上一章中介绍的BRDF和BTDF仅解决了描述表面如何散射光的部分问题。尽管它们描述了光如何在表面上的特定点处散射，但是渲染器需要确定在表面上的点处存在哪些BRDF和BTDF，以及它们的参数是什么。在本章中，我们描述了解决此问题的程序着色机制。

pbrt方法背后的基本思想是将表面着色器绑定到场景中的每个图元。表面着色器由Material接口类的实例表示，该实例具有一种方法，该方法获取表面上的一个点并创建一个BSDF对象（可能还包括BSSRDF），该对象表示该点处的散射。BSDF类包含一组BxDF，将其贡献相加得出完整的散射函数。反过来，材质使用Texture类的实例（将在下一章中定义）来确定曲面上特定点处的材质属性.例如，可以使用ImageTexture来调制整个表面上的漫反射颜色。这与许多渲染系统使用的着色范例有所不同。通常的做法是将表面着色器和照明积分器（请参见第14章）的功能组合到一个模块中，并让着色器返回该点反射光的颜色。但是，通过分离这两个组件并使Material返回BSDF，pbrt能够更好地处理各种光传输算法。

9.1 BSDFs

BSDF类代表BRDF和BTDF的集合.以这种方式对它们进行分组可以使系统的其余部分直接与复合BSDF一起使用,而不必考虑它们可能是由其构建的所有组件.同样重要的是,BSDF类对系统其余部分隐藏了一些法线着色细节.从三角形网格中的每个顶点法线或凹凸贴图生成的阴影法线可以显着改善渲染场景的视觉丰富度，但是由于它们是临时结构，因此很难将其合并到基于物理的渲染器中.他们引入的问题在BSDF实现中处理.

BSDF构造函数采用一个SurfaceInteraction对象，该对象包含有关表面上该点的微分几何的信息以及一个参数eta,该参数给出边界上的相对折射率.对于不透明的表面,不使用eta,调用者应提供值1.(针对eta的默认值为1就是这种情况.)构造函数以着色法线为轴之一来计算正交法线坐标系.此坐标系对于在图8.2中描述的BxDF坐标系和从BxDF坐标系转换方向将很有用.在本节中,我们将使用以下约定:表示着色法线,而表示几何法线(图9.1).

BSDF实现仅存储有限数量的单个BxDF组件.如果给它更多的组件,它可以很容易地扩展以分配更多的空间,尽管到目前为止pbrt中的任何Material实现都不是必需的,并且当前限制为8,几乎可以满足所有实际应用.

对于需要有关存在的特定BRDF和BTDF的附加信息的系统其他部分,一种方法返回由BSDF存储的与特定BxDFType标志集匹配的BxDF数量.

在实践中,**着色法线会导致各种不良现象**(图9.2).图9.2(a)显示了光泄漏:几何法线表示和位于表面的相对两侧.因此,如果表面不透射光,则光应该没有贡献.但是,如果我们直接评估围绕以着色法线为中心的半球的散射方程(5.9),则会错误地合并来自的光.这种情况表明不能仅用作渲染计算中的直接替代.

图9.2(b)显示了类似的棘手情况:着色法线指示不应将任何光反射给观看者,因为它与照明不在同一个半球中,而几何法线指示它们在同一半球中.直接使用会在这种情况下在表面上造成难看的黑点.

幸运的是,对于这些问题有一个很好的解决方案.在评估BSDF时,我们可以使用几何法线来决定是否应该评估反射或透射:如果和相对于位于同一个半球,则评估BRDF,否则评估BTDF.但是，在评估散射方程时,法线和入射方向的点积仍采用着色法线而不是几何法线.

现在应该清楚为什么pbrt要求BxDF评估它们的值而不考虑和是在相同还是不同的半球中.这种约定意味着可以避免漏光,因为我们将仅针对图9.2(a)中的情况评估BTDF,而对于纯反射表面则不给出反射.同样,避免了黑点,因为我们将针对图9.2(b)中的情况评估BRDF,即使着色法线表明方向在不同的半球中也是如此.

在给出这些约定的情况下,直接遵循针对给定方向对评估BSDF的方法.首先将世界空间方向向量转换为局部BSDF空间,然后确定是否应使用BRDF或BTDF.然后,它遍历适当的集合并评估其贡献的总和.

pbrt还提供了BSDF方法,这些方法可以返回BSDF的反射率.(回想一下8.1.1节中反射率的定义.)两种相应的方法只是在BxDF上循环,并对它们的BxDF::rho()方法返回的值求和.它们的直接实现不在此处.这些方法获取BxDF的样本数组,以在需要时用于Monte Carlo采样算法中(请参阅第8.1.1节中定义的BxDF::rho()接口,该接口也将此类样本也包含在内).

9.2 材质接口和实现

抽象的Material类定义了材料实现必须提供的接口.Material类在文件core/material.h和core/material.cpp中定义.

9.2.1 哑光材料

MatteMaterial材料在materials / matte.h和materials / matte.cpp中定义.它是pbrt中最简单的材料,并描述了纯漫反射表面.

9.2.2 塑料材质

可以将塑料建模为散射和光泽散射函数的混合,并使用参数控制特定的颜色和镜面高光大小.PlasticMaterial的参数是两个反射率Kd和Ks,它们分别控制漫反射和光泽镜面反射的量.接下来是一个粗糙度参数,用于确定镜面高光的大小.可以通过两种方式指定.首先,如果remapRoughness参数为true,则给定的粗糙度应在0到1之间变化,其中粗糙度值越高，高光越大.(此变体的目的是相当方便用户使用.)或者,如果参数为false,则可以使用粗糙度直接初始化微面分布的α参数（请参见8.4.2节）.

9.2.3 混合材质

能够将两种不同权重的材料结合在一起很有用.MixMaterial使用其他两种材质和一个Spectrum值的纹理,并使用该纹理返回的Spectrum在着色点处在两种材质之间进行混合.它在文件materials / mixmat.h和materials / mixmat.cpp中定义.

9.2.4 Fourier Material

FourierMaterial类支持已测量的数据或合成的BSDF数据,这些数据已列表到8.6节中引入的方向基础中.它在文件materials / fourier.h和materials / fourier.cpp中定义.

9.2.5 额外材质

除了这些材料之外，pbrt中还有八个可用的Material实现，所有实现都在material /目录中。我们不会在此处显示其所有实现，因为它们只是上述材料实现中引入的基本主题的变体.全部采用定义散射参数的纹理,这些纹理在材料各自的ComputeScatteringFunctions（）方法中进行评估,并创建适当的BxDF并在BSDF中返回.有关这些材料采用的参数的摘要,请参见pbrt文件格式的文档.

这些材质包括:

GlassMaterial:完美或光滑的镜面反射和透射,通过菲涅耳项加权,以实现精确的角度相关变化.

MetalMaterial:金属,基于导体的菲涅耳方程和Torrance-Sparrow模型.与塑料不同,金属不包含漫反射成分.有关各种金属的折射率η和吸收系数k的测量光谱数据,请参见scenery / spds / metals /目录中的文件.

MirrorMaterial:简单的镜子,以完美镜面反射为模型.

SubstrateMaterial:一种分层模型,根据视角在光泽镜面反射和漫反射之间变化(基于FresnelBlend BRDF).

SubsurfaceMaterial和KdSubsurfaceMaterial:返回BSSRDF的材质.这种材质描述了次表面闪射效应.

TranslucentMaterial:描述漫反射和光泽镜面反射和通过表面透射的材料.

UberMaterial:一种“厨房水槽”材料,代表许多前述材料的结合.这是一种高度参数化的材料,在将场景从其他文件格式转换为pbrt格式时特别有用.

上一章中的图8.10显示了使用GlassMaterial渲染的龙模型,图9.4使用了MetalMaterial渲染了它.图9.5展示了KdSubsurfaceMaterial.

9.3 Bump映射

上一节中定义的所有材料都采用可选的浮点纹理，该纹理定义了表面上每个点的位移:每个点都有一个与之关联的位移点,由,其中是处的位移纹理返回的偏移量,是处的表面法线(图9.6),我们希望使用该纹理来计算着色法线,以便曲面看起来好像实际上已由位移函数偏移了一样,而没有修改其几何形状.此过程称为凹凸贴图.对于相对较小的位移函数,凹凸贴图的视觉效果可能非常令人信服.Blinn(1978)提出了这种想法和一种特殊的技术,以一种看起来真实的置换表面外观的方式来计算这些着色法线.

图9.7显示了应用由网格的图像映射定义的凹凸映射的效果.线到一个球体.图9.8显示了一个更复杂的示例,该图显示了带有和不带有凹凸贴图的场景.在那里,凹凸贴图使墙壁和地板上的大量细节看上去没有几何模型中实际存在的.图9.9显示了用于定义图9.8中的凹凸函数的图像映射之一。Material::Bump()方法是供Material实现使用的实用程序例程.它负责计算在给定特定位移Texture的情况下被着色点的凹凸贴图的效果.为了使将来的Material实现不需要使用此特定机制(或根本不支持)凹凸映射,我们已将此方法放置在硬编码的Material Evaluation管道之外,并将其保留为特定Material实现可以调用的函数他们自己.

Material::Bump()的实现基于找到置换曲面的偏导数和的近似值,并使用它们代替曲面的实际偏导数来计算着色法线.(回想一下,表面法线是由这些向量的叉积给出的,.)假定原始表面由参数函数定义,并且凹凸偏移函数是标量函数.然后通过

其中是曲面在处的法线.

可以使用链规则找到该函数的偏导数.例如,中的偏导数为

我们已经计算出的值;它在SurfaceInteraction结构中使用,该结构还存储表面法线和偏导数.位移函数可以根据需要进行评估,只剩下.

有两种可能的方法来找到和的值.一种选择是使用一种计算基础纹理函数的偏导数的方法来扩展Texture接口.例如,对于使用其参数化直接映射到表面的图像地图纹理,可以通过在和方向上减去相邻纹理像素来计算这些偏导数.但是,这种方法很难扩展到像第10章中定义的那样的复杂过程纹理.因此,pbrt可以直接在Material::Bump()方法中使用前向差异来计算这些值,而无需修改Texture接口.偏导数的定义为:

前向微分使用有限值并在两个位置评估来近似值.因此,的最终表达式如下(为简单起见,对于某些项,我们放弃了对的显式依赖):

有趣的是,在预期相对较小的假设下,大多数凹凸映射实现都忽略了最后一项.(由于凹凸贴图对于近似较小的扰动最有用,所以这是一个合理的假设.)许多渲染器不计算值和的事实也可能与这种简化有关.忽略最后一项的含义是,位移函数的大小不影响凹凸映射的偏导数.在全局范围内为其添加一个常数值不会影响最终结果,因为只有凹凸函数的差异会影响最终结果.pbrt计算所有三个项,因为它容易获得和,尽管在实践中该最后一项很少在视觉上引起明显的区别.

Bump（）定义中的一个重要细节是d参数声明为const shared\_ptr <Texture <Float >>＆类型，而不是例如shared\_ ptr <Texture <Float >>.这种差异对于性能非常重要，但是原因很微妙。 如果此处未使用C ++引用，则shared\_ptr实现将需要为传递给该方法的临时值增加引用计数，并且在方法返回时需要减少引用计数。 这是使用串行代码进行的高效操作，但是具有多个执行线程，因此会导致以下情况：每当不同的渲染任务使用相同的置换纹理运行此方法时，多个处理核心最终将修改同一内存位置。 这种状况反过来导致了第A.6.1节中描述的昂贵的“读取所有权”操作.

剩下的一个问题是如何为有限差分计算选择偏移量和.它们应足够小以捕获的细微变化,但又应足够大以使可用的浮点精度足以给出良好的结果.在这里,我们将选择导致偏移量约为图像空间像素采样间隔一半的和值,并使用它们来更新SurfaceInteraction中的相应成员变量以反映向偏移位置的偏移.(有关如何计算图像空间距离的说明,请参见第10.1.1节.)以下代码中要注意的另一细节:我们将表面法线重新计算为和的叉积.而不是直接使用si-> shading.n.原因是的方向可能已经翻转(请参见第2.10.1节中的片段,根据方向和惯用性来调整法线).但是,我们需要这里的原始法线.稍后,当计算结果传递给SurfaceInteraction::SetShadingGeometry()时,如果需要,我们计算的法线本身将被翻转.