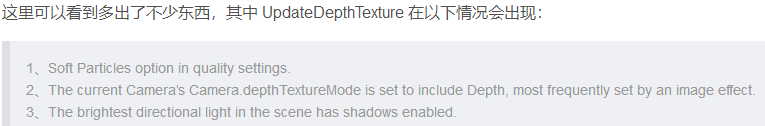
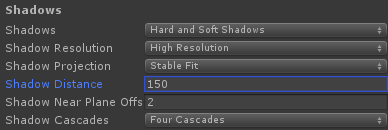
[**https://blog.csdn.net/NotMz/article/details/82053659**](https://blog.csdn.net/NotMz/article/details/82053659)



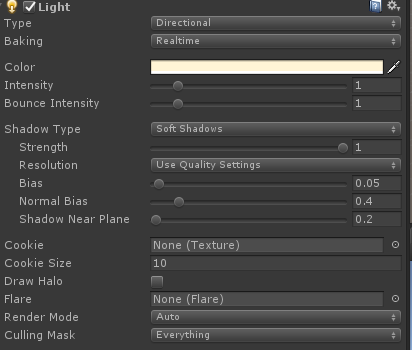
**平行光阴影**

Unity不支持半阴影，unity支持的是软阴影，但软阴影是一种阴影滤波技术，而不是半阴影的模拟

阴影是项目级别的质量设置的一部分，通过Edit->Project Setting->Quality可以找到具体的设置项，我们将在高质量等级中启动阴影。高质量等级意味着支持硬阴影和软阴影，使用高分辨率、稳定的适配投影、150的投影距离以及四个联级阴影贴图。

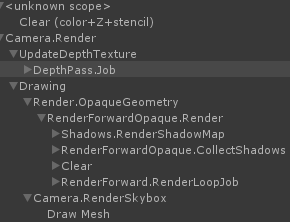


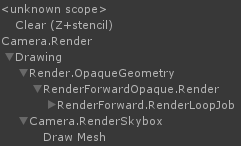
确保方向光源都设置为投射软阴影，他们的分辨率应该取决于质量设置。



**阴影贴图**

有几种技术都能支持实时阴影，每种技术都有优点和缺点。Unity使用的是现在最常见的技术，即阴影贴图。这意味着Unity以纹理方式存储阴影信息。





通过frame debugger看到有阴影有没有阴影的渲染情况，当禁用阴影时，所有物体都会照常进行渲染，但是当启用阴影的时候，渲染过程变得更加复杂，渲染阶段变多了，还多了非常多的绘制调用，所有渲染阴影是非常昂贵的。

**渲染到深度纹理**

当启用平行光阴影的渲染选项时，unity开始在渲染过程中加入一个深度渲染pass，所得到的结果会放一张与屏幕分辨率相匹配的纹理贴图中去，这个深度渲染pass会渲染整个场景，但是仅记录每个片段的深度信息，这是与图形处理器用来确定片段与先前渲染的片段的深度相对关系相同的信息。

**渲染到阴影贴图之中**

Unity渲染的下一个内容是第一个光源的阴影贴图吗，稍后，它将渲染第二个光源的阴影贴图。再一次，整个场景都被渲染了一遍，并且同样也是只有深度信息被存储在纹理贴图之中。然而，这一次是从光源的位置来渲染整个场景，最为特别的是，光源作为摄像机，这意味着深度值告诉我们的是在光线射到某物之前到底行进了有多远，这可以用来确定是否有东西被遮挡。

事实上，unity不只对每个光源都要渲染整个场景一次，而是对每个光源都要渲染整个场景四次！纹理贴图被分成四个象限，每个象限从不同的角度进行渲染。这是因为我们选择使用四个级联阴影。如果你要切换到两个级联阴影，场景将对每个光源渲染整个场景两次。如果没有级联阴影的话，它只是对每个光源渲染整个场景一次。当我们光擦阴影的质量时，我们将看到为什么unity会这么做。

**收集阴影**

我们现在有从相机视角得到的场景深度信息。我们也有从每个光源的视角得到的场景深度信息。当然，这些数据存储在不同的裁剪空间中，但我们知道这些空间的相对位置和方向，所以我们可以从一个空间转换到另一个空间中去，这允许我们从两个视角来比较深度的测量。

**当线程相机无法看到一个点的时候怎么办呢？**

对每个光源而言的场景空间的阴影，unity通过渲染覆盖整个视图的单个四边形来创建这些纹理，它使用Hidden/Internal-ScreenSpaceShadows这个着色器进行这个pass的渲染，从场景相机和光源的深度纹理贴图中进行每个片段的采样、进行比较，并将最终的阴影值渲染为屏幕空间的阴影贴图。被照亮的纹理像素设置为1，阴影贴图的纹理像素设置为0，此时，unity还可以执行过滤，来创建那些柔和的阴影。

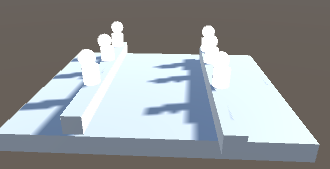
**阴影质量**

当从光源的角度渲染整个场景时，光源的方向与场景相机的方向不匹配，因此，阴影贴图的纹理像素不与最终图像的纹理像素一一对应，阴影贴图的分辨率也与最终图像的分辨率不同，最终图像的分辨率由显示设置类决定，而阴影贴图的分辨率是由阴影质量来决定的。

当阴影贴图的纹理像素最终渲染的时候比最终图像的纹理像素大的时候，它们将变得非常明显，阴影的边缘将会出现走样，这在硬阴影的时候最为明显。为了尽可能使这个走样更明显，我们更改阴影质量设置，所有我们只在场景中得到硬阴影，分辨率为最低，并且没有级联



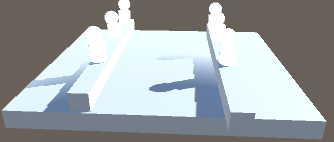




在阴影越靠近场景摄像机的地方，阴影的纹理像素就变得越大，这是因为阴影贴图目前覆盖场景摄像机可见的整个区域，我们可以通过在质量设置减少阴影覆盖的距离来提高相机附近的阴影质量。

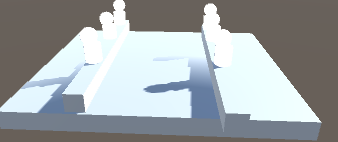
这面这是Shadow Distance = 20的情况，远处的阴影消失了





通过将阴影限制到接近场景摄像机的区域，我们可以使用相同的阴影贴图来覆盖更小的区域，结果就是，我们得到效果更好的阴影，但我们失去了在远离场景摄像机区域的阴影，阴影随着这些区域接近最大距离而消失。

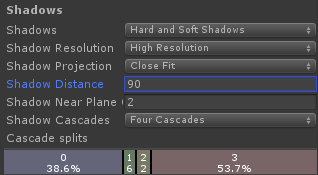
在最理想的情况下，我们能够在接近场景社详细的区域得到高质量的阴影，同时还能在远离场景摄像机的区域保持阴影，因为远处的阴影最终渲染到更小的屏幕区域，那些阴影可以使用较低分辨率的阴影贴图，这是通过级联阴影做的。启动级联阴影的时候，多个阴影贴图将渲染到同一个纹理贴图中，每个阴影贴图只用于一定距离的区域。使用四个级联的低精度阴影显示如下：

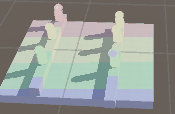


使用四个级联阴影的低精度阴影贴图所得到的效果好多了，缺点是我们现在必须在多渲染三次场景。当渲染到屏幕空间的阴影贴图的时候，unity会从正确的级联阴影中进行采样，你可以通过选在阴影纹理尺寸的突然变化来找到一个级联阴影结束和另一个级联阴影开始的位置。



级联区域的形状取决于阴影投影质量的设置，阴影投影质量设置的默认值为“稳定适配”，在这个模式下，可以根据渲染点到相机位置的距离来渲染级联区域。另一个选项时“关闭适配”，那么级联区域的形状会使用相机的深度来决定，这会产生在相机的观察方向上的矩形区域带。在左上角的向下的箭头选择Shadow Cascades，可以看到级联区域，我们在设置中的阴影质量设置四级级联，可以看到如下结果，并且它会随着摄像机的位置变化而变化：







**阴影瑕疵**

当我们使用低质量的硬阴影时，我们看到一些阴影出现在他们不应该出现的地方，不幸的是，不管质量设置如何，这种情况都可能发生。由于阴影贴图受限于解析度，在距离光源比较远的情况下，多个片元可能从深度贴图的同一个值去采样（这是很容易想象到的，在摄像机的片元渲染中很可能有多个片元都在光源方向上的同一条直线上，它们就会在深度纹理中采样同一个纹素，因为它们在光源方向上对应到了屏幕的同一个像素），这样，就有可能因为精度问题有些片元被认为是在阴影之中，有些不在。这种情况下，我们可以用一个叫做阴影偏移（shadow bias）的技巧来解决这个问题，我们简单对表面的深度（或深度贴图）应用一个偏移量，这些片元就不会被错误地认为在表面之下了，使用偏移量之后，所有采样点都获得了比表面深度更小的深度值



一个0.005的偏移量就能帮到很大的忙（实际上这个值越大，我们就会发现靠近物体的一部分阴影就没有了，这个很好理解，因为偏移量较大，靠近的一些片元就被认为不在阴影中了），但是有些表面坡度很大，仍然会产生阴影失真，有一个更加可靠的办法能够根据表面朝向光源的角度更改偏移量：使用点乘：



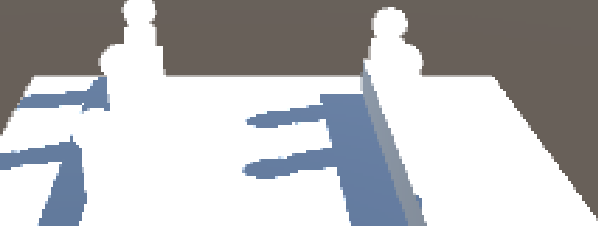
这里我们有一个偏移量的最大值0.05和一个最小值0.005，他们是基于表面法线和光照方向的，也就是说如果表面与光源垂直，就将偏移量变大一点，角度越小，沿着光线的采样点与当前判断的片元越远，偏移量可以变小一些。

在unity中，当将bias设置为0时：









除了这个距离偏移意外，还有一个法线偏移，这是对阴影投射提的微调，这种偏移将阴影投射提的顶点沿着它们的法线向内推，这也减少了自阴影的出现，但将阴影投射提的顶点沿着它们的法线向内推也会使阴影更小，并可能导致在阴影中出现孔。

**抗锯齿**

如果在质量设置中启用了抗锯齿，那么可能会发现阴影贴图这种技术存在的另外一个问题，那就是它们不能与标准的抗锯齿技术混合在一起使用。当在质量设置中启用抗锯齿的时候，unity将使用多重采样抗锯齿技术，也就是MSAA。多重采样抗锯齿技术通过沿着三角形边缘执行一些超采样来去除三角形边缘的混叠。细节纹理贴图对多重采样抗锯齿技术没有什么影响。重要的是，当unity渲染屏幕空间的阴影贴图时，它会使用覆盖整个视图的单个四边形。因此，在屏幕空间的阴影贴图中根本就没有三角形边缘，因此多重采样抗锯齿技术不会影响屏幕空间阴影贴图。多重采样抗锯齿技术支队最终图像起作用，但是阴影值是从屏幕空间的阴影贴图中直接获取的。

**投射阴影**

我们知道Unity会为平行光阴影多次渲染场景，其中一次渲染场景时为了深度pass，并且还会为每个光源每个级联阴影贴图渲染场景一次，屏幕空间的阴影贴图是一个屏幕空间效果，不会涉及到我们。为了支持所有相关的pass，我们必须向我们的着色器添加一个pass，在这个pass中光照模式设置为ShadowCaster，因为我们只对深度值感兴趣，所以它会比我们的其他pass更加简单。

Pass

{

Tags { "LightMode" = "ShadowCaster" }

CGPROGRAM

#pragma target 3.0

#pragma vertex MyShadowVertexProgram

#pragma fragment MyShadowFragmentProgram

#include "UnityCG.cginc"

struct VertexData

{

float4 vertex : POSITION;

};

float4 MyShadowVertexProgram(VertexData v) : SV\_POSITION

{

return UnityObjectToClipPos(v.vertex);

}

fixed4 MyShadowFragmentProgram() : SV\_Target

{

return 0;

}

ENDCG

}

添加下面的pass后就可以投射平行光阴影了

**偏移**

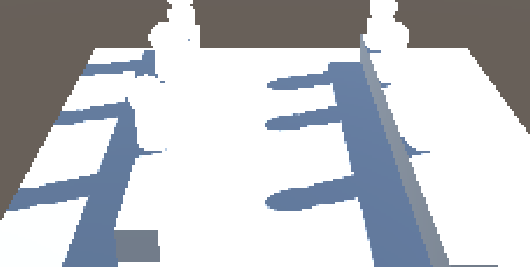
我们还必须支持阴影的偏移，在深度pass渲染期间，这个阴影的偏移为0，但是当渲染阴影贴图的时候，阴影的偏移会相对光源进行设置，我们可以通过应用深度偏差到顶点着色器中顶点在裁剪空间的位置来实现这一点，为了支持阴影的深度偏移，我们可以使用在UnityCG中定义的UnityAppliLinearShadowBias函数

float4 MyShadowVertexProgram(VertexData v) : SV\_POSITION

{

return UnityObjectToClipPos(v.vertex);

}



float4 MyShadowVertexProgram(VertexData v) : SV\_POSITION

{

float4 position = UnityObjectToClipPos(v.vertex);

return UnityApplyLinearShadowBias(position);

}



从上面的结果我们可以看出，加了偏移之后一些阴影瑕疵没有了，光源的shadow bias也可以生效了，否则不会生效（已经验证）

看看UnityApplyLinearShadowBias是如何工作的？

float4 UnityApplyLinearShadowBias(float4 clipPos)

{

clipPos.z += saturate(unity\_LightShadowBias.x/clipPos.w);

float clamped = max(clipPos.z, clipPos.w\*UNITY\_NEAR\_CLIP\_VALUE);

clipPos.z = lerp(clipPos.z, clamped, unity\_LightShadowBias.y);

return clipPos;

}

UnityApplyLinearShadowBias函数是在裁剪空间中线性增加z坐标的值，让这个事情变复杂的原因是它正在使用的是齐次坐标，它必须对透视投影进行补偿，以便偏移不随着渲染的位置离相机的距离变化而变化，它还必须确保结果不会超出范围。

为了支持阴影的法线偏差，我们必须根据法线向量来移动顶点的位置，所有我们必须在顶点数据中加上法线数据。然后我们可以使用UnityClipSpaceShadowCasterPos函数来应用这个法线偏差

float4 MyShadowVertexProgram(VertexData v) : SV\_POSITION

{

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

return UnityApplyLinearShadowBias(position);

}

使用这个函数以后我们就会发现调节Normal Bias会有作用

看看UnityClipSpaceShadowCasterPos是如何工作的？

float4 UnityClipSpaceShadowCasterPos(float3 vertex, float3 normal)

{

float4 clipPos;

// Important to match MVP transform precision exactly while rendering

// into the depth texture, so branch on normal bias being zero.

if (unity\_LightShadowBias.z != 0.0)

{

float3 wPos = mul(unity\_ObjectToWorld, float4(vertex,1)).xyz;

float3 wNormal = UnityObjectToWorldNormal(normal);

float3 wLight = normalize(UnityWorldSpaceLightDir(wPos));

// apply normal offset bias (inset position along the normal)

// bias needs to be scaled by sine between normal and light direction

// (http://the-witness.net/news/2013/09/shadow-mapping-summary-part-1/)

//

// unity\_LightShadowBias.z contains user-specified normal offset amount

// scaled by world space texel size.

float shadowCos = dot(wNormal, wLight);

float shadowSine = sqrt(1-shadowCos\*shadowCos);

float normalBias = unity\_LightShadowBias.z \* shadowSine;

wPos -= wNormal \* normalBias;

clipPos = mul(UNITY\_MATRIX\_VP, float4(wPos,1));

}

else

{

clipPos = UnityObjectToClipPos(vertex);

}

return clipPos;

}

这个函数将顶点的位置转换到世界空间中去，然后应用正常偏置，在转换到裁剪空间中去，精确的偏移取决于法线和光线方向之间的角度，以及阴影纹理像素的尺寸。

**接受阴影**

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

}

}

}

#define TRANSFER\_SHADOW\_CASTER\_NORMALOFFSET(o) TRANSFER\_SHADOW\_CASTER\_NOPOS(o,o.pos)

#define TRANSFER\_SHADOW\_CASTER\_NOPOS(o,opos) \

opos = UnityClipSpaceShadowCasterPos(v.vertex.xyz, v.normal); \

opos = UnityApplyLinearShadowBias(opos);

这里我们可以看到和我们上面注释掉的代码一样

#define SHADOW\_CASTER\_FRAGMENT(i) return 0;

此时直接返回，因为这里光照方向的阴影贴图实际上就是正常渲染的z值，它会自动插值进入深度纹理，这个pass就放入到了阴影贴图，也即可以产生阴影

关于接受阴影：

1）SHADOW\_COORDS(2) 实际上定义了一个float4 \_ShadowCoord : TEXCOORD2

2）TRANSFER\_SHADOW(o)

查看源码：

// 没有定义屏幕空间阴影，则直接得到光源空间下的坐标

#if defined(UNITY\_NO\_SCREENSPACE\_SHADOWS) UNITY\_DECLARE\_SHADOWMAP(\_ShadowMapTexture);

#define TRANSFER\_SHADOW(a) a.\_ShadowCoord = mul( unity\_WorldToShadow[0], mul( unity\_ObjectToWorld, v.vertex ) );

// 定义了屏幕空间阴影，则通过ComputeScreenPos得到

sampler2D \_ShadowMapTexture;

#define TRANSFER\_SHADOW(a) a.\_ShadowCoord = ComputeScreenPos(a.pos); // 其中a.pos是裁剪空间坐标系的位置

**看看ComputeScreenPos是如何工作的**

inline float4 **ComputeNonStereoScreenPos**(float4 pos) {

float4 o = pos \* 0.5f;

#if defined(UNITY\_HALF\_TEXEL\_OFFSET)

o.xy = float2(o.x, o.y\*\_ProjectionParams.x) + o.w \* \_ScreenParams.zw;

#else

o.xy = float2(o.x, o.y\*\_ProjectionParams.x) + o.w;

#endif

o.zw = pos.zw;

return o;

}

inline float4 **ComputeScreenPos** (float4 pos) {

float4 o = ComputeNonStereoScreenPos(pos);

#ifdef UNITY\_SINGLE\_PASS\_STEREO

o.xy = TransformStereoScreenSpaceTex(o.xy, pos.w);

#endif

return o;

}

其中UNITY\_SINGLE\_PASS\_STEREO用于控制在不同平台上计算方式的差异，而UNITY\_HALF\_TEXEL\_OFFSET用于支持DirectX 9的像素偏移。该函数返回的是齐次坐标下的屏幕坐标值

剔除不必要的代码，核心代码为：

inline float4 ComputeScreenPos (float4 pos)

{

float4 o = pos \* 0.5f;

o.xy = float2(o.x, o.y\*\_ProjectionParams.x) + o.w;

o.zw = pos.zw; return o;

}

首先该函数传入的参数pos为顶点变换到齐次坐标系下的坐标，也就是说我们在调用该函数的时候需要先将模型空间下的顶点坐标转换为齐次坐标系下的顶点坐标

我们先看正常流程的屏幕坐标是如何计算的

ScreenX = ((x / w) \* 0.5 + 0.5) \* width

ScreenY = ((y / w) \* 0.5 + 0.5) \* height

我们再看看ComputeScreenPos是如何做的，它的计算方法为

o.x = (pos.x \* 0.5 + pos.w \* 0.5)

o.y = (pos.y \* 0.5 \* \_ProjectionParams.x + pos.w \* 0.5)

其中\_ProjectionParams.x用于在使用翻转投影矩阵时（此时其值为-1.0）翻转y的坐标值，现在我们将正常屏幕坐标计算通过乘以w

W \* screenX = (x \* 0.5 + w \* 0.5) \* width

W \* screenY = (y \* 0.5 + w \* 0.5) \* height

现在一目了然了，ComputeScreenPos返回的值是齐次坐标系下的屏幕坐标值，其范围为[0,w]。

那么unity为什么要这么做呢？

这是因为如果在顶点着色器中除的话，会破坏插值结果，从顶点着色器到偏远着色器会有一个插值过程，如果我们队x/w、y/w进行插值，结果会不准确，因为投影空间不是线性空间，插值往往是线性的，所以不要在投影空间进行插值

Unity本意是希望你把该坐标值用作tex2Dproj指令的参数值，tex2Dproj会在对纹理采样前除以w分量。当然你也可以自己除以w分量后进行采样，但是效率不如内置指令tex2Dproj

pos = UnityObjectToClipPos(v.vertex);

screenPos = ComputeScreenPos(o.pos);

tex2D(\_ScreenTex, float2(screenPos.xy / screenPos.w))

3）SHADOW\_ATTENUATION

#define SHADOW\_ATTENUATION(a) unitySampleShadow(a.\_ShadowCoord)

// 不使用屏幕空间阴影技术

#if defined(UNITY\_NO\_SCREENSPACE\_SHADOWS)

inline fixed unitySampleShadow (unityShadowCoord4 shadowCoord)

{

#if defined(SHADOWS\_NATIVE)

fixed shadow = UNITY\_SAMPLE\_SHADOW(\_ShadowMapTexture, shadowCoord.xyz);

shadow = \_LightShadowData.r + shSHADOW\_COORDSadow \* (1-\_LightShadowData.r);

return shadow;

#else

unityShadowCoord dist = SAMPLE\_DEPTH\_TEXTURE(\_ShadowMapTexture, shadowCoord.xy);

// tegra is confused if we use \_LightShadowData.x directly

// with "ambiguous overloaded function reference max(mediump float, float)"

unityShadowCoord lightShadowDataX = \_LightShadowData.x;

unityShadowCoord threshold = shadowCoord.z;

**// 通过光源空间的z值与阴影贴图的z值进行比较**

return max(dist > threshold, lightShadowDataX);

#endif

}

#else // UNITY\_NO\_SCREENSPACE\_SHADOWS

sampler2D \_ShadowMapTexture;

#define TRANSFER\_SHADOW(a) a.\_ShadowCoord = ComputeScreenPos(a.pos);

inline fixed unitySampleShadow (unityShadowCoord4 shadowCoord)

{

**// 直接采样**

fixed shadow = tex2Dproj( \_ShadowMapTexture, UNITY\_PROJ\_COORD(shadowCoord) ).r;

return shadow;

}