Journal of Scientific Computing (© 2006)
DOI: 10.1007/s10915-005-9062-8

Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets

Michael B. Nielsen[1] **and Ken Museth**[2]

Received November 12, 2004; accepted (in revised form) January 26, 2005

Level set methods [Osher and Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton–Jacobi formulations. *J. Comput. Phys.* 79 (1988) 12] have proved very successful for interface tracking in many different areas of computational science. However, current level set methods are limited by a poor balance between computational efficiency and storage requirements. Tree-based methods have relatively slow access times, whereas narrow band schemes lead to very large memory footprints for high resolution interfaces. In this paper we present a level set scheme for which *both* computational complexity and storage requirements scale with the size of the interface. Our novel level set data structure and algorithms are fast, cache efficient and allow for a very low memory footprint when representing high resolution level sets. We use a time-dependent and interface adapting grid dubbed the "Dynamic Tubular Grid" or DT-Grid. Additionally, it has been optimized for advanced finite difference schemes currently employed in accurate level set computations. As a key feature of the DT-Grid, the associated interface propagations are not limited to any computational box and can expand freely. We present several numerical evaluations, including a level set simulation on a grid with an effective resolution of 1024^3.

KEY WORDS: Interface tracking; level sets; compact model representations; deformable surfaces; partial differential equations.

1. INTRODUCTION

This paper presents an efficient data structure and algorithms for the tracking of propagating interfaces in various computer simulations. Interface tracking is an important problem in many different scientific fields

[1] University of Århus, Arhus, Denmark.
[2] Department of Science and Technology, Linköping Institute of Technology, 601 74 Norrköping, Sweden. E-mail: museth@acm.org

ranging from physics and chemistry to computer vision and computer graphics. To mention a few: two-phase tracking in computational fluid dynamics (e.g. water/air), tracking of the blue-core in simulations of chemical combustion in burning flames, propagation of curves in image segmentation and deforming surfaces in geometric modeling.

All of these fields have found the level set method [23, 25, 32] by Osher and Sethian to be very useful and robust [5, 9, 10, 18, 22 24, 28, 35, 41]. This PDE based method represents the dynamic interface implicitly as the zero level set (i.e. iso-surface) of a time-dependent signed distance function discretized on a computational grid. The level set method essentially adds a spatial dimension to address the problem of efficiently representing and tracking the interface. This allows level set surfaces to undergo arbitrary changes in topology while at the same time preventing them from self-intersecting, which is very hard to avoid with explicit surface representations.

However, the robustness and flexibility of the level set method comes at the price of having to solve a time-dependent PDE on a discrete grid of dimension one higher than that of the actual interface. The related computational issue is effectively addressed by so-called narrow band schemes that utilize the fact that it is sufficient to solve the level set PDE in the vicinity of the interface in order to track it. Consequently, narrow band implementations of the level set method have the desired property that the computational complexity scales with the size of the interface (i.e. area in 3D and arc-length in 2D) rather than with the volume (or area) in which it is embedded. However, all current publications on these narrow band schemes either require the full computational grid to be stored or the narrow band grid points to be stored in a hierarchical tree structure. This leads to either very large memory footprints or complicated tree data structures with relatively slow access and construction times. Furthermore, these narrow band methods are limited by a convex boundary of the underlying computational grid, which typically restricts the extent of interface expansion to a predefined box. The main contribution of this paper is to present a novel efficient data structure and algorithms that are not hampered by these limitations.

Our general approach to addressing these limitations is to introduce a dynamic uniform grid that is only defined in a tubular region around the propagating interface. So, in contrast to existing narrow band methods we do not store *any* information outside of this dynamic tube. Also, since we apply a uniform sampling around the interface we are not hampered by some of the limitations related to multi-resolution techniques such as the use of hierarchical trees. Our data structure can readily be used with all of the finite difference schemes developed for uniform full grids. Moreover,

Lipschitz discontinuities from interpolation over non-uniform grid cells is not an issue. In fact, our studies show that our 3D DT-Grid is faster and more memory efficient than a state-of-the-art octree implementation [12, 33]. Finally, our data structure is free of any boundary restrictions on the interface expansion which leads to what we call "out-of-the-box" level set simulations. Throughout this paper we shall refer to our approach as the Dynamic Tubular Grid, or simply DT-Grid.

1.1. Previous Work

The celebrated level set method was proposed by Osher and Sethian in 1988 [25]. In its original formulation a discretized function of the distance to an initial interface is propagated by solving a time-dependent PDE on a full Cartesian grid by means of sophisticated finite difference schemes developed for Hamilton–Jacobi equations. While an incredibly elegant solution to the challenging problem of interface tracking, it had the disadvantage that the associated computational complexity scaled with the size of the grid in which the interface was embedded, rather than with the actual size of the interface itself.

This limitation was removed by the introduction of so-called narrow band schemes that simply solve the level set PDE in close vicinity of the zero level set, i.e. the interface. This idea was first proposed by Adalsteinsson and Sethian [1], but the numerical implementation was based on a very wide band around the interface to avoid frequent and costly rebuilds. Later Whitaker [39] proposed an approximate but faster narrow band scheme dubbed the "Sparse Field Method". In addition to storing the full grid, the actual interface grid points were arranged in linked lists, and the level set PDE only solved at these grid points. This solution was then propagated to neighboring grid points by means of an approximate city-block distance metric. The width of the resulting narrow band was only as wide as the size of the finite difference stencils used on the interface grid points. Additionally, costly re-initializations were avoided by employing speed function extension [2] that preserves the approximate signed distance to the interface. Finally, Peng *et al.* [27] proposed a fast narrow band scheme that accurately solves the level set PDE in a narrow band around the interface. Subsequently, this solution is propagated out to an extra band by means of an Euclidian distance metric. Their method employs data structures based on simple arrays as opposed to the linked lists used in [39].

All of these narrow band schemes effectively address the problem of computational complexity present in the original level set formulation [25]. However, they all explicitly store a full cartesian grid and additional data

structures to identify the narrow band grid points. Hence, the associated memory requirements scale with the size of the full grid, as opposed to the size of the interface. This can be a severely limiting factor for level set simulations that require large grids to resolve details of complex interfaces or large deformations over time. To the best of our knowledge, the only published previous work that attempts to address this serious problem is a relatively small body of work based on tree structures as the underlying grid representation. For contours this typically amounts to using quadtrees [6, 21, 39] and more recently this idea was extended to surfaces by means of octrees [16]. Furthermore, Milne [20] and Sussman *et al.* [29] have explored level sets in conjunction with patch based AMR.

Tree based approaches do indeed reduce the memory footprint of the associated interface representation. However, a problem seems to be that the data structures are relatively slow to access and modify during interface propagations. This can lead to significant reductions in performance when compared to regular narrow band schemes based on full uniform Cartesian grids, see the evaluation in Sec. 4 and e.g., [4]. However, a reviewer of this paper has pointed out to us that work in submission by Lossasso *et al.* [11] proposes using a uniform grid where every cell is an octree of its own. This should remove much of the slow access and make it easy to extend the grid in space by simply adding uniform grid cells without the need to modify the octrees in other cells or their depth structure. However, we lack implementation details to test these improvements against DT-Grid.

While tree data structures allow for multi-resolution representations, all practical tree methods appear to use uniform resolution near the interface [6, 16, 21, 39]. This is partly due to the fact that it is hard to design reliable "refinement oracles" which can guarantee that no fine features are missed due to under-sampling as the interface propagates in time. Refining uniformly near the interface and storing only these grid points in the octree (as we have done for evaluation purposes in Sec. 4) enables using the standard higher order finite difference schemes like ENO [26] or WENO [17] in space and the TVD Runge–Kutta methods [34] in time. However, a non-uniform discretization makes it non-trivial to accurately employ these finite difference schemes. Tree based methods often use a semi-Lagrangian scheme [36] which is strictly limited to hyperbolic problems like advection in external velocity fields. While very efficient for problems typically encountered in CFD, it is unclear how to extend this approach to parabolic problems like curvature based interface flow. Additionally, since the semi-Lagrangian method uses interpolation on the non-uniform grid, nontrivial issues like Lipschitz continuity also has to be explicitly addressed [21]. However, as demonstrated by Losasso *et al.* [16] a simple adaptive interpolation scheme,

like the one described in [40], works fine when high accuracy is not a major concern.

To the best of our knowledge, the only work directly related to ours is the work by Bridson [4] and Houston *et al.* [14]. Bridson [4] suggests to store the level set in a sparse block grid which is a coarse uniform grid with finer uniform grids nested in the coarse grid cells that intersect the interface. This approach allows for higher resolutions, but the memory usage is not proportional to the size of interface. Bridson also suggests a solution based on a hashtable to allow the grid to expand, but does not demonstrate it. The method of Houston *et al.* [14] was described in a technical sketch (one page abstract) recently presented at a graphics conference, where we concurrently summarized the main features of DT-Grid [3]. Their work primarily focuses on fluid simulations, and they propose a data structure based on Run-Length-Encoding which decouples the storage of the elements from the actual encoding. While we do not have enough details to reproduce their method for evaluation, we list the following characteristics based on the abstract and private communication. In 3D, their approach requires $O(M_X M_Y + M_3)$ storage, where M_X, M_Y and M_3 are the number of grid points in, respectively, the X and Y dimensions of a bounding box and the narrow band. Hence their memory usage is not proportional to the interface. Sequential access time is $O(\frac{M_X M_Y}{M_3} + 1)$ whereas random and stencil access times are logarithmic in the number of runs in each scan-line. Their method maintains a dynamically resizing bounding box which allows the level set to grow dynamically. However, if the $M_X M_Y$ dependency in their storage requirements becomes dominant, their method does not allow for out-of-the-box level set simulations. The characteristics of our method, DT-Grid, will be outlined in the next section.

1.2. Contributions

Our work stands apart from previously published work in several ways. We do not use any tree structures or full grids with additional data structures to represent the narrow band. DT-Grid takes an entirely different approach by storing the narrow band in a very compact non-hierarchical data structure that uses less memory than previous methods without compromising the computational efficiency. Below we summarize our contributions.

- The memory usage of DT-Grid is proportional to the size of the interface. More specifically the storage requirements are $O(M_N)$ (in 3D $O(M_3)$) where M_N is the number of grid points in the

N-dimensional narrow band. In fact our evaluations show DT-Grid to be more compact than other grid or tree-based level set schemes that employ a uniform sampling of the interface. As a result, our data structure allows for higher resolutions of level sets before hardware memory restrictions are potentially violated.

- All our evaluations have shown that the computational efficiency of accurate level set deformations based on DT-Grid is better than both narrow band and tree-based approaches. We strongly believe this is due to the added cache coherency from the dramatically reduced memory footprints. More specifically we have developed efficient algorithms that guarantee the following properties of DT-Grid:

    1. Access to grid points has time complexity $O(1)$, when the grid is accessed sequentially.
    2. Access to neighboring grid points within finite difference stencils has time complexity $O(1)$.
    3. The time complexity of random and neighbor access to grid points outside the finite difference stencil is logarithmic in the number of connected components within p-columns (see Sec. 2.1 for the definition).
    4. The time complexity of constructing and rebuilding the DT-Grid is linear in the number of grid points in the narrow band.

- Our data structure allows level set interfaces to freely deform without boundary restrictions imposed by underlying grids or trees employed in other methods. This effectively implies that interfaces can expand arbitrarily. We demonstrate this with *out-of-the-box* level set deformations.
- Our compact data structure generalizes and scales well to any number of dimensions.
- Unlike approaches employing non-uniform grids our flexible data structure can transparently be integrated with all existing finite difference schemes typically used to numerically solve both hyperbolic and parabolic level set equations on uniform dense grids.

This paper is organized as follows. Section 2 introduces the DT-Grid data structure. A general N-dimensional definition is given and a detailed explanation is presented in 2D. In Sec. 3, we describe efficient algorithms that are fundamental to improved level set simulations on the DT-grid. Many of the details of these algorithms are given in appendices. We realize that not all readers are interested in these details, and as such effort has

been made to make the main part of this paper self-contained even if the appendices are skipped. In Sec. 4, we evaluate the time and memory efficiency of the DT-Grid compared to previous methods. We also demonstrate the low memory footprint in a 1024^3 high resolution level set simulation. We then show how level set simulations on a DT-Grid can go out-of-the-box, a feature not shared with any existing narrow band or tree-based level set method. Finally, Sec. 5 concludes the paper and outlines future work.

2. DATA STRUCTURE

Throughout this paper by tubular grid we mean a subset of grid points, defined on an infinite grid, within a fixed distance from an interface. As the interface propagates this subset changes, thus giving rise to the term dynamic tubular grid. In this section we define the DT-Grid, an efficient data structure for N-dimensional dynamic tubular grids.

A straightforward non-hierarchical approach to representing a tubular grid is to explicitly store float values and indices of all its grid points. To obtain constant access times to neighboring grid points, one could also store additional pointers. However, this approach does not scale well as the number of grid points in the tubular grid increases. The DT-Grid employs a better approach by combining a compressed index storage scheme with knowledge of the connectivity properties of the tubular grid to obtain a memory and time efficient data structure. This is achievable by means of a *lexicographical* storage order of the grid points.

2.1. Definition of the DT-Grid

The DT-Grid is defined recursively in terms of DT-Grids of lower dimensionality, and as such our approach readily generalizes to any dimension. However, for simplicity we shall limit a detailed description of the data structure to 2D and illustrate with the example depicted in Fig. 1.

As a prelude to this description it is convenient to introduce the following general terminology based on the nomenclature given in Table I.

- In N-dimensions, *p-column* (short for projection column) number $\mathbf{X}_{N-1} = (x_1, x_2, \ldots, x_{N-1})$ is defined as the set of grid points in the tubular grid that project to $(x_1, x_2, \ldots, x_{N-1}, 0)$ by orthogonal projection onto the subspace spanned by the first $N-1$ coordinate directions. Thus a p-column is always 1D.

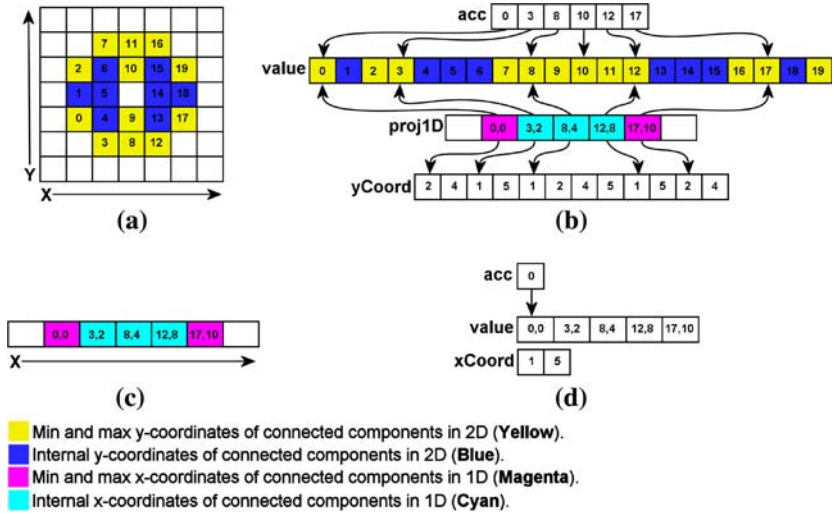


Fig. 1. (a) A dense 2D grid. (b) Corresponding 2D DT-Grid. (c) A dense 1D grid. (d) Corresponding 1D DT-Grid. Note the lexicographic storage order.

Table I. Nomenclature Used Throughout This Paper

N	The Dimension.
$\mathbf{X}_N$	Grid point or p-column number $(x_1, x_2, \ldots, x_N)$
$\phi(\mathbf{X}_N)$	Scalar level set function.
Ω^-	Interior region.
Ω^+	Exterior region.
dx	The uniform grid spacing.
T_α	The tubular grid $\{\mathbf{X}_N \in \Re^N \mid \lvert\phi(\mathbf{X}_N)\rvert < \alpha\}$.
M_N	Number of grid points in the ND tubular grid.
γ	Width of the tubular grid.
$C_{\mathbf{X}_N}$	Number of connected components in p-column $\mathbf{X}_N$.
C_N	Total number of connected components in ND DT-Grid.

- A *connected component* in N-dimensions is defined as a maximal set of adjacent grid points within a p-column.

For example, in 2D, p-column number x is defined as the set of grid points in the tubular grid that project to $(x, 0)$ by orthogonal projection onto the X axis. In Fig. 1a, p-column number 3 is defined as the set of grid points $\{(3,1),(3,2),(3,4),(3,5)\}$, and it contains two connected components, $\{(3,1),(3,2)\}$ and $\{(3,4),(3,5)\}$. Note that the lower leftmost grid point in Fig. 1a is $(0,0)$. A N-dimensional DT-Grid can be defined recursively in terms of a $(N-1)$-dimensional DT-Grid using pseudo C++ syntax as follows

```
template<typename Type> class DTGridND<Type> {
    Array1D<Type> value;
    Array1D<Index> nCoord;
    Array1D<unsigned int> acc;
    DTGrid(N-1)D<IndexPair> proj(N-1)D;
}
```

Below we define the 2D DT-Grid in pseudo C++ syntax and explain its constituents in detail.

```
template<typename Type> class DTGrid2D<Type> {
    Array1D<Type>  value;
    Array1D<Index> yCoord;
    Array1D<unsigned int> acc;
    DTGrid1D<IndexPair> proj1D;
}

template<typename Type> class DTGrid1D<Type> {
    Array1D<Type> value;
    Array1D<Index> xCoord;
    Array1D<unsigned int> acc;
}
```

value: The `value` array (in `DTGrid2D`) stores the numerical values of all grid points in the two-dimensional tubular grid in (x, y) *lexicographic* order. Typically the associated `Type` will be `float` or `double`. In Fig. 1{a,b} the grid points contained in the tubular grid are colored yellow and blue. In this illustrative example the numerical values of the grid points in the tubular grid are simply chosen to be the corresponding lexicographic storage order in the DT-Grid.

yCoord: The `yCoord` array stores the min and max y-coordinate of each connected component. In Fig. 1{a,b} these grid points are shown in yellow. Thus, rather than simply storing y-coordinates of all grid points, we exploit the connectivity in the tubular grid.

acc: The `acc` array (in `DTGrid2D`) stores pointers into the `value` array which identifies the first tubular grid point in each connected component. As will be explained later this information is essential for obtaining a fast random access operation.

proj1D: The `proj1D` constituent holds pairs of indices into the `value` and `yCoord` arrays, for the first grid point in each p-column in the tubular grid. This is illustrated with arrows in Fig. 1b. Also note that `proj1D` is defined recursively as a `DTGrid1D` with `Type=IndexPair`, see Fig. 1{c,d}. The constituents of the 1D DT-Grid are defined similarly, except for the fact that is does not have a `proj0D` constituent. `proj1D` introduces additional structure into the 2D DT-Grid and allows for fast access to each p-column independently. As will be explained in the next section, this structure is used extensively in most of the algorithms of the data structure. Note that it is possible to reduce the storage, at the tradeoff of an extra array lookup, by only storing the index into the `yCoord` (in

general `nCoord`) array. The corresponding index into the `value` array can then be obtained by looking it up in the `acc` array using the index into the `yCoord` array divided by two.

Figure 2 shows an example of a sphere represented in a 3D DT-Grid. The 2D and 1D DT-Grid constituents are also included in the illustrations. The red and green grid points are the grid points in the 3D tubular grid. The red grid points are the start and end grid points of connected components. Fig. 2c shows p-column number $(25, 25)$ consisting of two connected components. The white pixel in Fig. 2c illustrates the `IndexPair` that points to p-column number $(25, 25)$.

The storage requirements of a N-dimensional DT-Grid are $O(M_N)$ (see Table I) which can be justified as follows: Clearly, the storage requirements of a 1D DT-Grid are $O(M_1)$ since it does not contain a `proj0D` constituent. The storage requirements of a N-dimensional DT-Grid are $O(M_{N-1} + M_N)$ which by induction equals $O(M_N)$.

One additional and important property of the DT-Grid can be deduced from the definition given above. Since the DT-Grid is defined recursively, the coordinate vectors of all grid points are explicitly stored, albeit in a compressed format. This means that the grid points of the DT-Grid are not restricted to a particular range of indices as is the case with the traditional full grid or tree-based methods. Hence, the DT-Grid is in fact capable of representing unbounded, dynamically expanding and non-convex grids. This allows for truly out-of-the-box level set simulations which we demonstrate in Sec. 4.3.

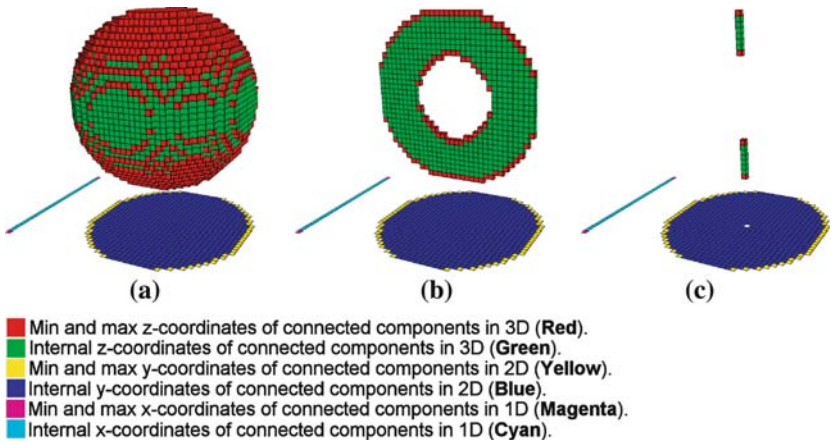


Fig. 2. Color coded representation of the tubular grid of a sphere in a 3D DT-Grid. (a) Entire sphere. (b) Middle slice of sphere. (c) P-column consisting of two connected components.

3. ALGORITHMS OF THE DATA STRUCTURE

In this section, we describe in detail the key algorithms of our memory-, cache- and time-efficient DT-Grid data structure. The DT-Grid has the exact same algorithmic interface as a full grid. Furthermore, even though our data structure only stores the values of a tubular grid, methods that provide access to *any* grid point are supported. In our case these methods simply return a signed value, positive in Ω^+ and negative in Ω^-, with absolute value equal to the width of the tubular grid. This design approach hides the added complexity of our improved data structure and makes it almost trivial to integrate DT-Grid with existing level set simulation code.

Due to the recursive nature of the storage format of the DT-Grid, many of the operations presented here are also recursive in nature. The rest of this section is structured as follows. Section 3.1 describes a constant time operation for inserting grid points into the DT-Grid. In Secs. 3.2 and 3.3 we describe how constant time sequential access to all grid points within a finite difference stencil can be obtained when iterating over the grid. This is essential in obtaining a fast data structure. Section 3.4 describes a logarithmic time algorithm for random access to grid points based on binary search. This algorithm is used if grid points are accessed non-sequentially or lie outside of the stencil. As will be justified in the evaluation section, this random access algorithm, albeit asymptotically logarithmic, has proven to be almost as fast as random access in a full grid, due to cache coherency. Next, Sec. 3.5 describes how constant and logarithmic time neighbor access operations can be constructed. Finally, Sec. 3.6 describes how the tubular grid is rebuilt. In particular, we describe a generic algorithm for rebuilding the tubular grid, which can be used independently of the method employed for re-initializing the level set function to a signed distance function. Table II gives an overview of the operations and their associated time complexities.

3.1. Push – Inserting Grid Points in Constant Time

The DT-Grid supports a low-level constant time `push` operation to add new grid points to the data structure. Since the grid points are stored in memory lexicographically as $(x_1, x_2, \ldots, x_N)$, new grid points must be pushed in this order.[1] If, on the other hand, grid points were inserted in an order different from their lexicographic order, each insertion would take worst case linear time in the number of grid points stored. Fortunately, such insertions can be avoided altogether in level set computations.

[1] i.e. in 3D, push(2,2,6) should be issued before push (2,5,1).

Table II. Time Complexities for Important Algorithms of a N-Dimensional DT-Grid

Algorithm	Time complexity
Push	$O(1)$
Access to Stencil Grid Points	$O(1)$
Sequential access	$O(1)$
Random access	$O(\sum_{n=0}^{N-1} \log C_{\mathbf{x}_n})$
Neighbor access in mth coordinate direction	$O(1+\sum_{n=m}^{N-1} \log C_{\mathbf{x}_n})$
Rebuilding the tubular grid	$O(M_N)$
Dilating the tubular grid	$O(M_N)$

See subsequent sections for details.

A `pop` operation could be implemented similarly to the `push` operation, but is not needed. This is due to the fact that the structure of the tubular grid only changes when the tubular grid is rebuilt. In that case the new tubular grid is constructed from scratch.

The `push` method updates the array constituents of the DT-Grid (defined in Sec. 2.1) and has to deal with the following three cases:

1. The new grid point is the first in a p-column. (As an example see grid points {0,3,8,12,17} in Fig. 1a.)
2. The new grid point is the first grid point in a connected component (and not the first in a p-column). (See grid point 10 in Fig. 1a.)
3. The new grid point is the last in an existing connected component at insertion time. (See the remaining colored grid points in Fig. 1a.)

In Appendix A, we give the full details of the `push` operation.

3.2. Constant Time Sequential Access Using Iterators

The DT-Grid has support for an *Iterator* which is a construct that provides constant time sequential access to grid points in the DT-Grid. The Iterator is in effect a wrapper around a Locator (see Sec. 3.5) that uniquely identifies a grid point. Note that using the Locator constituent it is possible to obtain logarithmic time access to neighboring grid points. However, as will be described in the next section, the *Stencil Iterator* provides constant time access to neighboring grid points within a stencil.

The key method of the Iterator is the `increment` operation which simply increments the associated Locator to point to the next grid point in the tubular grid. This has time complexity $O(1)$. In Appendix B, we describe this `increment` method in detail.

3.3. Constant Time Stencil Access Using Iterators

Most level set methods require access to a finite difference stencil of grid points in order to compute approximations to derivatives like gradients and curvature. Hence, fast access to all members of the stencil is a necessity to ensure optimal performance. By shifting a stencil of Iterators over the tubular grid it is possible to gain constant time access on average to all grid points within the stencil. This is optimal and applies when iterating over the entire tubular grid, which is the case e.g., when advecting, propagating or reinitializing the level set function.

To achieve the above, the DT-Grid has support for a *Stencil Iterator*, which contains a stencil of Iterators, one for each grid point within the stencil.

Incrementing a Stencil Iterator is a bit more involved than incrementing a single Iterator. Here we give an overview of the process.

1. First the Iterator corresponding to the center grid point of the stencil is incremented using the `increment` method described in the previous section. This center Iterator dictates the movement of the entire stencil.
2. Next the remaining Iterators, corresponding to non-center stencil grid points, are incremented until they point to the correct stencil grid point. This is done using the `incrementUntil` method which is described in detail in Appendix C. A non-center stencil grid point may not exist in the tubular grid. If this is the case, access to that particular stencil grid point returns $-\gamma$ if the grid point lies in Ω^- and γ otherwise.

Narrow band level set algorithms typically operate on a number of concentric tubes of increasing width centered about the interface, see [1, 27]. If the DT-Grid is a signed distance field, the Stencil Iterator can be parameterized to return only the grid points within a certain tube, e.g., the zero crossing, without requiring additional storage. This is done simply by incrementing the Iterator of the stencil center grid point until it points to a grid point with absolute value below some threshold.

Incrementing a stencil of Iterators across the DT-Grid provides constant time access on average to all stencil grid points in a particular tube as long as all grid points in the tube are visited. This is the case since: (a) each Iterator of the Stencil Iterator passes over the tubular grid exactly once, which has complexity $O(M_N)$, where M_N is the number of grid points in the tubular grid, (b) M_N is proportional to the number of grid points in any tube centered about the interface, (c) the number of grid

points within the stencil is a constant, (d) access to a grid point through an Iterator has time complexity $O(1)$.

3.4. Logarithmic Time Random Access

As described above, the DT-Grid supports constant time access to grid points within a stencil when accessing the grid sequentially. This is used in all level set algorithms that we have considered, except for the fast marching method [31, 37, 38], which instead uses random and neighbor access (neighbor access is described in the next section). The DT-Grid supports fast operations for random access, which is the mapping from an arbitrary N-dimensional grid point to its corresponding numerical value. A full grid provides constant time random access to all its grid points since this simply amounts to an array access. Constant time random access to grid points is not possible in a DT-Grid. However, logarithmic time, in the number of connected components within p-columns, can be obtained. Note that this is optimal with respect to the storage format. Furthermore, since the number of connected components is usually very small compared to the number of grid points, and since the DT-Grid is very cache coherent, random access is almost as fast as for full grids in practice (see evaluation section for details).

Random access to the grid point $\mathbf{X}_N$ on a N-dimensional DT-Grid is defined recursively in dimensionality as follows.

1. The random access algorithm of the $N-1$-dimensional DT-Grid constituent is used to determine if p-column number $\mathbf{X}_{N-1}$ is contained in the projection of the tubular grid.
2. If this is the case, it is determined if the grid point's Nth coordinate, x_N, lies between the min and max Nth coordinates in p-column number $\mathbf{X}_{N-1}$.
3. If this is the case, binary search for x_N in p-column number $\mathbf{X}_{N-1}$ in the `nCoord` array is employed to find the nearest start or end coordinate of a connected component in the Nth coordinate direction.
4. Finally, it is determined whether the grid point actually exists in the p-column (i.e. is inside the tubular grid) and if this is the case its value is returned.

Access to grid points outside the tubular grid simply return $-\gamma$ if the grid point lies in Ω^- and γ if the grid point lies in Ω^+. The time complexity of random access to a grid point, $\mathbf{X}_N = (x_1, x_2, \ldots, x_N)$, is $O(\sum_{n=0}^{N-1} \log C_{\mathbf{X}_n})$, where $C_{\mathbf{X}_n}$ is the number of connected components in p-column $\mathbf{X}_n$. Appendix D describes the random access operation in full

detail. We finally note that in cases where the number of connected components within p-columns is small it is actually advantageous to employ a linear search instead of the asymptotically optimal binary search.

Employing the random access operation it is easy to implement the following fundamental operations: (1) an operation that determines if a grid point is inside or outside of the interface, (2) an operation that determines if a grid point is in the tubular grid, (3) an operation that determines the closest point on the interface to a grid point inside the tubular grid.

3.5. Logarithmic Time Neighbor Access

This section describes how the DT-Grid implements fast neighbor access to grid points by utilizing structural information about the grid. Constant access time to a grid point is possible if its index into the `value` array constituent is known. However, this index does not provide any structural information about the location of the grid point in relation to neighboring grid points in the coordinate directions. For this reason the DT-Grid supports *Locator* based access. A Locator points to and provides structural information about a grid point in a DT-Grid. It allows for constant access time to the grid point itself and faster neighbor access than can be achieved using random access alone. Locators are not explicitly stored but can be computed by an operation similar to a random access operation. A N-dimensional Locator is defined recursively with respect to dimensionality as

```
struct LocatorND {
   Locator(N-1)D loc;
   unsigned int iv;
   unsigned int ic;
   Index Xn;
};
```

where `loc` is a $N-1$-dimensional Locator. The components `iv` and `ic` point respectively into the `value` and `nCoord` arrays of `DTGridND`. In particular `iv`, points to the value of the grid point and `ic` points to the Nth coordinate of the first grid point in the connected component in which the grid point lies. The last component, `Xn`, is the Nth coordinate of the grid point.

As mentioned above, neighbor access using locators is faster than neighbor access using random access. In fact, when doing neighbor search in the mth coordinate direction, the structural information about the original and neighboring grid point is identical in the first $m-1$ coordinate directions. For the sake of simplicity we explain this in 2D, but stress that the general N-dimensional case is similar. Recall that the storage order

of grid points in a 2D DT-Grid follows the (x, y) lexicographic ordering. Thus, the numerical values of the neighbors in the Y coordinate direction, $(x, y-1)$ and $(x, y+1)$, can be found in constant time from a Locator using the indices `iv`± 1, respectively. If the particular neighbor does not exist in the tubular grid, γ is returned if the neighbor is outside the interface, and $-\gamma$ otherwise.

Neighbors in the X coordinate direction can be found in time $O(\log C_{x\pm 1})$, where $C_x \pm 1$ is the number of connected components in p-column number $x \pm 1$. This is done by first locating the neighbor in the `proj1D` constituent using `iv`± 1 of the 1D Locator constituent. Next, one can apply a binary search for Y in the $x \pm 1$th column.

In general, neighbor search in the mth coordinate direction in a N-dimensional DT-Grid takes time $O(1 + \sum_{n=m}^{N-1} \log C_{\mathbf{x}_n})$.

3.6. Dilating and Rebuilding the Tubular Grid

To ensure numerical stability, level set methods typically apply a reinitialization procedure (after the propagation step) to reset the level set function to a signed distance function. Existing narrow band level set methods furthermore combine this reinitialization step with a method to rebuild the narrow band to ensure that it includes all grid points within a tube of a certain width. When reinitializing the level set function using the fast marching method [31, 37, 38] the tubular grid is rebuilt simultaneously with the reinitialization process. The remainder of this section is organized as follows: Sec. 3.6.1 describes an algorithm that allows us to dilate the tubular grid. The basic idea is simple, but leads to an inefficient algorithm. However, by taking advantage of the storage format of the DT-Grid we show how to construct a very efficient algorithm. Section 3.6.2 describes a generic and efficient algorithm for rebuilding the DT-Grid. By *generic* we mean that the algorithm can be applied independently of the method used to reinitialize the tubular grid (i.e., solve the Eikonal equation, $|\nabla\phi| = 1$). A main building block in this algorithm is the tubular grid dilation algorithm described in Sec. 3.6.1.

3.6.1. Dilating the Tubular Grid in Linear Time

In this section, we present a fast algorithm for dilating a N-dimensional DT-Grid. This algorithm is essential for obtaining feasible asymptotic and practical execution times when rebuilding the tubular grid, which is the topic of the next section.

Dynamic Tubular Grid

A simple idea is to construct a new tubular grid by adding all grid points that pass under a stencil iterated over the original tubular grid. In N-dimensions a desired dilation of $H \cdot dx$ can be achieved with a stencil shaped as a hypercube with $2H+1$ grid points along each edge. Clearly, the resulting tubular grid is a conservative estimate of the grid points no more than a distance of $H \cdot dx$ away from the original tubular grid. The estimate in 1D is exact, but for a N-dimensional stencil, the maximal distance within the stencil measured from the stencil center is $\sqrt{N}H \cdot dx$.

A direct implementation of the simple idea outlined above yields a time complexity of $O(M_N \cdot (2H+1)^N)$. Asymptotically, this amounts to $O(M_N)$, since $(2H+1)^N$ is constant. However, in practice this approach is slow and grid points are added to the tubular grid in an order that is not cache coherent (i.e. not lexicographically ordered).

The dilation algorithm on a N-dimensional DT-Grid exploits the recursive definition of the DT-Grid and, except for the 1D DT-Grid, uses the dilation algorithm of its $N-1$ dimensional DT-Grid constituent recursively. This results in a fast dilation algorithm that ensures cache coherency for subsequent traversals.

The dilation algorithm consists of two steps: an allocation step that computes and allocates the dilated tubular grid, followed by a step that copies the values of the original tubular grid to the dilated tubular grid. The time complexity of the allocation step is $O(C_N)$, where C_N is the total number of connected components in the original N-dimensional tubular grid. In most practical cases C_N is sublinear, i.e. $C_N \ll M_N$, where M_N is the number of grid points in the original tubular grid. The time complexity of the copying step is $O(M_N)$. Note that the number of grid points in the original and the dilated tubular grid are proportional.

Next we give an overview of the allocation step of the algorithm, omitting the copy step, since this is trivial. We start with 1D, followed by a description in 2D which readily generalizes to any number of dimensions. In appendices E and F, we provide pseudo code and give the full detail of the algorithms in 1D and ND, respectively.

An illustration of a tubular grid dilation in 2D is depicted in Fig. 3. Figure 3a shows an initial 2D DT-Grid as well as its 1D DT-Grid constituent. Figure 3c shows the result of dilating the 2D tubular grid by a stencil with $H=1$. Grid points added to the DT-Grids by the dilation algorithm are colored red.

As mentioned earlier, the `xCoord` array of `DTGrid1D` stores a number of connected components, each identified by a start and an end index. The dilation algorithm in 1D simply amounts to dilating each of the connected components by H grid points in both directions and

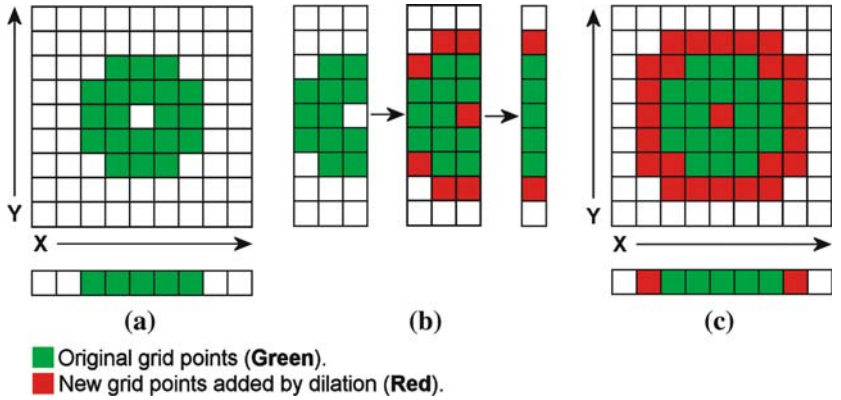


Fig. 3. (a) The original DT-Grid. (b) The process of computing a new p-column (number 3) in the dilated DT-Grid (adding the red grid points) (c) The final dilated DT-Grid.

merging adjacent and overlapping connected components into a single connected component, see Fig. 3.

The 2D dilation algorithm starts by invoking the 1D dilation algorithm on the 1D DT-Grid constituent. Recall that the 1D DT-Grid constituent stores the projection of the 2D tubular grid onto the X-axis. The result of the 1D dilation is that the 1D DT-Grid constituent contains the dilated projection of the 2D tubular grid which is in fact equal to the projection of the dilated 2D tubular grid, see Fig. 3c.

Next, each p-column of the dilated 2D DT-Grid has to be computed. Note that each element in the 1D DT-Grid constituent identifies a p-column in the 2D DT-Grid. Hence, we know exactly which p-columns should be computed in the dilated 2D tubular grid. The process of computing p-column number x proceeds as follows: First dilate the original p-columns numbered $x-H,\ldots,x-1,x,x+1,\ldots,x+H$ independently in the Y direction by H grid points. Next form the union of all the connected components resulting from this dilation in order to obtain p-column number x in the dilated 2D tubular grid. This is illustrated in Fig. 3b. The index pairs contained in the dilated 1D DT-Grid constituent are computed simultaneously with the p-columns. Repeating the process above for each new p-column completes the dilation.

To sum up, the process outlined above dilates the DT-Grid in each dimension independently, and forms new p-columns by taking the union of the dilated original p-columns touched by the stencil. It should be clear, that this method is equivalent to shifting a stencil over the grid and including all grid points that pass under the stencil.

3.6.2. Rebuilding the Tubular Grid in Linear Time

In this section we outline an algorithm to rebuild a DT-Grid, denoted T_α, to include all grid points within a distance α from the interface. The algorithm devised here assumes that the original tubular grid is a distance field and contains all grid points in T_δ, where $\delta < \alpha$. The difference $\alpha - \delta$ may be equal to the maximal movement of the interface between rebuilds. Note that if a method is not restricted by the CFL condition [23], as e.g. with semi-Lagrangian integration, the maximal movement need not be equal to dx. Rebuilding the tubular grid is composed of the following steps

1. Remove from the original tubular grid all grid points outside T_δ. In practice this is done by constructing an intermediate tubular grid and copying all grid points within T_δ to it. This step has time complexity $O(M_N)$.
2. Dilate the intermediate tubular grid by $\alpha - \delta$, where $\alpha - \delta$ corresponds to the difference in width between the tubes T_α and T_δ, using the tubular grid dilation algorithm of Sec. 3.6.1. This step also has time complexity $O(M_N)$.
3. Initialize the values of the grid points included in the new tubular grid by the dilation algorithm to $\pm\delta$ depending on whether they are interior or exterior to the region bounded by the interface. This step also has time complexity $O(M_N)$.

Since each of the above three steps has time complexity $O(M_N)$, so does rebuilding of the tubular grid.

4. EVALUATION AND RESULTS

In this section we present results from three different level set simulations that each demonstrate different features of our DT-Grid. In particular, Sec. 4.1 compares the memory- and time-usage of the DT-Grid to existing methods. Next, Sec. 4.2 verifies that the low memory footprint of the DT-Grid allows for very high resolution simulations. Finally, Sec. 4.3 shows that level set simulations can go out-of-the-box when employed on a DT-Grid.

4.1. Memory and Time Complexity

The following comparisons use the exact same level set simulation code on top of all narrow band approaches and the DT-Grid. Extensive effort was devoted to optimizing implementations of all methods.

Our benchmark test is an extruded spiral evolving under volume-conserving mean curvature flow [27]

$$\frac{\partial \phi}{\partial t} = (\kappa - \bar{\kappa})|\nabla \phi|, \tag{1}$$

where κ is local mean curvature and $\bar{\kappa}$ is the average mean curvature on the interface. We demonstrate the efficiency of the DT-Grid with respect to two numerical methods on a 128^3 grid as shown in Fig. 4.

4.1.1. Numerical Method 1

In the first approach we advect the level set using second order central differences for the spatial parabolic term ($\kappa|\nabla\phi|$) and fifth order accurate HJ-WENO [17] discretization for the spatial hyperbolic term ($\bar{\kappa}|\nabla\phi|$). We use a third order accurate TVD Runge–Kutta [34] time-stepping method. In the reinitialization step we solve the PDE of Peng *et al.* [27] to steady state, again using a fifth order accurate HJ-WENO discretization of the spatial terms and a third order accurate TVD Runge–Kutta time stepping method.

In Figs. 5 and 6 we compare the following methods: (1) The DT-Grid (blue), (2) The method of Peng *et al.* [27] (green), (3) The method of Peng *et al.* improved with an $O(M_N)$ method for rebuilding the narrow band [3] (red), (4) An octree data structure [12, 30, 33] (cyan) storing only the grid points inside the narrow band and using a uniform resolution. We describe the octree implementation in more detail in Appendix G.

Figure 5 shows the time-usage of the different methods. In each iteration of the simulation, we take the maximal time step allowed by the CFL condition. As can be seen, the DT-Grid approach is faster than both the narrow band methods and the octree approach. In this simulation the $O(M_N)$ Peng approach is slower than the original method of Peng. This is partly due to the relatively small simulation grid and the cache ineffectiveness of the $O(M_N)$ method. For simulations on larger grids, the original method of Peng will be slower than the improved $O(M_N)$ Peng method, since it requires a pass through the entire full grid to rebuild the narrow band structure. We have run a large variety of simulations on a wide range of grid sizes and in all cases the DT-Grid has performed better than existing approaches.

In Fig. 6, the memory usage is depicted. The index compression scheme of the DT-Grid makes it more memory efficient than the other approaches. The index arrays of the method of Peng, scale with the size of the interface [27]. However the full grid and a mask array are stored at all times thus forming a lower bound (shown as the black horizontal line in Fig. 6). The memory usage of the octree, scales with the size of the interface, but the storage format is not as effective as that of the DT-Grid.

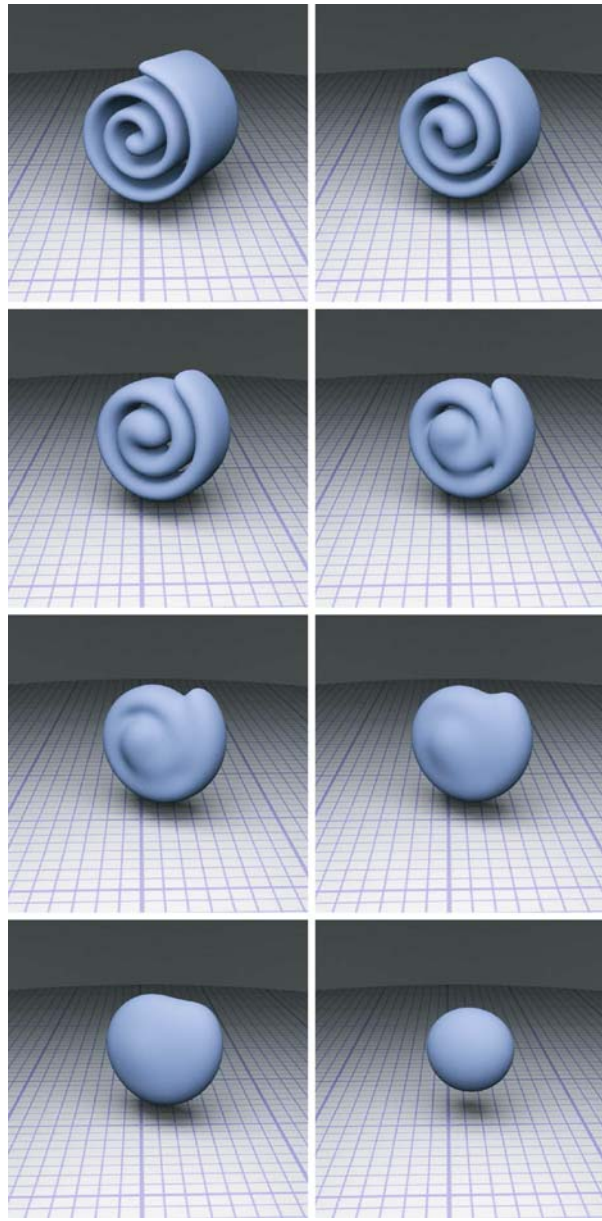


Fig. 4. Extruded spiral evolving under volume conserving mean curvature flow [27] with effective resolution 128^3. The evolution of the interface is depicted from top-left to bottom-right. This example is used as a benchmark test for the narrow band [27], Octree and DT-Grid level set methods. Timings and memory usage are given in Figs. 5–8.

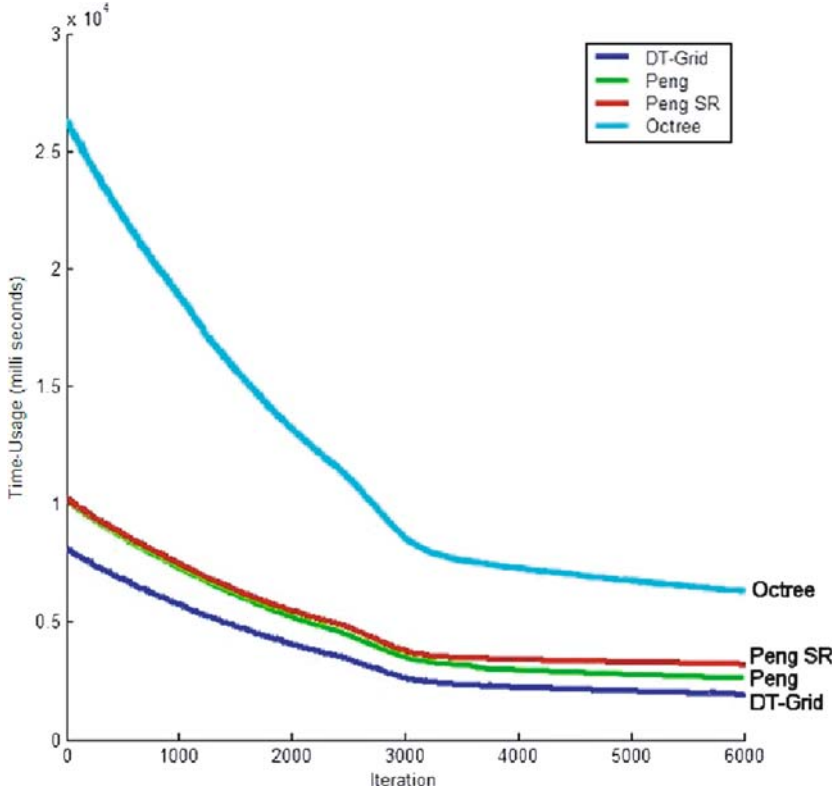


Fig. 5. Time usage in milli seconds on a 3 GHz P4 machine with HJ WENO RK3 for both advection and remormalization.

As a reference we have also plotted the memory usage (computed analytically) of a non-hierarchical storage format that stores the indices of all grid points (magenta) and the very compact linear octree [30] storage format (yellow). Both of these are also less efficient than the DT-Grid. Note that due to the relatively small size of the simulation grid in this particular example, the memory savings of the DT-Grid may appear to be moderate. However, we emphasize that the memory consumption of the DT-Grid is in fact close to optimal as will be described below. In Sec. 4.2 we demonstrate a simulation on a high resolution grid, where the memory consumptions of the DT-Grid and the method of Peng are 67.2 MB and 5.2 GB, respectively.

Figure 7 illustrates the efficiency of the index compression scheme of the DT-Grid. We have plotted the total amount of memory occupied by

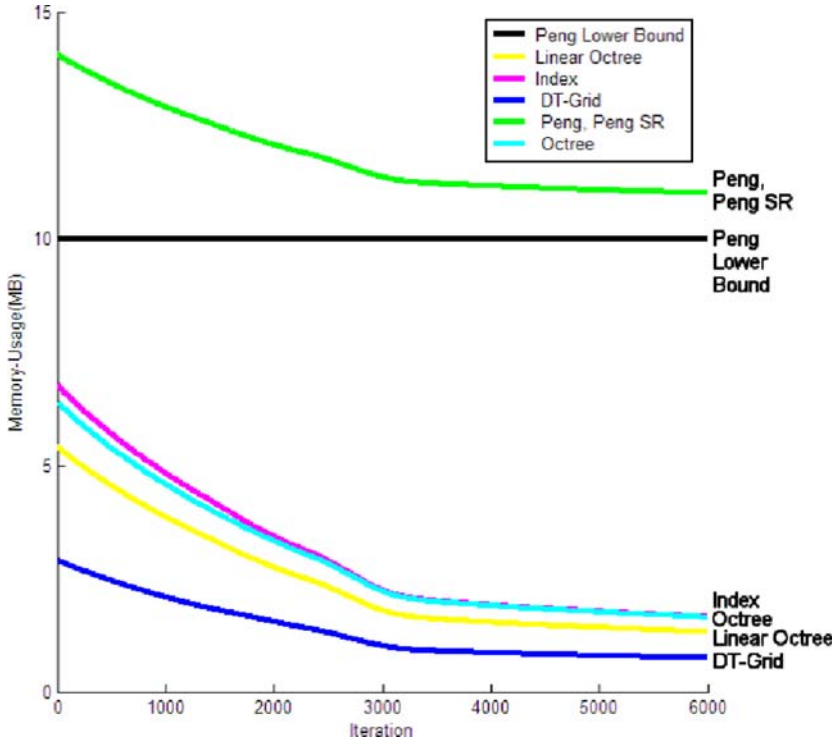


Fig. 6. Memory usage.

the schemes to the amount of memory occupied by the numerical values in the tubular grid (or narrow band) alone. As can be seen this quantity is very close to 1 for the DT-Grid. This is optimal with respect to schemes that store the numerical values in the narrow band uncompressed.

To sum up: The narrow band method of Peng is fast but not memory efficient. The memory usage of the octree scales with the size of the interface, however our tests indicate that the approach is relatively slow. The DT-Grid appears to be the only approach which is *both memory and time efficient*. In fact it is both faster than the narrow band method of Peng *et al.* [27] and more memory efficient than the octree.

4.1.2. Numerical Method 2

In the second approach we advect the level set using HJ-WENO and TVD Runge–Kutta as above, but reinitialize the level set in the narrow band (or tubular grid) using the Fast Marching method [32].

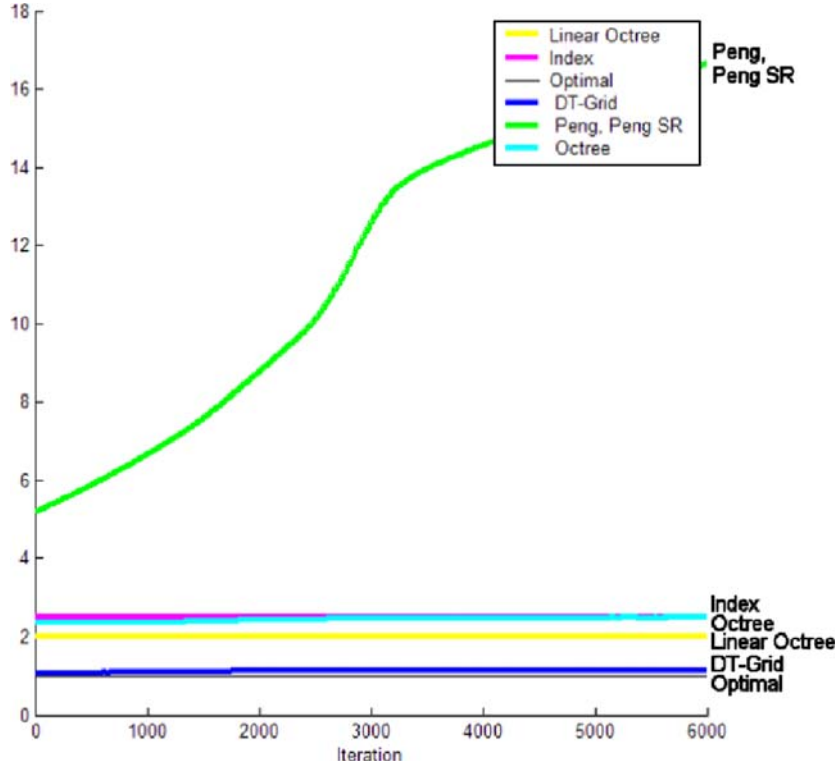


Fig. 7. Efficiency of index compression.

We compare the following methods: (1) DT-Grid (figure-label DT-Grid FMM), (2) Peng (figure-label Peng FMM).

The time usage is plotted in Fig. 8. In this case the DT-Grid also performs better than the full grid Peng approach. This can be attributed to the cache coherency of the DT-Grid. The fast marching method employs a lot of random and neighbor access operations since it does not visit a grid sequentially. Even though these operations on the DT-Grid are asymptotically slower than in a full grid, its cache efficiency makes the total simulation time faster than the Peng method.

4.2. High Resolution Level Sets

The DT-Grid has a very low memory footprint, hence allowing very large or equivalently very high resolution level set surfaces to be represented without exceeding the main memory limit. Enright *et al.* [7]

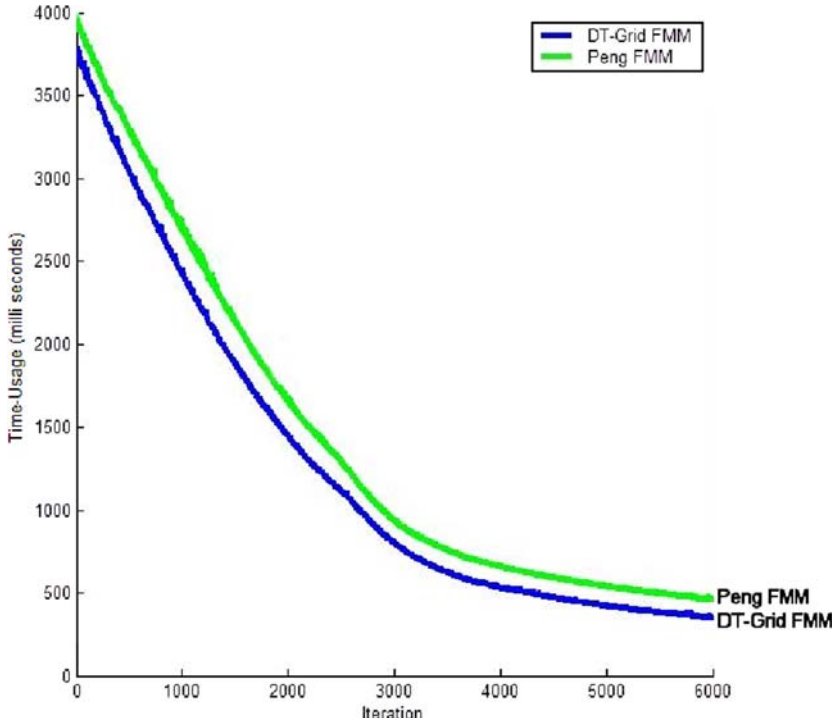


Fig. 8. Time usage in milli seconds on a 3 GHz P4 machine with HJ WENO RK3 for advection and FMM for renormalization.

introduced a test based on a three dimensional incompressible flow field initially proposed by LeVeque [15] to demonstrate the volume conserving properties of the particle level set method. The test was run on a 100^3 full grid with and without particles. The particle level set method proved excellent in conserving the volume, however, the resolution of the computational grid was insufficient to capture the thin filaments. Later, Enright *et al.* [8] demonstrated that the interface could be fully resolved on an octree grid with an effective resolution of 512^3, using a combined semi-Lagrangian and particle level set method.

From Fig. 9 it can be concluded that the interface can also be fully resolved on a DT-Grid with an effective resolution of 1024^3 *without particles*. However, we stress that the DT-Grid should not be considered an alternative to the particle level set approach. The setup of this simulation is identical to that of Enright *et al.* [7] which uses HJ-WENO and TVD Runge–Kutta for both the advection and reinitialization steps. This

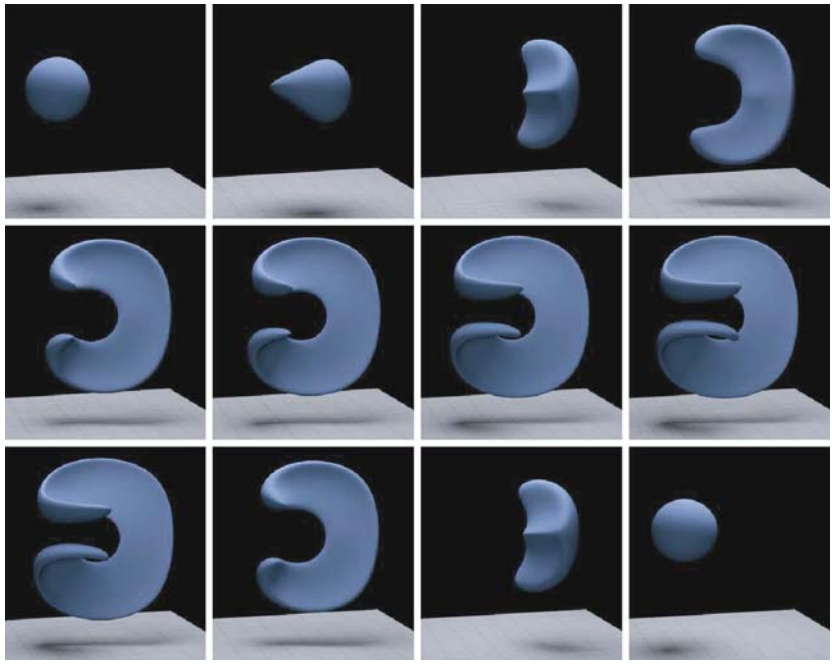


Fig. 9. The DT-Grid enables high resolution grids to be represented with a very low memory footprint. In this example the resolution was 1024^3 and the maximal memory usage 67.2 MB. The same simulation run with the method of Peng *et al.* [27] would consume 5.2 GB of storage.

particular example clearly demonstrates that the DT-Grid enables high resolution interfaces to be represented.

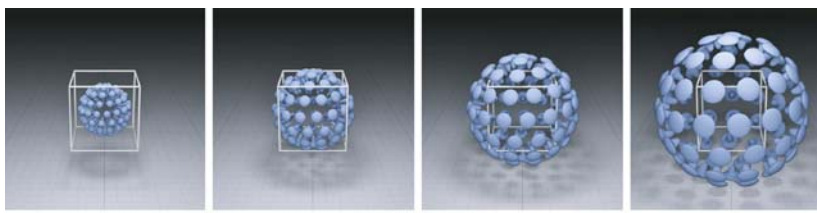
At the time step where the number of grid points in the tubular grid peaks, 1.41% of the grid points in the entire full 1024^3 volume is occupied by the tubular grid and hence stored by the DT-Grid. At this time, the DT-Grid uses only 67.2 MB of storage in total. Furthermore, the memory used by the DT-Grid is only 1.64% of the storage that the 1024^3 full volume would occupy alone. The additional $0.23 = 1.64 - 1.41\%$ of storage used by the DT-Grid is occupied by the compressed indices and lower dimensionality DT-Grid constituents. From the number of grid points in the tubular grid at peak time, we computed analytically that the method of Peng would occupy at least 5.2 GB of storage. Other non-hierarchical approaches to storing the tubular grid, such as storing the indices explicity would result in a total memory usage of 9.86% and additionally storing pointers to the neighbors would yield 43.68%, giving a total memory

usage of 144 and 490 MB, respectively. The above analysis clearly illustrates the effectiveness of the index compression scheme employed by the DT-Grid.

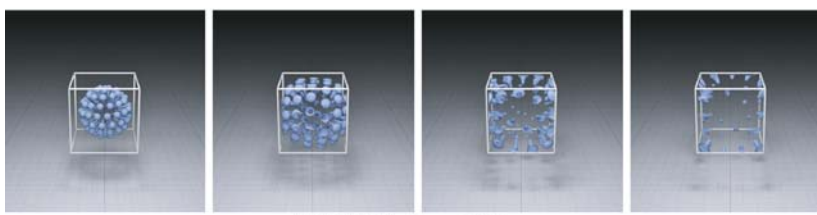
4.3. Out-of-the-Box Level Sets

The DT-Grid is not bounded by an underlying box of a fixed computational domain. In this section we illustrate this important property by evolving the level set shown in Fig. 10(a) using convection-diffusion [23]. In the convection-diffusion equation we combine motion by mean curvature with motion by a vector field that at each point is the normalized radial direction from the origin. The mean curvature term creates multiple pinch-offs at the center of the level set surface (see the left-most image in Fig. 10(a)), and we force its contribution to zero over time. The radial vector field advects the level set surface away from the origin. This simulation is designed merely to demonstrate the out-of-the-box feature of our DT-Grid, and as such the detail of the setup (including the initial shape) is irrelevant.

Figure 10(b) illustrates what happens on full grids and octrees. As the level set moves beyond the boundary, it disappears. In contrast, the level set on the DT-Grid (Fig. 10(a)) moves beyond the box as the *grid is not bounded*. The memory usage in Fig. 10(a) is 14.0, 25.2, 39.3 and



(a) DT-Grid



(b) Full Grid and Octrees

Fig. 10. DT-Grid level set simulations can go out-of-the-box whereas level set simulations based on full grids and Octrees are limited to the box of the underlying grid.

64.1 MB, respectively, numbered from left to right. (Note that the increase in memory is exclusively due to the fact that surface area increases with the expansion). Effective grid sizes are approximately 256^3, 343^3, 455^3 and 630^3, while the DT-Grid in general is a non-convex tubular grid.

This "out-of-the-box" feature is not obtainable using existing narrow band or standard octree based approaches without either compromising memory consumption or computational efficiency. Using full grids, one could progressively allocate larger grids as the level set approaches the boundaries. However, due to memory constraints this becomes impossible already at relatively small grid sizes. Octrees are more memory efficient than the full grid narrow band approach, but if progressively reallocating to larger octrees, the depth of the tree grows, making the traversal more inefficient. Furthermore, the octree is not as memory efficient and fast as the DT-Grid. The combined uniform and octree grid by Losasso *et al.* [11], currently under submission, decouples the depth of the octree from the overall domain size, hence making an expansion of the domain more feasible.

Out-of-the-box level set simulations are extremely convenient and useful since no boundary conditions are needed and the level set can move freely without ever colliding with the boundaries of an underlying grid. A large body of existing work could take advantage of this, e.g., the simulation of dendritic growth in [13].

5. CONCLUSIONS AND FUTURE WORK

Level set surfaces are of great importance in computational physics and chemistry for the tracking of interfaces. In this paper we have presented the DT-Grid, a novel, non-hierarchical data structure and efficient algorithms for representing and manipulating level sets. The DT-Grid is designed to provide a very low memory footprint, making it possible to represent higher resolution interfaces than previous narrow band and tree based methods. Furthermore, both the data structure and the algorithms have been designed to take advantage of the memory hierarchies of modern computers, thus making the DT-Grid very fast. In fact, all evaluations that we have performed show that the DT-Grid is both more time and memory efficient than existing narrow-band and tree-based algorithms for level set surfaces. The most important novel feature of the DT-Grid is that it allows for level set simulations to be out-of-the-box, which is demonstrated for the first time in Sec. 4.3: the DT-Grid can expand freely without being limited by the computational box of an underlying grid, and boundary conditions can be avoided. The DT-Grid has been specifically

designed for the sequential access used in most level set algorithms and provides constant time access to grid points within the stencil. A drawback is that random and neighbor access times become asymptotically logarithmic if accessing grid points outside the stencil or non-sequentially. However, the reader should keep in mind that random access is logarithmic in *the number of connected components* within a p-column, as opposed to the actual number of elements within the connected components in a p-column. Furthermore, all our practical experiments have shown that due to the cache coherency of the DT-Grid, random access is almost as fast as the constant time access on full grids in practice. This is evident from the fact that our level set simulations based on the fast marching method – which in turn employs random and neighbor access operations – are still faster on the DT-Grid.

In this paper, we have focused on the ability of the DT-Grid to store the narrow band of a level set interface. However, it can readily be used to store any closed non-convex volume e.g. the velocities in a volume enclosed by a fluid. Hence, it should be rather straightforward to store and manipulate both a fluid interface and the velocities in the enclosed volume in a combined DT-Grid data structure.

The DT-Grid is not an adaptive representation in its traditional sense since the interface and/or volume is represented uniformly. However, in the future we plan to extend it to a multi-resolution approach.

We believe that the DT-Grid will be of great importance in the area of practical level set simulations and foresee many exciting applications of level sets that were not previously possible.

ACKNOWLEDGMENTS

The authors are very grateful to Ola Nilsson for raytracing images and for providing many helpful comments on drafts of this paper. The comments of Anders Brodersen also improved the paper. KM would also like to thank Mike Giles at Oxford University for a very stimulating discussion that took place at Caltech in the summer of 2003. This conversation largely inspired the initial phase of the current work. This work was partly funded by the Swedish Research Council (VR grant # 621-2004-5017) and partly supported by Center for Interactive Spaces under ISIS Katrinebjerg, Århus, Denmark. Finally we would like to thank the referees of this paper for useful comments and suggestions for improvements.

Appendix A: The Push Algorithm

In this appendix, we present a C++ pseudo code representation of the `push` operation for a 2D DT-Grid. The general N-dimensional version is similar.

```
void push(Index x, Index y, Scalar val)
{
  if ( proj1D.xCoord.last() != x)        // (x,y) lies in new p-column
  {
    Pair p(value.size(), yCoord.size());
    proj1D.push(x, p);
    yCoord.push(y);
    yCoord.push(y);
    acc.push(value.size());
  }
  else if ( yCoord.last() != y-1 )       // (x,y) first in connected
                                           component
  {
    yCoord.push(y);
    yCoord.push(y);
    acc.push(value.size());
  }
  else                                // (x,y) last in connected component
  {
    yCoord.last() = y;
  }
  value.push(val);
}
```

Remarks: In **case 1**, `(x,y)` is the first grid point in the `x`'th p-column and the `proj1D` data structure must be set to point to this grid point. The `y` coordinate is pushed twice onto the `yCoord` array since it denotes both the start and end of a new connected component. This is also reflected in **case 2**. In **case 3** the end of the connected component is set to `y` since `(x,y)` was adjacent to the previous, `yCoord.last()`, grid point pushed. All operations above are $O(1)$, hence the `push` operation is $O(1)$.

Appendix B: The Increment Algorithm

Here we give the implementation details of the `increment` operation supported by an Iterator. The `increment` algorithm simply increments the Iterator to point to the next grid point in the tubular grid. We assume that the Iterator of an N-dimensional DT-Grid, `IteratorND`, contains

- A reference, `grid`, to the DT-Grid being iterated over.
- A reference, `iterator(N-1)D`, to an $N-1$-dimensional iterator (if $N > 1$) defined equivalently.
- A value, `value`.

Furthermore we assume that the `IndexPair` described in Sec. 2.1 has two members, `iv` that points into the `value` array and `ic` that points into the `nCoord` array. Using the definition of the `LocatorND` presented in Sec. 3.5, the `increment` operation looks as follows in C++ pseudo code

```
void IteratorND::increment(LocatorND loc)
{
  loc.iv++;

  value = grid.value[loc.iv];

  if (loc.Xn == grid.nCoord(loc.ic+1))
  {
    loc.ic += 2;
    loc.Xn = grid.nCoord(loc.ic);

    if (grid.proj(N-1)D.value[loc.loc.iv+1].ic == loc.ic)
    {
      iterator(N-1)D.increment(loc.loc);
    }
  }
  else
  {
    loc.Xn++;
  }
}
```

Remarks: The first `if` statement tests if `IteratorND` exits a connected component. The second `if` statement tests if `IteratorND` passes to a new p-column and increments the Locators of lower dimensionalities recursively to e.g. set the grid point coordinates appropriately. Since all steps above are $O(1)$, the `increment` method is $O(1)$.

Appendix C: The IncrementUntil Algorithm

The `incrementUntil(GridPoint` $\mathbf{X}_N$`, LocatorND loc)` operation of `IteratorND` increments an Iterator until it points to the grid point with the coordinates given by $\mathbf{X}_N$, see Secs. 3.2 and 3.3 and Appendix 5. It works as follows

1. Increment `loc` until it points to the lexicographically smallest grid point, $\mathbf{Y}_N$, in the tubular grid, that is larger than or equal to $\mathbf{X}_N$.
2. If $\mathbf{X}_N == \mathbf{Y}_N$ set `value=grid.value[loc.iv]`, otherwise set `value=sign(grid.value[loc.iv])`$\cdot\gamma$.

Remarks: In practice it is crucial how **step 1** above is implemented and a few optimizations are possible:

- If the center grid point of the stencil stays in the same p-column after an `increment` operation, we know that non-center stencil grid points will also stay in the same p-column. In this case we only need to increment `loc` until either $y_N = x_N$ or $y_{N-1} \neq x_{N-1}$.
- If the center grid point of the stencil only moves one grid point in the Nth coordinate direction by an `increment` operation, we know that those non-center stencil grid points, that will not pass out of the tubular grid, will also move exactly one grid point in the Nth coordinate direction. In that case **step 1** and **step 2** above can be implemented with the following three lines

```
loc.iv++;
value = grid.value[loc.iv];
loc.Xn++;
```

  This optimization can typically be applied if the stencil is guaranteed never to pass out of the tubular grid when iterating over a certain tube.

Note that the optimizations described above can be applied recursively. Since we assume that the entire tubular grid is visited, all steps above take time $O(1)$ on average given the arguments in Sec. 3.3, and hence access to grid points within the stencil is $O(1)$ on average.

Appendix D: The Random Access Algorithm

Let γ be the width of the tubular grid and $\mathbf{X}_N = (\mathbf{X}_{N-1}, x_N)$ an N-dimensional grid point with $N-1$ and 1 dimensional sub components $\mathbf{X}_{N-1}$ and x_N respectively. The method `randomAccess` returns a triple `(inside, i, v)`. `inside` is a boolean telling whether $\mathbf{X}_N$ is inside the tubular grid, `i` is the index of $\mathbf{X}_N$ into the `value` array of `DTGridND` (which is valid if `inside==true`) and `v` is the value of $\mathbf{X}_N$. Below we assume that the `IndexPair` described in Sec. 2.1 has two members, `iv` that points into the `value` array and `ic` that points into the `nCoord` array. The algorithm proceeds as follows:

1. If $N == 1$ set $k^{\min} = 0$ and $k^{\max} =$ `xCoord.size()-1`. Otherwise
   (a) Set `(inside,i,v) = proj(N-1)D.randomAccess(`$\mathbf{X}_{N-1}$`)`.
   (b) If `inside==false` return `(false,0,`γ`)`.
   (c) Set $k^{\min} =$ `proj(N-1)D.value[i].ic` and $k^{\max} =$ `proj(N-1)D.value[i+1].ic-1`.
2. If $x_N <$ `nCoord[`$k^{\min}$`]` || $x_N >$ `nCoord[`$k^{\max}$`]` return `(false, 0,`γ`)`.

3. Perform a binary search for x_N at even positions (indices that point to the start of a connected component) in the `nCoord` array. The search is delimited by the indices $k^{\min}$ and $k^{\max}-1$ (both inclusive), and the index determined is denoted `k`.
4. `nCoord[k]` $\leqslant x_N <$ `nCoord[k+2]` and there are two cases:
   (a) If x_N `<= nCoord[k+1]`, set `i=acc[(k>>1)]+`x_N `-nCoord[k]` and return `(true,i,value[i])`.
   (b) Else return `(false,0,sign(value[acc[k>>1]])·`γ`)`.

Remarks: Step 1 determines $k^{\min}$ and $k^{\max}$ which are the indices into the `nCoord` array of the minimum and maximum Nth coordinate in p-column number $(x_1, x_2, \ldots, x_{N-1})$. In **step 4**, the >> is the right-shift operator. `k` is right-shifted by one since `k` is an index into the `nCoord` array which has twice as many elements as the `acc` array.

Steps 2 and 4 above have time complexity $O(1)$. **Step 3** has time complexity $O(\log C_{\mathbf{X}_{N-1}})$, where $C_{\mathbf{X}_{N-1}}$ is the number of connected components in the $\mathbf{X}_{N-1}$th column. Hence by applying this argument recursively in **step 1** it can be seen that random access in a N-dimensional DT-Grid has time complexity $O(\sum_{n=0}^{N-1} \log C_{\mathbf{X}_n})$. Note also that this complexity is optimal with respect to the storage format, e.g. $O(1)$ random access time is not possible.

Appendix E: The 1D Dilation Algorithm

The 1D dilation algorithm is sufficiently simple to be presented in C++ pseudo code. Below we only describe the allocation step of the dilation algorithm, since the copying step is trivial, see Sec. 3.6.1. Below the `DTGrid1D` named `d1D` eventually contains the dilated DT-Grid and we assume that the DT-Grid is dilated by a hypercube stencil with $2H+1$ grid points along each edge, see Sec. 3.6.1.

```
void DTGrid1D::dilate(unsigned int H, DTGrid1D d1D) {
  unsigned int numValues;
  Index start = xCoord[0]-H;
  Index end = xCoord[1]+H;
  unsigned int i = 2;

  d1D.xCoord.push(start);
  d1D.xCoord.push(end);
  d1D.acc.push(0);

  while ( i < xCoord.size() )
  {
     start = xCoord[i]-H;
     i++;
     if ( start <= end+1 )
```

```
    {
      d1D.xCoord.last() = end = xCoord[i]+H;
    }
    else
    {
      unsigned int j = d1D.xCoord.size();
      unsigned int connectedComponentLength
      = d1D.xCoord[j-1]-d1D.xCoord[j-2]+1;
      d1D.acc.push(d1D.acc.last()+connectedComponentLength);
      d1D.xCoord.push(start);
      end = xCoord[i]+H;
      d1D.xCoord.push(end);
    }
    i++;
  }

  numValues = d1D.acc.last() + d1D.xCoord[j-1]-d1D.xCoord[j-2]+1;
  values.allocate(numValues);
}
```

Remarks: If the test `start <= end+1` in the `if` statement is true, it means that two adjacent connected components overlap. Hence storage of the new connected component in the `xCoord` and `acc` arrays is postponed. If the `else` case is entered, the connected component currently being processed does not overlap with the previous. In that case it can be stored. Clearly the allocation step of the 1D dilation algorithm has time complexity $O(C_1)$, where C_1 is the total number of connected components in the 1D DT-Grid.

Appendix F: The N-Dimensional Dilation Algorithm

In this section we describe the allocation step of the N-dimensional tubular grid dilation algorithm for $N > 1$ (again, the copying step is trivial, see Sec. 3.6.1). Recall that this algorithm effectively corresponds to shifting a stencil shaped as a hypercube with $2H + 1$ grid points along each edge over the original DT-Grid and including, in the new DT-Grid, all grid points that pass under the stencil.

Below we assume the existence of a data structure named the *Column Union*. It maintains a queue of p-columns and the maximal number of p-columns allowed in the queue is $(2H+1)^{N-1}$, which is a constant. It supports the following operations

- **InsertColumn:** Inserts a p-column at the end of the queue. The p-column is identified by its start- and end-index into the `nCoord` array of `DTGridND`. This operation has time complexity $O(1)$.
- **RemoveColumn:** Removes the first p-column in the queue. This operation has time complexity $O(1)$.

- **ComputeUnion:** Computes a new p-column consisting of the connected components formed by taking the union of all connected components in the p-columns stored in the Column Union data structure. Furthermore each connected component is dilated independently by H grid points in each direction on the fly before including it in the union. Since the start- and end-coordinates of connected components are sorted within each p-column in the `nCoord` array, forming the union is simple: Scan through the coordinates of all connected components simultaneously in ascending order. Maintain a count that is incremented when a connected component is entered and decremented when a connected component is exited. The coordinates encountered when the count is zero can be taken as the coordinates of the connected components of the union. Adjacent connected components must be merged into a single connected component. This operation has a time complexity that is linear in the number of connected components in the p-columns stored, since finding the minimum coordinate at each step takes time max $O((2H+1)^{N-1}) = O(1)$.

Next we describe the allocation step of the N-dimensional tubular grid dilation algorithm in detail.

1. Call `proj(N-1)D.dilate(H, dilatedProj(N-1)D)`. After this call, `dilatedProj(N-1)D` will hold the dilated N-1 dimensional DT-Grid constituent. The pairs of indices, `IndexPair`, stored in `dilatedProj(N-1)D` are not yet initialized, only the raw storage is allocated.
2. Obtain an `Iterator`, `iteratorDilate`, from `dilatedProj(N-1)D`.
3. Obtain a `StencilIterator`, `iteratorOrig`, from `proj(N-1)D`. The stencil of the `StencilIterator` effectively has dimensions $(2H + 1)^{N-1}$, however as the stencil moves over `proj(N-1)` it is only necessary to monitor which grid points enter and exit the stencil respectively, so the number of `Iterator` instances maintained by `iteratorOrig` is in fact $2(2H + 1)^{N-2}$. The movement of the stencil will be dictated by the movement of the `iterator Dilate Iterator`.
4. Iterate over all grid points in `dilatedProj(N-1)D` using `iteratorDilate`. For each such grid point, do the following
   (a) For each of the $2(2H+1)^{N-2}$ `Iterator` instances of `iteratorOrig` that point to an existing grid point in

proj(N-1)D, do the following: The existing grid point in proj(N-1)D contains an IndexPair and hence identify a p-column in the original DTGridND. If the p-column is entering the stencil, call insertColumn on the Column Union data structure to insert the p-column. If the p-column is exiting the stencil, call removeColumn on the Column Union data structure. Iterating the stencil over the grid points of proj(N-1)D (and hence the p-columns of the original DTGridND) in lexicographic order ensures that p-columns enter and leave the stencil like a FIFO queue.

(b) Call computeUnion on the Column Union to compute the connected components of the new p-column pointed to by iteratorDilate in the dilated DTGridND. At this point the ColumnUnion contains all p-columns from the original DTGridND touched by a $(2H+1)^N$ stencil centered at the new p-column. The IndexPair in dilatedProj(N-1)D is set to point to this new p-column and the connected components of the new p-column are inserted directly into the nCoord array of the dilated DTGridND. Furthermore the content of the corresponding part of the acc array is computed by a scan through the start and end indices of the connected components in the new p-column just computed. At the same time a variable numValues indicating the total number of grid points included so far in the dilated DTGridND is incremented by the number of grid points in this p-column.

(c) Increment iteratorDilate. The movement of this Iterator dictates the movement of the iterOrig StencilIterator.

5. Allocate memory for the value array. It will include numValues elements.

The time complexity of the allocation step of the ND tubular grid dilation algorithm can be derived as follows: In 2D, we first dilate the 1D DT-Grid constituent which takes time $O(C_1)$ as described in the previous appendix. Next, every p-column in the original 2D DT-Grid (of which there are M_1) is inserted into and removed from the Column Union data structure $2G+1$ times. Each operation takes time $O(1)$. Hence the total time for this is $O(M_1)$, since $2H+1$ is a constant. To compute the new

p-columns, each connected component (of which there are C_2) is used $2H+1$ times, since this is the number of unions that it contributes to. Each time a connected component is used to compute a union this takes time $O(2H+1)$ since it was located from a total of $2G+1$ possibilities to be the connected component with the smallest start or end Y coordinate. In total, computing the new p-columns takes time $O((2H+1)^2 C_2)$, which equals $O(C_2)$, since $(2H+1)^2$ is a constant. Finally, summing all the contributions gives a time complexity of $O(C_1+M_1+C_2)$ which is $O(C_2)$.

The complexity analysis proceeds similarly in N-dimensions except that $2H+1$ must be replaced by $(2H+1)^{N-1}$ and $(2H+1)^2$ replaced by $(2H+1)^N$, hence the allocation step of the dilation algorithm on an N dimensional DT-Grid has time complexity $O(C_N)$.

Appendix G: Octree Implementation

In Sec. 4 we compare an octree based level set implementation to the DT-Grid. The octree used stores only the grid points inside the narrow band and uses a uniform resolution. A cell is defined as

```
struct OctreeCell
{
  OctreeCell *parent;
  union
  {
    OctreeCell *children[8];
    float *data[8];
  } u;
};
```

The implementation uses the algorithm of Stolte *et al.* [33] to traverse the octree sequentially and the neighbor access algorithms of Frisken *et al.* [12]. Octree cells are allocated in a memory pool [19] to speed up allocation and deallocation and to increase the cache coherency. The octree is constructed and rebuilt in time $O(M_N)$. Our implementation is optimized and appropriate methods inlined.

REFERENCES

1. Adalsteinsson, D., and Sethian, J. A. (1995). A fast level set method for propagating interfaces. *J. Comput. Phys.* **118**(2), 269–277.
2. Adalsteinsson, D., and Sethian, J. A. (1999). The fast construction of extension velocities in level set methods. *J. Comput. Phys.* **148**, 2–22.
3. Breen, D., Fedkiw, R., Museth, K., Osher, S., Sapiro, G., and Whitaker, R. (2004). *Level Sets and PDE Methods for Computer Graphics*. ACM SIGGRAPH '04 COURSE #27. ACM SIGGRAPH, Los Angeles, CA, August 2004. ISBN 1-58113-950-X.

4. Bridson, R. (2003). *Compuational Aspects of Dynamic Surfaces*. Ph.D., thesis, Stanford University, Stanford, California.
5. Chan, T., and Vese, L. (2001). Active contours without edges. *IEEE Trans. Image Process.* **10**, 266–277.
6. Droske, M., Meyer, B., Rumpf, M., and Schaller, C. (2001). An adaptive level set method for medical image segmentation. *Lect. Notes Comput. Sci.* pp. 412–422.
7. 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.
8. Enright, D., Losasso, F., and Fedkiw, R. (2005). A fast and accurate semi-lagrangian particle level set method. *Comput. Struct*. **83**(6–7), 479–490.
9. Fedkiw, R., Aslam, T., and Xu, S. (1999). The ghost fluid method for deflagration and detonation discontinuities. *J. Comput. Phys.* **154**, 393–427.
10. Foster, N., and Fedkiw, R. (2001). Practical animation of liquids. In *ACM SIGGRAPH '01*, ACM Press, pp. 23–30.
11. Osher, S., Losasso, F., and Fedkiw, R. Spatially adaptive techniques for level set methods and incompressible flow. *Comput. Fluids* (in review).
12. Frisken, S., and Perry, R. (2003). Simple and efficient traversal methods for quadtrees and octrees. *J. Graphics Tools* **7**(3), 1–11.
13. Gibou, F., Fedkiw, R., Caflisch, R., and Osher, S. (2003). A level set approach for the numerical simulation of dendritic growth. *J. Sci. Comput.* **19**(1–3), 183–199.
14. Houston, B., Wiebe, M., and Batty, C. (2004). Rle sparse level sets. In *Proceedings of the SIGGRAPH 2004 Conference on Sketches & Applications*. ACM Press.
15. Leveque, R. J. (1996). High-resolution conservative algorithms for advection in incompressible flow. *SIAM* **33**(2), 627–665.
16. Losasso, F., Gibou, F., and Fedkiw, R. (2004). Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH '04*, ACM Press, pp. 457–462.
17. Liu, X. D., Osher, S. J., and Chan, T. (1994). Weighted essentially nonoscillatory schemes. *J. Comput. Phys.* **115**, 200–212.
18. Museth, K., Breen, D., Whitaker, R., and Barr, A. (2002). Level set surface editing operators. *ACM Trans. on Graphics (Proc. SIGGRAPH)* **21**(3), 330–338.
19. Meyers, S. (1997). *Effective C++ (2nd ed.): 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc.
20. Milne, R. B. (2003). *An Adaptive Level Set Method*. Ph.D., thesis, Berkeley National Laboratory, Physics Division, Mathematics Department.
21. Min, C. (2004). Local level set method in high dimension and codimension. *J. Comput. Phys.* **200**, 368–382.
22. Nguyen, D., Fedkiw, R., and Jensen, H. W. (2002). Physically based modeling and animation of fire. In *ACM SIGGRAPH '02*, ACM Press, pp. 721–728.
23. Osher, S. J., and Fedkiw, R. P. (2002). *Level Set Methods and Dynamic Implicit Surfaces*. Springer, Berlin.
24. Osher, S., and Paragios, N. (eds). (2003). *Geometric Level Set Methods in Imaging, Vision and Graphics*. Springer-Verlag.
25. Osher, S., and Sethian, J. (1988). Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.* **79**, 12–49.
26. Osher, S., and Shu, C. W. (1991). High-order essentially nonoscillatory schemes for hamilton-jacobi equations. *SIAM J. Num. Anal.* **28**, 907–922.
27. Peng, D., Merriman, B., Osher, S., Zhao, H., and Kang, M. (1999). A pde-based fast local level set method. *J. Comput. Phys.* **155**(2), 410–438.
28. Rudin, L., Osher, S., and Fatemi, E. (1992). Nonlinear total variation based noise removal algorithms. *Physica D* **60**, 259–268.

29. Sussman, M., Almgren, A. S., Bell, J. B., Colella, P., Howell, L. H., and Welcome, M. L. (1999). An adaptive level set approach for incompressible two-phase flows. *J. Comput. Phys.* **148**, 81–124.
30. Samet, H. (1990). *Applications of Spatial Data Structures: Computer Graphics, Image Processing, GIS*. Addison-Wesley, Reading, MA.
31. Sethian, J. A. (1996). A fast marching level set method for monotonically advancing fronts. *Proc. Nat. Acad. Sci. USA* **93**(4), 1591–1595.
32. Sethian, J. A. (1999). *Level Set Methods and Fast Marching Methods*, Second edition Cambridge University Press, Cambridge, UK.
33. Stolte, N., and Kaufman, A. (1998). Parallel spatial enumeration of implicit surfaces using interval arithmetic for octree generation and its direct visualization. In *Implicit Surfaces '98*, pp. 81–87.
34. Shu, C. W., and Osher, S. (1988). Efficient implementation of essentially non-oscillatory shock capturing schemes. *J. Comput. Phys.* **77**, 439–471.
35. Sussman, M., Smereka, P., and Osher, S. (1994). A level set approach to computing solutions to incompressible two-flow. *J. Comput. Phys.* **114**, 146–159.
36. Strain, J. A. (1999). Tree methods for moving interfaces. *J. Comput. Phys.* **151**(2), 616–648.
37. Tsitsiklis, J. N. (1994). Efficient algorithms for globally optimal trajectories. *Proceedings of the 33rd Conference on Decision and Control, Lake Buena Vista, LF*, pp. 1368–1373.
38. Tsitsiklis, J. N. (1995). Efficient algorithms for globally optimal trajectories. *IEEE Trans. Automat. Contr.* **40**, 1528–1538.
39. Whitaker. R. T. (1998). A level-set approach to 3d reconstruction from range data. *Int. J. Comput. Vision* **29**(3), 203–231.
40. Westermann, R., Kobbelt, L., and Ertl, T. (1999). Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. *Visual Comput.* **15**(2), 100–111.
41. Zhao, H., Osher, S., Merrian, B., and Kang, M. (2000). Implicit and nonparametric shape reconstruction from unorganized data using a variational level set method. *Comput. Vision Image Understand.* **80**, 294–314.