第 18 章　基于物理的渲染

长期以来，图形学一直信奉着“如果这看起来是对的，那么这就是对的”的信条，这其中一方面的原因是由于当时计算机的计算能力较低，且硬件价格昂贵。随着计算机的处理能力越来越强，人们开始考虑使用更加复杂的算法来渲染更加真实的画面。在二十世纪八十年代左右，**基于物理的渲染技术（Physically Based Shading, PBS）**首次被引入图形学的正统研究中，学者们提出了使用光线追踪的方法来渲染全局光照，由此打开了精确渲染光线传播的大门。然而，当时的工业界更加依赖于使用各种 tricks 来得到希望的艺术风格，而并不需要非常精确的物理模拟。因此，这种基于物理的渲染方法实际上是在二十世纪九十年代末才开始逐渐为工业界所接受。之后，随着 Mental Ray、Arnold、RenderMan 等渲染器的出世，基于物理的蒙特卡洛渲染方法在（离线渲染）工业界取得了巨大成功。

而在实时渲染领域，人们也发现了这种基于物理的光照模型的巨大优势。在这之前，Lambert 光照模型、Phong 光照模型和 Blinn-Phong 光照模型等经验模型占据了主流。然而，这种不满足能量守恒的光照模型使得美术人员需要花费大量的时间在参数调节上。尤其是，美术人员往往好不容易为一个物体调节好了所有参数，使得它在当前的光照条件下看起来是满意的。然而，一旦光照环境发生了变化，这一切都得从头再来。因此，近年来游戏从业者开始着手把基于物理的光照模型应用于实时渲染中。

Unity 最早在 2012 年的《蝴蝶效应》（ 英文名：Butterfly Effect）的 demo 中大量使用了 PBS，并在 Unity 5 中正式将 PBS 引入到引擎渲染中。Unity 5 引入了一个名为 Standard Shader 的可在不同材质之间通用的着色器，而该着色器就是使用了基于物理的光照模型。需要注意的是，PBS 并不意味着渲染出来的画面一定是像照片一样真实的，例如，Pixar 和 Disney 尽管长期使用 PBS 渲染电影画面，但它们得到的风格是非常有特色的艺术风格。相信很多读者或多或少看到过使用 PBS 渲染出来的画面是多么的酷炫，并很想了解这背后的技术原理。如果你是一个程序员，可能有很大的冲动想要自己实现一个 PBS 渲染框架，但往往走到后面会发现有很多看不懂的名词以及一大堆与之相关的论文；如果你是一个美工人员，你可能会找到很多关于如何制作 PBS 中使用的纹理教程，但你大概也了解，想要使用 PBS 实现出色的渲染效果，并不是纹理+一个 Shader 这么简单的问题。

现在，我们有一个好消息和一个坏消息要告诉大家。先说好消息，Unity 5 引入的基于物理的渲染不需要我们过多地了解 PBS 是如何实现的，就能利用各种内置工具来实现一个不错的渲染效果。然而坏消息是，我们很难通过短短几万文字来非常详细地告诉读者这些渲染到底是如何实现的，因为这其中需要牵扯许多复杂的光照模型，如果要完全理解每一种模型的话，大概还要讲很多论文和其他参考文献。不过还有一个好消息是，我们相信读者在学完本章后可以了解 PBS 的基本原理，并可以实现一个包含了简单的基于物理的光照模型的 Shader。如果你对 PBS 有着浓厚的兴趣，想要尝试自己构建一个更加复杂的 PBS 渲染框架，可以在本章的扩展阅读部分找到许多非

常有价值的参考资料。

在本章中，我们首先会讲解 PBS 的基本原理，让读者了解它们与我们之前所学的渲染方式到底有哪些不同。接着，我们会按照 PBS 的相关公式来实现一个非常简单的基于物理渲染的 Shader。尽管本书的定位并不是“教你如何使用 Unity”，但我们决定花一点时间来告诉读者 Unity 5 引入的 Standard Shader 是如何工作的，以及如何在 Unity 5 中使用它和其他工具来渲染一个场景，我们希望通过这些内容来让读者明白 PBS 中的一些关键因素。从本书第一版出版到现在，我们已经看到了越来越多的手机游戏也开始使用 PBS 进行渲染，我们相信这是未来的发展趋势，希望本章可以为读者打开 PBS 的大门。

18.1 PBS 的理论和数学基础

在学习如何实现 PBS 之前，我们非常有必要来了解基于物理的渲染所基于的理论和数学基础。我们不会过多地牵扯一些论文资料，但如果在阅读过程中读者发现无法理解一些光照模型的实现原理，这可能意味着你需要阅读更多的参考文献。本节主要参考了 Naty Hoffman 在 SIGGRAPH 2013 上做的名为 **Background: Physics and Math of Shading** 的演讲[1]以及 Brent Burley 等人在 SIGGRAPH 2012 上做的名为 **Physically-Based Shading at Disney** 的演讲[2]。在 Disney 的演讲中，作者介绍了 Disney 是如何把各种现有的基于物理的光照模型与自身需求结合起来，得到满足 Disney 渲染需求的 BRDF 光照模型，并成功应用在电影《无敌破坏王》（英文名：Wreck-It Ralph）的材质渲染中。自此之后，这个 Disney BRDF 模型被广泛应用于实时渲染领域，各大游戏引擎在此基础上实现了各自的 PBS 光照模型，大大提升了游戏画面渲染质量。那么，到底什么叫基于物理的渲染？什么又是 BRDF 呢？这就是我们现在要来学习的内容。既然要得到基于物理的光照模型，我们首先必须了解在真实的物理世界中光是什么以及它是如何与各种材质发生交互的。

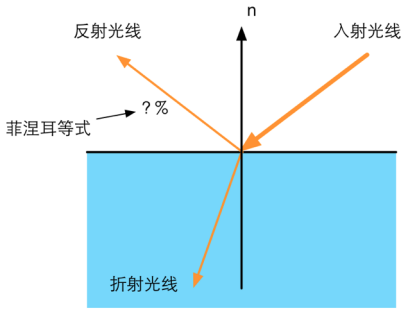
18.1.1　光是什么

尽管我们之前一直讲光照模型，但要问读者，光到底是什么，可能没有多少人可以解释清楚。在物理学中，光是一种电磁波。首先，光由太阳或其他光源中被发射出来，然后与场景中的对象相交，一些光线被**吸收（absorption）**，而另一些则被**散射（scattering）**，最后光线被一个感应器（例如我们的眼睛）吸收成像。

通过上面的过程，我们知道材质和光线相交会发生两种物理现象：散射和吸收（其实还有自发光现象）。光线会被吸收是由于光被转化成了其他能量，但吸收并不会改变光的传播方向。相反的，散射则不会改变光的能量，但会改变它的传播方向。在光的传播过程中，影响光的一个重要的特性是材质的**折射率（refractive index）**。我们知道，在均匀的介质中，光是沿直线传播的。但如果光在传播时介质的折射率发生了变化，光的传播方向就会发生变化。特别是，如果折射率是突变的，就会发生光的散射现象。

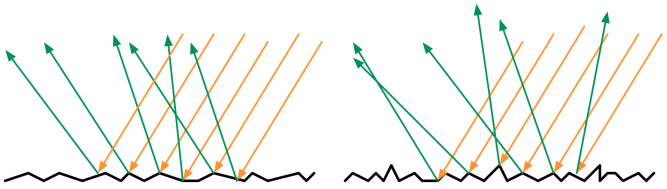
实际上，在现实生活中，光和物体之间的交互过程是非常复杂的，大多数情况下并不存在一种可分析的解决方法。但为了在渲染中对光照进行建模，我们往往只考虑一种特殊情况，即只考虑两个介质的边界是无限大并且是光学平滑（optically flat）的。尽管真实物体的表面并不是无限延伸的，也不是绝对光滑的，但和光的波长相比，它们的大小可以被近似认为是无限大以及光学平滑的。在这样的前提下，光在不同介质的边界会被分割成两个方向：反射方向和折射方向。而有多少百分比的光会被反射（另一部分就是被折射了）则是由**菲涅耳等式（Fresnel equations）**来描述的，如图 18.1 所示。

但是，这些与光线的交界处真的是像镜子一样平坦吗？尽管在上面我们已经说过，相对于光的波长来说，它们的确可以被认为是光学平坦的。但是，如果想象我们有一个高倍放大镜，去放大这些被照亮的物体表面，就会发现有很多之前肉眼不可见的凹凸不平的平面。在这种情况下，物体的表面和光照发生的各种行为，更像是一系列微小的光学平滑平面和光交互的结果，其中每个小平面会把光分割成不同的方向。



▲图 18.1　在理想的边界处，折射率的突变会把光线分成两个方向

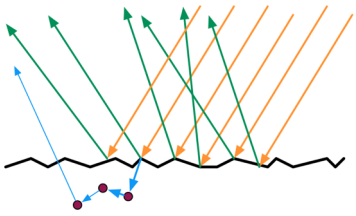
这种建立在微表面的模型更容易解释为什么有些物体看起来粗糙，而有些看起来就平滑，如图 18.2 所示。想象我们用一个放大镜去观察一个光滑物体的表面，尽管它的表面仍然由许多凹凸不平的微表面构成，但这些微表面的法线方向变化角度小，因此，由这些表面反射的光线方向变化也比较小，如图 18.2 左图所示，这使得物体的高光反射更加清晰。而图 18.2 右图所示的粗糙表面则相反，由此得到的高光反射效果更模糊。



▲图 18.2　左边：光滑表面的微平面的法线变化较小，反射光线的方向变化也更小。右边：粗糙表面的微平面的法线变化较大，反射光线的方向变化也更大

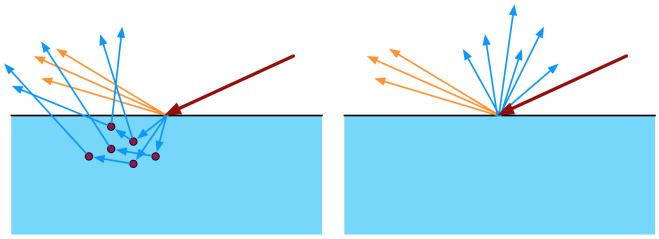
在上面的内容中，我们并没有讨论那些被微表面折射的光。这些光被折射到物体的内部，一

部分被介质吸收，一部分又被散射到外部。金属材质具有很高的吸收系数，因此，所有被折射的光往往会被立刻吸收，被金属内部的自由电子转化成其他形式的能量。而非金属材质则会同时表现出吸收和散射两种现象，这些被散射出去的光又被称为**次表面散射光（subsurface-scattered light）**。在图 18.3 中，我们给出了一条由微表面折射的光的传播路径（如图 18.3 所示的蓝线，读者可参考随书资源中的彩图）。



▲图 18.3 微表面对光的折射。这些被折射的光中一部分被吸收，一部分又被散射到外部

现在，我们把放大镜从物体表面拿开，继续从渲染的层级大小上考虑光与表面一点的交互行为。那么，由微表面反射的光可以被认为是该点上一些方向变化不大的反射光，如图 18.4 中的黄线所示（可参考随书资源中的彩图）。而折射光线（蓝线）则需要更多的考虑。那些次表面散射光会从不同于入射点的位置从物体内部再次射出，如图 18.4 左图所示。而这些离入射点的距离值和像素大小之间的关系会产生两种建模结果。如果像素要大于这些散射距离的话，意味着这些次表面散射产生的距离可以被忽略，那我们的渲染就可以在局部进行，如图 18.4 右图所示。如果像素要小于这些散射距离，我们就不可以选择忽略它们了，要实现更真实的次表面散射效果，我们需要使用特殊的渲染模型，也就是所谓的**次表面散射渲染技术**。



▲图 18.4 次表面散射。左边：次表面散射的光线会从不同于入射点的位置射出。如果这些距离值小于需要被着色的像素大小，那么渲染就可以完全在局部完成（右边）。否则，就需要使用次表面散射渲染技术

我们下面的内容均建立在不考虑次表面散射的距离，而完全使用局部着色渲染的前提下。

18.1.2 渲染方程

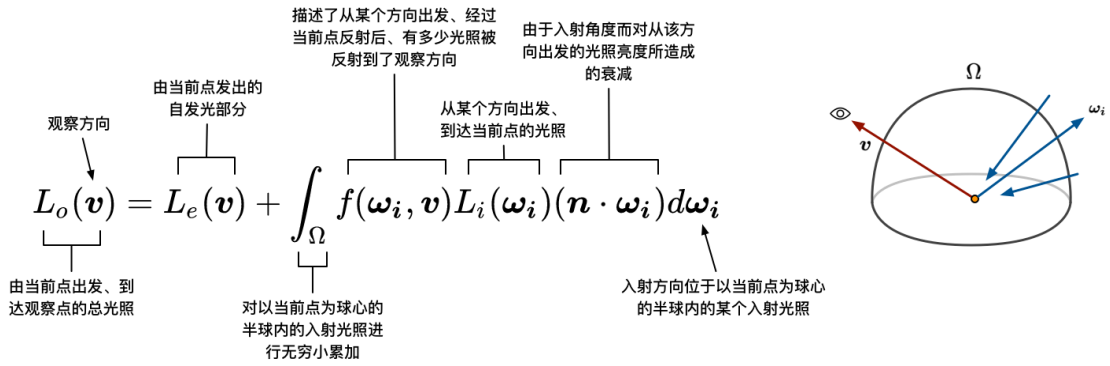
在了解了上面的理论基础后，我们现在来学习如何用数学表达式来表示上面的光照模型。这意味着，我们要对光这个看似抽象的概念进行量化。我们可以用**辐射率（radiance）**来量化光。辐射率是单位面积、单位方向上光源的辐射通量，通常用 $\boldsymbol{L}$ 来表示，被认为是对单一光线的亮度和颜色评估。在渲染中，我们通常会使用入射光线的入射辐射率 $\boldsymbol{L_i}$ 来计算出射辐射率 $\boldsymbol{L_o}$，这个过程也往往被称为是**着色（shading）**过程。

James Kajiya 在他 1986 年那篇著名的 ***The Rendering Equation*** 论文中给出了**渲染方程**的表述形式，由此在理论上完美地给出了基于物理渲染的方向。这个公式第一次从数学建模的角度描述了渲染到底是在解决什么问题，可以说自此之前人们都是在靠各种各样的 hacks 来尝试进行各种渲染，当渲染方程横空出世后，人们终于不用再盲人摸象，人们有了统一的目标，那就是如何更快更好地去求解渲染方程。

这个在图形学中大名鼎鼎的渲染方程公式如下：

$$L_o(\boldsymbol{v}) = L_e(\boldsymbol{v}) + \int_{\Omega} f(\boldsymbol{\omega_i}, \boldsymbol{v}) L_i(\boldsymbol{\omega_i})(\boldsymbol{n} \cdot \boldsymbol{\omega_i}) d\boldsymbol{\omega_i}$$

尽管上面的式子看起来有些复杂，但很好理解，即给定观察视角 $\boldsymbol{v}$，该方向上的出射辐射率 $L_o(\boldsymbol{v})$等于该点向观察方向发出的自发光辐射率 $L_e(\boldsymbol{v})$加上所有有效的入射光 $L_i(\boldsymbol{\omega_i})$到达观察点的辐射率积分和。图 18.5 给出了渲染等式中各个部分的通俗解释。



▲图 18.5　渲染方程参数的通俗解释

渲染方程是计算机图形学的核心公式，当去掉其中的自发光项 $L_e(\boldsymbol{v})$后，剩余的部分就是著名的**反射等式（Reflectance Equation）**。我们可以这样理解反射等式：想象我们现在要计算表面上某点的出射辐射率，我们已知到该点的观察方向，该点的出射辐射率是由从许多不同方向的入射辐射率叠加后的结果。其中，$f(\boldsymbol{\omega_i}, \boldsymbol{v})$表示了不同方向的入射光在该观察方向上的权重分布。我们把这些不同方向的光辐射率（$L_i(\boldsymbol{\omega_i})$部分）乘以观察方向上所占的权重（$f(\boldsymbol{\omega_i}, \boldsymbol{v})$部分），再乘以它们在该表面的投影结果（$(\boldsymbol{n} \cdot \boldsymbol{\omega_i})$部分），最后再把这些值加起来（即做积分）就是最后的出射辐射率。

在实时渲染中，自发光项通常就是直接加上某个自发光值。除此之外，积分累加部分在实时渲染中也基本无法实现，因此积分部分通常会被若干精确光源的叠加所代替，而不需要计算所有入射光线在半球面上的积分。

18.1.3 精确光源

在真实的物理世界中，所有的光源都是有面积概念的，即所谓的面光源。由于面光源的光照计算通常要耗费大量的时间，因此在实时渲染中，我们通常会使用**精确光源（punctual light sources）**来近似模拟这些面光源。图形学中常见的精确光源类型有点光源、平行光和聚光灯等，这些精确光源被认为是大小为无限小且方向确定的，尽管这并不符合真实的物理定义，但它们在大多数情况下都能得到令人满意的渲染效果。

我们使用 $\boldsymbol{l_c}$ 来表示它的方向，使用 $\boldsymbol{c}_{light}$ 表示它的颜色。使用精确光源的最大的好处在于，我们可以大大简化上面的反射等式。我们在这里省略推导过程（有兴趣的读者可以阅读参考文献[1]），直接给出结论，即对于一个精确光源，我们可以使用下面的等式来计算它在某个观察方向 $\boldsymbol{v}$ 上的出射辐射率：

$$L_o(\boldsymbol{v}) = \pi f(\boldsymbol{l_c}, \boldsymbol{v}) \boldsymbol{c}_{light} (\boldsymbol{n} \cdot \boldsymbol{l_c})$$

和之前使用积分形式的原始公式相比，上面的式子使用一个特定的方向的$f(\boldsymbol{l}_c, \boldsymbol{v})$值来代替积分操作，这大大简化了计算。如果场景中包含了多个精确光源，我们可以把它们分别代入上面的式子进行计算，然后把它们的结果相加即可。也就是说，反射等式可以简化成下面的形式：

$$L_o(\boldsymbol{v}) = \sum_{i=0}^{n} L_o^i(\boldsymbol{v}) = \sum_{i=0}^{n} \pi f(\boldsymbol{l}_c^i, \boldsymbol{v}) \boldsymbol{c}_{light}(\boldsymbol{n} \cdot \boldsymbol{l}_c^i)$$

那么，现在剩下的问题就是，$f(\boldsymbol{l}_c, \boldsymbol{v})$项怎么算呢？$f(\boldsymbol{l}_c, \boldsymbol{v})$实际上描述了当前点是如何与入射光线进行交互的：当给定某个入射方向的入射光后，有多少百分比的光照被反射到了观察方向上。在图形学中，这一项有一个专门的名字，那就是双向反射分布函数，即 BRDF。

18.1.3 双向反射分布函数（BRDF）

BRDF（Bidirectional Reflectance Distribution Function，中文名称为双向反射分布函数）定量描述了物体表面一点是如何和光进行交互的。大多数情况下，BRDF 可以用$f(\boldsymbol{l}, \boldsymbol{v})$来表示，其中$\boldsymbol{l}$为入射方向和$\boldsymbol{v}$为观察方向（双向的含义）。这种情况下，绕着表面法线旋转入射方向或观察方向并不会影响 BRDF 的结果，这种 BRDF 被称为是**各项同性（isotropic）**的 BRDF。与之对应的则是**各向异性（anisotropic）**的 BRDF。

那么，BRDF 到底表示的含义是什么呢？BRDF 有两种理解方式——第一种理解是，当给定入射角度后，BRDF 可以给出所有出射方向上的反射和散射光线的相对分布情况；第二种理解是，当给定观察方向（即出射方向）后，BRDF 可以给出从所有入射方向到该出射方向的光线分布。一个更直观的理解是，当一束光线沿着入射方向$\boldsymbol{l}$到达表面某点时，$f(\boldsymbol{l}, \boldsymbol{v})$表示了有多少部分的能量被反射到了观察方向$\boldsymbol{v}$上。

下面，我们来看一下反射等式中的重要组成部分——BRDF 是如何计算的。可以看出，BRDF 决定了着色过程是否是基于物理的。这可以由 BRDF 是否满足两个特性来判断：它是否满足**交换律（reciprocity）**和**能量守恒（energy conservation）**。

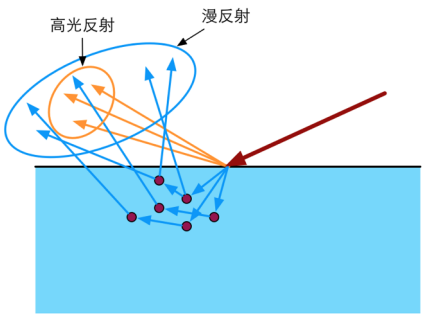
交换律要求当交换$\boldsymbol{l}$和$\boldsymbol{v}$的值后，BRDF 的值不变，即

$$f(\boldsymbol{l}, \boldsymbol{v}) = f(\boldsymbol{v}, \boldsymbol{l})$$

而能量守恒则要求表面反射的能量不能超过入射的光能，即

$$\forall \boldsymbol{l}, \int_{\Omega} f(\boldsymbol{l}, \boldsymbol{v})(\boldsymbol{n} \cdot \boldsymbol{l}) d\boldsymbol{\omega}_{\boldsymbol{o}} \leq 1$$

基于这些理论，BRDF 可以用于描述两种不同的物理现象：表面反射和次表面散射。针对每种现象，BRDF 通常会包含一个单独的部分来描述它们——用于描述表面反射的部分被称为**高光反射项（specular term）**，以及用于描述次表面散射的**漫反射项（diffuse term）**，如图 18.6 所示。



▲图 18.6 BRDF 描述的两种现象。高光反射部分

那么，我们如何得到不同材质的 BRDF 呢？一种完全真实的方法是使用精确的光学仪器在真实的物理世界中对这些材质进行测量，通过不断改变光照入射方向和观察方向并对当前材质的反射光照进行采样，我们就可以得到它的 BRDF 图像切片。一些机构和组织向公众公开了他们测量出来的 BRDF 数据库，以便供研究人员进行分析和研究，

例如 MERL BRDF 数据库（http://www.merl.com/brdf/）以及 MIT CSAIL 数据库（http://people.csail.mit.edu/addy/research/brdf/），这些真实材质的 BRDF 数据反映了它们在不同光照和观察角度下的反射情况。在学术界，学者们也基于复杂的物理和光学理论分析归纳出了一些通用的 BRDF 数学模型，来对 BRDF 分析模型中的**高光反射项**和**漫反射项**进行数学建模。这些由研究人员得到的 BRDF 模型也可以被称为是分析型 BRDF 模型，这些 BRDF 模型通常包含了大量的物理和光学参数。通过对真实材质的 BRDF 图像和现有的分析型 BRDF 模型进行对比，研究人员发现，其实很多材质是无法被现有的任何一种分析型 BRDF 模型所良好的描述出来。因此，许多最新的研究都选择使用基于数据驱动的方法，来开发出一些新的分析型 BRDF 模型，从而使其可以和真实材质的 BRDF 图像尽可能地接近。但遗憾的是，由于真实世界的光照和材质都非常复杂，因此很难有一种可以满足所有真实材质特性的 BRDF 模型。Disney 向公众开源了一个名为 **BRDF Explorer**（http://github.com/wdas/brdf）的软件，来让用户可以直观地对比各种分析型 BRDF 模型与真实测量得到的 BRDF 值之间的差异。

可以认为，这些测量得到的真实 BRDF 数据库是研究人员在开发新的基于物理的 BRDF 光照模型的重要依据，除了满足交换律和能量守恒两个条件外，一个分析型 BRDF 模型应当与测量得到的 BRDF 数据在尽可能多得材质范围内（或这些特定的材质）、具有尽可能得相似的表现才可能被广泛应用。Disney 通过对大量现有的分析型 BRDF 模型进行对比，并结合对真实材质的 BRDF 数据的观察，开发出了适用于 Disney 动画渲染流程的 BRDF 模型，也被称为 Disney BRDF。在下面的内容中，我们会介绍**漫反射项**和**高光反射项**的一些常见的 BRDF 数学模型，同时给出 Disney BRDF 模型中的相应表示。

18.1.4　漫反射项

我们之前所学习的 Lambert 模型就是最简单、也是应用最广泛的漫反射 BRDF。准确的 Lambertian BRDF 的表示为：

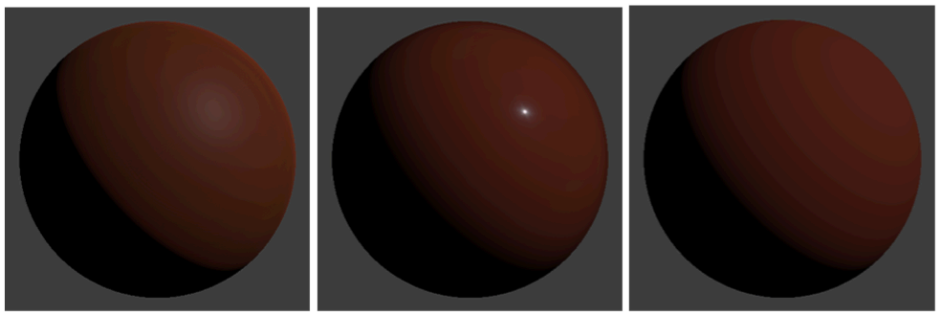
$$f_{Lambert}(\boldsymbol{l},\boldsymbol{v}) = \frac{\boldsymbol{c}_{diff}}{\pi}$$

其中，$\boldsymbol{c}_{diff}$表示漫反射光线所占的比例，它也通常被称为是漫反射颜色（diffuse color）。与我们之前讲过的 Lambert 光照模型不太一样的是，上面的式子实际上是一个定值，我们常见到的余弦因子部分（即$(\boldsymbol{n}\cdot\boldsymbol{l})$）实际是反射等式的一部分，而不是 BRDF 的部分。上面的式子之所以要除以π，是因为我们假设漫反射在所有方向上的强度都是相同的，而 BRDF 要求在半球内的积分值为 1。因此，给定入射方向 $\boldsymbol{l}$ 的光源在表面某点的出射漫反射辐射率为：

$$L_o(\boldsymbol{v}) = \pi f_{Lambert}(\boldsymbol{l},\boldsymbol{v})\boldsymbol{c}_{light}(\boldsymbol{n}\cdot\boldsymbol{l}) = \boldsymbol{c}_{diff} \times \boldsymbol{c}_{light}(\boldsymbol{n}\cdot\boldsymbol{l})$$

可以看出，最右项和我们之前一直使用的漫反射项基本一样。尽管 Lambert 模型简单且易于实现，但真实世界中很少有材质符合上述 Lambert 的数学描述,即具有完美均匀的散射。不过，一些游戏引擎和实时渲染器出于性能的考虑会使用 Lambert 模型作为它们 PBS 模型中的漫反射项，例如虚幻引擎 4（Unreal Engine 4）[11]。

通过对真实材质的 BRDF 数据进行分析，研究人员发现许多材质在掠射角度表现出了明显的高光反射峰值，而且还与表面的粗糙度有着强烈的联系。粗糙表面在掠射角容易形成一条亮边，而相反地光滑表面则容易在掠射角形成一条阴影边。这些都是 Lambert 模型所无法描述的。图 18.7 显示了这样的例子，注意图中在掠射角的光照效果。



▲图 18.7 从左到右：粗糙材质和光滑材质的真实漫反射结果，以及 Lambert 漫反射结果

因此，许多基于物理的渲染选择使用更加复杂的漫反射项来模拟更加真实次表面散射的结果。例如，在 Disney BRDF[2]中，它的漫反射项为：

$$f_{diff}(\boldsymbol{l},\boldsymbol{v}) = \frac{baseColor}{\pi}(1+(F_{D90}-1)(1-\boldsymbol{n}\cdot\boldsymbol{l})^5)(1+(F_{D90}-1)(1-\boldsymbol{n}\cdot\boldsymbol{v})^5)$$

$$\text{其中，}F_{D90} = 0.5 + 2roughness(\boldsymbol{h}\cdot\boldsymbol{l})^2$$

其中，*baseColor* 是表面颜色，通常由纹理采样得到，*roughness* 是表面的粗糙度。上面的漫反射项既考虑了在掠射角漫反射项的能量变化，还考虑了表面的粗糙度对漫反射的影响。Disney 使用了 Schlick 菲涅耳近似等式[7]来模拟在掠射角的反射变化，同时使用表面粗糙度来进一步修改它，这使得光滑材质可以在掠射角具有更为明显的阴影边，而又使得粗糙材质在掠射角具有亮边。而上面的式子也正是 Unity 5 内部使用的漫反射项。

18.1.5 高光反射项

在现实生活中，几乎所有的物体都或多或少有高光反射现象。John Hable 在他的文章中就强调了 **Everything is Shiny**。但在许多传统的 Shader 中，很多材质只考虑了漫反射效果，而并没有添加高光反射，这使得渲染出来的画面并不那么真实可信。在基于物理的渲染中，BDRF 中的高光反射项大多数都是建立在**微面元理论（microfacet theory）**的假设上的。微面元理论认为，物体表面实际是由许多人眼看不到的微面元组成的，虽然物体表面并不是光学平滑的，但这些微面元可以被认为是光学平滑的，也就是说它们具有完美的高光反射。当光线和物体表面一点相交时，实际上是和一系列微面元交互的结果。正如我们在 18.1.1 节中看到的，当光和这些微面元相交时，光线会被分割成两个方向——反射方向和折射方向。这里我们只需要考虑被反射的光线，而折射光线已经在之前的漫反射项中考虑过了。当然，微面元理论也仅仅是真实世界的散射的一种近似理论，它也有自身的缺陷，仍然有一些材质是无法使用微面元理论来描述的。

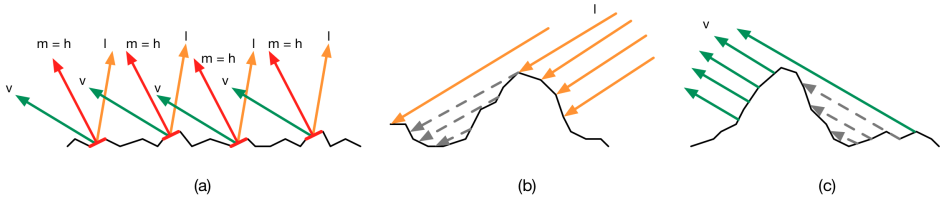
假设表面法线为 $\boldsymbol{n}$，这些微面元的法线 $\boldsymbol{m}$ 并不都等于 $\boldsymbol{n}$，因此，不同的微面元会把同一入射方向的光线反射到不同的方向上。而当我们计算 BRDF 时，入射方向 $\boldsymbol{l}$ 和观察方向 $\boldsymbol{v}$ 都会被给定，这意味着只有一部分微面元反射的光线才会进入到我们的眼睛中，这部分微面元会恰好把光线反射到方向 $\boldsymbol{v}$ 上，即它们的法线 $\boldsymbol{m}$ 等于 $\boldsymbol{l}$ 和 $\boldsymbol{v}$ 的一半，也就是我们一直看到的半角度矢量 $\boldsymbol{h}$（half-angle vector，也被称为 half vector），如图 18.6（a）所示。

然而，这些 $\boldsymbol{m} = \boldsymbol{h}$ 的微面元反射也并不会全部添加到 BRDF 的计算中。这是因为，它们其中一部分会在入射方向 $\boldsymbol{l}$ 上被其他微面元挡住（shadowing），如图 18.6（b）所示，或是在它们的反

射方向 $\boldsymbol{v}$ 上被其他微面元挡住了（masking），如图 18.6（c）所示。微面元理论认为，所有这些被遮挡住的微面元不会添加到高光反射项的计算中（实际上它们中的一些由于多次反射仍然会被我们看到，但这不在微面元理论的考虑范围内）。

基于微面元理论的这些假设，BRDF 的高光反射项可以用下面的通用形式来表示：

$$f_{spec}(\boldsymbol{l},\boldsymbol{v}) = \frac{F(\boldsymbol{l},\boldsymbol{h})G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})D(\boldsymbol{h})}{4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})}$$



▲图 18.6（a）　那些 $\boldsymbol{m}=\boldsymbol{h}$ 的微面元会恰好把入射光从 $\boldsymbol{l}$ 反射到 $\boldsymbol{v}$ 上，只有这部分微面元才可以添加到 BRDF 的计算中。（b）一部分满足（a）的微面元会在 $\boldsymbol{l}$ 方向上被其他微面元遮挡住，它们不会接受到光照，因此会形成阴影。（c）还有一部分满足（a）的微面元会在反射方向 $\boldsymbol{v}$ 上被其他微面元挡住，因此，这部分反射光也不会被看到

这就是著名的 Torrance-Sparrow 微面元模型[5]（Torrance 和 Sparrow 来源于两个作者的姓名）。上面的式子看起来难以理解，实际上其中的各个项对应了我们之前讲到的不同现象。$D(\boldsymbol{h})$是微面元的**法线分布函数（normal distribution function，NDF）**，它用于计算有多少比例的微面元的法线满足 $\boldsymbol{m}=\boldsymbol{h}$，只有这部分微面元才会把光线从 $\boldsymbol{l}$ 方向反射到 $\boldsymbol{v}$ 上。$G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$是**阴影—遮掩函数（shadowing-masking function）**，它用于计算那些满足 $\boldsymbol{m}=\boldsymbol{h}$ 的微面元中有多少会由于遮挡而不会被人眼看到，因此它给出了活跃的微面元（active microfacets）所占的浓度，只有活跃的微面元才会成功地把光线反射到观察方向上。$F(\boldsymbol{l},\boldsymbol{h})$则是这些活跃微面元的**菲涅尔反射（Fresnel reflectance）**函数，它可以告诉我们每个活跃的微面元会把多少入射光线反射到观察方向上，即表示了反射光线占入射光线的比率。事实上，现实生活中几乎所有的物体都会表现出菲涅耳现象。最后，分母$4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})$是用于校正从微面元的局部空间到整体宏观表面数量差异的校正因子。

这些不同的部分又可以衍生出很多不同的 BRDF 模型。首先是菲涅耳反射函数部分 $F(\boldsymbol{l},\boldsymbol{h})$。

18.1.5.1 菲涅耳反射函数

菲涅耳反射函数计算了光学表面反射光线所占的部分，它表明了当光照方向和观察方向夹角逐渐增大时高光反射强度增大的现象。完整的菲涅耳等式非常复杂，包含了诸如复杂的折射率等与材质相关的参数。为了给美术人员提供更加直观且方便调节的参数，大多数 PBS 实现选择使用 Schlick 菲涅耳近似等式[7]来得到近似的菲涅尔反射效果：

$$F_{Schlick}(\boldsymbol{l},\boldsymbol{h}) = \boldsymbol{c}_{spec} + (1-\boldsymbol{c}_{spec})(1-(\boldsymbol{l}\cdot\boldsymbol{h}))^5$$

其中，$\boldsymbol{c}_{spec}$ 是材质的高光反射颜色。通过对真实世界材质的观察，人们发现金属材质的高光反射颜色值往往比较大，而非金属材质的反射颜色值则往往较小。

18.1.5.2 法线分布函数

法线分布函数 $D(\boldsymbol{h})$表示了对于当前表面来说有多少比例的微面元的法线满足 $\boldsymbol{m}=\boldsymbol{h}$，这意味着只有这些微面元才会把光线从 $\boldsymbol{l}$ 方向反射到 $\boldsymbol{v}$ 上。对于大多数表面来说，微面元的法线朝向并不是均匀分布的，更多的微面元会具有和表面法线 $\boldsymbol{n}$ 相同的面法线。法线分布函数的值必须是非负

的标量值，它决定了高光区域的大小、亮度和形状，因此是高光反射项中非常重要的一项。一个直观的感受的，当表面的粗糙度下降时，应该有更多的微面元的面法线满足 $\boldsymbol{m}=\boldsymbol{n}$。，因此法线分布函数应该考虑到表面粗糙度的影响。

我们之前学习的 Blinn-Phong 模型[7]就是一种非常简单的模型。Blinn 在他的论文中改进了 Phong 模型并提出了 Blinn-Phong 模型，使它更贴合微面元 BRDF 模型的理论。Blinn-Phong 模型使用的法线分布函数 ***D(h)***为：

$$D_{blinn}(\boldsymbol{h}) = \frac{gloss+2}{2\pi}(\boldsymbol{n}\cdot\boldsymbol{h})^{gloss}$$

其中，*gloss* 是与表面粗糙度相关的参数，它的值可以是任意非负数。上面的式子和我们之前所见的 Blinn-Phong 模型有所不同，这是因为我们在里面加入了归一化因子，这是因为法线分布函数必须满足一个条件，即所有微面元的投影面积必须等于该区域宏观表面的投影面积。因此，上述公式也被称为是归一化的 Phong 法线分布函数。

但实际上，Blinn-Phong 模型并不能真实地反映很多真实世界中物体的微面元法线方向分布，它其实完全是一种经验型模型，因此，很多更加复杂的分布函数被提了出来，例如 GGX[3]、Beckmann[4]等。Beckmann 分布来源于高斯粗糙分布的一种假设，而且在表现上和 Phong 分布非常类似，但它的计算却要复杂很多。 GGX 分布（也被称为 Trowbridge-Reitz 法线分布函数）是一种更新的法线分布函数，它的公式如下：

$$D_{GGX}(\boldsymbol{h}) = \frac{\alpha^2}{\pi((\alpha^2-1)(\boldsymbol{n}\cdot\boldsymbol{h})^2+1)^2}$$

其中，参数 α 是与表面粗糙度相关的参数。与 Blinn-Phong 的法线分布相比，GGX 分布具有更明亮、更狭窄且拖尾更长的高光区域，它的结果更接近于一些测量得到的真实材质的 BRDF 分布。在 Disney BRDF 中，Disney 认为对于很多材质来说，GGX 表现出来的高光拖尾仍然不够长。他们选择使用一种更加广义的法线分布模型，即 Generalized-Trowbridge-Reitz（GTR）分布。GTR 分布于 GGX 分布很类似，但它的分母部分的指数不是 2，而是一个可调参数。Disney 使用两个不同指数的 GTR 分布作为两个高光反射片，其中第一个反射片用于表示基本材质层，第二个反射片用于表示基本材质表面的清漆层。除此之外，他们还发现令 $\alpha = roughness^2$ 可以在材质粗糙度上得到更加线性的变化。否则，直接使用 *roughness* 作为参数的话会导致在光滑材质和粗糙材质之间插值出来的材质总是偏粗糙的。

18.1.5.2 阴影-遮挡函数

阴影-遮挡函数 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$也被称为**几何函数（geometry function）**，它表明了具有给定面法线 ***m*** 的微面元在沿着入射方向 ***l*** 和观察方向 ***v*** 上不会被其他微面元挡住的概率。在微面元理论的 BRDF 中，***m*** 可以使用半向量 ***h*** 来代替，因为只有这部分微面元才会把光线从 ***l*** 方向反射到 ***v*** 上。由于 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$表示的是一个概率值，因此它的值是一个范围在 0 到 1 之间的标量。学术界发表了许多对于 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$的分析模型，这些公式大多建立在一些简化的表面模型基础下。许多已发表的微面元 BRDF 模型习惯把 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$和高光反射项的分母$(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})$部分结合起来，即把 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$除以$(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})$的部分合在一起讨论，这是因为这两个部分都和微面元的可见性有关，因此 Naty Hoffman 在他的演讲[1]中称这个合项为**可见性项（visibility term）**。

一些 BRDF 模型选择完全省略可见性项，即把该项的值设为 1。这意味着，这些 BRDF 中的 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$表达式等同于：

$$G_{implicit}(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})=(\boldsymbol{n}\cdot\boldsymbol{l}_c)(\boldsymbol{n}\cdot\boldsymbol{v})$$

上述的 $\boldsymbol{G}_{implicit}$ 实现不需要任何计算量（因为可以直接和高光反射项的分母进行抵消），并且在一些程度上可以反映正确的变化趋势。例如，当从掠射角进行观察或光线从掠射角射入时，该项会趋近于 0，这是符合我们的认知的，因为在掠射角时微面元被其他微面元遮挡的概率会非常大。然而，这种 $\boldsymbol{G}_{implicit}$ 的实现忽略了材质粗糙度的影响，缺乏一定的物理真实性，因为我们希望粗糙的表面具有更高阴影和遮挡概率。

通常，阴影-遮挡函数 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$依赖于法线分布函数 $D(\boldsymbol{h})$，因为它需要结合 $D(\boldsymbol{h})$来保持 BRDF 能量守恒的规定。最早的阴影-遮挡函数之一是 Cook-Torrance 阴影遮挡函数[9]（Cook 和 Torrance 来源于两个作者的姓名）：

$$G_{ct}(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})=\min(1,\frac{2(\boldsymbol{n}\cdot\boldsymbol{h})(\boldsymbol{n}\cdot\boldsymbol{v})}{(\boldsymbol{v}\cdot\boldsymbol{h})},\frac{2(\boldsymbol{n}\cdot\boldsymbol{h})(\boldsymbol{n}\cdot\boldsymbol{l})}{(\boldsymbol{v}\cdot\boldsymbol{h})})$$

Cook-Torrance 阴影遮挡函数在电影行业被应用了很长时间，但它实际上是基于一个非真实的微几何模型，而且同样不受材质粗糙度的影响。后来，Kelemen 等人[10]提出了一个对于 Cook-Torrance 阴影遮挡函数非常快速且有效的近似实现：

$$\frac{G_{ct}(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})}{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})}\approx\frac{1}{(\boldsymbol{l}\cdot\boldsymbol{h})^2}$$

目前在图形学中广受推崇的是 Smith 阴影-遮掩函数[6]。Smith 函数比 Cook-Torrance 函数更加精确，而且考虑进了表面粗糙度和法线分布的影响。原始的 Smith 函数是为 Beckmann 法线分布函数所涉及的，而 Walter 等人[3]随后将其通用化，使其可以匹配任何法线分布函数，并给出了针对 Beckmann 和 GGX 法线分布函数的更加高效的近似 Smith 模型。在 Disney 的 BRDF 模型[2]中，它的阴影-遮掩函数 $G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})$就使用了 Walter 等人[3]提出的为 GGX 设计的 Smith 模型：

$$G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})=\frac{2}{1+\sqrt{1+\alpha_g^2\tan\theta_{\boldsymbol{v}}{}^2}}$$

其中，$\alpha_g=(0.5+\frac{roughness}{2})^2$

上述公式中的 $\theta_{\boldsymbol{v}}$表示观察方向 $\boldsymbol{v}$ 和表面法线 $\boldsymbol{n}$ 之间的夹角。根据艺术家的反馈以及对测量得到的 BRDF 图像的观察，Disney 在上述式子中重新映射了 α_g 和 *roughness* 之间的关系，由此得到了一个在视觉上让艺术家更加满意的效果。这种 Smith 阴影-遮掩函数被广泛应用在电影行业中，但可以看出，相比于 Kelemen 改进后的 Cook-Torrance 阴影遮挡函数，Smith 函数的计算量要明显高很多。

至此，我们已经介绍完了基于物理的 BRDF 模型的基础理论，并给出了一些常见的 BRDF 模型，例如 Phong、Beckmann、GGX 模型以及 Disney BRDF 模型的实现等。尽管存在很多基于物理的 BRDF 模型，但在真实的电影或游戏制作中，我们希望在直观性和物理可信度之间找到一个平衡点，使得实现的 BRDF 既可以让美术人员直观地调节各个参数，而又有一定的物理可信度。当然，有时候为了满足直观性我们不得不牺牲一定的物理特性，得到的 BRDF 可能不是严格基于物理原理的。Disney 的 BRDF 模型就是一个很好的例子，它使用尽可能少的若干直观参数来代替那些晦涩难懂的物理变量，而且这些参数的范围被精妙地设计为 0 到 1，为此 Disney 会在必要时

重新映射 BRDF 中的变量范围，例如在阴影-遮掩函数中他们重新映射了粗糙度变量的范围。

18.1.6 PBS 中的光照

尽管基于物理渲染的理论比较复杂，但在实际应用中绝大部分情况下我们其实只需要按照上面提到的各种公式来实现相应的 BRDF 模型即可。然而，要想得到画面出色的渲染效果，仅仅应用这些公式是远远不够的，我们还需要为这些 PBS 材质搭配以出色的光照。

在上面的内容中我们已经介绍了精确光源。随着新的技术不断被提出，实时面光源也不再是一个奢侈的梦想。在 SIGGRAPH 2016 上，Eric Heitz 和 Stephen Hill 在他们名为 **Real-Time Area Lighting: a Journey from Research to Production** 的演讲中就分享了如何实现实时面光源的渲染，这也是 Unity 的 **Adam Demo** 中使用的技术，读者可以在 Unity Labs 的相关文章[①]中找到相关内容和源码实现。除了上述两种光源外，**基于图像的光照（image-based lighting, IBL）**同样是非常重要的光照来源。

基于图像的光照通常指的是把场景中远处的光照存储在类似环境贴图的图像中。这些环境贴图可以表示光滑物体表面反射的环境光，从而允许我们可以快速得到拥有很高细节的真实光照效果。在 Unity 中，这种光照通常是由反射探针（Reflection Probes）机制来实现的，我们可以在 Shader 中获取当前物体所在的反射探针并在需要时对它们的采样结果进行混合。当然，我们也可以实现一套自己的 IBL 机制，Sébastien Lagarde 在他的博文[②]里详细介绍了 IBL 的一些实现方法以及如何得到视差正确的局部环境贴图的方法，非常值得一看。

在实际开发过程中，如何放置各种光源以及调整它们和材质的参数绝对是一门学问，对场景美术人员和灯光师来说也是一项挑战。而采用 PBS 的好处之一就是，我们暴露给美术人员的材质编辑页面可以是统一的，而不再是各种令人眼花缭乱的光照模型的“大杂烩”。

18.1.7 Unity 中的 PBS 实现

在下面的内容中，我们将给出 Unity 中 PBS 的实现。需要注意的是，随着 Unity 的不断更新，其选择使用的 BRDF 模型也可能会发生变化，例如在 Unity 5.3 之前，Unity 的 PBS 中的法线分布函数 $D(\boldsymbol{h})$采用的是我们之前提到的归一化的 Phong 法线分布函数，而在 Unity 5.3 及其之后的版本（截止到本书完成时为 Unity 5.6）中，法线分布函数改为采用 GGX 分布。因此，我们这里旨在让读者了解完整的 PBS 数学模型，如果读者希望了解当前所用 Unity 版本的 PBS 所使用的 BRDF 模型，我们强烈建议读者去翻看内置的 Shader 文件。

在之前的内容中，我们提到了 Unity 5 的 PBS 实际上是受 Disney BRDF[2]的启发。这种 BRDF 最大的好处之一就是很直观，只需要提供一个万能的 Shader 就可以让美术人员通过调整少量参数来渲染绝大部分常见的材质。我们可以在 Unity 内置的 UnityStandardBRDF.cginc 文件中找到它的实现。

总体来说，Unity 5 一共实现了两种 PBS 模型。一种是基于 GGX 模型的，另一种则是基于归一化的 Blinn-Phong 模型的，这两种模型使用了不同的公式来计算高光反射项中的法线分布函数 $D(\boldsymbol{h})$和阴影-遮掩函数 $G(\boldsymbol{l}, \boldsymbol{v}, \boldsymbol{h})$。Unity 5.3 以前的版本默认会使用基于归一化后的 Blinn-Phong 模型来实现基于物理的渲染，但在 Unity 5.3 及后续版本中，默认将使用 GGX 模型，这和很多其他主流引擎的选择一致。

在这两种实现中，Unity 使用的 BRDF 中的漫反射项都与 Disney BRDF 中的漫反射项相同，

---

① https://labs.unity.com/article/real-time-polygonal-light-shading-linearly-transformed-cosines
② https://seblagarde.wordpress.com/2012/09/29/image-based-lighting-approaches-and-parallax-corrected-cubemap/

即：

漫反射项：

$$f_{diff}(\boldsymbol{l},\boldsymbol{v}) = \frac{baseColor}{\pi}(1+(F_{D90}-1)(1-\boldsymbol{n}\cdot\boldsymbol{l})^5)(1+(F_{D90}-1)(1-\boldsymbol{n}\cdot\boldsymbol{v})^5)$$

$$\text{其中，}\ F_{D90} = 0.5 + 2roughness(\boldsymbol{h}\cdot\boldsymbol{l})^2$$

baseColor 一般由纹理采样和颜色参数共同决定。

高光反射项中的菲涅耳反射函数 $F(\boldsymbol{l}, \boldsymbol{h})$也与 Disney BRDF 中的一致，即使用的 Schlick 菲涅耳近似等式[7]：

菲涅耳反射函数：

$$F_{Schlick}(\boldsymbol{l},\boldsymbol{h}) = \boldsymbol{c}_{spec} + (1-\boldsymbol{c}_{spec})(1-(\boldsymbol{l}\cdot\boldsymbol{h}))^5$$

其中，$\boldsymbol{c}_{spec}$一般由纹理采样或高光颜色所决定。

Unity 5 两种 PBS 模型的主要区别在于它们所选择的法线分布函数及其对应的阴影-遮掩函数的不同。基于 GGX 模型的 PBS 的法线分布函数为 GGX 分布，基于归一化 Blinn-Phong 模型的 PBS 的法线分布函数则为归一化后的 Phong 分布。它们的公式分别如下：

法线分布函数：

$$D_{GGX}(\boldsymbol{h}) = \frac{\alpha^2}{\pi((\alpha^2-1)(\boldsymbol{n}\cdot\boldsymbol{h})^2+1)^2},\ \alpha = roughness^2$$

$$D_{blinn}(\boldsymbol{h}) = \frac{\alpha+2}{2\pi}(\boldsymbol{n}\cdot\boldsymbol{h})^{\alpha},\ \alpha = \frac{2}{roughness^4} - 2$$

需要注意的是，不同 Unity 版本在上述实现上可能会略有不同，比如 *roughness* 的指数部分会有所不同。

阴影-遮掩函数的选择更加复杂一些。在 Unity 5.3 之前的版本中，基于 GGX 模型和归一化 Blinn-Phong 模型的 PBS 的阴影-遮掩函数分别是为 GGX 和 Beckmann 设计的 Smith-Schlick 模型[8]。 它们的公式分别如下：

Unity 5.3 之前的阴影-遮掩函数：

$$\frac{G_{GGX}(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})}{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})} = \frac{1}{((\boldsymbol{n}\cdot\boldsymbol{l})(1-k)+k)((\boldsymbol{n}\cdot\boldsymbol{v})(1-k)+k)},\ k = \frac{roughness^2}{2}$$

$$\frac{G_{Beckmann}(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})}{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})} = \frac{1}{((\boldsymbol{n}\cdot\boldsymbol{l})(1-k)+k)((\boldsymbol{n}\cdot\boldsymbol{v})(1-k)+k)},\ k = roughness^2\sqrt{\frac{2}{\pi}}$$

尽管很多文献都曾推荐使用上述的 Smith-Schlick 阴影-遮掩函数，然而，Naty Hoffman[1]和 Eric Heitz[12]都指出，这种 Smith-Schlick 阴影-遮掩函数是作者 Schlick 对一个错误版本的 Smith 模型的近似公式，这意味着这些 Smith-Schlick 阴影-遮掩函数并不是基于物理的，因为它不能保证微面元投影区域面积的守恒定律。在 Unity 5.3 及其后续版本中，Unity 为基于 GGX 的 PBS 模型改用了 Smith-Joint 阴影-遮掩函数[12]。Smith-Joint 阴影-遮掩函数的公式如下：

Unity 5.3 以后 GGX 的阴影-遮掩函数：

$$\begin{aligned}\frac{G_{SmithJoint}(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})}{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})} &= \frac{1}{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})(1+\Lambda(\omega_o)+\Lambda(\omega_i))} \\ &= \frac{1}{(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})(1+0.5(-1+\sqrt{1+\alpha_g^2\tan\theta_v{}^2})+0.5(-1+\sqrt{1+\alpha_g^2\tan\theta_l{}^2}))} \\ &= \frac{2}{(\boldsymbol{n}\cdot\boldsymbol{l})\sqrt{\alpha_g^2+(\boldsymbol{n}\cdot\boldsymbol{v})^2(1-\alpha_g^2)}+(\boldsymbol{n}\cdot\boldsymbol{v})\sqrt{\alpha_g^2+(\boldsymbol{n}\cdot\boldsymbol{l})^2(1-\alpha_g^2)}} \\ &\approx \frac{2}{(\boldsymbol{n}\cdot\boldsymbol{l})((\boldsymbol{n}\cdot\boldsymbol{v})(1-\alpha_g)+\alpha_g)+(\boldsymbol{n}\cdot\boldsymbol{v})((\boldsymbol{n}\cdot\boldsymbol{l})(1-\alpha_g)+\alpha_g)}\end{aligned}$$

其中，$\alpha_g = roughness^2$

在上述的式子中，$\boldsymbol{\Lambda}(\omega_o)$ 和 $\boldsymbol{\Lambda}(\omega_i)$分别评估出射方向和入射方向上的阴影和遮掩，基于这种分开计算的 $\boldsymbol{\Lambda}(\omega_o)$ 和 $\boldsymbol{\Lambda}(\omega_i)$的 Smith 模型，Eric Heitz[12]针对 $\boldsymbol{\Lambda}(\omega_o)$ 和 $\boldsymbol{\Lambda}(\omega_i)$的不同组合方式列举了四种形式的阴影-遮掩函数。上述的公式显示了其中一种被称为 **Height-Correlated Masking and Shadowing** 的组合方式，也是 Eric 建议在实践中使用的一种方式。由于原始的 Smith-Joint 阴影-遮掩函数涉及两个开根号操作，处于性能方面的考虑，Unity 在实现上选择使用上述仅包含乘法的近似公式来简化计算。尽管在数学上这个近似公式并不正确，但从效果上来看是足够接受的。

尽管我们已经省略了大量的数学推导和物理原理，上面大量的公式对于某些读者来说可能仍然十分晦涩难懂，这也是为什么 PBS 总是令人望而却步的原因之一。如果读者想要深入了解基于物理的渲染的数学原理和应用的话，可以参见本章的扩展阅读部分。**需要再次强调的是**，由于 Unity 版本的不同，内置 PBS 的实现也可能会发生变化。除此之外，在学术界和工业界仍然不断有新的或改良后的 BRDF 模型的出现，读者也可以根据项目需要选择与 Unity 实现不同的 BRDF 模型。尤其是如果需要在移动端应用基于物理的渲染，除了效果外性能是我们最应当关心的问题之一，此时我们可能需要针对移动平台对采用的 BRDF 模型进行一些修改，读者可以在本章的扩展阅读部分中找到更多的资料。

18.2 动手：PBS 实践

我们已经介绍了足够多的理论内容了，现在是时候动手自己实现一个基于物理渲染的 Shader 了！在本节中，我们将在 Unity Shader 中实现 18.1.7 节提到的 BRDF 模型。读者可以发现，把 PBS 应用到自己的材质中并不是一件非常困难的事情。

我们回顾使用了精确光源简化后的渲染方程：

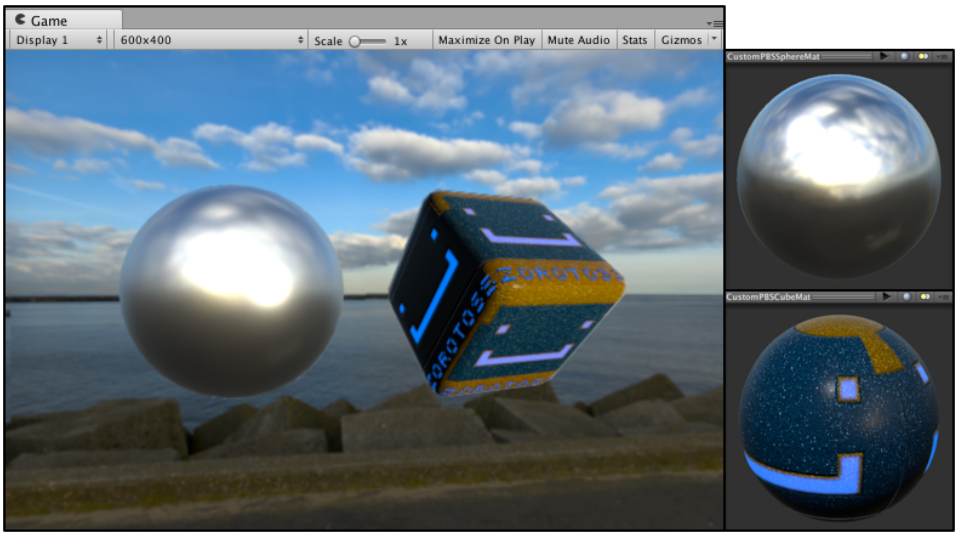
$$L_o(\boldsymbol{v}) = L_e(\boldsymbol{v}) + \sum_{i=0}^{n} L_o^i(\boldsymbol{v}) = L_e(\boldsymbol{v}) + \sum_{i=0}^{n} \pi f(\boldsymbol{l}_c^i, \boldsymbol{v}) \boldsymbol{c}_{light}(\boldsymbol{n}\cdot\boldsymbol{l}_c^i)$$

其中，$L_e(\boldsymbol{v})$是自发光部分，$f(\boldsymbol{l}^i_c, \boldsymbol{v})$是最为关键的 BRDF 模型部分。BRDF 的高光反射项则可以用下面的通用形式来表示：

$$f_{spec}(\boldsymbol{l},\boldsymbol{v}) = \frac{F(\boldsymbol{l},\boldsymbol{h})G(\boldsymbol{l},\boldsymbol{v},\boldsymbol{h})D(\boldsymbol{h})}{4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})}$$

在本例中，我们会使用 Disney BRDF 中的漫反射项、Schlick 菲涅耳近似等式、基于 GGX 模型的法线分布函数和 Smith-Joint 阴影-遮掩函数作为 BRDF 光照模型的实现。在学习完本节后，

我们会得到类似图 18.7 中的效果：



▲图 18.7　使用自定义的基于物理渲染的材质（立方体材质的纹理来源：Standard Shader Gallery[①]）

7.1.1　实践

我们的准备工作如下。

（1）在 Unity 中新建一个场景。在本书资源中，该场景名为 Scene_18_2。我们使用本书资源中的天空盒材质 EveningSkyboxHDR，在 *Window -> Lighting -> Skybox* 中代替场景默认的天空盒。

（2）新建两个材质。在本书资源中，这两个材质分别名为 CustomPBSCubeMat 和 CustomPBSSphereMat。

（3）新建一个 Unity Shader。在本书资源中，该 Unity Shader 名为 Chapter18-CustomPBR。把新的 Unity Shader 赋给第 2 步中创建的材质。

（4）在场景中放置一个球体和立方体，并把第 2 步中的两个材质分别赋给两个物体。

（5）保存场景。

打开新建的 Chapter18-CustomPBR，删除所有已有代码，并进行如下修改。

（1）首先，我们需要为这个 Unity Shader 起一个名字：

```
Shader "Unity Shaders Book v2/Chapter 18/Custom PBR"
```

（2）然后，我们需要在 Properties 语义块中声明 PBR 中需要的所有材质属性：

```
Properties {
    _Color ("Color", Color) = (1, 1, 1, 1)
    _MainTex ("Albedo", 2D) = "white" {}
    _Glossiness ("Smoothness", Range(0.0, 1.0)) = 0.5
    _SpecColor ("Specular", Color) = (0.2, 0.2, 0.2)
    _SpecGlossMap ("Specular (RGB) Smoothness (A)", 2D) = "white" {}
    _BumpScale ("Bump Scale", Float) = 1.0
    _BumpMap ("Normal Map", 2D) = "bump" {}
    _EmissionColor ("Color", Color) = (0, 0, 0)
    _EmissionMap ("Emission", 2D) = "white" {}
```

[①] https://www.assetstore.unity3d.com/en/#!/content/58870

```
}
```

其中，_MainTex 和_Color 用于控制漫反射项中的材质纹理和颜色，_SpecColor 和_SpecGlossMap 的 RGB 通道值用于控制材质的高光反射颜色。_SpecGlossMap 的 A 通道值和_Glossiness 用于共同控制材质的粗糙度。_BumpMap 则是材质的法线纹理，它的凹凸程度可以依靠_BumpScale 属性来控制。最后，_EmissionColor 和_EmissionMap 用于控制材质的自发光颜色。

3）定义 Forwad Base Pass：

```
SubShader {
     Tags { "RenderType"="Opaque" }
     LOD 300

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

          CGPROGRAM
          #pragma target 3.0

          #pragma multi_compile_fwdbase
          #pragma multi_compile_fog
```

注意，在上面的代码中我们通过使用#pragma target 3.0 来指明使用 Shader Target 3.0，这是因为基于物理渲染涉及了较多的公式，因此需要较多的数学指令来进行计算，这可能会超过 Shader Target 2.0 对指令数目的规定，因此我们选择使用更高的 Shader Target 3.0。

4）接下来，我们来定义顶点着色器：

```
struct v2f {
     float4 pos : SV_POSITION;
     float2 uv : TEXCOORD0;
     float4 TtoW0 : TEXCOORD1;
     float4 TtoW1 : TEXCOORD2;
     float4 TtoW2 : TEXCOORD3;
     SHADOW_COORDS(4)     // Defined in AutoLight.cginc
     UNITY_FOG_COORDS(5) // Defined in UnityCG.cginc
};

v2f vert(a2v v) {
     v2f o;
     UNITY_INITIALIZE_OUTPUT(v2f, o); // Defined in HLSLSupport.cginc

     o.pos = UnityObjectToClipPos(v.vertex); // Defined in UnityCG.cginc
     o.uv = TRANSFORM_TEX(v.texcoord, _MainTex); // Defined in UnityCG.cginc

     float3 worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
     fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
     fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
     fixed3 worldBinormal = cross(worldNormal, worldTangent) * v.tangent.w;

     o.TtoW0 = float4(worldTangent.x, worldBinormal.x, worldNormal.x, worldPos.x);
     o.TtoW1 = float4(worldTangent.y, worldBinormal.y, worldNormal.y, worldPos.y);
     o.TtoW2 = float4(worldTangent.z, worldBinormal.z, worldNormal.z, worldPos.z);

     //We need this for shadow receving
     TRANSFER_SHADOW(o); // Defined in AutoLight.cginc

     //We need this for fog rendering
     UNITY_TRANSFER_FOG(o, o.pos); // Defined in UnityCG.cginc

     return o;
}
```

顶点着色器中的计算比较简单，我们在之前的章节中也进行过类似的计算。为了在片元着色器中把采样得到的切线空间下的法线方向转换到世界空间下，我们把变换矩阵的相关数据存储在

了 o.TtoW0、o.TtoW1 和 o.TtoW2 中。除此之外，我们还使用内置宏 SHADOW_COORDS、UNITY_FOG_COORDS、TRANSFER_SHADOW 和 UNITY_TRANSFER_FOG 等声明和计算了阴影和雾效所需要的一些纹理坐标参数。

5）片元着色器是我们的重点，我们来一步步看它是怎样实现的：

```
half4 frag(v2f i) : SV_Target {
     ///// Prepare all the inputs
     half4 specGloss = tex2D(_SpecGlossMap, i.uv);
     specGloss.a *= _Glossiness;
     half3 specColor = specGloss.rgb * _SpecColor.rgb;
     half roughness = 1 - specGloss.a;

     half oneMinusReflectivity = 1 - max(max(specColor.r, specColor.g), specColor.b);

     half3 diffColor = _Color.rgb * tex2D(_MainTex, i.uv).rgb * oneMinusReflectivity;

     half3 normalTangent = UnpackNormal(tex2D(_BumpMap, i.uv));
     normalTangent.xy *= _BumpScale;
     normalTangent.z = sqrt(1.0 - saturate(dot(normalTangent.xy, normalTangent.xy)));
     half3    normalWorld    =    normalize(half3(dot(i.TtoW0.xyz,    normalTangent),
dot(i.TtoW1.xyz, normalTangent), dot(i.TtoW2.xyz, normalTangent)));

     float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
     half3 lightDir = normalize(UnityWorldSpaceLightDir(worldPos)); // Defined in
UnityCG.cginc
     half3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos)); // Defined in
UnityCG.cginc

     half3 reflDir = reflect(-viewDir, normalWorld);

     UNITY_LIGHT_ATTENUATION(atten, i, worldPos); // Defined in AutoLight.cginc
```

我们首先需要为后续计算准备好所有的输入数据，这些输入大多来源于材质面板中的各个属性，例如漫反射颜色 diffColor 和高光反射颜色 specColor、粗糙度 roughness、世界空间下的法线方向、光源方向、观察方向、反射方向等。我们还使用内置宏 UNITY_LIGHT_ATTENUATION 计算了阴影和光照衰减值 atten。除此之外，我们还计算了一个变量 oneMinusReflectivity，这个变量并不是我们之前提到的 BRDF 中需要的变量，它主要是为了计算掠射角的反射颜色，从而得到效果更好的菲涅耳反射效果。

接下来，我们开始计算最重要的 BRDF 光照模型。在此之前，我们先准备好各个角度的余弦值，即之前公式中的各个点乘项，如$(\boldsymbol{n}\cdot\boldsymbol{v})$、$(\boldsymbol{n}\cdot\boldsymbol{l})$等：

```
///// Compute BRDF terms
half3 halfDir = normalize(lightDir + viewDir);
half nv = saturate(dot(normalWorld, viewDir));
half nl = saturate(dot(normalWorld, lightDir));
half nh = saturate(dot(normalWorld, halfDir));
half lv = saturate(dot(lightDir, viewDir));
half lh = saturate(dot(lightDir, halfDir));
```

通过使用 Cg 的 saturate 函数，我们把这些点乘值的范围截取到了[0, 1]之间，来避免背光面的光照。然后，我们来计算 BRDF 中的漫反射项：

```
// Diffuse term
half3 diffuseTerm = CustomDisneyDiffuseTerm(nv, nl, lh, roughness, diffColor);
```

6）对于漫反射项，我们选择使用 Disney BRDF 中的漫反射项实现，CustomDisneyDiffuseTerm 函数的实现（依照 Disney BRDF 中的漫反射项公式）如下：

```
inline half3 CustomDisneyDiffuseTerm(half NdotV, half NdotL, half LdotH, half
roughness, half3 baseColor) {
     half fd90 = 0.5 + 2 * LdotH * LdotH * roughness;
```

```
        // Two schlick fresnel term
        half lightScatter    = (1 + (fd90 - 1) * pow(1 - NdotL, 5));
        half viewScatter     = (1 + (fd90 - 1) * pow(1 - NdotV, 5));

        return baseColor * UNITY_INV_PI * lightScatter * viewScatter;
    }
```

这个函数非常简单，我们就是按照之前提到的公式实现相应代码而已。UNITY_INV_PI 是在 UnityCG.cginc 文件中定义的宏变量，即圆周率 π 的倒数。在上面的实现中，我们还使用了 Cg 关键词 inline[①]来修饰函数声明，inline 的作用是用于告诉编译器应该尽可能使用内联调用的方式来调用该函数，减少函数调用的开销。

7）下面，我们来实现高光反射项：

```
// Specular term
half V = CustomSmithJointGGXVisibilityTerm(nl, nv, roughness);
half D = CustomGGXTerm(nh, roughness * roughness);
half3 F = CustomFresnelTerm(specColor, lh);
half3 specularTerm = F * V * D;
```

首先是可见性项 V，它计算的是阴影-遮掩函数除以高光反射项的分母部分后的结果。CustomSmithJointGGXVisibilityTerm 函数的实现（依照 Eric Heitz[12]提出的按 Height-Correlated Masking and Shadowing 方式组合的 Smith-Joint 阴影-遮掩函数）如下：

```
inline half CustomSmithJointGGXVisibilityTerm(half NdotL, half NdotV, half roughness)
{
    // Original formulation:
    //   lambda_v   = (-1 + sqrt(a2 * (1 - NdotL2) / NdotL2 + 1)) * 0.5f;
    //   lambda_l   = (-1 + sqrt(a2 * (1 - NdotV2) / NdotV2 + 1)) * 0.5f;
    //   G              = 1 / (1 + lambda_v + lambda_l);

    // Approximation of the above formulation (simplify the sqrt, not mathematically
correct but close enough)
    half a2 = roughness * roughness;
    half lambdaV = NdotL * (NdotV * (1 - a2) + a2);
    half lambdaL = NdotV * (NdotL * (1 - a2) + a2);

    return 0.5f / (lambdaV + lambdaL + 1e-5f);
}
```

接下来是法线分布项 D，CustomGGXTerm 函数的实现（依照基于 GGX 模型的法线分布函数）如下：

```
inline half CustomGGXTerm(half NdotH, half roughness) {
    half a2 = roughness * roughness;
    half d = (NdotH * a2 - NdotH) * NdotH + 1.0f;
    return UNITY_INV_PI * a2 / (d * d + 1e-7f);
}
```

最后是菲涅耳反射项 F，CustomFresnelTerm 函数（依照 Schlick 菲涅耳近似等式[7]）的实现如下：

```
inline half3 CustomFresnelTerm(half3 c, half cosA) {
    half t = pow(1 - cosA, 5);
    return c + (1 - c) * t;
}
```

最后的高光反射项就是把 V、D 和 F 相乘后的结果。

8）接下来，我们还需要计算自发光项：

① http://http.developer.nvidia.com/Cg/Cg_language.html

```
// Emission term
half3 emisstionTerm = tex2D(_EmissionMap, i.uv).rgb * _EmissionColor.rgb;
```

自发光项非常简单，我们只需要从自发光纹理中进行采样再乘以自发光颜色即可。

9）为了得到更加真实的光照，我们还需要计算基于图像的光照部分（IBL）：

```
// IBL
half perceptualRoughness = roughness * (1.7 - 0.7 * roughness);
half mip = perceptualRoughness * 6;
half4 envMap = UNITY_SAMPLE_TEXCUBE_LOD(unity_SpecCube0, reflDir, mip); // Defined in
HLSLSupport.cginc
half grazingTerm = saturate((1 - roughness) + (1 - oneMinusReflectivity));
half surfaceReduction = 1.0 / (roughness * roughness + 1.0);
half3 indirectSpecular = surfaceReduction * envMap.rgb * CustomFresnelLerp(specColor,
grazingTerm, nv);
```

IBL 部分的主要思想是使用材质粗糙度对环境贴图进行 LOD（Level Of Detail）采样，这是因为粗糙度越大的材质，反射的环境光照应该越模糊，而这可以通过对环境贴图不同级数的多级渐远纹理（mipmaps）进行采样来模拟得到。级数越高，在多级渐远纹理中对应的纹理就越小，图像也就越模糊。

为了计算需要采样的多级渐远纹理的级数，我们将材质粗糙度乘以某个常数（在上述实现中该常数为 6），这个常数表明了整个粗糙度范围内多级渐远纹理的总级数。需要注意的是，这种由粗糙度计算级数的方法并不是唯一的，读者可以在 UnityImageBasedLighting.cginc 文件的 perceptualRoughnessToMipmapLevel 函数中找到相关实现。然后，我们使用该级数和反射方向来对环境贴图进行采样。其中，unity_SpecCube0 包含了该物体周围当前活跃的反射探针（Reflection Probe）中所包含的环境贴图。尽管我们没有在场景中手动放置任何反射探针，但 Unity 会根据 *Window -> Lighting -> Skybox* 中的设置，在场景中生成一个默认的反射探针。由于在本节的准备工作中我们在 *Window -> Lighting -> Skybox* 中设置了自定义的天空盒，因此此时 unity_SpecCube0 中包含的就是这个自定义天空盒的环境贴图。如果我们在场景中放置了其他反射探针，Unity 则会根据相关设置和物体所在的位置自动把距离该物体最近的一个或几个反射探针数据传递给 Shader。尽管在之前的内容中，我们是使用 samplerCUBE 来声明一个立方体贴图并使用 texCUBE 来采样它，但是 Unity 内置反射探针的立方体贴图则是以一种特殊的方式声明的，这主要是为了在某些平台下可以节省 sampler slots。读者可以在 UnityShaderVariables.cginc 文件中找到 unity_SpecCube0 的声明，Unity 主要是通过 HLSLSupport.cginc 文件中定义的内置宏 UNITY_DECLARE_TEXCUBE 来实现的。由于这样的特殊性，在采样 unity_SpecCube0 时我们也应该使用内置宏如 UNITY_SAMPLE_TEXCUBE（在 HLSLSupport.cginc 文件中被定义）来采样。由于在这里我们还需要对指定级数的多级渐远纹理采样，因此我们使用内置宏 UNITY_SAMPLE_TEXCUBE_LOD（在 HLSLSupport.cginc 文件中被定义）来实现。至此，我们得到了采样后的环境光照颜色 envMap。

然后，为了给 IBL 添加更加真实的菲涅耳反射，我们对高光反射颜色 specColor 和掠射颜色 grazingTerm 进行菲涅耳插值。掠射颜色 grazingTerm 是由材质粗糙度和之前计算得到的 oneMinusReflectivity 共同决定的。使用掠射角度进行菲涅耳插值的好处是，我们可以在掠射角得到更加真实的菲涅耳反射效果，同时还考虑了材质粗糙度的影响。除此之外，我们还使用了由粗糙度计算得到的 surfaceReduction 参数进一步对 IBL 的进行修正。CustomFresnelLerp 的函数实现如下：

```
inline half3 CustomFresnelLerp(half3 c0, half3 c1, half cosA) {
     half t = pow(1 - cosA, 5);
     return lerp (c0, c1, t);
}
```

它的实现和之前实现的 CustomFresnelTerm 函数很类似，不同的是这里使用参数 t 来混合两

个颜色。尽管 grazingTerm 被声明为单一维数的 half 变量，在传递给 CustomFresnelLerp 时它会自动被转换成 half3 类型的变量，这在 Cg 中被称为是**"Smearing" Of Scalars To Vectors**[①]。

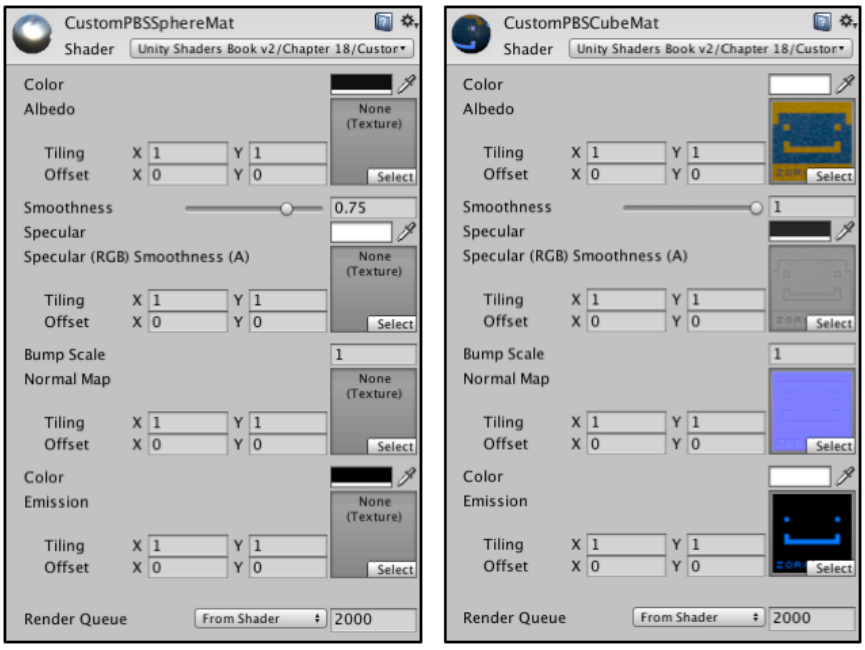
10）最后，我们只需要按照渲染方程把所有项加起来即可：

```
// Combine all togather
half3 col = emisstionTerm + UNITY_PI * (diffuseTerm + specularTerm) * _LightColor0.rgb
* nl * atten + indirectSpecular;

UNITY_APPLY_FOG(i.fogCoord, c.rgb); // Defined in UnityCG.cginc

return half4(col, 1);
```

在返回最后的像素颜色前，我们还添加了雾效的影响。至此我们完成了 Forward Base Pass 中的所有实现。

11）由于场景中可能存在多个光源，我们还需要实现 Forward Add Pass。Forward Add Pass 的实现与 Forward Base Pass 基本一致，其中不同的是，Forward Add Pass 不需要计算雾效、自发光和 IBL 的部分，因为这些只需要在 Forward Base Pass 计算一遍即可。其他实现在此不再赘述。至此，我们就完成了一个较为完整的基于物理渲染的 Shader。

保存后返回场景，再调整相关参数即可得到类似图 18.7 中的效果。图 18.7 中物体所用的材质面板如图 18.8 所示，它们分别对应了一个金属类型的材质和一个塑料类型的材质。关于如何使用 PBR 设置各种材质参数，读者可以参见 18.3.2 节中的内容。



▲图 18.8　本例中物体使用的材质参数

需要注意的是，我们还需要保证 *Player Settings* → *Other Settings* → *Rendering* → *Color Space* 中的选项是 *Linear*，即线性空间，只有这样才能保证我们的计算是在线性空间下进行的，且输出的为线性颜色。与线性空间相关的是伽马校正，这部分内容读者可以参见本章的 18.4.2 节。

① http://http.developer.nvidia.com/Cg/Cg_language.html

在上面的内容中，我们依靠自定义的函数实现了一个基于 GGX BRDF 模型的 Shader。实际上，Unity 已经帮我们实现了很多 BRDF 模型中的函数，并为我们提供了现成的基于物理着色的 Shader，也就是 Standard Shader。

18.3 Unity 5 的 Standard Shader

当我们在 Unity 5 中新创建一个模型或是新创建一个材质时，其默认使用的着色器都是一个名为 Standard 的着色器。这个 Standard Shader 就使用了我们之前所讲的基于物理的渲染。

Unity 支持两种流行的基于物理的工作流程：**金属工作流（metallic workflow）**和**高光反射工作流（specular workflow）**。其中，金属工作流是默认的工作流程，对应的 Shader 为 Standard Shader。而如果想要使用高光反射工作流，就需要在材质的 Shader 下拉框中选择 Standard（Specular setup）。需要注意的是，通常来讲，使用不同的工作流可以实现相同的效果，只是它们使用的参数不同而已。金属工作流也不意味着它只能模拟金属类型的材质，金属工作流的名字来源于它定义了材质表面的金属值（是金属类型的还是非金属类型的）。高光反射工作流的名字来源于它可以直接指定表面的高光反射颜色（有很强的高光反射还是很弱的高光反射）等，我们在 18.2 节中实现的自定义的 PBS 实际上也是按高光反射工作流的方式来定义的。而在金属工作流中这个颜色需要由漫反射颜色和金属值衍生而来。在实际的游戏制作过程中，我们可以选择自己更偏好的工作流来制作场景，这更多的是个人喜好的问题。当然也可以同时混用两种工作流。而在内部实现上，这两种工作流实际上最终都会使用同一套 BRDF 模型，不同的是 BRDF 模型中各个输入参数的来源不同而已。

在下面的内容中，我们用 Standard Shader 来统称 Standard 和 Standard（Specular setup）Shader。Unity 提供的 Standard Shader 允许让我们只使用这一种 Shader 来为场景中所有的物体进行着色，而不需要考虑它们是否是金属材质还是塑料材质等，从而大大减少我们不断调整材质参数所花费的时间。

18.3.1　它们是如何实现的

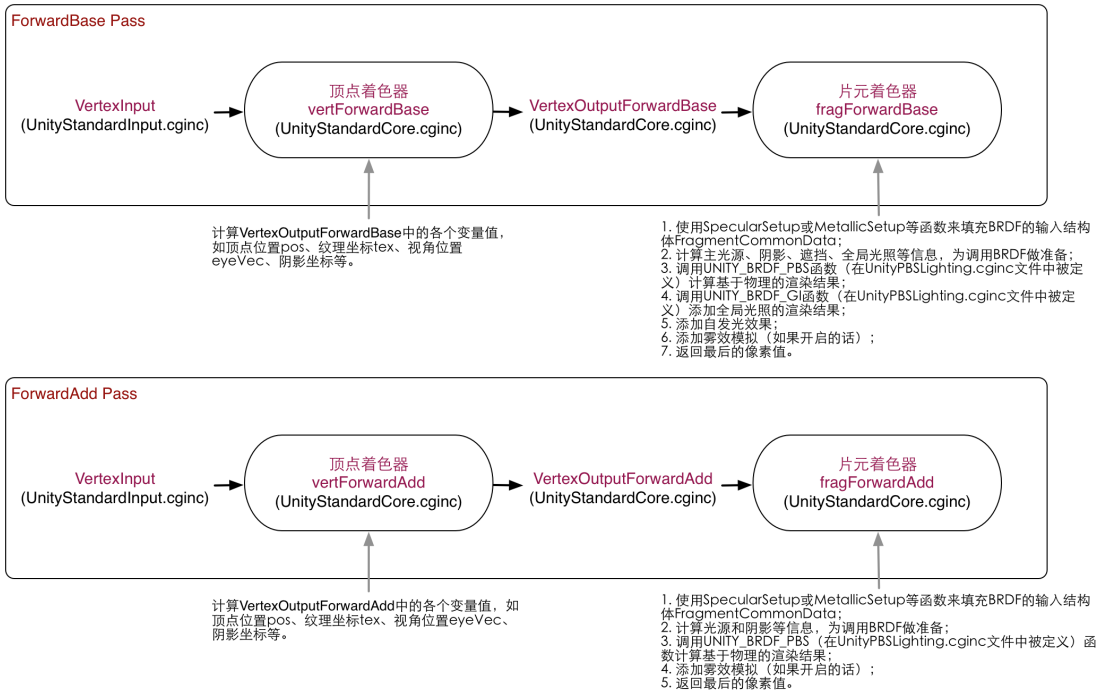
Standard 和 Standard（Specular setup）的 Shader 源代码可以在 Unity 内置的 builtin_shaders-5.x/DefaultResourcesExtra 文件夹中找到，这些 Shader 依赖于 builtin_shaders-5.x/CGIncludes 文件夹中定义的一些头文件。这些相关的头文件的名称大多类似于 UnityStandardXXX.cginc，其中定义了和 PBS 相关的各个函数、结构体和宏等。表 18.1 列出了这些头文件的名称以及它们的主要用处。

表 18.1 Unity 内置的与 PBS 相关的各个头文件以及相关描述

文　件	描　述
UnityStandardInput.cginc	声明了 Standard 和 Standard (Specular setup) Shader 使用的所有材质参数（如 _Color、_MainTex、_EmissionMap），定义了顶点着色器的输入结构体 VertexInput，还定义了相关辅助函数用于从这些输入中计算得到相关的材质变量，例如 Albedo 函数可以从_MainTex 和_Color 参数中计算得到漫反射颜色，Occlusion 函数可以从_OcclusionMap 中计算得到遮挡值
UnityLightingCommon.cginc	定义了和 PBS 光照相关的各个结构体，例如 UnityLight、UnityIndirect、UnityGI 和 UnityGIInput 等
UnityStandardCore.cginc	定义了 Standard 和 Standard (Specular setup) Shader 使用的顶点/片元着色器（如 vertForwardBase 和 fragForwardBase）、相关的结构体（如 VertexOutputForwardBase 和 FragmentCommonData）和辅助函数（如

	MetallicSetup、SpecularSetup、MainLight、PerPixelWorldNormal、FragmentGI）等
UnityStandardCoreForwardSimple.cginc	定义了简化版的顶点/片元着色器（如 vertForwardBaseSimple 和 fragForwardBaseSimple）、相关的结构体（如 VertexOutputBaseSimple）和辅助函数（如 FragmentSetupSimple、MainLightSimple）等。在默认情况下，Unity 会将 UNITY_STANDARD_SIMPLE（在 UnityStandardConfig.cginc 文件中被定义）设为 0，即不使用这些简化后的实现
UnityPBSLighting.cginc	定义了表面着色器使用的标准光照函数和相关的结构体等，如 LightingStandardSpecular 函数和 SurfaceOutputStandardSpecular 结构体，这些定义主要是为 Unity 表面着色器（Surface Shader）服务的。除此之外，还定义 PBS 函数的调用宏 UNITY_BRDF_PBS，Unity 会根据当前平台设置等为 UNITY_BRDF_PBS 设置不同性能的函数入口，如 BRDF1_Unity_PBS、BRDF2_Unity_PBS 和 BRDF3_Unity_PBS 等函数，而这些函数是在 UnityStandardBRDF.cginc 文件中被定义的
UnityStandardBRDF.cginc	实现了 Unity 中基于物理的渲染技术，定义了 BRDF1_Unity_PBS、BRDF2_Unity_PBS 和 BRDF3_Unity_PBS 等函数，来实现不同平台下的 BRDF。这个文件包含了关键的 BRDF 模型的实现部分，包括漫反射项和高光反射项的函数实现（如 DisneyDiffuse、SmithJointGGXVisibilityTerm）和相关的辅助函数（如常用的数学函数 Pow4 和 Pow5、PerceptualRoughnessToSpecPower）
UnityGlobalIllumination.cginc	定义了计算全局光照的 UnityGlobalIllumination 函数，该函数在 FragmentGI 函数（在 UnityStandardCore.cginc 中被定义）中被调用，它会从光照贴图、光照探针、反射探针等输入中读取数据，计算全局光照中的间接漫反射颜色和高光反射颜色，并存储 UnityGI 结构体中的 UnityIndirect 结构体变量中（两个结构体均在 UnityLightingCommon.cginc 中被定义）
UnityImageBasedLighting.cginc	定义了和基于图像的光照相关的结构体和函数，这些结构体和函数会在计算全局光照时被使用，例如 Unity_GlossyEnvironment 函数会采样反射探针中的数据，它可以用于计算间接的高光反射
UnityStandardUtils.cginc	Standard Shader 使用的一些辅助函数，将来可能会移到 UnityCG.cginc 文件中
UnityStandardConfig.cginc	对 Standard Shader 的相关配置，例如默认情况下使用 GGX 模型来实现 BRDF（将 UNITY_BRDF_GGX 设为 1）
UnityStandardMeta.cginc	定义了 Standard Shader 中“LightMode”为“Meta”的 Pass（用于提取光照纹理和全局光照的相关信息）使用的顶点/片元着色器，以及它们使用的输入/输出结构体
UnityStandardShadow.cginc	定义了 Standard Shader 中“LightMode”为“ShadowCaster”的 Pass（用于投射阴影）使用的顶点/片元着色器，以及它们使用的输入/输出结构体

我们可以打开 Standard.shader 和 StandardSpecular.shader 文件来分析 Unity 是如何实现基于物理的渲染的。总体来讲，这两个 Shader 的代码基本相同——它们都定义了两个 SubShader，第一个 SubShader 使用的计算更加复杂，Unity 为其定义了前向渲染路径和延迟渲染路径使用的 Pass，以及用于投射阴影和提取元数据的 Pass；第二个 SubShader 也定义了 4 个 Pass，其中两个 Pass 用于前向渲染路径，一个 Pass 用于投射阴影，另一个 Pass 用于提取元数据，该 SubShader 和第一个 SubShader 的主要区别在于取消了一些计算，例如不计算视差贴图、不计算软阴影等。Standard.shader 和 StandardSpecular.shader 最大的不同之处在于，它们在设置 BRDF 的输入时使用了不同的函数来设置各个参数——基于金属工作流的 Standard Shader 使用 MetallicSetup 函数来设置各个参数，基于高光反射工作流的 Standard（Specular setup）Shader 使用 SpecularSetup 函数来设置。MetallicSetup 和 SpecularSetup 函数均在 UnityStandardCore.cginc 文件中被定义。图 18.7 给出了 Standard Shader 中用于前向渲染路径的典型实现，这是由对内置文件的分析所得。



▲图 18.7　Standard Shader 中前向渲染路径使用的 Pass（简化版本的 PBS 使用了 VertexOutputBaseSimple 等结构体来代替相应的结构体）

从图 18.7 中可以看出，两个 Pass 的代码大体相同，只是 ForwardBase Pass 进行了更多的光照计算，例如，计算全局光照、自发光等效果，这些计算只需要在物体的整个渲染过程中计算一次即可，因此不需要在 FarwardAdd Pass 中再计算一次，这与我们之前学习前向渲染时的经验一致。

18.3.2　如何使用 Standard Shader

我们之前提到，Unity 5 的 Standard Shader 适用于各种材质的物体，但是，我们应该如何使用 Standard Shader 来得到不同的材质效果呢？

我们首先来回答一个问题，为什么不同的材质看起来是如此不同呢？这需要回顾我们在 18.1 节讲到的内容。我们知道，材质和光的交互可以分成漫反射和高光反射两个部分，其中漫反射对应了次表面散射的结果，而高光反射则对应了表面反射的结果。通过对金属材质和非金属材质的分析，我们可以得到它们的漫反射和高光反射的一些特点。

1．金属材质

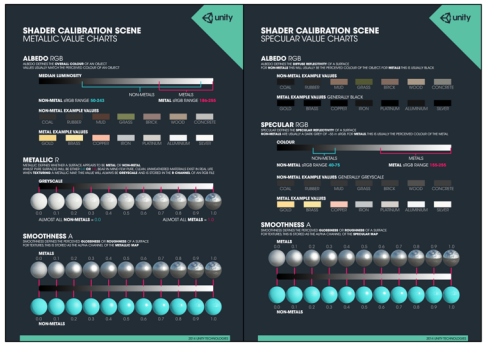
- 几乎没有漫反射，因为所有被吸收的光都会被自由电子立刻转化为其他形式的能量；
- 有非常强烈的高光反射；
- 高光反射通常是有颜色的，例如金子的反光颜色为黄色。

2．非金属材质

- 大多数角度高光反射的强度比较弱，但在掠射角时高光反射强度反而会增强，即菲涅耳现象；

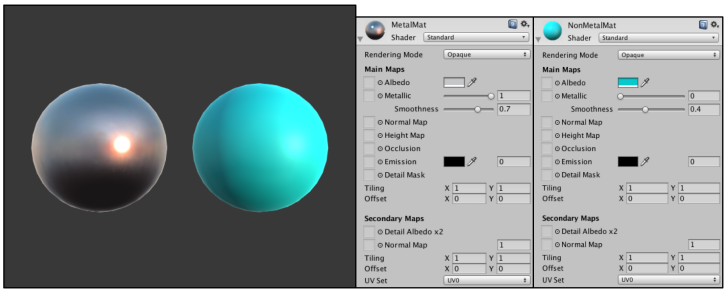
- 高光反射的颜色比较单一；
- 漫反射的颜色多种多样。

但真实的材质大多混合了上面的这些特性，Unity 提供的工作流就是为了更加方便地让我们针对以上特性来调整材质效果。在 Unity 官方提供的示例项目 **Shader Calibration Scene**[①]中，Unity 提供了两个非常有参考价值的校准表格，如图 18.8 所示，它们分别对应了金属工作流和高光反射工作流使用的参考属性值，来方便我们针对不同类型的材质来调整参数。读者也可以在本书资源的 Assets/Textures/Chapter18/Charts 文件夹找到这两张校准表格。



▲图 18.8 Unity 提供的校准表格。左边：金属工作流使用的校准表格，右边：高光反射工作流使用的校准表格

我们以图 18.8 的左图，即金属工作流使用的校准表格为例，来解释如何使用这张校准表格来指导我们调整材质。在本书资源的场景文件 Scene_18_3 中，我们提供了一个简单的场景来展示不同材质的结果。图 18.9 显示了场景结果以及物体使用的材质。需要注意的是，读者需要在 *Edit → Project Setttings→Player→Other Settings→Color Space* 中选择 *Linear* 才可以得到和图 18.9 中相同的效果，这是因为基于物理的渲染需要使用线性空间（详见 18.3.4 节）来进行相关计算。



▲图 18.9 使用金属工作流来实现不同类型的材质。左边的球体：金属材质，右边的球体：塑料材质

在金属工作流中，材质面板中的 **Albedo** 定义了物体的整体颜色，它通常就是我们视觉上认为的物体颜色。从亮度来看，非金属材质的亮度范围通常在 50～243，而金属材质的亮度一般在 186 255 之间。Unity 给的校准表格（见图 18.8 中的左图）中还给出了一些非金属材质和金属材质使用的示例 Albedo 属性值，我们可以直接使用这些示例值来作为材质属性。当然，也可以直接使用一张纹理作为材质的 Albedo 值。在我们的例子中，我们把金属材质（图 18.9 中的左边的球体）的 Albedo 设为银灰色，而把塑料材质（图 18.9 中的右边的球体）的设为蓝绿色。材质面板中的下一个属性是 **Metallic**，它定义了该物体表面看起来是否更像金属或非金属。同样，我们也可以使用一张纹理来采样得到表面的 Metallic 值，此时该纹理中的 R 通道值将对应了 Metallic 值。在我们的例子中，我们把金属材质的 Metallic 值设为 1，表明该物体几乎完全是一个金属材质，同时把塑料材质的 Metallic 值设为 0，表明该物体几乎没有任何金属特性。最后一个重要的材质属性是 **Smoothness**，它是上一个属性 **Metallic** 的附属值，定义了从视觉上来看该表面的光滑程度。如果我们在设置 **Metallic** 属性时使用的是一张纹理，那么这张纹理的 A 通道就对应了表面的 Smoothness 值（此时纹理的 GB 通

① https://www.assetstore.unity3d.com/en/#!/content/25422

道则被忽略）。在我们的例子中，我们把金属材质的 Smoothness 值设置为相对较大的 0.7，表明该金属表面比较光滑，而把塑料材质的 Smoothness 值设为 0.4，表明该塑料表明比较粗糙。

高光反射工作流使用的面板和上述金属工作流使用的基本相同，只是使用了不同含义的 **Albedo** 属性，并使用 **Specular** 代替了上述的 **Metallic** 属性。在高光反射工作流中，材质的 **Albedo** 属性定义了表面的漫反射强度。对于非金属材质，它的值通常仍然是视觉上认为的物体颜色，但对于金属材质，Albedo 的值通常非常接近黑色（还记得吗，金属材质几乎不存在次表面散射的现象）。高光反射工作流的 **Specular** 属性则定义了表面的高光反射强度。非金属材质通常使用一个灰度值范围在 0～55 的深灰色来作为 Specular 值，表明非金属材质的高光反射较弱。金属材质则通常会使用视觉上认为的该金属的颜色作为它的 Specular 值。**Specular** 属性同样也有一个子属性 **Smoothness**，它定义了从视觉上来看该表面的光滑程度。和上面的金属工作流类似，如果使用了一张纹理来为 **Specular** 属性赋值，那么纹理的 RGB 通道对应了 Specular 属性值，A 通道对应了 Smoothness 属性值。我们在 18.2 节中实现的自定义的 PBS 实际上就是遵循了高光反射工作流。

上述材质属性都属于材质面板中的 **Main Maps** 部分，除了上述提到的属性外，Main Maps 还包含了其他材质属性，例如，切线空间下的法线纹理、遮挡纹理、自发光纹理等。Main Maps 部分的下面还有一个 **Secondary Maps** 的属性部分，这个部分的属性是用来定义额外的细节信息，这些细节通常会直接绘制在 Main Maps 的上面，来为材质提供更多的微表面或细节表现。

除了上述属性，我们还可以为 Standard Shader 选择它使用的渲染模式，即材质面板上的 **Render Mode** 选项。Standard Shader 支持 4 种渲染模式，分别是 Opaque、Cutout、Fade 和 Transparent。其中，Opaque 用于渲染最常见的不透明物体，这也是默认的渲染模式。对于像玻璃这样的材质，我们可以选择 Transparent 模式，在这个渲染模式下，Albedo 属性的 A 通道用于控制材质的透明度。而在 Cutout 渲染模式下，Albedo 属性中纹理的 A 通道会成为一个遮掩纹理，而它的子属性 Alpha Cutoff 将是透明度测试时使用的阈值。Fade 模式和 Transparent 模式是类似的，不同的是，在 Transparent 模式下，当材质的透明值不断降低时，它的反射仍然能被保留，而在 Fade 模式下，该材质的所有渲染效果都会逐渐从屏幕上淡出。

需要注意的是，尽管 Standard Shader 的材质面板有许多可供调节的属性，但我们不用担心由于没有使用一些属性而会对性能有所影响。Unity 在背后已经进行了高度优化，在我们生成可执行程序时，Unity 会检查哪些属性没有被使用到，同时也会针对目标平台进行相应的优化。这些逻辑是通过使用一个 C#脚本文件来自定义材质面板的行为来实现的，读者可以在 builtin_shaders-5.x/Editor/ StandardShaderGUI.cs 中找到这个文件。这个脚本的核心思想是通过判断用户是否设置了某一材质属性来决定是否开启相应的 shader feature，例如，如果用户没有设置法线纹理，那么该脚本就会关闭名为_NORMALMAP 的 shader feature，从而在 Shader 逻辑中跳过相关代码。

从上面的内容可以看出，要想得到可信度更高的渲染结果，我们需要对不同材质使用合适的属性值，尤其是一些重要的属性值，例如 Albedo、Metallic 和 Specular。当然，想要让整个场景的渲染结果令人满意，尤其包含了复杂光照的场景，仅仅有这些使用了 PBS 的材质是不够的，我们需要使用 Unity 提供的其他一些重要的技术，例如 HDR 格式的 Skybox、全局光照、反射探针、光照探针、HDR 和屏幕后处理等。更多内容读者阅读本章的扩展阅读部分。

18.3 一个更加复杂的例子

在本章最后，我们将以一个更加复杂的、基于物理渲染的场景结束，该场景对应了本书资源中的 Scene_18_3。本场景使用的元素大多来源于 Unity 官方的示例项目 **Viking Village**（https://www.assetstore.unity3d.com/jp/#!/content/29140），读者可以下载完整的项目来更加深入地学习 Unity 中的 PBS。

图 18.10 展示了在不同光照条件下本例实现的效果。需要注意的是，读者需要在 Edit → Project Setttings→Player→Color Space 中选择 Linear 才可以得到和图 18.9 中相同的效果，这是因为基于物理的渲染需要使用线性空间（详见 18.3.4 节）来进行相关计算。

▲图 18.10 在 Unity 5 中使用基于物理的渲染技术，场景在不同光照下的渲染结果

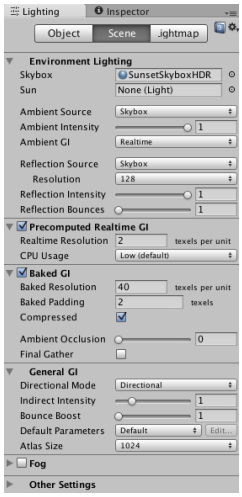
那么，基于物理的 Standard Shader 是如何与其他 Unity 功能相互配合得到这样的场景呢？

18.3.1 设置光照环境

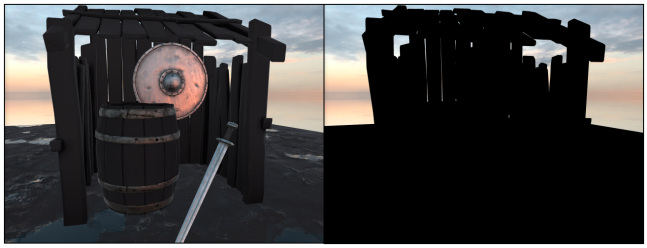
我们首先需要为场景设置光照环境。在默认情况下，Unity 5 中一个新创建的场景会包含一个默认的 Skybox。在本例中，我们使用一个自定义的 Skybox 来代替默认值。做法是，打开 Window→Lighting，在 Scene 标签页下把本例使用的 SunsetSkyboxHDR 拖曳到 **Skybox** 选项中，如图 18.11 所示。

本例中的 Skybox 使用了一个 HDR 格式的 Cubemap，这与我们之前在 10.1 节中制作 Skybox 时使用的纹理不同。这需要解释 HDR（High Dynamic Range）的相关知识，我们将在 18.4.3 节更加详细地介绍 HDR 的原理和应用。但在这里，我们只需要知道，使用 HDR 格式的 Skybox 可以让场景中物体的反射更加真实，有利于我们得到更加可信的光照效果。

我们还可以设置场景使用的**环境光照**，这些环境光照可以对场景中所有的物体表面产生影响。在图 18.11 所示的设置面板中，我们可以选择环境光照的来源（**Ambient Source** 选项），是来自于场景使用的 Skybox，还是使用渐变值，亦或是某个固定的颜色。我们还可以设置环境光照的强度（**Ambient Intensity** 参数），如果想要场景中的所有物体不接受任何环境光照，可以把该值设为 0。在使用了 Standard Shader 的前提下，如果我们关闭场景中所有的光源，并把环境光照的强度设为 0，场景中的物体仍然可以接受一些光照，如图 18.12 中的左图所示。



▲图 18.11　光照面板下的 Scene 标签页



▲图 18.12　左边：当关闭场景中的所有光源并把环境光照强度设为 0 后，使用了 Standard Shader 的物体仍然具有光照效果，右边：在左图的基础上，把反射源设置为空，使得物体不接受任何默认的反射信息

那么，这些光照是从哪里来的呢？答案就是**反射**。默认的反射源（**Reflection Source** 选项）是场景使用的 Skybox。如果我们不想让场景中的物体接受任何默认的反射光照，可以把反射源设置为自定义（即 **Custom**），并把自定义的 Cubemap 保留为空即可（另一种方式是直接把场景使用的 Skybox 设置为空），如图 18.12 右图所示。但为了得到更加逼真的渲染结果，我们通常是不会这样做的。在渲染实现上，即便场景中没有任何光源，Unity 在内部仍然会调用 ForwardBase Pass（假设使用的是前向渲染路径的话），并使用反射的光照信息来填充光源信息，再进行基于物理的渲染计算。读者可以通过帧调试器（Frame Debugger）来查看渲染过程。需要注意的是，这里设置的反射源是默认的反射源，如果我们在场景中添加了其他反射探针（Reflection Probes，见 18.3.2 节），物体可能会使用其他反射源。当默认反射源是 Skybox 时，Unity 会由场景使用的 Skybox 生成一个 Cubemap，我们可以通过 **Resolution** 选项来控制它每个面的分辨率。

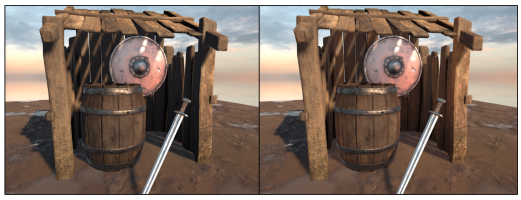
除了 Standard Shader 外，Unity 还引入了一个重要的流水线——实时**全局光照（Global Illumination，GI）**流水线。使用 GI，场景中的物体不仅可以受直接光照的影响，还可以接受间接光照的影响。直接光照指的是那些直接把光照射到物体表面的光源，在本书之前的章节中，我们使用的都是直接光照来渲染场景中的物体。但在现实生活中，物体还会受到间接光照的影响。例如，想象一个红色墙壁旁边放置了一个球体，尽管墙壁本身不发光，但球体靠近墙的一面仍会有少许的红色，这是由于红色墙壁把一些间接光照投射到了球体上。在 Unity 中，间接光照指的就是那些被场景中其他物体反弹的光，这些间接光照会受反弹光的表面的颜色影响（例如之前例子中的红色的墙壁），这些表面会在反弹光线时把自身表面的颜色添加到反射光的计算中。在 Unity 5 中，我们可以使用这些直接光照和间接光照来创建更加真实的视觉效果。

下面，我们首先设置场景使用的**直接光照**——一个平行光。在 PBR（Physically Based Rendering）中，想要让渲染效果更加真实可信，我们需要保证平行光的方向和 Skybox 中的太阳或其他光源的位置一致，使得物体产生的光照信息可以与 Skybox 互相吻合。有时，我们可能会使用一张耀斑纹理（Flare Texture）来模拟太阳等光源，此时我们同样需要确保平行光的方向与耀斑纹理的位置一致。与之类似的还有平行光的颜色，我们应该尽量让平行光的颜色和场景环境相匹配。例如，在图 18.10 的左图中，场景的光照环境为日落时分，因此平行光的颜色为浅黄色，如图 18.13 所示，而在图 18.10 的右图中，场景的光照环境更接近傍晚，此时平行光的颜色为淡蓝色。我们还在 Skybox 的材质面板上调整天空的旋转角度及曝光度，来调整场景的背景。

在平行光面板的烘焙选项（即 **Baking**）中，我们选择了 Realtime 模式，这意味着，场景中受平行光影响的所有物体都会进行实时的光照计算，当光源或场景中其他物体的位置、旋转角度等发生变化时，场景中的光照结果也会随之变化。然而，实时光照往往需要较大的性能消耗，对于移动平台这样资源比较短缺的平台，我们可以选择 Baked 模式，此时，Unity 会把该光源的光照效果烘焙到一张光照纹理（lightmap）中，这样我们就不用实时为物体计算复杂的光照，而只需要通过纹理采样来得到光照结果。选择烘焙模式的缺点在于，如果场景中的物体发生了移动，但是它的阴影等光照效果并不会发生变化。烘焙选项中的 Mix 模式则允许我们混合使用实时模式和烘焙模式，它会把场景中的静态物体（即那些被标识为 Static 的物体）的光照烘焙到光照纹理中，但仍然会对动态物体产生实时光照。

Unity 5 引入了实时间接光照的功能，在这个系统下，场景中的直接光照会在场景中各个物体之间来回反射，产生**间接光照**。正如我们之前讲到的，间接光照可以让那些没有直接被光源照亮的物体同样可以接受到一定的光照信息，这些光照是由它周围的物体反射到它的表面上的。当一条光线从光源被发射出来后，它会与场景中的一些物体相交，第一个和光线相交的物体受到的光照即为直接光照。当得到直接光照在该物体上的光照结果后，该物体还会继续反射该光线，从而对其他物体产生间接光照。此后与该光线相交的物体，就会受到间接光照的影响，同时它们也会继续反射。当经过多次反射后，该光线最后完全消失。这些间接光照的强度是由 GI 系统计算得到的默认亮度值。图 18.13 所示的光源面板中的 **Bounce Intensity** 参数可以让我们调节这些间接光照的强度。当我们把它设为 0 时，意味着一条光线仅会和一个物体相交，不再被继续反射，也就是说，场景中的物体只会受到直接光照的影响。图 18.14 显示了 Bounce Intensity 分别为 0 和 8 时，场景的渲染结果，注意其中阴影部分的细节。

▲图 18.13　使用的平行光



▲图 18.14　左边：将 Bounce Intensity 设置为 0，物体不再受到间接光照的影响，木屋内阴影部分的可见细节很少。右边：将 Bounce Intensity 设为 8，阴影部分的细节更加清楚

除了上述调整单个光源的间接光照强度，我们也可以对整个场景的间接光照强度进行调整。这是按照图 18.11 所示的光照面板来实现的。在光照面板的 Scene 标签页下，我们可以调整 General

GI 参数块中的 Bounce Boost 参数来控制场景中反射的间接光照的强度，它会和单个光源的 Bounce Intensity 参数来一起控制间接光照的反射强度。除此之外，把 Indirect Intensity 参数调大同样可以增大间接光照的强度。需要注意的是，间接光照还有可能来自一些自发光的物体。

18.3.2　放置反射探针

回忆我们在 10.1 节中讲到的环境映射，在实时渲染中，我们经常会使用 Cubemap 来模拟物体的反射效果。例如，在赛车游戏中，我们需要对车身或车窗使用反射映射的技术来模拟它们的反光材质。然而，如果我们永远使用同一个 Cubemap，那么，当赛车周围的场景发生较大变化时，就很容易出现“穿帮镜头”，因为车身或车窗的环境反射并没有随着环境变化而发生变化。一种解决办法是可以在脚本中控制何时生成从当前位置观察到的 Cubemap，而 Unity 5 为我们提供了一种更加方便的途径，即使用**反射探针（Reflection Probes）**。反射探针的工作原理和光照探针（Light Probes）类似，它允许我们在场景中的特定位置上对整个场景的环境反射进行采样，并把采样结果存储在每个探针上。当游戏中包含反射效果的物体从这些探针附近经过时，Unity 会把从这些邻近探针存储的反射结果传递给物体使用的反射纹理。如果物体周围存在多个反射探针，Unity 还会在这些反射结果之间进行插值，来得到平滑渐变的反射效果。实际上，Unity 会在场景中放置一个默认的反射探针，这个反射探针存储了对场景使用的 Skybox 的反射结果，来作为场景的环境光照（见 18.3.1 节）。如果我们需要让场景中的物体包含额外的反射效果，就需要放置更多的反射探针。

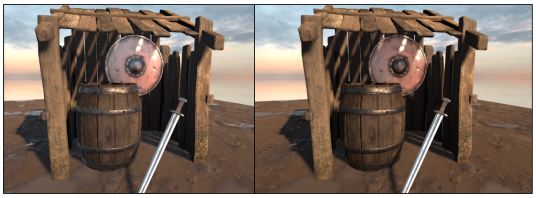
反射探针同样有 3 种类型：Baked，这种类型的反射探针是通过提前烘焙来得到该位置使用的 Cubemap 的，在游戏运行时反射探针中存储的 Cubemap 并不会发生变化。需要注意的是，这种类型的反射探针在烘焙时同样只会处理那些静态物体（即那些被标识为 Reflection Probe Static 的物体）；Realtime，这种类型则会实时更新当前的 Cubemap，并且不受静态物体还是动态物体的影响。当然，这种类型的反射探针需要花费更多的处理时间，因此，在使用时应当非常小心它们的性能。幸运的是，Unity 允许我们从脚本中通过触发来精确控制反射探针的更新；最后一种类型是 Custom，这种类型的探针既可以让我们从编辑器中烘焙它，也可以让我们使用一个自定义的 Cubemap 来作为反射映射，但自定义的 Cubemap 不会被实时更新。

我们在本节使用的场景中放置了 3 个反射探针，它们的类型都是 Baked（前提是我们把场景中的物体标识成了 Static）。使用反射探针前后的对比效果如图 18.15 所示。

需要注意的是，在放置反射探针时，我们选取的位置并不是任意的。通常来说，反射探针应该被放置在那些具有明显反射现象的物体的旁边，或是一些墙角等容易发生遮挡的物体周围。在本例使用的场景中，木屋内的盾牌具有比较明显的反射效果，而盾牌本身又被木屋遮挡，因此，其中一个反射探针的位置就在盾牌附近。当我们放置好探针后，我们还需要为它们定义每个探针的影响区域，当反射物体进入到这个区域后，反射探针就会对物体的反射产生影响。通常情况下，反射探针的影响区域之间往往会有所重叠，例如，本例中盾牌附近的反射探针和另外两个（一个在木屋前方，一个在木屋后方）的影响区域都有所重叠。此时，Unity 会计算反射物体的包围盒与这些重叠区域的交叉部分，并据此来选择使用的反射映射。如果当前的目标平台使用的是 SM 3.0 及以上的话，Unity 还可以允许我们在这些互相重叠的反射探针之间进行混合，来实现平缓的反射过渡效果。

使用 Unity 内置的反射探针的另一个好处是，我们可以模拟**互相反射（interreflections）**。我们曾在 10.1 节中讲到使用传统的 Cubemap 方法无法模拟互相反射的效果。例如，假设场景中有两面互相面对面的镜子，在理想情况下，它们不仅会反射自己对面的那面镜子，也会反射那面镜子里反射的图像。只要反射光线没有被完全吸收，反射就会一直进行下去。要实现这种效果，就

需要追踪光线的反射轨迹，这是传统的反射方法所无法实现的。Unity 5 引入的 GI 系统让这种效果变成了可能，我们在本书资源的 Scene_18_3_2 场景中展示了这样的一个例子，如图 18.16 所示。我们可以在图 18.16 中看到，两个金属反射的图像包含了两次互相反射的效果。



▲图 18.15　左边：未使用反射探针。右边：在场景中放置了两个反射探针，注意墙上的盾牌与左图的差别

▲图 18.16　使用反射探针实现相互反射的效果

在图 18.16 所示的场景中，我们在每个金属球的位置处放置了一个反射探针，并把每个金属球上的 Mesh Renderer 组件中的 Reflection Probes 设置为 Simple，这样保证它们只会使用离它们最近的一个反射探针。默认情况下，反射探针只会捕捉一次反射，也就是说，左边金属球使用的反射探针只会捕捉到由右边的金属球第一次反射过来的光线。但在理想情况下，反射过来的光线会继续被左边的金属球反射，并对右边的金属球造成影响。Unity 允许我们控制物体之间这样来回反射的次数，这可以通过改变图 18.11 中的 Reflection Bounces 参数来实现。在图 18.16 所示的场景中，我们把该值设为了 2。

然而，正如本节一开始所提到的，使用反射探针往往会需要更多的计算时间。这些探针实际上也是通过在它的位置上放置一个摄像机，来渲染得到一个 Cubemap。如果我们把反弹次数设置的很大，或是使用实时渲染，那么这些探针很可能会造成性能瓶颈。更多关于如何优化反射探针以及它的高级用法，读者可以参见 Unity 的官方手册（http://docs.unity3d.com/Manual/ReflectionProbes.html）。

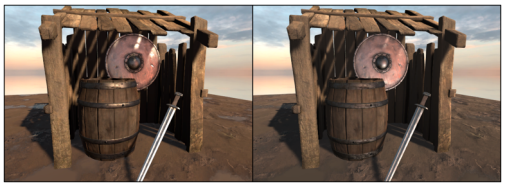
18.3.3　调整材质

要得到真实可信的渲染效果，我们需要为场景中的物体指定合适的材质。需要再次提醒读者的是，基于物理的渲染并不意味着一定要模拟像照片真实的效果。基于物理的渲染更多的好处在于，可以让我们的场景在各种光照条件下都能得到令人满意的效果，同时不需要频繁地调整材质参数。

在 Unity 中，要想和全局光照、反射探针等内置功能良好地配合来得到出色的渲染结果，就需要使用 Unity 内置的 Standard Shader。我们已经在 18.2.2 节中学习了如何针对不同类别的物体来调整它们使用的材质属性。在本例中，我们使用了更复杂的纹理和模型，它们都来自于 Unity 官方的示例项目 **Viking Village**。这些材质可以为读者制作自己的材质提供一些参考，例如，场景中所有物体都使用了高光反射纹理（Specular Texture）、遮挡纹理（Occlusion Texture）、法线纹理（Normal Texture），一些材质还使用了细节纹理来提供更多的细节表现。

18.3.4　线性空间

在使用基于物理的渲染方法渲染整个场景时，我们应该使用**线性空间（Linear Space）**来得到最好的渲染效果。默认情况下，Unity 会使用伽马空间（Gamma Space），如果要使用线性空间的话，我们需要在 Edit→Project Settings→Player→Other Settings→Color Space 中选择 Linear 选项。图 18.17 显示了分别在线性空间和伽马空间下场景的渲染结果。



▲图 18.17　左边：在线性空间下的渲染结果。右边：在伽马空间下的渲染结果

从图 18.17 中可以看出，使用线性空间可以得到更加真实的效果。但它的缺点在于，需要一些硬件支持来实现线性计算，但一些移动平台对它的支持并不好。这种情况下，我们往往只能退而求其次，选择伽马空间进行渲染和计算。

那么，线性空间、伽马空间到底是什么意思？为什么线性空间可以得到更加真实的效果呢？这就需要介绍**伽马校正（Gamma Correction）**的相关内容了。实际上，当我们在默认的伽马空间下进行渲染计算时，由于使用了非线性的输入数据，导致很多计算都是在非线性空间下进行的，这意味着我们得到的结果并不符合真实的物理期望。除此之外，由于输出时没有考虑显示器的显示伽马的影响，会导致渲染出来的画面整体偏暗，总是和真实世界不像。

尽管在 Unity 中我们可以通过之前所说的步骤直接选择在线性空间进行渲染，Unity 会在背后为我们照顾好一切，但了解伽马校正的原理对我们理解渲染计算有很大帮助，读者可以在 18.4.2 节找到更多的解释。

18.4　答疑解惑

在上面的内容中，我们首先介绍了 PBS 实现的数学和理论基础，并简单概括了 Unity 中 Standard Shader 的实现原理，以及如何使用它来为不同类型的物体调整适合它们的材质参数。随后，我们通过一个更加复杂的场景，来展示如何在 Unity 中使用环境光照、实时光源、反射探针以及 Standard Shader 来渲染一个基于物理渲染的场景。但我们相信，读者在读完后仍有很多困惑，考虑到内容的连贯性，我们未能在文中对某些概念进行展开。在本节中，我们将对一些重要的概念进行更为深入地解释。

18.4.1　什么是全局光照

在上面的内容中，我们可以发现全局光照对得到真实的渲染结果有着举足轻重的作用。全局光照，指的就是模拟光线是如何在场景中传播的，它不仅会考虑那些直接光照的结果，还会计算光线被不同的物体表面反射而产生的间接光照。在使用基于物理的着色技术时，当渲染表面上一点时，我们需要计算该点的半球范围内所有会反射到观察方向的入射光线的光照结果，这些入射光线中就包含了直接光照和间接光照。

通常来讲，这些间接光照的计算是非常耗时间的，通常不会用在实时渲染中。一个传统的方法是使用光线追踪，来追踪场景中每一条重要的光线的传播路径。使用光线追踪能得到非常出色的画面效果，因此，被大量应用在电影制作中。但是，这种方法往往需要大量时间才能得到一帧，并不能满足实时的要求。

Unity 采用了 Enlighten 解决方案来让全局光照能够在各种平台上有不错的性能表现。事实

上，Enlighten 也已经被集成在虚幻引擎（Unreal Engine）中，它已经在很多 3A 大作中展现了自身强大的渲染能力。总体来讲，Unity 使用了实时+预计算的方法来模拟场景中的光照。其中，实时光照用于计算那些直接光源对场景的影响，当物体移动时，光照也会随之发生变化。但正如我们之前所说，实时光照无法模拟光线被多次反射的效果。为了得到更加真实的渲染效果，Unity 又引入了预计算光照的方法，使得全局光照甚至在一些高端的移动设备上也可以达到实时的要求。

预计算光照包含了我们常见的光照烘焙，也就是指我们把光源对场景中静态物体的光照效果提前烘焙到一张光照纹理中，然后把这张光照纹理直接贴在这些物体的表面，来得到光照效果。这些光照纹理不仅存储了直接光照的结果，还包含了那些由物体反射得到的间接光照。但是，这些光照纹理无法在游戏运行时不断更新，也就是说，它们是静态的。不过这种方法的确为移动平台的复杂光照模拟提供了一个有效途径。以上提到的这些技术很多读者都已非常熟悉，并可能已经在实际工作中大量使用了它们。

由于静态的光照烘焙无法在光照条件改变时更新物体的光照效果，因此，Unity 使用了**预计算实时全局光照（Precomputed Realtime GI）**为我们提供了一个解决途径，来动态地为场景实时更新复杂的光照结果。正如我们之前看到的，使用这种技术我们可以让场景中的物体包含丰富的全局光照效果，例如多次反射等，并且这些计算都是实时的，可以随着光源和物体的移动而发生变化。这是使用之前的实时光照或烘焙光照所无法实现的。

那么，这些是如何实现的呢？它们实际上都利用了一个事实—— 一旦物体和光源的位置被固定了，这些物体对光线的反弹路径以及漫反射光照（我们假设漫反射光照在各个方向的分布是相同的）也是固定的，也就是说是和摄像机无关的。因此，我们可以使用预计算方法来把这些物体之间的关系提前计算出来，而在实时运行时，只要光源的位置（光源的颜色是可以实时变化的）不变，即便改变了光源颜色和强度、物体材质属性（指的是漫反射和自发光相关的属性），这些信息就一直有效，不需要实时更新。在预计算阶段，Enlighten 会在由所有静态物体组成的场景上，进行简化的“光线追踪”过程。在这个过程中 Enlighten 会自动把场景分割成很多个子系统，它并不是为了得到精确的光照效果，而是为了得到场景中物体之间的关系。需要注意的是，这些预计算都是在静态物体上进行的，因此，为了利用上述的预计算方法，我们至少需要把场景中的一个物体标识为 Static（至少需要把 Lightmap Static 勾选上）。一个例外是物体的高光反射，这是和摄像机的位置相关的，Unity 的解决方案是使用反射探针，正如我们之前看到的那样。对于动态移动的物体来说，我们可以使用光照探针来模拟它的光照环境。因此，在实时运行时，Unity 会利用预计算得到的信息来计算光照信息，并把它们存储在额外的光照纹理、光照探针或 Cubemap 中，再和物体材质进行必要的光照计算，得到最后的渲染效果。

Unity 全新的全局光照解决方案可以大大提高一些基于 PC/游戏机等平台的大型游戏的画面质量，但如果要在移动平台上使用仍需要非常小心它的性能。一些低端手机是不适合使用这种比较复杂的基于物理的渲染，不过，Unity 会在后续的版本中持续更新和优化。而且随着手机硬件的发展，未来在移动平台上大量使用 PBS 也已经不再是遥不可及的梦想了。更多关于 Unity 中全局光照的内容，读者可以在 Unity 官方手册的**全局光照**（http://docs.unity3d.com/Manual/GIIntro.html）一文中找到更多内容，本章最后的扩展阅读部分也会给出更多的学习资料。

18.4.2 什么是伽马校正

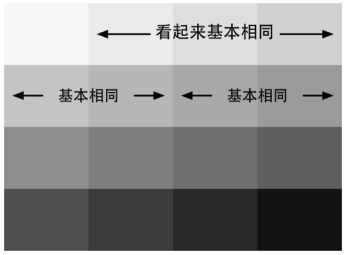
我们在 18.3.4 节中讲到，要想渲染出更符合真实光照环境的场景就需要使用线性空间。而 Unity 默认的空间是伽马空间，在伽马空间下进行渲染会导致很多非线性空间下的计算，从而引入

了一些误差。而要把伽马空间转换到线性空间，就需要进行**伽马校正（Gamma Correction）**。

相信很多读者都听过伽马校正这个名词，但对于伽马校正是什么、为什么要有它、怎么使用它都存在着很多疑问。伽马校正中的伽马一词来源伽马曲线。通常，伽马曲线的表达式如下：

$$L_{out} = L_{in}^{\gamma}$$

其中指数部分的发音就是伽马。最开始的时候，人们使用伽马曲线来对拍摄的图像进行**伽马编码（gamma encoding）**。事情的起因可以从在真实环境中拍摄一张图片说起。摄像机的原理可以简化为，把进入到镜头内的光线亮度编码成图像（例如一张 JEPG）中的像素。如果采集到的亮度是 0，像素就是 0 亮度是 1，像素就是 1 亮度是 0.5，像素就是 0.5。如果我们只用 8 位空间来存储像素的每个通道的话，这意味着 0～1 区间可以对应 256 种不同的亮度值。但是，后来人们发现，人眼有一个有趣的特性，就是对光的灵敏度在不同亮度上是不一样的。在正常的光照条件下，人眼对较暗区域的变化更加敏感，如图 18.18 所示。

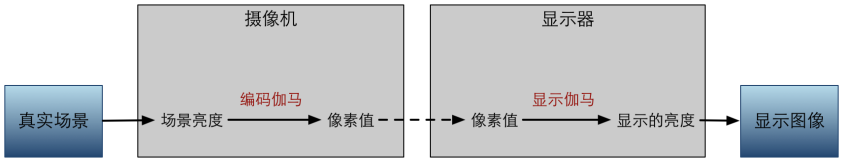


▲图 18.18　人眼更容易感知暗部区域的变换，

图 18.18 说明了一件事情，亮度上的线性变化对人眼感知来说是非均匀的。Youtube 上有一个名为 **Color is Broken** 的非常有趣的视频，在这个视频中，作者用了一个非常生动的例子来说明这个现象。当一个屋子的光照由一盏灯增加到两盏灯的时候，人眼对这种亮度变化的感知性要远远大于从 101 盏灯增加到 102 盏灯的变化，尽管从物理上来说这两种变化基本是相同的。那么，这和之前讲的拍照有什么关系呢？如果使用 8 位空间来存储每个通道的话，我们仍然把 0.5 亮度编码成值为 0.5 的像素，那么暗部和亮部区域我们都使用了 128 种颜色来表示，但实际上，对亮部区域使用这么多颜色是种存储浪费。一种更好的方法是，我们应该把把更多的空间来存储更多的暗部区域，这样存储空间就可以被充分利用起来了。摄影设备如果使用了 8 位空间来存储照片的话，会使用大约为 0.45 的编码伽马来对输入的亮度进行编码，得到一张编码后的图像。因此，图像中 0.5 像素值对应的亮度其实并不是 0.5，而大约为 0.22。这是因为：

$$0.5 \approx 0.22^{0.45}$$

如上所见，对拍摄图像使用的伽马编码使得我们可以充分利用图像的存储空间。但当把图片放到显示器里显示时，我们应该对图像再进行一次解码操作，使得屏幕输出的亮度和捕捉到的亮度是符合线性的。这时，人们发现了一个奇妙的巧合——CRT 显示器本身几乎已经自动做了这个解码操作！这又从何说起呢？在早期，CRT（Cathode Ray Tube，阴极射线管）几乎是唯一的显示设备。这类设备的显示机制是，使用一个电压轰击它屏幕上的一种图层，这个图层就可以发亮，我们就可以看到图像了。但 CRT 显示器有一个特性，它的输入电压和显示出来的亮度关系不是线性的，也就是说，如果我们把输入电压调高两倍，屏幕亮度并没有提高两倍。我们把显示器的这个伽马曲线称为**显示伽马（diplay gamma）**。非常巧合的是，CRT 的显示伽马值大约就是编码伽马的倒数。CRT 显示器的这种特性，正好补偿了图像捕捉设备的伽马曲线，人们想，“天呐，太棒了，我们不需要做任何调整就可以让拍摄的图像在电脑上看起来和原来的一样了！”虽然现在 CRT 设备很少见了，并且后来出现的显示设备有着不同的伽马响应曲线，但是，人们仍在硬件上做了调整来提供兼容性。 图 18.19 展示了编码伽马和显示伽马在图像捕捉和显示时的作用。



▲图 18.19 编码伽马和显示伽马

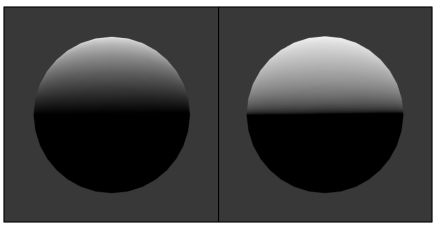
随后，微软联合爱普生、惠普提供了 sRGB 颜色空间标准，推荐显示器的显示伽马值为 2.2，并配合 0.45 的编码伽马就可以保证最后伽马曲线之间可以相互抵消（因为 2.2×0.45≈1）。绝大多数的摄像机、PC 和打印机都使用了上述的 sRGB 标准。

读到现在，读者可能还是有所疑问，这和渲染有什么关系？答案是关系很大。事实上，由于游戏界长期以来都忽视了伽马校正的问题，造成了我们渲染出来的游戏总是暗沉沉的，总是和真实世界不像。由于编码伽马和显示伽马的存在，我们一不小心就可能在非线性空间下进行计算，或是使得输出的图像是非线性的。

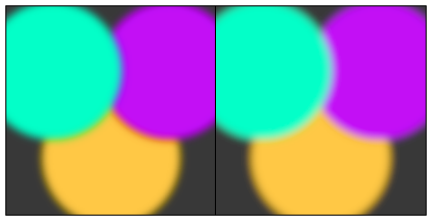
对于输出来说，如果我们直接输出渲染结果而不进行任何处理，在经过显示器的显示伽马处理后，会导致图像整体偏暗，出现失真的状况。我们在本书资源的 Scene_18_4_2_a 显示了伽马**对光照效果的影响**。在场景 Scene_18_4_2_a 中，我们放置了一个球体，并把场景中的环境光照设为全黑，再把平行光的方向设置为从上方直接射到球体表面，球体使用的材质为内置的漫反射材质。图 18.20 显示了在伽马空间和线性空间下的渲染结果。

从图 18.20 可以看出，伽马空间下的渲染结果整体偏暗，一些读者甚至认为这看起来更加正确。然而，实际此时屏幕输出的亮度和球面的光照结果并不是线性的。假设球面上有一点 A，它的法线和光线方向成 60°，还有一点 B，它的法线和光线方向成 90°。那么，在 Shader 中计算漫反射光照时，我们会得出 A 的输出是（0.5, 0.5, 0.5），B 的输出是（1.0, 1.0, 1.0）。在图 18.20 的左图中，我们没有进行伽马校正，因此，由于显示器存在显示伽马就引入了非线性关系，也就是说 A 点的亮度其实并不是 B 亮度的一半，而约为它的 1/4。在图 18.20 的右图中，我们使用了线性空间，Unity 会在把像素写入颜色缓冲前进行一次伽马校正，来抵消屏幕的显示伽马的作用，此时得到屏幕亮度才是真正跟像素值成正比的。

伽马的存在还会**对混合造成影响**。在场景 Scene_18_4_2_b 中演示了一个简单的场景来说明这个现象。在场景 Scene_18_4_2_b 中，我们放置了 3 个互相重叠的圆，它们使用的材质均为简单的透明混合材质，并使用了一个边界模糊的圆作为输入纹理。场景在伽马空间和线性空间下的效果如图 18.21 所示。



▲图 18.20 左边：伽马空间下的渲染结果，右边：线性空间下的渲染结果



▲图 18.21 左边：伽马空间下的混合结果，右边：线性空间下的混合结果

在图 18.21 左图所示的伽马空间下，我们可以看到在绿色和红色的混合边界处出现了不正常的蓝色渐变。而正确的混合结果应该是如图 18.21 右边图所示的从绿色到红色的渐变。除此之外，

我们也可以看到图 18.21 左边图中交叉的边界似乎都变暗了。这是因为在混合后进行输出时，显示器的显示伽马导致接缝处颜色变暗。

实际上，渲染中非线性输入最有可能的来源就是纹理。为了充分利用存储空间，大多数图像文件都进行了提前的校正，即已经使用了一个编码伽马对像素值编码。但这意味着它们是非线性的，如果我们在 Shader 中直接使用纹理采样值就会造成在非线性空间的计算，使得结果和真实世界的结果不一致。我们在使用多级渐远纹理（mipmaps）时也需要注意。如果纹理存储在非线性空间中，那么在计算多级渐远纹理时就会在非线性空间里计算。由于多级渐远纹理的计算是种线性计算——即采样的过程，需要对某个方形区域内的像素取平均值，这样就会得到错误的结果。正确的做法是，我们要把非线性的纹理转换到线性空间后再计算多级渐远纹理。

如上所说，伽马的存在使得我们很容易得到非线性空间下的渲染结果。在游戏渲染中，我们应该保证所有的输入都被转换到了线性空间下，并在线性空间下进行各种光照计算，最后在输出前通过一个编码伽马进行伽马校正后再输出到颜色缓冲中。Untiy 的颜色空间设置就可以满足我们的需求。当我们选择伽马空间时，实际上就是“放任模式”，不会对 Shader 的输入进行任何处理，即使输入可能是非线性的；也不会对输出像素进行任何处理，这意味着输出的像素会经过显示器的显示伽马转换后得到非预期的亮度，通常表现为整个场景会比较昏暗。当选择线性空间时，Unity 会把输入纹理设置为 sRGB 模式，在这种模式下，硬件在对纹理进行采样时会自动将其转换到线性空间中；并且，GPU 会在 Shader 写入颜色缓冲前自动进行伽马校正或是保持线性在后面进行伽马校正，这取决于当前的渲染配置。如果我们开启了 HDR（见 18.4.3 节）的话，渲染就会使用一个浮点精度的缓冲。这些缓冲有足够的精度不需要我们进行任何伽马校正，此时所有的混合和屏幕后处理都是在线性空间下进行的。当渲染完成要写入显示设备的后备缓冲区（back buffer）时，再进行一次最后的伽马校正。如果我们没有使用 HDR，那么 Unity 就会把缓冲设置成 sRGB 格式，这种格式的缓冲就像一个普通的纹理一样，在写入缓冲前需要进行伽马校正，在读取缓冲时需要再进行一次解码操作。如果此时开启了混合（像我们之前的那样），在每次混合时，硬件会首先把之前颜色缓冲中存储的颜色值转换回线性空间中，然后再与当前的颜色进行混合，完成后再进行伽马校正，最后把校正后的混合结果写入颜色缓冲中。这里需要注意，透明通道是不会参与伽马校正的。

然而，Unity 的线性空间并不是所有平台都支持的，例如，移动平台就无法使用线性空间。此时，我们就需要自己在 Shader 中进行伽马校正。对非线性输入纹理的校正代码通常如下：

```
float3 diffuseCol = pow(tex2D( diffTex, texCoord ), 2.2 );
```

在最后输出前，对输出像素值的校正代码通常如下面这样：

```
fragColor.rgb = pow(fragColor.rgb, 1.0/2.2);
return fragColor;
```

但是，手工对输出像素进行伽马校正会在使用混合时出现问题。这是因为，校正会导致写入颜色缓冲内的颜色是非线性的，这样混合就发生在非线性空间中。一种解决方法是，在中间计算时不要对输出颜色值进行伽马校正，但在最后需要进行一个屏幕后处理操作来对最后的输出进行伽马校正，也就是说我们需要保证伽马校正发生在渲染的最后一步中，但这可能会造成一定的性能损耗。

你会说，伽马这么麻烦，什么时候可以舍弃它呢？如果有一天我们对图像的存储空间能够大大提升，通用的格式不再是 8 位时，例如是 32 位时，伽马也许就会消失。因为，我们有足够多的颜色空间可以利用，不需要为了充分利用存储空间进行伽马编码的工作了。这就是我们下面要讲的 HDR。

18.4.3 什么是 HDR

在使用基于物理的渲染时，我们经常会听到一个名词就是 HDR。HDR 是 High Dynamic Range 的缩写，即高动态范围，与之相对的是**低动态范围（Low Dynamic Range，LDR）**。那么这个动态范围是指什么呢？通俗来讲，动态范围指的就是最高的和最低的亮度值之间的比值。在真实世界中，一个场景中最亮和最暗区域的范围可以非常大，例如，太阳发出的光可能要比场景中某个影子上的点的亮度要高出几万倍，这些范围远远超过图像或显示器能够显示的范围。通常在显示设备使用的颜色缓冲中每个通道的精度为 8 位，意味着我们只能用这 256 种不同的亮度来表示真实世界中所有的亮度，因此，在这个过程中一定会存在一定的精度损失。早期的拍摄设备利用人眼的特点，使用了伽马曲线来对捕捉到的图像进行编码，尽可能充分地利用这些有限的存储空间，这点我们已经在 18.4.2 节解释过了。然而，HDR 的出现给我们带来了新的希望，HDR 使用远远高于 8 位的精度（如 32 位）来记录亮度信息，使得我们可以表示超过 0～1 内的亮度值，从而可以更加精确地反映真实的光照环境。尽管我们最后还是需要把信息转换到显示设备使用的 LDR 内，但中间的计算却可以让我们得到更加真实可信的效果。Nvidia 曾总结过使用 HDR 进行渲染的**动机**：让亮的物体可以真的非常亮，暗的物体可以真的非常暗，同时又可以看到两者之间的细节。

使用 HDR 来存储的图像被称为高动态范围图像（HDRI），例如，我们在 18.3 节中就是使用了一张 HDRI 图像来作为场景的 Skybox。这样的 Skybox 可以更加真实地反映物体周围的环境，从而得到更加真实的反射效果。不仅如此，HDR 对与光照叠加也有非常重要的作用。如果我们的场景中有很多光源或是光源强度很大，那么一个物体在经过多次光照渲染叠加后最终得到的光照亮度很可能会超过 1。如果没有使用 HDR，这些超过 1 的部分全部会截取到 1，使得场景丢失了很多亮部区域的细节。但如果开启了 HDR，我们就可以保留这些超过范围的光照结果，尽管最后我们仍然需要把它们转换到 LDR 进行显示，但我们可以使用**色调映射（tonemapping）**技术来控制这个转换的过程，从而允许我们最大限度地保留需要的亮度细节。

HDR 的使用可以允许我们在屏幕后处理中拥有更多的控制权。例如，我们常常同时使用 HDR 和 Bloom 效果。我们曾在 12.5 节解释了 Bloom 特效的实现原理，Bloom 效果需要检测屏幕中亮度大于某个阈值的像素，把它们提取出来后进行模糊，再叠加到原图像中。但是，如果不使用 HDR 的话，我们只能使用小于 1 的阈值来提取需要的像素，但很多时候我们实际上是需要提取那些非常亮的区域，例如车窗上对太阳的强烈反光。由于没有使用 HDR，这些值实际上很可能和街上一些颜色偏白的区域几乎一样，造成不希望的区域也会出现泛光的效果。如果我们使用 HDR，这些就都可以解决了，我们只需要使用超过 1 的阈值来只提取那些非常亮的区域即可。

总体来说，使用 HDR 可以让我们不会丢失高亮度区域的颜色值，提供了更真实的光照效果，并为一些屏幕后处理提供了更多的控制能力。但 HDR 也有自身的缺点，首先由于使用了浮点缓冲来存储高精度图像，不仅需要更大的显存空间，渲染速度会变慢，除此之外，一些硬件并不支持 HDR。而且一旦使用了 HDR，我们无法再利用硬件的抗锯齿功能。事实上，在 Unity 中如果我们同时打开了硬件的抗锯齿（在 Edit → Project Settings → Quality → Anti Aliasing 中打开）和摄像机的 HDR，Unity 会发出警告来提示我们由于开启了抗锯齿，因此，无法使用 HDR 缓冲。尽管如此，我们可以使用基于屏幕后处理的抗锯齿操作来弥补这一点。

在 Unity 中使用 HDR 也非常简单，我们可以在 Camera 组件面板中打开 HDR 选项即可。此时，场景就会被渲染到一个 HDR 的图像缓冲中，这个缓冲的精度范围可以远远超过 0～1。最后，我们可以再使用一个色调映射的屏幕后处理脚本来把 HDR 图像转换到 LDR 图像进行显示。读者

可以在 Unity 官方手册中的**高动态范围渲染**一节（http://docs.unity3d.com/Manual/HDR.html）以及本章最后的扩展阅读中找到更多的内容。

18.4.4　那么，PBS 适合什么样的游戏

在把 PBS 引入当前的游戏项目之前，我们需要权衡一下它的优缺点。需要再次提醒读者的是，PBS 并不意味着游戏画面需要追求和照片一样真实的效果。事实上，很多游戏都不需要刻意去追求与照片一样的真实感，玩家眼中的真实感大多也并不是如此。PBS 的优点在于，我们只需要一个万能的 Shader 就可以渲染相当一大部分类型的材质，而不是使用传统的做法为每种材质写一个特定的 Shader。同时，PBS 可以保证在各种光照条件下，材质都可以自然地和光源进行交互，而不需要我们反复地调整材质参数。

然而，在使用 PBS 时我们也需要考虑到它带来的代价。如上面提到的，PBS 往往需要更复杂的光照配合，例如大量使用光照探针和反射探针等。而且 PBS 也需要开启 HDR 以及一些必不可少的屏幕特效，例如抗锯齿、Bloom 和色调映射，如果这些屏幕特效对当前游戏来说需要消耗过多的性能，那么 PBS 就不适合当前的游戏，我们应该使用传统的 Shader 来渲染游戏。使用 PBS 对美工人员来说同样是个挑战。美术资源的制作过程和使用传统的 Shader 有很大不同，普通的法线纹理+高光反射纹理的组合不再适用，我们需要创建更细腻复杂的纹理集，包括金属值纹理、高光反射纹理、粗糙度纹理、遮挡纹理，有些还需要使用额外的细节纹理来给材质添加更多的细节表面。除了使用图片扫描的传统辅助方法外，这些纹理的制作通常还需要更专业的工具来绘制，例如 **Allegorithmic Substance Painter** 和 **Quixel Suite**。

18.5　扩展阅读

Unity 官方提供了很多学习 PBS 的资料。在 **Unity** 官方博客**中的全局光照**一文[①]中，简明地阐述了全局光照的解决方案。在另外两篇博客[②③]中，介绍了 Standard Shader 的用法和注意事项。在 Unity 官方教程的 **Graphics** 分项[④]下，也有很多关于 PBS 的教程，例如，在预计算实时 GI 的系列文章[⑤]中，介绍了 Unity 5 的 Enlighten 系统中预计算实时 GI 的实现原理，以及如何正确地把它应用在大型项目中来提升画面效果并减少烘焙时间。在 **Unity 5 的光照概览**[⑥]中，介绍了 Unity 5 中使用的各种全局光照技术；在 **Standard Shader** 的视频教程[⑦]中，Unity 介绍了 Standard Shader 的基本用法以及和光照之间的配合。

官方项目也是很好的学习资料。Unity 开放了基于物理着色器的示例项目 **Viking Village** 以及两个更小的示例项目 **Shader Calibration Scene** 和 **Corridor Lighting Example** 来着重介绍如何使用 Unity 5 全新的 Standard Shader 和全局光照系统。看过 Unity 5 宣传视频的读者想必对 Unity 5 制作出来的电影短片 **The Blacksmith** 印象深刻，尽管 Unity 没有开放出完整的工程，但把许多关键的技术实现放到了资源商店里，例如，人物角色使用的 Shader、头发使用的 Shader、人物阴影、大气次散射等。与之类似的还有 **Adam** 的相关资源[⑧]，这些都是非常好的学习资料，读者可以在

① https://blogs.unity3d.com/2014/09/18/global-illumination-in-unity-5/
② https://blogs.unity3d.com/2015/02/18/working-with-physically-based-shading-a-practical-approach/
③ https://blogs.unity3d.com/2014/10/29/physically-based-shading-in-unity-5-a-primer/
④ https://unity3d.com/learn/tutorials/topics/graphics
⑤ https://unity3d.com/learn/tutorials/topics/graphics/introduction-precomputed-realtime-gi
⑥ https://unity3d.com/learn/tutorials/topics/graphics/lighting-overview
⑦ https://unity3d.com/learn/tutorials/topics/graphics/standard-shader
⑧ https://blogs.unity3d.com/2016/11/01/adam-demo-executable-and-assets-released/

Asset Store 上找到并下载这些资源。

近年来，Unity 在 Unite 和 SIGGRAPH 等大会上也分享不少关于 PBS 的技术资料。在 Unite 2016 上，Mathieu Muller 在他的演讲 **A Step by Step Guide to Making a Scene That Looks Like The Adam Demo** 中简要介绍了如何使用 Unity 开放出来的 **Adam** 资源在 Unity 5 中一步步渲染出视频中的一个画面；在 Unite 2015 上的 **Advanced Global Illumination in Unity 5** 演讲中，官方人员简要介绍了全局光照的相关概念以及如何使用 Enlighten 来得到高质量的实时渲染效果；来自 Allegorithmic 的技术人员在 Unite 2015 上做了名为 **PBR Workflows & Guidelines for Unity 5** 的演讲，他介绍了如何使用 Substance 系列产品来制作适合于 PBS 的高质量贴图；在 Unite 2014 会议上，Anton Hand 在他的演讲（名称：**Best Practices For Physically Based Content Creation**）中给出了很多关于如何创建 PBS 中使用的资源的最佳实践；Renaldas Zioma 和 Erland Körner 则在 Unite 2014 上讲解了如何在 Unity 5 中更加有效地使用 PBS（名称：**Mastering Physically Based Shading**）。

在 SIGGRAPH 2016 上，Eric Heitz 和 Stephen Hill 在他们名为 **Real-Time Area Lighting: a Journey from Research to Production** 的演讲中分享了如何实现实时面光源的渲染，这也是 Unity 的 **Adam Demo** 中使用的技术；Sébastien Lagarde 在 SIGGRAPH 2016 上介绍了 Unity 的技术人员是如何制作全景 HDRI 图像的（名称：**An Artist-Friendly Workflow for Panoramic HDRI**）；在 SIGGRPAH 2015 上，来自 Unity 的技术人员分享了 **The Blacksmith** 的环境制作过程（名称：**Authoring of Procedural Environments in "The Blacksmith"**）；Renaldas Zioma 在他的 SIGGRPAH 2015 演讲 **Optimizing PBR for Mobile** 中介绍了如何在移动平台上应用和优化 PBR。

如果读者希望更深入地学习 PBS 的理论和实践，可以在近年来的 SIGGRAPH 课程上找到非常丰富的资料。SIGGRAPH 自 2006 年起开始出现与 PBS 相关的课程，连续 5 年（2012～2016）在名为 **Physically Based Shading in Theory and Practice** 的课程中由来自各大游戏公司和影视公司的技术人员分享他们在 PBS 上的实践。例如在 2012 年的课程上[2]，Disney 公布了他们在离线渲染时使用的 BRDF 模型，这也是 Unity 等很多游戏引擎使用的 PBR 的理论基础；在 2013 的课程上，Brian Karis 在名为 **Real Shading in Unreal Engine 4** 的演讲中分享了虚幻引擎 4（Unreal Engine 4）中的 PBR 实现；在 2014 年的课程上，Sébastien Lagarde 和 Charles de Rousiers 在名为 **Moving Frostbite to PBR** 的演讲[①]中则介绍了寒霜引擎（Frostbite）中的 PBR 实现。Kostas Anagnostou 在他的文章[②]中整理了非常多的关于 PBR 的相关文章，包括我们上面提到的 SIGGRAPH 课程，**强烈建议**有兴趣的读者去浏览一番。

国内的相关资料则相对较少。龚敏敏在他的 KlayGE 引擎中引入了 PBS，并写了系列博文[③]来简明地阐述其中的理论基础；在知乎专栏 **Behind the Pixels**[④]中，作者文刀秋二给出了三篇关于基于物理着色的系列文章；FOXhunt 在他的专栏文章**如何看懂这些"该死的"图形学公式**一文[⑤]中以比较详细易懂的方式解释和推导了 BRDF 模型中的各种参数和公式。

18.6 参考文献

[1] Hoffman N. Background: physics and math of shading[C]//Fourth International Conference and Exhibition on Computer Graphics and Interactive Techniques, Anaheim, USA. 2013: 21-25.

① http://www.frostbite.com/2014/11/moving-frostbite-to-pbr/
② https://interplayoflight.wordpress.com/2013/12/30/readings-on-physically-based-rendering/
③ http://www.klayge.org/tag/pbr/
④ http://zhuanlan.zhihu.com/graphics
⑤ https://zhuanlan.zhihu.com/p/21489591?refer=c_37032701

[2] Burley B, Studios W D A. Physically-based shading at disney[C]//ACM SIGGRAPH. 2012: 1-7.

[3] Walter B, Marschner S R, Li H, et al. Microfacet models for refraction through rough surfaces[C]//Proceedings of the 18th Eurographics conference on Rendering Techniques. Eurographics Association, 2007: 195-206. http://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.html

[4] Beckmann P, Spizzichino A. The scattering of electromagnetic waves from rough surfaces[J]. Norwood, MA, Artech House, Inc., 1987, 511 p., 1987, 1.

[5] Torrance K E, Sparrow E M. Theory for off-specular reflection from roughened surfaces[J]. JOSA, 1967, 57(9): 1105-1112.

[6] Smith B G. Geometrical shadowing of a random rough surface[J]. Antennas and Propagation, IEEE Transactions on, 1967, 15(5): 668-671.

[7] Blinn J F. Models of light reflection for computer synthesized pictures[C]//ACM SIGGRAPH Computer Graphics. ACM, 1977, 11(2): 192-198.

[8] Schlick C. An inexpensive BRDF model for physically-based rendering[C]//Computer graphics forum. 1994, 13(3): 233-246.

[9] Cook, Robert L., and Kenneth E. Torrance, “A Reflectance Model for Computer Graphics,” ACM Trans- actions on Graphics, vol. 1, no. 1, pp. 7–24, January 1982. http://graphics.pixar.com/library/ReflectanceModel/

[10] Kelemen, Csaba, and L´azl´o Szirmay-Kalos, “A Microfacet Based Coupled Specular-Matte BRDF Model with Importance Sampling,” Eurographics 2001, short presentation, pp. 25–34, September 2001. http: //www.fsz.bme.hu/~szirmay/scook_link.htm

[11] Brian Karis. Real Shading in Unreal Engine 4[C]//ACM SIGGRAPH. 2013.

[12] Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs