Institut für Informatik
Technische Universität München

Real-Time Simulation and Visualization of Deformable Objects

Joachim Georgii

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. Hans-Joachim Bungartz
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Rüdiger Westermann
	2. Hon.-Prof. Hans-Christian Hege, Zuse Institute Berlin

Die Dissertation wurde am 19.09.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 06.12.2007 angenommen.

To my wife, my children, my family, and my friends

Abstract

In this thesis, I present a framework for physical simulation and visualization of deformable volumetric bodies in real time. Based on the implicit finite element method a multigrid approach for the efficient numerical simulation of elastic materials has been developed. Due to the optimized implementation of the multigrid scheme, 200,000 elements can be simulated at a rate of 10 time steps per second. The approach enables realistic and numerically stable simulation of bodies that are described by tetrahedral or hexahedral grids. It can efficiently simulate heterogeneous bodies—i.e., bodies that are composed of material with varying stiffness—and includes linear as well as non-linear material laws.

To visualize deformable bodies, a novel rendering method has been developed on programmable graphics hardware. It includes the efficient rendering of surfaces as well as interior volumetric structures. Both the physical simulation framework and the rendering approach have been integrated into a single simulation support system. Thereby, available communication bandwidths have been efficiently exploited. To enable the use of the system in practical applications, a novel approach for collision detection has been included. This approach allows one to handle geometries that are deformed or even created on the graphical subsystem.

Zusammenfassung

In dieser Arbeit präsentiere ich ein Framework für die physikalische Simulation und Visualisierung von deformierbaren volumetrischen Körpern in Echtzeit. Basierend auf der Methode der impliziten finiten Elemente wurde ein Mehrgitteransatz zur effizienten numerischen Simulation elastischer Materialien entwickelt. Durch die optimierte Implementierung des Mehrgitterverfahrens können 200.000 Elemente mit einer Rate von 10 Zeitschritten pro Sekunden simuliert werden. Der Ansatz ermöglicht die realistische und numerisch stabile Simulation von Körpern, die durch Tetraeder- oder Hexaedergitter beschrieben sind. Er kann heterogene Körper – das heißt Körper, die aus unterschiedlich steifem Material bestehen – effizient simulieren und berücksichtigt lineare sowie nicht-lineare Materialgesetze.

Zur Visualisierung deformierbarer Körper wurde eine neuartige Renderingmethode auf programmierbarer Graphikhardware entwickelt. Sie beinhaltet sowohl das effiziente Rendering von Oberflächen als auch von internen volumetrischen Strukturen. Das Simulationsframework und die Renderingmethode wurden in ein eigenständiges Simulationssystem integriert. Dabei wurden die verfügbaren Kommunikationsbandbreiten effizient ausgenutzt. Um den Einsatz des Systems in praktischen Anwendungen zu ermöglichen, wurde ein neuer Ansatz zur Kollisionserkennung integriert. Dieser Ansatz ermöglicht die Handhabung von Gittern, die auf dem graphischen Subsystem deformiert oder sogar erst konstruiert werden.

Acknowledgements

I want to thank all persons that helped to make this work possible. First of all, I really have to thank all my colleagues, Kai Bürger, Christian Dick, Stefan Hertel, Polina Kondratieva, Dr. Martin Kraus, Dr. Jens Krüger, Thomas Schiwietz, and Jens Schneider. Not only did they support me in proof-reading this thesis but also provided me with ideas and discussions over the whole period of the last years.

At this point, I have to include my previous colleagues, namely Konstantinos Panagiotou and Dr. Peter Kipfer. Particularly, I want to thank my students Florian Echtler, Benjamin Herrmann, Stefan Plafka, Jochen Strunck, Michael Henze, Matthias Wagner, and Alexandru Duliu, who worked on several parts of the overall system. Special thanks go to Gerhard Schillhuber, who supported me in including the interface for the PHANTOM® haptic device, and Dr. Mario Botsch, who assisted me with the comparison to the Cholesky solver. Furthermore, I want to thank the physicians I have worked with, namely Dr. Maximilian Eder, Dr. Laszlo Kovacz and Prof. Dr. Hubertus Feußner.

Last but not least I want to thank my advisor, Prof. Dr. Rüdiger Westermann, who all the time had an open office and ear. This thesis would certainly not have been possible without his inspiration. He really taught me a lot during my time at his chair, and at this point I just can say: *Thank you!*

Contents

Abstract i

Zusammenfassung iii

Acknowledgements v

List of Figures xiii

List of Tables xv

List of Algorithms xvii

Notation xix

1 Introduction 1

1.1 Contribution 1

1.2 Research Publications 4

1.3 About this Thesis 5

2 Simulation 7

2.1 Related Work 8

2.2 Elasticity Theory 11

2.3 Finite Element Framework 16

2.3.1 Finite Elements 16

2.3.2 Linear Strain Approximation 20

2.3.3 Corotated Strain Approximation 22

2.3.4 Non-Linear Strain 23

2.3.5 Higher-Order Finite Elements 25

2.3.6 Dynamics 26

2.3.7 Time Integration 27
2.3.8 Boundary Conditions 30
2.3.9 Mixed Boundary Conditions 32
2.3.10 Material Update 33
2.3.11 Modified Material Laws 34
2.4 Implicit Multigrid Solver 37
2.4.1 The Multigrid Idea 37
2.4.2 Nested Hierarchies 38
2.4.3 Non-Nested Hierarchies 39
2.4.4 Coarse Grid Correction 40
2.4.5 Galerkin Property 42
2.4.6 Convergence and Numerical Error 43
2.5 Fast Sparse Matrix-Matrix Products 46
2.5.1 Matrix Data Structures 47
2.5.2 Naive Approach 48
2.5.3 1-Step Approach 48
2.5.4 1-Step Stream Acceleration 49
2.5.5 Symmetry Optimization 52
2.5.6 Parallelization 52
2.5.7 Performance Measurement 54
2.6 Mass-Spring Systems 56
2.6.1 Theory 57
2.6.2 Volume Preservation 58
2.6.3 GPU Architecture and Functionality 59
2.6.4 GPU Implementation 61
2.6.5 Irregular Volumetric Models 64
2.6.6 Discussion 67
2.7 Results and Validation 72
2.7.1 Real-Time Multigrid Simulation Framework 72
2.7.2 Comparison with Cholesky Solver 82
2.7.3 Soft Tissue Validation 83

3 Rendering **87**
3.1 Related Work 88
3.1.1 Render Surface 88
3.1.2 Volume Rendering of Unstructured Grids 88

3.2 Direct3D 10 Graphics Pipeline 90
3.3 Deformable Surface Rendering 91
3.3.1 High-Resolution Render Mesh 91
3.3.2 Advanced Shading Techniques 96
3.4 Deformable Volume Rendering 97
3.4.1 Tetrahedral Grid Rendering Pipeline 98
3.4.2 Iso-Surface Rendering 106
3.4.3 Cell Projection 109
3.4.4 Implementation 110
3.4.5 Visualization of Internal Material Properties 111
3.4.6 High-Resolution Render Volumes 112
3.5 Performance Measurements 114
3.5.1 Surface Rendering 114
3.5.2 Volume Rendering 115

4 Collision Detection 119
4.1 Related Work 120
4.2 Contribution 123
4.3 Screenspace-Accurate Object Intersection 125
4.3.1 Object Sampling 125
4.3.2 Ray Merging 128
4.3.3 Primitive Separation 130
4.4 GPU-CPU Data Transfer 131
4.4.1 Texture Packing 132
4.4.2 Intersection Tests 135
4.5 Collision Response 135
4.6 Results 136
4.6.1 Scenes 136
4.6.2 Analysis 139
4.6.3 Non-Closed Polygonal Objects 141

5 The Deformable Bodies System 143
5.1 Overview 143
5.1.1 Generating Models 145
5.1.2 Runtime Computations 145
5.1.3 Changing Parameters at Runtime 146
5.2 Parallelization 146

5.2.1 Multiple Threads 147
5.2.2 Multiple Nodes 149
5.3 Results 150

6 Conclusion 155
6.1 Future Work 156

Bibliography 160

List of Figures

1.1 Overview of publications on parts of the research covered in this thesis . 4
1.2 Basic interplay of the system components described in Chapters 2–4 . . 5

2.1 Strain tensor 12
2.2 Poisson's Ratio 13
2.3 Comparison of linear Cauchy and non-linear Green strain 14
2.4 Finite element types 17
2.5 Coordinate transformation for finite elements 20
2.6 Comparison of the linear, corotational and non-linear simulation 25
2.7 Comparison between linear tetrahedra and Serendipity tetrahedra 26
2.8 Vertex Fixation 31
2.9 Modified material laws in 2D 35
2.10 Modified material laws in 3D 36
2.11 Regular subdivision scheme for various finite element types 39
2.12 Geometric relations between elements in a non-nested hierarchy 40
2.13 Comparison of multigrid V-cycle and full multigrid algorithm 41
2.14 Comparison of pure geometric and Galerkin multigrid approach 43
2.15 Analysis of the algebraic error of the multigrid method 44
2.16 Convergence of the multigrid V-cycles and full multigrid algorithm . . . 45
2.17 Row-compressed matrix format 47
2.18 Matrix-vector products using a symmetric sparse matrix format 53
2.19 Volume preserving forces in mass-spring systems 59
2.20 Stages of the programmable graphics pipeline 60
2.21 Sequence of valence textures 66
2.22 GPU cloth simulation 69
2.23 Interactive GPU-based deformations of the bunny model 70
2.24 Tetrahedral models 73

2.25 Linear time complexity of the multigrid solver 74
2.26 Deformation of a tetrahedral horse model 75
2.27 Visualization of internal stress 75
2.28 Comparison of the multigrid and the conjugate gradient solver for linear strain 76
2.29 Comparison of the multigrid and the conjugate gradient solver for corotational strain 78
2.30 Simulation of non-linear strain 81
2.31 Experimental setup of the soft tissue validation 84
2.32 Soft tissue validation by means of a pig's liver 85
2.33 Surface distance error of the liver experiment 85

3.1 Stages of the Direct3D 10 graphics pipeline implemented on recent GPUs 90
3.2 Displacing a high-resolution render surface on the GPU 93
3.3 Normal calculation on the GPU 94
3.4 Lighting on GPU deformed render surface 95
3.5 Fur shading 96
3.6 Overview of the GPU tetrahedral grid rendering pipeline 98
3.7 Ray-based tetrahedra sampling 100
3.8 Data stream overview of the GPU tetrahedra rendering pipeline 101
3.9 Deforming Cartesian grids 102
3.10 Iso-surface rendering of the deformed visible male data set 108
3.11 Drawbacks of the Powersort algorithm 109
3.12 Visualization of internal properties of the liver data set 112
3.13 Visualization of internal properties of the heterogeneous horse model . . 113
3.14 Direct volume rendering of the engine data-set 113
3.15 Different rendering modes of the bunny model 114
3.16 Interactive deformation and rendering of various objects 115
3.17 Direct volume rendering of the NASA bluntfin data set 117
3.18 Close-up views of the NASA bluntfin data set 117
3.19 Direct volume rendering of the deformed visible human data set 117

4.1 Interactive collision detection between polygonal objects 123
4.2 A diagrammatic overview of the proposed collision detection algorithm 125
4.3 Illustration of interference, self-interference, and partial inversion . . . 126
4.4 Calculation of collision rays by using depth-peeling 128
4.5 Mipmap texture of collision rays 129

4.6 Mipmap construction 130
4.7 Primitive separation 130
4.8 Ray merging and primitive separation 131
4.9 Texture Packing: Counting 132
4.10 Texture Packing: Shifting 133
4.11 Texture Packing: Moving 134
4.12 Repulsion forces 136
4.13 (Self-) collisions between deformable objects 137
4.14 Collisions between dynamic GPU objects 138
4.15 Rigid body collisions 138
4.16 (Self-) Collisions between non-closed dynamic and rigid objects 142

5.1 Interplay of the simulation, render, and collision engine 144
5.2 Interplay of the system threads 147
5.3 Interplay of the simulation threads 148
5.4 System distribution on multiple compute nodes 149
5.5 Screenshot of the tum.3D defo application 150
5.6 Deforming the Mount St. Helens terrain 151
5.7 Force fields allow for intuitive deformations 152
5.8 Breast Augmentation 152
5.9 Deformation of the dragon model 153
5.10 Stress visualization of a bending beam 153

6.1 Validation of the soft tissue deformation by means of volumetric organic data sets 158

List of Tables

2.1 Elastic modulus and Poisson's ratio of some materials 13
2.2 Finite element shape functions 18
2.3 Timing statistics for the update of nested multigrid hierarchies 54
2.4 Timing statistics for the update of non-nested multigrid hierarchies . . 55
2.5 Memory and arithmetic analysis of the point-centric and the edge-centric mass-spring approach 68
2.6 Performance comparison between the point-centric and edge-centric mass-spring approach 69
2.7 GPU simulation performance rates for volumetric mass-spring models . 71
2.8 Timing results for different models using the linearized Cauchy strain measure 74
2.9 Timing statistics for different models using the corotational strain . . . 77
2.10 Timing statistics for different models using the non-linear Green strain . 79
2.11 Timing statistics for higher-order finite elements 82
2.12 Comparison of the multigrid solver with a direct Cholesky solver 83

3.1 Performance of the GPU surface render engine 114
3.2 Element, memory, and timing statistics of the tetrahedral grid rendering pipeline for various data sets 116
3.3 Timing statistics for different rendering modes 118

4.1 Triangle counts of various collision scenes 139
4.2 Timing statistics of the collision detection pipeline 140

List of Algorithms

1 Two grid correction 41
2 1-step multiplication of sparse matrices 49
3 Stream construction for sparse matrix-matrix products 51
4 Point-centric mass-spring system 62
5 Edge-centric mass-spring system 63
6 Volumetric mass-spring system 65
7 Volumetric element-centric mass-spring system 67
8 Pseudo-code snippets for ray-based GPU tetrahedra rendering 104

Notation

Notation	Description	Definition on page
Ω	Reference Configuration of the object	11
λ, μ	Lamé coefficients	13
E	Elastic modulus (Young's modulus)	13
ν	Poisson's ratio	13
$\mathcal{E}$	Strain tensor	12, 14
ϵ	Strain tensor, linearized notation	11
Σ	Stress tensor	12
σ	Stress tensor, linearized notation	11
σ_{Mises}	Von-Mises stress norm	111
D	Matrix of material law: $\sigma = D\epsilon$	14
$u, u(x)$	Displacement field	11
$f, f(x)$	External (surface) forces	11
$g, g(x)$	Volumetric forces	11
ρ	Density	27
$N_i(x)$	Finite element shape functions	17
Φ	Finite element shape matrix	17
B	Finite element strain matrix	21
K^e	Finite element stiffness matrix	21
K	Global stiffness matrix	21
$\mathbb{K}(u)$	Global non-linear stiffness function	23
M^e	Finite element mass matrix	27
M	Global mass matrix	27
C^e	Finite element damping matrix	27
C	Global damping matrix	27

Chapter 1

Introduction

Efficient and physics-based simulation of *deformable objects* is of increasing interest in a number of applications such as virtual environments, computer games, and medical simulators. Due to the computational complexity of the underlying physical models, simplified models are typically used in real-time applications. Such models sacrifice physical correctness for computational speed, which makes it difficult in general to verify how well these model can simulate the behavior of real objects and materials. Usually, these models are either limited by the kind of materials that can be simulated efficiently, or they use very coarse approximations to simulate the underlying physical phenomenons, which are only valid for specific kinds of deformations. On the contrary, *physics-based* simulation is advantageous, because the underlying model is well understood, and thus these methods are applicable in scenarios where physical accuracy is required, for example in medical applications such as surgical training or pre-operative and intra-operative planning.

In many real-time applications, not only fast and accurate simulation is of growing interest but also high-quality visualizations of such deformable bodies. The efficient coupling of numerical simulation methods with advanced rendering techniques has not been considered so far to the best of our knowledge. In particular, interactive volume rendering techniques that allow one to visualize internal material properties at simulation time are important in a wide range of applications.

1.1 Contribution

Due to these observations, the main goal of my PhD is to review the question whether physics-based simulation is really too slow as to be applied in real time, and to even

challenge it. Another goal was to demonstrate that physically accurate simulation and advanced visualization techniques of deformable volumetric bodies can be integrated efficiently into one single simulation support system, which can be used in many practical applications.

Finite element methods are the best known techniques to accurately model the behavior of deformable objects based on the theory of elasticity—and thus they are the state-of-the-art technique for physics-based simulation. Consequently, I started out with an analysis of the linear finite element methods commonly applied to solve problems in structural mechanics. When going through the related work in the field of real-time approaches, it was most striking that the employed numerical techniques did not make use of well established optimization methods at all. In particular, although multigrid approaches were known to be optimal solvers for second-order elliptical partial differential equations, they have not been applied to the simulation of deformable objects in real-time environments. Therefore, my first objective was the development of a multigrid solver suitable for the simulation of deformable objects based on the linear elasticity theory. Compared to previous finite-element-based methods, this solver allows for the real-time simulation of considerably larger models consisting of up to 200,000 elements. The developed physics-based simulation engine even outperforms many simplified approaches at the same time observing physical laws. Therefore, one important result of this thesis is that physics-based simulation is *not* necessarily too slow for real-time environments.

Along a different avenue, numerical simulation performed on graphics processing units (GPUs) has become popular in recent years. Consequently, the evaluation of GPU-based simulation techniques for deformable bodies was an additional research goal. It turned out, however, that only for simplified models, e.g., mass-spring systems, the GPU can clearly outperform the CPU[1]. This is mainly due to the fact that the irregular data structures involved in the computations induce a significant overhead on the GPU. Moreover, due to the limitation of the floating-point precision to 32 bit on recent GPUs up to the NVIDIA G80 series, it became apparent that physical simulation—exhibiting heterogeneous materials—on the GPU will most likely not pay off at the end. Nevertheless, interesting algorithmic concepts for GPU mass-spring systems have been developed, which can be applied in other fields, too.

Therefore, I focused on the optimization of the CPU simulation engine. Primary, the finite element method had to be extended to account for the geometric non-linearity of the underlying model—a favored method is based on corotated finite elements. More-

[1] CPU = Central Processing Unit

over, the full elastic model yielding a system of non-linear equations has been integrated, too. Solving for these approaches basically results in systems of linear equations that continuously change in every simulation step. Therefore, at the core of these extensions was the development of novel algorithms that allow the multigrid solver to be updated very quickly. Finally, these extensions result in a generic multigrid framework, that can be applied in many other application fields, too.

In many computer graphics applications, visually pleasant renderings of the deformed objects are required—especially in real-time environments. Typically, the simulation meshes are too small (despite the achieved multigrid optimizations) to allow for high-quality renderings. Therefore, a high-resolution render mesh can be bound to the simulation mesh to improve the visual quality. The load of deforming and rendering these meshes according to the simulation has been put entirely on the GPU, thereby minimizing the CPU-GPU bandwidth requirements.

Since the visualization of internal states of the simulation is of great interest—particularly in medical applications—I have also developed a volume rendering engine that can visualize deforming unstructured meshes. Volume rendering of unstructured tetrahedral meshes is an ongoing research area in the visualization community. Although many papers have been published in this area, a large majority of them focuses on static meshes. This is possibly due to the fact that simulation and visualization are mostly separate systems, where the simulation is performed offline on a supercomputer and the results are analyzed afterwards. For each time step the mesh geometry has to be uploaded to the GPU anyway. However, since my aim was to tightly couple simulation and visualization, there was a strong need to develop a volume rendering technique for *dynamic* and *unstructured* tetrahedral grids. In this context, a novel generic and scalable pipeline for tetrahedral grid rendering has been designed, which exploits the programmable functionality of current graphics hardware.

The conceptual separation of simulation and rendering has also a deep impact on contact-handling algorithms. Since the CPU is no longer aware of the rendered geometry, it cannot determine collisions of that geometry. On the other hand, because collision detection between n convex primitives can produce up to $\mathcal{O}(n^2)$ overlapping primitives, whereas typically much less intersections occur, the GPU's parallel architecture is not very well suited for this kind of algorithms. Although the GPU can efficiently render and transform this geometry, the whole collision detection pipeline realized on the GPU does not seem to be compatible to CPU solutions. Therefore, a novel GPU-CPU hybrid approach has been developed that allows to combine the best of "both worlds." Although my motivation for this research was the conceptual separation of the sys-

tem into a simulation engine (executed on the CPU) and a render engine (executed on the GPU), the collision approach proposed is of great interest in virtual environments and computer games, since there is an ongoing trend to modify or generate geometry exclusively on the GPU using programmable shaders.

1.2 Research Publications

The work described in this thesis has been partially published in several research papers, which can be categorized in three groups as shown in Figure 1.1. Among the papers on numerical simulation of deformable objects, the GPU mass-spring approaches [GEW05, GW05c] have to be mentioned. The core of the multigrid framework has been presented in [GW05a, GW06a]. GPU-CPU hybrid collision detection has been addressed in [GKW07]. Recently, an extension of the simulation to incorporate modi-

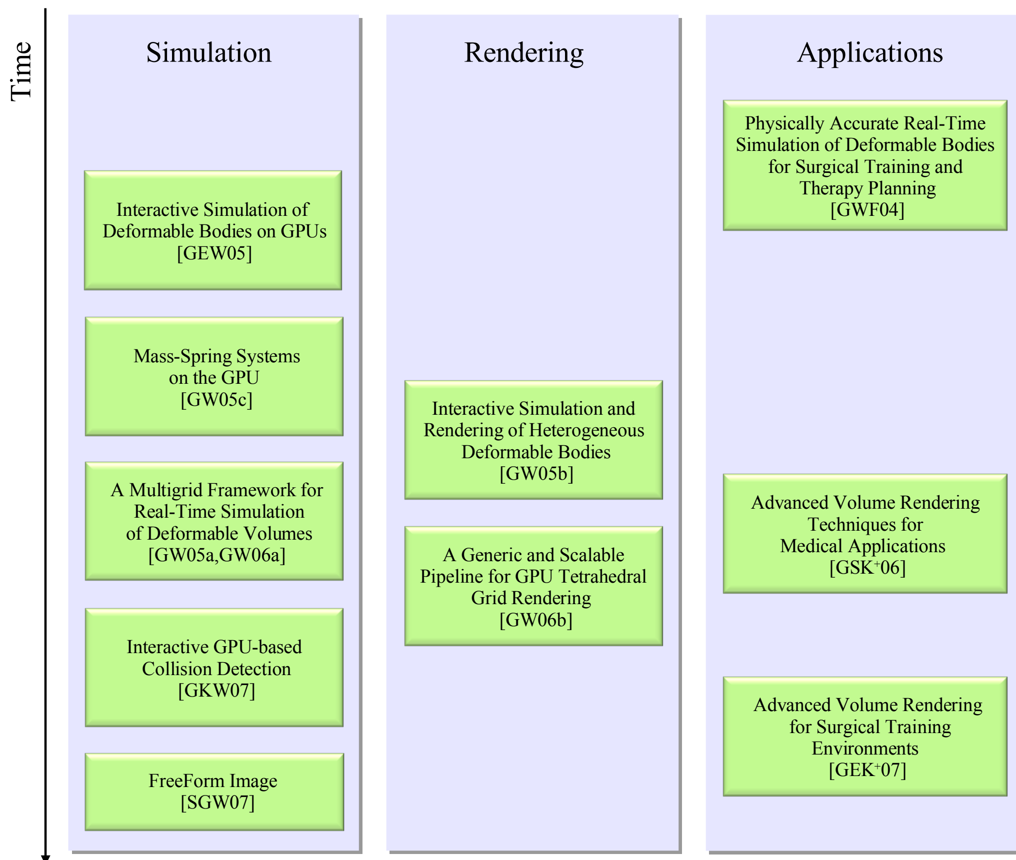


Figure 1.1: *Overview of publications on parts of the research covered in this thesis.*

fied material laws has been proposed in the context of image deformations [SGW07]. This extension (generalized to 3D) is discussed thoroughly in Section 2.3.11. On the rendering side, the coupling of the CPU simulation with the rendering geometry on the GPU has been published in [GW05b]. The tetrahedral grid rendering pipeline has been proposed in [GW06b]. The importance of my work for medical applications has been demonstrated from the very beginning [GWF04, GSK+06, GEK+07].

As customary in most scientific publications, this thesis is written in the academic plural. This should not obscure that the research is achieved by myself, but it should demonstrate that many others have helped to obtain the results presented in this thesis (as outlined in the acknowledgments).

1.3 About this Thesis

This thesis is structured in three main chapters—*simulation*, *rendering* and *collision detection*. Although there are dependencies between these chapters, they can essentially be read separately. Related work, methods and results are presented in each chapter individually. The basic interplay of these three chapters (also defining the basic dependencies) is illustrated in Figure 1.2.

Especially Chapter 2, entitled "*Simulation*," consists of a number of building blocks. The basics of the theory of elasticity are briefly described in Section 2.2. Section 2.3 gives a detailed description of the finite element method. Thus, this section might be skipped by the experienced reader. It should be noted, however, that some applied techniques differ from the state-of-the-art to allow for symbolic calculations. In Section 2.4,

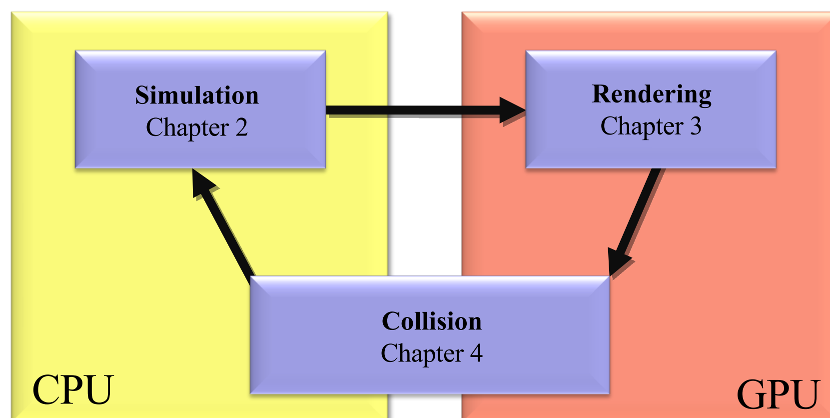


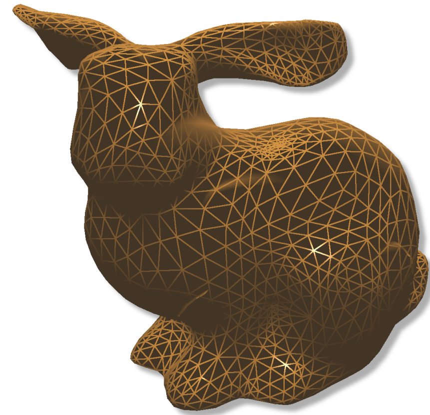
Figure 1.2: *Basic interplay of the system components described in Chapters 2–4.*

the core multigrid technique is presented in detail, including a thorough convergence analysis. To allow for real-time performance even if the system matrices are subject to changes, optimized algorithms to evaluate sparse matrix-matrix products have been developed. Since these algorithms are a general means of sparse matrix computations, Section 2.5 is interesting in its own. It also includes performance measurements of the developed algorithms. Then, I make an excursion to GPU simulation techniques in Section 2.6, focusing on mass-spring systems due to their simplicity. For that reason, this section is also self-contained, and the reader interested only in mass-spring systems on GPUs can concentrate on this section. Finally, I give the results for the CPU multigrid framework, considering all parts of the system using different strain formulations. This includes comparisons with other numerical solvers, in particular the conjugate gradient method and the Cholesky factorization. Additionally, the engine is evaluated with respect to soft tissue simulation required in medical applications.

Chapter 3 exclusively discusses *rendering* aspects. In Section 3.3, I start with high-resolution triangular meshes that are displaced on the GPU according to the simulation, thereby enabling high-quality visualizations without affecting the simulation update rates. The proposed volume rendering algorithms are thoroughly described in Section 3.4. Finally, I give results for all rendering algorithms including performance measurements on current graphics hardware.

In Chapter 4, I describe a general method for *detecting collisions* between geometries that are deformed or even generated on the graphics subsystem. A detailed discussion of all stages of the pipeline is given. Additionally, special effort has been put on the reduction of bandwith requirements. Therefore, a GPU technique to convert sparsely filled textures into dense textures of reduced size has been developed.

In the following Chapter 5, "*The Deformable Bodies System*," the interplay of the three main system components as shown in Figure 1.2 is described. Parallelization strategies including multiple CPU cores and multiple compute nodes connected over a fast network are addressed. In Section 5.3, I summarize the benefits of the deformation engine developed. Finally, I conclude this thesis with a short summary and discussion of future research directions in Chapter 6.



Chapter 2

Simulation

The efficient numerical simulation of deformable bodies is an ongoing research area with applications in a number of different fields. On the one hand, real-time entertainment scenarios such as computer games require *plausible* simulation. Therefore, simplified models yielding interactive update rates are frequently applied. On the other hand, in medical applications—including pre-operative and intra-operative planning as well as surgical training—*physical accuracy* is one of the core requirements. Typically, this implies that real-time performance cannot be achieved due to the computational complexity of the simulation. For that reason, accuracy is often sacrificed for computational speed, yielding plausible rather than accurate results. In this chapter, we show that *physics-based* simulation of deformable bodies can be performed in real time even on commodity computer hardware, because advanced numerical schemes, data structures, and algorithms allow for a highly efficient implementation.

We present a *physics-based* method for the real-time simulation of deformable bodies. The simulation builds upon the physical laws of continuum mechanics. Thus, such a simulation can directly incorporate real-world material constants. To reduce the computational complexity of the applied finite element discretization, multiresolution techniques have been considered. In particular, we show that due to the efficiency of a numerical multigrid scheme, the gap in simulation time between mass-spring systems and FEM simulations can be reduced dramatically. In the following, we give a detailed description of our simulation based on the finite element method. After a very brief summary of the theory of elasticity in Section 2.2, we describe the implemented finite element framework in Section 2.3—including linear Cauchy strain, corotated Cauchy strain, and non-linear Green strain as well as higher-order finite elements. Additionally, we introduce modified material laws in Section 2.3.11,

which also allow for intuitive non-physical deformations. In Section 2.4, we introduce the multigrid technique utilized to efficiently solve the resulting sparse system of linear equations. The results of the latter two sections have been partially published [GW05a, GW05b, GW06a, SGW07]. As the system matrices in some simulation settings are not constant over time, we have developed a novel algorithm for the fast computation of sparse matrix-matrix products required in the update of the multigrid hierarchy as described in Section 2.5. The results of the multigrid simulation framework are presented in Section 2.7.

Simple approximations such as mass-spring systems cannot fulfill the requirement of physical accuracy. However, they still allow for *plausible* results. A special advantage is that pipelined SIMD hardware architectures like GPUs can be utilized to accelerate the simulation significantly. However, there are several restrictions on the stability of such systems, and therefore application-specific adaptions of simulation parameters are required. We introduce mass-spring systems and their implementation on last-generation GPUs in Section 2.6. The results of this section have been previously published [GEW05, GW05c]. Finally, we also compare the GPU mass-spring implementations to the previously explained multigrid framework in terms of speed.

2.1 Related Work

To study the motion of a mechanical system caused by external forces, physics-based simulation is needed. The equations of motion can be formulated and solved to predict the dynamic behavior of deformable objects exhibiting material-dependent properties by utilizing finite element methods [Bat02, Bra01]. Typically, the workings in mechanics, material science and numerics do not focus on interactivity or even real time. In contrary, the computer graphics community has been focusing on interactive approaches to the simulation of such systems for over 20 years. In the following, we restrict ourselves to the research performed in the context of interactivity. From a large scale perspective, these techniques can be classified according to the underlying object discretization, the object's intrinsic deformation behavior, i.e., strain measure, and the method employed to integrate the equations of motion over time (see [GM97, NMK+05] for thorough overviews of the state of the art in this field).

Mass-Spring Systems

The most efficient and most popular techniques for simulating deformable objects based on physical properties are mass-spring systems. There is a large body of literature on

this specialized field [PB81, LTW95, BW98, DSB99, BFA02, BMF03, FGL03, BA04]. Mass-spring systems cannot simulate the real physical behavior of a deformable body as they only use a simplified model. Continuous bodies are approximated by a finite set of point masses that are connected via links to account for material stiffness. Determining proper spring constants to realistically simulate real materials is quite cumbersome. Gelder et al. [Gel98] showed how to choose spring constants to model homogeneous materials. Spring constants can also be configured automatically by neural networks that have been trained to mimic the dynamic behavior of special materials [RNP01]. Nonetheless, very plausible results can be achieved with this method if additional constraints are imposed, such as volume preservation [Pro95] and plasticity [LTW95, THMG04].

Physics-Based Methods in Graphics

In the computer graphics community, finite difference models have been considered very early [TPBF87, TF88, TW88] to solve the partial differential equations derived from the theory of elasticity. Such models approximate continuous bodies by a set of discrete, regularly distributed sample points, and they approximate the stress and strain by finite difference equations. They are amenable to simulate the elastic behavior of curves, surfaces, and solids.

Finite element methods (FEM) [Bat02] have been employed more and more frequently to derive accurate numerical schemes for the governing equations of motion of deformable volumetric bodies. Based on a discretization of the body into a set of elements, e.g., linear tetrahedral elements, boundary elements [JP99], or finite volumes [TBNF03], the solution of the equations to be solved on the domain is then characterized by parameters of these elements.

Implicit solution methods require the assembly of all element equations into a large system of algebraic equations, which can be solved using matrix pre-inversion [BNC96] or the conjugate gradient method [MDM$^+$02, EKS03, HS04, MG04]. An acceleration method was proposed in [CDA99], where a pre-computed linear elastic model is interpolated at run-time. Besides the use of implicit methods in finite element simulations, they have also been employed in finite difference and mass-spring models [TPBF87, LTW95, BW98, DSB99] to enable stable simulations even for large time steps. Capell et al. proposed a shape manipulation tool to control the elastic simulation by a skeleton [CGC$^+$02a].

While the mentioned approaches consider the linear strain measure, i.e., the Cauchy

strain[1], a corotational formulation of the linear strain has become popular. This formulation has been introduced by Rankin et al. [RNO88]. It eliminates artifacts typically introduced by the Cauchy strain when applying large deformations. In Rankin et al.'s method, the rotational part of the deformation is extracted for each finite element, and the forces are computed with respect to the initial reference frame. In this way, stable and fast simulations can be obtained. In the graphics community, the corotational formulation has been integrated recently into interactive applications [EKS03, HS04, MG04, WBG07]. These methods cannot handle more than very few thousands of elements, because conjugate gradient methods are applied as numerical solvers, which have a quadratic worst case complexity. In contrast to an earlier approach, where the rotational part was extracted per vertex [MDM+02], the global stiffness matrix has to be reassembled in every time step to account for element-local rotations.

Explicit finite element methods avoid the construction and solution of a large system of equations. Therefore, the non-linear Green strain[2] can be integrated much more efficiently into these methods. Interactive simulation techniques using this measure have been presented in [ZC99, WDGT01, PDA01, DDCB01, ML03]. However, methods based on explicit time integration are limited due to the Courant condition[3], which significantly restricts the largest possible time step for stiff materials.

To accelerate finite element methods, multiresolution techniques based on adaptive refinement have been proposed [CGC+02b, DDCB01, GKS02]. Multigrid schemes for the deformation of surfaces were presented in [WT04, SYBF06]. To the best of our knowledge, an implicit yet interactive multigrid approach for volumetric bodies has not yet been used in the graphics community. A multigrid method computes the correct FEM solution on the entire mesh at the finest level. This is in contrast to other multiresolution techniques, e.g., [DDBC99, DDCB01], where the solution is computed adaptively for sub-meshes at different resolutions resulting in inconsistent deformations at different hierarchy levels.

Apart from finite element methods, many specialized simulation techniques were developed in the graphics community. They are not based on well-established models such as the elasticity differential equations but define ad-hoc energies to achieve plausible deformations. However, none of these approaches has been validated with respect to the real physical behavior. Botsch and Kobbelt presented a free-form shape editing tool based on radial basis functions [BK05]. Later, this work was extended to a more

[1] A definition is found in Equation (2.9) on page 14

[2] A definition is found in Equation (2.4) on page 12

[3] see Section 2.3.6

realistic shape model minimizing an energy functional [BPWG07]. Other approaches are physics-based, but they consider only a set of modes that are interpolated at runtime [HSO03]. James and Pai achieve reasonable speed-ups by exploiting graphics hardware to perform this interpolation [JP02]. A simplified model for the simulation of thin shells has been proposed [GHDS03]. A point-based approach for simulation has been considered in [MKN$^+$04].

2.2 Elasticity Theory

To simulate the dynamic behavior of deformable volumetric objects, we first have to derive the equations describing the deformation of an elastic solid in equilibrium. Given the elastic solid in the reference configuration $\Omega \subset \mathbb{R}^3$, the deformed object is modeled using a displacement function $u(x), u : \Omega \to \mathbb{R}^3$. The displacement function describes the displacement vector of every point $x \in \Omega$, yielding the deformed configuration $\{x + u(x) \mid x \in \Omega\}$. Analogously, the object can be represented in 1D or 2D.

The potential energy of a system in static equilibrium (zero kinetic energy) is stationary. This leads to the following well-known variational formulation for static elasticity problems [Bat02]:

$$\Pi = \frac{1}{2}\int_{\Omega} \epsilon^{\mathrm{T}} \sigma \,\mathrm{d}x \; - \int_{\Omega} g^{\mathrm{T}} u \,\mathrm{d}x \; - \int_{\partial\Omega} f^{\mathrm{T}} u \,\mathrm{d}x = \min. \tag{2.1}$$

Π is the (virtual) potential energy density associated with the displacement field u. The first term represents the elastic energy stored in the solid, while the second and third terms represent the work done by the body force $g = g(x)$ and by the applied tractions $f = f(x)$ through the displacement u. Solving the equation finally comes down to finding a displacement field u minimizing Π with respect to given external forces f and g.

In the equilibrium equation (2.1), ϵ and σ are derived from the symmetric strain tensor $\mathcal{E}$ and the symmetric stress tensor Σ:

$$\epsilon = (\epsilon_1, \ldots, \epsilon_6)^{\mathrm{T}} = (\mathcal{E}_{11}, \mathcal{E}_{22}, \mathcal{E}_{33}, \mathcal{E}_{12}, \mathcal{E}_{13}, \mathcal{E}_{23})^{\mathrm{T}}, \tag{2.2}$$

$$\sigma = (\sigma_1, \ldots, \sigma_6)^{\mathrm{T}} = (\Sigma_{11}, \Sigma_{22}, \Sigma_{33}, 2\,\Sigma_{12}, 2\,\Sigma_{13}, 2\,\Sigma_{23})^{\mathrm{T}}. \tag{2.3}$$

In its general form, the Green strain tensor describes the non-linear relation between deformation and displacement. Given an infinitesimal cube in the objects reference

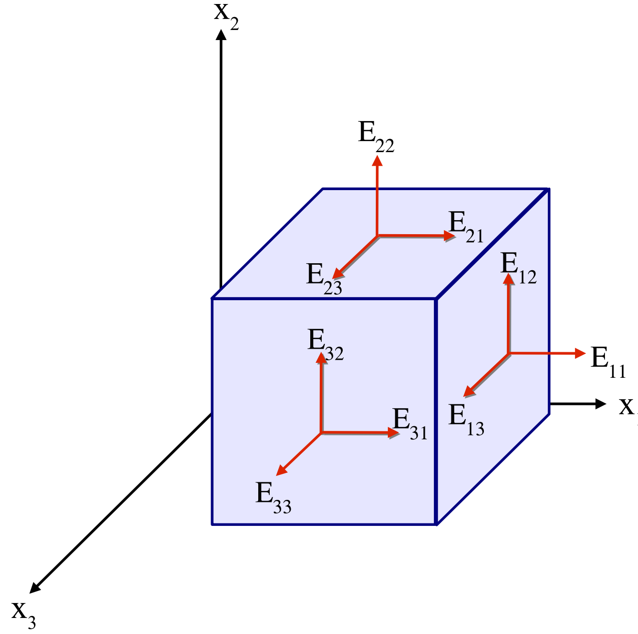


Figure 2.1: *The strain tensor describes the ratio of elongation of an infinitesimal cube for each of the red directions. Due to the symmetry $\mathcal{E}_{ij} = \mathcal{E}_{ji} \;\forall i, j$.*

configuration, the strain tensor describes the ratio of elongation of that cube in each of the directions of the coordinate system, yielding a symmetric tensor (see Figure 2.1)

$$\mathcal{E} = \frac{1}{2} (\nabla(x+u))^{\mathrm{T}} \nabla(x+u) - \frac{1}{2} I_{3,3} = \frac{1}{2} \left(\nabla u + (\nabla u)^{\mathrm{T}} + (\nabla u)^{\mathrm{T}} \nabla u \right),$$

where $I_{3,3}$ is the 3×3 identity matrix. The single components of the tensor thus are

$$\mathcal{E}_{ij} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) + \frac{1}{2} \sum_{k=1}^{3} \frac{\partial u_k}{\partial x_i} \frac{\partial u_k}{\partial x_j}. \tag{2.4}$$

Analogously, the stress tensor Σ describes the respective forces acting on planes within a body. The relation between stress and strain in general is described by the tensor of elastic constants C as $\Sigma_{ij} = C_{ijkl} \mathcal{E}_{kl}$. In an isotropic and fully elastic body, stress and strain tensors are coupled through Hooke's law (linear material law):

$$\Sigma = \lambda \left(\sum_{i=1}^{3} \mathcal{E}_{ii} \right) \cdot I_{3,3} + 2\mu\, \mathcal{E}. \tag{2.5}$$

Here, λ and μ are the Lamé coefficients, which can be derived from the more intuitive

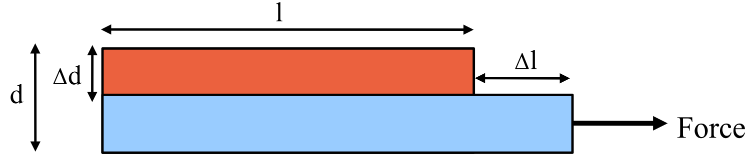


Figure 2.2: *The Poisson's ratio describes the ratio of transverse contraction $\frac{\Delta d}{d}$ to longitudinal stretching $\frac{\Delta l}{l}$ in the direction of the force.*

Material	Elastic modulus $[10^9 N/m^2]$	Poisson's ratio
Rubber	$0.01 - 0.1$	0.5
Magnesium metal (Mg)	45	0.35
Aluminium alloy	69	0.33
Glass (all types)	72	0.24
Titanium (Ti)	$105 - 120$	0.34
steel	$190 - 210$	$0.27 - 0.30$

Table 2.1: *For some materials the elastic modulus and Poisson's ratio are given. Parameters have been taken from* `http://en.wikipedia.org/wiki/Young's_modulus` *and* `http://en.wikipedia.org/wiki/Poisson_ratio`.

elastic modulus E and Poisson's ratio ν:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \tag{2.6}$$

$$\mu = \frac{E}{2(1+\nu)}. \tag{2.7}$$

The elastic modulus is a measure for the "stiffness" of the material. Poisson's ratio describes the ratio of transverse contraction to longitudinal stretching[4] in the direction of the force (see Figure 2.2). A Poisson ratio close to 0.5 guarantees volume preservation. Typical materials have values in the range from 0.3–0.4. The elastic modulus and Poisson's ratio for some common materials are listed in Table 2.1. In particular, organic material is known to have values in the range from 0.45–0.49. Therefore, these materials are almost incompressible.

In short, we write $\sigma = D\epsilon$ with D being the matrix of the material law containing

[4] In the 3D case this is only valid for infinitesimal small longitudinal stretching.

the Lamé coefficients:

$$D = \begin{pmatrix} \lambda + 2\mu & \lambda & \lambda & & & \\ \lambda & \lambda + 2\mu & \lambda & & & \\ \lambda & \lambda & \lambda + 2\mu & & & \\ & & & 4\mu & & \\ & & & & 4\mu & \\ & & & & & 4\mu \end{pmatrix}. \tag{2.8}$$

To account for the fact that only half of the tensor is stored in σ and ϵ, the last three elements in the diagonal of D are weighted by a factor of 2. It can thus be guaranteed that the determined elastic energy $\frac{1}{2}\int_\Omega \epsilon^{\mathrm{T}}\sigma \, \mathrm{d}x$ is the same as if the full tensors $\mathcal{E}$ and Σ would have been considered.

A common simplification is to neglect the quadratic terms in the definition of the strain tensor (2.4), yielding the Cauchy strain tensor $\mathcal{E} = \frac{1}{2}\left(\nabla u + (\nabla u)^{\mathrm{T}}\right)$, which consists of the components:

$$\mathcal{E}_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right). \tag{2.9}$$

While this approximation is appropriate for small deformations, it leads to non-realistic results for large deformations. In particular, as the Cauchy tensor is not invariant under rotations, incorrect forces are very likely to occur in the linear setting (see Figure 2.3). To analyze this fact in more detail, we examine the Cauchy strain of a deformed

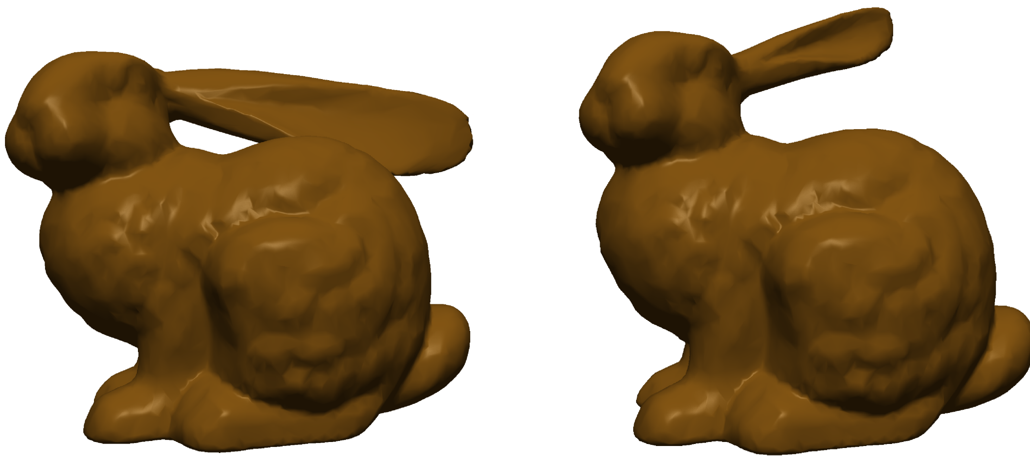


Figure 2.3: *The images show the displacements due to external forces as they were simulated using the linear Cauchy strain tensor (left) and the non-linear Green strain tensor (right).*

configuration $x + u_0$,

$$\mathcal{E}(u_0) = \frac{1}{2}\left(\nabla u_0 + (\nabla u_0)^{\mathrm{T}}\right).$$

We consider a rigid rotation of this configuration $R(x+u_0)$ using an orthogonal matrix R. Then, the Cauchy strain belonging to the respective displacement $u = R(x+u_0)-x$ is

$$\begin{aligned}\mathcal{E}(u) &= \frac{1}{2}\left(\nabla\left(R(x+u_0)-x\right) + \left(\nabla\left(R(x+u_0)-x\right)\right)^{\mathrm{T}}\right)\\ &= \frac{1}{2}\left(R(I+\nabla u_0) - I + (I + (\nabla u_0)^{\mathrm{T}})R^{\mathrm{T}} - I\right),\end{aligned}$$

which differs from $\mathcal{E}(u_0)$ in general. In contrast, the Green strain for the rotated configuration is (due to the orthogonality of R)

$$\begin{aligned}\mathcal{E}(u) &= \frac{1}{2}\left(\left(\nabla(R(x+u_0))\right)^{\mathrm{T}}\left(\nabla(R(x+u_0))\right)\right)\\ &= \frac{1}{2}\left(\left(\nabla(x+u_0)\right)^{\mathrm{T}} R^{\mathrm{T}} R\, \nabla(x+u_0)\right)\\ &= \mathcal{E}(u_0),\end{aligned}$$

and thus equals the strain $\mathcal{E}(u_0)$ of the unrotated configuration.

The linearized strain tensor is commonly used in real-time applications due to performance considerations. As the system including the Green strain tensor becomes non-linear, its solution requires more complex numerical evaluations compared to the linear case. As a matter of fact, only a few attempts have been reported to integrate the non-linear strain tensor into real-time applications [DDCB01]. These approaches, however, have always been combined with explicit time integration schemes, thus requiring small step sizes to guarantee stability (Courant condition).

A good trade-off between Cauchy and Green strain tensor is the so-called corotated strain tensor. It is only a linear approximation just as the Cauchy strain, but it accounts for the geometric non-linearity by calculating per-element rotations. The corotated strain was first introduced by Rankin and Nour-Omid [RNO88] and has been used in the computer graphics community for the physics-based simulation of deformable bodies [MG04, EKS03, HS04].

2.3 Finite Element Framework

In this section, we describe the finite element framework which we have developed. Although this section can be skipped by an experienced reader, the implementation differs in some parts from standard approaches. Our work was inspired by the early approach by Bro-Nielsen and Cotin [BNC96], who introduced the implicit finite element method to the graphics community to the best of our knowledge.

In contrast to many other approaches, the non-linear simulation is handled symbolically in our framework. Because the complexity of the assembly process is thereby shifted to the pre-processing stage, system matrices can be quickly updated in every frame. To perform the symbolical pre-calculations, we make use of a polynomial algebra library. It can be utilized to perform analytic integration of finite element equations rather than numerical integration as it is commonly applied. This analytic integration is required by the non-linear simulation. However, it also simplifies the incorporation of higher-order finite elements into our framework, since it disburdens from the need to carefully adapt the numerical integration schemes for each element type. Therefore, all kinds of finite elements can be likewise used for linear, corotated, and non-linear strain simulation.

In the following, we will first describe the theory behind finite elements and how they are used to approximate continuous equations. Then, the finite element types used in the current framework are introduced. This also includes higher-order elements and their analytical integration. Next, the basic steps that have to be performed for the three simulation types—linear, corotated, and non-linear strain—are described. Then, we present the dynamics of the system, the time integration schemes accomplished, and the boundary conditions used. Finally, we show that the framework can be extended by modified material laws to allow for rigid deformations that do not obey physical laws.

2.3.1 Finite Elements

Finite elements are discrete elements with the basic property that they have a well-defined interpolation function. In contrast to finite difference methods, where values are considered only at discrete samples, finite element methods take the continuum within an element into account. Therefore, finite element methods can achieve higher accuracy in general. Today, finite element methods are used in many different applications including static and dynamic structural analysis, acoustic calculations, thermal calculations, fluid simulation, and determination of electric and magnetic fields.

A finite element with DoF degrees of freedom (numbers of “free” vertices) is de-

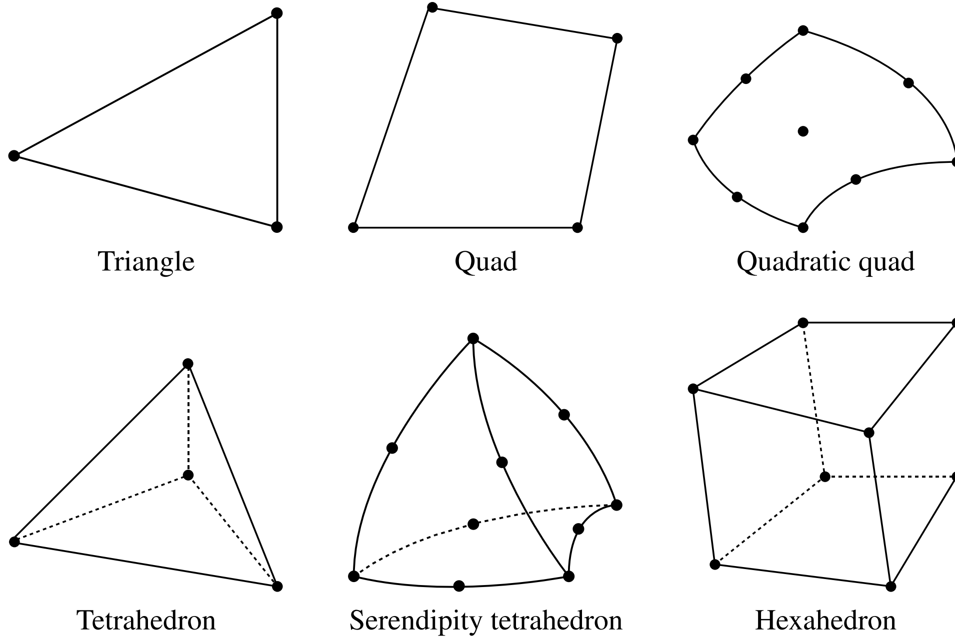


Figure 2.4: *Finite element types.*

scribed by its vertices $v_j, 0 \leq j < \mathrm{DoF}$. Figure 2.4 shows some commonly used finite element types. Data values are only given at the vertices, and arbitrary (element local) interpolation schemes are used to obtain values in the interior of an element.

Based on the so-called shape functions $N_i : \mathbb{R}^3 \rightarrow \mathbb{R}$, an interpolation function for every 3D point can be defined. Let us assume that vectors $f_i \in \mathbb{R}^3$ are given for every vertex of the finite element, then a vector $f \in \mathbb{R}^3$ can be obtained for every point $x \in \mathbb{R}^3$ within one single finite element by

$$f(x) = \sum_{i=0}^{\mathrm{DoF}-1} N_i(x) \cdot f_i. \tag{2.10}$$

With the shape matrix

$$\Phi(x) = \begin{pmatrix} N_0(x) & & & & N_{\mathrm{DoF}-1}(x) & & \\ & N_0(x) & & \dots & & N_{\mathrm{DoF}-1}(x) & \\ & & N_0(x) & & & & N_{\mathrm{DoF}-1}(x) \end{pmatrix} \tag{2.11}$$

the interpolation can be written shortly as

$$f(x) = \Phi(x)\, f^e,$$

Element type	DoF	Shape function for the i-th vertex of one finite element
Triangles	3	$N_i(x_1,x_2) = c_0^i + c_1^i x_1 + c_2^i x_2$
Quads	4	$N_i(x_1,x_2) = c_0^i + c_1^i x_1 + c_2^i x_2 + c_3^i x_1 x_2$
Quadratic Quads	9	$N_i(x_1,x_2) = c_0^i + c_1^i x_1 + c_2^i x_2 + c_3^i x_1 x_2 + c_4^i x_1^2 + c_5^i x_2^2 + $ $+ c_6^i x_1^2 x_2 + c_7^i x_1 x_2^2 + c_8^i x_1^2 x_2^2$
Tetrahedra	4	$N_i(x_1,x_2,x_3) = c_0^i + c_1^i x_1 + c_2^i x_2 + c_3^i x_3$
Serendipity Tetrahedra	10	$N_i(x_1,x_2,x_3) = c_0^i + c_1^i x_1 + c_2^i x_2 + c_3^i x_3$ $+ c_4^i x_1 x_2 + c_5^i x_1 x_3 + c_6^i x_2 x_3$ $+ c_7^i x_1^2 + c_8^i x_2^2 + c_9^i x_3^2$
Hexahedron	8	$N_i(x_1,x_2,x_3) = c_0^i + c_1^i x_1 + c_2^i x_2 + c_3^i x_3 +$ $+ c_4^i x_1 x_2 + c_5^i x_1 x_3 + c_6^i x_2 x_3 + c_7^i x_1 x_2 x_3$

Table 2.2: *Finite element shape functions.*

where $f^e = (f_0^{\mathrm{T}}, \ldots, f_{\mathrm{DoF}-1}^{\mathrm{T}})^{\mathrm{T}}$ is the linearization of the vectors f_i of one finite element.

For a number of common finite element types the shape functions are listed in Table 2.2. For the 2D case, we use triangular and quadrilateral elements. Triangles allow for linear interpolation functions, while quadrangular elements either use bilinear or quadratic shape functions. In the 3D case, tetrahedral elements with linear shape functions are commonly used. A special semi-quadratic interpolation scheme defining the Serendipity tetrahedra is proven to give good results in the context of finite element methods [Bat02]. We have also implemented hexahedral elements with trilinear interpolation.

To determine the coefficients $c_j^i, 0 \leq i,j < \mathrm{DoF}$ of the shape functions, a system of linear equations is build from the interpolation conditions

$$N_i(v_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}.$$

Since the shape functions are linear with respect to the coefficients, these conditions can be formulated as a system of equations with $S \in \mathbb{R}^{\mathrm{DoF} \times \mathrm{DoF}}$:

$$S\, c^i = e_i \quad 0 \leq i < \mathrm{DoF}\,.$$

Solving this system yields the unknown coefficients c^i. Here, e_i is the i-th unit vector, and the matrix entry S_{ij} contains the j-th summand of the shape function (without the coefficient c^i_j) evaluated for the coordinates of v_i. Since this small system has to be solved for every unit vector, the coefficients c^i_j can be determined most efficiently from the inverse of S as

$$c^i_j = (S^{-1})_{ji}.$$

Symbolic Finite Element Integration

Solving partial differential equations with finite elements usually requires to integrate over each finite element (since the continuous solution is approximated by means of the interpolation schemes). To obtain full flexibility, this integration is performed analytically in our framework. Consequently, all finite element types can be integrated very easily in all kinds of simulations. To discretize the equilibrium equation of the potential elastic energy (2.1) using finite elements, the integral

$$\int_{\Omega} \epsilon^{\mathrm{T}}(x)\, \sigma(x)\, \mathrm{d}x$$

has to be determined. Both ϵ and σ depend on the displacement field $u(x)$, and Ω denotes the domain of the finite element considered.

Given displacement values $U_i \in \mathbb{R}^3$ at the supporting vertices of the element, the unknown solution is approximated by $u(x) = \sum_{i=0}^{\mathrm{DoF}-1} N_i(x) \cdot U_i$. As the unknowns U_i are constant with respect to x and $u(x)$ is linear in the U_i's, integration requires only the shape functions $N_i(x)$ to be considered. As these are polynomials in x_1, x_2, and x_3, the analytic integration can be performed in a standard finite element and transformed to Ω by means of a coordinate transformation. This transformation involves the Jacobian determinant as an additional factor in the integration. Here, a standard finite element denotes a finite element with all (outer) vertex coordinates being either zero or one. For higher-order elements, the middle vertices on each edge are determined by linear interpolation. The integration boundaries can be determined more easily for a standard element.

The mapping $\mathcal{J} : \mathbb{R}^3 \to \mathbb{R}^3$ from the standard finite element to Ω can be determined by using the interpolation function of the standard element. Given a point w in the coordinate space of the standard finite element and its associated shape functions $\hat{N}_i : \mathbb{R}^3 \to \mathbb{R}$, $\hat{N}_i(w)$ can be evaluated. Then, these values can be used to determine

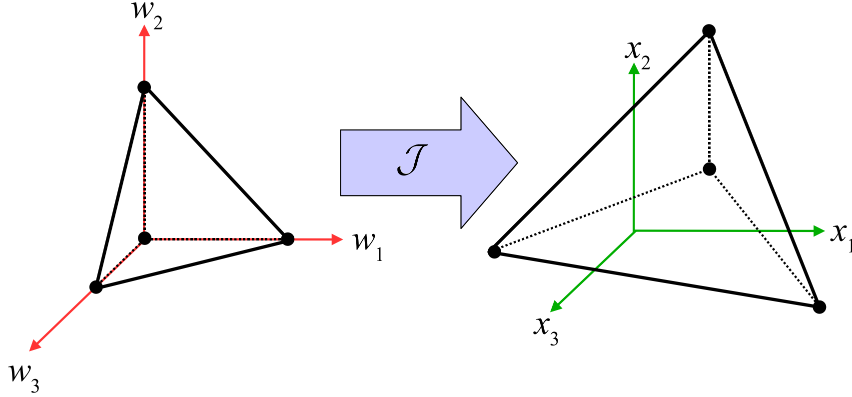


Figure 2.5: *A standard tetrahedral element is mapped to an arbitrary tetrahedral element using the coordinate transformation J.*

the respective position in the element Ω by interpolating the vertex positions v_i:

$$\mathcal{J}(w) = \sum_{i=0}^{\mathrm{DoF}-1} \hat{N}_i(w)\, v_i.$$

Thus, by calculating the shape functions $\hat{N}_i$ once for the standard finite element, the coordinate transformation $\mathcal{J}$ and the required Jacobian determinant $\det\left(\frac{\partial x}{\partial w}\right)$ can be determined for every finite element.

Given the mapping $\mathcal{J}$ from a standard tetrahedron to a given tetrahedral element Ω (see Figure 2.5), the element integral then is:

$$\int_{\Omega} \epsilon^{\mathrm{T}}(x)\,\sigma(x)\,\mathrm{d}x = \int_0^1 \int_0^{1-w_1} \int_0^{1-w_1-w_2} \epsilon^{\mathrm{T}}(\mathcal{J}(w))\,\sigma(\mathcal{J}(w)) \left|\det\left(\frac{\partial x}{\partial w}\right)\right| \mathrm{d}w_3\,\mathrm{d}w_2\,\mathrm{d}w_1.$$

Since both $\mathcal{J}$ and its Jacobian determinant $\det\left(\frac{\partial x}{\partial w}\right)$ are polynomials in w_1, w_2, and w_3, the respective integral can be solved for analytically. The described integration method can also be used for all other finite elements described above by adapting the integration boundaries to the respective element type.

2.3.2 Linear Strain Approximation

If linear Cauchy strain is considered and linear shape functions are used, the finite element equations become rather simple. Given displacements vectors $U_i \in \mathbb{R}^3$ at

every supporting vertex, the displacement field is approximated by

$$u(x) = \sum_{i=0}^{\mathrm{DoF}-1} N_i(x)\, U_i. \tag{2.12}$$

Due to the linearity of the shape function $N_i(x)$, all partial derivatives of $N_i(x)$ and also of $u(x)$ are constant. Therefore, the Cauchy strain is constant within each element:

$$\epsilon = \left(\frac{\partial u_1}{\partial x_1}, \frac{\partial u_2}{\partial x_2}, \frac{\partial u_3}{\partial x_3}, \frac{1}{2}\left(\frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1}\right), \frac{1}{2}\left(\frac{\partial u_1}{\partial x_3} + \frac{\partial u_3}{\partial x_1}\right), \frac{1}{2}\left(\frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2}\right) \right)^{\mathrm{T}}.$$

Due to the linearity of the interpolation (2.12) with respect to U_i, ϵ can be written as $\epsilon = B\, u^e$, where u^e is the linearization of the vectors U_i of one finite element, $u^e = (U_0^{\mathrm{T}}, \ldots, U_{\mathrm{DoF}-1}^{\mathrm{T}})^{\mathrm{T}}$. This is not only true for linear elements, but also for higher-order elements. The matrix $B \in \mathbb{R}^{6 \times 3\,\mathrm{DoF}}$ contains the respective partial derivatives of the shape functions $N_i(x)$:

$$B = \begin{pmatrix} \frac{\partial N_0(x)}{\partial x_1} & & & \cdots & \frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_1} & & \\ & \frac{\partial N_0(x)}{\partial x_2} & & \cdots & & \frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_2} & \\ & & \frac{\partial N_0(x)}{\partial x_3} & \cdots & & & \frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_3} \\ \frac{1}{2}\frac{\partial N_0(x)}{\partial x_2} & \frac{1}{2}\frac{\partial N_0(x)}{\partial x_1} & & \cdots & \frac{1}{2}\frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_2} & \frac{1}{2}\frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_1} & \\ \frac{1}{2}\frac{\partial N_0(x)}{\partial x_3} & & \frac{1}{2}\frac{\partial N_0(x)}{\partial x_1} & \cdots & \frac{1}{2}\frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_3} & & \frac{1}{2}\frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_1} \\ & \frac{1}{2}\frac{\partial N_0(x)}{\partial x_3} & \frac{1}{2}\frac{\partial N_0(x)}{\partial x_2} & \cdots & & \frac{1}{2}\frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_3} & \frac{1}{2}\frac{\partial N_{\mathrm{DoF}-1}(x)}{\partial x_2} \end{pmatrix}. \tag{2.13}$$

Then, the integrals in the variational formulation (2.1) can be calculated by taking the first variation with respect to the unknown element displacement vector u^e:

$$\frac{\partial}{\partial u^e}\frac{1}{2}\int_\Omega \epsilon^{\mathrm{T}}\sigma \,\mathrm{d}x = \frac{\partial}{\partial u^e}\frac{1}{2}\int_\Omega (u^e)^{\mathrm{T}} B^{\mathrm{T}} D\, B\, u^e \,\mathrm{d}x = \left(\int_\Omega B^{\mathrm{T}} D\, B \,\mathrm{d}x\right) u^e = K^e u^e.$$

For linear elements, the partial derivatives in the matrix B are all constant (independent of x), and thus the integral can be determined by multiplying with the finite element volume: $K^e = \int_\Omega B^{\mathrm{T}} DB \,\mathrm{d}x = B^{\mathrm{T}} DB \int_\Omega 1 \,\mathrm{d}x$.

In general, we obtain the element stiffness matrix $K^e \in \mathbb{R}^{3\,\mathrm{DoF} \times 3\,\mathrm{DoF}}$ by performing an integration of the matrix $B^{\mathrm{T}} DB$ over Ω as described in the last section,

$$K^e = \int_\Omega B^{\mathrm{T}} DB \,\mathrm{d}x.$$

All element stiffness matrices are assembled into the global stiffness matrix K with

respect to the global indices of the shared vertices (supporting points). Finally, the system of linear equations

$$Ku = f$$

is deduced, where f now contains external vertex, surface as well as body forces such as gravity or momentum of force. More details about the derivation of the force vectors is given in Section 2.3.8. The vectors u and f are both constructed by linearizing the vertex displacement and force vectors with respect to the global ordering of the vertices.

2.3.3 Corotated Strain Approximation

A rotational invariant formulation of the Cauchy strain tensor is obtained using the so-called corotated strain of linear elasticity introduced by Rankin and Nour-Omid [RNO88]. In this formulation, finite elements are first rotated into their initial configuration before the strain is computed. In this way, although the strain is still approximated linearly, artificial forces as they are introduced by the Cauchy strain (see Section 2.2) are significantly reduced. Rotations are calculated per element using a polar decomposition of the deformation gradient $\nabla(x + u(x))$ as proposed in [EKS03, HS04].

Given the interpolated displacement field u within a finite element at a fixed timestep, the deformation gradient can be calculated by

$$\nabla(x + u(x)) = \nabla \left(x + \sum_{i=0}^{\mathrm{DoF}-1} N_i(x)\, U_i \right) = I_{3,3} + \sum_{i=0}^{\mathrm{DoF}-1} (\nabla N_i(x))\, U_i.$$

The values $\nabla N_i(x)$ can be pre-computed, and in case of linear shape functions the $\nabla N_i(x)$ are constant within each element. A constant deformation gradient based on the current displacement field u can finally be calculated at runtime for each element. If higher-order finite elements are used, the deformation gradient for each element is not constant any more. In this case, either the deformation gradient can be evaluated at a specific point of the element, for instance the center of gravity, or the values $\nabla N_i(x)$ can be integrated over the element domain Ω in a pre-process, yielding an averaged deformation tensor for every element[5]. Again, the current deformation gradient can be determined from the displacement field u at runtime in the same way as it was done for the linear elements. By using an algorithm for the polar decomposition of an arbitrary matrix as described by Higham et al. [Hig86, HS90], the decomposition $\nabla(x + u(x)) = O^e H$ can be determined in a fast and stable way. The quadratically

[5] The values $\nabla N_i(x)$ have to be normalized with respect to the finite element volume.

convergent algorithm proposed is given by the following iterative scheme:

$$\begin{aligned} A_0 &= \nabla(x + u(x)), \\ A_{n+1} &= \frac{1}{2}\left(A_n + (A_n^{-1})^{\mathrm{T}}\right). \end{aligned}$$

The matrix A obtained after a few steps (typically, we use 5 iterations) is the rotational part, $O^e = A_5$, and it is used as per-element rotation. Once the rotation matrices O^e have been calculated for all finite elements, the element stiffness matrix K^e is replaced by $O^e K^e (O^e)^{\mathrm{T}}$, and the global stiffness matrix is reassembled.

A solution to the variational problem (2.1) is found by solving a system of linear equations with the updated system matrix K. As will be demonstrated in Figure 2.6, the corotated strain—although it is still an approximation—yields nearly the same results as the non-linear Green strain for typical examples. On the other hand, due to the linear approximation a significant difference between the Green strain and the corotated Cauchy strain can be observed in cases of large displacements. It is worth noting that the rotations are calculated explicitly—or in other words, the rotations are calculated from the displacement field of the last simulation step. Due to this fact, if rotations change drastically from one time step to another, instabilities can occur.

2.3.4 Non-Linear Strain

The simulation of deformations based on the Green strain tensor using an implicit time integration scheme requires a non-linear system of equations $\mathbb{K}(u) = f$ to be solved. Basically, the functional $\mathbb{K}(u)$ adds higher-order terms to the stiffness matrix K:

$$\mathbb{K}(u) = Ku + \text{higher-order terms}.$$

To calculate a solution to this system we employ the Newton method, which is based on the first order Taylor approximation of the system of equations

$$\mathbb{K}(\tilde{u} + e) \approx \mathbb{K}(\tilde{u}) + \mathbb{K}'(\tilde{u})\, e.$$

Here, $\mathbb{K}'$ is the Jacobian matrix of $\mathbb{K}$. Given an initial solution $\tilde{u}$, this solution can be corrected by solving the equation for $e = u - \tilde{u}$, which only requires a system of linear equations based on the Jacobian matrix to be solved:

$$\mathbb{K}'(\tilde{u})\, e = f - \mathbb{K}(\tilde{u}).$$

Newton solvers are well-known to be sensitive with respect to the initial guess. To improve stability, we could account for the total elastic energy as an indicator. Increasing energy during the current Newton step indicates a poor initial guess. In this case, a new displacement vector is calculated from the previous time steps using extrapolation, and the Newton step is repeated. These steps are performed until convergence is achieved. However, accounting for the total energy has a significant performance impact, and thus we stick with the simple Newton method.

To construct the system of non-linear equations $\mathbb{K}(u)$, we utilize symbolic algebra operations in the pre-processing step. The set of all non-linear element stiffness equations is assembled symbolically into a system of non-linear equations. From the definition of the strain (2.4) it can be seen that $\Sigma_{ij}\mathcal{E}_{ij}$ can be expressed in terms of nodal shape functions $N_i(x)$. By first applying the material law to express Σ in terms of $\mathcal{E}$, $\mathcal{E}$ can then be expressed by the partial derivatives of u, where u is interpolated as shown in equation (2.10) using the shape functions $N_i(x)$. Symbolic calculation of $\Sigma_{ij}\mathcal{E}_{ij}$ results in a polynomial in the unknown variables u_i of each finite element, with its coefficients being polynomials in x_1, x_2, x_3. After spatial element integration has been performed as described in Section 2.3.1, only variables u_i are left. To account for the variational problem (2.1), the first derivative with respect to the unknowns u_i is calculated, resulting in $3\,\mathrm{DoF}$ equations for each element in the 3D simulation case.

These polynomials, which share a large number of monomials, can then be assembled into a global system of symbolic equations. Therefore, the local vertex indices are converted into the respective global indices within the whole polynomial. The number of monomials to be evaluated in this system is significantly smaller (about a factor of 3) than the number of monomials contained in the set of element equations. Consequently, multiple evaluations of monomials can be avoided at runtime. In the same way, the Jacobian matrix can be expressed symbolically by applying symbolic derivation and then be evaluated using the current parameter values at runtime. However, this approach has the disadvantage that the whole system of non-linear equations with all polynomials for $\mathbb{K}$ and its Jacobian matrix $\mathbb{K}'$ has to be stored in memory. Due to this reason, for the simulation of large models where the respective system of equations does not fit into main memory, incremental approaches as described by Bathe [Bat02] should be favored. Similar to the corotated simulation, these approaches require the system matrix to be reassembled in every time step. However, in contrast to the corotated simulation, element matrices are not constant any more and have to be rebuilt in every time step, too.

To further improve the evaluation of the polynomials at runtime, all required mono-

mials $\prod_{i,j} u_i^j$ are computed once at the beginning of the simulation step from the current displacement vector u. As these monomials occur in several equations of the system as well as in the Jacobian matrix, an additional speed-up of about a factor of 2 can be achieved. In Figure 2.6, a comparison of the linear, corotational and non-linear simulation exhibiting the same external forces is shown.

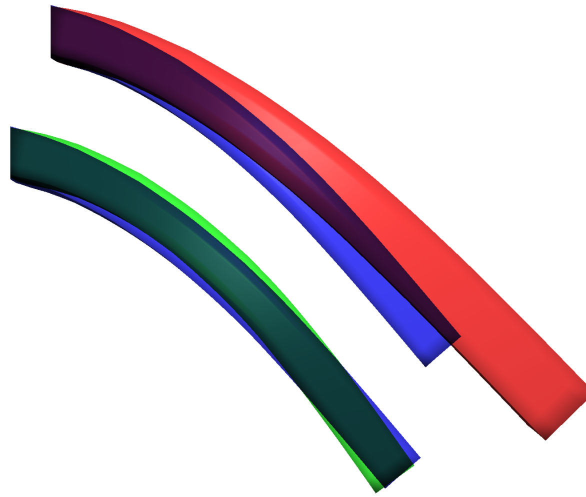


Figure 2.6: *Comparison of the linear, corotational, and non-linear simulation of gravity. The deformation of the latter one is shown as reference in blue color. While the linear Cauchy strain (red color) fails to approximate the deformation properly, only very small differences can be observed in case of the corotational strain (green color).*

2.3.5 Higher-Order Finite Elements

It is well known that the approximation of parameters such as strain and stress across the deformable body tends to produce *locking*. This phenomenon describes the fact that a finite element simulation based on the variational formulation predicts a stiffer behavior of the body than it would arise in reality. Due to this effect, linear finite element approaches using linear shape functions produce significantly different results on different ranges of scale (see Figure 2.7 (a), (b), and (c)). The less elements are used, the stiffer the behavior of the deformable object. To achieve best accuracy, higher-order finite elements such as Serendipity tetrahedra are included in the simulation framework. Although locking effects cannot be avoided entirely by means of such elements, they produce much more realistic simulations of material properties using the same number of vertices as in the linear case (see Figure 2.7 (d), (e), and (f)). In particular, the

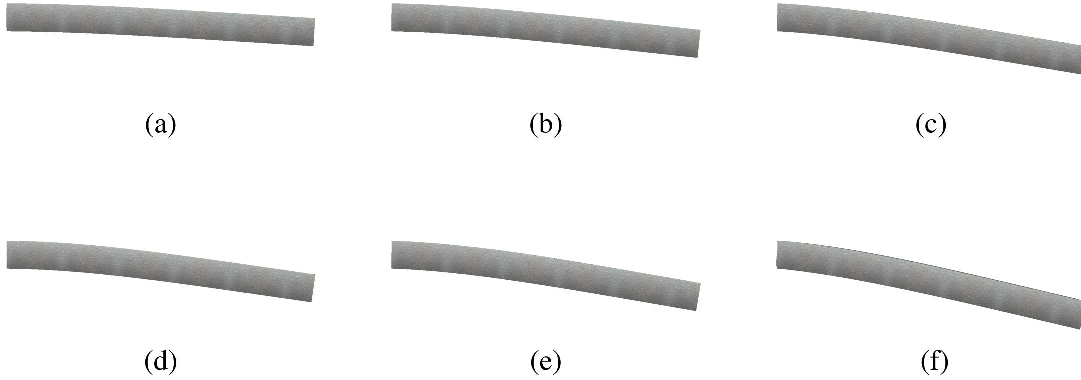


Figure 2.7: *Comparison between linear tetrahedra ((a), (b), and (c)) and Serendipity tetrahedra ((d), (e), and (f)) on different levels of resolution (384 (a), 3072 (b), 24576 (c) resp. 48 (d), 384 (e), 3072 (f) elements) using the Cauchy strain. Note that compared to linear elements, Serendipity tetrahedra introduce additional nodes per element. Thus, to allow for a fair comparison, the same number of nodes is used in both examples. Therefore, every Serendipity element is split into linear tetrahedra (as will be shown in Figure 2.11), resulting in 8 times as many elements.*

simulation based on 384 Serendipity elements produces more accurate results than 24k linear elements (see Figure 2.7).

Calculations

Higher-order finite elements like quadrangular elements or Serendipity tetrahedra are supported in Cauchy strain, corotated Cauchy strain, and Green strain simulation. Pre-calculations can be performed in the same manner as described in Section 2.3.2 and 2.3.4 to build the stiffness matrix K or the system of non-linear equations $\mathbb{K}$. However, since not all partial derivatives of the shape functions $N_i(x)$ are constant, the integration $\int_\Omega B^{\mathrm{T}}DB \ \mathrm{d}x$ of the matrix $B^{\mathrm{T}}DB$ has to be calculated component-wise using the analytical integration method as described in Section 2.3.1. Note that all entries of the matrix $B = B(x)$ are polynomials in x_1, x_2, x_3.

2.3.6 Dynamics

In the previous sections we have focused on computing the static equilibrium state of the displacement field with respect to external forces. We now extend this approach to the simulation of the system dynamics using the Lagrangian equation of motion [Bat02]:

$$M\ddot{u} + C\dot{u} + Ku = f. \tag{2.14}$$

M is called the mass matrix, and C denotes the damping matrix. Consequently, to achieve best accuracy, we avoid simple mass lumping[6] (especially in case of higher-order finite elements) and use the correct matrix instead:

$$M^e = \int_\Omega \rho\, \Phi^{\mathrm{T}}(x)\, \Phi(x)\, \mathrm{d}x.$$

Here, $\Phi(x)$ is the shape matrix as defined (2.11) and ρ is the density of the finite element. Note that ρ is not required to be constant within an element. With the damping constant α, the damping matrix is calculated as $C^e = \alpha\, M^e$. We restrict ourselves to mass-proportional damping here. However, more general Rayleigh damping can be integrated as well. Then, an additional damping constant β is introduced, and C^e is computed as $C^e = \alpha\, M^e + \beta\, K^e$. Both the global mass matrix M and the damping matrix C are built by assembling all single element matrices.

In case of the non-linear simulation, the equation of motion is analogously given as

$$M\ddot{u} + C\dot{u} + \mathbb{K}(u) = f.$$

2.3.7 Time Integration

To solve for the dynamic simulation based on the equation of motion (2.14), a time integration scheme has to be selected. We use discrete time steps $t_0 + k \cdot dt, k \in \mathbb{N}$, to capture the motion of a deformable body starting at time t_0. The temporal derivatives of u, $\dot{u}$ and $\ddot{u}$, have to be approximated appropriately. We distinguish between the implicit Euler scheme and the implicit Newmark scheme.

Implicit Euler Time Integration

Using a finite difference discretization of $\dot{u}$ and $\ddot{u}$, the equation of motion (2.14) leads to

$$M\, \frac{u^{t+dt} - 2u^t + u^{t-dt}}{dt^2} + C\, \frac{u^{t+dt} - u^{t-dt}}{2\, dt} + K\, u^{t+dt} = f^{t+dt}.$$

This equation can be written as $\tilde{K}\, u^{t+dt} = \tilde{f}^{t+dt}$ by updating the right-hand side f^{t+dt} and the stiffness matrix K (or the system of equations $\mathbb{K}(u)$ in the non-linear case) with

[6] Mass lumping: the mass is concentrated on the vertices, thus yielding a diagonal mass matrix.

the appropriate terms:

$$\begin{aligned}\tilde{K} &= K + \frac{M}{dt^2} + \frac{C}{2dt}, \\ \tilde{f}^{t+dt} &= f^{t+dt} + M\frac{2u^t - u^{t-dt}}{dt^2} + C\frac{u^{t-dt}}{2dt}.\end{aligned}$$

To update the force vector $\tilde{f}^{t+dt}$ in every time step, only one additional vector u_{old} storing u^{t-dt} is required. If the matrix C is a scaled version of M, one matrix-vector product and two vector-vector linear combinations are required to update the force vector. In case of Rayleigh damping, an additional matrix-vector product and vector addition is needed. After updating the force vector, u_{old} is set to u^t for the next time step. Then, the system of equations $\tilde{K}\,u^{t+dt} = \tilde{f}^{t+dt}$ is solved for u^{t+dt}.

Implicit Newmark Time Integration

The Newmark scheme uses another discretization of $\dot{u}$ and $\ddot{u}$ to transform the equation of motion (2.14) into a set of difference equations. A second-order-accurate implicit Newmark scheme avoids artificial damping typical to implicit Euler integration. Second order approximation is obtained by setting the parameters adequately (more details can be found in [Bat02, Wil98]):

$$\begin{aligned}\dot{u}^{t+dt} &= \dot{u}^t + \left(0.5\,\ddot{u}^t + 0.5\ddot{u}^{t+dt}\right)\,dt, \\ u^{t+dt} &= u^t + \dot{u}^t\,dt + \left(0.25\,\ddot{u}^t + 0.25\,\ddot{u}^{t+dt}\right)\,dt^2.\end{aligned}$$

By replacing $\dot{u}^{t+dt}$ and $\ddot{u}^{t+dt}$ (deduced from the second equation),

$$\ddot{u}^{t+dt} = 4\frac{u^{t+dt} - u^t - \dot{u}^t dt}{dt^2} - \ddot{u}^t,$$

in the equation of motion (2.14), the system of algebraic equations $\tilde{K}u^{t+dt} = \tilde{f}^{t+dt}$ (or $\tilde{\mathbb{K}}(u^{t+dt}) = \tilde{f}^{t+dt}$ in the non-linear setting) reads as

$$\begin{aligned}\tilde{K} &= K + \frac{4M}{dt^2} + \frac{2C}{dt} \\ \tilde{f}^{t+dt} &= f^{t+dt} + M\left(\frac{4u^t}{dt^2} + \frac{4\dot{u}^t}{dt} + \ddot{u}^t\right) + C\left(\frac{2u^t}{dt} + \dot{u}^t\right).\end{aligned}$$

If the matrix C is a scaled version of M, one matrix-vector product, a component-wise linear combination of u^t, $\dot{u}^t$, and $\ddot{u}^t$, and a vector addition is required to determine $\tilde{f}^{t+dt}$ in every time step. In case of Rayleigh damping, an additional matrix-vector product

and vector addition is required. The system of equations $\tilde{K}\, u^{t+dt} = \tilde{f}^{t+dt}$ is solved for u^{t+dt} in every time step. In contrast to the Euler time integration scheme, additional vectors storing $\dot{u}^t$ and $\ddot{u}^t$ have to be kept and updated in every time step.

The Problem of Explicit Methods

Explicit time integration schemes determine the unknown u^{t+dt} by approximating the elastic force Ku^t based on the displacement field of the previous time step. Solving a system of equations can be avoided, if M and C are approximated by mass-lumping. The explicit Euler method yields:

$$u^{t+dt} = \left(\frac{M}{dt^2} + \frac{C}{2dt}\right)^{-1} \left(f^{t+dt} - Ku^t + M\frac{2u^t - u^{t-dt}}{dt^2} + C\frac{u^{t-dt}}{2dt}\right).$$

However, the stability of explicit schemes is limited by the Courant condition, which restricts the size of the time step dt:

$$dt < h\sqrt{\frac{\rho}{\lambda + 2\mu}}.$$

More details can be found in the contributions by Debunne et al. [DDBC99] and the original work by Courant et al. [CFL28]. Here, h denotes the smallest distance of two vertices in the reference configuration, ρ is the density of the object in its reference configuration and λ, μ are the Lamé coefficients. Even worse, the time step depends on the stiffness and density of the underlying material. In fact, the stiffer the simulated materials are, the smaller time steps are required. This means that a large number of simulation steps has to be performed to accurately simulate very stiff materials. On the one hand, this increases the simulation time significantly. On the other hand, simulation gets numerically instable, because numerical rounding errors accumulate very quickly if several thousands of time steps are performed per second to simulate real physical materials. The problems aggravate if larger simulation meshes are used since h is smaller in this case. Therefore, explicit time integration can in general not provide high update rates and numerical stability. Constant update rates—as they are typically required in interactive applications—cannot be achieved if different materials are used. In contrast, implicit schemes are numerically more complex but allow for larger time steps. As we will show, by using our advanced multigrid scheme presented in Section 2.4 for the implicit time integration, the simulation of physical materials can be performed at significantly lower computational costs, and it clearly outperforms explicit methods for stiff materials.

2.3.8 Boundary Conditions

Vertex Fixation

For the task of object deformation it is essential that particular mesh vertices or even regions can be fixed in order to restrict the effects of the applied deformations. This is accomplished by zeroing all entries in the respective rows of the system matrix K (or the system of equations $\mathbb{K}$) a vertex belongs to and by setting the respective vertex force to zero. To keep our numerical schemes stable, the matrix is kept at full rank. Therefore, the diagonal elements in the intersection of the respective rows and columns are set to a base stiffness value E_I specified for the entire object. In this way the matrix structure does not have to be changed, and the update operation for single rows can be implemented very efficiently.

Moreover, we cannot only fix a vertex at its initial position but at any deformed position, too. As a deformation is specified by a global force field acting on the reference configuration, a vertex can be fixed by simply associating a constant force to it. This means that one can first deform the model as desired, yielding the appropriate force vectors, and then fix parts of the deformed model by freezing the respective force vectors. In summary, for a fixed vertex i at position v_i we have the constraint $E_I \cdot u_i = f_i$. If $f_i \neq 0$ the vertex is fixed at the deformed position $v_i + u_i = v_i + \frac{1}{E_I} f_i$.

Gravity

It is straightforward to consider volume forces such as gravity in the simulation. The integral

$$\int_\Omega g^{\mathrm{T}} u \; \mathrm{d}x$$

from the variational formulation of the elasticity problem (2.1) defines the contribution of gravity to the potential energy. Using the shape function of the finite element, the displacement field can be approximated as usual (2.10). Given the shape matrix $\Phi(x)$ (2.11) the integral is then written as

$$\int_\Omega g^{\mathrm{T}} \Phi(x) \, u^e \; \mathrm{d}x = \left(\int_\Omega g^{\mathrm{T}} \Phi(x) \; \mathrm{d}x \right) u^e.$$

Taking the first derivative of the variational problem with respect to u^e yields the volume force $\int_\Omega \Phi^{\mathrm{T}}(x) \, g \mathrm{d}x$. Here, g is the acceleration of gravity multiplied by the density of an infinitesimal small volume element. For homogeneous elements the density is

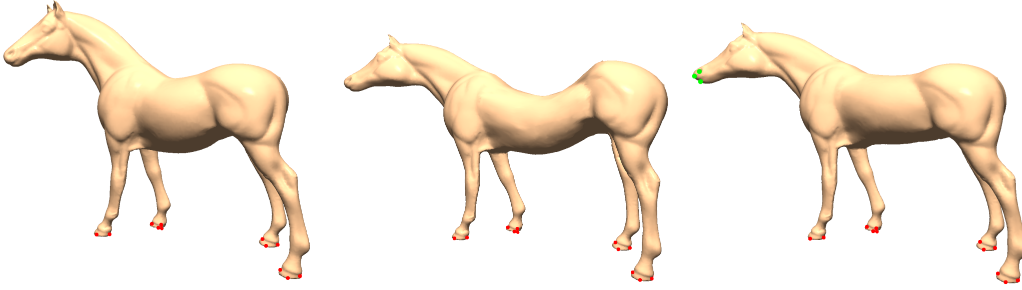


Figure 2.8: *Vertices can also be fixed at deformed positions. On the left, the horse model in its reference configuration is shown. In the middle, a deformation is applied by specifying external forces. On the right, the result is shown after the green vertices have been fixed and all external forces have been released. Due to the vertex fixation forces, the object is kept in its deformed state. Note, that red vertices in all images illustrate vertices that are fixed in their reference configuration.*

constant, and g can be factored out of the integral. For each finite element, the contributions of all vertices belonging to that element are accumulated into the global gravity vector taking the global vertex indices into account.

External Point Forces

In addition to volume forces, external forces can be specified for every vertex of the simulation mesh. These forces are directly added to the respective entries in the global force vector. Since in animations it is often preferable to apply external forces smoothly, vertex forces are optionally scaled with a weighting function $\omega(t)$. This function scales the forces from zero to one over a number of time steps, thus avoiding any unnatural jumps in the animation.

Haptic Input Devices

Our simulation framework is also coupled with a force feedback device, the SensAble's PHANTOM®. As such devices require high update rates for the force feedback, an extra thread is spawned to communicate with the device. Note that the real force vector f^t has to be used for force feedback instead of the updated right-hand side of the time discretization $\tilde{f}^t$. The force feedback for the current step f^{t+dt} is extrapolated based on the last two force fields f^t and f^{t-dt}:

$$f^{t+\Delta t} = f^t + \frac{\Delta t}{dt}\left(f^t - f^{t-dt}\right) \quad \forall 0 \leq \Delta t \leq dt.$$

From the interface of that device, a 3D movement vector is determined, which is transformed into object space to apply the appropriate forces. This movement vector is used to add an impulse to the dynamic system, which is proportional to the movement since the previous time step. Based on the respective discretization scheme, the previous time step's displacement vector u^t is updated to account for the specified movements. Implicitly, this comes down to an additional impulse in the dynamic system, which is also incorporated in the current force feedback field.

Using a force feedback device also allows to "feel" the forces that are applied by the user. This is a very important property in applications such as surgical training, where the surgeon should be trained with the real forces required by a real intervention.

So far, the positional information is only used heuristically to control the deformation process by adding an impulse. If it is required to match the specified control points exactly—the user moves a mesh vertex to a specific position—, a mixed boundary formulation of the finite element method has to be used. It comes at the expense of matrix updates if the user moves different vertices of the mesh. Thus, costly updates are likely to occur in every simulation step. In the case of corotated strain—where the matrix is reassembled anyway in every step—mixed boundary conditions might be integrated at lower costs. Mixed boundary conditions will be discussed in detail in the next section.

2.3.9 Mixed Boundary Conditions

So far, the deformation process was exclusively controlled by external forces. If mixed boundary condition are applied, the mesh vertices are basically split into two parts: vertices, for which displacements are given, and vertices, for which forces are given. For example, if for a number of vertices a displacement is specified, and we set the forces of all other vertices to zero, we obtain a deformed configuration that matches the specified displacements exactly, while the other parts of the object are deformed according to the physical model.

More precisely, the system of linear equations

$$\begin{pmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

with known displacements u_1 and known forces f_2 but unknown displacements u_2 and

unknown forces f_1 has to be solved. A solution can be determined in two steps:

$$\begin{aligned} \text{Solve for } u_2: \quad K_{22}u_2 &= f_2 - K_{21}u_1 \\ \text{Compute } f_1: \quad f_1 &= K_{11}u_1 + K_{12}u_2 \end{aligned}$$

To avoid the reconstruction of a sparse matrix data structure every time the boundary conditions change, the system matrix can be updated by setting the respective entries to zero; thus, the solution process is restricted to the sub block K_{22} of the matrix. Note, that a copy of the matrix is kept to account for the sub blocks K_{11}, K_{12}, and K_{21}. More details about the specific matrix format can be found in Section 2.5.1.

Mixed boundary conditions cannot only be used to improve the positional accuracy of input-device-driven deformations, but they also allow for more advanced interaction schemes. Skeleton-driven deformation would be an application of mixed boundary conditions. By specifying a skeleton in the volumetric mesh, displacement vectors u_i at the respective skeleton vertices can be determined by an arbitrary skeleton animation technique. These displacements are used as boundary conditions for the deformation problem, and any other regions in the deformation mesh can be influenced by external forces, too. More details can be found in various skeleton-driven animation publications such as [CGC$^+$02a].

2.3.10 Material Update

Changing Element Stiffness Values

An important feature a deformation tool should provide is the possibility to flexibly adjust stiffness values. As we aim at assigning these values interactively while the simulation is running, the simulation method must be able to instantaneously react to such changes.

Fortunately, it can be observed that the element matrices K^e do not have to be rebuilt if only the stiffness E of the respective finite element has to be changed. This is because Σ is a multiple of the elastic modulus E in the equation of the material law (2.5). Thus, E can be factored out of the element matrix. This means that the element matrix only has to be scaled by the factor E_{new}/E_{old}, and the global stiffness matrix has to be reassembled to account for the new stiffness values. As the matrix structure does not change, this update can be achieved efficiently. In the corotated setting, where the system matrix has to be reassembled in every simulation step anyway, updating stiffness values comes at no additional costs.

Plasticity Simulation

So far, our discussion was restricted to the deformation of purely elastic materials. In shape deformation, however, the user does not expect the object to move back into its reference configuration once the control handles are released. To avoid this behavior, forces induced by the user can be accumulated into a global force field, and the resulting displacements of the grid points are computed just at the very end of the user interaction. This is advantageous because the system matrix does not have to be updated in order to account for the plasticity. Instead, we consider the system of linear equations in the form

$$K\,(u_{plastic} + u) = f_{plastic} + f$$

where $K\,u_{plastic} = f_{plastic}$ are the plastic deformations computed so far, and f are the forces applied by the current user input. The plastic forces $f_{plastic}$ are determined empirically. Alternatively, they can be defined by the user to keep the object in a specific state. In this case, all forces acting at the time of the user input are copied to the plastic forces $f_{plastic}$.

Note that physically accurate simulation of plasticity could be integrated in the framework as well. Then, plasticity is determined from the current stress (e.g. using the von Mises stress norm), and the variational formulation is extended to account for plasticity. More details can be found in the book by Bathe [Bat02].

2.3.11 Modified Material Laws

We have focused on the physical behavior of the simulation of deformable bodies. In shape manipulation, however, the user possibly wants to achieve a particular, intuitive deformation that typically does not obey the physical model. For example, as-rigid-as-possible deformations introduced by Alexa et al. [ACOL00] are a well-established method for shape manipulation. They are characterized by a minimum amount of scaling and shearing to enforce rigidity. However, as-rigid-as-possible deformations contradict physical behavior. For instance, volume preservation is no longer a preferable property. In this section, we thus introduce modified material laws that mimic this rigid behavior. Moreover, we can control to either favor rotations or shearing of the material. These material laws extend the proposed physical finite element framework towards a flexible shape manipulation tool.

For instance, such modified material laws are of great interest in the context of image deformation as has been shown in a joined work with Thomas Schiwietz [SGW07]. There, we have presented a flexible image deformation technique using different mate-

rial laws. In combination with image segmentation algorithms to quickly fixate vertices or assign stiffness values to different parts of the image, an intuitive and flexible user interface for image deformation tasks has been demonstrated. Figure 2.9 shows some images generated with this approach. Now, we extend the material laws to 3D.

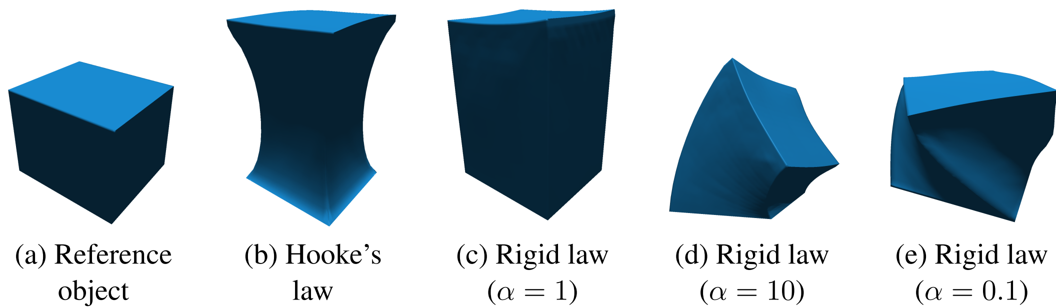
In an isotropic and fully elastic body stress (Σ) and strain ($\mathcal{E}$) tensors are coupled through Hooke's law (linear material law). As observed by many authors, shape deformations obeying physical laws are often not desired when manipulating objects. For example volume preservation as enforced by Hooke's law is a physical phenomenon that contradicts shape-preserving as-rigid-as-possible deformations. Volume preservation, on the other hand, can be avoided by prohibiting transversal contractions of the material being deformed. This is achieved by appropriately varying Poisson's ratio ν in Hooke's law (2.5), which defines the ratio of transverse contraction to longitudinal stretching in the direction of the force. In particular, setting $\nu = 0$ yields $\lambda = 0$; thus, any curvature in a direction perpendicular to the direction of stretching or bending is avoided. In Figure 2.9 and 2.10 we compare the deformation of a material under stretching with (b) and without (c) transversal contractions in the 2D and 3D case.

To enable as-rigid-as-possible transformations, we further enforce the physical simulation with respect to an anisotropic material law. By doing so we enable the user to flexibly control the resulting deformations by continuously varying between rigid and highly elastic materials within one object. The anisotropic material law is simulated by adding a scaling factor α to the off-diagonal elements in the stress tensor, yielding the *rigid* material law

$$\Sigma = E \begin{pmatrix} \mathcal{E}_{11} & \alpha\, \mathcal{E}_{12} & \alpha\, \mathcal{E}_{13} \\ \alpha\, \mathcal{E}_{21} & \mathcal{E}_{22} & \alpha\, \mathcal{E}_{23} \\ \alpha\, \mathcal{E}_{31} & \alpha\, \mathcal{E}_{32} & \mathcal{E}_{33} \end{pmatrix} \tag{2.15}$$

(a) Reference image (b) Hooke's law (c) Rigid law ($\alpha = 1$) (d) Rigid law ($\alpha = 100$) (e) Rigid law ($\alpha = 0.1$)

Figure 2.9: *Modified material laws in 2D: Images (b) and (c) show the difference between Hooke's law and the rigid law if the castle is stretched vertically. Images (d) and (e) demonstrate the effects of reducing the amount of shearing (d) and rotation (e) while the castle is dragged to the right.*



(a) Reference object (b) Hooke's law (c) Rigid law ($\alpha = 1$) (d) Rigid law ($\alpha = 10$) (e) Rigid law ($\alpha = 0.1$)

Figure 2.10: *Modified material laws in 3D: Images (b) and (c) show the difference between Hooke's law and the rigid law if the cube is stretched vertically. Images (d) and (e) demonstrate the effects of reducing the amount of shearing (d) and rotation (e) while the cube is picked on a single corner.*

with E being the elastic modulus.

The modified material law allows for transformations minimizing the amount of shearing or rotation, respectively, and it can thus effectively be used to produce as-rigid-as-possible transformations. For a value of $\alpha = 1$ the material law is isotropic. By decreasing the value of α, the internal stress is reduced in direction of x_2 (and x_3) if strain is induced in direction of x_1, and vice versa. Consequently, such a setting favors shearing instead of rotation. Contrarily, by setting α to a value larger than 1 rotations instead of shearing will be favored. The effects of different values of α are demonstrated in Figure 2.9 (d) and (e) for the 2D case and in Figure 2.10 (d) and (e) for the 3D case. Note that generally the rigid material law does not preserve volume. However, a larger value of α leads to volume growth if shearing forces are applied, while small values of α tend to preserve volume in this case.

Updating Material Laws

Not only the stiffness values can be updated, but also the material law considered for each element. In this case, the element matrices have to be rebuilt (including time-consuming integration in case of higher-order finite elements). Therefore, the update process is not interactive in general. However, if only a small number of elements is changed, the update can still be achieved at interactive rates.

2.4 Implicit Multigrid Solver

In this section, we present an advanced numerical solver for the governing equation of motion of elastic materials. From the discussions in the last section, we can derive some important properties the solver must take into account:

1. The matrices derived from typical finite element meshes are scattered (random) sparse. Therefore, efficient data structures and algorithms for these matrices are required in the solution process.

2. The system matrix is frequently updated, i.e., in every time step. Fortunately, the structure of the matrices stays the same as long as the topology of the simulation mesh does not change. Thus, data structures and algorithms to efficiently support matrix updates are required, thereby enabling a simulation framework that provides the linear, corotated, and non-linear setting likewise.

In the following, we present a numerical multigrid solver operating on different grid resolutions. We start with an introduction to multigrid techniques, and we then describe our approach including the generation of appropriate mesh hierarchies. In Section 2.5 we will show how the multigrid solver can be efficiently implemented to address the aforementioned requirements.

2.4.1 The Multigrid Idea

Multigrid methods provide a general means for constructing scalable linear solvers for elliptical partial differential equations [Bra77, Hac85, BHM00]. In the graphics community, multigrid methods have just recently gained attention in a number of different applications [BFGS03, AKS05, SYBF06]. Multigrid methods exploit the fact that a problem can be solved on different scales of resolution. Two observations are of major importance in such approaches. First, a basic property of many iterative solvers for systems of linear or non-linear equations is smoothing. Many relaxation methods such as Gauss-Seidel reduce high frequencies in the error very quickly while low frequencies are damped rather slowly. Second, the remaining low-frequency errors can be accurately and efficiently solved for on a coarser grid. A multigrid strategy combines both observations by transferring the smoothed error to the coarser grid and vice versa. Recursive application of this basic idea to each consecutive system on a hierarchy of grid levels leads to a multigrid V-cycle.

For the efficient simulation of an elastic deformable solid we have developed a

geometric multigrid method[7]. This method includes geometry-specific restriction and interpolation operators. These operators form the essential multigrid components as they are used to transfer quantities between different levels of the object hierarchy.

In this work, we define an appropriate finite element hierarchy, which allows for an efficient implementation of multigrid components. The result is a method that uniformly damps all error frequencies with a computational cost that depends only linearly on the problem size.

2.4.2 Nested Hierarchies

The geometric multigrid method requires a mesh hierarchy that represents the deformable object at different levels of resolution. On this hierarchy, appropriate transfer operators to map quantities between different levels have to be designed. Starting with a finite element mesh at the coarsest resolution level, a common way to construct the hierarchy in a top-down approach is to split every element as shown in Figure 2.11. By subdividing all elements synchronously, T-vertices[8] can be avoided. This approach results in a nested hierarchy, which allows the transfer operators to be defined in a straightforward way, but it requires the initial mesh to be fine enough to achieve a proper representation of the object's boundary at ever finer resolution levels. Furthermore, subsequent subdivisions might lead to a fine mesh containing ill-shaped tetrahedra that are not well-suited for finite element simulations. Analogously, not only tetrahedral elements can be refined uniformly but all element types presented in Section 2.3.1. Figure 2.11 gives an overview of the subdivision performed in each of the cases.

For all element types, the transfer operators can be defined by interpolation along the element edges. For most element types this yields a linear interpolation scheme accounting for two attributes of the coarse grid. In cases of quadratic quads or Serendipity tetrahedra, quadratic interpolation, which takes three attributes into account, may be performed to achieve better approximations. The respective restriction operator R is defined by the inverse interpolation scheme I. In matrix notation, it is defined by the transposed interpolation matrix $R = I^{\mathrm{T}}$.

For volumetric objects given on a Cartesian grid this generation of nested hierarchies does not pose a restriction, because the entire domain is covered for every level. If we start with a coarse representation of an arbitrary object, however, rather shallow hierarchies are constructed and the multigrid method cannot be used to its full potential. In addition, because the subdivision scheme does not account for the object's surface

[7]Geometric multigrid methods determine the restriction and interpolation operators from geometric properties.

[8]A T-vertex is a vertex on an edge of an element that is only considered by adjacent elements.

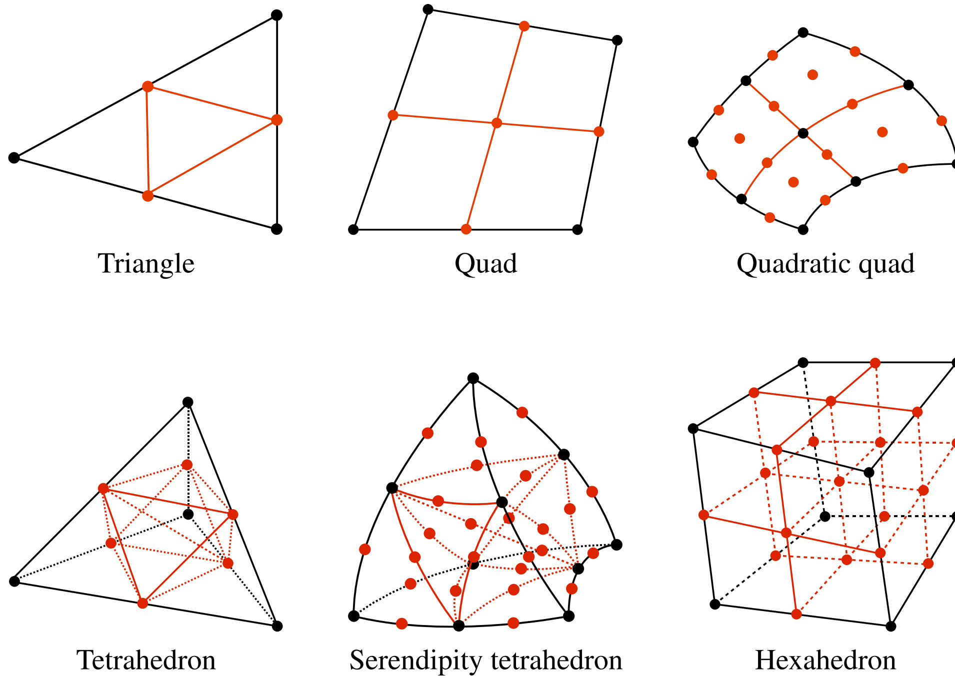


Figure 2.11: *Regular subdivision scheme for various finite element types. The new vertices and edges are highlighted in red.*

at the finest scale, it leads to poor visual results, and the simulation cannot achieve full accuracy due to the improper object approximation.

2.4.3 Non-Nested Hierarchies

To avoid the previously mentioned drawbacks, we propose linear transfer operators that do not require a nested hierarchy and can be integrated efficiently into the multigrid scheme. These operators establish relations in a multilevel hierarchy of unstructured and unrelated meshes by means of barycentric interpolation as illustrated in Figure 2.12.

Initially, we start with a coarse mesh H and a fine mesh h. For every tetrahedron in H, all vertices of h inside this tetrahedron are determined. The barycentric coordinates of these vertices with respect to the circumscribed element are calculated, and they are used as interpolation weights to map values from the coarse grid to vertices on the fine grid via the interpolation operator I_h. Each fine grid vertex stores the respective coarse grid vertices and corresponding weights. For vertices in h that lie outside the coarse mesh, barycentric coordinates to the closest tetrahedron in the coarse mesh are computed. The restriction operator that is required in the multigrid method to gather

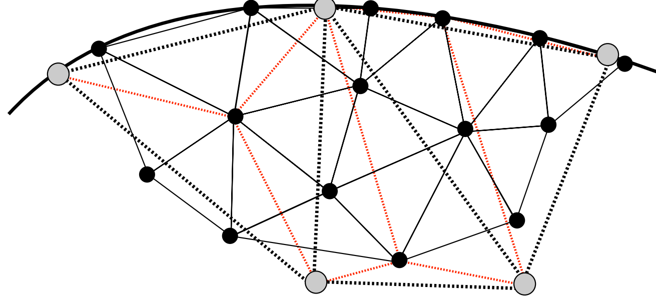


Figure 2.12: *Geometric relations between elements in a non-nested hierarchy are illustrated (the 2D case is shown for simplicity). Dotted and solid lines indicate the coarse and the fine mesh respectively. For some fine mesh vertices, barycentric interpolation weights are highlighted by dotted red lines.*

values from a finer resolution level is determined from the interpolation operator $R_h = I_h^{\mathrm{T}}$.

For higher-order elements, the respective finite element shape functions can be used instead of barycentric interpolation. However, they usually yield a larger support of the interpolation kernel, which can affect the solver's performance. Therefore, a higher-order element can be split into smaller linear elements. These linear elements can be completely defined by the vertices of the higher-order element, and barycentric interpolation within such elements can be used to define the interpolation operator. Of course, edges are approximated linearly and thus the interpolation is not as precise as in the previous case. On the other hand, multigrid solvers are typically not too sensitive to the interpolation scheme, and it turned out that the simpler and faster interpolation schemes lead to good results, too.

2.4.4 Coarse Grid Correction

The essential step in the multigrid method is the coarse grid correction. The equation to be solved on the finer grid with grid size h is

$$K^h u^h = f^h.$$

Using a coarser grid with grid size H, the defect equation $K^H e^H = r^H$ can be considered in addition to the fine grid defect equation (given an initial solution $\hat{u}^h$)

$$K^h e^h = r^h = f^h - K^h \hat{u}^h.$$

K^h and K^H are the system matrices on the fine and the coarse grid, respectively. e^h and e^H are the absolute errors between the exact solution and the approximate solution on either grid. r^h and r^H are the residuals on these grids. To establish a relation between e^h and e^H, and between r^h and r^H, the linear transfer operators I_h and R_h are employed. As I_h can be replaced by $I_h = R_h^{\mathrm{T}}$, only the restriction operator is required by the coarse grid correction scheme.

Given the linear transfer operator R_h, as well as an initial approximation $\hat{u}^h$ of the displacement values on the fine grid h of the deformable solid, a new approximation u^h can be computed as shown in Algorithm 1.

Algorithm 1 Two grid correction

① Pre-smooth	Relaxation of $\hat{u}^h$
② Compute residual	$r^h = f^h - K^h\hat{u}^h$
③ Restrict residual to coarse grid	$r^H = R_h r^h$
④ Solution on coarse grid	$K^H e^H = r^H$
⑤ Transfer correction	$e^h = R_h^{\mathrm{T}} e^H$
⑥ Correction	$u^h = \hat{u}^h + e^h$
⑦ Post-smooth	Relaxation of u^h

The relaxation of $\hat{u}^h$ in stage ① avoids the transfer of quantities from the fine grid that cannot be reduced on the coarse grid, and the relaxation of the result u^h in stage ⑦ avoids high frequencies introduced by numerical inaccuracies. In general, 1–2 Gauss-Seidel steps are sufficient for both pre- and post-smoothing.

V-Cycle By recursive application of the coarse grid correction to stage ③ and by using a pre-conditioned conjugate gradient method to compute the solution of stage ③ on the

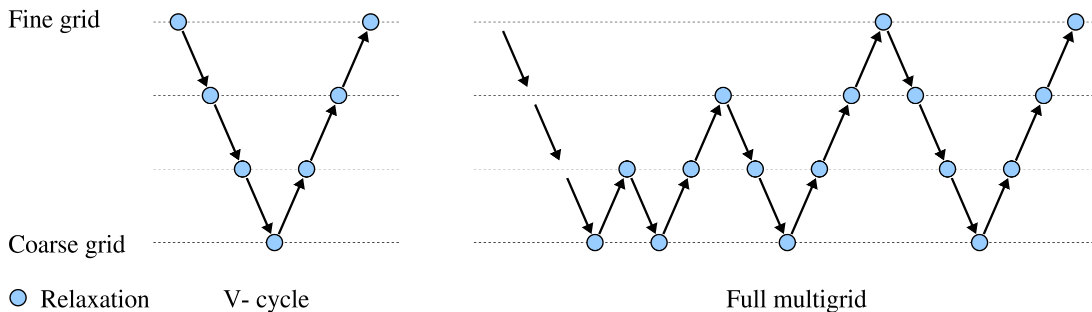


Figure 2.13: *Illustration of the computations performed within a single multigrid V-cycle (left) and a full multigrid algorithm with $\mu = 1$ (right). The arrows indicate transfer operations. The circles denote relaxation steps.*

coarsest grid, a full multigrid V-cycle is derived.

Full Multigrid Algorithm (FMG) The idea of the full multigrid algorithm is to calculate a good initial guess by solving the problem on the coarser grids, where the number of grid points is typically small compared to the fine grid. By interpolating this solution on a finer grid, an approximate solution can be found very efficiently. On the fine grid, this solution can then be corrected by one (or generally μ) multigrid V-cycle. This leads directly to a multigrid scheme as illustrated in Figure 2.13.

2.4.5 Galerkin Property

A purely geometric multigrid approach typically determines the coarse grid matrices by discretizing the problem at the respective coarser grids. With the restriction and interpolation operators we have deduced, this approach fails to converge quickly in some settings as will be shown in Section 2.4.6. In our multigrid approach, we construct the coarse grid matrices in a different way. Given the fine grid equation

$$K^h u^h = f^h,$$

we build a coarse grid equation by multiplying with R_h from the left and utilizing $u^h = R_h^{\mathrm{T}} u^H$ and $f^H = R_h f^h$:

$$\underbrace{R_h K^h R_h^{\mathrm{T}}}_{K^H} u^H = f^H.$$

In fact, the interrelationship $K^H = R_h K^h R_h^{\mathrm{T}}$ is known as Galerkin property [BHM00]. In our approach, for all but the finest hierarchy level the system matrices are computed as

$$K^H = R_h \, K^h \, R_h^{\mathrm{T}}.$$

The Galerkin property guarantees a consistent calculation on the different levels of resolution with respect to the grid transfer operators. Thus, it assures optimal convergence of the multigrid scheme for both nested and non-nested geometric hierarchies, and it allows us to use unstructured and irregular meshes. On the other hand, this approach requires complex numerical operations to build the coarse grid matrices. For that reason, we have developed a novel algorithm for this task, and we will present it in Section 2.5.

2.4.6 Convergence and Numerical Error

Geometric Multigrid

In this section, we show in more detail why a purely geometric multigrid approach does not work well in our case. It constructs the coarse grid matrices by applying the same discretization scheme as for the finest matrix. In contrast, the Galerkin approach builds the coarse grid matrices by applying the Galerkin property. Figure 2.14 demonstrates the different convergence behavior for various models. All examples are based on a two-grid correction step using only two different hierarchy levels. Given an initial tetrahedralization of some hundred elements, a regular subdivision was performed to build the finer grid. As shown, although all models are based on nested and collateral grids, the purely geometric multigrid diverges for some examples. This is related to the irregular structure of the underlying base meshes. If the structure is regular (i.e., all vertices have the same valence) as for example in the bridge model, the Galerkin and geometric multigrid show the same behavior.

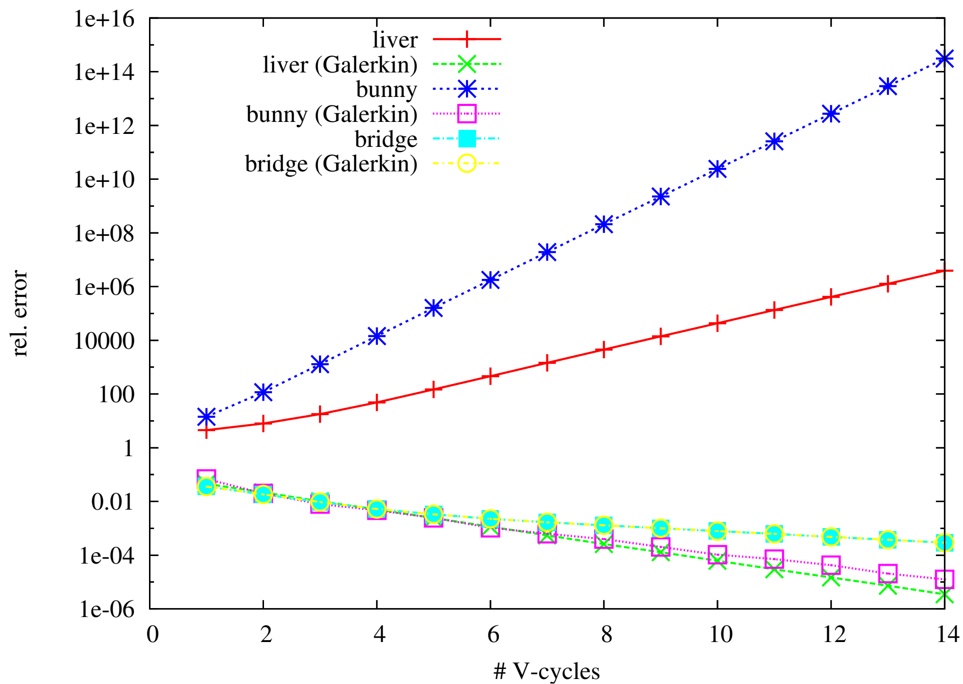


Figure 2.14: *We analyze the relative error using a two grid correction for various nested models. The Galerkin approach always shows good convergence while the geometric approach only works well for regular meshes (bridge example). In this case, the Galerkin approach basically comes down to the purely geometric approach.*

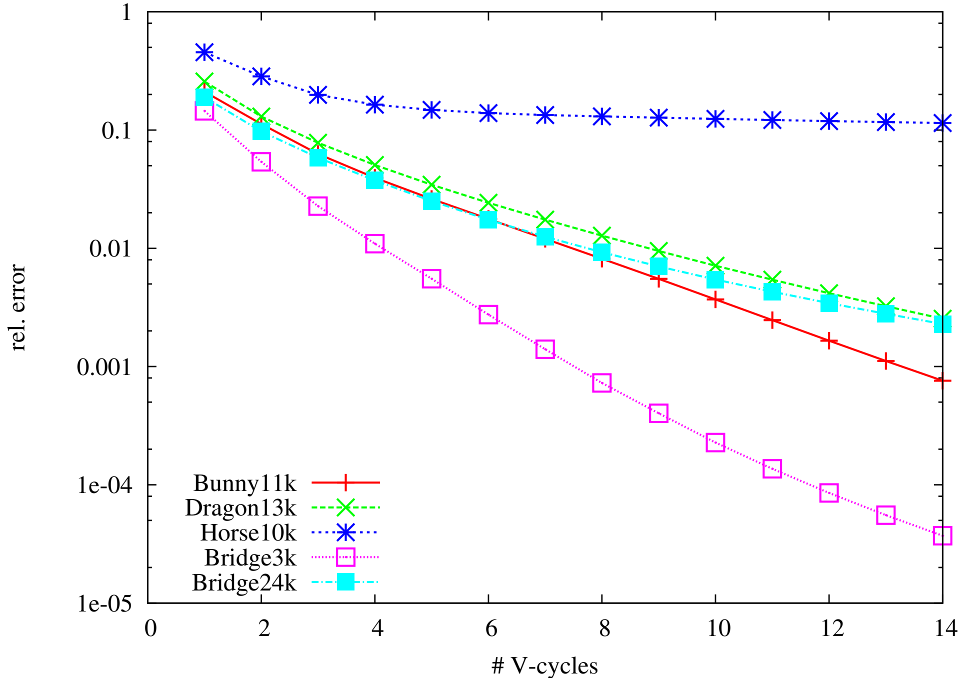


Figure 2.15: *Analysis of the algebraic error of the multigrid method: The relative error is plotted against the number of V-cycles. Note that the convergence strongly depends on the models and the underlying mesh hierarchies. In case of the horse model, an unsuited hierarchy is used. Therefore, fast convergence cannot be achieved.*

We focus on the Galerkin approach in the remaining parts of this thesis. As a side note let us mention that, in contrast to the geometric approach, the Galerkin approach can handle non-nested mesh hierarchies as well. Therefore, it enables more flexibility in the mesh generation procedure.

Galerkin Multigrid

We prove the effectiveness of the Galerkin multigrid solver by analyzing the numerical error for several models using nested and non-nested hierarchies. In all of our tests, we have simulated gravity for the respective object while it was fixed at some vertices. Figure 2.15 plots the numerical error against the number of multigrid V-cycles.

To measure the relative error after each V-cycle, we first calculate the exact solution u, and we then determine the relative error ε of the current approximation $\tilde{u}$ as

$$\varepsilon = \frac{\|u - \tilde{u}\|_2}{\|u\|_2}.$$

As shown in Figure 2.15, the best convergence rates can be achieved when nested hierarchies are applied (bridge models). Due to the limited floating-point precision of 64 bit, numerical errors accumulate noticeably for stiff matrices and larger models. Therefore, many more V-cycles are required in the bridge24k simulation to yield the same relative error as in the bridge3k simulation. For non-nested hierarchies, convergence rates are typically very good. In case of the horse model, however, the convergence is poor due to an unsuited mesh hierarchy, i.e., the vertices of the coarse grid are not a uniform thinning of the fine mesh vertices. Nonetheless, good results are obtained. Despite the poor condition number of the heterogeneous model, the Galerkin multigrid approach converges, and the numerical error after three V-cycles is small enough to be invisible in the rendering of the object. This is due to the fact that the multiscale approach tends to distribute the numerical error uniformly across all vertices.

Faster convergence can be achieved with the *Full Multigrid algorithm* (FMG), where an initial guess is calculated from the coarse grids. Note that in dynamic simulations

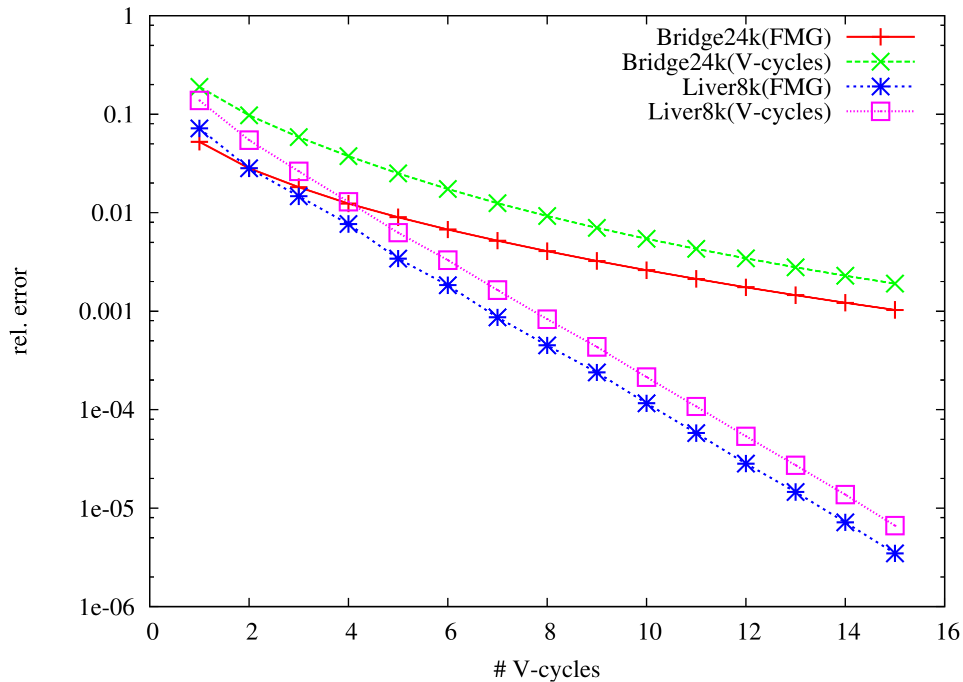


Figure 2.16: *Convergence of the multigrid V-cycles and full multigrid algorithm: The relative error is plotted against the number of V-cycles / parameter μ for FMG. Note that the convergence strongly depends on the models and the underlying mesh hierarchies. In all examples, gravity is applied yielding a poor initial guess for the solver. The FMG results in significant smaller errors, especially in the first steps. Taking into account that the computational costs of one FMG step are roughly as much as two V-cycles, the FMG algorithm is not always advantageous.*

the result of the previous time step is usually an appropriate initial guess, and thus a full multigrid algorithm is not necessary. On the other hand, to analyze the numerical strength of the multigrid scheme, it is worth analyzing the FMG scheme. In cases of rapid changes in the right-hand side of the system of equations, for example if gravity is switched on, the FMG algorithm can achieve slightly better convergence rates as shown in Figure 2.16.

2.5 Fast Sparse Matrix-Matrix Products

Up to now we have described a multigrid approach that uses geometric restriction and interpolation operators on nested and non-nested hierarchies. In particular, this approach respects the Galerkin property to determine coarse grid matrices and thus to guarantee optimal convergence for the given transfer operators. As we have shown, many applications require frequent updates of the system matrix, which implies that the respective coarse grid matrices have to be updated, too. Now, we present a novel approach to perform these update operations very efficiently. In particular, we describe several algorithms that are optimized to build a multigrid hierarchy based on the Galerkin property. We also analyze the algorithms with respect to *in-place* updates, where the structure of the matrices is determined symbolically in a pre-processing step, and only the values of the non-zero entries are replaced in the update operation. These algorithms are a general means for building multigrid solvers for a wide range of sparse matrix problems. Moreover, the algorithms can be easily modified to accommodate efficient calculations of sparse matrix-matrix products where only one matrix is substance to changes.

To achieve improved memory and computational complexity, sparse matrices are stored in optimized data structures. At the core of sparse matrix operations are optimized data structures such as the Yale sparse matrix format [Gus78, EGSS82] and improvements thereof [McN83, BS87]. Due to the low compute-to-memory ratio of sparse matrix computations, formats that try to store dense blocks were developed [BW99, PV05]. Additionally, parallelization strategies for sparse matrix vector products have been discussed [BM05]. Recently, fast sparse matrix algorithms have been analyzed theoretically [YZ05]. It is worth noting that despite the advances in the development of sparse matrix data structures and algorithms, sparse matrix-matrix products are still not standard in sparse libraries, e.g. PETSc [BBE$^+$04].

In the accurate numerical simulation of deformable bodies using multigrid schemes, one of the main computational challenges is the frequent update of the system matrix.

If the corotated strain formulation is used, element rotations are applied in every simulation step, which requires to reassemble the entire system matrix. As long as the topology of the simulation mesh does not change, the structure of the system matrix and of all matrices of the multigrid hierarchy remain unchanged. Therefore, while the data values stored in the sparse matrix data structure have to be updated in every simulation step, the data structure itself can be reused. To update the multigrid hierarchy, the Galerkin property is exploited. For that reason, we now focus on the fast calculation of sparse matrix products $E = R\,K\,R^{\mathrm{T}}$, where both R and K are given in a sparse matrix format. We assume, that R is constant, while K is subject to non-structural changes.

2.5.1 Matrix Data Structures

Our improved sparse matrix data structure is row-based (Yale or compressed row format [EGSS82]). We store the non-zero entries and the respective column indices of a matrix K in two separate arrays, row by row. Additionally, for every row i, an index to the first non-zero element of the respective row is stored as depicted in Figure 2.17. The set of indices to non-zero entries for every row i is denoted by S_i^K. In the following, we refer to this format as *row-compressed* (RC) matrix format.

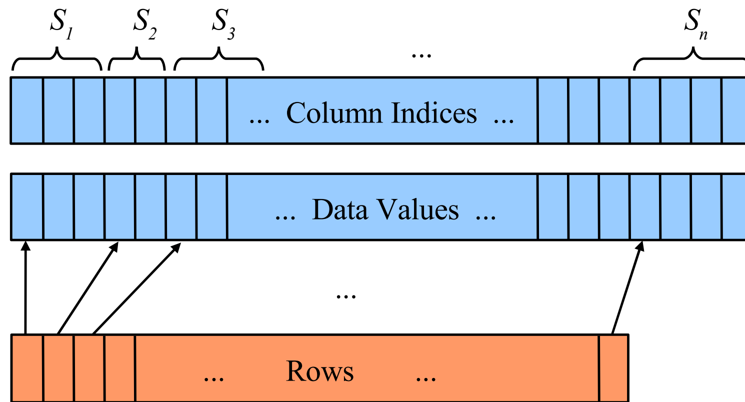


Figure 2.17: *Row-compressed matrix format: A pair of data value and column index is stored for every non-zero entry of the matrix, row by row. For every row, a reference to the respective positions in these arrays is kept.*

Since in the formulation of the 3D elasticity problem as described previously the system matrix consists of 3×3 blocks of generally non-zero elements, we also provide a matrix format that stores entire blocks of data instead of single data values. In this way, we can reduce the memory that is required to store the indices significantly. Analogously, 2×2 blocks are stored in the 2D setting. In the following, we refer to

this format as *block-row-compressed* (BRC) matrix format. For the sake of clarity we assume a *row-compressed* format throughout the following discussion. The extension to the BRC format is straightforward.

2.5.2 Naive Approach

The naive approach to perform a multiplication $R\,K\,R^{\mathrm{T}}$ of two sparse matrices R and K is to use an intermediate representation $F = R\,K^{\mathrm{T}}$, which can then be used to calculate $E = R\,F^{\mathrm{T}}$. Splitting the product in this way is best suited for the RC matrix format, because it requires only the calculation of sparse dot products and therefore allows accessing the data structures in an optimal way. Note that the second matrix in both single products is transposed, and thus these products efficiently access the rows of the data structure of the non-transposed matrix.

2.5.3 1-Step Approach

To avoid the intermediate representation F, let us now have a closer look at the matrix product to be performed. Expanding the matrix product $E = R\,K\,R^{\mathrm{T}}$ yields

$$E_{ij} = \sum_{l \in S_i^R} R_{il} \left(\sum_{k \in S_l^K \cap S_j^R} K_{lk} R_{jk} \right).$$

The outer sum is evaluated only for non-zero entries in the index set S_i^R. The inner sum is optimized by only considering indices in the intersection of the two index sets S_l^K and S_j^R, as in all other cases the resulting summand is zero. It is worth noting that now all sums access our data structure row-wise, such that we have optimal cache coherence. To perform an in-place update of the matrix E (assuming the structure of E is already known), only indices $j \in S_i^E$ have to be considered. The pseudo-code for this case is listed in Algorithm 2. To create the matrix structure of E, the respective loop is performed for all indices $j = 0$ to E.numCols -1. An entry in the sparse matrix structure is only created if $E_{ij} \neq 0$. The benefits of this matrix multiplication strategy are the following:

1. No additional memory is required to store an intermediate product as in the naive approach.

2. All matrices are accessed in a row-wise order resulting in optimal memory access patterns.

Algorithm 2 1-step multiplication (in-place)

Require: Matrices K, R, matrix structure of E
Ensure: $E = RKR^{\mathrm{T}}$

```
for i = 0 to E.numRows do
  for j ∈ S_i^E do
    E_ij = 0;
    for l ∈ S_i^R do
      double sum = 0;
      for k ∈ S_l^K ∩ S_j^R do
        sum = sum + K_lk · R_jk;
      end for
      E_ij = E_ij + sum · R_il;
    end for
  end for
end for
```

However, due to the avoidance of additional memory the inner sum might be calculated several times (the same index j and l can occur for different indices i). Note that this effect is only noticeable if an in-place update is performed. For the symbolic processing, where the structure of the destination matrix has to be determined, the avoidance of an intermediate representation performs fastest in all of our tests. However, this observation might only be true if the matrix R is very sparse as in the case of geometric transfer operators.

In the next section, we will describe how to build an acceleration structure for *in-place* matrix multiplication. Although it comes at the expense of additional memory requirements, an acceleration of up to a factor of 15 is achieved compared to the naive approach described in Section 2.5.2.

2.5.4 1-Step Stream Acceleration

Random sparse matrix-matrix products are well-known to be memory bound rather than compute bound. The 1-step approach as described is mainly limited by two observations. First, the whole (ordered) sets S_l^K and S_j^R in Algorithm 2 have to be processed to account for their intersection $S_l^K \cap S_j^R$, whereas this intersection is typically very small or even empty. Second, the indices l and j themselves are determined by processing sparse index sets. Therefore, accessing these sets S_l^K and S_j^R produces scattered read operations, which can most likely not be served from the cache.

To address the first issue, we have developed a novel acceleration data structure that stores the intersection of the index sets S_l^K and S_j^R for all indices l and j. To address

the second issue, we construct a stream that is aligned with the matrix K and thus avoid scattered memory read operations to access the pre-computed intersections $S_l^K \cap S_j^R$. Furthermore, because the left and right matrices of the product are the same (except for transposition) and do not change over time, their contributions to the product can be encoded into the stream to avoid scattered memory read operations to access the data values of R. Summarizing, we build a stream that encodes data values of R along with instructions how these values are multiplied with respective non-zero entries of the matrix K as well as their destination indices. These indices are used to scatter the appropriate fractions into the destination matrix E. In this way, only the final write operation accesses the memory randomly. Due to the fact that the destination matrix E is smaller in size than the source matrix, memory access operations are optimized compared to the setting where we stream over entries of E while randomly accessing values of K.

Stream Design

The acceleration data structure is aligned with the sparse matrix data structure of K, and it is constructed from two streams: A *control stream* contains control flags and a *data stream* contains copied data values of R and respective indices to E. A single byte of the control stream is interpreted as follows: The sign flag indicates whether the next non-zero entry of the matrix K should be fetched or the previous entry of K is further used. The remaining seven bits indicate the number of data value/index pairs from the data stream that should be processed. Note that at most 127 pairs can be encoded in one single byte. If a matrix entry from K is scattered more often into the result matrix, an additional control byte has to be used with the sign flag set to false.

Stream Construction

The stream construction can be performed analogously to a 1-step multiplication as described in Algorithm 2. However, this approach performs the operations in the wrong memory layout, since the outer loops process the destination matrix E rather than the matrix K. Therefore, we change the ordering of the loops. The outer loops running over all entries of the destination matrix now becomes the innermost loop, yielding outer loops over all elements of the matrix K (using the indices l and k). Then, for each entry of the matrix K, all products $R_{il} \cdot R_{jk}$ and indices i, j into the destination matrix E are determined and can be directly encoded into the data stream. Algorithm 3 lists the respective pseudo code for the stream construction phase. E.getIndex(i, j)

Algorithm 3 Stream construction (in-place)

Require: Matrices K, R^{T}, structure of matrix E
Ensure: $E = RKR^{\mathrm{T}}$

```
for i = 1 to E.numRows do
  for j ∈ S_i^E do
    E_ij = 0;
  end for
end for
for l = 0 to K.numRows do
  for k ∈ S_l^K do
    for i ∈ S_l^{R^T} do
      for j ∈ S_i^E ∩ S_k^{R^T} do
        E_ij = E_ij + K_lk · R^T_li · R^T_kj;
        stream.push(R^T_li · R^T_kj, E.getIndex(i, j));
      end for
    end for
    stream.setNext();
  end for
end for
```

calculates the index of the element in the linearized data array of E. This index is used to quickly access the respective element in the stream processing stage. The stream's push() operation stores the passed value and index into the data stream and increments the number of pairs stored in the last control byte. If the maximum number of 127 is exceeded, a new control byte with a next-entry flag set to false is appended to the control stream. The stream's setNext() operation creates a new control byte with the next-entry flag set to true, thus advancing to the next non-zero element of K.

Stream Processing

The processing of the stream to update the destination matrix E is performed as follows. Initially, l and k are the row and column indices of the first non-zero entry of K.

1. If a next-entry flag is encountered, the index k is advanced to the next non-zero entry in row l. If no such entry is available, the row index l is incremented to the next non-empty row, and k is set to the respective first non-zero column index. The value K_{lk} is stored in a temporary register t. From the control byte, the number p of weight/index pairs that have to be processed next are determined.

2. The following steps are performed p times:
   A data value w and index value i are read from the data stream. The product $w \cdot t$

is accumulated at the position $E(i)$, where $E(i)$ addresses the i-th position in the linearized representation of E.

3. Steps (1.) and (2.) are repeated until the entire stream has been processed (all non-zero entries of K are encountered).

Stream Optimization

The stream can be optimized with respect to the data values w stored in the stream. As in some settings, e.g., in cases of nested hierarchies, it is likely that the same value w is repeated several times, we can save memory by storing w only once together with a set of destination indices. Therefore, the data value/index pairs are sorted with respect to their values w after all pairs belonging to a single entry of K have been generated. Finally, the control stream needs to be adjusted to store for each data value w the number of destination indices to be considered.

2.5.5 Symmetry Optimization

So far we have not considered any symmetry of the matrix K. If the matrix K is symmetric, the 1-step algorithm and the stream acceleration can be performed nearly twice as fast. This is due to the fact, that only the upper triangular matrix of E has to be computed, and the lower triangular part can be determined from the respective mirrored entries. We do not introduce a symmetric row-compressed format, as in this case matrix-vector products cannot be processed at full performance rates due to the improper memory access patterns. A symmetric row-compressed format only stores the upper triangular matrix of K. On average, a single row-vector product then can only access half of the data values of K efficiently, while the other half of the values have to be fetched from different rows (see Figure 2.18).

For that reason, we do not change the matrix format. Instead, the lower triangular matrix is determined from the upper triangular part. If the block-row-compressed matrix format is used, this step can be performed efficiently, as 3×3 blocks can be copied at once. For the pure row-compressed format, this symmetry optimization is not efficient since single data values have to be copied.

2.5.6 Parallelization

The 1-step stream acceleration algorithm can efficiently be parallelized by distributing the data and control streams as well as the destination matrix E to N compute nodes.

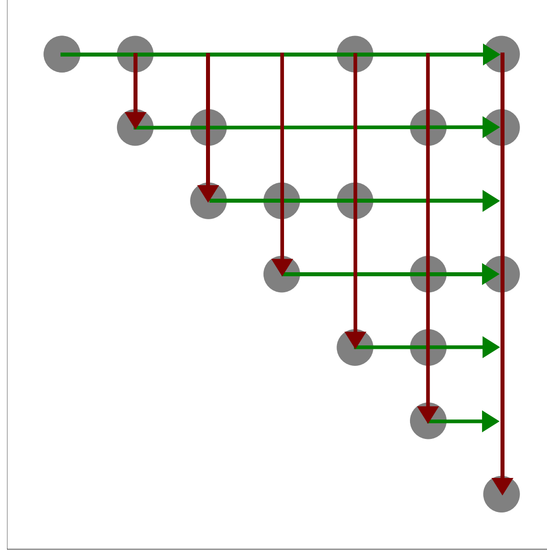


Figure 2.18: *Matrix-vector products using a symmetric sparse matrix format: On average, a row-vector product can only access half of the data values of K memory-efficiently (green), while the other half of the values have to be fetched from different rows (red).*

Each node k only has to store a part of the stream, together with a copy of the result matrix E^k. For the purpose of clarity, we assume that the matrix K is duplicated on each node, although only the non-zero elements corresponding to the respective parts of the stream are required. The stream can be split at the next-entry flags. Additionally, an offset into the matrix K has to be stored at each node such that the part of the stream kept at this node can be aligned with the matrix K. In general, splitting the stream into disjoint parts for which the same number of operations have to be performed requires to count not only the number of non-zero entries of K but also the number of write operations (addresses into the matrix E^k). Only in case of regular meshes, i.e., constant valences, it is sufficient to ensure that each node processes the same number of entries of K.

Finally, the matrices E^k have to be gathered and accumulated to calculate the result $E = \sum_{k=1}^{N} E^k$. This can be achieved most efficiently in a log step reduce operation by combining each two matrices at every second node and proceeding recursively with the results. Since all matrices E^k have the same structure, this summation can be performed in place without the need to newly build sparse matrix data structures.

2.5.7 Performance Measurement

In this section, we analyze the performance and memory requirements of the presented sparse matrix algorithms. In Section 2.7 we will give thorough timings statistics for the overall simulation. All timings in this section are measured on an Intel Core™ 2 Duo 2.4 GHz equipped with 2 GB RAM but only one CPU core is employed in our examples.

In the following, we employ the 1-step stream acceleration approach to compute the RKR^{T} matrix products deduced from the Galerkin property on each level of the multigrid hierarchy. We distinguish between nested and non-nested hierarchies as the latter are more expensive to be updated due to the higher fill rate of both restriction and interpolation matrices.

Table 2.3 shows the timings for building the multigrid hierarchy for various models using nested meshes. We give timings for the algorithms used in the pre-processing stage (including the generation of sparse matrix data structures) and the respective *in-place* variants that can be applied at runtime once the structures of the matrices are already known. For each example we give timings for both the row-compressed (RC) and the block-row-compressed (BRC) matrix format. As the latter severely benefits from the symmetry optimization described in Section 2.5.5, the timings for the symmetric algorithms are given in parentheses if applicable. Columns four and five show the time required by the naive implementation, columns six and seven list the timings for the 1-step algorithm. Finally, the last three columns give the initialization time,

model	dimension / fillratio	sparse format	naive	naive (inplace)	1-step	1-step (inplace)	stream (init)	stream (inplace)	stream (mem)
bridge3k	2.46k	RC	406	25	168	24	31	1	1.8 MB
	1.4%	BRC	-	-	131	22	27	2	2.1 MB
					(61)	(13)	(26)	(1)	(1.7 MB)
bridge24k	15.7k	RC	19.0k	233	4690	204	247	10	13 MB
	0.25%	BRC	-	-	3673	180	233	13	16 MB
					(1720)	(103)	(207)	(12)	(13 MB)

Table 2.3: *Timing statistics in [ms] for construction and update of various nested multigrid hierarchies using various sparse matrix formats and algorithms. The numbers in parentheses give the timings for the symmetric variants described in Section 2.5.5. Note that the stream initialization requires the structure of the destination matrix to be known, thus the 1-step algorithm has to be performed before.*

model	dimension / fillratio	sparse format	naive	naive (inplace)	1-step	1-step (inplace)	stream (init)	stream (inplace)	stream (mem)
liver3k	2.52k	RC	720	163	446	219	193	7	15 MB
	1.4%	BRC	-	-	379	212	180	7	15 MB
					(177)	(102)	(131)	(5)	(9 MB)
bunny11k	9.00k	RC	8258	732	4770	906	677	25	46 MB
	0.39%	BRC	-	-	4098	997	566	26	46 MB
					(2161)	(514)	(424)	(17)	(28 MB)
horse50k	36.7k	RC	212k	3347	144k	5802	6311	180	300 MB
	0.10%	BRC	-	-	123k	6654	4198	167	300 MB
					(67.8k)	(3646)	(2664)	(111)	(179 MB)

Table 2.4: *Timing statistics in [ms] for construction and update of various non-nested multigrid hierarchies using various sparse matrix formats and algorithms. The numbers in parentheses give the timings for the symmetric variants described in Section 2.5.5. Note that the stream initialization requires the structure of the destination matrix to be known, thus the 1-step algorithm has to performed before.*

update time, and required memory for the 1-step stream accelerated algorithm. Note that both initialization and update are *in-place* variants. Thus, the 1-step algorithm has to be used initially to determine the structure of the result matrix. Table 2.4 gives the same information but now the models use a non-nested hierarchy. From the given performance measurements the following results can be concluded:

1. The block-row-compressed matrix format is preferable as it allows for efficient symmetry optimizations at the same time avoiding the previously mentioned drawbacks of a symmetric matrix format.

2. The 1-step algorithm shortens pre-processing times significantly. It generates the structure of the matrices 2–4 times faster than the naive approach. This is both true for the RC and BRC sparse matrix formats[9].

3. The *in-place* variant of the 1-step algorithm can only outperform the naive approach if nested hierarchies are applied. In case of non-nested hierarchies, the naive approach computes the results faster than the 1-step algorithm. However, the naive approach requires additional memory (the temporary matrix F, which

[9]Note that the naive algorithm could benefit from the symmetry optimizations in the same way as the 1-step algorithm if the BRC format is applied.

has to be constructed in a pre-process with the naive approach).

4. The stream-accelerated 1-step algorithm for the *in-place* update outperforms the naive approach by a factor of 10–30. It comes at the expense of additional memory but the data can be efficiently streamed through the CPU. The performance benefit clearly compensates for the additional memory requirements.

Due to these results, we employ the 1-step algorithm in the pre-processing stage of the deformable object's simulation to determine the structure of the matrices. In cases of time-dependent matrices, the 1-step stream acceleration is performed to quickly update the data values of the respective matrix hierarchy. In particular, this algorithm is used in both the corotated and the non-linear strain simulations in every time step.

It is worth noting that the proposed data structures and algorithms for computing sparse matrix-matrix products can be used in many other applications, too. The streaming approach can be easily modified to compute sparse matrix-matrix products in which one of the matrices is constant. Since the approach can easily be parallelized on multi-core architectures or multiple compute nodes, it can also be employed to handle large matrices.

2.6 Mass-Spring Systems

In a range of applications, especially in gaming and entertainment, simulation of deformable objects is only required to appear plausible. In this case, simpler physical models, such as mass-spring systems, might lead to visually acceptable results and achieve a significant performance gain at the same time. However, these models introduce inherent drawbacks compared to the "real" physical simulation described before. In particular, heterogeneous or stiff materials are hard to be handled due to the underlying explicit time integration scheme. Therefore, the application of this model is restricted to the Courant condition.

To visualize system dynamics, the geometric representation of the system has to be modified according to the computed motion. If the simulation is carried out on the CPU, the displaced geometry has to be sent to the GPU in every animation frame for rendering purposes, thus decreasing performance. Therefore, the goal of this research project was to evaluate how mass-spring systems can be implemented on the GPU most efficiently, at the same time avoiding any transfer over the graphics bus for the rendering of the deformed object.

In the following, we present and analyze different implementations of a mechanical system on recent GPUs. Although we focus on mass-spring models, the concepts we propose can also be employed in other applications. In many applications, retrieval and evaluation of adjacent elements' states is an intrinsic mechanism despite different rules to update each part of the system.

In particular, we have implemented a mass-spring system based on regular triangular mesh structures. Edges are treated as springs connecting pairs of mass points. Under the influence of external forces, e.g., forces exerted by user interaction, gravity, or collision, the object deforms into a configuration where the external forces are compensated by opposing internal forces. In the most basic form, only the springs themselves apply forces, seeking to preserve their rest length when compressed or stretched.

Furthermore, we demonstrate a mass-spring system based on irregular tetrahedral grids. Here, volume preservation as proposed by Lee et al. [LTW95] was included. The proposed implementation is distinct from previous approaches in that physics-based simulation and rendering of deformable bodies is performed entirely on programmable graphics hardware. For example, forces applied to a vertex are computed and accumulated in parallel fragment units. Additionally, displaced vertex coordinates can be directly rendered without any read-back to CPU memory. In this way, a significant speed up can be achieved both for the numerical simulation as well as for the rendering. Moreover, it is not only possible to display the surface of the body but also interior properties, such as forces, because the entire body resides in GPU memory.

2.6.1 Theory

Starting with a volumetric body or surface representation, we imagine the mass of the body or surface to be condensed at discrete vertices, which are connected to each other via springs. Doing so, we get a representation based on single mass-points, which are generally connected in an irregular way. Springs can rotate arbitrarily, and the forces they exert on connected points are obtained from Hooke's law

$$F_{ij} = F_{ij}(x_i, x_j) = D_{ij} \frac{\|l_{ij}\| - \|l_{ij}^0\|}{\|l_{ij}\|} \cdot l_{ij}. \tag{2.16}$$

Here, $D_{ij} > 0$ describes the stiffness of the spring connecting points x_i and x_j, and $l_{ij} = x_j - x_i$ is the distance between these points. The rest length of the spring in its initial configuration is denoted by $\|l_{ij}^0\|$. The force F_{ij} is acting on x_i. For symmetry reasons it is essential that $F_{ij} = -F_{ji}$. For every mass point, forces exerted by all

connected springs have to be accumulated. These forces should balance the external forces F_i^{ext} acting on a single mass point. As external forces are exerted continually, the balance between internal and external forces has to be achieved dynamically.

In the current implementation, positions of mass points x_i are updated with respect to their velocity and acceleration using the Lagrangian law of motion

$$m_i \, \ddot{x}_i + c_i \, \dot{x}_i - \sum_{j \in \Gamma_i} F_{ij}(x_i, x_j) = F_i^{ext} \tag{2.17}$$

with m_i being the mass and c_i the damping constant. Γ_i denotes the 1-neighborhood of point x_i. As forces F_{ij} depend non-linearly on the positions of all mass points, the equation of motion (2.17) yields in general a non-linear system of equations. To avoid this, we restrict ourselves to explicit time integration. Given a time step dt, the new position of each point is calculated using the Verlet integration. As this scheme does not require point velocities to be explicitly calculated or stored, the current velocity is always consistent with the current point position. New point positions x_i can then be computed as

$$x_i^{t+dt} = \frac{F_i^{tot}}{m_i} dt^2 + 2\, x_i^t - x_i^{t-dt},$$

where the total force F_i^{tot} is computed as

$$F_i^{tot} = F_i^{ext} + \sum_{j \in \Gamma_i} F_{ij}(x_i^t, x_j^t) - c_i \frac{x_i^t - x_i^{t-dt}}{dt}.$$

As the force calculation is solely based on point positions at the current time step, forces F_i^{tot} as well as updated point positions can be computed in parallel. Since the position update in general affects all springs, external and internal forces are no longer in balance. This results in a dynamic behavior of the system. To guarantee convergence, a reasonably small integration time step satisfying the Courant condition[10] [CFL67] has to be chosen.

2.6.2 Volume Preservation

We want to mention an addition to conventional mass-spring systems to enforce volume preservation. It is well known that mass-spring simulations cannot preserve volume as they are not based on a volumetric simulation. Therefore, stable situations can (and will) occur where parts of the model are inverted. Since the springs are no longer

[10] See also Section 2.3.7

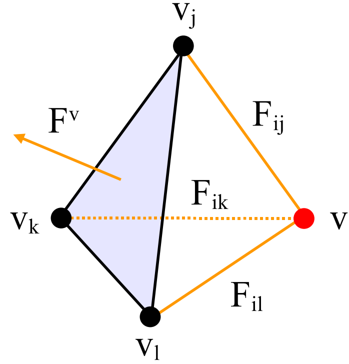


Figure 2.19: *Illustration of volume preserving forces. For the red vertex v_i, forces are calculated with respect to the stretching of the springs ij, ik, il. The volume force F^v at vertex v_i in the direction of the opposite face's normal ensures that each element resists changes of its initial volume.*

stretched in this case, the model is in a totally stable configuration.

To avoid such an unwanted behavior, we introduce artificial volume preserving forces as introduced by Lee et al. [LTW95]. We regard the volumetric body as an irregular tetrahedral mesh with vertices used as mass points and edges interpreted as springs. Then, based on its initial configuration, we can establish a rest length for every spring and a rest volume for every tetrahedron. For a single mass point x, we then collect additional forces for every adjacent tetrahedron e based on the actual volume v_e compared to its rest volume v_e^0 using an artificial volume stiffness parameter D_v:

$$F_e^v = D_v(v_e - v_e^0)\, n_e.$$

Here, n_e is the (outfacing) unit-length normal of the opposite tetrahedral face. As points might move, the normal has to be recalculated in every time step. Using this technique, we avoid inverted tetrahedral elements since they would result in negative volumes. The resulting force F_e^v strongly pushes the tetrahedron e towards its initial state. All internal forces acting on a single vertex are illustrated in Figure 2.19.

2.6.3 GPU Architecture and Functionality

Early generations of graphics processors were solely optimized for the rendering of lit, shaded and textured triangles. The rendering pipeline was implemented by a set of special-purpose but fixed-function engines, prohibiting the use of such chips in non-rendering applications. Nowadays, this design is abandoned in favor of programmable

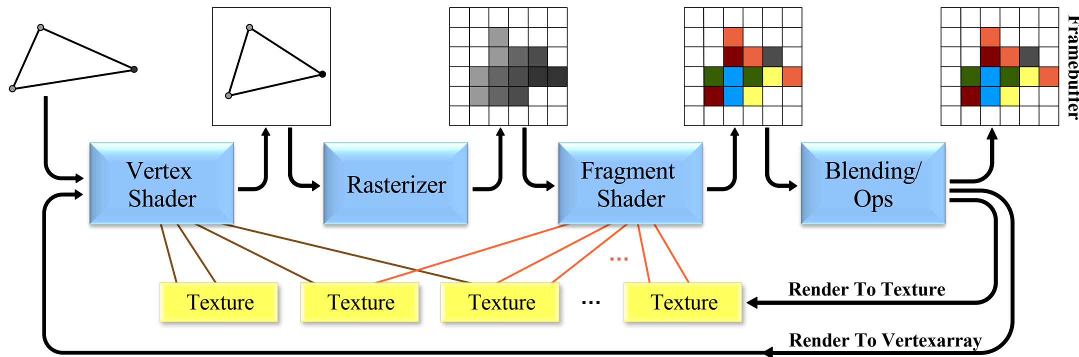


Figure 2.20: *Stages of the programmable graphics pipeline.*

function pipelines that can be accessed via high-level shading languages integrated into Direct3D or OpenGL [Mic02, MGAK03].

On current GPUs, fully programmable parallel vertex and fragment (or pixel) units are available, which provide powerful instruction sets to perform arithmetic and logical operations. In addition to computational functionality, fragment and vertex units also provide an efficient memory interface to server-side data, i.e., texture maps and frame buffer objects. Not only can application data be encoded into such objects to allow for high performance access on the graphics chip, but rendering results can also be written to such objects, thus providing an efficient means for the communication between successive rendering passes. Figure 2.20 gives an overview of the rendering pipeline as it is implemented on last generation GPUs.

A GPU can be seen as a parallel SIMD[11] stream processor. The input data are streams of vertices, which are then processed by the *programmable* vertex stage. In the rasterization stage, the stream of vertices (defining a set of primitives, e.g., triangles, lines, points) is converted into a stream of fragments, where each fragment covers a pixel of every primitive. These fragments are processed in the highly parallel *programmable* fragment (pixel) stage. Before the fragments are written into the frame buffer, the blending stage combines the incoming fragments with the fragments already stored in the frame buffer.

In recent years, a popular direction of research is leading towards the implementation of general techniques of numerical computations on graphics hardware [HBSL03, BFGS03, KW03b, Krü06]. This research is mainly driven by the fact that the GPU's floating point performance and memory bandwidth outperforms current CPUs by far.

[11] Single Instruction Multiple Data

Additionally, the GPUs computational power is growing 2–3 times faster than Moore's Law. The results of these efforts have shown that for compute-bound applications as well as for bandwidth-bound applications, the GPU has the potential to outperform software solutions. However, this statement is valid only for such algorithms that can be compiled to a stream program, which then can be processed by SIMD kernels as provided on recent GPUs. Moreover, the local memory on GPUs is limited; thus the algorithms are required to be adapted carefully to meet this restriction.

2.6.4 GPU Implementation

In the following, we discuss the implementation of mass-spring systems on the graphics hardware architecture introduced in Section 2.6.3. Typically, the implementation requires the following steps:

1. Calculation and accumulation of spring forces at mass points
2. Time integration of mass points
3. Update of point positions

As the first step requires adjacent points to be accessed, a GPU data structure able to efficiently perform this operation needs to be developed. In the following we will investigate the use of two different data structures in this particular scenario—a *point-centric* and an *edge-centric* data structure. Depending on which data structure is used, mass-spring simulation is implemented as shown in Algorithm 4 and 5.

The major difference is the way spring forces are calculated. In the point-centric approach, for every mass point adjacency information is gathered. In the edge-centric approach, every edge computes its spring force only once and scatters this force to the mass points it connects.

Point-Centric Approach (PCA)

In the point-centric approach, a surface point needs to maintain its current and last position for time integration, its mass, and references to all adjacent points including spring stiffness and rest length. Consequently, the memory requirement for each vertex is not constant and depends on its valence (the number of incident edges).

On the GPU, per-point attributes and references are stored in equally sized 2D texture maps. We store references into a 2D texture in a single float component, and we use shader arithmetic to decode the appropriate 2D texture coordinates. A stream of

Algorithm 4 Point-centric mass-spring system

for all mass points i **do**
  initialize total force F_i
  // Gather force
  for all neighboring mass points j **do**
    calculate spring force F_{ij}
    add F_{ij} to total force F_i
  end for
  update vertex position x_i
end for

as many fragments as there are points is generated by rendering a view plane aligned quadrilateral that covers exactly this number of pixels. In the fragment units the point-centric algorithm is carried out by fetching attributes of adjacent points from the respective texture maps. Forces are then calculated and used to update the position as described above. These positions are written into an additional render target, which becomes the container of point coordinates in the next simulation pass.

Using the proposed data structure, two different realizations of mass-spring simulation on the GPU are possible. First, if all vertices have the same valence, all computations necessary to update mass point positions can be performed in one rendering pass. The execution has to be broken into multiple passes only if the valence exceeds the number of available texture units. Second, if the valence is not constant, the computation has to be split into multiple rendering passes. To avoid processing of points that have no further neighbor, a particular texture layout can be employed. We introduce this layout in the next Section 2.6.5 when discussing irregular volumetric meshes.

The point-centric approach comes at the expense of calculating each spring force twice as every spring is incident to two mass points. Moreover, the data structure becomes very inefficient for large, irregular valences. Since as many textures as the maximal valence of the mesh are required, the number of memory access operations as well as the amount of memory to be kept can become the bottleneck of the simulation.

Edge-Centric Approach (ECA)

An edge-centric data structure overcomes the aforementioned drawbacks of a point-centric approach. For each edge, references to both incident points as well as the spring stiffness and rest length are stored in an appropriately sized *edge texture*. In a first rendering pass, spring forces are calculated and rendered into a texture target—the *force texture*. As there are three times more edges than points in a triangulation, this

Algorithm 5 Edge-centric mass-spring system

for all mass points i **do**
  initialize total force F_i
end for
for all springs ij **do**
  calculate spring force F_{ij}
  // Scatter force to incident mass points
  add F_{ij} to total force F_i of left mass point
  add F_{ij} to total force F_j of right mass point
end for
for all mass points i **do**
  update vertex position x_i
end for

texture is different in size than the texture keeping point coordinates— the *point texture*. The problem now becomes to scatter the computed forces to the respective points, and for every point to accumulate the received contributions. Such an operation is not supported on recent GPUs, and gathering the forces using a point-centric data structure results in the same problems as described above.

To enable GPU scattering, we harness the power of vertex processing. For every edge, a point primitive is rendered twice into a render target that has the same size as the point texture. First, primitives are rendered to the respective position of the left mass point of each edge. In the second pass, the target position is the right mass point in this texture. As both target positions are already stored in the edge texture, this structure can directly be rendered as server-side vertex array. A vertex shader program decodes the stored references into appropriate point coordinates to determine the respective raster position in the render target. In every pass, the force texture is used as additional vertex attribute array. Forces are negated in the fragment shader of the second pass, before they are accumulated in the current render target. In this way, multiple points are rendered into the same entry of a point-centric render target, which finally stores the force per mass point.

Independent of the points' valences in the mesh, the edge-centric approach only requires four rendering passes. The first three passes operate as just described, and in the fourth pass the time integration of mass point positions is performed using computed per-point forces. In addition, forces are only computed once for each edge, reducing the number of arithmetic operations and texture fetches to be performed.

2.6.5 Irregular Volumetric Models

In the following, we restrict the discussion to tetrahedral meshes, and we include the volume preservation described in Section 2.6.2. As the volume preservation requires the information of neighboring tetrahedral elements, storing only edges as described above is not sufficient. To develop an appropriate data structure for this kind of meshes we first have to think about the operations to be performed on this structure.

Considering one tetrahedral element, for each of the four corner points a force vector has to be computed. This vector depends on the elements' edge lengths and volumes as well as the respective rest and stiffness values. For each mesh vertex these forces are finally summed up by looping over all incident tetrahedra.

So-called dependent texture fetches slow down the performance considerably. Dependent texture fetches read from a texture address that has been determined from a texture value that was fetched earlier. Such operations prevent the GPU from issuing all texture fetches in parallel, and it is therefore desirable to have as few fetches depending on each other as possible.

If we investigate the aforementioned approach, it is easy to see that it requires four (internal) rendering passes, each pass performing four dependent texture fetches as the shader program first has to read the indices of the four corner points. This makes a total of 16 dependent fetches (one level of indirection) per tetrahedron and simulation step.

A more efficient data structure is based on the observation that to predict the dynamic behavior of the mass-spring model, every mesh vertex v must calculate the exerting force vector in each frame. This vector is influenced by the one-ring neighborhood around v, i.e., all mesh vertices that are connected to v via a spring. To account for volume preservation, every vertex computes the volume loss or gain of all adjacent tetrahedra and tries to correct this change by an additional displacement. Therefore, access to all tetrahedral elements sharing the center vertex v is required. Since these operations are performed in parallel for every vertex in the mesh, they are perfectly suited for an implementation on data parallel stream architectures such as GPUs.

To optimally exploit the architecture of recent GPUs (including parallel computations and high memory bandwidth), we have integrated volume preservation into the point-centric data structure. First, a 2D vertex texture is created, which stores mesh vertices in the RGB color components. We then construct a sequence of equally sized 2D textures, each of which encodes one of the tetrahedral elements adjacent to the respective vertex in the vertex texture. Tetrahedra are encoded by three references into the vertex texture and a stiffness value as well as three spring rest lengths and the rest volume of the tetrahedron. These values are stored in a pair of RGBA textures.

Algorithm 6 Volumetric mass-spring system

for all points x_i **do**
  for all tetrahedra e incident on $x_i = x_0^e$ **do**
    // via three texture fetches
    get center vertex coordinate x_0^e
    get corner indices i_1, i_2, i_3, get element stiffness D_e
    get rest spring lengths l_1^0, l_2^0, l_3^0, get rest volume v_e^0,
    // via three dependent fetches
    get coordinates of x_1^e, x_2^e, x_3^e through i_1, i_2, i_3
    calculate force at x_0^e
    add to total force F_0^e
  end for
end for

On a per-vertex basis, we keep track of forces due to compression or stretching by following the references to connected mass points. In a fragment shader, exerted forces are computed for every vertex in parallel, and at every node the resulting forces are accumulated. If the force calculation is not executed per tetrahedron but per mass point, only three dependent fetches are needed per point, making a total of 12 such operations per tetrahedron and simulation step. Note that the index of one of the points (the current center vertex) is already known and does not need to be fetched. In pseudo code notation the program reads as in Algorithm 6.

Valence Textures

The presented data structure has the drawback that the texture sequence must be large enough to keep a number of references equal to the maximum valence of any of the mesh vertices. In typical meshes, however, we see a rather inhomogeneous distribution of valences. For instance, in the meshes we have used to demonstrate our approach valences ranging from 6 to 32 can be found. To avoid the memory overhead that is introduced by storing for every vertex as many neighbors as the maximum valence in the mesh, we construct different sized textures.

Initially, mesh vertices are sorted with respect to their valence. Then, we recursively generate a sequence of 2D textures at ever decreasing size, which store the topology and additional parameters. We build a 2D texture large enough to keep all vertices, and we construct a list of V_{min} equally sized textures, where V_{min} is the minimum valence of all vertices. These textures are filled with respective references into the vertex texture. Note that the layout of values in these textures is with respect to decreasing valence from top/left to bottom/right (see Figure 2.21). We then build the remaining $V_{max} - V_{min}$

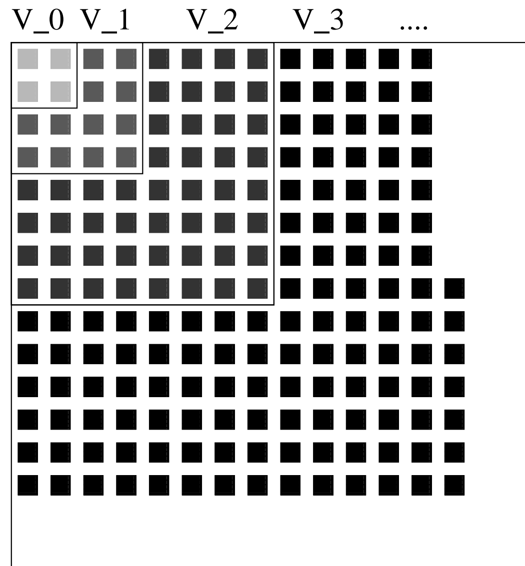


Figure 2.21: *Sequence of valence textures. The smallest square contains all points that have highest valence. This results in valences in the range V_0. The next bigger power-of-two square contains all vertices from the smallest square, and it is filled with vertices with remaining highest valences. Therefore, the range of valence of the new vertices can be determined as V_1. This can be repeated until no vertices are left, resulting in a sequence of power-of-two textures.*

textures of reduced size by removing all vertices with a valence equal to V_{min} from the texture, and we continue the recursive process with this texture. This procedure is repeated until all vertices have been discarded.

In each animation frame, a set of quadrilaterals covering as many fragments as there are values in the corresponding texture from the list is rendered, and force contributions are computed for each remaining vertex. Already accumulated results are rendered into a texture render target, which can be accessed in upcoming passes to retrieve summed values. To account for the differently sized render targets, the viewport is adjusted in each rendering pass.

Element-Centric Approach

To include volume preservation into the edge-centric approach as described above, an appropriately sized tetrahedra texture has to be stored. Therefore, we call the resulting method *element-centric*. Besides updating spring forces, volume repulsion forces have to be calculated for every tetrahedron as shown in Algorithm 7.

In a first rendering pass, for every tetrahedron the geometry information is gathered using four dependent texture fetches. Then, the respective spring (edge) forces as well

Algorithm 7 Volumetric element-centric mass-spring system

for all mass points i **do**
  initialize total force F_i
end for
for all tetrahedra e **do**
  // via three texture fetches
  get corner indices $i_0, \dots, i_3$
  get rest length for each edge $l^0_{01}, l^0_{02}, l^0_{03}, l^0_{12}, l^0_{13}, l^0_{23}$
  get element stiffness D_e, get rest volume v^0_e
  // via four dependent texture fetches
  get vertex coordinates $x^e_0, \dots, x^e_3$ through $i_0, \dots, i_3$
  // Force calculation
  calculate spring force for every edge $F_{01}, F_{02}, F_{03}, F_{12}, F_{13}, F_{23}$
  calculate volume force for every vertex $F^v_0, \dots, F^v_3$
  // Scatter force to incident mass points
  add $F^v_0 + F_{01} + F_{02} + F_{03}$ to total force F^e_0 of 0th vertex of e
  add $F^v_1 + F_{12} + F_{13} - F_{01}$ to total force F^e_1 of 1st vertex of e
  add $F^v_2 + F_{23} - F_{20} - F_{21}$ to total force F^e_2 of 2nd vertex of e
  add $F^v_3 - F_{03} - F_{13} - F_{23}$ to total force F^e_3 of 3rd vertex of e
end for
for all mass points i **do**
  update vertex position x_i
end for

as volume forces can be determined for each corner vertex. These values are written to four force textures of the same size employing the multiple render targets extensions, e.g., OpenGL `ARB_draw_buffers` extension. Then, four tetrahedral scattering passes scatter the appropriate forces to the respective position in the vertex texture, where the final forces are accumulated.

2.6.6 Discussion

In the following, we will evaluate and compare the developed GPU data structures. In particular, memory requirements as well as the number of texture fetches and arithmetic operations are compared for point-centric and edge-centric approaches. We further distinguish between texture fetches and dependent texture fetches, with the latter ones being dependent on the result of an earlier texture look up. Such fetches are known to be a potential bottleneck in GPU applications as the pipeline has to be stalled until the result of the first texture fetch is available.

Analysis

In this section, N denotes the maximal valence of a vertex in a triangulation. The number of vertices and edges are denoted n_v and n_e, respectively. For regular meshes, n_v and n_e are related by $N = 2n_e/n_v$. In the general case, N can be significantly larger.

Table 2.5 shows the statistics for the point-centric (PCA) and the edge-centric (ECA) approach. In PCA, for every mass point and every adjacent point the reference to this point, the stiffness of the spring connecting both points, and the spring's rest length are encoded in an RGB texel. In ECA, two references to the points connected by the spring, spring stiffness, and rest length are encoded in one RGBA quadruple. While forces are stored in an RGB texture map, an RGBA texture is used to keep each point position and its mass. Throughout the discussion we do not consider the overhead introduced by the Verlet time integration as it adds the same additional expense to both approaches, i.e., $2n_v$ texture fetches to get the current and the previous point position and about 10 arithmetic operations to perform the integration.

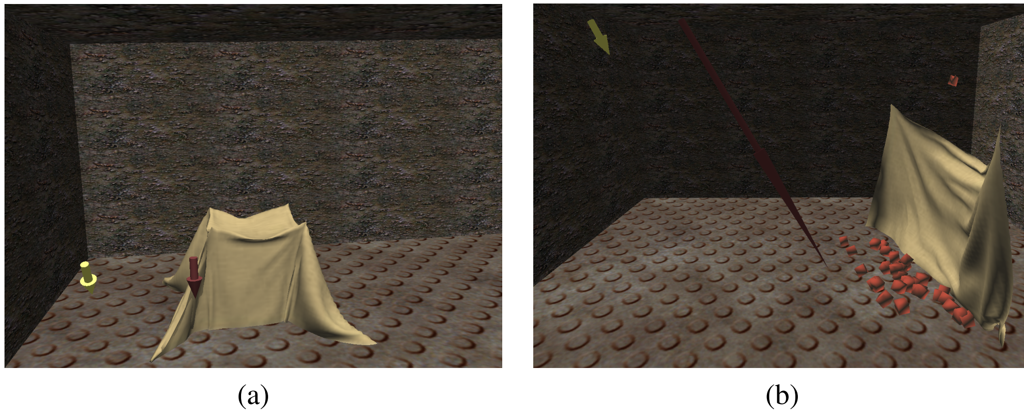
Performance Measurement

On our target architecture, a Pentium4 3 GHz processor equipped with an ATI X800 XT card, we can render about 240 million points per second from the server-side edge texture as described above. Even for the largest mesh we consider in our investigations, consisting of 512^2 vertices, this throughput allows us to perform GPU scattering in ECA about 480 times per second. As will be shown below, this time is justifiable compared to the time required by force calculation and time integration.

For a regular triangular mesh of valence 6 as shown in Figure 2.22 (a), the PCA requires $n_v + n_e$ more texture fetches than ECA and the same number of dependent fetches. In addition, the number of operations to be performed in the fragment units is slightly increased. The memory requirement of ECA, on the other hand, is slightly higher compared to PCA. This is due to the texture needed to store the resulting forces of the first pass. As can be seen in Table 2.6, due to the lower number of arithmetic and memory access operations, even for regular meshes exhibiting rather low valence,

	mem. RGB	mem. RGBA	tex. fetches	dep. tex. fetches	ops
PCA	$N \cdot n_v$	n_v	$n_v + N \cdot n_v$	$N \cdot n_v$	$10\, N \cdot n_v$
ECA	n_e	$n_e + n_v$	n_e	$2\, n_e$	$14\, n_e$

Table 2.5: *Comparison of memory requirement, texture fetches, and arithmetic operations for the point-centric approach (PCA) and the edge-centric approach (ECA).*



(a) (b)

Figure 2.22: *GPU cloth simulation: (a) A cloth patch fixed on 4 points under influence of gravity. (b) Interaction of different objects in an example scenario. The arrows describe the direction of wind forces and additional forces applied to the balls.*

	force calculation	force accumulation	Verlet integration	total
PCA 128^2	0.54 ms		0.20 ms	0.74 ms
ECA 128^2	0.40 ms	0.12 ms	0.20 ms	0.72 ms
PCA 256^2	2.34 ms		0.74 ms	3.08 ms
ECA 256^2	1.76 ms	0.52 ms	0.74 ms	3.02 ms
PCA 512^2	9.82 ms		3.18 ms	13.0 ms
ECA 512^2	7.34 ms	2.08 ms	3.18 ms	12.6 ms

Table 2.6: *Performance comparison (ATI X800 XT) between the point-centric (PCA) and edge-centric (ECA) approach. All timings are measured for a regular mesh with valence 6.*

ECA outperforms PCA in terms of runtime.

For irregular meshes, on the other hand, the benefits of ECA will grow substantially as ECA does not depend on the maximal valence of the mesh. With increasing valence, both memory requirements and texture fetch operations of PCA will increase as well. Moreover, a potentially large number of rendering passes has to be performed. Even if an optimized texture layout is employed to minimize the number of fragments to be processed, this texture cannot be packed densely in general and thus introduces some overhead in the current application. Since each successive rendering pass covers less pixels, the setup costs for a single rendering pass become more and more noticeable. For example, PCA performs 1.6 times slower in total for a 256^2 mesh with valences in the range from 3 to 12.

The most crucial limitation of ECA in the current scenario is with respect to additive blending in the render target that is used to accumulate the force contributions.

As 32 bit floating-point blending was not supported on any GPU when this project was developed, force accumulation was performed inadequately in 8 bit fixed-point precision. In contrast to PCA, where force accumulation is carried out in the fragment shader with 24 bit floating point precision on an ATI Radeon X 800 graphics cards, numerical precision is therefore a problem in ECA. However, very recent graphics cards (e.g., NVIDIA 8800 GTX) overcome this limitation as they allow for full precision floating-point blending. Examples for a cloth simulation environment based on our PCA implementation for regular meshes are shown in Figure 2.22 (a) and (b).

Volumetric Objects

Figure 2.23 shows deformations on volumetric objects that have been performed using the proposed GPU mass-spring system. As our implementation accelerates both simulation and rendering of the deformed bodies, our timings include the entire system. Table 2.7 shows the timings for differently sized models. The peak performance is achieved on the ATI Radeon X800 with about $84,000$ tetrahedral elements. For larger models, simulation performance basically remains the same, but rendering becomes significantly slower due to the increased geometry load.

When comparing our results to those published in [THMG04], we recognize a speedup of about a factor of 20. Our peak-rate is 8.9 million tetrahedra per second (TPS) compared to 310 thousands TPS (including rendering) reported by Teschner et al. Please note that—as it is typical for explicit time integration schemes—rendering does not take place in every simulation frame, but in every fifth step, such that we achieve a visual update rate of about 50 Hz. If the net simulation time excluding rendering is compared, we are still faster by a factor of about 10.

However, the drawback of our method is that it introduces a significant memory overhead. Neighboring tetrahedra are stored for every vertex separately, hence each tetrahedron is stored four times in total. Furthermore, as tetrahedra share common

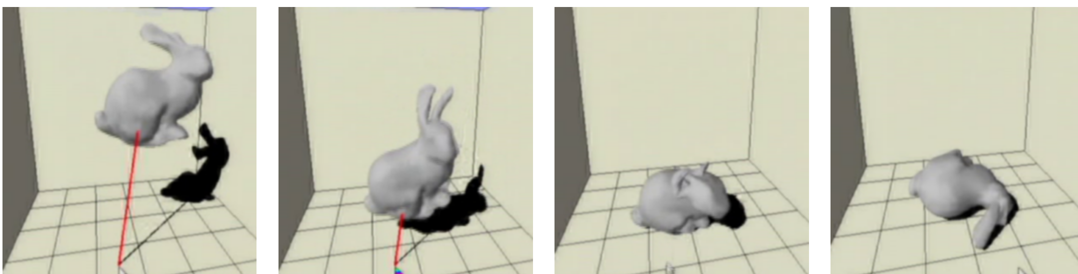


Figure 2.23: *Interactive GPU-based deformations of the bunny model.*

Model	#Tetrahedra	Computation & rendering time [ms]	TPS rating	FPS
Cuboid	5012	2.70	1854440	370
Liver	7536	2.80	2690352	357
Bunny	9804	2.95	3313752	338
Double Bunny	19608	3.26	6019656	307
Large Bunny	84104	8.26	8999128	121

Table 2.7: *GPU simulation performance rates (including rendering) for volumetric meshes on ATI Radeon X800. As the integration time step is fixed to 4ms, a framerate of 250 fps or above denotes real-time simulation.*

edges, spring forces are calculated multiple times, depending on the valence of the adjacent vertices.

Note that due to the restrictions on floating-point blending operations on graphics hardware up to the NVIDIA GeForce 7 Series, the element-centric approach described in Section 2.6.5 has not been implemented. However, since 32 bit floating-point blending is available on NVIDIA GeForce 8800 graphics cards, the element-centric approach becomes very attractive. This approach is independent of the mesh valences, requires less amount of memory and is expected to be faster than the point-centric implementation on current graphics hardware.

Explicit Time Integration

Due to the Courant condition there are hard constraints on the largest time step for which stable simulation can still be achieved. In particular, the stiffer the materials are, the smaller the time steps have to be chosen, resulting in a high numerical workload. Because on current GPUs floating-point precision is limited to 32 bit, and numerical errors thus accumulate rather fast, stability cannot be guaranteed for very stiff materials. In particular, this prohibits the simulation of physical materials, since the elastic modulus of such materials is in the range 10^4–10^{12}. Therefore, GPU mass-spring systems are limited with respect to both the model size and the spring constants. On the other hand, implicit time integration schemes require a system of linear equations to be solved on the GPU. In the case of regular meshes, the resulting sparse matrices can be represented by a few diagonals of non-zero elements, and thus linear algebra operations can be performed efficiently on the GPU [KW03b]. For irregular meshes, the required GPU data structures (for scattered-sparse matrices) are less efficient [KW03b], and thus an optimized CPU approach is likely to outperform an implicit GPU implementation.

Tejada and Ertl [TE05] describe how to solve an implicit mass-spring system on the GPU using a conjugate gradient method for the derived system of linear equations. The basic idea of this method is the reformulation of the Lagrangian equation of motion into a constant and a time-dependent part as described by Baraff et al. [BW98]. At runtime, the matrix of the system of linear equations derived by implicit Euler time integration is updated to take the current mesh deformation into account. However, from the experience we have made with the optimized CPU multigrid solver for the corotated strain formulation, we expect a CPU solution based on our multigrid framework to be considerably faster than the GPU implementation suggested by Tejada. It is worth noting that the multigrid framework can be easily adapted to handle implicit mass-spring systems, too. The matrix structure does not change, but the matrix values are continuously being changed by the update procedure based on the deformed vertex positions. However, the Galerkin update of the multigrid hierarchy can be performed at very high rates using our acceleration structures as has been shown in Section 2.5.

2.7 Results and Validation

2.7.1 Real-Time Multigrid Simulation Framework

In the following, we give several examples that demonstrate the efficiency of the proposed multigrid method. The models used in these examples are shown in Figure 2.24. All experiments were run on an Intel Core™ 2 Duo 6600 2.4 GHz processor equipped with 2 GB RAM. Since the simulation performed in 32 bit floating point precision in some settings—especially for heterogeneous materials—fails to converge, all numerical operations are calculated with double floating-point precision in our framework. Since current GPUs only support 32 bit floating-point arithmetic, a GPU implementation of the finite element framework has not been considered in this work. We now discuss the simulation of three different types of strain—Cauchy strain, corotated Cauchy strain, and Green strain.

Linear Cauchy Strain

As shown in Figure 2.25, the multigrid method scales linearly with the number of elements, and it achieves excellent performance rates even for large models. For the employed example of triangular finite elements in a 2D domain, interactive performance can still be achieved for half a million simulated elements. It is worth noting that this yields a sparse system matrix of dimension $2 \cdot 513^2 \times 2 \cdot 513^2$.

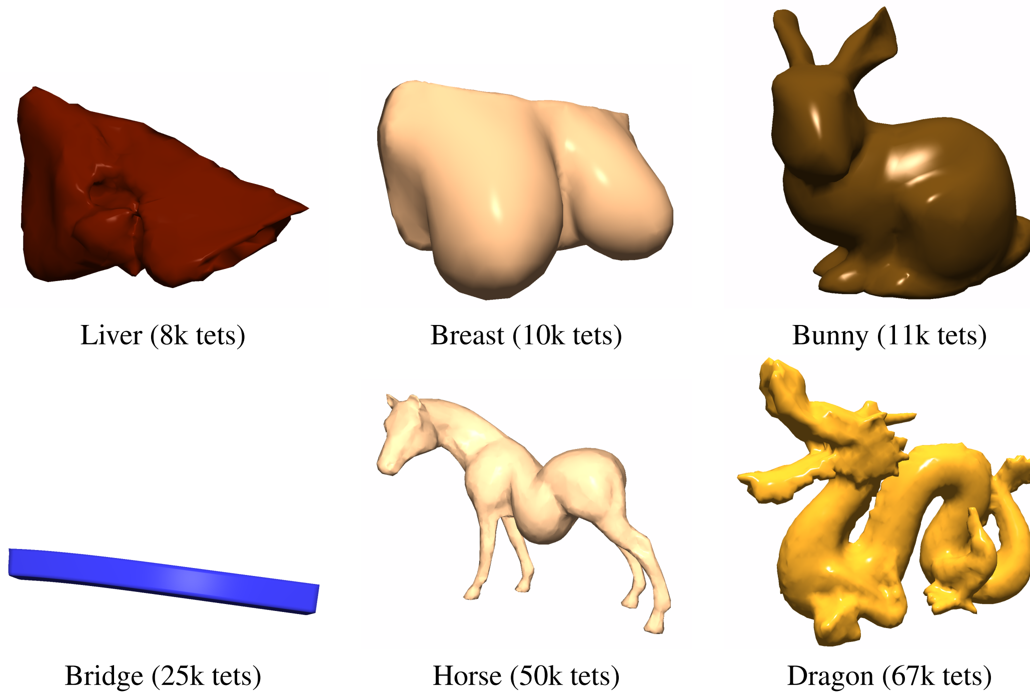


Figure 2.24: *Some tetrahedral models used throughout the discussion are shown. The surfaces of the tetrahedral simulation meshes are rendered.*

We discuss the performance results for 3D simulations using tetrahedral elements in much more detail. In Table 2.8, timings for various models using nested and non-nested hierarchies of different depths are listed. Tetrahedral meshes with up to 200,000 elements can be simulated interactively using the linear Cauchy strain measure.

Particularly in the last example of Table 2.8, where a deeper hierarchy allows the multigrid approach to reach its full potential, the method is considerably faster than implicit approaches utilizing the conjugate gradient method. In contrast to explicit methods, the implicit multigrid solver enables much larger integration time steps and guarantees stability at the same time. Even more importantly, the time step does not depend on the material stiffness. This property enables stable simulations of heterogeneous bodies with an elastic modulus varying from 10^3 N/m^2 to 10^{12} N/m^2.

In Figure 2.26, different stiffness values have been assigned to different parts of a finite tetrahedra mesh, which was constructed from a triangular horse model. In particular, all four legs exhibit very high stiffness while the abdomen is made of soft material. Consequently, the abdomen moves to the ground due to gravity while the legs keep the rest of the horse in shape. Figure 2.27 shows the influence of wind forces and gravity to bars of different density and stiffness. As the force is constant everywhere, softer bars are deformed much more significantly than stiffer ones.

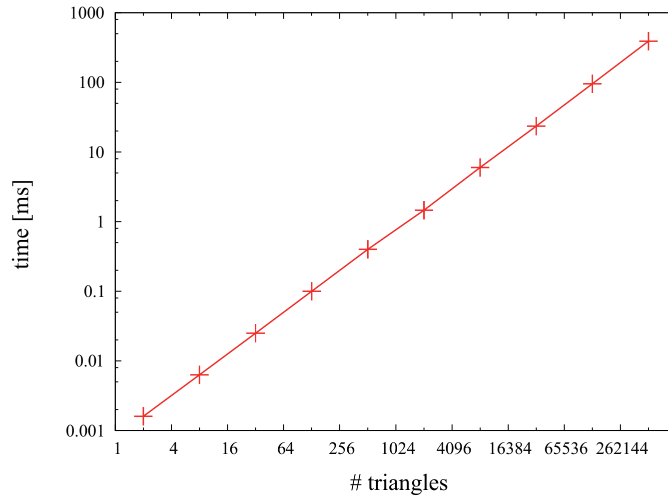


Figure 2.25: *This figure illustrates the linear time complexity of the proposed multigrid solver. For the Cauchy strain simulation of a 2D quad under gravity, ever more triangular finite elements are used by applying a regular subdivision scheme. This leads directly to a nested hierarchy with up to 10 levels. The time for one multigrid V-cycle is measured in milliseconds.*

Model	# Level	# Tet	# Vert	TPS	Time [ms]
Liver	2*	1467	464	1110	0.90
Bridge	3	3072	825	545	1.83
Liver	3*	8078	1915	189	5.29
Breast	2*	10437	2542	174	5.75
Bunny	2*	11206	3019	125	8.00
Bridge	4	24576	5265	76	13.20
Horse	3*	49735	12233	30	33.30
Dragon	4*	67309	16943	17	58.50
Bridge	5	196608	37281	10	98.80

Table 2.8: *Timing results in time steps per second (TPS) for a single V-cycle for different models using the linearized Cauchy strain measure. Since a single simulation thread is used, only one kernel of the dual core CPU is employed. The star $*$ denotes the use of a non-nested grid hierarchy.*

In comparison to the solution proposed by Bro-Nielsen and Cotin [BNC96], our multigrid solver can effectively exploit the sparsity of the problem. Bro-Nielsen and Cotin suggested to use matrix pre-inversion as they observed this method of solution to be fastest at runtime (ignoring the time-consuming pre-processing phase). However, this observation is only true for rather small models, since the inverse matrix is generally not sparse anymore, and thus the CPU's floating point units can be fully exploited. On the other hand, if larger models are applied the dense inverse matrix introduces

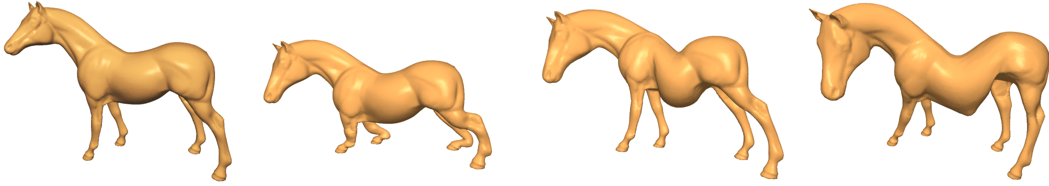


Figure 2.26: *The deformation of a tetrahedral horse model is shown. From left to right: the initial model, the model exhibiting homogeneous and inhomogeneous stiffness under gravity, and the heterogeneous model under additional external forces.*

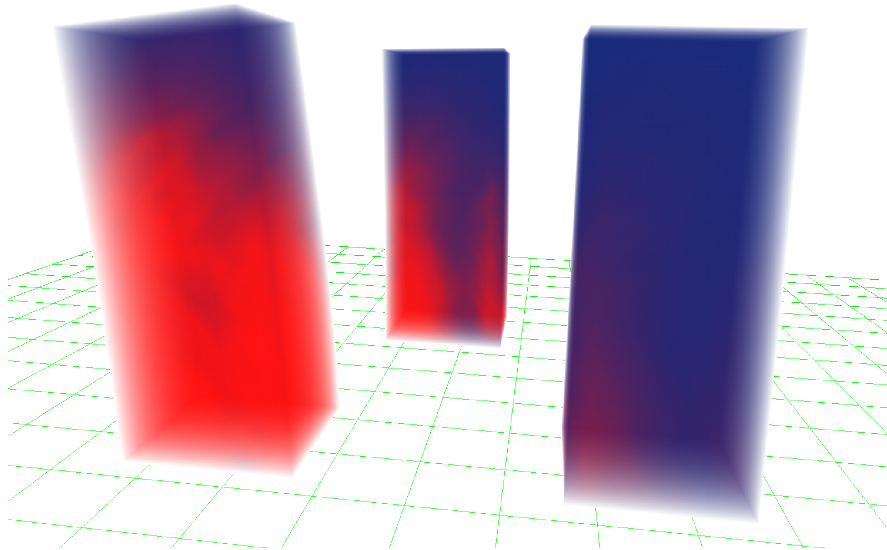


Figure 2.27: *Visualization of the internal states (i.e., von Mises stress, see Section 3.4.5) of three towers exhibiting different stiffness. A constant wind force is applied to all models. Stress values are color-coded ranging from blue (low) to red (high). The volume rendering is achieved with the techniques described in Chapter 3.*

significantly more arithmetic operations compared to iterative approaches exploiting the sparsity of the problem. In particular, if more than approximately 1,000 elements are used, a conjugate gradient method operating on the sparse matrix data structures is favorable to the dense pre-inversion approach on current CPUs.

The comparison to a diagonal pre-conditioned conjugate gradient method (see Figure 2.28 left) shows that the performance gain of our multigrid method increases with the mesh size. This is due to the linear-time complexity of the multigrid approach, which cannot be achieved by the conjugate gradient methods because the number of

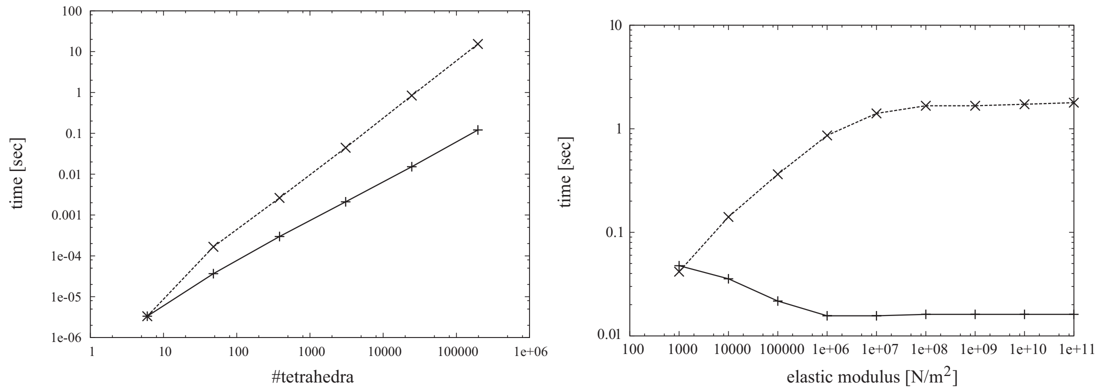


Figure 2.28: *Left: Comparison of the time required until convergence on a double logarithmic scale for the multigrid method (solid line) and the diagonal pre-conditioned conjugate gradient method (dashed line). The former method scales linearly with the number of tetrahedra, while the latter one requires more and more iterations to achieve the same relative error of* 10^{-2} *in the solution. The timings where measured using a cube model that was subsequently refined by the split operation shown in Figure 2.11, and a fixed integration time step of 0.02 sec.*
Right: Performance measurements for a bridge model consisting of 3k tetrahedral elements. For ever stiffer materials, the conjugate gradient method (dashed line) requires more steps to achieve the same relative error of 10^{-2} *as the multigrid method (solid line). The elastic modulus affects the performance of the conjugate gradient method significantly while it does not affect the performance of the multigrid method. Only for extremely soft materials the performance of the multigrid method drops down because forces cause only very local deformations that cannot be solved for on a coarser grid.*

required iterations grows with the matrix dimension. Even more importantly, both the time step and the number of iterations until convergence of the multigrid approach do not depend on the material stiffness as illustrated in Figure 2.28 on the right. Especially for stiff materials, the multigrid method taking advantage of multiple scales of the problem is far superior to the ill-conditioned conjugate gradient method.

Corotated Strain

As already shown in Figure 2.6, the artifacts introduced by the linear strain measure can almost entirely be avoided by using the corotational formulation of the linear Cauchy strain. Extra costs are involved to update the system matrix according to the current rotation frame. Consequently, the multigrid solver has to be updated, too. This is more costly for a non-nested hierarchy (marked by a $*$ in the table) than for a nested hierarchy. As shown in Section 2.5.7, matrix updates based on the novel 1-step stream accelerated algorithm are roughly 2–3 times faster for nested hierarchies than for non-nested ones. To improve the timings in the case of non-nested hierarchies, we clamp

matrix entries smaller than a certain threshold, e.g. 10^{-3}, to zero. Thereby, the structure of the restriction and interpolation matrix can be optimized, and the multigrid hierarchy updates can be computed slightly faster. On the other hand, the convergence rate of the multigrid method is not noticeably affected by these changes.

The timings given in Table 2.9 demonstrate, that most of the time is spent for matrix reassembling. Updating and solving the system is typically less than a third of the total time. The table lists in detail the time it takes to reassemble the system matrix ("Assem.") in the corotational setting and to built up the matrix hierarchy used by the multigrid approach ("Update"). As the implemented system optionally uses multiple threads for matrix reassembling, multigrid update and multigrid solve, the respective timings for the multi-threaded variants on an Intel Core™ 2 Duo CPU are given, too.

Model	# Level	# Tet	# Vert	Assem. [ms]	Update [ms]	Solve [ms]	ST Total [ms]	MT TPS [1/sec]	MT Time [ms]
Liver	2*	1467	464	7	1	5	13	126.0	7.9
Bridge	3	3072	825	15	2	3	20	62.0	16.1
Liver	3*	8078	1915	42	12	14	68	21.4	46.7
Breast	2*	10437	2542	49	10	23	82	16.5	60.6
Bunny	2*	11206	3019	57	11	23	91	15.6	64.1
Bridge	4	24576	5265	119	12	25	156	7.7	129.0
Horse	3*	49735	12233	285	77	74	436	3.1	322.0

Table 2.9: *Timing statistics in milliseconds [ms] for different models using the corotational simulation of linear strain. The major computational parts of the approach are analyzed. Beyond the single threaded (ST) algorithm, a multi-threaded (MT) variant is measured on the dual core architecture. The star $*$ denotes the use of a non-nested grid hierarchy.*

Recently, corotated strain has become popular in the graphics community. Müller and Gross [MG04] used element-based stiffness warping together with an optimized conjugate gradient method to solve the respective system of linear equations. They stated that 5000 tetrahedral elements can be simulated in 200 ms on an older Pentium 4 architecture. Additionally, they observed that the reassembling time is negligible compared to the time required by the numerical solver. This is in contrast to our method, where the multigrid solver is much cheaper than the reassembling process. Therefore, we can roughly handle five times more tetrahedral elements within the same time using a single simulation thread. Hauth and Strasser [HS04] proposed a similar method. For 337 tetrahedral elements they reported a simulation time of 15 ms per time step. They achieve further speed-ups by performing the update of the corotational frame and thus of the system matrix only in every 10th time step. However, this restricts the maximum

deformation that can occur between two consecutive time steps significantly. They propose a hierarchical approach that determines the rotation on a coarser grid and propagates it to a finer grid. However, this is only shown to be effective in combination with the delayed update of the corotational frame.

In comparison to the conjugate gradient methods, a significant speedup can be achieved by the multigrid approach. It was stated before that the time required by the conjugate gradient solver is the bottleneck of the corotated simulation [HS04]. However, due to the significant speedup of the multigrid solver, the reassembling task is now the most time-consuming part of the application. As a matter of fact, a performance gain of up to 10 compared to the conjugate gradient method is achieved for nested hierarchies (see Figure 2.29 left). Due to the fact that the performance is dominated by the reassembling phase (and not by the multigrid hierarchy update), nearly the same speedup can be observed for nested and non-nested hierarchies. However, for stiffer materials the performance gain becomes ever better as illustrated in Figure 2.29 on the right.

It should be stated that the simulation of the corotated Cauchy strain is not unconditionally stable in contrast to the linear Cauchy strain. This is due to the fact that the

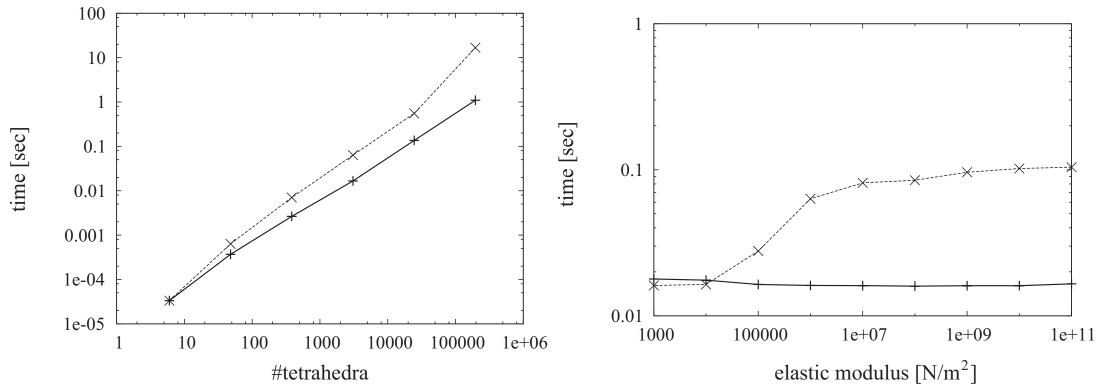


Figure 2.29: *Left: On a double logarithmic scale, timings are shown for the corotational simulation using the multigrid method including matrix reassembling and matrix hierarchy update (solid line) and the conjugate gradient method (dashed line). Timings were measured using an increasingly refined tetrahedral cube model (elastic modulus = $2 \cdot 10^6$ N/m^2, integration time step = 0.02 sec).*
Right: Performance measures for a bridge model (3k tetrahedra). For ever stiffer materials, the conjugate gradient method requires more steps to achieve the same relative error of 10^{-4} as the multigrid method. The elastic modulus affects the performance of the conjugate gradient method significantly while it does not affect the performance of the multigrid method. Only for extremely soft materials the performance of the multigrid method drops down because forces cause only very local deformations that cannot be solved for on a coarser grid.

element rotations are calculated explicitly. Thus, element rotations from the previous time step are used in the current time step. Nevertheless, in all examples using time steps between 20 ms and 30 ms we could not observe any instabilities. If the update of the rotational frame is delayed to every fifth time step as proposed by Hauth and Strasser [HS04], we quickly run into instabilities if larger forces are applied. Moreover, visual artifacts can occur due to large rotational discontinuities immediately after the rotation update. Therefore, we decided to perform the update in every frame, also achieving a constant numerical workload for every time step. Furthermore, we interleave the rotation update including the matrix reassembly and the solution of the system, which did not impose any drawbacks. This means that the rotations from the second recent time step are used within the current time step, but large discontinuities could not be observed because rotations are updated in every frame. This implementation can roughly double the performance on dual core architectures, since no communication besides the synchronization of the threads is required.

Non-Linear Green Strain

Compared to the linear setting, real-time performance for the non-linear strain setting can only be achieved if the number of elements is significantly reduced. However, compared to the corotational setting the performance is lower by only a factor of about 2 (see Table 2.10). The table includes explicit timings for the evaluation of the system

Model	# Level	# Tet	# Vert	Eval. [ms]	Update [ms]	Solve [ms]	ST Total [ms]	MT TPS [1/sec]	MT Time [ms]
Liver	2*	1467	464	15	2	7	24	44.4	23
Bridge	3	3072	825	33	2	5	40	29.5	34
Liver	3*	8078	1915	93	13	21	127	9.2	109
Breast	2*	10437	2542	121	12	32	165	6.5	155
Bunny	2*	11206	3019	151	13	31	195	5.9	170
Bridge	4	24576	5265	283	13	44	340	3.2	313

Table 2.10: *Timing statistics in milliseconds for different models using the non-linear Green strain measure. These include the evaluation of the system of non-linear equations and its Jacobian at runtime (Eval.), the update of the multigrid solver of the Jacobian (Update) and the solution time for one single Newton step (Solve). The total time in [ms] is given as well as the time steps per second (TPS) required if multiple threads (MT) are spawned. The star $*$ denotes the use of a non-nested grid hierarchy.*

of non-linear equations and its Jacobian at runtime. Furthermore, the update of the multigrid solver using the Jacobian matrix is shown as well as the solution time for one single Newton step. It is worth to mention that in dynamic simulations, which typically provide one with a good initial guess for the current solution, 1–2 Newton steps are usually sufficient.

In the examples given in the table, the compression due to the assembly of the system of equations is up to $3:1$. For the liver model, only 33% of the total amount of element monomials have to be evaluated. The worst compression is achieved for the large bridge model, where the number of monomials is reduced to 40% of the total amount. The amount of memory required to store the polynomials of the system of non-linear equations and its Jacobian is considerable. However, for all of our examples the total memory consumption of the application is still below 1 GB of RAM. In particular, the polynomials of the system of non-linear equations and its Jacobian require roughly 800 MB for the largest 25,000 tetrahedral model. The number of monomials to be evaluated at runtime ranges from one million for the smallest example to 40 million for the largest one.

The multi-threaded variant shown in Table 2.10 uses multiple threads, each of which evaluates a part of all polynomials. However, the performance gain is not significant because the evaluation is mainly memory bound on our architecture and both CPU cores use the same memory interface.

In comparison to other real-time approaches based on the Green strain tensor, our timings are promising. Debunne et al. [DDCB01] stated to simulate a few hundred elements in real-time on an admittedly outdated Pentium 3. However, due to the limitations of the explicit time integration scheme, the performance strongly depends on the stiffness of the simulated object. For the stiffest material with an elastic modulus of $10^6\,\mathrm{N/m^2}$ they used 5000 time steps per second, and thus the number of elements for which the simulation still achieves interactivity is significantly smaller. Picinbono et al. [PDA00] and Zhuang and Canny [ZC99] reported similar performances for their non-linear explicit methods. The Newton method applied in our solver is not unconditionally stable, since the system of equations $\mathbb{K}$ contains multivariate cubic polynomials, and thus the Newton method can run into local minima. Nevertheless, we can typically use larger time steps of up to 30 ms. Therefore, we can achieve a significantly better performance for stiff materials in general. On the other hand, for unnaturally soft materials, e.g. materials with an elastic modulus of $10\,\mathrm{N/m^2}$ as used in some previous publications, explicit methods are likely to outperform our method on todays CPUs.

The efficiency and effectiveness of the non-linear solver is also demonstrated in

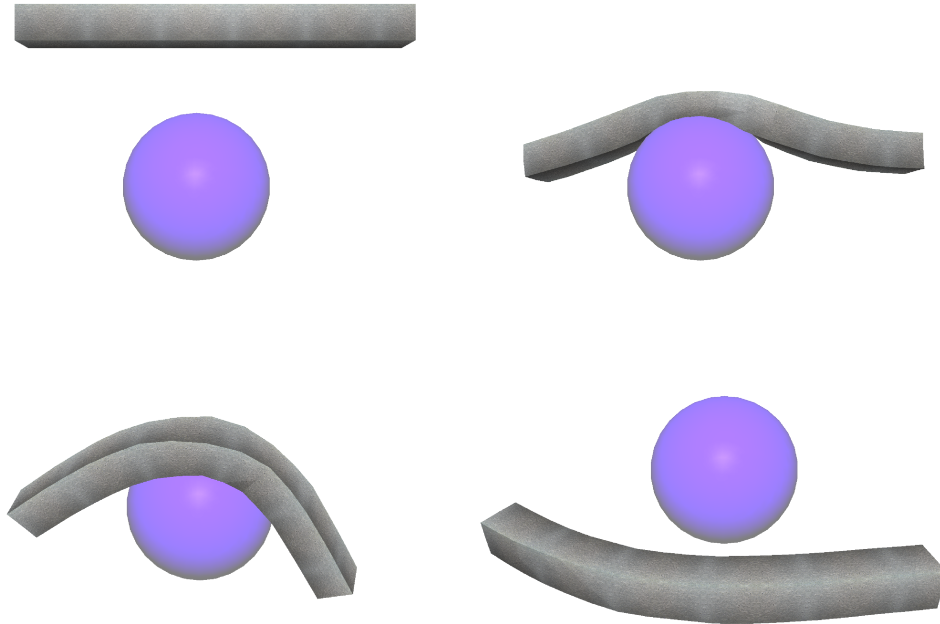


Figure 2.30: *A simple example demonstrates the potential of the proposed non-linear simulation engine. The bridge is discretized into 3k tetrahedral elements. Simulation and collision detection with static obstacles are run at 30 time steps per second.*

Figure 2.30, where large and global deformations of about 3,000 tetrahedral elements including collision detection and response to static objects is performed in real-time.

Higher-Order Elements

In addition to linear finite elements discussed previously, higher-order elements can be used in the simulation of the three strain types as well. In Table 2.11, we show timings for quadrangular elements using bilinear interpolation and Serendipity tetrahedra.

From the performance measurements, one can see that higher-order elements can still be used in real-time environments. Moreover, they allow for physically more accurate results due to the improved interpolation of the finite element solution. To compare the timings in Table 2.11 to linear elements, we consider the same number of vertices. For that reason, the respective linear elements are subdivided to yield the same number of vertices as the simulation based on higher-order elements. In case of Cauchy strain simulation, the simulation time for higher-order elements increases due to the higher fill ratio of the system matrix (compared to linear elements). In case of corotated strain, however, the matrix reassembly can be performed much more efficiently

Model	# ele	# vert	Cauchy strain [ms]	Corotated Strain [ms]	Green strain [ms]
quad (2D)	4096	4225	5	17	57
quad (2D)	16384	16641	21	56	228
quad (2D)	65536	66049	116	238	-
bridge (3D)	48	153	0.4	1.8	24
bridge (3D)	384	825	3.3	12.0	218
bridge (3D)	3072	5265	24.0	98.0	-

Table 2.11: *Timing statistics of the simulation of higher-order finite elements using the three strain measures. The total time required by one single time step is given in [ms]. In the 2D quad example, quadrangular elements with bilinear shape function were used. In the 3D bridge example, Serendipity tetrahedra were used.*

in case of higher-order elements due to the reduced number of elements, and thus the simulation times are noticeable faster. In case of non-linear simulation based on the Green strain formulation, the simulation of higher-order finite elements comes at the expense of more complex polynomials of higher order, which roughly doubles the simulation times compared to linear elements. We conclude, that higher-order elements are strongly advantageous in the corotated setting, since they allow to reduce the number of simulated elements without affecting the accuracy of the results. Therefore, an improved overall simulation performance can be achieved.

2.7.2 Comparison with Cholesky Solver

In a number of applications sparse direct solvers based on the Cholesky factorization have shown to be very efficient. Botsch et al. state that a direct Cholesky solver is superior to iterative solvers as well as multigrid solvers in the applications they have tested (Laplacian and bi-Laplacian systems) [BBK05]. Fortunately, we could directly compare our multigrid solver to the solver proposed by Botsch (based on the TAUCS library [TCR03]). For this purpose we used various models and we abruptly applied gravity to these models. Consequently, the multigrid solver could not benefit from the initial guess as it typically does in dynamic simulation environments. Note that the Cholesky solver as a direct solver can in no case benefit from an initial guess.

We choose the bunny model (11k tetrahedra), the bridge model (24k tetrahedra) and the horse model (50k tetrahedra). In Table 2.12, we show the relative error and solution times of both methods. The multigrid solver applied between 2 and 3 V-cycles. Timings for both the multigrid solver and the Cholesky solver were measured on an

	Multigrid			Cholesky		
model	init/update [ms]	error	solve [ms]	init/update [ms]	error	solve [ms]
Bunny11k	2218 / 11	0.051	14	295 / 175	3.7e-8	11
Bridge24k	1788 / 13	0.036	31	1266 / 761	1.5e-12	33
Horse50k	67916 / 81	0.19	70	1344 / 809	0.017	48

Table 2.12: *Comparison of the multigrid solver with a direct Cholesky solver [TCR03] for various models.*

Intel Core™ 2 Duo 2.4 GHz. We also list the initialization time of both solvers as well as the time required to update the system matrix *in-place* (the structure of the matrix stays the same, but the data values change).

Table 2.12 shows that the Cholesky solver (at least for relatively small models) achieves a significantly higher numerical accuracy. In addition, solution times of the Cholesky solver are slightly faster. Nevertheless, the errors achieved by the multigrid solver are visually sufficient. Thus, the deformations of the models do not change when solving more accurately. Especially in dynamic simulations, where a good initial guess is typically available, the multigrid solver is clearly superior in our experience. For the measurements, we took a snapshot out of a dynamic simulation for the bridge example. The multigrid solver achieved a relative error of $6.3 \cdot 10^{-3}$ within 14 ms, while the Cholesky solver required 28 ms yielding a relative error of $2.9 \cdot 10^{-14}$. The multigrid solver allows to trade performance for accuracy, thus yielding a more flexible setup. On the other hand, if high numerical accuracy is required and/or a good initial guess to the solution is not available (e.g., in a static elasticity problem), the Cholesky solver is favorable. Generally, the initialization times are lower, and the same numerical accuracy cannot be achieved with the multigrid solver in the same time.

However, the situation is totally different in simulation environments where the system matrix changes frequently (corotated Cauchy strain, non-linear Green strain), and thus the solvers have to be updated in every time step, too. Due to the novel 1-step stream acceleration scheme (see Section 2.5.4) used in our multigrid framework, in-place updates are several orders of magnitude faster than a respective in-place Cholesky factorization. Therefore, in these scenarios the multigrid solver shows a great advantage over the Cholesky solver.

2.7.3 Soft Tissue Validation

We have validated the simulation engine in the context of soft tissue deformations. We performed measurements on a liver of a pig since its material parameters are known

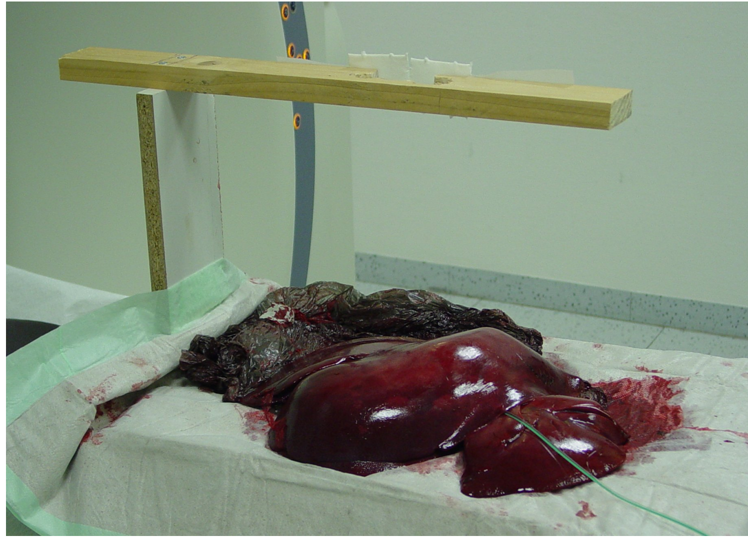


Figure 2.31: *Experimental setup of the soft tissue validation: The liver of a pig is placed in the CT scanner. The carrier allows one to put different weights on the liver while it is scanned.*

relatively well, and it can be obtained much easier than human tissue. To get a simulation model, the undeformed liver was digitized using a clinical CT scanner. In a well-specified environment we put different weights on the liver and digitized the deformed organ again with the CT scanner. Styrofoam was put in between the liver and the weight such that the surface of the liver is still clearly visible in the CT scan (Styrofoam does not block x-ray radiation). We have used water in a plastic bin as weight, since neither the water nor the plastic bin disturb the acquisition process. Styrofoam was glued at the bottom of this bin. The experimental setup that carries this bin is shown in Figure 2.31. After having scanned the deformed organs, the deformation of the initial model under the same load was simulated using the finite element multigrid approach.

In our experiment, we analyze the difference between the deformed surface of the simulation model and the real data set. We extracted the object's geometry from the volumetric scans using the marching cubes algorithm [LC87]. The surface mesh of the initial scan was then simplified [GH97], and a tetrahedralization was calculated using the NETGEN package [Sch97], yielding a tetrahedral mesh of about 40,000 elements. The material parameters were set homogeneously to an elastic modulus of $7500\ \mathrm{N/m^2}$ and a Poisson's ratio of 0.4. We then simulated a load of $175\,\mathrm{g}$ (contact surface has $24\,\mathrm{mm}$ in diameter) acting on the liver model. The location of the force template to be applied in the simulation is determined from the CT scan of the liver under load, since it shows a clear mold of the weight. The result of the simulation can be seen

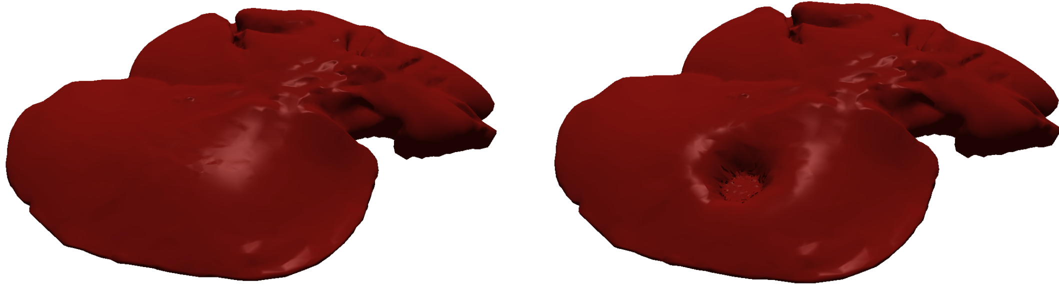


Figure 2.32: *Soft tissue validation by means of a pig's liver: The weight on the liver model (left) is simulated (right). The average distance between the simulated and measured surface was below the slice distance of the CT scanner.*

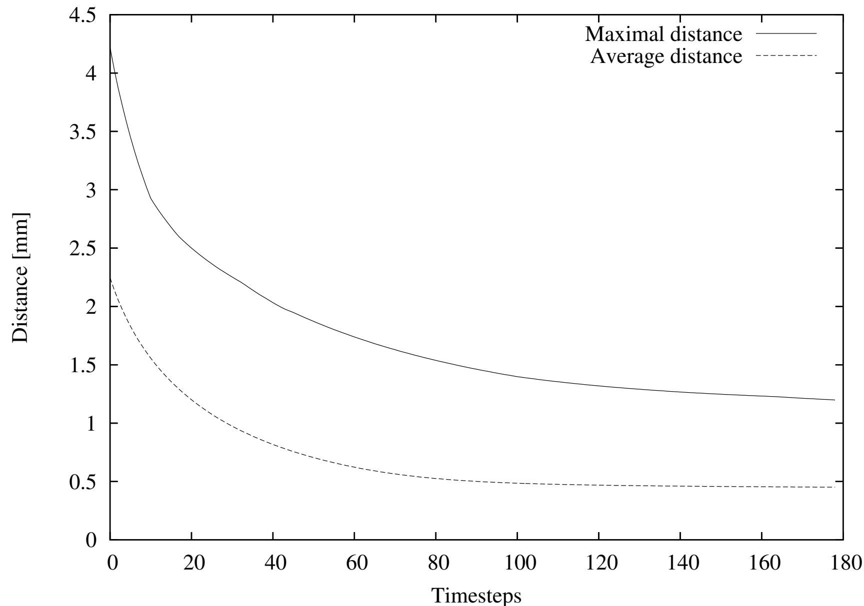


Figure 2.33: *Surface distance error of liver experiment: The average distance error in the dynamic simulation of a specific load on the liver quickly falls below 0.6 mm, which is the accuracy of the CT scanner used for digitization. The maximum error is about 3 times higher and occurs on the sharp boundary of the weight. Due to the limited resolution of the tetrahedral mesh the error does not disappear.*

in Figure 2.32. The distance of the dynamic simulation with respect to the surface extracted from the scan of the deformed liver is plotted in Figure 2.33. For every vertex of the simulation mesh, the shortest Euclidean distance to the deformed surface has been determined. From that information, the average vertex distance and maximum vertex distance as shown in Figure 2.33 have been obtained.

The results obtained with linear Cauchy strain are very good. This is due to the fact that the liver is only locally compressed and thus elements do not rotate too much. On the other hand, the precise material parameters of the real liver are not known. Our best results where achieved with an elastic modulus of $7500\ \mathrm{N/m^2}$. In the literature, a range of values from 3–$40\,\mathrm{kN/m^2}$ can be found, where higher values correspond to abnormal tissue and compressed tissue [YJH$^+$01]. Therefore we have the feeling that the assumed stiffness value was quite realistic for our ex-vivo experiments.

Chapter 3

Rendering

In this chapter, we focus on rendering aspects within the context of deformable objects simulation. The main contribution is the development of an advanced render engine on the GPU. It displaces and renders the deformable body according to the simulation mesh. First, a triangular render surface can be bound to the simulation mesh yielding high-quality visualizations of the surface of the deformable body. Second, a novel GPU volume rendering technique especially designed for deformable tetrahedral meshes has been developed. Analogously, the render engine can displace high-resolution tetrahedral meshes to achieve high-quality volume visualizations. Due to the system and algorithm design, only minimal bus bandwidth is required to transfer data from the CPU to the GPU for rendering purpose. The results of this chapter have been partially published [GW05b, GW06b].

The reason why we have also considered volume rendering of deformable bodies is twofold: First, volume rendering is useful to show internal structures of the deforming object. Second, volume rendering allows us to analyze internal material properties directly determined by the simulation, such as internal material stress or forces. The challenge here is to visualize time-varying volumetric data given on dynamic unstructured grids.

In the following, we first summarize related work in this field, and we then briefly describe very recent extensions of the programmable GPU rendering pipeline already introduced in Section 2.6.3. In Sections 3.3 and 3.4, we describe the design of our novel GPU render engine as well as the algorithms employed to deform and render high resolution render meshes and grids. Finally, we demonstrate the results we have achieved.

3.1 Related Work

3.1.1 Render Surface

High-quality renderings of deformable objects is an ongoing demand in the graphics community. Due to the limited resolution of the simulation grids in real-time applications, visualizations obtained from these grids are far below the rendering capabilities of current graphics hardware. Therefore, typically high-resolution render meshes are bound to the simulation grid, and they are displaced according to the deformations calculated on the coarse simulation grid.

High-resolution render surfaces have been frequently applied to improve the visual quality of animations [DDBC99, MDM$^+$02, MKN$^+$04, MG04, BK05, BPWG07]. All of the referenced approaches typically use functions with compact support to bind the high-resolution mesh to the simulation mesh. However, the deformed high-resolution mesh has to be transferred to GPU memory for the final rendering, since all of the calculations are performed on the CPU. Usually, this affects the performance of the entire application significantly, yielding non-interactive behavior in the worst case.

3.1.2 Volume Rendering of Unstructured Grids

We now briefly review techniques for direct volume rendering of unstructured grids. Object-space rendering techniques for tetrahedral grids accomplish the rendering task by projecting each element onto the view plane to approximate the visual stimulus of viewing the element. Three principal methods have been shown to be very effective in performing this task: slicing, cell projection and ray-casting.

Slicing approaches can be distinguished in the way the computation of the sectional polygons is performed. This can either be done explicitly on the CPU [WGTG96, YRL$^+$96], or implicitly on a per-pixel basis by taking advantage of dedicated graphics hardware providing efficient vertex and fragment computations [WE98, WE01, Wes01].

Tetrahedral cell projection [ST90], on the other hand, relies on explicitly computing the projection of each element onto the view plane. Different extensions to the cell-projection algorithm have been proposed in order to achieve better accuracy [SBM94, WMS98] and to enable post-shading using arbitrary transfer functions [RKE00]. To reduce visual artifacts that are typically for the cell projection approach, further improvements where achieved by Kraus et al. by applying correct perspective screen space interpolation and logarithmic sampling of the pre-computed transfer functions

[KQE04]. Graphics hardware based approaches for cell projection have been suggested, too [WKE02, WKME03b, WMFC02].

The most difficult problem in tetrahedral cell projection is to determine the correct visibility order of elements. The fastest way is PowerSort [CMSS95, KLNR03], which exploits the fact that for tetrahedral meshes exhibiting a Delaunay property the correct order can be found by sorting the tangential distances to circumscribing spheres using any customized algorithm. As grids in practical applications are usually not Delaunay meshes this approach might lead to incorrect results, because topological cycles in the data are not resolved correctly.

A different alternative is the sweep-plane approach [Gie92, SM97, SMK96, WE97]. In this approach the coherence within cutting planes in object space is exploited in order to determine the visibility ordering of the available primitives. In addition, much work has been spent on accelerating the visibility ordering of unstructured elements. The MPVO method [Wil92], and later extended variants of it [CKM$^+$99, SMW98], were designed to account for topological information for visibility ordering. Techniques using convexification to make concave meshes amenable to MPVO sorting have been proposed in [RE03]. Recently, a method to overcome the topological sorting of unstructured grids has been presented [CICS05] and extended to account for large meshes [CBPS06]. By using an initial sorter on the CPU a small set of GPU-buffers can be used to determine the visibility order on a per-fragment basis. Based on the early work on GPU ray-casting [PBMH02] a ray-based approach for the rendering of tetrahedral grids has been proposed in [WKME03a].

Besides the direct volume rendering of tetrahedral grids there has also been an ongoing effort to employ GPUs for iso-surface extraction in such grids [DK91]. The calculation of the iso-surface inside the tetrahedral elements was carried out in the vertex units of programmable graphics hardware [Pas04, RDG$^+$04]. Significant accelerations were later achieved by employing parallel computations and memory access operations in the fragment units of recent GPUs in combination with new functionality to render constructed geometry without any read-back to the CPU [KSE04, RDG$^+$04, KW05]. To avoid the explicit construction of iso-surface geometry, approaches based on tetrahedra slicing [WE98] or cell projection [RKE00, WKE02, WKME03b, WMFC02] have been used as well for iso-surface rendering. Very recently, an optimized hierarchical CPU approach for iso-surface rendering has been proposed [WFKH07].

3.2 Direct3D 10 Graphics Pipeline

In this chapter, we exploit the GPU for rendering purpose. We have already introduced the architecture of GPUs in Section 2.6.3. Although early generations of graphics cards were optimized for this purpose using a fixed-function pipeline, the programmability enables the use of advanced rendering techniques. For example, the possibility to displace vertex positions in the vertex stage allows us to effectively deform geometry directly on the GPU.

The basic programmable rendering pipeline has already been illustrated in Figure 2.20. Note that very recent hardware architectures include an additional programmable stage—the geometry shader. This stage allows us to amplify the incoming stream of primitives. In other words, from each incoming primitive, an arbitrary (but bounded above) number of output primitives can be generated. The general pipeline as it is proposed by Direct3D 10 [BG05] and implemented on latest GPUs is shown in Figure 3.1. Due to the great impact of the geometry shader on the hardware architecture, current graphics hardware implies a strong performance impact if the geometry shader is to be used to its full potential, i.e., on NVIDIA GeForce 8800 cards. For that reason, all implementations and timings given in the following were measured without the use of the geometry stage. However, some of the presented algorithms can benefit from the geometry shader stage once its performance is comparable to the other shader stages.

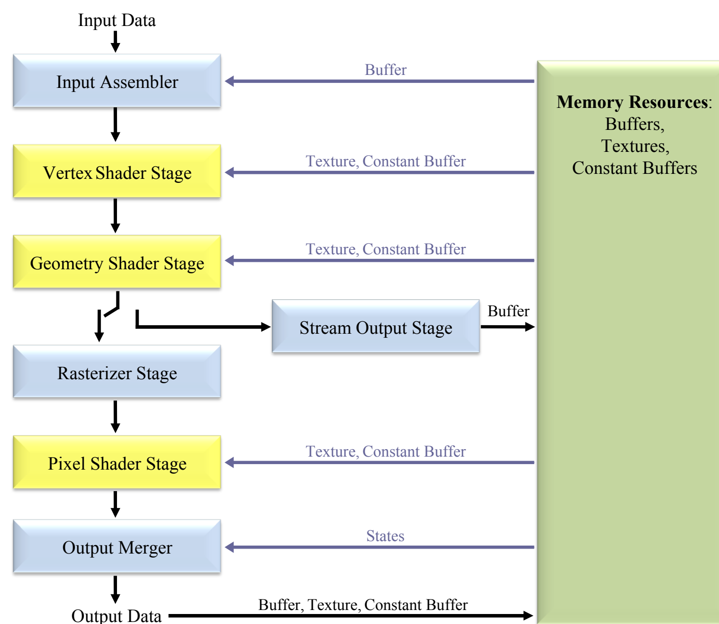


Figure 3.1: *Stages of the Direct3D 10 graphics pipeline implemented on recent GPUs.*

3.3 Deformable Surface Rendering

We now describe an advanced method to update a render mesh, i.e., a triangular mesh, on the GPU—according to uploaded displacements of the boundary of the simulation mesh. In this way, high-quality renderings of deformable bodies can be obtained. Since bandwidth requirements are significantly reduced, these renderings can be achieved at highly interactive rates. Vertices of the render mesh are bound to vertices of the simulation mesh via interpolation weights, which are pre-computed and stored on the GPU. At runtime, only the computed displacements of each simulation vertex need to be transferred, and thus the deformation of high-resolution meshes can be driven by the simulation engine at minimum bus transfer. Furthermore, high parallelism and memory bandwidth of recent GPUs can be exploited to update the render mesh.

3.3.1 High-Resolution Render Mesh

The high-resolution render surface, which resides in local GPU memory, is represented as an index array that contains references into a geometry image with associated per-vertex attributes for every triangle. Both the index array as well as the geometry image are internally stored as 2D textures. Every vertex in the geometry image gets assigned additional references into the displacement texture that is sent from the CPU. These references are accompanied by barycentric interpolation weights. Four references are stored, one for each vertex of the (surface) tetrahedron closest to the vertex of the render mesh. Once displacements of the simulation mesh are uploaded to the GPU, a fragment program fetches respective displacement coordinates u_i and interpolation weights b_i and computes the new vertex position as

$$v = v_{orig} + \sum_{i=0}^{3} u_i \cdot b_i, \tag{3.1}$$

where v_{orig} is the undeformed reference position. If additional vertex attributes are sent with the displacement texture, e.g. color, these values are interpolated using the same weights if desired. The fragment output is rendered into a 2D texture render target. To finally render the displaced triangular mesh, different possibilities are available on recent GPUs: for example, OpenGL `PixelBufferObjects` and vertex texture fetches using Direct3D Shader Model 3.0 or GLSL, just to name the most prominent ones.

On traditional graphics architectures, textures could only be accessed in a fragment

shader program. The Shader Model 3.0 and the GLSL specification also enable texture access in the vertex units hence providing an effective means for displacing geometry on the GPU. This functionality is supported on most graphics hardware currently available. To render a displaced surface, we render a static vertex array stored in GPU memory. In a vertex shader program, vertex positions are fetched from the texture that contains the displacements, and the vertex position is translated by the displacement vector. Therefore, any read back of data to CPU memory is avoided.

Render Surface Binding

In this section, we show how to bind a high-resolution render surface to the simulation grid. We decided to bind a single vertex of the render surface to a tetrahedral element of the simulation mesh. In this way, we can guarantee continuous (piecewise linear) displacements of the render surface for vertices bound to adjacent (at least one common face) tetrahedral elements.

For vertices of the render surface that lie in the interior of the simulation mesh, the barycentric coordinates of the respective tetrahedral elements are stored as interpolation weights. For all other vertices, a unique tetrahedral element is determined. The barycentric coordinates b_0, b_1, b_2, and b_3 of the render surface vertex with respect to each tetrahedral element are calculated, and a tetrahedron is selected by minimizing the following function:

$$w = \max\left(\{0\} \cup \{|b_i| : b_i \leq 0, i = 0, \ldots, 3\}\right).$$

This function effectively penalizes elements that are far away from the render surface vertex considered (they have at least one negative value b_i).

It is obvious that this metric is only a rough approximation and is likely to fail in some settings. However, for all models considered in this thesis, the results were convincing and the render surface deformed smoothly with the simulation mesh. Finally, for each vertex of the render surface, we store four barycentric weights and references to the respective vertices in appropriate texture maps on the GPU.

GPU Update of the Render Surface

The vertices of the render surface are displaced by using a pre-pass shader that generates the shared vertex array of the displaced geometry. The data required is a texture storing the barycentric weights b_0, b_1, b_2, and b_3, a texture storing the respective simulation grid indices j_0, j_1, j_2, and j_3 as well as a texture containing the displacements u_i of

the simulation grid's vertices. Then, a quad is rendered to generate as many fragments as there are vertices in the high resolution surface. A fragment shader calculates the displacement vector $\sum_{i=0}^{3} b_i \, u_{j_i}$ for each vertex and writes it into the render target. In a final render pass, the non-deformed geometry is rendered, and a vertex shader reads the displacement texture generated in the pre-pass to update the vertex position respectively. An schematic overview of the processing steps is given in Figure 3.2.

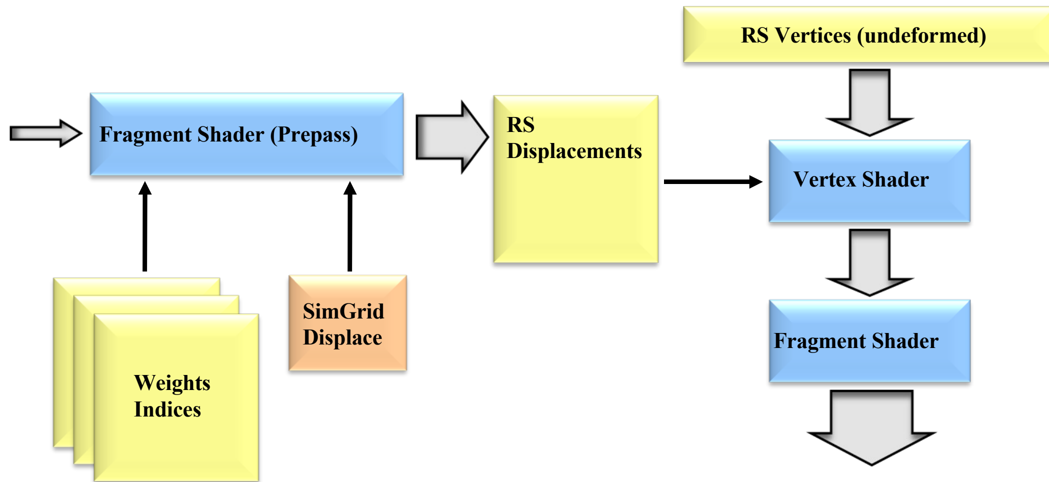


Figure 3.2: *Processing steps of displacing a high resolution render surface (RS) on the GPU.*

Normal Calculation

Recalculation of per-vertex normals of the deforming render surface can effectively be done in two different ways on the GPU. Here we assume that the normal is determined by averaging the per-face normals of all adjacent triangles.

Scattering: For every face of the render surface, a cross product has to be performed and the result has to be scattered to the respective vertex position of the geometry image, where it is accumulated to achieve a per-vertex normal. The scatter operation produces both memory and computation overhead.

Gathering: For each vertex of the render surface, per-face normals are gathered. This operation is inefficient if the render surface mesh is not regular and thus not every vertex has the same number of adjacent triangles to fetch normals from.

To accelerate the calculation of per-vertex normals, we describe an update technique for the normals based on an initial decomposition of the render surface normals n. To the

best of our knowledge, such an update strategy has not been applied to the rendering of deformable objects so far. Here we assume that we know the normals n of the undeformed render surface. They are decomposed into two parts: an interpolated coarse grid normal N and an offset vector d. To determine the first part, we project the render surface vertex on the respective tetrahedral surface face. Given the barycentric weights w_1, w_2, and w_3 of this projected vertex with respect to the tetrahedron and the simulation grid normals $N_1, \ldots, N_3$ at the respective face's vertices $v_1, \ldots, v_3$, the interpolated normal is obtained by $N = \sum_{i=1}^{3} N_i\, w_i$. Finally, the normal of the undeformed render surface, n, can be decomposed into a simulation mesh normal part $w_0 N$ and a face-parallel part d:

$$n = w_0\, N + d,$$

where $d = \sum_{i=1}^{3} w_{i+3}\, v_i$ can be expressed in local barycentric coordinates w_4, w_5, and w_6 of the respective face. Since d is a vector (rather than a position), the sum $\sum_{i=1}^{3} w_{i+3} = 0$ equals zero. The simulation normal scaling w_0 can be obtained by projection of N and n onto the face normal n_f:

$$w_0 = \frac{n_f^{\mathrm{T}}\, N}{n_f^{\mathrm{T}}\, n}.$$

Figure 3.3 illustrates this decomposition. If w_0 is determined, we can directly calculate the barycentric coordinates $w_4, \ldots, w_6$ of the remaining offset $d = n - w_0 N$. Therefore, we store the weights $w_0, \ldots, w_6$ and the vector d in appropriate textures for each vertex of the render surface.

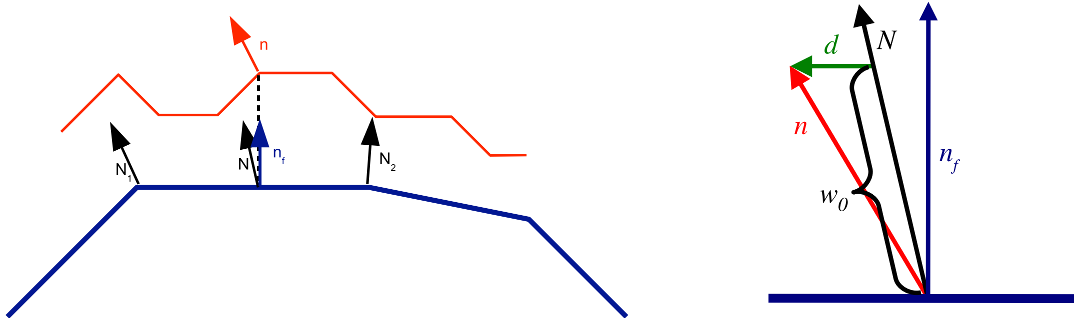


Figure 3.3: *Decomposition of the high-resolution render surface normal n (red) into an interpolated simulation normal $w_0 N$ (black) and an offset vector d parallel to the face with normal n_f (green).*

At runtime, the normal $\tilde{n}$ of a vertex of the deformed render surface can be approximated by

$$\tilde{n} = w_0 \sum_{i=1}^{3} w_i \, \tilde{N}_i + d + \sum_{i=1}^{3} w_{i+3} \, u_i,$$

where w_i are the pre-calculated weights, $\tilde{N}_i$ are the normals of the appropriate deformed simulation grid and u_i are the displacement vectors of the corresponding vertices in the simulation grid[1]. By pre-multiplying w_1, w_2, and w_3 with w_0, we can avoid to store w_0 explicitly. Because the offset vector is obtained in local barycentric coordinates of the deformed triangle, calculated normals are consistent with the structure of the render surface.

One remaining problem is that if triangles are scaled, the offset part d scales, too. Normalizing the normal N does not cure the problem as this yields to unnaturally looking normals due to the changing ratios of both parts. Therefore, we use area-weighted normals N_i and $\tilde{N}_i$ in both the pre-processing stage as well as at runtime. Before lighting calculations are actually performed, the normals $\tilde{n}$ are normalized.

Figure 3.4: *Per-pixel lighting of the deformed render surface (32k triangles) looks naturally even for large deformations. The surface can be displaced and rendered on the GPU at high frame rates.*

Although the normal calculation is only approximately, the images shown in Figure 3.4 demonstrate that the quality of the generated normals is visually pleasant. The calculations of the new normals can be performed in the same shader that updates the vertex positions using the OpenGL `MultipleRenderTarget` extension. An additional (dependent) texture fetch into the texture storing the normals of the simulation grid using the same texture coordinates that are used to access the displacement texture has to be performed. In addition, the weights w_i and the pre-computed offset d have

[1]By re-ordering the indices and weights used to displace the mesh in Equation (3.1) in such a way, that u_1, u_2, and u_3 correspond to the face vertices, we do not need to store additional references.

to be accessed in the shader. In the current implementation, the performance difference between the shader executed with and without normal update is negligible. This is further validated in Section 3.5.

3.3.2 Advanced Shading Techniques

To achieve realistically looking animations, a fur shader has been integrated. Based on the work by Tomohide [Tom02], Praun et al. and Lengyel et al. [PFH00, LPFH01], fur rendering can be performed by blending several shells, which are textured with an appropriate fur texture. In contrast to the previous work, our method geometrically extrudes the shells in the vertex stage. Therefore, so-called fin textures [PFH00] rendered at the silhouettes can be avoided if an adequate number of shells is used. Moreover, our approach allows for dynamic fur effects.

The fur texture used has the important property that α-values decrease when moving to outer shells. It is realized as a 3D texture, where the third coordinate encodes the shell identifier. Typically, 20–30 shells are enough to generate images comparable to those in Figure 3.5. The shells are generated by extruding the mesh in the vertex shader using the normals of the (shared) vertices. To avoid multiple vertex texture fetches to determine the vertex coordinates of the deformed mesh, the deformed vertices are copied into a separate vertex array using the OpenGL `PixelbufferObject` extension [Ope04]. This array can be directly rendered for each of the shells to be considered. Each shell's identifier is used both to determine how far each vertex is extruded into its normal direction, and which layer of the 3D fur texture is fetched in the fragment stage.

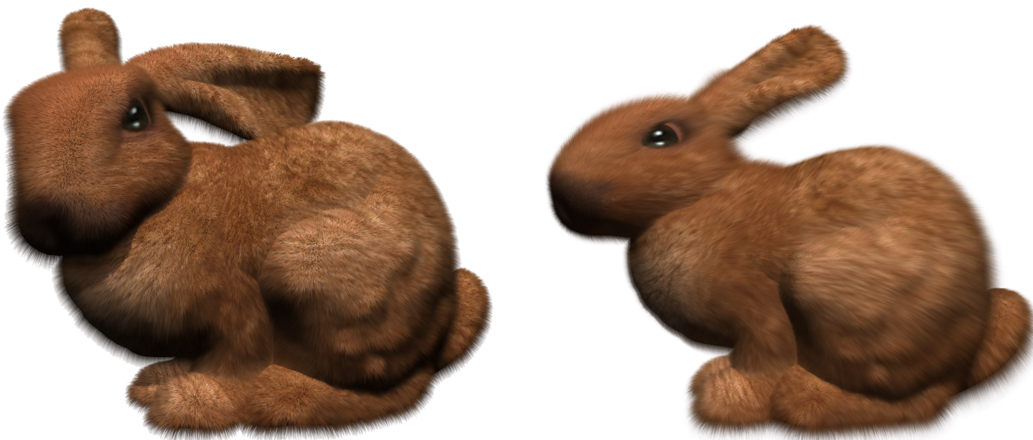


Figure 3.5: *Fur shading yields realistic-looking imagery (left). On the right, the effect of dynamic fur is demonstrated.*

To account for the motion of fur in real world, a simple dynamic model is included by storing the vertex array of the previous time step. All in all, the vertex position v_l^t for shell layer l is determined as follows:

$$v_l^t = v^t + f\frac{l}{m}\tilde{n}^t + 3 \cdot 2^{\frac{l}{m}} \cdot \log_2(v^{t-1} - v^t + 1).$$

Here, v^t and v^{t-1} are the deformed positions of the base shell at time steps t and $t-1$, $\tilde{n}^t$ is the normal of vertex v^t, f is the fur length, m is the maximum number of used shell layers, and the $\log_2$ is applied to each component of the vector. To avoid intersections of consecutive shells, the dynamic term (the last term) is enforced to be clamped such that its Euclidean norm is smaller than the distance to the base shell $f\frac{l}{m}$.

3.4 Deformable Volume Rendering

Although recent advances in graphics hardware have opened the possibility to efficiently render tetrahedral grids on commodity PCs, interactive rendering of large and deformable grids is still one of the main challenges in scientific visualization. Such grids are more and more frequently encountered in a number of different applications such as plastic and reconstructive surgery, virtual training simulators, and visualization in fluid and solid mechanics.

If we imagine a ray-casting approach, for each sample point along a ray the tetrahedral element the sample point is contained in has to be determined. The geometry of this element is used to compute the point's position in the local coordinate space of the element. For this purpose, an element matrix built from the element's vertex coordinates can be used. For each element this matrix has to be computed only once and can then be used to re-sample the data at every sample point in its interior. To do so, a container storing the matrices of all elements has to be created on the GPU. It is clear that this approach significantly increases the memory requirements. Moreover, because the re-sampling is performed in the fragment stage, every fragment needs to be assigned the unique identifier of the element it is contained in to address the respective matrix. In scan-conversion algorithms this can only be done by issuing these identifiers as additional per-vertex attributes during the rendering of the tetrahedral elements. Unfortunately, because every vertex is in general shared by many elements, a shared vertex list can no longer be used to represent the grid geometry on the GPU. This causes an additional increase in memory.

To avoid the memory overhead induced by pre-computations, element matrices can

be calculated on the fly for every sample point. But then the same computations, including multiple memory access operations to fetch the respective coordinates, have to be performed for all sample points in the interior of a single element, thereby wasting a significant portion of the GPUs compute power. As before, identifiers are required to access vertex coordinates, and thus a shared vertex array cannot be used.

3.4.1 Tetrahedral Grid Rendering Pipeline

In this section we present a novel GPU pipeline for the rendering of tetrahedral grids that avoids the aforementioned drawbacks. This pipeline is scalable with respect to both large data sets as well as future graphics hardware. The proposed method has the following properties:

- Per-element calculations are performed only once.
- Tetrahedral vertices and attributes can be shared in vertex and attribute arrays.
- Besides the shared vertex and attribute arrays nearly no additional memory is required on the GPU.
- Re-sampling of (deforming) tetrahedral elements is performed using a minimal memory footprint.

To achieve our goal we propose a generic and scalable GPU rendering pipeline for tetrahedral elements. This pipeline is illustrated in Figure 3.6. It consists of multiple stages performing element assembly, primitive construction, rasterization, and per-fragment operations.

To render a tetrahedral element the pipeline is fed with one single vertex, which carries all information necessary to assemble the element geometry on the GPU. As-

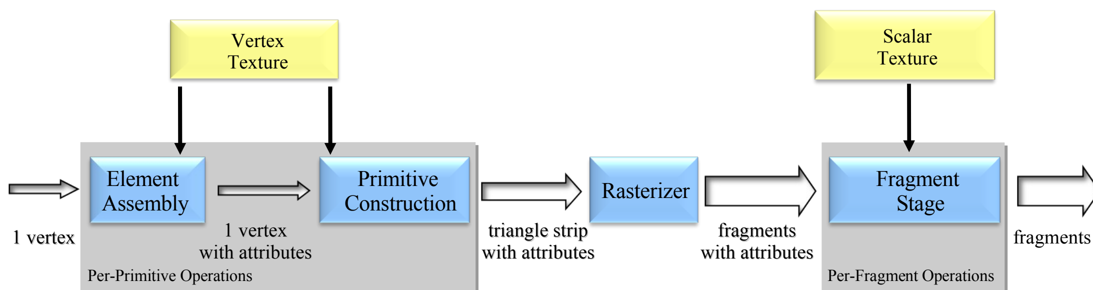


Figure 3.6: *Overview of the GPU tetrahedral grid rendering pipeline.*

sembled geometry is then passed to the construction stage where a renderable representation is built.

The construction stage is explicitly designed to account for the functionality on very recent graphics hardware. With Direct3D 10 compliant hardware and geometry shaders [BG05] it is possible to create additional geometry on the graphics subsystem. In particular, triangle strips composed of several vertices, each of which can be assigned individual per-vertex attributes, can be spawned from one single vertex. As the geometry shader itself can perform arithmetic and texture access operations, these attributes can be computed to take application-specific needs into account. By using the aforementioned functionality the renderable representation can be constructed in turn without sacrificing the feed-forward nature of the proposed rendering pipeline.

Although hardware-assisted geometry shaders are available on recent NVIDIA GPUs, the use of the geometry stage slows down the overall rendering pipeline significantly. In particular, it is still faster to render the triangle strips from a vertex array than to generate it on the fly using the geometry stage. Therefore, we have implemented the proposed pipeline using the older functionality as provided by Direct3D 9 or OpenGL 2.0. We emulate the primitive construction step using a *render-to-vertexbuffer* functionality. The specific implementation will be discussed in Section 3.4.4. Although this emulation requires additional rendering passes, it still results in frame rates superior to what can be achieved by the fastest methods known so far.

Once the renderable representation has been built it is sent to the GPU rasterizer. On the fragment level a number of different rendering techniques can be performed for each tetrahedron, including a ray-based approach, iso-surface rendering, and cell projection. The discussion in the remainder of this section is focused on the first approach, and we describe the other rendering variants later in Sections 3.4.2 and 3.4.3.

The ray-based approach operates in a similar way as ray-casting by sampling the tetrahedral grid along the view rays. In contrast, it does not compute the set of elements that are hit consecutively along each ray but it lets the rasterizer compute the set of rays intersecting each element. The interpolation of the scalar field at the sample points in the interior of each element is then performed in the fragment stage, and the results are finally blended in the color buffer.

The developed approach requires the tetrahedral elements to be sampled in correct visibility order. To avoid the explicit computation of this ordering, we first partition the eye coordinate space into spherical shells around the point of view. Each shell consists of a fixed number of spherical slices (which define the sample points on each ray). Figure 3.7 illustrates this partitioning strategy.

These shells are consecutively processed in front-to-back order, simultaneously keeping the list of elements that intersect the current shell. Intra-shell visibility ordering is then achieved by re-sampling the elements onto spherical slices positioned at equidistant intervals in each shell (see right of Figure 3.7). Elements smaller than the selected sampling rate can thus be missed.

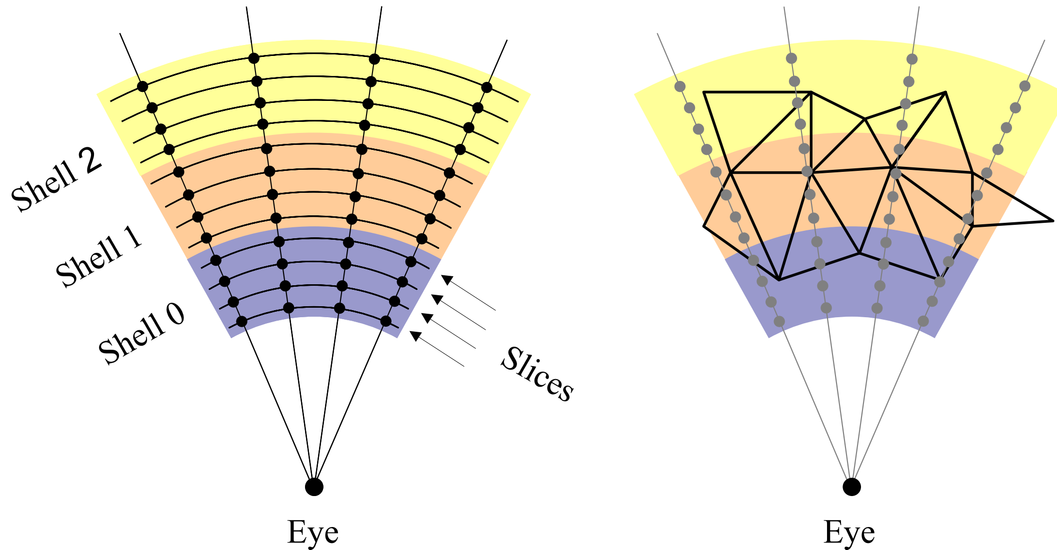


Figure 3.7: *Ray-based tetrahedra sampling: Tetrahedral elements are re-sampled onto spherical slices around the point of view.*

Tetrahedra intersecting more than one shell are stored in each of the respective lists, and they might thus be rendered multiple times. In every rendering pass, however, every element is only re-sampled onto the slices contained in the current partition.

In the following we describe the rendering pipeline for tetrahedral grids in more detail. Essentially, it is a sampling of the attribute field at discrete points along the view rays through the grid. The sampling process effectively comes down to determining for each sampling point the tetrahedron that contains this point as well as the point's position in local barycentric coordinates with respect to this tetrahedron. Due to this observation we decided to rigorously perform the rendering of each element in local barycentric space. The following benefits are achieved by this approach:

1. Barycentric coordinates of sample points can directly be used to interpolate the scalar values given at the grid vertices.

2. Barycentric coordinates can efficiently be used to determine whether a point lies inside or outside an element.

3. By transforming both the point of view and the view rays into the barycentric coordinate space of an element, barycentric coordinates of sample points along the rays can be computed with a minimum number of arithmetic operations.

4. Barycentric coordinates of vertices as well as barycentric coordinates of the view rays through the vertices can be issued as per-vertex attributes, which then become interpolated across the element faces during rasterization.

Figure 3.8 shows a conceptual overview of the entire rendering pipeline for tetrahedral grids. The process is also described in Algorithm 8. For the sake of clarity, only pseudo-code is given.

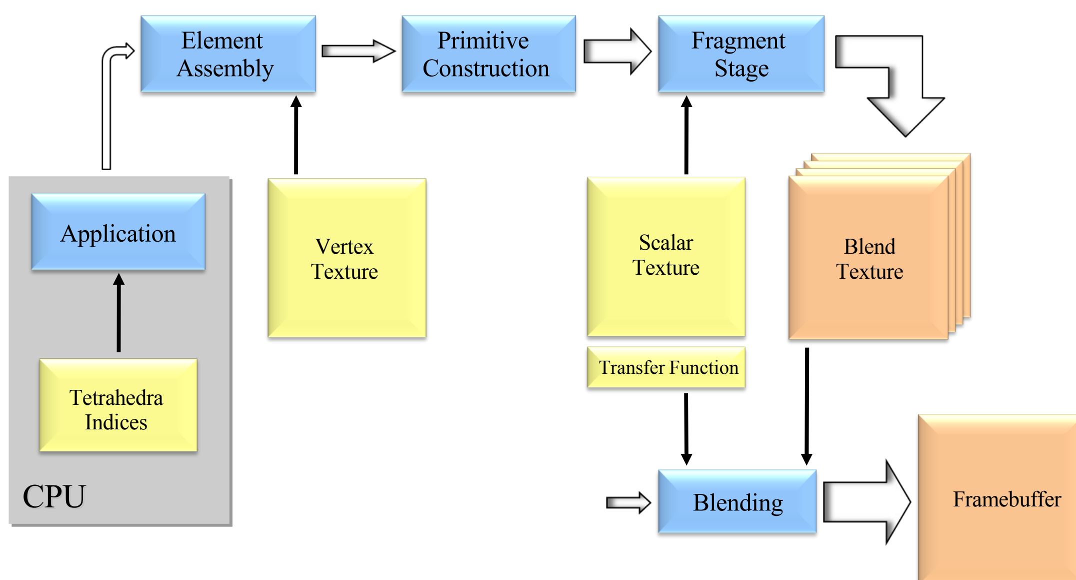
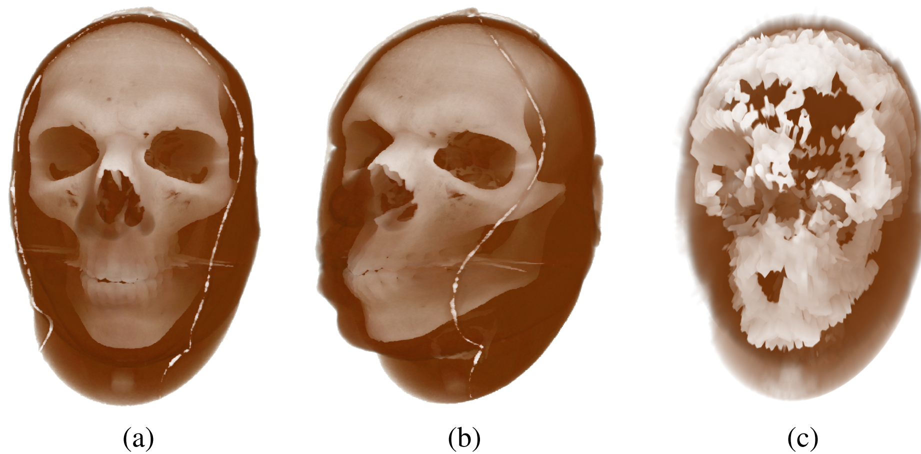


Figure 3.8: *Data stream overview of the GPU tetrahedra rendering pipeline.*

Data Representation and Transfer

The tetrahedral grid is maintained in a compact representation: a shared vertex array that contains all vertex coordinates, and an index array consisting of one 4-component entry per element. Each component represents a reference to the vertex array. While the index array only resides in CPU memory, the vertex array is stored both on the CPU and in a 2D floating-point texture on the GPU. Additional per-vertex attributes such as scalar or color values are only held on the GPU.

By assigning a 3D texture coordinate to each vertex it is also possible to bind a 3D texture map to the tetrahedral grid. By one additional texture indirection the scalar



(a) (b) (c)

Figure 3.9: *Deforming a Cartesian grid (visible male data set): the* $512^2 \times 302$ *Cartesian grid (a) is displaced and rendered by means of a textured tetrahedral mesh consisting of* $500k$ *elements using our approach (b). Image (c) clearly demonstrate that the mesh resolution is not sufficient if scalar values are only sampled at the vertices.*

or color values can then be sampled via interpolated texture coordinates from a 3D texture map. This strategy is particularly useful for the efficient rendering of deformed Cartesian grids. By deforming the geometry of a tetrahedral grid while keeping the 3D texture coordinates fixed, the deformed object can be rendered at considerably higher resolutions compared to an approach using linear interpolation of the scalar field given at the displaced tetrahedra vertices (see Figure 3.9).

To render a tetrahedral grid, the CPU computes the set of elements (active elements) intersecting each spherical shell. This can be easily achieved by determining the minimum and maximum distances to the view point for each tetrahedron, and by comparing these values to the shell borders. Each time a shell is to be rendered, the CPU uploads its active element list, which is represented by a 4-component index array. This list is then passed through the proposed rendering pipeline.

Element Assembly

For each shell to be rendered the active element list contains one vertex per element, each of which stores four references into the vertex texture. In the element assembly stage these indices are resolved by interpreting them as 2D texture coordinates. The four vertices are obtained by four texture access operations, and they are then transformed into the eye coordinate space. The transformed vertices are passed to the primitive construction stage along with the four indices.

Primitive Construction

In the primitive construction stage the per-element information that is needed in the upcoming stages is constructed. First, for every element the matrix required to transform eye coordinates into local barycentric coordinates is computed. The vertices, given in homogeneous eye coordinates, are denoted by $v_i, i \in \{0, 1, 2, 3\}$. The transformation matrix can then be computed as

$$V_B = \begin{pmatrix} v_0 & v_1 & v_2 & v_3 \end{pmatrix}^{-1}.$$

Next, the eye position $v_{eye} = (0, 0, 0, 1)^{\mathrm{T}}$ is transformed into the barycentric coordinate space of each element: $b_{eye} = V_B \, v_{eye}$. It is important to note that only the last column of V_B is required. Thus, the number of required arithmetic operations is significantly reduced. The barycentric coordinates of each vertex v_i are given by the canonical unit vectors e_i. Finally, the directions of all four view rays passing through the element's vertices are transformed into barycentric coordinates via $b_i = e_i - b_{eye}$. As the mapping from eye coordinate space to barycentric coordinate space is affine, these directions can later be linearly interpolated across the element faces. In addition, the length of the view vector $l_i = \|v_i - v_{eye}\|_2$ is computed for each vertex in the primitive construction stage. It is used in the fragment stage to normalize the barycentric ray directions b_i.

Once the aforementioned per-element computations have been performed, each tetrahedron is rendered as a triangle strip consisting of four triangles. These strips are composed of six element vertices, which are first transformed to normalized device coordinates. The respective b_i, the barycentric eye position b_{eye}, and the length of the view vector l_i are assigned to each vertex as additional attributes. Moreover, four indices into the GPU attribute array are assigned to each vertex. These indices are later used in the fragment stage to access the scalar field or the 3D texture coordinates used to bind a texture map.

The rasterizer generates one fragment for every view ray passing through a tetrahedron and interpolates the given per-vertex attributes. To reduce the number of generated fragments, only front-faces are rendered using back face culling.

Fragment Stage

When rendering the primitives composed of attributed vertices in the described manner, the rasterizer interpolates the b_i and l_i and generates a local barycentric ray direction b as well as its length l in eye coordinates for each fragment. By using the barycentric

Algorithm 8 Pseudo-code snippets for ray-based GPU tetrahedra rendering.

elementAssembly $(index)$

  for $i = 0, \ldots, 3$ **do**
    $v_i =$ texture $(vertexTex,\ index_i)$;
    $v_i =$ Modelview $* v_i$;
  end for
  return $(index, v_0, v_1, v_2, v_3)$;

primitiveConstruction $(index, v_0, v_1, v_2, v_3)$

  $V_B =$ inverse $((v_0, v_1, v_2, v_3))$;
  $b_{eye} = V_B * (0, 0, 0, 1)^{\mathrm{T}}$;
  for $i = 0, \ldots, 3$ **do**
    $l_i =$ length $(v_i - (0, 0, 0, 1)^{\mathrm{T}})$;
    $b_i = e_i - b_{eye}$;
    $v_i =$ Projection $* v_i$;
  end for
  Rasterize strip using
    $\begin{pmatrix} v_0 \\ b_0 \\ l_0 \end{pmatrix}, \begin{pmatrix} v_1 \\ b_1 \\ l_1 \end{pmatrix}, \begin{pmatrix} v_2 \\ b_2 \\ l_2 \end{pmatrix}, \begin{pmatrix} v_3 \\ b_3 \\ l_3 \end{pmatrix}, \begin{pmatrix} v_0 \\ b_0 \\ l_0 \end{pmatrix}, \begin{pmatrix} v_1 \\ b_1 \\ l_1 \end{pmatrix}$
  return $(index, b_{eye})$;

fragmentStage(interpol. v, interpol. b, interpol. l,
        $index, b_{eye}$, const z_s, const Δz_s)

  for $i = 0, \ldots, 3$ **do**
    $s[i] =$ texture $(scalarsTex,\ index_i)$;
  end for
  for $k = 0, \ldots, n$ **do**
    $bc = b_{eye} + b/l * (z_s + k * \Delta z_s)$;
    if min $(bc[0], bc[1], bc[2], bc[3]) < 0$
      $out[k] = 0$;
    else
      $out[k] =$ dot (s, bc);
    end if
  end for
  return (out);

coordinates of the eye position b_{eye}, the view ray in local barycentric space can be computed for each fragment as

$$\frac{b}{l} \cdot t + b_{eye}, \quad t > 0,$$

where t denotes the ray parameter. This ray is sampled on a spherical slice with distance z_s from the eye point. The barycentric coordinate of the sample point is obtained by setting t as the depth of the actual spherical slice, z_s.

It is now clear that every fragment has all the information to determine the barycentric coordinates of multiple sample points along the ray passing through it. If an equidistant sampling step size Δz_s along the view rays is assumed, the coordinates of each point are determined as

$$b^k = \frac{b}{l} \cdot (z_s + k \cdot \Delta z_s) + b_{eye}, \quad k \in \{0, 1, \ldots, n-1\}. \tag{3.2}$$

where n is the number of samples. The fragment program obtains the depth z_s of the first sample point and the sample spacing Δz_s as constant parameters.

A fragment can trivially decide whether a sample point is inside or outside the tetrahedron by comparing the minimum of all components of b^k to zero. A minimum greater or equal to zero indicates an interior point. In this case the sample point is valid and thus has a contribution to the color being accumulated along the ray. Otherwise, the sample point is invalid and has to be discarded.

The barycentric coordinates are directly used to interpolate per-vertex attributes. These can be scalar values that are first fetched from the attribute texture via the per-vertex indices issued, or it can be a 3D texture coordinate that is used to fetch a scalar value from a texture map. Finally, each fragment has determined one scalar value for each of its n samples.

Once the scalar field has been re-sampled onto a number of sample points along the view rays, these values in principle can be directly composited in the fragment program. Unfortunately, as the elements within one spherical shell have not been rendered in correct visibility order, this would lead to visible artifacts. On the other hand, we can write four scalar values at once into one RGBA render target. Moreover, recent graphics APIs allow for the simultaneous rendering into multiple render targets. Let m denote the number of render targets available. This means that up to four times m spherical slices can be re-sampled by one single fragment. Sampled values are rendered into the respective component and render target using a $\max$ blend function. If a sample point is outside the element, a zero value is written into the texture component and the sample is

ignored. As no two tetrahedra contain the same sample point along any ray (apart from samples at a shared face that yield the same scalar value for both elements), erroneous results are avoided.

The number of samples that can be processed efficiently at once is restricted by the output bandwidth of the fragment program. Because four render targets are available on recent GPUs, up to 16 slices can be processed at once. This implies that the thickness of each spherical shell is chosen to contain exactly 16 slices with regard to the current sampling step size. To allow for this number, four additional texture render targets have to be used to keep intermediate sampling results. Without utilizing the multiple render target extension, four samples can be processed at once. To store the scalar values, either an 8 bit fixed-point or a 16 bit half-floating-point format can be used, thus keeping the memory bandwidth low.

Blending Stage

In the final stage up to four texture render targets are blended into the frame buffer. In each of its four components these textures contain the sampled scalar value on one spherical slice of the shell. The blending stage now performs the following two steps in front-to-back order. First, scalar values are mapped to color values via a user-defined transfer function. Second, a simple fragment program performs the blending of the color values via alpha-compositing and finally outputs the result to the frame buffer.

3.4.2 Iso-Surface Rendering

To avoid the explicit construction of iso-surface geometry on the GPU, per-pixel iso-surface rendering can be integrated into our proposed rendering pipeline. Instead of sampling all the values along the view rays, only the intersection points between these rays and the iso-surface are determined on a per-fragment basis. Thereby, the primitive assembly and element construction stage remain unchanged, and only the fragment stage needs minor modifications.

The combination of the ray equation in barycentric coordinates (with the unknown parameter t_{iso})

$$b_{iso} = b \cdot t_{iso} + b_{eye}$$

with the barycentric interpolation of the known iso-value $s_{iso} = \sum_{i=0}^{3} s_i \cdot (b_{iso})_i$ yields

$$s_{iso} = t_{iso} \sum_{i=0}^{3} s_i(b)_i + \sum_{i=0}^{3} s_i(b_{eye})_i \,.$$

Therefore, the view ray passing through a fragment intersects the iso-surface at depth

$$t_{iso} = \frac{s_{iso} - \sum_{i=0}^{3} s_i \cdot (b_{eye})_i}{\sum_{i=0}^{3} s_i \cdot (b)_i}.$$

It is worth noting that t_{iso} is undefined if the denominator is zero. In this case the interpolated scalar values along the ray are constant, and we can either choose any valid value for t_{iso} if the scalar value is equal to s_{iso} (the frontmost value is given at the entry point of the ray, which can be determined as described in the Section 3.4.3) or the ray has no intersection with the iso-surface.

The computed barycentric coordinate b_{iso} of the intersection point is tested against the tetrahedron as described above. Only if the point is in the interior of the element an output fragment is generated. Otherwise the fragment is discarded.

In this particular rendering mode the data representation stage has to be modified slightly. Instead of building an active element list for every shell, only one list that contains all elements being intersected by the iso-surface is built. These tetrahedra can then be rendered in one single pass, or in multiple passes if more elements are intersected by the surface than can be stored in a single texture map. The blending stage becomes obsolete and can be replaced by the standard depth test to keep the front-most fragments in the frame buffer. A fragments' depth value is set to the depth of the intersection point in the fragment program.

Lighting

To perform lighting on the extracted iso-surface on a per-pixel basis, per-element gradients can be computed in the primitive construction stage as well. Given the scalar values s (s is the vector of all scalar values of the elements' vertices) and the barycentric matrix V_B, the gradient can be determined by

$$\nabla S = \left[V_B^{\mathrm{T}} s\right]_{0-2}.$$

The last component of the vector $V_B^{\mathrm{T}} s$ is ignored. This formula is derived from the fact that the scalar field is interpolated in the tetrahedral element. Given the barycentric coordinates $b_j = b_j(x), 0 \leq j \leq 3$, for an arbitrary point x in the interior of the element, the gradient reads as

$$(\nabla S)_i = \frac{\partial}{\partial x_i} \left(\sum_{j=0}^{3} b_j \cdot s_j \right) = \sum_{j=0}^{3} \frac{\partial b_j}{\partial x_i} s_j \;\; \forall\, 0 \leq i < 3,$$

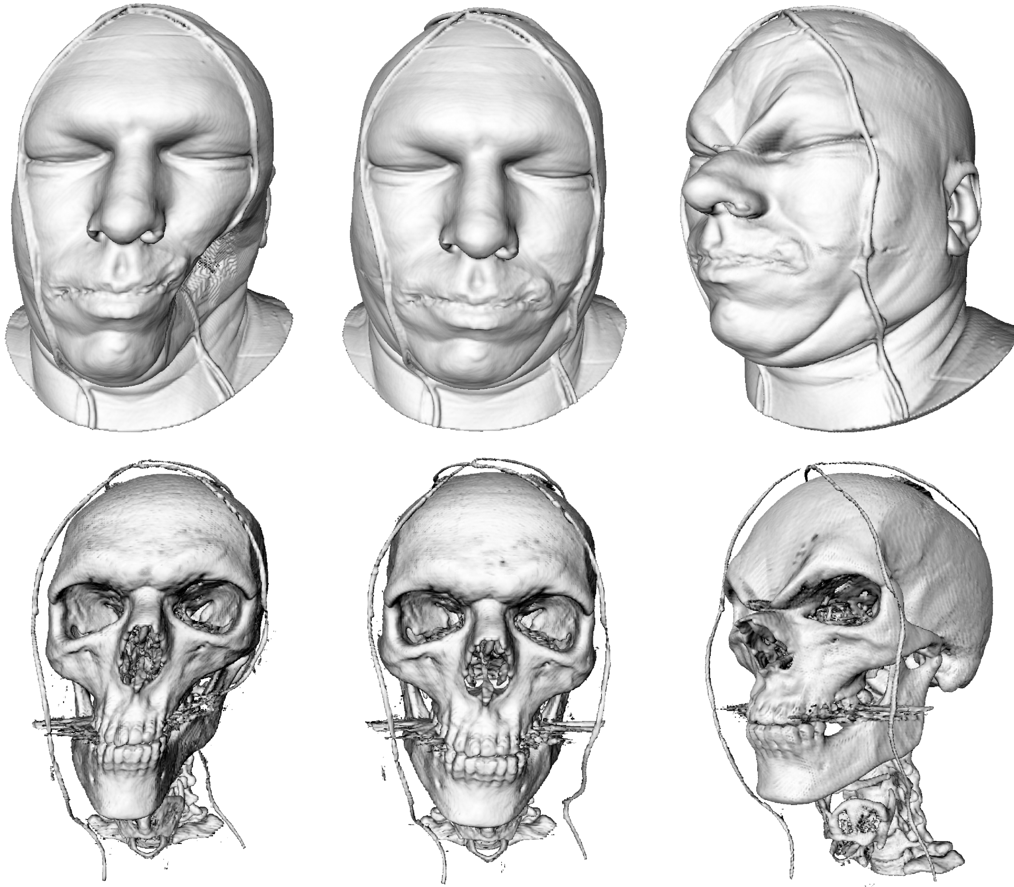


Figure 3.10: *Iso-surface rendering of the deformed visible male data set. The tetrahedral mesh was adaptively refined to recover the skin and bone structures and consists of 5.1 million elements. Per-vertex scalar values were re-sampled from the original 3D data set.*

where b_j can be written using the barycentric matrix V_B:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = V_B \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ 1 \end{pmatrix}.$$

Thus, $\frac{\partial b_j(x)}{\partial x_i} = (V_B)_{ji}$ and finally

$$(\nabla S)_i = \sum_{j=0}^{3} (V_B)_{ji}\, s_j = (V_B^{\mathrm{T}} s)_i.$$

Gradients are assigned as additional per-vertex attributes, and they are then used

in the fragment stage for lighting calculations. Since the gradients are constant within each tetrahedral element, a flat shaded iso-surface is obtained. To get smooth lighting, gradients of all incident tetrahedra have to be averaged at the (shared) vertices. This requires a pre-pass, which calculates per-element gradients and scatters their contribution to all element vertices, where vertex gradients are finally accumulated. These gradients are stored in a separate gradient texture that is accessed in the fragment stage of the pipeline. By means of the barycentric coordinates b_{iso} of the considered sample point, an interpolated gradient can be determined from the four accessed vertex gradients. An example of smooth lighting is given in Figure 3.10.

3.4.3 Cell Projection

Tetrahedral cell projection is among the fastest rendering techniques for unstructured grids as every element is only rendered once but it requires a correct visibility ordering of the elements. To integrate tetrahedral cell projection into our pipeline we employ the tangential distance or Powersort [KLNR03] on the CPU to determine the visibility ordering, because this method is both fast and simple to implement. It guarantees correct visibility sorting for meshes exhibiting the Delaunay property.

Although finite element meshes often have this property, it can be violated due to the deformations applied. As a matter of fact, the meshes we render by means of the proposed method may not be sorted correctly. In particular, when applying large forces to deform the models, such artifacts can appear (see Figure 3.11). These artifacts are totally avoided in the ray sampling approach. On the other hand, it requires tetrahedra to be rendered several times as they might overlap more than one shell.

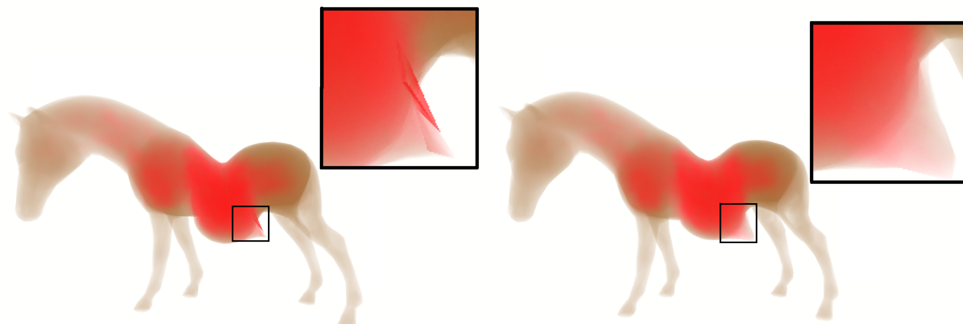


Figure 3.11: *The left image shows visual artifacts that can occur due to the approximate visibility sorting of the Powersort algorithm. The right image demonstrates that these artifacts are avoided by means of our ray sampling approach.*

Tetrahedral cell projection is achieved by a modification of the fragment stage. Given the fragment's distance to the eye position z_{in}, the barycentric coordinates of this fragment can be computed as

$$b_{in} = \frac{b}{l} \cdot z_{in} + b_{eye}. \tag{3.3}$$

The intersection of a view ray with each of the faces of the corresponding tetrahedron can be calculated using the ray equation $b_{out} = b/l \cdot t + b_{eye}$ in barycentric coordinates. To compute b_{out}, four candidate parameters $t_k, k \in \{0, \ldots, 3\}$ are obtained by solving the equations $0 = b_k/l \cdot t_k + (b_{eye})_k$. These equations describe the fact that the sample point b_{out} has to lie on one of the tetrahedral faces; thus, at least one barycentric coordinate of b_{out} is zero. As the ray parameter $t = z_{in}$ corresponds to the entry point of the ray, the value of t at the exit point is determined by

$$z_{out} = \min\{t_k : t_k > z_{in}, k = 0, \ldots, 3\}.$$

The barycentric coordinate of the exit point can then be derived according to equation (3.3), and it can be used to calculate the scalar value at the ray's exit point.

The length of the ray segment inside the tetrahedron can be calculated by $z_{out} - z_{in}$. This information is required to compute a correct attenuation value for every fragment as described by Stein et al. [SBM94]. The barycentric coordinates are used to obtain scalar values at the entry and exit point, which are then accumulated along the ray[2].

3.4.4 Implementation

On current graphics hardware, e.g. NVIDIA 8800 cards, the fastest implementation is achieved if the geometry shader is not used. This is due to the fact, that the geometry shader still introduces significant performance drawbacks. Hence, the primitive assembly stage and the primitive construction stage are simulated via multiple rendering passes. The implementation is then also suited for older graphics cards.

Once the CPU has uploaded the index texture to the GPU (see Section 3.4.1), a quad covering four times as many fragments as active elements is rendered. Each fragment reads its respective index to perform one dependent texture fetch to get the corresponding vertex coordinate. The 4th component of each vertex is used to store the element index. This index is used in the final fragment stage to fetch the barycentric transformation matrix. The vertex coordinates are written to a texture render target, which is

[2]Linear transfer functions are assumed here for simplicity.

either copied into a vertex array (on NVIDIA cards) or directly used as a vertex array (on ATI cards). In this pass, the transformation of vertices into eye coordinates can already be performed.

In a second pass, each active tetrahedron reads its four vertices as described and computes the last row of the barycentric transformation matrix, b_{eye}, which is stored in an RGBA float texture. Due to the fact that only one index per tetrahedron can be stored, we also build an RGBA texture that stores the four scalar values associated with each active element in one single texel. If 3D texture coordinates are required they are stored analogously in three RGBA textures.

We then use an additional index array to render the tetrahedral faces. We either use 7 indices per tetrahedron to render a triangle strip followed by a primitive restart mark (on NVIDIA cards only) or we use 12 indices to render the tetrahedral faces separately. Note that the index array does not change and can be kept in local GPU memory.

Finally, the fragment stage has to be modified such that every fragment now fetches b_{eye} and performs all operations required to sample the element along the view rays in local barycentric space. Although this increases the number of arithmetic and memory access operations considerably, we will show later that the implementation already achieves excellent frame rates on customary graphics hardware.

3.4.5 Visualization of Internal Material Properties

The direct volume rendering method can be used to visualize internal properties of the materials being deformed. One property that is of particular interest in a number of applications is the internal stress. To compute the stress, we utilize the element stress matrices

$$\int_\Omega D\,B\,\mathrm{d}x$$

to assemble a global stress matrix. Here, D is the material law (2.8) and B is the element strain matrix (2.13). By applying the computed displacement field to this matrix, an averaged stress tensor is derived for each vertex. According to the linearization of the stress tensor (2.2), the six entries of the symmetric tensor are obtained per vertex.

To visualize the tensor, we calculate the so-called von Mises stress norm [Bat02] using the linearized form σ of the tensor:

$$\sigma_{Mises} = \sqrt{3\sum_{k=4}^{6}\sigma_k^2 + \frac{3}{2}\sum_{k=1}^{3}(\sigma_k - \bar{\sigma})^2} \quad \text{with} \quad \bar{\sigma} = \frac{1}{3}\sum_{k=1}^{3}\sigma_k.$$

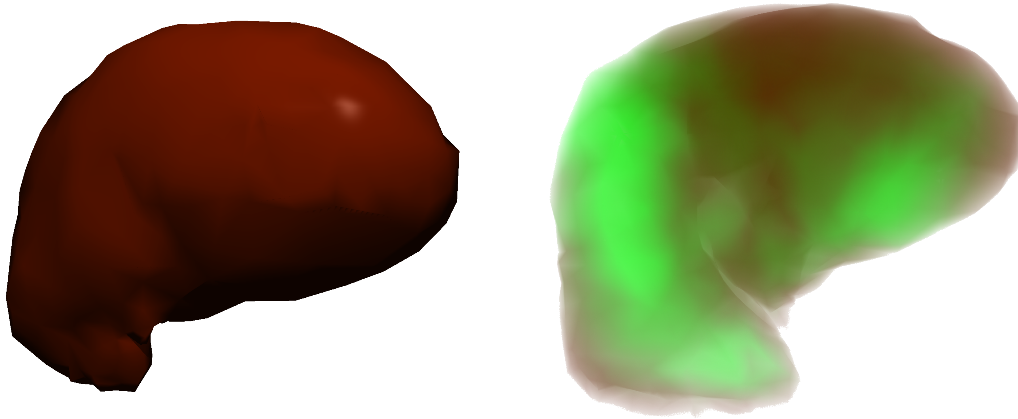


Figure 3.12: *Left: Boundary of an undeformed liver. Right: Visualization of the deformed liver by means of direct volume rendering. The internal stress induced by the deformation is color-coded from red (low stress) to green (high stress).*

This norm yields a scalar value for each vertex, which can then be visualized by the presented tetrahedral grid rendering approach. This norm is used in a number of applications to determine the critical stress of the deformed material. The critical stress indicates that the material is substantially changed and does not show an elastic behavior any more.

In medical applications, the von Mises stress norm can be employed to visualize tissue stress. Due to a clinical intervention tissue can be stressed too much, thereby inducing additional risks for the patient. By means of the proposed simulation and visualization support system a surgeon can obtain immediate visual feedback about the induced stress in all parts of the used volumetric model during surgical training or pre-operative planning. And he can immediately adjust the intervention procedure adequately. Figure 3.12 shows a stress visualization using a human liver data set. Due to the influence of external forces the liver is highly stressed on its left side. To demonstrate the effect of heterogeneous materials on the internal stress, we show the heterogeneous horse model under gravity. As can be seen in Figure 3.13, the soft abdomen is highly stressed whereas the stiffer legs induce much lower stress.

3.4.6 High-Resolution Render Volumes

Analogously to the high-resolution render surfaces described in Section 3.3, a high-resolution tetrahedral grid can be bound to the simulation grid using barycentric coordinates. Since the vertex positions of this mesh can be updated on the GPU in exactly the same manner as described, the upload of updated vertex positions of the high-resolution

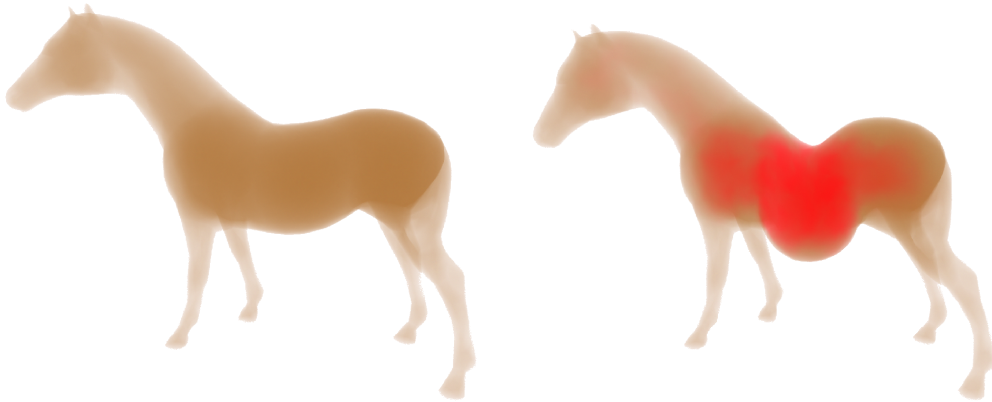


Figure 3.13: *Our method can be applied to visualize internal states of deforming volumetric bodies efficiently. In the example, the internal stress of the heterogeneous horse model under gravity is visualized in red.*

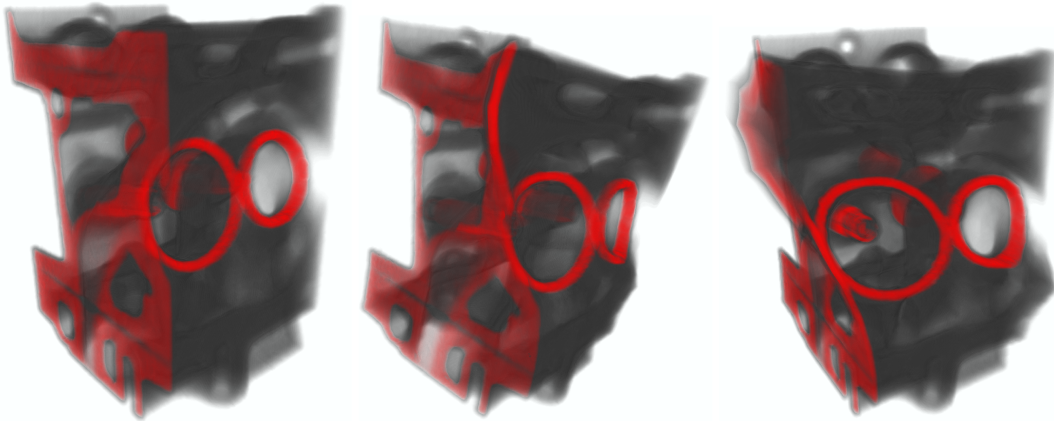


Figure 3.14: *These images show direct volume renderings of a tetrahedral mesh consisting of 1600k elements, which are bound to a 24k simulation mesh. A 3D texture of size* $256^2 \times 110$ *storing the engine data set is bound to the mesh.*

mesh can be avoided. This allows us to displace high-resolution meshes interactively while at the same time achieving high quality images.

In Figure 3.14, the deformation of a Cartesian data set of resolution $256^2 \times 110$ is shown. The simulation mesh consists of 24k tetrahedral elements, and the render mesh is built of 1600k tetrahedra. To achieve best visual quality, the rendering of the high-resolution mesh is further improved by storing the data set into a 3D texture and applying this texture in the volume rendering pass. Every vertex stores coordinates into this 3D texture instead of a scalar value. In the fragment stage, interpolated coordinates are then used to fetch color values from the 3D texture map, which are finally blended in the frame buffer.

3.5 Performance Measurements

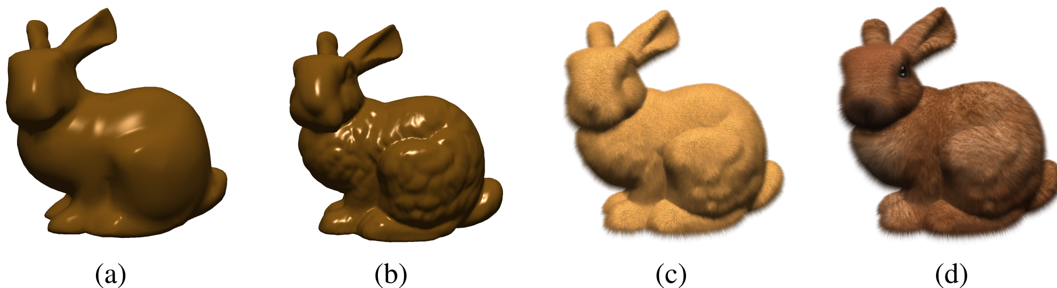
3.5.1 Surface Rendering

In the following we give performance measurements for the GPU render engine. All timings were measured on an NVIDIA 8800 GTX graphics card. As shown in Table 3.1, the update of the high-resolution render surface comes nearly for free such that the rendering of these surfaces can be achieved at highly interactive frame rates. The rendering includes the time to displace the geometry in the vertex shader. The timings measured include also normal calculations as described in Section 3.3.1 and per-pixel lighting in the fragment shader.

Figure 3.15 shows the bunny model consisting of 11k tetrahedral elements. Besides

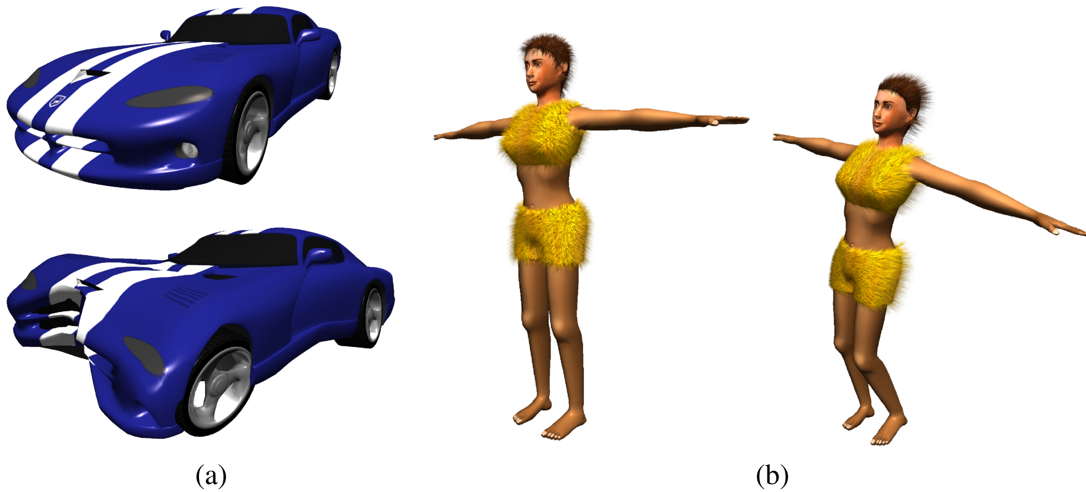
Model	#Tris	Update	Rendering	Fur (20 layers)
Sarah	16k	$<$1 ms	2980 fps	212.0 fps
Car	36k	$<$1 ms	1920 fps	102.0 fps
Bunny	64k	$<$1 ms	756 fps	23.1 fps
Horse	64k	$<$1 ms	842 fps	20.3 fps
Dragon	871k	1 ms	181 fps	3.7 fps

Table 3.1: *Performance of the GPU surface render engine for different models (NVIDIA 8800 GTX). In the third column, timings are given for uploading displacement textures of the simulation mesh (size 16^2 to 64^2) to the GPU, and for updating the render geometry. The fourth column shows the overall performance of the GPU render engine including the update of the render surface, normal calculation, texture fetches in the vertex shaders to access the displaced vertex coordinates and the normals, as well as per-pixel lighting. The last column shows the performance (including update) that can be achieved if a fur shader using 20 extruded shells is used for rendering.*



(a) (b) (c) (d)

Figure 3.15: *(a) Interactive deformation and rendering of the bunny finite element model (11206 simulation tetrahedra). (b) A high resolution surface with 64k triangles is bound to the simulation mesh. (c) A reduced surface (16k triangles) is rendered using the GPU-based fur shader. (d) By applying a texture map to the fur, further realism can be achieved.*



(a) (b)

Figure 3.16: *(a) Interactive deformation and rendering of the car model (7k simulation tetrahedra) can be performed with 150 fps. The high resolution surface consists of 36k triangles. (b) Interactive deformation and rendering of a manikin. The fur shader takes the dynamic of the body into account as can be seen on the right hand side.*

a high-resolution render surface (64k triangles), a GPU fur shader can be applied as described in Section 3.3.2. Rendering performance decreases as can be seen in Table 3.1 since several layers have to be rendered. However, the geometry has only to be updated once. Therefore, instead of using a vertex texture fetch in every rendering pass, the displaced vertices are copied into a vertex buffer once using the OpenGL `PixelBufferObject` extension. Figure 3.16 gives additional examples for interactive deformations using the proposed GPU render engine.

3.5.2 Volume Rendering

We now analyze the performance of the tetrahedral grid rendering pipeline, and we give timings for different parts of it. All tests were run on an Intel Core™ 2 Duo CPU equipped with an NVIDIA 7900 GTX graphics processor. The size of the viewport was set to 512×512.

We have tested the tetrahedral rendering pipeline for both static and deformable meshes using the simulation engine as described before. The GPU render engine receives computed displacements and updates the geometry of a volumetric body accordingly. While the simulation engine consecutively displaces the underlying finite element grid, the render engine subsequently changes the geometry of the volumetric render object. It is worth noting that all timings presented in this section exclude the amount of time required by the simulation engine. In all our examples, the time re-

quired to send updated vertices to the GPU is less than 1% of the overall rendering time.

The proposed technique for direct volume rendering of unstructured grids is demonstrated in Figures 3.17 to 3.19. Table 3.2 shows performance rates on our target architecture implementing the rendering pipeline described in Section 3.4.4. Timing statistics for the alternative rendering modes are given in Table 3.3.

Scene	horse	bluntfin	engine	vmhead
# Tetrahedra	50k	190k	1600k	3800k
# Samples / ray	300	400	500	600
# Samples / shell	4	8	8	8
# Tets rendered	133k	424k	3438k	6618k
Vertices/scalars [MB]	0.27	1.1	17	17
Blend textures [MB]	1	2	2	2
Intermediate [MB]	3.3	13	13	13
GPU memory [MB]	4.6	16.1	32	32
CPU [ms]	2	6	63	156
GPU Geometry [ms]	11	12	65	135
GPU Fragments [ms]	43	85	445	732
Total time [ms]	56	103	573	1023

Table 3.2: *Element, memory, and timing statistics of the tetrahedral grid rendering pipeline for various data sets (NVIDIA 7900 GTX).*

The first four rows of Table 3.2 show the number of tetrahedral mesh elements, the number of sample points per ray, the number of samples per shell and the total number of elements being rendered. As elements are likely to intersect more than one shell, the number of rendered elements is approximately 2 times higher than the mesh element count. Next, GPU memory requirements (excluding 3D texture maps) are shown. The memory required by the vertex, scalar, and blend textures is listed. Additional memory that is necessary due to the emulation of the construction stage on GPUs without support for geometry shaders is summarized in the next row.

As shown by the measurements, the proposed rendering pipeline exploits the limited GPU memory very effectively. On the other hand, even if the mesh does not fit into local GPU memory the method can still be used efficiently. One possibility is to partition the grid, and thus the vertex and attribute textures, into equally sized blocks. These blocks can then be rendered in multiple passes, only requiring a separate active element list for each partition and shell.

The next rows in Table 3.2 give detailed timings of the different rendering stages.

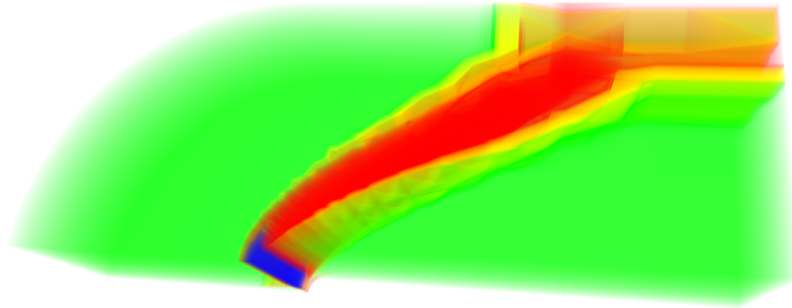


Figure 3.17: *Direct volume rendering of the NASA bluntfin data set.*

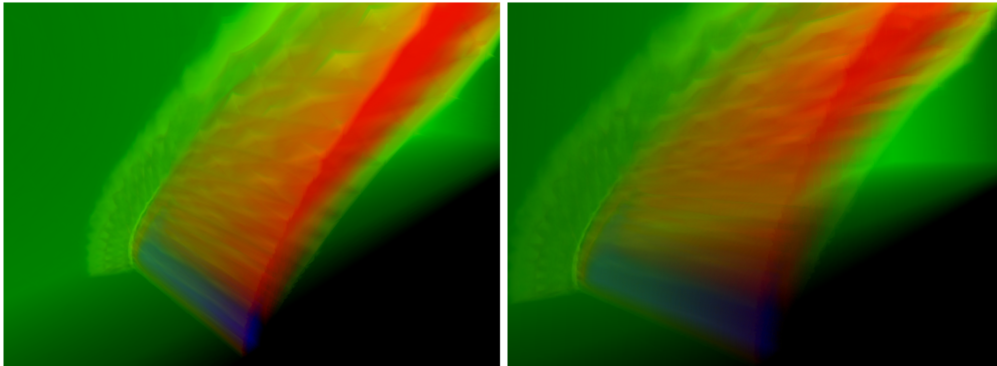


Figure 3.18: *Close-up views of the NASA bluntfin data set.*

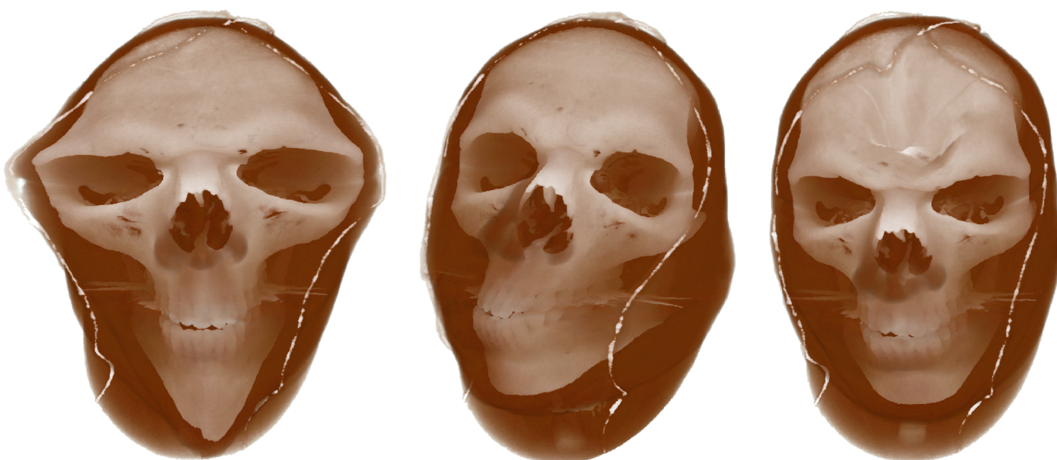


Figure 3.19: *Direct volume rendering of the deformed visible human data set. The tetrahedral mesh consists of 3.8 million elements, and it is textured with a* $512^2 \times 302$ *3D texture map.*

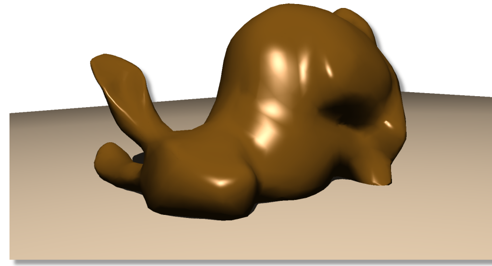
Scene	horse	bluntfin	engine	vmhead
Iso-Value	0.5	0.2	0.5	0.27
Iso-Surface [ms]	4.6	5.7	51	124
Cell Projection [ms]	19	54	341	1176

Table 3.3: *Timing statistics for different rendering modes (NVIDIA 7900 GTX).*

All timings are given in milliseconds. Starting with the time required by the CPU to calculate the active element sets and to transfer all required data to the GPU, timings for GPU primitive assembly and construction as well as per-fragment computations are given.

From the timing statistics the following can be concluded: Although the current implementation requires additional memory on the GPU and introduces an overhead in terms of arithmetic and memory access operations, performance rates similar to the fastest techniques so far can be achieved. A maximum throughput of 1.8 M tetrahedra/sec has been reported recently by Cahallan et al. [CICS05] on an ATI Radeon 9800. In comparison, our pipeline already achieves a peak rate that is over a factor of three higher. In particular, it can be seen that one of the drawbacks of slice-based techniques, i.e., multiple access to individual elements, can significantly be reduced due to the simultaneous evaluation of multiple sample points. It is clear, however, that in case of elements that intersect only very few slices, some of these evaluations might be unnecessary. For this reason, we have chosen data-dependent numbers of slices as shown in Table 3.2.

In Table 3.3, timing statistics for iso-surface rendering and cell projection are given. In particular, the iso-surface rendering shows excellent performance rates. Even the largest mesh consisting of 3.8 million tetrahedral elements can be rendered with 9 fps on commodity hardware. The simple active tetrahedra table stored on the CPU in combination with the ray-based iso-surface shader is superior to hierarchical data structures recently developed [WFKH07]. In particular, for the bluntfin data set about 17 ms have been reported on a 4-core Xeon architecture. In fact, our method runs three times faster on commodity hardware for this example.



Chapter 4

Collision Detection

While it is clear how to detect collisions between polygonal models under weak time constraints, there is an ongoing effort to develop techniques for interactive or even real-time applications. The difficulty arises from the fact that the size of dynamic 3D objects that can be rendered interactively has dramatically increased. Today, real-time raster systems can render moving objects composed of many millions of triangles at interactive rates. Such systems are used in many different areas of entertainment, industrial applications, and research. Moreover, as graphics capabilities become more advanced, the list of applications is growing rapidly. These applications impose significant performance requirements on the collision detection system, and they require algorithms and data structures to deal with hard time constraints.

Over the last years there is also a growing demand for interactive collision detection between objects that can deform, and can thus self-interfere. Typical applications include surgery simulators, cloth simulation, virtual sculpting, and free-form deformations. Interactive collision detection between deforming objects is complicated because it requires frequent updates of the data structures used to accelerate the detection process.

Even more important, geometric changes are increasingly performed on programmable graphics hardware using vertex programs and access to displacement textures [GFG04, SJP05, BK05, GBK05]. In this case, the changes in geometry might not even be known to the application program, which makes it difficult to maintain a data structure that appropriately represents the modified geometry. This problem is aggravated due to recent graphics hardware supporting geometry shaders. With Direct3D 10 compliant hardware [BG05] it is possible to create additional geometry on the graphics subsystem. For instance, triangle strips or fans composed of several vertices can be

spawned from one single vertex. By using this functionality the renderable representation cannot only be modified but it can be constructed in turn on the graphics chip.

The implications thereof with respect to collision detection are dramatic: As large parts of the geometry will continuously be modified and created on the GPU, CPU algorithms relying on the explicit knowledge of the object geometry can no longer be used. As a consequence, collision detection must either be performed entirely on the GPU, or the information required to perform the collision test on the CPU has to be created on the GPU and downloaded to the CPU.

In particular, this is true if high-resolution render surfaces are bound to the simulation meshes. As the deformed render geometry is kept in local GPU memory, the CPU cannot account for its collisions. On the other hand, keeping and updating a copy of the render surface on the CPU for collision purpose affects the overall time of the simulation system noticeably. Advanced hierarchical data structures have to be implemented to allow for efficient determination of colliding triangle pairs in this scenario.

Therefore, we propose a novel GPU-based collision detection pipeline that addresses the aforementioned issues. It neither requires a high-level data structure to be kept, nor does it need any assumptions about the movements or deformations of triangles that can occur between two consecutive time steps. Finally, due to this design the pipeline can even handle geometry that is created on the graphics subsystem. In the following, we describe the single steps of the pipeline in detail. These results have been published partially in [GKW07].

4.1 Related Work

Although a vast amount of literature has been published over the last decades, the efficient detection of collisions between large and dynamic polygonal models is still a fundamental problem in a number of different areas ranging from computer animation and geometric modeling to virtual engineering and robotics. For thorough surveys of the various species of collision detection algorithms let us refer here to the work by Lin et al. [LG98, LM04] and Teschner et al. [TKH+05].

According to the classification of collision detection algorithms into the three basic categories *static*, *pseudo-dynamic*, and *dynamic* [HKM95], our method belongs to the second class as it detects collision between moving objects at regular time intervals. While a large number of static and pseudo-dynamic techniques have been developed in the past, fewer approaches have explicitly addressed exact collision detection in time

and space [MC95, LSW99, RKC02, CS05].

Especially if a large number of objects have to be considered, collision detection algorithms usually proceed in a *broad phase* and a *narrow phase* [SKTK95, CLMP95, Hub95]. In the broad phase, approximate tests are performed to identify the potentially colliding objects out of the entire set of objects. This step effectively prunes the majority of all $O(n^2)$ possible object-object tests for n objects. In the narrow phase, further tests identify the object primitives causing interference. Usually this is done in a hierarchical manner by considering several levels of intersection testing between two objects at increasing accuracy.

Hierarchical methods are often based on bounding volumes and spatial decomposition techniques. Such methods enable the efficient localization of those areas where the actual collisions occur, thus reducing the number of primitive intersection tests [Möl97]. Over the last years, a number of different variants of hierarchical representations have been used, such as bounding sphere hierarchies [PG95, Hub96], axis aligned bounding boxes [vdB97, BFA02], oriented bounding boxes [GLM96] and k-DOPs (discrete oriented polytopes with k faces) [KHM$^+$98, VT00].

In the context of deformable objects, the emphasis has been placed on the efficient update of hierarchical object representations [DKT98, MKE03, JP04]. In particular, the idea to delay tree updates has been shown to be an efficient means to accelerate inter-object collision detection [LAM05]. A method for continuous collision detection between deformable meshes based on graph theory has been suggested [GKJ$^+$05]. At the core of this method is the partitioning of polygonal meshes into sets of independent primitives in which collisions can be detected in linear time. To avoid the time-consuming pre-processing of the previous approach, a method based on Voronoi diagrams has been presented recently by the same group [SGG$^+$06]. The properties of Voronoi diagrams are utilized to determine the closest primitives in a given set of primitives. A GPU collision detection algorithm for deforming NURBS objects has been developed recently in [GGK06]. This technique is based on a hierarchical AABB traversal scheme entirely implemented on the GPU.

The algorithm we propose in this work originates in the early idea of using rasterization hardware for interference detection between polygonal objects [RMS92, MOK95]. The underlying theoretical basis is given by Jordan's theorem on the separation of a plane by a closed curve in this plane. The same idea is employed in the shadow volume algorithm proposed by Crow [Cro77] to detect whether a point lies inside or outside of a polygonal shadow object. By using depth and stencil buffer hardware, the number of points being in front of a surface point can efficiently be counted and used for

classification. Building on this theory, methods based on voxels [BW02, HTG04] and view-frusta [LCN99] have been proposed.

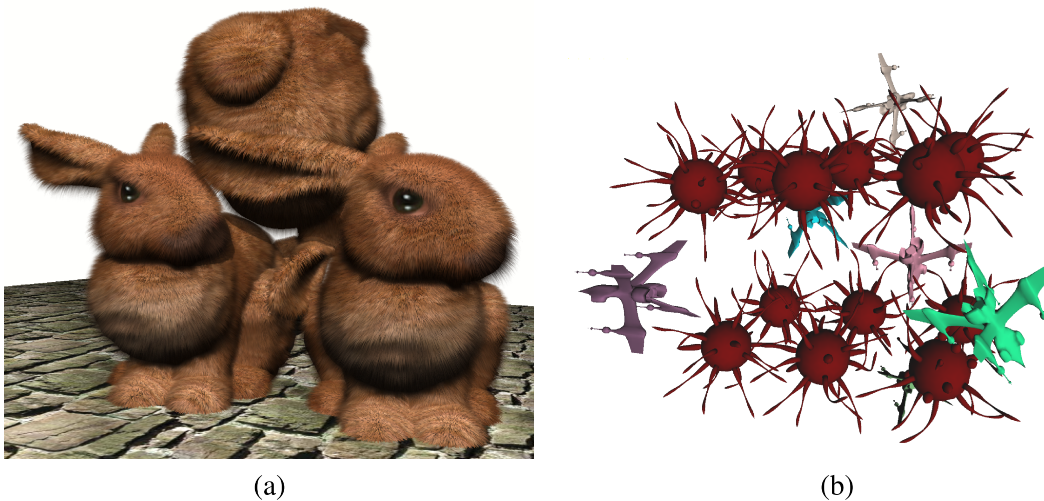
In comparison to previous approaches, our method is similar to suggestions made by Knott and Pai [KP03]. By observing that scan-conversion algorithms only count the number of intersections between a polygon P and the view ray up to the first point q of the penetrating object, a variant that also detects additional points along these rays was proposed. This is achieved by rendering the penetrating object as wire-frame such that lines not entirely occluded by others “shine through” and leave a unique ID in the frame buffer. As the method distinguishes between penetrating and penetrated objects, it cannot handle self-intersections of one single deformable object.

Along a different line of research, hardware-supported occlusion queries have been employed to accelerate collision detection [GRLM03]. Via an occlusion query the application program can request the number of fragments that survive the depth test in the rendering of a set of primitives. These queries can thus be used to accelerate the broad phase of collision detection, i.e., by testing whether two objects can be trivially rejected because they do not interfere in screen-space. To overcome sampling and precision errors that can result in collisions being missed, Govindaraju et al. [GLM04] proposed to “fatten” the objects, i.e., to extend the screen-space footprint of rendered primitives both with respect to the number of covered fragments and the primitives depth. The use of occlusion queries for intra-object collision detection including strategies to reduce the number of potentially colliding primitive pairs was proposed in [GLM05].

Occlusion queries have been shown to be a very powerful means to reduce the number of potentially colliding primitive pairs. The reason why we decided not to use occlusion queries is threefold: First, as occlusion queries have to be issued by the application program, this mechanism seems to be problematic for the handling of collisions between GPU objects (objects that are only known on the graphics subsystem). Second, in the context of collision detection, occlusion queries work most effectively if used in combination with a narrow-phase acceleration strategy on the CPU. This implies a dynamic data structure to be kept on the CPU, which is difficult to maintain efficiently in case of deformable objects or even GPU objects. Third, the effectiveness in pruning a majority of intersection tests strongly depends on the camera position and viewport. Thus, in general several views of the scene have to be rendered to prune a considerable number of primitives.

4.2 Contribution

In this chapter, we present a novel collision detection algorithm for closed manifold meshes that addresses the aforementioned issues. We also show how this algorithm extends to the detection of collisions between arbitrary, open 2D-manifolds. The method is specifically designed for interactive handling of deformable objects and GPU objects, i.e., polygonal objects that are modified or constructed on the GPU. Figure 4.1 shows some examples.



(a) (b)

Figure 4.1: *We present a method for interactive collision detection between polygonal objects. This method can handle deformable objects (a) and objects that are constructed on the GPU (b). For the latter scene consisting of 320k moving triangles the time spent for collision detection using our approach is less than 25 milliseconds.*

Our algorithm is developed in consideration of the observation that the Achilles' heel of almost all collision detection algorithms for deformable objects is the dynamic data structure used to represent the changing object geometry. We avoid the construction and repetitive update of such a data structure by shifting parts of the collision detection algorithm onto the GPU. Our algorithm takes as input a renderable polygonal object representation, and it only requires a GPU array containing the polygons to be rendered. In particular, this allows the method to handle geometry that is arbitrarily modified or created on the GPU. The proposed method proceeds in five passes:

1. **Object sampling:** Colliding objects are sampled along a set of rays via depth-peeling, and all rays along which a collision occurs are detected. This is done by exploiting the intrinsic strength of recent GPUs to interactively render high-resolution polygonal meshes and to efficiently perform fragment operations.

2. **Ray merging:** On the GPU a mipmap texture containing screen-space bounding boxes of ray-bundles at an ever increasing width is built.

3. **Primitive separation:** Primitives access the mipmap to test whether they are potentially colliding or not. For each primitive, potentially colliding areas are encoded as screen-space bounding boxes in a texture map. From this information, the set of potentially interfering primitives can be computed efficiently. In particular, due to this pass, the precision of interference computations is not constrained by the resolution of the frame buffer we rendered to in the first pass.

4. **Texture packing:** The results generated in the previous pass are transferred to the CPU. To keep bandwidth requirements and CPU processing as low as possible the sparse texture map is first converted into a packed representation.

5. **Intersection testing:** Exact intersection testing is performed on the CPU. Per-primitive screen-space bounding boxes that have been computed in pass three are used to prune most of the remaining primitive pair tests.

A diagrammatic overview of the collision detection process as suggested in this work is shown in Figure 4.2. It illustrates how the proposed method fits the GPU stream architecture. It is designed as a pipeline of stages being successively applied to the stream of model geometry and generated fragments. In contrast to previous collision detection algorithms, it is a very unique feature of the proposed method that it can handle geometry that is arbitrarily modified or even created on the GPU. The input for the algorithm is a geometry stream as generated by a geometry transformation unit, i.e., the vertex shader or the geometry shader on very recent graphics hardware. The result is a fragment stream consisting of only the potentially colliding primitives, which is written into a geometry texture.

The geometry texture is transferred to the CPU for exact intersection testing. Although it is in general possible to compute polygon-polygon intersections on the GPU, its data-parallel nature is not very well suited for the kind of operations required to determine the colliding partners from the entire set of possible candidates. Therefore, a CPU-GPU hybrid method is employed, in which the CPU is responsible for exact intersection testing.

For many scenarios, the data transfer from the GPU to the CPU is most likely to become the bottleneck in the entire collision detection system. Thus, we present a novel GPU technique to convert a sparse texture into a packed texture of reduced size. The proposed technique converts an input stream containing a few randomly scattered valid

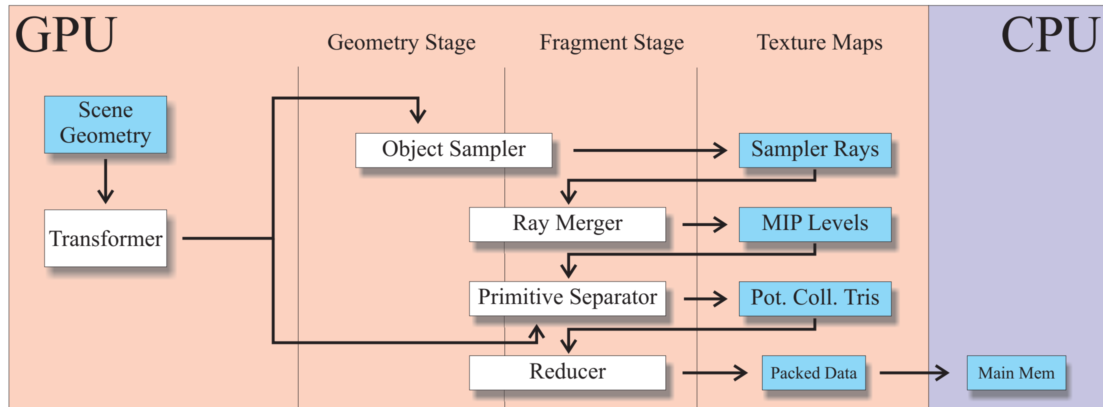


Figure 4.2: *A diagrammatic overview of the proposed collision detection algorithm.*

data items into a stream consisting of only these items. As this stream is downloaded to the CPU, bandwidth requirements are considerably reduced. Since the approach can also be used to filter out fragments not going to participate in upcoming rendering passes, it has a number of possible applications. At the end of Section 4.4.1, we will outline some of them.

The remainder of this chapter is organized as follows. The infrastructure on the GPU that is required to implement the collision detection pipeline is presented in Section 4.3. First we describe how recent GPUs are used to determine the rays along which potentially colliding polygons are hit. Furthermore, the use of parallel fragment units for mipmap generation and intersection testing is outlined. Section 4.4 illustrates the reduction technique for sparse textures on GPUs, and we then describe how intersections are finally determined on the CPU. Section 4.5 briefly describes the collision response method applied in our test scenarios. We lastly analyze the properties of our algorithm in a number of different scenarios and we present our results including the extension to non-closed manifolds.

4.3 Screenspace-Accurate Object Intersection

4.3.1 Object Sampling

In the first pass of the proposed collision detection algorithm the polygonal scene is sampled to detect rays along which at least one potentially colliding polygon is hit. These rays will subsequently be called *collision rays*. Here we are searching for rays that have either at least two consecutive hits with a front facing polygon or at least two consecutive hits with a back facing polygon, or that first hit a back facing polygon.

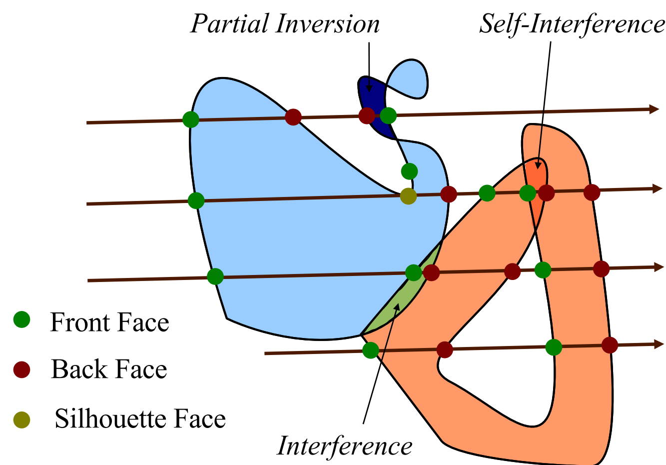


Figure 4.3: *Illustration of interference, self-interference, and partial inversion. All cases result in two consecutive front or back facing polygons, or a back facing polygon as first hit along a ray.*

The underlying theoretical basis of this method is given by the generalization of Jordan's theorem to higher dimensions. A closed polyhedron P separates space into an "inside" and an "outside." If it has outfacing normals, any ray starting outside of P and intersecting P has alternating front and back facing intersection points (silhouette points are ignored). At the front facing intersection points the ray enters P and at the back facing intersection points it is leaving P. If along a ray two consecutive front or back facing intersection points are found, the ray enters a second object at the second front facing point before the first object was left, or it leaves an object but was still in another object [1]. This also holds for an arbitrary number of objects. Examples of (self-) interference are illustrated in Figure 4.3.

To detect intersecting closed polyhedra, it is therefore sufficient to detect consecutive front facing polygons or consecutive back facing polygons (silhouette polygons are ignored), or a back facing polygon as first hit along any ray starting outside the potentially colliding set. To detect all collisions, the space in which the polygons exist has to be sampled as densely as possible.

Implementation

Although the rays being used to sample the objects' faces can be chosen arbitrarily, a uniform sampling along parallel rays leads to the most isotropic sampling in object space. Scanline rendering algorithms simulate this by projecting the objects along an

[1] Here we assume that no object is entirely contained in any other object.

arbitrary but constant direction. To detect consecutive pairs of front or back faces, all faces have to be rendered in correct visibility order with respect to an infinite viewer in the direction of the projection. Depth-peeling is employed to achieve this ordering.

The use of depth-peeling to track intersection events in an ordered way was suggested before by Guha et al. [GKMV03] and Hable and Rossignac [HR05] in the context of GPU-based CSG[2] rendering. By tracking the state of intersection points between view rays and bounded solids, i.e., entering (1) or leaving (0), the state of a boolean expression of these events can be determined correctly at any point along the rays. We are interested in extending this idea to find consecutive events of the same state.

Depth-peeling requires multiple rendering passes. For each pixel, in the n-th pass the $(n-1)$-th nearest fragments are rejected in a fragment program and the closest of all remaining fragments is retained by the standard depth test. A floating-point texture map—the depth map—is used to communicate the depth of the surviving fragments to the next pass. A detailed description of this method is given by Everitt [Eve01]. The number of rendering passes is equal to the objects' depth complexity, i.e., the maximum number of object points being projected to a single pixel. The depth complexity can be determined by rendering the objects once and by counting the number of fragments being projected to each pixel during scan-conversion. The maximum coverage of all pixels is then collected in a log-step reduce-max operation [KW03b].

To detect two front or two back facing polygons in consecutive rendering passes it is sufficient to store an additional tag indicating the expected facing of the fragment in each entry of the depth map. The expected facing is determined by alternating between front and back facing states, initially starting with a front facing state. In addition to only comparing the current depth of the fragment to the value stored in the depth map, a fragment shader also compares the expected facing to the actual one. If they differ, the fragment is marked as a collision ray and is discarded in upcoming rendering passes.

The modified depth-peeling technique generates a texture map—the sampler ray—in which the status is set to "on" for all collision rays, and to "off" for all other rays (see Figure 4.4). As the sampling rate is constrained by the resolution of the frame buffer, some interfering primitives, and thus collision rays, might not be detected. This problem can be weakened by increasing the resolution to its maximum size, but it still exists. On the other hand, a collision ray is only missed if all interfering primitives along that ray are missed. If at least one of the intersections is detected, the upcoming stage of the collision detection process will find *all* intersecting primitives.

[2] Constructive Solid Geometry

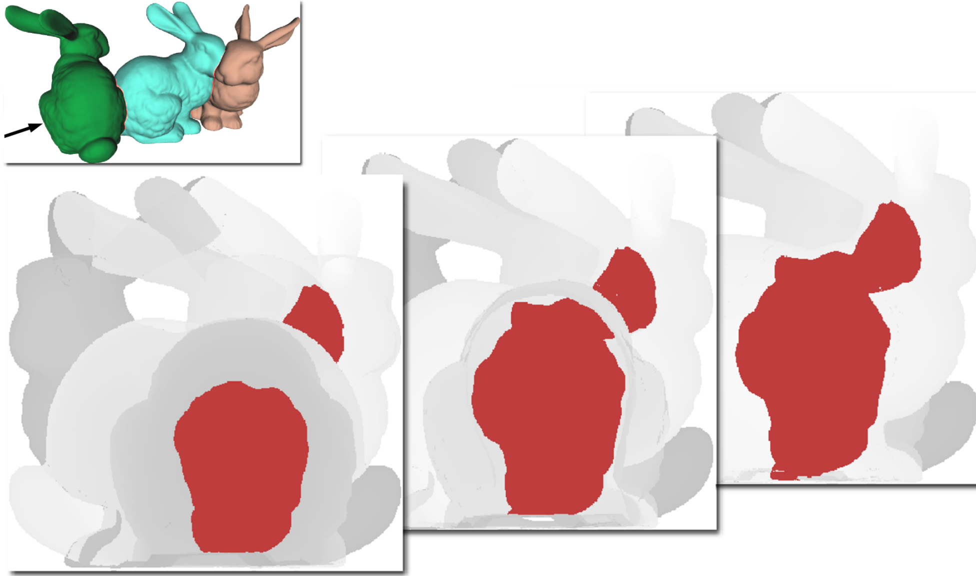


Figure 4.4: *By using depth-peeling and fragment operations to detect front or back faces, collision rays in each layer are determined (colored red). Three layers of the scene consisting of bunny models are shown here.*

4.3.2 Ray Merging

To determine potentially colliding primitives, i.e., polygons that are hit by a collision ray, the information that was generated on a per-ray basis in screen-space has to be carried over to the set of polygons. One possibility is to rasterize one fragment for each polygon and to compute the rays intersecting the polygon in a fragment shader. The status of each ray can be retrieved from the sampler ray texture, and the primitive is assigned a flag indicating whether it is hit by at least one collision ray or not. Unfortunately, from this information alone it is quite cumbersome to determine the set of potentially colliding primitive pairs being needed for exact intersection testing. Moreover, due to the different size of triangles in screenspace, the GPU's fragment processing cannot reach its maximum performance.

Therefore, we propose a more efficient strategy. At the core of this strategy is the idea to generate the information required to efficiently determine the set of potentially colliding primitives for each polygon. This inter-object relation is established via the collision rays. In general, every polygon is hit by many collision rays. Thus, for a particular polygon several of these rays are required to detect all potentially colliding partners. In the following we describe a simple GPU data structure in which the relations between a primitive and all of its potential partners are encoded in one single ray

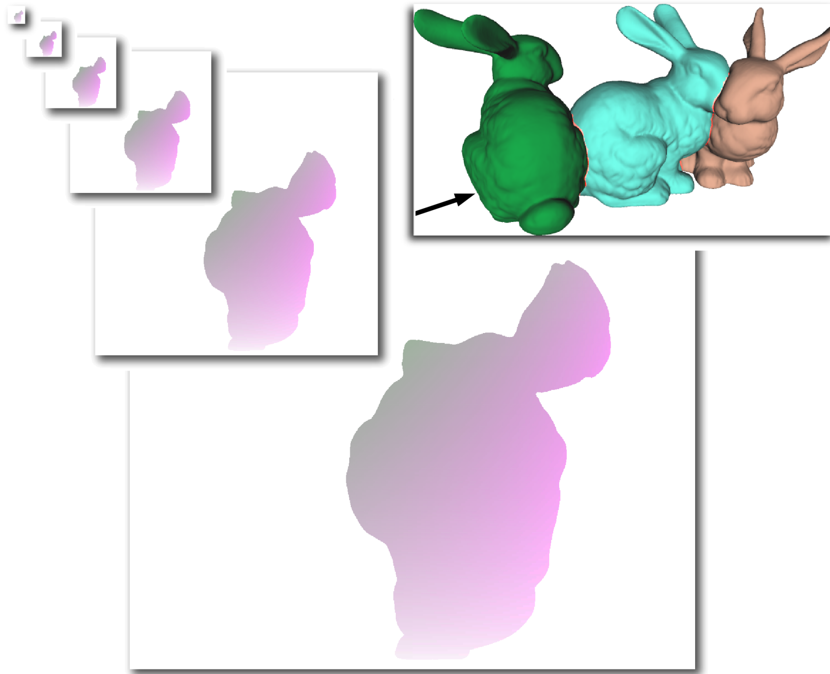


Figure 4.5: *A mipmap texture represents bundles of collision rays at increasing width.*

bundle. An example of the resulting mipmap texture is given in Figure 4.5.

The data structure consists of ray bundles at increasing width. For every collision ray, the screen-space bounding box $bb = (x^<, y^<, x^>, y^>)$ of the pixel this ray is passing through is computed. The first two and last two components of the quadruple specify the left-bottom and the top-right corner of this bounding box. Bounding boxes of sampler rays that are "off" are set to $(1, 1, 0, 0)$, such that they do not affect the union with any other box. Bounding boxes are rendered into an RGBA 16 bit floating-point texture of the same size as the sampler ray texture. From this texture a mipmap hierarchy is generated by computing at each level l the union of bounding boxes of the 2×2 corresponding texels at level $l - 1$. The union of two bounding boxes bb_1 and bb_2 is calculated as

$$bb_1 \cup bb_2 = (\min(x_1^<, x_2^<), \min(y_1^<, y_2^<), \max(x_1^>, x_2^>), \max(y_1^>, y_2^>)) .$$

This process is performed recursively until only one bounding box is left over (see Figure 4.6).

For the sake of simplicity the initial texture size is assumed to be a power of two, and we limit ourselves to quadratic textures. The mipmap finally stores screen-space

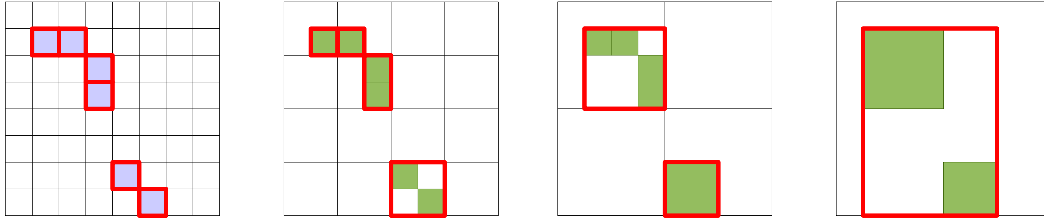


Figure 4.6: *Mipmap construction: On the finest level, the screen-space bounding boxes (red) are set to the pixel border of each collision ray (blue), or the texels are empty otherwise. For* 2×2 *texels the union of their boxes (green) is calculated to obtain a bounding box (red) at the next coarser level.*

aligned bounding boxes of ray bundles, where a bundle only contains those rays that have been marked as "on." This mipmap can be used to find all potentially collision rays of a particular primitive as explained next.

4.3.3 Primitive Separation

To find the ray bundle that contains all colliding rays intersecting a certain primitive, we first compute the minimum mipmap level where the screen-space extent of one texel is larger than the screen-space bounding box of the primitive itself (see Figure 4.7).

To efficiently find this level, we generate one fragment for each polygon and com-

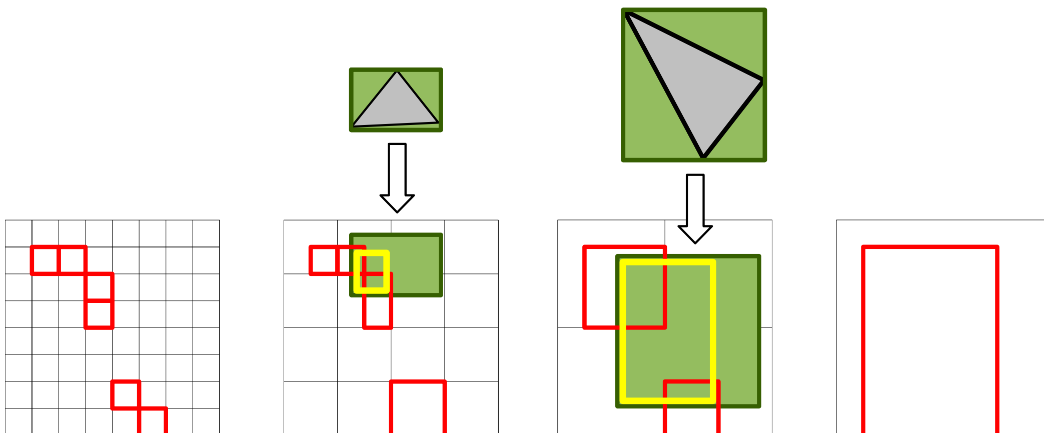


Figure 4.7: *Primitive separation: The screen-space bounding box of two triangles is shown in green. The maximum extent of these boxes yields the mipmap level. The mipmap is sampled (nearest neighbor filtering) at every corner of the box. Since the texels of the corners are still neighboring at the next finer mipmap level, we can use this level instead. The bounding boxes stored at each sample point (red) are combined to a single box. The output is the intersection between this box and the triangle bounding box (yellow).*

pute the primitive's screen-space bounding box in a fragment shader. From the extent of this box the appropriate mipmap level is derived. As a primitive can intersect multiple ray bundles at this level, at every corner of the primitive's screen-space bounding box, one bundle along with its screen-space bounding box is fetched from the mipmap hierarchy. (If the corresponding texels are still neighboring at the next finer mipmap level, this level can be used instead.) The union of these boxes is then determined, and the resulting bounding box is intersected with the screen-space bounding box of the triangle. The coordinates of this box are stored in a target texture. If only empty ray bundles are fetched from the mipmap, the respective entry in the render target is set to zero, resulting in a texture that is sparsely filled. An overview of the ray merging and primitive separation step is also given in Figure 4.8.

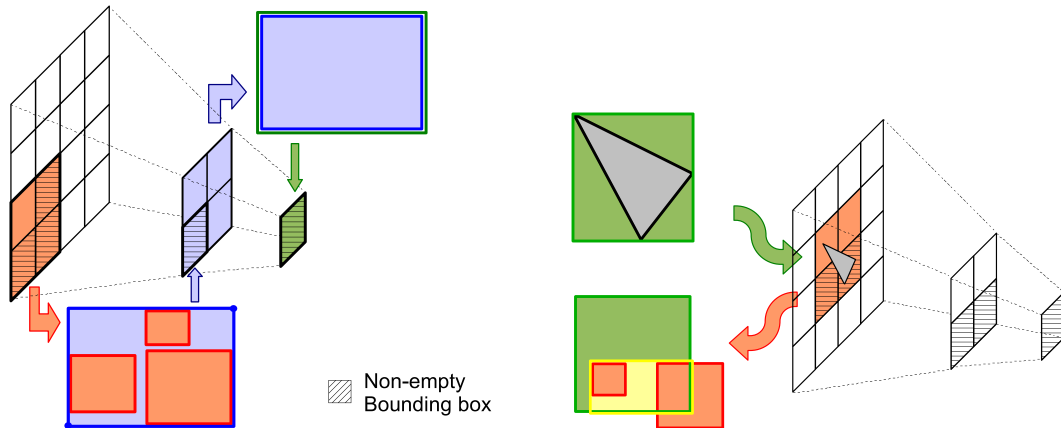


Figure 4.8: *Ray merging and primitive separation.*

4.4 GPU-CPU Data Transfer

The texture that is generated in the previous pass has to be transferred to the CPU for exact intersection testing. Although it is possible to compute polygon-polygon intersections on the GPU, the GPU's data parallel nature is not very well suited for the kind of operations required to determine the colliding partners. This can be performed most efficiently by using a sweep-and-prune strategy which reduces the set of potential partners by considering the bounding boxes of the polygons along the screen-space $x-$, $y-$, and $z-$direction. The implementation of such a strategy on the GPU does not seem to be very promising and has not been considered in this work.

By applying the technique described in the next section, a packed texture containing

the set of potentially colliding triangles is generated on the GPU. Each texel stores a unique triangle ID as well as the screen-space bounding box of the set of rays intersecting this triangle. The packed texture is finally transferred to the CPU for intersection testing.

4.4.1 Texture Packing

This stage implements a general method to convert a sparse texture into a packed texture that consists of only the non-empty texels in the sparse texture. The proposed method significantly differs from the one presented in [GGK06] since it does not rely on a global scattering pass and the sequence of operations is not data-dependent.

The reduction stage—although it is not a mandatory stage in the proposed collision detection algorithm—is essential for our technique to perform most efficiently due to the following reasons: First, the packed texture can be downloaded to the CPU at much higher rates. Second, the processing of a large number of empty cells can be avoided on the CPU. The texture reduce operation on the GPU is accomplished in three stages:

- **Counting:** Non-empty texels per row are counted in a log-step reduce-add operation along texture rows. A single-column texture storing these counts is read to the CPU (see Figure 4.9).
- **Shifting:** In each row non-empty texels are shifted to the right of the texture (see Figure 4.10). All rows are processed in parallel by rendering a vertical line.
- **Moving:** Packed rows are moved into the reduced texture. This texture is finally downloaded to the CPU (see Figure 4.11).

The application program computes the total number of non-empty texels from the single-column texture being read in the first step. This information is used to set the size of the reduced target texture. In addition, the maximum number M of non-empty texels per row is computed.

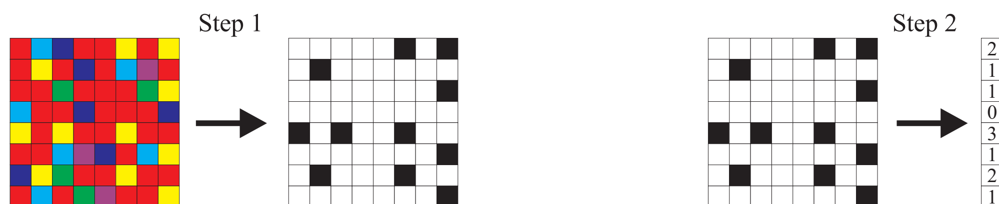


Figure 4.9: *Counting: Step 1 executes a classification shader (selects only yellow texels here). Step 2 counts the positive texels per line and reads them back to the CPU.*

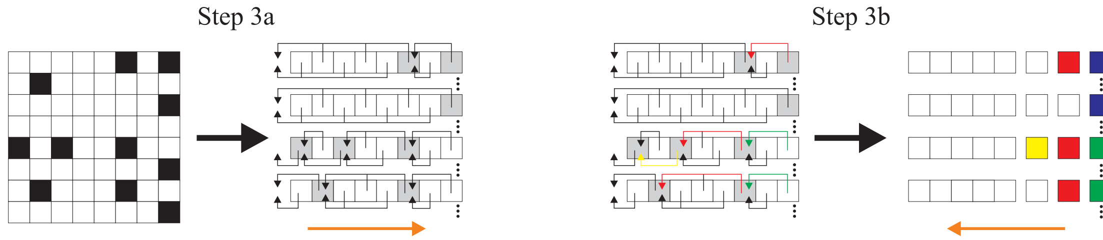


Figure 4.10: *Shifting: First, a next-pointer list is constructed. Second, this list is traversed to gather texels from the appropriate position. The orange arrow depicts the traversal direction. For the sake of clarity only the shifting of every second line is shown.*

In the second step the sparse texture is reduced horizontally. This is done in two passes, the first of which proceeds from left to right and the second from right to left (see Figure 4.10). To do such a reduction on the CPU we would simply traverse each row from left to right, keeping a pointer to the current element in the reduced row and copying the next non-empty element to this position. Unfortunately such a copying (or scattering) operation is not available on recent GPUs. Thus, we have to convert this operation into a gather operation. Therefore, we first sweep over the texture from left to right and store into each texel the position of the preceding non-empty texel in the same row. Before the first non-empty entry is encountered, a special key is stored. In the second pass we sweep from right to left for M steps. If the rightmost texel in a row is not empty we write zero into the texture, otherwise the address found in that texel is written (Figure 4.10, blue texels). In all subsequent sweeps the address at the preceding position in the same row is dereferenced first, and the retrieved address is written to the render target. Sweeping is accomplished by rendering a vertical line primitive covering as many pixels as there are rows in the sparse texture. As we only need to access a single address per line, which can be stored in one component of an RGBA color value, with every rendered line four columns can be processed at once.

After the horizontal reduction, the contiguous sets of texels in each row of the sparse texture have to be copied into a render target of reduced size. This is done by rendering for each row in the sparse texture a horizontal line covering as many pixels as there are non-empty texels in this row. If the size of the packed texture is $X \times Y$ and rendering the first L lines has generated N fragments, then the $(L+1)$-th line starts at position $(N \bmod X, \lfloor N/X \rfloor)$ in the target texture. If the length of this line is larger than $X - (N \bmod X)$ it has to be split into two or more lines of reduced length. This is illustrated in Figure 4.11. The fragments generated for each line can fetch the corresponding values from the sparse texture via appropriately chosen texture coordinates. The read values are copied to the render target as shown in Figure 4.11. Due to perfor-

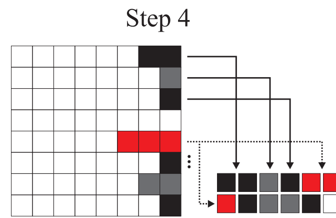


Figure 4.11: *Moving: Each packed row is finally copied at the appropriate position into the target texture.*

mance issues, the application program creates a vertex array containing all the required information and renders this array using one single call.

Applications

The need for a texture reduction scheme on the GPU is paramount in a number of graphics applications, where data is modified or generated on the GPU and has to be processed further on the CPU. Popular examples include physics on GPUs [GH06], where the results of approximate occlusion queries and collision response calculations are used in the broad phase of collision detection on the CPU.

The texture reduction scheme provides a powerful mechanism to minimize the bandwidth requirements in the proposed collision detection algorithm. On the other hand it can also be used to discard fragments not going to participate in upcoming rendering passes on the GPU. In typical fragment-based rendering algorithms it is often the case that the fragment program is conditionally executed by only a small fraction of the entire set of fragments generated by the rasterizer. Potential applications include adaptive techniques for texture filtering, rendering, or numerical simulation on the GPU. A similar texture reduction technique based on histogram pyramids has been presented recently by Ziegler et al. [ZTTS06]. In particular, they use their technique to generate point clouds of arbitrary geometries for particle explosions.

As an alternative to the proposed texture reduction technique, fragments can also be discarded using advanced GPU features like the early-z test or breaks in the fragment stage. Unfortunately, on current graphics cards early-out mechanisms introduce some overhead, i.e., either a branch instruction or additional rendering passes. Moreover, since the pixel shader hardware runs in lock-step, a performance gain can only be achieved if all fragments in a contiguous array exit the program early. These observations are backed up by latest GPUBench [BFH04] results. Results for our target architecture, the GeForce 7800 GTX, are available at `http://graphics.stanford.edu/`

projects/gpubench/results/. These results attest current GPUs a rather bad branching performance even if all fragments in a 4×4 block exit the program simultaneously.

As a matter of fact we believe that the proposed texture packing has the potential to become a general means to efficiently filter out fragments from a generated fragment stream. Thus, the method is not only highly beneficial in hybrid CPU-GPU approaches to reduce bandwidth requirements but also in pure GPU techniques to minimize the load in both the rasterization and the fragment stage.

4.4.2 Intersection Tests

After having received the texture containing all potentially colliding primitives, a sweep-and-prune strategy [CLMP95] is utilized on the CPU to determine the colliding primitive pairs. These are the primitives whose bounding boxes overlap along each of the three screen-space axes. The extent of the bounding boxes in screen-space z-direction is only computed if an overlap along the x- and y-direction has been detected. For those primitives being detected a triangle-triangle intersection test is performed [Möl97], and for each triangle a response vector is computed taking into account its own normal and the normal of the colliding partner. Alternatively, any other CPU collision algorithm based on axis-aligned bounding boxes can be used instead of the sweep-and-prune strategy.

4.5 Collision Response

In this work, we only consider simple approaches to perform collision response. For rigid bodies, the response vectors of all triangles belonging to a body are averaged and used to compute the changes in the position and rotation of the body. For deformable objects, per triangle response vectors are used to compute appropriate repulsion forces. Given two colliding triangles, forces are calculated to push the triangles away from each other as shown in Figure 4.12. Forces are only assigned to penetrating vertices, and the repulsion forces are accumulated to account for all incident triangles.

It is worth noting here that more accurate and physically motivated techniques to compute the collision response can be integrated into our approach but have not been considered in this work. In particular, to accurately resolve collisions the simulation step has to be stalled until all collisions are resolved, and this, in general, implies that the per-frame computational time increases significantly. In real-time applications, the use of repulsion forces has a beneficial property. Since the collision response is handled

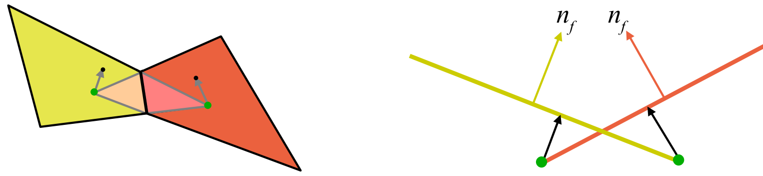


Figure 4.12: *Calculation of repulsion forces: Penetrating vertices (green) are pushed back in the direction of the face normal of the other triangle.*

via external forces, the per frame computational load is nearly constant. On the other hand, repulsion forces cannot guarantee to resolve the collision within the next time step (when the effect of the forces is determined), and objects can penetrate each other.

4.6 Results

We have tested the proposed collision detection algorithm in three different scenarios consisting of several thousands up to a million triangles. All of our tests were run on an Intel Core™ 2 Duo equipped with an NVIDIA GeForce 7800 GTX. In all of our tests a 1k × 1k frame buffer was used to sample the objects along parallel view rays. All objects are encoded as indexed vertex arrays stored in GPU memory. The timings given below do not include the amount of time required for collision response calculations.

With the help of three test scenes the overall performance of our algorithm is demonstrated in the next section, prior to a more detailed analysis. We verified that all intersecting primitive pairs at one time step were detected in the presented examples. However, because our algorithm belongs to the class of pseudo-dynamic collision detection algorithms, some collisions might be missed due to the movement of objects.

4.6.1 Scenes

Deformable object collisions: In the first example the collision detection algorithm was integrated into our interactive deformable bodies simulation. Collision detection was performed on the render surface, which is bound to the tetrahedral simulation mesh as described in Section 3.3. To perform the triangle intersection tests on the CPU, an undeformed copy of the render surface geometry is stored. For the triangles to be taken into account in the final step of the algorithm, the CPU copy of the triangle is first displaced by interpolating the simulation displacement field for each vertex using barycentric weights. The calculated repulsion forces are distributed to the simulation vertices using the same weights. In Figure 4.13 two snapshots of an animation sequence

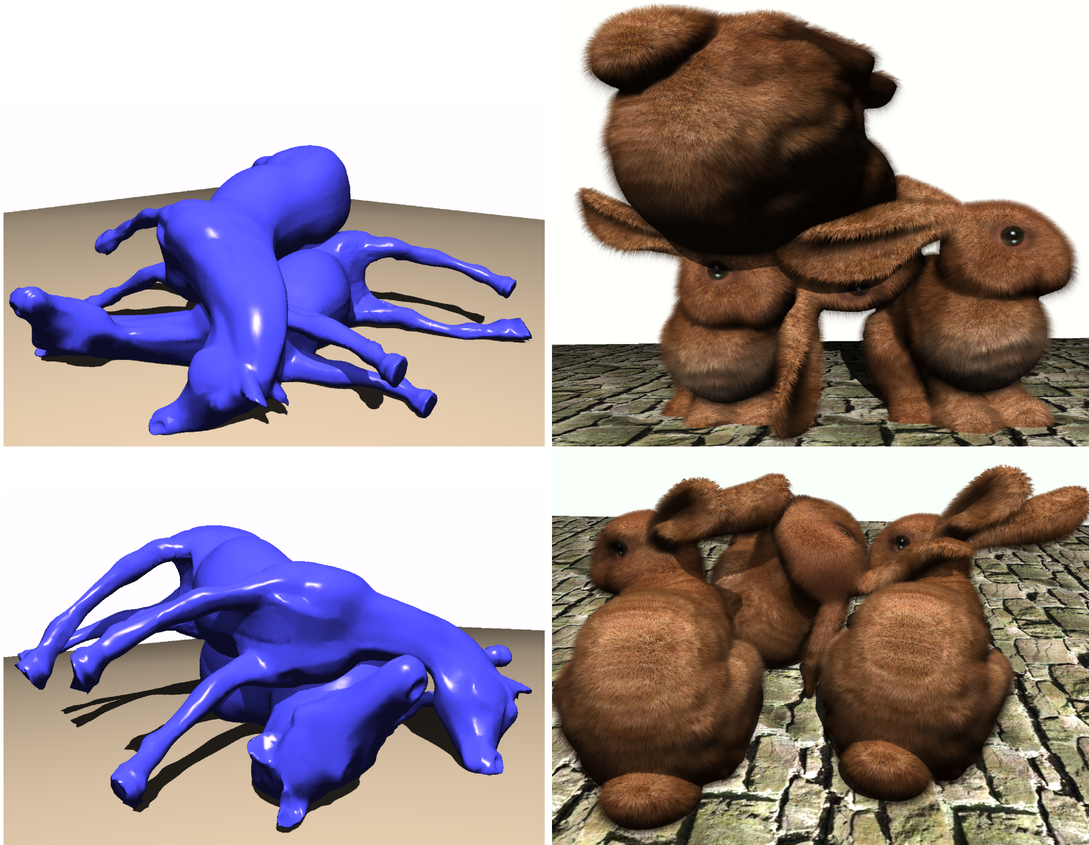


Figure 4.13: *(Self-) collisions between deformable objects.*

with bunny and horse models are shown. The depth complexity of the scenes along the collision rays is 8. All collisions between the deforming objects were detected in less than 40 ms.

GPU object collisions: To further demonstrate the ability of the proposed algorithm to deal with geometry that is modified or even created on the GPU, we have used a scene that is made of dynamic meshes being procedurally deformed on the GPU. Figure 4.14 shows this scene consisting of artificial creatures made of a spherical body and a number of moving tentacles attached to it. Tentacles are animated and deformed by a vertex shader program on the GPU. Rigid starships try to pass these creatures. Upon collision with any other creature or any of the shuttles, tentacles retract and start growing again.

In contrast to all other examples, for every potentially colliding triangle that is detected on the GPU the three vertex coordinates of this triangle along with its normal and a unique tentacle ID are stored in different texture maps. These textures are then packed and downloaded to the CPU. In this way the CPU can control the retraction

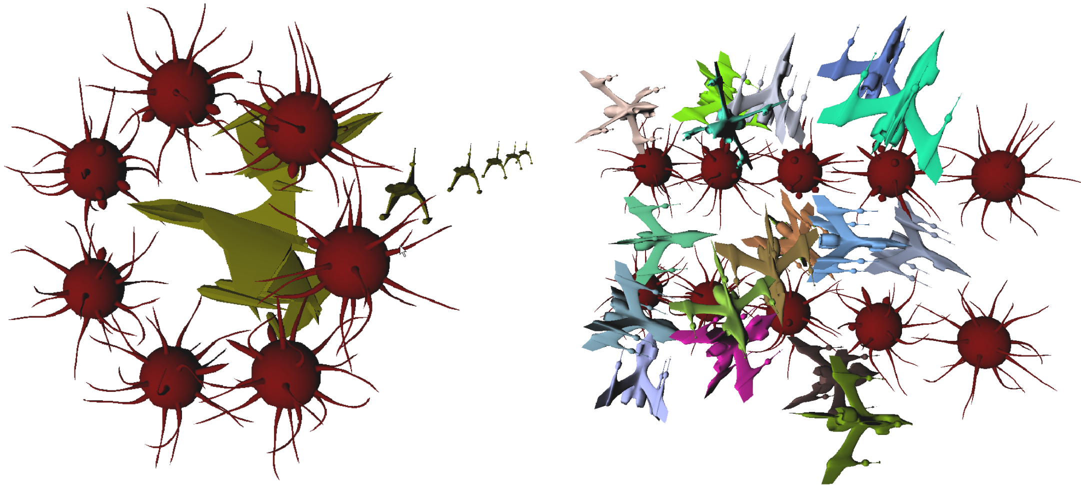
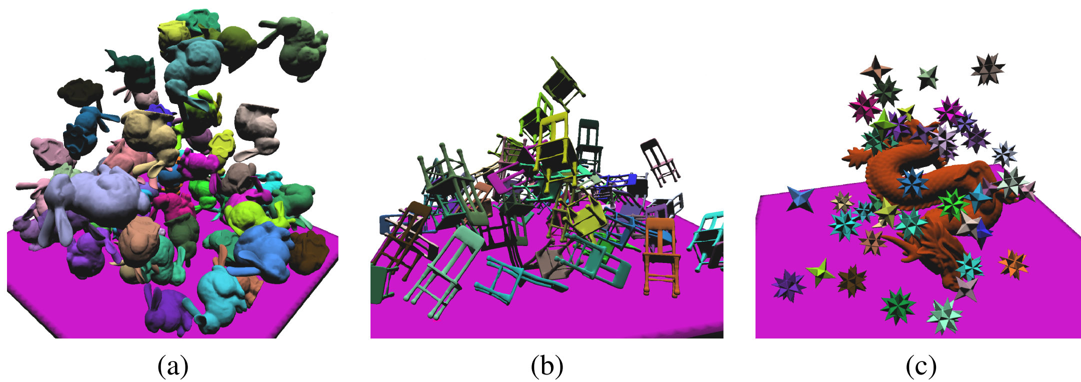


Figure 4.14: *Collisions between dynamic GPU objects.*



(a) (b) (c)

Figure 4.15: *Rigid body collisions.*

of tentacles via shader parameters, and it can simulate rigid body motion of the shuttles. The overall scene consists of 320k triangles. All collisions were detected in about 25 ms.

Rigid body collisions: Our last example demonstrates the capability of our method to handle collisions between rigid bodies (see Figure 4.15). Although we are aware of the fact that optimized CPU collision detection algorithms are probably more suited to this particular application, the given examples allow for a clear analysis of the different parts of our method.

The first scene in Figure 4.15 (a) consists of 60 rigid bunnies moving through space due to gravity and collisions. The entire scene consists of half a million triangles. The depth complexity of the scene as seen in the image is 16. The detection of all collisions in the scene took 200 ms. We have also run the experiment by restricting the number

of rendering passes performed by the depth-peeling routine to six. In this case, the performance increased to 120 ms, and even more interestingly only less than 5% of all penetrating triangle-triangle pairs were missed. Alternatively, we have stopped depth-peeling after seven passes and all fragments with a depth count greater than 6 were marked as potentially colliding. In this case the performance dropped down slightly, but on the other hand all triangle-triangle collisions were again resolved. The efficiency of our approach is further demonstrates by the example in Figure 4.15 (c). Even for a triangle count of 800k in the dragon scene and a number of 50k potentially colliding primitives processed on the CPU, the method is still able to achieve about 4 frames per second.

4.6.2 Analysis

Detailed statistics of all test scenes are presented in Table 4.1. The overall triangle count is given in the first column. The second column presents the number of collision rays. The number of triangles which are downloaded to the CPU is given next. In the fourth column the number of triangles that survive the CPU pruning of overlapping pairs of primitive bounding boxes is listed. The fifth column contains for each scene the number of primitive intersection tests. The last column shows the number of detected colliding triangle pairs.

Scene	# triangles	# collision rays	# triangles downloaded	# pot. coll. triangles	# tri-tri tests	# positive tests
Defo bunnies	12k	0.2k	0.4k	0.2k	0.4k	0.1k
Defo horses	32k	21k	13k	4.3k	9.0k	3.7k
Art. creatures	320k	0.3k	1.0k	0.4k	1.5k	0.2k
Rigid bunnies	500k	2.8k	12k	4.4k	7.7k	1.5k
Rigid dragon	800k	7.7k	54k	7.5k	6.6k	1.5k

Table 4.1: *Triangle counts of the collision scenes.*

Representative timings for collision detection in the example scenes are listed in Table 4.2. All timings are given in milliseconds. The first column shows the amount of time spent by the GPU for *object sampling*, *ray merging*, and *primitive separation*. The time required for reducing the sparse texture and downloading the packed texture to the CPU is given next, followed by the time required for CPU processing. Finally, the overall performance is specified in frames per second (fps).

Scene	GPU	Reduce	CPU	overall
Defo bunnies	3 ms	1 ms	1 ms	5 ms / 200 fps
Defo horses	5 ms	4 ms	24 ms	33 ms / 30 fps
Art. creatures	17 ms	6 ms	1 ms	24 ms / 42 fps
Rigid bunnies	56 ms	18 ms	21 ms	95 ms / 11 fps
Rigid dragon	53 ms	32 ms	139 ms	224 ms / 4.4 fps

Table 4.2: *Timing statistics for collision detection (NVIDIA 7800 GTX).*

In comparison to previous work on GPU-based collision detection between CPU objects, the examples we have shown indicate comparable performance. It can be observed, however, that our algorithm has the tendency to generate a larger set of potentially colliding primitives on the GPU, which then has to be processed on the CPU. Due to the fact that we sample the scene along parallel rays through each pixel, we might also miss collisions if the size of polygons is below the pixel size. On the other hand, such inaccuracies can only occur if all interfering primitives under one pixel are missed. If at least one of them is detected all intersecting primitives will be detected in the upcoming stage of the collision detection process. By “fattening” the objects similar to the method proposed in [GLM04] this problem can be solved entirely, but it comes at the expense of an increasing number of false positives.

An advantageous feature of our method is that only a very simple data structure is required to determine potentially colliding primitive pairs. Besides GPU-friendly geometry representations that are used for depth-peeling, such as vertex arrays or display lists, an indexed face set and a shared vertex array is needed. Updates of the geometry only require the vertex array to be updated accordingly.

As can be seen in the rigid bunny scene, depth-peeling can become the performance bottleneck if the depth complexity of the scene is increasing. On the other hand, this limitation only becomes severe if very large objects with high depth complexity have to be tested, because we can use GPU-resident geometry representations for rendering. In typical real-time scenarios, e.g., virtual surgery simulators or computer games, such scenes are not very likely to occur. It is also worth noting that collision detection between many rigid bodies can be improved by means of any appropriate broad-phase strategy. It is, however, one of our major research goals in the future to reduce the relative load of depth-peeling in the current scenarios. A promising strategy relies on the exploitation of multiple render targets including separate depth buffers on upcoming hardware.

In contrast to previous collision detection algorithms, our method can handle ge-

ometry arbitrarily modified on the GPU. This is demonstrated by our second example above. In this case the GPU sends the modified geometry of all potentially colliding polygons to the CPU, thus taking advantage of the texture reduction we have presented. If additional geometry is created on the GPU, e.g., by using geometry shaders, only slight modifications to our algorithm are required. In particular, as the entire geometry has to be stored in one container, i.e., a texture map that can be accessed in step 3 of the collision detection algorithm, this container has to be generated in an intermediate rendering pass. All stages of the collision detection can then be executed as described. In case that the colliding geometry is not known to the CPU, computing an adequate collision response is of course a more complicated task.

In the case of deformable bodies, the proposed pipeline can be exploited very efficiently to determine collisions on the high-resolution render surface bound to the simulation mesh. Since the deformed geometry of the surface is stored on the GPU anyway, the collision pipeline fits nicely into the overall approach. Only for potentially colliding triangles (whose indices are read back to the CPU), the CPU first has to displace the triangle from its reference position to account for the actual deformation. Then, the intersection tests can be performed as described.

4.6.3 Non-Closed Polygonal Objects

Our collision detection algorithm has been developed for closed polygonal objects. However, collision detection is also required in environments where GPU objects, deformable objects, and rigid bodies interact with each other. Moreover, these scenes often include non-closed objects, such as the ground, walls, or cloth patches. To be able to deal with such environments, we present an extension of our algorithm that allows for the handling of non-closed objects. This extension comes at the expense of an increasing number of false positives to be downloaded to the CPU. Therefore, if used to detect collisions between open 2D manifolds only, we do not expect the method to perform superior compared to CPU collision detection algorithms that have been optimized for the handling of such meshes.

In general, an arbitrary open polygonal object can be closed by constructing a tight enclosing hull around it. We avoid this construction by slightly changing the way the collision rays are determined in the *object sampling* stage: a potentially colliding ray is detected if two consecutive fragments are in close depth proximity. In general, due to this modification a greater number of collision rays, i.e., false positives, is produced, resulting in increasing bandwidth requirements as well as a higher load on the CPU to finally detect the intersecting triangle pairs. The result of the proposed modification is

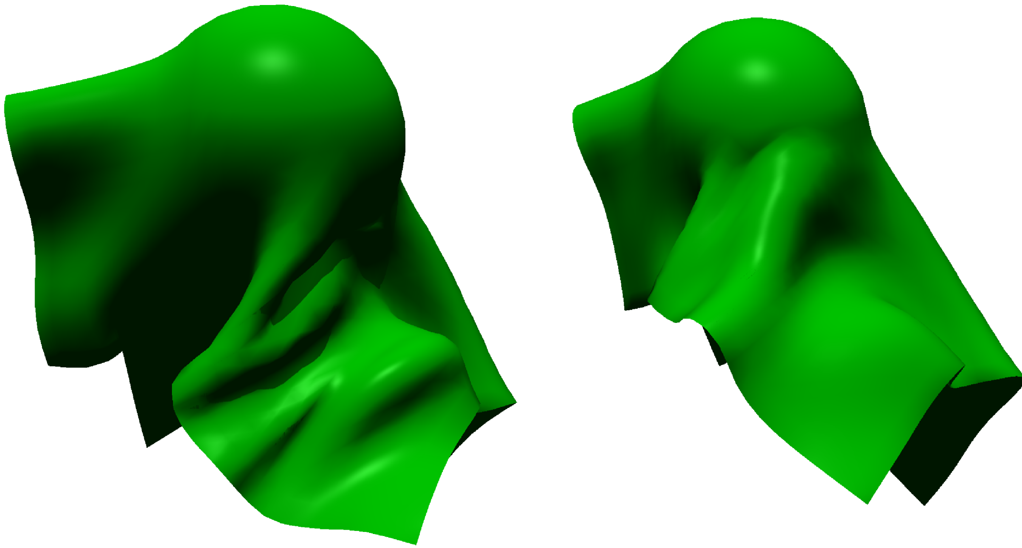
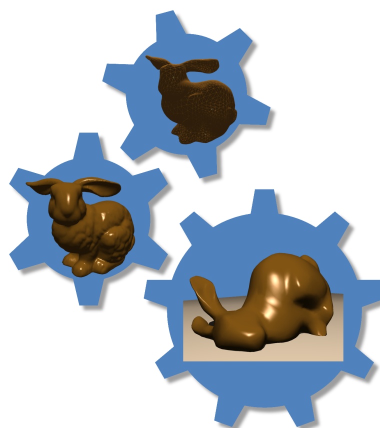


Figure 4.16: *(Self-) Collisions between non-closed dynamic and rigid objects.*

demonstrated in Figure 4.16, where a cloth patch self-interferes when colliding with a sphere object. The cloth patch consists of 2k triangles. Up to 2k triangle-triangle tests have to be performed on the CPU, and up to 300 intersecting triangle pairs have been detected. The maximum time spent for collision detection and response in one single time step of the animation sequence (the cloth patch falling down on the sphere) was 12 ms. The average time per-frame was only 5 ms.



Chapter 5

The Deformable Bodies System

This chapter focuses on the interplay of the three parts described so far—CPU simulation engine, GPU render engine, and GPU-CPU hybrid collision engine—in one single simulation support system. We give a short overview of the steps performed in the pre-processing stage to setup a simulation environment, followed by an overview of the runtime computations. In Section 5.2, we show the temporal dependencies of the system components, and how these components can effectively be distributed onto multiple CPU cores or compute nodes. Finally, we give some results obtained with this system.

5.1 Overview

The simulation support system consists of three parallel processes. The simulation engine is implemented on the CPU, and it computes the displacements of an elastic solid under external forces. The render engine is implemented on the GPU. It receives computed displacements and updates the geometry of an associated surface or volume model accordingly. While the simulation engine consecutively deforms the underlying finite element grid, i.e., the simulation geometry, the render engine subsequently displaces the geometry of the render object, i.e., the render geometry, which is attached to the simulation geometry via a weighting function. A clear conceptual separation between the simulation geometry and the render geometry enables the flexible variation of the used geometries, i.e., regular grids, point sets, or volumetric render grids. Due to this separation, a third process—the collision engine—is necessary. It has to perform

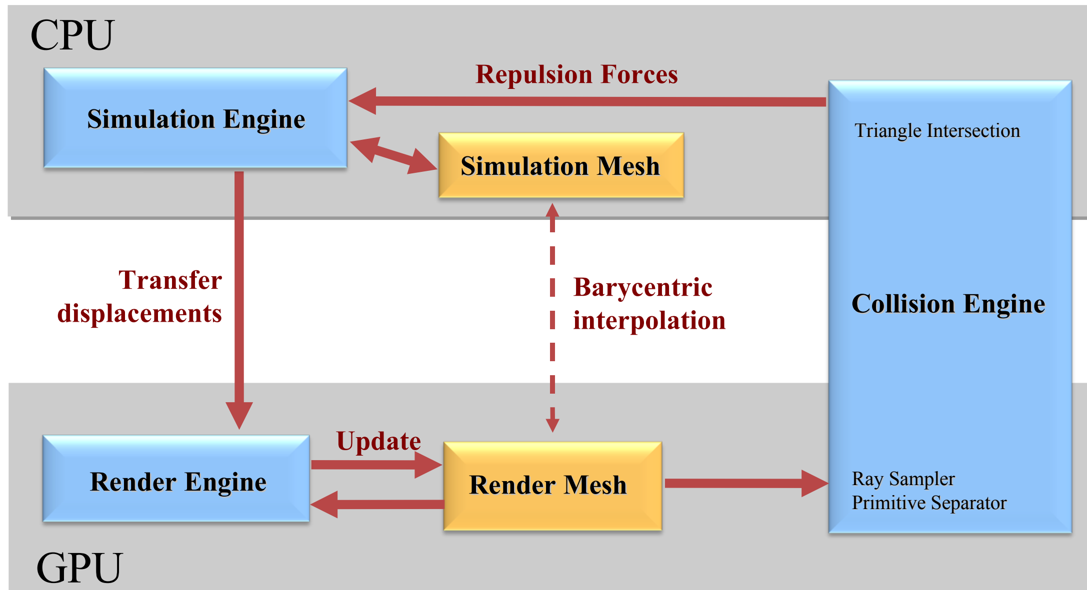


Figure 5.1: *Interplay of the simulation, render, and collision engine.*

the collision detection on the render mesh (utilizing the GPU) and resolve the collisions by displacing the simulation mesh accordingly (utilizing the CPU). Therefore, the collision engine is implemented as a GPU-CPU hybrid approach. The interplay and communication of the system parts is shown in Figure 5.1.

The particular system design has several additional properties that accommodate its use in a number of applications:

- By decoupling the resolution of the simulation geometry from the resolution of the render object, one can flexibly trade simulation or rendering quality for speed.
- The use of two separate grids, of which the render grid is attached to the simulation grid via barycentric weights, enables approximate deformation of objects that are made of far more elements than the simulation engine can handle.
- To update the render geometry, only the displaced simulation vertices have to be transferred. Bandwidth requirements can thus be reduced.

Running the proposed system involves a number of pre-processing steps as well as model driven computations at runtime. In the following we describe the different modules the system is comprised from a high-level view.

5.1.1 Generating Models

Starting with an initial object representation, a tetrahedral hierarchy that constitutes the basis for the multigrid method is generated. If such a hierarchy is already given, it can be used directly. Dedicated data structures to render the deformed high-resolution render surface are created and initialized on the GPU. The model generation is performed in the following steps:

1. Construct a 3D finite element mesh, either by using a triangle mesh and a tetrahedral mesh generation package such as TetGen [Si04], NETGEN [Sch97], or by using an adaptive subdivision scheme for tetrahedral elements [GG00].

2. Assign material properties such as stiffness and density to finite elements depending on material characteristics using pre-computed transfer functions [KD98] or segmentation results.

3. Where deformations are not allowed, fix vertices of the finite element model.

4. Generate a finite element mesh hierarchy including geometric correspondences between the meshes.

   Generate a triangle mesh hierarchy, e.g., by using a mesh decimation package [GH97], and generate a finite element mesh for each hierarchy level as described. Alternatively, use the meshes generated by the element subdivision scheme.

5. Construct a triangular render geometry. This can be the surface of the finest resolution finite element mesh, a simplified or detailed version of this mesh, or a completely different mesh.

6. Bind the render mesh vertices to vertices of the finite element mesh.

7. Store vertices of the highest resolution finite element model into a 2D texture map. Upload both the 2D texture and the render geometry including per-vertex indices into that texture and associated weights to the GPU.

5.1.2 Runtime Computations

At runtime, the following steps are performed by the simulation support system:

1. Based on external forces, compute the displacements of finite element vertices using the multigrid solver.

2. Store the displacement vectors in a 2D texture map and upload this texture to the GPU.
3. Displace the render surface on the GPU and render this surface.
4. Perform collision detection on the displaced render surface.
5. If interference is detected, calculate repulsion forces and transfer these forces to the simulation mesh using the stored weights and indices.

5.1.3 Changing Parameters at Runtime

At runtime, some parameters can be updated relatively fast. Among these, stiffness parameters and fixed vertices are the most important. As described in Section 2.3.8 and 2.3.10, both updates can be performed on the original sparse matrix data structure; to fixate vertices, some entries of the matrix are set to zero; to update stiffness values, the whole matrix is reassembled from the element matrices. In the linear setting using the Cauchy strain, both updates require to rebuild the multigrid hierarchy. Although we have developed fast algorithms for this task in Section 2.5.4, for a model consisting of 50,000 elements this still requires about 100 ms. If the stiffness matrix has to be reassembled, too, another 250 ms are required approximately.

In the simulation of the corotated Cauchy strain the stiffness matrix and the multigrid hierarchy have to be rebuilt anyway in every frame. Therefore, in this setting the updates come at no additional costs. In the nonlinear setting as described, the fixation of vertices can be performed at runtime by zeroing the entries of the non-linear equation system analogously. However, updating the stiffness values is not supported yet, since this update requires to reassemble the system of non-linear equations using the symbolic polynomial data structures.

5.2 Parallelization

Due to the separation of the system in three parts—the simulation engine, the render engine, and the collision engine—the concurrency of these engines has to be analyzed. In this section, we first describe how multiple threads can be used to interleave the calculations performed by these engines using a shared memory architecture. Then, we present a parallelization on multiple nodes, each of which only has access to its local memory. The message passing interface (MPI) is used for both communication as well as synchronization in this case.

5.2.1 Multiple Threads

The system is parallelized using POSIX® threads to instantiate a separate simulation and rendering thread. This is necessary because the execution of rendering commands on the GPU may block the application process whereas only the calling thread is blocked in a multi-threaded environment. The spawning process allocates memory that is shared by both threads to write and to read computed displacements. Both threads are synchronized via conditional variables, which allow synchronization based upon the actual value of data. By exploiting dual core architectures, idle times of the threads can be reduced noticeably.

The collision detection stages are distributed to both threads using additional syn-

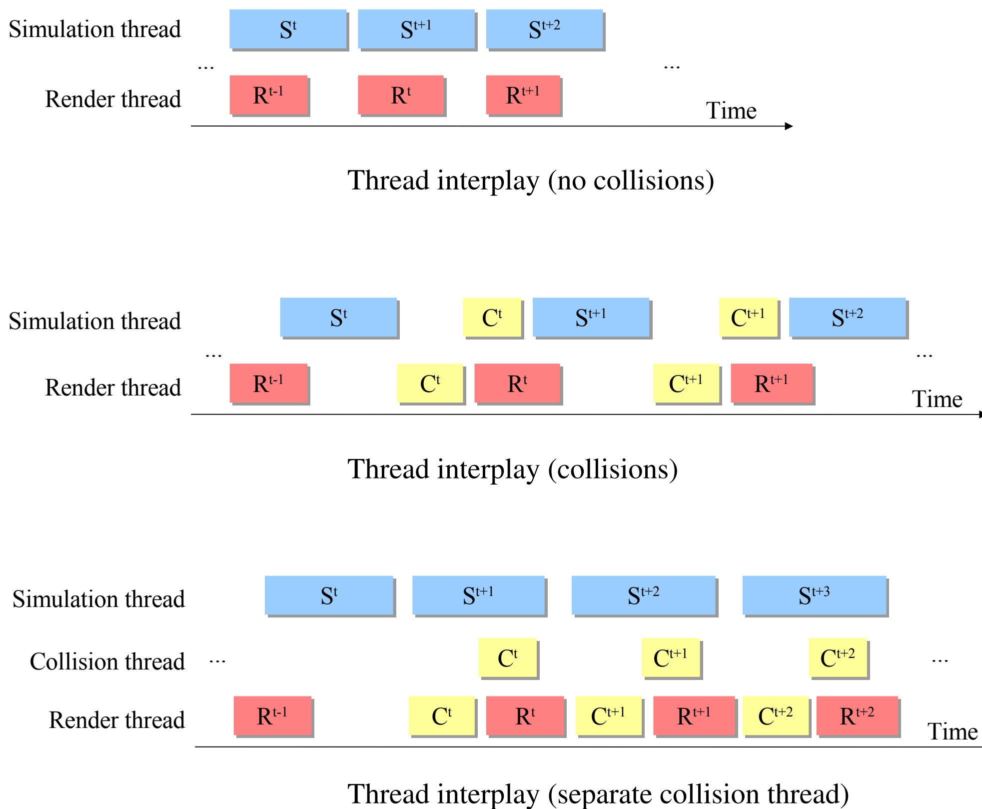


Figure 5.2: *The temporal interplay of the system threads (S = simulation, R = rendering, C = collision) is shown. If a separate collision thread is spawned, CPU and GPU idle times can be reduced significantly. On the other hand, repulsion forces of the collision response are only considered in the time step after the next.*

chronization mechanisms. This guarantees that the GPU is only used by one thread at any time. The simulation thread is suspended until the results of the collision detection are transferred from the GPU to the CPU. Then, the thread resumes its work by first determining the colliding pairs and calculating repulsion forces that are considered in the next simulation step. However, both CPU cores are not used to their full capacity in this approach (see Figure 5.2).

To exploit the full potential of the cores, a third collision thread that handles the CPU calculations of the collision algorithm can be spawned. However, due to the interleaving of these threads, the repulsion forces are considered by the simulation step after the next, thus decreasing the accuracy of the collision response noticeably. The temporal execution of the threads is also illustrated in Figure 5.2 based on the most relevant parts of the system.

Simulation Concurrency

Depending on the kind of simulation (linear, corotated, non-linear) that is performed, there is an opportunity to parallelize computations within the simulation engine, too. In the corotated setting, multiple threads performing matrix reassembly, multigrid update, and multigrid solution can be efficiently used as shown in Section 2.7. The temporal interplay of these threads is shown in Figure 5.3. In the non-linear setting, an arbitrary number of threads can be used to evaluate both the Jacobian matrix and the system of equations given the current displacement field u. All of these additional computing threads are synchronized with the main simulation thread in every frame. Matrix vector products as well as Jacobi (instead of Gauss-Seidel) relaxation could be parallelized in principle, too. However, this would result in rather small parallel blocks introducing a noticeable communication overhead. For that reason we have not considered this option in this work.

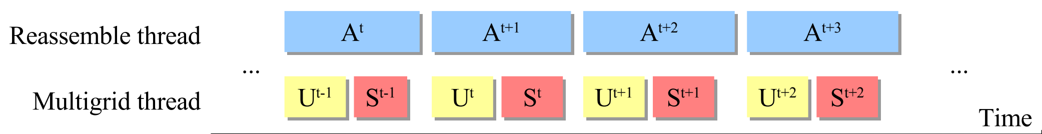


Figure 5.3: *Interplay of the simulation threads in the corotated setting. Since the time for matrix assembly (A) is more than 50% of the overall time (see Table 2.9), one thread for multigrid update (U) and multigrid solution (S) is sufficient.*

5.2.2 Multiple Nodes

We also consider how the system can be distributed on several compute nodes that do not share memory. The nodes communicate via the message-passing interface (MPI) [For03] implemented on an InfiniBand® network. For example, to support stereo rendering or tiled displays at full performance rate, the system can optionally be run on multiple render nodes. Each render node holds its own render geometry on the local GPU and updates it according to the received displacement field. This field is distributed to all render nodes by the MPI broadcast functionality.

We have tested the system with four render nodes connected via an InfiniBand® network. Due to the fast and low-latency connections, the communication overhead in the current scenario is negligible. The simulation runs on a separate node providing two to four CPUs for the additional simulation threads. The collision engine can be run on an additional node. It updates the render geometry according to the render nodes and performs object sampling, ray merging, and primitive separation on the graphics

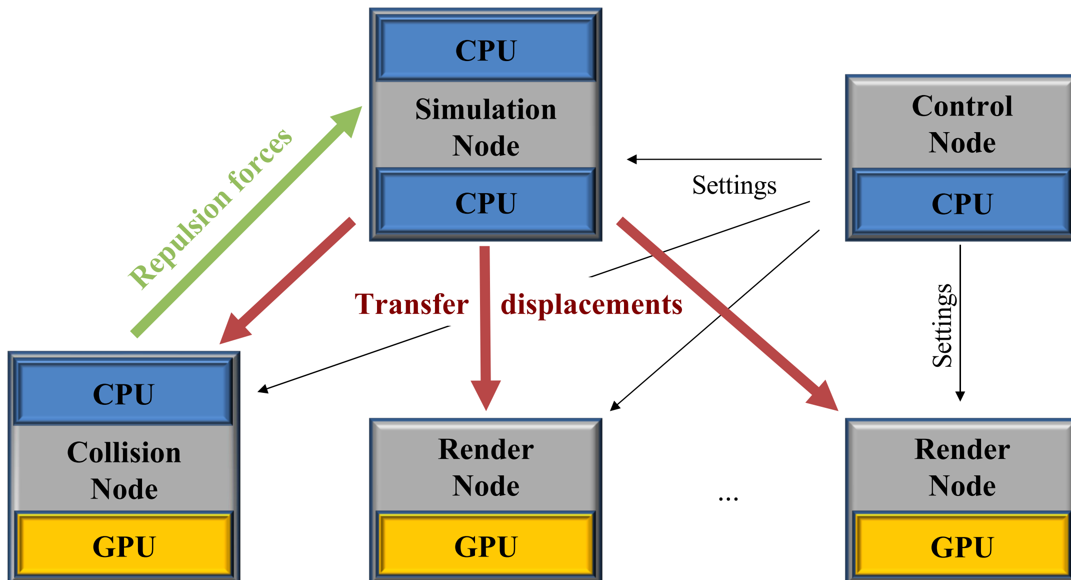


Figure 5.4: *System distribution on multiple compute nodes. The simulation node runs on multiple CPU cores with shared memory. Displacements are transferred to an arbitrary number of render nodes, each of which stores its own copy of the render geometry in local GPU memory. A separate collision node is used to perform collision detection with the GPU-CPU hybrid approach. The whole application is controlled by a separate node, which handles the graphical user interfaces and distributes the settings to all nodes. All nodes are synchronized at the end of each simulation step using an MPI barrier command.*

hardware. The intersection tests are performed on the local CPU after potentially colliding primitives have been downloaded. Finally, calculated repulsion forces are sent to the simulation node. The system can be controlled by a separate node handling the graphical user interface. However, this node does not necessarily run on a detached hardware. The interplay of all components is shown in Figure 5.4.

5.3 Results

Most of the algorithms and methods presented in this thesis have been integrated into one single tool—the tum.3D defo application. Note that the GPU simulation based on the mass-spring approaches is not included in this application. The interplay of the components has been thoroughly discussed previously. A screenshot of the user interface is given in Figure 5.5.

On the simulation side, linear and corotated Cauchy strain as well as non-linear Green strain are supported. Material parameters, boundary conditions such as vertex fixation and the time integration scheme to be used can be selected via the graphical user interface. Objects can be deformed by picking arbitrary parts of these objects with the mouse—or optionally by using a haptic input device. In both cases, a ray-object intersection test is performed at the screenspace position of the cursor to determine

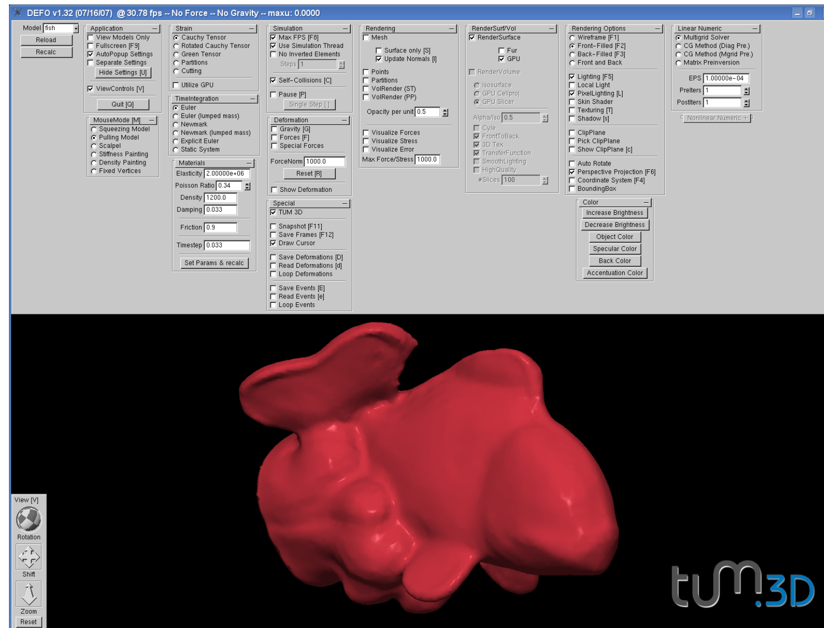


Figure 5.5: *Screenshot of the tum.3D defo application.*

the finite element which is picked by the user. On the rendering side, all the options described in Chapter 3 are integrated into the system and can be selected from the menus.

Force Fields

A number of static and dynamic force fields have been integrated into the system to achieve specific simulation effects. Figure 5.6 shows an example where a height field composed of triangular elements is deformed using wave-like force fields to simulate earth quakes. Although this simple approach is of course not a serious earth quake simulation, the effect looks plausible and can be computed very quickly. The mesh consists of roughly 60k triangular elements and the simulation still runs at 20 time steps per second. Source or sink vector fields can be used to achieve intuitive deformations. In Figure 5.7, a balloon image is blown up or shrunk, respectively. In Figure 5.8 an additional example of the use of specific force fields is demonstrated. By applying a radial symmetric force field, implants can be simulated at low computational costs, thereby assisting the surgeon in the selection of the best implant parameters, implant position, and operative access for the patient.

In Figure 2.27, other kinds of force fields such as wind have been shown. In combination with the collision detection approach, realistic simulation of cloth is achieved (see Figure 4.16 at the end of Chapter 4). However, the presented simulation model is

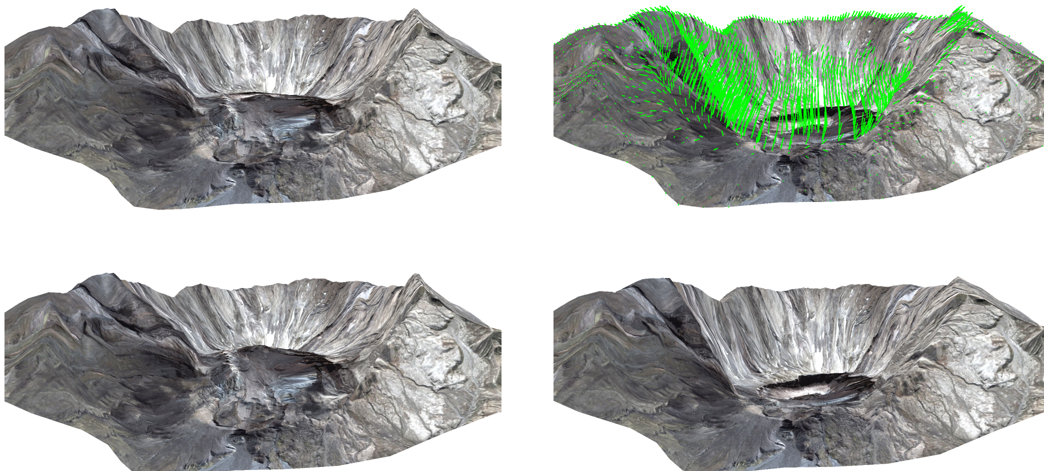


Figure 5.6: *Deforming the Mount St. Helens terrain to achieve an earthquake effect for gaming environments. In the top-right image, the displacement field for one time step of the animation is visualized.*

(a) (b) (c)

Figure 5.7: *A balloon image (a) can be blown up by inserting a source vector field (b) or shrunk by inserting a sink vector field (c).*

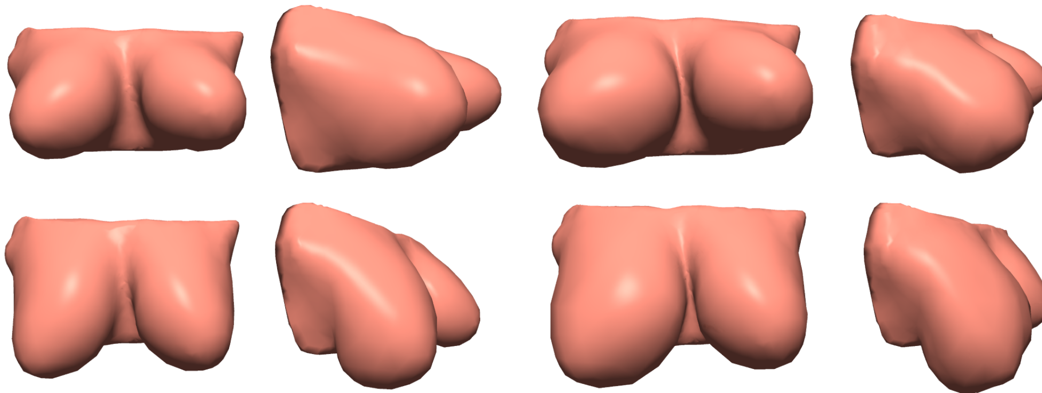


Figure 5.8: *Breast augmentation as one potential application of the proposed multigrid simulation framework: Gravity, different material properties as well as additional forces induced by implants can be simulated interactively.*

not very well suited for the physics of cloth, since folds and wrinkles cannot be simulated natively. In fact, there exist many specialized papers that focus on cloth simulation [BW98, BMF03, GHDS03, EKS03]. Although we have not included the proposed so-called bending forces in our tum.3D defo engine, the mentioned approaches could still benefit from the multigrid framework.

High Performance

The example of the deformable dragon model in Figure 5.9 demonstrates the effectiveness of the interplay of the CPU simulation engine and GPU render engine. Due to the interleaving of both system components, the overall frame rate drops down only slightly from 16.7 fps (simulation only) to 16.5 fps when a high-resolution render mesh consisting of 800k triangles is displaced according to the 67k tetrahedra simulation

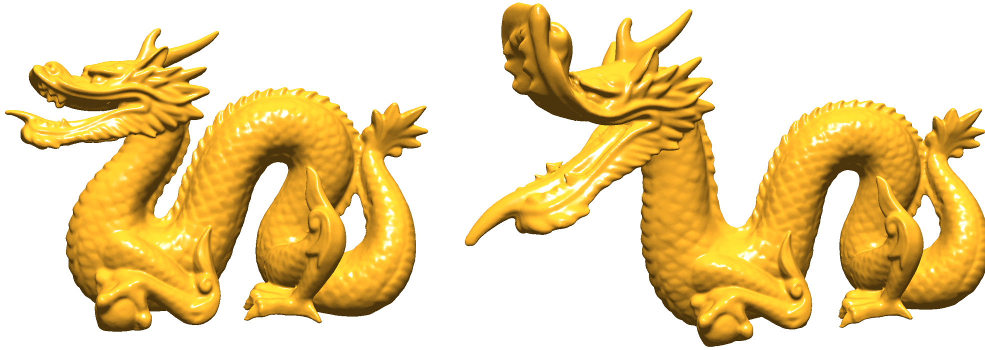


Figure 5.9: *The deformation of the dragon model demonstrates the effectiveness of the interplay of the simulation and render engine. While a 67k tetrahedral mesh is simulated on the CPU, 800k triangles are displaced accordingly on the GPU without affecting the overall performance.*

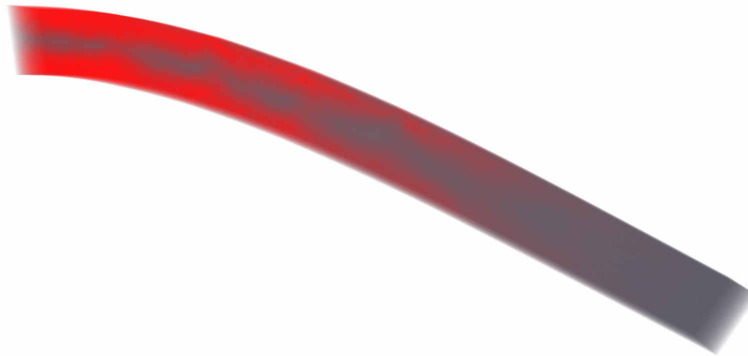
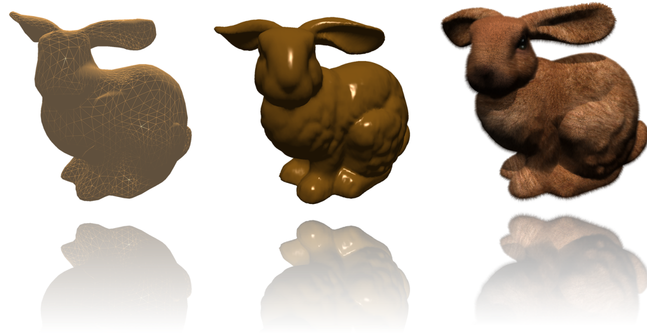


Figure 5.10: *Stress visualization of a bending beam under gravity. Both the simulation mesh and the render mesh consist of 24k tetrahedral elements.*

mesh. If this mesh is displaced on the CPU as in traditional approaches, the overall frame rate drops down to 12 fps on an Intel Core™ 2 Duo system equipped with an NVIDIA 8800 GTX graphics card.

The coupling with our volume rendering engine enables online observation of internal material properties. In Figure 5.10 the internal stress of a bridge (fixed on the left-hand side) under gravity is shown. Furthermore, the deformation of regular 3D volumes has been demonstrated in Figure 3.14 and 3.19 in Chapter 3. The effectiveness and efficiency of the collision detection approach in combination with the render engine has already been thoroughly discussed in Chapter 4 (see Figure 4.13 for an example).



Chapter 6

Conclusion

In this thesis, I have presented an implicit multigrid framework for interactive and physics-based simulation of deformable volumetric bodies, which is open to a variety of different strain measures. I demonstrated how this framework has been extended by a wide range of modified material laws, which also allow for intuitive non-physical deformations. The proposed multigrid solver effectively benefits from coarse grid correction in that it produces numerically stable results yet minimizing the number of iterations to be performed until convergence. The proposed multigrid solver allows for the simulation of homogeneous materials as well as heterogeneous materials, i.e., materials exhibiting varying stiffness and density, without sacrificing speed or quality.

I have further demonstrated, that the 1-step stream acceleration approach for sparse matrices efficiently updates the multigrid hierarchy based on the Galerkin property. This technique yields a generic multigrid framework, since only sparse (optionally symmetric) matrices and appropriate restriction/interpolation operators are required. Therefore, the proposed algorithms can greatly accelerate other applications, too. As a consequence, the multigrid framework has the potential to be integrated into many real-time scenarios such as surgical simulators or virtual environments. In particular, I demonstrated the capabilities of the proposed methods in the context of soft tissue deformation and medical applications. It has been validated that highly accurate results can be achieved at interactive rates.

It has been demonstrated that the GPU can outperform the CPU when simulating simple physical models such as mass-spring systems. Furthermore, I have identified the problems of GPU simulation if high numerical accuracy is required. However,

this judgment might be weakened in the future due to improved graphics architectures providing double precision and improved support for scattered write operations.

I have developed a method to improve the rendering of deformable objects. By exploiting the capabilities of modern graphics architectures, high-resolution meshes can be displaced according to the underlying simulation mesh. Due to the elaborate system design, simulation rates are usually not affected by the rendering. Then, I presented a generic and scalable rendering pipeline for tetrahedral grids. The pipeline is designed to facilitate its use on recent and upcoming graphics hardware and to accommodate the rendering of large and deformable grids. In particular, it has been shown that the proposed concept supports upcoming features on programmable graphics hardware and thus has the potential to achieve significant performance gains in the near future.

A novel collision detection algorithm that is particularly designed for recent and future graphics hardware has been presented. It exploits the intrinsic strength of GPUs to scan-convert large sets of polygons and to shade billions of fragments at interactive rates. The suggested design makes the method suitable for applications where geometry is deformed or even created on the GPU. In a number of different examples these statements have been verified.

I believe that the suggested algorithm is influential for future research in the field of collision detection. For the first time it has been shown that collision detection between objects that are modified or created on the GPU can successfully be accomplished. With Direct3D 10 compliant hardware and geometry shaders being available, this feature will be required in many different applications. In contrast to previous GPU-based algorithms for collision detection, all objects can remain in their renderable representation and do not have to be converted into another format. The burden of frequently updating hierarchical data structures is removed from the application program.

Finally, I have demonstrated the effectiveness and efficiency of the interplay of the simulation, rendering, and collision engine. I further demonstrated how force fields applied to the simulation can be used as a powerful modeling tool.

6.1 Future Work

The presented results allow for further improvements. On the side of the numerical simulation, I want to mention that the applied numerical operations, e.g., matrix-vector products, can be further improved by applying hardware-specific optimization techniques such as SSE optimizations. Moreover, performance improvements can be achieved by optimizing the memory access patterns of sparse matrix-vector products.

In particular, the Cuthill-McKee algorithm [CM69] reduces the bandwidth of a sparse matrix by reordering the rows and columns (the respective vertices of the finite element mesh). Especially on CPUs with small second level cache, I expect faster simulation rates if matrices with reduced bandwidth are used.

The multigrid framework requires to have a hierarchy of grids to operate on. Typically, a data acquisition process, e.g. using a laser or CT scanner, cannot guarantee to generate watertight surface meshes at the very end. Thus, a post-processing step is necessary to construct meshes that can be used by packages such as TetGen [Si04] or NETGEN [Sch97]. Furthermore, simplified meshes have to be generated to construct a hierarchy of volumetric grids. In future, I will investigate how the mesh generation process can be simplified by suggestions of Molino et al. [MBTF03]. There, models are generated by deforming a body-centered cubic (BCC) lattice to match the given surface mesh. For the initial BCC lattice, a nested grid hierarchy can be obtained easily. During the matching process, the coarser grids can be deformed according to the finest grid yielding a grid hierarchy for the object of interest. Fortunately, since the grids deform only close to the surface, the resulting hierarchies are still widely nested at the same time well approximating the object's boundaries. Due to the nestedness, the multigrid framework would benefit significantly from these meshes in terms of performance.

The GPU render engine can be further improved by applying higher-order interpolation to bind the render mesh to the simulation grid. Due to the excellent performance rates that have been achieved with linear functions, interpolation functions with an increased support can still be evaluated at highly interactive rates on the GPU. The higher-order interpolation yields better deformations of the render mesh, especially if the render mesh features much more details than the simulation mesh can resolve. In particular, in case of higher-order finite elements, the respective shape functions can be utilized to bind the render mesh. Furthermore, a vertex of the render mesh might not only be bound to just one finite element but to a larger number of elements.

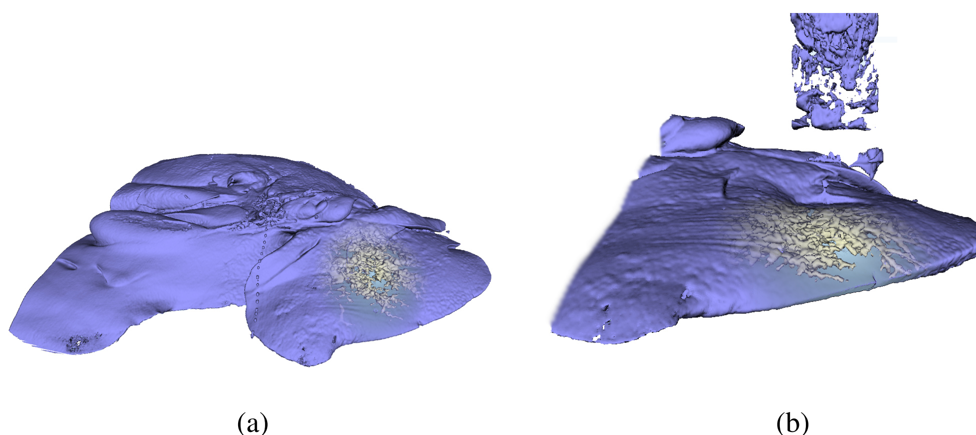
Besides the verification of my current results for the tetrahedral grid rendering pipeline on future Direct3D 10 graphics hardware, I will investigate the integration of acceleration techniques for volume ray-casting into the current approach. In particular, early ray termination as proposed for texture-based volume ray casting [KW03a] seems to be a promising acceleration strategy that perfectly fits into the dedicated rendering pipeline. Furthermore, the rendering pipeline can be improved by the following observation: Elements, that intersect several shells have to be rendered multiple times. By determining these elements in advance and by making use of the stream output stage (see Figure 3.1), multiple computations of the element assembly and primitive

construction stage can effectively be avoided, because these elements can be directly rendered from their previously saved state.

The improvement of the collision response calculations is an interesting point for future research. The proposed repulsion forces can be easily integrated into real-time environments. However, they cannot guarantee to properly resolve collisions in the next time step. Therefore, more advanced response strategies should be integrated into the GPU-CPU hybrid pipeline. In that context it is of special interest, which parts of the collision response can eventually be accelerated by a GPU implementation, or which information the GPU can additionally provide to potentially simplify the collision response calculations on the CPU.

Concerning medical applications, especially the validation of the simulation can be further considered. By comparing the vascular structure of the simulated liver with the real-world CT scan of the liver deformed by specific weights, the accuracy of the simulation with respect to interior structures has to be further validated. In Figure 6.1 (a), a reference scan of a liver is shown. Due to the radiopaque material, the vascular structure becomes clearly visible. In Figure 6.1 (b) the liver is deformed by a specific weight, and the internal structures have deformed, too. In the future I want to investigate whether the vascular structures in the simulation match the measured ones. Moreover, besides largely homogeneous materials, also inhomogeneous organs have to be considered. In this case, however, accurate measurements of the material parameters of the different tissue types the organ is composed of have to be performed in advance.

Surgical training environments allowing for the simulation of complex deforma-



(a) (b)

Figure 6.1: *Validation of the soft tissue deformation by means of volumetric organic data sets. (a) The reference scan clearly shows the vascular structures in the interior. (b) The data set is deformed using a specific weight. Vascular structures in the simulated deformation of the reference scan should match the vascular structures of the measured deformed liver.*

tions of soft-tissue models are more and more frequently employed in practice due to recent advances in real-time simulation of elastic soft-tissue. One of the challenges in such simulators is to provide the surgeon not only a view of the outer model boundary but also a detailed view of the interior structures and their deformation due to the applied operation. During surgeon training, one important aspect is to track the tissue stress induced by an intervention. If tissue is stressed too much, this often implies additional risks for the patient during a real surgical intervention. I want to investigate how the efficient algorithms developed in this thesis can be effectively applied in these environments.

For an exemplary application, I plan to apply the system to female breast reconstruction and augmentation. Major parts of the breast consist of homogeneous material, and thus this organ can be approximated adequately with rather coarse tetrahedral meshes (about 50,000 tetrahedra) simulating linear elasticity. Moreover, female breast augmentation is a huge market, and therefore simulation-supported systems are likely to pay off in the clinical practice. It is planned to generate so-called breast templates, which are finite-element models of the breast, where different stiffness values are assigned to each tetrahedral element based on the type of the underlying tissue (muscle, skin, adipose tissue). The model can be adapted to a specific patient by deforming it according to the patient's surface information. Then, one can simulate gravity and different kinds of implants through artificial force fields, which are carefully adapted to reflect the kind of implant (anatomic vs. round) and its volume. In combination with the stress visualization technique described, the surgeon can ensure that tissue close to the implant is not stressed too much. Moreover, the surgeon can potentially minimize post-operative complications (e.g., swelling, bleeding) by optimizing the surgical approach regarding implant parameters, implant position, and operative access taking the material properties of the different breast soft tissue types into account.

Bibliography

[ACOL00] Marc Alexa, Daniel Cohen-Or, and David Levin, *As-rigid-as-possible shape interpolation*, Proceedings of SIGGRAPH (New York, NY, USA), ACM Press/Addison-Wesley Publishing Co., 2000, pp. 157–164.

[AKS05] Burak Aksoylu, Andrei Khodakovsky, and Peter Schröder, *Multilevel solvers for unstructured surface meshes*, SIAM Journal on Scientific Computing **26** (2005), no. 4, 1146–1165.

[BA04] Eddy Boxerman and Uri Ascher, *Decomposing cloth*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2004, pp. 153–161.

[Bat02] Klaus-Jürgen Bathe, *Finite element procedures*, Prentice Hall, 2002.

[BBE+04] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang, *PETSc users manual*, Tech. Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[BBK05] Mario Botsch, David Bommes, and Leif Kobbelt, *Efficient linear system solvers for mesh processing*, Lecture Notes in Computer Science: Mathematics of Surfaces XI **3604** (2005), 62–83.

[BFA02] Robert Bridson, Ronald Fedkiw, and John Anderson, *Robust treatment of collisions, contact and friction for cloth animation*, Proceedings of SIGGRAPH, 2002, pp. 594–603.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, Proceedings of SIGGRAPH, 2003, pp. 917–924.

[BFH04] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan, *GPUBench: Evaluating GPU performance for numerical and scientific applications*, Proceedings of ACM Workshop on General-Purpose Computing on Graphics Processors, 2004.

[BG05] Rudolph Balaz and Sam Glassenberg, *DirectX and Windows Vista Presentations*, http://msdn.microsoft.com/directx/archives/pdc2005/, 2005.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick, *A multigrid tutorial, second edition*, SIAM, 2000.

[BK05] Mario Botsch and Leif Kobbelt, *Real-time shape editing using radial basis functions*, Proceedings of Eurographics, 2005, pp. 611–621.

[BM05] Rob H. Bisseling and Wouter Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis **21** (2005), 47–65.

[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, pp. 28–36.

[BNC96] Morten Bro-Nielsen and Stephane Cotin, *Real-time volumetric deformable models for surgery simulation using finite elements and condensation*, Proceedings of Eurographics, 1996, pp. 57–66.

[BPWG07] Mario Botsch, Mark Pauly, Martin Wicke, and Markus Gross, *Adaptive space deformations based on rigid cells*, Proceedings of Eurographics, 2007.

[Bra77] Archi Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of Computation **31** (1977), no. 138, 333–390.

[Bra01] Dietrich Braess, *Finite elements: Theory, fast solvers, and applications in solid mechanics*, Cambridge Univ. Press, 2001.

[BS87] Randolph E. Bank and R. Kent Smith, *General sparse elimination requires no permanent integer storage*, SIAM Journal on Scientific and Statistical Computing **8** (1987), no. 4, 574–584.

[BW98] David Baraff and Andrew Witkin, *Large steps in cloth simulation*, Proceedings of SIGGRAPH, 1998, pp. 43–54.

[BW99] A. J. C. Bik and H. A. G. Wijshoff, *Automatic nonzero structure analysis*, SIAM J. Comput. **28** (1999), 1576–1587.

[BW02] George Baciu and Wingo Sai-Keung Wong, *Hardware-assisted self-collision for deformable surfaces*, Proceedings of the ACM symposium on Virtual reality software and technology, 2002, pp. 129–136.

[CBPS06] Steven P. Callahan, Louis Bavoil, Valerio Pascucci, and Claudio T. Silva, *Progressive volume rendering of large unstructured grids*, IEEE Transactions on Visualization and Computer Graphics **12** (2006), no. 5, 1307–1314.

[CDA99] Stephane Cotin, Herve Delingette, and Nicholas Ayache, *Real-time elastic deformations of soft tissues for surgery simulation*, IEEE Transactions on Visualization and Computer Graphics, 1999, pp. 62–73.

[CFL28] Richard Courant, Kurt Friedrichs, and Hans Lewy, *Über die partiellen Differenzengleichungen der mathematischen Physik*, Mathematische Annalen **100** (1928), no. 1, 32–74.

[CFL67] Richard Courant, Kurt O. Friedrichs, and Hans Lewy, *On the partial difference equations of mathematical physics*, IBM Journal (1967), 215–234.

[CGC+02a] Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović, *Interactive skeleton-driven dynamic deformations*, Proceedings of SIGGRAPH, 2002, pp. 586–593.

[CGC+02b] ———, *A multiresolution framework for dynamic deformations*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2002, pp. 41–47.

[CICS05] Steven P. Callahan, Milan Ikits, João L. D. Comba, and Claudio T. Silva, *Hardware-assisted visibility sorting for unstructured volume rendering*, IEEE Transactions on Visualization and Computer Graphics **11** (2005), no. 3, 285–295.

[CKM+99] João Comba, James T. Klosowsky, Nelson Max, Joseph S. B. Mitchell, Claudio T. Silva, and Peter L. Williams, *Fast polyhedral cell sorting for interactive rendering of unstructured grids*, Proceedings of Eurographics, 1999, pp. 369–376.

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi, *I-COLLIDE: An interactive and exact collision detection system for large-scale environments*, Proceedings of the symposium on Interactive 3D graphics, 1995, pp. 189–196.

[CM69] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, Proceedings of the 1969 24th national conference (New York, NY, USA), ACM Press, 1969, pp. 157–172.

[CMSS95] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno, *On the optimization of projective volume rendering*, Proceedings of EG Workshop, Scientific Visualization in Scientific Computing, 1995, pp. 58–71.

[Cro77] Franklin C. Crow, *Shadow algorithms for computer graphics*, Proceedings of SIGGRAPH, 1977, pp. 242–248.

[CS05] Daniel S. Coming and Oliver G. Staadt, *Kinetic sweep and prune for collision detection*, Proceedings of 2nd Workshop On Virtual Reality Interaction and Physical Simulation, 2005, pp. 81–90.

[DDBC99] Gilles Debunne, Mathieu Desbrun, Alan Barr, and Marie-Paule Cani, *Interactive multiresolution animation of deformable models*, Eurographics Workshop on Computer Animation and Simulation, 1999, pp. 133–144.

[DDCB01] Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr, *Dynamic real-time deformations using space & time adaptive sampling*, Proceedings of SIGGRAPH, 2001, pp. 31–36.

[DK91] Akio Doi and Akio Koide, *An efficient method of triangulating equi-valued surfaces by using tetrahedral cells*, IEICE Transactions on Information and Systems **E74-D** (1991), no. 1, 214–224.

[DKT98] Tony DeRose, Michael Kass, and Tien Truong, *Subdivision surfaces in character animation*, Proceedings of SIGGRAPH, 1998, pp. 85–94.

[DSB99] Mathieu Desbrun, Peter Schröder, and Alan Barr, *Interactive animation of structured deformable objects*, Proceedings of Graphics Interface, 1999, pp. 1–8.

[EGSS82] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *Yale sparse matrix package*, International Journal of Numerical Methods for Engineering (1982), 1145–1151.

[EKS03] Olaf Etzmuß, Michael Keckeisen, and Wolfgang Straßer, *A fast finite element solution for cloth modelling*, Proceedings of Pacific Conference on Computer Graphics and Applications, 2003, p. 244.

[Eve01] Cass Everitt, *Interactive order-independent transparency*, Tech. report, NVIDIA Corporation, 2001.

[FGL03] Arnulph Fuhrmann, Clemens Groß, and Volker Luckas, *Interactive animation of cloth including self collision detection*, Proceedings of WSCG, 2003, pp. 141–148.

[For03] The MPI Forum, *MPI: A message-passing interface standard*, http://www.mpi-forum.org/docs/docs.html, 2003.

[GBK05] Michael Guthe, Ákos Balázs, and Reinhard Klein, *GPU-based trimming and tessellation of NURBS and T-Spline surfaces*, ACM Transactions Graphics **24** (2005), no. 3, 1016–1023.

[GEK+07] Joachim Georgii, Maximilian Eder, Laszlo Kovacs, Armin Schneider, Martin Dobritz, and Rüdiger Westermann, *Advanced volume rendering for surgical training environments*, International Journal of Computer Assisted Radiology and Surgery **2** (2007), no. 1, S285.

[Gel98] Allen Van Gelder, *Approximate simulation of elastic membranes by triangulated spring meshes*, Journal of Graphics Tools **3** (1998), no. 2, 21–42.

[GEW05] Joachim Georgii, Florian Echtler, and Rüdiger Westermann, *Interactive Simulation of Deformable Bodies on GPUs*, Proceedings of Simulation and Visualisation, 2005, pp. 247–258.

[GFG04] Philip Gerasimov, Randima Fernando, and Simon Green, *Whitepaper: Shader Model 3.0 Using Vertex Textures*, http://developer.nvidia.com/object/using_vertex_ textures.html, 2004.

[GG00] Gunther Greiner and Roberto Grosso, *Hierarchical tetrahedral-octahedral subdivision for volume visualization*, The Visual Computer **16** (2000), no. 6, 357–369.

[GGK06] Alexander Greß, Michael Guthe, and Reinhard Klein, *GPU-based collision detection for deformable parameterized surfaces*, Computer Graphics Forum **25** (2006), no. 3, 497–506.

[GH97] Michael Garland and Paul S. Heckbert, *Surface simplification using quadric error metrics*, Proceedings of SIGGRAPH (New York, NY, USA), ACM Press/Addison-Wesley Publishing Co., 1997, pp. 209–216.

[GH06] Simon Green and Mark Harris, *Physics simulation on NVIDIA GPUs*, http://developer.nvidia.com/object/havok-fx-gdc-2006.html, 2006.

[GHDS03] Eitan Grinspun, Anil N. Hirani, Mathieu Desbrun, and Peter Schröder, *Discrete shells*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2003, pp. 62–67.

[Gie92] Christopher Giertsen, *Volume visualization of sparse irregular meshes*, IEEE Computer Graphics and Applications **12** (1992), no. 2, 40–48.

[GKJ+05] Naga K. Govindaraju, David Knott, Nitin Jain, Ilknur Kabul, Rasmus Tamstorf, Russell Gayle, Ming C. Lin, and Dinesh Manocha, *Interactive collision detection between deformable models using chromatic decomposition*, ACM Transaction on Graphics **24** (2005), no. 3, 991–999.

[GKMV03] Sudipto Guha, Shankar Krishnan, Kamesh Munagala, and Suresh Venkatasubramanian, *Application of the two-sided depth test to CSG rendering*, Proceedings of the symposium on Interactive 3D graphics, 2003, pp. 177–180.

[GKS02] Eitan Grinspun, Petr Krysl, and Peter Schröder, *CHARMS: a simple framework for adaptive simulation*, Proceeding of SIGGRAPH, 2002, pp. 281–290.

[GKW07] Joachim Georgii, Jens Krüger, and Rüdiger Westermann, *Interactive GPU-based collision detection*, Proceedings of IADIS Computer Graphics and Visualization, 2007, pp. 3–10.

[GLM96] Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha, *OBBTree: a hierarchical structure for rapid interference detection*, Proceedings of SIGGRAPH, 1996, pp. 171–180.

[GLM04] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha, *Fast and reliable collision culling using graphics hardware*, Proceedings of the ACM symposium on Virtual reality software and technology, 2004, pp. 2–9.

[GLM05] ———, *Quick-CULLIDE: Fast Inter- and Intra-Object Collision Culling Using Graphics Hardware*, Proceedings of IEEE Virtual Reality Conference, 2005, pp. 59–66.

[GM97] Sarah F. Gibson and Brian Mirtich, *A survey of deformable models in computer graphics*, Technical Report TR-97-19, Mitsubishi, 1997.

[GRLM03] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha, *CULLIDE: interactive collision detection between complex models in large environments using graphics hardware*, Proceedings of the ACM SIGGRAPH/Eurographics conference on Graphics hardware, 2003, pp. 25–32.

[GSK+06] Joachim Georgii, Jens Schneider, Jens Krüger, Rüdiger Westermann, Maximilian Eder, and Laszlo Kovacs, *Advanced volume rendering techniques for medical applications*, 5. Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie (CURAC), 2006, pp. 146–147.

[Gus78] Fred G. Gustavson, *Two fast algorithms for sparse matrices: Multiplication and permuted transposition*, ACM Trans. Math. Softw. **4** (1978), no. 3, 250–269.

[GW05a] Joachim Georgii and Rüdiger Westermann, *A Multigrid Framework for Real-Time Simulation of Deformable Volumes*, Proceedings of the 2nd Workshop On Virtual Reality Interaction and Physical Simulation, 2005, pp. 50–57.

[GW05b] ______, *Interactive Simulation and Rendering of Heterogeneous Deformable Bodies*, Proceedings of Vision, Modeling and Visualization, 2005, pp. 383–390.

[GW05c] ______, *Mass-Spring Systems on the GPU*, Simulation Modelling Practice and Theory **13** (2005), 693–702.

[GW06a] ______, *A Multigrid Framework for Real-Time Simulation of Deformable Bodies*, Computer & Graphics **30** (2006), 408–415.

[GW06b] Joachim Georgii and Rüdiger Westermann, *A generic and scalable pipeline for GPU tetrahedral grid rendering*, Proceedings of IEEE Visualization, 2006, pp. 1345–1352.

[GWF04] Joachim Georgii, Rüdiger Westermann, and Hubertus Feussner, *Physically accurate real-time simulation of deformable bodies for surgical training and therapy planning*, 3. Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie (CURAC), 2004.

[Hac85] Wolfgang Hackbusch, *Multi-grid methods and applications*, Springer Series in Computational Mathematics, Springer, 1985.

[HBSL03] Mark J. Harris, Williams V. Baxter, Thorsten Scheuermann, and Anselmo Lastra, *Simulation of cloud dynamics on graphics hardware*, Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2003, pp. 92–101.

[Hig86] Nicholas J. Higham, *Computing the polar decomposition—with applications*, SIAM Journal on Scientific and Statistical Computing **7** (1986), no. 4, 1160–1174.

[HKM95] Martin Held, James T. Klosowski, and Joseph S. B. Mitchell, *Evaluation of collision detection methods for virtual reality fly-throughs*, Proceedings of the Seventh Canadian Conference on Computational Geometry, 1995, pp. 205–210.

[HR05] John Hable and Jarek Rossignac, *Blister: GPU-based rendering of boolean combinations of free-form triangulated shapes*, Proceedings of SIGGRAPH, 2005, pp. 1024–1031.

[HS90] Nicholas J. Higham and Robert S. Schreiber, *Fast polar decomposition of an arbitrary matrix*, SIAM J. Sci. Stat. Comput. **11** (1990), no. 4, 648–655.

[HS04] Michael Hauth and Wolfgang Straßer, *Corotational simulation of deformable solids*, Proceedings of WSCG, 2004, pp. 137–145.

[HSO03] Kris K. Hauser, Chen Shen, and James F. O'Brien, *Interactive deformation using modal analysis with constraints*, Proceedings of Graphics Interface, 2003, pp. 247–256.

[HTG04] Bruno Heidelberger, Matthias Teschner, and Markus Gross, *Detection of collisions and self-collisions using image-space techniques*, Proceedings of WSCG, 2004, pp. 145–152.

[Hub95] Philip M. Hubbard, *Collision detection for interactive graphics applications*, IEEE Transactions on Visualization and Computer Graphics **1** (1995), no. 3, 218–230.

[Hub96] ______, *Approximating polyhedra with spheres for time-critical collision detection*, ACM Transactions on Graphics **15** (1996), no. 3, 179–210.

[JP99] Doug L. James and Dinesh K. Pai, *ArtDefo: accurate real time deformable objects*, Proceedings of SIGGRAPH, 1999, pp. 65–72.

[JP02] ______, *DyRT: dynamic response textures for real time deformation simulation with graphics hardware*, Proceedings of SIGGRAPH, 2002, pp. 582–585.

[JP04] ______, *BD-tree: output-sensitive collision detection for reduced deformable models*, ACM Transactions on Graphics **23** (2004), no. 3, 393–398.

[KD98] Gordon Kindlmann and James W. Durkin, *Semi-automatic generation of transfer functions for direct volume rendering*, Proceedings of the IEEE Symposium on Volume Visualization, 1998, pp. 79–86.

[KHM+98] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan, *Efficient collision detection using bounding volume hierarchies of k-DOPs*, IEEE Transactions on Visualization and Computer Graphics **4** (1998), no. 1, 21–36.

[KLNR03] Michael S. Karasick, Derek Lieber, Lee R. Nackman, and V. T. Rajan, *Visualization of three-dimensional delaunay meshes*, Algorithmica **19** (2003), no. 1-2, 114–128.

[KP03] Dave Knott and Dinesh K. Pai, *CInDeR: Collision and interference detection in real-time using graphics hardware*, Proceedings of Graphics Interface, 2003, pp. 73–80.

[KQE04] Martin Kraus, Wei Qiao, and David S. Ebert, *Projecting tetrahedra without rendering artifacts*, Proceedings of IEEE Visualization, IEEE Computer Society, 2004, pp. 27–34.

[Krü06] Jens Krüger, *A GPU Framework for Interactive Simulation and Rendering of Fluid Effects*, Ph.D. thesis, Technische Universität München, 2006.

[KSE04] Thomas Klein, Simon Stegmaier, and Thomas Ertl, *Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids*, Proceedings of Pacific Graphics, 2004, pp. 186–195.

[KW03a] Jens Krüger and Rüdiger Westermann, *Acceleration techniques for GPU-based volume rendering*, Proceedings of IEEE Visualization, 2003, pp. 38–45.

[KW03b] Jens Krüger and Rüdiger Westermann, *Linear algebra operators for GPU implementation of numerical algorithms*, ACM Transactions on Graphics **22** (2003), no. 3, 908–916.

[KW05] Peter Kipfer and Rüdiger Westermann, *GPU construction and transparent rendering of iso-surfaces*, Proceedings of Vision, Modeling and Visualization, 2005, pp. 241–248.

[LAM05] Thomas Larsson and Tomas Akenine-Möller, *A dynamic bounding volume hierarchy for generalized collision detection*, Proceedings of 2nd Workshop On Virtual Reality Interaction and Physical Simulation, 2005, pp. 91–100.

[LC87] William E. Lorensen and Harvey E. Cline, *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, Proceedings of SIGGRAPH, 1987, pp. 163–169.

[LCN99] Jean-Christophe Lombardo, Marie-Paule Cani, and Fabrice Neyret, *Real-time collision detection for virtual surgery*, Proceedings of the Computer Animation, 1999, pp. 82–91.

[LG98] Ming C. Lin and Stefan Gottschalk, *Collision detection between geometric models: A survey*, Proceedings of IMA Conference on Mathematics of Surfaces, 1998, pp. 37–56.

[LM04] Ming C. Lin and Dinesh Manocha, *Collision and proximity queries*, Handbook of Discrete and Computational Geometry, 2nd Ed. (J. E. Goodman and J. O'Rourke, eds.), Chapman and Hall/CRC Press, New York, 2004, pp. 787–807.

[LPFH01] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe, *Real-time fur over arbitrary surfaces*, Proceedings of the Symposium on Interactive 3D graphics, 2001, pp. 227–232.

[LSW99] C. Lennerz, E. Schömer, and T. Warken, *A framework for collision detection and response*, Proceedings of 11th European Simulation Symposium, 1999, pp. 309–314.

[LTW95] Yuencheng Lee, Demetri Terzopoulos, and Keith Walters, *Realistic modeling for facial animation*, Proceedings of SIGGRAPH, 1995, pp. 55–62.

[MBTF03] Neil Molino, Robert Bridson, Joseph Teran, and Ronald Fedkiw, *A crystalline, red green strategy for meshing highly deformable objects with tetrahedra*, Proceedings of the 12th International Meshing Roundtable, 2003, pp. 103–114.

[MC95] Brian Mirtich and John Canny, *Impulse-based simulation of rigid bodies*, Proceedings of the symposium on Interactive 3D graphics, 1995, pp. 181–188.

[McN83] J. M. McNamee, *A sparse matrix package - part II: Special cases*, ACM Transactions on Mathematical Software (1983), 344–345.

[MDM+02] Matthias Müller, Julie Dorsey, Leonard McMillan, Robert Jagnow, and Barbara Cutler, *Stable real-time deformations*, Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2002, pp. 49–54.

[MG04] Matthias Müller and Markus Gross, *Interactive virtual materials*, Proceedings of Graphics Interface, 2004, pp. 239–246.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard, *Cg: A system for programming graphics hardware in a C-like language*, Proceedings of SIGGRAPH, 2003, pp. 896–907.

[Mic02] Microsoft, *DirectX9 SDK*, http://www.microsoft.com/DirectX, 2002.

[MKE03] Johannes Mezger, Stefan Kimmerle, and Olaf Etzmuß, *Hierarchical techniques in collision detection for cloth animation*, Proceedings of WSCG, 2003, pp. 322–329.

[MKN+04] Matthias Müller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus Gross, and Marc Alexa, *Point based animation of elastic, plastic and melting objects*, Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2004, pp. 141–151.

[ML03] Cesar Mendoza and Christian Laugier, *Simulating soft tissue cutting using finite element models*, Proceedings of IEEE International Conference on Robotics and Automation, 2003, pp. 1109–1114.

[MOK95] Karol Myszkowski, Oleg G. Okunev, and Tosiyasu L. Kunii, *Fast collision detection between complex solids using rasterizing graphics hardware*, The Visual Computer **11** (1995), no. 9, 497–511.

[Möl97] Tomas Möller, *A fast triangle-triangle intersection test*, Journal of Graphics Tools **2** (1997), no. 2, 25–30.

[NMK+05] Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman, and Mark Carlson, *Physically based deformable models in computer graphics*, Proceedings of Eurographics, 2005, pp. 71–94.

[Ope04] OpenGL, *Pixelbuffer objects*, http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt, 2004.

[Pas04] Valerio Pascucci, *Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping*, Proceedings of IEEE TCVG Symposium on Visualization, 2004, pp. 293–300.

[PB81] Stephen M. Platt and Norman I. Badler, *Animating facial expressions*, Proceedings of SIGGRAPH (New York, NY, USA), ACM Press, 1981, pp. 245–252.

[PBMH02] Timothy Purcell, Ian Buck, William R. Mark, and Pat Hanrahan, *Ray tracing on programmable graphics hardware*, ACM Transactions on Graphics **21** (2002), no. 3, 703–712.

[PDA00] Guillaume Picinbono, Herve Delingette, and Nicholas Ayache, *Real-time large displacement elasticity for surgery simulation: Non-linear tensor-mass model*, Proceedings of MICCAI, 2000, pp. 643–652.

[PDA01] ______, *Non-linear and anisotropic elastic soft tissue models for medical simulation*, Proceedings of IEEE International Conference on Robotics and Automation, 2001, pp. 1370–1375.

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe, *Lapped textures*, Proceedings of SIGGRAPH, 2000, pp. 465–470.

[PG95] I. J. Palmer and R. L. Grimsdale, *Collision detection for animation using sphere-trees*, Computer Graphics Forum **14** (1995), no. 2, 105–116.

[Pro95] Xavier Provot, *Deformation constraints in a mass-spring model to describe rigid cloth behavior*, Proceedings of Graphics Interface, 1995, pp. 147–154.

[PV05] Ali Pinar and Virginia Vassilevska, *Finding nonoverlapping substructures of a sparse matrix*, Electronic Transactions on Numerical Analysis **21** (2005), 107–124.

[RDG+04] Frank Reck, Carsten Dachsbacher, Roberto Grosso, Günther Greiner, and Marc Stamminger, *Realtime isosurface extraction with graphics hardware*, Proceedings of Eurographics (Short Presentations), 2004, pp. 33–36.

[RE03] Stefan Röttger and Thomas Ertl, *Cell projection of convex polyhedra*, Proceedings Eurographics/IEEE TVCG Workshop on Volume Graphics, 2003, pp. 103–107.

[RKC02] S. Redon, A. Kheddar, and S. Coquillart, *Fast continuous collision detection between rigid bodies*, Proceedings of Eurographics, 2002, pp. 279–288.

[RKE00] Stefan Röttger, Martin Kraus, and Thomas Ertl, *Hardware-accelerated volume and isosurface rendering based on cell-projection*, Proceedings of IEEE Visualization, 2000, pp. 109–116.

[RMS92] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider, *Interactive inspection of solids: Cross-sections and interferences*, Proceedings of SIGGRAPH, 1992, pp. 353–360.

[RNO88] C. C. Rankin and B. Nour-Omid, *The use of projectors to improve finite element performance*, Computer & Structures **30** (1988), 257–267.

[RNP01] Arne Radetzky, Andreas Nürnberger, and Dietrich P. Pretschner, *The simulation of elastic tissues in virtual medicine using neuro-fuzzy systems*, Proceedings of Medical Imaging (SPIE Proceedings Volume 3335), 2001, pp. 399–409.

[SBM94] Clifford Stein, Barry Becker, and Nelson Max, *Sorting and hardware assisted rendering for volume visualization*, Proceedings of ACM Symposium on Volume Visualization, 1994, pp. 83–90.

[Sch97] Joachim Schöberl, *NETGEN - an advancing front 2D/3D-mesh generator based on abstract rules*, Computing and Visualization in Science **1** (1997), no. 1, 41–52.

[SGG+06] Avneesh Sud, Naga Govindaraju, Russell Gayle, Ilknur Kabul, and Dinesh Manocha, *Fast proximity computation among deformable models using discrete voronoi diagrams*, ACM Transaction on Graphics **25** (2006), no. 3, 1144–1153.

[SGW07] Thomas Schiwietz, Joachim Georgii, and Rüdiger Westermann, *Freeform image*, Proceedings of Pacific Graphics, 2007.

[Si04] Hang Si, *TetGen, a quality tetrahedral mesh generator and three-dimensional delaunay triangulator, v1.3 user's manual*, Tech. Report 9, Weierstrass Institute for Applied Analysis and Stochastics, Berlin, 2004.

[SJP05] Le-Jeng Shiue, Ian Jones, and Jörg Peters, *A realtime GPU subdivision kernel*, ACM Transactions on Graphics **24** (2005), no. 3, 1010–1015.

[SKTK95] A. Smith, Y. Kitamura, H. Takemura, and F. Kishino, *A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion*, Proceedings of the IEEE Virtual Reality Annual International Symposium, 1995, p. 136.

[SM97] Claudio T. Silva and Joseph S. B. Mitchell, *The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids*, IEEE Transactions on Visualization and Computer Graphics **4** (1997), no. 2, 142–157.

[SMK96] Claudio T. Silva, Joseph S. B. Mitchell, and Arie E. Kaufman, *Fast Rendering of Irregular Grids*, Proceedings of ACM Symposium on Volume Visualization, 1996, pp. 15–23.

[SMW98] Claudio T. Silva, Joseph S. B. Mitchell, and Peter L. Williams, *An exact interactive time visibility ordering algorithm for polyhedral cell complexes*, Proceedings of the IEEE symposium on Volume Visualization, 1998, pp. 87–94.

[ST90] Peter Shirley and Allan Tuchman, *A polygonal approximation to direct scalar volume rendering*, ACM SIGGRAPH Computer Graphics **24** (1990), no. 5, 63–70.

[SYBF06] Lin Shi, Yizhou Yu, Nathan Bell, and Wei-Wen Feng, *A fast multigrid algorithm for mesh deformation*, ACM Trans. Graph. **25** (2006), no. 3, 1108–1117.

[TBNF03] Joseph Teran, Silvia Blemker, Victor Ng Thow Hing, and Ronald Fedkiw, *Finite volume methods for the simulation of skeletal muscle*, Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2003, pp. 68–74.

[TCR03] Sivan Toledo, Doron Chen, and Vladimir Rotkin, *Taucs: A library of sparse linear solvers*, http://www.tau.ac.il/∼stoledo/taucs, 2003.

[TE05] Eduardo Tejada and Thomas Ertl, *Large Steps in GPU-based Deformable Bodies Simulation*, Simulation Practice and Theory. Special Issue on Programmable Graphics Hardware **13** (2005), no. 9, 703–715.

[TF88] Demetri Terzopoulos and Kurt Fleischer, *Modeling inelastic deformation: Viscoelasticity, plasticity, fracture*, Proceedings of SIGGRAPH, 1988, pp. 269–278.

[THMG04] Matthias Teschner, Bruno Heidelberger, Matthias Müller, and Markus Gross, *A versatile and robust model for geometrically complex deformable solids*, Proceedings of Computer Graphics International, 2004, pp. 312–319.

[TKH+05] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, *Collision detection for deformable objects*, Computer Graphics Forum **24** (2005), no. 1, 61–81.

[Tom02] Kano Tomohide, *Dynamic fur using smartshaders*, http://ati.amd.com/developer/indexsc.html, 2002.

[TPBF87] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer, *Elastically deformable models*, Proceedings of SIGGRAPH, 1987, pp. 205–214.

[TW88] Demetri Terzopoulos and Andrew Witkin, *Physically based models with rigid and deformable components*, IEEE Computer Graphics & Applications **8** (1988), no. 6, 41–51.

[vdB97] Gino van den Bergen, *Efficient collision detection of complex deformable models using AABB trees*, Journal of Graphics Tools **2** (1997), no. 4, 1–14.

[VT00] Pascal Volino and Nadia Magnenat Thalmann, *Accurate collision response on polygonal meshes*, Proceedings of the Computer Animation, 2000, p. 154.

[WBG07] Martin Wicke, Mario Botsch, and Markus Gross, *A finite element method on convex polyhedra*, Proceedings of Eurographics, 2007.

[WDGT01] Xunlei Wu, Michael S. Downes, Tolga Goktekin, and Frank Tendick, *Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes*, Proceedings of Eurographics, 2001, pp. 349–358.

[WE97] Rüdiger Westermann and Thomas Ertl, *The VSBUFFER: Visibility Ordering unstructured Volume Primitives by Polygon Drawing*, Proceedings of IEEE Visualization, 1997, pp. 35–43.

[WE98] Rüdiger Westermann and Thomas Ertl, *Efficiently using graphics hardware in volume rendering applications*, Proceedings of SIGGRAPH, 1998, pp. 169–177.

[WE01] Manfred Weiler and Thomas Ertl, *Hardware-software-balanced resampling for the interactive visualization of unstructured grids*, Proceedings of IEEE Visualization, 2001, pp. 199–206.

[Wes01] Rüdiger Westermann, *The rendering of unstructured grids revisited*, Proceedings of the EG/IEEE TCVG Symposium on Visualization, 2001.

[WFKH07] Ingo Wald, Heiko Friedrich, Aaron Knoll, and Charles D Hansen, *Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes*, IEEE Transactions on Visualization and Computer Graphics (2007).

[WGTG96] Jane Wilhelms, Allen Van Gelder, Paul Tarantino, and Jonathan Gibbs, *Hierarchical and parallelizable direct volume rendering for irregular and multiple grids*, Proceedings of IEEE Visualization, 1996, pp. 57–63.

[Wil92] Peter L. Williams, *Visibility Ordering Meshed Polyhedra*, ACM Transactions on Graphics **11** (1992), no. 2, 103–126.

[Wil98] Ed Wilson, *Dynamic analysis by numerical integration: Normally, for earthquake loading direct numerical integration is very slow*, Tech. report, CSI: Computer & Structures, Inc., 1998.

[WKE02] Manfred Weiler, Martin Kraus, and Thomas Ertl, *Hardware-based view-independent cell projection*, Proceedings of the IEEE Symposium on Volume Visualization and Graphics, 2002, pp. 13–22.

[WKME03a] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl, *Hardware-based ray casting for tetrahedral meshes*, Procceedings of IEEE Visualization, 2003, pp. 333–340.

[WKME03b] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl, *Hardware-based view-independent cell projection*, IEEE Transactions on Visualization and Computer Graphics **9** (2003), no. 2, 163–175.

[WMFC02] Brian Wylie, Kenneth Moreland, Lee Ann Fisk, and Patricia Crossno, *Tetrahedral projection using vertex shaders*, Proceedings of the IEEE symposium on Volume visualization and Graphics, 2002, pp. 7–12.

[WMS98] Peter L. Williams, Nelson L. Max, and Clifford M. Stein, *A high accuracy volume renderer for unstructured data*, IEEE Transactions on Visualization and Computer Graphics **4** (1998), no. 1, 37–54.

[WT04] Xunlei Wu and Frank Tendick, *Multigrid integration for interactive deformable body simulation*, Proceedings of International Symposium on Medical Simulation, 2004, pp. 92–104.

[YJH+01] Wen-Chun Yeh, Yung-Ming Jeng, Hey-Chi Hsu, Po-Ling Kuo, Meng-Lin Li, Pei-Ming Yang, Po Huang Lee, and Pai-Chi Li, *Young's modulus measurements of human liver and correlation with pathological findings*, Proceedings of IEEE Ultrasonics Symposium, 2001, pp. 1233–1236.

[YRL+96] Roni Yagel, David M. Reed, Asish Law, Po-Wen Shih, and Naeem Shareef, *Hardware assisted volume rendering of unstructured grids by incremental slicing*, Proceedings of the Symposium on Volume Visualization, 1996, pp. 55–62.

[YZ05] Raphael Yuster and Uri Zwick, *Fast sparse matrix multiplication*, ACM Trans. Algorithms **1** (2005), no. 1, 2–13.

[ZC99] Yan Zhuang and John Canny, *Real-time simulation of physically realistic global deformation*, Proceedings of IEEE Visualization, 1999, pp. 270–273.

[ZTTS06] Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel, *On-the-fly point clouds through histogram pyramids*, Proceedings of Vision, Modeling and Visualization, 2006, pp. 137–144.