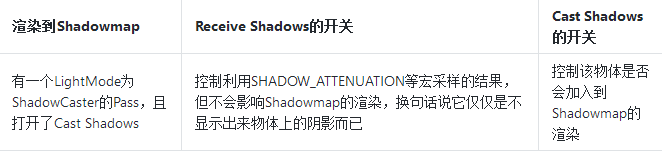
如果要产生阴影必须要放到shadowmap中，是否放到depth texture并不是必要的

如果要接受阴影必须要放到depth texture中，是否放到shadowmap并不是必要的



传统的shadow map

由于不会提前渲染Depth Texture，此时LightMode为ShadowCaster的Pass仅会影响阴影投射的效果。对于需要接收阴影的物体来说，由于它们会在普通的Pass中进行空间转换和判断是否在光源阴影内的计算，而不是提前生成的信息，因此它不需要定义LightMode为ShadowCaster的Pass也可以正确接收到其他物体的阴影。





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

Shader "Unlit/TestShader"

{

Properties

{

\_Gloss ("Gloss", Range(8.0, 256.0)) = 20.0

}

SubShader

{

Tags { "Queue" = "Geometry" }

Pass

{

Tags { "LightMode" = "ForwardBase" }

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_fwdbase // 注意没有这句话就不会接受阴影

#include "UnityCG.cginc"

#include "AutoLight.cginc"

#include "Lighting.cginc"

struct a2v

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct v2f

{

float4 pos : SV\_POSITION;

float3 worldNormal : TEXCOORD0;

float3 worldPos : TEXCOORD1;

SHADOW\_COORDS(2)

};

float \_Gloss;

v2f vert (a2v v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.worldNormal = UnityObjectToWorldNormal(v.normal);

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

TRANSFER\_SHADOW(o);

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

float3 worldNormal = normalize(i.worldNormal);

fixed3 ambient = UNITY\_LIGHTMODEL\_AMBIENT.xyz;

fixed3 diffuse = \_LightColor0.rgb \* saturate(dot(worldNormal, UnityWorldSpaceLightDir(i.worldPos)));

fixed shadow = SHADOW\_ATTENUATION(i);

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

}

ENDCG

}

// 产生光源方向深度贴图或者摄像机方向的深度贴图pass

Pass

{

Tags { "LightMode" = "ShadowCaster" }

CGPROGRAM

#pragma target 3.0

#pragma vertex MyShadowVertexProgram

#pragma fragment MyShadowFragmentProgram

#include "UnityCG.cginc"

struct VertexData

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct v2f

{

V2F\_SHADOW\_CASTER;

};

v2f MyShadowVertexProgram(VertexData v)

{

v2f o;

//float4 position = UnityClipSpaceShadowCasterPos(v.vertex, v.normal);

//o.pos = UnityApplyLinearShadowBias(position);

TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET(o);

return o;

}

fixed4 MyShadowFragmentProgram(v2f i) : SV\_Target

{

SHADOW\_CASTER\_FRAGMENT(i);

}

ENDCG

}

}

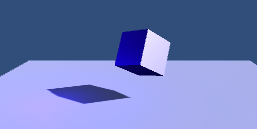
}

**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



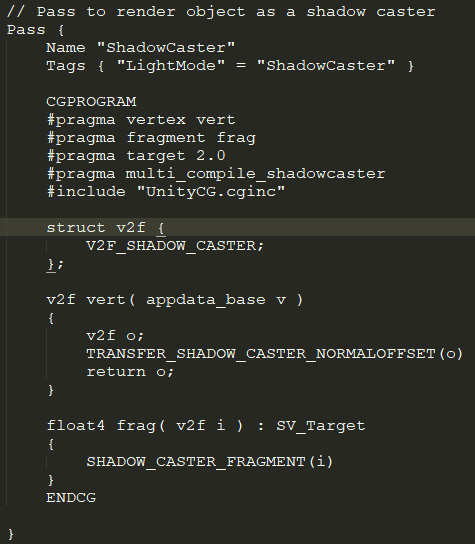


Cast Shadows可以被设置为开启（On）或关闭（Off）。如果开启了Cast Shadow属性，那么Unity就会把该物体加入到光源的阴影映射纹理的计算中，从而让其他物体在对阴影映射纹理采样时可以得到该物体的相关信息。正如之前所说，这个过程是通过为该物体执行LightMode为ShadowCast的Pass来实现的。Reveive Shadows则可以选择是否让物体接受来自其他物体的阴影。如果没有开始Receive Shadows，那么当我们调用Unity的内置宏和变量计算阴影时，这些宏通过判断该物体没有开启接受阴影的功能，就不会在内部为我们计算阴影。

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。



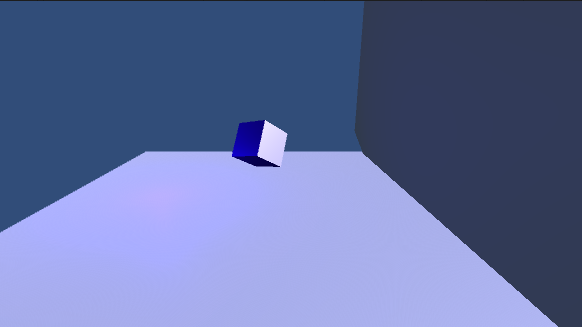
这个shader对应于Normal-VertexLit.shader，我们在这个shader中看到了传说中的Pass标记为ShadowCaster



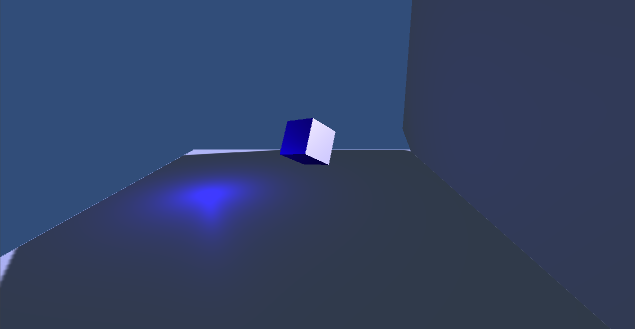
实际上我们改为FallBack “Diffuse”也仍然会有阴影，Diffuse对应Normal-Diffuse.shader文件，Diffuse的Fallback也会对应到Legacy-Shaders/VertexLit

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，因此直接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。

**统一管理光照衰减和阴影**

在base pass中，平行光的衰减因子总是等于1，而在Additional Pass中，我们需要判断该Pass处理的光源类型，再使用内置变量和宏计算衰减因子。实际上，光照衰减和阴影对物体最终的渲染结果的影响本质上是相同的---我们都是把光照衰减因子和阴影值以及光照结果相乘得到最终的渲染结果，那么，是不是可以有一个方法可以同时计算两个信息呢？好消息是，unity在shader里提供了这样的功能，这主要是通过内置的UNITY\_LIGHT\_ATTENUATION宏来实现的。

fixed4 frag (v2f i) : SV\_Target

{

float3 worldNormal = normalize(i.worldNormal);

fixed3 diffuse = \_LightColor0.rgb \* saturate(dot(worldNormal, UnityWorldSpaceLightDir(i.worldPos)));

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

return fixed4(diffuse \* atten , 1.0);

}

ENDCG

UNITY\_LIGHT\_ATTENUATION是unity内置的用于计算光照衰减和阴影的宏，我们可以在内置的AutoLight.cginc里找到它的声明，它接受3个参数，它会将光照衰减和阴影值相乘后的结果存储在第一个参数中。

我们看看点光源的定义：

#ifdef POINT

sampler2D \_LightTexture0;

unityShadowCoord4x4 unity\_WorldToLight;

#define UNITY\_LIGHT\_ATTENUATION(destName, input, worldPos) \

unityShadowCoord3 lightCoord = mul(unity\_WorldToLight, unityShadowCoord4(worldPos, 1)).xyz; \

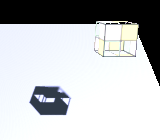
fixed destName = (tex2D(\_LightTexture0, dot(lightCoord, lightCoord).rr).UNITY\_ATTEN\_CHANNEL \* **SHADOW\_ATTENUATION(input)**);

#endif

**透明物体的阴影**

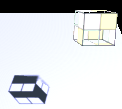
对于大多数不透明物体来说，把FallBack设置为VertexLit就可以得到正确的阴影，但对于透明物体来说，我们就需要小心处理它的阴影。透明物体的实现通常会使用透明度测试或透明度混合，我们需要小心设置这些物体的FallBack

透明度测试的处理比较简单，但如果我们仍然直接使用VertexList、Diffuse、Specular等作为回调，往往无法得到正确的阴影，这是因为透明度测试需要在片元着色器中舍弃某些片元，而VertexLit中的阴影并没有进行这样的操作。为了让使用透明度测试的物体得到正确的阴影效果，我们可以设置FallBack为FallBack "Transparent/Cutout/VertexLit"

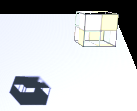


但是，这样的结果仍然有一些问题，比如出现了一些不应该透过光的部分，出现这种情况的原因是，默认情况下把物体渲染到深度图和阴影映射纹理中仅考虑物体的正面，但对于上面的立方体来说，由于一些面完全背对光源，因此这些面的深度信息没有计入到阴影映射纹理的计算中，为了得到正确的结果，我们可以将正方形的Mesh Render组件中的Cast Shadows属性设置为Two Sided，强制Unity在计算阴影映射纹理时机计算所有面的深度信息。









我们也可以写一个ShadowCater的Pass

Pass

{

Tags { "LightMode" = "ShadowCaster" }

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

sampler2D \_MainTex;

float4 \_MainTex\_ST;

float \_Cutoff;

struct v2f

{

V2F\_SHADOW\_CASTER;

float2 uv : TEXCOORD0;

};

v2f vert(appdata\_base v)

{

v2f o;

o.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);

TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET(o);

return o;

}

fixed4 frag(v2f i) : SV\_Target

{

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

clip(albedo.a - \_Cutoff);

SHADOW\_CASTER\_FRAGMENT(i);

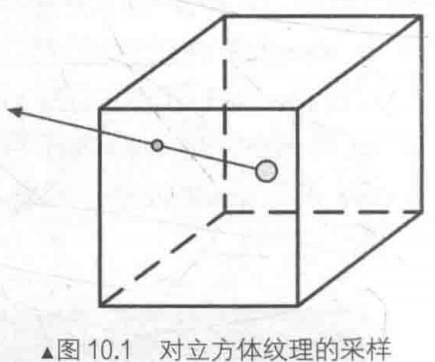
}

ENDCG

}

**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。

1）反射：

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。

2）折射

通常来说，当得到折射方向后我们就会直接使用它来对立方体纹理进行采样，但是这是不符合物理规律的。对于一个透明物体来说，一种更准确的模拟方法需要计算两次折射—---一次是当光线进入它的内部时，而另一次则是从它内部射出时。但是要实时渲染第二次折射方向是比较复杂的，而且仅仅模拟一次得到的效果也可以接受，因此在实时渲染中我们通常仅模拟第一次折射。

Shader "Unlit/Refaction"

{

Properties

{

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

\_RefactionColor ("Refaction Color", Color) = (1.0, 1.0, 1.0, 1.0)

\_RefactionAmount ("Refaction Amount", Range(0.0, 1.0)) = 0.5

\_RefactionRatio ("Refaction Rate", Range(0.1, 1.0)) = 0.5

\_CubeMap ("Cube Map", Cube) = "\_Skybox" {}

}

SubShader

{

Tags { "Queue" = "Geometry" }

Pass

{

Tags { "LightMode" = "ForwardBase" }

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

#include "Lighting.cginc"

struct a2v

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct v2f

{

float4 pos : SV\_POSITION;

float4 worldPosition : TEXCOORD0;

float3 worldNormal : TEXCOORD1;

};

fixed4 \_Color;

fixed4 \_RefactionColor;

float \_RefactionAmount;

float \_RefactionRatio;

samplerCUBE \_CubeMap;

v2f vert(a2v v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

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

o.worldNormal = UnityObjectToWorldNormal(v.normal);

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

fixed3 worldNormal = normalize(o.worldNormal);

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

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

fixed3 ambient = UNITY\_LIGHTMODEL\_AMBIENT.rgb;

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

fixed3 refactionDir = refract(-worldViewDir, worldNormal, \_RefactionRatio);

fixed3 refaction = texCUBE(\_CubeMap, refactionDir) \* \_RefactionColor.rgb;

return fixed4(ambient + lerp(diffuse, refaction, \_RefactionAmount), 1.0);

}

ENDCG

}

}

FallBack "Diffuse"

}



**菲涅尔反射**

菲涅尔反射描述了一种光学现象，即当光线照射到物体表面上时，一部分发生发射，一部分进入物体内部发生折射或散射。被反射的光和入射光之间存在一定的比率关系，这个比率关系可以通过菲涅尔等式进行计算。一个例子是，站在湖边，直接低头看水面会发现几乎是透明的，但是看远处的水面，就只能看到水面反射的环境。事实上，几乎任何物体都或多或少包含了菲涅尔效果，这是基于物体的渲染中非常重要的一项高光反射计算因子。那么，我们如何极端菲涅尔反射呢？这就需要使用菲尼尔等式。真实世界的菲涅尔等式是非常复杂的，在实时渲染中，我们通常会使用一些近似公式来计算，其中一个著名的近似公式就是Schlick 菲涅尔近似等式：



其中，F0是一个反射系数，同于控制菲涅尔反射的强度，V是视角方向，n是表面法线。从这个公式可以看出看的角度和点的法线越垂直，反射现象越明显

另外一个应用比较广泛的等式是Empricial 菲涅尔近似等式：



其中，bias、scale和power是控制项。

Shader "Unlit/Fresnel"

{

Properties

{

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

\_Fresnel ("Fresnel Scale", Range(0.0, 1.0)) = 0.5

\_Gloss ("Gloss", Range(8.0, 256.0)) = 20.0

\_Cubemap ("Cubemap", Cube) = "\_SkyBox" {}

}

SubShader

{

Tags { "LightMode" = "ForwardBase" }

pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

#include "Lighting.cginc"

struct a2v

{

float4 vertex : POSITION;

float3 normal : NORMAL;

};

struct v2f

{

float4 pos : SV\_POSITION;

float4 worldPosition : TEXCOORD0;

float3 worldNormal : TEXCOORD1;

float3 worldLightDir : TEXCOORD2;

float3 worldViewDir : TEXCOORD3;

float3 worldReflectDir : TEXCOORD4;

};

fixed4 \_Color;

float \_Fresnel;

float \_Gloss;

samplerCUBE \_Cubemap;

v2f vert(a2v v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

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

o.worldNormal = UnityObjectToWorldNormal(v.normal);

o.worldLightDir = UnityWorldSpaceLightDir(o.worldPosition);

o.worldViewDir = UnityWorldSpaceViewDir(o.worldPosition);

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

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

fixed3 worldNormal = normalize(o.worldNormal);

fixed3 worldLightDir = normalize(o.worldLightDir);

fixed3 worldViewDir = normalize(o.worldViewDir);

// ambient

fixed3 ambient = \_Color.rgb \* UNITY\_LIGHTMODEL\_AMBIENT.rgb;

// diffuse

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

// specular

//fixed3 halfDir = normalize(worldLightDir + worldViewDir);

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

float3 reflect = texCUBE(\_Cubemap, o.worldReflectDir).rgb;

fixed fresnel = \_Fresnel + (1 - \_Fresnel) \* pow(1 - dot(worldViewDir, worldNormal), 5);

return fixed4(ambient + lerp(diffuse, reflect, saturate(fresnel)), 1.0);

}

ENDCG

}

}

FallBack "Diffuse"

}

\_Fresnel=0.48



\_Fresnel=0



\_Fresnel=1



**渲染纹理**

Unity为渲染目标纹理定义了一种专门的纹理类型---渲染纹理（Render Texture）。在Unity中使用渲染纹理通常有两种方式：

1. 在Project目录下创建一个渲染纹理，然后把某个这相机的渲染目标设置成该渲染纹理，这样一来该摄像机的渲染结果就会实时更新到渲染纹理中，而不会显示在屏幕上。使用这种方法，我们还可以选择渲染纹理的分辨率、滤波模式等纹理属性。
2. 在屏幕后处理时使用GrabPass命令或OnRenderImage函数来获取当前屏幕图像，Unity会把这个屏幕图像放到一张和屏幕分辨率等同的渲染纹理中，我们可以在自定义的Pass中把它们当成普通的纹理来处理，从而实现各种屏幕特效。

**镜子效果**

原理：另外取一个camera,创建一个RenderTexture，将这个Camera的场景渲染到RenderTexture中，然后将这个RenderTexture给到镜子的采样纹理中即可，注意镜子的左右颠倒

Shader "Unlit/Mirror"

{

Properties

{

\_MainTex ("Main Tex", 2D) = "white" {}

}

SubShader

{

Tags { "LightMode" = "ForwardBase" }

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

#include "Lighting.cginc"

struct a2v

{

float4 vertex : POSITION;

float4 texcoord : TEXCOORD0;

};

struct v2f

{

float4 pos : SV\_POSITION;

float4 uv : TEXCOORD0;

};

sampler2D \_MainTex;

v2f vert(a2v v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = v.texcoord;

o.uv.x = 1 - o.uv.x;

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

return tex2D(\_MainTex, o.uv);

}

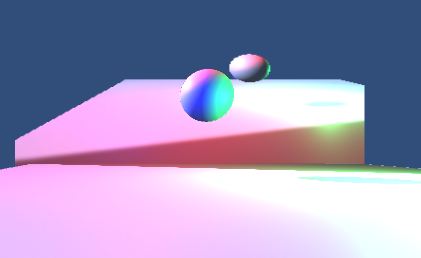
ENDCG

}

}

FallBack "Diffuse"

}

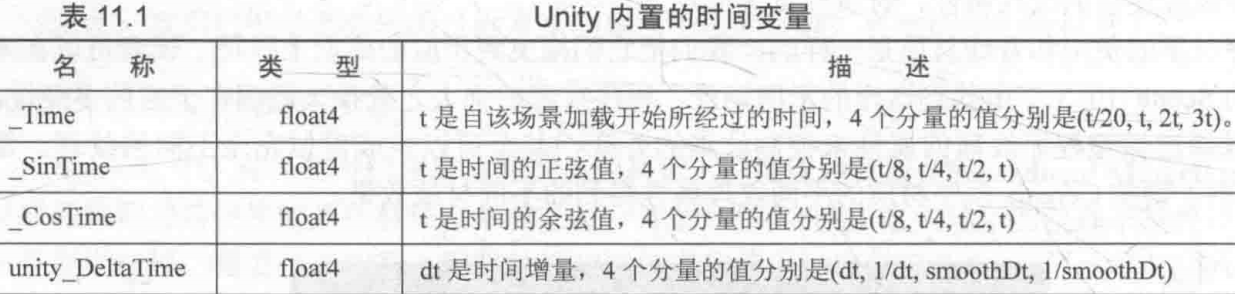


**玻璃效果**

在Unity中，我们还可以在Unity Shader中使用一种特殊的Pass来完成获取屏幕图像的目的，这就是GrabPass。

**让画面动起来**

Unity提供了一些内置的时间变量



**纹理动画**

Shader "Unlit/ImageSequenceAnimation"

{

Properties

{

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

\_MainTex ("Image Sequence", 2D) = "white" {}

\_HorizontalAmount ("Horizontal Amount", Float) = 4

\_VerticalAmount ("Vertical Amount", Float) = 4

\_Speed ("Speed", Range(1.0, 100.0)) = 30.0

}

SubShader

{

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

Pass

{

Tags { "LightMode" = "ForwardBase" }

ZWrite Off

Blend SrcAlpha OneMinusSrcAlpha

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

struct a2v

{

float4 vertex : POSITION;

float4 texcoord : TEXCOORD0;

};

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

};

fixed4 \_Color;

sampler2D \_MainTex;

float4 \_MainTex\_ST;

float \_HorizontalAmount;

float \_VerticalAmount;

float \_Speed;

v2f vert(a2v v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

float time = floor(\_Time.y \* \_Speed);

float row = floor(time / \_HorizontalAmount);

float column = time - row \* \_VerticalAmount;

half2 uv = o.uv + half2(column, -row);

uv.x /= \_HorizontalAmount;

uv.y /= \_VerticalAmount;

fixed4 c = tex2D(\_MainTex, uv);

c.rgba \*= \_Color;

return c;

}

ENDCG

}

}

FallBack "Transparent/VertexLit"

}

由于是透明纹理，因此需要勾选该纹理的Alpha Is Transparency属性



**滚动背景**

Shader "Unlit/ScrollingBackground"

{

Properties

{

\_MainTex ("Base Layer (RGB)", 2D) = "white" {}

\_DetailTex ("2nd Layer (RGB)", 2D) = "white" {}

\_ScrollX ("Base Layer Scroll Speed", Range(0.1, 1.0)) = 1.0

\_Scroll2X ("2nd Layer Scroll Speed", Range(0.1, 1.0)) = 1.0

\_Multiplier ("Layer Multiplier", Range(0.1, 1.0)) = 1.0

}

SubShader

{

Pass

{

Tags { "LightMode" = "ForwardBase" }

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

struct a2v

{

float4 vertex : POSITION;

float4 texcoord : TEXCOORD0;

};

struct v2f

{

float4 pos : SV\_POSITION;

float4 uv : TEXCOORD0;

};

sampler2D \_MainTex;

float4 \_MainTex\_ST;

sampler2D \_DetailTex;

float4 \_DetailTex\_ST;

float \_ScrollX;

float \_Scroll2X;

float \_Multiplier;

v2f vert(a2v v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.uv.xy = TRANSFORM\_TEX(v.texcoord, \_MainTex) + fixed2(frac(\_ScrollX \* \_Time.y), 0.0);

o.uv.zw = TRANSFORM\_TEX(v.texcoord, \_DetailTex) + fixed2(frac(\_Scroll2X \* \_Time.y), 0.0);

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

fixed4 firstLayer = tex2D(\_MainTex, o.uv.xy);

fixed4 secondLayer = tex2D(\_DetailTex, o.uv.zw);

fixed4 c = lerp(firstLayer, secondLayer, secondLayer.a);

c.rgb \*= \_Multiplier;

return c;

}

ENDCG

}

}

FallBack "Diffuse"

}



上面的例子有几个地方需要注意：

1. frac函数是用于截取小数部分
2. lerp(firstLayer, secondLayer, secondLayer.a)这里实际上是实现了混合
3. 图片的Wrap Mode必须为Repeat

**顶点动画**

流动的河流

Shader "Unlit/Water"

{

Properties

{

\_MainTex ("Main Tex", 2D) = "white" {}

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

\_Magnitude ("Distortion Magnitude", Range(0.0, 10.0)) = 1

\_Frequency ("Distortion Frequency", Range(0.0, 10.0)) = 1.0

\_InvWaveLength ("Distortion Inverse Wave Length", Range(0.0, 100.0)) = 10

\_Speed ("Speed", Float) = 0.5

}

SubShader

{

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

Pass

{

Tags { "LightMode" = "ForwardBase" }

ZWrite Off

Blend SrcAlpha OneMinusSrcAlpha

Cull Off

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

struct a2v

{

float4 vertex : POSITION;

float4 texcoord : TEXCOORD0;

};

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

};

sampler2D \_MainTex;

float4 \_MainTex\_ST;

fixed4 \_Color;

float \_Magnitude;

float \_Frequency;

float \_InvWaveLength;

float \_Speed;

v2f vert(a2v v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

float4 offset;

offset.yzw = float3(0.0, 0.0, 0.0);

offset.x = \_Magnitude \* sin(\_Frequency \* \_Time.y + v.vertex.x \* \_InvWaveLength +

v.vertex.y \* \_InvWaveLength + v.vertex.z \* \_InvWaveLength);

o.pos = UnityObjectToClipPos(v.vertex + offset);

o.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);

o.uv += float2(0.0, \_Time.y \* \_Speed);

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

fixed4 c = tex2D(\_MainTex, o.uv);

c.rgb \*= \_Color.rgb;

return c;

}

ENDCG

}

}

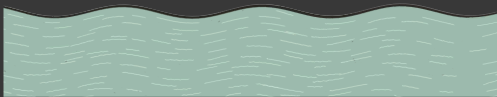
FallBack "Transparent/VertexLit"

}

其中\_MainTex是河流的纹理，\_Color用于控制整体颜色，\_Magnitude用于控制水流波动的幅度，\_Frequency用于控制波动频率，\_InvWaveLength用于控制波长的倒数（\_InvWaveLength越大，波长越小），\_Speed用于控制河流纹理的移动速度

在上面的例子中，我们设置了一个新的标签---DisableBatching。一些SubShader在使用Unity的批处理功能时会出现纹理，这时可以通过该标签来直接指明是否对该SubShader使用批处理。而这些需要特殊处理的Shader通常就是包含了模型空间的顶点动画的Shader。这时因为，批处理会合并所有相关的模型，而这些模型各自的模型空间就会丢失，在本例中，我们需要在物体的模型空间下对顶点位置进行偏移。因此，在这里需要取消对该Shader的批处理操作。

在上面的例子中，我们使用Cull Off，如果不使用，背面就看不见了（已证实）

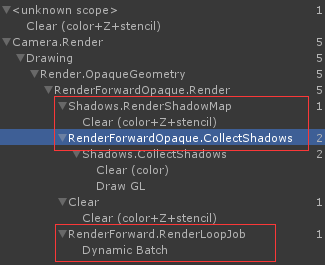


广告牌

注意事项：

首先，如果我们在模型空间下进行了一些顶点动画，那么批处理往往就会破坏这种动画效果。这时我们可以通过SubShader的DisableBatching标签来强制取消对该Unity Shader的批处理。然而，取消批处理会带来一定的性能下降，增加了Draw Call，因此我们应该尽量避免使用模型空间下的一些绝对位置和方向来进行计算。我们还可以利用顶点颜色来存储每个顶点到锚点的距离值，可以避免显式使用模型空间的中心来作为锚点

其次，如果我们想要对包含了顶点动画的物体添加阴影，那么如果仍然使用默认的ShadowCaster，则得不到正确的阴影效果（这里指的是无法向其他物体正确地投影阴影），这是因为内置的ShadowCaster没有进行相关的顶点动画，因此，unity仍然按照原来的顶点位置来计算阴影，这个从Frame Debugger可以看到，是先收集阴影了然后再进行所有物体的渲染



**屏幕后处理效果**

想要实现屏幕后处理的基础在于得到渲染后的屏幕图像，即抓取屏幕，而Unity为我们提供了这样一个方便的接口—OnRenderImage函数，它的函数声明如下：

MonoBehaviour.OnRenderImage(RenderTexture src, RenderTexture dest)

当我们在脚本中声明此函数后，Unity会把当前渲染得到的图像存储在第一个参数对应的源渲染纹理中，通过函数中一系列操作后，再把目标渲染纹理，即第二个参数对应的额渲染纹理显示到屏幕上。在OnRenderImage函数中，我们通常是利用Graphics.Blit函数来完成对渲染纹理的处理，它有3中函数声明：

Public static void Blit(Texture src, RenderTexture dest)

Public static void Blit(Texture src, RenderTexture dest, Material mat, int pass = -1)

Public static void Blit(Texture src, Material mat, int pass = -1)

其中，参数src对应了源纹理，在屏幕后处理技术中，这个参数通常就是当前屏幕的渲染纹理或是上一步处理后得到的渲染纹理。参数dest是目标渲染纹理，如果它的值为null就会直接将结果显示在屏幕上。参数mat是我们使用的材质，这个材质使用的Unity Shader将会进行各种屏幕后处理操作，，而src纹理将会被传递给Shader中名为\_MainTex的纹理属性。参数pass的默认值为-1，表示将会一次调用Shader内的所有Pass。否则，只会调用给定索引的Pass

在默认情况下，OnRenderImage函数会在所有的不透明和透明的Pass执行完毕后被调用，以便对场景中所有游戏对象都产生影响。但有时，我们希望在不透明的Pass（即渲染队列小于等于2500的Pass，内置的Background、Geometry和AlphaTest渲染队列均在此范围内）执行完毕后立即调用OnRenderImage函数，从而不对透明物体产生任何影响。此时我们可以在OnRenderImage函数前添加ImageEffectOpaque属性来实现这样的目的。

因此，要在Unity中实现屏幕后处理效果，过程通常如下：我们首先需要在摄像中添加一个用于屏幕后处理的脚本。在这个脚本中，我们会实现OnRenderImage函数来获取当前屏幕的渲染纹理。然后，再调用Graphics.Blit函数使用特定的Unity Shader来对当前图像进行处理，再把返回的渲染纹理显示到屏幕上。对于一些复杂的屏幕特效，我们可能需要多次调用Graphics.Blit函数来对上一步的输出结果进行下一步处理。

但是，在进行屏幕后处理之前，我们需要检查一系列条件是否满足，例如当前平台是否支持渲染纹理和屏幕特效，是否支持当前使用的Unity Shader。为此，我们创建了一个用于屏幕后处理效果的基类，在实现各种屏幕特效时，我们只需要继承自该基类，再实现派生类种不同的操作即可。

1）调整屏幕的亮度与饱和度

我们可以将一张纹理的Texture Type设置为Sprite(2D and UI)格式，就产生了一个Sprite，可以直接拖入到场景中

饱和度是指彩色的鲜艳程度。饱和度取决于该色中含色成分和消色成分（灰色）的比例。含色成分越大，饱和度越大，否则越小。

首先添加一个基类脚本:

**[ExecuteInEditMode]**

**[RequireComponent (typeof(Camera))]**

public class PostEffectsBase : MonoBehaviour {

protected bool CheckSupport()

{

if (!SystemInfo.supportsImageEffects || !SystemInfo.supportsRenderTextures)

{

Debug.LogWarning("This platform does not support image effects or render textures.");

return false;

}

return true;

}

protected void CheckResources()

{

bool supported = CheckSupport();

if (!supported) this.enabled = false;

}

void Start () { **// 这里在Start的时候会检查平台是否支持Image Effect与RenderTexture，如果不支持则不执行该脚本，也就不会执行后面的OnRenderImage**

Debug.Log("post effects base start.");

CheckResources();

}

protected Material CheckShaderAndCreateMaterial(Shader shader, Material material)

{

if (shader == null) return null;

if (shader.isSupported && material && material.shader == shader) return material;

if (shader.isSupported)

{

material = new Material(shader);

material.hideFlags = HideFlags.DontSave;

if (material) return material;

}

return null;

}

}

然后添加挂在Camera上的脚本（继承于上面的脚本）

public class BrightnessSaturationAndContrast : PostEffectsBase {

public Shader briSatConShader;

private Material briSatConMaterial;

public Material material

{

get

{

briSatConMaterial = CheckShaderAndCreateMaterial(briSatConShader, briSatConMaterial);

return briSatConMaterial;

}

}

[Range(0.0f, 3.0f)]

public float brightness = 1.0f;

[Range(0.0f, 3.0f)]

public float saturation = 1.0f;

[Range(0.0f, 3.0f)]

public float contrast = 1.0f;

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material) // 这里会检查材质是否可用

{

material.SetFloat("\_Brightness", brightness);

material.SetFloat("\_Saturation", saturation);

material.SetFloat("\_Contrast", contrast);

Graphics.Blit(src, dest, material);

}

else // 没有材质则直接将原图显示到屏幕上行

{

Graphics.Blit(src, dest);

}

}

}

**Graphics.Blit会将第一个参数传递给Shader中名为\_MainTex的属性**

真正的修改实现在下面的shader：

Shader "Unlit/BrightnessSaturationAndContrast"

{

Properties

{

\_MainTex ("Base (RGB)", 2D) = "white" {}

\_Brightness ("Brightness", Float) = 1.0

\_Saturation ("Saturation", Float) = 1.0

\_Contrast ("Contrast", Float) = 1.0

}

SubShader

{

ZTest Always

Cull Off

ZWrite Off

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

struct a2v

{

float4 vertex : POSITION;

float2 texcoord : TEXCOORD0;

};

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

};

sampler2D \_MainTex;

float4 \_MainTex\_ST;

half \_Brightness;

half \_Saturation;

half \_Contrast;

v2f vert(a2v v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

fixed4 renderTex = tex2D(\_MainTex, o.uv);

// 亮度

fixed3 finalColor = renderTex.rgb \* \_Brightness;

// 饱和度（为0就表示灰度图）

fixed luminance = 0.2125 \* renderTex.r + 0.7154 \* renderTex.g + 0.0721 \* renderTex.b;

fixed3 luminanceColor = fixed3(luminance, luminance, luminance);

finalColor = lerp(luminanceColor, finalColor, \_Saturation);

// 对比度

fixed3 avgColor = fixed3(0.5, 0.5, 0.5);

finalColor = lerp(avgColor, finalColor, \_Contrast);

return fixed4(finalColor, renderTex.a);

}

ENDCG

}

}

}



OnRenderImage在OnGUI之后执行，并且该GameObject上必须有Camera组件

ZTest Always

Cull Off

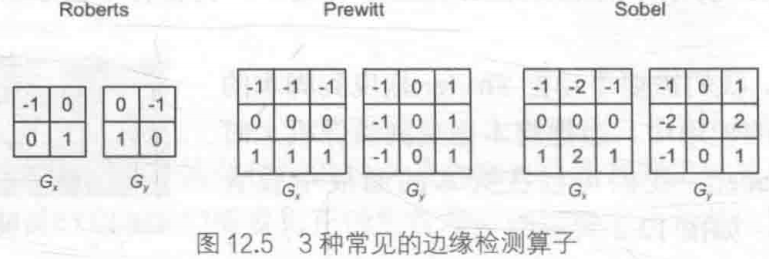
ZWrite Off

屏幕后处理实际上是在场景中绘制了一个与屏幕同宽同高的四边形面片，为了防止它对其他物体产生影响，我们需要设置相关的渲染状态。在这里，我们关闭了深度写入，是为了防止它“挡住”在其后面被渲染的物体。例如，如果当前的OnRenderImage函数在所有不透明的Pass执行完毕后立即被调用，不关闭深度写入就会影响后面透明的Pass的渲染。这些状态设置可以认为是用于屏幕后处理的Shader的“标配”

2）边缘检测

卷积

在图像处理中，卷积操作指的就是使用一个卷积核（kernel）对一张图像中的每个像素进行一个系列操作。卷积核通常是一个四方形网格结构（例如2x2、3x3的方形区域），该区域内每个方格都有一个权重值。当对图像中某个像素进行卷积时，我们会把卷积核的中心放置于该像素上，翻转核之后再一次计算核中每个元素和其覆盖的图像像素值的乘积并求和，得到的结果就是该位置的新像素值。卷积操作的神奇之处就在于选择的卷积核。



上面的三种常见的边缘检测算子都包含了两个方向的卷积核，分别用于检测水平方向和数值方向上的边缘信息。在进行边缘检测时，我们需要对每个像素分别进行一次卷积计算，得到两个方向上的梯度值Gx和Gy，而整体的梯度可以按下面的公式计算而得：



由于上面的计算包含了开根号操作，处于性能考虑，我们有时会使用绝对值操作来代替开根号操作：



当得到梯度G后，我们就可以根据此来判断哪些像素对应了边缘（梯度值越大，越有可能是边缘点）

Shader "Unlit/EdgeDetect"

{

Properties

{

\_MainTex ("Base (RGB)", 2D) = "white" {}

\_EdgeOnly ("Edge Only", Float) = 1.0

\_EdgeColor ("Edge Color", Color) = (0.0, 0.0, 0.0, 1.0)

\_BackgroundColor ("Background Color", Color) = (1.0, 1.0, 1.0, 1.0)

}

SubShader

{

Pass

{

ZTest Always

Cull Off

ZWrite Off

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

sampler2D \_MainTex;

float4 \_MainTex\_ST;

half4 \_MainTex\_TexelSize; // 纹理对应的每个纹素大小

float \_EdgeOnly;

fixed4 \_EdgeColor;

fixed4 \_BackgroundColor;

struct v2f

{

float4 pos : SV\_POSITION;

half2 uv[9] : TEXCOORD0;

};

v2f vert(appdata\_img v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

half2 uv = v.texcoord;

o.uv[0] = uv + \_MainTex\_TexelSize.xy \* half2(-1, -1);

o.uv[1] = uv + \_MainTex\_TexelSize.xy \* half2(0, -1);

o.uv[2] = uv + \_MainTex\_TexelSize.xy \* half2(1, -1);

o.uv[3] = uv + \_MainTex\_TexelSize.xy \* half2(-1, 0);

o.uv[4] = uv + \_MainTex\_TexelSize.xy \* half2(0, 0);

o.uv[5] = uv + \_MainTex\_TexelSize.xy \* half2(1, 0);

o.uv[6] = uv + \_MainTex\_TexelSize.xy \* half2(-1, 1);

o.uv[7] = uv + \_MainTex\_TexelSize.xy \* half2(0, 1);

o.uv[8] = uv + \_MainTex\_TexelSize.xy \* half2(1, 1);

return o;

}

fixed luminance(fixed4 color)

{

return 0.2125 \* color.r + 0.7154 \* color.g + 0.0721 \* color.b;

}

half Sobel(v2f o)

{

const half Gx[9] = { -1, -2, -1, 0, 0, 0, 1, 2, 1};

const half Gy[9] = { -1, 0, 1, -2, 0, 2, -1, 0, 1};

half texColor;

half edgeX = 0;

half edgeY = 0;

for (int i = 0; i < 9; i++)

{

texColor = luminance(tex2D(\_MainTex, o.uv[i])); // 先转换为灰度图，取9个纹素能消除噪声污染

edgeX += texColor \* Gx[i];

edgeY += texColor \* Gy[i];

}

half edge = 1 - abs(edgeX) - abs(edgeY); // 如果edgeX+edgeY比较大，则可能为0，就表示为边缘

return edge;

}

fixed4 frag(v2f o) : SV\_Target

{

half edge = Sobel(o);

fixed4 withEdgeColor = lerp(\_EdgeColor, tex2D(\_MainTex, o.uv[4]), edge);

fixed4 onlyEdgeColor = lerp(\_EdgeColor, \_BackgroundColor, edge);

return lerp(withEdgeColor, onlyEdgeColor, \_EdgeOnly);

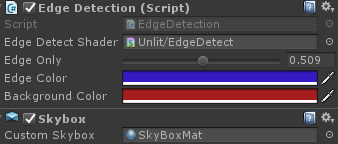
}

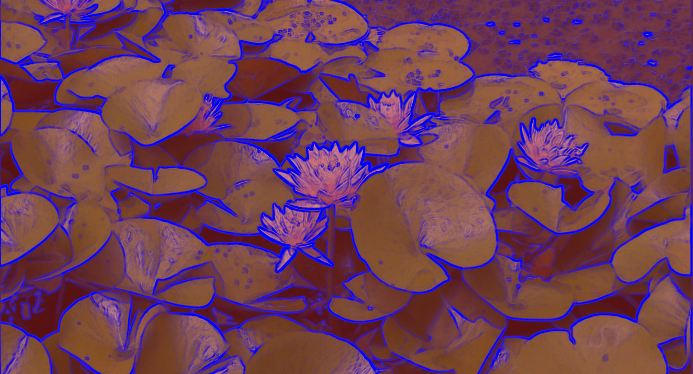
ENDCG

}

}

}





3）均值模糊

所谓模糊，也就是不清楚，各个像素之间会有明显的过渡，而如果像素之间的差距不是很大，那么图像就会模糊了，极端一点的情况，当一张图片所有的像素之间颜色都差不多时，那么这张图片就是一张纯色的图片了。模糊操作就是让像素间的颜色差距变小。

影响模糊程度的重要因素是模糊半径，模糊半径越大，模糊程度越大

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material)

{

material.SetFloat("\_BlurRadius", blurRadius);

Graphics.Blit(src, dest, material, -1);

}

else

{

Graphics.Blit(src, dest);

}

}

Shader "Unlit/SimpleBlur"

{

Properties

{

\_MainTex ("Texture Tint", 2D) = "white" {}

\_BlurRadius ("Blur Raidus", Float) = 1.0

}

CGINCLUDE

#include "UnityCG.cginc"

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv[9] : TEXCOORD0;

};

sampler2D \_MainTex;

half4 \_MainTex\_TexelSize;

float \_BlurRadius;

v2f vert(appdata\_img v)

{

v2f o;

half2 texcoordOffset = half2(\_MainTex\_TexelSize.x, \_MainTex\_TexelSize.y);

o.pos = UnityObjectToClipPos(v.vertex);

o.uv[0] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(1, 1);

o.uv[1] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(-1, 1);

o.uv[2] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(-1, -1);

o.uv[3] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(1, -1);

o.uv[4] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(0, 1);

o.uv[5] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(0, -1);

o.uv[6] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(-1, 0);

o.uv[7] = v.texcoord + \_BlurRadius \* texcoordOffset \* half2(1, 0);

o.uv[8] = v.texcoord;

return o;

}

fixed4 frag(v2f o) : SV\_Target

{

fixed3 finalColor = fixed3(0.0, 0.0, 0.0);

for (int i = 0; i < 9; i++)

{

fixed3 color0 = tex2D(\_MainTex, o.uv[i]);

finalColor += color0 \* 0.111;

}

return fixed4(finalColor, 1.0);

}

ENDCG

SubShader

{

ZTest Always

ZWrite Off

Cull Off

Pass

{

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

ENDCG

}

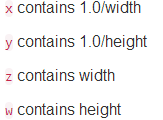
}

}

注意上面的CGINCLUDE与ENDCG标签，它实际上定义了一段CG代码段，下面可以引用上面的代码段

half4 \_MainTex\_TexelSize;

这个存储了相邻坐标的纹理偏移量

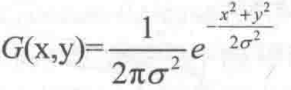




**均值模糊会出现方块化的效果，并不是很好**

4）高斯模糊

模糊的实现有很多方法，例如均值模糊和中值模糊。均值模糊同样适用了卷积操作，它使用的卷积核中的各个元素值都相等，且相加等于1，也就是说，卷积后得到的像素值是其领域内各个像素值的平均值。而中值模糊则是选择领域内对所有像素排序后的中值替换掉原颜色，一个更高级的模糊方法是高斯模糊。高斯模糊同样利用了卷积计算，它使用的卷积核名为高斯核。高斯核是一个正方形大小的滤波核，其中每个元素的计算都是基于下面的高斯方程：



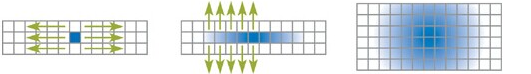
其中是标准方差（一般取值为1），x和y分别对应了当前位置到卷积核重心的整数距离。为了构建一个高斯核，我们只需要计算高斯核中各个位置对应的高斯值。为了保证滤波后的图像不会变暗，我们需要对高斯核中的权重归一化，即让每个权重除以所有权重的和，这样可以保证所有权重的和为1。高斯方程很好地模拟了领域每个像素对当前处理像素的影响程度—-距离越近，影响越大。高斯的维数越高，模糊程度越大。



高斯方程很好地模拟了邻域每个像素对当前处理像素的影响程度—距离越近，影响越大。高斯核的维数越高，模糊程度越大。使用一个N\*N的高斯核对图像进行卷积滤波，就需要N\*N\*W\*H（W和H分别是图像的宽和高）次纹理采样。当N的大小不断增大时，采样次数就会变得非常巨大。幸运的是，我们可以把这个二维高斯函数拆分成两个一维函数。也就是说，我们可以使用两个一维的高斯核先后对图像进行滤波，他们得到的结果和直接使用二维高斯和是一样的，但采样次数只需要2\*N\*W\*H。

再游戏中的模糊通常都是近似高斯模糊，只要保证权重采样是一条类似高斯模糊的钟形曲线就行，即从中心到边缘是平滑渐变的，一般来说游戏中对高斯模糊的优化有几点：

1. 降低阶数：降低采样的阶数，比如5阶的滤波器就比7阶的滤波器效率高，实际上一般不超过7阶
2. 迭代计算：采用低阶采样的同时，可以进行迭代计算，就是把上一次的模糊结果，再进行同样的采样模糊，以达到更好的效果
3. 固定权重：权重预先计算好，并且归一化固定下来，不在游戏过程中实时计算
4. 降维计算：因为高斯模糊在二维图像上是线性可分的，可以二维计算拆分成两次一维计算。具体就是先在水平方向做一维高斯矩阵变换后，将其结果再进行垂直方向的一维高斯矩阵变换。如下图来看对于一个9阶的二维运算来说，需要9\*9=81次采样，但是如果拆分成以为运算的话，只需要9+9=18次采样，只不过需要缓存第一次计算的结果，以空间换时间还是划算的，也可以得出阶数越高，效率相差也越大的结论。



Shader代码：

Shader "Unlit/GaussianBlur"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

\_BlurSize ("Blur Size", Float) = 1.0

}

SubShader

{

ZTest Always

Cull Off

ZWrite Off

CGINCLUDE

#include "UnityCG.cginc"

sampler2D \_MainTex;

float4 \_MainTex\_ST;

float4 \_MainTex\_TexelSize;

float \_BlurSize;

struct v2f

{

float4 pos : SV\_POSITION;

half2 uv[5] : TEXCOORD0;

};

v2f vertBlurVertical(appdata\_img v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

half2 uv = v.texcoord;

o.uv[0] = uv;

o.uv[1] = uv + float2(0.0, \_MainTex\_TexelSize.y \* 1.0) \* \_BlurSize;

o.uv[2] = uv - float2(0.0, \_MainTex\_TexelSize.y \* 1.0) \* \_BlurSize;

o.uv[3] = uv + float2(0.0, \_MainTex\_TexelSize.y \* 2.0) \* \_BlurSize;

o.uv[4] = uv - float2(0.0, \_MainTex\_TexelSize.y \* 2.0) \* \_BlurSize;

return o;

}

v2f vertBlurHorizontal(appdata\_img v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

half2 uv = v.texcoord;

o.uv[0] = uv;

o.uv[1] = uv + float2(\_MainTex\_TexelSize.x \* 1.0, 0.0) \* \_BlurSize;

o.uv[2] = uv - float2(\_MainTex\_TexelSize.x \* 1.0, 0.0) \* \_BlurSize;

o.uv[3] = uv + float2(\_MainTex\_TexelSize.x \* 2.0, 0.0) \* \_BlurSize;

o.uv[4] = uv - float2(\_MainTex\_TexelSize.x \* 2.0, 0.0) \* \_BlurSize;

return o;

}

fixed4 fragBlur(v2f o) : SV\_Target

{

float weight[3] = { 0.4026, 0.2442, 0.0545 };

fixed3 sum = tex2D(\_MainTex, o.uv[0]).rgb \* weight[0];

for (int i = 1; i < 3; i++)

{

sum += tex2D(\_MainTex, o.uv[i]).rgb \* weight[i];

sum += tex2D(\_MainTex, o.uv[2 \* i]).rgb \* weight[i];

}

return fixed4(sum, 1.0);

}

ENDCG

Pass

{

NAME "GAUSSIAN\_BLUR\_VERTICAL"

CGPROGRAM

#pragma vertex vertBlurVertical

#pragma fragment fragBlur

ENDCG

}

Pass

{

NAME "GAUSSIAN\_BLUR\_HORIZONTAL"

CGPROGRAM

#pragma vertex vertBlurHorizontal

#pragma fragment fragBlur

ENDCG

}

}

}

1）简单的c#代码

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material)

{

material.SetFloat("\_BlurSize", blurSpread);

int rtW = src.width;

int rtH = src.height;

RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH, 0);

// Render the vertical pass (采用第一个pass)

Graphics.Blit(src, buffer, material, 0);

// Render the horizontal pass (采用第二个pass)

Graphics.Blit(buffer, dest, material, 1);

RenderTexture.ReleaseTemporary(buffer);

}

else

{

Graphics.Blit(src, dest);

}

}





2）降采样

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material)

{

material.SetFloat("\_BlurSize", blurSpread);

int rtW = src.width / downSample;

int rtH = src.height / downSample;

RenderTexture buffer = RenderTexture.GetTemporary(rtW, rtH, 0);

buffer.filterMode = FilterMode.Bilinear;

// Render the vertical pass (采用第一个pass，渲染到了buffer中)

Graphics.Blit(src, buffer, material, 0);

// Render the horizontal pass (采用第二个pass)

Graphics.Blit(buffer, dest, material, 1);

RenderTexture.ReleaseTemporary(buffer);

}

else

{

Graphics.Blit(src, dest);

}

}



3）多次迭代

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material)

{

int rtW = src.width / downSample;

int rtH = src.height / downSample;

RenderTexture buffer0 = RenderTexture.GetTemporary(rtW, rtH, 0);

buffer0.filterMode = FilterMode.Bilinear;

Graphics.Blit(src, buffer0); // 直接将采样，在迭代的过程中处理的是降采样的纹理

for (int i = 0; i < iterations; i++)

{

material.SetFloat("\_BlurSize", 1.0f + i \* blurSpread);

// 处理垂直模糊

RenderTexture buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

Graphics.Blit(buffer0, buffer1, material, 0);

RenderTexture.ReleaseTemporary(buffer0);

buffer0 = buffer1;

// 处理水平模糊

buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

Graphics.Blit(buffer0, buffer1, material, 1);

RenderTexture.ReleaseTemporary(buffer0);

buffer0 = buffer1;

}

Graphics.Blit(buffer0, dest);

RenderTexture.ReleaseTemporary(buffer0);

}

else

{

Graphics.Blit(src, dest);

}

}



在高斯核维数不变的情况下，\_BlurSize越大，模糊程度越高，但采样数却不会收到影响。但过大的\_BlurSize值会造成虚影，这可能并不是我们希望的，所以应该将这个值设置为一个合理值。

5）bloom效果

Bloom特效时游戏中常见的一种屏幕特效。这种特效可以模糊真实摄像机的一种图像效果，它让画面中较亮的区域“扩散”到周围的区域中，造成一种朦胧效果。

Bloom的实现原理非常简单：我们首先根据一个阈值提取出图像中较亮区域，把他们存储在一张渲染纹理中，再利用高斯模糊对这张渲染纹理进行模糊处理，模拟光线扩散的效果，最后再将其和原图像进行混合，得到最终的效果。

Shader "Unlit/Bloom"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

\_Bloom ("Bloom", 2D) = "black" {}

\_LuminanceThreshold ("Luminance Threshold", Float) = 0.5

\_BlurSize ("Blur Size", Float) = 1.0

}

SubShader

{

ZTest Always

Cull Off

ZWrite Off

CGINCLUDE

#include "UnityCG.cginc"

sampler2D \_MainTex;

float4 \_MainTex\_TexelSize;

sampler2D \_Bloom;

float \_LuminanceThreshold;

float \_BlurSize;

struct v2f

{

float4 pos : SV\_POSITION;

half2 uv : TEXCOORD0;

};

v2f vertExtractBright(appdata\_img v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = v.texcoord;

return o;

}

fixed luminance(fixed4 color)

{

return 0.2125 \* color.r + 0.7154 \* color.g + 0.0721 \* color.b;

}

fixed4 fragExtractBright(v2f o) : SV\_Target

{

fixed4 c = tex2D(\_MainTex, o.uv);

fixed4 val = clamp(luminance(c) - \_LuminanceThreshold, 0.0, 1.0);

return c \* val;

}

struct v2fBloom

{

float4 pos : SV\_POSITION;

half4 uv : TEXCOORD0;

};

v2fBloom vertBloom(appdata\_img v)

{

v2fBloom o;

UNITY\_INITIALIZE\_OUTPUT(v2fBloom, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.uv.xy = v.texcoord;

o.uv.zw = v.texcoord;

#if UNITY\_UV\_STARTS\_AT\_TOP

if (\_MainTex\_TexelSize.y < 0.0)

o.uv.w = 1.0 - o.uv.w;

#endif

return o;

}

fixed4 fragBloom(v2fBloom o) : SV\_Target

{

return tex2D(\_MainTex, o.uv.xy) + tex2D(\_Bloom, o.uv.zw);

}

ENDCG

Pass

{

CGPROGRAM

#pragma vertex vertExtractBright

#pragma fragment fragExtractBright

ENDCG

}

UsePass "Unlit/GaussianBlur/GAUSSIAN\_BLUR\_VERTICAL"

UsePass "Unlit/GaussianBlur/GAUSSIAN\_BLUR\_HORIZONTAL"

Pass

{

CGPROGRAM

#pragma vertex vertBloom

#pragma fragment fragBloom

ENDCG

}

}

}

C#代码：

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material)

{

material.SetFloat("\_LuminanceThreshold", luminanThreshold);

int rtW = src.width / downSample;

int rtH = src.height / downSample;

RenderTexture buffer0 = RenderTexture.GetTemporary(src.width, src.height, 0);

buffer0.filterMode = FilterMode.Bilinear;

Graphics.Blit(src, buffer0, material, 0); // 第一个pass，提取较亮区域

for (int i = 0; i < iterations; i++)

{

material.SetFloat("\_BlurSize", 1.0f + blurSpread);

RenderTexture buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

Graphics.Blit(buffer0, buffer1, material, 1); // 第二个pass

RenderTexture.ReleaseTemporary(buffer0);

buffer0 = buffer1;

buffer1 = RenderTexture.GetTemporary(rtW, rtH, 0);

Graphics.Blit(buffer0, buffer1, material, 2); // 第三个pass

RenderTexture.ReleaseTemporary(buffer0);

buffer0 = buffer1;

}

material.SetTexture("\_Bloom", buffer0);

Graphics.Blit(src, dest, material, 3); // 第四个pass

RenderTexture.ReleaseTemporary(buffer0);

}

else

{

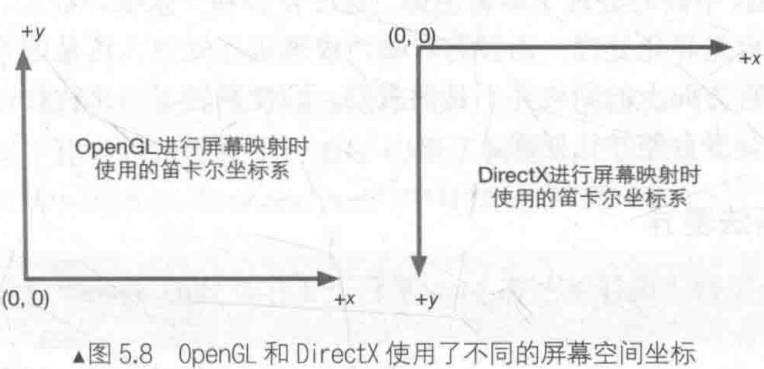
Graphics.Blit(src, dest);

}

}



**渲染纹理的坐标差异**



需要注意的是，我们不仅可以把渲染结果输出到屏幕上，还可以输出到不同的渲染目标（Render Texture）中。这时，我们需要使用渲染纹理（Render Texture）来保存这些渲染结果。大多数情况下，这样的差异并不会对我们造成任何影响。但当我们要使用渲染到纹理技术，把屏幕图像渲染到一张渲染纹理中时，如果不采取任何措施的话，就会出现纹理翻转的情况。

UNITY\_UV\_STARTS\_AT\_TOP用于判断当前平台是否是DirectX类型的平台，而当在这样的平台下开启了抗锯齿后，主纹理的纹素大小在竖直方向上会变成负数，以方便我们队主纹理进行正确的采样。因此我们可以通过判断\_MainTex\_TexelSize.y是否小于0来检验是否开启了抗锯齿，如果是，我们就需要对除主纹理外的其他纹理的采样坐标进行竖直方向上的翻转。

UNITY\_UV\_STARTS\_AT\_TOP 总是用1或0定义；   
当纹理的V坐标系原点在纹理顶部的平台上值是1。   
Direct3D类似平台使用1；OpenGL类似平台使用0。

在Direct3D平台下，如果我们开启了抗锯齿，则xxx\_TexelSize.y 会变成负值，好让我们能够正确的进行采样。   
所以if (\_MainTex\_TexelSize.y < 0)的作用就是判断我们当前是否开启了抗锯齿。

**UNITY\_UV\_STARTS\_AT\_TOP用于判断是否是Direct3D平台**

**if (\_MainTex\_TexelSize.y < 0)判断是否开启了抗锯齿**

如果我们做的屏幕后期特效简单（一次处理一个纹理），这无关紧要，因为 Graphics.Blit 方法会自动进行处理。

然而，如果在屏幕后期特效中同时处理一个以上的 RenderTexture，它们很可能会在不同的垂直方向出现（仅在类似 Direct3D 的平台上，并且仅在使用抗锯齿选项时）。您需要在顶点着色器中手动“翻转”屏幕纹理。

6）运动模糊

运动模糊的实现有多重方法。一种实现方法是利用一块累积缓存区来混合多张连续的图像。当物体快速移动产生多张图像后，我们取它们之间的平均值作为最后的运动模糊图像，然而，这种暴力的方法对性能的消耗很大，因此想要获取多张帧图像往往意味着我们需要在同一帧里渲染多次场景。另一种应用广泛的方法是创建和使用速度缓存，这个缓存中存储了各个像素当前的运动速度，然后利用该值来决定模糊的方向和大小

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

if (material)

{

if (accumulationTexture == null || accumulationTexture.width != src.width || accumulationTexture.height != src.height)

{

DestroyImmediate(accumulationTexture);

accumulationTexture = new RenderTexture(src.width, src.height, 0);

accumulationTexture.hideFlags = HideFlags.HideAndDontSave;

Graphics.Blit(src, accumulationTexture);

}

accumulationTexture.MarkRestoreExpected();

material.SetFloat("\_BlurAmount", 1 - blurAmount);

Graphics.Blit(src, accumulationTexture, material);

Graphics.Blit(accumulationTexture, dest);

}

else

{

Graphics.Blit(src, dest);

}

}

Shader "Unlit/MotionBlur"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

\_BlurAmount ("Blur Amount", Float) = 1.0

}

CGINCLUDE

#include "UnityCG.cginc"

sampler2D \_MainTex;

float4 \_MainTex\_ST;

fixed \_BlurAmount;

struct v2f

{

float4 pos : SV\_POSITION;

half2 uv : TEXCOORD0;

};

v2f vert(appdata\_img v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);

return o;

}

fixed4 fragRGB(v2f o) : SV\_Target

{

return fixed4(tex2D(\_MainTex, o.uv).rgb, \_BlurAmount);

}

fixed4 fragA(v2f o) : SV\_Target

{

return tex2D(\_MainTex, o.uv);

}

ENDCG

SubShader

{

ZTest Always

ZWrite Off

Cull Off

Pass

{

Blend SrcAlpha OneMinusSrcAlpha

ColorMask RGB

CGPROGRAM

#pragma vertex vert

#pragma fragment fragRGB

ENDCG

}

Pass

{

Blend One Zero

ColorMask A

CGPROGRAM

#pragma vertex vert

#pragma fragment fragA

ENDCG

}

}

}

RGB通道版本的Shader对当前图像进行采样，并将其A通道的值设为\_BlurAmount，以便在后面混合时候可以使用它的透明通道进行混合。A通道版本直接返回采样结果。实际上，这个版本只是为了维护渲染纹理的透明通道值，不让其受到混合时使用的透明度值的影响。之所以要把A通道和RGB通道分开，是因为在更新RGB时我们需要设置它的A通道来混合图像，但又不希望A通道的值写入渲染纹理中。

**获取深度和法线纹理**

1）背后的原理

深度纹理实际就是一张渲染纹理，只不过它里面存储的像素值不是颜色值，而是一个高精度的深度值。由于被存储在一张纹理中，深度纹理里的深度值范围是[0,1]，而且通常是非线性分部的。总体来说，这些深度值来自于顶点变换后得到的归一化的设备坐标（NDC坐标）。由于NDC中z分量的范围再[-1,1]中，为了让这些值能都存储在一张图中，我们需要使用下面的公式对其进行映射



那么，Unity是怎么得到这样一张深度纹理呢？在Unity中，深度纹理可以直接来自于真正的深度缓存，也可以是由一个单独的Pass而得，这取决于使用的渲染路径和硬件。通常来讲，当使用延迟渲染路径（包括遗留的延迟渲染路径）时，深度纹理理所当然可以访问到，因为延迟渲染会把这些信息渲染到G-buffer中。而当无法直接获取深度缓存时，深度和法线纹理是通过一个单独的Pass渲染而得的。具体实现是，Unity会使用着色器替换（Shader Replacement）技术选择那些渲染类型（即SubShader的RenderType标签）为Opaque的物体，判断他们使用的渲染队列是否小于等于2500（内置的Background、Geometry和Alpha Test渲染队列均在此范围内），如果满足条件，就把它渲染到深度和法线纹理中。因此，想要让物体能够出现在深度和法线纹理中，就必须在Shader中设置正确的RenderType标签。

在Unity中，我们可以选择让一个摄像机生成一张深度纹理或是一张深度+法线纹理。

1. 只需要一张单独的深度纹理时，Unity会直接获取深度缓存或是按之前提到的着色器替换技术，选择需要的不透明物体，并使用它投射阴影时使用的Pass（即LightMode被设置为ShadowCaster）来得到深度纹理。如果Shader中不包含这样一个Pass，那么这个物体就不会出现在深度纹理中（当然，它也不能像其他物体投射阴影）。深度纹理的精度通常是24位或16位，这取决于使用的深度缓存的精度
2. 需要深度+法线纹理，Unity会创建一张和屏幕分辨率相同、精度为32位的纹理，其中观察空间下的法线信息会被编码到R和G通道，而深度信息会被编码到B和A通道。法线信息的获取在延迟渲染中可以非常容易就得到的，Unity只需要合并深度和法线缓存即可。而在前向渲染中，默认情况下是不会创建法线缓存的，因此Unity底层使用了一个单独的Pass把整个场景在此渲染一遍来完成，这个Pass被包含在Unity内置的一个Shader中，我们可以在内置的builtin\_shaderx-xxx/DefaultResources/Camera-DepthNormalTexture.shader文件中找到这个用于渲染深度和法线信息的Pass

如何获取？

在unity中，获取深度纹理是非常简单的，我们只需要告诉unity：“嘿，把深度纹理给我”，然后再在Shader中直接访问特定的纹理属性即可。这个与Unity沟通的过程是通过在脚本中设置摄像机的depathTextureMode来完成的，例如我们可以通过下面的代码来获取深度纹理：

Camera.depthTextureMode = DepthTextureMode.Depth

一旦设置好了上面的摄像机模式后，我们就可以在Shader中通过声明\_CameraDepthTexture变量来访问它。这个过程非常简单，但我们需要知道这两行代码的背后，Unity为我们做了很多事情。

同理，如果我们想要获取深度+法线纹理，我们只需要在代码总这样设置：

Camera.depthTextureMode = DepthTextureMode.DepthNormals

然后在Shader中通过声明\_CameraDepthNormalsTexture变量来访问它

我们还可以组合这些模式，让一个摄像机同时产生一张深度和深度+法线纹理

Camera.depthTextureMode |= DepthTextureMode.Depth

Camera.depthTextureMode |= DepthTextureMode.DepthNormals

通常我们需要知道眼坐标线性的深度值，这个其实很容易推导，不过unity提供了两个辅助函数来为我们进行这个计算过程---LinearEyeDepth和Linear01Depth. LinearEyeDepth负责把深度纹理的采样结果转换到视角空间下的深度值，Linear01Depth则会返回一个范围在[0,1]的线性深度值。这两个函数内部使用了内置变量\_ZBufferParams变量来得到远近裁剪平面的距离。float4 \_ZBufferParams //x=1-Far/Near,y=Far/Near,z=x/Far,w=y/Far,该变量用于线性化Z缓存中的深度值

如果我们需要获取深度+法线纹理，可以直接使用tex2D函数对\_CameraDepthNormalsTexture进行采样，得到里面存储的深度和法线信息。Unity提供了辅助函数来为我们对这个采样结果进行编码，从而的得到深度值和法线方向。这个函数是DecodeDepthNormal，它在UnityCG.cginc里被定义：

inline void DecodeDepthNormal( float4 enc, out float depth, out float3 normal )

{

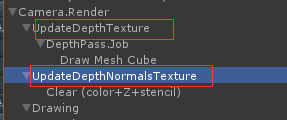
depth = DecodeFloatRG (enc.zw);

normal = DecodeViewNormalStereo (enc);

}

该函数的第一个参数是对深度+法线纹理的采样结果，它的xy分量存储的是视角空间下的法线信息，而深度信息被编码进了zw分量，通过调用该函数我们就可以得到解码都得深度值和法线。这个深度值是范围在[0,1]的线性深度值（这与单独的深度纹理中存储的深度值不同，单独的深度纹理存储的是非线性的深度值），而得到的法线则是视角空间下的法线方向。

我们可以通过Frame Debugger查看深度纹理和深度+法线纹理



注意对于深度+法线纹理，需要物体的Tags有“RenderType”=”Opaque”，这样它会进行替换渲染

**再谈运动模糊**

之前学习了如何通过混合多张屏幕图像来模拟运动模糊效果，但是，另医用应用更加广泛的技术则是使用速度映射图。速度映射图中存储了每个像素的速度，然后使用这个速度来决定模糊的方向和大小。速度缓冲的生成有多种方法，一种方法是把场景中所有物体的速度渲染到一张纹理中。但这种方法的缺点在于需要修改场景中所有物体的Shader代码，使其添加计算速度的代码并输出到一个渲染纹理中。

一种生成速度映射图的方法：

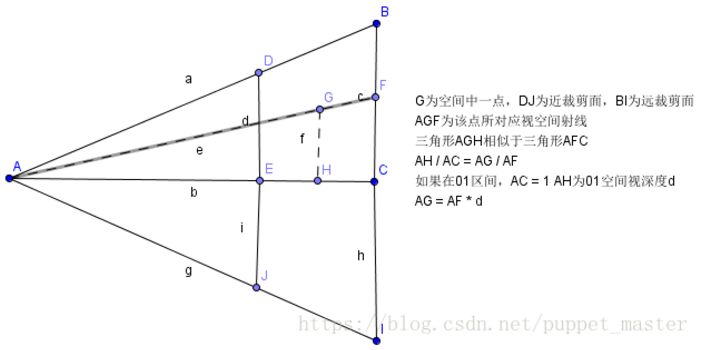
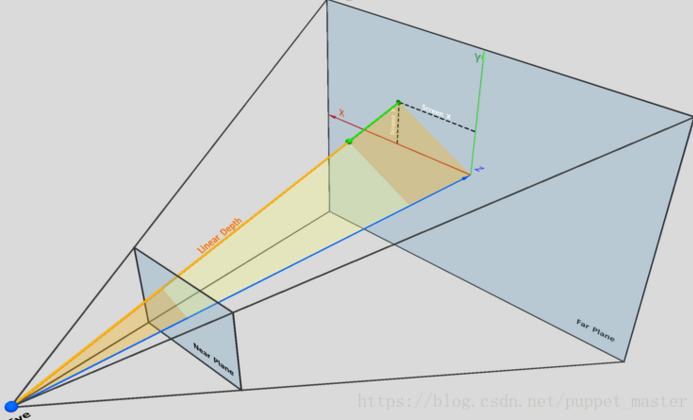
利用深度纹理，在片元着色器中为每个像素计算器在世界空间下的位置，这是通过使用当前的视角\*投影矩阵的你矩阵对ndc下的顶点坐标进行变换得到的。当得到世界空间中的顶点坐标后，我们使用前一帧的视角\*投影矩阵对其进行变换，得到该位置在前一帧中的NDC坐标。然后，计算前一帧和当前帧的位置差，形成该像素的速度，这种方法的优点是可以在一个屏幕后处理步骤中完成整个效果的模拟，但缺点是需要在片元着色器中进行两次矩阵乘法的操作，对性能有所影响。

**全局雾效**

Unity内置的雾效可以产生基于距离的线性或指数雾效。要想在我们的说则其中实现这些雾效，需要在shader中添加#pragma multi\_compile\_fog指令，同时还需要使用相关的内置宏，例如UNITY\_FOG\_COORDS、UNITY\_TRANSFER\_FOG和UNITY\_APPLY\_FOG等。这种方法的缺点是如果需要对雾效进行一些个性化操作时，例如使用基于高度的雾效等，仅仅使用Unity内置的雾效就变得不在可行。用一次屏幕后处理，我们可以很方便地模拟各种雾效，例如均匀的雾效、基于距离的线性/指数雾效、基于高度的雾效等。

上面的方法虽然可以得到世界坐标，但是需要逐像素进行矩阵计算，性能比较差。

这种射线方式的重建，是Cryket公司2011年创建的



也就是通过插值得到每个片元的射线，这个插值是通过四边形的四个顶点射线得到的

由上图可知，最终世界坐标位置

worldPos = OA + AG(世界坐标系上的向量)

其中OA就是摄像机在世界坐标系的位置，AG是世界坐标系上的向量

看看在C#如何实现：

void OnRenderImage(RenderTexture src, RenderTexture dest)

{

Matrix4x4 frustumCorners = Matrix4x4.identity;

float fov = camera.fieldOfView;

float far = camera.farClipPlane;

float near = camera.nearClipPlane;

float aspect = camera.aspect;

float halfHeight = near \* Mathf.Tan(fov \* 0.5f \* Mathf.Deg2Rad);

// toRight与toTop是世界空间坐标系下的向量（该向量的起点为近平面的中心点）

Vector3 toRight = cameraTransform.right \* halfHeight \* aspect;

Vector3 toTop = cameraTransform.up \* halfHeight;

// topLeft为向量（起点为世界坐标摄像机位置，终点为世界坐标四边形左上角）

Vector3 topLeft = cameraTransform.forward \* near + toTop - toRight;

/\* 现在已知摄像机到topLeft的向量，我们现在需要得到一个深度为depth像素的世界坐标

\* 也就是只需要用topLeft的单位向量\*摄像机到该像素的长度即可

\* 像素长度 / |topLeft| = depth / near

\* 得到像素长度 = depth \* |topLeft| / near

\*/

float scale = topLeft.magnitude / near;

topLeft.Normalize();

topLeft \*= scale;

Vector3 topRight = cameraTransform.forward \* near + toTop + toRight;

topRight.Normalize();

topRight \*= scale;

Vector3 bottomLeft = cameraTransform.forward \* near - toTop - toRight;

bottomLeft.Normalize();

bottomLeft \*= scale;

Vector3 bottomRight = cameraTransform.forward \* near - toTop + toRight;

bottomRight.Normalize();

bottomRight \*= scale;

frustumCorners.SetRow(0, bottomLeft);

frustumCorners.SetRow(1, bottomRight);

frustumCorners.SetRow(2, topRight);

frustumCorners.SetRow(3, topLeft);

material.SetMatrix("\_FrustumCornerRay", frustumCorners);

material.SetMatrix("\_ViewProjectionInverseMatrix", (camera.projectionMatrix \* camera.worldToCameraMatrix).inverse);

material.SetFloat("\_FogDensity", fogDesity);

material.SetColor("\_FogColor", fogColor);

material.SetFloat("\_FogStart", fogStart);

material.SetFloat("\_FogEnd", fogEnd);

if (material)

{

Graphics.Blit(src, dest, material);

}

else

{

Graphics.Blit(src, dest);

}

}

Shader "Unlit/FogWithDepthTexture"

{

Properties

{

\_MainTex ("Texture", 2D) = "white" {}

\_FogDensity ("Fog Density", Float) = 1.0

\_FogColor ("Fog Color", Color) = (1.0, 1.0, 1.0, 1.0)

\_FogStart ("Fog Start", Float) = 0.0

\_FogEnd ("Fog End", Float) = 1.0

}

CGINCLUDE

#include "UnityCG.cginc"

sampler2D \_CameraDepthTexture;

sampler2D \_MainTex;

float4 \_MainTex\_ST;

float4 \_MainTex\_TexelSize;

float \_FogDensity;

fixed4 \_FogColor;

float \_FogStart;

float \_FogEnd;

float4x4 \_FrustumCornerRay;

struct v2f

{

float4 pos : SV\_POSITION;

half2 uv : TEXCOORD0;

half2 uv\_depth : TEXCOORD1;

float4 interpolatedRay : TEXCOORD2;

};

v2f vert(appdata\_img v)

{

v2f o;

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = v.texcoord;

o.uv\_depth = v.texcoord;

int index = 0;

if (v.texcoord.x < 0.5 && v.texcoord.y < 0.5)

{

index = 0;

}

else if (v.texcoord.x >= 0.5 && v.texcoord.y < 0.5)

{

index = 1;

}

else if (v.texcoord.x >= 0.5 && v.texcoord.y >= 0.5)

{

index = 2;

}

else

{

index = 3;

}

#if UNITY\_UV\_STARTS\_AT\_TOP

if (\_MainTex\_TexelSize.y < 0) // 如果没有开启抗锯齿则unity会自动翻转，此时y>0

{

o.uv\_depth = 1 - o.uv\_depth.y;

index = 3 - index;

}

#endif

o.interpolatedRay = \_FrustumCornerRay[index];

return o;

}

fixed4 frag (v2f i) : SV\_Target

{

float linearDepth = LinearEyeDepth(SAMPLE\_DEPTH\_TEXTURE(\_CameraDepthTexture, i.uv\_depth));

// 通过深度计算世界坐标

float3 worldPos = \_WorldSpaceCameraPos + linearDepth \* i.interpolatedRay.xyz;

// 计算雾

float fogDensity = (\_FogEnd - worldPos.y) / (\_FogEnd - \_FogStart);

fogDensity = saturate(fogDensity \* \_FogDensity);

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

finalColor.rgb = lerp(finalColor.rgb, \_FogColor.rgb, fogDensity);

return finalColor;

}

ENDCG

SubShader

{

Tags { "RenderType"="Opaque" }

Pass

{

ZWrite Off

ZTest Off

Cull Off

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

ENDCG

}

}

}

**非真实感渲染（Non-Photorealistic Rendering，NPR）**

非真实感渲染的一个主要目标是，使用一些渲染方法使得画面得到和某些特殊的绘画风格相似的效果，例如卡通、水彩风格等

1）卡通风格的渲染

要实现卡通渲染有很多方法，其中之一就是使用**基于色调的着色技术**。在实现中，我们往往会使用漫反射系数对一张一维纹理进行采样，以控制漫反射的色调。卡通风格的高光效果也和我们之前的光照不同，在卡通风格中，模型的高光往往是一块块分界明显的纯色区域。除了光照模型不同外，卡通风格通常还需要在物体边缘部分绘制轮廓，在前面中我们介绍使用屏幕后处理技术对屏幕图像进行描边，在这里，我们将会介绍基于模型的描边方法，这种实现方法更加简单，而且在很多情况下也能得到不错的效果。

Shader "Unlit/ToonShader"

{

Properties

{

\_Color ("Color Tint", Color) = (1, 1, 1, 1)

\_MainTex ("Texture", 2D) = "white" {}

// 漫反射色调的渐变纹理

\_Ramp ("Ramp Texture", 2D) = "white" {}

// 轮廓线宽度

\_Outline ("Outline", Range(0, 1)) = 0.1

// 轮廓线颜色

\_OutlineColor ("Outline Color", Color) = (0, 0, 0, 1)

// 高光反射颜色

\_Specular ("Specular", Color) = (1, 1, 1, 1)

// 高光反射使用的阈值

\_SpecularScale ("Specular Scale", Range(0, 0.1)) = 0.01

}

SubShader

{

Name "OutLine"

Pass

{

Cull Front

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#include "UnityCG.cginc"

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

};

float \_Outline;

fixed4 \_OutlineColor;

v2f vert(appdata\_base v)

{

v2f o;

float4 pos = mul(UNITY\_MATRIX\_MV, v.vertex);

float3 normal = mul((float3x3)UNITY\_MATRIX\_IT\_MV, v.normal);

// 尽可能让背面扁平化，避免凹多边形背面扩张后的顶点挡住正面的面片

normal.z = -0.5;

pos = pos + float4(normalize(normal), 0) \* \_Outline;

o.pos = mul(UNITY\_MATRIX\_P, pos);

return o;

}

fixed4 frag(v2f i) : SV\_Target

{

return fixed4(\_OutlineColor.rgb, 1.0);

}

ENDCG

}

Pass

{

Tags { "LightMode" = "ForwardBase" }

Cull Back

CGPROGRAM

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_fwdbase

#include "UnityCG.cginc"

#include "AutoLight.cginc"

#include "Lighting.cginc"

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

float3 worldNormal : TEXCOORD1;

float3 worldPos : TEXCOORD2;

SHADOW\_COORDS(3)

};

sampler2D \_MainTex;

float4 \_MainTex\_ST;

sampler2D \_Ramp;

float \_Outline;

fixed4 \_OutlineColor;

fixed4 \_Color;

fixed4 \_Specular;

float \_SpecularScale;

v2f vert(appdata\_base v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = TRANSFORM\_TEX(v.texcoord, \_MainTex);

o.worldNormal = UnityObjectToWorldNormal(v.normal);

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

TRANSFER\_SHADOW(o);

return o;

}

fixed4 frag(v2f i) : SV\_Target

{

fixed3 worldNormal = normalize(i.worldNormal);

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

fixed3 worldViewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));

fixed3 worldHalfDir = normalize(worldLightDir + worldViewDir);

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

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

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

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

fixed diff = dot(worldNormal, worldLightDir);

diff = (diff \* 0.5 + 0.5) \* atten;

fixed3 diffuse = \_LightColor0.rgb \* albedo \* diff; // tex2D(\_Ramp, float2(diff, diff)).rgb;

fixed spec = dot(worldNormal, worldHalfDir);

fixed3 w = fwidth(spec);

fixed3 specular = \_Specular.rgb \* lerp(0, 1, smoothstep(-w, w, spec + \_SpecularScale - 1)) \* step(0.0001, \_SpecularScale);

return fixed4(ambient + diffuse + specular, 1.0);

}

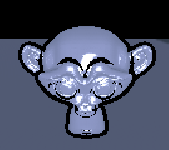
ENDCG

}

}

FallBack "Diffuse"

}



**素描风格渲染**

Shader "Unlit/HatchShader"

{

Properties

{

\_Color ("Color Tint", Color) = (1, 1, 1, 1)

\_TileFactor ("Tile Factor", Float) = 1

\_Outline ("Outline", Range(0, 1)) = 0.1

\_Hatch0 ("Hatch 0", 2D) = "white" {}

\_Hatch1 ("Hatch 1", 2D) = "white" {}

\_Hatch2 ("Hatch 2", 2D) = "white" {}

\_Hatch3 ("Hatch 3", 2D) = "white" {}

\_Hatch4 ("Hatch 4", 2D) = "white" {}

\_Hatch5 ("Hatch 5", 2D) = "white" {}

}

SubShader

{

Tags { "RenderType" = "Opaque" "Queue" = "Geometry" }

UsePass "Unlit/ToonShader/OUTLINE"

Pass

{

Tags { "LightMode" = "ForwardBase" }

CGPROGRAM

#include "UnityCG.cginc"

#include "AutoLight.cginc"

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_fwdbase

struct v2f

{

float4 pos : SV\_POSITION;

float2 uv : TEXCOORD0;

fixed3 hatchWeights0 : TEXCOORD1;

fixed3 hatchWeights1 : TEXCOORD2;

float3 worldPos : TEXCOORD3;

SHADOW\_COORDS(4)

};

fixed4 \_Color;

float \_TileFactor;

float \_Outline;

sampler2D \_Hatch0;

sampler2D \_Hatch1;

sampler2D \_Hatch2;

sampler2D \_Hatch3;

sampler2D \_Hatch4;

sampler2D \_Hatch5;

v2f vert(appdata\_base v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.uv = v.texcoord.xy \* \_TileFactor;

fixed3 worldLightDir = normalize(WorldSpaceLightDir(v.vertex));

fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);

fixed diff = max(0, dot(worldLightDir, worldNormal));

o.hatchWeights0 = fixed3(0, 0, 0);

o.hatchWeights1 = fixed3(0, 0, 0);

float hatchFactor = diff \* 7.0;

if (hatchFactor > 6.0)

{

}

else if (hatchFactor > 5.0)

{

o.hatchWeights0.x = hatchFactor - 5.0;

}

else if (hatchFactor > 4.0)

{

o.hatchWeights0.x = hatchFactor - 4.0;

o.hatchWeights0.y = 1.0 - o.hatchWeights0.x;

}

else if (hatchFactor > 3.0)

{

o.hatchWeights0.y = hatchFactor - 3.0;

o.hatchWeights0.z = 1.0 - o.hatchWeights0.y;

}

else if (hatchFactor > 2.0)

{

o.hatchWeights0.z = hatchFactor - 2.0;

o.hatchWeights1.x = 1.0 - o.hatchWeights0.z;

}

else if (hatchFactor > 1.0)

{

o.hatchWeights1.x = hatchFactor - 1.0;

o.hatchWeights1.y = 1.0 - o.hatchWeights1.x;

}

else

{

o.hatchWeights1.y = hatchFactor;

o.hatchWeights1.z = 1.0 - o.hatchWeights1.y;

}

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

TRANSFER\_SHADOW(o);

return o;

}

fixed4 frag(v2f i) : SV\_Target

{

fixed4 hatchTex0 = tex2D(\_Hatch0, i.uv) \* i.hatchWeights0.x;

fixed4 hatchTex1 = tex2D(\_Hatch1, i.uv) \* i.hatchWeights0.y;

fixed4 hatchTex2 = tex2D(\_Hatch2, i.uv) \* i.hatchWeights0.z;

fixed4 hatchTex3 = tex2D(\_Hatch3, i.uv) \* i.hatchWeights1.x;

fixed4 hatchTex4 = tex2D(\_Hatch4, i.uv) \* i.hatchWeights1.y;

fixed4 hatchTex5 = tex2D(\_Hatch5, i.uv) \* i.hatchWeights1.z;

fixed4 whiteColor = fixed4(1, 1, 1, 1) \* (1 - i.hatchWeights0.x - i.hatchWeights0.y - i.hatchWeights0.z -

i.hatchWeights1.x - i.hatchWeights1.y - i.hatchWeights1.z);

fixed4 hatchColor = hatchTex0 + hatchTex1 + hatchTex2 + hatchTex3 + hatchTex4 + hatchTex5 + whiteColor;

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

return fixed4(hatchColor.rgb \* \_Color.rgb \* atten, 1.0);

}

ENDCG

}

}

FallBack "Diffuse"

}



**消融效果**

消融效果的原理非常简单，概括来说就是噪声纹理+透明度测试。我们使用对噪声纹理采样的结果和某个控制消融程度的阈值比较，如果小于阈值，就使用clip函数把它对应的像素裁剪吊，这些部分就对应了图中被“烧毁”的区域。而镂空区域边缘的烧焦效果则是将两种颜色混合，在用pow函数处理后，与原纹理颜色混合后的结果。

Shader "Unlit/Dissolve"

{

Properties

{

\_MainTex ("Main Tex", 2D) = "white" {}

\_DissolveTex ("Dissolve Tex", 2D) = "white" {}

\_NormalTex ("Normal Tex", 2D) = "bump" {}

\_LineWidth ("Line Width", Range(0.0, 0.2)) = 0.1

\_BurnFirstColor ("Burn First Color", Color) = (1, 1, 1, 1)

\_BurnSecondColor ("Burn Second Color", Color) = (1, 1, 1, 1)

\_BurnAmount ("Burn Amount", Range(0.0, 1.0)) = 0.0

}

SubShader

{

Pass

{

Tags { "LightMode" = "ForwardBase" }

CGPROGRAM

#include "UnityCG.cginc"

#include "AutoLight.cginc"

#include "Lighting.cginc"

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_fwdbase

struct v2f

{

float4 pos : SV\_POSITION;

float2 mainTexUV : TEXCOORD0;

float2 normalTexUV : TEXCOORD1;

float2 dissolveUV : TEXCOORD2;

float3 tangentLightDir : TEXCOORD3;

float3 tangentViewDir : TEXCOORD4;

};

sampler2D \_MainTex;

float4 \_MainTex\_ST;

sampler2D \_DissolveTex;

float4 \_DissolveTex\_ST;

sampler2D \_NormalTex;

float4 \_NormalTex\_ST;

float \_LineWidth;

fixed4 \_BurnFirstColor;

fixed4 \_BurnSecondColor;

float \_BurnAmount;

v2f vert(appdata\_tan v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.mainTexUV = TRANSFORM\_TEX(v.texcoord, \_MainTex);

o.normalTexUV = TRANSFORM\_TEX(v.texcoord, \_NormalTex);

o.dissolveUV = TRANSFORM\_TEX(v.texcoord, \_DissolveTex);

TANGENT\_SPACE\_ROTATION;

o.tangentLightDir = mul(rotation, ObjSpaceLightDir(v.vertex).xyz);

o.tangentViewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;

return o;

}

fixed4 frag(v2f i) : SV\_Target

{

fixed4 dissolveColor = tex2D(\_DissolveTex, i.dissolveUV);

clip(dissolveColor.r - \_BurnAmount);

fixed3 tangentLightDir = normalize(i.tangentLightDir);

fixed3 tangentNormalDir = normalize(UnpackNormal(tex2D(\_NormalTex, i.normalTexUV)));

fixed3 tangentViewDir = normalize(i.tangentViewDir);

fixed3 tangentHalfDir = normalize(tangentLightDir + tangentViewDir);

fixed3 albedo = tex2D(\_MainTex, i.mainTexUV);

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

fixed3 diffuse = \_LightColor0.rgb \* saturate(dot(tangentLightDir, tangentNormalDir)) \* albedo;

fixed3 specular = \_LightColor0.rgb \* pow(saturate(dot(tangentHalfDir, tangentNormalDir)), 20);

// \_LineWidth是表示到消融点前面多大的比例从消融颜色1变化到消融颜色2

fixed ratio = 1.0 - smoothstep(0, \_LineWidth, dissolveColor.r - \_BurnAmount);

fixed4 burnColor = lerp(\_BurnFirstColor, \_BurnSecondColor, ratio);

burnColor = pow(burnColor, 5);

return fixed4(lerp(ambient + diffuse + specular, burnColor, ratio), 1.0);

}

ENDCG

}

Pass

{

Tags { "LightMode" = "ShadowCaster" }

CGPROGRAM

#include "UnityCG.cginc"

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_shadowcaster

struct v2f

{

V2F\_SHADOW\_CASTER;

fixed2 dissolveUV : TEXCOORD0;

};

sampler2D \_DissolveTex;

float4 \_DissolveTex\_ST;

float \_BurnAmount;

v2f vert(appdata\_base v)

{

v2f o;

TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET(o);

o.dissolveUV = TRANSFORM\_TEX(v.texcoord, \_DissolveTex);

return o;

}

fixed4 frag(v2f i) : SV\_Target

{

fixed4 dissolveColor = tex2D(\_DissolveTex, i.dissolveUV);

clip(dissolveColor.r - \_BurnAmount);

SHADOW\_CASTER\_FRAGMENT(i);

}

ENDCG

}

}

}



**水波效果**

在模拟实时水面的过程中，我们往往也会使用噪声纹理，此时，噪声纹理通常会用作一个高度图，以不断修改水面的法线方向。为了模拟水不断流动的效果，我们ui使用和时间相关的变量来对噪声纹理进行采样，当得到法线信息后，再进行正常的反射+折射机选，得到最后的水面波动效果。

Shader "Unlit/WaterWave"

{

Properties

{

// 水面颜色

\_Color ("Main Color", Color) = (0, 0.15, 0.115, 1)

// 水面波纹材质纹理

\_MainTex ("Texture", 2D) = "white" {}

// 噪声纹理生成的法线纹理

\_WaveMap ("Wave Map", 2D) = "bump" {}

// 模拟反射的立方体纹理

\_Cubemap ("Environment Cubemao", Cube) = "\_Skybox" {}

\_WaveXSpeed ("Wave Horizontal Speed", Range(-0.1, 0.1)) = 0.01

\_WaveYSpeed ("Wave Vertical Speed", Range(-0.1, 0.1)) = 0.01

// 模拟折射时图像的扭曲程度

\_Distortion ("Distortion", Range(0, 100)) = 10

}

SubShader

{

Tags { "Queue" = "Transparent" "RenderType" = "Opaque" }

GrabPass

{

"\_RefactionTex"

}

Pass

{

Tags { "LightMode" = "ForwardBase" }

ZWrite Off

CGPROGRAM

#include "UnityCG.cginc"

#include "AutoLight.cginc"

#pragma vertex vert

#pragma fragment frag

#pragma multi\_compile\_fwdbase

struct v2f

{

float4 pos : SV\_POSITION;

float4 scrPos : TEXCOORD0;

float4 uv : TEXCOORD1;

float4 TtoW0 : TEXCOORD2;

float4 TtoW1 : TEXCOORD3;

float4 TtoW2 : TEXCOORD4;

};

fixed4 \_Color;

sampler2D \_MainTex;

float4 \_MainTex\_ST;

sampler2D \_WaveMap;

float4 \_WaveMap\_ST;

samplerCUBE \_Cubemap;

fixed \_WaveXSpeed;

fixed \_WaveYSpeed;

sampler2D \_RefactionTex;

float4 \_RefractionTex\_TexelSize;

float \_Distortion;

v2f vert(appdata\_tan v)

{

v2f o;

UNITY\_INITIALIZE\_OUTPUT(v2f, o);

o.pos = UnityObjectToClipPos(v.vertex);

o.scrPos = ComputeGrabScreenPos(o.pos);

o.uv.xy = TRANSFORM\_TEX(v.texcoord, \_MainTex);

o.uv.zw = TRANSFORM\_TEX(v.texcoord, \_WaveMap);

float3 worldPos = mul(unity\_ObjectToWorld, v.vertex).xyz;

float3 worldNormal = UnityObjectToWorldNormal(v.normal);

float3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);

float3 worldBinormal = cross(worldNormal, worldTangent) \* v.tangent.w;

// 求得切线空间转换到世界空间的矩阵

o.TtoW0 = float4(worldTangent.x, worldBinormal.x, worldNormal.x, worldPos.x);

o.TtoW1 = float4(worldTangent.y, worldBinormal.y, worldNormal.y, worldPos.y);

o.TtoW2 = float4(worldTangent.z, worldBinormal.z, worldNormal.z, worldPos.z);

return o;

}

fixed4 frag(v2f i) : SV\_Target

{

float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);

fixed3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos));

float2 speed = \_Time.y \* float2(\_WaveXSpeed, \_WaveYSpeed);

// 两次采样，模拟两层交叉水面波动

fixed3 bump1 = UnpackNormal(tex2D(\_WaveMap, i.uv.zw + speed)).rgb;

fixed3 bump2 = UnpackNormal(tex2D(\_WaveMap, i.uv.zw - speed)).rgb;

fixed3 bump = normalize(bump1 + bump2);

// 计算折射

float2 offset = bump.xy \* \_Distortion \* \_RefractionTex\_TexelSize.xy;

// 这里乘以z是为了模拟深度越大，折射越大的效果

i.scrPos.xy = offset \* i.scrPos.z + i.scrPos.xy;

fixed3 refrCol = tex2D(\_RefactionTex, i.scrPos.xy / i.scrPos.w).rgb;

// 计算反射，bump实际上是世界坐标的法线

bump = normalize(half3(dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz, bump), dot(i.TtoW2.xyz, bump)));

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

fixed3 reflDir = reflect(-viewDir, bump);

fixed3 reflCol = texCUBE(\_Cubemap, reflDir).rgb \* texColor.rgb \* \_Color.rgb;

fixed fresnel = pow(1 - saturate(dot(viewDir, bump)), 4);

fixed3 finalColor = reflCol \* fresnel + refrCol \* (1 - fresnel);

return fixed4(finalColor, 1.0);

}

ENDCG

}

}

}



在上面的例子中我们使用了GrabPass来获取当前颜色缓冲区的数据并将它放置在\_RefractionTex纹理中，这个比较影响性能，下面使用CommandBuffer实现获取图像，可以去掉GrabPass，更灵活且效率更高

public class WaterWave : MonoBehaviour

{

private CameraEvent cameraEvent = CameraEvent.AfterForwardOpaque;

private Dictionary<Camera, CommandBuffer> cameras = new Dictionary<Camera, CommandBuffer>();

void ClearUp()

{

foreach (var camera in cameras)

{

if (camera.Key)

{

camera.Key.RemoveCommandBuffer(cameraEvent, camera.Value);

}

}

cameras.Clear();

}

void OnEnable()

{

ClearUp();

}

void OnDisable()

{

ClearUp();

}

public void OnWillRenderObject()

{

ClearUp();

var active = gameObject.activeInHierarchy && enabled;

if (!active) return;

var camera = Camera.current;

if (!camera) return;

if (cameras.ContainsKey(camera)) return;

CommandBuffer buffer = new CommandBuffer();

buffer.name = "GrabScreen";

cameras[camera] = buffer;

int screenCopyID = Shader.PropertyToID("\_ScreenCopyTexture");

buffer.GetTemporaryRT(screenCopyID, -1, -1, 0, FilterMode.Bilinear);

buffer.SetRenderTarget(BuiltinRenderTextureType.CurrentActive, BuiltinRenderTextureType.CameraTarget);

buffer.Blit(BuiltinRenderTextureType.CurrentActive, screenCopyID);

buffer.SetGlobalTexture("\_RefractionTex", screenCopyID);

camera.AddCommandBuffer(cameraEvent, buffer);

}

}



**优化技术**

1）减少draw call

Unity支持两种批处理：一种是动态批处理，另一种是静态批处理。对于动态批处理来说，有点是一切处理都是unity自动完成的，而且物体是可以移动的，但缺点是，限制很多，可能一不小心就会破坏了这种机制，导致unity无法动态批处理一些使用了相同材质的物体。而对于静态批处理来说，它的优点是自由度很高，限制很少，但缺点是可能会占用更多的内存，而且静态静态批处理后的所有物体都不可以再移动了（即使在脚本中尝试改变物体的位置也是无效的）

**动态批处理**

动态批处理的基本原理是，每一帧可可以进行批处理的模型网格进行合并，再把合并后的模型数据传递给GPU，然后使用同一个材质对其渲染。除了实现方便，动态批处理的另一个好处是，经过批处理的物体仍然可以移动，这是由于在处理每帧时unity都会重新合并一次网格。

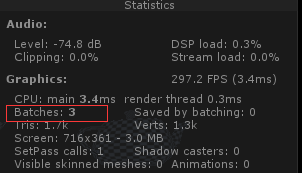
**静态批处理**

静态批处理在开始阶段，把需要进行静态批处理的模型合并到一个新的网格结构中，这意味着这些模型不可以在运行时刻被移动。但由于它只需要进行一次合并操作，因此，比动态批处理更加高效。静态批处理的一个缺点在于，它往往需要占用更多的内存来存储合并后的几何结构。这是因为，如果在静态批处理前一些物体共享了相同的网格，那么在内存中每个物体都会对应一个该网格的复制品。即一个网格会变成多个网格再发送给GPU，如果这类使用同一网格的对象很多，那么这就会称为一个性能瓶颈了。例如，如果在一个使用了1000个相同树模型的森林中使用静态批处理，那么就会多使用1000倍的内存，这会造成严重的内存影响。这种时候，解决方法要么忍受这种牺牲内存换取性能的方法，要么不要使用静态批处理，而使用动态批处理（但要小心控制模型的顶点属性数目），或者自己编写批处理的方法。

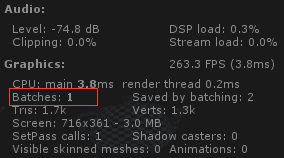
一个例子：

添加三个teapot，都是用相同的材质





可以看到，teapot有437个顶点，属性有4个，这样就不能动态批处理，然后我们使用静态批处理，将三者全都设置为静态

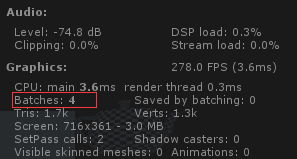


运行的时候就发现batch数为1了

我们在三个teapot中间添加一个不同材质的正方体

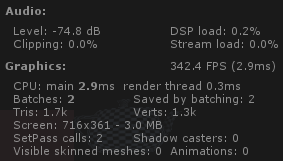
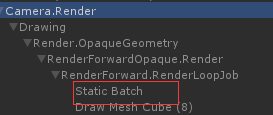


在不使用静态批处理的情况下



将三个teapot使用静态批处理的情况下：



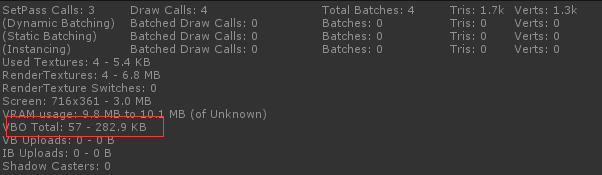
 

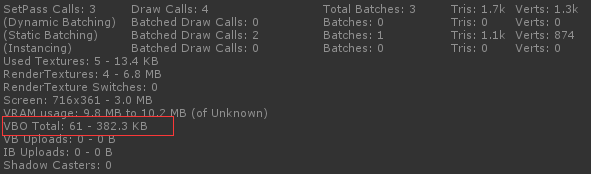
就会发现实际上将三个teapot合并成了一个，同时可以看到所有合并物体的mesh都变为了



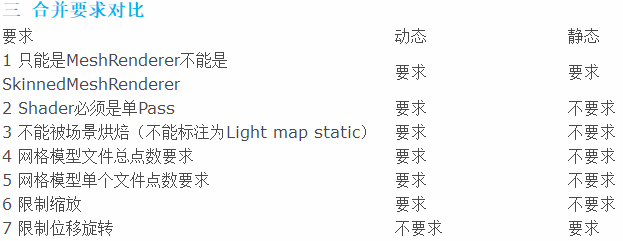
这个网格是unity合并了所有标志为“Static”的物体的结果，在我们的例子中，就是3个teapot和一个立方体，可能会有一个疑问，这4个对象命名并不是都使用了同一个材质，为什么可以合并成一个呢？从上面的图中我们可以看到有一个4 submeshes，也就是说，这个合并后的网格其实包含了4个子网格，即场景中的4个对象。对于合并后的网格，unity会判断其中使用同一个材质的子网格，然后对它们进行批处理。

下面我们也可以看看vbo数量





在内部实现上，unity首先把这些静态物体变换到世界空间下，然后为它们构建一个更大的顶点和索引缓存，对于使用了同一个材质的物体，unity只需要调用一个draw call就可以绘制全部物体，而对于使用了不同材质的物体，静态批处理同样可以提升渲染性能，尽管这物体仍然需要调用多个draw call，但静态批处理可以减少这些draw call之间的状态切换，而这些切换往往是费时的操作。从合并后的网格结构中我们还可以发现，尽管3个teapot对象使用了同一个网格，但合并后却变成了3个独立网格，而且，我们可以从unity的分析器中观察到在应用静态批处理前后vbo total的变化。静态批处理需要占用更多的内存来存储合并后的几何结构，如果一些物体共享了相同的网格，那么在内存中每一个物体都会对应一个该网格的复制品。



实验：

如果用批处理，那么传给顶点着色器的顶点坐标是世界坐标系中的坐标，unity\_Object2World实际上是一个单位矩阵

如果不用批处理，那么传给顶点着色器的顶点坐标是模型坐标系中的坐标，unity\_Object2World是模型变换矩阵

这样不论该物体是否使用批处理，UNITY\_MATRIX\_MVP都能正常变换到裁剪坐标系中

选择使用动态批处理还是静态批处理，我们有一些小小的建议：

1. 尽可能选择静态批处理，但得时刻小心对内存的消耗，并且基础静态静态批处理的物体不可以再被移动
2. 如果无法进行静态批处理，而要使用动态批处理的话，那么要小心上面提到的各种条件限制
3. 对于游戏中的小道具，例如可以得到的金币，可以使用动态批处理
4. 对于包含动画的这些物体，我们无法全部使用静态批处理，但体重如果有不动的部分，可以把这部分标志为static

除了上面的注意点外，在使用批处理时，由于会把多个模型变换到世界空间下再合并它们，因此，如果shader中存在一些基于

模型空间下的坐标的运算，那么往往得到错误的结果，一个解决办法是在shader中使用DisableBatching标签来强制使用该shader的材质不会被批处理。另一个主意事项是，使用半透明材质的物体通常需要使用严格的从后向前的绘制顺序来保证透明混合的正确性，对于这些物体，unity会首先保证他们的绘制顺序，再尝试对它们进行批处理，这意味着，当绘制顺序无法满足时，批处理无法在这些物体上被成功应用。

**减少需要处理的顶点数目**

1. 优化几何体

尽量减少模型的面数，一些对于模型没有影响，或者难以观察的顶点都要尽可能去掉。同时unity中看到的顶点数比美术看到

的顶点数要大一些，这是因为在gpu看来有时需要把一个顶顶啊拆分成两个或者更多的顶点，主要原因有两个：一个是为了分离纹理坐标，另一个是为了产生平滑的边界。

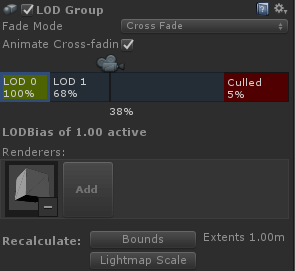
1. 模型的LOD技术

这种技术的原理是，当一个物体离摄像机很远时，模型上的很多细节是无法被观察到的，因此，LOD允许当对象逐渐原理摄像机

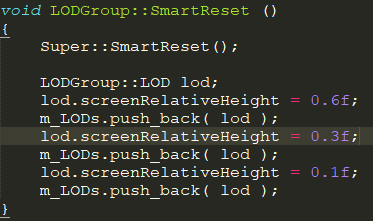
时，减少模型上的面片数量，从而提高性能。在unity中，我们可以使用LOD Group组件来为一个物体构建一个LOD，我们需要为同一个对象预备多个包含不同细节程序的模型，然后把他们赋给LOD Group组件中的不同等级，unity就会自动判断当前位置上需要使用哪个等级的模型。



在GameObject上挂脚本LodGroup：



实际上这里的比例是物体显示大小相对于屏幕大小的比例，默认为10%，30%，60%，这几个值从源码中可以看到：



接下来我们操作摄像机离物体很近时显示高模，更远时，则显示低模，直到小于5%时消失

3）遮挡剔除

遮挡剔除可以用来消除那些在其他物体后面看不到的物体，这意味着资源不会浪费在计算那些看不到的顶点上，进而提升性能。我们需要把遮挡剔除和摄像机的视椎体剔除区分开来。视椎体剔除只会剔除那些不在摄像机视野范围内的对象，但不会判断视野中是否有物体被其他物体挡住。而遮挡剔除会使用一个虚拟的摄像机来遍历场景，从而构建一个潜在可见的对象集合的层级结构，在运行时刻，每个相机及将会使用这个数据来识别那些物体是可见的，而那些被其他物体挡住不可见的。使用遮挡剔除技术，不仅可以减少处理顶顶点数目，还可以减少overdraw，提高游戏性能

**减少需要处理片元数量**

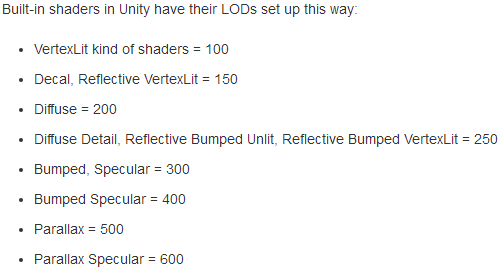
1）控制绘制顺序

由于深度测试的存在，如果我们可以保证物体都是从前往后绘制的，那么就可以很大程度上减少overdraw，这是因为，在后面绘制的物体由于无法通过深度测试，因此，就不会再进行后面的渲染处理

在unity中，那些渲染队列数目小于2500（如“Background”、“Geometry”和“AlphaTest”）的对象都被认为是不透明的物体，这些物体总体上是从前往后绘制的，而使用其他的队列（如“Transparent”、“Overlay”等）的物体，则是从后向前绘制的。这意味我们尽可能地把物体的队列设置为不透明物体的渲染队列，而尽量避免使用半透明队列。

2）Shader的LOD技术

Shader的LOD技术可以控制使用的Shader等级，它的原理是，只有Shader的LOD值小于某个设定的值，这个Shader才会被使用，而使用了那么超过设定值的Shader的物体将不会被渲染，也就是说会找到第一个lod小于设定值的subshader进行执行，但是主要默认的shader的lod，不要设置的太小，默认的都无法执行了



**基于物理的渲染（PBS）**

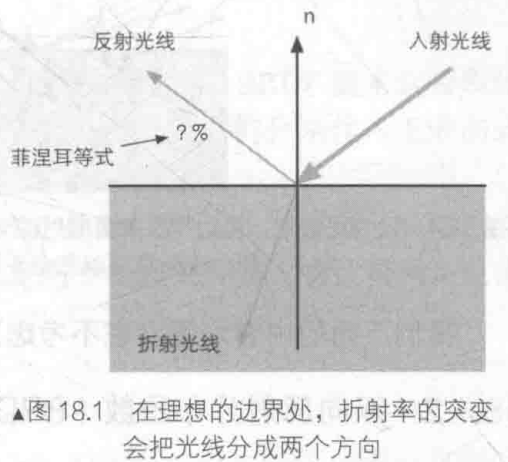
1. PBS的理论和数学基础

判断一种PBR光照模型是否是基于物理的，必须满足以下三个条件：

1. 基于微平面（Microfacet）的表面模型
2. 能量守恒
3. 应用基于物理的BRDF

在现实生活中，光和物体之间的交互过程是非常复杂的，大多数情况下并不存在一种可分析的解决方法，但为了在渲染中对光照

进行建模，我们往往只考虑一种特殊情况，即只考虑两个介质的边界是无限大并且是光学平滑的。尽管真实物体的表面并不是无限延伸的，也不是绝对光滑的，但和光的波长相比，它们的大小可以被近似认为是无限大以及光学平滑的。在这样的前提下，光在不同介质的边界会被分割成两个方向：反射方向和折射方向。而有多少百分比的光会被发射（另一部分就是被折射了）则是由菲涅尔等式来描述的。



但是，这些与光线的交接处真的是像镜子一样平坦吗？尽管在上面我们已经说过，相对于光的波长来说，它们的确可以被认为是光学平坦的。但是，如果想象我们有一个高倍放大镜，去放大这些被照亮的物体表面，就会发现有很多之前肉眼不可见的凹凸不平的平面，在这种情况下，物体的表面和光照发生的各种行为，更像是一系列微小的光学平滑平面和光交互的结果，其中每个小平面会把光分割成不同的方向。这种建立在微表面的模型更容易解释为什么有些物体看起来粗糙，而有些看起来就平滑，如下图所示，想象我们用一个放大镜去观察一个光滑物体的表面，尽管它的表面仍然由许多凹凸不平的微表面构成，但这些微表面的法线方向变化角度小，因此，由这些表面发射的光线方向变化也比较小，这使得物体的高光反射更加清晰，而粗糙表面则相反，由此得到的高光反射效果更模糊。产生的效果就是：一个平面越是粗糙，这个平面上的微平面的排列就越混乱，这些微小镜面这样无序取向排列的影响就是，当我们特指镜面光/镜面反射时，入射光线更趋向于向完全不同的方向发散开来，进而产生出分部范围更广泛的镜面反射。而与之相反的是，对于一个光滑平面，光线大体上会趋向于向同一个方向反射，造成更小更锐利的反射。

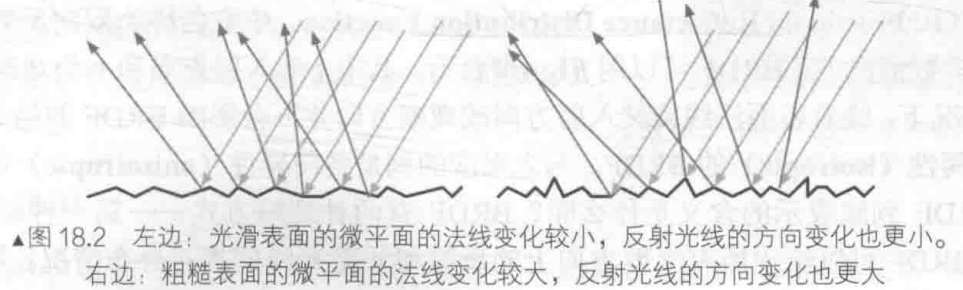
为了遵守能量守恒定律，我们需要对漫反射光和镜面发射光之间做出明确的区分，当一束光线碰撞到一个表面的时候，它就会分离成一个折射部分和一个反射部分，反射部分就是会直接发射开来而不会进入平面的那部分光线，这就是我们所说的镜面光照，而折射部分就是余下的会进入表面并被吸收的那部分光线，这也就是我们所说的漫反射光照。

一般来说，并非所有能量都会被全部吸收，而光线也会继续沿着随机方向发散，然后再和其他的例子碰撞直至能量完全耗尽或者在此离开这个表面，而光线脱离物体表面后将会协同构成该表面（漫反射）的颜色，不过在就物理的渲染之中我们进行简化，假设对平面上的每一点所有的折射光都会被完全吸收而不会散开，而有一些被称为次表面散射技术的着色器技术将这个问题考虑了进去，它们显著的提升了一些诸如皮肤、大理石或者蜡质这样材质的视觉效果，不过伴随而来的则是性能下降代价。

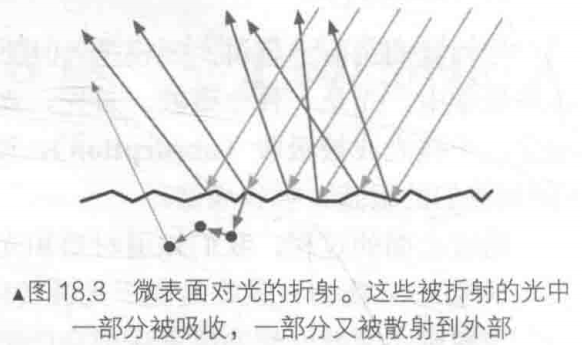
对于金属表面，金属表面对光的反应与非金属材质还有电介质材料表面相比是不同的，他们遵从的反射与折射原理是相同的，但是所有的折射光都会被直接吸收而不会散开，只留下反射光或者说镜面反射光。也就是说，金属表面不会显示出漫反射颜色，由于金属与电介质之间存在这样明显的区别，因此它们两者在PBR渲染管线中被区别处理。

我们按照能量守恒的关系，首先计算出镜面反射部分，它的值等于入射光线被反射能量所占的百分比，然后折射光部分就可以直接由镜面反射部分计算得到：

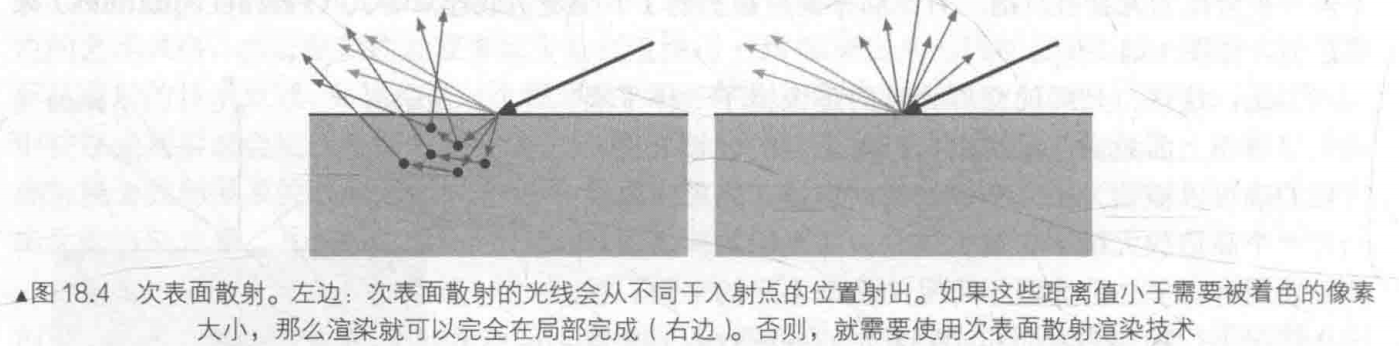




在上面的内容中，我们并没有讨论那些被微表面折射的光。这些光被折射到物体的内部，一部分被介质吸收，一部分又被散射到外部。金属材质具有很高的吸收系数，因此，所有被折射的光往往会被立刻吸收，被金属内部的自由电子转化成其他形式的能量。而非金属材质则会同时表面出吸收和散射两种现象，这些在折射之后又被反射出去的光又被称为次表面散射光（subsurface-scattered light），下图中我们给出了一条由微表面折射的光的传播路径。



现在，我们把放大镜从物体表面拿开，继续从渲染的层级大小上考虑光与表面一点的交互行为。那么，由微表面反射的光可以被认为是该点上一些方向变化不大的反射光，如黄线所示，而折射光线（蓝线）则需要更多的考虑。那些次表面散射光会从不同入射点的位置从物体内部再次射出，而这些离入射点的距离值和像素大小之间的关系会产生两种建模结果。如果像素要大于这些散射距离的话，意味着这些次表面散射产生的距离可以被忽略，那我们的渲染就可以在局部进行，如果像素要小于这些散射距离，我们就不可以选择忽略它们了，要实现更真实的次表面散射效果，我们需要使用特殊的渲染模型，也就是所谓的次表面散射渲染技术。



下面的内容均建立在不考虑次表面散射的距离，而完全使用局部着色渲染的前提下：

（2）双向反射分布函数（Bidirectional Reflectance Distribution Function）

我们可以用**辐射率**来量化光。辐射率是单位面积、单位方向上光源的辐射通量，通常用L来表示，被认为是对单一光线的亮度和颜色评估。在渲染中，我们通常会基于表面的入射光线的入射辐射率Li来计算出射辐射率L0，这个过程也往往被称为是着色过程。

要得到出射辐射率L0，我们需要知道物体表面一点是如何和光进行交互的，而这个过程就可以使用BRDF（双向反射分布函数）来定量分析，多数情况下BRDF可以用f(I,v)来表示，其中I为辐射方向，v为观察方向（双向的含义）。这种情况下，绕着表面法线旋转入射方向或者观察方向并不会影响BRDF的结果，这种BRDF被称为是各项同性的BRDF，与之对应的则是各项异性的BRDF

那么，BRDF到底表示的含义是什么呢？BRDF有两种理解方式---第一种理解是，当给定入射角度后，BRDF可以给出所有出射方向上的反射和散射光线的分部情况；第二种理解是，当给定观察方向（即出射方向）后，BRDF可以给出从所有入射方向到该出射方向的光线分布。一个更直观的理解就是，当一束光线沿着入射方向I到达表面某点时，f(I,v)表示有多少部分的能量被发射到了观察方向v上，据此，给出基于物理渲染的技术中，第一个重要的等式---反射等式



反射等式实际上是渲染方程的一个特殊情况，但它是基于物理基础的。尽管上面的式子看起来有些复杂，但很好理解，即给定观察视角v，该方向的出射辐射率L0(v)等于所有入射方向的辐射率积分乘以它的BRDF值f(I,v)，再乘以一个余弦值(n,I)。我们使用更简单的方式来理解，想象我们现在要计算表面上某点的出射辐射率，我们已知到该点的观察方向，该点的出射辐射率是由许多不同方向的入射辐射率叠加后的结果。其中，BRDF表示了不同方向的入射光在该观察方向上的权重分布，**我们把这些不同方向的光的辐射率(Li(I))乘以观察方向上所占的权重（f(I,v)部分），再乘以它们在该表面的投影结果（(n\*I)部分），最后再把这些值加起来（即做积分）就是最后的出射辐射率**

Cook-Torrance光照模型