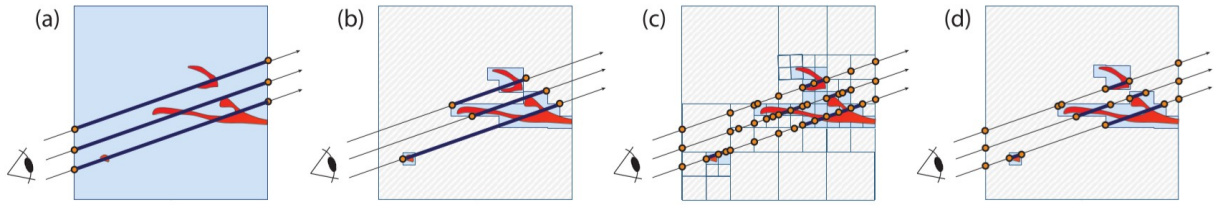
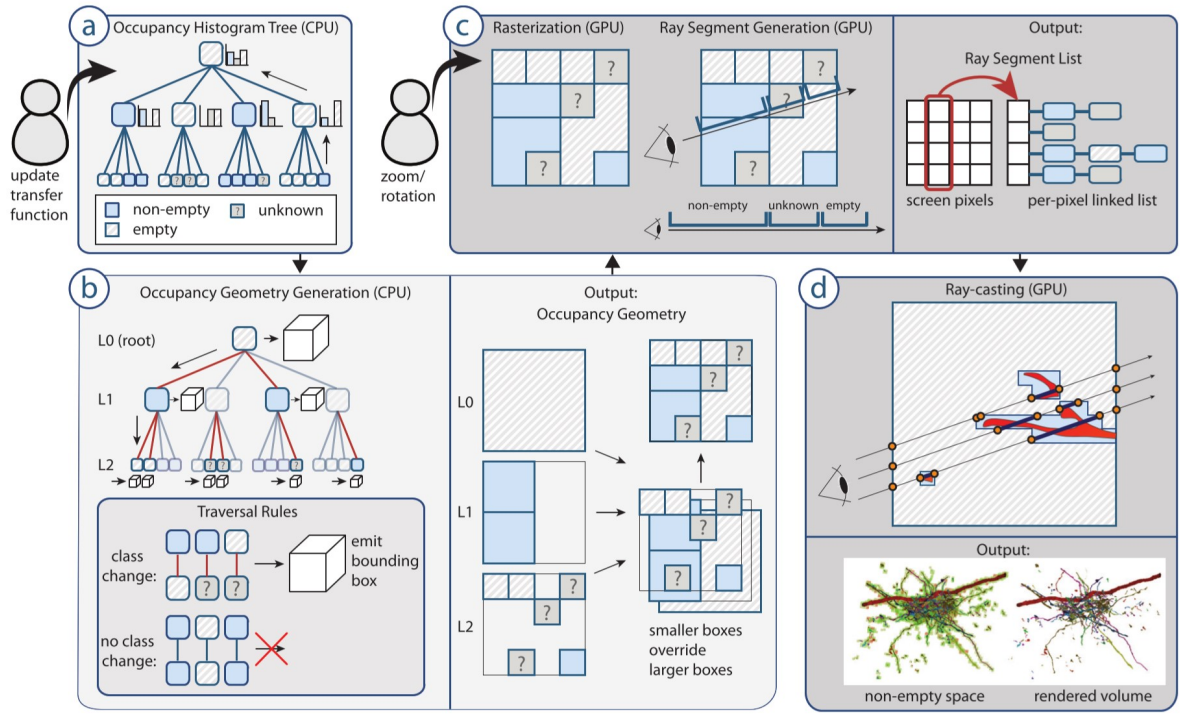
SparseLeap

这是一种高效率的跳过空体素的优化方法，用于优化大体积的体渲染效率。



- 图(a) 为直接渲染。
- 图(b) 跳过最前面和最后面的渲染。
- 图(c) 八叉树优化渲染。
- 图(d) SparseLeap方法渲染。

算法流程



构建八叉树

首先我们对我们的体数据建立一棵八叉树，论文中称之为*Occupancy Histogram Tree*，其相比普通八叉树的区别是，每个结点都有一个类型，我们称之为*Occupancy Class*，且每

个节点都记录其子树下每个类型的**叶子节点**数量。且，每个非叶子节点的类型则为，子树中哪种类型的叶子节点最多，那么这个结点就为那个类型。

对于*Occupancy Class*，其有三个类型。

- `Empty`：这个结点所表示的范围是空的。
- `NoEmpty`：这个结点所表示的范围是非空的。
- `Unknown`：这个结点所表示的范围不确定，用于延迟更新。

确立包围盒排列顺序

然后对于一个给定的视点(**View Point**)，对八叉树进行深度优先搜索，对于每个结点的儿子来说，其遍历顺序由视点决定，保证访问到一个结点的时候，能够保证能够遮挡住这个结点的结点已经被访问过了[1]，这整个过程中，我们需要得到了一个深度优先搜索序列(**dfs序**)，这将决定我们生成RaySegment List的时候，光栅化包围盒的顺序。要求满足下面的条件：

- 对于每个包围盒，优先渲染大的包围盒的正面，然后访问在其范围内的小包围盒，结束后渲染大的包围盒背面。
- 对于拥有同一个父亲(在八叉树中)的包围盒，要优先渲染距离视点近的包围盒，然后渲染远的。

那么很显然的一个方法就是，进行深度优先搜索的时候，进入一个结点时，给计数器增加一，然后在结束一个结点的时候，给计数器增加一。那么每个结点都可以得到两个数，这两个数就是这个结点代表的包围盒的正面和背面渲染的顺序。

生成需要光栅化的包围盒组

然后我们准备生成**Occupancy Geometry**，准确来说他是一组按照特定顺序排列的包围盒。首先我们按照如下规则生成这一组包围盒：

- 根节点始终加入进去。
- 对于任意结点，当他的类型和父亲类型不相同的时候，将这个结点代表的包围盒加入进去(注意还要记录这个结点父亲的包围盒类型)。

得到一组包围盒后，我们还需要为其制定顺序，这个时候我们之前生成的序列就有用了。我们可以根据序列进行排序，最后得到一个生成RaySegment List正确顺序的包围盒。

对包围盒进行光栅化

得到包围盒后，我们按照顺序对其进行光栅化，并在像素着色器中生成RaySegment List[2]。

如何在像素着色器中生成RaySegment List，我们定义一组事件，表示从视点的光线与包围盒相交的时候发生。其记录如下信息：

- 深度，记录这个相交的点和视点的距离[3]。
- 事件类型，分别为进入事件(Entry)和离开事件(Exit)[4]。
- 类型，即**Occupancy Class**，对于进入事件来说，其类型为包围盒所表示的结点的类型，对于离开事件来说，其类型为包围盒所表示的结点的父亲的类型。

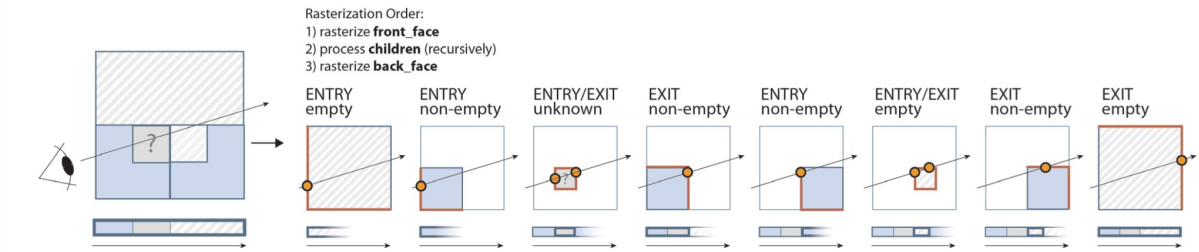


Fig. 5. ***Ray events and ray segment list generation.*** Ray segment lists are generated by rasterizing *nested* occupancy geometry bounding boxes. The rasterization of their front- and back-faces, respectively, results in *ray events* at the corresponding intersection positions: *entry* events for front faces, and *exit* events for back faces. After all ray events have been rasterized (right), including possibly merging or deleting events, the result is the final ray segment list (left): a sequence of ray segments (intervals between consecutive ray events), with one occupancy class per segment: *empty*, *non-empty*, or *unknown*. Left: Occupancy geometry and corresponding output ray segment list. Right: Step-by-step generation.

事件的合并和删除

对于我们生成的事件来说，有很多是重复的事件。我们可以将其剔除，以优化性能。我们对于深度值同样的事件，可以遵循如下规则进行合并或删除。

- 两个事件的类型是一样的，那么保留后面的一个事件。
- 两个事件，第一个事件是离开，第二个事件是进入，那么第一个事件之前的那个事件和第二个事件的类型是相同的话，两个事件都可以删除。
- 两个事件，第一个是进入，第二个是离开的话，那么可以都直接删除，这意味着刚刚擦边过去。

Cache Misses

算法提及了两种类型的Misses。分别是 `Occupancy Misses` 和 `Data Cache Misses` 。

这两种Misses都是为了适用于大数据从而出现的。

- `Occupancy Misses`：对于RaySegmentList里面一段 `Unknown` 内容，我们在 `shader` 内是直接当作和 `NoEmpty` 一样处理的，但是有一点不同的是，我们可以告诉我的程序这一段是 `Unknown` ，然后让程序在八叉树去确定这部分到底是什么类型。这样的优

势是可以在刚开始的时候减少处理量，不需要直接将所有内容都去处理(树刚开始就只有一个根节点)，而在渲染的时候我们可以汇报 `Misses` 然后让程序在渲染结束的时候再去处理。这样对于完全不可视的物体，根本不会处理到，或者延迟处理。

- `Data Cache Misses`：主要是因为大数据的话，无法将整个数据存在内存中，因此要考虑当前需要的数据是否在内存中。

对于第一种 `Misses` 可以在 `shader` 用一个缓存来记录哪些块(被八叉树划分后的块)需要确定去加载，然后在 `CPU` 中读取后去处理。

对于第二种，还未有方案处理。

括号内容

- $[1]$: 由于需要考虑的是不重叠的八个体积一样的包围盒，因此我直接计算包围盒中点到视点的距离来作为排序依据，可能还有其他做法需要考虑。
- $[2]$: 着色器中生成RaySegment List，我使用了几个纹理来存储数据，一个2D纹理存储每个像素对应的List大小，然后三个3D纹理存储事件对应的属性。并且需要注意的是，必须限制同一个位置的像素，最多同时运行一个着色器程序，不然会导致读写冲突。由于目前没有找到适用于HLSL的方法来限制，因此无法使用实例技术。
- $[3]$: 对于深度计算，目前我是使用的点到摄像机的距离作为深度。
- $[4]$: 着色器中可以获取当前像素是正面还是背面，因此我们只需要判断这个像素所在的包围盒是需要的面和当前像素所在的面是否相同，就可以确定这个像素是否合法，并且可以确定事件到底是进入事件还是离开事件。

一些优化

节点规模优化

- 八叉树应该不需要建立完整的版本，即对于一个结点如果他子树中的叶子节点的类型都是一样的话，那么完全就可以不建立子树了，这里我们可以在插入结点的时候，回溯的时候判断是否需要删除，如果需要那么就删除。

虚树优化

- 对于非大数据来说，当摄像机的位置改变的时候，我们需要重新生成RaySegmentList，重新遍历树以获取最新的顺序。但是直接遍历树的开销是非常大的，考虑数据范围是直接存储在内存中的，因此树在构建的时候就可以建立好，而不需要考虑Cache Miss，那么哪些包围盒需要提取出来是确定的。因此我们对树建立一棵虚树，即只存储需要的包围盒所在的结点以及一些连接节点(通常是LCA)，那

么对于位置的更新，我们只需要遍历虚树(规模可以小很多，主要看需要的包围盒数量)就可以重新生成顺序，而不需要遍历整棵树。

关于实现

关于适应大数据的两种 `Cache Misses` 都没有去实现。 关于生成RaySegmentList的光栅化部份，目前处于低效状态，因为没有办法使用实例技术减少绘制调用。文章中使用了ARB拓展来进行锁定，防止读写冲突。

Direct3D Version

- 使用了RWTexture属于UAV类型，需要将其和RTV绑定在一起。
- 请不要在 `Shader` 里面使用循环展开，不然编译时间很长。
- 存储RaySegmentList的三维纹理不要开得太大，不然调试很难，会超过显存使用量。

Vulkan Version

待填。

算法理论解释

待填。

引用

- SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering
- Unraveling The Connectome: Visualizing and Abstracting Large-Scale Connectomics Data