VDB: High-Resolution Sparse Volumes with Dynamic Topology

KEN MUSETH
DreamWorks Animation

We have developed a novel hierarchical data structure for the efficient representation of sparse, time-varying volumetric data discretized on a 3D grid. Our "VDB", so named because it is a Volumetric, Dynamic grid that shares several characteristics with B+trees, exploits spatial coherency of time-varying data to separately and compactly encode data values and grid topology. VDB models a virtually infinite 3D index space that allows for cache-coherent and fast data access into sparse volumes of high resolution. It imposes no topology restrictions on the sparsity of the volumetric data, and it supports fast (average $O(1)$) random access patterns when the data are inserted, retrieved, or deleted. This is in contrast to most existing sparse volumetric data structures, which assume either static or manifold topology and require specific data access patterns to compensate for slow random access. Since the VDB data structure is fundamentally hierarchical, it also facilitates adaptive grid sampling, and the inherent acceleration structure leads to fast algorithms that are well-suited for simulations. As such, VDB has proven useful for several applications that call for large, sparse, animated volumes, for example, level set dynamics and cloud modeling. In this article, we showcase some of these algorithms and compare VDB with existing, state-of-the-art data structures.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Physically based modeling*

General Terms: Algorithms

Additional Key Words and Phrases: Volumes, level sets, implicit surfaces, fluid animation

ACM Reference Format:

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

1. INTRODUCTION

Volumetric data is essential to numerous important applications in computer graphics, medical imaging, and VFX production, including volume rendering, fluid simulation, fracture simulation, modeling with implicit surfaces, and level set propagation. In most cases volumetric data is represented on spatially uniform, regular 3D grids, in part because such representations are simple and convenient. Moreover, most volumetric algorithms and numerical schemes have a strong preference for uniform sampling. Discretization of differential operators, interpolation, convolution kernels, and other such techniques can only be generalized to nonuniform grids with difficulty. This is particularly true for accurate finite-difference discretization of hyperbolic and parabolic partial differential equations governing time-dependent level sets and fluid dynamics. Such numerical simulations are commonly used in VFX where simplicity is a virtue, and they form the underlying motivation for most of the applications presented in this article. However, the fact remains that some volumetric applications, such as volumetric modeling and ray marching, benefit from sampling at varying resolution, so it is desirable to employ a data structure that supports both uniform and hierarchical sampling. Additionally, many volumetric algorithms like Computational Solid Geometry (CSG) and flood-filling benefit significantly from a hierarchical data representation.

Although dense regular grids are convenient for several reasons, they suffer at least one major shortcoming: their memory footprint grows in proportion to the volume of the embedding space. Even moderately sized dense, regular grids can impose memory bottlenecks if multiple instances are required or if the 3D data is animated and the grid domain dynamically changes, both of which are typical for simulations. Since most volumetric applications used in VFX production do not require data to be uniformly sampled everywhere in a dense grid, the solution is clearly to employ a sparse volumetric data structure. Such a sparse volume should ideally have the advantage that the memory footprint scales only with the number of voxels that contain meaningful sample values, and not with the volume of the dense embedding space. While numerous sparse 3D data structures have been proposed, most are designed explicitly for adaptive sampling, have slow or restrictive data access, do not scale to extreme resolution, or cannot easily handle numerical simulations with dynamic topology[1]. A few sparse data structures have been developed specifically for level sets and fluid simulation, but as will be explained later, they impose restrictions on the topology and access patterns of the data and as such cannot easily be generalized to other volumetric applications.

The VDB data structure is memory efficient, supports simulation of time-varying data, is capable of encoding arbitrary topology, and facilitates both uniform and adaptive sampling, while permitting fast and unrestricted data access. While we do not claim VDB to be a "silver bullet", we will show that it offers many advantages over existing dynamic data structures. In fact, VDB has already found use in several movie productions, for example, *Puss in Boots* [Miller et al. 2012] and *Rise of the Guardians*, where it was instrumental for the creation of high-resolution animated clouds, see Figure 1.

As will be discussed in more detail in the next section, the hierarchical structure of VDB may at first glance seem similar to other data structures, but these similarities are superficial for at least two reasons: either the existing data structures are not well-suited for

[1]Throughout this article "topology" is used to denote both the hierarchical layout of a data structure and the spatial layout of voxels.

Fig. 1. Top: Shot from the animated feature *Puss in Boots*, showing high-resolution animated clouds generated using VDB [Miller et al. 2012]. Left: The clouds are initially modelled as polygonal surfaces, then scan-converted into narrow-band level sets, after which procedural noise is applied to create the puffy volumetric look. Right: The final animated sparse volumes typically have bounding voxel resolutions of $15,000 \times 900 \times 500$ and are rendered using a proprietary renderer that exploits VDB's hierarchical tree structure. Images are courtesy of *DreamWorks Animation*.

time-varying simulations with dynamic topology, or they do not lend themselves to the implementation of the class of efficient algorithms that constitute an integral part of what we collectively denote VDB. In other words, VDB should be evaluated against the relevant target applications of dynamic sparse volumes and simulations.

1.1 Previous Work

Considering the importance of sparse data structures in computer graphics, it is no surprise that there is a large body of prior work, too large to be fully covered here. We limit our discussion to data structures for volumes without time-varying topology and those for compact representations of narrow-band level sets often intended for fluid dynamics and other types of simulations. While the latter is the main focus of this article, the former is still relevant, since VDB can also be used as a generic, compact spatial data structure.

As a prelude we elaborate on the distinction between these two categories. Data structures for dynamics must allow for both the grid *values* (e.g., simulation data) and *topology* (e.g., sparsity of values), to vary over time, and this imposes specific algorithmic challenges. For instance, to support simulations that employ numerical finite differencing and explicit time integration, fast sequential stencil iterators, uniform sampling, and temporal value buffers are needed. In contrast, static data structures are typically optimized for adaptive sampling and random access. Likewise, simulations often impose restrictions on the dynamic grid topology (e.g., that they are closed, manifold surfaces) and require efficient algorithms for topological operations like dilation, inside/outside flood-filling, and dynamic rebuilding of narrow-band level sets. In contrast, static data structures generally support arbitrary but static grid topology. As such, existing static data structures are not readily applicable to a large class of challenging numerical simulations, nor can current sparse dynamic data structures be used in volumetric applications that require efficient random access, nonrestricted topology, or adaptive sampling. In essence, this dichotomy has motivated the current work.

Octrees have an especially long history in the context of rendering, modeling, and mesh extraction. To mention just a few, Veenstra and Ahuja [1988] proposed a fast line-drawing algorithm for objects represented by octrees, Stolte and Kaufman [1998], and Frisken and Perry [2002] described several optimizations for tree traversals, Frisken et al. [2000] used octrees to encode adaptive distance fields for modeling, Ju et al. [2002] used octrees for adaptive meshing by means of dual contouring, and Ohtake et al. [2003] developed multilevel-partition-of-unity based on octrees. Sparse block-partitioning of grids, sometimes referred to as bricking or tiling, has also been successfully applied to rendering. Renderman uses a so-called brickmap [Christensen and Batali 2004] to store global illumination data, Lefebvre et al. [2005] employed an N^3-tree for interactive texturing on the GPU, and Crassin et al. [2009] used a similar tree for view-dependent out-of-core volume rendering. Of

these, the latter two seem to be most closely related to our work. However, we stress that to the best of our knowledge none of these data structures has successfully been applied to simulations or animated volumes. To quote from the conclusion of Crassin et al. [2009]: "Currently, animation is a big problem for volume data. In the future, we would like to investigate possible solutions."

The main application that initially drove the development of VDB was high-resolution sparse level sets [Osher and Fedkiw 2002]. These are time-dependent, narrow-band signed distance fields used to represent and deform complex implicit geometry, and they have found applications in fields like geometric modeling and free-surface fluid simulations. In recent years, a number of compact level set data structures have been proposed, including height-balanced octrees [Strain 1999; Min 2004; Losasso et al. 2004, 2005], one-level blocking or tiling [Bridson 2003; Lefohn et al. 2003], individual voxels stored in hash tables [Eyiyurekli and Breen 2011; Brun et al. 2012], Dynamic Tubular Grids (DT-Grid) [Nielsen and Museth 2006] and Hierarchical Run-Length Encoding (H-RLE) [Houston et al. 2006]. Of these, DT-Grid and H-RLE are most relevant to the current work given the fact that they solve a similar problem and have been shown to outperform the others [Nielsen 2006]. However, they take a very different (nonblocked and nonadaptive) approach, which imposes limitations not shared by VDB. Specifically DT-Grid employs a 3D analog of Compressed-Row-Storage (CRS), originally developed for compact representations of sparse matrices. H-RLE generalizes DT-Grid by replacing CRS with run-length encoding progressively applied in each of the three Cartesian directions.

In closing we note that the work detailed in this article marks the culmination of a long series of SIGGRAPH presentations on VFX applications of novel sparse data structures that eventually evolved into VDB. The related Dynamic Blocked Grid, DB-Grid, was introduced in Museth et al. [2007] and applied in Museth and Clive [2008] and Zafar et al. [2010]. Next, Museth [2009, 2011] described the development of the much improved hierarchical DB+Grid, so named because it conceptually combines DB-Grid with B+Trees. DB+Grid was finally renamed VDB[2], applied in production [Miller et al. 2012], and open sourced during SIGGRAPH 2012 [OpenVDB 2012], more than half a year after this article was first submitted. Thus, a few minor implementation details presented in this article do not apply directly to OpenVDB.

1.2 Contributions and characteristics of VDB

VDB is a volumetric data structure and algorithms with the following characteristics.

—*Dynamic*. Unlike most sparse volumetric data structures, VDB is developed for both dynamic topology and dynamic values typical of time-dependent numerical simulations and animated volumes. This requires efficient implementation of sparse finite-difference iterators, topological morphology, and rebuild algorithms, as well as temporal value buffers for cache-coherent numerical integration schemes.

—*Memory efficient*. The dynamic and hierarchical allocation of compact nodes leads to a memory-efficient sparse data structure that allows for extreme grid resolution. To further reduce the footprint on disk we present an efficient, topology-based compression technique that can be combined with bit-quantization and standard compression schemes.

—*General topology*. Unlike most existing dynamic data structures for narrow-band level sets, VDB can effectively represent sparse volume data with arbitrary dynamic topology. This implies that in the context of regular structured grids VDB can be used as a generic volumetric data structure, as opposed to merely supporting dynamic level set applications.

—*Fast random and sequential data access*. VDB supports fast constant-time random data lookup, insertion, and deletion. It also offers fast (constant-time) sequential stencil access iterators, which are essential for efficient simulations employing finite-difference schemes. Spatially coherent access patterns even have an amortized computational complexity that is independent of the depth of the underlying B+tree.

—*Virtually infinite*. VDB in concept models an unbounded grid in the sense that the accessible coordinate space is only limited by the bit-precision of the *signed* coordinates[3]. This support for unrestricted, and potentially negative, grid coordinates is nontrivial and especially desirable for grid applications with dynamic topology.

—*Efficient hierarchical algorithms*. Our B+tree structure offers the benefits of cache coherency, inherent bounding-volume acceleration, and fast per-branch (versus per-voxel) operations. Additionally it lends itself well to standard optimization techniques like blocking, SSE vectorization, and multithreading.

—*Adaptive resolution*. Unlike most existing narrow-band level set data structures, VDB is hierarchical and can store values at any level of the underlying tree structure. However, it is important to note that these multilevel values by design do not overlap in index space. Thus, VDB is different from multiresolution data structures like brickmaps or mipmaps that conceptually represent the same values in index space at multiple levels of detail.

—*Simplicity*. Compared to some existing sparse data structures VDB is relatively simple to both implement and apply. It is based on well-known concepts like blocks and trees, and it supports random access through an interface similar to regular dense volumes. To demonstrate this point we shall disclose most implementation details.

—*Configurable*. By design VDB is highly configurable in terms of tree depth, branching factors, and node sizes. This allows the grid to be tailored to specific applications in order to optimize factors like memory footprint, cache utilization, and expected access patterns.

—*Out-of-core*. VDB supports simple out-of-core streaming. More specifically, we can reduce the in-memory footprint by storing grid values out-of-core and only keeping the grid topology in memory. Values are then loaded on demand, for example, during block access while ray-tracing.

While many existing data structures certainly possess subsets of these characteristics, to the best of our knowledge, VDB is the first to embody all.

2. THE VDB DATA STRUCTURE

As can be surmised from the preceding list, VDB comprises both a compact dynamic data structure and several accompanying

[2]Note, this name is completely unrelated to the similarly-named unstructured mesh data structure presented in Williams [1992].

[3]Clearly the accessible coordinate space of *any* computer implementation of a discrete grid is limited by the bit-precision of the voxels' coordinates. However, most implementations are unable to utilize the full coordinate range due to their far more restrictive memory overhead. Notable exceptions are VDB and DT-Grid.

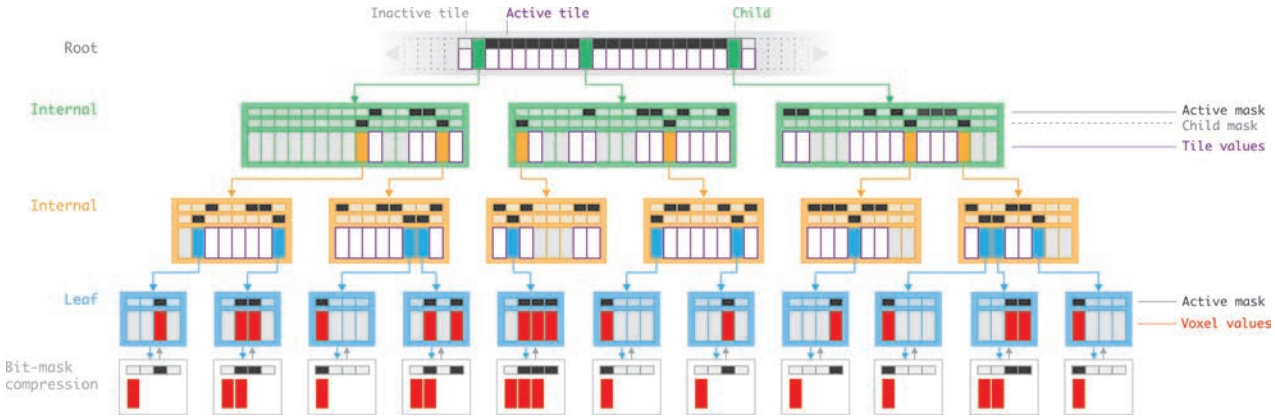

Fig. 2. 1D Illustration of a VDB with one `RootNode` (gray), two levels of `InternalNodes` (green and orange), and `LeafNodes` (blue). The `RootNode` is sparse and resizable, whereas the other nodes are dense and have decreasing branching factors restricted to powers of two. The large horizontal arrays in each node represent respectively the hash-map table `mRootMap`, and the direct access tables `mInternalDAT` and `mLeafDAT`. These tables encode pointers to child nodes (green, orange, or blue), upper-level tile values (white/gray), or voxel values (red/gray). Gray is used to distinguish inactive values from active values (white and red). The white nodes at the bottom show compressed `LeafNodes` where the inactive voxels (gray) have been removed, leaving only active voxels (red). The small arrays above the direct access tables illustrate the compact bit masks, for example, `mChildMask` and `mValueMask`, that compactly encode local topology.

algorithms. While they are clearly interdependent, we shall introduce them separately.

2.1 Concepts and Analogies

One of the core ideas behind VDB is to dynamically arrange blocks (or tiles) of a grid in a hierarchical data structure resembling a B+tree. Blocks are leaf nodes at the same fixed depth of an acyclic, connected graph with large but variable branching factors that are restricted to powers of two; see Figure 2 and Figure 3 for 1D and 2D illustrations. This implies that the tree is height-balanced by construction, but shallow and wide. This effectively increases the domain while reducing the depth of the tree and consequently the number of I/O operations required to traverse the tree from root node to leaf node. In contrast, octrees are typically tall, due to the small branching factor of two in each of the spatial dimensions. However, similar to octrees or N-trees, VDB can also encode values in the non-leaf nodes, serving as an adaptive multilevel grid. As such it might be tempting to characterize VDB as merely a generalized octree or N-tree. However, such a comparison is superficial, since the real value of VDB is its unique implementation, described in Section 3, which facilitates efficient algorithms for data access and manipulation of dynamic data.

As hinted earlier, a better analogy to the VDB data structure is the B+tree [Bayer and McCreight 1972] often used in file systems and relational databases, for example, NTFS and Oracle. Our variant of the B+tree retains the favorable performance characteristics that result from the use of large, variable branching factors, but the volumetric application allowed us to exploit representational and algorithmic optimizations. Whereas we encode grid values indexed by their spatial coordinate in all nodes of the tree, a standard B+tree encodes abstract records indexed by keys in a block-oriented storage context at the leaf nodes only. In other words, VDB is a mutlilevel data structure and does not employ keys in the sense of a traditional B+tree. Another distinction is that the leaves of a B+tree are often linked to one another to allow for rapid sequential iteration. As will be explained in Section 3, VDB adopts a more efficient strategy that avoids maintenance of this linked list when nodes are dynamically inserted or deleted. Finally, whereas B+trees employ random lookup with logarithmic complexity, VDB offers constant-time random access. Nevertheless, the VDB data structure may be viewed as a particular optimization of B+trees, and we believe it is the first in the context of sparse volumetric grids.

Another interesting comparison is to the design of memory hierarchies in modern CPUs. Like VDB, they employ a fixed number of cache levels of decreasing size and increasing random-access performance, and only the topmost level, the main memory, can be dynamically resized. We shall elaborate more on this analogy in Section 3.2.

A data structure or algorithm is only as efficient and useful as its software implementation permits. VDB is clearly no exception, and many of its attractive features can exactly be ascribed to such efficient implementations in terms of both memory footprint and computational performance. Consequently we have chosen to include several details of our C++ implementation in the discussions that follow. This should not only allow the reader to reproduce VDB, but also serves to demonstrate how simple it is, considering the complexity of the problems it addresses. We make extensive use of C++ template metaprogramming and inlining, and consistently avoid virtual functions. This turns out to be an important optimization that, among other things, leads to computationally efficient random access. More specifically, the tree node classes are recursively templated on their child node types, rather than inheriting from a common base class. The reason is that templated node configurations are expanded inline at compile time, and consequently some template functions have no extraneous runtime overhead. Another advantage of this approach is that it allows us to easily customize the tree depth and branching factors, but at compile time rather than runtime. Other noteworthy implementation techniques are fast bit-wise operations, compact bit-wise storage, type unions for memory reuse, and straightforward implementation of threading and SIMD operations. The fact that VDB lends itself to these techniques is an important practical consideration. In what follows, we concentrate on the implementation of 3D grids, though for the sake of clarity most of our illustrations are 1D or 2D, and VDB can arguably be implemented for any spatial dimensions.

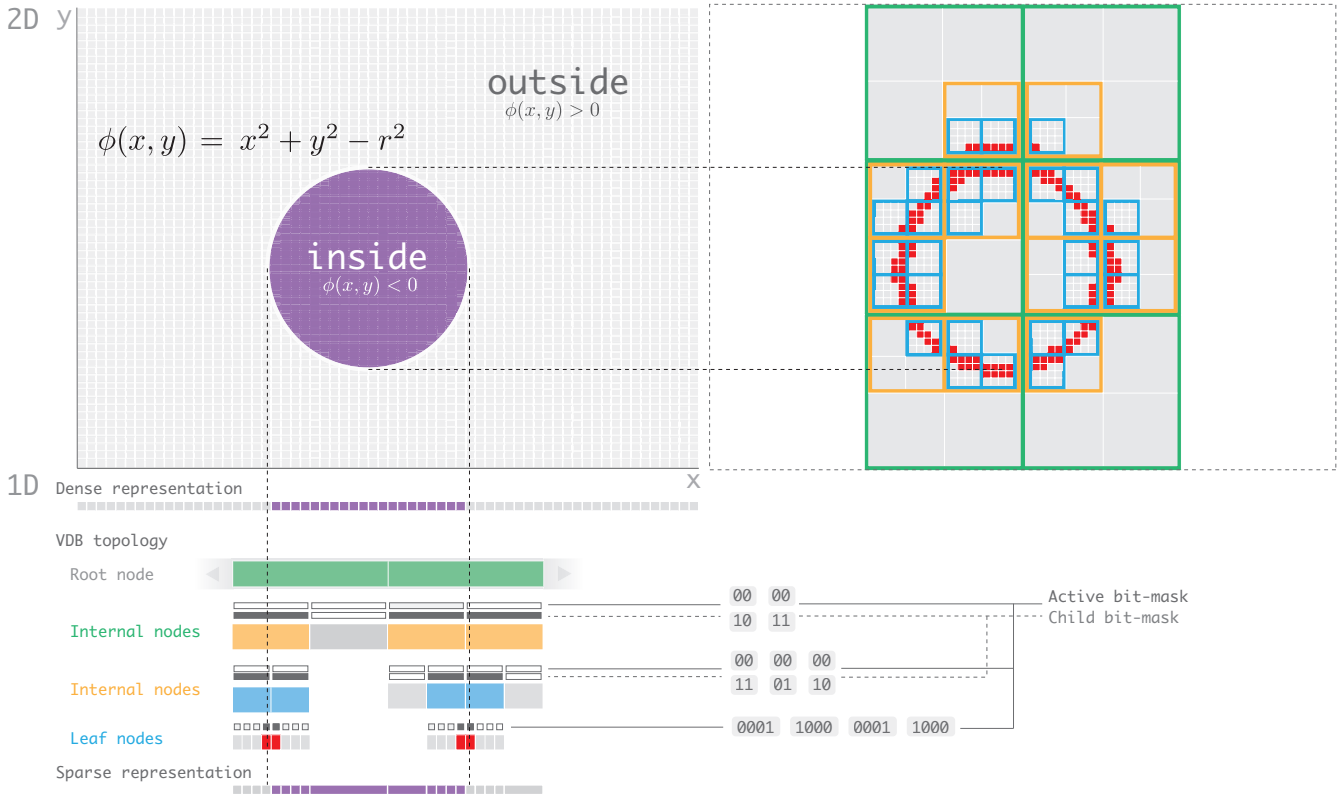

Fig. 3. Illustration of a narrow-band level set of a circle represented in, respectively, a 1D and 2D VDB. Top Left: The implicit signed distance, that is, level set, of a circle is discretized on a uniform dense grid. Bottom: Tree structure of a 1D VDB representing a single y-row of the narrow-band level set. Top Right: Illustration of the adaptive grid corresponding to a VDB representation of the 2D narrow-band level set. The tree structure of the 2D VDB is too big to be shown. Voxels correspond to the smallest squares, and tiles to the larger squares. The small branching factors at each level of the tree are chosen to avoid visual cluttering; in practice they are typically much larger.

2.2 Terminology

VDB in concept models an infinitely large 3D index space (x, y, z), although in practice it is naturally limited by the bit-precision of indices and the available memory. The data encoded into VDB consist of a `Value` type, defined by means of templating, and the corresponding discrete indices, (x, y, z), specifying its spatial sample location, that is, *topology* of the *value* within the tree structure. For convenience we shall occasionally use the symbol w to collectively denote one of the three Cartesian coordinate directions. We refer to the smallest volume elements of index space as *voxels*, shaded red in Figure 2. A single data *value* is associated with each voxel. Every such voxel in VDB can exist in one of two discrete states, namely *active* or *inactive*. The interpretation of this binary state is application specific, but typically an active voxel is more "important" or "interesting" than an inactive one. For example, in scalar density grids, inactive voxels have a default background value (e.g., zero) and active voxels have a value different from this default value. For narrow-band level sets, all voxels inside the narrow band are active, and all other voxels are inactive and have a constant nonzero distance whose sign indicates inside versus outside topology, as in Figure 3. VDB separately encodes voxel *topology* in a tree whose root node covers all of index space and whose leaf nodes each cover a fixed subset of index space. More precisely, topology is implicitly encoded into bit masks, and values are explicitly stored in buffers residing at any level of the tree. Areas of index space in which all voxels have the same value can be represented with a single value stored at the appropriate level of the tree, as in Figure 3. We shall adopt the term *tile value* to denote these upper-level values. Like voxels, tiles can be either active or inactive. The overall goal of VDB is to consume only as much memory as is required to represent active voxels, while maintaining the flexibility and performance characteristics of a typical dense volumetric data structure.

In the C++ code snippets that follow we use the following conventions: Identifiers with leading caps, for example, `LeafNode`, are either class, type, or template parameter names; those in camel case with a leading "s", for example, `sSize`, denote constant static data members, and those in camel case with a leading "m", for example, `mFlags`, denote regular member variables. All lower case identifiers, for example, `x`, denote nonmember variables.

2.3 Building Blocks

While VDB can be configured in many different ways, we shall describe the components that are common to all configurations, starting from the leaf node and ending with the root node, following a discussion of a data structure fundamental to all nodes.

Direct access bit masks. A fundamental component of VDB is the bit masks embedded in the various nodes of the tree structure.

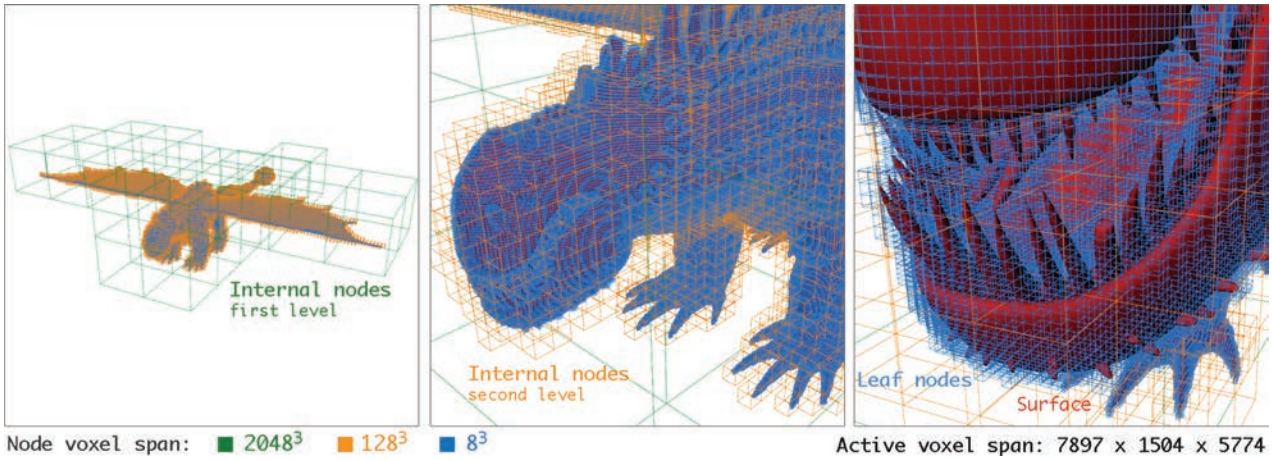

Fig. 4. High-resolution VDB created by converting polygonal model from *How To Train Your Dragon* to a narrow-band level set. The bounding resolution of the 228 million active voxels is 7897 × 1504 × 5774 and the memory footprint of the VDB is 1GB, versus the $\frac{1}{4}$ TB for a corresponding dense volume. This VDB is configured with `LeafNodes` (blue) of size 8^3 and two levels of `InternalNodes` (green/orange) of size 16^3. The index extents of the various nodes are shown as colored wireframes, and a polygonal mesh representation of the zero level set is shaded red. Images are courtesy of *DreamWorks Animation*.

They provide fast and compact direct access to a binary representation of the *topology* local to the node, and we are, to the best of our knowledge, the first to employ them simultaneously for: (1) hierarchical topology encoding, (2) fast sequential iterators, (3) lossless compression, (4) boolean operations, and (5) efficient implementations of algorithms like topology dilation. All of these operations are essential for dynamic sparse data structures applied to fluids, narrow-band level sets, and volumetric modeling. We will discuss these four applications and the explicit deployment of bit masks in more detail later.

Leaf nodes. These nodes (shaded blue in Figures 2, 3, and 4) act as the lowest-level blocks of the grid and by construction all reside at the same tree depth. They effectively tile index space into nonoverlapping subdomains with $2^{\text{Log2}w}$ voxels along each coordinate axis, where $\text{Log2}w = 1, 2, 3, \ldots$. A typical configuration would be $\text{Log2}w = 3$, corresponding to an $8 \times 8 \times 8$ block. We restrict the leaf (and internal) node dimensions to powers of two, as this allows for fast bit operations during tree traversals.

```
template<class Value, int Log2X,
         int Log2Y=Log2X, int Log2Z=Log2Y>
class LeafNode {
  static const int sSize=1<<Log2X+Log2Y+Log2Z,
    sLog2X=Log2X, sLog2Y=Log2Y, sLog2Z=Log2Z;
  union LeafData {
    streamoff     offset;//out-of-core streaming
    Value*        values;//temporal buffers
  }               mLeafDAT;//direct access table
  BitMask<sSize> mValueMask;//active states
 [BitMask<sSize> mInsideMask];//optional for LS
  uint64_t       mFlags;//64 bit flags
};
```

As can be seen in the preceding code, the `LeafNode` dimensions are fixed at compile time, and the size of the node is readily computed as $1 \ll \sum_w \text{sLog2}w$, where $\ll$ denotes a bit-wise left shift. The leaf nodes encode voxel data values into a Direct Access Table[4], `mLeafDAT`, and the active voxel topology into the direct access bit mask `mValueMask`. It is important to note that whereas the bit mask has a fixed size equal to that of the `LeafNode`, the size of the value array, `mLeafDAT.values`, is dynamic for the following reasons.

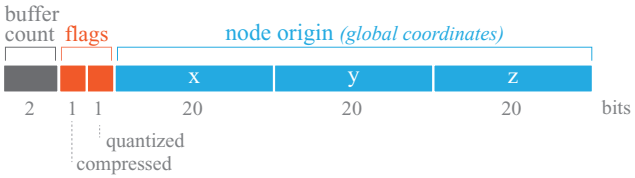

Fig. 5. Compact 64-bit representation of `mFlags` encoding; buffer count (2), compression (1), quantization (1), and leaf node origin (3 × 20).

First, for some applications the memory footprint of the `mLeafDAT`s can be significantly reduced by means of various compression techniques. We support several different codecs including ones that take advantage of the topology of active voxels, that is, `mValueMask`, and the inside/outside topology of level sets, that is, `mInsideMask`, as well as more traditional entropy-based and bit-truncation schemes. The details can be found in Appendix A. Second, to best facilitate numerical time integration, the `LeafNode` can contain multiple value buffers, each storing voxels for a different time level. For example, a third-order accurate TVD-Runge-Kutta scheme [Shu and Osher 1988] requires three temporal buffers. Finally, the voxel values can also reside out-of-core, which is facilitated by the `offset` into a file stream. Note that this offset does not incur additional memory overhead since it is encoded in a C++ union in the `mLeafDAT`.

The variable number and size of value buffers as well as other information about the `LeafNode` is compactly encoded in the 64-bit variable `mFlags`; see Figure 5. The first two bits encode the four states: 0 buffers, that is, values are out-of-core, 1 buffer, that is, in-core values with no support for temporal integration, 2 buffers, that is, in-core values with support for first-and second-order temporal integration or 3 buffers, that is, in-core values with support for third-order temporal integration. The third bit is on if the block is compressed, and the fourth bit is on if the leaf is bit-quantized. Finally, the remaining $3x20$ bits of `mFlags` are used to encode the global

[4]Throughout this article we use the term "direct access table" to denote an array of elements that has a worst-case random access complexity of $O(1)$.

origin of the node in the virtual grid[5]. This turns out to be very convenient since global voxel coordinates can then be derived by combining the local voxel topology encoded in `mValueMask` with the global node origin from `mFlags`. Consequently, `LeafNodes` are self-contained and need not reference their parent nodes, which reduces memory footprints and simplifies derivation of voxel coordinates.

Internal nodes. As the name suggests, these nodes reside at all intermediate levels of the tree between the root and leaf nodes, and essentially define the depth and shape of the B+tree. This is illustrated by the green and orange nodes in Figures 2, 3, and 4.

```
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
};
```

As can be seen, they share several implementation details with the `LeafNodes`. The branching factors are configurable by means of the template parameters, Log2w, restricting branching to powers of two which facilitates efficient tree traversals. However, unlike `LeafNodes`, `InternalNodes` encode both values and tree topology, that is, pointers to other internal or leaf nodes. This is efficiently implemented by a union structure in the direct access table `mInternalDAT`. The corresponding topology is compactly encoded in the bit mask `mChildMask`, and `mValueMask` is used to indicate if a tile value is active or not. Note that since the branching factors, Log2w, are fixed at compile time, so are the sizes of `mInternalDAT`, `mChildMask`, and `mValueMask`. However, it is important to stress that internal nodes at different tree levels are allowed to have different branching factors, thereby adding flexibility to the overall shape of the tree; see Figure 2. This is nontrivial since the memory footprint and computational performance can be affected by the tree configurations, as will be discussed in Section 5.1.

Root node. This is the topmost node at which operations on the tree commonly begin. We note that the `InternalNode` can also serve as a root node, however, with significant limitations. Since all nodes introduced thus far have templated branching factors, the overall domain of the corresponding grid is effectively fixed at compile time. For extreme resolutions this will require large branching factors, which in turn can incur memory overheads due to the dense direct access tables, `mInternalDAT` (more on this in Section 2.4). Hence, to achieve conceptually unbounded grid domains, with small memory footprints, a sparse dynamic root node is required. Such a `RootNode` is illustrated in 1D in Figure 2, gray node, and the structure is as follows.

```
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
};
```

All configurations of a VDB have at most one `RootNode` which, unlike all the other tree nodes, is sparse and can be dynamically resized. This is facilitated by a hash-map table that encodes child pointers or tile values. If a table entry represents a tile value (i.e., `child=NULL`), a boolean indicates the state of the tile (i.e., active or inactive). It is important to note that by design `mRootMap` typically contains very few entries due to the huge index domains represented by the tile or child nodes, for example, 4096^3. In fact in Section 5 we will show that a red-black tree data structure like `std::map` can perform better than a sophisticated hash map that requires the computation of more expensive hash keys. Regardless, random access into a dynamic sparse data structure like a map or hash table is generally slower than a lookup into the fixed dense direct access table `mInternalDAT`. In other words it would seem the proposed `RootNode` suffers a major disadvantage when compared to the fast `InternalNode`. However, there is a surprisingly simple and very efficient solution to this problem of slow access into the `mRootMap`. The `RootNode` contains a registry of `Accessors`, which can significantly improve spatially coherent grid access by reusing cached node pointers from a previous access to perform bottom-up, versus top-down, tree traversals. This inverted tree traversal effectively amortizes the cost of full traversals initiated at the `RootNode` level. While we will leave the implementation details for Section 3.2, it suffices to stress that a majority of practical access patterns, even seemingly random ones, typically exhibit some type of spatial coherence. Finally, `mBackground` is the value that is returned when accessing any location in space that does not resolve to either a tile or a voxel within a child node.

2.4 Putting it All Together

No single configuration of any spatial data structure can claim to handle all applications equally well, and VDB is no exception, so it is deliberately designed for customization. Different combinations of the nodes and their parameters can alter the tree depth and branching factors which in turn impacts characteristics like available grid resolution, adaptivity, access performance, memory footprint, and even hardware efficiency. We will discuss some general guidelines for such configurations, and then present a configuration that balances most of the aforementioned factors, and that has proven useful for all of the applications presented in Section 5. As a prelude, let us consider some extreme configurations that will serve to motivate the balanced configuration of VDB, namely tiled grids, hash maps, and N-trees.

Tiled grids. Suppose we decided to configure the grid as simply one `InternalNode` connected directly to multiple `LeafNodes`. This is conceptually the idea behind most existing tiled grids, like the proprietary DB-Grid [Museth et al. 2007; Museth and Clive 2008] and the open-source SparseField [Field3D 2009], except they typically

[5]To best utilize the 20 bits the origin is divided by the `LeafNode` size, that is, `X>>=Log2X` where `X` denotes the origin of the node along the x-axis and `>>` is a bit-wise right shift. If the node is 8^3, that is, `Log2X=3`, this allows for grid domains exceeding eight million grid points in each coordinate direction, which has proven more than enough for all our applications. However, in OpenVDB this restriction is overcome by encoding the origin at full 32 bit-precision.

employ a nonstatic direct access table at the top-level. Since the path from root to leaf is extremely short, and all nodes employ fast direct access tables, we expect efficient random access performance. However, this fast access comes at the cost of limited grid resolutions due to significant memory overheads. This follows from the fact that the available resolution of the grid is given by the product of the branching factors at each level of the tree. Hence, if high grid resolution is desired the branching factors of the `InternalNode` (currently the top node) and the multiple `LeafNodes` need to be large. Conversely, it is generally desirable to keep the size of the `LeafNodes` relatively small for two reasons: cache performance and memory overhead from partially filled `LeafNodes` with few active voxels. This typically implies that the branching factors of the top node need to be several orders of magnitude larger than those of the `LeafNodes`, which in turn introduces a significant memory overhead from the top-level dense direct access table; see Section 5. In order to achieve a modest grid resolution of 8192^3 with a `LeafNode` size of 8^3 the footprint of the top `mInternalDAT` is a massive 8GB, even before voxel values are inserted. Finally, since this particular configuration essentially corresponds to a flat tree, it has no support for efficiently storing contiguous regions of index space with constant (upper-level) tile values. Equally important, it lacks an acceleration structure, which is important for hierarchical algorithms.

Hash maps. At this point the attentive reader might speculate that an efficient solution to the aforementioned memory issue would simply be to replace the dense `InternalNode` with the sparse `RootNode` that employs a hash-map table. However, careful consideration reveals several performance issues, starting with the fact that the `Accessors` registered in the `RootNode` aren't very efficient with this simple configuration. To understand why let's recap the idea behind the `Accessor`; to amortize the overhead of slow lookup into the `mRootTable` by means of reusing cached child nodes. Now, as emphasized before it is typically advantageous to use small `LeafNodes` when representing sparse volumes. Consequently the child nodes of the `RootNode` would cover very small index domains, which in turn implies cache misses in the `Accessor`. In other words, the complexity of random access for this configuration is dominated by the bottleneck of the slower `mRootMap`. Furthermore, since the `LeafNodes` are small the `mRootMap` will typically contain many entries which can further impair lookup performance due to collisions of hash keys and logarithmic search complexity. For narrow-band level sets this situation is even worse since the `mRootMap` also has to contain all the tile values that represent the constant interior of the interface; see large green tiles in Figure 3. While good hash functions can minimize collisions they tend to be more computationally expensive than the (perfect) keys into a direct access table, and the resulting hash key will arrange the inserted data randomly. Consequently even spatially coherent access to `LeafNodes` can lead to random memory access which, generally, leads to poor cache performance for simulations. Finally, the proposed tree configuration is clearly flat and therefore lacks support for fast hierarchical algorithms as well as adaptive sampling.

For the sake of completeness we note that one could also consider a non-hierarchical configuration where the `RootNode` stores individual active voxels directly in a single hash table. This is conceptually the approach taken by Eyiyurekli and Breen [2011], and Brun et al. [2012]. While offering a simple and sparse data structure it suffers several performance issues. The first problem is of course that random access to voxels always requires lookup into a hash table, which is slower than lookup into a direct access table. Even worse, since there are typically millions of active voxels, hash-key collisions, which lead to slow access, are difficult, if not impossible, to avoid. Finally, as stressed earlier good hash functions introduce randomness which impairs cache performance, even during sequential access.

N-trees. Now consider another extreme case with a fixed branching factor at all levels, for instance `Log2X=Log2Y=Log2Z=1`, corresponding to a height-balanced octree. With an extremely small branching factor of two in each coordinate direction, this is clearly optimal for adaptive grid sampling. However, it suffers slow random access; see Section 5. This is a consequence of the fact that for a given tree depth D, the corresponding resolution is only 2^D. For a modest grid resolution of 8192^3, the octree would have to be at least 13 levels deep. This situation can be improved by increasing the branching factor, so as to resemble an N-tree, originally proposed for static volumes [Lefebvre et al. 2005; Crassin et al. 2009]. However, as mentioned before, large nodes can impair cache performance and inflate the memory footprint. This is especially true for large `LeafNodes`, which can result in an unfavorable ratio of active to inactive voxels when encoding sparse data; see Section 5. Additionally, a large constant branching factor can lead to poor grid adaptivity since the change of resolution between consecutive levels of the tree is too drastic. Finally, since the depth of VDB is fixed by design, a fixed branching factor would result in a fixed grid resolution, which is generally undesirable when dealing with dynamic volumes.

We summarize observations to consider when configuring a VDB:

—*Resolution* scales with branching factors and trees depth;

—*Random access* is generally faster for shorter trees;

—*Adaptivity* favors small branching factors and tall trees;

—*Memory* scales with the number and size of nodes;

—*Cache reuse* is improved with smaller nodes and deep trees.

Given these seemingly conflicting guidelines it would seem that optimal performance can be achieved only if the tree configuration is customized for every grid application. However, we have identified a class of trees that balance most of these factors, at least for the level set and sparse volume applications presented in this article. This configuration can be regarded as a hybrid between the best features of the extreme cases discussed before: a flat tiled grid, a hash map, and a regular N-tree structure. It is a short and wide height-balanced B+tree with small leaf nodes, typically three or four levels deep and with increasing branching factors from the bottom up. An example is a tree with a dynamic `RootNode`, followed by two levels of `InternalNodes` with static branchings of respectively 32^3 and 16^3, followed by `LeafNodes` of size 8^3, that is, each child node of the `RootNode` spans an index domain of $4,096^3$. Figure 2 shows a 1D example of a three-level VDB with a similar profile. Alternatively the `RootNode` can be replaced by an `InternalNode` with a branching factor of 64^3, which results in a VDB with a fixed available resolution of $262,144^3$, but a worst-case constant-time random access. We shall study the performance of these configurations, and many others, in Section 5, and will demonstrate that they offer fast random access and efficient sequential access, support extreme resolutions, are hierarchical and adaptive, and have a relatively small memory overhead and good cache performance.

A curious but useful variant is a VDB that only encodes grid topology, by means of the bit masks, but stores no data values. This VDB is very compact and can be used for topology algorithms on the voxels of a regular VDB, for example, dilation, erosion, or narrow-band rebuild, or for applications that only require binary representations of index space, that is, voxelized masks.

We have also experimented with trees that can dynamically grow vertically, as opposed to horizontally, but with disappointing results. We found that such configurations complicate efficient implementations of data access patterns like insertion and deletion, and prevent us from optimizing data access when the depth and branching factors are known at compile time. Finally, a fixed depth leads to constant-time random access. These considerations should become clearer when we discuss implementation details in the following section.

3. VDB ACCESS ALGORITHMS

So far we have focused exclusively on the VDB data structure, which constitutes only half of our contribution, and arguably the simpler half. The other half concerns a toolbox of efficient algorithms and optimization tricks to navigate and operate on this dynamic data structure. We shall focus on tree access algorithms in this section and discuss more application-specific techniques in the next section. In other words, it is the combination of the data structure presented in Section 2 and the algorithms to be discussed in Section 3 and Section 4 that we collectively refer to as VDB.

There are three general access patterns for a spatial data structure: random, sequential, and stencil. We will next describe how each of these is efficciently implemented for VDB. Subsequently, we will present various other algorithms that are important for dynamic volumes such as level sets.

3.1 Random Access

The most fundamental, but also the most challenging, pattern is random access to arbitrary voxels. What distinguishes this pattern is the fact that, in the worst case, each voxel access requires a complete top-down traversal of the tree, starting from the `RootNode` and possibly terminating at a `LeafNode`. In practice, though, random access can be significantly improved by inverting the traversal order, a topic we shall postpone until Section 3.2. As such it is easier to use a contrapositive definition of random access, namely to say it is any access that is neither sequential nor stencil based. The latter two are easily defined as access with a fixed pattern, defined either from the underlying data layout in memory or some predefined neighborhood stencil.

Random lookup is the most common type of random access and shall serve as a prelude to all the others. We start by identifying the fundamental operations required to effectively traverse the tree structure starting from the `RootNode`. Obviously these operations depend on the actual implementation of the `mRootMap`, so let's start with the simple `std::map`. To access the voxel at index coordinates `(x,y,z)`, we begin by computing the following signed `rootKey`

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

where `&` and `~` denote, respectively, bit-wise AND and NOT operations. At compile time this reduces to just three hardware AND instructions, which mask out the lower bits corresponding to the index space of the associated child nodes. The resulting values are the coordinates of the origin of the child node that contains `(x,y,z)`, and hash collisions are thus avoided. This perfect key is then used to perform a lookup into `mRootMap` which stores the `RootData` (line 31 of the code) in lexicographic order of the three `rootKey` components. If no entry is found, the background value (`mBackground`) is returned, and if a tile value is found, that upper-level value is returned. In either case the traversal terminates. However, if a child node is found the traversal continues until a tile value is encountered or a `LeafNode` is reached. The only modification needed to replace the `std::map` with a hash map, like the fast `google::dense_hash_map` [sparsehash 2009], is a good hash function that generates uniformly distributed random numbers. To this end we combine the perfect `rootKey` given earlier with the (imperfect) hash function proposed in Teschner et al. [2003].

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

Here the three constants are large prime numbers, `^` is the binary XOR operator, `&` is the bit-wise AND operator, `<<` is the bit-wise left shift operator, and $Log2N$ is a static constant estimating the base two logarithm of the size of `mRootMap`. Note that we have improved the original hash function in Teschner et al. [2003] by replacing an expensive modulus operation with a faster bit-wise AND operation. We have observed three important benefits from this seemingly small change: the computational performance is improved by more than $2\times$, the hash function works correctly with negative coordinates, and it is less prone to 32-bit integer overflow. Using the standard Pearson's chi-squared test we have verified that the modified hash function preserves its uniformity over signed coordinate domains. While it is well-known that hash maps asymptotically have better time complexity than a simple `std::map` it should be clear that the `rootHash` is more expensive to compute than the corresponding `rootKey`. Combine this with the fact that in practice the `mRootMap` is expected to be very small, and it is not obvious which implementation is faster, a question we shall address in Section 5.

When an `InternalNode` is encountered (either at the top or internal levels), the following direct access offset is derived from the global grid coordinates

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

where `Log2YZ=Log2Y+Log2Z`. At compile time this reduces to only three bit-wise AND operations, five bit shifts, and two additions. Next, if bit `internalOffset` of `mChildMask` is off, then `(x,y,z)` lies within a constant tile, so the tile value is returned from `mInternalDAT` and traversal terminates. Otherwise, the child node is extracted from `mInternalDAT` and traversal continues until either a zero bit is encountered in the `mChildMask` of an `InternalNode` or a `LeafNode` is reached. The corresponding direct access offset for a `LeafNode` is even faster to compute.

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

This is because it reduces to just three bit-wise AND operations, two bit-wise left shifts, and two additions.

Let us make a few important observations. First, with the exception of the three multiplications in the hash function, all of these computations are performed with single-instruction bit operations, as opposed to much slower arithmetic operations like division, multiplication, and modulus—a simple consequence of the fact that all branching factors by design are powers of two. Second, all keys and offsets are computed from the same global grid coordinates

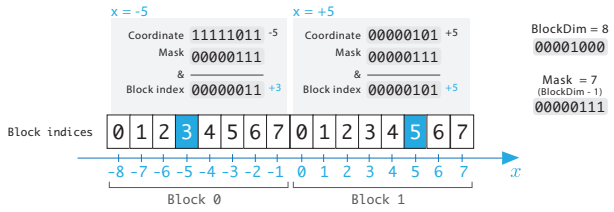

Fig. 6. Illustration of the bit masking in lines 46–52 of the code. The example on the left (Block 0) shows how the bit operations work for negative indices, by virtue of the two's complement of binary numbers. In this number system the bit representation of a negative value (e.g., −5) is derived by flipping the bits of the corresponding absolute value (i.e., 5) and adding one. Thus, after bit masking the indices −5 and 5, the unsigned offsets into block 0 and 1 are respectively 3 and 5.

`(x,y,z)`, that is, they are level independent and nonrecursive. This is a consequence of the fact that tree depth and branching factors are known at compile time. Finally, all these bit-wise computations work even for negative values of `(x,y,z)`. This follows from the facts that bit-wise AND, unlike bit-wise shifts, is well-defined for signed integers, and from the two's complement representation of signed integers in modern hardware (see Figure 6). Inspecting lines 39–52 in the code, it is evident that only bit-wise AND is applied directly on `(x,y,z)`, and that all other bit-wise shift operations involve unsigned integers.

Since VDB by construction is height-balanced with a runtime fixed depth, every path from the `RootNode` to a `LeafNode` is equally long and we conclude that every random lookup of leaf values involves the same worst-case time complexity. Random access to tile values stored at shallower depths of the tree is obviously faster since the traversal terminates early, but it is still bounded by the computational complexity of access to voxel values stored in leaf nodes. Given that both `InternalNodes` and `LeafNodes` employ direct access tables, with $O(1)$ worst-case random access time, it should be clear that the overall complexity of VDB is given by that of the `RootNode`. If an `InternalNode` is used at the top level the worst-case random lookup complexity of VDB is $O(1)$. If, on the other hand, the `RootNode` is implemented with a `std::map` the complexity is $\log(n)$, where n is the number of entries in `mRootMap`, which as mentioned earlier is expected to be very low due to the size of the child nodes. However, if `RootNode` uses an optimized hash map with a good hash function the average complexity becomes $O(1)$, whereas the worst-case time is $O(n)$. Only perfect hashing will allow for constant-time lookup in the worst case, which is not theoretically possible for a truly unbounded domain, that is, infinite number of hash keys. However, in practice we expect few entries with sufficiently uniform hash keys, and as such we conclude that even with a `RootNode` that supports unbounded domains, random lookup of VDB has on *average* constant-time complexity.

Random insert is typically used when initializing grids, but can also play an essential role in the context of dynamic data, like rebuilding a narrow band of a level set. Traversal is performed using the fast bit operations (lines 39–52 in the code), but now a child node is allocated if the corresponding bit is off in a `mChildMask`. Traversal terminates at a (possibly newly constructed) `LeafNode`, with the voxel value set in the desired temporal buffer and the corresponding bit set in `mValueMask`. Because nodes below the `RootNode` are allocated only on insert, the memory footprint for sparse volumetric data is low. Although this dynamic allocation of nodes implies that random insert can be slower than random lookup, the overhead is typically amortized over multiple coherent insert operations, and on average it has constant-time complexity.

Random deletion is another example of an operation that requires efficiency when dealing with dynamic data. The traversal is implemented similarly to random insert, except now bits are unset in `mChildMask` and `mValueMask` and nodes are pruned away if they contain no child nodes or active voxels. Pruning, which is performed bottom-up, can be implemented efficiently by simultaneously checking multiple bits in `mChildMask` and `mValueMask`, for example, simultaneously checking 64 entries by means of 64-bit arithmetic.

In conclusion, VDB supports constant-time random access operations like lookup, insertion, and deletion, and this is on average independent of the topology or resolution of the underlying dataset. In comparison, neither DT-Grid nor H-RLE support any type of random insertion or deletion, and random lookup has logarithmic time complexity in the local topology of the dataset.

3.2 Improving Random Access

In practice, virtually all grid operations have some degree of spatial coherence, so grids are rarely accessed in a truly random pattern. In fact, regardless of the underlying data structure, truly uniform random access should always be avoided, since it represents the worst-case scenario in terms of memory reuse. For truly random access the computational bottleneck is typically the memory subsystem of the CPU, rather than the data structure itself. The extreme cases of spatial coherence are stencil or sequential access which are studied separately shortly. However, even for nonsequential and non-stencil access, there is very often some exploitable spatial coherence.

The core idea is to improve random access by virtue of inverted tree traversal, which in turn is facilitated by retracing cached access patterns. Rather than initialize each random access from the `RootNode`, we cache the sequence of nodes visited during a previous access operation, allowing subsequent access operations to traverse the tree from the bottom up until a common parent node is reached. On average, the greater the spatial clustering of random accesses, the shorter the traversal path and hence the greater the speedup. In particular, the slower access time of the `RootNode` can be amortized if its child nodes are large enough to introduce average spatial coherence for all practical access patterns.

This technique essentially resembles the caching mechanism in memory hierarchies of modern CPUs that employ multiple cache levels. In this analogy the `RootNode` corresponds to main memory, `InternalNodes` to L3/L2 cache, and `LeafNodes` to L1 cache. Just as access to data in L1 cache is much faster than any of the other memory levels, access to values in cached `LeafNodes` is faster than in any other tree level. Further, even the relative sizes of the cache levels in current CPUs resemble the node sizes in the balanced VDB tree structure proposed in Section 2.4. The depth of the memory hierarchy in CPUs is fixed, typically consisting of a large, relatively slow, and dynamically resizable portion of main memory (RAM) and three levels of progressively smaller, faster, and fixed sized cache levels, exactly mimicking the configuration of VDB illustrated in Figure 2. This complexity of the CPU's physical memory hierarchy is hidden to the operating system by employing a nonhierarchical Virtual Address Space (VAS). To accelerate the mapping from VAS to pages of physical memory in the various data caches, the CPU employs a Translation Look-aside Buffer (TLB) that caches previous lookups from virtual to physical memory. To complete this CPU metaphor for VDB we can think of grid coordinates `(x,y,z)` as a virtual memory address, and pointers to tree nodes as addresses of physical memory pages containing `(x,y,z)`.

Thus, to accelerate random access into a VDB we are basically seeking the analogy of a TLB for our tree structure.

While caching as outlined earlier might seem straightforward, it is actually not trivial to implement efficiently. The problem is that the random access operations detailed in Section 3.1 are already fast, so there is little room for computational overhead. This precludes the use of a standard hash table to store the cached tree nodes. Instead, we propose the following algorithm which is optimized for our specific tree configuration.

First imagine we buffer all the nodes visited in a previous root to leaf traversal in a small cache list whose size is equal to the depth of the tree, for example, 3, and whose ordering is leaf to root. In other words a forward traversal of this list effectively corresponds to an *inverse tree traversal*. Next each entry in this cache list is paired with the following perfect key

```
53 int cacheKey[3] = {x&~((1<<sLog2X)-1),
54                    y&~((1<<sLog2Y)-1),
55                    z&~((1<<sLog2Z)-1)};
```

where `(x,y,z)` are the global coordinates used to access the corresponding node. Because this by construction avoids collisions and at compile time collapses to just three bit-wise AND instructions per node, determining whether `(x,y,z)` lie within a cached node can be done very efficiently in constant time. Then to locate a new voxel, we traverse the small cache list, and for each entry compute the corresponding `cacheKey` and check for a match with the buffered node's paired key. If a match is found, a shortened tree traversal is initiated from the corresponding node, and all newly visited child nodes and keys are updated in the cache list. Specifically this shortened tree traversal starts by deriving an offset into the corresponding node's own direct access table, for example, `mInternalDAT`, using the fast bit-wise operations in lines 46–52 of the code. Finally the tree traversal is terminated once a tile or `LeafNode` is reached, which on average results in an improved constant access time. However, in worst case, traversal of the list proceeds all the way to the root level, in which case a regular, top-down tree traversal is initiated and the cache list is completely rebuilt. This is all implemented as a compile-time-fixed linked list of paired cached nodes and keys in the `Accessor` class. This light-weight encapsulation of node caching provides convenient accelerated random access, and the various instances of this class are registered in the `mAccessors` member in the `RootNode` (see line 36 of the code).

It is important to note that, for this technique to work efficiently nodes at any level of the tree must be accessible directly via the global coordinates, rather than through expensive sequences of local coordinate transformations. Examining lines 39–52 of the code closely, it should be evident that this assumption is satisfied, even for signed global coordinates `(x,y,z)`. This is a unique and nontrivial consequence of VDB's design, in particular of the fact that it is height-balanced, has a fixed tree depth, and that the branching factors below the `RootNode` are powers of two and known at compile time.

3.3 Sequential Access

Many algorithms access or modify all active voxels in a grid, but are invariant to the specific sequence in which this happens. In particular, this is true for most time-dependent simulations, such as level set propagation or fluid advection. This invariance can be exploited if we can define a sequential access pattern that outperforms random access. We refer to the optimal sequence of data access, given by the order in which the data are physically laid out in memory, as "sequential access". With the advances of sophisticated cache hierarchies and prefetching algorithms in modern CPUs, it is increasingly important to fetch and process data in the order they are stored in main memory. Similarly, spatial data structures often have a significantly lower computational overhead associated with sequential access compared to random access. Both DT-Grid and HRLE store grid points in lexicographic order of their global coordinates, `(x,y,z)`, and they offer constant-time lookup and insertion for this type of sequential access only. In contrast, VDB features constant-time complexity of both random and sequential access (see Table I for a summary).

Table I. Average Access Time complexity for VDB vs DT-Grid

Access type	VDB	DT-Grid/H-RLE
Sequential lookup	O(1)	O(1)
Sequential insert	O(1)	O(1)
Sequential delete	O(1)	no support
Random lookup	O(1)	O(log[topology])
Random insert	O(1)	no support
Random delete	O(1)	no support

Random access for VDB is in the worst case linear in the depth of the tree, and since this depth is fixed we effectively achieve constant-time random access. In contrast, random lookup for DT-Grid/H-RLE depends on search of local grid topology. The order of sequential access for VDB is lexicographic in local `(x,y,z)` of the nodes whereas for DT-Grid/H-RLE it is lexicographic in the global grid `(x,y,z)`.

The challenge is to implement sequential access so that it outperforms the fast random access outlined before. The problem amounts to locating the next active value or child node, and the solution is surprisingly simple: iterate over the extremely compact `mChildMask` and `mValueMask` direct access bit masks embedded in each node. Their compactness makes them cache friendly (no need to load and search the much bigger direct access tables `mInternalDAT` and `mLeafDAT`), and multiple entries (bits) can be processed simultaneously. Computing the linear offset to the next set bit in a mask, `mMask`, starting from the bit position `offset`, is efficiently achieved by

```
56 uint32_t n=offset>>5, m=offset&31, b=mMask[n];
57 if (b & (1<<m)) return offset;//trivial case
58 b &= 0xFFFFFFFF << m;//mask out lower bits
59 while (!b) b=mMask[++n]; //find non-zero 32bits
60 return (n<<5)+DeBruijn[(b&-b)*0x077CB531U>>27];
```

where `DeBruijn` is a static table with 32 entries, and the constant 0x077CB531U is the hex representation of the 32-bit de Bruijn sequence [Leiserson et al. 1998]. Note that `b&-b` deletes all but the lowest bit and `>>27` isolates the upper five bits. Multiplication by the de Bruijn constant makes the upper five bits unique for each power of two, and the table lists the corresponding bit position of the lowest bit in `b`. Also note that line 60 of the code computes the lowest bit position without branching, that is, conditionals. The combination of these techniques allows us to implement sequential iterators over bit masks that very efficiently compute the linear offset into the corresponding direct access table. Since bit masks reside in all the nodes of a VDB tree, we can readily combine bit mask iterators at multiple levels of the tree, allowing us to derive iterators over active voxel values, active tile values, or `LeafNode`s, to mention just a few.

3.4 Stencil Access

Efficient stencil access on uniform grids is a fundamental requirement for Finite Difference (FD) computations. These schemes approximate differential operators with discrete differences of grid

values in a local neighborhood called the support stencil. Other common stencil applications are interpolation and filtering with convolution kernels of local support. The sizes and shapes of such stencils can vary widely depending on the accuracy and type of the FD scheme. For example, the optimally fifth-order accurate WENO FD scheme uses 18 neighboring grid points in 3D, whereas first-order single-side FD uses only six neighboring grid points. Typically these stencil access methods are combined with sequential access, which leads to stencil iterators, an essential concept for many numerical simulations.

Both DT-Grid and H-RLE achieve constant-time sequential stencil access by grouping multiple sequential iterators, one for each element in the support stencil. Hence, this approach has a computational overhead that scales linearly with the size of the stencil. With VDB we can take a simpler and more efficient approach that does not require the costly synchronization of multiple iterators. Instead we combine a single sequential iterator, Section 3.3, with the improved random access technique detailed in Section 3.2. The iterator defines the location of the center point of the stencil, and an `Accessor` provides accelerated access to the remaining stencil points. Since stencils are typically compact, this is an ideal application of node caching. For very large stencils, for example, during convolution, we can even employ multiple `Accessors`, each associated with a compact subset of the stencil, to decrease `LeafNode` cache misses that trigger more expensive tree traversals. Thus, in theory VDB has the same constant-time complexity for stencil access as DT-Grid, but in practice VDB amortizes the overhead of retrieving multiple stencil elements thanks to cache reuse.

To combine a sequential iterator with random access, we need an efficient way to derive the *global* coordinates, (x,y,z), from the linear `offset`, which uniquely identifies the position of the iterator within a node's bit mask. This is achieved by

```
x  = x0 + (offset >> sLog2Y + sLog2Z);
n  = offset & ((1 << sLog2Y + sLog2Z) - 1);
y  = y0 + (n >> sLog2Z);
z  = z0 + (n & (1 << sLog2Z) - 1);
```

where `(x0,y0,z0)` denotes the signed origin of the node encoded into `mFlags` for `LeafNodes` and `mX,mY,mZ` for `InternalNodes`.

Finally, we note that some applications actually require random stencil access. Examples are interpolation and normal computations during ray marching for volume rendering. It should be evident from the discussions so far that VDB also supports this type of access in constant time.

To shield the user from the complexity of the various access techniques detailed in Section 3, they are all neatly encapsulated in high-level STL-like `iterators` and `Accessors` with `setValue` and `getValue` methods that are easy to use and require no knowledge of the underlying data structure.

4. VDB APPLICATION ALGORITHMS

In this section we will focus on algorithms and techniques that have significant value for practical applications of VDB to simulations and sparse dynamic volumes. Additionally Appendices A and B describe how VDB lends itself to efficient yet simple compression and out-of-core streaming as well as optimizations like multithreading and vectorization.

4.1 Topological Morphology Operations

Besides voxel access methods, some of the most fundamental operations on dynamic sparse grids are topology-based morphology operations like dilation and erosion. These operations add or delete extra layers of voxels around the existing set, analogous to adding or removing rings of an onion, and they are used in interface tracking during deformations, convolution-based filtering, resampling, ray marching, etc. It is important to note that these operations work purely on the grid topology and not on voxel values[6].

Since VDB supports constant-time insertion and deletion of random voxels, it is tempting to implement topological morphology using these simple access methods. For instance, dilation could be achieved by iterating a neighborhood stencil containing the six closest neighbors over existing voxels and randomly inserting all new grid points intersected by the stencil. A disadvantage of this approach is that it generally requires two copies of the grid: one to iterate over and one to store the dilation in. Another major disadvantage is that this algorithm involves many redundant random access operations per voxel, since most of the stencil will intersect existing active voxels. Luckily, there is a much more efficient approach that exploits the fact that VDB separately encodes topology and values. Since topology is encoded into the `mValueMasks`, we can formulate topology algorithms directly in terms of these bit masks, which, as will be demonstrated in Section 5.5, turns out to be much faster than employing random-access operations. Thus, dilation can be implemented as the following *scatter* algorithm, where the contribution from each z projection of the `mValueMask` is scattered onto neighboring bit masks by means of simple bit-wise OR and shift operations.

```
for (int x=0; x<8; ++x) {
  for (int y=0, n=x<<3; y<8; ++y, ++n) {
    uint8_t b = oldValueMask[n];
    if (b==0) continue;//skip empty z-columns
    NN[0][n] |= b>>1 | b<<1;// +-z by itself
    NN[5][n] |= b<<7;//-z by NN[5]
    NN[6][n] |= b>>7;//+z by NN[6]
    (y>0 ? NN[0][n-1] : NN[3][n+ 7]) |= b;//-y
    (y<7 ? NN[0][n+1] : NN[4][n- 7]) |= b;//+y
    (x>0 ? NN[0][n-8] : NN[1][n+56]) |= b;//-x
    (x<7 ? NN[0][n+8] : NN[2][n-56]) |= b;//+x
}}
```

Here `oldValueMask` denotes the original undilated bit mask of a `LeafNode`, and `NN` is an array with seven pointers to the modified `mValueMasks` of the nearest neighbor `LeafNodes`, arranged in the order `0,-x,+x,-y,+y,-z,+z`. So `NN[0][7]` denotes the seventh byte of the `mValueMask` in the `LeafNode` currently being dilated, and `NN[4][0]` denotes the first byte of the `mValueMask` in the nearest neighbor `LeafNode` in the positive y direction. To simplify the code we have assumed that the `LeafNodes` have dimensions 8^3, but it is easy to generalize it to other dimensions as long as `Log2Z` is 3,4,5, or 6, corresponding to 8-, 16-, 32-, or 64-bit operations. This code performs a topological dilation by one voxel and is very fast, since it updates eight bits at a time and avoids random access to voxels. Also note that this algorithm does not require two copies of the grid; only the very compact `oldValueMasks` are copied, and only for the duration of the dilation. A topological erosion can be implemented in a similar fashion, using bit-wise AND and shift operations to *gather* the contribution from each neighboring z projection of the bit mask. By now it should be apparent that the bit masks, however simple they may seem, are key elements of VDB. To recap, we use them for hierarchical topology encoding,

[6]Topological morphology operations should not be confused with level set morphology operations that modify both voxel values and grid topology.

sequential iterators, lossless compression, topological morphology, and boolean operations.

4.2 Numerical Simulation on VDB

We shall briefly describe how VDB can be used for time-dependent simulations. We will demonstrate this on level sets, but emphasize that other time-dependent PDEs like the Navier-Stokes equations can be solved using similar techniques.

Level set methods typically consist of two fundamental steps [Breen et al. 2004; Museth et al. 2005]. First, the voxel values are updated by solving a Hamilton-Jacobi equation, and second, the topology is updated by rebuilding the narrow band so that it tracks the moving interface.

The first step allocates as many temporal value buffers in the `LeafNodes` as the integration scheme requires and then sequentially updates the voxels using a stencil iterator appropriate for the spatial finite-difference scheme used to discretize the Hamilton-Jacobi equation. For instance, to accurately solve hyperbolic advection for applications like free-surface fluid advection, we would allocate three buffers to support third-order TVD-Runge-Kutta [Shu and Osher 1988] and a stencil iterator with 19 elements for optimally fifth-order WENO [Liu et al. 1994].

The second step is implemented as follows: first, dilate the existing narrow band by as many voxels as the interface has moved. Then, renormalize the voxels, and finally, trim away voxels that have moved too far away from the interface. Because the CFL condition for the explicit TVD-RK integration scheme implies that the interface never moves more than one voxel per integration, we just apply the dilation algorithm in lines 65–76 of the code once. To renormalize the dilated voxels we solve the hyperbolic Eikonal PDE using the same discretization schemes employed for the Hamilton-Jacobi PDE. Finally, we use a sequential iterator to delete voxels with values larger than the fixed width of the narrow band, and we prune away any nodes or branches of the tree with an empty (zero) `mValueMask`.

4.3 Hierarchical Constructive Solid Geometry

Constructive Solid Geometry (CSG) is one of the most important applications for implicit geometry representations like level sets. It turns out that the underlying tree structure of VDB acts as an acceleration data structure that facilitates efficient CSG operations. In fact, as will be demonstrated in Section 5.7, we can achieve near-real-time performance for boolean operations on very high-resolution level sets. The trick is surprisingly simple: rather than perform CSG voxel-by-voxel, we can often process whole branches of the VDB tree in a single operation. Thus, the computational complexity scales only with the number of intersecting `LeafNodes`, which is typically small. In lines 77–90 we illustrate this idea with code to perform a boolean union at the level of the `InternalNodes`. (The logic for the `RootNode` is similar, whereas the `LeafNodes` implement the CSG operation (e.g., min or max) at voxel level.)

4.4 Hierarchical Boolean Topology Operations

Whereas CSG conceptually performs boolean operations on geometry, it is sometimes useful to define boolean operations directly on the topology of the active values in two grids, particularly if the grids are of different types. For example, it might be useful to union the active value topology of a scalar grid with that of a vector grid. As should be evident from the discussions previous this is efficiently achieved in VDB with hierarchical bit-wise operations on the respective `mValueMasks`.

```
77 void csgUnion(InternalNode *other) {
78 for (Index n=0; n!=Size; ++n) {
79   if (this->isChildInside(n)) continue;//skip
80   if (other->isChildInside(n)) {
81     this->makeChildInside(n);
82   } else if (this->isChildOutside(n)) {
83     if (other->isChild(n)) this->setChild(n,
84       other->unsetChild(n));//transfer child
85   } else if (other->isChild(n)) {
86     mInternalDAT[n].child->csgUnion(
87     other->InternalDAT[n].child));//recurse down
88     if (InternalDAT[n].child->isEmpty())
89       this->makeChildInside(n);//prune empty
90 }}}
```

4.5 Mesh To Level Set Scan Conversion

A common way to construct a volume is to convert an existing polygonal mesh into a signed distance function. This process of converting explicit geometry into an implicit representation is commonly referred to as scan conversion. While there are several fast algorithms for scan conversion [Breen et al. 1998; Chang et al. 2008], they are all based on regular dense grids, and as such have significant limitations on achievable resolution. However, as demonstrated in Houston et al. [2006, Section 6.2], it is relatively easy to modify these dense scan converters to work with sparse data structures. The fundamental idea is to partition the mesh into nonoverlapping tiles such that each tile contains all the polygons of the original mesh that lie within a distance of $\beta/2$, where β is the width of the narrow band. Then, the submeshes are scan converted into the appropriate dense tiles using a dense grid algorithm like Mauch [1999]. The problem is that most existing sparse level set data structures, like DT-Grid and H-RLE, do not support random insertion (or deletion), but require the grid points to be inserted in lexicographic order of the coordinates. Hence, an additional step is required before the level set can be generated, namely three bucket sorts which are both time and memory consuming. With VDB, however, the distance values can be inserted in the random order in which they are computed. In fact, since VDB is already a blocked grid we can further improve performance by using the `LeafNodes` as tiles. This allows us to apply the scan converter directly to the `LeafNodes`, avoiding the overhead of the auxiliary data structure that allocates the dense tiles. Overall, VDB lends itself well to sparse scan conversion, making it attractive for movie production, where geometry is usually modeled explicitly.

Finally, we note that support for random insertion in VDB is also very important when converting other types of geometry into volumes. Examples include "skinning" of animated particle systems typically generated from SPH, PIC, or FLIP fluid simulations, where VDB allows us to rasterize the particles in arbitrary order, again unlike DT-Grid and H-RLE.

4.6 Hierarchical Flood-Filling

The scan conversion of meshes or particles described earlier produces a sparse VDB with active voxels, that is, `LeafNode` values, within a *closed* manifold narrow band only. For inactive voxels and and tiles outside the narrow band a little more work is needed to determine correct signed values. Fortunately, the outward propagation of signs from the active voxels in the narrow band can be implemented efficiently with VDB and incurs negligible computational overhead in the overall scan conversion. The basic idea is to recast the problem as a hierarchical boundary-value propagation of the known (frozen) signs of active voxels. Since, by construction,

all `LeafNodes` are guaranteed to contain active values, they can be flood-filled *concurrently* using the following algorithm.

```
int i = mValueMask.findFirstOn();
for (int x=0; x!=1<<Log2X; ++x) {
  int x00 = x << Log2Y + Log2Z;
  if (mValueMask.isOn(x00)) i = x00;
  for (int y=0, j=i; y!=1<<Log2Y; ++y) {
    int xy0 = x00 + (y << Log2Z);
    if (mValueMask.isOn(xy0)) j = xy0;
    for (int z=0, k=j; z!=1<<Log2Z; ++z) {
      int xyz = xy0 + z;
      if (mValueMask.isOn(xyz))
        k = xyz;
      else
        mLeafDAT.values[xyz] = copysign(
        mLeafDAT.values[xyz],mLeafDAT.values[k])
}}}
```

This algorithm works by scanning the active values in the mLeafDAT in lexicographic order and consistently transferring signs from active to inactive values. The resulting signs are then used as boundary values for a similar scanline algorithm applied recursively to the parent nodes until the process terminates at the top level. Thus, by virtue of the hierarchical representation of the narrow band, no sign is ever required to be propagated between nodes at the same (or a lower) tree level. Consequently, the overall signed flood-filling is strictly bottom-up, and in practice, this operation can be performed in near real time (see Section 5.6).

4.7 Accelerated Ray Marching

Another benefit of VDB over existing level set data structures like DT-Grid, and tiled grids like DB-Grid, is that its hierarchical tree structure acts as a multilevel bounding volume acceleration structure for ray marching. Narrow-band level set data structures can only ray leap a distance corresponding to the width of the narrow band, and tiled grids can only skip over regions corresponding to a single tile size. However, VDB can leap over larger regions of space by recursively intersecting the ray against nodes at any level of the tree. One possible manifestation of this idea is a hierarchical, that is, multilevel, 3D Bresenham algorithm that improves the performance of ray integration in empty or constant regions of index space represented by multilevel tile values. Figure 1 shows two final frames from the animated feature *Puss in Boots*, both generated using VDB-accelerated volume rendering of time-varying clouds [Miller et al. 2012]. The right image in Figure 7 shows accelerated direct ray-tracing of a deforming narrow-band level set represented in a VDB.

5. EVALUATION AND BENCHMARKS

To support our claims regarding performance of VDB we benchmark various configurations of VDB against existing sparse data structures, focusing especially on DT-Grid [Nielsen and Museth 2006], since it also supports numerical simulations and has been demonstrated to outperform octrees and H-RLE in both access time and memory footprint (see Chapter 7 of Nielsen [2006]). However, for completeness, and whenever feasible, we will also include benchmark comparisons to other sparse data structures, including tiled grids, octrees, and N-trees.

To best facilitate comparisons to other data structures, we strive to present benchmarks that are easily reproducible. Hence we limit

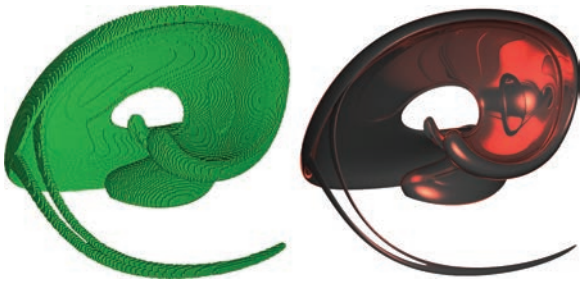

Fig. 7. Middle frame of deformation test [Enright et al. 2002] @ 2048^3 on the Utah teapot. Left: `LeafNodes`, Right: Direct ray-tracing of VDB.

our comparisons to simple, well-documented setups, as opposed to complex production examples, and to open-source reference implementations whenever possible. To this end we use the following open-source reference data structures: DT-Grid [2009], sparsehash [2009], and Field3D [2009]. The latter includes a tiled sparse grid, SparseField, developed specifically for VFX, which we shall refer to as F3DSF, short for Field3D::SparseField. DB-Grid[Museth et al. 2007] is a proprietary data structure developed at Digital Domain, so we cannot include it in our evaluation, but like F3DSF it is a sparse tiled grid, though the implementations are rather different.

The setup for the actual data on which we perform the benchmark tests is described in Enright et al. [2002], and our reference implementation is taken from DT-Grid [2009]. This so-called "Enright test" has become a de facto standard for benchmarking level set data structures and is easy to duplicate, but since the standard test employs extremely simple topology (a sphere), it favors the DT-Grid due to its logarithmic dependency on local topology for random access. Consequently, we will additionally employ an "8x" version of the Enright test, which was recently introduced in Christensen et al. [2011] (see Figure 10). All narrow-band level sets will be $(2 \times 5) = 10$ voxels wide, which is the configuration of choice in DT-Grid [2009] when using the high-order WENO scheme.

All reported CPU times and memory footprints are computed as the statistical average of five runs on a workstation with dual quad-core Intel Nehalem-EP W5590 CPUs with 4×32KB L1, $4 \times$ 256KB L2, 8MB L3, and 48GB RAM (DDR3-1333). Unless otherwise specified, all benchmark numbers are measured on a single computational thread, and memory usage is the memory consumption registered by the operating system (Red Hat Enterprise Linux v5.4). Our testing has shown that this can be reliably determined by examining `/proc/self/statm` between grid allocations. This produces larger but more realistic memory footprints than theoretical estimates that ignore issues like data structure misalignment.

Throughout this section we shall use the following notation for the various configurations of VDB: $[R, I_1, \ldots, I_{D-1}, L]$, where R specifies the type of hash map employed by the `RootNode`, I_d denotes the base two logarithmic branching factors (Log2w) of the `InternalNodes` at level $1 \leq d < D$, and L denotes the Log2w of the `LeafNodes`. Thus, D specifies the *depth* of the underlying B+tree, and $[Hash, 5, 4, 3]$ denotes a dynamic VDB of depth three with a hash table encoding branches of size $32 \times 16 \times 8 = 4096$. Alternatively we use the notation $[I_0, I_1, \ldots, I_{D-1}, L]$ for static VDB configurations with `InternalNodes` at all levels $0 \leq d < D$, that is, [10, 3] denotes a tiled grid with an available grid resolution of $(2^{10} \times 2^3)^3 = (1024 \times 8)^3 = 8,192^3$.

5.1 Random Access

Table II shows random access times, memory footprints, and effective resolutions of various configurations of VDB and other sparse data structures measured on two different datasets. The red values correspond to the original Enright sphere and the blue values to the 8x variation shown in Figure 10. Both sparse level sets have an effective data resolution of $4,096^3 = (2^{12})^3$, as defined in Enright et al. [2002], and a narrow-band width of ten voxels corresponding to, respectively, 51,033,829 and 263,418,462 active voxels.

The random access tests in Table II measure the average computational overhead of looking up an arbitrary active voxel value. The reported times are averages of times per lookup over 100 random coordinates, each repeated 10^7 times to eliminate the overhead of fetching data from main memory. Note that for this benchmark test it is essential to declare the reused coordinates `volatile`, since most C++ compilers will otherwise optimize away repeated lookups with identical arguments. The specific access pattern consists of randomized coordinates constrained to the narrow band, so as to measure lookup times for leaf voxels rather than tile values. Since each of the random lookups is repeated several times, on average, the data being accessed reside entirely in the cache. This effectively amortizes cache misses that could otherwise dominate the cost of navigating the data structure. The access pattern and number of repetitions are of course identical for *all* the timings listed in Table II, and the average times per lookup are measured relative to the fastest observed access time. As a cautionary note, we stress that absolute timings of random access can be sensitive to the specifics of the underlying test, hardware and even compiler. However, we feel confident that the relative CPU times are useful for a baseline comparison of the various data structures.

For all the VDB configurations in Table II (the first 4×4 rows), random access times are computed using two different algorithms: the optimization trick described in Section 3.2 and the slower "brute-force" top-down tree traversal described in Section 3.1. These two measurements represent respectively the best and worst random access performance of VDB in terms of varying spatial coherence of the access pattern. However, it is important to stress that the best-case scenario of `LeafNode`-coherent access is common in practice, so the low constant access times in Table II are actually typical for random access of VDB.

The last column of Table II lists the available resolution of the various grid configurations, not to be confused with the effective resolution of the data, which is fixed at $4,096^3$, as noted before. We define the available resolution as the actual index range of the grid. It is important to recall that only VDB configurations with a `RootNode` have conceptually infinite available resolution, a nontrivial feature only shared by DT-Grid. However, the available resolution of any grid is ultimately limited by the bit-precision of the coordinates (a grid that uses 32-bit integer coordinates cannot be larger than $(2^{32})^3 = (4,294,967,296)^3$) and of course by the available memory. All the grids appearing in Table II are configured with an available resolution of at least $8,192^3$, which we consider the minimum for a grid to be regarded as high resolution. The only exception is [3, 3, 3, 3], which by construction has a small, fixed resolution of $4,096^3$.

In the first four rows of Table II the VDB tree depth varies from 4 to 1, while the `LeafNode` size is fixed at 8^3, and branching factors increase from the bottom up. This is the general configuration recommended in Section 2.4. The results suggest that worst-case random access depends linearly on the tree depth but is otherwise independent of both the data topology and size, whereas spatially coherent random access is independent of the tree depth and has almost negligible cost. In fact, we use this constant `LeafNode`-coherent access time as the unit of relative time for all the reported lookups in Table II. In addition, we observe that the memory footprint of the very shallow [10,3] configuration is very large, even compared to [6,4,3], which has the exact same available resolution. This is a consequence of the large, $(2^{10})^3 = 1024^3$, `mInternalDAT` of [10,3], a significant limitation shared by other tiled sparse grids (see the following). Finally, the available resolutions of the first two configurations of depth 4 and 3 are orders of magnitude larger than the last two. In conclusion, [6,5,4,3] seems to offer the best balance of fast random access, low memory foot prints, and high available resolution.

In the next four rows of Table II (5–8) the VDB depth is fixed to three, and only the size of the `LeafNode` varies. As expected, both the worst- and best-case random access are constant, however the memory foot,prints and available resolutions change dramatically. The memory footprint decreases as the `LeafNode` size decreases, because the number of inactive voxels decreases. This clearly favors [6,5,4,3] and [6,5,4,2], which have almost the same memory footprint, whereas the larger `LeafNodes` of [6,5,4,3] are favored for available resolution. Again, the best balance is arguably [6,5,4,3].

Table II. Random Lookup Times and Memory Footprints

Enright|8× Grid config.	Random Lookup Section 3.2 ↑ | Section 3.1 ↓				Memory Footprint (megabytes)		Available Resolution
[7,6,5,4,3]	1	14	1	15	439	2,275	$33,554,432^3$
[6,5,4,3]	1	10	1	10	422	2,259	$262,144^3$
[6,4,3]	1	7	1	7	420	2,256	$8,192^3$
[10,3]	1	3	1	3	8,723	10,492	$8,192^3$
[6,5,4,5]	1	11	1	11	1,078	5,904	$1,048,576^3$
[6,5,4,4]	1	11	1	11	625	3,398	$524,288^3$
[6,5,4,3]	1	10	1	10	422	2259	$262,144^3$
[6,5,4,2]	1	10	1	10	389	2,076	$131,072^3$
[6,6,6,6]	1	11	1	11	1,977	11,191	$16,777,216^3$
[5,5,5,5]	1	11	1	11	1,072	5,899	$1,048,576^3$
[4,4,4,4]	1	11	1	11	625	3,398	$65,536^3$
[3,3,3,3]	1	11	1	11	413	2,218	$4,096^3$
[Hash,4,3,2]	1	18	1	18	362	1,917	∞
[Hash,5,4,3]	1	17	1	17	420	2,257	∞
[Map,4,3,2]	1	25	1	33	362	1,917	∞
[Map,5,4,3]	1	14	1	14	420	2,257	∞
[F3DSF,3]	14		14		33,156	34,846	$8,192^3$
[F3DSF,4]	14		14		4,698	7,366	$8,192^3$
[F3DSF,5]	15		15		1,586	6,407	$8,192^3$
Octree#	50		50		390	2,048	$8,192^3$
Octree*	70		70		390	2,048	$262,144^3$
DT-Grid	22		25		253	1,182	∞

Relative CPU times for 10^7 lookups of 100 random lookups in two different data sets (Enright et al. [2002] and the 8×[Christensen et al. 2011] variant), both at an effective resolution of 4096^3. For all tests, one time unit is approximately $\frac{1}{2}$ second. For VDB, times are reported using the two different random access techniques described in Section 3.1 (worst case top-down) and Section 3.2 (cached bottom-up). "Hash" and "Map" denote `RootNodes` with respectively a hash table [sparsehash 2009] and `std::map`, and "[F3DSF,X]" refers to Field3D [2009] with a block size of 2^X. The memory footprints are given in 1024^2 bytes (MB) as reported by the OS (RHLE v5.4), and no in-core compression or bit quantization is applied.

The next four rows of Table II (9–12) benchmark configurations of VDB with fixed depth and branching factors, effectively corresponding to N-trees [7]. While the random access times are constant, both the memory footprints and available resolutions show extreme variations. For example, [3,3,3,3] has a very small memory footprint but unfortunately also the lowest available resolution of any configuration in Table II. In contrast both [6,6,6,6] and [5,5,5,5] have very significant memory overheads. Of these N-trees, [4,4,4,4] offers the best balance, but the memory footprint is still larger than almost all trees with variable branching factors, and the available resolution is significantly smaller than, in particular, [7,6,5,4,3] and [6,5,4,3].

The next four rows of Table II (13–16) benchmark configurations of VDB employing `RootNodes` with either a hash table or a `std::map` (the former combines the improved hash function in lines 39–45 of the code with a dense_hash_map[sparsehash 2009] which is the fastest open-source implementation we are aware of) together with internal and leaf nodes with branching factors $2^5 \times 2^4 \times 2^3 = 4096$ and $2^4 \times 2^3 \times 2^2 = 512$. Table II clearly shows that the memory footprints are relatively small and identical for the two types of sparse root table. However, of the two sets of branching factors, [4,3,2] has the smallest footprint, primarily because the dense `LeafNodes` are smallest. While the top-down random lookup times for [Hash,5,4,3] and [Hash,4,3,2] are virtually identical for both data sets, the same is not true for the `std::map` variants. In fact [Map,5,4,3] has the fastest constant access time for both datasets, whereas [Map,4,3,2] is significantly slower and also increases with the size of the dataset. This is a consequence of the fact that (on average) hash tables have constant random access whereas map lookup is logarithmic in the table size. Specifically, for the two datasets in Table II, all [5,4,3] configurations have only one entry in the `mRootTable`, whereas the [4,3,2] variants have respectively 29 and 160 entries. However, all configurations have identical `LeafNode`-coherent lookup times, but [5,4,3] is on average more likely to amortize the `mRootTable` access complexity since the tree branches span a larger coordinate domain than [4,3,2]. In summary [Hash,5,4,3] offers the best balance of all the factors.

The last six rows of Table II list results generated with competing sparse data structures for dynamic volumes, including three configurations of F3DSF, two octrees, and DT-Grid. The three F3DSF configurations employ block sizes of, respectively, 8^3, 16^3, and 32^3, 16^3 being the default configuration. All F3DSF configurations have relatively large memory footprints and slow random access, especially when compared to [10,3] which is conceptually a similarly configured tiled grid. The better performance of [10,3] is likely due to the implementation details presented in Section 2.3 and Section 3.1. Regardless, this illustrates why a tiled grid with a dense root table does not generally scale to high resolutions. Conversely, both octrees have relatively small memory footprints due to their extremely small branching factor of two, but unfortunately they suffer slow random access due to their tall tree structures, even with a modest available resolution of $8,192^3$. This is despite the fact that we implemented the octrees using the optimizations described in Stolte and Kaufman [1998] and Frisken and Perry [2002]. The only differences between Octree$^{\#}$ and Octree* in Table II are the tree depths of, respectively, 13 and 18, with corresponding available resolutions $8,192^3$ and $262,144^3$. DT-Grid on the other hand was developed specifically for narrow-band level sets and hence has an impressively small memory footprint, but its random access is the slowest of all in the table, with the notable exceptions of octrees and top-down traversal of [Map,4,3,2]. As with [Map,4,3,2], random access for DT-Grid has a dependency on the dataset that is caused by the logarithmic lookup complexity on local topology. However, the biggest disadvantages of DT-Grid are that, by construction, it is limited to narrow-band level sets and it does not offer random insertion and deletion. Finally we note that DT-Grid also supports conceptually unbounded grids.

[7] We stress that similarity to the N-tree proposed in Crassin et al. [2009] is only superficial, since VDB introduces novel access algorithms.

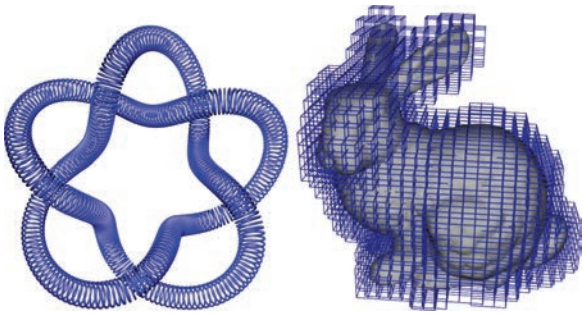

Fig. 8. Left: Torus-knot with a sweeping helix represented as a level set *surface* in a VDB with an effective resolution of 4096^3. Included as an example of complex grid topology, it was created by taking the CSG union of *5 million* spheres positioned randomly along a parametric representation of the Torus-Knot-Helix. It only took about three seconds to generate the level sets using multithreading. Right: Stanford bunny shown with `LeafNodes`.

As a practical application of random insertion consider Figure 8, which shows a complex level set surface (*not* a curve), generated by rasterizing into a [6,5,4,3] VDB five million animated spheres distributed randomly along an analytical Torus-Knot-Helix (TKH) curve. This narrow-band level set, with an effective resolution of $4,096^3$, was rasterized on our eight-core workstation in about three seconds per frame.

In summary, VDB offers fast average random access and relatively small memory footprints. In fact, for the common case of spatially coherent access, our specific tests suggest that VDB can be significantly faster than octrees, F3DSF, and DT-Grid. Of the many different configurations of VDB listed in Table II, [Hash,5,4,3] and [6,5,4,3] seem to strike a good balance between random access, memory footprint and available resolution, which is consistent with the predictions in Section 2.4.

5.2 Simulation Using Sequential Stencil Access

To evaluate sequential stencil access we use the level set simulation described in Enright et al. [2002]. This "Enright test" essentially mimics the numerical computations (solving hyperbolic PDEs) typical for interface tracking in fluid animations. Specifically, geometry is advected in an analytical divergence-free and periodic velocity field [Leveque 1996], so that the volume loss is an indication of numerical dissipation due to finite differencing discretization errors. Since this truncation error diminishes (often nonlinearly) as the voxel size decreases, volume loss diminishes as the grid resolution increases. This is illustrated in Figure 9, where very detailed geometry (the Armadillo model) is advected at an effective resolution of 4096^3 and by the final frame shows few signs of volume loss. Similar advections in the Enright velocity field are shown for the "half-period" frames in Figure 7 and Figure 10 for, respectively, the Utah teapot at 2048^3 and the eight-sphere dataset used in Table II at 4096^3, the latter containing more than a billion active voxels. However, for the following benchmarks we use the original dataset (a single sphere) described in Enright et al. [2002]. This makes our results easier to reproduce, especially since this "vanilla Enright

Fig. 9. Level set application of high-resolution VDB with configuration [5,4,3,2]. Divergence-free advection of the Armadillo at an effective resolution of 4096^3. Far left: Close-up of the dynamic 4^3 `LeafNodes`. Remaining frames show time= 0, $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, 1 period. At the half-period the voxel count is 90,398,733. Notice how the initial (time=0) and last (time=1) frames appear almost identical, indicating low volume loss due to numerical dissipation.

Table III. Simulations Times For VDB and DT-Grid in Seconds

Enright	Active Voxels	DT-Grid sec/itera	[6,5,4,3] sec/itera	$[6, 5, 4, 3]^{\|\|}$ sec/itera
512^3	794,720	2.40	0.61 (3.9×)	0.08 (30×)
$1,024^3$	3,189,240	9.56	2.46 (3.9×)	0.29 (33×)
$2,048^3$	12,784,621	37.8	9.75 (3.9×)	1.13 (34×)
$4,096^3$	51,033,829	154	39.1 (3.9×)	4.39 (35×)
$8,192^3$	203,923,476	613	157 (3.9×)	17.7 (35×)
$16,384^3$	815,936,095	2,454	636 (3.9×)	70.8 (35×)

CPU times in seconds for a single time integration step of the standard Enright test [Enright et al. 2002] at various effective data resolutions. For VDB, the speedup over DT-Grid is shown in parentheses. In the last column, $[6, 5, 4, 3]^{||}$, VDB is fully multithreaded on two quad-core CPUs. Note that [Hash,5,4,3] and [Map,5,4,3] exhibit virtually identical performance to [6,5,4,3] and are therefore not listed.

test" is included in the open-source DT-Grid distribution [DT-Grid 2009]. As noted before, the extreme simplicity of this dataset is likely to favor the DT-Grid data structure.

Table III lists CPU times in seconds for a single-time integration step of the original Enright test at various effective data resolutions. We define this integration step as one hyperbolic advection step in the analytic incompressible Enright field with $t = 0$, followed by three renormalization steps of the level set, and finally a complete rebuild of the narrow band. In the case of VDB, these steps are implemented using the techniques described in Section 4.2, and as in DT-Grid [2009], we employ third-order TVD-Runge-Kutta [Shu and Osher 1988], optimally fifth-order WENO [Liu et al. 1994], and a narrow-band width of ten voxels. Similarly, for the time integration of the advection PDE the CFL is set to 0.5, whereas a CFL of 0.3 is used for the Eikonal PDE. Our comparison is necessarily limited to DT-Grid since, to the best of our knowledge, it is the only open-source sparse data structure that includes support for all the required level set operations, that is, advection, renormalization, and narrrow-band rebuild. However, as shown in Nielsen and Museth [2006] and Nielsen [2006], DT-Grid outperforms octrees, H-RLE, and dense grids for level set computations.

We first note that VDB ([6,5,4,3]) is consistently faster than DT-Grid, despite the fact that DT-Grid was designed for exactly this type of level set computation. On a single computational thread, the speedup factor is approximately four, which, curiously, corresponds to the increase in surface voxels for a doubling of the grid resolution. Hence, in the time it takes DT-Grid to complete one time integration step at a given resolution, VDB can perform the same simulation at twice the resolution, that is, four times the data size. We suspect that this performance advantage is due to a combination of factors, including improved data locality (i.e., cache reuse), better hardware utilization (see Appendix B), more efficient stencil iterators, and the general algorithmic efficiency of random insertion/deletion and topological dilation (Section 4.1) for the narrow-band rebuild.

The last column in Table III lists CPU times for a time integration step using multithreaded advection and renormalization on a [6,5,4,3] VDB. The speedup is almost equivalent to the number of physical cores (8), and for five out of six resolutions it is actually superlinear, suggesting an effect from the larger aggregate cache of multiple CPUs. This explanation is supported by the fact that superlinear speedup is not observed for the smallest problem size, 512^3, which, as we shall see in Table IV, fits into the 8MB L3 cache of a single CPU. Unfortunately, DT-Grid [2009] is not multithreaded, and while there is no reason to believe that it could not be parallelized, with the possible exception of the rebuild step, the complexity of this data structure makes this nontrivial. As such we conclude that for these numerical simulations [6,5,4,3] offers very significant performance advantages (4× to 35×) over DT-Grid. Finally we note that both [Hash,5,4,3] and [Map,5,4,3] perform the benchmarks in Table III indistinguishably from [6,5,4,3], which is to be expected since this simulation is dominated by sequential stencil access.

In the more practical context of surface tracking in fluid simulation, level set advection often involves velocity fields represented on MAC grids. These "staggered" grids conceptually store the Cartesian components of the velocities at cell faces rather than colocated at cell centers. This is a popular technique to suppress so-called checkerboard frequencies in the pressure projection step of most Navier-Stokes solvers. However, instead of devising specialized data structures for MAC grids, it is customary to use standard collocated grids, and simply move the staggered interpretations of the grid indexing and values to client code. In other words, VDB fully supports MAC grids, and we use special interpolation schemes to convert from staggered to colocated velocities required for level set advection.

5.3 Memory Footprints

Table IV lists memory footprints of DT-Grid, F3DSF and two configurations of VDB, encoding the Enright dataset reported in Table III. For VDB we include memory footprints for both in-core (red) and partially out-of-core (blue) storage (see Appendix A). Unlike in Table II, the available resolution of F3DSF is identical to the effective data resolution, since this leads to the smallest memory footprint. The block size of F3DSF is kept at its default value of 16^3. As observed in Table II, the significant overhead of allocating the dense root table in F3DSF can be reduced by increasing the block

Table IV. Comparing Memory Footprints in MB

Enright	DT-Grid	[F3DSF,4]	[Hash,4,3,2]	[Hash,5,4,3]
512^3	4.3	10.5	5.7/1.0	6.7/0.4
$1,024^3$	16.9	45.9	22.7/3.6	26.5/1.9
$2,048^3$	69.8	215.6	90.8/17.2	105.3/7.1
$4,096^3$	253.0	1,114.2	361.9/68.7	420.2/29.8
$8,192^3$	904.9	6,501.2	1,444.3/274.0	1,679.4/117.2
$16,384^3$	3,619.7	>48,000.0	5,780.1/1,097.6	6,722.9/468.2

Memory footprints in MB for DT-Grid, F3DSF, and VDB. The dataset is the standard Enright test [Enright et al. 2002] at various effective data resolutions. The corresponding active voxel counts are given in Table III. Red indicates completely in-core and blue indicates partially out-of-core (see Appendix A). No compression or bit quantization is applied to any of the in-core data structures. F3DSF at $16,384^3$ ran out of memory on our 48GB workstation.

size. However, it should be noted that F3DSF, unlike VDB, cannot distinguish between active and inactive voxels within the blocks, which implies that large block sizes can lead to computational overheads from processing dense blocks versus sparse voxels.

From Table IV we conclude that, for the Enright dataset, DT-Grid has the smallest in-core memory footprint, which is consistent with Table II and is not surprising, considering that DT-Grid was specifically optimized for (but also limited to) narrow-band level sets. Of the two VDB configurations, the in-core [Hash,4,3,2] consistently has the smaller memory footprint, due to its smaller `LeafNode` size of 4^3, whereas the out-of-core [Hash,5,4,3] has the smaller memory footprint due to its smaller tree. Both in-core VDB configurations have a memory footprint less than twice that of DT-Grid, whereas the out-of-core configurations are almost an order of magnitude smaller. However, the current implementation of out-of-core VDB does not support dynamic topology or stencil-based finite-difference computations, and as such is limited to load-on-read applications like volume rendering. F3DSF consistently has the largest in-core memory footprint, and Table IV clearly illustrates how the overhead of the dense block table increases rapidly with the resolution. In fact, $16,384^3$ could not be achieved with the available 48GB of RAM.

5.4 File Sizes and Write Speeds

For narrow-band level sets, we saw that DT-Grid has the smallest in-core memory footprint, but in a production environment where network-attached storage arrays are expensive and bandwidth is limited, file sizes are often a bigger concern, especially when dealing with time-varying data.

Table V compares files sizes of DT-Grid, F3DSF, and VDB representing the Enright dataset reported in Table III and Table IV. DT-Grid employs the Compressed-Row-Storage (CRS) codec [Nielsen and Museth 2006], whereas F3DSF uses [HDF5 2010] for I/O, which employs GZIP that is based on the lossless "Deflate" codec.[8] The F3DSF results are for the default block size of 16^3, whereas VDB is configured as [6,5,4,3].

The three last columns list VDB file sizes for various combinations of compression techniques. The first of these is the fast active-mask compression described in Appendix A, which leads to

Table V. Comparing File Sizes of DT-Grid, F3DSF and VDB

Enright	DT-Grid CRS	F3DSF Deflate	[6, 5, 4, 3] bit-mask	[6, 5, 4, 3] +Deflate	[6, 5, 4, 3] +Half
512^3	3.40	2.66	3.28	2.84	1.68
$1,024^3$	13.60	11.41	13.06	11.11	6.67
$2,048^3$	54.49	50.16	52.23	43.47	26.74
$4,096^3$	217.40	234.66	208.46	166.93	106.46
$8,192^3$	868.53	1235.88	832.84	648.72	422.39
$16,384^3$	3,474.81	?	3,332.45	2,510.33	1,669.48

Files sizes in MB for DT-Grid, F3DSF and VDB [6,5,4,3]. CRS denotes the "compressed row storage" scheme employed by DT-Grid. "Deflate" is the codec used in GZIP, F3DSF and optionally VDB. Bit-mask refers to the compression scheme described in Appendix A. In the last column, bit-mask and Deflate compression are combined with 32- to 16-bit float quantization. F3DSF at 16384^3 requires more than the available 48GB of RAM, so the file size could not be determined.

file sizes that are slightly smaller than for DT-Grid. Next, we apply the lossless Deflate[9] codec to the bit-mask compressed voxel array. We observe that this reduces the files sizes of VDB by 13% to 25%. Finally we apply lossy 32- to 16-bit quantization using the "half float" implementation from the OpenEXR library. While Deflate and bit quantization are not currently supported by DT-Grid [2009], they could certainly be added. Regardless, for all the level sets in Table V VDB consistently has the smallest file sizes, despite the fact that DT-Grid was explicitly optimized for narrow-band level sets. This is largely because VDB saves only active voxels and bit masks, eliminating the overhead of dense tree nodes. It is also interesting to note that, with Deflate compression, VDB produces much smaller file sizes than F3DSF (which also uses Deflate), with the exception of the very smallest data resolution of 512^3. In fact, the difference in file sizes between VDB and F3DSF grows significantly as the data size increases. Since the in-core F3DSF at $16,384^3$ could not be generated, neither could the corresponding file.

Though small file sizes are desirable for obvious reasons, write times can also be important, especially in the context of simulations, where output of temporal data can be a bottleneck. Table VI reports CPU write times corresponding to the files listed in Table V. The first thing to note is that despite the fact that DT-Grid and [6,5,4,3] have similar file sizes, DT-Grid writes 3–4 times slower than the bit mask-compressed VDB. We are somewhat surprised by this, and can only attribute it to the superior performance of the bit-based codec in VDB over CRS in DT-Grid. With Deflate compression, VDB is also significantly faster than F3DSF, the more so when bit quantization is also applied (prior to Deflate). Finally we note that [Hash,5,4,3] and [Map,5,4,3] produce file sizes and write times similar to [6,5,4,3].

5.5 Morphological Dilation

Table VII lists timings for topology dilations of the datasets in Table III for VDB and DT-Grid, the only two formats with native support for this operation. For the column labeled "[6,5,4,3] Optimal", we use the bit mask-based topology dilation algorithm described in Section 4.1, and for the last column we use brute-force random access. Observe that VDB is consistently faster ($7\times$) than DT-Grid, and that the algorithm in Section 4.1 is more than an order of magnitude ($17\times$) faster than the brute-force approach using

[8] HDF5 is distributed with the proprietary compression scheme SZIP, which was enabled by default since HDF5 1.6.0. SZIP embodies patents that are held by the National Aeronautics and Space Administration and requires a license for commercial use. Thus, for all the HDF5-related tests presented in this article we have replaced SZIP with the free GZIP compression scheme.

[9] Deflate, also used by F3DSF, is a generic data compression algorithm that combines the LZ77 algorithm with Huffman coding. A reference implementation of Deflate is available in the open-source zlib library.

Table VI. Comparing Write Times for DT-Grid, F3DSF and VDB

Enright	DT-Grid CRS	F3DSF Deflate	[6, 5, 4, 3] bit-mask	[6, 5, 4, 3] +Deflate	[6, 5, 4, 3] +Half
512^3	0.04	0.58	<0.01	0.25	0.19
$1,024^3$	0.13	2.20	0.04	1.02	0.74
$2,048^3$	0.47	8.81	0.18	4.18	3.21
$4,096^3$	1.84	35.35	0.65	16.83	13.11
$8,192^3$	9.53	148.75	2.34	67.08	48.67
$16,384^3$	47.22	?	10.33	258.31	192.92

CPU times in seconds to write DT-Grid, F3DSF, and VDB [6,5,4,3] to files. CRS denotes the "compressed-row-storage" scheme employed by DT-Grid. "Deflate" is the codec used in GZIP, F3DSF, and optionally VDB. Bit mask refers to the compression scheme described in Appendix A. In the last column, bit mask and Deflate compression are combined with 32- to 16-bit float quantization. F3DSF at 16384^3 requires more than the available 48GB of RAM, so the write time could not be determined.

Table VII. Dilation Times for VDB and DT-Grid

Enright	Active Voxels	DT-Grid Optimal	[6,5,4,3] Optimal	[6,5,4,3] Brute-Force
512^3	794,720	0.03	<0.01 (?×)	0.05 (?×)
$1,024^3$	3,189,240	0.07	0.01 (7×)	0.18 (18×)
$2,048^3$	12,784,621	0.28	0.04 (7×)	0.73 (18×)
$4,096^3$	51,033,829	1.17	0.17 (7×)	2.90 (17×)
$8,192^3$	203,923,476	4.60	0.70 (7×)	11.61 (17×)
$16,384^3$	815,936,095	17.93	2.70 (7×)	46.53 (17×)

CPU times in seconds for a narrow-band dilation by one voxel of the standard Enright dataset [Enright et al. 2002] at various effective data resolutions. By "Optimal" we mean the fastest algorithm available, whereas "Brute-Force" refers to a naive approach using random access. The first VDB column lists the speedups over DT-Grid in parentheses. The last column lists the speedup of the optimal VDB algorithm over the brute-force approach in parentheses.

random insert to add new voxels. It is also worth emphasizing that the algorithm employed by DT-Grid is limited to closed narrow-band level sets, whereas the dilation algorithm of VDB has no such topology limitations.

5.6 Hierarchical Flood-Fill

Table VIII lists CPU times for hierarchical flood-fill of various VDB configurations encoding the standard Enright dataset in Table III. This algorithm is typically used to define the signs of voxels *outside* the narrow band of a level set (i.e., the inside/outside topology). The performance advantage of this hierarchical algorithm is especially evident when comparing the timings for [10,3], which is conceptually a tiled grid like F3DSF or DB-Grid, to any of the other VDB configurations with more levels. For low resolutions, the flood-fill of tall VDB configurations is orders of magnitude faster than the shallow [10,3]. This is because at low resolutions the computational bottleneck of [10,3] is the flood-filling of the large dense `RootNode`. Finally, recall that [10,3] has an available resolution of only $8,192^3$, as well as large memory footprints (Table IV).

5.7 Constructive Solid Geometry

To evaluate Constructive Solid Geometry (CSG) we need two intersecting level sets. Table IX compares CPU times for CSG operations between the Torus-Knot and the Stanford bunny, shown in Figure 8, again set up as in Enright et al. [2002]. We include results for both VDB and DT-Grid, but not F3DSF, since it lacks native implementations of CSG. Thanks to the hierarchical block partitioning,

Table VIII. Hierarchical Flood-Filling Times for VDB

Enright	Active Voxels	[7,6,5,4,3]	[6,5,4,3]	[6,5,3]	[10,3]
512^3	794,720	<0.01	<0.01	<0.01	1.87
$1,024^3$	3,189,240	0.01	0.01	0.01	1.93
$2,048^3$	12,784,621	0.06	0.06	0.05	1.99
$4,096^3$	51,033,829	0.22	0.22	0.21	2.17
$8,192^3$	203,923,476	0.88	0.87	0.85	2.85
$16,384^3$	815,936,095	3.45	3.45	3.38	N/A

CPU times in seconds for hierarchical sign flood-fill of a variety of VDB configurations encoding the standard Enright dataset [Enright et al. 2002] at various effective *data* resolutions.

Table IX. Comparing CSG for VDB and DT-Grid

[6,5,4,3]	DT-Grid	Union		Intersection		Difference	
CSG@ $2,048^3$/sec		<0.01	0.37	<0.01	0.24	<0.01	0.36
CSG@ $4,096^3$/sec		0.01	1.34	0.01	0.86	0.01	1.29
CSG@ $8,192^3$/sec		0.03	4.60	0.02	2.46	0.03	4.53

CPU times in seconds for (grid-aligned) CSG operations on the Stanford bunny and the Torus-Knot-Helix in Figure 8 using a DT-Grid and a VDB[6,5,4,3].

described in Section 4.3, which allows for fast node-based operations, VDB is several orders of magnitude faster than DT-Grid.

6. LIMITATIONS AND FUTURE WORK

While we have demonstrated several significant advantages of VDB, it is obviously not a silver bullet. Most notably, it is not as memory efficient for level set applications as DT-Grid. In Table II this difference is less than 1.5× for $[Hash, 5, 4, 3]$ and is a consequence of the fact that for most narrow-band level sets the nodes of VDB, especially `LeafNodes`, are only partially full of active voxels. In the future, we wish to explore two strategies to reduce the in-core memory footprint: in-core compression and improved out-of-core streaming.

The former amounts to runtime compression and decompression of the dense `LeafNodes`, and the challenge is not to degrade random access performance (as described in Appendix A, bit mask compression necessitates sequential access). The idea would be to efficiently prefetch, decompress, cache, and evict nodes based on the access pattern. We hypothesize that this could eliminate the memory overhead compared to DT-Grid, however the variable sizes of compressed nodes could lead to memory fragmentation over time, possibly necessitating a custom thread-safe memory pool.

The other strategy is to further develop out-of-core streaming. Currently, VDB supports deferred loading of voxel buffers, but this simple technique is limited to read-only operations. To allow for operations that modify both values and topology, we will need to develop a much more complex out-of-core data structure, like the one recently proposed in Christensen et al. [2011]. Interestingly, the cache manager for runtime compression could potentially be adapted for out-of-core compression as well.

Another limitation of VDB is that, despite the fact that it is a hierarchical data structure, it is probably not the best choice for multiresolution sampling, due to the large branching factors between levels of the tree. As such, a single VDB is not a general replacement for octrees when optimal adaptivity is desired. To a lesser extent the same is true for volumetric mipmaps, like the brickmaps of Christensen and Batali [2004], that store LOD representations in a single tree structure. Since tile and voxel values by

Fig. 10. Left: The "8x" variation of the Enright sphere [Enright et al. 2002], recently introduced in Christensen et al. [2011]. Where the original Enright sphere uses a single sphere of radius 0.15 placed at (0.35, 0.35, 0.35) in a normalized unit box, this "8x" version uses eight spheres with radius 0.125 positioned at (0.35, 0.35, 0.35), (0.15, 0.15, 0.85), (0.15, 0.85, 0.15), (0.35, 0.65, 0.65), (0.85, 0.15, 0.15), (0.65, 0.35, 0.65), (0.65, 0.65, 0.35), and (0.85, 0.85, 0.85). Results are shown for an effective data resolution of $4,096^3$ and a narrow-band width of 10, corresponding to 263,418,462 active voxels in the initial level set. Right: Result of divergence-free advection at half-period, where the active voxel count peaks at 1,096,281,344 voxels!

design cannot overlap in index space, mipmapping is currently not supported within a single VDB tree. However, preliminary work has demonstrated that multiple VDB trees, each representing one component in a LOD sequence of shader attributes, is an efficient alternative to a traditional brickmap. Simularly we are optimistic about a multi-VDB strategy for an elliptic multigrid solver.

7. CONCLUSION

We have presented a novel data structure and algorithms, collectively dubbed VDB, that offer several advantages over existing state-of-the-art sparse data structures for *dynamic* volumes. VDB builds on B+trees and is analogous in several respects to the proven design principles of memory management and cache hierarchies in modern CPUs. Specifically, the VDB data structure consists of a large or dynamically resizable root node, followed by a fixed number of intermediate node levels with decreasing branching factors and small leaf nodes. Random access is significantly accelerated by inverted tree traversal of cached nodes, which in turn is facilitated by the fixed tree depth and branching factors. The main benefits of VDB are fast and flexible random access, unbounded signed index domains, support for arbitrary grid topology, small memory and disk footprints, fast write times, adaptive sampling, hierarchical CSG and flood-filling, efficient topological dilation and erosion, a native acceleration structure, and fast sequential stencil access for numerical simulations.

While we have benchmarked VDB against several existing data structures, our performance comparisons focused on DT-Grid, since it is the fastest and most compact existing data structure that supports operations like sparse numerical simulations, CSG, and efficient topology dilation. Our evaluations suggest that VDB can easily be configured to outperform DT-Grid in all of these operations, and it offers much faster random access. Equally important is the fact that VDB can be used on virtually any volumetric data, not just level sets, and it is more flexible since it also supports fast random insertion and deletion. However, DT-Grid has a smaller memory footprint. In essence, VDB trades some memory ($<2\times$) for additional flexibility (e.g., random insertion/deletion and adaptive sampling) as well as significant improvements in computational efficiency ($4-35\times$ for advection, $>10\times$ for spatially coherent lookup, $3-4\times$ write performance, $7\times$ morphological dilation, and $>100\times$ for CSG).

VDB is currently being applied to a large body of simulation, modeling, and rendering problems at *DreamWorks Animation*, including: cloud modeling, seamless volumetric fracturing, scan conversion, adaptive mesh extraction, implicit surface fairing, morphing, 3D compositing and CSG, brickmaps, elliptic pressure solvers, fluid simulations, level set methods, GPU rendering of isosurfaces and volumes for interactive previsualization, multiscattered volume rendering, volume-based hair growth and grooming, particle advection, and collision detection. It is our hope that the open, source release of VDB [OpenVDB 2012] will find as much use outside of *DreamWorks Animation*.

APPENDIXES

A. OUT-OF-CORE COMPRESSION

The `LeafNodes` described so far are dense, in the sense that each temporal buffer in the `mLeafDAT` contains exactly `sSize =1<<Log2X+Log2Y+Log2Z` voxel values. However, when the volume is sparse, many `LeafNode` voxels may be inactive and set to a fixed background value that need not be stored per voxel. For narrow band level sets, the memory overhead of storing these redundant values can vary significantly with the width of the narrow-band, the topology of the level set surface, and the `LeafNode`'s size. In general it is desirable to keep the `LeafNodes` small, but a simple way to reduce the footprint of partially full `LeafNodes` is to eliminate the inactive voxels altogether, provided that they are restricted to a small number of discrete values. This might not be practical for all applications, though.

While we can readily design an encoder that removes inactive voxel values from `mLeafDAT`, using the set bits in `mValueMask`, the

same is not true for the decoder. For the simple case of a unique background value associated with all inactive voxels, for example, zero for density fields, we can trivially reconstruct the missing inactive voxel values. However, for narrow band level sets there are typically two values for the inactive voxels: a negative value for voxels inside the narrow-band and a positive value of equal magnitude for voxels outside the band. In many cases, the level set surface is physically realizable, that is, it is a closed manifold, in which case the proper signs for inactive voxels can be derived efficiently via the hierarchical flood-fill algorithm discussed in Section 4.6. For the general case of nonclosed level sets, we can employ the optional bit mask, `mInsideMask` (see line 11 of the code) to compactly encode the inside/outside topology.

While we can sequentially access active voxels in compressed `LeafNodes`, the same is not true for stencil or random access. For those access patterns, `LeafNodes` typically need to first be decompressed, though decompression can be incorporated conveniently into the various caching schemes. Another more subtle issue with runtime compression is memory fragmentation due to the changing sizes of the numerous `mLeafDAT`s. One way to address this problem is to allocate `mLeafDAT`s from a memory pool, which also improves performance by amortizing the cost of memory allocations. However, it is generally complicated to develop efficient and thread-safe memory pools when allocating data of nonfixed size, such as compressed `mLeafDAT`s. For this reason and because any runtime decompression would slow down nonsequential access, we use this technique only for out-of-core compression during disk I/O (see Section 5.4).

So far, we have only discussed the special cases in which there are either one or two inactive voxel values, corresponding to the background or, for level sets, the exterior or interior. For the less common case of an arbitrary number of inactive voxel values, we have to resort to more generic compression techniques, like lossless RLE and Zip, or lossy bit quantization, applied directly to the `mLeafDAT`. Obviously, these compression schemes can also be applied to the special cases discussed earlier, but as shown in Section 5.4, they can be orders of magnitude slower than the bit-mask-based codecs.

Finally, it is easy to add support for out-of-core streaming by encoding file offsets into the `LeafNodes`, as in line 7 of the code. This allows us to delay the loading of `mLeafDAT`s until the voxel values, rather than just the grid topology, are actually needed, significantly reducing the in-memory footprint. This deferred loading is especially useful when rendering very large volumes that are culled by a camera frustum or dominated by early ray termination.

In summary, `LeafNodes` can exist in several states, including bit-mask- and/or Zip-compressed, bit quantized, or even out-of-core. These different states are compactly encoded in the `mFlags` (see Figure 5 and line 12 of the code) and can even vary among `LeafNodes` belonging to the same tree.

B. MULTITHREADING AND VECTORIZATION

Performance on modern CPUs is influenced predominantly by three factors: utilization of the cache memory system, the ability to distribute computation to multiple cores, and the degree of saturation of the vector pipeline. A good data structure should be optimized for all of these factors.

Cache efficiency of VDB is achieved through data locality. Since most algorithms can be written to process whole `LeafNodes` at a time, we process data more or less in the order they are stored in main memory, which limits cache misses as long as the size of a `LeafNode` doesn't exceed the cache size of the CPU.

Whereas caching exploits memory coherency, vectorization takes advantage of execution coherency. We can utilize the extended SIMD instruction sets found in modern processors (e.g., SSE on x86 hardware) when performing computation on both the voxel coordinates and values. For example, lines 39–55 of the code involve similar operations on the x, y, and z components of the global coordinates. This means a runtime gain can be achieved by executing those operations in parallel via a small number of SIMD instructions. The same is true for many of the floating point operations used to update voxel values during numerical simulations, such as the finite difference computations described in Section 4.2.

Finally, multithreading is essential to exploit the full potential of modern CPUs. Many algorithms, like the sequential finite-difference schemes we use to solve level set equations, are trivial to parallelize since they only modify voxel values, not grid topology. The strategy is simply to distribute `LeafNodes` over multiple computational threads, for instance by means of `tbb::parallel_for` in Intel's open-source Threading Building Blocks library. The problem is slightly more complicated when dealing with algorithms that modify grid topology, for example, when creating a new VDB from other geometric representations like polygons or particles. In short, the issue is that multiple threads might attempt to allocate the same node or even modify the same byte in a bit mask, so random insertion and deletion are not thread safe in VDB[10]. Again the solution is simple and efficient: assign a separate grid to each computational thread, then merge grids as threads terminate. (We implement this type of threading with dynamic topolgy by means of `tbb::parallel_reduce`.) For level set applications like mesh scan conversion and particle rasterization the merge operation is typically a CSG union, which, as was discussed in Section 4.3, can be implemented very efficiently since it operates directly on the branches of a VDB tree. The memory overhead of this threading strategy is also negligible as long as the spatial domains over which the threads operate have little or no overlap. In other words, the worst-case scenario would be if all threads performed computations over the entire grid domain. This might happen if one were to rasterize a very high density of particles with a uniform random spatial distribution, but this is rarely encountered in production and can always be addressed by a fast bucketing of the particles into the `LeafNodes`, leading to optimal spatial partitioning. In practice, we often omit this bucketing step, yet we have still observed near-optimal scaling in terms of memory and compute time.

All the algorithms discussed in this article are multithreaded using one of these two simple techniques. The only exception is the fast dilation algorithm in Section 4.1, which is memory-bound rather than CPU-bound due to the fast bit operations and the slower allocation/deallocation of nodes.

ACKNOWLEDGMENTS

Foremost the author wishes to express deep gratitude to DreamWorks Animation for supporting the development and release of VDB as an open-source project, an effort spearheaded by Ron Henderson, Andrew Pearce, and Lincoln Wallen. Major thanks also go to Peter Cucka and David Hill for proofreading early versions of the manuscript, Mihai Alden and Katrine Museth for help with some of the figures, and the anonymous reviewers for their insightful advice. Finally, the author wishes to dedicate this article to Hanne Museth.

[10]Note that many sparse data structures don't even support random insert or deletion, and the few that do are not thread safe. The reason is simply that sparseness is achieved by "allocation-on-insert", which is inherently not thread safe!

REFERENCES

BAYER, R. AND MCCREIGHT, E. M. 1972. Organization and maintenance of large ordered indices. *Acta Informatica 1*, 173–189.

BREEN, D., FEDKIW, R., MUSETH, K., OSHER, S., SAPIRO, G., AND WHITAKER, R. 2004. Level set and pde methods for computer graphics. In *ACM SIGGRAPH Course Notes*. ACM Press, New York.

BREEN, D. E., MAUCH, S., AND WHITAKER, R.T. 1988. 3D scan conversion of csg models into distance volumes. In *Proceedings of the IEEE Symposium on Volume Visualization*. 7–14.

BRIDSON, R. 2003. Computational aspects of dynamic surfaces. Ph.D. thesis, Stanford University.

BRUN, E., GUITTET, A., AND GIBOU, F. 2012. A local level-set method using a hash table data structure. *J. Comput. Phys. 231*, 6, 2528–2536.

CHANG, B., CHA, D., AND IHM, I. 2008. Computing local signed distance fields for large polygonal models. *Comput. Graph. Forum 27*, 3, 799–806.

CHRISTENSEN, B., NIELSEN, M., AND MUSETH, K. 2011. Out-of-core computations of high-resolution level sets by means of code transformation. *J. Sci. Comput. 50*, 2, 1–37.

CHRISTENSEN, P. H. AND BATALI, D. 2004. An irradiance atlas for global illumination in complex production scenes. In *Proceedings of the Eurographics Symposium on Rendering Techniques*. Eurographics/ACM Press, 133–141.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM Press, New York, 15–22.

DT-GRID. 2009. Version 0.92. http://code.google.com/p/dt-grid.

ENRIGHT, D., FEDKIW, R., FERZIGER, J., AND MITCHELL, I. 2002. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys. 183*, 1, 83–116.

EYIYUREKLI, M. AND BREEN, D. E. 2011. Data structures for interactive high resolution level-set surface editing. In *Proceedings of the Conference on Graphics Interface*. 95–102.

FIELD3D. 2009. Version 1.2.0. https://sites.google.com/site/field3d.

FRISKEN, S. F. AND PERRY, R. 2002. Simple and efficient traversal methods for quadtrees and octrees. *J. Graph. GPU Game Tools 7*, 3, 1–11.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th Annual ACM SIGGRAPH Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison Wesley, New York. 249–254.

HDFS. 2010. Version 1.8.4. http://www.hdfgroup.org/HDFS.

HOUSTON, B., NIELSEN, M. B., BATTY, C., NILSSON, O., AND MUSETH, K. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph. 25*, 151–175.

JU, T., LOSASSO, F., SCHAEFER, S., AND WARREN, J. 2002. Dual contouring of hermite data. In *Proceedings of the 29th Annual ACM SIGGRAPH Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, 339–346.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2005. Texture sprites: Texture elements splatted on surfaces. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM Press, New York, 163–170.

LEFOHN, A. E., KNISS, J. M., HANSEN, C. D., AND WHITAKER, R. T. 2003. Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proceedings of the 14th IEEE Visualization Conference*. IEEE Computer Society, 75–82.

LEISERSON, C. E., PROKOP, H., AND RANDALL, K. H. 1998. Using de bruijn sequences to index a 1 in a computer word. http://supertech.csail.mit.edu/papers/debruijn.pdf.

LEVEQUE, R. J. 1996. High-resolution conservative algorithms for advection in incompressible flow. *SIAM J. Numer. Anal. 33*, 2, 627–665.

LIU, X., OSHER, S., AND CHAN, T. 1994. Weighted essentially nonoscillatory schemes. *J. Comput. Phys. 115*, 200–212.

LOSASSO, F., GIBOU, F., AND FEDKIW, R. 2004. Simulating water and smoke with an octree data structure. *ACM Trans. Graph. 23*, 457–462.

MAUCH, S. 1999. Stlib. https://bitbucket.org/seanmauch/stlib/wiki/Home.

MILLER, B., MUSETH, K., PENNY, D., AND ZAFAR, N. B. 2012. Cloud modeling and rendering for "Puss in Boots". In *ACM SIGGRAPH Talks*. ACM Press, New York, 5:1.

MIN, C. 2004. Local level set method in high dimension and codimension. *J. Comput. Phys. 200*, 368–382.

MUSETH, K. 2009. An efficient level set toolkit for visual effects. In *ACM SIGGRAPH Talks*. ACM Press, New York, 5:1.

MUSETH, K. 2011. DB+Grid: A novel dynamic blocked grid for sparse high-resolution volumes and level sets. In *ACM SIGGRAPH Talks*. ACM Press, New York.

MUSETH, K., BREEN, D., WHITAKER, R., MAUCH, S., AND JOHNSON, D. 2005. Algorithms for interactive editing of level set models. *Comput. Graph. Forum 24*, 4, 821–841.

MUSETH, K. AND CLIVE, M. 2008. CrackTastic: Fast 3D fragmentation in "The Mummy: Tomb of the Dragon Emperor". In *ACM SIGGRAPH Talks*. ACM Press, New York, 61:1.

MUSETH, K., CLIVE, M., AND ZAFAR, N. B. 2007. Blobtacular: Surfacing particle system in "Pirates of the Caribbean 3". In *ACM SIGGRAPH Sketches*. ACM Press, New York.

NIELSEN, M. B. 2006. Efficient and high resolution level set simulations. Ph.D. thesis, Aarhus University.

NIELSEN, M. B. AND MUSETH, K. 2006. Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *J. Sci. Comput. 26*, 261–299.

OHTAKE, Y., BELYAEV, A., ALEXA, M., TURK, G., AND SEIDEL, H.-P. 2003. Multi-level partition of unity implicits. *ACM Trans. Graph. 22*, 3, 463–470.

OPENVDB. 2012. August 3. http://www.openvdb.org.

OSHER, S. AND FEDKIW, R. 2002. *Level Set Methods and Dynamic Implicit Surfaces*. Springer.

SHU, C. AND OSHER, S. 1988. Efficient implementation of essentially non-oscillatory shock capturing schemes. *J. Comput. Phys. 77*, 439–471.

SPARSEHASH. 2009. Version 1.12. http://goog-sparsehash.sourceforge.net.

STOLTE, N. AND KAUFMAN, A. 1998. Parallel spatial enumeration of implicit surfaces using interval arithmetic for octree generation and its direct visualization. In *Proceedings of the 3rd International Workshop on Implicit Surfaces*. 81–87.

STRAIN, J. 1999. Tree methods for moving interfaces. *J. Comput. Phys. 151*, 2, 616–648.

TESCHNER, M., HEIDELBERGER, B., MUELLER, M., POMERANETS, D., AND GROSS, M. 2003. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of the Conference on Vision, Modeling, and Visualization*. 47–54.

VEENSTRA, J. AND AHUJA, N. 1988. Line drawings of octree-represented objects. *ACM Trans. Graph. 7*, 1, 61–75.

WILLIAMS, R. D. 1992. Voxel databases: A paradigm for parallelism with spatial structure. *Concurr. Pract. Exper. 4*, 8, 619–636.

ZAFAR, N. B., STEPHENS, D., LARSSON, M., SAKAGUCHI, R., CLIVE, M., SAMPATH, R., MUSETH, K., BLAKEY, D., GAZDIK, B., AND THOMAS, R. 2010. Destroying la for "2012". In *ACM SIGGRAPH Talks*. ACM Press, New York, 25:1.

Received January 2012; revised December 2010; accepted January 2013