

硕 士 研 究 生 读 书 报 告



题目VDB: High-Resolution Sparse Volumes with Dynamic Topology

作者姓名 王继航

作者学号 22451211

指导教师 李启雷

学科专业 软件工程

所在学院 软件学院

提交日期 2025年1月

# 摘要

本报告详细介绍了VDB（Volumetric Dynamic Grid），一种三维体数据结构，旨在解决传统方法在处理稀疏、时间变化的数据时遇到的限制。VDB通过采用类似B+树的分层结构，结合了动态拓扑处理和高效的空间编码技术，实现了几乎无限的3D索引空间及高速缓存一致性访问。其核心组成部分包括根节点、内部节点和叶节点，每个节点都经过精心设计以优化存储和访问效率。VDB不仅支持快速的随机访问、顺序访问和模板访问，而且还具备出色的内存效率和自适应分辨率特性。VDB凭借其独特的优势，在处理复杂动态场景方面展现出了巨大的潜力。

**关键词：**VDB，三维重建, 动态场景

Abstract

This report provides an in-depth introduction to VDB (Volumetric Dynamic Grid), a 3D volumetric data structure designed to address the limitations encountered by traditional methods when dealing with sparse, time-varying data. By adopting a hierarchical structure akin to a B+ tree, VDB integrates dynamic topology processing and efficient spatial encoding techniques, achieving virtually infinite 3D indexing space and high-speed cache-consistent access. The core components of VDB include the root node, internal nodes, and leaf nodes, each meticulously designed to optimize storage and access efficiency. VDB not only supports rapid random access, sequential access, and stencil access but also boasts excellent memory efficiency and adaptive resolution features. With its distinct advantages, VDB shows tremendous potential in handling complex dynamic scenarios.

**Keywords：**VDB, 3D Reconstruction, Dynamic Scene

1 引言

在三维动画中，数据结构的选择对效率、性能和效果有着至关重要的影响。VDB是一种专为高效表示稀疏、时间变化的三维体数据而设计的数据结构。它结合了B+树的特性来处理动态拓扑和值，旨在优化存储空间的同时提供快速的数据访问。随着计算机图形学的发展，尤其是在流体模拟、云建模等应用中，对高分辨率三维体数据的需求日益增长，传统的数据结构在处理这类数据时遇到了瓶颈。因此，VDB作为一种新的解决方案被提出，以解决这些挑战。

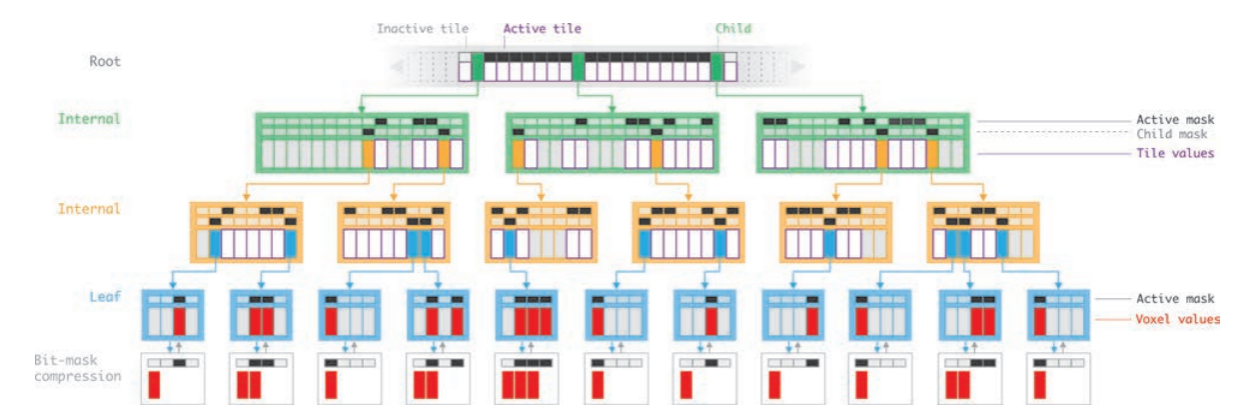
VDB通过利用空间相干性将数据值和网格拓扑分离编码，使得信息能够以紧凑的方式存储，并且可以模拟几乎无限的3D索引空间，从而允许对高分辨率的稀疏体数据进行高速缓存一致性的访问。这不仅提高了效率，而且支持了更为复杂的动态场景模拟。

对于三维动画来说，VDB支持恒定时间的随机访问操作，如查找、插入和删除，这对于实时编辑和交互式应用非常重要。此外，VDB还通过反向树遍历的方法改进了随机访问速度，进一步提升了性能。同时，顺序访问和模板访问也得到了优化，确保了连续数据读取和邻域查询时的高效性。这些特点让VDB成为流体模拟、云建模等复杂场景的理想选择，因为它不仅提高了渲染质量和速度，还简化了开发人员的工作流程，使他们可以更专注于创造逼真的视觉效果。

而在VDB中，其最核心的部分就是数据结构的具体实现方式以及基于此数据结构的数据访问方式。因此，本报告重点研究并讨论VDB的数据结构（第2章）和VDB的数据访问方式（第3章），以对VDB本质有更深入的了解。

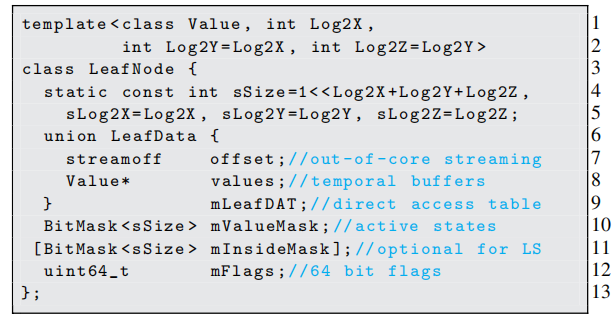
2 VDB数据结构

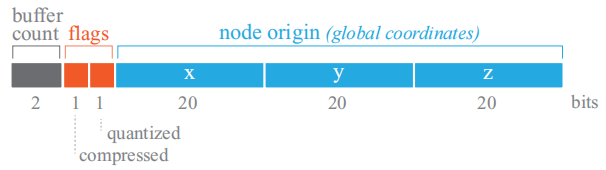
常见的VDB树一般树高为4，即包含一个根节点，两层中间节点以及一层叶节点，如图2-1所示。其每一层节点具体结构与相关实现将在本章节详细介绍。

图2-1 VDB树结构示意图

## 2.1 Leaf Node

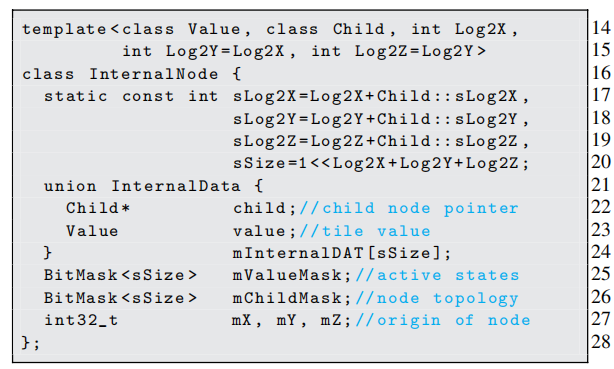
VDB树中叶节点对应于图2-1中蓝色节点，其结构主要包括以联合体形式存在的直接访问表mLeafDAT，标记是否存储有效体素值的掩码mValueMask，标记是否位于物体内部的掩码mInsideMask（仅在水平集应用程序中使用）以及一个64位的标识mFlags。其中mLeafDAT作为联合体，当数据量非常大以至于不能完全加载到内存时存储数据在外存（如磁盘）上的偏移位置offset，否则直接存储对应位置体素值values。对于64位的标识mFlags，其前2位确定缓冲区数目，第3位标识节点是否被压缩，第4位标识节点是否被量化，后60位分别存储全局节点的原点坐标，如图2-3所示。叶节点的大小由其在x，y，z三个维度的大小确定，标准状况下三个维度大小一致且均为3，则此时一个叶节点大小为8\*8\*8的块（为了便利于位运算，叶节点以及中间节点的维度大小均限制为2的幂次）。叶节点的具体实现代码如图2-2所示。

图2-2 Leaf Node 实现代码

图2-3 mFlags 结构示意

## 2.2 Internal Node

VDB树中间节点对应于图2-1中绿色和黄色节点，主要包含一个以联合体形式存在的直接访问表数组mInternalDAT[]，标记节点内部全部是否存储有效体素的掩码mValueMask，标记节点是否存在子节点的掩码mChildMask，以及分别以32位整形存储的节点原点坐标mX，mY，mZ。其中联合体数组mInternalDAT[]中每个元素，根据对应位置状态分别存放：若对应位置元素存在子节点，则存储指向子节点的指针child；否则存储对应位置元素的值value。中间节点的大小相较于叶节点，除了考虑自身对应方向的大小，还需要加入计算其子节点对应方向的大小。中间节点的具体实现代码如图2-4所示。

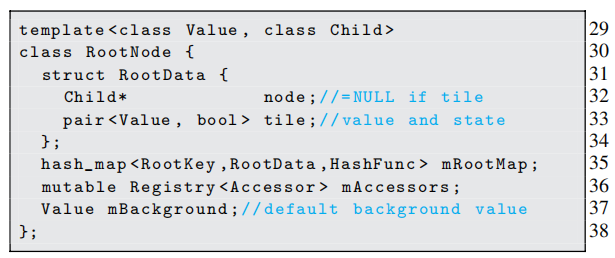
图2-4 Internal Node 实现代码

mValueMask和mChildMask分别对应于图2-1中Active mask和Child mask。对于Active mask，需其内部所有子元素都被填充，其才被置为1；对于Child mask，当其内部存在子元素被填充，同时不是全部子元素被填充时（存在子节点），其值置为1。如图2-5所示，假设该4\*4方格被划分为左上、右上、左下和右下四个部分，则其相应的Active mask值位0001，Child mask值位1100。

图2-5 Active mask与Child mask示例

## 2.3 Root Node

VDB树中根节点对应于图2-1顶部灰色节点，为概念上的无限网格域，具有小的内存占用，是稀疏并且可以动态调整大小的。其主要包括：一个存储根节点数据的结构体，若对应位置存在子节点，则存储指向子节点的指针node，否则存储对应位置元素值对tile，pair内部包括元素值以及其是否活跃的状态值；一个哈希表mRootMap，根据RootKey和对应的哈希函数HashFunc寻找对应数据RootData；一个可变注册表mAccessors，用于缓存访问器（Accessor），帮助优化遍历路径，加速随机访问；一个默认元素值mBackground。根节点具体实现代码如图2-6所示。

图2-6 Root Node 实现代码

3 VDB数据访问方式

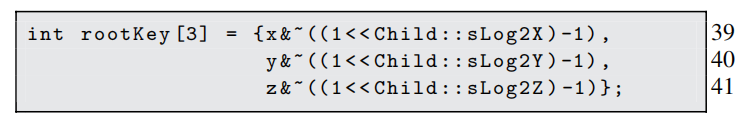
VDB树支持常见的多种数据访问方式，包括随机访问、顺序访问和模板访问，且各种访问方式都有着较好的时间复杂度。尤其对于随机访问，作者更是提出了以缓存的方式进行反向树遍历，进一步提升了访问效率。

## 3.1 Random Access

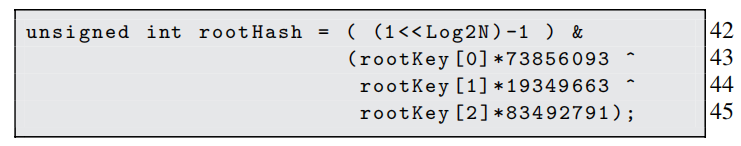
VDB支持恒定时间的随机访问操作，如查找、插入和删除，而且平均而言，这种操作与底层数 据集的拓扑结构或解析无关。相比之下，DT-Grid和H-RLE都不支持任何类型的随机插入或删除，并且随机查找在数据集的局部拓扑中具有对数的时间复杂度。

以访问索引坐标(x,y,z)处存储的体素值为例，具体随机访问流程大致可分为以下步骤：

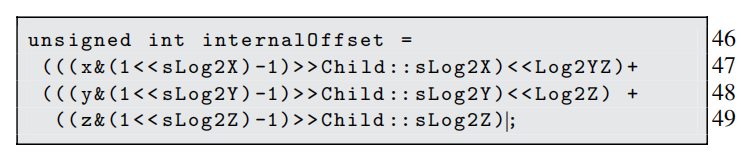
首先是根据索引坐标以及根节点下一级中间节点的大小（Child::sLog2X等）计算RootKey的值，如图3-1所示。其中~((1<<Child::sLog2X)-1)即为位运算的方式得到一个共有Child::sLog2X位且值全为0的掩码，再将其与x值按位与(&)得到x去除后Child::sLog2X位之后的结果，从而获得对应元素在整个VDB树中x方向上的起始位置（被去除的后Child::sLog2X位表示子节点内部的x方向上的相对位置），其余y,z方向同理。

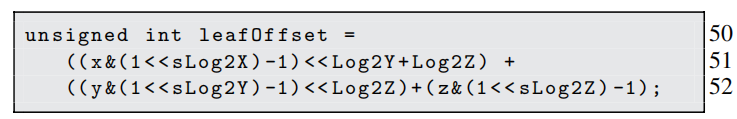
图3-1 RootKey计算实现代码

其次是本文作者对哈希函数进行了一定的优化，以减小哈希冲突，从而提升查找效率，如图3-2所示。Log2N为估计的根节点中哈希表mRootMap的大小，限制计算范围；rootKey中各方向值分别与三个大素数相乘并取异或可较好保证均匀分布从而减小哈希冲突。

图3-2 hash函数优化实现代码

最后，当访问层级进入中间节点或叶节点时，即可直接计算索引相对于全局起点的偏移量，并通过其中存储的直接访问表mInternalDAT或mLeafDAT直接查找得到对应的下一层级子节点或对应的体素值。中间节点的偏移量具体计算代码如图3-3所示，其中sLog2X表示该中间节点x方向上的大小，Child::sLog2X表示该中间节点下一层级节点在x方向上的大小，>>Child::sLog2X即表示计算该中间节点x方向偏移量时移除子节点在x方向上的大小。最终得到的偏移量将会以xyz的顺序重新拼接成一维索引，因此Log2Z表示拼接后的偏移量在z方向上的大小（Log2YZ = Log2Y + Log2Z）。叶节点的偏移量计算方式与中间节点几乎相同，仅无需考虑去除子节点信息一步，具体代码如图3-4所示。

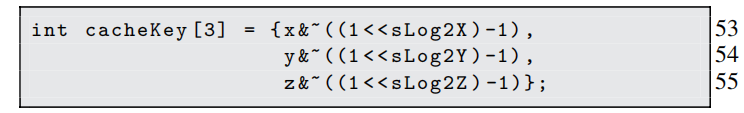
图3-3 中间节点偏移量计算实现代码

图3-4 叶节点偏移量计算实现代码

## 3.2 Improving Random Access

虽然理论上随机访问意味着每个访问都是独立的，但在实际应用中，大多数网格操作都具有某种程度的空间相干性。这意味着访问的体素通常集中在某些区域，而不是完全随机分布。因此，提高VDB随机访问效率的核心思想是通过反向树遍历来改进随机访问，进而通过追溯缓存的访问模式来促进随机访问。缓存在上一次访问操作中访问的节点序列，允许后续访问操作从下到上遍历树。

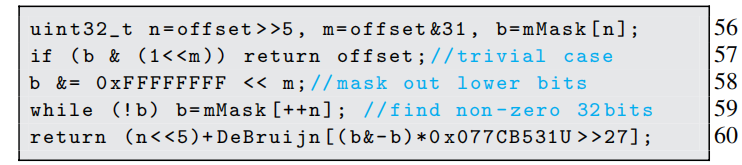
因而，本文优化的随机访问主要包括：将前一个根中访问的所有节点缓冲到一个小缓存列表中，该列表的大小等于树的深度；当访问进行到列表某一元素时，计算对应的缓存编码cacheKey，计算代码如图3-5所示。cacheKey的计算类似于RootKey，只是此时考虑的不再是子节点的大小，而是对应节点本身的大小，即判断对应节点与缓存列表此时元素的起始位置是否一致，若一致则表示要访问的点即在此节点中，否则继续查找缓存列表下一元素再进行比对，直至出现一致或者缓存列表到达根节点，再进行正常自上而下的遍历。

图3-5 cacheKey计算实现代码

## 3.3 Sequential Access

顺序访问的核心问题就是如何定位下一个活动值或者子节点，而基于VDB数据结构解决这个问题十分简单：借助每个节点中嵌入的极其紧凑的mChildMask和mValueMask直接访问位掩码。它们的紧凑性使它们对缓存很友好（不需要加载和搜索更大的直接访问表mInteralDAT和mLeafDAT），访问效率高。

如题3-6所示为顺序访问实现代码。其中offset是相对于mMask数组起始位置的偏置。n 是 offset 的高位部分，表示当前处理的 mMask 数组中的索引。m 是 offset 的低5位，表示在当前32位块内的具体位置，式b & -b 为找到最低设置位（即值为1的最低位）。

图3-6 顺序访问实现代码

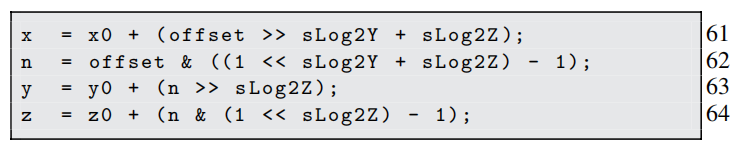
## 图3-7为顺序访问示例。右上角红色方框内mMask表示左图自上而下所有行的掩码，标红掩码以及后续相关计算皆对应于左图中左边黑色圆圈圈出的体素。右下角红色方框表示查找继左图左侧黑色圆圈内体素后的下一活动体素，最终对应于左图右侧黑色圆圈内体素。

## 图3-7 顺序访问示例

## 3.4 Stencil Access

模板访问是指同时访问某个体素或像素的邻域内的一组特定位置。DT-Grid和H-RLE都通过将多个顺序迭代器分组来实现恒定时间的顺序模板访问，每个元素都针对支持模板中的一个元素。因此，这种方法的计算开销与模板的大小成线性关系。VDB将单个顺序迭代器与改进的随机访问技术结合起来。顺序迭代器访问模板中心点的位置，再通过加速随机访问的形式访问其余各点。

图3-8为对模板中心点索引的计算。其中offset是以xyz顺序拼接而成的一维索引；x0,y0,z0表示节点的全局起始坐标，对于叶节点为其mFlags中后60位（如图2-2和2-3所示），对于中间节点则为其存储的mX,mY,mZ（如图2-4所示）。根据节点的全局其实坐标，再加上对应方向上的偏移量，即可得到模板中心的索引坐标。再以缓存列表的形式，采用加速随机访问可以较高效率访问到模板中的其余各点。

图3-8 模板中心点索引计算实现代码

4 **总结**

VDB是一种优秀的三维体数据结构，在多个方面表现出色。从概念上来讲，它可以模拟几乎无限的3D索引空间，适用于各种规模的数据集；从实现角度来看，它具有良好的内存效率和自适应分辨率的能力；而在数据访问方面，则展示了优秀的随机访问速度以及高效的顺序和模板访问能力。这些特点共同构成了VDB的独特优势，使其成为处理稀疏、时间变化的三维体数据的理想选择。

深入探讨VDB的特性，其设计核心在于对稀疏性数据的有效管理。传统的体素化方法在表示空洞或低密度区域时浪费大量资源，而VDB通过仅存储非零值或重要信息来显著减少内存占用。这种高效的数据压缩方式不仅节省了存储空间，而且提高了缓存命中率，进而提升了渲染速度和其他操作的性能。VDB采用的动态拓扑结构允许随着数据的变化自动调整网格的精细度，确保了不同分辨率下的表现一致性。

此外，VDB在数据访问上的优化同样不可忽视。它支持恒定时间的随机访问操作，包括查找、插入和删除，这对于实时编辑和交互式应用至关重要。这意味着平均而言，这些操作的时间复杂度不受底层数集的拓扑结构影响，从而保证了稳定且快速的响应。对于需要遍历所有活动体素的情况，VDB提供了高效的顺序访问算法，利用紧凑的位掩码直接访问数据，避免了加载更大访问表所带来的开销。同时，针对特定邻域的查询，即模板访问，VDB实现了单个顺序迭代器与改进的随机访问技术相结合的方式，有效减少了计算开销，特别是对于较大模板的支持更加出色。

总之，VDB凭借其独特的优势，在处理复杂动态场景方面展现出了巨大的潜力。它不仅解决了传统体数据结构在面对高分辨率、稀疏分布数据时遇到的问题，还为开发者提供了一个灵活且强大的工具，以创造出更逼真、更高效的三维动画效果。无论是流体模拟还是云建模，VDB都能够提供出色的性能，是现代计算机图形学不可或缺的一部分。

# 参考文献

[1] Museth, K. 2013. VDB: High-resolution sparse volumes with dynamic topology. ACMTrans. Graph. 32, 3, Article 27 (June 2013 ) 22 pages. DOI:http://dx.doi.org/10.1145/2487228.2487235.