Lecture6 Unity 中的基础光照

- 3D Math Primer for Graphics and Game Development

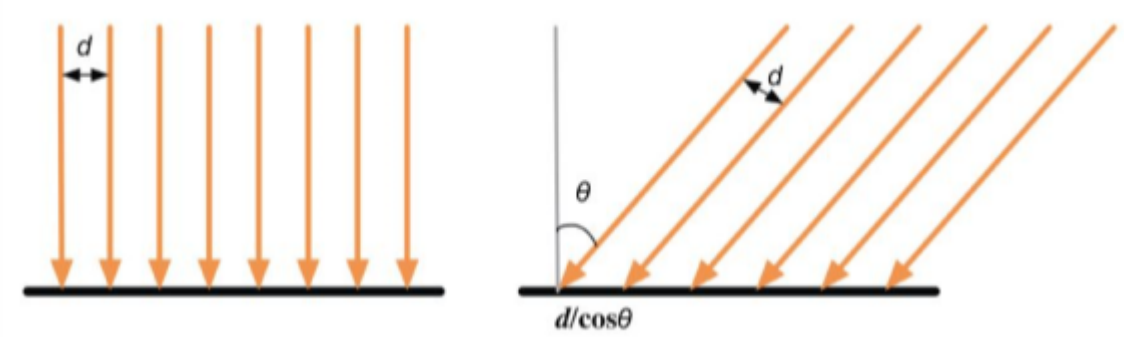
1. 我们是如何看到这个世界的

我们要模拟从真实光照环境中生成图像，需要考虑 3 种物理现象

- 光线从光源中被发射出来
- 光线和场景中的一些物体相交，一些光线被物体吸收了，另一些光线被散射到其他方向
- 摄像机吸收了一些光，产生了一张图形

平行光源

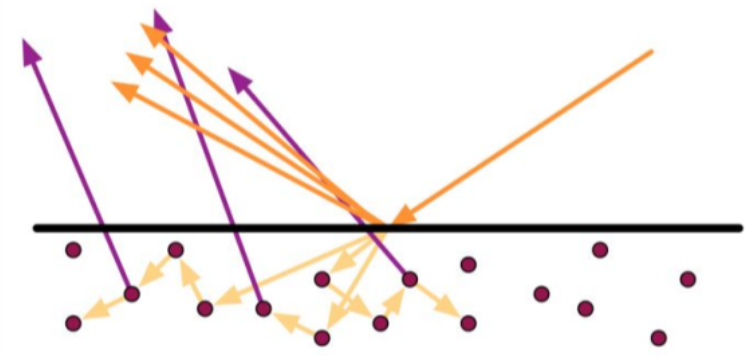
- 在实时渲染中，我们通常把光源当成一个**没有体积的点**，用 l 表示它的**方向**
- 测量一个光源发射了多少光，可以用**辐照度 irradiance** 来量化光
  - 对于平行光来说，计算垂直于 l 的单位面积上单位时间内穿过的能量
  - 通常，物体表面与 l 不是垂直的，我们可以使用光源方向 l 和平面法线 n 之间的夹角的余弦值来得到
- 辐照度与 $\frac{d}{cos\theta}$ 成**反比**
  - d：照射到物体表面的光线距离
  - $cos\theta$：光源方向 l 和平面法线 n 的夹角余弦



吸收与散射

光线从光源发射出来后，就会与一些物体相交，通常相交的结果有

- **吸收 absorption**
  - 改变光线的密度和颜色，但不改变光线方向
- **散射 scattering**
  - 改变光线方向，不改变光线的密度和颜色
  - 散射到物体内部，通常被称为**折射 refraction** 或**透射 transmission**
  - 散射到外部，通常被称为**反射 reflection**
  - 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，另一些责备物体吸收



- 我们在光照模型中使用不同的部分计算折射和反射
  - **高光反射 specular**：表示物体表面是如何反射光线的
  - **漫反射 diffuse**：表示有多少光线会被折射、吸收和散射出表面
- 我们可以计算出射光线的数量和方向，通常使用**出射度 exitance** 来描述它，辐照度和出射度之间满足线性关系

着色 Shading

- 根据材质属性、光源信息等，使用一个等式去计算某个观察方向的出射度的过程，我们把这个等式成为**光照模型 Lighting Model**

BRDF 光照模型

> 计算机图形学第一定律：如果它看起来是对的，那么它就是对的

- 当给定模型表面的一个点时，BRDF (Bidirectional Reflectance Distribution Function) 包含了该点外观的完整描述
  - BRDF 大多使用一个数学公式，并提供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 可以给出某个出射方向的光照能力分布
- BRDF 模型大部分都是**经验模型**

2. 标准光照模型 Blinn-Phong 模型

> 1975 年，著名学者 Bui Tuong Phong 提出标准光照模型

标准光照模型**只关心直接光照 direct light**，也就是那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，主要分成四个部分

名称	符号	描述
自发光 emissive	$c_{emissive}$	给定一个方向时，一个表面自身本身会向该方向发射多少辐射量 如果没有使用**全局光照 global illumination** 技术，这些自发光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
高光反射 specular	$c_{specular}$	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发射方向散射多少辐射量
漫反射 diffuse	$c_{diffuse}$	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
环境光 ambient	$c_{ambient}$	描述所有其它间接光照

环境光 ambient

> **间接光照 indirect light** 光线通常会在多个物体之间发射，最后进入摄像机

在标准光照模型中，使用一种被称为环境光的部分来模拟间接光照，环境光的计算非常简单，它通常是一个**全局变量**，即场景中的所有物体都使用这个环境光

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

自发光 emissive

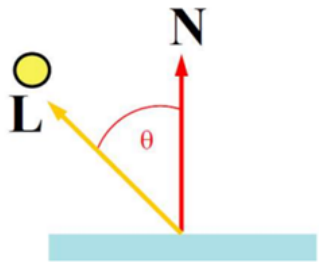
光线可以**直接由光源发射进摄像机**，不需要经过物体的反射，标准光照模型使用自发光来描述这部分，它直接使用材质的自发光颜色

$$\boldsymbol{c}_{emissive} = \boldsymbol{m}_{emissive}$$

在实时渲染中，**自发光的表面往往不会照亮周围的表面**，不过 Unity 引入的全局光照系统则可以模拟这类自发光物体对周围环境的影响

漫反射 diffuse

用于被物体表面随机散射到各个方向的辐射度进行建模



$$\boldsymbol{c}_{diffuse} = (\boldsymbol{c}_{light} \cdot \boldsymbol{m}_{diffuse}) \max(0, \boldsymbol{n} \cdot \boldsymbol{l})$$

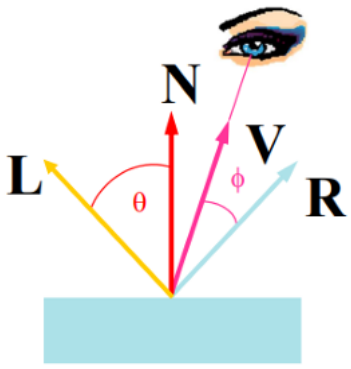
- $\boldsymbol{n}$：表面法线的单位向量
- $\boldsymbol{l}$：指向光源的单位向量
- $\boldsymbol{m}_{diffuse}$：材质的漫反射颜色
- $\boldsymbol{c}_{light}$：光源颜色

为了防止法线和光源的点乘结果为负数，使用 $\max$ 函数截断到 0，可以防止物体被从背后来的光源照亮

高光反射 specular

Phong 模型

高光反射是一种经验模型，用于计算沿着完全镜面发射方向被反射的光线，使得物体看起来是有光泽的



- $\boldsymbol{n}$：表面法线方向
- $\boldsymbol{l}$：指向光源的方向
- $\boldsymbol{v}$：指向视角方向
- $\boldsymbol{r}$：反射方向

在这四个向量中，只需要知道前三个，反射方向可以通过其他信息计算得到

$$\boldsymbol{r} = 2(\boldsymbol{n} \cdot \boldsymbol{l}) \cdot \boldsymbol{n} - \boldsymbol{l}$$

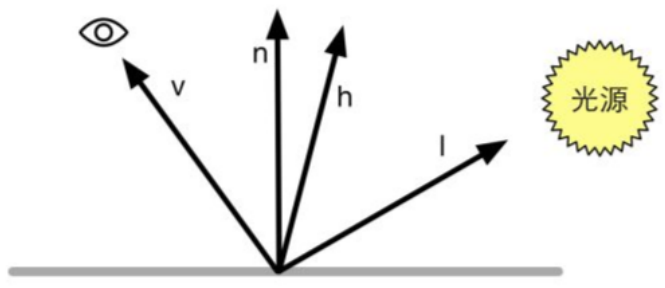
这样，我们可以用 Phong 模型来计算高光反射

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

- m_{gloss}：**光泽度 gloss/反光度 shininess**，控制高光区域亮点有多宽
- $\boldsymbol{m}_{specular}$：材质的高光反射颜色
- $\boldsymbol{c}_{light}$：光源的颜色和强度

Blinn 模型

Blinn 提出了一个简单的修改方法来得到类似的效果
为了避免计算反射方向 $\boldsymbol{r}$，引入了一个新的向量 $\boldsymbol{h}$



$$\boldsymbol{h} = \frac{\boldsymbol{v} + \boldsymbol{l}}{|\boldsymbol{v} + \boldsymbol{l}|}$$

然后，使用 $\boldsymbol{n}$ 和 $\boldsymbol{h}$ 之间的夹角来进行计算，而不是 $\boldsymbol{v}$ 和 $\boldsymbol{r}$

着色

逐像素光照 per-pixel lighting / Phone shading

- 在片元着色器中计算
- 以每个像素为基础，得到它的法线，然后进行光照模型的计算

逐顶点光照 per-vertex lighting / Gouraud shading

- 在顶点着色器中计算
- 在每个顶点上计算光照，然后在渲染图元内部进行线性插值，最终输出像素颜色
- 顶点数目远远小于像素数目，因此逐顶点光照的计算量往往小于逐像素光照

小结

标准光照模型是一种经验模型，它不完全符合真实世界的光照现象，但是它易用且计算速度效果较好，被更广泛使用。

它被称为 Phong 光照模型，而后，由于 Blinn 的方法简化了计算且在某些情况下计算更快，于是这种模型也被称为 Blinn-Phong 光照模型

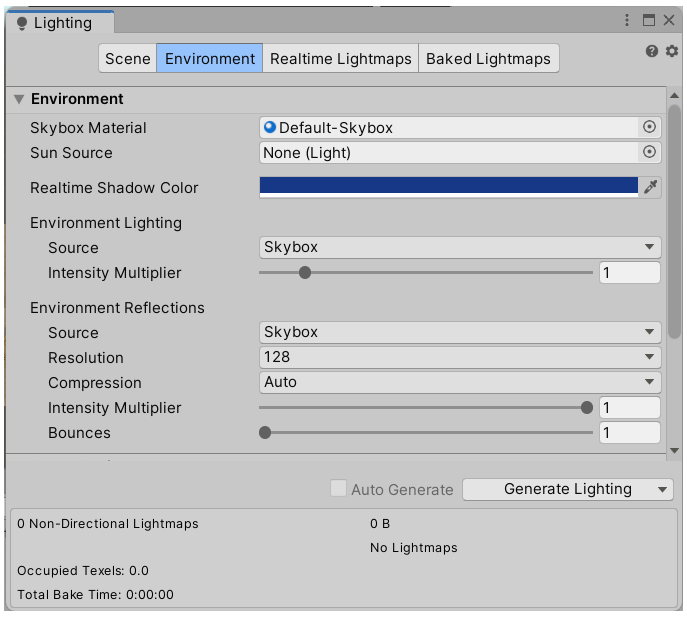
局限性

有些重要的物理现象无法用过 Blinn-Phong 模型表现出来

- 菲涅尔反射 Fersnel reflection
- Blinn-Phong 模型是各向同性 isotropic 的，当我们固定视角和光源旋转这个表面时，反射不会发生任何比那花，但是有的表面是各向异性的 anisotropic，如拉丝金属、毛发等

3. Unity 中的环境光和自发光

- Unity 场景中的环境光可以在 Window -> Rendering -> Lighting -> Environment -> Environment Lighting 中控制



- Unity 大多数物体是没有自发光特性的，如果要计算自发光，只需要在片元着色器输出最后颜色之前，把材质的自发光颜色添加到输出颜色上即可

4. 在 Unity Shader 中实现漫反射光照模型

- 漫反射光照模型也被称为**兰伯特光照模型**

漫反射（顶点着色器操作）

$$\boldsymbol{c}_{diffuse} = (\boldsymbol{c}_{light} \cdot \boldsymbol{m}_{diffuse}) \max(0, \boldsymbol{n} \cdot \boldsymbol{l})$$


```
// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'

Shader "Custom//Assignment/Chapter6/1_DiffuseVertexLevel"
{
    // 控制材质的漫反射颜色
    // 在Properties语义块中声明了一个Color类型的属性，它的初始值设置为白色
    Properties
    {
        _Diffuse("Diffuse", Color) = (1, 1, 1, 1)
```

```
}
SubShader
{
    // 定义Pass语义块
    // 顶点/片元着色器的代码需要写在Pass而不是SubShader语义块中
    Pass
    {
        // 指明Pass的光照模式
        Tags
        {
            // LightMode标签是Pass标签中的一种，用于定义该Pass在光照流水线的角色
            // 只有定义正确的Pass，我们才能得到一些Unity内置的光照变量
            "LightMode"="ForwardBase"
        }

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        // 为了使用Unity内置的变量，还需要包含内置文件
        #include "Lighting.cginc"

        // 在Shader中使用Properties块中声明的属性，定义一个与该属性类型相匹配的变量
        // 颜色属性范围在0-1，我们可以用fixed精度变量存储
        fixed4 _Diffuse;

        // 顶点着色器输入结构体
        struct a2v
        {
            float4 vertex : POSITION; // 模型空间顶点坐标
            float3 normal : NORMAL; // 模型空间顶点法向量
        };

        // 顶点着色器输出结构体
        struct v2f
        {
            float4 pos: SV_POSITION; // 裁剪空间的顶点坐标
            fixed3 color : COLOR; // 存储颜色信息，并不一定要用COLOR语义，也可以用TECOORD0
        };

        // 顶点着色器的基本任务是把顶点位置从模型空间转换到裁剪空间中
        v2f vert(a2v v)
        {
            // 定义返回值o
            v2f o;

            // 完成坐标从模型空间->裁剪空间
            o.pos = UnityObjectToClipPos(v.vertex);

            // 通过Unity内置变量获得环境光部分
            fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

            // 将法线转换到世界空间中，可以使用顶点变换矩阵的逆转置矩阵对法线进行相同的变换
            // 先得到模型空间->世界空间的变换矩阵逆矩阵_World2Object，然后调换它在mul函数中的位置，得到和专转置矩阵相同的矩阵乘法
            // 只需要截取_World2Object的前三行和前三列
            fixed3 world_normal = normalize(mul(v.normal, (float3x3)unity_WorldToObject));

            // 光源方向可以由_WorldSpaceLightPos0得到
            // [注意]这里对光源方向的计算并不具有通用性，在本节中我们假设只有一个光源，且光源类型为平行光
            // 如果有多个光源且类型可能是点光源等，则不能用_WorldSpaceLightPos0得到正确的结果
            fixed3 world_light = normalize(_WorldSpaceLightPos0.xyz);
```

```
                // Unity提供内置变量_LightColor0来访问该Pass处理的光源颜色和强度信息
                // 计算漫反射光照部分，只有法线和光源方向在同一个坐标空间下，它们的点积才有意义，这里我们选择了世界坐标空间
                // 法线和光源方向都需要归一化
                fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * saturate(dot(world_normal, world_light));

                // 环境光+漫反射光 得到最终光照效果
                o.color = ambient + diffuse;

                return o;
            }

            // 片元着色器只需要输出顶点颜色即可
            fixed4 frag(v2f i) : SV_Target
            {
                return fixed4(i.color, 1.0);
            }
            ENDCG
        }

    }
    FallBack "Diffuse"
}
```

satuate(x) 函数

- 参数：用于操作的标量/向量，可以是 float、float2、float3 等类型
- 描述：把 x 截取在 [0, 1] 范围内，如果 x 是一个向量，那么会对它的每一个分量这样操作

漫反射（片元着色器操作）

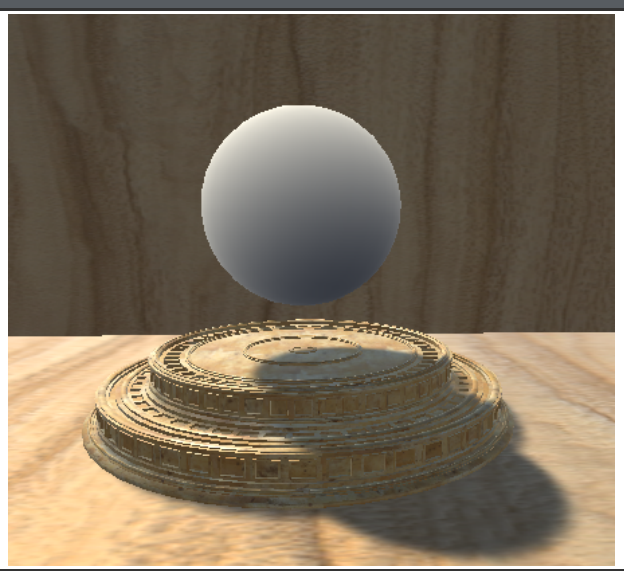

```
// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'

Shader "Custom//Assignment/Chapter6/2_DiffusePixelLevel"
{
    Properties
    {
        _Diffuse("Diffuse", Color) = (1, 1, 1, 1)

    }
    SubShader
    {
        Pass
        {
            // set the light mode
            Tags
            {
                "LightMode"="ForwardBase"
            }
```

```
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "Lighting.cginc"

        // use the property from the properties block,
        // range [0,1], we can use fixed 4 to store it
        fixed4 _Diffuse;

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

        struct v2f
        {
            float4 pos: SV_POSITION;
            fixed3 world_normal : TEXCOORD0;
        };

        // 顶点着色器不需要计算光照模型，只需要把世界空间下的法线传递给片元着色器即可
        v2f vert(a2v v)
        {
            v2f o;
            // transform the vertex object space -> projection space
            o.pos = UnityObjectToClipPos(v.vertex);

            // transform the normal from object space -> world space
            o.world_normal = mul(v.normal, (float3x3)unity_WorldToObject);
            return o;
        }

        fixed4 frag(v2f i) : SV_Target
        {
            // get ambient term
            fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

            // get normal vector in world space
            fixed3 world_normal = normalize(i.world_normal);

            // get light direction in world space
            fixed3 world_light_dir = normalize(_WorldSpaceLightPos0.xyz);

            // compute diffuse term
            fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
saturate(dot(world_normal, world_light_dir));

            fixed3 color = ambient + diffuse;
            return fixed4(color, 1.0);
        }
        ENDCG
    }

    }
    FallBack "Diffuse"
}
```

- 逐像素光照可以得到更平滑的光照效果

半兰伯特模型 Half Lambert

- 漫反射模型背面外观通常是全黑的，没有任何明暗变化，整个背光区域看起来就像平面一样，失去了模型的细节表现

正面	背面
	

广义半兰伯特光照模型定义如下

$$\boldsymbol{c}_{diffuse} = (\boldsymbol{c}_{light} \cdot \boldsymbol{m}_{diffuse})(\alpha(\boldsymbol{n} \cdot \boldsymbol{l}) + \beta)$$

- 大多数情况下，α 和 β 的值均为 0.5，即公式为

$$\boldsymbol{c}_{diffuse} = (\boldsymbol{c}_{light} \cdot \boldsymbol{m}_{diffuse})(0.5(\boldsymbol{n} \cdot \boldsymbol{l}) + 0.5)$$

- 通过这样将 $\boldsymbol{n} \cdot \boldsymbol{l}$ 从 [-1, 1] 映射到 [0, 1] 范围内，**对于模型的背光面，也会有明暗变化**，不同的点积结果会映射到不同的值上

5. 在 Unity Shader 中实现高光反射光照模型

reflect(i, n) 函数

- Unity 提供了计算反射方向的函数 reflect
- 参数：i 入射方向，n 法线方向，可以是 float、float2、float3 等类型
- 昂给你改了入射方向和法线方时，reflect 函数嗯可以返回反射方向

高光反射（顶点着色器操作）

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

```
// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'

Shader "Custom/Assignment/Chapter6/4_SpecularVertexLevel"
{
    Properties
    {
        _Diffuse("Diffuse", Color) = (1, 1, 1, 1)
        _Specular("Specular", Color) = (1, 1, 1, 1)
        _Gloss("Gloss", Range(8.0, 256)) = 20

    }
    SubShader
    {
        Pass
        {
            // set the light mode
            Tags
            {
                "LightMode"="ForwardBase"
            }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
```

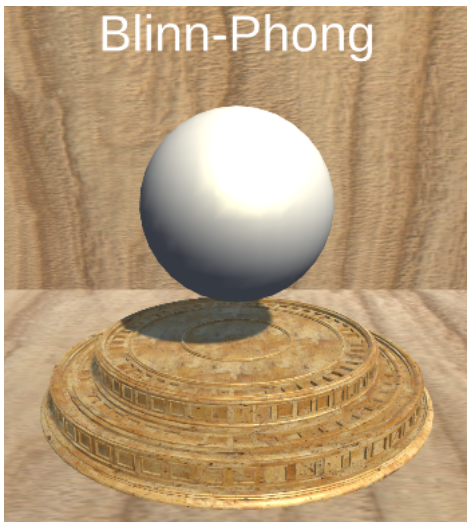
```
            #include "Lighting.cginc"

            // use the property from the properties block,
            // range [0,1], we can use fixed 4 to store it
            fixed4 _Diffuse;
            fixed4 _Specular;
            float _Gloss;

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

            struct v2f
            {
                float4 pos: SV_POSITION;
                fixed3 color : COLOR;
            };

            v2f vert(a2v v)
            {
                v2f o;
                // transform the vertex object space -> projection space
                o.pos = UnityObjectToClipPos(v.vertex);

                // get ambient term
                fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

                // transform the normal from object space -> world space
                fixed3 world_normal = normalize(mul(v.normal,
(float3x3)unity_WorldToObject));

                // get the light direction in world space
                fixed3 world_light_dir = normalize(_WorldSpaceLightPos0.xyz);

                // compute diffuse term
                fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
saturate(dot(world_normal, world_light_dir));

                // get the reflect direction in world space
                fixed3 reflect_dir = normalize(reflect(world_light_dir, world_normal));

                // get the view direction in world space
                fixed3 view_dir = normalize( mul(unity_ObjectToWorld, v.vertex).xyz -
_WorldSpaceCameraPos.xyz);

                // compute specular term
                fixed3 specular = _LightColor0.rgb * _Specular.rgb *
pow(saturate(dot(reflect_dir, view_dir)), _Gloss);

                o.color = ambient + diffuse + specular;

                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                return fixed4(i.color, 1.0);
            }
            ENDCG
        }
```

```
    }
    FallBack "Diffuse"
}
```

- Gloss 越小，高光越大

高光反射（片元着色器操作）

Blinn-Phone 光照模型



我们之前还提到了另一种高光反射的实现方法，Blinn 光照模型，它引用了一个新的向量 $\boldsymbol{h}$

$$\boldsymbol{h}=\frac{\boldsymbol{v}+\boldsymbol{l}}{|\boldsymbol{v}+\boldsymbol{l}|}$$

```
// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'

Shader "Custom/Assignment/Chapter6/5_Blinn"
{
    Properties
    {
        _Diffuse("Diffuse", Color) = (1, 1, 1, 1)
        _Specular("Specular", Color) = (1, 1, 1, 1)
        _Gloss("Gloss", Range(8.0, 256)) = 20

    }
    SubShader
    {
        Pass
        {
            // set the light mode
            Tags
            {
                "LightMode"="ForwardBase"
            }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "Lighting.cginc"

            // use the property from the properties block,
            // range [0,1], we can use fixed 4 to store it
            fixed4 _Diffuse;
            fixed4 _Specular;
            float _Gloss;

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

            struct v2f
```

```
            {
                float4 pos: SV_POSITION;
                fixed3 color : COLOR;
            };

            v2f vert(a2v v)
            {
                v2f o;
                // transform the vertex object space -> projection space
                o.pos = UnityObjectToClipPos(v.vertex);

                // get ambient term
                fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

                // transform the normal from object space -> world space
                fixed3 world_normal = normalize(mul(v.normal,
(float3x3)unity_WorldToObject));

                // get the light direction in world space
                fixed3 world_light_dir = normalize(_WorldSpaceLightPos0.xyz);

                // compute diffuse term
                fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb *
saturate(dot(world_normal, world_light_dir));

                // get the view direction in world space
                fixed3 view_dir = normalize(_WorldSpaceCameraPos.xyz -
mul(unity_ObjectToWorld, v.vertex).xyz);

                // get the half direction in world space
                fixed3 half_dir = normalize(view_dir + world_light_dir);

                // compute specular term
                fixed3 specular = _LightColor0.rgb * _Specular.rgb *
                    pow(
                        saturate(
                            max(
                                0, dot(half_dir, world_normal)
                            )
                        ), _Gloss
                    );

                o.color = ambient + diffuse + specular;

                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                return fixed4(i.color, 1.0);
            }
            ENDCG
        }

    }
    FallBack "Diffuse"
}
```

6. 使用 Unity 内置函数

在上面的例子中，我们计算光照模型的时候都是自己计算光源方向和视角方向的，在处理更加复杂的光照类型（如点光源和聚光灯），我们计算光源方向的方法是错误的，这需要在代码中先判断光源类型，再计算光源信息

手动计算这些光源信息相对比较麻烦，Unity 提供了一些内置函数帮我们计算这个信息

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**
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)	**仅用于前向渲染** 输入一个**模型空间中的顶点位置**，返回**世界空间中该点到光源的光照方向**，没有被归一化
float3 ObjSpaceLightDir(float4 v)	**仅用于前向渲染** 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**从模型空间中转换到世界空间中
float3 UnityObjectToWorlir(in float3 dir)	把**方向向量**从模型空间变换到世界空间中
float3 UnityWorlToObjectir(float3 idr)	把**方向向量**从世界空间变换到模型空间中

- 这些帮助函数使得我们不需要跟各种变换矩阵、内置变量打交道，也不需要考虑不同的情况，仅仅需要调用一个函数就可以得到需要的信息
- 由内置函数得到的方向都是**没有归一化**的，所以我们需要使用 **normalize 函数**对结果进行归一化，再进行光照模型的计算