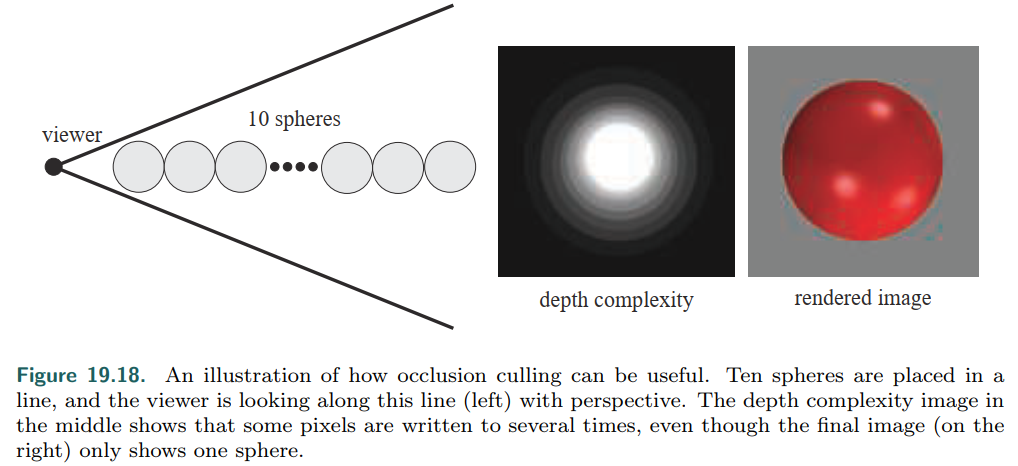
19.7 遮挡剔除(Occlusion Culling)

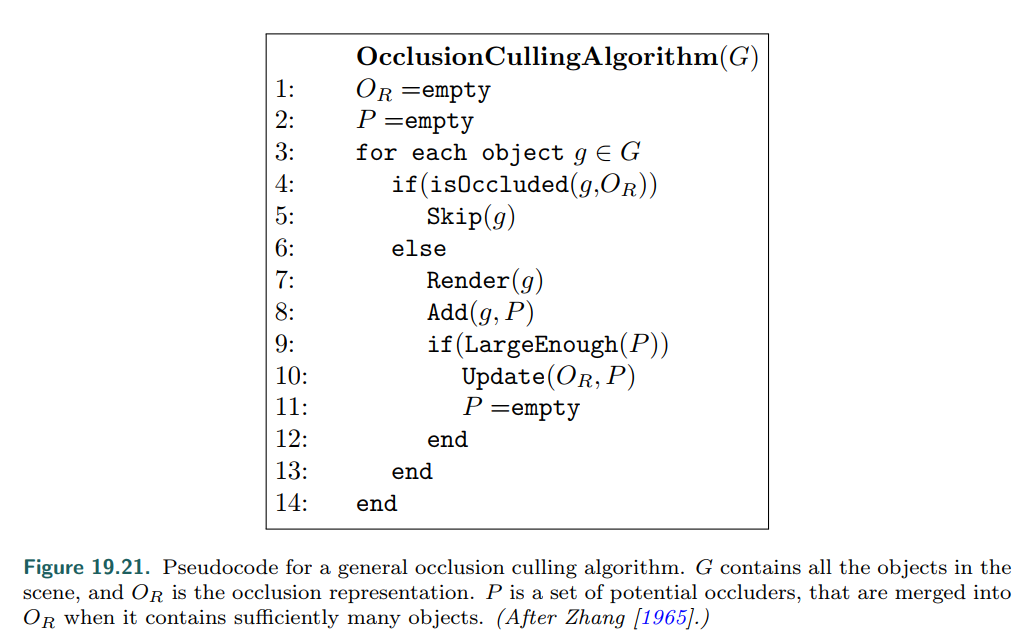
如我们所见，可见性可以通过z缓冲区解决。即使z-buffer正确解决了可见性，它还是相对简单且蛮力的，因此并不总是最有效的解决方案。例如，假设观察者正在沿着放置10个球体的直线看。如图19.18所示。从此视点渲染的图像将仅显示一个球体，即使所有10个球体都将被光栅化并与z缓冲区进行比较，然后可能写入到颜色缓冲区和z缓冲区中。图19.18的中间部分从给定的角度显示了该场景的深度复杂度。深度复杂度是一个像素覆盖的表面数量。对于10个球体，假设背面剔除已启用，则所有10个球体都位于该位置时，中间像素的深度复杂度为10。如果场景是从背面渲染的，则中间的像素将被像素着色10次，即执行9次不必要的像素着色器。即使场景是从前到后渲染的，所有10个球体的三角形仍将被栅格化，并且即使生成单个球体的图像，深度也会被计算并与z缓冲区中的深度进行比较。在现实中不太可能发现这种无趣的场景，但是（从给定的角度来看）它描述了一个人口稠密的模型。在实际场景中会发现这些类型的配置，例如雨林，引擎，城市以及摩天大楼内部的场景。有关示例，请参见图19.19。



考虑到上一段中的示例，似乎可以避免这种效率低下的算法方法可以在性能上有所回报。 这种方法被称为遮挡剔除算法，因为它们试图剔除被遮挡的对象，即被场景中其他对象隐藏的对象。 最佳遮挡剔除算法将仅选择可见的对象。从某种意义上讲，z缓冲区仅选择并渲染那些可见的对象，但并非必须通过大多数管道将视锥内部的所有对象发送出去。有效的遮挡剔除算法的思想是尽早执行一些简单的测试，以剔除隐藏对象集。从某种意义上说，背面剔除是遮挡剔除的一种简单形式。如果我们事先知道对象是实体并且是不透明的，则背面将被正面遮住，因此不需要渲染。

遮挡剔除算法有两种主要形式，即基于点和基于单元。这些如图19.20所示。基于点的可见性就是渲染中通常使用的，即从单个查看位置看到的点。另一方面，对单元格执行基于单元格的可见性，该单元格是空间中包含一组观看位置（通常是盒子或球形）的区域。基于单元格可见性的不可见对象必须在单元格中的所有点都是不可见的。基于单元格的可见性的优点是，一旦为单元格计算了可见性，通常只要观察者在单元格内，它就可以用于几帧。但是，与基于点的可见性相比，计算通常更耗时。因此，通常将其作为预处理步骤。基于点和基于单元的可见性本质上与点光源和面光源相似，在点光源和面光源中，可以将光视为查看场景。对于不可见的对象来说，这等效于它位于本影区域中，即完全处于阴影中。

还可以将遮挡剔除算法分类为在图像空间，物体空间或射线空间中运行的那些算法。 图像空间算法在投影后进行二维可视性测试，而物体空间算法则使用原始的三维对象。射线空间方法[150，151，923]在双重空间中执行测试。每个关注的点（通常是二维的）在此双重空间中转换为射线。对于实时图形，在这三种中，最广泛使用的是图像空间遮挡剔除算法。



一种遮挡剔除算法的伪代码如图19.21所示，其中函数isOccluded被称为可见性测试，用于检查对象是否被遮挡.是要渲染的几何对象的集合,是遮挡表示,是可以与合并的一组潜在遮挡物.根据特定的算法,表示某种遮挡信息.开头将设置为空.此后，将处理所有通过视图视锥剔除测试的对象.

考虑一个特定的对象.首先，我们测试物体相对于遮挡表示是否被遮挡。如果它被遮挡，则不会对其进行进一步处理，因为我们知道它不会对图像有所贡献。如果无法确定该对象被遮挡，则必须渲染该对象，因为它可能对图像有所贡献（在渲染的那一刻）。然后将对象添加到中，如果中的对象数量足够大，则我们有能力将这些对象的遮挡力合并为.因此，中的每个对象都可以用作遮挡物.

请注意，对于大多数遮挡剔除算法，性能取决于绘制对象的顺序。例如，考虑装有发动机的汽车。如果首先拉上汽车的引擎盖，那么发动机（很可能）将被剔除。另一方面，如果先绘制发动机，则不会被剔除。从前到后的粗略排序和渲染可以显着提高性能。同样，值得注意的是，小物体可能是优秀的遮挡物，因为到遮挡物的距离决定了它可以遮挡多少。例如，如果观看者足够靠近火柴盒，则火柴盒可以遮盖金门大桥.

19.7.1 遮挡查询

GPU通过使用特殊的渲染模式来支持遮挡剔除。用户可以查询GPU，以了解与z缓冲区的当前内容相比，一组三角形是否可见.三角形通常形成更复杂对象的边界体积（例如，盒子或k-DOP）。如果这些三角形都不可见，则可以将对象剔除。GPU栅格化查询的三角形并将其深度与z缓冲区进行比较，即，它在图像空间中运行。尽管实际上未修改任何像素或深度，但仍生成了其中可见这些三角形的像素数n的计数。如果n为零，则所有三角形都被遮挡或修剪。

但是，零计数不足以确定边界体积是否不可见。更准确地说，摄像机视锥台可见的近平面的任何部分都不应在边界体积内。假设满足此条件，则将完全包围整个边界体积，并且可以安全地丢弃所包含的对象。如果n> 0，则一小部分像素未通过测试。如果n小于阈值像素数，则该对象可能会被丢弃，因为它不太可能对最终图像做出很大贡献[1894]。 以这种方式，速度可以换来可能的质量损失。 另一个用途是让n帮助确定对象的LOD（第19.9节）。 如果n小，则对象的一小部分（可能）可见（因此可见），因此可以使用细节较少的LOD。

当发现边界量被遮盖时，我们可以避免通过渲染管道发送可能复杂的对象来获得性能。 但是，如果测试失败，则实际上会损失一些性能，因为我们花费额外的时间测试此边界量没有任何好处。

此测试有变体。 出于剔除目的，不需要确切数量的可见片段-只需一个布尔值即可，表明至少一个片段是否通过了深度测试。 OpenGL 3.3和DirectX 11及更高版本支持这种类型的遮挡查询，在OpenGL [1598]中枚举为“任何示例通过”。 这些测试可以更快，因为一旦看到一个片段，它们便可以终止查询。 OpenGL 4.3和更高版本还允许此查询的更快变体，称为ANY SAMPLES PASSED CONSERVATIVE。 该实现可能会选择提供不太精确的测试，只要它是保守的并且在正确的方面出错。 硬件供应商可以通过仅针对粗略深度缓冲区（第23.7节）而不是针对每个像素深度执行深度测试来实现此目的。

查询的等待时间通常是相对较长的时间。通常，在这段时间内可以绘制成百上千个三角形（有关延迟的更多信息，请参见第23.3节）。因此，当边界框包含大量对象并且发生相对大量的遮挡时，这种基于GPU的遮挡剔除方法值得。 GPU使用遮挡查询模型，在该模型中，CPU可以向GPU发送任意数量的查询，然后它会定期检查是否有可用的结果，即查询模型是异步的。就其本身而言，GPU执行每个查询并将结果放入队列中。CPU进行队列检查的速度非常快，CPU可以继续发送查询或实际的可渲染对象，而不必暂停。DirectX和OpenGL均支持谓词/条件遮挡查询，其中查询和对应绘制调用的ID均同时提交。仅当指示遮挡查询的几何形状可见时，GPU才会自动处理相应的绘制调用。这使得模型实质上更加有用。

通常，应该对最有可能被遮挡的对象执行查询。Koval`ek和Sochor [932]在应用程序运行时收集有关每个对象几个帧的查询的运行统计信息。发现对象被隐藏的帧数会影响将来测试该对象被遮挡的频率。也就是说，可见的对象很可能保持可见状态，因此可以减少测试频率。如果可能的话，每隔一帧都会对隐藏的对象进行测试，因为这些对象最有可能从遮挡查询中受益。Mattausch等人[1136]提出了对没有谓词/条件渲染的遮挡查询（OC）的一些优化。他们使用批处理OC，将几个OC组合为一个OC，使用多个边界框而不是一个较大的边界框，并使用时间抖动采样来调度以前可见的对象。

这里讨论的方案给出了遮挡剔除方法的潜力和问题。 何时使用遮挡查询或一般使用大多数遮挡方案通常并不十分清楚。 如果一切都可见，则遮挡算法只会花费额外的时间，而永远不会节省时间。 一个挑战是迅速确定该算法无济于事，因此减少了它徒劳无功的尝试以节省时间。 另一个问题是确定要用作遮挡物的对象集。 视锥内部的第一个对象必须是可见的，因此在这些对象上花费查询是浪费的。 在实施大多数遮挡剔除算法时，确定以什么顺序渲染以及何时测试遮挡是一项艰巨的任务。

19.7.2 分层Z-Buffer

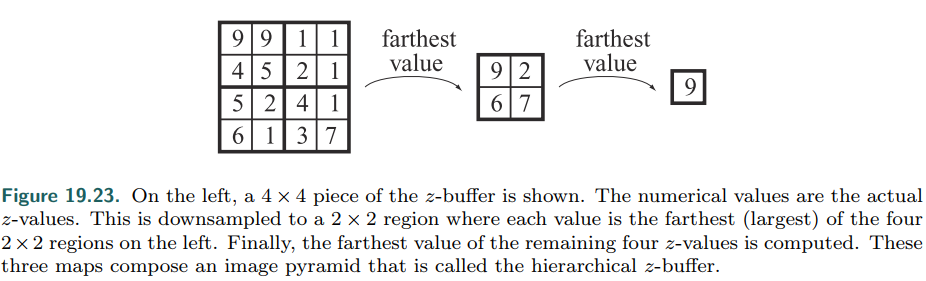
分层z缓冲（HZB）[591，593]对遮挡剔除研究有重大影响.尽管很少使用原始的CPU端形式，但是该算法是z剔除（第23.7节）的GPU硬件方法以及使用在GPU或CPU上运行的软件的自定义遮挡剔除的基础.我们首先描述基本算法，然后介绍该技术如何在各种渲染引擎中采用.

该算法将场景模型保存在八叉树中，并将帧的z缓冲区作为图像金字塔，我们称为z金字塔。该算法因此在图像空间中操作。八叉树可对场景的被遮挡区域进行分层剔除，而Z金字塔可对图元进行分层Z缓冲。因此，z金字塔是此算法的遮挡表示。这些数据结构的示例如图19.22所示。

八叉树节点的层次剔除如下进行。 以大致的前后顺序遍历八叉树节点。 使用扩展遮挡查询（见第19.7.1节），针对z金字塔测试八叉树的边界框。 我们开始在最粗的Z金字塔单元格中进行测试，该单元格封装了盒子的屏幕投影。 然后将框在单元格中最接近的深度（znear）与z金字塔值进行比较，如果znear更远，则已知该框被遮挡了。 此测试将递归地从Z金字塔继续进行，直到发现该框被封闭，或者直到到达Z金字塔的底部为止，然后才可以看到该框。 对于可见八叉树盒，测试将在八叉树中递归向下进行，最后将潜在的可见几何图形渲染到分层z缓冲区中。 这样做是为了使后续测试可以使用先前渲染的对象的遮挡力。

目前，尚未使用完整的HZB算法，但已对其进行了简化和调整，使其可以在GPU上使用自定义剔除或在CPU上使用软件光栅化来与计算过程一起很好地工作。 通常，大多数基于HZB的遮挡剔除算法的工作方式如下：

1. 使用遮挡信息生成完整的分层Z金字塔.
2. 要测试对象是否被遮挡，请将其包围和投影到屏幕空间并估计Z金字塔中的MipLevel。
3. 针对选定的mip level作遮挡测试.如果结果有歧义，可以选择继续使用更好的Mip级别进行测试.



大多数实现不使用八叉树或任何BVH，也不会在渲染对象后更新zpyramid，因为这被认为执行起来过于昂贵。

可以使用“最佳”遮挡物[1637]完成步骤1，该遮挡物可以选择为n个对象的最接近集合[625]，也可以使用简化的艺术家生成的遮挡物图元，也可以使用有关可见物体组的统计信息来完成。 前一帧。 或者，可以使用前一帧中的z缓冲区[856]，但这并不保守，因为有时可能由于不正确的剔除而弹出对象，尤其是在快速摄像头或对象移动下。 Haar和Aaltonen [625]都渲染出最佳遮挡物，并将它们与前一帧深度的1/16低分辨率重新投影结合起来。 然后使用GPU构造z金字塔，如图19.23所示。有些人使用AMD GCN架构的HTILE（第23.10.3节）来加快z金字塔的生成[625]。