图1 我们提出一种新算法来处理实时应用中的多光源.我们的算法建立在“光树[light tree]”的基础上,它使GPU能够有效处理各种不同的光源分布.从左至右,我们显示了着色场景,使用“practical clustered shading”并在右侧使用我们的算法对每个图块评估的光源数量.与现有技术相比,我们的算法显着减少了必须处理的光源数量,尤其是在场景的较远区域.

概览

实时处理许多光源仍然是实时图形中的重大挑战之一[Sousa and Geffroy 2016; Garawany 2016].即使是最新的方法(如practical clustered shading),仍然存在各种性能低下的问题案例.特别是在深度变化较大的场景中,现有算法无法正确适应光源的分布,最终会计算许多对最终图像没有帮助的光.

我们提出了一种新方法,即“帖块光树[tiled light trees]”——一种适应于光源分布的分层加速结构.我们的方法改进了现有解决方案的最坏情况下的性能.由于遍历开销,所提出的算法有时可能比群簇着色[clustered shading]慢.为了最佳地处理这些情况,我们提出了一种混合方法,该方法将光树的优势与群簇着色相结合,几乎在任何情况下都优于单独的解决方案,我们的新混合算法易于实现,适合在游戏等实时应用中使用.

1 介绍

帖块着色[tiled shading]及其更现代的变体(例如tiled deferred shading)通过将屏幕空间中的灯光进行装箱[binning],解决了多光源为场景照明的问题.这极大地减少了每个帖块需要处理的光源数量,从而使游戏可以有效使用数百至数千个光源.

帖块着色算法的一般方法是将屏幕拆分为矩形帖块,并使每个光源与帖块的锥体相交. 它使用屏幕空间的帖块范围以及帖块的最小/最大深度来创建一个视锥,以测试所有灯光[Andersson 2009]的相交与否.这样的算法存在一个问题,由于二维性质，每块视椎体通常具有很大的深度范围，并且可以与许多光源相交，而这些光源不会影响帖块内的任何几何形状. 这导致了2.5D算法的引入,该算法试图减少误报的数量.关键思想是不仅要在屏幕空间内细分,而且还要沿视角深度方向细分.通常使用视锥体对齐的网格并将光分类到其中，由于边界更严格，因此减少了光/像素测试的次数[Persson 2013].所使用的数据结构高度统一，以实现高效分配.

我们的贡献是一种新颖的方法,它引入了“光树”作为光剔除的新的空间加速结构.我们的方法特别适合处理许多不重叠的光源,这对于当前算法是具有挑战性的.与以前的技术类似,我们的光源树建立在CPU上,并且不需要CPU/GPU同步,因为它们与几何形状无关.我们还提出了一种使用2.5D网格的混合方法,其中每个单元格包含一个列表或一棵树.它结合了光树结构的优势和网格单元的低开销列表遍历功能,其中大多数光源覆盖整个单元.使用此组合算法,我们能够提高2.5D照明算法的鲁棒性,同时保留使其流行的所有属性.

2 之前工作

[Andersson 2009;Ferrier和Coffin 2011年;原田等人2012],这是第一个引入帖块照明的论文.在帖块照明中,屏幕空间可细分为帖块.第一步,将光源平行地拣选到每个帖块,最后评估与帖块相交的每个光源.从那时起,许多游戏引擎都使用了基于帖块的延迟照明的变体.[Harada 2012]引入了2.5D算法,该算法可拆分深度范围并为每个单元分配灯光.该算法的主要优点是可以快速跳过不与场景几何图形相交的灯光，因为每个阴影像素仅处理与其网格单元相交的灯光.该算法已用于具有多种调整和改进的多个游戏中.通常,各个变体之间的主要区别在于深度的细分方式以及光/帖块相交测试.在所有情况下,目标都是减少错误测试的灯的数量.

该领域的最新进展是群簇着色[Olsson等人2012].群簇着色对单个着色点进行重新排序,以提高光剔除效率.通过处理几何样本群簇,它可以针对单个光源执行背面剔除.它使用构建在场景中所有光源上的全局BVH.对于每个群簇,都需要遍历该树.尽管与我们的工作类似，但我们不需要灯光的全局数据结构,而应使用两级层次结构.由于我们的树很小,因此可以提高性能,从而实现更好的遍历.

[Persson 2013]提出了群簇着色的生产变体.与原始的群簇着色相比,新算法更简单,并且仅对光源本身而不是群簇.灯光被分配到相机空间中的2.5D网格,该网格沿z轴以指数级递增细分。这样可以使每个网格单元的大小在屏幕空间中大致保持不变，并防止遥远的单元由于投影而变成像素大小.该技术的最新更新包括使用保守光栅化技术在GPU上进行更高效的光源分配[Örtegren2015],这也可以与我们的方法一起使用.

3 我们的方法

我们将首先介绍“光树”，因为这是我们算法背后的核心思想.然后,我们描述一种将混合方法将其与群簇着色相结合.

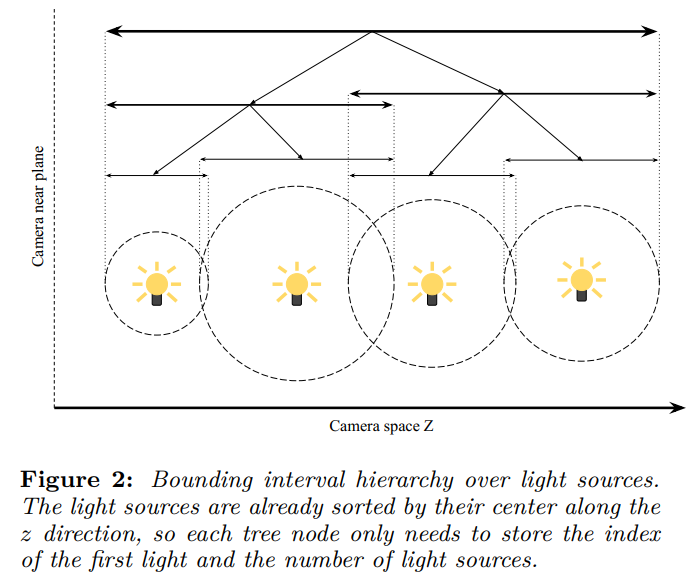
在所有情况下,我们的技术在每一帧都同时使用CPU和GPU.光源始终在CPU上进行预处理,然后上传到GPU,最后用于照明.在CPU,我们执行以下步骤:首先,使用视椎体对所有光源作剔除,然后根据与相机平面的距离对它们进行排序.接下来,我们将光源分为屏幕空间帖块，最后,我们为每个屏幕空间帖块构建光树.

在GPU上,我们确定覆盖当前像素的光树并在光照过程中遍历它.请注意,我们的技术仅需要有关相机和光源的信息,而无需场景几何图形,因此,可以在正向和延迟着色上下文中使用它.不需要几何数据的另一个优点是,可以在渲染G-buffer或阴影贴图的同时准备光树.

我们算法的关键思想是为每个2D屏幕帖块使用一棵光树.光树可以完美地适应深度光源的分布，因为它们不限于沿z轴的固定网格.由于树的局部性质-每个帖块都有自己独特的树-我们还可以生成具有高相干性的浅树[shallow trees],而不是为所有光源生成全局树[Olsson等人2012].小而浅的树非常适合依赖非常宽的SIMD单元执行并需要针对一致性和内存访问进行优化以实现良好性能的GPU.在以下部分中,我们将描述光树的数据结构.

3.1 光树结构

我们使用光源在相机空间中的深度范围构建了简化的一维边界间隔树[Cormen等人2009],如图2所示.这假设每个光源的影响半径都是有限的.该结构是一个二叉树,其中的每个节点都包含一个灯光列表和聚合深度范围的.我们总是生成完整的二叉树,将叶节点的数量四舍五入到下一个2的幂.使用完整的二叉树使我们能够简化遍历并使其更加连贯.



如前所述,执行一致性不足以确保良好的GPU性能-我们还需要高效的内存访问模式.为此,使用深度优先顺序将树布置在内存中.这允许使用无堆栈算法在GPU上快速遍历,并提高了缓存效率.

遍历严格按照与存储树相同的顺序进行.也就是说，组中的各个线程仅在树中向前跳转.如果必须访问所有节点,则将线性遍历内存,从而获得最佳的内存访问模式.如果读取稀疏(表示树遍历具有很高的一致性),则执行将跳过连续的节点,但永远不会跳回树中.因此,遍历总是高效的,因为作为缓存行的一部分加载的树节点已被完全处理或被跳过.

3.2 树的构造

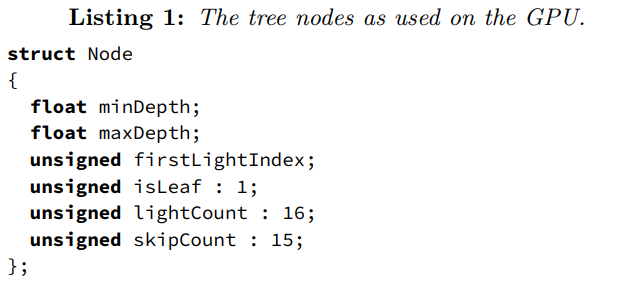
在开始构建树之前，我们首先使用平行基数排序[parallel radix sort]按光源到相机平面的距离对全局光源列表进行排序.这很重要,因为它允许我们按照有效的树构建所需的顺序将光分配给图块,从而避免了分块排序步骤.我们发现,当有很多灯与多个单元重叠时,全局排序比每块排序便宜.一旦获得了单个图块的光源列表,便可以独立处理每个图块并构建二叉树.

我们使用自下而上的树构建算法,该算法以广度优先的节点内存布局构建二叉树.利用所有光源已经按照它们到相机平面的距离进行分类这一事实.通过遍历上一级别并组合每对连续的节点,可以完成树的每个级别的构建.整个树是在时间中构造的.

对于GPU,我们想要一棵深度优先的树,因为我们希望将连续的节点彼此相邻地存储在内存中.因此,我们需要将树从广度优先顺序转换为深度优先顺序.由于叶节点的数量始终是2的幂,并且通常很小,因此我们预先计算了一些转换表,因此可以通过一次遍历数据就将广度优先树转换为深度优先树.

另一种方法是简单地将树自上而下构建.这需要更昂贵的构建算法,该算法需要计算每个级别的每个节点的边界.整个构建需要时间.有利的一面是,它可以允许对分割位置进行最佳选择,并直接生成深度优先的树.

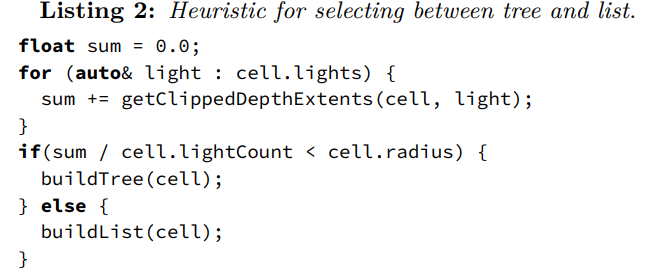
在将树上传到GPU之前,我们将其打包成清单1中所示的紧凑表示形式.请注意,我们还在每个节点上存储了一个跳过计数,这使我们可以遍历以深度优先顺序排列的任何二叉树(不仅是完整的二叉树)[Smits 1998].我们将跳过计数,光计数和叶子标志存储为32位整数. 跳过计数使用15位存储.我们使用一位来指示节点是否为叶.剩余的16位用于灯光数量.这使我们每个帖块最多可存储个光源.我们为每个叶节点指定了几个光源,因此树中包含的节点通常比光源少得多.

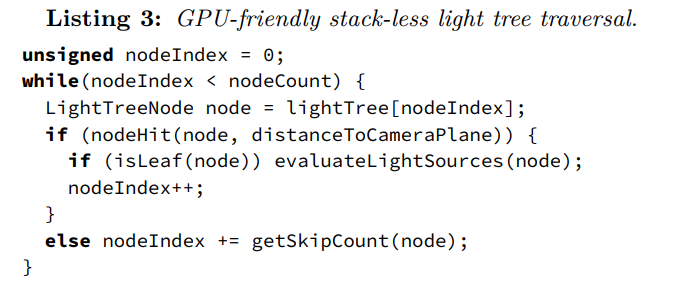


3.3 混合方法

光树本身提供了高度自适应的数据结构,并且通常比“实用的群簇着色”方法提高了GPU的照明性能.不幸的是,仍然存在着在某些视角条件下光树比群簇着色要慢一些的情况.在这些情况下,光树仅存储光源列表并对其进行迭代,就会增加少量但可观的开销.因此,我们将光树与2.5D网格结合起来,并为每个网格单元选择最佳算法.组合后的算法如下所示:与以前一样，我们先从光源的视椎体剔除开始,然后按照到相机平面的距离进行分类.主要变化在接下来的两个步骤中.我们首先将光源分配给视锥对齐的网格单元.最后，我们确定昂贵的网格单元,并为这些单元构建光树.

我们细分深度类似于[Persson 2013],并确定每个网格是否要存储简单的灯光列表或使用灯光树.这就需要试探法在两个选项之间进行选择.我们使用单元格与分配给它的灯光之间在z轴上的平均重叠量来确定列表与树之间的距离.当平均重叠比单元格的范围小时,单元格可能包含大量的空白空间,可以通过使用浅色树有效地跳过这些空白空间.当平均重叠接近像元范围时,表明有很多光线覆盖了像元的大部分深度,应使用列表.使用平均重叠(光范围裁剪到单元格范围)而不是平均光直径会使启发式方法在混合大小光源的场景中更加稳健.最终结果是,在有许多重叠大灯光的情况下,我们的算法可以适当地回退到[Persson 2013]，这对于该情况是最佳的.对于许多小光源,仅会使用光树,而对于包含大光源和小光源的实际分布,我们的启发式方法会选择最佳技术.小灯的替代方法以及在不同技术之间切换的能力,是使我们的算法为任何光分布提供最佳性能的关键原因之一.





我们的混合算法必须结合树遍历和列表迭代才能获得最佳性能.由于我们的树遍历代码已经允许每个叶节点使用可变数量的光源，因此我们选择了以下方法来将两者结合：如果单元格包含项目列表,我们将存储一棵单节点树,其中根也是叶.在这种情况下,线程将仅在各个光源上进行迭代,唯一的开销是相关的获取和位测试.

请注意,对于单个帖块而言,这并不是一个统一的决定–一个帖块中的线程可能会同时命中许多不同的单元,并且必须同时遍历列表和树.我们使用内在指令对各种情况进行了优化，将遍历分为三种不同的情况：a）所有线程遍历一个列表，b）所有线程遍历一棵树，c）存在分歧,并且遍历列表和树.

执行哪种情况主要取决于场景中的光线分布.具有许多重叠光源的图块将沿着列表路径; 剩下的大部分帖块将遍历树.遍历列表和树仅适用于深度变化较大的图块(另请参见图3).

对于遍历树的情况,我们已经进行了优化,将树加载到本地内存（LDS）中以改善主内存访问模式.仅在子组中的所有线程都遍历同一棵树的情况下才使用此优化,这是常见的情况.

4 结果

我们已经使用Vulkan实现了我们的算法，并在AMD RX480和NVIDIA GTX 980上评估了性能。我们已将我们的算法与“实用的聚类阴影”进行了比较[Persson 2013]。对于每个场景，我们根据场景范围设置了平截头体的最大深度范围。这是我们完成的唯一特定于场景的设置。虽然光树是完全自适应的，不需要任何特定于场景的配置，但是“实用聚类阴影”算法的性能对深度分区，光源大小和分布高度敏感。 [Persson 2013]描述了一种分区方案，具有15个从5到500米的指数分布深度切片和一个特殊的“近”切片。我们的基准测试使用类似的方案（清单4），仅对最大深度进行了少量修改。相同的网格配置用于混合模式和集群模式。即使我们的混合算法可以适当地降级为两种情况，我们对于每个实现都有单独的代码路径，以使每个实现具有最少的着色器。

我们选择了许多游戏中常见的轻量级分布； 通常会有大量的小灯，例如路灯，手电筒，汽车的头灯和尾灯，但是只有少数几个大光源，尤其是在屏幕上占很大比重。 我们试图通过使用较浅的尺寸分布来模拟这一点，从而导致较低的每像素透支； 也就是说，只有很少的光源会影响每个像素。

4.1 预处理

我们的算法要求在CPU上每帧进行预处理-包括光视椎体剔除,光分类,光装箱以及树的创建.到目前为止,光装箱是最昂贵的步骤,占总CPU时间的50％,其次是树创建（25％），排序（12％），数据上传到GPU（7％）和平截头剔除（3％）.对于不同的光量,不同的预处理步骤的相对成本是一致的.

我们的分箱算法使用2D边界[Mara and McGuire 2013]和深度范围为每个光源计算一个与视锥对齐的网格边界框.边界框用于将光散射到网格单元中，这占用了此步骤的大部分CPU时间。群集着色算法也需要合并，该合并占CPU成本的85％。

我们使用并行模式库[Microsoft 2016]并行化了树的创建，该库提供了并行排序以及循环和其他构造的通用并行。无需锁定或同步即可处理所有树。这是可能的，因为每个树都是完全独立的，并且所有必要的内存都预先分配有来自合并步骤的信息。我们尚未执行任何其他低级CPU优化，例如使用SIMD内在函数或ISPC进行矢量化[Intel 2011]。我们期望通过使用SIMD指令，可以大大加快分仓，平截锥体剔除和部分树木建造的速度。即使没有进一步的优化，对于实时应用而言，除了最高的光计数之外，CPU成本还是可以接受的（见图4）。