

Lecture9 更复杂的光照

1. Unity 的渲染路径

- Unity 的**渲染路径 rendering path** 决定了光照是如何应用到 Unity Shader 中的
- 与光源交互，需要为每个 Pass 指定渲染路径，此时 Shader 的光照计算才能被正确执行

Unity 支持的渲染路径主要有三种

- **前向渲染路径 Forward Rendering Path**
- **延迟渲染路径 Deferred Rendering Path**
- 顶点照明渲染路径 Vertex Lit Rendering Path（弃用）

大多数情况下，一个项目只使用一种渲染路径，默认情况下是使用前向渲染路径。但是有时候可以在其他组件上（比如摄像机）选择覆盖 Project Setting 中设置的渲染路径

如果当前显卡不支持所选择的渲染路径，Unity 会自动使用更低一级的渲染路径

我们会在每个 Pass 中使用标签来指定该 Pass 的渲染路径，这是通过 **LightMode 标签**实现的

LightMode 支持的渲染路径设置选项

标签名	描述
Always	不管使用哪种渲染路径，该 Pass 总是会被渲染，但不会计算任何光照
ForwardBase	前向渲染，该 Pass 会计算环境光、重要的平行光，逐顶点/SH 光源和 Lightmaps
ForwardAdd	前向渲染，该 Pass 会计算额外的逐像素光照，每个 Pass 对应一个光源
Deferred	延迟渲染，该 Pass 会渲染 G 缓冲
ShadowCaster	把物体的深度信息渲染到阴影映射纹理 shadowmap 或一张深度纹理中
PrepassBase	遗留的延迟渲染，该 Pass 会渲染发现和高光反射的指数部分
PrebassFinal	遗留的延迟渲染，该 Pass 会通过合并纹理、光照和自发光来渲染得到最后颜色
Vertex/VertexLMRGBM/VertexLM	遗留的顶点照明渲染

设置渲染路径相当让 Unity 底层引擎为我们**准备好所需的光照属性**，然后通过内置光照变量来访问这些属性

如果我们没有指定渲染路径，那么一些光照变量可能不会被正确赋值，我们计算出来的效果也就可能是错误

前向渲染路径

传统且最常用渲染方式

原理

每进行一次完整的前向渲染，我们需要渲染该对象的渲染图元，并计算两个缓冲区的信息

- **深度缓冲区**：决定一个片元是否可见，如果可见就更新颜色缓冲区的颜色值
- **颜色缓冲区**：最终使用颜色缓冲区的值渲染画面

伪代码

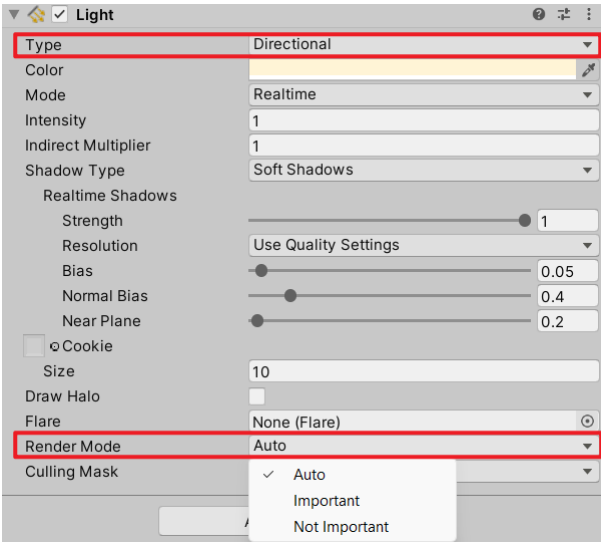
```
Pass{
    for(each primitive in this model){
        for(each fragment covere by this primitive){
            if(faile in depth test){
                discard;
            }else{
                float4 color = Shaing(materialInfo, pos, normal, lightDir, viewDir);
                // 更新帧缓冲
                writeFrameBuffer(fragment, color);
            }
        }
    }
}
```

- 对于每个**逐像素光源**，我们都需要进行该渲染流程
- 如果一个物体在**多个逐像素光源**的影响区域内，则需要**执行多个 Pass**，每个 Pass 计算一个逐像素光源的光照结果，然后再帧缓冲中把这些光照结果混合得到最终的颜色值

- 假设场景中有 N 个物体，每个物体受到 M 个光源的影响，则需要渲染整个场景 $N \times M$ 个 Pass
- 如果有大量逐像素光照，需要执行的 Pass 数目会很大，因此渲染引擎通常限制每个物体的逐像素光照数目

Unity 中前向渲染

- Unity 前向渲染路径有 3 种处理光照（照亮物体） 方式：**逐顶点处理、逐像素处理、球谐函数 spherical harmonics SH 处理**
- 决定光源使用哪种模式取决于它 的类型和渲染模式
 - **类型**：平行光还是其它光源
 - **渲染模式**：该光源是否是重要的（需要逐像素对待它）
- 我们可以在光源的 Light 组件中设置这些属性



- 在前向渲染中，Unity 会根据场景中的各个光源的设置以及这些光源对物体 的影响程度（距离物体的远近、光源强度等）对光源进行重要度排序，一定数目的光源会按照逐像素的方式进行处理，最多 4 个光源按照逐顶点的方式处理，剩下的光源按照 SH 的方式处理

判断规则

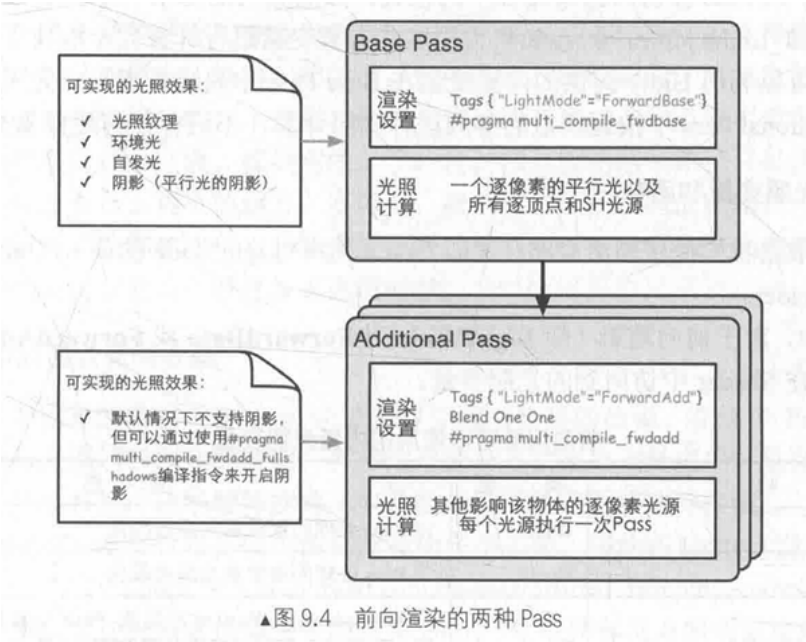
- 场景中**最亮的平行光**总是按逐像素处理
- 渲染模式被设置成 Not Important 的光源，会按照逐顶点或者 SH 处理
- 渲染模式被设置成 **Important** 的光源，会按照逐像素处理
- 根据上述规则德奥的逐像素光源数量小于 Quality Setting 中的逐像素光源数量 Light Count，则会有更多的光源按照逐像素的方式渲染

光照计算位置

前向渲染的光照计算在两种 Pass 中计算

- Base Pass
- Additional Pass

	Base Pass	Additional Pass
可实现的光照效果	光照纹理 环境光 自发光 阴影（平行光的阴影）	默认情况下不支持阴影 但可以通过 #pragma multi_compile_fwdadd_fullshadows 编译指令来开启阴影
渲染设置	Tags{"LightMode" = "ForwardBase" #pragma multi_compile_fwdbase	Tags{"LightMode" = "ForwardBase" Blend One One #pragma multi_compile_fwdadd
光照计算	一个逐像素的平行光以及所有逐顶点和 SH 光源	其它影响该物体的逐像素光源，每个光源执行一次 Pass-



- Base Pass 中渲染的平行光**默认支持阴影**，而 Additional Pass 中渲染的光源在默认情况下没有阴影效果，但是可以使用 `#pragma multi_compile_fwdbase` 开启
- 环境光**和**自发光**在 Base Pass 中计算
- Additional Pass 中开启了**混合模式**，因为我们希望每个 Additional Pass 可以与上一次光照结果在帧缓存中进行叠加，如果没有开启混合模式，那么 Additional Pass 的渲染结果会覆盖掉之前的结果，看起来好像该物体只受该光源的影响，通常使用的混合模式是 Blend One One
- 前向渲染中，一个 Unity Shader 通常会定义**一个 Base Pass**（也可以定义多次，比如双面渲染），以及**一个 Additional Pass**，一个 Base Pass 仅执行一次，而一个 Additional Pass 会根据该影响该物体的其它逐像素光源数目被多次调用，即每个逐像素光源都会执行一次 Additional Pass

内置光照变量和函数

对于前向渲染 LightMode 为 ForwardBase 或 ForwardAdd 来说，我们可以在 Shader 中访问到的光照变量

前向渲染可用内置光照变量

名称	类型	描述
_LightColor0	float4	该 Pass 处理逐像素光源的颜色
_WorldSpaceLightPos0	float4	_WorldSpaceLightPos0.xyz 是该 Pass 处理逐像素光源的位置 如果该光源是平行光，那么 _WorldSpaceLightpos0.w 是 0，其它光源类型 w 值为 1
_LightMatrix0	float4x4	从世界空间到光源空间的变换矩阵，用于采样 cookie 和光强衰减 attenuation 纹理
unity_4LightPosX0, unity_4LightPosY0, unity_4LightPosz0	float4	仅用于 Base Pass，前四个非重要点光源在世界空间中的位置
unity_4LightAtten0	float4	仅用于 BasePass，前四个非重要点光源的衰减因子
unity_LightColor	half4[4]	仅用于 Base Pass，前四个非重要点光源的颜色

前向渲染可用内置光照函数

函数名	描述
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，返回世界空间中从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 Shade4PointLights(...)	仅用于前向渲染，计算四个点光源逐顶点光照（参数是上面提到的内置变量）

顶点照明渲染路径

- 顶点光照对硬件配置要求最少，性能最高，效果最差
- 它不支持逐像素得到的效果：阴影、法线映射、高精度的高光反射

内置光照变量和函数

- 一个顶点照明的 Pass 最多可以访问到 8 个逐顶点光源

顶点照明渲染路径可用内置光照变量

名称	类型	描述
unity_LightColor	half4[8]	光源颜色
unity_LightPosition	float4[8]	xyz 分量为视角空间中光源位置，如果是平行光 z 分量为 0，其它为 1
unity_LightAtten	half[8]	光源衰减因子，如果光源是聚光灯，x 分量是 cos(spotAngle/2)，y 分量是 1/cos(spotAangle/4)，如果是其它类型的光源，x 分量是 -1，y 分量是 1，z 分量是衰减的平方，w 分量是光源范围开根号的结果
unity_SpotDirection	float[8]	如果光源是聚光灯，值为视角空间的聚光灯位置，其他类型的值为 (0, 0, 1, 0)

顶点照明渲染路径可用内置光照函数

函数名	描述
float3 ShadeVertexLights(float4 vertex, float3 normal)	输入模型空间中的顶点位置和法线，计算四个逐顶点光源的光照以及环境光

延迟渲染路径

- 前向渲染的问题：当场景中包含**大量实时光源**时，这些光源影响的区域互相重叠，该区域内的每个物体执行多个 Pass 来计算不同光源对该物体的影响，然后再颜色缓存中混合得到最终的光照，然而每执行一个 Pass 我们需要**重新渲染**一遍物体，很多计算实际上是重复的
- 延迟渲染利用额外的缓冲区，**G 缓冲 G-buffer**，存储我们关心的表面的信息（法线、位置、材质属性翰）

延迟渲染原理

主要包含两个 Pass

- 第一个 Pass：不进行任何光照计算，仅计算哪些片元可见（深度缓冲），当一个片元可见，我们就把它的相关信息存储到 G-buffer 中
- 第二个 Pass：利用 G-buffer 的各个片元信息（表面法线、视角方向、漫反射系数等）进行光照计算

伪代码

```
Pass1{
    for(each primitive in this model){
        for(each fragment covere dby this primitive){
            if(failed in depth test){
                discard;
            }else{
                writeGBuffer(materialInfo, pos, normal, lightDir, viewDir);
            }
        }
    }
}

Pass2{
    for(each pixel in the screen){
        if(pixel is valid){
            pixel, materialInfo, pos, normal, lightDir, viewDir = readGBuffer();
            float4 color = Shading(materialInfo, pos, normal, lightDir, viewDir);
            writeFrameBuffer(pixel, color);
        }
    }
}
```

- 延迟渲染的效率不依赖场景复杂度，而和屏幕空间大小有关

Unity 中的延迟渲染

- 延迟渲染路径适合场景中**光源数目多，使用前向渲染造成性能瓶颈**的情况下使用
- 缺点
 - 不支持抗锯齿功能
 - 不能处理半透明物体
 - 对显卡有一定要求
- 使用延迟渲染 Unity 要求我们提供两个 Pass

- 第一个 Pass：渲染 G-Buffer，在这个 Pass 中，我们会把物体的漫反射颜色、高光反射颜色、平滑度、法线、自发光和深度信息等渲染到屏幕空间的 G-Buffer 中，对每个物体该 Pass 仅执行一次
- 第二个 Pass：计算真正的光照模型
- G-Buffer 中包含以下渲染纹理
 - RT0：RGB 通道存储漫反射颜色
 - RT1：RGB 通道存储高光反射颜色，A 通道存储高光反射指数部分
 - RT2：RGB 通常存储法线
 - RT3：存储自发光 + lightmap + 反射探针
 - 深度缓冲和模板缓冲

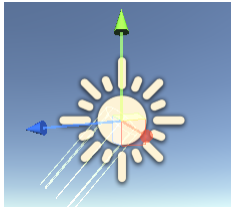
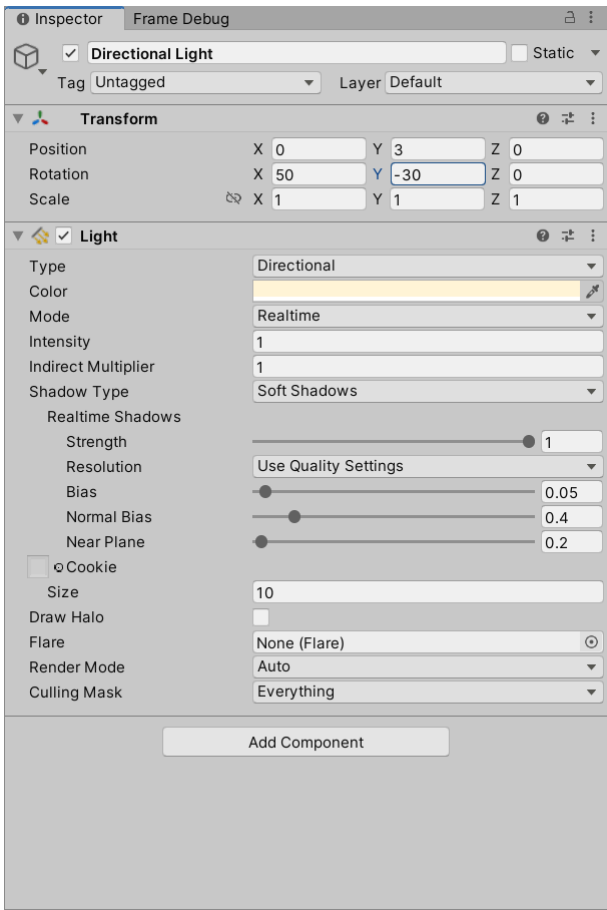
2. Unity 的光源类型

光源类型有什么影响

光源属性

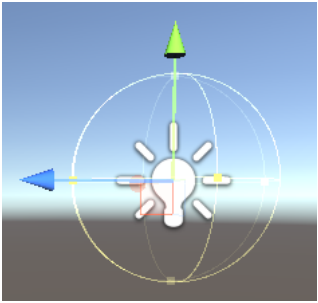
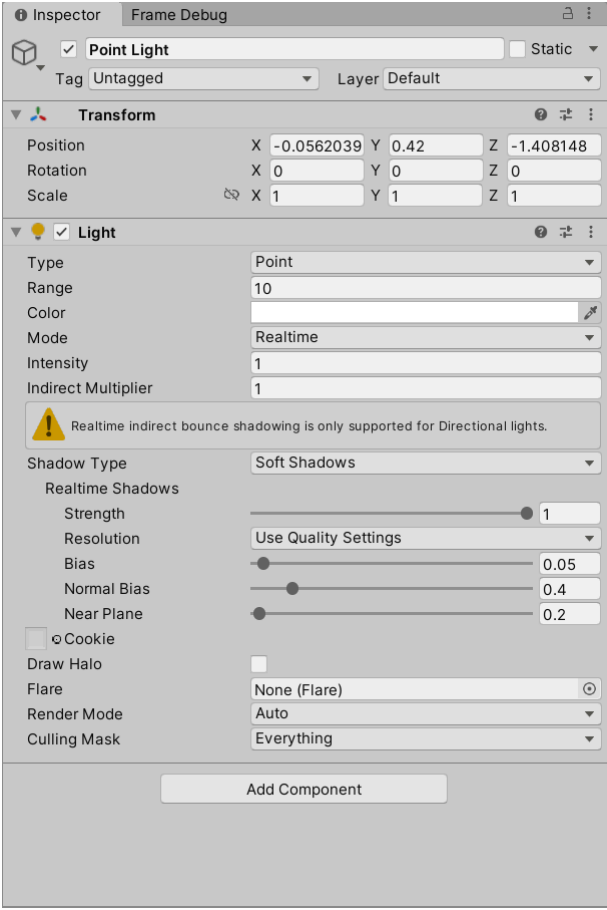
Shader 中最常使用的光源属性有：**光源位置、（到某点的）方向、颜色、强度、衰减**

平行光



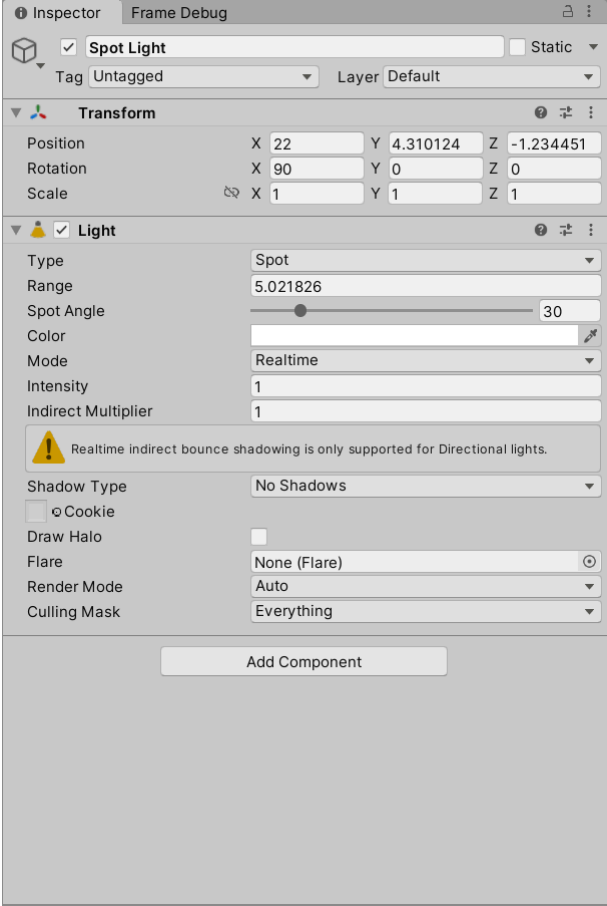
- 照亮的范围没有限制，通常作为**太阳**的角色在场景中出现
- 它**没有位置**，几何属性**只有方向**，可以调节平行光 **Transform 组件的 Rotation 属性**来改变光源方向
- 平行光**到场景中所有点的方向一样**

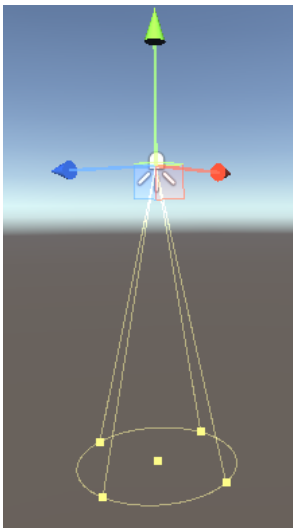
点光源



- 照亮空间有限，是一个球体
- 由一个点发出，向所有方向延伸的光
- 方向属性：点光源位置 - 某点位置
- 点光源会衰减，随着物体远离点光源，它接收到的光照强度会逐渐减小

聚光灯





- 照亮空间有限，是一块锥形区域
- 由特定位置出发，向特定方向延伸的光
- 方向属性：点光源位置 - 某点位置
- 聚光灯的衰减随着物体远离而减小，在锥形顶点的光照强度最强，在边界处强度为 0

前向渲染中处理不同的光源类型

实践 Bulin-Phong 前向渲染 Shader

```
Shader "Unity Shaders Book/Chapter 9/Forward Rendering" {
    Properties {
        _Diffuse ("Diffuse", Color) = (1, 1, 1, 1)
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }
    SubShader {
        Tags { "RenderType"="Opaque" }

        // Base Pass
        Pass {
            // Pass for ambient light & first pixel light (directional light)
            Tags { "LightMode"="ForwardBase" }

            CGPROGRAM

            // Apparently need to add this declaration
            // 正确赋值光照衰减等光照变量
            #pragma multi_compile_fwdbase

            #pragma vertex vert
            #pragma fragment frag

            #include "Lighting.cginc"

            fixed4 _Diffuse;
            fixed4 _Specular;
            float _Gloss;

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

            struct v2f {
                float4 pos : SV_POSITION;
                float3 worldNormal : TEXCOORD0;
                float3 worldPos : TEXCOORD1;
            };

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

```

        o.worldNormal = UnityObjectToWorldNormal(v.normal);

        o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;

        return o;
    }

    fixed4 frag(v2f i) : SV_Target {
        fixed3 worldNormal = normalize(i.worldNormal);
        fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz); //
        _WorldSpaceLightPos0获取平行光方向（位置没有意义）

        // 计算场景环境光（只在Base Pass中计算一次，后续Additional Pass不再计算）
        fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

        // 如果场景中包含多个平行光，Unity会选择最亮的平行光传递给Base Pass及逆行处理，其它平
        行光按照逐点或在Additional Pass中逐像素处理
        // Base Pass中处理的逐像素光源类型一定是平行光
        fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * max(0,
        dot(worldNormal, worldLightDir));

        fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);
        fixed3 halfDir = normalize(worldLightDir + viewDir);
        fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
        dot(worldNormal, halfDir)), _Gloss);

        fixed atten = 1.0; // 平行光没有衰减的，这里可以令衰减值为1.0

        return fixed4(ambient + (diffuse + specular) * atten, 1.0);
    }

    ENDCG
}

// Additional Pass
Pass {
    // Pass for other pixel lights
    Tags { "LightMode"="ForwardAdd" }

    Blend One One // Blend混合模式，光照叠加

    CGPROGRAM

    // Apparently need to add this declaration
    // 在Additional Pass中访问正确的光照变量
    #pragma multi_compile_fwdadd

    #pragma vertex vert
    #pragma fragment frag

    #include "Lighting.cginc"
    #include "AutoLight.cginc"

    fixed4 _Diffuse;
    fixed4 _Specular;
    float _Gloss;

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

```



```

struct v2f {
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
};

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

    o.worldNormal = UnityObjectToWorldNormal(v.normal);

    o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;

    return o;
}

fixed4 frag(v2f i) : SV_Target {
    // 不再计算场景环境光ambient

    fixed3 worldNormal = normalize(i.worldNormal);

    // 计算不同光源的方向
#ifdef USING_DIRECTIONAL_LIGHT
        fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
    #else
        fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz -
i.worldPos.xyz);
    #endif

    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * max(0,
dot(worldNormal, worldLightDir));

    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);
    fixed3 halfDir = normalize(worldLightDir + viewDir);
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

    // 处理不同光源的衰减
#ifdef USING_DIRECTIONAL_LIGHT
        fixed atten = 1.0;
    #else
        // Unity 使用一张纹理作为查找表，以在片元着色器中得到光照的衰减
        #if defined (POINT)
            float3 lightCoord = mul(unity_WorldToLight, float4(i.worldPos,
1)).xyz; // 首先得到光源空间下的坐标
            fixed atten = tex2D(_LightTexture0, dot(lightCoord,
lightCoord).rr).UNITY_ATTEN_CHANNEL; // 使用该坐标对衰减纹理进行采样得到衰减值
        #elif defined (SPOT)
            float4 lightCoord = mul(unity_WorldToLight, float4(i.worldPos,
1));
            fixed atten = (lightCoord.z > 0) * tex2D(_LightTexture0,
lightCoord.xy / lightCoord.w + 0.5).w * tex2D(_LightTextureB0, dot(lightCoord,
lightCoord).rr).UNITY_ATTEN_CHANNEL;
        #else
            fixed atten = 1.0;
        #endif
    #endif

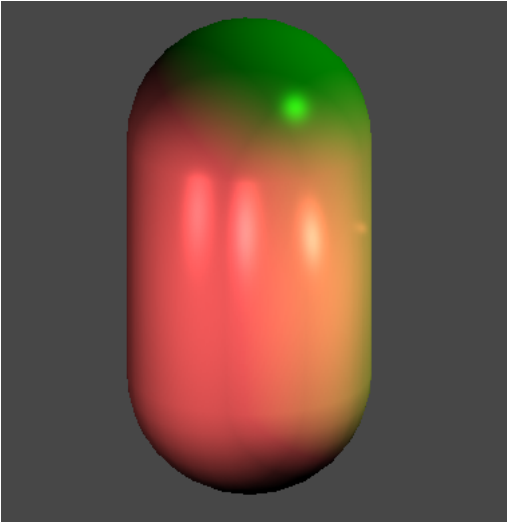
    return fixed4((diffuse + specular) * atten, 1.0);
}

```



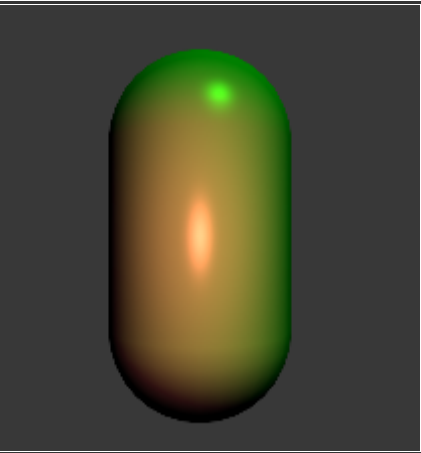
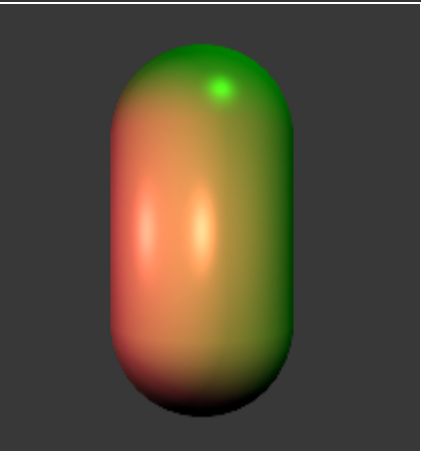
```
        ENDCG
    }
}
Fallback "Specular"
}
```

实验 Base Pass 和 Additional Pass 的调用

创建四个点光源和一个平行光源



- 默认情况下一个物体可以接受除最亮的平行光外 4 个逐像素光照
- 平行光会在 Base Pass 中按照逐像素的方式被处理
- 4 个点光源会在 Additional Pass 中逐像素的方式被处理

渲染事件			描述
			清除颜色、深度和模板缓冲，为后面的渲染做准备
			使用 Base Pass 将平行光的光照渲染到帧缓存中
			使用 Additional Pass 渲染第一个点光源
			使用 Additional Pass 渲染第二个点光源

渲染事件		描述
		使用 Additional Pass 渲染第三个点光源
		使用 Additional Pass 渲染第四个点光源

- 处理光源的顺序按照它们的重要度排序
- 点光源的颜色强度在这里都相同，因此重要程度取决于距离物体的远近

3. Unity 的光照衰减

用于光照衰减的纹理

Unity 内部使用一张名为 `_LightTexture0` 的纹理来计算光照衰减，通常只关心对角线上的纹理颜色值，这表明在光源空间中不同位置的点的衰减程度

- (0, 0)：与光源位置重合的点的衰减值
- (1, 1)：在光源空间所关心距离最远的点的衰减值

为了对 `_LightTexture0` 采样，我们需要首先知道该点在光源空间的坐标

```
float3 lightCoord = mul(unity_WorldToLight, float4(i.worldPos, 1)).xyz;
```

然后可以使用这个坐标的模的平方对衰减纹理进行采样

```
fixed atten = tex2D(_LightTexture0, dot(lightCoord, lightCoord).rr).UNITY_ATTEN_CHANNEL;
```

- 使用 `UNITY_ATTEN_CHANNEL` 得到衰减纹理中衰减值所在的分量，得到最终的衰减值

使用数学公式计算衰减

有时候可以使用数学公式计算光源的衰减

4. Unity 的阴影

如何实现阴影

当一个光源发射一条光线到一个不透明的物体时，这条光线就不可以继续照亮其他物体（不考虑反射），因此，物体会向旁边的物体投射阴影

Shadow Map

在实时渲染中，最常使用这个技术。它会首先把**摄像机的位置放在与光源重合的位置上**，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**

在前向渲染路径中，如果场景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**，该纹理记录了从光源位置出发，能看到的场景中距离它最近的表面位置（深度信息）

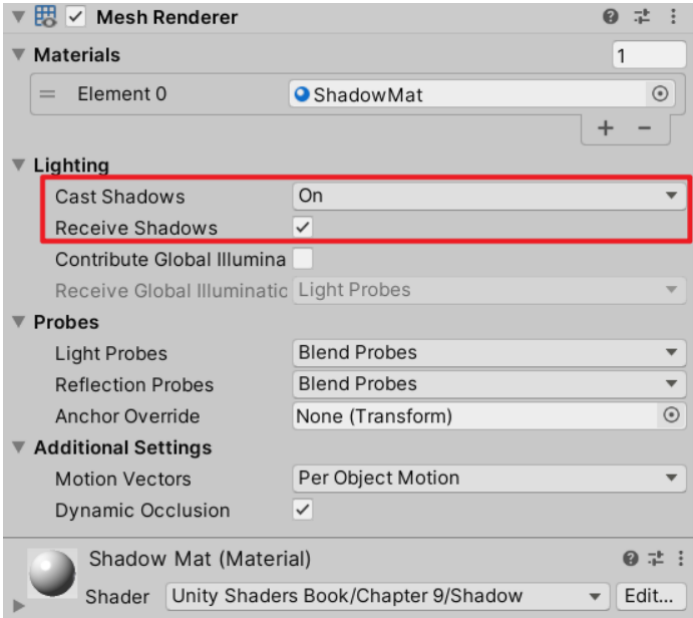
Unity 使用一个额外的 Pass 来专门计算阴影映射纹理，这个 Pass 就是 **LightMode** 标签被设置为 **ShadowCaster** 的 Pass，Unity 会把摄影机**放在光源的位置上**，然后调用该 Pass，并据此输出深度信息来存储到阴影映射纹理中，在渲染时，Unity 会首先找到 ShadowCaster 的 Pass，使用该 Pass 来更新光源的阴影映射纹理

接收阴影与阴影投射

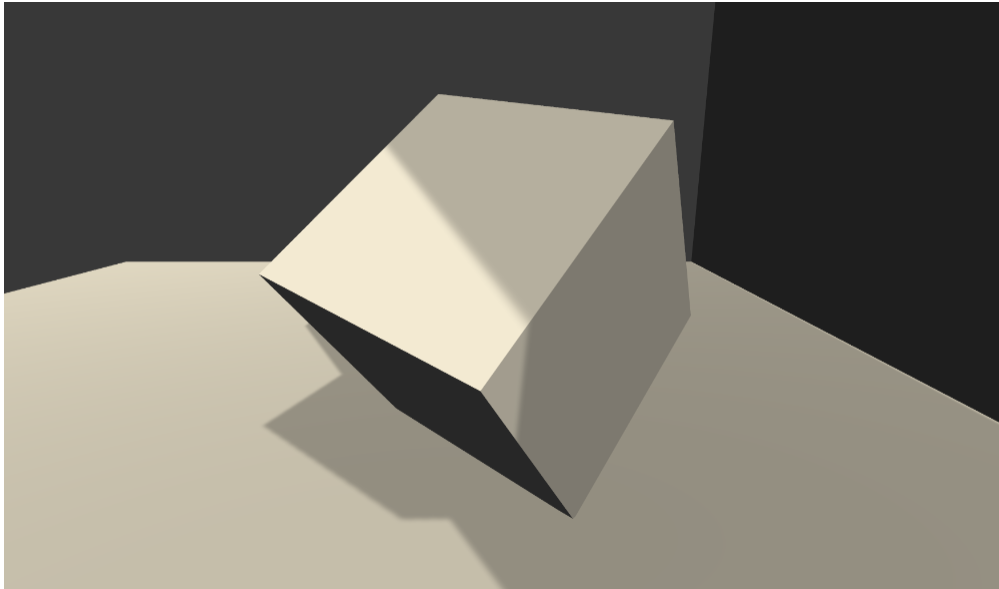
- **物体接收阴影**：在 Shader 的阴影映射纹理（包括屏幕空间的阴影图）进行采样，把采样结果和最后光照结果**相乘**得到阴影效果
- **物体投射阴影**：把该物体加入到光照阴影映射纹理的计算中，该物体执行 LightMode 为 ShadowCaster 的 Pass

不透明物体的阴影

在 Unity 中，可以选择是否让物体投射或接受阴影



- **Cast Shadow**: Unity 会把该物体加入到光源的阴影映射纹理计算中，从而让其他物体在对阴影纹理计算时可以得到该物体的相关信息
 - **Two Sided**: 允许对物体的所有面计算阴影信息
- **Receive Shadow**: 选择是否让物体接收来自其它物体的阴影，如果没有开启，那么当我们调用 Unity 内置的宏和变量计算阴影时，这些宏就不会计算



投射阴影

在 ForwardRendering 的 Shader 中，我们使用了内置的 Specular 回调，里面涉及处理投射阴影的代码

```
Pass{
    Name "ShadowCaster"
    Tags {"LightMode" = "ShadowCaster"}

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_shadowcaster
    #include "UnityCG.cginc"

    struct v2f
    {
        V2F_SHADOW_CASTER;
    };

    v2f vert(appdata_base v)
    {
        v2f o;
        TRANSFER_SHADOW_CASTER_NORMALOFFSET(o)
        return o;
    }

    float4 frag(v2f i) : SV_Target
    {
        SHADOW_CASTER_FRAGMENT(i);
    }
}
```

```

    }
    ENGCG
}

```

- 把深度信息写入到渲染目标中，比如光源的阴影映射纹理，或者摄像机的深度纹理
- 如果我们把 Fallback = "Specular" 注释掉，**正方体就不会再向平面投射阴影了**

我们也可以定义自己的 LightMode 为 ShadowCaster 的 Pass，灵活控制阴影的产生，由于这个 Pass 的功能通常是在多个 Unity Shader 间通用，因此直接 Fallback 是更加方便的调用方法

接收阴影

```

Shader "Unity Shaders Book/Chapter 9/Shadow" {
    Properties {
        _Diffuse ("Diffuse", Color) = (1, 1, 1, 1)
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }
    SubShader {
        Tags { "RenderType"="Opaque" }

        Pass {
            // Pass for ambient light & first pixel light (directional light)
            Tags { "LightMode"="ForwardBase" }

            CGPROGRAM

            // Apparently need to add this declaration
            #pragma multi_compile_fwdbase

            #pragma vertex vert
            #pragma fragment frag

            // Need these files to get built-in macros
            #include "Lighting.cginc" // 计算阴影所需的内置宏
            #include "AutoLight.cginc"

            fixed4 _Diffuse;
            fixed4 _Specular;
            float _Gloss;

            // 宏会使用上下文变量进相关计算，需要保证自定义的变量名和这些宏中使用的变量名相匹配
            struct a2v {
                float4 vertex : POSITION; // 注意这个变量的命名
                float3 normal : NORMAL;
            };

            struct v2f {
                float4 pos : SV_POSITION; // 注意这个变量的命名
                float3 worldNormal : TEXCOORD0;
                float3 worldPos : TEXCOORD1;
                SHADOW_COORDS(2) // 新的内置宏，对阴影纹理坐标的采样，注意这个宏需要的参数是下一个可用的插值索引器的索引值
            };

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

                o.worldNormal = UnityObjectToWorldNormal(v.normal);

                o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;

                // Pass shadow coordinates to pixel shader
            }
        }
    }
}

```

```

TRANSFER_SHADOW(o); // 另一个内置宏, 用于在顶点着色器上进一步声明阴影纹理坐标

return o;
}

fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);

    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * max(0,
dot(worldNormal, worldLightDir));

    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);
    fixed3 halfDir = normalize(worldLightDir + viewDir);
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

    fixed atten = 1.0;

    fixed shadow = SHADOW_ATTENUATION(i); // 计算阴影值

    return fixed4(ambient + (diffuse + specular) * atten * shadow, 1.0);
}

ENDCG

}

Pass {
    // Pass for other pixel lights
    Tags { "LightMode"="ForwardAdd" }

    Blend One One

    CGPROGRAM

    // Apparently need to add this declaration
    #pragma multi_compile_fwdadd
    // Use the line below to add shadows for point and spot lights
    // #pragma multi_compile_fwdadd_fullshadows

    #pragma vertex vert
    #pragma fragment frag

    #include "Lighting.cginc"
    #include "AutoLight.cginc"

    fixed4 _Diffuse;
    fixed4 _Specular;
    float _Gloss;

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

    struct v2f {
        float4 position : SV_POSITION;
        float3 worldNormal : TEXCOORD0;
        float3 worldPos : TEXCOORD1;
    };

```



```

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

    o.worldNormal = UnityObjectToWorldNormal(v.normal);

    o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;

    return o;
}

fixed4 frag(v2f i) : SV_Target {
    fixed3 worldNormal = normalize(i.worldNormal);
    #ifdef USING_DIRECTIONAL_LIGHT
        fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz);
    #else
        fixed3 worldLightDir = normalize(_WorldSpaceLightPos0.xyz -
i.worldPos.xyz);
    #endif

    fixed3 diffuse = _LightColor0.rgb * _Diffuse.rgb * max(0,
dot(worldNormal, worldLightDir));

    fixed3 viewDir = normalize(_WorldSpaceCameraPos.xyz - i.worldPos.xyz);
    fixed3 halfDir = normalize(worldLightDir + viewDir);
    fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

    #ifdef USING_DIRECTIONAL_LIGHT
        fixed atten = 1.0;
    #else
        float3 lightCoord = mul(unity_WorldToLight, float4(i.worldPos,
1)).xyz;
        fixed atten = tex2D(_LightTexture0, dot(lightCoord,
lightCoord).rr).UNITY_ATTEN_CHANNEL;
    #endif

    return fixed4((diffuse + specular) * atten, 1.0);
}

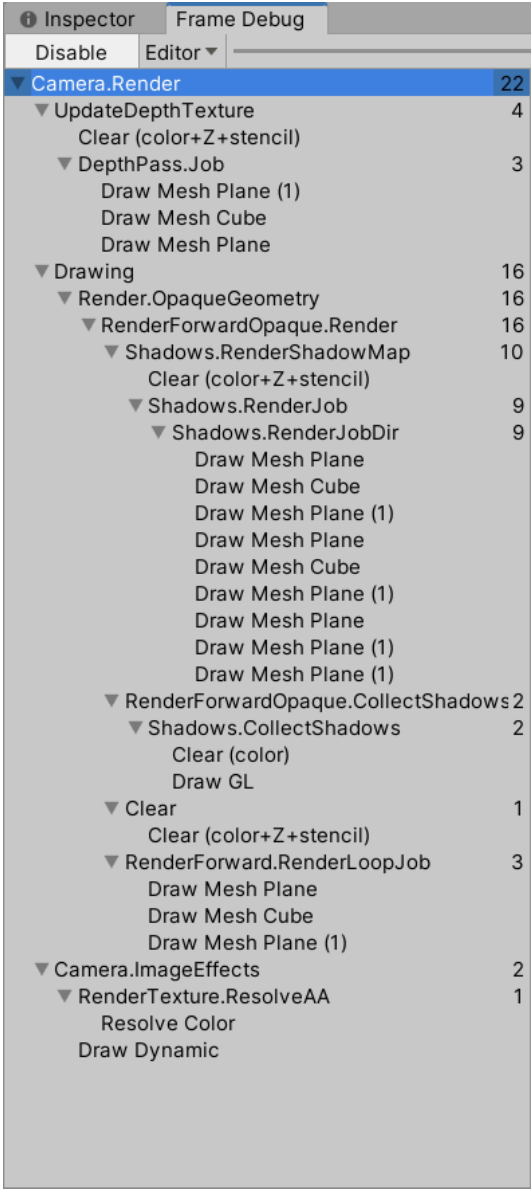
ENDCG
}

}
FallBack "Specular"
}

```

阴影绘制过程

可以使用 Unity 自带的 Frame Debugger 分析结果



绘制该场景主要分为几个重点渲染事件

事件	说明	渲染图像
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得到屏幕空间的阴影图	

透明物体的阴影

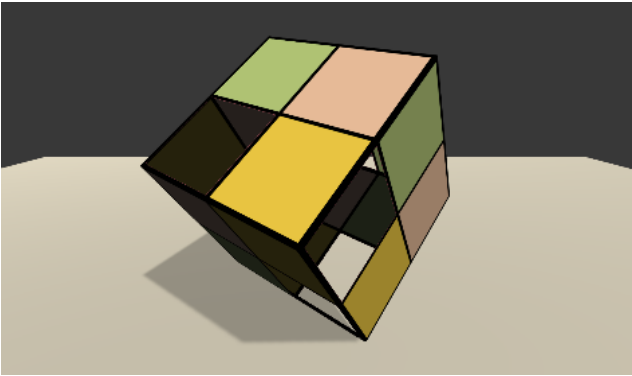
投射阴影

对于透明物体来说，我们需要小心处理它的阴影（不能简单的使用 FallBack Specular 或者 VertexLit)

透明度测试

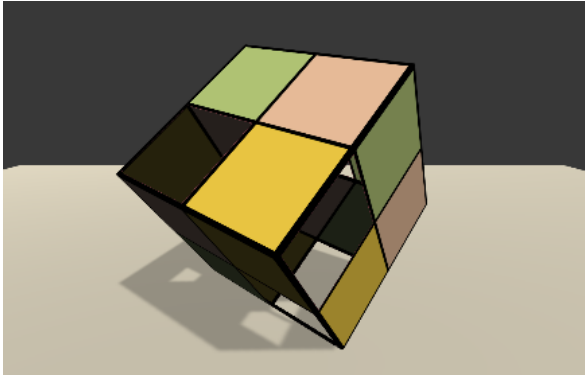
- 如果使用 VertexLit，Diffuse 或者 Specular 等作为回调，无法得到正确阴影

-



- 如图所示，镂空区域出现不正常的阴影，看起来像是一个普通的正方体，然而我们想要有些光可以透过来
- 为了得到正确的效果，可以将 Fallback 改成 Transparent/Cutout/VertexLit

-



- 注意，Transparent/Cutout/VertexLit 计算透明度测试时，使用了名为 `_Cutoff` 的属性进行透明度测试，所以我们的 Shader 也**必须使用一样名字的属性**，否则无法得到正确结果
- 另外，我们这里将 CastShadows 属性设置为 TwoSided，强制 Unity 再计算阴影映射纹理时计算所有面的深度信息

接收阴影

所有**内置透明度混合的 Unity Shader 都没有包含阴影投射的 Pass**，这些半透明物体不会参与深度图和阴影映射纹理的计算，它们不会向其他物体投射阴影，也不会接收来自其他物体的阴影

为了使得半透明物体产生正确的阴影，需要再每个光源空间下严格按照从后往前的顺序进行渲染，这非常复杂且影响性能

我们可以使用一些 dirty trick，将 Fallback 设置为 Vertex、Diffuse 这些不透明物体使用的 Shader，然后在 Mesh Renderer 组件上的 Cast Shadows 和 Receive Shadows 选项来控制是否需要向其他物体投射或者接收阴影

管理光照衰减和阴影

前面已经讲过，Unity Shader 在前向路径中计算光照衰减的方法

- Base Pass：平行光的衰减因子为 1
- Additional Pass：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

光照衰减与阴影值对物体最终的渲染结果本质上是相同的，都是把光照衰减因子和阴影值及光照结果相乘，得到最终的渲染结果，Unity 中有内置的 `UNITY_LIGHT_ATTENUATION` 实现

实践 - 光照衰减阴影 Shader

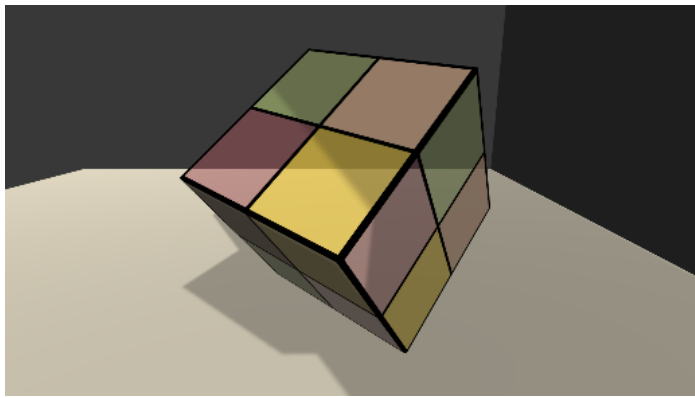
与之前的 Shader 的区别在于，我们在片元着色器中使用内置宏 `UNITY_LIGHT_ATTENUATION` 计算光照衰减和阴影

```
fixed4 frag(v2f i) : SV_Target
{
    ...

    // UNITY_LIGHT_ATTENUATION not only compute attenuation, but also shadow infos
    UNITY_LIGHT_ATTENUATION(atten, i, i.worldPos);
    return fixed4(ambient + (diffuse + specular) * atten, 1.0);
}
```

- `UNITY_LIGHT_ATTENUATION`：内置用于计算光照衰减和阴影的宏，定义在 AutoLight.cginc 下面，它接收 3 个参数，
 - 第一个参数：光照衰减和阴影值相乘的结果（注意 `atten` 是这个宏声明的变量，不需要我们声明）
 - 第二个参数：结构体 `v2f`
 - 第三个参数：世界空间坐标（用于计算光源空间下的坐标，再对光照衰减纹理采样得到光照衰减）

如果希望再 Additional Pass 中添加阴影效果，则需要使用 `#pragma multi_compile_fwdadd_fullshadows` 编译指令代替之前的 `#pragma multi_compile_fwdadd`，这样 Unity 会额外逐像素计算阴影



5. 标准 Unity Shader

提供两个标准的 Unity Shader，包括了多光源、阴影和光照衰减的标准光照着色器

- **BumpedDiffuse**：使用 Phong 光照模型
- **BumpedSpecular**：使用 Blinn-Phong 光照模型