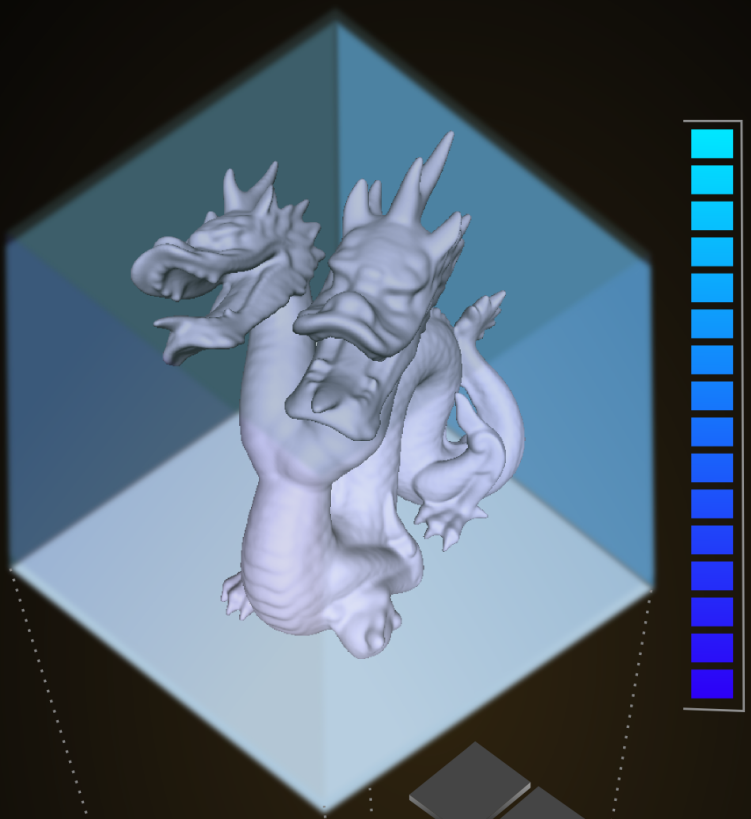
Multiresolution Volume Processing and Visualization on Graphics Hardware



Wladimir van der Laan

This research was financially supported by the Netherlands Organisation for Scientific Research (NWO) under project number 643.100.501, as part of the VIEW (Visual Interactive Effective Worlds) program.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

van der Laan, W. J.

Multiresolution Volume Processing and Visualization on Graphics Hardware
Wladimir J. van der Laan
Thesis Rijksuniversiteit Groningen. - With ref.
ISBN 978-90-367-4616-8

RIJKSUNIVERSITEIT GRONINGEN

Multiresolution Volume Processing and Visualization on Graphics Hardware

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. F. Zwarts,
in het openbaar te verdedigen op
vrijdag 4 februari 2011
om 16:15 uur

door

Wladimir Jasper van der Laan
geboren op 15 juni 1981
te Groningen

Promotor:	Prof. dr. J. B. T. M. Roerdink
Copromotor:	Dr. A. C. Jalba
Beoordelingscommissie:	Prof. dr. D. Weiskopf Prof. dr. ir. J. J. van Wijk Prof. dr. S. Zaroubi

ISBN: 978-90-367-4616-8

Contents

1 Introduction **1**
- 1.1 Coping with the data stream in interactive visualization . . . 1
- 1.2 Interactive morphological and wavelet-based volume processing and visualization 2
- 1.3 Level sets . . . 6
- 1.4 General purpose computation on graphics hardware . . . 6
- 1.5 Thesis contributions and organization . . . 8

2 Multiresolution MIP rendering on graphics hardware **11**
- 2.1 Introduction . . . 11
- 2.2 Previous and related work . . . 12
- 2.3 Overview of the multiresolution MIP algorithm . . . 13
  - 2.3.1 Morphological operators . . . 13
  - 2.3.2 Pyramids . . . 13
  - 2.3.3 Multiresolution MIP algorithm . . . 14
  - 2.3.4 Streaming MIP . . . 15
- 2.4 Implementation on graphics hardware . . . 16
  - 2.4.1 Per-voxel projection . . . 16
  - 2.4.2 Representing the detail coefficients . . . 16
  - 2.4.3 Projecting the detail coefficients . . . 17
  - 2.4.4 Load balancing . . . 18
  - 2.4.5 Streaming MIP . . . 18
  - 2.4.6 Optimized streaming MIP . . . 19
  - 2.4.7 Post-processing . . . 19
- 2.5 Results . . . 19
- 2.6 Discussion . . . 22
- 2.7 Conclusion . . . 24

3 Accelerating Wavelet Lifting on Graphics Hardware using CUDA **25**
- 3.1 Introduction . . . 25
- 3.2 Previous and related work . . . 27
- 3.3 Wavelet lifting . . . 28
  - 3.3.1 Wavelet transform by subband filtering . . . 28
  - 3.3.2 Wavelet transform by lifting . . . 29

3.4 Wavelet lifting on GPUs using CUDA . . . 31
3.4.1 CUDA overview . . . 31
3.4.2 Performance considerations for parallel CUDA programs (kernels) . . . 33
3.4.3 Parallel wavelet lifting . . . 36
3.4.4 Separable wavelets . . . 36
3.4.5 Horizontal pass . . . 37
3.4.6 Vertical pass . . . 39
3.4.7 3-D and higher dimensions . . . 42
3.5 Results . . . 43
3.5.1 Wavelet filters used for benchmarking . . . 43
3.5.2 Experimental results and comparison to other methods . . . 44
3.5.3 Performance Analysis . . . 49
3.6 Conclusion . . . 52

4 Accelerating Wavelet-Based Video Coding on Graphics Hardware 53
4.1 Introduction . . . 53
4.2 CUDA-based implementation of the DWT . . . 54
4.2.1 CUDA overview . . . 54
4.2.2 Wavelet lifting . . . 55
4.2.3 Wavelet lifting in CUDA . . . 55
4.3 Accelerating the Dirac Video Codec . . . 56
4.3.1 Motion compensation . . . 58
4.3.2 Frame arithmetic . . . 60
4.4 Performance results . . . 60
4.5 Conclusion . . . 61

5 Screen Space Fluid Rendering with Curvature Flow 63
5.1 Introduction . . . 63
5.2 Related work . . . 64
5.3 Method . . . 65
5.3.1 Surface depth . . . 66
5.3.2 Smoothing methods . . . 66
5.3.3 Thickness . . . 68
5.3.4 Noise . . . 68
5.3.5 Rendering . . . 69
5.4 Results and discussion . . . 71
5.5 Conclusions and future work . . . 72

6 A Memory and Computation Efficient Sparse Level-Set Method 81
6.1 Introduction . . . 81
6.2 Previous and related work . . . 82
6.3 Overview of the level set method . . . 84
6.3.1 Sparse-grid level set representations . . . 85

6.3.2 Reshaping the level set function . . . 85
6.4 The proposed method . . . 86
6.4.1 The data structure . . . 87
6.4.2 Initialization . . . 87
6.4.3 Append operation . . . 88
6.4.4 Sequential access with stencil . . . 88
6.4.5 Random access . . . 89
6.4.6 Tile management . . . 89
6.4.7 Updating the level-set . . . 93
6.5 Results . . . 97
6.5.1 Mean curvature flow . . . 97
6.5.2 Volume-conserving mean curvature flow . . . 97
6.5.3 Memory usage . . . 100
6.5.4 Periodic velocity field advection . . . 100
6.5.5 Tile size considerations . . . 102
6.5.6 Tile management overhead . . . 102
6.5.7 Discussion of our method . . . 103
6.5.8 Parallelization over multiple CPUs . . . 103
6.6 Conclusions and future work . . . 104

7 Real-Time Sparse Level-Sets on Graphics Hardware 105
7.1 Introduction . . . 105
7.2 Previous and related work . . . 106
7.2.1 Efficient level set methods on the CPU . . . 106
7.2.2 Level set GPU methods . . . 107
7.2.3 Sparse CPU methods . . . 107
7.2.4 Surface reconstruction . . . 108
7.3 Proposed GPU level set method . . . 108
7.3.1 Generic level set equation . . . 108
7.3.2 CPU STL method . . . 109
7.3.3 GPU sparse level sets . . . 110
7.3.4 Rendering the interface using CUDA and OpenGL . . . 117
7.4 Proposed surface reconstruction method . . . 118
7.4.1 Efficiency and multi-resolution . . . 119
7.5 Comparison with previous approaches . . . 120
7.6 Results . . . 122
7.6.1 Efficiency: Comparison to other methods . . . 122
7.6.2 Surface reconstruction . . . 123
7.6.3 Interactive level-set surface editing . . . 127
7.6.4 Limitations . . . 128
7.7 Conclusions and future work . . . 129

8 Concluding remarks **131**
8.1 Summary and Conclusions 131
8.2 Future outlook 132
8.2.1 GPUs 132
8.2.2 Computer Graphics APIs 133
8.2.3 CUDA 133
8.2.4 GPGPU for embedded systems 134

Bibliography **137**

Publications **151**

Samenvatting **153**

Dankwoord **155**

Chapter 1

Introduction

1.1 Coping with the data stream in interactive visualization

Visualization of large data sets requires advanced techniques in image processing, hierarchical data management, and data reduction. This is the case for facilities covering a wide range, from classical medical imaging to simulation of natural phenomena. Data volumes generated by scientific simulations can easily grow into the range of giga-bytes. Medical scanners routinely generate data volumes with a resolution of 512^3 voxels or higher, whereas scanners like multi-slice CT generate more than 1000 image slices in one scan. In a functional neuroimaging experiment (PET, fMRI), a large number of data volumes is obtained, thus increasing data size even further. Both the increasing size and complexity of these data ask for new techniques for interactive visualization. The possibility of interaction during evaluation will significantly reduce the time required to interpret and present results.

Between data acquisition and display of the final image on the screen, a structured sequence of steps are executed in the *visualization pipeline* [**?**]. Several processing steps can be included in this pipeline in order to expose relevant data features. In the mapping step, data values are mapped to graphical attributes such as color and opacity. Advanced multidimensional transfer functions can be adapted by the user through interactive tools [65]. However, increasingly one employs extensive *volume processing* before the mapping step, such as filtering, classification, segmentation and feature extraction of the raw data.

To arrive at interactivity for large data sets, a number of techniques are available, such as: (i) auxiliary data structures; (ii) special (multiresolution) data transforms, e.g., Fast Fourier Transform, wavelets or multiscale morphological methods (see below); (iii) extraction of *features* from the volume data, and visualizing the features only. Although these techniques all have their merit, the development of new algorithms remains necessary because data sizes remain growing at a fast pace, and demands on processing capabilities are especially high in exploratory visualization, where the user wants to adjust application parameters and resource allocation in an on-line fashion. In such interactive data visualization, the speed of the data processing stage (data filtering, feature extraction, segmentation, etc.) should be comparable to that of the visualization step. Otherwise, the computational burden will render the overall technique ineffective and far from interactive. Therefore, the efficiency of the data-processing method represents another important issue which should be addressed.

A recent development which is having a major impact on interactive data processing and visualization, is new programmable graphics hardware. Through the use of faster memory access rates, intrinsic data parallelism and high arithmetic intensity, modern yet affordable high-performance graphics processors (GPUs) are already capable of outperforming current CPUs in certain computationally intensive applications (relevant for visualization), and the performance gap is expected to increase rapidly. GPU processing is no longer restricted to the mapping and rendering stages of the visualization pipeline, but can now also be fruitfully applied in the data processing stage and even the data generation stage (in the case of simulations). Nevertheless, there are still technological obstacles which need to be overcome. A major limitation is the relatively small amount of memory (1 GB) available on current graphics cards, which greatly restricts the amount of data which can be loaded into the texture memory. One solution is to use clusters of computers equipped with modern graphics hardware [128]. The recent PCI Express bus (`http://www.pcisig.com`), with double the bandwidth of the AGP 8X graphics bus, accelerates data swapping between texture memory and main computer memory. This allows for storing large data sets in main memory, and chunks from the main memory are uploaded in texture memory only when needed.

Obviously, a fast method is not of much help when it produces unreliable results. Therefore, evaluation of the effectiveness of new methods is also of prime importance. Here one may think of issues such as accuracy of approximate representations, correctness of segmentations, perceptual quality of visualizations, user-friendliness of interactive tools, etc. A major problem is that it is very hard to come up with good evaluation methodologies, although the need for them is increasingly recognized in the visualization community.

1.2 Interactive morphological and wavelet-based volume processing and visualization

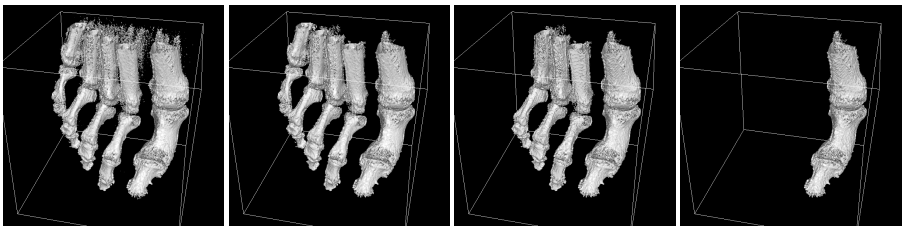
Coupled morphological volume processing and feature based volume visualization

Volume processing usually is carried out by operations which are 3D extensions of well-known signal or image processing algorithms. An important class of such algorithms are the so-called morphological filters or operators. In contrast to the classical linear filters, morphological filters are nonlinear, making their mathematical analysis much harder. Fortunately, a unified theoretical framework, called mathematical morphology [48, 114, 115], has been developed to describe and analyze these filters, both for binary and grey-scale or color images.

Today, algorithms for morphological image operations are available within any state-of-the-art image processing toolbox. Features extracted by morphological techniques may subsequently be used in a pattern recognition step [57–59, 146, 155]. A notable advantage of morphological operators is that they can work in the integer domain, which gives computational savings since no floating-point operations are required. This is very important when one wants to implement these operations on graphics hardware. Effectiveness may be improved by integrating the two

steps of data processing and data visualization into a single step in the processing pipeline. For this purpose, shared data structures for processing and rendering are required; cf. Fig.1.1.

Although some work has been done on applying morphological techniques in computer graphics and visualization [75, 80, 85, 108, 110], a large body of morphological techniques remains unexplored in this area. Given their excellent feature extraction capabilities, investigating the usefulness of these techniques in interactive volume processing and visualization is a very promising research area. Although this thesis focuses primarily on *generic* techniques, we will test our methods in various application domains such as medical imaging, but the methods will also be relevant for areas like feature-based flow visualization [103].



$\lambda = 0$ $\lambda = 10000$ $\lambda = 60000$ $\lambda = 90000$

Figure 1.1. *Volumetric filtering of the foot dataset using* volume *as an attribute [151]. The iso-value threshold t is fixed to $t = 80$. As the threshold λ increases, more and more objects disappear. For $\lambda = 10000$ most of the noise still visible in the left-most image disappears. Ultimately, only the largest components remain, i.e., the metatarsal and phalanges of the great toe.*

Multiscale visualization based on adaptive wavelets and morphological pyramids

Wavelets provide an excellent mathematical framework for systematic decomposition of data into versions at different levels of detail. By suppressing small wavelet coefficients corresponding to noise or uninteresting details in the data, a much more sparse data representation is obtained, making wavelets an interesting tool for data compression and denoising. Wavelet-based visualization provides the user with views of a coarse approximation of the data, which can be refined on demand. Examples are wavelet ray-casting [42, 45, 55, 152], wavelet splatting [41] and extensions thereof [148, 149], and wavelet-based Fourier volume rendering [43, 147, 150]. Recently, the combination of a wavelet-based octree representation and rendering by 3-D texture mapping has been introduced [46].

Goutsias and Heijmans [39,49] developed a general framework for multiresolution decomposition of multidimensional data, which includes both linear wavelet analysis, linear and morphological pyramids and morphological wavelets as special cases. In contrast to standard wavelets which require floating point computations, morphological pyramids and morphological wavelets have the advantage that they require only fast integer-arithmetic operations.

For the case of nonlinear pyramids (see Fig. 1.2), approximations $\{f_j\}$ of increasingly reduced size are computed by a reduction operation:

$$f_j = \text{REDUCE}\,(f_{j-1}),\ j = 1, 2, \ldots L.$$

Here j is called the level of the decomposition, and f_0 is the initial data set. In the case of a Gaussian pyramid, the REDUCE operation consists of Gaussian low-pass filtering followed by downsampling [15]. By taking the (generalized) difference between f_j and an expanded version of f_{j+1}, detail signals $d_j = f_j \dot{-} \text{EXPAND}\,(f_{j+1})$ are computed. Under mild conditions, one has *perfect reconstruction*, that is, f_0 can be exactly reconstructed by the recursion:

$$f_j = \text{EXPAND}\,(f_{j+1}) \dot{+} d_j,\ j = L-1, \ldots, 0.$$

For the linear case (Laplacian pyramid), $\dot{+}$ and $\dot{-}$ are ordinary addition and subtraction, and the EXPAND operation consists of upsampling followed by Gaussian low-pass filtering [15]. In the case of morphological pyramids, $\dot{+}$ is the maximum operation (and $\dot{-}$ is defined such that perfect reconstruction will hold) and the REDUCE and EXPAND operations involve morphological filtering [39, 49].

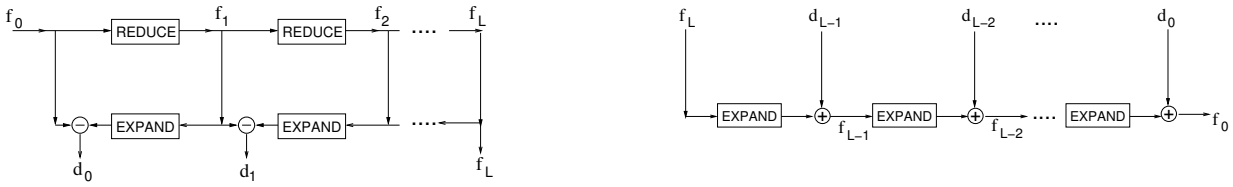
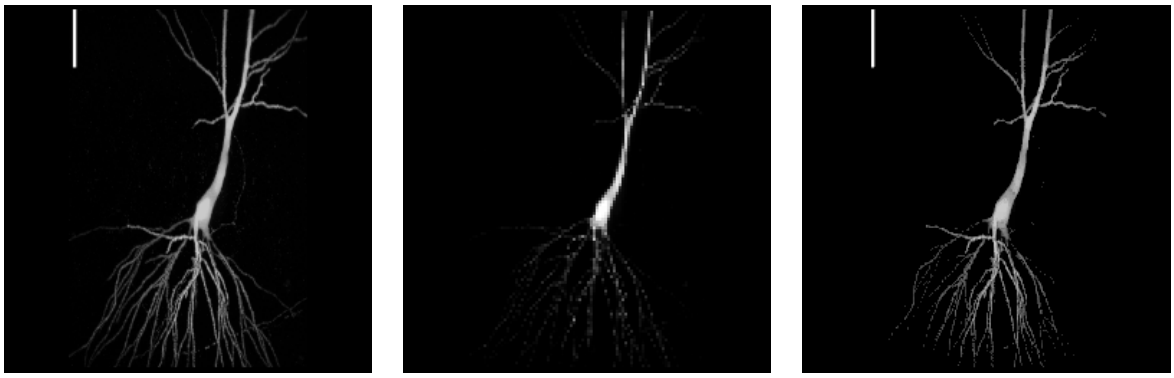


Figure 1.2. *Pyramid scheme; left: analysis, right: synthesis.*

Building upon this work, a multiresolution approach based on morphological pyramids was developed by Roerdink and applied to maximum intensity projection (MIP) volume rendering, often used in visualization of medical angiography data sets [104–106]. This allows integrated filtering and fast data exploration (based on reduced pyramid data). Such user interaction may take the form of setting window and level, a common practice in a medical setting, or rendering only thin slices of data, as in sliding thin slab visualization [140]. Special care is required in choosing the morphological operators to ensure that (i) the MIP operation can be carried out in the transformed domain; (ii) the EXPAND operations are only carried out in the 2D image plane; (iii) detail coefficients can be efficiently coded (only nonzero voxels should be coded, which may save up to 95% of memory for sparse data) [104, 106]. Further improvements are possible by using morphological connectivity preserving filters in the construction of the pyramid [105, 107]. The best results so far have been obtained by *streaming MIP-splatting*, which resorts all detail coefficients in decreasing magnitude and projects only those coefficients needed to attain a user-defined accuracy [107], see Fig. 1.3.



(a) original (b) level-1 (c) streaming

Figure 1.3. *Multiresolution MIP reconstruction from a 2-level morphological pyramid. (a): full-scale MIP; (b) level-1 MIP; (c) streaming MIP. The absolute errors are 0.676 for (b) and 0.301 for (c). The fraction of nonzero detail coefficients taken into account is* 3.4% *in both cases [107].*

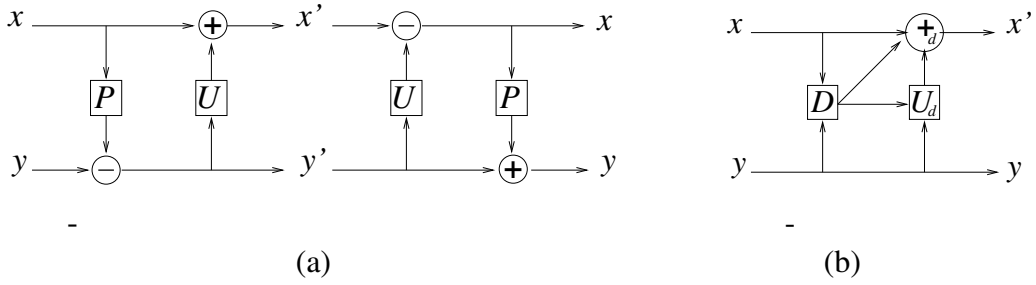


Figure 1.4. *(a) Lifting scheme (left: analysis, right: synthesis). (b) Adaptive update lifting scheme (analysis phase).*

Wavelet lifting

A general and flexible approach for the construction of nonlinear (morphological) wavelets is the *lifting* technique of Sweldens [131, 133]. Starting with a given approximation part x and a detail part y, a *prediction* operator acting on the input x is used to modify y, resulting in a new detail signal y'. Next, an *update* operator U acting on y' modifies x, leading to a new approximation signal x'. In the synthesis phase, the signals x and y are exactly recovered by reversing the lifting steps, see Fig. 1.4(a). One may concatenate any number of lifting and prediction steps to improve an original decomposition.

An extension to wavelet lifting is *adaptive* lifting, where update and/or prediction filters adapt themselves to the local data content. In [102] a special update lifting scheme is presented which allows perfect reconstruction without any additional bookkeeping, see Fig. 1.4(b). Here a

decision map D is used whose output is a set of parameters d at each data location, which govern the choice of the update step, and also the addition step which produces x'.

An important property of the lifting scheme is that it allows *in-place* calculation, so that no additional memory has to be allocated. This feature is extremely useful in handling large data sets, and is yet another motivation for applying this technique in interactive volume processing and visualization.

1.3 Level sets

It has been shown [50] that in the continuous case, an implicitly defined surface (curve) that expands/shrinks with constant speed corresponds to a flat morphological dilation/erosion of the underlying implicit function. In fact, materializations of these ideas are usually provided within the level-set framework [97], which has found applications in a wide variety of scientific fields, ranging from chemistry and physics to computer vision and graphics. For example, in computer vision, most state-of-the-art segmentation techniques are based on level sets to steer the evolving contour or surface towards the objects of interest [138].

The main idea of the level set method is to represent the dynamic interface (e.g., contour, surface, etc.) implicitly and embed it as the zero level set of a time-dependent, higher-dimensional function. Then, evolving the interface with a given velocity in the normal direction becomes equivalent to solving a time-dependent partial differential equation (PDE) for the embedding level set function, see Fig. 1.5. The main advantage of the level set method is that it allows the interface to undergo arbitrary topological changes, which is much more difficult to handle using explicit representations.

The flexibility offered by the level set method does not come for free. One has to solve the time-dependent level-set PDE in a higher-dimensional space than that of the embedded interface, which makes the solution slow to compute. Also, the memory requirements are higher than the size of the interface, as one needs to explicitly store a uniform Cartesian grid for solving the level set PDE.

To address these issues, a number of techniques have been proposed, such as the narrow-band schemes [2,21]. Such methods rely on the fact that it suffices to solve the PDE only in the vicinity of the interface in order to preserve the embedding. Thus, the computational requirements scale with the size of the interface.

1.4 General purpose computation on graphics hardware

In the last five years the growth in computation speed of single CPU cores has nearly halted. Physical limitations make it impossible, or at least prohibitively expensive, to increase the clock speed further above 4 GHz. Also, the additional benefit of dedicating more transistors to deeper pipelines, bigger caches, and more advanced branch prediction is decreasing quickly [130]. The ever-increasing demand for computational power creates a necessity for parallel computing. Chip makers have resorted to adding an increasing number of cores to a chip (multi-core).

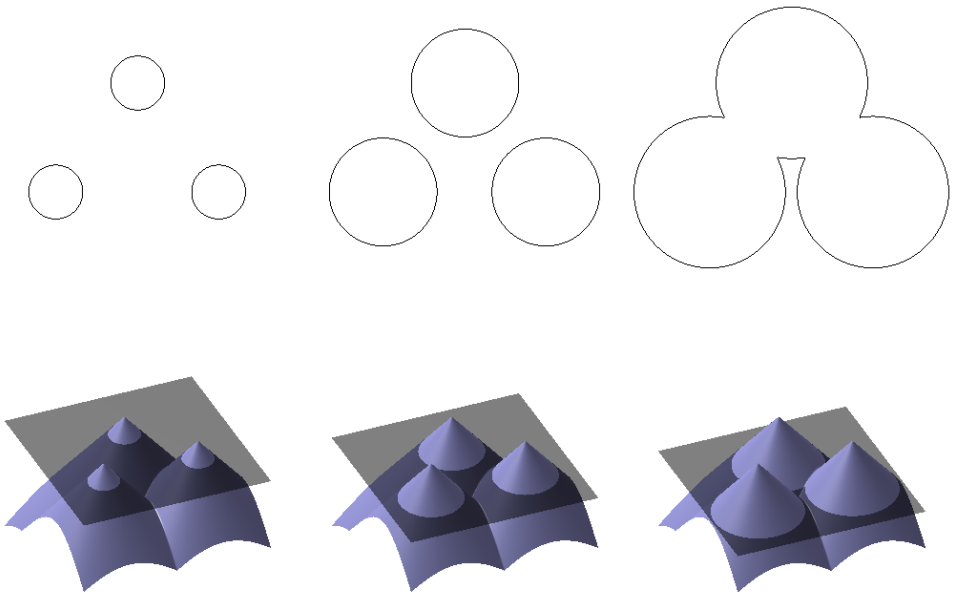


Figure 1.5. *Level-set example in 2D.* Top row*: evolving interface.* Bottom row*: corresponding graphs of the embedding function at three different time steps;* left-to-right*: three initial contours expand at constant speed and eventually merge. The interface is given by the intersection of a plane (indicated in dark grey) with the graph of the function.*

In order to utilize these new cores, software must be adapted. Although basic auto-parallelization is part of every modern compiler, the general problem of deciding how to parallelize a serial program is too hard for a compiler to solve [130]. Additionally, in many cases, specialized parallel algorithms perform better than blindly parallelized serial algorithms. This means that the developer has to "think in parallel", and hence, parallel computing has returned as an active area of research.

Multi-core parallelism is only one approach to parallelism. In recent years, GPUs have become increasingly powerful and more programmable. The GPU has moved from being used solely for graphical tasks to a fully-fledged parallel co-processor, a so-called General Purpose GPU (GPGPU). Due to the high number of threads that can run at once (more than 1000), the GPU excels at fine-grain parallelism. From a high level of generalization GPUs can be regarded as stream processors. The computing power of GPUs is currently increasing at a faster pace than that of CPUs, which has led to the use of the GPU as the main computation device for diverse applications, such as physics simulations, neural networks, image compression and even database sorting.

Compute Unified Device Architecture (CUDA)

Initially, GPGPU applications, even though not concerned with graphics rendering, did need to use the rendering paradigm. With their G80 series of graphics processors in 2006, NVidia introduced a programming environment called CUDA [70] which allows the GPU to be programmed through more traditional means: a C-like language and compiler. This opened up the world of GPGPU to researchers and developers outside Computer Graphics.

In addition to this generalization, CUDA also adds some features that are missing in shader languages: random access to memory, fast integer arithmetic, bitwise operations, and shared memory. Unlike previous attempts to generalize GPGPU computing such as Brook [14], the usage of CUDA does not add any overhead, as it is a native interface to the hardware.

Even though GPU programming has grown to be more like conventional programming the CUDA execution model is quite different from that of CPUs, and also different from that of older GPUs. A degree of specialization, beyond parallel programming, is still required to develop algorithms that perform efficiently on GPU hardware.

In conventional multi-core parallelism each thread has a fully functional, independent core. This is not generally the case in GPU programming, which has multiple levels of parallelism, with specific features and limitations at each level. Originally, GPUs follow a data-parallel model of computation, and this is still broadly the case [70].

The CPU invokes the GPU by calling a *kernel*, which is a C-function that is executed by every thread. The invocation of this kernel creates a *grid*, which is the highest level of parallelism. The grid consists of *blocks* that execute in parallel, if multiprocessors are available, or sequentially if this condition is not met. Blocks within a grid cannot communicate with each other, which is unlikely to change as independent blocks are a means to scalability.

The lowest level of parallelism is formed by *threads*. As in conventional parallel programming, a thread is a single scalar execution unit. Threads within a block can cooperate efficiently by sharing data through fast shared memory. Synchronization points (barriers) can be used to coordinate operations closely. Even though each thread can perform different operations, the highest performance is realized if all threads within a (implementation dependent subset) of a block take the same execution path.

1.5 Thesis contributions and organization

This thesis describes a number of methods and algorithms for real time processing and visualization of large data sets.

Chapter 2 introduces a multiresolution representation for maximum intensity projection (MIP) volume rendering based on morphological pyramids, which allows progressive refinement. We consider two algorithms for progressive rendering from the morphological pyramid: one which projects detail coefficients level by level, and a second one, called streaming MIP, which resorts the detail coefficients of all levels simultaneously with respect to decreasing magnitude of a suitable error measure. The latter method outperforms the level-by-level method, both with respect to image quality with a fixed amount of detail data, and in terms of flexibility of controlling

approximation error or computation time. We improve the streaming MIP algorithm, present a GPU implementation for both methods, and perform a comparison with existing CPU and GPU implementations.

Chapter 3 shows that the Discrete Wavelet Transform (DWT), which has a wide range of applications from signal processing to video and image compression, can be performed by means of the lifting scheme in a memory and computation efficient way on modern, programmable GPUs, which can be regarded as massively parallel co-processors through NVidia's CUDA compute paradigm. The three main hardware architectures for the 2D DWT (row-column, line-based, block-based) are shown to be unsuitable for a CUDA implementation. Our CUDA-specific design can be regarded as a hybrid method between the row-column and block-based methods. We achieve considerable speedups compared to an *optimized* CPU implementation and earlier non-CUDA based GPU DWT methods, both for 2D images and 3D volume data. Additionally, memory usage can be reduced significantly compared to previous GPU DWT methods. The method is scalable and the fastest GPU implementation among the methods considered. A performance analysis shows that the results of our CUDA-specific design are in close agreement with our theoretical complexity analysis.

Chapter 4 integrates the scalable and fast GPU Discrete Wavelet Transform implementation of the previous chapter into the Dirac Wavelet Video Codec (DWVC), which is a royalty-free open-source wavelet-based video codec. The overlapped block motion compensation and frame arithmetic have been accelerated using CUDA as well.

Chapter 5 presents an approach for rendering the surface of particle-based fluids that is simple to implement, has real-time performance with a configurable speed/quality trade-off, and smoothes the surface to prevent the fluid from looking "blobby" or jelly-like. The method is not based on polygonization and as such circumvents the usual grid artifacts of marching cubes. It only renders the surface where it is visible, and has inherent view-dependent level-of-detail. Perlin noise is used to add detail to the surface of the fluid. All the processing, rendering and shading steps are directly implemented on graphics hardware.

Chapter 6 presents efficient data structures and algorithms for tracking dynamic interfaces through the level set method, which has become the favorite technique for capturing and tracking moving interfaces, and found applications in a wide variety of scientific fields. Several approaches which address both computational and memory requirements have been very recently introduced. We show that our method is up to 7.2 times faster than these recent approaches. More importantly, our algorithm can greatly benefit from both fine- and coarse-grain parallelization by leveraging SIMD and/or multi-core parallel architectures.

In chapter 7 we leverage the increased computing power of graphics processors, to achieve fast simulations based on level sets. Our highly-efficient, sparse level set method is about two orders of magnitude faster than other state-of-the art techniques. To further investigate its efficiency, we present two standard graphics applications: surface reconstruction and level-set surface editing. Our novel multi-resolution method for surface reconstruction from unorganized point clouds compares favorably with recent, existing techniques and other parallel implementations. Additionally, we show that free-form editing operations and rendering of level-set surfaces can be performed at interactive rates, even on large volumetric grids. Finally, virtually any other application based on level sets can benefit from our sparse level set method.

Finally, in chapter 8 we summarize our work, draw some general conclusions and present a future outlook.

First published as: Wladimir J. van der Laan, Andrei C. Jalba, and Jos B. T. M. Roerdink. Multiresolution MIP Rendering of Large Volumetric Data Accelerated on Graphics Hardware. In *Proc. Eurographics/IEEE VGTC Symposium on Visualization (EuroVis)*, pages 243-250, 2007.

Chapter 2

Multiresolution MIP rendering on graphics hardware

2.1 Introduction

This chapter is concerned with the development of efficient algorithms for Maximum Intensity Projection (MIP) on graphics hardware. MIP is a method frequently used to visualize volumetric data originating from magnetic resonance angiography (MRA) and ultrasound data. As scanner precision increases larger datasets are generated, for which a large amount of memory and processing capacity is consumed for representing and rendering these volumes. Hence, research is necessary into more efficient MIP rendering algorithms that maintain image quality, e.g., have a fixed error bound. An established method to obtain this goal is the use of multiresolution methods. Since the MIP transform is nonlinear, the standard linear multiresolution models based on wavelets [44, 71, 147] are not applicable. Instead, morphological pyramids involving nonlinear filtering operations can be used for multiresolution rendering.

We present novel MIP algorithms which exploit the programmability of modern graphics hardware. The class of algorithms which we investigate makes use of morphological pyramids as the underlying representation of the volumetric data. Our algorithms are based on the so-called Multiresolution Maximum Intensity Projection (MMIP) method developed by one of us [106, 107]. This method enables the user to visualize large datasets in real time with progressive refinement. After computing the pyramid representation in a preprocessing step, the pyramid levels can be projected individually for progressive rendering. In preview mode, the lower levels of the pyramid are projected first to show a coarse approximation, which can be quickly refined to the original on demand.

A disadvantage of projecting one level at a time is that the approximation quality improves in discrete jumps, as determined by the levels of the pyramid. To overcome this, *streaming MIP* was introduced in [107], which is based on resorting the detail coefficients of all pyramid levels simultaneously with respect to decreasing magnitude of a suitable error measure. All resorted coefficients are projected successively, until a desired accuracy of the resulting MIP image is obtained, thus allowing for continuous error control.

The main contributions are:

- An improved method for computing streaming MIP rendering

- A GPU implementation of the level-by-level MMIP and the streaming MIP algorithms, with the advantage that we can spread the load and the dataset over multiple graphics cards in a straightforward way, thereby achieving support for large volume data with an almost optimal speedup.

We perform comparisons with existing CPU and GPU techniques for MIP volume rendering on large datasets and show the improvement which we obtain by our approach.

2.2 Previous and related work

In the case of MIP a multiresolution decomposition scheme cannot rely on linear operations. Therefore, in [104] a morphological pyramid scheme was proposed for MIP volume rendering with progressive refinement. Such pyramids, which involve nonlinear spatial filtering by morphological operators, systematically split the volume data into approximation and detail signals [39]. As the level of the pyramid is increased, spatial features of increasing size are extracted. Morphological pyramids combine feature extraction with accelerated rendering in preview mode. A disadvantage of the above method is that the approximation accuracy makes a jump each time as an additional level of detail signals is taken into account. To allow for continuous error control, the streaming MIP method was proposed in [107] and it was found to outperform the MMIP method, both with respect to image quality with a fixed amount of detail data, and in terms of flexibility of controlling approximation error or computation time.

The type of morphological pyramid considered here is appropriate in the context of MIP because the morphological operation of dilation (involving the computation of maxima of voxel values in a local neighborhood) is compatible with the maximum computation involved in MIP, just as linear pyramids or wavelet representations [71, 147] are the right tool for the case of linear X-ray rendering. Even though the morphological operators are nonlinear and non-invertible, the pyramid scheme does allow perfect reconstruction. Therefore, after the pyramid has been constructed the original volume data can be discarded. Also, only integer computations are required.

To allow for compression domain rendering, it is essential to use a (fast) MIP implementation which can work directly on the data structures used to represent the pyramid. This can be achieved by a voxel projection method with an efficient volume data storage scheme [86], see section 2.4.1. This method is analogous to point-based rendering (PBR) which has been used for both surface and volume representations [112]. A major challenge of PBR algorithms is accurate interpolation between discrete point samples. We achieve this by means of an additional morphological closing of the output image.

Previous work on mapping elementary morphological operations to graphics hardware in the context of volume rendering and analysis can be found in [51].

2.3 Overview of the multiresolution MIP algorithm

We first define some elementary morphological operators [114]. Next we introduce morphological pyramids, in particular adjunction pyramids. Finally, we will discuss how morphological pyramids are used for efficient MIP rendering.

2.3.1 Morphological operators

Let f be a signal with domain $F \subseteq \mathbb{Z}^d$, and A a subset of $\mathbb{Z}^d$ called the *structuring element*. The *dilation* $d_A(f)$ and *erosion* $\varepsilon_A(f)$ of f by A are defined by

$$d_A(f)(x) = \max_{y \in A, x-y \in F} f(x-y), \tag{2.1}$$

$$\varepsilon_A(f)(x) = \min_{y \in A, x+y \in F} f(x+y). \tag{2.2}$$

Dilation and erosion simply replace each signal value by the maximum or minimum in a neighborhood defined by the structuring element A. The *opening* $\alpha_A(f)$ and *closing* $\phi_A(f)$ of f by A are defined by $\alpha_A(f)(x) = d_A(\varepsilon_A(f))(x)$, and $\phi_A(f)(x) = \varepsilon_A(d_A(f))(x)$. So openings and closings are products of a dilation and an erosion. The opening eliminates signal peaks, the closing valleys.

2.3.2 Pyramids

The general structure of (non)linear pyramids is as follows. From an initial (2-D or 3-D) data set f_0, approximations $\{f_j\}$ of increasingly reduced size are computed by a reduction operation:

$$f_j = \text{REDUCE}\,(f_{j-1}), \qquad j = 1, 2, \ldots L.$$

Here j is called the level of the decomposition. An approximation signal associated to f_{j+1} may be defined by taking the difference between f_j and an expanded version of f_{j+1}:

$$d_j = f_j \mathbin{\dot{-}} \text{EXPAND}\,(f_{j+1}). \tag{2.3}$$

The set $d_0, d_1, \ldots, d_{L-1}, f_L$ is referred to as a *detail pyramid*. Here $\dot{-}$ is a generalized subtraction operator (see below). Assuming there exists an associated generalized addition operator $\dot{+}$ such that, for all j,

$$\hat{f}_j \mathbin{\dot{+}} (f_j \mathbin{\dot{-}} \hat{f}_j) = f_j, \;\; \text{where } \hat{f}_j = \text{EXPAND}(\,\text{REDUCE}\,(f_j)),$$

we have *perfect reconstruction*, that is, f_0 can be exactly reconstructed by the recursion

$$f_j = \text{EXPAND}\,(f_{j+1}) \mathbin{\dot{+}} d_j, \quad j = L-1, \ldots, 0. \tag{2.4}$$

To guarantee that information lost during analysis can be recovered in the synthesis phase in a non-redundant way, one needs the so-called *pyramid condition*:

$$\text{REDUCE}\,(\text{EXPAND}\,(f)) = f, \text{ for all } f. \tag{2.5}$$

In the case of morphological pyramids, the REDUCE and EXPAND operations involve morphological filtering [39]. For the simplest case of the so-called adjunction pyramids [104], the morphological operators are the dilation $d_A(f)$ and erosion $\varepsilon_A(f)$ with structuring element A defined in (2.1) and (2.2), respectively. Then the REDUCE and EXPAND operators are denoted by $\psi_A^{\uparrow}$ and $\psi_A^{\downarrow}$, respectively, and have the form

$$\text{REDUCE} : \psi_A^{\uparrow}(f) = \text{DOWNSAMPLE}\,(\varepsilon_A(f)), \tag{2.6}$$

$$\text{EXPAND} : \psi_A^{\downarrow}(f) = d_A\,(\text{UPSAMPLE}\,(f)), \tag{2.7}$$

where the arrows indicate transformations to higher (coarser) or lower (finer) levels of the pyramid. Here DOWNSAMPLE and UPSAMPLE denote downsampling and upsampling by a factor of 2 in each spatial dimension. The pyramid condition (2.5) is satisfied, if there exists an $a \in A$ such that the translates of a over an even number of grid steps are never contained in the structuring element A; see [39] for more details.

In an adjunction pyramid, the product $\psi_A^{\downarrow}\psi_A^{\uparrow}$ is an *opening*. The anti-extensivity property of openings [48] implies that $\psi_A^{\downarrow}\psi_A^{\uparrow}(f) \leq f$. Therefore, we can define the generalized addition and subtraction operators $\dot{+}$ and $\dot{-}$ appearing in (2.3) by (cf. [39]):

$$t \mathbin{\dot{+}} s = t \vee s = \max(t, s), \qquad t \mathbin{\dot{-}} s = \begin{cases} t, & \text{if } t > s \\ 0, & \text{if } t = s \end{cases} \tag{2.8}$$

where 0 is the smallest image or voxel value possible. As a consequence, the detail signals are nonnegative:

$$d_j(n) = f_j(n) \mathbin{\dot{-}} \psi_A^{\downarrow}\,\psi_A^{\uparrow}\,(f_j)(n) \geq 0. \tag{2.9}$$

Note that the definition of $\dot{-}$ in (2.8) implies that the detail signal $d_j(n)$ equals $f_j(n)$, except at points n for which $f_j(n) = \psi_A^{\downarrow}\,\psi_A^{\uparrow}\,(f_j)(n)$, where $d_j(n) = 0$. So, detail signals are not 'small' in regions where the structuring element does not fit well to the data.

For an adjunction pyramid with the generalized addition being defined as the maximum operation (see (2.8)), the reconstruction takes a special form [106]:

$$f = \psi_A^{\downarrow L}(f_L) \vee \bigvee_{k=0}^{L-1} \psi_A^{\downarrow k}(d_k). \tag{2.10}$$

Here L is the decomposition depth, $\psi_A^{\downarrow k}$ denotes k-fold composition of $\psi_A^{\downarrow}$ with itself, and $\vee$ denotes the maximum operator.

2.3.3 Multiresolution MIP algorithm

The adjunction pyramid representation does allow to interchange the MIP operator (computing maxima along the line of sight) with the pyramidal synthesis operator, because both the upsampling operation and the dilation d_A commute with the maximum operation [104,106]. As a result, the MIP operation can be performed on a coarse level (reduced data size) before performing a

cheap 2-D EXPAND operation to a finer resolution, thus leading to a computationally efficient algorithm.

We write the MIP operation as $\mathcal{M}_{\mathbf{\Theta}}$, with $\mathbf{\Theta} = (\theta, \phi, \alpha)$, where θ and ϕ are the two angles defining the projection direction vector perpendicular to the view plane, and α gives the orientation of the view plane with respect to this vector. Successive approximations of the MIP of f are denoted by $\hat{\mathbf{M}}_{\mathbf{\Theta}}^{(j)}(f)$, $j = L, L-1, \ldots, 0$. These approximations all have the size of the MIP of the full data f in the image plane.

The MMIP algorithm for an adjunction pyramid is as follows. From a level-j approximation, the next approximation on level $j-1$ is obtained by first computing the MIP of d_{j-1}, then $j-1$ times applying the 2-D pyramid synthesis operator $\psi_{\tilde{A}}^{\downarrow}$ to the projection, and finally taking the maximum of the image so obtained with the previous approximation. Here $\psi_{\tilde{A}}^{\downarrow}$ is a 2-D EXPAND operator which has the same form as (2.7), that is, 2-D upsampling followed by a 2-D dilation, but with a structuring element $\tilde{A}$ which is the MIP of A, that is, $\tilde{A} := \mathcal{M}_{\mathbf{\Theta}}(A)$. It is clear that $\hat{\mathbf{M}}_{\mathbf{\Theta}}^{(j-1)}(f) \geq \hat{\mathbf{M}}_{\mathbf{\Theta}}^{(j)}(f)$, since from (2.9) the details signals d_{j-1} are nonnegative. So the projections increase pointwise as one goes down the pyramid.

2.3.4 Streaming MIP

Here we summarize the construction of the coefficient stream and the rendering for streaming MIP.

Construction of the detail coefficient stream. By the commutativity of the pyramidal synthesis operator with the maximum operation we can project the elements of the detail coefficients in any order on the image plane, not necessarily level by level, as we have done so far. So one can join the detail coefficients of all levels and sort these according to some error measure. We do not sort the detail coefficients $\{d_j\}$ directly. As can be seen from (2.9), the detail coefficients are not small in regions where the structuring element does fit approximately, but not exactly, to the data. Therefore an auxiliary set of detail coefficients can be defined which can be sorted with respect to decreasing magnitude. Then the original detail coefficients $\{d_j\}$ are resorted by giving them the same order as the resorted auxiliary coefficients, which are subsequently discarded; see [107] for details.

As a result of the construction phase, we have obtained an ordered list of detail coefficients $d_j(x, y, z)$, which can be used to compute the MIP of the input data. By construction, the order is such that each successive coefficient, when taken into account, maximally reduces the L_1-error between the partial reconstruction and the original data.

Rendering phase. In the MIP rendering phase, all sorted coefficients $\{d_j(x, y, z)\}$ are projected successively, until an a-priori chosen maximum number of coefficients has been projected, or a desired accuracy of the resulting MIP image is obtained. When a coefficient k is projected, its value val is compared to the current value $curval$ at the point of projection in the image plane, and $curval$ is overwritten when $curval < val$. Also, when the level of the coefficient is j, a local dilation of size j has to be carried out, i.e., all pixels in the scaled neighborhood $j \cdot \tilde{A}$ around the

point of projection are overwritten by val when their current value is smaller than val. Note that we cannot do the dilation globally, in contrast to the case of the level-by-level projection where the scale index is constant per level.

2.4 Implementation on graphics hardware

As discussed in section 2.3.2, the morphological pyramid of a dataset is built in a preprocessing step. For the MMIP projection which works level by level, each level is rendered separately using a MIP volume rendering method. Intermediate levels are rendered to a texture, starting from the coarsest level, after which the 2-D synthesis operator $\psi^{\downarrow}_{\tilde{A}}$ is applied and the result combined with the previous approximation, successively taking the detail signals into account. This two-dimensional synthesis operator is implemented as a fragment program which maps a $N \times M$ texture to a $2N \times 2M$ texture. The upsampling and dilation steps are rolled into one pass for efficiency. The levels are combined using the frame-buffer maximum operation. In what follows we discuss the separate steps in detail.

2.4.1 Per-voxel projection

To avoid processing empty space we use an object-order voxel-projection method [86], where one loops through the volume and projects all non-zero voxels in value-sorted order with each voxel contributing to exactly one pixel. By projecting the voxels from low to high value, old values in the image plane can simply be overwritten by current values. This allows for MIP rendering without expensive read-compare-write GPU cycles. This method also uses an efficient scheme for storing the volume data, based on histogram-based sorting of non-zero voxels according to their grey value. After the sorting step the voxels are represented by an array of positions. An additional array contains the cumulative histogram values. All levels of the pyramid are created and stored as value-sorted arrays.

2.4.2 Representing the detail coefficients

The voxel data are stored in a buffer that is created with the `ARB_vertex_buffer_object` extension. This extension defines an interface that allows data to be cached in high-performance graphics memory tailored to the use of these buffers, thereby increasing the rate of data transfers. A static buffer is requested that will be filled once by the application, and used many times as the source for GL drawing commands. The voxels of the volume are stored in a continuous region sorted per intensity value. The second structure, the histogram, is kept with the begin and end offsets in this buffer, for each intensity value. Voxels with one intensity level can then simply be sent to the vertex processor by invoking `glDrawArrays` once, with the begin and end values as found in this histogram.

The most naive implementation stores the intensity, x, y and z coordinate in shorts, and uses 8 bytes per voxel. The different attributes are provided to the vertex program as texture coordinates. Rows of consecutive voxels will have the same intensity value, as represented in the

histogram. Storing this with each voxel is very redundant, as the histogram acts as a kind of run-length-encoding. A shader constant or texture coordinate can be set to the intensity for each span of voxels with equal intensity. Now we are left with 6 bytes per voxel (2 bytes per coordinate). Theoretically this will allow for volumes up to 65536^3. In practice we cannot support such large volumes as they will not fit into memory on current hardware.

Let us assume that we want to reduce the memory requirements to 4 bytes per voxel (two `GL_SHORT`s). This means we have 32 bits available. Distributing this over X, Y and Z like $12+10+10$ ($4096 \times 1024 \times 1024$) will suffice for even huge volumes. We can unpack these in a vertex program. For GPUs that do not have bit shift and logical operators, these can be emulated with multiplication by a fraction (computed using the *floor* Cg function) and subtraction. Overall, this results in a major speedup for large volumes. Even though this unpacking requires some extra computation this is easily out-weighted by the savings in memory usage and the associated gain in the ratio of memory bandwidth to voxel count.

2.4.3 Projecting the detail coefficients

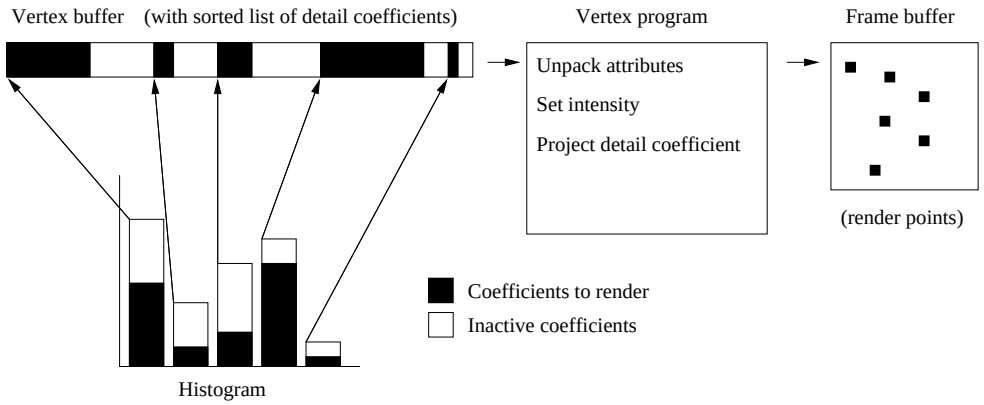


Figure 2.1. *The process of selection, projection and rendering of detail coefficients for one level of the pyramid. The white histogram is the histogram of the dataset, the (overlaid) black histogram shows the subset of the detail coefficients that is selected according to some the criterion.*

For the sake of clarity, but without sacrificing generality, we have used orthogonal projection throughout all examples. We build a model-view matrix from a quaternion which represents the orientation of the volume and a zoom factor (which defaults to 1 voxel on 1 pixel), both of which can be interactively adjusted. The projection matrix and the model-view matrix are then combined into a 4×4 model-view-projection matrix and passed to the vertex shader.

The entire volume is projected by iterating over the intensity values present in the volume, starting from the lowest (or from a user defined threshold below which everything is background) and stopping at the highest. If there is an upper intensity threshold above which everything has the maximum intensity, all these voxels can be projected with one call to `glDrawArrays`,

which is set to draw point primitives (without rounding or anti-aliasing). This is illustrated in Figure 2.1.

A multi-level representation is incorporated into the algorithm by storing the voxels for the different levels separately. One needs to store $L+1$ volumes for an L-level decomposition: detail levels $d_0 \dots d_{L-1}$ and the approximation f_L. These volumes are rendered in the same fashion as described already. Level 0 is rendered to a quad of the same size as the viewport, level 1 is rendered to a half-sized quad, level 2 to a quarter-sized one, and so on.

2.4.4 Load balancing

The MMIP algorithm has the advantage that we can spread the load and the dataset over multiple graphics cards in a straightforward way. In this way large volumes can be supported, and an almost optimal speedup is obtained. Fortunately, the newer NVidia drivers support multiple cards for one X-server for multiple screens. To render to both simultaneously, a `GL Context` is created on both X-screens, and rendering is switched between them one time per frame. Finally, the results of both cards are combined. Because the result of the second card is combined with that of the first, the second card does not necessarily have to be connected to a monitor at all. At the moment there is no extension for directly shuffling data between two cards, so the intermediate result will have to pass through the CPU. We found that using `glReadPixels` on one context and then `glDrawPixels` on the other one was the only currently available way of doing this. Our timings show that the preferred pixel format for this is `GL_UNSIGNED_INT_8_8_8_8`.

To utilize multiple cores or multiple processors, it was also attempted to put the rendering loop for each context in a different thread. This did not result in any performance gain. This is most probably because the algorithm is GPU bound, and the only processing done by the CPU is queuing commands and data to the GPU. Spreading this tiny load over multiple CPUs also does not outweigh the synchronization overhead.

The best performance was achieved by splitting the sorted array of voxels equally over both cards by interleaving the detail coefficients (also splitting the histogram) for each pyramid level. The coefficients are interleaved instead of being split in the middle to better divide the load in case of progressive refinement. In the end, the resulting images are combined using a pixel-wise maximum operator.

2.4.5 Streaming MIP

Streaming MIP can be implemented similarly using point rendering. For each voxel, we will have to store an extra attribute, namely the originating pyramid level. The level can be rolled into two bits of the position attribute (see section 2.4.2), which spans 4 bytes, and for intensity another short is required, making a total of 6 bytes per voxel.

To implement the level-dependent local dilation (see section 2.3.4) the size of the output voxel can be set using the point size extension to the ARB vertex program assembly language (which maps to the `PSIZE` output semantic in Cg). For example, assuming a square structuring element of size 2×2, the point size will be 1, 2 and 4 at level 0, 1 and 2, respectively. The specific shape of this point for each level depends on the structuring element A and the projection angle.

If the projection of A is not square, the *point sprite* extension is used to apply a specific shape to the point.

In addition to the increased memory usage, the fact that voxels are no longer ordered by intensity means that the writes to the frame buffer now have to be done with the maximum operator enabled. This results in some loss of performance, but not a huge one as the blended primitives are small. However, we will show next that this can be done in an even more efficient way.

2.4.6 Optimized streaming MIP

Note that the order in which the detail coefficients are rendered is not important, even though this method sorts them in a specific order. Therefore, the error-sorted list of details is not used to set the rendering order but only to guide setting the error for speed/quality tradeoff. The sorted array of detail coefficients, resulting from the preprocessing as required for streaming MIP, is subjected to a second histogram-sorting step with one bin per (level, intensity) pair. The ordering by error is maintained within these bins, having the bins store the position attribute of each detail coefficient. This results in a more efficient method for streaming MIP that avoids using point sprites and frame buffer blending, and which uses only four bytes per voxel.

The level and intensity attributes of the detail coefficients are kept in their original sorted order in one big list. The resulting histogram and sorted list can be used for rendering the entire dataset (or a per-level approximation) and also for streaming MIP rendering. This is implemented as follows. A percentage of detail coefficients to be rendered is chosen, after which the corresponding part of the sorted list is traversed. For each pyramid level the voxels are then rendered in a low-to-high intensity order, guided by the complete histogram and the number of coefficients that need to be rendered. This is the same algorithm as in section 2.4.1, except that an additional histogram is used to select which coefficients to render. With this new algorithm we have all the advantages of streaming MIP without any of the drawbacks mentioned in section 2.4.5.

2.4.7 Post-processing

For non axis-aligned parallel projections the voxel-based method can yield pixel-sized holes in the result. A post-processing step based on a morphological closing step with the structuring element $\tilde{A}$ used for synthesis can fill these holes effectively and efficiently.

2.5 Results

In this section we report some experimental results. All performance measurements were carried out on a machine with dual AMD Opteron 280 processor and two GeForce 7900GTX graphics cards. Unless mentioned otherwise, only one of the cards (and one of the CPUs) is active.

Table 2.1 shows the percentage of detail coefficients rendered for the Visible Woman dataset (dimensions $512 \times 512 \times 1734$, and a total of 210 million detail coefficients) versus three different

Table 2.1. *Approximation error as a function of the percentage of detail coefficients kept to render the VisibleWoman dataset (dimensions $512 \times 512 \times 1734$ and a total of 210 million detail coefficients). Three different error measures are shown (maximum, relative L_1 and median), and the performance is given in frames per second (FPS).*

Coeffs.	**Error**			**FPS**
(%)	max	relative L_1	$median$	
100	0	0	0	1.14
50	3	$9.6 \times 10^{-}5$	1	2.27
25	8	$2.0 \times 10^{-}3$	3	4.41
13	13	$1.2 \times 10^{-}2$	4	8.40
6	21	$2.6 \times 10^{-}2$	4	15.4
1	63	$1.0 \times 10^{-}1$	7	51.2

Table 2.2. *Approximation error as a function of the percentage of detail coefficients kept to render the XMasTree dataset (dimensions $512 \times 512 \times 512$ and a total of 105 million detail coefficients). Three different error measures are shown (maximum, relative L_1 and the median), and the performance is given in frames per second (FPS).*

Coeffs.	**Error**			**FPS**
(%)	max	relative L_1	$median$	
100	0	0	0	2.80
50	5	$1.3 \times 10^{-}4$	3	5.60
25	10	$5.8 \times 10^{-}4$	4	10.6
13	22	$1.8 \times 10^{-}3$	4	19.6
6	33	$7.2 \times 10^{-}3$	7	33.6
1	177	$7.0 \times 10^{-}2$	25	100.6

error measures, and the performance in frames per second. Table 2.2 shows the same results for the XMasTree dataset ($512 \times 512 \times 512$, and a total of 105 million detail coefficients).

In both tables, the *maximum* error measure (L_∞ norm) is the maximum difference in grey level between the original full quality MIP image and the approximation. However, this measure is not very representative for the perceived error. The *median* error measure is calculated by taking the median of the grey level differences (excluding the zero ones). The median error is expressed in grey levels and provides a good indication of how much the approximation differs from the original, not being very sensitive to outliers and noise. The relative L_1-approximation

Table 2.3. *Performance in frames per second (FPS) for the Visible Woman dataset in a* 512×512 *viewport for various MIP methods: (i) the most common texture-based volume rendering method; (ii) GPU ray-casting; (iii) MMIP, rendering everything but the highest detail level (iiii) streaming MIP, for an optimized software implementation, the GPU implementation (for two error settings), and for the workload distributed over two GPUs.*

Method	**Error** (L_1)	**FPS**
3-D Texture-based	-	1.0
GPU ray-casting [66]	-	2.0
MMIP, 2 levels (GPU)	0.16 (median 8)	8.0
Streaming MIP (software)	0.0	0.2
Streaming MIP (software)	0.07 (median 4)	3.5
Streaming MIP (GPU)	0.0	1.1
Streaming MIP (GPU)	0.07 (median 4)	18.4
Streaming MIP (2x GPU)	0.0	2.0
Streaming MIP (2x GPU)	0.07 (median 4)	30.2

error measure between the original image $\mathbf{M}^{(0)}$ and approximation $\mathbf{M}^{(j)}$ is defined as:

$$\varepsilon^{(j)} = \left\|\mathbf{M}^{(0)} - \mathbf{M}^{(j)}\right\| / \left\|\mathbf{M}^{(0)}\right\|,$$

where $\|\cdot\|$ represents the L_1 norm.

Visually, the error remains quite unnoticeable until the median error reaches a value around 5 to 7 (on the datasets which we experimented with), after which it generally starts to rise and artifacts appear. For interactive purposes a large reduction percentage (1-5 %) of the detail coefficients is acceptable, with a corresponding increase in performance. Also, it can be seen that the frame rate approximately doubles each time the number of detail coefficients is halved, so the relation between rendering time and amount of retained detail coefficients is linear.

Table 2.3 shows a comparison of various methods on the VisibleWoman dataset: (i) a brute force 3D texture-based method using view-aligned slices and frame buffer blending; (ii) GPU ray-casting [66]; and (iii) our streaming MIP implementations, both in software and on one or two GPUs, for various error levels. The render viewport was set to 512×512 in all cases. For both texture-based methods the dataset was split into four, three blocks of $512 \times 512 \times 512$ and one of $512 \times 512 \times 198$, because the hardware does not support 3D textures larger than 512^3.

When full reconstruction is performed, the streaming MIP in software achieves approximately the same performance as the 3D texture-based method. GPU ray-casting outperforms that method by making smart use of fragment programs and early ray termination. Streaming MIP starts to become interesting when allowing for a certain error. For example, when we admit a hardly visible error (L_1 error bound of 0.07, or median of 4 grey levels) we can achieve an interactive frame rate of 18 frames per second. Compared to the optimized software implementation, the GPU version is about six times faster given the same dataset and error bound. Using

two cards we achieve a speed up of almost a factor of two. In preview mode, a larger error may be acceptable (say median 7), which increases the performance to 50 FPS.

From this it can be concluded that our method works especially well on large datasets that do not fit into the memory of the graphics card(s) at once, and take too long to render using standard GPU ray-casting at the original resolution. With our method, such volumes can be rendered in real time, albeit a tradeoff between rendering quality and performance/resource usage has to be made.

Some MIP images obtained by our optimized method are shown in Figures 2.2 and 2.3. The first figure displays the MIP rendering of the XMasTree data set, for two different error levels. In the lower quality rendering (the left image) some degradation is visible in the background and at the base of the tree, but there is only a small difference in the tree itself. Figure 2.3 shows a rendering of the entire visible woman dataset at four error levels. The images on the first row show hardly any visible differences. However, the zoomed-in excerpts reveal some differences for strongly reduced coefficient percentages: at 5% the fine details of the ribs are still visible, whereas at 1% the image is visibly somewhat degraded.

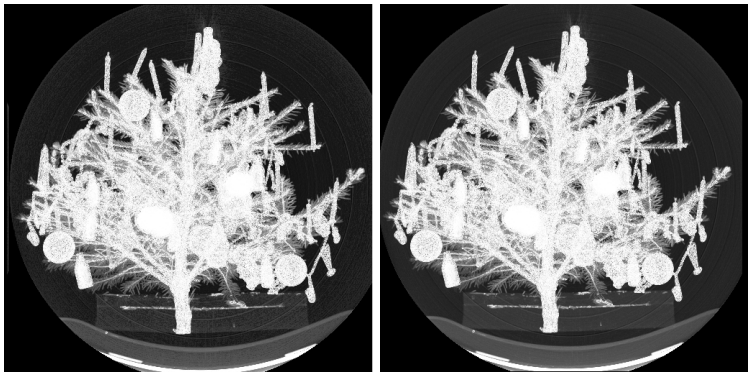


Figure 2.2. *MIP rendering of the XMasTree data set with 5% of the detail coefficients (left image) and at full reconstruction (right image).*

2.6 Discussion

A number of issues arise which require some further comments.

The sorting step during pyramid construction can take some time depending on the number of non-zero voxels, which is up to five minutes for the VisibleWoman dataset. Normally this will not be a problem, but if there is a hard time constraint in processing incoming data the method is unsuitable. A GPU sorting method can be used to accelerate this step. The most efficient method we are aware of was introduced by Gress and Zachmann [40], and is based on adaptive bitonic sorting (complexity $O(n \log n)$). For sorting n values utilizing p stream processor units, this approach has the optimal time complexity $O((n \log n)/p)$. On recent GPUs, this approach has shown to be remarkably faster than sequential sorting on the CPU.

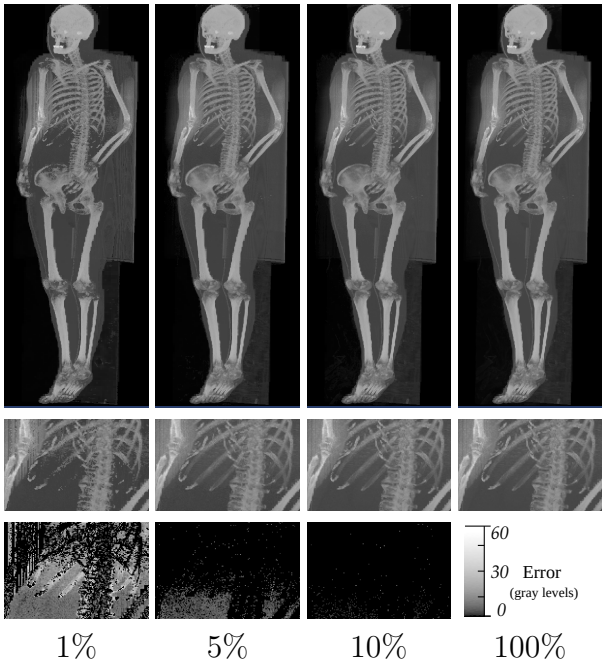


Figure 2.3. *MIP rendering of the complete Visible Woman dataset in a* 800 *by* 2000 *window (using streaming MIP Projection). The rendering is shown at various quality settings (given as percentages of the total number of detail coefficients). The second row shows a detail image for each quality setting, and the third row shows the difference image in gray levels.*

Current graphics hardware does not have enough memory to accommodate very large datasets. For this method, this is less of a problem as one can decide to only store the most important detail coefficients. With the upcoming generation of Shader version 4 GPUs, virtualisation of the memory is possible such that GPU memory is mapped on the host system. This will lessen the memory problem, but the increased access time of external memory can still make it worthwhile to use an streaming method like this.

Even though point rendering on the current generation of GPU hardware is faster than the equivalent on the CPU, the algorithm is bound by the vertex units of the GPU, which are still slower and less in number than the fragment units. This problem will disappear with the upcoming hardware which has unified shader units. This will give a large improvement to rendering speeds as the graphics hardware can allocate all of the shader units to process points.

Throughout this chapter a three-level pyramid (two detail levels and one approximation level) was used. It is also possible to use pyramids with more levels but we found no improvement in image quality or rendering speed with the data sets that were used. Performance even decreased a bit because more render, upsample, dilation steps are needed. The reason for no gain appears to be that structuring elements become larger than any structure present in the data, so that the higher levels have little non-zero coefficients. For even larger datasets the results might improve with more levels.

No attempt was made to remove voxels that do not contribute to the image from any viewing

angle. According to [86], a view-independent hidden voxel removal step is able to remove about half of the voxels in a dataset, at the cost of more preprocessing. Assuming this, a speedup in rendering by another factor of two could be achieved.

2.7 Conclusion

We have investigated a number of algorithms based on morphological pyramids for multiresolution MIP volume rendering on graphics hardware. We found that our highly-optimized streaming MIP GPU-method outperforms both its software implementation as well as existing ray-casting and 3-D texture-based methods.

Using the basic texture-based MIP method for each level, then synthesizing the result is the most obvious way of implementing multilevel volume rendering. But it is, by definition, not faster than plain volume rendering, as it does not take much advantage of storing only necessary voxels and near-continuous error control.

Per-voxel projection, as discussed in this chapter, is more advantageous. It applies point-based rendering to project the dataset a voxel at a time, and implicitly uses empty space skipping by only rendering non-empty voxels. The user can interactively adjust thresholds and set the performance/quality ratio as desired. The algorithm is also cache efficient, as it always addresses GPU memory in a linear way. In addition, the load and the dataset can be divided over multiple GPUs to achieve a near-optimal speedup, even for large volume data.

Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink. Accelerating Wavelet Lifting on Graphics Hardware using CUDA. *IEEE Transactions on Parallel and Distributed Systems*, **22**, 2011, pp. 132-146.

Chapter 3

Accelerating Wavelet Lifting on Graphics Hardware using CUDA

3.1 Introduction

The wavelet transform, originally developed as a tool for the analysis of seismic data, has been applied in areas as diverse as signal processing, video and image coding, compression, data mining and seismic analysis. The theory of wavelets bears a large similarity to Fourier analysis, where a signal is approximated by superposition of sinusoidal functions. A problem, however, is that the sinusoids have an infinite support, which makes Fourier analysis less suitable to approximate sharp transitions in the function or signal. Wavelet analysis overcomes this problem by using small waves, called *wavelets*, which have a compact support. One starts with a wavelet prototype function, called a *basic wavelet* or *mother wavelet*. Then a wavelet basis is constructed by translated and dilated (i.e., rescaled) versions of the basic wavelet. The fundamental idea is to decompose a signal into components with respect to this wavelet basis, and to reconstruct the original signal as a superposition of wavelet basis functions; therefore we speak a *multiresolution analysis*. If the shape of the wavelets resembles that of the data, the wavelet analysis results in a sparse representation of the signal, making wavelets an interesting tool for data compression. This also allows a client-server model of data exchange, where data is first decomposed into different levels of resolution on the server, then progressively transmitted to the client, where the data can be *incrementally* restored as it arrives ('progressive refinement'). This is especially useful when the data sets are very large, as in the case of 3D data visualization [147]. For some general background on wavelets, the reader is referred to the books by Daubechies [28] or Mallat [82].

In the theory of wavelet analysis both continuous and discrete wavelet transforms are defined. If discrete and finite data are used it is appropriate to consider the *Discrete Wavelet Transform* (DWT). Like the discrete Fourier transform (DFT), the DWT is a linear and invertible transform that operates on a data vector whose length is (usually) an integer power of two. The elements of the transformed vector are called *wavelet coefficients*, in analogy of Fourier coefficients in case of the DFT. The DWT and its inverse can be computed by an efficient filter bank algorithm, called Mallat's pyramid algorithm [82]. This algorithm involves repeated downsampling (forward transform) or upsampling (inverse transform) and convolution filtering by the application

of high and low pass filters. Its complexity is linear in the number of data elements.

In the construction of so-called *first generation* wavelet bases, which are translates and dilates of a single basic function, Fourier transform techniques played a major role [28]. To deal with situations where the Fourier transform is not applicable, such as wavelets on curves or surfaces, or wavelets for irregularly sampled data, *second generation* wavelets were proposed by Sweldens, based on the so-called *lifting scheme* [132]. This provides a flexible and efficient framework for building wavelets. It works entirely in the original time/space domain, and does not involve Fourier transforms.

The basic idea behind the lifting scheme is as follows. It starts with a simple wavelet, and then gradually builds a new wavelet, with improved properties, by adding new basis functions. So the simple wavelet is *lifted* to a new wavelet, and this can be done repeatedly. Alternatively, one can say that a complex wavelet transform is factored into a sequence of simple lifting steps [29]. More details on lifting are provided in section 3.3.

Also for first generation wavelets, constructing them by the lifting scheme has a number of advantages [132]. First, it results in a faster implementation of the wavelet transform than the straightforward convolution-based approach by reducing the number of arithmetic operations. Asymptotically for long filters, lifting is twice as fast as the standard algorithm. Second, given the forward transform, the inverse transform can be found in a trivial way. Third, no Fourier transforms are needed. Lastly, it allows a fully in-place calculation of the wavelet transform, so no auxiliary memory is needed. With the generally limited amount of high-speed memory available, and the large quantities of data that have to be processed in multimedia or visualization applications, this is a great advantage. Finally, the lifting scheme represents a universal discrete wavelet transform which involves only *integer* coefficients instead of the usual floating point coefficients [16]. Therefore we based our DWT implementation on the lifting scheme.

Custom hardware implementations of the DWT have been developed to meet the computational demands for systems that handle the enormous throughputs in, for example, real-time multimedia processing. However, cost and availability concerns, and the inherent inflexibility of this kind of solutions make it preferable to use a more widespread and general platform. NVidia's G80 architecture [70], introduced in 2006 with the GeForce 8800 GPU, provides such a platform. It is a highly parallel computing architecture available for systems ranging from laptops or desktop computers to high-end compute servers. In this chapter, we will present a hardware-accelerated DWT algorithm that makes use of the Compute Unified Device Architecture (CUDA) parallel programming model to fully exploit the new features offered by the G80 architecture when compared to traditional GPU programming.

The three main hardware architectures for the 2D DWT, i.e., row-column, line-based, or block-based, turn out to be unsuitable for a CUDA implementation (see Section 3.2). The biggest challenge of fitting wavelet lifting in the SIMD model is that data sharing is, in principle, needed after every lifting step. This makes the division into independent computational blocks difficult, and means that a compromise has to be made between minimizing the amount of data shared with neighbouring blocks (implying more synchronization overhead) and allowing larger data overlap in the computation at the borders (more computation overhead). This challenge is specifically difficult with CUDA, as blocks cannot exchange data at all without returning execution flow to the CPU. Our solution is a sliding window approach which enables us (in the case of separa-

ble wavelets) to keep intermediate results longer in shared memory, instead of being written to global memory. Our CUDA-specific design can be regarded as a hybrid method between the row-column and block-based methods. We implemented our methods both for 2D and 3D data, and obtained considerable speedups compared to an *optimized* CPU implementation and earlier non-CUDA based GPU DWT methods. Additionally, memory usage can be reduced significantly compared to previous GPU DWT methods. The method is scalable and the fastest GPU implementation among the methods considered. A performance analysis shows that the results of our CUDA-specific design are in close agreement with our theoretical complexity analysis.

The chapter is organized as follows. Section 3.2 gives a brief overview of GPU wavelet lifting methods, and previous work on GPU wavelet transforms. In Section 3.3 we present the basic theory of wavelet lifting. Section 3.4 first presents an overview of the CUDA programming environment and execution model, introduces some performance considerations for parallel CUDA programs, and gives the details of our wavelet lifting implementation on GPU hardware. Section 3.5 presents benchmark results and analyzes the performance of our method. Finally, in Section 3.6 we draw conclusions and discuss future avenues of research.

3.2 Previous and related work

In [52] a method was first proposed that makes use of OpenGL extensions on early non-programmable graphics hardware to perform the convolution and downsampling/upsampling for a 2-D DWT. Later, in [38] this was generalized to 3-D using a technique called tileboarding.

Wong *et al.* [157] implemented the DWT on programmable graphics hardware with the goal of speeding up JPEG2000 compression. They made the decision not to use wavelet lifting, based on the rationale that, although lifting requires less memory and less computations, it imposes an order of execution which is not fully parallelizable. They assumed that lifting would require more rendering passes, and therefore in the end be slower than the standard approach based on convolution.

However, Tenllado *et al.* [134] performed wavelet lifting on conventional graphics hardware by splitting the computation into four passes using fragment shaders. They concluded that a gain of 10-20% could be obtained by using lifting instead of the standard approach based on convolution. Similar to [157], Tenllado *et al.* [135] also found that the lifting scheme implemented using shaders requires more rendering steps, due to increased data dependencies. They showed that for shorter wavelets the convolution-based approach yields a speedup of 50-100% compared to lifting. However, for larger wavelets, on large images, the lifting scheme becomes 10-20% faster. A limitation of both [134] and [135] is that the methods are strictly focused on 2-D. It is uncertain whether, and if so, how they extend to three or more dimensions.

All previous methods are limited by the need to map the algorithms to graphics operations, constraining the kind of computations and memory accesses they could make use of. As we will show below, new advances in GPU programming allow us to do in-place transforms in a *single pass*, using intermediate fast shared memory.

Wavelet lifting on general parallel architectures was studied extensively in [60] for processor networks with large communications latencies. A technique called *boundary postprocessing*

was introduced that limits the amount of data sharing between processors working on individual blocks of data. This is similar to the technique we will use. More than in previous generations of graphics cards, general parallel programming paradigms can now be applied when designing GPU algorithms.

The three main hardware architectures for the 2D DWT are *row-column* (RC), *line-based* (LB) and *block-based* (BB), see for example [1, 4, 20, 158], and all three schemes are based on wavelet lifting. The simplest one is RC, which applies a separate 1D DWT in both the horizontal and vertical directions for a given number of lifting levels. Although this architecture provides the simplest control path (thus being the cheapest for a hardware realization), its major disadvantage is the lack of locality due to the use of large off-chip memory (i.e., the image memory), thus decreasing performance. Contrary to RC, both LB and BB involve a local memory that operates as a cache, thus increasing bandwidth utilization (throughput). On FPGA architectures, it was found [4] that the best instruction throughput is obtained by the LB method, followed by the RC and BB schemes which show comparable performances. As expected, both the LB and BB schemes have similar bandwidth requirements, which are at least two times smaller than that of RC. Theoretical results [20, 158] show that this holds as well for ASIC architectures. Thus, LB is the best choice with respect to overall performance for a hardware implementation.

Unfortunately, a CUDA realization of LB is impossible for all but the shortest wavelets (e.g., the Haar wavelet), due to the relatively large cache memory required. For example, the cache memory for the Deslauriers-Dubuc $(13, 7)$ wavelet should accommodate six rows of the original image (i.e., 22.5 KB for two-byte word data and HD resolutions), well in excess of the maximum amount of 16 KB of shared memory available per multi-processor, see Section 3.4.3. As an efficient implementation of BB requires similar amounts of cache memory, this choice is again not possible. Thus, the only feasible strategy remains RC. However, we show in Section 3.5 that even an improved (using cache memory) RC strategy is not optimal for a CUDA implementation. Nevertheless, our CUDA-specific design can be regarded as a hybrid method between RC and BB, which also has an optimal access pattern to the slow global memory (see Section 3.4.1).

3.3 Wavelet lifting

As explained in the introduction, lifting is a very flexible framework to construct wavelets with desired properties. When applied to first generation wavelets, lifting can be considered as a reorganization of the computations leading to increased speed and more efficient memory usage. In this section we explain in more detail how this process works. First we discuss the traditional wavelet transform computation by subband filtering and then outline the idea of wavelet lifting.

3.3.1 Wavelet transform by subband filtering

The main idea of (first generation) wavelet decomposition for finite 1-D signals is to start from a signal $c^0 = (c^0_0, c^0_1, \ldots, c^0_{N-1})$, with N samples (we assume that N is a power of 2). Then we apply convolution filtering of c^0 by a low pass analysis filter H and downsample the result by a factor of 2 to get an "approximation" signal (or "band") c^1 of length $N/2$, i.e., half the initial

length. Similarly, we apply convolution filtering of c^0 by a high pass analysis filter G, followed by downsampling, to get a detail signal (or "band") d_1. Then we continue with c^1 and repeat the same steps, to get further approximation and detail signals c^2 and d^2 of length $N/4$. This process is continued a number of times, say J. Here J is called the number of *levels* or *stages* of the decomposition. The explicit decomposition equations for the individual signal coefficients are:

$$c_k^{j+1} = \sum_n h_{n-2k}\, c_n^j,\ d_k^{j+1} = \sum_n g_{n-2k}\, c_n^j$$

where $\{h_n\}$ and $\{g_n\}$ are the coefficients of the filters H and G. Note that only the approximation bands are successively filtered, the detail bands are left "as is".

This process is presented graphically in Fig. 3.1, where the symbol $\downarrow_2$ (enclosed by a circle) indicates downsampling by a factor of 2. This means that after the decomposition the initial data

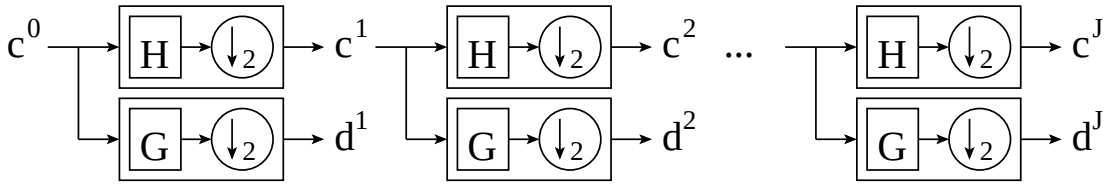


Figure 3.1. *Structure of the forward wavelet transform with J stages: recursively split a signal c^0 into approximation bands c^j and detail bands d^j.*

vector c^0 is represented by one approximation band c^J and J detail bands d^1, d^2, ..., d^J. The total length of these approximation and detail bands is equal to the length of the input signal c^0.

Signal reconstruction is performed by the *inverse* wavelet transform: first upsample the approximation and detail bands at the coarsest level J, then apply synthesis filters $\tilde{H}$ and $\tilde{G}$ to these, and add the resulting bands. (In the case of orthonormal filters, such as the Haar basis, the synthesis filters are essentially equal to the analysis filters.) Again this is done recursively. This process is presented graphically in Fig. 3.2, where the symbol $\uparrow_2$ indicates upsampling by a factor of 2.

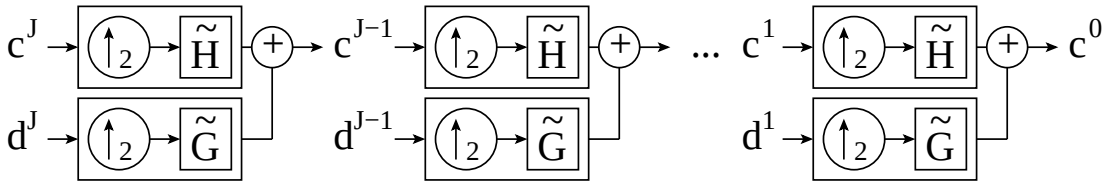


Figure 3.2. *Structure of the inverse wavelet transform with J stages: recursively upsample, filter and add approximation signals c^j and detail signals d^j.*

3.3.2 Wavelet transform by lifting

Lifting consists of four steps: split, predict, update, and scale, see Fig. 3.3 (left).

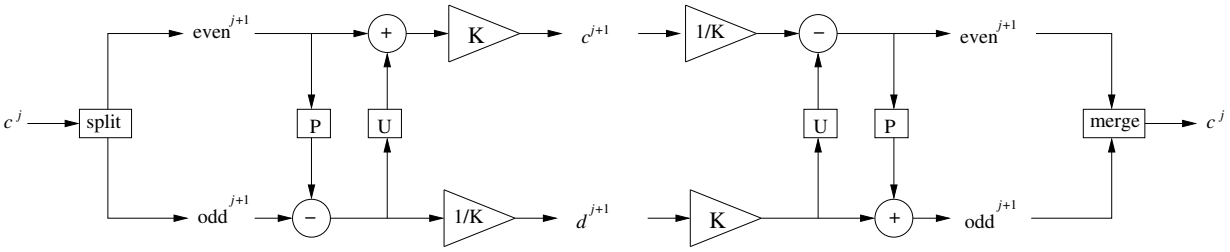


Figure 3.3. *Classical lifting scheme (one stage only). Left part: forward lifting. Right part: inverse lifting. Here "split" is the trivial wavelet transform, "merge" is the opposite operation, P is the prediction step, U the update step, and K the scaling factor.*

1. **Split**: this step splits a signal (of even length) into two sets of coefficients, those with even and those with odd index, indicated by even^{j+1} and odd^{j+1}. This is called the *lazy wavelet transform*.

2. **Predict lifting step**: as the even and odd coefficients are correlated, we can predict one from the other. More specifically, a prediction operator P is applied to the even coefficients and the result is subtracted from the odd coefficients to get the detail signal d^{j+1}:

$$d^{j+1} = \mathrm{odd}^{j+1} - P(\mathrm{even}^{j+1}) \tag{3.1}$$

3. **Update lifting step**: similarly, an update operator U is applied to the odd coefficients and added to the even coefficients to define c^{j+1}:

$$c^{j+1} = \mathrm{even}^{j+1} + U(d^{j+1}) \tag{3.2}$$

4. **Scale**: to ensure normalization, the approximation band c^{j+1} is scaled by a factor of K, and the detail band d^{j+1} by a factor of $1/K$.

Sometimes the scaling step is omitted; in that case we speak of an *unnormalized* transform.

A remarkable feature of the lifting technique is that the inverse transform can be found trivially. This is done by "inverting" the wiring diagram, see Fig. 3.3 (right): undo the scaling, undo the update step ($\mathrm{even}^{j+1} = c^{j+1} - U(d^{j+1})$), undo the predict step ($\mathrm{odd}^{j+1} = d^{j+1} + P(\mathrm{even}^{j+1})$), and merge the even and odd samples. Note that this scheme does not require the operators P and U to be invertible: nowhere does the inverse of P or U occur, only the roles of addition and subtraction are interchanged. For a multistage transform the process is repeatedly applied to the approximation bands, until a desired number of decomposition levels is reached. In the same way as discussed in section 3.3.1, the total length of the decomposition bands equals that of the initial signal. As an illustration, we give in Table 3.1 the explicit equations for one stage of the forward wavelet transform by the (unnormalized) Le Gall $(5, 3)$ filter, both by subband filtering and lifting (*in-place* computation). It is easily verified that both schemes give identical results for the computed approximation and detail coefficients.

The process above can be extended by including more predict and/or update steps in the wiring diagram [132]. In fact, any wavelet transform with finite filters can be decomposed into a sequence of lifting steps [29]. In practice, lifting steps are chosen to improve the decomposition, for example, by producing a lifted transform with better decorrelation properties or higher smoothness of the resulting wavelet basis functions.

Wavelet lifting has two properties which are very important for a GPU implementation. First, it allows a fully *in-place* calculation of the wavelet transform, so no auxiliary memory is needed. Second, the lifting scheme can be modified to a transform that maps *integers* to *integers* [16]. This is achieved by rounding the result of the P and U functions. This makes the predict and update operations nonlinear, but this does not affect the invertibility of the lifting transform. Integer-to-integer wavelet transforms are especially useful when the input data consists of integer samples. These schemes can avoid quantization, which is an attractive property for lossless data compression.

For many wavelets of interest, the coefficients of the predict and update steps (before truncation) are of the form $z/2^n$, with z integer and n a positive integer. In that case one can implement all lifting steps (apart from normalization) by integer operations: integer addition and multiplication, and integer division by powers of 2 (bit-shifting).

Table 3.1. *Forward wavelet transform (one stage only) by the (unnormalized) Le Gall* $(5,3)$ *filter.*

Subband filtering	$c_k^{j+1} = \frac{1}{8}\left(-c_{2k-2}^j + 2c_{2k-1}^j + 6c_{2k}^j + 2c_{2k+1}^j - c_{2k+2}^j\right)$
	$d_k^{j+1} = \frac{1}{2}(-c_{2k}^j + 2c_{2k+1}^j - c_{2k+2}^j)$
Lifting	Split: $c_k^{j+1} \leftarrow c_{2k}^j, \quad d_k^{j+1} \leftarrow c_{2k+1}^j$
	Predict: $d_k^{j+1} \leftarrow d_k^{j+1} - \frac{1}{2}(c_k^{j+1} + c_{k+1}^{j+1})$
	Update: $c_k^{j+1} \leftarrow c_k^{j+1} + \frac{1}{4}(d_{k-1}^{j+1} + d_k^{j+1})$

3.4 Wavelet lifting on GPUs using CUDA

3.4.1 CUDA overview

In recent years, GPUs have become increasingly powerful and more programmable. This combination has led to the use of the GPU as the main computation device for diverse applications, such as physics simulations, neural networks, image compression and even database sorting. The GPU has moved from being used solely for graphical tasks to a fully-fledged parallel co-processor. Until recently, General Purpose GPU (GPGPU) applications, even though not concerned with

graphics rendering, did use the rendering paradigm. In the most common scenario, textured quadrilaterals were rendered to a texture, with a fragment shader performing the computation for each fragment.

With their G80 series of graphics processors, NVidia introduced a programming environment called CUDA [70]. It is an API that allows the GPU to be programmed through more traditional means: a C-like language (with some C++-features such as templates) and compiler. The GPU programs, now called *kernels* instead of shaders, are invoked through procedure calls instead of rendering commands. This allows the programmer to focus on the main program structure, instead of details like color clamping, vertex coordinates and pixel offsets.

In addition to this generalization, CUDA also adds some features that are missing in shader languages: random access to memory, fast integer arithmetic, bitwise operations, and shared memory. The usage of CUDA does not add any overhead, as it is a native interface to the hardware, and not an abstraction layer.

Execution model

The CUDA execution model is quite different from that of CPUs, and also different from that of older GPUs. CUDA broadly follows the data-parallel model of computation [70]. The CPU invokes the GPU by calling a *kernel*, which is a special C-function.

The lowest level of parallelism is formed by *threads*. A thread is a single scalar execution unit, and a large number of threads can run in parallel. The thread can be compared to a fragment in traditional GPU programming. These threads are organized in *blocks*, and the threads of each block can cooperate efficiently by sharing data through fast shared memory. It is also possible to place synchronization points (barriers) to coordinate operations closely, as these will synchronize the control flow between all threads within a block. The Single Instruction Multiple Data (SIMD) aspect of CUDA is that the highest performance is realized if all threads within a *warp* of 32 consecutive threads take the same execution path. If flow control is used within such a *warp*, and the threads take different paths, they have to wait for each other. This is called *divergence*.

The highest level, which encompasses the entire kernel invocation, is called the *grid*. The grid consists of blocks that execute in parallel, if multiprocessors are available, or sequentially if this condition is not met. A limitation of CUDA is that blocks within a grid cannot communicate with each other, and this is unlikely to change as independent blocks are a means to scalability.

Memory layout

The CUDA architecture gives access to several kinds of memory, each tuned for a specific purpose. The largest chunk of memory consists of the *global* memory, also known as device memory. This memory is linearly addressable, and can be read and written at any position in any order (random access) from the device. No caching is done in G80, however there is limited caching in the newest generation (GT200) as part of the shared memory can be configured as automatic cache. This means that optimizing access patterns is up to the programmer. Global memory is also used for communication with the CPU, which can read and write using API calls. *Registers* are limited per-thread memory locations with very fast access, which are used for local storage.

Shared memory is a limited per-block chunk of memory which is used for communication between threads in a block. Variables are marked to be in shared memory using a specifier. Shared memory can be almost as fast as registers, provided that bank conflicts are avoided. *Texture memory* is a special case of device memory which is cached for locality. Textures in CUDA work the same as in traditional rendering, and support several addressing modes and filtering methods. *Constant memory* is cached memory that can be written by the CPU and read by the GPU. Once a constant is in the constant cache, subsequent reads are as fast as register access.

The device is capable of reading 32-bit, 64-bit, or 128-bit words from global memory into registers in a single instruction. When access to device memory is properly distributed over threads, it is compiled into 128-bit load instructions instead of 32-bit load instructions. The consecutive memory locations must be simultaneously accessed by the threads. This is called *memory access coalescing* [70], and it represents one of the most important optimizations in CUDA. We will confirm the huge difference in memory throughput between coalesced and non-coalesced access in our results.

3.4.2 Performance considerations for parallel CUDA programs (kernels)

Let us first define some metrics which we use later to analyze our results in Section 3.5.3 below.

Total execution time

Assume that a CUDA kernel performs computations on N data values, and organizes the CUDA 'execution model' as follows. Let T denote the number of threads in a block, W the number of threads in a warp, i.e., $W = 32$ for G80 GPUs, and B denote the number of thread blocks. Further, assume that the number of multiprocessors (device specific) is M, and that NVidia's occupancy calculator [90] indicates that k blocks can be assigned to one multiprocessor (MP); k is program specific and represents the total number of threads for which (re)scheduling costs are zero, i.e., context switching is done with no extra overhead. Given that the amount of resources per MP is fixed (and small), k simply indicates the occupancy of the resources for the given kernel. With this notation, the number of blocks assigned to one MP is given by $b = B/M$. Since in general k is smaller than b, it follows that the number α of times k blocks are rescheduled is $\alpha = \lceil \frac{B}{Mk} \rceil$.

Since each MP has 8 stream processors, a warp has 32 threads and there is no overhead when switching among the warp threads, it follows that each warp thread can execute one (arithmetic) instruction in four clock cycles. Thus, an estimate of the asymptotic time required by a CUDA kernel to execute n instructions over all available resources of a GPU, which also includes scheduling overhead, is given by

$$T_e = \frac{4\,n}{K}\,\frac{T}{W}\,\alpha\,k\,l_s, \tag{3.3}$$

where K is the clock frequency and l_s is the latency introduced by the scheduler of each MP.

The second component of the total execution time is given by the time T_m required to transfer N bytes from global memory to fast registers and shared memory. If thread transfers of m bytes

can be coalesced, given that a memory transaction is done per half-warp, it follows that the transfer time T_m is

$$T_m = \frac{2\,N}{W\,m\,M}\,l_m, \tag{3.4}$$

where l_m is the latency (in clock cycles) of a memory access. As indicated by NVidia [92], reported by others [145] and confirmed by us, the latency of a *non-cached* access can be as large as $400 - 600$ clock cycles. Compared to 24 cycle latency for accessing the shared memory, it means that transfers from global memory should be minimized. Note that for *cached* accesses the latency becomes about $250 - 350$ cycles.

One way to effectively address the relatively expensive memory-transfer operations is by using fine-grained *thread parallelism*. For instance, 24 cycle latency can be hidden by running 6 warps (192 threads) per MP. To hide even larger latencies, the number of threads should be raised (thus, increasing the degree of parallelism) up to a maximum of 768 threads per MP supported by the G80 architecture. However, increasing the number of threads while maintaining the size N of the problem fixed, implies that each thread has to perform less work. In doing so, one should still recall (i) the paramount importance of coalescing memory transactions and (ii) the Flops/word ratio, i.e., peak Gflop/s rate divided by global memory bandwidth in words [145], for a specific GPU. Thus, threads should not execute too few operations nor transfer too little data, such that memory transfers cannot be coalesced. To summarize, a tradeoff should be found between increased thread parallelism, suggesting more threads to hide memory-transfer latencies on the one hand, and on the other, memory coalescing and maintaining a specific Flops/word ratio, indicating fewer threads.

Let us assume that for a given kernel, one MP has an occupancy of $k\,T$ threads. Further, if the kernel has a ratio $r \in (0, 1)$ of arithmetic to arithmetic-and-memory-transfer instructions, and assuming a *round-robin* scheduling policy, then the *reduction* of memory-transfer latency due to latency hiding is

$$l_h = \sum_{i \geq 0} \left\lfloor \frac{k\,T}{8}\,r^i \right\rfloor. \tag{3.5}$$

For example, assume that $r = 0.5$, i.e., there are as many arithmetic instructions (flops) as memory transfers, and assume that $k\,T = 768$, i.e., each MP is fully occupied. The scheduler starts by assigning 8 threads to a MP. Since $r = 0.5$ chances are that 4 threads execute each a memory transfer instruction while the others execute one arithmetic operation. After one cycle, those 4 threads executing memory transfers are still asleep for at least 350 cycles, while the others just finished executing the flop and are put to sleep too. The scheduler assigns now another 8 threads, which again can execute either a memory transfer or a flop, with the same probability, and repeats the whole process. Counting the number of cycles in which 4 threads executed flops, reveals a number of 190 cycles, so that the latency is decreased in this way to just $l_m = 350 - 190 = 160$ cycles. In the general case, for a given r and occupancy, we postulate that formula (3.5) applies.

The remaining component of the total GPU time for a kernel is given by the synchronization time. To estimate this component, we proceed as follows. Assume all active threads (i.e., $k\,T \leq 768$) are about to execute a flop, after which they have to wait on a synchronization point (i.e.,

on a barrier). Then, assuming again a round-robin scheduling policy and reasoning similar as for Eq. (3.3), the idle time spent waiting on the barrier is

$$T_s = \frac{T_e}{n} = \frac{4}{K}\,\frac{T}{W}\,\alpha\,k\,l_s. \tag{3.6}$$

This agrees with NVidia's remark that, if within a warp thread divergence is minimal, then waiting on a synchronization barrier requires only four cycles [92]. Note that the expression for T_s from Eq. (3.6) represents the minimum synchronization time, as threads were assumed to execute (fast) flops. In the worst case scenario – at least one active thread has just started before the synchronization point, a slow global-memory transaction – this estimate has to be multiplied by a factor of about $500/4 = 125$ (the latency of a non-cached access divided by 4 threads).

To summarize, we estimate the total execution time T_t as $T_t = T_e + T_m + T_s$.

Instruction throughput

Assuming that a CUDA kernel performs n flops in a number c of cycles, then the *estimate* of the asymptotic Gflop/s rate is $G_e = \frac{32\,M\,n\,K}{c}$, whereas the *measured* Gflop/s rate is $G_m = \frac{n\,N}{T_t}$; here K is the clock rate and T_t the (measured) total execution time. For the 8800 GTX GPU the peak instruction throughput using register-to-register MAD instructions is about 338 Gflop/s and drops to 230 Gflop/s when using transfers in/from shared memory [145].

Memory bandwidth

Another factor which should be taken into account when developing CUDA kernels is the memory bandwidth, $M_b = \frac{N}{T_t}$. For example, parallel *reduction* has very low arithmetic intensity, i.e., 1 flop per loaded element, which makes it bandwidth-optimal. Thus, when implementing a parallel reduction in CUDA, one should strive for attaining peak bandwidth. On the contrary, if the problem at hand is matrix multiplication (a trivial parallel computation, with little synchronization overhead), one should optimize for peak throughput. For the 8800 GTX GPU the pin-bandwidth is 86 GB/s.

Complexity

With coalesced accesses the number of bytes retrieved with one memory request (and thus one latency) is maximized. In particular, coalescing reduces l_m (through l_h from Eq. 3.5) by a factor of about two. Hence one can safely assume that $l_m/(2\,W) \to 0$. It follows that the total execution time satisfies

$$T_t \sim 4\,n\,\frac{N}{W\,M\,D}, \tag{3.7}$$

where n is the number of instructions of a given CUDA kernel, N is the problem size, D is the problem size per thread, and $\sim$ means that both left and right-hand side quantities have the same order of magnitude.

The *efficiency* of a parallel algorithm is defined as

$$E = \frac{T_S}{C} = \frac{T_S}{M\,T_t}, \tag{3.8}$$

where T_S is the execution time of the (fastest) sequential algorithm, and $C = M\,T_t$ is the *cost* of the parallel algorithm. A parallel algorithm is called *cost efficient* (or cost optimal) if its cost is proportional to T_S. Let us assume $T_S \sim n_s\,N$, where n_s is the number of instructions for computing one data element and N denotes the problem size. Then, the efficiency becomes

$$E \sim \frac{n_s\,W\,D}{4\,n}. \tag{3.9}$$

Thus, according to our metric above, for a given problem, *any* CUDA kernel which (i) uses *coalesced* memory transfers (i.e., $l_m/(2\,W) \to 0$ is enforced), (ii) avoids thread divergence (so that our T_s estimate from Eq. 3.6 applies), (iii) minimizes transfers from global memory, and (iv) has an instruction count n proportional to $(n_s\,W\,D)$ is *cost efficient*. Of course, the smaller n is, the more efficient the kernel becomes.

3.4.3 Parallel wavelet lifting

Earlier parallel methods for wavelet lifting [60] assumed an MPI architecture with processors that have their own memory space. However, the CUDA architecture is different. Each processor has its own shared memory area of 16 KB, which is not enough to store a significant part of the dataset. As explained above, each processor is allocated a number of threads that run in parallel and can synchronize. The processors have no way to synchronize with each other, beyond their invocation by the host.

This means that data parallelism has to be used, and moreover, the dataset has to be split into parts that can be processed as independently as possible, so that each chunk of data can be allocated to a processor. For wavelet lifting, except for the Haar [132] transform, this task is not trivial, as the implied data re-use in lifting also requires the coefficients just outside the delimited block to be updated. This could be solved by duplicating part of the data in each processor. Wavelet bases with a large support will however need more data duplication. If we want to do a multilevel transform, each level of lifting doubles the amount of duplicated work and data. With the limited amount of shared memory available in CUDA, this is not a feasible solution.

As kernel invocations introduce some overhead each time, we should also try to do as much work within one kernel as possible, so that the occupancy of the GPU is maximized. The sliding window approach enables us (in the case of separable wavelets) to keep intermediate results longer in shared memory, instead of being written to global memory.

3.4.4 Separable wavelets

For separable wavelet bases in 2-D it is possible to split the operation into a horizontal and a vertical filtering step. For each filter level, a horizontal pass performs a 1-D transform on each

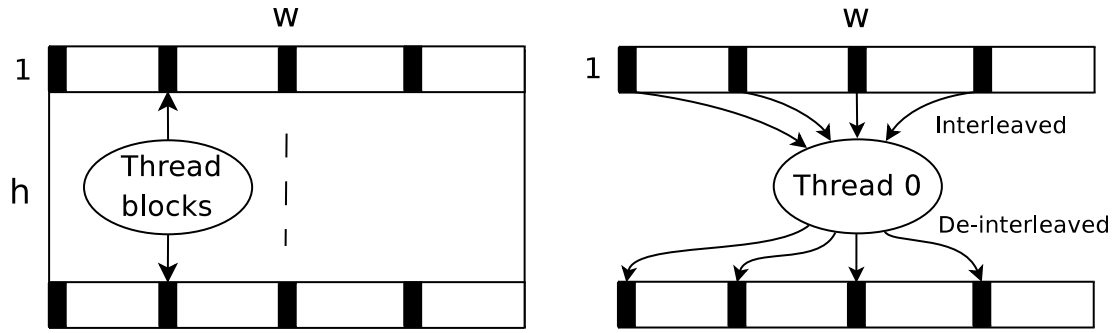


Figure 3.4. *Horizontal lifting step. h thread blocks are created, each containing T threads; each thread performs computations on $N/(hT) = w/T$ data. Black quads illustrate input for the thread with id 0. Here w and h are the the dimensions of the input and $N = w \cdot h$.*

row, while a vertical pass computes a 1-D transform on each column. This lends itself to easy parallelization: each row can be handled in parallel during the horizontal pass, and then each column can be handled in parallel during the vertical pass. In CUDA this implies the use of two kernels, one for each pass. The simple solution would be to have each block process a row with the horizontal kernel, while in the vertical step each block processes a column. Each thread within these blocks can then filter an element. We will discuss better, specific algorithms for both passes in the upcoming subsections.

3.4.5 Horizontal pass

The simple approach mentioned in the previous subsection works very well for the horizontal pass. Each block starts by reading a line into shared memory using so-called *coalesced reads* from device memory, executes the lifting steps in-place in fast shared memory, and writes back the result using *coalesced writes*. This amounts to the following steps:

1. Read a row from device memory into shared memory.
2. Duplicate border elements (implement boundary condition).
3. Do a 1-D lifting step on the elements in shared memory.
4. Repeat steps 2 and 3 for each lifting step of the transform.
5. Write back the row to device memory.

As each step is dependent on the output in shared memory of the previous step, the threads within the block have to be synchronized every time before the next step can start. This ensures that the previous step did finish and wrote back its work. Fig. 3.4 shows the configuration of the CUDA execution model for the horizontal step. Without loss of generality, assume that $N = w \cdot h$

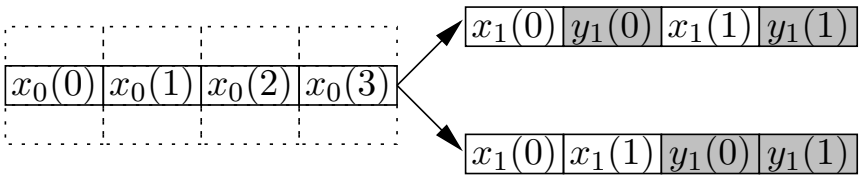


Figure 3.5. *Wavelet lifting for a row of data, representing the result in interleaved (top) and de-interleaved (bottom) form. Here x_i and y_i are the approximation and detail bands at level i.*

integers are lifted at level i. Note that, if the lifting level $i = 0$, then w and h are the dimensions of the input image. For this step, a number $B = h$ of thread blocks are used, with T threads per block. Thus, each thread performs computations on w/T integers. In the figure, black quads illustrate locations which are processed by the thread with id 0. Neither the number nor the positions of these quads need to correspond to the actual number and positions of locations where computations are performed, i.e., they are solely used for illustration purposes.

By reorganizing the coefficients [18] we can achieve higher efficiency for successive levels after the first transformation. If the approximation and detail coefficients are written back in interleaved form, as is usually the case with wavelet lifting, the reading step for the next level will have to read the approximation coefficients of the previous level in interleaved form. These reads cannot be coalesced, resulting in low memory performance. To still be able to coalesce, one writes the approximation and detail coefficients back to separate halves of the memory. This will result in a somewhat different memory layout for subbands (Fig. 3.5) but this could be reorganized if needed. Many compression algorithms require the coefficients stored per subband anyhow, in which case this organization is advantageous.

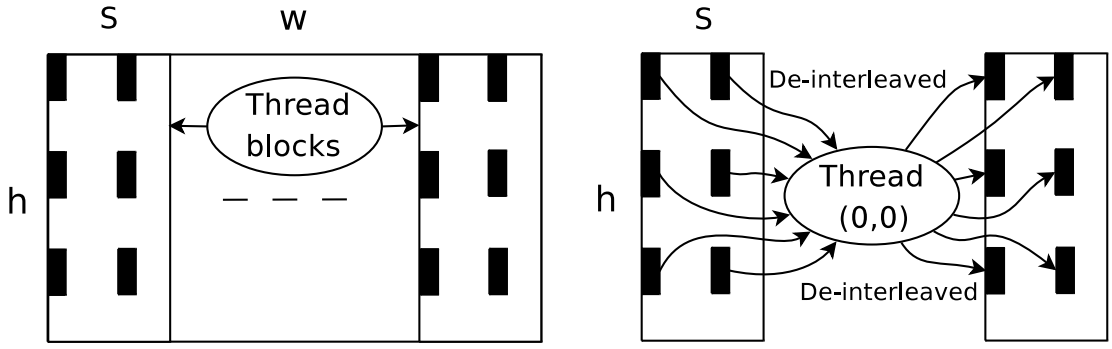


Figure 3.6. *Vertical lifting step. w/S blocks are created, each containing $T = V_x \cdot V_y$ threads; each thread performs computations on $S/V_x \times h/V_y$ data. Black quads illustrate input for the thread with id $(0, 0)$, whereas vertical lines depict boundaries between image-high slabs.*

3.4.6 Vertical pass

The vertical pass is more involved. Of course it is possible to use the same strategy as for the horizontal pass, substituting rows for columns. But this is far from efficient. Reading a column from the data would amount to reading one value per row. As only consecutive reads can be coalesced into one read, these are all performed individually. The processing steps would be the same as for the horizontal pass, after which writing back is again very inefficient.

We can gain a 10 times speedup by using coalesced memory access. Instead of having each block process a column, we make each block process multiple columns by dividing the image into vertical "slabs", see Fig. 3.6. Within a block, threads are organized into a 2D grid of size $V_x \times V_y$, instead of a 1D one, as in the horizontal step. The number S of columns in each slab is a multiple of V_x such that the resulting number of slab rows can still be coalesced, and has the height of the image. Each thread block processes one of the slabs, i.e., $S/V_x \times h/V_y$ data. Using this organization, a thread can do a coalesced read from each row within a slab, do filtering in shared memory, and do a coalesced write to each slab row.

Another problem arises here, namely that the shared memory in CUDA is not large enough to store all columns for any sizable dataset. This means that we cannot read and process the entire slab at once. The solution that we found is to use a sliding window within each slab, see Fig. 3.7(a). This window needs to have dimensions so that each thread in the block can transform a signal element, and additional space to make sure that the support of the wavelet does not exceed the top or bottom of the window. To determine the size of the window needed, how much to advance, and at which offset to start, we need to look at the support of each of the lifting steps.

In Fig. 3.7(a), `height` is the height of the working area. As each step updates either odd or even rows within a slab, each row of threads updates one row in each lifting step. Therefore, a good choice is to set it to two times the number of threads in the vertical direction. Similarly, `width` should be a multiple of the number of threads in the horizontal direction, and the size of a row should be a multiple of the coalescable size. In the figure, rows in the `top` area have been fully computed, while rows in the overlap area still need to go through at least one lifting step. The rows in the working area need to go through all lifting steps, whilst rows in the bottom area are untouched except as border extension. The sizes of `overlap`, `top` and `bottom` depend on the chosen wavelet. We will elaborate on this later.

The algorithm

Algorithm 3.1 shows the steps for the vertical lifting pass. Three basic operations are used: **read** copies rows from device memory into shared memory, **write** copies rows from shared memory back to device memory, and **copy** transfers rows from shared memory to another place in shared memory. The shared memory window is used as a cache, and to manage this we keep a read and a write pointer. The read pointer *inrow* indicates where to read from, the write pointer *outrow* indicates where to write back. After reading, we advance the read pointer, after writing we advance the write pointer. Both are initialized to the top of the slab at the beginning of the kernel (line 1 and 2 of Algorithm 3.1).

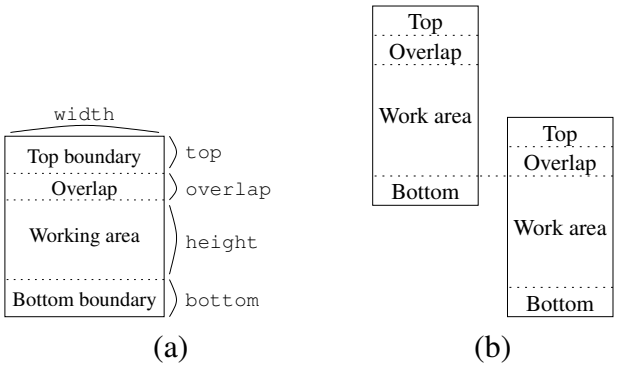


Figure 3.7. *(a): The sliding window used during the vertical pass for separable wavelets. (b): Advancing the sliding window: the next window is aligned at the bottom of the previous one, taking the overlap area into account.*

The first block has to be handled differently because we need to take the boundary conditions into account. So initially, rows are copied from the beginning of the slab to shared memory, filling it from a certain offset to the end (line 5). Next we apply a vertical wavelet lifting transform (**transformTop**, line 7) to the rows in shared memory (it may be required to leave some rows at the end untouched for some of the lifting steps, depending on their support; we will elaborate on this in the next section). After this we write back the fully transformed rows from shared memory to device memory (line 8). Then, for each block, the bottom part of the shared memory is copied to the top part (Fig. 3.7(b)), in order to align the next window at the bottom of the previous one, taking the overlap area into account (line 11). The rest of the shared memory is filled again by copying rows from the current read pointer of the slab (line 12).

Further, we apply a vertical wavelet lifting transform (**transformBlock**, line 14) to the rows in the working area. This does not need to take boundary conditions into account as the top and bottom are handled specifically with **transformTop** and **transformBottom**. Then, `height` rows are copied from shared memory row `top` to the current write pointer (line 15). This process is repeated until we have written back the entire slab, except for the last leftover part. When finishing up (line 20), we have to be careful to satisfy the bottom boundary condition.

Example

We will discuss the Deslauriers-Dubuc $(13, 7)$ wavelet as an example [31]. This example was chosen because it represents a non-trivial, but still compact enough case of the algorithm, that we can go through step by step. The filter weights for the two lifting steps of this transform are shown in Table 3.2. Both the prediction and update steps depend on two coefficients before and after the signal element to be computed. Fig. 3.8 shows an example of the performed computations. For this example, we choose $\texttt{top} = 3$, $\texttt{overlap} = 2$, $\texttt{height} = 8$ and $\texttt{bottom} = 3$. This is a toy example, as in practice `height` will be much larger when compared to the other parameters.

Starting with the first window at the start of the dataset, step 1 (first column), the odd rows

Algorithm 3.1 The sliding window algorithm for the vertical wavelet lifting transform (see section 3.4.6). Here `top`, `overlap`, `height`, `bottom` are the length parameters of the sliding window (see Fig. 3.7), and h is the number of rows of the dataset. The pointer *inrow* indicates where to read from, the pointer *outrow* indicates where to write back.

1: `inrow` $\leftarrow 0$ {*initialize read pointer*}
2: `outrow` $\leftarrow 0$ {*initialize write pointer*}
3: `windows` $\leftarrow (h -$ `height` $-$ `bottom`$)/$`height` {*number of times window fits in slab*}
4: `leftover` $\leftarrow (h -$ `height` $-$ `bottom`$)\%$`height` {*remainder*}
5: **read**(`height` $+$ `bottom` **from row** `inrow` **to row** `top` $+$ `overlap`) {*copy from global to shared memory*}
6: `inrow` $\leftarrow$ `inrow` $+$ `height` $+$ `bottom` {*advance read pointer*}
7: **transformTop**() {*apply vertical wavelet lifting to rows in shared memory*}
8: **write**(`height` $-$ `overlap` **from row** `top` $+$ `overlap` **to row** `outrow`) {*write transformed rows back to global memory*}
9: `outrow` $\leftarrow$ `outrow` $+$ `height` $-$ `overlap` {*advance write pointer*}
10: **for** $i = 1$ to `windows` **do** {*advance sliding window through slab and repeat above steps*}
11: **copy**(`top` $+$ `overlap` $+$ `bottom` **from row** `height` **to row** 0)
12: **read**(`height` **from row** `inrow` **to row** `top` $+$ `overlap` $+$ `bottom`)
13: `inrow` $\leftarrow$ `inrow` $+$ `height`
14: **transformBlock**() {*vertical wavelet lifting*}
15: **write**(`height` **from row** `top` **to row** `outrow`)
16: `outrow` $\leftarrow$ `outrow` $+$ `height`
17: **end for**
18: **copy**(`top` $+$ `overlap` $+$ `bottom` **from row** `height` **to row** 0)
19: **read**(`leftover` **from row** `inrow` **to row** `top` $+$ `overlap` $+$ `bottom`)
20: **transformBottom**() {*satisfy bottom boundary condition*}
21: **write**(`leftover` $+$ `overlap` $+$ `bottom` **from row** `top` **to row** `outrow`)

of the working area (offset $1, 3, 5, 7$) are lifted. The lifted rows are marked with a cross, and the rows they depend on are marked with a bullet. In step 2 (second column) the even rows are lifted. Again, the lifted rows are marked with a cross, and the dependencies are marked with a bullet. As the second step is dependent on the first, we cannot lift any rows that are dependent on values that were not yet calculated in the last step. In Fig. 3.8, this would be the case for row 6: this row requires data in rows 3, 5, 7 and 9, but row 9 is not yet available.

Here the `overlap` region of rows comes in. As row 6 of the window is not yet fully transformed, we cannot write it back to device memory yet. So we write everything up to this row back, copy the overlapping area to the top, and proceed with the second window. In the second window, we again start with step 1. The odd rows are lifted, except for the first one (offset 7) which was already computed, i.e., rows 9, 11, 13 and 15 are lifted. Then, in step 2 we start at row 6, i.e., three rows before the first step (row 9), but we do lift four rows.

After this we can write the top 8 rows back to device memory, and begin with the next window in exactly the same way. We repeat this until the entire dataset is transformed. By letting the second lifting step lag behind the first, one can do the same number of operations in each, making

Table 3.2. *Filter weights of the two lifting steps for the Deslauriers-Dubuc* $(13, 7)$ *[31] wavelet. The current element being updated is marked with* $\bullet$.

Offset	Prediction	Update
-3		$-\frac{1}{16}$
-2	$\frac{1}{16}$	
-1		$\frac{9}{16}$
0	$-\frac{9}{16}$	$\bullet$
1	$\bullet$	$\frac{9}{16}$
2	$-\frac{9}{16}$	
3		$-\frac{1}{16}$
4	$\frac{1}{16}$	

optimal use of the thread matrix (which should have a height of 4 in this case).

All separable wavelet lifting transforms, even those with more than two lifting steps, or with differently sized supports, can be computed in the same way. The transform can be inverted by running the steps in reverse order, and flipping the signs of the filter weights.

3.4.7 3-D and higher dimensions

The reason that the horizontal and vertical passes are asymmetric is because of the coalescing requirement for reads and writes. In the horizontal case, an entire line of the data-set could be processed at a time. In the vertical case the data-set was horizontally split into image-high slabs. This allowed the slabs to be treated independently and processed using a sliding window algorithm that uses coalesced reads and writes to access lines of the slab. A consecutive, horizontal span of values is stored at consecutive addresses in memory. This does not extend similarly to vertical spans of values, these will be separated by an offset at least the width of the image, known as the *row pitch*. As a slab is a rectangular region of the image of a certain width that spans the height of the image, it will be represented in memory by an array of consecutive spans of values, each separated by the *row pitch*.

When adding an extra dimension, let us say z, the volume is stored as an array of slices. In a span of values oriented along this dimension, each value is separated in memory by an offset that we call the *slice pitch*. By orienting the slabs in the xz-plane instead of the xy-plane, and thus using the *slice pitch* instead of the *row pitch* as offset between consecutive spans of values, the same algorithm as in the vertical case can be used to do a lifting transform along this dimension. To verify our claim, we implemented the method just described, and report results in section 3.5.2. More than three dimensions can be handled similarly, by orienting the slabs in the $D_i x$ plane (where D_i is dimension i) and using the pitch in that dimension instead of the row pitch.

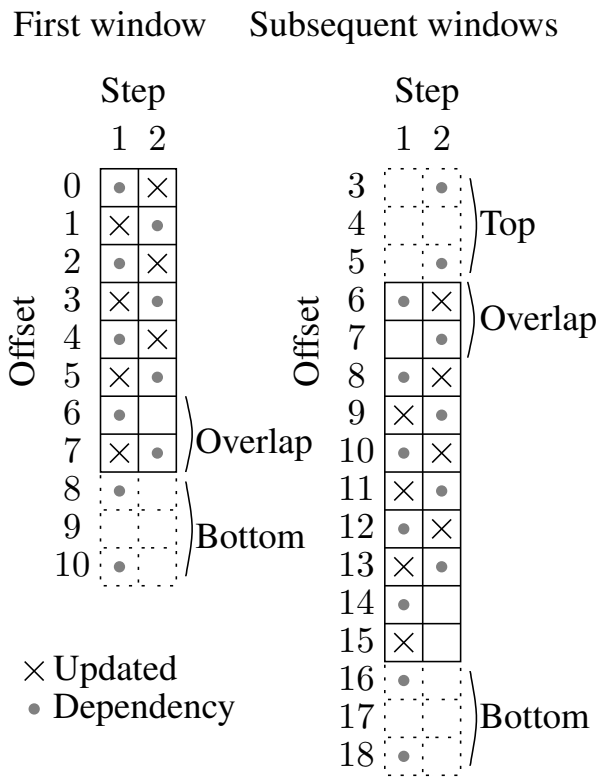


Figure 3.8. *The vertical pass for the Deslauriers-Dubuc* $(13, 7)$ *[31] wavelet. Lifted rows in each step are marked with a cross, dependent rows are marked with a bullet.*

3.5 Results

We first present a broad collection of experimental results. This is followed by a performance analysis which provides insight in the results obtained, and also shows that the design choices we made closely match our theoretical predictions.

The benchmarks in this section were run on a machine with a AMD Athlon 64 X2 Dual Core Processor 5200+ and a NVidia GeForce 8800 GTX 768MB graphics card, using CUDA version 2.1 for the CUDA programs. All reported timings exclude the time needed for reading and writing images or volumes from and to disc (both for the CPU and GPU versions).

3.5.1 Wavelet filters used for benchmarking

The wavelet filters that we used in our benchmarks are integer-to-integer versions (unnormalized) of the Haar [132], Deslauriers-Dubuc $(9, 7)$ [31], Deslauriers-Dubuc $(13, 7)$ [31], Le Gall $(5, 3)$ [69], (integer approximation of) Daubechies $(9, 7)$ [28] and the Fidelity wavelet – a custom wavelet with a large support [9]. In the filter naming convention (m, n), m refers to the length of the analysis low-pass and n to the analysis high-pass filters in the conventional wavelet subband filtering model, in which a convolution is applied before subsampling. They do not reflect the

length of the filters used in the lifting steps, which operate in the subsampled domain. The implementation only involves integer addition and multiplication, and integer division by powers of 2 (bit-shifting), cf. section 3.3.2. The coefficients of the lifting filters can be found in [9].

Table 3.3. *Performance of our CUDA GPU implementation of 2D wavelet lifting (column 5) compared to an optimized CPU implementation (column 2) and a CUDA GPU transpose method (column 3, see text), computing a three-level decomposition of a* 1920×1080 *image for both analysis and synthesis steps.*

Wavelet (analysis)	**CPU** (ms)	**GPU** transpose (ms)	**Speed-up**	**GPU** our method (ms)	**Speed-up**
Haar	10.31	5.58	1.9	0.80	12.9
Deslauriers-Dubuc $(9, 7)$	16.84	6.01	2.8	1.50	11.2
Le Gall $(5, 3)$	14.03	5.89	2.4	1.34	10.5
Deslauriers-Dubuc $(13, 7)$	19.52	6.08	3.2	1.62	12.0
Daubechies $(9, 7)$	22.66	6.54	3.5	2.05	11.1
Fidelity	28.82	6.45	4.5	2.11	13.7
Wavelet (synthesis)	**CPU** (ms)	**GPU** transpose (ms)	**Speed-up**	**GPU** our method (ms)	**Speed-up**
Haar	9.11	6.33	1.4	0.83	11.0
Deslauriers-Dubuc $(9, 7)$	15.93	6.40	2.5	1.45	11.0
Le Gall $(5, 3)$	13.02	6.29	2.1	1.28	10.2
Deslauriers-Dubuc $(13, 7)$	18.22	6.48	2.8	1.55	11.8
Daubechies $(9, 7)$	21.73	7.03	3.1	2.04	10.7
Fidelity	27.21	6.86	4.0	2.18	12.5

3.5.2 Experimental results and comparison to other methods

Comparison of 2D wavelet lifting, GPU versus CPU

First, we emphasize that the accuracies of the GPU and CPU implementations are the same. Because only integer operations are used (cf. section 3.5.1) the results are identical.

We compared the speed of performing various wavelet transforms using our optimized GPU implementation, to an optimized wavelet lifting implementation on the CPU, called Schrödinger [9]. The latter implementation makes use of vectorization using the MMX and SSE instruction set extensions, thus can be considered close to the maximum that can be achieved on the CPU with one core.

Table 3.3 shows the timings of both our GPU accelerated implementation and the Schrödinger implementation when computing a three-level transform with various wavelets of a 1920×1080 image consisting of 16-bit samples. As it is better from an optimization point of view to have a tailored kernel for each wavelet type, than to have a single kernel that handles everything, we used a code generation approach to create specific kernels for the horizontal and vertical pass for each of the wavelets. Both the analysis (forward) and synthesis (inverse) transform are benchmarked. We observe that speedups by a factor of 10 to 14 are reached, depending on the type of wavelet and the direction of the transform. The speedup factor appears to be roughly proportional to the length of the filters. The Haar wavelet is an exception, since the overlap problem does not arise in this case (the filter length being just 2), which explains the larger speedup factor.

To demonstrate the importance of coalesced memory access in CUDA, we also performed timings using a trivial CUDA implementation of the Haar wavelet, that uses the same algorithm for the vertical step as for the horizontal step, instead of our sliding window algorithm. Note that this method can be considered an improved (using cache) row-column, hardware-based strategy, see Section 3.2. Whilst our algorithm processes an image in 0.80 milliseconds, the trivial algorithm takes 15.23, which is almost 20 times slower. This is even slower than performing the transformation on the CPU.

Note that the timings in Table 3.3 do not include the time required to copy the data from (2.4 ms) or to (1.6 ms) the GPU.

Vertical step via transpose method

Another method that we have benchmarked consists in reusing the horizontal step as vertical step by using a "transpose" method. Here, the matrix of wavelet coefficients is transposed after the horizontal pass, the algorithm for the horizontal step is applied, and the results are transposed back. The results are shown in columns 3 and 4 of Table 3.3. Even though the transpose operation in CUDA is efficient and coalescable, and this approach is much easier to implement, the additional passes over the data reduce performance quite severely. Another drawback of this method is that transposition cannot be done in-place efficiently (in the general case), which doubles the required memory, so that the advantage of using the lifting strategy is lost.

Comparison of horizontal and vertical steps

Table 3.4 shows separate benchmarks for the horizontal and vertical steps, using various wavelet filters. From these results one can conclude that the vertical pass is not significantly slower (and in some cases even faster) than the horizontal pass, even though it performs more elaborate cache management, see Algorithm 3.1.

Timings for 16-bit versus 32-bit integers

We also benchmarked an implementation that uses 32-bit integers, see Table 3.5. For small wavelets like Haar, the timings for 16- and 32-bit differ by a factor of around 1.5, whereas for large wavelets the two are quite close. This is probably because the smaller wavelet transforms

Table 3.4. *Performance of our GPU implementation on 16-bit integers, separate timings of horizontal and vertical steps on a one-level decomposition of a* 1920×1080 *image.*

Wavelet (analysis)	**Horizontal (ms)**	**Vertical (ms)**
Haar	0.26	0.19
Deslauriers-Dubuc $(9, 7)$	0.44	0.42
Le Gall $(5, 3)$	0.39	0.34
Deslauriers-Dubuc $(13, 7)$	0.47	0.47
Daubechies $(9, 7)$	0.62	0.62
Fidelity	0.63	0.76
Wavelet (synthesis)	**Horizontal (ms)**	**Vertical (ms)**
Haar	0.29	0.19
Deslauriers-Dubuc $(9, 7)$	0.39	0.44
Le Gall $(5, 3)$	0.35	0.36
Deslauriers-Dubuc $(13, 7)$	0.42	0.48
Daubechies $(9, 7)$	0.58	0.79
Fidelity	0.59	0.64

Table 3.5. *Performance of our GPU implementation on 16 versus 32-bit integers (3 level transform,* 1920×1080 *image).*

Wavelet (analysis)	**16-bit (ms)**	**32-bit (ms)**
Haar	0.80	1.09
Deslauriers-Dubuc $(9, 7)$	1.50	1.64
Le Gall $(5, 3)$	1.34	1.45
Deslauriers-Dubuc $(13, 7)$	1.62	1.75
Daubechies $(9, 7)$	2.05	2.13
Fidelity	2.11	2.72
Wavelet (synthesis)	**16-bit (ms)**	**32-bit (ms)**
Haar	0.83	1.15
Deslauriers-Dubuc $(9, 7)$	1.45	1.81
Le Gall $(5, 3)$	1.28	1.66
Deslauriers-Dubuc $(13, 7)$	1.55	1.90
Daubechies $(9, 7)$	2.04	2.35
Fidelity	2.18	2.80

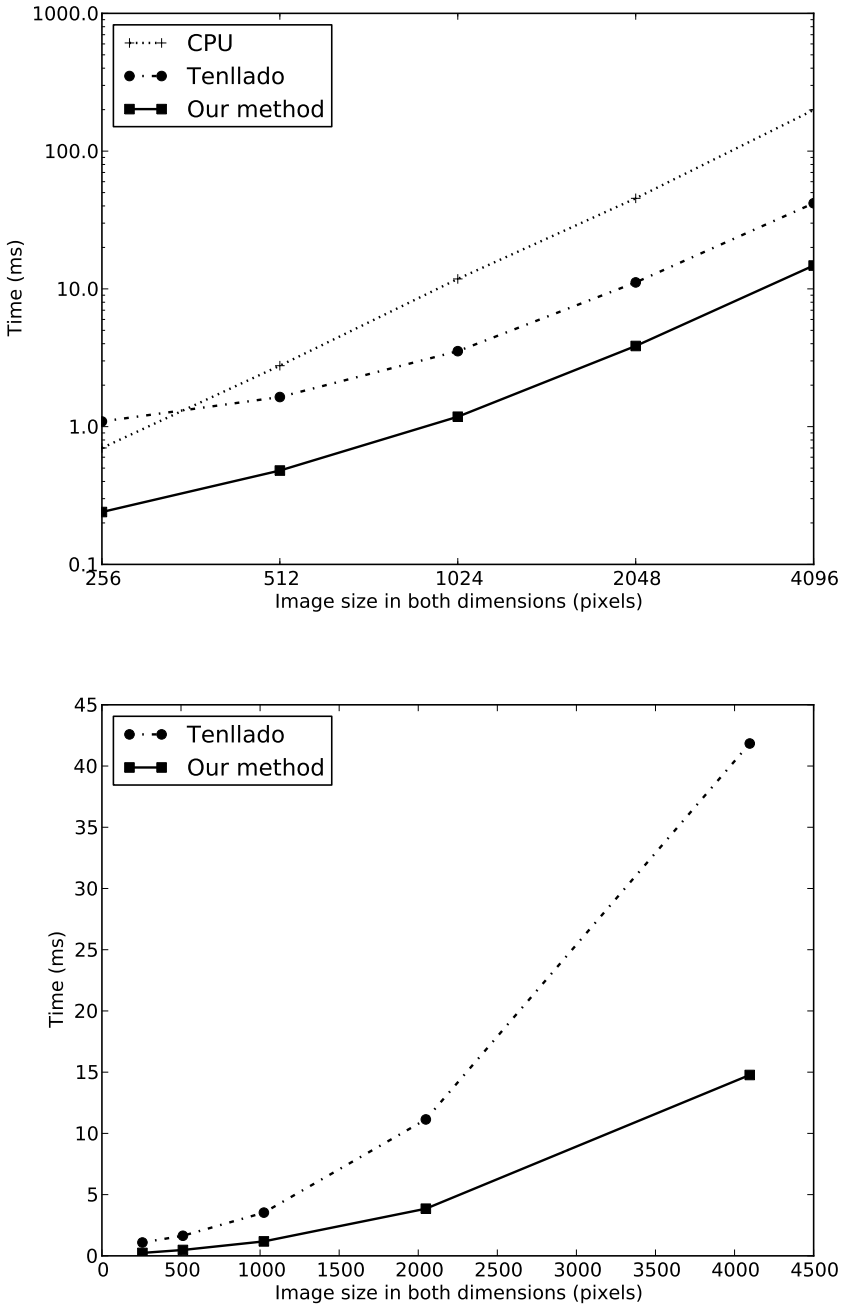


Figure 3.9. *Computation time versus image size for various lifting implementations; 3-level Daubechies* $(9, 7)$ *forward transform. Top: the Schrödinger CPU implementation, Tenllado* et al. *[135] and our CUDA accelerated method in a log-log plot. Bottom: just the two GPU methods in a linear plot.*

are more memory-bound and the larger wavelets are more compute-bound, hence the increased memory bandwidth does not affect the performance significantly.

Comparison of 2D wavelet lifting on GPU, CUDA versus fragment shaders

We also implemented the algorithm of Tenllado *et al.* [135] for wavelet lifting using conventional fragment shaders and performed timings on the same hardware. A three-level Daubechies $(9, 7)$ forward wavelet transform was applied to a 1920×1080 image, which took 5.99 milliseconds. In comparison, our CUDA-based implementation (see Table 3.3) does the same in 2.05 milliseconds, which is about 2.9 times faster. This speedup probably occurs because our method effectively makes use of CUDA shared memory to compute intermediate lifting steps, conserving GPU memory bandwidth, which is the bottleneck in the Tenllado method. Another drawback that we noticed while implementing the method is that an important advantage of wavelet lifting, i.e., that it can be done in place, appears to have been ignored. This is possibly due to an OpenGL restriction by which it is not allowed to use the source buffer as destination, the same result is achieved by alternating between two halves of a buffer, resulting in a doubling of memory usage.

Figure 3.9 further compares the performance of the Schrödinger CPU implementation, Tenllado *et al.* [135] and our CUDA accelerated method. A three-level Daubechies $(9, 7)$ forward wavelet decomposition was applied to images of different sizes, and the computation time was plotted versus image size in a log-log graph. This shows that our method is faster by a constant factor, regardless of the image size. Even for smaller images, our CUDA accelerated implementation is faster than the CPU implementation, whereas the shader-based method of Tenllado is slower for 256×256 images, due to OpenGL rendering and state set-up overhead. CUDA kernel calls are relatively lightweight, so this problem does not arise in our approach. For larger images the overhead averages out, but as the method is less bandwidth efficient it remains behind by a significant factor.

Comparison of lifting versus convolution in CUDA

Additionally, we compared our method to a convolution-based wavelet transform implemented in CUDA, one that uses shared memory to perform the convolution plus downsampling (analysis), or upsampling plus convolution (synthesis) efficiently. On a 1920×1080 image, for a three-level transform with the Daubechies $(9, 7)$ wavelet, the following timings are observed: 3.4 ms for analysis and 5.0 ms for synthesis. The analysis is faster than the synthesis because it requires less computations – only half of the coefficients have to be computed, while the other half is discarded in the downsampling step. Compared to the 2.0 ms of our own method for both transforms, this is significantly slower. This matches the expectation that a speedup factor of 1.5 to 2 can be achieved when using lifting [132].

Timings for 3D wavelet lifting in CUDA

Timings for the 3-D approach outlined in Section 3.4.7 are given in Table 3.6. A three-level transform was applied to a 512^3 volume, using various wavelets. The timings are compared to

Table 3.6. *Performance of our GPU lifting implementation in 3D, compared to an optimized CPU implementation; a three-level decomposition for both analysis and synthesis is performed, on a* 512^3 *volume.*

Wavelet (analysis)	**CPU** (ms)	**GPU** (ms)	**Speed-up**
Haar	1037.4	147.2	7.0
Deslauriers-Dubuc $(9, 7)$	2333.1	192.0	12.2
Le Gall $(5, 3)$	1636.2	179.3	9.1
Deslauriers-Dubuc $(13, 7)$	3056.1	200.5	15.2
Daubechies $(9, 7)$	3041.3	234.6	13.0
Fidelity	5918.4	239.2	24.7
Wavelet (synthesis)	**CPU** (ms)	**GPU** (ms)	**Speed-up**
Haar	926.5	150.9	6.1
Deslauriers-Dubuc $(9, 7)$	2289.9	184.7	12.4
Le Gall $(5, 3)$	1631.1	173.7	9.4
Deslauriers-Dubuc $(13, 7)$	2983.9	192.1	15.5
Daubechies $(9, 7)$	2943.5	232.0	12.7
Fidelity	5830.7	230.9	25.3

the same CPU implementation as before, extended to 3-D. The numbers show that the speed-ups that can be achieved for higher dimensional transforms are considerable, especially for the larger wavelets such as Deslauriers-Dubuc $(13, 7)$ or Fidelity.

Summary of experimental results

Compared to an optimized CPU implementation, we have seen performance gains of up to nearly 14 times for $2D$ and up to 25 times for $3D$ images by using our CUDA based wavelet lifting method. Especially for the larger wavelets, the gains are substantial. When compared to the trivial transpose-based method our method came out about two times faster over the entire spectrum of wavelets. When regarding computation time versus image size, our GPU based wavelet lifting method was measured to be the fastest of three methods for all image sizes, with the factor mostly independent of the image size.

3.5.3 Performance Analysis

We analyze the performance of our GPU implementation, according to the metrics from Section 3.4.2, for performing one lifting (analysis) step. Without loss of generality, we discuss the Deslauriers-Dubuc $(13, 7)$ wavelet, cf. Section 3.4.6. Our systematic approach consists first in explaining the total execution time, throughput and bandwidth of our method, and then in discussing the design decisions we made. The overhead of data transfer between CPU and GPU

was excluded, since the wavelet transform is usually part of a larger processing pipeline (such as a video codec), of which multiple steps can be carried out on the GPU.

Horizontal step

The size of the input data set is $N = w \cdot h = 1920 \cdot 1080$ two-byte words. We set $T = 256$ threads per block, and given the number of registers and the size of the shared memory used by our kernel, NVidia's occupancy calculator indicates that $k = 3$ blocks are active per MP, such that each MP is fully occupied (i.e., $kT = 768$ threads will be scheduled); the number of thread blocks for the horizontal step is $B = 1080$. Given that the 8800 GTX GPU has $M = 16$ MPs it follows that $\alpha = 23$, see Section 3.4.2. Further, we used *decuda* (a disassembler of GPU binaries; see http://wiki.github.com/laanwj/decuda) to count the number and type of instructions performed. After unrolling the loops, we found that the kernel has 309 instructions, 182 of which are arithmetic operations in local memory and registers, 15 instructions are half-width (i.e., instruction code is 32-bit wide), 82 are memory transfers and 30 are other instructions (mostly type conversions). Assuming that half-width instructions have a throughput of 2 cycles, and others take 4 cycles per warp, and since the clock rate of this device is $K = 1.35$ GHz, the asymptotic execution time is $T_e = 0.48$ ms. Here we assumed that the extra overhead due to rescheduling is negligible, as was confirmed by our experiments.

For the transfer time, we first computed the ratio of arithmetic to arithmetic-and-transfers instructions, which is $r = 0.67$. Thus, from Eq. (3.5) it follows that as many as 301 cycles can be spared due to latency hiding. As the amount of shared memory used by the kernel is relatively small (i.e., 3×3.75 KB used out of 16 KB per MP) and the size of the L2 cache is about 12 KB per MP [145], we can safely assume that the latency of a global memory access is about 350 cycles, so that $l_m = 49$ cycles. Since $m = 4$ (i.e., two two-byte words are coalesced), the transfer time is $T_m = 0.15$ ms. Note that as two MPs also share a small but faster L1 cache of 1.5 KB, the real transfer time could be even smaller than our estimate. Moreover, as we included also in our counting shared-memory transfers (whose latency is at least 10 times smaller than that of global memory), the real transfer time should be much smaller than its estimate.

According to our discussion in Section 3.4.5, five synchronization points are needed to ensure data consistency between individual steps. For one barrier, in the ideal case, the estimated waiting time is $T_s = 1.65\ \mu s$, thus the total time is about $8.25\ \mu s$. In the worst case $T_s = 0.2$ ms, so that the total time can be as large as 1 ms.

To summarize, the estimated execution time for the horizontal step is about $T_t = 0.63$ ms, neglecting the synchronization time. Comparing this result with the measured one from Table 3.4, one sees that the estimated total time is 0.16 ms larger than the measured one. Probably this is due to L1 caching contributing to a further decrease of T_m. However, essential is that the total time is dominated by the execution time, indicating a compute-bound kernel. As the timing remains essentially the same (cf. Tables 3.3 and 3.5) when switching from two-byte words to four-byte words data, this further strengthens our finding.

The measured throughput is $G_m = 98$ Gflop/s, whereas the estimated one is $G_e = 104$ Gflop/s, indicating on average an instruction throughput of about 100 Gflop/s. Note that with some abuse of terminology we refer to flops, when in fact we mean arithmetic instructions on in-

tegers. The measured bandwidth is $M_b = 8.8$ GB/s, i.e., we are quite far from the pin-bandwidth (86 GB/s) of the GPU, thus one can conclude again that our kernel is indeed compute-bound. This conclusion is further supported by the fact that the flop-to-byte ratio of the GPU is 5, while in our case this ratio is about 11. The fact that the kernel does not achieve the maximum throughput (using shared memory) of about 230 Gflop/s is most likely due to the fact that the synchronization time cannot simply be neglected and seems to play an important role in the overall performance.

Let us now focus on the design choices we have made. Using $T = 256$ threads per block amounts to optimal time slicing (latency hiding), see discussion above and in Section 3.4.2, while we are still able to coalesce memory transfers. To decrease the synchronization time, lighter threads are suggested implying that their number should increase, while maintaining a fixed size of the problem. NVidia's performance guidelines [92] suggest that the optimal number of threads per block should be a multiple of 64. The next higher than 256 multiple of 64 is 320. Unfortunately, using 320 threads per block means that at most two blocks can be allocated to one MP, and thus the MP will not be fully occupied. This in turn implies that an important amount of idle cycles spent on memory transfers cannot be saved, rendering the method less optimal with respect to time slicing. Accordingly, our choice of $T = 256$ threads per block is optimal. Further, our choice on the number of blocks also fulfills NVidia's guidelines with respect to current and future GPUs, see [92].

Vertical step

While conceptually more involved than the horizontal step, the overall performance figure for the vertical step is rather similar to the horizontal one. The CUDA configuration for this kernel is as follows. Each 2D thread block contains a number of $16 \times 8 = 128$ threads, while the number of columns within each slab is $S = 32$, see Figure 3.6. Thus, since the input consists of two-byte words, each thread performs coalesced memory transfers of $m = 4$ bytes, similar to the horizontal step. As the number of blocks is $w/S = 60$, $k = 4$ (i.e., four blocks are scheduled per MP), and the kernel takes 39240 cycles per warp to execute, the execution time for the vertical step is $T_e = 0.46$.

Unlike the horizontal step, now $r = 0.83$ so that no less than 352 cycles can be spared in global-memory transaction. Note that when computing r we only counted global-memory transfers, as in this case more, much faster shared-memory transfers take place, see Algorithm 3.1. As the shared-memory usage is only 4×1.8 KB, this suggests that the overhead due to slow accesses to global memory can be neglected, so that the transfer time T_m can be neglected. The waiting time is $T_s = 0.047\ \mu s$, and there are 344 synchronization points for the vertical-step kernel, so that the total time is about $15.6\ \mu s$. In the worst case, this time can be as large as 1.9 ms. Thus, as $T_t = 0.46$ (without waiting time), our estimate is very close to the measured execution time from Table 3.4 – this being in turn the same as that of the horizontal step. Finally, both the measured and estimated throughputs are comparable to their counterparts of the horizontal step.

Note that compared to the manually-tuned, optimally-designed matrix-multiplication algorithm of [145] which is able to achieve a maximum throughput of 186 Gflop/s, the performance of 100 Gflop/s of our lifting algorithms may not seem impressive. However, one should keep in mind that matrix-multiplication is much easier to parallelize efficiently, as it requires little

synchronization. Unlike matrix-multiplication, the lifting algorithm requires a lot more synchronization points to ensure data consistency between steps, as the transformation is done in-place.

The configuration we chose for this kernel is $16 \times 8 = 128$ threads per block and $w/S = 60$ thread blocks. This results in an occupancy of 512 threads per MP, which may seem less optimal. However, to increase the number of threads per block to 192 (next larger multiple of 64, see above), would mean that either we cannot perform essential, coalesced memory accesses, or that extra overhead due to the requirements of the moving-window algorithm would have to be accommodated. Note that we verified this possibility, but the results were unsatisfactory.

Complexity

Based on the formulae from Section 3.4.2 we can analyze the complexity of our problem. For any of the lifting steps using the Deslauriers-Dubuc $(13, 7)$ wavelet, considering that the number of flops per data element is $n_s = 22$ (20 multiply or additions and 2 register-shifts to increase accuracy), the numerator of (3.9) becomes about $700\,D$. For the horizontal step, $D = w/T = 7.5$, so that the numerator becomes about 5000. In this case the number of cycles is about 1250, so that one can conclude that the horizontal step is indeed *cost efficient*. For the vertical step, $D = (S\,h)/T = 270$, so that the numerator in (3.9) becomes about 190000, while the denominator is 39240. Thus, the vertical step is also cost efficient, and actually its performance is similar to that of the horizontal step (because $5000/1250 \approx 190000/39240 \approx 5$). Of course, this result was already obtained experimentally, see Table 3.4. Note that using vectorized MMX and SSE instructions, the *optimized* CPU implementation (see Table 3.3) can be up to four times faster than our T_S estimate above. However, even in this case, both our CUDA kernels are still cost-efficient. Obviously both steps are also *work efficient*, as their CUDA realizations do not perform asymptotically more operations than the sequential algorithm.

3.6 Conclusion

We presented a novel, fast wavelet lifting implementation on graphics hardware using CUDA, which extends to any number of dimensions. The method tries to maximize coalesced memory access. We compared our method to an optimized CPU implementation of the lifting scheme, to another (non-CUDA based) GPU wavelet lifting method, and also to an implementation of the wavelet transform in CUDA via convolution. We implemented our method both for 2D and 3D data. The method is scalable and was shown to be the fastest GPU implementation among the methods considered. Our theoretical performance estimates turned out to be in fairly close agreement with the experimental observations. The complexity analysis revealed that our CUDA kernels are cost- and work-efficient.

Our proposed GPU algorithm can be applied in all cases were the Discrete Wavelet Transform based on the lifting scheme is part of a pipeline for processing large amounts of data. Examples are the encoding of static images, such as the wavelet-based successor to JPEG, JPEG2000 [123], or video coding schemes [9], which we already considered in [142].

Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink. Accelerating Wavelet-Based Video Coding on Graphics Hardware using CUDA. In *Proc. 6th International Symposium on Image and Signal Processing and Analysis (ISPA 2009)*, September 16–18, Salzburg, Austria. Pages 614–619, 2009.

Chapter 4

Accelerating Wavelet-Based Video Coding on Graphics Hardware

4.1 Introduction

The Discrete Wavelet Transform (DWT) has been widely applied in signal and image processing. To meet the computational requirements for systems that handle very large throughputs, for example in real-time multimedia processing, custom hardware has been developed. Another option is to use a more general, widely available and relatively cheap platform, such as GPU hardware.

We recently developed a hardware-accelerated DWT algorithm that makes use of NVidia's Compute Unified Device Architecture (CUDA) parallel programming model [70] to fully exploit the new features offered by the Tesla architecture, introduced in 2006 with the GeForce 8800 GPU [143]. It is a highly parallel computing architecture available for systems ranging from laptops to high-end compute servers. Our DWT implementation is based on the Lifting Scheme [132] which reduces the number of arithmetic operations compared to the straightforward convolution-based approach. Also, the memory usage is reduced by factoring the wavelet transform into a sequence of steps that can be performed *in-place*. This is a great advantage given the generally limited amount of high-speed memory and the large data sizes that have to be processed in multimedia applications. The method is scalable and the fastest GPU implementation available to date.

In this chapter, we will show how to integrate our accelerated wavelet lifting into an implementation of the Dirac Wavelet Video Codec (DWVC) [9]. This codec, first introduced by the BBC, is gaining popularity as a free, open source alternative to H.264 [154]. It is a modern video compression scheme that employs wavelet transforms for inter- and intra- frame image compression, and makes use of motion compensation for compact storage of the difference between successive frames. Moreover, it is on-par with other modern video codec systems, e.g., H.264, which has gained wide acceptance in many applications like Internet broadcasting. It is a good alternative because the usage of H.264 incurs royalty fees, and while these costs are manageable for commercial applications, they could become prohibitive for public domain initiatives such as video archives. DWVC provides an alternative that is free of these fees and is equal in compression rates and quality [139]. Another advantage of wavelet-based video compression is that, as

it uses a global transform, it does not suffer from the block artifacts otherwise seen in traditional DCT-based codecs.

The acceleration of video decoding using GPU hardware was also studied by Shen *et al.* [118], with the aim to provide an architecture for video coding on the GPU, with special focus on the motion compensation and frame arithmetic parts. As they were programming the GPU using vertex and fragment shaders, the authors had to overcome the additional complexity of mapping the video decoding process to the rendering pipeline. In this chapter we employ a more general programming architecture, which means that we can focus on the actual parallel implementation of the algorithms. Doing so, we achieve speedups of a factor 15 for the image operations, and a factor 7.2 for the entire pipeline. In addition to the wavelet transform, we will discuss how the motion compensation and frame arithmetic steps of this codec can be accelerated using CUDA. Our proposed algorithm applies similarly to other wavelet-based video coding schemes that make use of the lifting scheme, such as [78, 79, 101, 136].

The remainder of this chapter is organized as follows. In Section 4.2 we summarize the essentials of the hardware-accelerated DWT algorithm, including a brief discussion of the CUDA programming environment and execution model. Then in Section 4.3 we give the details of our implementation of the DWVC in CUDA on the GPU. Section 4.4 presents benchmark results and analyzes the performance of our method. Finally, in Section 4.5 we draw conclusions and discuss future avenues of research.

4.2 CUDA-based implementation of the DWT

4.2.1 CUDA overview

NVidia's CUDA programming environment allows the GPU to be programmed through traditional CPU means: a C++-like language and compiler. The usage of CUDA does not add any overhead over rendering-based approaches, as it is a native interface to the hardware, and not an abstraction layer. CUDA broadly follows the data-parallel model of computation [70]. The CPU invokes the GPU by calling a *kernel*, which is a special C++ function.

The lowest level of parallelism is formed by scalar execution units called *threads*. A large number of threads can run in parallel. Threads are organized in *blocks*, and the threads within a block can share data through fast shared memory. It is also possible to place synchronization points (barriers) to control flow between all threads within a block. The highest performance is realized if all threads within a *warp* of 32 consecutive threads take the same execution path. If flow control is used within a warp, and the threads take different paths, they have to wait for each other (*divergence*). The highest level, which encompasses the entire kernel invocation, is called the *grid*, which consists of blocks that execute in parallel (if multiprocessors are available). Currently blocks within a grid cannot communicate with each other.

The CUDA architecture provides access to several kinds of memory. *Global* (device) memory can be read and written in any order (random access). *Registers* are limited per-thread memory locations for local storage with very fast access. *Shared memory* is a limited per-block chunk of memory which is used for communication between threads in a block. *Texture memory* is a

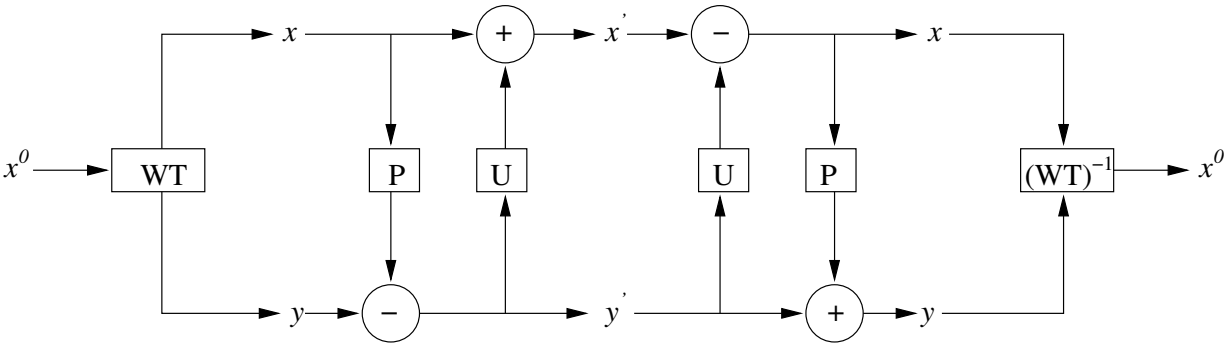


Figure 4.1. *Classical lifting scheme with one detail band. x^0 is the original image, WT is the lazy (trivial) wavelet transform that transforms the signal into approximation band x and detail band y, P is the prediction step, U the update step. $(WT)^{-1}$ is the inverse of the trivial wavelet transform.*

special case of device memory which is cached for locality. *Constant memory* is cached memory that can be written by the CPU and read by the GPU. To achieve highest throughput, consecutive memory locations must be simultaneously accessed by the threads. This is called *memory access coalescing* [70], and it represents one of the most important optimizations in CUDA.

4.2.2 Wavelet lifting

In wavelet lifting an input signal or image is decomposed into an *approximation band* x and one or more *detail bands* y_s using polyphase decomposition, see Fig. 4.1. In lifting theory, this polyphase decomposition is a (trivial) wavelet called the *lazy wavelet transform*, which splits the signal into two parts, containing the even and odd coefficients, respectively. A prediction step computes a prediction $P(x)$ from the approximation band x, after which the predicted values are subtracted from the detail band y_s, to produce a new detail band y'_s, i.e.,

$$y'_s = y_s - P(x). \tag{4.1}$$

An update step updates the approximation band x using the detail bands y_s,

$$x' = x + U\left(y_s\right). \tag{4.2}$$

The entire scheme is reversible by applying the steps in reverse order while interchanging $-$ and $+$. For a multilevel transform the process is repeatedly applied to the approximation bands, until a desired number of decomposition levels is reached. Wavelet lifting has the additional property that it can be done entirely with *integer operations*, resulting in a lossless scheme when applied to discrete images.

4.2.3 Wavelet lifting in CUDA

For separable wavelet bases in 2-D it is possible to split the operation into a horizontal and a vertical filtering step. For each filter level, a horizontal pass performs a 1-D transform on each

row, while a vertical pass computes a 1-D transform on each column. Each row can be handled in parallel during the horizontal pass, and then each column can be handled in parallel during the vertical pass. In CUDA this implies the use of two kernels, one for each pass.

In the horizontal pass, each block starts by reading a line into shared memory using so-called *coalesced reads* from device memory, executes the lifting steps in-place in fast shared memory, and writes back the result using *coalesced writes*. Some duplication of border elements is necessary to properly implement the boundary conditions. Each step is dependent on the output in shared memory of the previous step, therefore the threads within a block have to be synchronized every time before the next step can start. By reorganizing the coefficients [18] we can achieve higher efficiency for successive levels after the first transformation. To be able to coalesce, it must be possible to read back the coefficients consecutively, thus one writes the approximation and detail coefficients back to separate halves of the memory, de-interleaving them.

For the vertical pass we could use the same strategy as for the horizontal pass, substituting rows for columns. But we have shown [143] that a more efficient solution is possible which makes optimal use of coalesced memory access. Instead of having each block process a column, we make each block process multiple columns by dividing the image into vertical *slabs*. The number of columns in each slab is such that the resulting number of slab rows can still be coalesced. Each thread block then processes one of the slabs, so that a thread can do a coalesced read from each row within a slab, do filtering in shared memory, and do a coalesced write to each slab row.

Because the shared memory in CUDA is usually not large enough to store all columns, we use a sliding window within each slab. The dimensions of this window need to be such that each thread in the block can transform a signal element, and additional space to make sure that the support of the wavelet does not exceed the top or bottom of the window.

4.3 Accelerating the Dirac Video Codec

The weakest point of DWVC is currently its execution time [139]. Real-time decoding is limited to smaller resolutions (such as 800×600), even for the latest processors. The relatively heavy computational load of the global wavelet transform compared to more traditional block-wise Discrete Cosine Transforms (DCT) has prevented wavelets from being used in mainstream video compression. In an aim to speed up decoding, we implemented all the image operations of DWVC on the GPU, including the wavelet transform, motion compensation and frame arithmetic. As the CPU implementation (called Schrödinger) is already heavily optimized, it provides a good basis for performance comparison of our GPU wavelet lifting algorithm.

A DWVC stream consists of intra- and inter-frames. Intra-frames are self-contained images, while inter-frames store the difference with respect to one or two reference frames.

Decoding consists of three major parts (Fig. 4.2): arithmetic decoding, motion compensation, and inverse wavelet transform. Arithmetic decoding takes the bitstream and extracts parameters, motion vectors and wavelet coefficients needed to reconstruct the video sequence. It reverses the work of the entropy coder, which removes statistical redundancies from the data by representing

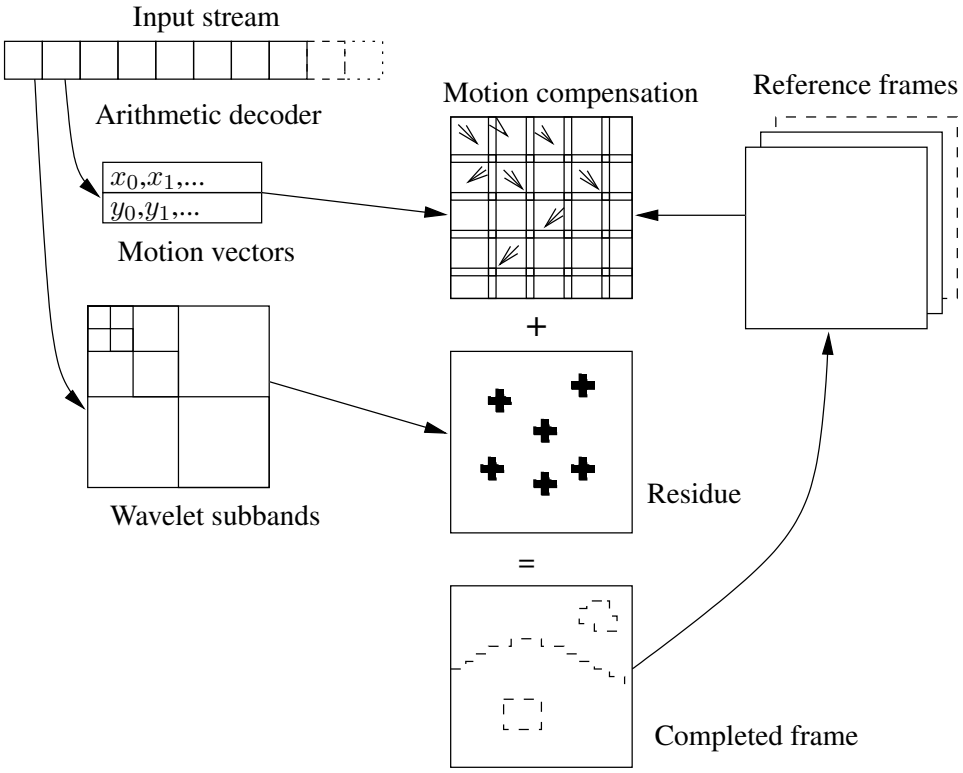


Figure 4.2. *Overview of decoding a DWVC stream. The steps are: arithmetic decoding, motion compensation, and inverse wavelet transform.*

common values with shorter bit sequences. This part is most conveniently handled by the CPU, as there is very little inherent parallelism in the process.

Motion compensation exploits the similarity between neighboring video frames. It reconstructs a frame from one or two frames preceding the current one in the stream. Motion compensation is done both on a global and local level. Global motion compensation seeks to describe movements of the camera, while local motion compensation acts on a per-block basis for smaller moving objects.

The images (for intra-frames) and residue (for inter-frames) are stored as wavelet coefficients in a per-component, per-subband basis in the video stream. Subbands that are zero or mostly zero are encoded as empty subbands, represented by only one bit. Only the non-zero subbands are transferred to the hardware. The wavelet filters that are used in the default settings of the encoder are the Deslauriers-Dubuc $(9, 7)$ filter [31] for intra-frames and the LeGall $(5, 3)$ filter [69] for inter-frames. A full list of wavelet filters used in DWVC can be found in [9].

The result of the motion compensation process is a prediction. The reconstructed residue is added to this prediction to form the final decoded frame which is shown on the screen. If the frame was marked as a reference for a future frame, it is stored until the stream tells it to retire.

As we already mentioned, we implemented the wavelet transforms, motion compensation and frame operations like adding, subtracting, conversion and (un)packing on the GPU. In the

upcoming sections we will discuss these operations, and show how they can be applied to decoding and encoding of video data.

4.3.1 Motion compensation

Traditional motion compensation algorithms divide the image into equally-sized, disjoint blocks of pixels. This has the disadvantage that there can be strong discontinuities between neighboring blocks, and moreover, the prediction accuracy on block edges is low. The residual difference image should be as smooth as possible to achieve the best compression, as jumps and discontinuities in the image cause large values in the detail subbands after the wavelet transform, which in turn results in a less compact representation. Overlapped Block Motion Compensation (OBMC) [7] overlaps neighboring blocks a bit, blending them together in the area which they share, thus increasing prediction accuracy.

The coverage of the image by blocks is defined using four parameters. The first two, `xlen` and `ylen`, define the size of the blocks in the horizontal and vertical direction. The second two parameters, `xsep` and `ysep`, define the separation of the beginning of a block to the beginning of the next one in the x- and y-direction, respectively. OBMC is a generalization of traditional motion compensation, as it equals standard disjoint motion compensation if `xlen` $=$ `xsep` and `ylen` $=$ `ysep`.

One or two reference frames can be arbitrarily selected from preceding or subsequent frames. For example, it is possible to do a blending between the previous and the next frame, useful in the case of a fade-in or fade-out, but it is also possible to use an image a few frames back for reference, if that image provides a better match to the current one. If two reference frames are used, these are blended together with weights w_1 and w_2.

A motion vector is a two-dimensional vector that stores the displacement of a block as compared the the reference frame. For example, if the previous frame is used as a reference, and an object moved two pixels to the right since the last frame, the motion vector for the block containing the object would be $(2, 0)$. Sub-pixel precision is supported by interpolating the reference frame using a bicubic spline filter. This fits perfectly to texture mapping hardware, if one stores the reference frames as textures.

As each pixel of the resulting image is computed independently, motion compensation is very well suited to a parallel GPU implementation. Each pixel can be part of up to four motion compensation blocks per reference frame. The value of an output pixel (x, y) is calculated by

$$I(x,y) = \sum_{m \in M} w_m(x,y) \Big(w_1 R_1(x + m_{x1}, y + m_{y1}) + w_2 R_2(x + m_{x2}, y + m_{y2}) \Big), \tag{4.3}$$

in which I is the output frame, M is the set of all blocks, $w_m(x,y)$ is the weight of block m at position (x,y), w_1 and w_2 are the reference frame weights, R_i are the reference frames, and (m_{xi}, m_{yi}), $i = 1, 2$, are the two motion vectors for block m. The block weights are defined so

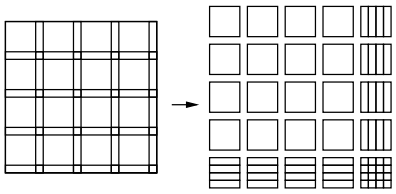


Figure 4.3. *Dividing the frame into four block types according to the number of overlapping blocks, for efficient motion compensation.*

that

$$\sum_{m \in M} w_m (x, y) = 1,$$

and the weights have a linear fall-off at the block edges.

The most obvious approach to a GPU implementation is to divide the image into equally-sized CUDA blocks, whose pixels are then processed by a CUDA thread. This thread determines which motion compensation blocks overlap a pixel, and calculates the output value using Eq. (4.3).

Such an approach will result in quite some flow control per pixel, and as neighboring pixels might need values from different motion compensation blocks, thread divergence arises. In other words, in the optimal setting threads within each CUDA block should perform the same operations, while different CUDA blocks can do different computations. An improved algorithm divides the image into regions according to the number of overlapping blocks and orientation of overlap (Fig. 4.3); in the center of the blocks, it suffices to take a sample from one block. Then there are the cases in which two blocks overlap, horizontally or vertically. Here, the two blocks need to be blended together linearly, so that there is a smooth transition. And finally there are the diagonal overlaps where four blocks overlap, which must be blended together. A bilinear blending is used to create a smooth transition here.

Therefore, for each region, we know exactly how many, and which blocks it depends on, and how to blend the blocks. Thus, regions can be handled in parallel, but each needs to be handled in a different way. In CUDA, different blocks can execute different kernels when a large conditional statement is put around them that depends on the block identifier. As the threads within the block all take the same path, no divergence is introduced. To use this, we pass a block type parameter to each CUDA block. This block type tells the kernel which of the four regions mentioned before should take part in the computation. As CUDA supports scattered writes, each block can write to the part of the image that it computed.

The motion vectors themselves can be passed to the kernel through an array in constant memory, or by using a texture. We noticed that using a texture is significantly faster than constant memory in a case like this, where each thread potentially accesses a different location, so this is preferred to the other approach. Also, unlike constant memory, textures have no 64KB limit.

4.3.2 Frame arithmetic

The frames resulting from the inverse wavelet transform and the motion compensation are added together to compute the resulting frame. All computations are done on 16-bit per component, but the final rendering needs an 8-bit image, so the values are clamped between 0 and 255. The output of the algorithm can be directly sent to a texture, without having to pass through the host CPU, by using the CUDA-OpenGL interoperability API. This texture can then be rendered on a quadrilateral to show the frame. If the frame is marked as a reference frame, a copy of the texture is kept for use in a later motion compensation stage.

The fact that DWVC uses 16-bit integers instead of 32-bit complicates the implementation, as both shared and global memory access is geared toward 32-bit values. Two 16-bit integers can be combined into a 32-bit read only as long as the address is aligned to four bytes, which means that reading or writing cannot start from an odd column. To get around this constraint, a one-column border at the left or right of the rectangle must be processed using 16-bit memory operations. Even though this gives some overhead, it is much faster than using only 16-bit memory accesses.

4.4 Performance results

The benchmarks in this section were run on a machine with a Dual Core AMD Opteron(tm) Processor 280 and a NVidia GeForce GTX280 graphics card, using CUDA version 2.2 for the CUDA programs. The codec was benchmarked in single-threaded mode. Using multiple threads on a multi-core machine would increase the performance of both the CPU and GPU implementations, but the coordination involved in using a GPU from multiple threads, though it became possible in version 2 of CUDA, is quite difficult. For the CUDA implementation, the result was not copied back to the CPU after each frame, as we used direct rendering through OpenGL textures.

In Table 4.1 we compare the overall performance of the DWVC accelerated by our GPU implementation to the optimized CPU implementation. The experiment was performed using two HD video sequences and one lower resolution sequence. Our method runs at an average of 50.9 frames per second for a 1920×1080 sequence, while the CPU version runs at 10.5 frames per second on the same video sequence. This means we achieve a speedup of 7.2 of the entire process.

A breakdown of the computation time into different stages for two different video sequences is shown in Table 4.2. To make a valid comparison of the total time spent in each stage, the CPU and GPU were synchronized between stages. This prevents overlap in computation and thus results in a somewhat lower overall performance. Stage 1 performs motion compensation (Section 4.3.1), and is a factor 8 to 12 faster in our CUDA implementation. Stage 2 performs arithmetic decoding of motion vectors and is the same in both implementations. Stage 3 decodes the wavelet subbands from the input stream. This stage is a bit slower for the CUDA version, because it includes copying the non-zero subbands to GPU memory. Stage 4 performs the inverse integer wavelet lifting transform (Section 4.2.2) on the decoded residue, and is a factor 9 to 13 faster in the CUDA implementation. Stage 5 combines the motion compensation result and

residue (Section 4.3.2), and is a factor 10 to 28 faster in the CUDA implementation. Stage 6 performs upsampling of reference frames for sub-pixel motion compensation, and is a factor 10 to 17 faster in the CUDA implementation.

By excluding *DECODE* (arithmetic decoding) stages, subtracting 12.64 from both totals for the sequence in Table 4.2, one can determine the speedup of the GPU accelerated operations compared to their CPU counterpart. This amounts to a factor of about 15.

The frame-rate achieved with our method (50.9) allows for playback of high definition video significantly faster than strictly needed for playback of those movie sequences (25).

Table 4.1. *Performance in frames per second (FPS) for decoding various DWVC video sequences with: (i) Schrödinger CPU implementation; (ii) Our CUDA implementation*

Sequence	**Frame size**	**CPU**	**CUDA**
Big Buck Bunny trailer	1920×1080	10.5	56.4
Elephant's Dream	1024×576	33.7	125.6
2012 movie trailer	1920×800	13.2	71.2

Table 4.2. *Big Buck Bunny trailer (813 frames, 1920×1080) decoded with: (i) Schrödinger CPU implementation; (ii) Our CUDA implementation*

Stage	**CPU** (s)	**CUDA** (s)
1 MOTION_DECODE	0.64	0.64
2 MOTION_RENDER	16.16	1.33
3 RESIDUAL_DECODE	12.00	12.94
4 WAVELET_TRANSFORM	22.52	1.63
5 COMBINE	11.27	0.39
6 UPSAMPLE	14.53	0.85
Total	77.13	17.76

4.5 Conclusion

In this chapter, we showed how to accelerate the Dirac Video Codec by a fast wavelet lifting implementation on graphics hardware using CUDA. The method maximizes coalesced memory access. We also accelerated the motion compensation and frame arithmetic stages of this codec.

The experiments on high definition video sequences have demonstrated that one can achieve a speedup factor of 7.2 for the entire decoding process including the CPU steps, and of 15 times

for just the GPU part. In our benchmark we could playback a 1080p resolution video at 50.9 frames per second.

As the decoding stages that remain on the CPU are quite involved, further work could involve the acceleration of the arithmetic decoding on (future) GPU hardware, or the development of statistics-based data compression methods that are more paralellizable.

First published as: Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen Space Fluid Rendering with Curvature Flow. In *Proc. I3D 2009: The 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 91-98, 2009.

Chapter 5

Screen Space Fluid Rendering with Curvature Flow

5.1 Introduction

For interactive applications such as games, particle based fluid simulation methods like Smoothed Particle Hydrodynamics (SPH) [30] are commonly preferred to Eulerian fluid representations. This is because the fluid is able to flow everywhere in the scene without the need to define a finite grid, which is costly in terms of memory and computation. Particle methods are also more convenient to integrate into existing physics systems as particles can collide against the scene geometry just like other rigid objects, without the need to voxelize the scene geometry into the grid. The drawback is that it is more difficult to extract a surface for rendering. Although there are extensive contributions in the literature [12, 22, 73] on particle-based fluid simulations, there is very little on rendering particle fluids. Of the methods that have been developed, most are not suitable for real-time use in games. Usually, the fluid surface is constructed in world-space, either directly as a mesh [126], or as implicit surface and then polygonized using Marching Cubes [76] or a similar method [109, 156]. After this, relaxation and optimization operations can be applied to the entire mesh to increase the smoothness of the surface, which is both computation and memory intensive.

Implicit surface polygonization methods also suffer from grid discretization artifacts in frame-to-frame coherence [3], as the grid is static and does not move with the fluid. This is especially visible when using low resolution grids in real-time rendering. In [160] an interesting point-based rendering approach is presented where ray-metaball intersections are computed entirely on the GPU in a two pass rendering approach. This method removes the requirements for a grid discretization.

If the fluid is moving and rendering is only desired from only one, or at most a few viewpoints per frame, a more memory and compute efficient method is to only construct the surface represented by the particles that are visible to the camera in view-space as in [87].

The main contribution of this chapter is a splatting-based fluid rendering method that

- Achieves real-time performance, with a configurable speed versus quality trade-off.
- Does all the processing, rendering and shading steps directly on the graphics hardware.

- smoothes the surface to prevent the fluid from looking blobby or jelly-like.
- Is not based on polygonization, and thus does not suffer from the associated tesellation artifacts.
- Is simple to implement, consisting of a few passes using fragment shaders and intermediate render targets.
- Has inherent view-dependent level-of-detail, since the method is based on a grid on screen-space.

Another contribution is a way to generate noise that moves with the particles on the surface of the fluid, which can be used to add foam-like effects and surface detail on a smaller scale than the particles themselves.

5.2 Related work

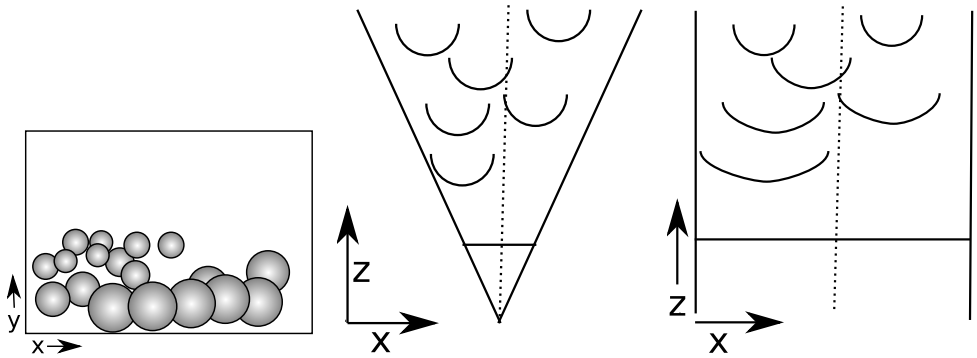
Rosenberg and Birdwell [109] optimized Marching Cubes specifically in the context of particle isosurface extraction, achieving real-time performance for up to 3000 particles. Although relatively fast, the results look quite blobby, and as their method directly renders the resulting mesh there is no way to post-process the result to improve quality. Also, for SPH fluids we typically need at least 10000 particles for the simulation to look realistic.

Williams outlines in his thesis [156] a new approach to surfacing particle-based fluid simulations. A generalization of Marching Cubes, called Marching Tiles, is used which allows constraints to be put on the quality and smoothness of the mesh. This results in nice smooth surfaces, but the approach is designed for offline rendering and is not real-time. It also suffers from the same drawbacks as Marching Cubes, such as having a fixed grid.

Somewhat related to our method is the *projected grid* as was introduced in [61] which transforms a displaced surface so that it resembles a uniform grid in post-perspective space as closely as possible. This provides spatial scalability as well as high relative resolution without resorting to LOD schemes. Our method relies on a uniform (per-pixel) grid in post-perspective space as well.

In [87] the authors present an approach for generating the boundary of a three-dimensional point cloud as a mesh in screen-space, generating the surface only where it is visible. It first computes the depth to the surface at each pixel on the screen, smoothes this depth map using a binomial filter, then polygonizes the depth buffer. The polygonization step is computationally very intensive, and does not map to graphics hardware in a straightforward way. In our method, instead of generating an intermediate mesh, the depth buffer is used for rendering directly.

Finally, in [160] the authors present an implicit ray-metaball algorithm that does not require an explicit metaball reconstruction, using point based rendering techniques and assimilating the metaballs as point based splats. First, non overlapping fluid surface points are determined using a GPU based dynamic grouping algorithm [159] and an initial ray-metaball intersection is computed. In a second pass the final intersection with the isosurface is determined for each pixel



(a) (b) (c)

Figure 5.1. *Drawing the particles as spheres (a) front view (b) in view-space and (c) after perspective projection.*

from the contributions of disjoint sets. The advantage of this method is that it does not require a subdivision grid for the surface reconstruction. In order to minimize the bumpy appearance resulting from the simulations, a fairly big metaball radius has to be used, at the cost of very thick surfaces.

Level-sets [81] have been used extensively in fluid simulations to track the interface between fluid and air as they provide a deformable surface representation that allows for topology changes. We have incorporated a few ideas from level-sets but have not applied the full method, as even efficient level-set methods involve too much computation to be used for this purpose in real-time.

Subsequent to our research we became aware of another publication [23] that uses a similar image-space fluid rendering technique but lacks the advanced smoothing and thickness-based rendering of our method.

5.3 Method

In this work we will assume a SPH particle simulation has already been carried out. The input data consists of the positions $\mathbf{x}_i$ of particles p_i, $i \in \{0..n\}$, in any order. Optionally, the particles can have an associated density ρ_i and velocity $\mathbf{v}_i$ coming from the simulation. A *high level overview* of the method is as follows: Starting from the fluid particle positions, surface depth (explained in section 5.3.1) and thickness (section 5.3.3) is written to two render targets. The surface depth is then smoothed (section 5.3.2), and a dynamic noise texture is generated on the surface of the fluid (section 5.3.4). Then, a compositing pass is performed that combines the smoothed surface depth, the noise texture and an image of the scene behind the fluid into the final rendering of the fluid (section 5.3.5).

5.3.1 Surface depth

Before anything can be rendered the front-most surface of the fluid from the viewpoint of the camera is determined. We do this by rendering the particles as spheres, and retaining the closest value at each pixel using the hardware depth test (see Figure 5.1).

In order to obtain a representation of the surface of the fluid from the viewer's point of view, we render the particles as spheres using *point sprites* (screen oriented quads) with depth replacement in the fragment shader. This avoids the use of complex geometry and is a well known technique. Unlike in surface splatting [3, 11], we do not explicitly splat the normal or shaded color values, but calculate the normals from the depth values while rendering. The reason for this is that the depth values will be manipulated by the smoothing step which will be discussed in next section. In some cases it is desirable to exclude stray particles from rendering as these do not form part of any surface. This is easily accomplished by putting a threshold on the density ρ_i obtained from the simulation. To make the transition more smooth, the low-density particles can be rendered separately as spray.

5.3.2 Smoothing methods

It is not desirable for the particles to be visible as spheres since this results in an unrealistic jelly-like appearance. We would like a smooth, flat surface that approximates the particle positions. In our method, we achieve this by smoothing the surface in screen-space.

An obvious approach is use a Gaussian blur or variants such as Bilateral Gaussian filters [6] or more advanced filters like [19]. However, straightforward Gaussian blurs will cause blurring over silhouette edges and can cause plateaus of equal depth when using large kernels. Bilateral filters preserve edges, but are non-separable and therefore expensive. It is difficult to implement a blur with a variable-width kernel efficiently on graphics hardware.

As an alternative to Gaussian smoothing, we can look at the problem in a different way: we are interested in a method that smoothes out sudden changes in curvature between the particles, forming a smooth and continuous surface. One way to think of this is to minimize the curvature. This also has a natural motivation, as it is similar to surface tension in fluids which is responsible for the formation of water drops and puddles. A more general name for this process is called *curvature flow* [81].

Curvature flow evolves a surface along its normal direction with the speed depending on the magnitude and sign of the mean curvature of the surface, and is well-known from the level-set literature. In our application we are working on a depth buffer, which means that the surface can only be moved in the z direction perpendicular to the view plane. However, as the viewpoint is constant we still achieve the desired effect of smoothing the surface by moving the z value in proportion to the curvature, thus we define

$$\frac{\partial z}{\partial t} = H, \tag{5.1}$$

in which t is a smoothing time step, and H is the mean curvature. From now on, we will call this method *screen-space curvature flow*.

Mean curvature is defined as the divergence of the unit normal of a surface,

$$2H = \nabla \cdot \hat{n} \tag{5.2}$$

By inverting the projection transformation, a value in the depth buffer is mapped back to a point P in view space. V_x and V_y are the dimensions of the viewport, and F_x and F_y is the focal length in the x and y direction subsequently,

$$\mathbf{P}(x,y) = \begin{pmatrix} \frac{\frac{2x}{V_x} - 1.0}{F_x} \\ \frac{\frac{2y}{V_y} - 1.0}{F_y} \\ 1 \end{pmatrix} z(x,y) = \begin{pmatrix} W_x \\ W_y \\ 1 \end{pmatrix} z(x,y) \tag{5.3}$$

The normal is calculated by taking the cross product between the derivatives of $\mathbf{P}$ in the x and y direction,

$$\begin{aligned} \mathbf{n}(x,y) &= \tfrac{\partial \mathbf{P}}{\partial x} \times \tfrac{\partial \mathbf{P}}{\partial y} \\ &= \begin{pmatrix} C_x z + W_x \frac{\partial z}{\partial x} \\ W_y \frac{\partial z}{\partial x} \\ \frac{\partial z}{\partial x} \end{pmatrix} \times \begin{pmatrix} W_x \frac{\partial z}{\partial y} \\ C_y z + W_y \frac{\partial z}{\partial y} \\ \frac{\partial z}{\partial y} \end{pmatrix} \\ &\approx \begin{pmatrix} C_x z \\ 0 \\ \frac{\partial z}{\partial x} \end{pmatrix} \times \begin{pmatrix} 0 \\ C_y z \\ \frac{\partial z}{\partial y} \end{pmatrix} = \begin{pmatrix} -C_y \frac{\partial z}{\partial x} \\ -C_x \frac{\partial z}{\partial y} \\ C_x C_y z \end{pmatrix} z, \end{aligned}$$

in which $C_x = \frac{2}{V_x F_x}$, $C_y = \frac{2}{V_y F_y}$, we chose to ignore the terms of the derivative of P that depend on the view position W_x, W_y because it simplifies the computations a lot, and the difference is negligible as the contributions are small. The unit normal

$$\hat{n}(x,y) = \tfrac{\mathbf{n}(x,y)}{|\mathbf{n}(x,y)|} = \tfrac{\left(-C_y \frac{\partial z}{\partial x}, -C_x \frac{\partial z}{\partial y}, C_x C_y z\right)^T}{\sqrt{D}}, \tag{5.4}$$

in which

$$D = C_y^2 \left(\frac{\partial z}{\partial x}\right)^2 + C_x^2 \left(\frac{\partial z}{\partial y}\right)^2 + C_x^2 C_y^2 z^2 \tag{5.5}$$

is substituted in the equation for mean curvature (Eq. 5.2), so that H can be derived. The z component of the divergence is always zero, as z is a function of x, and y and thus does not change when these are kept constant. We get

$$2H = \frac{\partial \hat{n}_x}{\partial x} + \frac{\partial \hat{n}_y}{\partial y} = \frac{C_y E_x + C_x E_y}{D^{\frac{3}{2}}} \tag{5.6}$$

in which

$$E_x = \frac{1}{2} \frac{\partial z}{\partial x} \frac{\partial D}{\partial x} - \frac{\partial^2 z}{\partial x^2} D, \tag{5.7}$$

$$E_y = \frac{1}{2} \frac{\partial z}{\partial y} \frac{\partial D}{\partial y} - \frac{\partial^2 z}{\partial y^2} D \tag{5.8}$$

A simple Euler integration of Eq. 5.1 in time is used to modify the z values in each iteration. The spatial derivatives of z are computed using finite differencing.

The surface may be discontinuous because of silhouettes in screen-space. To prevent blending different patches of surface together it is important to make sure that boundary conditions are enforced where large changes in depth occur between one pixel and the next. At these boundaries, and the edges of the screen, we force the spatial derivatives to be 0 to prevent any smoothing from taking place.

The number of iterations is chosen depending on the smoothness that is desired. The more iterations the smoother the surface will be, but this comes at the expense of increased computation time.

5.3.3 Thickness

One expects an object to become less visible depending on the amount of fluid that is in front of it. To accomplish this we need to compute the amount of fluid between the camera and the nearest opaque object for each pixel, which we refer to as the "thickness". When rendering, the thickness is used to attenuate the color and transparency of the fluid.

The particles are regarded as spheres of fluid with a fixed size in world space. The rendering process is the same as that in Section 5.3.1, with the difference that the fragment shader outputs the thickness of the particle at that position instead of a depth value. Additive blending is used so that the amount of fluid is accumulated at each position on the screen. Depth test is enabled, so that only particles in front of the scene geometry are rendered,

$$T(x,y) = \sum_{i=0}^{n} d(\frac{x-x_i}{\sigma_i}, \frac{y-y_i}{\sigma_i}), \tag{5.9}$$

where d is the depth kernel function, x_i and y_i are the projected position of the particle, x and y are screen coordinates, and σ_i is the projected size.

Strictly speaking this measure of thickness is only correct if the particles do not overlap, but this is a reasonable assumption in SPH due to repulsive inter-particle forces.

5.3.4 Noise

Although our method helps to hide the particle-based nature of the fluid the result can still look artificially smooth. Surface detail and foam is an important visual element in real fluids. A straightforward way to improve this would be to perturb the surface using a noise texture and thus add small-scale detail, as in [61]. However, generating fixed noise in world space or eye space makes it appear as if the noise is stuck in place. The challenge is to have noise that is advected by the fluid, but is of a smaller scale and higher frequency than the simulated, particle based fluid.

Instead we propose to use Perlin noise [100] by assigning one octave of noise to each projected particle based on its index value, so that a certain pattern of noise remains with each

particle. By using additive blending, this results in a Perlin noise texture in which the octaves move relative to each other and along with the flow.

For each particle a *point sprite* is rendered with a Gaussian kernel. The resulting value is multiplied with an exponential fall-off based on the depth below the surface, so that particles contribute less as they submerge,

$$I(x, y) = \text{noise}(x, y) * e^{-x^2-y^2-(\mathbf{p}_z(x,y)-d(x,y))^2}, \tag{5.10}$$

in which $\mathbf{p}$ is the view-space position of this pixel, d the depth as sampled from the surface depth texture, and x and y vary between -1 and 1. The noise texture noise is varied per particle to prevent patterns from becoming apparent.

This noise kernel is then summed for every particle on the screen to get a noise value at every pixel to be used for shading,

$$N(x, y) = \sum_{i=0}^{n} I(\frac{x - x_i}{\sigma_i}, \frac{y - y_i}{\sigma_i}), \tag{5.11}$$

Fluid should become more perturbed when the flow is violent, and this is achieved by marking the fluid particles when a large change in velocity $\mathbf{v}_i$ happens,

$$|\mathbf{v}_i(t) - \mathbf{v}_i(t - 1)| > \tau, \tag{5.12}$$

where τ is a threshold value. For these particles, the noise amplitude will be higher. After a while, the particles cool down and revert to normal.

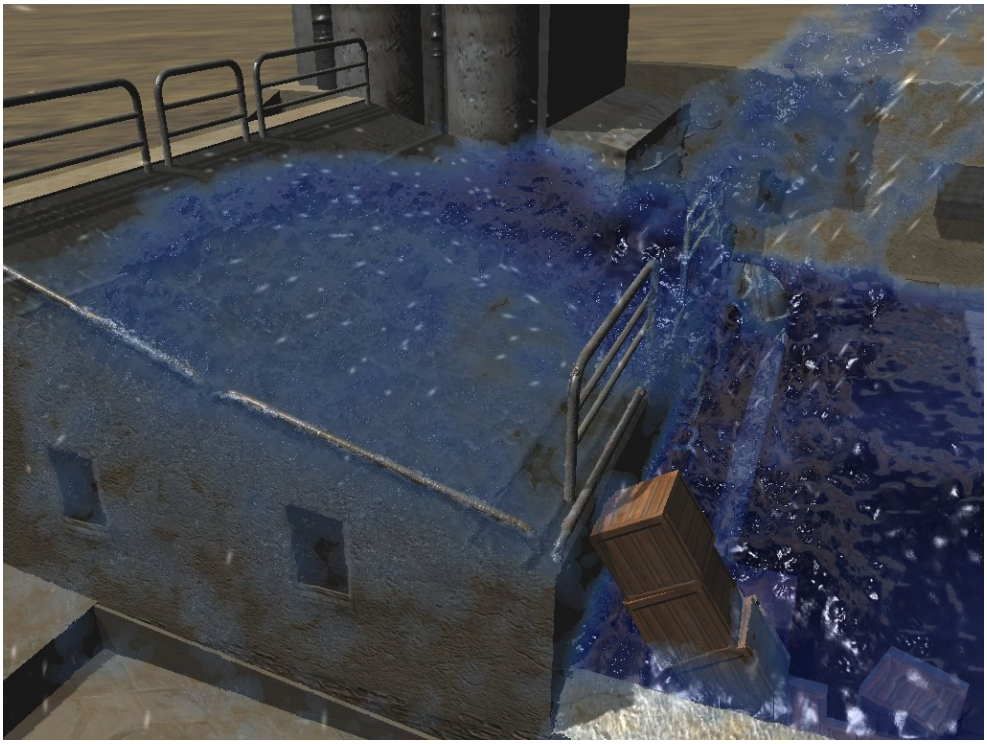
5.3.5 Rendering

In the final step, all the intermediate results are composited into a final image by rendering a full-screen quad. The optical properties of the fluid are based on the Fresnel equation, with a reflection and refraction component and a Phong specular highlight, computing the output color

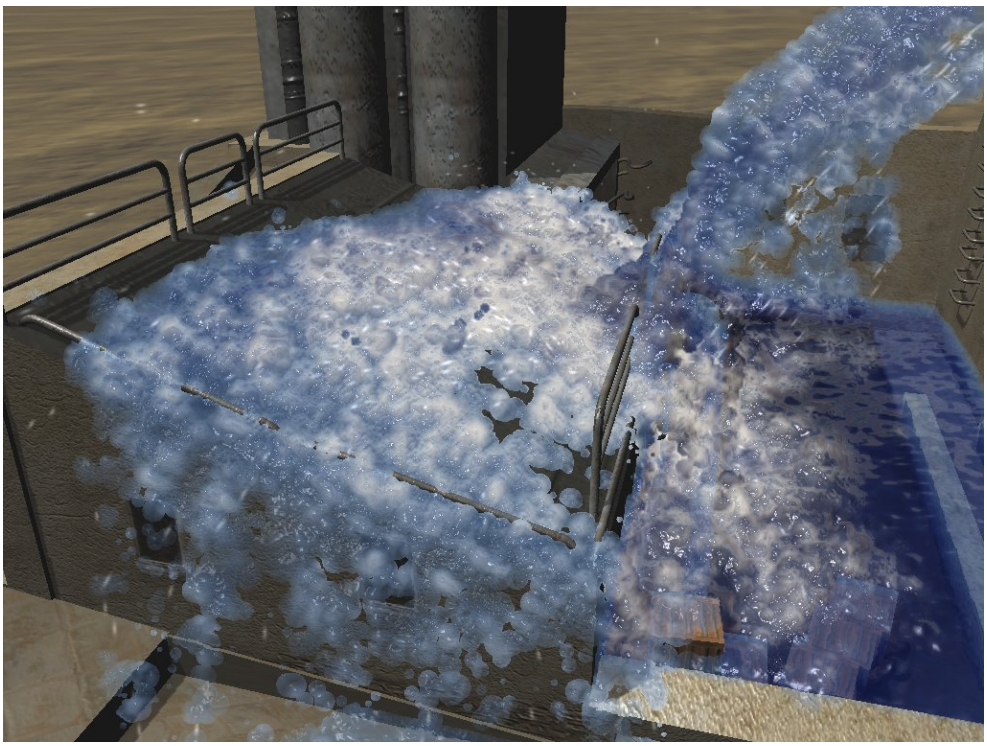
$$C_{out} = a(1 - F(\mathbf{n} \cdot \mathbf{v})) + bF(\mathbf{n} \cdot \mathbf{v}) + k_s(\mathbf{n} \cdot \mathbf{h})^\alpha, \tag{5.13}$$

where F is the Fresnel function, a is the refracted fluid color, b is the reflected scene color, k_s and α are constants for the specular highlight, $\mathbf{n}$ is the surface normal and $\mathbf{h}$ is the half-angle between the camera and the light, and $\mathbf{v}$ is the camera vector. Depth test is enabled when rendering the fluid, and the depth returned by the fragment shader is copied from the surface depth (see section 5.3.1).

To shade the surface of the fluid the view-space normals $\mathbf{n}$ are calculated using the finite differences of the surface depth $d(x, y)$, as in equation 5.4. Simply using the finite differences in one direction to calculate the normal will result in artifacts along the silhouettes. When a discontinuity is detected, by comparing the difference in depth to a threshold, we chose the smallest absolute finite difference (for example, the smallest of $|z(x, y) - z(x + 1, y)|$ and $|z(x, y) - z(x - 1, y)|$). In addition to this, the noise texture $N(x, y)$ is used to perturb the normals to add small, wave-like



(a) Without foam



(b) With foam

Figure 5.2. *Same scene with foam enabled and disabled. Rendered using screen-space curvature flow, with smoothing computed at half resolution.*

surface detail to the fluid by adding the partial derivatives of the noise texture to the calculated normals. Furthermore, a grayish color can be added depending on the magnitude of the noise to simulate a surface foam effect like in Figure 5.2.

The thickness $T(x, y)$ is used to attenuate the refracted color of the fluid a,

$$a = \text{lerp}(C_{\textit{fluid}}, S(x + \beta \mathbf{n}_x, y + \beta \mathbf{n}_y), e^{-T(x,y)}), \tag{5.14}$$

the thicker the fluid, the more it attenuates the background color. Thin areas of the fluid show through the background scene. When shading the fluid we use a slightly different exponential fall-off for each color channel, so that the color varies in an interesting way with the thickness. For the transparency, the scene without the fluid is first rendered to a background texture $S(x, y)$. The texture coordinates used to sample the background scene texture are perturbed based on the normal of the surface $\mathbf{n}$ to give the illusion of refracting the object behind the fluid. β increases linearly with the thickness,

$$\beta = T(x, y)\gamma, \tag{5.15}$$

in which γ is a constant that depends on the kind of fluid, and determines how much the background is refracted. The reflected color b is determined by sampling a cubemap texture of the environment based on the reflected direction, computed from the surface normal and the view vector.

Interpolation

As the PDE for curvature minimization is stiff, and an explicit integration scheme is used, stability issues can arise causing the system to oscillate. For this reason, at high resolutions it takes a lot of iterations at a small timestep to retain stability. A trade-off can be made to sacrifice some quality for performance by using an approach like that in [17], doing both the fluid rendering and post-processing steps at a lower resolution. The scaling is difficult due to the presence of silhouettes. Inside a body of fluid, the depth is interpolated linearly, but silhouettes are handled as a special case. These should not look sharp or jagged, if they look blurry or smooth it is more acceptable. For this reason, we blend the final shaded color, computed at low resolution, over edges instead of the normal or depth value. This has the effect of smoothing the silhouettes.

Interpolation has the result that high frequency features will be lost due to the sampling. Because of the smoothness, half or quarter resolution fluid can look better than full resolution from close up.

5.4 Results and discussion

All benchmarks were performed on a NVIDIA GForce 8800GTS 512 in 1024×768 resolution. The result of a NVIDIA PhysX SPH fluid simulation of 64000 particles was rendered using our method. The computation time of the simulation is not included in the results.

Performance figures are shown in Table 5.1 for both the corridor scene (Figure 5.5) and the NVIDIA eye logo (Figure 5.6). In the table, smoothing based on a a two-pass bilateral blur

using a Gaussian kernel is compared against various settings of screen-space curvature flow. This method, at quarter resolution is even a little bit faster than the Gaussian smoothing. Half resolution is slower, and full resolution is much slower, because the number of iterations needs to be increased to achieve stability.

It is important to note that the advantage of the presented method is that it allows to achieve a higher degree of smoothness at a lower cost than the Bilateral Gaussian smoothing method, and in particular avoids disruptive artifacts caused by using a separable filter approach on non-separable kernels. In Figure 5.8 we can see that a similar image quality is achieved by running six iterations of the Bilateral Gaussian, with a performance degradation. In Table 5.2 we present the overall frame cost for different settings of the Bilateral Gaussian blur filter. It shows that the most significant penalty is paid when the number of iterations is increased to produce a similar image to the curvature flow method, presented in this chapter.

When adding noise and foam the rendering becomes significantly slower, because of the extra rendering pass that splats the noise kernels. The performance could be improved by rendering only particles close to the surface, if this information is available from the simulation. For example, the density ρ_i might be used, as the density is lower the closer you get to the surface.

Figure 5.2 shows a waterfall with and without foam. The detail added by the foam makes the waterfall look more rough, like a real waterfall, and makes it look less like a synthetic smooth fluid. Figure 5.3 shows a close-up of the fluid itself, with three rendering methods. With Gaussian smoothing, bumps are clearly visible, with *screen-space curvature flow* the bumps are smoothed out, but the fluid looks unnatural. By adding surface noise, the surface gets a bit more realism. Figure 5.4 shows a close-up of the waterfall in the corridor scene, comparing the three rendering methods under somewhat more turbulent conditions. Figure 5.5 shows a close-up of fluid flowing out of a pipe for both our Curvature Flow method and the Gaussian smoothing approach. Figure 5.6 shows a comparison of our method and the Gaussian smoothing on another scene simulating a smooth green liquid inside a transparent container. We do not show an image with foam for this case as the fluid is too viscous to form foam.

Figure 5.7 shows the screen-space curvature flow process at work. On the left side it displays the rendered images, and on the right side a color-coded image of the curvature. Black is zero curvature, green is positive curvature and red is negative curvature. As the number of iterations increases, the curvature decreases, which can be seen as the curvature images become darker. The number of iterations can be freely chosen based on the desired smoothness.

Only the surface that is nearest to the camera is rendered. In most cases this is acceptable, because the thickness-based shading gives an illusion of volume to the fluid, but it is not entirely correct if there are multiple layers of fluid with air in between them.

5.5 Conclusions and future work

In this chapter we have presented a new method for rendering fluids in real-time directly from particle based representations without the need for intermediate triangulation, but which still produces a high-quality fluid surface. We have also introduced new ideas to add thickness-based shading and small-scale surface detail to fluids.

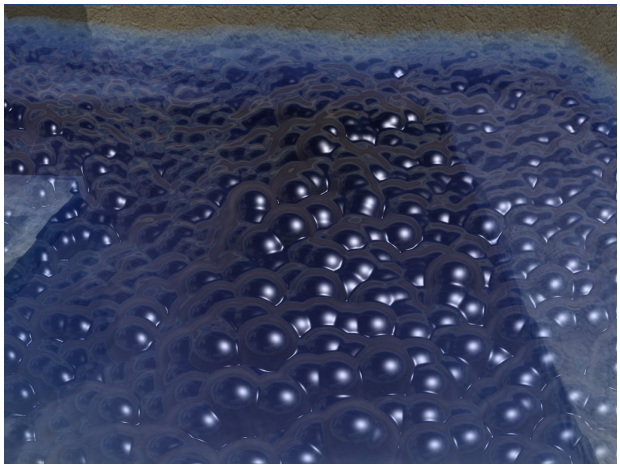
Table 5.1. *Performance comparison (in Frames Per Second) of* screen-space curvature flow *with different settings, to separable bilateral Gaussian blur (Corridor and NVIDIA logo)*

Corridor dataset	Frame (ms)
Bilateral Gaussian smoothing	18.1
Quarter res., 15 iterations	17.5
Half res., 40 iterations	19.6
Full res., 100 iterations	50.0
Foam+noise, quarter res, 15 iterations	30.0
Logo dataset	
Bilateral Gaussian smoothing	22.7
Quarter res., 15 iterations	23.3
Half res., 40 iterations	28.6
Full res., 100 iterations	50.0

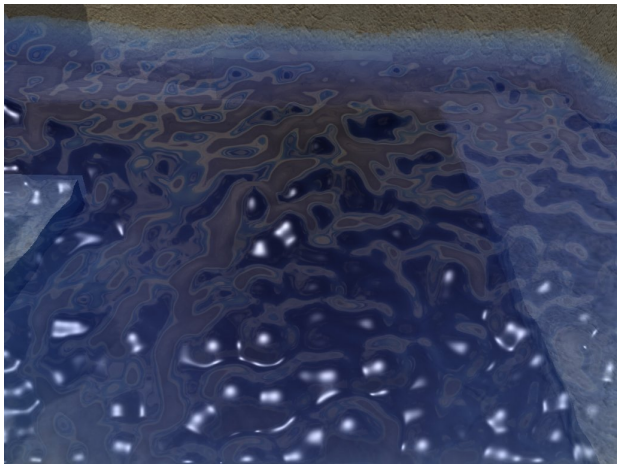
Table 5.2. *Performance degradation of the Bilateral Gaussian*

Method	Iterations	Frame (ms)
Curvature Flow	-	27.8
Bilateral Gaussian	1	25.6
Bilateral Gaussian	2	31.3
Bilateral Gaussian	4	38.5
Bilateral Gaussian	6	47.6

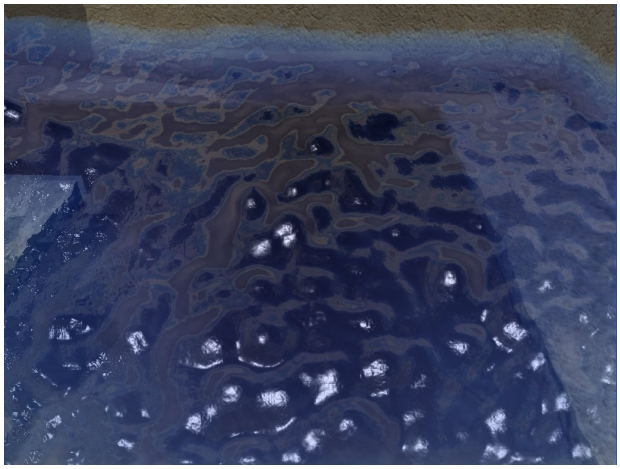
Future work may involve looking at using an implicit formulation of the integration scheme, as this would be more stable and require fewer time steps and thus improve performance. This might be difficult as the PDEs for curvature flow are quadratic, not linear. A semi-implicit [124] formulation of the curvature flow could also help. We would also like to investigate using CUDA or DirectX 11 Compute Shaders to improve the performance of the blur stage.



(a) Gaussian smoothing

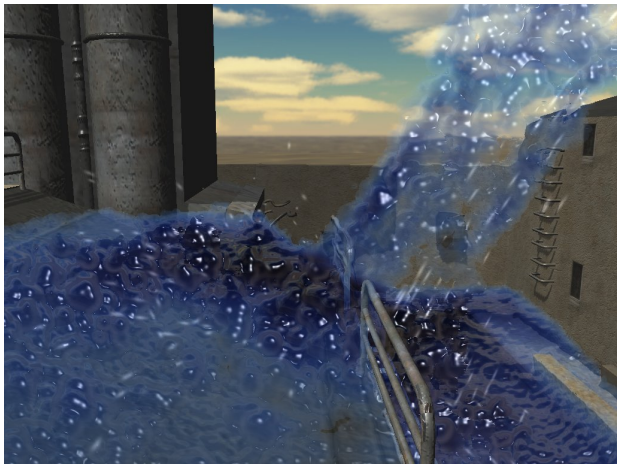


(b) Screen-space curvature flow

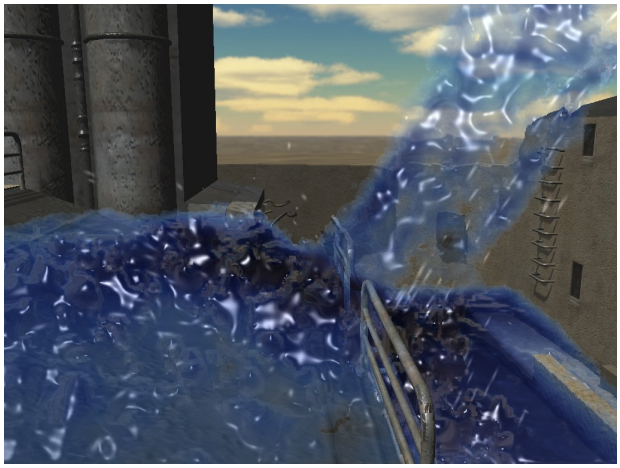


(c) With surface noise

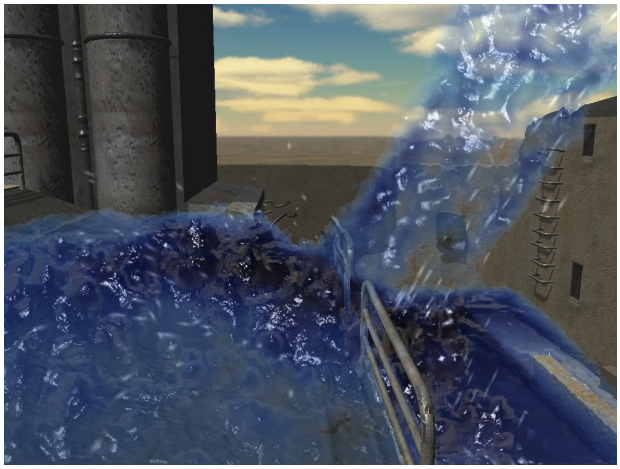
Figure 5.3. *Comparing Gaussian,* screen-space curvature flow *and surface noise for close-up view, with the smoothing computed in quarter resolution.*



(a) Gaussian smoothing

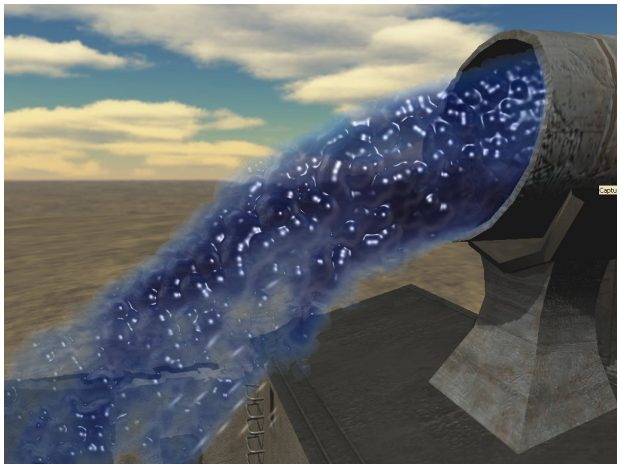


(b) Screen-space curvature flow

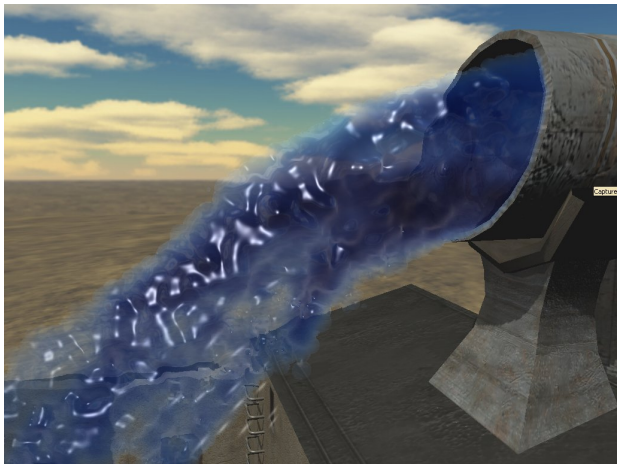


(c) Screen-space curvature flow with surface Perlin noise

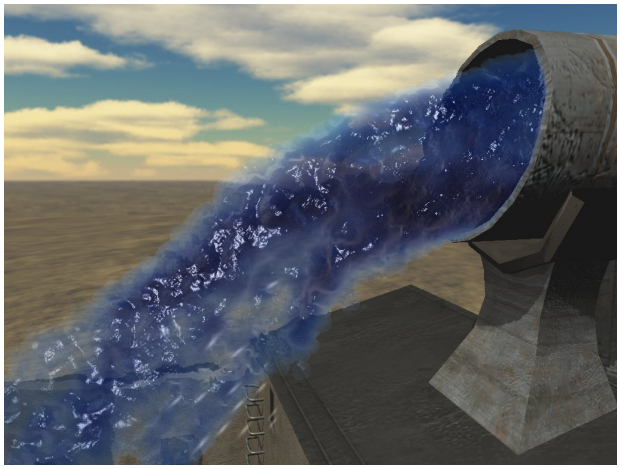
Figure 5.4. *Waterfall, comparing Gaussian and* screen-space curvature flow *with and without surface noise (smoothing computed in quarter resolution).*



(a) Gaussian smoothing

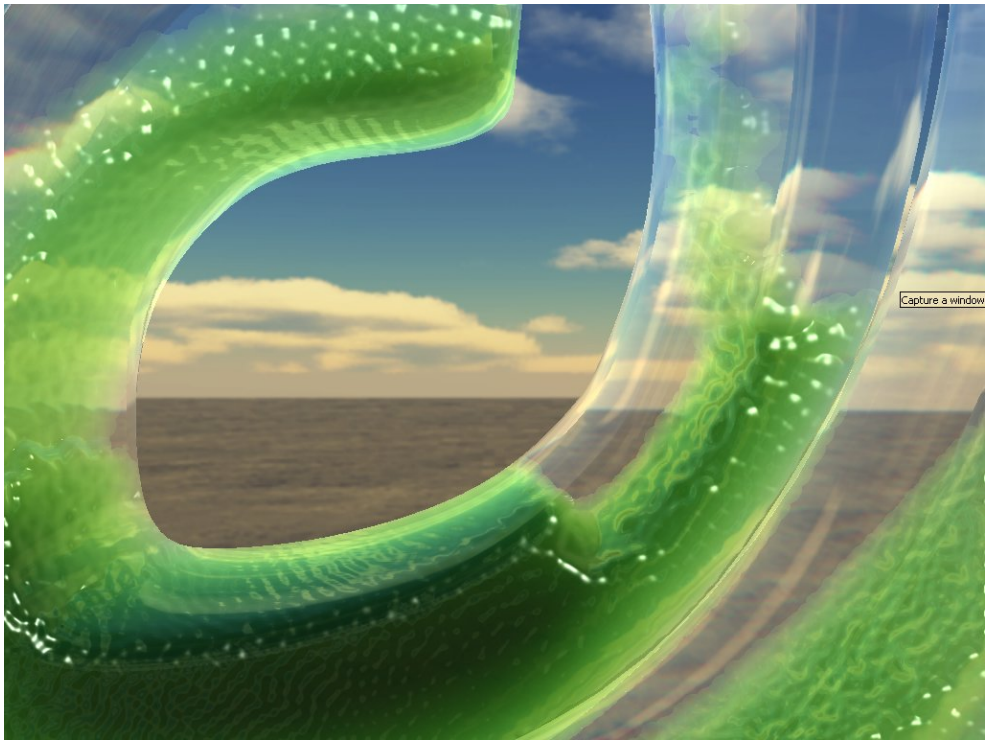


(b) Screen-space curvature flow

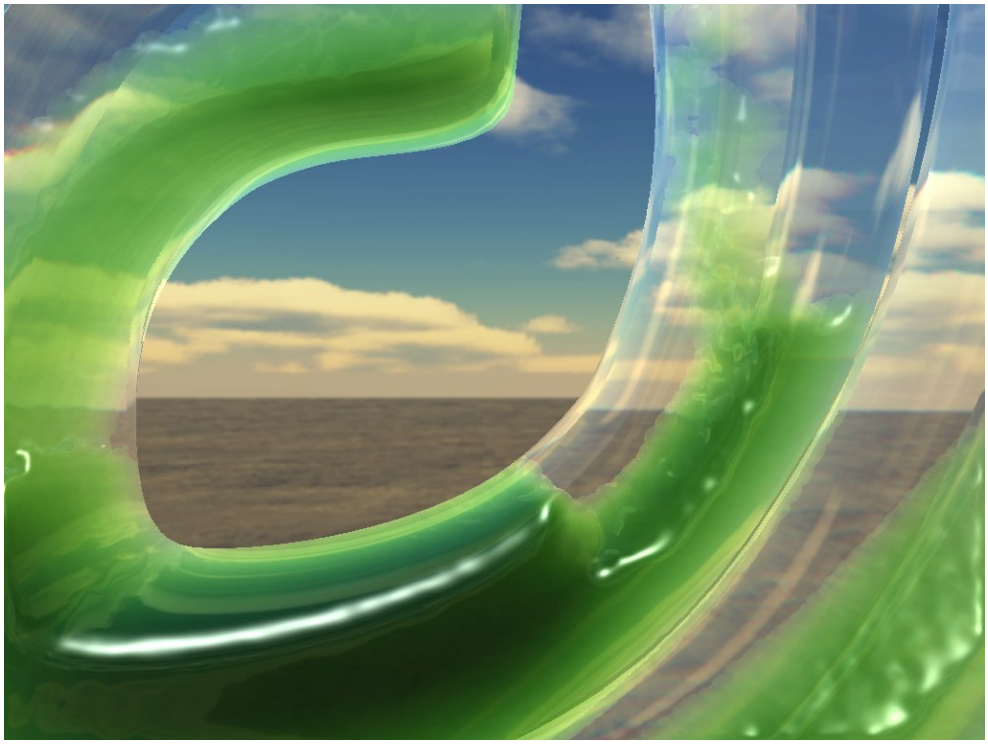


(c) Screen-space curvature flow with surface Perlin noise

Figure 5.5. *comparing Gaussian and* screen-space curvature flow *with and without surface noise (smoothing is computed at half instead of quarter resolution).*

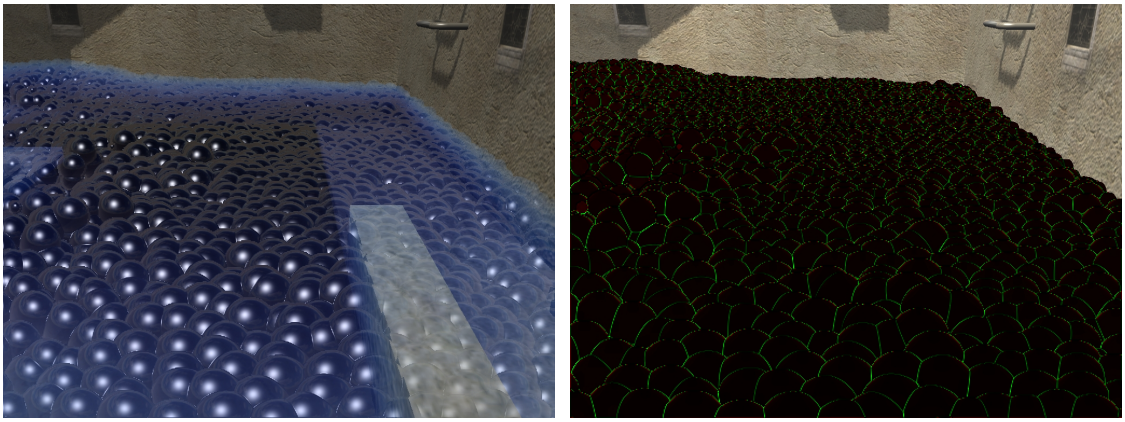


(a) Gaussian smoothing

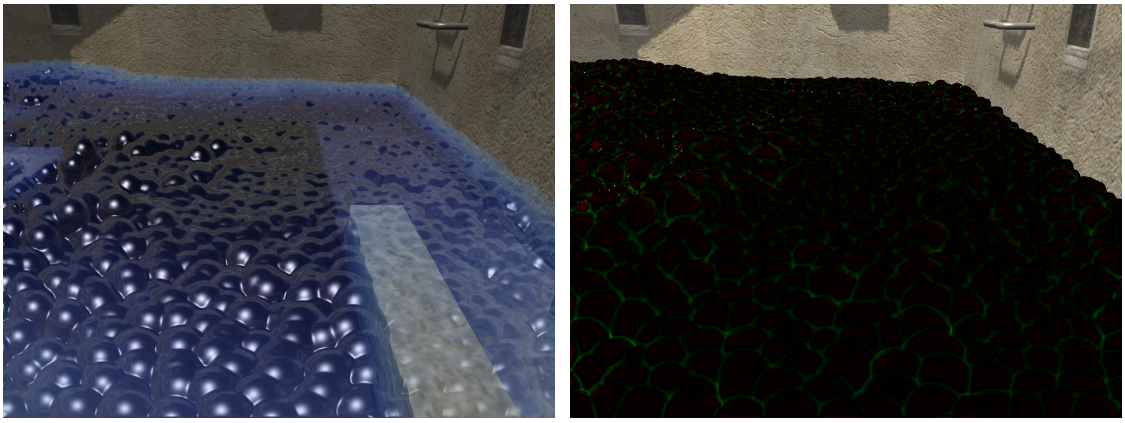


(b) Screen-space curvature flow

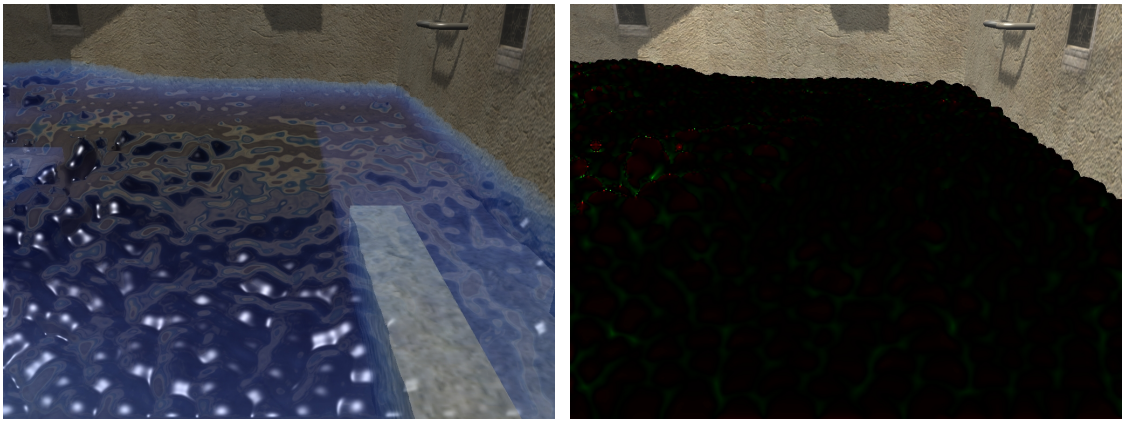
Figure 5.6. *comparing Gaussian (a) and* screen-space curvature flow *(b) on NVIDIA logo (quarter resolution)*



(a) 0 iterations, image (b) 0 iterations, curvature

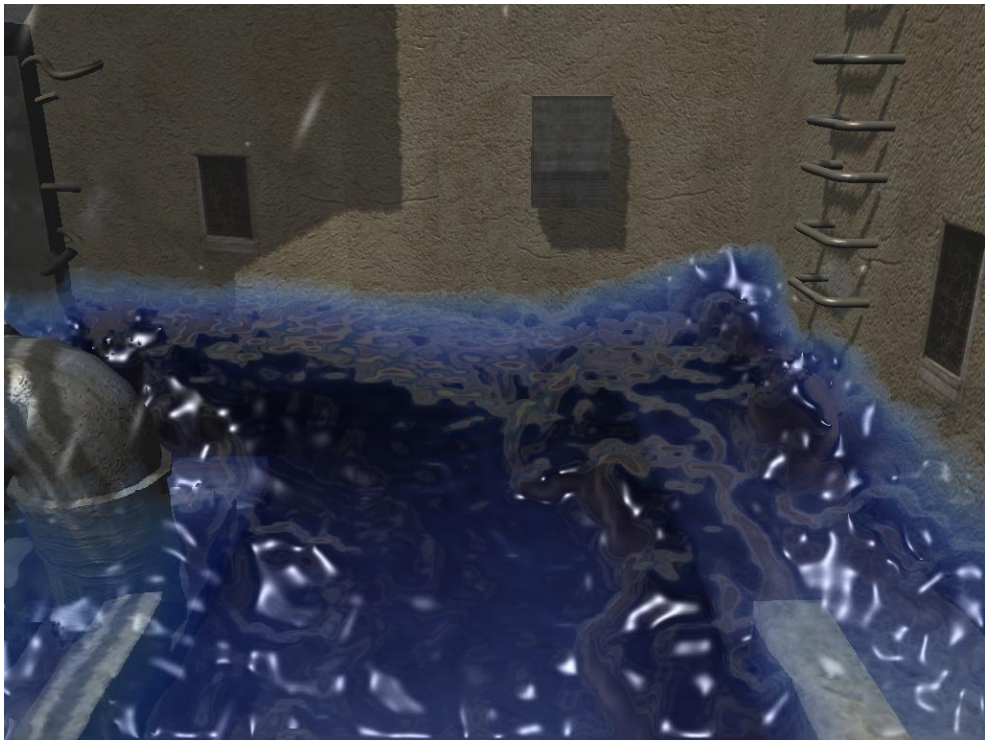


(c) 40 iterations, image (d) 40 iterations, curvature

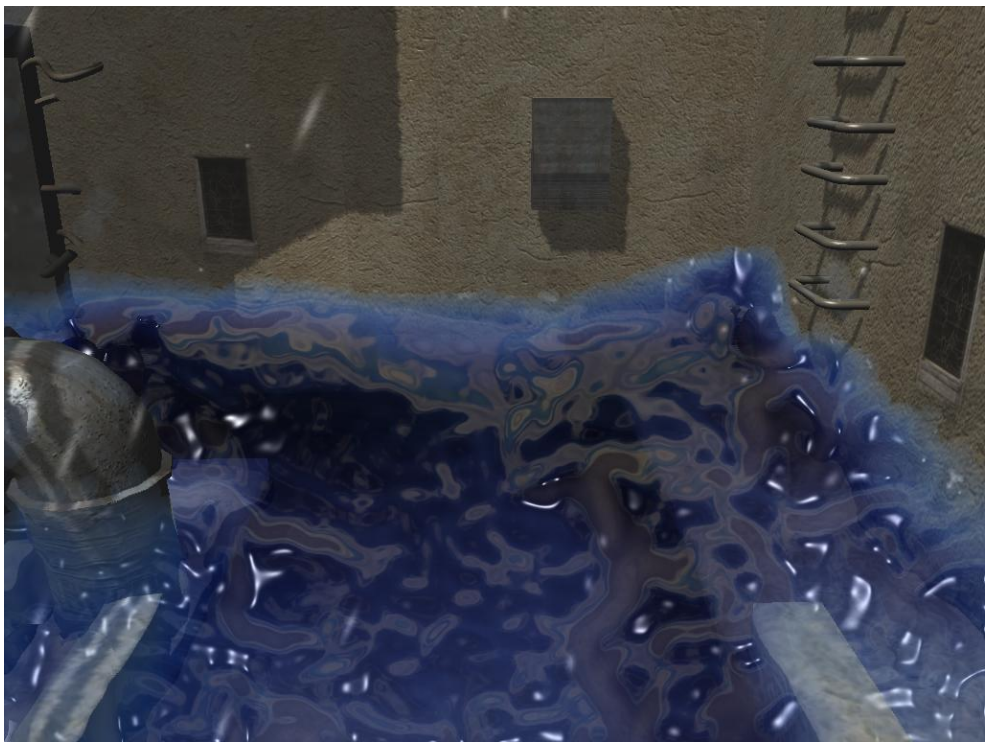


(e) 60 iterations, image (f) 60 iterations, curvature

Figure 5.7. *The screen-space curvature flow process. Left: rendered images, Right: color-coded curvature. As the number of iterations increases, the curvature decreases.*



(a) Curvature flow method



(b) Bilateral Gaussian

Figure 5.8. *Close-up of fluid using curvature flow and the equivalent visual results using 6 iterations of Bilateral Gaussian*

Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink. A Memory and Computation Efficient Sparse Level-Set Method. *Journal of Scientific Computing*, 2010, http://dx.doi.org/10.1007/s10915-010-9399-5, p. 1-22.

Chapter 6

A Memory and Computation Efficient Sparse Level-Set Method

6.1 Introduction

Since its introduction by Osher and Sethian [98], the level set method has become the method of choice for capturing and tracking moving interfaces. It has found applications in a wide variety of scientific fields, ranging from chemistry and physics to computer vision and graphics. For example, in computer vision, most state-of-the-art segmentation techniques are based on level sets to steer the evolving contour or surface towards the objects of interest [138].

The main idea of the level set method is to represent the dynamic interface (e.g., contour, surface, etc.) implicitly and embed it as the zero level set of a time-dependent, higher-dimensional function. Then, evolving the interface with a given velocity in the normal direction becomes equivalent to solving a time-dependent partial differential equation (PDE) for the embedding level set function. The main advantage of the level set method is that it allows the interface to undergo arbitrary topological changes, which is much more difficult to handle using explicit representations. The cost which has to be paid for the flexibility offered by the level set method is twofold. First, *computationally*, one has to solve the time-dependent level-set PDE in a higher-dimensional space than that of the embedded interface, and secondly, the *memory requirements* are higher than the size of the interface, as one needs to explicitly store a uniform Cartesian grid for solving the level set PDE. To address the computational issue, a number of techniques have been proposed, such as the so-called narrow-band schemes, see Section 6.2. Such methods rely on the fact that it suffices to solve the PDE only in the vicinity of the interface in order to preserve the embedding. Thus, the computational requirements scale with the size of the interface. However, most narrow-band methods require the (uniform) computational grid to be explicitly stored. As shown by Nielsen and Museth [89], hierarchical structures (e.g., based on octrees) only need to store points of the finest grid at the interface, while courser grids are used in the remaining part of the narrow band [33, 77, 84, 127]. However, these methods still require large amounts of memory, use complicated data structures, and additionally, increase computational requirements compared to narrow-band methods, as access to grid points is relatively slow.

In this chapter we present efficient data structures and algorithms for tracking dynamic interfaces through the level set method.

Our method, which we call Sorted Tile List, uses *tiles*, i.e., small, fixed-size blocks, to represent the narrow band around the interface. Instead of using a coarse grid of pointers to tiles that intersect the interface, our method implicitly locates the neighbours of each tile by maintaining an active list of tiles lexicographically sorted by coordinates. While several methods which address both computational and memory requirements have been very recently introduced [13, 54, 89], we show that our *sequential algorithm* is faster than these recent approaches and more importantly, that our algorithm can greatly benefit from both fine- and coarse-grain parallelization by leveraging SIMD and/or multi-core configurations.

The main contributions are:

- A highly efficient, tile-based, sparse-grid method for the level-set representation, which works in any number of dimensions.
- A fast, parallelizable and memory-efficient data structure that can be used to manage sparse grids of unbounded size.
- Fast, scalable algorithms for iterating and maintaining the proposed data structure, enabling efficient simulations based on level sets. More specifically, the temporal and spatial complexities of our data structure with the associated sequential algorithms are as follows:
  - *Storage requirements* are proportional to the number N of tiles intersecting the interface, it follows that the memory footprint of our method is *optimal* and scales with $O(N^{d-1})$, where $d > 1$ is the number of spatial dimensions.
  - *Sequential access* to grid points has time complexity $O(N)$.
  - *Access to neighbouring grid points* within the computational finite-difference stencil has time complexity $O(1)$.
  - *Random access* to grid points has complexity $O(\log N)$.
  - *Maintaining* the sparse data structure is linear, i.e., $O(N)$.

6.2 Previous and related work

As the level-set method has proven to be a very useful tool in any application requiring the tracking of moving interfaces, there has been continuous interest in developing efficient algorithms to address the large computational and memory requirements involved, see Section 6.1.

The computational issue was first addressed with the introduction of the so-called "narrow-band schemes" [2, 21]. The basic idea is to restrict the computations to a small vicinity around the zero level set used to represent the deforming interface. Whitaker [153] further improved the efficiency of this scheme, by performing calculations only at grid locations corresponding to the interface, resulting in a stencil only as wide as necessary for the finite-difference calculations at these locations. Peng *et al.* [99] also solved the level set PDE only in a narrow band around the interface, but used for its storage simple arrays as opposed to the more complex linked lists used in [153].

Whereas narrow-band methods effectively address the computational issue, they still need to explicitly store a full, regular Cartesian grid and additional data structures (e.g., arrays or lists) to pinpoint grid points belonging to the narrow band. Thus, such methods still have storage complexities scaling with the size of the grid. The first attempt to overcome this limitation was due to Strain [127], who used quadtree meshes to reduce memory requirements to the size of the interface, as opposed to the size of the grid. Improvements to the original quadtree method were later developed [33, 84], and an extension to surfaces by means of octree grids was recently introduced [77]. As pointed out by Nielsen and Museth [89], while tree-based methods achieve smaller memory footprints, they also have a number of drawbacks. Most notably, the non-uniform discretization of tree-based meshes makes it non-trivial to use high-order, finite difference schemes. Therefore, such methods use semi-Lagrangian schemes [127], which limit the class of problems which can be tackled to hyperbolic ones; see [89] for further details.

An interesting approach to reduce the memory requirements of the level set method was presented by Bridson in [13], dubbed the "Sparse Block Grid" (SBG) method. In 3D, this method divides the volume of size n^3 into small blocks of size m^3 voxels each. A coarse grid of size $(n/m)^3$ stores pointers to blocks that intersect the interface. Although this method has non-optimal storage complexity, it maintains constant access time similar to the full-grid method.

Recently, Nielsen and Museth introduced the *Dynamic Tubular Grid* (DT-Grid) method [89], a recursive, compressed level-set representation inspired by the compressed-row-storage technique used to represent sparse matrices. The authors showed that the memory requirement of DT-Grid is optimal, i.e., it is proportional to the size of the interface. Moreover, their experiments showed that the 3D DT-Grid is faster and more memory efficient than state-of-the-art octree-based approaches. Huston *et al.* [54] used hierarchical run-length encoding (RLE) in a dimensional-recursive fashion to encode the domain in a series of runs, each associated with a specific run code. Regions away from the narrow band are encoded to just their sign representation, while the narrow band is stored in full precision. Although this method is more flexible than DT-Grid [54], the price paid is a slight increase in computation time and memory usage.

Our method is similar to the SBG method [13], in that it uses tiles, i.e., small, fixed-size blocks, to represent the narrow band around the interface, see Fig. 6.2. Unlike SBG, our method does not use a coarse grid of pointers to tiles that intersect the interface. Instead, we propose an approach that can implicitly locate the neighbours of each tile, by maintaining a list of active tiles lexicographically sorted by coordinates. Thus, a requirement in every step of our method is to preserve the ordering of the active tiles, such that re-sorting them is not necessary. The proposed method also bears some similarities to the method of Lefohn *et al.* [68]. Similar to this approach, our method is tile-based and also uses gradient information from all elements of each active page to determine whether the page has still to be active during the next iteration. However, in contrast to this method, we do not need to store a map of the complete domain or to maintain a list of neighbours for each tile. Further, the complex paging mechanism of Lefohn's method is avoided altogether in our method, and updating the active list involves a simple, sequential traversal of the list. Moreover, our method is not bound to a fixed domain. Instead, it allocates and deallocates new tiles as the interface propagates to accommodate the deformations. To conclude, the main advantages of our tile-based approach are that the resulting method is highly efficient and straightforward.

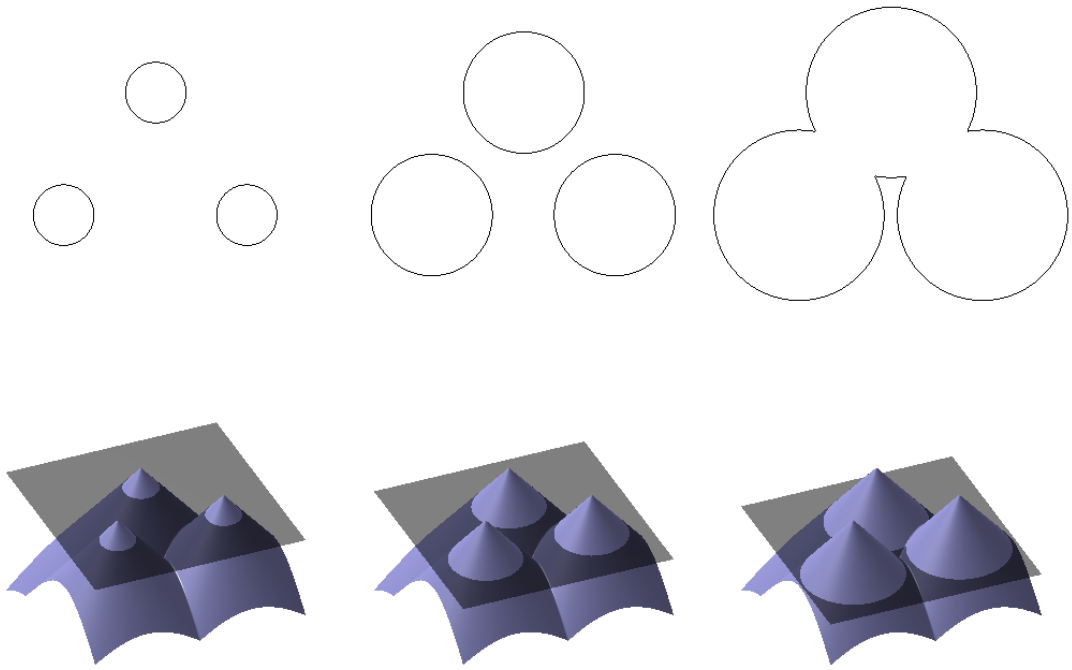


Figure 6.1. *Level-set example in 2D.* Top row*: evolving interface* $\Gamma(t)$. Bottom row*: corresponding graphs of* $\phi(\mathbf{x}, t)$ *at three different time steps;* left-to-right*: three initial contours expand at constant speed and eventually merge. The interface is given by the intersection of the plane* $z = 0$ *(indicated in dark grey) with the graph of* ϕ.

6.3 Overview of the level set method

In the level set method, a closed $(d-1)$-dimensional hyper-surface $\Gamma(t=0)$ is implicitly defined as the zero set of an d-dimensional Lipschitz continuous function $\phi(\mathbf{x}, t=0) : \mathbb{R}^d \to \mathbb{R}$, e.g., the distance to $\Gamma(t=0)$, with $\mathbf{x} \in \mathbb{R}^d$. Propagating $\Gamma(t)$ along its normal direction with speed v can be done by evolving the function ϕ defined as follows. Let $\phi(\mathbf{x}, t=0) = \pm\delta$, with δ the (signed) distance from $\mathbf{x}$ to $\Gamma(t=0)$. Thus, the set $S = \{\mathbf{x} \in \mathbb{R}^d \,|\, \phi(\mathbf{x}, t=0) = 0\}$ corresponds to the location of the embedded hyper-surface $\Gamma(t=0)$, see Fig. 6.1. Throughout the chapter we assume that the function ϕ has positive values outside the contour, and negative values inside.

With this notation, the equation for the function $\phi(\mathbf{x}, t)$ that represents the evolution of $\Gamma(t)$ is then [98]:

$$\frac{\partial \phi}{\partial t} = -v\,|\nabla \phi|\,. \tag{6.1}$$

If the speed function v is a positive constant, ϕ will shift up along the z-axis, so the contour will expand, see Fig. 6.1; if v is negative, the contour will shrink.

Intrinsic geometric properties of the evolving hyper-surface are easily determined from the level set function ϕ. In 3D for example, the mean curvature κ of each level set of ϕ is

$$\kappa = \frac{1}{2} \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|}. \tag{6.2}$$

For more details and applications of the level set method we refer to [95, 96, 98, 116, 138] and the references therein.

6.3.1 Sparse-grid level set representations

Considering that the goal is to track a moving interface represented by the zero level set of the embedding function ϕ, solving the PDE in Eq. (6.1) for ϕ on the entire domain would be inefficient, both in terms of memory and computation. Therefore, efficient algorithms for solving level-set equations perform the required computations only in a narrow band along the zero level set, see [2, 21, 99, 153]. Further, to minimize memory requirements sparse methods [13, 54, 89] have been introduced, see Section 6.2.

When a signed distance transform is used as the underlying function ϕ, the calculation can be restricted to a certain distance from the interface, by setting a range of values that we are interested in. The result can be clamped to the range $(-\gamma, \gamma)$, so that the function ϕ at a certain distance away from the interface has a constant value and a zero gradient magnitude, see [99]. This is illustrated in Fig. 6.2.

Similar to [13, 68], our method divides the domain into fixed-size tiles. Each tile represents a part of the domain of the function ϕ. Tiles that only contain locations with values outside the range $(-\gamma, \gamma)$ are considered irrelevant and thus are not computed or stored. Hence, the remaining tiles will necessarily contain the zero level set (or be very close to it); we call the collection of such tiles the *active set*.

The active set has to be constantly maintained while the simulation is running, so that the moving interface remains inside the active tiles. It follows that new tiles need to be dynamically created when the interface approaches a boundary of a tile, and that tiles can be deleted when they become irrelevant to the simulation.

6.3.2 Reshaping the level set function

Due to small numerical inaccuracies that build up over time when integrating the PDE in Eq. (6.1), the density of the level sets might not remain constant over the domain, i.e., ϕ is no longer the signed distance transform to the interface. If the density becomes too low, a sparse representation becomes inefficient, as the range $(-\gamma, \gamma)$ spreads over a wider band. On the other hand, if the density becomes too high, instabilities can result. To maintain a consistent level-set density, we make use of the *rescaling speed term* introduced in [68].

If ϕ is a signed distance transform, the gradient magnitude of the function will be unity over the entire active domain, i.e., $|\nabla \phi(\mathbf{x})| = 1$. We can set up a PDE of the form $\frac{\partial \phi}{\partial t} = \operatorname{sgn}(\phi)(1 - |\nabla \phi|)$, which constrains the level sets to have the desired density or gradient magnitude. When discretizing this PDE using a central difference approximation, instabilities arise.

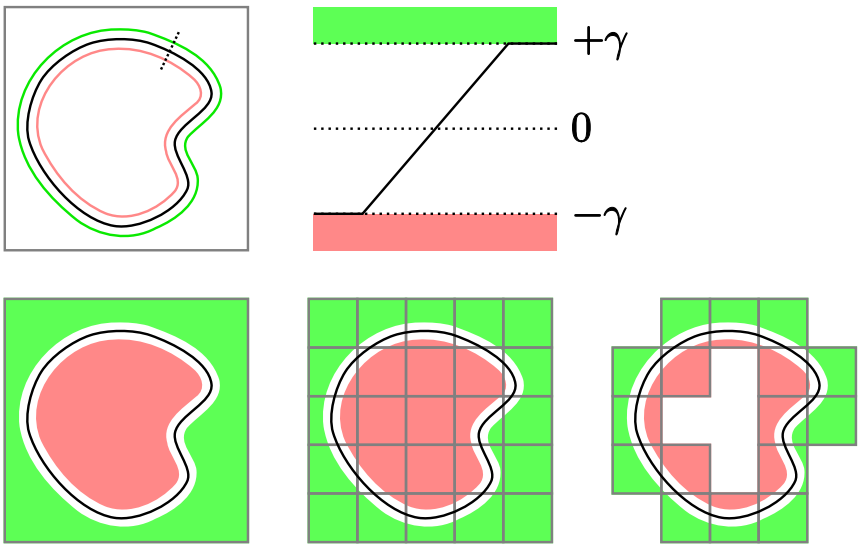


Figure 6.2. Top row*:* Left image - *current interface (black), and the* $-\gamma$ *and* γ *iso-contours inside/outside the black curve.* Right image - *profile of function* ϕ *along the dotted line in the first image.* Bottom row*:* Left image - *the domain of function* ϕ*, the* $\geq \gamma$ *area outside the interface and the* $\leq -\gamma$ *area inside the interface, with the curve in the middle.* Middle image - *the domain divided into tiles of equal size.* Right image - *the sparse domain, with inactive tiles (completely inside or outside) removed.*

This happens because, numerically, information propagates outward from the boundary. Therefore, as suggested in [117], the correct way of discretizing the spatial derivatives is by using an upwind scheme.

Further, using a smoother function to replace the step function in $\mathrm{sgn}(\cdot)$, stability can be improved. If the distance transform is bounded in a very narrow band, the function ϕ itself can be used instead of its sign, see [68].

6.4 The proposed method

In this section we present our memory and computationally-efficient method for evolving the level-set function. As explained in Section 6.3.1, our algorithm makes use of a sparse representation of the the level-set function ϕ, and uses a tile-based storage format that only keeps the function values in the vicinity of the zero level. As in other narrow-band methods, all other tiles are considered to be either inside or outside, but too far from the zero level set to have any influence on its evolution.

Our method keeps a list of active tiles, lexicographically sorted by coordinates. Moreover, all our processing steps maintain the list of active tiles sorted in lexicographical order, such that an expensive sorting step during each iteration is avoided. In the following section, the data structure will be explained in further detail.

Table 6.1. *Major operations acting on the data structure in the Sorted Tile List method. N is the number of tiles intersecting the interface.*

Algorithm	Time Complexity
Append	$O(1)$
Sequential Access With Stencil	$O(N)$
Random Access	$O(\log N)$
Tile Management	$O(N)$

6.4.1 The data structure

In this section we present the data structure that is central to our approach. The operations that can be applied are the same as those on a full grid, but differ in complexity. The theoretical complexities for various operations as derived in later sections are shown in Table 6.1.

Basically, the idea behind our structure is to have a list of tiles which are ordered lexicographically by coordinate. Additionally, some basic tile management functionality is needed to maintain the structure over time in an efficient way.

For each tile the following attributes are stored: the coordinates of the tile, a small cube of floating point data, and a set of flags that we will call the *border flags*. The border flags store the sign of the data outside the tile in every direction, in case the tile has no direct neighbour there.

In what follows, we shall call this list of tiles the *active list*. In the list an index or pointer to the actual data is stored. The alternative would be to store the data in the active list itself, but this would incur some extra copying in the management step. Also, in this way, the entire active list might fit into the cache, and it is easier to align the tile data to a larger power of two without wasting memory (given that the tile dimensions are powers of two).

In the following subsections, the initialization and other basic operations (e.g., append, sequential access within stencil, random access, tile management) associated with the data structure are presented.

6.4.2 Initialization

To build our structure from existing data, we set up a sorted list of active tiles containing the clamped distance transform to the interface $\Gamma(t = 0)$, see also Section 6.3. Depending on the input data it might be efficient to compute the distance transform on the whole domain, then delete inactive tiles, and finally, sort the resulting structure. Other methods, similar to that in [36] for converting a triangle mesh to a distance transform, already calculate a clamped distance transform close to the surface of the objects; such methods can be used directly.

The steps performed during the initialization stage are:

- Compute the signed distance transform to the input surface. Note that it is only necessary to compute this up to distance $\pm\gamma$. This results in a sparse volume, defined only near the interface.

- Divide the volume resulting from the first step into tiles, see Fig. 6.2.
- Add those tiles to the active list that contain parts of the active level set, i.e., locations where $-\gamma < \phi_{ijk} < \gamma$, with ϕ_{ijk} the level set function at position (i, j, k).
- For each tile that does not contain the interface, but shares at least one border with the tile set defined by the active list, update the appropriate border flag of the neighbouring tiles to signify that this tile is inside (filled with value $-\gamma$) or outside (filled with value γ).
- Sort the resulting *active list* in lexicographical order, if the list was not generated in this order.

6.4.3 Append operation

The append operation adds a tile to the end of the list. This is analogous to the 'push' operation in [89]. As the structure is lexicographically ordered by coordinates, the new tile must have a coordinate higher than that of the existing last tile. To add a tile, the new tile attributes are written to the end of the active list. Hence, adding one tile has complexity of order $O(1)$.

6.4.4 Sequential access with stencil

Sequential access can be simply achieved by moving a pointer over the list. However, as many operations also need access to neighbouring tiles adjacent to the tile that is being iterated, and since this requires some extra bookkeeping, we illustrate such a traversal of the active list in Algorithm 6.1. In this algorithm, `size` corresponds to the number N of tiles, `coord` contains the coordinates for each tile, `B` denotes the number of neighbours and `neighbourhood` denotes the relative coordinates of the neighbourhood tiles. Coordinates of tiles are compared lexicographically, and are added and subtracted component-wise like vectors. The user-defined function *iter* is called for each active tile, passing two parameters: `match`, a bit field signifying which neighbours are present and which are not, and `ptr`, the offset into the active list for each neighbour (only valid if the according bit in `match` is set). In 2D, an example of a 3^2 `neighbourhood` definition would be $((-1,-1),(-1,0),\ldots,(1,0),(1,1))$, and thus `B` would be 8. This extends trivially to three or more dimensions.

The complexity of this algorithm is linear in the number of tiles. This can be derived in the following way. There are `B` $+ 1$ pointers, `offset` and `ptr[0..B-1]`, that are initialized to the start of the active list, at the beginning of the algorithm. With each top-level iteration of the **while** loop, `offset` is incremented, and usually several of `ptr[0..B-1]` will also be incremented at least once, as these track a certain neighbourhood around the current tile with index `offset`. All pointers are incremented at most `size` times, after which they have reached the end of the active list. At the end of the algorithm, `offset` will always be equal to `size`. The other pointers will not necessarily have reached the end, so that the total number of pointer increments is smaller or equal to `(B` $+ 1$`)size`. As `B` is constant, the time complexity is thus linear in the number of tiles.

Algorithm 6.1 Iterating over the sorted tile list. The function *iter* is called for each tile.

Input: `size,coord[size],B,neighbourhood[B]`

```
 1: offset ← 0 {beginning of tile-set}
 2: for i = 0 to B − 1 do
 3:    ptr[i] ← 0
 4: end for
 5: while offset < size do
 6:    cur ← coord[offset] {take coord of current tile}
 7:    match ← 0 {bit field of neighbours that exist}
 8:    for i = 0 to B − 1 do
 9:       c ← cur + neighbourhood[i] {calculate coord of neighbour i}
10:       while ptr[i] < size and coord[ptr[i]] < c do
11:          ptr[i] ← ptr[i] + 1 {track neighbour}
12:       end while
13:       if ptr[i] < size and coord[ptr[i]] = c then
14:          match ← match | 2^i {if coord matches, neighbour i exists}
15:       end if
16:    end for
17:    iter (match, ptr) {call iterator}
18:    offset ← offset + 1 {advance to next tile}
19: end while
```

The last tile iteration in a time step should write its resulting data blocks in active list order, consecutively, and thus remove deleted blocks without the need for an extra iteration over the data. This ensures that the tile data is always at consecutive memory locations, and ordered in the same (lexicographical) order as the active list. Moreover, this also means that during the next tile management step, free tiles can easily be allocated with a simple counter starting from the end of the data, without the need for more complex memory management strategies.

6.4.5 Random access

As the *active list* is an ordered list of tile coordinates, an efficient method to look for a random tile is to do a binary search, comparing at each step the desired coordinate to the coordinate at the current position. Thus, the complexity of a random access operation is $O(\log N)$.

6.4.6 Tile management

In the tile management step, the list of active tiles is updated as follows. Tiles that are no longer close to the zero level set are removed, and tiles bordering the zero level set that are needed in the next time-step will be added. For each currently-active tile, it is first determined which of the neighbouring tiles are needed in the next time step. If the interface approaches a tile border, the tile at the other side of that border has to be present in the next time-step to continue the computation. The borders of a tile that are being approached by the evolving contour, are signalled

through a set of activity flags by the time-stepping computation (as explained in Section 6.4.7). If the activity flag for a certain border is set, a tile has to be created if it is not yet present in the direction of that flag. If the interface has just left a certain tile, all the activity flags for that tile will be zero. If none of the neighbouring tiles requests for the tile to be retained, it can be safely removed to free memory.

The basic idea then is to iterate over the list of tiles, and for each tile to expand the tile set by creating tiles in the directions whose activity flags are set. Along the way, one has to remember which tiles are inactive and not requested by any other neighbouring tile. Such tiles will be discarded.

A straightforward implementation of this idea would create tiles multiple times when they are requested by different tiles, resulting in tile duplicates. It helps to look at this in another way: as new tiles will always be direct neighbours of the current tile-set, the process can be seen as a morphological dilation of the set of active tiles by a 3^3 structuring element [114]. For each element in the dilated version of the set, it should then be determined whether to create, remove or keep the tile at that position. This assures that new tiles will only be created at most one time.

Dilation of a sorted list

Algorithm 6.2 shows the main steps needed for iterating over a sorted tile list, while dilating it on the fly. Here `size` is the number of tiles, and `coord` is a sorted list of coordinate vectors for each tile. The variables `B` and `neighbourhood` define the size and coordinate tuples of the structuring element. Here, in contrast to the algorithm presented in Section 6.4.4, `neighbourhood` must include the middle tile $(0, 0)$. The variable `maxcoord` tells where to stop iterating: a value of (∞, ∞) indicates that iteration should proceed until the entire tile-set is dilated. The user-defined function *iter* is called for each tile in the dilated version of the list, passing three parameters: `cur`, the current coordinate, `match`, a bit field signifying which neighbours are present, and `ptr`, the offset into the tile list for each neighbour (only valid if the according bit in `match` is set).

The algorithm works by moving the structuring element over the tile-set in lexicographical order, continuously selecting the first tile that is intersected by it, see Fig. 6.3. In the beginning, all pointers are initialized to 0 (the beginning of the active list). Then, the relative position of the first tile that intersects the structuring element is chosen, and its position is set as the current one (lines 5-11). Next, the algorithm checks if it has reached the end of the data (line 12). If not, it builds a bit field of all the neighbours that are present (lines 15-20) and calls the iterator (line 21). At the end of the loop, all the pointers that point to matched neighbours are increased (line 22-26). Clearly, the complexity of this algorithm is linear in the number of tiles, provided that the complexity of the iteration function *iter* is constant.

Tile update

Now that the algorithm for iterating over the sorted and dilated list of tiles is established, we use it to do the actual tile management (function *iter*) shown in Algorithm 6.3.

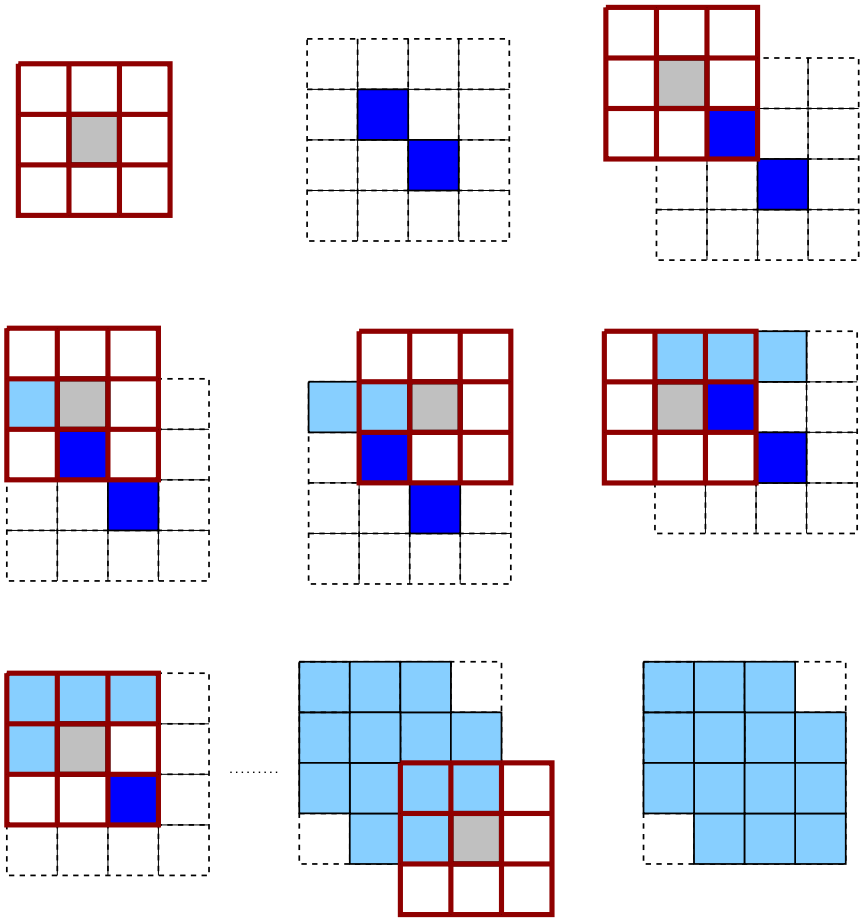


Figure 6.3. *A* 3×3 *structuring element is moved over the tile-set in lexicographical order, dilating the set in a tile-by-tile fashion.* Top-left*: structuring element,* center*: original set consisting of two active tiles,* left-to-right, top-to-bottom*: the structuring element starts at the first possible position, and shifts over the set, until the end result is obtained (bottom right).*

The goal of this iterator is to determine which tiles to keep, which to add and which to remove. For this purpose, the active flags of the neighbours are examined (lines 1-6). Here `neighbour_trigger` is a list of bit fields that signify when a tile needs to be retained or created, given the activity flags of the neighbours and the tile itself. We want to "trigger" a tile when there is activity in the tile itself, or if the border facing that tile or one of the surrounding tiles is active; this is illustrated in Fig. 6.4. There are two different cases (line 7): either this is a currently active tile (Section 6.4.6), or this is a possibly new tile (Section 6.4.6). This is determined by checking if the middle (current) tile exists. Here `MID` is the offset of the middle tile $(0, 0)$ in the *neighbours* list.

Algorithm 6.2 Iterating over a dilated version of a lexicographically-sorted list of tile coordinates, maintaining the sorted order. Function *iter* is called for each tile.

Input: `size,coord[size],B,neighbourhood[B],maxcoord`

```
 1: for i = 0 to B − 1 do
 2:    ptr[i] ← 0 {beginning of tile-set}
 3: end for
 4: loop
 5:    cur ← maxcoord {init to max. coord.}
 6:    for i = 0 to B − 1 do {loop over structuring element}
 7:       if ptr[i] < size then
 8:          {remember first tile in lexicographic order}
 9:          cur ← min(cur, coord[ptr[i]] − neighbourhood[i])
10:       end if
11:    end for
12:    if cur = maxcoord then
13:       break {end of tile-set reached}
14:    end if
15:    match ← 0 {bit field of neighbours that exist}
16:    for i = 0 to B − 1 do
17:       if ptr[i] < size and coord[ptr[i]] = (cur + neighbourhood[i]) then
18:          match ← match | 2^i {if coord matches, neighbour i exists}
19:       end if
20:    end for
21:    iter (cur, match, ptr) {call the iterator}
22:    for i = 0 to B − 1 do
23:       if match & 2^i then
24:          ptr[i] ← ptr[i] + 1 {advance structuring element}
25:       end if
26:    end for
27: end loop
```

Existing tile

If the tile is a current tile, we determine whether we need to keep it, by looking at the value of `trigger` (line 9). If it is set, we keep the tile and just append it to the new active list for the next time-step, using activelist_append(coord,tileid) (line 10). Otherwise, it is not added to this list, and will not be active during the next time step. Another step that needs to be taken when a tile is removed, is to notify the neighbours whether the removed tile was inside or outside by appropriately setting their border flags. First, we fetch the classification (inside or outside) using `classify(x)`, which extracts the bit in activity flags `x` that classifies the tile as either inside or outside (line 12), see Section 6.4.7. We then set the border flag using `setborderflag(p,q,v)` which sets the value of border flag `q` of tile `p` to `v`, see Section 6.4.1. Here `reverse(p)` is a function that reverses a border direction. This is needed to set the flag of the border facing the current tile. In 3D, this is simply defined as $27 - \mathtt{p}$ (line

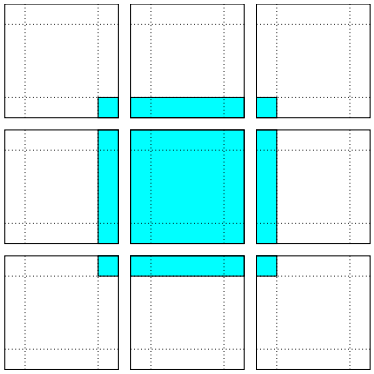


Figure 6.4. *The set* neighbour_trigger*, in the 2D case. The current tile and its 8 neighbours are shown, each divided into 9 areas. The marked areas are included in the set.*

16).

New tile

For a tile that we are considering to create, we first look at the value of `trigger` (line 21). If this tile was triggered, then it must be created, otherwise nothing is done. Next, we check the border flags of the surrounding neighbours using `borderflag(p,q)`, which computes the value of border flag `q` of tile `p`, to determine how to initialize the new tile (lines 22-27). Then we allocate a new tile using `allocate_tile(flag)`, which gives us the next free tile offset, and fills it with values $-\gamma$ or γ depending on `flag`. The border flags of the new tile are initialized (lines 29-31), and finally, the tile is appended to the active list for the next time-step (line 32).

Note that, as the complexity of Algorithm 6.2 is linear in the number of tiles, and Algorithm 6.3 executes a constant number of steps, the overall complexity of the tile management stage of the proposed method is linear in the number of tiles.

6.4.7 Updating the level-set

The time-integration step evolves the level set function ϕ according to the generic level-set PDE in Eq. (6.1). To do this, we make use of the *sequential access with stencil* algorithm described in Section 6.4.4.

For the sake of clarity, we explain the time-integration step using our method for a specific level set PDE (see Eq. (6.3)), using forward first order differences in time, and first and second order differences in space with a 3^3 stencil. However, we also implemented fifth-order accurate HJ-WENO discretization for spatial hyperbolic terms (see Section 6.5) and third-order accurate TVD Runge-Kutta for time integration, the details of which can be found in, e.g., [74] and [122]. Note that larger stencils can easily be accommodated by our method if the application at hand requires it, by increasing the tile size.

The level-set PDE we are here concerned with, consists of three terms: a data-dependent speed term, a surface-area minimizing term (based on the mean curvature of the surface, see

Algorithm 6.3 *iter* function for tile management.

Input: `cur`, `match`, `ptr`, `B`, `neighbourhood[B]`, `neighbour_trigger[B]`, `activeflags[]`, `index[]`

```
1: trigger ← 0 {default to 0, unless triggered}
2: for i = 0 to B − 1 do
3:   if match & 2^i and (activeflags[ptr[i]] & neighbour_trigger[i]) then
4:     trigger ← 1 {neighbour i has contour approaching this tile}
5:   end if
6: end for
7: if match & 2^MID then {current tile already exists}
8:   tileid ← index[ptr[MID]]
9:   if trigger then {if triggered, keep tile}
10:    activelist_append(cur, tileid)
11:  else {otherwise, remove it}
12:    flag ← classify(ptr[i]) {classify as inside or outside}
13:    for i = 0 to B − 1 do
14:      if match & 2^i then
15:        {notify neighbours of removal}
16:        setborderflag(index[ptr[i]], reverse(i), flag)
17:      end if
18:    end for
19:  end if
20: else {current tile doesn't exist yet}
21:  if trigger then {if triggered, create tile}
22:    flag ← 0 {determine if new tile should be inside or outside}
23:    for i = 0 to B − 1 do
24:      if match & 2^i then
25:        flag ← flag | borderflag(index[ptr[p]], reverse(p))
26:      end if
27:    end for
28:    tileid ← allocate_tile(flag) {create and init. new tile}
29:    for i = 0 to B − 1 do {init. border flags}
30:      setborderflag(tileid, i, flag)
31:    end for
32:    activelist_append(cur, tileid)
33:  end if
34: end if
```

Eq. (6.2)), and a rescaling-speed term (see Section 6.3.2) to keep the level set as closely as possible to a signed distance transform, i.e.,

$$\frac{\partial \phi}{\partial t} = -F(\mathbf{x}, t)\,|\nabla\phi| + \epsilon\kappa\,|\nabla\phi| + \operatorname{sgn}(\phi)\,(1 - |\nabla\phi|)\,. \tag{6.3}$$

The data-dependent speed term F can vary in both position and time. The strength of the curvature-based term (the second term) is determined by a constant ϵ, which is usually fixed.

The rescaling-speed term completely depends on the $\mathrm{sgn}(\cdot)$ function used, which otherwise is free of parameters, see Section 6.3.2.

When discretizing this equation, the first and third terms must be approximated using an upwind scheme, which satisfies the entropy condition [117]. The curvature-based term can simply be approximated using central differences. Initially, the first order approximations to the spatial derivatives of the level set function ϕ at each position (i, j, k) in the tiles are calculated: central differences D^{0x}_{ijk}, D^{0y}_{ijk}, D^{0z}_{ijk}, backward differences D^{-x}_{ijk},D^{-y}_{ijk}, D^{-z}_{ijk} and forward differences D^{+x}_{ijk},D^{+y}_{ijk}, D^{+z}_{ijk}. Central differences are necessary for the computation of the curvature term, whereas backward and forward ones are needed for the upwind scheme. For evaluating the curvature we also need to compute central difference approximations to all second order spatial derivatives, D^{0xx}_{ijk}, D^{0xy}_{ijk}, $\ldots$, D^{0zz}_{ijk}.

To compute the curvature we start from Eq. (6.2). The gradient of ϕ is approximated by central differences, i.e., $\nabla\phi \approx (D^{0x}_{ijk}, D^{0y}_{ijk}, D^{0z}_{ijk})$. The central difference approximation of the gradient magnitude is denoted by ∇^0, i.e.,

$$\nabla^0 = \left({D^{0x}_{ijk}}^2 + {D^{0y}_{ijk}}^2 + {D^{0z}_{ijk}}^2 \right)^{\frac{1}{2}}$$

By approximating all first and second order derivatives in Eq. (6.2) one finds the following approximation for the curvature κ:

$$\kappa \approx \frac{1}{2} \sum_{\alpha=x,y,z} \left(\frac{D^{0\alpha\alpha}_{ijk}}{\nabla^0} - \frac{D^{0\alpha}_{ijk}}{(\nabla^0)^3} \sum_{\beta=x,y,z} D^{0\beta}_{ijk} D^{0\beta\alpha}_{ijk} \right). \tag{6.4}$$

Three approximations of the gradient magnitude need to be computed: the central difference approximation ∇^0 as defined above, and two other ones for the upwind scheme [117], i.e.,

$$\begin{aligned} \nabla^+ = \Big[& \max^2\left(D^{-x}_{ijk}, 0\right) + \min^2\left(D^{+x}_{ijk}, 0\right) + \max^2\left(D^{-y}_{ijk}, 0\right) \\ & + \min^2\left(D^{+y}_{ijk}, 0\right) + \max^2\left(D^{-z}_{ijk}, 0\right) + \min^2\left(D^{+z}_{ijk}, 0\right) \Big]^{\frac{1}{2}}, \end{aligned}$$

$$\begin{aligned} \nabla^- = \Big[& \max^2\left(D^{+x}_{ijk}, 0\right) + \min^2\left(D^{-x}_{ijk}, 0\right) + \max^2\left(D^{+y}_{ijk}, 0\right) \\ & + \min^2\left(D^{-y}_{ijk}, 0\right) + \max^2\left(D^{+z}_{ijk}, 0\right) + \min^2\left(D^{-z}_{ijk}, 0\right) \Big]^{\frac{1}{2}}. \end{aligned}$$

All calculations above need a 3^3 stencil centered around the current element. At the borders of a tile, values from neighbouring tiles are required. These values can be fetched using the neighbour pointers provided by the algorithm in Section 6.4.4. If a neighbour tile is not active at the moment, a dummy tile filled with either $-\gamma$ (inside) or γ (outside) is used, depending on the border flag in the direction of the required tile.

The value at the current position for the next time-step is then calculated using Euler integration in time, i.e.,

$$\phi^{n+1}_{ijk} = \phi^n_{ijk} + \Delta t\, S,$$

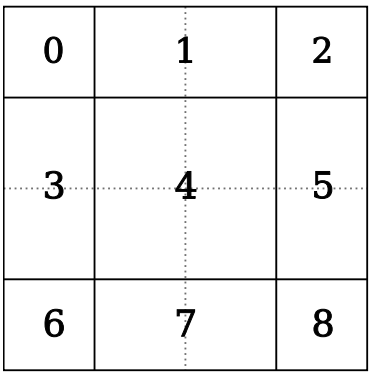


Figure 6.5. Guard bands *in the time-step iteration guarding the borders of a* 4×4 *tile, used to set activity flags for corners* $0, 2, 6, 8$*; borders* $1, 3, 5, 7$*; and center* 4.

in which Δt is the time-step and S is defined as the following sum of speed terms:

$$\begin{aligned} S = &- \left(\max\left(F_{ijk}, 0\right)\nabla^{+} + \min\left(F_{ijk}, 0\right)\nabla^{-}\right) \\ &+ \left(\epsilon\,\kappa_{ijk}^{n}\nabla^{0}\right) \\ &+ \max\left(\operatorname{sgn}\left(\phi_{ijk}^{n}\right), 0\right)\left(1 - \nabla^{+}\right) \\ &+ \min\left(\operatorname{sgn}\left(\phi_{ijk}^{n}\right), 0\right)\left(1 - \nabla^{-}\right), \end{aligned}$$

where the sign of F_{ijk} determines whether the motion is inward or outward w.r.t. the interface, and κ_{ijk}^{n} is the central difference approximation to the mean curvature at the current position.

During these computations, the old set of tiles from the previous time-step needs to be read, while the values for the new time-step have to be written. To accomplish this, we temporarily need to double the memory requirements for each tile. After the values for the next time-step have been computed for all tiles, the old values are discarded, and the whole process is repeated.

Activity flags

While calculating the level set function for each element in the tile, an activity flag for each border, corner and central part is maintained using guard bands (see Fig. 6.5). This flag is set if the result in that band is between the two cut-off values, that is $-\gamma < \phi_{ijk} < \gamma$. The stored flags are used in the tile management step (Section 6.4.6), which determines which tiles need to be activated and which can be deactivated.

In case that all the resulting activity flags are zero, meaning that there is no activity in a tile at all, the tile can be classified as entirely inside ($\phi_{ijk} = -\gamma$) or entirely outside ($\phi_{ijk} = \gamma$). This classification is stored with the activity flags, and used to update the border flags of neighbouring tiles when the tile is removed.

6.5 Results

In this section we present both numerical and performance-related results that show the strengths and weaknesses of our new approach, compared to previous methods.

In [89], the DT-Grid method was already compared to other state-of-the-art level-set data structures, and it was shown that this method achieves the highest performance. Therefore, to measure the relative performance of our new method, we implemented the DT-Grid method [89] and used it as a reference. In all comparisons in this section, the same level-set simulation code was used for the computational routines which are common to both our method and DT-Grid, so as to fairly compare the performance of both representations. Only in the indicated cases, our method was additionally optimized using Intel SIMD instructions (SSE2) which compute four floating point values at once, in a single operation. The machine used for benchmarking was an Intel Core 2 Quad 2.4 GHz, running a recent 32-bit Linux kernel; the compiler used was GNU g++ 4.2.4.

6.5.1 Mean curvature flow

In the first experiment, we simulated a sphere collapsing under mean curvature flow. The simulation was performed on a 256^3 grid. The simple discretizations from subsection 6.4.7 were used. The number of time-steps it took for the sphere to collapse to a point and then disappear was equal for both the DT-Grid and our method, confirming that the evolution of the sphere was the same. The complete simulation took 2674 seconds using the DT-Grid and 429 seconds using our method, thus resulting in a speed-up factor of about 6.2.

6.5.2 Volume-conserving mean curvature flow

Next we benchmarked our algorithm by performing the simulation of an extruded spiral evolving under volume-conserving mean curvature flow [99], see Fig. 6.6. The PDE describing the evolution is

$$\frac{\partial \phi}{\partial t} = (\kappa - \overline{\kappa})|\nabla \phi|, \tag{6.5}$$

where κ is the mean curvature of the level sets in $(-\gamma, \gamma)$ and $\overline{\kappa}$ is the average curvature of the interface. To approximate Eq. (6.5), second-order central differences (see subsection 6.4.7 and Eq. (6.4)) were used for the parabolic term ($\kappa|\nabla\phi|$) and the fifth-order accurate HJ-WENO scheme [94] for the spatial hyperbolic term ($\overline{\kappa}|\nabla\phi|$). The average curvature of the interface $\overline{\kappa}$ is computed by

$$\overline{\kappa} = \frac{\int_\Gamma \kappa\, \delta(\phi)|\nabla\phi|\, dx}{\int_\Gamma \delta(\phi)|\nabla\phi|\, dx}, \tag{6.6}$$

where the delta function is approximated by [99]

$$\delta(\phi) = \begin{cases} \frac{1}{2\epsilon}(1 + \cos(\frac{\pi\phi}{\epsilon})) & \text{if } \phi < \epsilon \\ 0 & \text{otherwise} \end{cases}, \tag{6.7}$$

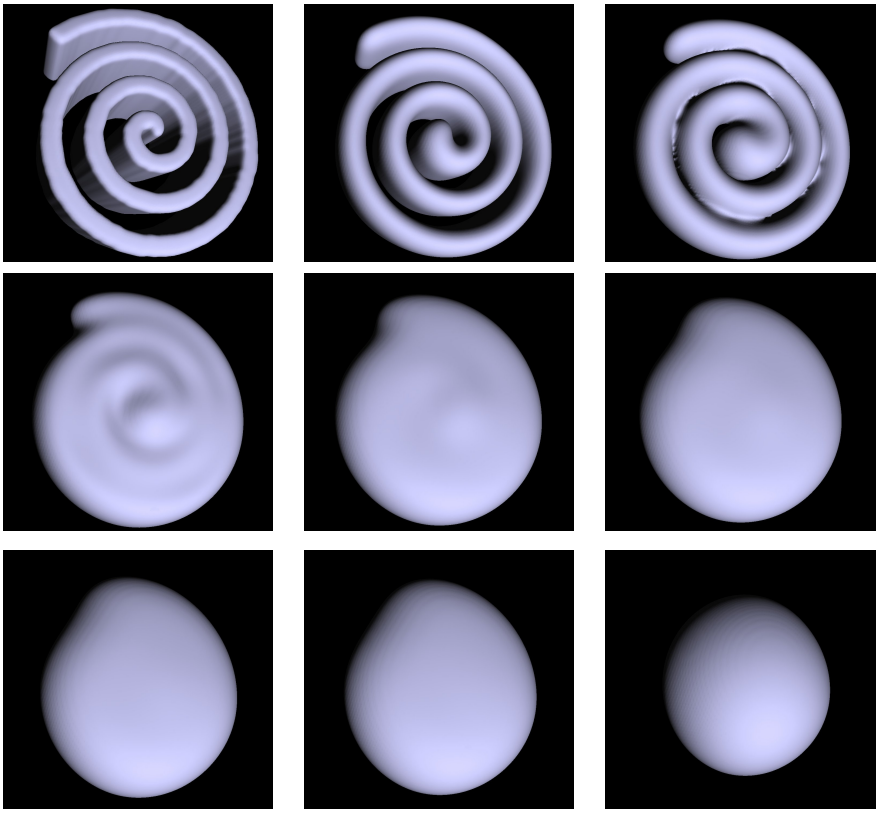


Figure 6.6. *Left-to-right, top-to-bottom: snapshots of the volume-conserving mean curvature flow simulation by the Sorted Tile List method, taken every* 100 *time steps. The final image shows the result after* 1700 *time steps.*

where ϵ is a parameter proportional to the grid spacing. Finally, a third-order accurate TVD Runge-Kutta time-stepping method [122] was employed. The simulation was performed on a 128^3 grid, and the spiral initially had a radius of 50 and a depth of 40.

In the SSE-optimized version of our method, we used aligned loads to fetch stencil values from the tile set and perform all stencil computations on four values at once. The tile-based memory storage format of our method allows for this vectorization to be done easily: the only change in the computation code was to replace the occurrences of *float* with *vec4*, an overloaded type that represents four floats at once. These optimizations are not feasible for DT-Grid, as the extra work spent in collecting stencil values from the voxel-centric storage and storing them into aligned memory areas would outweigh the gain of computing four floating point values at once.

Timings for the first 1000 iterations are shown in Table 6.2. It can be seen that our method is about three times faster than the DT-Grid method, and the SSE-optimized version of our method is about 7.2times faster. Figure 6.7 shows the computation time per time step, for DT-Grid, our method, and our SSE-optimized method. As can be seen, the time usage of our method is almost constant, as voxels are grouped in tiles and only full tiles are switched on and off for

Table 6.2. *Performance comparison of level-set data structures.*

Method	Time (s)
DT-Grid	680.0
Sorted Tile List method (tile size 4^3)	240.0
Sorted Tile List method (tile size 3^3)	230.0
Sorted Tile List method (SSE2, tile size 4^3)	80.1

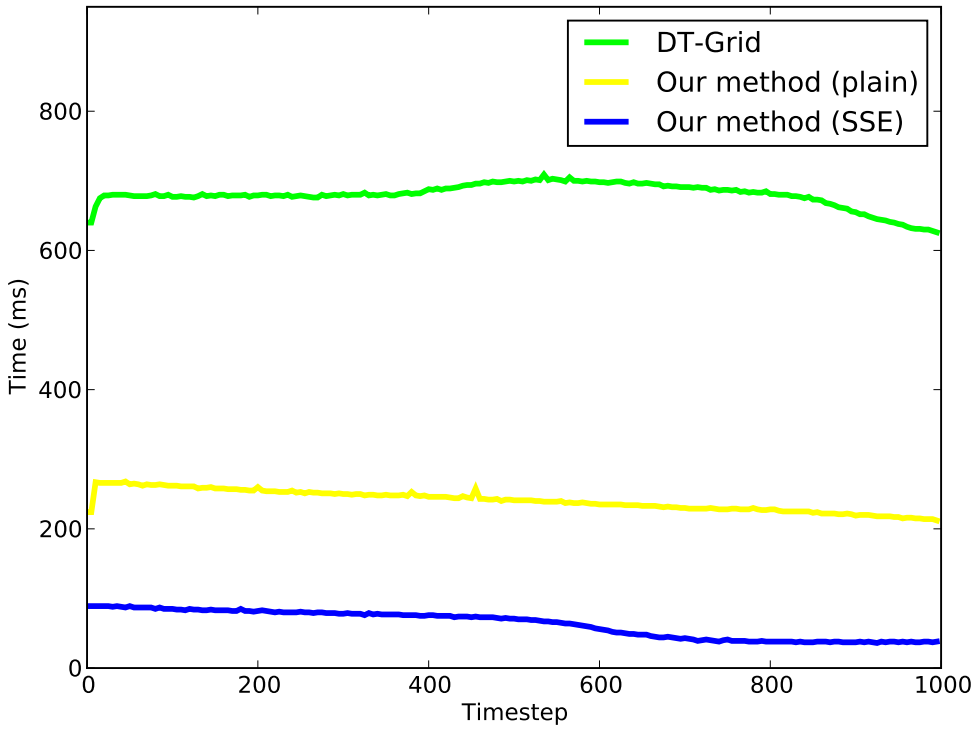


Figure 6.7. *Computation time per time step, for DT-Grid, our method, and our SSE-optimized method.*

computation. Thus, the amount of computations differs less from time step to time step, and is consistently lower than that of DT-Grid. The graph of the SSE-optimized method follows the graph for the non-optimized method.

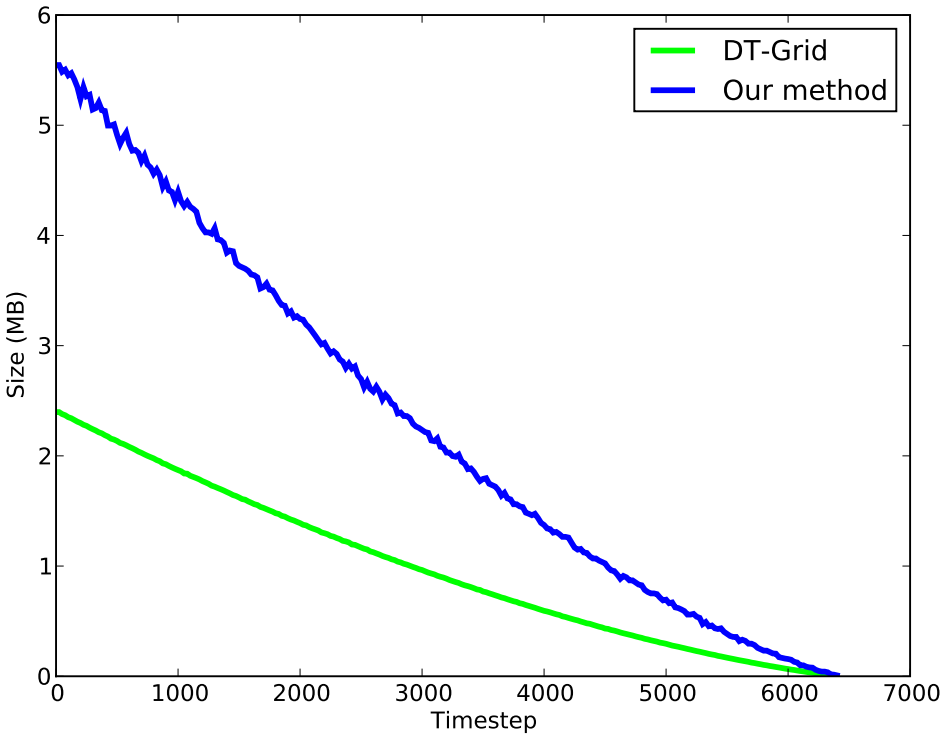


Figure 6.8. *Memory usage of our method and the DT-Grid method for the same level set computation (for comparison, a full* 256^3 *grid would take* 67.1*Mb of memory).*

6.5.3 Memory usage

We also compared the memory usage of our method (using tiles of size 4^3 voxels) and the DT-Grid method, for the same simulation as in section 6.5.1, see Fig. 6.8. During the simulation, the size of the interface shrinks to a point, and then around time step 6500 it disappears. From Fig. 6.8 it can be concluded that the memory usage of our method is a factor of about 2.5 larger than that of DT-Grid, seemingly independent of the size of the interface.

6.5.4 Periodic velocity field advection

Similar to DT-Grid, our method has a low memory footprint, allowing large computational grids, or conversely, large resolution level-set representations. Enright *et al.* [34] proposed an experiment to demonstrate the excellent volume-conservation property of their particle level set (PLS) method. Whereas they used a 100^3 grid and were able to resolve the interface, we ran the experiment on a 1024^3 grid, similar to Nielsen and Museth [89] for their DT-Grid method. Fig. 6.9 shows some snapshots taken during the simulation. As can be seen, our method can fully resolve the interface on a 1024^3 grid, similar to the DT-Grid method. The peak memory usage of our

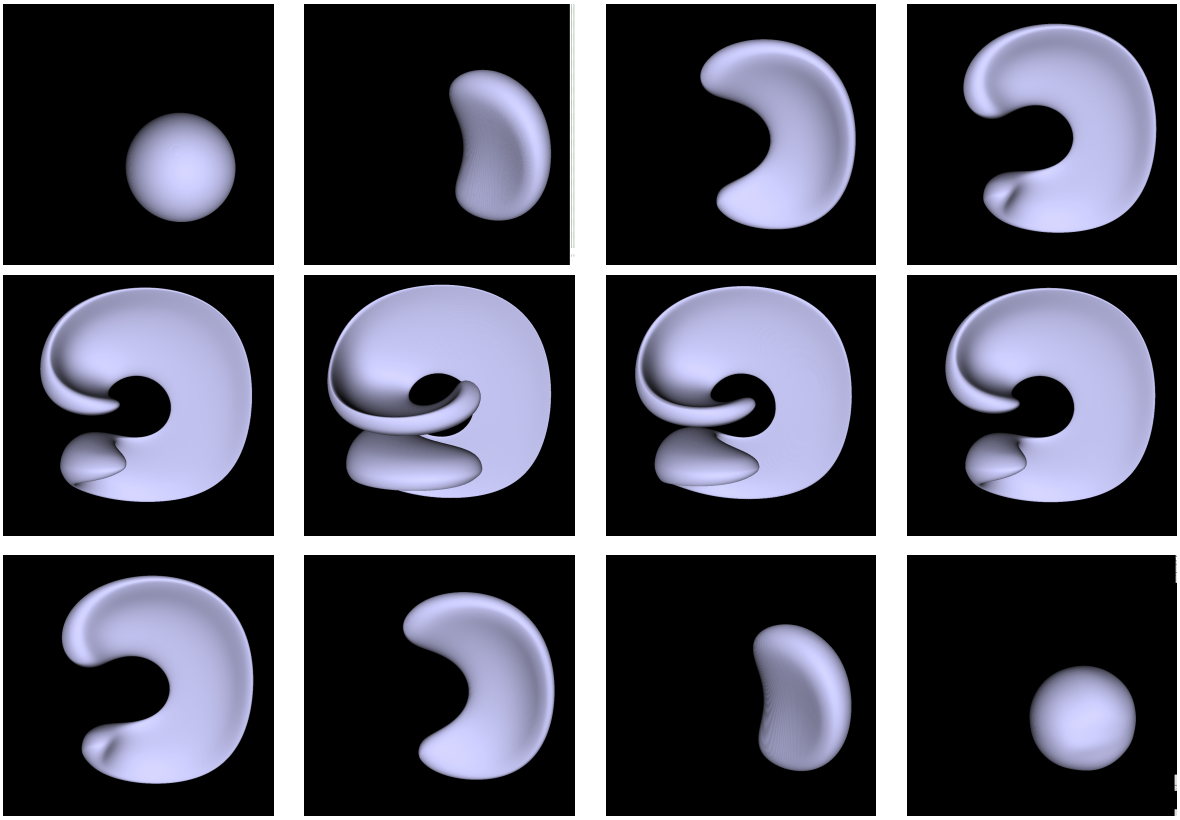


Figure 6.9. *Simulation of periodic velocity field advection, as proposed by Enright* et al. *[34]. Our method allows high resolution level set representations (a* 1024^3 *grid was used for this experiment). Left-to-right, top-to-bottom: snapshots taken during the simulation.*

method was 149 MB, which is about twice as much as for DT-Grid, whereas a full-grid method would need more than 4 GB of memory, see [34, 89] for further details.

In cases where the velocity field cannot be evaluated analytically (as is the case for Enright's test), or at arbitrary grid locations, standard interpolation methods such as trilinear interpolation have to be used. Further, if the velocity field is defined on a grid with a different resolution than the level set grid, resampling followed by interpolation can also be employed.

To further test the accuracy of our method, we successively increased the grid resolution and observed the maximum time t_d that can be achieved without deteriorating the developing thin sheets. The grid resolution G was set to $G = 128^3, 256^3, 512^3, 1024^3$ and the period of the field was set to $T = 5$, so that the maximum deformation takes place when the simulation time t becomes $t_d = T/2$. The results of this experiment are shown in Fig. 6.10. As also found in [**?**] and shown in Fig. 6.10, a grid resolution of 512^3 is sufficient to accommodate the deformation, provided that the period of the field is at most $T = 3.0$. Note that, periods as large as $T = 4$ are also possible if the grid resolution satisfies $G \geq 1024^3$.

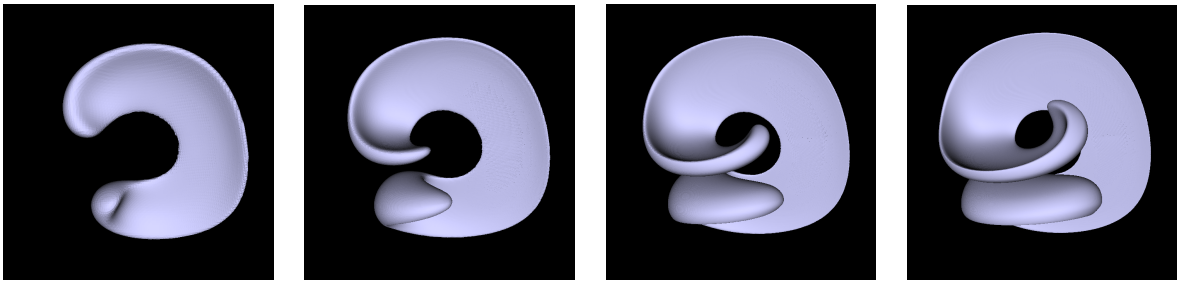


Figure 6.10. *Maximum deformation versus increasing grid resolution for Enright's test. Left-to-right: snapshots at maximum deformation time* $t_d = 0.77, 1.1, 1.55, 2.05$ *obtained at increasing grid resolutions* $G = 128^3, 256^3, 512^3, 1024^3$.

6.5.5 Tile size considerations

The drawback of our approach is that it has a larger memory overhead than DT-Grid, because of the granularity of the tiles. This overhead depends on the tile size – up to a point, the tile size is a compromise between computational efficiency and memory overhead. Choosing a tile size of 3^3 voxels hampers performance as it becomes inconvenient to use SSE SIMD instructions. The use of even smaller tiles would make the tile management overhead larger and similar to that of DT-Grid (see below), as more smaller active tiles need to be managed. Choosing a larger tile size results in both memory and computation overhead, as more values are computed unnecessarily.

We have also performed the experiment of subsection 6.5.1 with 3^3 tiles and without SIMD optimizations, and did not observe any significant difference with regard to speed, compared to the case when 4^3 tiles were used. Thus, in this case, one can conclude that the gain due to less numerical computation was offset by the added tile management overhead.

6.5.6 Tile management overhead

In practice the DT-Grid method has the advantage that the interface can be represented more sparsely. This implies less memory usage and less time spent on actual computation, e.g., no need to compute 4^3 voxels for those tiles in which only one voxel of the interface resides. Yet, DT-Grid adds significantly more overhead for maintaining the stencils. The following are some of the requirements of DT-Grid:

- an iterator for each stencil voxel is needed, which means 39 instead of just the 27 that we have for the surrounding blocks (HJ-WENO case);
- the iterators have to be advanced after each computed voxel; in our case, time can be spent entirely on computing the values of a tile before advancing them;
- DT-Grid uses 9 arrays (three for each dimension), while our method uses only a flat sorted list, making it easier to append ("push") without having to manage various connected components.

Quantifying the exact impact of the tile management overhead is quite complex in practice. This overhead is always lower for our method than for DT-Grid, as the number of tiles is always smaller (assuming the tile size is not 1^3) than the number of voxels. Furthermore, assuming a tile size of 4^3, the overhead can be 64 times less. This would happen only in the specific situation that DT-Grid would fill up every 4^3 tile entirely. In this case, the memory usage of DT-Grid and our method would also be (approximately) the same. In realistic cases, the interface will only occupy part of the tiles, and DT-Grid would only store voxels close to the interface. The average number of voxels per tile that are part of the interface will also be substantially larger than 1, as the level set embeds a 2-D surface in 3-D space. Simplistically spoken, a tile embedding an axis-aligned plane would use 4^2 of the 4^3 voxels, implying 4 times overhead. Experimental measurements using profiling tools revealed this factor to be consistently around 2.8 (bigger is worse), which approximately coincides with the memory overhead that we measured in subsection 6.5.3. This implies that the effective savings in tile management overhead will be a factor of about $\frac{64}{2.8} \approx 23$.

6.5.7 Discussion of our method

As our method is a hybrid between sparse methods with data structures for efficient level set computation such as DT-Grid on the one hand, and dense representation such as the full grid-based method on the other, it has favorable properties of both.

First, the speed and optimization potential of our method is crucial. As the main data structure is a sorted list processed in order, cache coherence can be maximized. Also, the tile size can be set so that tiles start at memory page boundaries, thus minimizing memory transfer overhead. Furthermore, as the values for a tile are stored together, and there is little data dependency (conditional logic) in the computation of the values for a tile, fine-grain parallelism in the form of SIMD instructions can be utilized to compute up to a whole tile of values at the same time.

Hierarchical structures such as octrees and DT-Grid can be quite intricate, while our method is much more straightforward to implement. As tiles are computed in a fashion similar to the full-grid approach, this constitutes an advantage when starting from a grid-based or narrow-band implementation. The only major difference is that an active list has to be maintained around the level set surface. There are no special structures with pointers that need to be maintained, which also facilitates hardware implementations.

The memory locality within tiles can also increase the performance of methods used to *visualize* interfaces, such as the Marching Cubes algorithm, as finding neighbour values is very fast. Here too, fine-grain parallelism can be used, for example to find the zero crossings for the entire tile at once.

6.5.8 Parallelization over multiple CPUs

Our method is well suited for both fine and coarse grain parallelism. Within a tile, all values can be computed at once, which we demonstrated above by using SIMD instructions. Secondly, all the tiles could in principle be computed at the same time, by parallelizing this process over multiple CPUs.

Moving the stencil of neighbouring tiles over the data structure (Section 6.4.4) is inherently a serial operation. This has to be taken into account when dividing the computational work over processors. For example, we could index into the structure using random access (Section 6.4.5), and use this to initialize the neighbour pointers for each processor once at the start of the iteration process.

The other challenge lies in parallelizing the tile management step, as the active list has to be maintained lexicographically sorted by coordinate. This could be achieved by allocating a part of the active list to each processor, having it process the dilation for that part of the list, then collecting the results in the correct order into a new active list.

6.6 Conclusions and future work

In this chapter we have presented an efficient data structure with associated operations for the level set representation, and compared the resulting method with the current method of choice, the DT-Grid method. With regard to performance, given the same numerical simulation code, our method turned out to be faster by a significant factor. After fine-grain parallelization using SIMD instructions the performance gain was further increased to a factor of 8.5.

As our method is also well suited for coarse-grain parallelization, we are in the process of devoting further work to various possibilities for leveraging highly parallel architectures such as GPUs. It would also be interesting to see whether it is possible to reduce the larger memory requirements of our method, without sacrificing one of its prominent advantages, namely, low data structure management overhead.

To be submitted.

Chapter 7

Real-Time Sparse Level-Sets on Graphics Hardware

7.1 Introduction

Since its inception [98], the level set method has become the favorite technique for capturing and tracking moving interfaces, and found a host of applications in a wide variety of research fields ranging from chemistry and physics to computer vision and graphics. The basic idea is to represent the dynamic interface (*e.g.*, contour or surface) implicitly and embed it as the zero level set of a time-dependent, higher-dimensional function. Evolving the interface with a given velocity in the normal direction then becomes equivalent to solving a time-dependent PDE for the embedding level set function. The main advantages of the level set method are that it allows the interface to undergo arbitrary topological changes and conveniently provides intrinsic geometric properties such as normal and curvature information.

Unfortunately, although the level set method is well suited for tracking highly deformable models such as mud, water and smoke in accurate, physically-based simulations [24, 35, 77], its use for real-time interactive systems has been hampered due to the high computational demands. The cost which has to be paid for the flexibility offered by the level set method is of twofold nature: first, *computationally*, one has to solve the level-set PDE in a higher-dimensional space than that of the embedded interface, and secondly, the *memory requirements* are higher than the size of the interface, as one needs to explicitly store a uniform Cartesian grid for solving the level set PDE. To address these issues, a number of techniques have been proposed, see Section 7.2. These methods rely on the fact that it suffices to solve the PDE only in the vicinity of the interface in order to preserve the embedding. Thus, the computational requirements scale with the size of the interface.

We leverage the increased computing power of the GPU to achieve real-time simulations based on level sets. This requires specially designed data structures, as most CPU methods rely on complex data structures that do not fit well in the streaming model of GPU computing. Although interactive level set methods on the GPU do exist, they are constrained to small grid resolutions of typically 128^3 voxels, see Section 7.2. The Sorted Tile List (STL), introduced recently by [144], constitutes an efficient data structure for tracking dynamic interfaces through the level set method. Inspired by the increased potential for parallelism of the STL method,

we present efficient and scalable parallel algorithms for the level set method on graphics hardware, see Section 7.3. Although we focus on Nvidia's Compute Unified Device Architecture (CUDA) parallel programming model in our GPU implementation, our algorithms can certainly be adapted to OpenCL [63], the upcoming open standard for parallel and GPU computing, and even to DirectX 11 Compute Shaders.

Our fast GPU method brings the use of level set methods into the realm of interactive simulations. To demonstrate its effectiveness, we consider two well-known graphics applications: surface reconstruction and free-form level-set surface editing.

Specifically, the main contributions are:

- A highly efficient, tile-based, sparse-grid method for the level-set representation, which works in any number of dimensions on grids of unbounded size and runs entirely on graphics hardware or other highly-parallel computing platforms.
- Scalable parallel algorithms for iterating and maintaining the proposed GPU data structure, enabling efficient simulations based on level sets, at high grid resolutions.
- An efficient rendering method, based on marching cubes [76] that displays implicit surfaces defined on the sparse volumetric grid in real-time, attaining high performances and small memory footprints for large volumetric grids.
- A novel, multi-resolution method for surface reconstruction based on GPU level sets, which compares favorably with existing state-of-the-art methods.

Finally, our free-form level-set surface editing application performs at interactive rates on large models, obtained, *e.g.*, through our surface reconstruction method.

7.2 Previous and related work

As the level-set method is a tremendously popular approach for tracking moving interfaces, there has been continuous interest in developing improved algorithms to address the computational issues involved. Here we focus our brief review only on closely-related methods and state-of-the-art results. For some general issues in designing GPU-based algorithms, see [67, 164].

7.2.1 Efficient level set methods on the CPU

The computational issue was first addressed with the introduction of the narrow-band schemes [2, 99, 153], which restrict the computations to a small vicinity around the zero level set representing the deforming interface. Although narrow-band methods improve computational efficiency, they still need to explicitly store a full grid and additional data structures. Thus, such methods have storage complexities scaling with the size of the grid. Quadtree and octree-based methods [77, 84, 127] do achieve smaller memory footprints, but the non-uniform discretizations limit the class of problems which can be tackled to hyperbolic ones [89]. An alternative approach for reducing memory requirements, called the "Sparse Block Grid" method, was presented in [13]. In 3D, this method divides the volume of size n^3 into small blocks of size m^3 voxels each. A coarse grid

of size $(n/m)^3$ stores pointers to blocks that intersect the interface. Although this method has non-optimal storage complexity, it maintains constant access time similar to the full-grid method.

7.2.2 Level set GPU methods

Because the computing power of GPUs is currently increasing at a faster pace than that of CPUs, there have been several efforts to accelerate level set simulations by using graphics hardware. The first GPU implementation of level sets is due to Rumpf and Strzodka [111]. More recently, parallel implementations of particle [27] and marker [83] level sets were also proposed. These methods achieve 15 and 24 fps respectively, on full grids of size 128^3 voxels. We stress that both methods are more computationally involved than the pure level-set method, and thus a direct comparison with regard to efficiency is unfair.

To the best of our knowledge, the only memory-adaptive model for the level set representation on the GPU is due to Lefohn et al. [68]. In this method, the domain is decomposed into small 2D tiles, of which only the tiles with non-zero derivatives are stored on the GPU. A look-up table spanning the entire domain stores a pointer to the data for every tile. Memory management is performed by transferring a bit-vector image of about 64 kB from the GPU in every iteration, after which the CPU loads and unloads tiles based on their necessity for the computation during the next iteration.

7.2.3 Sparse CPU methods

Recently, Nielsen and Museth introduced the so-called Dynamic Tubular Grid (DT-Grid) [89], a recursive, compressed level-set representation inspired by the compressed-row-storage technique used to represent sparse matrices. The authors show that the memory requirement of DT-Grid is optimal, *i.e.*, it is proportional to the size of the interface. Moreover, their experiments showed that the 3D DT-Grid is faster and more memory efficient than state-of-the-art octree-based approaches.

Huston *et al.* [54] use hierarchical run-length encoding (RLE) in a dimensional-recursive fashion to encode the domain in a series of runs, each associated with a specific run code. Regions away from the narrow band are encoded to just their sign representation, while the narrow band is stored in full precision. Although this method is more flexible than DT-Grid, the price paid is a slight increase in computation time and memory usage, compared to the DT-Grid.

Similar to [13, 68], the Sorted Tile List (STL) method [144] divides the domain into fixed-size *tiles*, such that each tile represents a part of the domain of the level set function ϕ. Tiles outside the narrow-band are discarded. The remaining narrow-band tiles form the so-called active set. A key aspect of the STL method is that the active set is just a list of (active) tiles, *lexicographically ordered* by coordinates. This allows finding the active neighbouring tiles of a given tile in constant time, such that updating the level-set values of all tiles is linear in the number of active tiles. The STL method was found to be faster than the recent approaches mentioned above [13, 54, 89] and more importantly, the algorithm can greatly benefit from both fine- and coarse-grain parallelization by leveraging SIMD and/or multi-core configurations.

7.2.4 Surface reconstruction

Surface reconstruction from unorganized point clouds has been intensively studied. For recent surveys see [10, 56, 62, 163] and references therein. Despite the increased availability of commodity parallel platforms, there has been very little work on parallel algorithms for surface reconstruction. Only Zhou et al. [163] and Bolitho et al. [10] implement the Poisson method [62] on the GPU, and on shared and distributed-memory parallel platforms, respectively. In [163] significant speedups were obtained on the GPU at small grid resolutions. As pointed out by Bolitho et al. [10] a limitation of this method is that it maintains the entire octree and additional data structures in GPU memory, thus drastically limiting the maximum resolution. Moreover, the method is more susceptible to noise than the original one, due to some computational simplifications that were introduced. Finally, the results in [10] indicate that the Poisson method scales well on distributed-memory parallel computers and badly on shared-memory architectures.

7.3 Proposed GPU level set method

We first give a brief overview on level sets, and provide a brief summary of the CPU STL method [144]. Then we introduce our fast GPU method, based on the CUDA programming environment [70].

7.3.1 Generic level set equation

In the level set method, a closed $(d-1)$-dimensional hyper-surface $\mathbf{\Gamma}(t=0)$ is implicitly defined as the zero set of a d-dimensional Lipschitz continuous function $\phi(\mathbf{x}, t=0) : \mathbb{R}^d \rightarrow \mathbb{R}$, *e.g.*, the signed distance to $\mathbf{\Gamma}(t=0)$, with $\mathbf{x} \in \mathbb{R}^d$. A generic equation for $\phi(\mathbf{x}, t)$, representing the evolution of $\mathbf{\Gamma}(t)$, is [98]

$$\frac{\partial \phi}{\partial t} = -F(\mathbf{x}) \left|\nabla \phi\right| - \mathbf{U}(\mathbf{x}, t) \cdot \nabla \phi + \alpha \kappa \left|\nabla \phi\right|, \tag{7.1}$$

where α is a constant, $\mathbf{n} = \nabla\phi / \left|\nabla\phi\right|$ denotes the normal and $\kappa = \nabla \cdot \mathbf{n}$ is the mean curvature of the hyper-surface. Accordingly, the interface (hyper-surface) moves under three simultaneous influences. The first right-hand side term, involving the position-dependent signed scalar function $F(\mathbf{x})$, defines its motion in the normal direction. Second, it is being passively convected by an external velocity field $\mathbf{U}(\mathbf{x}, t)$, whose direction and strength depend on position and (possibly) time, but not on the front itself. Third, the interface collapses with a speed proportional to its curvature. Since during the simulation, ϕ should be maintained close to a distance transform, an additional rescaling-speed term $\mathrm{sgn}(\phi)(1 - \left|\nabla\phi\right|)$ [68] is used, which enforces $\left|\nabla\phi\right| = 1$.

The curvature term is discretized using central differences, whereas all the other terms (including the rescaling-speed term), are discretized using upwind differences in the appropriate direction. Here we use either first-order upwind differencing, or the fifth-order accurate HJ-WENO scheme [74] ensuring less numerical dissipation. For the time derivative we either use forward differences, or the third-order accurate TVD Runge-Kutta scheme. Unless mentioned explicitly, we use the simpler numerical schemes.

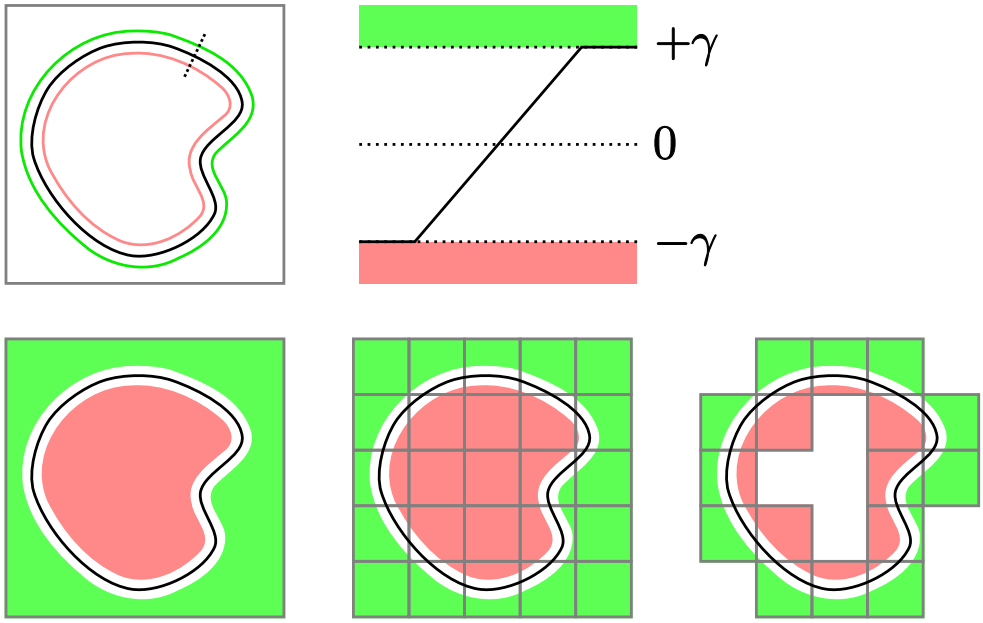


Figure 7.1. *Sparse tile-based representation in 2D.* Top row, left-to-right*: interface (black), and the $-\gamma$ (red) and γ (green) iso-contours; profile of level-set function ϕ.* Bottom row, left-to-right*: domain with $\phi \geq \gamma$ (green), and $\phi \leq -\gamma$ (red); domain divided into tiles; sparse domain with inactive tiles (completely red or green) removed.*

7.3.2 CPU STL method

The Sorted Tile List (STL) method [144] divides the domain into fixed-size *tiles*, such that each tile represents a part of the domain of the level set function ϕ. Tiles outside the narrow-band, with values outside the range $(-\gamma, \gamma)$, are discarded, see Fig. 7.1. The remaining narrow-band tiles form the so-called *active set*. A key aspect of the STL method is that the active set is just a list of (active) tiles, *lexicographically ordered* by coordinates. This allows finding the active neighbouring tiles of a given tile in constant time, such that updating the level-set values of all tiles is linear in the number of active tiles.

As the interface evolves, tiles that are no longer close to the zero level set have to be removed, and new tiles needed during the next time step must be added to the list of active tiles. This is accomplished in the so-called tile-management step, whose basic idea is as follows. For each currently-active tile, it is first determined which of the neighbouring tiles are needed in the next time step. The borders of a tile that are being approached by the evolving interface are signalled through a set of activity flags. If the activity flag for a certain border is set, a tile has to be created if it is not yet present in the direction of the border. If the interface has just left a certain tile, all activity flags for that tile are set to zero. If none of the neighbouring tiles request for the tile to be retained, it is safely removed from memory. During tile management an expensive re-sorting step is avoided by carefully tracking newly added and removed tiles.

For full details on the STL method we refer to [144].

7.3.3 GPU sparse level sets

Although the STL method maps well to the CPU computational model, since it relies on tiles to represent the narrow band, an analogous, sparse, tile-based GPU method is more complex. For example, in the STL method, tile management is performed in one traversal of the active list. In contrast, to achieve good memory throughput and thus efficiency, our GPU method performs five passes, using more conceptually-involved algorithms for iterating over the active list.

First we introduce the tile-based data structure that is central to our approach and then we present the details of both computation and tile-management steps.

Data structure

Considering the level set method, the most important means of achieving high performance with CUDA is to optimize memory throughput. One needs to make sure that data structures are laid out efficiently in memory and (slow) global memory transfers are minimized. In SIMD-like architectures, *Array of Structures* (AOS) approaches are generally less efficient than *Structures of Arrays* (SOA), in the common case in which steps of the algorithm use a subset of the structure fields. In an AOS, the entire record would have to be read into cache, while in a SOA a consecutive span of a certain field can be read and processed. Even though the global memory in CUDA is uncached, the reasoning still applies because the device writes and reads quantities of 64 or 128 bytes at once. The consecutive memory locations must be simultaneously accessed by the threads (scalar execution units). This is called *memory access coalescing* [92], and it represents one of the most important optimizations in CUDA.

The main data structure consists of the following arrays, see Fig. 7.2:

- The *active list* is an array of 8-byte structures serving as storage for two integers: *position* (concatenated tile coordinates) and *tile id* (an index into other arrays holding, *e.g.*, the data or border flags of this tile). This structure is provided to CUDA kernels as a 1D `int2` texture for cached read access.
- The *data array*, stores values of function ϕ for each tile. We use tiles of size 4^3 voxels, stored in single precision, as this setting represents the best tradeoff between efficiency and memory footprint. Thus, the data array stores $4^3 \times 4 = 256$ bytes per tile, which can be accessed conveniently with coalesced global memory accesses. Depending on the time-integration scheme used, two or three data arrays have to be maintained on the GPU, see subsection 7.3.1.
- The array of *border* and *activity flags* of each tile. This array stores the sign of the data outside a tile in each of the 26 directions, in case the tile has no direct neighbour there. The overall activity bit signifies whether a tile is active or not during the next iteration. This structure is provided to CUDA kernels as a 1D `int` texture.
- An array of unused indices, the *free stack*, is also maintained, in the form of a stack of *tile id* values.

Tile coordinates are stored as 32-bit integers, as this makes lexicographic comparison very efficient, and one can apply neighbour offsets by simple addition and subtraction operations. In 3D, we have at our disposal only 10 bits on average for each dimension, which limits the

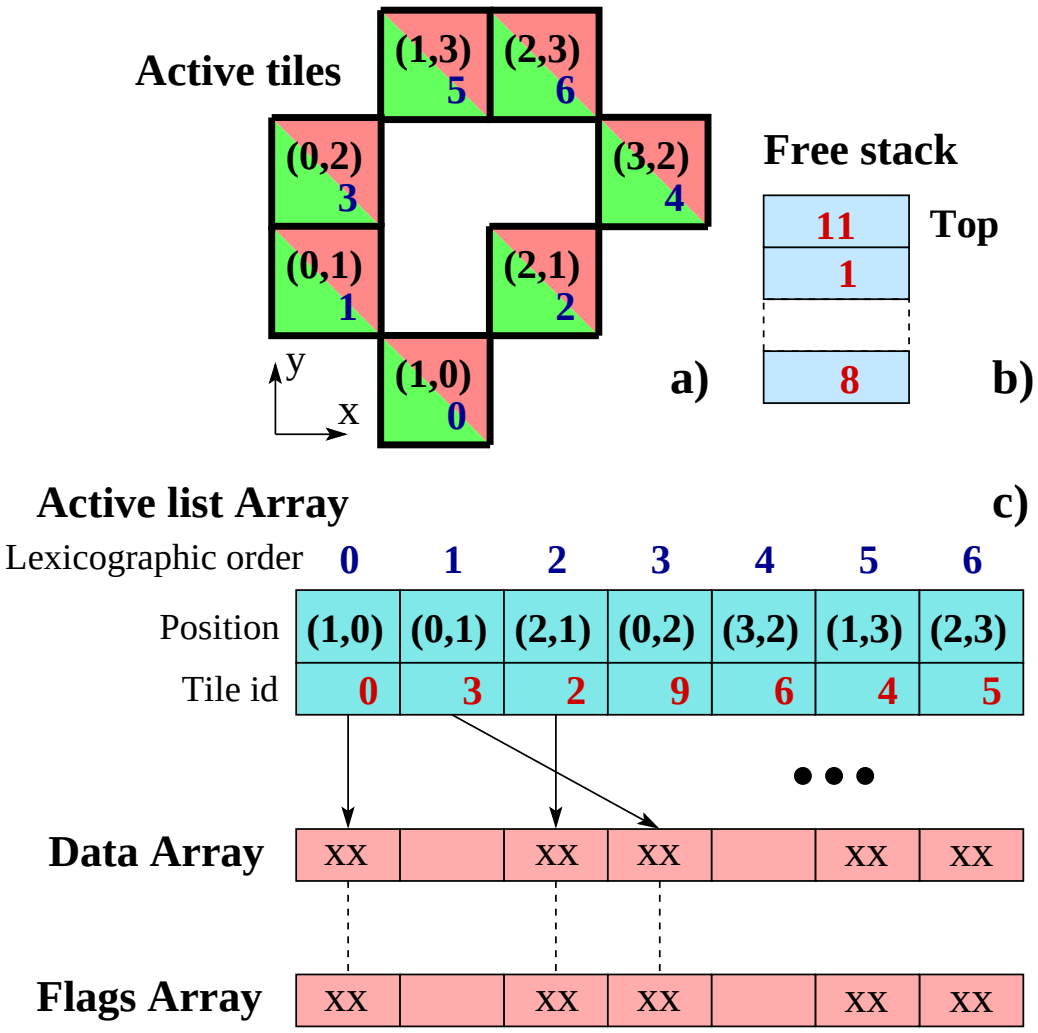


Figure 7.2. *GPU data structures (arrays).* a): *active tiles with numbers representing tile positions and indices in the active list, according to the lexicographic order by tile position;* b): *the stack of free tile identifiers; when a tile becomes inactive, its* tile id *is pushed on the stack; conversely, if a new tile has to be created, its* tile id *is popped from the stack;* c): *the active list contains tile positions and identifiers; The* tile id *is used as index in the data and (border) flags arrays.*

maximum size of the volume to $(2^{10} \times 4)^3$, using 4^3 tiles. If larger volumes are needed, 64-bit position identifiers could be used at the expense of some speed and storage space.

When a tile becomes inactive, its identifier is pushed on the free stack, and when a tile becomes active, an identifier is popped from the stack, see Fig. 7.2. Only if the stack is empty, new memory must be allocated by the CPU. Maintaining a stack in CUDA is non-trivial, as it must be accessible by all threads in parallel, and there is no way to synchronize thread accesses across blocks. Although it is possible to use atomic integer operations to maintain a stack pointer,

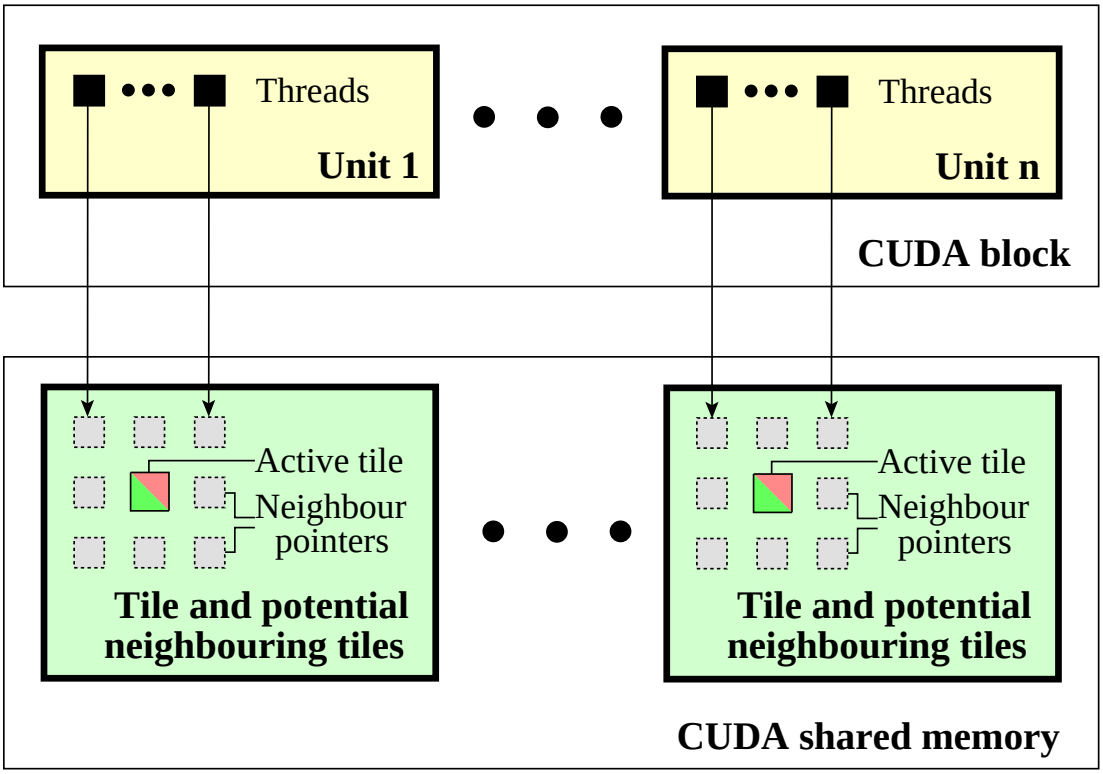


Figure 7.3. *Tile iteration with access to neighbouring tiles. Each thread of a unit tracks the pointer of a neighbouring tile. A CUDA block consists of multiple units.*

this results in a lot of added communication with the global memory. A better option is to count the number of push and pop operations that each thread needs to perform, and then use a parallel prefix-sum algorithm [47] to make sure each thread only accesses its own part of the stack.

Overall flow

An efficient CUDA algorithm divides the work between CUDA *blocks* and *threads* in a way that yields minimal overhead. For this, we introduce the concept of *units*, *i.e.*, groups of threads working together on one tile at a time. These consist of a number T of threads at least equal to the number N of neighbouring tiles ($N = 26$ in 3D), rounded to the closest power of two. Each thread inside the unit tracks the pointer of a neighbouring tile. A CUDA block can consist of a multiple of these units, which act independently of each other, see Fig. 7.3. The total number of units is chosen so as to saturate all GPU multiprocessors.

Each level-set simulation step consists of one *computation* (subsection 7.3.3) and one *tile management* (subsection 7.3.3) substep. During the computation step, the active tiles are visited and the level set PDE is used to update function ϕ. In the tile management step, new tiles are created and existing tiles are either removed or kept.

Algorithm 7.1 Iterating over the sorted tile list with T parallel threads and access to the $N = 26$ potential neighbouring tiles. Function *iter* is called for each tile by each of the first N threads, $T \geq N$.

Input: `unit` {unit number}, `sub` {thread in unit}, `size` {active list size}, `coord[size]` {tile coords}, `neighbourhood[N]` {neighbour coords}

```
 1: offset ← unit * size/num_units {begin of work for this unit}
 2: end ← (unit + 1) * size/num_units {end of work}
 3: ptr ← BSEARCH(size, offset, sub) {locate neighbour, binary search}
 4: while offset < end do
 5:    cur ← coord[offset] {take coord of current tile}
 6:    c ← cur + neighbourhood[sub] {neighbour coord for thread}
 7:    while coord[ptr] < c do
 8:       ptr ← ptr + 1 {track neighbour}
 9:    end while
10:    match ← coord[ptr] = c {if coord matches, neighbour exists}
11:    iter (match, ptr, cur, unit, sub) {call iterator}
12:    offset ← offset + 1 {advance to next tile}
13: end while
```

Parallel tile iteration with compute stencil

Tile iteration with compute stencil, *i.e.*, with access to neighbouring tiles, is an essential building block of our method, as it is used for updating the level set function and for rendering the deformable surface (subsection 7.3.4). Its pseudo-code is given in Algorithm 7.1. First, on a coarse-grained level, the active list is divided evenly into parts and each part is assigned to a unit (line 1 and 2). Secondly, each thread of a unit assumes responsibility for one neighbour pointer, see Fig. 7.3.

At the beginning of the kernel (line 3), each thread in each unit performs a *binary search* for one of the neighbours of the first tile. This is the only binary search that is required, as advancing to the next tile can be done linearly, lines 4–13. The algorithm advances until it reaches the beginning of the next part (variable `end`), which is handled by the next unit. For each tile, in each (active) thread, function *iter* is called. The unit (`unit`) and thread (`sub`) identifiers are also passed to the iterator for convenience, as the values of `match` and `ptr` are different for each thread of a unit.

Computation

The data flow in this step is illustrated in Fig. 7.4. This step is implemented by one CUDA kernel (**calc_kernel**), which requires one iteration through the active tile list, using Algorithm 7.1 (iterator **iter** is set to function **compute**). For this kernel, each unit consists of $T = 64$ threads, such that in the **compute** function, all threads of a unit collectively update the 4^3 data elements of a tile. Since $T > N$, only $N = 26$ threads are active when tracking the pointers of the neighbouring tiles, in Algorithm 7.1. However, since clearly the most expensive part in this step

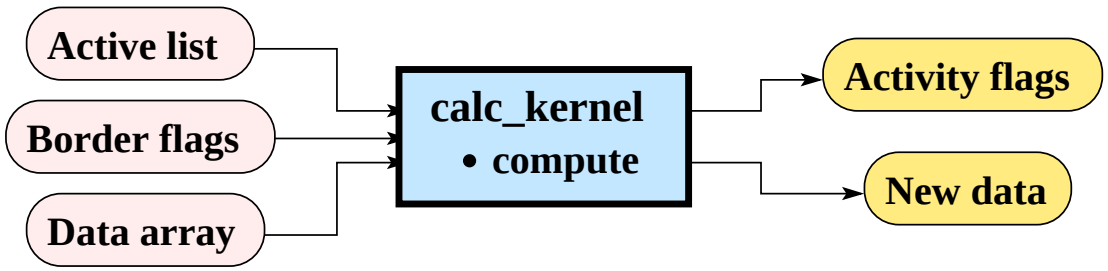


Figure 7.4. *Data flow in the computation step. CUDA kernel* **calc_kernel** *receives as input the list of active tiles and their attributes, and updates ϕ according to the level set PDE at each tile voxel. It also updates the* activity flags *used in the tile management step for deciding which tiles are needed during the next iteration.*

is the actual computation of ϕ values, the resulting CUDA kernel is highly efficient. Note that the parameter γ that defines the narrow band is set to $\gamma = 1.5$, as in [144]. Through the **compute** function, thread units perform the following:

1. Read an entire tile into shared memory, having each thread of a unit read a floating-point value. Given that tiles are stored in a consecutive and aligned fashion, this read is coalesced. Tile data is stored in the center of a $6 \times 6 \times 6$ cubic array in shared memory.
2. Read the $6^3 - 4^3 = 152$ border elements. If a neighbour exists in the direction of the border, read the value from device memory, otherwise substitute $-\gamma$ or γ depending on the border flag.
3. Each thread updates ϕ at its location, by evaluating Eq. (7.1).
4. If the resulting value is outside the range $(-\gamma, \gamma)$, it is set to the nearest value within the range. Otherwise, the activity bit for this thread is set.
5. Write the entire tile back to device memory, having each thread writing a floating-point value. Similar to the reading step, this write operation is coalesced.
6. Determine the activity flags for this tile by doing a reduction (bitwise-OR the activity flags of the threads together), and write the result to device memory. This write is not coalesced, as only the first thread of the unit performs the write operation. However, due to the small amount of data written this write operation is also efficient.

When reading the border elements, accessing 152 scattered values from multiple tiles has to be distributed over 64 threads. The access pattern should minimize the number of memory transactions that the hardware has to perform, given the CUDA constraints. Thus, for each operation, we store the destination offset, source tile, and source-tile offset encoded into one word. The operations are sorted so that 34 memory transactions are needed on current hardware, which is optimal as this number is equal to the number of 128 byte-aligned segments accessed. To store this pattern a small look-up table in shared memory is used, which is copied from device memory at the beginning of the kernel.

Since tiles contain 4^3 voxels, this implies that the larger finite-difference discretization stencil required by the HJ-WENO scheme (a 3D cross centered in a 5^3 axis-aligned cube), can also be

implemented using only direct neighbouring tiles.

Tile management

For each currently-active tile, it is first determined which of the neighbouring tiles are needed in the next time step. If the interface approaches a tile border, the tile at the other side of that border has to be present in the next time-step to continue the computation. If the activity flag for a certain border is set, a tile has to be created if it is not yet present in the direction of that flag. The basic idea then is to iterate over the list of tiles, and for each tile to expand the tile set by creating tiles in the directions whose activity flags are set. A straightforward implementation of this idea is to perform a morphological "dilation" of the set of active tiles by a 3^3 structuring element [114]. For each element in the dilated version of the set, it should then be determined whether to create, remove or keep the tile at that position. This assures that new tiles will only be created at most one time.

During the tile management step, Algorithm 7.2 is used twice to iterate over the dilated version of the active list. This is parallelized similarly to the tile iteration step from subsection 7.3.3.

The minimal unit size of $T = 32$ threads is chosen, which equals the warp size of the underlying hardware. Since threads within a warp are automatically synchronized, it is convenient to use this approach if values need to be combined from the variables of individual threads, such as in line 6 and 15 of Algorithm 7.2.

In the GPU implementation, tile management cannot be done in just one pass over the active list as in [144], because it is not known in advance how many tiles will be created, removed or kept. For this reason, we split the tile-management step in multiple passes, each implemented by a separate CUDA kernel, see Fig. 7.5:

1. **count_tiles**: In the first pass, Algorithm 7.2 is used to iterate over the data structure. This CUDA kernel simply counts for each unit how many tiles are created, removed, and kept, using the activity and border flags. For each unit, it outputs the number of tiles in each of these categories.
2. **prefix_sum**: The second pass performs a parallel prefix-sum [47] on the previously-computed counts, to calculate offsets into the old and new active list, for each unit. As the number of thread blocks and units is limited and fixed, this scan can be done quickly in shared memory and then the result written back to device memory. This step converts the number of tiles which are created into an offset into the stack, the number of tiles which are removed into an offset in a list of tiles added to the stack, and finally, the number of tiles which are either kept or created into an offset into the new active list. After this pass it is possible to check whether there is enough space left to accommodate the new tile set, or if new memory needs to be allocated. To implement this, the total counts are passed back to the CPU.
3. **stat_cpu**: The third pass is a very small, one-thread kernel that computes and outputs a status record for the CPU, to determine how much memory is used and whether the structure should be resized. This record contains the following fields: the total number of tiles added, removed and kept, and the starting offset into the list of free tiles, so that the new tiles are allocated from the end, thus maintaining the LIFO ordering.

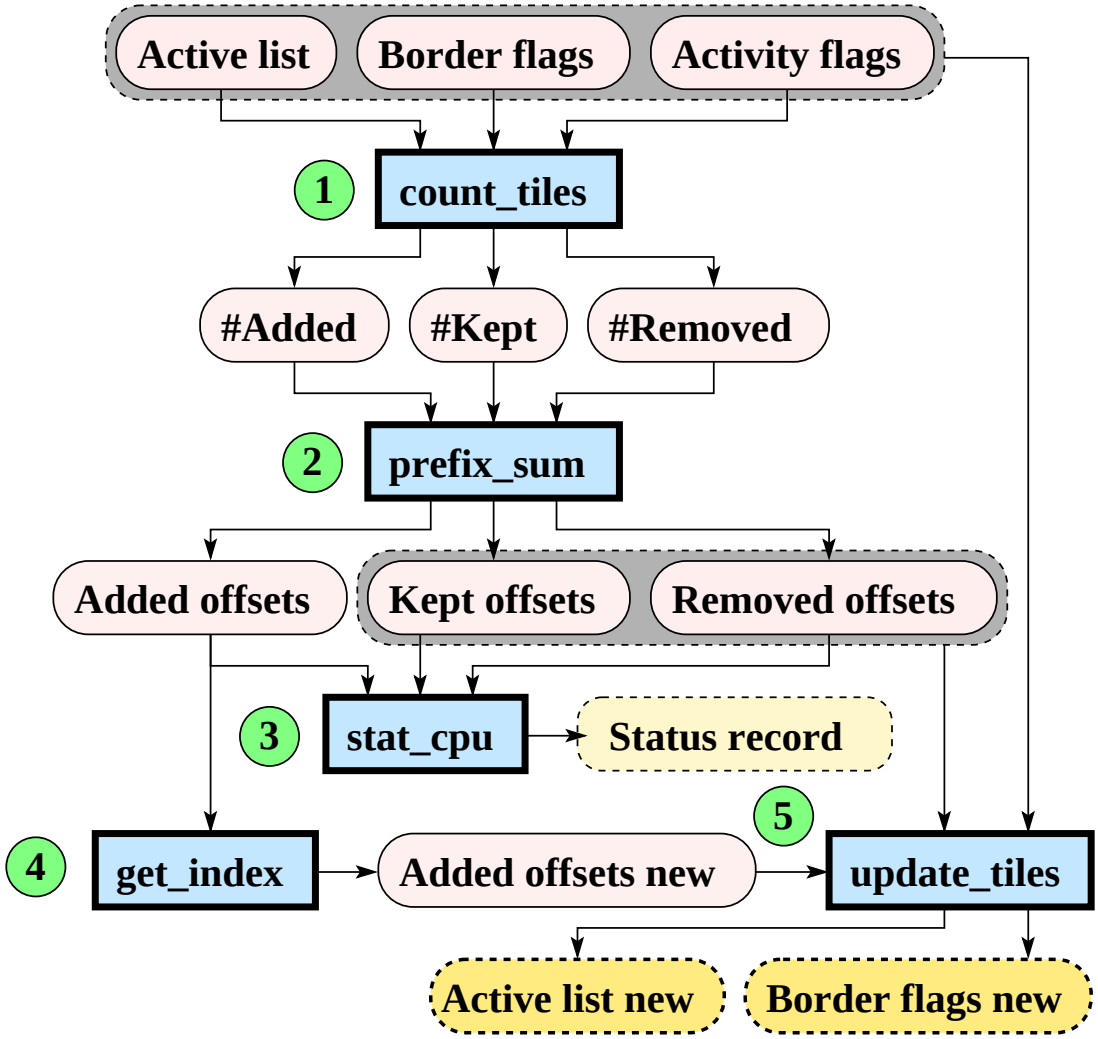


Figure 7.5. *The overall data flow in the tile management step. Rounded boxes represent data streams, rectangular boxes denote CUDA kernels and arrows depict directions of data flows; the order in which the kernels are invoked is shown with numbers inside green circular boxes.*

4. **get_index**: In the following step, a full pass over the offsets of newly added tiles is performed, and a constant value is added so that they can be used as index into the list of free tiles.
5. **update_tiles**: In the final pass, Algorithm 7.2 is reused, however, this time the new active list is created, new tiles are initialized and the border flags are updated.

When a tile is removed, one of the border bits of each of its neighbours must be updated to reflect whether the tile was inside or outside the interface. This results in a race condition, since more tiles could be changing the same neighbour at the same time. To avoid this one could use atomic bitwise operations, but these are not supported on all hardware. Therefore we use another, albeit slower, possibility, by storing a "will be deleted" bit during the first pass in the activity flags for each deleted tile. This is then taken into account in the last pass, when collecting

Algorithm 7.2 Iterating over a dilated version of a lexicographically-ordered list of tile coordinates, maintaining the order. Function *iter* is called for each tile by each thread. Here N is the number of neighbours of a tile.

Input: `unit` {unit number}, `sub` {thread in unit}, `size` {active list size}, `coord[size]` {tile coords}, `neighbourhood[N]` {neighbour coords}

```
 1: offset ← unit * size/num_units {begin of work for this unit}
 2: end ← (unit + 1) * size/num_units {end of work}
 3: ptr ← bsearch(size, offset, sub) {locate neighbour, binary search}
 4: loop
 5:    cur ← coord[ptr] − neighbourhood[sub]
 6:    cur ← REDUCE32(min, cur) {reduction, minimum value}
 7:    if cur >= coord[end] then
 8:       break {end of tile-set reached}
 9:    end if
10:    if coord[ptr] = (cur + neighbourhood[sub]) then
11:       my_match ← 2^sub {if coord matches, this neighbour exists}
12:    else
13:       my_match ← 0 {otherwise, it does not exist}
14:    end if
15:    match ← REDUCE32(or, my_match) {reduction, bitwise-or}
16:    iter (cur, match, ptr, unit, sub) {call the iterator}
17:    if my_match then
18:       ptr ← ptr + 1 {advance structuring element position}
19:    end if
20: end loop
```

the border-flag mutations of all neighbours and integrating them into the new value for itself.

7.3.4 Rendering the interface using CUDA and OpenGL

Traditionally, volume rendering methods [68], implicit surface polygonization [153] and ray-tracing [54] have been used to render the evolving/resulting interface. Our method uses implicit surface polygonization, employing marching cubes and runs entirely on the GPU. With modern programmable GPUs that support *geometry shaders*, it is possible to generate geometry on the fly [25]. We use an efficient intermediate-storage structure for the cube attributes, so that the output of a CUDA or CPU algorithm that computes a sparse volume can be directly visualized, without the need to access the entire volume in the geometry shader (in the form of a 3D texture, or otherwise).

Our polygonization algorithm is split into two parts. The first part, based on CUDA, iterates through the level-set data structure and generates a compact record for each cube (voxel) that is intersected by the surface. The second part, using a geometry shader, processes these records, generates triangle positions and normals, and sends these through the rendering pipeline.

7.4 Proposed surface reconstruction method

The proposed multi-resolution method for surface reconstruction employs convection of the evolving level-set surface (current approximation to the final reconstructed surface) towards the input sample points. Since for this application our aim is to achieve very large resolutions, we had to leverage the computational power of both GPU and CPU. Accordingly, the convection of the level-set surface runs entirely on the GPU, whereas the computation of the velocity field in which the surface is convected runs in parallel on the multiple cores of the host CPU. Using velocity fields based on the distance transform as in [161] is not an option, since at very large grid resolutions, *e.g.*, 2048^3 voxels, the storage requirements would be more than 30 GB of RAM. Instead, we use inverse-distance velocity fields similar to [56], which can be evaluated on the fly on the CPU using memory-efficient octree grids. Although octrees can be very efficiently built and maintained on the GPU [163], at high grid resolutions their storage requirements on current GPUs become problematic. Therefore, and since the most expensive part of our surface reconstruction method is the convection of the level-set surface, we perform convection on the GPU, and let the octree computations be handled on the CPU side. This approach also shows how communication between the GPU and the CPU (required also by other level-set applications) can be *effectively* accommodated by our level-set method, through the use of a simple caching scheme minimizing GPU – CPU memory transfers.

For evolving the level-set surface we compute a velocity field based on Shepard interpolation [119] of normalized direction vectors between locations in the narrow band and input point samples. Formally, let S denote the input set of point samples lying on or near the surface $\partial\mathbf{M}$ of an unknown object M. The problem is to accurately reconstruct the indicator function of M, and then to approximate its surface $\partial\mathbf{M}$ by a smooth triangulated iso-surface. Given a flexible, enclosing level-set surface $\Phi = \{\mathbf{x} \,|\, \phi(\mathbf{x}, t) = 0\}$, we formulate the reconstruction problem as the convection of Φ in the velocity field $\mathbf{V}$ due to the input samples $\mathbf{x}_i$ given by

$$\mathbf{V}(\mathbf{x}) = \sum_{i=1}^{N} w_i(\mathbf{x})\hat{\mathbf{f}}_i(\mathbf{x}), \quad \text{where} \tag{7.2}$$

$$w_i(\mathbf{x}) = \frac{d_i(\mathbf{x})^{-p}}{\sum_{j=1}^{N} d_j(\mathbf{x})^{-p}}, \qquad \hat{\mathbf{f}}_i(\mathbf{x}) = \frac{\mathbf{x} - \mathbf{x}_i}{d_i(\mathbf{x})}, \tag{7.3}$$

with $d_i(\mathbf{x}) = \sqrt{D_i^2(\mathbf{x}) + \epsilon^2}$, $D_i(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_i||$ is the Euclidean distance between $\mathbf{x}$ and $\mathbf{x}_i$, $\epsilon > 0$ is a small softening constant, $N = |S|$ denotes the number of input samples, and p is a constant parameter. Thus, $\mathbf{V}$ is the minimizer of a functional $E(\mathbf{x}, \mathbf{V})$ measuring deviations between tuples of interpolating points $\{\mathbf{x}, \mathbf{V}\}$ and N tuples of interpolated points $\{\mathbf{x}_i, \hat{\mathbf{f}}_i\}$, defined as

$$E(\mathbf{x}, \mathbf{V}) = \frac{1}{2} \sum_{i=1}^{N} d_i(\mathbf{x})^{-p} \left|\left| \mathbf{V} - \hat{\mathbf{f}}_i \right|\right|^2. \tag{7.4}$$

Passive convection in the velocity field $\mathbf{V}$ and flexibility of the level-set surface Φ can be obtained using Eq. (7.1), and setting $F(\mathbf{x}) \equiv 0$, $\mathbf{U}(\mathbf{x}, t) \equiv \mathbf{V}(\mathbf{x})$ and $\alpha > 0$. Note that the

Algorithm 7.3 Multi-resolution surface reconstruction.

1: Initialize level-set surface Φ to a box at resolution d
2: Construct octree $\mathcal{O}$ with maximum resolution D
3: Compute centroid and number of samples of each node in $\mathcal{O}$
4: Evolve level-set surface at resolution d
5: **for** $r = d + 1$ to D **do**
6: Upsample level-set surface to resolution r
7: Evolve level-set surface at resolution r
8: **end for**
9: Output is the final reconstructed surface Φ at resolution D.

velocity field $\mathbf{V}$ has to be evaluated at all locations in the narrow band, not only at Φ, such that *extending the velocity* to all level sets (within the narrow band) is avoided.

7.4.1 Efficiency and multi-resolution

Letting $P(\mathbf{x}) = \sum_{i=1}^{N} d_i(\mathbf{x})^{-p+1}$ and assuming $p > 1$, $\epsilon > 0$, it can be shown by the triangle inequality that $\mathbf{V}(\mathbf{x}) = c \cdot (-\nabla P(\mathbf{x}) / ||\nabla P(\mathbf{x})||)$, where $0 < c < 1$. Thus since $-\nabla P(\mathbf{x})$ and $\mathbf{V}(\mathbf{x})$ have the same directions and $-\nabla P(\mathbf{x}) / ||\nabla P(\mathbf{x})||$ is a unit vector, convecting the level set function towards the sample points by taking unit-size steps is optimal. Therefore, evaluating $\mathbf{V}$ for an active tile can be done as follows. First, the potential P is efficiently approximated on the CPU using the Barnes-Hut algorithm [8] similar to [56]. Then, the computed *scalar* values are passed back to the GPU, where the normalized gradient of P is evaluated using central differences to yield $\mathbf{V}$, during the tile computation step.

The following simple *caching scheme* on the GPU was devised to improve efficiency and minimize CPU – GPU memory transfers. An additional array is used on the GPU, indexed by *tile id* (see subsection 7.3.3), that stores the P values of the currently active tiles. Whenever new tiles are created on the GPU during the tile-management step, their corresponding pages (containing tile coordinates and identifiers) are sent to the CPU, so that the CPU threads can start immediately evaluating P at the required locations. After the CPU computations terminate, the P values of the new tiles are transferred to the GPU, so that it can continue with updating the level-set function. When inactive tiles are removed, their storage is easily reclaimed and reused, as the array is indexed by tile identifier.

A simple *multi-resolution* scheme was also deployed, so as to further improve the efficiency of the method. That is, instead of convecting the level-set surface at the highest resolution, we successively evolve it at gradually increasing resolutions, see Algorithm 7.3. After initializing the level-set surface to a box at a small, starting resolution d (line 1), the octree $\mathcal{O}$ required by the Barnes-Hut algorithm [8] is built (line 2). Given a maximum depth D (corresponding to a grid of size 2^{3D} voxels), the octree $\mathcal{O}$ is built top-to-bottom, in an attempt to allocate one sample point per octree leaf. If upon insertion of a new sample point, a leaf at depth D is reached that already contains a sample, both samples are discarded and replaced by their centroid. In the process, the number of samples contained by each leaf is also stored. After the tree is built, the centroid and

number of samples is computed for each octree node, by propagating information from leaves towards the root node (line 3). Next, the level-set surface Φ is convected at the starting resolution d. Following the Barnes-Hut algorithm, to evaluate P at a location $\mathbf{x}$ in the narrow band, octree nodes are traversed in depth-first order. If $\mathbf{x}$ is far from the centroid of a given node n, P is computed using the total number and centroid of the samples in n. Thus, instead of computing *all* contributions of the samples in n, only one contribution due to all samples within the node is considered. Otherwise, if $\mathbf{x}$ is close to centroid of node n, the traversal continues with the child nodes of n.

Convergence of the level-set surface at any resolution $r = d, \ldots, D$ is automatically detected, as follows. For each active tile, an 8-bit variable is stored, signifying the "age" of the tile. During the tile-management step, this variable is incremented for each tile that remains active during the next time step, whereas for newly-added tiles it is set to zero. When the age of all active tiles is larger than a given value, the level-set surface is assumed to have converged. This criterion is efficiently implemented in CUDA using a reduction primitive.

Before advancing to the next resolution, the list of active tiles has to be *upsampled* (line 6). Upsampling the narrow band is accomplished in two steps:

1. Create a new active list using Algorithm 7.1, by replacing each active tile by 8 new tiles. The coordinates of the new tiles are set to $\mathbf{x}' \equiv 2\mathbf{x} + [b_0, b_1, b_2]$, where $b_0 b_1 b_2$ is the binary representation of $c = 0, \ldots, 7$, the index of the newly-created tile. The tile data of the newly-created tiles are computed by trilinear interpolation of the initial data.
2. Sort the resulting active list in lexicographic order, using the radix-sort algorithm [113].

The first step is implemented in CUDA in only one compute pass, since each thread working on one initial tile generates a constant number of tiles in the new active list. Note that, although the new active list is eight times larger than the initial one, after the first tile-management step, usually the number of active tiles is halved.

To evolve the level-set surface Φ at resolution $r = d, \ldots, D$, given the octree $\mathcal{O}$ built at resolution D, we proceed as follows. When evaluating P at location $\mathbf{x}$, at resolution r, we in fact evaluate it at location $\mathbf{x}'' \equiv 2^{D-r}\mathbf{x}$. Moreover, we limit the maximum depth during the octree traversal to r, *i.e.*, instead of visiting children of nodes at level r, their centroid and number of samples are used.

At the end of the algorithm, the final reconstructed surface is given by the level-set surface Φ evolved at resolution D (line 9).

7.5 Comparison with previous approaches

In this section we make a methodological comparison with previous approaches which were reviewed in Section 7.2.

Our GPU sparse level-set method is similar to that of Lefohn et al. [68], in that it uses small, fixed-size blocks, *i.e.*, tiles, to represent the narrow band around the interface. However, in contrast to Lefohn's method, we do not need to store a map of the complete domain nor have we to maintain a list of neighbours for each tile. Furthermore, the complex paging mechanism

involved by Lefohn's method is avoided altogether, and updating the active tiles is a simple list traversal. Moreover, our method is not bound to a fixed domain. Instead, it allocates and de-allocates new tiles as the interface propagates to accommodate the deformations. Another difference is that in our method the GPU handles the tile management step; only a very small data structure (16 bytes) has to be transferred to the CPU in every iteration so that the CPU can check whether the tile list is large enough, or has to be resized.

Our approach using fixed-size tiles fits very well the computational model of CUDA. By contrast, the DT-Grid [89] requires potentially different handling of every voxel, and furthermore it relies on more complex iterator structures specific for every neighbouring voxel, which would result in more registers being used in a CUDA implementation. Additionally, to implement in CUDA the 'push' operation used to insert grid points to the DT-Grid data structure, one has to compare the last inserted coordinate to the current one and execute potentially different code consisting of write operations in a random-access fashion. This also means that in a parallel CUDA implementation, the merging step would be more complex. By contrast, in our approach similar to the STL method, implementing the same operation is trivial, as all one has to do is simply append the new tile to the active list. The DT-Grid needs more complex steps to maintain the narrow band, whereas the STL method requires one tile-management step that updates the active list in linear time [144]. Finally, when rebuilding the tubular grid, the entire current domain is dilated, while in the STL method only the tiles which are active at the next time step are added to the active list. Note that since the hierarchical RLE level set method of Huston *et al.* [54] is based on the DT-Grid enhanced with RLE compression, the resulting data structure is even more complex and thus even more difficult to parallelize efficiently. In our tile-based method, access to neighbouring tiles has a similar pattern (*i.e.*, all GPU threads execute similar operations), which results in very fast parallel execution. Moreover, tiles can be read or written using coalesced access operations, which is desirable in CUDA to achieve maximum performance [91]. Thus, the proposed tile-based data structure further maximizes the potential for parallelism, compared to other state-of-the-art level set representations.

Although the STL requires about two times as much memory as the DT-Grid, it was shown that this method is about nine times faster than the latter [144]. The increased memory usage of the STL method stems from the fact that the narrow band is tile-based, see Fig. 7.1. Note that, as the DT-Grid was inspired by the compressed-row layout for representing sparse matrices, the STL and our GPU methods are similar to the compressed-block layout. However, since values in the narrow band are in a very narrow range, *i.e.*, $(-1.5, 1.5)$ when $\gamma = 1.5$, fixed-point representations on 16-bits can be used to store single-precision ϕ values, thus improving the memory usage by almost a factor of two, see subsection 7.6.2.

The proposed multi-resolution method for surface reconstruction is related to [161] in that we rely on the level set method to represent the approximating surface, which unlike [161] is convected in an inverse-distance velocity field based on Shepard interpolation [119]. The field is evaluated on the fly using the "tree algorithm" [8], see Section 7.4 and [56].

Unlike other modeling approaches based on level sets, *e.g.*, [88], our surface-editing method does not need first to scan-convert the edited models, nor does it require performing volumetric distance calculations. Therefore, our editing method is not bound to a given grid size, and since it relies on the proposed sparse, tile-based representation of the narrow band, it is highly efficient,

allowing interactive frame rates on large, volumetric grids.

7.6 Results

In this section we present experimental results obtained by the proposed methods. All experiments were performed on a machine equipped with an Intel Core 2 Quad CPU at 2.4 GHz, 6 GB RAM and a GeForce GTX 280 GPU (1 GB).

7.6.1 Efficiency: Comparison to other methods

We performed a direct comparison of our GPU level set method with state-of-the-art, sparse counterparts running on the CPU. The parameters of all level set methods were set to $F(\mathbf{x}) = 0.1$, $\alpha = 1$, $\mathbf{U}(\mathbf{x}, t) = 0$, such that the interface collapses to a point, mostly due to motion with speed proportional to its curvature, see Eq. 7.1. In all cases, the initialization was the surface of the Lucy model (see subsection 7.6.2) reconstructed by our method (Section 7.4) on a 1024^3 grid; the initial number of active tiles was $127,535$. The average timings per iteration in five runs, of the STL method (SSE-optimized and plain), DT-Grid and our GPU method are shown in Fig. 7.6. As can be seen, all methods show similar performance patterns, implying that our GPU method is (in practice) *work efficient*. Only after $4,000$ iterations, when the number of active tiles becomes smaller than $10,000$, our GPU method becomes slightly less efficient, which is to be expected as the GPU compute resources are not fully used. Note that all methods converge in the same number of iterations (5830), indicating that the evolutions of the surfaces were the same.

Table 7.1. *Final timings and overall performance for both STL methods (SSE and plain), DT-Grid and our method.*

Method	Timing (s)	Speedup
DT-Grid	$7,051$	1.0
STL (plain)	$1,506$	4.7
STL (SSE)	616.0	11.4
Our method	29.5	239.0

Final timings of the simulation and speedups with respect to the slowest method (DT-Grid) are given in Table 7.1. Accordingly, our GPU method is about 20 times faster than the SSE-optimized STL method and two orders of magnitude faster than the DT-Grid method. The memory requirements of our GPU method are similar to those of the STL method, which in turn are about 2.5 times larger than those of the storage-optimal DT-Grid. However, in section 7.6.2 we show that the memory footprint of our method can almost be halved.

Since a direct comparison between GPU particle (marker) level sets [27, 83] and a pure level set method cannot be done, due to the additional computations involved by the first methods, we just indicate that their computational complexities scale with the size of the grid, while ours

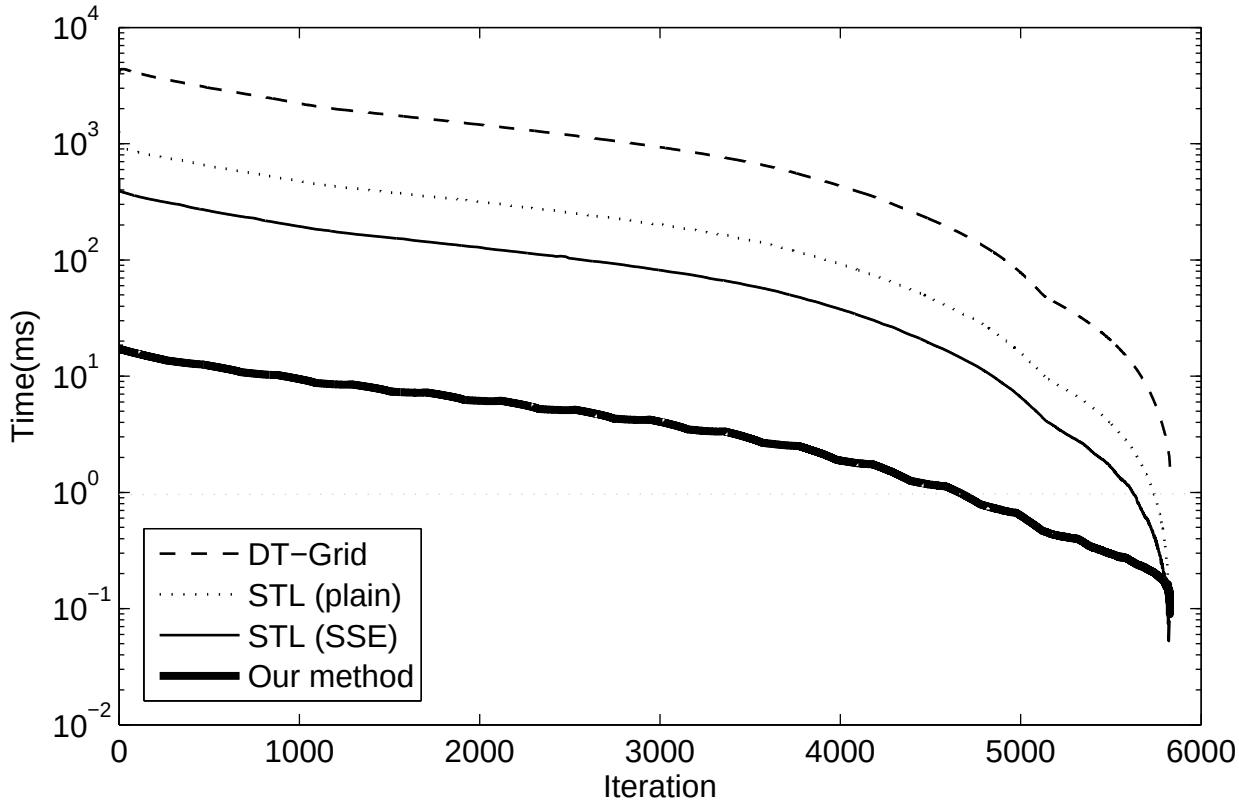


Figure 7.6. *Efficiency. Logarithmic plot: time (ms) per iteration for the STL method (SSE-hand optimized and plain), DT-Grid and our GPU method. Initial surface was the Lucy model reconstructed by our method on a* 1024^3 *grid.*

scales with the size of the level-set surface. Moreover, particle (marker) level-set methods perform at about 20 fps on small, 128^3 grids, whereas our method runs at interactive rates on substantially larger (1024^3) grids, see subsection 7.6.3.

7.6.2 Surface reconstruction

Throughout all our surface reconstruction experiments, we set $p = 3$ in Eq. (7.2), and use *flat shading* when rendering the reconstructed models, so as to emphasize the smoothness of the surfaces delivered by our method.

According to our discussion from Section 7.4, the proposed method for surface reconstruction delivers multi-resolution representations at increasing grid resolutions. In the first experiment, the octree depth was set to $D = 10$, and we start the reconstruction process at level $d = 7$ from a cube surrounding the Armadillo model, see Fig. 7.7. The evolution of the level-set surface Eq. (7.1), is steered using $F(\mathbf{x}) = 0$ and $\alpha = 0.1$, where $\mathbf{U}$ is evaluated on the GPU from inverse-distance potentials, see subsection 7.4.1. It took 700 iterations for the level-set surface

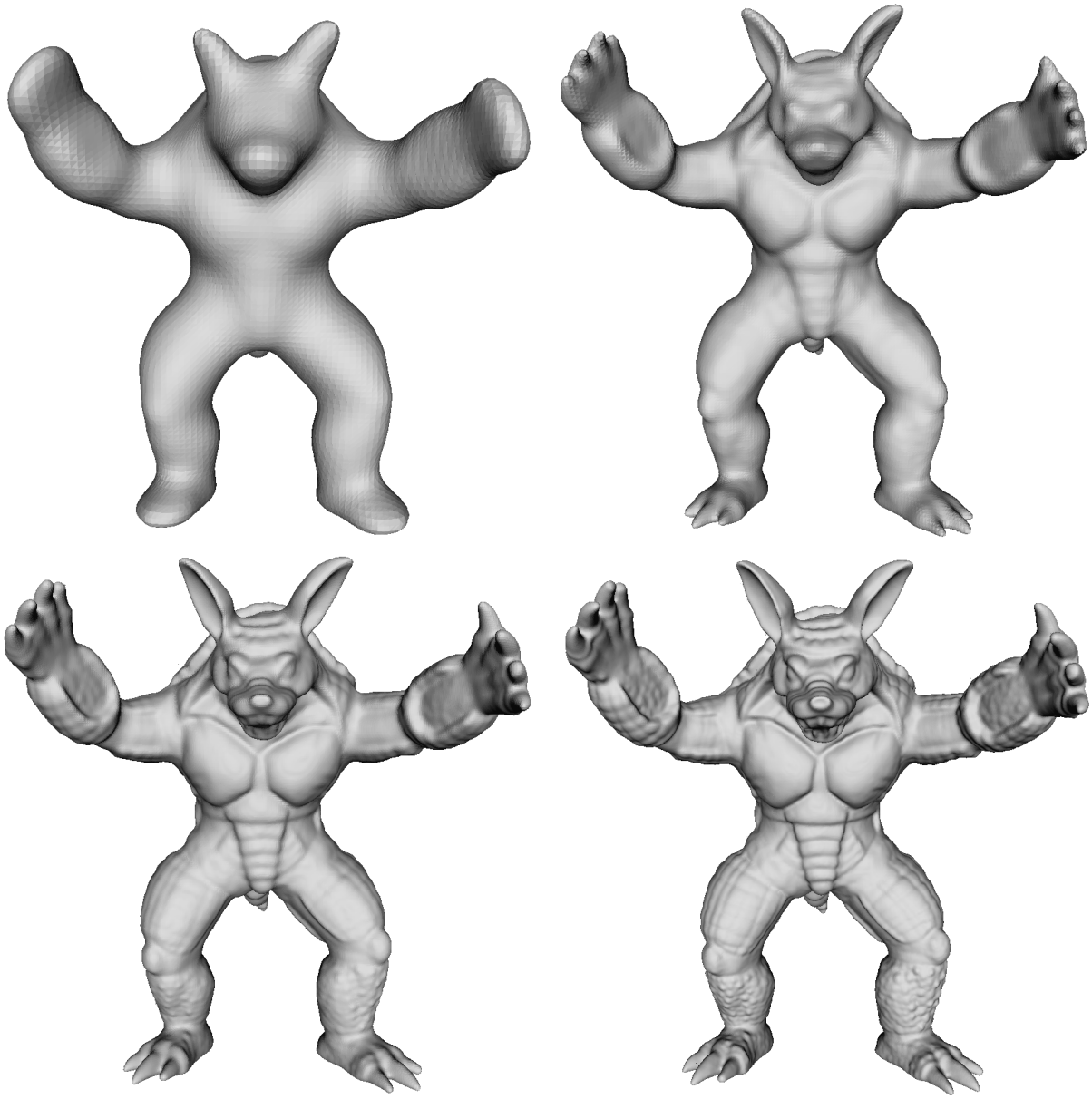


Figure 7.7. *Multi-resolution surface reconstruction.* Left-to-right, top-to-bottom*: octree depths* $d = 7, 8, 9, 10$.

to converge and reconstruct the Armadillo model at level $d = 7$. Before advancing to the next resolutions, 10 curvature flow iterations were used to produce the models in Fig. 7.7. Further, at each resolution $r = 8, 9, 10$, less than 100 full iterations were necessary for the level-set surface to converge. Since directly evolving the level-set surface at the maximum resolution $D = 10$ would require well in excess of $1,000$ $(700 + 3 \times 100)$ iterations, which are also more computationally expensive, our choice of using a multi-resolution scheme is, in our opinion,

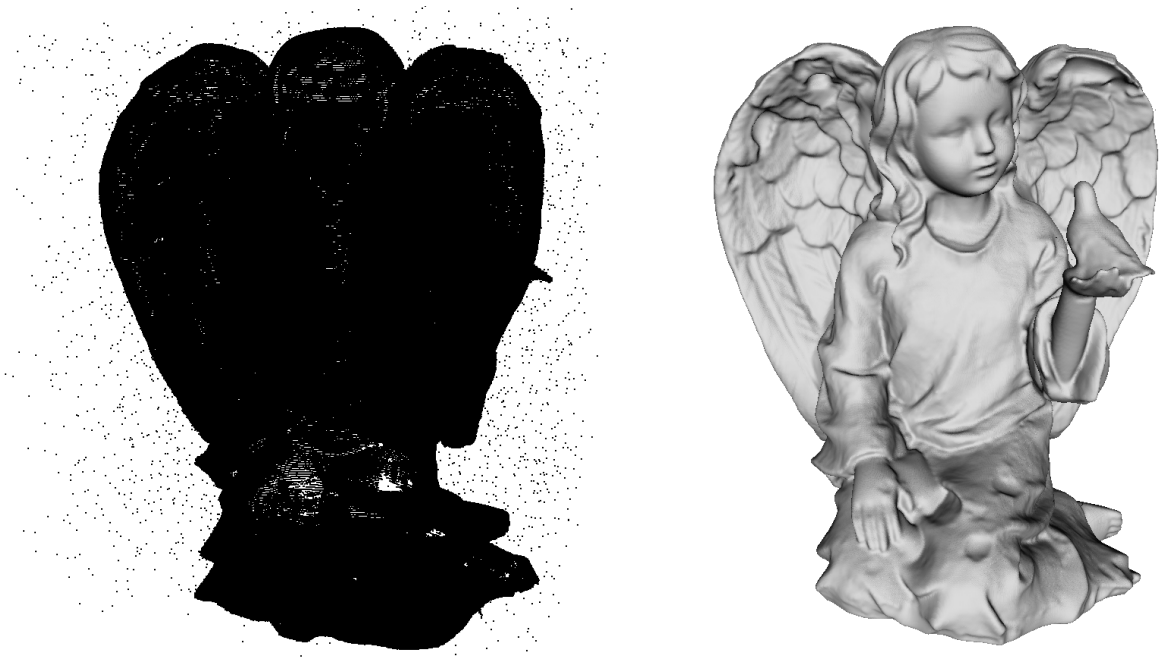


Figure 7.8. *Noise behaviour.* Left*: noisy range data with* $2 \cdot 10^6$ *samples and* $4,000$ *outliers.* Right*: reconstructed surface,* $D = 10$.

Table 7.2. *Reconstruction statistics for the Thai-statue model (*5 *million samples). Octree construction time (at level* 12*) was* 13 *seconds (s), the peak memory usage was* 860 *MB on the GPU and* 2.5 *GB on the CPU.*

Octree depth d	Number of iterations	Number of tiles	GPU time(s)	CPU time(s)	Total time(s)
7	704	1,724	1.3	0.9	2.2
8	297	7,127	0.4	1.8	4.4
9	128	38,425	1.3	4.4	10.1
10	107	138,812	2.2	18.2	30.5
11	101	557,030	11.6	80.8	122.9
12	100	2,246,782	60.2	323.2	**506.3**

justified.

The proposed method for surface reconstruction withstands large numbers of outliers, due to its reliance on inverse-distance potentials [56]. Moreover, increasing the stiffness of the interface, by adjusting the curvature term in Eq. (7.1), allows it to bypass outlier locations during its evolution, see Fig. 7.8.

Our method allows reconstruction of large models on octree grids of up to $2^{11\times3}$ voxels,

Figure 7.9. *Reconstruction of two models (Thai statue and Lucy) at very large resolutions (octree level $D = 12$).*

i.e., the maximum octree depth is set to $D = 11$. At larger grid resolutions, the storage requirements of the narrow-band on the GPU become larger than the GPU memory. However, by trading accuracy for storage space we can push the maximum resolution to one octree-level higher, without introducing any visible artifacts. Since within the narrow band the level-set function satisfies $|\phi| < 1.5$ (assuming $\gamma = 1.5$), a *fixed-point* representation on 16-bits is used to

Table 7.3. *Reconstruction statistics for the Lucy model (*14 *million samples). Octree construction time was* 17 *seconds (s), the peak memory usage was* 800 *MB on the GPU and* 1.8 *GB on the CPU.*

Octree depth d	Number of iterations	Number of tiles	GPU time(s)	CPU time(s)	Total time(s)
7	602	1,320	1.1	0.8	1.9
8	210	6,691	0.2	0.4	2.5
9	180	35,956	1.1	2.8	6.4
10	120	127,537	2.7	13.7	22.8
11	112	408,362	10.1	62.1	95.0
12	114	2,047,288	58.2	279.2	**432.4**

store its 32-bit, single precision values. Similarly, after clamping P values (see Section 7.3.1) in the range $(0, 10^3)$ and adaptively-compressing them using a logarithmic function, such that more precision is used towards the maximum end of the range, the resulting values are stored again using a fixed-point representation. This simple storage scheme reduces the overall GPU memory footprint by almost a factor of two, thus allowing us to reconstruct large models at even higher resolution grids. The results of two such experiments are shown in Fig. 7.9, and some statistics about the overall reconstruction process is given in Tables 7.2 and 7.3. As can be seen, with increasing resolution the size of the narrow band becomes about four times larger than that at the previous resolution, resulting in about the same penalty factor at which the speed of both CPU and GPU computations decreases. Thus, at any resolution, both CPU and GPU computational requirements *scale* with the size of the interface, which makes our reconstruction method highly efficient. For comparison, the recent method in [56], which was considerably faster than other recent approaches, reconstructs both the Thai-statue and Lucy models at octree depth $D = 11$ in 28 and 21 minutes, respectively. Our method needs only 9 and 8 minutes, respectively, to reconstruct both models at depth $D = 12$, which makes our method at least one order of magnitude faster. To mention another comparison, the parallel Poisson method for surface reconstruction yields the Lucy model at depth $D = 12$ on a distributed-memory cluster with 12 processors in 17 minutes [10], at the expense of large data replication across the three workstations constituting the cluster.

7.6.3 Interactive level-set surface editing

Finally, we extended our GPU level-set framework to accommodate free-form level-set surface editing operators [88]. The following operators are currently available:

- Global *morphological operators* (erosion, dilation, opening and closing) [114]. For example, the erosion operator is easily obtained by setting $F(\mathbf{x}) = 1$ in Eq. (7.1).
- Local, *weighted, morphological operators*. For example, a Gaussian-weighted, local dilation

operator is performed by setting

$$F(\mathbf{x}) = \begin{cases} f_0\, G_\sigma(d(\mathbf{x})), & d(\mathbf{x}) \leq 3\sigma \\ 0, & d(\mathbf{x}) > 3\sigma, \end{cases} \tag{7.5}$$

with G_σ a Gaussian kernel of width σ, f_0 a positive constant, and $d(\mathbf{x})$ the Euclidean distance between location $\mathbf{x}$ of the narrow band and $\mathbf{x_m}$, the center of the local manipulator widget.

- *Selection* is performed in three steps. First, the tiles intersecting the selection box are found using binary search within the list of active tiles. Second, at their locations, the object is disconnected using local erosion operators. Finally, tiles within the selection box (forming one or more disconnected objects) are selected.
- *Pasting* consists in simply adding tiles representing selected objects to the active list. Here we make the assumption that the tile sets of the pasted objects are disjoint.
- *Deletion* removes selected tiles from the active list.
- *Rotation* around an arbitrary point is obtained by convecting the level-set surface representing the selected object in a rotational velocity field implementing the desired rotation. The fifth-order accurate HJ-WENO scheme is used, see subsection 7.3.1.
- *Translation* is performed by manipulating tile coordinates.
- *Point set attraction and repulsion*. Given a point set, a velocity field is constructed, which either attracts or repels the level-set surface.
- *Local and global smoothing* operators. Global curvature smoothing is obtained by using only the curvature term in Eq. (7.1). Local smoothing is implemented by constraining global smoothing to a Gaussian neighbourhood.

Figure 7.10 shows a subset of our surface-editing operators, while constructing a double-headed dragon model. First, the head of the dragon is *selected*, saved on disk, and then *deleted*. In a new modeling session, the head of the dragon is *selected* and duplicated by *pasting*. Then, both heads are *rotated* and *pasted* to the previously-saved dragon body. After the two heads are glued using *local morphological operators* and *attractors*, the resulting model is *smoothed*. The initial dragon model was obtained by surface reconstruction (Section 7.4), on an equivalent 1024^3 voxel grid. The minimum frame rate we observed during the manipulation of the level-set surface was about 20 *fps*.

7.6.4 Limitations

Our current method has a number of limitations which we want to address in future work.

- Currently, the entire dataset has to be kept in GPU memory, i.e., there is no out-of-core support. We could however, use our convergence criterion to remove from GPU memory those tiles in which the level-set method already converged, to make space for new tiles.
- Merging two (unsorted) tile lists requires a re-sorting of the resulting list. On the other hand, if the two lists are already sorted, this can be done in a less expensive merge pass.
- Tile deletion requires a pass over the entire active list. For this purpose, using a hierarchical structure or a hash table would be more efficient.

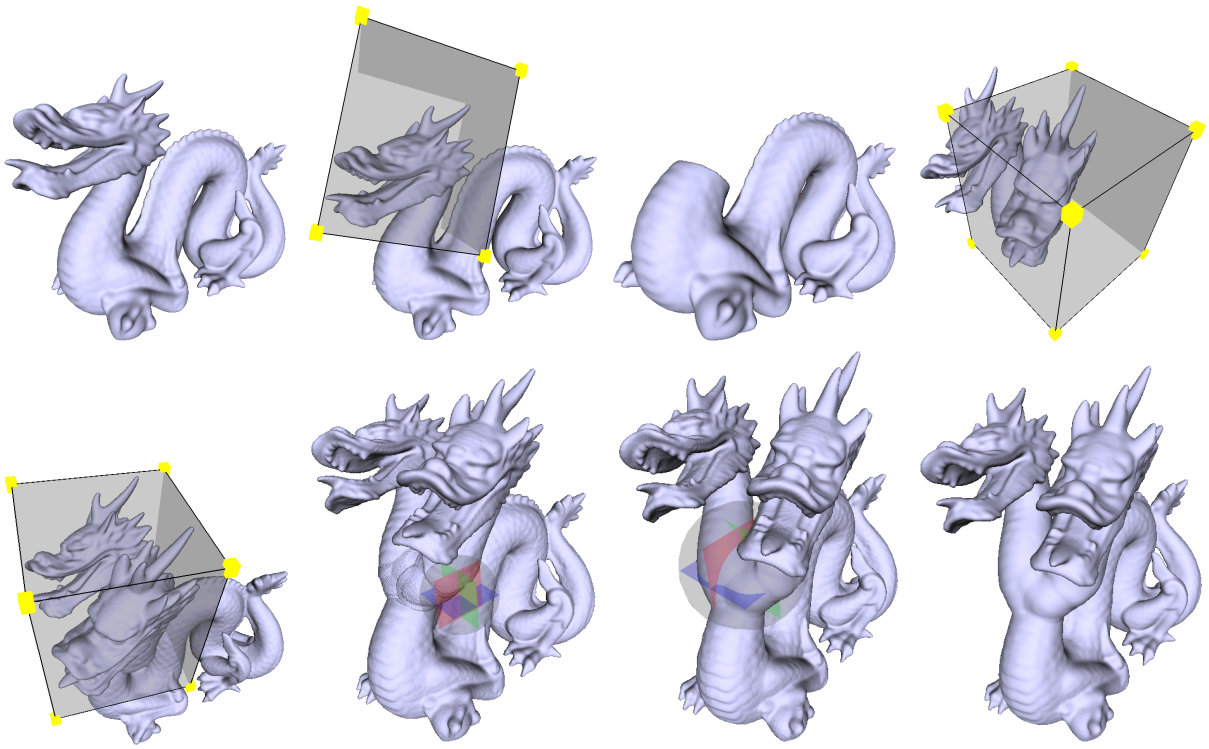


Figure 7.10. *Editing the Dragon model. First, the head of the dragon is cut off, pasted, rotated and glued back onto the body. Local and global smoothing operations are then applied to produce the final double-headed dragon.*

7.7 Conclusions and future work

We have presented a highly-efficient, sparse, tile-based level set method, which runs entirely on commodity graphics hardware. We compared our method to other state-of-the-art, sparse approaches, and have shown that ours is about 20 times faster than the optimized CPU version of the Sorted Tile List method, and two orders of magnitude faster than the DT-Grid method.

Many level-set applications can benefit from our level-set GPU infrastructure. To demonstrate its efficiency, we discussed two graphics applications: surface reconstruction from point clouds and level-set surface editing. Our novel multi-resolution method for surface reconstruction compares favorably with recent, existing techniques and parallel implementations. Finally, our free-form surface editing tool runs at interactive frame rates on large volumetric grids of 1024^3 voxels.

In future work, we shall investigate the possibility of extending our GPU level-set framework to accommodate the particles of the Particle/Marker level-set method. Moreover, work is in progress to adapt our data structure to implement simulations based on the Smoothed Particle Hydrodynamics (SPH) method. Finally, we plan to combine the GPU octree implementation of [163] with our GPU level sets, across a number of GPUs, to achieve very efficient, scalable parallel surface reconstruction.

Chapter 8

Concluding remarks

8.1 Summary and Conclusions

In this thesis we investigated several advanced techniques in visualization of large data sets, multidimensional signal processing, deformable models, and data reduction. As we aimed to develop fast algorithms, all of these can be used in an interactive pipeline.

In chapter 2 we have investigated a number of algorithms based on morphological pyramids for multiresolution MIP volume rendering on graphics hardware. We found that our highly-optimized streaming MIP GPU-method outperforms both its software implementation as well as existing ray-casting and 3-D texture-based methods.

In chapter 3 we presented a novel, fast wavelet lifting implementation on graphics hardware using CUDA, which extends to any number of dimensions. We compared our method to an optimized CPU implementation of the lifting scheme, to another (non-CUDA based) GPU wavelet lifting method, and also to an implementation of the wavelet transform in CUDA via convolution. We implemented our method both for 2D and 3D data. The method is scalable and was shown to be the fastest GPU implementation among the methods considered. Our theoretical performance estimates turned out to be in fairly close agreement with the experimental observations. The complexity analysis revealed that our CUDA kernels are cost- and work-efficient. Our proposed GPU algorithm can be applied in all cases were the Discrete Wavelet Transform is part of a pipeline for processing large amounts of data. Examples are the encoding of static images, such as the wavelet-based successor to JPEG, JPEG2000 [123], or video coding schemes [9].

In chapter 4, we showed how to accelerate the Dirac Video Codec by our fast wavelet lifting implementation on graphics hardware using CUDA. We also accelerated the motion compensation and frame arithmetic stages of this codec. The experiments on high definition video sequences have demonstrated that one can achieve a speedup factor of more than 7 for the entire decoding process including the CPU steps, and a factor of 15 for just the GPU part. In our benchmark we could play back a 1080p resolution Dirac video sequence at roughly 50 frames per second on basic consumer hardware.

In chapter 5 we presented a new method for rendering fluids in real-time directly from particle based representations without the need for intermediate triangulation, but which still produces a high-quality fluid surface. We also introduced new ideas to add thickness-based shading and small-scale surface detail to fluids.

In chapter 6 we proposed an efficient data structure, the Sorted Tile List, with associated operations for the level set representation, and compared the resulting method with the current method of choice, the DT-Grid method. With regard to performance, given the same numerical simulation code, our method turned out to be faster by a significant factor. After fine-grain parallelization using SIMD instructions our method was shown to be roughly 8 times faster.

In chapter 7 we adapted our highly-efficient, sparse, tile-based level set method to leverage highly parallel architectures such as GPUs We compared our method to other state-of-the-art, sparse approaches, and showed that our method is about 20 times faster than the optimized CPU version of the Sorted Tile List method, and two orders of magnitude faster than the DT-Grid method. Many level-set applications can benefit from our level-set GPU infrastructure. To demonstrate its efficiency, we discussed two graphics applications: surface reconstruction from point clouds and level-set surface editing. Our novel multi-resolution method for surface reconstruction compares favorably with recent, existing techniques and parallel implementations. Finally, our free-form surface editing tool runs at interactive frame rates on large volumetric grids of 1024^3 voxels.

8.2 Future outlook

8.2.1 GPUs

In the land of GPUs, things change fast, very fast. This makes it very difficult to say something about the future which is not outdated already. Certain is that slowly but steadily the typical GPU restrictions are being removed. Several limitations existed when this thesis project was started: 3D floating-point textures were not available, rendering directly into a texture was not fully implemented, writing to multiple (output) buffers was not yet allowed, and instruction sets were limited. These obstacles were removed in the course of perhaps not more than a year. With the advent of CUDA, which was introduced after the work on MIP rendering with morphological pyramids was completed, even more limitations of traditional GPGPU programming disappeared.

Other limitations have been alleviated but remain a limiting factor in performance. For example, the NVidia Tesla (G80) architecture did not support any sort of caching for global memory, so the programmer had to rely on optimized memory access patterns to reach a significant throughput. The Fermi (GF100) [93] generation added a cache hierarchy, and relaxed the constraints on memory access patterns. The cache comes at a price, as part of shared memory is traded for it. Even with caching, the maximum throughput is achieved by using an optimized access pattern, so the results described in this thesis remain important.

Lifting such architectural restrictions makes the GPU stream processors become more complex and more like CPU cores. On the other hand, CPUs increasingly include more GPU-like features (e.g., instruction sets for exploiting inherent parallelism, or an increasing number of cores). It is probable that convergence will occur, although it is not yet clear what the end result should be. GPU vendors should be careful not to generalize too much, as the strength of GPGPU computing lies in massive parallelism with simple execution units.

8.2.2 Computer Graphics APIs

As graphics cards grow toward full programmability and generality, APIs such as OpenGL/DirectX will, for graphics programming, probably be overtaken by more convenient higher level graphics APIs (such as OGRE [**?**]), which allow full programmability and extensibility under the hood, by making use of GPGPU APIs such as OpenCL [137]. As these libraries are designed with the user in mind, they might rely on abstractions such as Renderman [5] that are used in 3D rendering for motion pictures.

If programmable GPUs mature like CPUs, which they will probably do, there will be multiple programming languages to choose from, but the interface to the hardware will be standardized by the operating system. As this will provide the low level interface, the graphics APIs lose their status as low level interface to the hardware and become intermediate level interfaces. Will there still be a place for them?

OpenGL started out as a complete set of rendering commands for the professional graphics hardware. DirectX started as a light programming interface for customer graphics hardware. Both have evolved enormously since, and have long ago converged with respect to capabilities. At their heart, they both have the triangle rasterization-oriented graphics pipeline. With programmable hardware, the full graphics pipeline has been implemented in software [72] which is much more flexible. In many cases, the focus on a rigid rendering pipeline only gets in the way. For example when implementing advanced rendering techniques (such as ray-tracing [120], voxels [26], or irregular volume rendering [162]) using complex data structures, one does not want to worry about the triangle rasterization state.

Also, the graphics APIs have their own specific problems, that might make them fade out of scope eventually. OpenGL suffers from a very slow political decision process, which results in vendors bolting on their own hardware specific extensions. This makes it very complex for users, which have to cope with all the combinations of extensions and hardware details. DirectX has the opposite problem, its decision process is fast and pragmatic, and thus its programming interface is changed wildly in every major release. However, it is inefficient for programmers to have to learn a new API every two years, and an application needs to be able to make use of two or three versions to be able to support older hardware. Also, it is restricted to the MS Windows operating system, which has a large installed base, but is not quite the whole story nowadays, especially in the growing mobile realm.

8.2.3 CUDA

CUDA still has a few user (developer) friendliness issues that hamper its adoption. Although currently used by companies in oil and gas and finance, and at many universities, far outside the scope of computer graphics and games where it began, adoption of CUDA would be helped by finding ways to integrate GPGPU into day to day experimentation. Some of these issues are the following.

Ease of programming Even though GPUs can be programmed in C nowadays, a lot of hardware pitfalls exist which make that code which is not specifically optimized runs slower than the

CPU implementation. Specific techniques that take knowledge of the hardware into account are needed, especially since the hardware continuously changes. The NVidia Fermi [93] architecture takes a step in the right direction by adding cache and relaxing the constraints on memory access patterns. This makes it easier for a programmer to write moderately efficient code. The authors of CUDA-lite [141] have developed a tool to simplify CUDA programming by helping the programmer to deal with the complex memory hierarchy. Another programming issue is the difficulty of writing shared memory parallel code. Programmers tend to think in blocks and objects communicating with each other, for example using MPI [125]. The GRAMPS [129] programming model is an interesting development in this direction, as it considers the GPU as a general set of stages connected by queues.

Debugging It has always been very difficult to debug GPU code. There used to be no support for single-stepping through code or using breakpoints, and also no way to log simple tracing messages. The only possibility of debugging was to write to some memory area or texture, and inspect this result from the CPU. This has been addressed with the introduction of the CUDA GPU debugger in CUDA 3.0, which does offer those features. Also, there has been research to make debugging GPU code more convenient, see Hou *et al.* [53]. However there is a general problem in debugging multi-threaded code: conventional debugger paradigms were invented with single-threaded usage in mind, and hardly consider thread communication and synchronization, which are exactly the areas that tend to introduce most bugs.

Scalability With so-called cloud computing (general-purpose utility computing clusters) on the rise, scalabilty is more important than ever. Recently, Amazon EC2, a popular cloud-computing platform, introduced a new node type with two GPUs. Everyone can now hire a full-blown GPU cluster for a few hours for a relatively low cost.

Utilizing this cluster is another story, though. Many of the challenges in GPU cluster building, management and programming are addressed in [64]. However, one issue remains: the use of clusters adds yet another level to the already complex GPU execution and memory hierarchy. It would be most efficient if the programmer could write code that runs on clusters as well as single and multiple GPUs of various sizes, without having to worry about the different levels of parallelism and data distribution between nodes, except when fine-tuning.

Library support There is always a demand for more readily usable general purpose and application-specific software libraries. Libraries currently provided by NVidia are “cublas” for linear algebra, “cufft” for FFT transforms, and “cudpp” for generic parallel primitives. Third-party libraries have also been developed, such as OpenVIDIA [37] for computer vision.

8.2.4 GPGPU for embedded systems

In my opinion a great, mostly open opportunity is the use of GPGPUs in embedded systems, such as intelligent vehicles that process large amounts of incoming sensor data, cameras that recognize faces, and other upcoming “smart devices”. GPUs excel at fast signal processing, for

adaptive controller systems, artificial intelligence, pattern matching, and so on. However, there are still a few issues with this.

Noise and heat GPU boards have the reputation of producing much heat which has to be cooled with fans which produce a large amount of noise. Also, even though the performance-to-watt ratio of GPUs compares very favorably to CPUs, they use a lot of power, also when (partially) inactive. The latter is being addressed with on-chip power saving techniques.

Platform openness GPUs are hard to embed due to closed platforms and drivers; there are no drivers for platforms generally used in embedded systems such as MIPS or ARM. Also, only a few operating systems are supported.

I/O capability There is currently no direct I/O possibility except to and from the host system (and the graphics output). Direct input from sensors and output to peripherals would be useful. It is an interesting open question how to best integrate the mostly serial nature of external I/O with the parallel nature of GPU programming.

Bibliography

[1] T. Acharya and C. Chakrabarti. A survey on lifting-based discrete wavelet transform architectures. *J. VLSI Signal Process. Syst.*, 42(3):321–339, 2006.

[2] D. Adalsteinsson and J. A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, 1995.

[3] B. Adams, T. Lenaerts, and P. Dutre. Particle splatting: Interactive rendering of particle-based simulation data. Technical Report CW 453, Department of Computer Science, K.U. Leuven, July 2006.

[4] M. Angelopoulou, K. Masselos, P. Cheung, and Y. Andreopoulos. Implementation and comparison of the 5/3 lifting 2d discrete wavelet transform computation schedules on FPGAs. *J. Signal Process. Syst.*, 51(1):3–21, 2008.

[5] A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[6] V. Aurich and J. Weule. Non-linear gaussian filters performing edge preserving diffusion. In *DAGM-Symposium*, pages 538–545, 1995.

[7] C. Auyeung, J. J. Kosmach, M. T. Orchard, and T. Kalafatis. Overlapped block motion compensation. In P. Maragos, editor, *Proc. SPIE Visual Communications and Image Processing '92*, pages 561–572, Nov. 1992.

[8] J. E. Barnes and P. Hut. A hierarchical O(N Log N) force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[9] BBC Research. *Dirac Specification 1.0.0pre7*. Available at http://dirac.sourceforge.net/specification.html.

[10] M. Bolitho, M. Kazhdan, R. Burns, and H. Hoppe. Parallel poisson surface reconstruction. In *International Symposium on Visual Computing 2009*, 2009.

[11] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics*, pages 17–141, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[12] R. Bridson. *Fluid Simulation for Computer Graphics*. A K Peters, 2008.

[13] R. E. Bridson. *Computational aspects of dynamic surfaces*. PhD thesis, Stanford University, Stanford, CA, USA, 2003. Adviser-Ronald Fedkiw.

[14] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[15] P. J. Burt and E. H. Adelson. The Laplacian pyramid as a compact image code. *IEEE Trans. Communications*, 31:532–540, 1983.

[16] A. R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Wavelet transforms that map integers to integers. *Applied and Computational Harmonic Analysis*, 5(3):332–369, 1998.

[17] I. Cantlay. High Speed, Off-Screen Particles. In H. Nguyen, editor, *GPU Gems 3*. NVIDIA, 2007.

[18] S. Chatterjee and C. D. Brooks. Cache-efficient wavelet lifting in JPEG 2000. In *Proc. of the IEEE International Conference on Multimedia and Expo*, pages 797–800, 2002.

[19] J. Chen, S. Paris, and F. Durand. Real-time edge-aware image processing with the bilateral grid. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 103, New York, NY, USA, 2007. ACM.

[20] M. Y. Chiu, K.-B. Lee, and C.-W. Jen. Optimal data transfer and buffering schemes for JPEG 20000 encoder. In *Proc. IEEE Workshop on Design and Implementation of Signal Processing Systems*, pages 177–182, 2003.

[21] D. L. Chopp. Computing minimal surfaces via level set curvature flow. *J. Comput. Phys.*, 106(1):77–91, 1993.

[22] S. Clavet, P. Beaudoin, and P. Poulin. Particle-based viscoelastic fluid simulation. In *Symposium on Computer Animation 2005*, pages 219–228, July 2005.

[23] H. Cords and O. Staadt. Instant liquids. In *Poster Proceedings of ACM Siggraph/Eurographics Symposium on Computer Animation*, Dublin, Ireland, July 2008.

[24] K. Crane, I. Llamas, and S. Tariq. Real-time simulation and rendering of 3D fluid. In H. Nguyen, editor, *GPU Gems 3*, chapter 30, pages 633–675. Addison Wesley, August 2007.

[25] C. Crassin. *OpenGL Geometry Shader Marching Cubes*, 2007. http://www.icare3d.org/blog_techno/gpu/opengl_geometry_shader_marching_cubes.html.

[26] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D 2009, February, 2009*, pages 15–22, Boston, MA, Etats-Unis, 2009. ACM.

[27] N. Cuntz, A. Kolb, R. Strzodka, and D. Weiskopf. Particle level set advection for the interactive visualization of unsteady 3D flow. *Computer Graphics Forum*, 27(3):719–726, May 2008.

[28] I. Daubechies. *Ten Lectures on Wavelets*, volume 61 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[29] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl.*, 4(3):247–269, 1998.

[30] M. Desbrun and M.-P. Gascuel. Smoothed particles : A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation '96*, pages 61–76, 1996.

[31] G. Deslauriers and S. Dubuc. Symmetric iterative interpolation processes. *Constructive Approximation*, 5(1):49–68, dec 1989.

[32] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics (SIGGRAPH '88 proceedings)*, 22(4):65–74, 1988.

[33] M. Droske, B. Meyer, M. Rumpf, and C. Schaller. An adaptive level set method for medical image segmentation. In *IPMI '01: Proceedings of the 17th International Conference on Information Processing in Medical Imaging*, pages 416–422, London, UK, 2001. Springer-Verlag.

[34] D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys*, 183:83–116, 2002.

[35] R. Fedkiw. Simulating natural phenomena for computer graphics. In *Geometric Level Sets in Imaging, Vision and Graphics*, pages 461–479, 2002.

[36] M. Fournier, J.-M. Dischler, and D. Bechmann. 3D distance transform adaptive filtering for smoothing and denoising triangle meshes. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 407–416, New York, NY, USA, 2006. ACM.

[37] J. Fung and S. Mann. Openvidia: parallel gpu computer vision. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, New York, NY, USA, 2005. ACM.

[38] A. Garcia and H. W. Shen. GPU-based 3D wavelet reconstruction with tileboarding. *The Visual Computer*, 21:755–763, 2005.

[39] J. Goutsias and H. J. A. M. Heijmans. Multiresolution signal decomposition schemes. Part 1: Linear and morphological pyramids. *IEEE Trans. Image Processing*, 9(11):1862–1876, 2000.

[40] A. Greß and G. Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proc. 20th IEEE Intern. Parallel and Distributed Processing Symposium (IPDPS)*, pages 25–29, Rhodes Island, Greece, apr 2006.

[41] M. H. Gross, L. Lippert, R. Dittrich, and S. Häring. Two methods for wavelet-based volume rendering. *Computer & Graphics*, 21(2):237–252, 1997.

[42] M. H. Gross, L. Lippert, A. Dreger, and R. Koch. A new method to approximate the volume rendering equation using wavelet bases and piecewise polynomials. *Computers & Graphics*, 19(1):47–62, 1995.

[43] R. Grosso and T. Ertl. Biorthogonal wavelet filters for frequency domain volume rendering. In *Visualization in Scientific Computing '95*, pages 81–95. Springer Verlag, 1995.

[44] R. Grosso and T. Ertl. Biorthogonal wavelet filters for frequency domain volume rendering. In R. Scateni, J. van Wijk, and P. Zanarini, editors, *Proceedings of Visualization in Scientific Computing '95*, pages 81–95. Springer, Wien, New York, 1995.

[45] R. Grosso, T. Ertl, and J. Aschoff. Efficient data structures for volume rendering of wavelet-compressed data. In *Fourth Int. Conference in Central Europe on Computer Graphics and Visualization '96*, volume I, pages 103–112. WSCG'96, 1996.

[46] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *Proc. IEEE Visualization 2002*, pages 53–60, Boston, Massachusetts, USA, 2002. IEEE Computer Society.

[47] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[48] H. J. A. M. Heijmans. *Morphological Image Operators*, volume 25 of *Advances in Electronics and Electron Physics, Supplement*. Academic Press, New York, 1994.

[49] H. J. A. M. Heijmans and J. Goutsias. Multiresolution signal decomposition schemes. Part 2: morphological wavelets. *IEEE Trans. Image Processing*, 9(11):1897–1913, 2000.

[50] H. J. A. M. Heijmans and J. B. T. M. Roerdink, editors. *Mathematical Morphology and its Applications to Image and Signal Processing*, volume 12 of *Computational imaging and vision*, chapter Chapter 3: Shape Analysis and Partial Differential Equations. Kluwer Academic Publishers, 1998.

[51] M. Hopf and T. Ertl. Accelerating Morphological Analysis with Graphics Hardware. In *Workshop on Vision, Modelling, and Visualization VMV '00*, pages 337–345. infix, 2000.

[52] M. Hopf and T. Ertl. Hardware Accelerated Wavelet Transformations. In *Proc. of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 93–103, May 2000.

[53] Q. Hou, K. Zhou, and B. Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. *ACM Trans. Graph.*, 28(5), 2009.

[54] B. Houston, M. B. Nielsen, C. Batty, O. Nilsson, and K. Museth. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Trans. Graph.*, 25(1):151–175, 2006.

[55] I. Ihm and S. Park. Wavelet-based 3d compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, March 1999.

[56] A. C. Jalba and J. B. T. M. Roerdink. Efficient surface reconstruction using generalized Coulomb potentials. *IEEE Trans. Visualization and Computer Graphics*, 13(6):1512–1519, 2007.

[57] A. C. Jalba, M. H. Wilkinson, and J. B. T. M. Roerdink. Automatic segmentation of diatom images for classification. *Microscopy Research and Technique*, 65(1-2):72–85, Sept. 2004.

[58] A. C. Jalba, M. H. Wilkinson, and J. B. T. M. Roerdink. Morphological hat-transform scale spaces and their use in pattern classification. *Pattern Recognition*, 37(5):901–915, 2004.

[59] A. C. Jalba, M. H. Wilkinson, and J. B. T. M. Roerdink. Shape representation and recognition through morphological curvature scale spaces. *IEEE Trans. Image Processing*, 15(2):331–341, Feb. 2006.

[60] W. Jiang and A. Ortega. Parallel Architecture for the Discrete Wavelet Transform based on the Lifting Factorization. *Journal of Parallel and Distributed Computing*, 57(2):257–269, 1999.

[61] C. Johanson. Real-time water rendering - introducing the projected grid concept. Master's thesis, Department of Computer Science, Lund University, March 2004.

[62] M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson surface reconstruction. In *Eurographics Symposium on Geometry Processing*, pages 61–70, 2006.

[63] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[64] V. V. Kindratenko, J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. mei W. Hwu. GPU clusters for high-performance computing. In *CLUSTER*, pages 1–8. IEEE, 2009.

[65] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Trans. Visualization and Computer Graphics*, 8(3):270–285, 2002.

[66] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[67] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[68] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Trans. Vis. Comput. Graph.*, 10(4):422–433, 2004.

[69] D. LeGall and A. Tabatabai. Sub-band coding of digital images using symmetric short kernel filters and arithmetic coding techniques. In *IEEE Int. Conf. Acoustics, Speech and Signal Processing*, volume 2, pages 761–764, 1988.

[70] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008.

[71] L. Lippert and M. H. Gross. Fast wavelet based volume rendering by accumulation of transparent texture maps. *Computer Graphics Forum*, 14(3):431–443, 1995.

[72] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. Cuda renderer: a programmable graphics pipeline. In *SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 Sketches*, pages 1–1, New York, NY, USA, 2009. ACM.

[73] G. R. Liu and M. B. Liu. *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. World Scientific, 2003.

[74] X.-D. Liu, S. Osher, and T. Chan. Weighted essentially non-oscillatory schemes. *J. Comput. Phys.*, 115(1):200–212, 1994.

[75] N. Loménie, L. Gallo, N. Cambou, and G. Stamon. Morphological operations on delaunay triangulations. In *Int. Conf. Pattern Recognition (ICPR'00), Sept. 3-8, Barcelona, Spain*, pages 3556–9, 2000.

[76] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[77] 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.

[78] L. Luo, J. Li, S. Li, Z. Zhuang, and Y.-Q. Zhang. Motion compensated lifting wavelet and its application in video coding. In *Proc. IEEE International Conference on Multimedia and Expo (ICME)*, pages 365–368, 2001.

[79] L. Luo, L. Luo, J. Li, S. Li, and Z. Zhuang. A motion compensated lifting wavelet codec for 3D video coding. *J. Comput. Sci. Technol.*, 18(2):214–222, 2003.

[80] C. Lürig and T. Ertl. Hierarchical volume analysis and visualization based on morphological operators. In *Proc. IEEE Visualization '98*, pages 335–341. IEEE Computer Society Press, 1998.

[81] R. Malladi and J. A. Sethian. Level set methods for curvature flow, image enhancement, and shape recovery in medical images. In *In Proc. of Conf. on Visualization and Mathematics*, pages 329–345. Springer-Verlag, 1995.

[82] S. Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, New York, 1998.

[83] X. Mei, P. Decaudin, B.-G. Hu, and X. Zhang. Real-time marker level set on GPU. In *International Conference on Cyberworlds, CW '08, September, 2008*, Hangzhou, Chine, 2008. IEEE.

[84] C. Min. Local level set method in high dimension and codimension. *J. Comput. Phys.*, 200(1):368–382, 2004.

[85] P. J. Moran. An interpreted language and system for the visualization of unstructured meshes. In *Proceedings, 6th International Meshing Roundtable, Sandia National Laboratories, October*, pages 233–248, 1997.

[86] L. Mroz, A. König, and E. Gröller. Maximum intensity projection at warp speed. *Computers & Graphics*, 24:343–352, 2000.

[87] M. Müller, S. Schirm, and S. Duthaler. Screen space meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 9–15, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[88] K. Museth, D. E. Breen, R. T. Whitaker, and A. H. Barr. Level set surface editing operators. *ACM Trans. Graph.*, 21(3):330–338, 2002.

[89] M. B. Nielsen and K. Museth. Dynamic Tubular Grid: An efficient data structure and algorithms for high resolution level sets. *J. Sci. Comput.*, 26(3):261–299, 2006.

[90] NVidia. *CUDA Occupancy Calculator*. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[91] NVidia. *NVIDIA CUDA C Programming Best Practices Guide*.

[92] NVIDIA Corporation. *Compute Unified Device Architecture programming guide*. Available at http://developer.nvidia.com/cuda.

[93] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Available at http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[94] S. Osher, T. Chan, X. dong Liu, and X. dong Liu. Weighted essentially non-oscillatory schemes. *J. Comput. Phys*, 115:200–212, 1994.

[95] S. Osher and R. Fedkiw. Level set methods: an overview and some recent results. *J. Comput. Phys.*, 169(2):463–502, 2001.

[96] S. Osher and R. Fedkiw. *Level-Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, New York, 2002.

[97] 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.

[98] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[99] 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.

[100] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[101] B. Pesquet-Popescu and V. Bottreau. Three-dimensional lifting schemes for motion compensated video compression. In *ICASSP '01: Proc. of the Acoustics, Speech, and Signal Processing Conference*, pages 1793–1796, 2001.

[102] G. Piella and H. Heijmans. Adaptive lifting schemes with perfect reconstruction. *IEEE Trans. Image Processing*, 50(7):1620–1630, 2002.

[103] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):773–790, 2003.

[104] J. B. T. M. Roerdink. Multiresolution maximum intensity volume rendering by morphological pyramids. In D. Ebert, J. M. Favre, and R. Peikert, editors, *Data Visualization 2001. Proc. Joint Eurographics – IEEE TCVG Symposium on Visualization, May 28-30, 2001, Ascona, Switzerland*, pages 45–54. Springer, Wien, New York, 2001.

[105] J. B. T. M. Roerdink. Comparison of morphological pyramids for multiresolution MIP volume rendering. In D. Ebert, P. Brunet, and I. Navazo, editors, *Data Visualization 2002. Proc. Eurographics – IEEE TCVG Symposium, May 27-29, 2002, Barcelona, Spain*, pages 61–70. ACM, New York, 2002.

[106] J. B. T. M. Roerdink. Multiresolution maximum intensity volume rendering by morphological adjunction pyramids. *IEEE Trans. Image Processing*, 12(6):653–660, June 2003.

[107] J. B. T. M. Roerdink. Morphological pyramids in multiresolution MIP rendering of large volume data: Survey and new results. *J. Math. Imag. Vision*, 22(2/3):143–157, 2005.

[108] J. B. T. M. Roerdink and G. S. M. Blaauwgeers. Visualization of Minkowski operations by computer graphics techniques. In J. Serra and P. Soille, editors, *Mathematical Morphology and its Applications to Image Processing*, pages 289–296. Kluwer Acad. Publ., Dordrecht, 1994.

[109] I. D. Rosenberg and K. Birdwell. Real-time particle isosurface extraction. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 35–43, New York, NY, USA, 2008. ACM.

[110] C. Rössl, L. Kobbelt, and H.-P. Seidel. Extraction of feature lines on triangulated surfaces using morphological operators. In *AAAI 2000 Spring Symposium Series "Smart Graphics", March 20-22, Stanford, USA*, pages 71–75, 2000.

[111] M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, volume 3, pages 1103–1106, 2001.

[112] M. Sainz and R. Pajarola. Point-based rendering techniques. *Computers & Graphics*, 28(6):869–879, 2004.

[113] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proc. of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[114] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, New York, 1982.

[115] J. Serra, editor. *Image Analysis and Mathematical Morphology. II: Theoretical Advances*. Academic Press, New York, 1988.

[116] J. Sethian. *Level Set Methods: Evolving Interfaces in Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, June 1996.

[117] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999.

[118] G. Shen, G. P. Gao, S. Li, H. Y. Shum, and Y. Q. Zhang. Accelerate video decoding with generic GPU. *IEEE Trans. Circuits and Systems for Video Technology*, 15(5):685–693, May 2005.

[119] D. Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proc. of the 1968 23rd ACM national conference*, pages 517–524, New York, NY, USA, 1968. ACM.

[120] M. Shih, Y.-F. Chiu, Y.-C. Chen, and C.-F. Chang. Real-time ray tracing with cuda. In *ICA3PP '09: Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, pages 327–337, Berlin, Heidelberg, 2009. Springer-Verlag.

[121] C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *J. Comput. Phys.*, 77(2):439–471, 1988.

[122] C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *J. Comput. Phys.*, 77(2):439–471, 1988.

[123] A. N. Skodras, C. A. Christopoulos, and T. Ebrahimi. JPEG2000: The upcoming still image compression standard. *Pattern Recognition Letters*, 22(12):1337–1345, 2001.

[124] P. Smereka. Semi-implicit level set methods for curvature and surface diffusion motion. *J. Sci. Comput.*, 19(1-3):439–456, 2003.

[125] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[126] D. Stora, P.-O. Agliati, M.-P. Cani, F. Neyret, and J.-D. Gascuel. Animating lava flows. In *Graphics Interface*, pages 203–210, Jun 1999.

[127] J. Strain. Tree methods for moving interfaces. *J. Comput. Phys.*, 151(2):616–648, 1999.

[128] M. Strengert, M. Magallon, D. Weiskopf, S. Guthe, and T. Ertl. Hierarchical visualization and compression of large volume datasets using GPU clusters. *Parallel Graphics and Visualization*, 2004.

[129] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics*, 28(1):4:1–4:11, Jan. 2009.

[130] H. Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), March 2005.

[131] W. Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. Math. Anal.*, 29(2):511–546, 1997.

[132] W. Sweldens. The Lifting Scheme: A Construction of Second Generation Wavelets. *SIAM Journal on Mathematical Analysis*, 29(2):511–546, 1998.

[133] W. Sweldens and P. Schröder. Building your own wavelets at home. In *Wavelets in Computer Graphics*, pages 15–87. ACM SIGGRAPH Course notes, 1996.

[134] C. Tenllado, R. Lario, M. Prieto, and F. Tirado. The 2D discrete wavelet transform on programmable graphics hardware. In *Proc. of the 4th IASTED International Conference on Visualization, Imaging, and Image Processing*, 2004.

[135] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado. Parallel implementation of the 2D discrete wavelet transform on graphics processing units: Filter bank versus lifting. *IEEE Transactions on Parallel and Distributed Systems*, 19(3):299–310, 2008.

[136] J. Y. Tham, S. Ranganath, and A. A. Kassim. Highly scalable wavelet-based video codec for very low bit-rate environment. *IEEE Journal on selected areas in communications*, 16:12–27, jan 1998.

[137] The Khronos Group. *The OpenCL specification*. http://www.khronos.org/registry/cl/.

[138] R. Tsai and S. Osher. Level set methods and their applications in image science. *Comm. Math. Sci.*, 1(4):623–656, 2003.

[139] M. Tun, K. K. Loo, and J. Cosmas. Error-Resilient Performance of Dirac Video Codec Over Packet-Erasure Channel. *IEEE Transactions on Broadcasting*, 53(3):649–659, sep 2007.

[140] J. Z. Turlington and W. E. Higgins. New techniques for efficient sliding thin-slab volume visualization. *IEEE Trans. Med. Imaging*, 20(8):823–835, Aug. 2001.

[141] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. Cuda-lite: Reducing gpu programming complexity. In J. N. Amaral, editor, *LCPC*, volume 5335 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[142] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink. Accelerating wavelet-based video coding on graphics hardware using CUDA. In *Proc. 6th Intern. Symp. on Image and Signal Processing and Analysis (ISPA 2009), September 16-18, Salzburg, Austria*, pages 614–619, 2009.

[143] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink. Accelerating wavelet lifting on graphics hardware using CUDA. Technical report, Institute for Mathematics and Computing Science, University of Groningen, 2009. Submitted for publication.

[144] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink. A memory and computationally-efficient sparse level-set algorithm. Technical report, Institute for Mathematics and Computing Science, University of Groningen, 2009. Submitted for publication.

[145] V. Volkov and J. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical report, University of California at Berkeley, 2008.

[146] M. A. Westenberg and J. B. Roerdink. Mixed-method identifications. In J. M. H. Du Buf and M. M. Bayer, editors, *Automatic Diatom Identification*, volume 51 of *Series in Machine Perception and Artificial Intelligence*, chapter 12, pages 245–257. World Scientific Publishing Co., Singapore, 2002.

[147] M. A. Westenberg and J. B. T. M. Roerdink. Frequency domain volume rendering by the wavelet X-ray transform. *IEEE Trans. Image Processing*, 9(7):1249–1261, 2000.

[148] M. A. Westenberg and J. B. T. M. Roerdink. X-ray volume rendering by hierarchical wavelet splatting. In *Proc. 15th Intern. Conf. on Pattern Recognition (ICPR'2000), Barcelona, Sep. 3-7*, pages 163–166, 2000.

[149] M. A. Westenberg and J. B. T. M. Roerdink. X-ray volume rendering through two-stage splatting. *Machine Graphics & Vision*, 9(1/2):307–314, 2000.

[150] M. A. Westenberg and J. B. T. M. Roerdink. An extension of Fourier-wavelet volume rendering by view interpolation. *J. Math. Imag. Vision*, 14(2):103–115, 2001.

[151] M. A. Westenberg, M. H. F. Wilkinson, and J. B. T. M. Roerdink. Nonlinear volumetric filtering and interactive visualization using the max-tree representation. Technical report, Institute for Mathematics and Computing Science, University of Groningen, 2004.

[152] R. Westermann. A multiresolution framework for volume rendering. In *ACM workshop on Volume Visualization*, pages 51–58, 1994.

[153] R. T. Whitaker. A level-set approach to 3D reconstruction from range data. *Int. J. Comput. Vision*, 29(3):203–231, 1998.

[154] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.

[155] M. H. F. Wilkinson, A. C. Jalba, E. R. Urbach, and J. B. T. M. Roerdink. Identification by mathematical morphology. In J. M. H. Du Buf and M. M. Bayer, editors, *Automatic Diatom Identification*, volume 51 of *Series in Machine Perception and Artificial Intelligence*, chapter 11, pages 221–244. World Scientific Publishing Co., Singapore, 2002.

[156] B. W. Williams. Fluid surface reconstruction from particles. Master's thesis, The University Of British Columbia, February 2008.

[157] T. T. Wong, C. S. Leung, P. A. Heng, and J. Wang. Discrete Wavelet Transform on Consumer-Level Graphics Hardware. *IEEE Transactions on Multimedia*, 9(3):668–673, apr 2007.

[158] N. D. Zervas, G. P. Anagnostopoulos, V. Spiliotopoulos, and Y. Andreopoulos. Evaluation of design alternatives for the 2D discrete wavelet transform. *IEEE Trans. Circ. and Syst. for Video Tech.*, 11:1246–1262, 2001.

[159] Y. Zhang and R. Pajarola. Deferred blending: Image composition for single-pass point rendering. *Computers & Graphics*, 31(2):175–189, 2007.

[160] Y. Zhang, B. Solenthaler, and R. Pajarola. Adaptive sampling and rendering of fluids on the gpu. In *In Proc. of Symposium on Point-Based Graphics*, pages 137–146, 2008.

[161] H. Zhao, S. Osher, and R. Fedkiw. Fast surface reconstruction using the level set method. In *Proceedings of the IEEE Workshop on Variational and Level Set Methods in Computer Vision*, pages 194–202, 2001.

[162] D. Zhongminga, T. Kawamuraa, N. Sakamotob, and K. Koyamadab. Particle-based multiple irregular volume rendering on cuda. *Simulation Modelling Practice and Theory*, 18(8):1172–1183, sep 2010.

[163] K. Zhou, M. Gong, X. Huang, and B. Guo. Highly parallel surface reconstruction. Technical report, Microsoft Research, 2008.

[164] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.

Publications

Papers in scientific journals

Wladimir J. van der Laan, Andrei C. Jalba, and Jos B. T. M. Roerdink. Accelerating Wavelet Lifting on Graphics Hardware using CUDA. *IEEE Transactions on Parallel and Distributed Systems*, **22**, 2011, pp. 132-146, http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.143.

Wladimir J. van der Laan, Andrei C. Jalba, and Jos B. T. M. Roerdink. A Memory and Computation Efficient Sparse Level-Set Method. *Journal of Scientific Computing*, 2010, http://dx.doi.org/10.1007/s10915-010-9399-5, p. 1-22.

Andrei C. Jalba, Wladimir J. van der Laan, and Jos B. T. M. Roerdink. Real-Time Sparse Level-Sets on Graphics Hardware. Submitted.

Full papers in conference proceedings

Wladimir J. van der Laan, Andrei C. Jalba, and Jos B. T. M. Roerdink. Multiresolution MIP Rendering of Large Volumetric Data Accelerated on Graphics Hardware. In *Proc. Eurographics/IEEE VGTC Symposium on Visualization (EuroVis)*, pages 243-250, 2007.

Wladimir J. van der Laan, Simon Green, and Miguel Sainz. Screen Space Fluid Rendering with Curvature Flow. In *Proc. I3D 2009: The 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 91-98, 2009.

Wladimir J. van der Laan, Andrei C. Jalba, and Jos B.T.M. Roerdink (2009) Accelerating Wavelet-Based Video Coding on Graphics Hardware using CUDA. *In Proc. 6th International Symposium on Image and Signal Processing and Analysis (ISPA 2009, September 16–18, Salzburg, Austria)*. Pages 614–619, 2009.

Other publications

W.J. van der Laan and J.B.T.M. Roerdink, MIP rendering using adjunction pyramids accelerated on graphics hardware, *SIREN: Scientific ICT Research Event Netherlands*, 12 October 2006, Utrecht (poster).

W.J. van der Laan, A.C. Jalba and J.B.T.M. Roerdink, Wavelet Lifting on Graphics Hardware for Faster Video Decoding, *SIREN: Scientific ICT Research Event Netherlands*, 30 October 2007, TU Delft (poster).

W. J. van der Laan, A. C. Jalba and J. B. T. M. Roerdink. Multiresolution MIP rendering of large volumetric data accelerated on graphics hardware. In *Proc. 13th Ann. Conf. Advanced School for Computing and Imaging (ASCI'07)*, pp. 59-66.

Samenvatting

Visualisatie speelt een steeds grotere rol bij de interpretatie van gegevens uit medische scans, simulaties van natuurkundige processen, driedimensionale microscopie, astronomische waarnemingssystemen, enz. Doordat niet van tevoren duidelijk is welke elementen in de gegevens van belang zijn, dient de gebruiker invloed te kunnen uitoefenen op het verloop van het visualisatieproces, via het veranderen van aanzichten of parameters; we spreken dan van interactieve visualisatie. Hierdoor wordt de tijd die nodig is om gegevens te interpreteren of interessante verbanden op het spoor te komen sterk verkleind. De wens tot interactieve visualisatie stelt echter hoge eisen aan de visualisatiesystemen in termen van snelheid, economisch geheugengebruik, en datamanagement. In dit proefschrift hebben we dit probleem op verschillende manieren aangepakt.

In hoofdstuk 2 hebben we een aantal algoritmen onderzocht die gebaseerd zijn op morfologische piramiden voor zgn. multiresolutie MIP ("MaximumIntensiteitsProjectie"). MIP is een methode voor volume-visualisatie waarbij de hoogste intensiteit langs een willekeurige lijn door het volume in het beeldvlak wordt weergegeven. Deze algoritmen zijn zo geïmplementeerd dat ze rechtstreeks op een grafische kaart werken. We hebben geconstateerd dat onze geoptimaliseerde "streaming MIP" methode zowel de software-implementatie als bestaande grafische hardware-gebaseerde methoden voorbij streeft.

In hoofdstuk 3 hebben we een nieuwe, snelle implementatie op grafische kaarten gepresenteerd van de "wavelet lifting" transformatie. De "wavelet lifting" methode construeert ingewikkelde wavelet-transformaties vanuit eenvoudige basistransformaties: dit proces noemen we "lifting". Hierbij hebben we gebruik gemaakt van CUDA, een recente programmeer-architectuur voor grafische hardware. Deze methode is te gebruiken voor ruimten van willekeurige dimensie. We hebben onze methode vergeleken met een geoptimaliseerde software-implementatie van "wavelet lifting" en met een CUDA-implementatie die gebaseerd is op convolutie. De methode is schaalbaar en we hebben laten zien dat onze methode de snelste is van alle methoden die we hebben vergeleken. Ons algoritme kan worden toegepast op bijvoorbeeld beeld- of videocompressie.

In hoofdstuk 4 hebben we laten zien hoe de "Dirac Video Codec", een compacte manier om videomateriaal op te slaan, kan worden versneld met behulp van de "wavelet lifting" implementatie op grafische hardware die we in het vorige hoofdstuk hebben behandeld. Ook hebben we twee andere fasen van het complete videocoderingsproces op grafische hardware geïmplementeerd, namelijk de bewegingscompensatie en de compositie van individuele beelden. Bij onze experimenten konden we een videosignaal, bestaande uit beelden met 1920×1080 beeldpunten, afspelen met zo'n 50 beelden per seconde op een gewone videokaart.

In hoofdstuk 5 hebben we een nieuwe methode gepresenteerd om het oppervlak van vloeistof-

fen zoals water op een realistische manier af te beelden. Met deze methode kan het resultaat van een vloeistofsimulatie rechtstreeks en snel worden afgebeeld door de vloeistof als deeltjessysteem voor te stellen. Op deze manier kunnen we een vloeiende en interactieve animatie bewerkstelligen. Ook hebben we nieuwe ideeën geïntroduceerd voor schaduweffecten die gebaseerd zijn op de dikte van de vloeistoflaag. Tevens hebben we kleine oppervlakte-details aangebracht op de afgebeelde vloeistoffen.

In hoofdstuk 6 hebben we een efficiënte datastructuur voorgesteld voor zgn. niveauverzamelingen ("level-sets"). Niveauverzamelingen worden gebruikt om vervormbare oppervlakken te beschrijven. De vervorming van een oppervlak in de tijd kan beschreven worden met behulp van een partiële differentiaalvergelijking. We hebben onze methode "Sorted Tile Grid" genoemd. Het volume wordt namelijk opgedeeld in blokken ("tiles") die in gesorteerde volgorde worden gehouden om veranderingen in het volume zo efficiënt mogelijk te kunnen doorrekenen. We hebben onze methode vergeleken met de thans meest gebruikte methode, het "DT-Grid". Hierbij hebben we geconstateerd dat onze methode, met gebruik van dezelfde numerieke simulatie, ruim 8 keer sneller is.

In hoofdstuk 7 hebben we de "Sorted Tile Grid" methode uit het vorige hoofdstuk geschikt gemaakt voor implementatie op grafische hardware en andere parallelle architecturen. We hebben laten zien dat onze implementatie op grafische hardware ongeveer 20 keer sneller is dan de geoptimaliseerde software-versie van het vorige hoofdstuk. Veel toepassingen van de "level-set" methode kunnen profiteren van onze vinding. Om dit te demonstreren hebben we twee toepassingen laten zien: reconstructie van oppervlakken vanuit een wolk van punten, en het interactief bewerken van oppervlakken.

Dankwoord

Bijzondere dank gaat uit naar mijn promotor, Prof. Dr. Jos Roerdink, voor het aanreiken van het onderwerp, het verstrekken van onderliggende theoretische informatie en het kritisch evalueren van de tekst. En ook voor zijn steun tijdens het tot stand komen van dit werk. Op momenten dat het wat minder hard ging wist hij veel vertrouwen erin te houden, dit heeft mij geholpen dit proefschrift te maken tot wat het is.

Daarnaast ook een gemeend woord van dank aan mijn copromotor, Dr. Andrei Jalba, voor het geven van richting aan het onderzoek, zijn hulp met de tekst, software-implementaties, en zijn gedetailleerde technische commentaar. Ook is hij degene die oorspronkelijk mijn belangstelling gewekt heeft om een promotie-onderzoek te gaan doen.

Speciale dank gaat uit naar Miguel Sainz, Simon Green en Nikki Gravestock van NVIDIA voor het aanbieden van een stage in Londen, waar ik veel over GPU hardware heb geleerd. Ook was het een geweldige ervaring om met de groep van NVIDIA naar SIGGRAPH 2008 te gaan. Verder wil ik NVIDIA bedanken voor het ter beschikking stellen van een grafische kaart voor berekeningen en experimenten.

Mijn dank gaat verder uit naar Thomas Ertl en de medewerkers van het Visualisatie Instituut van de Universiteit van Stuttgart, waar ik met plezier een maand doorgebracht heb en waar ik mijn eerste stappen richting CUDA heb gemaakt.

Ook gaat mijn dank uit naar de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO), specifiek het “VIEW: Visual Interactive Effective Worlds” programma, voor het sponsoren van mijn onderzoek.

Tenslotte wil ik mijn vrienden, mijn vriendin, mijn zus en mijn ouders bedanken voor hun niet aflatende steun gedurende de afgelopen jaren.