

# 基于KD树深度遍历的交互式光线跟踪算法

曹家音

## 摘要

本文介绍了一种基于KD树的GPU光线跟踪遍历算法。该算法利用GPU的高度并行计算能力，充分优化了光线跟踪算法的性能。

在GPU的算法实现中，一级光线的求交结果通过光栅化的方式得到，使得一级光线的求交计算加速很多。除此之外，算法还在GPU内核中实现了虚拟栈，光线在GPU内核中同样对于KD树进行了深度遍历。从而有效的提高了光线遍历过程的性能。

算法最终达到了相对于传统CPU算法20到80倍的加速比。对于分辨率较小的图片，可以达到交互式的性能。

## 1. 光线跟踪简介

光线跟踪是一种全局图形渲染技术，可以生成照片级质量的图片。光线跟踪可以很好的解决一些全局特效，例如阴影，反射和折射等，而这些特效在传统的光栅化的局部渲染模型中，很难得到完美的实现。

光线跟踪对于硬件的消耗，要远远大于光栅化算法。当前处理器的内存吞吐量很难使光线跟踪达到实时的性能。一直以来，光线跟踪都被视为一种在图片质量远远比渲染速度重要的情况下才使用的离线渲染技术。

在可编程Shader技术发布后，很多学者利用GPU来进行光线跟踪的加速。2002年，[CHH02]在他们的光线引擎中，第一次试图在GPU上实现光线跟踪算法，然而只有三角形求交算法是在GPU上实现的。带宽的限制很快的成为了算法的瓶颈。为了避免大量数据在CPU与GPU之间传输，[PBMH05]等学者把光线跟踪中的绝大部分工作移植到了GPU上。然而由于当时的可编程GPU接口仍然不完善，所以大部分的GPU光线



图 1. 本文实现的光线跟踪算法生成的图片

跟踪算法都没有超过一个优化后的CPU算法。

随着可编程GPU的发布，更多的学者试图着在GPU上实现光线跟踪算法。其中[FS05]提出了kd-restart算法，在GPU上进行KD树的遍历，从而有效的改进了之前算法的局限性。然而由于kd-restart的最坏性能，导致了算法的效率仍然不是很好。[PGSS07]在KD树上增加了‘绳子’的概念，从而改进了kd-restart的遍历性能。

[ZHWG08]利用CUDA实现了实时的KD树创建算法，并且对于动态的场景进行了交互式光线跟踪渲染。

本文提出的算法，同样相对于复杂的场景进行交互式渲染。本课题的工作，在GPU内核中实现了光线跟踪算法中的所有过程，其中包括一级光线的生成，光线与三角形求交，次级光线的生成，KD树的深度优先遍历，像素着色等过程。并且在GPU内核中模拟了栈的行为，从而充分优化了kd-restart算法。该算法相对于CPU加速高达20-80倍，每秒钟可以遍历1.5-3M的光线。

## 2. 本文的光线跟踪算法

本文的光线跟踪算法采用基于SAH模型分割的KD树进行空间划分，从而加速求交过程[Hav00][MB90]。

### 2.1. 光线跟踪算法概述

该算法利用CUDA提供的高度并行计算能力，充分优化了原有算法中的串行部分。算法的大体思路如下：

---

#### Algorithm 1 Ray Tracing Algorithm Overview

---

```
generate primary rays
a raster pass for triangle index
while currentDepth < maximumDepth do
    if currentDepth == 0 then
        get the intersected points between rays and triangles directly.
    else
        traverse kd-tree to get the intersected point.
    end if
    shade the pixels.
    generate next level rays.
    currentDepth++
end while
post process for clearing the noise in the image.
```

---

首先，算法会根据视点以及投影矩阵信息生成一级光线。然后会利用光栅化的原理，生成覆盖每个像素的三角形ID，这里会采用一个像素着色器把三角形的ID输出到图形接口的渲染目标上。

在上述处理结束后，算法开始进入光线跟踪部分，在一级光线的求交过程中，由于与每条光线相交的三角形的ID已经输出到了渲染目标上，所以没有必要再进行KD树的遍历，这里GPU内核直接把光线与相应的三角形所在的平面求交，即得到了求交结果。对于次级光线的求交，由于场景的拓扑结构不同，很难有一个定式来描述光线的分布，所以光线只能遍历KD树。

在得到求交结果后，会根据当前交点与光线等信息为像素着色。这里的着色部分包括环境光，漫反射，高光部分，以及阴影的生成。其中环境光，漫反射，高光的着色部分与传统的光栅化着色没有本质区别。而阴影的计算是从光源生成一条到交点的线段，然后

利用KD树把场景中的三角形与线段求交。如果交点存在，则当前光源对于这个像素没有光照贡献。反之进行上述着色过程。

着色完成后，算法会根据当前相交三角形的材质信息生成下一级的光线。这里生成的光线会离散的分布在内存中，待光线生成后，利用一个prefix sum的算法[HSO07]，把所有离散的光线集中到内存前端，从而得到一个内存连续的光线集合。

在上述循环结束后，算法的核心部分已经计算结束了。但是由于光线跟踪本身的一些限制，会产生一些噪音。本文采用一个简单后处理的方法对产生的图像降噪，从而得到更优质量的图片。

上面算法的所有流程都是在GPU上进行并行计算的。下面的部分会对上述算法中的几个核心部分进行详细的描述。

### 2.2. 利用光栅化模拟一级光线求交过程

光线跟踪与光栅化算法的本质区别在于光线跟踪是全局光照模型，而光栅化是局部光照模型。其本身的特性就决定了各自的优势。光线跟踪的优势在于引入了光线的递归遍历，从而可以得到不同图元之间的交互信息，进而生成反射和折射等信息。光栅化的优点在于可以以实时的速度生成图片，并且支持一定的光照模拟算法。而除去着色过程，两者在为图元生成像素位置信息是基本一致的。

如果光栅化过程输出到每块像素内存中的元素不是颜色信息，而是三角形索引，并且光线跟踪也同样生成三角形索引。两块索引缓冲的相似度高达90%。而不同的索引也是由于三角形的边缘而引起的。所以可以用光栅化的方法来取代一级光线的KD树遍历过程。

在用光栅化输出了三角形的索引后，算法可以利用光线和三角形求交的方法，直接一对一的生成每一个交点信息。然而这里的三角形和光线是已知相交的，所以上述过程的三角形求交过程可以简化为平面与射线的求交算法。从而进一步优化了一级光线的求交过程。

### 2.3. KD树遍历算法

KD树遍历算法对于光线跟踪的性能非常重要。一个优秀的遍历算法可以使光线跟踪的性能提升很多。然而KD树的深度遍历是一种基于栈的递归遍历模

型，而GPU内核中对于递归的调用会限制一些，所以在GPU内核中遍历KD树一直是采用非栈的方式进行调用的。

本文的算法利用位操作在GPU内核中模拟了一个虚拟栈，从而使GPU内核同样以一种深度优先的方式进行KD树的遍历。

由于GPU内核中对于递归调用有一定的限制，所以这里的回溯算法是采用非递归方式。用一个模拟栈的形式进行同样行为的遍历。这里的虚拟栈是用32位无符号整数模拟的。原因有以下两点：

- 32位无符号整数是在内核的局部内存空间中的，访问效率非常高。而且内核中的位操作效率也很高。
- 没有采用内核中的局部数组进行虚拟栈的模拟，因为这样会耗用GPU硬件更多的寄存器。从而导致线程分配效率相对低一些。32位无符号整数只用硬件一个寄存器就可以完成，可以更好的节省硬件资源。

32层的KD树已经能满足本课题实验中的所有要求了，所以一个无符号整数模拟的栈可以完成光线遍历的过程。当然，如果需要遍历更深层次的KD树，则可以用多个无符号整数或者其他局部内存空间来模拟。

算法2描述了GPU内核遍历KD树的详细过程。首先，算法对于光线与根节点的包围盒进行求交测试。如果光线没有和包围盒相交，直接返回。

然后，开始对光线进行KD树的遍历。刚进入循环时，根据光线和包围盒的交点就可以判断光线先进入当前包围盒的那个子节点。通过交点与分割面的关系，可以对于KD树进行向下遍历。这里不需要做多余的光线和包围盒的求交过程。

在向下遍历到第一个叶子节点后，对于该叶子节点进行三角形求交。GPU内核程序会遍历当前叶子节点中的所有三角形，然后依次与光线进行求交测试。如果内部有三角形与光线相交，则返回交点离光线原点最近的三角形ID。如果内部没有三角形与光线相交，则需要进行回溯过程。

回溯过程的工作方式是这样的。首先检查当前层数标记位是否为1。这里1代表当前节点的兄弟节点已经被访问过了，而0代表当前节点的兄弟孩子没有被访问。如果当前层数所对应的标记为是1，那么直接

---

**Algorithm 2** Stack based KD-Tree Traverse

---

```
ray ← Ray
currentNode ← Node
intersectedPoint ← float4
currentTraverseDepth ← int(0)
mask ← uint(0)
currentNode = root
if intersect(ray,currentNode.BoundingBox,intersectedPoint)==0
then
    return -1
end if

repeat
while TRUE do
    if currentNode is a leaf then
        break
    end if
    if intersectedPoint < currentNode.splitPos then
        currentNode = currentNode.left
    else
        currentNode = currentNode.right
    end if
    currentTraverseDepth++
end while

triId = -1
minLen = FLTMAX
for all tri in currentNode do
    if intersect( ray , tri ) < minLen then
        update triId
        update minLen
    end if
end for
if triId > 0 then
    return triId
end if

while currentTraverseDepth > 0 do
    flag = 0x00000001 << currentTraverseDepth
    if mask & flag then
        mask &= ~ flag
        currentTraverseDepth -= 1
        currentNode = currentNode.parent
        continue
    end if
    otherNode = currentNode.brother
    if intersect(ray,otherNode.BoundingBox,intersectedPoint)
    then
        currentNode = otherNode
        mask |= flag
        break
    else
        mask &= ~ flag
        currentTraverseDepth -= 1
        currentNode = currentNode.parent
    end if
end while
until currentTraverseDepth <= 0
return -1
```

---

回溯到当前节点的父亲节点，而不再对于兄弟节点进行检查。如果当前层数所对应的标记为是0，就需要把

光线与当前节点的兄弟节点包围盒进行求交测试。当光线与其相交后，更新交点，并且更新对应的标记为1，从而防止当前节点被重复遍历。否则，直接回溯到KD树上一层。

根据上述算法，GPU内核可以执行与CPU迭代模型函数行为完全一致的遍历操作。从而达到了理论上遍历KD树的较优方式。这里的GPU遍历会更有优势一些，因为没有入栈和出栈等操作，相对来说指令操作要比CPU少很多。

#### 2.4. 去除噪音的后处理过程

由于光线跟踪算法的一些局限性，其生成的图片仍然会有一些噪音存在。其主要原因有以下几点：

- 很多三角形与光线几乎平行，所以导致出现病态方程。由于计算机内部浮点运算精度有限，所以导致这些三角形的求交结果很不精确。从而产生了不合理的法线以及反射折射光线等。
- 为了避免反射折射光线与当前三角形相交，本文的算法会把生成的光线原点沿着光线方向移动一个位移。从而避免一些错误的阴影和反射折射计算。然而在一些精度很高的模型中，这个位移经常会直接穿过一些原本与其相交的三角形，所以导致求出错误结果。

上述问题很难从根本上避免其误差，可以用多采样的方式来弥补这些损失。不过这会使算法的性能下降的非常快。

这里我们采用一种后处理的简单快速的方法去除噪音。对于每个像素，算法会检查周围其与周围四个像素的差值。如果这些差值的绝对值之和大于一定阈值，那么算法会认为当前像素是由于上述原因产生的噪音，就会对于这个像素进行一定的平滑处理。



图 2. 左图为未处理图片，右图为处理后图片

这种噪音处理方法有一定的局限性，可能会模糊一些物体的边缘。但是人的视觉对于噪音要比边缘敏感的多，所以对于大部分情况，其结果还是比未处理时更优。

### 3. 实验结果

本课题的所有实验数据在以下条件下测试的。

- CPU: AMD Athlon(tm) 64 X2 Dual Core 4400+ 23.0 GHz
- GPU: Nvidia Geforce GTX 285
- Memory: 4 GB DDR2
- Operating System: Windows Vista 32 bits

得到的数据显示，对于较简单的场景，本文所设计的算法相对于CPU可以达到70-80倍左右的加速比。并且其加速比是随着分辨率增加而严格增加的。

其中第三个场景有19万个三角形组成，并且带有大量的折射材质。该场景为光线跟踪的内存读取带来了很大的要求，并且使一个warp内部的分支现象非常严重。但是本文的算法仍然可以达到20多倍左右的加速比。

最后的场景中，含有87万个三角形面片，并且龙的模型上具有反射属性，从而导致大量的反射光线从KD树最密集处反射。GPU算法相对于CPU算法的加速依旧非常明显。

对于分辨率较小的图片，即使场景中的多边形数量非常多，本文的算法仍然可以达到交互式的性能。

### 4. 结论

本文介绍了CUDA实现的光线跟踪算法中的一些优化技巧。利用GPU内核中的虚拟栈实现了基于KD树的深度遍历搜索。充分优化了原有的CPU光线跟踪算法，达到了较高的加速比，达到了交互式的性能。

然而由于过多的全局内存访问，该算法的性能很难达到实时。并且还存在一些噪音问题，没有完全解决。CUDA给用户提供了图形interop的接口，但是由于一些工程上的原因，代码实现中没有利用这个功能。从而导致了其缓存两次在PCIE接口传输，这也导致了光栅化生成一级光线求交结果的优势没有特别明显。另外，由于MFC和GDI的效率较低，所以程序在交互渲染时，受到了很大的影响。

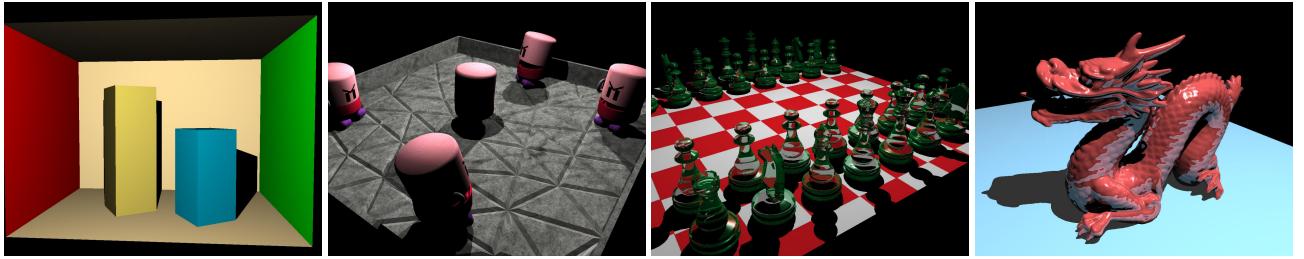


图 3. 我们的光线跟踪算法生成的图片

Resoultion	Scene1 ( 34个三角形)			Scene2 ( 11141个三角形)			Scene3 ( 193598个三角形)			Scene4 ( 872870 trianlges )		
	CPU	GPU	加速比	CPU	GPU	加速比	CPU	GPU	加速比	CPU	GPU	加速比
640 * 480	2316	32	72.37	5590	82	68.17	10059	411	24.47	6459	374	17.27
800 * 600	3551	49	72.47	8523	108	78.91	15054	539	27.92	10103	534	18.91
1024 * 768	5723	77	74.32	14106	188	75.03	24736	730	33.88	16213	848	19.19

图 4. 算法的性能对比 (单位: 毫秒)

## 参考文献

- [CHH02] N.A. Carr, J.D. Hall, and J.C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2002.
- [FS05] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware; 30-31 July 2005. 2005*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 2005.
- [Hav00] V. Havran. Heuristic ray shooting algorithms. *Unpublished doctoral dissertation, Czech Technical University in Prague*, 2000.
- [HSO07] M. Harris, S. Sengupta, and J.D. Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 3(39):851–876, 2007.
- [MB90] J.D. MacDonald and K.S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.
- [PBMH05] T.J. Purcell, I. Buck, W.R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*. ACM New York, NY, USA, 2005.
- [PGSS07] S. Popov, J. Gunther, H.P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. In *Computer Graphics Forum*, volume 26, pages 415–424. Blackwell Publishing Ltd, 2007.
- [ZHWG08] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. ACM New York, NY, USA, 2008.

## 5. 附录：代码编译

编译本文的算法源代码需要以下环境：

- CUDA SDK Release Notes Version 3.0 (32 bits) 或更高版本
- Microsoft DirectX SDK

算法实现在Windows Vista(32 bits)下开发，在Win7 (32 bits)和windows xp (32 bits)下进行过测试，均可正常运行。但是显卡需要GTX 260或者更高级的Nvidia显卡才可以正常运行程序，其他显卡未作测试。