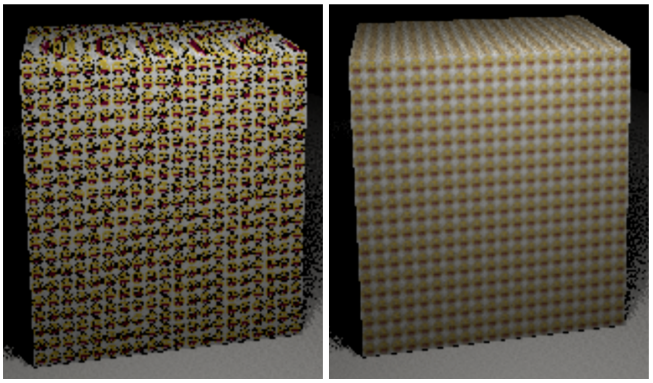
PBRT 系列 14-代码实战-光线微分与纹理

Dezeming Family

2021 年 3 月 30 日

因为本书是电子书，所以会不断进行更新和再版（更新频率会很高）。如果您从其他地方得到了这本书，可以从官方网站：https://dezeming.top/ 下载新的版本（免费下载）。

本书目标：学习 PBRT 中的光线微分技术，实现光线微分并对比效果。实现 PBRT 的图像纹理，渲染得到如下效果：



本文于 2022 年 7 月 16 日进行再版，提供了源码。注意图形 GUI 界面和本文中展示的有点区别，但并不影响学习。源码见网址 [https://github.com/feimos32/PBRT3-DezemingFamily]。再版中增加了一些章节，比如详细地列出了所有需要修改和移植的文件和类。

前言

本书开始实现光线微分与纹理。

虽然简单地理解光线微分并不需要什么微分几何方面的知识，但是缺少这些知识对自己实际推导还是有一定影响。所幸论文和书中有比较详细的推导过程，因此可以参考学习。哪怕您有些步骤看不懂，最起码也应该知道光线微分的过程和意义。简单来说，光线微分就是在跟踪 Ray 时，判断一个像素格内的 Ray 可以采样到哪些范围，主要用于纹理滤波。光线在被有 Specular 类型的物体反射时，需要根据表面的微分参数来决定反射的范围，也就是说，不同的 shape 类型有着不同的光线微分的计算方案。

本书的售价是 4 元（电子版），但是并不直接收取费用。如果您免费得到了这本书的电子版，在学习和实现时觉得有用，可以往我们的支付宝账户（17853140351，可备注：PBRT）进行支持，您的赞助将是我们 Dezeming Family 继续创作各种图形学、机器学习、以及数学原理小册子的动力！

目录

一　光线微分的原理

本章内容总结自 [5]，为了语言准确性保留英文描述（除此之外，光线跟踪用中文描述，发出的光线用 Ray 描述。翻译整理的比较潦草，有兴趣还是去看论文比较合适，因为确实不难）。

对于追踪 Ray 的 footprint（足印，也就是一个像素格中 Ray 可以覆盖的面积），有多种方法：

LOD(Geometric level-of-detail) 可以反混淆几何细节，将物体按照不同层次进行细分，然后根据 Ray 密度（视点离物体的远近）来判断选择的 LOD 层次；光线跟踪焦散时，采样光线的强度取决于波前的会聚或发散；类似地，在照明贴图中，来自光源的光线所携带的 flux 必须根据光线的密度沉积在曲面照明贴图的某个区域上；Dull reflections 可以通过过滤超出光线实际 footprint 的内核上的纹理来建模。

使用近似的 Ray 的 footprint 方法为 Ray differentials，采用导数的一阶泰勒近似可以得到 Ray 的估计。已经存在了几种其他的估计 Ray 密度的技术，但是 Ray differentials 比这些算法更快、更简单且更鲁棒。特别是由于光线微分是基于初等微分，而不是微分几何或理解 3D 世界的另一个数学基础，所以该技术可以很容易地应用于非物理现象，例如凹凸映射和正常插值三角形。该论文导出了传输、反射和折射光线跟踪微分的一般公式，以及处理法向插值三角形的具体公式，最后还演示了光线微分在纹理过滤中的应用。

1 1　相关工作

在多边形绘制管线中，研究者们已经开发了几种估计纹理过滤核的算法，在光线跟踪器中也使用了类似的算法来计算纹理坐标在图像平面上的投影，但是这样的投影只对 eye Ray 有效。这项技术可以通过计算总行进距离来扩展到反射和折射光线，但这种近似是无效的，因为曲面可以极大地修改光线的会聚或发散。因此研究者开发了一些考虑到这一点的算法，我们将简要地回顾它们：

在光线跟踪器的环境中，通过检查相邻 Ray 的照明图坐标，有限差分已被用于计算光线的照度沉积在 illumination map 上的程度。有限差分法的主要优点是它可以很容易地处理任何类型的曲面，然而主要的缺点是与处理光线树不同的相 eye Ray 相关的困难问题：该算法必须检测相邻 Ray 何时不击中相同的基元，并对这种情况进行特殊处理。处理这种情况的一种可行方法是发出一条特殊的相邻 Ray，该光线遵循同一光线树，并在三角形边之外与同一三角形的平面相交，但这对球体和其他高阶基本体不起作用。此外，不击中相同的基元这种特殊情况可能是很常见情况，因为相邻 Ray 可能与实际上是同一对象的不同基元相交（例如三角形网格）。光线微分算法通过利用独立于相邻 Ray 的微分量来规避这些困难。

圆锥体跟踪是一种方法，在这种方法中，对每条光线跟踪光线足迹的圆锥体近似值。除了纹理过滤外，此圆锥体还用于边缘抗锯齿。圆锥近似的主要问题是圆锥是各向同性的。这不仅意味着必须在图像平面上对像素进行各向同性采样，而且当光线迹线在反射或折射后变得各向异性时，必须使用各向同性迹线对其进行重新近似。因此，该方法不能用于各向异性纹理滤波等算法。此外，将该技术扩展到支持平面多边形和球体以外的曲面并非易事。

波前跟踪法 (Wavefront tracing) 是一种跟踪光线波前表面微小区域特性的方法，它被用于计算曲面照明产生的焦散强度。波前也被用来计算人眼的聚焦特性，虽然与我们的算法很相关，但波前跟踪和我们的光线微分方法之间的主要区别在于，我们的方法跟踪一个微分小距离的方向特性，而波前跟踪跟踪一个微分小区域的表面特性。因此，波前不能处理像素样本之间的各向异性间距。此外，由于微分面积本身是一个更复杂的量，因此与波前跟踪相关的计算步骤更为复杂。波前跟踪是基于微分几何的，而我们的算法是基于初等微分学的，这样做的一个技术结果是，我们可以很容易地处理非物理现象，如法线插值三角形，凹凸映射曲面，以及其他在微分几何框架下自相矛盾的算法。基于简单的初等微分学领域的一个实际结果是，我们的公式更容易理解，并且扩展技术来处理不同的现象是简单的。

傍轴光线理论 (Paraxial ray theory) 是最初为透镜设计开发的一种近似技术，它在光线跟踪中的应用被称为 Pencil Tracing。在 Pencil Tracing 中，近轴光线到轴光线是由垂直于轴光线的平面上的点矢量对参数化的。这些近轴光线的传播用系统矩阵线性近似。与波前跟踪一样，Pencil Tracing 的计算复杂度明显高于我们的方法。此外，铅笔追踪还提出了一组独特的简化假设，使每个现象相对于系统矩阵呈线性。即使在传输时也会进行近似，这是一种相对于光线呈线性的现象。此外，对非物质现象的处理还不清楚。通过比较，我们所做的单一统一近似只是 Ray 函数的一阶泰勒近似。

1 2 跟踪 Ray Differentials

本节内容并不难，而且即使不掌握也不耽误使用光线微分技术。为了完整我们还是进行一下叙述，更详细的内容请参考论文。

我们可以把从像素上的 (x,y) 点进行光线跟踪想象为一个函数，并可以求其导数：

$$v = f_n(f_{n-1}(\cdots f_2(f_1(x,y)))) \tag{一.1}$$

$$\frac{\partial f_n}{\partial x} = \frac{\partial f_n}{\partial f_{n-1}} \cdots \frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial x} \tag{一.2}$$

当变换应用于光线以模拟通过场景的传播时，我们只是应用一组函数 f_i 来跟踪光线。我们还可以通过应用函数的导数来跟踪光线微分，即光线相对于图像空间坐标的导数。然后可以使用这些光线微分给出光线作为图像空间坐标函数的一阶泰勒近似。在接下来的推导中，我们用斜体表示标量，用粗体表示点、方向和具有齐次坐标的平面。

可以通过表示光线上位置的点和表示其方向的单位向量来参数化光线：

$$\vec{\mathbf{R}} = \langle \mathbf{P}\ \mathbf{D} \rangle \tag{一.3}$$

光线的初始值取决于图像平面的参数：针孔相机由视点、观察方向、右方向和向上方向来描述。光线穿过图像平面上像素的方向可以表示为这些方向的线性组合：

$$\mathbf{d}(x,y) = \mathbf{View} + x\mathbf{Right} + y\mathbf{Up} \tag{一.4}$$

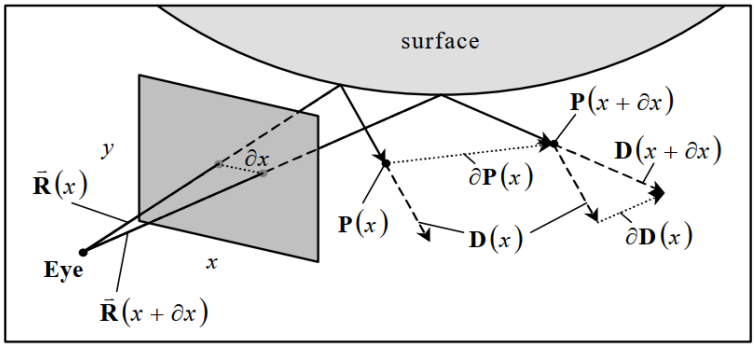
$$\mathbf{P}(x,y) = \mathbf{Eye} \tag{一.5}$$

$$\mathbf{D}(x,y) = \frac{\mathbf{d}}{(\mathbf{d}\cdot\mathbf{d})^{1/2}} \tag{一.6}$$

每一条微分偏移光线都可以用一对方向来表示，我们称之为光线微分：

$$\frac{\partial \vec{\mathbf{R}}}{\partial x} = \langle \frac{\partial \mathbf{P}}{\partial x}\ \frac{\partial \mathbf{D}}{\partial x} \rangle \tag{一.7}$$

$$\frac{\partial \vec{\mathbf{R}}}{\partial y} = \langle \frac{\partial \mathbf{P}}{\partial y}\ \frac{\partial \mathbf{D}}{\partial y} \rangle \tag{一.8}$$



如果只考虑 x 方向，临近光线的距离可以表示为一阶微分近似：

$$\left[\vec{\mathbf{R}}(x+\Delta x, y) - \vec{\mathbf{R}}(x,y)\right] \approx \Delta x \frac{\partial \vec{\mathbf{R}}(x,y)}{\partial x} \tag{一.9}$$

$$\left[\vec{\mathbf{R}}(x, y+\Delta y) - \vec{\mathbf{R}}(x,y)\right] \approx \Delta y \frac{\partial \vec{\mathbf{R}}(x,y)}{\partial y} \tag{一.10}$$

我们可以计算一下光线微分的初始值：

$$\frac{\partial \mathbf{P}}{\partial x} = \mathbf{0} \tag{一.11}$$

$$\frac{\partial \mathbf{D}}{\partial x} = \frac{(\mathbf{d}\cdot\mathbf{d})\mathbf{Right} - (\mathbf{d}\cdot\mathbf{Right})\mathbf{d}}{(\mathbf{d}\cdot\mathbf{d})^{3/2}} \tag{一.12}$$

注意上面公式的推导其实很简单，把 D 和 d 代进去就好了。

同样可以为 y 方向导出类似的表达式。虽然我们只跟踪一阶导数，但为了得到更好的近似值或误差边界，也可以计算高阶导数。然而，我们发现不连续性限制了高阶近似的有效性，并且一阶近似在实践中是足够的。

我们将导出三种常见光线跟踪操作的公式：传输、反射和折射。

1 2.1 传输

由下面的公式可以进行推导：

$$\mathbf{P'} = \mathbf{P} + t\mathbf{D} \tag{一.13}$$

$$\mathbf{D'} = \mathbf{D} \tag{一.14}$$

论文中，我们定义了一个表面 N（**P'** 在表面 **N** 上），这里的 N 也可以表示为表面法向量。其实不光平面，对于任意曲面，N 是曲面在交点处的法线。这背后的直觉是，曲面在交点处具有固定的形状和曲率。当我们将偏移光线与该曲面相交的偏移量越来越小时，曲面将越来越像交点处的切面。

1 2.2 反射

反射求光线微分也非常容易：

$$\mathbf{P'} = \mathbf{P} \tag{一.15}$$

$$\mathbf{D'} = \mathbf{D} - 2(\mathbf{D} \cdot \mathbf{N})\mathbf{N} \tag{一.16}$$

1 2.3 折射

我们只考虑折射发生时的情况（忽略其他效果，比如全反射可以使用上面的反射计算）：

$$\mathbf{P'} = \mathbf{P} \tag{一.17}$$

$$\mathbf{D'} = \eta\mathbf{D} - \mu\mathbf{N} \tag{一.18}$$

1 2.4 表面法向量

在微分几何中，shape operator(S) 表示为单位法向量相对于曲面切线方向的负导数。在我们的计算中，目标法向量方向由光线交点的导数给出，因此：

$$\frac{\partial \mathbf{N}}{\partial x} = -\mathbf{S}\big(\frac{\partial \mathbf{P}}{\partial x}\big) \tag{一.19}$$

1 3 纹理滤波

光线微分的一个重要作用就是纹理滤波。

如果我们能近似一条光线与其相邻光线对应的纹理坐标之间的差值，那么我们就可以在纹理空间中找到滤波核的大小和形状。曲面的纹理坐标取决于曲面的纹理参数化。这样的参数化对于参数化曲面来说很简单，并且存在参数化隐式曲面的算法。

纹理坐标可以表示为：

$$\mathbf{T} = f(\mathbf{P}) \tag{一.20}$$

我们可以对 x 进行微分，得到一个依赖于交点及其导数的函数：

$$\frac{\partial \mathbf{T}}{\partial x} = \frac{\partial [f(\mathbf{P})]}{\partial x} = g\big(\mathbf{P}, \frac{\partial \mathbf{P}}{\partial x}\big) \tag{一.21}$$

应用一阶泰勒近似，我们得到一个基于像素间距的像素在纹理空间中的 footprint 范围表达式：

$$\Delta\mathbf{T}_x = [\mathbf{T}(x + \Delta x, y) - \mathbf{T}(x, y)] \approx \Delta x \frac{\partial \mathbf{T}}{\partial x} \tag{一.22}$$

$$\Delta\mathbf{T}_y = [\mathbf{T}(x, y + \Delta y) - \mathbf{T}(x, y)] \approx \Delta y \frac{\partial \mathbf{T}}{\partial y} \tag{一.23}$$

论文中还提供了三角形内纹理滤波的方法，很简单因此这里不再涉及。

二　PBRT 光线微分的描述

本章研究的内容是 PBRT 中的光线微分技术。

2 1　光线微分基类与光线微分的产生

光线微分基类定义在 Geometry.hpp 文件中，继承自 Ray 类。里面各个变量的定义都很清晰，只需要注意 ScaleDifferentials() 方法缩放差分光线，以考虑每个像素采集多个样本的情况下，Film 平面上样本之间的实际间距。

生成光线微分的过程见 SamplerIntegrator 的 Render 函数：

```
// Generate camera ray for current sample
RayDifferential ray;
Float rayWeight =
    camera->GenerateRayDifferential(cameraSample, &ray);
ray.ScaleDifferentials(
    1 / std::sqrt((Float)tileSampler->samplesPerPixel));
```

在 Camera 基类中就定义了产生光线微分的程序。

```
    //Camera::GenerateRayDifferential函数：
    //C++11新特性，for循环里要么是0.05，要么是-0.05
    for (float eps : { .05, -.05 })
```

为了防止离焦模糊等效果导致偏差太大，这里首先只对 Film 上的位置偏移了 eps，然后再在得到的微分 Ray 上除以 eps。

在相机类的派生类中，例如透视投影相机类也定义了产生光线微分的程序，首先先产生 Ray，然后再根据相机偏移来产生光线微分。

我们先把上面叙述过的类和方法都移植到自己的代码里（本章末尾会说全部需要移植的内容清单）。

2 2　光线微分的传播

我们跟随路径追踪的 Li 程序，来定位光线微分的传播和计算。本节我们先把以前没有补充过的变量都补充到里面，然后下一节我们讲解这些变量（SurfaceInteraction 类内的）究竟代表什么含义。

SamplerIntegrator 类内部的函数都是使用的光线微分参数，我们需要对此修改。

Light 基类的 Le 函数也需要光线微分作为参数，该函数主要是为了全景图照明取样，其他光源派生类的返回值都是 0。

表面交点类 SurfaceInteraction 的 ComputeScatteringFunctions 函数也都是输入光线微分作为参数，其它的 ComputeScatteringFunctions 函数并没有输入 Ray，而是以表面交点对象作为参数。该类通过 ComputeDifferentials 来计算光线微分，我们还需要补充一些表面交点类的成员变量。

对于图形相交的算法，我们需要移植完整的 shape 里的 Intersect 和 IntersectP 函数，这里面有我们想要的信息。

2 3　法向量、切线与微分

SurfaceInteraction 类内有这些描述表面几何特性的变量：

```
    // Interaction Public Data
    Point3f p; //Ray与物体的交点
    Vector3f wo; //Ray射入到交点的反方向
    Normal3f n; //物体法向量
    // SurfaceInteraction Public Data
```

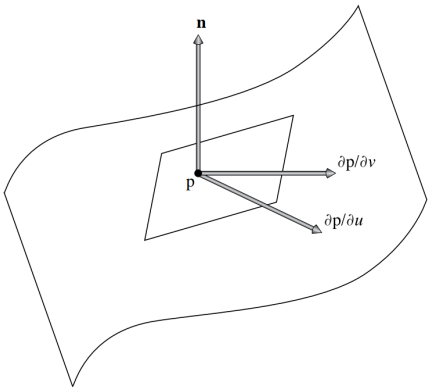
```
    Point2f uv; //纹理坐标
    Vector3f dpdu, dpdv;
    Normal3f dndu, dndv;
    struct {
        Normal3f n;
        Vector3f dpdu, dpdv;
        Normal3f dndu, dndv;
    } shading;
    mutable Vector3f dpdx, dpdy;
    mutable Float dudx = 0, dvdx = 0, dudy = 0, dvdy = 0;
```

我们以前分别在《形状和加速器》以及《反射与材质初步了解》中讲过一部分，但并不全面。现在我们重新完整的描述一下出了上面代码中有注释的，其他所有变量的来源及作用。

首先我先介绍两个函数：CoordinateSystem 函数通过第一个向量来得到一个局部坐标系，该坐标系的第二个轴的向量是第一个轴产生的，然后叉积得到第三个轴。SolveLinearSystem2x2 函数是根据克莱姆法则求解 2*2 线性方程组的，后面用到再来细说。

2 3.1 dpdu,dpdv,dndu,dndv 的计算

如下图所示，曲面的参数偏导数 $\partial p/\partial u$ 和 $\partial p/\partial v$ 表示为 dpdu 和 dpdv，位于切平面上，但不一定正交。曲面法线 n 由 $\partial p/\partial u$ 和 $\partial p/\partial v$ 的叉积给出。向量 $\partial n/\partial u$ 和 $\partial n/\partial v$（此处未显示）记录了沿曲面移动 u 和 v 时曲面法线的差异变化。



在 triangle.cpp 文件中的下面代码段来计算的切线方向：

```
    // Compute deltas for triangle partial derivatives
```

三角形可以表示为下式，虽然 uv 是二维坐标，但因为三角形就是一个面，所以下面的式子其实是成立的：

$$p_0 + u\frac{\partial p}{\partial u} + v\frac{\partial p}{\partial v} \tag{二.1}$$

$$\begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix} \begin{pmatrix} \frac{\partial p}{\partial u} \\ \frac{\partial p}{\partial v} \end{pmatrix} = \begin{pmatrix} p_0 - p_2 \\ p_1 - p_2 \end{pmatrix} \tag{二.2}$$

有必要处理奇异矩阵情况（不能求逆）。这仅在用户提供的逐顶点参数化值退化时发生（我认为就是三个顶点共边时）。在这种情况下，三角形只是选择一个关于三角形曲面法线的任意坐标系，确保它是正交的即可。

2 3.2 shading 结构中的变量

n 表示着色法向量：

```
//SetShadingGeometry函数:
shading.n = Normalize((Normal3f)Cross(dpdus, dpdvs));
```

是根据传入参数 dpdu 和 dpdv 计算出来的。

dpdu 和 dpdv 来自 triangle.cpp 的 Intersect 函数中计算得到的 ss 和 ts 值。一般我们的模型里没有切线，所以 ss 就是上一节算出的几何切线 dpdu，ns 是模型的法向量（如果模型不提供法向量就是上一节计算出的 n），之后根据 ss 和 ns 求出 ts。

如果模型提供了法向量，就据此计算出 shading 的 dndu 和 dndv（方法与计算 dpdu 和 dpdv 相似，如下），否则它们都是 0 向量。

$$n + u\frac{\partial n}{\partial u} + v\frac{\partial n}{\partial v} \tag{二.3}$$

$$\begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix} \begin{pmatrix} \frac{\partial n}{\partial u} \\ \frac{\partial n}{\partial v} \end{pmatrix} = \begin{pmatrix} n_0 - n_2 \\ n_1 - n_2 \end{pmatrix} \tag{二.4}$$

2 3.3 dpdx 和 dpdy

SurfaceInteraction 类剩下的几个变量都是与光线微分有关的。

注意计算光线微分是在调用材质类计算散射 BSDF 函数之前进行的。

ComputeDifferentials 首先计算出辅助交叉点 px 和 py，然后计算出 dpdx 和 dpdy。然后计算出二维平面上的偏移：dudx,dudy,dvdx,dvdy。这些值会用于纹理滤波。

2 4 反射与折射

SpecularReflect 和 SpecularTransmit 函数需要计算镜面反射和折射后的光线微分，计算方法可见 10.1.3 节，这里不再讲解。

至此，光线微分的基本内容我们就简单介绍完了，需要注意的是，路径追踪的再生光线中，没有根据镜面反射或者折射来实现光线微分（比如我们相机产生的 Ray 第一次跟镜面物体相交，则反射的时候应该需要计算光线微分才对，PBRT 路径追踪的实现中并没有跟 SpecularReflect 和 SpecularTransmit 一样的可以追踪镜面反射/穿透的光线微分，所以说 PBRT 中的路径追踪器对光线微分的支持并不是很好，但其实很多时候问题也不是很大）。

2 5 全部需要移植的内容清单

本节给出全部需要移植的内容清单，通过清单我们可以看出，光线微分是一个非常容易实现、不需要改变原来系统结构的非常好的工具。

内容一： Geometry.h 中的 RayDifferential 类。

内容二： Transform 对 RayDifferential 的变换函数：

```
inline RayDifferential operator()(const RayDifferential &r) const;
```

内容三： Camera 类的相关变量和函数。OrthographicCamera 类的相关变量和函数。PerspectiveCamera 类的相关变量和函数。

内容四： SurfaceInteraction 的如下内容：

```
void ComputeDifferentials(const RayDifferential &ray) const;
void SetShadingGeometry(const Vector3f &dpdu, const Vector3f &dpdv, const
    Normal3f &dndu, const Normal3f &dndv, bool orientationIsAuthoritative);
// 下面这个函数需要加入调用ComputeDifferentials()的代码:
SurfaceInteraction::ComputeScatteringFunctions
// 一些需要加入的变量
```

```
mutable Vector3f dpdx, dpdy; //光线微分方向
mutable float dudx = 0, dvdx = 0, dudy = 0, dvdy = 0;
```

内容五：Triangle 的 Intersect() 函数下面部分内容：

```
// Compute deltas for triangle partial derivatives
......
if (mesh->n || mesh->s){
    ......
}
```

内容六：SamplerIntegrator 的 SpecularReflect 函数和 SpecularTransmit 函数。Li() 函数的参数列表。

内容七：全部渲染积分器的 Li() 函数的第一个参数，都改为 RayDifferential。

内容八：在光源的 Le 函数的参数都要改为 RayDifferential，注意 Le 只在无限面光源中有效，会得到光源值；在其他光源都是返回 Spectrum(0.f)。

内容九：SamplerIntegrator 的 Render() 函数产生光线微分：

```
// 把下面两行代码：
Feimos::Ray ray;
camera->GenerateRay(cs, &ray);
// 改为：
Feimos::RayDifferential ray;
float rayWeight = //对于投射相机来说权重都是1
    camera->GenerateRayDifferential(cs, &ray);
ray.ScaleDifferentials(
    1 / std::sqrt((float)sampler_c->samplesPerPixel));
```

我们可以看出来，光线微分其实主要就是与求交有关。注意我们虽然已经阐述了需要移植的内容，但此时仅仅只能通过编译，我们的 Li() 函数内容还没有进行任何修改，但如果稍加思考，就会明白，其实 Li() 函数是不需要改的，。

三　PBRT 纹理

纹理的接口我们已经讲解过了，现在我们再回顾一下，然后实现图像纹理（我们目前只有最简单的常量纹理）。

3 1　2D 纹理映射

纹理有多种映射方式，我们这里只了解 2D(u,v) 映射。即提供纹理坐标，我们搜寻纹理值。该类的构造函数包含了四个参数，su,sv 表示 scale，du,dv 表示偏移（shift）。

Map 函数需要计算出纹理微分，用于滤波：

$$\frac{\partial s}{\partial x} = \frac{\partial u}{\partial x}\frac{\partial s}{\partial u} + \frac{\partial v}{\partial x}\frac{\partial s}{\partial v} \tag{三.1}$$

$$s = s_u u + d_u \tag{三.2}$$

$$\frac{\partial s}{\partial x} = s_u \frac{\partial u}{\partial x} \tag{三.3}$$

3 2　ImageTexture 类

该类的构造很简单，注意读入的时候需要使用 Gamma 矫正的逆变换（因为我们认为图像是经过 gamma 矫正后得到的，因此读入的时候需要进行逆变换）。maxAniso 表示最大各向异性，在 MIPMap 中有用到。

我们这样读入图像时，图像已经被上下翻转了（图像的 (0,0) 在左上角，但是我们显示物体的纹理坐标最小值表示为左下角，因此需要翻转一下）：

```
1    stbi_set_flip_vertically_on_load(true); //图像上下翻转
2    data = stbi_loadf(filename.c_str(), &imageWidth, &imageHeight, &
        nrComponents, 0);
```

因此构造函数里不用再进行反转了：

```
1    //可以去掉下面这些内容：
2        // Flip image in y; texture coordinate space has (0,0) at the lower
3        // left corner.
```

Evaluate 就是根据光线微分值来查找纹理的范围的，会调用 MIPMap 的 Lookup 函数。

移植的过程比较简单，基本上复制粘贴就可以，不再赘述。

3 3　纹理测试

本节我们简单测试一下纹理功能是否齐全。读者可以在阅读本节之前自己试试能否测试一下纹理，这对能力的提升有很大的好处。

我是 learnOpenGL[6] 的粉丝，所以我打算将 [6] 中的笑脸箱子的纹理进行实现，大家也可以随便找一些图片来显示。

我们加载图像的函数改为下面代码，用来替代 ReadImage 函数：

```
1  RGBSpectrum* loadImage(const std::string &filename, Point2i &resolution) {
2      float *data;
3      RGBSpectrum *data_s;
4      if (filename != "") {
5          //texels = ReadImage(texmap, &resolution);
6          int imageWidth, imageHeight, nrComponents;
```

```
        stbi_set_flip_vertically_on_load(true);
        data = stbi_loadf(filename.c_str(), &imageWidth, &imageHeight, &
            nrComponents, 0);
        if (!data)return nullptr;
        data_s = new RGBSpectrum[imageWidth * imageHeight];
        for (int j = 0; j < imageHeight; j++) {
            for (int i = 0; i < imageWidth; i++) {
                RGBSpectrum r;
                r[0] = data[(i + j * imageWidth) * nrComponents + 0];
                r[1] = data[(i + j * imageWidth) * nrComponents + 1];
                r[2] = data[(i + j * imageWidth) * nrComponents + 2];
                data_s[i + j * imageWidth] = r;
            }
        }
        resolution.x = imageWidth;
        resolution.y = imageHeight;
        free(data);
        return data_s;
    }
    return nullptr;
}
```

构造材质的程序如下：

```
inline std::shared_ptr<Material> getSmileFaceDiffuseMaterial() {
    std::unique_ptr<TextureMapping2D> map = std::make_unique<UVMapping2D>(1.
        f,1.f,0.f,0.f);
    std::string filename = "你的文件路径";
    ImageWrap wrapMode = ImageWrap::Repeat;
    bool trilerp = false;
    float maxAniso = 8.f;
    float scale = 1.f;
    bool gamma = false; //根据PBRT代码，如果是tga和png就是true;
    std::shared_ptr<Texture<Spectrum>> Kt = std::make_shared<ImageTexture<
        RGBSpectrum, Spectrum>>(std::move(map), filename, trilerp,
        maxAniso, wrapMode, scale, gamma);
    std::shared_ptr<Texture<float>> sigmaRed = std::make_shared<
        ConstantTexture<float>>(0.0f);
    std::shared_ptr<Texture<float>> bumpMap = std::make_shared<
        ConstantTexture<float>>(0.0f);
    return std::make_shared<MatteMaterial>(Kt, sigmaRed, bumpMap);
}
```

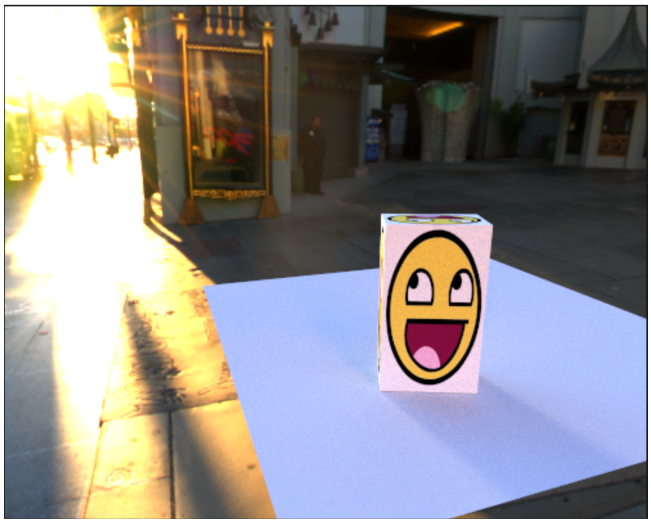
另外箱子的顶点局部坐标和纹理坐标表示如下：

```
Point3f P_wall[nVerticesWall] = {
    //底板
    Point3f(0.f,0.f,zlength),Point3f(0.f,0.f,0.f),Point3f(xlength,0.f,
        zlength),
```

```
    Point3f(xlength,0.f,zlength),Point3f(0.f,0.f,0.f),Point3f(xlength,0.f,0.f),
    //顶板
    Point3f(0.f,ylength,zlength),Point3f(xlength,ylength,zlength),Point3f(0.f,ylength,0.f),
    Point3f(xlength,ylength,zlength),Point3f(xlength,ylength,0.f),Point3f(0.f,ylength,0.f),
    //后板
    Point3f(0.f,0.f,0.f),Point3f(xlength,ylength,0.f),Point3f(xlength,0.f,0.f),
    Point3f(0.f,0.f,0.f),Point3f(0.f,ylength,0.f), Point3f(xlength,ylength,0.f),
    //前板
    Point3f(0.f,0.f,zlength),Point3f(xlength,0.f,zlength), Point3f(xlength,ylength,zlength),
    Point3f(0.f,0.f,zlength), Point3f(xlength,ylength,zlength),Point3f(0.f,ylength,zlength),
    //右板
    Point3f(0.f,0.f,0.f),Point3f(0.f,0.f,zlength),Point3f(0.f,ylength,zlength),
    Point3f(0.f,0.f,0.f),Point3f(0.f,ylength,zlength), Point3f(0.f,ylength,0.f),
    //左板
    Point3f(xlength,0.f,0.f),Point3f(xlength,0.f,zlength),Point3f(xlength,ylength,zlength),
    Point3f(xlength,0.f,0.f),Point3f(xlength,ylength,zlength), Point3f(xlength,ylength,0.f)
};
Point2f UV_wall[nVerticesWall] = {
    //注意坐标如果不在[0,1]之间就会根据选择的纹理warp方法进行采样
    //底板
    Point2f(0.f,1.f),Point2f(0.f,0.f),Point2f(1.f,1.f),
    Point2f(1.f,1.f),Point2f(0.f,0.f),Point2f(1.f,0.f),
    //顶板
    Point2f(0.f,1.f),Point2f(1.f,1.f),Point2f(0.f,0.f),
    Point2f(1.f,1.f),Point2f(1.f,0.f),Point2f(0.f,0.f),
    //后板
    Point2f(0.f,0.f),Point2f(1.f,1.f),Point2f(1.f,0.f),
    Point2f(0.f,0.f),Point2f(0.f,1.f),Point2f(1.f,1.f),
    //前板
    Point2f(0.f,0.f),Point2f(1.f,0.f),Point2f(1.f,1.f),
    Point2f(0.f,0.f),Point2f(1.f,1.f),Point2f(0.f,1.f),
    //右板
    Point2f(0.f,0.f),Point2f(0.f,1.f),Point2f(1.f,1.f),
    Point2f(0.f,0.f),Point2f(1.f,1.f),Point2f(1.f,0.f),
    //左板
    Point2f(0.f,0.f),Point2f(0.f,1.f),Point2f(1.f,1.f),
```

```
40      Point2f(0.f,0.f),Point2f(1.f,1.f),Point2f(1.f,0.f)
41  };
```

得到渲染结果如下：



3 4 光线微分和普通 Ray 渲染的区别

注意在 Transform 类里面实现下面的成员函数

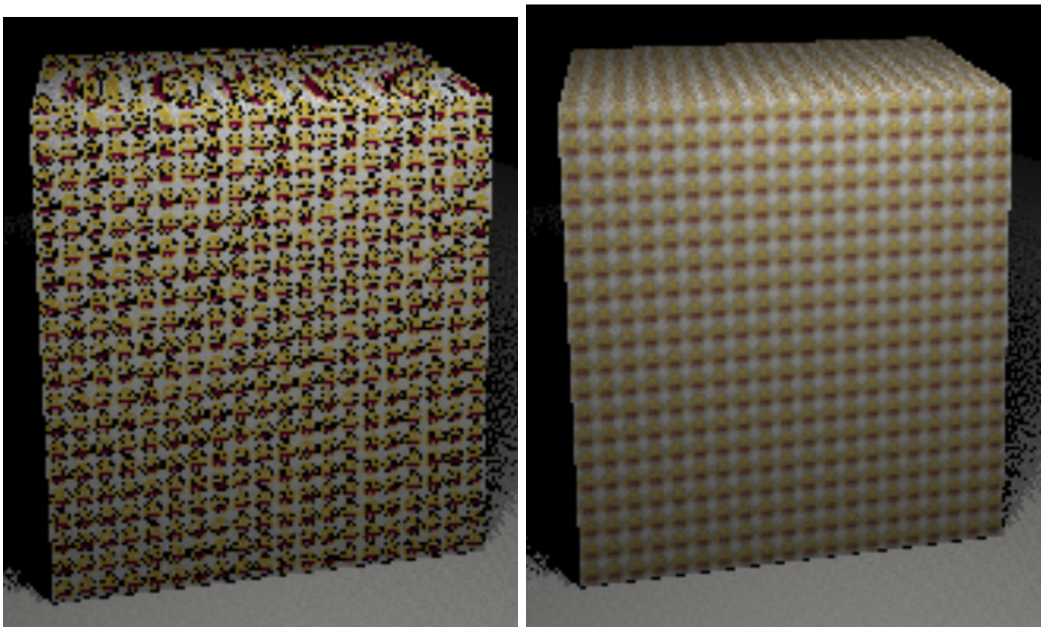
```
1   inline RayDifferential operator()(const RayDifferential &r) const;
```

而且在相机类中（正交投影相机和透视投影相机）的构造函数中要初始化 dxCamera 和 dyCamera。为了更好地体现效果，我们需要将采样点放在像素中心，所以下面的函数需要进行修改：

```
1   //Sampler::GetCameraSample
2   cs.pFilm = (Point2f)pRaster + Point2f(0.5f,0.5f);
```

分别使用普通光线和光线微分来渲染，分别得到下图的左和右。


可以看到有光线微分的渲染结果确实好很多。对于复杂的纹理而言更是如此：



本章就结束了。对于一些复杂的模型，其实可以通过 assimp[7] 来读取，只是需要写一个对应的接口，大家可以自己尝试实现一下（这里面有很多细节问题。可能甚至需要将 TriangleMesh 重新构建，因此比较麻烦），我会单独用一本小书进行介绍。

参考文献

[1] Pharr M, Jakob W, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2016.

[2] Shirley P. Ray Tracing in One Weekend[J]. 2016.

[3] Shirley P. Ray Tracing The Next Week[J]. 2016.

[4] Shirley P. Ray Tracing The Rest Of Your Life[J]. 2016.

[5] Igehy, H. . (1999). Tracing ray differentials.

[6] https://learnopengl.com/

[7] http://assimp.org/