Paper Reading

Intro:

在学习光线追踪的过程中，我们意识到在整个模拟的pipeline的过程中，最耗费时间的过程是检测光线是否与物体相交，我们知道，物体可以由很多面元来构建，因此我们在检测的时候，需要计算光线是否与这些面元相交，从而进行后续的计算。而对于精细的mesh，由于面元的数量相当庞大，相交检测将会相当耗时，因此我们需要加速这一过程。这一篇来自英伟达的paper就着重于加速这一过程，并达到了常数级别的时间。

Detail:

在assignment4中，我们采用了BVH结构来加速相交检测，相对于直接暴力遍历所有面元的方法，这种结构极大加速了这一过程。对于普通构建BVH的过程，我们采用二分的方法，构建二叉树，在非叶节点中，我们储存AABB包围盒，在叶节点中我们储存面元。在遍历的过程中，我们需要首先与每个节点的AABB做相交测试，若不相交，则停止，若相交，则对子节点做相交测试。若为叶子节点，则对叶子节点中的面元做相交测试。

由于这是must部分，大家都亲手实现过这一加速结构，相信绝大多数人都采用的递归的方式来实现对BVH二叉树的遍历。然而我们需要注意一个问题，递归其实是一个需要栈的遍历，虽然用在GPU上用栈的方式进行BVH很高效，然而采用栈需要考虑内存占用的问题，以及需要较高的GPU带宽。这篇文章在先前无栈遍历的基础上进行了进一步加速，优化主要分为一下三个方面。

1. important optimizations based on statistical evidence of what nodes are visited during backtracking 其在每个节点中储存了该节点的sibling, uncle, and grand uncle。
2. a perfect hashing scheme to map keys of nodes of a binary hierarchy to their addresses in memory 用完美哈希表将每个节点的key映射到内存的节点上。
3. perfect hashing’s parallel construction at moderate cost in time and memory 在gpu上并行构建哈希映射。

除此之外，作者还提到了两个优化的方法，一个是smart handling of disjoint traversal intervals，另一个是simple pausing and resuming of hierarchy traversal。

**第一方面：**

Paper提到了实现了无栈遍历，就是在当前节点不满足条件，需要向上进行回溯时，有栈的方法是将栈中的下一个元素pop出来，也就是需要测试的下一个节点，而其实我们可以直接找到其要测试的下一个节点，实现方法为：先在遍历树的每一层准备两个bit的空间：

一个叫做bit trail，一个叫做Cur Key，

1. bit trail 的作用，是用来表示在当前遍历的path中，在这一层上的遍历节点对应的sibling是否已经被遍历过。如下图中，假设39,18和8都已经被遍历过了，在Bit Trail中对应三个0。
2. 某一层的Cur Key则表示这一层上的遍历节点是否是index为偶数（0表示偶数）。例如38,19,9分别为偶，奇，奇，因此末尾对应0,1,1。

定义完这两个bit的数据，我们如何去使用他们呢？首先假设BVH的二叉树是一颗完全二叉树（在现实中其实并不成立，但我们会在下一节中解决这个问题），对于每一个节点我们可以按照顺序加编号，那么我们就可以根据节点的编号来进行backtracking，如果我们对编号为k的节点完成测试，需要进行backtracking时，这时只需要从后往前输出连续bit trail=0的次数M，N>>M即定位到了上一个节点，然后结合cur key选择精确的index。例如图6的例子中，38>>3=4，而往后数三层发现cur key=0，所以下一个该遍历的节点时4的sibling 5。

**第二方面：**

在第一节中我们提到我们先前是基于完全二叉树提出来的一种方案，然而在实际的BVH树中，我们并无法保证是完全二叉树，如果继续使用完全二叉树的index映射，会占用大量的memory。这篇paper采用了两个完美哈希表来解决这个问题。

我们首先来介绍一下完美哈希：

当关键字的集合是一个不变的静态集合（Static）时，我们可以使用完美哈希来获取出色的最坏情况性能。如果某一种哈希技术在进行查找时，其最坏情况的内存访问次数是O(1) 时，则称其为完美哈希（Perfect Hashing）。设计完美哈希的基本思想是利用两级的哈希策略，而每一级上都使用全域哈希（Universal Hashing）。其第一层结构类似chained hash，但其不以链表的形式储存，而是储存一个新的哈希表， 这个哈希表的大小要开到n方以防止发生哈希冲突。也就是说完美哈希是一种完美的映射，不会产生哈希碰撞，因此访问时间也就是O(1)的。

通过采用一个完美哈希映射，就可以继续采用之前的node编号策略，从而减少memory的占用。除此之外，由于我们对node检测完之后，我们还需要找到下一个node，我们通过在建立一个哈希表，储存其对应的index，在计算出index后，我们就可以常数级访问下一个节点。文章中还提到了一个小trick，在选择mod的常数D时，考虑选择一个2的指数，这样的好处是可以大大降低取模运算的cost，因为以2的指数取模时，就相当于和D-1作位运算。从而降低了计算的cost。

**第三方面：**

文中也提到了构建哈希表的过程是可以并行化的：其构建的

构建哈希表也有一定的缺点：首先由于构建出来的完美哈希并不是最小哈希，也就是哈希表中还有剩余的空间，整个BVH所占的空间会增大。其次由于叶节点也需要被进行哈希，而它的大小并不和内点相等也不是一个固定的常数，因此也是一个缺点。

**第四方面：**

接下来讲一下文中提到的一个disjoint的问题，在判断光线是否与场景包围盒相交的过程中，由于包围盒之间存在overlap，所以即使已经计算出光线与包围盒的第一个交点后，仍然需要继续往下找。毕竟我们是不能保证一旦一个node是postponed的，这个节点下所有的孩子都会在这个交点之后的。但是这一步其实是可以优化的，如果一条光线对两个包围盒的交点是disjoint的，那么就可以保证以postponed node为根的树是完全在先处理的那个node的后面，这样我们就可以避免继续处理postponed的node，因此我们需要去对每条光线检查其对两个boundingbox相交是否是disjoint的。他的方法是再加一个对树的每一层再加一个bit，来表示这一层不是disjoint

Disjoint t-Interval Mask

即使我们的遍历是沿着光线有序进行的， 在找到第一个interaction point之后，遍历仍要继续，这是因为在左右儿子overlap的地方，我们无法确定谁的mesh会更先和ray发生interaction

左右儿子AABB与ray发生interaction的区间[t0,t1]有交集，也就是说他们overlapping了，

第二种情况

虽然ray和左右儿子的AABB都相交了，但他们的相交区间[t0,t1]是disjoint的，所以可以减少postponed nodes的数量。

我们可以把左右儿子中t0较大的那一个进行剪枝。

（第二张图）

图上颜色越深的地方表示做相交检测的数量较少，右图下面这一块明显可以看出颜色要深一些，也就意味着比较有效的减少了无用的相交检测。

**第五方面：**

接下来讲一下pause和resume的问题，我们在学习path tracing的时候，我们需要在找到交点之后，生成新的ray之后重新去做遍历测试。这就会涉及到遍历的暂停和继续执行的问题。除非ray的方向变化，否则我们就可以将之前的bit trail储存下来，来避免再次遍历先前遍历过的节点。然而这种方法只适用于连续光路，光线方向不改变的情况，对于path tracing的这种情况由于光线方向随机，所以在path tracing中是无效的

Conclusion:

这篇来自英伟达的paper，出发于BVH递归遍历耗费内存庞大需要较高带宽的问题，在先前无栈遍历BVH的基础上，做了进一步的加速工作，并用完美哈希表使backtrack的过程达到了常数级别的时间，同时也有很多小的trick来完善加速。其paper中的对比实验也证明了其性能相比之前的方法有了较大提升。我们的分享到这里结束，谢谢大家。