# 无层次加速结构的十亿点云渐进式实时渲染

22021006 曾浩辉

1. **点云及其渲染方式介绍**

点云是由大量彩色的点组成的3D模型，通常是通过用各种类型的3D扫描仪扫描现实世界或通过基于图像的重建方法获得的。与基于mesh的模型不同，基于mesh的模型可以用纹理来经济且高效地表示顶点之间的附加细节，而基本的点云模型用单个点来表示所有的表面细节。点没有连通性，这很容易导致它们之间明显的间隙，除非它们被更多的点或覆盖点填充。因此，即使看起来很小的场景也得是由数百万个点组成的，甚至更大的模型可以由数千亿个点组成。并且由于缺乏支持细节层次(LOD)的标准分层点云文件格式，点云以顺序格式分布，如LAS[1]及其压缩版本LAZ[2]。点云处理和渲染的程序应该会构建自己的层次结构，但这需要不少时间，对特定层次结构的甚至只会支持创建它的应用程序。

渲染大型点云的研究传统上侧重于分层加速结构的生成和使用，该结构允许系统加载和渲染对输出影响最大的最小部分数据。然而，这些结构的生成是缓慢且耗时的，因此不适用于诸如快速查看以广泛使用的非结构化文件格式存储的扫描数据或立即显示点云处理任务的结果的任务。

用于渲染大型点云的数据结构通常不同于用于处理大型点云的数据结构。渲染要求数据结构能够根据查看者的位置和方向快速访问模型的不同细节级别。另一方面，处理通常需要数据结构提供对某个区域中所有点的快速访问，而不考虑细节层次。点云处理框架OPALS使用一个kd树，其中所有的点都存储在叶节点中。只不过这不会提供对详细级别数据的访问，但提供对所选区域内数据的高效访问。虽然OPALS可以快速修改、过滤和扩充一个区域中的所有数据，但它不能快速显示结果，做不到实时。使用最先进的方法，无论何时生成新结果，渲染结果都需要冗长的预处理步骤来生成分层结构[3]。

1. **传统方法的问题**

传统的分层次渲染方式会有好几个问题，使得这种渲染方式变得复杂起来：

点属性的数量。大多数点云至少包含一个XYZ坐标和一个颜色值或一个标量值，它们有不同的含义。这种基本格式每点至少消耗16个字节。然而，一些用例需要大量额外的每点属性。可能的属性包括强度、反射率、分类、返回数、扫描角度、GPS时间、回波比、光束方向、表面法线等。这会将存储需求增加到超过100字节/点。存储所有这些属性会对加载、处理和渲染时间产生负面影响，即使实际上只需要少量属性。

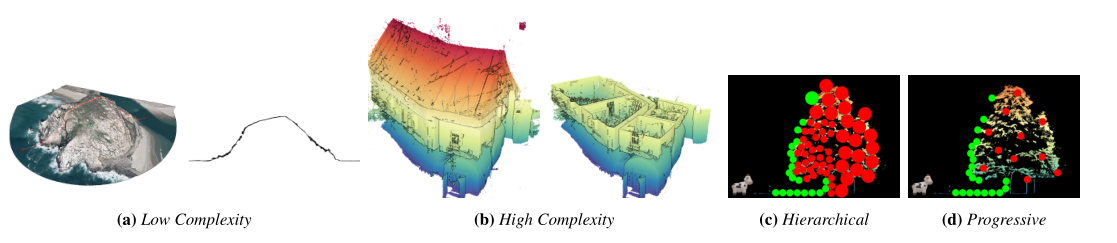


图1

深度的复杂性。点云层次结构的主要问题之一是它们不能处理深度复杂的模型——特雷迪尼克等人[4]是一个显著的例外。高深度复杂性意味着模型由相互遮挡的多层表面组成，如图1所示。被遮挡的点占用了大部分可用的点预算(一帧中应该渲染的最大点数)，而不会对图像产生影响。因此，模型要么以较低的细节级别呈现，因为没有更多的预算留给更高的级别，要么预算增加，这降低了性能。高复杂性模型的例子包括对建筑物内部、密集植被和高噪声数据的扫描。在有内部的建筑物中，大多数房间被墙壁、地板和天花板遮挡。所有这些被遮挡的房间仍然需要渲染，因为它们可能通过扫描点或孔之间的间隙部分可见。由于点密度不规则，特别是沿着激光扫描仪的扫描线和在扫描线之间，并且因为许多间隙是扫描仪没有获得的缺失数据的结果，所以调整点大小以缩小间隙是有问题的。

LOD建立时间。生成LOD结构需要时间，缺乏标准格式意味着每个支持LOD结构的应用程序都使用自己的格式。两种最广泛使用的分发格式，劳工会计制度和LAZ，是无等级的。建立层次结构的速度大约为每秒5万到100万个点(见[5，6])，这意味着我们需要300到6 000秒才能探索出一个有3亿个点的点云。通过我们的方法，我们能够在3到10秒内完全加载和渲染3亿个点，这取决于文件格式。3秒指的是与顶点缓冲区匹配的理想格式，而广泛使用的LAS格式需要10秒。

这些问题都使得传统方法要花费大量时间来进行预处理，增加不少渲染时间。

因此，一种新的渐进的方法从此诞生，能够实时渲染任何适合GPU的内存的点云，而不需要预先来生成分层加速结构。这种方法支持每个点，尽量避免浮点数运算以及乘法运算，尽量用整数来计算，可以采纳的方法比如说Bresenham算法，用的仅仅是加法、减法、移位运算，当然这算法到现在硬件发展的情况也是有作用。当初像素较少，所以在光栅化的时候，要把抗锯齿的作用做到最好。

1. **渐进式渲染方法**

一种新的渐进的渲染方法，可以实时渲染内存中的点云，而且还不需要分层结构，最多可测试渲染10亿个点。在每一帧中，渐进式方法通过渲染来填充漏洞点云的随机子集，这导致相对均匀地收敛到完整图像。得使用基于素数的伪随机数发生器然后在GPU上增加并行创建这些随机点，该伪随机数发生器在给定范围内生成唯一的整数值。我们的方法允许实时呈现已经加载的数据，同时仍然从磁盘加载剩余的数据。对于广泛使用的LAS点云文件格式，它实现了高达37M点/秒或1GB/秒的磁盘到GPU传输速率，对于匹配GPU顶点缓冲格式的简单二进制文件，实现了高达100M点/秒或1.6GB/秒的传输速率。它是为具有大量属性的点云量身定制的，并支持这些点云，经过测试，每个点最多有50个属性和107个字节。

虽然这种方法允许用户在没有预处理的情况下实时渲染任何可以进GPU内存的点云，但它不允许用户渲染比10亿更大的点云。几乎没有GPU内存的低端设备将不得不使用核外结构。然而，许多点云查看器目前显示原始的非结构化数据，这些数据需要与我们的渐进式方法大致相同的GPU内存。就性能而言，虽然我们的方法增加了一定的开销，但是对于开销大得多的点云都可以在此方法执行下表现得更好，即使在低端设备上也是如此。

渐进式渲染意味着要将渲染整个点云分布在多个帧上，而不是在单个帧中完成所有工作。目标是保持实时帧速率，并保持应用程序始终响应。实现这一目标的基本想法是重新投影前一帧，因为大多数以前可见的点可能在当前帧中再次可见，然后用随机选择的附加点填充由于不相交而出现的洞，以获得高质量的收敛行为。在多个帧的过程中，结果收敛到完整模型的图像。随机选择的填充孔的点的数量被称为点预算，类似于分级方法，分级方法是指从分级中选择并在帧中渲染的点的数量。在我们的渐进式方法中，可以调整预算，使其更有利于性能(低预算)，而不是更快地收敛到完整映像(高预算)。重新投影的点数不包括在预算中，因为重新投影具有不可调整的固定成本。

3.1数据结构

我们的方法采用两种数据结构，以便将新的属性数据流式传输到图形处理器，并在每帧中快速呈现一定数量的随机点。

在中央处理器方面，点属性以数组结构的方式存储，即一个数组只存储一个属性:[RRR][GGG][BBB]。这允许我们以最少的内存带宽使用将特定的属性从CPU流式传输到GPU，因为访问属性数组的值会将后续值的整个缓存行加载到CPU缓存中[7]。另一方面，交错阵列会导致将一个点的各种不同属性加载到CPU缓存中，如果只需要一个点的属性，这是没有用的。这很重要，因为我们在CPU内存中保持属性的原始形式(例如双精度或64位整数)，并且只有在切换到适应GPU的格式(例如浮点)时才会将其转换为GPU适应格式。在从点云的最大尺寸到属性大小乘以点的个数的字节数的转换过程中，数组的结构需要更少的内存带宽。我们保留属性的原始形式的原因是，它们中的许多属性以一种对渲染不直接有用的格式存储，但是它们包含的所有数据可能都很重要。例如，在我们的测试数据集中，每个通道需要2个字节的RGB数据，为了渲染的目的，先发制人地将其减少到每个1个字节会导致稍后可能需要的数据丢失。

在GPU端，我们使用每点16字节的打乱的内部顶点缓冲区作为渲染数据结构，它是通过在伪随机位置插入点来创建的。由于现代图形处理器的最大缓冲区大小为231字节，混合顶点缓冲区对象(VBO)实际上可能由多个缓冲区组成，每231/16≈1.34亿个点对应一个缓冲区。打乱之所以需要做，是因为它减少了连续N个点的渲染问题。每个点包含12个字节的XYZ坐标，以及另外4个字节的属性数据。属性数据可以包含一个4字节的浮点数，或者四个无符号字节。前者用于可视化单个标量属性值，后者用于可视化属性向量，如颜色和法线。顶点着色器根据需要解释数据。新加载的一批点数或新的一批属性不会直接上传到洗牌后的VBO。相反，它们被上传到一个单独的分布式缓冲区，该缓冲区包含一批500000个点。然后，计算着色器将点或属性插入到VBO中相应的位置。分布式缓冲区在最初从磁盘加载时接收16字节的XYZRGBA，但在切换到新属性时，每个点只有4字节，即只有属性数据。最后，重投影缓冲区包含在一帧结束时可见的所有点。除了位置和属性数据之外，它还将该点的索引存储在VBO中，这是在重新投影期间将点索引和点颜色一起写入帧缓冲区所需要的。

3.2加载方式

我们方法的目标之一是在加载剩余数据时实时显示中间结果。为了实现这一点，文件加载和转换并行处理准备好缓冲区，主线程的任务被简化为向GPU发送准备好的批处理。图2用时间线说明了这个过程。加载线程专用于从磁盘批量读取500k个点的二进制数据。三个额外的解析器线程转换二进制批处理，并将交错的点数据分成每个属性一个数组，然后将其附加到主内存中的数组结构中。在下一帧开始时，主线程将前一帧中完全加载和解析的所有批处理的XYZRGBA属性发送到GPU。复合XYZRGBA数组是一种特殊情况，在所有其他属性存储在单独的数组中之后，解析器线程会组装它，因为这是我们发送给GPU的初始数据。

全尺寸的顶点缓冲区是在开始时分配的，顶点将被默认初始化，直到所有的点都被加载。这些点将在位置(0，0，0)处呈现为黑点。

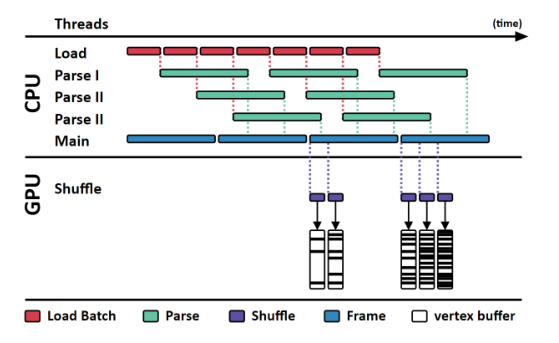


图2

3.3渐增的并行打乱方法

与按原始顺序和顺序渲染点相比，渲染随机选择的点可以提高收敛到最终图像时的视觉质量。在加载过程中，点被混洗，这样我们就可以通过渲染顶点缓冲区中N个连续点的子集来有效地渲染N个随机点。因为我们希望在从磁盘加载额外的点的同时，用渐进的方法显示这些点，所以我们需要使用一种打乱方法，这种方法能够逐渐地将点打乱，使它们变得可用。我们使用Preshing描述的方法来计算一系列数字[0，...，P1]，其中P是与3 (mod 4)全等的素数。这种方法将序列中的每个数字映射到同一个集合中的另一个数字，没有冲突，即没有重复。在我们的例子中，我们假设输入是原始点数组中该点的索引，输出是该点在混洗数组中的位置。这允许我们用计算着色器直接将点复制到它们在混洗数组中的位置，而不需要线程之间的同步。置换函数由下式给出:

(1)

最后一种情况包括点数N不等于合适素数的点云。在这种情况下，我们找到下一个较小的素数P ≤ N，洗牌该范围内的所有点，剩下的点不洗牌。因为连续质数之间的差距很小，所以未抽点数可以忽略不计。两个连续的素数P ≡ 3(mod 4)之间的最大差距高达5亿分，介于素数184007671和184008203之间。这意味着，对于高达5亿点，最多532点可能不会被打乱。试图将它们也混在一起会导致额外的工作，并且没有显著的改进。或者，可以找到下一个更大的素数，对整个数据集进行洗牌，并留下可忽略不计的顶点缓冲元素，这些元素是空的。基于素数的方法的一个缺点是仅经过一遍后置换的质量相对较低，这表现为明显的模式，如图3所示。

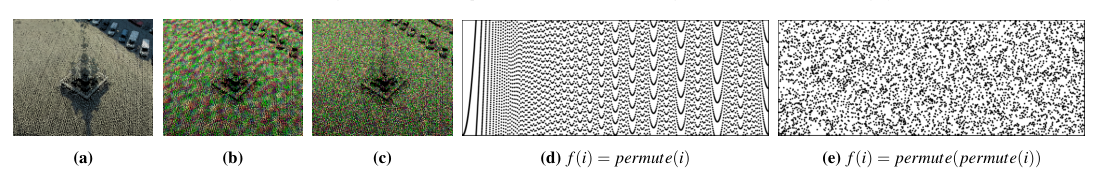


图3

因为等式1是双射的——输入集合[0，1，2，...，P1]映射到同一个集合中恰好一个不同的元素–我们可以简单地多次应用它，并且仍然获得相同数量的唯一目标索引。因此，我们的最终混洗函数给出如下:

(2)

应用两次permute会导致不一定是高质量的随机性，但对于我们的渐进渲染方法来说足够随机。“不是高质量的”是指有一定的模式，一些随机数可以从以前的随机数中预测出来。例如，等式1对于前数是单调递增的，等式2对于前数是单调递增的。前者在图4 (d)中非常明显，同样的模式在整个函数图中重复出现。后者在图4 (e)中不明显。只要模式在视觉上不是立即显而易见的，我们就认为随机数生成器对于我们的方法来说足够随机。基于素数的方法相对于其他方法(如Fisher-Yates shuffle)的最大优势在于，它可以单独应用于每个输入索I，而不依赖于以前计算的状态，也没有冲突。因此，它本质上是可并行的，可以在GPU上的计算着色器中实现，而无需线程之间的同步。

3.4渲染管线

渐进式渲染方法将前一帧重新投影到当前帧，然后通过渲染一定数量的随机点来填充丢失的数据。在多个帧的过程中，结果将会收敛到我们通过一次渲染所有点得到的相同图像，不考虑渲染顺序和z-fighting问题。此方法实现了三个渲染过程:1.重投影:渲染前一帧中可见的所有点，重投影到当前帧。2.填充:渲染一批随机点来填充孔。这是通过渲染乱顶点缓冲区的子集来有效完成的。3.准备:从渲染图像中可见的所有点创建新的顶点缓冲区。这个顶点缓冲区将用于传递下一帧。

3.5传入点的属性

我们的数据集由具有数亿个点的点云组成，每个点有多达107字节的多达50种不同属性。图4显示了点云可以包含的各种属性。假设10GB的GPU内存和每点107字节，我们最多可以在GPU上存储10 \* 10243107 = 100M个点。在大多数情况下，我们在任何时候都只需要坐标加上1到4个属性，所以剩下的属性不必要地消耗内存。除此之外，我们的渲染管道还受到内存带宽的强烈影响，因为每个帧都会重新计算顶点缓冲区。一个顶点的字节越多，生成一个新的顶点缓冲区就越慢。因此，我们在GPU内存中只保留每个点16个字节，包括XYZ坐标的3×4 = 12个字节和编码一到四个属性的另外4个字节。这允许我们在10GB的GPU内存中存储高达10 \* 1024316 = 671M的点，实际点的数量会更低，因为应用程序的其他部分以及其他应用程序和操作系统也需要一些GPU内存。

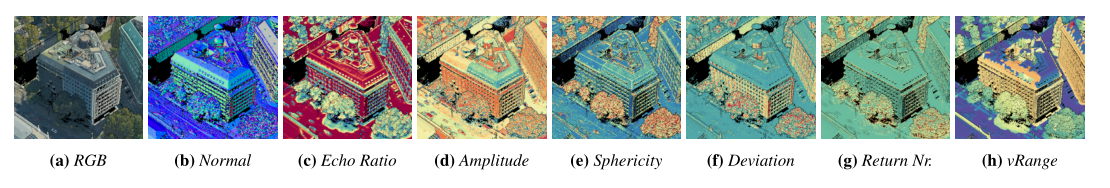


图4

1. **结论**

通过将自己限制在XYZ和RGBA值，将点以与我们在GPU顶点缓冲区中使用的相同格式存储在磁盘上，并在没有任何修改的情况下直接将缓冲区从磁盘传输到GPU，我们能够实现高达每秒1亿点的负载性能。坐标存储为单精度浮点值，颜色存储为无符号字节。每个点需要16个字节。从磁盘到GPU的最终传输速率为1.6GB/s。这些数字还包括转移数据的时间。根据用户基准测试套件，所用固态硬盘的可实现读取性能为2.5 GB/s。这使得我们实施的磁盘到GPU性能(1.6GB/s)为理论上可实现的磁盘到RAM性能(2.5GB/s)的64%。

渐进方法能够在HTC VIVE要求的不同视点下保持每秒2×90帧，速度如此之快可以做到虚拟现实的要求，在4xMSAA的RTX 2080Ti上每只眼睛1448 × 1608的分辨率，以及在数据完全加载后每帧300万点的固定填充预算。对于维也纳数据集，如果固定填充预算降低到100万个点，则在加载期间也可以实现帧速率目标。在虚拟现实中，整个渐进渲染管道执行两次，每只眼睛一次。由于帧速率被锁定在90帧/秒，填充预算为300万点，图像以每秒1.8亿点的速率收敛。模型的点数会影响收敛时间，但它不会影响渲染性能，因为我们在一帧中每只眼睛最多渲染的数量为重映射点的数量加上填充点数量。由于头戴式显示器的姿态总是逐帧变化，即使它坐在桌子上，由于跟踪噪声，结果将接近但永远不会真正达到收敛。在4xMSAA和1448x1608的分辨率下，每只眼睛可重投影的最大点数为4×1448 1608 = 9.3M。

缺点：这种方法目前仅在GPU内核中。完整的数据集必须适合CPU内存，位置数据和选择的属性必须适合GPU内存。GPU需要每点存储16个字节，但CPU需要存储所有属性，以实现属性的快速切换。然而，也可以选择将属性数据保存在磁盘上的单独文件中，以快速将它们流式传输到GPU，而不需要将它们保存在RAM中。渐进方法是为没有空间加速结构的数据开发的。然而，没有加速结构会增加收敛的持续时间，因为能使用平截头体剔除或LOD剔除来减少最可行候选的点数。未来的工作可能会探索在加载过程中创建简单加速结构的可能性，或者在之后并行创建，以提高运行时的性能和质量。目前没有提供成本效益高的质量改进方法。

1. **参考文献**

[1]ASPRS. LAS Specification 1.4 - R14. Rev. 14. The American Society for Photogrammetry & Remote Sensing (ASPRS). Mar. 2019 2,8.

[2]ISENBURG, MARTIN. “LASzip: lossless compression of LiDAR data”. Photogrammetric Engineering & Remote Sensing 79 (2013). DOI:10.14358/PERS.79.2.209 2, 8.

[3]Markus Schütz, Gottfried Mandlburger, Johannes Otepka, Michael Wimmer,Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures,EUROGRAPHICS 2020,DOI: 10.1111/cgf.13911

[4]TREDINNICK,R.,BROECKER,M.,and PONTO, K.’’Progressive feedback point cloud rendering for virtual reality display’’. 2016 IEEE Virtual Reality (VR). Mar. 2016, 301–302. DOI: 10.1109/VR.2016.7504773 2, 3, 12.

[5]SCHEIBLAUER, CLAUS.’’Interactions with Gigantic Point Clouds’’. PhD thesis. Favoritenstrasse 9-11/186, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2014.

[6] WAND, MICHAEL, BERNER, ALEXANDER, BOKELOH,MARTIN, et al. ‘’Interactive Editing of Large Point Clouds’’. SPBG.2007 2, 3.

[7] DREPPER, ULRICH. ‘’What Every Programmer Should Know About Memory’’. (2007) 4.