[2024.02.01]Unity中的基础光照

模拟物理世界的光

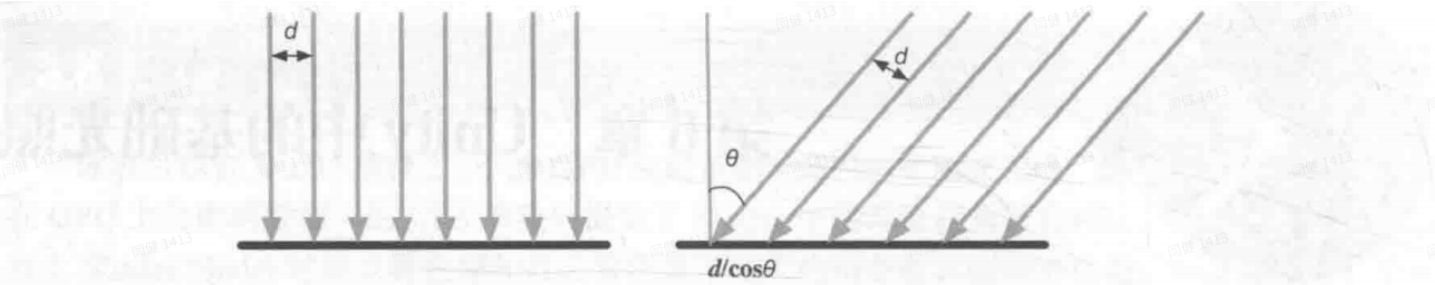
通常来讲，我们要模拟真实的光照环境来生成一张图像时，需要考虑3种物理现象：

1. 首先，光线从光源中射出。
2. 之后，光线和场景中的一些物体相交，一些光线被物体吸收了，另一些光线则被散射到其他方向。
3. 最后，摄像机也吸收了一部分光，产生了一张图像。

光源

光是由光源发射出来的，在实时渲染中，我们通常把光源当成一个没有体积的点，用l来表示它的方向。同时，我们使用**辐照度(irradiance)**来量化光，对于平行光来说，它的辐射度可以通过计算在垂直于l的单位面积上，单位时间内穿过的能量来得到。

在计算光照模型时，我们需要知道一个物体表面的辐照度，可以使用光源方向l和表面法线n之间的夹角的余弦值来得到。这里默认方向的向量的模都为1。



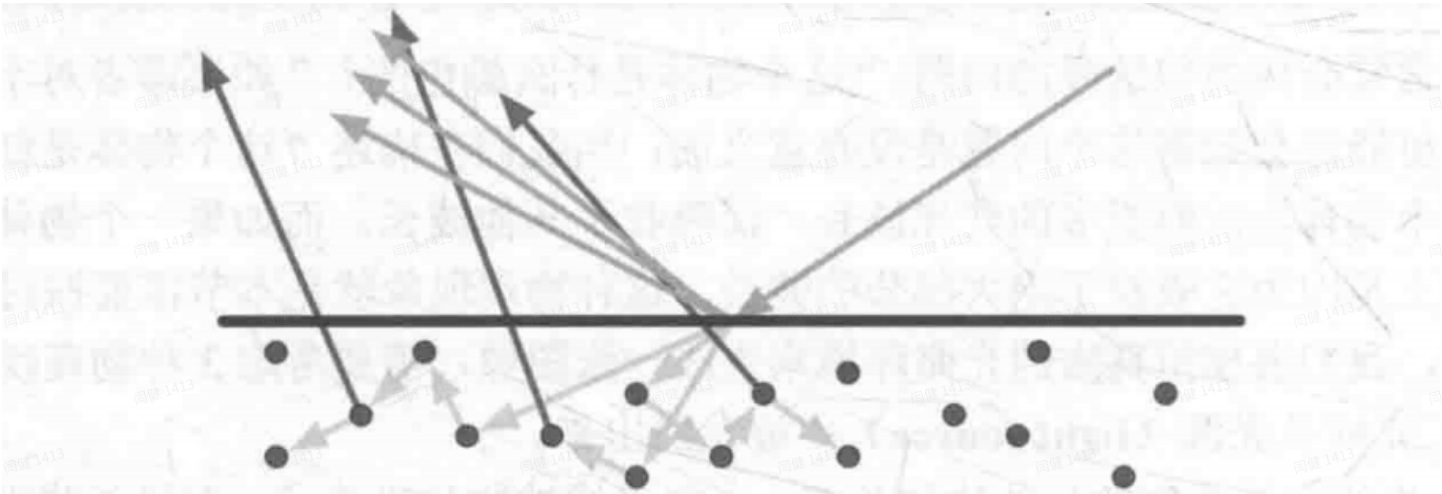
▲图 6.1　在左图中，光是垂直照射到物体表面，因此光线之间的垂直距离保持不变；而在右图中，光是斜着照射到物体表面，在物体表面光线之间的距离是 d/cosθ，因此单位面积上接收到的光线数目要少于左图

因为辐照度是和照射到物体表面时光线之间的距离d|cosθ|成反比的，因此辐照度就和cosθ成正比。cosθ可以使用光源方向l和表面法线n的点积来得到。

吸收和散射

光线由光源发射出来后，就会与一些物体相交。通常，相交的结果有两个：**散射和吸收**。

散射只改变光的方向，但不改变光线的密度和颜色。而吸收只改变光线的密度和颜色，但不改变光线的方向。光线在物体表面经过散射后，有两种方向：一种将会散射到物体内部，这种现象被称为**折射或透射**，另一种将会散射到外部，这种现象被称为反射。对于不透明物体，折射进入物体内部的光线还会继续与内部的颗粒进行相交，其中一些光线最后会重新发射出物体表面，而另一些则被物体吸收。那些从物体表面重新发射出的光线将具有和入射光线不同的方向分布和颜色。



▲图 6.2　散射时，光线会发生折射和反射现象。对于不透明物体，折射的光线会在物体内部继续传播，最终有一部分光线会重新从物体表面被发射出去

为了区分这两种不同的散射方向，我们在光照模型中用了不同的部分来计算它们：

1. **高光反射（specular）**部分表示物体表面如何反射光线。
2. **漫反射（diffuse）**部分则表示有多少光线会被折射、吸收和散射出表面。

根据入射光线的数量和方向，我们可以计算出射光线的数量和方向，通常我们使用**出射度**来描述它。辐照度和出射度之间是满足线性关系的，而它们之间的比值就是材质的漫反射和高光反射属性。

着色

着色指的是，根据材质属性（如漫反射属性等）、光源信息（如光源方向、辐照度等），使用一个等式去计算沿某个观察方向的出射度的过程。我们也把它称为**光照模型（Lighting Model）**。不同的光照模型有不同的目的，例如有些用于描述粗糙的物体表面，一些用于描述金属表面等。

BRDF光照模型

当已知光源位置和方向、视角方向时，我们就需要知道一个表面是和光照进行交互的。而**BRDF（Bidirectional Reflectance Distribution Function）**就是用来回答，光线与表面交互时，有多少光线被反射，反射方向有哪些的。

当给定表面模型上的一个点时，BRDF包含了对该点外观的完整描述。在图形学中，BRDF大多会使用一个数学公式来表示，并且提供了一些参数来调整材质属性。当给定入射光线的方向和辐照度后，BRDF可以给出在某个出射方向上的光照能量分布。

标准光照模型

标准光照模型只关心直接光照，也就是哪些直接从光源发射出来照射到物体表面后，经过物体表面的一次反射直接进入摄像机的光线。

它的基本方法是，把进入到摄像机内的光线分为4个部分，每个部分使用一种方法来计算它的贡献度，它们分别是：

1. **自发光（emissive）**，这个部分用于描述当给定一个方向时，一个表面本身会向该方向发射多少辐射量。但如果没有使用全局光照（global illumination），这些自发光的表面不会真的照亮周围的物体，而是它本身看起来更亮了而已。
2. **高光反射（specular）**，这部分用于描述当光线从光源照射到模型表面时，该表面会在完全镜面反射方向散射多少辐射量。
3. **漫反射（diffuse）**，这部分用于描述当光线从光源照射到模型表面时，该表面会向每个方向散射多少辐射量。
4. **环境光（ambient）**，这部分用于描述其他所有间接光照。

环境光（ambient）

虽然标准光照模型的重点在于描述直接光照，但在真实的世界中，物体也可以被**间接光照（indirect light）**所照亮。间接光照指的是，光线通常会在多个物体之间反射，最后进入摄像机，也就是说，在光线进入摄像机之前，经过了不止一次的物体反射。例如，在红地毯上放置一个浅灰色的沙发，那么沙发底部也会有红色，这些红色是由红地毯烦设了一部分光线，再反射到沙发上的。

在标准光照模型中，我们使用了一种被称为环境光的部分来近似模拟间接光照。环境光的计算非常简单，它通常是一个全局变量，即场景中所有物体都使用这个环境光，其计算公式为：

$$c_{ambient}=g_{ambient}$$

自发光（emissive）

光线也可以由光源发射进入摄像机，而不需要经过任何物体的反射。标准光照模型使用自发光来计算这部分的贡献度。它直接使用了该材质的自发光颜色：

$$c_{emissive}=m_{emissive}$$

通常在实时渲染中，自发光的表面往往不会照亮周围的表面，即它不会被认为是光源。全局光照系统则可以模拟这类自发光物体对周围物体的影响。

漫反射（diffuse）

漫反射光照是用于对那些被物体表面随机散射到各个方向的辐射度进行建模的。在漫反射中，视角的位置不重要，因为反射是完全随机的，因此可以认为在任何反射方向上的分布u都是一样的，但入射光线的角度很重要。

漫反射光照符合**兰伯特定律（Lambert）**：反射光线的强度与表面法线和光源之间的夹角的余弦值成正比，因此，漫反射的计算公式如下：

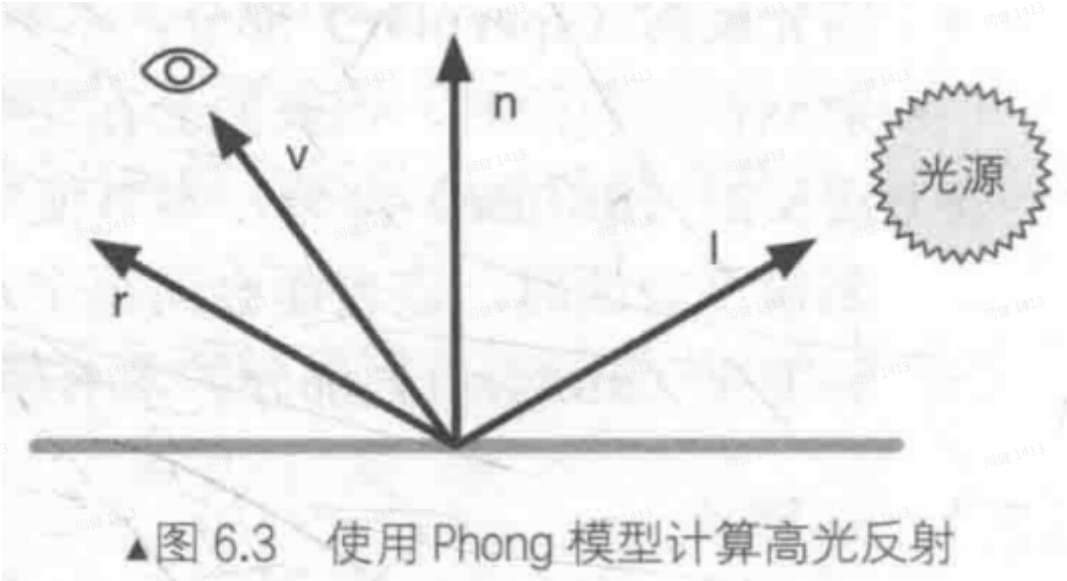
$$c_{diffuse}=(c_{light}\cdot m_{diffuse})\max(0,n\cdot l)$$

其中，n是表面法线，l是指向光源的单位向量，括号中的两个分别是光源颜色和材质的漫反射颜色。需要注意的是，我们要防止法线和光源方向点乘的结果为负值，为此，我们使用取最大值的函数来将其截取到0，这可以防止物体被从后面来的光源照亮。

高光反射（specular）

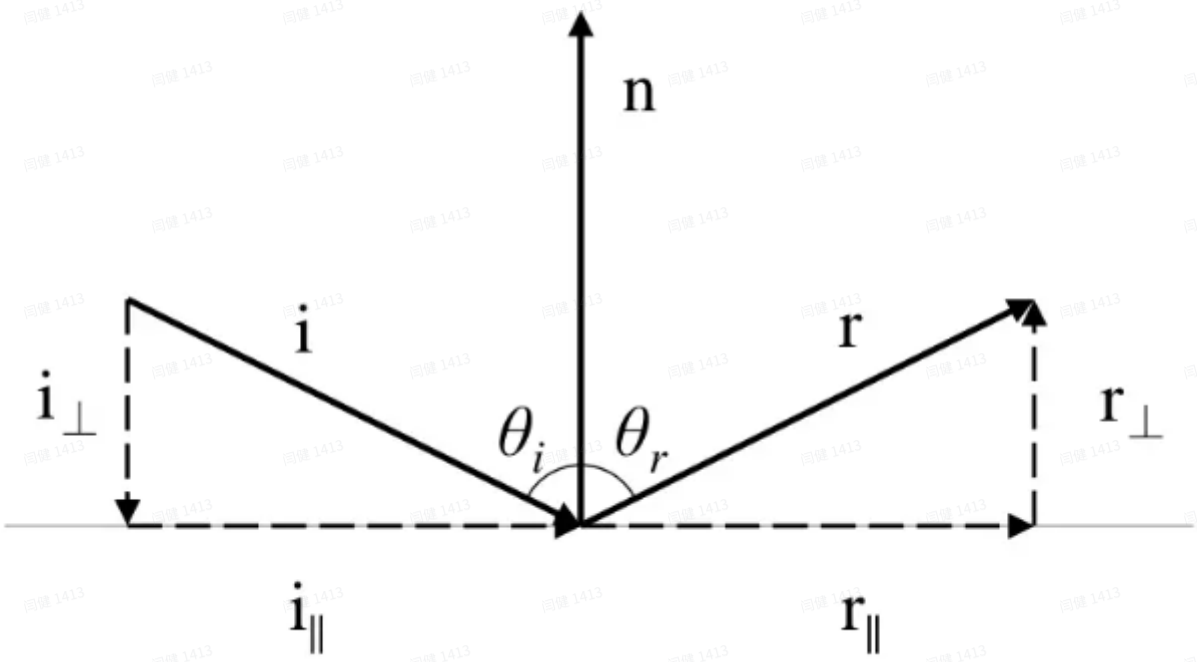
这里的高光反射是一种经验模型，也就是说它并不完全符合真实世界中的高光反射现象。它可用于计算那些沿着完全镜面反射方向被反射的光线，这可以让物体看起来是有光泽的，例如金属材质。

计算高光反射需要知道的信息比较多，比如表面法线、视角方向、光源方向、反射方向等。在本节中，我们假设这些向量都是单位向量：



▲图 6.3　使用 Phong 模型计算高光反射

在这四个向量中，我们实际上只需要知道其中三个向量即可，而第四个向量——反射方向，可以通过其他信息计算推导出来。



如图，根据反射定律，易知 `θi` = `θr` ，则有：

$$i_{\perp} = -|i_{\perp}|n = (i \cdot n)n$$
$$i_{||} = i - i_{\perp}$$

又知， `r∥` 和 `i∥` 是等向量， `r⊥` 是 `i⊥` 的反向量，则有：

$$r = r_{||} + r_{\perp} = i_{||} - i\perp = i - 2i_{\perp} = i - 2(i \cdot n)n$$

再带入我们之前看到的Phong模型中，-I是光线入射方向，N是法线向量，则反射方向为：

$$\mathbf{r} = 2\,(\hat{\mathbf{n}} \cdot \mathbf{I})\,\hat{\mathbf{n}} \quad -\mathbf{I}$$

这样，我们就可以用Phong模型来计算高光反射部分：

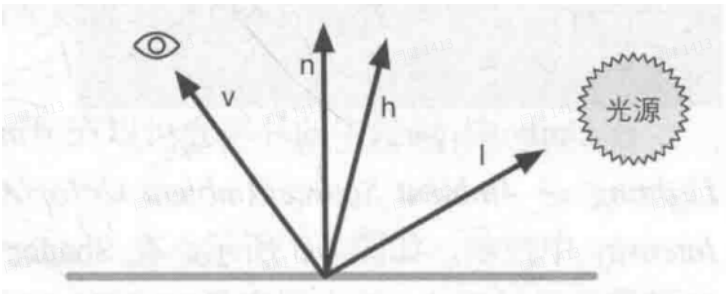
$$\mathbf{c}_{spscular} = (\mathbf{c}_{light} \cdot \boldsymbol{m}_{specular}) \max(0,\ \hat{\mathbf{v}} \cdot \mathbf{r})^{\,m_{gloss}}$$

Mgloss是材质的**光泽度（gloss）**，或称**反光度（shininess）**。它用于控制高光区域的"亮点"有多宽，Mgloss越大，亮点就越小。

Mspecular是材质的高光反射颜色，它用于控制该材质对于高光反射的强度和颜色。

Clight则是光源的颜色和强度。同样，这里也要防止v·r的结果为负数。

和上述的Phong相比，Blinn提出了一个简单的修改方法来得到类似的效果，它的基本思想是，避开计算反射方向r的步骤。为此Blinn模型引入了一个新的向量h，它是通过对v（人眼观察方向）和I（光照方向）去平均后在归一化得到的。



$$\hat{\mathbf{h}} = \frac{\hat{\mathbf{v}} + \mathbf{I}}{|\hat{\mathbf{v}} + \mathbf{I}|}$$

然后，使用n和h之间的夹角进行计算，而非v和r的夹角，总结下来，Blinn模型的公式如下：

$$\boldsymbol{c}_{specular} = (\mathbf{c}_{light} \cdot \boldsymbol{m}_{specular}) \max(0, \hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^{\,m_{gloss}}$$

在硬件实现时，如果摄像机和光源距离模型足够远的话，Blinn模型会快于Phong模型，这是因为，此时可以认为v和I都是定值，因此h将是一个常量。

但v或者l不是定值时，Phong模型可能更快一些。同时，再次强调，这两种光照模型都是经验模型，可以认为是数据堆砌出来的结果。实际上，在一些情况下，Blinn模型更符合实验结果。

逐像素或逐顶点

通常，在计算光照模型时，我们有两种选择：

1. 在片元着色器中计算，此时它被称作**逐像素光照(per-pexel lighting)**。
    a. 在逐像素光照中，我们会以每个像素为基础，得到它的法线（可以是对顶点法线插值得到的，也可以是从法线纹理中采样得到的），然后进行光照模型的计算。这种在面片之间对顶点法线进行插值的技术被称为**Phong着色（Phong shading）**，或称Phong插值或法线插值着色技术。它并不是我们之前说到的Phong模型。
2. 在顶点着色器中计算，此时它被称作**逐顶点光照(per-vertex lighting)**。
    a. 与之对应的逐顶点光照，也被称为**高洛德着色（Gouraud shading）**。在逐顶点广肇中，我们在每个顶点上计算光照，然后会在渲染图元内部进行线性差值，最后输出成像素颜色。
    b. 由于顶点数目往往远小于像素数目，因此逐顶点光照的计算量，往往要小于逐像素光照。但是，由于逐顶点光照依赖于线性插值来得到像素光照，因此，当光照模型中有非线性的计算（例如高光反射）时，逐顶点光照就会产生异常。

小结

虽然标准光照模型仅仅是一个经验模型，它不完全符合物理世界的真实现象，但它在易用性，速度和效果上都有较好的表现，因此仍然被广泛使用。基于这个模型是由裴祥风（Bui Tuong Phong）首先提出，并由Bling的方法进行了简化计算，我们把这种模型称为**Blinn-Phong光照模型**。

但这种模型有局限性，首先很多物理现象无法用Blinn-Phong模型来表现，例如**菲涅耳反射（Fresnel reflection）**。其次，Blinn-Phong模型是**各向同性（isotropic）**的，也就是说，当我们固定视角和光源方向旋转这个表面时，反射不会发生任何变化。但有些表面是具有**各向异性（anisotropic）**反射性质的，例如金属丝，毛发等，后续还会有更复杂的模型，更真实地反应光和物体的交互，但在这里我们先按下不表。

Unity中的环境光和自发光

在标准光照模型中，环境光和自发光的计算是很直接的。在Unity中，可以在Window→Lighting的Ambient Source中设置，Ambient Intensity就是环境光。在Shader中，我们只需要通过Unity的内置变量 `UNITY_LIGHTMODEL_AMBIENT` 就可以得到环境光的颜色和强度信息。

而大多数物体是没有自发光特性的，在计算自发光的时候，只需要在片元着色器输出最后的颜色之前，把材质的自发光颜色添加到输出颜色上即可。

Unity Shader中的漫反射光照模型

首先来看标准光照模型中的漫反射光照部分，之前我们的基本光照模型中的漫反射部分公式为：

$$c_{diffuse}=(c_{light} \cdot m_{diffuse})\max(0, \hat{n} \cdot I)$$

从公式中可以看出，要计算漫反射需要知道4个参数：入射光线的颜色和强度Clight，材质的漫反射系数Mdiffuse，表面法线n以及光源反向I。同时为了防止点积的结果为负值，我们需要使用max操作；而CG提供了这样的函数：

```
// 把参数v截取在[0,1]的范围内，如果v是一个向量，那么它会对向量的每一个分量进行这样的操作。
// v：float、float2、float3、float4...
// func:saturate(v)
```

逐顶点光照

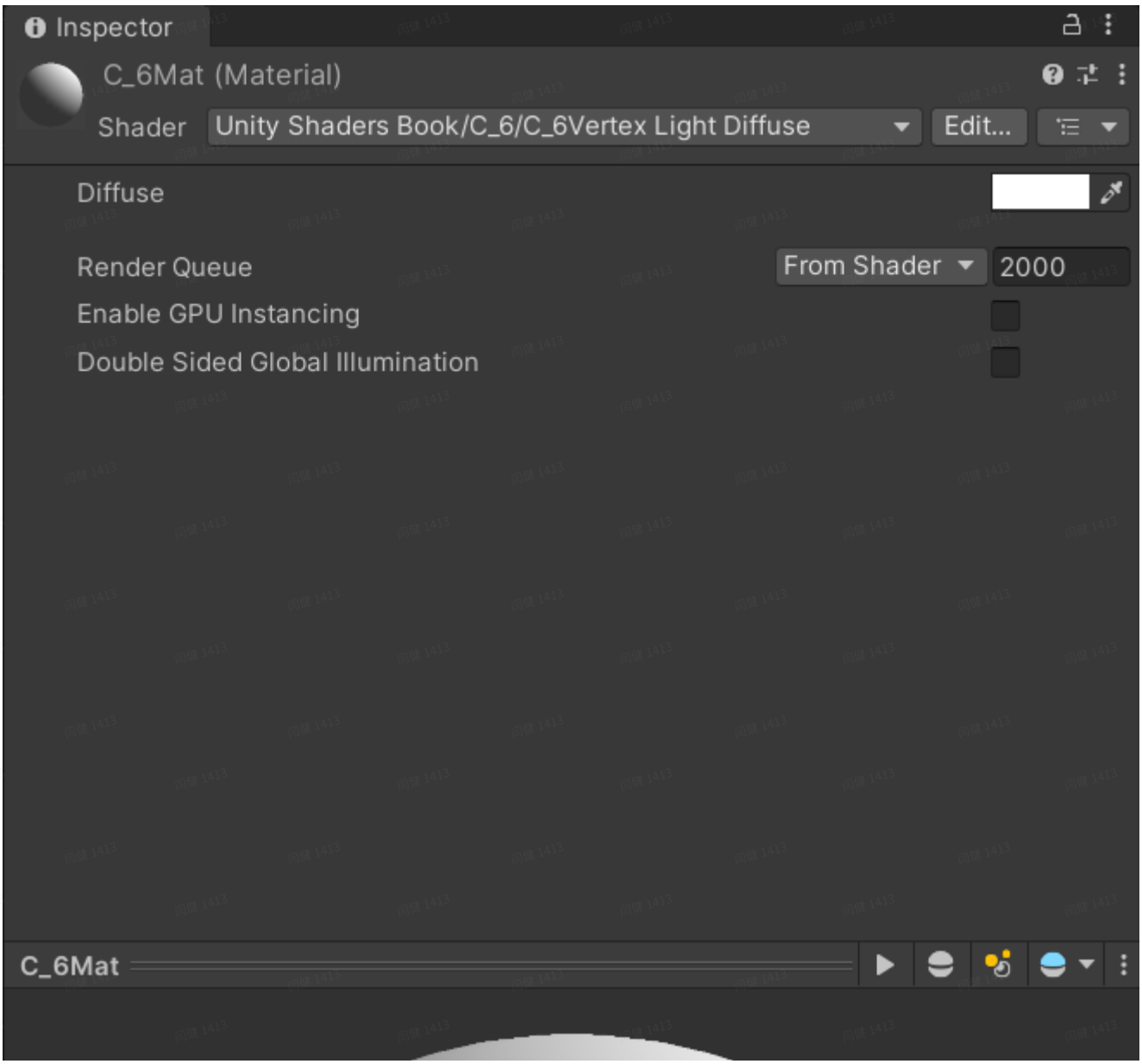
首先我们来实践高洛德着色，即逐顶点光照。

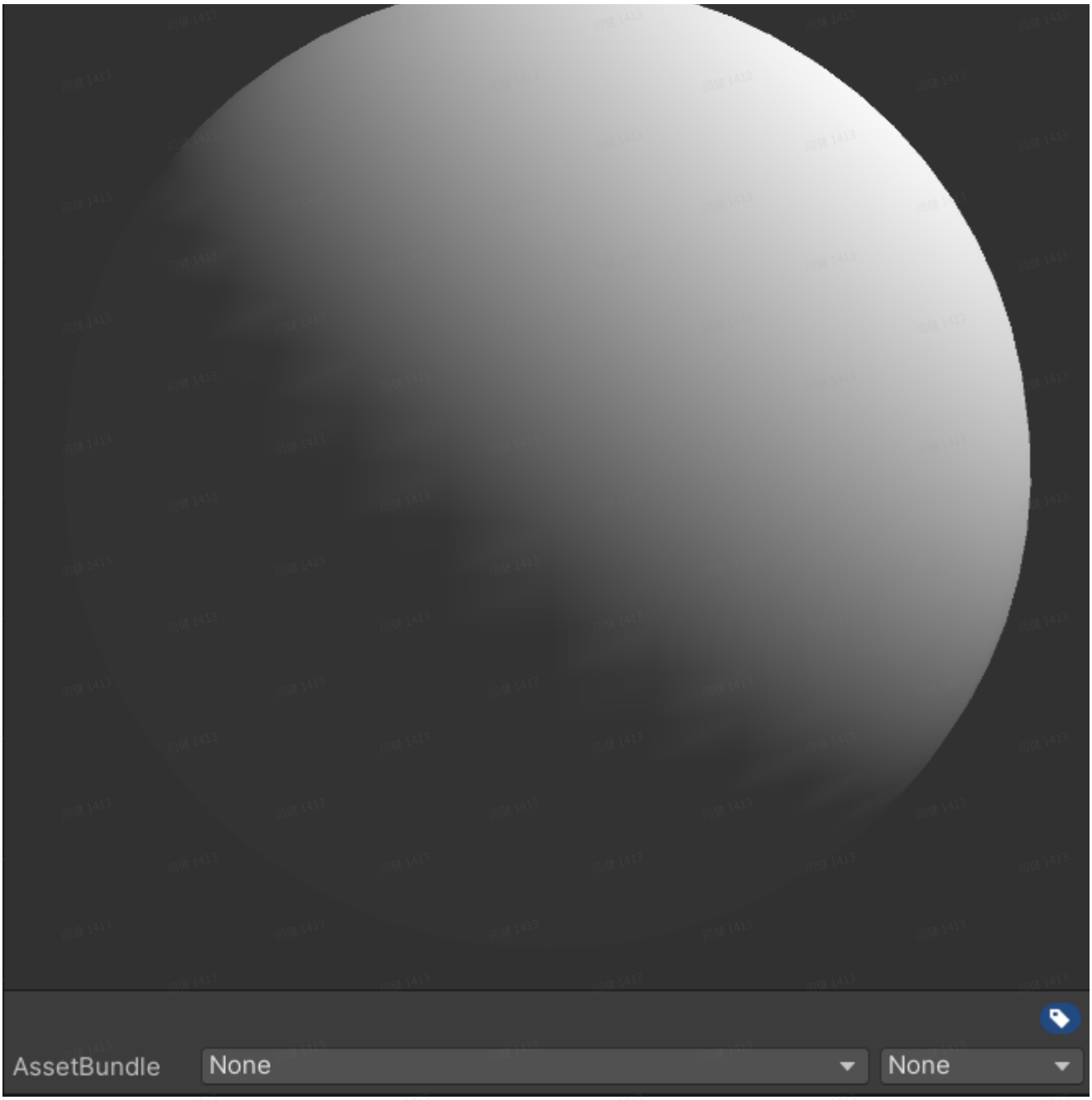
```
shader "Unity Shaders Book/C_6/C_6Vertex Light Diffuse"
{
    Properties
    {
        // 漫反射颜色的属性，默认为白色
        _Diffuse("Diffuse", Color) = (1, 1, 1, 1)
    }

    SubShader
    {
        Pass
        {
            // LightMode用于定义该Pass在Unity的光照流水线中的角色
            // 后续会详细地了解，这里有个概念即可
            // 只有定义了正确的LightMode，才能得到一些Unity的内置光照变量
            Tags { "LightMode" = "ForwardBase" }

            CGPROGRAM

            // 声明顶点/片元着色器的方法名
            #pragma vertex vert
            #pragma fragment frag

            // 为了使用Unity内置的一些变量，比如：_LightColor0
            #include "Lighting.cginc"

            // 声明属性变量
            fixed4 _Diffuse;
```

```
29
30              struct a2v
31              {
32                  // 顶点坐标
33                  float4 vertex : POSITION;
34                  // 顶点法线矢量
35                  float3 normal : NORMAL;
36              };
37
38              struct v2f
39              {
40                  float4 pos : SV_POSITION;
41                  // 顶点着色器输出的颜色值
42                  // 这里不是必须使用COLOR语义，也可以使用TEXCOORD0语义
43                  fixed3 color : COLOR;
44              };
45
46              v2f vert (a2v v)
47              {
48                  v2f o;
49                  // 将顶点坐标由模型空间转到裁剪空间
50                  o.pos = UnityObjectToClipPos(v.vertex);
51
52                  // 通过Unity内置变量得到环境光的颜色值
53                  fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
54
55                  // 将法线由模型空间转到世界空间，并进行归一化（法线的矩阵转换需要使用逆矩阵来实现）
56                  // 交换矢量和矩阵的相乘顺序来实现相同的效果
57                  // 因为法线是三维矢量，所以只需截取对应矩阵的前三行前三列即可
58                  fixed3 worldNormal = normalize(mul(v.normal, (float3x3)unity_WorldToObject));
59                  // 通过内置变量_WorldSpaceLightPos0获取世界空间下的光源方向，并进行归一化
60                  // 因为本案例中光源是平行光，所以可以直接取该变量进行归一化，若是其他类型光源，计算方式会有不同
61                  fixed3 worldLight = normalize(_WorldSpaceLightPos0.xyz);
62                  // 使用漫反射公式得到漫反射的颜色值
63                  // 通过内置变量_LightColor0，拿到光源的颜色信息
64                  // saturate，为取值范围限定[0, 1]的函数
65                  // dot，为矢量点积的函数，只有两个矢量处于同一坐标空间，点积才有意义
66                  fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * saturate(dot(worldNormal, worldLight));
67
68                  // 最后对环境光和漫反射光部分相加，得到最终的光照结果
69                  o.color = ambient + diffuse;
70
```

```
71                  return o;
72              }
73
74              fixed4 frag (v2f i) : SV_Target
75              {
76                  // 将顶点输出的颜色值作为片元着色器的颜色输出，输出到屏幕上
77                  return fixed4(i.color, 1.0);
78              }
79
80              ENDCG
81          }
82      }
83
84      // 使用内置的Diffuse作为保底着色器
85      Fallback "Diffuse"
86  }
```

最后效果如图：



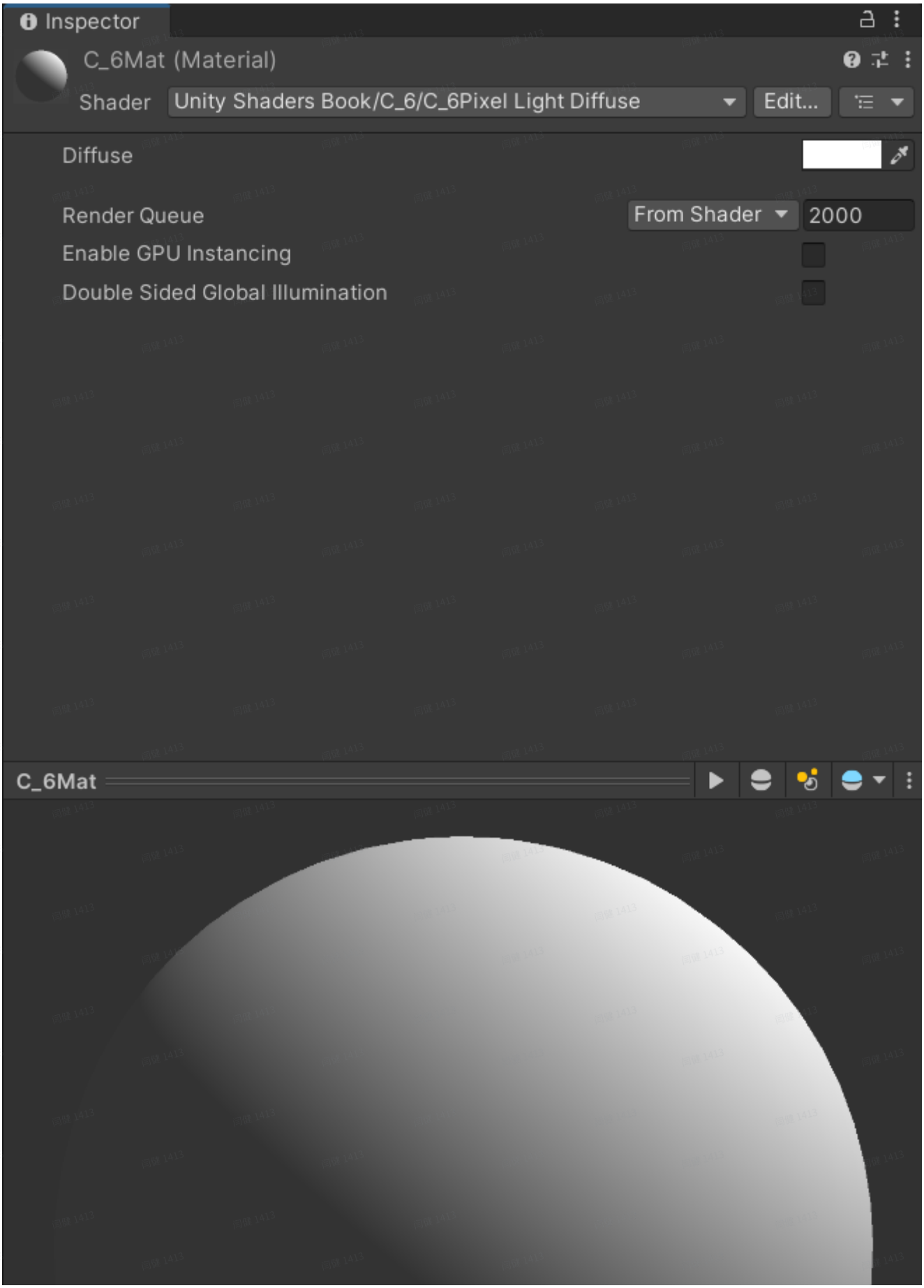


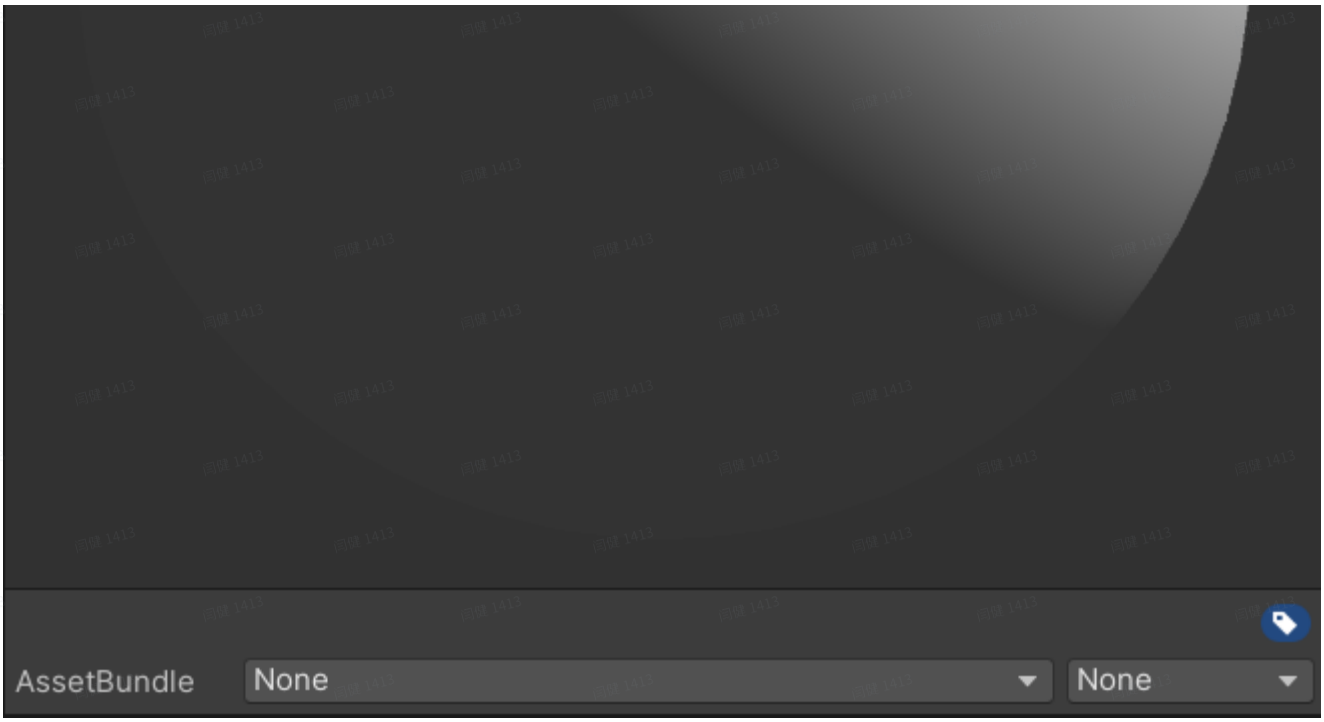
逐像素光照

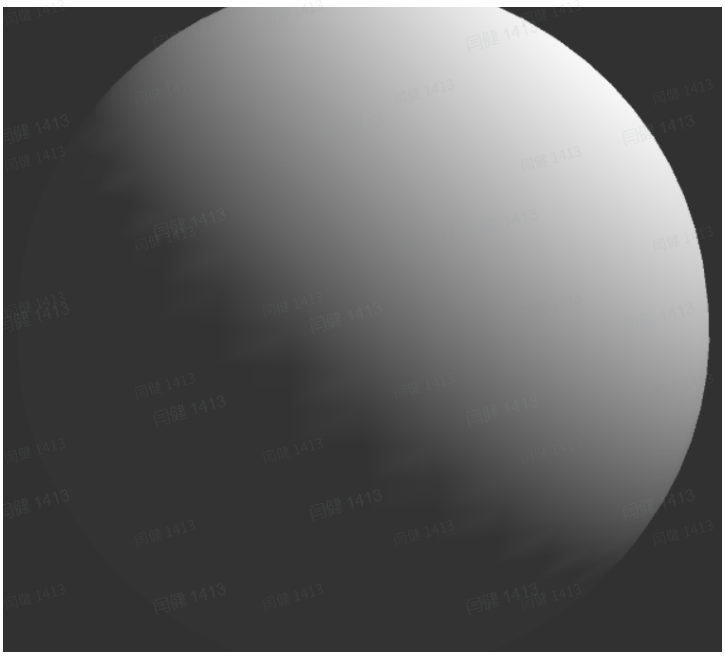
```
shader "Unity Shaders Book/C_6/C_6Pixel Light Diffuse"
{
Properties
    {
        _Diffuse("Diffuse", Color) = (1, 1, 1, 1)
    }

    SubShader
    {
        Pass
        {
            Tags { "LightMode" = "ForwardBase" }

```

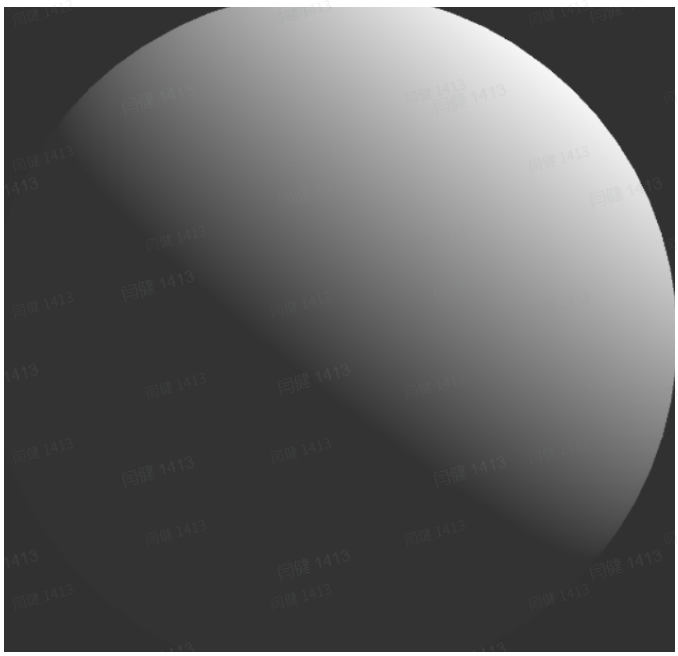
```
14              CGPROGRAM
15
16              #pragma vertex vert
17              #pragma fragment frag
18
19              #include "Lighting.cginc"
20
21              fixed4 _Diffuse;
22
23              struct a2v
24              {
25                  float4 vertex : POSITION;
26                  float3 normal : NORMAL;
27              };
28
29              struct v2f
30              {
31                  float4 pos : SV_POSITION;
32                  fixed3 worldNormal : TEXCOORD0;
33              };
34
35              v2f vert (a2v v)
36              {
37                  v2f o;
38                  //顶点着色器不需要计算光照，只需要把世界空间下的法线传递给片元着色器即可
39                  o.pos = UnityObjectToClipPos(v.vertex);
40                  o.worldNormal = normalize(mul(v.normal,
    (float3x3)unity_WorldToObject));
41                  return o;
42              }
43
44              fixed4 frag (v2f i) : SV_Target
45              {
46                  //片元反射器需要计算漫反射光照模型:
47                  fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
48
49                  fixed3 worldNormal = normalize(i.worldNormal);
50                  fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
51
52                  fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
    saturate(dot(worldNormal, worldLightDir));
53                  fixed3 color = ambient + diffuse;
54                  return fixed4(color, 1.0);
55              }
56
57              ENDCG
58          }
```

```
59      }
60
61      Fallback "Diffuse"
62  }
```









对比一下两种阴影，你可以明显发现逐像素光照可以得到更平滑的效果。但是可以看到，在光照无法到达的区域，魔性的外观通常是黑的，没有任何明暗变化，这回事的魔性的背光区域看起来像是一个平面一样。

为此，有一种改善技术被提了出来，这就是**半兰伯特（Half Lambert）光照模型**。

半兰伯特模型

之前我们使用的光照模型又称为兰伯特光照模型，因为它符合兰伯特定律——在平面某点漫反射光的光前与该反射点的法向量和入射光角度的余弦值成正比。为了应对上一小节提出的问题，Valve公司在开发《半条命》时提出了一种技术，由于该技术是在原兰伯特光照模型的基础上进行了修改，因此被称为半兰伯特光照模型。

广义的半兰伯特光照模型的公式如下：

$$\mathbf{c}_{diffuse}=(\mathbf{c}_{light}\cdot\mathbf{m}_{diffuse})(\alpha(\hat{\mathbf{n}}\cdot\mathbf{I})+\beta)$$

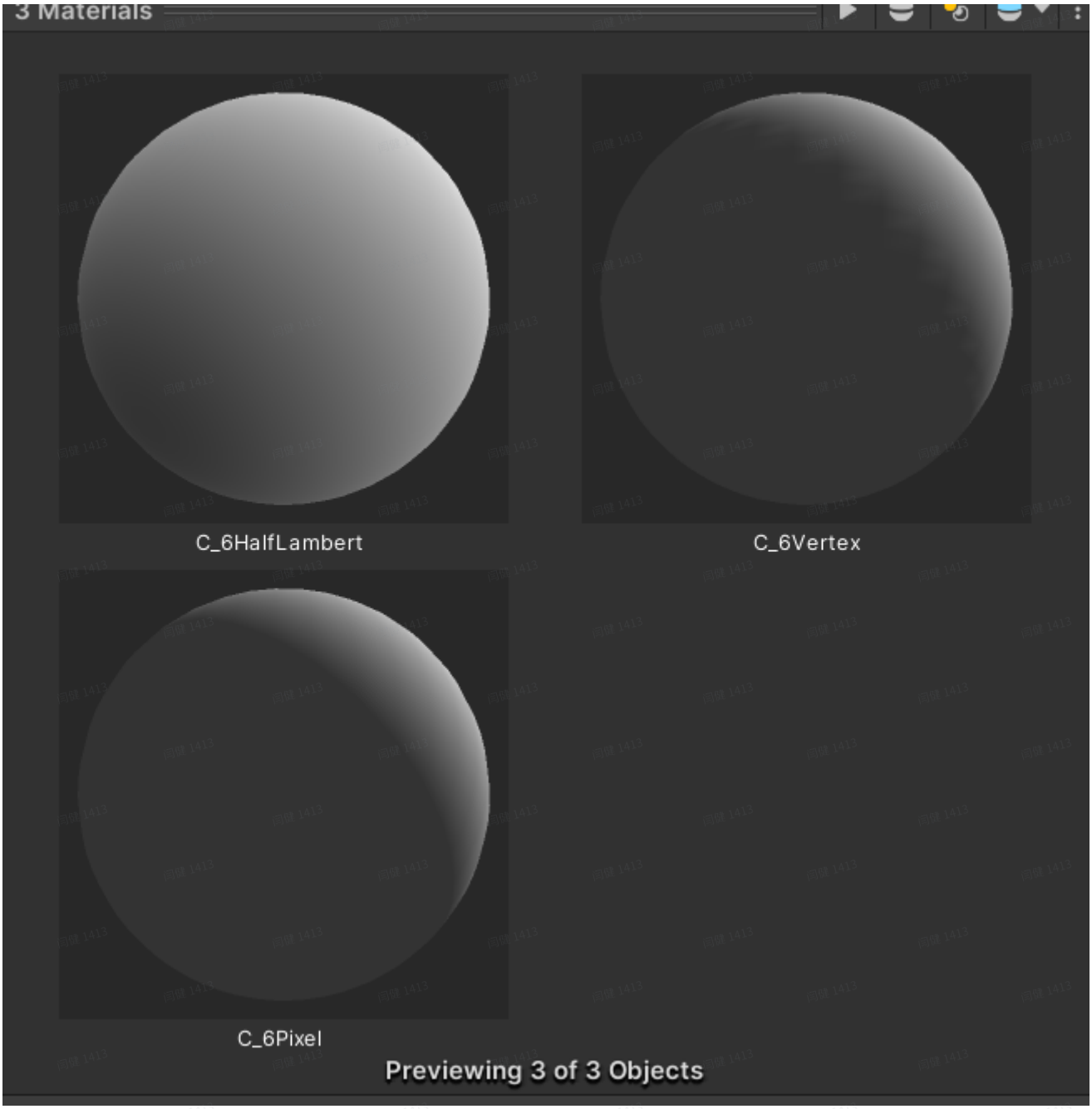
可以看到，与原版相比，半兰伯特光照模型没有使用max操作来防止n和l的点积为负数，而是对其结果进行了一个α倍的缩放再加上一个β大小的偏移。绝大多数情况下，α和β的值均为0.5，即公式为：

$$\mathbf{c}_{diffuse}=(\mathbf{c}_{light}\cdot\mathbf{m}_{diffuse})(0.5(\hat{\mathbf{n}}\cdot\mathbf{I})+0.5)$$

通过这样的方式，我们把n·l的结果范围从[-1,1]映射到了[0,1]范围内。即之前的0值处，现在也可以有明暗变化。但半兰伯特光照模型是没有任何物理依据的，它仅仅是一个视觉加强技术。

```
fixed4 frag (v2f i) : SV_Target
{
    //片元反射器需要计算漫反射光照模型:
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
    //应用半兰伯特光照公式
    fixed halfLambert = dot(worldNormal, worldLightDir) * 0.5 + 0.5;

    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * halfLambert;
    fixed3 color = ambient + diffuse;
    return fixed4(color, 1.0);
}
```

相较于逐像素光照，半兰伯特模型的区别仅仅在于公式的细微调整，可以在图片中感受三种方式对光照处理的细节变化：



Unity Shader中的高光反射光照模型

之前，高光反射部分的计算公式为：

$$\mathbf{c}_{specular}=(\mathbf{c}_{light}\cdot\mathbf{m}_{specular})\max(0,\hat{\mathbf{v}}\cdot\mathbf{r})^{m_{gloss}}$$

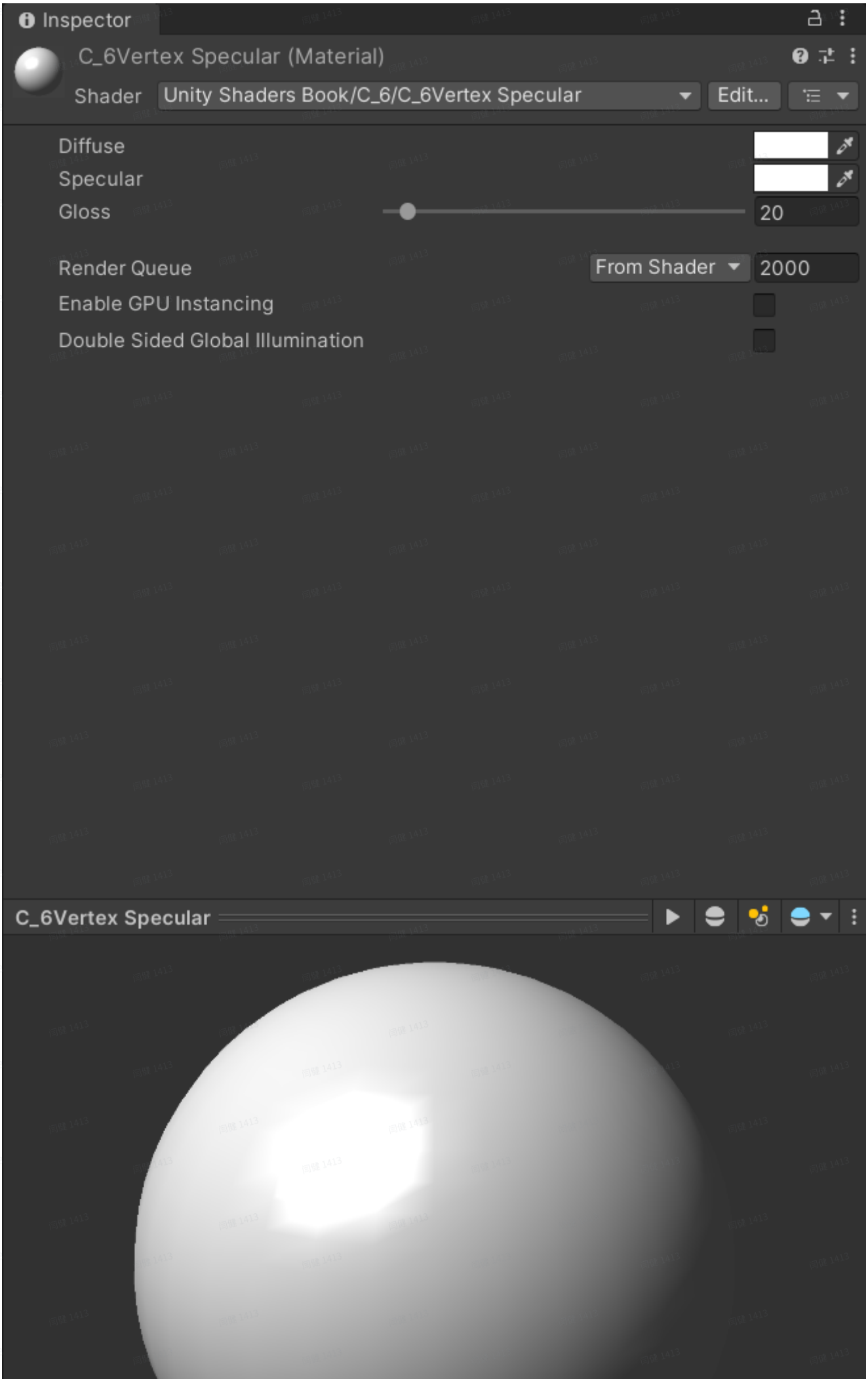
可以看出，要计算高光反射需要4个参数，入射光线的颜色和强度Clight，材质的高光反射系数Mspecular，视角方向v以及反射方向r。我们也知道反射方向r可以由表面法线n和光源方向I计算而得：r = 2(n·I)n - I；而CG提供了计算反射方向的函数reflect。

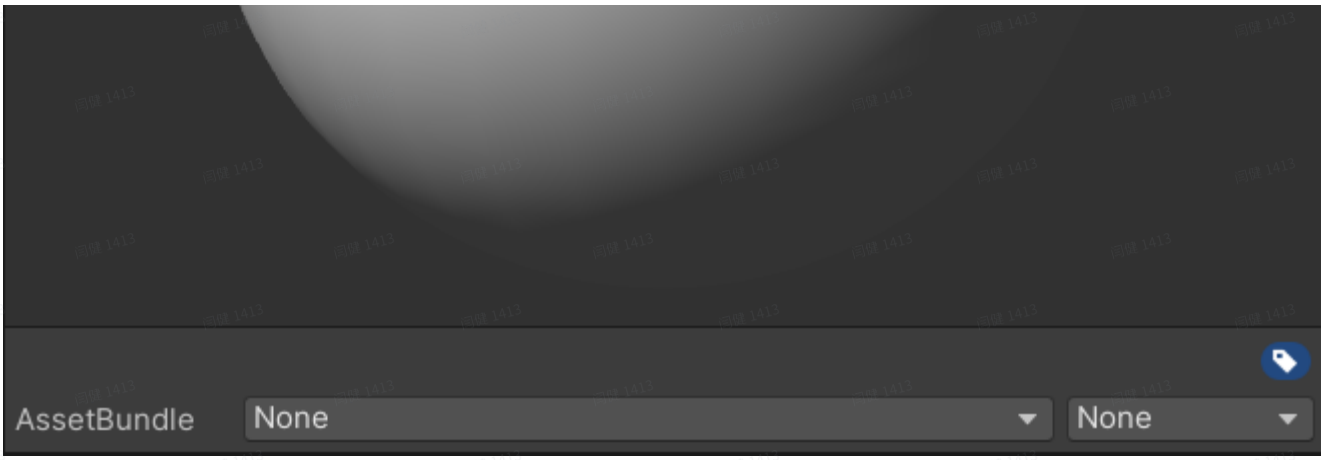
```
1 //当给定入射方向和法线方向n时，reflect可以返回反射方向。
2 //参数：i，入射方向；n，法线方向。可以是float、float2、float3
3 //函数：reflect(i, n)
```

逐顶点光照

```
shader "Unity Shaders Book/C_6/C_6Vertex Specular"
{
    Properties
    {
        _Diffuse ("Diffuse", Color) = (1, 1, 1, 1)
        // 高光反射叠加的颜色，默认为白色
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        // 光泽度，控制高光区域的大小
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }

    SubShader
    {
        Pass
        {
            // 为了正确地得到一些Unity的内置光照变量，如_LightColor0
            Tags { "LightMode" = "ForwardBase" }

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            // 为了使用Unity内置的一些变量，如:_LightColor0
            #include "Lighting.cginc"

            /* 声明属性变量 */
            // 颜色值的范围在0~1，故使用fixed精度
            fixed4 _Diffuse;
            fixed4 _Specular;
            // 光泽度数值范围较大，使用float精度
            float _Gloss;

            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                // 顶点着色器输出颜色值到片元着色器
                fixed3 color : COLOR;
```

```
45              };
46
47              v2f vert (a2v v)
48              {
49                  v2f o;
50                  o.pos = UnityObjectToClipPos(v.vertex);
51
52                  fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
53
54                  fixed3 worldNormal = normalize(mul(v.normal,
    (float3x3)unity_WorldToObject));
55                  fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
56
57                  // 使用兰伯特漫反射模型
58                  fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
    saturate(dot(worldNormal, worldLightDir));
59                  // 使用reflect函数求出入射光线关于表面法线的反射方向，并进行归一化
60                  // 因为reflect的入射方向要求由光源指向焦点处（worldLightDir是焦点处指
    向光源），所以需要取反
61                  fixed3 reflectDir = normalize(reflect(-worldLightDir,
    worldNormal));
62                  // 通过Unity内置变量_WorldSpaceCameraPos得到世界空间中的相机位置
63                  // 通过与世界空间中的顶点坐标进行相减，得到世界空间下的视角方向
64                  fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz -
    mul(unity_ObjectToWorld, v.vertex).xyz);
65
66                  // 根据高光反射公式求出高光反射颜色
67                  fixed3 specular = _LightColor0.rgb * _Specular.rgb *
    pow(saturate(dot(reflectDir, viewDir)), _Gloss);
68
69                  // 环境光+漫反射+高光反射
70                  o.color = ambient + diffuse + specular;
71                  return o;
72              }
73
74              fixed4 frag (v2f i) : SV_Target
75              {
76                  return fixed4(i.color, 1.0);
77              }
78              ENDCG
79          }
80      }
81
82      // 使用Unity内置的Specular Shader兜底
83      Fallback "Specular"
84  }
```


Inspector
C_6Vertex Specular (Material)
Shader
Unity Shaders Book/C_6/C_6Vertex Specular
Edit...
Diffuse
Specular
Gloss
20
Render Queue
From Shader
2000
Enable GPU Instancing
Double Sided Global Illumination
C_6Vertex Specular



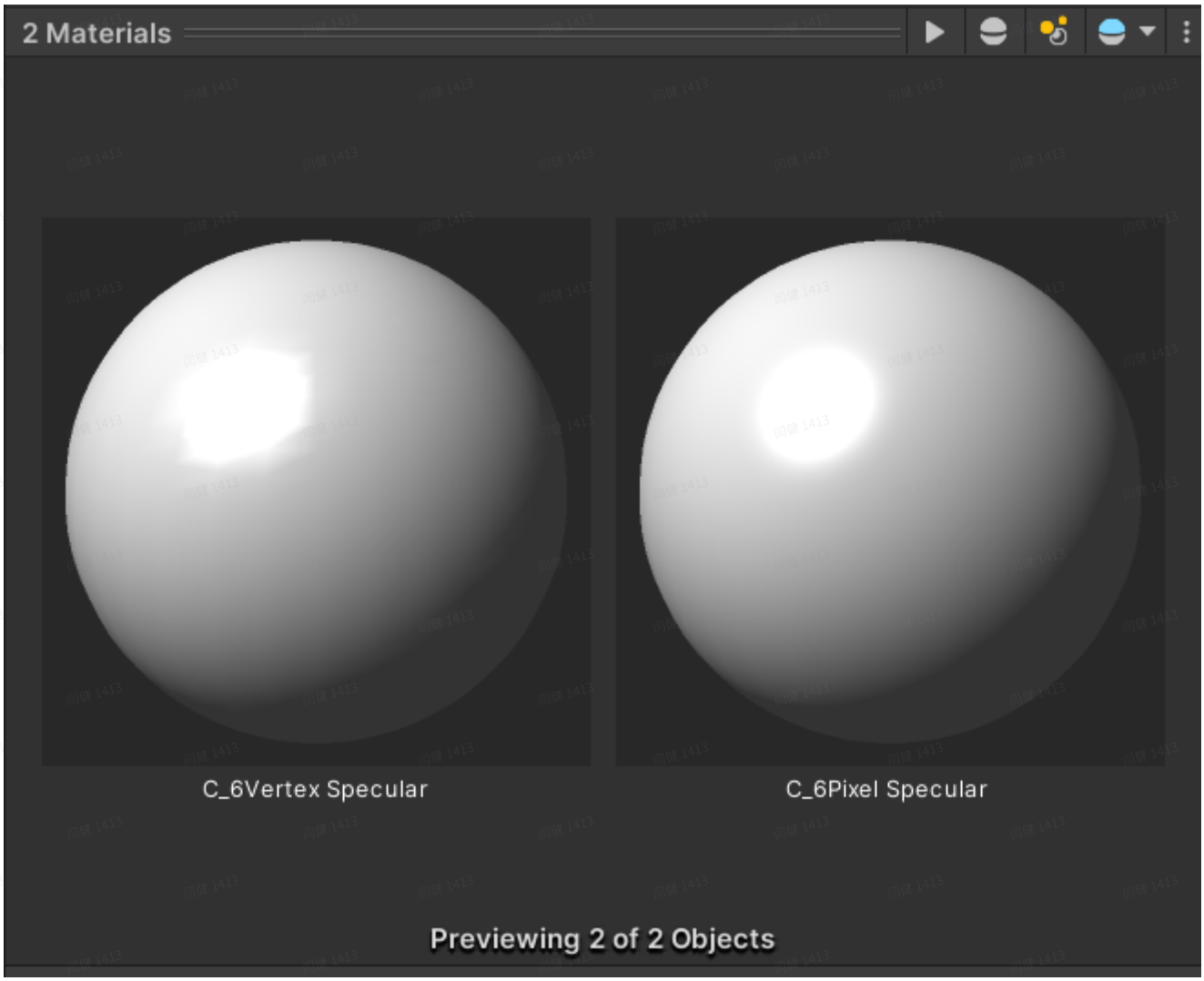
逐像素光照

同上，我们只需要把计算光照部分从顶点着色器挪到片元着色器即可，顶点着色器只传递法线和空间转换：

```
shader "Unity Shaders Book/C_6/C_6Pixel Specular"
{
    Properties
    {
        _Diffuse ("Diffuse", Color) = (1, 1, 1, 1)
        // 高光反射叠加的颜色，默认为白色
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        // 光泽度，控制高光区域的大小
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }

    SubShader
    {
        Pass
        {
            // 为了正确地得到一些Unity的内置光照变量，如_LightColor0
            Tags { "LightMode" = "ForwardBase" }

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            // 为了使用Unity内置的一些变量，如:_LightColor0
            #include "Lighting.cginc"

            /* 声明属性变量 */
            // 颜色值的范围在0~1，故使用fixed精度
            fixed4 _Diffuse;
            fixed4 _Specular;
```

```
31              // 光泽度数值范围较大，使用float精度
32              float _Gloss;
33
34              struct a2v
35              {
36                  float4 vertex : POSITION;
37                  float3 normal : NORMAL;
38              };
39
40              struct v2f
41              {
42                  float4 pos : SV_POSITION;
43                  // 顶点着色器输出颜色值到片元着色器
44                  float3 worldNormal : TEXCOORD0;
45                  float3 worldPos : TEXCOORD1;
46              };
47
48              v2f vert (a2v v)
49              {
50                  v2f o;
51                  o.pos = UnityObjectToClipPos(v.vertex);
52
53                  o.worldNormal = mul(v.normal, (float3x3)unity_WorldToObject);
54                  o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
55                  return o;
56              }
57
58              fixed4 frag (v2f i) : SV_Target
59              {
60                  fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
61                  fixed3 worldNormal = normalize(i.worldNormal);
62                  fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
63
64                  fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
    saturate(dot(worldNormal, worldLightDir));
65                  fixed3 reflectDir = normalize(reflect(-worldLightDir,
    worldNormal));
66                  fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz -
    i.worldPos.xyz);
67                  fixed3 specular = _LightColor0.rgb * _Specular.rgb *
    pow(saturate(dot(reflectDir, viewDir)), _Gloss);
68                  return fixed4(ambient + diffuse + specular, 1.0);
69              }
70              ENDCG
71          }
72      }
73
```

```
74      // 使用Unity内置的Specular Shader兜底
75      Fallback "Specular"
76  }
```



此时已经可以初步对比出顶点着色器和片元着色器在平滑度上的区别；

Blinn-Phong光照模型

Blinn模型没有使用反射方向，而是引入了一个新的向量h，它是通过观察方向v和光照方向l相加后再归一化得到的，它的公式和Blinn模型计算高光反射的公式如下：

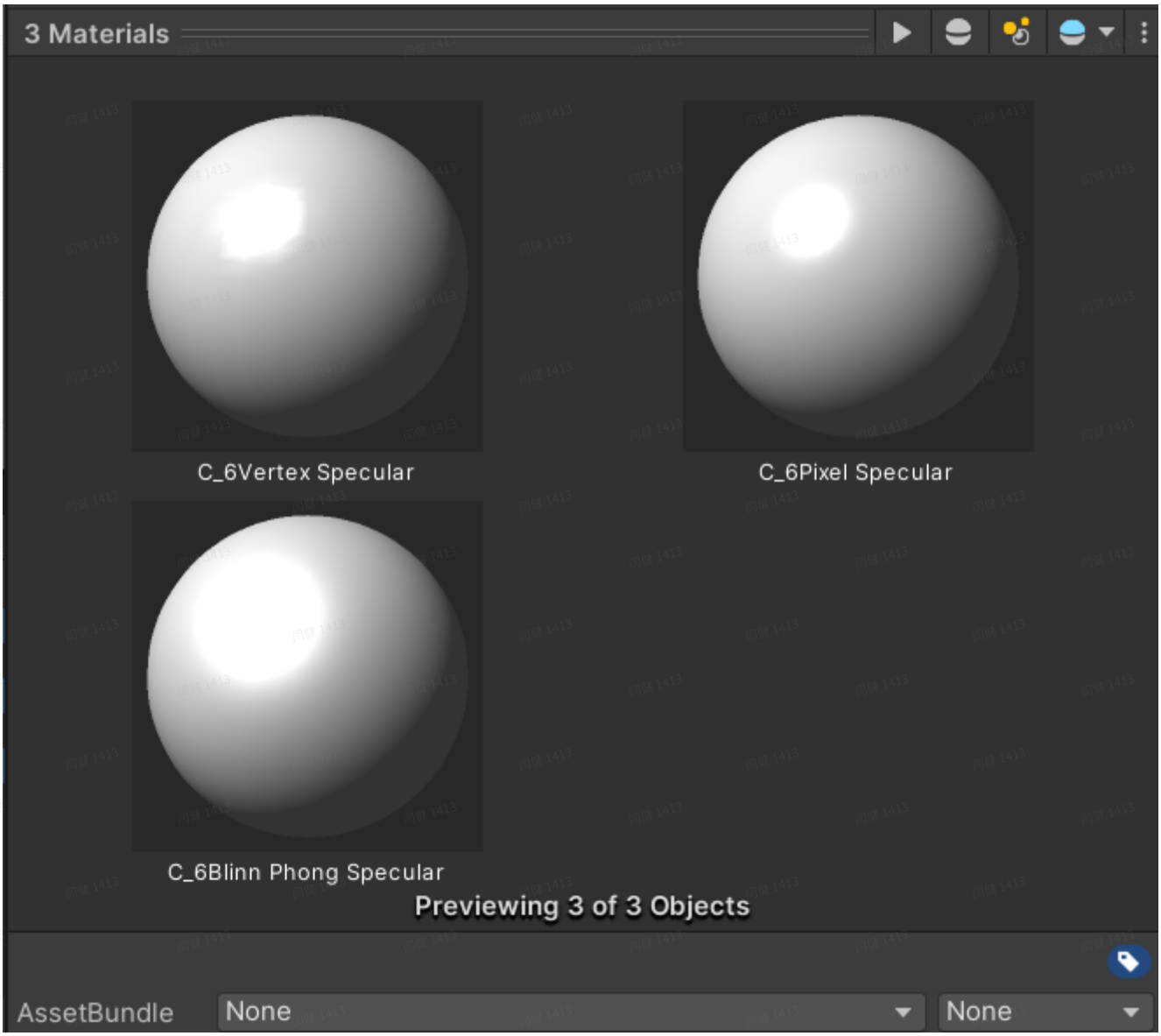
$$\hat{\mathbf{h}}=\frac{\hat{\mathbf{v}}+\hat{\mathbf{l}}}{|\hat{\mathbf{v}}+\hat{\mathbf{l}}|}$$

$$c_{specular}=(c_{light}\cdot\mathbf{m}_{specular})\max(0,\ \hat{\mathbf{n}}\cdot\hat{\mathbf{h}})^{m_{gloss}}$$

它只有片元着色器中计算部分有更改：

```
1  fixed4 frag (v2f i) : SV_Target
2  {
3      fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
4      fixed3 worldNormal = normalize(i.worldNormal);
```

```
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);

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

    // fixed3 reflectDir = normalize(reflect(-worldLightDir, worldNormal));
    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);
    //h
    fixed3 halfDir = normalize(worldLightDir + viewDir);

    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);
    return fixed4(ambient + diffuse + specular, 1.0);
}
```



你可以看出，Blinn-Phong光照模型的高光反射看起来更大更亮一些。

使用Unity内置函数

上面的着色器中，我们往往需要得到光源方向、视角方向这两个基本信息，例如使用 `normalize(_WorldSpaceLightPos0.xyz)` 来得到光源方向(这种方式只适用于平行光)，或使用 `normalize(_WorldSpaceCameraPos.xyz - i.worldPosition.xyz)` 来得到视角方向。但比如耦合了点光源和聚光灯，我们计算光源的光源方向就是错误的。

表 6.1 UnityCG.cginc 中一些常用的帮助函数

函 数 名	描 述
float3 WorldSpaceViewDir (float4 v)	输入一个模型空间中的顶点位置，返回世界空间中从该点到摄像机的观察方向。内部实现使用了 UnityWorldSpaceViewDir 函数
float3 UnityWorldSpaceViewDir (float4 v)	输入一个世界空间中的顶点位置，返回世界空间中从该点到摄像机的观察方向
float3 ObjSpaceViewDir (float4 v)	输入一个模型空间中的顶点位置，返回模型空间中从该点到摄像机的观察方向
float3 WorldSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个模型空间中的顶点位置，返回世界空间中从该点到光源的光照方向。内部实现使用了 UnityWorldSpaceLightDir 函数。没有被归一化
float3 UnityWorldSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个世界空间中的顶点位置，返回世界空间中从该点到光源的光照方向。没有被归一化
float3 ObjSpaceLightDir (float4 v)	仅可用于前向渲染中。输入一个模型空间中的顶点位置，返回模型空间中从该点到光源的光照方向。没有被归一化
float3 UnityObjectToWorldNormal (float3 norm)	把法线方向从模型空间转换到世界空间中
float3 UnityObjectToWorldDir (in float3 dir)	把方向矢量从模型空间变换到世界空间中
float3 UnityWorldToObjectDir(float3 dir)	把方向矢量从世界空间变换到模型空间中

上图中的某些函数，在如今的版本下（2022.3.15或更高）已经被命名为别的函数或者被删除，具体函数可以参照UnityCG内置着色器源码阅读那篇。

部分函数，在内部没有保证得到的方向是单位向量，我们使用前要把他们归一化。如果我们使用上面的函数，以Blinn-Phong光照模型为例，代码中，顶点着色器和片元着色器的部分可以改写为：

```
v2f vert (a2v v)
{
    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);

    // o.worldNormal = mul(v.normal, (float3x3)unity_WorldToObject);
    o.worldNormal = UnityObjectToWorldNormal(v.normal);
    o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
    fixed3 worldNormal = normalize(i.worldNormal);
```

```
16      // fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
17      fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));
18
19      fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
    saturate(dot(worldNormal, worldLightDir));
20
21      // fixed3 reflectDir = normalize(reflect(-worldLightDir, worldNormal));
22      // fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);
23      fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
24      //h
25      fixed3 halfDir = normalize(worldLightDir + viewDir);
26
27      fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
    dot(worldNormal, halfDir)), _Gloss);
28      return fixed4(ambient + diffuse + specular, 1.0);
29  }
```

再次说明的是，内置函数得到的方向是没有归一化的，因此我们需要使用normalize函数来对结果进行归一化。