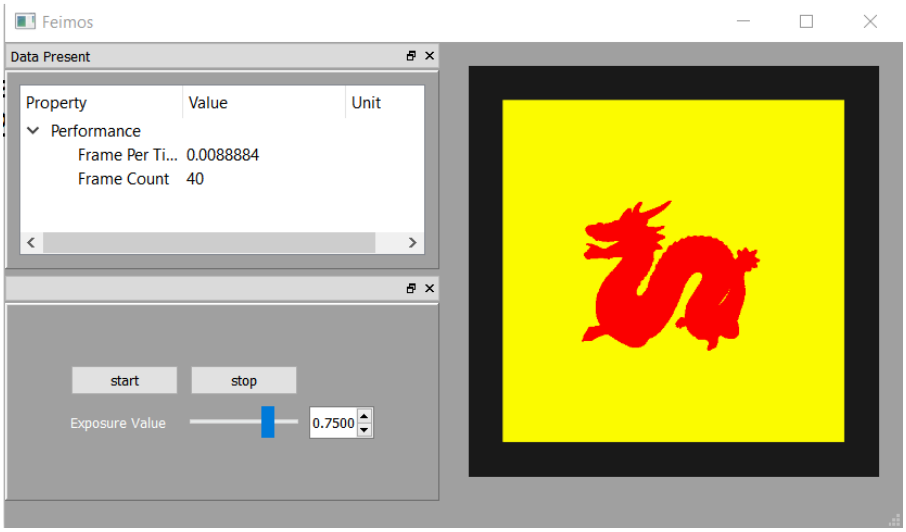
PBRT 系列 3-代码实战-形状和加速器

Dezeming Family

2021 年 1 月 28 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，最好从官方网站：https://dezeming.top/ 下载新的版本（免费下载）。

本书目标：移植 PBRT 基本的矩阵向量变换工具。学习 PBRT 的形状类和加速器包围盒。掌握 PBRT 的整个基元和加速结构的构建流程。将 PBRT 的形状类和加速器移植到自己的系统中，最终测试得到封面效果图。



本文于 2022 年 7 月 5 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]。

前言

按理来说，基本的向量工具这一部分应该是第一本来介绍的，最起码要放在场景文件解析后面，但是我将它放在了系列三，有两个原因。第一个原因，我当时构建这个系列的书时，并没有考虑这么多。第二个原因，也是最重要的原因，是因为既然我们可以很轻松地就理解 PBRT 中的 Whitted 渲染器流程，我们为什么不先实现这个渲染器呢。

我写的第一本小书是关于实现光子映射基本算法和实现的，当时我把以前的笔记整理了一下，然后增加了点基础介绍和调试方法，构成了第一本书。后来因为业余时间比较充足，打算再实现一个轻便简单的引擎，本想从 mitsuba 为参考，考虑到 PBRT 至少还有本书可以看，所以还是介绍 PBRT 吧。如果您在学习中遇到了困难，也不要灰心，哪怕是在图形学界摸爬滚打了一段时间的我，在写这个系列书时，也遇到了不少的困难。遇到困难虽然令人郁闷，但是这恰恰说明了你的工作是有价值的，简单的和重复性的工作东西大家都会做，怎么能提高你的能力呢。

其实我更想从头写一点一点构建光线追踪器的书，但是作为一个极致的完美主义者，我写东西和测试一般都追求面面俱到，所以如果要把构建光追器的过程和修改都写下来，估计得写个几千甚至上万页了。有些技术，真要展开讲清楚，不是几句话就能说完的。

本书开始，我们就自己去实现（移植，改造）PBRT 到自己的系统中。书中难免有些地方介绍的不够详细，大家可以配合着 PBRT 书 [1] 和虎书 [5] 来学习。移植的时候肯定会遇到很多问题，但把它们一个一个克服，当把这些东西移植到自己的系统上时，一种成就感油然而生。

本套系列书一共分为两个部分：

第一部分为 PBRT 基本知识《基础理论与代码实战》，一共 15 本书，售价 40 元，其中每本的售价都不同，本书的售价是 8 元（电子版）。我们不直接收取任何费用，如果其中某本书对大家学习有帮助，可以往我们的支付宝账户（17853140351，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！

第二部分为专业理论知识部分《专业知识理论与代码实战》，一共 12 本书，暂定售价 120 元，每本价格也各不相同。我于 2022 年 7 月，开始着手写这个系列，但在此之前，我会将第一部分的每本书都配备源码，然后再开始第二个部分的写作。

目录

一　基本介绍

这本小书我是这么规划的，先把最基本的 PBRT 向量矩阵工具介绍和移植（移植的时候我们需要测试一下），然后再实现球形，三角形等基本的形状类，最后再实现 BVH 树这种加速结构。

涉及光线微分技术、内存池管理技术和误差界定计算等的内容本书不作考虑，我们放在后面的系列书讲解。本书的代码实现也不需要使用到这三种技术。

向量工具类直接移植和使用 PBRT 的代码，这样不但方便而且保证了正确性。本书会首先讲解 PBRT 的向量矩阵工具，之后我们比较轻松就能将 PBRT 的矩阵向量工具应用到自己的项目里。

我们将在学习完形状类和加速器结构的内容后开始移植和编写测试代码到我们的系统中。我们会在架构和逻辑上遵循 PBRT 的相关实现，逐步在自己的渲染器中构建出仿 PBRT 的系统的架构。

本书会用到关于模型视点变换的内容 [5]，会用到矩阵相乘的知识，这些内容您可以从网上找到大量的资料，而且内容比较简单，所以就本书不会详细介绍。

二　最基本的几何和变换

这一章的思路很简单，能移植的我们绝对不手写。在以前看过的机器学习课程中，吴恩达说的一个观点让我印象很深刻，他说，对于一些数值计算的工具，尽量不要自己去实现，比如矩阵求逆。数值计算是一门单独的学科，如何在计算机的有限精度下表示我们计算的结果很重要，例如，你的采样 Ray 击中了某个包围盒，光线表示为 $o+t\omega$，那么计算的 t 就有可能不是精确值，这就出现了误差。对于求三角函数等，也会出现误差，使用通用的计算工具和专家写好的程序，会使这种误差可控。

2 1　点、向量

点和向量计算在 Geometry.h 和 Geometry.cpp 文件中。该文件包含了两个标准库头文件：

```
#include <cmath>
#include <cassert>
```

一个头文件是标准库的数学函数，另一个头文件是用来做断言的，即如果断言内容为真，就正常执行，否则直接退出程序，返回错误。我们可以考虑按照 PBRT 的方式给现有系统加个命名空间（我倾向于使用命名空间 namespace）。

首先是判断是否有非法数（比如除以 0 以后生成的数）的 isNaN() 函数。

对于向量，有的工程会用 vec3 来表示，有的工程会用 vec3f 来表示，我们这里直接用 PBRT 的表示方法，比如浮点型三维向量，就是 Vector3f。

向量 Vector 的类，有 2 维向量和 3 维向量，我们都可以移植和使用。operator« 操作符用来输出向量的信息到字符串中。

点 Point 和法向量 Normal 类也都是这样，可以直接移植过去。它们都包含了输出是否有非法数到调试日志的宏（CHECK_NE,DCHECK），我们可以选择移植过去，也可以选择保留功能，先使用没有任何作用的宏定义声明一下，等以后需要建立调试日志再说（调试日志也是一门学问，想写得比较规范也需要一定技术，这里不介绍这些，其实你打印输出结果也是可以的）：

```
    #define CHECK_NE(a)
    #define DCHECK(a)
```

注意 Geometry.h 文件包含了 pbrt.h，这里面有一些声明，我们在移植的时候也要加进去，否则就会出现一堆“已定义”和“未定义”错误：

```
    template <typename T>
    class Vector2;
    template <typename T>
    class Vector3;
    template <typename T>
    class Point3;
    template <typename T>
    class Point2;
    template <typename T>
    class Normal3;
```

还有一个问题是，在 PBRT 中，颜色的表示和向量是分开的。但是对于简易的系统来说，都是使用包含了 x,y,z 值作为 r,g,b 的三维向量，因为我们暂时还没有涉及颜色和光谱，所以我们也暂时使用 Vector3f 来表示颜色。

我们把除了操作符重载的其他向量和点的函数列在下面，有的函数参数可以是好几种，比如单位化函数通过重载应用于向量类和法向量类。

```
Dot //用于点乘操作
AbsDot //返回点乘的绝对值
Cross //求叉积
Normalize //求单位化的向量
MinComponent //返回向量（或点）中最小的x,y,z值
MaxComponent //返回向量（或点）中最大的x,y,z值
MaxDimension //返回向量（或点）中最大的x,y,z值的索引
Min //返回两个向量（或点）中最小的x,y,z值
Max //返回两个向量（或点）中最大的x,y,z值
Permute //返回输入向量x,y,z交换次序的值
Distance //求两点之间的距离
DistanceSquared //求两点之间距离的平方
Lerp //两点求线性和（具体功能请参考源码）
Min //两点各分量取最小值
Max //两点各分量取最大值
Floor //各分量向下取整
Ceil //各分量向上取整
Faceforward //取相同方向（具体功能请参考源码）
```

我们目前只需要这些函数，至于包围盒和光线 Ray 等函数我们暂时先不用考虑。我们把这些内容移植到自己的项目里，改改自己的系统使用到的函数，编译和测试成功即可。

2 2 变换

变换定义在 Transform.h 和 Transform.cpp 文件中。我们当前需要的变换主要就是矩阵变换，涉及旋转平移缩放矩阵等。

我们先不管与光线 Ray 有关的变换，也不管运动模糊的动态变换矩阵（主要是矩阵插值还得考虑四元数，以后如果我有时间会专门写一个四元数的原理和 PBRT 实现的小书，但是这里我不打算一次性搞这么多复杂的东西），我们只考虑最基本的变换。

首先是 struct Matrix4x4，里面的打印调试函数根据你的需要来更改。Matrix4x4 类里有两个重要的成员函数，求转置和求逆。

Transform 就是一个包含了一堆各种操作的类，那些包围盒、动态变换相关的内容我们暂时不需要，因此可以直接从该类中删掉。还有一些带着误差边界的函数我们都可以先去掉（如下代码第一行），因为有不带误差边界的版本（如下代码第二行）。

```
template <typename T> inline Vector3<T> Transform::operator()(const Vector3<T> &v,const Vector3<T> &vError, Vector3<T> *absError) const {}
template <typename T> inline Vector3<T> Transform::operator()(const Vector3<T> &v) const {}
```

最后再移植上旋转平移这种矩阵操作，以及 LookAt 变换，该变换将世界坐标系的物体变换到相机坐标系。移植完以后，编译一下，没有错误，那么这套比较标准的矩阵和向量运算工具就是我们自己的啦。插一句话，其实我们也完全可以使用例如 OpenCV、GLM 等库的矩阵向量运算工具，但为了尽量少增加第三方库，我们就使用 PBRT 的即可。

2 3 矩阵向量工具测试

我们测试一下这些变换到底是不是起作用（这些测试代码并不在源码中，我们每本书给出的源码是本书完成时的样子）。因为我们的场景暂时还没有三角形面片，都是 [2][3][4] 中的球体和矩形，因此我们就简

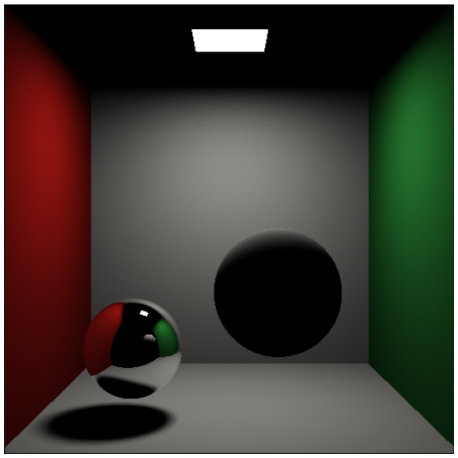
单测试一下矩形和球体能不能用。

（说句可能无关紧要的话。我之前在 [2][3][4] 构建的系统中实现了可移动相机，可能出现了左右颠倒错误，即左墙渲染到了右边，右墙渲染到了左边，如果您是按照 [2][3][4] 中构建的那样做的，就不会出现这种情况。）

还是先贴一下场景代码，如果不了解的可以看本系列的上一本书：《PBRT 系列 2-Whitted 光线追踪引擎》，我们把里面的立方体换成了球形，因为暂时没有针对坐标不对齐的立方体进行光线追踪求交的代码。

```
material *lightM = new diffuse_light(new constant_texture(Vector3f(12.0f, 12.0f, 12.0f)));
hitable *light_shape = new xz_rect(2.23, 3.33, 2.03, 3.13, 5.54, lightM);
hitable *cornell_box_PT() {
    hitable **list = new hitable*[100000];
    int index = 0;
    material *LeftWall = new lambertian(new constant_texture(Vector3f(0.630000, 0.065, 0.05)));
    material *RightWall = new lambertian(new constant_texture(Vector3f(0.12, 0.45, 0.15)));
    material *white = new lambertian(new constant_texture(Vector3f(0.725, 0.710, 0.680)));
    material *S = new lambertian(new constant_texture(Vector3f(0.725, 0.710, 0.680)));
    material *S1 = new metal(Vector3f(0.8, 0.8, 0.8), 0.0);
    //右左墙
    list[index++] = new flip_normals(new yz_rect(0, 5.55, 0.0, 5.55, 5.55, RightWall));
    list[index++] = new yz_rect(0, 5.55, 0.0, 5.55, 0, LeftWall);
    //光 天花板 地板
    list[index++] = new flip_normals(light_shape);
    list[index++] = new flip_normals(new xz_rect(0, 5.55, 0.0, 5.55, 5.55, white));
    list[index++] = new xz_rect(0, 5.55, 0.0, 5.55, 0, white);
    //前墙 后墙
    list[index++] = new flip_normals(new xy_rect(0, 5.55, 0, 5.55, 5.55, white));
    hitable *shortShpere = new translate(new sphere(Vector3f(0, 0, 0), 0.7, S), Vector3f(3.3, 2.0, 0));
    hitable *tallShpere = new translate(new sphere(Vector3f(0, 0, 0), 0.7, S1), Vector3f(1.4, 1.0, 2.0));
    list[index++] = shortShpere;
    list[index++] = tallShpere;
    return new bvh_node(list, index, 0.0, 1.0);
}
```

渲染得到如下结果（注意该图中 Lambertian 材质的球的位置在康奈尔盒外面，并没有遮住地板，所以没有阴影）：



我们运用一下移植的变换类：

```
        Transform t1 = Translate(Vector3f(3.3, 2.0, 0));
        Point3f c1 = t1(Point3f(0, 0, 0));
        Transform t2 = Translate(Vector3f(1.4, 1.0, 2.0));
        Point3f c2 = t2(Point3f(0, 0, 0));
        hitable *shortShpere = new sphere(Vector3f(c1), 0.7, S);
        hitable *tallShpere = new sphere(Vector3f(c2), 0.7, S1);
```

得到同样的渲染结果。

2 4 小节

通过这一章的学习，您应该已经可以自由使用 PBRT 的基本的几何和变换类了，但是也要注意，这只是最基本的几何变换。下一章我们研究如何编写自己的形状类。因为形状类会包含材质、纹理颜色等信息，我们还得将它们从中抽取出来，实现较为简单的 PBRT 形状。

三　PBRT 的形状类

这一章相对而言比较复杂，因为形状类左右相互牵扯，较难剥离。但是跟着本书一步一步走，相信您可以理所当然地对 PBRT 的形状类有个很好的掌控。

我们只需要实现里面的两个类就好了，一个是球体，一个是三角面片。球体最简单，可以快速掌握形状类的结构，三角面片可以表示最广泛的模型。我本想再讲解关于细分曲面的，但考虑到篇幅的原因，可能会留到以后作为补充专题。

3 1　形状类描述

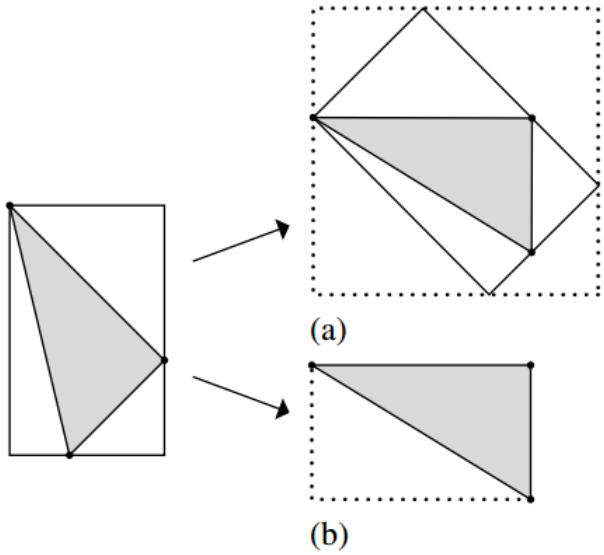
PBRT 的与形状有关的类是 Shape 和 Primitive 两个类，Shape 表示基本形状，Primitive 表示属性，将形状、材质等打包在一起，例如表面材质。我们这一章主要描述和应用形状类。

在 [2][3][4] 中我们可以看到，形状类内部应该有不少方法，例如计算面积，计算概率密度函数，求交，生成包围盒等，在 PBRT 的形状类中也有。

ObjectToWorld 变换类负责将模型的顶点位置变换到世界坐标系的位置（下一章会专门介绍）。写过形状类就知道，法向量方向决定了物体的外侧，在 PBRT 中，bool 变量 reverseOrientation 就表示当前法向量是否应该是反转的（PBRT 场景文件中可以设置，鉴于我们一般用不到，大家现在默认法向量是不需要反转的（直到系列 9 我们都可以认为法向量是不需要反转的））。

在 PBRT 的 Shape 类中有指针指向平移旋转等变换的 Transform 类，因为可能很多形状都共用同一个变换类，因此这样可以节约空间。在 PBRT 中的形状类里也有从世界坐标变换到局部坐标的类，PBRT 的形状类都有自己的局部坐标，比如球体的局部坐标中，球心位于坐标原点处。

包围盒是非常重要的结构，但因为包围盒主要是用在加速器结构上，因此这里只是简单介绍一下。Bounds3::IntersectP() 是包围盒求交的方法。PBRT 有两种包围盒，一种是局部坐标系下包围盒，一种是世界坐标系下的包围盒，两者可以互相转换，但是有些情况并不适用，比如下图中的三角形，先在局部生成的包围盒再变换到世界坐标就不是轴对称了，而先变换到世界坐标再生成包围盒会更好。



Shape 类中的求交分为两种，一是与 Shape 求交并需要知道其表面信息的 Intersect()，二是只需要知道 Ray 是否与 Shape 相交的 IntersectP()。

对于 Intersect() 函数，我们在《Whitted 光线追踪》书中已经了解过了一些内容，这里还是简单介绍一下。Intersect() 函数的参数中，tHit 可以理解为 Ray 从原点 o 到击中物体的距离（如果 Ray 的方向 d 是单位向量的话），SurfaceInteraction 表示交点的属性，例如法向量，表面材质计算的 bsdf 等。IntersectP() 简单做法是直接把参数 SurfaceInteraction 和 tHit 设为 nullptr，这样就不会得到表面交点属性信息了，但很效率很低，因此每个继承自 Shape 类的派生类都实现自己的 IntersectP() 方法。

顺便提一句，许多渲染系统，特别是那些基于扫描线或 z 缓冲区算法的渲染系统（如 OpenGL），都支持形状是“单面”的概念，即从前面看形状是可见的，但从后面看形状则消失了。特别是，如果一个几何对象是封闭的，并且总是从外面看，那么它的背面部分可以被丢弃，而不会改变结果图像。这种优化可

以大大提高这些类型的隐藏面去除算法的速度。但是，将此技术与光线跟踪结合使用时，性能提高的可能性会降低，因为在确定曲面法线以进行背面测试之前，通常需要执行光线与对象的相交。

此外，如果单侧对象实际上没有闭合，则该特征可能导致不一致的场景描述。例如，当阴影光线从光源跟踪到另一个曲面上的点时，曲面可能会阻挡灯光，但如果阴影光线是在另一个方向跟踪的则不会阻挡灯光，例如 PBRT 支持只有一半的球体，因此反面剔除会影响正常显示效果。由于所有这些原因，PBRT 不支持反面剔除功能。

3 2 球体类

PBRT 支持只有部分区域的球，类似于球体被劈开了。但是个人感觉现在全都介绍则比较啰嗦，因此这种情况我们不考虑，我们只研究完整的球体。其他的形状例如圆柱圆盘之类的我们现在也不考虑。球体类中的下面几个变量都是用来表示一个不完全的球体的，因此我们可以忽略这些变量参与到的计算：

```
1    const Float zMin, zMax;
2    const Float thetaMin, thetaMax, phiMax;
```

球体的包围盒非常简单，就是中心点的三轴坐标都加上半径以及都减去半径。空间中与球体求交的计算也非常简单，[2][3][4] 中有详细介绍，这里避免重复，不再赘述（其实就是判断二次方程有没有实数根的问题）。

球体类的表面求交形继承自形状类的表面求交方法可以参考 [2]，没有什么可说的内容。鉴于我们不考虑光线微分反混淆，因此球体类的实现并不会很复杂。同时我们不考虑不完全的球，因此就更简单了。

3 3 三角形类概述

本节是重点，[2][3][4] 中并没有实现三角形，尽管三角形的实现也同样非常简单，但一个模型通常有上万甚至百万个面片构成，如何规划该类，提高渲染效率是非常重要的。尽管我们有加速器机制，但形状本身的存储和表示也同样很重要。所以，虽然空间中直线与三角形求交的问题只是个初高中的几何问题，但我们在程序中，最重要的是保证效率。

本节只是对三角面片进行简单的讲解，下一节会开始介绍内部实现细节。

PBRT 存储形状类时，会把顶点坐标单独存在数组里，而每个三角形的三个顶点用索引的形式存储在每个三角形对象内，这样可以极大地节省空间（具体能节省多少空间可以去参考 PBRT 书第三章第 6 节）。三角面片存储在结构 TriangleMesh 中，而具体的三角形形状类表示为 class Triangle : public Shape。

我们首先介绍的是 TriangleMesh 结构，该结构其实通过构造函数就能了解其机理：

```
1 TriangleMesh(const Transform &ObjectToWorld, int nTriangles,
2 const int *vertexIndices, int nVertices, const Point3f *P,
3 const Vector3f *S, const Normal3f *N, const Point2f *uv,
4 const std::shared_ptr<Texture<Float>> &alphaMask,
5 const std::shared_ptr<Texture<Float>> &shadowAlphaMask,
6 const int *faceIndices);
```

nTriangles 表示一共有几个三角形，vertexIndices 表示存储顶点索引的数组，nVertices 表示一共有多少个顶点，P 表示存储这些顶点的数组，S 表示切线向量，用来做切线着色（暂时我们不予考虑，不过如何您做过 OpenGL 开发，想必会对切线空间，法向量映射等非常熟悉），N 表示法向量（注意有法向量的模型文件中每个顶点都有对应的法向量，面上的非顶点位置法向量由三个顶点的法向量插值得到），uv 表示纹理坐标（每个顶点都有对应的纹理坐标）。而且在该构造函数里，局部坐标，法向量和切线向量会被直接变换到全局坐标中，这样就避免了在程序执行时需要多次重复变换。

然后我们再介绍 Triangle 类。完整粘贴一下该类的构造函数：

```
1 Triangle(const Transform *ObjectToWorld, const Transform *WorldToObject,
```

```
2             bool reverseOrientation, const std::shared_ptr<TriangleMesh> &
                  mesh,
3             int triNumber)
4     : Shape(ObjectToWorld, WorldToObject, reverseOrientation), mesh(mesh) {
5     v = &mesh->vertexIndices[3 * triNumber];
6     triMeshBytes += sizeof(*this);
7     faceIndex = mesh->faceIndices.size() ? mesh->faceIndices[triNumber] : 0;
8 }
```

初始化的几个成员变量如下：

```
1 std::shared_ptr<TriangleMesh> mesh;
2 const int *v;
3 int faceIndex;
```

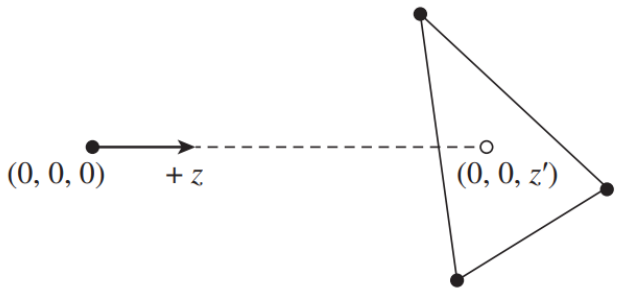
triNumber 表示是面片列表中的第几个三角形，因为一个三角形有三个顶点，所以该值赋值给表示指向顶点的指针 v 时地址要乘 3。triMeshBytes 记录了全部的三角面片对象占用的内存量。

PBRT 场景中可能会有大量三角面片，因此，一个函数 CreateTriangleMesh 随之出现，该函数的作用是把 TriangleMesh 内存储的三角形建立成一个个 Triangle 对象。

3 4 三角形求交

求交过程相对前面来说比较复杂。尽管网上您可以搜到各种三角面片求交的算法和代码（给定三角形的三个顶点位置和一个直线的向量表示，求是否相交及交点位置），并可以比较简单的移植到自己的系统里，但因为我们研究的对象是 PBRT，因此我们就学习一下 PBRT 的实现过程。

Triangle::Intersect() 是一个二百多行的函数，这是因为里面考虑了很多细节问题，我们需要从中将我们需要的内容剥离出来。求交涉及一系列的变换，比如由世界坐标转换到 ray 坐标，然后得到如下图的表示。



之后问题就变成了，(0,0) 点是否在变换后的三角形坐标内部的问题。我们暂时不需要知道它是怎么变换过去的，因为 PBRT 有一套严谨的坐标转换系统，我们下一章会重新进行讲解。

Intersect() 函数还会将计算的系数传入 SurfaceInteraction 里的 SetShadingGeometry 函数来计算插值后的法向量和切线向量。鉴于我们现在也用不到这些东西，所以这里也不用理会。

三角形面积的计算需要用到叉积，输入三个点的位置，得到面积。

3 5 圆柱形

这里并不详细介绍圆柱，只是简单说一下怎么做圆柱与光线求交，因为与球体和三角形都不同。

圆柱在没有变换前，圆柱的中心在坐标原点，计算较为容易，大家可以去 [1] 看细节。当圆柱进行物体变换后（例如旋转），这时首先使用 WorldToObject 函数，将世界坐标系下的 ray 变换到物体坐标系中，然后再进行求交。即比如圆柱经过了平移变换，向 x 轴方向平移了 +3 单位长度，则 WorldToObject 函数就会将 ray 向 x 轴方向平移-3 单位长度，然后再与圆柱计算求交。

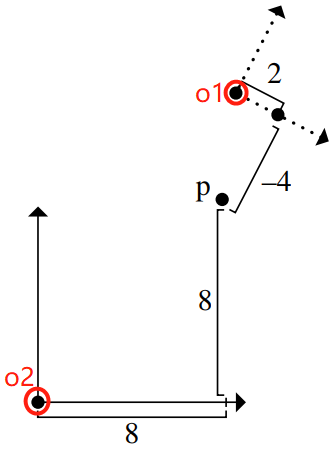
3 6　小节

本章简单讲解了 PBRT 的形状类以及里面的功能，具体实现方法将在后面（下一本关于光线微分的书中）进行介绍。

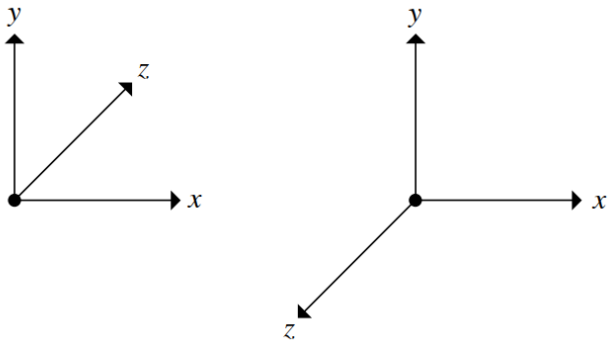
四　PBRT 坐标系统及三角面片求交

任何坐标表示的点，都可以写为坐标原点 + 向量（该点与坐标原点的相对位置）。而向量的表示则一般用坐标基表示，例如通常三维向量会使用 x,y,z 三轴的单位向量作为基。用 x,y,z 基表示的点坐标，我们这里称为世界空间下的世界坐标点。

坐标之间可以相互转化，比如下面的坐标系中，用 o1 为坐标原点, 在其坐标系中表示点 p 的位置是 (2,-4)，用 o2 为坐标原点表示点 p 的位置是 (8,8)。而在整个世界坐标中，可以设以 o1 坐标系的点 p 表示为 o1+2*ox1+(-4)*oy1，设以 o2 坐标系的点表示为 o2+8*ox2+8*oy2。



关于坐标变换的内容在虎书《计算机图形学基础》[5] 有很详细的讲解。PBRT 采用的是左手坐标系，即下图中左边的表示。而 OpenGL 中一般会使用右手坐标系，如下图中右边的表示（但其实我们不用太在意哪种坐标系，至少直到系列 9 结束之前，我们都不需要区分左手坐标系还是右手坐标系）。



当我们有两个相互垂直的单位向量时，我们通过叉积可以产生第三个向量，这三个相互垂直的向量构成一个坐标系统，通过 PBRT 中的 CoordinateSystem 函数来实现。我们可以计算一下来增加印象，(1,0,0)X(0,1,0)=(0,0,1)，（当为左手坐标系的时候，(0,0,1) 指向屏幕里面，而当为右手坐标系的时候，(0,0,1) 指向屏幕外，但 (1,0,0) 和 (0,1,0) 的叉积结果永远是 (0,0,1)）。

4 1　几个不得不说的变换函数

在 PBRT 中，我们经常会遇到这几个函数：

```
1 cameraToWorld
2 ObjectToWorld
3 WorldToLocal
4 LocalToWorld
```

cameraToWorld 函数保存在 Transform 类中，由 LookAt 函数构建（该函数生成世界坐标到相机坐标的变换，因此 cameraToWorld 是该变换的逆变换），因为 Transform 中保存了变换和逆变换，故可以从世界坐标变换到相机坐标系，也可以从相机坐标系变换到世界坐标系中。如果不知道什么是相机-世界空间变换，可以参考 [5] 或者网络上的讲解内容。

```
//api.cpp
Transform lookAt =
    LookAt(Point3f(ex, ey, ez), Point3f(lx, ly, lz), Vector3f(ux, uy, uz
        ));
//FOR_ACTIVE_TRANSFORMS 是一个for循环
FOR_ACTIVE_TRANSFORMS(curTransform[i] = curTransform[i] * lookAt;);
```

我们在《文件加载与设定》书中简单说过“变换”是怎么读取的，我们回顾一下文件加载的流程：curTransform 表示当前的变换（api.cpp 文件），对于全局变换的表示，例如《文件加载与设定》中恐龙的场景文件的基本设置中，读到 Rotate 时，会为 curTransform 赋值（curTransform 表示物体坐标与世界坐标之间的变换）。当 parse 到 WorldBegin 后，当前 curTransform 进栈保存；然后 curTransform 会被初始化为单位矩阵变换。在每个 AttributeBegin 和 AttributeEnd 变换对中，都会有自己的物体变换。

在 pbrtShape() 函数（api.cpp）中，我们可以看到，在构建形状的时候，MakeShapes 用到的 ObjToWorld 和 WorldToObj 为 curTransform。transformCache 的 Lookup 是把当前的变换加载到缓存里，如果已经有了当前的变换，就返回指针，否则就加载当前变换，然后返回指针（可能有多个形状共用同一个变换；transformCache 当前是使用了哈希表存储，而 [1] 书发布时用的是 std::map()）。

```
Transform *ObjToWorld = transformCache.Lookup(curTransform[0]);
Transform *WorldToObj = transformCache.Lookup(Inverse(curTransform[0]));
std::vector<std::shared_ptr<Shape>> shapes =
MakeShapes(name, ObjToWorld, WorldToObj,
           graphicsState.reverseOrientation, params);
```

这里的 ObjToWorld 就是模型怎么变换到世界坐标，因为我们的模型内部的顶点都是相对于自身的位置，而需要放在世界坐标的何处位置则需要借助该变换。前面我们还介绍过 WorldToObject 函数，该函数将世界坐标系统的物体变换到物体自身坐标系。

WorldToLocal 是为了方便计算 BxDF（包括 BSDF,BTDF 等）的，在局部空间中，设表面的法向量方向为 (0,0,1)（顺便提一句，现在不知道是什么意思也完全没有关系：主切线和次切线分别为 (1,0,0) 和 (0,1,0)，这三个轴互相垂直），所以该变换将世界坐标系中的方向向量变换到局部空间中（这样再做点积运算会效率非常高）。LocalToWorld 是其反变换，注意该过程并不是可逆的，因此需要用到两个完全不同的函数。具体的工作方式我们暂时不提，直到我们需要研究 BSDF 时，我们才会用到该变换，到时候再进行详细推导。

4 2 三角面片求交中的求交变换

上一章只是做了一点简单的介绍，这里会详细解释如何做求交。通过仿射变换使 ray 的原点在 (0,0,0)，方向是 z 轴正方向，三角形也会变换到该空间中（上一章已有示意图）。因为求交后的点的 x 和 y 的坐标值都是 0，因此在计算插值等都会很容易。方便起见，我们把该坐标空间称为 RTC 空间（Ray-Triangle intersection coordinate system）。

转换到 RTC 空间需要三步，一个变换矩阵 T，一个坐标交换矩阵 P，一个剪切矩阵 S，我们不需要将三个矩阵相乘计算一个综合矩阵 M=SPT，而是分步作用于三角形的三个顶点。

T 最简单，就是把三角形顶点进行移位：

```
Point3f p0t = p0 - Vector3f(ray.o);
Point3f p1t = p1 - Vector3f(ray.o);
Point3f p2t = p2 - Vector3f(ray.o);
```

为了计算更准确，进行坐标交换，令 z 为坐标绝对值最大的值，x 和 y 任意分配，这样就能保证 z 方向不为 0。同理，用矩阵计算太麻烦，直接这样即可：

```
int kz = MaxDimension(Abs(ray.d));
```

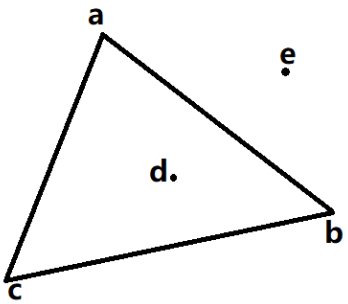
```
    int kx = kz + 1; if (kx == 3) kx = 0;
    int ky = kx + 1; if (ky == 3) ky = 0;
    Vector3f d = Permute(ray.d, kx, ky, kz);
    p0t = Permute(p0t, kx, ky, kz);
    p1t = Permute(p1t, kx, ky, kz);
    p2t = Permute(p2t, kx, ky, kz);
```

剪切变换则是变换到以 ray 方向视角的位置，dx,dy,dz 表示 ray 的方向向量 d 的世界坐标系下的三个分量：

$$S = \begin{pmatrix} 1 & 0 & -d_x/d_z & 0 \\ 0 & 1 & -d_y/d_z & 0 \\ 0 & 0 & 1/d_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

该变换将正对着 ray 的方向变为 (0,0,z1)，z1 取决于 T 和 P 变换后的位置。比如某个点经过 T 和 P 变换后，为 [dx,dy,dz,0]，则经过 S 变换后正好为 (0,0,1)，该点离着 ray 在 RTC 坐标空间的原点 (0,0,0) 距离为 1。S 变换也可以不使用矩阵，直接用程序来实现，这里不再粘贴代码。实际实现的时候，PBRT 先计算了每个三角形顶点的 x 坐标和 y 坐标的 S 变换，之后等求交测试后，如果相交则再计算 z 坐标的 S 变换。

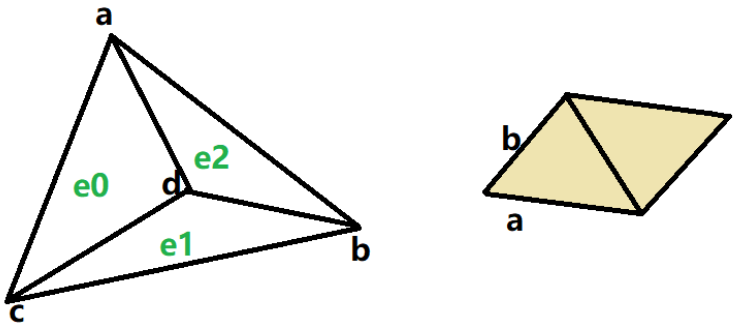
接下来就是判断 (0,0) 点在不在投影后的三角形内部了。判断一个点是否在三角形内部很简单，如下图。我们遍历三个点，比如顺序是 b-a-c，可以构成三个向量，即 ba,ac 和 cb，这三个向量都与当前点到检测点的向量进行叉积，即 $\overrightarrow{bd}X\overrightarrow{ba}$,$\overrightarrow{ad}X\overrightarrow{ac}$ 和 $\overrightarrow{cd}X\overrightarrow{cb}$，如果三个值同号，则表示 d 在三个边的同侧，则 d 就在三角形内部，否则 d 就在三角形外部。



在程序中表示为计算这三个量：

```
    Float e0 = p1t.x * p2t.y - p1t.y * p2t.x;
    Float e1 = p2t.x * p0t.y - p2t.y * p0t.x;
    Float e2 = p0t.x * p1t.y - p0t.y * p1t.x;
    ......
    Float det = e0 + e1 + e2;
```

如果正好击中了边界，则会有为 0 的值，这时我们进行判断，如果与三个边的叉积都是 0，则说明该三角形投影后是一条线，此时光线与三角形“擦肩而过”，我们认为它们并不相交。如果相交了，则计算 z 轴的剪切 S 变换值。注意两个向量叉积的绝对值是计算这两个向量构成的平行四边形的面积（如下图的右图），因此就可以得到击中点的重心坐标插值后的 z 值（因为重心坐标与比例有关，故我们不需要通过平行四边形除以 2 来得到三角形的面积）。



之后再计算相交点得到的 ray 中 o+t*d 中的 t 的值是否大于 t_{max}，大于则返回 false。这里 t 的计算也用到了重心插值。注意 PBRT 为了效率，首先计算的是 t 只是加权：比如当 e 都大于 0 时，e0*z0+e1*z1+e2*z2 要小于 tmax*det。如果相交的 t 值在设定范围内，则计算真正的 t：

```
Float tScaled = e0 * p0t.z + e1 * p1t.z + e2 * p2t.z;
//判断相交的t是否在范围内
......
Float b0 = e0 * invDet;
Float b1 = e1 * invDet;
Float b2 = e2 * invDet;
Float t = tScaled * invDet;
```

此时的 b0,b1,b2 就是用来做三角形三个顶点的重心插值的（barycentric coordinates interpolation）。至此，三角形面片求交的算法已经全部介绍完毕，下面开始介绍加速器结构。

五 Primitives 类

PBRT 实现了两种加速结构，分别是 BVH 树和 kd 树，鉴于 kd 树在图形学的空间划分中因为存在各种不太方便和优化的地方，已经有点过时了，所以我们忽略它。我们仅研究 BVH 树的构建。在研究加速器前，我们首先需要知道什么是 Primitives 类以及其作用。

我们在本系列第一本书《文件加载和设定》中简单提到过，场景文件中读入的形状集会根据其类型被补充到数组 renderOptions->primitives 中。我们找到 core 文件夹下的 primitive.h 头文件，里面有一个基类 Primitive，以及三个派生类 GeometricPrimitive，TransformedPrimitive 和 Aggregate。

Primitive 用来表示形状、材料等一个物体的综合信息，基类的函数包括：

```
1  WorldBound
2  Intersect
3  IntersectP
4  GetAreaLight
5  GetMaterial
6  ComputeScatteringFunctions
```

本章我们会详细介绍 Primitive 类及其派生类。

5 1 包围盒

包围盒一共有两种，分别是 2D 和 3D 包围盒，在 geometry 文件里。

有一些关于 2D 包围盒的函数，包括一些成员函数和调用函数，例如求包围盒的并集包围盒，交集包围盒等方法：

```
1  Union
2  Intersect
3  Overlaps
4  Inside
5  InsideExclusive
6  Expand
```

Bounds2iIterator 是一个包围盒迭代器，因为 PBRT 中好像没有用到该迭代器，我们也不去研究和实现这个东西。

3D 包围盒也有各种操作函数。Bounds3 类中的 IntersectP 函数的参数 Ray 我们还没有实现 PBRT 中的相关函数，所以根据自己的实现进行修改（该函数表示包围盒与 Ray 求交的判断）。

我们把这两个类以及相关的操作函数都移植到自己的工程的 Geometry 文件里。如果您还想详细了解一下里面的功能，可以从 [1] 中的第二章中找到相关内容。

Transform 对包围盒的变换也需要进行移植，即该函数：

```
1  Bounds3f Transform::operator()(const Bounds3f &b) const
```

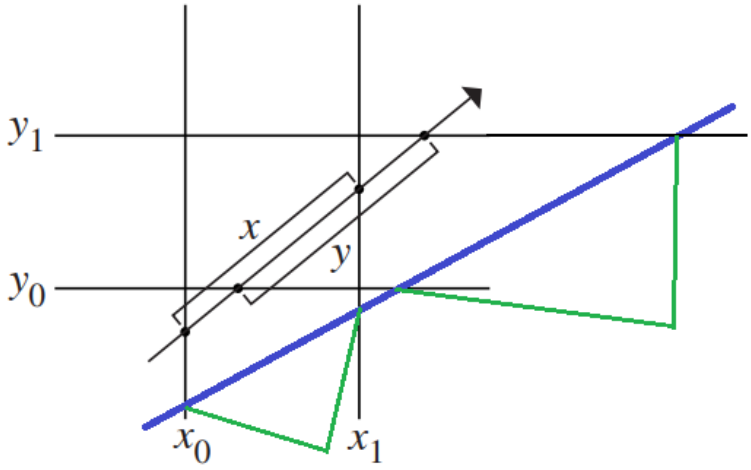
5 2 Ray 类

后面的系列中讲解光线微分时还会重新介绍，这里我们只是简单介绍 Ray 的结构。之前在文件加载中我们已经介绍过了。

主要就是两个向量，原点 o 和方向 d。还有记录当前时间的 time 和最大时间 tMax，用来处理运动模糊。

5 3 Ray 与包围盒相交测试

Bounds3<T>::IntersectP 函数返回 bool 值。因为包围盒是轴对齐的，所以求是否相交非常简单。令 Ray 的 t 值取值范围在 $(0, \infty)$ 之间，计算其在每个轴的近交点和远交点。2D 情况时，Ray 与包围盒相交的结果表示如下：



分别与 x 轴和 y 轴相交的点如上图，与各个轴相交后近交点中最远的值如果小于远交点中最近的值，则表示相交了。上图中黑线与轴相交，蓝线并没有相交。代码见 geometry.h 头文件，这里的 i 分别取值 0,1,2，表示 x,y,z 三轴：

```
1     Float invRayDir = 1 / ray.d[i];
2     Float tNear = (pMin[i] - ray.o[i]) * invRayDir;
3     Float tFar = (pMax[i] - ray.o[i]) * invRayDir;
4     if (tNear > tFar) std::swap(tNear, tFar);
```

如果除以 0 了应该怎么办呢？对于 C++ 来说，除以 0 以后会产生 NaN 值（非法数据），包含有 NaN 的判断语句都会返回 false，因此该逻辑可以用到代码中。完整的求交方法见 PBRT 代码。

PBRT 还提供了另一种 Bounds3<T>::IntersectP 方法如下，它的输入参数包含 Ray 的逆变换 invDir，而不用再在程序里计算逆。同时还包括另一个参数 dirIsNeg[3]，指示每个方向分量是否为负，这使得可以消除原始例程中计算的 tNear 和 tFar 值的比较，并且直接计算各自的近值和远值。

```
1 template <typename T>
2 inline bool Bounds3<T>::IntersectP(const Ray &ray, const Vector3f &invDir,
      const int dirIsNeg[3]) const {
3     const Bounds3f &bounds = *this;
4     Float tMin = (bounds[dirIsNeg[0]].x - ray.o.x) * invDir.x;
5     Float tMax = (bounds[1 - dirIsNeg[0]].x - ray.o.x) * invDir.x;
6     Float tyMin = (bounds[dirIsNeg[1]].y - ray.o.y) * invDir.y;
7     Float tyMax = (bounds[1 - dirIsNeg[1]].y - ray.o.y) * invDir.y;
8     ......
9 }
```

在 BVH 树中广泛使用了第二种方法，因为里面涉及海量包围盒求交运算，根据 PBRT 书 [1] 所述，该方法能够提高百分之 15 的效率。

5 4 Shape 类与包围盒绑定

Shape 里有两个函数，ObjectBound 和 WorldBound。我们已经知道，三角形会在初始化的时候变换到世界坐标系下，因此 ObjectBound 方法需要通过 WorldToObject 变换将世界坐标系的点变换到物体坐标系下，然后再返回该包围盒。

Shape 类里并没有记录其包围盒的变量对象。一个几何物体的全部特性（包括包围盒，BSDF 等）都在 Primitive 类里面，具体一点来说，是 GeometricPrimitive 类，该类的成员函数与父类 Primitive 一样。

```
1  WorldBound
2  Intersect
3  IntersectP
4  GetAreaLight
5  GetMaterial
6  ComputeScatteringFunctions
```

Intersect 和 IntersectP 就是调用了内部 Shape 对象的求交方法。GetAreaLight 返回的是该 Primitive 的 emission 属性，即发光值。GetMaterial 返回的是几何基元的材料。其他与 BRDF 等相关的东西我们暂时不提，会在以后的系列书中介绍。

TransformedPrimitive 同样继承自 Primitive 基类。Shape 本身就应用了对象到世界的变换，以将它们放置在场景中，而有些物体是可以运动的，而且可能一个物体包含一堆基本物体，例如满都是草的草坪。如果一个草坪是由几株草经过变换平移构成的，将所有变换后的基元进行存储则会消耗大量内存，因此 TransformedPrimitive 就是为了处理这样的场景而存在的。因为我们暂时不涉及实例化和运动型物体，因此暂时不考虑这一方面。

5 5　Aggregate 类

对于光线追踪加速结构，[2][3][4] 已经有过详细的描述。PBRT 的加速器有两种，一种是基于空间划分的 kdTree，另一种是基于对象划分的 BVHTree。Aggregate 类继承自 Primitive，它把多个 Primitive 对象组织到一起，构成一个 Primitive 整体，这样 Integrator 积分器的输入就只是一个 Aggregate 对象了。在 Aggregate 对象中，ComputeScatteringFunctions，GetMaterial 和 GetAreaLight 成员函数都不能被调用，如果调用了就会报错。

Aggregate 类同样有其派生类，即 accelerators 文件夹下的 BVHAccel 和 KdTreeAccel。

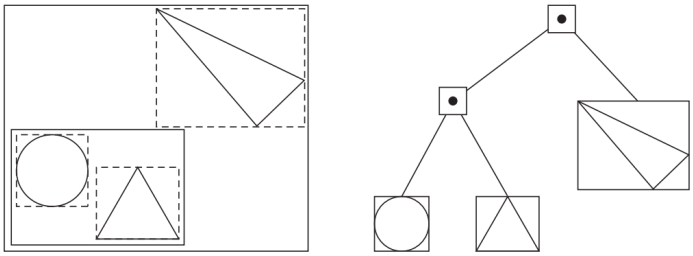
该铺垫的内容都讲完了，Shape，包围盒和 Primitive 的有关内容就是这些。下一章我们就开始研究加速器结构的实际实现。

六　加速器结构

这是本书至今为止最难又最简单的一章了。BVH 树的构建其实并不难，PBRT 书 [1] 讲得也非常详细，而且 [2][3][4] 也构建了一个比较简单的 BVH 树。本来我不是很想详细地写这一章，但是考虑到完整性，还是花了很多笔墨在这上面。说它最难是因为牵扯到数据结构，而简单是因为它的功能非常单一，基本方法也非常固定，与其他模块之间的相互牵扯比较少，因此比较容易叙述（写其他章节的时候，怎么把一步步需要掌握的内容进行剥离实在是痛苦而且麻烦的事情）。本章按照 [1] 的 4.3 节讲述顺序进行。

6 1　BVH 树简介

把物体根据位置绑定包围盒，一个简单的场景表示如下：



在树的末梢称为叶节点，其他节点到内部节点。对于 n 个 Primitives 基元的场景，构建 BVH 树后就有 n-1 个内部节点，总共 2n-1 个节点。建树的划分方法有四种，如下代码，一种是“surface area heuristic”（SAH），使用效率最高；另一种是“HLBVH”方法，构建的效率更高，也更容易并行化，但是使用上效率不如上一种；剩下两种建树时计算更少，但是使用时的效率也更低。这四种方法，除了 HLBVH 以外，我们会在本章后续小结进行介绍。

```
1    enum class SplitMethod { SAH, HLBVH, Middle, EqualCounts };
```

在构造函数中我们可以看到划分方法默认是 SAH，每个 Node 最大的 Primitive 数为 1，p 表示包含了基元的 vector。

```
1    BVHAccel(std::vector<std::shared_ptr<Primitive>> p,
2             int maxPrimsInNode = 1,
3             SplitMethod splitMethod = SplitMethod::SAH);
```

6 2　构建 BVH 树

建树分三步，第一步，计算每个基元的绑定盒，并存储在数组里；第二步，使用 splitMethod 方法建树；第三步，转化为更紧凑的方式。

每个存储到 BVH 树里的基元将其包围盒的质心、完整包围盒及其在基元数组中的索引存储在 BVHPrimitiveInfo 对象中。BVHPrimitiveInfo 是一个结构，里面有三个成员变量，size_t primitiveNumber 表示索引，Point3f centroid 表示质心，Bounds3f bounds 表示包围盒。

之后，如果我们选择的是 HLBVH 方法，则调用 HLBVHBuild() 函数来建树，否则其他三种建树方法都调用 recursiveBuild() 方法来建树。建树后，会排列出一个新顺序的 Primitive 数组，存储在 orderedPrims 里面，需要用 swap 方法交换一下：

```
1    primitives.swap(orderedPrims);
```

BVHBuildNode 是表示节点的结构，children 数组表示左右子节点，如果都是 nullptr，则说明此节点为叶节点。firstPrimOffset 表示第一个基元在 primitives 数组的位置索引，nPrimitives 表示该叶节点一共有多少个基元，一般是 1 个，PBRT 也支持多个基元划分到同一个叶节点上（后面会说到，当当前区间的所有基元的包围盒质心都在同一个点上时，就会划分到同一个叶节点上）。

```
struct BVHBuildNode {
    // BVHBuildNode Public Methods
    void InitLeaf(int first, int n, const Bounds3f &b);
    void InitInterior(int axis, BVHBuildNode *c0, BVHBuildNode *c1);
    Bounds3f bounds;
    BVHBuildNode *children[2];
    int splitAxis, firstPrimOffset, nPrimitives;
};
```

6.3 递归建树

递归建树的程序为 recursiveBuild() 方法，该方法参数列表如下：

```
BVHBuildNode *BVHAccel::recursiveBuild(MemoryArena &arena, std::vector<
    BVHPrimitiveInfo> &primitiveInfo, int start, int end, int *totalNodes
    , std::vector<std::shared_ptr<Primitive>> &orderedPrims)
```

arena 是用来在运行时分配内存的，我们暂时不管这玩意，等到后面的书我们再详细解释。在这里 arena 唯一的作用就是给节点分配内存：

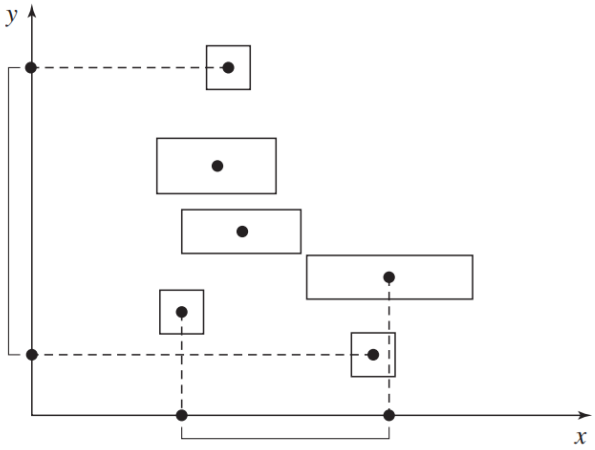
```
BVHBuildNode *node = arena.Alloc<BVHBuildNode>();
```

此外，start 和 end 参数表示 primitiveInfo 数组子集的范围 $[start, end)$。totalNodes 参数追踪 BVH 树已经构建好的节点个数，该值用于后面构建更紧凑的 BVH 树表示。orderedPrims 数组来存储已经存储在叶节点上的基元的索引，最初它是空的，当有叶节点被创建就 push 到该数组中的末尾。上面说过，当 recursiveBuild 调用完后，orderedPrims 会通过 swap 函数替换构建 BVHAccel 时的基元数组 primitives。整体的代码表示如下：

```
BVHBuildNode *node = arena.Alloc<BVHBuildNode>();
(*totalNodes)++;
//计算从start到end的整个包围盒
Bounds3f bounds;
for (int i = start; i < end; ++i)
    bounds = Union(bounds, primitiveInfo[i].bounds);
int nPrimitives = end - start;
if (nPrimitives == 1) {
    //创建当前节点为叶节点
    ......
} else {
    //选择划分方法
    ......
    //把基元primitives划分到两个子集，并构建子集节点（children）
    ......
}
return node;
```

创建当前节点为叶节点的方法可以从 PBRT 源码中找到（bvh.cpp），因为很简单，所以不再赘述。

创建内部节点时，primitives 会被分为两部分。首先会计算子区间的包围盒的质心变化范围，我们选择三个轴中的一个来划分基元，通常选质心变换范围在三个轴上最大的来划分。用 2 维图简单表示一下：



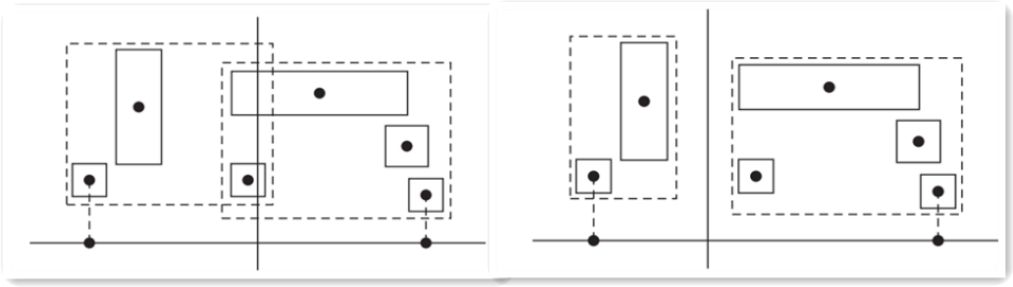
该区间内的物体在 y 轴的质心变化范围更大，因此选择用 y 轴进行划分。如果质心都在同一个位置，则直接创建叶节点，将该区间所有物体都放在一个叶节点里。否则就使用 splitMethod 进行划分，然后分别对两个区间递归调用 recursiveBuild。

6 4　子集划分方法：splitMethod

本节详细介绍 4 种 splitMethod 方法。

6 4.1　中位数方法

首先，找到该区间（即 $[start, end)$]）质心位置的中间值 (质心包围盒在划分轴的中间值)，然后把大于中间值的划分到一个区间内，小于中间值的划分到另一个区间内。但是可能会造成包围盒的重叠，重叠的包围盒在做光线追踪时不如不重叠的效率高，比如下图中的右边划分效果就比左边的好。



找中间值的方法是利用了 std::partition() 方法，具体步骤可以参考 PBRT 源码。

6 4.2　等量法

将当前集划分为两个相等数量的子集，使得 n 个基元的前半部分是沿所选轴具有最小质心坐标值的 n/2 个，后半部分是具有最大质心坐标值的 n/2 个。划分方法通过 std::nth_element() 来进行（该方法不同于 std::sort，而且复杂度是 $O(nlogn)$）。

6 4.3　表面积启发式 SAH 法

该方法是我要实现在自己系统里的方法，因为它的求交速度很快，虽然比上面的两种方法难很多，但是值得我们去学习和移植。

相比于上面的两种方法，SAH 考虑“划分为哪两部分可以得到更好的 BVH 树”以及“使用哪个值来划分可以得到更好的 BVH 树”。SAH 模型估计执行射线相交测试的计算开销，包括遍历树的节点所花费的时间，以及为划分的基元进行射线-基元相交测试所花费的时间。

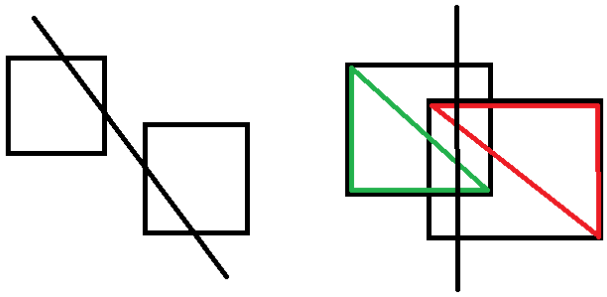
在构建自适应加速结构（原始细分或空间细分）的任何时候，我们都可以为当前区域和几何体创建一个叶节点。在这种情况下，通过该区域的任何光线都将针对所有重叠的基元进行测试（注意这里的基元既可以表示为包围盒，也可能是叶节点），当有 N 个基元时，遍历产生开销：

$$\sum_{i=1}^{N} t_{isect}(i) \tag{六.1}$$

当把该区间分割成两个区域时，遍历的开销就变为了：

$$c(A,B) = t_{trav} + p_A \sum_{i=1}^{N_A} t_{isect}(a_i) + p_B \sum_{i=1}^{N_B} t_{isect}(b_i) \tag{六.2}$$

t_{trav} 表示的是求交该内部节点然后选择需要求交的子节点耗费的时间。N_A 和 N_B 是子节点中基元相互重叠的区域数。如下图，Ray 从上到下打过来，左边当 Ray 与上面的包围盒中的 primitives 相交后，就不用再计算下面的求交了；但是右图当两个包围盒重叠时，里面的基元就都得计算，虽然 Ray 先与上面的包围盒相交，但是最终相交的 primitive 是下面的包围盒中的红色三角形。



我们假设求交的方法对所有基元耗费的时间都是一样的（事实证明这种假设对性能影响并不是很大，毕竟场景中的三角面片一般都是聚集在每个模型上，因此很可能一个包围盒内部都是三角形基元），我们也可以对每个求交方法计算其消耗的时钟周期。

p_A 和 p_B 的计算采用几何概率，对于包含在凸面体 B 中的凸面体 A，设 A 的表面积为 s_A，B 的表面积为 s_B，我们认为与 B 相交的 Ray 与 A 相交的概率为：

$$p(A|B) = \frac{s_A}{s_B} \tag{六.3}$$

所以对于包围盒 C 里面包含包围盒 A 和包围盒 B，与 C 相交的 Ray 与 A 和 B 相交的概率分别为 $\frac{s_A}{s_C}$ 和 $\frac{s_B}{s_C}$

当选用 SAH 方法后，通过考虑多个候选分区，找到一个沿所选轴的分区，该分区给出了最小的 SAH 代价估计。（这是默认的 SplitMethod，它为渲染创建了最有效的树。）但是，一旦它细化到一小批基元，实现就会切换到划分成大小相等的子集，因为此时应用 SAH 增加的计算成本是不值得的。

```
1    //如果该区间基元数小于某个值（PBRT书里是4，源码中设置的是2），就划分为等
         份
2    if(nPrimitives <= 4){
3        mid = (start + end) / 2;
4        //使用std::nth_element方法划分到两个子区间
5        ......
6    }else{
7        //启动SAH划分
8        ......
9    }
```

SAH 划分分为五步：

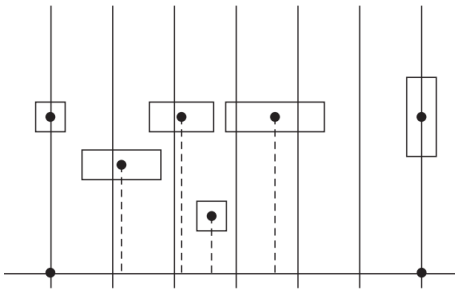
```
1    //创建buckets数组
2    ......
```

```
    //初始化buckets数组
    ......
    //计算根据每个bucket的分割成本（即假如有12个buckets，我们只需要计算11个
        分割位置）
    ......
    //找到最小化SAH量的bucket来分割
    ......
    //要么创建叶节点，要么进行区域划分（使用std::partition()方法）
    ......
```

首先介绍一个新的结构体：

```
    struct BucketInfo {
        int count = 0;
        Bounds3f bounds;
    };
```

这里的 SAH 实现不是穷尽地考虑沿轴的所有可能的划分（一共有 2n 种可能，n 表示该区间基元数，见 [1] 的 4.3.1 和 4.3.2 节）即计算每个分区的 SAH 以选择最佳分区，而是将沿轴的范围划分为具有相等范围的少量 buckets。然后它只考虑 buckets 包围盒处的分区。这种方法比考虑所有分区更有效率，而通常仍然产生几乎同样效果的分区。如下图所示：



基元包围盒质心的投影范围投影到选定的分割轴上。每个基元根据其包围盒的质心沿轴放置在 buckets 中。然后，该实现估计沿着每个 buckets 边界（实线）沿平面分割原语的成本；选择每个“surface area heuristic”的最小成本。

```
    //源码中设定了12个buckets
    int nBuckets = 12;
    BucketInfo buckets[nBuckets];
```

对于当前区间的每个基元，确定其质心在哪个 buckets 内，并更新 buckets 的边界包围盒：

```
    for (int i = start; i < end; ++i) {
        int b = nBuckets * centroidBounds.Offset(primitiveInfo[i].centroid)[
            dim];
        if (b == nBuckets) b = nBuckets - 1;
        CHECK_GE(b, 0); //检查b是否 >= 0
        CHECK_LT(b, nBuckets); //检查b是否 < nBuckets
        buckets[b].count++;
        buckets[b].bounds =
            Union(buckets[b].bounds, primitiveInfo[i].bounds);
    }
```

接下来，我们希望使用 SAH 来估计在每个 buckets 边界处拆分的成本。下面的代码片段在所有 bucket 上循环，并初始化 cost[i] 数组以存储估计的 SAH 开销，以便在第 i 个 bucket 之后进行拆分。（它不考虑在最后一个 bucket 之后进行分割，因为相当于不分割）。

接下来说点无关紧要的东西（因为具体实现的时候确实用不到）：我们将 Ray 与基元求交的 cost 设置为 1，然后将估计的遍历 cost 设置为 1/8。（其中一个总是可以设置为 1，因为它们之间是相对的，而不是绝对的，确定估计的遍历和求交 cost 的大小）。虽然节点遍历和 Ray-包围盒相交的绝对计算量仅略小于射线与形状相交所需的计算量，但 pbrt 中的射线-基元相交测试要经过两个虚函数调用增加了大量开销，因此我们估计它们的开销是 Ray-包围盒求交的八倍（此为经验值）。该计算在每个 buckets 里有 $O(n^2)$ 的复杂度，可能有更好的优化方法，只是因为每个 buckets 里的基元数量可能并不多，因此这点开销可以接受。

```
1     //计算在每个buckets后面分割的损耗
2     Float cost[nBuckets - 1];
3     for (int i = 0; i < nBuckets - 1; ++i) {
4         Bounds3f b0, b1;
5         int count0 = 0, count1 = 0;
6         for (int j = 0; j <= i; ++j) {
7             b0 = Union(b0, buckets[j].bounds);
8             count0 += buckets[j].count;
9         }
10        for (int j = i + 1; j < nBuckets; ++j) {
11            b1 = Union(b1, buckets[j].bounds);
12            count1 += buckets[j].count;
13        }
14        //这里提一句，其实代码里使用1/8还是其他值都是可以的，因为它只用了一
              次。PBRT源码里写的是1，不过其实无所谓，如果不是一定要弄懂的话大家
              不用深究。
15        cost[i] = .125 +
16                  (count0 * b0.SurfaceArea() +
17                   count1 * b1.SurfaceArea()) /
18                      bounds.SurfaceArea();
19    }
20    //找到最小化SAH量的bucket来分割
21    Float minCost = cost[0];
22    int minCostSplitBucket = 0;
23    for (int i = 1; i < nBuckets - 1; ++i) {
24        if (cost[i] < minCost) {
25            minCost = cost[i];
26            minCostSplitBucket = i;
27        }
28    }
```

最后，就通过 std::partition() 函数来根据选择的位置进行划分了。PBRT 里 partition 函数传入的判断函数同样是使用 Lambda 表达式实现的。

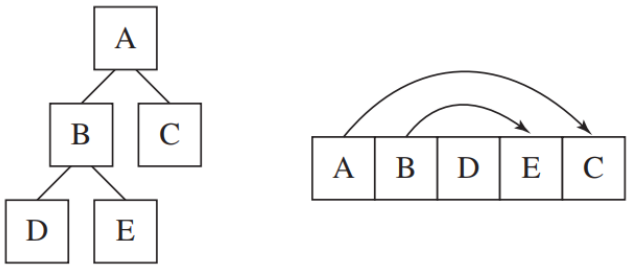
6 4.4 线性绑定体层次盒

因为该方法实现起来比较麻烦，鉴于 PBRT 书已经描述的非常详细了，想研究的同学可以自己翻阅 PBRT 书。这里我不再详细介绍这种方法。想尝试使用多个线程来加速 BVH 书构建的同学也可以研究一下（P.S. 目前基于 GPU 并行构建 kdTree 和 BVH 树的方法已经有不少了，大家可以去网上找相关资料

和论文）。

6 5 构建紧凑的 BVH 树

一旦构建了 BVH 树，最后一步就是将其转换为紧凑的表示，这样做可以提高缓存、内存，从而提高系统的整体性能。注意因为我们的树并不是完全二叉树，所以需要记录每个内部节点的子节点的索引，当然只需要记录第二个子节点的索引位置：



这是因为除了叶节点没有子节点，其他任意的内部节点的左子节点在数组上索引都在当前节点的右边+1 位置，例如 A 索引为 0，左子节点 B 的索引为 1。

PBRT 把线性 BVH 树存储在结构 LinearBVHNode 中：

```
struct LinearBVHNode {
    Bounds3f bounds;
    union {
        int primitivesOffset;    // leaf
        int secondChildOffset;   // interior
    };
    uint16_t nPrimitives;  // 如果为0说明是内部节点，大于0则是叶节点
    uint8_t axis;          // interior node: xyz
    uint8_t pad[1];        // 填充结构体，使字节变为32的倍数，访问会更快且稳
       定。
};
```

flattenBVHTree() 方法将 BVH 树存储在紧凑表示的结构中，以深度优先遍历的方法，并以线性顺序将节点存储在内存中：

```
    nodes = AllocAligned<LinearBVHNode>(totalNodes);
    int offset = 0;
    flattenBVHTree(root, &offset);
    //比较totalNodes是否等于offset
    CHECK_EQ(totalNodes, offset);
```

nodes 是 BVHAccel 类的成员变量，offset 记录最终一共有多少个节点。深度优先遍历的方法比较简单，这里不再赘述，您可以直接看 PBRT 源码。

6 6 BVH 树遍历

BVH 树的遍历非常简单，使用 BVHAccel::Intersect() 方法，得到是否与树中的基元相交的信息，该信息保存在 SurfaceInteraction 对象中：

```
    bool BVHAccel::Intersect(const Ray &ray, SurfaceInteraction *isect)
       const {
        if (!nodes) return false;
        bool hit = false;
```

```
4         //之前说过，PBRT采用预先计算的Ray的倒数和方向分量是否为负来做包围盒
              快速求交计算
5         Vector3f invDir(1 / ray.d.x, 1 / ray.d.y, 1 / ray.d.z);
6         int dirIsNeg[3] = {invDir.x < 0, invDir.y < 0, invDir.z < 0};
7         //遍历BVH树，如果与基元相交则找到最近的基元交点。结束后返回hit
8         ......
9     }
```

遍历 BVH 树的方法是一个循环，currentNodeIndex 记录当前访问的节点；nodesToVisit[] 是一个数组，记录了需要去被访问的节点；toVisitOffset 表示数组中下一个需要被访问的节点位置。

```
1     int toVisitOffset = 0, currentNodeIndex = 0;
2     int nodesToVisit[64];
3     //开始循环
4     while (true) {
5         const LinearBVHNode *node = &nodes[currentNodeIndex];
6         ......
7     }
8     //循环结束后
9     return hit;
```

在每个节点上，我们检查光线是否与节点的包围盒相交。如果相交，我们访问该节点，如果它是叶节点，则测试其与 primitives 基元的是否相交，如果是内部节点，则处理其子节点。如果没有交点，则从 nodesToVisit[] 相当于一个栈，检索下一个要访问的节点的偏移量（如果堆栈为空，则遍历完成），伪代码表示如下:

```
1     if (node->bounds.IntersectP(ray, invDir, dirIsNeg)) {
2         if (node->nPrimitives > 0) {
3             //测试Ray与基元相交
4             ......
5         } else {
6             //把BVH的右节点（在LinearBVHNode数组中索引更大的子节点）入
                  nodesToVisit[]栈
7             ......
8         }
9     } else {
10        if (toVisitOffset == 0) break;
11        currentNodeIndex = nodesToVisit[--toVisitOffset];
12    }
```

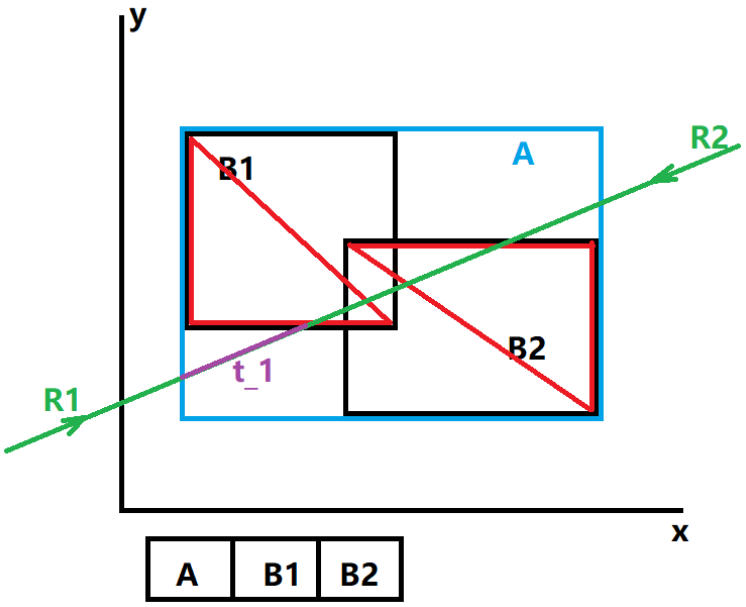
叶节点需要测试 Ray 与基元相交，如果有基元相交了，则 tMax 会被更新为相交距离，因为 Ray 的 t 值需要小于 tMax，因此这样会找到最近的交点。

```
1     //测试Ray与基元相交
2     for (int i = 0; i < node->nPrimitives; ++i)
3         if (primitives[node->primitivesOffset + i]->Intersect(ray, isect))
4             hit = true;
5     if (toVisitOffset == 0) break;
6     currentNodeIndex = nodesToVisit[--toVisitOffset];
```

对于光线击中的内部节点，有必要访问其两个子节点。希望在访问后面的子集之前访问光线经过的第一个子集，以防光线与第一个子集中的基元相交，从而可以更新光线的 tMax 值，降低 Ray 的 t 范围，从而降低其相交的节点包围盒的数量。一个有效的执行从前到后遍历的方法是使用 Ray 的方向向量的标志 dirIsNeg，用的标志轴是划分子区间的轴。如果标志是负的，就先访问第二个子节点，否则就先访问第一个子节点：

```
1    if (dirIsNeg[node->axis]) {
2        nodesToVisit[toVisitOffset++] = currentNodeIndex + 1;
3        currentNodeIndex = node->secondChildOffset;
4    } else {
5        nodesToVisit[toVisitOffset++] = node->secondChildOffset;
6        currentNodeIndex = currentNodeIndex + 1;
7    }
```

我们画个示意图来表示一下上述过程：



如图，蓝色 A 表示当前包围盒，黑色 B1 和 B2 表示两个子包围盒，绿线表示 Ray。在 LinearBVHNode 数组中,A 的索引为 0，B1 索引为 1，B2 索引为 2。假如此时两个子包围盒是按照 x 轴划分的，当光线为 R1 方向，则令 R1 先与 B1 计算求交，B2 入栈。这样，当 R1 与 B2 求交计算时，因为 t 的范围已经限定在了紫色区间 t_1 内，因此会判定与包围盒 B2 不相交，省去了访问包围盒 B2。当光线为 R2 方向时，则令 R1 先与 B2 求交，B1 入栈。

除了 BVHAccel::Intersect() 方法，还有 BVHAccel::IntersectP() 方法，该方法最后与基元中的 Shape 对象求交时调用 Shape 的 IntersectP() 函数，仅仅判断是否与基元相交，而不考虑其他内容。

6 7 小结

当前，我们已经学完了 PBRT 的形状类和加速器结构的各个细节，下一章我们会将整个光线求交的过程连成一个整体，然后动手编程实现到自己的引擎上。

注意在编译运行时，如果出现一些内存报错的原因，很可能是因为编译器进行了编译优化，需要在编译选项中取消优化。

七　代码逻辑与构建

到现在为止，相信您已经对 PBRT 的形状类、加速器结构等有了一定的认识。但是仅仅看懂别人写的代码，并不能说明任何问题，只有当您能够从头到尾实现出来的时候，您才有资格说自己已经真正的掌握了 PBRT。

按照前几章的内容，我们已经在自己的引擎里移植了 PBRT 的基本工具类，包括点、向量、法向量和包围盒，还有矩阵和变换等。我们的原始引擎的构建来自于 [2][3][4]，现在我们的目标就是构建出基于 PBRT 的引擎。

7 1　Integrator 与加速结构、形状类的程序逻辑

回顾《文件加载与设定》，程序启动渲染是在 pbrtWorldEnd() 函数中：

```
1    //如果场景和积分器顺利构建，就开始渲染
2    if (scene && integrator) integrator->Render(*scene);
```

scene 是场景类，类里面有四个成员：

```
1    //公共成员
2    std::vector<std::shared_ptr<Light>> lights;
3    std::vector<std::shared_ptr<Light>> infiniteLights;
4    //私有成员
5    //包含primitive及其加速结构的成员
6    std::shared_ptr<Primitive> aggregate;
7    //aggregate的包围盒（全部场景）
8    Bounds3f worldBound;
```

在积分器渲染时，SurfaceInteraction 对象用来保存交点信息，计算光学特性时会调用函数 SurfaceInteraction::ComputeScatteringFunctions，而该函数又调用的是 primitive->ComputeScatteringFunctions，它又调用的是 primitive 对象里的绑定的材料 material->ComputeScatteringFunctions。

我们暂时还没有光源类，但是我们知道，发光的 Shape 对象和不发光 Shape 对象都会被存到 primitives 基元里，然后构建到 BVH 树中。

7 2　渲染器基类的构建

为了方便叙述，我们可以给自己移植的引擎取一个名字，本文移植的引擎就叫做 FeimosRender 了。我们为整个工程构建名称空间：Feimos。下面所有的内容都是在名称空间里完成的，为了简洁不再附带名称空间表示。

我们将原工程里的 RayTracer 目录下的内容全部删除（您其实只需要保留一个显示图像的程序和之前移植的矩阵向量工具即可）。我们从零开始构建形状和加速器结构。

建立一个包含所有类名和基本工具的文件，例如 FeimosRender.h，里面对各种工具类以及其他基类、派生类都进行声明。

```
1  template <typename T>
2  class Vector2;
3  template <typename T>
4  class Vector3;
5  typedef Vector2<float> Vector2f;
6  typedef Vector2<int> Vector2i;
7  typedef Vector3<float> Vector3f;
8  typedef Vector3<int> Vector3i;
```

我们本节构建 PBRT 基本类型到自己的项目里。

将 Ray 类实现到 geometry.hpp 头文件里（PBRT 就是这么做的，把 Ray 当成一个基本类型）。

```
class Ray {
public:
    // Ray Public Methods
    Ray() : tMax(Infinity), time(0.f) {}
    Ray(const Point3f &o, const Vector3f &d, float tMax = Infinity, float
        time = 0.f) : o(o), d(d), tMax(tMax), time(time) {}
    Point3f operator()(float t) const { return o + d * t; }
    bool HasNaNs() const { return (o.HasNaNs() || d.HasNaNs() || isNaN(tMax)
        ); }
    // Ray Public Data
    Point3f o;
    Vector3f d;
    mutable float tMax;
    float time;
};
//我们暂时去掉了所有与误差界定有关的内容
inline Ray Transform::operator()(const Ray &r) const {
    Point3f o = (*this)(r.o);
    Vector3f d = (*this)(r.d);
    float tMax = r.tMax;
    return Ray(o, d, tMax, r.time);
}
```

之后，包围盒的求交函数我们现在补充上，直接 copy 过去就好了。注意我们不区分不同精度的浮点数，以后都把 Float 改成 float（在 PBRT 的 CMake 中，默认 Float 就是 float）。另外有 gamma 函数的代码行都可以去掉，因为误差校验我们暂时还没有涉及。

建立 interaction.h 和 interaction.cpp 文件，用来保存求交信息。因为我们暂时没有用于体渲染的媒介，错误绑定 pError 也没有研究，所以都不添加进该结构中。我们当前应用的该结构如下：

```
struct Interaction {
    // Interaction Public Methods
    Interaction() : time(0) {}
    Interaction(const Point3f &p, const Normal3f &n, const Vector3f &pError,
        const Vector3f &wo, float time)
        : p(p),
        time(time),
        wo(Normalize(wo)),
        n(n) {}
    // Interaction Public Data
    Point3f p;
    float time;
    Vector3f wo;
    Normal3f n;
};
```

建立基元头文件 primitive.h 和 primitive.cpp，用于定义基元类。我们的目标是先能使用加速结构等

进行渲染，所以暂时完全不需要材料、光源等的设置。

```
class Primitive {
public:
    // Primitive Interface
    virtual ~Primitive();
    virtual Bounds3f WorldBound() const = 0;
    virtual bool Intersect(const Ray &r, SurfaceInteraction *) const = 0;
    virtual bool IntersectP(const Ray &r) const = 0;
    virtual void ComputeScatteringFunctions() const = 0;
};
//派生类
class GeometricPrimitive : public Primitive {
public:
    // GeometricPrimitive Public Methods
    virtual Bounds3f WorldBound() const;
    virtual bool Intersect(const Ray &r, SurfaceInteraction *isect) const;
    virtual bool IntersectP(const Ray &r) const;
    GeometricPrimitive(const std::shared_ptr<Shape> &shape);
    void ComputeScatteringFunctions() const;
private:
    // GeometricPrimitive Private Data
    std::shared_ptr<Shape> shape;
};
```

SurfaceInteraction 类派生自 Interaction：

```
class SurfaceInteraction : public Interaction {
    public:
        // SurfaceInteraction Public Methods
        SurfaceInteraction() {}
        void ComputeScatteringFunctions();
        const Shape *shape = nullptr;
        const Primitive *primitive = nullptr;
};
```

Shape 类的定义如下，我们仅仅保留自己目前需要的部分，虽然 testAlphaTexture 用不到，但是参数已经被默认初始化了，所以带着也无妨。

```
class Shape {
public:
    // Shape Interface
    Shape(const Transform *ObjectToWorld, const Transform *WorldToObject,
        bool reverseOrientation);
    virtual ~Shape();
    virtual Bounds3f ObjectBound() const = 0;
    virtual Bounds3f WorldBound() const;
    virtual bool Intersect(const Ray &ray, float *tHit, SurfaceInteraction *
        isect, bool testAlphaTexture = true) const = 0;
```

```
    virtual bool IntersectP(const Ray &ray, bool testAlphaTexture = true)
        const {
        return Intersect(ray, nullptr, nullptr, testAlphaTexture);
    }
    virtual float Area() const = 0;
    const Transform *ObjectToWorld, *WorldToObject;
    //表示当前法向量的方向是否需要翻转
    const bool reverseOrientation;
};
```

在 primitive.h 头文件加入 Aggregate 类：

```
    class Aggregate : public Primitive {
        public:
            // Aggregate Public Methods
            void ComputeScatteringFunctions() const {}
        };
```

建立 BVHAccel.cpp 和 BVHAccel.h 头文件，因为我们目前还没有内存管理类 MemoryArena，所以将参数列表的该参数删掉。

```
class BVHAccel : public Aggregate {
public:
    // BVHAccel Public Types
    enum class SplitMethod { SAH, HLBVH, Middle, EqualCounts };
    // BVHAccel Public Methods
    BVHAccel(std::vector<std::shared_ptr<Primitive>> p, int maxPrimsInNode =
        1, SplitMethod splitMethod = SplitMethod::SAH);
    Bounds3f WorldBound() const;
    ~BVHAccel();
    bool Intersect(const Ray &ray, SurfaceInteraction *isect) const;
    bool IntersectP(const Ray &ray) const;
private:
    // BVHAccel Private Methods
    BVHBuildNode *recursiveBuild(std::vector<BVHPrimitiveInfo> &
        primitiveInfo, int start, int end, int *totalNodes, std::vector<std::
        shared_ptr<Primitive>> &orderedPrims);
    int flattenBVHTree(BVHBuildNode *node, int *offset);
    // BVHAccel Private Data
    const int maxPrimsInNode;
    const SplitMethod splitMethod;
    std::vector<std::shared_ptr<Primitive>> primitives;
    LinearBVHNode *nodes = nullptr;
};
```

目前为止，我们仅仅移植了类，里面的方法基本上没有移植。移植完类以后，编译通过，本节的目标就完成了。

7 3 Shape 类和 Sphere 类的填充和实现

我们先在 Shape.cpp 文件里补充完 Shape 的成员函数：

```
1   static long long nShapesCreated = 0;
2   Shape::Shape(const Transform *ObjectToWorld, const Transform *
        WorldToObject, bool reverseOrientation)
3       : ObjectToWorld(ObjectToWorld),
4         WorldToObject(WorldToObject),
5         reverseOrientation(reverseOrientation) {
6       ++nShapesCreated;
7   }
8   Shape::~Shape() {}
9   Bounds3f Shape::WorldBound() const { return (*ObjectToWorld)(ObjectBound
        ()); }
```

移植球体。我们不需要考虑半球或者不完整球，因此球体类里面删除了不少变量：

```
1  // Sphere Declarations
2  class Sphere : public Shape {
3  public:
4      // Sphere Public Methods
5      Sphere(const Transform *ObjectToWorld, const Transform *WorldToObject,
           bool reverseOrientation, float radius) : Shape(ObjectToWorld,
           WorldToObject, reverseOrientation), radius(radius) {}
6      Bounds3f ObjectBound() const;
7      bool Intersect(const Ray &ray, float *tHit, SurfaceInteraction *isect,
           bool testAlphaTexture) const;
8      bool IntersectP(const Ray &ray, bool testAlphaTexture) const;
9      float Area() const;
10 private:
11     // Sphere Private Data
12     const float radius;
13 };
```

求交程序我们使用 [2] 的方法，暂时我们只判断是否相交：

```
1  bool Sphere::Intersect(const Ray &r, float *tHit, SurfaceInteraction *isect,
       bool testAlphaTexture) const {
2      Point3f pHit;
3      // Transform _Ray_ to object space
4      Ray ray = (*WorldToObject)(r);
5      Vector3f oc = ray.o - Point3f(0.0f,0.0f,0.0f);
6      float a = Dot(ray.d, ray.d);
7      float b = 2.0 * Dot(oc, ray.d);
8      float c = Dot(oc, oc) - radius*radius;
9      float discriminant = b*b - 4 * a*c;
10         return (discriminant > 0);
11 }
12 bool Sphere::IntersectP(const Ray &r, bool testAlphaTexture) const {
13         //和Intersect函数一样
```

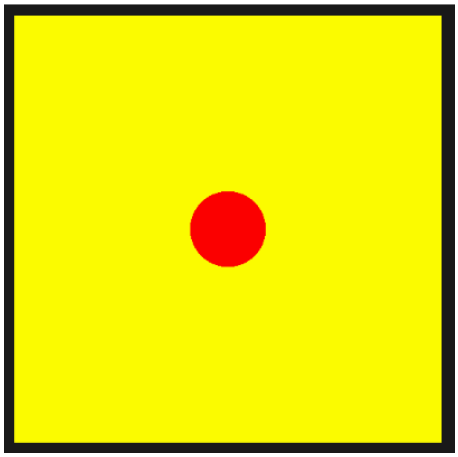
```
}
```

目前基本功能都填充完了，我们简单测试一下。首先我们通过球体测试一下形状类是否完全实现了：

```
//都设置为单位矩阵的变换，即球的中心在世界坐标原点，暂时不考虑世界坐标–相机坐标变换
Transform sphereT_Object2World, sphereT_World2Object;
Shape* s = new Sphere(&sphereT_Object2World, &sphereT_World2Object, false, 1.0);

void renderFrame() {
    Vector3f lower_left_corner(-2.0, -2.0, -1.0);
    Vector3f horizontal(4.0, 0.0, 0.0);
    Vector3f vertical(0.0, 4.0, 0.0);
    Point3f origin(0.0, 0.0, -3.0);
    for (int i = 0; i < ThreadNum; i++) {
        for (int j = 0; j < ThreadNum; j++) {
            float u = float(i + 0.5) / float(ThreadNum);//random()
            float v = float(j + 0.5) / float(ThreadNum);//0.5
            Ray r(origin, (lower_left_corner + u*horizontal + v*vertical) - Vector3f(origin));
            float tHit;
            SurfaceInteraction* isect = nullptr;
            Vector3f colObj(1.0, 1.0, 0.0);
            if (s->Intersect(r, &tHit, isect)) {
                colObj = Vector3f(1.0, 0.0, 0.0);
            }
            //将结果保存在你的文件里或者GUI界面显示
            ......
        }
    }
}
```

得到如下结果，说明我们的形状类的接口已经成功实现。



7 4　坐标变换的填充和实现

虽然我们已经移植了 LookAt 函数，也可以自己去定义坐标变换了：

```
    Point3f eye(0.0, 0.0, 3.0), look(0.0, 0.0, 1.0);
    Vector3f up(0.0, 1.0, 0.0);
    Transform lookat = LookAt(eye, look, up);
```

但因为两种相机，包括透视相机和正交投影相机都还没有介绍，因此我们暂时不使用 LookAt 函数。等后面的系列书时我们会重新介绍与坐标系有关的内容。

因此，当前的相机位置为 Point3f(0.0, 0.0, -3.0)，看向 z 轴的正方向。（此时我们并不需要区分左手坐标系和右手坐标系）

7 5 三角面片的构建和实现

虽然我们并没有与纹理坐标之类相关的内容，但毕竟纹理坐标就是 Point2f 定义的变量，带着也无妨。但是 Texture 类鉴于我们还没有（我们不需要使用 [2][3][4] 中的纹理方法），所以去掉与 Texture 相关的内容。其中 Mesh 类的定义如下：

```
struct TriangleMesh {
    // TriangleMesh Public Methods
    TriangleMesh(const Transform &ObjectToWorld, int nTriangles,
        const int *vertexIndices, int nVertices, const Point3f *P,
        const Vector3f *S, const Normal3f *N, const Point2f *uv,
        const int *faceIndices);
    // TriangleMesh Data
    const int nTriangles, nVertices;
    std::vector<int> vertexIndices;
    std::unique_ptr<Point3f[]> p;
    std::unique_ptr<Normal3f[]> n;
    std::unique_ptr<Vector3f[]> s;
    std::unique_ptr<Point2f[]> uv;
    std::vector<int> faceIndices;
};
static long long triMeshBytes = 0;
```

三角形类的定义如下：

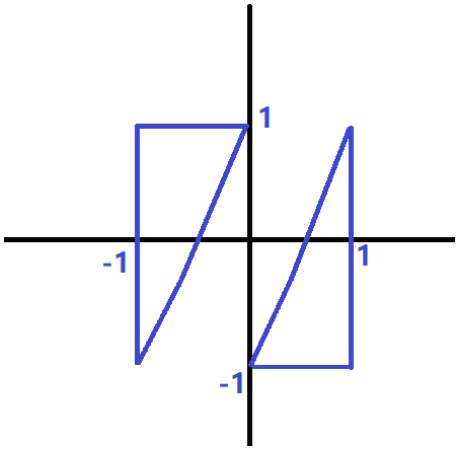
```
class Triangle : public Shape {
public:
    // Triangle Public Methods
    Triangle(const Transform *ObjectToWorld, const Transform *WorldToObject,
        bool reverseOrientation, const std::shared_ptr<TriangleMesh> &mesh,
        int triNumber)
        : Shape(ObjectToWorld, WorldToObject, reverseOrientation), mesh(mesh
            ) {
        v = &mesh->vertexIndices[3 * triNumber];
        triMeshBytes += sizeof(*this);
        faceIndex = mesh->faceIndices.size() ? mesh->faceIndices[triNumber]
            : 0;
    }
    Bounds3f ObjectBound() const;
    Bounds3f WorldBound() const;
```

```
    bool Intersect(const Ray &ray, float *tHit, SurfaceInteraction *isect,
        bool testAlphaTexture = true) const;
    bool IntersectP(const Ray &ray, bool testAlphaTexture = true) const;
    float Area() const;
private:
    // Triangle Private Methods
    //因为太长了，就省略一下
    void GetUVs(Point2f uv[3]) const { ..... }
    // Triangle Private Data
    std::shared_ptr<TriangleMesh> mesh;
    const int *v;
    int faceIndex;
};
```

补充 Triangle.cpp 文件中的成员函数：

```
    //一些全局变量，用于记录信息
    static long long nTris = 0;
    static long long nMeshes = 0;
    static long long nHits = 0;
    static long long nTests = 0;
    //TriangleMesh构造函数参数去掉纹理类即可
    .....
    Bounds3f Triangle::ObjectBound() const; //直接拷贝
    Bounds3f Triangle::WorldBound() const; //直接拷贝
    float Triangle::Area() const; //直接拷贝
    //Intersect函数截掉后面的部分，我们当前只需要计算求交即可
    bool Triangle::Intersect(const Ray &ray, float *tHit, SurfaceInteraction
        *isect,
        bool testAlphaTexture) const;
    //该函数暂时用不到
    bool Triangle::IntersectP(const Ray &ray, bool testAlphaTexture) const {
            //与Intersect一样，只是不需要求交的其他信息了
        .....
    }
```

移植完了以后我们测试一下。我们设置这样的三角形，它的顶点的 x 和 y 坐标如图，z 轴坐标都是 0。



故第一个三角形顶点坐标为 (-1.0,1.0,0.0),(-1.0,-1.0,0.0),(0.0,1.0,0.0)，第二个三角形顶点坐标为 (1.0,1.0,0.0),(1.0,-

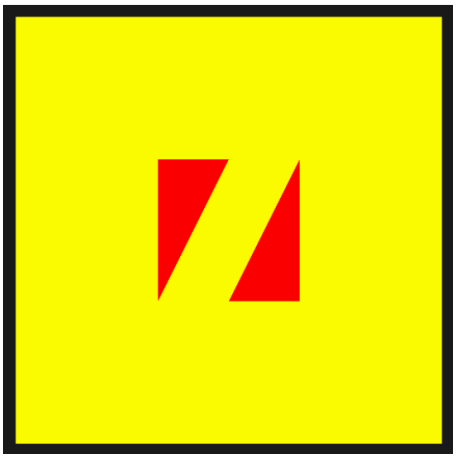
1.0,0.0),(0.0,-1.0,0.0)。法向量、纹理坐标等都不考虑。初始化程序如下：

```
Transform tri_Object2World, tri_World2Object;
int nTriangles = 2;
int vertexIndices[6] = { 0,1,2,3,4,5 };
int nVertices = 6;
Point3f P[6] = {
    Point3f(-1.0,1.0,0.0),Point3f(-1.0,-1.0,0.0),Point3f(0.0,1.0,0.0),
    Point3f(1.0,1.0,0.0),Point3f(1.0,-1.0,0.0),Point3f(0.0,-1.0,0.0)
};
//存入mesh里，该行代码可以从triangle.cpp中找到
std::shared_ptr<TriangleMesh> mesh = std::make_shared<TriangleMesh>(
   tri_Object2World, nTriangles, vertexIndices, nVertices, P, nullptr,
   nullptr, nullptr, nullptr);
std::vector<std::shared_ptr<Shape>> tris;
void sceneInit() {
    //从mesh转为Shape类型，该代码可以从triangle.cpp文件里找到
    tris.reserve(nTriangles);
    for (int i = 0; i < nTriangles; ++i)
        tris.push_back(std::make_shared<Triangle>(&tri_Object2World, &
           tri_World2Object,      false, mesh, i));
}
```

在渲染的 for 循环里我们如此做：

```
if (tris[0]->Intersect(r, &tHit, isect) || tris[1]->Intersect(r, &tHit,
   isect)) {
    colObj = Vector3f(1.0, 0.0, 0.0);
}
```

最终得到如下渲染结果，证明我们的代码移植成功了！



7 6 复杂的物体的渲染

为了对比加速器和不使用加速结构，我们这次渲染一个面片数比较多的物体。虽然我们目前没有文件加载器，但是构建一个简单的加载器还是很容易的！

我们可以去 [6] 网站上下载模型，或者网上去找各种模型。三维文件一般都是.obj 或者.ply 结尾的，解析这种文件格式的程序网上一搜一大堆，PBRT 源码中 plymesh.h 也有解析.ply 的文件。我们的模型文件都放在了网站上，而不是 Github 链接上。我们当前使用的 dragon.ply 文件我们只需要读取顶点位置和顶

点索引即可。您可以自行修改自己的渲染代码。

```
for (int count = 0; count < nTriangles; ++count) {
    if (tris[count]->Intersect(r, &tHit, isect)){
        colObj = Vector3f(1.0, 0.0, 0.0);
        break;
    }
}
```

我渲染下面这张图在我的台式机上用了两个多小时，总共需要计算 512*512*871414 次三角面片的求交运算：

因为它的坐标 y 值都比较大，所以渲染出来以后龙在靠上的位置。不过我实在懒得再渲染一帧了，就这样吧。至少说明了我们的引擎当前都是可以工作了，那么剩下的任务就是移植和测试加速器结构了。

7 7 加速器功能的实现

突然觉得有点激动，终于快到尾声了。形状类接口我们已经全都搞定，只剩下加速器了。

```
    //定义公共变量:
    static long long treeBytes = 0;
    static long long totalPrimitives = 0;
    static long long totalLeafNodes = 0;
    static long long interiorNodes = 0;
    static long long leafNodes = 0;
```

BVHPrimitiveInfo、BVHBuildNode、LinearBVHNode、BucketInfo 结构我们都是可以拿来直接用的，甚至不需要修改。

BVHAccel 的构造函数需要去掉与 MemoryArena 有关的内容，并且分配内存方面我们还得自己修改。

```
    //我们将:
    nodes = AllocAligned<LinearBVHNode>(totalNodes);
    //暂且改为:
    nodes = new LinearBVHNode[totalNodes];
```

BVHAccel::recursiveBuild 函数中：

```
    //我们将:
    BVHBuildNode *node = arena.Alloc<BVHBuildNode>();
    //改为:
    BVHBuildNode *node = new BVHBuildNode;
```

移植 BVHAccel::flattenBVHTree 函数，不需要任何修改。

然后移植上 BVHAccel::Intersect 和 BVHAccel::IntersectP 函数，里面需要改几个 Float 为 float。

7 8 Primitive 类的实现和移植

需要注意的是，虽然我们已经实现了 Shape:

```
std::vector<std::shared_ptr<Shape>> tris;
```

但是形状类不能直接绑定到 BVH 树里，需要构建成包含几何和材料纹理信息的 Primitive 类，更准确一点，GeometricPrimitive 类。

幸运的是，只需要简单的修改，就能移植到我们的系统上：

```
static long long primitiveMemory = 0;
Primitive::~Primitive() {}
// GeometricPrimitive Method Definitions
GeometricPrimitive::GeometricPrimitive(const std::shared_ptr<Shape> &
    shape)
    : shape(shape){
    primitiveMemory += sizeof(*this);
}
Bounds3f GeometricPrimitive::WorldBound() const { return shape->
    WorldBound(); }
bool GeometricPrimitive::IntersectP(const Ray &r) const {
    return shape->IntersectP(r);
}
bool GeometricPrimitive::Intersect(const Ray &r,
    SurfaceInteraction *isect) const {
    float tHit;
    if (!shape->Intersect(r, &tHit, isect)) return false;
    r.tMax = tHit;
    // Initialize _SurfaceInteraction::mediumInterface_ after _Shape_
    // intersection
    return true;
}
```

那么现在移植的工作就已经完全结束了，我们来写代码测试一下：

```
//初始化程序
std::vector<std::shared_ptr<Primitive>> prims;
Aggregate *agg;
    for (int i = 0; i < nTriangles; ++i)
            tris.push_back(std::make_shared<Triangle>(&tri_Object2World,
                &tri_World2Object, false, mesh, i));
    for (int i = 0; i < nTriangles; ++i)
            prims.push_back(std::make_shared<GeometricPrimitive>(tris[i
                ]));
    agg = new BVHAccel(prims, 1);

//渲染程序
Vector3f colObj(1.0, 1.0, 0.0);
```

```
12      if (agg->Intersect(r, isect)){
13          colObj = Vector3f(1.0, 0.0, 0.0);
14      }
```

得到和上面的图一样的结果，在我的台式机这次只用了 0.012 秒!!（注意我使用了 OpenMP 加速，需要在 Visual Studio 上开启 OpenMP 加速；VS 工程也设为了 Release 模式）超级快有木有，对比不使用加速器的渲染需要两个多小时，使用加速器简直快到逆天。

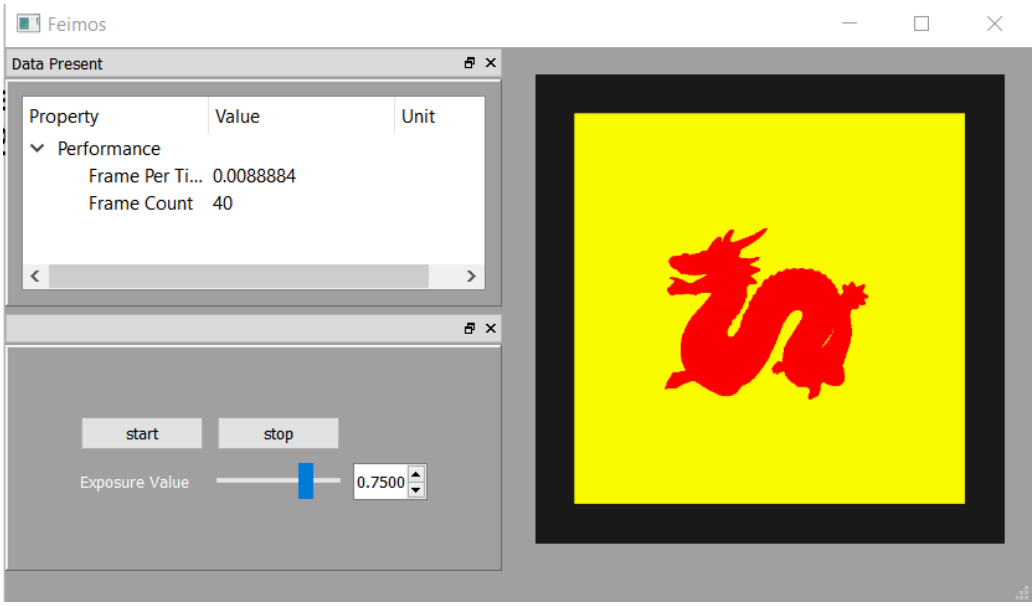
至此，PBRT 的形状类和包围盒加速机制我们就全都掌握和移植成功了。

八　本书结语

我将龙的坐标使用变换矩阵：

```
1       tri_Object2World = Translate(Vector3f(0.0,-2.5,0.0))*tri_Object2World;
```

向下移动了一定的距离，显示结果如下：



这样就好看多了。

本书从 1 月 28 日开始写，过年休息了一段时间，2 月 26 日写完，大概用了不到两周的时间。到目前为止，[2][3][4] 书中的代码已经基本上被删得一干二净了，我们的系统也升级为了 PBRT 系统，实现了 Shape 和包围盒加速，下一本书我计划写关于光线微分算法的内容，并讲解误差绑定和内存管理。

最后，谢谢大家阅读本书，如果您有问题和建议，欢迎在网站上留言。

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] Marschner S , Shirley P . Fundamentals of computer graphics. 4th edition.[J]. World Scientific Publishers Singapore, 2009, 9(1):29-51.

[6] URL:https://benedikt-bitterli.me/resources/