* 1. 介绍

我们提供了一个用于实时剔除和渲染复杂场景的综合系统，该系统是我们内部开发的，供以后在Dawn Engine中使用。 这是Eidos-Montreal研发团队（名为Labs）进行的几个研究项目之一，但是该系统并未在《杀出重围：人类分裂》游戏中使用。我们研究的主要目标是开发一种系统，该系统可以满足对视觉保真度和运行时性能不断增长的要求，但仍将与传统游戏资产兼容并允许在游戏制作期间实现快速迭代。

我们的剔除系统将基于分层深度缓冲区的方法的低延迟和低开销与传统GPU硬件遮挡查询的像素精度结合在一起[Hill and Collin 2011]。 它有效地剔除高度动态，复杂的环境，同时保持与标准网格资产的兼容性。 我们的渲染系统采用了一种实用的方法来进行延迟纹理化处理[Reed 2014]，并在使用常规纹理资源的同时有效地支持高度多样化和复杂的材质。拟议系统的两个部分都利用DirectX 12可用的新图形功能，其中最著名的是增强的间接渲染和新的着色器资源绑定模型。

* 1. 概览

对于像视频游戏这样的实时渲染应用，至关重要的是要使用有效的遮挡剔除系统，以便能够在确保高交互帧率的同时渲染大型环境。传统的基于CPU的剔除系统（例如门户剔除）无法支持动态的，复杂的，经过Alpha测试的遮挡体，因此它们不适用于此类环境。我们的剔除系统部分基于[Haar and Aaltonen 2015]提出的思想，其中前一帧的深度缓冲区用于获取初始可见性，并使用当前帧中更新的深度缓冲区重新测试潜在的假阴性。这样，我们避免渲染专用的遮挡几何体，这可能很难生成。例如自然环境。但是，不是使用基于分层深度缓冲区的方法，而是根据[Kubisch and Tavenrath 2014]的精神使用了一个概念，该概念依赖于现代消费类图形硬件的早期深度模板测试功能。主要思想是使用从前一帧开始的降采样和重新投影的深度缓冲区，使用专用的图形硬件光栅化遮挡物的边界。通过强制关联的像素着色器使用早期的深度模板测试，只有可见的片段在每个网格实例唯一的公共位置的通用GPU缓冲区中标记，相应实例才可见。与基于分层深度缓冲区的方法相比，使用非聚类标准网格资产可显着提高剔除效率。后续的计算着色器根据获取的可见性信息生成数据，该数据以后将用于间接渲染。正如[Haar and Aaltonen 2015]所提出的那样，将使用当前帧中更新的深度缓冲区重新测试被遮挡的对象，以避免遗漏误报。

对于现代游戏，利用可处理日益复杂的网格几何体和逼真的表面材质的渲染系统也很重要。前向渲染系统支持很高的材质多样性，但是它们要么透支，要么需要深度预先通过，这对于三角形数较高的网格，GPU硬件细分，alpha测试或顶点着色器蒙皮来说可能是昂贵的。延迟渲染系统无需深度预先管理就可以高效运行，但仅支持有限范围的材质，因此，对于更多种材质，通常需要进行额外的正向渲染。我们的延迟纹理化实用方法通过支持高材质多样性同时仅执行单个几何图形遍历，结合了两个渲染系统的优势。我们比传统的延迟渲染更进一步，将几何形状与材质和照明完全分离。在初始几何图形遍历中，所有经过GPU剔除阶段的网格实例均被间接渲染，并将其顶点属性写入，压缩到一组几何缓冲区中。没有进行特定于材料的操作和纹理获取（除了alpha测试和某些类型的GPU硬件镶嵌技术）。随后的全屏扫描将材料ID从几何缓冲区传输到16位深度缓冲区。最终，在每种材质的着色通道中，将渲染一个屏幕空间矩形，该矩形将所有可见网格的边界包围起来。矩形顶点的深度设置为与当前处理的材质ID对应的值，并使用早期的深度模板测试拒绝其他材质中的像素。使用相同着色器和资源绑定布局的所有标准材质都通过动态索引的纹理一次性渲染。此时，特定材质的渲染和光照（例如平铺[M. Billeter和Assarsson 2013]或群集[Olsson等人2012]）同时完成。

* 1. 实现

在以下各节中，将详细描述系统的每个步骤。 这些说明假定使用DirectX 12，但该系统也可以用OpenGL或Vulkan实施。 没有进行图形硬件特定的假设，并且提供的着色器代码与硬件无关。

* + 1. 剔除

该系统的剔除部分可分为两遍，每遍包含几个不同的步骤（图1.1）。 我们决定使用实例化而不是扁平化的网格列表。 尽管将在GPU上生成绘制调用，但内部测试表明，实例化与紧凑的间接绘制缓冲区配合使用，可将GPU渲染到几何缓冲区的GPU时间减少了约45％。

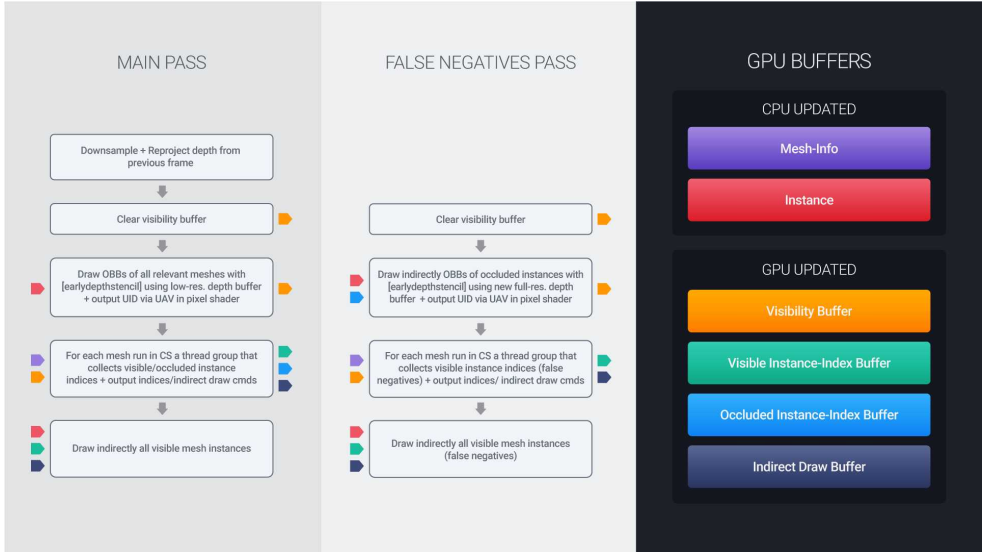


图1.1 筛选系统涉及的每个步骤的概述。左侧的灰色块表示每个剔除步骤，右侧的块表示所需的GPU资源。每个剔除步骤左侧的箭头代表输入数据，右侧的箭头则输出数据。颜色与相应的GPU缓冲区的颜色匹配.

Main Pass 第一遍获得初始可见性，并且可以细分为5个不同的步骤.

深度缓冲区下采样和重新投影。为了加速该算法，首先通过保守地取4×4像素区域的最大值，将前一帧的深度缓冲区降采样为四分之一分辨率。我们发现深度缓冲器的进一步下采样没有提供任何明显的性能改进和降低的剔除效率。由于较大的平面物体可能导致错误的自遮挡，因此将降采样的深度缓冲区重新投影到当前帧。这是通过在无序访问视图（UAV）上通过原子最大操作将重新投影的深度值散布到新的像素位置来完成的。不幸的是，在高摄像机运动下，这会产生明显降低剔除效率的孔。为了封闭这些孔，另外将重新投影的深度值写入当前处理的像素位置。摄像机后面的较大的重新投影深度值可能是孔的另一个来源。在这种情况下，我们使用从最后一帧开始的非重新投影深度。这样，即使在高摄像机运动情况下也可以实现高剔除效率，同时大大减少了错误的自我遮挡（图1.2）。

清单1.1显示了用于计算着色器的HLSL代码，该着色器从最后一帧开始对深度缓冲区进行降采样并重新投影。 由于深度缓冲区不支持UAV，因此先将降采样和重新投影的深度值写入颜色缓冲区，然后再复制到最终的深度缓冲区中。





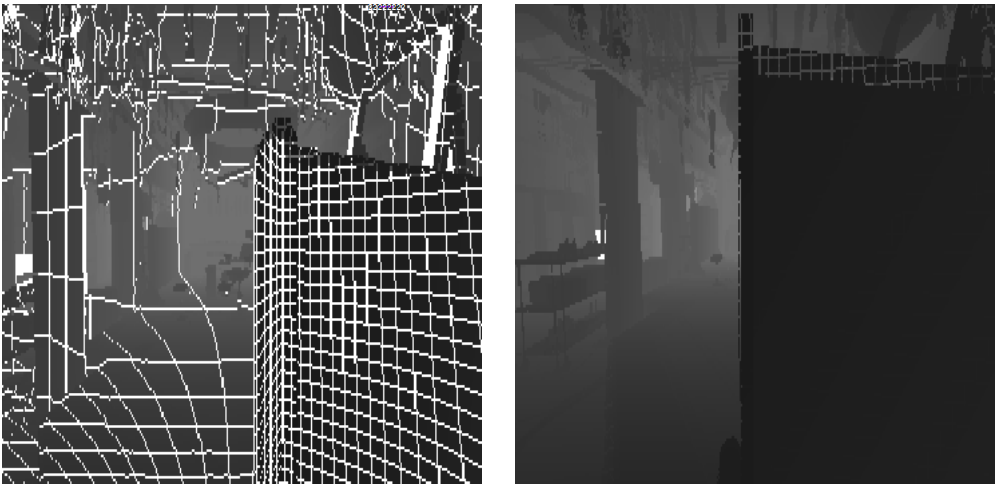
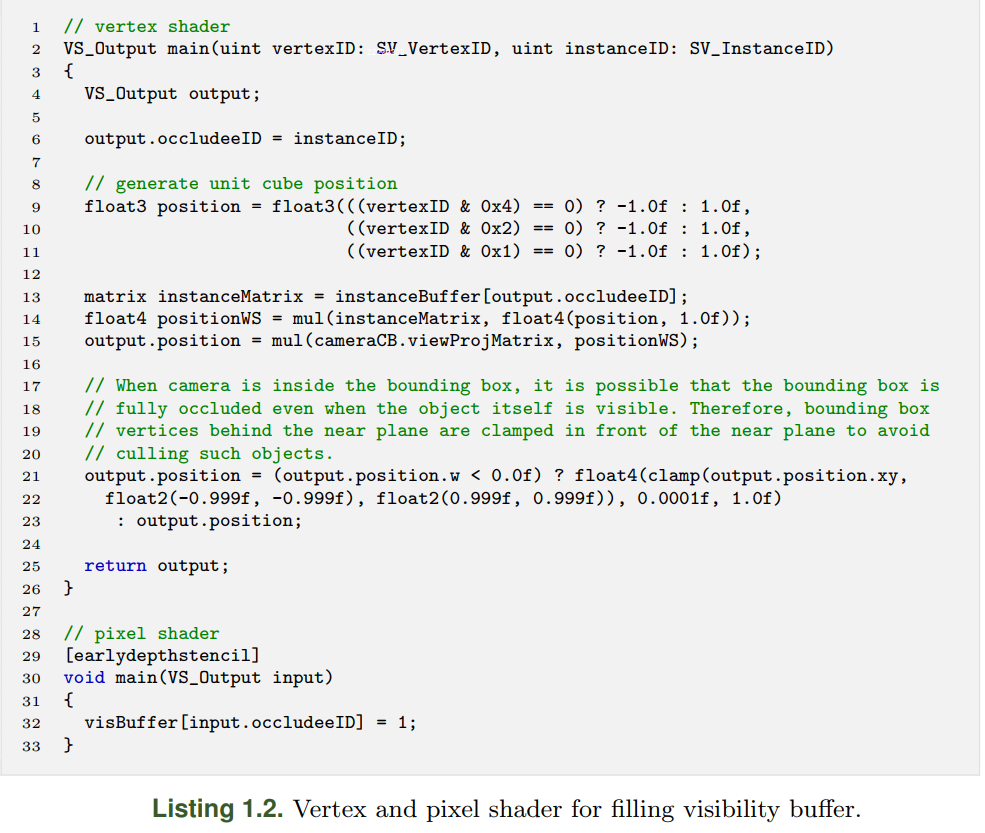


图1.2 左图显示了从最后一帧开始向下采样的，重新投影的深度缓冲区，该缓冲区有很多孔，显着降低了剔除效率。右图显示了建议的措施如何填补此类漏洞。

清除可见性缓冲区。 可见性缓冲区是全局结构化缓冲区，用于跟踪所有已处理的网格实例的可见性。 通过为可见性缓冲区中的每个网格实例使用32位UINT，我们避免了下一步的原子存储操作。 每个条目剩下31位未使用； 但是，即使是能够处理一百万个网格实例的可见性缓冲区，也只会消耗4 MB的GPU内存，这在性能与内存之间取得了很好的折衷。 通过使用DirectX 12的专用UAV清除API或通过在缓冲区上手动运行简单的计算着色器，将可见性缓冲区的条目清除为零。

填充可见性缓冲区。 在此步骤中，使用从上一步进行深度测试的向下采样的，重新投影的深度缓冲区，在单个索引的实例化绘制调用中渲染在CPU上通过截锥体剔除的所有网格实例的定向边界框（OBB）。关联的像素着色器带有[earlydepthstencil]标记，因此只有通过的片段将借助UAV写入可见性缓冲区中的唯一网格实例位置。 通过避免原子操作和ROP，可以实现快速的执行时间。 清单1.2显示了用于填充可见性缓冲区的顶点和像素着色器。



使用第二个实例缓冲区渲染OBB，其中每个实例矩阵都与比例/偏差矩阵预先组合，该比例/偏差矩阵会在应用实际实例变换之前将单位轴对齐的边界框（AABB）转换为网格AABB。 由于渲染三角形计数较低的实例化网格曾经次优[Bilodeau 2014]，因此我们尝试在一次绘制调用中渲染所有OBB，而无需实例化。结果表明，在大约7000个总体边界盒中，该方法比实例方法（在NVIDIA Geforce GTX 970上测试）略慢。因此，我们决定使用实例化渲染边界框。

与传统的硬件遮挡查询一样，当摄像机位于遮挡物的边界框内时，相应的网格会遮挡自身边界框，从而导致错误的遮挡（图1.3）。因此，将摄像机平面附近的边界框顶点直接夹在摄像机平面附近，以确保始终将这些对象标记为可见。

代替使用自动计算的边界框，在执行间接的帮助下，还可以选择使用艺术家创建的三角形数较少的网格，以更好地近似要剔除的网格的形状。

生成间接抽奖信息。对于每个相关的网格（在CPU上至少有一个实例通过视锥筛选），将调度计算着色器线程组。组中的每个线程都会检查先前生成的可见性缓冲区，以查看相应的网格实例是否可见。将所有可见实例的索引写入可见实例索引缓冲区，并将所有被阻塞的实例的索引写入阻塞实例索引缓冲区。另外，如果发现至少一个网格实例可见，则将新的间接绘制命令写入间接绘制缓冲区。为了支持需要不同着色器以有效填充几何缓冲区的多种网格类型（不透明，经过阿尔法测试，镶嵌，蒙皮等），间接绘制命令从特定于网格类型的偏移量写入间接绘制缓冲区。在可见的实例索引缓冲区的开头，每种网格类型的间接绘制命令的数量被累积到单独的条目中，并且被遮挡的实例的数量被写入第一个间接绘制命令的InstanceCount成员中。稍后将其用于间接间接渲染所有遮挡的边界框，以测试假阴性。在调度生成间接绘图信息的计算着色器之前，必须清除这些缓冲区条目（清单1.3）。

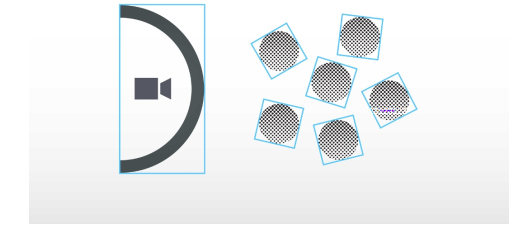
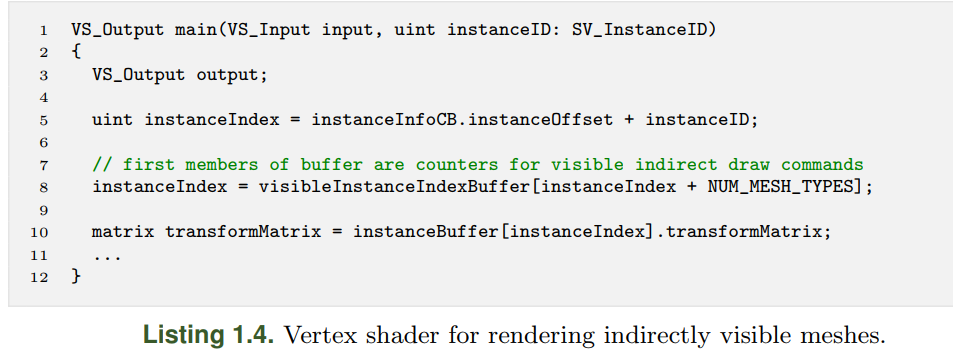


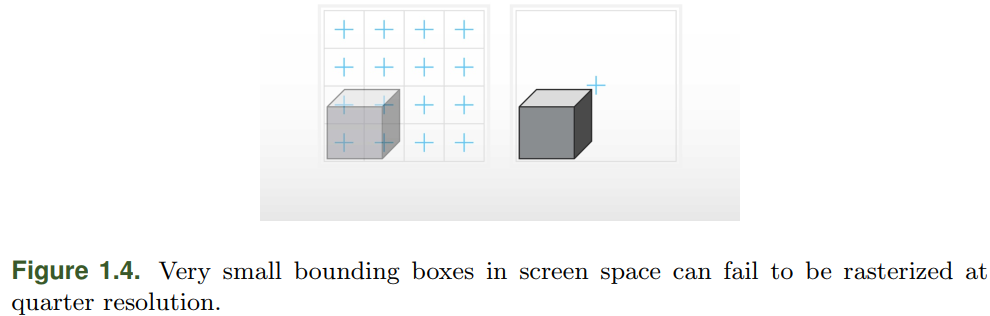
图1.3 由于左侧的对象遮挡了其自己的边界框，因此不会渲染相应的网格，因此在更新的深度缓冲区中将其丢失。 由于更新后的深度缓冲区不包含此主要遮挡物，因此几乎所有右侧的遮挡对象都将被标记为可见并稍后渲染。

*清单1.3 GPU zen p.84-85*

间接渲染可见的网格。在此步骤中，所有可见的网格实例都被渲染到几何缓冲区中，稍后用于材质渲染和照明。对于每种网格类型（不透明，经过阿尔法测试，镶嵌，蒙皮等），将使用先前生成的间接绘图缓冲区作为间接绘图调用的源，发出单独的ExecuteIndirect命令。对于每个ExecuteIndirect命令，将从上一步开始的对应于特定网格类型的偏移量用作到间接绘图缓冲区的偏移量，并在可见的实例索引缓冲区的开始处使用对应的条目来指定间接调用的次数。借助于每个网格实例的可见实例索引缓冲区，可以从实例缓冲区中检索转换矩阵。每个顶点必须知道从可见的实例索引缓冲区开始读取的偏移量。此信息由根常量参数提供，该根常量参数在每个间接绘制命令的开头指定。第二个根常数参数提供材料ID，需要将其写入几何缓冲区以进行后续渲染。清单1.4显示了用于渲染的顶点着色器。



假阴性通过。因为在中，使用从最后一帧开始的向下采样的重新投影深度缓冲区执行可见性剔除，所以可能会出现假阴性。造成这种情况的第一个明显原因是重新投影从前一帧获得的信息而引入的错误。第二个原因是，出于性能方面的考虑，在四分之一分辨率的第一个遮挡通道中渲染了边界框。因此，可能无法对屏幕空间中的很小的边界框进行栅格化，因此，相应的对象将被标记为遮挡（图1.4）。



为防止此类对象被错误地剔除，必须针对全分辨率深度缓冲区检查潜在的假阴性。从理论上讲，通过保守光栅化，第二次遮挡通道也可以使用四分之一分辨率深度缓冲区。但是，我们测试了这种方法并遇到了多个问题：

* 我们用于测试的显卡（NVIDIA GeForce GTX 970，第二代Maxwell架构）仅支持保守栅格化的第1层。 不幸的是，第1层剔除了非常小的三角形，这些三角形由于子像素网格的捕捉而退化，从而潜在地淘汰了可见对象。 此外，第1层具有很大的不确定性区域（像素大小的一半），从而导致过度保守性并显着降低了剔除效率。 尽管在撰写本文时第2层解决了这些问题，但英特尔的Skylake架构是唯一支持该层的架构。
* 在我们以1920×1080屏幕分辨率进行的测试中，将当前深度缓冲区下采样到四分之一分辨率所花费的时间大于第二次遮挡通过使用具有保守光栅化功能的四分之一分辨率深度缓冲区所获得的收益。 因此，第二次遮挡通道使用当前分辨率下从当前帧开始的深度缓冲区（在第一次遮挡通道中渲染可见网格之后获得）来检测假阴性，并且可以细分为4个不同的步骤。

**清除可见缓冲区** 如在主要遮挡通道中所做的那样，可见缓冲区被清除为零。

**填充可见性缓冲区** 与主遮挡通道相似，所有遮挡的网格实例的OBB都在一个索引的实例化绘制调用中呈现.但是这次，渲染是使用保留给被遮挡对象的间接绘制缓冲区的第一个条目间接完成的.借助封闭的instanceindex缓冲区，可以从实例缓冲区中检索变换矩阵，以变换边界框.与主遮挡通道一样，如果至少一个片段通过了早期深度模板测试，则像素着色器会将每个实例标记为在可见性缓冲区中可见.如上所述，当前帧的深度缓冲器以全分辨率用于深度测试.由于在第一遍中无论如何都会渲染所有有问题的对象，因此不必对位于相机附近平面后的边界框顶点执行夹紧代码。

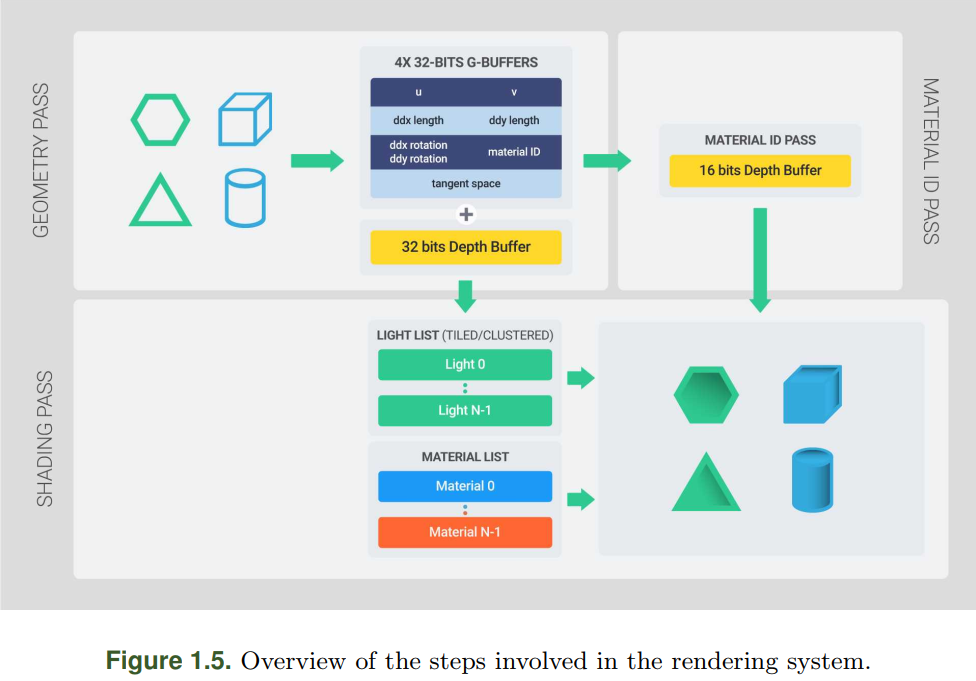
**生成间接绘图信息** 间接抽签信息的生成与主要遮挡过程相同。 但是，这次只生成可见的实例绘制信息，因为对于被阻塞的实例不需要进一步的处理。

**间接呈现假阴性** 此步骤与主要遮挡通道中的相应步骤没有什么不同，并确保建议的系统仅剔除遮挡的物体。

* + 1. 渲染

系统的渲染部分可以细分为3个不同的步骤（图1.5）。

**Geometry Pass.** 所有通过剔除系统的网格实例都将间接渲染到一组压缩的几何缓冲区中（每个像素4×32位）。 与传统的延迟渲染相比，除某些特殊情况外，不执行任何特定于材料的代码或获取纹理，例如使用alpha蒙版纹理进行alpha测试或使用对高度图进行采样的位移映射进行GPU硬件细分。 通过将顶点属性输出到随后的材质渲染和光照所需的几何缓冲区中，可以将几何处理与材质和光照完全分开。



**几何缓冲区布局** 由于当前用户图形硬件的内存带宽比计算能力受到更多限制，因此，保持几何缓冲区尽可能小很重要。在以下部分中，我们说明如何将所需的顶点属性压缩到几何缓冲区中.

纹理坐标。为了避免出现精度伪影，纹理坐标必须以每个像素至少2×16位的形式存储。 包裹纹理坐标后，即立即超过[0; 1]范围，每个组件16位通常已经不够了。幸运的是，存在一种简单的方法，即通过在插值后仅将它们的小数部分存储在像素着色器中，从而以2×16位/像素的方式以足够的精度存储包装的纹理坐标。 由于原始纹理坐标的导数存储在旁边，因此以后将看不到接缝。 纹理坐标以DXGI\_FORMAT\_R16G16\_SNORM的形式存储在第一个几何缓冲区中。

纹理坐标导数。从理论上讲，可以通过使用相邻纹理坐标在着色通道中重建导数。但是，在几何形状边缘的情况下，有时无法始终获得适当的相邻纹理坐标，因此会出现伪影。 这在带有密集的经过Alpha测试的树叶的相机运动下尤其明显，可以观察到时间伪像。 因此，我们决定将纹理坐标及其派生值一起存储。

纹理坐标导数每像素至少需要4×16位，才能获得足够的精度。但是，通过将X方向和Y方向的导数视为2D向量，可以从64位压缩到48位。通过将矢量长度与方向解耦，可以将矢量长度存储为2×16位，将方向存储为2×8位，这仍然为各向异性纹理过滤提供了足够的精度。清单1.5显示了如何在HLSL中对派生进行编码/解码.导数的长度以DXGI\_FORMAT\_R16G16\_FLOAT的形式存储在第二个几何缓冲区中，第三个几何缓冲区的红色通道中的方向存储为（DXGI\_FORMAT\_R16G16\_UINT）.

*清单1.5 GPU zen p.101*

切线空间。对于法线贴图，视差遮挡贴图等，需要存储由切线，双切线和法线组成的切线空间。根据[McAuley 2015]，我们考虑将切线空间作为四元数以每像素32位的形式进行存储，但是由于在光滑发亮的表面上具有可见的刻面，因此放弃了这种方法。相反，我们将切线空间存储为轴角表示，将法线存储为轴，将切线存储为角度。通过使用八面体法向矢量编码，法线以2×10位存储[Meyer等。 2010]。对于切线，首先必须找到一个可以轻松重构以进行解码的向量，并保证该向量与法线正交。由于奇异性问题和不同频率的噪声，选择任意正交向量会产生严重的伪影。为了克服这些问题，我们首先通过获取切线的最大分量来选择参考矢量，并将该矢量存储为两位以进行解码。借助此向量，我们可以计算正交向量，并将该向量与切线之间的角度存储为8位。切线和双切线的顺手性都存储在一个位中。这样，我们可以将整个切线空间作为DXGI\_FORMAT\_R10G10B10A2\_UINT存储在第四个几何缓冲区中（图1.6，清单1.6）。应当注意，该方法需要大约一半的指令数才能将切线空间编码为32位，就像以数学上稳定，精确的方式将TBN矩阵转换为四元数并将其打包为32位时一样。

*清单1.6 GPU zen p.102-103*

用这种压缩方法可以实现的质量几乎等同于以3×30位/像素存储未切线，切线和法线（图1.7）.

材质ID应用到延迟材质，每种材质必须存储唯一的ID。 该材料ID以16位的形式存储为第三个几何缓冲区（DXGI\_FORMAT\_R16G16\_UINT）的绿色通道，并支持多达65,536种单独的材料，即使对于大型项目也应足够。

其他顶点属性。 实际上，可能需要其他顶点属性，例如多纹理坐标集和顶点颜色。 只要没有包装存储在几何缓冲区中的纹理坐标（在[0,1]范围内），就应该有可能为每种材质重建仅相对于存储的纹理坐标进行缩放和偏置的其他纹理坐标集。 。 然后可以通过缩放存储的导数来获得适当的导数。

对于所有其他情况，可以将此类网格视为单独的网格类型进行剔除，这使它们可以将其他顶点属性写入其他几何缓冲区。 在着色过程中，将分别渲染相应的材质，如稍后所述，这可以有效地获取其他几何缓冲区。 显然，这只是可行的解决方案，只要附加顶点属性的数量很少或不经常使用此类材料即可。

**材质ID流程** 填充几何图形缓冲区后，通过简单的全屏扫描将存储的材料ID传输到16位深度的缓冲区中。 通过将材料ID除以支持的材料的最大数量并通过像素着色器中的SV\_Depth输出此值来完成此操作。 对于使用相同着色器并在着色过程中一起渲染的标准材质，将输出一个特殊的保留深度值（例如0）。

**着色流程** 在此过程中，将应用材质和照明。 对于使用相同着色器和资源绑定布局的所有标准材质，将执行一次全屏传递，并借助来自几何缓冲区的材质ID将索引动态索引到公共着色器-资源描述符表中，以获取所需的纹理。

但是，在Ubershader中一起处理需要不同着色器路径的材料是一个坏主意，因为相邻的屏幕像素可以完全使用不同的着色器路径。这导致具有较差性能特征的动态分支。相反，对于每种非标准材质，将渲染一个屏幕空间矩形，该矩形包围使用该材质的所有可见网格的边界。矩形顶点的深度设置为与在材料ID传递中为此材料输出的深度相同。通过将深度比较设置为相等，早期的深度模板测试将防止进一步处理具有不同材质ID的像素。由于使用了与材料ID检验中输出的相同的二进制深度值，因此等深度测试不会产生任何精度问题，并且可以避免错误的材料分类。常见渲染的标准材质使用相同的剔除技术，以防止使用非标准材质的像素通过使用与材质ID传递中输出的相同的特殊保留深度值来处理。

屏幕空间矩形边界的计算是在GPU上完成的。为此，将按截断法在CPU上筛选的所有网格实例的索引按每种材料存储在连续的GPU缓冲区中。每个材料的第二个GPU缓冲区将第一个索引和最后一个索引存储到前一个缓冲区中。另外，必须为第一和第二遮挡通道使用单独的可见性缓冲区，以便能够检查每个实例的可见性。分派了一个计算着色器，该着色器针对每种材质运行一个线程组，其中每个线程处理一个网格实例。如果可见网格实例，则将其OBB投影到屏幕空间中，并在共享线程组内存中扩大边界矩形。最后，每个线程组在结构化缓冲区中针对每种材料特定的位置输出膨胀的边界矩形（清单1.7）。因此，将在具有SV\_VertexID的顶点着色器中手动获取此结构化缓冲区，以为每种处理的材质在剪辑空间中构造边界矩形的角。

*清单1.7 GPU zen p.105-106*

尽管早期的深度模板测试会拒绝使用不同材质的像素，但当前的消费类GPU会以32或64线程的扭曲运行，并使用辅助像素来确保像素以2×2像素进行处理以进行导数计算。 因此，即使不相关像素被拒绝，也可能存在大量已处理的辅助像素和无效的GPU线程。 但是，我们还在OpenGL中实现了渲染原型，并借助OpenGL扩展GL\_NV\_shader\_thread\_group（清单1.8）测量了活动（内存输出），辅助线程和非活动GPU线程的数量。

*清单1.8 GPU zen p.107*

事实证明，与正向渲染器相比，用于材质渲染和照明的辅助线程和非活动GPU线程的数量显着减少，而使用GPU硬件细分技术则大大降低了。 甚至在每种材料都单独经过处理且经过Alpha测试的叶子数量很多的情况下，也是如此。

GPU硬件细分在“ deferred +”下表现更好的原因是，即使使用自适应细分技术，小三角形的数量也会急剧增加。 随着小三角形的数量增加，用于辅助像素着色的辅助线程和无效GPU线程的数量也会增加。 这使得每个像素的操作效率低下。 由于使用deferred +，大多数每个像素的操作都从几何遍历推迟到具有较高扭曲利用率的后续屏幕遍历，因此小三角形的负面影响就不那么明显了。

我们还尝试了另一种渲染策略。初始几何图形通过后，借助原子计数器，对具有相同材质ID的像素数进行计数，并在公共GPU缓冲区中保留相应的块。然后，将每个像素的屏幕位置记录到具有相应材质ID的块中。之后，通过将像素计数除以每组的线程数，将处理每种材质的像素所需的线程组数写入GPU缓冲区。最后，对于每种材料，发出间接调度命令，从上述GPU缓冲区中获取线程组的数量。在此步骤中，将应用材料和照明。不幸的是，该系统的性能比早期的深度模板方法要慢得多。一方面，将每种材质的屏幕像素进行分档会产生额外的开销。另一方面，阴影像素也比早期的深度模板方法慢。我们认为其原因是，合并像素的空间局部性不匹配所提取的几何缓冲区纹理的纹理记忆模糊模式。因此，我们还尝试将屏幕图块中的像素装箱以提高纹理缓存效率，但无法实现任何显着的性能提升。

如前所述，可以使用平铺的[M. Billeter and Assarsson 2013]或成簇[Olsson等。 2012]方法。

* 1. 与基于分层深度Buffer剔除相比

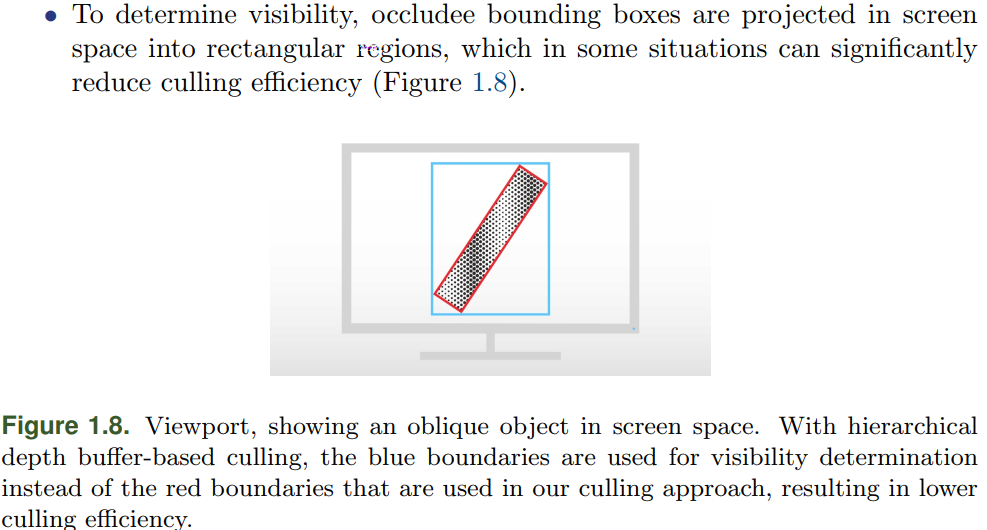
类似的基于GPU的剔除系统已经被提出并用于游戏中[Hill and Collin 2011，Haar and Aaltonen 2015]。 但是，这些系统基于通过保守的下采样生成的分层深度缓冲区。 遮挡边界框被手动投影到屏幕空间中，并针对分层深度缓冲区进行测试以确定可见性。 此方法导致使用建议的剔除系统可以避免的几个问题：

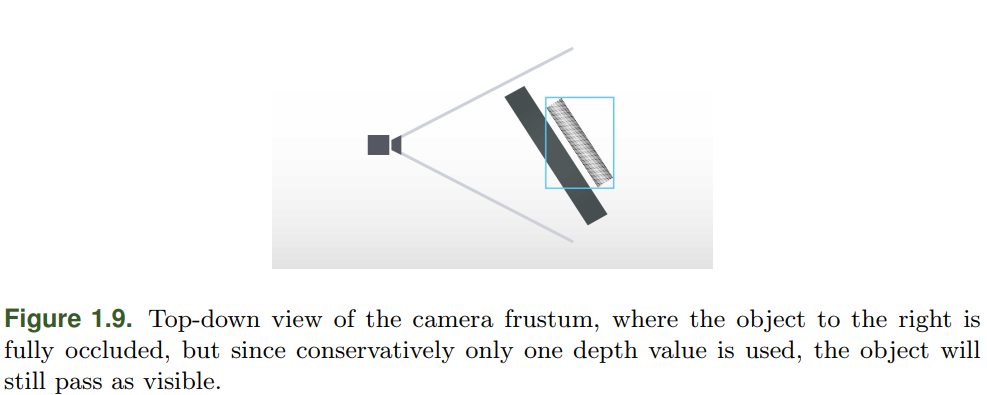
为了确定可见性，遮挡边界框会在屏幕空间中投影到矩形区域，这在某些情况下会大大降低剔除效率（图1.8）

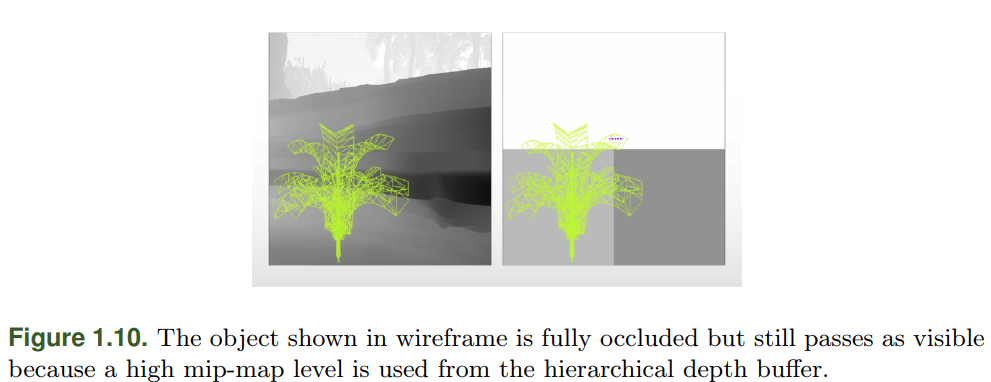
为了确定可见性，每个遮挡物使用一个保守的深度值，在某些情况下会导致误报（图1.9）。

为了能够使用固定数量的纹理样本针对分层深度缓冲区测试每个投影的边界框，对于较大的遮挡物，必须使用更高的mip贴图级别。 这种方法的问题在于，mip-map是通过保守的下采样生成的，即，始终从先前的mip-map级别中获取最大的深度值。 结果，较大的深度值（例如，来自天空的深度）会传播到更高的Mip贴图级别，从而使屏幕空间中较大物体的剔除效率很低（图1.10）。

如果我们将基于分层深度缓冲区的方法用于先前描述的两遍剔除系统，则首先需要从最后一帧的深度缓冲区构建一个分层深度缓冲区，然后在主遮挡通过之后，使用 当前帧的更新深度缓冲区。 在无法访问深度缓冲区的Hi-Z切片的PC上，与淘汰系统的其余步骤相比，这会产生较高的性能开销。







前三个问题可以通过将网格资产细分为小集群来缓解[Haar and Aaltonen 2015]。但是，在我们的案例中，我们需要一种不依赖网格聚类并且与传统资产管道兼容的剔除系统。由于使用Dawn Engine当前资产管道构建的场景由相对较小的模块化块组成，因此使用建议的剔除系统，我们可以避免引入将网格细分为小型群集的系统.

与基于分层深度缓冲区的剔除方法相比，我们在不使用网格聚类的自然丛林环境中比较了建议的剔除系统的性能。 使用提出的剔除系统，我们平均可以实现2.3倍的较高剔除速率和1.6倍的更快帧时间。

* 1. 利弊

以下小节概述了所建议系统的剔除和渲染部分的利弊。

* + 1. 剔除

利

* 达到与传统硬件遮挡查询相同的像素精度，同时消除了延迟问题（弹出）.
* 支持高度动态，复杂，经过Alpha测试的遮挡体，而无需编写和渲染专用的遮挡体几何形状.
* 无需为模块化组合场景的网格聚类，即可实现高剔除效率，从而保持与标准网格资产和资产管道的完全兼容性。
* 该系统与基于CPU的剔除系统（例如，视锥剔除，门户剔除等）兼容，可以将其视为粗略的预过滤。
* 剔除系统本身的性能开销较低。
* 大幅减少了绘图调用的次数，即使使用DirectX 12和Vulkan等开销较小的图形API，也可以提高性能。

弊

* 由于间接绘制缓冲区是在计算着色器中并行生成的，因此绘制命令不再以确定的顺序进行。 因此，几乎共面的表面更有可能引起Z角碰撞，应该避免。此外，不再给出绘制调用的深度排序，从而导致更高的透支。 但是，由于使用deferred +，几何遍历重量轻且剔除效率高，因此与传统渲染系统相比，对性能的负面影响要小得多.
* 在摄像机从一帧急剧变化到另一帧的情况下（例如，采用隐形传送），剔除效率可能会在短时间内显着下降
  + 1. 渲染

利

* 由于几何遍历轻巧，因此不再需要深度预遍，对于具有较高三角形数的网格，GPU硬件细分，alpha测试或顶点着色器蒙皮来说，深度前遍可能是昂贵的.
* 与使用群集前向阴影相比，在应用材料和照明时翘曲利用率要好得多[Olsson等。 2012]。 因此，小三角形的问题要少得多，并且GPU硬件细分性能要好得多。
* Deferred +是一个统一的渲染系统，与延迟渲染相比，它可以有效处理多种材料。
* 几何处理与材质渲染和光照完全脱钩，从而减少了着色器排列，并缩短了游戏制作的迭代时间。
* 通过将几何处理与着色分离，可以显着减少GPU资源的切换。
* 与延迟获取顶点属性的系统相比[Burns and Hunt 2013]，几何信息每帧仅以缓存友好，一致的方式获取一次。
* 不需要像延迟渲染那样将压缩的纹理数据解压缩到GPU内存中，因此可以显着减少纹理内存的带宽。
* HDR纹理不再像延迟渲染那样成为问题。
* 修改后的几何缓冲区包含普通延迟渲染没有的信息：
* 纹理坐标导数可用于修复延迟贴花的Mip-mapping问题。
* 顶点法线可用于增强屏幕空间环境光遮挡技术。
* 顶点切线可用于各向异性照明。
* 拟议的渲染系统不依赖于特定于供应商的图形功能，并且与支持DirectX 12的整个图形硬件范围兼容。 当动态索引纹理的支持范围太低时，应用程序仍会退回到单独渲染常用材质的角度。

弊

* 与传统渲染技术相比，顶点属性受到的限制更大。
* 透明对象必须单独处理。
* 抗锯齿仍然很难处理。
  1. 结果

为了捕获结果，我们使用了Deus Ex游戏中的两个场景：Mankind Divided，我们将其转换为可以在基于DirectX 12的实验框架中加载和渲染的格式。第一个场景的亮度为1024，第二个场景的亮度为 256倍，使用聚类照明移动球形区域照明。 为了模拟第一个场景的动态对象，每个面光源的光源都渲染为一个发射球体。 每种材料都使用带有可选alpha蒙版的漫反射纹理，正常纹理，镜面纹理和粗糙度纹理。 对于漫射照明，使用一个简单的Lambert术语，而镜面照明则使用GGX微面模型。 这两个术语都适用于球形区域光。 间接照明使用简单的恒定环境项。 在GPU剔除之前，在CPU上执行截锥体剔除。 测试机器使用NVIDIA GeForce GTX 970显卡，屏幕分辨率设置为1920×1080。

为了捕获剖析结果，从一个视点渲染了第一个场景，其中在锥台视图中处理了23,116个实例，分布在4073个网格，5,556,614个三角形和316个材料上（图1.11）。 为了在使用GPU剔除时能够将我们的渲染系统与参考群集前向渲染器进行比较，除发射球体材质外，所有材质都使用相同的着色代码。 表1.1和1.2比较了deferred +与参考群集前向渲染器的时序； 表1.3和1.4列出了详细的GPU时序和剔除系统的效率。

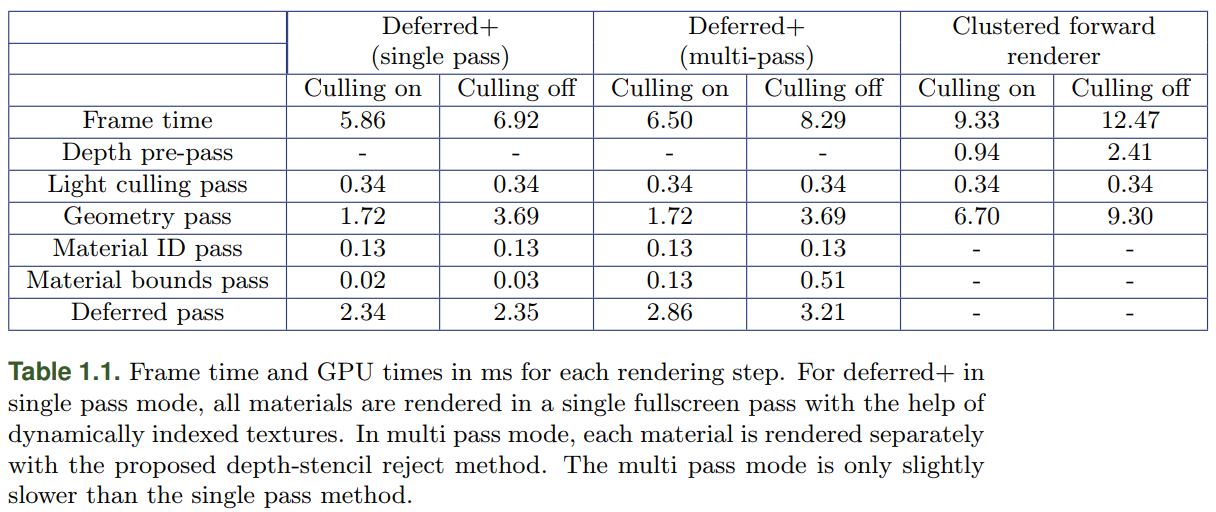


图1.12在第二个场景中将deferred +的视觉渲染结果与参考群集正向渲染系统的视觉渲染结果进行了比较。 我们使用8倍各向异性纹理滤波来确保deferred +中纹理坐标导数的压缩正常工作。

捕获的结果表明，deferred +的运行速度比群集的正向着色更快，尤其是在使用GPU硬件细分的情况下，同时在质量上几乎可以得到等效的结果。 在具有更复杂的材质（使用4种以上的纹理）和更复杂的照明（不同的灯光类型，阴影贴图）的真实游戏条件下，deferred +的性能优势将更加突出。

