VDB 数据结构

Dezeming Family

2023 年 4 月 9 日

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列电子书，可以从我们的网站 [https://dezeming.top/] 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

目录

一　基本介绍

VDB 可以用于存储几乎无限大小的 $3D$ 索引空间，允许快速的数据访问。它对体数据的稀疏性没有拓扑限制，并且在插入、检索或删除数据时支持快速（平均 $O(1)$）的随机访问模式。这与大多数现有的稀疏体数据结构形成了对比：现有的稀疏体数据一般假设静态或流形拓扑，并需要特定的数据访问模式来补偿缓慢的随机访问。

由于 VDB 数据结构基本上是分层的，它也有助于自适应网格采样，并且固有的加速结构导致了非常适合物理模拟的快速算法。因此，VDB 已被证明对一些需要大型、稀疏、动画化体的应用程序非常有用。

VDB 数据结构具有存储效率，支持对随时间变化的数据的模拟，能够对任意拓扑进行编码，并有助于统一和自适应采样，同时允许快速和不受限制的数据访问。主要特性就是动态 (Dynamic) 和内存高效 (Memory efficient)，即可以实现动态拓扑和动态变化的值，且是内存效率高的稀疏数据结构。

列举几个之前常用的相关数据结构：

- 八叉树 (Octree)、kd-Tree(k-dimensional Tree) 和 BSP tree(Binary Space Partitioning Tree) 等基本数据结构。
- 网格稀疏块划分 (Sparse block-partitioning of grids)。
- DT-Grid[2]。

VDB 是一系列相关工作的积累，这些工作都在 SIGGRAPH 上有过发表：Dynamic Blocked Grid(DB-Grid)，是 Museth 等人 2007 年的成果。之后，Museth 等人在 2009 年和 2011 年提出了大幅度提升的 hierarchical DB+Grid，把 DB-Grid 与 B+Trees 进行了结合。然后重新命名为 VDB 结构。

VDB 可以描述为一个紧凑的数据结构和相应的一些算法组成。这些内容之间都相互独立，因此可以分开介绍。本文是根据 [1] 来讲解的，但是对一些晦涩难懂的地方增加了大量的解释说明。

二　VDB 数据结构原理

关于八叉树和 B-Tree、B+Tree，Dezeming Family 网站都有文章介绍，有需要可以自行查找。

VDB 结构很像 B+Tree，在层次数据结构 (hierarchical data structure) 下动态分配的网格块，块 (block) 是在非循环连通图 (acyclic, connected graph) 的相同固定深度处的叶节点，该图具有大但可变的分支 (branching factors)，这些分支数被限制为 2 的幂。VDB 的高度在结构上是平衡的，且又浅又宽，不但能增加数据范围（数据量，domain），还减少了从根节点到叶节点遍历树所需的 I/O 的操作量（八叉树的 I/O 操作量其实还是挺高的，可以类比二叉树与 B+ 树的区别）。

然而，类似于八叉树或 N-trees，VDB 也可以对非叶节点中的值进行编码，用作自适应多级网格。但是，将 VDB 定性为仅仅是一个广义八叉树或 N-Tree 是并不准确且比较浅的，因为 VDB 的真正价值是它的独特实现，正如后面会介绍的那样，它有助于数据访问和动态数据操作。

我们的 VDB 在每个节点都可以有大量、可变的分支数，它更像一个 B+Tree，但是也有一些不同：

(1) • 标准 B+ 树仅在叶节点处编码真实的数据，其他位置仅仅存 keys；VDB 在树的所有节点中都会编码由其空间坐标索引的网格值。换句话说，VDB 是一种多级数据结构，不采用传统 B+ 树意义上的 keys。

(2) • 另一个区别是，B+ 树的叶节点经常彼此链接，以允许快速的顺序迭代；VDB 则不是这样，后面会介绍 VDB 更有效的策略，使得 VDB 不要在插入或者删除节点时还要维持一个链。

(3) • 此外，B+ 树采用对数复杂度的随机查找，VDB 提供恒定时间随机访问。尽管如此，VDB 数据结构可以被视为 B+ 树在稀疏体网格应用中的专门的优化。

另一个有趣的比较是与现代 CPU 中的内存层次结构的设计。与 VDB 一样，它们采用固定数量的缓存级别，这些级别的大小不断减小，随机存取性能不断提高（可以简单理解为，上层节点因为包括了很多子节点，区域更大，所以缓存量更大；下级节点的缓存量就会减少），并且只有最顶层的主存储器才能动态调整大小。我们将在后面详细阐述这种类比。

数据结构算法只有在其软件实现允许的情况下才是有效和有用的。VDB 的许多吸引人的功能可以完全归因于在内存占用和计算性能方面的这种高效实现。因此，在接下来的讨论中还会包括 C++ 实现的几个细节。这不仅应该允许读者再现 VDB，而且考虑到它所解决的问题的复杂性，还可以证明它是多么简单。

我们广泛使用 C++ 模板元编程和内联，并一贯避免使用虚函数，这被证明是一个重要的优化，导致了计算高效的随机访问。更具体地说，树节点类是在其子节点类型上递归模板化的，而不是从公共基类继承。原因是模板化的节点配置在编译时是内联扩展的，因此一些模板函数没有额外的运行时开销。这种方法的另一个优点是它允许我们轻松地自定义树的深度和分支因子，但要在编译时而不是运行时进行。其他值得注意的实现技术包括快速位操作、紧凑的位存储、用于内存重用的类型联合，以及线程和 SIMD 操作的直接实现。

在接下来的内容中，我们专注于 $3D$ 网格的实现，尽管为了清晰起见，我们的大多数图示都是 $1D$ 或 $2D$，但要注意 VDB 可以针对任何空间维度实现。

2.1　术语

VDB 理论上包含的空间可以无限大的，但受限于内存和数据精度，也不能无限大。

编码到 VDB 中的数据由 Value 类型（由 C++ 模板定义）组成，以及其对应的坐标 (x,y,z) 来指定其空间样本位置（即树结构中的 value 的 topology）。

为了方便起见，我们偶尔会使用符号 w 来表示三个笛卡尔坐标方向 (x, y, z) 之一。我们将索引空间的最小体积元素称为体素，如图1中的红色阴影所示。

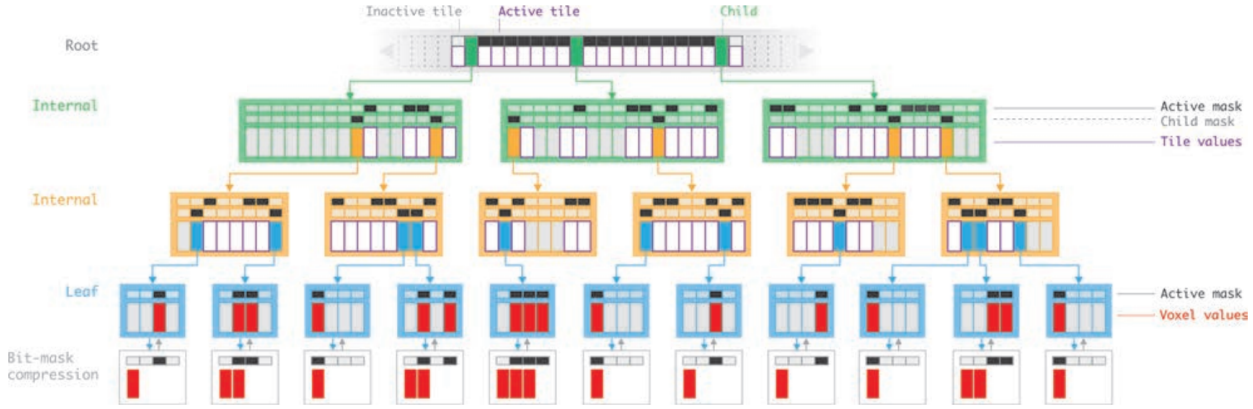


图 1: 1D Illustration of a VDB

图1：有一个 RootNode（最上方灰框）的 VDB 的示意图，有两个内节点 (InternalNodes) 层级（绿框和橙框），以及叶节点（蓝框）。根节点的大小可变，而且是稀疏的，其他节点都是稠密的，且每层的分支数都会下降两倍（比如绿色层每个节点 16 个分支，橙色层是 8 个分支，蓝色层只有 4 个分支）。

每个节点在图中都有一个比较大的数组（内部节点有三行数组，上面两行块比较小，下面一行块比较大），分为三种：哈希映射表 (hash-map table)mRootMap、直接访问表 (direct access tables)mInternalDAT 和直接访问表 mLeafDAT。这些表编码指向子节点的指针（绿块、橙块或者蓝块）、编码指向上层片段值 (upper-level tile values) 的指针（灰块/白块，可以理解为，在这个节点下的全部体素的值都是相等的，要么都是属于非活动区，要么都是某个相同的值。tile 值其实可以理解为，整个片段下的元素都是该值）、编码指向体素值的指针（红块/灰块）。注意底部的白色节点显示压缩的 LeafNodes，其中非活动体素（灰色）已被移除，只留下活动体素（红色）。

直接访问表上方的小数组说明了对局部拓扑进行紧凑编码的紧凑位掩码 (compact bit masks)，例如 mChildMask 和 mValueMask。

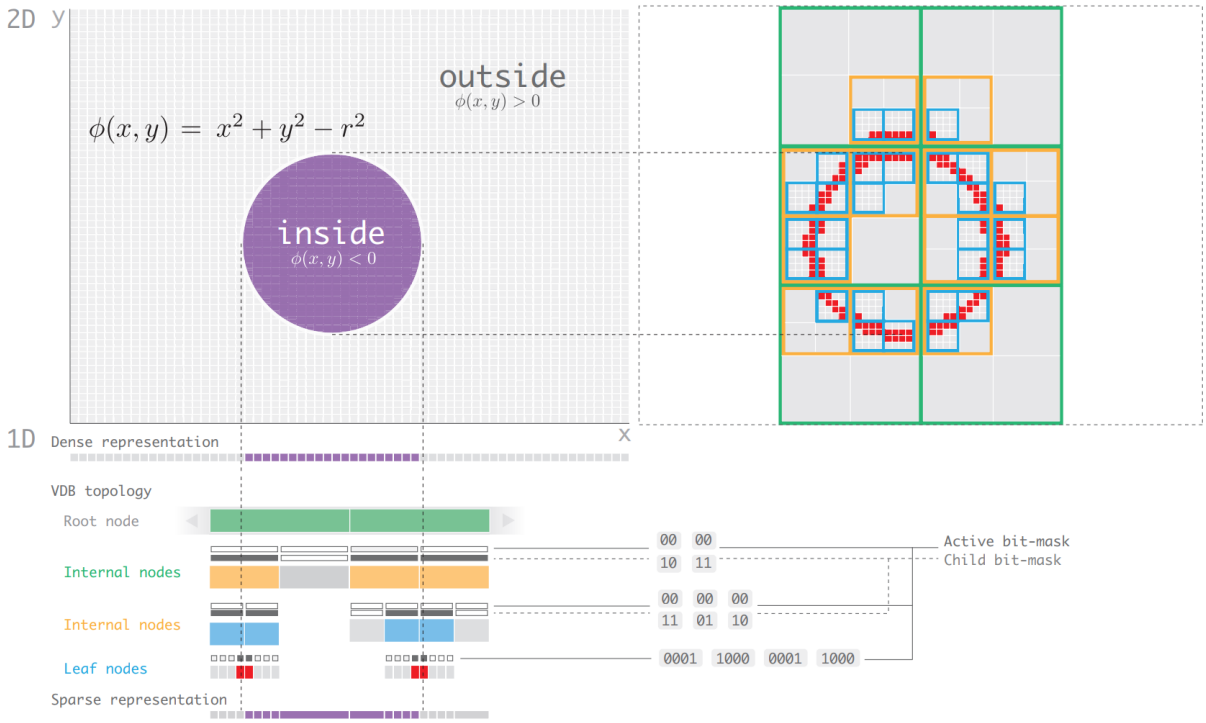


图 2: Illustration of a narrow-band level set

水平集也是一种比较有用的数据结构，图2是分别在 1D 和 2D VDB 中表示的圆的窄带水平集的图示。左上角：圆的隐式有符号距离，即水平集，在均匀密集网格上离散化。底部：1D VDB 的树形结构，表示窄带水平集的单行。右上角：与 2D 窄带水平集的 VDB 表示相对应的自适应网格的图示。2D VDB 的树形结构太大，无法显示。体素对应于最小的正方形，tiles 对应于较大的正方形。树的每一级都选了比较小

的分支数 (branching factors)，比如橙框内有 4 个蓝框，绿框内有 4 个橙框，这样避免视觉混乱；在实践中，通常要大得多，而不只是 4 个。

单个值 value 与每个体素相关，VDB 中的每个这样的体素可以存在于两种离散状态中的一种，即活动或非活动状态。这种二元状态的解释是特定于应用的，但通常活动体素比非活动体素更“重要”或“感兴趣”。例如，在标量密度网格中（比如用于表示烟雾的体网格），非活动体素具有默认背景值（例如该网格内的烟尘微粒密度为 0），而活动体素的值不同于该默认值。对于窄带水平集 (narrow-band level sets)（位于零水平集周围很窄的一个带状区域水平集函数，其实在图2上可以理解为是表示勾勒圆的边界的像素集），窄带内的所有体素都是活动的；而所有其他体素都不是活动的，并且具有恒定的非零距离，其符号表示内部拓扑与外部拓扑。

VDB 在树中单独编码体素拓扑，该树的根节点覆盖所有索引空间，并且其每个叶节点覆盖索引空间的固定子集。更确切地说，拓扑被隐式地编码到位掩码中，并且值被显式地存储在位于树的任何级别的缓冲区中。所有体素具有相同值的索引空间区域可以用存储在树的适当级别的单个值来表示，如图2所示（比如一个层级下所有的体素值都一样，就在这个层级表示下面所有的体素值）。我们将采用“tiles”一词来表示这些较高级别的值 (upper-level)。与体素一样，tiles 可以是活动的，也可以是非活动的。VDB 的总体目标是只消耗表示活动体素所需的内存，同时保持典型密集体积数据结构的灵活性和性能特征。

在接下来的 C++ 代码片段中，我们使用以下约定：带前导大写的标识符，例如 LeafNode，是类、类型或模板参数名称；带前导“s”的 camel 大小写中的那些（例如“sSize”）表示常量静态数据成员，带前导“m”的 camel 大小写中，例如“mFlags”表示常规成员变量。所有小写标识符，例如“x”，都表示非成员变量。

2.2 构建块

虽然 VDB 可以以许多不同的方式进行配置，但在讨论了所有节点的基本数据结构后，我们将描述所有配置的通用组件，从叶节点开始，到根节点结束。

Direct access bit masks

VDB 的一个基本组成部分是嵌入在树结构的各个节点中的位掩码。它们提供了对节点局部拓扑的二进制表示的快速而紧凑的直接访问，据我们所知，我们是第一个同时使用它们进行以下操作的人：（1）分层拓扑编码，（2）快速顺序迭代器，（3）无损压缩，（4）布尔运算，以及（5）拓扑扩展等算法的有效实现。所有这些操作对于应用于流体、窄带水平集和体建模的动态稀疏数据结构至关重要。我们稍后将更详细地讨论这四个应用程序以及位掩码的显式部署。

Leaf nodes

图1和2中的蓝色节点是最底层 (lowest-level) 的网格块，所有的蓝色节点都有同样的树深度。它们有效地将索引空间平铺到沿着每个坐标轴具有 2^{Log2w}（注意 w 是任意一个坐标轴）个体素的非重叠子域中，其中 $Log2w = 1, 2, 3, ...$，比如当 $Log2w = 3$ 时，相当于每个叶节点内都有 $8 \times 8 \times 8$ 个体素。我们将叶（和内部）节点维度限制为 2 的幂（这里的维度即每个节点的子节点个数），因为这允许在树遍历期间进行快速位操作。

可以看到，叶节点维度在编译时就已经确定了：

```
1 template < class Value, int Log2X, int Log2Y =Log2X, int Log2Z=Log2Y >
2 class LeafNode {
3     static const int sSize = 1<< Log2X + Log2Y + Log2Z, sLog2X = Log2X,
        sLog2Y =Log2Y, sLog2Z = Log2Z;
4     union LeafData {
5         // out-of-core streaming
6         streamoff offset;
7         // temporal buffers
8         Value* values;
```

```
    } mLeafDAT; // direct access table
    // active states
    BitMask <sSize > mValueMask;
    // optional for LS
    [BitMask <sSize > mInsideMask];
    // 64 bit flags
    uint64_t mFlags;
};
```

叶节点体素数计算为：$1 << \sum_w sLog2w$，其实就是几个维度相乘：$2^{sLog2x} \times 2^{sLog2y} \times 2^{sLog2z}$。叶节点把体素值编码到一个直接访问表 (Direct Access Table)mLeafDAT 里，以及将活动体素 (active voxel) 拓扑转换为直接访问位掩码 mValueMask。直接访问表在本文中表示最坏情况下随机访问复杂度为 $O(1)$ 的元素数组。

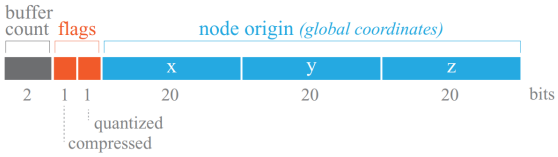
需要注意的是，尽管位掩码的固定大小等于 LeafNode 的大小，但由于以下原因，值数组 (value array) 的大小 mLeafDAT.values 是动态的：

(1) • 首先，对于一些应用，可以通过各种压缩技术显著减少 mLeafDAT 的内存占用。我们支持几种不同的编解码器，包括利用活动体素拓扑的编解码器（即 mValueMask）和水平集的内部/外部拓扑（即 mInsideMask），以及更传统的 entropy-based 和 bit-truncation 方案。细节可以在论文 [1] 的附录 A 中找到。

(2) • 其次，为了最好地促进数值时序 integration，LeafNode 可以包含多个值缓冲区，每个缓冲区存储不同时间级（time level）的体素。例如，三阶精确 TVD RungeKutta 方案需要三个时序缓冲 (temporal buffers)。当我们的体数据需要计算的内容与时间相关，需要考虑体素变化前后的值，则就需要这种缓冲。

(3) • 最后，体素值也可以位于核心之外 (out-of-core)（即可能数据量太大，需要存储到磁盘上，而不能直接存储到内存里，则 offset 用来记录这个节点在文件流中的偏移量，比如该节点在文件的第 1208 个字节处），这得益于偏移 (offset) 到文件流中。请注意，此偏移量不会产生额外的内存开销，因为它是在 mLeafDAT 的 C++ union 中编码的。

LeafNode 的 value buffers 的大小和变量数以及其他信息都是被紧凑地编码到了 64 位的变量 mFlags 里：



前两位对四种状态进行编码：0 个缓冲区，即值是 out-of-core 的，1 个缓冲区（即不支持时序 integration 的 in-core 值）；2 个缓冲区，即支持一阶和二阶时序 integration 的 in-core 值；3 个缓冲区，支持三阶时序 integration。如果块被压缩，则第三个 bit 是 1，如果叶是 bit-quantized，则第四个 bit 是 1。

最后，mFlags 的剩余 3×20 个位用于对虚拟网格中节点的全局原点 (global origin) 进行编码。这是非常方便的，因为这个叶中的某个体素的全局体素坐标 (global voxel coordinates) 可以通过将 mValueMask 中编码的局部体素拓扑与 mFlags 中的全局节点原点相结合来导出。

为了最好地利用 20 个 bit，这个叶的原点会除以 LeafNode 大小，即 $X >>= Log2X$，然后保存到这 20 个 bit 中，其中 X 表示节点沿 X 轴的原点，$>>$ 是逐位右移。如果节点为 8^3，即 $Log2X = 3$，则允许在每个坐标方向上超过 800 万个网格点的网格域（$0xFFFFF \times 8 \approx 840$ 万），这已被证明对我们的所有应用程序来说绰绰有余。然而，在 OpenVDB 中，通过以完全 32 位精度对原点进行编码来克服这种限制。

因此可知，LeafNodes 是自包含的，不需要引用它们的父节点就能求出叶内的任何体素所在全局的位置，这减少了内存占用并简化了体素坐标的计算。

再通俗的介绍一下上面的内容，就是说，每个叶 LeafNode 都会有一个全局的位置，即 global voxel coordinates；同时，每个叶中的体素在这个叶上也都有对应的偏移；这样，给定任意一个叶中的任意一个体素，用这个偏移和全局的位置就能求出叶节点在全局的位置，这样会很方便。如果不在叶中记录全局的位置，那么想知道一个体素的全局坐标就得向上访问父节点，来得到父节点在全局的位置，然后再计算出当前叶中的该体素的全局位置。

Internal nodes

内部节点类如下：

```
template <class Value, class Child, int Log2X, int Log2Y=Log2X, int Log2Z=
    Log2Y>
class InternalNode {
    static const int sLog2X = Log2X + Child::sLog2X,
    sLog2Y = Log2Y + Child::sLog2Y,
    sLog2Z = Log2Z + Child::sLog2Z,
    sSize = 1<< Log2X + Log2Y + Log2Z;
    union InternalData {
        Child * child; // child node pointer
        Value value; // tile value
    } mInternalDAT [ sSize ];
    BitMask <sSize > mValueMask; // active states
    BitMask <sSize > mChildMask; // node topology
    int32_t mX, mY, mZ; //origin of node
};
```

可以看出，它们与 LeafNodes 共享了几个实现细节。分支数可通过模板参数 Log2w 进行配置，将分支限制为 2 的幂，这有助于高效的树遍历。然而，与 LeafNodes 不同，InternalNodes 能编码树拓扑，还能编码值，即指向其他内部或叶节点的指针。这是通过直接访问表 mInternalDAT 中的 union 结构有效地实现的。相应的拓扑结构被紧凑地编码在比特掩码 mChildMask 中，并且 mValueMask 用于指示 tiles 值是否处于活动状态。注意，由于分支数 Log2w 在编译时是固定的，因此 mInternalDAT、mChildMask 和 mValueMask 的大小也是固定的。然而，重要的是要强调，允许不同树级别的内部节点具有不同的分支数，从而为树的整体形状增加灵活性（比如图1中绿色的内部节点有 16 个分支数，橙色内部节点有 8 个分支数）。这并不重要，因为内存占用和计算性能可能会受到树配置的影响，后面会进行描述。

Root node

这是树操作通常开始的最高节点。我们注意到 InternalNode 也可以作为根节点，但是有很大的局限性。由于到目前为止引入的所有节点都具有模板化的分支数，因此相应网格的整个域在编译时是有效固定的。对于极端的分辨率，这将需要较大的分支数，这反过来又会由于密集的直接访问表 mInternalDAT 而导致内存开销（第2.3小节对此进行了详细介绍）。因此，为了实现概念上无边界的网格域，具有较小的内存占用，需要稀疏的动态根节点。这样的根节点如图1中的灰色节点所示，其结构如下。

```
template < class Value , class Child >
class RootNode {
    struct RootData {
        Child * node; //= NULL if tile
        pair <Value, bool> tile; // value and state
    };
    hash_map <RootKey, RootData, HashFunc> mRootMap;
```

```
    mutable Registry<Accessor> mAccessors;
    Value mBackground; // default background value
};
```

VDB 的所有配置最多有一个 RootNode，与所有其他树节点不同，它是稀疏的，可以动态调整大小。通过对子指针或 tile 值进行编码的哈希映射表可以实现这一点。如果表条目 (table entry) 表示 tile 值（即 child=NULL），则布尔值表示 tile 的状态（即活动或非活动）。需要注意的是，根据设计，mRootMap 通常包含很少的条目，这是由于 tile 或子节点表示的巨大索引域，例如 4096^3。事实上，在论文第五节中展示了像 std::map 这样的红黑树数据结构可以比需要计算更昂贵的哈希 keys 的复杂哈希映射执行得更好。

无论如何，对动态稀疏数据结构（如 map 或哈希表）的随机访问通常比对固定密集直接访问表 mInternalDAT 的查找慢。换句话说，与快速的 InternalNode 相比，所提出的 RootNode 访问慢似乎是主要的缺点。

然而，对于 mRootMap 访问速度慢的问题，有一个让人惊奇的简单而高效的解决方案。RootNode 包含一个访问器注册表 (registry of Accessors)Accessor，它可以通过重用以前访问中缓存的节点指针来执行自下而上的树遍历，而不是自上而下的树遍历来显著提高空间一致性网格访问 (spatially coherent grid access)。这种反向树遍历有效地摊销了在根节点级别启动的完全遍历的成本。虽然我们将把实现细节留给第3.2小节，但我们需要强调的是，大多数实际的访问模式，即使是看似随机的模式，通常都表现出某种类型的空间一致性。

最后，mBackground 是访问空间中未解析为子节点内的 tile 或体素的任何位置时返回的值（背景值，比如对于描述烟尘粒子密度的体网格，该值可以设为密度 0）。

2.3 把它们合在一起

任何空间数据结构的单一配置 (single configuration) 都不能声称能同样好地处理所有应用程序，VDB 也不例外，因此它是专门为定制而设计的。节点及其参数的不同组合可以改变树的深度和分支数，这反过来又会影响可用网格分辨率、自适应性、访问性能、内存占用甚至硬件效率等特性。我们将讨论此类配置的一些通用指南，然后提出一种平衡大多数上述因素的配置，该配置已被证明对很多应用程序都有用。

让我们考虑一些极端配置，这些配置将有助于启发 VDB 的平衡配置，即平铺网格 (Tiled grids)、哈希图 (hash maps) 和 N 树 (N-trees)。下面前三部分就是对比一些其他的结构，可以略过。

Tiled grids

假设我们决定将网格配置为一个直接连接到多个 LeafNode 的 InternalNode。这是大多数现有平铺网格背后的概念，除了它们通常在顶层使用非静态直接访问表 (nonstatic direct access table) 之外。

由于从根到叶的路径极短，并且所有节点都使用快速直接访问表，因此我们期望有高效的随机访问性能。然而，由于大量的内存开销，这种快速访问是以有限的网格分辨率为代价的。这是因为网格的可用分辨率是由树的每个级别的分支数的乘积给出的。因此，如果需要高网格分辨率，则 InternalNode（当前为顶部节点）和多个 LeafNode 的分支数需要较大。相反，通常希望保持 LeafNodes 的大小相对较小，原因有两个：缓存性能和具有少量活动体素的部分填充 LeafNode 的内存开销。这通常意味着顶级节点的分支数需要比 LeafNodes 的分支数大几个数量级，这反过来又从顶级密集直接访问表引入了显著的内存开销。为了实现 8192^3 的适度网格分辨率和 8^3 的 LeafNode 大小，即使在插入体素值之前，顶部 mInternalDAT 的存储也是巨大的 8GB（$8192^3 = 0x8000000000$）。最后，由于该特定配置本质上对应于平面树 (flat tree)，因此它不支持有效地存储具有常量（upper-level）tile 值的索引空间的连续区域。同样重要的是，它缺乏加速结构，这对分层算法很重要。

Hash maps

在这一点上，细心的读者可能会推测，解决上述内存问题的有效方法只是用使用哈希映射表的稀疏根节点替换密集的 InternalNode。

然而，仔细考虑会发现一些性能问题，首先是在 RootNode 中注册的 Accessors 在这种简单的配置下效率不是很高。为了理解为什么让我们回顾一下 Accessor 背后的想法；通过重用缓存的子节点，将缓慢查找的开销分摊到 mRootTable 中（当我们在访问节点时，会把叶节点加载缓存，以便后续查找能够更快）。

现在，正如前面强调的那样，在表示稀疏体时使用小的 LeafNodes 通常是有利的。因此，RootNode 的子节点将覆盖非常小的索引域，这反过来意味着 Accessor 中的缓存未命中。

换句话说，这种配置的随机访问的复杂性主要由较慢的 mRootMap 的瓶颈决定。此外，由于 LeafNodes 很小，mRootMap 通常会包含许多条目 (entries)，由于哈希 keys 和对数搜索复杂度的冲突，这些条目可能会进一步削弱查找性能。对于窄带水平集，这种情况甚至更糟，因为 mRootMap 还必须包含表示接口内部恒定的所有 tiles 值；请参见图2中的很大区域的绿色 tiles。虽然好的散列函数 (hash functions) 可以最大限度地减少冲突，但它们往往比直接访问表中的（完美的）keys 在计算上更昂贵，并且由此产生的 hash key 将随机排列插入的数据。因此，即使对 LeafNodes 的空间相干访问也可能导致随机内存访问，这通常会导致模拟的缓存性能较差。最后，所提出的树配置显然是平坦的，因此缺乏对快速分层算法和自适应采样的支持。

N-trees

现在考虑在所有级别上具有固定分支数的另一个极端情况，例如 Log2X=Log2Y=Log2Z=1，对应于高度平衡的八叉树。在每个坐标方向上有两个极小的分支数，这显然是自适应网格采样的最佳选择。然而，它的随机访问速度较慢。这是由于对于给定的树深度 D，对应的分辨率仅为 $2D$。对于 8192^3 的适度网格分辨率，八叉树必须至少有 13 级深。这种情况可以通过增加分支数来改善，从而类似于最初针对静态体提出的 N 树。

但是，如前所述，大型节点可能会损害缓存性能并增加内存占用。对于大型 LeafNodes 尤其如此，这可能导致在编码稀疏数据时活动体素与非活动体素的比例不利。此外，由于树的连续级别之间的分辨率变化过于剧烈，因此较大的常量分支数可能导致较差的网格自适应性。

最后，由于 VDB 的深度是通过设计固定的，固定的分支数将导致固定的网格分辨率，这在处理动态体积时通常是不希望的。

总结

我们总结了配置 VDB 时需要考虑的观察结果：

- 与分支数和树深度有关的分辨率尺度；
- 对于较短的树，随机访问通常更快；
- 自适应地支持小分支数和很深的树；
- 内存随着节点的数量和大小而变化；
- 需要在使用较小的节点和较深的树时提高缓存重用性。

考虑到这些看似矛盾的指导方针，似乎只有为每个网格应用程序定制树配置，才能实现最佳性能。然而，我们已经确定了一类能够平衡大多数这些因素的树，至少对于本文中介绍的水平集和稀疏体应用程序来说是这样。这种配置可以被视为之前讨论的极端情况的最佳特征的混合：平坦的平铺网格、哈希图和规则的 N 树结构。它是一种短而宽的高度平衡 B+ 树，有小的叶节点，通常有三到四级深，从下到上分支数不断增加。

一个例子是一个具有动态 RootNode 的树，下面是两个级别的 InternalNodes，静态分支分别为 32^3 和 16^3，后面是大小为 8^3 的 LeafNodes，也就是说，RootNode 的每个子节点都跨越 4096^3 的索引域。图2显示了具有类似轮廓的三级 VDB 的 $1D$ 示例。

或者，根节点可以被分支数为 64^3 的 InternalNode 取代，这导致 VDB 具有 262144^3 的固定可用分辨率，最坏情况下是常量时间随机访问。

论文第 5 节研究了这些配置和许多其他配置的性能，并将证明它们提供快速随机访问和高效顺序访问，支持极端分辨率，具有层次性和自适应性，并且具有相对较小的内存开销和良好的缓存性能。

一个奇怪但有用的变体是只通过位掩码对网格拓扑进行编码，但不存储数据值的 VDB。这种 VDB 非常紧凑，可以用于常规 VDB 的体素上的拓扑算法，例如，膨胀、侵蚀或窄带重建，或者用于仅需要索引空间的二进制表示的应用，即体素化掩模。

我们还试验了可以动态垂直生长而不是水平生长的树，但结果令人失望。我们发现，这样的配置使插入和删除等数据访问模式的有效实现复杂化，并使我们无法在编译时知道深度和分支数时优化数据访问。最后，固定的深度导致恒定时间的随机访问。当我们在下一节中讨论实现细节时，这些考虑因素应该会变得更加清晰。

三　VDB 数据结构的访问操作

到目前为止，我们只关注 VDB 数据结构，它只占我们贡献的一半，可以说是更简单的一半。另一半涉及一个高效算法和优化技巧的工具集，用于导航 (navigate) 和操作这种动态数据结构。

我们将在本节中重点讨论树访问算法，并在下一节中讨论更多特定于应用程序的技术。换句话说，我们将第二节中介绍的数据结构与本节和下一节中讨论的算法的组合统称为 VDB。

空间数据结构有三种通用的访问模式：随机 (random)、顺序 (sequential) 和模板 (stencil)。接下来我们将描述如何在 VDB 中有效地实现这些功能。随后，我们将介绍对动态体很重要的各种其他算法，如水平集。

3.1　随机访问

最基本但也是最具挑战性的模式是随机访问任意体素。这种模式的区别在于在最坏的情况下，每个体素访问都需要从根节点开始，可能在叶节点结束，对树进行完全自上而下的遍历。

然而，在实践中，通过反转遍历顺序可以显著改善随机访问，我们将把这个主题推迟到下一节。因此，更容易使用随机访问的对立定义，也就是说，它是任何既不是顺序的 (sequential based) 也不是基于模板 (stencil based) 的访问。后两者很容易被定义为具有固定模式的访问，该模式是从内存中的底层数据布局或一些预定义的邻域模板 (neighborhood stencil) 中定义的。

Random lookup

随机查找是最常见的随机访问类型，应作为所有其他访问的前奏。我们首先确定从根节点开始有效遍历树结构所需的基本操作。显然，这些操作取决于 mRootMap 的实际实现，所以让我们从简单的 std::map 开始。为了访问索引坐标 (x, y, z) 处的体素，我们首先计算以下有符号的 rootKey：

```
int rootKey [3] =
    {x &~((1 << Child :: sLog2X ) -1),
     y &~((1 << Child :: sLog2Y ) -1),
     z &~((1 << Child :: sLog2Z ) -1)};
```

在编译时，该操作减少到只有三个硬件 AND 指令，它们屏蔽了与相关联的子节点的索引空间相对应的低位。结果值是包含 (x, y, z) 的子节点的原点坐标，从而避免了哈希冲突。然后使用该完美 key 来执行对 mRootMap 的查找，mRootMap 以三个 rootKey 分量的字典顺序存储 RootData（参考前面的代码）。如果未找到条目，则返回背景值（mBackground），如果找到 tile 值，则返回 upper-level 值。遇到这些情况，遍历就会终止。

如果找到子节点，则遍历将继续，直到遇到 tile 值或达到 LeafNode 为止。将 std::map 替换为 hash map 所需的唯一修改，如快速 google::dense_hash_map，是一个生成均匀分布随机数的很好的 hash 函数。为此，我们将前面给出的完美 rootKey 与不完美的散列函数（本文中“哈希”和“散列”是混用的，意思相同）相结合。

```
unsigned int rootHash =
    ((1<< Log2N )-1 ) &
    (rootKey [0]*73856093 ^
     rootKey [1]*19349663 ^
     rootKey [2]*83492791);
```

Log2N 是一个静态常数，用于估计 mRootMap 大小的以 2 为底的对数。注意论文已经改进了 Teschner 等人 [2003] 中的原始哈希函数，用更快的逐位 AND 运算取代了昂贵的模运算。我们从这个看似微小的变化中观察到了三个重要的好处：计算性能提高了 2 倍以上，哈希函数在负坐标下正常工作，并且不太容易出现 32 位整数溢出。使用标准的皮尔逊卡方检验 (Pearson’s chi-squared test)，我们已经验证了修改后的哈希函数在有符号坐标域上保持了其一致性。众所周知，哈希映射渐近地比简单的 std::map 具有更好的时间复杂度，但应该清楚的是，rootHash 的计算成本比相应的 rootKey 更高。结合这一点，在实践中，mRootMap 预计会非常小，并且不清楚哪种实现更快，这是论文第 5 节解决的问题。

当遇到 InternalNode（在顶层或内部级别）时，从全局网格坐标导出以下直接访问偏移：

```
unsigned int internalOffset =
    (((x&(1 << sLog2X )-1)>> Child :: sLog2X )<< Log2YZ )+
    (((y&(1 << sLog2Y )-1)>> Child :: sLog2Y )<< Log2Z ) +
     ((z&(1 << sLog2Z )-1)>> Child :: sLog2Z );
```

其中 Log2YZ = Log2Y+Log2Z。在编译时，这减少到只有三个逐位 AND 运算、五个位移位和两个加法。接下来，如果 mChildMask 的位 internalOffset 为 off，则 (x, y, z) 位于一个常量 tile 内，因此 tile 值从 mInternalDAT 返回，遍历终止。否则，将从 mInternalDAT 中提取子节点，并继续遍历，直到在 InternalNode 的 mChildMask 中遇到零位 (zero bit) 或到达 LeafNode。LeafNode 的相应直接访问偏移计算速度甚至更快：

```
unsigned int leafOffset =
    ((x&(1 << sLog2X )-1)<< Log2Y + Log2Z ) +
    ((y&(1 << sLog2Y )-1)<< Log2Z )+(z&(1 << sLog2Z ) -1);
```

这是因为它减少到只有三个逐位 AND 运算、两个逐位左移和两个加法。让我们做一些重要的观察。首先，除了散列函数中的三次乘法之外，所有这些计算都是用单指令位运算执行的，而不是像除法、乘法和取模这样慢得多的算术运算——这是因为所有分支数都是 2 的幂。其次，所有关键点和偏移都是从相同的全局网格坐标 (x, y, z) 计算的，也就是说，它们是级别无关的，不递归的。这是因为在编译时树的深度和分支数是已知的。

最后，所有这些逐位计算即使对于 (x, y, z) 的负值也是有效的。关于原理可以参考论文 [1]。

由于 VDB 通过构造是高度平衡的，具有运行时固定的深度，因此从 RootNode 到 LeafNode 的每条路径都一样长，我们得出结论，叶值的每次随机查找都涉及相同的最坏情况时间度。

随机访问存储在树的较浅深度的 tile 值显然更快，因为遍历提前终止，但它仍然受到访问存储在叶节点中的体素值的计算复杂度的限制。考虑到 InternalNodes 和 LeafNodes 都使用直接访问表，具有 $O(1)$ 最坏情况下的随机访问时间，应该清楚的是，VDB 的总体复杂度由 RootNode 的复杂度给出。如果在顶层使用 InternalNode，则 VDB 的最坏情况随机查找复杂度为 $O(1)$。另一方面，如果 RootNode 是用 std::map 实现的，则复杂度为 $\log(n)$，其中 n 是 mRootMap 中的条目数，如前所述，由于子节点的大小，该条目数预计非常低。然而，如果 RootNode 使用具有良好哈希函数的优化哈希映射，则平均复杂度变为 $O(1)$，而最坏情况下的时间为 $O(n)$。

只有完美的哈希才能在最坏的情况下允许常量的时间查找，这在理论上对于真正的无界域（即无限数量的哈希 key）是不可能的。然而，在实践中，我们预计很少有条目具有足够统一的哈希密钥，因此我们得出结论，即使使用支持无界域的 RootNode，VDB 的随机查找也具有平均恒定的时间度。

Random insert

随机插入通常在初始化网格时使用，但也可以在动态数据的上下文中发挥重要作用，例如重建水平集的窄带。遍历是使用快速位操作执行的，但现在如果 mChildMask 中的相应位关闭，则会分配一个子节点。遍历终止于（可能是新构建的）LeafNode，体素值设置在所需的时间缓冲区中，相应的位设置在 mValueMask 中。

由于 RootNode 下方的节点仅在插入时分配，因此稀疏体数据的内存占用率较低。尽管节点的这种动态分配意味着随机插入可能比随机查找慢，但开销通常在多个相干插入操作上分摊，并且平均而言，它具有恒定的时间复杂度。

Random deletion

随机删除是处理动态数据时需要效率的操作的另一个例子。遍历的实现类似于随机插入，只是现在在 mChildMask 和 mValueMask 中未设置位，并且如果节点不包含子节点或活动体素，则节点会被修剪掉

(pruned away)。通过同时检查 mChildMask 和 mValueMask 中的多个位，例如，通过 64 位算术同时检查 64 个条目，可以有效地实现自下而上执行的修剪。

总之，VDB 支持恒定时间的随机访问操作，如查找、插入和删除，这平均与底层数据集的拓扑或分辨率无关。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时间复杂度。

3.2 提升的随机访问

在实践中，几乎所有的网格操作都具有一定程度的空间一致性（可以理解为短时间内多次访问到同一个区域），因此很少以真正随机的模式访问网格。事实上，无论底层数据结构如何，都应该避免真正的统一随机访问，因为它代表了内存重用方面最糟糕的情况。对于真正的随机访问，计算瓶颈通常是 CPU 的内存子系统，而不是数据结构本身。空间一致性的极端情况是模板或顺序访问，稍后将分别进行研究。然而，即使对于非顺序和非模板访问，也经常存在一些可利用的空间一致性。

其核心思想是通过反向树遍历来改进随机访问，而反向树遍历又通过回溯缓存的访问模式来促进随机访问。我们不是从 RootNode 初始化每个随机访问，而是缓存在前一次访问操作中访问的节点序列，允许随后的访问操作从下到上遍历树，直到到达公共父节点。平均而言，随机访问的空间聚集越大，遍历路径越短，因此加速越大。特别地，如果根节点的子节点足够大以引入所有实际访问模式的平均空间一致性，则可以摊销其较慢的访问时间。因此，均衡效率，如果我们在多次随机访问时都能命中缓存区，且缓存区缓存的节点不特别大（特别大，比如直接把包含一大片区域的内部节点缓存，效率就不会很高），那么效率就会很高

这种技术本质上类似于采用多个缓存级别的现代 CPU 的内存层次结构中的缓存机制。在这种类比中，RootNode 对应于主内存，InternalNodes 对应于 L3/L2 缓存，LeafNodes 对应着 L1 缓存。正如访问一级缓存中的数据比任何其他内存级别都快得多一样，访问缓存的 LeafNodes 中的值也比任何其他树级别都快。此外，即使当前 CPU 中高速缓存级别的相对大小也类似于平衡 VDB 树结构中的节点大小。CPU 中的内存层次结构的深度是固定的，通常由主内存（RAM）的一个大的、相对较慢的、可动态调整大小的部分和三个级别的逐渐变小、更快和固定大小的缓存级别组成，完全模仿了图1所示的 VDB 配置。通过使用非分层虚拟地址空间（VAS），CPU 物理内存层次结构的这种复杂性对操作系统是隐藏的。为了加速从 VAS 到各种数据缓存中物理内存页面的映射，CPU 使用了一个 Translation Look-aside Buffer(TLB)，它将以前的查找从虚拟内存缓存到物理内存。为了完成 VDB 的 CPU 比喻，我们可以将网格坐标 (x, y, z) 视为虚拟内存地址，将指向树节点的指针视为包含 (x, y, z) 的物理内存页的地址。因此，为了加速对 VDB 的随机访问，我们基本上是在为我们的树结构寻找 TLB 的类比。

虽然前面概述的缓存可能看起来很简单，但实际上高效实现并不是一件小事。问题是，上一小节中详细介绍的随机访问操作已经很快了，因此计算开销的空间很小。这排除了使用标准哈希表来存储缓存的树节点。详细过程可以参考论文 [1] 原文，值得注意的一点是，在最坏的情况下，列表的从下而上的遍历会一直进行到根级别，在这种情况下，启动常规的自上而下的树遍历，并完全重建缓存列表。

3.3 Sequential Access

许多算法访问或修改网格中的所有活动体素，但对发生这种情况的特定序列是不变的。特别是，对于大多数与时间相关的模拟，如水平集传播 (level set propagation) 或流体平流 (fluid advection)，情况也是如此。

如果我们能够定义一种优于随机访问的顺序访问模式，那么这种不变性就可以被利用。我们将数据访问的最佳顺序称为“顺序访问”，该顺序由数据在内存中的物理布局顺序给出。随着现代 CPU 中复杂的缓存层次结构和预取算法的进步，按照数据存储在主存 (main memory) 中的顺序提取和处理数据变得越来越重要。

类似地，与随机访问相比，空间数据结构通常具有与顺序访问相关联的显著更低的计算开销。DT 网格和 HRLE 都以全局坐标 (x, y, z) 的字典顺序存储网格点，并且它们仅为这种类型的顺序访问提供恒定时间查找和插入。相比之下，VDB 的特点是随机访问和顺序访问的时间复杂度都是恒定的。挑战在于实现顺序访问，使其优于前面概述的快速随机访问。

这个问题相当于定位下一个活动值或子节点，解决方案非常简单：迭代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友好（无需加载和搜索更大的直接访问表 mInternalDAT 和 mLeafDAT），并且可以同时处理多个条目（位）。

3.4 Stencil Access

均匀网格上高效的模版访问是有限差分 (FD) 计算的基本要求。这些方案近似于在称为支持模板的局部邻域中具有离散网格值差的微分算子。其他常见的模版应用是具有局部支持的卷积核的插值和滤波。这种模板的尺寸和形状可以根据 FD 方案的精度和类型而广泛变化。例如，最佳五阶精确 WENO FD 方案在 3D 中使用 18 个相邻网格点，而一阶单侧 FD 仅使用 6 个相邻网格点。通常，这些模板访问方法与顺序访问相结合，从而产生模板迭代器 (stencil iterators)，这是许多数值模拟的基本概念。

DT-Grid 和 H-RLE 都通过对多个顺序迭代器进行分组来实现恒定时间顺序的模板访问，支持模板中的每个元素都有一个迭代器。因此，这种方法的计算开销与模版的大小成线性比例。使用 VDB，我们可以采取一种更简单、更高效的方法，不需要昂贵的多个迭代器同步。相反，我们将第3.3节中的单个顺序迭代器与第3.2节中详细介绍的改进的随机访问技术相结合。迭代器定义模板中心点的位置，访问器 (Accessors) 提供对其余模板点的加速访问。由于模板通常很紧凑，因此这是节点缓存的理想应用程序。例如，对于非常大的模板，在卷积过程中，我们甚至可以使用多个访问器，每个访问器都与模板的一个子集相关联，以减少触发更昂贵的树遍历的 LeafNode 缓存未命中。因此，在理论上，VDB 对于模版访问具有与 DT 网格相同的恒定时间复杂性，但在实践中，由于高速缓存重用，VDB 分摊了检索多个模版元素的开销。

最后，我们注意到一些应用程序实际上需要随机访问模板。示例是用于体渲染的光线步进过程中的插值和法线计算。从目前的讨论中可以明显看出，VDB 也支持这种类型的恒定时间访问。为了保护用户免受本节中详述的各种访问技术的复杂性的影响，它们都被巧妙地封装在类似 STL 的高级迭代器和访问器中，具有易于使用且不需要了解底层数据结构的 setValue 和 getValue 方法。

四 VDB 数据结构的应用

在本节中，我们将重点介绍对 VDB 在模拟和稀疏动态体积中的实际应用具有重要价值的算法和技术。此外，论文附录 A 和 B 描述了 VDB 如何实现高效而简单的压缩和 out-of-core 流以及多线程和矢量化 (vectorization) 等优化。

4.1 拓扑形态学运算 (Topological Morphology Operations)

除了体素访问方法外，动态稀疏网格上的一些最基本的操作是基于拓扑的形态学操作，如膨胀和侵蚀。这些操作在现有集合周围添加或删除额外的体素层，类似于添加或删除洋葱环 (rings of an onion)，它们用于变形、基于卷积的滤波、重采样、ray marching 等过程中的界面跟踪 (interface tracking)。

需要注意的是，这些操作纯粹在网格拓扑上工作，而不是在体素值上工作。由于 VDB 支持随机体素的恒定时间插入和删除，因此使用这些简单的访问方法来实现拓扑形态学是很有吸引力的。例如，可以通过在现有体素上迭代包含六个最近邻的邻居模板 (neighborhood stencil)，并随机插入与模板相交的所有新网格点来实现扩展。这种方法的一个缺点是，它通常需要两个网格副本：一个用于迭代，另一个用于存储膨胀。另一个主要缺点是，这种算法涉及每个体素的许多冗余随机访问操作，因为大多数模板将与现有的活动体素相交。幸运的是，有一种更有效的方法，它利用了 VDB 分别编码拓扑和值的这一事实。由于拓扑被编码到 mValueMasks 中，我们可以直接根据这些位掩码来制定拓扑算法，如论文第 5.5 节所示，这比使用随机访问操作要快得多。

4.2 Numerical Simulation on VDB

这里简要描述 VDB 如何用于时间相关的模拟。我们将在水平集上演示这一点，但强调其他与时间相关的偏微分方程，如 Navier-Stokes 方程，可以使用类似的技术求解。

水平集方法通常由两个基本步骤组成。首先，通过求解 Hamilton-Jacobi 方程来更新体素值，其次，通过重建窄带来更新拓扑，使其跟踪移动界面 (moving interface)。

第一步在 LeafNodes 中根据积分方案的需要分配尽可能多的时间值缓冲区，然后使用适用于用于离散 Hamilton-Jacobi 方程的空间有限差分方案的模板迭代器顺序更新体素。

4.3 Hierarchical Constructive Solid Geometry

构造实体几何（CSG）是隐式几何表示（如水平集）最重要的应用之一。事实证明，VDB 的底层树结构充当加速数据结构，有助于有效的 CSG 操作。事实上，我们可以在非常高分辨率的级别集上实现布尔运算的接近实时的性能。

技巧非常简单：我们通常可以在一次操作中处理 VDB 树的整个分支，而不是逐个体素执行 CSG。因此，计算复杂度仅随着相交的 LeafNodes 的数量而缩放，而相交的 Leaf Nodes 通常很小。

4.4 Hierarchical Boolean Topology Operations

虽然 CSG 在概念上对几何体执行布尔运算，但有时直接在两个网格中的活动值的拓扑上定义布尔运算是有用的，特别是在网格类型不同的情况下。例如，将标量网格的活动值拓扑与向量网格的活动值域拓扑并集可能很有用。

4.5 Mesh To Level Set Scan Conversion

构造体的一种常见方法是将现有的多边形网格 (polygonal mesh) 转换为符号距离函数。将显式几何图形转换为隐式表示的过程通常称为扫描转换。虽然有几种用于扫描转换的快速算法，但它们都基于规则密集网格，因此对可实现的分辨率有很大限制。

VDB 非常适合稀疏扫描转换，这使得它对电影制作很有吸引力，因为电影制作中的几何图形通常是明确建模的。最后，我们注意到，在将其他类型的几何体转换为体时，对 VDB 中随机插入的支持也非常重要。

4.6 Hierarchical Flood-Filling

前面描述的网格或粒子的扫描转换仅在闭合流形窄带内产生具有活动体素的稀疏 VDB，即 LeafNode 值。对于窄带之外的非活动体素和 tiles，需要做更多的工作来确定正确的有符号值。

幸运的是，来自窄带中的活动体素的符号的向外传播可以用 VDB 有效地实现，并且在整个扫描转换中产生可忽略不计的计算开销。

4.7 Accelerated Ray Marching

VDB 相对于现有的级别集数据结构（如 DT 网格）和平铺网格（如 DB 网格）的另一个好处是，它的分层树结构可以作为 Ray Marching 的多级边界体加速结构。

窄带级集合数据结构只能跨越与窄带宽度相对应的距离，平铺网格只能跨越与单个平铺大小相对应的区域。然而，VDB 可以通过递归地将光线与树的任何级别的节点相交来跨越更大的空间区域。这种想法的一种可能的表现是分层的，即多级的 3D Bresenham 算法，该算法提高了由多级 tiles 值表示的索引空间的空区域或恒定区域中的光线积分的性能。

关于加速结构如何实现空域跳跃（比如渲染参与介质时，跳过粒子密度为 0 的区域），可以参考《基于八叉树的体渲染优化》一文。

五　小结

在研究使用 VDB 时，很多内部原理我也没有完全弄懂，但使用时可以根据一些案例来应用，尤其是 NanoVDB 的应用更广泛，也被 Nvidia 的 Optix 所支持。

VDB 的开发者 Ken Museth[4] 有着长达十年的加速结构研究历史，而且四五十岁时依然活跃在技术研发的第一线，是一个很厉害又可敬的人物。

参考文献

[1] Museth K. VDB: High-resolution sparse volumes with dynamic topology[J]. ACM transactions on graphics (TOG), 2013, 32(3): 1-22.

[2] https://code.google.com/archive/p/dt-grid/

[3] https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/

[4] https://ken.museth.org/Biography.html