

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

Michael B. Nielsen

Ken Museth

Linköping University Electronic Press  
Linköping, Sweden

<http://www.ep.liu.se/ea/cis/2004/001/>

*Published on November 11, 2004 by  
Linköping University Electronic Press  
581 83 Linköping, Sweden*

**Linköping Electronic Articles in  
Computer and Information Science**

*ISSN 1401-9841*

*Series editor: Erik Sandewall*

©2004 Michael B. Nielsen and Ken Museth

*Typeset by the author using L<sup>A</sup>T<sub>E</sub>X*

*Formatted using étendu style*

**Recommended citation:**

*<Author>. <Title>. Linköping electronic articles  
in computer and information science, Vol. 9(2004): nr 1.  
<http://www.ep.liu.se/ea/cis/2004/001/>. November 11, 2004.*

*This URL will also contain a link to the author's home page.*

*The publishers will keep this article on-line on the Internet  
(or its possible replacement network in the future)  
for a period of 25 years from the date of publication,  
barring exceptional circumstances as described separately.*

*The on-line availability of the article implies  
a permanent permission for anyone to read the article on-line,  
to print out single copies of it, and to use it unchanged  
for any non-commercial research and educational purpose,  
including making copies for classroom use.*

*This permission can not be revoked by subsequent  
transfers of copyright. All other uses of the article are  
conditional on the consent of the copyright owner.*

*The publication of the article on the date stated above  
included also the production of a limited number of copies  
on paper, which were archived in Swedish university libraries  
like all other written works published in Sweden.*

*The publisher has taken technical and administrative measures  
to assure that the on-line version of the article will be  
permanently accessible using the URL stated above,  
unchanged, and permanently equal to the archived printed copies  
at least until the expiration of the publication period.*

*For additional information about the Linköping University  
Electronic Press and its procedures for publication and for  
assurance of document integrity, please refer to  
its WWW home page: <http://www.ep.liu.se/>  
or by conventional mail to the address stated above.*

## Abstract

Level set methods [OS88] 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  $1024^3$ .

## Authors’ Affiliations

Michael B. Nielsen  
University of Aarhus  
Department of Computer Science  
DK-8200 Aarhus N, Denmark  
E-mail: [bang@daimi.au.dk](mailto:bang@daimi.au.dk)

Ken Museth (corresponding author)  
Department of Science and Technology  
Linköping University  
SE-60174 Norrköping, Sweden  
E-mail: [kenmu@itn.liu.se](mailto:kenmu@itn.liu.se)  
Webpage: <http://www.gg.itn.liu.se>

# 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 extremely important problem in many different scientific fields ranging from physics and chemistry to computer vision and computer graphics. To mention just 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 [OS88, OF02, Set99] by Osher and Sethian to be very useful and robust [ROF92, SSO94, FAX99, ZOMK00, CV01, FF01, FSJ01, MBWB02, OP03]. 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 and temporal 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 simply 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 [SK98, FP03]. Finally, our data structure is completely 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 [OS88]. 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 [AS95], but the numerical implementation was based on a very wide band around the interface to avoid frequent and costly rebuilds. Later Whitaker [Whi98] proposed an approximate but faster narrow band scheme (dubbed “Sparse Field”) that only solves the level set PDE on the actual interface grid points arranged in linked lists. 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 [AS99] that preserves the approximate signed distance to the interface. Finally, Peng *et al.* [PMO<sup>+</sup>99] 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 [Whi98].

All of these narrow band schemes effectively address the problem of computational complexity present in the original level set formulation [OS88]. 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 [Str99, DMRS01, Min03] and more recently this idea was extended to surfaces by means of octrees [LGF04].

While tree based approaches do indeed reduce the memory footprint of the associated interface representation they are hampered by a number of other limitations. Most importantly the non-uniform discretization makes it non-trivial to accurately use higher order finite difference schemes like ENO [OS91] or WENO [LOC94] in space and the TVD Runge-Kutta methods [SO88] in time. Instead, these tree based methods often use a semi-Lagrangian scheme [Str99] 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 [Min03]. Another problem seems to be that tree based data structures are relatively slow to access and modify during interface propagations which can lead to significant reductions in performance when compared to regular narrow band schemes based on full uniform Cartesian grids. This problem is closely related to the fact that it can be rather tedious to implement optimized versions of these tree structures. Finally, we note that while tree data structures allow for multi-resolution representations, all practical tree methods appear to use uniform resolution of the interface [Str99, DMRS01, Min03, LGF04]. 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.

To the best of our knowledge, the only work directly related to ours is described in a technical sketch (one page abstract) by Houston *et al.* [HWB04] recently presented at a graphics conference, where we concurrently summarized the main features of DT-Grid [BFM<sup>+</sup>04]. 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 does not scale with 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 scales with 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 section 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 for the first time 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 section 3 we describe efficient algorithms that are fundamental to improved level set simulations on the DT-grid. In section 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, section 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 Figure 1. As a prelude to this description it is convenient to introduce the following general terminology based on the nomenclature given in table 1.

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

$N$	The Dimension.
$\mathbf{X}_N$	Grid point or p-column number $(x_1, x_2, \dots, x_N)$
$\phi(\mathbf{X}_N)$	Scalar level set function.
$\Omega^-$	Interior region.
$\Omega^+$	Exterior region.
$dx$	The uniform grid spacing.
$T_\alpha$	The tubular grid $\{\mathbf{X}_N \in \mathbb{R}^N \mid  \phi(\mathbf{X}_N)  < \alpha\}$ .
$M_N$	Number of grid points in the $ND$ tubular grid.
$\gamma$	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.

Table 1: Nomenclature used throughout the paper.

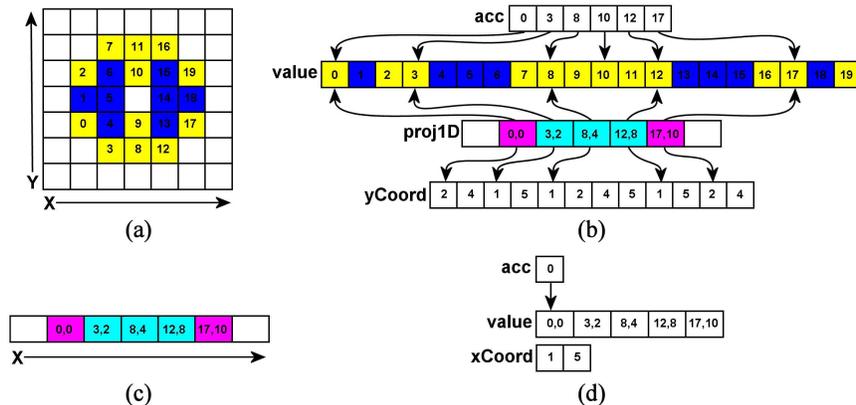


Figure 1: (a) A dense 2D grid. (b) Corresponding 2D DT-Grid. (c) A dense 1D grid. (d) Corresponding 1D DT-Grid. Note the lexicographic storage order.

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 figure 1.a, 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 figure 1.a 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 figure 1.{a,b} the grid

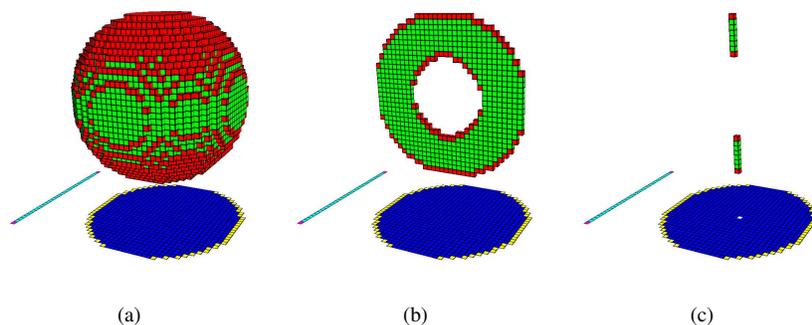


Figure 2: Color coded representation of the tubular grid of a sphere in a 3D DT-Grid. a) Entire sphere. b) Middle slice of sphere. c) P-column consisting of two connected components.

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 figure 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 figure 1.b. Also note that `proj1D` is defined recursively as a `DTGrid1D` with `Type=IndexPair`, see figure 1.{c,d}. The constituents of the 1D DT-Grid are defined similarly, except for the fact that it 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.

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. Figure 2.c shows p-column number (25, 25) consisting of two connected components. The white pixel in Figure 2.c 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 1) 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 section 4.3.

### 3 Algorithms of the Data Structure

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 m'th 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	$O(M_N)$

Table 2: Key algorithms of a N-dimensional DT-Grid. The associated time complexities are derived in subsequent sections.

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  $\Omega^+$  and negative in  $\Omega^-$ , 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 sections 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, section 3.5 describes how constant and logarithmic time neighbor access operations can be constructed. Finally, section 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 2 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 lexico-

graphically as  $(x_1, x_2, \dots, x_N)$ , new grid points must be pushed in this order<sup>1</sup>. 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.

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 section 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 figure 1.a.)
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 figure 1.a.)
3. The new grid point is the last in an existing connected component. (See the remaining colored grid points in figure 1.a.)

### 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 section 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  $O(1)$ .

### 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. A non-center stencil grid point may not exist in the tubular grid. If this is the case,

---

<sup>1</sup>i.e. in 3D, `push(2,2,6)` should be issued before `push(2,5,1)`.

access to that particular stencil grid point returns  $-\gamma$  if the grid point lies in  $\Omega^-$  and  $\gamma$  otherwise.

Narrow band level set algorithms typically operate on a number of concentric tubes of increasing width centered about the interface, see [AS95, PMO<sup>+</sup>99]. 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 [Tsi94, Tsi95, Set96], 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 N'th coordinate,  $x_N$ , lies between the min and max N'th 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 N'th coordinate direction.
4. Finally it is determined whether the grid point actually exists in the p-column (ie. 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  $\Omega^-$  and  $\gamma$  if the grid point lies in  $\Omega^+$ . The time complexity of random access to a grid point,  $\mathbf{X}_N = (x_1, x_2, \dots, x_N)$ , 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  $\mathbf{X}_n$ .

Employing this 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  $N$ 'th coordinate of the first grid point in the connected component in which the grid point lies. The last component, `Xn`, is the  $N$ 'th 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  $m$ 'th 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` $\pm 1$ , respectively. If the particular neighbor does not exist in the tubular grid,  $\gamma$  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` $\pm 1$  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  $m$ 'th coordinate direction in a  $N$ -dimensional DT-Grid takes time  $O(1 + \sum_{n=m}^{N-1} \log C_{X_n})$ .

### 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 [Tsi94, Tsi95, Set96] the tubular grid is rebuilt simultaneously with the reinitialization process. The remainder of this section is organized as follows: Section 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 section 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.

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).

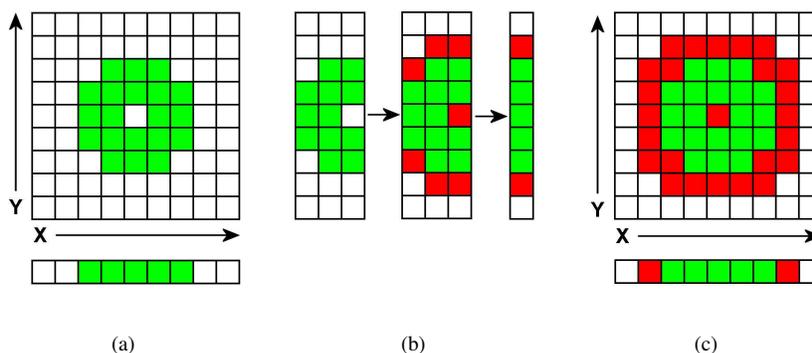


Figure 3: a) The original DT-Grid. b) The process of computing a new p-column (number 3) in the dilated DT-Grid (adding the red grid points) c) The final dilated DT-Grid.

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.

An illustration of a tubular grid dilation in 2D is depicted in figure 3. Figure 3.a shows an initial 2D DT-Grid as well as its 1D DT-Grid constituent. Figure 3.c 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 merging adjacent and overlapping connected components into a single connected component, see figure 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 figure 3.c.

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  $x-H, \dots, x-1, x, x+1, \dots, x+H$  independently in the Y direction by  $H$  grid points. Next form their union in order to obtain p-column number  $x$  in the dilated 2D tubular grid. This is illustrated in figure 3.b. 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  $T_\alpha$ , to include all grid points within a distance  $\alpha$  from the interface. The algorithm devised here assumes that the original tubular grid is a distance field and contains all grid points in  $T_\delta$ , 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 [OF02], 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  $T_\delta$ . In practice

this is done by constructing an intermediate tubular grid and copying all grid points within  $T_\delta$  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  $T_\alpha$  and  $T_\delta$ , using the tubular grid dilation algorithm of section 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, section 4.1 compares the memory- and time-usage of the DT-Grid to existing methods. Next, section 4.2 verifies that the low memory footprint of the DT-Grid allows for very high resolution simulations. Finally, section 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 [PMO<sup>+</sup>99]

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

where  $\kappa$  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 figure 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 [LOC94] discretization for the spatial hyperbolic term ( $\bar{\kappa}|\nabla\phi|$ ). We use a third order accurate TVD Runge-Kutta [SO88] time-stepping method. In the reinitialization step we solve the PDE of Peng et al. [PMO<sup>+</sup>99] 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 figure 5 and 7 we compare the following methods: 1) The DT-Grid (blue), 2) The method of Peng et al. [PMO<sup>+</sup>99] (green), 3) The method of Peng et al. improved with an  $O(M_N)$  method for rebuilding the narrow band [BFM<sup>+</sup>04] (red), 4) An Octree data structure [Sam90, SK98, FP03] (cyan) storing only the grid points inside the narrow band using a uniform resolution. We describe the Octree implementation in more detail in appendix A.

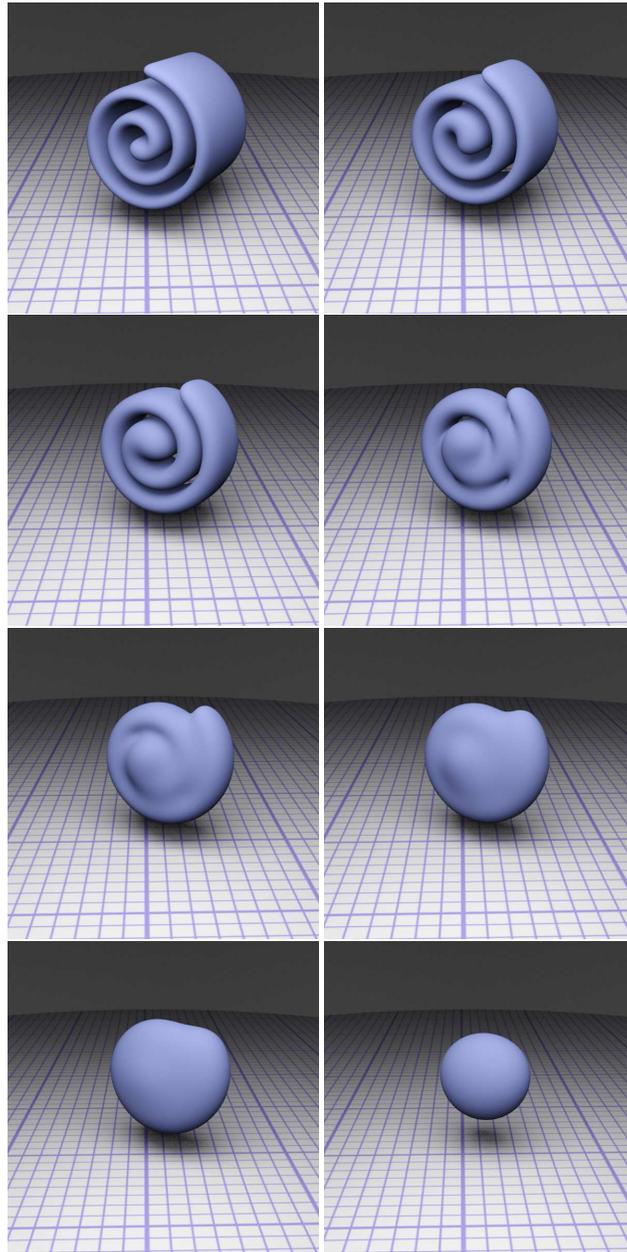


Figure 4: Extruded spiral evolving under volume conserving mean curvature flow. Benchmark test for narrow band, Octree and DT-Grid level set methods. Timings and memory usage is given in figures 5 to 8.

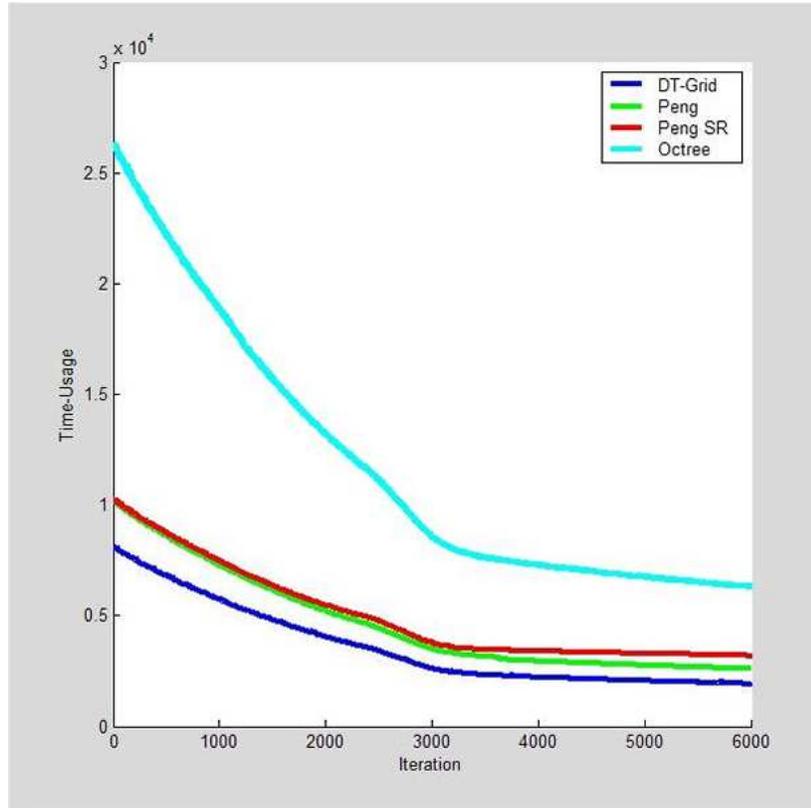


Figure 5: Time usage with HJ WENO RK3 for both advection and remormalization.

Figure 5 shows the time-usage of the different methods. 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 figure 7, 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 [PMO<sup>+</sup>99]. 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 figure 7). 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. 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 [Sam90] 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 section 4.2 we demonstrate a simulation on a high resolution grid, where the memory consumptions of the DT-Grid and the method

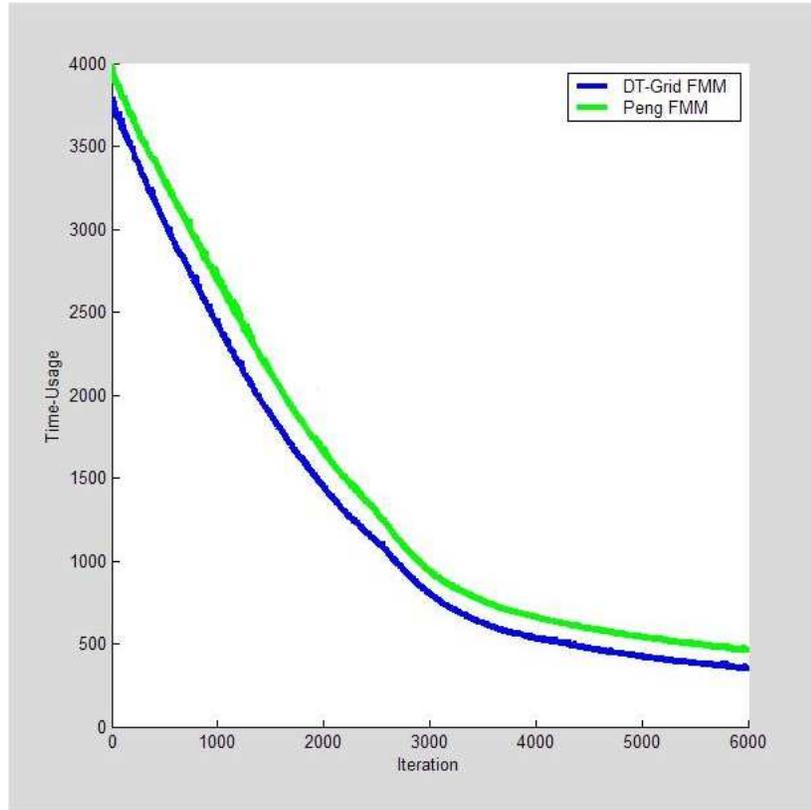


Figure 6: Time usage with HJ WENO RK3 for advection and FMM for renormalization.

of Peng are 67.2MB and 5.2GB respectively.

Figure 8 illustrates the efficiency of the index compression scheme of the DT-Grid. We have plotted the total amount of memory occupied by 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.* [PMO<sup>+</sup>99] 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 [Set99].

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 figure 6. 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

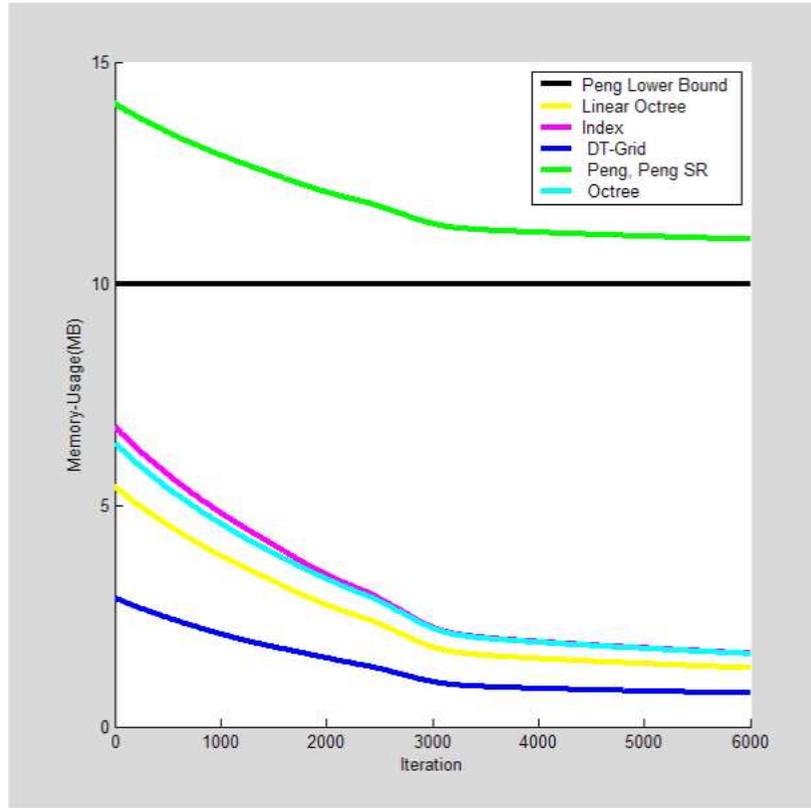


Figure 7: Memory Usage.

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.* [EFFM02] introduced a test based on a three dimensional incompressible flow field initially proposed by LeVeque [Lev96] 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.* [ELF04] 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 figure 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.* [EFFM02] which uses HJ-WENO and TVD Runge-Kutta for both the advection and reinitialization steps. This particular example clearly demonstrates that the DT-Grid enables high resolution interfaces to be represented. To the best of our knowledge this is the largest level set simulation to date.

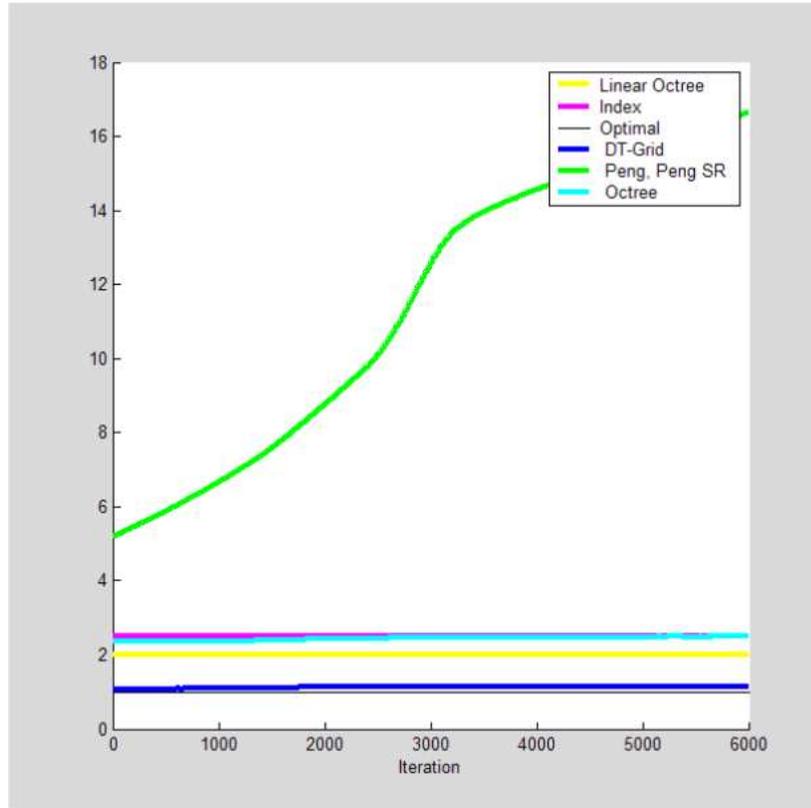


Figure 8: Efficiency of index compression.

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 5.2GB of storage. Other non-hierarchical approaches to storing the tubular grid, such as storing the indices explicitly 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 MB 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, in contrast to full grids and Octrees, 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 figure 10(a) using convection-diffusion [OF02]. 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 figure 10(a)), and we force its contribution to zero over time. The radial vector field advects the level

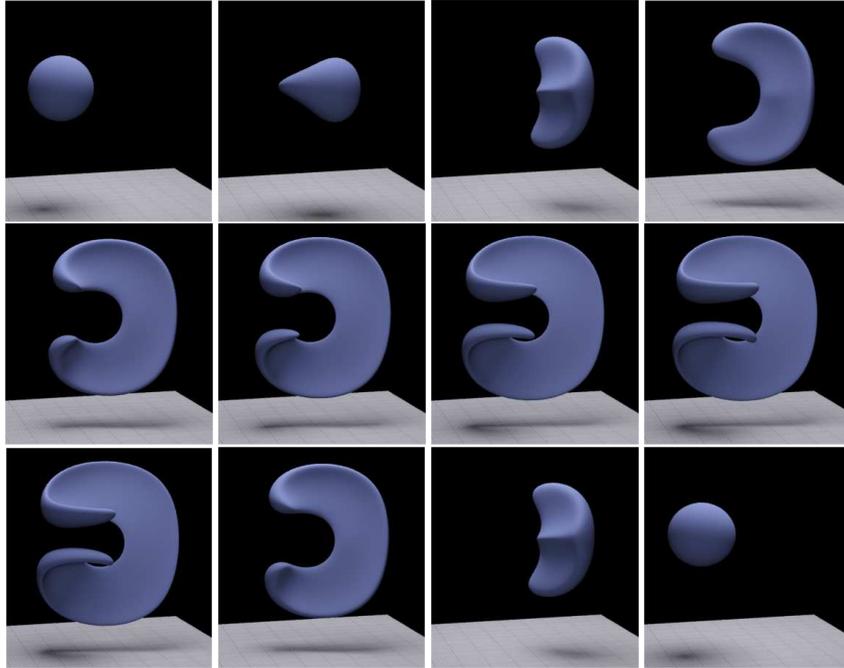


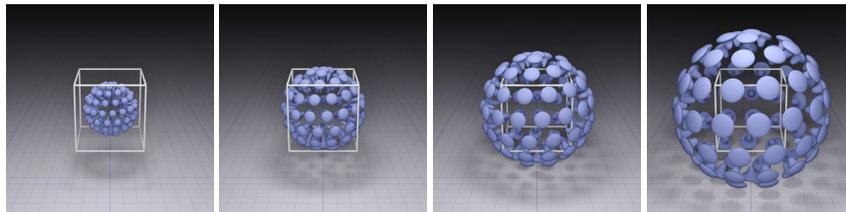
Figure 9: The DT-Grid enables high resolution grids to be represented with a very low memory footprint. In this example the resolution was  $1024^3$  and the maximal memory usage 67.2MB. The same simulation run with the method of Peng *et al.* [PMO<sup>+</sup>99] would consume 5.2GB of storage.

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.

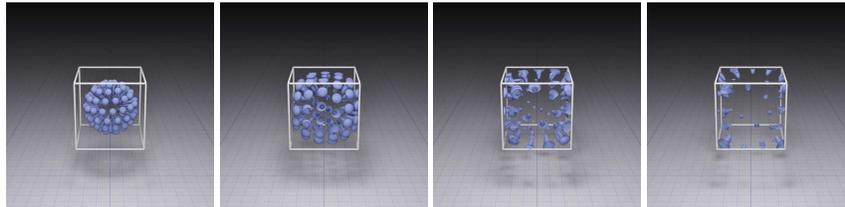
Figure 10(b) illustrates what happens on full grids and Octrees. As the level set moves beyond the boundary, it disappears. In contrast, the level set on the DT-Grid (figure 10(a)) moves beyond the box as the *grid is not bounded*. The memory usage in figure 10(a) is 14.0MB, 25.2MB, 39.3MB and 64.1MB, 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 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.

Out-of-the-box level set simulations are unique to the DT-Grid and extremely 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 [GFCO03].



(a) DT-Grid



(b) Full Grid and Octrees

Figure 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.

## 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 for the first time: 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, 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.

The DT-Grid is not an adaptive representation in its traditional sense since the interface 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.

## 6 Acknowledgements

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.

## A Octree Implementation

In section 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.*[SK98] to traverse the Octree sequentially and the neighbor access algorithms of Frisken *et al.*[FP03]. Octree cells are allocated in a memory pool [Mey97] 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

- [AS95] D. Adalsteinsson and J.A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, 1995.
- [AS99] D. Adalsteinsson and J.A. Sethian. The fast construction of extension velocities in level set methods. *J. Computational Physics*, 148:2–22, 1999.
- [BFM<sup>+</sup>04] D. Breen, R. Fedkiw, K. Museth, S. Osher, G Sapiro, and R. Whitaker. *Level Sets and PDE Methods for Computer Graphics*. ACM SIGGRAPH '04 COURSE #27. Los Angeles, CA, August 2004. ISBN 1-58113-896-2.
- [CV01] T. Chan and L. Vese. Active contours without edges. *IEEE Trans. on Image Processing*, 10:266–277, 2001.
- [DMRS01] M. Droske, B. Meyer, M. Rumpf, and C. Schaller. An adaptive level set method for medical image segmentation. *Lecture Notes in Computer Science*, pages 412–422, 2001.

- [EFFM02] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.*, 183(1):83–116, 2002.
- [ELF04] Douglas Enright, Frank Losasso, and Ronald Fedkiw. A fast and accurate semi-lagrangian particle level set method. *Computers and Structures*, 2004.
- [FAX99] R. Fedkiw, T. Aslam, and S. Xu. The ghost fluid method for deflagration and detonation discontinuities. *Journal of Computational Physics*, 154:393–427, 1999.
- [FF01] N. Foster and R. Fedkiw. Practical animation of liquids. In *ACM SIGGRAPH '01*, pages 23–30. ACM Press, 2001.
- [FP03] S. Frisken and R. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3), 2003.
- [FSJ01] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In *ACM SIGGRAPH '01*, pages 15–22. ACM Press, 2001.
- [GFCO03] Frédéric Gibou, Ronald Fedkiw, Russel Caflisch, and Stanley Osher. A level set approach for the numerical simulation of dendritic growth. *J. Sci. Comput.*, 19(1-3):183–199, 2003.
- [HWB04] Ben Houston, Mark Wiebe, and Chris Batty. Rle sparse level sets. In *Proceedings of the SIGGRAPH 2004 conference on Sketches & applications*. ACM Press, 2004.
- [Lev96] Randall J. Leveque. High-resolution conservative algorithms for advection in incompressible flow. 33(2):627–665, April 1996.
- [LGF04] Frank Losasso, Frédéric Gibou, and Ronald Fedkiw. Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH '04*, pages 457–462. ACM Press, 2004.
- [LOC94] X.D. Liu, S.J. Osher, and T. Chan. Weighted essentially nonoscillatory schemes. *J. Comput. Phys.*, 115:200–212, 1994.
- [MBWB02] K. Museth, D. Breen, R. Whitaker, and A. Barr. Level set surface editing operators. *ACM Trans. on Graphics (Proc. SIGGRAPH)*, 21(3):330–338, July 2002.
- [Mey97] Scott Meyers. *Effective C++ (2nd ed.): 50 specific ways to improve your programs and designs*. Addison-Wesley Longman Publishing Co., Inc., 1997.
- [Min03] Chohong Min. Local level set method in high dimension and codimension. Technical Report CAM 03-67, UCLA, November 2003.
- [OF02] S.J. Osher and R.P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, Berlin, 2002.
- [OP03] S. Osher and N. Paragios, editors. *Geometric Level Set Methods in Imaging, Vision and Graphics*. Springer Verlag, 2003.
- [OS88] S. Osher and J. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

- [OS91] S. Osher and C.W. Shu. High-order essentially nonoscillatory schemes for hamilton-jacobi equations. *SIAM J. Num. Anal.*, 28:907–922, 1991.
- [PMO<sup>+</sup>99] Danping Peng, Barry Merriman, Stanley Osher, Hongkai Zhao, and Myungjoo Kang. A pde-based fast local level set method. *J. Comput. Phys.*, 155(2):410–438, 1999.
- [ROF92] L. Rudin, S. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D*, 60:259–268, 1992.
- [Sam90] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, GIS*. Addison-Wesley, Reading, MA, 1990.
- [Set96] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. of the National Academy of Sciences of the USA*, 93(4):1591–1595, February 1996.
- [Set99] J.A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, Cambridge, UK, second edition, 1999.
- [SK98] N. Stolte and Ari Kaufman. Parallel spatial enumeration of implicit surfaces using interval arithmetic for octree generation and its direct visualization. In *Implicit Surfaces '98*, pages 81–87, 1998.
- [SO88] C.W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock capturing schemes. *J. Comput. Phys.*, 77:439–471, 1988.
- [SSO94] M. Sussman, P. Smereka, and S. Osher. A level set approach to computing solutions to incompressible two-flow. *Journal of Computational Physics*, 114:146–159, 1994.
- [Str99] John A. Strain. Tree methods for moving interfaces. *Journal of Computational Physics*, 151, 1999.
- [Tsi94] J. N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *Proceedings of the 33rd Conference on Decision and Control, Lake Buena Vista, LF*, pages 1368–1373, December 1994.
- [Tsi95] J. N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Trans. Automat. Contr.*, 40:1528–1538, September 1995.
- [Whi98] R.T. Whitaker. A level-set approach to 3d reconstruction from range data. *Int. J. Comput. Vision*, 29(3):203–231, 1998.
- [ZOMK00] H. Zhao, S. Osher, B. Merrian, and M. Kang. Implicit and non-parametric shape reconstruction from unorganized data using a variational level set method. *Computer Vision and Image Understnading*, 80:294–314, 2000.