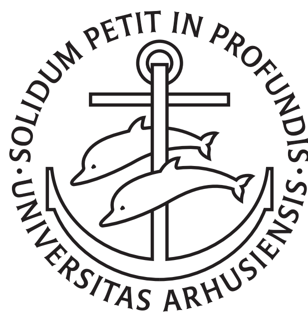
Efficient and High Resolution Level Set Simulations

- Data Structures, Algorithms and Applications

Michael Bang Nielsen

PhD Dissertation



Department of Computer Science
University of Aarhus
Denmark

Efficient and High Resolution Level Set Simulations - Data Structures, Algorithms and Applications

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Michael Bang Nielsen
(This edition contains the revisions suggested by the PhD committee)

Abstract

Level sets are dynamic implicit surfaces originally proposed for front propagation in computational physics by Osher and Sethian [113]. Due to the many advantages offered, including the ability to describe arbitrary topological changes, level sets have been applied not only in physics, but also in computer graphics and computer vision. In computer graphics level sets have been employed as the underlying surface representation in simulations of water, geometric modeling, shape metamorphosis and many other applications. Among the most resource-demanding effects in today's feature films is the simulation of water. Due in part to its advantages in representing this phenomenon, the level set method has already found wide spread use in the visual effects industry. As a result, existing level set technology is constantly being pushed to its limits as the demand for larger and more detailed simulations becomes ubiquitous. In particular three disadvantages hamper the level set method as originally proposed: Computational inefficiency, storage inefficiency and the confinement of deformations to a static predefined computational domain. Previous works have improved on these issues, but storage efficient algorithms tend to lower the computational efficiency and computationally efficient algorithms tend to increase the storage requirements. The research presented in this dissertation addresses these limitations. In particular the contributions fall into three categories: *Level Set Representations and Algorithms*, *Conversion* and enabled *Level Set Applications*.

Level Set Representations and Algorithms: This dissertation presents the Dynamic Tubular Grid (DT-Grid), a data structure for high resolution level set simulations not confined to a static predefined domain (*i.e.* level set deformations are *out-of-the-box*). The time- and storage-complexities of the DT-Grid scale with the number of grid points in the narrow band as opposed to the embedding volume, and performance evaluations show that DT-Grid is in general faster and requires less memory than previous approaches, including octrees, narrow band methods and a concurrently developed RLE data structure. Additionally the DT-Grid can take advantage of existing numerical schemes and generalizes to any number of dimensions. This dissertation also presents the Hierarchical Run-Length Encoded (H-RLE) grid which allows for greater versatility in the encoding of a level set than the DT-Grid. The H-RLE remains faster than and comparable to previous methods, but there is a slight degradation in performance when compared to the DT-Grid, due to the H-RLE's additional flexibility. Motivated by the fact that disk space in general offers much higher storage capacity and is two to three orders of magnitude cheaper than memory, an out-of-core and compression framework for level set simulations is finally proposed. The framework can be implemented as an extension to the DT-Grid and H-RLE and out-performs the original DT-Grid, when the DT-Grid must rely on virtual memory. Furthermore the framework can sustain a throughput of upto 65% on desktop computers with limited memory resources for simulations requiring several gigabytes of storage and can also be applied as an offline compressor. The out-of-core and compression framework allows for level sets of very high resolution and is to the best of our knowledge the first work

that applies out-of-core and compression methods to online level set simulation.

Conversion: As the data structures presented allow for very high resolution level sets, two algorithms are proposed for converting the most widely used surface exchange format, polygonal meshes, into level sets. The first method works in-core and allows for higher resolutions than previous conversion methods since both its storage and time complexity are linear in the number of faces of the polygonal mesh and the grid points in the level set. The second conversion algorithm retains the time- and storage-complexities of the in-core method, but in addition works out-of-core. In particular this method poses only the restriction that the size of the input mesh and the size of the output level set must be smaller than the available disk space.

Level Set Applications: The data structures and algorithms presented in this dissertation *enable* many applications of high resolution level sets. This dissertation contributes with several applications including high resolution and out-of-the-box shape deformations as well as out-of-core shape metamorphosis. In addition several applications using the data structure and algorithm implementations developed as part of this dissertation are briefly reviewed. This includes fluid simulation where the DT-Grid and H-RLE can be utilized for representing the fluid surface, obstacles, particles, fluid pressure, velocities and so on. In doing so the computational and storage requirements of fluid simulation become proportional to the volume of the fluid as opposed to the volume of the enclosing bounding box. Several other applications are reviewed, including volume segmentation, the solution of PDEs on surfaces, geometric texturing, modeling and animation of snow as well as ray tracing.

The research presented in this dissertation has been adopted in visual effects production, where some of the data structures are in experimental use in two major companies, and in academia where other researchers have employed and leveraged on the techniques proposed here.

Preface

This dissertation describes most of the research in which I have been involved in the three year period from August 2003 to July 2006 while being employed as a PhD student at the Department of Computer Science, University of Aarhus, Denmark. Most of the time I have conducted my research at the University of Aarhus, but I have had two long-term and inspiring stays abroad. From September 2004 to March 2005 I visited my advisor Ken Museth at Linköping University, and from April 2005 to June 2005 I visited the academy award winning visual effects company Digital Domain in Venice, California. I have also enjoyed teaching at University of Aarhus. In particular I benefited from teaching with Ken Museth in the course on "Modeling and Animation with Level Sets" as well as in the course on "Simulating Smoke and Water in Computer Graphics" where I was the main lecturer.

During the first six months I was involved in writing two papers on work that I had contributed to prior to my PhD studies. The first paper on inverse rendering under uncontrolled illumination [101] was co-authored by Anders Brodersen and accepted for publication at the WSCG 2004 conference, where we jointly presented the paper. This paper described results from our master's thesis from December 2002 on inverse rendering [100]. The second paper on mobile augmented reality based on feature tracking techniques [102], was co-authored by Gunnar Kramp and Kaj Grønbæk, and accepted at ICCS 2004, where I presented the paper.

My PhD application and initial PhD plan outlined several possible projects for computer graphics, primarily with emphasis on inverse rendering and level set methods. While inverse rendering remains an interesting field, my recent research has focused only on level set methods. By choosing this direction of research I have also been fortunate to become more involved in the interesting work done in the graphics group led by my advisor Ken Museth.

From August 2003 to July 2004 most of my time was dedicated to implementing a state-of-the-art level set software suite, and designing and implementing the Dynamic Tubular Grid (DT-Grid). My implementations now form part of the Graphics Group software Library, *GGL*. The DT-Grid research was published both as a technical sketch at SIGRAD 2004 [104], where I presented it, as a technical report at Linköping University [103] and as a paper in the Journal of Scientific Computing [105]. These publications were all co-authored by my advisor Ken Museth. My work on the DT-Grid was based on the ubiquitous need for a storage and computationally efficient data structure for level sets initially recognized by Ken Museth. Ken Museth was a major contributor in the problem-definition phase of this project and his great overview of the level set field to some extent guided my research. We had several inspiring discussions along the way in which Ken Museth provided advice and ideas. During the writing phase from April 2004 to early November 2004, Ken Museth's expertise played a key role in forming the final shape of our joint papers.

From November 2004 to January 2005 I worked together with Ola Nilsson and Ken Museth from Linköping University and Ben Houston and Christopher Batty from the R&D group at

the visual effects company Frantic Films. The R&D group at Frantic Films had worked on a level set data structure for computer graphics concurrently with our work on the DT-Grid and presented their method in a technical sketch at SIGGRAPH 2004 [50]. At SIGGRAPH 2004 Ken Museth also summarized the main features of our DT-Grid in the course on level sets and PDE methods for computer graphics [12]. In our collaboration we combined the DT-Grid with part of the work done by the R&D group at Frantic Films to create the Hierarchical RLE Grid (H-RLE). The work was accepted with major revisions by the SIGGRAPH 2005 committee and published in the ACM Transactions on Graphics [48] . It also appeared as a technical sketch at SIGGRAPH 2005 [49] which was presented by Ben Houston and I. My work focused on developing the actual data structure and algorithms on which I worked in close collaboration with Ben Houston. I also developed a fast polygon to level set conversion method, was the main responsible for the benchmarks and was involved in running some of the shape deformations. My co-authors Ben Houston, Christopher Batty, Ola Nilsson and Ken Museth contributed with the remaining parts of the work including ray tracing, fluid simulation, robust mesh to level set conversion and collision detection. The final state of our joint paper benefited from the large number of authors and we all provided critical perusals and exchanged ideas along the way. Ken Museth played a key role in the problem-definition, idea-generation and writing phases. His overview of and expertise in the field to a large extent guided us and significantly improved the paper.

From May 2005 to early August 2005 I worked on a robust conversion method from polygonal meshes with holes and self-intersections to high resolution level sets. This project was shelved due to the start-up of another project, and I did not find time to complete it prior to the conclusion of my PhD studies. Hence this is still on-going work and I plan to report on this in a future paper. Both my advisor Ken Museth as well as Doug Roble and Nafees Bin Zafar at Digital Domain provided helpful discussions in the initial phases of this project.

From September 2005 to June 2006 I worked together with Ola Nilsson, Andreas Söderström and Ken Museth from Linköping University on compressed and out-of-core (external memory) level set methods. This research was submitted to ACM Transactions on Graphics in July 2006 [106] and also appeared as a technical sketch at SIGGRAPH 2006 [107] jointly presented by Ola Nilsson, Andreas Söderström and I. My work focused on designing, implementing and benchmarking the compression and out-of-core level set framework as well as the out-of-core mesh to level set conversion algorithm and the out-of-core shape metamorphosis application. During the course of this project I had many inspiring discussions with my advisor Ken Museth as well as with my co-authors Ola Nilsson and Andreas Söderström. In particular Ken Museth and Ola Nilsson provided ideas and performed initial tests that served as an inspiration for the final compression methods. Whereas I focused on the main framework, Ola Nilsson, Andreas Söderström and Ken Museth contributed with most of the challenging applications and extensions including fluid simulation, PDEs on manifolds, out-of-core linear algebra and an out-of-core particle level set method. The problem-definition phase and final state of the paper largely benefited from Ken Museth's overview and expertise in the level set field, and all co-authors provided critical comments and corrections throughout the paper.

The papers mentioned above as well as additional videos are available on the CD-ROM accompanying this dissertation.

First and foremost I would like to acknowledge my advisor Ken Museth. He introduced me to the exciting research field of level set methods and his enthusiasm, support and hard work

has been a constant source of motivation for me. Many hours of his time have been devoted to answering my questions and I have truly enjoyed the privilege of expert advice. I would also like to emphasize that without his ideas, guidance and expertise the research presented in this dissertation would not have been possible. Furthermore I am grateful to him for initiating the contact to Digital Domain and for helping with all the formalities. Secondly I acknowledge my advisor Kaj Grønbæk at University of Aarhus, Denmark. I am truly grateful to him for encouraging me to pursue a PhD study in computer graphics and for always being supportive and showing great interest in my work. Additionally I thank him for providing advice on research and scientific writing in general. I am also grateful to my collaborators in the graphics group, Ola Nilsson and Andreas Söderström, for many helpful discussions, for being a source of inspiration and for allowing me to include images in this dissertation from their work using my data structure and algorithm implementations. I also thank my collaborators Ben Houston and Christopher Batty for many discussions, and furthermore I thank Anders Brodersen, Tommy Hinks and Gunnar Johansson for allowing me to include images from their applications of DT-Grid in this dissertation. During my two-month stay at Digital Domain I had the privilege of working with Doug Roble and Nafees Bin Zafar. This stay gave me some insight into the exciting world of visual effects and knowledge of how computer graphics is applied in practice. What I learned will always play a role in motivating the applicability of my research. I am particularly grateful to Doug Roble and Nafees Bin Zafar for always taking the time to answer my many questions on computer graphics and film-making in general. I also wish to thank Tony Chan, Stanley Osher and Luminita Vese for giving me the opportunity to speak at a UCLA Image Processing Seminar in August 2005 and I am grateful to Doug Roble for initiating the contact. Finally I wish to thank Ole Østerby at University of Aarhus who has provided answers to my many questions on numerical analysis and partial differential equations.

More concretely I would like to thank Ola Nilsson for help with the renderings in figures 1.4, 1.5, 5.3, 5.8, 5.9, 14.1, 16.1, for providing figures 1.6, 16.2, 16.6, 16.7 and for help with figures 6.1, 10.1, 10.3. I also thank him for allowing me to use his ray tracing plugin for PBRT. Additionally I thank Andreas Söderström for providing figures 1.6, 16.4, 16.5, Anders Brodersen for providing figure 16.8, Tommy Hinks for providing figure 16.9, Gunnar Johansson for providing figure 16.10 and Ben Houston for help with figures 6.1 and 6.2. I also thank Ken Museth for help with figure 6.1 and for allowing me to use his Marching Cubes implementation. Finally I would like to thank Ken Museth, Ola Nilsson, Andreas Söderström, Kaj Grønbæk, Anders Brodersen and Lars Bo Kristensen for commenting on drafts of this dissertation.

The research presented in this dissertation was funded in part by Aarhus University and Center for Interactive Spaces under ISIS Katrinebjerg, Aarhus.

Michael Bang Nielsen,
Århus, August 2006.

Contents

Abstract **v**

Preface **vii**

1 Introduction **1**

1.1 The Level Set Pipeline 5
1.2 Contributions 6
1.3 Outline 10

I Level Set Methods **15**

2 Introduction to Level Set Methods **17**

2.1 Implicit Surfaces 18
2.2 The Theory of the Level Set Method 22
2.3 The Numerics of the Level Set Method 25
2.3.1 Approximation of Derivatives with Finite Differences 25
2.3.2 Numerical Stability 27
2.3.3 Numerical Solution of the Level Set Equations 28
2.3.4 Vanishing Viscosity Solutions and Numerical Dissipation 30
2.4 The Narrow Band Level Set Method 31
2.5 Advantages and Disadvantages of Level Sets 34
2.6 Summary 35

3 Overview of Level Set Methods, Algorithms and Grid Representations **37**

3.1 Narrow Band Level Set Methods 37
3.2 Octree-Based Level Set Methods 38
3.3 Sparse Non-Tree-Based Level Set Representations 39
3.4 Adaptive Level Set Methods 40
3.5 The Particle Level Set Method 41
3.6 Moving and Resizing Grids 42
3.7 GPU Based Level Sets 43
3.8 Summary 43

II Data Structures and Algorithms for High Resolution Level Set Simulations 45

4 Introduction 47

5 The DT-Grid - High Resolution Level Set Simulations 49
5.1 Contributions 49
5.2 DT-Grid Data Structure 51
5.2.1 Definition of the DT-Grid 51
5.3 DT-Grid Algorithms 57
5.3.1 Push - Inserting Grid Points in Constant Time 58
5.3.2 Logarithmic Time Random Access 59
5.3.3 Logarithmic and Constant Time Neighbor Access 61
5.3.4 Constant Time Sequential Access Using Iterators 62
5.3.5 Constant Time Stencil Access Using Iterators 63
5.3.6 Dilating the Tubular Grid in Linear Time 65
5.3.7 Rebuilding the Tubular Grid in Linear Time 70
5.3.8 CSG Operations in Linear Time 71
5.4 Augmenting the DT-Grid with Auxiliary Data 72
5.5 Open Level Sets 72
5.6 Examples 75
5.6.1 An Out-Of-The-Box Simulation 75
5.6.2 A High Resolution Simulation - The Enright Test 76
5.7 Summary 78

6 The H-RLE Grid - Flexible High Resolution Level Set Simulations 79
6.1 Contributions 79
6.2 H-RLE Data Structure 81
6.2.1 Taxonomy of the RLE Data Structure 81
6.2.2 The Hierarchical Data Structure 82
6.3 H-RLE Algorithms 85
6.3.1 Constant Time Push 85
6.3.2 Logarithmic Time Random Access 87
6.3.3 Logarithmic Time Neighbor Access 87
6.3.4 Constant Time Sequential Access 88
6.3.5 Rebuilding the Hierarchical RLE Level Set in Linear Time 88
6.4 Versatility of the H-RLE 89
6.5 Summary 91

7 Evaluation and Discussion of DT-Grid and H-RLE Grid 93
7.1 Evaluated Data Structures and Methodology 93
7.2 Evaluation of Level Set Simulation Performance 96
7.3 Evaluation of Random Access Performance 106
7.3.1 Evaluation of Fast Marching 106
7.3.2 Evaluation of Level Set Ray Tracing 107
7.4 Evaluation of Storage Requirements for High Resolution Level Sets 109
7.5 Discussion 112

7.6 Summary 113

III Algorithms for Compression and Out-Of-Core Level Set Methods 115

8 Introduction 117

9 Related Work on Compression and Out-Of-Core Methods 119
9.0.1 Compression Methods 119
9.0.2 Out-Of-Core Methods 120

10 The Out-Of-Core and Compressed DT-Grid - External Memory Level Set Methods 123
10.1 Contributions 125
10.2 Out-Of-Core and Compression Level Set Framework 125
10.2.1 Terminology 126
10.3 Out-Of-Core Data Management 127
10.3.1 Page-Replacement 128
10.3.2 Prefetching 129
10.4 Compression Algorithms 130
10.4.1 Compressing The Topology 130
10.4.2 Compressing The Values 133
10.5 Example - An Out-Of-Core Shape Metamorphosis 135
10.6 Summary 136

11 Evaluation and Discussion of the Compressed and Out-Of-Core DT-Grid 137
11.1 Page-Replacement and Prefetching 137
11.2 Online Out-of-Core and Compression Framework 139
11.3 Offline Compression 144
11.4 Discussion 148
11.5 Summary 149

IV Methods for Converting Polygonal Meshes into High Resolution Level Set Surfaces 151

12 Introduction 153

13 Previous Methods for Converting Polygonal Meshes 155

14 Converting Polygonal Meshes 157
14.1 Contributions 157
14.2 In-Core Scan Conversion 158
14.3 Out-Of-Core Scan Conversion 159
14.4 Evaluation and Discussion 162
14.5 Summary 166

V Applications **167**

15 Applications **169**
15.1 High Resolution Surface Deformations 170
15.1.1 Bunny Enright Test 170
15.1.2 Shape Metamorphosis 170
15.2 Fluid Simulation 171
15.3 Ray Tracing 175
15.4 PDEs on Manifolds 175
15.5 Geometric Texturing 176
15.6 Snow Modeling and Simulation 177
15.7 Volume Segmentation 178
15.8 Summary 179

VI Conclusions and Future Work **181**

16 Future Work **183**

17 Conclusions **185**

Bibliography **187**

Chapter 1

Introduction

The main topic of this dissertation is the development of novel data structures and algorithms that decrease the storage requirements and increase the computational efficiency of the widely used level set method. As this research enables level set surfaces of high resolution and detail to be represented and simulated, methods are also proposed for converting polygonal meshes, the prevalent surface representation, into high resolution level sets.

The degree of realism achievable in today's feature films is astonishing. The techniques in computer graphics and visual effects are now at a point in evolution where practically everything that can be imagined can be put to life on the big screen. This has in recent years resulted in a large number of great epic tales such as "Lord of the Rings" and "Chronicles of Narnia" as well as catastrophe-movies like "The Day after Tomorrow". Even though the images of our imagination can now to a large extent be visualized it does require the skills of very talented and hard working artists exploiting existing technology to its maximum. Hence, researchers in computer graphics still face a fair amount of challenges. The basic research areas within the field are well established, and each area holds unique problems. Some techniques are too time- and/or memory-demanding to be compatible with the ever decreasing post production time and limited resources available. Some techniques only work in laboratory settings and others still are too hard to control in order to achieve the desired artistic expression. This means that graphics researchers in many cases must now focus on improving the efficiency and lowering the resource requirements of existing methods, thereby increasing the scale and detail achievable. In many cases this entails taking advanced and entirely new mathematical and algorithmic approaches to the problems posed.

The request for more efficiency also arises frequently in the rapidly evolving gaming industry that continues to challenge contemporary techniques as the demand for more realistic games becomes ubiquitous. Even though the gap between real time and offline graphics techniques is decreasing, efficient and less resource demanding effects are more likely to become feasible for game-console implementation in the near future.

Part of the success of computer graphics in recent challenging feature films can be attributed to the use of physically based rendering and simulation. The level set method [113], originally developed for front propagation in computational physics, is an example of this. Technically speaking, level sets are dynamic implicit surfaces controlled by a set of partial differential equations that describe the motion or dynamics mathematically. These partial differential equations, or PDEs, are in general referred to as the level set equations. As opposed to many traditional surface representations level sets pose no restrictions on the degree and complexity of topologi-

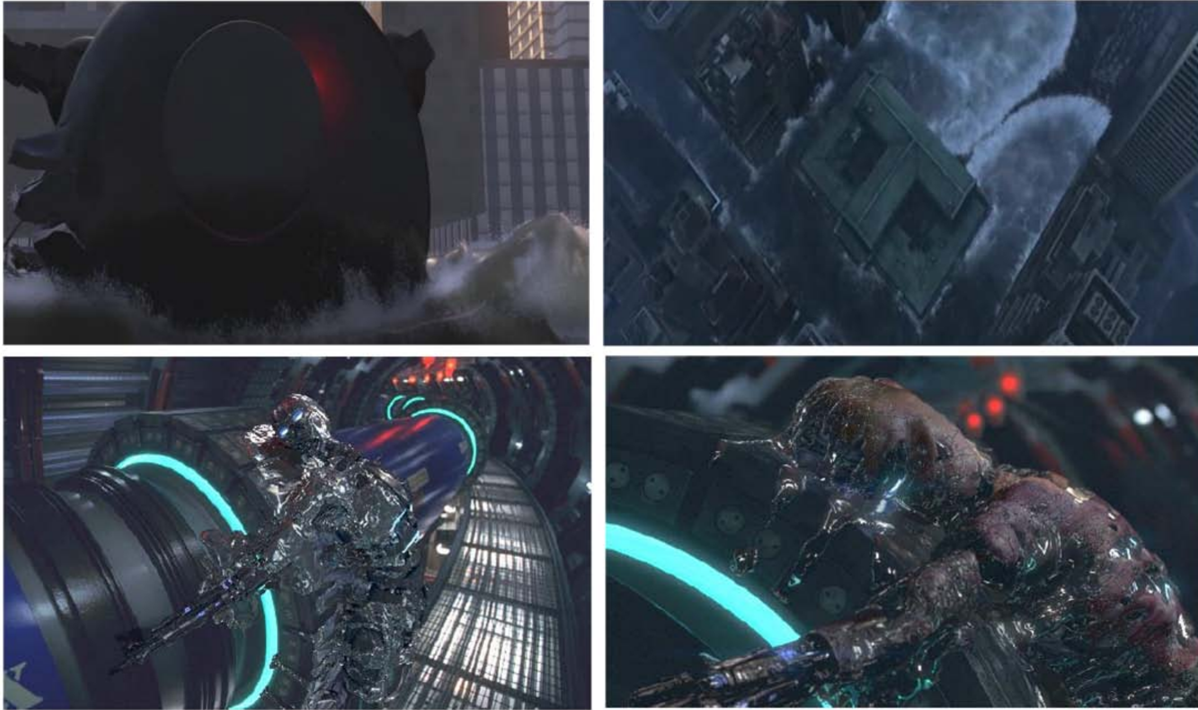


Figure 1.1: Screen shots showing level set based fluid simulations in recent feature films: Top left: A robot tips over and clashes with the sea. From *The Incredibles*. ©Pixar. Top right: The New York Public Library being flooded in *The Day after Tomorrow*. ©Twentieth Century Fox. Bottom: The melting terminatrix in *Terminator 3*. ©Warner Bros. Pictures.

cal change and can by construction not self-intersect. Hence, being ideally suited for physically based simulation of dynamic surfaces such as free surface fluids [35, 37] the level set representation has achieved wide spread use not only in the area of computational fluid dynamics, but also in computer graphics and entertainment such as movie visual effects. Computational fluid dynamics and engineering applications of level set and fluid simulations rely on the numerical accuracy and explicit error control offered by these methods. The visual effects industry on the other hand is more concerned with the visual expression. In fact, visually convincing simulation of water is one of the most demanding effects of today's feature films. The success of computer based simulations of water for visual effects is attributable in part to its flexibility: A director can choose the appropriate view points and camera paths *after* the simulation has been performed in order to obtain the most suitable look. This is not possible with the alternatives that also break down in other cases: Miniature water often looks fake due to surface tension, and owing to the scope of the desired effects, large scale physical water is in many cases simply not an option.

Since a major utilization of the level set method in computer graphics has so far been water and other types of fluid simulations it is used as an example throughout this chapter. However, it is stressed that level sets have been applied successfully in many different and equally important areas such as for example geometric modeling [96], including shape metamorphosis [14]. Related fields like computer vision and visualization have also benefited from the level set method which for example has given rise to new methods in the area of volume segmentation [160] that *e.g.*

segments internal structures such as organs from scannings of animals and humans.

Figure 1.1 illustrates the use of level sets for simulating fluids in three recent feature films, “The Incredibles”, “The Day after Tomorrow” and “Terminator 3”. Other films that have exploited the advantages of level set methods include, but are not limited to, “Pirates of the Caribbean”, “Peter Pan”, “Shrek” and “Scooby Doo 2”.

(a) (b)

Figure 1.2: This figure shows two stages in the creation of a shot from “The Day after Tomorrow”. ©Twentieth Century Fox. (a) The raw low resolution output from the level set based fluid simulation. (b) The final composite with particle and texturing effects applied.

The ubiquitous use of the level set method clearly manifests its importance not only in computational fluid dynamics but also in computer graphics and visual effects. However, the unique advantages of level sets come at a price. In particular the original level set method [113] is inefficient in terms of memory consumption and is computationally expensive as well. Additionally, the simulation of fluid pressure and velocity needed when utilizing the level set method for fluid surfaces is hampered by the same limitations. Looking at the screen shots from “The Day after Tomorrow” present in figure 1.1, the foam, the ripples, the debris and the splashes as the waves collide with the buildings are all at a very high level of detail. Most of these details are however added as a post-process by artists utilizing sophisticated texturing and particle effects. The underlying level set simulation is in fact very coarse and provides only a basis for the artists to work with. Figure 1.2 shows a side-by-side comparison of another shot from “The Day after Tomorrow”. The image on the left shows the raw level set based fluid geometry as output by the simulation, and the image on the right shows the same geometry after post-production. The output from the simulation exhibits only a low degree of detail and exposes one of the limiting factors of level set simulations in production and research: Their low resolution. This property which results in lack of detail in contemporary level set simulations is to a large extent ascribable to the storage inefficiencies of existing techniques in use today.

The inefficiencies of the original level set method [113] arise in part because a level set surface, or *interface*, is in fact embedded in a volume and sampled on a dense uniform lattice known as a *grid*. Both storage and computational requirements of this grid scale as L^3 in three dimensions, where L is the side-length, or equivalently the number of grid points along each axial direction, of the grid. L^3 will generally be referred to as the resolution or dimensions of the grid. Obviously, the requirements should rather scale with the area of the surface itself. Similarly, when utilizing level sets for fluid simulation, the fluid velocity and pressure are typically represented on a grid.

While these quantities are inherently volumetric they are typically only required in the fluid interior. Hence ideally the storage and computational requirements imposed by these quantities should scale with the volume of the fluid as opposed to the volume of the enclosing grid.

Another inherent problem of the original level set method is that deformations are confined to stay within the boundaries of a static predefined grid. In practice it means that the extents of the grid in which the deformation is taking place must be determined prior to the execution of the simulation. Not only may these extents be difficult to estimate but they may also lead to an even higher memory usage if the deformation is taking part in a large region of space, since the entire grid must be allocated at once.

Based on the observation that only grid points in the vicinity of the level set surface are significant for computing its movement, a number of researchers have proposed so-called *narrow band methods* [1, 108, 120, 161] that successfully address the computational overhead of the original level set method. In particular these methods in general make the computational requirements scale with surface area by restricting the solution of the level set equations to a narrow band of grid points centered about the surface. Even though the narrow band methods do indeed increase the computational efficiency, they all incur additional storage overhead due to the introduction of auxiliary data structures that track the narrow band.

Even at relatively low resolutions, the storage requirements of a dense uniform grid are limiting. This has the consequence that simulation on higher resolution grids is rendered impossible. As an example, a grid of dimension 512^3 storing single precision floats in itself requires half a gigabyte of storage. Additionally, multiple grids, including scalar and velocity fields, are often needed in memory during simulation. This usually makes it practically infeasible to run a level set simulation on a grid of these dimensions even when given a computer with one gigabyte of physical memory. As a result, previous research has only employed level sets at relatively low resolutions. In [96] from 2002, for example, the largest model has effective resolution $356 \times 251 \times 161$. In comparison the largest model demonstrated in this dissertation has effective resolution $35000 \times 20000 \times 11500$ and more than 7 billion grid points in the narrow band.

The storage requirement overhead inherent in the original as well as the narrow band level set methods has been addressed by several authors suggesting the use of octree grids [31, 83, 84, 94, 139–142] instead of dense uniform grids. Although the idea of octree grids is attractive because this representation decreases the amount of storage necessary, a state-of-the-art octree implementation tends to perform worse than a state-of-the-art narrow band method in terms of computational efficiency [48].

The infeasibility of octree structures is also elaborated on by Bridson [15] who as an alternative introduces the notion of sparse block grids. The method of sparse blocks restricts computations to a narrow band, but the memory consumption may not scale with the size of the interface.

A more recent approach to decreasing the storage requirements of level sets is based on a Run-Length Encoding (RLE) of the grid regions outside the narrow band [50]. This method - developed concurrently with the work presented in this dissertation - does however not perform as well as techniques presented in this dissertation, both in terms of memory consumption and computational efficiency [48].

Altogether the issues identified above impose limitations on the scale and amount of detail achievable with existing methods. This is a problem since engineers, graphics researchers and artists in fact all desire to run simulations at higher resolutions: The engineers for improved numerical accuracy and the graphics researchers and artists for improved visual accuracy. Hence we have arrived at a predominant problem in the level set arena: Less storage intensive and more

computationally efficient data structures and algorithms are required in order to fully utilize the potential of the level set method. Due to the prevalence of level sets, methods of solution to this problem will be of immediate interest in a diversity of research areas.

The contributions presented in this dissertation address the limitations of existing level set methods revealed above. In particular this work takes a *computer scientific approach* to overcome the inefficiencies of contemporary level set methods: Instead of attempting to devise new level set methods incompatible with the existing numerical schemes building on years of extensive research efforts, this dissertation proposes novel data structures and algorithms for level sets that can take advantage of existing schemes. This allows us to employ techniques such as *finite differences*[1] known for their robustness, explicit error control and, if desired, high numerical accuracy. Another benefit of the computer scientific approach taken is that the new techniques proposed can be integrated into existing level set pipelines with minimal efforts since no changes are required to existing numerical level set schemes. All the new algorithms and data structures in this dissertation are evaluated extensively, both qualitatively and quantitatively, and compared to custom implementations of state-of-the-art alternatives, hence forming a strong basis for the conclusions drawn from our work. Although the details of these evaluations are nuanced, this dissertation does in particular present data structures and algorithms for narrow band level set simulations that outperform both octree grids, previous RLE grid and narrow band methods in terms of storage and computational efficiency. In addition *all* the data structures proposed in this dissertation can be used to also represent the volumetric fluid velocities and pressure hence making the storage requirements of these quantities scale with the volume of the fluid interior as opposed to the enclosing grid. Since the data structures presented allow for level set simulations at high resolution, this dissertation also considers and proposes new methods for converting polygonal mesh representations, which is the most common interchange format for 3D models today, into these high resolution level set representations.

1.1 The Level Set Pipeline

To view the contributions of this dissertation in the appropriate context, an overview of the work flow associated with a typical level set pipeline is provided in figure 1.3. The contributions of this dissertation fall into three distinct categories highlighted in color in figure 1.3: *Level set representations and algorithms*, *repair and conversion* and finally enabled *level set applications*.

Referring to figure 1.3, the terms *level set representation and algorithms* covers the data structure storing the level set surface as well as the algorithms used to access the data structure and deform the data stored. Typical examples of level set data structures are dense uniform grids, octree grids as well as the data structures proposed in this dissertation. The most widely employed volumetric embedding of a level set surface is a *signed distance field* since it offers numerical robustness and enables optimizations elaborated in chapter 3. Briefly explained the signed distance field represents a surface by storing at each point in the grid the distance to the surface multiplied by ± 1 depending on whether the grid point is inside or outside of the surface.

On the other hand, the prevalent representation of surfaces in computer graphics is the *boundary surface representation*, also called an explicit surface representation, which comes in the form of polygonal meshes, NURBS, subdivision surfaces and others. The *repair and conversion* convert boundary surface representations to level sets.Due to the nature of the repair and

[1]A finite difference is an approximation to a derivative. An elaborate explanation follows in chapter 2.

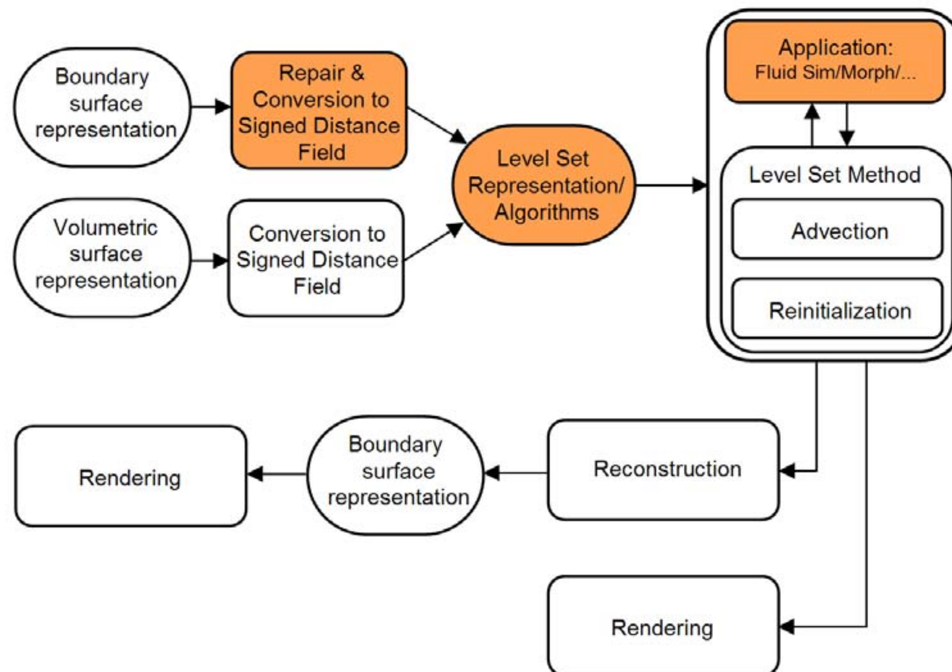


Figure 1.3: The level set pipeline. The areas into which the contributions of this dissertation fall are highlighted in orange.

conversion algorithms they are often referred to as *Scan conversion algorithms*. The *Volumetric surface representation* typically arises when 3D models are generated from CT or MRI scans. To be amenable for level set simulation a volumetric representation must first be converted into a signed distance field. Many techniques exist for this task, some of which will be covered in chapter 3. *Applications* such as fluid simulation and shape metamorphosis take as input one or several level sets and utilize an underlying *level set method* which deforms the surface in question. The level set method consists of two modules: An *advection* module that advects or propagates the surface, and a *reinitialization* module that reinitializes the level set representation to a signed distance field after a number of advection steps. Depending on what kind of level set method is employed, other steps may be performed as well. For visualization, the level set can be subject to *rendering* directly by ray tracing, or to conversion into a boundary surface representation using a *reconstruction* technique such as for example marching cubes [82]. Finally the boundary surface representation can be rendered with conventional scan line rasterization or ray tracing techniques. Note that the level set representations proposed in this dissertation are directly amenable for ray tracing at speeds comparable to existing methods using conventional ray leaping techniques [35, 48, 85], see chapter 15.

1.2 Contributions

I have been *the* major contributor to the design of all data structures and algorithms proposed in this dissertation, except for the H-RLE data structure (see below) where I was *a* major contributor together with Ben Houston. Furthermore *all* implementations of these data structures and algorithms evaluated and utilized in this dissertation have been implemented by me. My

advisor Ken Museth has been a major contributor in the problem-definition phases of all the contributions in this dissertation, and throughout the process he has provided advice, guidance and ideas. In the paper-writing phases his expertise and insight into the field have to a large extent formed the final state of our joint papers. I have also had fruitful discussions with my co-authors, in particular Ola Nilsson and Andreas Söderström. Consequently I will consistently be using the pronouns *we* and *our* when referring to the origins of the work presented here.

The novel data structures and algorithms presented in this dissertation have *enabled* several different applications of high resolution level sets. In particular I have been fortunate that other members of the graphics group have used my implementations of the data structures and algorithms in their applications, hence now allowing me to demonstrate the applicability of my work. Thus when presenting applications of my work I will in many cases be referring to and showing images from the work of other people.

The work presented in this dissertation is predominantly taken from the following papers on which I am a co-author:

[105] Michael B. Nielsen and Ken Museth 2006. Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets. *Journal of Scientific Computing 26*, 3, 261-299. (submitted November 2004; accepted January 2005).

[48] Ben Houston, Michael B. Nielsen, Christopher Batty, Ola Nilsson and Ken Museth 2006. Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation. *ACM Transactions on Graphics 25*, 1, 1-24. (submitted January 2005, accepted with major revisions by the SIGGRAPH paper committee April, 2005, accepted October, 2005).

[106] Michael B. Nielsen, Ola Nilsson, Andreas Söderström and Ken Museth 2006. Out-Of-Core and Compressed Level Set Methods. (submitted July 2006 and accepted November 2006 to ACM Transactions on Graphics).

This work has also appeared in slightly different form in the following technical reports and technical sketches:

[104] Michael B. Nielsen and Ken Museth 2004. An Optimized, Grid Independent, Narrow Band Data Structure for High Resolution Level Sets. *Proceedings of SIGRAD 2004*.

[103] Michael B. Nielsen and Ken Museth 2004. Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets. *Linköping Electronic Articles in Computer and Information Science*, 9, 1. (ISSN 1401-9841).

[49] Ben Houston, Michael B. Nielsen, Christopher Batty, Ola Nilsson and Ken Museth 2005. Gigantic Deformable Surfaces. *Proceedings of the SIGGRAPH 2005 Conference on Sketches & Applications.*

[107] Michael B. Nielsen, Ola Nilsson, Andreas Söderström and Ken Museth 2006. Virtually Infinite Resolution Deformable Surfaces. *Proceedings of the SIGGRAPH 2006 Conference on Sketches & Applications.*

Each chapter in this dissertation explicitly states a detailed description of its contributions. Below follows an overview of the contributions of the entire dissertation, divided into three distinct categories: *Level set representations and algorithms*, *repair and conversion* and enabled *level set applications*:

Level set representation and algorithms

- **The Dynamic Tubular Grid (DT-Grid)**: A data structure and algorithms for high resolution narrow band level set simulations. Both storage and computational requirements of the DT-Grid scale with the size of the surface as opposed to the volume of the embedding. Similar to the level set method, the DT-Grid generalizes to any number of dimensions. Existing numerical methods can be used with the DT-Grid, and a novel feature is that in contrast to all previous work the DT-Grid does not have a predetermined bounding box. In other words, a DT-Grid based level set can expand and contract freely. We name this feature *out-of-the-box*. Extensive tests have shown that in general the DT-Grid is faster and requires less memory than state-of-the-art implementations of octree grids, previous RLE methods and narrow band methods. Due to the features above, the DT-Grid allows for level set simulations at high resolutions. Figure 1.5 shows an example of a high resolution simulation (1024^3) using our data structure.
- **The Hierarchical Run Length Encoded Grid (H-RLE)**: A *flexible* data structure and algorithms for high resolution narrow band level set simulations. This work combines the DT-Grid with the run-length encoding of the Sparse RLE level set of Houston *et al.* [50]. Additionally the H-RLE adapts the DT-Grid algorithms to this storage format. Whereas the DT-Grid does not store any information on regions outside the narrow band, the H-RLE employs a Run-Length Encoding (RLE) of these regions. Due to its close relation to the DT-Grid data structure and algorithms, the H-RLE inherits many of the properties of the DT-Grid: Its memory and computational requirements scale with the size of the surface, it generalizes to any number of dimensions, it is compatible with existing numerical methods and it is out-of-the-box. In addition, the H-RLE offers versatility over the DT-Grid: Flexible encoding of regions outside the narrow band, adaptive encoding of the regions inside the narrow band and a decoupling of the numerical values from the data structure itself. However, for standard narrow band level set simulations, the DT-Grid remains more memory and computationally efficient than H-RLE but the H-RLE still performs relatively fast compared to other existing methods. Therefore we stress that the H-RLE and DT-Grid complement each other. For standard level set simulations the DT-Grid is preferable, whereas for versatile narrow band encodings, the H-RLE offers advantages.
- **Out-Of-Core and Compression Level Set Framework**: A generic framework that allows level set simulations based on the DT-Grid and H-RLE data structures to take advantage of statistical compression methods and external memory in the form of disk space[2]. In particular the contributions related to external memory are a near optimal page-replacement algorithm and a prefetching strategy that optimize sequential access with finite differences to out-of-core level sets. An out-of-core DT-Grid relying on the new page-replacement and prefetching algorithms out-performs the original DT-Grid relying solely

[2] Algorithms that utilize external memory (such as disks) as a means to overcome the limited physical memory during execution are usually referred to as *external memory* or *out-of-core* algorithms.

on OS paging and prefetching when level sets do not fit in memory. The contributions of the framework also include compression schemes for the DT-Grid that reduce both memory consumption online during simulation and offline as a tool for reducing disk storage and transmission bandwidth. The framework proposed is flexible in that the compression and out-of-core components can be arbitrarily combined or left out. Furthermore the only limitation in practice being the amount of available disk space. We stress that this framework complements the in-core versions of the DT-Grid and H-RLE: When the level sets and auxiliary data structures fit in memory, the DT-Grid and H-RLE are preferable, but for gigabyte sized level set simulations that do not fit in memory, this framework is faster. Figure 1.6 shows a fluid simulation run partially out-of-core using the proposed out-of-core framework.

Conversion:

- **Conversion of Consistent Meshes into High Resolution Level Sets**: A conversion method from *consistent* polygonal meshes[3] into high resolution DT-Grid/H-RLE representations. The method leverages on and performs comparable to [89] for low resolutions, but allows for much higher resolution level sets to be generated. In particular both time- and memory-consumption of our method are proportional to the number of faces of the polygonal mesh and the grid points in the generated level set. Figure 1.4 shows a high resolution level set generated with this method.
- **Conversion of Consistent Meshes into High Resolution Out-Of-Core Level Sets**: The in-core conversion method above extended to take advantage of external memory. The only limitation imposed is the amount of available disk space. For scan conversions that fit in memory this method is slower than the in-core algorithm, but when the in-core method must rely on OS virtual memory, the out-of-core conversion approach performs several times faster. Using this method we have created out-of-core DT-Grids with up to seven billion grid points in the narrow band and an effective resolution of $35000 \times 20000 \times 11500$.

Level Set Applications:

- The data structures and algorithms proposed in this dissertation *enable* a wide range of applications in computer graphics. In particular high resolution shape deformations, out-of-the-box simulations and out-of-core shape metamorphosis are demonstrated. Furthermore applications contributed by other members of the graphics group that use my data structure and algorithm implementations are briefly reviewed. This includes ray tracing, the solution of PDEs on manifolds, volume segmentation, geometric texturing, modeling and simulation of snow as well as fluid simulation. In particular *all* the data structures proposed in this dissertation can be used to represent volumetric fluid velocity and pressure hence also addressing and improving on the storage and computational requirements of fluid simulations.

The key implication of the contributions presented in this dissertation is that level set simulations can now be run at high resolutions without compromising computational efficiency compared to previous methods. Owing in large extent to the prevalence of the level set method,

[3] A consistent polygonal mesh is a closed manifold *i.e.* has no holes or self-intersections.

the new methods developed are useful in a wide range of fields including computational physics, computer vision and computer graphics. We concentrate on computer graphics and document several applications.

1.3 Outline

This dissertation consists of six parts: *Part I: Level Set Methods*, *Part II: Data Structures and Algorithms for High Resolution Level Set Simulations*, *Part III: Algorithms for Compression and Out-Of-Core Level Set Methods*, *Part IV: Methods for Converting Polygonal Meshes into Level Set Surfaces*, *Part V: Applications* and finally *Part VI: Conclusions and Future Work*. Each of these parts commences with an introduction (not included in the outline below). The content of the individual chapters is as follows.

Part I: Chapter 2 introduces implicit surfaces as well as the basic level set theory, numerical schemes and the narrow band level set methods. Chapter 3 provides an overview of previous and concurrent level set methods and grid representations relevant for the work in this dissertation.

Part II: Chapter 5 describes the DT-Grid and shows two examples illustrating its ability to represent high resolution and out-of-the-box level set simulations. Chapter 6 subsequently introduces the H-RLE grid and describes the differences between this and the DT-Grid representation. Chapter 7 evaluates the performance of the DT-Grid and the H-RLE and compares to existing octree, narrow band and RLE methods as well as provides a discussion of our methods.

Part III: Chapter 9 first reviews related work in the area of compression and out-of-core methods. Next chapter 10 introduces the compression components and the out-of-core techniques of our generic level set framework. Subsequently chapter 11 evaluates the compression and out-of-core framework as well as compares its performance to peak DT-Grid performance for level set simulations that fit in memory. A detailed discussion on the feasibility and limitations of the framework is also provided in this chapter.

Part IV: Chapter 13 describes previous work on converting polygonal meshes to level sets, and following that chapter 14 describes both the in-core and out-of-core converters for consistent meshes as well as evaluates their performance and discusses their pros and cons.

Part V: Having described the main contributions of this dissertation we switch to a presentation of some of the applications enabled by our research in chapter 15. This includes shape deformations, ray tracing, geometric texturing, segmentation, snow modeling, solving PDEs on manifolds and fluid simulations.

Part VI: Finally chapters 16 and 17 respectively describe future work and concludes this dissertation.

Figure 1.4: The Lucy statue represented as a level set in effective resolution $3000 \times 1726 \times 5144$. DT-Grid memory usage is 700 MB and H-RLE memory usage is 738 MB. The narrow band method of Peng [120] would require 128 GB (computed analytically). Rendering by Ola Nilsson based on my DT-Grid implementation and scan conversion. Lucy statue model courtesy of the Stanford Scanning Repository.

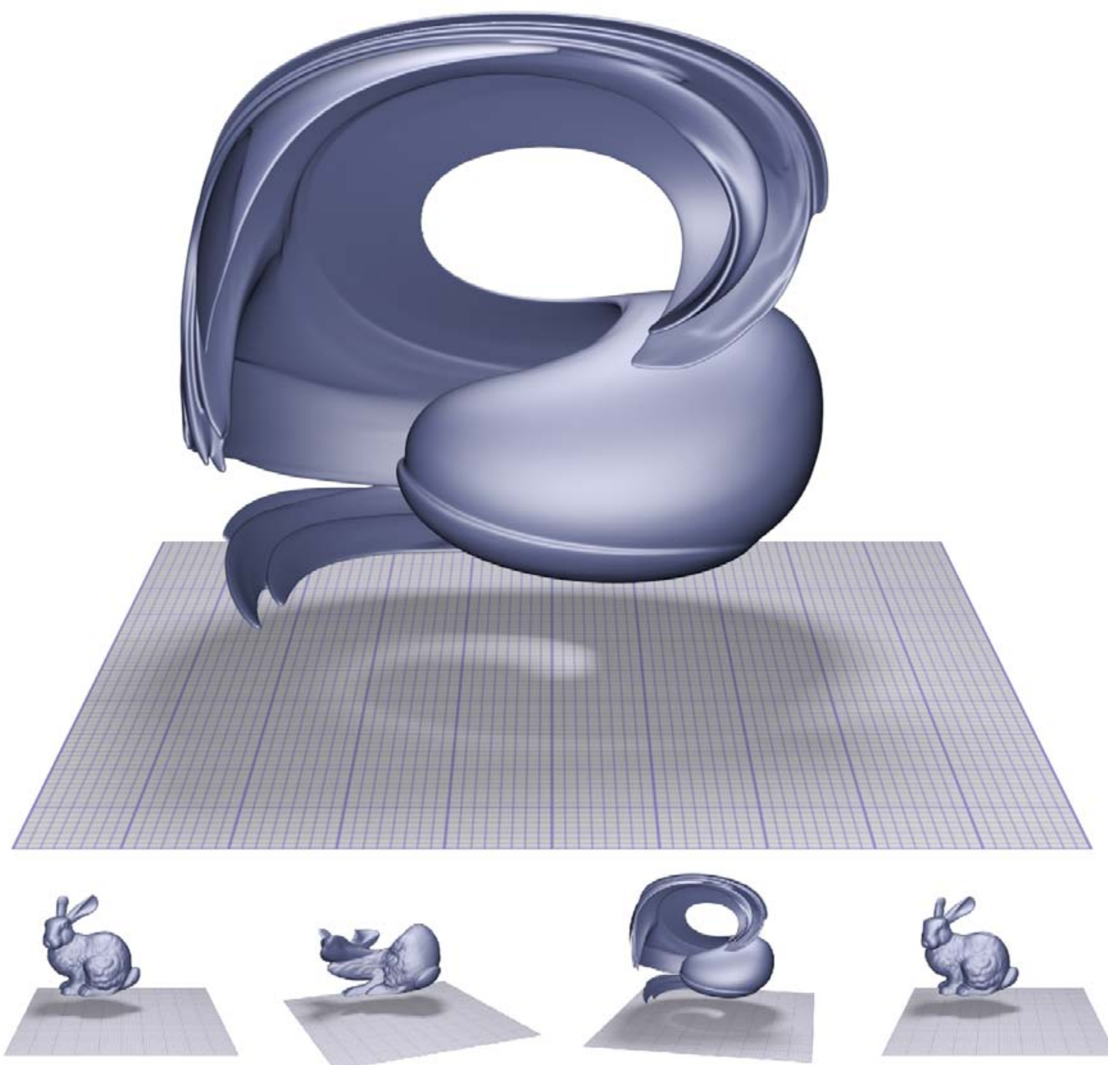


Figure 1.5: Level set based advection of the Stanford Bunny in a periodically symmetric and divergence free velocity field. After one period the Stanford Bunny returns to its original shape due to the properties of the velocity field as shown in the bottom-right picture. The deformation results in very thin walls requiring high resolution in order to be properly resolved. If the resolution was too low the Stanford Bunny would not return to its original shape. Effective resolution is 1024^3. Rendering by Ola Nilsson based on my simulation data and data structure implementation. Stanford Bunny courtesy of the Stanford Scanning Repository.

Figure 1.6: Partially out-of-core level set based fluid simulation of a splashing fountain with thin high resolution sheets of water. Both water surface, interior, velocities and boundaries are represented on a DT-Grid. Effective resolution is $931 \times 1567 \times 931$. The scene contains a total of 61.7M grid points and 332M particles. Rendering by Ola Nilsson and fluid simulation by Andreas Söderström. Andreas Söderström's fluid simulation code uses my implementations of the DT-Grid and out-of-core framework. Model courtesy of the Stanford Scanning Repository.

Part I

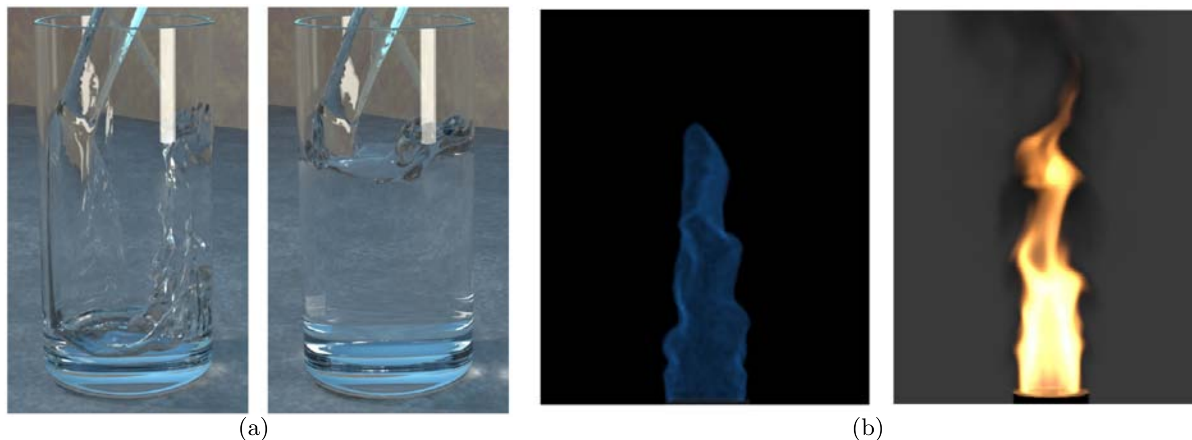
Level Set Methods

Chapter 2

Introduction to Level Set Methods

Practically every day of our lives we interact with and observe the behavior of complex surface deformations. Interacting with water for example seems so natural to us that we often fail to notice and appreciate the beautiful and complex ways in which it forms and splits apart. The techniques of computer graphics to a large extent strive to simulate or reproduce the visual and dynamic appearance of the world around us. This is particularly useful in cases where a visual expression is not easily obtained by an artist or animator attempting to portray a certain effect. Computer graphics unites several different scientific disciplines. For example, in order to arrive at techniques capable of reproducing more realistic behavior of water, we need to resolve to theory and practice from the disciplines of mathematics and physics. A *level set* is a mathematical description of a *dynamic implicit* surface possessing the properties necessary to represent complex surface deformations such as water. In the simple characterization of a level set as a dynamic implicit surface, *dynamic* refers to the ability of the surface to change over time, and *implicit* refers to the way the surface, or *interface*, is represented. The level set method was introduced in 1988 by Osher and Sethian [113] for interface tracking in computational physics. Since then significant efforts have been put into developing more accurate and robust numerical methods for solving the equations that govern the dynamic behavior of level sets. In addition, level sets have been applied as the fundamental surface representation in a wide range of problems spanning over a wide range of fields. In computer graphics and vision these include, but are not limited to, fluid simulations of water and fire [35, 37, 98], geometric modeling [96], shape metamorphosis [14], segmentation of volumetric datasets [160], the solution of partial differential equations on manifolds [10], collision detection in cloth simulation [16] and surface reconstruction [163]. Figures 2.1 and 2.2 show images from samples of the applications mentioned above. The interested reader should consult the book edited by Osher and Paragios [112] which contains descriptions of applications of level sets in imaging, vision and computer graphics.

Level sets offer significant and unique advantages over many other surface representations. By definition a level set cannot self-intersect (*i.e.* the situation where a surface crosses over itself) and complex changes in topology are handled automatically by the underlying mathematics. These advantages are not offered by explicit representations such as for example triangle meshes in wide use in computer graphics today. Furthermore, the numerical schemes for level set dynamics are in many cases simple to apply and offer explicit error control and generality. The applications of the level set method mentioned above all utilize its unique features. Fluid surfaces such as water for example behave in topologically complex ways by merging and pinching off in intricate and rich manners. By utilizing the level set as the underlying surface representation these properties are handled automatically [37], see figure 2.1. In geometric



(a) (b)

Figure 2.1: (a) Even the simplest interaction with water gives rise to topologically complex behavior. Here the pouring of water into a glass. The water surface is represented using level sets. Reprinted from [35]. (b) Level sets have been applied to physically based simulation of fire. Here a level set is used to represent the blue core of the flames. Reprinted from [98].

modeling the common work flows inherently lead to self-intersecting geometry when using traditional surface representations such as triangle meshes. This is undesirable if the models are to be used in for example physically based simulation or physical prototyping. By using level sets, self-intersecting geometry can be avoided and various surface editing operators that were previously very difficult can be utilized [96], see figure 2.2. Such surface editing operators are also useful for repairing 3D models scanned from real world geometry. Using scanned geometry is very common, an example of this is the Gollum character in the "Lord of the Rings" feature. Gollum was first modeled physically and later scanned into digital form and animated. While the fundamental idea of a level set as an implicit surface is relatively simple, its dynamic nature rests on a rather advanced mathematical and numerical framework that generalizes to any dimension. However, as we are primarily concerned with computer graphics we will mostly be working in three dimensions. Hence in this chapter we will restrict our attention accordingly to descriptions in three and - when simplifying the exposition - one or two dimensions. The present chapter is mostly devoted to describing the basics of the level set method. The next chapter presents an overview of extensions and improvements, the limitations of which to a large extent provide the motivation for our work presented in this dissertation. Briefly outlined we will first describe the ideas behind the implicit surface representation and contrast it to the explicit surface representation. Next we will describe the theory behind the dynamics of the level set method, or more precisely the equations that govern the movement. We end this chapter by describing how the level set theory is implemented numerically on a computer.

2.1 Implicit Surfaces

The properties of implicit surfaces in general are well understood mathematically. In computer graphics implicit surfaces come in many distinct forms, each associated with its own theory and unique properties. For an introduction see [11]. An *explicit* surface representation explicitly specifies the points on the surface. Mathematically speaking an explicit surface representation

Figure 2.2: An example in geometric modeling benefiting from the underlying level set representation. A winged two-headed dragon is created by reusing parts of existing models. The wings and heads of the dragon are smoothly blended with the base geometry without introducing self-intersections. Reprinted from [96].

provides a map between a parameter space and the surface. In computer graphics, explicit surface representations are typically *sampled* representations explicitly specifying a finite set of points lying on the surface as well as possibly the connectivity of these points and how to interpolate between them. Point-based surface representations, triangle meshes, subdivision surfaces and NURBS are all examples of such explicit surface representations.

An *implicit* surface representation on the other hand defines a surface as the *isocontour* of some scalar function. More specifically, given a scalar function, also called *surface embedding*, $\phi : \Re^3 \rightarrow \Re$, an implicit surface is represented as the preimage, $\phi^{-1}(k)$, of some scalar, k. This means that it consists of the set of points in $\Re^3$ described by $\{\mathbf{x}|\phi(\mathbf{x}) = k\}$. In this dissertation we will without loss of generality restrict our attention to *zero-isocontours* where $k = 0$ and hence surfaces given by the set $\{\mathbf{x}|\phi(\mathbf{x}) = 0\}$. Since a two-dimensional surface is defined by a three dimensional embedding function, the implicit surface is said to have co-dimension one.

The sphere and circle are shapes described very easily in implicit analytical form. A circle of radius r, for example, is given by the expression $\phi(x, y) = x^2 + y^2 - r^2 = 0$ and is shown in figure 2.3.a. An explicit representation of the same circle is depicted in figure 2.3.b. In contrast to explicit surface representations, an implicit surface does not explicitly specify which points lie on the surface. Instead it allows you to evaluate, given some point, whether that particular point actually lies on the surface or not. At first glance, this property may seem to make implicit surfaces inferior to explicit representations, in particular in the context of computer graphics. With respect to some applications this is true, however, the implicit representation also turns out to be powerful as will become evident in what follows.

The level set method assumes that implicit surfaces are closed. This means that the surface partitions $\Re^3$ into clearly defined interior and exterior regions. Note that the interior and exterior may consist of several disconnected components. Here we will assume that the implicit surface

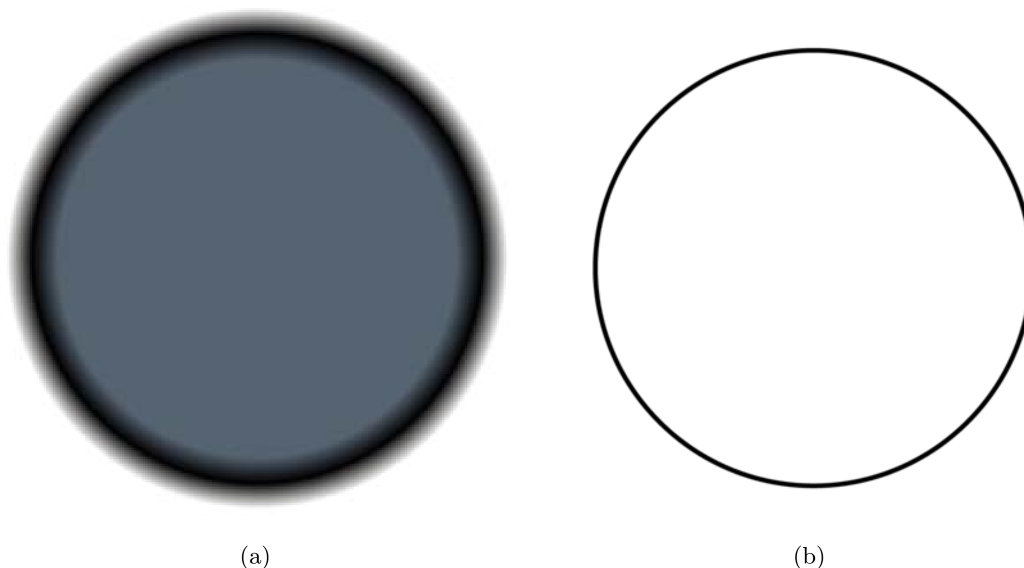


Figure 2.3: (a) A circle represented in implicit form as the zero isocontour of the 2D embedding function $\phi(x, y) = \sqrt{x^2 + y^2} - r$. In order to get a noticeable color gradient close to the interface, the color is clamped away from it. (b) A circle represented explicitly as $(\cos(\theta), \sin(\theta))$ where $\theta \in [0; 2\pi)$.

is being described by the zero-isocontour and that the surface embedding has the additional property that it maps points in interior and exterior regions to values of different sign. Looking again at the circle in figure 2.3.a, the color blue is associated with the interior region of negative sign, and the color grey/white is associated with the exterior region of positive sign. This is the sign convention that we will use throughout this dissertation. In particular we will denote the interior region by Ω^- and the exterior region by Ω^+.

The above properties immediately give rise to two important advantages of level sets. The first advantage is that given some location in space, estimating whether it is inside or outside of the surface simply amounts to evaluating the embedding function at the corresponding point and determining its sign. For explicit representations such as meshes this operation is more involved [5] and ambiguous in the case of meshes with holes and self-intersections. The second advantage is that a level set, and in general an implicit surface, cannot cross over itself and self-intersect. This property arises from the fact that an implicit surface is represented by a *single-valued* embedding function, *i.e.* a point in $\Re^3$ cannot at the same time have both negative and positive sign. This is very important in for example water simulations that require the specification of distinct physically realizable regions of space. Explicit representations on the other hand can easily form self-intersecting geometry when deformed.

In computer graphics and numerical simulation, level set surfaces are usually not represented analytically. Instead the embedding function is sampled on a *grid* with sample locations placed at the nodes of a lattice as shown in figure 2.4.a. A sampling of an explicit function is shown in figure 2.4.b. An implicit representation sampled on a grid is often referred to as an *Eulerian* representation, since it captures the interface instead of tracking it like the explicit representation which is referred to as a *Lagrangian* representation. Note that during interface deformations,

the grid points in an Eulerian representation remain fixed; it is the modification of the scalar values of the embedding function that causes the interface to move. In contrast it is the sample locations that move when a Lagrangian representation is subject to deformation [1]. There are various kinds of grids, but for level set simulations, dense uniform grids such as the one in figure 2.4.a are so far the most common. We will deal more with the issue of grid representations in chapter 3 and again in chapters 4 to 11. The reason that analytical representations are not used is that for the surfaces we are dealing with, no analytical expressions are immediately available or known. A sampled representation on the other hand can be computed to any closed and non-self-intersecting boundary representation as long as the surface is sufficiently band-limited, see part IV.

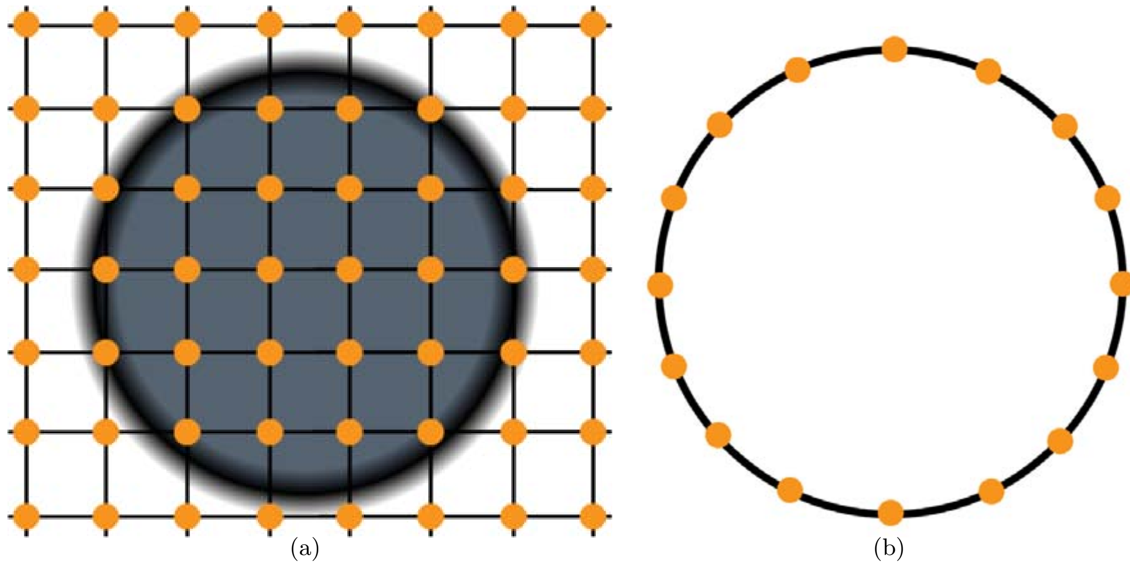


Figure 2.4: (a) A circle represented in implicit form as the zero-isocontour of the 2D embedding function $\phi(x, y) = \sqrt{x^2 + y^2} - r$ sampled on a dense uniform grid. Note that a relatively coarse grid is used in order to illustrate the principle of an Eulerian representation. (b) A circle represented explicitly by marker points connected by line-segments. Relatively few markers are used in order to properly illustrate the principle of the Lagrangian representation.

Many differential properties of implicit surfaces are easy to compute directly from the surface embedding. The outwards pointing surface normal is given by the normalized gradient:

$$\vec{N} = \frac{\nabla\phi}{|\nabla\phi|} \tag{2.1}$$

and the mean curvature is in three dimensions given by the divergence of the normal multiplied by one half (for a recent elaborate proof see [97])

$$\kappa = \frac{1}{2}\nabla \cdot \vec{N} = \frac{1}{2}\nabla \cdot \frac{\nabla\phi}{|\nabla\phi|} \tag{2.2}$$

where $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right)$ is a differential operator. In 2D the curvature is simply the divergence of the normal and hence the factor of one half should be omitted.

[1]Note that also hybrid Eulerian and Lagrangian representations exist. See for example [147] in which the grid dynamically adapts to the interface in order to capture it more accurately using fewer grid points.

Similarly, the expressions for evaluating integrals over the surface or its interior region are easily computed. This includes the computation of area and volume. For an overview the reader should consult the book by Osher and Fedkiw [111].

From a theoretical point of view, the embedding function is irrelevant as long as it is Lipschitz continuous. However, one particular implicit surface representation that has proven itself useful for both computer graphics and level set simulation is the *signed distance field*. In the signed distance field representation the embedding function ϕ assigns to each point in $\Re^3$ the shortest distance to the surface multiplied by a sign which is $+1$ in the exterior and -1 in the interior. In fact, the equation for a circle in implicit form given above and shown in figure 2.3.a is a signed distance function. Many operations and formulas simplify as the result of the unique properties of the signed distance field representation. In particular, not only the interior/exterior status of a given grid point can be evaluated by a single lookup, the same now goes for the minimum distance to the surface. This is useful in ray tracing where a certain technique known as *ray leaping* can be employed [35,48,85]. We will return to this in chapter 15. An additional property is that the length of the gradient in a signed distance field is identically one, *i.e.* $|\nabla\phi| = 1$, except at corners or kinks (jumps/discontinuities in the derivatives)[2] where the gradient is not defined. Since $|\nabla\phi| = 1$, the formula for the normal simplifies to the gradient and the formula for the mean curvature simplifies to the laplacian, *i.e.* $\vec{N} = \nabla\phi$ and $\kappa = \frac{1}{2}\Delta\phi$.

Another important fact that arises from utilizing that $|\nabla\phi| = 1$ and that the value in a point is the minimum (signed) distance, is that the closest point on the surface, $\mathbf{x_s}$, from any point, $\mathbf{x}$, in $\Re^3$ can be found by the formula $\mathbf{x_s} = \mathbf{x} - \phi(\mathbf{x})\nabla\phi(\mathbf{x})$. See [89] for a list of applications of this. Finally note that for explicit representations, computing differential properties [28] or locating the closest point on or the minimal distance to an explicit representation such as a triangle mesh is not as simple [5].

A level set will throughout this dissertation be represented in implicit form by means of a signed distance field. Next we turn our attention to the theory and practice of the level set method which adds dynamics to implicit surfaces. In chapters 13 to 14 we will deal with the issue of initializing signed distance fields and hence level set surfaces from polygonal meshes, which is the most widely used interchange format between today's commercial modeling tools.

2.2 The Theory of the Level Set Method

In the previous section we considered how a level set was *represented*. In particular it was stated that a level set is given in implicit form by an embedding scalar function, ϕ, which for our purposes will be the signed distance field. The level set theory assumes that ϕ is a Lipschitz function[3] which allows for sharp corners and edges on a surface and at the same time ensures that approximations to derivatives are bounded. In particular the signed distance field representation is a Lipschitz function, and in addition it is smooth except at kinks, making it well suited for numerical simulation. We now turn our attention to the theory behind the level set method [113] which adds *dynamics* to the implicit surface. In the next section we consider how the equations introduced in the present section can be discretized on a computer and summarize the basic numerical methods developed for solving them.

[2]For example points equidistant to several distinct points on the surface *e.g.* the center of a circle or sphere.

[3]A Lipschitz function, f, satisfies that $\frac{f(x)-f(x')}{x-x'} < C$ for some constant C and arbitrary x and x'.

The dynamics of a level set is manifested in a *level set equation* that must be solved in order to make the surface move. Several different level set equations exist and the one to use depends on the form most convenient for the problem at hand. We will start by considering the situation where we are given a time-dependent velocity field $V(\mathbf{x}, t) : \Re^3 \to \Re^3$ that assigns to each point in space a vector describing the velocity of the surface (or surface embedding) at that point. This form is for example the most convenient in fluid simulations, where the Navier Stokes equations are solved to produce a velocity field at each time step. Consider a point, $\mathbf{x}$, resting on an implicit surface moving over time. The movement of $\mathbf{x}$ traces out a path which can be given by an expression of the form $\mathbf{x}(t) = (x(t), y(t), z(t))$. Coupling this expression together with the surface embedding, ϕ, hence adding dynamics to ϕ, results in the following time-dependent equation for the surface given by the zero-crossing: $\phi(\mathbf{x}(t), t) = 0$. By applying differentiation using the chain rule to the time-dependent surface embedding, we obtain the equation:

$$\frac{\partial \phi}{\partial t} + \frac{d\mathbf{x}}{dt} \cdot \nabla \phi = 0 \tag{2.3}$$

Since $\frac{d\mathbf{x}}{dt}$ is tangent to the curve traced out by $\mathbf{x}$ we notice that $V(\mathbf{x}, t)$ can be used in place of $\frac{d\mathbf{x}}{dt}$. Hence, given a time dependent velocity field, $V(\mathbf{x}, t)$, the above equation can be used to evolve or *advect* the surface forward in time.

Equation 2.3 is a *Partial Differential Equation (PDE)* since it involves derivatives with respect to several variables. For an introduction to PDEs and their numerics consult the book by Strikwerda [143]. In addition equation 2.3 is an *initial value problem*: Starting from an initial surface embedding, the above equation is solved at each time step in order to obtain the deformed surface.

Another level set equation can be derived directly from equation 2.3 by using the relationship between the gradient and the normal of the surface embedding, *i.e.* use that $\vec{N} = \frac{\nabla \phi}{|\nabla \phi|}$. Decomposing the velocity field V into its normal, $V_N \vec{N}$, and tangential, $V_T \vec{T}$, vector components, where V_N and V_T are scalar fields specifying the speed in the normal and tangential directions respectively, we get $V \cdot \nabla \phi = (V_N \vec{N} + V_T \vec{T}) \cdot \nabla \phi = V_N |\nabla \phi|$. Hence the level set equation 2.3 becomes

$$\frac{\partial \phi}{\partial t} + V_N |\nabla \phi| = 0 \tag{2.4}$$

As can be seen from equation 2.4 the tangential component vanishes and in fact only the normal component of the velocity is significant for the motion of the surface. This does not mean however that equation 2.3 is rendered redundant. As explained above, when a velocity field V is available, the form of equation 2.3 is the most convenient. Many other problems though are more easily described by their speed in the normal direction, V_N, and hence by equation 2.4. This goes for shape metamorphosis, smoothing as well as erosion, dilation and others.

In some contexts, the speed in the normal direction is dubbed the *speed function* and in general it may depend on spatial position, time, geometrical and differential properties of ϕ and so on. Take surface smoothing for example. In that case the speed function is simply given by the negated mean curvature, *i.e.* $V_N = -\kappa$.

The two equations above are the most commonly occurring level set equations in graphics. However, more terms can be added, and in particular equations 2.3 and 2.4 can be combined into a single equation. For more examples, see [111].

Although there are situations where it is not the case [111, 130], in general a signed distance field representation subjected to movement by any of equations 2.3 and 2.4 will cease to remain

a signed distance field. This is due to the fact that not all isocontours move at identical speeds. Hence in some areas the isocontours will cluster and in others they will spread apart, in this way departing from the signed distance field representation. Since the signed distance function comes with many advantages, the common solution to this is to reset the surface embedding to a signed distance field[4] after each advection step. There are several ways to do this, and in this dissertation we will focus on two of these.

The first approach is to solve what is known as the *reinitialization equation* [120, 124, 145] which has the form

$$\frac{\partial \phi}{\partial t} + S(\phi)(|\nabla \phi| - 1) = 0 \tag{2.5}$$

where $S(\phi)$ is a *sign function*. The simplest possible sign function is given by $S(\phi) = \text{sign}(\phi) \in \{-1, +1\}$ which just takes on the sign of ϕ in the grid point evaluated. To obtain a signed distance field, equation 2.5 must be solved to steady state, meaning that $\frac{\partial \phi}{\partial t} = 0$ and hence that $|\nabla \phi| = 1$, a unique characterization of the signed distance field. Notice the similarity of equations 2.5 and 2.4. In fact equation 2.5 propagates distance information in the normal direction with speed $S(\phi)$.

When applied numerically, the sign function proposed above does not work sufficiently well as it tends to move the zero-crossing which must remain fixed [111]. Instead we typically apply the smeared out sign function proposed in [120]

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + |\nabla \phi|^2 (\Delta x)^2}} \tag{2.6}$$

where Δx is the distance between two axially adjacent points in the grid (we return to this issue in the next section). Even though this smeared out sign function still has an undesired tendency to move or smooth the surface slightly, it works quite well in practice. See [111] for a recent overview of reinitialization methods and sign functions.

An alternative to the construction of a signed distance field comes from considering the *Eikonal equation*:

$$|\nabla \phi| = 1 \tag{2.7}$$

This Eikonal equation is again a PDE, but in contrast to the other PDEs presented in this chapter it has no time-dependence. In particular it is not an initial value problem but rather a *boundary value problem*: Given the values of the signed distance field on the boundary of the zero-isocontour, the above equation can be solved to determine the signed distance values away from the zero crossing. Even though equation 2.7 is not time dependent, it can be viewed as propagating the boundary outwards in the normal direction with unit speed. The value computed at each grid point is then just the *time of arrival* of the boundary, which due to the unit speed propagation, equals the distance. As we will see in the next section, fast methods exist for solving this equation.

The theory of level sets is a vast area in itself. Above we have elaborated on the most basic theory comprising the material needed to comprehend the methods presented in this dissertation. For a more thorough overview consult the book by Osher and Fedkiw [111]. Having described

[4] Resetting the surface embedding to a signed field *without* moving the location of the zero crossing

the necessary theory of level set methods we now turn to their discretization and solution on a computer.

2.3 The Numerics of the Level Set Method

In this section we consider how the theory developed in the previous section can be discretized and solved on a computer. We will also consider how discretized versions of the differential operators such as normal and mean curvature can be computed. However, before proceeding with these tasks let us first consider why explicit surface representations are not always well suited for general surface deformations based on numerical simulation. Surface deformations of explicit surface representations, such as meshes, occur frequently in computer graphics. While these representations work very well for *e.g.* character animations, rigid body simulations and minor surface deformations, they usually break down when surfaces are undergoing large and/or topologically complex deformations. When an explicit surface representation is subject to a large deformation, the distribution of points on the surface can change dramatically no matter how good it is initially. If the points come too far apart, aliasing occurs, and if they come too close, singularities in differential properties may arise. The latter in particular is disastrous from a numerical simulation point of view, see [130] for an example. Although there are remedies for this such as re-meshing, smoothing and filtering, these corrections are non-trivial and non-physical. Hence they may affect the simulations in unpredictable and undesirable ways. Yet another important fact which has already been mentioned is that nothing prevents explicit representations from self-intersecting. Front tracking methods that detect self-intersections and merge and break up the geometry have been proposed (consult [111] for a recent overview), but are usually quite complex. Numerical level set methods on the other hand do not suffer from increased aliasing and numerical instability. This is because the surface embedding is sampled on a grid where the sample points are *fixed*, recall figure 2.4.a. Furthermore topological changes are automatically handled by the surface embedding. As with many other Eulerian schemes in computational fluid mechanics, level set methods do however implicitly introduce a smoothing of the solution known as *numerical dissipation* or *artificial viscosity*. We will return to this and explain its origin and consequences later in this chapter. In chapter 3 we will furthermore review a method known as the *particle level set method* [33] which reduces the numerical dissipation for certain types of deformations.

2.3.1 Approximation of Derivatives with Finite Differences

Having motivated the implicit surface approach to deforming surfaces we now turn to the discretization of differential operators on an Eulerian grid. Let us first introduce some notation. Assume that the spacing between adjacent grid points in the grid is given by Δx, Δy and Δz in each dimension respectively, and that the time step in the simulation is Δt. The notation ϕ^n_{ijk}, where i, j, k and n are integers, is shorthand for $\phi(i\Delta x, j\Delta y, k\Delta z, n\Delta t)$. The time and/or spatial dependence may be omitted for clarity when not relevant.

A *finite difference (FD)* [143] is a discrete approximation to a derivative and can be derived directly from a Taylor expansion. For example, a one-sided *forward* FD approximation to $\frac{\partial \phi}{\partial x}$ can be derived from the Taylor series approximation to the point $x + \Delta x$ about x given by $\phi(x + \Delta x, y, z) = \phi(x, y, z) + \Delta x \frac{\partial \phi}{\partial x} + O(\Delta x^2)$. Rearranging terms this becomes $\frac{\partial \phi}{\partial x} = \frac{\phi(x+\Delta x,y,z)-\phi(x,y,z)}{\Delta x} + O(\Delta x)$. Omitting the $O(\Delta x)$ term and hence introducing a *truncation*

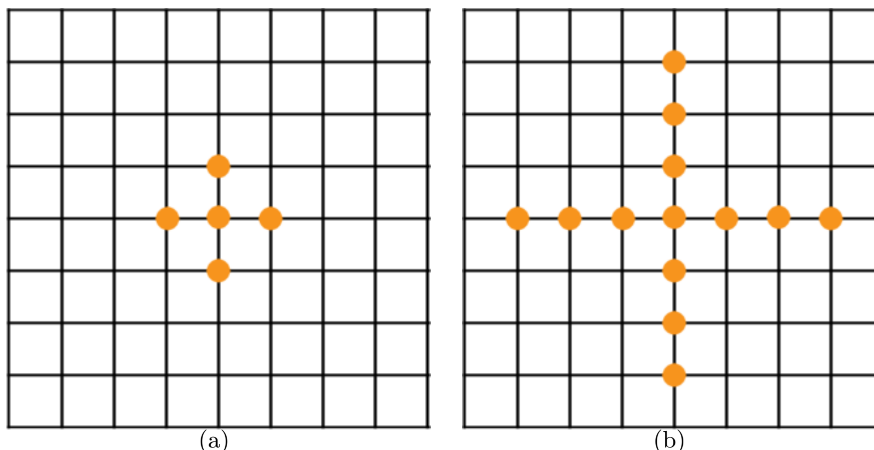


Figure 2.5: (a) The FD stencil for first order one-sided (backwards and forwards) differencing and second order central differencing. (b) The FD stencil for third order HJ ENO differencing and fifth order HJ WENO differencing. These finite difference methods are described in detail in the text.

error we obtain the one-sided *forward* FD, $\phi_x^+ = \frac{\phi_{i+1jk}-\phi_{ijk}}{\Delta x}$, which is *first order* accurate since the order of the truncation is $O(\Delta x)$. In general a FD approximation with a truncation error of $O(\Delta x^m)$ is said to be *m'th* order accurate. The point in which the finite difference is evaluated is only specified if not clear from context or if of relevance to the exposition. Derived similarly, first order one-sided *backwards* and second order *central* approximations to $\frac{\partial \phi}{\partial x}$ are respectively $\phi_x^- = \frac{\phi_{ijk}-\phi_{i-1jk}}{\Delta x} \approx \frac{\partial \phi}{\partial x}$ and $\phi_x^0 = \frac{\phi_{i+1jk}-\phi_{i-1jk}}{2\Delta x} \approx \frac{\partial \phi}{\partial x}$. In general an $n'th$ order accurate FD approximation to an $m'th$ order derivative requires the contribution from $m+n$ adjacent grid points [5].

The set of grid points needed for the computation of a finite difference is generally referred to as the *stencil*. Figure 2.5 shows two different FD stencils.

Differential properties such as normal and curvature can easily be approximated with the above finite differences. The actual approximation chosen depends on the desired accuracy as well as the properties of the numerical method involved.

In practice spatial and temporal discretizations are treated somewhat differently. We elaborate on the spatial derivatives first.

Spatial Discretizations

A number of higher order finite difference methods have been developed specifically for level sets. In addition to representing the surface with higher accuracy we wish to maintain sharp corners and edges on the surface *i.e.* places where the derivatives are discontinous.

The *Hamilton-Jacobi Essentially Non Oscillatory* (HJ ENO) [46] FD scheme uses Newton polynomial interpolation [68] to reconstruct ϕ and then differentiates this polynomial to obtain approximations to the derivatives. In each step of the construction of the interpolating Newton polynomial, the minimal possible *divided difference* [68] is chosen since this usually corresponds

[5] Fewer grid points may be needed due to symmetries in the Taylor expansions. For example it is possible to construct a second order accurate approximation to the second order derivative using only three grid points.

to the smoothest polynomial. In this way HJ ENO attempts to minimize the oscillations that can arise from interpolating across a discontinuity in the derivative and thus to obtain better approximations near corners and edges [111]. Although *third order* accurate HJ ENO schemes are the most common in level set computations, it is possible to construct them to any order of accuracy.

The variable three to fifth order accurate *Hamilton-Jacobi Weighted ENO* (HJ WENO) [58, 59, 80] FD scheme employs a convex combination of the three different third order accurate HJ ENO approximations possible in each grid point. By choosing a certain weighting of the HJ ENO approximations, a fifth order accurate scheme is obtained in smooth regions. Away from smooth regions, the HJ WENO adapts to the local neighborhood and uses a weighting that favors the third order accurate HJ ENO approximations not interpolating across discontinuities.

Although the expressions get more involved and the computations more time-consuming, the higher order schemes can in many cases dramatically improve both the numerical accuracy and visual quality of the surface deformation. Due to complexity of the formulations, we refer the reader to the original papers or [111] for the implementation details of the HJ ENO and HJ WENO schemes.

Temporal Discretizations

For level set equations a *forward Euler* time step is employed when first order temporal accuracy is sufficient. Basically a forward Euler time step is just a one-sided forward difference in time: $\frac{\phi^{n+1}-\phi^n}{\Delta t}$. To obtain higher order accuracy in time, *Total Variation Diminishing Runge-Kutta* (TVD RK) [133] methods that diminish oscillations can be employed.

A second order accurate TVD RK method proceeds as follows. First the solution, ϕ^n, is advanced two steps forward in time using a forward Euler time step to obtain ϕ^{n+2}. Finally the average is formed:

$$\phi^{n+1} = \frac{1}{2}\phi^n + \frac{1}{2}\phi^{n+2} \tag{2.8}$$

A third order accurate TVD RK method proceeds in a similar fashion. Consult the original papers or [111] for the implementation details.

2.3.2 Numerical Stability

In order to be useful, any numerical scheme for solving a PDE must be *convergent* [143]. In brief this means that the approximate solution computed by the numerical scheme must converge to the exact solution as $\Delta x, \Delta t \to 0$. In general convergence is hard to prove, but according to the Lax-Richtmyer theorem [143], convergence is equivalent to *stability* **and** *consistency*, two properties that are easier to prove [143]. In particular the concept of stability has implications in practical implementations. In brief stability means that the norm of the numerical solution at any point in time must be bounded by the sum of the norm of the numerical solution at a fixed number of earlier time steps. Hence without the requirement of stability, the approximate solution of a numerical scheme could potentially blow up and grow unempidedly. In the level set community, *explicit numerical schemes* are typically used to solve the level set equations. This can in part be attributed to the non-linearity of equation 2.4. In an explicit numerical scheme, the approximation in a single grid point to time $t + \Delta t$ depends only on values at grid points to time t and/or earlier. This type of scheme usually has a limited stability region, meaning that there are restrictions on the size of Δt given Δx. One exception though is the semi-Lagrangian

scheme widely used for fluid simulations [137] and to some extent also level set simulations [141]. The degree of numerical dissipation introduced by the semi-Lagrangian methods however limit their utilization for level sets unless used in combination with the particle level set method [33] for improved interface capture or higher order accurate monotonic interpolation [36]. A review of the particle level set method and semi-Lagrangian techniques will be provided in the next chapter.

Below we state the time step restrictions required for each type of level set equation in order to enforce stability of the numerical solution.

2.3.3 Numerical Solution of the Level Set Equations

Next we concentrate on the numerical solution methods for level set equations. The numerical schemes are given here in one dimension only. The extension to higher dimensions can be done in a straightforward coordinate-wise manner.

Recall that equations 2.3 and 2.4 defined in section 2.2 are mathematically equivalent. However, seen from a numerical perspective, different methods have to be applied, since equation 2.3 is linear in the partial derivatives, whereas equation 2.4 is non-linear due to its $|\nabla\phi|$ term.

Hyperbolic Level Set Equations

When the speed function and velocity field does not depend on derivatives of higher than first order, equations 2.3 and 2.4 are known as *Hamilton-Jacobi Equations*, a special type of *hyperbolic* PDEs [111]. In fact their solutions are constant along curves, $\mathbf{x}(t) = (x(t), y(t), z(t))$, known as *characteristics* [6]. The characteristic curves constitute the *domain of dependence*. This mathematical property immediately suggests a method of solution to equation 2.3 known as *upwinding*, where the direction of movement of a point on the surface is denoted the *downwind* direction. The upwind method includes more grid points in the upwind direction than in the downwind direction when computing the finite difference approximations to the spatial derivatives and is required for numerical stability. For first order FD approximations this corresponds to one-sided forward or backward differences respectively. The main physical intuition is that the upwind direction is the *direction* from which known information is flowing or emanating.

Let us write up a solution method for equation 2.3 based on our observations. If discretizing the temporal derivative in equation 2.3 by a forward Euler time step, we get the following method of solution

$$\phi_{ijk}^{n+1} = \phi_{ijk}^{n} + \Delta t \max(V,0)\phi_x^- + \min(V,0)\phi_x^+ \tag{2.9}$$

where the max and min operators are evaluated in a single grid point. Note how the upwind scheme above computes the derivatives: If $V(x) < 0$ we use ϕ_x^+, *i.e.* a forward difference, and if $V(x) > 0$ we use ϕ_x^-, *i.e.* a backward difference. In any case we estimate the derivatives *upwind* by favoring grid points in the direction from which information is flowing. The derivatives ϕ_x^- and ϕ_x^+ can be computed either using first order one-sided differences or using the HJ ENO or HJ WENO finite difference schemes.

The above method of solution has a limited *stability region*, meaning that the time steps are restricted by the grid spacing. A necessary (but not sufficient) requirement for numerical stability is to abide to the *Courant-Friedreichs-Lewy* (CFL) [143] condition. In the case of

[6] In general the solution along a characteristic needs not be constant. Rather it should satisfy an *Ordinary Differential Equation* (ODE), involving derivatives of a single variable only.

the equation above it leads to a time step restriction of $\Delta t < \frac{\Delta x}{\max\{|V|\}}$ [111]. Note that this restricts interface movements to maximally one grid point per time step as the above implies $\max\{|V|\}\Delta t < \Delta x$.

As mentioned above, equation 2.4 is nonlinear due to the $|\nabla\phi|$ term. A widely used numerical scheme for computing this term is the scheme due to Gudonov [44, 124]:

$$\phi_x^2 = \begin{cases} \max(\max(\phi_x^-, 0)^2, \min(\phi_x^+, 0)^2) & \text{if } V_N > 0 \\ \max(\min(\phi_x^-, 0)^2, \max(\phi_x^+, 0)^2) & \text{if } V_N < 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

For higher dimensions, *e.g.* three, ϕ_y^2 and ϕ_z^2 are computed similarly, and the norm of the gradient subsequently calculated as $|\nabla\phi| = \sqrt{\phi_x^2 + \phi_y^2 + \phi_z^2}$. Combined with for example a forward Euler time step the Godunov scheme provides a solution method for level set equation 2.4. For a good explanation of the Godunov scheme the reader should consult the original references or see [111].

For the equation above, the CFL condition is a bit more involved, but a conservative estimate leads to a time step restriction of $\Delta t < \frac{\Delta x}{\max\{|V_N|\}d}$, where d is the dimension [111].

Due to the similarities between level set equation 2.4 and the reinitialization equation 2.6, the Godunov scheme can immediately be applied to the solution of the reinitialization equation as well. Note that the reinitialization equation must be solved to steady state. However in practice a fixed number of iterations is usually used as opposed to checking for steady state [120]. This fixed number of iterations can be chosen based on the width of the *band* (see below for the narrow band level set method) of grid points around the zero isocontour that needs to be reinitialized. One just has to keep in mind that distance information propagates in the normal direction with approximately unit speed.

Above we used a first order forward Euler time step for discretizing the temporal derivatives. In practice, higher order methods may be needed to improve accuracy and visual quality. Second and third order TVD Runge Kutta methods [133] are typically used.

Parabolic Level Set Equations

When the speed function or velocity field depends on mean curvature, the level set equations typically become *parabolic*. The parabolic equations have different mathematical, numerical and physical properties than the hyperbolic equations and hence we must solve them differently. Mathematically, the solution to a parabolic equation at a point in space does not flow along characteristic curves. Rather information flows into this single point from all other positions in space, and physically speaking information travels through the domain with infinite speed, *i.e.* a perturbation in one part of the domain will have immediate influence at all other positions. One could also say that infinitely many characteristics intersect in each grid point or that the parabolic equation has an *infinite domain of dependence*. Consider level set equation 2.4 with $V_N = -b\kappa$, where κ is the mean curvature and b is a positive scalar. This equation is parabolic and for $b = 1$ describes a smoothing of the surface. The parabolic equations have the more strict time step requirement of $\Delta t < \frac{\Delta x^2}{2\max\{b\}d}$, where d is the dimension of the level set embedding.

In practice the velocity field in level set equation 2.3 is typically an *externally generated* velocity field, meaning that it does not depend directly on any differential properties of the surface. Hence we do not consider it here as it is usually a hyperbolic equation which was treated above.

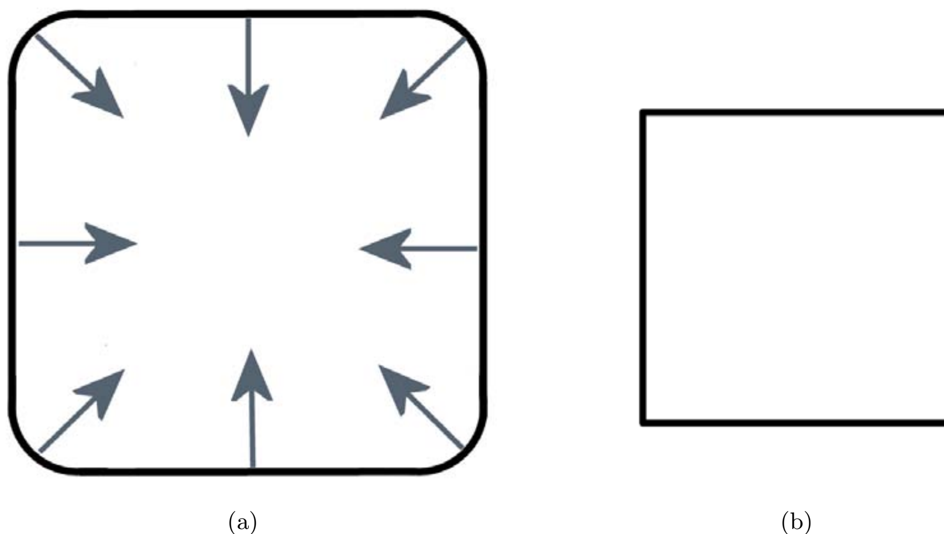
The Eikonal Equation

A prevalent method for computing a first or second order accurate solution to the Eikonal equation 2.7, which is an *elliptic* PDE, is known as the *Fast Marching Method* (FMM) [131, 154, 155]. In particular FMM has time complexity $O(M\log M)$ where M is the number of grid points in the grid.

Very briefly described, FMM resets the level set function to a signed distance function as follows: First, the grid points on the zero crossing (grid points with at least one neighbor of different sign) are located. Second, all grid points for which the signed distance should be computed are tagged *Alive*, *Close* or *Far* depending on whether the grid point in question is contained in the zero crossing, is adjacent to the zero crossing or is further away from the zero-crossing respectively. Third, a tentative distance for all grid points tagged *Close* is computed and the grid points are inserted into a heap sorted in ascending order of distance (minimal element on top). Fourth, iteratively the smallest grid point is removed from the heap and its tag changed to *Alive*. Next its final (signed) distance is set equal to its tentative distance and the distances of its neighboring grid points are updated. Any neighbors tagged *Far* change status to *Close* and are inserted into the heap along with their tentative distance. This procedure continues as long as the heap is non-empty and at termination the (signed) distance at all grid points has been computed. In the computations above it is important that tentative distances are estimated based on neighbors tagged *Alive* only.

The order of distance calculations in FMM ensures that only grid points closer to the surface contribute to the computation of the distance at a given grid point. This important property guarantees that characteristics flow away from the surface which is inherent to PDE-based solutions to the signed distance function.

2.3.4 Vanishing Viscosity Solutions and Numerical Dissipation



(a) (b)

Figure 2.6: (a) An initial C^1 box with rounded corners propagating inwards with unit normal speed. (b) A non-differentiable box with sharp corners resulting from the propagation in (a).

The theory underlying level set methods was studied by Osher and Sethian [111, 113, 130]. One of the main challenges is that when considering interface deformations, the desired physically plausible and non-self-intersecting solution is not necessarily differentiable. This is true even when the interface is initially differentiable. As an example, consider the box with rounded corners shown in figure 2.6.a. Clearly this interface is C^1. However, if propagated inwards with unit speed in the normal direction, it will eventually turn into the box with sharp corners shown in figure 2.6.b. These corners, at which the interface is not differentiable, will continue to exist until the box collapses into a single point. This behaviour is desirable from a physical and visual point of view. However, it is not trivial to construct a scheme that behaves in exactly this way.

So how does one mathematically define the desired physically plausible solution to the level set equations and how does one construct numerical schemes that automatically select this desired solution?

Since the solution we are interested in is not necessarily differentiable it is known as a *weak solution* [7]. Weak solutions are not unique however, so to single out a solution, the notion of an *entropy condition*, which defines the physically desirable and unique solution, is introduced. The solution dictated by the entropy condition in turn equals the *vanishing viscosity solution*. In brief the vanishing viscosity solution is obtained by adding a smoothing term, $-\epsilon\kappa$, to the right hand side of the level set equation and letting the *artificial viscosity*, ϵ, tend to zero. In computational fluid mechanics this term is often added explicitly to obtain stability in the presence of shock waves (discontinuities). The numerical methods for level set equations do not add the smoothing term explicitly as this typically results in excessive smoothing. However, the schemes *implicitly* cause a smoothing effect known as *numerical dissipation*. Numerical dissipation can be understood by looking at the discretized schemes from a different point of view. Instead of thinking of the discretizations as producing a truncated solution to the level equation, one can instead view them as producing an exact (when disregarding floating point roundoff error) solution to a slightly *different* equation. This slightly different equation can be derived and in fact contains a smoothing term which for a first order one-dimensional method has the form $O(\Delta x \frac{\partial^2 \phi}{\partial x^2})$ [3] where Δx implicitly functions as artificial viscosity. This explains why the first order numerical schemes in practice introduce smoothing even though no artificial viscosity is explicitly introduced on the right hand side of the equations. Note however that the numerical dissipation tends to zero as the resolution of the grid is increased and $\Delta x \to 0$. Finally we note that both the upwind and the Godunov schemes described in the previous section automatically pick out the physically plausible vanishing viscosity solution. If the reader is further interested in entropy conditions and vanishing viscosity solutions, we refer to the books by Sethian [130] and Osher and Fedkiw [111] as well as the references therein.

2.4 The Narrow Band Level Set Method

The *narrow band*, or *localized*, level set methods [1,22,108,120,161] exploit the property that only grid points in close vicinity of the zero crossing are required in order to compute the movement of the surface. More specifically, a narrow band level set method maintains only the grid points inside a *tube*, or *narrow band*, of width δ. A tube of width δ is defined as the set of grid points $\{\mathbf{x} | |\phi(\mathbf{x})| < \delta\}$. The actual width of the narrow band depends on the number of grid points in the stencil of the FD schemes applied as well as the maximum distance the surface can travel

[7] In general a weak solution satisfies an integral formulation of the PDE, where no derivatives of the solution are involved. The weak solution agrees with the solution of the PDE at all differentiable points.

between time steps. Recall that due to stability issues the explicit methods are restricted with respect to the time step they are allowed to take. In case of the CFL condition, this implies that the surface moves less than one grid point in each time step. For example, from the time step restriction for equation 2.3 it immediately follows that $\max\{|V|\}\Delta t < \Delta x$, where $\max\{|V|\}\Delta t$ is the maximum distance traveled by the surface in one time step, and Δx is the distance between grid points. Even though unconditionally stable schemes, such as the semi-Lagrangian scheme, are not hampered by time step restrictions, a maximum travel distance between time steps is usually enforced. This is done in order to ensure a desired numerical accuracy (which depends on the size of the time step) and to make sure the surface does not move beyond the narrow band in one time step.

In the description to follow we concentrate on the narrow band method of Peng *et al.* [120] as well as a recent extension that increases its computational efficiency [108]. In addition we assume that the surface moves at most one grid point (a distance of Δx) in each time step.

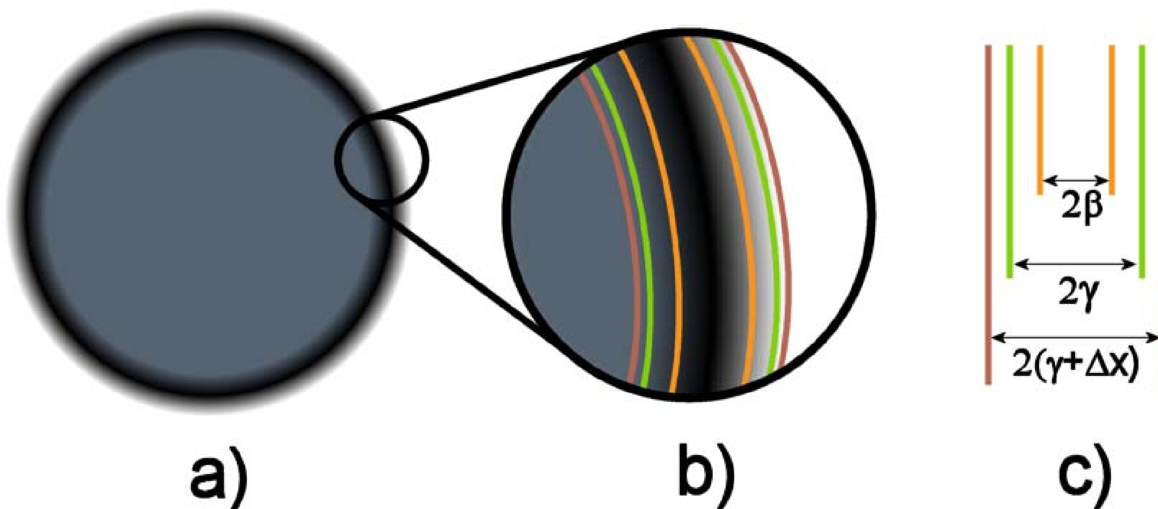


Figure 2.7: (a) A narrow band level set with values outside the narrow band clamped to $\pm\gamma$. (b) The outline of the tubes constituting the narrow band. In particular the β tube is shown in orange, the γ tube shown in green and the entire narrow band tube shown in brown. (c) The specifications of the absolute distance range covered by each tube.

The method of Peng *et al.* represents the narrow band as three concentric overlapping tubes. Figure 2.7 depicts this situation. In particular the narrow band equals the largest tube of width $\gamma + \Delta x$ and contained therein are two smaller tubes, the γ and β tubes respectively ($\gamma > \beta$). Values inside the γ tube are signed distances, whereas the values outside are explicitly clamped to $\pm\gamma$ with negative values assigned to the interior and positive values assigned to the exterior [8]. The reason that the narrow band is one grid point wider than the γ tube will become evident below. For the first order upwind schemes typically $\beta = 2\Delta x$ and $\gamma = 3\Delta x$ whereas for a fifth order WENO scheme $\beta = 3\Delta x$ and $\gamma = 6\Delta x$ [120].

A few additional data structures are required to support the narrow band method in order to obtain a computational complexity linear in the number of grid points in the narrow band.

[8] Note that this means that the unique properties of the signed distance function are only satisfied in the narrow band. For example, outside the narrow band only the clamped signed distance is available. This is however still useful for *e.g.* ray tracing.

The first data structure is a *coordinate array* storing the coordinates of the grid points in the narrow band. This enables iteration over the narrow band in time proportional to the number of grid points it contains. The second data structure called the *mask array* is a dense uniform grid of the same dimensions as the computational grid which for each grid point indicates which tubes it lies in, if any.

The computational cycle of the narrow band level set method consists of three steps:

1. Advect the level set in the γ tube by solving the level set equations.
2. Reinitialize the level set in the narrow band by solving the reinitialization equation.
3. Rebuild the narrow band to account for surface movement.

Each of these steps is described in detail below.

Advection

For grid points inside the β tube, the numerical solution to the level set equations 2.3 and 2.4 is computed exactly as outlined in section 2.3.3. However, for grid points inside the γ tube with $\beta \leq |\phi(\mathbf{x})| < \gamma$, the solution is modified by a cut-off function in order to avoid numerical oscillations at the boundary of the narrow band. Thus, equation 2.4 for example is modified to

$$\frac{\partial \phi}{\partial t} + c(\phi) V_N |\nabla \phi| = 0 \tag{2.11}$$

where $c(\phi)$ is defined as

$$c(\phi) = \begin{cases} 1 & \text{if } |\phi| \leq \beta \\ \frac{(|\phi|-\gamma)^2(2\phi|+\gamma-3\beta)}{(\gamma-\beta)^3} & \text{if } \beta < |\phi| \leq \gamma \\ 0 & \text{otherwise} \end{cases}$$

Note that the $c(\phi)$ function transitions smoothly from one to zero as the distance inside the γ tube increases. For grid points outside the γ tube the solution to the level set equation is not computed.

Reinitialization

After advection, the reinitialization equation 2.5 is solved to steady state. Contrary to the level set equations, the reinitialization equation is solved exactly as described in section 2.3.3 in the *entire* narrow band. The reason that the computations in this case encompass all grid points in the narrow band is that after reinitialization, the γ tube of the *propagated* surface will be contained within the narrow band. This is due to the fact that the surface will at most move one grid point between time steps and that the narrow band is exactly one grid point wider than the γ tube, see figure 2.7.

Rebuilding The Narrow Band

After reinitialization, the narrow band must be rebuilt to accommodate the surface movement. In particular, the new β, γ and $\gamma + \Delta x$ tubes must be determined. In its original formulation, the method of Peng *et al.* [120] proposed a method for rebuilding the narrow band having a computational complexity of $O(L^3)$, where L is the side-length of the grid. Their argument was that

the rebuild procedure was at most carried out once per computational cycle whereas iteration over the narrow band was performed several times during both advection and reinitialization. However, as we will see in chapter 7, this $O(L^3)$ time complexity can become quite dominant in the case of larger grids and hence a method with time complexity linear in the number of grid points in the narrow band is preferable. One such solution was recently proposed in [108]. In practice this can for a grid of size 256^3 give rise to a speedup of two [48]. The optimized rebuild algorithm works by iterating over the grid points in the old narrow band and in a single pass determines the grid points in the new narrow band. In particular, the following procedure is executed for each grid point, (x, y, z), visited: If $|\phi(x, y, z)| < \gamma$, (x, y, z) is included in the new narrow band. Additionally, if (x, y, z) was not inside the old γ tube, all neighbors of (x, y, z) not inside the old narrow band are inserted into the new narrow band as well[9]. If on the other hand $|\phi(x, y, z)| > \gamma$, all neighbors of (x, y, z) are examined, and if any of them has an absolute value below γ, the grid point (x, y, z) is included in the new narrow band.

2.5 Advantages and Disadvantages of Level Sets

We now proceed to summarize and outline the advantages and disadvantages of the level set method. Due to our main application area, we focus on the properties most significant to computer graphics. We stress that so far no silver bullet surface representation has been proposed. Triangle meshes, NURBS, subdivision surfaces, point based representations and level sets each have their unique set of advantages. However, as elaborated above, level sets are particularly powerful in the context of physically based animation. A field which in recent years has experienced a significant increase in attention and practical use.

The main advantages of level sets are that they:

- Avoid self-intersecting geometry.
- Allow for arbitrary change in topology and are free of mesh connectivity issues.
- Allow for easy computation of local differential properties such as gradient, normal and curvature, even with higher order of accuracy.
- Rely on vanishing viscosity solution to get physically and visually plausible behavior.
- Offer explicit error control and high order of accuracy.
- Generalize to any dimension.
- Enable global properties such as area and volume to be easily computed.
- Make geometric queries such as inside, outside and closest point tests trivial.
- Avoid aliasing when the surface undergoes large deformations.
- Allow for easy computation of Constructive Solid Geometry (CSG) operations such as the union, intersection and difference of solids.
- Make surface off-setting simple and very fast.
- Can be rendering directly using ray tracing, hence conversion to a mesh representation may not be necessary.

[9]Note that this is the only way grid points can enter the narrow band.

Our work addresses three disadvantages of the original level set method:

- Computational inefficiency: Computational requirements are proportional to the volume of the embedding.
- Storage inefficiency: Storage requirements are proportional to the volume of the embedding.
- Predefined static computational domain: Level set deformations are confined to stay within the boundaries of a predefined computational grid.

As we will see in the next chapter these disadvantages have to some extent also been addressed by previous and concurrent work, however no single previous method has eliminated all disadvantages. The remaining significant disadvantages of level sets include:

- Even though the solution of the level set equations can now be restricted to a narrow band and hence scales with the size of the interface, accurate level set methods (*e.g.* HJ WENO combined with TVD Runge Kutta) are still very time consuming compared to mesh based deformations. This is the case as the relatively heavy PDE computations must be applied at every grid point. In most cases, reinitialization is by far the most time-consuming part of a level set deformation. Note however that not all graphics applications require highly accurate schemes in order to provide visually plausible results. For morphing, first order upwind methods are usually sufficient. Still, simulations do not run interactively except at relatively small resolutions or when implemented on the GPU [74, 125].
- Lagrangian marker particles (see next chapter) and/or high resolution combined with high order accurate FD schemes are usually required for maintaining sharp edges. This fact makes level sets computationally intensive.
- In computer graphics, meshes and sub-division surfaces are adaptive along the interface, whereas level sets are still represented uniformly.
- Frequent conversions to and from triangle meshes are often required, as triangle-based boundary representations are still the most common and widely supported. However, such conversions are not invertible with current methods.
- Level sets do not currently allow for interactive direct manipulations *e.g.* via the use of control vertices, edges and so on. On the contrary, explicit representations have this feature which makes them very well suited for interactive modeling.
- In contrast to explicit representations, level sets have no inherent dynamic parameterization. Parameterizations are useful for many different applications in computer graphics *e.g.* texturing.

2.6 Summary

This chapter briefly motivated the level set method for representing dynamic implicit surfaces in computer graphics. We also described the basic theory as well as numerical methods employed to solve the level set equations. In particular implicit surfaces were presented and their advantages argued to contrast explicit representations. Next the most fundamental level set equations were described followed by the introduction of finite difference methods and the concepts of numerical stability and numerical dissipation. Subsequently an exposition of the upwind and Godunov schemes for solving the most commonly occurring level set equations were given. The chapter concluded with an outline of the advantages and disadvantages of the level set method. The

three primary disadvantages, memory inefficiency, computational inefficiency and the restriction of simulations to a predefined static computational domain, largely set the stage for the next chapter which provides an overview of previous and concurrent work addressing some of these limitations.

Chapter 3

Overview of Level Set Methods, Algorithms and Grid Representations

Since its conception, the level set method [113] has attained widespread use due to its advantages such as ease of topological change and avoidance of self-intersections. However, mainly three issues limit the practical feasibility of the method as originally proposed:

- *Computational inefficiency*: The time complexity of the original level set method is $O(L^3)$ where L is the side-length of the grid. Ideally, when we are working in co-dimension one, the time complexity should instead be $O(A)$ where A is the area of the surface.
- *Storage inefficiency*: The storage requirements of the original level set method are $O(L^3)$ as opposed to the desired $O(A)$.
- *Predefined static computational domain*: Level set deformations are confined to stay within the boundaries of a predefined computational grid. Rather, a level set surface should be free of any boundaries and be able to move and expand anywhere in a semi-infinite computational domain.

These disadvantages were identified in chapter 1 and are targeted by the data structures and algorithms presented in this dissertation. However, other researchers have addressed the same problems either previous to or concurrently with our work. Below we briefly review all such relevant existing work. We will consistently refer to the area of the surface as A and to the side-length of the grid as L.

3.1 Narrow Band Level Set Methods

The limitation of computational inefficiency was addressed by the introduction of the narrow band methods that made the observation that it is only necessary to solve the level set PDE in close vicinity of the zero level set, *i.e.* the interface. The first narrow band method was proposed by Chopp [22] and later analyzed extensively by Adalsteinsson and Sethian [1]. In particular Adalsteinsson and Sethian [1] introduced a narrow band method which restricts most computations to a band of active grid points immediately surrounding the interface, thus reducing the time complexity to $O(A)$. In their examples the band radius is 12 grid points in order to sustain the same band for several iterations. The reason for this is that their narrow band rebuild procedure requires an $O(L^3)$ operation in which grid points over the entire volume are accessed

to rebuild the list of active grid points in the narrow band. Similarly the storage complexity for this narrow band method is still $O(L^3)$. Interestingly, Adalsteinsson and Sethian also proposed a grid resizing strategy as part of their narrow band rebuilding procedure.

The $O(L^3)$ time complexity was eliminated by the *sparse field level set method* of Whitaker [161]. In addition to storing the dense uniform grid, the sparse field level set method employs a set of linked lists to track a minimal set of active grid points around the interface. In a sense the sparse field method takes the narrow band concept to its extreme by identifying the minimal set of grid points required to resolve the position of the interface. The level set PDEs are subsequently only solved on this minimal set of grid points. In the original paper [161], distance is propagated to grid points neighboring the minimal set of grid points by means of an approximate city-block distance metric. The resulting narrow band is only as wide as the size of the finite difference stencils used on the interface grid points. Additionally, costly reinitializations are avoided by employing speed function extension [1] as well as division by the length of the gradient which preserves the approximate signed distance to the interface. Other more accurate first order redistancing approaches may be employed with the sparse field method as well, however it remains to investigated how the sparse field method can be extended to higher order accurate schemes. While consistently efficient in time, $O(L^3)$ storage space is required by the sparse field level set method.

Peng *et al.* [120] next introduced a narrow band method tailored for more accurate finite difference schemes. This method solves the level set PDE in a narrow band around the interface, with a band radius of typically three to five grid points depending on the accuracy of the numerical scheme employed. Subsequently, this solution is propagated out in a tube of grid points by means of an Euclidean distance metric. Their method employs data structures based on simple arrays as opposed to the linked lists used in [161]. However their method does not overcome the $O(L^3)$ storage requirements and the $O(L^3)$ periodic rebuild of the narrow band, although the possibility of using an $O(A)$ time rebuild is mentioned.

Recently an extension to the method of Peng *et al.* [120] was proposed by Nilsson *et al.* [108]. The extension is an $O(A)$ method for rebuilding the narrow band in a single pass.

All of these narrow band methods effectively address the problem of computational inefficiency present in the original level set formulation [113]. However, they all explicitly store a dense uniform grid and additional data structures to identify the narrow band grid points. Hence, the associated memory requirements remain $O(L^3)$. This can be a limiting factor for level set simulations that require large high resolution grids to resolve details of complex deforming interfaces.

3.2 Octree-Based Level Set Methods

Quadtrees (2D) and *Octrees* (3D) [27] have in recent years been applied to level sets [31, 34, 45, 83, 84, 94, 139–142] and adaptively sampled distance fields [39, 121] to reduce the storage and computational requirements compared to the original level set method.

While these tree data structures do indeed allow for multi-resolution representations, all current tree-based level set methods use uniform resolution near the interface. This is partly due to numerical accuracy but also because it can be hard to design reliable refinement strategies which guarantee that no fine features are missed due to under-sampling as the interface propagates in time. Refining uniformly near the interface and storing only the grid points inside the narrow

band in the octree enables the use of higher order finite difference schemes like ENO [114] and WENO [79] in space and the TVD Runge-Kutta methods [133] in time. On the other hand a non-uniform discretization makes it non-trivial to accurately employ these finite difference schemes. Tree based methods that solve the level set equation over the entire computational domain need to account for grid cells of varying sizes and are hence prevented from using these standard higher order finite difference schemes as is [94, 139–142]. Instead the semi-Lagrangian scheme [141] is typically employed.

The pointer-based quadtree and octree data structures reduce the storage requirements of level sets to $O((d+1)A)$, but also introduce an $O(d)$ access time, where d is the depth of the quadtree or octree. Note that it may be the case that $d >> \log A$. The octree data structure can be modified to reduce storage requirements to $O(A)$ and access time to $O(\log A)$ (see [27]). This access time is nevertheless still penalizing in the context of the level set method, and state-of-the-art octree-traversal and search methods [38, 138] utilize bit-arithmetic that cannot immediately be used in conjunction with these modifications.

The method of Losasso et al. [83, 84] proposed concurrently with the work presented in this thesis addresses some of the performance issues associated with octrees. Instead of using a traditional octree they propose to use a coarse uniform grid in which each grid cell stores an octree of its own. This decouples the depth of the octree from the size of the computational domain and hereby lowers the depth d. In addition they introduce an iterator construct that speeds up access locally during interpolation for semi-Lagrangian advection. Unfortunately a comparative study of the practical performance of this method has not been documented. Furthermore, no method has been published on how to ensure cache coherency in the octree storage format as the narrow band changes due to the temporal evolution of the level set.

In general a state-of-the-art octree based level set method storing and processing only grid points in a narrow band tends to perform worse than a narrow band level set method [48, 105] (see also chapter 7). This is to some degree due to the large number of accesses required in the finite difference computations.

Additionally, level set deformations on octrees, including the modification suggested by Losasso *et al.* are still restricted by the boundaries of the underlying grid, although the decrease in memory consumption due to the octree structure does allow for larger computational domains. It is possible to combine an octree approach with a moving and resizing grid strategy (reviewed below), but one should note that an octree approach will incur storage and/or computational overhead for large computational domains. A traditional octree will incur both storage and computational overhead since the number of levels in the tree increases as the computational domain grows. As the modification proposed by Losasso *et al.* decouples the number of octree levels from the overall size of the computational domain, the search depth is not increased as the computational domain is expanded. However, the coarse uniform grid in itself will eventually incur a storage overhead, in particular in situations where large computational domains are required.

3.3 Sparse Non-Tree-Based Level Set Representations

The Sparse Block Grid method of Bridson [15] divides the entire bounding volume of size L^3 into cubic blocks of P^3 grid points each. A coarse grid of size $(\frac{L}{P})^3$ then stores pointers only to those blocks that intersect the narrow band of the level set. Block allocation and de-allocation occurs as the surface propagates, and a custom reinitialization procedure is required when blocks are

allocated. Due to its two level hierarchy this method retains the constant access time inherent to dense grids. Bridson's approach allows for higher resolutions with $O(1)$ random access times, but has a storage complexity of $O((\frac{L}{P})^3 + A)$. As long as the storage requirements of the coarse grid, $\frac{L}{P})^3$, are $O(A)$, the overall storage requirements are $O(A)$ as desired. However for very high resolutions or simulations where interfaces with small area are spread out in a large domain, the storate-requirements may be dominated by $\frac{L}{P})^3$. While the method of sparse blocks restricts computations to a blocked narrow band, it employs a rather conservative estimate of the grid points actually needed[1] and will hence result in more computations than required unless grid points inside the narrow band are explicitly identified. No comparisons or performance evaluations of this method has been published.

An approach similar to Bridson's was taken for simulating water drops on surfaces in the recent [159].

Concurrently with and independently of our work, Houston *et al.* [50] presented the RLE Sparse Level Set in a technical sketch (one-page abstract). Their work is specifically tailored for computer graphics and primarily focuses on fluid simulations. They propose a data structure based on run-length encoding (RLE) which decouples the storage of the elements from the actual encoding.

In the past, run-length encoding has been successfully applied to volumes, *e.g.* Curless *et al.* [26]. While Bridson [15] mentions the potential value of applying RLE to level set volumes, Houston *et al.* [50] were the first to design and implement an RLE compressed level set data structure.

While their technical sketch is rather sparse on detail, we list the following characteristics based on the sketch and personal correspondence with the authors. In 3D, their approach requires $O(L^2 + A)$ storage. Hence their memory usage is not proportional to the interface, however random access may be relatively fast because a logarithmic search is only required on one level of the data structure. Sequential access time is $O(\frac{L^2}{A} + 1)$ per element whereas random and stencil access times are logarithmic in the number of runs, r, *i.e.* $O(\log r)$ in a single scan-line (see chapter 6). Consequently, advection and PDE based reinitialization time on the RLE Sparse Level Set is $O(A \log r)$ in contrast to the desired $O(A)$, since random access is employed for all grid points in the stencil. Their method maintains a dynamically resizing bounding box which allows the level set to grow dynamically. If the L^2 dependency in their storage requirements becomes dominant (large sparsely populated computational domains), their method does not allow for truly out-of-the-box level set simulations.

While their data structure has several resemblances with our initial work presented in chapter 5, our data structure requires $O(A)$ storage, is out-of-the-box and out-performs the Sparse RLE Level Set [48]. In chapter 6 we propose a combination of our initial work with the run-length encoding of the RLE Sparse Level Set for versatile representations of narrow bands useful in graphics.

3.4 Adaptive Level Set Methods

The idea of an *adaptive level set method* allowing for variable resolution along the surface is intriguing. Despite this fact, adaptive level set methods are yet to be explored in the context

[1] Only the grid points in the γ tube are required, and the blocked narrow band is clearly a conservative estimate of the γ tube.

of computer graphics. Since the methods presented in this dissertation enable level set simulations at very high resolutions, the main argument in favor of an adaptive level set method is its potential ability to decrease simulation time. To the best of our knowledge, two adaptive level set methods have been proposed [93, 144], both in the context of computational physics. The two methods rely on *Adaptive Mesh Refinement* techniques [7–9] that refine the grid in areas of interest by superimposing a number of dense uniform grids at increasing resolution. Spatially adaptive techniques add complexity because the adaptive grid must adjust to the characteristics of the evolving surface. In particular error analysis and refinement strategies must be implemented. Unfortunately, none of the papers on adaptive methods seem to demonstrate simulations at high resolutions that are infeasible with dense uniform grids.

Representing the surface adaptively also has implications for the speed functions and velocity fields defining the surface dynamics. In order to profit from the benefits of a fully adaptive method, these quantities will also have to be represented and simulated adaptively.

More recent work [147] explores the use of transformed uniform grids that cluster grid points close to the surface. Hence computational effort is concentrated where it is most needed and higher accuracy is obtainable with fewer grid points. However, again no high resolution simulations are reported.

3.5 The Particle Level Set Method

The particle level set method [33] combines a Lagrangian marker particle approach with the Eulerian level set method to obtain the advantages of both. The particle level set method reduces the amount of numerical dissipation particularly present at coarse resolutions. The marker particles ensure greater accuracy, especially in high-curvature and under-resolved regions of the surface[2] where the level set method alone would incorrectly merge characteristic curves in order to obtain the correct weak solution dictated by the entropy condition. On the other hand, the level set method ensures that the dynamic topology of the surface is correctly resolved. The above properties have made the particle level set method widespread when implementing fluid simulations for computer graphics. While perfectly suited for this kind of passive surface advection in an external velocity field free of shocks (as is the case with water simulations), its use beyond this application area however is limited. First of all, the particle level set method does not work if shocks are present in the velocity field. As illustrated in [33], the existence of a shock will cause particles to move inconsistently with the surface in the vicinity of the shock and hence ruin the desired weak solution. Similarly, the particle level set method is not suited, as is, to surface deformations described as motion in the normal direction. This makes the method infeasible in many graphics applications such as shape metamorphosis and geometric modeling that employ motion in the normal direction and cause shocks in the underlying velocity field when the models have corners and edges.

The particle level set method increases the memory requirements as up to 64 marker particles[3] are placed in each grid cell in a band three cells wide on each side of the interface. Particularly at lot of memory is required in cases where a fluid surface has a large area [106]. Unfortunately, the storage requirements of the particle level set method have not been documented in any published work.

Unlike the particle level set method, our data structures and algorithms are applicable to all

[2] Consequently it preserves volume.
[3] For each such particle, both position and radius needs to be stored.

level set equations and can hence be utilized for limiting numerical dissipation in a brute force way as they allow for an increase in resolution. The particle level set method does however have the advantage that for fluid simulation it does not require the fluid interior to be represented at higher resolution. For our methods, the resolution of the fluid interior needs to be increased when the resolution of the level set is increased. Fortunately our methods can be combined with the particle level set method which was actually done in figure 1.6 in chapter 1.

.

3.6 Moving and Resizing Grids

The level set data structures presented in this dissertation are out-of-the-box. As previously indicated this means that the level set deformation is not limited by the boundaries of an underlying grid and the memory consumption will be $O(A)$ no matter how large the computational domain is.

Related work, developed concurrently with ours, has also addressed the issue of an expanding computational domain, although in fundamentally different ways. To the best of our knowledge, Adalsteinsson and Sethian [1] were the first to describe a grid-resizing strategy as mentioned above. Later, Bridson [15] posed the idea of a dynamically expanding domain, but he did not demonstrate nor devise a concrete algorithm for it, although a combination of a hash table and his sparse block grid was indicated as a possible approach. Rasmussen *et al.* [122] proposed the concept of dynamically moving and resizing uniform grids. This method can handle the situation where the deforming surface in question to a large degree remains confined within a small volume but is free to resize within the storage limits and move anywhere in space. To alleviate the problems of the above strategy, a *grid sourcing* strategy was proposed in the same paper. The grid sourcing strategy splits the simulation domain into a number of uniform grids and arranges them into an acyclic graph whenever the size of a single uniform grid becomes infeasible. Each grid can then be simulated separately, either in parallel or in turn, by maintaining the appropriate boundary conditions between grids adjacent in the acyclic graph. However, the method assumes a *down flow i.e.* the surface deformation taking place in one grid can only be affected by the deformation in adjacent grids closer to the root of the graph. Hence this method is not applicable in general, but works well for the type of flow at which it was aimed. In particular the melting terminatrix shown in figure 1.1 in chapter 1 was simulated using this approach.

Similar approaches have been proposed in [132] and more recently [117]. [132] is particularly aimed at buoyancy driven flows such as smoke and [117] is an extension of [122] allowing for both splitting and merging of grids.

A limitation of the methods above is that the memory consumption of the individual dense uniform grids may be prohibitive even though the actual narrow band may only occupy a small portion of the total volume.

Finally, the Sparse RLE Level Set [50] also allows for a dynamically resizing bounding box. However, since the memory consumption of this grid is not proportional to the surface area, the method's out-of-the-box capabilities may be limited. In particular excessive memory may be required in the case of deforming objects sparsely populating a large volume.

In comparison our method is out-of-the-box as storage requirements scale with the area of the surface, and it does not require the simulation domain to be split.

3.7 GPU Based Level Sets

In [74] Lefohn *et al.* demonstrated a narrow band level set method implemented on graphics hardware. In particular the volume is partitioned into blocks (similarly to the Sparse Block Grid of Bridson [15]) intersecting the level set surface. Each of these non-empty blocks is then processed on the GPU. Using their method they obtain speedups of ten to fifteen times over the fast Sparse Field Method of Whitaker [161] which brings the updates of the level set method closer to interactive rates. On the CPU side, the method requires the storage of the dense uniform as well as the blocked grid and additional data structures. On the GPU side, only the blocked representation is stored.

3.8 Summary

In this chapter we covered the previous and concurrent work relevant to the novel techniques proposed in this dissertation. In particular three issues limited the practical feasibility of the original level set method: Storage inefficiency, computational inefficiency and the restriction that simulations must take place within the boundaries of a fixed computational domain. We discussed the narrow band level set methods that reduce the computational efficiency of the level set method, the octree, sparse block grid and RLE methods that address the storage inefficiencies and the moving and resizing grid strategies that target the restriction of a static computational domain. Additionally the particle level set method, adaptive level set methods and GPU based level sets were covered.

Part II

Data Structures and Algorithms for High Resolution Level Set Simulations

Chapter 4

Introduction

Part I of this dissertation motivated and introduced the level set method in the context of computer graphics. In particular the importance of level set surfaces for fluid representations, geometric modeling and shape metamorphosis were emphasized. However, as also identified in part I, a number of issues limit the practical applicability of level sets. Our work presented in part II targets the *storage inefficiency*, *computational inefficiency* as well as the restriction that *deformations must take place within a static rectangular computational domain.* In particular we propose new data structures and algorithms for representing and manipulating high resolution level sets. The first of two data structures proposed in part II is entitled the Dynamic Tubular Grid, or DT-Grid [105]. The DT-Grid allows for high resolution level set simulations with lower memory footprints and in general higher computational efficiency than previous narrow band, octree and RLE methods. We demonstrate the properties of the DT-Grid by performance evaluations and by showing a high resolution level set simulation at effective resolution 1024^3. Additionally we illustrate an example of an out-of-the-box simulation not suitable for representation on traditional octrees and dense uniform grids. The second data structure we propose is named the Hierarchical Run Length Encoded, or H-RLE, grid [48]. The H-RLE is derived from a combination of the DT-Grid and the Sparse RLE Level Set of Houston *et al.* [50]. It allows for a more flexible encoding of the level set, but results in a slight degradation in performance for level set computations when compared to the DT-Grid. We stress that the H-RLE and DT-Grid complement each other. In particular the DT-Grid is preferable for standard high resolution level set simulations, whereas the H-RLE offers advantages over the DT-Grid when flexible encodings of the narrow band are required. Finally we note that chapter 15 in part V presents several applications of the DT-Grid and H-RLE data structures in computer graphics. This includes fluid simulation, geometric texturing and surface reconstruction.

Briefly outlined, the structure of part II is as follows. Chapter 5 presents the DT-Grid and its associated algorithms for representing, manipulating and tracking high resolution level sets. Next chapter 6 introduces the H-RLE data structure and discusses its versatility. Following that chapter 7 provides a detailed performance analysis that includes comparisons with previous and concurrent work. In addition we discuss the known advantages and disadvantages of our new data structures.

Chapter 5

The DT-Grid - High Resolution Level Set Simulations

Level set methods [113] have proven very successful for interface tracking in many different areas of computational science. However, current level set methods are limited by a poor balance between computational efficiency and storage requirements. Tree-based methods decrease the storage requirements but have relatively slow access times when utilized for level set computations. Narrow band schemes on the other hand are relatively fast, but lead to increased memory requirements that become prohibitive for very high resolution interfaces.

Our general approach to addressing these limitations is to introduce a dynamic uniform grid that is only defined in a tubular region around the propagating interface. We dub this time-dependent and interface adapting grid the "Dynamic Tubular Grid" or **DT-Grid** [12,103–105]. In contrast to existing narrow band methods we do not store *any* information outside of this dynamic tube. As a result, the computational complexity and storage requirements scale with the size of the interface. Our novel level set data structure and algorithms are relatively fast, cache coherent and allow for a relatively low memory footprint when representing high resolution level sets. Our studies show that our 3D DT-Grid is in most cases faster and more memory efficient than both state-of-the-art narrow band [108,120], octree [38,138] and RLE implementations [50]. Additionally our data structure can readily be used with the finite difference schemes already developed for dense uniform grids. Finally, our data structure is free of any boundary restrictions on the interface expansion which leads to what we call "out-of-the-box" level set simulations.

In this chapter we show two examples illustrating features of the DT-Grid. One is a high resolution surface deformation and the other illustrates DT-Grid's capability of handling out-of-the-box simulations. The high resolution simulation presented in this chapter is the *Enright Test* at effective resolution 1024^3. The original paper introducing the Enright Test ran it at an effective resolution of 100^3 [33]. More recently, it was presented in effective resolution 512^3 on an octree structure by Enright *et al.* [34].

5.1 Contributions

DT-Grid stands apart from previously published work in several ways. We do not use tree structures or dense uniform grids with additional data structures to represent the narrow band. DT-Grid takes a different approach by storing the narrow band in a compact non-tree-based data structure that uses less memory than previous methods without compromising the computational efficiency. The DT-Grid data structure is inspired by sparse matrix formats such as

the compressed column-storage format [41], and leverages on the fundamental assumption made by the narrow band methods that it is only necessary to solve the level set equation in close vicinity of the interface. Since narrow bands generally exhibit a higher degree of connectivity than sparse matrices and support for relatively fast random access is required, our approach for storing the topology of the narrow band deviates from compressed column-storage. Also fundamentally different from sparse matrices, the DT-Grid is defined recursively by DT-Grids of lower dimensionality. This is an essential property which, as we will see, makes DT-Grid free of any boundaries and makes it generalize to any number of dimensions.

Below we summarize our contributions:

- The memory usage of DT-Grid is proportional to the size of the interface. More specifically the storage requirements are $O(M_N)$ (in 3D $O(M_3)$) where M_N is the number of grid points in the N-dimensional narrow band. In fact our evaluations show DT-Grid to be more compact than other grid or tree-based level set schemes that employ a uniform sampling of the interface. As a result, our data structure allows for higher resolutions of level sets before hardware memory restrictions are potentially violated.
- Our evaluations have shown that the computational efficiency of accurate level set deformations based on DT-Grid is in most cases better than both narrow band, tree- and RLE-based approaches. As we demonstrate in chapter 7 this is partly due to the combined effect of a relatively low memory footprint and the cache coherency of the storage format and associated algorithms. More specifically we have developed algorithms that guarantee the following properties of DT-Grid:
    1. Access to grid points has time complexity $O(1)$, when the grid is accessed sequentially.
    2. Access to neighboring grid points within finite difference stencils has time complexity $O(1)$.
    3. The time complexity of random and neighbor access to grid points outside the finite difference stencil is logarithmic in the number of connected components within p-columns (as opposed to the number of elements within p-columns) (see section 5.2.1 for the definition of p-column). This time complexity is in many cases asymptotically better than random and neighbor access time in octrees.
    4. The time complexity of constructing and rebuilding the DT-Grid is linear in the number of grid points in the narrow band.
- Our data structure allows level set interfaces to freely deform without boundary restrictions imposed by underlying grids or trees employed in other methods. This effectively implies that interfaces can expand arbitrarily. We demonstrate this with *out-of-the-box* level set deformations.
- Our data structure generalizes to any number of dimensions.
- Unlike approaches employing non-uniform grids our flexible data structure can transparently be integrated with all existing finite difference schemes typically used to numerically solve both hyperbolic and parabolic level set equations on dense uniform grids.
- The DT-Grid can be used to store the usually non-convex volume of fluid velocity and pressure used in fluid simulations. This makes the memory usage scale with the volume of the fluid as opposed to the volume of an enclosing bounding box.

- The DT-Grid algorithms are relatively straightforward to parallelize [52].

This chapter is organized as follows. Section 5.2 introduces the DT-Grid data structure. A general N-dimensional definition is given and a detailed explanation is presented in 2D. In section 5.3 we describe algorithms that are fundamental to level set simulations on the DT-grid. Next in section 5.6 we demonstrate the relatively low memory footprint of a 1024^3 resolution level set simulation, and then show how level set simulations on a DT-Grid can go out-of-the-box, a feature not shared with any existing narrow band or tree-based level set method except the H-RLE data structure presented in chapter 6. Finally, section 5.7 summarizes this chapter.

5.2 DT-Grid Data Structure

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

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

5.2.1 Definition of the DT-Grid

The main concept behind the definition of the DT-Grid is to store a minimal amount of information in order to describe the topology and values of a dynamic tubular grid. Exactly how this information can be utilized for constructing algorithms will become evident in section 5.3. The DT-Grid is defined recursively in terms of DT-Grids of lower dimensionality, and as such our approach readily generalizes to any dimension. However, for simplicity we shall limit a detailed description of the data structure to 2D and illustrate with the example depicted in Figure 5.1. As a prelude to this description it is convenient to introduce the following general terminology

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

Table 5.1: Nomenclature used throughout the chapter.

based on the nomenclature given in table 5.1.

- In N-dimensions, *p-column* (short for projection column) number $\mathbf{X}_{N-1} = (x_1, x_2, ..., x_{N-1})$ is defined as the set of grid points in the tubular grid that project to $(x_1, x_2, ..., x_{N-1}, 0)$ by

orthogonal projection onto the subspace spanned by the first $N-1$ coordinate directions. Thus a p-column is always 1D.

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

For example, in 2D, p-column number x is defined as the set of grid points in the tubular grid that project to $(x, 0)$ by orthogonal projection onto the X axis. In figure 5.1.a, p-column number 3 is defined as the set of grid points $\{(3,1),(3,2),(3,4),(3,5)\}$, and it contains two connected components, $\{(3,1),(3,2)\}$ and $\{(3,4),(3,5)\}$. Note that the lower leftmost grid point in figure 5.1.a is $(0,0)$.

A N-dimensional DT-Grid can be defined recursively in terms of a (N-1)-dimensional DT-Grid using pseudo C++ syntax as follows

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

Below we define the 2D DT-Grid in pseudo C++ syntax and explain its constituents in detail. In particular it is defined as

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

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

where the individual constituents are:

value: The `value` array (in `DTGrid2D`) stores the numerical values of all grid points in the two-dimensional tubular grid in (x, y) *lexicographic* order. Typically the associated `Type` will be `float` or `double`. In figure 5.1.{a,b} the grid points contained in the tubular grid are colored yellow and blue. In this illustrative example the numerical values of the grid points in the tubular grid are simply chosen to be the corresponding lexicographic storage order in the DT-Grid.

yCoord: The `yCoord` array stores the min and max y-coordinate of each connected component. In figure 5.1.{a,b} these grid points are shown in yellow. Thus, rather than simply storing y-coordinates of all grid points, we exploit the connectivity in the tubular grid.

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

proj1D: The `proj1D` constituent holds pairs of indices into the `value` and `yCoord` arrays for the first grid point in each p-column in the tubular grid. This is illustrated with arrows in figure 5.1.{b,c}. Also note that `proj1D` is defined recursively as a `DTGrid1D` with `Type=IndexPair`, see

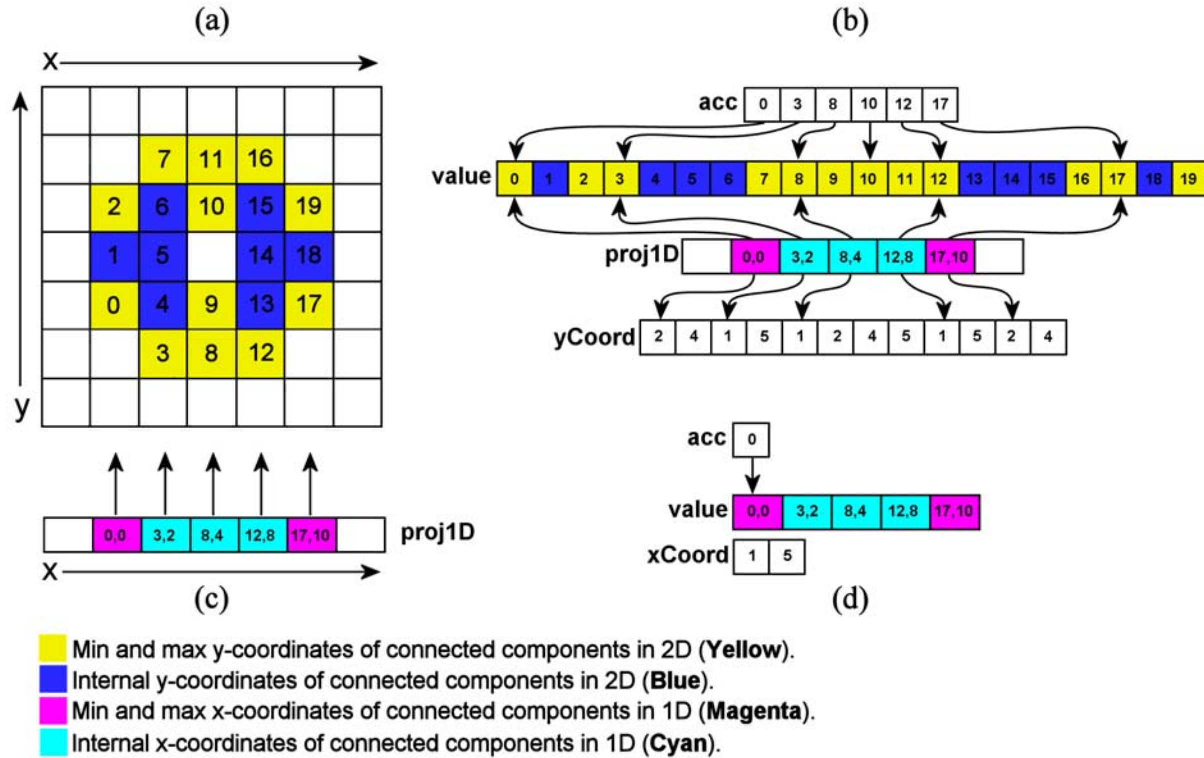


Figure 5.1: **(a)** A uniform 2D grid with the grid points in the narrow band depicted in yellow and blue. **(b)** 2D DT-Grid corresponding to (a). Note the lexicographic storage order. **(c)** A uniform 1D grid, corresponding to the projection of the 2D uniform grid onto the X axis. **(d)** 1D DT-Grid corresponding to (c). This 1D DT-Grid forms part of the 2D DT-Grid as depicted in (b) and explained in detail in the text.

figure 5.1.{c,d}. By defining `proj1D` recursively as a DT-Grid we obtain the property that the grid becomes free of any boundaries, since grid points are stored explicitly albeit compressed. This definition also leads to the property that memory requirements are proportional to the size of the interface as we will prove later.

The constituents of the 1D DT-Grid are defined similarly to those of the 2D DT-Grid, except for the fact that it does not have a `proj0D` constituent. `proj1D` introduces additional structure into the 2D DT-Grid and allows for fast access to each p-column independently. As will become clear in the next section, this structure is used extensively in most of the algorithms of the data structure.

Figure 5.1 shows the definition of the DT-Grid in its most general and comprehensive form. However, a number of space optimizing variations, valid at each recursive level of the DT-Grid, can be constructed, and we briefly review these next. Note that these variations require either none or at most trivial modifications to the algorithms of the DT-Grid described in the subsequent sections. In particular the space optimizing variations are:

1. Due to the presence of the `acc` array an optimization is possible. This optimization comes at the trade off of a single extra array lookup. By looking at figure 5.1 the reader should

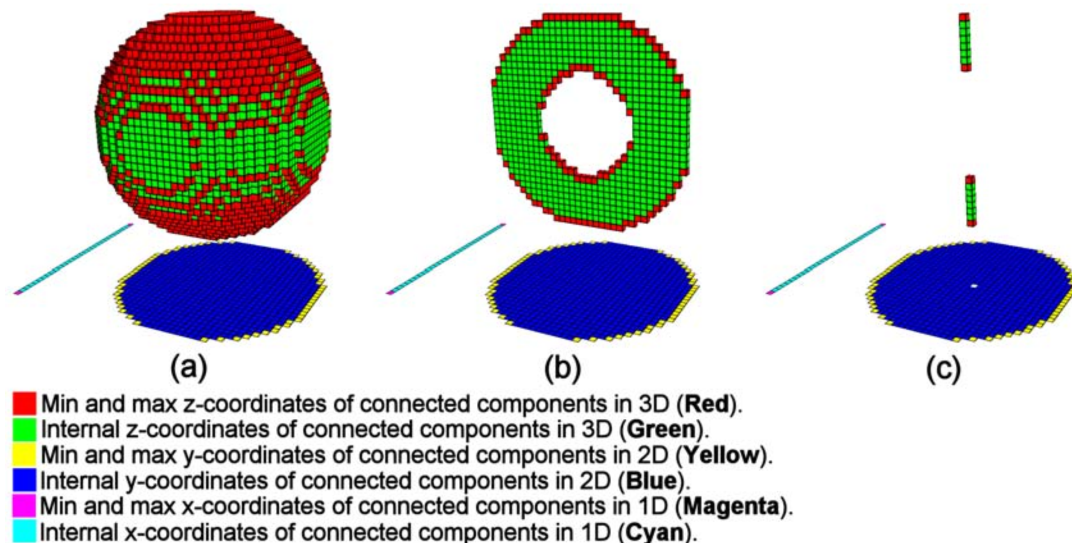


Figure 5.2: Color coded representation of the tubular grid of a 64^3 sphere in a 3D DT-Grid. a) Entire sphere. b) Middle slice of sphere. c) P-column consisting of two connected components.

notice the redundancy apparent between the `acc` array and `proj1D` structure regarding indices that point into the `value` array. In fact `proj1D` needs only store the index into the `yCoord` (in general `nCoord`) array. The corresponding index into the `value` array can then be obtained by looking it up in the `acc` array using the index into the `yCoord` array divided by two [1].

2. Another optimization is possible when storing the `acc` array. In fact storing the max coordinates of connected components becomes redundant. This is because the difference between two subsequent entries in the `acc` array equals the number of grid points in the corresponding connected component. Thus, the max coordinate of a given connected component can be found as the min coordinate plus the difference between the two corresponding entries in the `acc` array minus one.

3. The `acc` array is only required for the above optimizations and when random access based on binary search is used. However, for many typical models, the number of connected components in a DT-Grid is very small. This means that linear search time is often comparable to or faster than binary search in practice, despite its asymptotically inferior time complexity. Hence if either random access is not required or linear search performs better or comparable in the given situation, the `acc` array does not need to be stored at all.

4. It is only necessary to represent coordinates stored in *e.g.* the `yCoord` array with the number of bits required by the simulation. If for example the coordinates of narrow band grid points for a particular simulation are confined to stay within the range $[-2048; 2047]^3$, only 12 bits are required to store each coordinate.

[1] The reason for the division by two is that the **nCoord** array stores exactly twice as many elements as the **acc** array.

Depending on the particular application of the data structure, it can either be implemented in its most general form or using one or several of the space optimizations above.

As clarified above, the DT-Grid generalizes to any number of dimensions. In particular, figure 5.2 shows an example of a 64^3 sphere represented in a 3D DT-Grid. The 2D and 1D DT-Grid constituents are also included in the illustrations. The red and green grid points are the grid points in the 3D tubular grid. The red grid points are the start and end grid points of connected components in the z-direction. Figure 5.2.c shows p-column number $(25, 25)$ consisting of two connected components. The white pixel in Figure 5.2.c illustrates the `IndexPair` that points to p-column number $(25, 25)$.

The storage requirements of a N-dimensional DT-Grid are $O(M_N)$ (see table 5.1) which can be justified as follows: Clearly, the storage requirements of a 1D DT-Grid are $O(M_1)$ since it does not contain a `proj0D` constituent, and since the number of `xCoord` indices stored in the worst case equals twice the number of grid points in the 1D DT-Grid. The storage requirements of a N-dimensional DT-Grid are $O(M_{N-1} + M_N)$ which by induction equals $O(M_N)$ since $M_{N-1} \leq M_N$ by the properties of orthogonal projection.

One additional and important property of the DT-Grid can be deduced from the definition given above. Since the DT-Grid is defined recursively, the coordinate vectors of all grid points are explicitly stored, albeit in a compressed format. This means that the grid points of the DT-Grid are not restricted to a particular range of indices as is the case with the traditional dense uniform grid or tree-based methods. Hence, the DT-Grid is capable of representing semi-unbounded, dynamically expanding and non-convex grids. This allows for out-of-the-box level set simulations which we demonstrate in section 5.6.1.

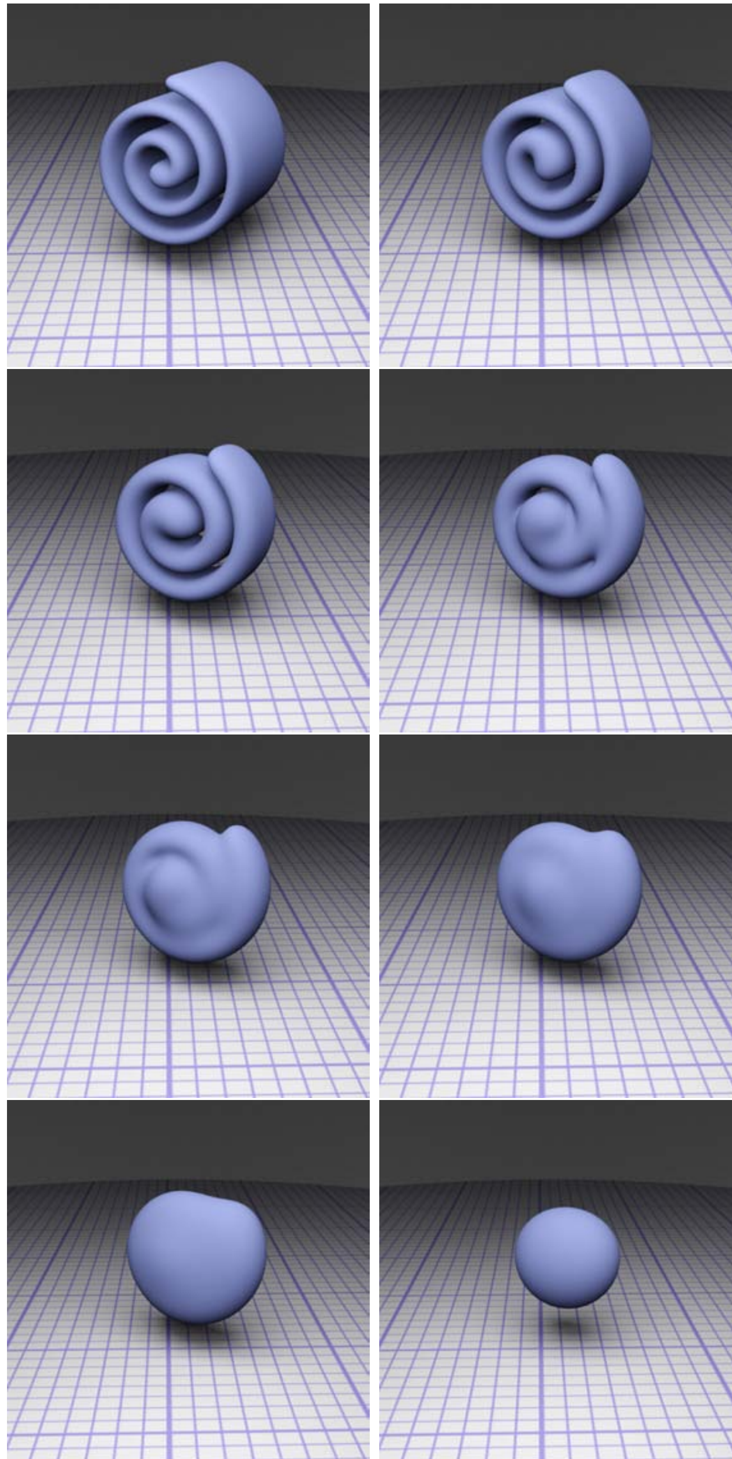


Figure 5.3: The DT-Grid dynamically adjusts the tube of grid points as the level set deforms. Hence the DT-Grid does not in any way limit or put restrictions on the type of surface deformation possible. Here an extruded spiral evolving under volume conserving mean curvature flow [120] and exhibiting complex changes in topology during the simulation. The evolution of the interface is depicted from top-left to bottom-right. Renderings by Ola Nilsson based on my simulation data.

5.3 DT-Grid Algorithms

Algorithm	**Time Complexity**
Push	$O(1)$
Access to Stencil Grid Points	$O(1)$
Sequential Access	$O(1)$
Random Access to $\mathbf{X}_N$	$O(1 + \sum_{n=0}^{N-1} \log C_{\mathbf{X}_n})$
Neighbor Access to $\mathbf{X}_N$ in m'th Coordinate Direction ($m = 1$: x-direction, $m = 2$: y-direction,...)	$O(1 + \sum_{n=m}^{N-1} \log C_{\mathbf{X}_n})$
Rebuilding the tubular grid	$O(M_N)$
Dilating the tubular grid	$O(M_N)$
CSG operation	$O(M_N + Q_N)$

Table 5.2: Key algorithms of a N-dimensional DT-Grid.

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

The key feature of the DT-Grid algorithms is that they allow us to solve the level set and reinitialization PDEs in time $O(M_N)$ due to the fact that constant time access to all grid points within the stencil is provided. This is the same asymptotic time complexity that narrow band level set methods have. Note that in comparison level set propagation on a state-of-the-art octree implementation relying on random and neighbor access has an asymptotic time complexity of $O(M_N \log M_N)$. The fast marching method can be implemented on the DT-Grid with asymptotic time complexity $O(M_N \log M_N)$ which again is the same as for a narrow band level set method on a dense uniform grid. Finally, rebuilding the narrow band has asymptotic time complexity $O(M_N)$ on the DT-Grid similar to the narrow band level set method.

Generally, narrow band level set methods are considered to be fast, and the above means that using the DT-Grid we can obtain asymptotic time complexities that match those of the narrow band level set methods and are better than those of the octree. Furthermore, the DT-Grid data structure is very compact and the associated algorithms to a large degree work cache coherently. The DT-Grid was initially designed to be more memory-efficient than previous approaches, but the last fact has the consequence that it is in most cases also faster. Measurements of L1 and L2 cache-hits and -misses presented in chapter 7 have shown that other approaches introduce more cache-misses than the DT-Grid. A memory access that results in a cache-hit can be orders of magnitude faster than a memory access that introduces a cache-miss. This explains why the DT-Grid can be faster than previous approaches even though its operations are more complex.

For a detailed evaluation of the DT-Grid against previous approaches the reader is referred to chapter 7.

Due to the recursive nature of the storage format of the DT-Grid, many of the operations presented here are also recursive in nature. The rest of this section is structured as follows.

Section 5.3.1 describes a constant time operation for inserting grid points into the DT-Grid. Section 5.3.2 describes a logarithmic time algorithm for random access to grid points based on binary search. This algorithm is used if grid points are accessed non-sequentially or lie outside of the stencil. Next, section 5.3.3 describes how constant and logarithmic time neighbor access operations can be constructed. In sections 5.3.4 and 5.3.5 we describe how constant time sequential access to all grid points within a finite difference stencil can be obtained when iterating over the grid. This is essential in obtaining a fast data structure. Finally, section 5.3.6 describes how the tubular grid is rebuilt. In particular, we describe a generic algorithm for rebuilding the tubular grid, which can be used independently of the method employed for reinitializing the level set function to a signed distance function.

Many of the essential details of the DT-Grid algorithms as well as more elaborate proofs of the time complexities are given in independent subsections following the more intuitive descriptions. We realize that not all readers are interested in these details, and as such effort has been made to make the main part of this chapter self-contained even if the detailed subsections are skipped on a first reading.

Table 5.2 gives an overview of the DT-Grid operations and their associated time complexities.

5.3.1 Push - Inserting Grid Points in Constant Time

The DT-Grid supports a low-level constant time `push` operation to add new grid points to the data structure. Since the grid points are stored in memory lexicographically as $(x_1, x_2, ..., x_N)$, new grid points must be pushed in this order[2]. If, on the other hand, grid points were inserted in an order different from their lexicographic order, each insertion would take worst case linear time in the number of grid points stored [3]. However, random insertions can be avoided altogether in level set computations.

A `pop` operation could be implemented similarly to the `push` operation, but is not needed for level set simulations. This is due to the fact that the structure of the tubular grid only changes when the tubular grid is rebuilt. In that case the new tubular grid is constructed from scratch using a fast dilation algorithm (see section 5.3.6).

The `push` method updates the array constituents of the DT-Grid (defined in section 5.2.1) and has to deal with the following three cases:

1. The new grid point is the first in a p-column. (As an example see grid points {0,3,8,12,17} in figure 5.1.a.)
2. The new grid point is the first grid point in a connected component (and not the first in a p-column). (See grid point 10 in figure 5.1.a.)
3. The new grid point is the last in an existing connected component at insertion time. (See the remaining colored grid points in figure 5.1.a: {2,7,9,11,16,19}.)

Below we give the full detail of the `push` operation.

Details of the Push Algorithm

Here we present a C++ pseudo code representation of the `push` operation for a 3D DT-Grid. The general N dimensional version is similar.

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

[3] Sorting all grid points can be done in worst case time $O(M_N \log M_N)$ and in linear $O(M_N)$ time if bucket sorting is applicable.

```
void push(Index x, Index y, Index z, Real val)
{
  if ( proj2D.last() != (x,y) )          // (x,y,z) lies in new p-column
  {
    IndexPair p(value.size(), zCoord.size());
    proj2D.push(x, y, p);
    zCoord.push(z);
    zCoord.push(z);
    acc.push(value.size());
  }
  else if ( zCoord.last() != z-1 )       // (x,y,z) first in connected component
  {
    zCoord.push(z);
    zCoord.push(z);
    acc.push(value.size());
  }
  else                                   // (x,y,z) last in connected component
  {
    zCoord.last() = z;
  }
  value.push(val);
}
```

Remarks: In **case 1**, `(x,y,z)` is the first grid point in the `(x,y)`'th p-column and the `proj2D` data structure must be set to point to this grid point. The `z` coordinate is pushed twice onto the `zCoord` array since it denotes both the start and end of a new connected component. This is also reflected in **case 2**. In **case 3** the end of the connected component is set to `z` since `(x,y,z)` was adjacent to the previous, `zCoord.last()`, grid point pushed. All operations above are $O(1)$, hence the `push` operation is $O(1)$, when considering N to be a constant.

5.3.2 Logarithmic Time Random Access

As described in the introduction, the DT-Grid supports constant time access to grid points within a stencil when accessing the grid sequentially (see section 5.3.4). This is used in most level set algorithms that we have considered, except for the fast marching method [131, 154, 155] and ray tracing, which instead uses random and neighbor access (neighbor access is described in the next section). The DT-Grid supports operations for random access, which is the mapping from an arbitrary N-dimensional grid point to its corresponding numerical value. A dense uniform grid provides constant time random access to all its grid points since this simply amounts to an array access. Constant time random access to grid points is not possible in a DT-Grid. However, logarithmic time, in the number of connected components within p-columns, can be obtained. Note that this is optimal with respect to the storage format since searching in a sorted range has a logarithmic lower bound.

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

1. The random access algorithm of the N-1-dimensional DT-Grid constituent is used to determine if p-column number $\mathbf{X}_{N-1}$ is contained in the projection of the N-dimensional tubular grid.

2. If this is the case, it is determined if the grid point's N'th coordinate, x_N, lies between the min and max N'th coordinates in p-column number $\mathbf{X}_{N-1}$.

3. If this is the case, binary search for x_N in p-column number $\mathbf{X}_{N-1}$ in the `nCoord` array is employed to find the nearest start coordinate of a connected component in the N'th

coordinate direction.

4. Finally it is determined whether the grid point actually exists in the p-column (ie. is inside a connected component in this p-column) and if this is the case its value is returned.

Access to grid points outside the tubular grid simply return $-\gamma$ if the grid point lies in Ω^- and γ if the grid point lies in Ω^+. The time complexity of random access to a grid point, $\mathbf{X}_N = (x_1, x_2, ..., x_N)$, is $O(1 + \sum_{n=0}^{N-1} \log C_{\mathbf{X}_n})$, where $C_{\mathbf{X}_n}$ is the number of connected components in p-column $\mathbf{X}_n$. We finally note that in cases where the number of connected components within p-columns is small it may actually be advantageous to employ a linear search instead of the asymptotically optimal binary search, see chapter 7.

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

In cases where a large range of random accesses into a DT-Grid fall outside of the narrow band it may be an advantage to add an explicit bounding box to the DT-Grid data structure. This will enable fast, $O(1)$, lookups away from the narrow band by just checking coordinates against this bounding box. A dynamic bounding box tracking the narrow band can be computed trivially and with almost no additional cost during the DT-Grid's dilation algorithm described in section 5.3.6.

Details of the Random Access Algorithm

Let γ be the width of the tubular grid and $\mathbf{X}_N = (\mathbf{X}_{N-1}, x_N)$ an N dimensional grid point with $N-1$ and 1 dimensional sub-components $\mathbf{X}_{N-1}$ and x_N respectively. The method `randomAccess` returns a triple `(inside, i, v)`. `inside` is a boolean telling whether $\mathbf{X}_N$ is inside the tubular grid, `i` is the index of $\mathbf{X}_N$ into the `value` array of `DTGridND` (which is valid only if `inside==true`) and `v` is the value at $\mathbf{X}_N$. Below we assume that the `IndexPair` described in section 5.2.1 has two members, `iv` that points into the `value` array and `ic` that points into the `nCoord` array. Note that the numbering of the individual steps in the detailed algorithm description below follows the numbering of the more intuitive description above, and that clarifying remarks are given immediately below the algorithmic description. The algorithm proceeds as follows:

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

2. If $x_N <$ `nCoord[`k^{min}`]` `||` $x_N >$ `nCoord[`k^{max}`]` return `(false,0,`γ`)`.

3. Perform a binary search for x_N at *even* positions (indices that point to the start of a connected component) in the `nCoord` array. The search is delimited by the indices k^{min} and $k^{max} - 1$ (both inclusive), and the index determined is denoted `k`. We assume the binary search is constructed such that `nCoord[k]` $\leq x_N <$ `nCoord[k+2]`.

4. There are now two cases:

   (a) If x_N <= `nCoord[k+1]`, set `i=acc[(k>>1)]+`x_N`-nCoord[k]` and return `(true,i,value[i])`.

   (b) Else return `(false,0,sign(value[acc[(k+2)>>1]])`$\cdot\gamma$`)`.

Remarks: **Step 1** determines k^{min} and k^{max} which are the indices into the `nCoord` array of the minimum and maximum N'th coordinate in p-column number $(x_1, x_2, ..., x_{N-1})$. In **step 4**, the `>>` is the right-shift operator. `k` is right-shifted by one since `k` is an index into the `nCoord` array which has exactly twice as many elements as the `acc` array. Note that for simplifying the presentation of the algorithm, a value was returned at all recursive levels of the data structure. In practice however this is only done at level N.

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

As noted previously it may sometimes be advantageous to apply a linear search in place of a binary search. Despite its asymptotically inferior time complexity of $O(\sum_{n=0}^{N-1} C_{\mathbf{X}_n})$ it will in practice often perform comparable to or even better than binary search. This is due to the relatively low number of connected components in typical models as well as the greater overhead associated with binary search. The only algorithmic change is in step 3 above where a linear search is employed instead of a binary search.

The time complexity of random access depends only on the number of connected components within a total of N p-columns in the DT-Grid - one p-column at each level of the encoding. Hence the time complexity of the random access operation is not affected if a model projects to a large lower-dimensional area. In the worst case, the number of connected components will be equal to the number of grid points in the grid and in that case, the random access operation will be logarithmic in the total number of grid points stored in the grid.

5.3.3 Logarithmic and Constant Time Neighbor Access

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

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

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

As mentioned above, neighbor access using locators is faster than neighbor access using random access. In fact, when doing neighbor search in the m'th coordinate direction, the structural information about the original and neighboring grid point is identical in the first *m-1* coordinate directions. For the sake of simplicity we explain this in 2D, but stress that the general N-dimensional case is similar. Recall that the storage order of grid points in a 2D DT-Grid follows the (x, y) lexicographic ordering. Thus, the numerical values of the neighbors in the Y coordinate direction, $(x, y-1)$ and $(x, y+1)$, can be found in *constant time* from a Locator using the indices `iv`± 1, respectively. If the particular neighbor does not exist in the tubular grid, γ is returned if the neighbor is outside the interface, and $-\gamma$ otherwise.

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

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

5.3.4 Constant Time Sequential Access Using Iterators

The DT-Grid has support for an *Iterator* which is a construct that provides constant time sequential access to grid points in the DT-Grid. The Iterator is in effect a wrapper around a Locator (see section 5.3.3) that uniquely identifies a grid point. Note that using the Locator constituent it is possible to obtain logarithmic, and in a single case constant, time access to neighboring grid points. However, as will be described in the next section, the *Stencil Iterator* provides constant time access to neighboring grid points within a stencil.

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

Details of the Increment Algorithm

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

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

Furthermore we assume that the `IndexPair` described in section 5.2.1 has two members, `iv` that points into the `value` array and `ic` that points into the `nCoord` array. Using the definition of the `LocatorND` presented in section 5.3.3, the `increment` operation looks as follows in C++ pseudo code

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

  value = grid.value[loc.iv];

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

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

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

5.3.5 Constant Time Stencil Access Using Iterators

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

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

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

1. First the Iterator corresponding to the center grid point of the stencil is incremented using the `increment` method described in the previous section. This center Iterator dictates the movement of the entire stencil.

2. Next the remaining Iterators, corresponding to non-center stencil grid points, are incremented until they point to the correct stencil grid point. This is done using the `incrementUntil` method which is described in detail in the next subsection. A non-center stencil grid point may not exist in the tubular grid. If this is the case, access to that particular stencil grid point returns $-\gamma$ if the grid point lies in Ω^- and γ otherwise.

Narrow band level set algorithms typically operate on a number of concentric tubes of increasing width centered about the interface, see [1,120] as well as chapters 2 and 3. If the DT-Grid is a signed distance field, the Stencil Iterator can be parameterized to return only the grid points

within a certain tube, e.g. the zero crossing, without requiring additional storage (see figure 2.7). This is done simply by incrementing the Iterator of the stencil center grid point until it points to a grid point with absolute value below some threshold.

Incrementing a stencil of Iterators across the DT-Grid provides constant time access on average to all stencil grid points in a particular tube as long as all grid points in the tube are visited. This is the case since: a) Each Iterator of the Stencil Iterator passes over the tubular grid exactly once, which has complexity $O(M_N)$, where M_N is the number of grid points in the tubular grid, b) M_N is proportional to the number of grid points in any tube centered about the interface (because tube has fixed width) and inside the tubular grid, c) The number of grid points within the stencil is a constant, d) Access to a grid point through an Iterator has time complexity $O(1)$.

Details of the IncrementUntil Algorithm

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

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

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

- If the center grid point of the stencil only moves one grid point in the N'th coordinate direction by an `increment` operation, we know that those non-center stencil grid points that will not pass out of the tubular grid, will also move exactly one grid point in the N'th coordinate direction. In that case **step 1** and **step 2** above can be implemented with the following three lines

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

  This optimization can typically be applied if the stencil is guaranteed never to pass out of the tubular grid when iterating over a certain tube, or similarly for those grid points of the stencil that are guaranteed not to pass out of the tube at a given grid point.

- If the center grid point of the stencil moves only one grid point in the N'th coordinate direction, but we are not sure if a particular non-center stencil grid point will move outside the tube or not, slightly more processing is required. However, the iterator corresponding to the non-center stencil grid point needs to be incremented at most once. In the case where it already points to the next correct non-center grid point (because it moved out of the tubular grid in a previous iteration) no increments are of course needed.

Note that the optimizations described above can be applied recursively to DT-Grids of lower dimensionality. Since we assume that the entire tubular grid is visited, all steps above take time $O(1)$ on average given the arguments in section 5.3.5, and hence access to grid points within the stencil is $O(1)$ on average.

5.3.6 Dilating the Tubular Grid in Linear Time

Level set methods typically apply a reinitialization procedure (after the advection/propagation step) to reset the level set function to a signed distance function (see chapter 2). Existing narrow band level set methods furthermore combine this reinitialization step with a method to rebuild the narrow band to ensure that it includes all grid points within a tube of a certain width.

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

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

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

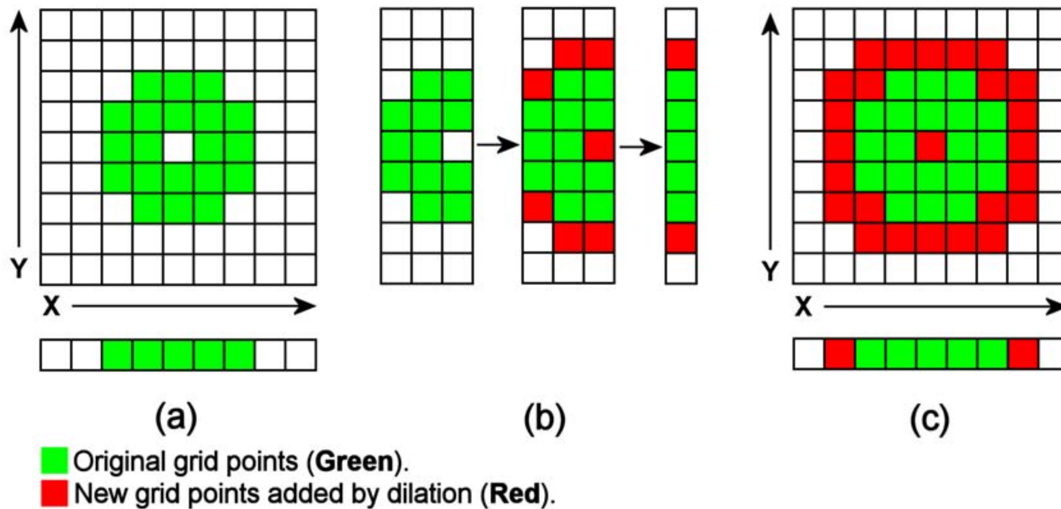


Figure 5.4: (a) The original DT-Grid. (b) The process of computing a new p-column (number three when counting from zero) in the dilated DT-Grid (adding the red grid points) (c) The final dilated DT-Grid.

We have taken a different less obvious but faster approach. In particular the dilation algorithm on a N-dimensional DT-Grid exploits the recursive definition of the DT-Grid and, except for the 1D DT-Grid, uses the dilation algorithm of its $N-1$ dimensional DT-Grid constituent recursively. This results in a fast dilation algorithm that ensures cache coherency for subsequent traversals.

The dilation algorithm consists of two steps: An allocation step that computes and allocates the dilated tubular grid, followed by a step that copies the values of the original tubular grid

to the dilated tubular grid. The time complexity of the allocation step is $O(C_N)$, where C_N is the total number of connected components in the original N-dimensional tubular grid. In most practical cases C_N is sublinear, *i.e.* $C_N << M_N$, where M_N is the number of grid points in the original tubular grid. The time complexity of the copying step is $O(M_N)$. Note that the number of grid points in the original and the dilated tubular grid are proportional [4]

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

An illustration of a tubular grid dilation in 2D is depicted in figure 5.4. Figure 5.4.a shows an initial 2D DT-Grid as well as its 1D DT-Grid constituent. Figure 5.4.c shows the result of dilating the 2D tubular grid by a stencil with $H = 1$. Grid points added to the DT-Grids by the dilation algorithm are colored red.

As mentioned earlier, the `xCoord` array of `DTGrid1D` stores a number of connected components, each identified by a start and an end index. The dilation algorithm in 1D simply amounts to dilating each of the connected components by H grid points in both directions and merging adjacent and overlapping connected components into a single connected component, see figure 5.4.

The 2D dilation algorithm starts by invoking the 1D dilation algorithm on the 1D DT-Grid constituent. Recall that the 1D DT-Grid constituent stores the orthogonal projection of the 2D tubular grid onto the X axis. The result of the 1D dilation is that the 1D DT-Grid constituent contains the dilated projection of the 2D tubular grid which is in fact equal to the projection of the dilated 2D tubular grid, see figure 5.4.c.

Next, each p-column of the dilated 2D DT-Grid has to be computed. Note that each element in the 1D DT-Grid constituent identifies a p-column in the 2D DT-Grid. Hence, we know exactly which p-columns should be computed in the dilated 2D tubular grid. The process of computing the dilated p-column number x proceeds as follows: First dilate the original p-columns numbered $x-H, .., x-1, x, x+1, .., x+H$ independently in the Y direction by H grid points. Next form the union of all the connected components resulting from this dilation in order to obtain p-column number x in the dilated 2D tubular grid. This is illustrated in figure 5.4.b. The index pairs contained in the dilated 1D DT-Grid constituent are computed simultaneously with the p-columns. Repeating the process above for each new p-column completes the dilation.

To sum up, the process outlined above dilates the DT-Grid in each dimension independently, and forms new p-columns by taking the union of the dilated original p-columns touched by the stencil. It should be clear, that this method is equivalent to shifting a hypercube-shaped stencil over the grid and including all grid points that pass under the stencil. In particular our approach essentially corresponds to separating the N dimensional hypercube into N differently oriented and axis aligned 1D hypercubes which are applied in each coordinate direction independently, much like a separable filter in image analysis.

To assist in the intuition of the dilation algorithm, figure 5.5 shows dilation applied to a slightly more complicated example than that in figure 5.4.

[4]This is due to the fact that the number of grid points in each tube are proportional to each other for a given level set. Tubes further away from the interface naturally contain more grid points than tubes closer to the interface. The constant of proportionality depends on the dilation procedure used to construct the tubes.

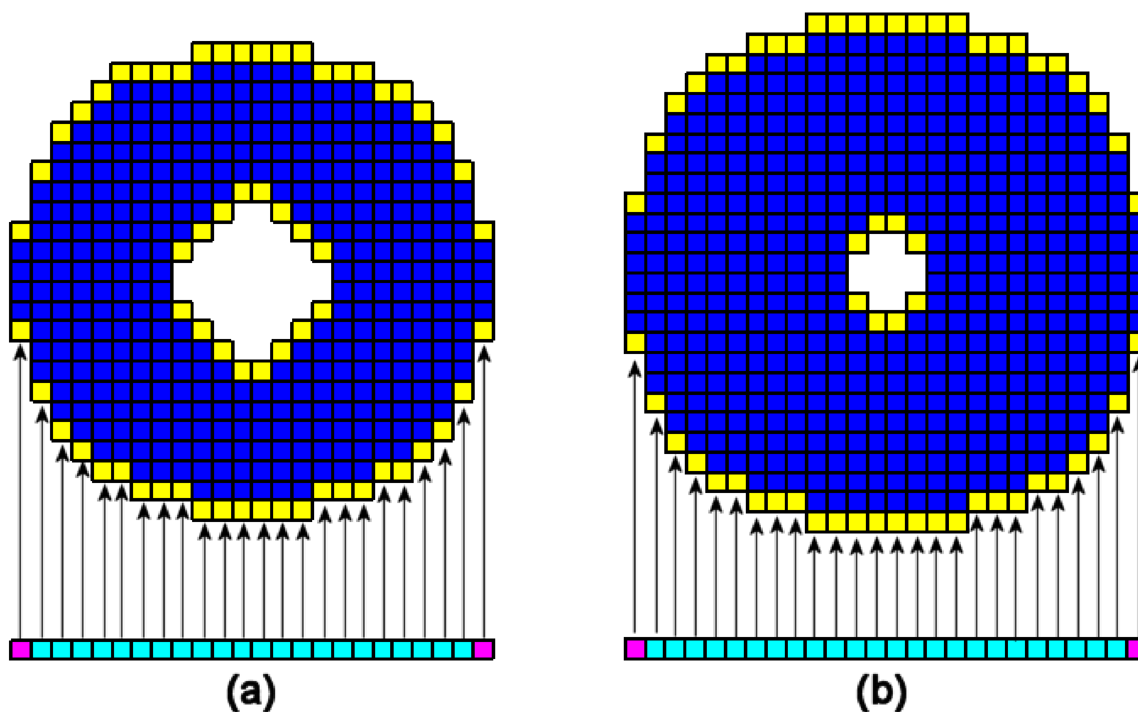


Figure 5.5: **(a)** The original 2D DT-Grid as well as its 1D DT-Grid constituent. Start and end coordinates of connected components are colored yellow and the internal grid points are colored blue. **(b)** The original 2D DT-Grid dilated with a stencil of width three ($H = 1$). Same color coding as in (a).

Details of the 1D Dilation Algorithm

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

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

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

  while ( i < xCoord.size() )
  {
     start = xCoord[i]-H;
     i++;
     if ( start <= end+1 )
     {
        // connected components overlap
        d1D.xCoord.last() = end = xCoord[i]+H;
     }
```

```
    else
    {
      // connected components do not overlap
      unsigned int j = d1D.xCoord.size();
      unsigned int connectedComponentLength = d1D.xCoord[j-1]-d1D.xCoord[j-2]+1;
      d1D.acc.push(d1D.acc.last()+connectedComponentLength);
      d1D.xCoord.push(start);
      end = xCoord[i]+H;
      d1D.xCoord.push(end);
    }
    i++;
  }

  // i points one past the end
  // d1D.xCoord[i-1]-d1D.xCoord[i-2]+1 are the number of grid points in the last connected component
  numValues = d1D.acc.last() + d1D.xCoord[i-1]-d1D.xCoord[i-2]+1;
  values.allocate(numValues);
}
```

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

Details of the N Dimensional Dilation Algorithm

In this section we describe the allocation step of the N dimensional tubular grid dilation algorithm for $N > 1$ (again, the copying step is trivial, see section 5.3.6). Recall that this algorithm effectively corresponds to shifting a stencil shaped as a hypercube with $2H + 1$ grid points along each edge over the original DT-Grid and including, in the new DT-Grid, all grid points that pass under the stencil.

Below we assume the existence of a data structure named the *Column Union*. The Column Union maintains a FIFO (First-In First-Out) queue of p-columns (given by the start and end coordinates of their connected components) and the maximal number of p-columns allowed in the queue is $(2H + 1)^{N-1}$, which is a constant. The Column Union furthermore supports the following operations

- **InsertColumn:** Inserts a p-column at the end of the queue. The p-column is identified by its start- and end-index into the `nCoord` array of `DTGridND`. This operation has time complexity $O(1)$.
- **RemoveColumn:** Removes the first p-column in the queue. This operation has time complexity $O(1)$.
- **ComputeUnion:** Computes a new p-column consisting of the connected components formed by taking the union of all connected components in the p-columns stored in the Column Union data structure. Furthermore each connected component is dilated independently by H grid points in each direction before including it in the union. Since the start- and end-coordinates of connected components are sorted within each p-column in the `nCoord` array, forming the union is simple: Scan through the coordinates of all connected components simultaneously in ascending order. Maintain a count that is incremented

whenever a connected component is entered and decremented whenever a connected component is exited. The coordinates encountered whenever the count is zero can be taken as the coordinates of the connected components of the union. Note that adjacent connected components must be merged into a single connected component. This operation has a time complexity that is linear in the number of connected components in the p-columns stored, since finding the minimum coordinate among all p-columns in the Column Union at each step takes time at most $O((2H+1)^{N-1}) = O(1)$.

Next we describe the allocation step of the N-dimensional tubular grid dilation algorithm in detail. We assume that the call to the method is initiated on a N-dimensional DT-Grid, `dtGridND`, as `dtGridND.dilate(H, dilatedDTGridND)`. When the call returns, `dilatedDTGridND` holds the dilated version of `dtGridND`. The `dilate` method proceeds as follows

1. Call `proj(N-1)D.dilate(H, dilatedDTGridND.proj(N-1)D)` recursively. After this call, `dilatedDTGridND.proj(N-1)D` will hold the dilated $N-1$ dimensional DT-Grid constituent. The pairs of indices, `IndexPair`, stored in `dilatedDTGridND.proj(N-1)D` are not yet initialized, only the raw storage is allocated. Recall that the pairs of indices identify p-columns in the dilated N-dimensional DT-Grid computed next.

2. Obtain an Iterator, `iteratorDilated`, from `dilatedDTGridND.proj(N-1)D`.

3. Obtain a Stencil Iterator, `stencilIteratorOrig`, from `proj(N-1)D`. At all time, `iteratorDilated` and `stencilIteratorOrig` will be centered over the same grid point, $\mathbf{X}_{N-1}$, although they operate on two different $N-1$ dimensional DT-Grid instances. The stencil of the Stencil Iterator should be a $N-1$ dimensional hyper-cube with size $(2H+1)^{N-1}$. Recall that a grid point $\mathbf{X}_{N-1}$ in `proj(N-1)D` identifies a p-column in the original N dimensional DT-Grid. The purpose of the stencil is to identify *all* p-columns in the original N dimensional DT-Grid required to form p-column number $\mathbf{X}_{N-1}$ in the dilated N dimensional DT-Grid by a union, see figure 5.4.b. An important observation is that as the stencil moves over `proj(N-1)`, it is only necessary to monitor which grid points, or equivalently p-columns, in the original N dimensional DT-Grid, enter and exit the stencil respectively. This means that the number of Iterator instances maintained by `stencilIteratorOrig` is in fact only $2(2H+1)^{N-2}$, corresponding to the back and front faces of the $N-1$ dimensional hyper-cube. The movement of `stencilIteratorOrig` will at all time be dictated by the movement of the `iteratorDilate` Iterator.

4. Iterate over all grid points in `dilatedDTGridND.proj(N-1)D` using `iteratorDilated`. For each such grid point, $\mathbf{X}_{N-1}$, do the following:

   (a) For each of the $2(2H+1)^{N-2}$ Iterator instances of `iteratorOrig` that point to an existing grid point, $\mathbf{Y}_{N-1}$, in `proj(N-1)D`, do the following: At $\mathbf{Y}_{N-1}$ `proj(N-1)D` contains an `IndexPair` and hence identifies p-column $\mathbf{Y}_{N-1}$ in the original N dimensional DT-Grid. If p-column $\mathbf{Y}_{N-1}$ is entering the stencil, call `insertColumn` on the Column Union data structure to insert p-column $\mathbf{Y}_{N-1}$. If p-column $\mathbf{Y}_{N-1}$ is exiting the stencil, call `removeColumn` on the Column Union data structure. Iterating the stencil over the grid points of `proj(N-1)D` (and hence the p-columns of the original N dimensional DT-Grid) in lexicographic order, ensures that p-columns enter and exit the stencil, and hence the Column Union data structure, like a FIFO queue.

(b) Call `computeUnion` on the Column Union to compute the connected components of the new p-column $\mathbf{X}_{N-1}$ in the dilated N dimensional DT-Grid. At this point the `ColumnUnion` contains all p-columns from the original `DTGridND` touched by the $(2H+1)^{N-1}$ stencil centered at $\mathbf{X}_{N-1}$. The `IndexPair` in `dilatedDTGridND.proj(N-1)D` is set to point to the dilated p-column $\mathbf{X}_{N-1}$ and the connected components of $\mathbf{X}_{N-1}$ are inserted directly into the `nCoord` array of the `dilatedDTGridND`. Furthermore the content of the `acc` array is computed by a $O(C_N)$ scan through the start and end indices of the connected components of $\mathbf{X}_{N-1}$ to determine the number of grid points in the p-column. At the same time a variable `numValues` indicating the total number of grid points included so far in `dilatedDTGridND` is incremented by the number of grid points in p-column $\mathbf{X}_{N-1}$.

(c) Increment `iteratorDilated`. The movement of this Iterator dictates the movement of the `stencilIteratorOrig` Stencil Iterator.

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

The time complexity of the allocation step of the ND tubular grid dilation algorithm can be derived as follows. We first exemplify in 2D and generalize to ND in the end. In 2D, we first dilate the 1D DT-Grid constituent which takes time $O(C_1)$ as described in the previous subsection [5]. Next every p-column in the original 2D DT-Grid (of which there are M_1) is inserted into and removed from the Column Union data structure $(H+1)^{(2-2)} = 1$ times. Each such operation takes time $O(1)$. Hence the total time for this is $O(M_1)$, since $(H+1)^{(2-2)} = 1$ is a constant. To compute the new p-columns, each connected component (of which there are C_2) is used $2H+1$ times, since this is the number of dilated p-columns, or unions, that it contributes to. Each step of a union takes time $O(2H+1)$ since the selected connected component was located from a total of $2H+1$ possibilities to be the connected component with the smallest start or end Y coordinate. In total, computing the new p-columns takes time $O((2H+1)^2C_2)$, which equals $O(C_2)$, since $(2H+1)^2$ is a constant. Finally, summing all the contributions gives a time complexity of $O(C_1 + M_1 + C_2)$ which is $O(C_2)$ since $M_1 = O(C_2)$ and $C_1 = O(C_2)$.

The complexity analysis proceeds similarly in N dimensions except that in analysing each of the N levels of the data structure, $(H+1)^{(2-2)}$ must be replaced by $(H+1)^{N-2}$, $(2H+1)$ must be replaced by $(2H+1)^{N-1}$ and $(2H+1)^2$ replaced by $(2H+1)^{2N-2}$, all of which are constants. Hence the allocation step of the dilation algorithm on an N dimensional DT-Grid has time complexity $O(C_N)$.

5.3.7 Rebuilding the Tubular Grid in Linear Time

In this section we outline a generic algorithm to rebuild a DT-Grid, denoted T_α, to include all grid points within a distance α from the interface. By *generic* we mean that the algorithm can be applied independently of the method used to reinitialize the tubular grid (*i.e.* , solve the Eikonal equation, $|\nabla\phi| = 1$). A main building block in this algorithm is the tubular grid dilation algorithm described in section 5.3.6. The algorithm devised here assumes that the original tubular grid is a distance field and contains all grid points in T_δ, where $\delta < \alpha$. The difference $\alpha - \delta$ may be equal to the maximal movement of the interface between rebuilds.

[5] Note that the dilation process cannot add new connected components, it can only merge connected components

Note that if a method is not restricted by the CFL condition [111], as *e.g.* the case with semi-Lagrangian integration, the maximal movement need not be equal to dx. Rebuilding the tubular grid is composed of the following steps

1. Remove from the original tubular grid all grid points outside T_δ. In practice this is done by constructing an intermediate tubular grid and copying all grid points within T_δ to it. This step has time complexity $O(M_N)$.

2. Dilate the intermediate tubular grid by $\alpha - \delta$, where $\alpha - \delta$ corresponds to the difference in width between the tubes T_α and T_δ, using the tubular grid dilation algorithm of section 5.3.6. This step also has time complexity $O(M_N)$.

3. Initialize the values of the grid points included in the new tubular grid by the dilation algorithm to $\pm\delta$ depending on whether they are interior or exterior to the region bounded by the interface. This step also has time complexity $O(M_N)$.

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

5.3.8 CSG Operations in Linear Time

The standard CSG (Constructive Solid Geometry) operations of *union*, *intersection* and *difference* can be computed between two ND DT-Grids in time $O(M_N + Q_N)$ where M_N and Q_N are the number of grid points in the first and second DT-Grid respectively. We limit our presentation here to CSG operations between two aligned grids, *i.e.* two grids that are subject to the same scale and rotation.

On a dense uniform grid, the CSG operation between two level set embeddings, ϕ_1 and ϕ_2, can be computed in a single simultaneous pass over both grids where a function is evaluated at each grid point. Assuming the negative inside and positive outside sign convention, the function to be evaluated amounts to $\min(\phi_1, \phi_2)$ for union, $\max(\phi_1, \phi_2)$ for intersection and $\max(\phi_1, -\phi_2)$ for subtracting ϕ_2 from ϕ_1 [111].

Even though ϕ_1 and ϕ_2 are signed distance fields, the embedding function resulting from a CSG operation is not necessarily a signed distance field. There are several cases where this occurs and the reader is referred to [13] for a comprehensive list of observations. For this reason, reinitialization techniques and narrow band rebuild must be applied after a CSG operation. This is of course no different on a DT-Grid.

The CSG operation between two DT-Grids $\mathcal{M}$ and $\mathcal{Q}$ resulting in a DT-Grid $\mathcal{S}$ proceeds similarly to the case with a uniform grid outlined above. The main difference is that on the DT-Grid we visit only the grid points included in the two tubular grids as opposed to the full volumetric embeddings. Furthermore, only when the CSG function evaluated at a grid point results in a value that is numerically less than γ (the width of the narrow band), is that grid point included in $\mathcal{S}$. However, due to the fact that the CSG operation does not in all cases preserve the signed distance field property, grid points further than γ away from the zero crossing of $\mathcal{S}$ may actually be included in the tubular grid of $\mathcal{S}$. These grid points can be removed after the CSG operation by discarding grid points away from the zero crossing and subsequently re-computing a signed distance field.

The CSG operation between $\mathcal{M}$ and $\mathcal{Q}$ proceeds as follows. First an iterator from $\mathcal{M}$ and $\mathcal{Q}$ is obtained. Each of these iterators provides access to the grid points in the corresponding

DT-Grid in lexicographic order. Next continue iteratively with a simultaneous sequential scan through both DT-Grids utilizing the two iterators as follows: In each iteration the coordinates of the current grid point pointed to by each iterator are compared. If the coordinates are equal, the given CSG function is applied and the grid point inserted into $\mathcal{S}$. If the grid points are *not* equal, it means that a grid point that does not exist in both tubular grids has been encountered. Choose the lexicographically smaller of the two grid points and denote it $\mathbf{X}_N$. Assume without loss of generality that $\mathbf{X}_N \in \mathcal{M}$ and $\mathbf{X}_N \notin \mathcal{Q}$. Whether the given grid point should be included in $\mathcal{S}$ depends on the type of CSG operation being applied and the inside/outside status of $\mathbf{X}_N$ in $\mathcal{Q}$. For example, if the union is being computed, $\mathbf{X}_N$ should be included in $\mathcal{S}$ if it has positive sign in $\mathcal{Q}$ and otherwise not. This is due to the simple fact that since $\mathbf{X}_N \notin \mathcal{Q}$ and $\mathbf{X}_N$ has positive sign in $\mathcal{Q}$, its value is implicitly γ. Since $\mathbf{X}_N \in \mathcal{M}$ its value is less than γ. Hence so is the minimum and $\mathbf{X}_N$ should be included in $\mathcal{Q}$. Using similar observations one can deduce that in the case of CSG intersection, $\mathbf{X}_N$ should be included in $\mathcal{S}$ if it has negative sign in $\mathcal{Q}$, and for CSG difference it should be included if we are subtracting $\mathcal{Q}$ from $\mathcal{M}$ and it has positive sign in $\mathcal{Q}$ or if we are subtracting $\mathcal{M}$ from $\mathcal{Q}$ and it has negative sign in $\mathcal{Q}$.

Since a single simultaneous scan over both tubular grids is the only processing required, it is clear that the time complexity amounts to $O(M_N + Q_N)$. No detailed subsection is provided in this case since our textual description translates directly into an algorithm utilizing higher level DT-Grid concepts such as iterators and push operations introduced earlier.

5.4 Augmenting the DT-Grid with Auxiliary Data

The numerical values of the level set are an integral part of the DT-Grid. This is the case since they are required to determine the signs of grid points outside the narrow band during stencil iteration as well as during neighbor and random access. However, due to the fact that the topology and values of the DT-Grid are stored separately, a value is uniquely determined by a single index into the sequentially allocated `value` constituent of the ND DT-Grid. Although this index is not explicitly stored for each grid point, it is computed during access to the data struture, both sequential and random. This also means that upon access, the index of the value can conveniently be used to access additional auxiliary properties (or *subordinate fields* as denoted in [48]) attached to each grid point and stored outside the core DT-Grid data structure. For example, this has been used to succesfully implement DT-Grid based surface velocities for fluid simulation, DT-Grid based particle level sets and the solution of PDEs on DT-Grid based level set manifolds [106], see chapter 15 for additional detail. In particular these applications require particles, scalars and velocities attached to each grid point.

5.5 Open Level Sets

To increase the efficiency of level set operations in practice, several applications work only on subsets, or sub-volumes, of the level set embedding. This is a widely used technique in both visual effects production and academia. One example is the utilization of level sets for geometric modeling as introduced by Museth *et al.* [96]. In their work, the editing operations are usually applied locally. This makes it possible to achieve near interactive response times by deploying the editing operators on sub-volumes enclosing their support/domain. Had the computations been used on the entire level set, it would have resulted in unnecessary computation away from the support of the localized editing operators and hence implied a significant slowdown.

A similar approach can be taken when utilizing level sets for collision detection in *e.g.* cloth simulation [16], simulation of deformable bodies based on finite elements and level sets [53], or fluid simulation interacting with boundaries [35]. If a conservative estimate can be made on the part of the level set relevant for the interaction, the particular subset can be extracted from the global level set embedding, thus resulting in lower memory footprints and improved response times due to *e.g.* increased cache coherency.

Extracting subvolumes from a level set surface may result in so-called *open* or *unenclosed* level sets, where the surface is not necessarily closed. Although the level set method in general assumes a closed surface, open level sets should pose no problems as long as the deformations tend to zero at the boundary (as in [96]), proper boundary conditions are specified at the boundaries or if only lookups of level set values are required. When representing level sets on dense uniform grids, open level sets require no specialized attention, but for a sparse representation such as the DT-Grid which as described up to now assumes a closed surface, the associated algorithms break down in certain cases. The H-RLE grid described in the next chapter allows for the representation of open level sets without any modification to its algorithms due to the fact that it stores information about regions outside the narrow band. In contrast the DT-Grid does not store any information on regions outside the narrow band. Retrospectively, it is in fact possible to represent open level sets on a DT-Grid as long as the sub-volume, from which the open level set is extracted, is convex and non-empty (*i.e.* intersects the narrow band). However, it does require a few minor modifications to the algorithms, and in the following we briefly review these modifications.

Consider figure 5.6.a where only the portion of the level set inside the blue rectangle is required for a particular application. The immediate solution to representing this subset on a DT-Grid is to perform a CSG intersection between the rectangle and the level set. The result of this can be seen in 5.6.c. However this introduces an undesirable narrow band around the border of the rectangle thus adding to the storage requirements and search times in the level set subvolume. Instead it is preferable to only store the part of the narrow band relevant for the interaction with the simulation at hand as shown in 5.6.b. Unfortunately the random access as well as stencil iteration algorithms break down outside the narrow band in such cases. In particular, the random access algorithm in section 5.3.2 will return $+\gamma$, where γ is the width of the narrow band, for all grid points above the narrow band in figure 5.6.b [6], whereas it should in fact return $-\gamma$ since these grid points are inside the original surface. The situation complicates further when the narrow band of the subset is not connected as shown in figure 5.7 corresponding to the part of the level set inside the red rectangle in figure 5.6.a.

Since the endpoints of any connected curve, fully contained within a convex bounding volume and not crossing over the narrow band must have the same sign, it is indeed possible to deduce the correct sign from a DT-Grid representation of an open level set. Consider random access into a grid point outside the narrow band in the open level set depicted in figure 5.7. In 2D there are two cases we must consider. First consider point a. Since it lies in between the two connected narrow band components it must have sign identical to any of the first[7] values encountered in the $\pm X$ directions [8] and illustrated in red in figure 5.7. Next consider point b. In that case we need to take into account the signs of the two lexicographically closest grid points, depicted in blue. If the signs at these grid points are equal we just return γ multiplied by this sign. If the

[6] This is due to the fact that since the grid point does not exist, the sign of the nearest lexicographically larger grid point is used.

[7] By *first* we refer to the lexicographic ordering of the elements.

[8] Note that in some cases only grid points in one of the directions are available.

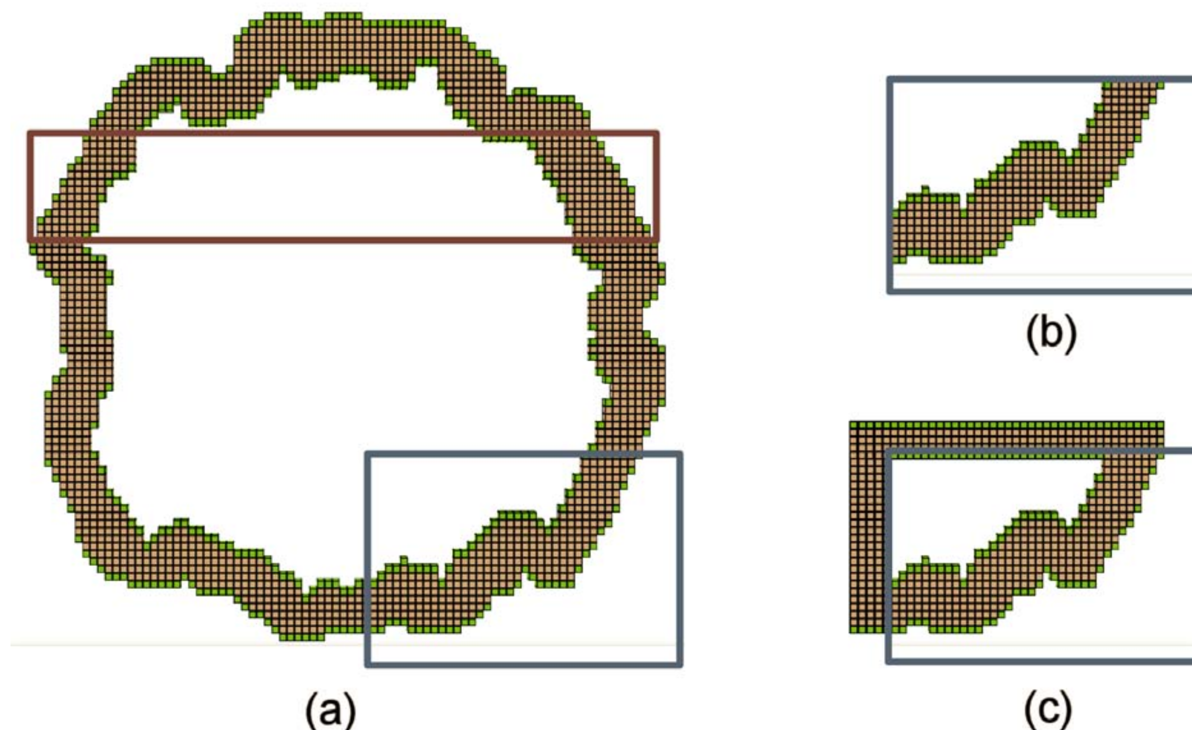


Figure 5.6: **(a)** Original DT-Grid encoding of a narrow band level set. **(b)** The open narrow band level set resulting from discarding all values of the original level set outside the blue rectangle. **(c)** The closed narrow band level set resulting from a CSG intersection between the blue rectangle and the original level set.

signs are not equal, one of the grid points *must* be in the same p-column as b and hence we can just pick the sign of this value [9]. To sum up, random and neighbor access into open level sets on a DT-Grid is possible, but requires modifications to the algorithms. In general the treatment of open level sets will slow down random access in regions outside the narrow band because we always need to lookup the sign of the lexicographically closest value which is not incorporated into the algorithm given in section 5.3.2.

In the case of stencil iteration and CSG operations, a similar approach considering the lexicographically closest elements must be taken when determining the signs of grid points outside the narrow band. A simpler strategy can also be employed when performing stencil iteration under the assumption that if a grid point in the stencil is outside the narrow band it will have the same sign as the sign of the center element. This is true for all the applications we have considered and eliminates the need to employ the more elaborate strategy described above. Narrow band dilation and rebuild is also possible: The changes required here include the approach for determining the correct sign as above as well as the need to intersect the dilation of the individual p-columns with the bounding box of the open level set.

Given the arguments above we conclude that DT-Grid is indeed capable of representing and manipulating open level sets. It does however require slight modifications to the algorithms as

[9] Otherwise we could construct a connected curve which connects two values of different sign and which does not cross over the narrow band and the boundaries of the bounding box.

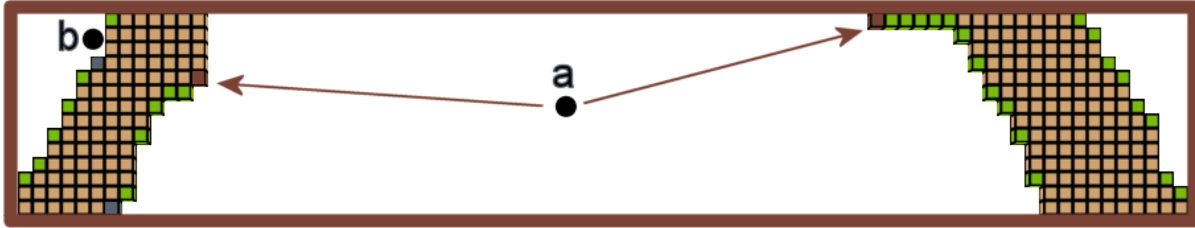


Figure 5.7: The open narrow band level set resulting from discarding all values of the original level set outside the red rectangle in figure 5.6.a.

well as the addition of a bounding box enclosing the narrow band. It is left for future work to investigate exactly how much these changes affect performance.

5.6 Examples

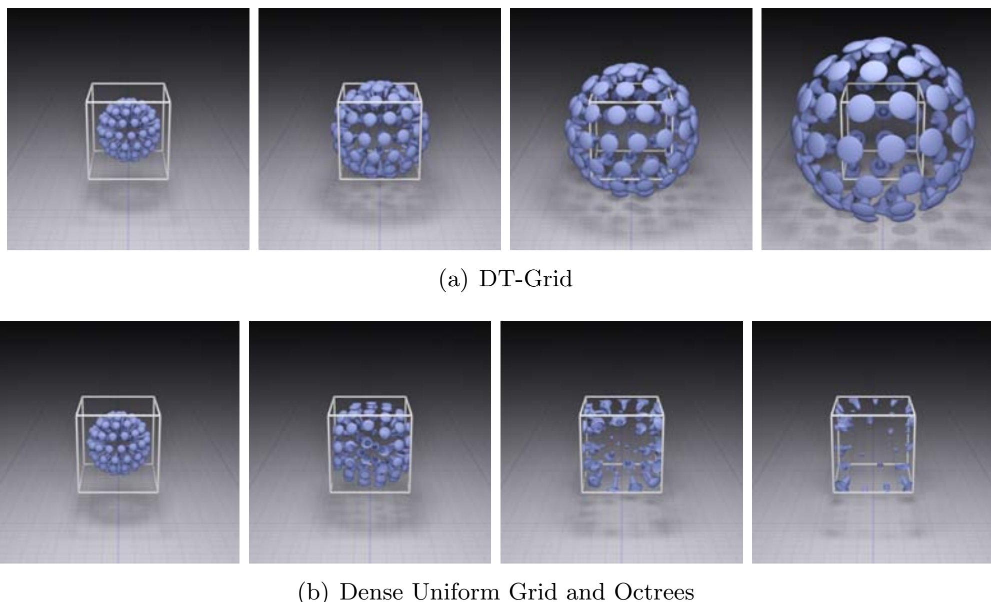
We next proceed with the illustration of two important qualitative features of the DT-Grid. In particular we show its ability to support high resolution level set deformations and its intrinsic out-of-the-box feature. In chapter 7 we present a more detailed quantitative evaluation of the DT-Grid and finally in chapter 15 several applications in computer graphics are demonstrated. This includes the utilization of the DT-Grid for storing both the surface and fluid interior in high resolution fluid simulation.

5.6.1 An Out-Of-The-Box Simulation

The DT-Grid is not bounded by a fixed computational domain. In this section we illustrate this important property by evolving the left-most level set shown in figure 5.8(a) using convection-diffusion [111]. In the convection-diffusion equation we combine propagation by mean curvature with advection in a velocity field that at each point is the normalized radial direction from the origin. The mean curvature term creates multiple pinch-offs at the center of the level set surface (see the left-most image in figure 5.8(a)), and we force its contribution to zero over time. The radial velocity field advects the level set surface away from the origin. This simulation is designed merely to demonstrate the out-of-the-box feature of our DT-Grid, and as such the detail of the setup (including the initial shape) is irrelevant.

Figure 5.8(b) illustrates what happens on statically allocated dense uniform grids and octrees. As the level set moves beyond the boundary, it disappears. In contrast, the level set on the DT-Grid (figure 5.8(a)) moves beyond the box as the *grid is not bounded*. The memory usage of the DT-Grids in figure 5.8(a) is 14.0MB, 25.2MB, 39.3MB and 64.1MB, respectively, numbered from left to right. (Note that the increase in memory is exclusively due to the fact that surface area increases with the expansion). While the DT-Grid in general is a non-convex tubular grid, the effective grid sizes in figure 5.8(a) are approximately 256^3, 343^3, 455^3 and 630^3.

This "out-of-the-box" feature is not obtainable using existing narrow band or standard octree based approaches without either compromising memory consumption or computational efficiency. Using dense uniform grids, one could progressively allocate larger grids as the level set approaches the boundaries. However, due to memory constraints this becomes impossible already at relatively small grid sizes. Octrees are more memory efficient than the dense uniform grid and narrow band approaches, but if progressively reallocating to larger octrees, the depth



(a) DT-Grid

(b) Dense Uniform Grid and Octrees

Figure 5.8: DT-Grid level set simulations can go out-of-the-box whereas level set simulations based on dense uniform grids and octrees are limited to the domain of the underlying grid. Renderings by Ola Nilsson based on my simulation data.

of the tree grows, making the traversal less efficient. The combined uniform and octree grid by Losasso *et al.* [83] decouples the depth of the octree from the overall domain size, hence making an expansion of the domain more feasible. Nevertheless their coupling of the octree structure with a coarser uniform grid will eventually face excessive memory usage, in particular in the case of large and sparsely populated computational domains. Finally recall that in chapter 3 several other approaches for obtaining approximate out-of-the-box behavior were reviewed. As explained, they face certain limitations. On the DT-Grid, deformations can expand semi-indefinitely as long as the storage requirements of the narrow band stay within the bounds of the available physical memory. In particular the out-of-the-box feature of the DT-Grid is enabled automatically by its recursive definition as well as the dilation algorithm described previously.

Out-of-the-box level set simulations can be convenient since no boundary conditions are required unless needed by the particular application. Furthermore the level set can move freely without ever colliding with the boundaries of an underlying grid. A large body of existing work could take advantage of this, e.g., the simulation of dendritic growth in [40]. In chapter 15 we illustrate several applications taking advantage of the DT-Grid's unique out-of-the-box feature. These include fluid and snow simulations.

5.6.2 A High Resolution Simulation - The Enright Test

The DT-Grid has a relatively low memory footprint, hence allowing high resolution level set surfaces to be represented without exceeding the main memory limit. We illustrate this here using the *Enright Test* [33]. To demonstrate the volume conserving properties of the particle level set method, Enright *et al.* [33] introduced the Enright Test based on a three dimensional incompressible (*i.e.* divergence free) flow field initially proposed by LeVeque [75]. The setup used is: A sphere with a radius of 0.15 is placed within a unit computational domain at position

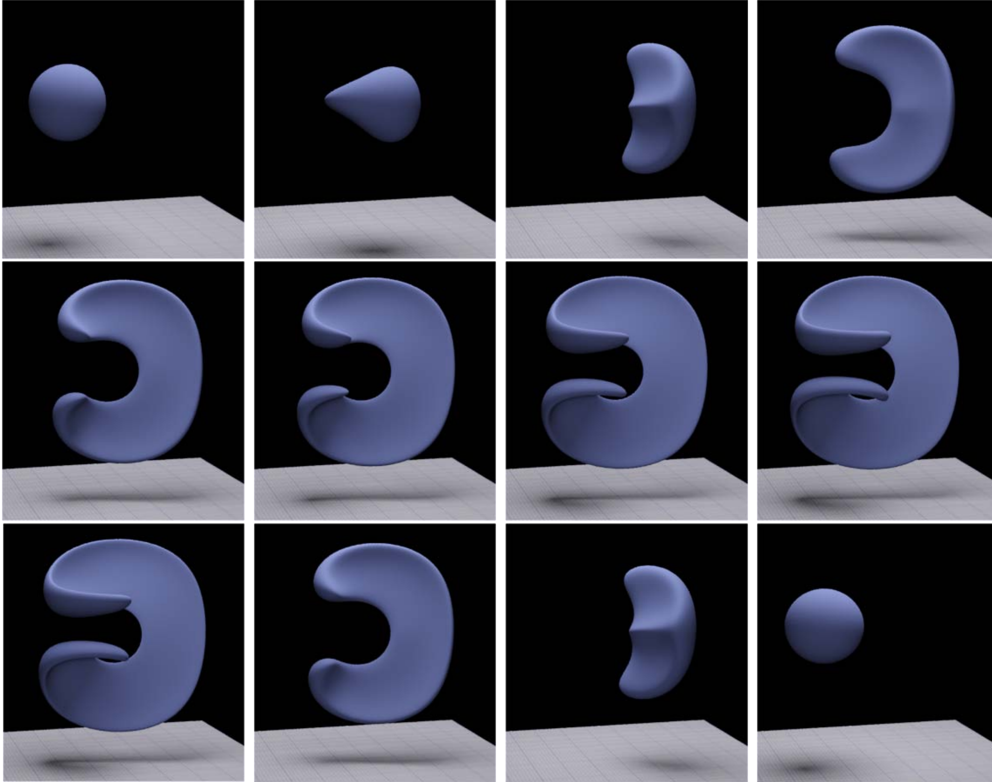


Figure 5.9: The DT-Grid enables high resolution grids to be represented with a very low memory footprint. In this example the resolution was 1024^3 and the maximal memory usage of the corresponding DT-Grid was 67.2MB. The same simulation run with the method of Peng *et al.* [120] would consume 5.2GB of storage (computed analytically). Renderings by Ola Nilsson based on my simulation data.

(0.35, 0.35, 0.35). The sphere is then advected in the velocity field:

$$\begin{aligned} u(x,y,z) &= 2sin^2(\pi x)\,sin(2\pi y)\,sin(2\pi z)\,cos\left(\tfrac{t2\pi}{T}\right) \\ v(x,y,z) &= -sin(2\pi x)\,sin^2(\pi y)\,sin(2\pi z)\,cos\left(\tfrac{t2\pi}{T}\right) \\ w(x,y,z) &= -sin(2\pi x)\,sin(2\pi y)\,sin^2(\pi z)\,cos\left(\tfrac{t2\pi}{T}\right) \end{aligned} \tag{5.1}$$

where $T = 3$ is the period of t. This velocity field is divergence free and is reversed at time $t = 1.5$. As a result, a level set advected in this velocity field should return to its original shape at time $t = 3$, provided that sufficient resolution is used.

In [33] the Enright Test was run on a 100^3 dense uniform grid with and without particles as shown in figure 5.10. The particle level set method proved better in conserving the volume and thin features, however the resolution of the computational grid was still insufficient to capture the thin filaments. Later, Enright *et al.* [34] demonstrated that the interface could be fully resolved on an octree grid with an effective resolution of 512^3, using a combined semi-Lagrangian and particle level set method.

From figure 5.9 it can be concluded that the interface can also be fully resolved on a DT-Grid with an effective resolution of 1024^3 *without particles*. However, we stress that the DT-Grid should not be considered an alternative to the particle level set approach. The setup of this

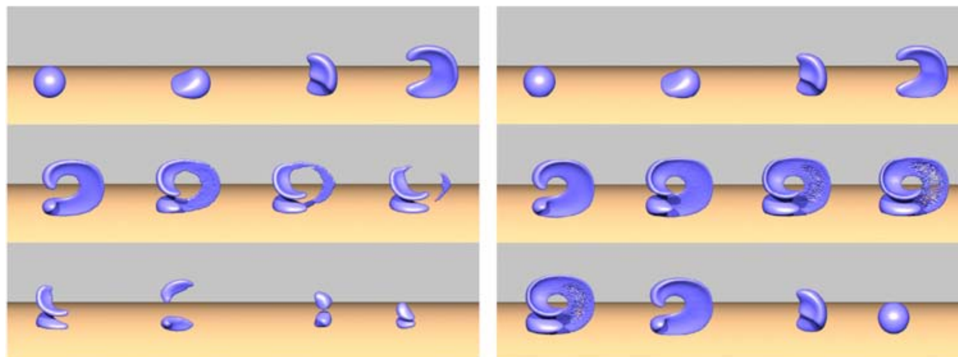


Figure 5.10: The Enright test in resolution $100 \times 100 \times 100$, reprinted from [33]. Left: Without particles. Right: Using particles.

simulation is identical to that of Enright *et al.* [33] which uses HJ-WENO [58, 59, 80] and TVD Runge-Kutta [133] for both the advection and reinitialization steps. This particular example clearly demonstrates that the DT-Grid enables high resolution interfaces to be represented. In particular the grid resolution can be refined to a level where particles are not needed in order to preserve the fine features and ensure a surface without holes.

At the time step where the number of grid points in the tubular grid peaks, 1.41% of the grid points in the full 1024^3 dense volume is occupied by the tubular grid and hence stored by the DT-Grid. At this time, the DT-Grid uses only 67.2 MB of storage in total. Furthermore, the memory used by the DT-Grid is only 1.64% of the storage that the 1024^3 full volume would occupy alone. The additional $0.23 = 1.64 - 1.41$% of storage used by the DT-Grid is occupied by the compressed indices and lower dimensionality DT-Grid constituents. From the number of grid points in the tubular grid at peak time, we computed analytically that the method of Peng *et al.* [120] would occupy at least 5.2GB of storage. Other non-hierarchical approaches to storing the tubular grid, such as representing the indices explicitly would result in a total memory usage of 9.86% and additionally storing pointers to the neighbors would yield 43.68%, giving a total memory usage of 144 MB and 490 MB respectively. The above analysis clearly illustrates the effectiveness of the index compression scheme employed by the DT-Grid.

5.7 Summary

This chapter presented the Dynamic Tubular Grid, or DT-Grid, and set of algorithms for representing high resolution level sets. We described the data structure as well as the push, random access, neighbor access, stencil iteration, dilation and rebuild algorithms in detail and argued for their asymptotic time- and storage-complexities. Finally we demonstrated that the DT-Grid enables high resolution level set simulations and a provided proof of concept example of its intrinsic out-of-the-box capability.

Chapter 6

The H-RLE Grid - Flexible High Resolution Level Set Simulations

Run-Length Encoding, or (RLE), is a widely used lossless data compression technique based on the simple idea of replacing sequences (runs) of identical data values by a count number and a single value (run code). The idea of RLE compressed level sets was originally proposed by Bridson [15], but Houston *et al.* [50] were the first to actually design and implement a RLE-based level set data structure. The reason RLE is particularly feasible in the context of level set embeddings is that level sets are most typically represented as *clamped* signed distance fields. The clamped regions consist of constant values well suited for RLE. Houston *et al.* employed compression of these clamped regions and left the values inside the narrow band uncompressed.

In this chapter we consider how to replace the p-column encoding of the DT-Grid data structure with a run-length encoding similar to that of Houston *et al.* [50]. We also describe how to modify the DT-Grid algorithms in order to accommodate the run-length encoding, and in fact only minor changes are required. We call the new data structure arising from these modifications the *Hierarchical Run Length Encoded* or *H-RLE* grid [48, 49].

The benefit of replacing the p-column encoding of the DT-Grid with a run-length encoding is that RLE allows for greater versatility in the encoding of the narrow band. This is in large part due to the notion of flexible run-codes which we define more carefully below. In particular RLE allows for flexible encodings of the narrow band, including grid cells of varying width, it decouples the level set values from the actual data structure and provides a solution to storing and processing open/unenclosed level sets created from intersecting a narrow band level set with an arbitrary shaped, *i.e.* possibly non-convex, volume. The price we have to pay for this flexibility is a slight degradation in performance with respect to storage requirements and computational efficiency for level set computations when compared to the DT-Grid. However, due to the adaptation of the DT-Grid algorithms to the H-RLE data structure, the H-RLE still in many cases performs better than octrees, narrow band level sets and the original Sparse RLE level set of Houston *et al.* [50] (see chapter 7). Next we outline the exact contributions of the H-RLE data structure.

6.1 Contributions

In many respects, the DT-Grid and H-RLE data structures are similar. However, in contrast to the DT-Grid, the H-RLE grid stores information (in compressed form) about the regions outside the grid by utilizing so-called run-codes. In this way greater versatility is obtained in

the H-RLE. In particular the H-RLE level set differs from the DT-Grid via the following three contributions:

- **Flexible P-Column Encoding with Run Codes:** As will be explained in section 6.2, the H-RLE data structure employs RLE to encode the domain into a series of runs, each associated with a specific *run code*. A run code categorizes a run as being either defined (inside the narrow band), or undefined. One benefit of run codes is that they allow for random access into open/unenclosed level sets and in general unenclosed scalar or vector fields. A defined or undefined run can be identified by wide range of possible run codes[1] and hence adds great flexibility to the encoding of defined and undefined regions. In fact the flexible encoding gives the possibility of representing defined grid cells of varying length in the direction of a run, hence essentially creating an adaptive representation [52]. In contrast the DT-Grid is restricted to storing uniform grid cells.
- **Decoupling:** Contrary to the DT-Grid, the H-RLE data structure decouples the RLE-compressed grid structure from the defined values in the narrow band. This can save storage (compared to the DT-Grid approach) when defined values are not required for the processing at hand. In particular the H-RLE can avoid storing any defined values and still correctly represent multiple regions of constant values. Additionally, in contrast to the DT-Grid, the H-RLE does not require any defined values in order to deduce properties of undefined regions.
- **Modified Algorithms:** The fundamental structures of *all* the H-RLE algorithms are identical to those of the DT-Grid. However, since the p-column encoding has changed to an RLE utilizing run codes and since information on regions outside the narrow band is available, modifications are required.

Practical applications of these features for computer graphics are discussed further in section 6.4. In addition to the three contributions just outlined, the H-RLE level set representation inherits the following features from the DT-Grid:

- Sequential traversal of the narrow band grid points with $O(1)$ access time to grid points within the finite difference stencil.
- Logarithmic time random access (and neighbor access) to any grid point inside the bounding volume, and constant time access outside the bounding volume.
- Both the encoding computational complexity and the overall memory consumption is proportional to and scales with the area of the geometric surface.
- Compatible with existing finite difference schemes used to solve the level set partial differential equation on dense uniform grids.
- "Out-of-the-box" dynamic grid representation that frees surface deformations from dealing with any fixed boundaries of the computational domain, hence allowing surfaces to expand and contract freely without encountering boundaries and without any extra book-keeping.
- Generalizes to any number of dimensions.

[1] Wide range of possible run codes, that is, within the bit-precision of a n bit run code this includes 2^n possibilities.

In the next section, 6.2, we define the H-RLE data structure. We start in 1D, next move to 2D and finally argue that the data structure generalizes to any dimension. Section 6.3 describe the fundamental algorithms of the H-RLE. Due to the similarities with the DT-Grid algorithms the descriptions are more brief and mainly give the overall outline as well as pinpoint the modifications required. After that section 6.4 focuses on and discusses the versatility of the H-RLE. Finally section 6.5 provides a brief summary of this chapter.

6.2 H-RLE Data Structure

As mentioned above, Run-Length Encoding (RLE) is a popular lossless data compression algorithm based on the simple idea of replacing sequences (runs) of identical data values by a count number and a single value (run code). A sequence of level set values, $\phi_1, .., \phi_n$, can be considered as a stream of data to which one can apply RLE. To facilitate compression of a narrow band level set of width β we introduce the following three run codes: *Negative* (interior region with $\phi < -0.5\beta$), *positive* (exterior region with $\phi > 0.5\beta$) and *defined* (within the narrow band, $|\phi| \leq 0.5\beta$). Each continuous sequence of adjacent values not within the narrow band is compressed to just a single run, whereas values within the narrow band are stored explicitly and uncompressed. In our implementation, the *defined* run code for grid points inside the narrow band will be an index into a separate array storing the corresponding values of ϕ contained in the run.

6.2.1 Taxonomy of the RLE Data Structure

As a prelude to the presentation of our dimensionally hierarchical data structure, it is convenient to introduce the following terms. A *run* is a sequence of connected values with the same *run code*. An *RLE segment* is a collection of adjacent runs corresponding to a single axis aligned scan-line through the level set. An *RLE block* represents the encoding of a collection of segments not necessarily immediately adjacent to each other and oriented along a particular axis. The RLE block serves as a fundamental building block in our hierarchical data structure. A *hierarchical RLE grid* is a recursive entity composed of n linked RLE blocks in R^n and finally a *hierarchical RLE level set* is a hierarchical RLE grid together with the separate array of defined values, see Figure 6.1. Within the hierarchical RLE grid, it is convenient to think of the hierarchy of RLE blocks as beginning with the *top* RLE block (highest level of encoding) and proceeding to *lower* RLE blocks (lowest level of encoding) until the *bottom* RLE block is encountered. Furthermore the bottom RLE block will correspond to the primary direction of encoding. In 3D, the top (level 3) and bottom (level 1) RLE blocks are respectively the z-axis and the x-axis RLE blocks [2].

Let us first consider the simple 1D level set function, $\phi(x)$, illustrated on the left in Figure 6.1. The rectangle at the bottom shows the RLE segment containing five (colored) runs corresponding to $\beta = 3$. The right hand side of Figure 6.1 illustrates the corresponding hierarchical RLE level set and its two components: An RLE block that encodes the topology of the segment (gray), and an array with discrete values of $\phi(x)$ within the narrow band (bottom). This decoupling of values and topology is essential in our data structure as it facilitates great flexibility. In fact, this separation is somewhat analogous to the separation of topology (connectivity) and geometry (coordinates of grid points) common in mesh data structures.

[2] This choice of assigning levels to axes is arbitrary and any choice is feasible.

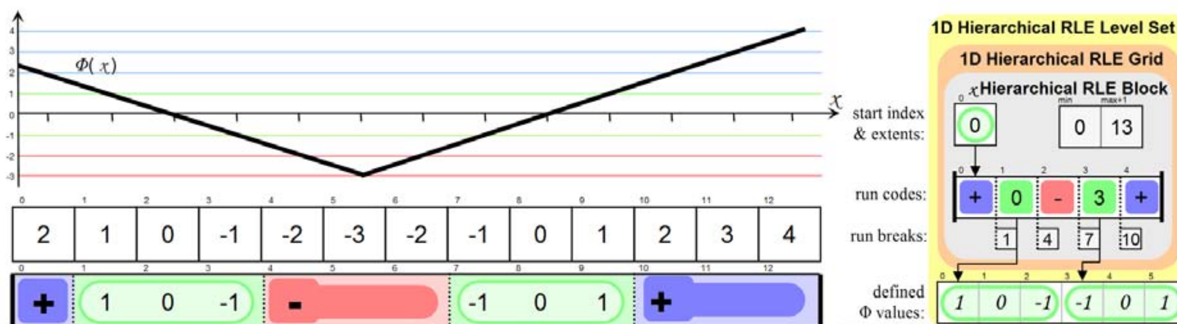


Figure 6.1: **Left:** 1D RLE encoding of a level set function with two zero crossings. The width of the narrow band is $\beta = 3$ and corresponds to the region of the green guidelines. **Right:** The corresponding 1-level hierarchical RLE level set data structure. Note the 1D RLE block illustrated by the gray box. See Section 6.2 for details.

Each RLE block consists of the following four separate arrays:

- *Start indices* for each segment are stored as pointers into the run-codes array. (In 1D a single pointer since the block only has one segment.)
- *Extents* of the coordinates along the encoding axis are stored as a $[\min, \max + 1)$ pair. Note that the $+1$ makes the extents consistent with the definition of the run breaks below.
- *Run codes* define how each segment is divided into different runs. *Negative* and *positive* run codes denote compressed runs while *defined* run codes are pointers into an associated defined data array or the start indices array of the next lower RLE block if it exists.
- *Run breaks* contain, in an order mirroring that of the run codes, the coordinates along the encoding axis at which each run starts, except for the first run in each segment. The start coordinates of the first run of each segment are determined by the above-defined minimum extents.

Note that although the hierarchical RLE grid coincides with the RLE block in 1D this is not the case for higher dimensions (see Figure 6.2.)

6.2.2 The Hierarchical Data Structure

Let us consider the 2D example in Figure 6.2 illustrating a 7×7 bounding box containing a subset of a level set function. In the bottom of Figure 6.2 is the corresponding 2D H-RLE level set. It is composed of two RLE blocks related hierarchically, the top RLE block encoded along the y-axis and the bottom RLE block encoded along the x-axis. First, each linear traversal of the 2D level set function along the x-axis is encoded resulting in the x-block shown in the bottom-right part of Figure 6.2. Note that traversals resulting in a single non-defined run are not stored but rather offloaded to the next higher level RLE block. The higher level RLE block is encoded along the y-axis leading to the y-RLE block shown in the bottom-left part of Figure 6.2. As shown in Figure 6.2 (bottom), the RLE blocks store the run-codes sequentially. Run-codes in the x-RLE block reference into the single array of defined values and indicate where that particular run of defined values begins. The run-codes of defined runs in the y-RLE block reference into

the x-RLE block start index array. This is related to the fact that the lowest level RLE block encodes level set values, whereas all higher RLE blocks are actually encoding the results of the previous RLE encoding. Note also that the bounding box extents are stored explicitly as part of the data structure. The bounding box associated with the H-RLE block extents is very useful for many graphics applications and is fully dynamic. In fact, the rebuild algorithm described in 6.3 applies a dilation algorithm that dynamically expands or shrinks this bounding box as the interface moves.

Similar to the DT-Grid, the H-RLE generalizes to any dimension through dimensionally hierarchical encoding based on the taxonomy introduced above. In particular the H-RLE stores one RLE block for each axial direction (or dimension) as well as an array of defined values. We stress that doing a hierarchical encoding, as opposed to only encoding in a single direction (as in Houston *et al.* [50]), ensures that the memory requirements of the data structure are $O(M_N)$, where M_N is the number of defined grid points inside the narrow band.

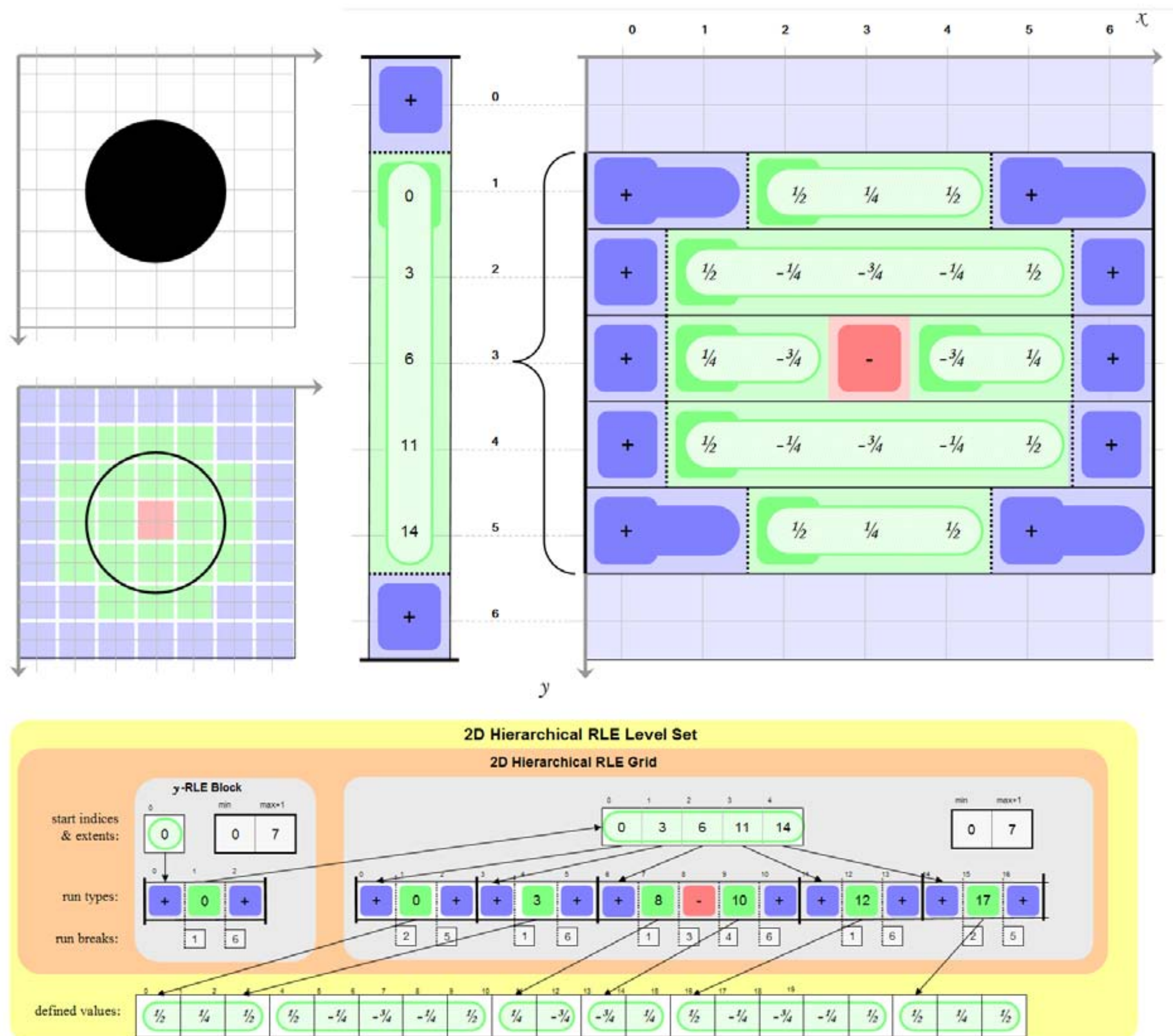


Figure 6.2: **Top Left:** A sphere shape overlaid on a coarse grid (notice the similarity with figure 5.1 in chapter 5). **Middle Left:** The sphere interface is now indicated. The grid points are categorized into narrow band (green), interior (red) and exterior (blue) grid points. **Top Right:** A schematic representation of a 2D H-RLE level set. The top-level y-axis encoding is given in the 1×7 grid and the x-axis encoding is the 7×7 grid. Along the y-axis encoding, the ranges 0 and 6 are categorized as fully exterior, while the range 1-5 is defined. The defined values of the y-axis encoding indicate the offset of the corresponding x-axis segment encoding. The x-axis encoding contains no information for regions not encoded by the y-axis as defined. The values of the defined ranges of the x-axis encoding are the level set values of the narrow band. **Bottom:** The layout and contents of the H-RLE level set data structure corresponding to the schematic representation above. The defined values of the y-RLE grid's defined runs are stored in the x-RLE grid's start indices table. The defined values of the x-RLE grid's defined runs are stored in the decoupled defined values arrays. See Section 6.2 for further explanation.

6.3 H-RLE Algorithms

In this section we describe a set of algorithms for the use of our H-RLE level set in graphics and level set applications. To ease the description, we refer the reader to the nomenclature defined in table 6.1.

N	The dimension of the H-RLE grid.
M_N, Q_N	The total count of defined grid points within the ND narrow band.
s_i	A segment index in the i'th dimension RLE block.
r_{s_i}	The number of runs in segment s_i in the i'th dimension RLE block.

Table 6.1: Nomenclature used throughout the chapter.

An overview as well as the time complexities of the algorithms being described in this section are given in table 6.2. Next we describe these algorithms in turn.

6.3.1 Constant Time Push

Due to the H-RLE's versatility in describing regions outside the narrow band as well as its decoupling of the defined values, the exact implementation of a push operation is problem dependent. Here we describe two low level push operations. As input the first push operation accepts the specification of a run whereas the second push operation accepts the coordinates of a grid point and a defined value. In both cases we assume that data is pushed in (Z, Y, X) lexicographic order [3] and that a bounding box for the grid is given[4]. The value returned when accessing grid points outside of the given bounding box is application specific. For level set simulations it is typically feasible to return the *positive* run code.

Push Operation I: The first push operation is applicable in general, not only for narrow band level sets. We assume that every grid point inside the bounding box is pushed exactly once (as part of a run) [5]. The push operation works recursively and is initially called on the bottom RLE block with a run code as well as the length and start coordinates[6] of the run. Note that values are not passed to the push operation, as the values are maintained separately from the topology.

- If the run begins a new segment, the start index of the segment is initialized (to the number of runs present in the RLE block) and stored along with the run code and run break (the run break is computed from the start coordinates and length of the run).
- If the run does not start a new segment it is checked whether the type of the run is identical to the type of the immediately adjacent run [7]. If this is not the case the new run is stored, and otherwise it is merged with the adjacent run.
- In the case where a higher level i'th RLE block exist, we call push recursively on it if the run ended the current segment (this can be determined from the extents known prior

[3]The (Z, Y, X) lexicographic order corresponds to the order of the hierarchical run-length encoding.

[4]Contrary to the DT-Grid, the H-RLE requires that the bounding box is known prior to the encoding.

[5]Depending on the particular application it should be relatively straightforward to relax this assumption in order to obtain a faster push operation. For example to allow pushing each grid point *at most* once.

[6]Essentially it is not necessary to specify the start coordinates of the run, since the bounding box and the previously pushed run uniquely determines the position of the next run.

[7]In the case of a standard level set encoding it is thus checked if they are both positive, negative or defined.

Algorithm	**Time Complexity**
Push	$O(1)$
Access to Stencil Grid Points	$O(1)$
Sequential Access	$O(1)$
Random Access	$O(1 + \sum_{i=1}^{N} \log r_{s_i})$
Neighbor Access in m'th Coordinate Direction	$O(1 + \sum_{i=1}^{m-1} \log r_{s_i})$
Rebuilding the tubular grid	$O(M_N)$
Dilating the tubular grid	$O(M_N)$
CSG operation	$O(M_N + Q_N)$

Table 6.2: Algorithms of a N-dimensional H-RLE data structure. Asymptotic time complexities are similar to those of the DT-Grid, except that r_{s_i} denotes number of runs instead of number of connected components. Furthermore, in the exposition given here, the encoding-order of the axes of the H-RLE data structure are XYZ, whereas for the DT-Grid they were ZYX. This makes the time complexity for neighbor search appear slightly different.

to the encoding). The parameters to the recursive push call are the last i coordinates of the start point of the run, a run length of one and a specific run code. The value of the specific run code is determined as follows: If the completed segment contains at least one defined run we use a *defined* run code equal to the index of the segment in the start indices array. If the completed segment consists of a single run we use the run code of this run. In the case where the completed segment consists of a single run, the encoding is completely deferred to the higher level RLE block by removing the completed segment from the current RLE block before the recursive call.

The above operation uses constant time on at most N RLE blocks and hence the time complexity is $O(1)$ for pushing a run.

Push Operation II: The second push operation is specifically designed to push values of a signed distance field level set. In this case it is convenient to only push the defined grid points inside the narrow band, hence leaving the encoding of the outside regions to the push operation. The push operation works recursively and is initially called on the bottom RLE block with the coordinates and value of the grid point as well as a *positive* or *negative* run-code in accordance with the sign of the value:

- If the grid point lies in a new segment, the encoding of the previous segment, if one exists, is completed with either a *positive* or *negative* run depending on which is consistent with the last defined value pushed onto the previous segment. Next a new segment (index) is initialized and, unless the grid point coincides with the minimum extent, an undefined run consistent with the sign of value being pushed is inserted first. Finally if $i \geq 0$, where i is N (the dimension) minus the level of the next RLE block, push is called recursively on the next higher level RLE block with the last $i+1$ coordinates of the grid point, the index into the segment start indices array and the *positive*/*negative* run-code as parameters.
- If the grid point lies in an existing segment there are two cases: If the grid point is adjacent to a defined run, this run is extended with the grid point. Otherwise a run consistent with

both the current and previously pushed grid point is inserted before the grid point itself.

Again, the above operation uses constant time on at most N RLE blocks and hence the time complexity for pushing a defined value is $O(1)$.

6.3.2 Logarithmic Time Random Access

The random access algorithm of the H-RLE level set begins as a procedural call on the top RLE block with two parameters: A vector of query coordinates, $Q = (q_1, .., q_N)$, and a segment index, s_N, initialized to zero. The algorithm then proceeds recursively similar to the algorithm described for the DT-Grid in chapter 5. Due to the different terminology of the H-RLE data structure we describe the random access operation in more detail below.

1. The vector of query coordinates $Q = (q_1, .., q_i)$ is split to isolate the i'th query coordinate, q_i.

2. From the given segment index, s_i, one can determine the indices of both the *first run* and the *first run break* as well as the *total number of runs in the segment* (the *segment length*) from the i'th RLE block. In particular, the index of the first run is given by the s_i'th entry in the start indices array, the index of the first run break is given by the index of the first run code minus s_i, and finally the number of runs in the segment is given by the difference between the s_{i+1}'th and s_i'th entries in the start indices array. The run code of the run containing q_i can be determined via a binary (or linear) search within the run breaks array in the region of the current segment (defined by the first run, first run break and the segment length). If the determined run code is *negative* or *positive*, then return the corresponding run code value. Otherwise, compute the *defined data index* by adding together the run code with the offset of q_i from the start coordinate of the run.

3. If a lower RLE block exists, then this procedure is recursively called on it (go to Step 1 above), using the remaining vector of query coordinates and the defined data index as the respective values of its vector of query coordinates and segment index parameters. If no lower RLE block exists, return the defined data index and/or the corresponding value in an associated array, *e.g.* the defined values of the level set function ϕ.

The computational complexity of the random access algorithm is as follows: Step 1 above takes constant time. Step 2 takes time logarithmic in the number of runs (as opposed to the number of grid points *within* the runs) in a single segment in an RLE block. Note that a binary search is possible since the run breaks are sorted in increasing order within each segment as part of the encoding. Step three calls recursively on a lower level RLE block of which there are at most $N - 1$, where N is the dimension of the H-RLE grid. The total time required amounts to $O(\sum_{i=1..N} \log r_{s_i})$, where r_{s_i} is the number of runs in the segment identified by segment index s_i and contained in the i-th dimension RLE block.

6.3.3 Logarithmic Time Neighbor Access

Neighbor search proceeds similarly to the algorithm employed for the DT-Grid. For example, finding a neighbor in the Y-direction requires first locating the neighboring segment in the Y-direction which can be done in constant time. Next a binary search is performed (as described in Step 2 of the random access operation above) amongst the run breaks of the neighboring segment

in the X-direction. In general the time complexity of a neighbor search in the m'th direction (where X is the first direction, Y is the second and so on) is $O(1 + \sum_{i=1..(m-1)} \log r_{s_i})$, where again r_{s_i} is the number of runs in the segment identified by segment index s_i and contained in the i-th dimension RLE block.

6.3.4 Constant Time Sequential Access

Constant time access to single grid points during sequential access is facilitated by means of iterators that proceed similarly to the iterators defined on the DT-Grid. The sequential access iterator begins as a procedure call on the top RLE block with two parameters: A segment index initialized to zero, and an empty vector of *parent coordinates* (specified below). The procedure begins by traversing the runs of the specified segment. Non-defined runs are ignored when encountered. When a defined run is encountered, both its length and its start coordinate along the current encoding axis are determined (from the segment indices s_i and s_{i+1}. Then an iteration along the individual coordinates of this run commences. At each coordinate, the defined data index is computed by adding together the run code with the current coordinate's offset from the start of the run. Also, a vector of current coordinates is created by concatenating the input vector of parent coordinates with the current coordinate. If the RLE block is the bottommost, then append both the current coordinate vector and the defined data index to the resulting enumeration. Otherwise, the above described procedure is called recursively on the next lower RLE block using the defined data index and the constructed current coordinate vector as the respective values of its segment index and parent coordinates parameters. This sequential access iterator traverses each defined run in each of the RLE blocks in addition to enumerating each of the defined data indices. The resulting computational complexity is $O(M_N)$ and hence $O(1)$ per defined data element.

In order to obtain constant time access to all grid points within the finite difference stencils typically used in level set deformations, one can iterate an entire stencil of iterators over the narrow band. Whenever a grid point in the stencil moves outside the defined narrow band region, the corresponding iterator continues into an undefined *positive* or *negative* run and is assigned the value associated with this run[8]. Similar to the DT-Grid case it is possible to exploit the movement of the center of the stencil to further improve the practical access time. All iterators within the stencil pass over the grid once, so - on average - access to grid points within the stencil is $O(1)$ if the entire grid is visited.

6.3.5 Rebuilding the Hierarchical RLE Level Set in Linear Time

Rebuilding the Hierarchical RLE Level Set proceeds exactly as in the case of the DT-Grid. In particular the operation is comprised of two steps: First grid points that are too far from the interface are removed. Next, the remaining subset of the original H-RLE level set is dilated recursively by effectively adding grid points that pass under a hypercube-shaped stencil being iterated over the subset. Below we briefly summarize the dilation operation in the terminology of the H-RLE data structure.

The dilation of a 1D H-RLE level set simply corresponds to dilating the corresponding 1D RLE segment as illustrated in Figure 6.3. It consists of two steps: First each defined run in the segment is dilated independently. Next, a new RLE segment is constructed by forming the union

[8] This type of iterator is also used during CSG operations on the H-RLE and constitutes the main difference from CSG operations on the DT-Grid.

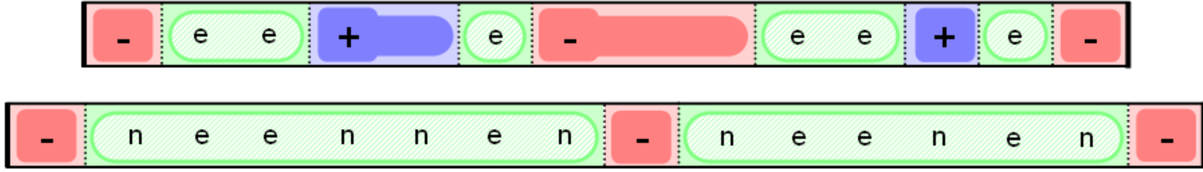


Figure 6.3: Dilation of an RLE segment by one grid point. **Top:** Original segment with existing defined grid points labeled **e**. **Bottom:** Dilated segment with new grid points labeled **n**.

of each dilated run and keeping the remaining inside/outside runs consistent with the original segment.

The dilation algorithm of a 2D H-RLE level set operates in each coordinate direction independently and proceeds as follows:

1. Dilate the y-axis RLE block as a 1D RLE segment. The y-axis RLE block corresponds to the projection of the 2D H-RLE level set onto the y-axis, and the dilated y-axis RLE block corresponds to the projection of the dilated 2D H-RLE level set.

2. To dilate the x-axis RLE block, the defined grid points of the dilated y-axis RLE block are processed iteratively: Let the current grid point have y-coordinate y_m. The corresponding dilated x-axis RLE segment is computed by first dilating all defined x-axis RLE segments in the original x-axis RLE block with y coordinate in the range $y_m - n$ to $y_m + n$, where n is the number of dilation grid points. Next the union is taken of these individually dilated x-axis RLE segments.

3. Allocate storage for the decoupled value array of the dilated 2D H-RLE level set. In a single pass, the value of grid points also existing in the original 2D H-RLE level set can be set to their original value and the value of grid points added as a result of the dilation can be initialized according to the *positive* and *negative* runs of the undilated H-RLE level set.

Contrary to the DT-Grid, the dilation of the H-RLE requires that the bounding box, or extents, of the resulting grid are known in advance as this information is required for doing the actual run-length encoding. However, given the extents of the original H-RLE grid as well as the dilation width, the new extents are straightforward to compute. Whenever new grid points enter the narrow band defined by the H-RLE, their sign is determined by the sign of the undefined run they were part of in the un-dilated grid.

6.4 Versatility of the H-RLE

Open/Unenclosed Level Sets:

In situations where large occlusions only have portions of their surface in close proximity to the body or fluid under simulation, it is advantageous to clip the occlusions to these simulation regions, especially if the whole occlusion object requires significant memory or requires per-frame scan conversion because of dynamic properties. This issue was discussed in more depth in chapter 5. Even though both the DT-Grid and H-RLE can represent open level sets, the H-RLE allows for more flexible encodings of open level sets, in that the open level set can be generated from intersection with an arbitrarily shaped, *i.e.* possibly non-convex and non-connected, bounding volume. On the DT-Grid the open level set is restricted to be generated

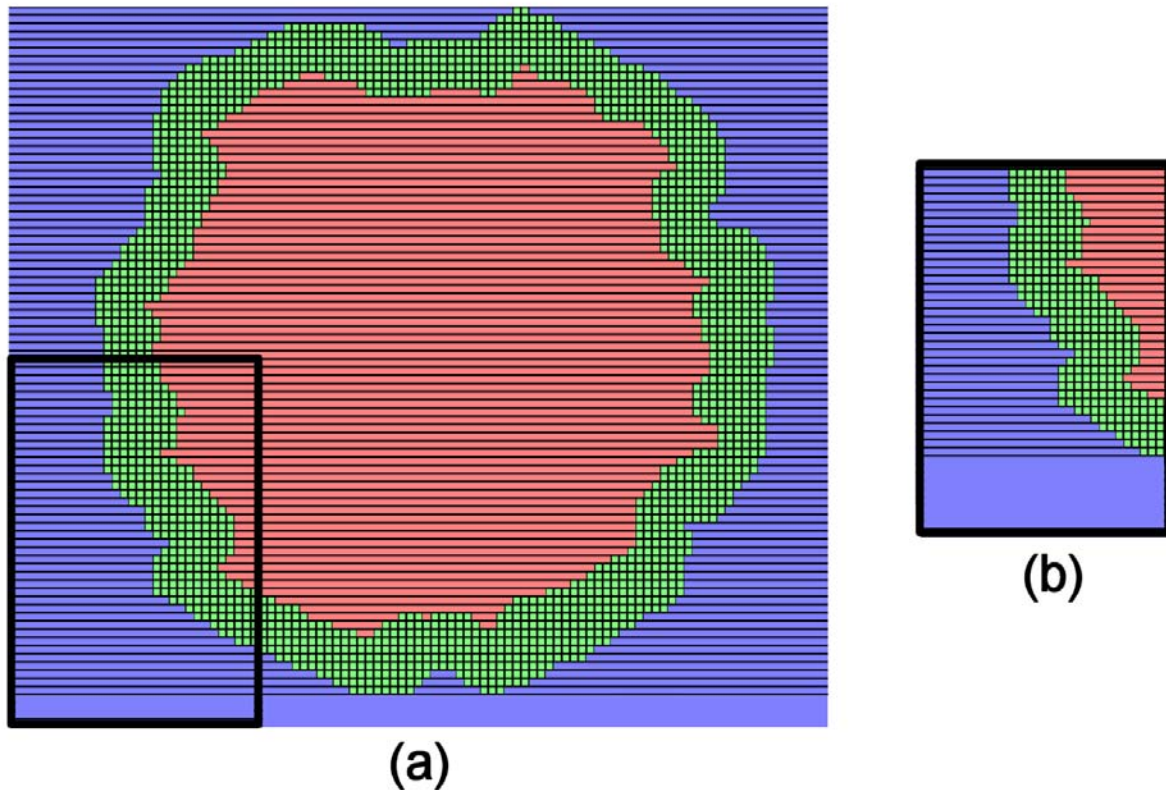


Figure 6.4: **(a)** A H-RLE encoding of a narrow band level set. Red denotes negative inside regions, blue denotes positive outside regions and green denotes the narrow band. **(b)** The open level set representation resulting from intersecting the H-RLE grid in (a) with the bounding box in the lower-left corner.

by a connected and convex bounding shape. Non-convex open level sets could be applied *e.g.* in the level set modeling framework of Museth *et al.* [96] when blending surfaces along intersection curves. Using the H-RLE, the small possibly non-convex volume within which the blending will take place can be extracted and used for processing the blend, thereby speeding up the computations. Additionally, random access into open level sets on the H-RLE have the possibility of being faster than on the DT-Grid. Consider the open level set in figure 6.4.b generated from the larger closed level set in figure 6.4.a. The wider uniform outside (blue) region in the bottom is represented at the top level in the encoding of the H-RLE. All queries falling into this range can be satisfied at the top level of the H-RLE encoding, since sign information is stored at this level. On the DT-Grid the search would need to progress to the finest level of the encoding in order to determine the sign of the closest value because the level set is open.

Flexible Encoding

Due to the H-RLE's utilization of run codes it is possible to obtain flexible encodings of both defined and undefined regions. For describing a basic encoding of level sets that allowed for both open and closed representations we employed three different run types in this chapter; positive, negative and defined. Recall that the defined run code refers to a set of run codes

denoting indices into either a defined value array or the segment start indices array on a lower level in the encoding. However, as argued previously a run can be represented by a wide range of possible run codes. For example the flexible encoding can be used to have several types of defined regions. An application of this is to have cells of varying length in the direction of a run, hence essentially creating an adaptive representation. One type of defined run can be used for runs of consecutive unit-length grid cells whereas another type can be used for grid cells of length greater than one. The idea of utilizing the flexibility of the H-RLE for this type of encoding was proposed in the recent [52] which leverages on the DT-Grid and H-RLE techniques presented in this dissertation. In particular [52] demonstrates the feasibility of grid cells of varying length in the context of fluid simulation of large bodies of water where tall columns typically have linear pressure profiles. Note that the DT-Grid is only able to represent grid cells of unit length.

Decoupling

The H-RLE decouples the defined (level set) values from the data structure and algorithms due to the utilization of run codes in specifying defined and undefined regions. One implication of this is that it may not be necessary to store any (level set) values at all. On the other hand, the DT-Grid data structure contains no information about inside or outside if no narrow band values are specified. However, using the H-RLE it is straightforward to *e.g.* represent multiple constant value regions efficiently simply by encoding them as runs with different run codes. A possible application of this is H-RLE representations of matrices with sequences of constant values. Finally we note that the H-RLE, contrary to the DT-Grid, is capable of storing both closed and open/unenclosed scalar and vector fields and still provide arbitrary encodings of outside regions. This property is not available with the DT-Grid which can only assign values/properties to regions outside the narrow band based on adjacent (level set) values inside the narrow band.

6.5 Summary

This chapter introduced the Hierarchical Run-Length Encoded (H-RLE) Grid, a versatile data structure for high resolution level sets. The H-RLE has many similarities to the DT-Grid, but employs a run-length encoding in place of the p-column encoding utilized by the DT-Grid. This makes the H-RLE more flexible than the DT-Grid in terms of the encoding of the narrow band, the decoupling of level set values from the hierarchical encoding and allows for more flexible representations of open level sets. The DT-Grid algorithms were adopted to the run-length encoding of the H-RLE with minor modifications.

Chapter 7

Evaluation and Discussion of DT-Grid and H-RLE Grid

The two previous chapters have presented the Dynamic Tubular Grid (DT-Grid) and the Hierarchical Run Length Encoded (H-RLE) Grid for representing and manipulating high resolution level sets. Algorithmic details and asymptotical time- and storage-complexities were provided along with a few examples demonstrating the out-of-the-box and versatility capabilities of these representations. Chapter 15 in part V reviews additional applications of these data structures for high resolution level set and fluid simulations. In this chapter we perform an evaluation of and discuss known strengths and weaknesses of the DT-Grid and H-RLE. Our evaluations cover level set simulations with several numerical schemes, memory requirements when representing high resolution level sets and render times for ray tracing which employs a high number of random access operations. The level sets used in our evaluations are converted from polygonal meshes using our methods described in chapter 14. These polygonal mesh models are publically available from the Stanford Scanning Repository and depicted in figure 7.1.

Briefly outlined this chapter proceeds as follows. First section 7.1 outlines the methodology and presents the various data structures against which we evaluate the DT-Grid and H-RLE. Next section 7.2 presents the benchmarks for level set simulations. Following that sections 7.3 and 7.4 evaluate the random access performance as well as the storage requirements for high resolution level set representations. Finally section 7.5 discusses known strengths and weaknesses of our work, and section 7.6 concludes this chapter with a brief summary.

7.1 Evaluated Data Structures and Methodology

The performance evaluations presented in this chapter compare several different level set representations and algorithms with respect to time- and memory-usage. Several of the evaluated data structures have identical asymptotic time complexities associated with their operations, and hence comparing their actual run-times is a delicate matter. In particular the constants involved in the time-complexities depend strongly on the level of optimization applied to the implementation. Additionally the relative performance of the various data structures may vary across computer architectures, compilers and operating systems. In this dissertation, all implementations are based on the most recent references available, and insights gained continuously have enabled continuously optimized implementations of the evaluated data structures. For this reason the relative performance of the evaluated data structures is also not completely identical to the results reported in [48], however the main conclusion remains the same.

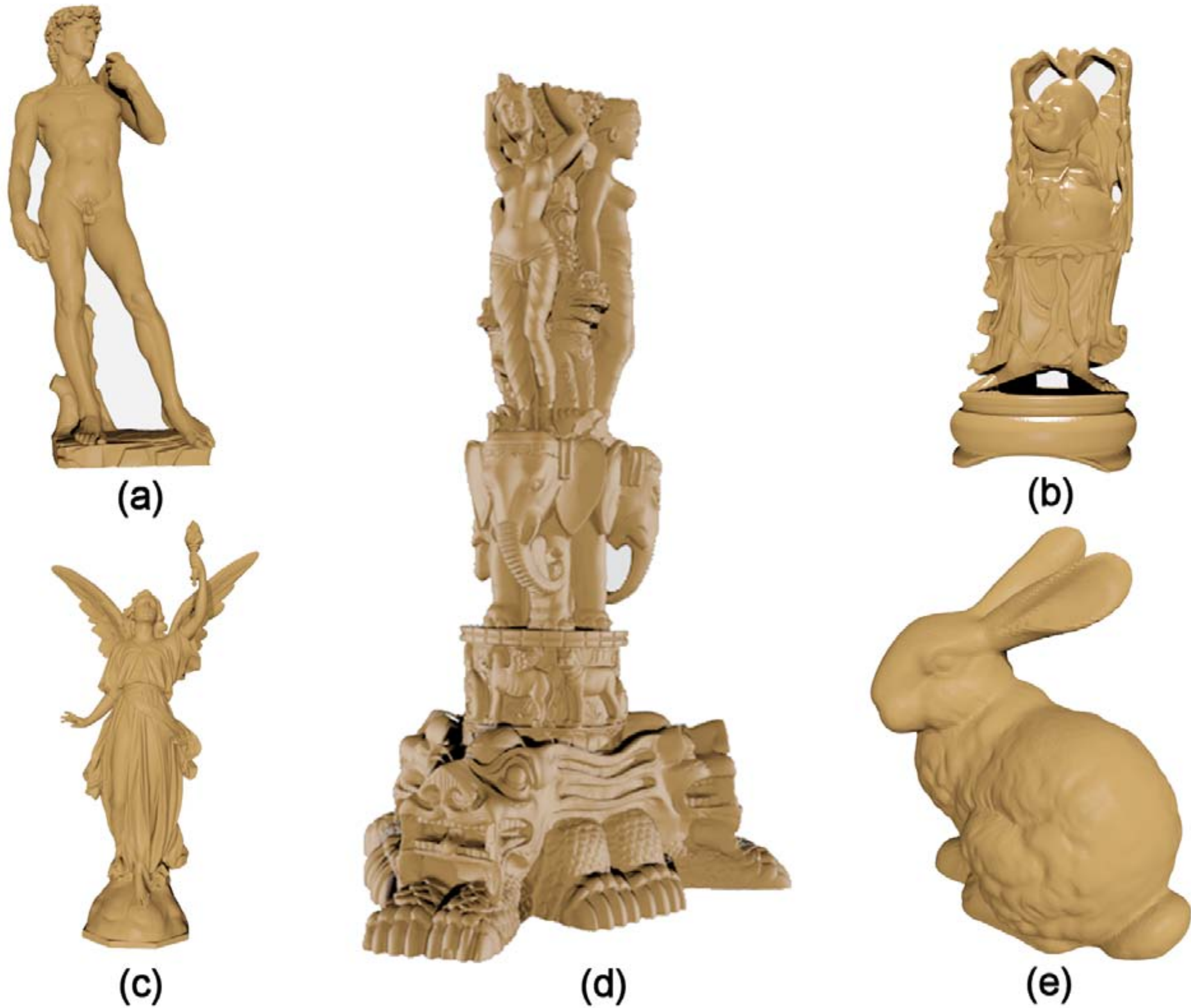


Figure 7.1: 3D level set models used for performance evaluations. (a) The **David** (b) The Happy **Buddha** (c) The **Lucy** Angel (d) The **Thai Statuette** (e) The Stanford **Bunny**. The terms in boldface will be used in the tables in this chapter as a shorthand notation for the corresponding models. All models courtesy of the Stanford Scanning Repository.

We focus in this chapter on performance evaluations in the configuration listed in table 7.1, and leave for future work to investigate the robustness of the results across different architectures, compilers and operating systems. We have implemented all test programs in C++ and used the Visual Studio 2005 version 8.0 compiler with maximal optimization enabled. The level set simulation and ray tracing code utilized in our benchmarks is *templated*[1] with respect to the underlying grid representation, meaning that identical and equally optimized simulation code is used for the tests of all data structures. In particular only the implementations of the underlying data structure and associated operations such as sequential access, random access and narrow band rebuild differ between the tests.

We have evaluated the following data structure implementations:

[1]Templated in the C++ meaning of the word.

Processor	AMD Athlon 64
Clock rate	2.41 GHz
Operating system	Windows XP Pro
L1 data cache size	64 KB
L1 instruction cache size	64 KB
L1 line size	64 B
L1 associativity	2-way
L1 data latency	1.25
L2 data cache size	1 MB
L2 line size	64 B
L2 associativity	16-way
L2 data latency	5.04
Physical memory	2 GB
Physical memory latency	52.6
Physical memory random latency	113.9

Table 7.1: The hardware and operating system configurations used for the performance evaluations in this chapter. Latency measurements are reported in nanoseconds and were found using LMbench 3.0 [90] (see also http://www.bitmover.com/lmbench/).

- **Peng I:** The narrow band level set method of Peng *et al.* [120]. This method rebuilds the narrow band by visiting the entire dense uniform grid, hence the time complexity is $O(L^3)$, where L is the side length of the enclosing grid. In addition to storing the dense uniform grid, mask and coordinate arrays are also stored.

- **Peng II:** The narrow band level set method of Peng *et al.* [120] combined with the method for rebuilding the narrow band proposed by Nilsson *et al.* [108]. The method for rebuilding the narrow band visits only the grid points contained in the union between the old and the new narrow bands, hence it has a time complexity that is linear in the number of grid points in the narrow band: $O(M_N)$.

- **Octree I:** An octree that stores only the grid points in the narrow band. The nodes of the octree are defined as:

```
struct OctreeCell
{
   OctreeCell *parent;
   union
   {
      OctreeCell *children;
      float *values;
   } u;
};
```

Note that the octree nodes at the finest level are omitted and the octree nodes at the second-finest level instead point directly to the values. This saves the storage required by

the finest level of refinement in the octree and at the same time makes the depth of the octree one less, hence making random and neighbor access faster. Notice also that values are referenced by a single pointer. This saves the additional storage of one pointer per grid point in the narrow band (in contrast to Octree II below). The implementation uses the algorithm of Stolte *et al.* [138] to traverse the octree sequentially and the neighbor access algorithms of Frisken *et al.* [38]. Octree cells are allocated in a memory pool [92] to speed up allocation and deallocation and to increase the cache coherency. The octree is constructed and rebuilt in time $O(M_N \log M_N)$.

- **Octree II:** An octree that stores only the grid points in the narrow band. The nodes are defined as:

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

  Similar to and with the same benefits as Octree I, the octree nodes at the finest level of Octree II are omitted and the octree nodes at the second-finest level instead point directly to values. In contrast to Octree I, the values are not referenced by a single pointer in the octree node. This adds storage but appears to improve the random access speed. The remaining properties of Octree II are identical to those of Octree I.

- **RLE Sparse:** The RLE Sparse Level Set data structure of Houston *et al.* [50]. My implementation is based on the details given in their technical sketch as well as on personal correspondence with the authors.

- **H-RLE:** The Hierarchical Run Length Encoded (H-RLE) Grid described in chapter 6.

- **DT-Grid:** The Dynamic Tubular Grid (DT-Grid) described in chapter 5.

- **DT-Grid NR:** The Dynamic Tubular Grid (DT-Grid) described in chapter 5, but with only one level of encoding (NR = Non Recursive). Hence in 3D the `proj2D` constituent is represented as a dense uniform grid instead of a 2D DT-Grid.

7.2 Evaluation of Level Set Simulation Performance

In this section we evaluate the performance of the level set representations introduced in the previous section for level set simulations utilizing sequential access alone. We consider the following three model problems:

- **Level Set Erosion:** As the first model problem we consider erosion of the Thai Statuette in which the surface is propagated inwards in the normal direction with unit speed. The

corresponding level set equation is given by

$$\frac{\partial \phi}{\partial t} - |\nabla \phi| = 0 \tag{7.1}$$

where ϕ is the level set function. Equation 7.1 is hyperbolic.

- **Mean Curvature Flow:** As the second model problem we evaluate mean curvature flow on the Stanford Bunny. Geometrically mean curvature flow corresponds to a smoothing of the surface and the corresponding level set equation is

$$\frac{\partial \phi}{\partial t} - \kappa|\nabla \phi| = 0 \tag{7.2}$$

where κ is the mean curvature. Due to the κ term, equation 7.2 is parabolic.

- **Level Set Advection:** As the third model problem we consider advection on the Stanford Bunny in a constant velocity field, $\mathbf{V}$. The corresponding level set equation is

$$\frac{\partial \phi}{\partial t} + \mathbf{V} \cdot \nabla \phi = 0 \tag{7.3}$$

Equation 7.3 is hyperbolic.

We consider each of these model problems using two discretizations differing in the order of numerical accuracy. In both of the two discretizations, the term $\kappa|\nabla \phi|$ appearing in the parabolic equation 7.2 is discretized identically. In particular $|\nabla \phi|$ is discretized using second order accurate central differences, and the curvature is computed using second order accurate finite differences as well. The term $|\nabla \phi|$ appearing in the hyperbolic equations are evaluated according to Godunov's scheme. The details of the two different discretizations are:

- **Low order accurate method:** The spatial derivatives are computed using first order accurate one-sided upwind finite differences. In time a first order accurate forward Euler discretization is used. We use a narrow band radius of $\gamma = 3\Delta x$.
- **High order accurate method:** The spatial derivatives are computed using three-fifth order accurate WENO finite differences adhering to an upwind scheme. In time a third order accurate TVD Runge Kutta discretization is used. We use a narrow band radius of $\gamma = 6\Delta x$.

In each step of the simulation, the level set equation in question is first solved in the γ tube of the narrow band. Next the reinitialization PDE (see equation 2.5) is solved in the entire narrow band three times, *i.e.* a fixed number of times. At the boundary of the narrow band we use the clamped $\pm\gamma$ signed distance field values outside the narrow band. Finally the narrow band is rebuilt and the next simulation step begins. In each step of the simulation we take the maximal time-step allowed for a stable simulation, and a total of 200 simulation steps are taken in each test.

As initial condition for the simulations we consider level set data scan converted from the Stanford Bunny and the Thai Statuette shown in figure 7.1. We sample the models at various different resolutions to ensure that our evaluations cover cases ranging from the situation where the level set data fits into the L2 cache to situations where the level set data takes up a significant

part of main memory. In particular the models have been sampled at the following resolutions [2]:

- **Stanford Bunny: R0** is $24x24x20$, **R1** is $56x55x44$, **R2** is $120x119x94$, **R3** is $248x246x193$, **R4** is $504x500x392$.
- **Thai Statuette: R0** is $26x38x22$, **R1** is $58x93x50$, **R2** is $122x200x105$, **R3** is $250x416x216$, **R4** is $506x847x437$.

The evaluations are presented as follows: For each model problem, discretization, sampling resolution and data structure we evaluate the average time-usage per simulation step (reported in seconds) as well as the minimal and maximal memory usage (reported in MB) of a single grid instance during the course of the 200 simulation steps taken in each test. Average time and memory usage are reported in two different tables for each test due to layout considerations. Each test is run approximately five times and the average time reported here is the median of the average times found in these runs. Note that for the Peng methods, the memory usage includes the dense uniform grid, mask as well as index lists employed by the method. Furthermore the memory usage reported for all methods includes the storage occupied by the entire narrow band, *i.e.* the $\gamma + \Delta x$ tube. Finally note that for all tests, the peak memory will be roughly doubled during the simulation, since a double buffering approach is used.

Tables 7.3 to 7.14 show the results of the evaluations. Regarding memory usage the general tendency is that DT-Grid requires the least storage followed by H-RLE, RLE Sparse, the octrees and the Peng methods. Furthermore it is evident that for model problems 7.1 and 7.2, where the surface area tends to zero, the storage requirements of all evaluated data structures, except the dense uniform grid employed by the Peng methods. also tend to zero. The reduction in memory requirements gained by using the DT-Grid or H-RLE over the Peng methods varies. However, in the extreme case of model problem 7.1 at resolution **R4**, the minimal memory usage of the DT-Grid is 0.2% of the minimal memory usage of the Peng methods.

The evaluations of average time per iteration are not as clear-cut as the storage requirements. However, the general tendency is that the DT-Grid performs faster than the remaining methods, including the Peng methods and the H-RLE. In particular the DT-Grid appears to be faster than Peng II except in three cases, all at the lowest resolution. Intuitively this may be somewhat surprising if one would expect the Peng II method to be faster than the alternatives whenever the grids fit entirely into either the L1 or L2 cache. However, this reasoning implicitly assumes that the simulation is CPU limited. In particular we have found three explanations of the fact that the DT-Grid is in most cases faster than Peng II:

- **Peng II requires more storage than the DT-Grid:** We hypothesize that this fact partly explains that the DT-Grid is in some cases faster than Peng II even though all data is located in the cache. Using the AMD CodeAnalyst tool we found that even at low resolutions, the CPU is stalling, waiting for data from the L1 cache, during level set simulations. Hence the performance is bounded from below by the L1 access time as opposed to being CPU bound. Table 7.2 shows measurements of L1 and L2 accesses and misses using the AMD CodeAnalyst tool. As is evident from these measurements, the DT-Grid has fewer accesses (and misses) to the cache hierarchy than Peng II, even at resolution **R0**. Since the running-time is bounded by the L1 cache access time, it is very

[2] These resolutions are the ones used with the low order accurate method. The resolutions used with the high order accurate method have a slightly bigger bounding box since they employ a narrow band radius of $\gamma = 6\Delta x$

likely that this explains that DT-Grid appears to be faster than Peng II. We hypothesize that the reason that DT-Grid has fewer accesses to the cache hierarchy is that it requires less storage, even at low resolution. The operations on the DT-Grid all require more CPU cycles than the equivalent Peng II operations, however the number of non-idle CPU cycles is not the performance bottleneck. Some local variables may also reside in the L1 cache, and in particular the DT-Grid accesses more local variables than the Peng II method. However, it all amounts to accesses to the cache hierarchy, and as can be seen from table 7.2, the total number of accesses to the local variables and data residing in the cache is larger for the Peng II method.

- **Narrow band access-scattering:** At resolutions where all data does not reside in the cache another effect degrades the performance of the Peng II method compared to the DT-Grid. In particular, the Peng II method does not exploit spatial locality as well as the DT-Grid, because the level set data is stored in a dense uniform grid. Hence two consecutive grid points in the Peng II narrow band data structure may be far from each other in physical memory and hence trigger a cache miss when accessed. This effect can be seen by comparing the measurements at resolutions **R0** and **R4** in table 7.2. Note that similar to the average time per iteration, the cache coherency of the Peng II method has degraded compared to the DT-Grid and the measurements at resolution **R0**. In addition to this effect, the $O(M_N)$ rebuild method used by Peng II has the property that accesses to memory are becoming increasingly scattered over time as the simulation progresses. Figure 7.2 illustrates the cache coherency of the narrow band of a simulation at resolution 256^3 that simply translates the level set by the constant velocity field $\mathbf{V} = (1, 1, 1)$. In particular the cache coherency is shown initially and after 200 iterations. The cache coherency degrades both for a low order and a high order accurate method. However, the impact on performance over the course of 200 iterations is largest for the low order accurate method, most likely because the access time comprises a larger part of the total simulation time in this case. In particular we observed at 12% drop in performance for the low order method and a 4% drop in performance for the high order method over the course of the 200 iterations. Accessing all grid points in the narrow band in random order incurs drops in performance of 123% and 81% respectively for the low order and the high order accurate methods. Hence simulations running for longer periods of time may observe a significant drop in performance. An option during narrow band rebuild is to sort either all grid points in the narrow band or the new grid points entering the narrow band such that the most cache coherent access possible with a dense uniform grid is maintained throughout the simulation. However, preliminary tests show that sorting every iteration introduces an overhead in performance.

- **Swapping due to insufficient main memory:** At high resolutions where the storage requirements of the Peng methods exceed the physical memory available, the operating system starts swapping memory to and from disk during the simulation. Relying on the operating system for this task degrades performance, and we will return to this in part III of this dissertation. Swapping effects are not evident in the evaluations performed in this subsection, except in a single case where it is not possible to execute the Peng I method. However, in section 7.3.2 where we evaluate the performance of a ray tracing application on a computer with 1 GB of main memory, these effects are clearly visible.

Besides the tendency that the DT-Grid in general performs faster than the remaining data structures, the H-RLE method seems to perform comparable to the Peng II method. There is a clear tendency for H-RLE to be faster than Peng II for low order accurate discretizations at higher resolutions. However, except for the cases of all resolutions in model problem 7.3 and the highest resolution in model problem 7.1 where H-RLE is faster than Peng II, Peng II appears to be faster than H-RLE for the high order accurate discretization. The H-RLE appears to be slower than the DT-Grid, and most likely this can be explained by the fact that the H-RLE requires more storage and more accesses to local variables than the DT-Grid. In particular more local variables and data accesses are required by the H-RLE to iterate over the runs than required by the DT-Grid to iterate over the projection columns. The octree methods have simulation time complexities in the order of $O(M_N \log M_N)$ and only perform faster than the Peng I method which has a time complexity of $O(L^3)$.

The observations made in this section give rise to a number of properties that must be exhibited by a level set data structure in order to be faster than the DT-Grid. In particular such a data structure should ensure temporal locality, spatial locality, a small memory footprint and access to as few local variables as possible. In this way the number of accesses to the L1 cache is lowered, and thereby the performance improved.

	R0					**R4**				
	Avg Time	L1 Data Acc	L1 Data Miss	L2 Data Acc	L2 Data Miss	Avg Time	L1 Data Acc	L1 Data Miss	L2 Data Acc	L2 Data Miss
DT-Grid	0.0035	1.0000	1.0000	1.0000	0.0000	2.6908	1.0000	1.0000	1.0000	1.0000
Peng II	0.0036	1.0250	3.0000	2.6667	0.0000	3.7814	1.0686	11.6065	8.0440	48.9056

Table 7.2: Cache measurements using the AMD CodeAnalyst Tool for model problem 7.1 using the low order accurate discretization and the Thai Statuette as initial level set data. Internally in each column, the number of accesses and misses are divided by the corresponding number of accesses/misses for the DT-Grid.

Grid\Resolution	**R0** Avg Time	**R1** Avg Time	**R2** Avg Time	**R3** Avg Time	**R4** Avg Time
DT-Grid	0.0035	0.0184	0.0940	0.4017	2.6908
H-RLE	0.0048	0.0225	0.1058	0.4770	3.1641
Octree I	0.0085	0.0487	0.2469	1.1522	8.3792
Octree II	0.0085	0.0478	0.2508	1.1677	8.6671
Peng I	0.0120	0.0947	0.7806	6.0718	41.9458
Peng II	0.0036	0.0189	0.1060	0.5225	3.7814
RLE	0.0060	0.0312	0.1542	0.7994	5.0189

Table 7.3: Average time usage of model problem 7.1 using the low order accurate discretization and the Thai Statuette as initial level set data.

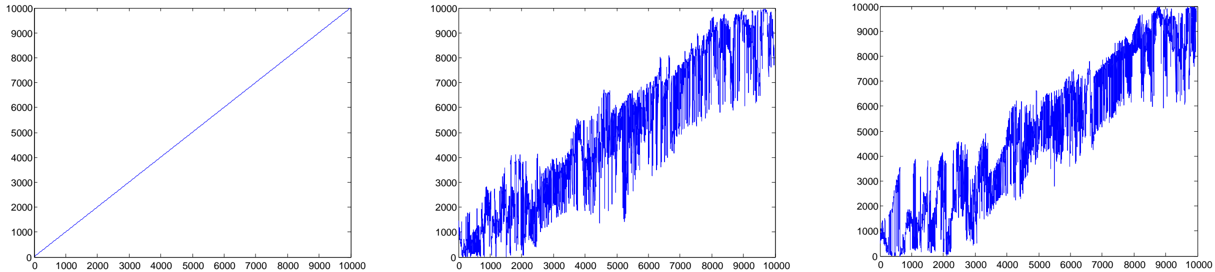


Figure 7.2: This figure shows how entries in the list of narrow band grid points map to addresses in memory using the method of Peng *et al.* [120] combined with the sparse rebuild of Nilsson *et al.* [108]. The data is normalized in such a way that 10000 samples of the memory locations (lowpass-filtered) of consecutive grid points in the narrow band are mapped to each integer in the range 0 to 9999 by retaining the order that the locations have in physical memory. The leftmost image shows the situation before the simulation begins. Since the level set is initialized in the lexicographic order consistent with stride-1 access, the graph is just a straight line, corresponding to the most memory-coherent narrow band access. The middle and rightmost image show the narrow band mapping after 200 iterations for a high-order and a low-order accurate simulation respectively.

Grid\Resolution	**R0**		**R1**		**R2**		**R3**		**R4**	
	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem
DT-Grid	0.05	0.00	0.31	0.00	1.75	0.00	8.59	0.00	38.33	1.76
H-RLE	0.05	0.00	0.34	0.00	1.87	0.00	9.14	0.00	40.78	1.89
Octree I	0.06	0.00	0.43	0.00	2.42	0.00	12.10	0.00	53.98	2.49
Octree II	0.10	0.00	0.68	0.00	3.91	0.00	19.42	0.00	86.60	3.94
Peng I	0.22	0.16	1.95	1.54	15.66	13.33	123.26	111.76	963.26	914.31
Peng II	0.22	0.16	1.95	1.54	15.66	13.33	123.26	111.76	963.26	914.31
RLE	0.06	0.00	0.37	0.00	2.01	0.00	9.76	0.00	43.42	1.99

Table 7.4: Memory usage of model problem 7.1 using the low order accurate discretization and the Thai Statuette as initial level set data.

Grid\Resolution	R0 Avg Time	R1 Avg Time	R2 Avg Time	R3 Avg Time	R4 Avg Time
DT-Grid	0.0591	0.3272	1.6627	7.9290	54.2011
H-RLE	0.0681	0.3741	1.8910	8.9763	61.6142
Octree I	0.1071	0.6165	3.1071	14.9504	118.9552
Octree II	0.1016	0.5875	2.9944	14.3973	118.2406
Peng I	0.0748	0.4460	2.6848	15.3447	NP
Peng II	0.0580	0.3313	1.7685	8.7615	62.3916
RLE	0.0714	0.4308	2.5301	13.0160	59.9009

Table 7.5: Average time usage of model problem 7.1 using the high order accurate discretization and the Thai Statuette as initial level set data.

Grid\Resolution	R0		R1		R2		R3		R4	
	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem
DT-Grid	0.05	0.00	0.38	0.00	2.30	0.00	12.31	0.00	58.03	2.46
H-RLE	0.05	0.00	0.40	0.00	2.41	0.01	12.81	0.00	60.34	2.59
Octree I	0.07	0.01	0.50	0.01	3.07	0.01	16.54	0.01	78.86	3.33
Octree II	0.10	0.01	0.83	0.01	5.15	0.01	27.72	0.01	131.51	5.51
Peng I	0.21	0.15	1.98	1.47	16.29	13.10	127.75	110.53	NP	NP
Peng II	0.21	0.15	1.98	1.47	16.29	13.10	127.75	110.53	988.23	910.41
RLE	0.06	0.00	0.42	0.00	2.53	0.00	13.41	0.00	62.91	2.67

Table 7.6: Memory usage of model problem 7.1 using the high order accurate discretization and the Thai Statuette as initial level set data.

Grid\Resolution	R0 Avg Time	R1 Avg Time	R2 Avg Time	R3 Avg Time	R4 Avg Time
DT-Grid	0.0030	0.0325	0.2174	1.1040	4.8327
H-RLE	0.0037	0.0378	0.2616	1.2913	5.6066
Octree I	0.0075	0.0926	0.6484	3.4012	16.0563
Octree II	0.0075	0.0914	0.6588	3.4281	16.5954
Peng I	0.0072	0.0716	0.6381	4.3819	25.8788
Peng II	0.0030	0.0328	0.2372	1.4188	6.2713
RLE	0.0051	0.0555	0.3690	1.8818	9.1996

Table 7.7: Average time usage of model problem 7.2 using the low order accurate discretization and the Stanford Bunny as initial level set data.

Grid\Resolution	R0		R1		R2		R3		R4	
	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem
DT-Grid	0.03	0.00	0.23	0.12	1.20	0.90	5.47	4.93	23.20	22.31
H-RLE	0.04	0.00	0.25	0.13	1.30	0.98	5.90	5.33	25.00	24.08
Octree I	0.05	0.00	0.32	0.17	1.67	1.26	7.64	6.95	32.69	31.49
Octree II	0.07	0.00	0.50	0.27	2.66	2.00	12.14	11.01	51.82	49.91
Peng I	0.13	0.09	1.09	0.95	8.62	8.22	66.01	65.32	512.57	511.44
Peng II	0.13	0.09	1.09	0.95	8.62	8.22	66.01	65.32	512.57	511.44
RLE	0.04	0.00	0.26	0.14	1.36	1.02	6.18	5.57	26.20	25.19

Table 7.8: Memory usage of model problem 7.2 using the low order accurate discretization and the Stanford Bunny as initial level set data.

Grid\Resolution	R0 Avg Time	R1 Avg Time	R2 Avg Time	R3 Avg Time	R4 Avg Time
DT-Grid	0.0421	0.4261	3.3881	17.9380	81.4784
H-RLE	0.0552	0.4845	3.8277	20.3261	92.4673
Octree I	0.0751	0.8636	6.7141	35.9063	187.4805
Octree II	0.0705	0.8288	6.4946	34.6557	185.5037
Peng I	0.0504	0.5273	4.4804	24.9956	107.4884
Peng II	0.0401	0.4412	3.6845	19.4587	89.4366
RLE	0.0520	0.6018	3.8757	25.5550	117.4520

Table 7.9: Average time usage of model problem 7.2 using the high order accurate discretization and the Stanford Bunny as initial level set data.

Grid\Resolution	R0		R1		R2		R3		R4	
	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem
DT-Grid	0.03	0.01	0.28	0.15	1.66	1.33	8.10	7.48	35.70	34.60
H-RLE	0.04	0.01	0.29	0.17	1.75	1.41	8.52	7.88	37.46	36.34
Octree I	0.04	0.01	0.36	0.23	2.23	1.86	10.97	10.27	48.61	47.30
Octree II	0.07	0.01	0.59	0.37	3.69	3.06	18.16	16.96	80.35	78.17
Peng I	0.12	0.08	1.13	0.99	9.20	8.80	69.55	68.79	529.52	528.18
Peng II	0.12	0.08	1.13	0.99	9.20	8.80	69.55	68.79	529.52	528.18
RLE	0.04	0.01	0.30	0.16	1.81	1.45	8.78	8.11	38.63	37.44

Table 7.10: Memory usage of model problem 7.2 using the high order accurate discretization and the Stanford Bunny as initial level set data.

Grid\Resolution	**R0** Avg Time	**R1** Avg Time	**R2** Avg Time	**R3** Avg Time	**R4** Avg Time
DT-Grid	0.0028	0.0332	0.2082	1.0061	4.3715
H-RLE	0.0035	0.0398	0.2498	1.2028	5.1950
Octree I	0.0069	0.0923	0.5806	2.8306	13.1044
Octree II	0.0067	0.0903	0.5931	2.8741	13.7486
Peng I	0.0615	0.2907	1.5337	8.2898	42.7709
Peng II	0.0027	0.0364	0.2436	1.3742	6.2306
RLE	0.0046	0.0597	0.3705	1.8998	9.0410

Table 7.11: Average time usage of model problem 7.3 using the low order accurate discretization and the Stanford Bunny as initial level set data.

Grid\Resolution	**R0**		**R1**		**R2**		**R3**		**R4**	
	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem
DT-Grid	0.03	0.00	0.23	0.16	1.20	1.03	5.47	5.15	23.20	22.66
H-RLE	0.04	0.00	0.25	0.17	1.30	1.12	5.90	5.57	25.00	24.44
Octree I	0.05	0.00	0.32	0.21	1.68	1.43	7.66	7.21	32.69	31.89
Octree II	0.08	0.00	0.51	0.34	2.67	2.26	12.16	11.43	51.83	50.53
Peng I	1.25	1.21	5.63	5.53	27.67	27.42	151.06	150.62	925.06	924.27
Peng II	1.25	1.21	5.63	5.53	27.67	27.42	151.06	150.62	925.06	924.27
RLE	0.04	0.00	0.26	0.18	1.36	1.17	6.18	5.82	26.20	25.60

Table 7.12: Memory usage of model problem 7.3 using the low order accurate discretization and the Stanford Bunny as initial level set data.

Grid\Resolution	**R0** Avg Time	**R1** Avg Time	**R2** Avg Time	**R3** Avg Time	**R4** Avg Time
DT-Grid	0.0380	0.4720	3.3888	17.7175	80.0852
H-RLE	0.0478	0.4787	3.6926	19.3733	87.7248
Octree I	0.0682	0.7827	5.9242	31.3724	166.0236
Octree II	0.0647	0.7525	5.7529	30.3128	163.9544
Peng I	0.1157	0.8631	5.7173	29.3470	126.2994
Peng II	0.0497	0.5528	4.1134	20.9014	98.3220
RLE	0.0443	0.3881	3.8393	21.8846	99.1223

Table 7.13: Average time usage of model problem 7.3 using the high order accurate discretization and the Stanford Bunny as initial level set data.

Grid\Resolution	**R0**		**R1**		**R2**		**R3**		**R4**	
	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem	Max Mem	Min Mem
DT-Grid	0.03	0.00	0.28	0.13	1.66	1.22	8.10	6.99	35.70	32.57
H-RLE	0.04	0.00	0.29	0.14	1.75	1.29	8.52	7.36	37.46	34.27
Octree I	0.05	0.00	0.37	0.18	2.24	1.66	10.98	9.51	48.61	44.50
Octree II	0.07	0.00	0.61	0.29	3.70	2.73	18.18	15.67	80.37	73.26
Peng I	1.25	1.24	5.71	5.65	28.37	28.29	155.07	154.97	943.94	943.69
Peng II	1.25	1.24	5.71	5.65	28.37	28.29	155.07	154.97	943.94	943.69
RLE	0.04	0.00	0.30	0.14	1.81	1.33	8.78	7.58	38.63	35.24

Table 7.14: Memory usage of model problem 7.3 using the high order accurate discretization and the Stanford Bunny as initial level set data.

7.3 Evaluation of Random Access Performance

In this section we evaluate the performance of two applications requiring the use of random and neighbor access operations. First section 7.3.1 evaluates the use of the fast marching method in a level set simulation and next we consider ray tracing in section 7.3.2.

7.3.1 Evaluation of Fast Marching

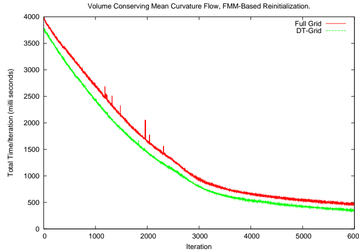


Figure 7.3: The total simulation time (in milliseconds) for volume conserving mean curvature flow.

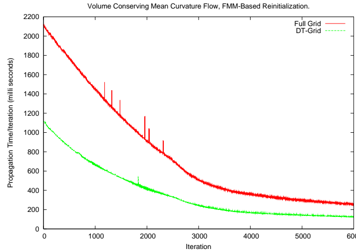


Figure 7.4: The simulation time (in milliseconds) spent solving the level set equation 7.3.

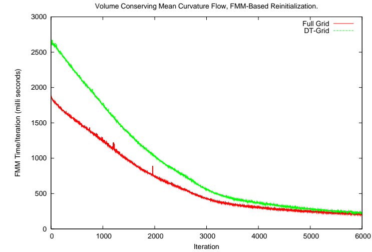


Figure 7.5: The reinitialization time (in milliseconds) spent by the fast marching method.

Here we consider the model problem of volume conserving mean curvature flow [120] which geometrically corresponds to a smoothing of the surface that conserves its enclosed volume. The corresponding level set equation is

$$\frac{\partial \phi}{\partial t} - (\kappa - \bar{\kappa})|\nabla \phi| = 0 \tag{7.4}$$

where κ is the mean curvature, and $\bar{\kappa}$ is the average mean curvature of the initial shape, in this case the extruded spiral depicted in figure 5.3 in chapter 5. Referring to equation 7.4 we discretize the spatial parabolic term ($\kappa|\nabla\phi|$) using second order accurate central differences and the spatial hyperbolic term ($\bar{\kappa}|\nabla\phi|$) using fifth order HJ-WENO. Temporally we employ a third order accurate TVD Runge Kutta method. We reinitialize ϕ to a signed distance field by solving the Eikonal equation using a first order accurate fast marching method [131]. In contrast to the PDE based reinitialization that we employed in the previous tests, the fast marching method does not visit the narrow band sequentially, recall the brief explanation in chapter 2. This means that for the solution of the Eikonal equation a large amount of random and neighbor access operations are required. Figures 7.3, 7.4 and 7.5 compare the performance of this benchmark on a dense uniform grid utilizing the Peng narrow band method for level set propagation to the performance on the DT-Grid. On the dense uniform grid, the narrow band is rebuilt simultaneously with solving the Eikonal equation in time $O(M_N \log M_N)$. The rebuild procedure is implemented by including into the narrow band the grid points whose computed distance values are numerically less than γ. As can be seen from figure 7.3 the total simulation time, including both the solution of the level set equation 7.4 and the Eikonal equation, is slightly better on the DT-Grid than on a dense uniform grid. However, due to the utilization of the fast marching method for reinitializing the level set function, the picture is somewhat different from the previous tests as evident from figure 7.5. In fact, the time spent on reinitialization is less on the dense uniform grid, but due to the fact that the solution of the level set equation is in this

case faster on the DT-Grid as depicted in figure 7.4, the *total* simulation time remains lower on the DT-Grid. However, for fast marching based reinitialization alone, the dense uniform grid is preferable as long as the grid sizes are small enough to fit in physical memory.

7.3.2 Evaluation of Level Set Ray Tracing

Grid\Resolution	R0	R1	R2	R3	R4	R5	R6	R7	R8
	Time	Time	Time	Time	Time	Time	Time	Time	Time
DT-Grid NR	11.58	25.31	39.27	48.89	57.16	65.69	109.88	230.45	547.00
DT-Grid, Binary Search	11.98	27.95	44.14	54.63	62.70	71.97	113.63	215.09	472.34
DT-Grid, Linear Search	11.72	27.55	42.89	53.06	61.59	69.97	112.75	214.75	476.77
H-RLE	11.67	28.98	45.63	57.05	65.94	74.73	117.44	211.30	453.78
Octree I	14.22	32.67	51.09	64.28	73.42	85.03	131.28	216.75	385.59
Octree II	13.78	32.14	50.00	62.47	71.31	82.22	126.16	209.64	561.25
Dense Uniform Grid	13.83	26.88	39.14	48.67	56.80	65.38	119.06	NP	NP
RLE	11.70	26.61	41.64	51.67	59.73	68.28	108.66	201.72	460.58

Table 7.15: Raytracing test of the David statue at resolutions **R0**(11x9x19), **R1**(38x24x83), **R2**(88x53x206), **R3**(138x82x329), **R4**(188x110x451), **R5**(238x139x574), **R6**(488x284x1187), **R7**(988x573x2413), **R8**(1988x1150x4865). Timings are given in seconds. NP indicates that the test was not possible due to insufficient memory. All models store a clamped signed distance field and have a narrow band radius of three grid points.

In this sub-section we evaluate performance by means of an application that requires a large amount of random access operations. In particular we compare the ray tracing times for various level set models and grid representations [3].

The tests were performed on the architecture shown in table 7.1 with 1GB of main memory. Tables 7.16 and 7.15 list the rendering times for various models, and we will briefly outline the general tendencies and draw conclusions from these.

At most grid sizes a dense uniform grid approach appears to perform best. This can be attributed to its constant time random access operation. All other data structures evaluated here have logarithmic access times. For larger models however ray tracing on a dense uniform grid either becomes slower than most or all grid representations due to memory-to-disk-swapping or impossible (NP) due to violating the virtual memory limits.

At low and intermediate grid resolutions, the non-recursive DT-Grid and the RLE in general perform faster than the other logarithmic time access data structures. This is due to the fact that DT-Grid NR and RLE only perform logarithmic search in a single projection column and scanline respectively. At higher resolutions the performance of DTGrid NR and RLE drops. Our hypothesis is that this is mainly due to the 2D dense uniform acceleration structure stored by these data structures. At higher resolutions this acceleration structure incurs less cache coherency, which is supported by the cache measurements given in table 7.17.

[3]While one could evaluate the performance of the random access operation by accessing grid points in a truly random or sequential fashion, these tests do not seem particularly relevant. The reason being that if an application requires access to the grid sequentially an iterator is a better choice, and applications that access the grid in a truly random manner are probably rare.

Grid\Resolution	R0 Time	R1 Time	R2 Time	R3 Time	R4 Time	R5 Time	R6 Time	R7 Time	R8 Time
DT-Grid NR	41.44	74.50	101.11	122.33	143.97	167.91	299.38	652.95	1724.13
DT-Grid, Binary Search	46.08	85.17	115.27	137.06	160.50	182.13	293.34	521.02	1082.77
DT-Grid, Linear Search	47.50	84.91	111.52	132.69	155.51	176.55	285.08	511.00	1061.08
H-RLE	50.41	93.20	123.45	148.52	171.25	195.41	320.36	596.14	1364.55
Octree I	51.63	100.11	132.61	160.16	182.17	210.24	331.44	569.92	1157.58
Octree II	50.31	96.83	129.75	155.13	176.41	204.06	320.01	545.91	NP
Dense Uniform Grid	41.47	71.59	95.95	118.08	138.19	161.50	820.30	NP	NP
RLE	45.52	81.41	107.97	129.19	151.81	174.09	292.36	561.41	1363.34

Table 7.16: Raytracing test of the Happy Buddha at resolutions **R0**(19x40x20), **R1**(44x101x45), **R2**(94x222x95), **R3**(144x344x145), **R4**(194x476x195), **R5**(244x588x245), **R6**(494x1196x495), **R7**(994x2414x996), **R8**(194x4849x1997). Timings are given in seconds. NP indicates that the test was not possible due to insufficient memory. All models store a clamped signed distance field and have a narrow band radius of three grid points.

Grid\Resolution	R0				R8			
	L1 Data Acc	L1 Data Miss	L2 Data Acc	L2 Data Miss	L1 Data Acc	L1 Data Miss	L2 Data Acc	L2 Data Miss
DT-Grid NR	0.7902	0.8669	0.8990	2.0561	0.9579	1.8374	1.8412	6.8878
DT-Grid, Linear Search	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Table 7.17: Cache measurements in raytracing test of the Happy Buddha at resolutions **R0**(19x40x20) and **R8**(194x4849x1997). The numbers in each column are normalized to the measurement for the DT-Grid. At high resolution, the relative number of cache misses increases for the DT-Grid NR data structure. We hopethesize that this at least partly explains its degradation in performance at higher resolutions.

The general tendency among the other data structures is that DT-Grid performs better than octrees, although there are a few exceptions. Cache measurements show that the octree has about 20% more accesses to the L1 cache than the DT-Grid at the highest resolution. L1 accesses cover both access to data and local variables, and we hypothesize that the slower performance is caused by the octree's search methods that need to access more data than the DT-Grid. The performance of the H-RLE is a hard to characterize. In most cases it performs better than the octree, and in some cases it performs better than the DT-Grid. In a few cases it performs worse than the octree. Similarly the performance of the RLE grid is hard to characterize generally at higher resolutions.

To sum up: For random access, a dense uniform grid seems to be preferable at resolutions that fit into memory. At higher resolutions the performance of the dense uniform grid drops, and the relative performance of the other data structures varies. At most resolutions the DT-Grid and H-RLE perform better than the octree data structures.

7.4 Evaluation of Storage Requirements for High Resolution Level Sets

In this section we consider the storage requirements of the various data structures for representing high resolution level set surfaces. We focus on two different methods of evaluation. In the first method we measure the number of *bits per grid point in the narrow band* required for storing the grid-topology. In the second method we measure the total storage requirements of the level set representation (*i.e.* values and topology).

Consider first table 7.18 which lists the number of bits per grid point in the narrow band required by the topology. The number of topology bits per narrow band grid point is computed by the following formula:

$$\frac{(\text{total memory - value memory}) \times 8}{M_3} \tag{7.5}$$

where *total memory* encompasses the total memory usage of the data structure in bytes, *value memory* measures the memory occupied by the values of the grid points in the narrow band in bytes and M_3 is the number of grid points in the narrow band. Note that figure 7.18 lists the number of topology bits for two different narrow band widths, $\gamma = 3$ and $\gamma = 5$ respectively. As can be seen from table 7.18, the efficiency of all representations increase when the narrow band is widened which is what one would intuitively expect. For all models the DT-Grid offers the best topology-storage efficiency followed by the non recursive DT-Grid and the H-RLE. Even though the asymptotic memory usage of the octree is proportional to the area of the surface, its inherent pointer structure leads to a relatively inefficient representation of the grid topology. The dense grid approaches are the least efficient. Note that in chapter 10 we introduce a method for compressing the topology of the DT-Grid that further increases the efficiency of the topology representation. In fact the DT-Grid topology compression method results in less than one bit per grid point in the narrow band (around 0.25 bits per grid point on average), while still allowing for relatively efficient level set simulations (although not as efficient as the original DT-Grid).

Table 7.19 lists the total memory usage of the data structures for the higher resolution models in table 7.18. Since all representations evaluated here store the values in the narrow band uncompressed the overall picture of table 7.19 is identical to that of table 7.18. However, to give an impression of the absolute difference measured in MB that the different representations of topology result in, we have included table 7.19. As can be seen the DT-Grid variants and the H-RLE represent the level set most efficiently although a bit of efficiency is sacrificed by the H-RLE due to its encoding of undefined regions. However, as already argued this results in a flexible encoding not shared by the DT-Grid. The dense grid representations require a prohibitively large amount of storage. For storing the Happy Buddha with $\gamma = 5$ for example, a dense grid representation including the additional data structures required by the narrow band method of Peng *et al.* [120] consumes 94GB of storage. On the DT-Grid the Happy Buddha with $\gamma = 5$ can be represented in 1.4GB when the most memory efficient optimization is applied. It should be noted that the *adaptive distance field* of Frisken *et al.* [39] probably would be more memory efficient than the representations listed here in some cases since it samples adaptively along the surface. However, as mentioned earlier, current octree-based level set implementations refine *uniformly* near the interface.

Model	Octree I	Octree II	Dense Grid	Dense Grid Incl Peng	DT-Grid	DT-Grid Non Recursive	H-RLE	RLE Sparse
Lucy, $\gamma = 3$								
$833 \times 487 \times 281$	24	54	867	1140	5.0/4.5/3.5/3.0	5.1	7.0	8.7
$3409 \times 1987 \times 1142$	25	55	3529	4492	5.0/4.6/3.5/3.0	5.1	7.1	8.7
Lucy, $\gamma = 5$								
$833 \times 487 \times 281$	18	49	521	707	3.0/2.7/2.1/1.8	3.0	4.2	5.2
$3409 \times 1987 \times 1142$	19	49	2112	2735	3.0/2.7/2.1/3.0	3.1	4.3	5.2
David, $\gamma = 3$								
$1186 \times 487 \times 283$	25	54	773	1023	5.2/4.7/3.6/3.1	4.8	6.7	7.3
$4864 \times 1987 \times 1149$	25	55	3157	4028	5.2/4.7/3.7/3.1	4.9	6.8	7.4
David, $\gamma = 5$								
$1186 \times 487 \times 283$	18	49	461	632	3.1/2.8/2.2/1.9	2.9	4.0	4.4
$4864 \times 1987 \times 1149$	19	49	1888	2456	3.1/2.8/2.2/1.9	2.9	4.1	4.5
Bunny, $\gamma = 3$								
$491 \times 487 \times 381$	25	54	830	1093	8.4/7.0/5.6/5.6	7.9	9.1	14
$1991 \times 1974 \times 1544$	25	54	3387	4315	8.3/6.9/5.5/5.5	7.8	9.0	14
Bunny, $\gamma = 5$								
$491 \times 487 \times 381$	19	49	493	673	5.1/4.2/3.4/3.4	4.8	5.5	8.5
$1991 \times 1974 \times 1544$	19	49	2027	2622	5.0/4.2/3.3/3.3	4.7	5.4	8.4
Buddha, $\gamma = 3$								
$1195 \times 494 \times 493$	25	54	739	980	7.5/6.4/5.1/4.9	6.8	9.3	12
$4848 \times 1996 \times 1993$	25	NP	3043	3887	7.5/6.4/5.1/4.9	6.8	9.3	12
Buddha, $\gamma = 5$								
$1195 \times 494 \times 493$	18	49	438	603	4.5/3.8/3.0/3.0	4.0	5.6	7.0
$4848 \times 1996 \times 1993$	18	NP	1818	2361	4.5/2.9/3.0/3.8	4.1	5.6	7.0

Table 7.18: This table demonstrates the efficiency of representing the topology for various level set data structures that store the values in the narrow band uncompressed. The reported numbers are the *number of topology bits per grid point in the narrow band*. The exact formula for computing this is given in the text. For the DT-Grid four numbers are presented due to the possible optimizations listed in chapter 5 (see page 53). The first number corresponds to the most general form of the data structure and the three others correspond to optimizations 1,1 + 2 and 3 respectively. The dense uniform grid is unable to represent the higher resolution models, but for this representation it is easy to compute the memory usage analytically. Note that *Dense Grid* corresponds to the storage required by the dense grid alone, whereas *Dense Grid Incl Peng* includes the mask and coordinate storage arrays of the narrow band method. Finally note that NP = Not Possible.

Model	Octree I	Octree II	Dense Grid	Dense Grid Incl Peng	DT-Grid	DT-Grid Non Recursive	H-RLE	RLE Sparse
Lucy, $\gamma = 3$								
$3409 \times 1987 \times 1142$	474	718	30.0GB	37.0GB	308/304/295/291	308	324	338
Lucy, $\gamma = 5$								
$3409 \times 1987 \times 1142$	696	1.10 GB	29.0GB	37.0GB	481/477/468/464	482	498	511
David, $\gamma = 3$								
$4864 \times 1987 \times 1149$	761	1.20GB	42.0GB	54.0GB	495/488/474/466	491	516	525
David, $\gamma = 5$								
$4864 \times 1987 \times 1149$	1.10GB	1.80GB	42.0GB	54.0GB	773/766/752/745	768	794	802
Bunny, $\gamma = 3$								
$1991 \times 1974 \times 1544$	386	586	23.0GB	29.0GB	273/264/255/254	270	278	311
Bunny, $\gamma = 5$								
$1991 \times 1974 \times 1544$	569	911	23.0GB	29.0GB	416/407/397/397	413	421	454
Buddha, $\gamma = 3$								
$4848 \times 1996 \times 1993$	1.40GB	NP	74.0GB	93.0GB	946/919/888/884	929	990	1.00GB
Buddha, $\gamma = 5$								
$4848 \times 1996 \times 1993$	2.00GB	NP	74.0GB	94.0GB	1.45/1.42/1.39/1.39 GB	1.43 GB	1.50 GB	1.55 GB

Table 7.19: This table demonstrates the memory usage of various level set representations for several high resolution models with narrow band widths of $\gamma = 3$ and $\gamma = 5$ respectively. The reported numbers are given in MB unless stated otherwise. For the DT-Grid four numbers are presented due to the possible optimizations listed in chapter 5 (see page 53). The first number corresponds to the most general form of the data structure and the three others correspond to optimizations 1,1+2 and 3 respectively. The dense uniform grid is unable to represent any of the models, but for this representation it is easy to compute the memory usage analytically. Note that *Dense Grid* corresponds to the storage required by the dense grid alone, whereas *Dense Grid Incl Peng* includes the mask and coordinate storage arrays of the narrow band method. Finally note that NP = Not Possible.

7.5 Discussion

Up to now this chapter has illustrated two of the main advantages of the DT-Grid and H-RLE representations; Their storage and computational efficiency. Furthermore, chapter 5 exemplified the out-of-the-box feature inherent to the DT-Grid and H-RLE, and in chapter 6 we elaborated on the versatility of the H-RLE representation over the DT-Grid. In this section we discuss additional advantages as well as disadvantages of the DT-Grid and H-RLE representations.

One advantage of the DT-Grid and H-RLE, compared to adaptive methods, is that existing numerical schemes and level set methods can be employed on these data structures without any change. In fact identical simulation code was employed on all data structures for the level set simulation benchmarks.

One level set method that we have not considered in our work is the fast sweeping method [153] for solving the Eikonal equation. The fast sweeping method performs several sweeps over the entire grid in different directions. On a dense uniform grid this is straightforward, but for sparse representations such as DT-Grid and H-RLE it is not immediately clear how to implement the fast sweeping method efficiently. This is due to the fact that grid points are stored in one particular lexicographic order and that accessing the narrow band in another order requires either a re-encoding of the narrow band or heavy utilization of random and neighbor access operations. While we do not exclude the possible existence of an efficient solution it does not seem straightforward. Note that traditional narrow band methods such as Peng [120] have the same problem if computations are restricted to the narrow band as opposed to the entire volume.

All narrow band methods, including the DT-Grid and H-RLE, incur an overhead both with respect to storage and simulation time if the volume of the narrow band becomes comparable to the volume of the embedding space. In such cases, which we have not encountered in practice in any of our applications, the full level set method should be applied.

The DT-Grid and H-RLE do not support the insertion of grid points at arbitrary positions. Instead grid points must be pushed onto the data structures in lexicographic order. When seen as a general purpose data structure for storing narrow band volumetric data, this is most likely the biggest practical limitation of the DT-Grid and H-RLE. However, as we have shown, level set simulations can be implemented without random inserts. An application that does not generate grid points in lexicographic order is the polygonal mesh to level set conversion algorithm presented in part IV of this dissertation. Three bucket sorts of linear time complexity are typically applied to the grid points prior to being pushed onto the DT-Grid or H-RLE to ensure a lexicographic ordering. The same strategy can be applied to any application that does not generate grid points in lexicographic order.

Both the DT-Grid and H-RLE favor some directions over others in encoding the narrow band. This implies that to some extent, memory usage and computation times depend on which direction is chosen as the primary encoding direction. However, informal tests have shown that in practice the variation in memory usage usually lies within a few percent although it is of course possible to construct examples that behave differently. In all benchmarks presented in this dissertation, no attention was paid to aligning models in any particular way.

Octrees are most likely preferable over the DT-Grid and H-RLE in ray tracing due to their ability to adaptively represent a signed distance field *away* from the interface. On the contrary, DT-Grid and H-RLE are narrow band representations. However it is possible that the DT-Grid and H-RLE can be utilized in a nested hierarchical manner, similar to the approach taken in AMR, and thereby provide coarser approximations to signed distances away from the surface. Whether this is feasible is left for investigation by future work.

7.6 Summary

This chapter compared the storage requirements and run-times of the DT-Grid and H-RLE to previous octree, narrow band and RLE methods. The benchmarks included several different numerical level set methods, memory usage for low and high resolution level sets as well as rendering times for ray tracing. In general the DT-Grid was shown to perform better than previous methods, including the H-RLE, in the context of level set simulations, both with respect to memory usage and run-time. The H-RLE performed comparable to the method of Peng *et al.* [120] combined with the narrow band rebuild of Nilsson *et al.* [108], with a tendency to be faster for low order accurate methods and slower for high order accurate methods. For ray tracing based on random access, a dense uniform grid is faster than the alternatives tested in this thesis as long as sufficient memory is present to keep the gridded data in memory. In most cases the DT-Grid and H-RLE perform faster than octrees. The chapter ended with a discussion of known advantages and disadvantages of the DT-Grid and H-RLE representations.

Part III

Algorithms for Compression and Out-Of-Core Level Set Methods

Chapter 8

Introduction

In part II we proposed the DT-Grid and H-RLE data structures for representing high resolution level set surfaces. These data structures allow for out-of-the-box simulations and were shown to outperform existing state-of-the-art level set methods both in terms of computational efficiency and storage requirements. While the DT-Grid and H-RLE do indeed allow for high resolution level sets, there are still cases where it is simply not possible to represent and deform the surfaces within the 1-2 GB limit of physical memory available on typical desktop computers. Whereas direct ray tracing of level sets based on DT-Grid and H-RLE requires basically no auxiliary data structures, level set simulations on the other hand typically require multiple surfaces (double buffers) as well as velocity fields, morph targets in memory simultaneously. Our novel work presented in part III targets these problems by proposing a generic framework for the representation and deformation of level set surfaces at extreme resolutions. In particular our 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 high resolutions of the deforming geometry, the only limitation being the amount of disk space, and 2) compact and relatively fast *compression* strategies that effectively 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 our work is the first to apply it in the context of level set *simulations*. Our framework is generic 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 the DT-Grid [105] and the H-RLE [48]. 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. 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 *i.e.* the DT-Grid, and can for gigabyte sized level sets sustain a throughput (grid points/sec) 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. Several applications in computer graphics can benefit from our framework, and here we give an example of a shape metamorphosis with peak-storage requirements close to 5GB simulated on a desktop machine with 1GB of memory. In chapter 15 in part V several other applications are demonstrated. This includes fluid simulations in-

teracting with large boundaries ($\approx 1024^3$) and the solution of partial differential equations on large surfaces ($\approx 4096^3$). Finally we note that chapter 14 in part IV presents an algorithm for converting triangle meshes into out-of-core level sets. Specifically we demonstrate the generation of a surface at effective resolution $35000 \times 20000 \times 11500$ (7 billion grid points in the narrow band).

Briefly outlined, the structure of part III is as follows. Chapter 9 describes related work in the areas of compression and out-of-core methods. Next chapter 10 describes our generic framework as well as the compression and out-of-core components comprising it. Following that chapter 11 presents an extensive evaluation of both the compression and out-of-core components of our framework, including level set simulation and offline compression. The advantages and limitations of our generic framework are also discussed.

Chapter 9

Related Work on Compression and Out-Of-Core Methods

Our out-of-core and compression framework is essentially based on two techniques that are well known in the field of computer science: Data compression and out-of-core methods like page-replacement and prefetching. 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 optimize and apply these techniques to level set methods. Consequently most of the work described here is not directly related to ours.

9.0.1 Compression Methods

The DT-Grid and H-RLE represent *narrow bands* of volumetric data. Mesh compression methods on the other hand, see [66] and references therein, compress only the surface itself and possibly the normals. Even though it is indeed feasible to compute differential properties from meshes [28], this is generally not an optimal storage format for implicit surfaces like level sets. The reason is primarily that the conversion (*i.e.* map) between the implicit level set and the explicit mesh is not guaranteed to be one-to-one. Consequently important higher order information is lost by converting to the mesh representation using [65, 69, 82] and this information is not recoverable by a subsequent mesh to level set scan conversion. Methods with very good compression ratios have also been proposed for isosurface meshes created from volumetric scalar fields [32, 73, 148]. 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 (*e.g.* a clamped signed distance field) is easy and makes it possible to employ existing volume compression methods [51, 99]. 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.* [71] 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 destroying 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 [55] and the work on volumetric encoding

for streaming isosurface extraction by Mascarenhas *et al.* [86]. 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 include the compression of 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 isosurface extraction. In particular, the grid is partitioned into *partial volume grids* (essentially narrow bands) 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 isosurface is guaranteed. This approach is however not suitable for online simulations. In particular it is not feasible to employ [86] to online compressed simulations, since a stencil of neighboring grid points, used for finite difference computations, must be available. Nevertheless, in chapter 11 we compare our compression method to [55, 86] as an offline compression tool for reducing storage requirements of the produced data.

9.0.2 Out-Of-Core Methods

The area of out-of-core[1] methods is very 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 fields where data intensive algorithms and applications are ubiquitous. This includes image repositories, digital libraries, relational and spatial databases, computational geometry, simulation, linear algebra and of course computer graphics. For a recent survey of the entire field see [157]. For the interested reader we refer to [150] for a specific survey in linear algebra and simulation, and [135] 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 isosurface extraction [86,162], compression of meshes [56] and scalar fields [51], streaming compression of triangle meshes [57], stream processing of points [115], mesh editing and simplification [23] and visualization [23, 25, 43].

Various approaches have been proposed for improving the access efficiency to out-of-core multi-dimensional grids during computation or for optimizing online range-queries in areas such as scientific computing, computational fluid dynamics and visualization. This includes blocking techniques [129], re-blocking and permutation of dimensions [70] as well as the exploitation of modern disks properties [128]. In computer graphics, improved indexing schemes for full three dimensional grids were proposed by Pascucci and Frank [116] in the context of planar subset visualization. The above techniques all deal with dense uniform 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 [126] for N-body simulation in astrophysics. Their work was based on a tree data structure 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 [152]. 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.

There is also a large body of work on general purpose page-replacement and prefetching

[1] Out-of-core algorithms are also often referred to as external memory algorithms.

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 possibly adaptivity to changing and mixed access patterns. In contrast, the page-replacement and prefetching algorithms we propose are *application-specific* and hence designed to work close-to-optimal for particular problems. For an introduction to the standard page-replacement techniques like "Least-Recently-Used" (LRU), "Most-Recently-Used" (MRU) and "Least-Frequently-Used" (LFU) see the excellent book by Tanenbaum [146]. For these standard 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 [109], LFRU [72], 2-Queue [63], LIRS [60], Multi-Queue [164] and FBR [123]. In chapter 10 we will provide an example of this.

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.* [67], application/file-level characterization of access patterns by Choi *et al.* [21], Early Eviction LRU by Smaragdakis *et al.* [136], SEQ by Glass and Cao [42], ARC by Megiddo and Modha [91] and CAR by Bansal and Modha [6]. However, the task of automatic access pattern detection is extremely difficult and page-replacement decisions may actually hurt performance when the estimate is incorrect [18]. 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 [18] for a recent overview. Patterson *et al.* [118], for example, describes a general purpose resource management system that balances hinted[2] 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 [18] 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 for which the general structures of the access patterns are known in advance. Hence we can exploit this to develop a close-to-optimal strategy which is easy to implement and light-weight 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 [56], the work on application controlled demand paging for out-of-core visualization by Cox and Ellsworth [25] and the octant caching on the *etree* by Lopez *et al.* [81].

In-core scan conversion algorithms for converting triangle meshes to signed distance fields have been in use for quite a while. See *e.g.* the successful method of Mauch [89] as well as the recent [4] and the references therein. 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.* [156].

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

Chapter 10

The Out-Of-Core and Compressed DT-Grid - External Memory Level Set Methods

Despite the introduction of the DT-Grid [105] and the H-RLE [48] data structures, a significant issue continues to be the restriction on level set resolutions 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 including the DT-Grid and H-RLE have indeed increased the potential resolution of level set surfaces, but they do not employ compression of the actual 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, 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 chapter.

We have developed a framework that allows for representations and deformations of level set surfaces, fluid velocities and additional fields at very high resolutions, the only limitation being the amount of available disk space. Our general approach is to employ new application-specific out-of-core prefetching and page-replacement techniques combined with statistical compression algorithms. The out-of-core component allows us to utilize the disk space, typically several orders of magnitude larger than main memory, by streaming level sets to and from disk during simulation. In addition, the compression component effectively reduces both offline storage requirements and online memory footprints during simulation. Reducing offline storage requirements is relevant for level set and fluid simulations since they typically produce large amounts of (temporal) data needed for post processing like direct ray tracing, higher order mesh extraction, motion blur, simulation restarts *etc.* 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 simple fact that hard disks are several orders of magnitude cheaper and larger than main memory [150], thus pushing the limits of feasible computations on desktop computers. In addition we have performed out-of-core scan conversions of huge meshes that would require close to 150 GB of main memory if run in-core.

[1] The St. Matthew [76] 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

Finally, when algorithms are CPU bound the performance of optimized out-of-core implementations can be close to in-core counterparts.

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 either DT-Grid, H-RLE, or traditional dense uniform grid representations. However, the sparse representations are preferable since they limit the amount of data that must be processed. For this dissertation we have chosen to build our out-of-core and compression framework on the DT-Grid since it performs slightly better than the H-RLE, see chapter 7. 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 the H-RLE. Our 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, sparse 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 grid point 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 the DT-Grid and H-RLE for deformations that fit in memory. However, by basing our framework on the DT-Grid 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) remains constant even for models of several gigabytes that do not fit in memory. In addition we show that our compression techniques out-perform related state-of-the-art compression algorithms for compressing partial volume grids or narrow bands of volumetric data.

We emphasize that while several of the techniques presented in this chapter 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 part III are produced on a desktop machine with 1 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.* [2] run earth quake simulations on an unstructured mesh with 4 billion grid points on 3000 AlphaServer processors. 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 several graphics applications are demonstrated in chapter 15 in part V as well as in section 10.5 of the current chapter. This includes high resolution partially out-of-core fluid simulations interacting with large out-of-core boundaries, out-of-core shape metamorphosis and the out-of-core solution of partial differential equations on 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 only limited by disk space with regard to both the size of the input mesh and the size of the output level set. This out-of-core scan converter is described in detail in chapter 14 in part IV. An explicit list of contributions and an outline of this chapter is given in the next section.

10.1 Contributions

Our main contribution is the development of a generic framework for the representation and deformation of level set surfaces and auxiliary fields, only limited by the amount of available disk space. Specifically this framework has the following contributions:

- Near optimal out-of-core *page-replacement* and *prefetching* algorithms optimized 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 and obtain a peak performance equal to 65% of state-of-the-art in-core performance.

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

The rest of this chapter is organized as follows: Section 10.2 introduces the basic terminology and structure of our framework. Next sections 10.3 and 10.4 describe the out-of-core and compression components of the framework in detail, and finally section 10.6 concludes the chapter with a brief summary.

10.2 Out-Of-Core and Compression Level Set Framework

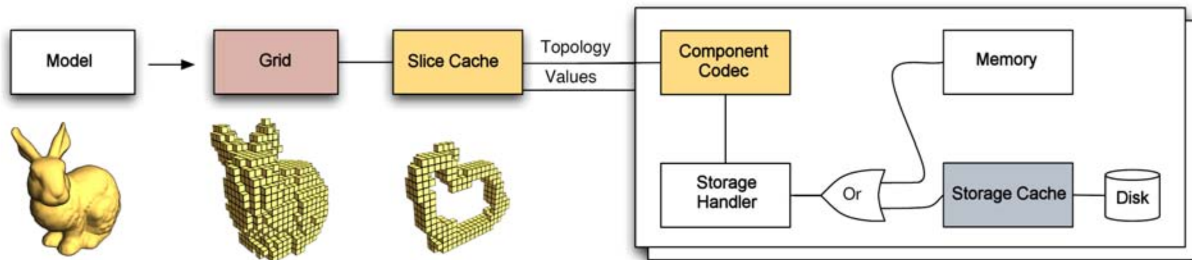


Figure 10.1: The generic framework.

An overview of our generic framework is illustrated in figure 10.1, and we will briefly describe the components from left to right: The level set is represented on a computational *Grid*. The *Slice Cache* allows for 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 computational Grid as illustrated with the bunny example. As the simulation or compression progresses through the grid, these slices are automatically modified and replaced by the 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*. For example, the topology typically requires less storage than the values. This can be exploited in a level set simulation; topology can be kept in-core and only 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 improved compression of each of the separate components. The Slice Cache and the Component Codecs together comprise 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

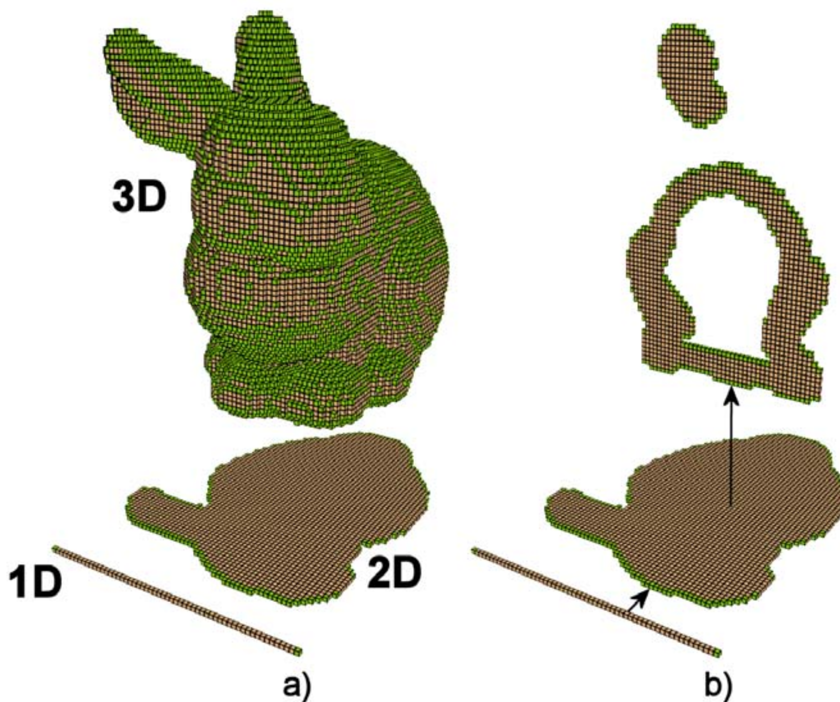


Figure 10.2: a) Illustration of the 1D, 2D and 3D components of a DT-Grid representation of the Stanford bunny at 64^3 resolution. b) Recall that the 1D and 2D DT-Grid components contain pointers to p-columns in respectively 2D and 3D, and the 3D DT-Grid component stores the actual distance values of the level set function. See chapter 5.

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 omitted and combined arbitrarily in our framework.

10.2.1 Terminology

For notational convenience we briefly recall the DT-Grid terminology. Figure 10.2 depicts a DT-Grid encoding of the Stanford Bunny in effective resolution 64^3. Throughout this chapter we will illustrate our techniques with this particular example. Recall from chapter 5 that the DT-Grid employs a hierarchical representation of the narrow band and that each level in this hierarchy contains three components: *value*, *coord* and *acc*. When referring to the components at the n'th level in the hierarchy we will be using the terminology $value_{nD}$, $coord_{nD}$ and acc_{nD} respectively. Furthermore we will denote the $value_{3D}$ component by the term *values* and the remaining components by the term *topology*.

For describing the IO performance of the algorithms presented in this chapter we adopt the terminology of the *Parallel Disk Model* introduced by Vitter and Shriver [158]. 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 disk drive ($D = 1$).

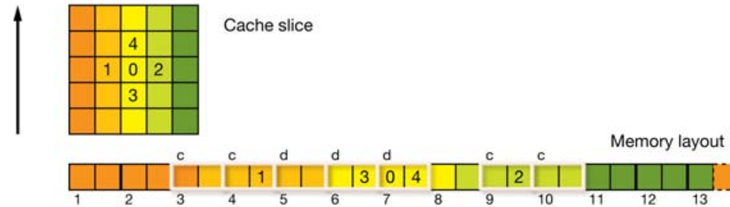


Figure 10.3: Outline of a stencil consisting of five grid points (0-4) as well as a cache slice and the corresponding memory/disk page layout. In this example all grid points are occupied with data and each page is two grid points long. The pages in memory are outlined in white and the others in black. c denotes a *clean page* that has not yet been written to, and d is a *dirty page* that has been written to.

10.3 Out-Of-Core Data Management

The Storage Cache component in figure 10.1 utilizes two different out-of-core data management schemes. For random access we simply employ a standard "Least-Recently-Used" (LRU) page replacement algorithm since it is acknowledged as being the best general choice in many cases (*cf.* 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 methods reduce the number of loaded disk blocks during sequential stencil access. We focus mainly on sequential streaming since a majority of state-of-the-art level set algorithms can be formulated in terms of sequential access operations exclusively. This is true for all the out-of-core examples presented in this chapter and in part V.

As illustrated in figure 10.3, 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[3] maps to contiguous locations on disk or in memory. To address this problem we need to develop page-replacement and prefetching schemes.

Even without prefetching and page-replacement strategies, the IO complexity of sequential stencil access on the DT-Grid is asymptotically optimal. This is due to the fact that it requires only $O(\frac{N}{B})$ IO operations to do stencil-iteration, which equals the lower bound for a sequential scan with respect to asymptotic O-notation [157]. Likewise dilation and rebuilding of the narrow band [105] is linear in the number of IOs. Sequential stencil access essentially corresponds to S sequential and simultaneous scans over the data, where S is the number of grid points in the stencil. However, to increase performance in practice it is necessary to minimize the actual number of loaded disk blocks. A straightforward IO 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 [79], a stencil with support for second order accurate curvature computations has $S = 31$, whereas for only first order upwind computations, $S = 7$. As we will demonstrate in chapter 11, our page-replacement and prefetching techniques do in practice lower the number of passes[4] such that it comes closer to the lower bound. This is the case even for large stencils

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

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

such as WENO.

The optimal page replacement strategy [146] for a demand-paged system (*i.e.* no prefetching) is simple; If a page must be evicted from the cache, always pick the page that will be used furthest in the future. This strategy is of course impossible to implement in practice except for offline 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 non-trivial. As argued previously, existing general purpose page-replacement techniques are not well suited for this access pattern. Consider for example the LRU replacement-strategy. Figure 10.3 shows a 2D grid, a stencil consisting of five iterators and the corresponding positions on the paged disk. For the exposition to follow 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.

Given the fact that our framework is application specific we exploit knowledge about the domain to obtain a replacement-strategy that comes close to the optimal. This strategy is verified in chapter 11. Our caching strategy accommodates the following three essential design criteria:

- The number of disk IO and seek operations is heuristically minimized. In particular seeking is expensive on modern hard drives.
- The disk is kept busy doing IO at all times.
- CPU-cycles are not wasted by copying pages in memory or waiting for disk IO 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 chapter 11 we provide some benchmarks indicating how these parameters affect performance and the page-hit-ratio.

10.3.1 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 above and illustrated in figure 10.3. 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 the cache) is to first 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 10.3 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 chapter 11 we verify that the above strategy is close to optimal by comparing it to the optimal strategy which we computed in an offline pass from logged sequences of page requests.

10.3.2 Prefetching

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

The IO thread iteratively performs the following steps in prioritized order, and as soon as a step is satisfied, continues from the beginning. The tasks are 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 IO 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 only occur 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 [146], originally designed for elevator and disk arm movements. 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 [146] in the context of disk arm movements, this strategy heuristically results in fewer disk seek operations and ensures that no page-requests are left un-serviced for long periods of time. Note that if all pages between two iterator positions are already in-core, *e.g.* positions 1 and 3 in figure 10.3, no pages need to be prefetched in this interval. In this case our elevator algorithm will move more than one iterator position forward in order to locate the next page to be prefetched.

2. **Write-Back:** If no page was prefetched, the IO 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. Write-Back is accomplished by first checking to see if there exist any dirty pages in the cache. If this is the case, the IO thread continually loops through the pages in the cache, starting from the page with the lowest page-id and until a dirty page is encountered. 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 IO thread sleeps until an iterator enters a new page.

The above strategy out-performed 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 IO 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.

10.4 Compression Algorithms

The compression framework can be applied both online during simulation and offline as a tool for application specific storage reduction of simulation data amenable to further processing in a production pipeline. Using the proposed compression framework it is possible to compress very large level set grids out-of-core with a relatively low memory footprint. The Component Codecs we propose in this chapter are based on prediction-based statistical encoding methods and separately compress the topology and values of the grid. In practice we use the fast arithmetic coder described by Moffat *et al.* [95] combined with our own optimized adaptive probability tables. While these methods are ideal for sequential access, random access is typically not feasible into a statistically encoded stream of data. To remedy this somewhat, synchronization points could be inserted into the streams 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 representation, and that the topology component-codecs presented in this chapter are specific for the DT-Grid. However, very similar codecs can be applied to the H-RLE.

10.4.1 Compressing The Topology

The $value_{1D}$ constituent of the topology consists of monotonically increasing indices (figure 10.4.a) that point into the *coord* constituent of the 2D grid component. The difference between two such consecutive values (figure 10.4.b) is twice the number of connected components in a p-column in the 2D grid component. For an illustration, see the 2D grid component in figure 10.6.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, *i.e.* the number of connected components in p-columns, using a second order adaptive probability model [127]. The $value_{2D}$ component has characteristics similar to the $value_{1D}$ component as shown in figure 10.5.a, and

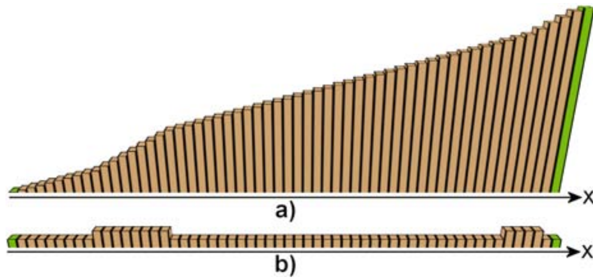


Figure 10.4: a) The values of the $value_{1D}$ constituent as a histogram. b) The difference between consecutive $value_{1D}$ values as a histogram.

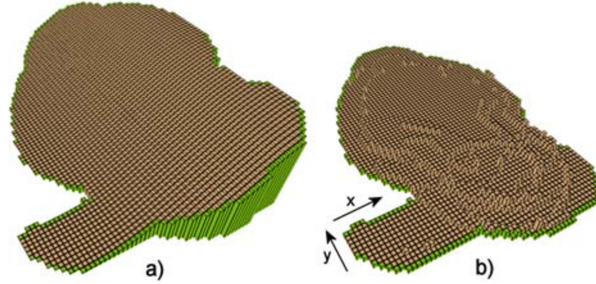


Figure 10.5: a) The values of the $value_{2D}$ constituent of the 2D grid component as a histogram. b) The difference between two consecutive $value_{2D}$ values as a histogram.

the semantics of the difference between two consecutive values in (x, y) lexicographic order is the same, see figure 10.5.b. Hence this constituent is also compressed using a second order adaptive probability model.

The $coord_{1D}$ (x-coordinates) constituent of the topology is encoded using a single differential encoding and a zeroth order adaptive probability model [127]. Typically the $coord_{1D}$ component constitutes only 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 10.2 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 10.6.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 p-columns in the XY plane. Figure 10.6.b illustrates how the y-coordinates in three previous p-columns, shown in blue, are used to predict the y-coordinate in the next p-column, shown in red-brown. In particular we use as predictor the Lagrange form of the unique interpolating polynomial [68] that passes through the y-coordinates in the previous p-columns [5]. Our tests show that higher order interpolants tend to degrade the quality of the prediction which explains why we only use the y-coordinates from the previous two or three p-columns.

Since the topology of these y-coordinate 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 actual y-coordinates in the previous p-columns, we utilize the known information of which connected component we are currently compressing[6]. We then predict from the y-coordinates of the connected components with identical y-coordinate ids in previous p-columns. Note that we cannot simply use the actual true y-coordinate as a means of determining the appropriate y-coordinates in the previous p-

[5] The Lagrange form of the unique interpolating polynomial is preferable over the Newton form because the local configuration of the points from which we are predicting the y coordinates is always the same. Hence the cardinal functions of the Lagrange form can be precomputed.

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

columns since it will not be available during decompression. The above selection criterion means that the prediction will degrade along p-columns where the number of connected components change, but in practice we have not found this to be a problem.

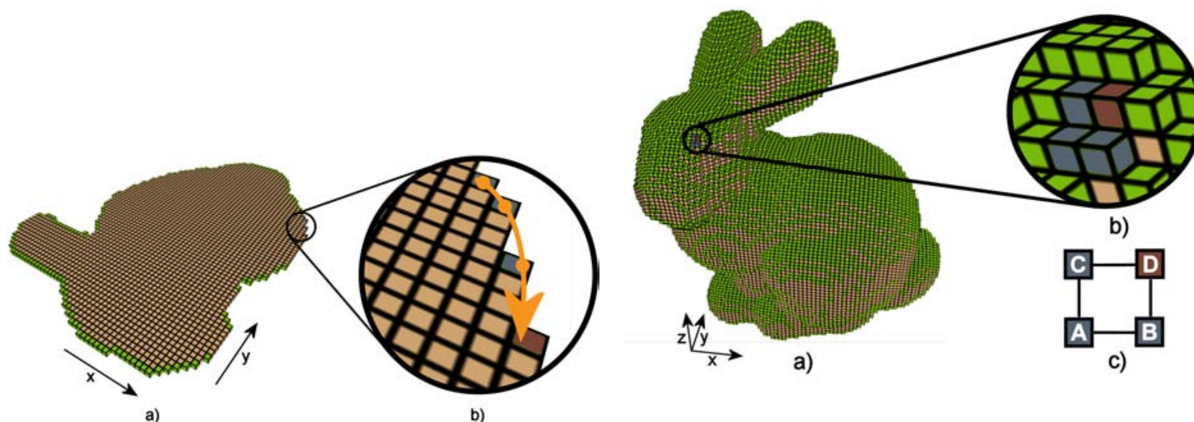


Figure 10.6: **a)** The 2D grid component of the 64^3 Stanford bunny DT-Grid. The y-coordinates of the $coord_{2D}$ constituent are shown in green. **b)** A close-up shows actual y-coordinate (red-brown) predicted by three previous y-coordinates (blue).

Figure 10.7: **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.

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 10.7.a. Only surpassed by the storage requirements of the actual 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 10.7.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 10.7.c. We predict $z(D)$ as $z(A) + \nabla z \mid_A \cdot \binom{1}{1} = z(B) + z(C) - z(A)$. Given the symmetry in this prediction with respect to $z(B)$ and $z(C)$ (both $z(B)$ and $z(C)$ are added) we compress the residual [7] using a context-based adaptive probability model (see *e.g.* [148]) with $z(B) + z(C) - 2z(A)$ 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 $z(B)$ 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. We distinguish between the following three cases: 1) If no grid points exist at all, we use 0 as the prediction. 2) If one grid point 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 the grid components is actually redundant. It is merely used to improve random access into DT-Grid. Hence we can simply exclude the

[7] The residual is the true value minus the prediction.

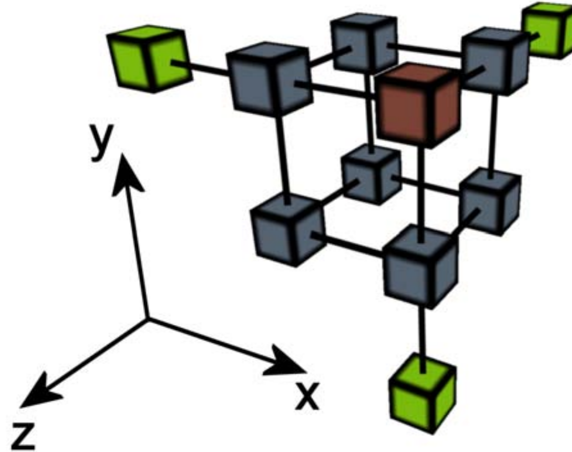


Figure 10.8: The values are compressed using a combination of the 3D Lorenzo predictor, the 2D parallelogram predictor and multi-dimensional differential predictor.

acc constituents in compressed form and rebuild them during decompression. This essentially corresponds to exploiting the Kolmogorov complexity [77] for the compression of *acc*.

10.4.2 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, the values are usually 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 on a combination of the following three techniques: A new *multi-dimensional differential predictor*, the 3D Lorenzo predictor of [51] and the 2D parallelogram predictor of [151]. Since we are compressing narrow bands as opposed to dense volumes, utilizing clamped values outside the narrow band (as is usually done in level set computations) to form predictions will result in a degradation of compression performance according to our tests. 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 [51] (with and without probability contexts), the distance field compression (DFC) method by M.W. Jones [64] as well as several other custom codecs. Note also that the DFC method and the Lorenzo predictor have been shown to perform comparable to various types of wavelets. The codec we propose here gives the best compression performance among the codecs we have benchmarked and at the same time remains fast.

Our multi-dimensional differential prediction is motivated by the fact that the axial acceleration in a signed distance field is typically 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 predictor 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 1st 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 [8]. This information is required during decompression when the actual value is not available. Tests also show that our combined predictor leads to better and faster compression than if compression is applied to the acceleration in the normal direction. This is due to the circumstances mentioned above that degrade the performance of compression of the acceleration in the normal direction.

The intuition behind our approach is to 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 the predictors given below. Consider now figure 10.8 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 in the narrow band 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 a plane orthogonal to one of the axial directions) where all grid points have already been processed. As can be seen from figure 10.8 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 points 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 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 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.

[8] This also means that one would have to solve the third degree polynomial during compression in order to compute the right two bit code.

The others are used on the boundary of the narrow band depending on the local configuration of previously processed neighbors. Note that in the predictions (2), (3) and (4) above 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 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 the predicted value above. 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 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 visual distortion and that the truncation error introduced by the order of the numerical simulation methods is not affected by the quantization rate. If quantization is not desirable one can apply the method for lossless encoding of floating point values by Isenburg *et al.* [78].

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 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 *e.g.* the morph targets demonstrated in [48].

10.5 Example - An Out-Of-Core Shape Metamorphosis

The right-most image in figure 10.9 shows a highly detailed level set model of a bunny (1.62GB and more than 350 million grid points in the narrow band) modeled as an out-of-core Constructive-Solid-Geometry (CSG) intersection between a large 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 10.9 depicts three frames from a shape metamorphosis between the bunny at resolution 2048^3 and the CSG bunny. The simulation was run in Windows XP Pro on a 2.41GHz AMD machine with 1GB of physical memory and a "Western Digital Raptor" disk. To the best of our knowledge this is the highest resolution level set simulations ever documented to run on a personal computer prior to our work. In the next chapter we present simulations at even higher resolutions as part

[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.

Figure 10.9: 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.

of our benchmark evaluations. The peak space requirements for this simulation are close to 5GB in uncompressed DT-Grid format. Note that for large simulations like this, OS paging is not even possible due to OS memory limits for a single application[10]. The uncompressed storage-requirements for the entire simulation were 342 GB. Using the compression method described in this chapter 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. For further applications of the out-of-core and compression framework we refer the reader to chapter 15 in part V, and in the next chapter we provide several benchmarks.

10.6 Summary

We have presented a novel level set framework that fits into existing level set pipelines based on the DT-Grid and the H-RLE data structures. This allows for the first time for representations and deformations of virtually unlimited resolution models, the only limitation being the amount of available disk space. 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 prefetching algorithm. The compression component contributes with compression codecs optimized for level set distance values and DT-Grid topology. To demonstrate the feasibility of the framework, a shape metamorphosis example requiring close to 5GB of storage was presented.

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

Chapter 11

Evaluation and Discussion of the Compressed and Out-Of-Core DT-Grid

In this chapter 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 in-core simulations - even for models of sizes in the order of several GB.

As emphasized previously the out-of-core and compression components can be combined arbitrarily to form methods with distinctive properties and performance for level set simulations. 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, streaming values to disk and compressing topology in-core usually gives the third-best overall performance and so on. 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 the 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. We stress that our framework is applicable on standard desktop machines. In particular all the benchmark tests presented in this chapter are run on a Windows XP PC with a 2.41GHz AMD CPU, 1GB of main memory and a "Western Digital Raptor" disk.

11.1 Page-Replacement and Prefetching

Table 11.1 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 [146] 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, *i.e.* where prefetching is not included. This explains why it is possible

Page-size (KB)	**32 pages**		**64 pages**		**128 pages**	
	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

Table 11.1: 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 were 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.

for our combined page-replacement and prefetching algorithm to achieve better hit ratios than the optimal demand-paged algorithm in table 11.1. 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 11.1 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 11.1 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 below). 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 $Q = \frac{R}{P} = 1$, *i.e.* 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 10.3).

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 11.1. The optimal

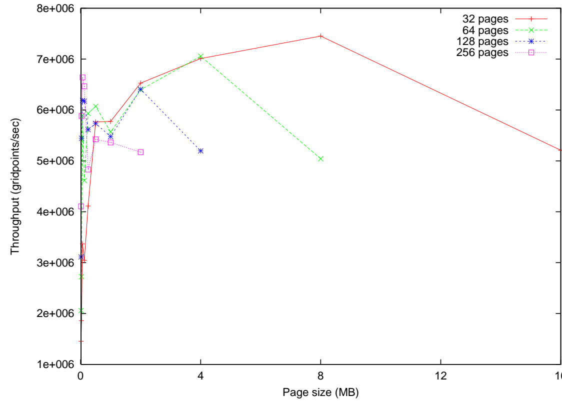


Figure 11.1: The throughput (processed gridpoints/second) as a function of page size (in MB) for various numbers of pages in the cache. The results were generated by sequential iteration over the Stanford Bunny in resolution 8000^3 with a finite difference stencil of seven grid points. The maximal memory usage of the cache was in this case restricted to 512 MB.

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 11.1 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 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.

11.2 Online Out-of-Core and Compression Framework

As a prelude to the performance evaluation of our 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 [1]. 2) The combined read- and write-throughput of sequential iteration with the same finite difference stencil. 3) The throughput of an actual level set simulation, in this case an erosion.

Tables 11.2 and 11.3 list the average throughputs of several tests with the framework variations defined above as well as an implementation of the in-core DT-Grid. For simulations that fit in-core, our framework introduces an overhead due to the additional software layer. 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/compression data structure close to these upper bounds indicates that the method is CPU limited, otherwise it is IO limited.

Table 11.2 and 11.3 indicate that the throughput of both read and read/write iterations for our framework does not depend on the number of grid points 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 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, close to the approximate upper bound, suggesting that stencil iterations are close to CPU 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 due to the fact that the numerical computations hide the IO latency, whereas an arithmetic coder will steal CPU time from the numerically 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 IO limited and the performance drops to approximately 65%. This performance remains constant even for simulations involving files 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 only takes up a relatively small part of the total size of a DT-Grid (see table 11.6), the high resolution of our level sets imply that topology combined with buffering

[1] Note that stencil iteration is an essential component in a level set simulation.

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, although the performance of "OOC DT-Grid II" seems to degrade a little for larger level sets. 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 a data structure such as DT-Grid. 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* DT-Grid-based level set (note that the total storage requirements of a simulation are significantly higher). 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 the compression error may accumulate unless care is taken to ensure that the quantization error remains below the truncation error. We therefore advocate utilizing the out-of-core framework combined with compression of the topology for online simulations and a combined compression and out-of-core framework for offline storage and transfer of data needed later in a production pipeline.

Grid	**Reading** GP/Sec	**Reading&Writing** GP/Sec	**Simulation** GP/Sec
	Res=1000^3, #GP=1.4e7, size=71MB, simsize=84MB		
DT-Grid	1.1e7 (100%)	9.9e6 (100%)	1.2e6 (100%)
OOC DT-Grid I	7.1e6 (65%)	7.4e6 (75%)	9.4e5 (78%)
OOC DT-Grid II	6.8e6 (60%)	6.5e6 (66%)	9.0e5 (75%)
OOC CDT-Grid I	5.2e6 (47%)	5.1e6 (52%)	6.6e5 (55%)
CDT-Grid I	1.2e6 (11%)	NP	8.6e4 (7%)
OOC CDT-Grid II	1.2e6 (11%)	NP	9.9e4 (8%)
	Res=2000^3, #GP=5.7e7, size=289MB, simsize=341MB		
DT-Grid	1.1e7 (100%)	9.8e6 (99%)	1.2e6 (100%)
OOC DT-Grid I	7.5e6 (68%)	7.4e6 (75%)	7.7e5 (64%)
OOC DT-Grid II	7.7e6 (70%)	7.4e6 (75%)	6.9e5 (58%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.5e5 (54%)
CDT-Grid I	1.2e6 (11%)	NP	8.6e4 (7%)
OOC CDT-Grid II	1.2e6 (11%)	NP	1.0e6 (8%)
	Res=2500^3, #GP=8.9e7, size=454MB, simsize=578MB		
DT-Grid	1.1e7 (100%)	9.9e6 (100%)	2.2e5 (18%)
OOC DT-Grid I	7.4e6 (67%)	7.4e6 (75%)	7.9e5 (66%)
OOC DT-Grid II	7.8e6 (71%)	7.1e6 (72%)	6.9e5 (57%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.6e5 (55%)
CDT-Grid I	1.2e6 (11%)	NP	8.6e4 (7%)
OOC CDT-Grid II	1.2e6 (11%)	NP	8.6e4 (7%)
	Res=3000^3, #GP=1.3e8, size=655MB, simsize=771MB		
DT-Grid	1.1e7 (100%)	9.8e6 (99%)	NP
OOC DT-Grid I	7.6e6 (69%)	7.5e6 (76%)	7.9e5 (66%)
OOC DT-Grid II	7.9e6 (72%)	7.8e6 (79%)	6.8e5 (57%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.5e5 (54%)
CDT-Grid I	1.2e6 (11%)	NP	8.6e4 (7%)
OOC CDT-Grid II	1.2e6 (11%)	NP	8.6e4 (7%)

Table 11.2: Throughput rates (gridpoints/second) for stencil iteration through the entire narrow band of the Stanford Bunny in resolution $1000^3 - 3000^3$ 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 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 we report 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 $\gamma + \Delta x$-tube, which is required for simulation, has been added to the level set (*simsize*). 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. NP = Not Possible.

Grid	**Reading** GP/Sec	**Reading&Writing** GP/Sec	**Simulation** GP/Sec
	Res=4000^3, #GP=2.3e8, size=1.2GB, simsize=1.4GB		
DT-Grid	3.3e6 (30%)	1.5e6 (15%)	NP
OOC DT-Grid I	7.6e6 (69%)	7.5e6 (76%)	7.9e5 (66%)
OOC DT-Grid II	8.1e6 (74%)	7.2e6 (73%)	6.8e5 (57%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.6e5 (55%)
CDT-Grid I	1.2e6 (11%)	NP	9.3e4 (8%)
OOC CDT-Grid II	1.2e6 (11%)	NP	9.4e4 (8%)
	Res=5000^3, #GP=3.6e8, size=1.8GB, simsize=2.1GB		
DT-Grid	2.1e6 (19%)	1.3e6 (13%)	NP
OOC DT-Grid I	7.7e6 (70%)	7.5e6 (76%)	7.9e5 (66%)
OOC DT-Grid II	8.2e6 (75%)	6.7e6 (68%)	6.6e5 (55%)
OOC CDT-Grid I	5.4e6 (49%)	5.3e6 (54%)	6.6e5 (55%)
CDT-Grid I	1.2e6 (11%)	NP	9.2e4 (8%)
OOC CDT-Grid II	1.2e6 (11%)	NP	9.4e4 (8%)
	Res=6000^3, #GP=5.2e8, size=2.6GB, simsize=3.1GB		
DT-Grid	NP	NP	NP
OOC DT-Grid I	7.7e6 (70%)	7.5e6 (76%)	7.9e5 (66%)
OOC DT-Grid II	8.3e6 (75%)	6.7e6 (68%)	6.5e5 (54%)
OOC CDT-Grid I	5.3e6 (48%)	5.2e6 (53%)	6.6e5 (55%)
CDT-Grid I	1.2e6 (11%)	NP	8.8e4 (7%)
OOC CDT-Grid II	1.2e6 (11%)	NP	9.3e4 (8%)
	Res=8000^3, #GP=9.2e8, size=4.7GB, simsize=5.5GB		
DT-Grid	NP	NP	NP
OOC DT-Grid I	7.6e6 (69%)	7.5e6 (76 %)	6.7e5 (56%)
OOC DT-Grid II	8.3e6 (75%)	6.7e6 (68%)	6.1e5 (51%)
OOC CDT-Grid I	5.3e6 (48%)	5.3e6 (54%)	6.4e5 (53%)
CDT-Grid I	1.2e6 (11%)	NP	NP
OOC CDT-Grid II	1.2e6 (11%)	NP	9.2e4 (8%)

Table 11.3: Throughput rates (gridpoints/second) for stencil iteration through the entire narrow band of the Stanford Bunny in resolution $4000^3 - 8000^3$ with reads only, for 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 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 we report 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 $\gamma + \Delta x$-tube, which is required for simulation, has been added to the level set (*simsize*). 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. NP = Not Possible.

11.3 Offline Compression

Model	**Our Method**		**Isenburg/Mascarenhas**	
	Comp Time	Comp Size	Comp Time	Comp Size
Bunny, $\gamma = 5$, 89x88x71 0.703 MB, 153818 grid points	0.188 s	0.160 MB	1.92 s	0.187 MB
Buddha, $\gamma = 5$ 128x58x57 0.778 MB, 173005 grid points	0.218 s	0.189 MB	2.19 s	0.217 MB
Statuette, $\gamma = 5$, 88x55x49 0.303 MB, 66942 grid points	0.094 s	0.0789 MB	0.844 s	0.0872 MB

Table 11.4: Comparison between the performance of our compression framework and the methods of Isenburg *et al.* and Mascarenhas *et al.* γ is the width of the narrow band.

Next we compare our compression framework to the compression scheme for hexahedral volume meshes by Isenburg and Alliez [55] and associated scalar values by Mascarenhas *et al.* [86]. Unfortunately we are not able to make a direct comparison to the benchmarks reported in [86]. Firstly only one of the data sets used in the chapter seems to be publicly available (the engine data-set). Secondly the tests are run on hardware that we do not have access to and finally only the cell counts (as opposed to grid point 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 [86]. 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 [54], chapter 6). Consequently we are limited to evaluating relatively small grids sizes compared to what is used elsewhere in this chapter. Table 11.4 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 11.4 it can be seen that on average our method is about 10 times faster and compresses 14% better than [86].

We also evaluated the performance of our framework against the performance of the widely used bzip2 compressor as shown in table 11.7. Additionally we compared against a custom variant of bzip2 where each component of the DT-Grid is compressed separately hence allowing the compressor to take advantage of the higher degree of redundancy present inside each individual component. We note that regarding compression performance our method performs significantly better than both approaches. Furthermore it is faster than bzip2 and in most cases comparable in speed (averaging over compression and decompression time) to the custom bzip2 implementation, although it in the case of the Buddha model compresses more than twice as fast as the custom bzip2 implementation.

Finally, tables 11.5 and 11.6 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 92% compression when compared to an uncompressed DT-Grid representation. When compared to an uncompressed dense uniform 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 11.5 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 our method compresses the topology of the grid. This is evident from table 11.6 where the percentages of compression for the individual components are listed. The only component that is not compressed well is $coord_{1D}$ which the DT-Grid in many cases already represents using very few bytes due to its hierarchical index compression.

Using the Metro tool [24] 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 de-compressed (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 11.5 and 11.6 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 [73]. We were not able to measure the distortion on the higher resolution distance fields due the large amount of triangles generated by the extraction, but as argued above the distortion decreases as the resolution is increased. In the future we plan to implement a Metro tool equivalent that operates directly on the level set. Using the out-of-core and compression framework described in this dissertation the Hausdorff distance can be evaluated simply by streaming the narrow band distance fields through memory.

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

Table 11.5: Compression Statistics for various level set models. Narrow band width, $\gamma = 3$. Quantization of the 32 bit values to 14 bits per grid point. Most of the captions should be self-explanatory except *BPGP* which is the Bits Per Grid Point, *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/DenseGrid* which is the percentage of compression measured against an uncompressed dense-volume grid containing the narrow band (this illustrates the efficiency of our framework seen as a dense-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	Orig Topology BPGP	Comp Topology 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	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	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	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	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	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	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	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	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	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	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	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	3.64	0 (0.0)	86 (0.0)	92 (6.8)	99 (0.0)	99 (2.9)	89 (83)	100 (6.8)

Table 11.6: Compression Statistics for various level set models. Narrow band width, $\gamma = 3$. Quantization of the 32 bit float values to 14 bits. 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 following the percentage of compression is the percentage that this particular component takes up of the entire uncompressed DT-Grid.

Model	OOC CDT-Grid II			Custom BZip2			BZip2		
	Comp Time	Decomp Time	% Comp	Comp Time	Decomp Time	% Comp	Comp Time	Decomp Time	% Comp
Lucy,6844 × 3987 × 2290	313	301	81%	373	221	60%	640.8	342.5	14%
David,9768 × 3987 × 2304	487	467	86%	592	349	63%	1006.2	532.8	19%
Bunny,3991 × 3956 × 3094	237	239	93%	336	185	62%	576.9	294.1	18%
Buddha,9718 × 3993 × 3998	888	873	85%	1855	624	67%	1927	1019	17%

Table 11.7: This table shows the performance of the block-sorting and dictionary based compressor **bzip2** and compares it to our out-of-core compression framework for several high resolution level set models represented as DT-Grids. *Custom BZip2* is a modified variant of the BZip2 algorithm where the individual components of the DT-Grid are compressed separately and the values quantized. Quantization rate is 14 bits per grid point for our method, *OOC CDT-Grid II*, and *Custom BZip2*. No quantization was applied in the *BZip2* case as this method simply operates on the uncompressed DT-Grid file stored on disk.

11.4 Discussion

We start by noting that our framework is intended for very large level set simulations given the fact that it obviously cannot outperform pure in-core simulations given sufficient memory.

One seemingly obvious limitation of our framework is related to random access. In particular random access into a statistically encoded stream of data based on an adaptive model is not feasible since a compressed value depends on all values compressed before the value itself. Hence potentially the entire stream of data would need to be decompressed to lookup a single value. Random access into our out-of-core framework is indeed possible, and we have used it to ray trace the models of the bunny shape-metamorphosis presented in chapter 10. Still it remains relatively slow compared to in-core random access. We note two things regarding these facts. First of all we are currently considering prefetching and page-replacement strategies than just basic LRU for ray tracing. Secondly, ray tracing is in fact the only application we have considered that requires random access. All other applications including the shape-metamorphosis in chapter 10 as well as the fluid simulation and PDE on manifold applications in chapter 15 rely solely on sequential stencil access.

Regarding our out-of-core framework a few additional limitations can be pinpointed. Our page-replacement and prefetching strategies were developed as application specific, particularly optimized for sequential access with finite difference stencils. Obviously any attempt to utilize them as general purpose strategies in another situation may fail dramatically. However, again we stress that all our applications, except ray tracing, access memory in a pattern adhering to the assumptions of our algorithms. A disadvantage of our out-of-core framework is 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.

One potential limitation of our streaming compression framework is that in order for it to be sufficiently fast, the Slice Cache must reside entirely in-core. However, generally the few slices required (for our compression codecs only 3-4) take up relatively little memory as evident from the memory benchmarks presented in table 11.5.

Using our out-of-core level set framework it is possible to run simulations at resolutions limited only by the disk space available. Furthermore, as only the narrow band is stored and processed, simulations can be run relatively efficiently compared to previous methods, as demonstrated throughout this dissertation. Hence in many respects one can argue that the level set method has now matured to a point where the resolution obtainable is no longer the major limiting factor. For visual effects production, low simulation times are of prime importance due to the often quite limited post-production schedules. Even though the methods we propose are capable of running out-of-core simulations at high throughputs, level set and fluid simulations at the scales we are able to obtain become computationally intensive due to many calculations required to solve the PDEs. Thus, even if given sufficient memory to run the simulations in-core, they would be quite time consuming. This also means that future research should look more closely into improving the computational efficiency of these high resolution simulations. Parallelization is a strong alternative to out-of-core methods for increasing the resolution of level set and fluid simulations, and distributed computations (*e.g.* using MPI) offer the possibility for more memory distributed across several machines. The paper [52], which leverages on our DT-Grid and H-RLE work, suggests this approach and advocates the feasibility of parallelization in the context of fluid simulations. Obviously parallelization has the advantage that computations are split between several processors which incurs an improvement in running time if the overhead

in communicating boundary conditions between processors does not eclipse the gain obtained by splitting the computations. On the contrary an out-of-core method such as ours is unlikely to outperform an in-core method unless physical memory is exceeded and virtual memory taken into use. However, our method has the advantage that it can be run on a single desktop machine with limited physical memory resources.

We believe it feasible to combine our out-of-core and compression framework with parallelization, hence facilitating both the use of several processors as well as enabling larger problems to be maintained on each computational node which is likely to have a limited set of resources.

11.5 Summary

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 in-core DT-Grid which in turn has been shown to outperform other level set data structures (see chapter 7). 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 compresses better and faster than a related volumetric band compression method. It also outperforms bzip2 a widely used standard compression tool. We strongly believe that our framework applicable in practice to several areas in computer graphics, including but not limited to, high-resolution fluid simulations and shape deformations.

Part IV

Methods for Converting Polygonal Meshes into High Resolution Level Set Surfaces

Chapter 12

Introduction

Triangle mesh representations are by far the most commonly used exchange formats for boundary representations. Surfaces generated from scannings of real world geometry, for example, are typically available as triangle meshes, although the raw scanned representation is a set of points. Furthermore, the models generated by artists are often converted to triangle meshes, even though representations such as NURBS and subdivision surfaces are usually employed at the modeling stage. We typically wish to utilize 3D models as initial shapes for level set surfaces including fluids and boundaries. These facts motivate why converting a triangle mesh into a level set representation is an essential part of the level set pipeline. Typically the conversion process entails computing the signed distance transform to the given boundary representation in a narrow band about the surface. In this dissertation we consider a special class of triangle meshes. The class we consider encompasses *consistent*, or closed, orientable and non-self-intersecting, meshes. This class of meshes is the simplest to consider since a mesh with these properties maps uniquely to a level set surface which by definition must be closed and non-self-intersecting[1]. The literature on this subject is vast, but nevertheless new techniques are needed for converting meshes into the high resolution data structures presented in parts II and III. The main reason for this is that a conversion algorithm must scale with the size of the surface, both in terms of computational efficiency and storage requirements. We present both an in-core [48] and an out-of-core [106] conversion algorithm for consistent polygonal meshes. The in-core converter can generate high resolution level sets relatively fast, but performance decreases significantly when utilizing virtual memory. In that case our out-of-core converter proves to be several times faster. Additionally the out-of-core converter imposes no restrictions on the size of the input mesh or output level set other than enough disk space must be available.

A brief outline of part IV is as follows. Chapter 13 summarizes previous work. Next chapter 14 presents the in-core and out-of-core converters along with performance evaluations and a discussion.

[1]Note that in order to unambiguously generate the open, or unenclosed, level set representations mentioned in part II, we need a closed representation initially.

Chapter 13

Previous Methods for Converting Polygonal Meshes

In this section we consider related work on the generation of signed distance fields from polygonal meshes. For our application, the signed distance fields are intended as input to level set deformations. As touched on in the introduction to part IV, we will in this dissertation concentrate on *consistent* polygonal meshes represented by closed orientable and non-self-intersecting objects with a clear separation of inside and outside.

The brute force method [119] for computing the signed distance field from a consistent polygonal mesh is easy to devise but prohibitively slow. It works as follows: For each grid point in the enclosing volume, the signed distance to the mesh is evaluated by considering the distance to each individual triangle in turn. Due to these limitations, Payne and Toga [119] proposed an optimized variant of the brute force method. In particular their method visits the entire grid and for each grid point queries the signed distance in a hierarchical tree of bounding boxes enclosing the triangles. The hierarchical organization allows for the relatively fast pruning of triangles based on estimates of the smallest and largest distance to any triangle within the bounding box. A related method is the LUB-tree approach for computing the minimum distance to a polygonal mesh by Johnson and Cohen [62]. Given that the hierarchical tree is refined to the level of individual triangles, these methods have time complexities in the order of $O(M \log F)$ where M is the number of grid points and F is the number of faces of the mesh.

A different approach was taken by Mauch [88, 89] who presented the Characteristics/Scan Conversion (CSC) algorithm for computing the signed distance field to a polygonal mesh. The method works by scan converting a set of overlapping polyhedra resembling conservative estimates of the voronoi cells of the faces, edges and vertices of the mesh respectively. Contrary to the true voronoi cells, the polyhedra are very fast to compute directly from the specification of the mesh and a user-specified threshold of maximum distance. Owing to the user specification of maximal distance this method is ideally suited for computing a narrow band signed distance field representation. Apart from the $O(L^3)$ initialization time of the embedding volume, the method of Mauch has a time complexity of $O(cM + F)$ where M is the number of grid points in the narrow band, F is the number of faces, and c is a constant expressing the degree of overlap of the polyhedra. The core method is fast due its linear time complexity and due to the fact that only the grid points in the narrow band are touched by the algorithm during distance computation. An alternative to the CSC algorithm is to only scan convert the individual triangles, corresponding to the zero-crossing grid points. Subsequently one can then employ an approximate method like the first order accurate fast marching method (FMM) [131] to propagate distance to grid points further away. However as illustrated in [89], the CSC algorithm compares favorably to FMM and tends to outperform it as the grid is refined. In addition the CSC method is accurate

to within machine precision.

More recently, Sigg *et al.* [134] proposed an improvement to the CSC method based on the scan conversion of fewer different types of polyhedra. Additionally the method was implemented on graphics hardware hereby exploiting the specialized and optimized scan conversion capabilities supported by the GPU.

In general any method for computing the signed distance from an arbitrary point to a mesh can be used to compute the signed distance field: The method can simply be evaluated in every grid point. A very recent approach to this was proposed by Baerentzen and Aanaes [5] . While this algorithm is very robust it is not documented how well it performs against previous scan conversion methods. Additionally it is not immediately feasible in the context of narrow band signed distance fields, as distances at all grid points in the embedded volume are evaluated.

Finally we note that none of the existing work has considered the computation of very large high resolution narrow band signed distance fields, such as those required as the input to our data structures.

Chapter 14

Converting Polygonal Meshes

This chapter considers the problem of converting a closed non-self-intersecting polygonal mesh into a narrow band signed distance field represented on a level set data structure such as the DT-Grid or H-RLE. While fast algorithms for computing signed distance fields from meshes have previously been proposed, none of them have considered level sets at the scale presented in this dissertation. To remedy this we present two algorithms, both of which are algorithmic constructs that extend and thus largely leverage on the CSC method [89]. The first algorithm [48] operates entirely in-core in order to retain the speed offered by such a strategy. Both its storage requirements and computational efficiency are linear in the number of faces of the mesh and the number of grid points in the narrow band. However, for very large meshes and/or level sets, this approach becomes infeasible and eventually breaks down due to its utilization of OS virtual memory. The second algorithm [106] overcomes this problem by employing an out-of-core conversion strategy. This method does not impose any restrictions on the size of the input mesh nor the output level set, other than the fact that sufficient disk space must be available. Although the second algorithm incurs an increase in asymptotic time complexity compared to the in-core algorithm, it is significantly faster in practice when the in-core approach relies on virtual memory. Using the out-of-core algorithm we demonstrate the generation of level sets that are close to two orders of magnitude larger than demonstrated in the graphics community prior to the work presented in this dissertation. All the level set models utilized throughout our work are generated using the methods in this chapter. Due to the way the CSC method [89] operates on the faces of the polygonal mesh representation it is generally referred to as a *scan conversion* algorithm. When describing our algorithmic extensions to [89] we will retain this descriptive term. Next we present our contributions as well as provide an outline of the sections this chapter contains.

14.1 Contributions

The contributions of this chapter are

- An in-core conversion algorithm for computing DT-Grid or H-RLE based level set representations from polygonal meshes. Storage requirements of the algorithm are $O(M_3 + F)$ and computational requirements are $O(cM_3 + F)$, where M_3 is the number of grid points in the 3D narrow band, F is the number of faces of the mesh and c is a scalar depending on the overlap of the characteristic polyhedra (see previous chapter and [89]) which in turn depend on the mesh and the width of the narrow band.

- An out-of-core conversion algorithm for computing out-of-core DT-Grid or H-RLE based level sets from polygonal meshes. The only practical limitation on the size of the input mesh and output level set is the amount of available disk space. The storage requirements are $O(M_3 + F)$, but the computational requirements increase to $O(cM_3 + M_3 \log M_3 + F \log F)$, and the complexity in terms of IO operations is $O\left(\frac{F}{B} \log_{M/B} \frac{F}{B} + \frac{M_3}{B} \log_{M/B} \frac{M_3}{B}\right)$, where B is the size of a disk block, and M is the memory size in the *Parallel Disk Model* [158].

The methods above enable us to generate level sets of very high resolution. In particular we have generated a narrow band level set with effective grid resolution $\mathbf{35000} \times \mathbf{20000} \times \mathbf{11500}$ and containing 7.08 billion grid points in the narrow band.

The remainder of this chapter is structured as follows. Section 14.2 presents the in-core conversion algorithm and presents an example of the level set detail achievable with this method. Next section 14.3 describes the out-of-core conversion algorithm and following that, chapter 14.4 evaluates and discusses the two methods proposed. Finally section 14.5 concludes this chapter with a brief summary.

14.2 In-Core Scan Conversion

The scan conversion technique of [88, 89] for computing the signed distance transform to a closed and non-self-intersecting mesh has gained wide-spread use. It sequentially scan converts a set of characteristic polyhedra, resembling a set of widened voronoi cells, derived from the mesh. In practice the method is fast and compares favorably with the fast marching method [131]. Furthermore the result is accurate to within machine precision. The method as originally proposed has memory requirements that are $O(L^3+F)$, where L is the side-length of the enclosing bounding box. The dependency on L^3 does not scale well as L increases and hence becomes a limiting factor. The core conversion algorithm in [88, 89] has a time complexity of $O(cM_3 + F)$, where M_3 is the number of grid points in the narrow band, F is the number of faces and c is a scalar that depends on the overlap of the characteristic polyhedra. However, due to an $O(L^3)$ initialization time, the overall time complexity is still $O(L^3 + F)$. It should be noted that in practice, the limiting factor of [88] that comes into play first is the memory consumption, but for higher resolutions a time usage depending on $O(L^3)$ is also severely limiting. In this section we describe how to augment the algorithm to only use $O(M_3 + F)$ memory and $O(cM_3 + F)$ time, and how to use it in the context of the DT-Grid and the H-RLE level set representations. As a prelude to introducing our DT-Grid / H-RLE scan converter, let us first consider a modified scan converter of a polygonal mesh into a clamped signed distance function ($|\phi| \leq \gamma$) on a dense uniform grid:

1. Partition the bounding volume of the mesh into $P \times Q \times R$ (non-overlapping) rectangular axis-aligned sub-volumes and initialize the values in each of these to γ.

2. Partition the mesh into $P \times Q \times R$ sub-meshes, where the (p, q, r)'th sub-mesh contains all polygons of the original mesh that lie within a distance of γ from the (p, q, r)'th sub-volume.

3. Scan convert each sub-mesh into the corresponding sub-volume using [88, 89].

Our approach is similar to the one above, except that in order to ensure the $O(cM_3 + F)$ time-complexity and the $O(M_3 + F)$ memory-complexity, only one representative sub-volume,

of size $O(M_3)$, is kept in memory. This representative sub-volume is only initialized once which takes time $O(M_3)$. When scan converting the narrow band of the (p, q, r)'th sub-mesh into the representative sub-volume, the coordinates of each grid point, for which a distance is computed, are stored in a data structure and compressed on-the-fly. When the sub-mesh has been scan converted, the computed signed distance values are saved into the data structure already holding the coordinates. Furthermore the corresponding grid points in the representative sub-volume are reset to γ. This procedure is then repeated for the remaining sub-meshes. The time complexity of scan converting the $P \times Q \times R$ sub-meshes is $O(cM_3 + F)$ since for each sub-mesh care is taken only to touch grid points inside the narrow band. Following the scan conversion process, three bucket sorts are performed on the coordinates of the recorded grid points in order to bring them into the lexicographic order required by the DT-Grid and H-RLE data structures. Each bucket sort takes time $O(L + M_3)$ where L is the side-length of the grid bounding box. In our analysis we assume that $L = O(M_3)$ which is usually the case in practice. Note also that we have implicitly assumed that $P \times Q \times R = O(M_3 + F)$ and that each face intersects at most a constant number of sub-volumes.

Our scan conversion method assumes that a sub-volume has a size in the order of $O(M_3)$ in order to ensure the $O(cM_3 + F)$ time complexity. In practice we usually choose the size of a sub-volume heuristically based on the memory available and the size of the mesh (see section 14.4). In order to guarantee the theoretical time complexity in practice, one could estimate a lower bound for M_3 in a preprocessing step in time $O(cM_3 + F)$ and space $O(F)$. However, setting the size of a sub-volume heuristically saves the extra preprocessing step and works well in practice.

The outlined scan conversion algorithm retains the speed of the original method [88] for smaller level sets, whilst at the same time allowing for much higher resolution. We demonstrate this in section 14.4. Finally we note that our conversion algorithm is fairly easy to parallelize since each sub-volume can be converted independently on a separate processor.

Figure 14.1 shows renderings of the Lucy model with the DT-Grid as the underlying surface representation. It demonstrates the wide range of scale represented and made possible by our novel level set representations and the new scan conversion algorithm presented above.

14.3 Out-Of-Core Scan Conversion

The algorithm described in the previous section only works in-core and consequently the resolutions of the input mesh and output level set are limited by the available memory. In fact for many of the models used in this dissertation, resolutions either exceed the virtual memory limits or result in slow OS page swapping. To address this problem we have developed an out-of-core extension to our in-core scan conversion algorithm based on [89]. This extension allows us to scan convert consistent mesh models into level sets of unprecedented resolution. The only practical limitation on the sizes of input meshes and output level sets is the available disk space. As we will demonstrate in section 14.4, the out-of-core converter out-performs its in-core equivalent when the memory requirements of the in-core scan converter exceed the physical memory limit and virtual memory is utilized. 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 out-of-core to create a list of faces where each face is represented by its vertex coordinates. This list of faces is next used to partition the mesh into a number of sub-meshes. Each sub-mesh is then scan converted in-core using the

Figure 14.1: Zooming in on The Lucy model ($3000 \times 1726 \times 5144$) using the DT-Grid as the underlying level set representation. The pictures demonstrate the high detail present in the model. Note that the bumps on the surface as seen from the closest viewpoint are not artifacts from the scan conversion process, but rather very fine scale detail also present in the corresponding mesh representation generated from scanned real world geometry. Renderings by Ola Nilsson based on my scan conversion.

method of [89], 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 lexicographic order using an external sort. The lexicographic order is required for the construction of an out-of-core DT-Grid or H-RLE 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 de-reference 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 $O(F)$ IO 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 [20]. This can be done in $O\left(\frac{F}{B}\log_{M/B}\frac{F}{B}\right)$ IO operations, where F is the number of faces and B is the number of faces per disk block. Briefly described the de-referencing 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. The time complexity of this step is $O(F \log F)$.

2. Next the mesh is partitioned into $P \times Q \times R$ sub-meshes. P, Q and R depend on the amount of available memory, and in general they are simply determined heuristically similar to the approach taken for the in-core scan converter. The $(p, q, r)'th$ sub-mesh consists of all the faces within a distance of γ from the $(p, q, r)'th$ sub-volume resulting from dividing the bounding box of the mesh into $P \times Q \times R$ equally sized axis-aligned and non-overlapping sub-volumes. To do the actual partitioning, a single scan through the list of faces determines for each face which sub-volumes it contributes to. During this scan, data is streamed into a file, f, of 7-tuples, each consisting of the three vertex indices, the coordinates of the three vertices, and the sub-volume id that this face maps into. Assuming that at most a constant number of sub-volumes intersect each face, the partitioning step requires $O(\frac{F}{B})IO$ operations and has a time complexity of $O(F)$.

3. To apply the method of [89], each of the sub-meshes are first converted into individual indexed triangle sets. This is done by a single external sort of and a subsequent scan through the sorted tuples in f. First the 7-tuples in f are sorted according to their sub-volume 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[1] to local indices. The individual indexed sub-meshes are progressively streamed to disk. Again this step requires $O\left(\frac{F}{B} \log_{M/B} \frac{F}{B}\right)$ IO operations and thus has a time complexity of $O(F \log F)$.

4. Next the in-core scan converter of [89] is separately applied on each sub-mesh. The coordinates of the narrow band grid points and their 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 the in-core scan converter which ensures an $O(F + cM_3)$ time complexity of step 4. In terms of IO operations the complexity of this step is $O(\frac{F}{B} + \frac{M_3}{B})$ IO operations.

5. Finally the 4-tuples generated above are sorted into lexicographic order using a single external sort and the out-of-core level set is constructed by sequentially pushing grid points into the DT-Grid. Due to the external sort of grid points, this step requires $O\left(\frac{M_3}{B} \log_{M/B} \frac{M_3}{B}\right)$ IO operations and has a $O(M_3 \log M_3)$ time complexity.

Note that steps 3 and 4 above can be performed simultaneously such that a sub-mesh is scan converted as soon as it is generated. In this way it is not necessary to stream the sub-meshes to disk. For large models or 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 14.3, the number of grid points are in 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 (three since this is the

[1] By *global index* we mean the index in the input mesh.

number required to generate the lexicographic order). In our experience this gives a factor of 2.5 improvement in the time spent on sorting. The peak memory consumptions of the algorithm corresponds to the size of the largest sub-mesh plus the size of a sub-volume. The overall time complexity of the algorithm is $O(cM_3 + M_3 \log M_3 + F \log F)$, and the complexity in terms of IO operations is $O\left(\frac{F}{B}\log_{M/B}\frac{F}{B} + \frac{M_3}{B}\log_{M/B}\frac{M_3}{B}\right)$. Our out-of-core scan conversion algorithm is a typical example of an algorithm that can be solved with a linear time complexity in-core but requires a linear times logarithmic number of IOs to be solved out-of-core [157].

14.4 Evaluation and Discussion

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
CSC	4.156	6.391	NP	NP	NP	NP
Our Method, In-Core	4.375	7.453	24.72	153.9	4227	NP
Our Method, Out-Of-Core	10.20	21.25	67.61	423.1	1096	2049

Table 14.1: Comparison of timings (measured in seconds) between the original CSC method of Mauch [89] and our in-core and out-of-core algorithms 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 original method it is Not Possible (NP) to scan convert in resolution 1000 and above (in fact the limit is much lower than that since a 1000^3 dense uniform grid of floats requires close to 4GB). For 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 the input mesh, the output level set and intermediate data structures exceed the virtual memory limits. The DT-Grid construction time is not included since the in-core scan converter is incapable of generating the DT-Grid for resolutions above 3000^3 due to virtual memory limits.

In this section we present an evaluation and discussion of our scan conversion methods, and compare them to the original CSC method of Mauch [89]. All scan conversions were done on an AMD 2.41GHz machine with 1GB of main memory and a Western Digital WD4000YR hard disk. The size of the sub-volumes was chosen heuristically and in all but one (clarified below) case we utilized a fixed sub-volume of size 256^3.

As can be surmised from table 14.1 our in-core scan converter retains the speed of the original method, and our out-of-core scan converter outperforms the in-core converters when model resolutions exceed the available memory. The original method reaches the virtual memory limits quite quickly which renders it impossible to do scan conversions at 1000^3 and higher. Note also that the running times of our in-core scan converter increase rapidly when the memory limit is approached. When scan converting the bunny at 3000^3, our out-of-core scan converter is roughly four times faster than our in-core scan converter.

Table 14.2 depicts the scan conversion statistics for our in-core scan converter on several polygonal meshes. In all cases the scan conversion times are in the order of seconds or minutes. Note that the reason the scan conversion time of the Lucy model at resolution 512^3 is higher

than at 1024^3 is due to the size of the Lucy mesh which forced us to use a smaller sub-volume size of 128^3 (otherwise the data structures that store the sub-mesh during scan conversion would take up too much memory). The main bottlenecks are in terms of memory. For example, the intermediate partitioned mesh structure (*Mesh II*) requires more memory than the original mesh. Note however that only the partitioned mesh needs to remain in memory during the scan conversion process. Another example is the relatively large size of the intermediate structure of grid points (*IVC*) that is eventually sorted into lexicographic order. This illustrates one of the consequences of the DT-Grid and H-RLE requirement that grid points must be pushed onto the data structures in lexicographic order. The size of the intermediate data structures could be reduced by introducing compression at the cost of longer execution times. However, despite these facts we note that the memory usage still scales as $O(M_3 + F)$ hence allowing for higher resolution.

Table 14.3 lists several very high resolution level set models and out-of-core scan conversion times. We realize that some of the models in table 14.3 are over-sampled in terms of triangle to grid point resolution and the high resolutions are meant merely to illustrate the capability of the out-of-core framework.

As can be seen from table 14.3, the out-of-core scan conversion times for these very high resolution models are in the order of hours, but scan conversion remains feasible. 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 of course be reduced by means of compression and a more compact sorting like [87]. This will however increase running times.

The highest resolution model is the Lucy which was converted into a $\mathbf{35000 \times 20000 \times 11500}$ narrow band level set containing 7.08 billion grid points in the narrow band. To the best of our knowledge this is about two orders of magnitude larger than demonstrated prior to the work presented in this dissertation and about one order of magnitude higher than achievable with our in-core scan converter.

Model	Time [sec]						Size [MB]				Grid
	Mesh Load	Mesh Part	Scan Conv	Lexi Sort	DTGrid Const	Tot	Mesh I	Mesh II	IVC	DTGrid	Point Count
Lucy, $\gamma = 3$	#faces=28055742, #vertices=14027872										
$512 \times 512 \times 512$	55.3	456	744	0.453	1.70	1258	482	565	41.3	18.7	4.09e6
$1024 \times 1024 \times 1024$	55.3	330	697	1.69	8.56	1093	482	524	173	78.5	17.1e6
Lucy, $\gamma = 5$											
$512 \times 512 \times 512$	55.4	424	997	0.703	2.16	1476	482	610	59.2	27.9	6.52e6
$1024 \times 1024 \times 1024$	55.5	312	899	2.81	13.5	1283	482	544	255	120	28.0e6
David, $\gamma = 3$	#faces=7227031, #vertices=3614098										
$512 \times 512 \times 512$	14.4	3.98	161	0.468	2.52	183	124	132	66.2	30.2	6.54e6
$1024 \times 1024 \times 1024$	6.42	4.31	199	2.28	13.0	225	124	136	278	126	27.3e6
David, $\gamma = 5$											
$512 \times 512 \times 512$	14.3	4.02	213	0.688	4.25	237	124	136	95.5	45.0	10.5e6
$1024 \times 1024 \times 1024$	6.42	4.36	269	3.49	21.1	304	124	142	409	192	44.7e6
Bunny, $\gamma = 3$	#faces=70064, #vertices=35034										
$512 \times 512 \times 512$	1.50	0.0310	4.88	0.187	0.860	7.45	1.20	1.32	34.7	17.0	3.40e6
$1024 \times 1024 \times 1024$	2.23	0.0470	14.9	0.906	6.63	24.7	1.20	1.43	143	69.8	14.0e6
Bunny, $\gamma = 5$											
$512 \times 512 \times 512$	1.83	0.0470	6.80	0.297	1.69	10.7	1.20	1.36	50.9	25.1	5.56e6
$1024 \times 1024 \times 1024$	2.61	0.0310	20.7	1.42	10.2	35.0	1.20	1.48	212	104	23.1e6
Buddha, $\gamma = 3$	#faces=1087716, #vertices=543652										
$512 \times 512 \times 512$	14.4	0.593	42.1	0.812	5.24	63.1	18.7	20.3	123	59.3	12.1e6
$1024 \times 1024 \times 1024$	12.3	0.641	88.1	4.08	27.2	132	18.7	21.2	505	243	46.7e6
Buddha, $\gamma = 5$											
$512 \times 512 \times 512$	13.9	0.594	58.7	1.16	8.51	82.9	18.7	21.1	181	88.3	19.8e6
$1024 \times 1024 \times 1024$	11.8	0.625	133	6.77	43.4	196	18.7	22.2	752	366	82.1e6

Table 14.2: Scan conversion statistics for the in-core scan converter. The statistics are divided into three categories: Timings, storage requirements and the (narrow band) grid point count. The timings (seconds) include the mesh load (*Mesh Load*) and partitioning time (*Mesh Part*), actual scan conversion time (*Scan Conv*), lexicographic sort time (*Lexi Sort*), DT-Grid construction time (*DTGrid Const*), and total time (*Tot*). The storage requirements (MB) include the original mesh (*Mesh I*), the intermediate sub-mesh structure (*Mesh II*), the Index-Value Container (*IVC*) storing grid point indices and corresponding signed distance values, and the final (uncompressed) DT-Grid (*DT-Grid*). The narrow band width $\gamma = 3$. All models are courtesy of the Stanford Scanning Repository.

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

Table 14.3: Scan conversion statistics for the out-of-core scan converter. The statistics are divided into three categories: Timings, storage requirements and the narrow band grid-point count. The timings (min::sec) include the mesh load and partitioning time (*Mesh L& P*), actual scan conversion time (*Scan Conv*), lexicographic sort time (*Lexi Sort*), DT-Grid construction time (*DTGrid Const*), and total time (*Tot*). The storage requirements (MB) 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 signed distance values, and the final (uncompressed) DT-Grid (*DT-Grid*). The narrow band width $\gamma = 3$. All models are courtesy of the Stanford Scanning Repository.

14.5 Summary

This chapter presented two methods for converting consistent polygonal meshes into signed distance field level set representations. Both methods leverage on the CSC method [89]. The first method works in-core and allows higher resolution level sets to be generated fast due to the fact that both time- and memory-requirements are proportional to the number of grid points in the narrow band and the number of faces in the mesh. The second method works out-of-core and poses no restrictions on the size of the input mesh nor the output level set, other than enough disk space must be available. The out-of-core algorithm out-performs the in-core algorithm when the storage requirements exceed the amount of physical memory available.

Part V

Applications

Chapter 15

Applications

The preceding chapters covered the technical contributions of this dissertation. In particular we introduced the DT-Grid and the H-RLE for high resolution in-core level set simulations not restricted by the boundaries of a computational domain. Next we presented a generic framework for out-of-core and compressed level set simulations that allowed very high resolution to be achieved on desktop computers. Following that we concentrated on scan conversion and presented efficient methods for generating high resolution level set surfaces from polygonal meshes. These technical contributions *enable* a large number of applications of high resolution level sets, and the techniques have already been taken into use by several other members of the graphics group. In this chapter we briefly review some of these applications and the main goal is to demonstrate the wide applicability, practical feasibility and importance of our work.

A few examples and applications were presented earlier in this dissertation. Chapter 5 demonstrated the Enright Test in effective resolution 1024^2, twice the resolution presented by concurrent work [34]. We also demonstrated by example the ability of the DT-Grid to run out-of-the-box level set simulations, and in this chapter we will see additional applications illustrating this feature. Finally in chapter 10 we demonstrated an example of a huge out-of-core level set shape-metamorphosis run on a desktop computer with 1GB of memory and requiring close to 5GB of storage.

This chapter reviews several other applications and our techniques are applied to shape deformations, ray tracing, fluid simulation, geometric texturing, modeling and animation of snow, volume segmentation and the simulation of PDEs on level set manifolds. However we stress that many other applications are possible, for example collision detection based on signed distance field level sets, surface reconstruction from points and the simulation of large bodies of water [52].

A substantial part of my time has been devoted to implementing from scratch a state-of-the-art level set software framework including the data structures and algorithms presented in this dissertation. In particular *all applications presented in this chapter use my implementations of the DT-Grid, H-RLE and out-of-core and compression framework.* This software now forms part of the shared *GGL* library available to all members of the graphics group. To date my contributions to the library consist of more than 350.000 lines of code [1] and include a wealth of data structures, algorithms, utilities and visualization components.

[1] According to a line count using the unix command *wc -l*

15.1 High Resolution Surface Deformations

15.1.1 Bunny Enright Test

Figure 15.1: Extreme level set deformation of a high resolution model resulting in very thin walls during the course of the simulation. The Stanford bunny is advected by a divergence free and periodic vector field [35, 75] and returns to its original shape after one period, provided the simulation is run at sufficient resolution. The maximum bounding grid is 1024^3, but memory requirements of the H-RLE grid never exceed 97MB. From left to right the simulation times are (in units of the period) 0, 1/15, 1/2, 1. Rendering by Ola Nilsson based on my simulation data. Stanford Bunny courtesy of the Stanford Scanning Repository.

The Stanford 3D Scanning Repository is quite prohibitive in terms of what one is allowed to do with their 3D models, as several of them include religious symbols of some sort. But as they say: *"...You can do anything to the Stanford Bunny."*. We took their word for it and placed the Stanford Bunny in the Enright Test velocity field [35] (see also chapter 5). Since the velocity field is periodic and divergence free (*i.e.* without sources or sinks) a geometric surface should return to its original shape after one period. However, very high grid resolutions are required to faithfully capture the thin level set surface resulting from this deformation. Figure (15.1) shows results from a simulation on a grid of effective resolution 1024^3 using a third order accurate TVD Runge-Kutta [133] time integration and a fifth order accurate HJ-WENO [79] space discretization. Using our effective data structures, this high resolution simulation was achieved with a dynamic memory footprint never exceeding 97 MB.

15.1.2 Shape Metamorphosis

Shape metamorphosis with level sets have previously been restricted to relatively low resolution models [14]. This is primarily due to the unfavorable memory requirements of the applied level set schemes. However, using our scalable H-RLE and DT-Grid level set representations we can perform high resolution morphs between geometric models with relatively fine surface details.

The morphing process from *source* to *target* can be achieved by a propagation of the *source* level set according to [48]

$$\frac{\partial \phi_{source}}{\partial t} + (\phi_{source} - \phi_{target})|\nabla \phi_{source}| = 0 \qquad (15.1)$$

This PDE is propagated in time until steady state, when the two level sets completely overlap. Figure 15.2 shows the result of morphing a human model $512 \times 797 \times 145$ into a statuette at resolution $1000 \times 1676 \times 865$. This morph uses an H-RLE level set to represent both the source and the target distance field resulting in a clamped speed function. Since the target level set is static, it could also be encoded by the adaptive distance field structure of Frisken *et al.* [39].

Figure 15.2: A level set shape metamorphosis. The human figure (left), courtesy of Frantic Films, has a bounding box of $512 \times 797 \times 145$. The Thai Statuette (right), retrieved from the Stanford 3D Scanning Repository, has a bounding box $1000 \times 1676 \times 865$. Rendering and Simulation by Ola Nilsson.

The shape metamorphosis shown in figure 15.2 was rendered and simulated by Ola Nilsson based on my data structure and level set framework implementations.

15.2 Fluid Simulation

The DT-Grid [106] and H-RLE [48] data structures can be applied to represent a significant part of the numerical quantities used in the simulation of free surface fluids [35, 37] hence allowing for higher resolutions fluids. In particular, our data structures are applicable to both fluid and boundary/obstacle level set surfaces as well as volumetric properties such as fluid velocities and pressure. In addition, any number of auxiliary fields such as for example surface and boundary velocities, as well as the marker particles used by the particle level set method [33], can be stored efficiently along with the DT-Grid and H-RLE representations. The primary advantage of employing the DT-Grid and H-RLE in this way is that the storage complexity of the volumetric properties scale with the volume of the fluid interior as opposed to the volume of the enclosing bounding box. The same is true for the computational complexity since both the advection of fluid velocities and the construction and solution of the Poisson matrix [36], required to solve for fluid pressure, are effectively restricted to the voxels in the fluid interior. This improves on traditional methods that are typically required to traverse the entire bounding box to determine the mapping between voxels in the grid and entries in the Poisson matrix. It should be noted that the main computational bottleneck of the fluid solver remains the solution of the Poisson equation itself which has computational complexity $O(N \log N)$ in the multigrid paradigm. The storage and computational requirements of the fluid and boundary *surfaces* scale with surface area when utilizing our data structures, and existing numerical schemes for fluid simulation [35,37], including the MAC grid and particle level set method, can be employed with essentially no change. Furthermore, the fluid inherits the out-of-the-box properties of the DT-Grid and H-RLE hence allowing the fluid to move and spread indefinitely. Figure 15.4 illustrates the potential of out-of-the-box fluid simulation. In this particular example fluid is leaking out

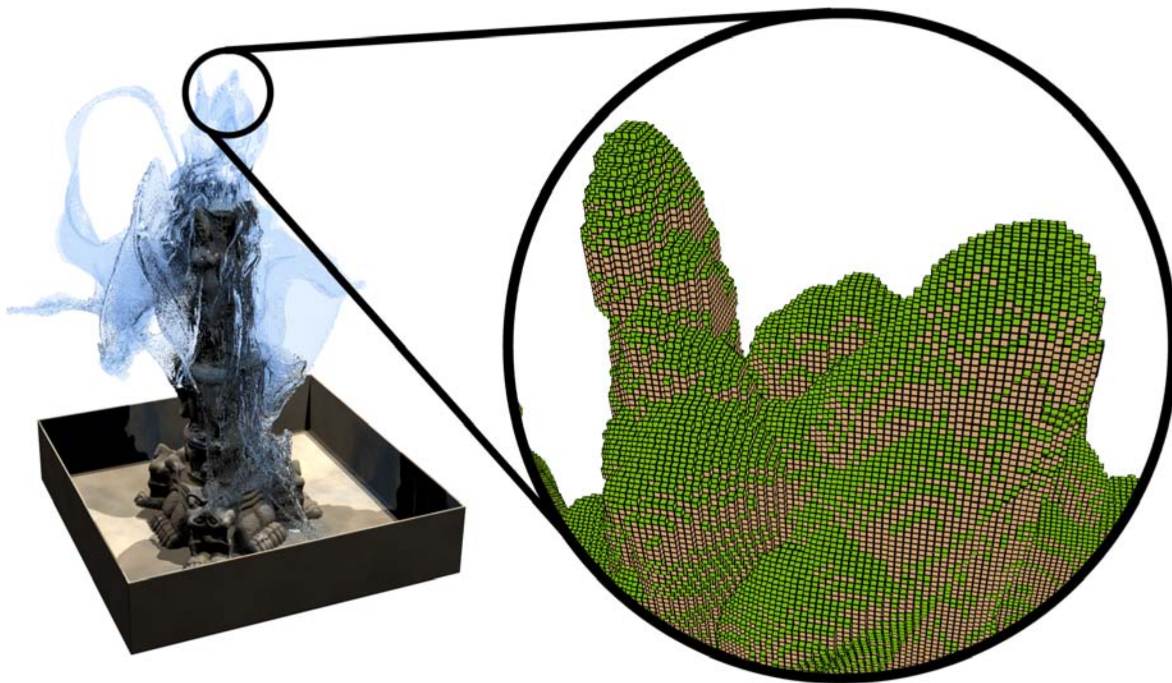


Figure 15.3: Closeup of the fluid simulation in figure 15.5 revealing the underlying DT-Grid based surface and fluid representation.

of a canyon model thus creating a very large effective bounding box. Since the out-of-the-box property is inherent to our data structures and automatically accounted for by the dilation algorithm (see chapter 5), no specific attention has to be paid to this kind of behavior.

It should be noted that since our approach represents the interior of the fluid with uniform grid cells, it is very likely that the octree-based fluid method of Losasso *et al.* [84] is both more memory and computationally efficient since it allows for an adaptive representation of the fluid interior away from the surface. However, the octree fluid method also incurs increased numerical dissipation resulting in viscous flows as identified by Irving *et al.* [52]). Irving *et al.* propose a method that leverages on our DT-Grid and H-RLE techniques for representing large bodies of water by employing vertically enlongated cells in favour of an octree approach.

Since our out-of-core and compression framework does not change the interface of the DT-Grid and H-RLE data structures, these techniques can be applied to fluid simulation as well. In the out-of-core paradigm, fluid and boundary level set surfaces as well as surface velocities and particles can be stored out-of-core hence allowing higher resolution fluid simulations. Figure 15.5 shows a frame from a partially out-of-core fluid simulation of a splashing fountain with an effective resolution close to 512^3. Figure 15.3 depicts the same frame and a closeup revealing the underlying DT-Grid encoding. A similar simulation with effective resolution $931 \times 1567 \times 931$ was shown in chapter 2.

The fluid simulation images in this chapter as well as in chapter 2 were generated using the fluid simulator designed and implemented by Andreas Söderström and Ken Museth. It employs MAC grids [35], DT-Grid (in-core and out-of-core) and marker particles [33] for improved accuracy and surface resolution. A similar approach to fluid simulation and utilizing an RLE based data structure was documented in our H-RLE paper [48].

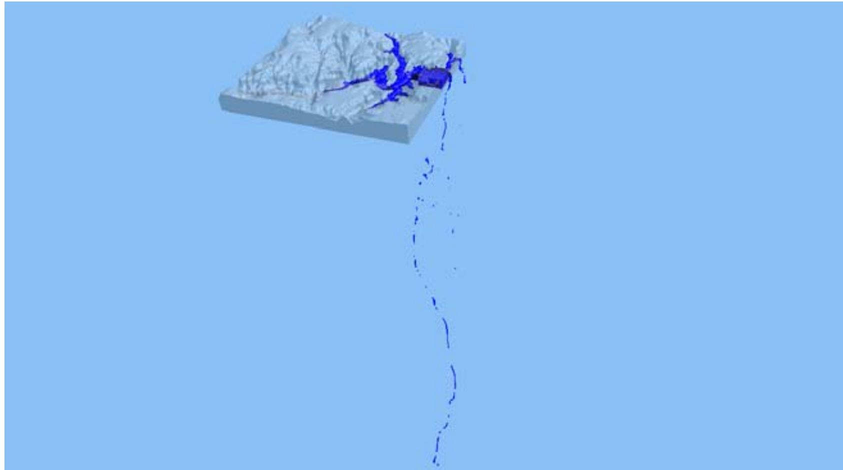


Figure 15.4: Out-Of-The-Box Fluid Simulation. Fluid is leaking out of a canyon model thus creating a very large effective bounding box. The out-of-the-box property is inherent to our data structures and storage requirements scale with the surface area and volume interior of the fluid. Simulation and rendering by Andreas Söderström.

Figure 15.5: Partially out-of-core fountain fluid simulation with effective resolution close to 512^3. Simulation and rendering by Andreas Söderström.

15.3 Ray Tracing

Ola Nilsson implemented a ray tracer for signed distance field level sets which can be used with both the DT-Grid and H-RLE data structures as well as our out-of-core framework. Standard ray tracing techniques [35,85] are directly applicable to the H-RLE and DT-Grid data structures. In particular, since our data structures store samples of a signed distance function this allows for ray leaping. Hence the data structures serve as acceleration structures as well and offer logarithmic random access without the additional build (and memory consumption) of external acceleration structures (*e.g.* kd-trees usually applied for meshes). It should be noted that our data structures only store a narrow band whereas an octree offers the additional advantage of storing distance samples throughout the domain. This can speed up ray leaping at the cost of an increase in memory consumption.

The ray tracing algorithm proceeds by adaptive stepping based on the magnitude of the level set values encountered along the ray's path. Given an accurate signed distance field, it is possible to take discrete steps along the ray's path as large as the magnitude of the level set values encountered until either a grid point immediately adjacent to the interface is discovered (which can be inferred from the level set value being less than the voxel width) or the ray exits the bounding domain of the level set. When near the interface an analytic solution to the ray-interface crossing is found with a cubic polynomial root finder. The normal is computed as the derivative of the cubic interpolation function. If no surface is found in the current narrow band region, ray tracing can once again begin taking large steps.

In practice this allows us to efficiently ray trace very large models on ordinary desktop machines. See for example figures 15.2 and 15.1 as well as the images of the out-of-core shape metamorphosis in chapter 10.

15.4 PDEs on Manifolds

As an application of our out-of-core framework Ola Nilsson and Ken Museth demonstrated how we can solve PDEs directly on large level set surfaces by employing out-of-core linear algebra operations [106]. Specifically we can solve the wave-equation PDE embedded on a surface, see [106] for a detailed description of the equations involved. The waves traveling on the surface are represented as a number of auxiliary scalar fields associated with the particular out-of-core DT-Grid. The several auxiliary fields make the method quite storage intensive. To avoid excessive storage usage the simulated scalar fields can be compressed with any of our proposed value codecs. Examples of the wave equation propagating on complex geometry is shown in figure 15.6 and 15.7. Note that the former visualizes the wave scalar field as actual 3D displacements of the geometry, whereas the latter uses a simple color map. Using the out-of-core framework we have solved the wave equation on surfaces of resolution up to 4096^3.

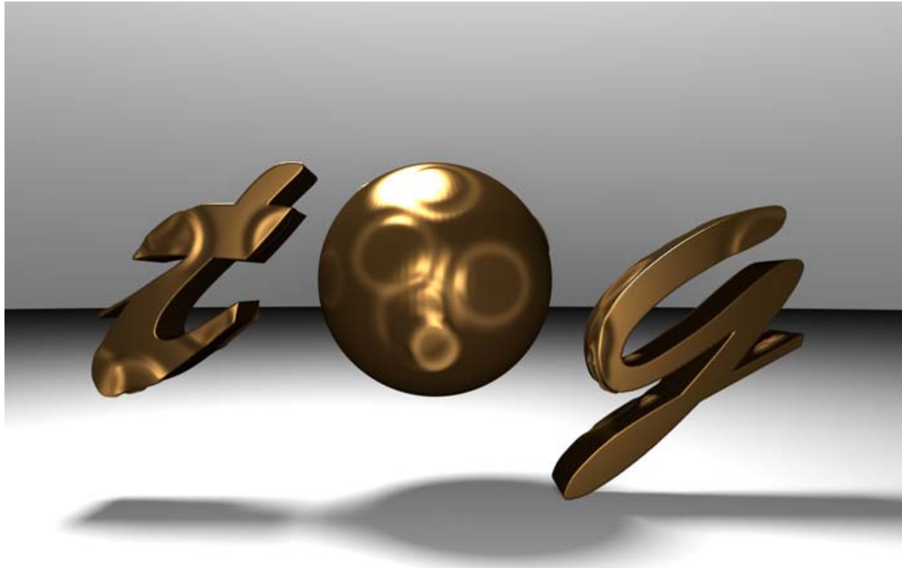


Figure 15.6: The wave equation simulated on "'TOG"' (Transactions On Graphics) 3D text. The wave is visualized both through the displaced surface and surface color properties. Resolution of model 512×250×200. Rendering and simulation by Ola Nilsson.

Figure 15.7: The wave equation on a Siggraph plaque. Resolution of model 1024×250×50. The wave is visualized by color coded surface properties. Rendering and simulation by Ola Nilsson.

15.5 Geometric Texturing

Anders Brodersen and Ken Museth applied DT-Grid based level sets as the fundamental surface representation for *geometric texturing* [17]. In geometric texturing 3D surface textures are applied to a surface instead of the traditional well known 2D textures. The challenges include making the geometric textures warp in order to accommodate the surface curvature and making the algorithm robust in the presence of sharp features such as edges. This must of course be done without introducing too much noticeable distortion into the warped 3D textures. A rendering of a dragon geometrically textured with smaller dragons is shown in figure 15.8. Both the large dragon and the 12 smaller texture dragons are sampled in resolution $512 \times 244 \times 350$ and represented as DT-Grids. Even though the individual level set resolutions are not particularly impressive, the quantity of individual level sets makes a dense uniform grid approach impossible. The DT-Grid is used in several stages of the geometric texturing algorithm. This includes the warping of the texture elements, the blending of the texture elements with the base geometry to obtain a smooth transition and finally for ray tracing of the base geometry augmented with

Figure 15.8: A large dragon geometrically textured with smaller dragons. Each individual dragon level set was represented using a DT-Grid with effective resolution $512 \times 244 \times 350$. Rendering and geometric texturing by Anders Brodersen.

the texture elements. Although not exploited in the current version, unenclosed/open level sets can advantageously be employed during the blending phase to speed up computations.

15.6 Snow Modeling and Simulation

Tommy Hinks and Ken Museth applied the DT-Grid in a level set based framework for animation of wind-driven snow buildup [47]. Figure 15.9 shows an example of a snow-cap generated using this method. In contrast to previous methods the utilization of level sets allows for snow of arbitrary topology and complexity to evolve in the dynamic wind fields. DT-Grids are used to represent both the boundary objects in the scene as well as the topologically complex snow-caps arising from snow buildup accumulating over time. The two main advantages of using the DT-Grid are that the memory footprint of the simulation is reduced dramatically, and that the out-of-the-box feature of the DT-Grid allows for the evolving snow-caps to dynamically expand over time. Although very high resolutions are not demonstrated, the method has the potential of high resolution due to the utilization of the DT-Grid. Furthermore since both boundaries and snow-caps are represented as individual level sets, traditional dense uniform grids would compromise the physical memory limits rather quickly. Although not explored in the current work, this particular application would greatly benefit from the utilization of unenclosed/open level sets, since snow buildup is a local operation. By using unenclosed level sets, expensive global level set operations could be confined to a smaller narrow bands which would result in a speedup of the overall simulation.

Figure 15.9: Level set based snow buildup in dynamic wind-driven velocity fields. Effective resolution of Buddha is 256^3. The Buddha, the ground and the snow-caps are represented as individual DT-Grids. Rendering and simulation by Tommy Hinks.

15.7 Volume Segmentation

The DT-Grid can also be applied to level set based segmentation as demonstrated by Gunnar Johansson and Ken Museth [61]. They present a method that improves the method of level set segmentation without edges by Chan and Vese [19]. In particular Johansson presents a two-stage segmentation framework which relies on an initial surface estimated by topological analysis followed by an iterative level set refinement method. An example level set surface generated from the segmentation of a brain from a human head is shown in figure 15.10. In this case a resolution of $256 \times 256 \times 109$ was used. While this is quite low, it was dictated by the resolution of the input, and in the future the method will be applied to higher resolution input data sets

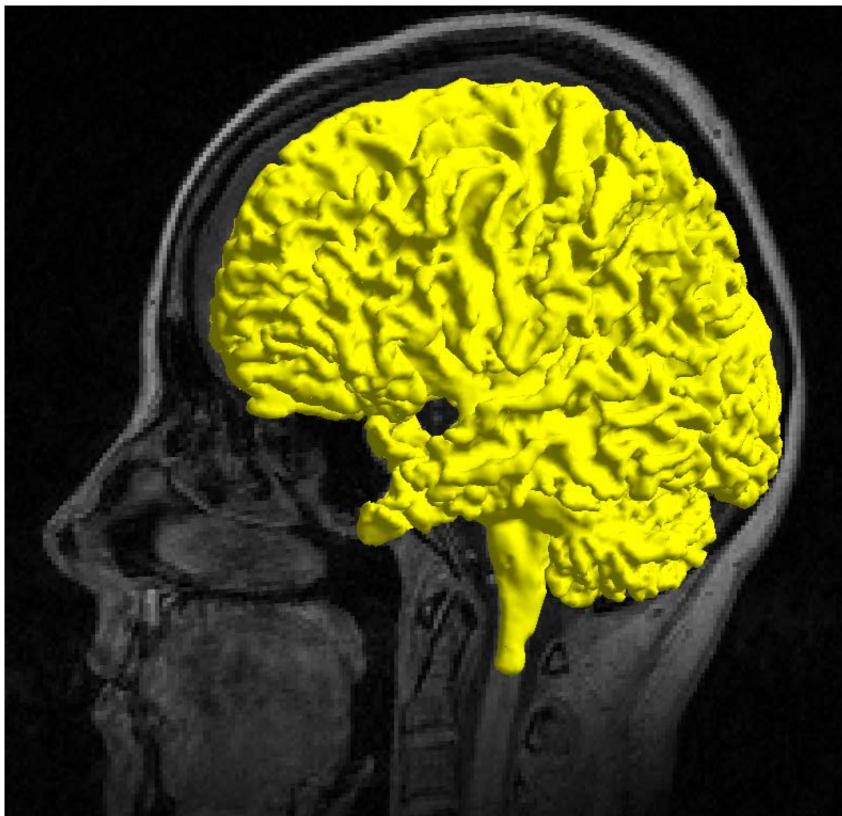


Figure 15.10: A DT-Grid based segmentation of a brain from scannings of a human head. The resolution in this case was restricted to $256 \times 256 \times 109$ due to the size of the input data set. However higher resolution is feasible and will be demonstrated as the methods are applied to higher resolution input data sets in the future. Rendering and level set segmentation by Gunnar Johansson.

such as the *visible human dataset*. Due to the utilization of the DT-Grid data structure, higher resolution is feasible, although the memory bottleneck remains the storage of the volumetric data set being segmented.

15.8 Summary

The DT-Grid and H-RLE data structures as well as the out-of-core and compression level set framework *enable* applications of high resolution level set representations. The techniques have matured rather quickly and my implementations have been used in a large number of level set applications by other members of the graphics group. This chapter briefly reviewed several of these applications including shape deformations, fluid simulations, ray tracing, volume segmentation, geometric texturing and the simulation of PDEs on manifolds.

Part VI

Conclusions and Future Work

Chapter 16

Future Work

The work presented in this dissertation has demonstrated level set representations and deformations at very high resolution. In addition the computational efficiency of level sets was improved and simulations are no longer confined to a predefined computational domain, rather they can move out-of-the-box. Despite these improvements, the demand for higher resolution simulations at lower simulation times is ubiquitous. Clearly, level set (and fluid) simulations are very CPU intensive, and future research should continue to concentrate effort in this field to improve both storage- and speed-requirements. Below we outline research directions likely to contribute to the area of level sets in the future.

Enhancing the Spatial and Temporal Locality of Level Set Computations: Several more or less standard methods for improving the utilization of the cache hierarchies of modern computers have been proposed in the context of solving partial differential equations [29, 30]. These techniques include loop-blocking, loop-fusion as well as array padding. Quite deliberately, such methods have not been considered in this dissertation as an advantage of our work is that it does not require existing level set methods to be re-written. However, the power of properly exploiting the cache hierarchy should not be neglected. In fact we have already seen the power of improved cache coherency in chapter 7 where we demonstrated that the DT-Grid and H-RLE can be faster than previous approaches despite the additional complexity and hence CPU cycles required to access and manipulate these data structures. It is an interesting direction for future work to investigate whether the performance of level set methods on very high resolution surfaces can be significantly improved by considering techniques like loop-fusion. In fact we are already investigating fusing level set advection, reinitialization and narrow band rebuild into a single pass over the data in order to improve both the spatial and temporal locality of level sets. One of the main design challenges is of course to avoid compromising generality with respect to numerical schemes. In addition we find it very likely that these techniques may actually significantly improve the performance of our out-of-core framework since locality is even more important in this field given the greater time-margin between accesses to memory and disk. Reducing the number of passes over the data in out-of-core level set simulation will amortize each IO operation over more CPU cycles hence better exploiting disk bandwidth.

External Memory and Parallel Algorithms: In this dissertation we have demonstrated the potential power of out-of-core techniques applied to level set representations and simulations. However, future work in this area still remains. In particular we wish to explore the feasibility

of representing all parts of the fluid solver in our out-of-core framework hence moving the entire simulation including the solution of the Poisson equation out-of-core. It is very likely that greater attention has to be paid to spatial and temporal locality. Since the computational time of fluid simulations remains a major bottleneck we intend to investigate the combination of parallel and out-of-core methods. The feasibility of parallelization has already been advocated by Irving *et al.* in [52] where they in fact parallelized the DT-Grid and H-RLE techniques presented in this dissertation. Parallelization was also very recently exploited in producing the water visual effects for the feature film "Poseidon". Finally, page-replacement and prefetching strategies should be devised for out-of-core applications using random access patterns such as ray tracing. Preliminary investigations suggest that accesses during ray tracing are in fact far from truly random and it would be interesting to investigate if this could be exploited.

Adaptive Level Set Methods: So far all applications of level sets in the area of computer graphics have sampled the interface uniformly. It would however be interesting to investigate whether adaptive level set methods are feasible for computer graphics. The concept of an adaptive method is attractive because it may lead to savings both with respect to memory and computational requirements. Introducing a non-uniform sampling also raises several issues including: Which properties of the level set should determine where the resolution should be increased or decreased and how should this be detected. Furthermore, the extra book-keeping and error-monitoring associated with an adaptive method should be worthwhile and hence not consume more computational power than saved by solving the level set equations adaptively. Finally, the velocity fields and speed functions governing the interface motion must also be represented adaptively which may not be a simple task in itself. In order for this to be feasible for computer graphics, it should not compromise the flexibility and ease of use of level sets. As mentioned in chapter 3, a few works within computational physics have investigated adaptive level set methods [93, 144]. However, the experimental examples are limited and it is not clear exactly how well these methods function in practice.

New Applications: Finally we intend to apply our techniques to new application areas, not solely within computer graphics, in the future. This includes level set interfaces of higher dimension and co-dimension important in studies of *e.g.* reflections in geometric optics [110,149]. See also [111] for an introduction. Due to the representation of higher (co-)dimensional interfaces and hence an increase in the dimensionality of the computational grids, this research area is hampered by excessive storage requirements. A previous approach targeting this problem by the use of hierarchical trees [94] considers reflections defined on five dimensional grids and allows for a maximal resolution of 64^5. Despite the fact that this resolution seems surprisingly low it should be kept in mind that due to the dimensionality (five), a single grid instance of size 64^5 with four-byte float entries represented on a dense uniform grid would take up 4GB of storage. We believe that the DT-Grid can be applied in this area to reduce both storage requirements and computational efficiency.

The above list of directions for future work is by no means exhaustive. However it does present numerous outstanding problems in direct continuation of our work hitherto. In multiple of the cases above preliminary investigations are already taking place.

Chapter 17

Conclusions

Due to their several advantages, including the ability to describe arbitrary topological changes, level sets have become prevalent not only in computer graphics but also in visual effects production and engineering. As a result, existing level set technology is constantly being pushed to its limits as the demand for larger and more detailed simulations becomes ubiquitous. Three disadvantages hamper the level set method as originally proposed: Computational inefficiency, storage inefficiency and the confinement of deformations to a static predefined domain. Previous work has to some extent addressed these limitations, but storage efficient algorithms tend to lower the computational efficiency and computationally efficient algorithms tend to increase the storage requirements. The research presented in this dissertation addresses these limitations from a computer scientific perspective. In particular the contributions fall into three categories: *Level Set Representations and Algorithms*, *Conversion* and enabled *Level Set Applications*. Below we briefly summarize the contributions to each category in turn.

Level Set Representations and Algorithms: We presented the Dynamic Tubular Grid (DT-Grid) [105], a memory and computationally efficient data structure allowing for high resolution level set simulations. Performance evaluations showed that the DT-Grid requires less storage and is in general faster than previous approaches, including octrees [38, 138] and narrow band methods [120] as well as a concurrently developed RLE based data structure [50]. Additionally the DT-Grid allows for out-of-the-box simulations, can take advantage of existing numerical level set schemes and generalizes to any number of dimensions. Next we introduced the Hierarchical Run-Length Encoded (H-RLE) [48] grid combining the DT-Grid and its algorithms with the run-length encoding of Houston *et al.* [50]. The advantage of this data structure is an increase in versatility over the DT-Grid. In particular, H-RLE allows for flexible encodings, efficient unenclosed level set representations and the decoupling of level set values from the data structure. The H-RLE remains relatively fast, but performs slightly worse than the DT-Grid.

Motivated by the fact that current desktop computers are typically equipped with 1-2 GB of memory and that huge level set simulations take up far more memory than this, we finally proposed a generic out-of-core and compression framework for level set simulations [106] that can be used in conjunction with the DT-Grid and H-RLE representations. The framework outperforms the original state-of-the-art DT-Grid when the DT-Grid must rely on virtual memory. The compression framework can also be applied as an efficient offline streaming compressor which we demonstrated by comparison to existing methods. The out-of-core and compression framework allows for level sets of very high resolutions and to the best of our knowledge no previous work has attempted to apply out-of-core and compression methods to online level set

simulation.

Conversion: As the DT-Grid, H-RLE and the out-of-core and compression framework allow for very high resolution level set representations, it was necessary to consider algorithms for converting polygonal meshes, the most common exchange format for 3D models today, into level sets. We presented two approaches for converting consistent meshes into level sets largely leveraging on the CSC method [89]. The first method works in-core and allows for higher resolutions than previous conversion methods since both storage and time complexity of the method is linear in the number of faces of the polygonal mesh and the grid points in the narrow band. The second conversion algorithm works out-of-core and does not pose restrictions on the size of the input mesh nor the size of the output level set other than enough disk space must be available. For problems that exceed physical memory the out-of-core conversion algorithm was shown to out-perform the in-core method relying on virtual memory.

Level Set Applications: The data structures and algorithms presented in this dissertation *enable* applications of high resolution level sets in areas of computational science where the narrow band level set method can be utilized. In particular the data structures and algorithms can be applied to a variety of problems in computer graphics. This dissertation contributed with several applications including high resolution and out-of-the-box shape deformations as well as out-of-core shape metamorphosis. In addition several applications developed by other researchers within the graphics group and enabled by the research presented here were briefly reviewed. One particularly important application is fluid simulation, one of the most requested and demanding effects in today's feature films. Both the DT-Grid and H-RLE can be utilized for representing many of the components used in fluid simulation. This includes the fluid surface, boundary surfaces, marker particles, interior fluid pressure and velocities as well as boundary and surface velocities. By utilizing our data structures, the computational and storage complexity of a fluid simulation scale with the volume of the fluid interior as opposed to the volume of the enclosing bounding box. Additionally fluid simulations are out-of-the-box and our out-of-core framework can be applied to many of the components involved in the fluid simulation as well. We also reviewed several other applications including volume segmentation, the solution of PDEs on surfaces, geometric texturing, modeling and animation of snow and ray tracing.

The research proposed in this dissertation has been developed over the past three years. It has been used in a variety of applications internally in the graphics group and other researchers have leveraged on the techniques presented here. Furthermore the DT-Grid is in experimental use in at least two major companies. This can be attributed to the fact that the research has addressed and presented solutions to limitations of the level set method widely used in a diversity of areas. In particular the techniques developed in this dissertation enable level sets of very high resolution to be represented and deformed relatively efficiently compared to previous work. This also implies that with respect to high resolution, meshes can no longer be considered superior to the level set representation. The data structures and algorithms presented in this dissertation are likely to form an integral part of many of our future research projects on level sets and fluids, and we foresee many exciting applications and extensions in the near future.

Bibliography

[1] D. Adalsteinsson and J. A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, 1995.

[2] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terasacale computers. In *Proceedings of SC2003*, Phoenix, AZ, Nov. 2003.

[3] J. Anderson. *Computational Fluid Dynamics: The Basics with Applications*. McGraw-Hill, 1995.

[4] J. Baerentzen and H. Aanaes. Signed distance computation using the angle weighted pseudo-normal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, may 2005.

[5] J. A. Baerentzen. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, 2005. Member-Henrik Aanaes.

[6] S. Bansal and D. S. Modha. Car: Clock with adaptive replacement. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[7] J. Bell, M. Berger, J. Saltzman, and M. Welcome. Three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM J. Sci. Comput.*, 15(1):127–138, 1994.

[8] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, 1989.

[9] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53(1):484–512, 1984.

[10] M. Bertalmo, L.-T. Cheng, S. Osher, and G. Sapiro. Variational problems and partial differential equations on implicit surfaces. *J. Comput. Phys.*, 174(2):759–780, 2001.

[11] J. Bloomenthal, C. Bajaj, J. Blinn, M.-P. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wyvill. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc, 1997.

[12] D. Breen, R. Fedkiw, K. Museth, S. Osher, G. Sapiro, and R. Whitaker. *Level Sets and PDE Methods for Computer Graphics*. ACM SIGGRAPH '04 COURSE #27. ACM SIGGRAPH, Los Angeles, CA, August 2004. ISBN 1-58113-950-X.

[13] D. E. Breen, S. Mauch, and R. T. Whitaker. 3d scan conversion of csg models into distance volumes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 7–14, New York, NY, USA, 1998. ACM Press.

[14] D. E. Breen and R. T. Whitaker. A level-set approach for the metamorphosis of solid models. *IEEE Trans. Vis. Comput. Graph.*, 7(2):173–192, 2001.

[15] R. Bridson. *Computational aspects of dynamic surfaces*. PhD thesis, Stanford University, 2003.

[16] R. Bridson, S. Marino, and R. Fedkiw. Simulation of clothing with folds and wrinkles. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, pages 28–36. Eurographics Association, 2003.

[17] A. Brodersen, K. Museth, S. Porumbescu, and B. Budge. Geometric texturing using level sets. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):277–288, 2008.

[18] A. D. Brown. *Explicit Compiler-based Memory Management for Out-of-core Applications*. PhD thesis, Carnegie Mellon University, 2005.

[19] T. F. Chan and L. A. Vese. Active contours without edges. *Image Processing, IEEE Transactions on*, 10(2):266–277, 2001.

[20] Y.-J. Chiang and C. T. Silva. I/o optimal isosurface extraction (extended abstract). In *IEEE Visualization*, pages 293–300, 1997.

[21] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 286–295, New York, NY, USA, 2000. ACM Press.

[22] D. L. Chopp. Computing minimal surfaces via level set curvature flow. *Journal of Computational Physics*, 106:77–91, 1993.

[23] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[24] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. Technical report, Paris, France, France, 1996.

[25] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 235–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[26] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, New York, NY, USA, 1996. ACM Press.

[27] M. de Berg. *Computational Geometry*. Springer, January 2000.

[28] M. Desbrun, M. Meyer, P. Schröder, and A. H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[29] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Portable memory hierarchy techniques for PDE solvers, part I. *SIAM News*, 33(5), 2000.

[30] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Portable memory hierarchy techniques for PDE solvers, part II. *SIAM News*, 33(6), 2000.

[31] M. Droske, B. Meyer, M. Rumpf, and C. Schaller. An adaptive level set method for medical image segmentation. *Lecture Notes in Computer Science*, pages 412–422, 2001.

[32] I. Eckstein, M. Desbrun, and C.-C. J. Kuo. Compression of time varying isosurfaces. In *GI '06: Proceedings of Graphics Interface 2006*, pages 99–105, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[33] D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.*, 183(1):83–116, 2002.

[34] D. Enright, F. Losasso, and R. Fedkiw. A fast and accurate semi-lagrangian particle level set method. *Computers and Structures*, 83:479–490, Feb. 2005.

[35] D. Enright, S. Marschner, and R. Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, pages 736–744. ACM, ACM Press / ACM SIGGRAPH, 2002.

[36] R. Fedkiw, J. Stam, and H. W. Jensen. Visual simulation of smoke. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 15–22, Aug. 2001.

[37] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 23–30. ACM SIGGRAPH, ACM Press / ACM SIGGRAPH, Aug. 2001.

[38] S. F. Frisken and R. Perry. Simple and efficient traversal methods for quadtrees and octrees. *journal of graphics, gpu, and game tools*, 7(3):1–11, 2002.

[39] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 249–254. ACM, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[40] F. Gibou, R. Fedkiw, R. Caflisch, and S. Osher. A level set approach for the numerical simulation of dendritic growth. *J. Sci. Comput.*, 19(1-3):183–199, 2003.

[41] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in matlab: design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992.

[42] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *SIGMETRICS '97: Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 115–126, New York, NY, USA, 1997. ACM Press.

[43] E. Gobbetti and F. Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph.*, 24(3):878–885, 2005.

[44] S. Godunov. A finite difference method for the computation of discontinuous solutions of the equations of fluid dynamics. *Mat. Sb.*, 47:271–306, 1959.

[45] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Trans. Graph.*, 22(3):871–878, 2003.

[46] A. Harten, B. Engquist, S. Osher, and S. R. Chakravarthy. Uniformly high order accurate essentially non-oscillatory schemes, iii. *J. Comput. Phys.*, 131(1):3–47, 1997.

[47] T. Hinks. Animating wind-driven snow buildup using an implicit approach. *Linköping Electronic Articles in Computer and Information Science*, (ISRN LITH-ITN-MT-EX–06/036–SE), 2006.

[48] B. Houston, M. Nielsen, C. Batty, O. Nilsson, and K. Museth. Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation. *ACM Transactions on Graphics*, 25(1):1–24, 2006.

[49] B. Houston, M. B. Nielsen, C. Batty, O. Nilsson, and K. Museth. Gigantic deformable surfaces. In *Proceedings of the SIGGRAPH 2005 Conference on Sketches & Applications*. ACM, ACM Press, 2005.

[50] B. Houston, M. Wiebe, and C. Batty. RLE sparse level sets. In *Proceedings of the SIGGRAPH 2004 Conference on Sketches & Applications*. ACM, ACM Press, 2004.

[51] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. 22(3):343–348, Sept. 2003.

[52] G. Irving, E. Guendelman, F. Losasso, and R. Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Trans. Graph.*, 25(3):805–811, 2006.

[53] G. Irving, J. Teran, and R. Fedkiw. Invertible finite elements for robust simulation of large deformation. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 131–140, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[54] M. Isenburg. Compression and streaming of polygon meshes. Technical report, UNC, Nov. 2004. PhD thesis.

[55] M. Isenburg and P. Alliez. Compressing hexahedral volume meshes. *Computer Graphics and Applications, Pacific Conference on*, 0:284, 2002.

[56] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM Trans. Graph.*, 22(3):935–942, 2003.

[57] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. In *Symposium on Geometry Processing*, pages 111–118, 2005.

[58] G.-S. Jiang and D. Peng. Weighted eno schemes for hamilton–jacobi equations. *SIAM J. Sci. Comput.*, 21(6):2126–2143, 1999.

[59] G.-S. Jiang and C.-W. Shu. Efficient implementation of weighted eno schemes. *J. Comput. Phys.*, 126(1):202–228, 1996.

[60] S. Jiang and X. Zhang. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 31–42, New York, NY, USA, 2002. ACM Press.

[61] G. Johansson. Local level set segmentation with topological structures. *Linköping Electronic Articles in Computer and Information Science*, (ISRN LITH-ITN-MT-EX–06/030–SE), 2006.

[62] D. E. Johnson and E. Cohen. A framework for efficient minimum distance computations. In *ICRA*, pages 3678–3684, 1998.

[63] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[64] M. W. Jones. Distance field compression. In *WSCG*, pages 199–204, 2004.

[65] T. Ju, F. Losasso, S. Schaefer, and J. Warren. Dual contouring of hermite data. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 339–346, New York, NY, USA, 2002. ACM Press.

[66] F. Kälberer, K. Polthier, U. Reitebuch, and M. Wardetzky. Free Lence - Coding with free valences. *Computer Graphics Forum*, 24(Issue 3):469–478, 2005.

[67] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C.-S. Kim. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *OSDI*, pages 119–134, 2000.

[68] D. Kincaid and W. Cheney. *Numerical analysis: mathematics of scientific computing.* Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1991.

[69] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature sensitive surface extraction from volume data. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 15–22, Aug. 2001.

[70] S. Krishnamoorthy, G. Baumgartner, C.-C. Lam, J. Nieplocha, and P. Sadayappan. Efficient layout transformation for disk-based multidimensional arrays. In *HiPC*, pages 386–398, 2004.

[71] D. E. Laney, M. Bertram, M. A. Duchaineau, and N. Max. Multiresolution distance volumes for progressive surface compression. In *3DPVT*, pages 470–479, 2002.

[72] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 134–143, New York, NY, USA, 1999. ACM Press.

[73] H. Lee, M. Desbrun, and P. Schroder. Progressive encoding of complex isosurfaces. *ACM Trans. Graph.*, 22(3):471–476, 2003.

[74] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 11, Washington, DC, USA, 2003. IEEE Computer Society.

[75] R. LeVeque. High-resolution conservative algorithms for advection in incompressible flow. *SIAM J. Numer. Anal.*, 33:627–665, 1996.

[76] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[77] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications.* 1997.

[78] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[79] X. Liu, S. Osher, and T. Chan. Weighted essentially nonoscillatory schemes. *J. Comput. Phys.*, 115:200–212, 1994.

[80] X.-D. Liu, S. Osher, and T. Chan. Weighted essentially non-oscillatory schemes. *J. Comput. Phys.*, 115(1):200–212, 1994.

[81] J. C. Lopez, D. R. O'Hallaron, and T. Tu. Big wins with small application-aware caches. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 20, Washington, DC, USA, 2004. IEEE Computer Society.

[82] W. Lorenson and H. Cline. Marching Cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (Proc. SIGGRAPH)*, 21(4):163–169, 1982.

[83] F. Losasso, R. Fedkiw, and S. Osher. Spatially adaptive techniques for level set methods and incompressible flow. *Computers & Fluids*, In Press, Corrected Proof.

[84] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 457–462, New York, NY, USA, 2004. ACM.

[85] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 100–107, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[86] A. Mascarenhas, M. Isenburg, V. Pascucci, and J. Snoeyink. Encoding volumetric grids for streaming isosurface extraction. In *Proceedings of 3DPVT'04*, pages 665–672, 2004.

[87] Y. Matias, E. Segal, and J. S. Vitter. Efficient bundle sorting. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 839–848, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[88] S. Mauch. A fast algorithm for computing the closest point and distance transform. *http://www.acm.caltech.edu/ seanm/software/cpt/cpt.pdf*, 2000.

[89] S. Mauch. *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, California Institute of Technology, 2003.

[90] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[91] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[92] S. Meyers. *Effective C++ (2nd ed.): 50 specific ways to improve your programs and designs*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[93] R. B. Milne. *An Adaptive Level-Set Method*. PhD thesis, University of California, Berkeley, 1995.

[94] C. Min. Local level set method in high dimension and codimension. *Journal of Computational Physics*, 200:368–382, 2004.

[95] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

[96] K. Museth, D. Breen, R. Whitaker, and A. Barr. Level set surface editing operators. *ACM Trans. on Graphics (Proc. SIGGRAPH)*, 21(3):330–338, July 2002.

[97] K. Museth, D. Breen, R. Whitaker, S. Mauch, and D. Johnson. Algorithms for interactive editing of level set models. *Computer Graphics Forum*, 24(4):821–841, 2005.

[98] D. Q. Nguyen, R. Fedkiw, and H. W. Jensen. Physically based modeling and animation of fire. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 721–728, New York, NY, USA, 2002. ACM Press.

[99] K. G. Nguyen and D. Saupe. Rapid high quality compression of volume data for visualization. *Comput. Graph. Forum*, 20(3), 2001.

[100] M. B. Nielsen and A. Brodersen. Inverse rendering under uncontrolled illumination. Master's thesis, University of Aarhus, 2002.

[101] M. B. Nielsen and A. Brodersen. Inverse rendering of polished materials under constant complex uncontrolled illumination. *Journal of WSCG*, 12:309–316, 2004.

[102] M. B. Nielsen, G. Kramp, and K. Grønbæk. Mobile augmented reality support for architects based on feature tracking techniques. In *International Conference on Computational Science*, pages 921–928, 2004.

[103] M. B. Nielsen and K. Museth. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *Linköping Electronic Articles in Computer and Information Science*, 9(001):ISSN 1401–9841, 2004.

[104] M. B. Nielsen and K. Museth. An optimized, grid independent, narrow band data structure for high resolution level sets. In *Proceedings of SIGRAD 2004*, november 2004.

[105] M. B. Nielsen and K. Museth. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *Journal of Scientific Computing*, 26(3):261–299, 2006. (submitted November, 2004; accepted January, 2005).

[106] M. B. Nielsen, O. Nilsson, A. Söderström, and K. Museth. Out-of-core and compressed level set methods. (in submission).

[107] M. B. Nielsen, O. Nilsson, A. Söderström, and K. Museth. Virtually infinite resolution deformable surfaces. In *Proceedings of the SIGGRAPH 2006 Conference on Sketches & Applications*. ACM, ACM Press, 2006.

[108] O. Nilsson, D. Breen, and K. Museth. Surface reconstruction via contour metamorphosis: An eulerian approach with lagrangian particle tracking. *Visualization Conference, IEEE*, 0:52, 2005.

[109] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[110] S. Osher, L.-T. Cheng, M. Kang, H. Shim, and Y.-H. Tsai. Geometric optics in a phase-space-based level set and eulerian framework. *J. Comput. Phys.*, 179(2):622–648, 2002.

[111] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, 2003.

[112] S. Osher and N. Paragios. *Geometric Level Set Methods in Imaging,Vision,and Graphics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[113] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[114] S. Osher and C. Shu. High-order essentially nonoscillatory schemes for Hamilton-Jacobi equations. *SIAM J. Num. Anal.*, 28:907–922, 1991.

[115] R. Pajarola. Stream-processing points. *Visualization Conference, IEEE*, 0:31, 2005.

[116] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 2–2, New York, NY, USA, 2001. ACM Press.

[117] S. Patel, A. Chu, J. Cohen, and F. Pighin. Fluid simulation via disjoint translating grids. In *Proceedings of the SIGGRAPH 2005 Conference on Sketches & Applications*. ACM, ACM Press, 2005.

[118] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95. ACM Press, December 1995.

[119] B. A. Payne and A. W. Toga. Distance field manipulation of surface models. *IEEE Comput. Graph. Appl.*, 12(1):65–71, 1992.

[120] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang. A PDE-based fast local level set method. *J. Comput. Phys.*, 155(2):410–438, 1999.

[121] R. N. Perry and S. F. Frisken. Kizamu: a system for sculpting digital characters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 47–56, New York, NY, USA, 2001. ACM Press.

[122] N. Rasmussen, D. Enright, D. Nguyen, S. Marino, N. Sumner, W. Geiger, S. Hoon, and R. Fedkiw. Directable photorealistic liquids. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 193–202, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[123] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142, New York, NY, USA, 1990. ACM Press.

[124] E. Rouy and A. Tourin. A viscosity solutions approach to shape-from-shading. *SIAM J. Numer. Anal.*, 29(3):867–884, 1992.

[125] M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. pages III: 1103–1106, 2001.

[126] J. K. Salmon and M. S. Warren. Parallel, out-of-core methods for n-body simulation. In *PPSC*, 1997.

[127] K. Sayood. *Introduction to data compression*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[128] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. In *FAST*, 2005.

[129] K. E. Seamons and M. Winslett. Multidimensional array i/o in panda 1.0. *J. Supercomput.*, 10(2):191–211, 1996.

[130] J. Sethian. Level set methods and fast marching methods: Evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science. *Robotica*, 18(1):89–92, 1999.

[131] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. of the National Academy of Sciences of the USA*, 93(4):1591–1595, February 1996.

[132] M. Shah, J. M. Cohen, S. Patel, P. Lee, and F. Pighin. Extended Galilean invariance for adaptive fluid simulation. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Computer Animation*. ACM, The Eurographics Association, 2004.

[133] C. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock capturing schemes. *J. Comput. Phys.*, 77:439–471, 1988.

[134] C. Sigg, R. Peikert, and M. Gross. Signed distance transform using graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 12, Washington, DC, USA, 2003. IEEE Computer Society.

[135] C. Silva, Y. jen Chiang, W. Corrêa, J. El-sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *In Visualization 02 Course Notes*, 2002.

[136] Y. Smaragdakis, S. Kaplan, and P. Wilson. Eelru: simple and effective adaptive page replacement. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 122–133, New York, NY, USA, 1999. ACM Press.

[137] J. Stam. Stable fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 121–128, Aug. 1999.

[138] N. Stolte and A. Kaufman. Parallel spatial enumeration of implicit surfaces using interval arithmetic for octree generation and its direct visualization. In *Implicit Surfaces '98*, pages 81–87, 1998.

[139] J. Strain. Fast tree-based redistancing for level set computations. *J. Comput. Phys.*, 152(2):664–686, 1999.

[140] J. Strain. Semi-lagrangian methods for level set equations. *J. Comput. Phys.*, 151(2):498–533, 1999.

[141] J. Strain. Tree methods for moving interfaces. *J. Comput. Phys.*, 151(2):616–648, 1999.

[142] J. Strain. A fast modular semi-lagrangian method for moving interfaces. *J. Comput. Phys.*, 161(2):512–536, 2000.

[143] J. C. Strikwerda. *Finite difference schemes and partial differential equations.* Wadsworth Publ. Co., Belmont, CA, USA, 1989.

[144] M. Sussman, A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome. An adaptive level set approach for incompressible two-phase flows. *J. Comput. Phys.*, 148(1):81–124, 1999.

[145] M. Sussman, P. Smereka, and S. Osher. A level set approach for computing solutions to incompressible two-phase flow. *J. Comput. Phys.*, 114(1):146–159, 1994.

[146] A. S. Tanenbaum. *Modern operating systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[147] H.-Z. Tang, T. Tang, and P. Zhang. An adaptive mesh redistribution method for nonlinear hamilton–jacobi equations in two-and three-dimensions. *J. Comput. Phys.*, 188(2):543–572, 2003.

[148] G. Taubin. Blic: bi-level isosurface compression. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 451–458, Washington, DC, USA, 2002. IEEE Computer Society.

[149] L. tien Cheng, M. Kang, S. Osher, H. Shim, and Y. hsi Tsai. Reflection in a level set framework for geometric optics. *CMES Comput. Model. Eng. Sci*, 5:347–360, 2004.

[150] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. pages 161–179, 1999.

[151] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface*, pages 26–34, 1998.

[152] H. Trac and U.-L. Pen. Out-of-core hydrodynamic simulations for cosmological applications. *New Astronomy*, 11(4):273 – 286, 2006.

[153] Y. Tsai, L. Cheng, S. Osher, and H. Zhao. Fast sweeping algorithms for a class of Hamilton-Jacobi equations. *SIAM Journal on Numerical Analysis*, 41(2):673–694, 1998.

[154] J. N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *Proceedings of the 33rd Conference on Decision and Control, Lake Buena Vista, LF*, pages 1368–1373, December 1994.

[155] J. N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Trans. Automat. Contr.*, 40:1528–1538, September 1995.

[156] T. Tu, R. O'Hallaron, and C. Lopez. Etree: a database-oriented method for generating large octree meshes. *Eng. with Comput.*, 20(2):117–128, 2004.

[157] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[158] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[159] H. Wang, P. J. Mucha, and G. Turk. Water drops on surfaces. *ACM Trans. Graph.*, 24(3):921–929, 2005.

[160] R. Whitaker, D. Breen, K. Museth, and N. Soni. A framework for level set segmentation of volume datasets. In *Volume Graphics 2001 Proceedings*, pages 159–168, 2001.

[161] R. T. Whitaker. A level-set approach to 3d reconstruction from range data. *Int. J. Comput. Vision*, 29(3):203–231, 1998.

[162] C.-K. Yang and T.-C. Chiueh. Integration of volume decompression and out-of-core isosurface extraction from irregular volume data. *Vis. Comput.*, 22(4):249–265, 2006.

[163] H.-K. Zhao, S. Osher, and R. Fedkiw. Fast surface reconstruction using the level set method. In *VLSM '01: Proceedings of the IEEE Workshop on Variational and Level Set Methods (VLSM'01)*, page 194, Washington, DC, USA, 2001. IEEE Computer Society.

[164] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):505–519, 2004.