目录

[算法层面的剔除 2](#_Toc69394423)

[视锥剔除 2](#_Toc69394424)

[基于Portal的剔除 3](#_Toc69394425)

[基于Occlusion Query的遮挡剔除 4](#_Toc69394426)

[基于Software Rasterization的遮挡剔除 6](#_Toc69394427)

[GPU Driven Rendering Pipeline 7](#_Toc69394428)

[硬件层面的剔除 10](#_Toc69394429)

[Clipping&Backface Culling 10](#_Toc69394430)

[Early-Z 10](#_Toc69394431)

[Z-Cull 11](#_Toc69394432)

[总结 12](#_Toc69394433)

[Unity的遮挡剔除 12](#_Toc69394434)

[遮挡剔除的工作原理 12](#_Toc69394435)

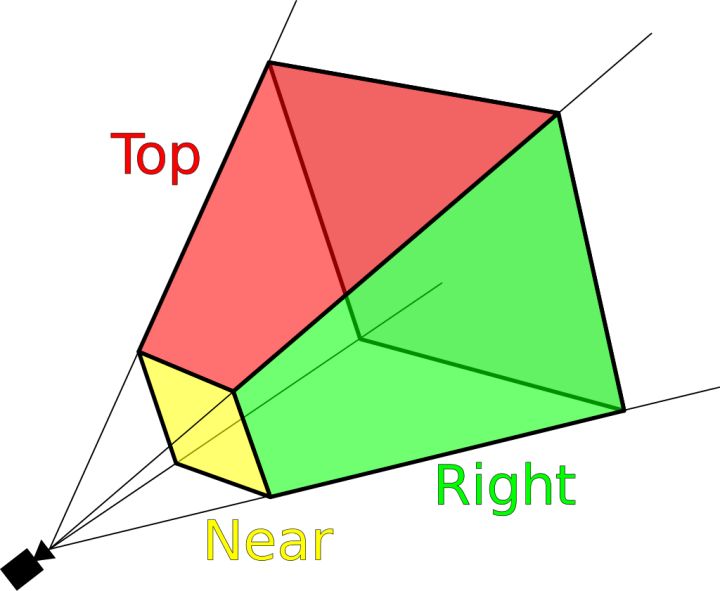
[何时使用遮挡剔除 13](#_Toc69394436)

[对动态游戏对象使用遮挡剔除 13](#_Toc69394437)

# 算法层面的剔除

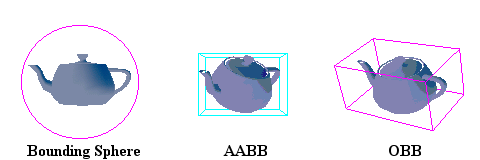
## 视锥剔除

视锥剔除是3D引擎的标配。实时图形学中，摄像机一般是基于针孔模型：光线经过一个小孔（可认为是一个点）照射到小孔背后的传感器阵列（通常是规则的矩形）上，从而形成图像。从小孔出发，只有与传感器阵列有交点的光线才能够真正进入最终的画面，这个区域就是我们一般说的**视锥体**。



理论上，视锥是一个半开放的锥体，但在图形渲染时，基于裁剪空间和性能的需求，我们会设定“近平面”和“远平面”，将这个锥体截取下来的棱台作为视锥

由此得到的最基本的剔除思路就是**视锥剔除**：即简单的判断一个物体是否位于视棱台内。在实践中，由于模型往往是比较复杂的，很难精确计算它和视锥体的交集，因此一般是用**轴对齐包围盒（AABB）**、**有向包围盒（OBB）**或者**包围球（BSphere）**代替模型本身进行相交计算。



对于复杂场景来说，线性数组的遍历方式往往不够高效，这时候也可以将场景以层次结构组织起来进行剔除（比如QuadTree、Octree等）。

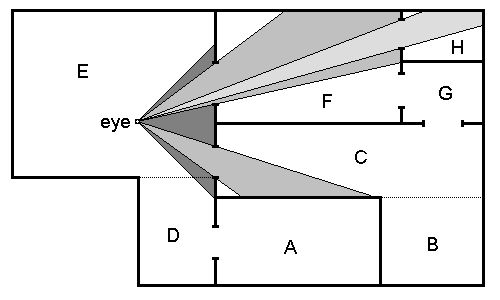
视锥剔除的原理虽然简单，但要做到又快又好仍然不容易，尤其对于性能更严苛的生产环境来说。除了上面提到的场景包围盒的组织结构以外，另一个能够加快剔除速度的方法是利用CPU的SSE指令集，这个指令集的执行方式是SIMD，即单指令多数据，可以同时处理多组浮点数的相同运算。利用这个特性，我们可以同时处理多个包围盒和视锥的相交测试。具体的实现以及性能分析可以参考[Culling the Battlefield: Data Oriented Design in Practice](https://www.gamedevs.org/uploads/culling-the-battlefield-battlefield3.pdf)。

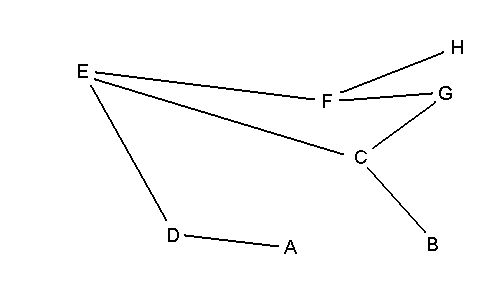
视锥剔除本质上还是一个场景遍历的过程，通常来说，场景是通过树的形式存储和访问，每个子节点的最终变换由所有父节点的变换和它自身的局部变换共同决定，因此如何快速地计算每个节点的变换数据也很重要。[Pitfalls of Object Oriented Programming](https://www.slideshare.net/EmanWebDev/pitfalls-of-object-oriented-programminggcap09)详细地阐述了面向数据编程（DOP）的思想，包括SoA和AoS两种数据布局的差异以及它们对于缓存和性能的影响。其中就以场景遍历作为例子，解释了如何利用DOP的编程思想去优化场景遍历和视锥剔除。

## 基于Portal的剔除

相较于视锥剔除的通用性来说，Portal的应用场景更受限制，同时需要更多的手工标定的步骤，但在某些特定环境下能够发挥很好的效果。

Protal的概念很简单，一般来说它适用于封闭的室内空间：假设我们身处一个封闭的大房子里，房子里有很多相连的房间，Portal就类似于房间之间的门。如果每个房间被看作一个孤立的节点，那Portal就是连接这些节点的桥梁。当我们处于某一个房间里（节点）的时候，只有那些和它相连的房间才有可能被我们看到。Portal就是美术或策划在制作关卡过程中人工标记出来的连接。在剔除的时候，我们只需要在标记好的图里找到我们摄像机所处的房间（节点），然后找到所有与它有Protal连接的节点，没有Protal连接的所有节点都可以直接被判定为不可见从而被剔除。





## 基于Occlusion Query的遮挡剔除

遮挡剔除的原理也很简单：**处于视锥内的模型也未必是可见的，因为它有可能被其它模型完全挡住**。如果我们能够用比较低的代价去找到那些被完全遮挡的模型，那么就不需要对它们再进行绘制，从而提高渲染性能。

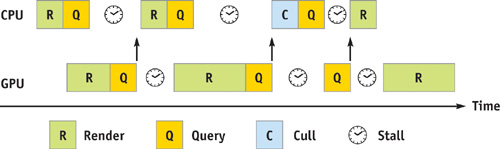
最早尝试在GPU上执行遮挡剔除应该是在occlusion query被支持之后。简单来说，occlusion query允许你在绘制命令执行之前，向GPU插入一条查询，并且在绘制结束之后的某个时刻，从GPU将查询结果回读到系统内存里。这条查询命令得到的是某次DrawCall中通过Depth Test的Sample数量，当这个Sample的数量大于0时，就表示当前模型是部分可见的，否则当前模型完全被遮挡。

基于这个API，我们就可以得到一个比较简单的遮挡剔除策略：

1. 用一个简单的depth only的pass绘制整个场景
2. 每次绘制前后插入occlusion query的命令，并根据passed sample count去标记某个物体是否被完全挡住
3. 执行正常的渲染流程，并剔除那些被标记为完全遮挡的模型

对于复杂的场景，即使只用简单的depth only pass也有很大的VS开销，一个显而易见的优化策略就是**用包围盒代替模型本身去做渲染，为了更加精确，我们也可以用多个紧贴的包围盒或者相对原模型更简单的Proxy Mesh去做occlusion query**[**（Efficient Occlusion Culling）**](https://developer.download.nvidia.cn/books/HTML/gpugems/gpugems_ch29.html)。此外，可以通过batch多个模型或包围盒去减少occlusion query阶段的draw call数量。

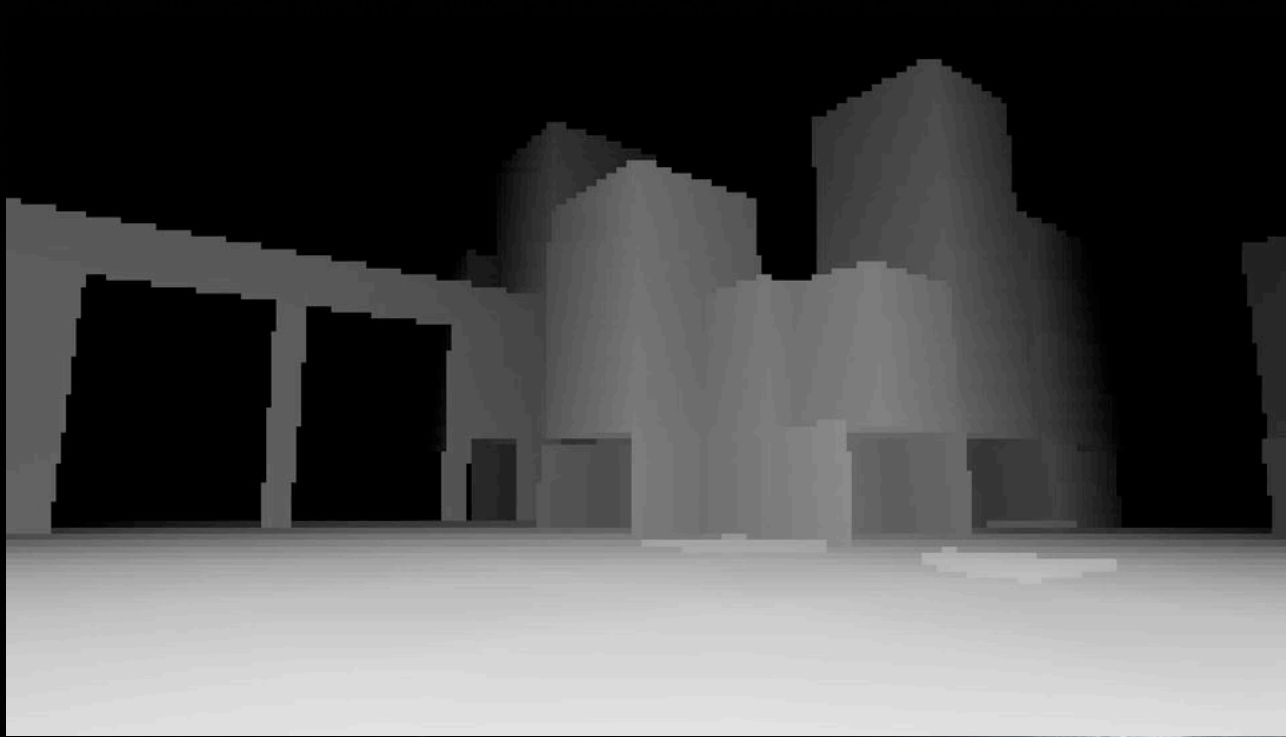
Occlusion query的另一个缺点（也是最致命的缺点）是，**它需要将查询结果回读到系统内存里**，这就意味着VRAM→System RAM的操作，走的是比较慢的PCI-E。同时数据回读可能意味着**图形API背后的驱动程序会在回读的位置Flush整个渲染命令缓冲队列并且等待之前所有的渲染命令执行完毕**，相当于强制在回读位置插入了一个CPU和GPU的同步点，很可能得不偿失（实际上基于Render To Texture的Object Picking也有类似的性能问题，原理同上）。



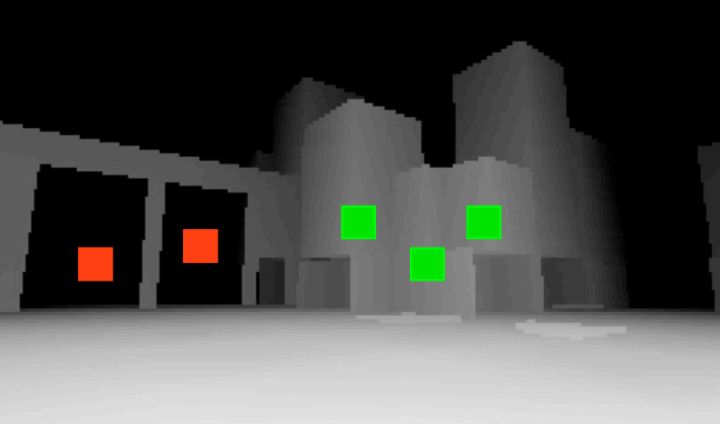
为了解决这个问题，比较常用的方法是让CPU回读前一帧的occlusion query的结果，用来决定当前帧某个物体是否visible，对于相机运动较快的场景，用上一帧的结果可能会导致出错，但由于一般是用包围盒，本身就是保守的剔除，所以总体来说影响不明显，UE4默认使用的就是这样的遮挡剔除方案。除了这个方法以外，GPU Gems的[Hardware Occlusion Queries Made Useful](https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter06.html)用了一个类似分支预测的思路，利用两帧图像的连续性假设，把整个渲染队列里的物体分成了上一帧可见的模型和上一帧不可见的模型。对于上一帧可见的模型，我们就认为它这一帧也可见并且直接渲染它；对于上一帧不可见的模型，我们就插入一个查询到查询队列中然后暂时不处理。当我们没有可以直接用于渲染的模型时，我们再从查询队列里回读查询结果，并根据查询结果去更新被查询物体的可见性状态（用于下一帧的预测），同时若该模型可见，则执行渲染。

## 基于Software Rasterization的遮挡剔除

这个方案最早应该是Frostbite提出来，用于BattleField3的剔除方案。这个方案的思路是，首先利用CPU构造一个低分辨率的Z-Buffer，在Z-Buffer上绘制一些场景中较大的遮挡体（美术设定的一些大物体+地形）：



在构造好的Z-Buffer上，绘制小物体的包围盒，然后执行类似于occlusion query的操作，查询当前物体是否被遮挡：



这个方法相对来说比较灵活（对于光源、物体均适用），由于是纯CPU的，集成起来也比较简单。对于主机平台还可能利用SPU之类的多余算力，同时不会有GPU停顿的问题。缺点是需要美术指定一些大的遮挡体，对于CPU限制的项目可能会负优化。天刀的自研引擎中应用了这个优化策略[（为实现极限性能的面向数据编程范式）](http://twvideo01.ubm-us.net/o1/vault/gdcchina14/presentations/833779_MiloYip_ADataOrientedCN.pdf)。

## GPU Driven Rendering Pipeline

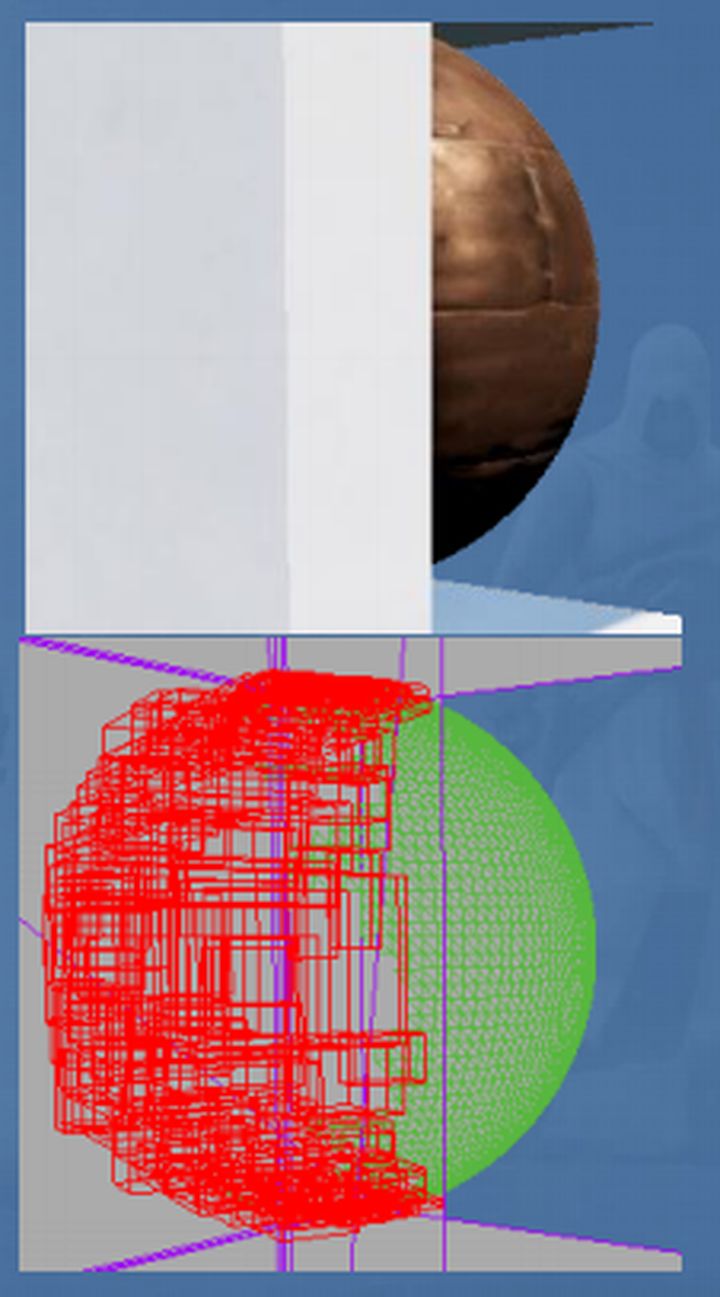
参考文章：

1. [GPU-Driven Rendering Pipelines](http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf)
2. [Optimizing the Graphics Pipeline with Compute](https://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/29204330/GDC_2016_Compute.pdf)

这个思路的产生和发展得益于图形API和硬件的发展，具体来说，有两个feature至关重要：**Compute Shader**以及**ExecuteIndirect**。前者允许我们在GPU上方便地执行各类和渲染无关的GPU运算，并且将计算结果以Buffer或者Texture的形式存储在VRAM上；后者允许我们以GPU Buffer的形式直接构建Draw Command List。这两者结合起来，就表示我们能够在Compute Shader里构造Draw Command List用于绘制，整个过程无需CPU参与。

先抛开具体实现细节，回到我们最初引入剔除的初衷：**我们希望GPU知道哪些物体是不需要被渲染的（视锥之外或被完全遮挡）**，这个信息仅供GPU使用。此外，剔除算法的并行性很好，计算过程又相对简单，没有太多的分支和跳转，非常适合GPU去做。为此，我们只需要将场景的所有渲染资源（包括几何信息、材质信息、变换信息、包围盒信息）以一定规则打包存储在Buffer中，然后提供摄像机（视锥）信息和遮挡体绘制的Z-Buffer（类似于Software Rasterization的Z-Buffer），通过Compute Shader去执行视锥剔除和遮挡剔除，并将通过剔除的模型经由ExecuteIndirect提交渲染。

具体执行的时候，得益于GPU强大的浮点数计算能力，我们可以做比模型更细粒度的剔除：将Mesh切分成Cluster，每个Cluster有64个顶点，并且重排IndexedBuffer（Cluster大小的选取以及重排IB主要是为了提高Vertex Fetch时缓存的命中率，进而提高Vertex Fetch的速度），基于Cluster计算包围盒，利用Cluster Bounding Box去做视锥剔除。

对于遮挡剔除，在构造用于剔除的hierarchy Z-Buffer的时候，也能够利用比Software Rasterization更快的Hardware Rasterization在GPU端去做。



构造Hi-Z的过程，首先基于美术指定的大遮挡体和地形去渲染一个Z-Buffer，然后down sample到低分辨率，混合上一帧Z-Buffer经过reprojection的结果，最后做一系列down sample得到一个层次Z-Buffer的结构

不同于Software Rasterization的方法，Hi-Z意味着我们可以从最粗粒度的Z-Buffer开始进行遮挡查询和剔除，相较于全分辨率的遮挡查询，这样的查询效率更高。

除了常用的Frustum Culling和Occlusion Culling，我们还可以在Compute Shader里去做Backface Culling和Small Primitive Culling，把背面的三角形和面积很小的三角形剔除掉。

在剔除工作完成后，通常会启动一个Compaction的Compute Shader，这个Shader会把通过culling的所有triangle复制到一个更紧凑的Buffer里面，并且执行一些基于材质Batch的合并策略。最后调用ExecuteIndirect来渲染最终场景：



GPU Driven Rendering Pipeline的核心思想是减少CPU和GPU之间的通信，尽量将所有渲染相关的事务（包括提交）都放在GPU端，解放CPU的算力用于构造物理和AI的规则。同时，利用GPU的算力能够更细粒度地去控制渲染命令队列内的生成和合并。在实际渲染时，除了我们这里提到的基于Compute Shader的Culling和Batching，还需要辅以Virtual Texturing、Mega Texture等技术并对渲染管线做配套改造，对原本的渲染引擎架构改动也较大，想要把这一技术植入引擎中并不容易。这部分更细节的内容，除了育碧和寒霜引擎的两篇分享，也有非常多的文章具体阐释，可以配合阅读。

1. <https://zhuanlan.zhihu.com/p/33881505>
2. <https://zhuanlan.zhihu.com/p/37084925>
3. <https://bazhenovc.github.io/blog/post/gpu-driven-occlusion-culling-slides-lif/>
4. <https://zhuanlan.zhihu.com/p/47615677>

# 硬件层面的剔除

## Clipping&Backface Culling

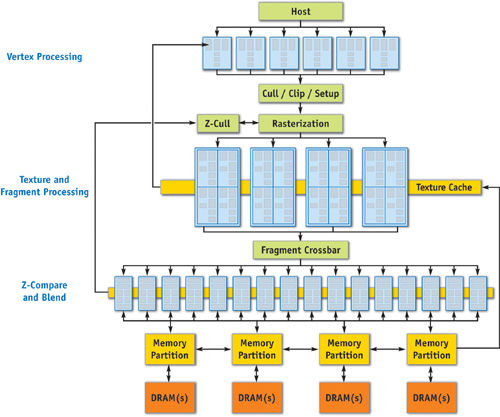
对[Graphics Pipeline](https://link.zhihu.com/?target=https%3A//www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview)稍有了解的同学对这两个环节应该都不陌生，Clipping是当一个三角形的顶点位置被变换到NDC外的三角形和穿过NDC的三角形，会执行剔除或者裁剪的操作，具体的裁剪规则，可以参见[这个问题](https://www.zhihu.com/question/28904971)。至于Backface Culling，则是在图元装配阶段结束之后，根据用户指定的方向，把面向摄像机或者背对摄像机的三角形剔除[（https://en.wikipedia.org/wiki/Back-face\_culling）](https://en.wikipedia.org/wiki/Back-face_culling)，剔除后的三角形就不会再进入到Pixel Shader和Rasterization的流程里。

## Early-Z

提到Early-Z就必须提对应的Late-Z：**在图形管线中，逻辑上Depth Test和Stencil Test是发生在Pixel Shader的执行之后的**，因为Pixel Depth在Pixel Shader阶段还有可能被修改，所以Pixel Shader→Depth Test的流程顺序就是Late-Z。但由于Pixel Depth修改的需求非常少（基于深度混合的Impostor和某些粒子效果），所以绝大部分情况下，Pixel Depth在Rasterization之后、Pixel Shader执行之前就可以被确定下来，如果我们能够把Depth Test放在Pixel Shader之前，对那些没通过Depth Test的像素不执行Pixel Shader，就能够一定程度上减少SM的压力，这就是Early-Z这个优化策略的初衷，现在已经是GPU的标配了。默认在Pixel Shader里没有修改Depth的操作时，这个优化就会开启。

## Z-Cull

很多人会将Z-Cull和Early-Z弄混，其实他俩并不一样，重点体现在剔除的粒度不同：Z-Cull的剔除是粗粒度的**Pixel Tile（比如一个8\*8的像素块）**，而Early-Z是细粒度的**2\*2的Pixel Quad**。在Z-Cull进行Depth Test的时候，Pixel Tile会被压缩以加速比较（主要是减少带宽开销），比如用平面方程的系数表示一块Pixel Tile，用平面方程去和Z-Buffer做Coarse Depth Test，而不是Tile内部逐个像素去做Depth Test。正因为如此，常用的Alpha Test会让一个原本完整的平面出现空洞，这就会破坏Pixel Tile的压缩算法，进入导致Z-Cull无法开启。



现代引擎基本都会利用Z-Cull和Early-Z的特性去减少SM的计算压力，具体方法是执行一个Z-Prepass（不论是forward、forward+还是deferred管线都一样）：先将不透明物体按照距离摄像机从前向后的顺序排序，然后只开启Z-Buffer write和compare，不执行Pixel Shader进行一遍渲染。在执行完Z-Prepass后，关闭Z-Buffer的写入，将compare function改为equal，然后执行后续复杂Pixel Shader（前向渲染的光照计算或者延迟渲染的G-Buffer填充）。

有关Early-Z和Z-Cull更多的开启条件，可以看：

1. <https://zhuanlan.zhihu.com/p/53092784>
2. <https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter30.html>

# 总结

总体来说，**可以认为从算法剔除到硬件剔除是一个粒度在逐渐变精细的过程（Mesh→Cluster→Triangle→Pixel Tile→Pixel Quad）**。而GPU Driven Rendering Pipeline作为一个相对来说比较特殊的存在，它不是一个具体的算法而是一种思路，这种思路代理了某些传统硬件上我们认为是固定管线的功能，同时尽可能地减少CPU和GPU的通信。它的出现符合现代硬件的发展趋势：一是可编程管线的功能日益强大进而代替更多的固定管线单元；二是相较于密集的计算量，现代程序的优化更多地依赖于如何提高硬件的并行程度，减少等待和同步，以及如何优化访存。

# Unity的遮挡剔除

## 遮挡剔除的工作原理

遮挡剔除会在Unity Editor中生成有关场景的数据，然后在运行时使用该数据来确定摄像机可以看到的内容。这种生成数据的过程称为**烘培**。

**在对遮挡剔除数据进行烘培时，Unity将场景划分为多个单元，并生成描述单元内几何体以及相邻单元之间可见性的数据**。然后，Unity尽可能合并单元，以减少生成的数据的大小。要配置烘培过程，可以在Occlusion Culling窗口中更改参数，并在场景中使用遮挡区域。

**在运行时，Unity会将这些烘培的数据加载到内存中，并且对于每个启用了Occlusion Culling属性的摄像机，将会对数据执行查询以确定该摄像机可以看到的内容**。请注意，启用遮挡剔除后，摄像机将执行视锥体剔除和遮挡剔除。

## 何时使用遮挡剔除

要确定遮挡剔除是否有可能改善项目的运行时性能，请考虑以下事项：

* 防止无意义的渲染操作可以节省CPU和GPU时间。Unity内置遮挡剔除在CPU上执行运行时计算，这可能会抵消其节省的CPU时间。因此，当项目因过度绘制而具有GPU密集型特征时，遮挡剔除最有可能提高性能。
* Unity在运行时将遮挡剔除数据加载到内存中。必须确保有足够多的内存来加载此数据。
* 当场景中一些界限明确的小区域被实体游戏对象彼此隔开时，遮挡剔除的效果最好。一个常见的例子是通过走廊连接的房间。
* 可以使用遮挡剔除来遮挡动态游戏对象，但动态游戏对象不能遮挡其它游戏对象。如果项目会在运行时生成场景几何体，则Unity的内置遮挡剔除不适用于该项目。

## 对动态游戏对象使用遮挡剔除

如果游戏对象是静态的，则Unity可以将这个游戏对象作为静态遮挡物（Static Occluder）或静态被遮挡物（Static Occludee）烘培到遮挡剔除数据中。如果游戏对象是动态的，这个游戏对象可以在运行时充当被遮挡物，而不能充当遮挡物。

要确定动态游戏对象是否充当被遮挡物，可以在任何类型的渲染器组件上设置Dynamic Occlusion属性。启用Dynamic Occlusion后，渲染器在摄像机的视图中被静态遮挡物（Static Occluder）阻挡时，Unity会剔除渲染器。禁用Dynamic Occlusion后，渲染器在摄像机的视图中被静态遮挡物（Static Occluder）阻挡时，Unity不会剔除渲染器。

默认情况下会启用Dynamic Occlusion。为了获得特定的效果，例如在墙后的角色周围绘制轮廓，可能需要禁用Dynamic Occlusion。

如果确定Unity绝对不应该将遮挡剔除应用于特定的游戏对象，则可以禁用Dynamic Occlusion以减少运行时计算并降低CPU使用率。虽然这些计算对每个游戏对象的影响很小，但如果规模足够大，减少这些计算可能有利于提高性能。