Institut für Informatik
Technische Universität München

Efficient Methods for the Display of highly detailed Models in Computer Graphics

Jens Schneider

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. U. Baumgarten
Prüfer der Dissertation:	Univ.-Prof. Dr. R. Westermann
	apl. Prof. Dr. T. Kuhlen
	(Rheinisch-Westfälische Technische Hochschule Aachen)

Die Dissertation wurde am 25.11.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.5.2009 angenommen.

To my family and friends.

Abstract

In 1965, Intel co-founder Gordon Moore observed that the number of devices (transistors, resistors, etc.) doubled every twelve months. Ten years later he predicted that the number of transistors of CPUs doubled every 24 months. A consequence of what is now known as “Moore’s Law” is that processing power also increases exponentially, albeit not at a factor of two per two years. The consequences of this still precise prognosis for today’s society are amazing. It is by now possible to rapidly generate or acquire data sets that are so large in size that processing or displaying these data sets has become a severe issue and typically requires both state-of-the-art hardware and sophisticated algorithms.

The recent introduction of graphics accelerators for mainstream PCs, collectively known as graphics processing units (GPUs), has offered the potential to explore these data sets at interactive rates. However, due to the still limited video memory of today’s GPUs and the von Neumann architecture of modern PCs, the storage and bandwidth requirements arising during the visualization of large data sets have to be carefully analyzed.

In this thesis, we explore a class of visualization algorithms commonly referred to as level-of-detail (LOD) algorithms. These algorithms typically perform a hierarchical analysis of large, highly detailed data sets during a preprocessing step. During runtime, the amount and detail of the data necessary to form an image for a given set of camera parameters is determined, and the respective data is sent to the GPU to be displayed. Since the data is usually too large to reside in host memory, paging strategies that hide latencies of external storage solutions are employed. We demonstrate that in this way highly interactive frame rates can be achieved for the visualization of massive point clouds, high-resolution terrain, large, triangulated models, and gigapixel-sized images. Furthermore, we demonstrate that interactivity leads to immediate visual feedback loops for user-made changes of the data set. This feedback offers the possibility

to design highly intuitive and powerful editing environments—e.g., for fractal landscapes and the design of digital filters that operate on gigapixel images—that follow the well-established WYSIWIG concept.

Zusammenfassung

Im Jahr 1965 beobachtete der Intel Mitbegründer Gordon Moore dass sich die Anzahl der Bauteile (Transistoren, Widerstände usw.) in etwa alle zwölf Monate verdoppelt. Zehn Jahre später formulierte er die nun als "Moores Gesetz" bekannte Prognose, dass sich die Anzahl der Transistoren von CPUs alle 24 Monate verdoppelt. Obwohl sich die Leistungsfähigkeit von Prozessoren langsamer entwickelt, ist doch aufgrund der auch heute noch präzisen Prognose eine exponentielle Beschleunigung von generellen CPUs zu beobachten. Die Konsequenzen für die heutige Gesellschaft sind erstaunlich. Es ist heutzutage möglich, in kurzer Zeit Datensätze zu generieren oder zu messen, deren schiere Größe bei der Darstellung echte Probleme aufwirft. Als Konsequenz sind üblicherweise aktuellste Rechner und ausgefeilte Algorithmen erforderlich.

Die erst seit recht kurzer Zeit für Standard-PCs verfügbaren Grafikbeschleuniger, die kollektiv auch GPUs (*graphics processing units*) genannt werden, haben das Potential eine Echtzeitexploration solcher Datensätze zu ermöglichen. Jedoch müssen die Speicher- und Bandbreitenanforderungen, die während der Visualisierung dieser Daten anfallen, genau untersucht werden, da sich sowohl der verfügbare Videospeicher als auch die verfügbaren Bandbreiten heutiger von Neumann Architekturen schnell als limitierend erweisen.

Diese Dissertation untersucht eine Klasse von Visualisierungsalgorithmen die kollektiv als *level-of-detail* (LOD) Algorithmen bekannt sind. Diese Algorithmen führen typischerweise eine hierarchische Analyse von großen, hochdetailierten Datensätzen in einem Vorverarbeitungsschritt durch. Zur Laufzeit werden dann die Teile und der Detailgrad bestimmt, die für die Berechnung des finalen Bildes unter den aktuellen Kameraparametern nötig sind. Diese Daten werden dann an die GPU gesendet und dargestellt. Weil die Datensätze üblicherweise zu groß sind, um im Hauptspeicher gehalten werden zu können, werden geeignete Auslagerungsstrategien beschrieben um die Latenzen externer Speichermedien zu verstecken. Wir zeigen am Beispiel einer

Reihe von unterschiedlichen Daten, etwa gigantischen Punktwolken, hochaufgelösten Landschaftsdaten sowie Gigapixel-Bildern, dass auf diese Weise interaktive Darstellungsraten erreicht werden. Diese Interaktivität kann dann für Editoren mit sofortigen visuellen Rückmeldungen im Stile des WYSIWIG-Konzeptes (*What-You-See-Is-What-You-Get*) genutzt werden. Solche Rückmeldungen erlauben das Design hochintuitiver und mächtiger Editierumgebungen, wie sie am Beispiel eines fraktalen Landschaftseditors und einer *Rapid Prototyping* Umgebung für digitale Bildfilter auf Gigapixel-Bildern demonstriert werden.

Acknowledgements

I would like to thank all persons who supported me and helped me to make this work possible. First of all, I would like to thank my collegues, current and former ones, namely (and in alphabetical order) Kai Bürger, Christian Dick, Roland Fraedrich, Raymund Fülöp, Dr. Joachim Georgii, Stefan Hertel, Dr. Peter Kipfer, Dr. Polina Kondratieva, Dr. Martin Kraus, Dr. Jens Krüger, Jörg Liebelt, Hans-Georg Menz, and Dr. Thomas Schiwietz. They have always been available for discussing ideas and many have helped proof-reading this thesis. I would also like to thank Sebastian Wohner for his relentless effort to keep my desktop system running despite my natural gift to wreck it in a wide variety of ways.

Also, I would like to thank some of my students, who helped me in implementing and validating several of the methods presented here, namely Moritz Bartl, Tobias Boldte, Dominik Meyer, Matthias Wagner, and Florian Wendel.

I would also like to thank my friends, who supported me with high morale and well-received distractions of different kinds.

Last but not least I would like to thank my advisor, Prof. Dr. Rüdiger Westermann, who provided me with the opportunity to conduct the studies presented in this thesis, who always was open for discussion, and who inspired many of the methods presented here. Without him, this thesis would not have been possible.

I would like to thank the many people and institutions who provided me with data sets used in our publications and in this thesis, namely the people from the Digital Michelangelo Project for scanning the statues used in Chapter 3 and for making the data publicly available. Also, I would like the people from the Lawrence Livermore National Laboratory in general and Peter Lindstrom in particular for providing the interface instability data set. I would like to thank Siemens Corporate Research for kindly providing the Wholebody CT scan.

Also, I would like to thank the DLR (Deutsches Zentrum für Luft- und Raumfahrt) Oberpfaffenhofen for the Alps data set and the DLR and ISTAR for the Paris data set used in Chapter 4. Furthermore, I am thankful to the people of GA Tech's (Georgia Institute of Technology) large models project for making the Puget Sound and Grand Canyon data sets publicitly available.

I would like to thank Thomas Heinrich for modelling and kindly providing the *Devil Head* data set used in Chapter 6. For all other meshes in that chapter that I did not generate myself, I have to thank the people of the Aim @ Shape project for making such a plethora of meshes publicly available.

Finally, I would like to thank Johannes Kopf and Prof. Dr. Oliver Deussen for providing us with one of their gigapixel images.

Contents

Abstract i

Zusammenfassung iii

Acknowledgements v

List of Figures xiii

List of Tables xv

List of Algorithms xvii

1 Introduction 1

1.1 Research Publications and Contributions 4

2 Basic Techniques 7

2.1 GPUs and Graphics APIs 7
2.2 Digital Photography and CCDs 10
2.3 Laser Range Scanning of 3D Objects 12
2.4 Acquisition of Digital Elevation Maps 16
2.5 CT Scanning 17
2.6 Principal Component Analysis 20
2.7 Vector Quantization 23
2.8 Gauss- and Laplace-Pyramids 25

3 Rendering Of Massive Point Scans 29

3.1 Related Work 30
3.2 Contribution 36
3.3 Algorithmic Overview 36

3.3.1 Uniform Point Clustering and Resampling 39
3.3.2 Path Generation 41
3.4 Grid Optimization 44
3.5 Encoding 46
3.5.1 Point Encoding 46
3.5.2 Normal Estimation and Encoding 47
3.6 Extension to Isosurface Compression 50
3.6.1 Isosurface Preprocessing 52
3.7 Rendering 55
3.7.1 Culling and LOD 56
3.7.2 High-Quality Rendering 57
3.8 Runtime And Memory Requirements 61
3.9 Results 63
3.10 Summary 66

4 Rendering Of Large-Scale Terrain 71
4.1 Contribution 71
4.2 Algorithmic Overview 72
4.3 Related Work 73
4.4 Nested Mesh Hierarchy 76
4.5 Rendering Framework 79
4.5.1 GPU Data Structures 79
4.5.2 Run-Time Processing 81
4.5.3 Level of detail 81
4.5.4 Geomorphing 83
4.5.5 Texturing 84
4.6 Memory Management 86
4.7 Results 87
4.8 Summary 88

5 Fractal Terrain 91
5.1 Related Work 92
5.2 Contribution 94
5.3 Multifractal Terrain Synthesis 96
5.3.1 The Rescale-and-Add Method 96
5.3.2 Domain Warping 97
5.4 Interactive Fractal Editing 100

5.5 Rendering . . . 102
5.5.1 Projected Grid and LOD . . . 102
5.5.2 Proto-Texturing . . . 106
5.5.3 Water . . . 107
5.5.4 Results . . . 109
5.6 Summary . . . 110

6 Interactive Displacement Editing 113
6.1 Contribution . . . 114
6.2 Related Work . . . 116
6.3 Displacement Mapping . . . 117
6.3.1 Initial Grid Layout . . . 118
6.3.2 Vertex Tracing on Triangle Meshes . . . 120
6.3.3 Grid Relaxation . . . 123
6.3.4 Mass-Spring Topology . . . 123
6.3.5 Mass-Spring Solver . . . 125
6.3.6 Euler Backward Formulation . . . 126
6.3.7 Gauss-Seidel Formulation . . . 127
6.3.8 Constrained Updates, Quality Metric, and Stepsize Control . . . 133
6.3.9 GPU Implementation . . . 135
6.4 Rendering . . . 137
6.5 Results . . . 139
6.6 Summary . . . 141

7 Gigapixel Images 145
7.1 Contribution . . . 145
7.2 Related Work . . . 147
7.3 Algorithmic Overview . . . 148
7.4 Preprocessing . . . 149
7.5 Rendering . . . 150
7.6 User Interaction . . . 154
7.7 Filtering Gigapixel Images . . . 155
7.7.1 Filter Graph . . . 159
7.7.2 Filter Templates . . . 162
7.7.3 Filter Graph Ordering and Resource Management . . . 163
7.8 Euclidean Distance Transforms Using Propagation Filters . . . 164
7.8.1 Related Work . . . 165

7.8.2 Problem Description 166
7.8.3 Basic Algorithm 167
7.8.4 GPU-based Implementation 169
7.8.5 Integration into the Gigapixel Filter Pipeline 172
7.8.6 Errors in 2D Vector Propagation 172
7.8.7 Performance of the Euclidean Distance Transform 173
7.9 Summary 176

8 Conclusions **179**

Bibliography **180**

List of Figures

2.1 The Direct3D 10 rendering pipeline. 8
2.2 Schematic view of a CCD. 10
2.3 Photograph of a CCD sensor. 11
2.4 Schematic view of a Full-Frame CCD. 12
2.5 A taxonomy for shape aqcuisition methods. 13
2.6 Schematic view of a LIDAR system. 14
2.7 Schematic view of a triangulation-based scanner. 15
2.8 Schematic view of the HRSC-AX. 16
2.9 Relation between Radon transformation and CT scanning. 18
2.10 Siemens Somatom 16 CT Scanner. 19
2.11 Gauss- and Laplace-pyramids in 2D. 27

3.1 Comparison between uncompressed and compressed Atlas point scan. . 30
3.2 Hexagonal closest sphere packing (HCP) grid. 32
3.3 The three largest scans provided by the Michelangelo project. 33
3.4 Two layers of an HCP grid. 39
3.5 Sampling of points into hexagonal grids. 41
3.6 Generation of a path visiting each vertex at least once. 43
3.7 Construction of short paths. 44
3.8 A cell with four neighbors. 45
3.9 Assignment of differential codes to paths. 47
3.10 A time step of an interface mixing instability simulation. 51
3.11 The Duodecim Engine. 53
3.12 Two large isosurfaces. 54
3.13 Encoding of runs into texture maps on the GPU. 57
3.14 David rendered at four different detail levels 58
3.15 Comparison of per splat and per pixel illumination. 59

3.16 The three passes needed for our deferred shading approach. 60
3.17 The David and Atlas statues of the Digital Michelangelo Project. 64
3.18 An isosurface of the Visible Human male data set. 68
3.19 Another time step of the interface mixing instability. 69
3.20 Zoom onto Michelangelo's St. Mathew Statue. 69

4.1 A $360°$ panorama of the Alps. 71
4.2 Levels of the nested mesh hierarchy. 76
4.3 Quadtree mesh and Π-order traversal. 78
4.4 Best and worst cases for vertex cache re-usal of fans. 79
4.5 The GPU data structures used in our terrain renderer. 80
4.6 Closing gaps at tile boundaries. 82
4.7 Basis-functions $\eta_i'(\lambda)$ for geomorphs. 84
4.8 Several data sets to test our terrain rendering algorithm. 85

5.1 A fractal landscape generated by our method. 95
5.2 Noise synthesis using varying parameters. 96
5.3 The effect of domain warping. 98
5.4 Real world granite structures. 98
5.5 The WYSIWYG interface of the fractal landscape editor. 101
5.6 Projected Grid concept. 103
5.7 Caveats of the projected grid. 104
5.8 Extending the projected grid to spherical domains. 105
5.9 Reducing the variance of the texture through adapting mipmap levels. . 106
5.10 Proto-textured Terrain. 107
5.11 Water surfaces in fractal landscapes. 108
5.12 Lake in a fractal landscape. 109
5.13 Terrain types made possible by domain warping. 110

6.1 Artifacts resulting from non-aligned base and displacement meshes. . . 114
6.2 Examples for our displacement method. 115
6.3 Displacement mapping overview. 118
6.4 Tracing of a quadrangular grid. 119
6.5 Avoiding folds in patches. 119
6.6 Demonstrating Fold-Avoidance. 120
6.7 Tracing a particle from $\boldsymbol{p_0}$ to $\boldsymbol{p_2}$. 122
6.8 A 2D grid placed on the nose of the Mannequin mesh. 123

6.9 The entangled topology 124
6.10 The three initial grids used for the mass-spring tests. 133
6.11 Benefits of immediate position constraints. 134
6.12 Vertex colorings of various mass-spring networks. 135
6.13 Demonstrating the cutout mask. 138
6.14 The textured models used to measure performance and quality. 139
6.15 Quality evaluation of grid relaxation using three different grids. 141
6.16 Results of the proposed displacement method (I). 143
6.17 Results of the proposed displacement method (II). 143
6.18 Results of the proposed displacement method (III). 144
6.19 Results of the proposed displacement method (IV). 144

7.1 An aerial photograph of the State of Utah, USA. 147
7.2 Layout of GPU-buffers containing encoded tiles. 151
7.3 Decoding of image tiles 152
7.4 Gigapixel landscape rendered using our system. 153
7.5 Photograph of a WiiMote with Nunchuk. 154
7.6 Different homogeneous point operators. 156
7.7 Sine-windowing function on an RGB image. 157
7.8 A Sobel-filtered image. 159
7.9 A Log-Reduce filter to compute the maximum. 160
7.10 Schematic view of a filter graph. 160
7.11 The editor used to write new filter modules. 161
7.12 Different Voronoi diagrams. 167
7.13 8SED vector templates. 168
7.14 Modified vector templates. 169
7.15 Ping-pong buffering for propagation filtering. 170
7.16 Example computation of a Voronoi diagram. 171
7.17 Worst-case error analysis for 2D vector propagation. 172
7.18 Empirical error analysis. 176
7.19 An example for artistic filters using Voronoi diagrams. 177

List of Tables

3.1 Timing and memory statistics for some of Michelangelo's statues. . . . 65
3.2 Timing and memory statistics for some volumetric data sets. 66

4.1 Timings and results of our terrain renderer. 88

6.1 Connectivity rules for the entangled grid topology. 125
6.2 Performance evaluation of our displacement method. 140

7.1 Veriables pre-defined by our filtering framework. 162
7.2 Performance of the Eucliden distance transform. 175

List of Algorithms

1 Linde-Buzo-Gray algorithm 24
2 Gauss-Seidel Mass-Spring Solver 132
3 GPU-based Mass-Spring Step 136
4 Vector Propagation algorithm 168

Chapter 1

Introduction

In 1965, intel co-founder Gordon Moore observed that the number of devices inside chips (including transistors, resistors, etc.) doubled every 12 months [Moo98]. Ten years later he further predicted that the number of transistors in chips—including modern CPUs—doubled every 24 months. This exponential growth has led to amazing aspects of today's society, including the capability to rapidly generate data so large in size that processing or displaying it consistently demands next-generation hardware as well as sophisticated, dedicated algorithms. To call large, highly detailed data sets a trend would be a clear understatement. Large data sets have been around for decades, and the actual meanings of the terms "large" and "detailed" have been constantly redefined by the amount of memory available on state-of-the-art computers. However, certain components in modern computers have not kept pace with the rapid growth in processing power, e.g., the typical amount and speed of memory, including core RAM as well as external storage. The result is that with the tremendous processing power currently available even in off-the-shelf PCs it is easy to generate data sets that cannot be explored interactively on the very same machine. Another aspect is the fact that the ratio between the size of data sets and the main memory found in typical PCs has been growing steadily during the last decades.

The rather recent introduction of mainstream graphics accelerators, collectively known as graphics processing units (GPUs) offers the potential to be able to explore these data sets visually and interactively. However, mostly—but not exclusively—due to the still very limited local video memory[1], GPUs also pose new challenges. GPU vendors thus realized a growing demand for higher flexibility and high-level language programming models stemming from both the programmers of video games and the

[1]GPUs with up to 4 GB video memory are available by now.

scientific community alike. As a result, GPUs have evolved from a very dedicated implementation of what is now known as the rasterization pipeline to very flexible and powerful devices. While they are still dedicated graphics chips and require programmers to follow specific paradigms tailored to the needs of realtime computer graphics, they can be programmed in high-level shader languages such as GLSL and HLSL [Kes08, Bly06] which are very similar to the C programming language [KR88]. For small and moderately sized data sets, the introduction and evolution of GPUs directly translated into a quantum leap in both performance and availability. Whereas before graphics capabilities had clearly been the domain of expensive work stations, computer graphics has now become ubiquitous. It is also worth noting that computer graphics has been exceptionally well received by society, especially taking into account that computers as such have not always been accepted without resentments. This is clearly demonstrated by recent successes such as Google Earth [Gooa], the video game industry, and many more.

In the academic community it is clear for quite some time that the real challenge are large, highly detailed data sets. Typically the demand is to visualize the data set entirely, with the ability to zoom in on specific parts. This has given rise to so-called level-of-detail (LOD) algorithms that seek to determine and display only the amount and detail necessary for a given set of camera parameters. Furthermore, interactivity has proven itself to be of paramount importance for several reasons. Firstly, interactivity allows the user to fully immerse into the process of data exploration. This is very helpful if the data is unkown or it is not a priori clear where interesting structures will be found. Secondly, humans are usually very productive in interactive environments. Possible reasons include that they are always provided with immediate visual feedback to their actions. This allows them to become accomodated with user interfaces of interactive simulations very quickly. Furthermore, if something "goes wrong", i.e., the presented image does not match the user's expectation, she or he can typically correct the fault rapidly if interactive interfaces are provided. Last but not least, humans tend to learn very well from trial-and-error, a learning paradigm that is best tapped using interactivity. At this point, we would like to stress the example of video games once more. When compard to board games it can be observed that video games typically offer a learning curve that is a lot more gentle, albeit they are not necessary less complex than board games. This effect is further fostered by the fact that only in interactive environments an adaptation of "rules" is feasible that take the user's behaviour into account. These rules can be plainly the game mechanics, or—in case of scientific visualization—they can be interaction metaphors changing depending on specific contextual features of the

data set to be visualized.

Interactivity can be achieved more or less straightforwardly by standard techniques that exploit recent GPUs for medium-sized data sets. Large, highly detailed data, however, requires the use of sophisticated, highly adapted visualization algorithms. One among many reasons is the von Neumann architecture of current computers. The GPU is connected to the CPU via a bus, i.e., AGP or PCI express. Since the GPU can only render efficiently from local video memory, the visible part of the data has either to reside on the GPU, or it has to be transferred there prior to rendering. In the optimum case data can be transmitted to the GPU while concurrently rendering another part of the data. As with all concurrent systems, this requires a sound timing of when to send data, and as with all bus-based systems, only necessary data should be transmitted to avoid that the bus becomes a bottleneck.

In this thesis we describe algorithms and methods to render such large and highly detailed data sets. The data sets presented in this work stem from a broad spectrum of sources, such as remote sensing, laser range scans, artistic content generation, digital photography, etc. To keep this thesis as self-contained as possible, we will address the data acquisition, the data processing, and, finally, the data visualization steps. We also show that by understanding the data acquisition step specific algorithms can be mandated that can exploit certain properties of the data. Specifically, this thesis will address the rendering of massive laser range point scans and large-scale terrain, the interactive editing and rendering of fractal terrain, the interactive editing of displacements on 2-manifold meshes, and a framework for rendering and filtering gigapixel images. In all these applications, similar techniques are used.

First of all, due to the sheer size of the data sets to be used, data compression in such a manner that the GPU can render directly from the compressed data stream is used. The most obvious result of integrating data decompression into the rendering step itself is a virtual increase of local video memory up to the point where actually the entire data set can reside on the GPU. Furthermore, compression allows to circumvent the bus bottleneck of von Neumann architectures by increasing virtual bandwidths across the bus. Last but not least, the host memory can be used more efficiently. It should be noted, however, that such compression schemes typically require bandwidths beyond those required when rendering directly from raw data. Luckily, local bandwidths of video memory currently surpass 100 GB/sec by a significant margin[2] and they are easily among the highest bandwidths in the entire system. Thus, decoding directly on the GPU will affect the overall performance least with respect to bandwidth requirements.

[2]For instace, NVIDIA specifies the video RAM to GPU bandwidth of the GeForce 280 GTX with 141 GB/sec.

Secondly, we show that interactivity is the key to intuitive user interfaces. By providing the user with immediate visual feedback, a WYSIWIG ("What-You-See-Is-What-You-Get") interface can be built to facilitate even such complex editing operations as to modify auxiliary functions of fractal synthesis.

Thirdly, specific algorithms that fall into the "GPGPU" (general purpose GPU) category are utilized to support rendering algorithms in their strive for interactivity. While we do not present general purpose algorithms on the GPU per se, we exploit the GPUs massive floating point processing potential to perform somewhat exotic tasks in terms of the classical rendering pipline—for instance a spring-mass system that runs on the GPU—in order to keep the entire application interactive. The appeal is generally that we try to move methods that are closely related to rendering, such as intermediate computations, directly to the GPU to avoid costly communication between the CPU and the GPU.

Last but not least, each application comes with its own rendering algorithm that is specifically tailored to the underlying data. These algorithms typically support rendering at different levels-of-detail (LODs) or dynamic geometry generation to avoid aliasing artifacts, as well as streaming data from host memory concurrently to rendering.

As we demonstrate in this thesis, the benefits of these techniques include the interactive editing or filtering of data without changing the underlying data representation (which is especially important for large data), the possibility to perform rapid prototyping in various applications, and of course all possibilities offered by visual realtime exploration of complex data, be it of scientific or artistic sources.

1.1 Research Publications and Contributions

The work described in this thesis has been partially published in a series of research papers. We now give an overview of the following chapters, including a list of contributors and academic publications.

The next chapter provides a broad overview of basic concepts and algorithms, that are not a major academic contribution of this thesis. However, they constitute basic building blocks and will be referenced by chapters following thereafter. The motivation for these overviews is the strive to make this thesis self-contained. In cases for which such ambitions must fail because the topic in discussion has been under active research for several decades, we at least provide references for further readings.

Chapter 3 deals with the compression and rendering of large point scans and volume

data sets. Two papers were published in collaboration with Jens Krüger and Rüdiger Westermann in that context. The first paper [KSW05] deals with the compression and rendering of the point scans in the Digital Michelangelo Project, while the second one [KSW06] extends the method towards the compression and rendering of isosurfaces. Also, the second paper discusses memory and runtime complexity as well as the geometric properties of the underlying clustering in more detail. In contrast to these publications, Chapter 3 provides a more concise description of the graph-theoretical foundations of the method, and it provides ready-to-use algorithms to perform the resampling and clustering steps in the method. Furthermore, the chapter in this thesis is a lot more self-contained in comparison to previous publications, since an overview of closest-sphere packings, an in-depth discussion of the differential encoding, and a concise description of the normal estimation process are provided.

Chapter 4 discusses a system to render large-scale terrain provided as digital elevation maps using nested mesh hierarchies and progressive transmission of data into video memory. It was partly published in collaboration with Rüdiger Westermann [SW06]. The chapter in this thesis has been extended by providing some additional details, and it now contains a description of the algorithm used to generate zero-area triangles in order to stitch chunks together. Furthermore, some parts of the system were re-used for the more recent publication by Christian Dick, Jens Schneider, and Rüdiger Westermann [DSW08], although this particular publication was not used for this thesis.

The synthesis of infinite terrain is the subject of Chapter 5. Parts of it have been published in collaboration with Tobias Boldte and Rüdiger Westermann [SBW06]. In this chapter, a WYSIWIG interface is proposed to allow interactive editing of fractal terrain, including the real-time synthesis of geotypical textures. A projected grid approach is utilized for rendering.

Chapter 6 describes a system to interactively edit and modify any given 2-manifold meshs by using displacements. Unlike previous approaches we do not rely on a homogeneous refinement of the base mesh, e.g., by evaluation of a subdivision kernel. This is made possible by first tracing a locally parameterized patch on the base mesh. This patch is then regularly tesselated and serves as the domain for a displacement height field. By utilizing a fragment-based cutout technique, displacements both along the positive and negative normal direction are possible. Also, this method avoids depth-peeling, a rather expensive operation that is frequently needed to establish correct visibility order. The work described in this chapter is the result of a collaboration with Joachim Georgii and Rüdiger Westermann. It has not yet been published.

Chapter 7 describes a system to render and filter gigapixel-sized images. It is closely

related to [DSW08], but the basic framework has not yet been published. The only subset of this chapter that has been published is the GPU-based propagation filter to compute discrete euclidean distance fields [SKW09]. The methods described in this chapter are a collaboration with Dominik Meyer and Florian Wendel, who contributed to this project during their bachelor and research projects, Christian Dick, who provided the terrain engine described in [DSW08], and Rüdiger Westermann. The propagation filter was developed jointly with Martin Kraus and Rüdiger Westermann.

Finally, we summarize the topics covered by this thesis, provide results, and give interesting directions for future research.

Chapter 2

Basic Techniques

In this chapter we briefly discuss recent advances of data acquisition techniques. This is accompanied by a broad overview of some of the more basic methods used in the remainder of this thesis, and a discussion of the programmable rendering pipeline as exposed through the DirectX 10 API. This chapter does not represent an academic contribution per se, but serves to make the thesis as self-contained as possible.

2.1 GPUs and Graphics APIs

This thesis describes methods that have been developed and validated using either the OpenGL or Direct3D 10 API [SA08, Bly06]. Since both APIs share very similar concepts, we describe the Direct3D 10 rendering pipeline in this section and we use Direct3D terminology when describing general concepts. Only in places where the details of the actual implementation are discussed or where the decidedly better documentation of OpenGL is beneficial will OpenGL terminology be used.

The Direct3D 10 rendering pipeline is depicted in Figure 2.1. In this figure, blue blocks correspond to the parts of the pipeline which are fully controlled by the programmer, i.e., the programmable vertex, geometry, and pixel shader stages, as well as the host- and GPU-side memory objects. GPU-sided parts are programmed in a high level shader language (typically GLSL or HLSL [Kes08, Bly06]) that is similar in style to the C programming language [KR88]. Gray blocks correspond to stages that operate in a configured-function mode, i.e., the operation of these stages is guided by so-called *state objects*. These state objects contain configuration data such as filtering and mipmapping parameters in case of a sampler state object. Although multiple such state objects can be stored at the GPU at one time, only one such object can be active per stage. In total, five different states are maintained: The input-layout state, the rasterizer state, the

blend state, and the sampler state.

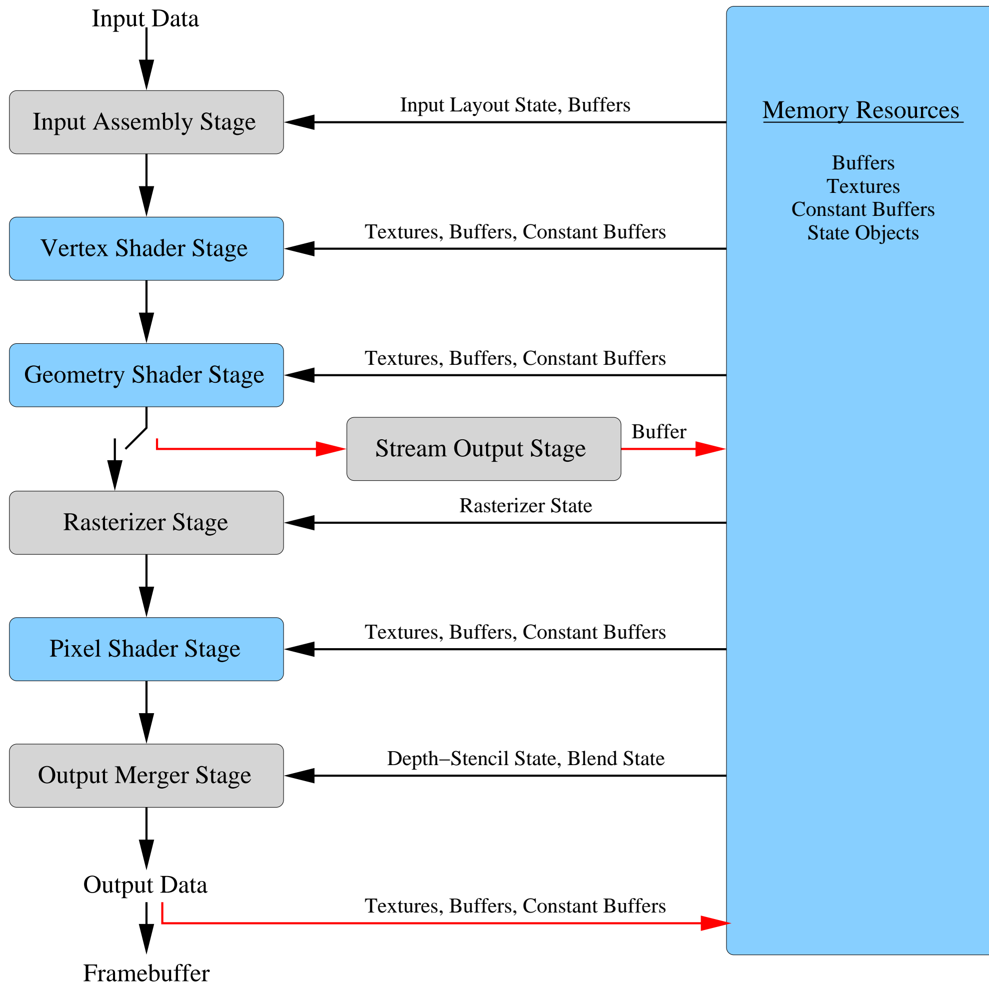


Figure 2.1: *The Direct3D 10 rendering pipeline. This pipeline is implemented by essentially all recent GPUs. The parts highlit in blue can be fully managed by the user. Specifically, the vertex, geometry, and pixel shader stages are fully programmable, and the user has full control of host- and GPU-side memory objects. Grayed blocks correspond to stages that operate in a configured-function mode.*

The pipeline follows the basic concepts and paradigms of rasterization-based computer graphics [FvDFH95]. For each draw call, an user-defined input buffer containing vertex data and optionally index data (to render shared vertex formats) is traversed. From these vertices, primitives are formed in the input assembly stage.

The vertices of these primitives are then passed on to the vertex shader stage. Here, each vertex can be modified, but no new data can be generated. Vertices can only be removed in this stage by transforming them to positions outside the viewing frustum.

In the geometry shader, the programmer can then access the entire primitive. Possible primitive configurations include simple triangles, triangles with neighborhood information, etc. The advantage of the geometry shader stage is that *all* vertices of the current primitive can be fully accessed, that new primitives can be generated (so-called *geometry amplification*), and that primitives can be removed from the pipeline. The major disadvantage of the geometry shader stage is—as of writing—its inferior performance. Although it can be expected that the performance of this stage will improve in the near future, it is currently worth the effort to emulate certain parts of it by utilizing multi-pass techniques and render targets. The programmer can choose to either directly output the primitives processed and/or generated into the rasterization stage, or to store primitives in a *stream output*—a specific type of buffer that resides in GPU-memory—for subsequent rendering passes

In the rasterizer stage, primitives are clipped against the view frustum and fragments—informally speaking pixels or sub-pixels augmented with additional information such as 3D position, interpolated texture coordinates etc.—are generated.

In the following stage, the pixel shader, no new fragments can be generated, but the user can choose to discard selected fragments. As soon as each fragment's depth is known, it can be depth and stencil tested, thereby conditionally discarding it. Note that fragments can be depth-tested even prior to the pixel shader, if depth values are not modified by the shader (so-called *early depth testing*).

Finally, all "surviving" fragments are merged in the output merger stage that also performs the configurable blending of incoming pixels.

Last but not least, the output data, i.e., pixels, can be written to the framebuffer or to textures and/or buffers. Note that these options are not mutually exclusive, since multiple render targets can be specified. As can be seen in Figure 2.1, the Direct3D 10 pipeline offers multiple possibilities (marked by red arrows in the figure) for *feedback loops*, i.e., loops that re-use data from previous rendering passes. This is an important concept and has become indispensable for most multi-pass algorithms.

One pecularity in Figure 2.1 still needs to be addressed. Direct3D 10 provides two fundamentally different operations to retrieve the data stored in textures or buffers. The first one is called *Load* and just fetches data without any interpolation or filtering. The second one is called *Sample*. This term actually refers to a class of instructions that are further configured by a smpler state. Sample operations perform filtering and interpolation and are hence defined for textures only. This additional functionality is the reason that we strictly differentiate between textures and buffers, although textures are in fact buffers in Direct3D 10.

As a final remark, we would like to note that recent GPUs supporting Direct3D 10 are required to support the full pipeline. This includes single-precision floating point operations adhering to the IEEE 754 standard, with the only exception of functions that are typically implemented using iterative approximations, namely square-root and division. These functions may deviate from the IEEE 754 standard by the last few bits in the result's mantissa. Furthermore, Direct3D 10 requires support for 32 bit integers—signed and unsigned—, including full bit arithmetic. Consequently, most algorithms that can be implemented on a CPU using only these data types can also be implemented on a GPU, albeit the latter implementation might not be straightforward. The reason is that concurrent read- and write access to buffers and textures is not yet possible in either Direct3D 10 or OpenGL 3.0. Direct3D 11 has been announced to address these issues by the so-called *compute shader* [Boy08], but hardware support for this API is not available as of writing.

2.2 Digital Photography and CCDs

A CCD (charge-coupled device) is a semiconductor device first conceived in 1970 by Willard Boyle and George E. Smith. It consists of a series of Germanium (or of comparable material) electrodes which are separated from a acceptor-impurity doped silicium (p-Si) layer by a quartz (resistor) layer. A schematic overview is given in Figure 2.2.

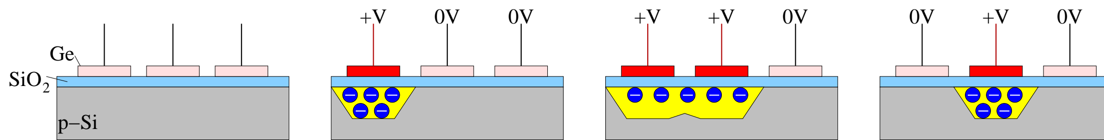


Figure 2.2: *Schematic view of a CCD. Charges (blue) reside in quantum wells (yellow) in the p-Si layer. The extent of these wells are controlled by the Germanium anodes, which are separated by a quartz layer from the p-Si layer. By the operations depicted from left to right it is thus possible to propagate charges in a controlled manner.*

The main functionality of such a device is that charges (depicted by a collection of electrons in Figure 2.2) can be transported in the p-Si layer in a controlled way by connecting the electrodes to positive current. Charges in the p-Si layer can be initially generated by a variety of other devices, most notably by photo-diodes. In this manner, two-dimensional CCD arrays can be constructed, in which charges are generated by exposing an array of photo-diodes to visible light. Then, light exposure is inhibited, thereby "freezing" charges in the CCDs. Thus, an image of the exposure can be stored

in the CCD array. The charges corresponding to that image can then be read and interpreted *sequentially* by means of digital signal processing (DSP) chips. The result are so-called CCD sensors (see Figure 2.3 for a picture of an actual sensor) that are commonly used in commercially available digital cameras.

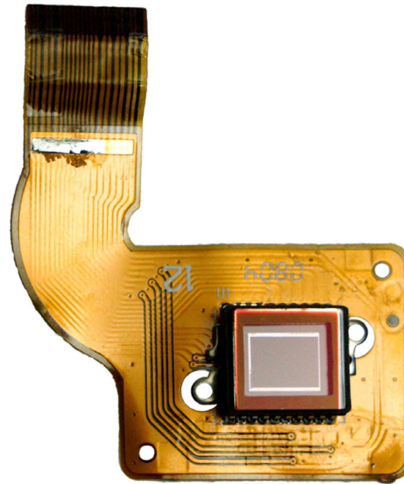


Figure 2.3: *Photograph of a CCD sensor. Image under the Wikimedia Commons licence.*

The importance of the CCD lies within the fact that the DSPs do not need to access each single one of the charge-cells at once, thereby resulting in a feasible design of such DSPs (see Figure 2.4 for a schematic view of a Full-Frame CCD). Miniaturization has lead to CCD sensors capable of capturing dozens of Megapixels while covering only a few square centimeters. The result is that digital cameras are now en route to become an equivalent replacement for traditional, analog camera systems, ranging from amateur to professional use. As a result, large images have been become ubiquitous, and mosaicking and registration techniques [KUDC07] have been developed to combine multiple images into a single, ultra-high resolution image comprising several gigapixels each.

Analog cameras can also be used to acquire such gigapixel resolutions. For instance, the Gigapxl Project [Fli] uses high-definition analog cameras to obtain high-resolution images. These images are then scanned afterwards, resulting in single-exposure images comprising up to 4 Gigapixels.

The advantage of single-exposure photographs is obvious, since visible changes in the observed scenery do not occur during a single exposure. This is in contrast to multiple-exposure mosaicked digital images, in which, in case of urban photography, cars and people change positions and thus lead to undesired artifacts.

Nevertheless, both techniques have been employed to obtain images at very high

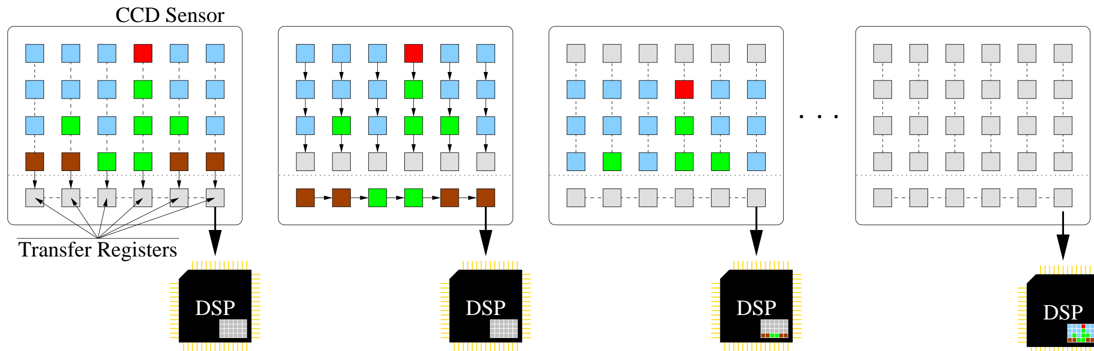


Figure 2.4: *Schematic view of a Full-Frame CCD. In this example, the CCD-sensor consists of six CCD columns plus one six-cell of row of transfer registers. Each row's charges are advanced one slot downwards, thereby filling the transfer registers. This step requires multiple cycles in order to keep the charges separated from each other. The transfer registers are then "emptied" in a similar manner, first into a signal amplifier (not depicted), and then into the DSP. The DSP forms the final image in multiple such steps. Note that, although other CCD sensor types exist, the basic mode of operation is always similar. Also note that photo-CCD cells are typically monochromatic, i.e., there are usually different kinds which are exclusively sensitive to different wavelengths.*

resolutions; and these images will eventually define a new standard. By now already, panoramic views comprising several tens to a few hundreds of megapixels can be generated by ambitioned amateurs. Camera vendors frequently include the necessary software with their products for free. This results in an entirely new type of data for which the classical image processing operations have to be redefined, e.g., viewing, filtering, etc. However, commercial image processing tools currently available are typically not yet suited for this amount of data, and their paradigms might proove completely unfit to deal with these images. Among others, these paradigms include the attempt to store the entire image in virtual memory, to compute adaptive resolutions on the fly in order to fit the image onto the view port, and to keep multiple copies in external memory to allow for "Undo"-functionality. Consequently, this type of large data has recently kindled a large academic interest, in the course of which we evaluate possibilities to display and filter such images. Our approaches are documented in Chapter 7 of this thesis.

2.3 Laser Range Scanning of 3D Objects

In Chapter 3 the compression and rendering of large point scans will be discussed. There are various ways to obtain such point clouds. By far the most prominent of such

methods is laser range scanning. In this section we will briefly describe the technical process of acquiring point clouds using a laser scanner. Since the exhaustive discussion of recent advances in laser range scanning would be beyond the scope of this thesis, only the fundamental ideas will be presented. For a thorough overview we would like to refer the reader to the 3D photography course held at SIGGRAPH 2000 [BCD+].

Scanning systems are typically classified by the hierarchy depicted in Figure 2.5. Laser range scanning is thus a non-contact, reflective, optical shape acquisition method. Optical acquisition methods can be further separated into *active* and *passive* systems, depending on whether they provide their own illumination (active) or not (passive). We discuss two basic classes of active systems here, namely those based on measuring the time-of-flight of a pulsed laser, and the "classical", triangulation-based methods.

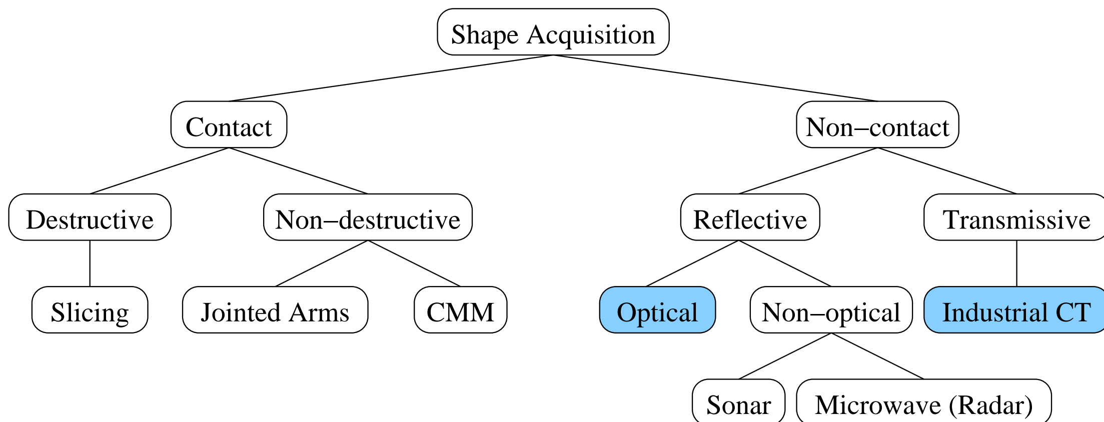


Figure 2.5: *A taxonomy for shape aqcuisition methods. Reproduced from [BCD+]. All data sets in this thesis stem from optical methods or CT scanners.*

LIDAR systems. LIDAR (Light Detection And Ranging) systems are time-of-flight based. A pulsed laser is deflected using a mirror into the direction in which a positional sample should be aquired. Then, the pulse is reflected by the environment to be measured and returns to the detector. The time between emitting and detecting the pulse is measured. Since the direction of the laser and its time-of-flight is known, the position of the occluder relative to the measuring device can be computed.

The precision of LIDARs is mainly dependent on three factors. Firstly, the speed of light has to be known precisely, which implies a specific knowledge of the medium through which the laser pulse travels. Secondly, time has to be measured with very high precision, since, for instance, achieving a resolution of less than 1 meter in a

vacuum is already equivalent to being able to measure time with an error margin of about 3 picoseconds. Thirdly, since pulses are emitted periodically, errors may occur if the actual time-of-flight exceeds the time between pulses. The reason is that in this case the current pulse is emitted before the previous one returns. Consequently, the time-of-flight measured will be that between the current's pulse emission and one of the previous pulses' returns (see also Figure 2.6).

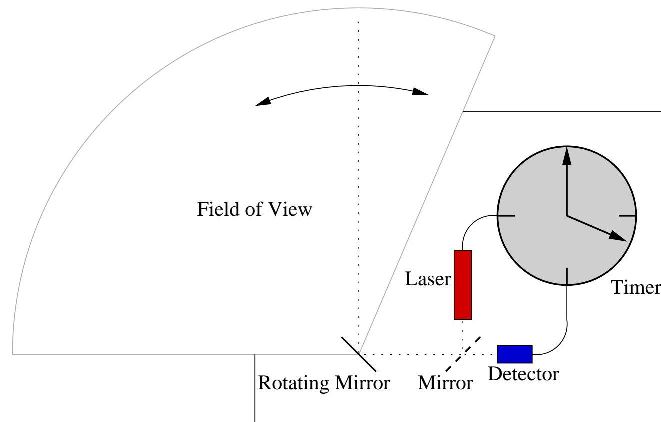


Figure 2.6: *Schematic view of a LIDAR system. The laser emits a pulse that is eventually reflected by the environment and returned. A detector is then used to measure the fime-of-flight. After each pulse return, the rotating mirror can be advanced in order to sweep the field of view, similarly to a radar system.*

Last but not least, as with all optical acquisition methods, LIDARs require a clean reflection of the object to be scanned. Also, it is assumed that the object to be scanned reflects a sufficiently large portion of the pulse back into the incoming direction, since the detector is typically close to the emitter. Furthermore, beam divergence might require additional correction computations. LIDAR systems are thus best suited to scan immobile objects, although a LIDAR system was used to achieve the particular look of the music video "House of Cards" by Radiohead. The most prominent use of LIDARs, however, was the MOLA mission (Mars Orbiter Laser Altimeter) [NAS] with its goal to achieve precise digital elevation maps of Mars. Typically, between 10,000 and 100,000 points samples per second can be taken by LIDAR systems, each at a resolution on the order of a few decimeters.

Triangulation-based 3D laser scanners. These scanners consist of a laser and a detector (i.e., a digital camera), whose position relative to the laser is well known. The laser then highlights a section of the object to be scanned by a planar sweep (see also Figure 2.7). This results in a curve in the image acquired by the camera. For each pixel row con-

taining a highlit point, the plane spanned by the camera position and the pixel row can then be intersected with the laser sweep plane to yield a parametric formulation of the laser beam. The ray determined by the camera position and the highlit pixel can then be intersected with the laser beam, which is a 2D problem. The final result is then a 3D coordinate for each highlit pixel. Typically, the object is rotated and/or translated relative to the measurement device (comprising camera and laser) in order to obtain a full scan.

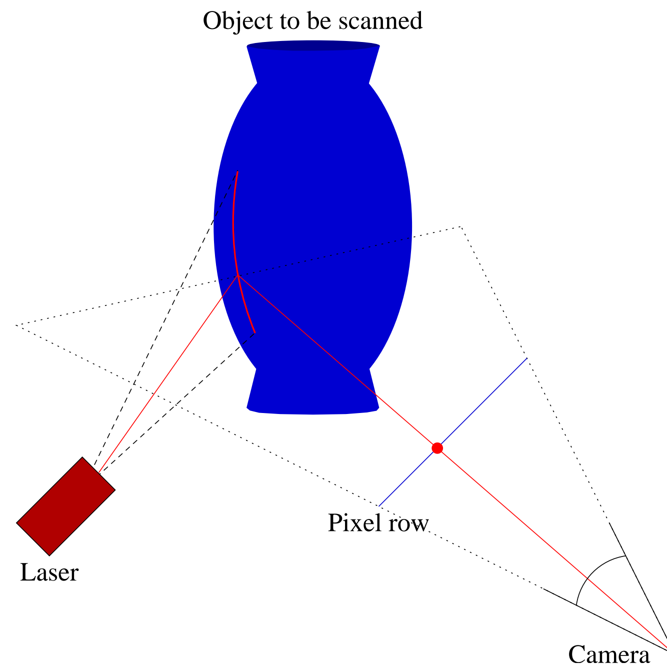


Figure 2.7: *Schematic view of a triangulation-based scanner.*

Although this epipolar setting is conceptually simple, a series of problems can arise. The most obvious one is that multiple highlights can occur in one row, in which case only an intensity maximum can be chosen to determine a 3D coordinate. This defect will generally occur for materials exhibiting strong subsurface scattering properties, but the effect is not limited to such materials. Furthermore, parts of the laser sweep can be obstructed, which might result in incomplete scans. Obviously, the precision of each point measurement is affected if either the laser or the camera attain a close-to-oblique angle with the object's surface. This can be addressed during the reconstruction process by assigning a confidence value to each point as soon as the object's surface normal can be estimated. Last but not least, any calibration error seriously affects the overall performance of the system.

All problems set aside, the methods to compute 3D positions using laser range scan-

ners have matured enough to yield very high precision position, colors, and normals. Such technology was employed in the Digital Michelangelo Project [Sta], of which some data sets have been used in Chapter 3

2.4 Acquisition of Digital Elevation Maps

Besides airborne LIDAR systems (see also Section 2.3), passive optical systems are frequently used. Among the more recent, prominent examples is the High Resolution Stereo Camera (HRSC) developed by the DLR (Deutsches Zentrum für Luft- und Raumfahrt) and built by EADS (European Aeronautic Defence and Space Company). It was most prominently used for the Mars Express mission [IfPb]. Among other data sets, the Paris data set used in Chapter 4 was obtained using a low-altitude HRSC scan.

The basic acquisition principle, as depicted in Figure 2.8, relies on 5 to 9 CCD line sensors, of which 3 to 5 are typically panchromatic and dedicated to obtaining 3D positions through stereo photogrammetry. The remaining sensors are monochromatically filtered and their output can be combined to yield aerial color textures.

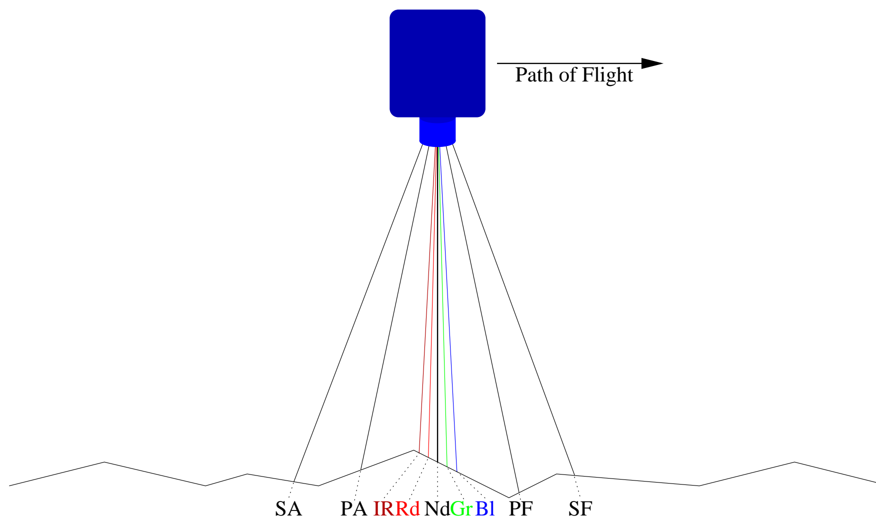


Figure 2.8: *Schematic view of the HRSC-AX. The camera operates according to the so-called pushbroom principle. Nine CCD line sensors are used, each with a resolution of 12,000 pixels. Four of the nine channels are filtered in order to obtain monochromatic measurements, i.e., to obtain aerial textures. The remaining four channels (namely SA, PA, Nadir, PF, SF) measure panchromatic light and are used for photogrammetric reconstruction of the height field. Detailed specifications can be obtained from [IfPa].*

The basic concept behind stereo photogrammetry is similar to 3D laser scanning by triangulation, however, salient points and their matchings are not intrinsically given.

Instead, for multiple scan lines, a best-possible match has to be performed in order to obtain the final 3D positions. For a thorough discussion of airborne data acquisition as well as its physical realization, data processing models, and limitations we would like to refer to the two-part tutorial of Claus Brenner [Bre06a, Bre06b].

The most recent HRSC-AX has 9 CCD sensors with 12,000 pixels each. Typical resolutions achieved at 6,000 meter altitude are about 30 cm. The HRSC-A used in the Mars Express mission has 9 sensors with 5,184 pixels each. At an altitude of 270 km and more, the distance between pixels on the ground is about 12 m, while the intra-line spacing is about 2.3 m [IfPb]. Thus, highly detailed data sets can be obtained on planetary scales, including geometry *and* textures.

2.5 CT Scanning

The abbreviation CT refers to computed tomography, an imaging method frequently used in medicine. The basic concept of a CT scanner is to obtain a large series of 2D X-ray images centered around a single axis of rotation. Each of these images contains for each of its pixels a measurement of the X-ray extinction due to the material. Mathematically, this measurement constitutes a line integral along the ray

$$\boldsymbol{p}(s) \quad = \quad \boldsymbol{p_0} + s\boldsymbol{d}, \tag{2.1}$$

where s is the X-ray's parameter, $\boldsymbol{p_0}$ is the position of the emitter, and $\boldsymbol{d}$ is the ray's direction. The observed intensity is then given by the absorption law

$$I\left(x,y\right) \quad = \quad I_0 e^{-\int_{s_{\text{enter}}}^{s_{\text{exit}}} \mu(s)ds}, \tag{2.2}$$

where I_0 is the X-ray intensity at the emitter, s_{enter} and s_{exit} are the ray parameters at the emitter and detector, and $\mu(s)$ is the extinction coefficient at position $\boldsymbol{p}(s)$.

A scalar data set on a 3D Cartesian grid can then be obtained by computing the inverse *Radon transformation*. The Radon transformation of a function $f(x,y)$ is given by

$$\begin{aligned} R(k,\phi)\left[f(x,y)\right] \quad &= \quad \int\int f(x,y)\delta(x\cos\phi + y\sin\phi - k)dxdy, \text{ where} \\ \delta(b-a) \quad &: \quad \int f(b)\delta(b-a)db = f(a) \text{ for any function f.} \end{aligned} \tag{2.3}$$

To see the relation to CT scanning, observe that $R(r,\phi)\left[f(x,y)\right]$ computes line-integrals

along a set of lines $L(\phi, k)$, where $L(\phi, k) := \{(x, y) : x \cos \phi + y \sin \phi - k = 0\}$. The geometry of these lines is depicted in Figure 2.9.

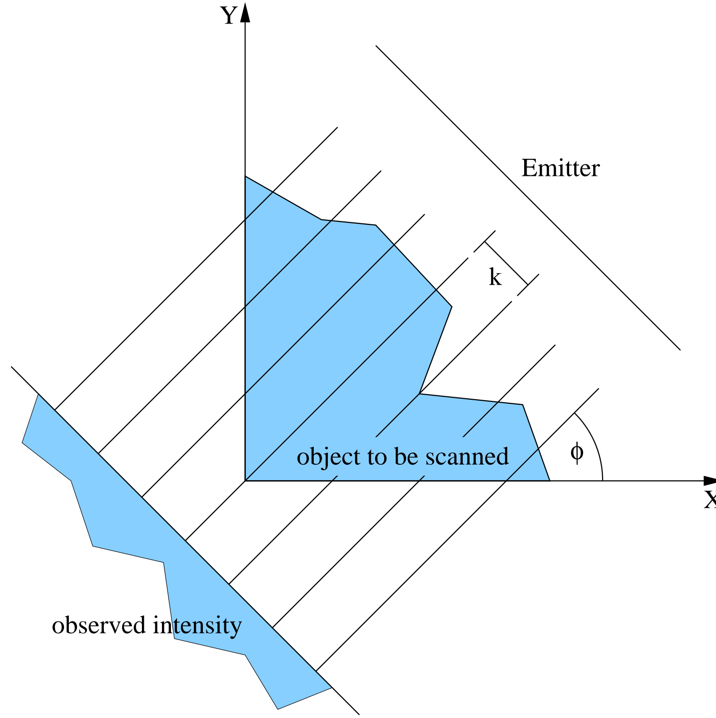


Figure 2.9: *Relation between Radon transformation and CT scanning. The parallely emitted X-rays are detected on a discrete grid. Each detector cell thus measures the line integral along a line that is rotated by ϕ and offset by a multiple of k.*

The observed intensity (see Equation (2.2)) is first transformed into an *attenuation profile*,

$$\rho(x, y) = \ln\left(\frac{I(x, y)}{I_0}\right) = \int_{s_{\text{enter}}}^{s_{\text{exit}}} \mu(s). \tag{2.4}$$

Then, the inverse Radon transform of the attenuation profile is computed to reconstruct $\mu(s)$. This can either be done using algebraic reconstruction [MYC95] or filtered back-projection [SBMW07]. For a detailed discussion of these methods, we would like to refer the reader to [Sch07].

The result is a grid of so-called *voxels*, volume elements, of which each stores a scalar value. This value is typically given as a 12 bit integer ranging from -1024 to 3071 and measured in so-called *Hounsfield units* (HU). Hounsfield units are normalized such

that

$$\mathrm{HU}(\mu) \quad := \quad 1000\frac{\mu - \mu_{\mathrm{water}}}{\mu_{\mathrm{water}}}. \tag{2.5}$$

Consequently, a value of 0 HU corresponds to water. Typical values are -1000 HU for air, about 400 HU for cancellous bone and up to 2000 HU for cranial bone. Titanium implants are around 1000 HU, while steel implants typically extinct X-rays almost completely. Since the absorption law also states that

$$\mu \quad \propto \quad \frac{z^{\beta}}{(h\nu)^{3}},\ 3 \leq \beta \leq 4, \tag{2.6}$$

where z is the atomic number of the material to be measured and $h\nu$ is the energy of the emitted photons, it is under certain circumstances possible (i.e., knowledge of ν and β) to clearly define the observed material.

There are two dominant types of CT scanners in use by today, namely the *helical* (also called *spiral*) CT, and its rather novel improvement, the multislice CT (also see Figure 2.10).

Figure 2.10: *Siemens Somatom 16 CT Scanner. The scanner depicted comprises of 16 scanning rings. Image under the Wikimedia Commons licence.*

Helical CT scanners. These scanners consist of a ring, along which a gantry-mounted emitter can freely move. The patient is placed on a examination couch that is drawn automatically through the ring. The combination of the circular gantry motion and the linear transportation of the patient results in a helical movement of the scanner around the patient. Typical scanning times range from 20 to about 60 seconds, i.e., it is clearly possible to obtain a scan without respiratory artifacts using this technology.

Multislice CT scanners. These scanners are a refinement of the helical scanners. They share the same mode of operation, but comprise multiple scanning gantries. For instance, the Siemens Sensation Somatom16 in Figure 2.10 comprises 16 such scanning devices. The obvious advantage is that rotation speed of the gantries can be increased without compromising the scanning quality. A less obvious advantage is that it is possible with some devices to obtain a single scan for two different radiation energies. This allows to separate, for instance, Titanium implants from bone structures in the final image by application of the proportionality given by Equation (2.6). The result of all multislice scanners is a drastically increased resolution when compared to helical scanners. Current state-of-the-art scanners offer an isotropic resolution of 0.35 mm and below while being able to maintain a scanning speed of more than 18 cm/sec.

The consequence are extremely large data sets, often comprising several gigabytes, at very high resolutions even for moderate radiation doses. For inanimate objects, where radiation doses are not of concern, even higher resolutions can be obtained. In Chapter 3 we use a wholebody scan by courtesy of Siemens Corporate Research, Inc., Princeton comprising $512^2 \times 3172$ voxels, but the largest-available scans currently comprise up to 4096^2 samples for each slice and similar amounts of slices.

2.6 Principal Component Analysis

The process of computing an Eigendecomposition of the (auto-)covariance matrix of multidimensional stochastic processes is typically referred to as *principal component analyisis* (PCA). The result of such an analysis are *principal directions*. If the underlying data is rotated into the space spanned by the principal directions, correlations between the components of the now-rotated data points are proovably minimized. Consequently, such an analysis is an important building block for data compression algorithms. In this context, it is also referred to as the Karhunen-Loève transformation [Say00] and can be approximized by a discrete cosine transformation (for highly auto-correlated data) or a discrete sine transformation (for highly auto-anticorrelated data).

Furthermore, the PCA can be employed to compute linear orthogonal regressions, i.e., line and hyperplane fits. The key observation from a geometric point of view is that the principal directions associated with large Eigenvalues define the global trend of the data, while directions associated with small Eigenvalues contribute only negligibly to the general distribution of data points. Consequently, fitting a hyperplane to a data set is equivalent to computing the smallest Eigenvalue and its corresponding principal direction.

Assuming data points of an n-dimensional vector space $\mathcal{X} := \{\boldsymbol{x_i}\}_{i=1}^{N} \subseteq \mathbb{R}^n$ with $N = |\mathcal{X}|$, the covariance matrix $\mathcal{A}$ is typically computed in two steps. First, the average of $\mathcal{X}$ is computed, i.e.,

$$\bar{\boldsymbol{x}} := \frac{1}{|\mathcal{X}|}\sum_{\boldsymbol{x}\in\mathcal{X}} \boldsymbol{x}. \tag{2.7}$$

Then $\mathcal{A}$ is computed by

$$\mathcal{A} := \frac{1}{|\mathcal{X}|-1}\sum_{\boldsymbol{x}\in\mathcal{X}} (\boldsymbol{x}-\bar{\boldsymbol{x}})(\boldsymbol{x}-\bar{\boldsymbol{x}})^T. \tag{2.8}$$

The result is a real, symmetric, positive definite matrix. Thus, $\mathcal{A}$'s Eigenvalues λ_i are known to be positive and real. Denoting the set of unit principal directions by $\{\widehat{\boldsymbol{e}}_{\boldsymbol{i}}\}_{i=1}^{n}$, $\mathcal{A}'s$ Eigendecomposition is given by

$$\begin{aligned}
\mathcal{A} &:= \sum_{i=1}^{n} \widehat{\boldsymbol{e}}_{\boldsymbol{i}}\lambda_i\widehat{\boldsymbol{e}}_{\boldsymbol{i}}^T, \\
\langle \widehat{\boldsymbol{e}}_{\boldsymbol{i}}, \widehat{\boldsymbol{e}}_{\boldsymbol{j}} \rangle &= \delta_{ij},\ \forall i,j \in \{1,\dots,n\}, \\
\mathcal{A}\widehat{\boldsymbol{e}}_{\boldsymbol{i}} &= \lambda_i \widehat{\boldsymbol{e}}_{\boldsymbol{i}}\ \forall i \in \{1,\dots,n\}.
\end{aligned} \tag{2.9}$$

Here, δ_{ij} denotes the Kronecker delta, i.e., the $\widehat{\boldsymbol{e}}_{\boldsymbol{i}}$ form a normalized orthogonal basis of the non-kernel subspace of $\mathcal{A}$ with $\operatorname{rank}(\mathcal{A})$ dimensions. By convention, principal directions and Eigenvalues are sorted such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$. The best-possible approximation of $\mathcal{A}$ using $k \leq n$ Eigenvalues can then be computed by

$$\mathcal{A} \approx \sum_{i=1}^{k} \widehat{\boldsymbol{e}}_{\boldsymbol{i}}\lambda_i\widehat{\boldsymbol{e}}_{\boldsymbol{i}}^T,\ 1 \leq k \leq n. \tag{2.10}$$

To see that a linear orthogonal regression can in fact solved by PCA, assume an

$n-1$ dimensional hyperplane

$$\eta_{n-1} = \left\{ \boldsymbol{x} : \boldsymbol{x}^T \boldsymbol{\nu} = \boldsymbol{x_0}^T \boldsymbol{\nu} \right\}, \tag{2.11}$$

where $\boldsymbol{\nu}$ denotes the plane's normal, and $\boldsymbol{x_0}$ is one arbitrary point of η_{n-1}. Trying to solve Equation (2.11) for all $\boldsymbol{x} \in \mathcal{X}$ then results in an overdetermined system of equations,

$$(\boldsymbol{x} - \boldsymbol{x_0})^T \boldsymbol{\nu} = 0 \; \forall \boldsymbol{x} \in \mathcal{X}, \tag{2.12}$$

which can be approximately solved by minimizing the sum of squared errors

$$\begin{aligned}
\varepsilon^2 &:= \sum_{\boldsymbol{x} \in \mathcal{X}} \left((\boldsymbol{x} - \boldsymbol{x_0})^T \boldsymbol{\nu} - 0 \right)^2, \Rightarrow \\
\varepsilon^2 &= \sum_{\boldsymbol{x} \in \mathcal{X}} \boldsymbol{\nu}^T (\boldsymbol{x} - \boldsymbol{x_0}) (\boldsymbol{x} - \boldsymbol{x_0})^T \boldsymbol{\nu}, \Rightarrow \\
\varepsilon^2 &= \boldsymbol{\nu}^T \left(\sum_{\boldsymbol{x} \in \mathcal{X}} (\boldsymbol{x} - \boldsymbol{x_0}) (\boldsymbol{x} - \boldsymbol{x_0})^T \right) \boldsymbol{\nu}, \Rightarrow \\
\varepsilon^2 &= \boldsymbol{\nu}^T \mathcal{B} \boldsymbol{\nu}, \text{ where} \\
\mathcal{B} &:= \sum_{\boldsymbol{x} \in \mathcal{X}} (\boldsymbol{x} - \boldsymbol{x_0}) (\boldsymbol{x} - \boldsymbol{x_0})^T
\end{aligned} \tag{2.13}$$

Clearly, choosing $\boldsymbol{\nu}$ to be the Eigenvector corresponding to the smallest Eigenvalue λ_n of $\mathcal{B}$ minimizes ε^2 to λ_n. Obviously, $\boldsymbol{x_0}$ should be chosen such as to minimize the smallest Eigenvalue of $\mathcal{B}$. This is the case if $\boldsymbol{x_0}$ is a best approximation of the set $\mathcal{X}$ in the least-squares sense, i.e.,

$$\boldsymbol{x_0} = \underset{\boldsymbol{y} \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{\boldsymbol{x} \in \mathcal{X}} (\boldsymbol{x} - \boldsymbol{y})^2 . \tag{2.14}$$

Solving Equation 2.14 yields $\boldsymbol{x_0} = \bar{\boldsymbol{x}}$ and thus $(|\mathcal{X}| - 1) \mathcal{A} = \mathcal{B}$. Hence, the plane η_{n-1} that approximates $\mathcal{X}$ best in terms of the squared error is determined by $\bar{\boldsymbol{x}}$ and the principal direction $\widehat{\boldsymbol{e_n}}$ corresponding to the Eigenvalue $\lambda_{\min}$ with the smallest modulus.

The PCAs performed for this thesis are typically small; most of the time only 3×3 systems need to be solved. Since

$$\begin{aligned}
\lambda_i (\mathcal{A}) &= \lambda_i^{-1} \left(\mathcal{A}^{-1} \right) \; \forall i = 1, \ldots, n, \text{ and thus} \\
\lambda_{\min} (\mathcal{A}) &= \lambda_{\max} \left(\mathcal{A}^{-1} \right),
\end{aligned} \tag{2.15}$$

it is fully feasible to compute the smallest Eigenvalue/vector pair by means of a matrix inversion of $\mathcal{A}$ followed by power iteration or Rayleigh quotient iteration [PTVF02a, GVL96a].

2.7 Vector Quantization

In several places of this thesis, vector quantization will be employed as a back-end to other, more specific data compression techniques. To keep this thesis self-contained, parts of this author's diploma thesis [Sch03] will be reproduced to give a compact overview on the topic. For a more thorough overview we would like to refer the reader to Gray and Neuhoff's excellent quantization survey [GN98] or to the books of Sayood [Say00] and Gersho and Gray [GG92].

In this section the following symbols will be used.

$\mathbb{N}$: The set of natural numbers, counting from 1.

$\mathbb{R}$: The set of real numbers.

$\Re$: An alphabet on which a vector quantizer will operate. Usually $\Re \subset \mathbb{R}$.

$\Im$: A countable index set. Usually $\Im \subset \mathbb{N}$.

I: An ordered input data set to a vector quantizer.

$\alpha_{\mathcal{C}}$: An encoder mapping $I \rightarrow \Im$.

$\beta_{\mathcal{C}}$: A decoder mapping $\Im \rightarrow \mathcal{C}$.

$\mathcal{C}$: A codebook (an ordered, countable set of vectors).

$\mathcal{V}_i$: $\mathcal{V}_i := \{\boldsymbol{x_j} \in I : \alpha_{\mathcal{C}}(\boldsymbol{x_j}) = i\}$, (i.e., a quantization cell).

δ: A distance metric. Usually $\delta \equiv \|\cdot\|_2^2$.

$\circ$: Concatenation of functions. Read $a \circ b$ as "apply a before b".

$\langle , \rangle$: Standard dot-product.

$\star$: Convolution between functions.

Basic Vector Quantization

Given input data $I \subseteq \Re^n$, a vector quantizer finds and optimizes an *encoder mapping* $\alpha_{\mathcal{C}} : \Re^n \to \Im$ and a *decoder mapping* $\beta_{\mathcal{C}} : \Im \to \Re'^n$. Here, $\Re$ is the input alphabet and $\Re'$ is the output alphabet of the quantizer. These alphabets need not to be identical, though it is appropriate for most applications (and this thesis) to assume that they are. $\Im$ specifies some index set, usually a subset of $\mathbb{N}$, and $\mathcal{C}$ a codebook that may either be generated during the computation of the encoder or may be known a priori.

Algorithm 1 Linde-Buzo-Gray algorithm

Input:

Initial codebook $\mathcal{C} = \left\{\boldsymbol{y_i^{(0)}}\right\}_{i=1}^{2^r} \subset \Re^n$, where r is the limit bit rate
Set of input vectors $I \subset \Re^n$
Threshold ε
Maximum number of iterations k_{max}

Output:

Codebook $\mathcal{C} = \left\{\boldsymbol{y_i^{(k)}}\right\}_{i=1}^{2^r}$
Partition $I = \dot{\bigcup}_{i=1}^{2^r} \mathcal{V}_i^{(k)}$

// Initialization:
$k \Leftarrow 0$
$d^{(0)} \Leftarrow 0$
repeat
// Update quantization regions:
$\mathcal{V}_i^{(k)} \Leftarrow \{\boldsymbol{x} \in I : \delta(\boldsymbol{x}, \boldsymbol{y_i}) < \delta(\boldsymbol{x}, \boldsymbol{y_j})\ \forall i \neq j\}$, where $j = 1, ..., 2^r$.
// Compute distortion:
$d^{(k)} \Leftarrow \sum_{i=1}^{2^r} \sum_{\boldsymbol{x} \in \mathcal{V}_i^{(k)}} \delta\left(\boldsymbol{x}, \boldsymbol{y_i}^{(k)}\right)$.
$k \Leftarrow k + 1$
// Update Codebook:
$\boldsymbol{y_i^{(k)}} \Leftarrow \frac{1}{\left|\mathcal{V}_i^{(k-1)}\right|} \sum_{\boldsymbol{x} \in \mathcal{V}_i^{(k-1)}} \boldsymbol{x}$
until $\frac{d^{(k-1)} - d^{(k)}}{d^{(k)}} < \varepsilon$ **or** $k > k_{max}$
return Codebook $\mathcal{C} = \left\{\boldsymbol{y_i^{(k)}}\right\}_{i=1}^{2^r}$ **and** Partition $I = \dot{\bigcup}_{i=1}^{2^r} \mathcal{V}_i^{(k)}$

The encoder maps an n-dimensional vector to a single index from $\Im$, wheras the decoder reverses this process to some extent, since vector quantization is typically lossy. If no codebook is known a priori, vector quantization can be seen as a non-linear optimization process trying to find the codebook $\mathcal{C}$ and the encoder mapping α that minimize the quantization error $\delta(I, \alpha_{\mathcal{C}} \circ \beta_{\mathcal{C}}(I))$. Typically, the hard constraint that $\mathcal{C}$ may contain at most 2^r pair-wise different entries from $\Re^n$ is additionally required, where r

is the limit bitrate of the encoder as $|I|$ approaches infinity. Here, $\delta : \Re^n \times \Re^n \rightarrow \mathbb{R}$ computes the reconstruction error of the process.

Once an initial codebook is obtained, it can be further refined using the Linde-Buzo-Gray (also known as Generalized Lloyd) algorithm [LBG80], see Algorithm 1.

The encoder mapping $\alpha_{\mathcal{C}}$ then maps each vector $\boldsymbol{x} \in I$ to the index $i : \boldsymbol{x} \in \mathcal{V}_i^{(k)}$ of the associated quantization cell. The decoder mapping $\beta_{\mathcal{C}}$ maps each index i to the codebook entry $\boldsymbol{y_i^{(k)}}$. A random codebook may serve as input to the Linde-Buzo-Gray algorithm, although this is not free of problems. The reason is that quantization cells can become deserted during runtime. A better choice is to start with a codebook that is obtained using some sort of quick pre-clustering, e.g., principal component analysis, uniform binning, etc. For all the work in this thesis, the vector quantizer described in the author's diploma thesis [Sch03] was used. This particular implementation obtains an initial codebook by means of recursive principal component analysis (see also Section 2.6).

2.8 Gauss- and Laplace-Pyramids

In this work, we will denote discrete filter kernels by $\kappa : \mathbb{Z} \rightarrow \mathbb{R}$. Unless noted otherwise, κ will be intrinsically one-dimensional. We define the effective radius from left ρ_l and from right ρ_r of κ to be the maximum distance of any non-zero filter coefficient from the DC position $i = 0$, i.e.,

$$\begin{aligned} \rho_l(\kappa) &:= \min_i \{i : \kappa(i) \neq 0\} \\ \rho_r(\kappa) &:= \max_i \{i : \kappa(i) \neq 0\}. \end{aligned} \tag{2.16}$$

Here, $\kappa(i)$ denotes the ith filter coefficient. Given a discrete signal on a regular grid, $\Phi : \mathbb{Z} \rightarrow \mathbb{R}^n$ the discrete convolution $\kappa \star \Phi$ in the spatial domain can thus be written as

$$\kappa \star \Phi(i) := \sum_{i'=\rho_l(\kappa)}^{\rho_r(\kappa)} \kappa(i')\Phi(i+i'). \tag{2.17}$$

Clearly, the evaluation of $\kappa \star \Phi(i)$ requires $\rho_r(\kappa) - \rho_l(\kappa) + 1$ multiplications and $\rho_r(\kappa) - \rho_l(\kappa)$ additions for each position i.

Both the Gauss- and the Laplace-pyramids of Φ are based on two discrete filtering

operations

$$\operatorname{reduce}_m(\kappa,\Phi)(i) \quad := \quad \sum_{i'=\rho_l(\kappa)}^{\rho_r(\kappa)} \kappa(i')\Phi(mi+i'), \tag{2.18}$$

and

$$\operatorname{expand}_m(\kappa',\Phi')(i) \quad := \quad \sum_{i'=\rho_l(\kappa')}^{\rho_r(\kappa')} \kappa'(i')\Phi'\left(\left\lfloor\frac{i}{m}\right\rfloor + i'\right). \tag{2.19}$$

Basically, reduce_m convolves Φ first with κ and then with a comb-filter with a sampling distance of m. The result is a discrete signal Φ' that is obtained from Φ by performing an $m : 1$ subsampling. On the other hand, expand_m performs an $1 : m$ supersampling by convolving with κ'. The division by m ensures that the position i of $\operatorname{expand}_m(\kappa', \operatorname{reduce}_m(\kappa,\Phi))$ will coincide with those of Φ. Typically, κ will be a lowpass-filter, e.g., a binomial filter, and κ' will be an interpolation filter. Thus, expand_m may be seen as a pseudo-inverse of reduce_m.

Repeated application of reduce_m will yield the Gauss-pyramid

$$G_m(\Phi) \quad := \quad \left\{\operatorname{reduce}_m^i(\kappa,\Phi)\right\}_{i=0}^{n-1}, \tag{2.20}$$

where n is the amount of levels in the pyramid. For convenience, we will denote the ith level of G_m by $G_m^{(i)}$. Then, the residual of a reduce/expand "roundtrip" can be computed as $\Delta_m^{(i)} := G_m^{(i)} - \operatorname{expand}_m\left(\kappa', G_m^{(i+1)}\right)$. These residuals together with the coarsest level of the Gauss-pyramid will then form the Laplace-pyramid,

$$L_m(\Phi) \quad := \quad \left\{\Delta_m^{(i)}\right\}_{i=0}^{n-2} \cup \left\{G_m^{(n-1)}\right\}. \tag{2.21}$$

The full pyramid of a signal with N samples has $\lceil \log_m N \rceil$ levels and each level except the coarsest one contains a factor of m less samples than the next finer one. Consequently, the pyramid can be constructed in $\mathcal{O}(m \cdot N \log_m N)$. The reconstruction of data given by a Laplace pyramid then proceeds by evaluating

$$\Phi \quad = \quad \operatorname{expand}_m\left(\kappa', G_m^{(n-1)}\right) + \sum_{i=0}^{n-2} \operatorname{expand}_m\left(\kappa', \Delta_m^{(i)}\right). \tag{2.22}$$

Since Δ is always computed to be the true residual between the original and its down-

sampled counterpart, the decomposition into a Laplace-pyramid is (up to floating point precision) lossless and fully invertible. However, similar in style to overcomplete wavelet bases, an additional storage requirement of a factor of

$$\sum_{i=0}^{\infty} \frac{1}{m^i} = \frac{m}{m-1} \; \forall \, m > 1 \tag{2.23}$$

for an 1D pyramid is implied.

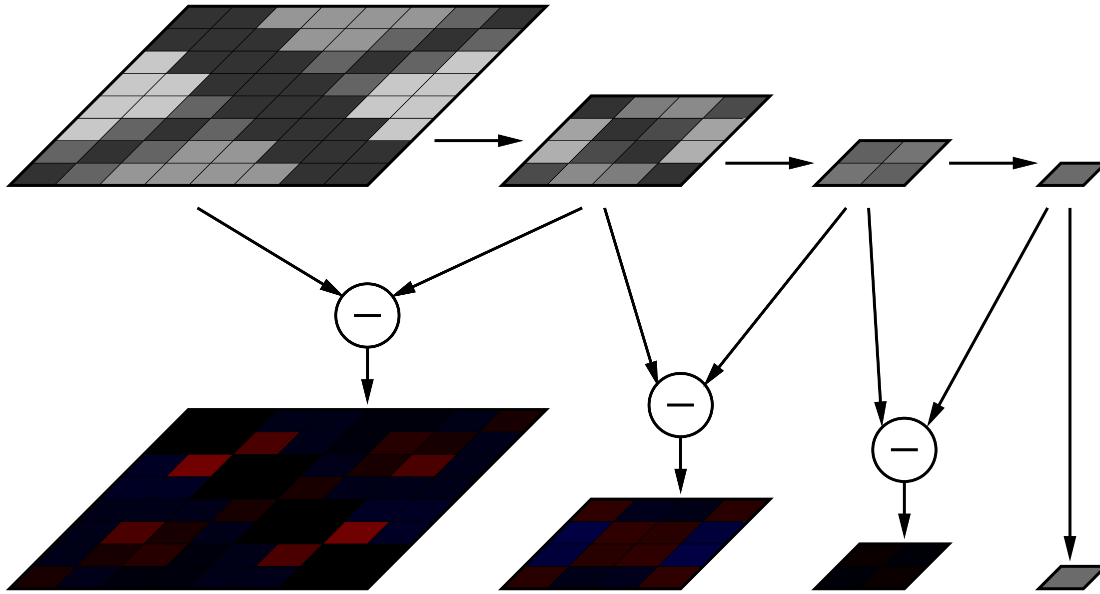


Figure 2.11: *Gauss- and Laplace-pyramids in 2D. The Gauss-pyramid (top) is obtained by iterative filtering using the* reduce_2 *operator with a box-filter. The Laplace-pyramid (bottom) is obtained by computing residuals* $\Delta_m^{(i)}$ *using a piecewise constant filter kernel for the* expand_2 *operation. Here, blue encodes positive and red negative values. (from [Sch03])*

The concept of Gauss- and Laplace-pyramids can be generalized to higher dimensions straightforwardly by following the tensor-product approach, as demonstrated in Figure 2.11. In case of a d-dimensional tensor-product pyramid, the storage requirements compared to a single image increase by a factor of

$$\sum_{i=0}^{\infty} \left(\frac{1}{m^d}\right)^i = \frac{m^d}{m^d - 1}. \tag{2.24}$$

The importance of Laplace-pyramids for image coding stems from the fact that for most natural images the amplitude decreases with frequency—and thus also the entropy

decreases with frequency. By using a lowpass filter for κ the average energy per pixel decreases from the coarsest to the finest level. It is thus advantageous to assign lower bitrates (in terms of bits per pixel) to levels with a higher resolution, thereby achieving image compression.

Chapter 3

Rendering Of Massive Point Scans

In this chapter, we present a method to compress and render gigantic point scans. These point scans are typically obtained by laser range scanning (see also Section 2.3), but the method is not limited to data stemming from this acquisition method. The input is a 3D point cloud, optionally with normal and/or color information. The method described here first resamples the point cloud to a hexagonal closest sphere packing (HCP) grid. The resulting regular grid is then sliced orthogonally to the major axis of the scanned object. For most sources of data, i.e., data stemming from laser range scanning or computer tomography (also see Section 2.3 and Section 2.5), such a major axis is intrinsically given. If it is not known a priori, the user can either specify it using additional knowledge about the acquisition process, or the axis can be retrieved automatically using a principal component analysis (see also Section 2.6).

Each such slice then comprises a two-dimensional, regular hexagonal grid. A planar graph is obtained in which vertices correspond to occupied cells and edges are induced by the adjacency relation defined on the grid cells. Occupied cells are then encoded using a coherent traversal of this graph. To find a quasi-optimal traversal, a graph-theoretical problem is established and solved using a linear-time 2-approximation. We would like to refer to the excellent book by Hromkovič [Hro01] for a thorough discussion of approximative algorithms. Once the coherent traversal path is obtained, occupied cells along this path can be encoded differentially with as few as $\log_2 5$ bits per point position in the limit case. Normals can be quantized at high fidelity using an additional 5 bits per point. Colors typically require 5 to 8 bits per point.

The resulting compressed representation can be decoded on the fly during rendering, thereby reducing both video and main memory storage requirements. Furthermore, bandwidth requirements across the graphics bus are significantly reduced. Especially the latter property is extremely important, since—although main and video memory

have dramatically increased in size during the last decades—the bandwidth between CPU and GPU has not scaled at the same pace.

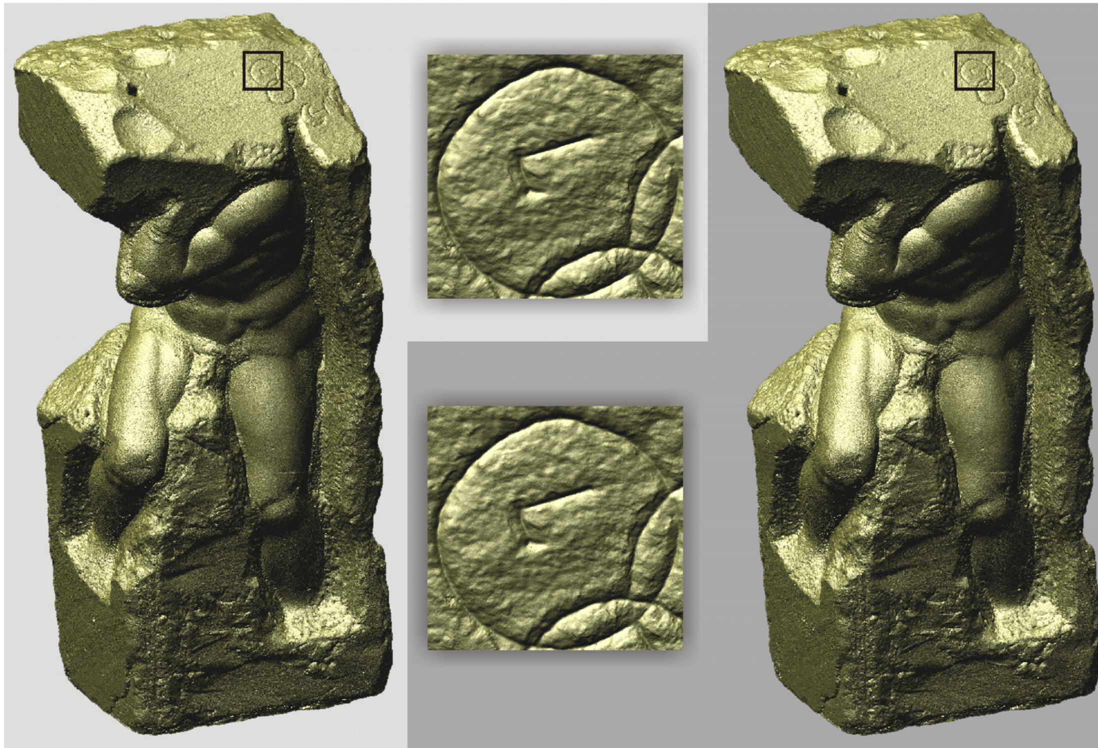


Figure 3.1: *Comparison between uncompressed and compressed Atlas point scan. The original Atlas point scan [LPC$^+$00] including normals (6 GB) was compressed by our method (231 MB). Note how the fine scale detail is preserved. Data set courtesy of the Digital Michelangelo Project.*

Although only about 8 bits are used per point to encode both position and normal of the Atlas point scan [LPC$^+$00], the method described here is almost lossless in terms of visual fidelity (see also Figure 3.1). The data set has a convenient size of 231 MB after compression, fits into video memory on most recent GPUs, and can be rendered at interactive rates *without* the need to establish a consistent triangulation first.

3.1 Related Work

In this section we will provide an overview of related work in the field of point-based rendering as well as point set compression. We begin with a very compact discussion of closest sphere packings. While a full treatment of these packings is well beyond the

scope of this thesis, we chose to provide the reader with the basic aspects in an attempt to make this thesis self-contained.

Closest Sphere Packings. Already Carl Friedrich Gauss proved that the hexagonal ("honeycomb") grid is the densest possible *regular* packing of circles in 2D with a *packing density* (ratio of area covered by circles to total area) of

$$\eta = \frac{\pi}{\sqrt{12}}. \tag{3.1}$$

The general proof that this is the densest possible packing among all possible grids—regular and irregular—was performed only in 1940 by László Fejes Tóth [Weia]. For three dimensions, Johannes Kepler conjectured in 1611 that the maximum packing density for both regular and irregular packings were

$$\eta = \frac{\pi}{\sqrt{18}}. \tag{3.2}$$

Following a concept by Tóth developed in 1953, Thomas Callister Hales formulated an exhaustive proof in 1998 in a series of papers totalling about 250 pages. This proof involves global optimization, linear programming, and interval arithmetic [Weib]. It was attested a 99% probability for correctness by its reviewers, since not all parts of the papers had been possible to verify.

Generally, closest sphere packings are an optimal packing of 3D space, i.e., the ratio of the volume occupied by the spheres' interiors and the total volume is maximal. Once the Kepler conjecture is established, the proof that both the face-centered cubic (FCC) and the hexagonal close packed (HCP) grids are in fact closest sphere packings is remarkably easy—it is only necessary to show that their packing density equal the one established in Equation (3.2).

Since their packing density is maximal, we know from closest sphere packing theory [CSB87] that an optimal sampling in the spatial domain corresponds to the tightest arrangements of spheres in the frequency domain. This can be derived from the observation that the spectrum of the sampled signal contains the replicas of the primary spectrum, centered at the points of the dual (or reciprocal) of the sampling grid. Optimal sampling of the signal is achieved if the overlap between these replicae is minimized. Thus, closest sphere packings can be used for optimal resampling if a spherically bandwidth-limited reconstruction kernel is to used. Another way to see this is that among all possible grids closest sphere packings will result in the fewest samples

needed to achieve any given reconstruction fidelity.

Compared to other grids, closest sphere packings not only lead to an optimal sampling density, but when compared to Cartesian grids they also yield a significantly smaller maximum sampling error. For the same region of 3D space and the same amount of cells, in Cartesian grids the maximum distance between the center of each cell and the farthest point of that cell is $\frac{\sqrt{3}}{2}h$, where h is the cell spacing. For HCP grids, on the other hand, this distance is only $\frac{\sqrt{2}}{2}h$, which is about 22.5% less when compared with Cartesian grids.

Resampling to such grids usually involves finding the closest grid node for each query point. A natural solution to this problem is to generate the Voronoi diagram [Vor08] using the grid nodes as Voronoi sites. This gives rise to the *dual grid*, which is comprised of regular cells called the *Wigner-Seitz* cells in solid state physics [IL02]. Each query point is then associated with the cell of the dual grid in which it is contained. The HCP grids used in this chapter are composed of Wigner-Seitz cells that are trapezo-rhombic dodecahedra (TRDs). The TRD is the dual of the Johnson solid 27 [Joh66] and a space filling twelve (twelve = duodecim (latin)) sided polyhedron, which constitutes a base element for a closest sphere packing of 3D space. Naturally, TRD cells can touch each other only in faces or in vertices (see also Figure 3.2). In contrast to the HCP grid, the FCC grid is composed of Wigner-Seitz cells that are rhombic dodecahedra [Mat04, NM02].

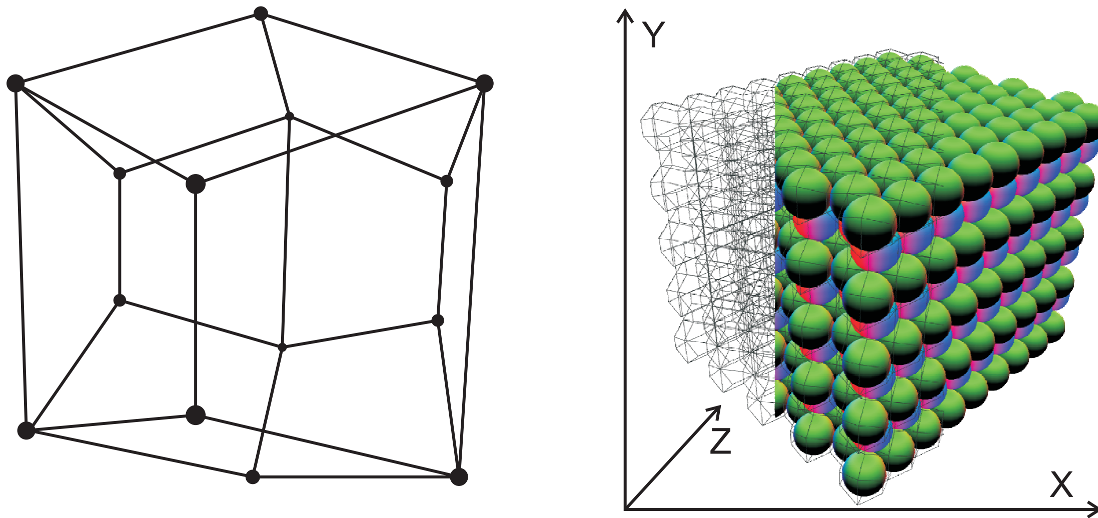


Figure 3.2: *Hexagonal closest sphere packing (HCP) grid. Left: A trapezo-rhombic dodecahedron (TRD). The dual of the HCP grid is a grid composed of TRDs. Right: The HCP grid. Around the spheres, the TRD cells of the dual grid can be seen.*

The important difference between the FCC and the HCP grid is the order of 2D "honeycomb" layers, which is an ABC-stacking for FCC and an AB-stacking for HCP grids. ABC means that each third layer is identical modulo translation along the packing direction, which we will assume to be the Y-direction (as depicted in Figure 3.2) for the remainder of this chapter. Other layers will be shifted against each other by a translation in the XZ-plane. Due to this AB-packing order the HCP grid can be decomposed into a staggered grid comprised of only four Cartesian grids, while the FCC decomposition requires six grids. This decomposition is explained in detail in Section 3.3.1. As will be shown, such a decomposition results in a highly efficient resampling algorithm that is 50% faster for HCP grids than for FCC grids.

Point-based Rendering. Despite the advances in CPU and graphics hardware technology, most point-based rendering applications still cannot run at acceptable frame rates for the largest available point scans. As rendering capabilities continue to increase, so do data acquisition and display technologies, resulting in a significant increase in resolution of both the available data and the displays being used. Today, laser range scans comprised of almost a billion of vertices are available [Lev00, LPC$^+$00], even making CPU processing difficult due to memory constraints. Figure 3.3 shows such gigantic scans, the largest of which consists of 250 millions of vertices and requires 6 GBytes to store positional and normal information. Because of the extraordinary amount of detail

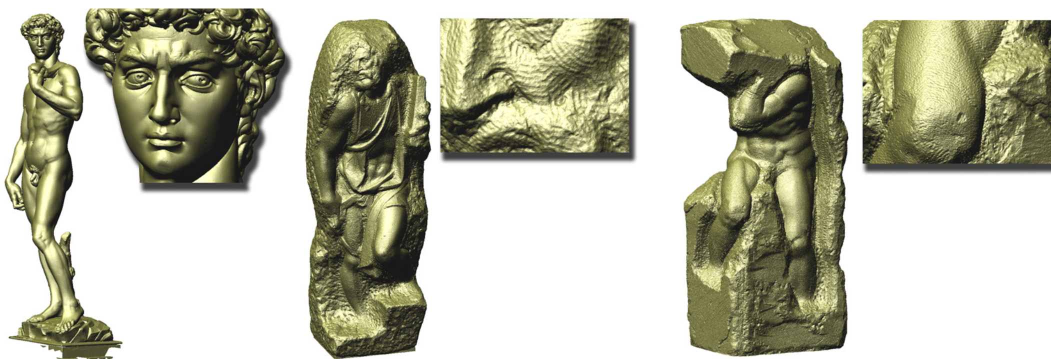


Figure 3.3: *The three largest scans provided by the Michelangelo project. All three scans, including per-point normals, have been compressed and now fit into video memory on virtually all recent graphics cards. Images are generated by rendering the scans directly from the compressed data stream on the GPU. Up to 50 million points per second can be decoded* and *rendered on a ATi X800 XT graphics card. Data sets courtesy of the Digital Michelangelo Project.*

in these scans, the need for techniques that are able to reveal even the finest structures

is of increasing importantance. In addition to such scans, high resolution displays of 9 Mpixels and more [IBM] are now available. The result is that the bandwidth required to transmit primitives to the GPU has substantially grown during the last decades. Since these requirements will increase continuously in the future, there is a dire need for point rendering techniques that address these issues comprehensively.

In computer graphics, point based rendering has recently gained increasing popularity due to the simple and memory friendly nature of points as rendering primitives. Such primitives do not require consistent topological information and they can reduce overdraw considerably if high resolution models are to be rendered. A thorough discussion of these issues as well as a summary of recent point rendering techniques, including various applications, can be found in [GPA$^+$, KB04].

Using points as the only rendering primitive was first considered by Levoy and Whitted [LW85] and later revived by Grossmann and Dally [GD98]. Point-based rendering systems have been proposed for the hierarchical rendering of large models [RL00] as well as for the high-quality rendering of point sampled geometry [PZvBG00, ZPvBG01]. Due to the frequent use of such systems in practical applications, there has been an ongoing improvement in this field during the last few years, both with respect to rendering speed and quality. This improvement has been made possible by exploiting graphics hardware through efficient GPU data structures [RPZ02, DVS03], by using high-quality point splats [BK03, ZRB$^+$04], and through the use of point hierarchies [GM04] to allow efficient LOD rendering.

Besides rendering quality and speed, today's point rendering systems are facing the problem of continually increasing point sets. To keep up with this process, several issues have to be considered. For large point scans the CPU might not be equipped with sufficient system memory. If the CPU works on a compressed data set, it might not be powerful enough to decode point positions and attributes at sufficient rates. On the other hand, if a streaming representation is available that enables out-of-core rendering, disk access will potentially limit the overall performance. But even with the recent advances in hardware technology—i.e., multicore architectures, RAIDs[1], and a dramatic increase in typical host memory sizes—the most severe limitation to all CPU-based approaches still remains. Even if the CPU provides point rendering primtives at sufficient rates, they have to transmitted to the GPU across the graphics bus. Despite the PCI express $16\times$ delivering a theoretical 4GB/sec bandwidth, practical bandwidths differ significantly, and latencies cannot always be hidden. Consequently, the net bandwidth between CPU and GPU easily becomes the bottle neck when attempting to transfer

[1]Redundant Array of Inexpensive Disks

point positions and attributes while maintaining 30 or more frames per second. Furthermore, since raw data occupies large amounts of video memory and transmitting data from the CPU to the GPU essentially serializes communication and rendering, the GPU itself might not be able to render the points within the requested time interval.

Point Set Compression. A natural way to deal with the aforementioned issues is point set compression. The benefit of such a compression is two-fold. Firstly, storage requirements are significantly reduced. Secondly, bandwidth requirements are alleviated. However, these benefits can only be maintained up to the point in the pipeline where the data has to be decompressed in order to be rendered. Consequently, this decoding process should take place as close as possible to the rendering. Since decoding usually requires additional bandwidth in excess of the one needed to render uncompressed data, it is highly fortunate that the GPU's internal memory bandwidths by now offer peak performances beyond 100GB/sec and are easily among the highest bandwidths in the entire system [2]. Still, decoding should provide efficient random access to the primitives to be rendered. This is important with respect to both the bandwidths available as well as the fact that different camera settings may require significantly different parts of the data to be decoded.

Although the compression of points has been proposed before, most approaches require significant CPU intervention during the decoding process due to their complexity.

A popular technique is to quantize point positions with respect to a Cartesian grid hierarchy, either by absolute position or by an offset to a parent node in this hierarchy [RL00, SK01, BWK02]. Although these approaches can significantly reduce the required number of bits to encode point positions—less than 2 bits have been achieved per position—, a similar compression ratio has not been shown for normals and colors yet. Ochotta and Saupe [OS04] parameterize point sets locally as height fields and resample the point sampled surface on a regular grid. This method achieves high compression ratios by using wavelet transforms to encode the resulting height fields, but it introduces smoothing artifacts and produces a non-uniform sampling of the surface. A progressive compression scheme for point sets including per-point attributes based on multiresolution predictive encoding was presented by Waschbüsch et al. [WWL$^+$04]. This scheme yields effective compression rates, but it suffers from both the $\mathcal{O}(N^2 \log N)$ runtime complexity of the matching process to detect similarities in the set of N points and the rather costly decoding process. This decoding requires recursive traversal of binary trees to calculate initial point positions, an operation that is not overly GPU-friendly

[2] For instance, NVIDIA specifies the GeForce 280 GTX at 141.7GB/sec.

due to the high amount of dereferenciations performed.

3.2 Contribution

In this chapter we present a system for the rendering of gigantic point scans that meets the aforementioned requirements. Namely, the system is based on a lossy compression scheme for point positions and normals, which reduces both storage and bandwidth requirements. Furthermore, compared to previous compression schemes for point sets, point coordinates can be decoded on the GPU. This results in the ability to both store extremely large point sets in video RAM and to render them directly from their compressed representation at interactive rates. Our scheme has the following advantageous properties.

- **Memory efficiency**: We present an effective compression scheme for large point scans based on an optimal sampling of these scans.
- **Decoding efficiency**: The compressed stream provides random access to encoded points and attributes, and it can be decoded using a few simple arithmetic and logical operations.
- **Bandwidth efficiency**: Due to its simplicity, decoding can be performed on the GPU. To render the point set, only the compressed data stream has to be transmitted.
- **Rendering efficiency**: On the GPU, decoded point positions and normals are used to render the point scan, which results in a significant performance gain compared to previous approaches.

To achieve these properties, we perform point clustering on HCP grids as detailed in the next section.

3.3 Algorithmic Overview

For each point x of the input set $\mathcal{X}$, the enclosing TRD cell of the HCP grid is determined. Each TRD cell is then assigned a binary tag, which can be set either to *occupied* or *empty*. A cell is said to be occupied if at least one of the input points is contained in this cell. Otherwise, the cell is said to be empty. The grid is then sliced perpendicular to the Y-axis (the axis along which layer stacking occurs). For each 2D hexagonal

"honeycomb" grid obtained in this manner, an undirected, planar graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is constructed, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Each occupied cell is interpreted as a vertex and edges are induced by the grid's adjacency, i.e.,

$$\begin{aligned} \mathcal{V} &:= \{\boldsymbol{v_i} \,:\, \exists\, \boldsymbol{x} \in \mathcal{X} \,:\, \boldsymbol{x} \in \text{cell } i\} \\ \mathcal{E} &:= \{\{\boldsymbol{v_i}, \boldsymbol{v_j}\} \,:\, \boldsymbol{v_i}, \boldsymbol{v_j} \in V \,\wedge\, \text{cell } i \text{ adjacent to cell } j\}\,. \end{aligned} \tag{3.3}$$

A path $p := (\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n})$ is then constructed for each connected component of $\mathcal{G}$, such that p contains each vertex of $\mathcal{V}$ at least once. Note that such a path may explicitly contain arbitrarily short circles. Since there is an infinite amount of such paths, we seek to obtain the shortest one. Denoting the set of all possible paths by $\mathcal{P}$ and the *length* of a path by $|p|$, i.e., the number of vertices visited including multiplicities, we thus seek to obtain

$$p' \in \mathcal{P} \quad : \quad |p'| \leq |q| \; \forall q \in \mathcal{P}. \tag{3.4}$$

Sadly, this problem is assumed to be NP-hard. However, we present a linear-time 2-approximation to this problem, i.e., an algorithm with linear time and space complexity that solves the problem of finding

$$p' \in \mathcal{P} \quad : \quad |p'| \leq 2|q| \; \forall q \in \mathcal{P}. \tag{3.5}$$

Once one p' is obtained, we "cut" p' in a set of n shorter paths, $\mathcal{S} := \{s_i\}_{i=1}^{n}$. We say that a path p *contains* a path q, $q \lhd p$, with $p = \left(\boldsymbol{v_1}, \ldots, \boldsymbol{v_{|p|}}\right)$ and $q = \left(\boldsymbol{v'_1}, \ldots, \boldsymbol{v'_{|q|}}\right)$ if

$$\exists i \in \mathbb{N} \quad : \quad \boldsymbol{v'_j} = \boldsymbol{v_{i+j}} \; \forall j = 1, \ldots, |q|. \tag{3.6}$$

Furthermore, we say that a path q can be *concatenated* to a path p, $p \circ q$, if $\left\{\boldsymbol{v_{|p|}}, \boldsymbol{v'_1}\right\} \in \mathcal{E} \vee \boldsymbol{v_{|p|}} = \boldsymbol{v'_1}$. The concatenated path is then

$$p \circ q \quad := \quad \begin{cases} \left(\boldsymbol{v_1}, \ldots, \boldsymbol{v_{|p|}}, \boldsymbol{v'_2}, \ldots, \boldsymbol{v'_{|q|}}\right) & \text{if } \boldsymbol{v_{|p|}} = \boldsymbol{v'_1} \\ \left(\boldsymbol{v_1}, \ldots, \boldsymbol{v_{|p|}}, \boldsymbol{v'_1}, \ldots, \boldsymbol{v'_{|q|}}\right) & \text{otherwise.} \end{cases} \tag{3.7}$$

Note that in the first case $|p \circ q| = |p| + |q| - 1$, since the vertex required to be contained by both p and q is not doubled. Only in the second case $|p \circ q| = |p| + |q|$. Albeit the lengths of paths constructed by concatenation can differ, we prefer this definition over other possibilities, since it makes the following discussion easier. Formally, cutting p'

into a set of shorter paths $\mathcal{S} = \{s_i\}_{i=0}^{n}$ can thus be described as

$$\begin{aligned} s &\lhd p' \; \forall \; s \in \mathcal{S}, \text{ and} \\ p' &= s_0 \circ s_1 \cdots \circ s_n. \end{aligned} \tag{3.8}$$

This means that the set of shorter paths S effectively partitions p'. We further require that $|s_i| = \sigma \; \forall i = 1, \cdots, n-1$ and that $|s_n| \leq \sigma$. This automatically implies that $n = \lfloor |p'|/\sigma \rfloor$. Such a partition is obtained straightforwardly and serves the purpose of making the processing of p' SIMD-tractable. However, these requirements are later weakened in order to optimize the set $\mathcal{S}$ further.

After obtaining the set of short paths $\mathcal{S}$, each such short path s is encoded separately. We store the first element of each such path in 2×16 bits to specify the xz-coordinates of the first occupied cell. The rest of each s is encoded differentially by assigning a number between 0 and 4 to each vertex, depending on the relative neighborhood with respect to the prior occupied cell. This results in a total of $32 + \sigma \log_2 5$ bits per short path. This encoding is very similar to the process described by Mroz et al. [MHG01] for the encoding of isosurfaces in volumetric data sets. Unlike Morz et al., however, we provide a graph-theoretical formulation of the problem to find an optimal coverage of all occupied cells, i.e., a path that is minimal in terms of the vertices and edges visited.

Normals are encoded in a similar manner. The first normal of each short path is encoded in 16 bits using vector quantization (see also Section 2.7). Each other normal is encoded differentially using spherical coordinates. These spherical differences are again vector quantized. Our experiments have shown that 5 bits per difference are sufficient to encode normals along each short path. This results in another $16 + 5\sigma$ bits per short path.

Colors can be encoded completely analogously to normals. However, unlike normals, colors are not necessarily highly coherent along each short path. Consequently, higher bit rates are sometimes necessary. For the first color of each short path 8 to 10 bits are usually sufficient, resulting in a total of less than $10 + \rho\sigma$ bits, where ρ is the amount of bits per color difference.

To support view frustum and back face culling, path-specific attributes such as cones of normals and bounding boxes are computed.

The compressed point set can be decoded on the CPU, and point primitives can be sent to the GPU for rendering. Alternatively, the compressed stream can be decoded on the GPU. To exploit the GPUs parallelism efficiently, short paths of equal length are stored in 2D texture maps and are decoded incrementally. Decoded point positions are

first stored in a temporary buffer in graphics memory before they are used by the GPU to render point primitives. This is realized using recent functionality like vertex texture fetches.

3.3.1 Uniform Point Clustering and Resampling

The resampling step requires to find the associated cell for each point $\boldsymbol{x}$ of the input set $\mathcal{X}$. This can be done by "sorting" $\boldsymbol{x}$ into four staggered Cartesian grids to compute four potential candidates for the appropriate cell. From these four candidates the correct cell can be chosen simply by selecting the candidate that results in the minimum distance between the center of the cell and $\boldsymbol{x}$.

To perform the resampling, we assign an index $(i, j, k) \subseteq \mathbb{Z}^3$ to each TRD cell. Considering the cell geometry, the 2D case of which is depicted in Figure 3.4, the center $\boldsymbol{c_{ijk}}$ of each cell can be computed from its index by

$$\boldsymbol{c_{ijk}} = \begin{pmatrix} 2i \cdot r + (j \bmod 2) \cdot r + (k \bmod 2) \cdot r \\ \frac{2\sqrt{6}}{3} j \cdot r \\ \sqrt{3}k \cdot r - \frac{1}{\sqrt{3}}(j \bmod 2) \cdot r \end{pmatrix}, \tag{3.9}$$

where r is the radius of each sphere.

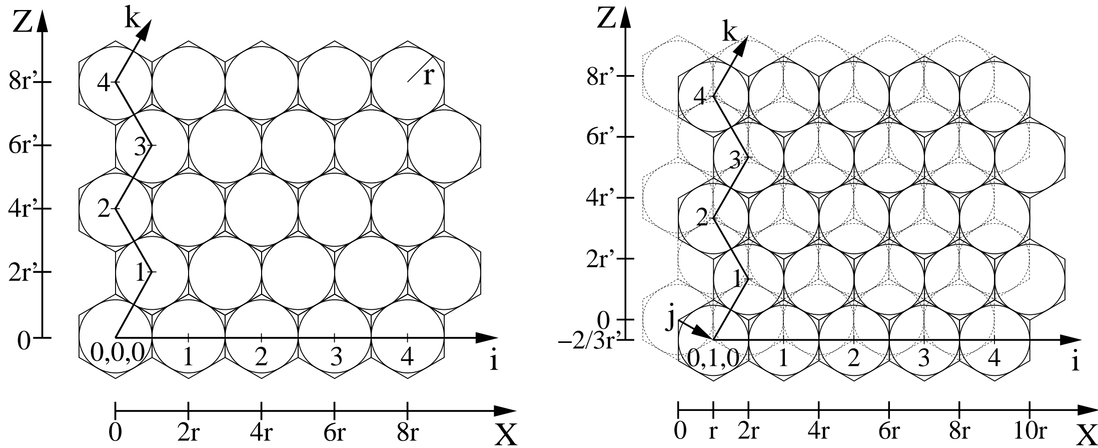


Figure 3.4: *Two layers of an HCP grid. Left: First layer of type A. This layer occurs for cell indices (i, j, k), where j is even. Here, r denotes the radius of the spheres and $r' := \frac{\sqrt{3}}{2}r$. As can be seen, centers of spheres on even rows (k even) give rise to a 2D Cartesian grid, while centers of spheres on odd rows (k odd) give rise to a second 2D Cartesian grid. Right: Packing another layer of type B (j odd) atop the first one only introduces a 3D pitch in Euclidean space. In 3D, four Cartesian grids are obtained by connecting centers of spheres belonging to (j even,k even), (j even,k odd), (j odd,k even), and (j odd,k odd).*

We now define four 3D Cartesian grids $\mathcal{C}_{0,\dots,3}$ such that $\mathcal{C}_l$ contains all points for which $l = 2(j \bmod 2) + (k \bmod 2)$. The size of each cell in these grids is $2r \times 2\sqrt{3}r \times \frac{4}{3}\sqrt{6}r$. The offsets for the grids are $\boldsymbol{c_{000}}$ for $\mathcal{C}_0$, $\boldsymbol{c_{001}}$ for $\mathcal{C}_1$, $\boldsymbol{c_{010}}$ for $\mathcal{C}_2$, and $\boldsymbol{c_{011}}$ for $\mathcal{C}_3$. The cell containing any given point $\boldsymbol{x}$ can then be determined for each of the four Cartesian grids simply by subtracting the respective grid's offset followed by a division by the grid's cell size and a floor operation. If $\boldsymbol{x}$ is inside a cell with Cartesian index i', j', k' for grid $\mathcal{C}_l$, then the corresponding TRD cell candidate with index i, j, k can be computed by

$$(i, j, k) = \begin{cases} (i', 2j', 2k' + 1) & \text{if } l = 0 \\ (i' + 1, 2j', 2k' + 2) & \text{if } l = 1 \\ (i', 2j' + 1, 2k' + 1) & \text{if } l = 2 \\ (i' + 1, 2j' + 1, 2k' + 2) & \text{if } l = 3 \end{cases} \tag{3.10}$$

Thus, a distance $\delta_l := \|\boldsymbol{x} - \boldsymbol{c_{ijk}}\|_2$ between $\boldsymbol{x}$ and the TRD cell's center $\boldsymbol{c_{ijk}}$ can be computed for each of the four grids $\mathcal{C}_l$. The correct TRD cell containing $\boldsymbol{x}$ is then found by selecting the cell C_l resulting in the smallest δ_l. Figure 3.5 illustrates this process, albeit in 2D for the sake of simplicity.

This resampling strategy is highly efficient, since it only requires the evaluation of four 3D-subtractions, 3D-divisions and 3D-floors, followed by four 3D-distance computations and one search for the minimum among four scalars for each point $\boldsymbol{x}$.

However, the optimal radius r is usually not known a priori. Clearly, we aim at a resampling for which as many of the occupied TRD cells as possible contain only one original point. On the other hand, by using ever smaller TRD cells, the graphs we construct will tend to have more connected components since occupied cells are less likely to be adjacent. This, in turn, makes the differential encoding scheme less efficient. Due to this reason, resampling is implemented as a two-step procedure that tries to optimize the resampling with respect to both constraints.

We start with an initial resolution r of the HCP grid. A good starting value for r can be obtained if the resolution of the acquisition device is known. If this resolution is not known, an arbitrary initial guess can be specified. This only comes at the potential cost of a longer optimization procedure—the initial choice of r affects the final result only marginally. To measure the quality of a certain resampling, we introduce a taxonomy we call the *hit rate* γ. This hit rate is defined as the ratio of the number of input points and the number of occupied TRD cells. The radius of the spheres in the HCP grid, and thus the size of the TRD cells, is then decreased iteratively until the hit rate drops below

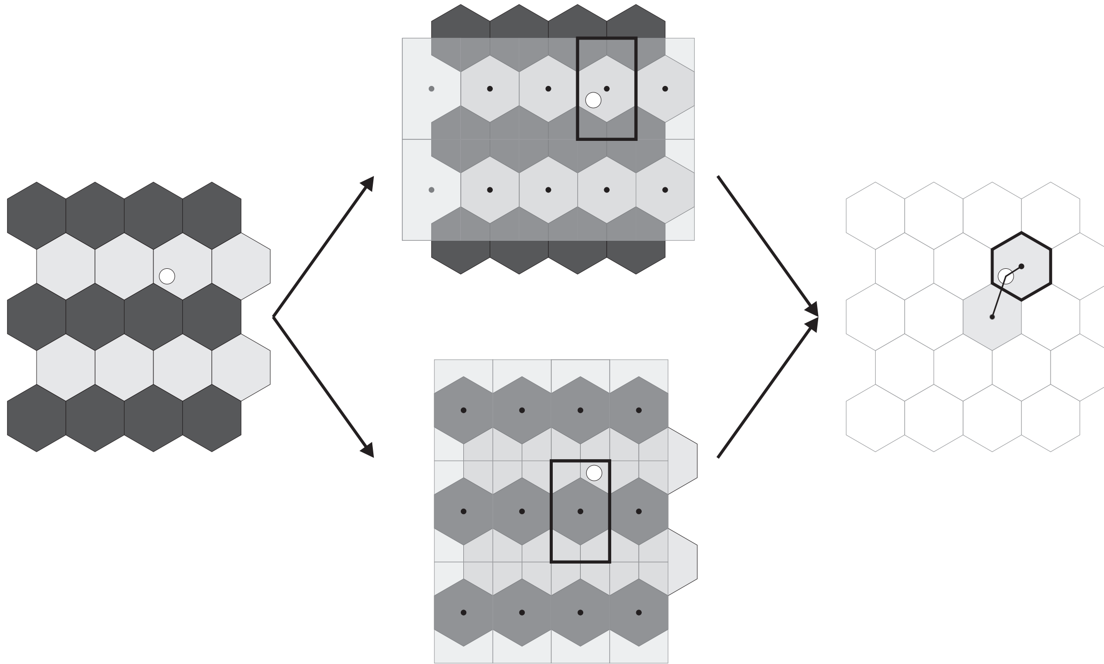


Figure 3.5: *Sampling of points into hexagonal grids. To determine the hexagonal cell containing a specific query point, the hexagonal grid is decomposed into a staggered grid comprising two Cartesian grids.*

a given threshold.

Since the point sets—and thus the associated grids—we are concerned with are very large and can usually not be stored in main memory, the entire sampling process is performed out-of-core. Point subsets are sequentially sampled into the grid, and they are then sorted on disk with respect to increasing cell index along the X-, Z-, and Y-axis. The sorted list can then be traversed sequentially both to determine duplicate samples in one cell and to compute the hit rate γ. At the end, the point set is implicitly given by the set of all occupied TRDs—or more precisely by the coordinates of their centers $\boldsymbol{c}_{ijk}$.

3.3.2 Path Generation

The HCP grid provides a structure to generate paths of occupied cells. This step is the transition from pure clustering to coherence based compression. The goal is to determine paths that are as long as possible and that contain as few redundancies as possible. The cells corresponding to vertices in these paths are then encoded differentially.

Path generation proceeds layer by layer, reading all occupied cells in the current

layer from disk. As discussed in Section 3.3, an undirected graph $\mathcal{G}$ is construced to reflect the occupied cells' topology. The actual process of finding an optimal path now operates on this graph, making the algorithm independent of the underlying grid structure. Ideally, the algorithm determines a set of short paths $\mathcal{S}$, where $|s| \leq \sigma \; \forall s \in \mathcal{S}$ such that $|\mathcal{S}|$ is minimal. This means that we seek to cover the vertices of path p' by shorter paths s such that no vertex is visited more than once. At the same time, these short paths should have a fixed length σ.

Since this problem is assumed to be NP-hard[3], we present a linear-time 2-approximation. The approximation ratio of 2 is a very conservative bound; in all meshes we processed so far the approximation ratio was typically less than 1.05 instead of 2. Note that although the length of the optimal path is not known, it can be conservatively estimated by the amount of vertices to be traversed.

For each connected component of $\mathcal{G}$, a single path p' is constructed as follows. Starting with an arbitrary node v in the graph, we mark v as being visited and store v in p'. If v has no unvisited neighbors, p' is terminated. If v has exactly one unvisited neighbor u, append u to p', mark u as visited and continue path construction with u. If v has more than one unvisited neighbor, recursively construct a path $\tilde{p}$ for each one of these neighbors. For each $\tilde{p}$ constructed in this manner except for the longest one, generate a *round-trip*. This requires to obtain the *reversed* path p^{-1} for a given path $p = \left(v_1, \ldots, v_{|p|}\right)$. This operation is naturally defined as

$$p^{-1} \quad := \quad \left(v_{|p|}, \ldots, v_1\right) . \tag{3.11}$$

A round-trip of p is then defined as $p \circ p^{-1}$ using the concatenation operator $\circ$. Note that according to Equation (3.7) $|p \circ p^{-1}| = 2|p| - 1$. These round-trips are then appended to p'. The longest path of the $\tilde{p}$'s is appended last, and the vertex visited last in this longest path is used to continue the algorithm.

The resulting path p' visits all vertices of the respective connected component at least once, and it traverses "branches" at vertices that have a higher valence than 2 forth and back—except for the longest branch. Figure 3.6 shows an example for such a path construction.

Once p' is obtained, it is cut into smaller paths of length σ in a greedy manner. Reduncancies are removed during this process, i.e., reverse paths occuring at the end or at the beginning of short paths. It is therefore necessary to keep track of which path segments are reversed and which are not. This process is demonstrated in Figure 3.7.

[3] Although we do not provide a proof we would like to refer to the striking similarity to the bin-packing problem

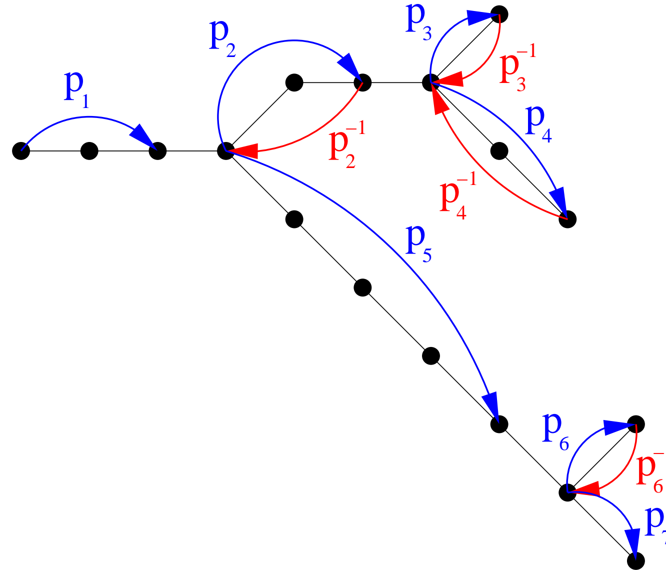


Figure 3.6: *Generation of a path visiting each vertex at least once. A recursive algorithm generates round-trips at vertices with a valence higher than 2, but the longest "branch" is not traversed back and forth. In this case, the path p' that contains each vertex at least once is* $p_1 \circ p_2 \circ p_3 \circ p_3^{-1} \circ p_4 \circ p_4^{-1} \circ p_2^{-1} \circ p_5 \circ p_6 \circ p_6^{-1} \circ p_7$

By decreasing the maximum path length σ, the number of paths generated increases, while at the same time the variation of the actual lengths is reduced. In this manner, σ is a very useful tuning parameter for GPU-based rendering, since GPUs owe their speed to lock-step SIMD computations. If multiple such SIMD units, i.e., fragment units, decode a number of short paths in parallel, it is desirable for each path to have exactly the same length. However, due to the per-path overheads in encoding the first vertex of each path, σ should not be chosen too small.

To see that our algorithm is really a 2-approximation, we consider the way paths are constructed. Since each "branch" returns to the respective forking vertex before any other branch is processed, and because sub-paths are constructed in a greedy fashion, it is sufficient to discuss vertices with valences less or equal to three only. All other cases reduce to this set of cases automatically due to the recursive nature of the algorithm. Figure 3.8 demonstrates that this is unaffected by the particular order in which neighbors are processed.

Thus, for each long path, a vertex cannot be visited more than three times, and only three times at T-junctions (vertices that are equivalent to valence-three vertices). However, for each vertex that is visited three times, there exists at least one vertex that is visited only once.

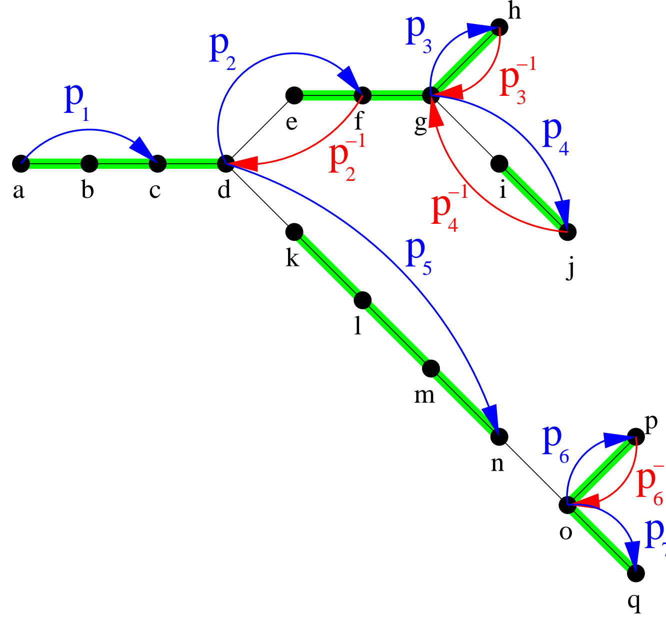


Figure 3.7: *Construction of short paths. A length of $\sigma = 4$ was used to cut the long path $p' = (a, b, c, d, e, f, g, h, g, i, j, i, g, f, e, d, k, l, m, n, o, p, o, q)$ with $|p'| = 24$ into short paths. The resulting short paths are (a, b, c, d), (e, f, g, h), (i, j), (k, l, m, n), and (o, p, o, q). Note that redundant parts are removed, resulting in a total of 18 vertices being visited instead of the initial 24.*

To generate a level-of-detail (LOD) hierarchy of the point set, sampling and path generation are repeated with decreasing resolution of the HCP grid. Starting with the optimal resolution, at each hierarchy level the resolution is reduced by a factor of two. Because the resolution at each coarser level is now fixed, grid size optimization as described in the next section has only to be performed for the finest level. Note that since each LOD contains only one eighth of the amount of cells of its predecessor, the storage requirements only increase by a factor of $\lim_{n\to\infty} \sum_{i=0}^{n-1} 8^{-i} = \frac{8}{7} < 1.143$.

3.4 Grid Optimization

To determine the optimal grid resolution for sampling and path generation, we consider the average length of short paths in addition to the hit rate. The hit rate measures how many points are lost due to the sampling process, while the average length is a measure of the compression efficiency. A perfect sampling would result in a hit rate of 1 and an average length of short paths equal to the desired length of short paths σ. Making the grid cells smaller results in a lower hit rate but reduces the average path length. To find the optimal cell size we first start with an initial radius of 1.5 times the average

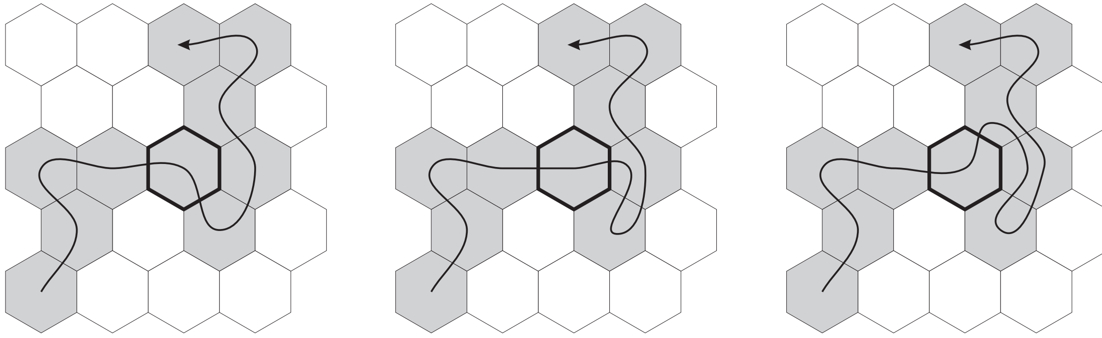


Figure 3.8: *A cell with four neighbors. Note that, independently of the choice of the first neighbor to be traversed, all neighbors are handled before the recursion returns to the initial node. For more than four neighbors, the method proceeds analogously. In consequence, a discussion of vertices with valence 3 (T-junctions) is sufficient, as all higher valences can be reduced to this case.*

pairwise closest distance of points in the input set $\mathcal{X}$. Since this task has a naïve runtime complexity of $\mathcal{O}(|\mathcal{X}|^2)$ (and an optimal complexity of $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$, it is worth noting that this average distance can be estimated quickly by taking only a compact subset of $\mathcal{X}$ of constant size into account, resulting in a $\mathcal{O}(1)$ complexity for this step. Since the data set is sorted with respect to its major axis before processing begins, this subset can be obtained simply by taking the first k points into consideration. Note that if the point density varies strongly in the model, a rather crude guess is obtained for the initial radius, resulting in more optimization passes. However, for the statues of the Digital Michelangelo Project, taking a few hundreds of thousands points as subset never resulted in a significant increase in the amount of optimization steps. If the hit rate is above a certain limit, usually $\gamma \geq 1.6$, the cell size is reduced according to the ratio between the maximally tolerated hit rate and the current hit rate.

The sampling process is repeated using the new grid resolution until the hit rate is below the maximum hit rate. Then, the path generation process is started. If the average length of short paths is below a given threshold, usually 0.7σ, we first try to close disconnected paths by inserting new cells. If this does not bring the average path length above 0.7σ, the grid cell size is increased, sampling is repeated, and new paths are generated. The process terminates if the average path length is acceptable. The output is then the last set of paths. Note that the factor 0.7 is empirically chosen based on our experience with different scans. Even for other starting values the algorithm will still find an optimal value but it will possibly take longer to converge.

In order to close disconnected paths, we search for filled cells in the 2-ring neigh-

borhood of those cells that are contained in paths shorter than 50% of the maximum length σ. We do not try to connect longer paths because this could result in paths that are so long that they are cut into short paths later. If such a cell is found and both cells belong to different paths, the cell in between is set to "occupied" in order to connect the two paths. These additional cells are also rendered. One could argue that this increases the approximation error and compromises the visual fidelity achieved. However, we would like to point out that the scanned data is typically noisy and might contain unintended holes. Consequently, inserting points at carefully selected cells could also be seen as a topological clean-up, under the assumption that topological features are originally larger than a few cells. To our experience, this process adds far less than 10% of the initial cells.

3.5 Encoding

The previous sections describe how to generate and optimize a large set of paths with lengths less or equal to a given length σ. In this section, we describe how these paths are encoded. The basic concept is identical for all point properties. The first vertex of each path is encoded using a relatively high bitrate, either simply by using uniform quantization (position), or by specifying an index into a rather large codebook obtained by vector quantization (normals and colors).

3.5.1 Point Encoding

The first position of each path is encoded using a 2×16 bits uniform hexagonal encoding, i.e., the 2D index of the starting cell is stored plainly for 2D honeycomb grids of up to $2^{16} \times 2^{16}$ cells. If larger grids are required, we mandate to use a partitioning or "bricking" strategy instead of further increasing the number of bits for the first vertex.

Each vertex but the first two are encoded relative to its predecessor using $\log_2 5$ bits in the limit case. The key observation here is that a path typically does not leave a cell through the same edge as it entered. The only exception to this rule are the "tips" of round-trips, and the second vertex, for which the entering edge of the predecessor is not defined. For the second cell, a specific edge is thus chosen as a reference. Potential successors for each following cell are enumerated clockwise, beginning with the "entering" neighbor (see also Figure 3.9). Thus, for the vast majority of vertices, a number of 0 to 4 is sufficient to encode them. However, in case that a path enters and leaves a cell through the same edge, a 5 is stored. While numbers $0, 1, 2, 3, 4$ are more or less

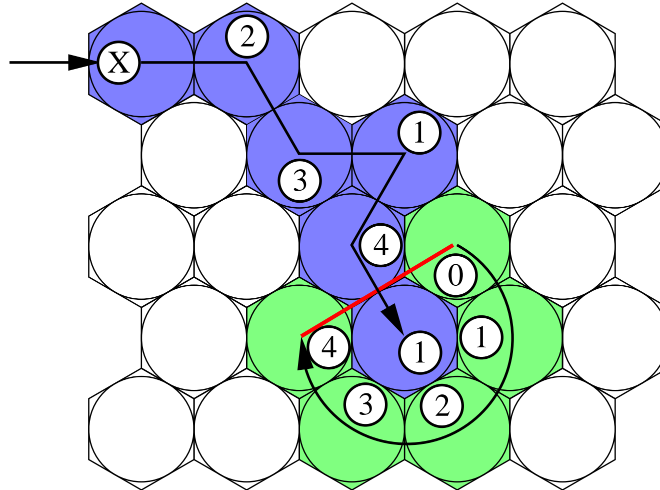


Figure 3.9: *Assignment of differential codes to paths. In this example, the path enters the cell marked with an X from the left. The previous cell from which the path enters the current cell is assigned the escape symbol "5" (not depicted). The remaining five cells are enumerated clockwise, beginning with the cell adjacent to the predecessor of the current cell. This is exemplified at the end of the depicted path. The edge through which the path entered is marked red, and the potential successors are colored in green. The path's code is thus* $\star 23141$*, where* $\star$ *represents a prefix that encodes the path preceeding and including cell X.*

equally likely to be stored in the stream, a 5 is highly unlikely to occur. Consequently, entropy coding of the stream approaches $\log_2 5$ bits per symbol. To see that a value of 5 is unlikely, consider that this only occurs at the tip of round-trips, and that round-trips are removed if redundancies arise.

However beneficial entropy coding of the code stream might be on the CPU, on the GPU this adds additional decoding overhead and seriously threatens overall performance due to the variable bit rates resulting from such codes. Consequently, we chose to allocate 3 bits to each differentially encoded vertex in order to improve rendering performance.

3.5.2 Normal Estimation and Encoding

Normal Estimation Normals are either given for the original point set, or they are computed prior to the compression stage, e.g., by computing normals on a given triangulation or by moving least squares [ABCO$^+$01]. Moving least squares first computes

a k-neighborhood for each input point $\boldsymbol{x} \in \mathcal{X}$, i.e.,

$$\begin{aligned} \mathcal{N}_k(\boldsymbol{x}) &:= \{\boldsymbol{n_i}\}_{i=1}^k \subseteq \mathcal{X}, \text{ such that} \\ \forall \boldsymbol{n} \in \mathcal{N}_k(\boldsymbol{x}) &: \|\boldsymbol{n} - \boldsymbol{x}\|_2 \leq \|\boldsymbol{y} - \boldsymbol{x}\|_2 \, \forall \boldsymbol{y} \in \mathcal{X} \backslash \mathcal{N}_k(x). \end{aligned} \tag{3.12}$$

This k-neighborhood can be computed efficiently using binary space partitioning libraries such as the Approximate Nearest Neighborhood (ANN) library [MA06]. Such libraries compute a binary space partitioning of $\mathcal{X}$ during their initialization. Subsequent queries are then answered in $\mathcal{O}(\log |\mathcal{X}|)$. Since our typical input point sets can be by far larger than the main memory, we sort point sets with respect to their Y-coordinate using an out-of-core sorting algorithm [Sed98]. We can then stream-process the sorted point set by computing the binary space partition for overlapping chunks of input points. The requirement that the k-nearest neighbors to each query point $\boldsymbol{x}$ are within a chunk is met by querying only points in the "middle" (with respect to the sorting along the Y-coordinate) of each chunk.

Once the k-neighborhood is obtained, a weighted covariance matrix $\mathcal{A}$ is computed, i.e.,

$$\mathcal{A}(\boldsymbol{x}) = \sum_{i=1}^{k} g(|\boldsymbol{n_i} - \boldsymbol{x}|) \, (\boldsymbol{n_i} - \boldsymbol{x}) \, (\boldsymbol{n_i} - \boldsymbol{x})^T \text{, where } \boldsymbol{n_i} \in \mathcal{N}_k(\boldsymbol{x}). \tag{3.13}$$

Here, $g(d)$ is a function that decreases each neighbor's influence with increasing distance d. If $\mathcal{A}(\boldsymbol{x})$ is not singular, the eigenvector $\boldsymbol{e_{\mathrm{min}}}$ corresponding to the smallest eigenvalue λ_{min} is the best possible estimate for the normal at $\boldsymbol{x}$. Since $\mathcal{A}(\boldsymbol{x})$ is symmetric and real, all of its Eigenvalues are real. Furthermore, λ_{min} is typically well separated from the other two eigenvalues—a fact that stems from the simple geometric observation that $\mathcal{X}$ consists of points on a surface. Consequently, both power and Rayleigh quotient iteration [GVL96a, PTVF02a] on $\mathcal{A}^{-1}(x)$ will converge quickly. If, however $\mathcal{A}(x)$ is singular, $\mathcal{N}_k(\boldsymbol{x})$ is either planar (if $\mathrm{rank}\,(\mathcal{A}(\boldsymbol{x})) = 2$) or it contains only a single point at a multiplicity of k (if $\mathrm{rank}\,(\mathcal{A}(\boldsymbol{x})) = 1$). If $\mathcal{N}_k(\boldsymbol{x})$ is planar, we compute the normal as a simple cross-product of any three pairwise different points of $\mathcal{X}$. If $\mathrm{rank}\,(\mathcal{A}(\boldsymbol{x})) = 1$, k must be increased in order to compute stable normals. Note that this is just a special case of a principal component analysis as described in Section 2.6.

The normals computed by this method are ambiguous, since their sign is not determined. Computing a consistent sign of the normals is NP-hard for general surfaces [HDD+92], but can be solved efficiently for 2-manifolds. However, computing such a

consistent orientation requires additional topological information—yet one of the main advantages of point-based rendering is that no triangulation has to be computed. Consequently, we utilize only the modulus of angles occuring in lighting computations.

Normal Encoding. Since adjacent normals in a short path show only slight variations, they can be encoded incrementally. Each normal but the first one is expressed in spherical coordinates relative to its predecessor. Let θ and ϕ be the azimuth and the longitude coordinates of the current normal. To avoid suboptimal compression at poles we compute both the negative and the positive angles and use the one that leads to a smaller difference. If the difference to the following normal in spherical coordinates is $(\Delta\theta, \Delta\phi)$, then the new normal in Euclidean coordinates is given by:

$$\begin{aligned} x &= \cos(\theta + \Delta\theta)\sin(\phi + \Delta\phi), \\ y &= \sin(\theta + \Delta\theta)\sin(\phi + \Delta\phi), \\ x &= \cos(\phi + \Delta\phi). \end{aligned} \tag{3.14}$$

Since this computation requires trigonometric functions to be evaluated, we employ the trigonometric relations

$$\begin{aligned} sin(\alpha + \beta) &= \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta), \text{ and} \\ cos(\alpha + \beta) &= \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta) \end{aligned} \tag{3.15}$$

to express Euclidean space coordinates in terms of pre-computed sine and cosine values. More precisely, given $\sin(\Delta\theta)$, $\cos(\Delta\theta)$, $\sin(\Delta\phi)$ and $\cos(\Delta\phi)$, as well as the respective values $\sin(\theta)$, $\cos(\theta)$, $\sin(\phi)$, and $\cos(\phi)$ of the previous normal, Euclidean coordinates for the current normal can be decoded using a few products and additions.

During path generation, we collect all normal increments in spherical coordinates that occur in the entire data set. These increments are then clustered using vector quantization [GG92, Sch03] (see also Section 2.7). The two angular increments in the codebook are stored as four sine and cosine values for each entry. To encode the start normal, we compute a vector quantization of all start normals using 2^{16} codebook entries. To avoid accumulation of quantization errors, we perform a rebinning in the following fashion. For each normal except the first two, we recompute the difference to the previous normal, which is reconstructed using the start normal and the sequence of previous differences. Then, the optimum codebook entry for the new difference is found and stored. In this context it is worth noting that re-quantization of differences, which may

seem an appealing alternative, is known to not converge in general. For all data sets in this chapter, we used 5 bits for each normal increment.

Color Encoding. Clearly, colors can be encoded similarly to normals. However, spherical coordinates are not meaningful for colors, and Euclidean distances should be encoded instead. Since colors—unlike normals—are not generally coherent across short paths, significantly more bits are required to encode them at sufficient precision. Tests show that the starting color requires about 8 to 10 bits to be represented faithfully, while differences require another 5 to 8 bits. The amount of bits needed strongly depends on the homogeneity of the colors to be encoded. We also tested various color spaces, i.e., YCbCr and HSV, but the gain in visual fidelity was rather small.

3.6 Extension to Isosurface Compression

So far, we discussed the encoding of point scans. In order to achieve significant compression ratios while maintaining high fidelity, the high coherence in these data sets was exploited. Another type of highly coherent data stems from CT (see also Section 2.5) or MRI[4] scans, as well as numerical simulations. Such data is typically given as a scalar function on a discrete Cartesian voxel grid $\mathcal{D}'$, i.e., $\Phi' : \mathcal{D}' \to \mathbb{R}$, where $\mathcal{D}' \subset \mathbb{N}^3$, although irregular (tetrahedral or hexahedral) cells are frequently used in numerical simulation. By means of interpolation with an interpolation kernel κ the domain $\mathcal{D}'$ on which Φ' is defined can be made continuous. Thus, a continuous function $\Phi := \kappa \star \Phi' : \mathcal{D} \to \mathbb{R}$ is obtained, where $\mathcal{D} \subset \mathbb{R}^3$ is used to denote the continuous domain of Φ. Note that the interpolation kernel κ should be chosen such to obey a convexity criterion, i.e., local minima and maxima of the interpolant can only occur at the control points.

An operation that is frequently performed on such data is the extraction of an isosurface, i.e., to determine the loci $\boldsymbol{x} \in \mathcal{D} : \Phi(\boldsymbol{x}) = \sigma_{\text{iso}}$. Here, σ_{iso} denotes the so-called *isovalue*.

Such an isosurface may either be rendered directly, e.g., by means of volume raycasting [KW03a], or a polygonal approximation may be obtained first. The latter approach has the distinct advantage that obtaining the isosurface is entirely decoupled from rendering. The polygonal approximation of the isosurface can thus be used for a wide range of purposes as is, without further specification of the rendering methods to be used. Since the time Lorensen and Cline proposed the Marching Cubes algorithm

[4] Magnetic Resonance Imaging

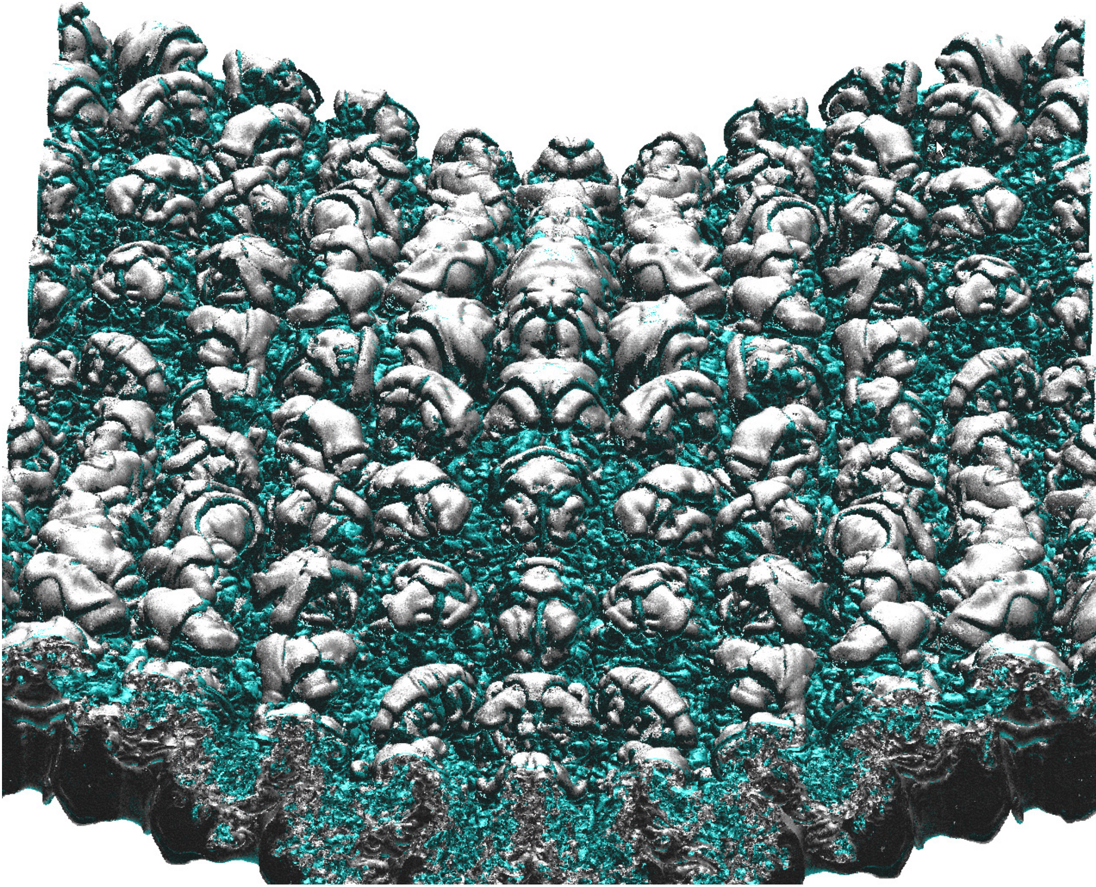


Figure 3.10: *A time step of an interface mixing instability simulation. In this image two isosurfaces (cyan and white) are rendered simultaneously. A Marching cubes isosurface extraction of these surfaces results in about 1 billion points consuming 24 GB for the point and normal information alone not considering connectivity. Due to the compression framework described in this chapter, both isosurfaces can be compressed to about 500MB. Data set courtesy of Lawrence Livermore National Laboratory.*

[LC87] for Cartesian voxel grids, many different methods to speed up the process of obtaining a triangle approximation or to render this polygonal representation have been proposed. These approaches can be roughly classified as follows.

Acceleration with hierarchical data structures. To speed up the rendering of isosurfaces, these methods strive to consider relevant parts of the data set only. To determine such relevant parts efficiently, hierarchical data structures such as the Octree [Lev90] or span-space trees [LSJ96] have been used.

View dependent isosurface reconstruction. These algorithms generate isosurfaces on the fly by using a view dependent error measure. This dramatically reduces the amount of data to be considered for isosurface extraction and the amount of geometry to be rendered [LH98].

3D texture based isosurface reconstruction. In texture-based isosurface extraction instead of generating the isosurface's geometry explicitly a shader program or a special GPU configuration is used to render only parts of the volume that do belong to the surface [WE98]. The normals required to compute the illumination are precomputed and stored in an additional gradient volume.

However, even if some of these algorithms can be modified to perform their tasks in an out-of-core fashion, the resulting tesselations can be overwhelmingly large. For instance, the marching cube algorithm generates up to four triangles for any cell that contains the isovalue. Combined with today's resolutions of up to 4096^2 samples per CT slice—not to mention typical resolutions of serious numerical simulations—the generated data can quickly become larger than available storage capacities. For instance, Figure 3.10 shows two isosurfaces in a interface mixing instability simulation. The data is originally given on a $2048^2 \times 1920$ Cartesian grid. The highly turbulent structure of the interface between two fluids results in a marching cubes reconstruction with 1 billion points that require roughly 24 GB to store both position and normal.

In order to be able to compress and render such large isosurfaces using the methods described in this chapter, only the offline encoding part of the DuoDecim engine (see also left half Figure 3.11; the rendering module depicted in the right half is discussed later) has to be modified to accept voxel data. This is achieved by replacing the uniform point clustering and resampling described in Section 3.3.1 by a pipeline that processes the voxel data and extracts an isosurface layer by layer. These layers then either have to be converted to 2D honeycomb grids, or the isosurface can be directly reconstructed on such a grid. The rest of the encoding pipeline can be kept completely unchanged.

3.6.1 Isosurface Preprocessing

To improve image quality, processing speed, and compression ratio we do not extract the isosurface in the original data volume but convert the volume into our HCP representation first. After the conversion we perform the isosurface extraction in HCP space and compress this data. This guarantees optimal connectivity of short paths and requires no further data transformation if the user changes the isovalue.

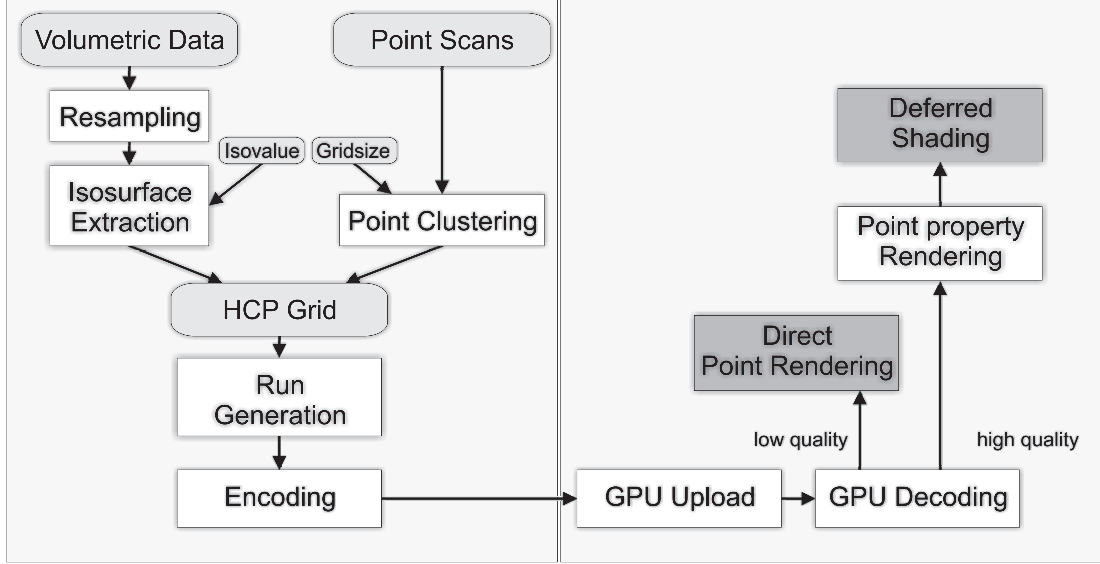


Figure 3.11: *The Duodecim Engine. There are two modules, the offline encoding part (depicted left), and the online rendering module. The offline encoding part takes either a volumetric data set or a point scan and compresses it into the DuoDecim format. The rendering module produces either fast low quality or high quality images. The performance impact of the high quality rendering mode is about 50%.*

Since optimal resampling to the HCP grid requires a spherical, bandwidth limited interpolation kernel [NM02], we use a spherical Lanczos kernel of radius ρ,

$$L_\rho(r) = \begin{cases} \mathrm{sinc}(r \cdot \pi/\rho) & \text{if } r \leq \rho,\, r \neq 0 \\ 1 & \text{if } r = 0 \\ 0 & \text{otherwise} \end{cases} . \quad (3.16)$$

Note that this kernel still has to be properly normalized. In the case of rectilinear grids, we keep enough slices in memory to cover the $2\rho + 1$ support of $L_\rho(r)$ and perform the convolution with $L_\rho(r)$ in the spatial domain. For unstructured grids this involves a cell search, which is implemented efficiently using standard binary space partitioning approaches for conforming grids with convex cells. Changing the radius ρ offers a speed/quality tradeoff.

This resampling results in a HCP grid with a single data value for every grid vertex. To compress an isosurface defined by the user, the system first classifies each vertex $\boldsymbol{v}$. If $\Phi(\boldsymbol{v}) \geq \sigma_{\text{iso}}$ the vertex $\boldsymbol{v}$ is assigned a tag $\oplus$, otherwise it is assigned a tag $\ominus$. Now the TRD cells for which a sign change occurs, i.e., there is at least one vertex

tagged ⊕ and at least one vertex tagged ⊖ incident to the cell, are determined. Under the assumption that Φ is convex in each TRD cell, the isosurface passes these and only these cells. Consequently, these cells are tagged as occupied and fed into the short-path generation and encoding modules exactly as is the case for point scans.

Normals of the isosurface can be computed as gradients in Φ. This is typically done using discrete differentials on the grid, e.g., centered differences in case of Cartesian grids. Since these normals are not ambiguous with respect to their direction, a standard Phong illumination [Pho75] can be computed. Figure 3.12 shows two data sets including normals. The two isosurfaces of the Wholebody CT scan in the left half of the figure comprise about 30 million points. Originally requiring about 1 GB to be stored, they can be compressed to only 21 MB using the DuoDecim framework. The interface mixing instability depicted in the right half of the figure consists of about 450 million points. Note the very fine details shown in the zoom. Both images show the compressed data.

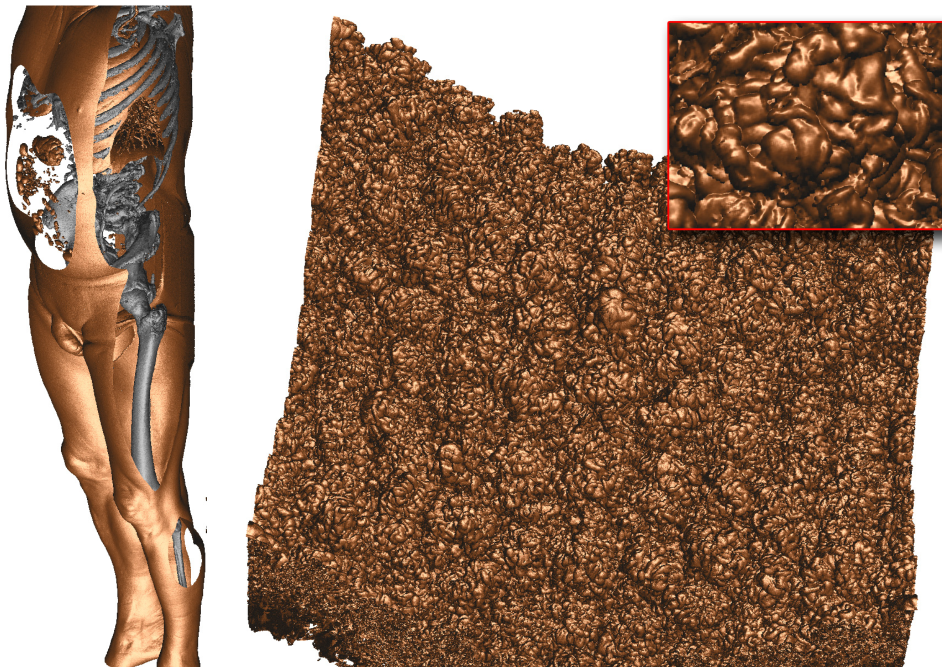


Figure 3.12: *Two large isosurfaces. Both data sets, including per-point normals, have been compressed and now fit into 256 MB video memory. Images are generated by rendering the scans directly from the compressed data stream on the GPU. Up to 50 million points per second can be decoded and rendered on an ATi X800 XT graphics card. Left: Two surfaces consisting of about 30 million points, compressed from about 1 GB to only 21 MB by our method. Data set courtesy of Siemens Corporate Research, Inc., Princeton. Right: Isosurface of an interface mixing instability. Data set courtesy of Lawrence Livermore National Laboratory.*

Since the optimum HCP grid size is known a priori for isosurface preprocessing, the costly grid size optimization step is skipped. Occupied cells are directly fed into the short-path generation step. Also note that the major axis across which the data set is to be sliced is either given or it can be computed trivially by considering the bounding box of the grid. This estimate of the major axis is not guaranteed to result in an optimum axis, but the result is typically sufficiently close for practical use.

3.7 Rendering

To render the compressed point set, the encoded data is traversed slice by slice. For each short path, the start position is decoded from the associated grid coordinate, and start normal and start color are fetched from the quantization codebook. All other point coordinates can then be decoded incrementally from the relative offsets that are stored with respect to the underlying grid structure. Only for the incremental decoding of normals and colors an additional lookup into the delta normal codebook is required. This process consecutively generates attributed vertices consisting of positition, normal, and optionally color, which are written either directly to a vertex array or to textures for future vertex texture fetches. Both functionalities are now available in the DirectX and OpenGL APIs. The decoded points can then be rendered using simple screen-aligned quads or other, more elaborate splatting methods (i.e., the one described in Section 3.7.2).

If decoding is performed on the CPU, the vertex and the normal array have to be sent to the GPU, making bus bandwidth a major bottleneck. To overcome this limitation, encoded "runs"[5] are sent to the GPU in compressed form. Due to the simplicity of the decoding process, runs can be directly decoded on the GPU using parallel streaming computations. Reconstructed point positions are rendered directly without any read back to application memory. In this scenario, the CPU is only used to control which runs are sent to the GPU, i.e., to accommodate view frustum and backface culling.

The GPU decoder exploits functionality on recent graphics cards, namely vertex texture fetches and render targets. The process is rather simple. First, the output generated by pixel shaders is stored in a floating point precision render target. This render target is then bound as a texture. In subsequent passes, the vertex shader can then fetch data from this texture in order to displace "dummy" vertices previously fed into the pipe. The rationale of this two-pass strategy—decoding data into a temporary buffer in

[5] It appears more appropriate to refer to the encoded paths as point runs instead of "point short paths", in analogy to other run-length encodings.

a first pass and rendering this buffer in a second pass—is that it is possible to generate multiple four-dimensional output vectors for each fragment rasterized. This is not possible for the vertex shader. The recent introduction of the geometry shader solves this issue, but so far geometry shaders do not offer satisfactory performance for most applications.

To prepare compressed point runs for GPU processing, they are stored in 2D texture maps. For each run, its 2D start position and 1D normal indices are stored in a 16 bit RGB texture map. Consecutive points in a run are encoded in 8 bit luminance textures, using the first 3 bits to store adjacency information and the remaining 5 bits to store quantized normal differences. These differences are decoded using a quantization codebook. Note that multiple runs up to the vendor-specific maximum texture resolution can be stored jointly in one texture. The principal layout of all data structures on the GPU is illustrated in Figure 3.13. If the data set has per-point colors, they are stored separately.

To render the model, the GPU decodes per-point properties and positions and renders them into intermediate buffers. In a second pass, the intermediate object containing point positions is fetched in the vertex shader to replace "dummy" vertices fed into the pipeline. Furthermore, the pixel shader of this pass reads per-point properties from the intermediate buffers in order to compute shading and illumination of the object. Note that this two-pass process does not require any read-back to the CPU. Also, the first stage can be implemented in the fragment shader that currently outperforms the geometry shader by great margins.

3.7.1 Culling and LOD

To render the compressed representation, the CPU determines the runs that need to be rendered under the current camera parameters. The buffers containing these runs are sent to the GPU and stored in textures. To keep bus transfer and GPU processing as low as possible, two different acceleration techniques have been integrated into our approach. Firstly, runs are clustered and stored in the same texture according to their cone of normals. Secondly, within one cone, runs are grouped according to their spatial position, i.e., each texture is split into a set of smaller textures for which axis aligned bounding boxes are computed. During run time, the CPU determines the partitions to be displayed based on the current viewing direction and the size and orientation of the view frustum. Only potentially visible partitions are send to the GPU, where they are finally decoded and rendered. Note that, since always contiguous blocks of runs are contained in a texture, a conservative visibility criterion is used to reduce the amount

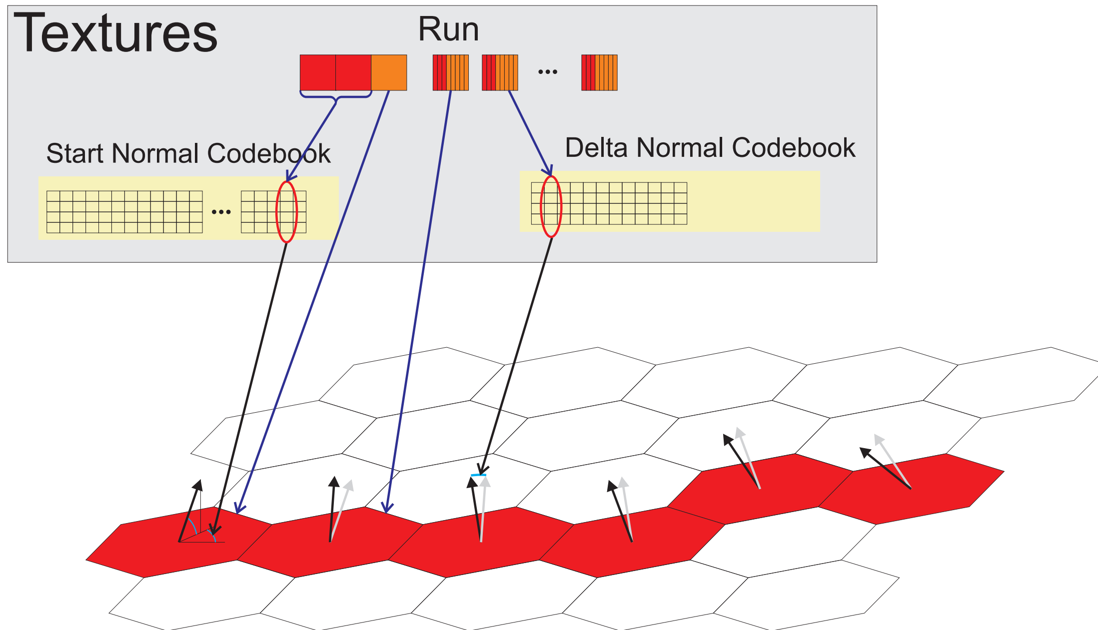


Figure 3.13: *Encoding of runs into texture maps on the GPU.*

of transfers to the GPU. Furthermore, for contiguous camera movements, the amount of uploads to the GPU per frame is very small.

The CPU also determines the most appropriate level of detail (LOD) to be rendered. We always select the resolution such that grid cells are projected to a screen-space area smaller than one pixel—with the only exception of the finest LOD of the data set being reached. An example of such a hierarchical LOD representation including a bounding box hierarchy is shown in Figure 3.14.

3.7.2 High-Quality Rendering

The default rendering mode uses decoded point positions and normals to compute a simple Phong illumination [Pho75], with the only change that the modulus of angles is used for this computation if the normal direction is ambiguous. Although this approach is very fast, it results in a per-splat constant color shading. Especially in close-ups for which the splat size can become significantly larger than a pixel this causes disturbing artifacts (see also Figure 3.15 left column).

To eliminate this problem a per-pixel illumination is required. As elaborated in Botsch et al. [BHZK05] deferred shading promises the best performance on current GPUs. To compute this per pixel lighting we accumulate the splat normals of the vis-

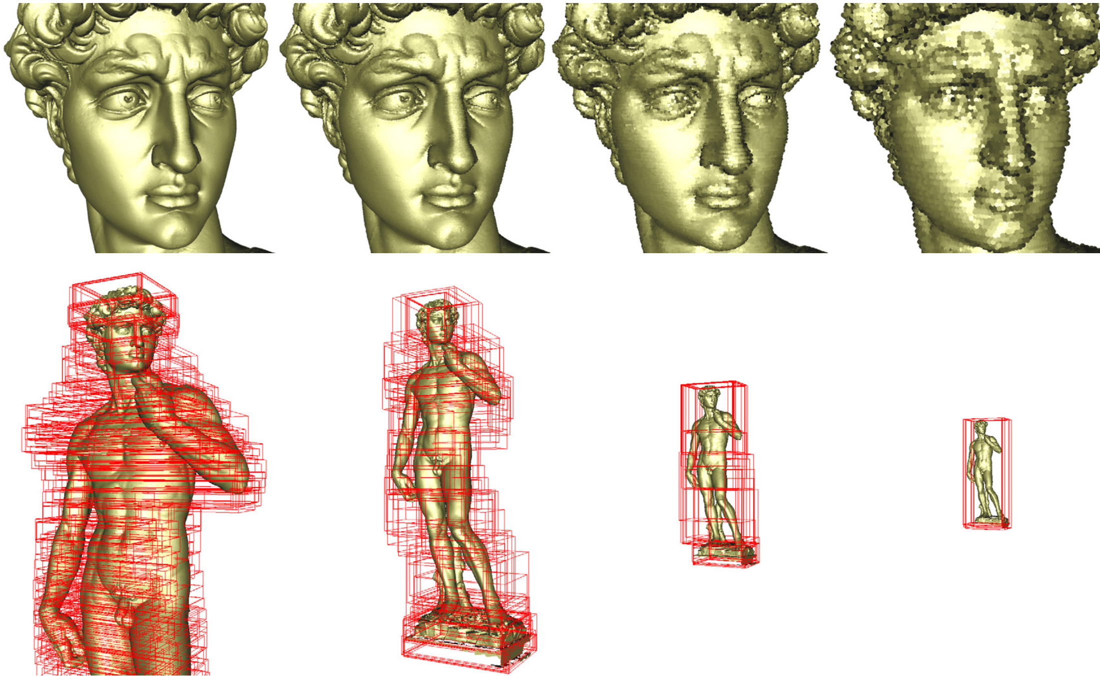


Figure 3.14: *David rendered at four different detail levels The upper row shows close ups of David's head rendered at 4 different levels of detail. The lower row shows the corresponding image of the David statue as it would be rendered. The bounding boxes enclose parts of the mesh that are tested for frustum culling. Data courtesy of the Digital Michelangelo Project.*

ible surface in the frame buffer. In the fragment stage, the engine weights the normal of each fragment with its distance to the splat center. Since this accumulation requires to blend potentially occluding splats, depth testing cannot be used at the same time. We hence first perform a depth-only pass to determine the splats visible under the current viewing parameters. If the target GPU does not support full 32 bit floating point blending (although all recent GPUs do), we accumulate normals in a 16 bit floating point render target. In our experiments, this did not result in any visible artifacts. After normals have been accumulated, we proceed by computing the illumination for each pixel. (see also Figure 3.16).

Since we need to render the points twice and do not assume to have enough GPU space for decoding the entire data set, we have to decode every point twice in the following multipass algorithm.

Depth-only pass. For each visible section of the data set we decode the point position and write the depth values into a single component floating point render target. The

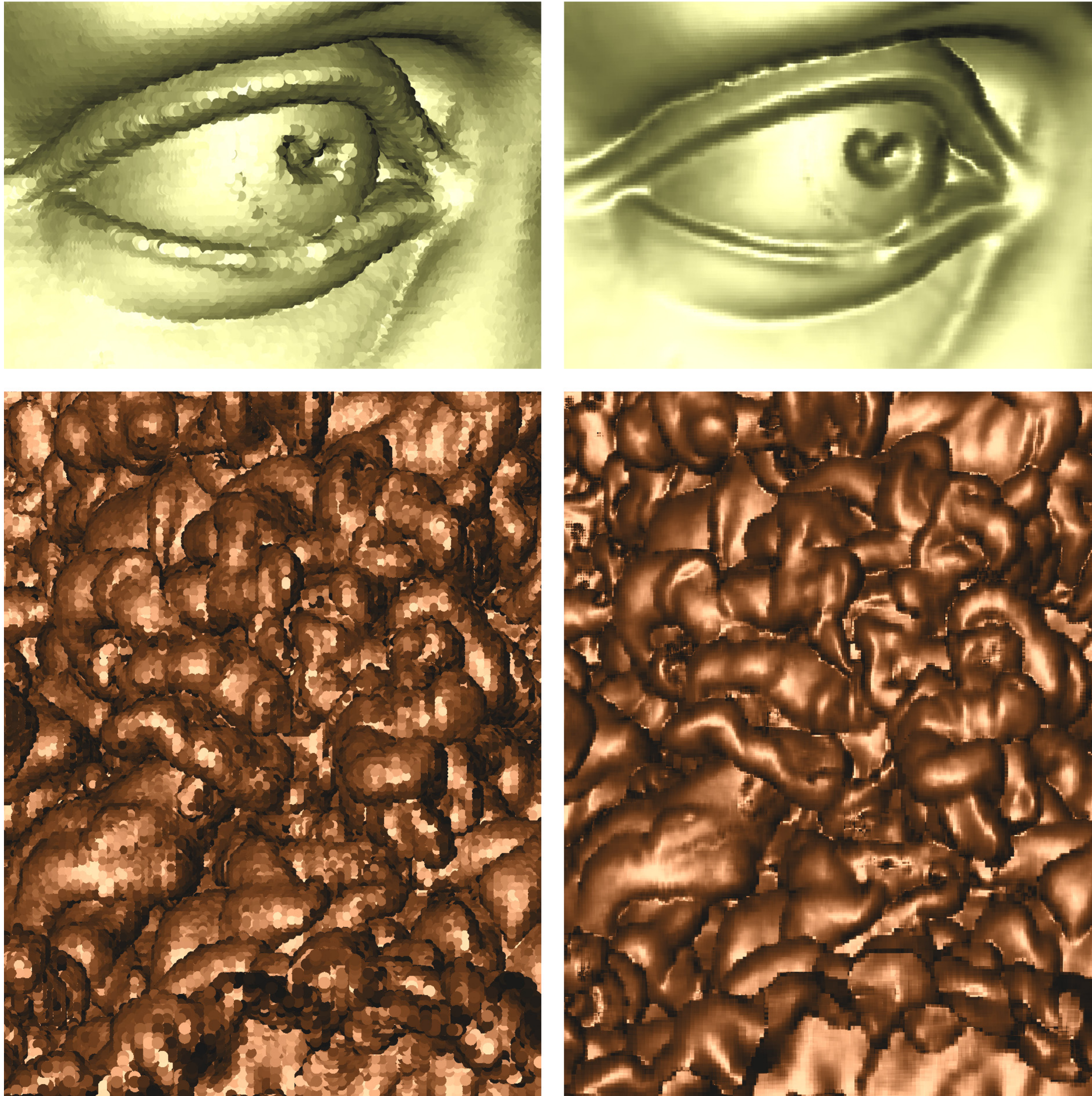


Figure 3.15: *Comparison of per splat and per pixel illumination. Left column: Per splat illumination is fast, but shows artifacts if the splat size exceeds the pixel size. Right column: Deferred per pixel illumination cures these artifacts. The top row shows a zoom onto David's eye, while the bottom row shows a zoom into the interface mixture instability. Data courtesy of the Digital Michelangelo Project.*

depth test is enabled during this phase in order to leave only the front-most depth values in the render target.

Property pass. For each visible section of the data set we decode the point position and point properties. In the pixel shader the depth value of every fragment is compared

Figure 3.16: *The three passes needed for our deferred shading approach. Left: The first pass generates a depth image. Middle: During the second pass the depth of incoming fragments is compared to the depth image obtained in the first pass. Fragments with a depth further than ε away from the respective depth of the first pass are discarded. The remaining fragments are blended in order to accumulate their propertiess. Right: In the third, purely image based pass the attributes accumulated in the second pass are used to illuminate the final image. Data courtesy of the Digital Michelangelo Project.*

to the depth value in the depth image of the first pass. Incoming fragments whose depth differs by more than a user-defined distance ε from the respective depth of the first pass are discarded. For all remaining fragments, their color is set to the property to be accumulated (i.e., normal, color, etc.). This color is premultiplied by a weight depending on the distance between fragment and splat center. If non-unit vectors are to be accumulated in this manner, e.g., colors, the weights are accumulated as well to allow for later re-normalization. Blending is then used to accumulate the respective property automatically. If other per splat properties such as colors or texture coordinates are present, they are stored in multiple render targets in this pass. Note that up to eight output textures or 32 output scalars can be written at the same time on recent GPUs. Thus, multiple passes are not generally needed during this step. Throughout this pass, the standard depth test has to be disabled. The ε depth test is necessary in order to avoid artifacts arising from the finite precision of the depth buffer and the fact that planar splats are to be blended. Two such splats with a common intersection, will—if not co-planar—diverge around their intersection. Thus, the depth of fragments generated around the intersection can vary significantly for the two splats. Consequently, ε has to be chosen significantly larger than the depth buffer's precision in order to guarantee smooth blending between such splats.

Illumination pass. In the final pass a single quad covering the entire view port is rendered. This quad generates a fragment for each pixel on the screen. For each fragment the pixel shader reads the accumulated normals and point properties. The normal as well as the properties are re-normalized and used to compute the illumination model. In this pass' pixel shader, any local illumination model can be evaluated, e.g., Phong shading, toon shading, edge enhancement shaders, and so forth. During this pass, fragment tests and blending are disabled.

This deferred scheme reduces the rendering performance by about a factor of two since every point needs to be decoded and data needs to be written to floating point targets. On the other hand, Figure 3.15 demonstrates that the smoother shading dramatically improves the overall image quality.

3.8 Runtime And Memory Requirements

In this section we provide asymptotic runtime and memory complexities for various parts of the algorithm. We denote the set of input points before resampling to a HCP grid as $\mathcal{X}$. We assume this HCP grid to be composed of N TRD cells. Note that despite the average hit rate γ being constant, this does not imply $N \in \mathcal{O}\left(|\mathcal{X}|\right)$ since γ is computed taking into account occupied cells only. Instead, since the topology of the vast majority of all models can be safely assumed to be dominated by 2-manifold parts, we can estimate the amount of occupied TRD cells by $\mathcal{O}\left(N^{2/3}\right)$ since the HCP grid encloses the model tightly. Furthermore, we assume each model to have a distinct major axis, such that at least $\mathcal{O}\left(N^{1/3}\right)$ slices each of which containing less than $\mathcal{O}\left(N^{2/3}\right)$ TRD cells are generated.

Isosurface extraction. For volumetric data, we assume that a data set has V voxels, or, in case of unstructured grids, V cells. The runtime for isosurface extraction depends heavily on the input grid type. For structured grids, interpolation and resampling is in $\mathcal{O}\left(V\right)$ as long as the filter size is constant. Since we stream the data set, we only need to store enough slices to cover the $2\rho + 1$ support of the Lanczos kernel, where each slice contains $\mathcal{O}\left(V^{2/3}\right)$ voxels. In the case of unstructured grids we usually have to perform a logarithmic cell search during interpolation, and hence the runtime is in $\mathcal{O}(V \log V)$. Memory requirements are still in $\mathcal{O}(V^{2/3})$. After resampling to the HCP grid, we store the entire grid to amortize the resampling process for multiple isosurfaces. Note that we do not consider any hierarchical acceleration structure for the isosurface extraction process.

Point clustering of laser range scans. First, points are inserted into the HCP grid, which requires linear runtime. However, range scans are first sorted, which clearly is in runtime of $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$. Each slice contains $\mathcal{O}\left(N^{2/3}\right)$ TRD cells, for which the possibility to be occupied is roughly $N^{2/3} : N$. Since we only hash filled voxels of one slice at a time memory requirements are as low as $\mathcal{O}\left(N^{4/9}\right)$.

It is worth noting that the assumption of a distinct major axis reduces memory requirements considerably. Finding a good guess for the best-possible major axis in case of point scans can be obtained by means of a principal component analysis. This step traverses each point in the data set once, without particular order, to compute the covariance matrix. Then, a real, symmetric 3×3 Eigenvalue problem is solved. Consequently, this step has a runtime complexity of $\mathcal{O}(|\mathcal{X}|)$ and a memory complexity of $\mathcal{O}(1)$. For volumetric data sets, the bounding box can be examined to determine the major axis in runtime and memory complexity of $\mathcal{O}(1)$.

Normal estimation. In case of laser range scans, if normals are not provided by the acquisition device, they have to be estimated from the point data. The complexity of this process is dominated by the computation of the k-neighborhood during the moving least squares fit [ABCO+01]. Utilizing a binary space partitioning of the input data, e.g., the ANN library [MA06], this k-neighborhood can be obtained in $\mathcal{O}(\log |\mathcal{X}|)$ for each point under the assumption that k is constant. Additional memory requirements for the binary search structure are typically in $\mathcal{O}(|\mathcal{X}|)$ and the structure is typically built in a runtime of $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$. For each point a 3×3 real, symmetric Eigensystem has to be solved, which can be assumed to be in constant time. Thus, the total runtime complexity for the entire normal estimation procedure is in $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$. For volumetric data, this step is straightforwardly solved by discrete differentials in a runtime of $\mathcal{O}(V)$ and a memory complexity of $\mathcal{O}(1)$.

Grid optimization for laser range scans. Finding a good initial guess for the grid's cell size requires to compute the average pairwise closest distance of all input points. This operation has a runtime complexity of $\mathcal{O}(|\mathcal{X}| \log |\mathcal{X}|)$ if space partitioning strategies are employed to accelerate the process. To store the binary space partition, a typical memory complexity of $\mathcal{O}(|\mathcal{X}|)$ arises. However, since the grid spacing is optimized in an automated process, it is generally sufficient to estimate a reasonable grid size using only a small (constant) fraction of the input points. This reduces both space and memory requirements to $\mathcal{O}(1)$. The remainder of the grid optimization requires only few resampling steps. Each resampling step has a runtime of $\mathcal{O}(|\mathcal{X}|)$ and memory

requirements of $\mathcal{O}\left(N^{4/9}\right)$.

Generation of short paths. So far, a single slice of the data sets always fitted into core memory. Runs are then generated using our linear-time 2-approximation. However, if in the future data sets should be so large that a single slice will not fit into memory any more, we mandate to "brick"[6] the data and to process each brick independently. Since there is a total number of $\frac{|\mathcal{X}|}{\gamma}$ occupied cells, where γ is the average hit rate, the total runtime of the run generation process is in $\mathcal{O}(|\mathcal{X}|)$. Since in case of bricking the grid can be partitioned (i.e., no overlaps arise), the asymptotic runtime complexity remains unaffected.

Quantization. To quantize the positions of each short path, we can traverse each such short path linearly. For the normals we first generate a codebook using vector quantization, which, if carefully implemented, is in $\mathcal{O}(p \log k)$, where k is the number of entries in the codebook and p is the number of vectors. However, since it is sufficient for large models to pick a small, representative subset p of constant size, obtaining a normal codebook is negligible in the entire process. To obtain such a subset, we choose up to 16 million start normals randomly from the entire data set. Since the number of start normals in this subset is not dependent on the size of the data set (that is, if the data set contains significantly more than 16 million start normals), memory requirements are in $\mathcal{O}(k+p)$. Once the codebook is obtained, we assign the closest entry to each normal. If k in $\mathcal{O}(1)$, this process is in $\mathcal{O}(|\mathcal{X}|)$, otherwise this is a full closest-entry search which requires a runtime of $\mathcal{O}(p \log k)$. Memory requirements throughout the quantization step are $\mathcal{O}(k)$ to store the codebook plus optional acceleration structures on this codebook. Furthermore, to obtain the quantization of start normals only $\mathcal{O}(1)$ memory is required if only a representative subset of all start normals is to be used.

3.9 Results

We validated the efficiency and effectiveness of the proposed point based rendering system using the large scans from the Digital Michelangelo Project (see Figure 3.17), CT scans (see Figure 3.18), and scalar fields resulting from numerical simulations (see Figure 3.19). Firstly, we would like to point out the rich amount of detail present in the original data sets. These details can be faithfully reconstructed even after significant

[6] *Bricking* refers to a partition into rectangular pieces of the data set. Sometimes it may also refer to a potentially overlapping cover of the data domain using—typically—rectangular pieces.

Figure 3.17: *The David and Atlas statues of the Digital Michelangelo Project.*

compression has been achieved by the methods described here. Table 3.1 shows comprehensive results for the three largest of the scans of the Digital Michelangelo archive. Table 3.2 gives results for a typcial CT scan and the interface mixing instability. All benchmarks were performend on a single-core P4 2.8 GHz CPU equipped with 1 GB RAM and an ATi X800 XT graphics card with 256 MB.

In all examples, run optimization resulted in a hit rate below 1.7 and a run efficiency above 70% of a maximum length of 25. As can be seen, even for the Atlas mesh the algorithm returns the result in less than 10 hours. Due to the hit rate larger than 1, the original point sets were reduced by a factor of 1.3 to 1.6.

It is obviously clear that the point clustering approach described introduces sampling errors. For the high-resolution examples presented, these errors are 0.11mm and 0.14mm. The scanners used for the Digital Michelangelo Project have a minimum sample spacing of $0.25\,\mathrm{mm} \times 0.25\,\mathrm{mm} \times 0.1\,\mathrm{mm}$ in a plane perpendicular to the laser [Cyb99]. In the worst case, two sample points are as much as

$$\sqrt{2 \times 0.25^2 + 0.1^2}\,\mathrm{mm} \approx 0.367\,\mathrm{mm} \tag{3.17}$$

apart, which is the minimum size of features that can be faithfully reconstructed by the scanning process. Because in all our examples the sampling spacing is significantly larger than the sampling error introduced by our compression method, features present

in the original data sets will not be destroyed. If the scanning device has sampled the data above the Nyquist rate of the original object, our sampling is well above this rate, too, resulting in equal visual quality of the original and the compressed point set.

Table 3.1 also shows the excellent compression ratio achieved by our method for real-world data sets exhibiting fine-scale details. When using ZIP compression, the DuoDecim-encoded VRIP version [CL96] of all of the Digital Michelangelo statues requires about 380MB and can thus be stored twice on an ordinary CD. Plainly encoded, it is still small enough to be stored in core for our target architecture. Due to the slice-based encoding scheme for point sets, which is at the core of our technique, it is also well suited for stream processing and progressive transmission of the data [IL05].

Although the point scans are considerably compressed, they can still be decoded very efficiently due to the simplicity of the decoding scheme. In the table we give timings for GPU decoding *and* rendering. To measure these timings, all acceleration techniques were switched off, therefore these timings are considerably slower in comparison to practical display times. If decoding is carried out on the CPU, we observe a loss in performance of about a factor of 13.

Table 3.1: *Timing and memory statistics for some of Michelangelo's statues. The benchmark was performed on an ATi X800 XT card using DirectX 9.*

Model	Atlas	St. Matthew	David
scan resolution	0.25 mm	0.25 mm	1.00 mm
# Points	254904158	186865425	28184522
# Samples	158877859	121718168	17190274
hit rate	1.60	1.53	1.64
run efficiency	72 %	74 %	72 %
max sampling error	0.11 mm	0.14 mm	0.48 mm
ply file size	9.94 GB	7.29 GB	1.1 GB
DD compressed size	231 MB	182 MB	28.5 MB
zip compressed file	172 MB	140 MB	21 MB
DD encoding time	9.5 hrs	6 hrs	57 min
decode & render time	3.91 sec	2.85 sec	0.39 sec

Table 3.2 summarizes the results for volumetric data sets. The Wholebody CT scan contains one 16 bit scalar per voxel cell, while the interface mixing instability contains one 8 bit scalar. As can be seen, due to the regular sampling of these data sets the run efficiency is slightly better than for the point scans. Note that even a binary encoding to

tag voxels through which the isosurface passes consumes considerably more memory than our DuoDecim representation for multiple isosurfaces. Memory requirements in the table are values for a single, typical isosurface.

Table 3.2: *Timing and memory statistics for some volumetric data sets. The benchmark was performed on an ATi X800 XT card using DirectX 9. Note that even compared to a binary encoding of voxels we achieve a significant compression ratio.*

Model	Wholebody	Interface mixing instability
volume resolution	$512 \times 512 \times 3172$	$2048 \times 2048 \times 1920$
volume size	1.6 GB (16 bit)	7.5 GB (8 bit)
triangulated mesh size	1.5 GB	24 GB
run efficiency	89%	81%
DD compressed size	20 MB	210 MB
zip compressed file	11 MB	120 MB
DD encoding time	1.7 hrs	8.2 hrs
decode & render time	0.23 sec	3.5 sec

On the GPU, the point rendering system achieves a throughput of about 50 million points per second. This rate includes the decoding of compressed point runs as well as the rendering of decoded points and normals. It is worth noting that we maintain this rate for large amounts of *distinct* points and normals. Figure 3.20 shows some more examples that demonstrate the need for a point based rendering system capable of handling such an amount of primitives. In all images, the point splat size is automatically set according to the screen space projection of the underlying grid cells.

3.10 Summary

In this chapter we presented an effective compression scheme for gigantic points scans based on close sphere packing grids. Such grids provide a structure for optimal point clustering, and they establish a spatial relation between points that can be exploited for compression purposes. Our results show that the compression scheme achieves an extraordinary compression ratio at very high fidelity. Due to the simplicity of the decoding scheme, point coordinates and normals can be reconstructed on the GPU. Since the GPU can also render the decoded primitives without any read-back to the CPU, bandwidth requirements are substantially reduced.

Furthermore we demonstrated that even though GPU rendering includes decoding

of point coordinates as well as processing of point geometry, a throughput of 50 million points per second can be achieved. To our best knowledge, this has not been achieved before.

Interesting directions for future research include the design of more elaborate LOD strategies, especially those that lead to an embedded coding [Say00] of the data set. Embedded codes partition the data set into an interleaved, hierarchical representation that can effectively avoid the storage overhead that is typically implied by hierarchies. In case of the compression scheme presented here this overhead is about 14%. Also, a different space partitioning scheme to allow improved culling seems promising. Another interesting direction would be the integration of even higher quality rendering modes, such as perspectively correct splats or elliptical weighted average (EWA) splats [GH86, PZvBG00].

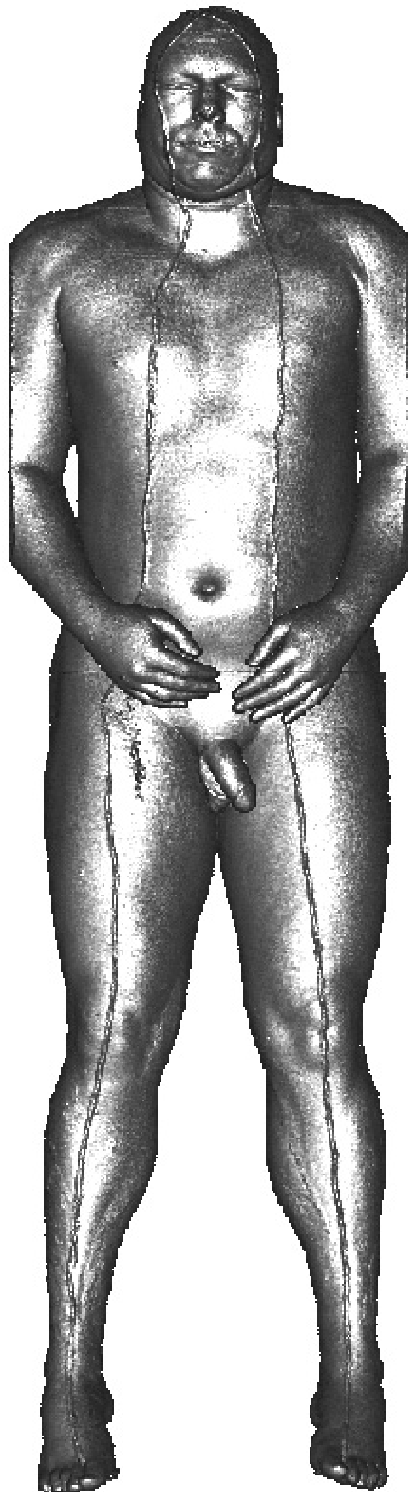


Figure 3.18: *An isosurface of the Visible Human male data set. While the volume data requires about 1 GB, the compressed data stream rendered in this image is only about 5MB in size. Data courtesy of the US National Library of Medicine.*

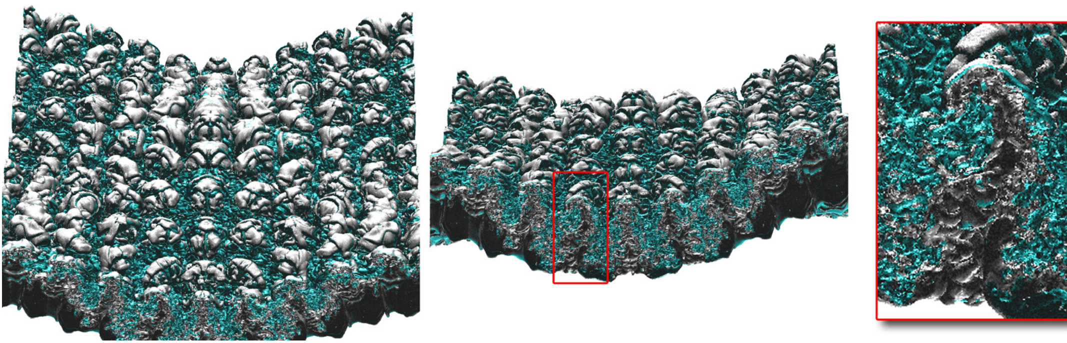


Figure 3.19: *Another time step of the interface mixing instability. Two isosurfaces (cyan and white) are rendered simultaneously directly from the compressed data stream. Note the fine scale detail visible at the bottom, where the mixing interface forms thousands of vortices. Data courtesy of the Lawrence Livermore National Laboratory.*

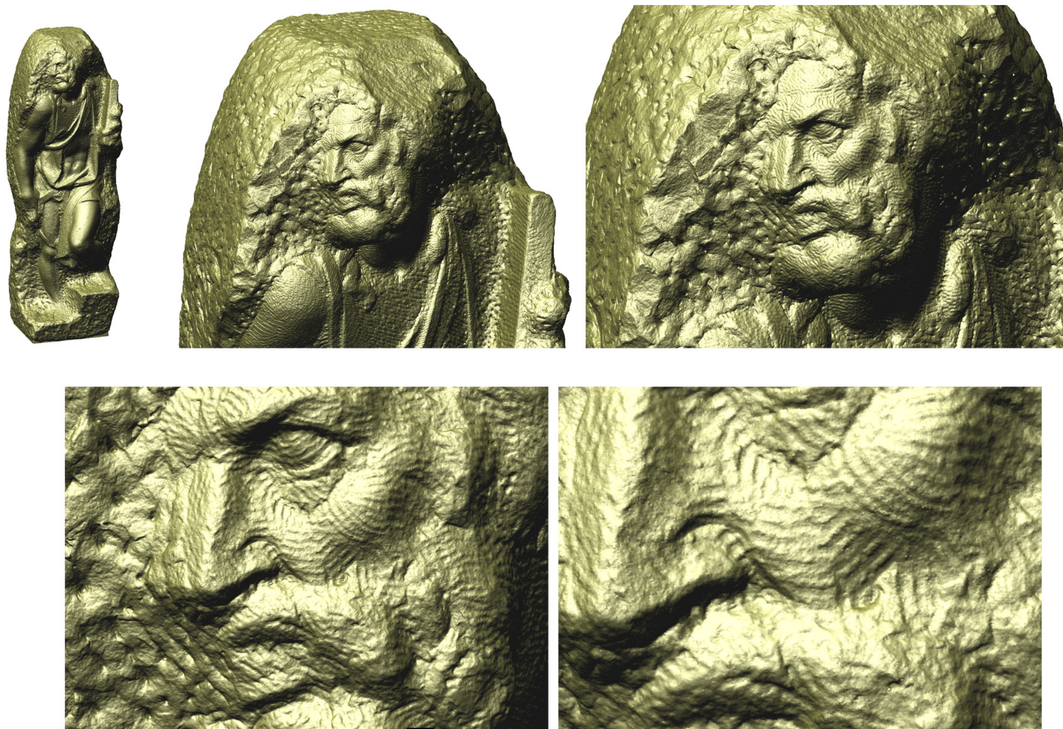


Figure 3.20: *Zoom onto Michelangelo's St. Mathew Statue. Note the fine scale details even in the lower rightmost closeup. Data courtesy of the Digital Michelangelo Project.*

Chapter 4

Rendering Of Large-Scale Terrain

Today, satellite range scans comprised of over a billion of samples are available, making even the handling of such data sets difficult to perform due to memory constraints. In addition, even single display solutions are surpassing 9 Mpixels [IBM]. The result is a demand for a substantially increased number of primitives to be transferred to and processed by the GPU. The requirements imposed by current and future data acquisition and display technologies make real-time visualizations difficult to perform on even the most powerful workstations. Therefore, the need for a terrain rendering system that comprehensively addresses the aforementioned issues is clear.

4.1 Contribution

In this chapter we describe a rendering technique based on a level-of-detail (LOD) approach for large, textured terrain. The method presented here is well-suited for recent GPUs and is thus able to tap the full potential of recent graphics hardware both with respect to their superior processing and memory bandwidth capabilities. To achieve

Figure 4.1: *A* 360° *panorama of the Alps. This image has a resolution of 7168* $\times$ *1024 pixels and was generated with our method in less than 4 seconds. This time includes rendering, reading data back from the GPU, and writing the final image to the disk. Data courtesy of DLR Oberpfaffenhofen (Deusches Zentrum für Luft- und Raumfahrt).*

highly interactive frame rates, we combine the advantages of semi-regular, continuous level of detail meshes with the advantages of discrete, precomputed LOD hierarchies. Thereby, any retriangulation is avoided during runtime. In contrast to some previous approaches such as the Batched Dynamic Adaptive Meshes [CGG$^+$03a], we show that by avoiding expensive irregular triangulations the preprocessing time to compute discrete LODs can be reduced from several hours to a few minutes. Furthermore, our method guarantees the input terrain to be refined within an user-defined screen- *and* world-space error. High-quality renderings such as the one depicted in Figure 4.1 are achieved by continuous LODs through geomorphing and photo texturing. During rendering, aliasing is avoided by employing optimal geometry filtering at the best possible geometric resolution. To reduce bandwidth requirements across the graphics bus, discrete sets of decimated mesh structures are transmitted progressively. Rendering is then performed entirely on the GPU by utilizing recent shader functionality.

4.2 Algorithmic Overview

Given a height field specified on a Cartesian grid, the domain is first decomposed into a set of equally sized *tiles*. For each such tile, a discrete set of LODs is computed by means of a *nested mesh hierarchy*. The construction of this hierarchy is described in Section 4.4. This particular choice of a nested hierarchy has several beneficial properties. Firstly, the terrain is decimated according to world-space errors given for each level. Similar to other decimation techniques, this reduces the amount of triangles greatly in planar regions. Secondly, a vertex introduced at a certain level in the hierarchy is present at that particular position of the domain in all finer levels ("nestedness"). This allows vertices of the mesh to be progressively transmitted to the GPU from one level to the next. Thus, bandwidth requirements between CPU and GPU are significantly reduced. Thirdly, geomorphing only requires the specification of multiple height values per vertex. The vertex shader can then interpolate between these values efficiently. As a result, we can guarantee very low pixel errors at high frame rates during typical fly-overs. The data structures used to represent and render this nested hierarchy on the GPU are discussed in detail in Section 4.5. Since per-vertex normals are known to result in artifacts even for very simple level-of-detail hierarchies (see, for instance, the recent paper by Han et al. [HSRG07]), we mandate the use of pre-lit photo textures. These are compressed using the S3TC standard [INH99], which allows high-resolution mipmaps and anisotropic filter kernels to be used with low performance overhead. All data is stored in vertex buffers and 2D textures which are handled by a memory manager

to minimize bus transfer. This component is described in Section 4.6.

4.3 Related Work

From a high-level view, previous approaches for terrain rendering can be classified into the three following categories.

View-dependent refinement. These methods construct a continuous LOD triangulation on the CPU with respect to a given world- and screen-space error. Gross et al. [GGS95] employ a wavelet decomposition to generate adaptive quadtree meshes for terrain data, combined with a lookup-table to store an irregular triangulation for each of the possible quadtree leafs. Restricted quadtrees [VHB87] were first introduced by Pajarola [Paj98] for the purpose of terrain rendering. In their seminal paper Duchaineau et al. [DWS+97] present the *ROAM* algorithm that uses triangle bintrees to perform the remeshing. Peucker et al. [PFL78] propose to use triangulated irregular networks (TINs). The first automatic process to compute such TINs is described by Fowler et al. [FL79]. Garland and Heckbert [GH95] employ a greedy insertion strategy to construct a TIN. Progressive meshes are modified with respect to the demands in terrain rendering by Hoppe [Hop98].

To speed up the remeshing process, frame-to-frame coherence can be exploited. Duchaineau et al. [DWS+97] use incrementally updated priority queues to guide the remeshing process. In a different approach, Lindstrom et al. [LKR+96] use a quadtree data structure with incremental updates of vertex dependencies. Hoppe proposes to keep track of *active cuts* to achieve an incremental update [Hop98]. While the exploitation of frame-to-frame coherence usually results in a reasonable speed up, for particular camera movements—such as shoulder views in an airplane simulation—a considerable loss in performance can be observed. Furthermore, frame-to-frame coherent approaches are usually harder to implement due to their additional LOD constraints. This was recognized by Lindstrom and Pascucci [LP01, LP02], who propose a simple, yet efficient method to rebuild the mesh from scratch in every frame. In their work, they improve the error metric proposed by Blow [Blo00].

If the terrain gets excessively large, many algorithms choose to partition the terrain into square blocks or chunks of data, which can then be processed independently of each other [KLR+95, SN95]. The advantage is that these chunks can also be paged from external storage independently. However, care has to be taken in order to avoid invalid vertices (so-called T-vertices) at chunk boundaries. Röttger et al. [RHSS98]

describe a particularly elegant approach to avoid these invalid vertices. By restricting the error metric, they automatically guarantee a valid mesh. However, a generalization to chunked meshes is not trivial and would also limit the error metric to a Manhattan distance.

More recently, Ulrich [Ulr00] suggests to use restricted quadtree meshes without boundary constraints for the chunks, and to fill possible cracks between chunks using *flanges* or *skirts*—fins of geometry along the boundaries pointing downwards from the terrain. However, ensuring correct anisotropic texture filtering at these boundaries is not trivial due to the different viewing angle. A more general approach is to stitch boundaries together using so-called zero-area triangles (also called *ribbons* in [Ulr00]), which guarantee correct filtering.

Pomeranz [Pom00] suggests a variant of the ROAM algorithm called RUSTiC that uses surface triangle clusters. To ensure validity, clusters are enforced to uphold an edge constraint. On shared edges the clusters must share vertices exactly. This approach is also one of the first terrain rendering algorithms exploiting graphics hardware. RUSTiC achieves its improved performance over ROAM by rendering clusters as triangle strips. Using 4-8 meshes is proposed by Hwa et al. [HDJ04] to induce a diamond-based hierarchy on both textures and height field. Combined with a memory layout that follows a space-filling curve this method allows for efficient out-of-core rendering of the terrain by utilizing GPU memory as a cache. However, since each other texture level is rotated by $45°$, a costly update of vertex texture coordinates has to be performed. Furthermore, ensuring correct anisotropic filtering is involved with this approach.

Pre-computed geometry batches. Based on the observation that the time that is saved by rendering less triangles due to adaptive re-triangulation is entirely amortized on recent GPUs by the time needed to perform the re-triangulation, several authors suggest to pre-triangulate the input data as much as possible. Cignoni et al. [CGG$^+$03a] suggest to replace triangles in the remeshing process by a *batch*, a new primitive that approximates the terrain across a triangular part of the input domain by a precomputed TIN. Computing a triangle strip for each such batch then results in a highly efficient rendering procedure. Similar to the ROAM algorithm, batches are kept in a bintree, for which the usual remeshing during run time is performed, hence the name of the method: Batched Dynamic Adaptive Meshes (BDAM).

In [CGG$^+$03b], the authors improve on their previous work to successfully render planet-size meshes at interactive rates. Their system does not support geomorphs, but a screen-space error of one pixel for a 640×480 view port is typically guaranteed.

However, the paper does not discuss scalability with respect to the size of the view port, which can become a severe issue in the near future. The reason is that with higher screen resolutions significantly more triangles have to be rendered in order to meet a given screen-space error.

A GPU-based system similar to the approach by Ulrich [Ulr00] that stores pre-meshed tiles in quadtrees was proposed by Wahl et al. [WMD+04]. This system also abandons a consistent triangulation across tile boundaries in favour of flanges.

Non-adaptive triangulations. Only very recently, the method proposed by Losasso and Hoppe [LH04] takes full advantage of the speed of current consumer class GPUs. This method abandons any view-dependent remeshing in favor of so-called *geometry clipmaps*, a triangulation that offers approximately uniform resolution in screen space. Specifically, concentric, uniformly tessellated, square patches around the camera are used which drop exponentially in resolution with distance. During run time, geometry is fetched from a toroidal buffer residing on the GPU. The update of this buffer is done by the CPU.

Since height fields show a very high spatial coherence on regular, Cartesian grids, they can be compressed very efficiently. The aforementioned approach exploits this fact by applying a compression scheme derived from Microsoft's WMV format [Mal00] and achieves compression ratios of up to 100:1. However, since decoding the compressed data puts a considerable amount of work on the CPU, the decoder can eventually fall behind faster camera motions, resulting in a blurry representation of the terrain. Contemporary multi-core architectures may be suited to alleviate this issue, but increasing display resolutions on the other hand may be able to effectively counter the effects of increased computing power. Furthermore, although geomorphs are not an issue for this system, both the screen- and world-space errors are hard to control, resulting in an world-space rms of about 1.5 meters in the original publication. Two additional problems of the method are that optimal geometry filtering cannot be performed due to the screen-space aligned topology, and that height fields compress a lot better than regular images. Especially the latter issue is likely to result in a major increase in memory requirements, if photo textures are to be used during rendering. Still, extremely high frame rates for virtually arbitrarily large data sets can be achieved using this method.

4.4 Nested Mesh Hierarchy

The most common way to avoid sampling artifacts in terrain rendering is by means of a LOD representation. Such a hierarchy can either be implicitly represented by performing an adaptive retriangulation at run time, or it can be explicitly precomputed for discrete LOD levels.

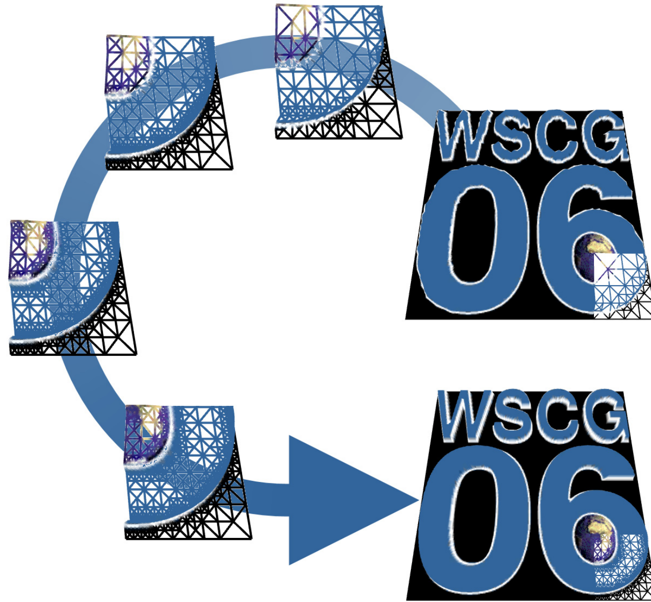


Figure 4.2: *Levels of the nested mesh hierarchy. From [SW06].*

A given height field $\mathcal{H} : \mathcal{D} \to \mathbb{R}$ can be approximated by a triangular mesh parameterized over its 2D domain $\mathcal{D} \subseteq \mathbb{Z}^2$. The surface of this mesh defines a reconstruction $\mathcal{H}'$ of $\mathcal{H}$. The approximation quality of the mesh is then measured by a point-wise error metric $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ that is extended to the entire domain simply by accumulating point-wise contributions. We use the canonical extension of the $L_{\max}$ error metric to measure the error between $\mathcal{H}$ and $\mathcal{H}'$,

$$\delta(\mathcal{H}, \mathcal{H}') \quad := \quad \max_{x,y} \delta(\mathcal{H}(x, y), \mathcal{H}'(x, y)). \tag{4.1}$$

By generating approximations of the height field with decreasingly lower approximation errors, a mesh hierarchy that represents the original terrain at ever finer scales is constructed. The hierarchy employed in this work is *nested* with respect to the triangulation. For each triangle of level i that covers a part $\Omega_i \subseteq \mathcal{D}$ of the height field's domain, there is a triangle on the next coarser level $i-1$ covering $\Omega_{i-1} \subseteq \mathcal{D}$ such that $\Omega_i \subseteq \Omega_{i-1}$.

That is, if both triangles are projected onto the domain, for each triangle T_i at level i there is exactly one triangle at the next-coarser level $i-1$ that completely contains T_i. Such a hierarchy is automatically generated by a restricted quadtree [VHB87, Paj98], bintree [DWS+97], or red-green refinement [BSW83].

To generate a discrete set of nested hierarchy levels, the terrain is decomposed into tiles of a fixed size of 257^2, with an overlap of one sample in either direction. Then, an error vector $(\varepsilon_0, \ldots, \varepsilon_{n-1})$ of exponentially decreasing thresholds $\varepsilon_i := 2^{n-1-i}$ is built, where the ε_i are usually measured in meters or feet. The particular choice of exponentially decreasing entries is motivated in Section 4.5. Starting with ε_0, a hierarchy $\{\mathcal{M}_i\}_{i=0}^{n-1}$ of restricted quadtree meshes is constructed. Each $\mathcal{M}_i$ can be associated with a planar graph, $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$, which is obtained by projecting $\mathcal{M}_i$ into the base domain. Furthermore, the surface of each $\mathcal{M}_i$ is a reconstruction $\mathcal{H}'_i$ of the original heightfield $\mathcal{H}$. The hierarchy is constructed such that

$$\begin{aligned} \mathcal{V}_i \subseteq \mathcal{V}_{i+1} \quad \forall \quad i = 0, \ldots, n-1 \text{ (nestedness criterion), and} \\ \varepsilon_{i+1} \le \delta(\mathcal{H}'_i, \mathcal{H}) \le \varepsilon_i \quad \forall \quad i = 0, \ldots, n-1 \text{ (error criterion).} \end{aligned} \tag{4.2}$$

This is achieved by constructing $\mathcal{M}_{i+1}$ in a recursive top-down approach from $\mathcal{M}_i$. Meshes constructed in this way automatically obey the “nestedness” criterion.

To generate the next finer hierarchy level from the current mesh, recursive quadtree refinement is performed until one of the following two conditions is met.

1. the maximum deviation between the new mesh and the original terrain is less than the error threshold defined for the level, i.e., the error criterion of Equation (4.2) is met.

2. the spacing between vertices of the mesh becomes smaller than the error threshold defined for the level.

The second criterion is enforced by prohibiting the quadtree from being refined below a certain scale. This weakens $\varepsilon_{i+1} \le \delta(\mathcal{H}'_i, \mathcal{H}) \le \varepsilon_i$, but $\delta(\mathcal{H}'_i, \mathcal{H})$ is typically still less than ε_i. In this way we can avoid aliasing artifacts due to subsampling along the domain axes. In a second step geometry changes are propagated from fine to coarse following the push/pull paradigm for data propagation in trees. Subquadtrees are refined as needed to avoid T-vertices.

The quadtree is then decomposed into recursive triangle fans [RHSS98] or a single triangle strip [LP02]. Using triangle strips is possible in our framework, but generating them increases the time spent for preprocessing considerably. Triangle fans, on

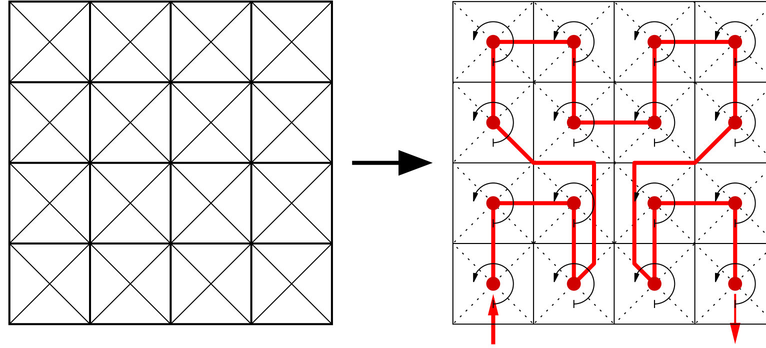


Figure 4.3: *Quadtree mesh and* Π*-order traversal. Note that since each sub-cell of the quadtree mesh is rendered as a separate triangle fan a consistent orientation can be achieved easily (black arrows). This is required in order to perform backface culling.*

the other hand, are easy to implement, reduce meshing time, and are similarly cache friendly as strips. However, generating fans results in a lot of separate primitives. In order to render these primitives efficiently, primitive restarts are employed. Primitive restarts are available on recent NVIDIA GPUs and are exposed in OpenGL by the GL_primitive_restart_NV extension. When rendering indexed vertices, the user may define a special index. Whenever this index is encountered, no vertex is fetched but instead a new primitive is started. In DirectX 9 and 10 these restarts are directly integrated into the standard API; the only difference is that the API reserves a specific index rather than using an user-defined value. This mechanism allows a list of fans to be rendered efficiently using only a single draw call, thereby reducing state changes and setup overhead. The quadtree is then traversed recursively in depth-first order to generate fans. These specific triangle fans are also sometimes referred to as *recursive fans*. As a result, we visit each fan along a Π-order space-filling curve (see Figure 4.3)—a curve that was successfully used in [LP02] to serialize memory layouts. This traversal has the beneficial property that subsequent fans in this serialization have a very high probability to be adjacent. If subsequent fans are adjacent, the second one can re-use two or even three vertices of the previous one. Since each fan has at most 9 vertices, the last fan will always be cached entirely on current GPUs[1]. Thus, recursive fans can re-use between $2/8$ and $3/6$ of their vertices (see also Figure 4.4).

To obtain a continuous LOD representation, we interpolate between the discrete LODs $\mathcal{M}_i$. This is known as *Geomorphing* [FEKR90]. In a nested hierarchy, vertices retain their position within the domain during morphing; only their height changes from one level to another. This is due to the property $\mathcal{V}_i \subseteq \mathcal{V}_{i+1}$. It is thus possible to store

[1]For instance, the GeForce 4 has 24 vertex slots to be used as general vertex caches.

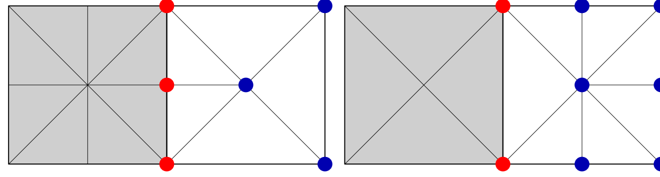


Figure 4.4: *Best and worst cases for vertex cache re-usal of fans. The gray fan can re-use the red vertices of the white fan, resulting in a cache coherence of at least 25%*

one height value per vertex for level i, i.e., the level in which a vertex $\boldsymbol{x} \in \mathcal{V}_i$ first "occurs", and one additional value for each coarser level $k < i$. Since $\boldsymbol{x}$'s domain position is not used for coarser levels, an appropriate height value can be computed as follows. First the triangle T_k at level k, $k < i$ that contains $\boldsymbol{x}$'s domain position is determined. Then, T's surface is interpolated using barycentric interpolation to yield a height value for $\boldsymbol{x}$'s domain position. To render a LOD ρ, the triangle mesh at the finer level $\mathcal{M}_{\lceil\rho\rceil}$ is rendered and vertices are morphed between the heights corresponding to levels $\lceil\rho\rceil$ and $\lfloor\rho\rfloor$ accordingly. Although higher order interpolation is possible, only linear interpolation is considered here for efficiency reasons. The details are described in Section 4.5.3 and Section 4.5.4.

4.5 Rendering Framework

As a benefit of the nested mesh hierarchy, tiles can be uploaded progressively to the GPU. On the GPU, an appropriate data structure accommodates real-time rendering at high quality, including photo texturing. Optionally, if high-resolution view ports require the screen-space error to be increased, geomorphing is performed on the GPU. At the same time, the CPU performs view frustum culling and level-of-detail computations for each tile. Since all GPU tasks are programmed in a high-level shading language, the entire framework is extensible and can easily be tailored to fit custom needs.

4.5.1 GPU Data Structures

As soon as a particular tile has to be rendered, a vertex buffer large enough to store all shared vertices of that tile is created. In this buffer, vertices are organized in blocks according to their respective hierarchy levels. (see also Figure 4.5). The associated topology is stored in one separate index array for each level. The ith index array contains only indices into the first $i+1$ blocks of the vertices. Such a shared vertex representation has two major advantages. Firstly, it reduces storage requirements compared to non-

shared representations. This is of special importance when additional vertex attributes, such as geomorphs have to be stored. Secondly, it enables progressive transmission by re-using vertices of coarser levels.

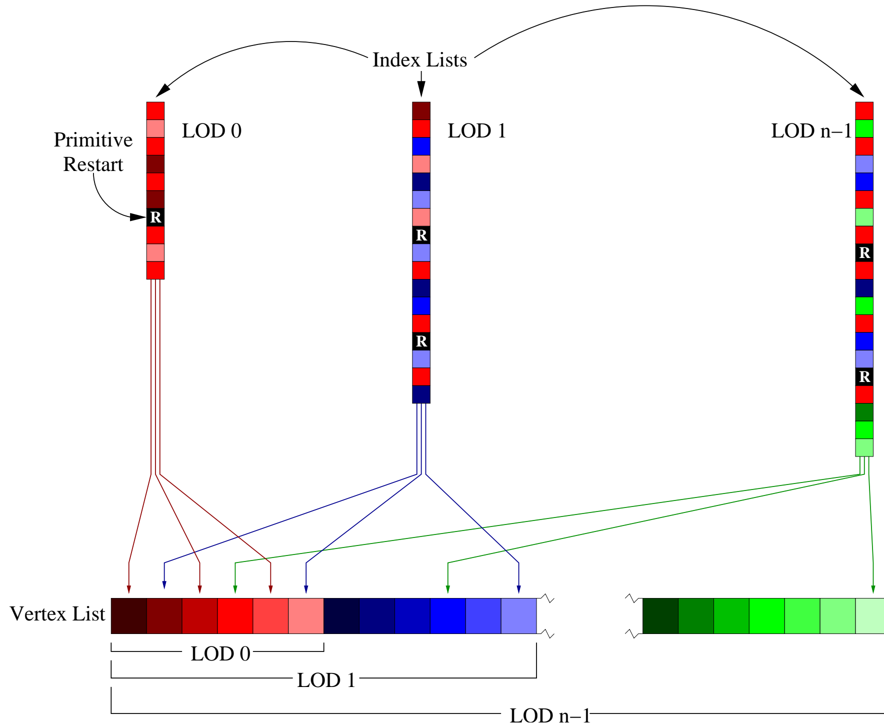


Figure 4.5: *The GPU data structures used in our terrain renderer. Index lists are always completely independent of each other and can be transmitted to the GPU one by one. For the shared vertex list, this is typically not the case. In contrast to other methods, vertices can be trivially partitioned to allow progressive transmission to the GPU due to the nestedness of our hierarchical LOD representation.*

Since all tiles used have a resolution of 257^2, relative domain coordinates are encoded in 9 bits. The height value can be considerably larger. It is therefore encoded using 14 bits. All three values are stored in two 16 bit vertex attribute components. These 32 bits are decoded in the vertex shader during rendering.

If geomorphs are enabled, additional storage requirements arise. The method is still memory efficient, since only one additional height value per coarser level needs to be stored. Since usually only small offsets to the original height are needed, 8 bits per value are sufficient. This allows us to morph vertices within a range of $[+127, \ldots, -128]$ units. Larger values are clamped to this interval.

4.5.2 Run-Time Processing

For each tile we store an axis-aligned bounding box to perform view frustum culling on the CPU. For each frame, visible tiles are depth-sorted to exploit the early-depth test, if available, thereby reducing overdraw. A memory manager, which is described below, ensures that all visible tiles can be rendered by paging in data that is not yet resident on the GPU.

Then, the appropriate LOD for each visible tile is computed. The index buffer as well as the vertices required to render the respective level are sent to the GPU, if not already resident. If a tile has been previously rendered, at least a subset of vertices has already been sent to the GPU. In this case, only the remaining vertices required to render the current level are transmitted and written to the respective vertex buffer on the GPU. In this way, even though an array large enough to keep all vertices has to be allocated on the GPU, bandwidth requirements at run time are substantially reduced.

To avoid mesh cracks at tile boundaries, neighboring tiles are stitched together using zero-area triangles. For each tile and each level in the hierarchy, the set of border vertices including all attributes is duplicated in system memory. Whenever two neighboring tiles are visible, the necessary triangles to fill out T-junctions are generated on the CPU and are then rendered. This can be done very efficiently by observing that these borders are always sorted with respect to a common axis. Generating zero-area triangles thus proceeds by “zippering” these borders, as depicted in Figure 4.6. Since this process uses the exact same data on the GPU and on the CPU, and since to all vertices the same GPU programs are employed, cracks are resolved without any numerical precision issues.

4.5.3 Level of detail

Determining the appropriate LOD for each tile and vertex requires the projection of the user-defined pixel error to object space. Previous approaches rely on conservative estimates of this error and often over-estimate the error. The result is that aliasing might still occur even for screen-space errors larger than one pixel. We compute a more precise error metric by using the current projection matrix directly, which maps homogeneous object coordinates $\boldsymbol{v} = (v_1, v_2, v_3, 1)$ to screen-space coordinates $\boldsymbol{s} = (s_1, s_2, s_3)$. Here, s_3 corresponds to the depth value. The appropriate scale of details ρ can then be computed in a similar way as the appropriate mipmap scale for texturing

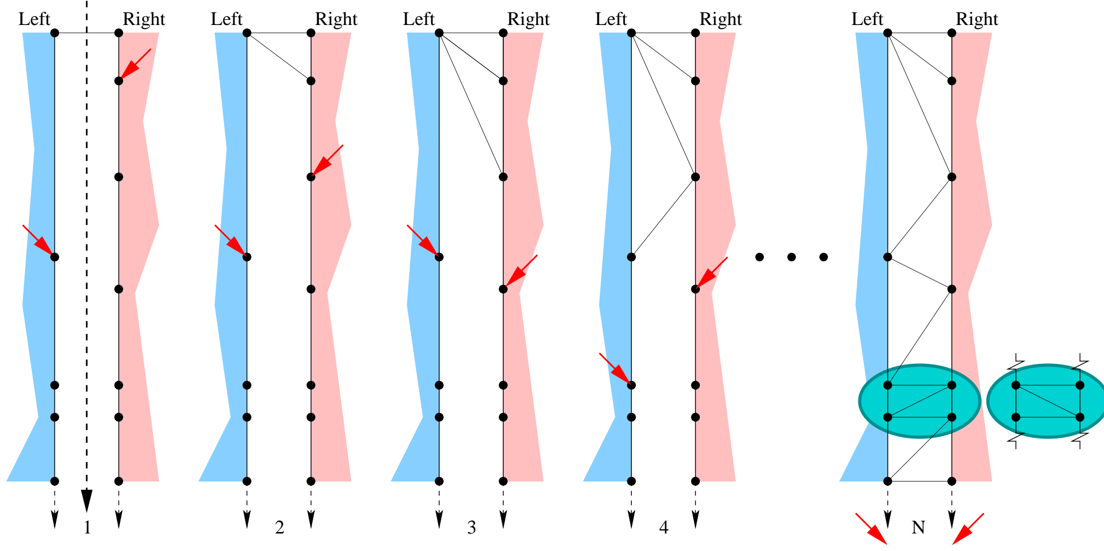


Figure 4.6: *Closing gaps at tile boundaries. We traverse the boundary of a left and a right tile. Two pointers (red arrows) are kept to indicate the next possible vertex on each side. Triangles are always inserted such that the "higher" vertex is connected using a triangle. To determine which vertex is higher, a common reference axis (dashed in step 1) is used. Note that ambiguities can arise (marked by the two cyan ellipses) depending on the implementation if two vertices share the same "height" with respect to reference axis. However, a consistent triangulation will always be generated.*

[Wil83].

$$\rho \quad := \quad \max\left\{ \sqrt{\sum_{i=1}^{3}\left(\frac{\partial v_i}{\partial s_1}\right)^2}, \sqrt{\sum_{i=1}^{3}\left(\frac{\partial v_i}{\partial s_2}\right)^2} \right\} \tag{4.3}$$

To compute ρ, $\boldsymbol{s}$ is expressed in parametric form $\boldsymbol{s}(\boldsymbol{v})$, which already includes perspective division and scaling of the canonic frustum to pixel coordinates. The Jacobi matrix of $\boldsymbol{s}(\boldsymbol{v})$ at $\boldsymbol{v}$ consists of the partial derivatives

$$J_{ij}(\boldsymbol{v}) \quad := \quad \frac{\partial s_i}{\partial v_j}. \tag{4.4}$$

The inverse transpose of $J(\boldsymbol{v})$ contains exactly the partial derivatives required to compute ρ as row vectors. Computing ρ yields the optimum scale corresponding to a screen-space error τ equal to 1 pixel. If the user selects a different screen-space error, the frustum is scaled to pixel coordinates divided by τ instead of using the entire resolution. Then, ρ is the object space distance that projects onto τ pixels. Note that

inverting the 3×3 matrix $J(\boldsymbol{v})$ can be performed highly efficient. Also note that the estimate obtained for ρ will be linear in s_3 (i.e., the depth), which can be proven by means of the theorem of intersecting lines, or by full expansion and differentiation of the perspective matrix.

On the CPU, a scale-of-detail ρ_k is computed per tile for each corner k of its bounding box. Because entries of the error vector are given by $\varepsilon_i = 2^{n-1-i}$ units, the optimum LOD value is computed by

$$\lambda_k \quad := \quad \lambda_{max} - \lfloor \log_2(\rho_k) \rfloor, \tag{4.5}$$

where $\lambda_{max} = n - 1$ is the number of available levels. The mesh $\mathcal{M}_{\min_k\{\lambda_k\}}$ is then selected for rendering the tile.

4.5.4 Geomorphing

As mentioned before, high-resolution displays coupled with a low screen-space error can require most of the terrain to be rendered at the highest-possible resolution. It can thus become necessary to increase the tolerable screen-space error in order to maintain stable and interactive frame rates. Geomorphs [FEKR90] are typically applied to prevent popping artifacts. For each vertex $\boldsymbol{x}$ in a tile, the LOD values λ_k at box corners are either tri-linearly interpolated in the vertex shader to get an approximate vertex LOD $\lambda(\boldsymbol{x})$, or the LOD calculations can be performed for each vertex[2]. Geomorphing now linearly interpolates values between successive levels in the hierarchy (assuming that the domain $\mathcal{D}$ of the heightfield is the x_0, x_1 plane):

$$\begin{aligned} \mathcal{H}_{\lambda(\boldsymbol{x})}(x_0, x_1) &= \operatorname{lerp}\left(\mathcal{H}_{\lfloor\lambda(\boldsymbol{x})\rfloor}(x_0, x_1), \mathcal{H}_{\lfloor\lambda(\boldsymbol{x})\rfloor+1}(x_0, x_1), \lambda(\boldsymbol{x}) - \lfloor\lambda(\boldsymbol{x})\rfloor\right), \text{ where} \\ \operatorname{lerp}(a, b, \alpha) &:= a + \alpha\,(b - a)\,. \end{aligned} \tag{4.6}$$

Finding the correct height values $\mathcal{H}_i$ to evaluate Equation (4.6) on the GPU could be implemented in a straightforward manner by using conditionals. However, since conditionals are costly on current GPUs[3], we avoid them by implementing a different approach based on clamped forward differences. In this approach, we treat height values $\{\mathcal{H}_i\}_{i=0}^{n-1}$ as the control points of a piecewise linear interpolant in λ. To obtain $\mathcal{H}(\lambda)$, we compute shifted basis-functions that can be reduced using simple dot-product

[2] The latter one is more accurate, but was not an option due to performance considerations in [SW06].

[3] Each conditional costs on the order of 5 cycles on an NVIDIA GeForce 6800 GT

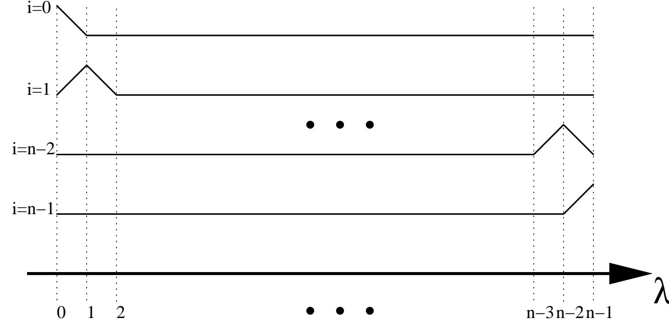


Figure 4.7: *Basis-functions $\eta_i'(\lambda)$ for geomorphs.*

arithmetic. Firstly, we compute a vector-valued function

$$\begin{aligned} \boldsymbol{\eta}(\lambda) &:= \operatorname{clamp}\left((\lambda,\lambda,\lambda,\lambda,\dots)^T - (0,1,2,3,\dots)^T, 0, 1\right), \text{ where} \\ \operatorname{clamp}(\boldsymbol{u},\alpha_0,\alpha_1) &:= \begin{pmatrix} \max(\min(u_0,\alpha_1),\alpha_0) \\ \dots \\ \max(\min(u_{n-1},\alpha_1),\alpha_0) \end{pmatrix} \wedge \; u \in \mathbb{R}^n. \end{aligned} \tag{4.7}$$

Each component η_i of $\boldsymbol{\eta}$ contains a linear ramp between $\lambda = i$ and $\lambda = i+1$. For $\lambda \le i$ it is 0, and for $\lambda \ge i+1$ it is 1. The desired basis function is then obtained by computing forward differences on $\boldsymbol{\eta}$:

$$\eta_i'(\lambda) := \begin{cases} 1 - \eta_0(\lambda), & \text{if } i = 0 \\ \eta_{i-1}(\lambda) - \eta_i(\lambda), & \text{otherwise} \end{cases} \tag{4.8}$$

Finally, $\boldsymbol{\eta'}$ contains the well-known basis functions for linear interpolation (see also Figure 4.7). Interpolation can now be written as the dot product $\mathcal{H}(\lambda) = \sum_{i=0}^{n-1} \eta_i'(\lambda)\mathcal{H}_i$. This method is highly efficient on the GPU and in our case ($n = 9$) outperformed the straight-forward implementation using conditionals by a factor of 2.5.

4.5.5 Texturing

By default, a pre-lit 2D texture is mapped onto the terrain. This can be a photo texture or, as for the Puget Sound, a synthesized 2D texture. During preprocessing, the texture

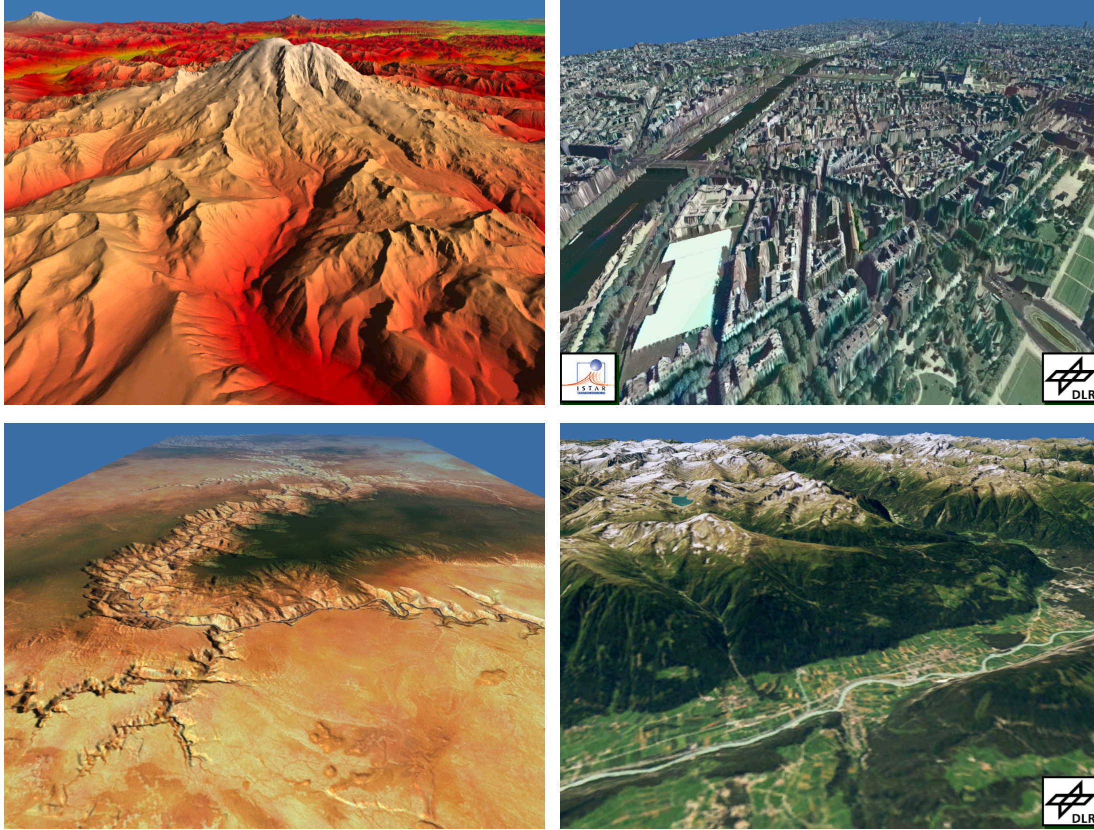


Figure 4.8: *Several data sets to test our terrain rendering algorithm. Top Left: Puget Sound, 16 K × 16 K samples (texture and geometry). Top Right: Paris, 9.7 K × 5.8 K geometry samples, 19.5 K × 11.7 K texture samples. Bottom Left: Grand Canyon, 4 K × 2 K geometry samples, 8 K × 4 K texture samples. Bottom Right: Alps, 8.9 K × 8.5 K samples (texture and geometry). Observe the high degree of geometric details present even in regions further away from the viewer. Puget Sound and Grand Canyon data courtesy of GA Tech. Paris and Alps data courtesy of DLR Oberpfaffenhofen (Deutsches Zentrum für Luft- und Raumfahrt) and ISTAR.*

is dyadically downsampled using a tensor product Lanczos filter

$$\begin{aligned} L_r(x) &:= \begin{cases} \mathrm{sinc}(r \cdot \pi / \rho) & \text{if } r \leq \rho,\, r \neq 0 \\ 1 & \text{if } r = 0 \\ 0 & \text{otherwise} \end{cases}, \\ L_r(x,y) &:= L_r(x) L_r(y) \end{aligned} \tag{4.9}$$

with radius $r = 2$ to obtain a single, large mipmap. Tiles are then cut out of the mipmap to precisely match the tiles of our mesh hierarchy. To save GPU memory and bandwidth, each texture tile is then compressed using the S3 compression scheme

[INH99]. More specific, tiles are encoded using the RGB-based DXT1 format, which yields good fidelity for most photographic or synthetic textures at a compression rate of 6:1. We store the $16\,K \times 16\,K$ Puget Sound texture including mipmap levels for the $16\,K \times 16\,K$ and $4\,K \times 4\,K$ geometries in about 170 MB.

If a pre-lit texture is not available, it is computed from the original terrain in a pre-process. Alternatively, normals can be stored as additional vertex attributes. However, besides the additional memory overhead that is introduced (at least two 8 bit values to cover the upper hemisphere), lighting artifacts due to non-continuous changes of normals during LOD transitions can only be resolved by storing one normal per vertex *and* level. Even then, artifacts cannot be fully avoided. For a further overview of the intricate problems that arise when using normals for even the simplest LOD hierarchies, we would like to refer to the recent paper by Han et al. [HSRG07]. On the other hand, a DXT1 pre-lit texture with an average of 4 texels per vertex has the same storage requirements as a single per-vertex normal (namely 16 bits), but it avoids any lighting artifacts because texture filtering is performed *after* lighting. Naturally, this comes at the cost that relighting cannot be trivially achieved while trying to maintain interactive frame rates.

4.6 Memory Management

After building the discrete LOD hierarchy, for high-resolution terrains including morph values and textures, the data is far too large to be stored in local video memory of recent GPUs. To avoid frequent paging of textures and vertex buffers, and to optimize progressive updates we have implemented a memory manager. At initialization time, the memory manager allocates chunks of exponentially growing sizes in GPU memory, to prevent external fragmentation. Sizes range from 32KB to a maximum size that allows the largest vertex buffer to be stored in such a chunk. Additionally, a number of textures with a fixed resolution is allocated. The memory manager stores meta-data for each memory block, i.e., its size, a time stamp, and the number of levels already sent to the GPU. Paging is now implemented as a mixture between a least recently used (LRU) and a tightest fit (TF) strategy.

Whenever a tile A is to be rendered, the system determines if there is already a chunk associated with A. If not, and also no appropriate chunk is available, the tile B with the earliest time stamp large enough to completely store A is determined. B is then marked as non-resident, and the chunk is overwritten with the data of A. To efficiently determine B, we keep a priority list for each available size. This allows us

to weight the LRU strategy against a TF criterion. Once a chunk has been associated with A, all data required to render the current level is sent to the GPU. If there already was a chunk associated with A, the memory manager determines whether the chunk contains all necessary data. If not, the CPU sends all missing vertices and the required index buffer to the GPU. Since vertices are shared across levels, this update is usually very cheap compared to the upload of all vertices. Whenever a tile is rendered, its time stamp is updated.

The memory manager supports uniform load on the bus connecting the CPU and the GPU, thus avoiding "paging hiccups". When a non-resident tile enters the view frustum, there is usually another one that has to be released, the texture tile has to be uploaded, and an initial LOD has to be sent to the GPU. However, with high probability this initial LOD requires only a few vertices. On the other hand, if a tile was already resident, performing an update only requires a fraction of the entire data to be sent.

Speculative prefetches are also supported, if there are unused memory chunks. If the number of chunks needed to render the current view falls below a certain fraction of all allocated chunks, the user's view is predicted. Whenever the user moves, a list containing the last viewing parameters is updated. By fitting a spline through these parameters, a set of few, discrete viewing parameters can be extrapolated and tiles that are predicted to become visible in the near future can be prefetched, as long as a maximum time budget is not expired. In this way, very smooth fly-overs at high frame rates can be achieved.

4.7 Results

Our results and timings are summarized in Table 4.1. All benchmarks were performed on an Intel P4 3.0 GHz with 2 GB RAM and an NVIDIA GeForce 6800 GT with 256 MB video RAM. The machine was equipped with a single, standard 120 GB IDE hard disk. All data sets were rendered to a 1024×768 view port. Enabling $8\times$ full-screen anti-aliasing and $4\times$ anisotropic texture supersampling resulted in a performance drop of about 30%. The timings should be fairly comparable to previous publications. Although we utilized a graphics card that is newer compared to the hardware used in previous publications, we also render a considerably larger view port.

Preprocessing of the geometry to a 9 level hierarchy achieves a throughput of approximately 15 million vertices per minute and is linear in the amount of vertices. Memory consumption is constant, as tiles are processed independently of each other. Generating a $16\,\mathrm{K} \times 16\,\mathrm{K}$ texture hierarchy including filtering takes about 5 minutes.

Table 4.1: *Timings and results of our terrain renderer. Original size only includes height field and texture, without taking mipmaps into account. τ refers to the pixel error. For $\tau = 1$ geomorphs were disabled, for $\tau = 5$ they were enabled.*

Data Set	Resolution	Texture	original Size	Storage	fps $\tau = 1$	MΔ/sec $\tau = 1$	fps $\tau = 5$
Puget4K	$4\,K \times 4\,K$	$16\,K \times 16\,K$	800 MB	412 MB	202	78.85	199
Puget16K	$16\,K \times 16\,K$	$16\,K \times 16\,K$	1.25 GB	1.25 GB	60	25.69	57
Grand Canyon	$4\,K \times 2\,K$	$8\,K \times 4\,K$	112 MB	80 MB	289	74.60	292
Paris	$9.7\,K \times 5.8\,K$	$19.5\,K \times 11.7\,K$	763 MB	267 MB	36	100.87	65
Alps	$8.9\,K \times 8.5\,K$	$8.9\,K \times 8.5\,K$	361 MB	546 MB	145	65.43	155

The Puget Sound ($4\,K \times 4\,K$ geometry) and the Grand Canyon data sets are only medium sized, and consequently our system is neither triangle nor memory limited. For the Paris data set our method becomes triangle limited. The reason is that our meshing strategy faithfully reconstruct all the steep sides of buildings ("curtains"). This results in up to 2.8 million triangles that have to be rendered per frame. A lot of these triangles are backfaces that are culled by OpenGL (but they are still counted since they pass the geometry stage). On the other hand, the Paris dataset is an excellent benchmark for the raw triangle throughput that our system can achieve.

The Puget Sound ($16\,K \times 16\,K$ geometry) data set on the other hand is large enough to demonstrate the effects of the memory system. The lower triangle throughput reflects that our paging strategy does not come for free, but it still allows for highly interactive fly-overs.

The Alps data set is a good mixture between these extremes. It contains lots of flat terrain around Munich and a considerable amount of very rough terrain around the Alps.

As our results show, frame rates for highly triangulated data sets, such as Paris, can also be improved by increasing the pixel error and enabling geomorphing. For these highly triangulated datasets we also hope to benefit from continuously increasing vertex processor throughput on future graphics chips.

4.8 Summary

We have presented an efficient rendering system for large, textured terrain data that provides excellent quality and highly detailed views. In particular, at equal frame rates our system guarantees a smaller pixel error than previous approaches. We achieve these properties by exploiting a special discrete LOD hierarchy, as well as processing and rendering functionality on recent GPUs.

Among interesting continuative research directions was—as of writing of the original paper—the question wether efficient, GPU-based compression schemes for large-scale geometry could be devised. As of writing of this thesis, a very recent publication discussing this issue is the paper by Dick et al. [DSW08]. Challenges that remain to be solved include to find efficient texture compression schemes, and to remove backfaces even before they enter the visualization pipe.

Another interesting direction is to synthesize textures on the fly. This would—at least for some scenarios—solve the issue of memory-heavy textures, as well as the purely geometry-based lighting problems in the presence of LOD hierarchies. One such approach will be outlined in the next chapter, although in a slightly different context.

Chapter 5

Fractal Terrain

In the previous chapter we showed that recent advances in algorithms and graphics hardware have opened the possibility to render large digital elevation models at interactive rates on commodity PCs. Due to these advances it is also possible to synthesize artificial terrains interactively by using procedural descriptions. This chapter describes a GPU method for real-time editing, synthesis, and rendering of infinite landscapes. These landscapes exhibit a wide range of geological structures that are remarkably close to granite formations found in reality. To render such infinite landscapes, we build upon the concept of *projected grids* [HNC02, Joh04] to achieve a near-optimal sampling of the landscape. The landscape is described as a procedural multifractal that permits efficient, point-wise evaluation. This description is realized using shaders in the OpenGL API.

Furthermore, we propose intuitive editing metaphors to change the shape of the resulting terrain interactively. The method is multi-scale and adaptive in nature, resulting in a very natural level-of-detail formulation. We extend the concept of projected grids to spherical domains in order to achieve the impression of a whole, fractal planet. Especially in combination with the synthesis of geo-typical textures that automatically adapt themselves to the shape being synthesized, a powerful method for the creation and rendering of realistic landscapes is presented.

An additional beneficial side effect is that the parameters required to synthesize each landscape occupy only very small amounts of memory, which makes the method feasible for scenarios that need to transmit such landscapes across narrow communication channels. Examples for this include terrain-based multiplayer games in which a server needs to provide hundreds or thousands of clients with a consistent description of the game level. If such a game is to be realized purely internet-based, it is highly advantageous to keep the amount of data to be transmitted before each game starts as

small as possible. Even worse, if players are allowed to modify the landscape during gameplay, the transmission of a full vertex-based description is not feasible at all.

5.1 Related Work

Based on the ideas of the organizing principles of natural phenomena [Man83], there exists at least empirical evidence on fractals in geological structures such as landscapes, rivers, and coastlines. Such structures obey the typical fractional Brownian or $1/f^{\beta}$ "motion", and they can thus be modelled as a random walk exhibiting certain stochastic properties. Based on this observation a number of different approaches for fractal terrain synthesis have been suggested over the last decades. For a thorough overview we refer to [EMP+03]. Today, fractal models are usually generated using Fourier filtering [Sak93], midpoint displacement [Mil86], or noise synthesis [Per85]. Since the approach described here is closely related to noise synthesis, we will briefly describe the underlying concept.

Perlin [Per85, PH89] introduced a synthetic, functional fractal model as a summation of several, appropriately scaled-down copies of a stochastic *noise* function with a limited frequency bandwidth. Much effort has been spent in order to make the noise function stationary, which includes invariance under translation and rotation [Per02]. If the noise function is not stationary, undesired artifacts might arise if the non-stationary properties of the noise cannot be controlled properly. The *Rescale-and-Add* method by Saupe [Sau88, Sau89] extends the heuristic formulas of Perlin to a full fractal model. It allows a multi-dimensional fractal noise with locally varying fractal dimension (a measure of the surface roughness) to be created by summing Perlin noise functions at different scales and frequencies. As a special case, fractal heightfields can be generated by displacing a two-dimensional basis domain. At the same time as Perlin, Musgrave [MKM89] introduced *noise synthesis* to control the fractal dimension and other parameters of the synthesis process. Fractals obtained by this type of synthesis are generally referred to as *multifractals* because they use multiple scalings to control the statistic properties of the underlying random process locally.

The advantages of the functional approach are:

- The function can be locally evaluated at only those locations required for rendering.
- The level of detail of the terrain to be generated can be adapted to match the local image resolution, thereby anti-aliasing the fractal [Pea].
- All parameters of the fractal may vary locally over the object domain.
- The basis functions of the fractal synthesis can be modulated to adapt locally to particular design features.
- The procedural evaluation of the terrain can be performed efficiently on programmable graphics hardware [Har01, Gre05].

Due to the high degree of sophistication to which functional approaches for fractal modelling have matured, it is by now possible to create landscapes with impressive realism, including many different geological formations as well as terrain-specific textures [Cor01, Pan]. Although fractal terrain synthesis is recognized as a fine technique, the most demanding part is to deal with the "parametric nightmare" of the synthesizer's high-dimensional feature space. So far, fractal landscape editors either restrict the designer's flexibility or overwhelm the user with a plethora of parameters. An embedding of these parameters into a tight visual feedback loop such that users can directly monitor all effects of their editing actions is not yet available. Such an embedding allows to simulate many kinds of different landscapes in an intuitive way, and it helps the designer to model the scenery after her or his expectations. At the same time both the design process and the time to market can considerably be shortened, which is of particular interest in applications like computer games or computer-animated films.

Techniques open to this kind of interaction can be directly integrated into virtual environments to continuously modify outdoor scenarios based on context. By modifying landscapes depending on external parameters like game status, player's expertise, or difficulty, the immersiveness of such scenarios can be increased significantly. Similarly to the *Populous* series of games [Mol89], such techniques furthermore allow users to create their own synthetic worlds and to alter them directly at run-time by individual and context-specific guidelines. Since it is impossible to pre-compute the per se infinite variety of different structures, parameterized methods for description and evaluation of fractal landscapes that permit continuous transitions between different formings (parameter sets) have to be considered.

There are at least two reasons why the aforementioned scenario has not yet become reality. Firstly, terrain generation methods that feature arbitrary local control of surface properties are generally too expensive to be used for interactive synthesis of high-resolution terrains. Secondly, the creation of synthetic images of *dynamic* terrain models consumes a lot of memory and computation time. Hence, it is not suited for use in interactive environments without further effort. For instance, ray-tracing of procedural fractal height values [Mus03, Pan] is far from being interactive. But even for non-procedural terrains, ray-tracing and scanline algorithms [CGG+03a, DWS+97, SW06, DSW08] require *static* height fields to harness their full potential (see also Chapter 4). In the context of fractal synthesis, the requirement of static height fields means that the landscape has to be synthesized up to the highest resolution required *locally*, severely limiting the use of high-resolution models due to memory constraints. Even worse, for large models residing in host memory, the performance of naïve hardware-accelerated scanline algorithms quickly becomes compromised by the limited bandwidth of the graphics bus, since such algorithms involve streaming renderable primitives to the GPU.

5.2 Contribution

In this chapter, we present a real-time fractal terrain synthesizer. The algorithms described here are designed with respect to the aforementioned requirements. They are combined into a WYSIWYG ("What-You-See-Is-What-You-Get") interface to allow the intuitive design of highly detailed terrain models. Our approach involves two distinct procedures: *editing* and *rendering*. The synthesis of the landscape is integrated directly into the rendering procedure. This avoids any intermediate data structures for storing height values—especially on the CPU side—other than the ones needed to form the final image. Editing and rendering involve a number of novel techniques and they provide many features not available in previous methods.

Editing. Because the user interacts directly with the same representation used to form the final image, the fractal's parameter space can be managed conveniently. We use painting and brushing on gray-scale images as editing operations, where the gray-scale images represent the fractal's *auxiliary functions*, sometimes also referred to as fractal basis functions. Against common knowledge, editing auxiliary functions can be highly intuitive, but only if coupled with immediate visual feedback. This approach turns out

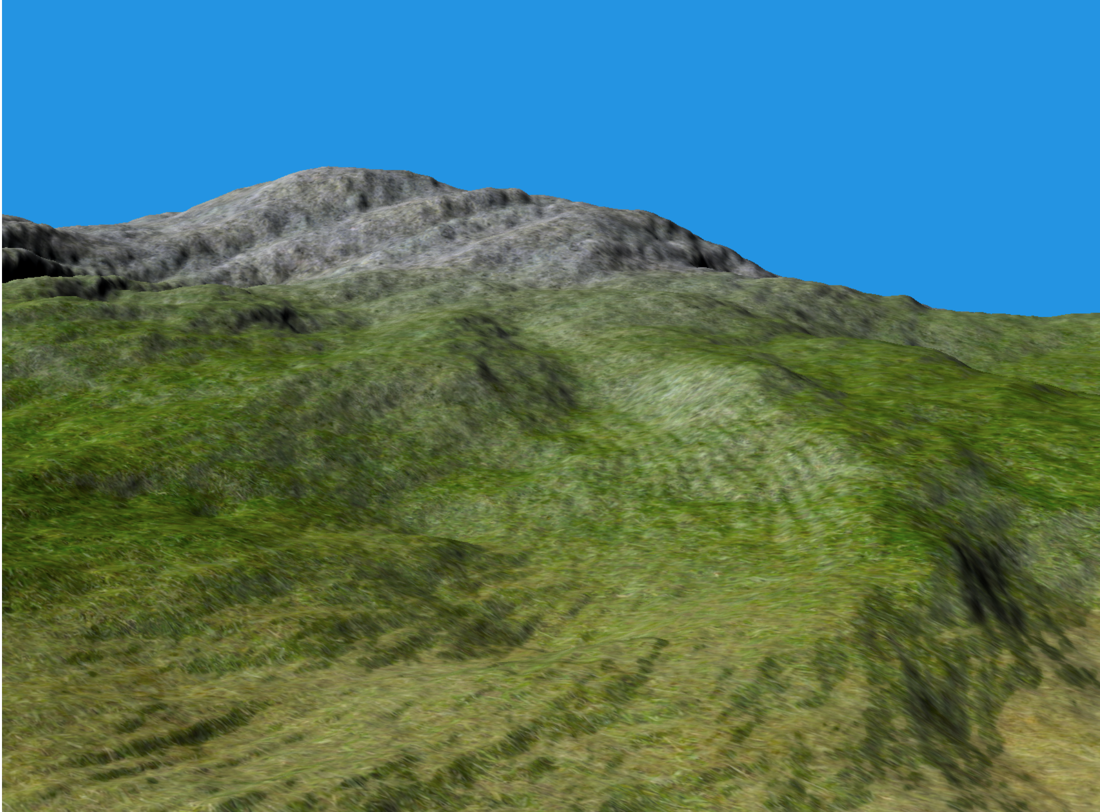


Figure 5.1: *A fractal landscape generated by our method. Note the striking geological features and geo-typical textures. On recent GPUs, synthesis and rendering on a* 1280×1024 *viewport runs at 70 frames per second.*

to be an amazingly powerful paradigm enabling multi-resolution terrain editing via the multiple frequency bands of auxiliary functions.

Rendering. We exploit the functional nature of the terrain, which allows for point-wise synthesis at arbitrary positions. Rather than performing the synthesis as a distinct and decoupled procedure, it is tightly integrated into the rendering process. The advantage is that both steps can be implemented on the GPU, thereby exploiting parallelism and memory bandwith while at the same time limiting any bus transfers to compact and local editing updates. We utilize a projected grid approach to minimize both the number of point evaluations and to achieve near-optimal sampling. In a nutshell, a screen-space aligned grid is first projected into the fractal's base domain. Then, points of the base-domain grid are displaced onto the fractal's surface. Anti-aliasing is performed for each

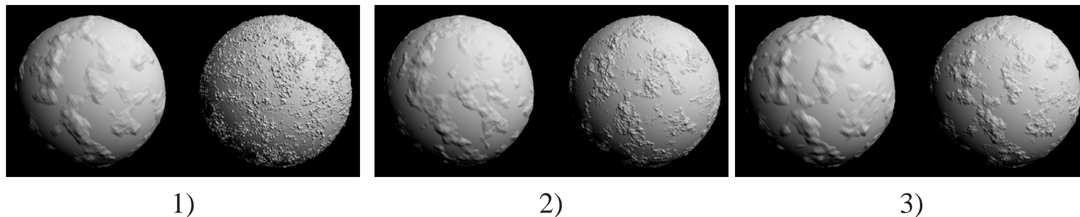
point by computing the appropriate level-of-detail. To further increase the landscape's realism, geo-typical materials are synthesized *by example* (see also Figure 5.1). For each point of the projected grid, so-called *proto-textures* [Cor01] are combined, either by using a slope/height parameterized weighting function [Dac05], or by using editable weights.

The remainder of this chapter is organized as follows. In Section 5.3 we describe the theory behind noise synthesis including the domain warping employed to improve the visual appeal of the generated terrains. We then present the WYSIWYG interface used for fractal editing in Section 5.4. In Section 5.5 we discuss synthesis and rendering of the terrain on the GPU. Finally, we point out directions for future work.

5.3 Multifractal Terrain Synthesis

In the following we will briefly review the theoretical basis behind noise synthesis and stochastic fractals. The functional approach described here is suited well for evaluation on recent GPUs, since it only requires simple arithmetic operations and locally contiguous texture access.

5.3.1 The Rescale-and-Add Method



1) 2) 3)

Figure 5.2: *Noise synthesis using varying parameters. The three image pairs illustrate the meaning of 1) varying lacunarity, 2) Hurst exponent, and 3) the number of octaves in Equation* (5.1). *The respective values are larger in the right half of each image.*

A 2D grid of $N(0,1)$ Gauss-distributed random numbers with an average of zero and variance equal to one, called the *noise lattice*, is generated and stored in a 2D texture map. This discrete lattice is then extended to a C^1 continuous, periodic function over an infinite domain by means of bilinear interpolation and texture repetition on the GPU. This function is either called noise [Per85] or, more accurately, *auxiliary function* $S(\boldsymbol{x})$ [Sau88]. A 2D fractal is generated by superposition of several appropriately scaled

copies of the auxiliary function.

$$\mathcal{H}(\boldsymbol{x}) = \sum_{k=k_0}^{k_1} \frac{1}{r^{kH}} S(r^k \boldsymbol{x}) \tag{5.1}$$

Equation (5.1) can be evaluated for any arbitrary position $\boldsymbol{x}$ of the domain independently of its neighbors. Consequently, it can be computed efficiently in a pixel shader that is parameterized by the domain coordinates $\boldsymbol{x}$, the lacunarity r, and the Hurst exponent $H = 3 - D$, where D is the fractal dimension of the surface. Equipped with these parameters, the shader computes fractal height values using multiple evaluations (texture fetches) of the noise lattice. The meaning of the different parameters is illustrated in Figure 5.2.

The summation limits are calculated in accordance with the smallest and largest structures (frequencies) desired at a certain position of the terrain. k_0 determines the global fractal structure and is typically computed only once. k_1, however, defines the smallest visible level of detail, which corresponds to the amount of high-frequency information in the final image. Consequently, it is usually changed from point to point depending on the perspective distortion. In Section 5.5 we will explicitly discuss how to select k_1 in order to avoid aliasing artifacts.

5.3.2 Domain Warping

The Rescale-and-Add method creates convincing terrain fields on medium scales. On larger scales, however, the result is too homogeneous to pass as realistic. The reason is that the auxiliary function $S(\boldsymbol{x})$ is designed to be stochastically invariant under rotation and translation. This results in very regular and artificially looking geological structures.

A more natural appearance can be achieved by using *domain warping*. Instead of synthesizing the terrain across the domain $\mathcal{D} \subseteq \mathbb{R}^2$, a continuous re-parametrization $\Phi : \mathcal{D} \mapsto \mathcal{D}$ is utilized, on which the synthesis is then performed. In [EMP+03] a similar mapping was described to simulate breaking waves on a 2D-only domain embedded in 3D. To overcome the homogeneous structure of the terrain we suggest roughness- and height-dependent rotations and translations of the domain. These operations break up the stochastic invariances of the auxiliary functions in a controlled and restricted way.

We assume that each domain coordinate $\boldsymbol{x} = (x_1, x_2)^T$ is transformed by a general rotation/translation matrix, where ϕ is the angle of rotation and $\boldsymbol{t} = (t_1, t_2)^T$ is the

translation vector.

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi & t_1 \\ \sin\phi & \cos\phi & t_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \tag{5.2}$$

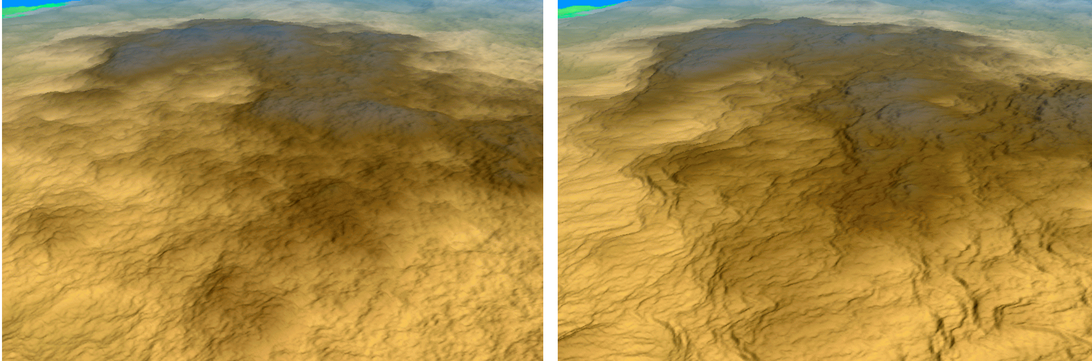


Figure 5.3: *The effect of domain warping. A simple noise function was used for the auxiliary function. Left: Without domain warping the terrain looks rather homogeneous and uniform. Right: By using domain warping, curved features like weathered lava are generated.*

Figure 5.4: *Real world granite structures. These photographs were taken in the Masthugget district of Gothenburg, Sweden. The rock formations depicted can be found throughout the granite cliffs of southern Sweden. The results that can be achieved with the domain warping presented in this chapter can be remarkably similar (also see Figure 5.1).*

To avoid evaluations of trigonometric functions, we represent the rotation by a unit vector $\boldsymbol{\rho} = (\cos\phi, \sin\phi)$, resulting in

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \rho_1 & -\rho_2 & t_1 \\ \rho_2 & \rho_1 & t_2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix}. \tag{5.3}$$

Thus, any arbitrary selection of $\boldsymbol{\rho}$ results in a proper rotation, as long as $\|\boldsymbol{\rho}\|_2 = 1$. Observing that geological structures and formations in nature vary depending on surface height [MKM89], we introduce a height-dependent roughness

$$
\begin{aligned}
\mathcal{R}_i &= \mathcal{R}_0 \mathcal{H}_i, \quad \text{with} \\
\mathcal{R}_0 &:= \frac{1}{rH}, \quad \text{and} \\
\mathcal{H}_i &= \sum_{k=k_0}^{k_0+i-1} \frac{1}{r^{kH}} S(r^k \boldsymbol{x}).
\end{aligned} \tag{5.4}
$$

Here, $\mathcal{H}_i$ denotes the height of a surface point after evaluation of the first i terms of Equation (5.1), i.e., the height after accumulating i octaves.To add domain warping to the synthesis process we apply a different mapping depending on $\mathcal{R}_i$ for every octave i:

$$
\begin{aligned}
\boldsymbol{\rho}^{(i+1)} &= \boldsymbol{c_1} \cdot \mathcal{R}_i + \boldsymbol{c_2} + \boldsymbol{\rho}^{(i)} \\
\boldsymbol{t_{i+1}} &= \boldsymbol{c_3} \cdot \frac{\mathcal{H}_i}{\mathcal{R}_i}
\end{aligned} \tag{5.5}
$$

The rotation vector $\boldsymbol{\rho}^{(i+1)}$ is re-normalized after each iteration. To perform the above updates, three additional constant 2D vectors $\boldsymbol{c_1}$, $\boldsymbol{c_2}$, and $\boldsymbol{c_3}$ are introduced. $\boldsymbol{c_1}$ controls the amount of rotation determined by the current roughness, while $\boldsymbol{c_2}$ performs a constant update. The third vector $\boldsymbol{c_3}$ controls the influence of the current height and roughness on the translation. The basic idea is that the amount of rotation increases with roughness, resulting in turbulent structures, similar to cooled-down lava, while stretching regions that are either smoother or at higher altitudes significantly more in order to make them appear *washed out*. Two examples that show the effects of the domain warping described here are depicted in Figure 5.3. These examples were generated using the following selection of parameters.

$$
\begin{aligned}
\boldsymbol{c_1} &:= (0.35, 0.16)^T, \\
\boldsymbol{c_2} &:= (-0.07, 0.13)^T, \\
\boldsymbol{c_3} &:= (0.11, 0.17)^T, \\
\boldsymbol{\rho}^{(0)} &:= (1, 0)^T, \quad \text{and} \\
\boldsymbol{t_0} &:= (0, 0)^T.
\end{aligned} \tag{5.6}
$$

Different formations can be achieved by modifying the control vectors $\boldsymbol{c_i}$ using the fractal editor described in the next chapter.

Even though this procedure comes at the expense of evaluating more parameters in the innermost loop of the synthesizer, it effectively breaks up the homogeneity of the terrain, since it adds organic shapes that resemble natural geo-evolution, such as depicted in Figure 5.4.

5.4 Interactive Fractal Editing

An important feature of the proposed system is the fractal editor. In the spirit of the WYSIWYG paradigm, the editor provides the user with immediate visual feedback for each action. The user can literally paint auxiliary functions represented by gray-scale images. Several of these gray-scale images $I_i : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ are used to replace the aforementioned *noise lattice*. These images are then composed into the auxiliary function $S(\boldsymbol{x})$ (see Equation (5.1)) by assigning individual weight functions $w_i : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ to them. The auxiliary function can then be reconstructed on a discrete lattice simply by computing a weighted sum

$$S(\boldsymbol{x}) = \sum_i w_i(\boldsymbol{x}) I_i(\boldsymbol{x}). \tag{5.7}$$

It is important to note that the weighting functions w_i are re-normalized such that they sum up to 1 at each lattice point. This can be done conveniently on the GPU after each editing command. Thus $S(\boldsymbol{x})$ is a convex combination of auxiliary functions which allows to achieve a particular look of the terrain, e.g., 30% desert and 70% craters, straightforwardly. To offer maximum flexibility, the user can also choose to load images from disk to serve as auxiliary functions and/or weights. This keeps the interface intuitive and simple, while at the same time offering the possibility to utilize any painting or imaging program. In addition to standard painting tools, a series of image filters such as low-pass, high-pass, normalization, and equalization is implemented.

Internally, all auxiliary functions and weights are stored in 2D textures. Exploiting the fact that textures may comprise up to four channels, and that auxiliary functions and weights will always be fetched at the same position in order to compute Equations (5.1) and (5.7), weights and auxiliary functions are packed together in groups of four to minimize fetches from different textures. This allows the efficient GPU-based implementation of the various editing features and filters. Also, bus transfer between CPU and GPU is largely avoided, thereby allowing the rapid visual feedback needed to design new virtual worlds quickly.

Subsequently we will refer to this part of the editor as the *fine-scale synthesizer*

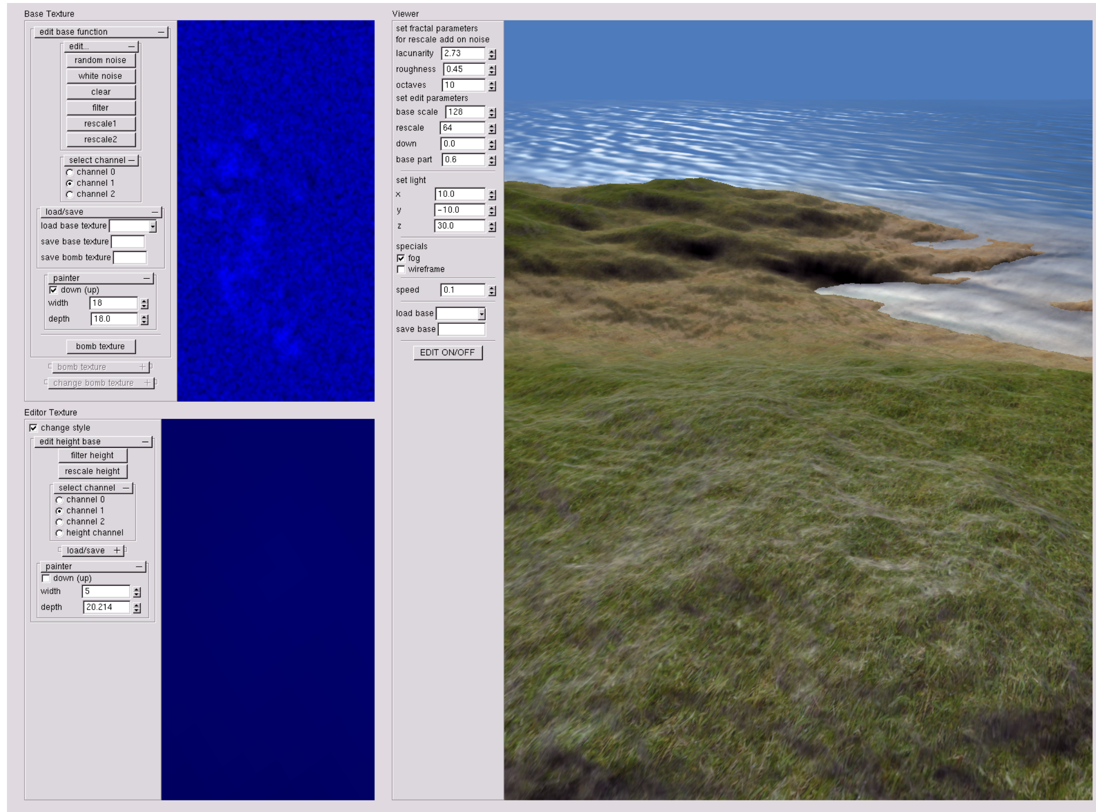


Figure 5.5: *The WYSIWYG interface of the fractal landscape editor. A a fine-scale input texture (top-left), a low-frequency sketch-pad (bottom-left), and the final editing result using proto-textures (right) is shown.*

because it controls the fine-grain *look* of the fractal terrain. Note that this leads to a particular look that is encountered across the entire terrain. Infinite terrains can theoretically be achieved simply by repeating the texture over the entire 2D base domain. However, the texture must be periodic (also known as *tile-able*) in order to avoid artifacts at the boundaries. In our system, periodicity is guaranteed by offering only painting operations that wrap around the image in two dimensions, i.e., the user essentially paints on a toroidal domain. Should the user choose to load an image which is not tiling, the lacunarity r and the domain warping can be adjusted easily in order to hide the otherwise apparent artifacts.

To decouple the fine-scale appearance from the macroscopic base shape, i.e., the general appearance of structures from their positions, a low-frequency fractal height field is added to the structures being generated by the fine-scale synthesizer. Since base-level and detail are largely uncorrelated, the designer can quickly develop a prototype

of the landscape by roughly sketching mountains, valleys, seas, and oceans on a coarse-resolution base domain. The final look-and-feel of the landscape is then modelled by means of the fine-scale synthesizer.

The benefits of this approach based on the manipulation of auxiliary functions are obvious: Interacting with gray-scale images representing weights or heights is highly intuitive and permits various external tools to be used seamlessly in order to rapidly achieve the desired look. Furthermore, all images needed to describe a fractal planet can be compressed using standard image compression schemes. This could become interesting in the near future, since 3D gaming becomes ubiquitous [Pul05], yet traditionally handheld devices feature only narrow communication channels.

The editing system also provides control over fractal parameters such as roughness, lacunarity, and water level, as well as over the vectors $\boldsymbol{c}_i$ that steer domain warping. Due to the intuitive use of all editing options and the direct visual feedback, user experiments have shown that persons not familiar with the editor achieve very convincing results within only a few minutes. In Figure 5.5 a snapshot of a typical editing session is shown, including a fine-scale input texture (top-left), a low-frequency sketch-pad (bottom-left), and the final editing result (right).

5.5 Rendering

The renderer evaluates the fractal height field procedurally for the visible fraction of the terrain only. For each vertex in the view port, Equation (5.1) and Equation (5.7) are evaluated with respect to the input textures and fractal parameters specified by the user. The entire rendering procedure is performed by means of GPU shader programs. To determine the vertices within the view port, a projected grid [HNC02, Joh04] is utilized and extended to allow rendering of spherical domains.

5.5.1 Projected Grid and LOD

The basic idea of the projected grid is as follows.

1. Start with an uniform grid in screen space.
2. Project this grid onto the landscape's base domain.
3. Evaluate the height at the respective domain coordinate and displace the grid, optionally evaluating domain warping.
4. Render the displaced grid.

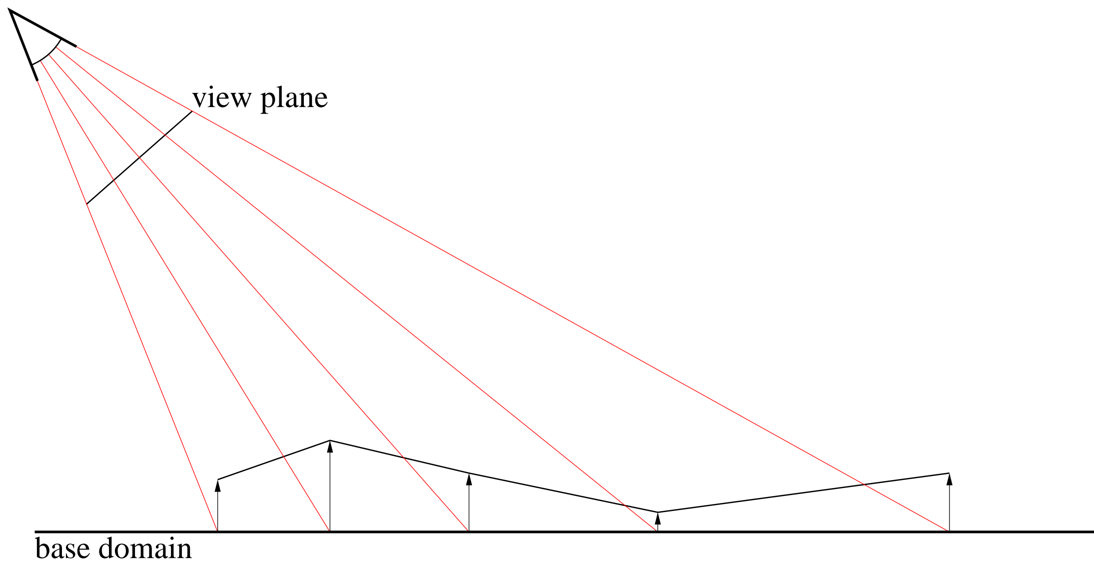


Figure 5.6: *Projected Grid concept. The basic idea behind the projected grid is to start with a regular grid in screen space, to project this grid onto the landscape's base domain, and to displace the vertices of this grid out of the base domain according to the height values.*

This method has some beneficial properties. Firstly, the projected grid tries to optimize object space triangles such that they project to approximately the same area in screen space. Secondly, since the topology of the grid is static, the grid projected to the base domain can be cached on the GPU as long as camera parameters are not altered. Since no topologic restrictions apply on a per-vertex basis, vertices can be processed independently of each other. Thirdly, the amount of triangles and thus the amount of workload on both the CPU and GPU is known a priori, and can be easily adapted to the available processing power. However, there are also some drawbacks. The computation of normals on the projected grid is more involved than on a regular grid. The grid also has to be extended slightly beyond visible vertices to avoid holes at the view port's boundaries. In order to circumvent the latter problem, the maximum height of the terrain needs to be known a priori (see Figure 5.7). Then a second, so-called *projector frustum* is used, obeying a soft constraint to match the camera frustum as close as possible. However, it may diverge from the camera frustum for aesthetic reasons, i.e., to prevent artifacts that would otherwise occur at grazing camera angles. For more details we refer to the work of Johanson [Joh04]. Last but not least, it should be noted that animations require a rather high-resolution projected grid, since a point-wise evaluation of the underlying terrain might otherwise lead to artifacts. Especially if domain warping is activated, the user can introduce additional high-frequency content in a rather unpredictable manner.

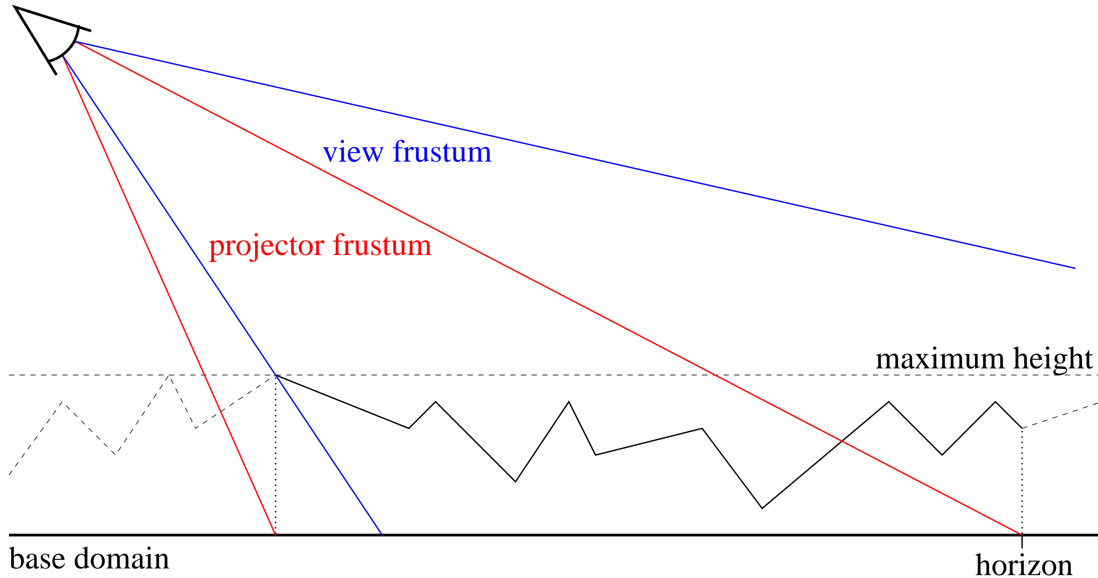


Figure 5.7: *Caveats of the projected grid. If using a projected grid for rendering, the maximum height of the terrain has to be known a priori. Otherwise, features might be missed. The projector frustum is then required to include each point in the base domain* ***potentially*** *contributing to the final image.*

In this case, either the resolution of the projected grid has to be increased up to the point where pixel-sized triangles are generated, or the user has to specify a LOD-bias in order to avoid such undesired effects.

For each vertex of the projected grid, a level-of-detail can be computed by projecting neighboring vertices into the base domain. Since the grid is regular in screen space, only the grid spacing is necessary to obtain an estimate of the local object-space grid spacing δ. It is worth noting that we do not perform anisotropic anti-aliasing due to the costs implied by methods such as texture footprint assembly [SKS96, OMD01]. Also note that certain standard algorithms such as summed area tables [Cro84] or RIP maps [LS93] are not an option since the landscape is never known a priori but synthesized on the fly. The number of octaves λ required to evaluate Equation (5.1) is then obtained by

$$\lambda = \frac{\log \delta}{H \cdot \log r}. \tag{5.8}$$

Here, λ is just the logarithm to the base r^H, i.e., the constant quotient between two consecutive amplitudes in the Rescale-and-Add method. Since λ is generally not an integer value, we interpolate linearly between $\lfloor \lambda \rfloor$ and $\lceil \lambda \rceil$ octaves. To do so, $\lfloor \lambda \rfloor$ octaves are

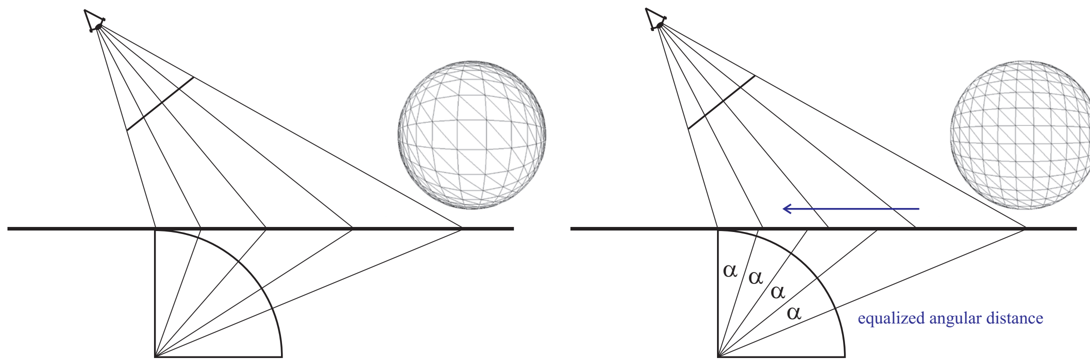


Figure 5.8: *Extending the projected grid to spherical domains. Left: A simple mapping of the planar base domain to a sphere results in anisotropic sampling patterns. Right: Samples are moved closer to each other such that they cover equal angular distances with respect to the sphere's center.*

summed up during synthesis and the next octave weighted by the fractional part $\lambda - \lfloor \lambda \rfloor$ is added. Thus, geomorphing [FEKR90, RHSS98] is virtually for free.

To perform fractal terrain synthesis over a spherical domain, the planar basis domain is warped around a sphere *after* the vertices have been projected. This is illustrated in Figure 5.8. To avoid anisotropic sampling around the sphere two concepts are used. The first is to introduce an artificial horizon behind which no landscape will be synthesized. The rationale is that atmospheric or fogging effects typically limit the viewing distance behind a certain point. This approach offers an intuitive quality versus performance tradeoff and is traditionally used in games and interactive environments. The second idea is to move samples closer together towards the viewer, with the goal that neighboring grid cells cover the same angular distance with respect to the sphere's center (see Figure 5.8). Since each vertex already stores its relative, pre-projective grid position, no further information is needed. If the viewer moves close to the surface, we switch back to the conventional projected grid.

The respective variant is evaluated in the fragment shader, just before the synthesis of the terrain takes place. After the vertices have been generated in the base domain, the fractal's height is computed for each position and the displaced vertex is stored in an intermediate vertex texture for future rendering.

Once the vertex texture describing the height field has been computed, additional properties for the rendering process are derived in a second pass. This includes normals for lighting as well as the slope (i.e., normal magnitude) and the water depth. These

properties are necessary for the proto-texturing described in the next section. The water depth is defined by a global water level specified by the user. All properties are then stored in textures with floating point precision.

5.5.2 Proto-Texturing

Proto-texturing is a common method to generate geo-typical textures [Cor01]. The idea is to use a set of textures that serve as prototypes for the simulation of real materials, e.g., grass, sand, rock, snow, etc. A height- and slope-dependent weighting function is typically utilized to blend the textures together. In this way, many of the textural variations found in nature can be reproduced. On the other hand, if proto-textures are used and the user moves further away from the height field, artifacts emerge due to the use of periodically repeated textures.

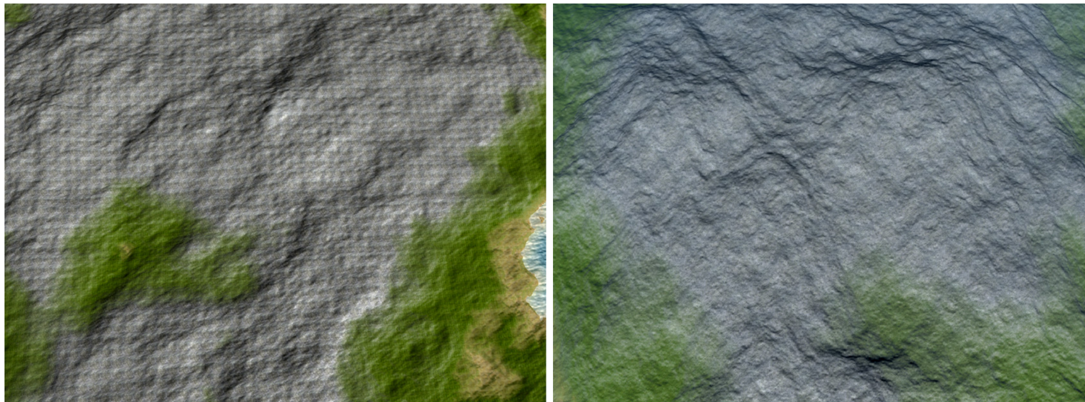


Figure 5.9: *Reducing the variance of the texture through adapting mipmap levels. Left: Normal mipmap using a Lanczos filter for downsampling. Right: mipmap with reduced variances.*

To avoid this effect, Dachsbacher et al. [Dac05] proposed to use color information only and to synthesize the texture procedurally at each fragment of the landscape during rendering. The intrinsically complex shader is amortized by caching parts of the results on the GPU. We suggest a different strategy based on the observation that the periodic structures are essentially caused by the variance of the color values in the proto-textures. Consequently we build a custom mipmap such that the variance is continuously decreased with each level. This can be accomplished by first applying the smoothing filter as usually in order to compute mipmaps. Then, we take a weighted average between the smoothed image and the global mean of the texture's colors. The user can control the process by selecting suitable filters and by providing a weighting parameter to affect the variance distribution across the levels. By exploiting log-step

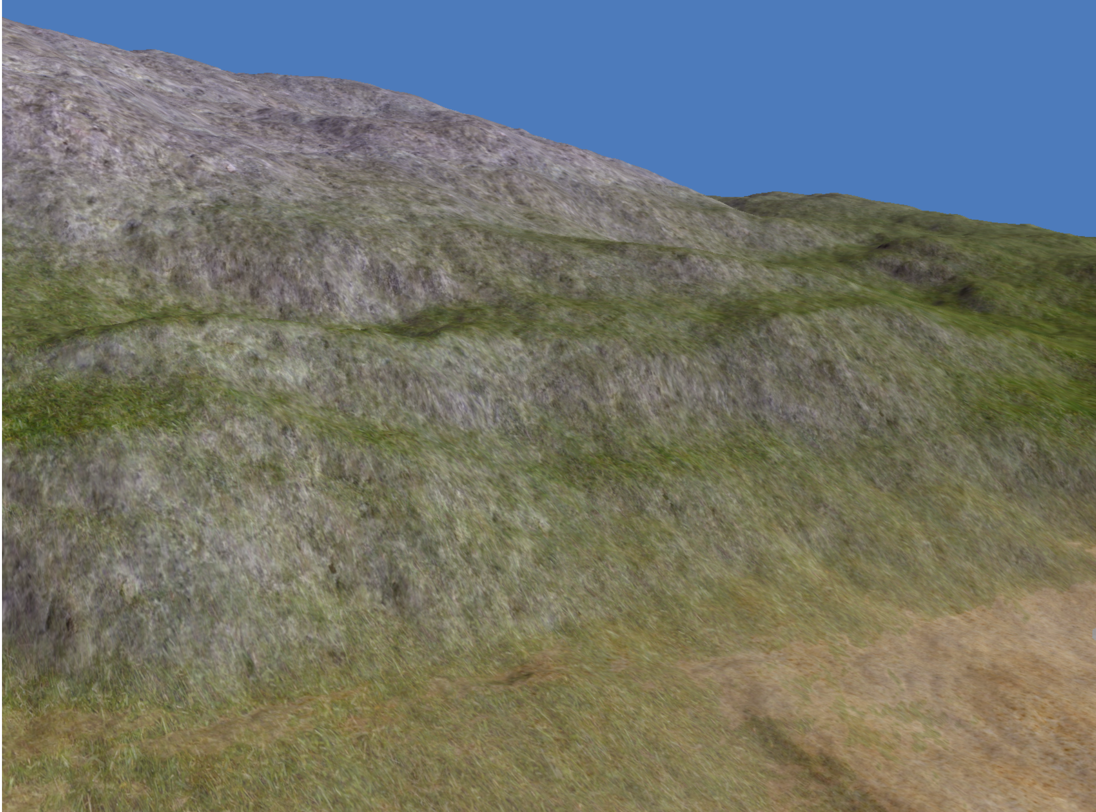


Figure 5.10: *Proto-textured Terrain. Proto-textures are combined using a height/slope-parameterized blending function. Note the absence of grass on steep rock formations.*

texture reduce operations [KW03b] this step can be fully implemented on the GPU. The effect is demonstrated in Figure 5.9.

Figure 5.10 shows an example to demonstrate the potential of proto-textures. As can be seen, at steep rock formations (where $\|\nabla\mathcal{H}\|_2$) is large) the grass texture is suppressed due to a low weight, while the rock texture is assigned a relatively high weight and is hence clearly visible.

5.5.3 Water

Water depths are computed per vertex according to an user-selected water level. As proposed by Schneider and Westermann [SW01], the water surface can then be rendered without major performance impact. While in the original paper the water surface was synthesized, we use a time-dependent normal map to obtain the appearance of moving waves. To achieve a different visual look for shallow and deep water, the strength of the

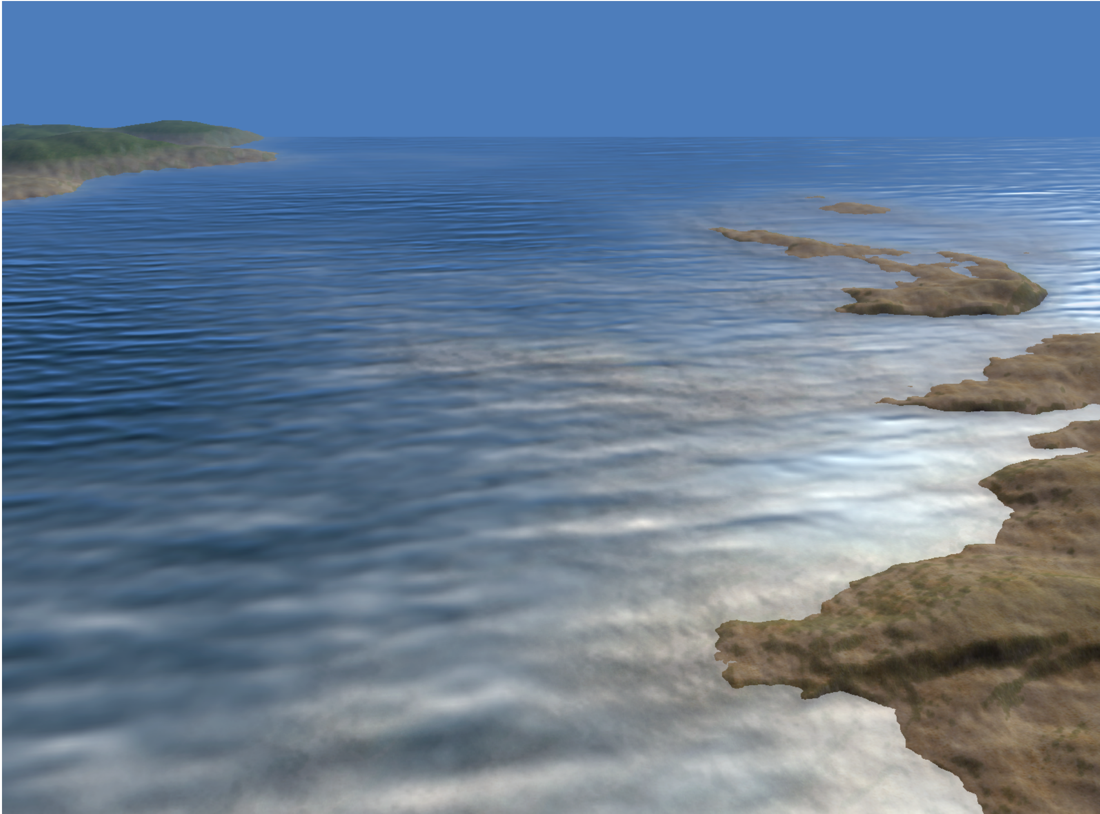


Figure 5.11: *Water surfaces in fractal landscapes. Animated water surfaces are integrated into the fractal synthesizer without sacrificing performance.*

bump effect, the reflectivity, and the color of the surface are modulated with the water depth. This gives shallow water a more transparent, less reflective look, while deep oceans get the green-ish hue observed in nature. Since the geometry of the ground is considerably more complex than the simple pool scene in the original paper, determining the length of the transmitted ray can no longer be done using a simple ray/hemicube intersection. Instead, we approximate the length by assuming a locally flat ground and computing a ray/plane intersection taking the interpolated per-pixel water depth into account. The resulting length can then be used to approximate caustics and extinction as proposed by Hall and Greenberg [HG83]. The results are shown in Figure 5.11 and Figure 5.12.

Figure 5.12: *Lake in a fractal landscape.*

5.5.4 Results

We used the proposed fractal synthesizer to generate a number of different scenes including auxiliary functions composed of several grayscale images, proto-textures, and a texture-based water surface. All of our tests were run on a single processor Pentium 4 equipped with an NVIDIA GeForce 7800 GTX. The described system was implemented using OpenGL. In all of our tests the landscape was evaluated at the vertices of a 512×512 projected grid. Finally, the grid was rendered to a 1280×1024 view port. Up to ten octaves were added to procedurally evaluate the fractal landscape. Some results can be seen in Figure 5.13.

Besides the appealing quality of the synthesized landscapes, the synthesizer still runs at highly interactive rates even at these high resolutions. All scenes are synthesized and rendered at about 70 frames per second on our target architecture. Of this time, roughly 30% is spent for projecting the grid vertices and evaluating the fractal at the projected grid points. The remaining time is spent for level of detail computations

(approximately 5%) and texturing (about 65%).

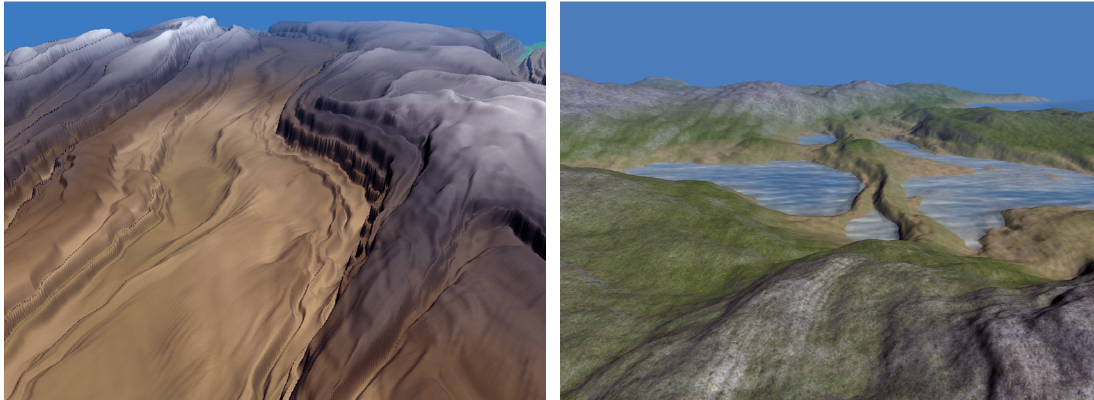


Figure 5.13: *Terrain types made possible by domain warping. Sand dunes (left) and rock formations (right) synthesized using our fractal model. Up to ten octaves were used in the evaluation of the landscape. Synthesis on a* 512^2 *grid and rendering to a* 1280×1024 *view port can be performed at about 70 frames per second on an NVIDIA GeForce 7800 GTX.*

5.6 Summary

This chapter describes an interactive fractal landscape synthesizer on programmable graphics hardware, which exploits the intrinsic strengths of GPUs to generate *and* render high-quality, high-resolution, textured and shaded terrains. Since the user interacts with the same data used to form the image, the parameter space can be intuitively managed. The interactive WYSIWIG interface for the synthesis of high-quality fractals has proven itself to be highly beneficial, especially for untrained users, in order to generate the desired results rapidly. Since the synthesis step is directly integrated into the rendering procedure, our method neither requires any polygonal representation nor a pre-processing stage. The suggested method is well suited for applications where the shape of the landscape is permanently being modified by the user.

Among other interesting directions for future research, one that seems particularly rewarding is to combine the synthesizer with local displacement overlays, for instance by using the techniques described in Chapter 6. Such a fusion of fractal synthesis and displacements could enable the realistic simulation of global and local erosion features. Also, due to the precise knowledge of the basis domain of the fractal landscape, parameterization computations as described in Chapter 6 can be entirely avoided. In return, simple erosion models could be evaluated in the time previously needed for parameterization computations. In this context it is important to notice that the fractal synthesizer

and editor run significantly faster on newer GPUs, e.g., the NVIDIA GeForce 280 GTX, effectively leaving enough computing power to pursue such a fusion of the two methods.

Another highly interesting direction is the extension towards the simulation of entire, virtual planets, including atmospheric effects and plant life. Especially the automatic creation of plants influenced by terrain characteristics, thereby automatically spawning geo-typical vegetation zones, seems to be a challenging but very promising route.

The projected grid, albeit a charming concept, is not free of problems. In this chapter we addressed certain problems arising in animations. Should high-quality rendering be desired, we propose to explicitly evaluate the terrain on the desired domain including proto-textures, and to feed it into a terrain rendering engine, e.g., the one presented in Chapter 4, or the more recently published system by Dick, Schneider, and Westermann [DSW08]. Another option would be to investigate whether the geometry clipmaps approach [LH04] is suited to replace the projected grid as a pre-viewing method for our fractal landscapes.

Chapter 6

Interactive Displacement Editing

Adding geometric details to a 3D polygonal object is typically used to apply a particular surface structure, thereby enhancing the object's visual appearance. Traditionally, this process is separate from object modeling and it is performed in a purely non-interactive way. On the other hand, there is a growing demand for integrating this process into the modeling phase. This requires techniques to add geometric details interactively that cannot be modeled directly on the object due to resolution limitations. Applications of such techniques are virtual prototyping and the creation of artistic content. Furthermore, these techniques also become important in real-time scenarios like computer games to instantly visualize shape changes caused by virtual characters. Examples for this include effects of contact between the user's avatar and parts of the scene, like footprints in snow, bullet holes, or objects sliding along each other. To support such operations, methods are required to "paste" geometric details over arbitrary surfaces in real time.

One such technique is displacement mapping, which manipulates the surface geometry by displacing surface points along the local surface normal [Coo84, CCC87]. The amount of displacement from the surface is usually determined via a surface parameterization that maps a displacement texture onto the surface. A continuous global parameterization that produces minimum distortion—and thus uniform resolution—of the texture map is required to assign geometric features at the same resolution everywhere on the surface. The computation of such a parameterization can be difficult—if not impossible—to achieve and is not suitable in applications where the surface undergoes frequent shape changes.

When used to augment polygonal surfaces, displacement mapping requires fine mesh subdivision such that the surface can accurately reflect selected geometric details. If geometric details are added to surface regions which are known a priori, they

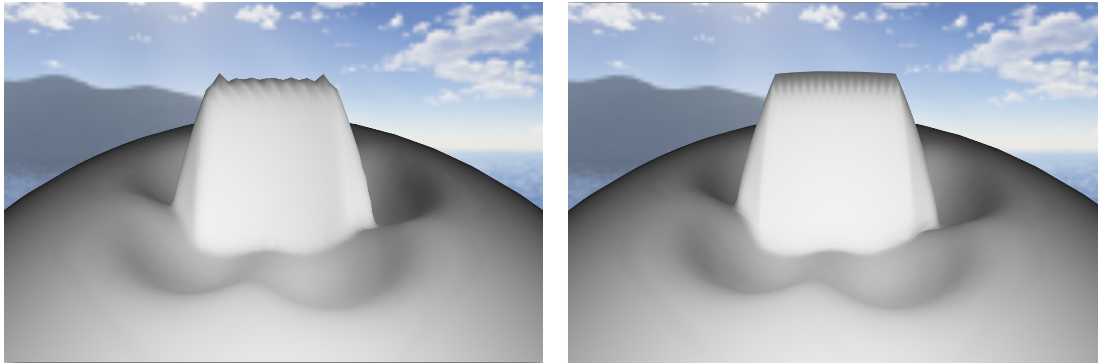


Figure 6.1: *Artifacts resulting from non-aligned base and displacement meshes. Left: A sphere is displaced by low (dent) and high (wedge) frequency geometric details using uniform subdivision. While low-frequency displacements are unproblematic, at sharp features artifacts are clearly visible since the base mesh is not aligned with the displacement. Right: Aligning base and detail geometry cures this problem.*

can be locally refined in a pre-process. However, in scenarios where the detail geometry is dynamically moved or re-positioned, this requirement limits the use of displacement mapping. In this case, the surface has either to be subdivided uniformly up to the maximum resolution required, or the affected surface regions have to be refined adaptively. Both approaches are problematic because they require either to store a huge amount of primitives or to evaluate a local refinement kernel in every frame. Furthermore, because vertex positions in the refined mesh do generally not coincide with the positions of samples in the displacement field, severe reconstruction artifacts such as shown in Figure 6.1 can occur.

6.1 Contribution

The primary focus of this chapter is the description of a fast and flexible method for displacement mapping. The method can render fine-detail geometric displacements on 2-manifold triangle base meshes. In contrast to previous methods, we do not displace the base mesh by a given displacement field. Instead a separate displacement geometry is used to replace parts of the base mesh. This makes the method independent of the resolution of the base mesh.

We present a novel approach to compute a local surface parameterization in real time to align the displacement mesh with the base surface. Firstly, we use particle-tracing to trace a regular 2D grid—the displacement grid—on the base surface. A par-

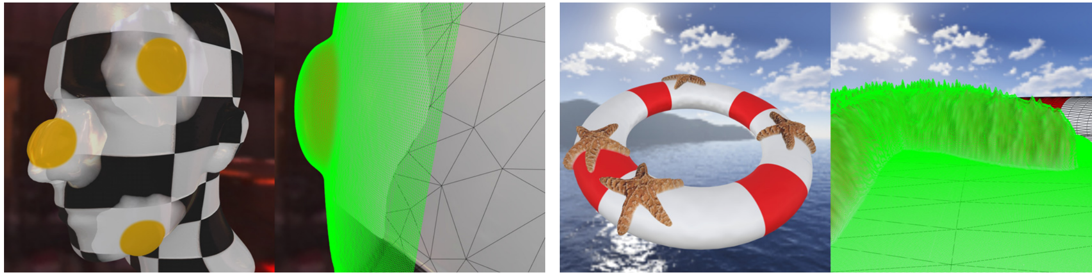


Figure 6.2: *Examples for our displacement method. These high-resolution displacement maps were rendered on triangle meshes in realtime. Both the rendering performance and the visual quality are almost entirely independent of the resolution of the base mesh. Image pairs show displacement effects on textured models, using displacement maps of size* 512×512*. Placement and rendering of displacement maps runs at interactive frame rates on a* 1600×1200 *viewport.*

ticular layout strategy is used to avoid folds in this grid. To improve its geometric quality towards isometry, we introduce a special 6-connected mass-spring system which relaxes the grid iteratively. Due to this approach, an explicit parameterization of the base surface is not required, and the method works even if a continuous 2D parameterization of the surface does not exist. Due to the special topology of the spring network, the relaxation process can be accelerated significantly by a GPU-based Gauss-Seidel solver. This allows us to position and animate high-resolution displacement geometries at very high speed.

By performing carving and extrusion operations in screen space on the GPU, we achieve a visually smooth transition between the base and the displacement mesh. Extrusion operations can be performed simply by using the z-buffer hardware. For carving, we introduce a novel approach to efficiently exclude an arbitrary part of the base surface from rendering. The displacement mesh can then be blended with the base mesh. To enable a seamless embedding of the displacement into the base mesh, appearance properties of the base mesh such as texture coordinates or tangent frames are propagated to the displacement mesh during rendering.

The benefits of our approach are (also see Figure 6.2):

- Arbitrarily fine geometric details on 2-manifold surfaces in real time.
- Dynamic displacement fields that can move on a surface.
- Immediate visual feedback of geometry “painting” on surfaces.

Our method is currently limited to displacement meshes that can be constructed from height fields. We also do not provide the possibility to cover the entire base surface

with a displacement map, i.e., we are restricted to local surface displacements. Finally, we assume that the geometric features to be added are small compared to features of the base surface, otherwise our method can produce folds.

The remainder of this chapter is organized as follows: In Section 6.2 we provide a detailed overview of related work. Next, our novel approach for computing a local parameterization is described, and the efficient realization of this approach on the GPU is outlined. Section 6.4 is dedicated to rendering aspects, including the transfer of appearance properties. We then present a number of examples produced by our method, and analyze the method's quality and speed. Finally we conclude the chapter with a discussion of the main contributions as well as potential limitations of our work.

6.2 Related Work

Due to recent advances in graphics hardware geometric surface refinements can efficiently be generated, displaced, and rendered using GPU-based subdivision techniques [DH00, VPBM01, SJP05, HLW07a, HLW07b, BS08]. Such approaches require a special base mesh, on which a refinement kernel is evaluated, as well as a surface parameterization, e.g., the natural parameterization of subdivision meshes, in order to map displacements to the vertices generated by the refinement kernel.

Displacement mapping via subdivision has additional problems in real-time applications if fine geometric details are dynamically moved on the base geometry. While uniform subdivision results in a tremendous amount of triangles to be generated, rendered, and stored, adaptivity cannot be achieved easily on recent GPUs due to the problem of dynamic memory allocation. Although future graphics APIs will allow adaptive evaluation of subdivision kernels [Gee08], the performance implications imposed by such an approach are not yet clear.

An alternative to geometric displacement mapping is image-based displacement mapping, which simulates the appearance of geometric displacements as viewed from a particular perspective by displacing texture coordinates on the initial surface accordingly. Prominent examples include bump mapping [Bli78], parallax [KTI$^+$01] and relief mapping [OBM00, POC05]. Image-based techniques can exploit the GPUs texture mapping and fragment processing capabilities efficiently, and they allow real-time rendering at reasonable image quality. On the other hand, these techniques require a surface parameterization to map the displacement texture onto the surface.

View-independent and generalized displacement maps [WWT$^+$03, WTL$^+$04] store a five-dimensional map of the displacements, in which the distance between the orig-

inal surface and the displaced geometry is encoded for each potential view. The major differences to view-independent displacement maps are that generalized displacement maps can simulate silhouettes correctly and that they do not require a surface parameterization. Both approaches require exhaustive pre-processing and compression, and therefore cannot handle deforming or moving displacement fields. Shell-based methods, which encode displacements into image layers or volumetric textures [Ney98, KS01, Elb05], model the displacement as a spatial structure. They also require extensive processing to rebuild this representation if these structures change.

6.3 Displacement Mapping

Most displacement techniques require a surface parameterization to map the given displacement field onto the surface. Generating a low-distortion parameterization on an arbitrary 2-manifold is numerically involved and often based on solving large systems of equations (see the surveys by Floater and Hormann [FH04, HSL$^+$] and Sheffer et al. [SPR06] for a thorough overview). A particular class of methods seek to compute surface parameterizations based on solving linear systems [DMA02, LPRM02, ZPKG02, SG03, ZRS05, SLMB05] or graph searches [SGW06], thus these approaches are suitable for interactive applications, since there exists a vast amount of highly efficient solvers for such systems. Specifically, local parameterization techniques using spring networks [Flo97] perform an energy-based relaxation of mesh vertices towards low distortion parameterizations.

The basic idea behind these methods is to interpret a connected part of a mesh as a mass-spring network. The boundary of this network is fixed in a 2D domain, and an energy functional is minimized that penalizes interior distortion. The network then deforms into a steady configuration, which yields a local parameterization. The quality of this parameterization, however, is strongly dependent on the shape of the initial boundary. Even though our technique is similar in spirit to these approaches, methodologically it is very different because the user only specifies a single boundary point, i.e., the position on the 3D surface where the patch should be centered. In particular, this allows moving the patch along the surface simply by re-positioning the patch center point. Furthermore, we solve the mass-spring system directly on the 3D surface, thereby enforcing all mass points to stay on this surface. To achieve fast convergence and stability, we propose a novel network topology which can be implemented efficiently on the GPU.

Our method starts with an arbitrary surface point $\boldsymbol{p_c}$, a height field, the height field's

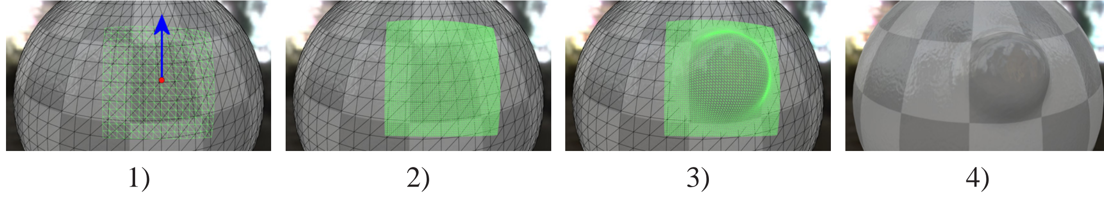


Figure 6.3: *Displacement mapping overview. From left to right: 1) A coarse grid is traced on the surface. The center point is marked in red, and the reference direction in blue. 2) A higher resolution grid. 3) The displaced grid. 4) The surface that is finally rendered.*

orientation $\boldsymbol{d_c}$ at $\boldsymbol{p_c}$ in the surface's local tangent plane, and its extent in object space. All these properties can be conveniently specified by the user, i.e., by a mouse click on the surface to determine $\boldsymbol{p_c}$. The direction $\boldsymbol{d_c}$ is then obtained automatically by projecting the current view's up direction onto the surface. As illustrated in Figure 6.3 the mapping of the displacement field onto the surface is then performed in two phases without any further user intervention. In the initial grid layout phase, vertices are traced on the base surface as described in Section 6.3.2. This phase is entirely performed on the CPU due to the inherently sequential nature of the algorithm utilized. Then, in the grid relaxation phase, the grid is optimized by a constrained mass-spring system. This phase is implemented to exploit the GPU's parallelism efficiently. Constraining the grid's vertices to the base mesh during this phase is again done by vertex tracing.

6.3.1 Initial Grid Layout

In the first phase of computing a local ad-hoc parameterization, a quadrangular grid around $\boldsymbol{p_c}$ is traced. The resulting grid then serves as a discrete mapping from the surface to the displacement field's domain. It is clear that the layout of a 2D grid on a surface can cause edges to compress or stretch. To avoid the folds resulting from such effects, we present two strategies.

Firstly, by tracing the grid in an advancing front manner, distortions around $\boldsymbol{p_c}$ can be distributed more homogeneously than with alternative tracing orders. Starting at $\boldsymbol{p_c}$, adjacent vertices are positioned by tracing along $\boldsymbol{d_c}$, a direction orthogonal to it, and the half way direction. Step sizes are chosen according to the grid spacing. As depicted in Figure 6.4, a total of four advancing fronts are generated—top, bottom, left, right—which are spanned by $\boldsymbol{p_c}$ and the newly generated points. Each new point stores the direction it was reached from. Before each advancing step, these directions are smoothed along each front by using a 3-tap binomial smoothing kernel, i.e., $[\frac{1}{4}, \frac{1}{2}, \frac{1}{4}]$. This is done to keep the front smooth in cases of sporadic small features of the base

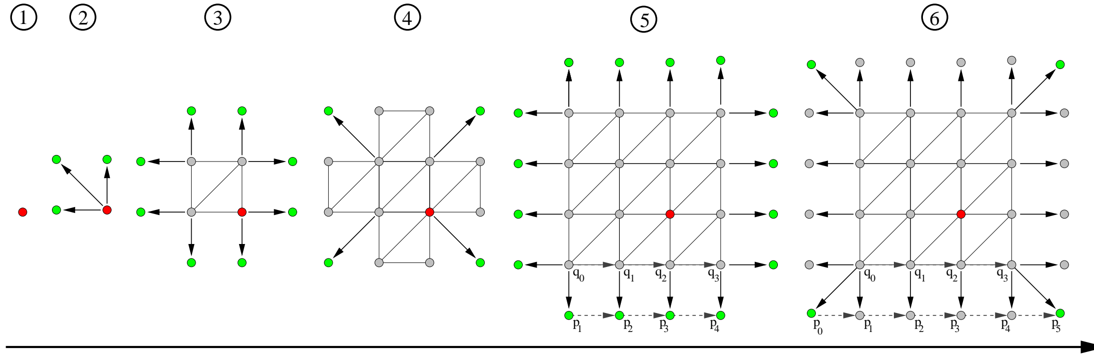


Figure 6.4: *Tracing of a quadrangular grid. The user picks a position (1) and the patch is automatically grown around this position (2-6) using an advancing front algorithm (new points are colored green).*

mesh. Afterwards, directions are re-projected into local tangent space. After smoothing, new directions are computed for corner points where two fronts meet. This is done by rotating vertices in the current fronts by $\pm 45°$ in the tangent plane, followed by averaging and re-projection into tangent space.

Secondly, we exploit our knowledge about the relative position of vertices to each other in the 2D grid. If a position $\boldsymbol{q_i}$ is "left" of a position $\boldsymbol{q_{i+1}}$, then, after advancing the front to obtain $\boldsymbol{p_{i+1}}$ from $\boldsymbol{q_i}$ and $\boldsymbol{p_{i+2}}$ from $\boldsymbol{q_{i+1}}$ (the index shift stems from corner points), the ordering between $\boldsymbol{q_i}$ and $\boldsymbol{q_{i+1}}$ is propagated, i.e., $\boldsymbol{p_{i+1}}$ should be "left" of $\boldsymbol{p_{i+2}}$. Although the notion of "left" seems intuitive enough in flatland, it can become amazingly complex on curved surfaces. Hence we rely on a criterion that first computes a plane η spanned by the base surface normal at $\boldsymbol{q_i}$ and the edge from $\boldsymbol{q_i}$ to $\boldsymbol{p_{i+1}}$. If the positions $\boldsymbol{q_{i+1}}$ and $\boldsymbol{p_{i+2}}$ are on the same side of η, we connect them by an edge and proceed. If not, positions $\boldsymbol{p_{i+1}}$ and $\boldsymbol{p_{i+2}}$ have to be swapped in the current front,

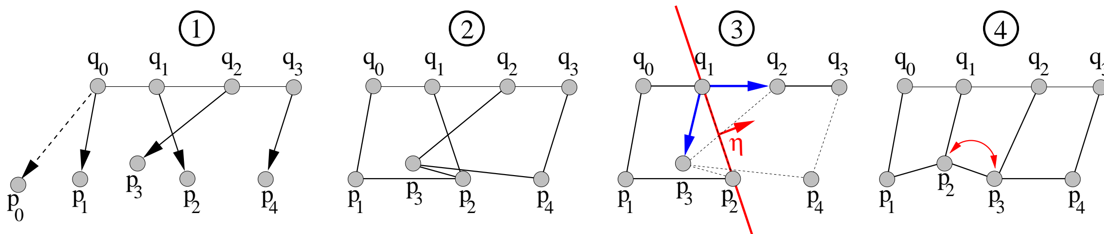


Figure 6.5: *Avoiding folds in patches. Since naïve connectivity of propagating fronts (1) results in folds (2), re-ordering the positions (3) is used to avoid these problems (4).*

and we proceed by checking the ordering between $\boldsymbol{p_i}$ and $\boldsymbol{p_{i+1}}$ by backtracking one position. This process is illustrated in Figure 6.5. To prevent trace trajectories from crossing again with other neighbors in an upcoming step, the tracing directions of two vertices are made parallel if these vertices have been swapped. This is achieved by replacing both participating directions by their halfway vector.

This method resolves all folds where only direct neighbors are involved (see also Figure 6.6). However, there is a possibility for folds to also affect positions that are not direct neighbors in any front. To resolve these cases, k-tuples of positions have to be examined. This extension is straightforward although we would like to point out that the runtime is in $\mathcal{O}(k^2)$. Also, for large k the underlying base mesh cannot safely be assumed to be locally planar, resulting in a plethora of pathological cases that seem to be very hard—if not impossible—to resolve. For these reasons we typically use a small neighborhood of $k = 3, \ldots, 5$ at the risk that folds affecting larger neighborhoods cannot be resolved effectively by this approach.

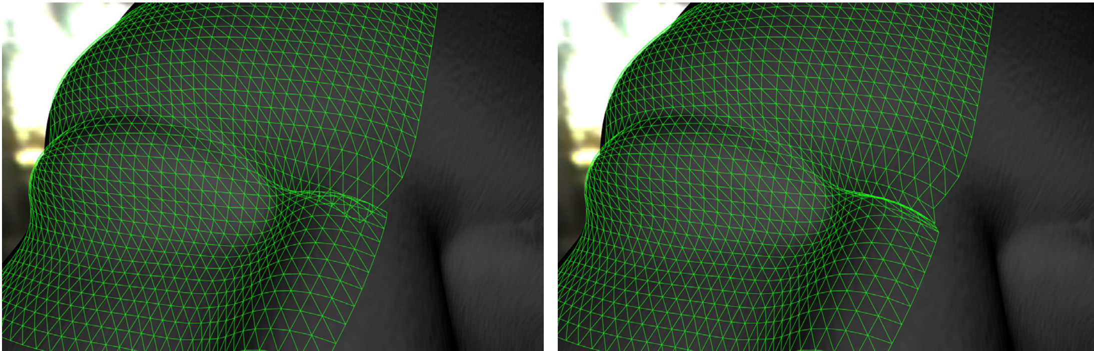


Figure 6.6: *Demonstrating Fold-Avoidance. Left: When a grid is traced in a region of relatively high curvature, folds can occur. Right: With our method we can avoid certain types of folds. The result is that particle trajectories that would otherwise cross now appear "bundled".*

These two steps are iterated for all fronts until an initial grid is obtained. This initial grid, however, typically shows distortions of grid cells. Before we discuss the relaxation process to improve the parameterization towards isometry, we first describe how to trace the grid positions on the triangle mesh.

6.3.2 Vertex Tracing on Triangle Meshes

Given the barycentric coordinates $\boldsymbol{p_b}$ and $\boldsymbol{d_b}$ of a position inside a triangle and a direction in the plane spanned by this triangle, tracing a particle from $\boldsymbol{p_b}$ in direction $\boldsymbol{d_b}$ leaves the triangle at $\boldsymbol{q} = \boldsymbol{p_b} + s_{\text{exit}} \cdot \boldsymbol{d_b}$. Denoting components of $\boldsymbol{p_b}, \boldsymbol{d_b}$ by

$p_{b,i}, d_{b,i},\ i \in [0 \dots 2]$, the parameter s_{exit} is obtained by observing that on each edge at least one of the barycentric coordinates vanishes. Hence,

$$s_{\text{exit}} = \min_{s} \{ s \geq 0 : \exists i : p_{b,i} + s \cdot d_{b,i} = 0 \} . \tag{6.1}$$

Vertex tracing now proceeds by successively jumping from edge to edge until the accumulated path length exceeds a given step size. If this is the case, the new vertex position inside the triangle is determined by barycentric interpolation. The direction $\boldsymbol{d_b}$ is propagated to the next triangle at each new point along the trace. This is detailed later in this section.

Given either a position or a direction in Euclidean coordinates, denoted $\boldsymbol{x}$, barycentric coordinates $\alpha_0, \alpha_1, \alpha_2$ with respect to a triangle T with Euclidean vertex positions $\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}$ can be computed as follows. First, a space for a plane containing T is computed, i.e.,

$$\begin{aligned} \boldsymbol{t} &:= \boldsymbol{v_1} - \boldsymbol{v_0} \\ \boldsymbol{b} &:= \boldsymbol{v_2} - \boldsymbol{v_0}. \end{aligned} \tag{6.2}$$

Next, obtaining barycentric coordinates is equivalent to solving the system

$$\sum_{i=0}^{2} \boldsymbol{v_i} \alpha_i = \boldsymbol{x}, \text{ constrained to} \tag{6.3}$$

$$\sum_{i=0}^{2} \alpha_i = 0|1, \tag{6.4}$$

where the right-hand side of Equation (6.4) is set to 0 if $\boldsymbol{x}$ is a direction and to 1 if $\boldsymbol{x}$ is a position. The problem with this formulation is its dimension of 3×4. This makes sense from a geometric point of view, since the system is uniquely solvable only if $\boldsymbol{x}$ lies in the plane η spanned by $\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}$, in which case the rank of the above system of equations is 3. However, even if $\boldsymbol{x}$ lies in η numerical instabilities are still likely to arise. A common solution is to first compute four-dimensional barycentric coordinates inside a tetrahedron $\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}$, where $\boldsymbol{v_3}$ may be any vertex that lies not in η. The barycentric coordinate associated with $\boldsymbol{v_3}$ is then discarded afterwards, thereby projecting $\boldsymbol{x}$ to η. Two problems arise with this method. Firstly, the projection of $\boldsymbol{x}$ is not necessarily orthogonal to η. Instead, the projected $\boldsymbol{x}$ is determined by the intersection between η and the line spanned by $\boldsymbol{x}$ and $\boldsymbol{v_3}$. This could be a problem for some applications. Secondly, a 4×4 system has to be inverted, which is unnecessarily

expensive. Another option would be so solve the system using the Moore-Penrose pseudo-inverse[1] [GVL96b], which cures the first issue, since the solution obtained in this manner is optimal in the least-squares sense. This, however, requires a 4×4 matrix-matrix product. The method we use is—to our knowledge—the most efficient method to compute triangular barycentric coordinates in arbitrary dimensions. We observe that by multiplying Equation (6.3) once by each $\boldsymbol{t}$ and $\boldsymbol{b}$ we obtain the following 3×3 system

$$\begin{pmatrix} \langle \boldsymbol{t}, \boldsymbol{v_0} \rangle & \langle \boldsymbol{t}, \boldsymbol{v_1} \rangle & \langle \boldsymbol{t}, \boldsymbol{v_2} \rangle \\ \langle \boldsymbol{b}, \boldsymbol{v_0} \rangle & \langle \boldsymbol{b}, \boldsymbol{v_1} \rangle & \langle \boldsymbol{b}, \boldsymbol{v_2} \rangle \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} \langle \boldsymbol{t}, \boldsymbol{x} \rangle \\ \langle \boldsymbol{b}, \boldsymbol{x} \rangle \\ 0 \,|\, 1 \end{pmatrix}. \tag{6.5}$$

This system can be solved highly efficiently. To speed up run-time computations even more, the matrix of Equation (6.5) is pre-computed in our method. Barycentric coordinates can be transformed back to Euclidean coordinates by $\boldsymbol{x} = \sum_i \alpha_i \boldsymbol{v_i}$. Note that $\boldsymbol{t}$ and $\boldsymbol{b}$ do not necessarily need to be orthogonal. Also note that using this method, $\boldsymbol{x}$ is projected orthogonally to both $\boldsymbol{t}$ and $\boldsymbol{b}$, and thus orthogonally to η.

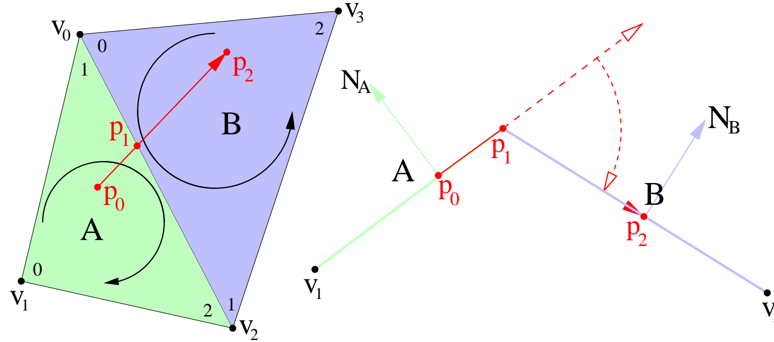


Figure 6.7: *Tracing a particle from $\boldsymbol{p_0}$ to $\boldsymbol{p_2}$. Left: Two triangles are shown with their global ($\boldsymbol{v_i}$) and local $0, 1, 2$ enumeration of vertices. Right: The same triangles viewed along their common edge. At the intermediate position $\boldsymbol{p_1}$ the neighboring face is determined, barycentric coordinates are permuted, and the tracing direction is propagated by rotation.*

Furthermore, we exploit the fact that if a vertex is traced from one triangle to the next, all three barycentric coordinates can be recycled when the vertex passes the common edge. The only operation necessary is to re-assign barycentric coordinates to the vertices of the next triangle. As depicted in Figure 6.7, the reason is that the same vertex may be assigned different local indices by each of the adjacent triangles. The re-

[1] Given a real-valued system $\mathcal{A}x = b$ with $\operatorname{rank}(\mathcal{A}) \geq \dim(\boldsymbol{x})$, the Moore-Penrose pseudo-inverse $\tilde{\mathcal{A}}$ is defined as $\left(\mathcal{A}^T \mathcal{A}\right)^{-1} \mathcal{A}^T$. A least-square solution can then be obtained by $x = \tilde{\mathcal{A}} b$.

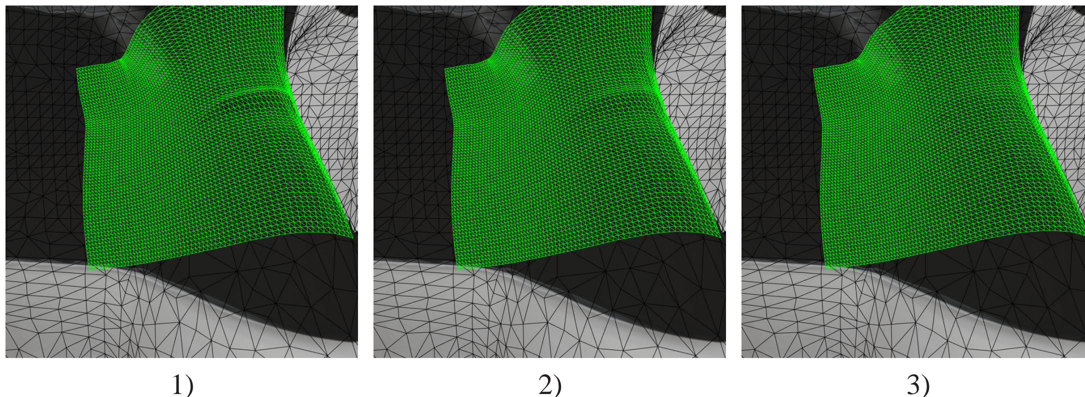
assignment is pre-computed in form of a permutation (also called *swizzle*) and stored for each edge. For instance, in Figure 6.7 the swizzle $(\alpha_0, \alpha_1, \alpha_2) \mapsto (\alpha_2, \alpha_0, \alpha_1)$ has to be performed at $\boldsymbol{p_1}$. If two adjacent triangles A and B with normals $\boldsymbol{N_A}, \boldsymbol{N_B}$ meet at acute angles, simple re-projection of the Euclidean direction from A into the plane of B is generally not correct. To always yield correct results, the direction is instead rotated around the common edge by an angle of $\arccos \langle \boldsymbol{N_A}, \boldsymbol{N_B} \rangle$.

6.3.3 Grid Relaxation

To improve the quality of the grid that has been traced on the base surface, we utilize a relaxation technique based on a mass-spring system. A center of mass is placed at each vertex, and each edge is interpreted as a spring. Note that this notion does not exclude additional springs that do not coincide with edges. If edges in the initial grid are compressed or stretched, the associated springs induce forces to restitute their rest lengths. Thus the system deforms into a configuration in which internal forces are compensated by opposing external forces. This process is demonstrated in Figure 6.8.

6.3.4 Mass-Spring Topology

In the development of a mass-spring systems for grid relaxation, two aspects have to be considered. Firstly, the system's topology should be chosen such as to automatically resolve folds in the grid. Secondly, the topology's effect on the numerical solver used to predict the dynamic behavior of the system has to be analyzed in terms of performance and stability.



1) 2) 3)

Figure 6.8: *A 2D grid placed on the nose of the Mannequin mesh. 1) The initial grid. 2) The relaxed grid after 16 iterations. 3) The final grid after 36 iterations.*

With respect to the first issue, we decided to use an *entangled* topology for the spring-mass system. The topology of this grid is shown in Figure 6.9.

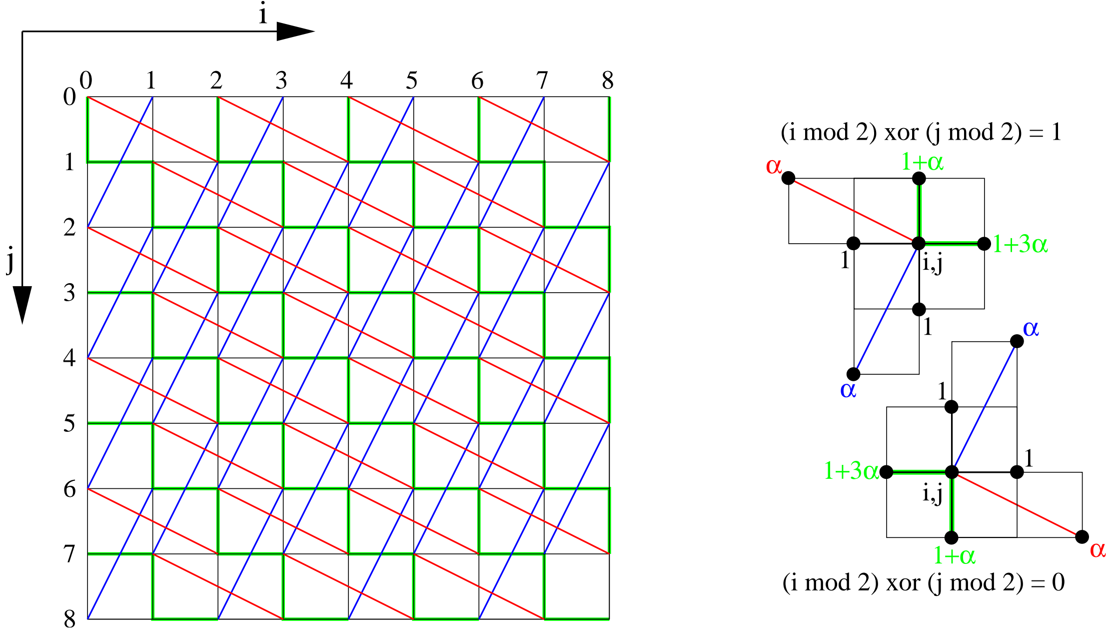


Figure 6.9: *The entangled topology of the mass-spring system used for grid relaxation is shown together with its connectivity rules.*

This particular topology has the following beneficial properties. Firstly, it is regularly 6-connected. Thus, only six forces have to be collected per mass point, as opposed to other common topologies that have higher valences. Secondly, regular topologies can be stored implicitly without requiring memory. Thirdly, regular topologies are highly advantageous for GPU-based implementations since they allow the parallel processing units to run in lock-step. Last but not least, the red and blue springs in Figure 6.9 effectively resolve folds because the existence of folds implies that some springs are not at rest length.

For the red, green, and blue springs in Figure 6.9, special connectivity rules are specified. At grid positions (i, j), we compute

$$\gamma \quad := \quad (i \bmod 2) \text{ xor } (j \bmod 2). \tag{6.6}$$

For positions (i, j) for which $\gamma = 1$, the upper rule of Figure 6.9 is used, while for $\gamma = 0$ the lower rule is used. Denoting edges of the mass-spring system as a tuple

$$(\,\{(i, j), (i', j')\}\,,\,\omega\,) \tag{6.7}$$

Table 6.1: *Connectivity rules for the entangled grid topology. The entangled grid topology has a single free weighting parameter α.*

$\gamma = 1$, "upper" rule	$\gamma = 0$, "lower" rule
$(\,\{(i,j),(i+1,j)\}\,,\,1+3\alpha\,)$	$(\,\{(i,j),(i-1,j)\}\,,\,1+3\alpha\,)$
$(\,\{(i,j),(i,j+1)\}\,,\,1\,)$	$(\,\{(i,j),(i,j-1)\}\,,\,1\,)$
$(\,\{(i,j),(i-1,j)\}\,,\,1\,)$	$(\,\{(i,j),(i+1,j)\}\,,\,1\,)$
$(\,\{(i,j),(i,j-1)\}\,,\,1+\alpha\,)$	$(\,\{(i,j),(i,j+1)\}\,,\,1+\alpha\,)$
$(\,\{(i,j),(i-1,j+2)\}\,,\,\alpha\,)$	$(\,\{(i,j),(i+1,j-2)\}\,,\,\alpha\,)$
$(\,\{(i,j),(i-2,j-1)\}\,,\,\alpha\,)$	$(\,\{(i,j),(i+2,j+1)\}\,,\,\alpha\,)$

where (i,j) and (i',j') are grid locations and ω denotes a stiffness parameter, these rules imply the insertion of the edges listed in Table 6.1.

Note that the upper and lower rules are antisymmetric in both i and j, i.e., if the rule for $\gamma = 1$ describes an edge $(\,\{(i,j),(i+i',j+j')\}\,,\,\omega\,)$ then the rule for $\gamma = 0$ describes an edge $(\,\{(i,j),(i-i',j-j')\}\,,\,\omega\,)$ and vice versa. The weights in Figure 6.9 and Table 6.1 are chosen such that the products of spring directions and weighting factors sum up to 0 at each vertex in order to keep the system isotropic. The system has a single free parameter α to weight the red and blue springs against the regular, black and green ones. The springs' rest lengths are in a ratio of $\sqrt{5} : 1$ (red and blue vs. green and black).

6.3.5 Mass-Spring Solver

We utilize a variation of the method proposed by Baraff and Witkin [BW98] to solve the mass-spring system. Positions of mass points are updated with respect to their velocity and acceleration using the Lagrangian law of motion:

$$\ddot{\boldsymbol{x}} \;=\; \mathcal{M}^{-1}\boldsymbol{f}(\boldsymbol{x},\dot{\boldsymbol{x}}) \tag{6.8}$$

In this formulation, $\mathcal{M}$ is the diagonal mass matrix and $\boldsymbol{f}(\boldsymbol{x},\dot{\boldsymbol{x}})$ is the net force vector accounting for all internal and external forces. The differential equation is first transformed into a system of first order differential equations

$$\frac{d}{dt}\begin{pmatrix}\boldsymbol{x}\\ \dot{\boldsymbol{x}}\end{pmatrix} \;=\; \begin{pmatrix}\boldsymbol{v}\\ \mathcal{M}^{-1}\boldsymbol{f}(\boldsymbol{x},\boldsymbol{v})\end{pmatrix} \tag{6.9}$$

in which $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{v})$ is approximated by the first two terms in its Taylor series expansion. The system can then be solved using an implicit Euler backward scheme, resulting in the following compact form (for details the reader is referred to [BW98]).

$$\left(\mathcal{M} - dt\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} - dt^2\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)\boldsymbol{\Delta v} = dt\left(\boldsymbol{f} + dt\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\boldsymbol{v}\right). \tag{6.10}$$

Solving Equation (6.10) for $\boldsymbol{\Delta v}$ allows integrating velocity and position to the next time step as $\boldsymbol{v}(t+dt) = \boldsymbol{v}(t) + \boldsymbol{\Delta v}$ and $\boldsymbol{x}(t+dt) = \boldsymbol{x}(t) + dt \cdot \boldsymbol{v}(t+dt)$. In the two following sections we describe two possible ways to solve Equation (6.10). First, Section 6.3.6 discusses the Euler backward solver first proposed by Baraff and Witkin [BW98] to make this thesis self-contained. After that, Section 6.3.7 presents a full derivation of our Gauss-Seidel formulation. The particular choice of using a Gauss-Seidel-based solver is detailed in Section 6.3.8.

6.3.6 Euler Backward Formulation

As originally proposed in [BW98], Equation (6.10) can be solved directly, effectively resulting in an implicit Euler backward scheme. Since the system matrix is sparse and usually well-conditioned, pre-conditioned conjugate gradient [She94, PTVF02b] methods—the method of choice to solve such systems—typically converge quickly. However, the system of equations can become very large—a 512^2 patch already results in a $2^{18} \times 2^{18}$ system matrix. As a consequence, frequent restarts of the conjugate gradient solver are necessary in order to avoid the accumulation of round-off errors. Clearly, this is even aggravated if the target architecture, i.e., the GPU, does not support full IEEE 754 double precision arithmetic. As a highly undesired side-effect, frequent restarts slow down the process of solving the system in terms of computing time. One possible way to avoid frequent restarts is to utilize a multigrid solver [Geo08] which uses a conjugate gradient method as smoothing backend.

We initially tried to solve the mass-spring system using the multigrid solver of Georgii [Geo08]. However, in terms of computing time, the Gauss-Seidel solver described in the next section outperformed the multigrid solver on the CPU. Still it should be noted that multigrid methods proovably converge quicker than “simple” conjugate gradient methods such as proposed in [BW98]. However, since a full multigrid solver is rather intricate to implement on the GPU, and for reasons that are discussed in Section 6.3.8, we consequently chose to not pursue the direction of a GPU-based multigrid conjugate gradient solver. Instead, we focussed on a Gauss-Seidel-based solver that can be implemented on the GPU efficiently.

6.3.7 Gauss-Seidel Formulation

To solve Equation (6.10) using Gauss-Seidel style updates, we first introduce indices denoting to which vertex the respective entity is associated, i.e., $\boldsymbol{v_i}$ denotes the velocity at the ith vertex and so forth. It should be noted that indices do not follow Einstein notation, i.e., unless explicitly stated, the automatic summation to reduce occurences of like indices does not take place. We define the neighborhood $\mathcal{N}(i)$ to contain all indices j, such that vertex i is adjacent to vertex j. $j \in \mathcal{N}(i)$ thus implies the existence of a spring connecting masses i and j. Since springs are not directed, $j \in \mathcal{N}(i)$ also implies $i \in \mathcal{N}(j)$. To make the formulation more compact, we further define $\mathcal{N}(i) \dot\cup \{i\} =: \widetilde{\mathcal{N}}(i)$. The notion of the neighborhood of vertex i allows for a compact and concise description of the ith row of the system matrix of Equation (6.10). We can thus reformulate the four sparse matrix vector products that arise if both sides are expanded as a gathering operation on $\boldsymbol{\Delta v}$ that requires a traversal of $\mathcal{N}(i)$. We denote the force exerted on vertex i by the spring between i and j by $\boldsymbol{f_{ij}}$ and define $\boldsymbol{f_i} := \sum_{j \in \mathcal{N}(i)} \boldsymbol{f_{ij}}$. Equation (6.10) is thus rewritten as

$$\left[\mathcal{M} - dt\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{v}} - dt^2\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right]_{ij} \boldsymbol{\Delta v_j} = \left[dt\boldsymbol{f} + dt^2\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\boldsymbol{v}\right]_i, \tag{6.11}$$

where i and j denote the respective vector and matrix entries. Expanding this equation and denoting sums explicitly, we obtain

$$\sum_{j \in \widetilde{\mathcal{N}}(i)} \left(\mathcal{M} - dt\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{v_j}} - dt^2\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}}\right) \boldsymbol{\Delta v_j} = \sum_{j \in \widetilde{\mathcal{N}}(i)} \left(dt\boldsymbol{f_{ij}} + dt^2\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}}\boldsymbol{v_j}\right), \tag{6.12}$$

In this approach, forces $\boldsymbol{f_{ij}}$ comprise both external and internal forces. Baraff and Witkin [BW98] observe that choosing an energy functional of the form $\mathcal{E}(\mathcal{C}, \boldsymbol{x}) = \frac{1}{2}\kappa^s \mathcal{C}(\boldsymbol{x})^T \mathcal{C}(\boldsymbol{x})$ minimizes $|\mathcal{C}(\boldsymbol{x})|$ for a condition $\mathcal{C}(\boldsymbol{x})$. Here, $\boldsymbol{x}$ is the position at which to evaluate $\mathcal{C}$, and κ^s is an arbitrary stiffness constant. Note that by construction, $\mathcal{E} : (\mathbb{R}^3 \to \mathbb{R}^3) \times \mathbb{R}^3 \to \mathbb{R}_0^+$, i.e., $\mathcal{E}$ is positive or zero on its entire domain. From a physical point of view, the potential $\mathcal{E}$ gives rise to a force field, denoted $\boldsymbol{f}$, of the form

$$\boldsymbol{f_i} = -\frac{\partial \mathcal{E}}{\partial \boldsymbol{x_i}} = -\kappa^s \frac{\partial \mathcal{C}(\boldsymbol{x})}{\partial \boldsymbol{x_i}} \mathcal{C}(\boldsymbol{x}) \tag{6.13}$$

Computing the partial derivatives of $\boldsymbol{f_i}$ occuring in Equation (6.11) yields

$$\frac{\partial \boldsymbol{f_i}}{\partial \boldsymbol{x_j}} = \frac{\partial^2 \mathcal{E}}{\partial \boldsymbol{x_i} \partial \boldsymbol{x_j}} = \left(\frac{\partial \mathcal{C}(\boldsymbol{x}) \partial \mathcal{C}(\boldsymbol{x})^T}{\partial \boldsymbol{x_i} \partial \boldsymbol{x_j}} + \frac{\partial^2 \mathcal{C}(\boldsymbol{x})}{\partial \boldsymbol{x_i} \partial \boldsymbol{x_j}} \mathcal{C}(\boldsymbol{x}) \right) \tag{6.14}$$

Baraff and Witkin [BW98] note that "since $\mathcal{C}(\boldsymbol{x})$ does not depend on $\boldsymbol{v}$, the matrix $\partial \boldsymbol{f} / \partial \boldsymbol{v}$ is zero". This makes sense from a physical point of view, since the corresponding potential field $\mathcal{E}$ is independent of $\boldsymbol{v}$ and thus excludes the existence of damping forces. However, Desbrun et al. [DMB00] propose to use a Rayleigh damping force of the following form instead.

$$\begin{aligned} \frac{\partial \boldsymbol{f_{ij}^d}}{\partial \boldsymbol{v_j}} &= \kappa^d \mathcal{I} \;\; \forall i \neq j, \text{ and} \\ \boldsymbol{f_{ij}^d} &= -\kappa^d \boldsymbol{v_{ij}} \end{aligned} \tag{6.15}$$

where κ^d is a damping constant, and $\mathcal{I}$ denotes a 3×3 identity matrix. We define $\boldsymbol{v_{ij}}$ and $\boldsymbol{x_{ij}}$, the difference in velocity and position of two vertices i and j, in the following manner.

$$\boldsymbol{x_{ij}} = \begin{cases} (\boldsymbol{x_i} - \boldsymbol{x_j}) & \text{if } j \in \mathcal{N}(i) \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{6.16}$$

$$\boldsymbol{v_{ij}} = \begin{cases} (\boldsymbol{v_i} - \boldsymbol{v_j}) & \text{if } j \in \mathcal{N}(i) \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{6.17}$$

Here, $\boldsymbol{x_i}, \boldsymbol{x_j}$ denote the ith and j^{th} vertex' position, while $\boldsymbol{v_i}, \boldsymbol{v_j}$ denote the respective velocities. In the presence of damping forces of the above form, a modified Hooke's Law with a potential field $\mathcal{E}$ subject to a Rayleigh dissipation function $\mathcal{D}$ is obeyed instead of the classical model that just takes $\mathcal{E}$ into account. In this case, the dissipation function is defined as

$$\begin{aligned} \mathcal{D}_{ij} &:= \frac{1}{2} \kappa^d \boldsymbol{v_{ij}^T} \boldsymbol{v_{ij}} \\ \mathcal{D} &:= \sum_{ij} \mathcal{D}_{ij}, \end{aligned} \tag{6.18}$$

and the forces arising due to $\mathcal{D}$ are computed as

$$\boldsymbol{f_{ij}^d} = -\frac{\partial \mathcal{D}_{ij}}{\partial \boldsymbol{v}} = -\kappa^d \boldsymbol{v_{ij}} \tag{6.19}$$

The classical Hooke's Law still provides the potential field $\mathcal{E}$, i.e.,

$$\begin{aligned} \mathcal{E}_{ij} &:= \frac{1}{2}\kappa^s \left(\boldsymbol{\Delta x_{ij}}\right)^T \left(\boldsymbol{\Delta x_{ij}}\right) \\ \mathcal{E} &:= \sum_{ij} \mathcal{E}_{ij}, \end{aligned} \tag{6.20}$$

where $\boldsymbol{\Delta x_{ij}}$ denotes the current vectorial elongation of the spring connecting vertices i and j with respect to the spring's (scalar) rest length L_{ij}.

$$\boldsymbol{\Delta x_{ij}} := \begin{cases} \left(|\boldsymbol{x_{ij}}| - L_{ij}\right) \frac{\boldsymbol{x_{ij}}}{|\boldsymbol{x_{ij}}|} & \text{if } j \in \mathcal{N}(i) \\ 0 & \text{otherwise} \end{cases} \tag{6.21}$$

The potential field $\mathcal{E}$ gives rise to forces which can be computed as

$$\boldsymbol{f^s_{ij}} = -\frac{\partial \mathcal{E}_{ij}}{\partial \boldsymbol{x_j}} = -\kappa^s \Delta \boldsymbol{x_{ij}}. \tag{6.22}$$

The total force due to a spring connecting vertices i and j is then $\boldsymbol{f_{ij}} = \boldsymbol{f^s_{ij}} + \boldsymbol{f^d_{ij}}$. Setting $\mathcal{C}(\boldsymbol{x}) = \boldsymbol{\Delta x_{ij}}$ in Equation (6.14) results in the following formulation [TE05] of the derivatives

$$\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}} = -\frac{\partial^2 \mathcal{E}_{ij}}{\partial \boldsymbol{x_i} \partial \boldsymbol{x_j}} = -\kappa^s \frac{\boldsymbol{x_{ij}} \boldsymbol{x^T_{ij}}}{\boldsymbol{x^T_{ij}} \boldsymbol{x_{ij}}} - \kappa^s \left(1 - \frac{L}{|\boldsymbol{x_{ij}}|}\right) \left(I - \frac{\boldsymbol{x_{ij}} \boldsymbol{x^T_{ij}}}{\boldsymbol{x^T_{ij}} \boldsymbol{x_{ij}}}\right), \quad \forall i \neq j \tag{6.23}$$

Since the matrix of forces $\boldsymbol{f_{ij}}$ is anti-symmetric, i.e., $\boldsymbol{f_{ij}} = -\boldsymbol{f_{ji}}$, the computation of $\frac{\partial \boldsymbol{f_i}}{\partial \boldsymbol{x_i}}$ is feasible. This is important because Equation (6.23) is undefined for $i = j$ (since for $\boldsymbol{x_{ii}} = 0$ singularities arise), preventing a naïve evaluation.

$$\begin{aligned} \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_i}} &= -\frac{\partial \boldsymbol{f_{ji}}}{\partial \boldsymbol{x_i}} = -\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}}, \text{ thus} \\ \frac{\partial \boldsymbol{f_i}}{\partial \boldsymbol{x_i}} &= -\sum_{j \in \mathcal{N}(i)} \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}}, \text{ and analogously} \\ \frac{\partial \boldsymbol{f_i}}{\partial \boldsymbol{v_i}} &= -\sum_{j \in \mathcal{N}(i)} \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{v_j}} \end{aligned} \tag{6.24}$$

Note that since each $\mathcal{E}_{ij}$ and $\mathcal{D}_{ij}$ are C^2-continuous in $\boldsymbol{x}$ and $\boldsymbol{v}$ respectively, Clairot's theorem [Cla43] guarantees all matrices $\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}}$ and $\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{v_j}}$ to be symmetric. Consequently, the system matrix of (6.11) is a sum of symmetric matrices which is itself symmetric.

The goal of this section is a reformulation of Equation (6.11) that allows solving for the unknowns $\boldsymbol{\Delta v}$ using a Gauss-Seidel solver. To achieve this, the summation over $j \in \widetilde{\mathcal{N}}(i)$ has to be split into a sum over $j \in \mathcal{N}(i)$ and a single term depending on $\boldsymbol{\Delta v_i}$. After reordering and by denoting the ith diagonal entry of $\mathcal{M}$ by M_i (the mass of vertex $\boldsymbol{x}_i$), the following system is to be solved.

$$\begin{aligned} \mathcal{A}_i \boldsymbol{\Delta v_i} &= \boldsymbol{b_i}, \quad \text{where} \\ \mathcal{A}_i &:= \left(M_i - dt \frac{\partial \boldsymbol{f_i}}{\partial \boldsymbol{v_i}} - dt^2 \frac{\partial \boldsymbol{f_i}}{\partial \boldsymbol{x_i}} \right) \\ \boldsymbol{b_i} &:= \left(\boldsymbol{f_i} + \sum_{j \in \mathcal{N}(i)} \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{v_j}} \boldsymbol{\Delta v_j} \right) dt + \left(\sum_{j \in \widetilde{\mathcal{N}}(i)} \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}} \boldsymbol{v_j} + \sum_{j \in \mathcal{N}(i)} \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}} \boldsymbol{\Delta v_j} \right) dt^2 \end{aligned} \tag{6.25}$$

This can be further condensed using Equation (6.24) to replace the two diagonal entries of the partial derivative matrices by a sum over $\mathcal{N}(i)$. Furthermore, Equation (6.15) is used to replace $\frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{v_j}}$ by $\kappa^d \mathcal{I}$.

$$\begin{aligned} \mathcal{A}_i &= M_i + \sum_{j \in \mathcal{N}(i)} \left(-\kappa^d \mathcal{I} dt + \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}} dt^2 \right) \\ b_i &= \sum_{j \in \mathcal{N}(i)} \left(\left(\boldsymbol{f_{ij}} + \kappa^d \boldsymbol{\Delta v_j} \right) dt + \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}} \left(\boldsymbol{v_j} + \boldsymbol{\Delta v_j} - \boldsymbol{v_i} \right) dt^2 \right) \end{aligned} \tag{6.26}$$

Each Gauss-Seidel iteration to solve this equation then traverses each vertex i, gathers $\mathcal{A}_i$ and $\boldsymbol{b_i}$ using the information currently available (i.e., updated information for $j < i$ and non-updated information otherwise). Then, a 3×3 system of equations of the form $\mathcal{A}_i \boldsymbol{\Delta v_i} = \boldsymbol{b_i}$ is solved in order to compute velocity updates. If after solving each such system an immediate update of the form

$$\begin{aligned} \boldsymbol{v_i} &\Leftarrow \boldsymbol{v_i} + \boldsymbol{\Delta v_i} \\ \boldsymbol{x_i} &\Leftarrow \boldsymbol{x_i} + dt \cdot \boldsymbol{v_i} \end{aligned} \tag{6.27}$$

is performed, the term $\boldsymbol{v_j} + \boldsymbol{\Delta v_j}$ can be further condensed to just $\boldsymbol{v_j}$ since $\boldsymbol{v_j}$ already contains any possible update by $\boldsymbol{\Delta v_j}$. It is worth noting that unlike using the fully implicit backward Euler solver by Baraff and Witkin [BW98] $\boldsymbol{\Delta v_j}$ might stem from a prior Gauss-Seidel step. Finally, using $\boldsymbol{v_{ij}} = -\boldsymbol{v_{ji}}$ reduces the right-hand side to

$$\boldsymbol{b_i} = \sum_{j \in \mathcal{N}(i)} \left(\left(\boldsymbol{f_{ij}} + \kappa^d \boldsymbol{\Delta v_j} \right) dt - \frac{\partial \boldsymbol{f_{ij}}}{\partial \boldsymbol{x_j}} \boldsymbol{v_{ij}} dt^2 \right) \tag{6.28}$$

Convergence Criterion

Normally, the residual error ρ of an approximate solution $\boldsymbol{\Delta v}$ to the system $\mathcal{A}\boldsymbol{\Delta v} = \boldsymbol{b}$ would be computed as $\rho = \|\mathcal{A}\boldsymbol{\Delta v} - \boldsymbol{b}\|$ using an appropriate vector metric $\| \cdot \|$. However, with Gauss-Seidel updates this is not always feasible. The reason is that $\mathcal{A}_i, \boldsymbol{b_i}$ have never to be stored explicitly at each vertex; so in order to achieve a minimum memory footprint one would typically recycle the memory occupied by $\mathcal{A}_i$ and $\boldsymbol{b_i}$ for the system of equations at the next vertex. Since a single Gauss-Seidel operation eliminates the residual at the respective row, the residual also cannot be computed incrementally in this manner. However, a way to obtain a quick guess about ρ that "behaves right" in this application is to use a metric that depends on both $\boldsymbol{\Delta v}$ and $\boldsymbol{v}$. The rationale is that $\frac{1}{dt}\boldsymbol{\Delta v_i}$ is the approximation of the acceleration at vertex i. Should, at any point in time, all velocities and accelerations vanish for every vertex in the system, no further progress will be made by additional iterations. On the other hand it might not be sufficient to rely on either acceleration or velocity alone, as accelerations typically give rise to velocities and velocities might give rise to forces (for instance by compressing springs), implying accelerations. The only care that has to be taken using a velocity/aceleration-based convergence criterion is that external velocities and accelerations have to be carefully separated from their internal counterparts. In this application, however, the mass-spring system is fixed in space and is free from external forces and velocities. From Equation (6.27) an equivalent metric can be obtained by accumulating the squared position updates $dt^2\boldsymbol{v_i}^T\boldsymbol{v_i}$ on $\boldsymbol{x_i}$. However, the residuum should not depend on dt and we consequently mandate the use of $\boldsymbol{v_i}^T\boldsymbol{v_i}$. It is also worth noting that this convergence criterion is not monotonically decreasing; it first increases over a few iterations before it starts decreasing. The reason is that the first few position updates tend to be very small due to the damping forces.

If one is interested in a quantitative and formal residuum, for instance because the system's residuum cannot vanish due to hard constraints, one option is to gather all matrices and right-hand sides $\mathcal{A}_i, \boldsymbol{b_i}$ *after* the previously described criterion attests convergence of the system to compute ρ. When immediate updates in the form of Equation (6.27) are used, this requires to traverse each vertex one more time. During traversal, the 3×3 system described above is gathered and solved to obtain the current $\boldsymbol{\Delta v}$, but the update of Equation (6.27) is not performed. Instead, the ith residual is computed as $\rho_i = \|\mathcal{A}_i\boldsymbol{\Delta v_i} - \boldsymbol{b_i}\|$ and accumulated. Since this is rather straight-forward but would require a lot of space, we did not include it in Algorithm 2. Note that this algorithm does not provides means to add external forces of any kind. Furthermore, it is assumed that stiffness and damping constants are fed to the algorithm on a per-spring

basis instead of the global constants used for simplicity's sake in the derivation.

Algorithm 2 Gauss-Seidel Mass-Spring Solver

Input:

Mass positions (vertices) $\boldsymbol{x_i}$ and masses M_i
A list of fixed positions F
Topological information $\mathcal{N}(i)$
Rest-lengths of springs $L_{ij}\ \forall i,j : j \in \mathcal{N}(i)$
Stiffness and damping konstants $\kappa^d_{ij}, \kappa^s_{ij}\ \forall i,j : j \in \mathcal{N}(i)$
Maximum number iterations $k_{\max}$ and threshold $\varepsilon > 0$
Integration step size dt

Output:

Relaxed mass positions $\boldsymbol{x_i}$

```
// Initialization:
v_i ⇐ 0, ∀i
Δv_i ⇐ 0, ∀i
k ⇐ 0
repeat
    r ⇐ 0
    for each vertex i ∉ F do
        // Gather matrix A_i and right-hand side b_i acc. to Equations (6.26) and (6.28).
        A_i ⇐ M_i
        b_i ⇐ 0
        for each j ∈ N(i) do
            ∂f_ij/∂x_j ⇐ −κ^s_ij (x_ij x_ij^T)/(x_ij^T x_ij) − κ^s_ij (1 − L_ij/|x_ij|) (I − (x_ij x_ij^T)/(x_ij^T x_ij)) // Equation (6.23)
            A_i ⇐ A_i − κ^d_ij I dt + (∂f_ij/∂x_j) dt^2
            f_ij ⇐ −κ^s_ij Δx_ij − κ^d_ij v_ij
            b_i ⇐ b_i + (f_ij + κ^d_ij Δv_j) dt − (∂f_ij/∂x_j) v_ij dt^2
        end for
        // Solve 3 × 3 system
        Δv_i ⇐ A_i^{-1} b_i
        // Immediate update of velocity and position.
        v_i ⇐ v_i + Δv_i
        x_i ⇐ x_i + dt v_i
        r ⇐ r + v_i^T v_i // Accumulate convergence criterion
    end for
    increment k
until k > k_max or r < ε
return x_i
```

6.3.8 Constrained Updates, Quality Metric, and Stepsize Control

Under the constraint of keeping $\boldsymbol{x}$ on the base surface, the position can be updated once $\boldsymbol{\Delta x} = dt \cdot \boldsymbol{v}(t + dt)$ is known by tracing $\boldsymbol{x}$ in the direction $\boldsymbol{\Delta x}$—projected onto the base surface—by a distance of $\|\boldsymbol{\Delta x}\|_2$. Thus, $\boldsymbol{x}$ never leaves the base mesh, and $\|\boldsymbol{\Delta x}\|_2$ is measured on the base surface. Afterwards, $\boldsymbol{v}(t + dt)$ and $\boldsymbol{\Delta v}$ have to be adjusted accordingly to ensure consistency. We picked three grids positioned in different places on the Mannequin as depicted in Figure 6.10. Then, we varied the

Figure 6.10: *The three initial grids used for the mass-spring tests. In these images, folds have not been resolved in order to give a better understanding for the complexity of the grids.*

amount of solving steps—while accumulating $\boldsymbol{\Delta x}$—in between position re-traces. Our results (as depicted in Figure 6.11) clearly indicate that updating constrains as soon as possible is advantageous. We consequently prefer to use a Gauss-Seidel solver instead of the conjugate gradient solver proposed in [BW98]. This allows us to perform an immediate update as soon as $\boldsymbol{\Delta x}$ is known for a specific position, as opposed to being able to performing updates only when $\boldsymbol{\Delta x}$ is known for all positions. Since a Gauss-Seidel solver processes one vertex at a time, a 3×3 system derived from Equation (6.10) needs to be solved per vertex. Note that analytical hard constraints using Lagrange multipliers are not feasible, since this would require to solve for an additional number of variables on the order of the number of faces of the underlying mesh.

The convergence criterion $\boldsymbol{v}_i^T \boldsymbol{v}_i$ described in Section 6.3.7 prooved problematic for various reasons. Firstly, it does not decrease monotonically. As a natural consequence this requires the use of heuristics to diagnose convergence. Secondly, it does not vanish due to the hard constraints present in our application. And thirdly, it does not measure quality very intuitively. In this section we describe a more intuitive, albeit computationally more involved metric that also can also be used to control the solver's stepsize adaptively.

The metric we use to assess the quality of the grids obtained during and after relax-

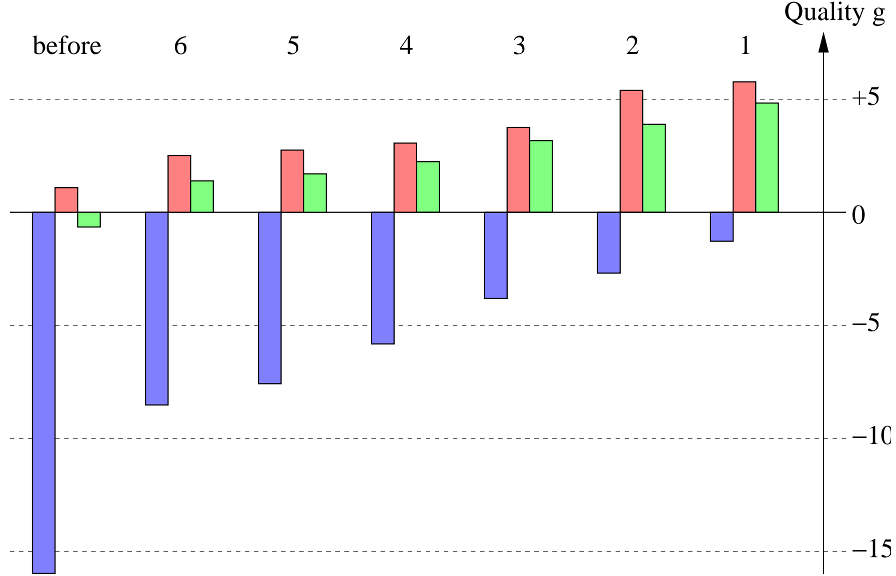


Figure 6.11: *Benefits of immediate position constraints. For the entangled topology, this bar chart demonstrates the quality gain (see Equation (6.29)) in spring relaxation when using the same number of Gauss-Seidel steps (60) and constraining spring vertices to the surface after every n^{th} step. From left to right, n goes from 6 to 1. Three initial patches (colored red, green, and blue) were used in this test (also see Figure 6.10).*

ation is of the form

$$g = 50 \cdot \left(1 - \frac{\sigma(A_i)}{\mu(A_i)} - \frac{1}{360°} \sum_{i,j} |\alpha_{ij} - 90°|\right) - 40, \tag{6.29}$$

where $\mu(A_i)$ is the average area of all quadrangular grid cells with a standard deviation of $\sigma(A_i)$, and α_{ij} refers to j^{th} interior angle of cell i. The metric thus measures isometry by placing a penalty on non-right angles and area deviations. Positive values (up to a maximum of 10) typically correspond to acceptable visual quality.

This metric, when computed on the GPU after each iteration, can be used to steer the parameters of the Gauss-Seidel solver effectively as follows. We require that the value of g is strictly monotonically decreasing with each iteration—a requirement that cannot be imposed on the residuum. If this is not the case, we perform a *partial restart* of our mass-spring system. Denoting the state before the last iteration by $\boldsymbol{x}_{\text{old}}$ etc. , we

set

$$
\begin{aligned}
\boldsymbol{x} &\Leftarrow \boldsymbol{x}_{\mathbf{old}}, \\
\boldsymbol{v} &\Leftarrow -\frac{1}{2}\boldsymbol{v}_{\mathbf{old}}, \\
dt &\Leftarrow \frac{1}{2}dt_{\text{old}}, \\
\kappa^d &\Leftarrow 1.05\kappa^d_{\text{old}}, \text{ and} \\
\boldsymbol{\Delta v} &\Leftarrow -\boldsymbol{\Delta v}_{\mathbf{old}}.
\end{aligned}
\tag{6.30}
$$

This restores the last positions and sets $\boldsymbol{v}$ and $\boldsymbol{\Delta v}$ such as to go back in the simulation even further. Then, the solver is resumed with a smaller time step and higher damping weights. If, on the other hand, from one step to the next step progress was made, but this progress was below a certain threshold, we increase dt by 10% and decrease damping factors by 2%. These values have been validated empirically and prevent the system from diverging. Furthermore, a gain of about 2 points in g is typically achieved—independently of the grid resolution—when compared to the best possible set of static Gauss-Seidel parameters.

6.3.9 GPU Implementation

To achieve interactivity, we have implemented the proposed relaxation process on the GPU. Among the first authors to recognize the GPU's potential to speed up the simulation of mass-spring systems were Georgii and Westermann [GW05] and Tejada and Ertl [TE05]. Specifically for triangle meshes, interactive performance for mid- to large-size networks due to the GPU's high degree of parallelism and superior memory bandwidth can be achieved. The main challenge in GPU-based approaches is to overcome the mu-

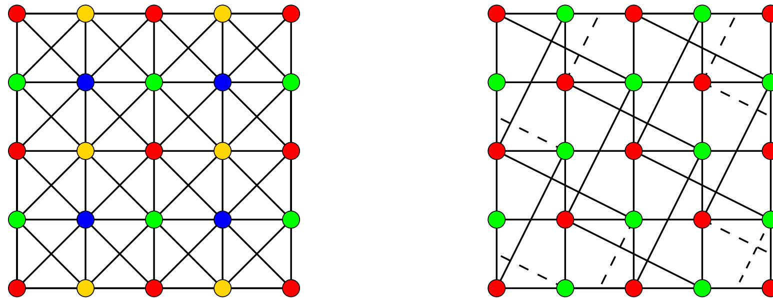


Figure 6.12: *Vertex colorings of various mass-spring networks. The chromatic number reflects the amount of rendering passes per Gauss-Seidel iteration.*

tual exclusion of read/write accesses to the same buffer in current graphics APIs. We use a regular-grid gathering approach. This has the advantage that topological information does not have to be stored explicitly [DCN06].

Algorithm 3 GPU-based Mass-Spring Step

```
Input:
    Initial positions x
    Vertex coloring c
Output:
    Updated positions x

  // Initialization:
  v_x ⇐ 0 ∀x
  Δv_x ⇐ 0 ∀Δv_x
  repeat
      for each vertex color c (rendering pass) do
          for each mass point x of color c do
              Generate a fragment
          end for
          for each fragment (pixel shader) do
              Read positions of mass points in x's 1-ring neighborhood.
              Calculate force f_x acting on x (Hooke's Law & Rayleigh damping).
              Calculate derivatives ∂f/∂x and ∂f/∂v.
              Perform block Gauss-Seidel step to compute Δv_x.
              Trace distance dt (v_x + Δv_x) on the base surface to update position x.
              Re-compute v and Δv based on updated x.
              Write updates x, v, and Δv to an output buffer.
          end for
          Swap old input and output buffers.
      end for
  until converged or maximum number of steps reached
  return x.
```

In contrast to previous work, we use an implicit Gauss-Seidel solver. This requires read access to the $k-1$ previously updated elements in order to update the kth element. Consequently, a naïve implementation requires one separate rendering pass for every vertex. If, however, the topology graph of the spring network can be vertex-colored using N colors, a GPU-implementation can perform a full Gauss-Seidel iteration in N rendering passes. This is because vertices of the same color depend only on vertices of other colors. Consequently, all vertices sharing a color can be processed in parallel. As demonstrated by Figure 6.12, our entangled topology has a chromatic number of 2, while a naïve topology already has a chromatic number of 4. This shows that an efficient

GPU-implementation generally benefits from the proposed entangled topology.

To implement the constrained mass-spring system on the GPU, vertices with different colors are stored in separate buffers. Due to the regular topology, the system matrix of Equation 6.10 does not have to be stored explicitly. Algorithm 3 requires N buffers to store the vertices' position $\boldsymbol{x}$, velocity $\boldsymbol{v}$, and change of velocity $\boldsymbol{\Delta v}$, grouped by color, and another output buffer to store the result (due to mutual R/W exclusion).

The implementation can benefit from computational and memory bandwidth capacities on recent GPUs. Thus, the relaxation process is greatly accelerated, particularly because several iterations of the Gauss-Seidel solver are typically required for convergence. Section 6.5 provides a detailed analysis of the speed-up achieved, including convergence plots. Note that due to the non-optimal order of Gauss-Seidel updates slightly more iterations are needed on the GPU than on the CPU to achieve the same accuracy. In our tests, however, these additional iterations proved to be negligible in terms of performance.

6.4 Rendering

Rendering the base surface with displacements is performed on the GPU by displacing the vertices of the displacement grid along the respective normal of the base surface in a vertex shader. These normals are obtained for each vertex of the displacement grid via barycentric interpolation from the base mesh. In addition, a local tangent frame is interpolated from tangent frames stored at the base mesh's vertices. The interpolated frames are then re-orthogonalized by using the interpolated normal as reference. Since the displacement field also stores a local tangent frame per sample, we can transform the interpolated normal twice using these two frames in order to obtain a properly aligned normal on the displacement mesh for shading. Note that this allows to propagate bump maps from the base mesh to the displacement, which results in a very smooth and natural appearance. The reason is that geometric displacements can be treated as meso-structures that are literally placed between the macro-structure of the base mesh and the micro-structure of bump maps. Less formally speaking, this allows effects like placing a lump "under" the skin of a virtual character.

Positive displacements—i.e., displacement along the outward-pointing normal of the base mesh—can be rendered using standard depth testing. In contrast, *negative* displacements can remove parts of the base mesh. Rendering such displacements can be performed using depth peeling [Eve01]. Since depth peeling significantly reduces performance in case of high depth complexity, we present a new method to render

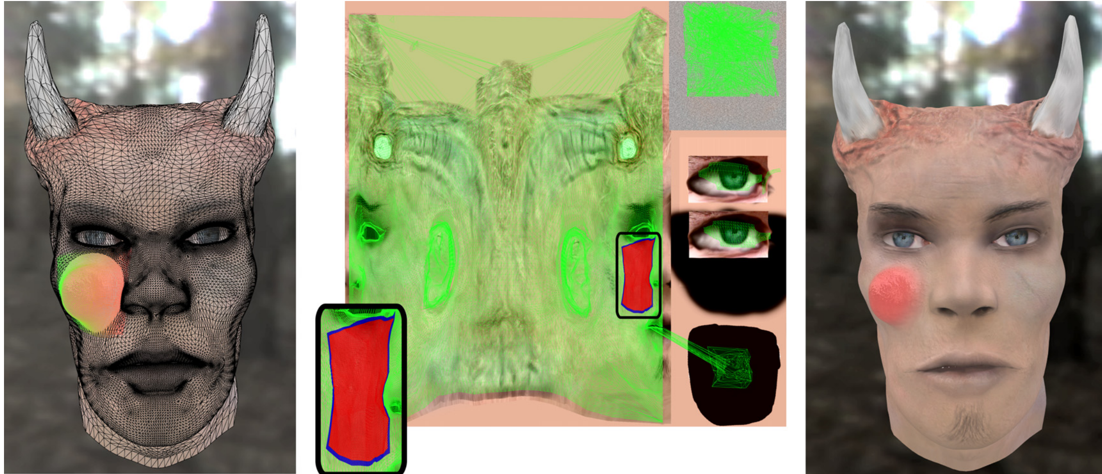


Figure 6.13: *Demonstrating the cutout mask. From left to right: Adding a bump to a mesh, the cutout mask in texture space, and final image, in which the bump is colored red. In the middle, the patch area (red) can be distinguished from the padding necessary to avoid artifacts (blue). Devil Head mesh courtesy of Thomas Heinrich.*

negative displacements by using a *cutout mask*. This mask allows to exclude parts of the base mesh from rendering on a per-fragment basis. It is similar to the *trim texture* used by Guthe et al. in [GBK05], but does not require any re-tesselation.

Assuming that the base mesh is equipped with texture coordinates, generating the cutout mask is performed by rendering the displacement grid into the texture space of the base mesh. If the texture parameterization contains discontinuities, patch polygons have to be clipped against these seams to avoid interpolation artifacts. The patch is rendered into a 2D render target by replacing each vertex coordinate by $(s, t, 0)$, where (s, t) is the texture coordinate carried over from the base mesh. This operation leaves an "imprint" of the triangles in the texture domain. The render target can then be mapped to the base mesh and used to mask out fragments overlapped by the displacement grid. To avoid undesirable artifacts due to the finite resolution of the cutout mask, we slightly increase the size of the patch by padding it with zero. As can be seen in Figure 6.13, although the texel aspect ratio is about 1:2, no artifacts at patch boundaries (red in the middle image) are apparent due to this padding (blue in the middle image). By choosing a sufficiently high resolution for the cutout mask, sub-pixel precision of the cutout region can be achieved easily.

If no global parameterization of the mesh is known, we can still achieve the same result, albeit at a lower rendering speed. In this case, the local parameterization of the decal is used as a domain for the cutout mask. In order to discards all fragments covered

by decals it is thus necessary to access one cutout mask per decal when rendering the base mesh instead of a global cutout mask. Note that all local cutout masks store the same information. It is therefore not necessary to generate them explicitly and a single, static cutout mask can be used.

In a final pass, the displacement grid is rendered and geometry displacements are performed by the vertex shader. Animations of the displacement field can be realized simply by using a sequence of displacements in the final rendering pass.

6.5 Results

To validate the effectiveness and efficiency of our approach, we have conducted several experiments, all of which were performed on an Intel Core2Duo 6600 CPU at 2.4 GHz, with 2 GB RAM and an NVIDIA GeForce 280 GTX. Rendering was performed to a fully anti-aliased 1600×1200 view port using 8 samples per pixel at a quality of 4.

To measure the time needed for the various steps of our algorithm, two models were used. The first one is a chess board that is tesselated using only 12 triangles. The second model is the Mannequin available from Aim @ Shape [AaS], which has been reduced to 32,000 triangles (16,406 vertices). Both models are textured with a decal, a bump map, and an environment map. Figure 6.14 shows both models with interactively placed displacements. We measured the times needed to trace, relax, and render a single displacement grid with resolutions varying from 32^2 to 512^2. Furthermore, we measured the time required to generate a 4096^2 cutout mask. Note that the cutout mask has to be re-generated only if displacements are being moved—not animated—on the base surface. Last but not least, Table 6.2 also provides the time required by a single

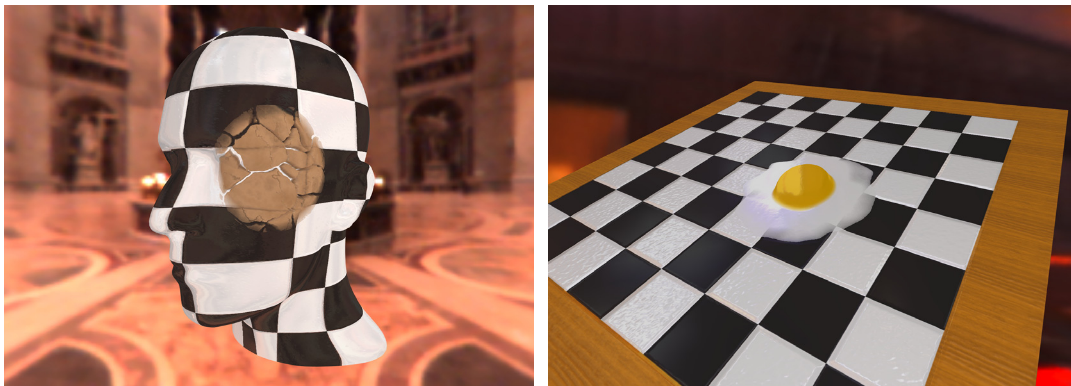


Figure 6.14: *The textured models used to measure performance and quality. Left: Mannequin, 32K triangles. Right: Chess board, 12 triangles. Mannequin mesh courtesy of Aim @ Shape.*

Table 6.2: *Performance evaluation of our displacement method. We provide timings for the Mannequin data set and the chess board data set (the latter ones in parentheses).*

Grid	Layout	Rendering	Cutout Mask	Mass-Spring / step	# steps	#Triangles
-	- (-)	2.0 ms (1.7 ms)	- (-)	- (-)	0 (0)	32 K (12)
32^2	3.2 ms (1.5 ms)	2.2 ms (1.8 ms)	0.3 ms (0.3 ms)	0.05 ms (0.05 ms)	24 (1)	34 K (2 K)
64^2	5.8 ms (4.0 ms)	2.4 ms (2.0 ms)	0.4 ms (0.4 ms)	0.25 ms (0.23 ms)	37 (1)	40 K (8 K)
128^2	18.0 ms (15.2 ms)	3.2 ms (2.8 ms)	0.7 ms (0.7 ms)	0.95 ms (0.89 ms)	53 (1)	64 K (32 K)
256^2	63.8 ms (61.2 ms)	7.0 ms (6.9 ms)	2.2 ms (2.2 ms)	3.08 ms (2.94 ms)	74 (1)	160 K (128 K)
512^2	272.7 ms (234.0 ms)	22.2 ms (20.0 ms)	7.8 ms (7.8 ms)	10.27 ms (9.97 ms)	112 (1)	544 K (512 K)

mass-spring relaxation step as well as the number of iterations needed for convergence. In our experiments, the GPU-based spring-mass system was consistently a factor of about 17 times faster than a carefully tuned CPU version. CPU timings are thus omitted from the table.

It can be seen from the first column in Table 6.2 that computing the layout of the displacement grid—which includes tracing the grid, fold avoidance, and the propagation of appearance properties from the base mesh—strongly affects the overall performance. Specifically, the initial tracing takes significantly longer than several iterations of the GPU spring-mass system. On the chess board, the initial grid can be traced slightly faster, since less crossings of vertices over mesh edges have to be computed. Relaxation of the grid on the chess board converges in a single iteration—which is needed only to compute the residuum. The reason is that the chess board is planar, and consequently all updates in this first iteration are 0.

The second column in Table 6.2 shows that rendering the base mesh with displacements is extremely fast even for high-resolution displacement meshes. The benchmarks include the time required to upload the displacement mesh to the GPU, which means that rendering is even faster as long as the displacements do not move. It is particularly interesting to note that our method is fast enough to handle multiple medium sized displacements at fully interactive rates. For resolutions up to 128^2 even a displacement that moves on the surface can be simulated at fully interactive rates. Compared to previous GPU-based displacement approaches using subdivision [HLW07b], we thus achieve a significant speed-up and we entirely avoid the need for a parameterization and dynamic memory allocation on the GPU.

The convergence analysis of the relaxation process used to improve the local parameterization indicates good behavior and stability of our method. Initially we set all masses to 3.9, damping and stiffness constants to $\kappa_d = 15$ and $\kappa_s = 59$, and the step-

size to $dt = 0.002$. These values have been obtained empirically, and generally show good convergence. It should be noted that κ_d as well as dt are adapted dynamically by our approach.

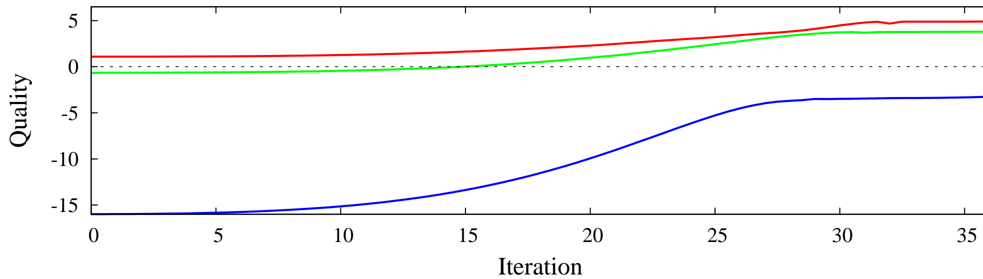


Figure 6.15: *Quality evaluation of grid relaxation using three different grids. For all grids, 36 iterations were sufficient for convergence. The blue curve corresponds to a patch that was placed in a region of high curvature, such that not all folds could be resolved. This results in a lower quality.*

In Figure 6.15, we show convergence plots for mesh relaxation using the mass-spring system. Three 64^2 grids with the entangled topology were placed at different positions on the Mannequin model (also see Figure 6.10). As can be seen, the relaxation process can improve the quality of all three meshes considerably, but on the other hand, the relaxation of the grid that was placed around the nose—indicated by the blue curve—stagnates after a number of iterations. This is due to the high curvature in this area, which prevents folds from being completely resolved.

Using the entangled topology, we observe a performance increase of about 25% for the solver when compared to the commonly used spring network depicted in the right half of Figure 6.12. This stems from the fact that this topology requires less forces to be gathered at each vertex, and that updates can be performed using less rendering passes. Although this specific topology results in a quality that is typically one point less when compared to the other topology—a fact which we mostly attribute to numerical precision—the resulting grid is visually indiscernible.

6.6 Summary

In this chapter, we presented a novel real-time method to simulate geometric displacements on triangle meshes which are independent of the mesh resolution. This is achieved by computing a local ad-hoc surface parameterization in order to map a given displacement field onto the surface.

Novel contributions include the use of a two step approach consisting of an initial

layout phase to obtain a displacement grid while at the same time avoiding folds, and a mass-spring-based relaxation step to optimize this grid towards isometry. We proposed an entangled topology for our spring network which results in an efficient GPU-based implementation using a constrained Gauss-Seidel solver with dynamic step control. By performing a detailed analysis of this solver we showed that it quickly converges and is thus well suited for real time applications. Furthermore, we have presented a pixel-based technique to cut out an arbitrary patch of the base surface to render *negative* displacements that place dents into the base mesh in order to avoid costly depth peeling. Our timings indicate that the method is fast enough to allow interactive placement, editing, animation, and re-positioning of highly detailed displacement maps on the surface.

Our method works best if the geometric features to be added are small in comparison to geometric features of the base surface. However, if the surface has features on a similar scale as the displacements, displacement mapping as such is ill-posed. In the presence of such small features, our method fails to resolve all folds in the initial displacement grid in the worst case. In this context it should also be noted that the local curvature radius naturally limits the maximum height of displacements, but this is a shortcoming shared by essentially all displacement mapping techniques.

In order to render carvings without resorting to depth peeling we intrinsically assumed the existence of a 2D parameterization of the base mesh. If such a parameterization is not possible or not provided, we can still either resort to an ad-hoc texture atlas, or we can utilize local cutout masks as described before. The major drawback of a texture atlas is that in the worst-case each triangle of the base mesh is mapped to a different location in texture space. This, however, requires to clip each displacement grid against each such parameter discontinuity. It is clear, that such a discontinuity meshing severely affects the overall performance if patches are being positioned or moved on the base surface. On the other hand, local cutout masks require to store a dynamic amount of local texture coordinates and imply a texture fetch per displacement for each fragment. As a third alternative it is possible to pre-compute a depth peel of static base surfaces, assuming that such a pre-computed peeling has less discontinuities than a texture atlas. However, this direction was not further pursued; if no 2D parameterization is present the default is a fallback to standard depth-peeling techniques to render negative displacements.

In the future, the following areas still offer potential for additional research. Firstly, GPU-friendly methods to obtain the initial displacement grid, including a method to resolve folds, would be beneficial in order to further reduce CPU workload. This direction

will most likely be supported by the so-called *Compute Shaders* [Boy08] introduced by DirectX 11. Secondly, alternative numerical solvers always have the potential to speed up the process of relaxing the initial grid. So far, we tried a multigrid/CG solver, and the contrained Gauss-Seidel solver discussed in Section 6.3.7. Specifically, the solver used by Bridson et al. [BMF03] for cloth simulation might allow larger step sizes and result in better numerical stability. Again, the implementational burden of more sophisticated solvers could be alleviated by Direct3D's novel Compute Shaders.

Figure 6.16: *Results of the proposed displacement method (I). Left: Demonstrating that using rotations to propagate directions does not fail at acute angles. Right: Augmenting a game character to obtain a highly detailed and naturally looking appearance. The displacement was given a light red hue in order to make it discernible. Devil Head mesh courtesy of Thomas Heinrich.*

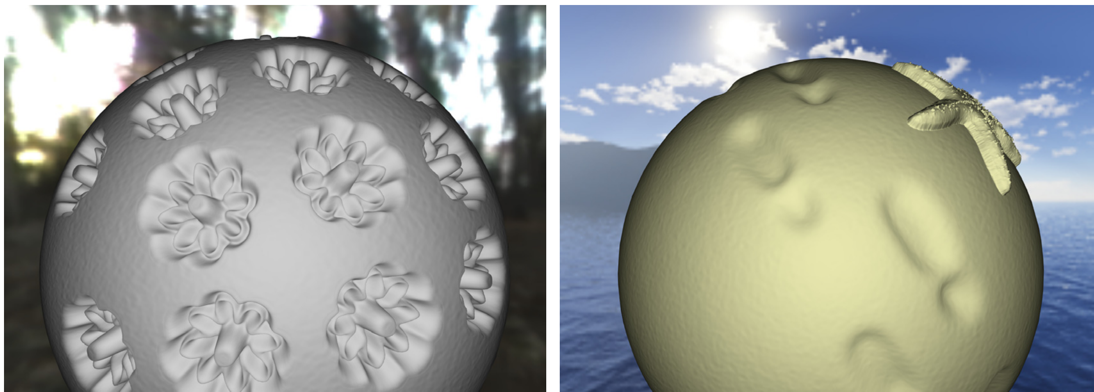


Figure 6.17: *Results of the proposed displacement method (II). Left: About 20 instances of a flower height field are used to displace a sphere. Each displacement has a resolution of* 128^2. *Right: A sphere is displaced by one starfish* (512^2) *and ten footsteps (each* 160×378*). Due to our cutout approach to render carvings, we achieve a rendering performance of about 15 frames per second on a* 1600×1200 *view port for each of these scenes.*

Figure 6.18: *Results of the proposed displacement method (III). A water simulation on a circular domain is used to animate the displacement field. The simulation was pre-computed on a* 256^2 *grid and has 319 time steps. We achieve a rendering performance of over 140 frames per second on a* 1600×1200 *view port. Base mesh courtesy of Aim @ Shape.*

Figure 6.19: *Results of the proposed displacement method (IV). Left: Our method is capable of smooth transitions between base surface and displaced geometry. This is achieved simply by providing height fields with a smooth fall-off to their zero-level. Furthermore, appearance properties such as tangent frames, colors, and texture coordinates can be propagated from the base mesh. Right: Positive and negative displacements at the same time can be handled by our method.*

Chapter 7

Gigapixel Images

Due to recent advances of CCDs (charge-coupled devices, see also Section 2.2) it is by now possible to obtain single-exposure digital images with resolutions in excess of 22 megapixels. Furthermore, these images can be combined to form larger, panoramic views by registration tools frequently bundled by camera vendors with their products. Combined with a broad acceptance of digital photography, especially in the consumer market, this fact has led to image resolutions of tens to a few hundreds of megapixels being available even to ambitioned hobbyists.

While gigapixel images are not yet widespread enough to form a significant market segment, they will eventually replace images of hundreds of megapixels in the same way as megapixel images replaced early resolutions of a few hundred kilopixels. In order to handle images at gigapixel resolutions and beyond, the traditional paradigms for viewing and manipulating digital images have to be redesigned. Although established techniques offer a large catalogue of useful techniques, they might prove completely unfit to deal with the immense amount of data associated with tomorrow's default resolutions.

Consequently, the first gigapixel images have sparked an academic interest, and the great public interest in applications such as Google Earth [Gooa], Google Maps [Goob], and their various commercial counterparts served as an additional catalysator.

7.1 Contribution

In this chapter we describe a framework to view and filter large, highly detailed images *without* changing their binary representation—except for an offline pre-processing step. The benefit of such a system is that multiple copies of the image are avoided. This is in contrast to many traditional, commercially available image manipulation softwares,

that try to keep the image in virtual memory, compute adaptive resolutions on the fly in order to match the user's display resolution, and that typically keep multiple copies to allow *Undo*-functionality.

The major advantage of an approach that does not alter the binary representation of the image can be plainly described by reduced storage requirements, which is of paramount importance for any application handling large data sets. We show that by a GPU-based implementation of filter operations that are evaluated only in a certain region of interest (ROI) such as, e.g., the current view port, even complex filters can be realized without sacrificing interactivity. The result is a gigapixel viewer that also supports rapid prototyping of filters in a WYSIWIG environment.

In order to achieve these benefits, similar concepts as described in previous chapters are employed. To name just a few, we store the image in a tile-quadtree, where each tile is encoded using a custom compression scheme. This reduces bandwidth and storage requirements, albeit at the cost of a pre-processing step. We show that by using a hierarchical vector quantization scheme based on Laplace-pyramids (see also Section 2.7 and Section 2.8) a significant compression at high image fidelities can be achieved *without* sacrificing rendering speed.

The viewer described here offers the standard viewing operations, i.e., pan, zoom, and rotation. To make the interaction user-friendly, we describe an user interface that uses the innovative and intuitive Nintendo WiiMote [Nin] as input device. The main rationale behind choosing the WiiMote over standard input devices such as mouse and keyboard is that we anticipate the future image viewer of choice to be high-definition TV sets powered by so-called set-top boxes[1]. For this setting, a remote control like a WiiMote seems to be the most reasonable and intuitive choice.

Our system addresses all key issues of a gigapixel viewer / manipulator, namely

1. **Space Requirements.** We describe how to compress and page gigapixel images efficiently.

2. **Bandwidth Requirements.** Whenever changes to the *working set*, i.e., the part of the image currently viewed are made, data has to be paged into core memory, uploaded to the GPU, decoded, and rendered. By performing the decoding step directly on the GPU, bandwidth requirements from the external storage device up to the GPU's video memory are efficiently reduced.

3. **Interactivity.** By utilizing a caching and pre-fetching mechanism, we hide latencies of external storage devices. By tapping the GPU's superior processing and

[1] A set-top box usually refers to a multimedia capable, ultra-compact PC that is directly attached to the TV set.

bandwidth capacities we achieve interactivity even for complex image filtering operations.

4. **Usability.** Using the Nintendo WiiMote as the input device of choice for our gigapixel viewer, an intuitive and flexible user interface is designed. Only the development of new filter operations is sufficiently complex to require the use of mouse and keyboard. A tight visual feedback loop allows the user to design filters interactively using a modularized filter graph.

7.2 Related Work

Acquiring gigapixel images is comparably new, even though the aforementioned registration tools have been around for a while. So far, the largest available images are aerial photographs obtained by airborne cameras (see also Section 2.4). For instance,

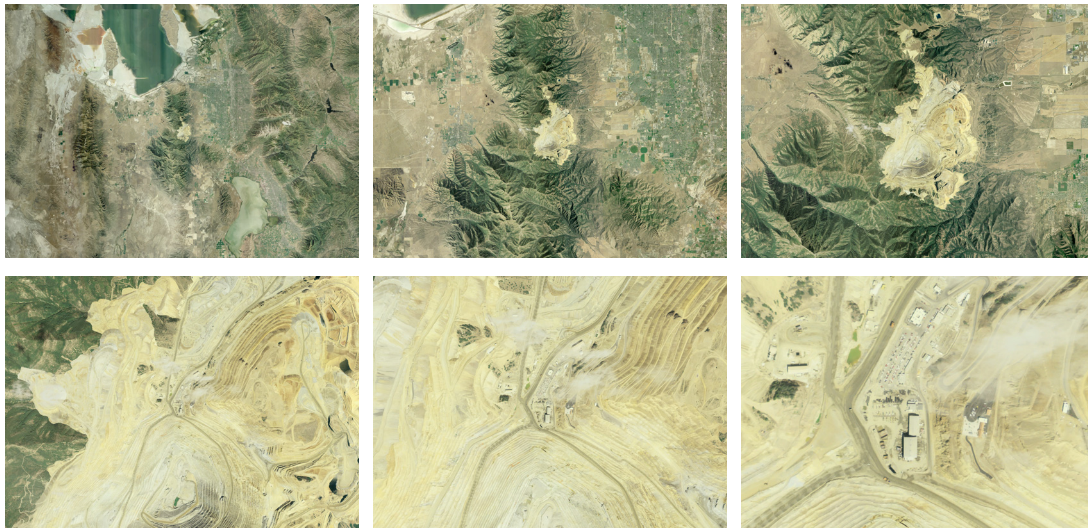


Figure 7.1: *An aerial photograph of the State of Utah, USA. Zoom onto the Kennecott Copper Mine south-east of Salt Lake City. The complete image comprises 228 gigapixels and covers an area of about 565 km $\times$ 435 km. At a bit rate of slightly under 2 bits (including LOD-hierarchy) this image occupies about 75 GB. Data courtesy of US Geological Survey and The State of Utah.*

Figure 7.1 shows a part of an aerial photograph of the entire State of Utah, USA. The image covers an area of about 565 km $\times$ 435 km at a resolution of 1 pixel per meter. The image comprises around 228 gigapixels and occupies close to 685 GB in uncompressed form. As demonstrated by Google Earth, such resolutions are becoming available for many regions on Earth; for some regions resolutions may even be significantly better.

Another source of high-resolution aerial photographs is the Blue Marble Next Generation project of the NASA [ASA] that covers the entire Earth at a resolution of 1 pixel per 500 m × 500 m.

But also cameras mounted on the ground have recently been used to acquire gigapixel images. Most notably, Graham Flint's Gigapxl Project [Fli] uses an analog camera equipped with high-definition, large photographic plates to acquire landscape photographs of up to 4 gigapixels in a single exposure. The image is scanned afterwards to yield a digital representation. The result is an unprecedented amount of detail that cannot be explored on screen without zooming into different regions interactively. Even when viewed on a full HD screen at a resolution of 1920×1080 pixels the finest details can only be observed after zooming in by a factor of 45.

Since single-exposure photographs naturally will hit physical limits, Kopf et al. describe a system to capture gigapixel images by registering several hundres of smaller images [KUDC07]. However, different exposure parameters result in different dynamic ranges for each of the subimages, and—potentially—different tones. The authors therefore describe a system that not only joins these subimages into a single, high-dynamic range image automatically, it also performs color equalization to yield a color-continuous image. Furthermore, they describe a sophisticated viewer that takes multiple viewing projections into account in order to result in a "natural" aspect ratio of the image during viewing.

In contrast, we do not describe any specific data acquisition procedure, and we do not address multiple projections. On the other hand, we address a compression scheme that can be decoded directly by the GPU, and we describe a filtering system that operates directly on the data to be viewed. This system is fundamentally different from the streaming multigrid approach described by Kazhdan and Hoppe [KH08]. While the latter system describes gradient-domain image operations that are employed as a pre-process, i.e., to alter to data prior to viewing in order to perform color adjustments etc., our system is targeted at rapid prototyping of digital image filters.

7.3 Algorithmic Overview

The framework presented here is based on the terrain viewer by Dick, Schneider, and Westermann [DSW08]. The terrain viewer starts with an digital elevation map and an orthographic texture. Then, a Laplace-pyramid is built as described in Section 2.8. This pyramid is cut into tiles, thereby inducing a quadtree on the original data. Each tile is encoded separately. During runtime, tiles are streamed to and decoded by the GPU as

needed.

The major change between the work by Dick et al. and the framework described in this chapter is that geometry has been completely removed from the terrain renderer. Only a single, textured quad is rendered for each tile. In turn, since anisotropic texture filtering is not necessary for an image viewer, a more sophisticated compression scheme is devised in order to achieve bitrates of about 2 bits per pixel at high visual fidelity.

The user can then select a rectangular portion of the view port to define the region of interest (ROI) in which filter operations are performed. Each filter is designed using a module-based, interactive filter graph. An immediate visual feedback of changes made to the filter is achieved by a GPU-based evaluation of the filter graph in the ROI. To provide sufficient support for the filtering operations, the ROI is automatically extruded by a filter-dependent amount of pixels. Furthermore, the ROI is always fixed in screen space. For each pixel inside this region, the filter is evaluated and the filtered image is presented. Outside this region, the original image is displayed.

Filter modules are written in HLSL augmented by a range of registers corresponding to external parameters, input textures, etc. The modular concept lifts the design of new filters to a higher layer of abstraction, since modules can be stored and re-used for later designs.

The system provides several filter templates that classify the type of operation as well as the amount of input and output data streams. These filter templates support log-reduce, propagation, linear, and point operators. Once the filter graph is established the system automatically determines the amount of temporary textures required to store intermediate results by computing the girth of the filter graph. To visualize the filter graph, an automatic layout is computed. The user can then modify the filter by interactively connecting modules with each other.

7.4 Preprocessing

The purpose of the preprocessing step is to transform the image into a compressed LOD-hierarchy, i.e., a quadtree. This tree contains image tiles of equal resolutions N^2. In our implementation, we chose $N = 513$. In an effort to balance disk seek times versus disk transfer rate, we group 8×8 such tiles into a page, which constitutes the smallest block that can be read from disk. Since tiles overlap in one pixel with their respective neighbors to the right and the bottom, each page has an effective resolution of 4096^2 pixels. By interpolating between two levels of this pyramid, the correct level of detail can then be chosen and displayed.

To compress the input image we first compute a Laplace-Pyramid using $m = 2$ and a box kernel, i.e., we recursively compute averages of non-overlapping 2×2 pixel blocks. Unlike the general algorithm described in Section 2.8, we stop this process as soon as the coarsest level (which stores averages of regions in the original image) is smaller or equal to $N - 1$ pixels. This level is then re-quantized to 24 bits per pixel[2]. Additional mipmap levels are computed only for this coarsest level, for reasons that will become clear later. The quantization error of the coarsest image is then propagated to the next level in order to avoid error accumulation.

Since each level has four children which cover a quarter of their parent's extent in object space, each level i but the coarsest one interpolates one fourth of its coarser parent level $i + 1$ bilinearly to obtain a prediction of the data in this image. That is, we use a bilinear interpolation kernel in the expand_m step of Section 2.8. The difference to level i is then encoded by treating non-overlapping blocks of 2×2 RGB-pixels as twelve-dimensional vectors. For these vectors, we start by computing a vector quantization with 2 codevectors. If the residual error between the reconstructed image and the original is above an user-defined threshold, we double the amount of codevectors. This process is repeated until the user-defined error threshold is met. We finally store the number of bits required to represent each index, the codebook, and the index set on disk. This is repeated for all levels and tiles, until the entire image has been encoded.

As an alternative, the image can also be encoded using the S3TC DXT1 format [INH99], but our custom compression scheme typically results in less than 2 bits per pixel while DXT1 uses 4 bits per pixel. It is worth noting that the visual fidelity achieved at this rate rivals JPEG compression at similar rates.

7.5 Rendering

To render the image, the CPU first computes the set of visible tiles as well as the LOD λ for these tiles. We use an orthographic projection in our framework, for which the view direction is always perpendicular to the image plane. Consequently, all visible tiles share the same LOD, which can be trivially computed as the logarithmic ratio of the width of the view port and its projection—measured in pixels—into the object space of the image. By convention, $\lambda = 0$ corresponds to the finest level available, and $\lambda = n-1$ corresponds to the coarsest level. We observe that at any time we need at most two different of the discrete LODs, namely $\lfloor \lambda \rfloor$ and $\lceil \lambda \rceil$. We therefore reconstruct two mipmap levels for each but the coarsest tile and store them on the GPU for further use.

[2]Note that if the input image has a higher dynamic range, the bitrate can be increased.

For the coarsest level, a full mipmap pyramid has been computed in the preprocessing step.

For each visible tile its associated data is fetched from disk and uploaded to a single GPU buffer. Buffers can be accessed in the pixel shader stage at a granularity of 32 bit words. We chose to use this type of memory object instead of multiple textures [SW03] for two reasons. Firstly, each tile can have a different bitrate. If this rate is not a multiple of 8 bits (and it rarely is) storing the compressed data in textures has the undesirable side-effect of wasting lots of bits for padding. This also implies that if tiles are stored densely on disk they have to be traversed on the CPU in order to insert these padding bits. Secondly, using buffers reduces the amount of uploads to the GPU, since codebook and index set can be stored together.

We proceed by sending the bit rate for the current tile to the GPU as a shader constant, as well as an offset (in bits) describing where the index set begins. Refer to Figure 7.2 for the layout of an exemplary buffer. The pixel shader can then reconstruct each 2×2 pixel block by first computing the address of its index using bit arithmetic. Then, up to two 32 bit fetches have to be performed in order to fetch the respective index. After that, another up to two 32 bit fetches yield the codebook entry. Note that we clamp codebook entries to a single signed 8 bit value per channel. Thus, each code-entry comprises 12 bytes, of which only 3 bytes are needed per pixel. It is therefore that two fetches are sufficient to obtain the correct codebook entry.

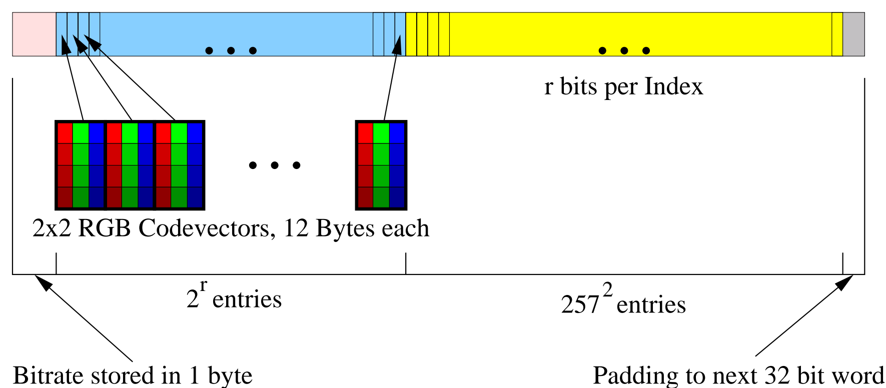


Figure 7.2: *Layout of GPU-buffers containing encoded tiles. Each buffer begins with 1 byte specifying the bit rate r of the indices. Then, $2^r \times 12$ bytes codebook are stored. After that, the 257^2 indices, one per 2×2 pixel block (rounded up) are stored at r bits each. Finally, the buffer is padded to the next 32 bit word.*

To decode a tile at level i, we also need full access to its coarser parent at level $i+1$. Thus, if the parent is not yet resident on the GPU, we load the parent tile (and its parents) recursively. To do so, each newly-loaded tile first checks whether its parent

is resident. If not, it explicitly loads it, thereby resulting in a recursive *pulling* of parents—in the worst case up to the root of the tree. Note that this worst case has only linear complexity in the amount of levels, which is logarithmic in the image resolution.

Once a tile at level i and its parent are resident in GPU memory, the GPU can decode the tile. Parents are decoded first and stored in a 24 bits per pixel RGB format. In a first pass, the quarter of the parent that covers the same extent of the image domain as the tile at level i is then copied to the current tile's second mip-level. In a second pass, the tile's second mip-level is bound as a shader resource, the first, finer mip-level is bound as a render target, and a quad covering all texels of this render target is rendered. For each fragment generated in this manner, a bilinear fetch into the coarser, second mip-level is performed and the difference that was encoded using vector quantization is decoded. These two values are added and written into the render target. Figure 7.3 demonstrates this decoding step.

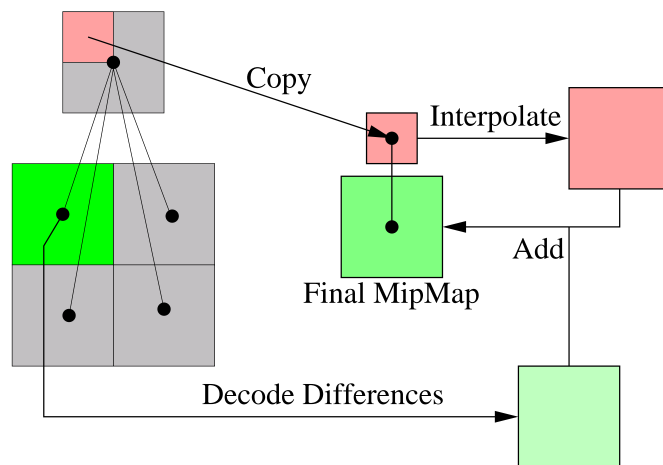


Figure 7.3: *Decoding of image tiles First, the quarter of the parent tile (pink) covering the same portion of the domain as the current tile (green) is copied into the final mipmap's second level. Then, the second level is upscaled bilinearly and the decoded differences of the current tile are added to result in the final mipmap's first level.*

The resulting tiles contain two mip-levels each. Unlike in terrain rendering, an image viewer does not generate overdraw and tiles are not perspectively projected. As a result, on a k megapixel view port, only $\frac{5}{4}k$ texels are required to form the final image. It is easy to see that even for large resolutions such as full HD $(1920 \times 1080$ pixels) only a small working set of about 7.5 MB is required. This leaves lots of GPU memory to cache decoded parents and to perform pre-fetching of tiles.

The pre-fetching works as follows. We determine those tiles that are covered by a circular pre-fetching region around the current center of view. As long as a time slice

Figure 7.4: *Gigapixel landscape rendered using our system. These images demonstrate the large amount of detail available in this 4.5 gigapixel image. The image is compressed at about 2 bits per pixel (including LOD-hierarchy), resulting in a total size of 1.2 GB. The same image encoded using JPEG tiles at a comparable fidelty occupies 1.93 GB. This is partly due to the fact that the JPEG tiles were not encoded hierarchically. Data courtesy of Johannes Kopf and Prof. Dr. Oliver Deussen, University of Konstanz, Germany.*

is not expired, we load and decode tiles of this region, including their parents. Due to the grouping of tiles into larger pages, disk seek times are reduced. The result is that the renderer successfully hides occasional rapid pans.

Each tile within the viewing frustum is then rendered by rasterizing a single, textured quad. Correct filtering is performed automatically, because texels cannot be minified by more than a factor of two due to the dyadic layout of the quadtree.

A final issue that requires discussion in tile-based image rendering are possible artifacts occuring at tile boundaries. To avoid such artifacts, we always store 1 pixel overlap to the right and bottom neighbors of each tile. Since mip resolutions are computed as

$$R(\lambda) = \left\lfloor \left(\frac{1}{2}\right)^{\lambda-\lambda_{\min}} R(\lambda_{\min}) \right\rfloor \quad (7.1)$$

by the Direct3D 10 API, each texture residing on the GPU has a nominal resolution of 514^2 instead of 513^2 to ensure a 1-pixel overlap for $\lambda = 1$. Texture coordinates are now chosen in the range $[0, 1 - \frac{512}{514}]$, thereby interpolating up to the overlapping pixels but not beyond. In theory, this results in a perfect removal of all discontinuities at tile boundaries. However, since the compression scheme used for the tiles may assign different colors to pixels adjacent to a boundary “from right” and “from left”, in some cases minor artifacts may still be visible under close inspection. However, they are not significant enough to be cognitively noticed by the user (see also the zoom sequence in Figure 7.4 which is virtually free of such defects.)

7.6 User Interaction

Figure 7.5: *Photograph of a WiiMote with Nunchuk. The WiiMote (left) is used in for user interaction. It is a low-cost device that is tracked by means of IR-based triangulation and accelerometers. The Nunchuk (right) can be disconnected from the WiiMote and is not used in this work. The image is under the Wikimedia Commons licence.*

To provide the user with an intuitive interface, we use a Nintendo WiiMote [Nin] as input device. Figure 7.5 shows a photograph of such a device. The WiiMote is tracked by means of IR triangulation and accelerometric sensors. It communicates with the Nintendo Wii console using a standard bluetooth protocol. Since it announces itself as a standard Human Interface Device, it can be paired straightforwardly with any desktop PC. The sensoric data can then be read by the PC at a rate of 100 Hz. Specifically, the WiiMote contains an IR camera with a resolution of 1024×768 pixels and a field of view of about $45°$. This camera is connected to a DSP that tracks up to 4 IR sources. In addition to relative IR positions, button and accelerometric state is sent to the PC. In the opposite direction, the PC can send commands to trigger the vibration function of the WiiMote, to switch the four LEDs on the front on or off, and to play sounds.

To ensure correct tracking, Nintendo uses a so-called WiiBar, a device containing 4 IR LEDs that is connected to the Wii by a power-supplying cable. Luckily, third party suppliers offer wireless WiiBars that basically consist of batteries and 4 IR LEDs. Consequently, they can be placed anywhere without the need to be connected to the Wii console. For a 30 inch display, we used two of these wireless WiiBars—one on top of the display and one at the bottom—to compensate for the rather narrow field of view. In our experiments, this setting worked sufficiently well.

To access the WiiMote, we used the publicly available WiiYourself! library [gl.] in combination with the commercial BlueSoleil bluetooth stack [IVT]. The entire setting including WiiMote, wireless WiiBar, and BlueSoleil software costs less than 90 EUR and works flawlessly. We used a point-and-click mechanism as navigation paradigm, i.e., the user points at a specific location in the image and presses the trigger-like button that is conveniently located on the bottom of the device. While the button is pressed, any motion performed with the WiiMote is translated into pans or rotations. Furthermore, by tilting the WiiMote up and down, the user can zoom in or out.

It should be noted that the WiiMote's intended use is not the high-precision tracking desirable for our purposes. Especially the accelerometric data is rather noisy. It is thus necessary to implement a *dead zone*, i.e., a threshold angle to prevent unintended zooming. Furthermore, it is extremely helpful to render a virtual cursor atop the image since in this way the positional imprecision is compensated by the user automatically. This avoids resorting to complex filters to smooth the input data, and results in a very good usability of the system.

The only problem that still needs to be addressed are extremely large displays, for which the distance between IR LEDs is too narrow to offer reasonable precision. In case of such displays, we mandate to build a custom WiiBar replacement—which is a trivial task, given that the only functional parts are batteries and IR LEDs.

In addition to the described interaction mode, we still offer the traditional mouse and keyboard input combination. These devices are indispensable for the design of filters, which is described in the next section.

7.7 Filtering Gigapixel Images

Image filters are an essential building block in many scientific areas. For instance, image understanding seeks to computationally extract features such as edges and salient points and to classify them. Such methods are employed in medicine, robotics, automatic surveillance systems and so forth. But the use of digital image filters is not limited to scientific use. It is an integral part of digital photography, ranging from the built-in DSPs of consumer-class digital cameras used to enhance the quality of the final image to the filters implemented in photo retouching programs.

We define an image as a vector-valued (RGB) function $\mathfrak{I} : \mathcal{D} \to [0,1]^3$ on a finite domain $\mathcal{D} \subset \mathbb{N}^2$. Although RGB values are typically represented by discrete values, we will assume them to be continuous and normalized, since this makes the following discussion easier. Digital image filters are then traditionally classified into the following

groups, which are not exhaustive. For a thorough discussion of digital image filters we would like to refer to the book by Jähne [Jäh05].

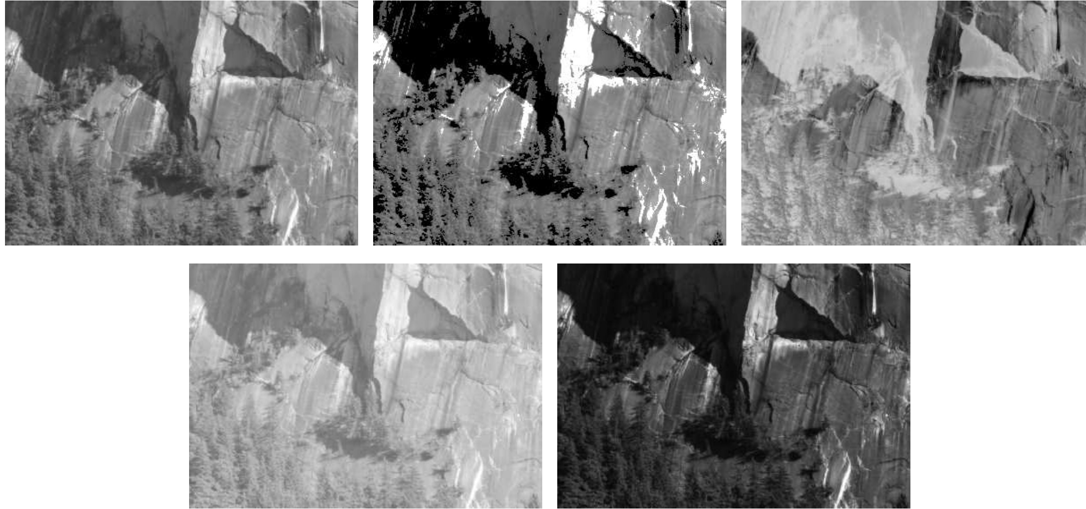


Figure 7.6: *Different homogeneous point operators. Upper row from left to right: Original grayscale image, thresholded image with $a = 0.3$ and $b = 0.7$, negative image. Bottom row from left to right: gamma transformation with $\gamma = 0.5$, and gamma transformation with $\gamma = 2.0$. Data courtesy of Johannes Kopf and Prof. Dr. Oliver Deussen, University of Konstanz, Germany.*

Point operators. Such operators define a mapping $\mathbf{\Phi} : [0,1]^3 \rightarrow [0,1]^3$ which may be further parameterized. The important feature of all point operators is that they can be evaluated for each pixel separately. Examples include color transformations, gray-scale conversion etc. Point operators can be further classified as either being homogeneous or inhomogeneous. Homogeneous point operators are independent of the actual position of the pixel in the image. Examples include thresholding,

$$\mathbf{\Phi_1}(\mathfrak{I}(x,y),a,b) := \begin{cases} \mathbf{0}, & \mathfrak{I}(x,y) \leq a \\ \mathbf{1}, & \mathfrak{I}(x,y) \geq b \\ \mathfrak{I}(x,y) & otherwise \end{cases}, \tag{7.2}$$

negation,

$$\mathbf{\Phi_2}(\mathfrak{I}(x,y)) := \mathbf{1} - \mathfrak{I}(x,y), \tag{7.3}$$

gamma transformation,

$$\mathbf{\Phi_3}(\mathfrak{I}(x,y),\gamma) \quad := \quad \mathfrak{I}^{\gamma}(x,y), \tag{7.4}$$

and many more. Figure 7.6 shows the result of applying $\mathbf{\Phi_1}$ through $\mathbf{\Phi_3}$ to an image.

In contrast, inhomogeneous point operators depend on the position of the pixel in the image. Typical examples include windowing functions, such as the *sine-window*

$$\mathbf{\Phi_4}(\mathfrak{I}(x,y)) \quad := \quad \sin\left(\frac{x\pi}{\mathrm{width}}\right)\sin\left(\frac{y\pi}{\mathrm{height}}\right)\mathfrak{I}(x,y)\ \forall x,y \in [1,\mathrm{width}] \times [1,\mathrm{height}]. \tag{7.5}$$

This mapping "fades" pixels to black towards the border of the image, as depicted in Figure 7.7. Such filters are frequently employed before convolving signals in the frequency domain (not discussed in this thesis) in order to avoid aliasing or frequency leaking artifacts.

Figure 7.7: *Sine-windowing function on an RGB image. Data courtesy of Johannes Kopf and Prof. Dr. Oliver Deussen, University of Konstanz, Germany.*

Linear filters. This group comprises all filters that can be realized using convolution. Discrete convolution is defined as

$$[g \star h](x,y) \quad := \quad \sum_{x'}\sum_{y'} g(x,y)h(x-x',y-y'). \tag{7.6}$$

Typically such filters convolve an image with a significantly smaller *filter kernel* κ. For each non-zero entry in κ a product between the respective filter coefficient and an image pixel has to be performed. These products are then summed in order to yield the final pixel value. For instance, a 3×3 averaging kernel can be denoted by

$$\kappa = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{7.7}$$

Edge detection can be performed by computing partial derivatives of pixel intensities with respect to the image's x- and y-axes. The basic idea is to approximate and threshold the modulus of the intensity gradient, $\|\nabla\mathcal{I}(x,y)\|_2$. The intensity gradient can be computed computed by centered differences, i.e.,

$$\frac{\partial\mathbf{\mathfrak{I}}(x,y)}{\partial x} = \frac{\mathbf{\mathfrak{I}}(x+\Delta x,y)-\mathbf{\mathfrak{I}}(x-\Delta x,y)}{2\Delta x} + \mathcal{O}\left((\Delta x)^3\right). \tag{7.8}$$

This equation can be derived by computing a linear Taylor expansion around (x, y). The corresponding one-dimensional filter to compute the derivative is thus given as

$$\kappa_x = \frac{1}{2\Delta x}\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}, \tag{7.9}$$

where Δx is the distance between two pixels. To achieve an estimate of the gradient that is less sensitive to noise, κ_x can be convolved by a smoothing operation along the y-axis before computing the gradient estimate. This results in the so-called *Sobel-operator*,

$$\kappa_{\text{Sobel},x} = \frac{1}{8}\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}. \tag{7.10}$$

Edge detection along the y-axis works analogously by using $\kappa_{\text{Sobel},y} := \kappa^T_{\text{Sobel},x}$. Figure 7.8 shows a Sobel-filtered image.

Log-Reduce filters. Two-dimensional Log-Reduce filters work in a similar fashion as the tensor-product reduce_m filters described in Section 2.8 in the context of Laplace pyramids. The basic concept stems from the field of parallel computing, where results that are distributed across several machines have to be reduced to a single result in an "as parallel as possible" way. The operation of choice is to first reduce pairs of results, then pairs of pairs of results, and so forth. This requires a logarithmic amount of re-

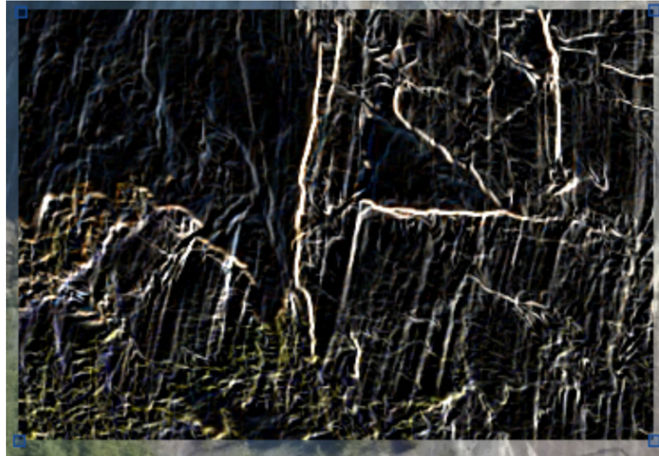


Figure 7.8: *A Sobel-filtered image. Data courtesy of Johannes Kopf and Prof. Dr. Oliver Deussen, University of Konstanz, Germany.*

duce steps. Since GPUs are also highly pixel-parallel, Log-Reduce filters are of great importance in the context of efficient GPU-based filtering systems. If intermediate results are stored, this method can also be used to compute mipmaps. Among the many uses of such filters is the efficient computation of minima and maxima of rectangular texture regions. In GPU lingo, such filters require a logarithmic amount of rendering passes, each of which requires a ping-pong swap due to the mutual read / write exclusion. In each such pass, potentially overlapping blocks of pixels are fetched, reduced to a single value, and written to a smaller output texture. Typically the resolutions of the textures generated in this manner decreases dyadically. Figure 7.9 shows a maximum filter implemented in this way.

Propagation filters. These filters are actually a sub-category of the more commonly known infinite impulse response filters [PTVF02c]. The basic concept is to propagate information in sweeps across the image. The propagation concept allows very powerful filtering operations, such as the Euclidean distance transform described in Section 7.8.

7.7.1 Filter Graph

Our framework provides the user with a visual filter graph. Filter graphs have been employed in a range of multimedia applications by using, for instance, Microsoft's DirectShow API. The basic idea is to compose complex filters from relatively simple, re-usable modules. Figure 7.10 shows a schematic view of a detail-enhancement filter. Using a filter graph results in an intuitive testbed for new filters, since it allows a higher

Figure 7.9: *A Log-Reduce filter to compute the maximum. The maximum of* 128^2 *pixel blocks is obtained in 7 dyadic steps. Data courtesy of Johannes Kopf and Prof. Dr. Oliver Deussen, University of Konstanz, Germany.*

level of abstraction than specifying filters in a single function.

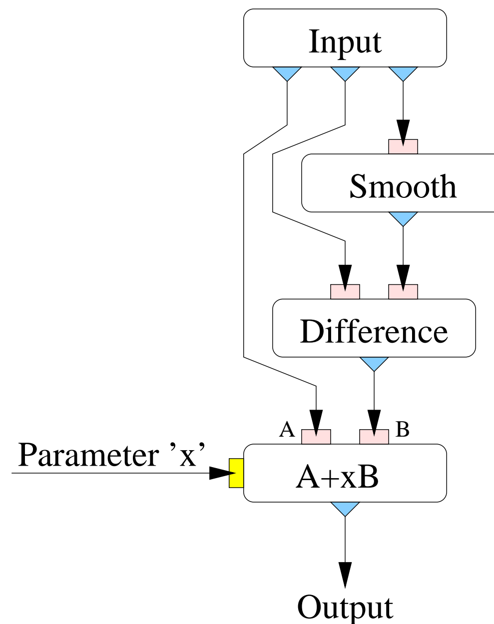


Figure 7.10: *Schematic view of a filter graph. This filter computes details by subtracting the input texture from a smoothed version. These details can then be enhanced* ($x > 0$) *or reduced* ($x < 0$)*.*

An important aspect of the framework is that we provide an editor (depicted in Figure 7.11) in which new modules can be written. Each module is written using an augmented version of Direct3D's High-Level Shading Language (HLSL). To allow access

to filter inputs, user-defined parameters, etc., special registers are introduced. They follow the naming convention "$(*name*)", where *name* is an unique identifier to determine the respective register. Currently, two types of registers are supported. $(FilterInput$i$) denotes the input texture to the current filter module, where i counts from 1. $($i$) denotes the ith user-defined parameter, counting from 1 as well. User-defined parameters are declared in the editor by specifying a name and a floating point range. The system then instantiates a slider for each parameter annotated by the respective name. The slider can then be manipulated within the specified range. The scalar values currently set by these sliders are sent to the GPU as a shader constant and they are used for each subsequent update of the view port.

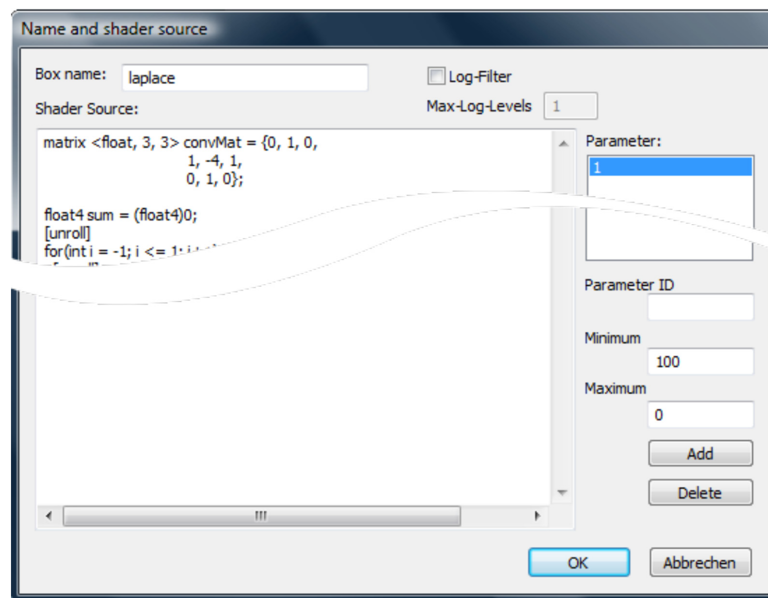


Figure 7.11: *The editor used to write new filter modules. Filter modules are written in DirectX's High Level Shading Language augmented with additional registers. Furthermore, the user can add filter parameters for which sliders are instantiated automatically in the user interface.*

Shader modules are first parsed by our application, replacing all occurences of $(·) by the respective entities which are managed by the application. After that, they are combined into a technique[3] by pre- and appending a header and a footer ASCII file. The technique is then compiled by Direct3D's HLSL compiler and sent to the GPU.

In addition to these registers, certain constants and variables are pre-defined, which are summarized in Table 7.1. Each module has free access to both pre-defined variables and its registers. The return value of each module must always be a single RGBα value.

[3]A technique is a Direct3D concept to group and maintain multiple shaders in one file.

Table 7.1: *Veriables pre-defined by our filtering framework.*

Name	Type	Description
Pos	float4	maps to the HLSL register SV_POSITION
rel_pos	int2	current pixel coordinates relative to the upper left corner of the ROI
pixelsAround	float4	amount of padding around the ROI
filterArea	float2	contains the width and height of the selected ROI
backBufferSize	float2	contains the width and height of the current view port
filterScaling	float2	ratio of filterArea and backBufferSize
Pass	int	current rendering pass (not for point template)
numPasses	int	total number of rendering passes (not for point template)

Each module is stored in a human-readable and -editable ASCII format that specifies further information about the module such as a name, an unique ID, its type, etc. The entire filter graph is also stored in a human-readable ASCII format. This file specifies the header and footer to be used, as well as the amount of pixels about which the user-selected ROI has to be extended in order to provide sufficient support for all filter operations. The source code of the filter modules used is inlined in the file describing the graph, and the connectivity between modules is also stored there. Thus, a complex filter is completely determined by only this file, although modules can still be added or removed.

7.7.2 Filter Templates

Since different filter modules may require different rendering operations, we classify modules by their required rendering operations and provide the following three filter templates. Each module must choose exactly one of these templates.

Point Template. This template is the easiest in terms of the rendering operations performed. A single quad covering all texels of the output texture is rasterized and the shader fragment specified in the module is executed. Since multiple texture fetches can be performed, arbitrary convolutions in the spatial domain can be performed, of which both kinds of point operators, homegeneous and inhomogeneous, can be seen as special cases that have a support of 1 pixel only.

Log-Reduce Template This template recursively fills a mipmap, i.e., a dyadic downsampling is intrinsically performed. Since the number of the current pass as well as the total number of passes that will be performed is important for modules following this template, both numbers are provided by pre-defined shader variables. Beginning with $i = 0$, mip-level i is bound as shader resource and mip-level $i + 1$ as render target. A quad covering all texels of mip-level $i + 1$ is rendered and for each fragment generated in this manner the shader fragment is executed. Then, i is incremented. The shader stops either after a user-defined maximum number of passes, or if the coarsest mip-level has been written. Subsequent modules can then access the *entire* mip pyramid generated in this way.

Propagation Template This template is further parameterized by one of four directions, thereby allowing propagation along the positive and negative x-direction as well as the positive and negative y-direction. Propagation along the x-direction requires $w - 1$ passes, where w is the width of the image. Analogously, propagation along the y-direction requires $h - 1$ passes, where h is the height of the image. The rendering of modules of this template begin by copying the input texture to a temporary texture. This is necessary due to the mutual exclusive read / write access to textures on current GPUs. One of these two identical textures is bound as a shader resource and the other one as a render target. In each pass a single line covering an entire row (y-direction) or column (x-direction) is rasterized into the render target. The shader fragment can then fetch values of the previously updated line or row, combine them into a new value, and write it to the current render target. After this *sweep*, results of odd passes will reside in one texture while results of even passes will reside in the other one. It is thus necessary to merge both textures such that one texture contains both odd and even results in a final pass. Section 7.8 describes how an Euclidean distance transform can be computed using this template.

7.7.3 Filter Graph Ordering and Resource Management

To layout the filter graph, we compute an ordering that allows us to place each module below any of its predecessors. If a module m_2's input depends in any way on another module m_1's output, then m_1 is a predecessor of m_2, denoted $m_1 \lhd m_2$. Note that this does not necessarily imply that m_2 is directly connected to m_1, it just implies that there is a path beginning at an output slot of m_1 that ends at an input slot of m_2. Consequently we seek to obtain an ordering where $m_1 \lhd m_2 \lhd \ldots \lhd m_n$. The modules in this ordering can then be traversed linearly and visualized from top to bottom without unnecessary

cluttering of edges. Since $\lhd$ is not a binary sorting predicate, this ordering is best computed by a method similar to selection sort. We first traverse all modules and select the module m that has no predecessors. m is then removed from the list of modules, and we proceed with the remaining modules until the ordering has been established. Note that this method is not optimal with respect to its runtime, but since the amount of modules in typical filter graphs is rather small and due to its implementational ease we chose this particular algorithm.

This ordering can be directly re-used for automated resource management, since the graph obtained by this ordering is the serialized dependency graph of the filter. A texture storing temporary results from a module m_1 is only needed for as long as there exist at least one other module m_2 that has not yet been evaluated such that $m_1 \lhd m_2$. Thus, the serialized filter graph can be traversed linearly for each output texture, and textures can be recycled by a simple greedy strategy. Consequently, the amount of temporary textures used corresponds to the girth of the serialized dependency graph.

7.8 Euclidean Distance Transforms Using Propagation Filters

In this section we discuss the implementation of an approximate discrete Euclidean distance transform that is based on propagation filters. The resulting distance fields have a very low error probability, and for each error a precise upper bound exists. The section will closely follow our recent publication [SKW09], but whereas the original paper discusses a stand-alone algorithm we also discuss its integration into the more general gigapixel filtering pipeline.

The demand for such Euclidean distance transforms should be clear, since algorithms that depend on distance transforms [RP66] or Voronoi diagrams [Vor08] seem to be ubiquitous. For instance, the automatic analyzation of real-time video images at ever increasing resolutions, medical data processing, and artistic applications are just a few examples of a widely established technique. In nearly all cases that require distance transforms, algorithms capable of achieving throughputs of several million pixels per second are highly advantageous. We show that these high throughputs can be achieved by exploting the GPU's superior memory bandwidth and computing power. The algorithm presented here is based on the vector propagation paradigm proposed by Danielsson [Dan80], however, in order to tap the GPUs full potential, certain SIMD-programming paradigms have to be used and the original algorithm has to be reformulated in a data-parallel way.

7.8.1 Related Work

Since Euclidean distance problems have been well studies, especially during the last two decades, we provide a short overview of work specifically related to this field in this section. For an exhaustive review of prior art we would like to refer the reader to [JBS06, Cui99]. Furthermore, a broad overview of the construction and applications of Voronoi diagrams is provided in [Aur91, OBSC99].

Considerable effort has been spent in order to accelerate the computation of distance transforms as much as possible. The most promising algorithms approximate or solve the aforementioned problems by using a sweeping strategy in $\mathcal{O}(N)$ [Dan80, Mul92, SJ01], where N is the amount of pixels in the image. In contrast, algorithms following the wavefront propagation principle such as the fast marching method [Tsi95, Set96, HPCD96] typically result in a complexity of $\mathcal{O}\left(\max(N, k \cdot \log_2 k)\right)$, where k is the amount of *features*, or Voronoi-sites.

Among the first approaches to approximate the distance transform were those that replace the Euclidean distance metric by more tractable ones such as the Manhattan distance [TvW02], chamfer metrics [RP66, BM98, SB02], or octagonal metrics [KK79]. Especially chamfer metrics allow for a trade-off between performance and error, but the distance fields computed with these metrics may not be acceptable in some cases due to the inherent approximation errors.

Another class of methods tries to generate a distance transform that is accurate for virtually all pixels with only spurious errors. The most prominent example is called *vector propagation* [Dan80]. Although being conceptionally simple, highly accurate results can be achieved with good performance [JBS06]. These methods store a vector-valued pointer to a feature candidate for each pixel. These pointers are then propagated using a structuring element called *vector template*. Multiple such templates are sweeped in a simple fashion across the image. Danielsson describes two methods, 4SED and 8SED (SED being an acronym for *sequential Euclidean distance*), that effectively operate on a von Neumann- and a Moore-neighborhood. 4SED is obviously faster and results in larger approximation errors.

Recently a practical algorithm to compute a precise discrete distance transform in $\mathcal{O}(N)$ was proposed [MQR03]. However, this algorithm relies on frequent concurrent read/write accesses—a very limited feature on GPUs that is not yet exposed in standard graphics APIs. It should be mentioned, however, that the compute shader in the upcoming DirectX 11 API will offer such memory access [Boy08].

On a different avenue the use of GPUs has been mandated by several authors. The potential of GPUs for various computational geometry tasks is discussed in [Den03].

Closely in style to the continuous sweepline algorithm [For86], the use of triangle meshes to model a local distance field around each feature is proposed in [HTCLM99]. Hardware depth-testing is exploited during rendering these meshes to generate a generalized Voronoi diagram. The distance transform can then be obtained from the depth buffer. For applications that only need a distance transform in a shell around features, variations of wavefront propagation methods have been shown to be highly efficient. Using graphics hardware, such methods extrude features to prisms and wedges which can be scan-converted efficiently [Mau03, SPG03]. Although these approaches generate precise results, they rely on generating triangle meshes and/or volumetric primitives, and their complexity is not independent of the number of features. To avoid excessive rasterization of distance meshes, a GPU-based framework to compute 3D distance transforms using slice-based culling and clamping was proposed in [SOM04]. Splatting the distance functions for each feature point [ST04] avoids the generation of meshes, but although even skeletons can be constructed this way, these approaches tend to be severely fill-rate-bound due to overdraws.

In [RT06] the jump flooding paradigm was presented, a communication pattern to quickly propagate information in highly SIMD-parallel computing environments such as GPUs. This method is among the most promising ways to compute distance transforms and generalized Voronoi diagrams since it offers a flexible trade-off between precision and speed.

7.8.2 Problem Description

Throughout the description the notion of a *feature* will be used to describe the geometric entities that will eventually become Voronoi sites. Features are distinguished by pairwise different IDs. In case of the classical Voronoi diagram, features are points. Among the generalizations commonly made, one allows lines and curve-segments as features. To be able to construct such generalized Voronoi diagrams (see also Figure 7.12, leftmost image), we extend the notion of a feature to refer to any non-empty set of (potentially disconnected) points that share an ID.

Given a set of points

$$P \;:=\; \{\boldsymbol{p_i}\}_{i=1}^{N} \subset \mathbb{R}^n \tag{7.11}$$

and a set of features

$$S \;:=\; \{F_j\}_{j=1}^{k}\,,\; F_j \subseteq P, \tag{7.12}$$

Figure 7.12: *Different Voronoi diagrams. From left to right: A generalized Voronoi diagram using points and curves as sites, an artistic Voronoi-based mosaicking filter using Gaussian-distributed sites, and a Voronoi diagram consisting of first- (red lines) and second-order (green lines) neighbor regions (please refer to the electronic version). Each image has a resolution of* 1600×1200 *and was generated completely on a GPU. The two left images took less than 22 ms, the rightmost image took less than 31 ms.*

an algorithm that computes a scalar field

$$\Phi(\boldsymbol{p_i}) \quad := \quad \min_{j \in \{1,\ldots,k\}} \min_{\boldsymbol{f} \in F_j} \|\boldsymbol{p_i} - \boldsymbol{f}\|_2 \tag{7.13}$$

is said to compute a discrete Euclidean distance transform of (P, S). Note that according to the definition of S, all points used as a feature are contained in P, which is a convention that does not affect generality. An algorithm that computes a labeling

$$L(\boldsymbol{p_i}) \quad := \quad \text{argmin}_{j \in \{1,\ldots,k\}} \min_{\boldsymbol{f} \in F_j} \|\boldsymbol{p_i} - \boldsymbol{f}\|_2 \tag{7.14}$$

is said to compute a (generalized) discrete Voronoi diagram of (P, S). These two problems are closely related; in fact Equation (7.13) and Equation (7.14) can be turned directly into a naïve algorithm with a runtime complexity of $\mathcal{O}\left(N \cdot |\cup_{j \in \{1,\ldots,k\}} F_j|\right)$ to compute both. Note that in the continuous case a practical algorithm with a runtime complexity of $\mathcal{O}\left(k \cdot log_2 k\right)$ is only known for the classical Voronoi diagram. Since the bounding curves and surfaces of the regions of continuous generalized Voronoi diagrams can be algebraic surfaces of arbitrary degree, a practical algorithm is not known.

7.8.3 Basic Algorithm

We first review the original vector propagation algorithm by Danielsson [Dan80] before adressing the changes necessary in order to execute the algorithm on the GPU efficiently. These changes then give rise to a more general *propagation filter paradigm* that can be seamlessly integrated into the gigapixel filtering pipe. For simplicity's sake, we will first assume all features to be single points and extend this restriction later to

the generalized case.

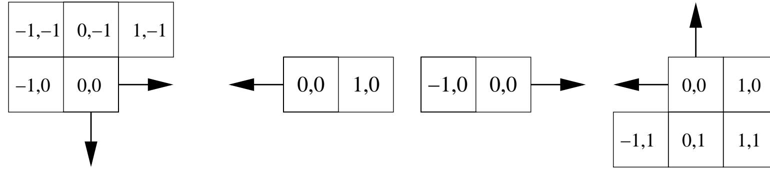


Figure 7.13: *8SED vector templates. These templates were first proposed by Danielsson [Dan80].*

Given an $N \times M$ image of quadratic pixels $P := \{(i, j)\} \equiv \{1, \ldots, N\} \times \{1, \ldots, M\}$, a set of features $S \subseteq P$, and a set of vector templates $T := \{\{(k, l)\} \subset \mathbb{Z}^2\}$, where $\{(k, l)\}$ specifies pixel offsets belonging to one template, vector propagation works as described in Algorithm 4.

Algorithm 4 Vector Propagation algorithm

Input:

Image $P := \{(i, j)\} \equiv \{1, \ldots, N\} \times \{1, \ldots, M\}$
Set of features $S \subseteq P$
Set of vector templates $T := \{\{(k, l)\} \subset \mathbb{Z}^2\}$

Output:

Euclidean distance transform of P

// Initialization:
for each $(i, j) \in P$ do
  if $(i, j) \in S$ then
    $\boldsymbol{v}(i, j) \Leftarrow (i, j)$
  else
    $\boldsymbol{v}(i, j) \Leftarrow (\infty, \infty)$
  end if
end for
// Propagation
for each $\boldsymbol{t} \in T$ do
  // Sweep all pixels
  for each $(i, j) \in P$ do
    // Propagation update
    $\boldsymbol{v}(i, j) \leftarrow \boldsymbol{v}\left(\text{argmin}_{(l,m)\in \boldsymbol{t}} d_{l,m} + (i, j)\right)$,
    where $d_{l,m} := \|\boldsymbol{v}(i + l, j + m) - (i, j)\|_2$.
  end for
end for
return $\|\boldsymbol{v}(i, j) - (i, j)\|_2$ for each $(i, j) \in P$.

Note that the sweeping passes depend on the current template's shape. Each of the

propagation updates computes a new best candidate for the feature closest to (i, j) by scanning the neighborhood defined by the template $\boldsymbol{t}$ around (i, j) for possible candidates. The templates originally used for 8SED are depicted in Figure 7.13. The vectors in each cell denote the offset to the current pixel (i, j), since this is the distance that has to be added to the current candidate of the respective cell to compute its distance to (i, j) (hence (i, j) corresponds to the cell marked $0, 0$). The arrows on the templates indicate the sweep direction, i.e., the leftmost template can be advanced from left to right and top to bottom in either a row-major or column-major sweep.

7.8.4 GPU-based Implementation

The problem with the original vector templates is that two row-major or column-major sweeps are required. Such sweeps cannot be parallelized efficiently. A simple modification however will result in a sweepline algorithm that can be efficiently implemented on a SIMD-parallel GPU, albeit at the cost of a slightly higher (by about 11%) memory bandwidth usage. This modification is shown in Figure 7.14.

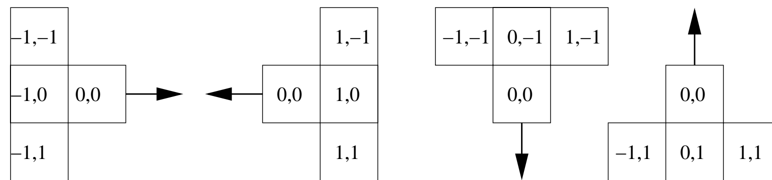


Figure 7.14: *Modified vector templates. These templaces can be swept in four simple line-sweeps.*

We begin by storing the original image in an ID-texture using a 32 bit integer per pixel. Each pixel stores an ID>0 if it is a feature and 0 otherwise. Furthermore, we need two textures for the vector propagation—since using standard graphics APIs read and write accesses are mutually exclusive—to store a 2D vector. We chose a format of 2×16 bit unsigned integers per pixel. Initialization proceeds as described by Algorithm 4. More precisely, we bind both textures as render targets and render a quad covering all texels. For each texel we then perform a texture lookup into the ID-texture.

If the ID for the respective pixel is 0, we store $(2^{16} - 1, 2^{16} - 1)$, which is the largest possible number in the chosen format. Otherwise, we store the fragment's 2D position in pixel coordinates (i.e., in the range $[0 \ldots N - 1] \times [0 \ldots M - 1]$). Prior to performing the actual vector propagation we generate all necessary sweeplines in a single vertex buffer. This buffer can be recycled for all input images of the same resolution. In this way, frequent costly allocation of vertex buffers is avoided.

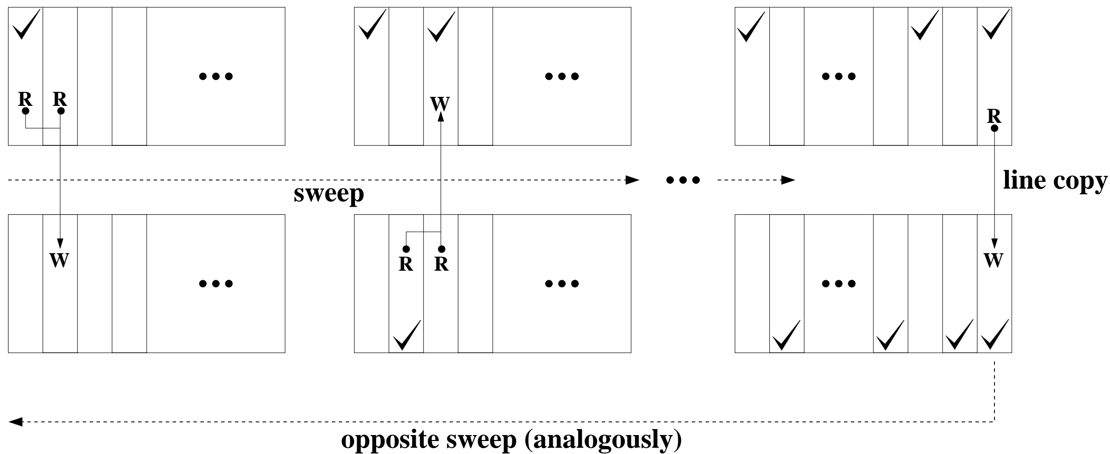


Figure 7.15: *Ping-pong buffering for propagation filtering. Mutual read/write exclusion on GPUs leads to the so-called* ping-pong buffering*. The first sweep reads from two lines of a* shader resource *texture and writes to a single line of a* render target *texture. Updated lines are indicated by ticks. Before the opposite sweep commences, a single line must be copied in order to ensure that updated information is properly propagated.*

We then start by binding one of the now-initialized textures as a (read-only) *shader resource*, and the other one as a (write-only) *render target*. A single line is then rasterized to cover a single row (vertical sweeps) or column (horizontal sweeps) of texels of the render target, thereby allowing SIMD-parallel processing. For each fragment generated in this way, four texels corresponding to the current template are fetched from the source texture in a pixel shader. From these texels a new best candidate is computed according to the propagation algorithm. The result is written to the render target. After each line a *ping-pong swap* is performed to exchange shader resource and render target. The sweepline is then advanced by one texel, and the sweep proceeds until the end of the texture is reached.

After each sweep one of the two textures will contain all updated even lines while the other will contain all odd updates (see Figure 7.15). Normally this requires a merge operation prior to the next sweep. However, by grouping sweeps with opposite directions into pairs the merge operation is reduced to a single line copy.

In this way, textures have to be merged only after each pair of sweeps by rendering a quad that covers the entire destination texture. For each fragment thus generated, a pixel shader discards every second fragment in order not to overwrite the updates rows or columns in the render target. All surviving fragments just copy their value from the source texture. After this merge operation is completed, it is repeated analogously to

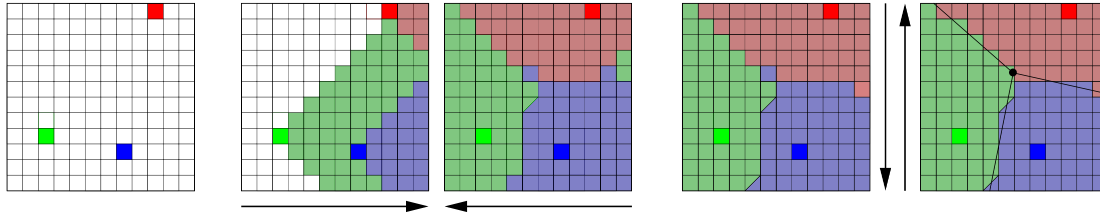


Figure 7.16: *Example computation of a Voronoi diagram. From left to right: Original image with three features, results after sweep to the right and left, and results after sweep down and up. In the final image, a precise continuous Voronoi diagram has been overlaid. Pixels that are colored using green/blue can be associated with green or blue, since they have exactly the same distance to the respective features.*

update the other texture.

Once the propagation is finished, each fragment's ID is obtained by a simple lookup into the ID-texture. Boundaries of Voronoi regions fall between pixels where IDs change. The distance transform is obtained by re-computing the distance between the closest feature and the fragment's position for each fragment. By assigning the same ID to multiple pixels in the ID-texture, generalized Voronoi diagrams are obtained. Furthemore, by propagating k best candidates and sorting them in each propagation update, $k-$NN Voronoi diagrams [Cui99] can be generated that have been employed in procedural texturing and modeling [Ols04]. The rightmost image in Figure 7.12 shows such a diagram for $k = 2$, where brightness corresponds to the difference in distance between the second nearest and the nearest feature. As a result, the brightness is strictly positive everywhere except at first-order Voronoi boundaries where it vanishes.

The result of a complete run of this algorithm is illustrated in Figure 7.16. Each diagram shows the classification of pixels after each sweep, including immediate merging of the two partial ping-pong results. After the first sweep features "fan out" at a $90°$ angle to the right. The sweep in the opposite direction is not able to correctly classify the two green cells to the right, since they do not have any candidates to choose from except for themselves. Note that such cases will always be removed with the next sweep and that such "islands" cannot occur at the line at which the last sweep begins, since one of the three prior sweeps would have removed them. In this example two pixels have the same distance to the blue and the green features. Their final classification is dependent on the sweep- and the computation-order.

7.8.5 Integration into the Gigapixel Filter Pipeline

From the discussion of the rendering operations involved for computing the distance transform it is immediately clear that the Propagation template is custom-tailored for this problem. Consequently this method is integrated into the gigapixel filtering pipeline straightforwardly. However, the merge operation that was described as a necessary step for the Propagation template can significantly slow down the computation of distance transforms. We thus check for all occurences where all modules succeeding a sweep perform a sweep into the opposite direction. If this is the case, the merging step can safely be skipped. Otherwise it must be performed. This simple change to the way the ropagation template works ensures that no unneccessary merging of data is performed.

7.8.6 Errors in 2D Vector Propagation

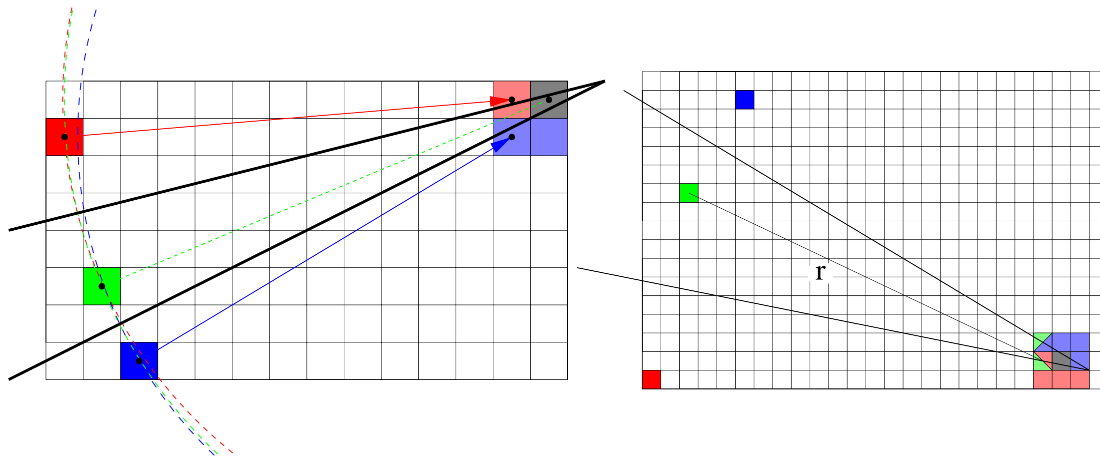


Figure 7.17: *Worst-case error analysis for 2D vector propagation. In both images, the gray point should be associated with the green feature. However the gray pixel's sight to the green feature is obstructed by direct neighbors that are closer or equally close to other features.*

Errors in vector propagation only occur if a pixel cannot be "reached" by its closest feature during propagation. This means there is a pixel whose entire Moore-neighborhood points to other features. Such a situation is depicted in the left part of Figure 7.17. The gray pixel in the upper right is closest to the green feature, but cannot be reached because all its neighbors are closer to other features. In terms of Voronoi regions (bold black lines in the figure) this means the existence of a Voronoi region that contains the center of a pixel but no center of any of its neighbors. Consequently, circles around each of the "obstructing" points through their associated feature must not con-

tain the feature that would be correct for the mis-classified pixel. As a conclusion, the closer the actual feature of the mis-classified pixel is to these circles, the higher the worst-case *relative* error. The relative error can be shown to be less or equal to $\left(\sqrt{170} - \sqrt{169}\right)/\sqrt{169} < 0.3\%$ [Cui99]. This case is depicted in the left half of Figure 7.17: the correct distance between the green feature and the gray pixel would be $\sqrt{169}$, while it is falsely asigned a value of $\sqrt{170}$.

In [Dan80] the maximum *absolute* error was computed as

$$\begin{aligned} \varepsilon_{\max}(r) &= r + \gamma - \sqrt{r^2 - \gamma}, \text{ where} \\ \gamma &\approx 1 - \cos 24.4698^\circ. \end{aligned} \tag{7.15}$$

Here, r is the distance between the correct feature of a mis-classified pixel and an obstructing pixel. Thus, the error can be bound by

$$\lim_{r \to \infty} \varepsilon_{\max}(r) = \gamma \approx 0.08982 \text{ pixels.} \tag{7.16}$$

However, in our tests we found a larger absolute error for our algorithm. Running an exhaustive search on all configurations of three features on a 32×32 image, we found the error to be bounded by

$$\varepsilon_{\max} \leq \sqrt{485} - \sqrt{481} \approx 0.091033 \text{ pixels.} \tag{7.17}$$

The corresponding case is shown in the right half of Figure 7.17. It is a very pathological case, though, since two of the obstructing pixels are equally far from the green feature and either the blue or the red one. Nevertheless, depending on the propagation order this can lead to the observed error. Note also that in this case Danielsson's assumption that the mis-classified pixel is assigned the value $r + 1$ is no longer valid. Since the absolute error decreases with increasing r, this case results in the largest maximum error possible.

7.8.7 Performance of the Euclidean Distance Transform

In this section we provide results and perform a thorough comparison to the jump flooding algorithm (JFA) [RT06]. Although other GPU-based methods have been proposed recently, e.g., the fast hierarchical algorithm (FHA) [CK07], in our opinion JFA offers the best trade-off between speed and approximation error among all previous approaches.

Bandwidth & Runtime Complexity. First we will compute the memory traffic caused by our method for an image of resolution N^2, since this is a major limiting factor. It is assumed that all references to features will be stored as 2×16 bit integer values. Each read and write access will be counted separately.

During line-sweeps, for each rasterized pixel four vectors are read and one is written. There are $(N-1) \cdot N \approx N^2$ intermediate output pixels per sweep. Furthermore, after each pair of sweeps, a merge-operation is necessary. This operation reads a total of $N^2/2$ pixels from one texture and copies them to another buffer. Since this has to be performed in both directions, it results in a total of $2 \cdot N^2$ accesses. For two pairs of sweeps less than $(2 \cdot 2 \cdot 5 + 2)N^2 = 22N^2$ 32 bit accesses are made, thus resulting in less than 88 bytes of memory traffic per pixel.

In comparison, JFA requires $\log_2 N$ passes, each writing N^2 intermediate output pixels. Per pixel, a total of 9 values (modulo boundary cases) is read. Hence, JFA results in about $\log_2 N \cdot N^2 \cdot (9+1)$ memory accesses, or less than $40 \cdot \log_2 N$ bytes per pixel. Consequently, our method is less likely to become bandwidth-limited than JFA for large images, since its traffic per pixel is independent of the image resolution.

Our method compares four distances per intermediate output pixel multiplied by four sweeps, while JFA requires nine comparisons per intermediate output pixel. Thus, the theoretical complexity of our method is $\mathcal{O}\left(16 \times N^2\right)$ and $\mathcal{O}\left(9 \times N^2 \cdot \log_2 N\right)$ for JFA, where N^2 is the image resolution.

However, it should be noted that the 2D JFA can achieve competitive results, since it generally exploits GPU parallelism better than 2D vector propagation.

Empirical Validation. All tests presented here were run on an Intel Core2Duo 6600 processor clocked at 2.4 GHz running Windows Vista. The machine was equipped with 2 GB DDR2 RAM and an NVIDIA GeForce 8800 GTX with 768 MB of video RAM. The CPU version of our algorithm is carefully hand-tuned and runs on a single core to maximize caching benefits. We were able to run the jump flooding algorithm (JFA) [RT06] on the very same machine achieving about 185 fps for a resolution of 512^2. This corresponds to roughly 46.25 Mpixels/sec. JFA is likely to perform differently in other resolutions, but sadly the original OpenGL-based application is locked at 512^2 pixels. Since the timings for JFA are incomplete, they are omitted from Table 7.2.

Most notable in the results displayed in Table 7.2 is the sudden decrease in CPU performance at resolutions of 2048^2 and beyond which is due to cache limitations. Since we store images on the CPU in x-major order, at a resolution of 2048^2 sweeps in the x-direction are about five times as expensive as sweeps in the y-direction. The

Table 7.2: *Performance of the Eucliden distance transform. We specify both the time per frame in milliseconds and the achieved pixel rate in pixels per second (1 Mpixel = 2^{20} pixels).*

Resolution	CPU time	CPU pixel rate	GPU time	GPU pixel rate	GPU Speedup
128^2	1.04 ms	14.96 Mpixel/sec	2.50 ms	6.23 Mpixel/sec	$0.42\times$
256^2	4.60 ms	13.60 Mpixel/sec	4.21 ms	14.84 Mpixel/sec	$1.09\times$
512^2	20.02 ms	12.49 Mpixel/sec	7.42 ms	33.68 Mpixel/sec	$2.70\times$
1024^2	91.83 ms	10.89 Mpixel/sec	15.14 ms	66.02 Mpixel/sec	$6.06\times$
2048^2	696.6 ms	5.74 Mpixel/sec	41.68 ms	95.96 Mpixel/sec	$16.72\times$
4096^2	2751 ms	5.82 Mpixel/sec	186.5 ms	85.79 Mpixel/sec	$14.74\times$
8192^2	11366 ms	5.63 Mpixel/sec	1262 ms	50.70 Mpixel/sec	$9.00\times$

reason is that sweeps in the y-direction are perfectly cache-coherent since in this case x-rows can be processed sequentially. Different storage layouts (i.e., block-major or Z-order) could alleviate this problem to a certain extent, but were not investigated.

On the GPU, caching issues only occur at 4096^2 and beyond, and they are by far less severe than on the CPU. On the other hand, for small resolutions the GPU's performance is comparable to the CPU implementation or even less. The reason is that in this case the GPU suffers from draw-call overheads and the relatively small amount of parallelism due to the short lines being rasterized. For our purpose, we can run full-screen filtering on a 2560×1600 Apple Cinema display at about 23 fps, which is sufficient for interactive exploration most of the time. If the user desires higher frame rates, a smaller area can still be selected for filtering.

To validate the likelihood of errors to occur and to measure the magnitude of errors, we reproduced the experiment of [RT06]. Our method was run on images of a resolution of 512^2 that were randomly filled with varying amounts of Laplacian-distributed features. Over 10,000 runs were generated for amounts of features between 100 and 10,000. From 100 to 5,000, the amount of features was varied in steps by 100, and between 5,000 and 10,000 in steps of 250. As can be seen in Figure 7.18, one of the most interesting properties of this algorithm is that the pathological cases leading to errors require a lot of empty area and a very specific configuration of spurious features. Consequently, with increasing amounts of features, the number of errors decreases. This is especially useful for applications seeking to compute distance transforms of contours, since errors are extremely unlikely to occur in this setting. For random distributions of features the error rate was less than 0.56 per Mpixel. The maximum absolute error that occured was exactly $\sqrt{485} - \sqrt{481}$ pixels, as discussed in Section 7.8.6. The corre-

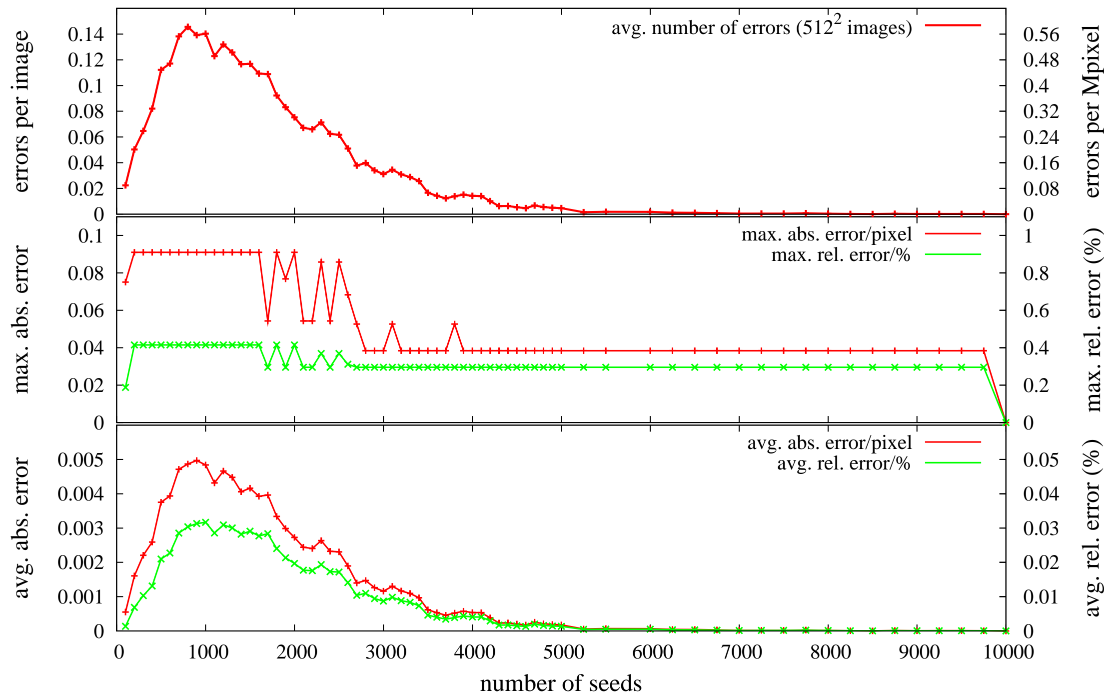


Figure 7.18: *Empirical error analysis. Top: Likelihood of an error to occur for different amounts of features. Middle: Maximum absolute and relative errors. Bottom: Average absolute and relative errors.*

sponding relative error is about 0.3%. Also, the average error that occured was about one order of magnitude smaller, as can be seen in the bottom diagram. This further indicates that the maximum error is highly unlikely. Furthermore, the average error decreases with increasing distance, which is a feature specific to vector propagation [JBS06].

7.9 Summary

We presented a framework to render and filter gigapixel images at interactive rates. This interactivity allows the immediate visual feedback of filter results. The framework can thus be used as a rapid prototyping testbed for digital filter design. Furthermore, the binary representation of the input data is never altered during visualization, resulting in significantly reduced storage and memory requirements when compared to previous image manipulation tools. The hierarchical vector quantization employed to compress the imput images results in bit rates around 2 bits per pixel while offering a visual fidelity that rivals the JPEG standard at similar resolutions. Compressing the 4.5 gigapixel im-

age of Figure 7.12 took about 40 minutes on an Intel Core2Quad Q6600 using four encoder threads. Frame rates naturally depend on the filter complexity. If no filtering is performed frame rates of several hundred frames per second can be maintained.

Furthermore, we integrated propagation filters into the gigapixel filter pipeline. We demonstrated the benefits of this particular filter types by a framework to compute discrete distance transforms, Voronoi diagrams, and generalized Voronoi diagrams. This particular filter type runs at high-speed and is precise in the sense that the absolute error can be strictly bounded from above by $\sqrt{485} - \sqrt{481} < 0.091034$ pixels. Also, errors are highly unlikely to occur. This filter can be used either for image-analysis purposes, or as a building block for artistic filters that operate on gigapixel images. Some of such artistic filters are shown in Figure 7.19.

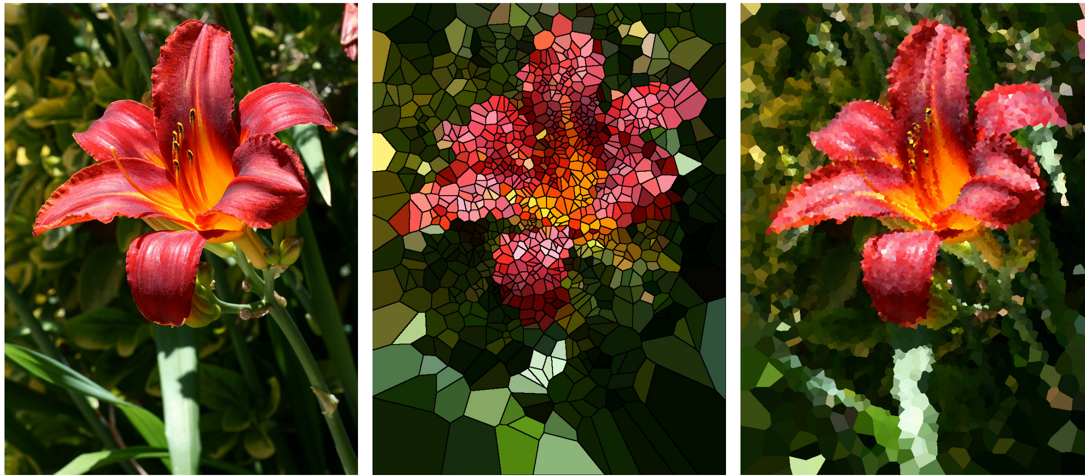


Figure 7.19: *An example for artistic filters using Voronoi diagrams. From left to right: Original* 768×1024 *picture of a Red Magic Daylily (Hemerocallis fulva longituba), Voronoi diagram with 1 K Gaussian distributed features and region boundaries, Voronoi diagram with 10 K Gaussian distributed features. Voronoi regions are colored by the color of the original image at the respective Voronoi site. Each image was generated in less than 13 ms.*

Chapter 8

Conclusions

This thesis presented methods to synthesize, augment, and render highly detailed digital models. Although the data used in the various chapters of this thesis differs significantly in size and type, we showed that certain paradigms are highly beneficial in order to tap the GPUs full potential. These include hierarchical data compression for large-scale data sets such as point clouds, digital elevation maps, and gigapixel images. Only by using an offline preprocessing stage to organize the data into a hierarchical LOD representation can interactive frame rates be achieved. The reason for this is two-fold. First of all, hierarchical methods are best suited to reflect and organize the overwhelming amount of details present in these data sets at *multiple scales*. Secondly, as data sets continue to grow, methods that seek to compute such hierarchical organizations during rendering will not be able to scale favourably.

But even if hierarchical preprocessing has been widely accepted in the computer graphics community, it is not the only key to achieve interactive frame rates for even the largest models available. The von Neumann architecture of today's computers implies relatively narrow busses that tend to limit the overall performance drastically. In order to overcome these limitations, GPU-friendly data compression is not only beneficial but mandatory. Only if the models to be displayed are decoded directly on the GPU can all bandwidths up to the GPU be virtually increased. The concept of decoding data as late as possible in the rendering pipeline is further fostered by the GPU's extremely high internal bandwidths.

Once interactivity is achieved, the user can successfully explore the rich amount of details present in the data, and highly intuitive and effective data manipulation environments can be devised. On the other hand, if applications cannot maintain interactivity, their usability value quickly deteriorates, up to the point where they actually become inferior to print media, since the latter offer vastly superior resolutions when compared

to today's display technologies.

A similar argumentation holds for models that are either fully synthesized or composed on the fly. Only with interactive environments can complex editing operations be performed effectively. In case of the fractal terrain editor discussed in this thesis we have also shown that editing operations that have been deemed infeasible before can suddenly become highly intuitive if only the user is provided with rapid visual feedback. The reason is that the creativity of humans cannot be over-estimated, especially if their capability to quickly correct actions that did not show the desired result is in question.

Finally, we would like to note that the tremendous success of systems like Google Earth has foremost demonstrated that people are tired of looking at still-images. They want intuitive, highly interactive and immersive environments that allow them to explore highly detailed models and to interact with them instead of just looking at them.

Bibliography

[AaS] *Aim @ shape*, http://www.aimatshape.net.

[ABCO$^+$01] Mark Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva, *Point set surfaces*, Proceedings of IEEE Visualization, 2001, pp. 21–28.

[ASA] (National Aeronautics and NASA Space Administration), *Blue marble next generation*, http://earthobservatory.nasa.gov.

[Aur91] F. Aurenhammer, *Voronoi diagrams–a fundamental geometric data structure*, ACM Computing Surveys **23** (1991), no. 3, 345–405.

[BCD$^+$] Jean-Yves Bouget, Brian Curless, Paul Debevec, Marc Levoy, Shree Nayar, and Steve Seitz, *3D photography*, ACM SIGGRAPH 2000 Course Notes.

[BHZK05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt, *High-quality surface splatting on today's GPUs*, Proceedings of Eurographics Symposium on Point-Based Graphics, 2005, pp. 17–24.

[BK03] Mario Botsch and Leif Kobbelt, *High-quality point-based rendering on modern GPUs*, Proceedings of Pacific Graphics 2003, 2003, pp. 335–343.

[Bli78] James F. Blinn, *Simulation of wrinkled surfaces*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH) **12** (1978), no. 3, 286–292.

[Blo00] Jonathan Blow, *Terrain rendering at high levels of detail*, Presentation at Game Developer's Conference, 2000.

[Bly06] David Blythe, *The Direct3D 10 system*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), 2006, pp. 724–734.

[BM98] M.A. Butt and P. Maragos, *Optimum design of chamfer distance transforms*, IEEE Transactions on Image Processing **7** (1998), no. 10, 1477–1484.

[BMF03] Robert Bridson, Sebastian Marino, and Ronald Fedkiw, *Simulation of clothing with folds and wrinkles*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2003, Article No. 3, pp. 28–36.

[Boy08] C. Boyd, *Direct3D 11 compute shader*, Talk at the Microsoft Gamefest Conference, 2008.

[Bre06a] Claus Brenner, *Arial laser scanning*, Talk at International Summer School ”Digital Recording and 3D Modelling”, April 2006, http://www.ikg.uni-hannover.de/publikationen/publikationen/2006/brenner_tutorial_part1.pdf.

[Bre06b] ______, *Extraction and modelling*, Talk at International Summer School ”Digital Recording and 3D Modelling”, April 2006, http://www.ikg.uni-hannover.de/publikationen/publikationen/2006/brenner_tutorial_part2.pdf.

[BS08] Tamy Boubekeur and Christophe Schlick, *A flexible kernel for adaptive mesh refinement on GPU*, Computer Graphics Forum **27** (2008), no. 1, 102–114.

[BSW83] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser, *Some refinement algorithms and data structures for regular local mesh refinement*, Scientific Computing, 1983, pp. 3–17.

[BW98] David Baraff and Andrew Witkin, *Large steps in cloth simulation*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 32, 1998, pp. 43–54.

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt, *Efficient high quality rendering of point sampled geometry*, Proceedings of the Eurographics Workshop on Rendering, 2002, pp. 53–64.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull, *The reyes image rendering architecture*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH) **21** (1987), no. 4, 95–102.

[CGG+03a] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Frederico Ponchio, and Roberto Scopigno, *BDAM – batched dynamic adaptive meshes for high performance terrain visualization*, Computer Graphics Forum (Proceedings of Eurographics) **22** (2003), no. 3, 505–514.

[CGG+03b] ______, *Planet-sized batched dynamic adaptive meshes (P-BDAM)*, Proceedings of IEEE Visualization, vol. 14, 2003, pp. 147–154.

[CK07] N. Cuntz and A. Kolb, *Fast hierarchical 3D distance transformations on the GPU*, Eurographics Short Papers, 2007, pp. 93–96.

[CL96] Brian Curless and Marc Levoy, *A volumetric method for building complex models from range images*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 23, 1996, pp. 303–312.

[Cla43] Alexis-Claude Clairaut, *Théorie de la figure de la terre*, tirée des principes de l’hydrostatique 8e (1743).

[Coo84] Robert L. Cook, *Shade trees*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH) **18** (1984), no. 3, 223–231.

[Cor01] Glenn Corpes, *Procedural landscapes*, Presentation at Game Developer’s Conference, 2001.

[Cro84] Franklin C. Crow, *Summed-area tables for texture mapping*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH) **18** (1984), no. 3, 207–212.

[CSB87] John Horton Conway, N. J. A. Sloane, and Etsuko Bannai, *Sphere packings, lattices, and groups*, first ed., Springer, 1987.

[Cui99] O. Cuisenaire, *Distance transformation: Fast algorithms and applications to medical image processing*, Phd. thesis, Univ. Catholique de Louvain, Oct. 1999.

[Cyb99] Cyberware, *3D Scanner Designed To Scrutinize Works of Michelangelo*, http://www.cyberware.com/news/pressReleases/scanningMichelangelo.html, 1999.

[Dac05] Carsten Dachsbacher, *ShaderX^4 - Advanced rendering techniques*, first ed., ch. Cached Procedural Textures for Terrain Rendering, Charles River Media, 2005.

[Dan80] P.E. Danielsson, *Euclidean distance mapping*, Computer Graphics and Image Processing **14** (1980), 227–248.

[DCN06] Carlos A. Dietrich, Joao L. D. Comba, and Luciana P. Nedel, *ShaderX^5 - Advanced rendering techniques*, ch. Storing and Accessing Topology on the GPU: A Case Study on Mass-Spring Systems, pp. 565–578, Charles River Media, 2006.

[Den03] M.O. Denny, *Algorithmic geometry via graphics hardware*, Phd. thesis, Universität des Saarlandes, Saarbrücken, Germany, Mar. 2003.

[DH00] Michael Doggett and Johannes Hirche, *Adaptive view dependent tessellation of displacement maps*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, 2000, pp. 59–66.

[DMA02] Mathieu Desbrun, Mark Meyer, and Pierre Alliez, *Intrinsic parameterizations of surface meshes*, Computer Graphics Forum (Proceedings of Eurographics) **21** (2002), no. 3, 209–218.

[DMB00] Mathieu Desbrun, Mark Meyer, and Alan H. Barr, *Cloth modeling and animation*, ch. Interactive Animation of cloth-like Objects in Virtual Reality, pp. 219–239, A.K. Peters Ltd., 2000.

[DSW08] Christian Dick, Jens Schneider, and Rüdiger Westermann, *Efficient geometry compression for GPU-based decoding in realtime terrain rendering*, Computer Graphics Forum (2008), to appear.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger, *Sequential point trees*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) **22** (2003), no. 3, 657–662.

[DWS+97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein, *ROAMing terrain: Real-time optimally adapting meshes*, Proceedings of IEEE Visualization, 1997, pp. 81–88.

[Elb05] Gershon Elber, *Geometric texture modeling*, IEEE Computer Graphics and Applications **25** (2005), no. 4, 66–76.

[EMP+03] David S. Ebert, F. Kenton Musgrave, Darwyn R. Peachey, Ken Perlin, and Steve Worley, *Texturing & modelling - a procedural approach*, third ed., Morgan Kaufmann Publishers, 2003.

[Eve01] Cass Everitt, *Interactive order-independent transparency*, Tech. report, NVIDIA White papers, 2001.

[FEKR90] R.L. Ferguson, R. Economy, W.A. Kelly, and P.P. Ramos, *Continuous terrain level of detail for visual simulation*, Proceedings of the IMAGE V Conference, June 1990, pp. 144–151.

[FH04] Michael S. Floater and Kai Hormann, *Advances in multiresolution for geometric modelling*, first ed., ch. Surface Parameterization: a Tutorial and Survey, pp. 157–186, Springer, 2004.

[FL79] Robert J. Fowler and James J. Little, *Automatic extraction of irregular network digital terrain models*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 6, 1979, pp. 199–207.

[Fli] Graham Flint, *The gigapxl project*, http://www.gigapxl.org.

[Flo97] Michael S. Floater, *Parametrization and smooth approximation of surface triangulations*, Computer Aided Geomtric Design **14** (1997), no. 3, 231–250.

[For86] S. Fortune, *A sweepline algorithm for Voronoi diagrams*, ACM Symposium on Computational Geometry, 1986, pp. 313–322.

[FvDFH95] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer graphics: Principles and practice in C*, second ed., Addison-Wesley Professional, August 1995.

[GBK05] Michael Guthe, Ákos Balázs, and Reinhard Klein, *GPU-based trimming and tessellation of NURBS and T-spline surfaces*, ACM Transactions on Graphics **24** (2005), no. 3, 1016–1023.

[GD98] J.P. Grossmann and William J. Dally, *Point sampled rendering*, Proceedings of the Eurographics Rendering Workshop, 1998, pp. 181–192.

[Gee08] Kevin Gee, *Direct3D 11 tesselation*, Talk at the Microsoft Gamefest Conference, 2008.

[Geo08] Joachim Georgii, *Real-time simulation and visualization of deformable objects*, Phd. thesis, Technische Universität München, 2008, http://mediatum2.ub.tum.de/node?id=627732.

[GG92] Allen Gersho and Robert M. Gray, *Vector quantization and signal compression*, first ed., Kluwer International Series in Engineering and Computer Science, 1992.

[GGS95] Markus Gross, Roger Gatti, and Oliver Staadt, *Fast multiresolution surface meshing*, Proceedings of IEEE Visualization, vol. 6, 1995, pp. 135–142.

[GH86] Ned Greene and Paul S. Heckbert, *Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter*, IEEE Compute Graphics And Applications **6** (1986), no. 6, 21–27.

[GH95] Michael Garland and Paul Heckbert, *Fast polygonal approximation of terrains and height fields*, Tech. Report CMU-CS-95-181, Carnegie Mellon University, 1995.

[gl.] gl.tter, *WiiYourself!—Native C++ WiiMote Library*, http://wiiyourself.gl.tter.org.

[GM04] Enrico Gobbetti and Fabio Marton, *Layered point clouds*, Proceedings of the Eurographics Symposium on Point Based Graphics, 2004, pp. 113–120,227.

[GN98] Robert M. Gray and David L. Neuhoff, *Quantization*, IEEE Transactions on Information Theory **44** (1998), no. 6, 2325–2384.

[Gooa] Google, *Google Earth*, http://earth.google.com.

[Goob] ______, *Google Maps*, http://maps.google.com.

[GPA+] Markus Gross, Hanspeter Pfister, Marc Alexa, Mark Pauly, Marc Stamminger, and Matthias Zwicker, *Point-based computer graphics*, ACM SIGGRAPH 2004 Course Notes.

[Gre05] Simon Green, *GPU gems 2*, ch. 26. Implementing Improved Perlin Noise, pp. 409–416, Addison-Wesley Professional, 2005.

[GVL96a] Gene H. Golub and Charles F. Van Loan, *Matrix computations*, third ed., ch. 8 The Symmetric Eigenvalue Problem, pp. 391–469, The John Hopkins University Press, 1996.

[GVL96b] ______, *Matrix computations*, third ed., ch. 5 Orthogonalization and Least Squares, pp. 206–274, The John Hopkins University Press, 1996.

[GW05] Joachim Georgii and Rüdiger Westermann, *Mass-spring systems on the GPU*, Simulation Modelling Practice and Theory **13** (2005), 693–702.

[Har01] John C. Hart, *Perlin noise pixel shaders*, Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2001, pp. 87–94.

[HDD+92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle, *Surface reconstruction from unorganized points*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 19, 1992, pp. 71–78.

[HDJ04] Lok M. Hwa, Mark A. Duchaineau, and Kenneth I. Joy, *Adaptive 4-8 texture hierarchies*, Proceedings IEEE Visualization, October 2004, pp. 219–226.

[HG83] Roy Hall and Donald Greenberg, *A testbed for realistic image synthesis*, IEEE Computer Graphics and Applications **3** (1983), no. 8, 10–20.

[HLW07a] Xin Huang, Sheng Li, and Guoping Wang, *Displacement modeling: Hardware-accelerated interactive feature modeling on subdivision surfaces*, The Visual Compututer **23** (2007), no. 9, 861–872.

[HLW07b] ______, *A GPU based interactive modeling approach to designing fine level features*, Proceedings of ACM Graphics Interface, 2007, pp. 305–311.

[HNC02] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani, *Interactive animation of ocean waves*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2002, pp. 161–166.

[Hop98] Hugues Hoppe, *Smooth view-fependent level-of-detail control and its application to terrain rendering*, Proceedings of IEEE Visualization, October 1998, pp. 35–42.

[HPCD96] J. Helmsen, E. Puckett, P. Colella, and M. Dorr, *Two new methods for simulating photolithography development in 3D*, SPIE 2726, 1996, pp. 253–261.

[Hro01] Juraj Hromkovič, *Algorithmics for hard problems*, first ed., Springer, 2001.

[HSL$^+$] Kai Hormann, Alla Sheffer, Bruno Lévy, Mathieu Desbrun, and Kun Zhou, *Mesh parameterization: Theory and practice*, ACM SIGGRAPH 2007 Course Notes.

[HSRG07] Charles Han, Bo Sun, Ravi Ramamoorthi, and Eitan Grinspun, *Frequency domain normal map filtering*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), 2007, Article No. 28.

[HTCLM99] K.E. Hoff, J. Keyser T. Culver, M. Lin, and D. Manocha, *Fast computation of generalized Voronoi diagrams using graphics hardware*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) **18** (1999), no. 3, 277–286.

[IBM] *IBM Corp. T221 Flat Panel Monitor*, http://www.ibm.com.

[IfPa] DLR Institut für Planetenforschung, *Forschung : HRSC-A*, http://www.dlr.de/pf/desktopdefault.aspx/tabid-331/.

[IfPb] ______, *Mars express*, http://www.dlr.de/mars/.

[IL02] Harald Ibach and Hans Lüth, *Festkörperphysik*, sixth ed., Springer, 2002.

[IL05] Martin Isenburg and Peter Lindstrom, *Streaming meshes*, Proceedings of IEEE Visualization, 2005, pp. 231–238.

[INH99] Konstantine I. Iourcha, Krishna S. Nayak, and Zhou Hong, *Fixed-rate block-based image compression with inferred pixel values*, Tech. report, S3 Graphics Co., Ltc., July 1999, US Patent No. 6,658,146.

[IVT] IVT, *Bluesoleil*, http://www.bluesoleil.com.

[Jäh05] Bernd Jähne, *Digitale bildverarbeitung*, sixth ed., Springer, April 2005.

[JBS06] M.W. Jones, J.A. Bærentzen, and M. Sramek, *3D distance fields: a survey of techniques and applications*, IEEE Transactions on Visualization and Computer Graphics **12** (2006), no. 4, 581–599.

[Joh66] N. W Johnson, *Convex Polyhedra with Regular Faces.*, Canadian Journal of Mathematics (1966), no. 18, 169–200.

[Joh04] Claes Johanson, *Real-time water rendering - introducing the projected grid concept*, Master's thesis, Lund University, 2004, http://graphics.cs.lth.se/theses/projects/projgrid.

[KB04] Leif Kobbelt and Mario Botsch, *A survey of point-based techniques in computer graphics*, Computers & Graphics **28** (2004), no. 6, 801–814.

[Kes08] John Kessenich, *OpenGL shading language specification v1.30.08*, Intel Corporation, August 2008, http://www.opengl.org/documentation/specs/.

[KH08] Michael Kazhdan and Hugues Hoppe, *Streaming multigrid for gradient-domain operations on large images*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), 2008, Article No. 21.

[KK79] Z. Kulpa and B. Kruse, *Methods of effective implementation of circular propagation in discrete images*, Internal Report LiTH-ISY-I-0274, Dept. of Electrical Engineering, Linköping Univ., Sweden, 1979.

[KLR+95] David Koller, Peter Lindstrom, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner, *Virtual GIS: A real-time 3D geographic information system*, Proceedings of IEEE Visualization, 1995, pp. 94–100.

[KR88] Brian W. Kernighan and Dennis M. Ritchie, *The C programming language*, second ed., Prentice Hall, April 1988.

[KS01] Jan Kautz and Hans-Peter Seidel, *Hardware accelerated displacement mapping for image based rendering*, Proceedings of CMCCC Graphics Interface, 2001, pp. 61–70.

[KSW05] Jens Krüger, Jens Schneider, and Rüdiger Westermann, *DuoDecim - a structure for point scan compression and rendering*, Proceedings of Eurographics/IEEE VGTC Symposium on Point-Based Graphics 2005, 2005, pp. 99–107,146.

[KSW06] Jens Krüger, Jens Schneider, and Rüdiger Westermann, *Compression and rendering of iso-surfaces and point sampled geometry*, The Visual Computer **22** (2006), no. 8, 517–530.

[KTI+01] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi, *Detailed shape representation with parallax mapping*, Proceedings of the ACM International Conference on Artificial Reality and Telexistence, vol. 11, 2001, pp. 205–208.

[KUDC07] Johannes Kopf, Matt Uyttendaele, Oliver Deussen, and Michael Cohen, *Capturing and viewing gigapixel images*, International Conference on Computer Graphics and Interactive Techniques(Proceedings of ACM SIGGRAPH), vol. 26, 2007, Article No. 93.

[KW03a] Jens Krüger and Rüdiger Westermann, *Acceleration techniques for GPU-based volume rendering*, Proceedings of IEEE Visualization, 2003, pp. 287–292.

[KW03b] ———, *Linear algebra operators for GPU implementation of numerical algorithms*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), 2003, pp. 908–916.

[LBG80] Yoseph Linde, Andrés Buzo, and Robert M. Gray, *An algorithm for vector quantization design*, IEEE Transactions on Communications **28** (1980), no. 1, 84–95.

[LC87] William E. Lorensen and Harvey E. Cline, *Marching cubes: A high resolution 3D surface construction algorithm*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), 1987, pp. 163–169.

[Lev90] Marc Levoy, *Efficient ray tracing of volume data*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) **9** (1990), no. 3, 245–261.

[Lev00] ______, *Digitizing the Forma Urbis Romae*, Proceedings of the ACM SIGGRAPH "Digital Campfire" on Computers and Archeology, 2000.

[LH98] Yarden Livnat and Charles Hansen, *View dependent isosurface extraction*, Proceedings of IEEE Visualization, 1998, pp. 175–180.

[LH04] Frank Losasso and Hugues Hoppe, *Geometry clipmaps: Terrain rendering using nested regular grids*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), 2004, pp. 769–776.

[LKR+96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner, *Real-time, continuous level of detail rendering of height fields*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGGRAPH), 1996, pp. 109–118.

[LP01] Peter Lindstrom and Valerio Pascucci, *Visualization of large terrains made easy*, Proceedings of IEEE Visualization, October 2001, pp. 363–370,574.

[LP02] ______, *Terrain simplification simplified: A general framework for view-dependent out-of-core visualization*, IEEE Transactions on Visualization and Computer Graphics **8** (2002), no. 3, 239–254.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk, *The Digital Michelangelo Project: 3D scanning of large statues*, ACM Computer Graphics (Proccedings of ACM SIGGRAPH), vol. 27, 2000, pp. 131–144.

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot, *Least squares conformal maps for automatic texture atlas generation*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) **21** (2002), no. 3, 362–371.

[LS93] Ronald D. Larson and Monish S. Shah, *Method for generating addresses to textured graphics primitives stored in RIP Maps*, Tech. report, Hewlett-Packard Company, Palo Alto, California, June 1993, US Patent No. 5,222,205.

[LSJ96] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson, *A near optimal isosurface extraction algorithm using the span space*, IEEE Transactions on Visualization and Computer Graphics **2** (1996), no. 1, 73–84.

[LW85] Mark Levoy and Turner Whitted, *The use of points as a display primitive*, Technical Report 85–022, University of North Carolina at Chapel Hill, January 1985.

[MA06] David M. Mount and Sunil Arya, *ANN: A library for approximate nearest neighbor searching*, http://www.cs.umd.edu/ mount/ANN, August 2006, Version 1.1.

[Mal00] Henrique S. Malvar, *Fast progressive image coding without wavelets*, Proceedings of IEEE Data Compression, 2000, pp. 243–252.

[Man83] Benôit Mandelbrot, *The fractal geometry of nature*, third ed., W.H. Freeman, 1983.

[Mat04] Oliver Mattausch, *Practical reconstruction and hardware-accelerated direct volume rendering on body-centered cubic grids*, Master's thesis, Technische Universität Wien, Austria, 2004, http://www.cg.tuwien.ac.at.

[Mau03] S. Mauch, *Efficient algorithms for solving static hamilton-jacobi equations*, Ph.D. thesis, California Institute of Technology, Pasadena, CA, Mar. 2003.

[MHG01] Lukas Mroz, Helwig Hauser, and Eduard Gröller, *Space-efficient boundary representation of volumetric objects*, Proceedings of IEEE/TVCG Symposium on Visualization, 2001, pp. 180–188.

[Mil86] Gavin S.P. Miller, *The definition and rendering of terrain maps*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 13, 1986, pp. 39–48.

[MKM89] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace Mace, *The synthesis and rendering of eroded fractal terrains*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH) **23** (1989), no. 3, 41–50.

[Mol89] Peter Molyneux, *Populous*, Bullfrog Productions, 1989.

[Moo98] Gordon E. Moore, *Cramming more components onto integrated circuits*, Proceedings of the IEEE **86** (1998), no. 1, 82–85, Reprint, original in Electronics **38** (April 1965), no. 8.

[MQR03] C.R. Maurer, R. Qi, and V. Raghavan, *A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions*, IEEE Transactions on Pattern Analysis and Machine Intelligence **25** (2003), no. 2, 265–270.

[Mul92] J.C. Mullikin, *The vector distance transform in two and three dimensions*, CVGIP: Graphical Models and Image Processing **54** (1992), no. 6, 526–535.

[Mus03] F. Kenton Musgrave, *Texturing & modelling - a procedural approach, 3rd edition*, ch. 17. QAEB Rendering For Procedural Models, Morgan Kaufmann Publishers, 2003.

[MYC95] Klaus Mueller, Roni Yagel, and J. Fredrick Cornhill, *Accelerating the anti-aliased algebraic reconstruction technique (ART) by table-based voxel backward projection*, Proceedings of IEEE Engineering in Medicine and Biology Society, September 1995, pp. 579–580.

[NAS] NASA (National Aeronautics and Space Administration), *MOLA–The Mars Orbiter Laser Altimeter*, http://mola.gsfc.nasa.gov.

[Ney98] Fabrice Neyret, *Modeling, animating, and rendering complex scenes using volumetric textures*, IEEE Transactions on Visualization and Computer Graphics **4** (1998), no. 1, 55–70.

[Nin] Nintendo, *Wii at nintendo*, http://www.nintendo.com/wii.

[NM02] Neophytos Neophytou and Klaus Mueller, *Space-time points 4D splatting on efficient grids*, Proceedings of the IEEE Symposium on Volume Visualization, 2002, pp. 97–106.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister, *Relief texture mapping*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 27, 2000, pp. 359–368.

[OBSC99] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu, *Spatial tesselations: Concepts and applications of voronoi diagrams*, John Wiley & Sons Ltd., 1999.

[Ols04] J. Olsen, *Realtime procedural terrain generation*, http://oddlabs.com/download/terrain_generation.pdf, 2004.

[OMD01] Marc Olano, Shrijeet Mukherjee, and Angus Dorbie, *Vertex-based anisotropic texturing*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, 2001, pp. 95–98.

[OS04] Tilo Ochotta and Dietmar Saupe, *Compression of point-based 3D models by shape-adaptive wavelet coding of multi-height fields*, Proceedings of the Eurographics Symposium on Point Based Graphics, June 2004, pp. 103–112.

[Paj98] Renato Pajarola, *Large scale terrain visualization using the restricted quadtree triangulation*, Proceedings of IEEE Visualization, 1998, pp. 19–26.

[Pan] Pandromeda, *MojoWorld*, www.mojoworld.com.

[Pea] Darwyn R. Peachey, *Antialiasing solid textures*, ACM SIGGRAPH 1988 'Functional Based Modeling' Course Notes.

[Per85] Ken Perlin, *An image synthesizer*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH) **19** (1985), no. 3, 287–296.

[Per02] ______, *Improving noise*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) **21** (2002), no. 3, 681–682.

[PFL78] Thomas K. Peucker, Robert J. Fowler, and James J. Little, *The triangulated irregular network*, Proceedings of the ASP-ACSM Symposium on DTM's, 1978, pp. 96–103.

[PH89] Ken Perlin and E.M. Hoffert, *Hypertexture*, Computer Graphics (Proceedings of ACM SIGGRAPH) **23** (1989), no. 3, 253–262.

[Pho75] Bui Tuong Phong, *Illumination for computer generated pictures*, Communications of the ACM **18** (1975), no. 6, 311–317.

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba, *Real-time relief mapping on arbitrary polygonal surfaces*, ACM Transactions on Graphics (ACM SIGGRAPH Symposium on Interactive 3D Graphics) **24** (2005), no. 3, 935–935.

[Pom00] A. A. Pomeranz, *ROAM using surface triangle clusters (RUSTiC)*, Master's thesis, Center for Image Processing and Integrated Computing, University of California, Davis, 2000.

[PTVF02a] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical recipes in C++*, second ed., ch. 11 Eigensystems, pp. 461–500, Cambridge University Press, 2002.

[PTVF02b] ———, *Numerical recipes in C++*, second ed., ch. 10.6 Conjugate Gradient Methods in Multidimensions, pp. 424–429, Cambridge University Press, 2002.

[PTVF02c] ———, *Numerical recipes in C++*, second ed., ch. 13.5 Digital Filtering in the Time Domain, pp. 563–568, Cambridge University Press, 2002.

[Pul05] Kari Pulli, *Ubiquitous 3D - graphics everywhere*, Point-Based Graphics keynote presentation, 2005, http://research.nokia.com/people/kari_pulli/.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen von Baar, and Markus Gross, *Surfels: surface elements as rendering primitives*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH), vol. 27, 2000, pp. 335–342.

[RHSS98] Stefan Röttger, Wolfgang Heidrich, Philipp Slussalek, and Hans-Peter Seidel, *Real-time generation of continuous levels of detail for height fields*, Proceedings of WSCG, 1998, pp. 315–322.

[RL00] Szymon Rusinkiewicz and Marc Levoy, *QSplat: a multiresolution point rendering system for large meshes*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH), vol. 27, 2000, pp. 343–352.

[RP66] A. Rosenfeld and J.L. Pfalz, *Sequential operations in digital picture processing*, Journal of the ACM **13** (1966), no. 4, 471–494.

[RPZ02] Liu Ren, Hanspeter Pfister, and Matthias Zwicker, *Object space ewa splatting: A hardware accelerated approach to high quality point rendering*, Computer Graphics Forum (Proceedings of Eurographics) **21** (2002), no. 3, 461–470.

[RT06] G. Rong and T.-S. Tan, *Jump flooding in gpu with applications to Voronoi diagram and distance transform*, ACM Symposium on Interactive 3D Graphics and Games, 2006, pp. 109–116.

[SA08] Mark Segal and Kurt Akeley, *The OpenGL® graphics system: A specification, version 3.0*, August 2008, http://www.opengl.org.

[Sak93] Georgios Sakas, *Modeling and animating turbulent gaseous phenomena using spectral synthesis*, The Visual Computer **9** (1993), no. 4, 200–212.

[Sau88] Dietmar Saupe, *Point evaluation of multi-variable random fractals*, Visualisierung in Mathematik und Naturwissenschaft, Bremer Computergraphik Tage, 1988.

[Sau89] ———, *Simulation und Animation von Wolken mit Fraktalen*, Informatik Fachberichte Vol. 222, Proceedings of GI Jahrestagung, I, Computergestützter Arbeitsplatz, 1989, pp. 373–384.

[Say00] Khalid Sayood, *Introduction to Data Compression*, second ed., Morgan Kaufmann Publishers, 2000.

[SB02] S. Svensson and G. Borgefors, *Digital distance transforms in 3D images using information from neighborhoods up to* $5 \times 5 \times 5$, Computer Vision and Image Understanding **88** (2002), 24–53.

[SBMW07] Thomas Schiwietz, Supratik Bose, Jonathan Maltz, and Rüdiger Westermann, *A fast and high-quality cone beam reconstruction pipeline using the GPU*, Proceedings of SPIE Medical Imaging, February 2007.

[SBW06] Jens Schneider, Tobias Boldte, and Ruediger Westermann, *Real-time editing, synthesis, and rendering of infinite landscapes on GPUs*, Vision, Modeling and Visualization 2006, 2006.

[Sch03] Jens Schneider, *Kompressions- und Darstellungsmethoden für hochaufgelöste Volumendaten*, Diploma thesis, RWTH Aachen, Germany, 2003, English version http://www.glhint.de.

[Sch07] Thomas Schiwietz, *Acceleration of medical imaging algorithms using programmable graphics hardware*, Phd. thesis, Technische Universität München, May 2007.

[Sed98] Robert Sedgewick, *Algorithms in C++. Parts 1–4: Fundamentals, data structures, sorting, searching*, third ed., Addison-Wesley Longman, 1998.

[Set96] J.A. Sethian, *A fast marching level set method for monotonically advancing fronts*, National Academy of Sciences US-Paper Edition **93** (1996), no. 4, 1591–1595.

[SG03] Vitaly Surazhsky and Craig Gotsman, *Explicit surface remeshing*, Proceedings of the Eurographics Symposium on Geometry Processing, vol. 1, June 2003, pp. 17–28.

[SGW06] Ryan Schmidt, Cindy Grimm, and Brian Wyvill, *Interactive decal compositing with discrete exponential maps*, ACM Transactions on Graphics **25** (2006), no. 3, 605–613.

[She94] Jonathan Richard Shewchuck, *An introduction to the conjugate gradient method without the agonizing pain*, http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf, 1994, Edition $1\frac{1}{4}$.

[SJ01] R. Satherly and M.W. Jones, *Vector-city vector distance transform*, Computer Vision and Image Understanding **82** (2001), no. 3, 238–254.

[SJP05] Le-Jeng Shiue, Ian Jones, and Jörg Peters, *A realtime GPU subdivision kernel*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 24, 2005, pp. 1010–1015.

[SK01] Dietmar Saupe and Jens-Peer Kuska, *Compression of iso-surfaces for structured volumes*, Proceedings of Vision, Modeling, and Visualization (VMV), 2001, pp. 330–340.

[SKS96] Andreas Schilling, Günter Knittel, and Wolfgang Strasser, *Texram: A smart memory for texturing*, IEEE Computer Graphics and Applications **16** (1996), no. 3, 32–41.

[SKW09] Jens Schneider, Martin Kraus, and Rüdiger Westermann, *GPU-based real-time discrete euclidean distance transforms with precise error bounds*, International Conference on Computer Vision Theory and Applications (VISAPP), 2009, to appear.

[SLMB05] Alla Sheffer, Bruno Lévy, Maxim Mogilnitsky, and Alexander Bogomyakov, *ABF++: fast and robust angle based flattening*, ACM Transactions on Graphics **24** (2005), no. 2, 311–330.

[SN95] Martin Suter and D. Nüesch, *Automated generation of visual simulation databases using remote sensing and GIS*, Proceedings of IEEE Visualization, 1995, pp. 86–93.

[SOM04] A. Sud, M.A. Otaduy, and D. Manocha, *DiFi: Fast 3D distance field computation using graphics hardware*, Computer Graphics Forum **23** (2004), no. 3, 557–566.

[SPG03] C. Sigg, R. Peikert, and M. Gross., *Signed distance transform using graphics hardware*, IEEE Visualization, 2003, pp. 83–90.

[SPR06] Alla Sheffer, Emil Praun, and Kenneth Rose, *Mesh parameterization methods and their applications*, Foundations and Trends ® in Computer Graphics and Vision **2** (2006), no. 2, 105–171.

[ST04] R. Strzodka and A. Telea, *Generalized distance transforms and skeletons in graphics hardware*, Joint EG/IEEE TVCG Symposium on Visualization, 2004, pp. 221–230.

[Sta] University of Stanford, *The Digital Michelangelo Project*, http://graphics.stanford.edu/projects/mich/.

[SW01] Jens Schneider and Rüdiger Westermann, *Towards real-time visual simulation of water surfaces*, Proceedings of Vision, Modeling, and Visualization (VMV), 2001, pp. 211–218.

[SW03] Jens Schneider and Rüdiger Westermann, *Compression domain volume rendering*, IEEE Visualization, 2003, pp. 293–300.

[SW06] Jens Schneider and Rüdiger Westermann, *GPU-friendly high-quality terrain rendering*, Journal of WSCG **14** (2006), no. 1–3, 49–56.

[TE05] Eduardo Tejada and Thomas Ertl, *Large steps in GPU-based deformable bodies simulation*, Simulation Practice and Theory (Special Issue on Programmable Graphics Hardware) **19** (2005), no. 9, 703–715.

[Tsi95] N. Tsitsiklis., *Efficient algorithms for globally optimal trajectories*, IEEE Transactions on Automatic Control **40** (1995), no. 9, 1528–1538.

[TvW02] A. Telea and J.J. van Wijk., *An augmented fast marching method for computing skeletons and centerlines*, Symposium on Visualization, 2002, pp. 251–260.

[Ulr00] T. Ulrich, *Rendering massive terrains using chunked level of detail*, ACM SIGGraph Course "Super-size it! Scaling up to Massive Virtual Worlds", 2000.

[VHB87] Brian Von Herzen and Alan H. Barr, *Accurate triangulations of deformed, intersecting surfaces*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH) **21** (1987), no. 4, 103–110.

[Vor08] G.M. Voronoi, *Nouvelles applications des paramètres continus à la théorie des formes quadratiques. deuxiéme mémoire: recherches sur les paralléloèdres primitifs*, Reine Angewandte Mathematik **134** (1908), 198–287.

[VPBM01] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell, *Curved PN triangles*, Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics, 2001, pp. 159–166.

[WE98] Rüdiger Westermann and Thomas Ertl, *Efficiently using graphics hardware in volume rendering applications*, International Conference on Computer Graphics and Interactive Techniques, 1998, pp. 291–294.

[Weia] Eric W. Weisstein, *Circle packing*, MathWorld – A Wolfram Web Resource, http://mathworld.wolfram.com/CirclePacking.html.

[Weib] ______, *Kepler conjecture*, MathWorld – A Wolfram Web Resource, http://mathworld.wolfram.com/KeplerConjecture.html.

[Wil83] Lance Williams, *Pyramidal parametrics*, International Conference on Computer Graphics and Interactive Techniques (Proceedings of ACM SIGGRAPH), vol. 10, 1983, pp. 1–11.

[WMD+04] Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, and Reinhard Klein, *Scalable compression and rendering of textured terrain data*, Journal of WSCG **12** (2004), no. 3, 521–528.

[WTL+04] Xi Wang, Xin Tong, Stephen Lin, Shi-Min Hu, Baining Guo, and Heung-Yeung Shum, *Generalized displacement maps*, Proceedings of the Eurographics Symposium on Rendering, 2004, pp. 227–234.

[WWL+04] Michael Waschbüsch, Stephan Würmlin, Edouard Lamboray, Felix Eberhard, and Markus Gross, *Progressive compression of point-sampled models*, Proceedings of the Eurographics Symposium on Point Based Graphics, June 2004, pp. 95–102.

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum, *View-dependent displacement mapping*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) **22** (2003), no. 3, 334–339.

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross, *Pointshop 3D: an interactive system for point-based surface editing*, ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) **21** (2002), no. 3, 322–329.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross, *Surface splatting*, ACM Computer Graphics (Proceedings of ACM SIGGRAPH), vol. 28, 2001, pp. 371–378.

[ZRB+04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly, *Perspective accurate splatting*, Proceedings of CMCCC Graphics Interface, 2004, pp. 247–254.

[ZRS05] Rhaleb Zayer, Christian Rössl, and Hans-Peter Seidel, *Setting the boundary free: a composite approach to surface parameterization*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Geometry Processing, vol. 3, 2005, Article No. 91, p. 91.