Out-of-Core and Compressed Level Set Methods

MICHAEL B. NIELSEN
University of Århus
OLA NILSSON
University of Linköping
ANDREAS SÖDERSTRÖM
University of Linköping
and
KEN MUSETH
University of Linköping and University of Århus

This article presents a generic framework for the representation and deformation of level set surfaces at extreme resolutions. The framework is composed of two modules that each utilize optimized and application specific algorithms: 1) A fast *out-of-core* data management scheme that allows for resolutions of the deforming geometry limited only by the available disk space as opposed to memory, and 2) compact and fast *compression* strategies that reduce both offline storage requirements and online memory footprints during simulation. Out-of-core and compression techniques have been applied to a wide range of computer graphics problems in recent years, but this article is the first to apply it in the context of level set and fluid *simulations*. Our framework is generic and flexible in the sense that the two modules can transparently be integrated, separately or in any combination, into existing level set and fluid simulation software based on recently proposed narrow band data structures like the DT-Grid of Nielsen and Museth [2006] and the H-RLE of Houston et al. [2006]. The framework can be applied to narrow band signed distances, fluid velocities, scalar fields, particle properties as well as standard graphics attributes like colors, texture coordinates, normals, displacements etc. In fact, our framework is applicable to a large body of computer graphics problems that involve sequential or random access to very large co-dimension one (level set) and zero (e.g. fluid) data sets. We demonstrate this with several applications, including fluid simulations interacting with large boundaries ($\approx 1500^3$), surface deformations ($\approx 2048^3$), the solution of partial differential equations on large surfaces ($\approx 4096^3$) and mesh-to-level set scan conversions of resolutions up to $\approx 35000^3$ (7 billion voxels in the narrow band). Our out-of-core framework is shown to be several times faster than current state-of-the-art level set data structures relying on OS paging. In particular we show sustained throughput (grid points/sec) for gigabyte sized level sets as high as 65% of state-of-the-art throughput for in-core simulations. We also demonstrate that our compression techniques out-perform state-of-the-art compression algorithms for narrow bands.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Animation; Curve, Surface, Solid and Object Representations*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Physically Based Modeling*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Animation*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Level set methods, out-of-core, compression, implicit surfaces, adaptive distance fields, computational fluid dynamics, geometric modeling, shape, morphology, mesh scan conversion, deformable surfaces, compression, streaming

ACM Reference Format:

Nielsen, M. B., Nilsson, O., Söderström, A., and Museth, K. 2007. Out-of-core and compressed level set methods. ACM Trans. Graph. 26, 4, Article 16 (October 2007), 26 pages. DOI = 10.1145/1289603.1289607 http://doi.acm.org/10.1145/1289603.1289607

1. INTRODUCTION

Implicit modeling has been around almost since the dawn of computer graphics. Such models represent geometry as iso-surfaces of some volumetric scalar function. The fact that the geometry is defined by functions embedded in a higher-dimensional space accounts for many of the attractive properties of implicit geometry. Implicit geometry can easily change topology (merge or bifurcate) and surface properties are readily derived from the embedding functions. However, for many years implicit geometry was considered inferior

Michael Bang Nielsen was partially supported by Center for Interactive Spaces under ISIS Katrinebjerg and the Danish Agency for Science, Technology and Innovation. Ken Museth acknowledges funding from both the University of Linköping and the Swedish Science Foundation to support respectively Ola Nilsson and Andreas Söderström.

Authors' addresses: M. B. Nielson {bang@daimi.au.dk}; O. Nilsson {olani@itn.liu.se}; A. Söderström {andso@itn.liu.se}; K. Museth {museth@acm.org}.

to explicit representations like parametric NURBS and in particular multiresolution subdivision meshes. While some might argue this is still the case, the so-called level set method [Osher and Sethian 1988] has proven very successful for modeling and animation with implicit and dynamic geometry. Graphics related examples that use level sets include fluid animations [Foster and Fedkiw 2001; Enright et al. 2002], geometric modeling [Museth et al. 2002; 2005], fire [Nguyen et al. 2002], shape reconstruction [Zhao et al. 2001], and metamorphosis [Breen and Whitaker 2001].

The level set method, originating from interface studies in applied mathematics [Osher and Sethian 1988], provides a mathematical toolbox that allows for direct control of surface properties and deformations. Although not required by the level set method, surfaces are typically represented implicitly by their sampled signed distance field, ϕ, and deformations are imposed by solving a time-dependent partial differential equation (PDE) by means of finite difference (FD) schemes [Osher and Fedkiw 2002; Sethian 1999]. To obtain a computational complexity that scales with the area of the surface, as opposed to the volume of its embedding, several narrow band schemes have been proposed [Adalsteinsson and Sethian 1995; Whitaker 1998; Peng et al. 1999]. These schemes exploit the fact that the zero level set of ϕ uniquely defines the surface and hence the PDE only needs to be solved in a narrow band around $\phi = 0$. While these methods decrease the computational complexity, memory still scales with the volume of the embedding. In recent years a number of improved data structures have addressed this issue and dramatically reduced the memory footprints of level sets, hereby allowing for the representation of geometry at higher resolutions. This includes the use of tree structures [Strain 1999; Min 2004; Losasso et al. 2004; 2005], blocked grids [Bridson 2003], blocked grids on the GPU [Lefohn et al. 2003], dynamic tubular grids (DT-Grid) [Nielsen and Museth 2004b, 2004a, 2006] as well as run-length encoding applied along a single dimension [Houston et al. 2004] and hierarchically (H-RLE) [Houston et al. 2006].

Despite the introduction of these data structures, current level set representations still have their limitations. A significant issue continues to be the restriction on model resolution when compared to state-of-the-art explicit representations. While it is not unusual to encounter out-of-core meshes today with several hundred millions of triangles,[1] the same level of detail is yet to be demonstrated with level set representations. Recent advances in level set data structures have indeed increased the potential resolution of level set surfaces, but they do not employ compression of the numerical values inside the narrow band[2] and they only work in-core. Consequently, current level set methods are effectively limited by the available main memory. Given the fact that level set and fluid simulations typically require additional storage for auxiliary fields (e.g., particles, scalars, velocities and pressure), this in turn imposes significant limitations on the practical resolutions of deformable models. These facts have motivated the work presented in this paper.

We have developed a framework that allows for representations and deformations of level set surfaces, fluid velocities, and additional fields at extreme resolutions. Our general approach is to employ new application-specific out-of-core prefetching and page-replacement schemes combined with new compression algorithms. The out-of-core component allows us to utilize the available disk space by streaming level sets to and from disk during simulation.

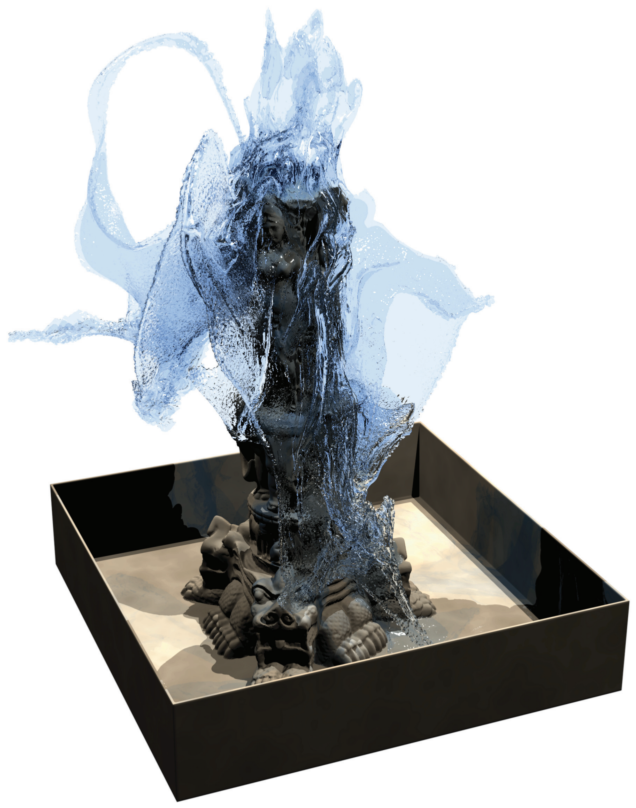
Fig. 1. Fountain fluid animation using our out-of-core framework. See section 9.3 for details.

In addition the compression component effectively reduces both offline storage requirements and online memory footprints during simulation. Reducing offline storage requirements is important in level set and fluid simulations since they typically produce large amounts of (temporal) data needed for postprocessing like direct ray tracing, higher order mesh extraction, motion blur, simulation restarts, and so on. While out-of-core and compression techniques are certainly not new in computer graphics, to the best of our knowledge we are the first to employ them for level set deformations and fluid animations.

Out-of-core algorithms are generally motivated by the fact that hard disks are several orders of magnitude cheaper and larger than main memory [Toledo 1999], thus pushing the limits of feasible computations on desktop computers. This trend continues despite the introduction of 64-bit operating systems allowing for larger address spaces, mainly due to the high cost of main memory. For example, using our framework we have performed out-of-core scan conversions of huge meshes on a desktop computer with 1 GB of memory that would require close to 150 GB of main memory if run in-core. Finally, for example, when algorithms are CPU bound, the performance of carefully designed out-of-core implementations can be close to in-core counterparts.

We have chosen to build our out-of-core and compression framework on a custom implementation of the narrow band DT-Grid data structure by Nielsen and Museth [2006]. This data structure has the advantage that for surfaces, both the computational complexity and storage requirements scale linearly with the size of the interface as opposed to the volume of the embedding. Furthermore, DT-Grid

[1]The St. Matthew [Levoy et al. 2000] model for example has more than 186 million vertices and takes up more than 6GB of storage

[2]Note that though H-RLE is based on run-length encoding it does *not* compress inside the narrow band

can readily be extended to accommodate volumetric attributes for fluid simulations in which case the scaling is limited to actual fluid voxels as opposed to the usual bounding-box. The data structure is particularly designed for sequential stencil access into the data structure. When streaming these data structures to and from disk this becomes an important feature. Finally, DT-Grid separates topology and numerical values, and more significantly has been demonstrated to out-perform both octrees and the H-RLE [Houston et al. 2006].

The out-of-core component of our framework is generic in the sense that it can easily be integrated with existing level set modeling and simulation software based on for example DT-Grid, H-RLE, or normal full grid representations. However, the sparse representations are preferable since they limit the amount of data that must be processed. Consequently, existing level set simulation code is not required to be rewritten in order to use our framework. Our compression schemes are optimized for the DT-Grid representation, but they can readily be modified to work on other narrow-band data structures like the H-RLE. The framework is flexible since the out-of-core and compression components can be integrated separately or in combination. In addition, the out-of-core framework can be applied to narrow band signed distances, fluid velocities, scalar fields, matrices, particle properties as well as standard graphics attributes like colors, texture coordinates, normals, displacements, and so on. No specialized hardware is required, but our framework does of course benefit from fast disks or disk arrays.

Our framework allows us to both represent and deform level set surfaces with resolutions and narrow band voxel counts higher than ever before documented. We will also demonstrate that our out-of-core framework is several times faster than current state-of-the-art data structures relying on OS paging and prefetching for models that do not fit in main memory. Naturally, our framework does not perform as fast as state-of-the art data structures for deformations that fit in memory. However, we obtain a performance that is as high as 65% of peak in-core performance. Remarkably, this 65% throughput (measured in processed grid points per second) is sustained even for models of several gigabytes that do not fit in memory. In addition, we show that our compression techniques outperform related state-of-the-art compression algorithms for compressing partial volume grids, that is, narrow bands of volumetric data.

We emphasize that while several of the techniques presented in this paper are probably applicable for large-scale scientific computing, this is not the main focus of our work. Instead we are targeting computer graphics applications—more specifically high-resolution level set and fluid simulations—on standard desktop computers. All the examples in this paper were produced on desktop machines with 1 or 2 GB of RAM. In spite of this we note that the grid sizes we are able to achieve on desktop machines are high even when compared to many super-computing simulations. For example, Akcelik et al. [2003] employed an unstructured grid with 4 billion cells to simulate earthquakes on 3000 AlphaServer processors. In comparison our largest Lucy statue scan conversion contains 7 billion grid points in the narrow band.

To demonstrate the versatility and significance of our novel framework we include several graphics applications. This includes high-resolution fluid simulations interacting with large boundaries, high-resolution surface deformations such as shape metamorphosis and the solution of partial differential equations on 2-manifolds. Also, to produce high resolution input to our out-of-core and compressed simulations we have developed a new mesh to level set scan converter that is limited only by the available disk space with regard to both the size of the input mesh and the output level set. An explicit list of contributions and an outline of this paper is given in the next section.

2. CONTRIBUTIONS AND OUTLINE

Our main contribution is the development of a generic and flexible framework for the representation and deformation of level set surfaces and auxiliary data out-of-core. Specifically, this framework offers the following technical features:

—Near optimal *page-replacement* and fast *prefetching* algorithms designed for sequential access with finite difference stencils used during simulation. Our algorithms outperform state-of-the-art level set data structures relying on OS paging and prefetching.

—Fast and compact *compression* schemes for narrow band level sets that work both online and offline.

—Fast out-of-core *particle level set* data structures and compression of particle data.

We also claim the following contributions based on novel applications of our framework:

—Partially out-of-core *fluid animations*. Using our framework we represent boundaries, surface velocities as well as the particle level set based fluid surface out-of-core, allowing us to simulate fluids interacting with boundaries at extreme resolutions.

—Out-of-core data structures and algorithms for *linear algebra*.

—Out-of-core *simulations of PDEs* embedded on large 2-manifolds. In particular we solve the wave equation on surfaces with more than hundred million voxels.

—Out-of-core polygonal mesh to level set *scan conversion*. Our method is only limited by the available disk space with respect to the size of the input mesh and the output level set. For instance we generate level sets with up to 7 billion voxels in the narrow band.

The rest of this article is organized as follows: Section 3 describes related work in the areas of compression and out-of-core algorithms. Next, Section 4 introduces the basic terminology and structure of our framework. Sections 5 and 6 describe the out-of-core and compression components of the framework in detail, and Section 7 proposes an out-of-core particle level set method utilizing compressed particles. Subsequently Section 8 justifies our claims and demonstrates the efficiency of our framework. Finally, Section 9 demonstrates several applications of our framework, and Section 10 concludes the paper and proposes some new directions for future work.

3. PREVIOUS WORK

Our framework is based on two techniques that are well known in the field of computer science: compression and out-of-core methods. As such there is a large body of related work and for the sake of clarity we shall review this work as two separate topics. However, we stress that our work stands apart from this previous work in several ways. Most importantly we are the first to design and apply these techniques to level set methods. Consequently, most of the work described here is not directly related to ours.

3.1 Compression Methods

This paper deals with *narrow bands* of volumetric data. Mesh compression methods on the other hand (see Kälberer et al. [2005] and references therein compress only the surface itself and possibly the normals. Even though it is indeed feasible to compute differential properties from meshes [Desbrun et al. 1999], this is generally not an optimal storage format for implicit surfaces like level sets. The reason is primarily that the map between the implicit level set and the explicit mesh is not guaranteed to be bijective. Consequently, important information is lost by converting to the mesh representation using Lorenson and Cline [1982]; Kobbelt et al. [2001]; Ju et al. [2002], and this information is not recoverable by a subsequent mesh to level set scan conversion. Primarily this regards higher order differential properties. Methods with high compression ratios have also been proposed for iso-surface meshes of volumetric scalar fields [Taubin 2002; Lee et al. 2003; Eckstein et al. 2006]. However, again information is lost for our purposes, and furthermore these methods work only in-core and consider all grid points in the bounding box, not just in a narrow band. Converting a narrow-band volume grid to a clamped dense volume grid is straightforward and makes it possible to employ existing volume compression methods [Nguyen and Saupe 2001; Ibarria et al. 2003]. However, for large dense grids this approach is far from optimal in terms of compression performance, memory and time usage. Note that the method for compressing surfaces represented as signed distance fields by Laney et al. [2002] also operates on the entire volume. In particular the method employs a wavelet compression of the signed distance field and applies an aggressive thresholding scheme that sets wavelet coefficients to zero if their support does not include the zero crossing. Thus, this method may actually discard information very close to the interface hence preventing higher order accurate content.

The work on compression most related to ours is the method for compressing unstructured hexahedral volume meshes by Isenburg and Alliez [2002] and the work on volumetric encoding for streaming iso-surface extraction by Mascarenhas et al. [2004]. Isenburg and Alliez consider in-core compression that separately compresses topology and geometry. Mascarenhas et al. later extended the method of Isenburg and Alliez to also compress scalar grid values and additionally proposed an out-of-core decoder. The method is applied to structured uniform grids and used in the context of streaming iso-surface extraction. More specifically, the grid is partitioned into *partial volume grids* in such a way that a bound on the ratio between the number of grid cells loaded and the number of grid cells intersecting any iso-surface is guaranteed. This approach is not suitable for online, or batched, simulations. In particular, it is not feasible to employ Mascarenhas et al. [2004] to online compressed simulations, since a stencil of neighboring grid points, used for finite difference computations, must be available. Nevertheless, in the results section we compare our compression method to Isenburg and Alliez [2002] and Mascarenhas et al. [2004] as an offline compression method for reducing storage requirements of the produced data.

3.2 Out-of-Core Methods

The field of out-of-core methods, also referred to as "external memory algorithms," is large and actually dates back as far as the fifties—not long after the emerge of digital computers. Out-of-core techniques are applicable in a wide range of problems where data intensive algorithms are ubiquitous. This includes image repositories, digital libraries, relational and spatial databases, computational geometry, simulation, linear algebra, and computer graphics. For a recent survey of the entire field, see Vitter [2001]. For the interested reader we refer to Toledo [1999] for a specific survey in linear algebra and simulation, and Silva et al. [2002] for a survey that focuses on computer graphics.

In computer graphics, out-of-core methods have been applied to a wide range of problems including iso-surface extraction [Mascarenhas et al. 2004; Yang and Chiueh 2006], compression of meshes [Isenburg and Gumhold 2003] and scalar fields [Ibarria et al. 2003], streaming compression of triangle meshes [Isenburg et al. 2005], stream processing of points [Pajarola 2005], mesh editing and simplification [Cignoni et al. 2003], and visualization [Cox and Ellsworth 1997; Gobbetti and Marton 2005; Cignoni et al. 2003].

Various approaches have been proposed for improving the access efficiency to out-of-core multidimensional grids during computation or for optimizing online range-queries in areas such as scientific computing, computational fluid dynamics, computational geometry and visualization. This includes blocking techniques [Seamons and Winslett 1996], reblocking, and permutation of dimensions [Krishnamoorthy et al. 2004], as well as the exploitation of the properties of modern disks [Schlosser et al. 2005]. In computer graphics, improved indexing schemes for full three dimensional grids were proposed by Pascucci and Frank [2001] in the context of planar subset visualization. The above techniques all deal with full grids whereas we consider topologically complex narrow bands of grid data. Furthermore, our method does not require the layout of data on disk to be changed.

We are not the first to apply out-of-core techniques for online simulation. Pioneering work was done by Salmon and Warren [1997] for N-body simulation in astrophysics. Their work was based on trees and applied reordered traversals and a Least-Recently-Used page-replacement policy for efficiency. More recently, an out-of-core algorithm for Eulerian grid based cosmological simulation was proposed by Trac and Pen [2006]. Global information is computed on a low resolution grid that fits entirely in memory, whereas local information is computed on an out-of-core high resolution grid tiled into individual blocks that fit into memory. The individual blocks are loaded and simulated in parallel for a number of time steps and then written back to disk. These previous methods are, however, not directly applicable to simulations on narrow band data structures of level sets.

There is also a large body of work on general purpose page-replacement and prefetching strategies developed for operating systems, scientific applications, data bases, web servers, etc. General purpose algorithms for page-replacement must meet many requirements including simplicity, good performance and adaptivity to changing and mixed access patterns. In contrast, the our proposed techniques are *application-specific* and hence designed to work close-to-optimal for particular problems. For an introduction to standard page-replacement techniques like Least Recently Used (LRU), Most Recently Used (MRU) and Least Frequently Used (LFU) see the excellent book by Tanenbaum [1992]. For these classical techniques it is simple to derive examples where the given page-replacement policy will not perform optimally for our application. This is also the case for several more advanced schemes like LRU-K [O'Neil et al. 1993], LFRU [Lee et al. 1999], 2-Queue [Johnson and Shasha 1994], LIRS [Jiang and Zhang 2002], Multi-Queue [Zhou et al. 2004], and FBR [Robinson and Devarakonda 1990]. This will be motivated and explained in more detail in Section 5.

Another category of recent general purpose page-replacement strategies exploits the regularity of the access patterns for a given application. Based on the results of an access analysis, a specific page-replacement algorithm is chosen. Work in this category includes the sequential and loop access pattern detection method, UBM, by Kim et al. [2000], application/file-level characterization of access patterns by Choi et al. [2000], Early Eviction LRU by Smaragdakis et al. [1999], SEQ by Glass and Cao [1997], ARC

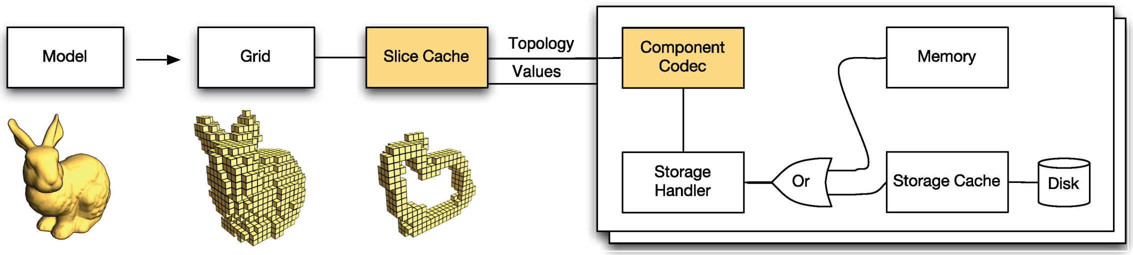

Fig. 2. The generic framework. The colored boxes represent components that exclusively form part of the compression framework.

by Megiddo and Modha [2003], and CAR by Bansal and Modha [2004]. However, the task of automatic access pattern detection is difficult, and page-replacement decisions may actually hurt performance when the estimate is incorrect [Brown 2005]. Furthermore, it takes some time for these prediction methods to actually start working, as an analysis is required before the appropriate strategy can be initiated.

Similarly to page-replacement techniques, a lot of effort has been put into developing general purpose prefetching methods. Consult Brown [2005] for a recent overview. Patterson et al. [1995], for example, describe a general purpose resource management system that balances hinted[3] and unhinted caching with hinted prefetching using cost-benefit analysis. There are several reasons why this framework is not feasible for our application. For example, the access patterns of our level set and fluid applications are not easily specified as hints in their system. In addition, explicit timings for disk access and disk driver times are required for the cost-benefit analysis. Brown [2005] focuses on a fully automatic system for prefetching by combining automatically generated compiler-inserted hints with a runtime layer and extensions to the operating system.

In contrast to all the work mentioned above, we focus on one particular application (level sets) for which the general structure of the access patterns is known in advance. Hence we can exploit this to develop a close-to-optimal strategy that is easy to implement and lightweight, so as to incur minimal performance overhead by avoiding costly online analysis.

Other examples of application-aware caches include the out-of-core mesh of Isenburg and Gumhold [2003], the work on application controlled demand paging for out-of-core visualization by Cox and Ellsworth [1997], and the octant caching on the *etree* by Lopez et al. [2004].

Finally, in-core scan conversion algorithms for converting triangular meshes to signed distance fields have been in use for quite a while [Mauch 2003]. However, to the best of our knowledge, no previous attempts have been made at out-of-core scan conversion algorithms. The work that comes closest to ours is the algorithm for generating out-of-core octrees on desktop machines by Tu et al. [2004].

4. OUT-OF-CORE LEVEL SET FRAMEWORK

An overview of our generic framework is illustrated in Figure 2, and we will briefly describe the components from left to right: The *Model* is represented as a level set sampled on a *Grid*. The *Slice Cache* allows for fast implementations of both sequential and random access to grid points in a local stencil. The Slice Cache stores a number of 2D slices of the 3D Grid topology, values, or both as illustrated with the bunny example. As the simulation or compression progresses through the grid, these slices are modified and replaced by the framework. If sufficient memory is available, the Slice Cache is stored in main memory to increase performance, otherwise it can be stored partially on the disk using the out-of-core framework. The staggered rectangular boxes shown on the right illustrate the fact that our framework separately stores the *topology* and numerical *values* of the grid as well as any *auxiliary fields*. This adds efficiency and flexibility to the framework: For example, since topology typically requires less storage than the values, in some cases topology can be kept in-core and only the numerical values stored out-of-core. The separation of topology, values, and auxiliary fields also enables the *Component Codecs* to take advantage of application specific knowledge to obtain good compression of each of the separate components. The Slice Cache and the Component Codecs together make up the compression component of the framework. A *Storage Handler* next takes care of storing the separate grid components either in memory or on disk. Finally, an application specific *Storage Cache*, between the *Disk* and the Storage Handler, implements our out-of-core scheme. Its function is to cache and stream pages of grid values and topology to and from disk. We emphasize again that the components for compression and out-of-core data management can be combined arbitrarily (or omitted) in our framework. Finally we note that our framework does not make any assumptions on the amount of main memory available: Level sets of any size can be processed as long as sufficient disk space is available.

4.1 Terminology

For the sake of completion we shall briefly summarize the terminology used throughout this paper, largely borrowing from Nielsen and Museth [2006]. The DT-Grid data structure is inspired by the compressed row storage format used for compact storage of sparse matrices. The DT-Grid stores the *topology* and *values* of the narrow band grid points in a compact form convenient for fast manipulation and access during for example level set and fluid simulations. In order to compactly represent the topology of the narrow band, a 3D DT-Grid consists of 1D, 2D, and 3D grid components as shown in Figure 3(a). The 3D grid component consists of the grid points in the narrow band, the 2D grid component is the projection of the narrow band onto the XY-plane, and the 1D grid component is the projection of the 2D grid onto the X-axis. Each grid component has three constituents: $value$, $coord$, and acc. As depicted in Figure 3(b), the $value_{1D}$ and $value_{2D}$ constituents link the 1D, 2D, and 3D grid components together by storing indices that point to the

[3]Hinted caching and prefetching accepts *hints* or *directives* from the user that specify the nature of future requests, for example, sequential access and so on.

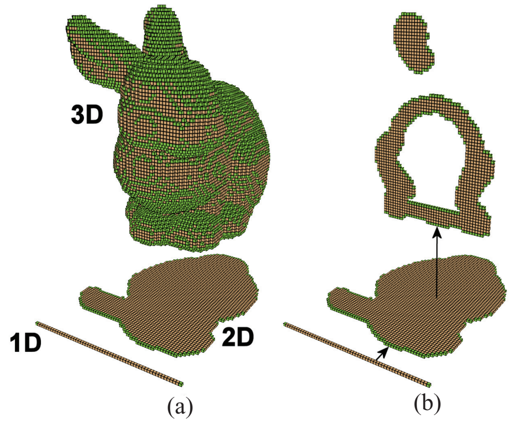

Fig. 3. a) Illustration of the 1D, 2D and 3D components of a DT-Grid representation of the Stanford bunny at 64^3 resolution. b) The 1D and 2D components contain pointers to columns in respectively 2D and 3D, and the 3D DT-Grid component stores the actual distance values of the level set function.

first coordinate in a column in the *coord* constituent of the 2D and 3D grids respectively. The $value_{3D}$ constituent stores the actual values of the level set function in the narrow band.[4] The *coord* constituent in the nD grid component stores the nth coordinate of the first and last grid point in each topologically connected component of grid points in a column of the nD grid component. These are colored green in Figure 3. The last component, *acc*, links the *coord* component of an nD grid constituent to its $value$ component by storing an index pointing to the $value$ component for the first grid point in each connected component. It is a redundant construct provided merely to speed up random access. In this article we refer to the $value_{3D}$ constituent as the *values* and the remaining constituents as the *topology*. All grid points are stored in (x, y, z) lexicographic order, and navigation through the narrow band is provided by iterators that sequentially visit each grid point in this order. As shown in Nielsen and Museth [2006], sequential access in lexicographic order allows for a wide range of algorithmic constructions and optimizations. Finally, to deform a level set and solve simulation PDEs using finite difference, fast access to neighboring grid points is paramount. This is conveniently facilitated by employing stencils of iterators that in turn allow for sequential stencil access with linear time complexity.

To describe the I/O performance of the algorithms presented in this paper we adopt the terminology of the *Parallel Disk Model* introduced by Vitter and Shriver [1994]. In particular, we denote the problem size by N, the internal memory size by M, and the block transfer size by B—all in units of data items. For this work we assume desktop machines with a single CPU ($P = 1$) and a single logical disk drive ($D = 1$).

5. OUT-OF-CORE DATA MANAGEMENT

The Storage Cache component in Figure 2 utilizes two different out-of-core data management schemes. For random access we employ the standard LRU page replacement algorithm since it is acknowledged as being the best general choice in many cases (Compare most operating systems). However, for sequential stencil access we have developed a new and near-optimal page-replacement policy as well as a new prefetching strategy. In combination these schemes reduce the number of disk blocks loaded during sequential stencil access. We focus mainly on sequential streaming since a majority of level set algorithms can be formulated in terms of sequential access operations exclusively. This is true for all the examples presented in this paper, with the only exception being ray tracing that inherently requires random access.

As illustrated in Figure 4, a sequential *stencil access pattern* in a narrow band data structure does not necessarily imply a sequential *memory or disk access pattern* when data is laid out in contiguous lexicographic order in memory or on disk. This characteristic becomes increasingly pronounced both in the case of larger level sets where the 2D slices become larger and in the case of stencils that include more grid points and hence span more 2D slices. Only data in the primary encoding direction[5] maps to contiguous locations on disk or in memory. To address this problem we need to develop new page-replacement and prefetching schemes.

Even without prefetching and page-replacement strategies, the time complexity of a sequential stencil access pattern on the DT-Grid is I/O-optimal. This is due to the fact that it requires only a linear, $O(\frac{N}{B})$, amount of I/O operations to do stencil-iteration, which equals the lower bound for a sequential scan with respect to asymptotic O-notation [Vitter 2001]. Sequential stencil access in the worst case essentially corresponds to S sequential and simultaneous scans over the data, where S is the number of grid points in the stencil. Likewise dilation and rebuilding of the narrow band [Nielsen and Museth 2006] is also linear in the number of I/O operations. However, to increase performance in practice it is important to minimize the actual number of loaded disk blocks. A straightforward I/O implementation will in the worst case result in loading $S\frac{N}{B}$ disk blocks. A lower bound is $\frac{N}{B}$ disk blocks since we need to access all grid points. Hence in practice S is a limiting constant of proportionality. For a high order FD scheme like WENO [Liu et al. 1994], a stencil with support for second-order accurate curvature computations has $S = 31$, whereas for first-order upwind computations, $S = 7$. As we will demonstrate in Section 8.1, our page-replacement and prefetching techniques do in practice lower the number of passes[6] such that it comes closer to the lower bound. This is the case even for large stencils such as WENO.

The optimal page replacement strategy [Tanenbaum 1992] for a demand-paged system (i.e., no prefetching) is simple: If a page must be evicted from the cache, it always picks the page that will be used furthest in the future. This strategy is of course impossible to implement in practice except for processes where the demand-sequence of pages is known in advance. Furthermore, since sequential stencil access into the (x, y, z) lexicographic storage order of the data structure differs from sequential access into the underlying blocks, or pages, of data on disk, the replacement issue is nontrivial. As argued previously, existing general purpose page-replacement techniques are not well suited for this access pattern. Consider for example the LRU replacement strategy. In some situations LRU will perform quite well for stencil computations. However in contrast to the replacement strategy we propose in this paper, LRU will in other cases perform far from optimally, which may degrade its overall performance: Figure 4 shows a 2D grid, a stencil consisting of five iterators, and the corresponding positions on the paged disk.

[4]Note that in the case where the DT-Grid stores volumetric fields such as velocities and pressure, the $value_{3D}$ constituent contains vectors or pressure scalars.

[5]For (x, y, z) lexicographic order this is the z direction.

[6]Measured as the ratio of the number of loaded disk blocks to the total number of disk blocks with data.

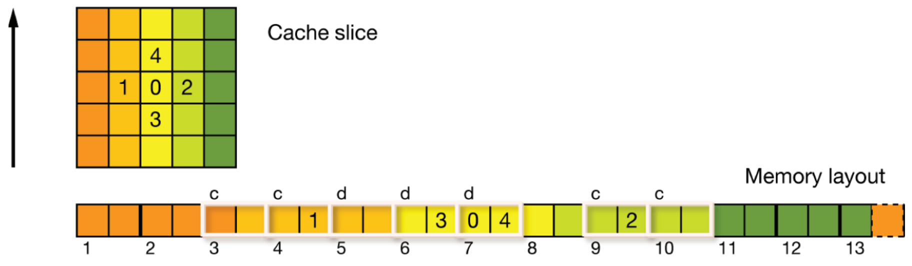

Fig. 4. Outline of a stencil consisting of five grid points (0-4) as well as a cache slice and the corresponding paged memory/disk layout. In this example all grid points are occupied with data and each page is two grid points long. The pages in memory are outlined in white and the others in black. *c* denotes a page *clean page* that has not yet been written to, and *d* is a *dirty page* that has been written to.

We assume that each iterator in the stencil contains the id of the page it currently points to. Additionally, each iterator is denoted a *reader* and/or *writer* depending on the type of access it provides. Assume that page five is the least recently used page. When iterator four moves forward, it generates a page fault (i.e., the page does not exist in memory) as page eight is not in memory. As a result page five is evicted to make room for page eight. Next consider that iterator one moves forward into page five which was just evicted. This generates a new page fault and page five is loaded again. Similar to the LRU strategy it is possible to construct examples where all other existing non-analysis-based page-replacement strategies, that we are aware of, fail. On the other hand the analysis based algorithms face other problems such as the fact that they need to detect a certain access pattern before they start working properly. In Section 8.1 we benchmark the LRU strategy in the context of stencil computations and compare to our methods.

Given that our framework is application specific, we exploit knowledge about the domain to obtain a replacement strategy that comes close to the optimal. Our strategy is verified in Section 8.1. Our page-replacement and prefetching strategy accommodates the following three essential design criteria:

—The number of disk I/O and seek operations is heuristically minimized. In particular seeking is expensive on modern hard drives.

—The disk is kept busy doing I/O at all times.

—CPU-cycles are not wasted by copying pages in memory or waiting for disk I/O to complete.

The Storage Cache, that implements the page-replacement and prefetching strategies, only depends on two parameters: The number of pages and the page size. In section 8.1 we provide some benchmarks indicating how these parameters affect performance and the page-hit-ratio.

5.1 Page-Replacement

Since the out-of-core framework stores and streams the grid values and topology in lexicographic order, the neighboring stencil iterators may be physically far apart as explained earlier and illustrated in Figure 4. The fundamental observation, however, is that during each increment of the stencil, the iterators in the stencil in most cases move forward at identical speeds. This property can only be violated at the boundaries of the narrow band where some iterators may move more grid points than others in order to be correctly positioned relative to the center stencil grid point.

Given this observation, the optimal page replacement strategy (which is invoked if the maximal number of pages allowed already reside in memory) is first to check if the page in memory with the lowest page-id does not have an iterator pointing to it. In that case we evict and return this page, and if the page is dirty it is first written to disk. In Figure 4, for example, page three can safely be evicted as it will not be used again in the future since all iterators move forward. If the first page in memory does indeed contain an iterator, the best strategy is instead to evict the page in memory that is furthest away from any of the iterators in the forward direction. This is the case since the optimal strategy is to evict the page in memory that will be used furthest in the future.

In Section 8.1 we verify that the above strategy is close to optimal by comparing it to the optimal strategy that we computed in an offline pass from logged sequences of page requests.

5.2 Prefetching

Prefetching is performed by a separate high-priority I/O thread contained in the Storage Cache. Using a separate thread to some extent hides I/O latency since this thread will wait for the I/O operations to complete.

The I/O thread iteratively performs the following steps in prioritized order, and as soon as a step is satisfied, continues from the beginning. The strategy is to prefetch pages into memory and evict pages that are no longer in use. The thread performs at most one read and one write operation per iteration. The individual steps are:

(1) *Prefetching*. The I/O thread first checks if all pages that will be accessed by the stencil iterator are already in-core. In particular this is the case if all pages ahead of the iterators in the stencil are in-core. If this is the case, no prefetching needs to be done. In addition the prefetching of a page should occur only if it does not result in the eviction of a page that is closer in the forward direction to any iterator in the stencil. This is in accordance with our replacement strategy. To determine which page to prefetch we use a variation of the elevator algorithm Tanenbaum [1992]. In our context the elevator algorithm maintains a position, which coincides with the position of an iterator in the stencil, and prefetches the nearest page in the forward direction that is not currently in-core. The variation of the elevator algorithm we employ always moves in the forward direction to the next iterator position and wraps around to continue from the beginning when the end of the data is reached. As illustrated in Tanenbaum [1992] in the context of disk arm movements, this strategy heuristically results in fewer disk seek operations and ensures that no page requests are left unserviced for long periods of time. Note that if all pages between two iterator positions are already in-core, for example, positions 1 and 3 in Figure 4, no pages need to be prefetched in this interval. In this case our elevator algorithm will move more than one

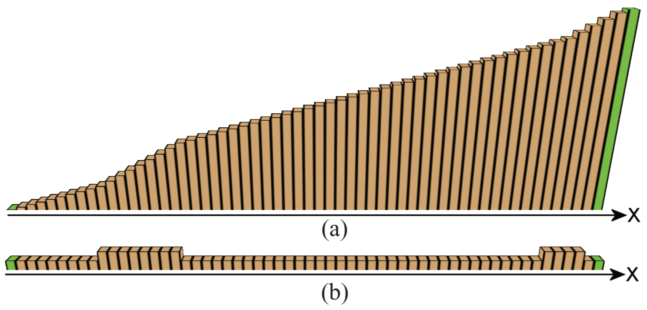

Fig. 5. a) The values of the $value_{1D}$ constituent as a histogram. b) The difference between consecutive $value_{1D}$ values as a histogram.

iterator position forward in order to locate the next page to be prefetched.

(2) *Write-Back*. If no page was prefetched, the I/O thread will attempt to write the dirty pages to disk that will not be written to again during the sequential stencil access. This is done to avoid first writing and evicting another page before prefetching a page in front of an iterator in step 1 above. Write-Back is accomplished by first checking to see if there exist any dirty pages in the cache. If this is the case, the I/O thread locates the first dirty page in the cache. If no *write* iterators point to the dirty page, it is written to disk. In some situations it is advantageous to limit Write-Back such that it is only invoked if the number of pages in the cache is above some threshold. This can ensure for example that if a file fits entirely in-core it will be kept in memory immediately ready for future access, and disk resources can hence be utilized for other purposes.

(3) *Idle mode*. If no read or write operations were performed, the I/O thread sleeps until an iterator enters a new page.

The given strategy outperformed a prefetching strategy that made its prefetching decision based on which iterator was closest to a page not residing in memory (in the forward direction) and in addition serviced page faults immediately. We believe that this result is due to an increase in the number of disk seek operations for the latter approach. In practice we use a dynamically expanding hierarchical page table to store the pages. We also employ direct I/O to prevent intermediate buffering by the OS. Hence we more effectively exploit direct memory access (DMA) and save CPU cycles and memory-bus bandwidth for numerical computations. We finally note that the Storage Cache is not dependent on any hardware or OS specific details, except that the page size is typically a multiple of the disk block size. Nor do we manually align data to fit into cache lines or similar optimizations.

6. COMPRESSION ALGORITHMS

The compression framework can be applied both online during simulation and offline as a tool for efficient application specific storage of simulation data amenable to further processing in a production pipeline. Using the proposed compression framework it is possible to compress large level set grids out-of-core with a low memory footprint. The Component Codecs we propose in this paper are based on prediction-based statistical encoding methods and separately compress the topology and values of the grid. The term *prediction-based* refers to the fact that the current symbol is predicted by previously observed symbols and it is in fact the difference between the true symbol, and the prediction that is encoded. Statistical compression

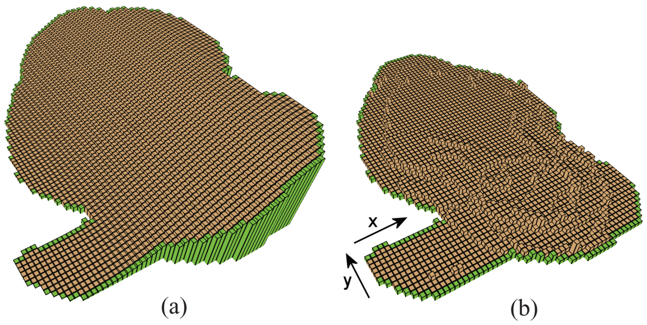

Fig. 6. a) The values of the $value_{2D}$ constituent of the 2D grid component as a histogram. b) The difference between two consecutive $value_{2D}$ values as a histogram.

methods assign probabilities to each possible symbol and in the encoding process symbols with higher probability are encoded using fewer bits. The average bit-length of a compressed symbol is expressed by the so-called entropy of the probability distribution. See Salomon [2007] for an introduction to and overview of statistical compression methods. In practice we use the fast arithmetic coder described by Moffat et al. [1998] combined with optimized adaptive probability tables. Adaptive probability tables assign probabilities to symbols based on the frequency with which they are observed in the previously processed stream of symbols [Salomon 2007]. While the adaptive statistical encoding methods are ideal for sequential access, random access is typically not feasible into a statistically encoded stream of data. This is because the encoding of a single element depends on all elements encountered before that. To remedy this somewhat, synchronization points could be inserted into the stream of data. Naturally this comes at the cost of decreasing compression efficiency. As discussed previously we use sequential algorithms and focus here solely on online as well as offline compression using sequential access.

Next we describe how to compress the topology and the signed distance field values of the grid. The topology is compressed lossless, whereas the values can be compressed in either a lossless or a lossy fashion. Note that the signed distance field is the most typical level set embedding, and that the topology component codecs presented in this section are specific for the DT-Grid. However, very similar codecs can be applied to other sparse representations such as the H-RLE [Houston et al. 2006].

6.1 Compressing the Topology

The $value_{1D}$ constituent of the topology consists of monotonically increasing indices (Figure 5(a)) that point into the *coord* constituent of the 2D grid component. The difference between two such consecutive values (Figure 5(b)) is twice the number of connected components in a column in the 2D grid component, see the 2D grid component in Figure 7(a). Due to the large spatial coherency in a level set narrow band, this quantity does not usually vary much. To compress it, we encode this difference, that is, the number of connected components per column, using a second order adaptive probability model [Salomon 2007]. The $value_{2D}$ component has characteristics similar to the $value_{1D}$ component as shown in Figure 6(a), and the semantics of the difference between two consecutive values in (x, y) lexicographic order is the same, see Figure 6(b). Hence this constituent is also compressed using a second order adaptive probability model.

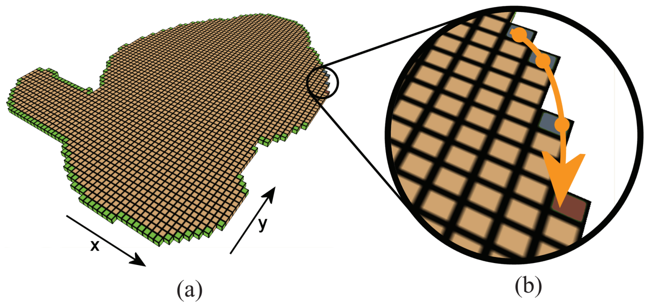

Fig. 7. a) The 2D grid component of the 64^3 Stanford bunny DT-Grid. The y-coordinates of the $coord_{2D}$ constituent are shown in green. b) A close-up shows actual y-coordinate (red-brown) predicted by three previous y-coordinates (blue).

The $coord_{1D}$ (x-coordinates) constituent of the topology is encoded using a single differential encoding and a zeroth-order adaptive probability model [Salomon 2007]. Typically the $coord_{1D}$ component constitutes an insignificant percentage of the aggregate space usage since it consists only of the end points of the connected components obtained by projecting the level set narrow band onto the X-axis. For example, only two x-coordinates are needed to store the Stanford bunny, since it projects to a single connected component on the X-axis. As a reference see Figure 3, where these two x coordinates are marked in green in the 1D component.

The $coord_{2D}$ (y-coordinates) constituent of the topology consists of the y-coordinates that trace out the boundary curves in the XY plane of the projection of the level set narrow band. This is illustrated with green in Figure 7(a). Again due to the large amount of coherency in the narrow band, these curves are fairly smooth. Hence it is feasible to employ a predictor that estimates a given y-coordinate from the y-coordinates in the (at most) three previous columns in the XY plane. Figure 7(b) illustrates how the y-coordinates in three previous columns, shown in blue, are used to predict the y-coordinate in the next column, shown in red-brown. In particular we use as predictor the Lagrange form of the unique interpolating polynomial [Kincaid and Cheney 1991] that passes through the y-coordinates in the previous columns. Our tests show that higher order interpolants tend to degrade the quality of the prediction.

Since the topology of these boundary curves is not explicitly given in the $coord_{2D}$ constituent, the curves become harder to predict. Recall that the $coord_{2D}$ constituent only lists the y-coordinates in lexicographic order. Hence to locate the y-coordinates in the previous columns that will form part of the prediction, we utilize the known information of which connected component we are compressing.[7] We then predict from the y-coordinates of connected components with identical ids in previous columns. Note that we cannot simply use the actual true y-coordinate as a means of determining the y-coordinates in the previous columns since it will not be available during decompression. The above selection criterion means that the prediction will degrade along columns where the number of connected components change, but in practice we have not found this to be a problem.

The $coord_{3D}$ constituent consists of the z-coordinates of the grid points that trace out the boundary surfaces of the level set narrow band. These are shown in green in Figure 8(a). Surpassed only by the storage requirements of the signed distance field values in the grid, the $coord_{3D}$ (z-coordinates) constituent of the topology usually requires the most space. To compress a given z-coordinate, shown in red-brown in Figure 8(b), the z-components of the three immediate neighbors, shown in blue, are used to predict the given z-coordinate as lying in the same plane, see Figure 8(c). We predict $z(D)$ as $z(A) + \nabla z \mid_A \cdot \binom{1}{1} = z(B) + z(C) - z(A)$ (using a backward one-sided first order accurate finite difference approximation to the gradient). Given the permutation symmetry of this expression with respect to $z(B)$ and $z(C)$ we compress the prediction using a context-based adaptive probability model [Taubin 2002]) with the integer value $z(B) + z(C) - 2z(A)$ as context. In particular the context is used to select a probability model, and the goal is to cluster similar predictions in the same model, hereby decreasing the entropy and consequently increasing the compression performance. The intuition behind our context is that it measures the deviation of both $z(B)$ and $z(C)$ from $z(A)$. The smaller the deviation, the smaller the residuals tend to be. Special care has to be taken when some grid points are not available for our predictor. Furthermore, we distinguish between and use a different context in the following three cases: 1) If no grid points exist at all, we use 0 as the prediction. 2) If one exists we use the z-coordinate of that grid point as the prediction. 3) If two exist we use the average of their z-coordinates as the prediction. All in all, this compression strategy turned out to outperform alternatives like differential encoding, 1D Lagrange polynomial interpolation, and 2D Shepard interpolation.

Finally, we recall that the acc constituent of grid components is actually redundant. It is merely used to improve random access into DT-Grid. Hence we can simply exclude the acc constituents in compressed form and rebuild them during decompression. This essentially corresponds to exploiting the Kolmogorov complexity [Li and Vitanyi 1997] for the compression of acc.

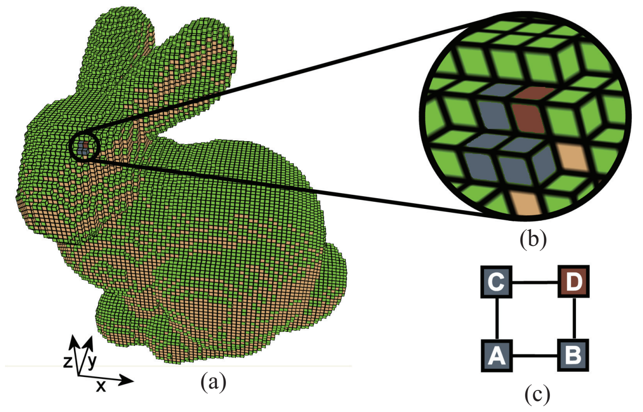

Fig. 8. a) The 3D grid component of the 64^3 Stanford bunny DT-Grid. The z-coordinates of the $coord_{3D}$ constituent are shown in green. b) A close-up shows actual z-coordinate (red-brown) predicted by three immediately adjacent z-coordinates (blue). c) Situation in b) shown from above.

6.2 Compressing the Values

The values in the narrow band are by far the most memory-consuming part of the data (typically at least 80%). For level sets, we assume the values are numerical approximations to signed distances, which has been shown to be convenient both during simulation as well as for other applications such as ray tracing. To compress the narrow band of signed distance values we propose a predictor based

[7]In particular its connected component id, starting from zero and counted in lexicographic order within a column.

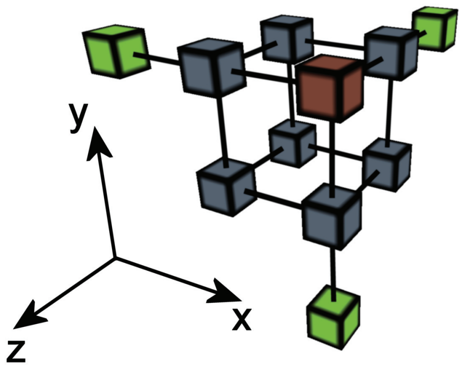

Fig. 9. The values are compressed using a combination of the 3D Lorenzo predictor, the 2D parallelogram predictor and multi-dimensional differential encoding.

on a combination of the following three techniques: a new *multidimensional differential predictor*, the 3D Lorenzo predictor of Ibarria et al. [2003], and the 2D parallelogram predictor of Touma and Gotsman [1998].[8] Since we are compressing narrow bands as opposed to dense volumes, it is tempting to utilize clamped values outside the narrow band (as is usually done in level set computations) to form predictions; however, this will result in a degradation of compression performance. Instead we propose to use different predictors depending on the local topology of the narrow band. We have benchmarked various predictors modified to accommodate the topology of a narrow band, including the Lorenzo predictor [Ibarria et al. 2003] (with and without probability contexts), the distance field compression (DFC) method by M.W. Jones [2004] as well as several other custom codecs. Note also that the DFC method and the Lorenzo predictor perform comparably to wavelets. According to our experiments, the codec we propose here gives the best compression performance and at the same time remains fast.

Our multidimensional differential prediction is motivated by the fact that the axial acceleration in a signed distance field is very small and that axial differential prediction applied twice is a measure of acceleration. In fact the acceleration in the normal direction of a perfect signed distance field is identically zero, except at medial axes. However, in practice several circumstances make a prediction based on the acceleration in the normal direction problematic. First of all, the signed distance fields used in the context of level set simulations are not entirely accurate as they are computed by approximate numerical schemes. Secondly, it can be shown (using first order FD) that the acceleration in the normal direction is a third-degree polynomial in the value of the current grid point. During decompression one would have to compute the roots of this polynomial in order to determine the decompressed value. This is time-consuming, and in addition to compressing the residuals themselves, one would also have to compress a two-bit code indicating which of the solutions to the third degree polynomial was the right residual. This information is required during decompression when the actual value is not available. Tests show that in practice our combined predictor in fact leads to better and faster compression than if compression is applied to the acceleration in the normal direction.

The intuition behind our approach is for the predictor to utilize as many of the previously processed locally connected grid points as possible (see Figure 9). In other words we always apply the predictor which uses the largest number of already processed grid points. In our experience this results in the best compression performance and explains the prioritized order of predictors given below. Consider now Figure 9, depicting eleven locally connected grid points. Assume that we wish to compress the value of the red grid point at position (x, y, z) and that the blue and green grid points that exist have already been processed. Our predictor takes the following steps to compute a residual which is then compressed using an arithmetic coder:

(1) If all the blue grid points exist in the narrow band, we predict the value at the red grid point using the 3D Lorenzo predictor by computing the following residual: $v_{(x,y,z)} - (v_{(x-1,y-1,z-1)} - v_{(x-1,y-1,z)} - v_{(x-1,y,z-1)} + v_{(x-1,y,z)} + v_{(x,y-1,z)} - v_{(x,y-1,z-1)} + v_{(x,y,z-1)})$.

(2) If some of the blue grid points do not exist in the narrow band, we determine if it is possible to apply the parallelogram predictor. This can be done if the red grid point is part of a face (four connected grid points in the same plane) where all grid points have already been processed. As can be seen from Figure 9 there are three such possible faces. Say that all the grid points in the face parallel to the *XZ* plane, $v_{(x-1,y,z)}$, $v_{(x,y,z-1)}$ and $v_{(x-1,y,z-1)}$, exist. The value at the red grid point is then predicted using the parallelogram predictor and the residual is computed as $v_{(x,y,z)} - (v_{(x-1,y,z)} + v_{(x,y,z-1)} - v_{(x-1,y,z-1)})$. The procedure for the remaining faces is the same. Each face is examined in turn, and the first face where the above conditions apply is used to compute the residual.

(3) If it is not possible to find a face as described above, we switch to using axial second-order differential prediction, which, as previously mentioned, is a measure of acceleration. We examine each coordinate direction in turn and the first direction where two previous grid points exist (a blue and a green) is used to compute the residual. Say that the two previous grid points in the *X* direction, $v_{(x-1,y,z)}$ and $v_{(x-2,y,z)}$, exist. Then we compute the residual at the red grid point as $v_{(x,y,z)} - 2v_{(x-1,y,z)} + v_{(x-2,y,z)}$.

(4) If it is not possible to apply axial second-order differential prediction we apply first-order differential prediction if the previous grid point in one of the coordinate directions exist. For example, if the previous grid point in the *X* direction exists, we compute the residual as $v_{(x,y,z)} - v_{(x-1,y,z)}$. Again we use the first coordinate direction that applies to compute the residual.

(5) Finally, if none of the above conditions apply, we simply encode the value itself.

How often each of the individual predictions above is utilized depends on the narrow band topology. Internally in the narrow band, (1) is always applied since all neighbors are available. The others are used on the boundary of the narrow band depending on the local configuration of previously processed neighbors. Note that in predictions (2), (3), and (4) we do not necessarily use the face or coordinate direction that results in the best prediction; instead we just pick the first one that applies. The reason is that this procedure can be done independently in both the encoder and the decoder. The alternative is to examine all possibilities, choose the best, and also encode a two-bit code indicating which of the possibilities was chosen. In our experience this overhead dominates the gain obtained by selecting the best prediction. We do, however, use a different probability context in each of the steps used to compute this predicted value. Employing contexts improves compression performance mainly for larger level sets. For smaller level sets the use of several contexts does not usually improve compression performance since we typically use relatively many bits (14 and above) in the quantization

[8]Note also that the Lorenzo predictor is a generalization of the parallelogram predictor

step. This means that the entropy in the individual adaptive contexts may be dominated by the probabilities allocated for unused symbols. Also note that when using relatively many bits, higher-order probability models are not feasible in practice due to the amount of memory usage they incur. This is contrary to text compression that uses fewer bits and where higher order models are frequently used.

Whereas the topology is compressed lossless, the values are typically compressed in a lossy fashion by employing uniform quantization within the range of the narrow band[9] to obtain better compression ratios. In doing so it is important that the quantization does not introduce noticeable distortion and that the truncation error introduced by the order of the numerical simulation methods is not affected by the quantization rate. In practice an exact analysis of this is difficult and highly problem dependent. For this reason we have so far simply used the heuristic of 14 bits or more, which has not resulted in visual artifacts in our simulations. If quantization is not desirable, the method for lossless encoding of floating point values by Isenburg et al. [2004] can be applied.

An additional compression strategy would be to only encode a subset of the narrow band and then recompute the rest from the definition of the level set as a distance function when decoding. This amounts to encoding as many layers around the zero-crossing as is needed to solve the Eikonal equation to a desired accuracy. The truncated narrow band will obviously lead to a more efficient subsequent compression. In our case we do typically not use narrow bands wider than required for the numerical accuracy, so we have not used this strategy in practice. However, it may be applicable in situations where the narrow band is relatively wide, such as for example, the morph targets demonstrated in Houston et al. [2006].

7. OUT-OF-CORE AND COMPRESSED PARTICLE LEVEL SETS

When using level sets for free surface fluids it is common to correct the interface with advected Lagrangian tracker particles. This Particle Level Set (PLS) method [Enright et al. 2002], greatly helps the level set preserve mass due to the built-in numerical dissipation (i.e., smoothing) in most Eulerian (i.e., grid-based) advection schemes. Since this undesired numerical dissipation is most pronounced in regions of the surface with high curvature (i.e., features), PLS considerably preserves fined surface details. Furthermore, PLS can be extended to create natural phenomena like splashes and foam see ([Foster and Fedkiw 2001; Takahashi et al. 2003; Geiger et al. 2006]) that are typically very hard to obtain with Eulerian fluid solvers. However, PLS comes at a significant price; the storage of the particles introduces a large memory overhead compared to the level set representation itself, especially when employing sparse grids like DT-Grid and H-RLE. In this section we will discuss two techniques that dramatically reduce the memory footprint of PLS; quantization and out-of-core streaming. Note that the PLS method has certain constraints, such as only being applicable under passive surface advection [Enright et al. 2002], and that newer methods overcoming these constraints have recently been proposed [Bargteil et al. 2006; Hieber and Koumoutsakos 2005]. However, the PLS method can, in contrast to the newer methods, easily be integrated with an existing Eulerian level set implementation, and improves the quality of fluid animations, which is a passive surface advection problem.

The source of the memory overhead associated with PLS is that many particles are needed to effectively correct the implicit interface. Enright et al. [2002] recommend using 64 particles per 3D grid cell close to the zero level set. If we assume that each particle stores 17 bytes of data[10] we need 1088 bytes to store the particles of each grid cell. This should be compared to the 4 bytes needed to store the signed distance floating point value of the level set function in the same grid cell. If the level set is stored using a sparse narrow band data structure, such as the DT-Grid or the H-RLE, the particles can use close to two orders of magnitude more memory than the level set itself. For instance, consider the fluid animation in figure 1 which is represented by a DT-Grid PLS containing 7.6M voxels. Storing this level set requires only 32 MB whereas the 172M particles associated with the interface requires an additional 2.8 GB.

Our primary approach to reducing the memory footprint of the PLS is lossy compression by means of quantization. We choose to use quantization since it is fast, flexible and relatively simple to implement. We consistently choose the level of quantization such that no visual artifacts are introduced.[11] Note that because the particle distribution is uniformly random around the interface, statistical encoders are not very effective.

7.1 Particle Quantization

The particle radii is limited to $[-0.5dx, 0.5dx]$, where dx denotes the uniform grid spacing in world coordinates. Enright et al. [2002] suggest using $[0.1dx, 0.5dx]$ as the range of allowed radii, but since we will use the sign of the radius to represent whether a particle is inside or outside the interface, we shall employ the wider symmetric interval. Our approach is more compact since we simply use one bit in the sign of the radius as opposed to 8 bits for a boolean. The interval of possible values for the coordinates is only limited by the size of the simulation domain.

To facilitate better compression and out-of-core performance we switch from world coordinates to a local particle-in-cell frame of reference. This means that the position of the particle in the beginning of each iteration is bounded to the local coordinate interval [0, 1] of the grid cell. Currently we assume that the level set advection adheres to the CFL condition [Osher and Fedkiw 2002; Sethian 1999], meaning that the surface is restricted to move at most one grid cell in any direction during each iteration. Due to this, the same restriction is imposed on the particles. Consequently the interval of possible values for the local particle coordinates is $[-1, 2]$. The local interval for the particle radii is simply $[-0.5, 0.5]$. Thus, the values of the particle radii and positions are bounded and independent of the size and resolution of the grid. Another essential feature of this local frame of reference approach is that it allows us to store bins of particles in the same (lexicographic) order as the DT-Grid. This in turn facilitates fast sequential access, which is critical for an efficient out-of-core implementation. Finally, the use of local coordinates significantly improves the precision for a given level of the quantization. Assume that we want to quantize particles stored in a regular, dense grid of the size $100 \times 100 \times 100$ grid cells. Using 10 bits to quantize each component of the particle world coordinates we achieve a precision of $\pm 0.5 \cdot 100/2^{10} \approx \pm 0.049$. If we instead used the local coordinate approach described earlier, we achieve a precision of $\pm 0.5 \cdot 1/2^{10} \approx \pm 0.00049$. In our implementation we quantize particles using 40 bits. 9 bits for the radius, 10 bits for each

[9]During simulation with compressed values, the quantization range is expanded to ensure that advected values fit within the quantization range. Narrow band level set methods need to limit the maximal movement between time steps anyway to ensure that the zero-crossing is captured correctly.

[10]3 4-byte floats for the 3D position of the particle center, one 4-byte float for the radius and one additional byte to store boolean information such as whether the particle has escaped the level set or not.

[11]This is generally verified through numerical experiments.

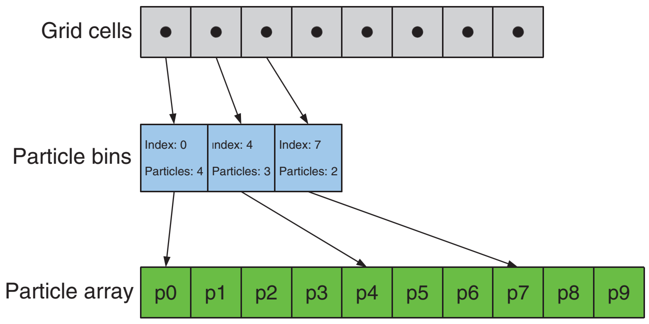

Fig. 10. Outline of the particle/bin data structure. Each grid cell contains a particle bin that in turn points to a range of particles in the particle array.

coordinate and the remaining bit to store Boolean information, such as whether the particle has escaped from the level set. If more precision is desired, we suggest increasing the size of the particle one byte at the time by adding 2 additional bits to the radius and the 3 coordinate components. However, we found that the 40-bit quantization produced simulations with a visual appearance that was very close to that of the unquantized PLS. Using this relatively simple quantization scheme we are able to reduce the size of a particle from 17 to 5 bytes, a reduction by a factor of 3.4. Since we rely on topology data already provided by the grid, we achieve a precision of $\pm 0.5 \cdot (2 - -1)/1024 \cdot dx \approx \pm 0.0015 \cdot dx$ for the coordinates and $\pm 0.5/512 \cdot dx \approx \pm 0.00098 \cdot dx$ for the radius. Note that in spite of the quantization our particle representation is still first order accurate.

Our implementation of the particle/bin data hierarchy employs one 4-byte integer pointer[12] and a 1-byte character to each grid cell in the narrow band. The integer points into an array containing all particles and the additional byte is used to store the number of particles present in the grid cell. This pair of values can be seen as a particle bin that can store up to 256 particles.

Storing the PLS representing the fluid in Figure 1 using the quantization scheme outlined above requires only 890 MB. 32 MB to store the 7.6M grid cells using a DT-Grid, 36 MB to store the particle bins and 821 MB to store the 172M particles. This is a reduction of data by a factor 3.2 compared to the 2.8 GB needed to store the uncompressed PLS.

7.2 Out-of-Core Particle Level Sets

Though the quantization scheme described above significantly reduces the memory footprint of the PLS we are still limited by the amount of available in-core memory. To allow for PLS of extreme resolution we employ the out-of-core framework described in Sections 4 and 5. We note that the technique described in this section can be used without quantization. However, since the PLS method is I/O intensive this typically leads to poor out-of-core performance. When quantization is employed the I/O throughput[13] increases by more then a factor of three.

For the out-of-core implementation we use the particle/bin data structure described in the previous section. We also store particles, particle bins, and the velocities used to advect the particles in sequential order. This storage order is readily obtained by initially iterating trough the grid in sequential order and adding data associated with every grid cells when that cell is encountered. The challenge is to keep this sequential order when the level set and the particles are advected. Since the values for the velocities and particle bins are directly associated with grid points, they will always be accessed sequentially as long as the grid is accessed sequentially. The particles on the other hand are more tricky. When they move between grid cells the sequential storage order of the particle array will be broken. Assume that all the particles have been advected but not yet moved to their new grid cells, that is, that all particles have local coordinates in the range of $[-1, 2]$. We can now move the particles to their target cells while keeping the sequential access order by the following algorithm:

(1) *Initialization*. Allocate two new out-of-core arrays; a particle array and a particle bin array both of the same size as the ones currently used. Fill the particle bin array with empty particle bins; that is, all bins contain a null pointer into the particle array and zero particles. Finally create an out-of-core array of unsigned bytes. This array will keep track of how many particles end up in each grid cell after they have been moved. Initialize each counter with the current number of particles in each cell.

(2) *Preprocessing*. Iterate through the grid sequentially and look at each particle. If the particle coordinates indicate that it is still in its current grid cell; do nothing. Otherwise add 1 to the particle counter corresponding to the neighboring grid cell into which the particle is moving and subtract 1 from the particle counter for the current grid cell. Do not move the particles to their new cells yet. If a particle intends to moves into a grid cell that does not have an allocated particle bin, that is, a bin with a null index pointer, mark this bin to be added by, for example, setting the pointer to a large value. Particle bins in cells neighboring the current one can be reached by using a stencil-iterator containing 27 stencil points. One stencil point for the current cell and one for each of the 26 neighbors a particle can potentially move into.

(3) *Particle bin creation*. Initialize an index counter starting at zero. Sequentially iterate through the grid again. For each particle bin that does not have a null pointer (this includes the ones flagged in the previous step) do the following: Read the number of particles that will exist in the bin after advection from the array of particle counters. Set the index pointer for the new particle bin corresponding to the current grid cell to the value of the index counter. Increase the index counter by the number of particles that will be present in the cell after advection. This will set the index pointer for each of the new particle bins to the correct position in the new particle array leaving room for the number of particles that will be present in that bin after the particles have been moved to their new cells.

(4) *Particle relocation*. Iterate through the grid one last time. For each particle in a populated cell determine which of the 27 possible cells the particle is moving into. Using the stencil, read the particle bin associated with this cell from the new array of particle bins. Write the particle into the previously allocated new particle array at the index position $index_{start} + num_p$ where $index_{start}$ is given by the target particle bin and num_p is the number of particles currently present in that bin.

(5) *Cleanup*. Delete the old particle array and the old particle bins.

This algorithm associates a new particle bin with each grid cell and writes a new particle array that stores all particles in such a way that they will be accessed sequentially when iterating through the grid in sequential order. Since we use one byte to describe the

[12] This allows us to address 4.3 billion particles. The size of the pointer can of course be increased if more particles are needed.

[13] The amount of particles that can be read/written to disk per time unit.

number of particles in each particle bin it is possible that more than the allowed 255 particles will enter the bin. If this happens we simply throw away the excess particles.

After advecting all particles the density of particles in each grid cell may have changed drastically—some cells may contain far more than the desired density of 64 particles and some may be completely empty. The PLS method requires us to enforce the density constraint once in a while by adding particles to underpopulated cells and deleting excess particles from crowded cells. Maintaining the sequential storage order of the particles during this step can be done using the following simple algorithm:

(1) *Initialization.* Create a new, empty out-of-core particle array. Also create a counter and initialize it to zero.
(2) *Density enforcement.* Iterate through the grid in sequential order and read all particles present in the current grid cell. If the cell has too few particles, seed new ones, if it has too many, remove the overhead. After this, add all the particles in the current cell to the end of the new particle array. Set the index pointer for the particle bin corresponding to the grid cell to the value of the counter and set the number of particles in that bin to the number of particles added. Finally increment the counter by the number of particles added.
(3) *Cleanup:* Delete the old particle array.

The two algorithms we have outlined ensure a sequential storage order of the particles when the particle density in a grid cell changes and when particles move between grid cells. Without them the particle array will fragment and reading/writing of particles will soon require a large amount of random access operations on the hard disk. The PLS method contains more then just advecting particles and enforcing a specific particle density, but the remaining steps are identical to their in-core counterparts. Performance benchmarks for the quantized and out-of-core PLS are available in Section 9.2. We have also used our out-of-core framework, including our compact PLS, in free surface fluid simulations. Section 9.3 reports results from these simulations, as well as a brief description of the employed method.

8. BENCHMARKS AND RESULTS

In this section we evaluate the performance of our out-of-core and compression framework. In particular, we demonstrate that the level set framework can sustain a throughput that is 65% of the peak performance of state-of-the-art in-core simulations—even for models of sizes in the order of several GB.

8.1 Benchmarks

As emphasized previously, the out-of-core and compression components can be combined arbitrarily which gives distinctive properties to the resulting framework. For instance, keeping both topology and values in-core gives the best performance, streaming values to disk and keeping topology in-core gives the second-best performance whereas streaming values to disk and compressing topology in-core usually gives the third-best performance. The two parameters of the out-of-core framework, the page size and the number of pages in the cache, as well as the number of quantization bits used in the compression also affect performance. Typically relatively few pages and large page sizes give the best results. Depending on the size of the problem at hand and the computing resources available, the user can choose an appropriate combination of framework components for his particular setting. In this section we report the performance resulting from combining the different components of the framework and elaborate on how to choose the parameters of the cache. We also verify the near-optimality of our page-replacement policy for stencil iteration. All the benchmark tests presented in Section 8 are run on the same 32-bit Windows XP Pro PC with a 2.41GHz AMD CPU, 1GB of main memory and a Western Digital Raptor disk.

8.1.1 *Page Replacement and Prefetching.* Table I lists the hit ratio (number of page hits to the number of total page requests) of LRU page-replacement without prefetching (demand-paging only), our page-replacement algorithm without prefetching (demand-paging only), the optimal page-replacement policy for a demand-paging algorithm [Tanenbaum 1992] and our page-replacement algorithm with prefetching enabled. Clearly a hit ratio of one is an upper bound. It is not possible to apply the optimal demand-paging strategy online since it requires knowledge of future requests, but by logging the page demands during stencil iteration one can compute the optimal strategy offline in order to do comparisons. The reader should note that the optimal demand-paging strategy is only optimal amongst the *demand-paged* algorithms, that is, where prefetching is not included. This explains why it is possible for our combined page-replacement and prefetching algorithm to achieve better hit ratios than the optimal demand-paged algorithm in Table I. The test case is a single sequential stencil iteration over an out-of-core DT-Grid of the Stanford Bunny in resolution 1000^3 using a WENO finite difference stencil with 19 grid points. Initially the cache contained no pages. As can be seen from Table I, our page-replacement policy without prefetching comes very close to the optimal and performs better than the LRU strategy. When combining with our prefetching strategy, the hit ratio is close to one for larger page sizes. Hence we conclude that our page-replacement algorithm comes close to optimal and that our prefetching algorithm heightens performance, bringing it near the optimal hit-ratio of one. Note that Table I lists relatively small page sizes. This is primarily to show how the hit ratio increases with page size, and that our replacement strategy works well even in the presence of relatively small page sizes. In order to increase the throughput, however, larger page sizes must typically be used (see the following). This is particularly important for larger level sets. In such cases the hit ratio usually remains close to one, even for gigabyte-sized level set models.

To measure the I/O bandwidth performance we also logged the total number of pages read, R, and compared this to the number of pages occupied by the level set model, P. Optimally $Q = \frac{R}{P} = 1$; that is, each page is loaded exactly once. For the tests above having a page size above 4K, Q is below 2 and in most cases close to 1. In this case the stencil contained $S = 19$ grid points, hence the improvement over the worst case ratio of 19 is significant (recall the discussion in Section 5).

The choice of parameters for the out-of-core framework, page size, and number of pages, can affect performance quite dramatically, as illustrated in the graph in Figure 11. The optimal parameters depend on the problem at hand: the size and topology of the level sets involved, the number of grid points in the stencil, and the underlying hardware. One can run benchmark tests to tune these parameters for a particular example, but this is typically not very practical. While we leave it for future work to determine exactly how the optimal parameters depend on hardware as well as characteristics of the simulation, we have found that a page size of 4MB and a total of 32 pages in the cache performs quite well over a wide range of level set sizes and stencils. The graph in Figure 11 shows an elaborate benchmark test where the number of pages and page size is varied for sequential stencil iteration over the Stanford Bunny in resolution 8000^3. In this particular case a page size of 8MB and a total of 32 pages performs best, but a page size of 4MB

Table I.
Comparison of the Page-Hit-Ratios of Our Page-Replacement Policy with Prefetching Disabled (*Our Demand*), Our Page-Replacement Policy with Prefetching Enabled (*Our Prefetch*), LRU Page-Replacement Without Prefetching (*LRU Demand*) and the Optimal Page-Replacement Policy for a Demand-Paged Replacement Policy Computed Offline (*Opt Demand*) from a Logged Sequence of Demand Requests. See the Text Below for an Exact Explanation of These Terms. The Results Eere Generated by a Single Sequential Stencil Iteration Over the Stanford Bunny in Resolution 1000^3 Using a WENO Finite Difference Stencil with $S = 19$ Grid Points, and the Cache Contained no Pages at the Beginning

	32 pages		64 pages		128 pages	
Page-size (KB)	LRU Demand	Opt Demand	LRU Demand	Opt Demand	LRU Demand	Opt Demand
0.5	0.631377	0.645004	0.631647	0.660637	0.631802	0.691117
1.0	0.631651	0.660015	0.631805	0.690580	0.640359	0.748979
2.0	0.631810	0.689569	0.640360	0.748286	0.642968	0.857327
4.0	0.640384	0.746922	0.643010	0.856420	0.888819	0.943889
	Our Demand	Our Prefetch	Our Demand	Our Prefetch	Our Demand	Our Prefetch
0.5	0.642074	0.945229	0.657989	0.946476	0.688768	0.946760
1.0	0.654621	0.946133	0.685807	0.946775	0.745399	0.947049
2.0	0.679419	0.947686	0.740625	0.948007	0.853331	0.954492
4.0	0.730505	0.989303	0.848156	0.990706	0.943635	0.959498

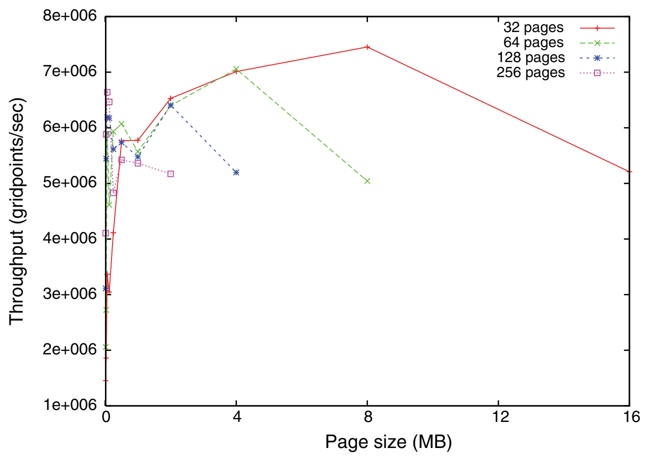

Fig. 11. The throughput (processed gridpoints/second) as a function of page size (in MB) for various numbers of pages in the cache. The results were generated by sequential iteration over the Stanford Bunny in resolution 8000^3 with a finite difference stencil of seven grid points. The maximal memory usage of the cache was in this case restricted to 512 MB.

and a total of 32 pages also performs quite well. The latter configuration is the one we used for all benchmarks presented in the next section.

Note also that we use direct I/O (i.e., DMA) to bypass OS caching and prefetching. In our experience this gives an average speedup of approximately 10%. If on the other hand we leave out our own prefetching algorithm and rely solely on OS prefetching through the use of higher level I/O system calls, the performance is roughly half of the performance obtained when we utilize our own prefetching algorithm.

8.1.2 *Online Out-of-Core and Compression Framework.* As a prelude to the performance evaluation of our flexible framework we define the following variations:

—OOC DT-Grid I: Values uncompressed out-of-core, topology uncompressed in-core.
—OOC DT-Grid II: Values uncompressed out-of-core, topology uncompressed out-of-core.
—OOC CDT-Grid I: Values uncompressed out-of-core, topology compressed in-core.
—OOC CDT-Grid II: Values compressed out-of-core, topology compressed out-of-core.
—CDT-Grid I: Values compressed in-core, topology compressed in-core.

The performance of these data structures is evaluated by comparing the throughputs measured in processed grid points per second. We use three test cases: 1) The read throughput of sequential iteration with a seven grid point finite difference stencil. 2) The combined read and write throughput with the same stencil iteration. 3) The practical throughput of an actual level set simulation, in this case an erosion.

Table II lists the average throughputs of several tests with the framework variations defined above as well as a custom implementation of the original in-core DT-Grid which is state of the art [Houston et al. 2006]. For simulations that fit in-core, our framework introduces an overhead due to the additional software layer (see Figure 2). In particular, when storing topology and values uncompressed in-core, simulations perform at about 76% of the performance of the original DT-Grid. Hence the framework overhead is approximately 24%. This also means that a throughput of 76% is an approximate upper bound for the peak performance of our framework. However, due to fluctuations in overall system performance this may vary slightly. For read and read/write iterations, the upper bounds on performance were estimated to be approximately 72% and 76% respectively. A performance of an out-of-core data structure close to these upper bounds indicates that the method is CPU or memory limited, otherwise it is I/O limited.

Table II indicates that the throughput of both read and read/write iterations for our framework does not depend on the number of voxels in the narrow band of the level set. This property is not shared by the original in-core DT-Grid for which the performance drops significantly around a resolution of 4000^3 which is when the limit of

Table II.
Throughput Rates (gridpoints/second) for Stencil Iteration Through the Entire Narrow Band of the Stanford Bunny with Reads Only, Stencil Iteration Through the Entire Narrow Band with Reads and Writes, and for a Level Set Simulation (erosion). The Numbers Given in Parenthesis are the Percentages of the State-of-the-Art In-Core DT-Grid Performance with Respect to the Given Test (read-iteration, read/write-iteration or simulation). For Each Instance of the Stanford Bunny, its Resolution (*res*), the Number of Grid Points in the Narrow Band (*#GP*), the Uncompressed DT-Grid Size (*size*) and the Uncompressed DT-Grid Size Where an Additional N-Tube Which is Required for Simulation has been Added to the Level Set (*simsize*) is Reported. A 14 Bit Quantization was Used for the Compressed Values. For the Data Structures Compressing the Values Only Read-Iteration was Considered Since During Compression it is not Possible to Both Read and Write to the Same Stream of Data. Furthermore Some of the Tests are not Possible Due to Virtual Memory Constraints. In These Cases the Result is Denoted NP = Not Possible

Grid	Reading GP/Sec	Reading&Writing GP/Sec	Simulation GP/Sec	Reading GP/Sec	Reading&Writing GP/Sec	Simulation GP/Sec
	Res=1000^3, #GP=1.4e7, size=71MB, simsize=84MB			Res=2000^3, #GP=5.7e7, size=289MB, simsize=341MB		
DT-Grid	1.1e7 (100%)	9.9e6 (100%)	1.2e6 (100%)	1.1e7 (100%)	9.8e6 (99%)	1.2e6 (100%)
OOC DT-Grid I	7.1e6 (65%)	7.4e6 (75%)	9.4e5 (78%)	7.5e6 (68%)	7.4e6 (75%)	7.7e5 (64%)
OOC DT-Grid II	6.8e6 (60%)	6.5e6 (66%)	9.0e5 (75%)	7.7e6 (70%)	7.4e6 (75%)	6.9e5 (58%)
OOC CDT-Grid I	5.2e6 (47%)	5.1e6 (52%)	6.6e5 (55%)	5.3e6 (48%)	5.2e6 (53%)	6.5e5 (54%)
CDT-Grid I	1.2e6 (11%)	NP	8.6e4 (7%)	1.2e6 (11%)	NP	8.6e4 (7%)
OOC CDT-Grid II	1.2e6 (11%)	NP	9.9e4 (8%)	1.2e6 (11%)	NP	1.0e6 (8%)
	Res = 2500^3, #GP = 8.9e7, size = 454MB, simsize = 578MB			Res = 3000^3, #GP = 1.3e8, size = 655MB, simsize = 771MB		
DT-Grid	1.1e7 (100%)	9.9e6 (100%)	2.2e5 (18%)	1.1e7 (100%)	9.8e6 (99%)	NP
OOC DT-Grid I	7.4e6 (67%)	7.4e6 (75%)	7.9e5 (66%)	7.6e6 (69%)	7.5e6 (76%)	7.9e5 (66%)
OOC DT-Grid II	7.8e6 (71%)	7.1e6 (72%)	6.9e5 (57%)	7.9e6 (72%)	7.8e6 (79%)	6.8e5 (57%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.6e5 (55%)	5.3e6 (48%)	5.2e6 (53%)	6.5e5 (54%)
CDT-Grid I	1.2e6 (11%)	NP	8.6e4 (7%)	1.2e6 (11%)	NP	8.6e4 (7%)
OOC CDT-Grid II	1.2e6 (11%)	NP	8.6e4 (7%)	1.2e6 (11%)	NP	8.6e4 (7%)
	Res = 4000^3, #GP = 2.3e8, size = 1.2GB, simsize = 1.4GB			Res = 5000^3, #GP = 3.6e8, size = 1.8GB, simsize = 2.1GB		
DT-Grid	3.3e6 (30%)	1.5e6 (15%)	NP	2.1e6 (19%)	1.3e6 (13%)	NP
OOC DT-Grid I	7.6e6 (69%)	7.5e6 (76%)	7.9e5 (66%)	7.7e6 (70%)	7.5e6 (76%)	7.9e5 (66%)
OOC DT-Grid II	8.1e6 (74%)	7.2e6 (73%)	6.8e5 (57%)	8.2e6 (75%)	6.7e6 (68%)	6.6e5 (55%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.6e5 (55%)	5.4e6 (49%)	5.3e6 (54%)	6.6e5 (55%)
CDT-Grid I	1.2e6 (11%)	NP	9.3e4 (8%)	1.2e6 (11%)	NP	9.2e4 (8%)
OOC CDT-Grid II	1.2e6 (11%)	NP	9.4e4 (8%)	1.2e6 (11%)	NP	9.4e4 (8%)
	Res = 6000^3, #GP = 5.2e8, size = 2.6GB, simsize = 3.1GB			Res = 8000^3, #GP = 9.2e8, size = 4.7GB, simsize = 5.5GB		
DT-Grid	NP	NP	NP	NP	NP	NP
OOC DT-Grid I	7.7e6 (70%)	7.5e6 (76%)	7.9e5 (66%)	7.6e6 (69%)	7.5e6 (76 %)	6.7e5 (56%)
OOC DT-Grid II	8.3e6 (75%)	6.7e6 (68%)	6.5e5 (54%)	8.3e6 (75%)	6.7e6 (68%)	6.1e5 (51%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.6e5 (55%)	5.3e6 (48%)	5.3e6 (54%)	6.4e5 (53%)
CDT-Grid I	1.2e6 (11%)	NP	8.8e4 (7%)	1.2e6 (11%)	NP	NP
OOC CDT-Grid II	1.2e6 (11%)	NP	9.3e4 (8%)	1.2e6 (11%)	NP	9.2e4 (8%)

physical memory is exceeded. At resolutions of 6000^3 and above it is not even possible to initialize the original DT-Grid data structure due to lack of virtual memory. The performance of iterations with OOC DT-Grid I and OOC DT-Grid II are, although fluctuating, very close to the approximate upper bound, suggesting that stencil iterations are close to CPU and/or memory bound, in contrast to I/O bound. For the OOC CDT-Grid I, which compresses the topology in-core and streams the values uncompressed to disk, the performance is just above 50%. Since compression is relatively CPU intensive, this data structure does not perform as well for iterations as the other out-of-core data structures. The performance is worst for the OOC CDT-Grid II and the CDT-Grid I that both compress the values and the topology. Recall that the values are the most memory consuming part of the level set. Since statistical coding is relatively CPU intensive, this is to be expected. In our experience even very light weight compression schemes for the values are out-performed by their out-of-core counterparts. This is because the numerical computations hide the I/O latency, whereas an arithmetic coder will compete for CPU time with the computationally demanding level set or fluid computations. This overall behavior is also supported by the throughputs of the level set simulations, although there are some differences. Again the performance of the original DT-Grid starts to degrade quite early due to lack of physical memory. At a resolution of 2500^3 the throughput has dropped to a mere 18%. In contrast the performance of OOC DT-Grid I is more than 3.5 times faster. For simulations where all pages fit in-core, the performance of OOC DT-Grid I, streaming only values to disk, is very close to its approximate upper bound. However, when not all of the pages fit in-core, the framework becomes I/O limited and the performance drops to approximately 65%. This performance remains constant even for simulations involving file sizes of several gigabytes. At resolutions of 8000^3 the performance starts to degrade for OOC DT-Grid I since it stores the topology uncompressed in-core. Note that even though the topology takes up only a relatively small part of the total size of a DT-Grid, the high resolution of our level sets imply that topology combined with buffering for the out-of-core component can start to fill up the available memory. For both OOC DT-Grid II and OOC CDT-Grid I the performance is roughly the same and centered around 55% throughout the tests. The performance of OOC DT-Grid II seems to degrade a little for very large level sets. In fact, given enough memory OOC CDT-Grid I performs superior since it compresses the topology in-core and only stores values out-of-core.

Clearly no variation of our framework performs as well as pure in-core simulations using state-of-the-art data structures. However we stress that our out-of-core framework generally delivers more than 50% of this in-core peak performance—even for very high resolution simulations requiring up to 5.5GB of storage for a single level set. The only exceptions are framework variations employing value compression. The value codecs are relatively CPU intensive and quantize the numerical distances to obtain good compression ratios. Given a maximum allowed distortion this is convenient for static models or level sets corresponding to particular frames in an animation. However for online simulations care must be taken in order to avoid the compression error to accumulate as discussed in Section 6.2. We therefore advocate utilizing the out-of-core framework possibly in combination with compression of the topology for online simulations and a combined compression and out-of-core framework for reduced offline storage and transfer of data needed later in a production pipeline.

8.1.3 *Offline Compression.* Next we compare our compression framework to the compression scheme for hexahedral volume meshes by Isenburg and Alliez [2002] and associated scalar values by Mascarenhas et al. [2004]. Unfortunately we are not able to make a direct comparison to the benchmarks reported in Mascarenhas et al. [2004]. First, only one of the data sets used in the paper seems to be publicly available. Secondly, the tests are run on hardware that we do not have access to, and finally only the cell counts are reported. Fortunately, source code for the encoder of Isenburg and Alliez is available online. It was then straightforward to extend this source code with the scalar value compression method introduced in Mascarenhas et al. [2004]. It should be noted that none of these compression techniques work out-of-core and actually use a significant amount of memory (this is also noted in Isenburg [2004], chapter 6). Consequently we are limited to evaluating relatively small grids sizes compared to what is used elsewhere in this paper. Table III lists the compression times and compressed sizes of several models. In these tests we used a 14-bit quantization for the values. The times listed include only the time spent on compression. This is due to the fact that the two methods we compare use different data structures and the setup and load time of these differ greatly. In particular the load time of the data structure by Isenburg is significantly longer than the time for loading a DT-Grid into memory. From Table III it can be seen that on average our method is about 10 times faster and compresses 14% better.

Finally, Tables IV and V summarize performance of our compression framework applied to level set models with original sizes in the order of several gigabytes. Since all models are available from the public Stanford Scanning Repository, we hope these tables might establish benchmarks for future evaluations of level set compressions. The timings include streaming to and from disk, and again a 14-bit quantization was applied to the distance values. For these models our compression method produces between 76% and 93% compression when compared to an uncompressed DT-Grid representation. When compared to an uncompressed full grid representation, our method consistently gives more than 99% compression for all these examples. Note that the latter percentage is the one that should be used when comparing our method to a volumetric method that compresses the entire clamped signed distance field volume. From

Table III.
Comparison Between the Performance of Our Compression Framework and the Methods of Isenburg et al. and Mascarenhas et al. . Narrow Band Width $\gamma = 5$. Tests Run on an 2.41GHz AMD with 1GB of Memory

Our Method		Isenburg/Mascarenhas	
Comp Time	Comp Size	Comp Time	Comp Size
Bunny, $89 \times 88 \times 71$, 0.703 MB, 153818 grid points			
0.188 s	0.160 MB	1.92 s	0.187 MB
Buddha, $128 \times 58 \times 57$, 0.778 MB, 173005 grid points			
0.218 s	0.189 MB	2.19 s	0.217 MB
Statuette, $88 \times 55 \times 49$, 0.303 MB, 66942 grid points			
0.094 s	0.0789 MB	0.844 s	0.0872 MB

Table IV we can furthermore see the very low memory footprint of the Slice Cache and probability tables associated with the arithmetic coder. These two components are the main consumers of memory in our offline compressor, since our prefetcher only utilized a single 32KB buffer for each component (6 in total). Hence for the largest model compressed in the examples presented here, the overall memory usage is approximately 13MB. Notice also how efficiently our method compresses the topology of the grid. This is evident from Table V where the percentages of compression for the individual components are listed. The only component that is not efficiently compressed is $coord_{1D}$ which the DT-Grid in many cases already represents using very few bytes due to its hierarchical index compression (recall Figure 3).

Using the Metro tool [Cignoni et al. 1996] it is possible to measure the distortion between two meshes as the Hausdorff distance normalized to the length of the bounding box diagonal. In our case we can measure the distortion between meshes extracted from the decompressed (includes quantization artifacts) and the original narrow band distance volumes. In general the distortion decreases as the resolution increases. This is simply due to the fact that quantization is applied in the narrow band whose (Euclidean) width is decreasing as the grid resolution increases. In Tables IV and V the distortion measured on the bunny in lowest resolution was 8.7^{-5}, which is the same order of magnitude as the distortions reported in Lee et al. [2003]. We were not able to measure the distortion on the higher resolution distance fields due to the large amount of triangles generated by the extraction, but as we have argued, the distortion decreases as the resolution is increased. However, in the future we plan to implement a Metro tool equivalent that operates directly on the level set. Using the framework described in this article the Hausdorff distance can be evaluated simply by streaming the narrow band distance fields through memory.

9. APPLICATIONS

In addition to the benchmarks presented in the previous section we now present several applications of our out-of-core and compression framework. This includes: 1) Shape modeling and deformations, 2) particle level sets, 3) out-of-core and compressed fluid simulations, 4) out-of-core mesh-to-level-set scan conversion, 5) out-of-core linear algebra and, 6) out-of-core solutions of Partial Differential Equations (PDEs) embedded on high resolution surfaces. In this section the machine specifications are listed along with the description of each particular application.

Table IV.
Compression Statistics for Various Level Set Models. Narrow Band Width, $\gamma = 3$. Quantization of Values to 14 Bits Per Grid Point. Most of the Captions Should be Self-Explanatory Except *Grid Point Count* Which is the Number of Grid Points in the Narrow Band, *% Compress/DT-Grid* Which is the Percentage of Compression Measured Against the Uncompressed DT-Grid Representation, *% Compress/FullGrid* Which is the Percentage of Compression Measured Against an Uncompressed Full-Volume Grid Containing the Narrow Band (this illustrates the efficiency of our framework seen as a full-volume compressor applied to a clamped signed distance field), and Finally *Max Mem* Which is Reported for the *Slice-Cache* and the Probability Tables (*Prob-Table*) in MB

Model	Compress Wall Time (secs)	Decompress Wall Time (secs)	Orig Size (MB)	Comp Size (MB)	Bits Per Grid Point Compressed	Grid Point Count	% Compress/ DT-Grid	% Compress/ Full-Grid	Max Mem Slice-Cache/ Prob-Table
Lucy									
487 × 281 × 833	4.61	4.42	17.8	4.43	9.10	4.09e6	75	99	0.31 / 1.5
1987 × 1142 × 3409	78.8	73.7	303	68.4	8.24	6.96e7	77	99	1.3 / 5.8
3987 × 2290 × 6844	313	301	1226	254	7.59	2.81e8	79	99	2.7 / 15
David									
1186 × 487 × 283	7.27	7.05	28.6	6.46	8.29	6.54e6	77	99	0.37 / 2.0
4864 × 1987 × 1149	122	117	489	93.2	7.01	1.12e8	81	99	1.6 / 9.6
9768 × 3987 × 2304	487	469	1975	303	5.63	4.51e8	85	99	3.3 / 23
Bunny									
491 × 487 × 381	3.73	3.63	15.8	2.91	7.18	3.41e6	82	99	0.20 / 0.82
1991 × 1974 × 1544	59.0	60.0	264	26.6	3.92	5.69e7	90	99	0.95 / 1.7
3991 × 3956 × 3094	237	239	1064	85	3.10	2.29e8	92	99	2.1 / 3.1
Buddha									
1195 × 494 × 493	13.7	12.9	55.5	11.1	7.65	1.21e7	80	99	0.55 / 1.2
4848 × 1996 × 1993	213	210	919	116	4.86	2.01e8	87	99	2.4 / 3.7
9718 × 3998 × 3993	888	873	3695	370	3.84	8.08e8	90	99	5.0 / 8.0

Table V.
Compression Statistics for Various Level Set Models. Narrow Band Width, $\gamma = 3$. Quantization of Values to 14 Bits Per Grid Point. The Table Lists the Bits Per Grid Point (*BPGP*) for the Original and Compressed Topology and Values Respectively. Additionally the Percentage of Compression (*% Comp*) is Listed for Each Individual Component of the Topology as Well as the Values. The Number in Parenthesis is the Percentage that this Particular Component Takes up of the Entire Uncompressed DT-Grid

Model	Orig Topology BPGP	Comp Topology BPGP	Orig Values BPGP	Comp Values BPGP	% Comp $coord_{1D}$	% Comp $coord_{2D}$	% Comp $coord_{3D}$	% Comp val_{1D}	% Comp val_{2D}	% Comp val_{3D}	% Comp acc
Lucy											
487 × 281 × 833	4.50	0.405	32	8.69	0 (0.0)	85 (0.0)	80 (5.4)	98 (0.0)	97 (1.4)	73 (88)	100 (5.5)
1987 × 1142 × 3409	4.57	0.256	32	7.34	0 (0.0)	89 (0.0)	88 (5.6)	99 (0.0)	99 (1.3)	77 (88)	100 (5.6)
3987 × 2290 × 6844	4.57	0.288	32	7.95	0 (0.0)	88 (0.0)	86 (5.6)	99 (0.0)	98 (1.3)	75 (88)	100 (5.6)
David											
1186 × 487 × 283	4.67	0.376	32	7.91	0 (0.0)	84 (0.0)	83 (5.7)	99 (0.0)	97 (1.4)	75 (87)	100 (5.7)
4864 × 1987 × 1149	4.74	0.270	32	6.74	0 (0.0)	87 (0.0)	88 (5.8)	99 (0.0)	99 (1.4)	79 (87)	100 (5.8)
9768 × 3987 × 2304	4.74	0.232	32	5.40	0 (0.0)	87 (0.0)	89 (5.8)	99 (0.0)	99 (1.4)	83 (87)	100 (5.8)
Bunny											
491 × 487 × 381	7.00	0.254	32	6.92	0 (0.0)	85 (0.0)	91 (7.2)	98 (0.0)	99 (3.6)	78 (82)	100 (7.2)
1991 × 1974 × 1544	6.94	0.223	32	3.69	0 (0.0)	88 (0.0)	92 (7.2)	99 (0.0)	99 (3.5)	88 (82)	100 (7.2)
3991 × 3956 × 3094	6.94	0.213	32	2.89	0 (0.0)	89 (0.0)	92 (7.2)	99 (0.0)	99 (3.5)	91 (82)	100 (7.2)
Buddha											
1195 × 494 × 493	6.38	0.265	32	7.38	0 (0.0)	79 (0.0)	90 (6.8)	96 (0.0)	99 (3.0)	77 (83)	100 (6.8)
4848 × 1996 × 1993	6.36	0.215	32	4.65	0 (0.0)	84 (0.0)	92 (6.8)	98 (0.0)	99 (2.9)	88 (83)	100 (6.8)
9718 × 3998 × 3993	6.35	0.203	32	3.64	0 (0.0)	86 (0.0)	92 (6.8)	99 (0.0)	99 (2.9)	89 (83)	100 (6.8)

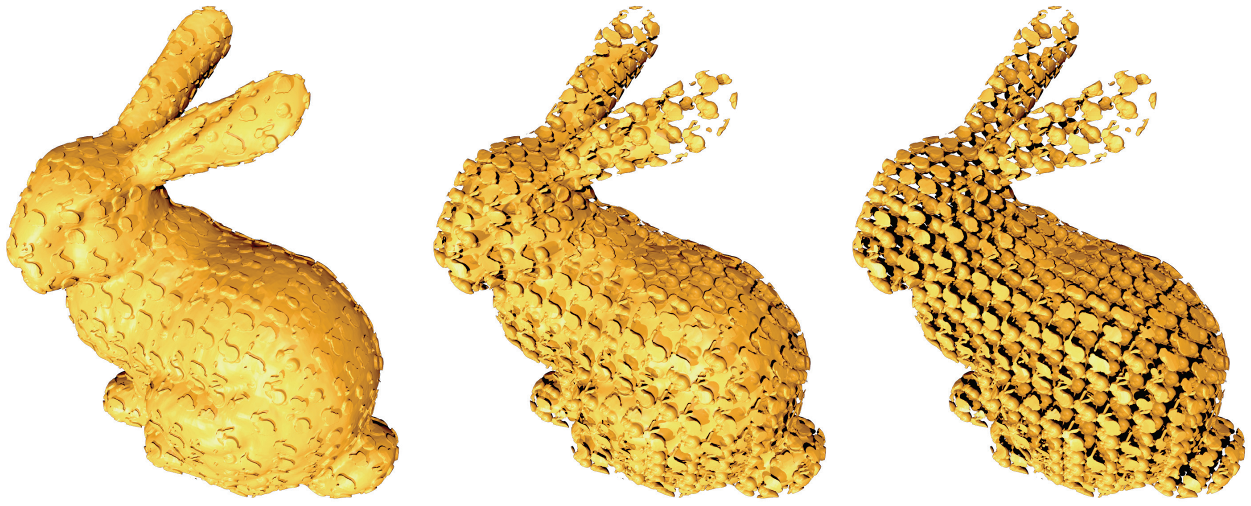
Fig. 12. Level set morph between a 2048^3 bunny (304MB) and a highly detailed level set CSG bunny (1.62GB) modeled as an out-of-core CSG intersection of the 2048^3 bunny (304MB) with a tiling of 20^3 smaller bunnies each at resolution 128^3 (7.47GB). Reported sizes are in uncompressed DT-Grid format. Peak storage requirements for the morph are close to 5GB.

Table VI.
Benchmark Numbers From the "Enright test". The Table Provides the Number of Grid Cells in the Narrow Band of the DT-Grid, the Number of Particles Used by the Particle Level Set and the Memory Footprint of the Simulation with and without Quantization. The Numbers Given in this Table are Calculated at the First Iteration of the Simulation

	800^3	1024^3	1500^3	4096^3
Number of Grid cells	2.35M	3.85M	8.26M	61.6M
Number of Particles	34.6M	56.7M	122M	910M
Simulation Memory Footprint	1.22 GB	2.20 GB	4.4 GB	32.1 GB
Simulation Memory Footprint (quantized)	428 MB	772 MB	1.55 GB	11.3 GB

9.1 Shape Modeling and Deformations

The rightmost image in Figure 12 shows a highly detailed level set model of a bunny (1.62GB) modeled as an out-of-core Constructive-Solid-Geometry (CSG) intersection between a level set bunny at resolution 2048^3 (304MB) and a 20^3 tiling of similar smaller level set bunnies each at resolution 128^3. The total size of the tiling of level set bunnies is 7.47GB. The sizes are in uncompressed DT-Grid format. Figure 12 depicts three frames from a shape metamorphosis between the CSG bunny and the bunny at resolution 2048^3. The simulation was run in 32bit Windows XP Pro on a 2.41GHz AMD machine with 1GB of physical memory and a Western Digital Raptor disk. The peak space requirements for this simulation are close to 5GB in uncompressed DT-Grid format. The simulation was run out-of-core at approximately 65% of the peak in-core DT-Grid performance. Note that for large simulations like this, OS paging is not even possible due to OS memory limits for a single application.[14] The uncompressed storage-requirements for the entire simulation were 342 GB. Using the compression method described in this paper we compressed it to 83.6 GB (258 GB saved in total) without introducing noticeable distortion. The grids were subsequently rendered directly using a ray tracer with ray leaping of the level set.

9.2 Out-of-Core and Compressed Particle Level Sets

To benchmark our out-of-core and compressed particle level set we use the test from [Enright et al. 2002] where a sphere with radius 0.15 is placed in a unit computational domain at (0.35, 0.35, 0.35) and advected in the following periodic and divergence free velocity field [LeVeque 1996]

$$\begin{aligned} u(x, y, z) &= 2sin^2(\pi x)\,sin(2\pi y)\,sin(2\pi z)\,cos(\pi t/T) \\ v(x, y, z) &= -sin(2\pi x)\,sin^2(\pi y)\,sin(2\pi z)\,cos(\pi t/T) \\ w(x, y, z) &= -sin(2\pi x)\,sin(2\pi y)\,sin^2(\pi z)\,cos(\pi t/T), \end{aligned} \tag{1}$$

where T is set to 3. In Table VI we list information about the size of the particle level set used to represent the sphere. Table VII summarizes the results when the simulation is run on a 2.4 GHz AMD CPU with 2 GB of paired DDR400 memory and a 300GB Maxtor SATA disk. The numbers provided are the average time per iteration divided by the number of grid cells[15] used by the simulation. This gives a good measurement of the performance of the particle level set since all our particle level set operations scale with the number of grid cells in the narrow band. Table VII clearly shows that the

[14]In 32-bit Windows XP Pro this is limited to 3 GB.

[15]The number of grid cells is measured in millions.

Table VII.
Benchmark Numbers Showing Average Iteration Time Divided by the Number of Grid Cells (measured in millions) for Different Resolutions of the "Enright test". Each Value is Based on an Average of the First 7 Iterations of the Solver. (Q) Means that Our Quantized Particle Level Set was Used. In-Core Tests were Not Possible (NP) Due to Hardware Limitations for the 4096^3 Simulation and Only Possible Using Quantization for the 1500^3 Simulation

	800^3	1024^3	1500^3	4096^3
In-Core	23.1	238	NP	NP
Out-Of-Core	67.4	74.1	99.3	113
In-Core (Q)	24.1	24.3	25.4	NP
Out-Of-Core (Q)	24.4	26.0	26.4	39.4

in-core performance decreases drastically when the simulation runs out of memory and is forced to rely on OS page-swapping. Even though the 1024^3 simulation uses only slightly more than the available 2GB of memory, the performance has decreased significantly compared to the one at 800^3. This is most likely because the OS cannot predict which memory pages will be accessed next and therefore inadvertently swaps out data that is soon needed again. Note also that for all benchmark resolutions the performance of the quantized out-of-core particle level set is close to that of the in-core. Part of this is caused by the in-core caching of data, but at large resolutions, like 4096^3, this effect is naturally less significant.

9.3 Fluid Simulations

To showcase the capabilities of our out-of-core and compression framework we have use it for large-scale fluid animations. Our fluid solver is based on the Stable Fluid method of [Stam 1999] solved on staggered volumetric DT-Grids similar to Houston et al. [2006]. Our out-of-core and compressed particle level set is used to represent the fluid interface and the out-of-core level set framework is used for solid boundaries and surface velocities.

We present two fluid simulations of a fountain, the first with the solid boundary level set sampled at a resolution of $471 \times 495 \times 471$ voxels and the second at the resolution $931 \times 1007 \times 931$ voxels. These will be referred to as respectively the *Fountain* and *Large Fountain* simulations. Both of these simulations were performed on a 2.4 GHz AMD CPU with 2GB of memory and 300GB Maxtor SATA disk. Figures 1 and 13 show snapshots from the *Fountain* simulation and Figure 14 shows a snapshots from the *Large Fountain* simulation. The effective simulation domain needed to enclose the *Fountain* simulation is $471 \times 1078 \times 471$ voxels. For the *Large Fountain* $931 \times 1567 \times 931$ voxels are needed.

9.4 Out-of-Core Scan Conversion

Based on Sean Mauch's thesis work [2003], Houston et al. [2006] developed an efficient mesh to level set scan converter. Specifically it converts a closed oriented 2-manifold polygonal meshes into a narrow band signed distance field representation using $O(F + D)$ time and memory, where F and D denote respectively the number of faces in the mesh and the number of grid points in the narrow band. However, this algorithm only works in-core, and consequently resolutions of the input mesh and output level sets are limited by the available memory. In fact for most models used in this paper resolutions either exceed the virtual memory limits or result in OS page swapping. To address this problem we have developed an out-of-core extension to the method of Mauch [2003]. This extension allows us to scan convert manifold mesh models into level sets of unprecedented resolution. The only limitation on the sizes of input meshes and output level sets is the available disk space. As can be surmised from Table IX our out-of-core scan converter outperforms in-core equivalents when model resolutions exceed the available memory. Note also that the running times of the in-core scan converter increases rapidly when the memory limit is approached. When scan converting the bunny at 3000^3, the out-of-core scan converter is roughly four times faster than the in-core scan converter. To the best of our knowledge this is the first demonstration of a scan converter that works fully out-of-core.

Our out-of-core algorithm begins by sorting the input mesh to create a list of faces each represented by the vertex coordinates. This list of faces is next used to partition the mesh into a number of submeshes. Each submesh is then scan converted in-core using the method of Mauch [2003], and the generated grid points are streamed to disk. When all sub-meshes have been scan converted, the collection of generated grid points are sorted into (x, y, z) lexicographic order using an external sort. The lexicographic order is required for the construction of an out-of-core DT-Grid which constitutes the final step. More specifically our algorithm performs the following steps:

(1) The input mesh file is assumed to be a simple indexed triangle set such as ply or obj. As a prelude to the mesh partitioning we dereference all the vertex indices of the faces and create a list of faces, l_f, where each face is represented by the coordinates of its three vertices. Doing this naively using random access will in the worst case result in $O(F)$ I/O operations, where F is the number of faces, since each index may reference a vertex currently on disk. Instead we can create the list of faces by applying a number of external sorts similar to Chiang and Silva [1997]. This can be done in $O\left(\frac{F}{B}\log_{M/B}\frac{F}{B}\right)$ I/O operations, where F is the number of faces, M is the memory size and B is the number of faces per disk block. The time complexity is $O(F \log F)$. Briefly described, the dereferencing works as follows: First we sort the list of faces according to the first vertex index of each face. Next we simultaneously scan through the sorted list of faces and the list of vertex positions and replace the first vertex id of each face with its actual vertex coordinate. This step is subsequently repeated for the second and third index of each face respectively. The result is the list of faces, l_f, represented by their coordinates required for partitioning the mesh.

(2) Next the mesh is partitioned into $P \times Q \times R$ submeshes. P, Q, and R depend on the amount of available memory, and in general they are simply determined heuristically. The $(p, q, r)'th$ submesh consists of all the faces intersected by the $(p, q, r)'th$ subvolume resulting from dividing the bounding box of the mesh into $P \times Q \times R$ equally sized axis-aligned and nonoverlapping subvolumes. To do the actual partitioning, a single scan through the list of faces determines for each face which subvolumes it intersects. During this scan, data is streamed into a file, f, containing 7-tuples consisting of the three vertex indices, the coordinates of the three vertices, and the subvolume id that this face maps into. Assuming that at most a constant number of subvolumes intersect each face, the partitioning step requires $O(\frac{F}{B})$ I/O operations and has a time complexity of $O(F)$.

(3) To apply the method of [Mauch 2003], all of the submeshes are first converted into individual indexed face sets. This is done by a single external sort and a subsequent scan through the

Table VIII.
Scan Conversion Statistics for the Out-of-Core 2-Manifold Scan Converter. The Statistics are Divided Into Three Categories: Timings, Storage Requirements and the Narrow Band Grid-Point Count. The Timings Include the Mesh Load and Partitioning Time (*Mesh L&P*), Actual Scan Conversion Time (*Scan Conv*), Lexicographic Sort Time (*Lexi Sort*) and DT-Grid Construction Time (*DTGrid Const*). The Storage Requirements Include the Original Mesh (*Mesh I*), the Intermediate Sub-Mesh Structure (*Mesh II*), the Index-Value Container (*IVC*) Storing 4-Tuples of Grid Point Indices and Corresponding Values (see step 5 below), and the Final (uncompressed) DT-Grid (*DT-Grid*). The Explanations of the Steps Involved in the Timings and the Representations for Which Storage Sizes are Given can be Found Below. The Narrow Band Width $\gamma = 3$. All Models are Courtesy of the Stanford Scanning Repository

	Time [min:sec]					Size [MB]				
Model	Mesh L&P	Scan Conv	Lexi Sort	DTGrid Const	Tot	Mesh I	Mesh II	IVC	DTGrid	Grid-point Count $\times 10^9$
Lucy, #faces = 28055742, #vertices = 14027872										
35000 × 20000 × 11500	70:50	278:24	839:40	198:21	1387:15	482	591	108060	36070	7.08
David, #faces = 7227031, #vertices = 3614098										
29500 × 12000 × 7000	12:58	146:06	465:51	108:12	733:09	124.1	159.0	62325	20991	4.08
Bunny, #faces = 70064, #vertices = 35034										
12000 × 12000 × 9500	0:07	179:29	272:04	54:37	506:17	1.203	3.652	31619	10307	2.07
Buddha, #faces = 1087716, #vertices = 543652										
24500 × 10000 × 10000	1:40	318:01	617:30	143:14	1080:30	18.67	32	77340	27910	5.07

Table IX.
Comparison of Timings (measured in seconds) Between the In-Core and the Out-of-Core Scan Converters for Scan Converting the Stanford Bunny in Increasing Resolution with a Narrow Band Width of $\gamma = 3$. The Size in MB of the Uncompressed DT-Grid is Given Below the Resolution. Using the In-Core Scan Converter it is Not Possible (NP) to Scan Convert the Stanford Bunny in Resolution 4000^3 Because the Overall Memory Usage of Mesh, Level Set and Intermediate Data Structures Exceed the Virtual Memory Limits. The DT-Grid Construction is Time Not Included Since the In-Core Scan Converter is Not Capable of Generating the DT-Grid for Resolutions Equal to and Above 3000^3 Due to Virtual Memory Limits

Method	250^3 4.005 MB	500^3 16.97 MB	1000^3 69.80 MB	2000^3 283.1 MB	3000^3 655.2 MB	4000^3 1718 MB
In-Core	4.375	7.453	24.72	153.9	4227	NP
Out-Of-Core	10.20	21.25	67.61	423.1	1096	2049

sorted tuples in f. First the 7-tuples in f are sorted according to their subvolume id. Next a scan through f generates an indexed mesh representation for each sub-mesh. Since f is sorted according to sub-volume id, the generation of a new sub mesh commences as soon as a new sub volume-id is encountered. Internally for each sub mesh, local vertex and face indices are created with the use of a map data structure mapping from global to local indices. The individual indexed sub meshes are progressively streamed to disk. Again this step requires $O\left(\frac{F}{B}\log_{M/B}\frac{F}{B}\right)$ I/O operations and has a time complexity of $O(F\log F)$.

(4) Next the in-core scan converter of [Mauch 2003] is separately applied on each sub mesh. The coordinates of the narrow band grid points and associated signed distance values generated for each sub mesh are streamed to disk as 4-tuples $\{x, y, z, \phi_{x,y,z}\}$. This is done in a way similar to Houston et al. [2006] which ensures an $O(F + D)$ time usage of the entire algorithm, where D is the number of defined grid points in the narrow band. The peak memory consumptions of the algorithm corresponds to the size of the largest sub mesh plus the size of a single sub volume. In terms of I/O operations the complexity of this step is $O(\frac{F+D}{B})$.

(5) Finally the 4-tuples generated above are sorted into (x, y, z) order using a single external sort and the out-of-core level set is constructed by sequentially inserting grid points into the lexicographic data structure of a DT-Grid. Due to the external sort of grid points, this step requires $O\left(\frac{D}{B}\log_{M/B}\frac{D}{B}\right)$ I/O operations and has a time complexity of $O(D\log D)$.

Note that steps (3) and (4) can be performed simultaneously such that a submesh is scan converted as soon as it is generated. In this way, it is not necessary to stream the submeshes to disk.

For large grids, the external sorting in Step 5 is usually the most time-consuming due to the large number of narrow band grid points generated. For the models in Table VIII, the number of grid points are on the order of billions. Hence it is important to sort the grid points using only a single pass over the file as opposed to three passes. In our experience this gives a factor of 2.5 improvement in the time spent on sorting. In practice the size of the sub volumes can just be set to a heuristically determined size. For the scan conversions presented in this paper we use a fixed subvolume of size 256^3. Our method uses a fairly large amount of intermediate disk space, but storage requirements are still linear in the number of mesh faces and narrow band grid points. The disk requirements can be reduced by

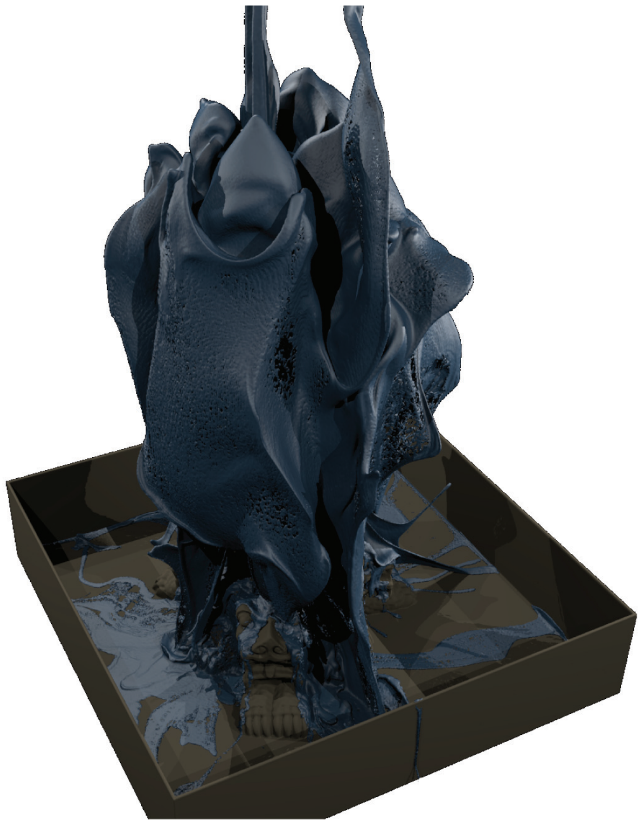
Fig. 13. Fluid simulation with an effective bounding box of 471 × 1078 × 471. The scene contains a total of 43.1M voxels and 573M particles.

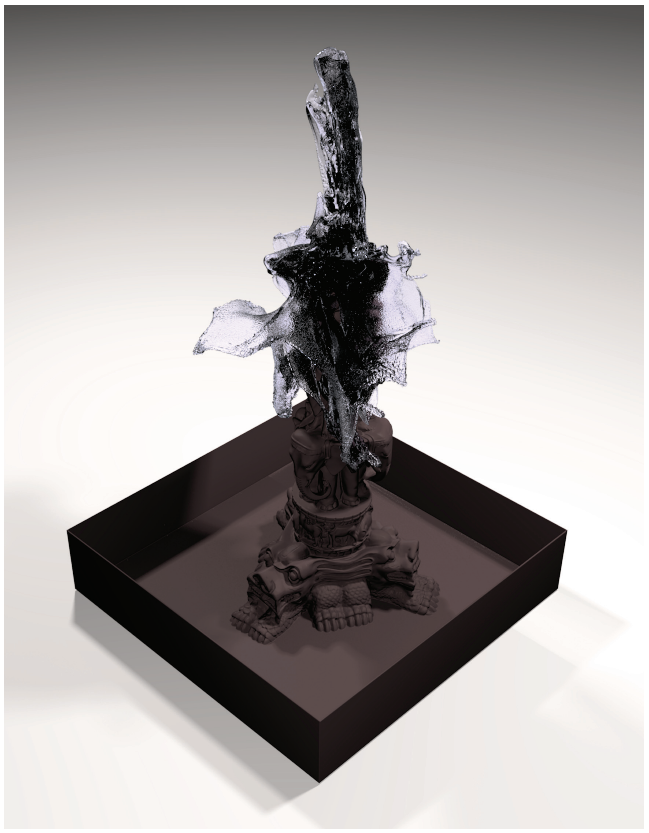
Fig. 14. Fluid simulation with an effective bounding box of 931 × 1567 × 931. The scene contains a total of 61.7M voxels and 332M particles.

means of compression and a more compact sorting like Matias et al. [2000]. This will however increase running times.

Table VIII lists several high resolution level set models and scan conversion times. All scan conversions were done on an AMD 2.41GHz machine with 1GB of main memory and a Western Digital WD4000YR hard disk. The highest resolution model is the Lucy which was scan converted into a 35000 × 20000 × 11500 narrow band level set containing 7.08 billion grid points in the narrow band. To the best of our knowledge this resolution is about an order of magnitude larger than previously demonstrated in each dimension. We realize that some of the models in Table VIII are over sampled in terms of triangle to voxel resolution and the high resolutions are meant merely to illustrate the power of the out-of-core framework.

9.5 Out-of-Core Linear Algebra

Systems of linear equations are often encountered in computer graphics applications. Examples include solving the Poisson equation in the pressure projection step of the stable fluids method Stam [1999] as well as integrations in time or space employing implicit techniques like backward Euler or Crank-Nicholson. Typically these systems of linear equations are sparse and positive definite which means they can conveniently be solved using techniques like the conjugate gradient (CG) method [Hestenes and Stiefel 1952]. This iterative algorithm, and several others like Gauss-Seidel, works by successively performing matrix vector operations until some convergence criteria are satisfied.

Given the fact that our framework facilitates level set models of sizes that far exceed the available memory it is therefore important to have access to an out-of-core linear algebra library. To achieve this we have augmented our out-of-core framework with a matrix vector library supporting standard linear algebra operations. Specifically we only consider algebraic operations that can be formulated using sequential stencil access patterns to the elements of the matrices and vectors. This is to avoid inefficient random access in the underlying out-of-core data structure on disk. Fortunately, as in the case of level set operations, most matrix vector operations are inherently sequential.

Our implementation is based on the same out-of-core array that stores scalar distance values in the level set framework. It maps currently accessed parts of an array to in-core pages and streams the remaining data to disk. It has been extended with out-of-core vector and matrix classes that supports basic operations like $+$, $-$, $*$, norms and dot-products. Our linear algebra library also supports the following sparse storage formats: Coordinate storage, compressed row storage, and diagonal banded storage.

We have made an extra effort to optimize a special sparse septa-diagonal matrix class since it is used extensively in the application discussed in the next section. We make the following constraint for this matrix class: Each of the column indices of the seven nonzero elements on each row is increasing between two consecutive rows. This constraint is naturally fulfilled by matrices derived from the discretization of linear operators on structured lexicographic grids like dense grids or sparse DT-Grids. It is important since it implies that the spacial coherency of a differential operator is preserved during discretization. This in turn means we can utilize the stencil iterators derived for cache-efficient access during level set operations.

Specifically the sparse septa-diagonal matrix is optimized to use less storage than a corresponding compressed row storage format, and in addition calculate matrix vector multiplications ($\mathbf{Ax} = \mathbf{b}$) while looping over the $\mathbf{x}$ vector and the matrix $\mathbf{A}$ *only once*. This is feasible by using stencil access into $\mathbf{x}$. To see this consider row r_i of the multiplication $\mathbf{Ax}$. The seven nonzero elements of $\mathbf{A}$ are multiplied with corresponding elements of $\mathbf{x}$ and the sum is formed. Next, to compute row r_{i+1} of $\mathbf{Ax}$ the column index of each of the elements in the row is *incremented* by at least one. The corresponding indices into $\mathbf{x}$ follow the same incremental access pattern. By employing a stencil of iterators in $\mathbf{x}$ we can perform the multiplication with a single stencil pass. The upper bound for a 7-point sequential stencil iteration is 7 passes. Using the out-of-core framework this number comes close to optimal in practice. Similar behaviors were discussed in section 5 and 8.1. We finally note that similar optimized matrix vector multiplication schemes can readily be derived as long as the matrix satisfies the simple constraint mentioned above.

To evaluate the performance of our out-of-core matrix vector library we have used a conjugate gradient reference implementation, SparseLib and IML++ [Dongarra et al. 1994], and benchmarked our sparse septa-diagonal matrix against the compressed sparse row matrix bundled with IML++. The physics behind the system of linear equations used in this benchmark is discussed in the next section. For now it suffices to say that the matrices are septa-diagonal and the systems are solved to the same precision.[16] The results are shown in Table X and clearly demonstrate the performance of the out-of-core matrix vector library. For reasonable in-core sizes the SparseLib classes naturally perform better, whereas the out-of-core counterpart is 50% slower. However, as the size of the system of equations increases beyond the available main memory the advantage of the out-of-core library is evident. Whereas the in-core solver suffers from considerable slowdown the throughput rate of the out-of-core solver is almost unaffected. Additionally, the out-of-core solver is not hampered by main memory address space limitations: We show simulations that would need to allocate approximately 9 GB of memory. Simulations of this size are not possible to run with algorithms that rely on 32-bit OS paging such as the Krylov-subspace method by Toledo [1995]. We conclude that with our out-of-core linear algebra library it is possible to solve very large sparse linear systems at roughly a sixth of the peak in-core speed. At this point, however, most of the matrix vector operations are I/O bound and a combination of a specialized method such as Toledo's and our high performance matrix vector library could prove fruitful but is left for future work.

9.6 Solving PDEs on Large Implicit Surfaces

As an out-of-core application of our framework we demonstrate how we can solve PDEs directly on large level set surfaces. Specifically we solve the wave-equation embedded on a surface which can be expressed as [Xu and Zhao 2003]

$$\frac{\partial^2 f}{\partial t^2} = \nabla_s^2 f = \nabla^2 f - \frac{\partial^2 f}{\partial \mathbf{n}^2} - \kappa \frac{\partial f}{\partial \mathbf{n}} \qquad (2)$$

where $\nabla_s^2 f$ is the surface Laplacian of some quantity f (e.g., pressure) embedded on a level set surface, ϕ, $\kappa = \nabla \cdot \nabla \mathbf{n}$ is the mean curvature of ϕ and $\partial/\partial\mathbf{n}$ denotes the directional derivative along the surface normal $\mathbf{n} = \nabla\phi/|\nabla\phi|$. This second order time-dependent partial differential equation describes the propagation of a wave embedded on a level set surface. To eliminate the last two terms in Equation (2) we simultaneously solve the transport equation $\partial f/\partial t = S(\phi)\partial f/\partial\mathbf{n}$ till steady state, so that $\partial f/\partial\mathbf{n} = \mathbf{0}$. $S(\phi)$ is simply a sign function of ϕ that guarantees that information propagates in the correct direction, i.e., away from the surface. For increased performance we employ an implicit Crank-Nicholson time discretization scheme that is unconditionally stable. To avoid excessive storage usage the simulated scalar fields can be compressed with any of our codecs. For this type of scalar field we have found the Lorenzo predictor to perform best. It can compress 14-bit quantized data down to 3 bits per grid point value without visual artifacts. Examples of the wave equation propagating on complex geometry is shown in Figures 15, 16, and 17. Note that the former visualizes the scalar field f as actual 3D displacements of the geometry, whereas the latter two uses a simple color map of f.

[16]Note however that the amortized throughputs listed in Table X are independent of the precision.

Table X.
Benchmarking the Solution of Equation Systems Using the Conjugate Gradient (CG) Algorithm with Different Data Structures. Performance is Measured as Throughput Grid Points Per Second. Specifically the Amortized Throughput Over a Full Solution: Number of CG Iterations Times Number of Unknowns Divided by Time in Seconds. Memory Footprint Denotes the Theoretical In-Core System and Includes an Uncompressed DT-Grid. Timings were Performed on a Macintosh G5 2.5GHz with 2GB Main Memory and a Western Digital WD1600JD Hard Drive

Model	*Memory*	*Unknowns*	*OOC MV*	*Sparselib*
Sphere 100^3	4MB	40K	107K	1.5M
Bunny 256^3	70MB	840K	970K	2.2M
Tog logo 1024^3	520MB	6.1M	1.3M	2.2M
Bunny 1024^3	2GB	28M	360K	150K
Tog logo 4096^3	9GB	110M	320K	N/A

10. CONCLUDING REMARKS

We have presented a novel level set framework that for the first time allows representation and deformation of extreme resolution models, the only limitation being the amount of available disk space. Our applications include fluid animations, shape modeling and transformations, scan conversion, and the solution of the wave equation on large surfaces. The framework is based on two key components: out-of-core data management and compression. The main contributions of the out-of-core component is an application-specific and near optimal paging policy as well as a fast prefetching algorithm. The compression component contributes with codecs optimized for level set distance values, topology as well as associated scalar fields and particles. The performance of level set simulations in our out-of-core component combined with compression of topology was shown to be 50%–65% of the peak performance of the original in-core DT-Grid which in turn has been shown to outperform other state-of-the-art level set data structures [Houston et al. 2006]. The performance of the out-of-core component is preserved for simulations at resolutions that far exceed physical memory. In contrast the performance of the normal in-core DT-Grid was shown to dramatically decrease until the OS virtual memory limit prevented the simulation from running. Benchmarks indicate that our application-specific compression scheme is better and faster than a related volumetric band compression method. It should be noted that our framework is intended for very large level set simulations,

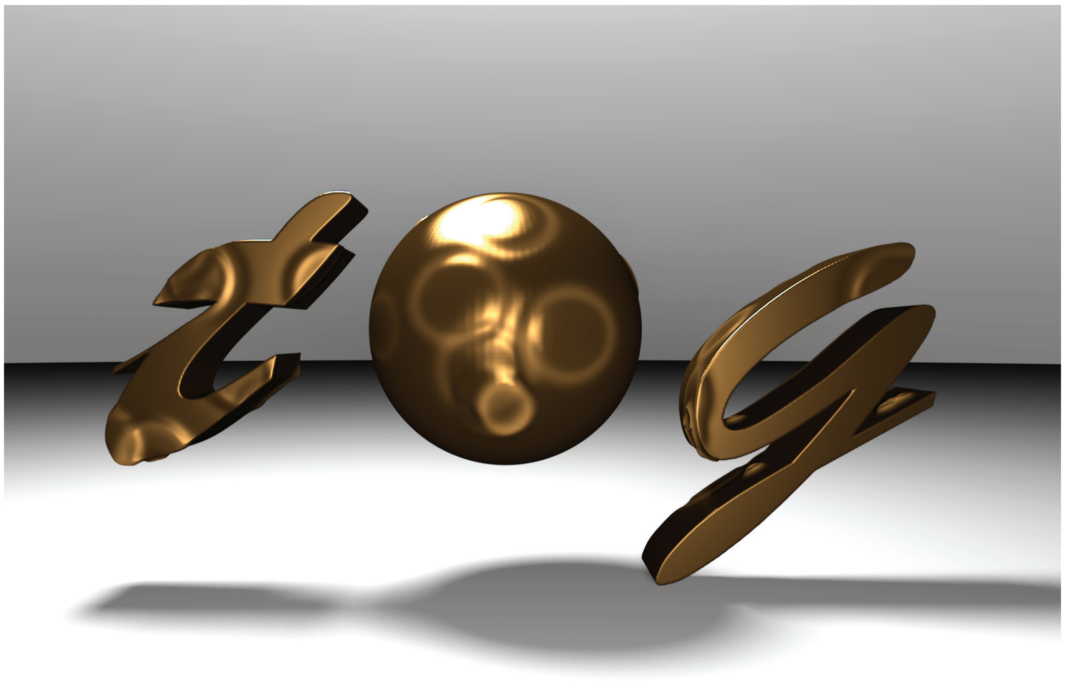

Fig. 15. The wave equation simulated on detailed 3D text. Distribution of pressure is visualized both through the displaced surface and surface properties. Resolution of model 512 × 250 × 200.

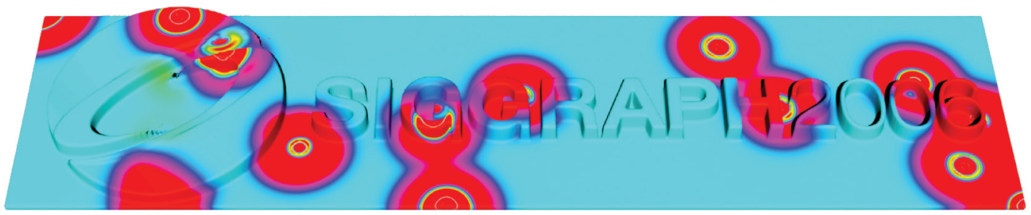

Fig. 16. The wave equation on highly detailed manifold. Resolution of model 1024 × 250 × 50. Distribution of pressure is visualized by color coded surface properties.

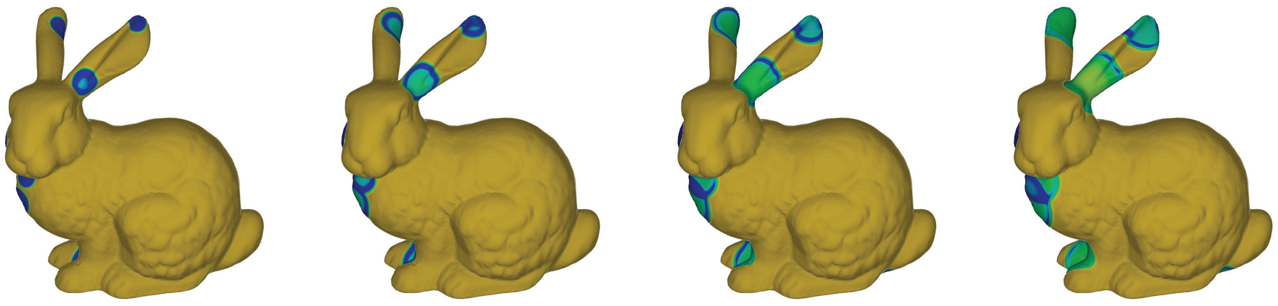

Fig. 17. Evolution of the wave equation on 1024^3 Stanford Bunny model. Distribution of pressure is visualized by color coded surface properties.

given the fact that it obviously cannot outperform pure in-core simulations given sufficient memory. The out-of-core and compression approach taken in this paper is not the only means for increasing the resolution of level set and fluid simulations. In particular parallelization techniques utilizing several computational nodes have recently been demonstrated [Irving et al. 2006; Geiger et al. 2006]. This effectively increases the total amount of memory available for simulation. We stress that our techniques should not be seen as an alternative to these parallelization methods, in fact combining them is a promising direction for future work. In addition we intend to investigate out-of-core, compression and parallelization techniques in the context of adaptive level set methods where the grid cell size can vary along the interface hence potentially increasing performance even further [Milne 1995; Sussman et al. 1999; Tang et al. 2003].

Our framework is primarily optimized for the sequential data access patterns encountered in level set simulations. Nevertheless, almost all of the applications included in this paper completely avoid random access. The only exception is ray-tracing, for which we are currently exploring improved prefetching strategies. We also note that simultaneous access to multiple out-of-core data structures will generally reduce performance due to the latency of disk seeks. We are currently investigating this by exploring strategies for automatically assigning resources when several out-of-core data structures are in play simultaneously. Our work addresses some of the major memory bottlenecks in fluid animations, namely particle level sets for the fluid interface, large boundaries and surface velocities. The out-of-core particle level set method as described in this paper is limited to particles moving at most one grid cell per iteration. Extending it to semi-Lagrangian advection techniques is straightforward as long as the maximum distance traveled by a particle is limited to a few grid cells. This is the case when utilizing narrow bands. However, we note that it will require iteration with larger stencils and require more bits to represent the quantized particles with the same precision. Hence we leave an investigation of the performance of such a method for future work. Furthermore, our particle level sets do not employ any strategies for adaptive particle seeding. Though such strategies can easily be used in conjunction with our compressed and out-of-core particle level sets, we leave this for future work. It should be stressed that this article does not focus on streaming and compression of the volumetric fluid velocity and pressure fields. These volumetric fields are already represented on DT-Grid data structures and hence immediately amenable for streaming and compression in our framework. However, due to the larger scope of this project we plan to report on this in a future paper.

In closing, we strongly believe our framework to have a significant impact in several key areas in computer graphics, including but not limited to, high-resolution fluid simulations and geometric modeling.

ACKNOWLEDGMENTS

We would like to acknowledge the Digital Michelangelo Project Archive, Stanford University. Thanks also goes to Brian Bunch Christensen and Gunnar Johansson for commenting on early drafts of this paper.

REFERENCES

ADALSTEINSSON, D. AND SETHIAN, J. A. 1995. A fast level set method for propagating interfaces. *J. Comput. Phys. 118*, 2, 269–277.

AKCELIK, V., BIELAK, J., BIROS, G., EPANOMERITAKIS, I., FERNANDEZ, A., GHATTAS, O., KIM, E. J., LOPEZ, J., O'HALLARON, D., TU, T., AND URBANIC, J. 2003. High resolution forward and inverse earthquake modeling on terasacale computers. In *Proceedings of SC03*. Phoenix, AZ.

BANSAL, S. AND MODHA, D. 2004. Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*.

BARGTEIL, A. W., GOKTEKIN, T. G., O'BRIEN, J. F., AND STRAIN, J. A. 2006. A semi-lagrangian contouring method for fluid simulation. *ACM Trans. Graph. 25*, 1.

BREEN, D. E. AND WHITAKER, R. T. 2001. A level-set approach for the metamorphosis of solid models. *IEEE Trans. Visualiz. Comput. Graph. 7*, 2, 173–192.

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

BROWN, A. D. 2005. Explicit compiler-based memory management for out-of-core applications. Ph.D. thesis, Carnegie Mellon University.

CHIANG, Y.-J. AND SILVA, C. T. 1997. I/o optimal isosurface extraction (extended abstract). *IEEE Visualiz.* 293–300.

CHOI, J., NOH, S. H., MIN, S. L., AND CHO, Y. 2000. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00)*. ACM Press, New York, NY, 286–295.

CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2003. External memory management and simplification of huge meshes. *IEEE Trans. Visualiz. Comput. Graph. 9*, 4, 525–537.

CIGNONI, P., ROCCHINI, C., AND SCOPIGNO, R. 1996. Metro: Measuring error on simplified surfaces. Tech. rep., Centre National de la Recherche Scientifique Paris, France.

COX, M. AND ELLSWORTH, D. 1997. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th Conference on Visualization (VIS'97)*. IEEE Computer Society Press, Los Alamitos, CA, 235–ff.

DESBRUN, M., MEYER, M., SCHRODER, P., AND BARR, A. H. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'99)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, 317–324.

DONGARRA, J., LUMSDAINE, A., NIU, X., POZO, R., AND REMINGTON, K. 1994. A sparse matrix library in C++ for high performance architectures. 214–218. http://citeseer.1st.psu.edu/dongarra94sparse.html.

ECKSTEIN, I., DESBRUN, M., AND KUO, C. J. 2006. Compression of time varying isosurfaces. In *Proceedings of Graphics Interface*.

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.

ENRIGHT, D., MARSCHNER, S., AND FEDKIW, R. 2002. Animation and rendering of complex water surfaces. In *Proceedings of SIGGRAPH*. Computer Graphics Proceedings, Annual Conference Series. ACM, 736–744.

FOSTER, N. AND FEDKIW, R. 2001. Practical animation of liquids. In *Proceedings of ACM SIGGRAPH*. Annual Conference Series. 23–30.

GEIGER, W., LEO, M., RASMUSSEN, N., LOSASSO, F., AND FEDKIW, R. 2006. So real it'll make you wet. In *Proceedings of the SIGGRAPH Conference on Sketches & Applications*. ACM.

GLASS, G. AND CAO, P. 1997. Adaptive page replacement based on memory reference behavior. In *SIGMETRICS '97: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 115–126.

GOBBETTI, E. AND MARTON, F. 2005. Far voxels: A multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph. 24*, 3, 878–885.

HESTENES, M. R. AND STIEFEL, E. L. 1952. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards Sect. 5*, 49, 409–436.

HIEBER, S. E. AND KOUMOUTSAKOS, P. 2005. A lagrangian particle level set method. *J. Comput. Phys. 210*, 1, 342–367.

HOUSTON, B., NIELSEN, M., BATTY, C., NILSSON, O., AND MUSETH, K. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph. 25*, 1, 1–24.

HOUSTON, B., WIEBE, M., AND BATTY, C. 2004. RLE sparse level sets. In *Proceedings of the SIGGRAPH Conference on Sketches and Applications*. ACM.

HY TRAC, U.-L. P. 2006. Out-of-core hydrodynamic simulations for cosmological applications. *New Astronomy*.

IBARRIA, L., LINDSTROM, P., ROSSIGNAC, J., AND SZYMCZAK, A. 2003. Out-of-core compression and decompression of large n-dimensional scalar fields. *Comput. Graph. For. 22*, 3, 343–348.

IRVING, G., GUENDELMAN, E., LOSASSO, F., AND FEDKIW, R. 2006. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Trans. Graph. 25*, 3, 805–811.

ISENBURG, M. 2004. Compression and streaming of polygon meshes. PhD thesis, University of North Carolina, In *Graphics Conference Proceedings*. 284–293.

ISENBURG, M. AND ALLIEZ, P. 2002. Compressing hexahedral volume meshes.

ISENBURG, M. AND GUMHOLD, S. 2003. Out-of-core compression for gigantic polygon meshes. *ACM Trans. Graph. 22*, 3, 935–942.

ISENBURG, M., LINDSTROM, P., AND SNOEYINK, J. 2004. Lossless compression of floating-point geometry. *CAD'3D*.

ISENBURG, M., LINDSTROM, P., AND SNOEYINK, J. 2005. Streaming compression of triangle meshes. In *Symposium on Geometry Processing*. 111–118.

JIANG, S. AND ZHANG, X. 2002. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*. 31–42.

JOHNSON, T. AND SHASHA, D. 1994. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 439–450.

JONES, M. W. 2004. Distance field compression. In *Proceedings of International Conference on Computer Graphics Visulization and Computer Vision. (WSCG)* 199–204.

JU, T., LOSASSO, F., SCHAEFER, S., AND WARREN, J. 2002. Dual contouring of hermite data. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'02)*. ACM, 339–346.

KÄLBERER, F., POLTHIER, K., REITEBUCH, U., AND WARDETZKY, M. 2005. Free lence—coding with free valences. In *Proceedings of Eurographics*.

KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C.-S. 2000. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. 119–134.

KINCAID, D. AND CHENEY, W. 1991. *Numerical analysis: mathematics of scientific computing*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA.

KOBBELT, L. P., BOTSCH, M., SCHWANECKE, U., AND SEIDEL, H.-P. 2001. Feature sensitive surface extraction from volume data. In *Proceedings of ACM SIGGRAPH*. Annual Conference Series. 15–22.

KRISHNAMOORTHY, S., BAUMGARTNER, G., LAM, C.-C., NIEPLOCHA, J., AND SADAYAPPAN, P. 2004. Efficient layout transformation for disk-based multidimensional arrays. In *Proceedings of International Conference in Performance Computing (HiPC)*. 386–398.

LANEY, D. E., BERTRAM, M., DUCHAINEAU, M. A., AND MAX, N. 2002. Multiresolution distance volumes for progressive surface compression. In *Proceedings of International Conference on 3D Data Processing Visualization and Transmission (3DPVT)*. 470–479.

LEE, D., CHOI, J., KIM, J.-H., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. 1999. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*. ACM. 134–143.

LEE, H., DESBRUN, M., AND SCHRODER, P. 2003. Progressive encoding of complex isosurfaces. *ACM Trans. Graph. 22*, 3, 471–476.

LEFOHN, A. E., KNISS, J., HANSEN, C., AND WHITAKER, R. 2003. Interactive deformation and visualization of level set surfaces using graphics hardware. In *IEEE Visual.* 75–82.

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

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'00)*. ACM. 131–144.

LI, M. AND VITANYI, P. 1997. *An Introduction to Kolmogorov Complexity and Its Applications* 2nd Ed. Springer-Verlag.

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

LOPEZ, J. C., O'HALLARON, D. R., AND TU, T. 2004. Big wins with small application-aware caches. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 20.

LORENSON, W. AND CLINE, H. 1982. Marching Cubes: A high resolution 3D surface construction algorithm. In *Proceedings of SIGGRAPH (Computer Graphics) 21*, 4, 163–169.

LOSASSO, F., FEDKIW, R., AND OSHER, S. 2005. Spatially adaptive techniques for level set methods and incompressible flow. *Comput. Fluids*. To appear.

LOSASSO, F., GIBOU, F., AND FEDKIW, R. 2004. Simulating water and smoke with an octree data structure. *ACM Trans. Grap. 23*, 3 (Aug.).

MASCARENHAS, A., ISENBURG, M., PASCUCCI, V., AND SNOEYINK, J. 2004. Encoding volumetric grids for streaming isosurface extraction. In *Proceedings of International Conference on 3D Data Processing Visulization and Transmission (3DPVT)*, 665–672.

MATIAS, Y., SEGAL, E., AND VITTER, J. S. 2000. Efficient bundle sorting. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*. Society for Industrial and Applied Mathematics, Philadelphia, PA. 839–848.

MAUCH, S. 2003. Efficient algorithms for solving static Hamilton-Jacobi equations. Ph.D. thesis, California Institute of Technology.

MEGIDDO, N. AND MODHA, D. S. 2003. Arc: *A self-tuning, low overhead replacement cache.* In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association, Berkeley, CA. 115–130.

MILNE, R. B. 1995. An adaptive level-set method. Ph.D. thesis, University of California, Berkeley.

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

MOFFAT, A., NEAL, R. M., AND WITTEN, I. H. 1998. Arithmetic coding revisited. *ACM Trans. Inf. Syst. 16*, 3, 256–294.

MUSETH, K., BREEN, D., WHITAKER, R., AND BARR, A. 2002. Level set surface editing operators. *ACM Trans. on Grap. 21*, 3 (July), 330–338.

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

NGUYEN, D. Q., FEDKIW, R., AND JENSEN, H. W. 2002. Physically based modeling and animation of fire. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'02)*. ACM Press, New York, NY, USA, 721–728.

NGUYEN, K. G. AND SAUPE, D. 2001. Rapid high quality compression of volume data for visualization. *Comput. Graph. For. 20*, 3.

NIELSEN, M. B. AND MUSETH, K. 2004a. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *Linköping Electronic Articles in Computer and Information Science 9*, 001.

NIELSEN, M. B. AND MUSETH, K. 2004b. An optimized, grid independent, narrow band data structure for high resolution level sets. In *Proceedings of SIGRAD'04*.

NIELSEN, M. B. AND MUSETH, K. 2006. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *J. Scient. Comput. 26*, 3, 261–299.

O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. 1993. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, P. Buneman and S. Jajodia, Eds. ACM, 297–306.

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

OSHER, S. AND SETHIAN, J. A. 1988. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys. 79*, 12–49.

PAJAROLA, R. 2005. Stream-processing points. in *IEEE Visualiz*. 31.

PASCUCCI, V. AND FRANK, R. J. 2001. Global static indexing for real-time exploration of very large regular grids. In *Proceedings of the ACM/IEEE conference on Supercomputing (Supercomputing'01)*. ACM, 2–2.

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. ACM. 79–95.

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.

ROBINSON, J. T. AND DEVARAKONDA, M. V. 1990. Data cache management using frequency-based replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*. ACM, 134–142.

SALMON, J. K. AND WARREN, M. S. 1997. Parallel, out-of-core methods for n-body simulation. In *Proceedings of Conference on Parallel Processing for Scientific Computing (PPSC)*.

SALOMON, D. 2007. *Data Compression: The Complete Reference*. With contributions by Giovanni Motta and David Bryant, Springer-Verlag..

SCHLOSSER, S. W., SCHINDLER, J., PAPADOMANOLAKIS, S., SHAO, M., AILAMAKI, A., FALOUTSOS, C., AND GANGER, G. R. 2005. On multidimensional data and modern disks. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*.

SEAMONS, K. E. AND WINSLETT, M. 1996. Multidimensional array i/o in panda 1.0. *J. Supercomput. 10*, 2, 191–211.

SETHIAN, J. A. 1999. *Level Set Methods and Fast Marching Methods* 2nd Ed. Cambridge University Press, Cambridge, UK.

SILVA, C., CHIANG, Y., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics.

SMARAGDAKIS, Y., KAPLAN, S., AND WILSON, P. 1999. Eelru: simple and effective adaptive page replacement. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*. ACM, NY, 122–133.

STAM, J. 1999. Stable fluids. In *Proceedings of SIGGRAPH'99*. Annual Conference Series. 121–128.

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

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*, 1, 81–124.

TAKAHASHI, T., FUJII, H., KUNIMATSU, A., KAZUHIRO HIWADA, T. S., TANAKA, K., AND UEKI, H. 2003. Realistic animation of fluid with splashes and foam. *Comput. Graph. For. 22*, 3, 391–401.

TANENBAUM, A. S. 1992. *Modern Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ.

TANG, H.-Z., TANG, T., AND ZHANG, P. 2003. An adaptive mesh redistribution method for nonlinear hamilton–jacobi equations in two-and three-dimensions. *J. Comput. Phys. 188*, 2, 543–572.

TAUBIN, G. 2002. Blic: Bi-level isosurface compression . In *IEEE Visualiz*.

TOLEDO, S. 1995. Quantitative performance modeling of scientific computations and creating locality in numerical algorithms. PhD thesis, MIT.

TOLEDO, S. 1999. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms*. American Mathamatical Society, 161–179.

TOUMA, C. AND GOTSMAN, C. 1998. Triangle mesh compression. In *Proceedings of Conference (Graphics Interface)*. 26–34.

TRAC, H. AND PEN, U.-L. 2006. Out of core hydrodynamic Simulations for Cosmological applications. *New Astronomy 11*, 4, 273–286.

TU, T., O'HALLARON, R., AND LOPEZ, C. 2004. Etree: A database-oriented method for generating large octree meshes. *Eng. Comput. 20*, 2, 117–128.

VITTER, J. S. 2001. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv. 33*, 2, 209–271.

VITTER, J. S. AND SHRIVER, E. A. M. 1994. Algorithms for parallel memory i: Two-level memories. *Algorithmica 12*, 2/3, 110–147.

WHITAKER, R. T. 1998. A level-set approach to 3d reconstruction from range data. *Int. J. Comput. Vision 29*, 3, 203–231.

XU, J. AND ZHAO, H. 2003. An eulerian formulation for solving partial differential equations along a moving interface. *J. Scient. Comput. 19*, 1–3, 573–594.

YANG, C.-K. AND CHIUEH, T.-C. 2006. Integration of volume decompression and out-of-core iso-surface extraction from irregular volume data. *Vis. Comput. 22*, 4, 249–265.

ZHAO, H.-K., OSHER, S., AND FEDKIW, R. 2001. Fast surface reconstruction using the level set method. In *Proceedings of the IEEE Workshop on Variational and Level Set Methods (VLSM'01)*. IEEE Computer Society, 194.

ZHOU, Y., CHEN, Z., AND LI, K. 2004. Second-level buffer cache management. *IEEE Trans. Parall. Distrib. Syst. 15*, 6, 505–519.

Received June 2006; revised April 2007; accepted August 2007