摘要

本文提出了一种使用硬件遮挡查询进行遮挡剔除的有效算法.根据来自多个先前帧的查询结果来优化查询数量.在最近的帧中发现未被遮挡的场景部分,测试频率较低,因此减少了每帧执行的查询数量.该算法适用于任何类型的场景,包括带有运动对象的场景.该算法利用包含场景中对象的树结构.

1 介绍

虚拟环境中的细节数量仍在增加,并且需要使用“聪明的”算法来显示场景.简单的蛮力渲染复杂场景的方法无法实现交互式帧频.因此,必须使用执行遮挡剔除的算法.这样的算法能够从用户的角度检测被其它物体遮挡的物体,并迅速丢弃这些隐藏的物体以进行进一步处理.

有许多执行遮挡剔除的方法(有关更多详细信息,请参见调查[Coh03a]).近年来,开始使用基于硬件的遮挡查询.该查询允许程序员间接访问Z-buffer,并测试物体是否可见或是否已被已渲染的物体遮挡.被测区域通常只是一个完全详细对象的边界框.基于查询的结果,应用程序可以决定是否渲染整个对象.

尽管遮挡查询功能很简单,但正确使用它来获得显着的性能提升并非易事.已经开发了几种使用遮挡查询的算法.

第一个是[Hil02a].场景分为网格，网格中的每个单元格都包含与之相交的物体列表.渲染框架时,网格按前后顺序由图层处理.对于每个单元格,查询其边界框的可见性,如果该框可见,则呈现该单元格列表中的对象(如果之前未呈现,因为它们与另一个已处理的单元格相交).

[Hey01a]中描述了另一种方法.与以前的方法相反,此算法在屏幕空间中工作.屏幕分为规则的低分辨率网格,其中每个单元格都有其自己的状态.它可以被遮挡,未被遮挡或未知.渲染开始时,每个单元的状态都设置为不遮挡.场景中的对象按从前到后的顺序处理.每个对象都投影在屏幕上,并且测试相交单元的状态.如果所有单元格都被遮挡,则不会渲染该对象.如果存在一些未被遮挡的单元格,则呈现该对象.如果有些单元格的状态未知,则使用遮挡查询测试其真实状态，并在结果可用时做出决定.渲染对象后,相交单元的状态更改为未知.由于遮挡查询不会在对象渲染后立即发出,而是仅在确实需要时才发出,因此该方法称为“延迟遮挡剔除”.

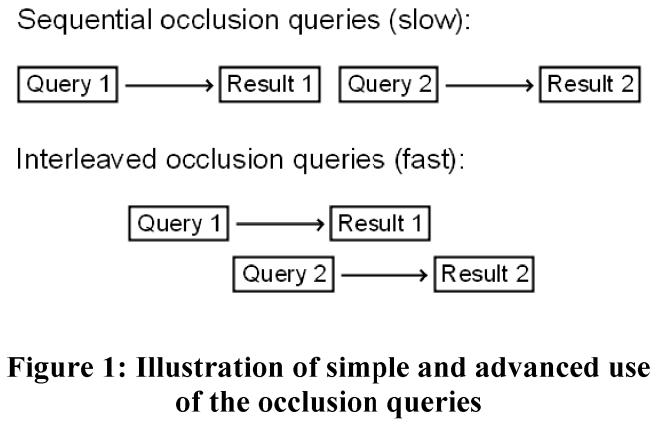
基于重要性的质量折衷算法使用遮挡查询,通过省略通常是可见的但对最终图像没有很大贡献的对象来加快渲染速度.场景再次被网格划分.该算法维护单元的优先级队列,并从具有最高优先级的单元开始对其进行处理.每次访问单元并发现其可见时,都会对其进行渲染,并将其未访问的邻居添加到队列中.这样可以渲染所有可见的单元格,但是也可以在没有足够的时间渲染整个帧时停止渲染.某些对象将不会被渲染,但是将是那些优先级最低的对象,并且不会对最终图像产生太大影响.

我们的方法使用的方案类似于[Bit04a]中最近描述的方案.整个场景以树状结构组织.在渲染期间,遍历节点,并使用其边界框上的遮挡查询测试其可见性.如果节点可见,则呈现其内容,否则将跳过.为了减少查询数量,并不是每帧都要测试所有节点.取而代之的是,使用一些启发式方法来检测可能可见的节点,并且算法无需发出查询即可渲染此类节点.

2 遮挡查询

遮挡查询是现代图形显卡中存在的一种硬件功能.原理很简单:将一部分场景渲染到屏幕上并渲染到Zbuffer后,将要渲染一些复杂的对象.并不是直接渲染它们,而是选择测试对象是否实际可见.通过检索对象的边界框并对其应用遮挡查询来执行此测试.如果渲染了该框,则遮挡查询将返回可见的像素数.这是通过将框与存储的Zbuffer值进行比较来完成的.如果可能的可见像素数等于零,则边框将被先前渲染的对象隐藏,并且无需渲染复杂的对象.

不幸的是,使用遮挡查询功能并不是那么简单.由于缓冲了发送到图形卡的数据,因此经常会在发出查询时未渲染场景的先前部分.但是,遮挡查询可能会异步处理.可以启动查询,然后渲染某些对象并在以后可用时使用查询结果.同样,不必等待下一个查询,直到上一个查询完成为止-查询可以同时运行.这些过程如图1所示:



3 算法

我们的算法要求将场景以树状结构分层组织.在我们的实验中,我们使用了与轴对齐的BSP树,但也可以使用octree，kD树或类似结构.场景中的每个对象都恰好放置在一个节点中,该节点完全且尽可能紧密地包围了该对象.

渲染帧时,算法会设置一个队列,其中包含要处理的节点.最初,它仅包含根节点.队列以自然顺序进行处理,算法会为每个节点决定是否丢弃,渲染节点中的对象而不使用遮挡查询或遮挡查询.

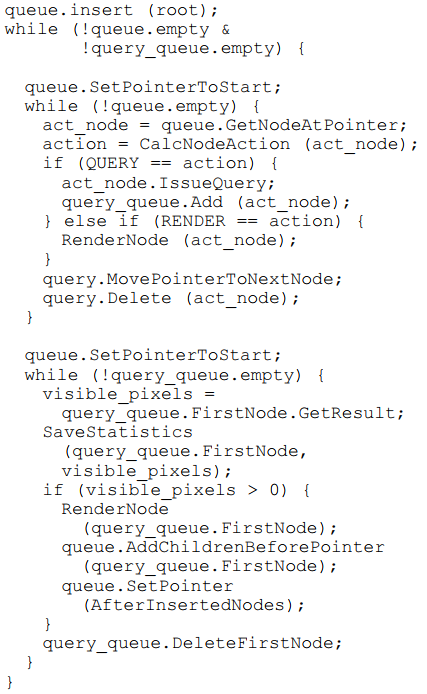
第一种情况很简单.如果节点被丢弃(例如,由于截锥体剔除),则将其从队列中删除,算法将移至队列中的下一个节点.

第二种情况稍微复杂一些.对于某些节点,该算法可能会决定不需要遮挡查询(该决定过程将在后面描述).存储在此类节点中的对象将立即呈现,并将该节点从队列中删除.它的后代放置在队列中已删除节点的位置.新插入的节点按从前到后的顺序排序.然后,算法从第一个后代继续.

第三种情况最复杂.如果没有有关最近遮挡查询结果的足够信息,则很难预测是否应渲染节点中的对象.在这种情况下,发出查询.一段时间后,查询结果将可用.可以等待查询完成,但是这会浪费时间来处理其他节点.因此,该算法开始处理队列中的下一个节点,而不是等待结果.

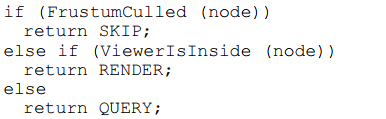
查询完成后,根据结果,可以跳过该节点,也可以渲染该节点中的对象,然后将队列中的节点替换为其后代.由于新插入的节点位于当前处理的节点之前,因此算法有时必须返回并再次传递队列.它可以随时停止处理队列并返回到队列的开头.通常,当查询数量超过某个阈值（约20个）并且很可能第一个查询已经完成时.在第一个查询的结果可用时,可能会停止处理队列并准确返回,但这将需要对查询的状态进行额外的检查,这非常耗时.

实际的实现使用两个队列–一个是上述主查询,另一个是带有发出了阻塞查询但尚未完成的节点的队列.这是伪代码的概述:



CalcNodeAction函数对于算法至关重要.它以一个节点为参数并返回该值,该值通知算法的其余部分应对给定节点采取哪些操作.这些动作是:

* 渲染. 在不发出查询的情况下渲染节点中的对象.
* 跳过. 该节点不可见,该节点中的对象将不被渲染.
* 查询. 将发出遮挡查询以确定该节点是否可见.

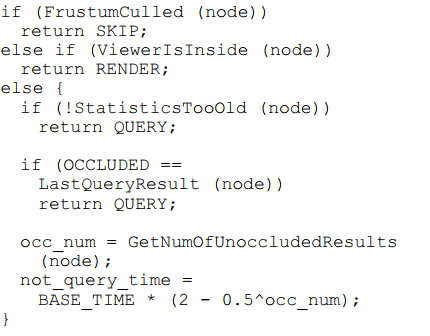
 这是CalcNodeAction函数的简单版本的伪代码.此版本不使用先前的occlsuion查询的任何结果.

优化

CalcNodeFunction可以进行估计(基于最近的遮挡查询的结果)并将返回值从QUERY更改为RENDER.这种估计必须谨慎进行,否则我们最终可能会渲染许多实际上被遮挡的对象.另一方面,我们不想使用很多遮挡查询,因为这可能会严重降低性能.

该算法为每个节点存储了几个最近的遮挡查询的结果,并使用它们来确定是否发起遮挡查询.发现该节点可见的次数越多,发出查询以检查该节点是否仍然可见的次数就越少.

优化的CalcNodeAction函数的伪代码如下:



BASE\_TIME常数取决于给定应用程序中查看者的移动速度.对于较高的速度,我们选择较低的数字.在我们的测试中,该常数等于2/3秒.