GPU光线跟踪算法加速结构研究

王世元　温柳英

西南石油大学计算机科学学院　成都　610500

摘要：基于GPU的光线跟踪算法是当前图形学研究的一个热点，也是将来用于广告、电影、游戏等娱乐产业的关键技术。本文论述了如何对基于GPU的光线跟踪算法进行实现，以及利用各种加速结构，加速算法实现，提高算法执行效率，并对各种加速结构的效果进行了比较研究。

关键词：GPGPU　光线跟踪　BVH　KD-Tree

doi:10.3969/j.issn.1006- 8554.2010.05.005

Research of Acceleration Structures for GPU Based Ray Tracing

Wang shiyuan，Wen liuying

(Computer Science College, Southwest Petroleum University, Chengdu Sichuan, 610500)

Abstract: The Ray tracing algorithm based on GPU is a hot research on Computer Graphics，it also an important technology for advertisement, movie and game. This paper discuss how to implement the GPU Ray Tracing Algorithm, and use various acceleration structures to accelerate the algorithm, improve the efficiency of execution, and compare the effect of the three kinds of acceleration structures.

Key words: GPGPU；Ray Tracing；BVH；KD- Tree

1. 引言

近年来，GPU无论在运算能力，还是在可编程性上都得到了大幅的提高，GPU已经在需要大量运算的密集运算领域发挥了举足轻重的作用。各种基于CPU的密集运算被移植到GPU上，以利用GPU巨大的运算能力，加速整个算法的运算过程。光线跟踪算法是生成真实感图形的一种非常重要的方法，在电影、游戏、广告等产业，获得广泛的应用，而光线跟踪算法也是典型的密集运算算法，利用原始的基于CPU的光线跟踪渲染一幅图片是非常耗时的操作。因此，如果能够将CPU上的光线跟踪算法，映射到GPU上，加速光线跟踪算法的执行时间，将会带来巨大的经济效益。因此，基于GPU的光线跟踪算法已成为国内外科研人员的研究热点。

2. 基于GPU的光线跟踪

2.1　相关工作

当前，主要由两种方法利用GPU来加速光线跟踪算法。第一种是Carr等人提出来的，将GPU转换为一个蛮力的执行光线—三角形求交的计算器，而将任何的光线生成以及着色过程在CPU上完成。这就需要CPU依然执行绝大部分的渲染工作。Carr等人指出，在ATI Radeon 8500上，每秒最快能够执行1亿2千万次的光线—三角形求交。同时，作者也指出，由于GPU的单精度浮点的限制，图片上依然存在一些不太真实的地方。

第二种方法由Purcell等人提出的，改种方法将整个光线跟踪器都移植到GPU上进行实现。从光线的产生，加速结构的遍历，到最后的着色过程都在GPU上执行。此后，有很多相同的项目都是基于Purcell的模型上进行的。

2.2　GPU上的光线跟踪算法的映射方式

将传统的CPU上执行的光线跟踪算法，映射成为一个GPU协助的，或者基于GPU的光线跟踪器有众多方法。下面重点介绍Purcell提出的映射模型，以及在本文的实现中提出的一个基于GPU的Whitted模型的光线跟踪器。该光线跟踪器的布局如图2.1所示：

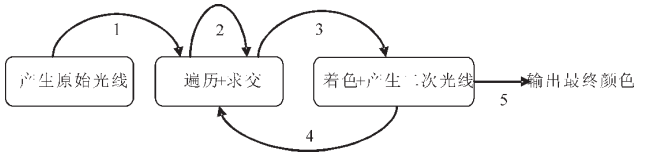

图2.1　基于GPU的光线跟踪的模型

在Purcell的论文中，它将光线—三角形求交，以及遍历过程分离成两个独立的遍历内核和求交内核。本文的实现中，也按照上述模型图，将光线跟踪算法分解成光线生成，光线—三角形求交，着色这三个步骤。

在对光线进行跟踪之前，需要生成从视点指向屏幕的原始光线（primary ray）。在一个GPU上，能够使用光栅器的插值的能力，在一个单一的内核调用中，产生所有的原始光线。

给定观察矩形（被采样用于产生图片的投影平面的一部分）的四个角，以及视点，首先计算出这个视锥体的四条边线。如果让光栅器在这4条光线之间，按照512× 512规格，在这四条光线之间按照方向进行插值，最终就可以获得能够产生一幅512× 512图片（一个像素一个采样点）的所有原始光线的方向。同时能够将这些方向存储在一个纹理里，并把它作为求交内核的输入。所有的原始光线具有相同的起始点，但是仍然把它存

储在一个同方向纹理具有相同维度的纹理内。因为当生成阴影光线或者反射光线的时候，光线的原点会发生改变。

求交内核把光线的原点，方向，以及场景的描述作为输入数据。在内核被调用数次之后，我们对于每一个像素输出一个击中记录。如果一条光线击中了场景中的某个三角形，返回击中点的3个重心坐标，以及相关的被击中的三角形。此外，还将输出被发现的交点沿光线的距离，以及被击中三角形的材质。这就需要使用5个浮点数值组成一个击中记录。纹理只能够支持4个颜色通道（RGBA），所以，如果能把击中记录裁减到4个值，那么将是非常有益的。

观察发现，只需要3个重心坐标的两个，因为在三角形内部，它们相加的和总是1。这就使得在一个单独的RGBA纹理中存储交点记录是可行的，并且它的维度同其它两个光线纹理的维度相同。

Moller和Trumbore提出了一个高效的光线—三角形求交算法，使用这个算法，并利用GPU在向量计算上的优势来进行求交计算。下面列出了求交的代码，这个代码也展示了如何利用向量指令来提高效率。

```
/**********************求交程序**********************
*****/
float4 Intersects ( float3 a, float3 b, float3 c, float3 o, float3 d,
float3 minT,
        float mat_index, float4 lasthit )
{
 float3 e1 = b- a;
 float3 e2 = c- a;
 float3 p = cross(d, e2);
 float det = dot(p, e1);
 bool isHit = det > 0.000001f;

 float invdet = 1.0f / det;
 float3 tvec = o- a;
 float u = dot(p, tvec) * invdet;

 float3 a = cross(tvec, e1);
 float v = dot(q, d)*invdet;
 float t = dot(a, e2)*invdet;

 isHit = (u >= 0.0f) && (v >= 0.0f)&&(u + v <= 1.0f)&&(t <
lasthit.z)&&(t > minT);

 return isHit ? float4(u, v, t, mat_index) : lasthit;
}
```

当所有的原始光线都已经计算出了相交的状态的时候，就能够查询着色过程所需要的表面法线和材质的信息。每一个击中记录都存储了一个指向材质纹理的索引，这个材质纹理包含了三角形的法线，材质颜色以及类型。三个顶点的法线根据击中记录的中心坐标进行了插值。最终的颜色能够按$(N\cdot L)C$进行计算，此处N是法线，L是光源的方向，C是三角形的颜色。

现在根据击中的三角形所具有的材质的类型（漫反射材质，或者镜面反射材质），需要产生二次光线，以此来计算阴影和反射。

1）如果一条光线射出场景之外，像素就被赋予全局的背景颜色。

2）如果一条光线击中了一个漫反射材质表面，就发射一条阴影射线（shdow ray）。这些光线的起始点在击中点，方向为从击中点指向光源。

3）如果一条光线击中了一个镜面反射材质表面。就发射一条镜面反射光线。镜面发射光线的起始点也在击中点，但是它的方向是在击中点处关于入射光线和插值后的法线对称的方向。一个真正的Whitted类型的光线跟踪器也支持透明材质，从而能够产生折射光线。但由于主要是研究加速结构，所以在本文的实现中，没有考虑折射光线。

4）如果阴影光线击中了某个几何体，这就说明在光源和击中点之间，存在某个几何体，所以这个像素就应该是黑色（处于阴影中）。当跟踪阴影光线的时候，不关心最近的那个击中点，更加关心的是是否存在这样的击中点。因此，当有一个交点被发现，就可以停止整个求交过程，从而加速算法的处理过程。在本文的实现中，以相同的方式跟踪阴影光线和反射光线，因此，就没有使用到这个优化策略。

已经对每一个像素产生了正确二次光线，如果需要，就能够执行另外一趟遍历/求交过程，对上述的二次光线进行跟踪。每一次调用着色程序就能够对每一个像素返回一个颜色值和一条新的光线。着色内核也可以将前一次着色程序的输出当作本次着色程序的输入。这就使得能够在跟踪连续的光线的时候合并这些连续的镜面反射的颜色。

同Carr等人的程序不同，本文所采用的程序不存在浮点精度太低的问题，因为Geforce 7300在整个管线中支持真正的32位浮点操作。

3. 加速结构的实现和比较

3.1 均匀栅格

均匀栅格是第一个在GPU上实现的加速结构。Purcell给出了很多选择均匀栅格作为加速结构的理由，但是Purcell没有详细的说明为什么均匀网格对于硬件实现而言比其它的加速结构要更加的简单。当在探讨了均匀栅格的一些主要特性的时候，更加清晰的知道了均匀栅格为什么会成为一个好的GPU机速结构。

首先，只用使用简单的算术运算，就能够对于每个体素的遍历在常量时间能被定位和存取。这就消除了对树的遍历的需要，以及重复的纹理查找工作，而纹理查找是相当耗时的。

其次，体素的遍历是通过递增算术运算来完成的。这就消除了对堆栈的需要，使得我们能够从光线的起始点开始，以距离递增的顺序访问体素成为可能。

再其次，由于对于体素的访问是沿着光线，以距离递增的方式遍历的，所以，一旦在一个被访问的体素中报道发现有一个交点，就可以停止这条光线对体素的遍历过程，从而提高整个遍历过程的速度。

最后，用于遍历的代码非常适合用向量编写，而向量形式

的编码风格又非常适合GPU的指令集。

然而，均匀栅格的缺点就是由于它是空间细分结构的一种特殊情况，多个体素可能包含相同三角形的多个引用。由于无法使用mailbox技术，这就意味着需要对于相同的光线和三角形之间进行不止一次的相交测试。

3.2 KD-tree

最近，Havran等人对基于CPU的光线跟踪算法的加速结构进行了比较，得出的结论是对于众多不同类型的测试场景，平均而言，KD-tree是最快的。所以，有必要考察一下对于基于KD-tree的GPU光线跟踪算法，是否也会有相似的结论。

就像均匀栅格一样，KD-tree也是一种空间细分结构。同均匀网格不同的是，KD-tree利用一个二叉树将场景表示成一个层次结构。

在二叉树中，我们将内部节点和叶子节点区分开。叶子节点用来表示体素和与之相关的保存在该体素内的三角形的引用。一个内部节点用来表示空间区域的某个部分。所以，内部节点包含一个分裂面的两个子树的引用，而叶子节点只包含一个三角形列表。

KD-tree的创建过程从上而下，根据一个评价函数，通过放置一个分离平面，递归的将场景分离成两个体素。我们能够以递归的方式遍历KD-tree，但是由于GPU没有堆栈结构，所以无法应用递归的策略。取而代之的是，我们能够通过记住我们沿着光线前进了多远来向上或者向下遍历树。这种策略消除了需要堆栈的限制，使得用GPU来完成对KD-tree结构的遍历成为可能。

当使用GPU对KD-tree进行遍历的时候，KD-tree像均匀栅格那样被表示成一个纹理的集合。这就意味着有一个保存树数据的纹理，一个保存三角形列表的纹理，和一个保存实际的三角形数据的纹理。GPU的遍历首先调用一个初始化内核，然后按照需要，多次调用合并后的遍历和求交内核。

3.3 包围体层次(BVH)

给定一些随机的光线，通过计算遍历包围体层次的平均花费，就可以测量出该包围体层次的质量。迄今为止，还没有构建最优的包围体层次的算法，也就是说，如何准确的测量一个包围体层次的平均遍历时间还不是很明显。

Goldsmith和Salmon提出了一个评价函数，通常被称为表面积启发式函数。他们通过父节点和孩子节点的表面积之比来形式化的表述这个关系，此评价函数如下所示：

$$p(hit(c)\mid hit(p)) \propto \frac{S_c}{S_p} \qquad (3.1)$$

此处，$hit(n)$是光线击中节点n的情况，S_n是节点n的表面积，c和p分别表示父节点和孩子节点。

这个评价函数给出了，当用一条随机的光线同层次结构求交的时候，成本上的估计。由于没有最优的方法去有效的构造一个最优的BVH，提出了不同的构造技巧。下面，将列出比较通用的方法。

在实践中，对于包围体应用的最广泛的就是轴对齐包围盒(AABB)。AABB易于实现，并且同光线的求交测试非常快。大多数有关BVH的论文在描述BVH的创建的时候，通常分别以Kay和Kajiya，或者Goldsmith和Salmon这两种基本的想法为基础。Kay和Kajiaya建议以自上而下递归的方式进行BVH的创建。

Goldsmith和Salmon提出了一个更加复杂的自底向上的构造方式。Goldsmith和Salmon指出，BVH的质量同作为输入传入的三角形的顺序有关。因此，他们建议在构造BVH之前，随机打乱三角形的顺序。下述算法就是利用Kay/Kajiya的思想创建某个场景的包围体层次的方法：

```
/******************Kay/Kajiya 的 BVH 创建方法*********************/
BVNODE BuildTree(triangles)
    If we passed just one triangle
        Return leaf holding the triangle
    Else
        Calculate best splitting axis and where to split it
        BVNODE result
        Result.leftChild = BulidTree(triangles left of split)
        Result.rightChild = BulidTree(triangles right of split)
        Result.boundingBox = bounding box of all given triangles
        Return result
```

4. 结束语

本文成功的在GPU上实现了用于光线跟踪算法中的各种加速结构，并对这些加速结构在GPU上的加速效果进行了比较。均匀栅格作为第一个在GPU上实现的光线跟踪器的加速结构，也被证明是最慢的，除非是只包含一个单独的物体的场景的情况。均匀栅格不适合几何体的密度非常高的场景。另外，对于均匀栅格的GPU上的遍历表示，也需要大量的数据。Foley和Sugerman认为，对于大多数场景，KD-tree的效率要比均匀栅格高。但是，在KD-tree的遍历过程中，无论是重置阶段还是回退阶段，片元程序都非常的复杂，但这种复杂性也使得其能够在场景的几何体的密度改变的时候做出适当的调整。本文实现的BVH被证明在加速效果上要超过均匀栅格和KD-tree，在现阶段，BVH是在GPU上实现的最快的加速结构。并且在GPU上实现BVH加速结构要比实现其他加速结构更加的简单。

参考文献：

[1] Randima Fernado 编，姚勇，王小琴译. GPU精粹-实时图形编程的技术，技巧和技艺[M]. 北京：人民邮电出版社，2006.

[2] Matt Pharr 编著，龚敏敏译.GPU精粹2-高性能图形芯片和通用计算编程技巧[M].北京：清华大学出版社.

[3] 吴恩华,柳有权.基于图形处理器(GPU)的通用计算[J].计算机辅助设计与图形学学报, 2004, 16(5): 601-612.

[4] Philip J.Schneider,David H.Eberly 著，周长发译. 计算机图形学几何工具算法详解[M].北京：电子工业出版社，2005.

[5] Martin Christen. Implementing ray tracing on GPU. Master's thesis, University of Applied Sciences Basel, Switzerland, 2005.