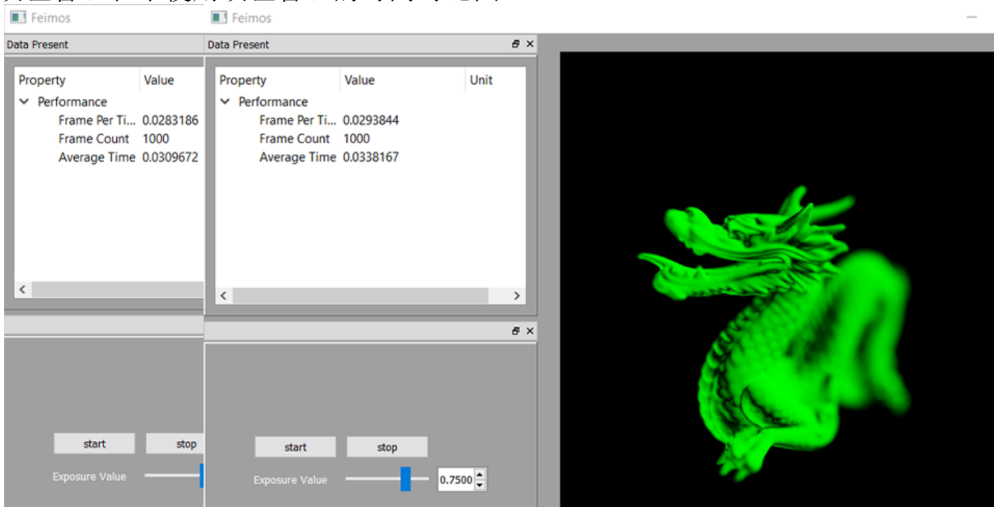
PBRT 系列 6-误差界定和内存管理

Dezeming Family

2021 年 3 月 1 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 的内存管理类和误差界定类，当在 PBRT 程序中遇到时知道它是在干什么就行了。如果想移植可以移植到自己的系统上，不想移植就算了。因为本文没有新渲染的东西，所以随便渲染一张使用误差管理和不使用误差管理的时间对比图：



本文于 2022 年 7 月 7 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

自从《形状与加速器》篇完结后，《颜色与光谱》和《相机系统》都是很简单的内容。本章主要是学习一下 PBRT 如何让系统更稳健，即让误差更可控。

本想将光线微分和误差界定放在一起讲，但好像内容过多。误差界定我不打算全部移植（我以前学习的时候只移植了 Ray 的产生、Ray 与包围盒求交、Ray 与三角形求交、Ray 的再产生中的误差界定，但我建议初学者不要移植，因为可能会让你的系统变得更乱（不就是有可能错过一些极端情况下的求交么，能忍则忍）），但是光线微分我是打算完全移植和应用的。

误差管理相对来说比较难，因为这里面涉及了不少计算机底层的知识。如果大家没有学习过《数值计算》（比如说我），那么学习的时候会非常吃力。因此我们并不要求完全学会误差界定的内容，因为它并不影响我们以后学习 PBRT 中最重要的内容——材质与路径追踪（误差管理只在基础的求交方面有应用），虽然在 Ray 递归中会不断界定更大的误差，但其实不考虑鲁棒性这些内容都是可以忽略的。我们只需要知道，为什么需要使用 gamma 函数，EFloat 类是干嘛用的，基本上就够了。不过为了全面我还是能多说一点就多说一点。

内存管理在 PBRT 中也是非常重要的一个模块，我们在前面的系列书中都是有 new Vector3f[n] 这种动态分配内存的方式，但是相比于 PBRT 的内存管理系统，它在很多方面都不完备，因此我们会介绍它，但我并不打算移植它，因为它带来的性能上的提升并不会特别显著，而且大家可以感受到，只要我们学会了 PBRT 主体，想扩展和加速其实并不难很容易。我们最主要的是需要理解里面用到的各种方法，这样当我们在 PBRT 程序里遇到的时候，我们就知道它分配了多少内存以及实际意义了。

这里必须要提一句使用内存管理类的好处：在于 BVH 树的管理。我们知道释放 BVH 树是一个递归的过程，比较耗费时间，而且容易出错；我之前写过一个动态场景，就是因为大型数据结构的内存释放的问题导致内存泄漏；而使用内存管理类则可以一次性将整个区域的内存都释放掉，完全解决了这个问题。我们因为精力问题，暂时先不考虑 BVH 树内存的释放，这也是我们本系列挖的一个坑。希望大家在研究完后面的系列时，能够有意识地补上这个坑——将内存管理类再移植上去，或者自己实现一个内存管理机制，或者直接使用智能指针来管理。

人生苦短，及时行乐，有的问题能跳过就及时跳过，多在有限的生命中做点更有意义的事情，岂不是更有趣。

本书的售价是 3 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 PBRT 的内存管理

本章我们主要学习一下 PBRT 的内存管理流程，我不打算移植，但是当我们阅读源码的时候，如果我们遇到了相关内容需要知道怎么去做。

关于内存管理优化的书和文章有很多，大家如果想自己做一个内存优化管理器，可以去自行搜寻相关文章。

1 1 MemoryArena 类

本节我们解决一个我们前面好几本书都遇到的类：MemoryArena 类，该类在 core 文件夹下。本章的大部分内容都来自于 PBRT 书 [1] 的附录 A.4。

MemoryArena（用于高性能临时内存分配）不适合多线程使用；这些类存储在调用其方法时修改的状态，与它们执行的计算量相比，互斥保护对其状态的修改的开销将是过大的。因此，我们可以认为，这个类的使用比较简单，毕竟没有多线程机制，省了很多事。

传统的观点认为，系统的内存分配例程（例如 malloc() 和 new()）速度很慢，对于频繁分配或释放的对象，使用自定义的分配内存方法可以提供一定的性能增益。然而，这种传统观点似乎是错误的。一些科学家都调查了实际应用程序中内存分配对性能的影响，并发现在执行时间和内存使用方面，专门定制的内存分配器几乎总是导致比优化的通用系统内存分配更差的性能。

一种已被证明在某些情况下有用的自定义分配技术是 arena-based 的分配，它允许用户从一个大的连续内存区域快速分配对象。在这个方案中，单个对象永远不会被显式释放；当所有分配对象的生存期结束时，整个内存区域都会被释放。这种类型的内存分配器自然适合 pbrt 中的许多对象。

arena-based 的分配有两个主要优点。首先，分配非常快，通常只需要一个指针增量。其次，由于分配的对象在内存中是连续的，因此它可以改进引用的局部性并减少缓存未命中。更通用的动态内存分配器通常会为它返回的每个块预先设置结构，这会对引用的局部性产生不利影响。pbrt 提供 MemoryArena 类来实现这种方法；它支持来自 arena 的可变大小的分配。MemoryRena 通过将指针分发到预先分配的块中，为大小可变的对象快速分配内存。它不支持释放单个内存块，只支持一次释放 arena 中的所有内存。因此，当需要快速完成大量分配并且所有分配的对象都具有相似的生命周期时，它非常有用。

MemoryArena 以 MemoryArena::blockSize 大小的块分配内存，其值由传递给构造函数的参数设置。如果没有向构造函数提供值，则使用默认值 256kB。

```
1    MemoryArena(size_t blockSize = 262144) : blockSize(blockSize) {}
```

该实现维护一个指向当前内存块 currentBlock 的指针，以及块中第一个空闲位置 currentPos 的偏移量。currentAllocSize 存储 currentBlock 分配的总大小；它通常具有值 blockSize，但在某些情况下更大：为了处理分配请求，分配例程首先对请求的内存量进行四舍五入，使其满足计算机的字对齐要求，然后检查当前块是否有足够的空间来处理请求，如有必要，分配一个新块。最后，它返回指针并更新当前块偏移量。我们使用 Alloc() 成员函数进行分配，该函数调用的 AllocAligned 函数会调用 windows 系统函数来执行分配。MemoryArena 有两个列表：usedBlocks 和 availableBlocks，分别记录使用中的内存和空闲可用的内存。

MemoryArena 还提供了根据类型和对象数量分配内存的 Alloc 函数：

```
1    //如果runConstructor为true，就为分配内存的每个对象调用其构造函数来初始化
         这片内存区
2    template <typename T>
3    T *Alloc(size_t n = 1, bool runConstructor = true);
```

使用完以后，使用 Reset 函数，将所有内存从 usedBlocks 列表移到 availableBlocks 列表。

它的使用方法我们也见过了，该类的对象作为一个传入参数，然后在函数中调用分配。

1 2 内存的释放

TriangleMesh 类的 faceIndices 和 vertexIndices 都是 vector 类型的数组，我们的基元存储在 std::vector<std::shared 里。vector 对象离开生存期以后会自动释放，但是如果里面是指针类型就不会自动释放了。所幸我们存储的数据是使用的智能指针，因此当 vector 释放时，指针指向的内容也会被释放掉。

BVHAccel 里存储了 LinearBVHNode *nodes。该指针存储的内容会在析构函数里进行释放：

```
1    BVHAccel::~BVHAccel() { FreeAligned(nodes); }
```

我们知道，AllocAligned 和 FreeAligned 是 memory.h 和 memory.cpp 里定义的两个函数，为了产生对齐的内存。我们是使用的 new 建立的内存，因此我们在析构函数里直接使用 free 或者 delete[] 就好了（在释放二叉树结构时，很可能会带来内存泄漏的问题，本文暂时不考虑这些问题）。

1 3 Blocked 2D 数组

在 C++ 中，2D 数组在内存中顺序排列，排列导致索引位置可能相离特别远，因此 BlockedArray 类诞生了，该类优化二维数组的存储。

它的输入也很简单，就是数组的长宽：

```
1    BlockedArray(int uRes, int vRes, const T *d = nullptr)
```

在图像中可能会用到，但鉴于我们没必要在这里优化太多，所以我们就不考虑使用它了（这里面涉及位操作和内存方面的知识，有点复杂），遇到的时候我们知道如何修改成自定义的二维数组就好了。

二　浮点数运算

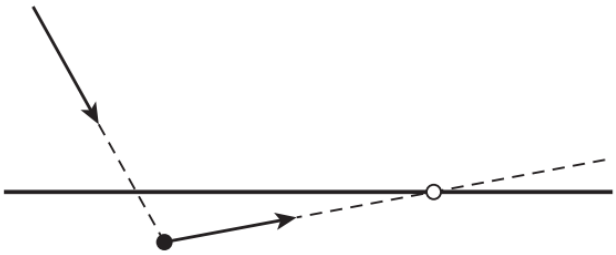
误差管理是一个并不复杂但也不简单的功能，尤其是在 PBRT 系统里。我们前面在学习和移植时，为了方便和简单，去除了与误差界定有关的代码，但是现在我们可以将它们移植回来了（或者不移植，但是要知道它是干嘛用的），因为我们只有两种物体，即球体和三角形，所以移植误差界定会相对容易很多。在计算机运算中，误差常是舍入误差（Rounding Error），因为计算机二进制表示小数一般不能很精确。本章有关内容在 PBRT 书 [1] 的 3.9 节讲述的比较全面，这里会按照书 [1] 上的方式进行讲解。

如果大家对计算误差有兴趣的话（估计是没有），可以参考一些《数值计算》方面的书。说实话我目前没有系统学习过数值计算，因此我的理解也比较片面。但我认为当前 PBRT 渲染系列里最重要的目标是材质和渲染器的学习，因此我也不会在误差界定上花费太多时间去介绍，我们只需要知道基本原理就好了。

在几何光线相交计算中，浮点舍入误差的影响一直是光线跟踪中的一个长期挑战：它们会导致在整个图像中出现小误差。PBRT 将重点放在这个问题上，并导出了这个误差的保守（但严格）界限，这使得 PBRT 的实现比以前的渲染系统对这个问题更加鲁棒。

误差在计算中会一步步积累，在 Ray 与物体表面求交中，如果计算的 t 值是负的，但是精确值是正值，就会错过相交。计算出的相交点有可能会偏离到物体表面的上面或者下面，当 Ray 重新被从表面发射出时，如果出发点在表面下，则又会与本表面相交。

在光线跟踪中解决此问题的典型做法是用固定的“ray epsilon”值偏移生成的光线，忽略沿光线 $p+td$ 的比某些 t_min 值更近的任何交点。如果生成的光线与曲面相当倾斜，则在距离光线原点相当远的地方可能会发生不正确的光线相交。不幸的是，较大的 tmin 值会导致光线原点与原始交点相对较远，这反过来又会导致丢失附近的有效交点，从而导致阴影和反射中的精细细节丢失，如下图。



本章将介绍浮点算法的基本思想，并描述分析浮点计算中错误的技术。然后，我们将把这些方法应用到本章前面介绍的 Ray-Shape 算法中，并演示如何计算有界误差的光线交点。这将允许我们保守地定位光线原点，这样就永远不会发现不正确的自交点，同时使光线原点非常接近实际交点，这样就可以将不正确的未命中最小化，同时也不需要额外的“ray epsilon”值。

2 1　浮点数运算

计算机只能表示有限实数。

一种这样的有限表示是定点表示法，定点数的优点就是可以精确表示想要表示的数值，不会像浮点数一样计算机内部无法精确的表示一些数值，比如要表示 [0,100) 之间的任何两位小数，定点数都能精确表示，并能使用整数的运算方法；缺点就是不适用于表示特别大或者特别小的数值，比如要表示 [0.00000000123,4560000000000) 之间的任何 12 位小数，那么就需要使用的字节高达 12 个 (算上符号位)。我们举个例子，比如我们要用 [0,65535] 范围的数表示 [0,256] 之间的小数，那么我们就对 [0,65535] 除以 256，得到 [0, 255+255/256]，也就是说这些小数之间的间隔是 1/256。

另一个表示法就是我们常用的浮点数，它们是基于用符号、有效位和指数来表示数字的：本质上，与科学记数法相同，但用固定数量的数字来表示有效位和指数。（在下文中，我们将专门假设基数为 2 位。）这种表示方式使得在使用固定存储量的情况下，可以表示和计算具有广泛数量级的数字。

IEEE 标准表示 32 位浮点数有 1bit 标志位，8bit 指数位，23bit 有效位。指数位 e_b 从-127 变化到 128，可以表示 2 的 e_b 次方。

当存储标准化浮点值时，有效位实际上有 24 位精度。因为都会表示成 $1.xxxxx$，前面的 1 就被省略

了（也叫有效位归一化）。标志位是 s，有效位是 m，指数位是 e，则计算得到的浮点数值是：

$$s * 1.m * 2^{e-127} \tag{二.1}$$

例如 6.5 会被表示为 $1.101_2 * 2^2$，这里的 1.101_2 表示以 2 为基底：$(1*2^0+1*2^{-1}+0*2^{-2}+1*2^{-3})*2^2 = 6.5$。因此在内存中表示为：

```
//符号位 指数位 有效位
0 10000001 10100000000000000000000 = 0x40d00000
```

这种表示法的一个含义是，两个相邻的二次幂之间的可表示浮点数之间的间距在整个范围内是一致的，$[2^e, 2^{e+1})$ 的间距（有效位中的表示）是 2^{e-23}（因为有效位是 23 位），对于 1 和 2 之间的浮点数，e=0，则 $2^{-23} * 2^0 \approx 1.19209.. * 10^{-7}$。这里的间隔随着浮点数更大，间隔也更大，浮点数较小，间隔也较小。我们知道有效位会被表示为 1.xxxxxx，所以浮点数不能精确表示 0：$s * 0.000..._2 * 2^{-127}$。绝对值最小的浮点数就是 $s * 1.000..._2 * 2^{-127}$ 了。

提供一些表示这些小值的功能可以避免将非常小的值舍入为零。注意此表示法同时存在“正值”和“负值”零值。这个细节对程序员来说是透明的。例如，该标准保证 $-0.0 == 0.0$ 的计算结果为 true，即使这两个值在内存中的表示形式不同。

最大指数 e=255 也保留作特殊处理。因此，可以表示的最大正则浮点值具有 e 254（或 eb 127）。

当 $e_b = 255$ 时，如果有效位均为零，则根据符号位，该值对应于正无穷大或负无穷大。例如，在执行浮点 1/0 之类的计算时会产生无穷大的值。无穷大的算术运算结果是无穷大。比较时，正无穷大比任何非无穷大的值都大，负无穷大也一样。

MaxFloat 和 Infinity 常量分别初始化为最大的可表示浮点值和“Infinity”浮点值。我们让它们在单独的常量中可用，这样使用这些值的代码不需要使用冗长的 C++ 标准库调用来获取它们的值。

当 $e_b = 255$ 时，非零有效位对应于特殊的 NaN 值，这是取负数的平方根或尝试计算 0/0 等操作的结果。NaN 通过计算传播：任何操作数之一是 NaN 本身的算术运算总是返回 NaN。因此，如果一个 NaN 是从一个长的计算链中产生的，我们就知道在这个过程中有些地方出了问题。在调试构建中，pbrt 有许多 Assert () 语句来检查 NaN 值，因为我们几乎从来都不期望它们在常规事件过程中出现。任何与 NaN 值的比较都返回 false；因此，检查!(x==x) 用于检查值是否不是数字。为了清晰起见，我们使用 C++ 标准库函数 std::isnan() 来检查。

2 2 PBRT 中的实用方法

本节我们会学习几个基本操作函数，如果您看不懂也没有关系，只需要注意，NextFloatUp(float v) 函数可以将浮点数增加一个可表示的最小量，NextFloatDown(float v) 可以将浮点数减小一个可表示的最小量。

下面是正文：

对于某些低级操作，能够根据浮点值的组成位来解释浮点值并将表示浮点值的位转换为实际的浮点值或双精度浮点值是有用的。一种自然的方法是将指向要转换的值的指针转换为指向另一种类型的指针：

```
float f = ...;
uint32_t bits = *(uint32_t *)&f);
```

然而，C++ 的现代版本规定，将一种类型的指针浮点到另一种类型 uint32_t 是非法的（这个限制允许编译器在分析两个指针是否指向相同的内存位置时更积极地进行优化，这可以抑制寄存器中的存储值）。另一种常见的方法是对两种类型的元素使用 Union，分配给一种类型并从另一种类型读取：

```
union FloatBits {
    float f;
    uint32_t ui;
```

```
    };
    FloatBits fb;
    fb.f = ...;
    uint32_t bits = fb.ui;
```

这也是非法的：C++ 标准表示读取与最后一次使用的不同的 Union 元素是未定义的行为。使用 memcpy() 将指向源类型的指针复制到指向目标类型的指针，可以正确地进行这些转换：

```
inline uint32_t FloatToBits(float f) {
    uint32_t ui;
    memcpy(&ui, &f, sizeof(float));
    return ui;
}
inline float BitsToFloat(uint32_t ui) {
    float f;
    memcpy(&f, &ui, sizeof(uint32_t));
    return f;
}
```

虽然对 memcpy() 函数的调用似乎要花费高昂的代价来避免这些问题，但在实践中，好的编译器会将其转换为 no-op，并将寄存器或内存的内容重新解释为另一种类型。（这些在 double 和 uint64_t 之间转换的函数的版本在 pbrt 中也可用，但它们是相似的，因此不包括在这里。）这些转换可用于实现将浮点值上下凹凸为下一个较大或下一个较小的可表示浮点值的函数。它们对于我们需要在代码中遵循的一些保守的舍入操作非常有用。由于 float 在内存中表示的细节，这些操作非常有效，使用 NextFloatUp() 和 NextFloatDown() 函数来实现，这里不描述其细节。

2 3 算术运算与误差传播

IEEE754 对浮点运算的特性提供了重要的保证：具体来说，它保证了加法、减法、乘法、除法，和平方根给出相同的结果，给定相同的输入，并且这些结果是最接近在无限精度算法中执行的计算结果。值得注意的是，这在有限精度的数字计算机上是可能的；IEEE754 的成就之一是证明了这种精度是可能的，并且可以在硬件上相当有效地实现。

在加法中的误差界定为：

$$round(a+b) = (a+b)(1\pm\epsilon) = [(a+b)(1-\epsilon), (a+b)(1+\epsilon)] \tag{二.2}$$

多次计算以后，误差是可以累积的。累积的方法可以通过逼近来得到范围，而且，有时候同样的计算误差界定范围会不同，比如 (((a+b)+c)+d) 与 ((a+b)+(c+d)) 是不同的。

这种计算误差的方法称为前向误差分析；给定计算的输入，我们可以应用一个相当机械的过程，为结果中的误差提供保守的界限。在实践中，结果中导出的界限可能会夸大实际误差，误差项的符号通常是混合的，因此在添加时会有抵消。另一种方法是反向误差分析，它将计算结果视为精确的，并提供给出相同结果的输入扰动的界限。这种方法在分析数值算法的稳定性时更有用，但不太适用于推导我们感兴趣的几何计算的保守误差界。

除了用代数方法计算出误差界限外，我们还可以让计算机在执行某些计算时为我们做这项工作。这种方法称为运行错误分析。它背后的思想很简单：每次执行浮点运算时，我们也会计算“计算间隔”的项，以计算迄今为止累积的误差的运行界限。虽然这种方法比直接导出给出错误界限的表达式有更高的运行时开销，但是当导出变得不易处理时，这种方法会很方便。

PBRT 提供了一个简单的 EFloat 类，它的作用主要类似于一个常规的 float，但是在计算这些错误边界时使用操作符重载来提供所有关于 float 的常规算术运算。与 Interval 类类似，EFloat 跟踪描述感兴趣

值的不确定性的 Interval。与区间相反，EFloat 的区间是由于中间浮点运算的错误而不是输入参数的不确定性引起的。

为了简单起见，这里简化一下描述。

$$round(round(round(a+b)+c)+d) = \tag{二.3}$$

$$((((a+b)(1\pm\epsilon_m))+c)(1\pm\epsilon_m)+d)(1\pm\epsilon_m) \tag{二.4}$$

$$= (a+b)(1\pm\epsilon_m)^3 + c(1\pm\epsilon_m)^2 + d(1\pm\epsilon_m) \tag{二.5}$$

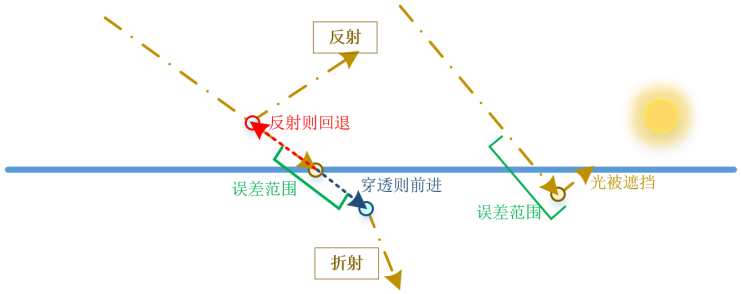
对于一定的错误界限 $(1\pm\epsilon_m)^n$，我们可以得到它的界限被界定在 $1+\theta_n$ 范围内（由 PBRT 书 3.9.1 节所得）：

$$|\theta_n| \leq \frac{n\epsilon_m}{1-n\epsilon_m} \tag{二.6}$$

在程序中，对应 inline constexpr Float gamma(int n) 函数（我们之前在 Ray 与三角形或者球体求交时遇到过，不过当时我们将它删除了）。

三　Ray 求交中的误差管理

本章剖析 PBRT 中对 Ray 求交等计算的误差管理方法。我以前写过的一些光线追踪系统中，当光线与物体求交点后，如果根据物体表面材质的计算是反射，则交点需要后退一个微小的单位；如果根据物体表面材质计算的结果是穿透过去，则交点需要前进一个极小单位，这是避免由于数值精度引起的 Ray 与物体表面重复求交（右边由于精度问题，在反射型材质中，光照被自己遮挡）：



而加入数值方法以后，我们就能更好地计算前进或者后退的合理长度，使得性能更鲁棒。

3 1　Ray 与包围盒求交

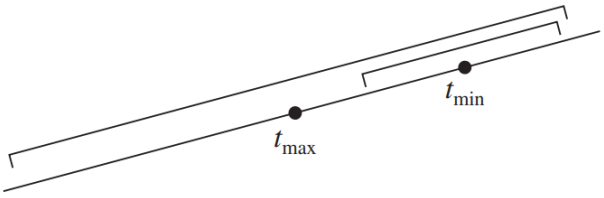
浮点舍入错误会导致错过 Ray 实际与边界框相交的情况。虽然偶尔的误报是可以接受的，但我们希望永远不会错过实际的相交。正确处理这一点对于 BVHAccel 加速数据结构的正确性非常重要，这样就不会错过有效的 Ray-Shape 相交。

Ray 与包围盒求交 t 的计算是 $t = (x - o_x)/d_x$，对于加减乘除的界定不一样 (我们用 round*(a,b) 表示 a 与 b 相乘带来的界定误差)：

$$round(t) = round * (round - (x, o_x), round/(a, d_x)) \tag{三.1}$$

$$= (1 \pm \epsilon)^3 \tag{三.2}$$

计算出的 t 值误差界定范围，我们关心的情况是区间重叠；如果它们不重叠，那么计算值的比较将给出正确的结果，否则就无法判断了：



但我们认为这种情况最好还是判断成相交了，否则可能会漏掉。因此我们让 t_{max} 的错误界定区域变大两倍，

```
1    tFar *= 1 + 2 * gamma(3);
2    t0 = tNear > t0 ? tNear : t0;
3    t1 = tFar < t1 ? tFar : t1;
4    if (t0 > t1) return false;
```

我们可以直接移植到自己的系统上（您要是实在不想移植也没关系，这种极端情况很少见，顶多漏掉几个边边角角那种很难发现的面片）。

3 2　鲁棒的 Ray-Shape 相交

还记得三角求交的 TPS 变换吗（见《形状和加速器》）？通过将顶点位置变换到 Ray 坐标系可能会引入舍入误差，但该误差不会影响交叉测试的水密性（即不会漏掉中间的三角形），因为相同的变换适用于所有三角形。（此外，该误差非常小，因此不会显著影响计算交点的精度。）至于实际三角形求交程序中的

误差界定问题并不复杂，但也可以看出是由加减乘除构成的，另外还可以引入“双精度测试”，将无法判断的情况使用双精度判断，这里不再提。

球体的求交和误差界定同理，根据求交计算的公式计算出。我们求交的程序已经改了，所以误差界定也要改，怎么改读者有兴趣可以自己计算（或者您考虑移植 PBRT 球体求交的程序）。

3 3 鲁棒的产生 Ray 原点

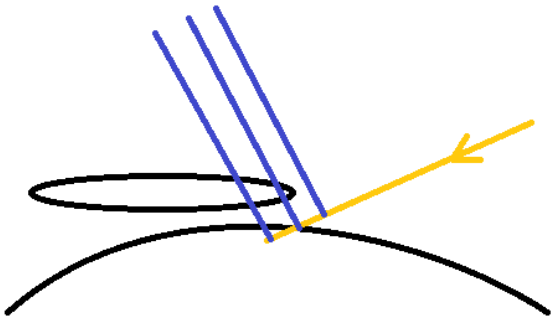
这一节还是有必要说说的产生新的 Ray 原点的。

我们必须保证 Ray 与面的交点在表面上下更合理的位置，否则产生的新 Ray 可能又会与自己本身相交。为了确保生成的光线原点肯定位于合理位置，我们沿法线移动足够远，以便垂直于法线的平面位于误差界定之外。要了解如何做到这一点，请考虑在原点处计算的交点，其中通过交点的平面的平面方程为 $f(x,y,z)=n_x x+n_y y+n_z z=0$，n 表示向量。

对于不在平面上的点，平面方程 $f(x,y,z)$ 的值给出了沿法线的偏移量，该偏移量给出了穿过该点的平面。我们想找出误差边界框的 8 个角点的 $f(x,y,z)$ 的最大值；如果我们偏移平面加上和减去这个偏移，我们有两个不与误差框相交的平面，它们应该（局部地）在曲面的相对边上，至少在沿法线计算的交点偏移处。计算了偏移后的 Ray：

```
1    // 产生新的Ray，d是产生的Ray的方向
2    Ray SpawnRay(const Vector3f &d) const;
3    // 产生阴影Ray，p2是光源上的采样点
4    Ray SpawnRayTo(const Point3f &p2);
5    // 从表面交点产生新的Ray
6    Ray SpawnRayTo(const Interaction &it);
```

关于书 [1] 中 232 页关于 SpawnRayTo 的表述，添加一点我的想法：当 Ray 击中 Shape 对象以后，t 值回返 1 个浮点值（即回退 1 个微小距离值）是可以的，毕竟一般来说 t 不算很大的时候，1 个浮点值真的非常小，书 [1] 中说可能会错过相距特别近的物体遮挡造成的阴影：



如上图，黑色代表物体，蓝色代表反射的线，可以看到，正确交点靠内的点和正确的交点一样，会被遮挡，但是后退一点以后可能就不会被遮挡了（有可能我们本身求出来的交点就在正确交点偏上一点，再往上移一点就更容易不被遮挡到了）。

四　内存管理和误差界定的移植

虽然前面说不建议初学者移植，但是对于想追求更稳定的读者还是可以移植一下的。这里只教怎么移植，在以后的系列书里我会默认大家已经移植了这些东西（虽然研究原理比较难，但实际代码其实只是增加了一些额外的内容项）。

内存管理就不说了，直接把类和用到的内容复制过去，用的时候直接对比着 PBRT 上面用就好了，反正也不是多线程安全的，没什么乱七八糟的线程和互斥的内容，很简单。以后的系列书我默认大家没有移植内存管理类的内容。

误差界定中，复制这些函数到 FeimosRender.h 头文件中：

```
FloatToBits
BitsToFloat
NextFloatUp
NextFloatDown
OffsetRayOrigin
```

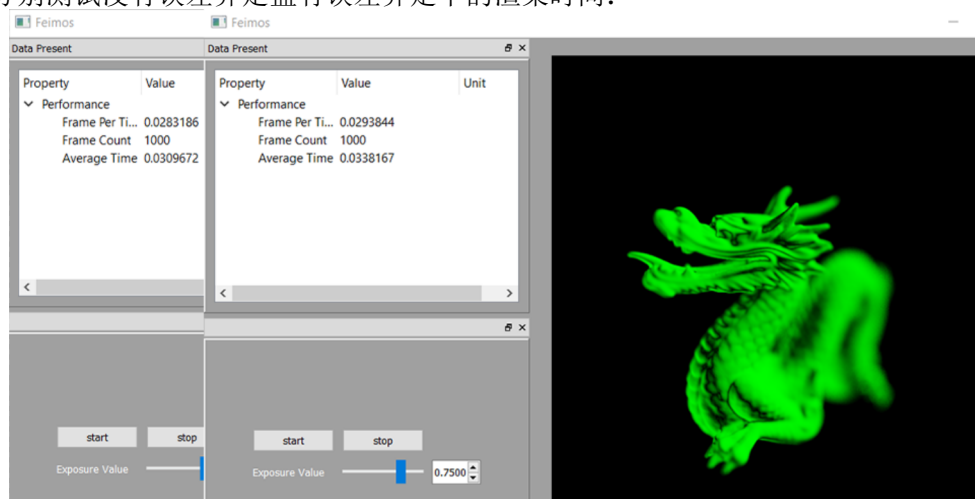
Bounds3<T>::IntersectP 的两个函数都可以将误差计算的内容直接放上去。

Ray 与三角形求交的计算中，把下面这些代码全都移植过来：

```
// Ensure that computed triangle $t$ is conservatively greater than zero
......
// Compute $\delta_z$ term for triangle $t$ error bounds
......
// Compute $\delta_x$ and $\delta_y$ terms for triangle $t$ error bounds
......
// Compute $\delta_e$ term for triangle $t$ error bounds
......
// Compute $\delta_t$ term for triangle $t$ error bounds and check _t_
......
```

剩下的内容就与 Ray 的再产生有关了，因为我们的 Interaction 类里面的 SpawnRay 和 SpawnRayTo 函数都没有放入（之前移植形状类的时候都删掉了，而现在暂时我们又用不到），所以我们暂时不移植它们，以后遇到的时候我会提几句。如果您还记得 Whitted 光线追踪器的流程，应该知道，积分器一开始会定义一个 SurfaceInteraction 对象，该对象的父类 Interaction 有一个成员变量 Vector3f pError，表示界定误差，它会记录 Ray 的产生和求交中的误差。求交时更新误差界定范围，并在调用函数 SpawnRay() 来产生 Ray 时会应用该误差界定范围。

最终我分别测试没有误差界定盒有误差界定下的渲染时间：



左边是没有使用误差校验的平均渲染时间，右边是使用了误差界定的平均渲染时间，使用了误差界定后的平均时间要慢百分之十。

由于我们现在已经介绍了误差界定的作用（其实就是场景求交点），以后我们在移植代码的过程中就不跳过这些内容了，而是把它们都移植上。

五　本书结语

可能是因为我确实对误差范围这方面的了解过少，所以写起来非常吃力，不知道如何取舍。

在误差界定中，我通常的做法是，只处理 Ray 和包围盒以及三角形相关的内容（Ray 的产生，Ray 与包围盒求交，Ray 与三角形求交，Ray 的再产生），其他形状类要是都搞一遍太麻烦，一般三角形就能胜任所有的场景，所有的形状都可以细分为三角面片。

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.