高效稀疏体素八叉树--分析、扩展和实现

Samuli Laine　　Tero Karras
英伟达研究院*

摘要

本技术报告扩展了我们之前关于稀疏体素八叉树的论文。我们首先讨论了体素表示法的优点和缺点，以及不同类型内容对存储空间的要求。然后，我们详细解释了我们用于存储体素的紧凑型数据结构，以及利用这种结构的高效光线投射算法，包括原始论文的贡献：额外的体素轮廓信息、用于存储高精度对象空间法线法线压缩垫、用于平滑遮光块的后处理过滤技术，以及用于加速光线投射的光束优化。

更详细地介绍了内存和磁盘中的体素数据管理，以及体素层次结构的构建。我们在结果部分做了大量工作，提供了测试案例的详细统计数据。最后，我们讨论了在将体素作为通用内容格式广泛采用之前需要克服的技术障碍和问题。

我们的 voxel 代码库开源，可在以下网址获取
http://code.google.com/p/efficient-sparse-voxel-octrees/

1　导言

体素可被视为三角管道的一种更简单的替代方案，而三角管道在当前的 GPU 中已变得相对复杂。传统上，体素被用于表示核磁共振成像扫描等体积数据，但在本文中，我们专注于体素用作不透明表面的高密度采样表示。与通常的看法相反，体素数据并不需要是体积数据。

使用三角形的一个令人信服的理由是，它在表示平面时非常紧凑。在表示建筑物等平面物体时，无论其空间尺寸有多大，都只需要少量三角形。如今，这一优势已不那么重要，因为内存消耗已被逼真外观所需的色彩纹理和法线贴图所占据。位移贴图可用于获得真实的几何细节，但只有最新的 GPU 才能高效地对其进行光栅化处理。位移几何图形也比平面三角形更难进行光线追踪。

人们习惯反复使用相同的纹理，以GPU 内存。*id Software* 在其 *id Tech 4* 引擎中开创性地将单个大型纹理用于地形，而当前的 *id Tech 5* 引擎则将这一技术扩展到了所有纹理。这种所谓的巨型纹理只有一个子集保存在内存中，缺失的部分会在需要时从磁盘流出。

e-mail：{slaine,tkarras}@nvidia*.com

本技术报告是对 2010 年 I3D 论文集[Laine 和 Karras 2010]中发表的 "高效稀疏体素八叉树 "的修订和扩展。

英伟达™（NVIDIA®）技术报告 NVR-2010-001，2010 年 2 月。

假设这种巨型贴图在未来将变得非常普遍，我们需要为所有表面的每个分辨率样本存储一个颜色值。如果使用巨纹理位移贴图来实现更高的几何复杂度，我们实际上还需要为每个样本存储一定量的几何数据。说到这里，我们不禁要问，为什么要粗略的几何图形（基础网格）和精细的细节（颜色和位移贴图）分开呢？如果我们无论如何都要为每个分辨率样本存储颜色和几何数据，为什么不使用一种更简单的表示方法，将相同的数据结构用于这两个目的呢？

数据分辨率。流式数据的目标是实现恒定的每像素像素比，以便始终有足够的分辨率用于图像合成。为便于讨论，我们将其称为数据的*目标分辨率*。它可能不同于存储介质上可用的分辨率（*存储分辨率*），也可能因数据而异。例如，地形的目标分辨率在远处比在摄像机附近要小。如果存储分辨率小于目标分辨率，合成图像就会比预期的粗糙。在这种情况下，必须采取措施避免体素阻塞变得明显。

不变形变形可能是将基本网格与细部网格分开的最重要好处。基础网格的变形通常很容易，理想情况下，细部图层也会很自然地跟着变形。如果做不到这一点，就需要使用其他变形方法，但这些方法都不如基于顶点的变形方法直观和强大。因此，目前似乎只有在场景的静态部分才可行。第 3.4 节简要讨论了基于体素的几何体变形的可行方法，但在本文中，我们将只讨论非变形几何体。

2　以前的工作

有关体积结构可视化的文献浩如烟海，因此我们将重点关注与我们的工作最直接相关的论文。我们特别省略了仅限于高度场的方法（如最近的一篇论文[Dick et al. 2009]）或基于光栅化和着色器中每像素光线投射的组合的方法（见[Szirmay-Kalos and Umenhoffer 2008]的精彩调查），因为这些方法无法执行通用的光线投射。

Amanatides 和 Woo [1987] 首次提出了规则网格遍历算法，该算法是包括我们在内的大多数衍生工作的基础。该算法的思路是计算沿每个轴线的下一个细分平面的 t 值，并在每次迭代中选择最小的一个来确定下一步的方向。

Knoll 等人[2006]提出了一种算法，用于对包含体积数据的八叉树进行光线追踪，这些数据需要使用不同的等值面层次进行可视化。他们的方法在概念上类似于 kd 树遍历，首先确定子节点的顺序，然后对子节点进行再粗略处理，从而以分层方式进行。这种算法并不适合在 GPU 上实现。Knoll 等人[2009]给出了对相干光线束的扩展，即选择一个主轴，在体积传播与光线束相对应的体素切片。其他数据包遍历算法一样，该算法的优点在于

基于 CPU 的实现似乎不太可能轻松转换到大规模并行 GPU 上。

Gobbetti 等人[2005]讨论了在图形硬件上渲染体素数据集的框架。他们的渲染是通过绘制反锯齿点基元来利用图形流水线的，因此不支持光线追踪。不过，他们的系统没有采用通用着色方法，该方法基于从不同方向对原始数据进行采样。然后，他们在每个象素的基础上选择最能表达观察结果的着色器，并根据观察结果计算着色参数。这样的方法在实际应用中非常有用，因为无需手动调整，不同 LOD 级别的着色都能自动处理。

Crassin 等人[2009]提出了一种基于 GPU 的体素渲染算法，该算法结合了两种遍历方法。第一阶段使用 kd-restart 算法对规则八叉树进行射线投射，以避免堆栈。八叉树的叶子是包含实际体素数据的砖块，即三维网格。当找到一个砖块时，就会沿着射线对其锥体进行采样。砖块通常包含 16^3 或 32^3 个体素，除了真正的体积数据或模糊数据外，会浪费大量内存。另一方面，硬件支持的 mipmapped 3D 纹理查找使砖块采样非常高效，而且采样结果会自动进行抗锯齿处理。该算法的一个有趣特点是 CPU 和 GPU 之间的数据管理。在遍历八叉树时，渲染器会检测 GPU 内存中是否缺少数据，并向 CPU 发出信号，CPU 会将所需数据流输入。这样，任何时候都只需在 GPU 内存中保留一部分节点和砖块。

我们的方法在几个方面与 Crassin 等人的方法不同。我们的目标是在 GPU 内存中存储大规模场景的表示，并为高质量渲染提供足够的细节，这就需要占用较小的内存空间。此外，我们的主要兴趣在于表示曲面而非体积，因为曲面在大多数真实世界的内容中扮演着更重要的角色。基于这些考虑，我们使用了单一的分层结构，而不是粗略数据和精细数据的独立方案。我们还引入了额外的轮廓数据，以便在单个体素内精确放置曲面。

3 三角形与体素

在本节中，我们将讨论基于三角形和基于体素的几何表示法之间的主要区别和相似之处。对于基于三角形的表示法，我们假定一个（粗糙的）基本网格配有独特的颜色纹理和位移贴图，以实现几何细节。这样，这两种表示法在表示静态、高度去尾的物体时，功能大致相同。不过，正如我们稍后将看到的，二者在其他方面存在很大差异。

3.1 内存使用情况

体素通常被认为会消耗大量内存。但如果我们只对表面进行编码，内存使用量就会下降到 $O(N^2)$，其中 N 是沿一个空间维度的分辨率。例如，分辨率可以定义为每单位长度上可区分特征的最大数量。就复杂度而言，这与三角形的情况相同，假定纹理是唯一的，总表面积也是影响内存使用量最大的因素。

三角形我们假设基础网格足够粗糙，因此可以完全忽略它们。剩下的就是颜色

	三角形	**象素** 带轮廓
形状	1	1
颜色	0.5	1
正常值	0	2
轮廓	-	1
总计	1.5	5

表 1：*唯一纹理、位移三角形和体素的每个体素内存用量。所有数字均以字节为单位。*

和位移细节贴图。假定使用 DX 纹理压缩，每个采样可以使用 4 比特对颜色进行编码，对于三轴位移，使用 DXT5 压缩（例如 van Waveren 和 Castan˜o，2008 年），每个采样 8 比特就足够了，总计 1.5 字节。不需要法线，因为可以从位移图中推导出法线。

体素下文第 5.6 节将全面分析体素的内存消耗情况。总的来说，每个体素加上颜色、符号和轮廓（第 5.2 节）大约需要 5 个字节。因此，内存消耗是基于三角形表示法的 3.33 倍。表 1 列出了相关数值的总和。如果考虑每个叶状体（即目标分辨率样本）的内存消耗，我们需要将每个叶状体的成本乘以 4^3，得出每个样本需要 6.67 字节。对于三角形，考虑mipmap 也需要同样的乘数，因此每个最高分辨率样本需要 2 字节。

值得注意的是，纹理分辨率无法进行精细控制，但体素分辨率却可以。，如果一个彩色纹理包含一大片恒定的颜色，纹理就不能轻易地利用这一点但通过分层体素编码，就可以简单地省略掉不会产生任何影响更多细节层次。这同样适用于位移贴图，因此适用于几何细节。从这个意义上说，纹理和体素之间直接比较有些粗糙，因为内容的性质决定了体素能从局部分辨率微调中获益多少。

为了估算英伟达 Quardo FX 5800 的 4GB 显存能容纳多少数据，我们假设几何图形的分辨率为 1 mm z 1 mm。这样，在 1080p 显示分辨率（1920 x 1080）下，使用 90° FOV 观看不超过约一米远的内容时，每个像素可获得一个样本。对于三角形，我们可以将 $2^{32}/2 = 2^{31}$ 个样本放入 GPU 内存，从而得到 $2^{31}/10^6 =$ 2147 平方米（≈23000 平方英尺）。对于体素，同样的计算结果为 644 平方米（≈6900 平方英尺）。

请注意，这种并不准确，因为它没有考虑到随着与摄像机距离的增加，对分辨率的要求也会降低。在上述情况下，假设每个像素一个样本就足够了，那么只有在半径为两米的球形区域内才需要全分辨率，之后内存需求将开始急剧下降。第 4 节将详细分析这种影响。

3.2 降采样

对三角形网格进行降采样相当简单，对单个位移和颜色映射进行降采样也很容易。据我们所知，目前尚未解决的一个主要问题是，如何在降采样时将高频几何遮挡（如毛皮、草地、密集光栅）转化为部分透明。此外，在某些情况下，三角形网格也无法恰当地表示降采样数据、

例如，在经过足够的下采样后，毛皮应变成部分透明的体积块。

对体素数据进行降采样理论上非常简单。父体象素的外观需要与其子体象素的综合外观相匹配，并将遮挡和透明度在内。这比基于三角形的通用细节级别更容易实现，而且体积透明度也很容易表示。在渲染过程中，可以有效地决定细节级别。例如，我们的光线投射器始终使用与光线穿越距离相适应的细节级别。

然而，对体素数据进行下采样存在实际困难。首先，在更细的细节中，部分二元遮挡应该在更粗的细节级别中转变为透明。我们还没有尝试过将其纳入我们的构建器中，因此可能会遇到一些无法预见的困难。另外，在对数据进行下采样时，总会丢失一些细节。例如，如果子体素具有高度镜面的 BRDF 和不同的法线，那么父体的 BRDF 就会根据子体素的法线产生多个峰值。拥有不同细节量的 BRDF 是不现实的，因此出现了一个优化问题：如何设置父体 BRDF（或更广义的阴影模型）参数，使其尽可能模仿子体素的组合 BRDF。

我们的生成器通过对输入数据进行空间滤波来计算颜色和法线，因此滤波器的大小是根据体素比例来选择的。在大多数情况下，单独处理每个参数都很有效，但显然也有处理不当的情况。例如，具有高频位移的高镜面表面。当对这样的表面进行下采样时，高频位移就会丢失。补偿损失频率的正确方法是增加 BRDF 的表面粗糙度。考虑到下采样，我们目前的构建器的另一个局限是它不能自下而上地工作，即父数据不是从子数据中推导出来的。这将使 LOD 级别之间更加匹配。

需要注意的是，在使用 mipmapped 法线贴图和具有不同 BRDF 参数的曲面时，也会遇到完全相同的困难，而这两种情况在当今的游戏中都很常见。我们希望目前用于处理这些问题的解决方案也能适用于处理下采样体素数据。

3.3 缩放

反对体素的一个常见理由是，在近距离观看时它们会变得块状。这与位图图像在放大到一定程度后会变得块状的道理是一样的，传统三角形图形中的纹理也是如此。

可以通过过滤数据来解决表面颜色、法线等细节丢失的问题。在基于三角形的图形中，这是在纹理查找阶段通过执行插值获取来实现的。体素也可以进行类似的内插，但成本较高因为数据没有组织成可以轻松处理的二维数组。我们发现，后处理滤波是更好的选择，第 6.3 节将对此进行讨论。，使用体素和使用纹理三角形同样可以获得平滑的纹理表面。

一个更严重的问题是轮廓的遮挡，而这一问题并不容易通过模糊处理来解决。第 6.4 节简要介绍了我们在这方面的经验。不过，第 5.2 节中详细介绍的附加轮廓数据可以在很大程度上减少剪影的阻塞性，而且还具有其他优点，如降低内存使用率和加快渲染速度。这是我们首选的剪影固定方案。

3.4 变形

三角形网格很容易制作动画，只需对顶点应用变换，然后让表面的其他部分跟随变换即可。通过插值顶点位置来混合多种变换是目前制作三角形网格动画的常用方法。

体素数据的动画制作与三角度数据的动画制作有很大不同。体素数据动画化的一个简单方法是为每个动画帧建立单独的数据集。作为一种粗暴的解决方案，这接近于由一系列独立位图帧组成视频序列。原始数据的数量可能过多，但类似于视频压缩的压缩方法可能会将数据集的大小控制在可承受的范围内。一个主要的缺点是只能播放预先录制的动画。总之，这不是一个特别有吸引力的解决方案。

针对更通用的空间数据变形的研究越来越多。模型通常被包围在一个外壳中，外壳经过变形后，物体的形状会尽可能自然，同时到体积守恒等约束条件。变形数据集的渲染是一个值得关注的问题。对于光线追踪，需要对最终变形函数进行反向映射，以便光线可以弯曲，同时模型保持参考姿态。

体素数据变形的主要问题是体素不能轻易从一个地方移动到另一个地方。正因为如此，另一种基于样本的表示方法--*表面劈裂*[Zwicker 等，2001 年；Weyrich 等人，2007 年]--可能更适合可变形物体。花瓣可以很容易地组织成一个标准的 BVH 树，当花瓣根据变形移动时，BVH 树可以在线性时间内更新。所需的 BVH 大小也会随着拼接点数量的增加而线性增长，因此即使内存使用量高于体素，也只是一个固定的乘数而已。

3.5 编写

基于三角形的三维建模有一套成熟的内容流程。自动工具可用于生成纹理的 UV 图集，艺术家可以自由地在模型上绘画，而无需直接修改纹理。

当内容需要几何细节时，通常会采用不同的方法。在这种情况下，通常使用 ZBrush 或 Mud-box 等雕刻软件生成内容。ZBrush 使用类似体素的数据结构，而 Mudbox 则使用具有本地可调分辨率的网格。这两种工具都允许艺术家使用雕刻工具修改几何体，并在模型上绘制纹理。完成后的模型可缩小为低多边形基础网格，并使用法线或位移贴图进行细化。

考虑到体素，类似雕刻的界面似乎是目前直接创作高细节数据的唯一方法。从三角网格转换是另一种方法，也是我们的系统目前唯一支持的方法。遗憾的是，将三角网格转换为体素有一个缺点，即两种模式都有局限性。除非源数据非常详细，否则体视网格无法充分发挥其潜力。

更广泛地说，制作具有极高几何细节的内容似乎还处于早期阶段。目前正在使用的方法有：数字雕刻、使用激光扫描设备获取、程序生成，以及多种特殊方法（如扫描人脸）。在这些方法中，只有从雕刻到网格的流水线足够成熟，可以广泛采用。当高细节几何体的实时渲染成为可能时，这一领域将如何发展，我们拭目以待。

3.6 收购

使用照相机或扫描设备获取真实世界的数据是体素和其他基于样本的表示方法的优势所在。很难想象有什么设备能在扫描的二维或三维区域内生成离散数据点以外的东西。点云当然可以转换成详细的彩色三维网格，然后再转换成基本网格和位移图。但是，对于不适合这种表示方法的结构，例如细枝或采样不足的数据，要进行这两个阶段的转换都非常困难。

如果我们能直接将原始输入数据转换为拼接数据或体素数据，就不需要这种转换链了。例如，如果我们有一个严重采样不足的草叶，它只有少数几个样本，没有形成一个连续的表面，那么对于基于样本的表示法来说，这并不是一个问题（只要不是从太近的距离观察物体）。我们仍然可以渲染现有的样本，而且几乎不会出现灾难性的故障。

3.7 效果图

位移映射三角形可直接使用 GPU 硬件细分光栅化。这使得渲染过程与当前基于三角形的图形类似。然而，对这种网格进行光线投射要困难得多。位移贴图的强大之处在于能够紧凑地编码相当于数百万个三角形的网格，因此将其 "解压 "为单个三角形并将其放入加速层次结构中以实现高效的光线投射是不可行的。因此，在实际操作中，光线投射器需要分别考虑两个细节层次，首先根据位移几何体的边界体积对光线进行交互处理，然后再根据位移图本身进行交互处理。

体素也可以直接光栅化，但正如第 6.4 节所述，光栅化的效率似乎不如光线投射。，由于八叉树的规则结构，对体素层次结构进行光线投射非常简单高效。在光线投射过程中，除了轮廓评估需要将光线与一对平行平面相交外，只需进行很少的运算。我们的基准测试结果见第 8 节。如第 3.2 节所述，投射到体素结构中的射线可以根据与射线原点的距离在适当的细节级别上工作。这样可以减少计算量和内存带宽的使用，同时降低高频内容造成的混叠。

3.8 摘要

考虑到内存使用情况，三角形和位移图比体素更紧凑，但也只是相差 3.33 倍。考虑到体素表示法可以适应比位移图更精细的数据重排，因此在实际应用中两者的差距较小。在第 5.7 节中，我们还提出了降低体素内存使用率的其他方法。

降低体素采样率的做法很有吸引力，因为单一算法就能处理所有可能的内容，将密集的部分遮挡几何体正确转换为体积转换。此外，将 8 个子体素的阴影模型合并为父体素可能比对三角形做同样的处理更直接。体素能够将经过缩样的高密度不透明数据（如毛皮）正确地表示为部分透明的体积 "云"。这是位移贴图无法做到的。

三角形数据能比体素数据更好地处理缩放，而参数补丁能更好地处理缩放。体素数据的块状性通常会影响缩放效果。

但在不提高分辨率的情况下，尖锐的特征不一定能很好地表现出来。

根据目前的知识，体素数据的一般变形无法有效完成，而三角形数据则没有问题。不过，分割体素数据（即分离数据块）非常容易，这对于模拟可破坏的墙壁和建筑物等就足够了。

对于三角形和体素来说，创作都是一个悬而未决的问题。虽然手动三角形建模是当今内容创作的实际方法，但要过渡到极其精细的几何图形并非易事。关键问题在于可表示的数据类型和艺术家所需的工具集。最能满足现实世界要求的数据结构才是制胜之道。

从扫描样本到体素的采集管道似乎比将原始数据转换成三角形和位移图更简单。尤其是在样本到样本的转换过程中，处理不完美数据应该更容易实现。

最后，三角形数据的渲染可以享受快速光栅化的好处。，正如我们的研究结果所显示的那样（第 8 节），使用体素对几何去尾曲面进行光线投射可能比使用三角数据更有效。因此，体素似乎更适用于高级照明效果（如反射和路径跟踪）的渲染。

4 内存消耗量的维度分析

在使用体素数据渲染图像时，最好在 GPU 内存中保留靠近摄像头的高分辨率数据，而对于远处的事物则保留较低分辨率的数据。虽然这一点显而易见，但要获得给定的观察距离需要多少数据，或者内存需求如何随场景分辨率、图像分辨率和观察距离而变化，却并不明显。在本节中，我们将为分析这些问题提供一个理论框架。

让我们考虑一下以任意点为中心、边长为 2^l 的立方体周围的世界内容。假设我们正在检测比例为 s 的体素，即空间中的非空立方体，这意味着每个体素立方体的边长为 2^s。在忽略立方体位置的情况下，我们定义*体素计数*函数 $N(l, s)$，以给出一个尺度为 l 的立方体中尺度为 s 的体素数量。由于我们只考虑非空的体素，所以一般情况下，这个数值要比 $2^{3((l)(-)(s)(l))}$的上限小得多。

为了研究改变*体素*比例 s 或立方体大小 l 时体素数量的变化情况，我们定义了两个辅助函数。首先，我们来看看体素比例加倍时体素数量的变化情况。这样就得到了*局部维度函数* $D_{(局部),}$ 其定义如下

$$D \underset{(本地)}{(l, s)} = \log \frac{N(l, s()) \text{、}}{^2 N(l, s+1)}$$

对数将体素的比例转换为维度，，如果体素数量增加了 8 倍，则维度为 3；如果增加了 4 倍，则维度为 2，以此类推。需要注意的是，维数不一定是整数，可以是范围内的任何实数。
]0, 3].

局部维度告诉我们，当数据压缩量增加或减少时，数据会有怎样的表现。因此，一个平面的局部维度为 2，一个体积块的局部维度为 3。

徘徊在 2 左右。图 24 显示了测试场景在不同比例下的测量维度。

现在，我们来定义*全局维度函数*，它说明了当立方体大小 l 增加一倍（即 l 增加 1）时，体素数量的变化情况。 定义如下

$$D_{global}(l, s) = \log_2 \frac{N(l+1, s)}{N(l, s)}.$$

全局维度说明了扩大观测区域对数据量的影响。，在一个内容位于地平面之上的世界中，对于涵盖整个垂直范围的尺度，全局维度约为 2。
的数据量。即使地面上有一层体积雾，这一点也是成立的--我们观测到的宇宙半径增加一倍并不会使数据量增加八倍，而八倍的数据量将是我们观测到的宇宙半径的两倍。
雾数据的局部维度，但只增加了四倍。

现在，立方体尺寸增加一倍和体素比例增加一倍所产生的综合影响，可以用局部和全局比例变化的组合来表示。我们称其为*尺度维度*，因为它反映了数据量是如何随着观察尺度的变化而变化的。

$$D_{scale}(l, s) = \text{对数}_{(2)} \frac{N(l+1, s+1)}{N(l, s)} = D_{global}(l, s) - D_{local}(l+1, s)。$$

将缩放的维度分解为局部和全局两个部分非常有用，因为这两个部分受体素内容的不同特征支配。局部维度取决于小邻域的形状，而全局维度则取决于场景的大尺度特征。

计算两种比例的综合效果的原理如下。假设我们内存中有一部分世界，其分辨率足以进行高质量的渲染。现在假设我们希望将可视区域的半径（或边长）增加一倍，即观看距离增加一倍。如果分辨率保持不变，我们将需要 $2^{D(global)}$的数据量，但由于新内容的平均距离是原来数据的两倍，我们可以根据 $2^{D(local)}$来降低该部分的分辨率。

我们可以看到，尤其是在局部维度低于全局维度的情况下，信息内容会随着视距的增加而迅速增长。植被就是这类数据的一个很好的例子，因为树叶和草叶是二维的，或许只有一维，但它们（或多或少）充满了整个空间，直到达到一定的尺度。，雾层的表现要好得多，因为当距离增加时，对分辨率的要求降低了八倍，但较大区域内包含的内容却只有四倍。因此，随着距离的增加，观看距离增加一倍的成本也会降低。

假设标度维数 D 不变，我们可以通过对 x^D 至 $\log(r)$ 的积分，得到获取 r *的*视距所需的总数据量的复杂程度，即

$$\int_0^{\log_2 r} x^D = \begin{cases} O((\log r)^{(D)+1}) & D > -1 \\ O(\log\log r) & D = -1 \\ O(-(\log r)^{(D)(+1)}) & D < -1 \end{cases}$$

我们可以看到，内存使用复杂度总是与半径呈近似线性增长。在 $D_{(scale)}$= 0 的通常情况下，复杂度与视图半径成对数关系，而在 $D(scale)$ 0 的通常情况下，复杂度与视图半径成对数关系。

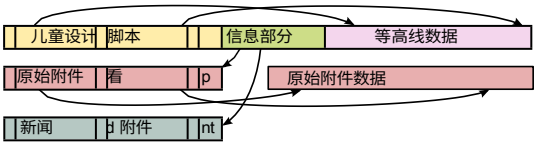


图 1：*单个八叉树数据块。子描述符和附件数组使用相同的索引寻址。在数组是由页眉和远指针造成的。页眉位于每 8 千字节的边界，指向信息部分。*

当 D 全球 $>$ Dlocal 的对数，其复杂度为
功率

提高显示分辨率的效果更容易分析。如果将显示分辨率乘以 2，这意味着 $D_{(局部)}$ 和 $D_{(全局)}$ 将用比例参数 s - 1 而不是
s.这相当于按 $2^{D局部}$进行缩放。因此，只有局部维度（因距离而适当缩放）才会影响显示分辨率变化时的内存使用量。局部低维数据，如表面碎片和毛发

因此，它比高维体积数据表现得更为出色。

5 体素数据结构

我们使用稀疏的八叉树数据结构在 GPU 内存中存储体素数据，其中每个节点代表一个体素，即与表面几何体相交的轴对齐立方体。体素可进一步细分为更小的体素，在这种情况下，父体素及其子体素都包含在八叉树中。设计这种数据结构的目的是在支持高效光线投射的同时尽量减少内存占用。有时两者可以同时实现，因为更紧凑的数据布局还能降低对内存带宽的要求。

为此，我们采用了一种方案，将与象素相关的大部分数据与其父象素一起存储。这样就不需要为单个叶片体素分配存储空间，也更方便压缩阴影属性。

在最高级别上，我们的八叉树数据被划分为*多个块*。块是内存中的连续区域，与树中的体素局部区域相对应，存储八叉树拓扑结构以及体素几何形状和阴影属性。块内的所有内存引用都是相对的，因此很容易在内存中重组块。这有利于动态更新八叉树，例如从磁盘流式传输数据时。

每个区块由子描述符数组、信息部分、轮廓数据和数量不等的附件组成。如图 1 所示。子描述符（第 5.1 节）和轮廓数据（第 5.2 节）分别表示八叉树的拓扑结构和体素的几何形状。附件（第 5.5 节）是单独的数组，用于存储每个体素的阴影属性。信息（info）部分包含了一个可用的附件目录。
以及指向第一个子描述符的指针。

在光线投射过程中，我们会访问子描述符和轮廓数据。一旦光线击中表面几何体，我们就会执行一个着色器，查找特定区块所包含的附件，并解码着色器。
属性。在本文介绍的数据集中，我们使用了一个简单的 Phong 阴影模型，每个体素都有唯一的颜色和法向量。

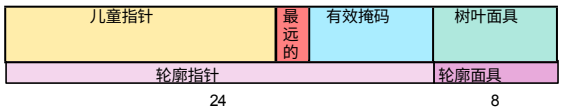


图 2： *为每个非叶体素存储的 64 位子描述符。*

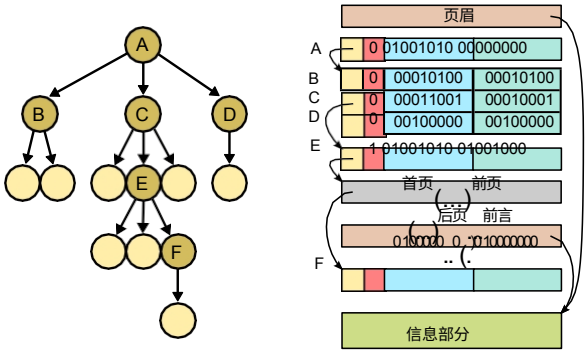


图 3： *子描述符阵列的布局。左：体素层次示例。右图子描述符数组，包含示例层次结构中每个非叶体素的一个描述符。*

5.1 儿童描述符

我们使用 64 位子描述符对八叉树的拓扑结构进行编码，每个子描述符对应一个非叶子体素。叶状体不需要自己的描述符，因为它们是由其父代描述的。如图 2 所示，子描述符分为两个 32 位部分。第一部分描述子体象素集，第二部分与轮廓相关（第 5.2 节）。

每个体素在空间上被细分为 8 个大小相等的子槽。子描述符包含两个比特掩码，每个比特*掩码*为每个子槽存储一个比特。*有效掩码*说明每个子槽是否实际包含一个体素，而*叶子掩码*则进一步说明这些体素是否都是叶子。根据位掩码，子槽的状态可以解释如下：

- 这两个位都不设置：槽没有与曲面相交，因此是空的。
- *有效掩码*中的位被置位：槽中包含一个非叶体素，可进一步细分。
- 两个位都设置为：槽中包含一个叶状体。

如果体素包含任何非叶子的子节点，我们会存储一个无符号的 15 *位子节点指针*，以便引用它们的数据。反过来，这些子节点会在连续的内存地址上存储自己的子节点描述符，*子节点指针*指向其中的第一个描述符，如图 3 所示。这样，我们就可以根据位掩码递增指针，找到特定的子代。子代既可以与父代位于同一个块中，也可以位于不同的块中，这样就可以在不考虑块边界的情况下遍历八叉树。

如果子代远离引用脚本器，15 位字段可能不足以容纳相对指针。为了说明这一点，我们在子代描述符中加入了一个远位。如果该位被设置，*子指针*将被解释为对单独的 32 位远指针的间接引用。远指针在同一数组中交错排列，必须放置在足够靠近引用描述符的位置。在实践中，远指针可以通过以下方式变得非常罕见

图 4： *轮廓值中的位域。法线的组成部分矢量和空间位置都是有符号整数。厚度以无符号整数编码。*

以近似深度优先的顺序对每个区块内的子描述符进行排序。

除了遍历体素层次结构外，我们还必须能够分辨出某个体素所在区块。这可以通过分布在子描述符中的 32 位页面头来实现。页眉放置在每 8 千字节的边界，每个页眉都包含一个指向块信息部分的相对指针。通过将子描述符数组的起始位置放在这样的边界上，我们只需清除任何子描述符指针的最低位，能找到页眉。

5.2 轮廓

将体素数据可视化的最直接方法是将每个体素所包含的几何体近似为一个立方体。只要数据过密，即每个体素在屏幕上的投影小于一个像素，所产生的视觉质量是可以接受的。但是，如果由于未充分放大而导致体素明显增大，近似值物体轮廓附近就会产生非常明显的伪影。这是由于用一个完整的立方体替换了体素与实际表面之间的每个交点，从而有效地扩大了表面。如图 5 顶部一行所示，这会带来明显的近似误差。

为了减小近似误差，我们将每个体素与一对与近似表面方向相匹配的平行平面相交，以此来限制每个体素的空间范围。我们将这对平面称为轮廓。如图 5 的中间行和最下面一行所示，结果是一组定向板，为曲面定义了一个紧密的边界体积。对于平面和相对光滑的表面，平面的方向可以与平均表面法线保持一致，以获得良好的拟合效果。对于曲面和采样不足的表面，平面仍可用于减小近似误差，如图 6 所示。

我们使用 32 位来存储一个体素对应的轮廓。该值分为五个部分：三个部分用于定义两个平面的法向量，两个部分用于定义它们在体素内的位置。详见图 4。

体素与相应轮廓之间的映射是由子描述符中的两个字段建立的（图 2）。*轮廓掩码*是一个 8 位掩码，用于说明每个子槽中的体素是否有相关的轮廓。存储一个单独的位掩码可以省略那些不能显著减少近似误差的体素中的轮廓。*与子指针*类似，无符号 24 位*轮廓指针*引用的是一个连续的轮廓值列表，*轮廓掩码*中每设置一位就引用一个轮廓值。

5.3 轮廓线之间的合作

虽然每个叶片体素只使用一条轮廓线就足以表示光滑的表面，但它也会在物体的尖锐边缘附近引入令人分心的假象。这是因为在包含此类边缘的体素中，表面的方向变化很大，而没有一个方向可以很好地代表表面上的所有点。

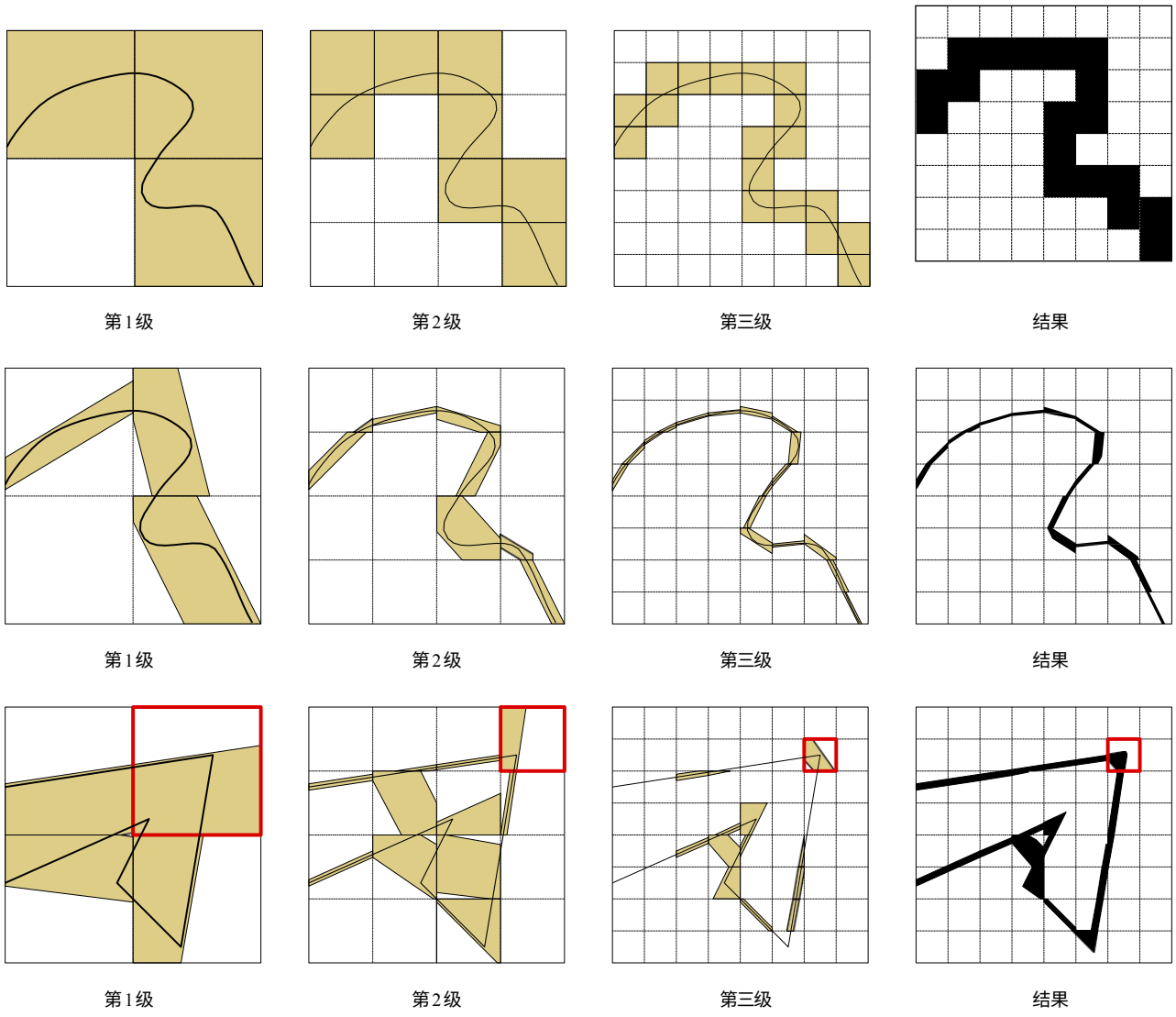


图 5： *轮廓对曲面近似的影响。排：立方体。中行：用轮廓增强的体素。所得到的近似表面比上排更接近原始表面，但在高曲率区域仍存在一些锯齿。需要注意的是，如果没有尖角，最详细的等高线就足以得到最终结果。下行：有尖锐边缘的表面。将各层重叠的轮廓线相交即可得到结果。在高亮区域，所有三个层次都对体素的最终形状做出了贡献。*

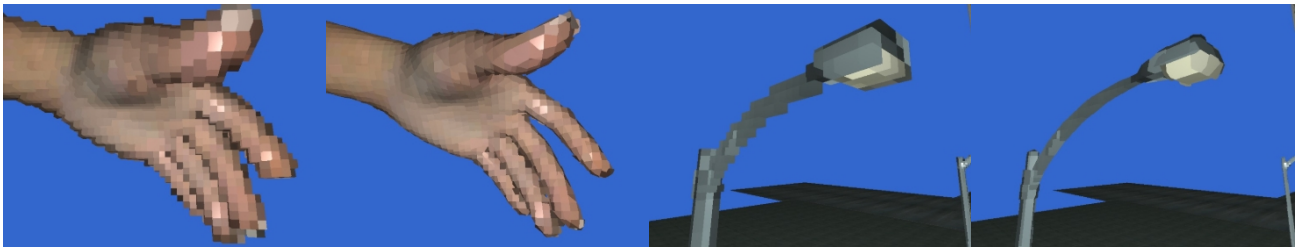
幸运的是，我们可以利用存储重叠体素的完整层次这一事实。为了实现多个轮廓之间的合作，我们将体素的最终形状定义为其立方体与所有祖先轮廓的交集。，我们就可以通过为八叉树每一层上的轮廓线选择不同的法向量，来逐步增加具有代表性的表面方向集，从而显著提高质量。在图 5 底行的高曲率区域附近可以看到这种效果。

5.4 未压缩阴影属性

除了体素的几何形状外，我们还需要存储一些用于着色的材质属性。这些属性数据以一个或多个附件的形式包含在八叉树块中。附件是辅助数据缓冲区，其解释是根据其类型而定的。每个数据块的信息部分存储了该数据块所包含的附件目录，每个附件由一个类型 ID 和一个相对数据指针标识。访问模型是这样的：在渲染过程中，一旦光线击中曲面，着色器就可以查询为特定体素提供附件，并解码所需的属性。

我们使用的 Phong 染色模型需要为每个体素提供颜色和法线向量。我们有两种不同的方案来存储这些属性。在本节中，我们将介绍一种直接存储未压缩（即原始）属性数据的方法，下一节（第 5.5 节）将介绍一种更紧凑的基于块的压缩方案。尽管我们只考虑了颜色和法线，但我们希望同样的想法也能适用于其他各种材料属性。

在简单但低效的非压缩存储格式中，我们使用 64 位编码来存储与单个体素相关的颜色和法线向量。如图 7a 和图 7b 所示，64 位值被分成 32 位颜色值和 32 位法线值。我们对颜色值使用标准 ABGR 编码，将每个通道存储为单独的 8 位整数。然而，对于法线向量，我们必须使用更专业的编码。



(a) (b) (c) (d)

图 6： *（a）和（c）：立方体体素。(b)和(d)：带有轮廓线的相同体素。在这种表面非常光滑的情况下，轮廓线可以提高几个层次的几何分辨率。请注意，为了说明效果，我们故意对模型采样不足。在实际内容中，改进体现在细节上，而不是整体形状上。*

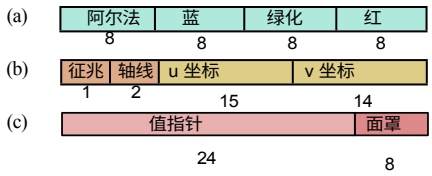


图 7： *与存储未压缩属性相关的位域。 (a) 颜色值。(b) 法线向量。(c) 查找条目。*

与基于三角形表示法的传统法线映射存储方式相反，我们必须使用对象空间法线来代替切线空间法线。实验表明，在大型平滑曲面上，物体空间法线至少需要 12 比特的精度才能避免可见的量化伪影。为了最大限度地提高精度，我们将每个法线存储为立方体上的一个点。面使用 3 位（符号和轴）识别，面上坐标使用两个定点整数存储，一个 14 位，一个 15 位，共 32 位。

存储未压缩颜色和法线的附件由两部分组成。第一部分是与子描述符数组布局相匹配的查找项数组。第二部分是由查找条目引用的 64 位属性值集合。子描述符数组中的每个条目都与查找数组中的条目直接对应，包括远指针和页眉造成的间隙。这样就可以将子描述符指针映射到相应的查找条目，而不需要额外的数据结构。请参见图 1 的说明。

查找条目的编码如图 7c 所示，与子描述符的第二部分类似。8 位属性掩码告诉我们体素的每个插槽中的子体素是否有贡品值。未设置相应位的子槽从父体素继承其属性，这样就可以在属性变化不大的地方省略属性。*值指针*引用附件第二部分中的连续属性值列表，属性掩码中每设置一个位就引用一个值。指针相对于附件的起始位置。

图 8 中的示例代码总结了查找指定体素对应的未压缩属性值的过程。尽管这一过程涉及多次间接内存查找，但与光线投射的成本相比还是相对较低的。

5.5 压缩阴影属性

假定平均分支系数为 4，子描述阵列每个体素大约需要 2 个字节（以叶体素计

```
int lookupUncompressedAttributeAddress( int
    childDescAddr、
    int childSlotIdx, int
    attachmentIdx)
{
    // 找到信息部分。

    int pageSize= 8192;
    int pageAddr = childDescAddr & ~(pageSize - 1); int
    infoAddr= pageAddr+ *((int*)pageAddr) * 4;
    InfoSection* info= (InfoSection*)infoAddr;

    // 确定块中子描述符的索引。

    int blockAddr= infoAddr+ info->blockPtr * 4;
    int childDescIdx= (childDescAddr - blockAddr) / 8;

    // 查找并解码查找条目。

    int attachPtr= info->attachPtr[attachmentIdx]; int
    attachAddr = infoAddr + attachPtr * 4;
    int lookupAddr= attachAddr+ childDescIdx * 4; int
    lookupEntry = *(int*)lookupAddr;
    int attribMask= lookupEntry & 0xFF;
    int valuePtr= (lookupEntry>> 8) & 0xFFFFFF;

    // 是否省略了体素的属性？ int childBit = 1 <<

    childSlotIdx;
    if ((attribMask & childBit)== 0) return 0;

    // 为所请求的子槽查找值。

    int valueIdx= popc8(attribMask & (childBit - 1)); return
    attachAddr + valuePtr * 4 + valueIdx * 8;
}
```

图 8： *查找给定体素未压缩属性值的代码示例。体素是使用子槽索引和相应子描述符的内存地址指定的。*

考虑在内），非常紧凑。然而，着色属性的原始编码很容易破坏这种紧凑性。由于大部分渲染时间都花在光线投射上，而光线投射根本不访问属性数据，因此用属性解码性能换取更少的内存占用是合理的。

为此，我们将颜色和法线编码在一起，采用基于块的压缩方案，每个体素平均使用 1 个字节表示颜色，2 个字节表示法线。与 DXT 一样（参见 [van Waveren 和 Castan˜o 2008]），每个压缩块可表示 16 个值。由于体素有 8 个子槽，我们将两个连续的子描述符所描述的体素分配到同一个压缩块，以建立压缩值和子槽之间的直接对应关系。平均约有一半的子槽是空的，因此有 8 个使用过的值和 8 个未使用的值。

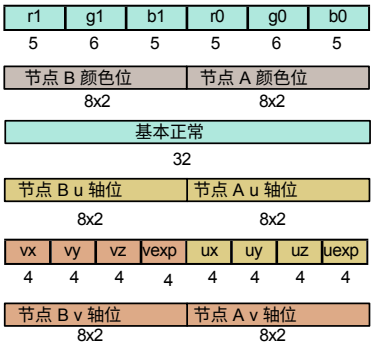


图 9：*压缩块由六个 32 位字组成。前两个字编码 16 个子槽的颜色，其余四个字编码相应的法线。*

每个压缩块中的值。我们只需在子描述符数组中留出空隙，确保只将具有相同父体象素的描述符配对，从而避免将层次结构中不同部分的值放入同一压缩块中。

尽管我们的方案浪费了压缩块中大约一半的可用容量，但却避免了额外查找表的成本。这样做的好处还在于，由于每个压缩块内的竞争较少，因此可以更准确地表示单个值。另一种方法是引入一个与用于未压缩属性的查找表类似的查找表，并将每个连续运行的 16 个属性编码为一个单独的块。

压缩块的编码如图 9 所示。

颜色体素颜色使用 DXT1 的简化变体进行编码，省略了与透明度相关的语义。压缩块的前 32 位使用 16 位 RGB-565 编码存储两种参考颜色（$\mathbf{c_0}$ 和 $\mathbf{c_1}$）。其余 32 位存储两位插值因子，用于从集合 $\{\mathbf{c_0}, \mathbf{c_1}, \frac{2}{3}\mathbf{c_0}+\frac{1}{3}\mathbf{c_1}, \frac{1}{3}\mathbf{c_0}+\frac{2}{3}\mathbf{c_1}\}$ 中选择 16 种颜色中的每一种。

法线现有的法线压缩文献大多只考虑切线空间法线（如 [ATI 2005；Munkberg 等人 2006；Munkberg 等人 2007]），因此不适用于我们的情况，因为无法隐式推导出切线框架。对于体素法线，我们可以利用现有的法线贴图压缩技术，如对象空间 DXT5 [van Waveren 和 Castan˜o 2008]。遗憾的是，这些方法所提供的 8 位精度不足以实现平滑的高光和反射。因此，我们采用了一种新颖的压缩方案，为平滑变化的法线提供高达 14 位的精度。

我们的法线压缩方案是在三维法线空间中放置一个线性 *4×4×4* 网格，然后从 16 个候选法线中选择每个法线。如图 10 所示，网格是由一个基本法线 $\mathbf{n_b}$ 和两个轴向量 $\mathbf{n_u}$ 和 $\mathbf{n_v}$ 定义的。每个候选法线的定义为 $\mathbf{n_b}+c_u\mathbf{n_u}+c_v\mathbf{n_v}$，$c_u$ 和 c_v 分别选自集合 $\{-1, -\frac{1}{3}, \frac{1}{3}, 1\}$。

为了更好地适应各种数据，我们对 $\mathbf{n_b}$、$\mathbf{n_u}$ 和 $\mathbf{n_v}$ 不做任何正交性要求。

基础法线 $\mathbf{n_b}$ 作为立方体上的一个点进行编码，编码方法与原始属性相同（图 7b）。轴向量 $\mathbf{n_u}$ 使用三个带符号的 4 位整数和一个 4 位指数来存储，即具有共同指数的浮点矢量。$\mathbf{n_v}$ 轴的存储方式与此类似，轴的总位数为 32 位。

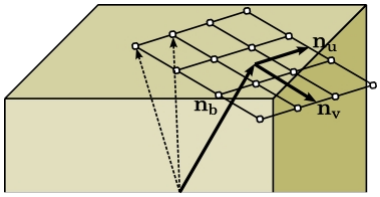


图 10：*法线压缩方案示意图。基本法线 $\mathbf{n_b}$ 指定了单位立方体上的一个点，两个轴向量 $\mathbf{n_u}$ 和 $\mathbf{n_v}$ 围绕这个点定义了一个任意的 4 × 4 × 4 网格。这三个矢量之间没有正交性要求，轴矢量也不受限制地位于立方体的面上，因此具有最大的灵活性。虚线箭头表示这组向量所定义的 16 个法线中的两个。*

向量。压缩块的其余部分包含每个法线的两个 *U* 轴位和两个 *V* 轴位，分别指定 c_u 和 c_v 值。

这种压缩方案可以灵活地处理不同类型的情况。如果区块内的法线方差较小，则可以使用 $\mathbf{n_b}$ 来存储高精度的平均法线，同时使用较小的 $\mathbf{n_u}$ 和 $\mathbf{n_v}$ 指数来最小化量化误差。如果法线只在一个方向上变化，则可以将 $\mathbf{n_u}$ 和 $\mathbf{n_v}$ 设置为方向相同但长度不同，以最大限度地提高沿该特定方向的精度。最后，如果法线的方向完全不同，则可以选择其中一个方向作为 $\mathbf{n_b}$同时使用 $\mathbf{n_u}$ 和 $\mathbf{n_v}$ 近似其他两个方向。

5.6 内存使用分析

让我们首先考虑体素层次结构，然后再考虑属性（颜色、法线、轮廓）。对于每个非叶体素，我们的层次结构编码需要两个 8 位掩码和一个 15+1 位子指针。对于大于 15 位字段所能容纳的偏移量，我们需要单独的远指针，但其占用的内存量可以忽略不计。对于叶状体素（对应最精细分辨率样本），不存储层次结构数据。因此，在平均每个象素有 4 个分支面的情况下，我们平均每个象素有 1 个字节的层次结构数据。

使用轮廓线时，每个体素需要 32 位。不过，只有在能提高表面质量的体素中才需要轮廓线，而在大多数情况下，轮廓线只占所有体素的一小部分。尽管如此，每个非叶状仍需要 32 位查找条目（轮廓指针和掩码），这使得每个体素的轮廓总成本略高于 1 字节。

现在让我们来考虑属性数据的大小。如上所述，颜色可压缩为每个槽 4 比特，因此最好不使用属性数据指针，而在属性数据缓冲区中留出空位。我们总是需要对所有 8 个子节点进行编码，因此每个非叶子体素需要 32 个颜色比特，即每个体素 1 个字节。我们的正常压缩格式要求每个样本 8 位，假设平均分支因子为 4，则每个体素总共需要 2 个字节。

将层次、轮廓、颜色和法线加在一起计算，每个体素大约需要 5 个字节（表 1）。

5.7 可能的改进

在本节中，我们列出了一些潜在的、但尚未经过测试的数据结构改进措施。其中一些很容易预先

而其他观点的影响则需要经过实证检验才能评估。请注意，以下一些观点并不相互兼容。

组合间接属性查找。当属性大小足够大时，使用查找条目（图 7c）是有益的。由于每个体素平均有 4 个子元素，盈亏平衡点为 1 字节，因此颜色和法线在压缩到每个体素 4 位和 8 位时，都不会从额外的间接中受益。不过，由于我们总是将颜色和法线存储在一起，因此设置查找条目是有意义的。

目前，由于压缩磁贴中浪费的插槽，颜色-法线对每个体素实际上需要 24 比特，但有了查找功能，我们可以将有效载荷减少到 12 比特，查找条目减少到大约 8 比特，总计 20 比特。此外，查找条目还允许省略不需要的属性，这与二整寻址方案不同。这样就可以在不影响质量的情况下，在叶片体素附近进行一定程度的组合。

重新利用未使用的子指针。只包含叶子象素作为子象素的象素会浪费子描述符中为*远指针和子指针*预留的 16 位（见图 2）。由于这种情况在树叶附近很常见，因此重新利用未使用的字段很有意义。一个有趣的可能性是空闲空间中存储轮廓指针和轮廓掩码。

由于轮廓指针只剩下 8 位，我们需要将轮廓数据与子描述符交错在一起，这与当前的方法不同。此外，让一些子描述符占用 32 位，一些占用 64 位，也会使内存获取复杂化。目前我们总是获取 64 位，而现在我们需要先获取 32 位，然后检查*叶遮罩*的内容。如果没有，则需要再获取 32 位。请注意，不可能始终读取 64 位，因为子描述符将不再适当对齐。

如果所有叶片体素都在同一水平面上，我们可以通过这种方案为每个叶片节省大约 1 字节。考虑到其他层面，每个体素可节省约 0.75 字节，从而消除了大部分与轮廓相关的内存消耗。在实践中，节省的内存会比这少，因为一个体素可能既有叶状子代，也有非叶状子代，在这种情况下就无法优化。我们还没有测量过一个体素只包含叶子作为子代的情况有多常见，因此这种优化的真正影响还只是推测。

重新利用未使用的轮廓指针。这种潜在的优化是对子描述符、轮廓和附件查找条目的重大重组。我们的想法是在节点数组中存储属性查询项，尽可能利用未使用的轮廓指针。

在下文中，我们假设所有属性都是通过查找条目进行独立访问的。让我们考虑一下 GPU 内存中连续排列的子描述符列表。目前，每个体素都有一个 64 位的条目，其中一半编码层次结构，一半指向轮廓数据。附件查找条目存储在一个单独的数组中。

为提高内存利用率，子描述符跨度可按如下方式布局。非叶子体素的每个子描述符仍占用 64 位，这样就可以根据父体素中子描述符的索引进行快速查找。不过，子描述符的后 32 位可能包含轮廓查找条目，也可能不包含。轮廓查找条目可以通过设置最高位等方式来识别，而该空间的其他可能内容的最高位则会被清零。这样，光线投射器就可以窥视该位，如果该条目不是等高线查找条目，则该体素的子节点都不会被窥视。

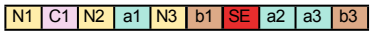


图 11：*子描述条目和查找条目的假设交错方案。图中是一个跨度，包含三个非叶体素* N1-N3*。所有方框对应 32 位。只有第一个体素有轮廓查找条目* C1*。附件查找条目* a1-a3 *和* b1,b3 *以不直观但高效的方式存储在跨度条目字段* SE *的两侧。讨论见正文。*

有轮廓。因此，光线投射器接触到的数据量与当前编码相比不会有任何变化。

在子描述符条目之后，span 会继续显示一个 32 位的 span 。跨度条目是子描述符跨度的目录，显示哪些体素有轮廓指针，每个子体素有哪些属性查找条目。这些属性查询条目可以部分存储在子描述符未使用的半部分中，部分存储在 span 条目之后。

请看图 11 所描述的情况。跨度中有三个体素，分别标记为 N1-N3。只有 N1 的子节点有轮廓，因此旁边有一个轮廓查找条目 C1。属性查找条目 a1 和 b1 保存在未使用的轮廓指针槽中。

这种方案最重要的一点是，它可以有选择性地省略轮廓和附件的查找条目。例如，如果我们的附件只包含极少体素的数据，那么当前的编码总是需要至少存储查询项，这样每个体素就需要 1 个字节。建议的编码方式可以消除这种开销。

未使用的组合。目前，我们在*有效掩码*和*叶掩码*中只使用了四种可能组合中的三种（图 2）。到目前为止，我们还没有使用这种组合，因为使用这种组合的最佳方法并不明显。

当前方案的一个有趣扩展是使用未使用的组合来指示部分透明的体素。这样就可以在光线投射过程中累积遮挡因子或进行颜色累积和衰减。一个小问题是，我们无法区分透明叶片体素和透明非叶片体素。因此，所有透明体素都需要有相应的子描述符条目，即使它们是叶状体素。

6 效果图

八叉树数据结构的规则性是实现高效射线投射的关键因素。由于与体素相关的大部分数据实际上都存储在其父体素中，因此我们需要使用父体素 *parent* 和 0 到 7 之间的子槽索引 *idx* 来表示射线当前正在穿越的体素。 由于我们不存储体素的空间位置信息，因此我们还需要维护一个与当前体素相对应的立方体。我们使用位置向量 *pos*（每个维度的范围从 0 到 1）和一个非负整数*标度*来表示立方体，标度定义立方体的范围为 $\exp_2(scale - s_{max})$。整个八叉树都包含在一个以原点为中心、大小为 s_{max} 立方体中。

基础知识让我们的射线定义为 $\mathbf{p}_{rt}(t) = \mathbf{p} + t\mathbf{d}$。求解轴线对齐平面的 t，得到

$$t_x(x') = \frac{(1)\, x'}{dx} + \frac{-p_{(x)}}{dx}$$

为 *x 轴*，*y* 轴和 *z* 轴的计算公式类似。如果使用预计算的射线系数，这相当于一个单一的乘法运算。

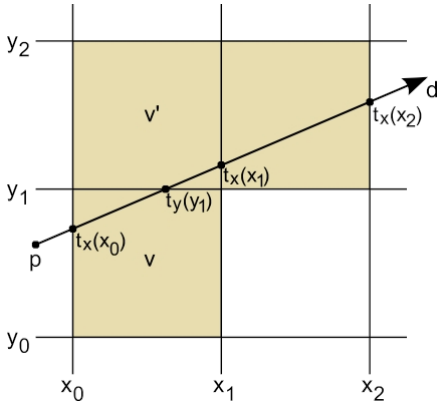


图 12：*通过等比例体素前进。射线由原点矢量* **p** *和方向矢量* **d** *定义。它在* $t = t_x(x_0)$ *处进入* v，*在* $t = t_y(y_1)$ *处离开* v，*然后到达* v'。

添加指令。我们可以将一个轴对齐的立方体表示为一对对角 (x_0, y_0, z_0) 和 (x_1, y_1, z_1)，这样 $t_x(x_0) \le t_x(x_1)$，$t_y(y_0) \le t_y(y_1)$，$t_z(z_0) \le t_z(z_1)$。根据这一定义，与立方体相交的 t 值跨度为 $t_{cmin} = \max(t_x(x_0), t_y(y_0), t_z(z_0))$ 和 $t_{cmax} = \min(t_x(x_1), t_y(y_1), t_z(z_1))$。

让我们考虑这样一个问题：根据*父节点*和 *idx* 指定的当前体素 v，确定射线上的下一个体素 v'。我们首先假设 v 和 v' 同级体，这意味着它们的比例相同，如图 12 所示。在这种特殊情况下，当前体素的立方体横向跨度为 $[x_0, x_1]$，纵向跨度为 $[y_0, y_1]$。这意味着射线必须通过 x_1 或 y_1 从 v 中穿出，以先相交者为准。交点对应的 t 值由函数 t_x 和 t_y 给出。我们可以看到，$t_y(y_1) < t_x(x_1)$，这意味着射线先于 x_1 与 y_1 相交。根据类似于 [Amanatides 和 Woo 1987] 的推理，我们可以得出 v' 位于 v *的*正上方的结论。

因此，我们可以通过比较 $t(x)$、$t(y)$ 和 $t(z)$ 与 t_c 并向前推进，来确定下一个相同比例的体素。x_1 y_1 z_1 *最大*
的立方体位置。假设这两个体素共享同一个父体，我们通过翻转 *idx* 中对应于相同轴线的位来获得新的子槽索引 $idx^{(\prime)}$。

层次遍历现在，我们将把递增遍历的概念扩展到体素的层次结构。这是必要的，因为我们的八叉树数据结构是稀疏的，我们不包括与空空间相对应的子树。以分层方式进行遍历还有一个好处，就是可以通过使用等值线作为相应子树的边界体来提高性能。

我们的算法以深度优先的顺序遍历与射线相交的体素集合。在每次迭代中，选择下一个体素有三种不同的情况：

- 推：前进到射线首先进入的子体素。
- 前进： 进入下一个同胞体素。
- POP：跳转：跳转到射线跳出的最高祖先的下一个兄弟姐妹。

图 13 显示了由此产生的遍历顺序示例。

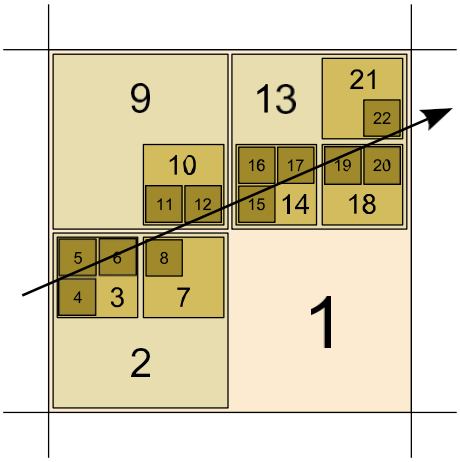


图 13：*分层遍历的顺序。算法从象素 2 开始，象素 2 是根 1 的子节点。在执行两次* ADVANCE *以遍历同胞 5 和 6 后，算法注意到射线从它们的共同父节点 3 穿出。因此，它执行了* POP，*到达了象素 7。当射线离开根部时，遍历在叶 22 之后结束。请注意，为了使数字更加清晰，图像中的体素已被缩小。*

该算法包含一个父体象素堆栈和与当前象素祖先相关的轮廓 t 值。堆栈的深度为 s_{max}，因此可以使用立方体比例值直接寻址堆栈条目。每当算法通过执行 PUSH（推入）来降低层次结构时，它都可能会根据保守检查将上一个父节点按*比例*存储到堆栈中。当射线退出当前父体体素时，算法会通过执行 POP 来提升层次结构。它首先使用当前位置 *pos* 来确定新的 pos'、$scale'$ 和 $idx^{(\prime)}$，所述。然后在 $scale'$ 处读取堆栈，恢复父级位置。

的情况下，确定射线首先进入的子体素。
PUSH 类似于在 ADVANCE 中选择下一个兄弟姐妹。我们评估
在体素中心测量 t_x、t_y 和 t_z，并将它们进行比较
来确定新 idx' 的每一位。

为了区分 ADVANCE 和 POP，我们需要找出射线是否停留在同一个父体体素内。我们首先假定它是，然后计算候选位置 pos^* 和子槽索引 $idx^{*)}$。然后，考虑到射线的方向，我们要检查得出的 idx^* 是否实际有效。如前所述，我们通过翻转 *idx* 的一个或多个位来获得 idx^*，每个位对应射线穿过的轴对齐平面。要idx^*有效，翻转的方向必须与射线方向 **d** 的相应分量的符号一致。例如，如果 $d_x c$ *为* 0，则对应 x *轴的*位只允许增加。如果所有的翻转都与射线方向一致，我们使用 pos^* 和 idx^* 分别作为新的 $pos^{(\prime)}$ 和 $idx^{(\prime)}$，执行 ADVANCE 操作。如果遇到任何冲突的翻转，我们将继续执行 POP。

在 POP 的情况下，我们可以通过查看 *pos* 和 $pos^{(*)}$ *的*位表示来确定下一个体素。图 14 展示了立方体位置和子槽索引之间的联系。给定比特位置上的每个比特三元组构成与特定立方体比例相对应的子槽索引。从最高比特位置开始，子槽索引在八叉树中定义了一条从根节点到当前体素的路径。

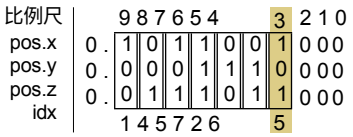


图 14： *正槽和子槽索引之间的连接。每个 pos 的位位置对应一个立方体比例值。将与标度对应的位三元组解释为整数后得到 idx。标度以上的比特定义了子槽索引的递增，形成了一条从根节点到当前体素的路径。scale 以下的位数为 0。*

让我们把与 *姿势* 和 *姿势** 与 *p* 和 *** 分别为 *p* 和 *p(*)*。我们知道，为了到达当前的体素，遍历必须遍历 *p* 沿线的所有体素，而 *p** 必须在路径的某个点偏离这组体素。光线从一个体素出来后，永远不会再进入该体素，这意味着 *p** 中第一个不同的体素必然是未访问过的。在深度优先遍历中，它也是我们下一步应该访问的体素。

因此，我们按以下方法确定下一个体素。首先，我们通过查找 *pos* 和 *pos** 之间不同的最高位来获得新的 *scale′*。然后，我们通过提取与 *scale(′)* 相对应的 *pos** 的位三元组，找到子槽索引 *idx(′)*。为了得到 *pos′*，我们取 *pos(*)*，但清除 *scale(′)* 下面的位。这样就得到了一个包含 *pos** 的具有正确比例的立方体。最后，我们从 *scale′* 的堆栈条目中恢复父体体素。

6.1 雷铸实施

图 15 给出了完整的光线投射算法的伪代码。代码由初始化阶段和沿射线遍历每个体素的循环组成。

算法首先在第 1-7 行初始化状态变量。射线的有效跨度存储为两个 *t* 值 $_{tmin}$ 和 $_{tmax}$ 之间的间隔，初始化为射线与根的。八叉树中的当前体素使用父子槽索引 *idx* 标识。通过比较 $_{tmin}$ 与八叉树中心的 $_{tx}$、$_{ty}$ 和 $_{tz}$，将其初始化为根节点的子节点。最后，*pos* 和 *scale* 被初始化为相应的立方体。

第 8-39 行循环迭代，直到射线击中一个体素或离开八叉树。每次迭代都会在第 11-16 行将当前体素与活动跨度相交，并有可能在第 18-25 行下降到其子节点。如果体素没有与射线相交，算法会在第 28-30 行执行 ADVANCE，然后可能第 32-37 行执行 POP。

第 9 行计算与当前立方体对应的跨度 *tc*，INTERSECT 和 ADVANCE 使用，第 10 行检查是处理当前体素还是跳过它。如果 *有效掩码* 中与体素对应的位未设置，或者活动跨度 *t* 为空，代码会判断射线无法与体素相交，并直接跳转到 ADVANCE。否则，该体素可能与射线相交，因此会被进一步处理。

第 11 行会检查体素是否足够小，以保证遍历的合理性。这提供了一种方法，通过动态调整体素分辨率以匹配屏幕分辨率来预先过滤几何图形，并通过比较 $\exp_2(scale)$ 与 $_{tcmax}$ 的线性函数来实现。检查可以在确定体素是否实际与射线相交之前执行，因为结果的准确性与非常小的体素无关。

```
2
1: (tmin , tmax )← (0, 1)
2: t← project cube(root, ray)
3: t← intersect(t, t′)4:
h← t(′)                    最大
初始化 6 4 5: 父← 根
6: idx← select child(root, ray, tmin )7:
(pos, scale)← child cube(root, idx )
8: while not terminated do
9:    TC← Project cube(pos, scale, ray) 10
:     if voxel exists and tmin≤ tmax then
2 11:    如果体素足够小，则返回 tmin 12:
         tv← intersect(tc, t)
13:      如果体素有轮廓，则
连接 6 4 14:    t′← Project contour(pos, scale, ray)
15:      tv← intersect(tv, t′)
16:      end if
17:      if tvmin≤ tvmax then
18:      如果体素是叶片，则返回 tvmin
2 19:    if tcmax< h then stack [scale]← (parent, tmax )
20:      h← tcmax
21:      parent← find child descriptor(parent, idx ) 22:
         idx← select child(pos, scale, ray, tvmin )23:    t
推 6 6 4 tv←
24:      (pos，scale）← child cube（pos，scale，idx
）25:    继续
26:      end if
27:   end if
"  28:   oldpos← pos
预付款 29:   (pos, idx )← 沿射线步进（pos, scale, ray） 30
:     tmin← tc(max)
31:   如果 idx 更新与 ray 不一致，则
2 32:      scale← highest differing bit(pos, oldpos) 33:
         if scale≥ smax then return miss
持久性 4 0 2 34:   (parent, tmax )← 堆栈 [标度]
35:   pos← round position(pos, scale)
36:      idx← 提取子槽索引（pos、scale） 37:
         h 0←
38:   end if
39: 结束 while
```

图 15： *光线投射算法的伪代码。*

第 12-16 行计算的跨度 *tv* 是当前立方体与活动跨度和体素轮廓的交点。与祖先体素相对应的轮廓效果包含在有效跨度中，因此 *tv* 代表与当前体素几何形状的精确交集。第 17 行检查交集是否为非空，如果是，则继续执行 PUSH。否则，执行 ADVANCE 跳过该体素。

如果 *从父* 节点的叶子 *掩码* 可以看出当前象素是叶子，那么第 18 行会终止遍历，因为已经找到了所需的交叉点。如果有必要，第 19 行会将 *父* 节点和 $_{tmax}$ *的*旧值存储到堆栈中。根据极限值 *h* 所做的决定如下：

- $_{tcmax}$= *h* 表示射线从体素和父体退出，在这种情况下，我们不需要存储 *父体*，因为不会再访问它。
- 由于 $_{tcmax}$≥ 0 始终为真，这样做的效果是防止同一个父节点再次被存储。

顺着层级往下，第 20-24 行会用当前的体素替换 *父体*，并设置 *idx*、*pos* 和 *scale* 以匹配射线进入的第一个子体素。最后，第 25 行重新启动循环，处理子体素。

第 28-30 行执行 ADVANCE。首先将当前立方体的位置存储到一个临时变量中，然后将 *pos* 和 *idx* 沿射线推进到下一个相同比例的立方体。第 9 行已经计算了决定沿哪条轴前进所需的所有 t 值，这里可以重复使用。最后，将 t_{min} 替换为射线进入新立方体的值，从而缩短射线的有效跨度。第 31 行会检查子索引位的翻转是否与射线的方向一致，即遍历是否停留在同一个父体象素中。如果是，循环将重新开始。否则，算法继续执行 POP。

第 32-36 行执行前面所述的 POP。如果新的*刻度*超过 s_{max}，第 33 行将判定射线退出八叉树，并以未命中终止遍历。最后，第 37 行将 h 设置为 0，以防止刚刚从堆栈中读取的父节点再次被存储。

为清晰起见，伪代码中省略了一些实现细节。最重要的细节概述如下。实际的光线投射器代码见附录 A。

替代坐标系。该算法包含多个必须明确检查射线方向符号的操作。可以通过镜像整个八叉树来重新定义坐标系，使 $\mathbf{d}$ 的每个分量都为负数，从而避免这些检查。在实际操作中，我们通过在初始化时根据射线方向确定一个八叉位掩码来实现这一点。然后，每当我们解释子描述符的字段时，就会使用这个掩码来翻转 *idx* 的位。同样，我们也可以偏移 *pos* 的原点，使其分量的范围在 [1, 2] 而[0, 1] 之间。这样做的好处是，我们可以直接操作 POP 中相应的浮点位表示。

缓存当前子描述符。每当算法执行 ADVANCE 时，父描述符在下一次迭代中保持不变。因此，只获取一次相应的子描述符来节省内存带宽是合理的。具体做法是在 PUSH 和 POP 中使其本地副本失效，只有当描述符失效时，才在循环开始时从内存中获取描述符。

在 INTERSECT 和 PUSH 之间共享计算。可以组织轮廓投影的计算，以便利用在体素中心评估的 t_x、t_y 和 t_z 值。在 PUSH 中，同样的值也用于确定射线进入的第一个子体素。因此，在 INTERSECT 开始时计算这些值并在 PUSH 中重新使用它们是有好处的。

命中位置。在大多数情况下，除了 t 值之外，还需要射线与体素几何体相交的位置。在实践中，简单地以 $\mathbf{p}+t\mathbf{d}$ 计算位置并不稳健，因为浮点误差会导致生成的点位于与相交体素相对应的立方体之外。因此，有必要明确地将位置的每个分量箝位到立方体的最小和最大坐标上。

6.2 光束优化

有一种相对简单的方法可以加快主光线的投射过程。利用立方体体素，可以渲染一个粗略的、保守的距离图像，然后用它来调整单条射线的起始位置。这样做的效果是使各条射线在与曲面相交之前跳过其起始位置的大部分空隙。

对于许多加速度结构来说，这种方法并不可行，因为通常无法保证粗网格射线不会遗漏对加速度结构非常重要的特征。

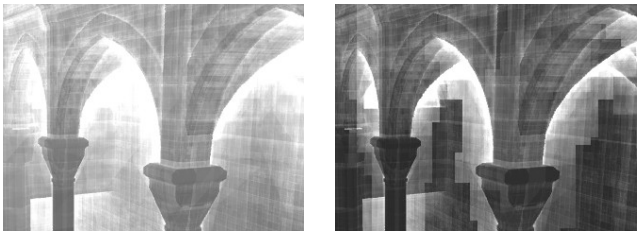


图 16：*横梁优化对迭代次数的影响说明。左图：未进行波束优化的* SIBENIK-D*。右图：在 8z8 块中进行了波束优化。两幅图中白色对应 64 次迭代。*

单条射线。不过，在使用体素数据时，如果体素的大小不足以覆盖粗网格中的至少一条射线，我们就可以通过终止遍历实现这一目的。需要注意的是，必须在粗网格中禁用轮廓测试才能实现这一功能。

在中，我们将图像划分为 4z4 或 8z8 像素块，并在粗渲染过程中为这些块的四角投射距离射线。在实际渲染中，对于每条射线，我们都要识别对应的图块，并获取四个角的距离值。然后从它们的最小值中减去一个适当的常数，以确定射线的起点。图 16 展示了光束优化对迭代次数的影响。

6.3 后处理过滤

为了消除阴影属性离散采样造成的阻塞感，我们在渲染图像的后期处理步骤中使用了自适应模糊过滤器。如果不使用滤波器，结果将与最近采样纹理查找的效果相似。需要注意的是，轮廓边缘通常可以通过轮廓线很好地表现出来，因此我们要避免在轮廓边缘过度模糊。

估算适当过滤半径的最可靠方法是查看屏幕上相交体素的大小。然而，由于体素分辨率可根据局部几何复杂性和阴影属性的差异而变化，因此同一表面上的相邻体素可能处于不同的中。因此，所需的滤波半径在整个表面上也会有所不同。如图 17 所示，在体素边界处滤波半径的突然变化会造成渲染伪像。

我们的方法基于稀疏的采样点，这些采样点按照与中心的距离升序存储在查找表中。我们使用一组 96 个样本，分布在一个半径为 24 像素的圆盘中。采样点的密度随着与中心距离的平方根而下降，每个采样点的权重与其所代表的圆盘面积相对应。平滑采样不足数据（如单样本阴影或反射）的算法往往需要两次处理（如 [Fernando 2005; Robi-son and Shirley 2009]），但有序查找表允许我们一次即可完成计算。我们在本文中使用的图像内核如图 18 所示。

图 19 给出了算法的伪代码。处理像素时，我们首先要根据像素本身的体素确定所需的过滤半径 r。如果半径为一个像素或更小，则无需过滤，我们将返回。否则，我们将按照查找表确定顺序开始处理采样点，直到它们与中心距离超过 r。为了使模糊半径适应邻域，我们将 r

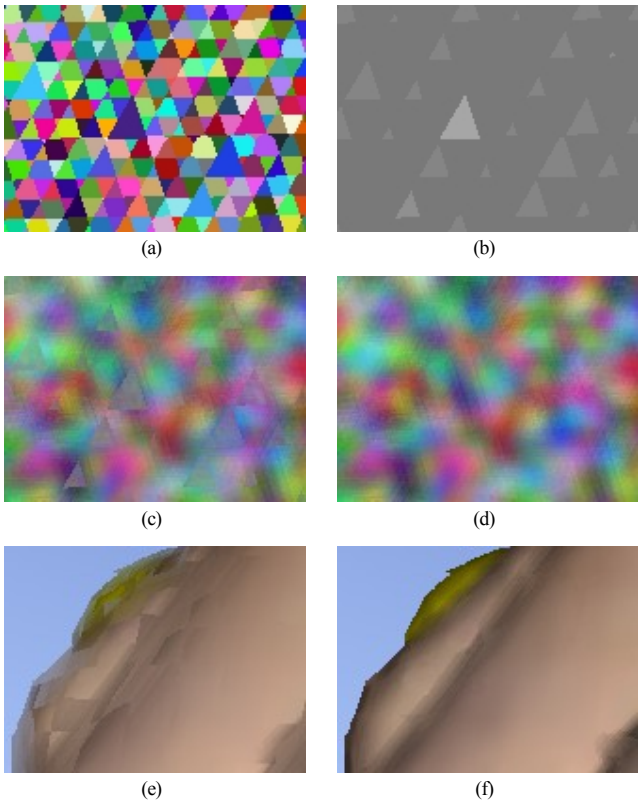


图 17： *滤波器半径变化的问题。(a) 斜面上的假色体素。(b) 每个点的八叉树深度，色调越亮表示树越浅，即体素越大。(c) 根据对应体素的大小推导出的半径对每个像素进行过滤。层级变化时可见接缝。(d) 我们的方法在累积相邻颜色的同时调整过滤半径，从而实现层次间的平滑过渡。(e)和(f)：标准方法和我们的方法分别应用于仙女手工绘制的图像，其中仙女手工绘制的图像经过了大量剪枝处理，体素层次较少。*

为 min(*r*，*r*′)。这样可以使附近区域的滤波半径一致，从而避免形成明显的接缝。累积权重的计算方法是提取样本权重，并对其进行调整，使其在 *r* - 1 和 *r* 之间线性地零。最后，根据计算出的权重累积颜色。

在实际操作中，我们会在投射射线时将体素大小的对数存储到结果图像的 alpha 通道中。以 3.5 值存储时，一个字节就足够了，范围从 1（无模糊）到约 128 像素。

6.4 失败的方法

我们尝试了许多看似可行的想法，但最终证明并不可行。本节从预期收益和观察结果的角度，收集了一些最基本的想法。

体素栅格化我们受 Ritschel 等人[2009]的微渲染技术启发，编写了一个简单的分层体素渲染器。每个线程都有一束 4 ⨯ 4 或 8 ⨯ 8 的射线需要处理。线程会遍历体素层次结构，以便剔除光束外的体素。这可以通过盒状与球面相交测试有效完成。每当遍历遇到一个投影大小约为 1 像素的体素时，线程就会开始处理、

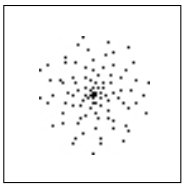


图 18： *本文图像使用的滤波核。样本存储在恒定存储器中，并按照与中心距离的递增顺序排列。每个样本的权重与其代表的圆盘面积成正比。核半径为 24 像素，有 96 个样本。从图 17d 中可以看出，这种特殊的核会造成一些带状伪影，但我们认为手动调整样本位置或使用更复杂的松弛方法可以大大改善这种情况。*

```
1: (c, r)← fetch(x, y) 2: if
r≤ 1 then return c 3:
accum← (0, 0, 0, )
4: 对内核中的每个样本 s 做
5: (c′, r′)← fetch(x+ s.x, y+ s.y)
6: r← min(r, r′)
7:     if s.dist> r then break
8: w←  s.weight - min(r - s.dist, 1) 9:
accum.rgb←     accum.rgb+  c′  -  10:
accum.w←  accum.w w+
11: 结束 for
12: 返回 accum.rgb/accum.w
```

图 19： *后处理过滤算法的伪代码。*

该像素会被绘制到本地内存中的一个小帧缓冲区中，并且不会访问该体素的子体素。

这种方法的主要优点是，由于只需对每个光束中的射线进行一次共用部分的遍历，因此需要完成的遍历工作大大减少。不过，也有两个主要的问题。首先，结果并不十分精确，因为体素是以点的形式绘制在帧缓冲区中的，没有进行精确的射线与体素对比测试。其次，大于一个像素的叶状体需要特殊处理。

除了这些问题外，各线程之间的一致性似乎也很差，这可能会使 SIMD 线程之间的执行和内存访问出现明显偏差。更新本地内存中的帧缓冲区时，线程之间没有一致性，因此更新成本很高。即使只更新深度缓冲区，不对大型体素进行特殊处理，光栅化器的速度也比光线投射器慢数倍。因此，我们停止了光栅化器的开发。

我们推测，光栅化的主要优势--共享遍历的公共部分--已经通过光线投射器的光束优化得到了充分发挥。此外，由于数据和执行一致性高，公共部分的执行成本也很低。

堆栈短。通过剖析（第 8.5 节），我们注意到光线投射器产生了大量的堆栈流量。因此，只在寄存器中存储堆栈顶部的内容似乎是减少堆栈流量的好办法。这就是所谓的*短堆栈* [Horn 等人，2007]。使用短栈的缺点是，每当栈耗尽时，遍历就必须从根重新开始。重启的成本相当高，但可以说重启的频率很低，因此总成本很小。

我们以前曾发现，在三角光线投射的情况下，短堆栈比全堆栈要慢一些，而在体素情况下，这种更为明显。原因是在这两种情况下，堆栈的用途不同。

对于三角形光线投射器来说，堆栈就像是要访问的节点列表，每次弹出都会移除最上面的一项。而对于体素，堆栈用于存储通往当前体素的路径，弹出的层级取决于射线在遍历时穿过的边界。例如，八叉树中间的轴对齐平面。当射线穿过其中任何一个平面时，我们需要一直弹出到堆栈的第一个条目，即最底层。与通常的弹出操作相比，这种随机访问方式更不适合短栈方法。

重新使用光线投射堆栈。这一优化与二次射线有关。由于光线投射堆栈总是包含通往当前体素的路径，因此可以将主光线的最终堆栈用作次光线的起点，前提是次光线的起点是主光线的终点。

在理想的情况下，这可以按原样工作，但在实际操作中，有必要将辅助射线的起点稍微移动一下，以考虑到误差。事实证明，只要偏移方向与射线方向的坐标八度相同，就可以解决这个问题，但偏移到其他八度就比较复杂了。因此，唯一实用的偏移方向是次射线的方向，幸运的是，这在大多数下都没有问题。不过，偏移量越大，堆栈的重复使用就越少。

我们仍不能完全确定堆栈重用为何不能加快渲染速度，但我们观察到，虽然辅助光线的迭代次数确实减少了，但整体性能却下降。一个可能的原因是，辅助光线都从自己的堆栈开始，这降低了整体的一致性。此外，体素层次结构中的初始下降非常连贯，而且成本很低。这样看来，即使为处理次射线起点偏移所需的额外代码，也是不值得的。

属性插值。我们花费了大量精力，试图以 GPU 的常规图形流水线处理纹理的方式，重新移动表面的块状结构。最初的插值属性获取实验令人沮丧--寻找邻域、解码和插值属性比实际的光线投射更昂贵。

由此可见，属性存储需要一定的冗余度，这样才能避免昂贵的邻居搜索。经过多次实验，我们最终确定了一种方案，即每个非叶子体素都有一个 3 × 3 × 3 的属性点网格，每个网格都可能有人或无人。这些点对应于子体素的角，而父体素面上的数据点会在相邻体素之间复制，因此无需查找邻居。

这种方案每个体素产生大约 4.5 个属性点，这是一个相当大的扩展因子。如果使用智能压缩方案，例如尽可能每个体素使用少于 8 个角，就有可能降低这一系数，但即使在最佳情况下，似乎也至少需要两倍于理想数据量的数据。

在这种结构中，属性查找速度相当快，因为不需要进行邻域搜索。不过，基于块的纹理压缩并不容易使用，因为压缩的自然单位是一组 8 个子节点，其中可能会占用多达 27 个属性点。最后，属性的构建变得非常困难。要保证相邻属性点之间的连续性并不难。

但要在不同的体素层次之间实现连续性却很困难。

目前看来，使用后处理模糊来平滑曲面要比执行插值属性查找更好。在不进行插值的情况下，内存中可以容纳更多的数据，而且在很多情况下，这种方法比插值更能提高质量。后处理模糊的另一个优点是可以平滑大模糊半径区域之间的轮廓，而这是插值无法实现的。

轮廓模糊在轮廓线出现之前，我们曾试图通过简单的模糊处理来掩盖剪影块状感。事实证明，这比最初想象的要困难得多。一个明显的问题是，模糊只能从表面向外进行，因为由于遮挡，我们无法知道轮廓后面是什么。这样做的后果是，当物体离摄像机越来越近时，就会显得越来越厚。对轮廓周围进行模糊处理还会导致本应非常清晰的细节变得模糊，例如在轮廓外就能看到的远处几何体。

我们尝试了各种方法来消除这些不必要的影响，但都没有找到合适的解决方案。最终，轮廓线以整洁、内存友好的方式消除了隐藏轮廓块的需要，同时提供了比立方体体素更好的几何分辨率。

7 数据管理

由于我们正在处理的数据集可能会大于 GPU 内存可容纳的容量，因此我们的数据格式被设计为适合按需进行局部分辨率调整。如第 4 节所述，当与摄像机的距离增加时，分辨率要求会迅速降低，我们希望能够利用这一点。

磁盘存储我们的八叉树数据是以 *切片*的形式存储的，这些切片保存在一个文件中，该文件一个切片目录。一个切片包含八叉树特定立方体部分中一个分辨率级别的体素数据。为便于按需流式传输，我们将一个切片可包含的数据量限制在 1 MB 左右。切片是按层次结构组织的，如果一个切片的下一个分辨率层次所消耗的内存超过了允许的容量，该切片就会在空间上被分割，使其拥有多个子切片。图 20a 展示了在磁盘上以切片形式存储八叉树的情况。

没有切片内容的切片层次结构很小，可以保存在内存中。因此，在特定空间区域找到包含所需分辨率数据的切片非常有效。在我们的实现中，所有数据更新都由 CPU 执行。

切片的内容与内存中块的内容有很大的不同，因为每当我们要加载一个切片时，我们的内存中已经有了它在八叉树中的所有上层体素。特别是，我们已经知道切片将包含哪些体素，因为父级包含了这些信息。因此，我们不需要在切片所包含的体素中存储任何空间位置、指针或索引。

内存存储。如第 5 节所述，八叉树的加载部分 *以块*的形式存储在 GPU 内存中。实际上，由于分辨率的动态变化，我们需要两种块。最高级别的八叉树存储在 *主干中*，主干是一个可以轻松分配和释放子描述符的池。主干以块的形式存储，这样光线投射器就不需要对主干中的体素进行不同的处理。