```
Shader "Unity Shaders Book/Chapter 17/Bumped Diffuse" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BumpMap ("Normalmap", 2D) = "bump" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 300

        CGPROGRAM
        #pragma surface surf Lambert
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _BumpMap;
        fixed4 _Color;

        struct Input {
            float2 uv_MainTex;
            float2 uv_BumpMap;
        };

        void surf (Input IN, inout SurfaceOutput o) {
            fixed4 tex = tex2D(_MainTex, IN.uv_MainTex);
            o.Albedo = tex.rgb * _Color.rgb;
            o.Alpha = tex.a * _Color.a;
            o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));
        }

        ENDCG
    }

    FallBack "Legacy Shaders/Diffuse"
}
```

==========

编译指令

17.2 编译指令

我们首先来看一下表面着色器的编译指令。编译指令是我们和 Unity 沟通的重要方式，通过它可以告诉 Unity：“嘿，用这个表面函数设置表面属性，用这个光照模型模拟光照，我不要阴影和环境光，不要雾效！”只需要一句代码，我们就可以完成这么多事情！

编译指令最重要的作用是指明该表面着色器使用的**表面函数**和**光照函数**，并设置一些可选参数。表面着色器的 CG 块中的第一句代码往往就是它的编译指令。编译指令的一般格式如下：

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

其中，**#pragma surface** 用于指明该编译指令是用于定义表面着色器的，在它的后面需要指明使用的表面函数（surfaceFunction）和光照模型（lightModel），同时，还可以使用一些可选参数来控制表面着色器的一些行为。

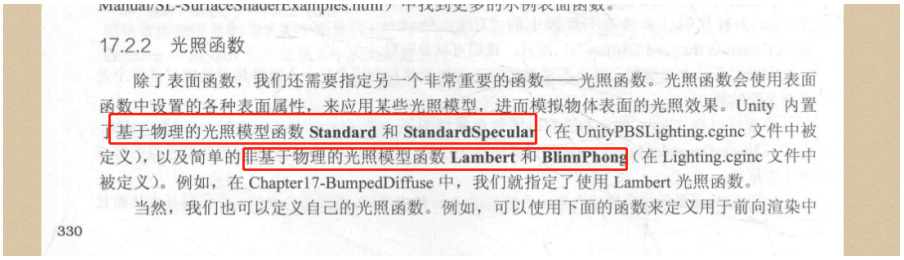
17.2.1 表面函数

我们之前说过，表面着色器的优点在于抽象出了“表面”这一概念。与之前遇到的顶点/片元抽象层不同，一个对象的表面属性定义了它的反射率、光滑度、透明度等值。而编译指令中的 surfaceFunction 就用于定义这些表面属性。surfaceFunction 通常就是名为 surf 的函数（函数名可以是任意的），它的函数格式是固定的：

```
void surf (Input IN, inout SurfaceOutput o)
void surf (Input IN, inout SurfaceOutputStandard o)
void surf (Input IN, inout SurfaceOutputStandardSpecular o)
```

其中，后两个是 Unity 5 中由于引入了基于物理的渲染而新添加的两种结构体。**SurfaceOutput**、**SurfaceOutputStandard** 和 **SurfaceOutputStandardSpecular** 都是 Unity 内置的结构体，它们需要配合不同的光照模型使用，我们会在下一节进行更详细地解释。

在表面函数中，会使用输入结构体 **Input IN** 来设置各种表面属性，并把这些属性存储在输出结构体 SurfaceOutput、SurfaceOutputStandard 或 SurfaceOutputStandardSpecular 中，再传递给光照函数计算光照结果。读者可以在 Unity 手册中的**表面着色器的例子**一文（http://docs.unity3d.com/Manual/SL-SurfaceShaderExamples.html）中找到更多的示例表面函数。



17.2.2 光照函数

除了表面函数，我们还需要指定另一个非常重要的函数——光照函数。光照函数会使用表面函数中设置的各种表面属性，来应用某些光照模型，进而模拟物体表面的光照效果。Unity 内置了基于物理的光照模型函数 **Standard** 和 **StandardSpecular**（在 UnityPBSLighting.cginc 文件中被定义），以及简单的非基于物理的光照模型函数 **Lambert** 和 **BlinnPhong**（在 Lighting.cginc 文件中被定义）。例如，在 Chapter17-BumpedDiffuse 中，我们就指定了使用 Lambert 光照函数。

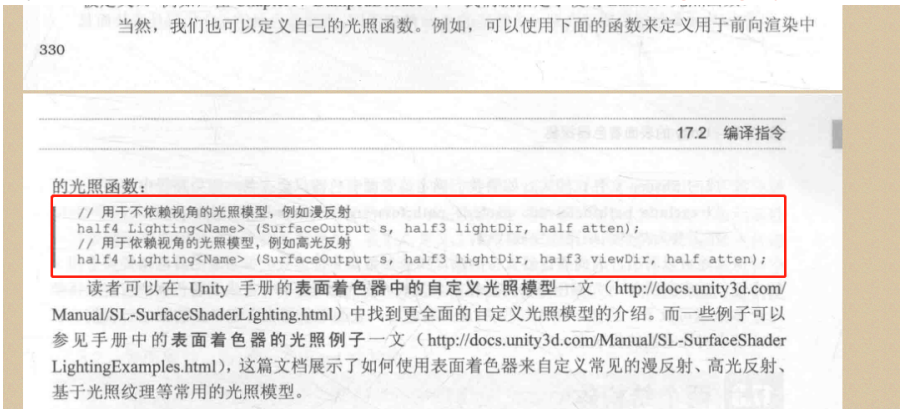
当然，我们也可以定义自己的光照函数。例如，可以使用下面的函数来定义用于前向渲染中

330

自定义光照模型，Lighting<Name>

例如:#pragma surface surf MyCustom

MyCustom就是自己定义的光照模型，然后光照函数名要定义成LightingMyCustom



当然，我们也可以定义自己的光照函数。例如，可以使用下面的函数来定义用于前向渲染中

330

17.2 编译指令

的光照函数：

```
// 用于不依赖视角的光照模型，例如漫反射
half4 Lighting<Name> (SurfaceOutput s, half3 lightDir, half atten);
// 用于依赖视角的光照模型，例如高光反射
half4 Lighting<Name> (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten);
```

读者可以在 Unity 手册的**表面着色器中的自定义光照模型**一文（http://docs.unity3d.com/Manual/SL-SurfaceShaderLighting.html）中找到更全面的自定义光照模型的介绍。而一些例子可以参见手册中的**表面着色器的光照例子**一文（http://docs.unity3d.com/Manual/SL-SurfaceShaderLightingExamples.html），这篇文档展示了如何使用表面着色器来自定义常见的漫反射、高光反射、基于光照纹理等常用的光照模型。

其他编译指令

```
自定义修改函数
{
    顶点修改函数
    最后的颜色修改函数
}

阴影
{
    阴影 addshadow      ==>为shader          生成一个阴影投射的pass
    fullforwardshadows  ==>在前向渲染中之所有光源类型
    noshadow            ==>禁用阴影
}

透明度测试和透明度混合
{
    alpha ==> 透明度混合
    alphaTest ==> 透明度测试
}

光照
{
    noambient ==>告诉unity不应用任何环境光照和光照探针
    novertexlights ==>告诉unity不应用不应用任何逐顶点光照
    nofrowardadd ==> 去掉所有前向渲染中额外的pass,
        也就是说这个shader只支持一个逐像素的平行光,其他光源会按照逐顶点或SH的方法计算光照影响
}

控制代码的生成 :
默认情况下，unity会为一个表面着色器生成对应的前向渲染路径，延迟渲染路径使用的pass,会导致shader文件较大
如果可以确定shader的渲染路径，就告诉unity不要为某些渲染路径生成代码
{
  exclude_path:deferred
  exclude_path:forward
  exclude_path:prepass
}
```

默认情况下，unity会为一个表面着色器生成对应的前向渲染路径，延迟渲染路径使用的pass,会导致shader文件较大 如果可以确定shader的渲染路径，就告诉unity不要为某些渲染路径生成代码

==========

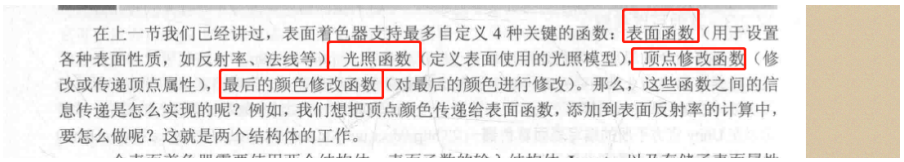
可以自定义的函数

表面函数 surf　　==> 用于设置各种表面性质： 反射率 法线

光照函数　　==> 表面使用的光照模型

顶点修改函数 ==> 修改或传递顶点的属性

最后颜色修改函数 ==> 对最后的颜色进行修改



在上一节我们已经讲过，表面着色器支持最多自定义 4 种关键的函数：表面函数（用于设置各种表面性质，如反射率、法线等），光照函数（定义表面使用的光照模型），顶点修改函数（修改或传递顶点属性），最后的颜色修改函数（对最后的颜色进行修改）。那么，这些函数之间的信息传递是怎么实现的呢？例如，我们想把顶点颜色传递给表面函数，添加到表面反射率的计算中，要怎么做呢？这就是两个结构体的工作。

==========

两个结构体

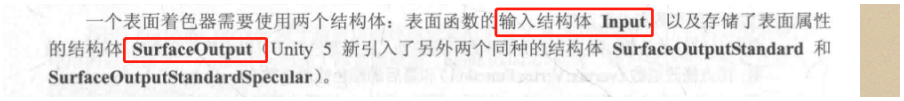
输入结构体 Input

输出结构体

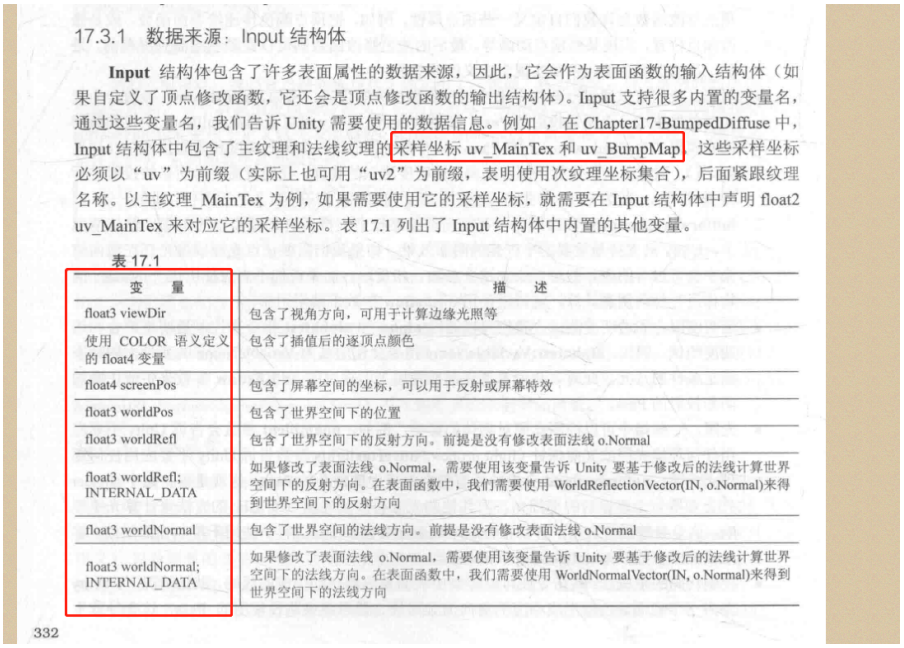
SurfaceOutput （经验光照模型使用）

SurfaceOutputStandard （PBR光照模型使用）

SurfaceOutputStandardSpecular （PBR光照模型使用）



一个表面着色器需要使用两个结构体：表面函数的输入结构体 **Input**，以及存储了表面属性的结构体 **SurfaceOutput**（Unity 5 新引入了另外两个同种的结构体 **SurfaceOutputStandard** 和 **SurfaceOutputStandardSpecular**）。



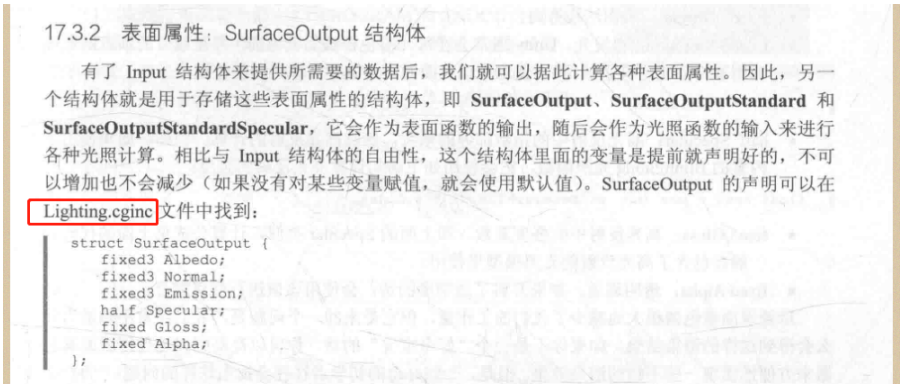
17.3.1 数据来源：Input 结构体

Input 结构体包含了许多表面属性的数据来源，因此，它会作为表面函数的输入结构体（如果自定义了顶点修改函数，它还会是顶点修改函数的输出结构体）。Input 支持很多内置的变量名，通过这些变量名，我们告诉 Unity 需要使用的数据信息。例如 ，在 Chapter17-BumpedDiffuse 中，Input 结构体中包含了主纹理和法线纹理的采样坐标 uv_MainTex 和 uv_BumpMap，这些采样坐标必须以“uv”为前缀（实际上也可用“uv2”为前缀，表明使用次纹理坐标集合），后面紧跟纹理名称。以主纹理_MainTex 为例，如果需要使用它的采样坐标，就需要在 Input 结构体中声明 float2 uv_MainTex 来对应它的采样坐标。表 17.1 列出了 Input 结构体中内置的其他变量。

表 17.1

变　量	描　述
float3 viewDir	包含了视角方向，可用于计算边缘光照等
使用 COLOR 语义定义的 float4 变量	包含了插值后的逐顶点颜色
float4 screenPos	包含了屏幕空间的坐标，可以用于反射或屏幕特效
float3 worldPos	包含了世界空间下的位置
float3 worldRefl	包含了世界空间下的反射方向。前提是没有修改表面法线 o.Normal
float3 worldRefl; INTERNAL_DATA	如果修改了表面法线 o.Normal，需要使用该变量告诉 Unity 要基于修改后的法线计算世界空间下的反射方向。在表面函数中，我们需要使用 WorldReflectionVector(IN, o.Normal)来得到世界空间下的反射方向
float3 worldNormal	包含了世界空间的法线方向。前提是没有修改表面法线 o.Normal
float3 worldNormal; INTERNAL_DATA	如果修改了表面法线 o.Normal，需要使用该变量告诉 Unity 要基于修改后的法线计算世界空间下的法线方向。在表面函数中，我们需要使用 WorldNormalVector(IN, o.Normal)来得到世界空间下的法线方向

332



17.3.2 表面属性：SurfaceOutput 结构体

有了 Input 结构体来提供所需要的数据后，我们就可以据此计算各种表面属性。因此，另一个结构体就是用于存储这些表面属性的结构体，即 **SurfaceOutput**、**SurfaceOutputStandard** 和 **SurfaceOutputStandardSpecular**，它会作为表面函数的输出，随后会作为光照函数的输入来进行各种光照计算。相比与 Input 结构体的自由性，这个结构体里面的变量是提前就声明好的，不可以增加也不会减少（如果没有对某些变量赋值，就会使用默认值）。SurfaceOutput 的声明可以在 Lighting.cginc 文件中找到：

```
struct SurfaceOutput {
    fixed3 Albedo;
    fixed3 Normal;
    fixed3 Emission;
    half Specular;
    fixed Gloss;
    fixed Alpha;
};
```

而 SurfaceOutputStandard 和 SurfaceOutputStandardSpecular 的声明可以在 UnityPBSLighting.cginc 中找到：

```
struct SurfaceOutputStandard
{
    fixed3 Albedo;      // base (diffuse or specular) color
    fixed3 Normal;      // tangent space normal, if written
    half3 Emission;
    half Metallic;      // 0=non-metal, 1=metal
    half Smoothness;    // 0=rough, 1=smooth
    half Occlusion;     // occlusion (default 1)
    fixed Alpha;        // alpha for transparencies
};

struct SurfaceOutputStandardSpecular
{
    fixed3 Albedo;      // diffuse color
    fixed3 Specular;    // specular color
    fixed3 Normal;      // tangent space normal, if written
    half3 Emission;
    half Smoothness;    // 0=rough, 1=smooth
    half Occlusion;     // occlusion (default 1)
    fixed Alpha;        // alpha for transparencies
};
```

在一个表面着色器中，只需要选择上述三者中的其一即可，这取决于我们选择使用的光照模型。Unity 内置的光照模型有两种，一种是 Unity 5 之前的、简单的、非基于物理的光照模型，包括了 **Lambert** 和 **BlinnPhong**；另一种是 Unity 5 添加的、基于物理的光照模型，包括 **Standard** 和 **StandardSpecular**，这种模型会更加符合物理规律，但计算也会复杂很多。如果使用了非基于物理的光照模型，我们通常会使用 **SurfaceOutput** 结构体，而如果使用了基于物理的光照模型 **Standard** 或 **StandardSpecular**，我们会分别使用 **SurfaceOutputStandard** 或 **SurfaceOutputStandardSpecular** 结构体。其中，SurfaceOutputStandard 结构体用于默认的金属工作流程（Metallic

333

在本节，我们着重介绍一下 SurfaceOutput 结构体中的变量和含义。在表面函数中，我们需要根据 Input 结构体传递的各个变量计算表面属性。在 SurfaceOutput 结构体，这些表面属性包括了。

- fixed3 Albedo：对光源的反射率。通常由纹理采样和颜色属性的乘积计算而得。
- fixed3 Normal：表面法线方向。
- fixed3 Emission：自发光。Unity 通常会在片元着色器最后输出前（并在最后的顶点函数被调用前，如果定义了的话），使用类似下面的语句进行简单的颜色叠加：

```
c.rgb += o.Emission;
```

- half Specular：高光反射中的指数部分的系数，影响高光反射的计算。例如，如果使用了内置的 BlinnPhong 光照函数，它会使用如下语句计算高光反射的强度：

```
float spec = pow (nh, s.Specular*128.0) * s.Gloss;
```

- fixed Gloss：高光反射中的强度系数。和上面的 Specular 类似，计算公式见上面的代码。一般在包含了高光反射的光照模型里使用。
- fixed Alpha：透明通道。如果开启了透明度的话，会使用该值进行颜色混合。

==========

表面着色器流程

其实就是把顶点着色器和片元着色器封装了下

顶点着色器： 顶点修改函数

片元着色器： 表面函数 + 光照函数 + 最后颜色处理函数

顶点修改函数输出结构为v2f_surf，然后把v2f_surf 传给片元着色器的数据结构，这里先传给**表面函数**，表面函数根据关照模型生成对应的输出结构SurfaceOutput ，把SurfaceOutput 数据传到**光照函数**，光照函数计算最后的光照结果颜色值，最后在**最后的颜色修改函数**里对颜色进行最后修改，输出颜色到颜色缓冲区

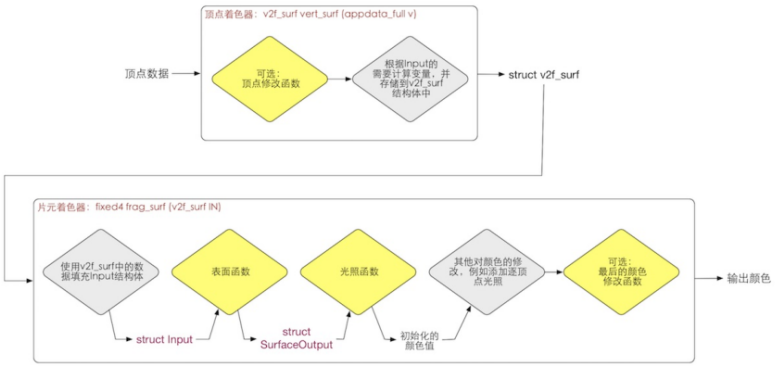


图17.3 表面着色器的渲染计算流水线。黄色：可以自定义的函数。灰色：Unity自动生成的计算步骤