Institut für Informatik
Technische Universität München

Computational Steering for Implant Planning in Orthopedics

Christian Dick

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.

Die Dissertation wurde am 24.11.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.01.2012 angenommen.

Abstract

In this thesis, I present novel, sophisticated simulation and visualization techniques for the realization of a computational steering environment for preoperative implant planning in orthopedics. In particular, I present real-time techniques for the physics-based simulation of deformable objects, for the visualization of stress tensor fields in elastic bodies, and for the visualization of the spatial distances between two objects. The presented methods are specifically designed to be executed on the GPU to exploit the GPU's massive computing power and memory bandwidth. In addition, this thesis addresses the efficient simulation of cuts in deformable objects.

First, I present a novel real-time multigrid finite element method for the physics-based simulation of deformable objects that is realized entirely on the GPU. This method is based on linear elasticity and the corotational formulation of strain. The governing system of partial differential equations is discretized by using hexahedral finite elements aligned on a uniform Cartesian grid and the implicit Newmark time integration scheme. To efficiently solve the arising linear system of equations, a geometric multigrid method is employed. I present a specific restructuring of the multigrid scheme that enables an efficient GPU implementation using the CUDA computing API. Compared to an optimized and parallelized CPU implementation, significant speed-ups are demonstrated.

Second, I present a novel approach for the simulation of cuts in deformable objects that is based on a geometric multigrid solver to achieve high computational efficiency. This approach is built upon an octree discretization of the simulation domain. Finite elements that are cut are regularly subdivided, until a finest level is reached. At the finest level, face adjacent elements are disconnected. I present a novel strategy to embed complex topology changes induced by cuts into the multigrid hierarchy, and I give a detailed analysis to demonstrate the superior performance of the proposed solver in comparison to alternative numerical solution methods.

Third, I present novel methods for the visualization of stress tensor fields arising in elasticity simulations. In particular, I present a method for the visualization of principal stresses, and a comparative visualization method that shows the differences between two stress tensor fields. Both methods are based on combining volume rendering and the rendering of semi-transparent lines to visualize stress magnitudes and stress directions, and are realized entirely on the GPU to enable the visualization of dynamically changing stress tensor fields at interactive frame rates.

Fourth, I present novel methods for the visualization of the spatial distances between two objects that support precise object positioning in interactive scenarios. These methods are based on introducing additional geometric structures into the visualization. In the first approach cylindrical glyphs are used, which smoothly adapt their shape and color to changing distances. In the second approach, a set of parallel slices is added, and color coding on surfaces is used to depict distances. To evaluate the effectiveness of the proposed methods, a user study has been performed.

Fifth, I demonstrate the use of the developed simulation and visualization techniques for the realization of the first computational steering environment for preoperative implant planning in orthopedics. This environment allows the surgeon to interactively determine the optimal implant shape, size, and position according to the prediction of the stress distribution in the patient-specific bone after insertion of the implant. The selected implant should replicate the preoperative stress distribution in order to avoid the degeneration of bone tissue, which can lead to the fracturing of the bone or the loosening of the implant.

Zusammenfassung

In dieser Arbeit stelle ich neuartige, hochentwickelte Simulations- und Visualisierungstechniken für die Realisierung einer Computational-Steering-Umgebung für die präoperative Implantatplanung in der Orthopädie vor. Insbesondere präsentiere ich Echtzeittechniken für die physikbasierte Simulation von deformierbaren Objekten, für die Visualisierung von Spannungstensorfeldern in elastischen Körpern, und für die Visualisierung der räumlichen Abstände zwischen zwei Objekten. Die vorgestellten Methoden sind speziell dafür entwickelt auf der GPU ausgeführt zu werden, um die massive Rechenleistung und Speicherbandbreite der GPU auszunutzen. Zusätzlich behandelt diese Arbeit die effiziente Simulation von Schnitten in deformierbaren Objekten.

Erstens stelle ich eine neuartige Echtzeit-Mehrgitter-Finite-Elemente-Methode für die physikbasierte Simulation von deformierbaren Objekten vor, die vollständig auf der GPU realisiert ist. Diese Methode basiert auf linearer Elastizität und der corotierten Formulierung der Dehnung. Das bestimmende System von partiellen Differentialgleichungen wird durch Verwendung von hexaedrischen, an einem uniformen kartesischen Gitter ausgerichteten, finiten Elementen und dem impliziten Newmark-Zeitintegrationsschema diskretisiert. Um effizient das auftretende lineare Gleichungssystem zu lösen, wird eine geometrische Mehrgittermethode verwendet. Ich stelle eine spezielle Restrukturierung des Mehrgitterschemas vor, die eine effiziente GPU-Implementierung unter Verwendung des CUDA Computing APIs ermöglicht. Verglichen mit einer optimierten und parallelisierten CPU-Implementierung werden signifikante Geschwindigkeitssteigerungen demonstriert.

Zweitens stelle ich einen neuartigen Ansatz für die Simulation von Schnitten in deformierbaren Objekten vor, der auf einem geometrischen Mehrgitterlöser basiert, um hohe Berechnungseffizienz zu erreichen. Dieser Ansatz basiert auf einer Octree-Diskretisierung der Simulationsdomäne. Finite Elemente die geschnitten werden, werden regulär unterteilt, bis ein feinstes Level erreicht ist. Auf dem feinsten Level werden flächenadjazente Elemente getrennt. Ich stelle eine neuartige Strategie vor, um durch Schnitte induzierte, komplexe Topologieänderungen in die Mehrgitterhierarchie einzu-

betten, und ich liefere eine detaillierte Analyse, um die höhere Leistung des vorgeschlagenen Lösers im Vergleich zu alternativen numerischen Lösungsmethoden zu zeigen.

Drittens stelle ich neuartige Methoden für die Visualisierung von Spannungstensorfeldern vor, die in Elastizitätssimulationen auftreten. Insbesondere stelle ich eine Methode für die Visualisierung von Hauptspannungen vor, und eine vergleichende Visualisierungsmethode, die die Unterschiede zwischen zwei Spannungstensorfeldern zeigt. Beide Methoden basieren auf dem Kombinieren von Volume Rendering und dem Rendern von semi-transparenten Linien, um Spannungsbeträge und Spannungsrichtungen darzustellen, und sind komplett auf der GPU realisiert, um die Visualisierung von sich dynamisch ändernden Spannungstensorfeldern bei interaktiven Frameraten zu ermöglichen.

Viertens stelle ich neuartige Methoden für die Visualisierung der räumlichen Abstände zwischen zwei Objekten vor, die eine präzise Objektpositionierung in interaktiven Szenarien unterstützen. Diese Methoden basieren auf dem Einführen von zusätzlichen geometrischen Strukturen in die Visualisierung. Im ersten Ansatz werden zylindrische Glyphen verwendet, die gleichmäßig ihre Form und Farbe an sich ändernde Abstände anpassen. Im zweiten Ansatz wird eine Menge von parallelen Schnitten hinzugefügt und Farbkodierung auf Oberflächen verwendet, um Abstände darzustellen. Um die Effektivität der vorgeschlagenen Methoden zu evaluieren, wurde eine Benutzerstudie durchgeführt.

Fünftens demonstriere ich die Verwendung der entwickelten Simulations- und Visualisierungstechniken zur Realisierung der ersten Computational-Steering-Umgebung für die präoperative Implantatplanung in der Orthopädie. Diese Umgebung ermöglicht dem Chirurgen interaktiv die optimale Implantatform, -größe und -position gemäß der Vorhersage der Spannungsverteilung im patientenspezifischen Knochen nach Einsetzen des Implantats zu bestimmen. Das ausgewählte Implantat sollte die präoperative Spannungsverteilung nachbilden, um die Rückbildung von Knochengewebe zu vermeiden, was zum Bruch des Knochens oder zur Lockerung des Implantats führen kann.

Acknowledgements

I want to thank all people who have helped to make this work possible.

First and foremost, I want to thank my advisor, Prof. Dr. Rüdiger Westermann, for getting me into computer graphics and visualization, and for providing me with an excellent academic education and training during my time at his chair. His advice and support were essential for the success of this work.

I want to thank Dr. Joachim Georgii for introducing me to the basics of elasticity theory, the finite element method, and geometric multigrid solvers. His work on real-time elasticity simulation was the basis for several of the developments presented in this thesis.

Special thanks go to PD Dr. Rainer Burgkart at the Klinikum Rechts der Isar for actively accompanying this work from a medical point of view. His deep interest in this work has always been a great motivation.

The results presented in this thesis have been originally published in a number of research papers that I have co-authored with Joachim Georgii, Rainer Burgkart, and Rüdiger Westermann. I want to thank my co-authors for helping me to obtain these results by contributing their ideas and suggestions.

I also want to thank my current and former colleagues, Stefan Auer, Dr. Kai Bürger, Shunting Cao, Matthäus G. Chajdas, Roland Fraedrich, Raymund Fülöp, Dr. Joachim Georgii, Stefan Hertel, Dr. Polina Kondratieva, Prof. Dr. Martin Kraus, Prof. Dr. Jens Krüger, Hans-Georg Menz, Tobias Pfaffelmoser, Marc Rautenhaus, Florian Reichl, Dr. Thomas Schiwietz, Dr. Jens Schneider, Marc Treib, Mikael Vaaraniemi, and Jun Wu, for many interesting discussions.

I particularly want to thank my parents for always supporting me during my years of study.

Finally, I want to thank the International Graduate School of Science and Engineering (IGSSE) of the Technische Universität München for funding my work by a doctoral scholarship under the project "Computational Steering for Orthopaedics", and all members of this project for fruitful discussions.

Contents

Chapter 1

Introduction

Scientific computing and scientific visualization are complementary disciplines, which are today widely used in sciences, engineering, and medicine.

Scientific computing is concerned with the numerical simulation of complex processes and phenomena, used for example to analyze the aerodynamic resistance of aircrafts and vehicles, to verify the structural integrity of machines and buildings, or to study protein folding or the formation of stars and galaxies.

Scientific visualization provides methods to transform the numerical data obtained from computer simulations or real-world measurements into visual representations. Considering the human's ability to assimilate information and recognize structures and coherences much better and faster from a visual than a numerical representation, scientific visualization facilitates gaining deep insight into the data and the underlying processes. In particular, *interactive* visualization techniques enable the user to visually explore and analyze large and complex data sets by interactively changing the view point, the data selection, and other visualization parameters.

In the traditional simulation and visualization scenario, simulation and visualization are separated processes. The simulation typically runs for several minutes or hours as an offline process, possibly on a supercomputer. The results are stored on disk and are later analyzed in an interactive visualization session. If the analysis reveals that the simulation parameters have to be adapted, the simulation must be run again. In particular, if the goal is to optimize the simulation parameters according to certain specific criteria, a large number of iterations might be required, leading to a rather non-intuitive and time-consuming process.

In contrast, in a *computational steering* or *interactive steering* scenario, simulation and visualization are running simultaneously and tightly coupled. Based on the visualization of the simulation results, the user can interactively change the simulation parameters, and new simulation results reflecting these changes are provided on-the-fly. This enables the user to visually explore the impact of the simulation parameters on the simulation results, and to visually guide the simulation parameters into the right direction.

To enable the interactive steering of a simulation, new simulation results reflecting changes of the simulation parameters must be provided in a very short time interval, ideally in a fraction of a second. This strict requirement limits the considered problem sizes and grid resolutions, depending on the complexity of the simulation, the efficiency of the underlying algorithms, and the performance of the employed hardware. Due to this reason, today for most applications the traditional simulation and visualization scenario is used, whereas computational steering is only employed for a small number of selected applications [SM99].

1.1 Goals

The primary goal of this thesis is the development of a computational steering environment for preoperative implant planning in orthopedics, with a focus on total hip joint replacement. The purpose of this environment is to enable the surgeon to interactively determine a patient-specific optimal implant according to the geometrical *and* biomechanical properties of the patient-specific bone and the implants.

Besides the geometrical properties, which restrict the implant's shape, size, and position for example by the requirements that the implant has to fit into the cortical bone shell and that the joint rotation center has to be preserved, the consideration of the biomechanical properties during implant selection is important due to the physiological reaction of bone tissue to changes in the bone stress distribution. Following a natural optimization principle to provide high mechanical stability at low weight, an increase or decrease of bone stresses leads to bone formation or resorption, respectively. The insertion of an implant changes the stress distribution in that stresses are bypassed by the implant, leading to a reduction of the stresses in certain regions of the bone (*stress shielding*). As a consequence of the bone's adaptation to changes in the stress patterns, a significant reduction of bone stresses leads to cortical thinning and increased porosity of the bone, which can finally lead to bone fracture or loosening of the implant. Therefore, during preoperative planning an implant should be selected that minimizes stress shielding, i.e., that leads to a stress distribution that is close to the preoperative, physiological stress distribution.

Existing implant planning systems, as well as the clinical state-of-the-art approach of using 2D X-rays and transparent template sheets with the outlines of the implants are purely geometry-based, such that the surgeon has to consider the biomechanical properties according to his own subjective medical experience.

The goal of this thesis is to address this limitation by developing an implant planning system that allows the surgeon to interactively investigate the effect of different implant shapes, sizes, and positions on the stress distribution in the bone. This system shall provide a virtual 3D environment for interactive implant selection and positioning, and shall quasi instantaneously compute and visualize the stress distribution resulting from the current implant configuration. In this way, the system shall allow for an interactive, visually guided selection of a patient-specific optimal implant that minimizes stress shielding. The system shall run on a standard desktop PC or workstation. This platform is particularly attractive due to its virtually unlimited availability.

The realization of the described computational steering environment requires real-time techniques for the physically accurate simulation of the internal stresses in the bone, and for the visualization of these stresses in such a way as to allow for a clear rating of an implant with respect to stress shielding. Furthermore, to enable an intuitive navigation and precise placement of the implant, visualization methods that allow for a quick and intuitive perception of the spatial relationships, i.e., the distances between bone and implant, are required. The specific challenges in developing these techniques result from the very high computational complexity of the simulation in combination with the strict time requirements, from the difficulty of considering a movable implant with specific mechanical properties in the simulation, and from the complexity of the information to be visualized. In particular, we have to deal with nested, dynamically changing surface and volume structures, leading to the problems of occlusion, visual cluttering, and limited perception of spatial relationships.

From a more general point of view, the goal of this thesis also is to demonstrate to what extent real-time simulation and computational steering are possible on today's desktop PCs and workstations, taking into account recent advances in hardware and numerics.

First, the introduction of multi-core CPUs and well as the utilization of graphics processing units (GPUs) for serious general purpose computing have led to the advent of parallel computing on desktop PC hardware. Initially dedicated solely to interactive 3D graphics rendering, the increased programmability and in particular the support of double precision floating point arithmetic with recent GPU generations enable using the GPU as a general purpose many-core processor, which exhibits a tremendous floating point performance and memory bandwidth.

Second, geometric multigrid solvers have proven to be very efficient solvers for the particular type of large, sparse linear systems of equations arising from the discretization of partial differential equations in continuum mechanics. Based on the observation that standard iterative solvers like Gauss-Seidel or Jacobi relaxation typically stall after a few iteration steps, the basic idea of multigrid is to couple multiple scales by employing a hierarchy of successively coarser grids to achieve improved convergence, resulting in a solver that exhibits asymptotically linear complexity in the number of unknowns. In particular, multigrid solvers typically exhibit a significantly better convergence rate than other iterative solvers, which makes them very attractive for computational steering and real-time simulation as they can provide a fast, approximate solution in a given, limited time interval.

1.2 Contributions

Real-time elasticity simulation using CUDA. Building upon previous work on geometric multigrid solvers for the real-time, physics-based simulation of deformable bodies on the CPU [GW06], we present the first multigrid approach for deformable body simulation that is realized entirely on the GPU. Our method is based on linear elasticity combined with the corotational formulation of strain, and the governing system of partial differential equations is discretized by using finite elements and the implicit Newmark time integration scheme. We propose a specific algorithmic restructuring of the multigrid scheme to expose a sufficient amount of fine-grained parallelism and thus to effectively exploit the GPU's massively parallel architecture via the CUDA computing API. Key to our approach is a regular hexahedral discretization of the simulation domain, which leads to a numerical stencil of the same shape at each vertex, and thus facilitates parallel processing of data elements using the same execution paths as well as coalescing of memory accesses into larger memory transactions. In comparison to an optimized parallel implementation on the CPU, we demonstrate performance gains of a factor of 27 with respect to a single CPU core and 4 with respect to 8 CPU cores. Our approach thus enables the real-time, physics-based simulation of deformable bodies at unprecedented simulation update rates and grid resolutions on standard desktop PC hardware.

Stress tensor field visualization. To allow for a clear rating of an implant configuration with respect to stress shielding, we have developed novel visualization methods for 3D stress tensor fields. In particular, we present a visualization method for the principal stresses, and a comparative visualization method that shows the differences between two stress tensor fields. The latter approach particularly supports the surgeon in finding an implant configuration that minimizes stress shielding, in that this method can directly show the differences between the physiological stress distribution and the stress distribution resulting from a certain implant configuration. Our visualization techniques are based on combining volume rendering for the visualization of stress magnitudes with the rendering of line segments for the visualization of stress directions. To allow for a detailed stress analysis in certain regions of interest while preserving spatial context information, we have integrated a focus+contex approach. We have realized our methods on the GPU to enable the interactive visualization of dynamically changing data within a computational steering environment.

Distance visualization. To enable a precise positioning of the implant, quick and intuitive perception of the spatial relationships between bone and implant is mandatory. To this end, we have developed two novel approaches for the visualization of the spatial distances between two objects in interactive scenarios. Based on the observation that we are used to interpret a distance value always with respect to two reference points, and that exactly these reference points are missing in distance visualization approaches based on pure color coding on surfaces, we introduce additional geometric structures into the visualization that bridge the space between the two objects. In particular, we use cylindrical glyphs that smoothly adapt their shape and color to changing distances, or a set of parallel slices in combination with color coding on surfaces. To the best of our knowledge, this is the first time that distances in an interactive scenario are visualized using a method other than pure color coding. To validate the effectiveness of the methods, a user study has been performed.

Computational steering in orthopedics. Based on the described simulation and visualization methods, we have developed a computational steering environment for preoperative implant planning in orthopedics, which for the first time allows the surgeon to interactively investigate the effect of different implant shapes, sizes, and positions on the stress distribution in the bone. The computation of the stress distribution is based on a finite element model of the bone and the implant. We first derive a finite element model of the bone from patient-specific computed tomography (CT) data, and then incorporate the implant into this model by assigning the material properties of the implant to the elements that are covered by the implant. To handle a movable implant in a computational steering environment, we have developed a novel GPU-based voxelization algorithm to efficiently detect these elements. Using our GPU-based method for deformable body simulation (reduced to the simulation of the static behavior of a linear elastic body) to compute the internal stresses in the bone and implant, we achieve simulation rates of more than 2 updates per second at a resolution of one hexahedral trilinear element per CT voxel (corresponding to 734,000 hexahedral elements).

Simulation of cuts in deformable objects. Separated from the specific application of computational steering for preoperative implant planning, this thesis also addresses the efficient simulation of cuts in deformable objects, which is required for example in virtual surgery simulations. We present the first cutting approach that is based on a computationally efficient geometric multigrid solver. The challenge in combining cutting and multigrid is to incorporate cuts into the coarse grids of the multigrid hierarchy, which to the best of our knowledge has not been addressed yet. Due to this reason, existing cutting approaches resort to less efficient "black box" solvers, such as conjugate gradient or Cholesky solvers. In contrast to these approaches, our method is characterized by a close intertwining of the cutting procedure and the numerical solver. In particular, we use a hexahedral octree finite element discretization that adaptively refines along cuts and the surface of the object, enabling thin and complicated cuts. Cuts are modeled by separating elements along cell faces at a certain finest octree level. The semi-regular hexahedral grid facilitates a fast and robust construction of a nested multigrid hierarchy, and thus enables to efficiently rebuild this hierarchy when the object has been cut. To incorporate complicated topologies into the multigrid hierarchy, we propose a novel strategy that is based on the duplication of cells on the coarse grids to represent separated material parts. Another challenge is the incorporation of cuts into the render surface such that this surface is topologically consistent with the finite element model. To achieve this, we have adapted the splitting cubes algorithm to reconstruct a smooth render surface directly from the finite element model. We present a detailed performance analysis of our approach, and demonstrate significant performance gains over alternative numerical solution methods.

1.3 Publications and Awards

The research results presented in this thesis have been originally published in the following peer-reviewed conference papers and journal articles (listed in chronological order):

1. C. Dick, J. Georgii, R. Burgkart, and R. Westermann, *Computational steering for patient-specific implant planning in orthopedics*, Proc. Eurographics Workshop on Visual Computing for Biomedicine, 2008, pp. 83–92 [DGBW08].

2. C. Dick, J. Georgii, R. Burgkart, and R. Westermann, *A 3D simulation system for*

hip joint replacement planning, Proc. World Congress on Medical Physics and Biomedical Engineering, IFMBE Proceedings, vol. 25/IV, 2009, pp. 363–366 [DGBW09a].

3. C. Dick, J. Georgii, R. Burgkart, and R. Westermann, *Stress tensor field visualization for implant planning in orthopedics*, IEEE Transactions on Visualization and Computer Graphics 15 (2009), no. 6, 1399–1406 [DGBW09b].

4. C. Dick, J. Georgii, and R. Westermann, *A real-time multigrid finite hexahedra method for elasticity simulation using CUDA*, Simulation Modelling Practice and Theory 19 (2011), no. 2, 801–816 [DGW11b].

5. C. Dick, J. Georgii, and R. Westermann, *A hexahedral multigrid approach for simulating cuts in deformable objects*, IEEE Transactions on Visualization and Computer Graphics 17 (2011), no. 11, 1663–1675 [DGW11a].

6. C. Dick, R. Burgkart, and R. Westermann, *Distance visualization for interactive 3D implant planning*, IEEE Transactions on Visualization and Computer Graphics 17 (2011), no. 12, 2173–2182 [DBW11].

The paper titled *"Computational steering for patient-specific implant planning in orthopedics"* won the Best Paper Award at VCBM 2008. Furthermore, the author of this thesis was awarded the Karl-Heinz-Höhne MedVis Award 2010 (1. Prize) for the papers titled *"Computational steering for patient-specific implant planning in orthopedics"* and *"Stress tensor field visualization for implant planning in orthopedics"*.

1.4 Structure of this Thesis

In the next chapter, we review the fundamentals of elasticity theory, the finite element method, geometric multigrid, and general purpose GPU computing that are employed in this thesis. In Chapter 3, we present our GPU multigrid approach for the real-time, physics-based simulation of deformable objects. Our multigrid approach for the simulation of cuts in deformable objects is discussed in Chapter 4. In Chapter 5, we present our computational steering environment for implant planning in orthopedics. Note that in this chapter the scalar von Mises stress norm is used for the visualization of the computed stress distributions. Our advanced stress and distance visualization techniques are discussed in Chapters 6 and 7, respectively. In Chapter 8, we summarize our results, and give an outlook on future work.

Chapter 2

Fundamentals

In this thesis, we consider the physically accurate simulation of deformable bodies based on the linear theory of elasticity combined with the corotational formulation of strain. We employ the finite element method and the implicit Newmark time integration scheme for the spatial and time discretization of the governing system of partial differential equations, and we employ an efficient geometric multigrid solver to solve the resulting system of linear equations.

In this chapter, we summarize the fundamentals of elasticity theory (Section 2.1), finite elements (Section 2.2), and multigrid (Section 2.3), and we introduce the notation that is used throughout this thesis. Since many of the methods presented in the following chapters use the GPU to achieve interactive simulation and visualization update rates, we also give a brief introduction to recent NVIDIA GPU architectures and general purpose computing on GPUs (Section 2.4). For additional information, we refer the reader to [Sla02], [Cia88] (elasticity theory), [Bat02], [Bra07] (finite elements), [BHM00], [TOS01] (multigrid), as well as [Mic10], [NVI10b] (Direct3D 10 and CUDA manuals).

2.1 Elasticity Theory

The theory of elasticity is the branch of continuum mechanics that describes how elastic objects deform under the influence of external forces. In continuum mechanics, a material is modeled at all scales as a continuum of material points occupying a certain domain. The physical quantities are continuously distributed over the domain and thus are described by a set of piecewise continuous fields: In the theory of elasticity, the deformation and the displacement vector field describe the positions of the material points, the strain tensor field describes the material's local changes of shape, and the

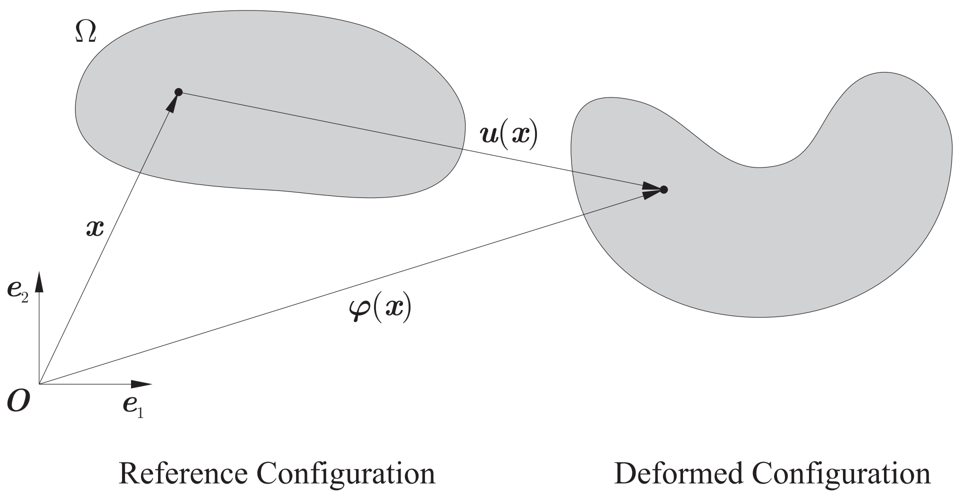


Figure 2.1: *Reference and deformed configuration of an elastic object. The deformation of the object is equivalently described by the deformation $\boldsymbol{\varphi}$ or the displacement vector $\boldsymbol{u}$. (For simplicity, this and the following figures show the 2D case.)*

stress tensor field describes the internal forces acting in the material. The fields are related by a set of field equations, which lead to a system of partial differential equations describing the physical behavior of a deformable body.

2.1.1 Deformation and Displacement Vector

We consider an elastic body that occupies the *reference configuration* $\overline{\Omega}$ in the undeformed state, where $\overline{\Omega}$ is the closure of an open set $\Omega \subset \mathbb{R}^3$, and $\mathbb{R}^3$ identifies the three-dimensional Euclidean space using a Cartesian coordinate system $(\boldsymbol{O}; \boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3)$ (see Figure 2.1). The (static) deformation of the body is described by a vector field $\boldsymbol{\varphi} : \overline{\Omega} \to \mathbb{R}^3$ that specifies the absolute position $\boldsymbol{\varphi}(\boldsymbol{x})$ of each material point, identified by its position $\boldsymbol{x} \in \overline{\Omega}$ in the reference configuration. $\boldsymbol{\varphi}$ is referred to as *deformation*. The deformed body occupies the *deformed configuration* $\{\boldsymbol{\varphi}(\boldsymbol{x})|\boldsymbol{x} \in \overline{\Omega}\}$. Equivalently, the deformation of the body is described by the vector field $\boldsymbol{u} : \overline{\Omega} \to \mathbb{R}^3$, $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{\varphi}(\boldsymbol{x}) - \boldsymbol{x}$, which specifies the relative positions of the material points with respect to the reference configuration. $\boldsymbol{u}$ is referred to as *displacement vector*.

The physical relationships can equivalently be formulated over the reference configuration (*Lagrangian formulation*) or over the deformed configuration (*Eulerian formulation*). In the following, the Lagrangian formulation is used.

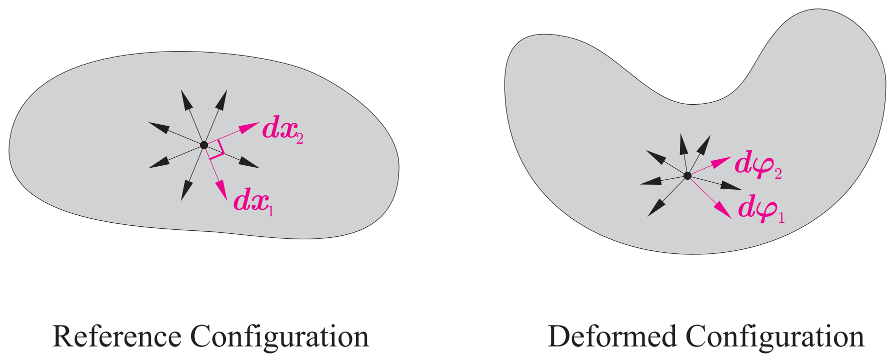


Figure 2.2: *The deformation of the body changes the lengths of the differential material line elements radially emanating from each material point, as well as the angles between initially perpendicular differential material line elements.*

2.1.2 Strain Tensor

Locally, the change of shape of the object is described by the concept of *strain*. Consider a particular material point of the object, as well as the differential material line elements radially emanating from that point (see Figure 2.2). The deformation of the object leads to a relative change of the lengths of the differential material line elements *(normal strains)*, as well as to a change of the angles between initially perpendicular differential material line elements *(shear strains)*.

Consider a particular differential material line element $\boldsymbol{dx}$, which occupies the differential line element $\boldsymbol{d\varphi} = \boldsymbol{F}\boldsymbol{dx}$ in the deformed configuration, where $\boldsymbol{F} = \nabla\boldsymbol{\varphi} = \nabla\boldsymbol{u} + \mathbf{1}_3$ is the *deformation gradient* ($\mathbf{1}_3$ denotes the identity 3×3-matrix). Then,

$$\|\boldsymbol{d\varphi}\|_2^2 = \boldsymbol{dx}^T \boldsymbol{F}^{\mathrm{T}} \boldsymbol{F} \boldsymbol{dx}. \tag{2.1}$$

Let $\boldsymbol{n} = \boldsymbol{dx}/\|\boldsymbol{dx}\|_2$ be the unit vector in the direction of $\boldsymbol{dx}$. Rearranging terms leads to

$$\frac{\|\boldsymbol{d\varphi}\|_2^2 - \|\boldsymbol{dx}\|_2^2}{\|\boldsymbol{dx}\|_2^2} = \boldsymbol{n}^T(\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F} - \mathbf{1}_3)\boldsymbol{n} = \boldsymbol{n}^T 2\boldsymbol{E}\boldsymbol{n}, \tag{2.2}$$

with

$$\boldsymbol{E} = \frac{1}{2}(\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F} - \mathbf{1}_3) = \frac{1}{2}\left(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^{\mathrm{T}} + (\nabla\boldsymbol{u})^{\mathrm{T}}\nabla\boldsymbol{u}\right) \tag{2.3}$$

being a symmetric second order tensor, which is called the *Green-St. Venant strain tensor*. The relative change of the length of the differential material line element in the

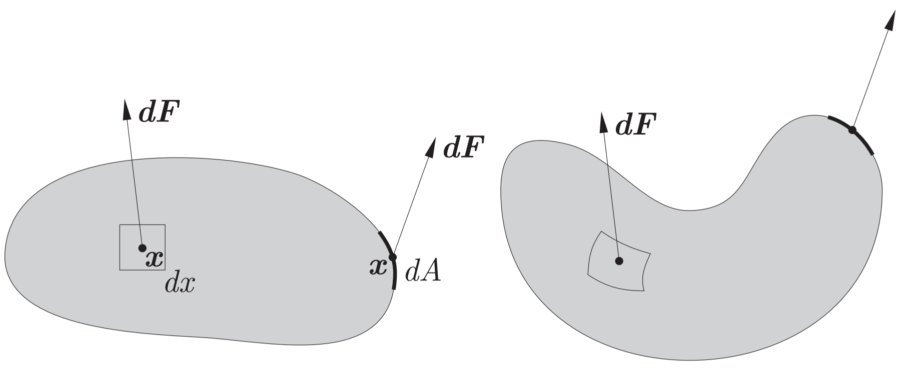


Figure 2.3: *The external body and surface forces are described as force densities $\boldsymbol{dF}/dx$ and $\boldsymbol{dF}/dA$, respectively. Note that volume and area are measured with respect to the reference configuration of the deformable body.*

direction of $\boldsymbol{n}$ is then determined by

$$\frac{\|\boldsymbol{d\varphi}\|_2 - \|\boldsymbol{dx}\|_2}{\|\boldsymbol{dx}\|_2} = \sqrt{\boldsymbol{n}^T 2\boldsymbol{E}\boldsymbol{n} + 1} - 1. \tag{2.4}$$

A positive sign indicates stretching, a negative sign indicates compression.

Now consider two orthogonal differential material line elements $\boldsymbol{dx}_1$ and $\boldsymbol{dx}_2$, which occupy the differential line elements $\boldsymbol{d\varphi}_1 = \boldsymbol{F}\boldsymbol{dx}_1$ and $\boldsymbol{d\varphi}_2 = \boldsymbol{F}\boldsymbol{dx}_2$ in the deformed configuration, respectively, and let $\boldsymbol{n}_i = \boldsymbol{dx}_i/\|\boldsymbol{dx}_i\|_2$ be the unit vector in the direction of $\boldsymbol{dx}_i$. Then,

$$\boldsymbol{d\varphi}_1 \cdot \boldsymbol{d\varphi}_2 = \boldsymbol{dx}_1^{\mathrm{T}} \boldsymbol{F}^{\mathrm{T}} \boldsymbol{F} \boldsymbol{dx}_2. \tag{2.5}$$

Rearranging terms and using $\boldsymbol{dx}_1 \cdot \boldsymbol{dx}_2 = 0$ leads to

$$\cos \angle(\boldsymbol{d\varphi}_1, \boldsymbol{d\varphi}_2) = \frac{(\boldsymbol{n}_1)^{\mathrm{T}} 2\boldsymbol{E}\boldsymbol{n}_2}{\sqrt{\boldsymbol{n}_1^{\mathrm{T}} 2\boldsymbol{E}\boldsymbol{n}_1 + 1}\sqrt{\boldsymbol{n}_2^{\mathrm{T}} 2\boldsymbol{E}\boldsymbol{n}_2 + 1}} = \sin\gamma, \tag{2.6}$$

where $\gamma = \frac{\pi}{2} - \angle(\boldsymbol{d\varphi}_1, \boldsymbol{d\varphi}_2)$ denotes the change of the angle between the initially perpendicular differential material line elements.

The strain tensor thus entirely describes the state of strain at the considered material point, in that it specifies the material's relative change of length (normal strain) for each direction, as well as the material's change of angle (shear strain) between each pair of perpendicular directions. However, the state of strain at a considered material point is

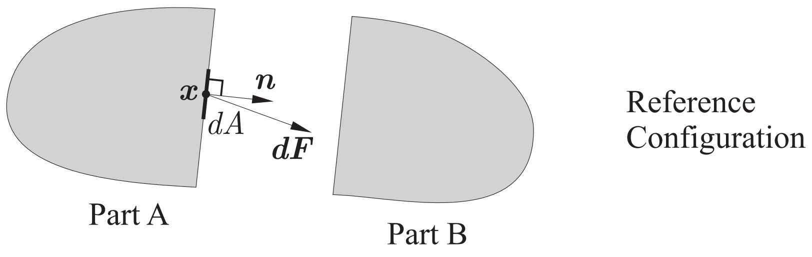


Figure 2.4: *Definition of the nominal stress vector* $\boldsymbol{t}(\boldsymbol{x}, \boldsymbol{n}) = \boldsymbol{dF}/dA$ *as surface force density on an imaginary cutting surface.*

already fully specified by the normal and shear strains for three mutually perpendicular directions. In particular, the entries of the strain tensor $\boldsymbol{E}$ are closely related to the normal and shear strains for the three directions $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, $\boldsymbol{e}_3$ corresponding to the axes of the Cartesian coordinate system. Let ε_i denote the normal strain in the direction of $\boldsymbol{e}_i$, and γ_{ij} denote the shear strain between the directions $\boldsymbol{e}_i$ and $\boldsymbol{e}_j$. Then

$$\varepsilon_i = \sqrt{2E_{ii} + 1} - 1 \qquad \text{and} \qquad \sin\gamma_{ij} = \frac{2E_{ij}}{\sqrt{2E_{ii}+1}\sqrt{2E_{jj}+1}}. \tag{2.7}$$

2.1.3 Body and Surface Forces

We consider the mechanical deformation of an object due the application of external forces. These external forces can be classified into two types: *Body forces*, resulting from force fields such as gravity, and *surface forces*, resulting from the physical contact with another body. The body forces $\boldsymbol{f}_B$ are described as a force density (force per unit volume) $\boldsymbol{f}_B(\boldsymbol{x}) = \frac{\boldsymbol{dF}}{dx}$, where $\boldsymbol{dF}$ denotes the differential force vector that is exerted on the differential material volume element dx around the material point $\boldsymbol{x}$ (see Figure 2.3). Analogously, the surface forces $\boldsymbol{f}_S$ are described as a force density (force per unit area) $\boldsymbol{f}_S(\boldsymbol{x}) = \frac{\boldsymbol{dF}}{dA}$, where $\boldsymbol{dF}$ denotes the differential force vector that is exerted on the differential material surface element dA around the material point $\boldsymbol{x}$.

2.1.4 Stress Tensor

When external forces are applied to a deformable body, internal forces are induced in this body. These internal forces are described by the concept of stress. Consider a material point $\boldsymbol{x}$, and an *imaginary* decomposition of the object into two parts A and B such that the imaginary cutting surface is passing through $\boldsymbol{x}$ (see Figure 2.4). The internal forces that part B exerts on part A are described as a force density (force per unit area) on the imaginary cutting surface: The *nominal stress vector* $\boldsymbol{t}(\boldsymbol{x}, \boldsymbol{n})$ is defined

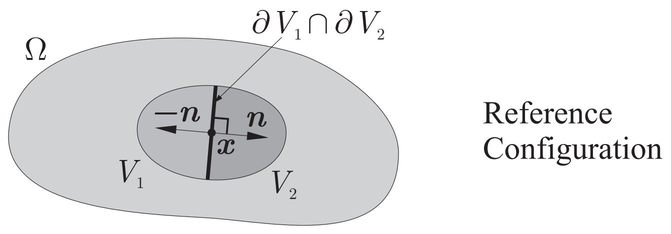


Figure 2.5: *Quantities used in the derivation of* $\boldsymbol{t}(\boldsymbol{x}, -\boldsymbol{n}) = -\boldsymbol{t}(\boldsymbol{x}, \boldsymbol{n})$.

as

$$\boldsymbol{t}(\boldsymbol{x}, \boldsymbol{n}) = \frac{\boldsymbol{dF}}{dA}, \tag{2.8}$$

where $\boldsymbol{dF}$ is the differential force vector that is exerted on the differential material surface element dA around $\boldsymbol{x}$ with unit outer normal $\boldsymbol{n}$.

For any subvolume $V \subset \Omega$, the sum of the forces acting on this subvolume must vanish (*balance of forces*), i.e.,

$$\int\limits_{\partial V} \boldsymbol{t}(., \boldsymbol{n})\, dA + \int\limits_{V} \boldsymbol{f}_B\, dx = \boldsymbol{0}, \tag{2.9}$$

where $\boldsymbol{n}$ is the unit outer normal on ∂V.

Consider a material point $\boldsymbol{x}$, and let $V_1, V_2 \subset \Omega$ be two disjoint subvolumes of the deformable body such that $\boldsymbol{x} \in \partial V_1 \cap \partial V_2$ (see Figure 2.5). Applying the balance of forces to V_1, V_2, and $V_1 \cup V_2$ leads to

$$\int\limits_{\partial V_1 \cap \partial V_2} \boldsymbol{t}(., \boldsymbol{n})\, dA + \int\limits_{\partial V_1 \cap \partial V_2} \boldsymbol{t}(., -\boldsymbol{n})\, dA = \boldsymbol{0}, \tag{2.10}$$

and since V_1 and V_2 are chosen arbitrarily, it follows that

$$\boldsymbol{t}(\boldsymbol{x}, -\boldsymbol{n}) = -\boldsymbol{t}(\boldsymbol{x}, \boldsymbol{n}), \tag{2.11}$$

which is a generalization of Newton's third law.

The state of stress at a specific material point consists of the stress vectors for all orientations of the virtual cutting surface, specified by the unit outer normal $\boldsymbol{n}$. However, the state of stress is already fully specified by the stress vectors for three mutually orthogonal orientations of the surface. Consider the tetrahedral subvolume $V \subset \Omega$

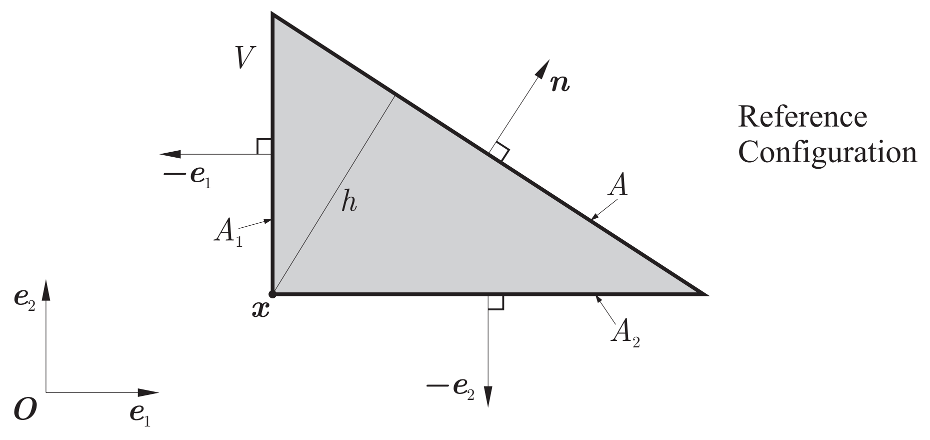


Figure 2.6: *Cauchy tetrahedron used for the derivation of the stress tensor.*

shown in Figure 2.6. Applying the balance of forces to this subvolume leads to

$$A_1\langle \boldsymbol{t}(.,-\boldsymbol{e}_1)\rangle_{A_1} + A_2\langle \boldsymbol{t}(.,-\boldsymbol{e}_2)\rangle_{A_2} + A_3\langle \boldsymbol{t}(.,-\boldsymbol{e}_3)\rangle_{A_3} + A\langle \boldsymbol{t}(.,\boldsymbol{n})\rangle_{A} + V\langle \boldsymbol{f}_B\rangle_V = \boldsymbol{0}. \quad (2.12)$$

Here, $\langle . \rangle_.$ denotes the average in the respective subsurface or subvolume. Using $A_i = (\boldsymbol{e}_i \cdot \boldsymbol{n})\,A = n_i\,A$ and $V = \frac{1}{3}Ah$ leads to

$$n_1\langle \boldsymbol{t}(.,-\boldsymbol{e}_1)\rangle_{A_1} + n_2\langle \boldsymbol{t}(.,-\boldsymbol{e}_2)\rangle_{A_2} + n_3\langle \boldsymbol{t}(.,-\boldsymbol{e}_3)\rangle_{A_3} + \langle \boldsymbol{t}(.,\boldsymbol{n})\rangle_{A} + \frac{1}{3}h\langle \boldsymbol{f}_B\rangle_V = \boldsymbol{0}, \quad (2.13)$$

and by taking the limit $h \to 0$ we obtain

$$\boldsymbol{t}(\boldsymbol{x},\boldsymbol{n}) = n_1\boldsymbol{t}(\boldsymbol{x},\boldsymbol{e}_1) + n_2\boldsymbol{t}(\boldsymbol{x},\boldsymbol{e}_2) + n_3\boldsymbol{t}(\boldsymbol{x},\boldsymbol{e}_3). \quad (2.14)$$

This relationship can be written as

$$\boldsymbol{t}(\boldsymbol{x},\boldsymbol{n}) = \boldsymbol{S}(\boldsymbol{x})^{\mathrm{T}}\boldsymbol{n}, \quad (2.15)$$

using the matrix

$$\boldsymbol{S}(\boldsymbol{x}) = \Big(\boldsymbol{t}(\boldsymbol{x},\boldsymbol{e}_1), \boldsymbol{t}(\boldsymbol{x},\boldsymbol{e}_2), \boldsymbol{t}(\boldsymbol{x},\boldsymbol{e}_3)\Big)^{\mathrm{T}}. \quad (2.16)$$

$\boldsymbol{S}$ is a second order tensor, which is referred to as *first Piola-Kirchhoff stress tensor*. In addition, the definition

$$\widetilde{\boldsymbol{S}} = \boldsymbol{S}\boldsymbol{F}^{-\mathrm{T}} \quad (2.17)$$

yields a second order tensor $\widetilde{\boldsymbol{S}}$, which is referred to as *second Piola-Kirchhoff stress tensor*.

2.1.5 Material Models

The relationship between strain and stress is described by a *material model*. We consider two material models: Elastic and hyperelastic materials.

Elastic materials are idealized materials for which the state of stress at each material point depends solely on the current deformation at this point. In particular, the stress is independent of the deformation rate or deformation history. For elastic materials, the stress thus is a function of the strain, i.e., $\widetilde{\boldsymbol{S}} = \widetilde{\boldsymbol{S}}(\boldsymbol{E})$.

In particular, *hyperelastic* materials are idealized materials for which the strain-stress relationship can be derived from a *strain energy density* (strain energy per unit volume) function $W = W(\boldsymbol{E})$ via

$$\widetilde{\boldsymbol{S}} = \widetilde{\boldsymbol{S}}(\boldsymbol{E}) = \frac{\partial W(\boldsymbol{E})}{\partial \boldsymbol{E}}. \tag{2.18}$$

This model thus describes idealized materials that store the work performed by the external forces as *strain energy* (potential energy) under deformation, and release this energy when returning to the undeformed state. The hyperelasticity model thus prohibits closed deformation cycles that release unlimited energy. An elastic material is hyperelastic, iff

$$\frac{\partial \widetilde{S}_{ij}}{\partial E_{kl}} = \frac{\partial \widetilde{S}_{kl}}{\partial E_{ij}}. \tag{2.19}$$

Physical reasons require $W(\boldsymbol{E})$ to be positive definite, i.e., $W(\boldsymbol{E}) \geq 0$ and $W(\boldsymbol{E}) = 0 \Leftrightarrow \boldsymbol{E} = \boldsymbol{0}$. The *strain energy density* then can be obtained via[1]

$$W(\boldsymbol{E}) = \int_{\boldsymbol{0}}^{\boldsymbol{E}} \widetilde{\boldsymbol{S}}(\boldsymbol{E}') : \boldsymbol{dE}'. \tag{2.20}$$

In the following, we tacitly assume that the considered materials are hyperelastic with positive definite strain energy density functions.

[1] The contraction $\boldsymbol{A} : \boldsymbol{B}$ of two second order tensors $\boldsymbol{A}$ and $\boldsymbol{B}$ is defined as $\boldsymbol{A} : \boldsymbol{B} = \sum_{i,j=1}^{3} A_{ij} B_{ji}$.

2.1.6 Equations of Equilibrium

For any subvolume $V \subset \Omega$ of the deformable body, the sum of the forces acting on this subvolume must vanish (*balance of forces*), i.e.,

$$\int_{\partial V} \boldsymbol{t}(.,\boldsymbol{n})\, dA + \int_{V} \boldsymbol{f}_B\, dx = \boldsymbol{0}. \tag{2.21}$$

Using $\boldsymbol{t}(.,\boldsymbol{n}) = \boldsymbol{S}^{\mathrm{T}}\boldsymbol{n}$, applying the divergence theorem, and considering that the equation is satisfied for any subvolume $V \subset \Omega$ leads to[2]

$$\operatorname{div} \boldsymbol{S}^{\mathrm{T}} + \boldsymbol{f}_B = \boldsymbol{0} \qquad \text{in} \quad \Omega. \tag{2.22}$$

In addition, for any subvolume $V \subset \Omega$ of the deformable body, the sum of the moments acting on this subvolume must vanish (*balance of moments*), i.e.,

$$\int_{\partial V} \boldsymbol{\varphi} \times \boldsymbol{t}(.,\boldsymbol{n})\, dA + \int_{V} \boldsymbol{\varphi} \times \boldsymbol{f}_B\, dx = \boldsymbol{0} \tag{2.23}$$

(the moments are computed around the origin). Using $\boldsymbol{t}(.,\boldsymbol{n}) = \boldsymbol{S}^{\mathrm{T}}\boldsymbol{n}$, $\operatorname{div} \boldsymbol{S}^{\mathrm{T}} + \boldsymbol{f}_B = \boldsymbol{0}$, applying the divergence theorem, and considering that the equation is satisfied for any subvolume $V \subset \Omega$ leads to

$$\boldsymbol{F}\boldsymbol{S} = \boldsymbol{S}^{\mathrm{T}}\boldsymbol{F}^{\mathrm{T}} \qquad \text{in} \quad \Omega, \tag{2.24}$$

which is equivalent to

$$\widetilde{\boldsymbol{S}} = \widetilde{\boldsymbol{S}}^{\mathrm{T}} \qquad \text{in} \quad \Omega, \tag{2.25}$$

i.e., the second Piola-Kirchhoff stress tensor is required to be symmetric.

The static elasticity problem can then be formulated as a boundary value problem: Find $\boldsymbol{u} : \overline{\Omega} \to \mathbb{R}^3$ such that

$$\begin{aligned} -\operatorname{div}\big(\boldsymbol{F}(\boldsymbol{u})\widetilde{\boldsymbol{S}}(\boldsymbol{E}(\boldsymbol{u}))\big) &= \boldsymbol{f}_B && \text{in} \quad \Omega, && (2.26\text{a})\\ \boldsymbol{u} &= \boldsymbol{u}^0 && \text{on} \quad \Gamma_D, && (2.26\text{b})\\ \boldsymbol{F}(\boldsymbol{u})\widetilde{\boldsymbol{S}}(\boldsymbol{E}(\boldsymbol{u}))\,\boldsymbol{n} &= \boldsymbol{f}_S && \text{on} \quad \Gamma_N. && (2.26\text{c}) \end{aligned}$$

Equation (2.26a) is a system of elliptic partial differential equations. The boundary

[2]The divergence $\operatorname{div} \boldsymbol{A}$ of a second order tensor $\boldsymbol{A}$ is defined as $\operatorname{div} \boldsymbol{A} = \left(\sum_{j=1}^{3} \frac{\partial A_{1j}}{\partial x_j}, \ldots, \sum_{j=1}^{3} \frac{\partial A_{3j}}{\partial x_j} \right)^{\mathrm{T}}$.

conditions on the boundary $\partial\Omega = \Gamma_D \,\dot{\cup}\, \Gamma_N$ consist of Dirichlet boundary conditions (2.26b), which prescribe the displacement $\boldsymbol{u}^0$ on Γ_D, and Neumann boundary conditions (2.26c), which prescribe external surface forces $\boldsymbol{f}_S$ on Γ_N. $\boldsymbol{n}$ is the unit outer normal on $\partial\Omega$. Note that $\boldsymbol{F}(\boldsymbol{u}) = \nabla\boldsymbol{u} + \mathbf{1}_3$, $\boldsymbol{E}(\boldsymbol{u}) = \frac{1}{2}(\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^{\mathrm{T}} + (\nabla\boldsymbol{u})^{\mathrm{T}}\nabla\boldsymbol{u})$, and that $\widetilde{\boldsymbol{S}}$ and $\boldsymbol{E}$ are related by the material model $\widetilde{\boldsymbol{S}} = \widetilde{\boldsymbol{S}}(\boldsymbol{E})$, which must ensure the symmetry of $\widetilde{\boldsymbol{S}}$ (Equation (2.25)).

For the simulation of the motion of a deformable body over a time interval $I = [0, t_{\text{end}}]$, inertial forces have to be additionally considered. This is achieved by replacing $\boldsymbol{f}_B$ with $\boldsymbol{f}_B - \rho\ddot{\boldsymbol{u}}$, where ρ is the material's density, and $\ddot{\boldsymbol{u}}$ denotes the second derivative of $\boldsymbol{u}$ with respect to time (i.e., the acceleration).

The dynamic elasticity problem can then be formulated as an initial boundary value problem: Find $\boldsymbol{u} : \overline{\Omega} \times I \to \mathbb{R}^3$ such that

$$\begin{aligned}
\rho\ddot{\boldsymbol{u}} - \operatorname{div}\big(\boldsymbol{F}(\boldsymbol{u})\widetilde{\boldsymbol{S}}(\boldsymbol{E}(\boldsymbol{u}))\big) &= \boldsymbol{f}_B && \text{in} \quad \Omega \times I, && \text{(2.27a)}\\
\boldsymbol{u} &= \boldsymbol{u}^0 && \text{on} \quad \Gamma_D \times I, && \text{(2.27b)}\\
\boldsymbol{F}(\boldsymbol{u})\widetilde{\boldsymbol{S}}(\boldsymbol{E}(\boldsymbol{u}))\,\boldsymbol{n} &= \boldsymbol{f}_S && \text{on} \quad \Gamma_N \times I, && \text{(2.27c)}\\
\boldsymbol{u} &= \boldsymbol{u}^0 && \text{in} \quad \Omega \times \{0\}, && \text{(2.27d)}\\
\dot{\boldsymbol{u}} &= \dot{\boldsymbol{u}}^0 && \text{in} \quad \Omega \times \{0\}. && \text{(2.27e)}
\end{aligned}$$

Here, Equation (2.27a) is a system of hyperbolic partial differential equations. The additional initial conditions (2.27d) and (2.27e) prescribe the displacement $\boldsymbol{u}^0$ and the velocity $\dot{\boldsymbol{u}}^0$ of the deformable body at time $t = 0$.

2.1.7 Linear Elasticity Theory

In this thesis, we employ the *linear* theory of elasticity, which deals with deformations that exhibit small strains and small rotations. More precisely, in the linear theory it is assumed that the *displacement gradient* $\nabla\boldsymbol{u}$ is small, i.e., $\|\nabla\boldsymbol{u}\| \ll 1$. This assumption allows for certain linearizations. In particular, the linear theory is based on a linear strain tensor (geometrical linearity) and a linear material model (physical linearity). The advantage of the linear theory is that the discretization of the (initial) boundary value problem leads to a linear system of equations, which can be solved very efficiently. By using a corotational formulation of strain, the linear theory can also be applied accurately to deformations that exhibit large rotations (but small strains).

Geometrical Linearization

In the linear elasticity theory, the linear *infinitesimal strain tensor*

$$\boldsymbol{\varepsilon} = \frac{1}{2}\left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^{\mathrm{T}}\right) \tag{2.28}$$

is employed, which is derived from the non-linear Green-St. Venant strain tensor $\boldsymbol{E}$ by neglecting second order terms. If the displacement gradient is small, i.e., $\|\nabla \boldsymbol{u}\| \ll 1$, it is $\boldsymbol{E} \approx \boldsymbol{\varepsilon}$. In contrast to the Green-St. Venant strain tensor, the infinitesimal strain tensor is not invariant under rotations, i.e., rotations are interpreted by the infinitesimal strain tensor as strains, which introduces artificial stresses.

Furthermore, if $\|\nabla \boldsymbol{u}\| \ll 1$, it is $\boldsymbol{F} \approx \boldsymbol{1}_3$, $\boldsymbol{S} \approx \boldsymbol{S}^{\mathrm{T}}$, and $\boldsymbol{S} \approx \widetilde{\boldsymbol{S}}$. Therefore, the first and the second Piola-Kirchhoff stress tensor can be approximated by a single, symmetric stress tensor $\boldsymbol{\sigma}$.

Using these approximations, the constitutive equations for elastic and hyperelastic materials become $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\boldsymbol{\varepsilon})$ and $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\boldsymbol{\varepsilon}) = \frac{\partial W(\boldsymbol{\varepsilon})}{\partial \boldsymbol{\varepsilon}}$, respectively.

The static elasticity problem (2.26) becomes: Find $\boldsymbol{u} : \overline{\Omega} \to \mathbb{R}^3$ such that

$$\begin{aligned} -\operatorname{div} \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u})) &= \boldsymbol{f}_B && \text{in} \quad \Omega, && (2.29\text{a})\\ \boldsymbol{u} &= \boldsymbol{u}^0 && \text{on} \quad \Gamma_D, && (2.29\text{b})\\ \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))\, \boldsymbol{n} &= \boldsymbol{f}_S && \text{on} \quad \Gamma_N. && (2.29\text{c}) \end{aligned}$$

The dynamic elasticity problem (2.27) becomes: Find $\boldsymbol{u} : \overline{\Omega} \times I \to \mathbb{R}^3$ such that

$$\begin{aligned} \rho \ddot{\boldsymbol{u}} - \operatorname{div} \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u})) &= \boldsymbol{f}_B && \text{in} \quad \Omega \times I, && (2.30\text{a})\\ \boldsymbol{u} &= \boldsymbol{u}^0 && \text{on} \quad \Gamma_D \times I, && (2.30\text{b})\\ \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))\, \boldsymbol{n} &= \boldsymbol{f}_S && \text{on} \quad \Gamma_N \times I, && (2.30\text{c})\\ \boldsymbol{u} &= \boldsymbol{u}^0 && \text{in} \quad \Omega \times \{0\}, && (2.30\text{d})\\ \dot{\boldsymbol{u}} &= \dot{\boldsymbol{u}}^0 && \text{in} \quad \Omega \times \{0\}. && (2.30\text{e}) \end{aligned}$$

Physical Linearization

The linear elasticity theory is limited to *linear elastic* materials, which are characterized by a linear relationship between stress and strain, i.e.,

$$\boldsymbol{\sigma} = \boldsymbol{C} : \boldsymbol{\varepsilon}. \tag{2.31}$$

This relationship is referred to as *generalized Hooke's law*. $\boldsymbol{C}$ is a fourth order tensor, which is referred to as *elasticity tensor*. A linear relationship between stress and strain is a reasonable approximation for many real-world materials in case of small strains.

A fourth order tensor has 81 scalar components. However, the symmetry of $\boldsymbol{\sigma}$ requires $C_{ijk\ell} = C_{jik\ell}$, and due to the symmetry of ε the skew symmetric part of $C = \left(\frac{1}{2}\left(C_{ijk\ell} + C_{ij\ell k}\right) + \frac{1}{2}\left(C_{ijk\ell} - C_{ij\ell k}\right)\right)$ with respect to the third and fourth index can be assumed to be zero without loss of generality (since $\sum_{k,\ell=1}^{3} \frac{1}{2}\left(C_{ijk\ell} - C_{ij\ell k}\right)\varepsilon_{k\ell} = 0$), leading to $C_{ijk\ell} = C_{ij\ell k}$. These two symmetries reduce the number of independent scalar components of the elasticity tensor to 36.

By defining

$$\overline{\boldsymbol{\sigma}} = \left(\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{13}, \sigma_{23}\right)^{\mathrm{T}}, \tag{2.32}$$

$$\overline{\boldsymbol{\varepsilon}} = \left(\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, 2\varepsilon_{12}, 2\varepsilon_{13}, 2\varepsilon_{23}\right)^{\mathrm{T}}, \tag{2.33}$$

$$\text{and} \qquad \overline{\boldsymbol{C}} = \begin{pmatrix} C_{1111} & C_{1122} & C_{1133} & C_{1112} & C_{1113} & C_{1123} \\ C_{2211} & C_{2222} & C_{2233} & C_{2212} & C_{2213} & C_{2223} \\ C_{3311} & C_{3322} & C_{3333} & C_{3312} & C_{3313} & C_{3323} \\ C_{1211} & C_{1222} & C_{1233} & C_{1212} & C_{1213} & C_{1223} \\ C_{1311} & C_{1322} & C_{1333} & C_{1312} & C_{1313} & C_{1323} \\ C_{2311} & C_{2322} & C_{2333} & C_{2312} & C_{2313} & C_{2323} \end{pmatrix}, \tag{2.34}$$

it is

$$\boldsymbol{\sigma} = \boldsymbol{C} : \boldsymbol{\varepsilon} \qquad \Leftrightarrow \qquad \overline{\boldsymbol{\sigma}} = \overline{\boldsymbol{C}}\,\overline{\boldsymbol{\varepsilon}}, \tag{2.35}$$

and

$$\boldsymbol{\sigma} : \boldsymbol{\varepsilon} = \overline{\boldsymbol{\sigma}}^{\mathrm{T}}\overline{\boldsymbol{\varepsilon}}, \tag{2.36}$$

i.e., the contraction can be written as a matrix-vector product or a scalar product, respectively (so-called *engineering notation*).

According to Equation (2.19), a linear elastic material is hyperelastic, iff

$$C_{ijk\ell} = \frac{\partial \sigma_{ij}}{\partial \varepsilon_{k\ell}} = \frac{\partial \sigma_{k\ell}}{\partial \varepsilon_{ij}} = C_{k\ell ij}. \tag{2.37}$$

This further reduces the number of independent scalar components of the elasticity tensor to 21. In particular, a linear elastic material is hyperelastic, iff $\overline{\boldsymbol{C}}$ is symmetric. According to Equation (2.20), for linear hyperelastic materials the strain energy density is

$$W(\boldsymbol{\varepsilon}) = \frac{1}{2}\boldsymbol{\varepsilon} : \boldsymbol{C} : \boldsymbol{\varepsilon}. \tag{2.38}$$

The deformation behavior of an *isotropic* material is independent of the material's spatial orientation. It can be shown that this leaves only two independent scalar components of the elasticity tensor, and that the strain-stress relationship can be written in the form

$$\boldsymbol{\sigma} = 2\mu\,\boldsymbol{\varepsilon} + \lambda\,\mathrm{tr}(\boldsymbol{\varepsilon})\,I, \tag{2.39}$$

with two scalar material parameters λ and μ, which are referred to as the *Lamé constants*. In engineering notation, Equation (2.39) is written as

$$\begin{pmatrix}\sigma_{11}\\ \sigma_{22}\\ \sigma_{33}\\ \sigma_{12}\\ \sigma_{13}\\ \sigma_{23}\end{pmatrix} = \begin{pmatrix}2\mu+\lambda & \lambda & \lambda & & & \\ \lambda & 2\mu+\lambda & \lambda & & & \\ \lambda & \lambda & 2\mu+\lambda & & & \\ & & & \mu & & \\ & & & & \mu & \\ & & & & & \mu\end{pmatrix}\begin{pmatrix}\varepsilon_{11}\\ \varepsilon_{22}\\ \varepsilon_{33}\\ 2\varepsilon_{12}\\ 2\varepsilon_{13}\\ 2\varepsilon_{23}\end{pmatrix}. \tag{2.40}$$

An isotropic material is always hyperelastic.

By considering the special case of *uniaxial stress*, i.e., $\sigma_{11} = \sigma$ and $\sigma_{22} = \sigma_{33} = \sigma_{12} = \sigma_{13} = \sigma_{23} = 0$, two more intuitive material parameters can be defined. Due to Equation (2.40) it is $\varepsilon_{12} = \varepsilon_{13} = \varepsilon_{23} = 0$ and $\varepsilon_{22} = \varepsilon_{33}$. The *Young's modulus* E is defined as the ratio of the longitudinal stress and the longitudinal strain, and the *Poisson's ratio* ν is defined as the negative ratio of the transversal and the longitudinal strain, i.e.,

$$E = \frac{\sigma}{\varepsilon_{11}}, \tag{2.41}$$

$$\nu = -\frac{\varepsilon_{22}}{\varepsilon_{11}} = -\frac{\varepsilon_{33}}{\varepsilon_{11}}. \tag{2.42}$$

The Poisson's ratio describes the fact that most materials contract transversally when they are stretched longitudinally. A perfectly incompressible material exhibits a Poisson's ratio of 0.5. For most materials, the Poisson's ratio is between 0 and 0.5.

The Young's modulus and the Poisson's ratio are related to the Lamé constants by

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}. \tag{2.43}$$

2.2 The Finite Element Method

In this thesis, we employ the finite element method for the spatial discretization of the (initial) boundary value problem (2.29) or (2.30). The solutions of the (initial) boundary value problem reside in an infinite dimensional function space. The basic idea underlying the finite element method is to replace this infinite dimensional function space by a finite dimensional space that is constructed by decomposing the simulation domain into a set of finite elements and by defining for each element a set of shape functions spanning a local function space. An approximate solution is then obtained by formulating the (initial) boundary value problem as a variational problem, and by considering this variational problem over the finite dimensional function space (*Galerkin approach*). For the static case, this directly leads to a linear system of equations. For the dynamic case, this leads to an initial value problem, consisting of a system of ordinary differential equations with initial conditions. In this thesis, the initial value problem is discretized in time by applying the implicit Newmark time integration scheme, which finally leads to a linear system of equations for each time step. In the following, we consider the spatial discretization for the dynamic case. The static case can be directly derived from the dynamic case by neglecting inertia and damping.

2.2.1 Weak Formulation of the Initial Boundary Value Problem

The initial boundary value problem (2.30) is defined over the function space $\mathcal{V} = (C^2(\Omega))^3$, i.e., a solution $\boldsymbol{u}$ must be a twice continuously differentiable function $\Omega \to \mathbb{R}^3$ for each point in time. Let $\mathcal{V}_0 = \{\boldsymbol{v} \in \mathcal{V} \mid \boldsymbol{v} = \boldsymbol{0} \text{ on } \Gamma_D\}$ denote the subspace of $\mathcal{V}$ containing all functions that vanish on Γ_D.

By multiplying Equation (2.30a) with an arbitrary test function $\boldsymbol{v} \in \mathcal{V}_0$ and integrating over the domain Ω, an equivalent formulation of the initial boundary value problem (2.30) is obtained (we omit the initial conditions (2.30d) and (2.30e) in the following):

$$\int_\Omega \boldsymbol{v} \cdot (\rho \ddot{\boldsymbol{u}} - \operatorname{div} \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u})) - \boldsymbol{f}_B)\, dx = 0 \qquad \forall \boldsymbol{v} \in \mathcal{V}_0, \tag{2.44a}$$

$$\boldsymbol{u} = \boldsymbol{u}^0 \qquad \text{on} \quad \Gamma_D, \tag{2.44b}$$

$$\boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))\, \boldsymbol{n} = \boldsymbol{f}_S \qquad \text{on} \quad \Gamma_N. \tag{2.44c}$$

Applying the divergence theorem (integration by parts)

$$\int\limits_{\Omega} \boldsymbol{v} \cdot \operatorname{div} \boldsymbol{\sigma} \, dx = \int\limits_{\partial\Omega} \boldsymbol{v} \cdot (\boldsymbol{\sigma}\, \boldsymbol{n}) \, dA - \int\limits_{\Omega} \nabla \boldsymbol{v} : \boldsymbol{\sigma} \, dx, \tag{2.45}$$

and using

$$\begin{aligned} \boldsymbol{v} &= \boldsymbol{0} \qquad \text{on} \quad \Gamma_D, \\ \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u}))\, \boldsymbol{n} &= \boldsymbol{f}_S \qquad \text{on} \quad \Gamma_N, \\ \nabla \boldsymbol{v} : \boldsymbol{\sigma} &= \frac{1}{2} \left(\nabla \boldsymbol{v} + (\nabla \boldsymbol{v})^{\mathrm{T}} \right) : \boldsymbol{\sigma} = \boldsymbol{\varepsilon}(\boldsymbol{v}) : \boldsymbol{\sigma}, \end{aligned}$$

leads to the variational problem

$$\int\limits_{\Omega} \rho \boldsymbol{v} \cdot \ddot{\boldsymbol{u}} \, dx + \int\limits_{\Omega} \boldsymbol{\varepsilon}(\boldsymbol{v}) : \boldsymbol{\sigma}(\boldsymbol{\varepsilon}(\boldsymbol{u})) \, dx - \int\limits_{\Omega} \boldsymbol{v} \cdot \boldsymbol{f}_B \, dx - \int\limits_{\Gamma_N} \boldsymbol{v} \cdot \boldsymbol{f}_S \, dA = 0 \qquad \forall \boldsymbol{v} \in \mathcal{V}_0, \tag{2.46a}$$

$$\boldsymbol{u} = \boldsymbol{u}^0 \quad \text{on} \quad \Gamma_D. \tag{2.46b}$$

Note that the Neumann boundary conditions can be recovered from Equation (2.46a) and thus are not explicitly included (*natural boundary conditions*), in contrast to the Dirichlet boundary conditions, which have to be explicitly enforced (*essential boundary conditions*). In structural mechanics, the variational formulation (2.46) is referred to as *principle of virtual displacements* or *principle of virtual work*.

Since the variational problem (2.46) does not contain second spatial derivatives, it can be considered over a function space $\mathcal{V}$ that exhibits weaker differentiability requirements than $(C^2(\Omega))^3$. The finite element method is based on using $\mathcal{V} = (H^1(\Omega))^3$, where $H^1(\Omega)$ denotes the *Sobolev space* of square-integrable and weakly differentiable functions $\Omega \to \mathbb{R}$. In particular, all continuous, piecewise infinitely differentiable functions $\Omega \to \mathbb{R}$ are in $H^1(\Omega)$.

The variational problem is referred to as *weak formulation* of the initial boundary value problem, and the respective solutions are called *weak solutions*—weak in the sense that these solutions do in general not satisfy the differentiability requirements imposed by the initial boundary value problem (the *strong formulation*).

2.2.2 Finite Element Discretization

The basic idea of the finite element method is to compute an approximate solution of the variational problem by considering this problem over a finite dimensional function space $\mathcal{V}_h \subset \mathcal{V}$. This finite dimensional function space is constructed piecewise by dividing the domain Ω into a finite number of subdomains, called *finite elements*. Typically, triangles or quadrilaterals are used in 2D, and tetrahedra or hexahedra are used in 3D. For each element, a local finite dimensional function space is spanned by a set of *shape functions*. Typically, this local function space is constructed by specifying a set of discrete locations, which are referred to as the element's *nodes*, as well as a certain interpolation scheme that interpolates the values (here: displacements) given at the nodes within the element. The local function space then consists of the interpolation functions. For simple elements, the nodes coincide with the vertices of the element. Since nodes are shared between adjacent elements, the piecewise constructed functions are continuous over the entire domain Ω.

In this thesis, we use a decomposition of the deformable object based on a uniform Cartesian grid or an adaptive octree grid. We use axis-aligned hexahedral finite elements with trilinear shape functions, i.e., each element has eight nodes that coincide with the element's vertices, and the values at the nodes are trilinearly interpolated within the element. Using a (semi-)regular hexahedral grid has several advantages: First, the grid can be generated from a surface mesh in a robust way by using a voxelization algorithm, including a multigrid hierarchy, which is essential for a geometric multigrid solver. Second, the regularity of the grid leads to a regular shape of the numerical stencil, enabling an efficient parallelization of computations and memory accesses on GPUs. Third, since all elements have the same shape, the element matrices can be directly derived from the element matrices of a single, generic element, which significantly reduces computational work for element integration as well as memory requirements.

In the following, we assume that the vertices of the finite element grid are enumerated from 1 to N_v, where N_v denotes the number of vertices of the finite element grid. The sharing of vertices between adjacent elements is modeled by a mapping s that specifies the global index $s(e,i) \in \{1,\dots,N_v\}$ of each element vertex, identified by the element e and the element-local vertex index $i \in \{1,\dots,8\}$.

The interpolation function is uniquely determined by the values at the vertices of the finite element grid. For a variable identifier $*$, we use the notations $*_h$, $*_k$, $\underline{*} = \left(*_1^{\mathrm{T}},\dots,*_{N_v}^{\mathrm{T}}\right)^{\mathrm{T}}$, and $\underline{*}^e = \left(*_{s(e,1)}^{\mathrm{T}},\dots,*_{s(e,8)}^{\mathrm{T}}\right)^{\mathrm{T}}$ to mutually refer to the interpolation function, the displacement at vertex k, the linearization of the displacements at all vertices, and the linearization of the displacements at the vertices of element e.

The trilinear interpolation scheme is defined on each hexahedral finite element e with domain $\Omega^e = [x_1^e, x_1^e + a_1^e] \times [x_2^e, x_2^e + a_2^e] \times [x_3^e, x_3^e + a_3^e]$ by

$$\boldsymbol{v}_h\big|_{\Omega^e}(\boldsymbol{x}) = \sum_{i=1}^{8} \boldsymbol{v}_{s(e,i)} \phi_i^e(\boldsymbol{x}), \tag{2.47}$$

using the *element shape functions* ϕ_i^e, which are defined by

$$\phi_i^e(\boldsymbol{x}) = N_i\left(\boldsymbol{J}^e(\boldsymbol{x})\right), \tag{2.48}$$

$$\boldsymbol{J}^e(\boldsymbol{x}) = \left(\frac{x_1 - x_1^e}{a_1^e}, \frac{x_2 - x_2^e}{a_2^e}, \frac{x_3 - x_3^e}{a_3^e} \right)^T, \tag{2.49}$$

and

$$N_1(\boldsymbol{x}) = (1 - x_1)(1 - x_2)(1 - x_3), \tag{2.50a}$$
$$N_2(\boldsymbol{x}) = x_1(1 - x_2)(1 - x_3), \tag{2.50b}$$
$$N_3(\boldsymbol{x}) = (1 - x_1)x_2(1 - x_3), \tag{2.50c}$$
$$N_4(\boldsymbol{x}) = x_1 x_2 (1 - x_3), \tag{2.50d}$$
$$N_5(\boldsymbol{x}) = (1 - x_1)(1 - x_2)x_3, \tag{2.50e}$$
$$N_6(\boldsymbol{x}) = x_1(1 - x_2)x_3, \tag{2.50f}$$
$$N_7(\boldsymbol{x}) = (1 - x_1)x_2 x_3, \tag{2.50g}$$
$$N_8(\boldsymbol{x}) = x_1 x_2 x_3. \tag{2.50h}$$

In the following, we first consider a finite element discretization based on a uniform hexahedral grid. The handling of hanging vertices in an adaptive octree grid will be explained in Section 2.2.4.

Based on the described finite element discretization, the variational problem (2.46) is considered over the finite dimensional function space

$$\mathcal{V}_h = \left\{ \boldsymbol{v}_h : \overline{\Omega} \to \mathbb{R}^3, \boldsymbol{v}_h\big|_{\Omega^e}(\boldsymbol{x}) = \sum_{i=1}^{8} \boldsymbol{v}_{s(e,i)} \phi_i^e(\boldsymbol{x}) \;\middle|\; \underline{\boldsymbol{v}} \in (\mathbb{R}^3)^{N_v} \right\}. \tag{2.51}$$

Let $\boldsymbol{u}_h$ denote the spatially discretized displacement field, and let $\boldsymbol{u}_k$, $\underline{\boldsymbol{u}}$, and $\underline{\boldsymbol{u}}^e$ be defined as described above. We incorporate Dirichlet boundary conditions by prescribing displacements $\boldsymbol{u}_k = \boldsymbol{u}_k^0$ at a subset of the vertices. Without loss of generality, let $\{1, \ldots, N_f\}$ and $\{N_f + 1, \ldots, N_v\}$ be the indices of the 'free' and the 'fixed' vertices, respectively. Then,

$$\mathcal{V}_{h,0} = \left\{ \boldsymbol{v}_h \,\middle|\, \underline{\boldsymbol{v}} \in (\mathbb{R}^3)^{N_f} \times \{\boldsymbol{0}\}^{N_v - N_f} \right\}. \tag{2.52}$$

For the sake of simplicity, external surface forces are not considered in the following ($\boldsymbol{f}_S = \boldsymbol{0}$). By applying the linear material law $\boldsymbol{\sigma} = \boldsymbol{C} : \boldsymbol{\varepsilon}$ we obtain

$$\int\limits_{\Omega} \rho \boldsymbol{v}_h \cdot \ddot{\boldsymbol{u}}_h \, dx + \int\limits_{\Omega} \boldsymbol{\varepsilon}(\boldsymbol{v}_h) : \boldsymbol{C} : \boldsymbol{\varepsilon}(\boldsymbol{u}_h) \, dx - \int\limits_{\Omega} \boldsymbol{v}_h \cdot \boldsymbol{f}_B \, dx = 0 \qquad \forall \boldsymbol{v}_h \in \mathcal{V}_{h,0}, \tag{2.53a}$$

$$\boldsymbol{u}_k = \boldsymbol{u}_k^0 \qquad \text{for} \quad k = N_f + 1, \dots, N_v. \tag{2.53b}$$

For each element e, we define the *element shape matrix* $\boldsymbol{\Phi}^e(\boldsymbol{x})$ and the *element strain matrix* $\boldsymbol{B}^e(\boldsymbol{x})$ as

$$\boldsymbol{\Phi}^e(\boldsymbol{x}) = \left(\begin{array}{ccc|c|ccc} \phi_1^e(\boldsymbol{x}) & & & & \phi_8^e(\boldsymbol{x}) & & \\ & \phi_1^e(\boldsymbol{x}) & & \cdots & & \phi_8^e(\boldsymbol{x}) & \\ & & \phi_1^e(\boldsymbol{x}) & & & & \phi_8^e(\boldsymbol{x}) \end{array} \right), \tag{2.54}$$

$$\boldsymbol{B}^e(\boldsymbol{x}) = \left(\begin{array}{ccc|c|ccc} \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_1} & & & & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_1} & & \\ & \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_2} & & & & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_2} & \\ & & \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_3} & \cdots & & & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_3} \\ \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_2} & \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_1} & & & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_2} & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_1} & \\ \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_3} & & \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_1} & & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_3} & & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_1} \\ & \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_3} & \frac{\partial \phi_1^e(\boldsymbol{x})}{\partial x_2} & & & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_3} & \frac{\partial \phi_8^e(\boldsymbol{x})}{\partial x_2} \end{array} \right). \tag{2.55}$$

Then,

$$\boldsymbol{u}_h\big|_{\Omega^e}(\boldsymbol{x}) = \boldsymbol{\Phi}^e(\boldsymbol{x})\underline{\boldsymbol{u}}^e, \tag{2.56}$$

$$\ddot{\boldsymbol{u}}_h\big|_{\Omega^e}(\boldsymbol{x}) = \boldsymbol{\Phi}^e(\boldsymbol{x})\underline{\ddot{\boldsymbol{u}}}^e, \tag{2.57}$$

and

$$\overline{\boldsymbol{\varepsilon}}(\boldsymbol{u}_h)\big|_{\Omega^e}(\boldsymbol{x}) = \boldsymbol{B}^e(\boldsymbol{x})\underline{\boldsymbol{u}}^e. \tag{2.58}$$

Using Equations (2.56) to (2.58), we obtain

$$\int_{\Omega^e} \rho \boldsymbol{v}_h \cdot \ddot{\boldsymbol{u}}_h \, dx = \int_{\Omega^e} \rho (\underline{\boldsymbol{v}}^e)^{\mathrm{T}} (\boldsymbol{\Phi}^e)^{\mathrm{T}} \boldsymbol{\Phi}^e \underline{\ddot{\boldsymbol{u}}}^e \, dx = (\underline{\boldsymbol{v}}^e)^{\mathrm{T}} \underbrace{\int_{\Omega^e} \rho (\boldsymbol{\Phi}^e)^{\mathrm{T}} \boldsymbol{\Phi}^e \, dx \, \underline{\ddot{\boldsymbol{u}}}^e}_{=\boldsymbol{M}^e}, \tag{2.59}$$

$$\int_{\Omega^e} \boldsymbol{\varepsilon}(\boldsymbol{v}_h) : \boldsymbol{C} : \boldsymbol{\varepsilon}(\boldsymbol{u}_h) \, dx = \int_{\Omega^e} (\underline{\boldsymbol{v}}^e)^{\mathrm{T}} (\boldsymbol{B}^e)^{\mathrm{T}} \overline{\boldsymbol{C}} \boldsymbol{B}^e \underline{\boldsymbol{u}}^e \, dx = (\underline{\boldsymbol{v}}^e)^{\mathrm{T}} \underbrace{\int_{\Omega^e} (\boldsymbol{B}^e)^{\mathrm{T}} \overline{\boldsymbol{C}} \boldsymbol{B}^e \, dx \, \underline{\boldsymbol{u}}^e}_{=\boldsymbol{K}^e}, \tag{2.60}$$

$$\int_{\Omega^e} \boldsymbol{v}_h \cdot \boldsymbol{f}_B \, dx = \int_{\Omega^e} (\underline{\boldsymbol{v}}^e)^{\mathrm{T}} (\boldsymbol{\Phi}^e)^{\mathrm{T}} \boldsymbol{f}_B \, dx = (\underline{\boldsymbol{v}}^e)^{\mathrm{T}} \underbrace{\int_{\Omega^e} (\boldsymbol{\Phi}^e)^{\mathrm{T}} \boldsymbol{f}_B \, dx}_{=\underline{\boldsymbol{f}}^e} . \tag{2.61}$$

For each element e, we define the *element mass matrix* $\boldsymbol{M}^e$, the *element stiffness matrix* $\boldsymbol{K}^e$, and the *element load vector* $\underline{\boldsymbol{f}}^e$ as

$$\boldsymbol{M}^e = \int_{\Omega^e} \rho (\boldsymbol{\Phi}^e)^{\mathrm{T}} \boldsymbol{\Phi}^e \, dx, \tag{2.62}$$

$$\boldsymbol{K}^e = \int_{\Omega^e} (\boldsymbol{B}^e)^{\mathrm{T}} \overline{\boldsymbol{C}} \boldsymbol{B}^e \, dx, \tag{2.63}$$

$$\underline{\boldsymbol{f}}^e = \int_{\Omega^e} (\boldsymbol{\Phi}^e)^{\mathrm{T}} \boldsymbol{f}_B \, dx. \tag{2.64}$$

Considering the decomposition of the domain Ω into finite elements and substituting Equations (2.59) to (2.61) into (2.53a) leads to

$$\sum_e (\underline{\boldsymbol{v}}^e)^{\mathrm{T}} \Big(\boldsymbol{M}^e \underline{\ddot{\boldsymbol{u}}}^e + \boldsymbol{K}^e \underline{\boldsymbol{u}}^e - \underline{\boldsymbol{f}}^e \Big) = 0 \qquad \forall \underline{\boldsymbol{v}} \in (\mathbb{R}^3)^{N_f} \times \{\boldsymbol{0}\}^{N_v - N_f}. \tag{2.65}$$

This can be written as

$$\underline{\boldsymbol{v}}^{\mathrm{T}} \Big(\boldsymbol{M} \underline{\ddot{\boldsymbol{u}}} + \boldsymbol{K} \underline{\boldsymbol{u}} - \underline{\boldsymbol{f}} \Big) = 0 \qquad \forall \underline{\boldsymbol{v}} \in (\mathbb{R}^3)^{N_f} \times \{\boldsymbol{0}\}^{N_v - N_f}, \tag{2.66}$$

using the (global) *mass matrix* $\boldsymbol{M}$, the (global) *stiffness matrix* $\boldsymbol{K}$, and the (global) *load vector* $\underline{\boldsymbol{f}}$, which are assembled from the individual element matrices and element load vectors by considering that adjacent elements share vertices:

$$\boldsymbol{M}_{[k\ell]} = \sum_{\substack{e,i,j\\ k=s(e,i)\\ \ell=s(e,j)}} \boldsymbol{M}^e_{[ij]} \qquad k,\ell = 1,\dots,N_v, \tag{2.67}$$

$$\underline{\boldsymbol{f}}_{[k]} = \sum_{\substack{e,i\\ k=s(e,i)}} \underline{\boldsymbol{f}}^e_{[i]} \qquad k = 1,\dots,N_v. \tag{2.68}$$

$\boldsymbol{K}$ is defined analogously to $\boldsymbol{M}$. The notation $\boldsymbol{A}_{[k\ell]}$ for a matrix $\boldsymbol{A}$ denotes entry (k,ℓ) when considering $\boldsymbol{A}$ as consisting of 3 $\times$ 3-matrix entries (i.e., $\boldsymbol{A}_{[k\ell]}$ is a 3 $\times$ 3-matrix). Analogously, the notation $\boldsymbol{b}_{[k]}$ for a vector $\boldsymbol{b}$ denotes entry k when considering $\boldsymbol{b}$ as consisting of 3-vector entries (i.e., $\boldsymbol{b}_{[k]}$ is a 3-vector).

The spatially discretized dynamic elasticity problem can finally be written as

$$[\boldsymbol{M}\ddot{\underline{\boldsymbol{u}}} + \boldsymbol{K}\underline{\boldsymbol{u}} = \underline{\boldsymbol{f}}]_{[k]} \qquad \text{for} \quad k = 1,\dots,N_f, \tag{2.69a}$$

$$\boldsymbol{u}_k = \boldsymbol{u}^0_k \qquad \text{for} \quad k = N_f+1,\dots,N_v, \tag{2.69b}$$

where the notation $[.=.]_{[k]}$ is used as a simplified notation for $[.]_{[k]} = [.]_{[k]}$. The dynamic elasticity problem is thus represented by N_v vector-valued equations. We associate equation k with vertex k, and therefore refer to this equation as *per-vertex equation* of vertex k.

In the following, we also consider for each element e the formal equations

$$[\boldsymbol{M}^e\ddot{\underline{\boldsymbol{u}}}^e + \boldsymbol{K}^e\underline{\boldsymbol{u}}^e = \underline{\boldsymbol{f}}^e]_{[i]} \qquad \text{for} \quad i = 1,\dots,8, \tag{2.70}$$

which we refer to as *per-element equations* of element e. The assembly of the per-vertex equations (2.69) from the per-element equations (2.70) according to Equations (2.67) and (2.68) can be interpreted as maintaining the balance of forces at each vertex of the finite element grid.

We apply velocity-dependent *Rayleigh damping* by replacing $\underline{\boldsymbol{f}}$ with $\underline{\boldsymbol{f}} - \boldsymbol{D}\dot{\underline{\boldsymbol{u}}}$, which leads to

$$[\boldsymbol{M}\ddot{\underline{\boldsymbol{u}}} + \boldsymbol{D}\dot{\underline{\boldsymbol{u}}} + \boldsymbol{K}\underline{\boldsymbol{u}} = \underline{\boldsymbol{f}}]_{[k]} \qquad \text{for} \quad k = 1,\dots,N_f, \tag{2.71a}$$

$$\boldsymbol{u}_k = \boldsymbol{u}^0_k \qquad \text{for} \quad k = N_f+1,\dots,N_v, \tag{2.71b}$$

where the (global) *damping matrix* $\boldsymbol{D}$ is defined as $\boldsymbol{D} = \alpha_1\boldsymbol{M} + \alpha_2\boldsymbol{K}$. $\alpha_1, \alpha_2 \geq 0$ are the mass and stiffness proportional damping parameters.

The respective per-element equations are

$$[\boldsymbol{M}^e\ddot{\underline{\boldsymbol{u}}}^e + \boldsymbol{D}^e\dot{\underline{\boldsymbol{u}}}^e + \boldsymbol{K}^e\underline{\boldsymbol{u}}^e = \underline{\boldsymbol{f}}^e]_{[i]} \qquad \text{for} \quad i = 1, \ldots, 8, \tag{2.72}$$

where $\boldsymbol{D}^e = \alpha_1\boldsymbol{M}^e + \alpha_2\boldsymbol{K}^e$ is the *element damping matrix* of element e.

For the static case, inertia and damping are neglected. This leads to the per-vertex equations

$$[\boldsymbol{K}\underline{\boldsymbol{u}} = \underline{\boldsymbol{f}}]_{[k]} \qquad \text{for} \quad k = 1, \ldots, N_f, \tag{2.73a}$$

$$\boldsymbol{u}_k = \boldsymbol{u}_k^0 \qquad \text{for} \quad k = N_f + 1, \ldots, N_v, \tag{2.73b}$$

and to the per-element equations

$$[\boldsymbol{K}^e\underline{\boldsymbol{u}}^e = \underline{\boldsymbol{f}}^e]_{[i]} \qquad \text{for} \quad i = 1, \ldots, 8. \tag{2.74}$$

2.2.3 Computation of the Element Matrices and the Load Vector

In this thesis, we employ uniform Cartesian and adaptive octree finite element grids. Therefore, all finite elements have the same regular shape. Assuming that the density and the Young's modulus are specified on a per-element basis, and that the Poisson's ratio is constant over the entire object, all element matrices can be derived from the element matrices of a single, generic element.

Consider a generic element e_0 with density ρ^{e_0}, Young's modulus E^{e_0}, and domain $\Omega^{e_0} = [x_1^{e_0}, x_1^{e_0} + a_1^{e_0}] \times [x_2^{e_0}, x_2^{e_0} + a_2^{e_0}] \times [x_3^{e_0}, x_3^{e_0} + a_3^{e_0}]$. Furthermore, consider an arbitrary element e with density $\rho^e = \rho\rho^{e_0}$, Young's modulus $E^e = EE^{e_0}$, and domain $\Omega^e = [x_1^e, x_1^e + aa_1^{e_0}] \times [x_2^e, x_2^e + aa_2^{e_0}] \times [x_3^e, x_3^e + aa_3^{e_0}]$, where ρ, E, and a denote the relative density, Young's modulus, and size of the element with respect to the generic element.

Using

$$\phi_i^e(\boldsymbol{x}) = \phi_i^{e_0}(\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}) \tag{2.75}$$

$$\Rightarrow \qquad \boldsymbol{\Phi}^e(\boldsymbol{x}) = \boldsymbol{\Phi}^{e_0}(\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}), \tag{2.76}$$

and the integration by substitution theorem, the element mass matrix $\boldsymbol{M}^e$ can be obtained from the generic element mass matrix $\boldsymbol{M}^{e_0}$ according to

$$\begin{aligned}
\boldsymbol{M}^e &= \int\limits_{\Omega^e} \rho^e (\boldsymbol{\Phi}^e)^{\mathrm{T}} \boldsymbol{\Phi}^e \, dx \\
&= \int\limits_{\Omega^e} \rho \rho^{e_0} (\boldsymbol{\Phi}^{e_0} (\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}))^{\mathrm{T}} \boldsymbol{\Phi}^e (\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}) \, dx \\
&= \rho \int\limits_{\Omega^{e_0}} \rho^{e_0} (\boldsymbol{\Phi}^{e_0}(\boldsymbol{x}))^{\mathrm{T}} \boldsymbol{\Phi}^{e_0}(\boldsymbol{x}) a^3 \, dx \\
&= \rho a^3 \int\limits_{\Omega^{e_0}} \rho^{e_0} (\boldsymbol{\Phi}^{e_0})^{\mathrm{T}} \boldsymbol{\Phi}^{e_0} \, dx \\
&= \rho a^3 \boldsymbol{M}^{e_0}.
\end{aligned} \tag{2.77}$$

According to Equations (2.40) and (2.43), for isotropic materials, the elasticity tensor depends linearly on the Young's modulus, i.e., $\boldsymbol{C}^e = E\boldsymbol{C}^{e_0}$. Using

$$\phi_i^e(\boldsymbol{x}) = \phi_i^{e_0}(\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}) \tag{2.78}$$

$$\Rightarrow \qquad \frac{\partial \phi_i^e(\boldsymbol{x})}{\partial x_j} = \frac{1}{a} \frac{\partial \phi_i^e(\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0})}{\partial x_j} \tag{2.79}$$

$$\Rightarrow \qquad \boldsymbol{B}^e(\boldsymbol{x}) = \frac{1}{a} \boldsymbol{B}^{e_0}(\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}), \tag{2.80}$$

and the integration by substitution theorem, the element stiffness matrix $\boldsymbol{K}^e$ can be obtained from the generic element stiffness matrix $\boldsymbol{K}^{e_0}$ according to

$$\begin{aligned}
\boldsymbol{K}^e &= \int\limits_{\Omega^e} (\boldsymbol{B}^e)^{\mathrm{T}} \overline{\boldsymbol{C}}^e \boldsymbol{B}^e \, dx \\
&= \int\limits_{\Omega^e} \frac{1}{a} (\boldsymbol{B}^{e_0}(\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}))^{\mathrm{T}} E \overline{\boldsymbol{C}}^{e_0} \frac{1}{a} \boldsymbol{B}^{e_0}(\frac{1}{a}(\boldsymbol{x} - \boldsymbol{x}^e) + \boldsymbol{x}^{e_0}) \, dx \\
&= E \int\limits_{\Omega^{e_0}} \frac{1}{a} (\boldsymbol{B}^{e_0}(\boldsymbol{x}))^{\mathrm{T}} \overline{\boldsymbol{C}}^{e_0} \frac{1}{a} \boldsymbol{B}^{e_0}(\boldsymbol{x}) a^3 \, dx \\
&= Ea \int\limits_{\Omega^{e_0}} (\boldsymbol{B}^{e_0})^{\mathrm{T}} \overline{\boldsymbol{C}}^{e_0} \boldsymbol{B}^{e_0} \, dx \\
&= Ea \boldsymbol{K}^{e_0}.
\end{aligned} \tag{2.81}$$

The assembly of the element mass matrices can be further simplified by assuming that the mass of the object is concentrated at the vertices of the finite element grid (*mass lumping*). This reduces the element mass matrices to diagonal matrices: $\boldsymbol{M}^e_{[ii]} = \frac{1}{8}m^e\mathbf{1}_3$, $\boldsymbol{M}^e_{[ij]} = \mathbf{0}$ for $i \neq j$ $(i, j = 1, \ldots, 8)$, where m^e denotes the mass of the respective element.

If the external forces are applied at a set of discrete points $\{\boldsymbol{x}_i\}$, the computation of the element load vectors (Equation (2.64)) is simplified to

$$\underline{\boldsymbol{f}}^e = \sum_{\substack{i \\ \boldsymbol{x}_i \in \Omega^e}} (\boldsymbol{\Phi}^e(\boldsymbol{x}_i))^{\mathrm{T}} \boldsymbol{f}_i. \tag{2.82}$$

Here, $\boldsymbol{f}_i$ denotes the force vector that is applied at position $\boldsymbol{x}_i$ in the reference configuration. For each element e, the sum iterates over those forces which are applied within the element's domain, i.e., $\boldsymbol{x}_i \in \Omega^e$. Note that if $\boldsymbol{x}_i$ is located on a boundary shared among multiple elements, the respective force must be assigned to exactly one of these elements.

If the forces are applied at the vertices of the finite element grid, the load vector can be directly assembled from these forces according to

$$\underline{\boldsymbol{f}} = \left(\boldsymbol{f}_1^{\mathrm{T}}, \ldots, \boldsymbol{f}_{N^v}^{\mathrm{T}}\right)^{\mathrm{T}}, \tag{2.83}$$

where $\boldsymbol{f}_k$ denotes the force vector that is applied at vertex k.

2.2.4 Handling of Hanging Vertices in an Adaptive Octree Grid

In an adaptive octree finite element grid, *hanging vertices* occur between adjacent elements of different size. Hanging vertices are lying in the interior of another element's edge or face. For a continuous discretization of the displacement field, the displacement at a hanging vertex must be determined by linear (along edges) or bilinear (on faces) interpolation of the displacements at the incident vertices of the respective edge or face.

Without loss of generality, let $\{1, \ldots, N_v'\}$ and $\{N_v' + 1, \ldots, N_v\}$ denote the indices of the non-hanging and the hanging vertices, respectively. The non-hanging vertices are further classified into free and fixed vertices with indices $\{1, \ldots, N_f\}$ and $\{N_f + 1, \ldots, N_v'\}$, respectively.

We use the notation $\underline{*}' = \left(*_1^{\mathrm{T}}, \ldots, *_{N_v'}^{\mathrm{T}}\right)^{\mathrm{T}}$ to refer to the linearization of the displacements at all non-hanging vertices. Let I be an interpolation matrix such that

$$\underline{\boldsymbol{v}} = \boldsymbol{I}\underline{\boldsymbol{v}}'. \tag{2.84}$$

The variational problem (2.46) is then considered over the function space

$$\mathcal{V}'_h = \left\{ \boldsymbol{v}_h \;\middle|\; \underline{\boldsymbol{v}} = \boldsymbol{I}\, \underline{\boldsymbol{v}}' \;,\; \underline{\boldsymbol{v}}' \in (\mathbb{R}^3)^{N'_v} \right\}, \tag{2.85}$$

which differs from the original function space (Equation (2.51)) in that the displacements at hanging vertices do not represent degrees of freedom, but are determined by interpolation from non-hanging vertices. It is

$$\mathcal{V}'_{h,0} = \left\{ \boldsymbol{v}_h \;\middle|\; \underline{\boldsymbol{v}} = \boldsymbol{I}\, \underline{\boldsymbol{v}}' \;,\; \underline{\boldsymbol{v}}' \in (\mathbb{R}^3)^{N_f} \times \{\mathbf{0}\}^{N'_v - N_f} \right\}. \tag{2.86}$$

Proceeding as in Section 2.2.2, Equation (2.66) becomes

$$\underline{\boldsymbol{v}}'^{\mathrm{T}} \boldsymbol{I}^{\mathrm{T}} \Big(\boldsymbol{M}\boldsymbol{I}\underline{\ddot{\boldsymbol{u}}}' + \boldsymbol{D}\boldsymbol{I}\underline{\dot{\boldsymbol{u}}}' + \boldsymbol{K}\boldsymbol{I}\underline{\boldsymbol{u}}' - \underline{\boldsymbol{f}} \Big) = 0 \qquad \forall \underline{\boldsymbol{v}}' \in (\mathbb{R}^3)^{N_f} \times \{\mathbf{0}\}^{N'_v - N_f}, \tag{2.87}$$

and the spatially discretized dynamic elasticity problem (2.71) becomes

$$\left[\boldsymbol{I}^{\mathrm{T}} \Big(\boldsymbol{M}\boldsymbol{I}\underline{\ddot{\boldsymbol{u}}}' + \boldsymbol{D}\boldsymbol{I}\underline{\dot{\boldsymbol{u}}}' + \boldsymbol{K}\boldsymbol{I}\underline{\boldsymbol{u}}' \Big) = \boldsymbol{I}^{\mathrm{T}} \underline{\boldsymbol{f}} \right]_{[k]} \qquad \text{for} \quad k = 1, \ldots, N_f, \tag{2.88a}$$

$$\boldsymbol{u}_k = \boldsymbol{u}_k^0 \qquad \text{for} \quad k = N_f + 1, \ldots, N'_v. \tag{2.88b}$$

Note that in this system of equations the displacement vectors $\boldsymbol{u}_k$ at hanging vertices have been eliminated. The system is obtained from the system (2.71) by substituting the displacements at hanging vertices by interpolation from the displacements at non-hanging vertices, and 'distributing' the equations at hanging vertices to non-hanging vertices using the same weights as used for interpolation.

2.2.5 Corotational Formulation of Strain

Linear elasticity comes with the drawback that it is accurate only for small strains and small rotations. It is based on the infinitesimal strain tensor, which is a linear approximation of the Green-St. Venant strain tensor, i.e., there is a linear relationship between strains and displacements. Since rotations are interpreted by the infinitesimal strain tensor as strains, rotations lead to the introduction of artificial stresses, which can result in a significant volume increase in case of large rotations. To overcome this limitation, we use the corotational strain formulation [RB86], which in principle removes the per-element rigid body rotations from the deformation field before the strain field is computed, and then reapplies the rotations to the resulting stress field. By using the corotational strain formulation, deformations with large rotations can be handled accurately. The efficient integration of the corotational formulation into real-time approaches has been demonstrated in [MDM$^+$02, HS04, GW08].

The corotation is carried out on the finite element discretization, i.e., a single rotation matrix $\boldsymbol{R}^e$ is computed for each finite element e, specifying the rigid body rotation component of the element's deformation (i.e., the rotation is from the reference into the deformed configuration). We obtain the element rotation by computing the polar decomposition of the element's average deformation gradient

$$\begin{aligned}\boldsymbol{F}^e &= \int\limits_{\Omega^e} \boldsymbol{F}\, dx \;/ \int\limits_{\Omega^e} 1\, dx \\ &= \mathbf{1}_3 + \frac{1}{4}\sum_{i=1}^{8} \boldsymbol{u}_{s(e,i)} \left((-1)^i \frac{1}{a_1^e}, (-1)^{\lceil i/2 \rceil} \frac{1}{a_2^e}, (-1)^{\lceil i/4 \rceil} \frac{1}{a_3^e} \right),\end{aligned} \tag{2.89}$$

where a_1^e, a_2^e, a_3^e denote the edge lengths of the hexahedral finite element. According to the polar decomposition theorem, every non-singular matrix $\boldsymbol{F}^e$ can be uniquely decomposed into the product of an orthogonal matrix $\boldsymbol{R}^e$ and a symmetric positive definite matrix $\boldsymbol{S}^e$, i.e., $\boldsymbol{F}^e = \boldsymbol{R}^e\boldsymbol{S}^e$. To approximately compute the polar decomposition, we perform 5 iterations of the following algorithm (i.e., $\boldsymbol{R}^e \approx \boldsymbol{R}_5^e$) [Hig86]:

$$\boldsymbol{R}_0^e = \boldsymbol{F}^e, \tag{2.90a}$$

$$\boldsymbol{R}_i^e = \frac{1}{2}\Big(\boldsymbol{R}_{i-1}^e + (\boldsymbol{R}_{i-1}^e)^{-\mathrm{T}}\Big) \qquad \text{for} \quad i > 0. \tag{2.90b}$$

Let

$$\widehat{\boldsymbol{R}}^e = \underbrace{\begin{pmatrix} \boldsymbol{R}^e & & \\ & \ddots & \\ & & \boldsymbol{R}^e \end{pmatrix}}_{8\,\times} \tag{2.91}$$

be a matrix with 8 instances of $\boldsymbol{R}^e$ on the diagonal. Furthermore, let $\boldsymbol{x}_k$ denote the position of vertex k in the reference configuration, and let $\underline{\boldsymbol{x}}^e = \big(\boldsymbol{x}_{s(e,1)}^{\mathrm{T}}, \ldots, \boldsymbol{x}_{s(e,8)}^{\mathrm{T}}\big)^{\mathrm{T}}$ be the linearization of the positions of the vertices of element e in the reference configuration.

The rotations are incorporated into the per-element equations (2.72) (for a uniform Cartesian finite element discretization) or into the per-element equations corresponding to the per-vertex equations (2.88a) (for an adaptive octree discretization) by replacing the term $\boldsymbol{K}^e\underline{\boldsymbol{u}}^e$ with the term $\widehat{\boldsymbol{R}}^e\boldsymbol{K}^e\left((\widehat{\boldsymbol{R}}^e)^{\mathrm{T}}(\underline{\boldsymbol{x}}^e + \underline{\boldsymbol{u}}^e) - \underline{\boldsymbol{x}}^e\right)$. For a uniform discretization, this leads to the modified per-element equations

$$\Big[\boldsymbol{M}^e\underline{\ddot{\boldsymbol{u}}}^e + \boldsymbol{D}^e\underline{\dot{\boldsymbol{u}}}^e + \underbrace{\widehat{\boldsymbol{R}}^e\boldsymbol{K}^e(\widehat{\boldsymbol{R}}^e)^{\mathrm{T}}}_{\widehat{\boldsymbol{A}}^e}\underline{\boldsymbol{u}}^e = \underbrace{\underline{\boldsymbol{f}}^e - \widehat{\boldsymbol{R}}^e\boldsymbol{K}^e\big((\widehat{\boldsymbol{R}}^e)^{\mathrm{T}}\underline{\boldsymbol{x}}^e - \underline{\boldsymbol{x}}^e\big)}_{\widehat{\boldsymbol{b}}^e}\Big]_{[i]} \qquad \text{for } i = 1, \dots, 8, \quad (2.92)$$

and to the corresponding modified per-vertex equations

$$[\boldsymbol{M}\underline{\ddot{\boldsymbol{u}}} + \boldsymbol{D}\underline{\dot{\boldsymbol{u}}} + \widehat{\boldsymbol{A}}\underline{\boldsymbol{u}} = \widehat{\boldsymbol{b}}]_{[k]} \qquad \text{for} \quad k = 1, \dots, N_f, \qquad (2.93a)$$

$$\boldsymbol{u}_k = \boldsymbol{u}_k^0 \qquad \text{for} \quad k = N_f + 1, \dots, N_v. \qquad (2.93b)$$

Here, $\widehat{A}$ and $\widehat{b}$ are assembled according to Equations (2.67) and (2.68), respectively.

In the next section, we explain how the semidiscrete initial value problem obtained from the finite element discretization is further discretized in time by introducing discrete time steps. The element rotations for the corotational strain formulation are updated once per time step. To incorporate the updated element rotations, the resulting linear system of equations has to be updated in every time step, too.

2.2.6 Time Discretization

We employ the implicit Newmark time integration scheme for the time discretization of the initial value problem. Using an implicit scheme allows us to use a reasonably large time step size dt. In the following, we use the superscript (n) to refer to quantities associated with the n-th time step.

The initial displacements $\boldsymbol{u}_k^{(0)}$ and initial velocities $\dot{\boldsymbol{u}}_k^{(0)}$ are given as initial conditions (2.30d) and (2.30e) of the initial boundary value problem, and the initial accelerations $\ddot{\boldsymbol{u}}_k^{(0)}$ are determined according to Equation (2.93) by

$$[\boldsymbol{M}\underline{\ddot{\boldsymbol{u}}}^{(0)} + \boldsymbol{D}\underline{\dot{\boldsymbol{u}}}^{(0)} + \widehat{\boldsymbol{A}}^{(0)}\underline{\boldsymbol{u}}^{(0)} = \underline{\widehat{\boldsymbol{b}}}^{(0)}]_{[k]} \qquad \text{for} \quad k = 1, \dots, N_f, \qquad (2.94a)$$

$$\ddot{\boldsymbol{u}}_k^{(0)} = \boldsymbol{0} \qquad \text{for} \quad k = N_f + 1, \dots, N_v. \qquad (2.94b)$$

The assembly of $\widehat{\boldsymbol{A}}^{(0)}$ and $\underline{\widehat{\boldsymbol{b}}}^{(0)}$ requires the element rotations $\boldsymbol{R}^{e\,(0)}$, which are computed based on the displacement vectors $\boldsymbol{u}_k^{(0)}$ according to Equations (2.89) and (2.90).

According to the Newmark time integration scheme, for $n > 0$ it is

$$\dot{\boldsymbol{u}}_k^{(n)} = \frac{2}{dt}\left(\boldsymbol{u}_k^{(n)} - \boldsymbol{u}_k^{(n-1)}\right) - \dot{\boldsymbol{u}}_k^{(n-1)}, \tag{2.95}$$

$$\ddot{\boldsymbol{u}}_k^{(n)} = \frac{4}{dt^2}\left(\boldsymbol{u}_k^{(n)} - \boldsymbol{u}_k^{(n-1)} - \dot{\boldsymbol{u}}_k^{(n-1)}dt\right) - \ddot{\boldsymbol{u}}_k^{(n-1)}. \tag{2.96}$$

Substituting Equations (2.95) and (2.96) into the initial value problem (2.93) leads for each time step n to a linear system of equations with the unknown displacement vectors $\boldsymbol{u}_k^{(n)}$:

$$\Bigg[\underbrace{\left(\frac{4}{dt^2}\boldsymbol{M} + \frac{2}{dt}\boldsymbol{D} + \widehat{\boldsymbol{A}}^{(n)}\right)}_{=\boldsymbol{A}^{(n)}}\underline{\boldsymbol{u}}^{(n)}$$

$$= \underbrace{\widehat{\boldsymbol{b}}^{(n)} + \boldsymbol{M}\left(\frac{4}{dt^2}\left(\underline{\boldsymbol{u}}^{(n-1)} + \underline{\dot{\boldsymbol{u}}}^{(n-1)}\right) + \underline{\ddot{\boldsymbol{u}}}^{(n-1)}\right) + \boldsymbol{D}\left(\frac{2}{dt}\underline{\boldsymbol{u}}^{(n-1)} + \underline{\dot{\boldsymbol{u}}}^{(n-1)}\right)}_{=\boldsymbol{b}^{(n)}}\Bigg]_{[k]}$$

$$\text{for} \quad k = 1, \dots, N_f \tag{2.97a}$$

$$\boldsymbol{u}_k^{(n)} = \boldsymbol{u}_k^0 \qquad \text{for} \quad k = N_f + 1, \dots, N_v \tag{2.97b}$$

The element rotations $\boldsymbol{R}^{e(n)}$ for the assembly of $\widehat{\boldsymbol{A}}^{(n)}$ and $\widehat{\underline{\boldsymbol{b}}}^{(n)}$ are computed based on the displacement vectors $\boldsymbol{u}_k^{(n-1)}$ of the previous time step.

In summary, in each time step $n > 0$, the following computations have to be performed:

		Equations
1.	For $n = 1$: Compute $\ddot{\boldsymbol{u}}_k^{(0)}$	(2.94)
	For $n > 1$: Compute $\dot{\boldsymbol{u}}_k^{(n-1)}$ and $\ddot{\boldsymbol{u}}_k^{(n-1)}$	(2.95), (2.96)
2.	Compute element rotations $\boldsymbol{R}^{e(n)}$ based on $\boldsymbol{u}_k^{(n-1)}$	(2.89), (2.90)
3.	Assemble $\widehat{\boldsymbol{A}}^{(n)}$ and $\widehat{\underline{\boldsymbol{b}}}^{(n)}$	(2.92), (2.97)
4.	Solve for $\underline{\boldsymbol{u}}^{(n)}$	(2.97)

2.3 Geometric Multigrid Solvers

Geometric multigrid solvers are known to be among the most efficient solvers for the linear systems of equations arising from the discretization of partial differential equations of the form described above. In this thesis, we employ a geometric multigrid method to solve the linear system of equations (2.97) (dynamic case) or (2.73) (static case).

In both cases, the linear system of equations is in the form

$$\begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{0} & \boldsymbol{1} \end{pmatrix} \begin{pmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \end{pmatrix} = \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{u}_2^0 \end{pmatrix}, \tag{2.98}$$

resulting from the classification of the vertices into N_f free and $N_v - N_f$ fixed vertices. This system is equivalent to the system

$$\underbrace{\begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{1} \end{pmatrix}}_{=\boldsymbol{A}^h} \underbrace{\begin{pmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \end{pmatrix}}_{=\boldsymbol{u}^h} = \underbrace{\begin{pmatrix} \boldsymbol{b}_1 - \boldsymbol{A}_{12}\boldsymbol{u}_2^0 \\ \boldsymbol{u}_2^0 \end{pmatrix}}_{=\boldsymbol{b}^h}. \tag{2.99}$$

$\boldsymbol{A}^h$ is a very large, sparse matrix, consisting of $3N_v$ rows and columns. Considering a uniform Cartesian finite element grid, each vertex has at most 3^3 adjacent vertices. Each matrix row and column thus has at most $3^3 \cdot 3 = 81$ potentially non-zero entries. $\boldsymbol{A}^h$ is symmetric and positive definite due to using hyperelastic materials with positive definite strain energy density functions, and due to a non-empty Dirichlet boundary in the static case or a non-zero mass in the dynamic case. Such a large, sparse linear system of equations is typically solved by using an iterative solution method.

2.3.1 The Multigrid V-Cycle

Let $\widetilde{\boldsymbol{u}}^h$ be the current approximate solution, and $\boldsymbol{r}^h = \boldsymbol{b}^h - \boldsymbol{A}^h\widetilde{\boldsymbol{u}}^h$ the current residual. The current error $\boldsymbol{e}^h = \boldsymbol{u}^h - \widetilde{\boldsymbol{u}}^h$ is determined by the residual equation $\boldsymbol{A}^h\boldsymbol{e}^h = \boldsymbol{r}^h$. By solving the residual equation, the exact solution can be obtained from the approximate solution according to $\boldsymbol{u}^h = \widetilde{\boldsymbol{u}}^h + \boldsymbol{e}^h$.

Basic iterative relaxation methods like Jacobi or Gauss-Seidel relaxation effectively reduce high-frequency (oscillatory) error components, but they are ineffective in reducing low-frequency (smooth) error components. This typically causes the error reduction to stall after a few iteration steps, i.e., the basic relaxation methods are very inefficient on a single grid.

The main idea of multigrid is to accelerate the error reduction by discretizing the considered problem on a hierarchy of successively coarser grids: The high-frequency error components can be effectively reduced on the original grid by performing a few relaxation steps. This leaves the low-frequency error components, i.e., the relaxation leads to a smoothing of the error. The error thus can be represented and solved for on a coarser grid, and then be used to correct the solution on the original grid.

On the coarser grid, the remaining low-frequency error components appear at a higher frequency. Therefore, the smoothing and coarse grid correction principle can be applied recursively on a hierarchy of successively coarser grids. On the coarsest grid, where the number of vertices/unknowns typically is very small, the residual equation can be solved directly.

This approach leads to significant performance gains since every error component can be effectively eliminated by a few relaxation steps on the appropriate grid, and, in addition, the number of vertices/unknowns (and thus the computational effort) on the successively coarser grids is significantly smaller than on the original grid. In particular, it can be shown that the resulting solver exhibits asymptotic linear runtime in the number of vertices/unknowns, considering the discretization of the underlying partial differential equation at ever smaller grid spacings.

In the following, we refer to the individual levels of the multigrid hierarchy by $h, 2h, 4h, \dots$, (relating to the ever coarser grid spacings), or by using level numbers $\ell = 0, 1, 2, \dots$, where level number 0 denotes the finest level of the hierarchy. Respective sub- and superscripts are used to refer to entities at these levels.

The components of a multigrid method are a smoothing procedure, a coarsening strategy to build the grid hierarchy, coarse grid operators (i.e., coarse grid versions of the system matrix), transfer operators to restrict the residual to the respectively next coarser grid and to interpolate the error back from this grid, and a cycle type (i.e., a specification of the traversal order of the levels of the multigrid hierarchy).

Using a two-level grid hierarchy leads to a *two-grid method*. In the following, we denote the coarse grid version of $\boldsymbol{A}^h$ by $\boldsymbol{A}^{2h}$, the restriction operator, which restricts the residual to the coarse grid, by $\boldsymbol{R}_h^{2h}$, and the interpolation operator, which interpolates the error back from the coarse grid, by $\boldsymbol{I}_{2h}^h$. Each iteration cycle of the two-grid method, which is referred to as *two-grid cycle*, consists of the following steps:

1. Relax $\boldsymbol{A}^h \widetilde{\boldsymbol{u}}^h \approx \boldsymbol{b}^h$ (n_1 times).
2. Compute residual $\boldsymbol{r}^h = \boldsymbol{b}^h - \boldsymbol{A}^h \widetilde{\boldsymbol{u}}^h$.
3. Restrict residual to the coarse grid: $\boldsymbol{r}^{2h} = \boldsymbol{R}_h^{2h} \boldsymbol{r}^h$.
4. Solve residual equation on the coarse grid: $\boldsymbol{A}^{2h} \boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$.
5. Interpolate error from the coarse grid: $\widetilde{\boldsymbol{e}}^h = \boldsymbol{I}_{2h}^h \boldsymbol{e}^{2h}$.
6. Apply coarse grid correction: $\widetilde{\boldsymbol{u}}^h \leftarrow \widetilde{\boldsymbol{u}}^h + \widetilde{\boldsymbol{e}}^h$.
7. Relax $\boldsymbol{A}^h \widetilde{\boldsymbol{u}}^h \approx \boldsymbol{b}^h$ (n_2 times).

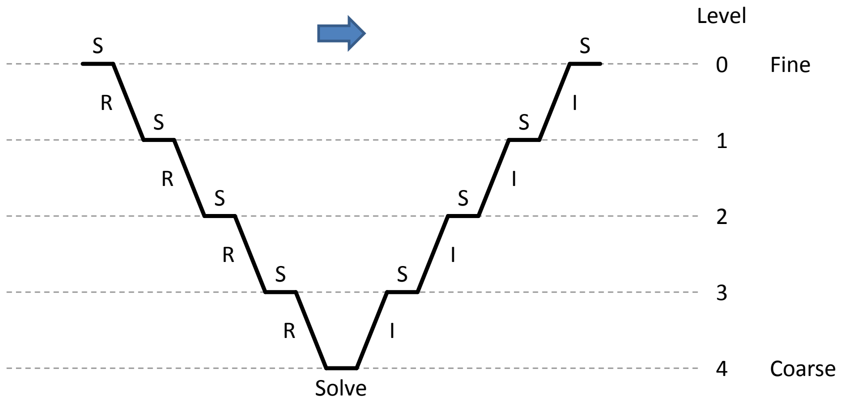


Figure 2.7: *Illustration of the multigrid V-cycle scheme. S denotes the pre- and post-smoothing, R denotes the computation of the residual and the restriction of the residual to the next coarser level, and I denotes the interpolation of the error from the previously visited coarser level to the current level and the coarse grid correction.*

Typical values for the number n_1 and n_2 of pre- and post-smoothing steps are 1 or 2. Throughout this thesis, we use decoupled Gauss-Seidel relaxation, which means that the three components of each (vector-valued) per-vertex equation are relaxed successively. The vertices are visited in lexicographical order with respect to their 3D position in the underlying grid. The relaxation of each scalar equation $i \in \{1, \ldots, N^h\}$ is performed according to

$$\widetilde{u}_i^h \leftarrow \widetilde{u}_i^h + \omega \frac{1}{A_{ii}^h} \left(b_i^h - \sum_{j=1}^{N^h} A_{ij}^h \widetilde{u}_j^h \right) . \tag{2.100}$$

Here, $0 < \omega < 2$ is an under-/overrelaxation parameter ($\omega = 1$ for no under-/over-relaxation), and N^h denotes the number of scalar unknowns.

On a hierarchy of multiple grids, the two-grid method is recursively applied to solve the residual equation (step 4) ($\boldsymbol{e}^{2h} \equiv \boldsymbol{u}^{2h}$, $\boldsymbol{r}^{2h} \equiv \boldsymbol{b}^{2h}$). In particular, using only a single two-grid cycle to approximately solve the residual equation leads to a *multigrid V-cycle* (see Figure 2.7): First, the grid hierarchy is traversed from the finest level to the second coarsest level. At each level, n_1 pre-smoothing relaxation steps are performed (step 1) (using an initial guess of $\mathbf{0}$ for $\boldsymbol{e}^{2h} \equiv \boldsymbol{u}^{2h}$, $\boldsymbol{e}^{4h} \equiv \boldsymbol{u}^{4h}, \ldots$), and the residual is computed (step 2) and restricted to the next coarser grid (step 3). On the coarsest grid, the residual equation is solved directly (step 4). Then, the hierarchy is traversed from the second coarsest level back to the finest level. At each level, the error is interpolated from the previously visited coarser grid back to the current grid (step 5). The interpolated

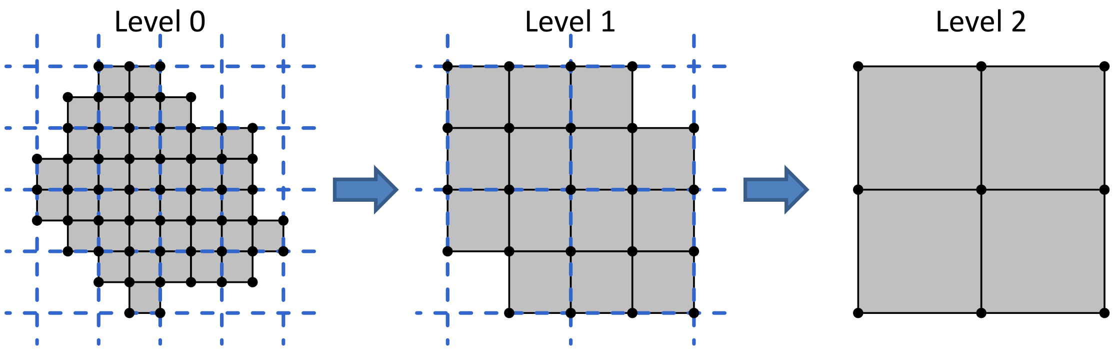


Figure 2.8: *The coarse grid hierarchy is constructed successively from the finest to the coarsest level.*

error is used to correct the solution on the current grid (step 6), and n_2 post-smoothing relaxation steps are performed (step 7).

More sophisticated traversal orders such as the *multigrid W-cycle* or the *full multigrid cycle* typically exhibit better convergence rates *per cycle*, but also increase the computing time per cycle. In our experiments we found that for our application the V-cycle offers the best convergence rate *with respect to computing time*.

In the following, we explain the coarsening strategy that we use, as well as our choices of the coarse grid and transfer operators.

2.3.2 Coarse Grid Hierarchy

In this thesis, we consider a finite element decomposition of the object based on a uniform Cartesian grid or an adaptive octree grid. The regular hexahedral grid structure gives rise to a very efficient construction of a nested grid hierarchy, which is essential for exploiting geometric multigrid schemes at their full potential.

Let us first consider a finite element decomposition based on a uniform Cartesian grid. We build the coarse grid hierarchy successively from the finest to the coarsest level (see Figure 2.8). With each coarser level, the grid cell size is doubled, such that the domain of a cell on the next coarser level coincides with the domain of a block of 2^3 cells on the current level. The respectively next coarser level is constructed by creating a cell if it covers at least one cell on the current level. At each hierarchy level a shared vertex representation is generated. Note that this construction process does not impose any restrictions on the size of the initial grid at the finest level. In particular, a cell at the next coarser level is allowed to be only partially ‘filled’ with cells on the current level, for instance at the object’s boundary. This construction principle, which has already been used in the context of composite finite elements [LPR+09], thus enables to automatically create a coarse grid hierarchy independently of the complexity of the shape of the object.

For a finite element decomposition based on an adaptive octree grid, the initial grid (level $\ell = 0$) consists of cells of sizes 2^i $(i = 0, 1, ...)$ with respect to the finest octree level. The construction of the coarse grid hierarchy is performed in a similar way as for the uniform case. At the transition from level ℓ to level $\ell + 1$, cells of size 2^ℓ are merged into cells of size $2^{\ell+1}$. Cells that are larger than 2^ℓ are passed on to the next coarser level.

According to the described construction principle, cells are merged solely based on their spatial location. Thus, for objects with complicated (concave) boundaries, physically disconnected parts might be merged into the same coarse grid cell, which leads to a reduced coarse grid approximation quality and therefore to a reduced convergence rate. In Chapter 4, we will present a novel approach for the efficient simulation of cuts in deformable objects. The cutting of an object leads to very complex boundaries, where disconnected parts are located directly adjacent to each other. To avoid a reduction of the multigrid convergence rate, we will propose a novel grid hierarchy that explicitly represents the cuts on the coarse grids.

2.3.3 Coarse Grid Operators and Transfer Operators

Generally, there are two approaches for the generation of the coarse grid operators for a geometric multigrid solver. The first approach explicitly discretizes the partial differential equation on the individual grids of the hierarchy, and independently selects restriction and interpolation operators. The second approach is based on the *variational properties of multigrid* [BHM00]. Here, the partial differential equation is only discretized on the finest grid. The coarse grid versions of $\boldsymbol{A}^h$ are constructed successively from fine to coarse via $\boldsymbol{A}^{2h} = \boldsymbol{R}_h^{2h} \boldsymbol{A}^h \boldsymbol{I}_{2h}^h$ (so-called *Galerkin-based coarsening*), and the restriction operators are obtained by transposition of the interpolation operators, i.e., $\boldsymbol{R}_h^{2h} = \left(\boldsymbol{I}_{2h}^h\right)^{\mathrm{T}}$. Typically, the first approach is computationally less expensive, but can fail for problems with variable coefficients and complicated boundary conditions, which possibly cannot be adequately discretized on the coarse grids. We therefore employ the second approach, which automatically handles variable coefficients and complicated boundary conditions, and is more robust in that the convergence of the resulting multigrid scheme is guaranteed. Corresponding to the discretization of the displacements by trilinear shape functions on the finest grid, the error is represented by trilinear shape functions on the coarse grids. This leads to trilinear interpolation operators $\boldsymbol{I}_{2h}^h$.

The variational properties are motivated as follows: The goal is to determine the coarse grid correction such that the current error is reduced as much as possible. In other words, the goal is to determine the error $\boldsymbol{e}^{2h} \in \mathbb{R}^{N^{2h}}$ on the coarse grid such that the interpolated error $\widetilde{\boldsymbol{e}}^h = \boldsymbol{I}_{2h}^h \boldsymbol{e}^{2h}$ is a best-approximation of the error $\boldsymbol{e}^h$ on the fine grid, which is determined by the residual equation $\boldsymbol{A}^h \boldsymbol{e}^h = \boldsymbol{r}^h$:

$$\|\boldsymbol{e}^h - \boldsymbol{I}_{2h}^h \boldsymbol{e}^{2h}\|_{\boldsymbol{A}^h} = \min_{\boldsymbol{w}^{2h} \in \mathbb{R}^{N^{2h}}} \|\boldsymbol{e}^h - \boldsymbol{I}_{2h}^h \boldsymbol{w}^{2h}\|_{\boldsymbol{A}^h}. \tag{2.101}$$

Here, $\|\boldsymbol{v}\|_{\boldsymbol{A}} = \sqrt{\boldsymbol{v}^{\mathrm{T}} \boldsymbol{A} \boldsymbol{v}}$ is the $\boldsymbol{A}$-energy norm, where $\boldsymbol{A}$ is a symmetric, positive definite matrix.

It can be shown that Equation (2.101) is equivalent to

$$(\boldsymbol{v}^{2h})^{\mathrm{T}} (\boldsymbol{I}_{2h}^h)^{\mathrm{T}} \left(\boldsymbol{A}^h \boldsymbol{I}_{2h}^h \boldsymbol{e}^{2h} - \boldsymbol{r}^h\right) = 0 \qquad \forall \boldsymbol{v}^{2h} \in \mathbb{R}^{N^{2h}}. \tag{2.102}$$

This is equivalent to

$$\underbrace{(\boldsymbol{I}_{2h}^h)^{\mathrm{T}} \boldsymbol{A}^h \boldsymbol{I}_{2h}^h}_{=\boldsymbol{A}^{2h}} \boldsymbol{e}^{2h} = \underbrace{(\boldsymbol{I}_{2h}^h)^{\mathrm{T}}}_{=\boldsymbol{R}_h^{2h}} \boldsymbol{r}^h, \tag{2.103}$$

which directly yields the coarse grid operator $\boldsymbol{A}^{2h} = \boldsymbol{R}_h^{2h} \boldsymbol{A}^h \boldsymbol{I}_{2h}^h$ and the restriction operator $\boldsymbol{R}_h^{2h} = (\boldsymbol{I}_{2h}^h)^{\mathrm{T}}$.

For a symmetric, positive definite matrix $\boldsymbol{A}^h$, it can be shown [TOS01] that a multigrid method based on the variational principles is converging for any coarse grid hierarchy, any full rank interpolation operators, and any converging smoother (i.e., Gauss-Seidel relaxation, or damped Jacobi relaxation with a suitably selected underrelaxation parameter). Note that this guarantees a minimum level of robustness, but does not state anything about the convergence rate, which depends on the quality of the interplay between smoother and coarse grid correction. We will demonstrate the efficiency of our solver and its superior performance compared to alternative numerical solution methods in Chapters 3 and 4 by showing convergence plots for a number of experiments.

Instead of the linear system (2.99), we could equivalently consider the reduced system

$$\underbrace{\boldsymbol{A}_{11}}_{=\boldsymbol{A}^h} \underbrace{\boldsymbol{u}_1}_{=\boldsymbol{u}^h} = \underbrace{\boldsymbol{b}_1 - \boldsymbol{A}_{12} \boldsymbol{u}_2^0}_{=\boldsymbol{b}^h}, \tag{2.104}$$

where the unknown vector $\boldsymbol{u}_2$, comprising the displacements at fixed vertices, has been eliminated. Then, however, we would also have to remove degrees of freedom on the coarse grids of the multigrid hierarchy to preserve the full rank of the interpolation operators, and thus to guarantee the convergence of the solver. By not eliminating the displacements at fixed vertices such an adaptation is not required, allowing us to handle free and fixed vertices identically from an algorithmic point of view.

2.4 GPUs and GPU Computing

A *graphics processing unit* (GPU) is a specialized processor dedicated to 3D graphics rendering. GPUs are ubiquitous in today's personal computers and workstations. Early graphics hardware was limited to the function of a simple digital-analog converter, generating analog signals for a computer display from a 2D raster image stored in video memory. For 3D graphics rendering, this 2D raster image had to be computed on the CPU. With the increasing demand for real-time 3D graphics in computer games in the 1990s, CPU performance became a severe bottleneck. This led to the development of specialized graphics processing units, particulary designed to perform the traditional operations in 3D computer graphics at a significantly higher performance than a CPU. At the beginning, these GPUs performed only the rasterization and fragment processing (in particular texture filtering), while the geometry processing was still done on the CPU. The first GPU implementing the entire graphics pipeline was NVIDIA's NV10 in 1999. In the next years, the developments in GPU technology led from a configurable fixed-function pipeline to a programmable pipeline, capable of executing user-defined programs in the vertex and fragment stages. With each further GPU generation, particular limitations with respect to programmability and accessing of memory resources were alleviated. Today, GPUs provide a level of programmability which is comparable to that of CPUs, and memory resources can be accessed from all programmable stages of the pipeline. The pipeline is exposed by the standardized graphics APIs OpenGL and Microsoft's Direct3D, which are accompanied by the shader languages GLSL and HLSL, respectively, to program the individual stages of the pipeline. For the work presented in this thesis we use the Direct3D API.

Due to their massively parallel architecture particularly designed for high throughput, GPUs exhibit a tremendous floating point performance and memory bandwidth. Starting with the first programmable GPUs, research has been pursued to exploit this performance for compute-intensive general purpose applications, not necessarily related to computer graphics [OLG+07]. At the beginning, the GPU hardware was only accessible via graphics APIs, meaning that computations had to be manually mapped onto the graphics pipeline. This was achieved by implementing a computational kernel as a fragment program running in the fragment stage of the pipeline. A kernel was launched by rendering a simple geometry (such as a rectangle) in order to generate fragments and thus execute this fragment program on a set of data elements. The input and output data had to be mapped onto 2D textures. Early work in GPU computing furthermore had to cope with the limitations of the first GPU generations, in particular including the restriction to non-IEEE compliant single precision floating point arithmetic, missing support for integer and bit arithmetic, and limitations in the flexibility of memory access operations, i.e., it was possible to read from arbitrary memory locations via texture fetches, but write access was possible only to the memory location in the frame buffer that corresponds to the respective pixel (no scattered writes).

However, GPU vendors recognized the potential of GPUs for general purpose computing, and particularly tailored the architecture of their GPUs not only to high performance 3D graphics rendering, but also to serious high performance computing. In 2006, NVIDIA presented CUDA[3], referring to a specific GPU architecture that supports GPGPU computing, as well as to a GPU computing API. The CUDA API provides a hardware abstraction (programming model) and a software environment that expose the GPU's general purpose computing capabilities. The computational kernels are implemented using a C-like programming language. The CUDA API allows to flexibly access the GPU's computing and memory resources, without resorting to the graphics pipeline. In particular, it allows to access video memory over a linear address space, and enables to perform read and write accesses at arbitrary locations. The CUDA architecture was first implemented by NVIDIA's G80 GPU (presented in 2006). This GPU provided integer and bit arithmetic, but floating point arithmetic was limited to IEEE compliant single precision. NVIDIA's GT200 GPU (presented in 2008) then provided support for IEEE compliant double precision floating point arithmetic. However, the double precision performance was only 1/8 of the single precision performance. This limitation was removed by NVIDIA's GF100 "Fermi" GPU (presented in 2010), where double precision operations are running at $1/2$ of the speed of single precision operations.

The CUDA API, which is used in this thesis, is only available on NVIDIA GPUs. There are other GPU computing APIs, such as OpenCL and Microsoft's DirectCompute, which provide a programming model that is virtually identical to the CUDA programming model, but are independent from a particular hardware vendor. OpenCL is even available on non-GPU parallel hardware architectures, such as multi-core CPUs.

The system architecture of a modern personal computer or workstation is depicted in Figure 3.7 (page 71). Here, the GPU along with a dedicated video memory is located on a dedicated graphics card, which is connected to the system via the PCI Express

[3]Acronym for 'Compute Unified Device Architecture'

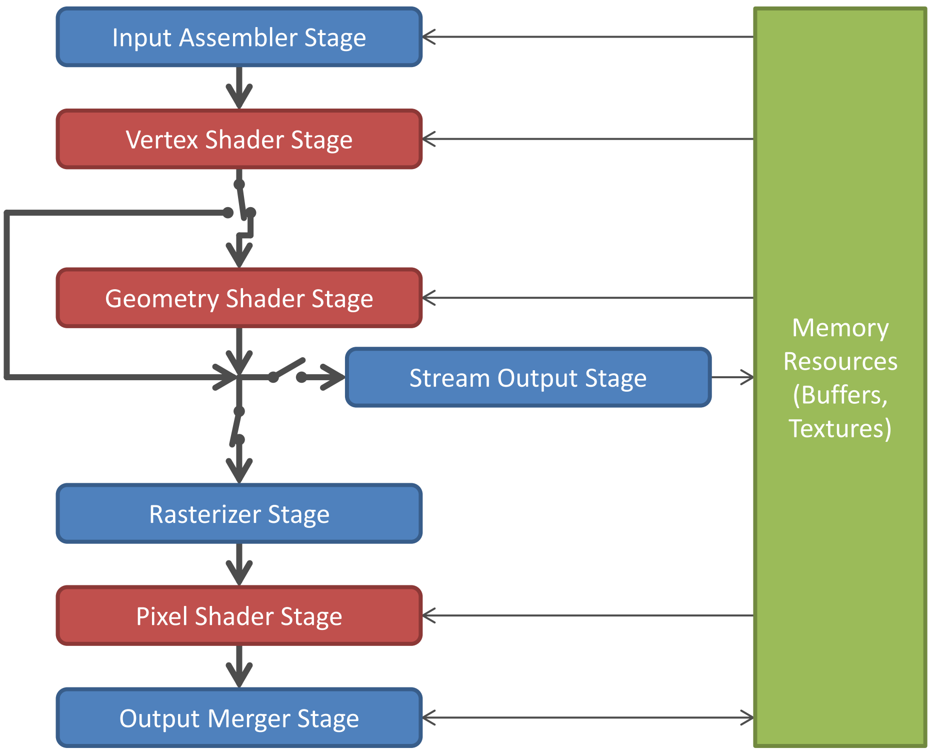


Figure 2.9: *Graphics pipeline as exposed by shader model 4.0 hardware and the Direct3D 10 API (according to [Mic10], adapted). The stages shown in red color are fully programmable. The geometry shader stage, the stream output stage, as well as the fragment processing part of the pipeline can be deactivated (this is indicated by the switches).*

bus. Note that low-end systems often are not equipped with a dedicated graphics card. Instead, the GPU is located on the motherboard, or CPU and GPU are even integrated into a single chip. The video memory then is often realized as a portion of the system's main memory (i.e., the main memory is shared between CPU and GPU).

In the following, we give a short introduction into the Direct3D 10 programmable graphics pipeline (Section 2.4.1) and into GPU computing based on the CUDA API (Section 2.4.2).

2.4.1 The Direct3D 10 Programmable Graphics Pipeline

The graphics pipeline is a hardware abstraction that describes the operation of the GPU for 3D graphics rendering. We describe the individual stages of this pipeline (see Figure 2.9) as it is exposed by shader model 4.0 hardware and the Direct3D 10 API [Mic10]:

- **Input Assembler Stage**. The geometry of a 3D scene is composed of geometric primitives, i.e., points, lines, and triangles, each consisting of one, two, or three vertices. The vertices are accompanied by per-vertex attributes such as position, normal, color, or texture coordinates. The input assembler stage reads the per-vertex attributes from video memory, assembles the vertices into primitives, and feeds these primitives as a stream into the graphics pipeline.

- **Vertex Shader Stage**. The vertex shader stage operates on the individual vertices by modifying their per-vertex attributes, including removing or adding attributes. This stage is fully programmable, and is typically used to apply a sequence of coordinate transformations to the vertices, i.e., the world, view, and projection transformation.
- **Geometry Shader Stage**. The geometry shader stage takes individual primitives as input. For each incoming primitive, the geometry shader can output a small, variable number of primitives (including the possibility to output zero primitives), with each primitive consisting of vertices accompanied by per-vertex attributes. The output primitives have to be of a fixed type, which may be different than the type of the input primitives. This stage is fully programmable, and is typically used for amplification or de-amplification of geometry.
- **Stream Output Stage**. The stream output stage can be used to write the stream of primitives emitted by the vertex or geometry shader stage back into video memory. The primitives can then be re-fed into the pipeline in a subsequent rendering pass.
- **Rasterizer Stage**. The rasterizer stage clips each primitive at the view frustum and performs the perspective division and view port transformation to transform the vertices from homogeneous clip space into screen space. It then performs the rasterization (scan-conversion) of the primitive, i.e., it generates a fragment for each covered pixel. In this way, the rasterizer stage converts the stream of primitives into a stream of fragments. It determines the per-fragment attributes such as depth, normal, color, or texture coordinates by linear interpolation (with perspective correction, corresponding to a linear interpolation in 3D view space) from the per-vertex attributes.
- **Pixel Shader Stage**. The pixel shader stage operates on the individual fragments and computes the fragment's color value from the per-fragment attributes. This stage is fully programmable and is typically used to perform texturing and lighting computations. It provides the possibility to modify a fragment's depth value, or to discard a fragment.

- **Output Merger Stage**. The output merger stage writes the fragments into the frame buffer. It performs depth and stencil testing, as well as blending operations.

2.4.2 The CUDA Architecture and Computing API

The architecture of a GPU is particularly tailored to achieving high throughput on data parallel computations, where the same operations are performed on a large number of data elements—in computer graphics, these data elements are geometric primitives, vertices, and fragments. Such an architecture is characterized by a massively parallel design based on a large number of processing units.

In 2006, NVIDIA presented *CUDA* [NVI10b], referring to a GPU architecture and a GPU computing API that enable general purpose GPU computing on NVIDIA GPUs. The CUDA architecture is based on a GPU design where dedicated fragment and vertex processors of previous GPU generations were replaced by unified processors executing vertex, geometry, fragment, and computing programs. In contrast to previous GPU generations, these processors solely execute scalar operations. An overview of different generations of NVIDIA GPUs based on this architecture is given in Table 2.1. The CUDA API provides a hardware abstraction (programming model) and a software environment that expose the GPU's computing capabilities. The computational kernels are implemented using a C-like programming language. In the following, we give a short introduction into GPU computing using the CUDA API and NVIDIA's GF100 ("Fermi") GPU, introduced in 2010 [NVI09].

CPUs and GPUs exhibit different architectures and programming models. CPUs are designed to execute one to a few different instruction streams (task parallelism). To increase operational throughput, modern CPU designs are based on detecting instruction level parallelism within the sequential instruction streams to simultaneously feed multiple execution units, leading to superscalar pipelines with out-of-order execution. To hide memory access latencies, large caches are used. As a consequence, a large portion of the CPU's die is used for sophisticated control logic and large caches, and only a small portion for a small number of execution units.

In contrast, GPUs are designed to execute a very high number of identical instruction streams, each performing the same operations on a single data element (data parallelism). To increase operational throughput, a GPU design is based on a large number of execution units, which can easily be fed by independent instructions from separate instruction streams. Memory access latencies are hidden by instantaneously switching between threads, which are permanently resident. This results in a die usage where a large portion is used for a large number of execution units, and only a small portion

Graphics Card	NVIDIA GeForce GTX 8800	NVIDIA GeForce GTX 280	NVIDIA GeForce GTX 480	NVIDIA Tesla C2070
Year of Introduction	2006	2008	2010	2010
GPU	G80	GT200	GF100	GF100
GPU Architecture Code Name	Tesla	Tesla	Fermi	Fermi
# Streaming Multiprocessors (SM)	16	30	15	14
# CUDA Cores per SM	8	8	32	32
# CUDA Cores total	128	240	480	448
# Registers per SM	8K	16K	32K	32K
Shared Memory per SM (KB)	16	16	48 or 16	48 or 16
Max. # Resident Threads per SM	768	1024	1536	1536
Processor Clock (MHz)	1350	1296	1401	1150
Peak # FLOPs per Clock Cycle				
• Single Precision	256	480	960	896
• Double Precision	—	60	120*	448
Peak FP Performance (GFLOPS)				
• Single Precision	346	622	1340	1030
• Double Precision	—	77.8	168*	515
Video Memory (MB)	768	1024	1536	6144
Memory Clock (MHz)	900	1107	1848	1500
Memory Interface Width (bit)	384	512	384	384
Memory Bandwidth (GB/s)	80.5	132	165	134
L1 Cache per SM (KB)	—	—	16 or 48	16 or 48
L2 Cache (KB)	—	—	768	768

Table 2.1: *Overview of NVIDIA graphics cards equipped with different generations of GPUs based on the CUDA architecture. *On the consumer-level NVIDIA GeForce GTX 480 graphics card, only 1/4 of the double precision floating point performance of the GF100 GPU is available. For comparison, a 4-core Intel Xeon X5560 processor (introduced in 2009) achieves a theoretical peak floating point performance of 89.6 GFLOPS for single and 44.8 GFLOPS for double precision, and offers a memory bandwidth of 29.8 GB/s (in combination with DDR3 1333MHz RAM).*

for a relatively simple control logic and small caches. In addition, since 3D graphics rendering is very memory-throughput intensive, GPUs are equipped with a memory interface that provides much higher throughput rates than that of a CPU.

As a consequence of the different architectures and programming models, GPUs exhibit a significantly higher (theoretical) peak floating point performance and memory throughput than CPUs, but are much more difficult to program, since computations and data structures have to be particularly mapped onto the specific, massively parallel GPU architecture in order to effectively exploit the GPU's resources.

In the CUDA programming model, the GPU acts as a coprocessor of the CPU. A computation is decomposed into sequential and parallel parts, which are executed on the CPU and GPU, respectively. Each parallel part is implemented as a single function

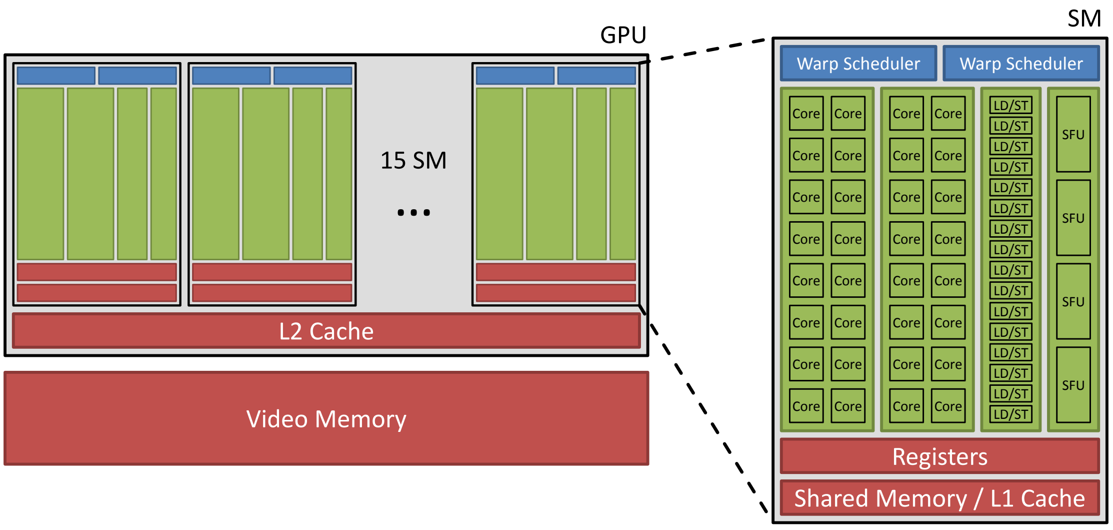


Figure 2.10: *Simplified schematic overview of the Fermi CUDA architecture (according to [NVI09], adapted).*

referred to as *kernel*, which is formulated in a C-like programming language. A kernel is executed many times in parallel. The GPU computation is initiated by the CPU, with the number of kernel executions being specified in the kernel call. For each execution of the kernel, a separate *thread* is spawned. The threads are executed in parallel on the GPU's processing units, with thread management and scheduling being performed entirely by the GPU. The kernels operate on data stored in video memory. These data are persistent over multiple kernel calls. Data between CPU and GPU are exchanged via memory copies between main memory and video memory, which are initiated by the CPU. These data are transferred over the PCI Express bus. Kernel and memory copies can be executed synchronously or asynchronously, i.e., the respective calls can be blocking or non-blocking. If asynchronous execution is used, computations on the CPU, computations on the GPU, and data transfers can overlap, which potentially increases overall performance.

The Fermi GPU consists of 15 *streaming multiprocessors* (SM) (see Figure 2.10). Each multiprocessor is equipped with 32 scalar *CUDA cores*[4] for integer and single and double precision floating point operations, 16 load/store units for memory access operations, as well as 4 special function units for transcendental operations. Each multiprocessor is equipped with a register file as well as a small low-latency on-chip memory block. Memory accesses to off-chip video memory are cached using a two-level cache hierarchy.

[4]In previous NVIDIA GPU generations, the CUDA cores were referred to as *streaming processors* (SP).

The threads are organized in groups of identical size, referred to as *thread blocks*. Threads are resident on the multiprocessors at thread block granularity: All threads of a thread block are concurrently and permanently resident on the same multiprocessor, until the execution of all threads of the thread block is completed. On each multiprocessor, multiple thread blocks can be concurrently resident. Only the threads in the same thread block can cooperate by sharing data via on-chip memory and by synchronizing their execution. The number of thread blocks as well as the number of threads per thread block is specified in the kernel call.

The thread blocks as well as the threads within each thread block are enumerated. Thus, each thread is identified by a unique $(\text{block ID}, \text{thread ID})$ pair. This pair can be accessed from within the kernel, and is used to adapt the runtime behavior of the threads, for example in such a way as each thread processes a different data element.

The threads of each thread block are executed in groups of 32 called *warps*. A multiprocessor executes the instructions of the threads of a warp in lock step. NVIDIA refers to this execution model as *SIMT* (single instruction, multiple threads), since the same instruction is executed simultaneously for multiple threads. Each instruction is issued to one of four groups of execution units (2 groups of 16 CUDA cores, 1 group of 4 load/store units, 1 group of 4 special function units). As a consequence, the GPU works most efficiently if all threads within one warp follow the same execution path. If the threads in a warp take different branches, these branches are executed sequentially (while some of the execution units are idling), which decreases overall performance. Note, however, that threads of different warps can take different branches without performance penalty. Each multiprocessor is equipped with two hardware schedulers, which simultaneously issue instructions from the set of warps that are resident on the multiprocessor. The hardware schedulers automatically schedule warps in such a way as to hide latencies, for example caused by memory access operations or data dependencies between successive instructions. Switching between warps has no cost, since the threads of a thread block are permanently resident on a multiprocessor (independently of whether they are running, blocked, or waiting). As a consequence, however, the register file is partitioned among *all* threads and the shared memory among *all* thread blocks residing on a multiprocessor, which significantly reduces the number of registers available to each thread and the amount of shared memory available to each thread block. Conversely, the number of threads and thread blocks that can be concurrently resident on a multiprocessor is determined by the amount of resources required by each thread and thread block. The more resources are required per thread and thread block, the smaller is the number of concurrently resident threads, and thus the smaller

Memory Space	Location	Read/Write	Scope	Lifetime
Register	On-chip	R/W	A single thread	Thread
Local	Off-chip	R/W	A single thread	Thread
Shared	On-chip	R/W	All threads in a single thread block	Thread block
Global	Off-chip	R/W	All threads and host	Application
Texture	Off-chip	R	All threads and host	Application
Constant	Off-chip	R	All threads and host	Application

Table 2.2: *Overview of the memory spaces available in the CUDA programming model (according to [NVI10a], adapted).*

are the thread scheduler's possibilities to hide latencies. Moreover, the total resource requirements of the threads of a thread block must not exceed the resource capacities of a multiprocessor, since all threads of a thread block are concurrently resident on a single multiprocessor.

The Fermi GPU executes accesses to off-chip video memory at a fix granularity of 128 bytes, i.e., the GPU reads or writes contiguous blocks of 128 bytes that are aligned at 128-byte boundaries. The hardware coalesces parallel memory accesses of the threads of a warp that lie in the same 128-byte segment into a single memory transaction. To reduce memory access latencies as well as to effectively exploit the GPU's memory bandwidth, parallel accesses of the threads of a warp should therefore lie closely packed in memory. Specifically, if the i-th thread of a warp (half warp) accesses the i-th 32-bit (64-bit) word of a 128-byte segment, these accesses are combined into a single memory transaction and the GPU's memory bandwidth is optimally used. Accesses to off-chip video memory are cached by a two-level cache hierarchy of limited size.

In the CUDA programming model, several memory spaces are available (see Table 2.2 for an overview). *Registers* are local to a single thread. Since they are located on-chip, register accesses are very fast. Registers are managed entirely by the compiler to store automatic variables. *Local memory* is also local to a single thread, but in contrast to registers, local memory is located in off-chip video memory. Access speed is the same as for global memory, i.e., accesses are cached by the GPU's two-level cache hierarchy, and cache misses come with high latencies. Local memory is used by the compiler to temporarily spill registers when the number of available registers per thread is not sufficient. *Shared memory* is local to a single thread block, i.e., can be accessed by all threads of this thread block. Since shared memory is located in the small on-chip memory block of each multiprocessor, accesses are very fast. Shared memory is typically used for thread cooperation, i.e., to exchange data among the threads of a thread block, or as a manually managed cache for global memory to reduce the number of redundant global memory accesses. *Global memory* is accessible from all threads and the host CPU. In particular, data stored in global memory are persistent for the entire lifetime of an application. Global memory is located in off-chip video memory. Accesses to global memory are cached by the GPU's two-level cache hierarchy. In case of a cache miss, a global memory access comes with a significantly higher latency than an access to on-chip registers or shared memory. *Texture memory* and *constant memory* are two further memory spaces that are globally accessible to all threads, persistent for the lifetime of an application, and located in off-chip video memory. These two memory spaces are read-only from the GPU, and can be read and written by the host CPU via memory copies. In addition, parts of the global memory space can be blended into the texture memory space. Accesses to texture and constant memory are cached by a small, separate texture and constant cache, respectively. Using texture memory enables to exploit the GPU's texture units for (low-precision) interpolation and handling of boundary cases.

As a consequence of the specific parallel architecture of the GPU, to effectively exploit the GPU's high computational power and memory bandwidth, several programming guidelines have to be considered. First, a computation must exhibit a sufficient amount of fine-grained parallelism to fully occupy the GPU's large number of parallel processing units. Second, the threads in the same warp should follow the same execution path to avoid serial execution of different branches. Third, it is important to choose memory layouts that support coalescing of memory access operations to optimally use the GPU's memory bandwidth. Fourth, the resource requirements (registers and shared memory) per thread and thread block should be kept at a minimum such that a high number of threads can be concurrently resident on the GPU, which enables the hardware scheduler to effectively hide latencies. Fifth, it must be considered that only the threads in the same thread block can be synchronized during runtime of a kernel. Global synchronization can only be achieved by sequentially executing separate kernels.

Chapter 3

A Real-Time Multigrid Finite Hexahedra Method for Elasticity Simulation using CUDA

In this chapter, we present a multigrid approach for simulating linear elastic deformable objects in real time on recent NVIDIA GPU architectures. To accurately simulate deformations exhibiting large rotations we use the corotational formulation of strain. Our method is based on a finite element discretization of the deformable object using hexahedra. It draws upon recent work on multigrid schemes for the efficient numerical solution of partial differential equations on such discretizations. Due to the regular shape of the numerical stencil induced by the hexahedral regime, and since we use matrix-free formulations of all multigrid steps, computations and data layout can be restructured to avoid execution divergence of parallel running threads and to enable coalescing of memory accesses into single memory transactions. This enables to effectively exploit the GPU's parallel processing units and high memory bandwidth via the CUDA computing API. We demonstrate performance gains of up to a factor of 27 and 4 compared to a highly optimized CPU implementation on a single CPU core and 8 CPU cores, respectively. For hexahedral models consisting of as many as 269,000 elements our approach achieves physics-based simulation at 11 time steps per second.

This chapter is based on material that has been originally published in C. Dick, J. Georgii, and R. Westermann, *A real-time multigrid finite hexahedra method for elasticity simulation using CUDA*, Simulation Modelling Practice and Theory 19 (2011), no. 2, 801–816 [DGW11b]. The copyright for this material is owned by Elsevier.

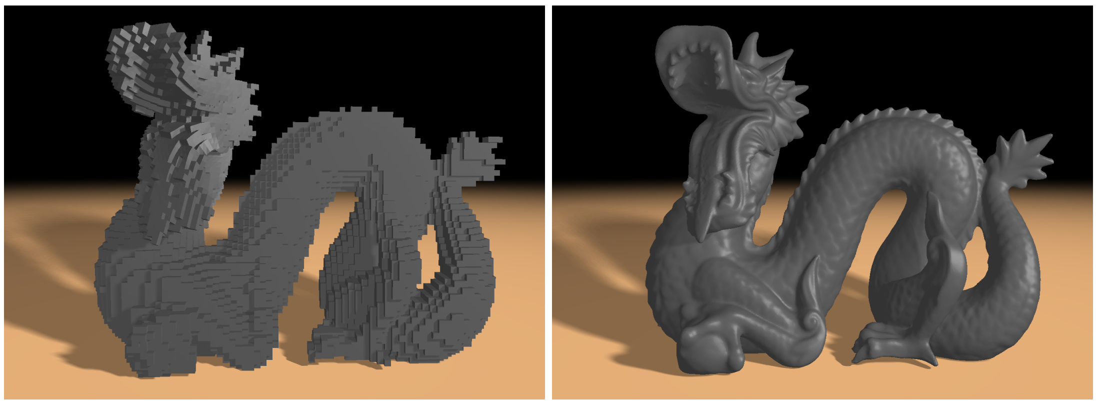


Figure 3.1: *Left: A deformed hexahedral object consisting of 30,000 elements is shown. Right: By using a high-resolution render surface that is bound to the deformed representation a visually continuous appearance is achieved.*

3.1 Introduction

Over the last years, graphics processing units (GPUs) have shown a substantial performance increase on intrinsically parallel computations. Key to this evolution is the GPU's design for massively parallel tasks, with the emphasis on maximizing total throughput of all parallel units. The ability to simultaneously use many processing units and to exploit thread level parallelism to hide latency have led to impressive performance increases in a number of scientific applications. One prominent example is NVIDIA's Fermi GPU [NVI09], on which we have based our current developments.

We present a novel geometric multigrid finite element method on the GPU, and we show the potential of this method for simulating elastic objects in real time on desktop PCs. To the best of our knowledge, this is the first multigrid finite element approach for solving linear elasticity problems that is realized entirely on the GPU. Since we use the corotational formulation of strain, even deformations that exhibit large rotations can be simulated at high physical accuracy. The CUDA computing API [NVI10b] is used because in contrast to graphics APIs like OpenGL or Direct3D it gives the programmer direct control over all available computing and memory resources on the GPU.

To effectively exploit the GPU's massively parallel multi-threading architecture, an algorithm must be restructured to expose a sufficient amount of fine-grained parallelism down or beyond one thread per data element. These threads should follow one common execution path and exhibit memory access patterns that enable coalescing of memory accesses to effectively exploit the massive number of processing units and the massive memory bandwidth available on the GPU.

The particular restructuring we propose is based on a regular hexahedral discretization of the simulation domain, which provides a number of advantages for GPU-based deformable object simulation: First, a hexahedral discretization of a given object boundary surface can be generated in a robust way and at very high speed, including a multi-resolution representation, which is required in a geometric multigrid approach. Second, the regular topology of the hexahedral grid leads to a numerical stencil of the same regular shape at each simulation vertex. This enables parallel processing of vertices using the same execution path and allows for memory layouts that support coalescing of memory access operations. Third, since all hexahedral elements have the same shape, only the precomputed element matrices of a single element are needed, which greatly reduces memory requirements. The mass and stiffness matrices of a specific finite element are obtained from these matrices by scaling with the element's density and Young's modulus, respectively.

Due to these advantages, we achieve performance gains of up to a factor of 27 compared to an optimized parallel CPU implementation running on a single CPU core. Even compared to the CPU implementation running on 8 CPU cores, our GPU implementation is a factor of up to 4 faster. This speed-up results from both the arithmetic and memory throughput on the Fermi GPU. Our CUDA implementation of the multigrid method achieves update rates of 120 time steps per second for models consisting of 12,000 hexahedral elements. For large models consisting of 269,000 elements, 11 time steps per second can still be achieved. Each time step includes the re-assembly of the system of equations, which is necessary due to the co-rotated strain formulation, as well as two multigrid V-cycles for solving this system. In combination with a high-resolution render surface, which is bound to the simulation model via precomputed interpolation weights, a visually continuous rendering of the deformable body is achieved (see Figure 3.1).

3.2 Related Work

Over the last years, considerable effort has been spent on the efficient realization of general techniques of numerical computing on programmable GPUs [HG07, OHL$^+$08]. Recent work in this field has increasingly focused on the use of the CUDA computing API [NVI10b], addressing a multitude of different applications ranging from image processing and scientific visualization to fluid simulation and protein folding.

Over the last decades, extensive research has been pursued on the use of three-dimensional finite element (FE) methods to predict the mechanical response of deformable materials to applied forces (see, for example, [Bat02] for a thorough overview). FE methods are attractive because they can realistically simulate the dynamic behavior of elastic materials, including the simulation of internal stresses due to exerted forces. Algorithmic improvements of FE methods, steering towards real-time simulation for computer animation and virtual surgery simulation have been addressed in [BNC96, CDA99, MDM$^+$02, EKS03].

In real-time applications, most commonly the linearized theory of elasticity based on the infinitesimal strain tensor is used. However, since the infinitesimal strain tensor is not invariant under rotations, computed displacements tend to diverge from the correct solution in case of large rotations. The corotational formulation of strain [RB86] accounts explicitly for the per-element rotations in the strain computation and thus can handle non-linear relations in the elastic quantities. The efficient integration of the corotational formulation into real-time approaches has been demonstrated in [MDM$^+$02, HS04, GW08].

Among the fastest numerical solution methods for solving the systems of linear equations arising in deformable model simulation are multigrid methods [Bra77, Hac85, BHM00]. In a number of previous works, geometric multigrid schemes for solving the partial differential equations describing elastic deformations have been developed [PH90, AD99, SB10]. Interactive multigrid approaches for simulating linear elastic materials on tetrahedral and hexahedral grids have been proposed in [WT04, GW06] and [DGBW08], respectively.

FE-based deformable body simulation on the GPU has been addressed in a number of publications. The exploitation of a GPU-based conjugate gradient solver for accelerating the numerical simulation of the FE model has been reported in [WH04, LJWD08]. [RNS06] presented a GPU-based FE surface method for cloth simulation. An overview of early GPU-accelerated techniques for surgical simulation is given by [SM06]. These approaches are mainly based on mass-spring systems [MHS05]. Finite element solvers for non-linear elasticity simulation using graphics APIs and CUDA were presented by [TCO08] and [CTA$^+$08], respectively. Both approaches build upon Lagrangian explicit dynamics [MJLW07] to avoid locking effects. While [TCO08] employed a tetrahedral domain discretization, a discretization using hexahedral finite elements was used by [CTA$^+$08]. [GSMY$^+$08] demonstrated clear performance gains for a multigrid Poisson solver on the GPU.

3.3 GPU-Aware Elasticity Simulation

In the following we outline the algorithms that are used to enable fast and stable numerical simulation of deformable bodies based on linear elasticity combined with the corotational formulation of strain to accurately simulate deformations with large rotations. Special emphasis is put on the restructuring of these algorithms to support an efficient mapping onto the GPU, involving matrix-free formulations of all computational steps.

3.3.1 Finite Element Discretization

Our approach is based on a finite element discretization of the deformable object using hexahedral elements with trilinear shape functions. The discretization is built from a voxelization of the object into a Cartesian grid structure, i.e., each voxel is classified as inside or outside of the object boundary. The simulation model is then obtained by creating a hexahedral finite element for each interior voxel.

The regular hexahedral structure gives rise to a very efficient construction of a nested grid hierarchy, which is essential for exploiting geometric multigrid schemes at their full potential. Due to the regular structure of the hexahedral discretization, computations can be parallelized effectively on SIMD architectures like GPUs.

The finite element discretization of the governing partial differential equation leads to the per-element equations

$$\sum_{j=1}^{8} \left(\boldsymbol{M}^e_{[ij]} \ddot{\boldsymbol{u}}_j + \boldsymbol{D}^e_{[ij]} \dot{\boldsymbol{u}}_j + \boldsymbol{K}^e_{[ij]} \boldsymbol{u}_j \right) = \boldsymbol{f}^e_{[i]} \quad , \quad i = 1, \ldots, 8, \tag{3.1}$$

where $\boldsymbol{M}^e$, $\boldsymbol{D}^e$, $\boldsymbol{K}^e$, and $\boldsymbol{f}^e$ are the element mass matrix, the element damping matrix, the element stiffness matrix, and the element load vector, respectively (see Chapter 2 for details). Here, $\boldsymbol{u}_j$ are the displacement vectors at the element's vertices ($j = 1, \ldots, 8$), and $\dot{\boldsymbol{u}}_j$ and $\ddot{\boldsymbol{u}}_j$ their first and second time derivatives.

Since all finite elements have the same shape, all element matrices can be obtained from the precomputed element matrices of a single, generic element by scaling with the element's density and Young's modulus values (see Section 2.2.3). We use velocity-dependent Rayleigh damping, meaning that the element damping matrices are defined by $\boldsymbol{D}^e = \alpha_1 \boldsymbol{M}^e + \alpha_2 \boldsymbol{K}^e$ with $\alpha_1, \alpha_2 \geq 0$. Requiring only a single element mass and stiffness matrix accelerates the setup phase for the simulation and significantly reduces memory requirements.

For the corotational formulation of strain, the term $\boldsymbol{K}^e_{[ij]} \boldsymbol{u}_j$ in Equation (3.1) is replaced by the term $\boldsymbol{R}^e\boldsymbol{K}^e_{[ij]}\left((\boldsymbol{R}^e)^{\mathrm{T}}(\boldsymbol{x}_j+\boldsymbol{u}_j)-\boldsymbol{x}_j\right)$, where $\boldsymbol{R}^e$ is the current element rotation, and $\boldsymbol{x}_j$ are the positions of the element's vertices in the reference configuration:

$$\sum_{j=1}^{8}\Big(\boldsymbol{M}^e_{[ij]}\ddot{\boldsymbol{u}}_j+\boldsymbol{D}^e_{[ij]}\dot{\boldsymbol{u}}_j+\underbrace{\boldsymbol{R}^e\boldsymbol{K}^e_{[ij]}(\boldsymbol{R}^e)^{\mathrm{T}}}_{\widehat{\boldsymbol{A}}^e_{[ij]}}\boldsymbol{u}_j\Big)=$$
$$\underbrace{\boldsymbol{f}^e_{[i]}-\boldsymbol{R}^e\boldsymbol{K}^e_{[ij]}\Big((\boldsymbol{R}^e)^{\mathrm{T}}\boldsymbol{x}_j-\boldsymbol{x}_j\Big)}_{\widehat{\boldsymbol{b}}^e_{[i]}}\quad,\quad i=1,\ldots,8. \tag{3.2}$$

Applying the Newmark time integration scheme

$$\dot{\boldsymbol{u}}_j=\frac{2}{dt}\left(\boldsymbol{u}_j-\boldsymbol{u}^{\mathrm{old}}_j\right)-\dot{\boldsymbol{u}}^{\mathrm{old}}_j, \tag{3.3a}$$
$$\ddot{\boldsymbol{u}}_j=\frac{4}{dt^2}\left(\boldsymbol{u}_j-\boldsymbol{u}^{\mathrm{old}}_j-\dot{\boldsymbol{u}}^{\mathrm{old}}_j dt\right)-\ddot{\boldsymbol{u}}^{\mathrm{old}}_j, \tag{3.3b}$$

finally leads to

$$\sum_{j=1}^{8}\underbrace{\left(\frac{4}{dt^2}\boldsymbol{M}^e_{[ij]}+\frac{2}{dt}\boldsymbol{D}^e_{[ij]}+\widehat{\boldsymbol{A}}^e_{[ij]}\right)}_{\boldsymbol{A}^e_{[ij]}}\boldsymbol{u}_j=$$
$$\underbrace{\widehat{\boldsymbol{b}}^e_{[i]}+\sum_{j=1}^{8}\left(\boldsymbol{M}^e_{[ij]}\left(\frac{4}{dt^2}\left(\boldsymbol{u}^{\mathrm{old}}_j+\dot{\boldsymbol{u}}^{\mathrm{old}}_j dt\right)+\ddot{\boldsymbol{u}}^{\mathrm{old}}_j\right)+\boldsymbol{D}^e_{[ij]}\left(\frac{2}{dt}\boldsymbol{u}^{\mathrm{old}}_j+\dot{\boldsymbol{u}}^{\mathrm{old}}_j\right)\right)}_{\boldsymbol{b}^e_{[i]}},$$
$$i=1,\ldots,8, \tag{3.4}$$

where $\boldsymbol{u}^{\mathrm{old}}_j$, $\dot{\boldsymbol{u}}_j{}^{\mathrm{old}}$, $\ddot{\boldsymbol{u}}^{\mathrm{old}}_j$ are the displacement vectors and their time derivatives of the previous time step, and dt denotes the length of the time step.

The global linear system of equations is assembled from the per-element equations by taking into account that elements share vertices, i.e., that there is one common $\boldsymbol{u}_j$ at a shared vertex. More precisely, a per-vertex equation is built for every vertex $\boldsymbol{x}=(x_1,x_2,x_3)$ of the finite element grid by accumulating the respective per-element equations from the 8 incident hexahedra. $\boldsymbol{x}$ denotes integer coordinates of the vertex with respect to the underlying hexahedral grid. This results in per-vertex equations that reside on a 3^3 stencil of 27 adjacent vertices (this stencil may be cropped at the object boundary):

$$\sum_{\boldsymbol{i}=-\boldsymbol{1}}^{\boldsymbol{1}}\boldsymbol{A}^{\boldsymbol{x}}_{\boldsymbol{i}}\;\boldsymbol{u}_{\boldsymbol{x}+\boldsymbol{i}}=\boldsymbol{b}_{\boldsymbol{x}}. \tag{3.5}$$

Here, $\boldsymbol{A}_{\boldsymbol{i}}^{\boldsymbol{x}}$ is the accumulated 3 × 3-matrix coefficient associated with the adjacent vertex $\boldsymbol{x}+\boldsymbol{i}$, where $\boldsymbol{i} = (i_1, i_2, i_3)$ is the relative position of the adjacent vertex with respect to vertex $\boldsymbol{x}$. The notation $\boldsymbol{i} = -\boldsymbol{1}, \ldots, \boldsymbol{1}$ means iterating over all 27 3-tuples of the set $\{-1, 0, 1\}^3$, i.e. $(-1,-1,-1), (0,-1,-1), (1,-1,-1), \ldots, (1,1,1)$. $\boldsymbol{u_x}$ denotes the displacement vector at vertex $\boldsymbol{x}$, and $\boldsymbol{b_x}$ is the accumulated right-hand side vector of the per-vertex equation of vertex $\boldsymbol{x}$. Dirichlet boundary conditions are implemented by replacing the per-vertex equations of fixed vertices with dummy equations $\boldsymbol{1}\boldsymbol{u_x} = \boldsymbol{u}_{\boldsymbol{x}}^0$. To maintain the symmetry of the global system matrix, we also adapt the per-vertex equations of the free vertices by bringing the coefficients associated with fixed vertices to the right-hand side (see Section 2.3).

3.3.2 Multigrid Solver

Numerical multigrid solvers are known to be among the most efficient solvers for the linear systems of equations arising from the discretization of elliptic second-order partial differential equations. Our geometric multigrid solver extends previous work by introducing a method to perform the computations for every element or vertex in lock-step using only coordinated memory accesses. Due to this property, the solver can effectively be mapped onto the GPU via the CUDA API.

Coarse Grid Hierarchy From a given hexahedral finite element grid, a coarse grid hierarchy is built in a bottom-up process by successively considering hexahedral grids of double cell size, i.e., the domain of a cell on the next coarser level coincides with the domain of a block of 2^3 cells on the current level. The respective next coarser level is constructed by creating a cell if it covers *at least one* cell on the current level. On each hierarchy level a shared vertex representation is computed. This process is repeated until the number of vertices on the coarsest level is below a given threshold. Note that this construction process does not impose any restrictions on the size of the initial hexahedral model on the finest level. In particular, a cell on the next coarser level is allowed to be only partially ‘filled’ with cells on the current level, for instance, at the object boundary.

Coarse Grid Equations In the following, the respective current level and the next coarser level of the grid hierarchy are indicated by sub- and superscripts h and $2h$, referring to the levels’ relative grid spacings. The multigrid solver requires coarse grid versions of the system matrix $\boldsymbol{A}$ as well as restriction and interpolation operators to transfer quantities between subsequent grid levels. We use trilinear interpolation for the

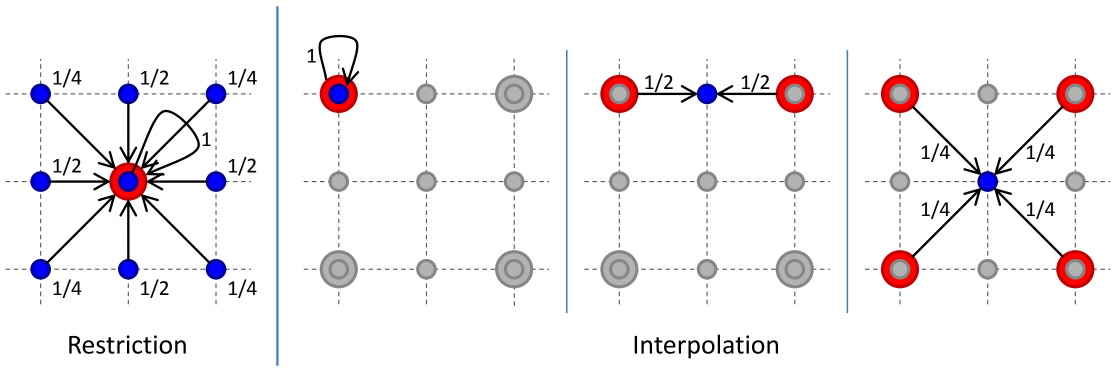


Figure 3.2: *Weights used to transfer quantities between two adjacent grids in the multigrid hierarchy (blue and red vertices belong to the finer and the coarser grid, respectively). For simplicity, the weights are shown only for selected vertices in 2D.*

multigrid interpolation operator $\boldsymbol{I}_{2h}^{h}$, and the multigrid restriction operator $\boldsymbol{R}_{h}^{2h}$ is chosen to be the transpose of the interpolation operator, i.e., $\boldsymbol{R}_{h}^{2h} = (\boldsymbol{I}_{2h}^{h})^{\mathrm{T}}$. Furthermore, we use Galerkin-based coarsening, i.e., the coarse grid versions of the system matrix are successively built from fine to coarse grids via $\boldsymbol{A}^{2h} = \boldsymbol{R}_{h}^{2h}\,\boldsymbol{A}^{h}\,\boldsymbol{I}_{2h}^{h}$. In a matrix-free formulation, the coarse grid equations are built by distributing the equations at the fine grid vertices to the coarse grid vertices and by simultaneously substituting the fine grid unknowns by interpolation from the coarse grid unknowns, using the weights illustrated in Figure 3.2.

To construct the equations on the coarse grids, we propose a two-step approach as illustrated in Figure 3.3. First, the equations at the fine grid vertices are distributed to the coarse grid vertices (this is the restriction $\boldsymbol{R}_{h}^{2h}$, corresponding to computing linear combinations of the rows of $\boldsymbol{A}^{h}$), yielding a 5^3 stencil on the fine grid with associated coefficients $\boldsymbol{B}$. Second, these coefficients are distributed to the coarse grid vertices (this is the interpolation $\boldsymbol{I}_{2h}^{h}$, corresponding to computing linear combinations of the columns of $\boldsymbol{A}^{h}$), thereby reducing the stencil to a 3^3 domain on the coarse grid. Note that the coarse grid vertex $\boldsymbol{x}$ corresponds to the fine grid vertex $2\boldsymbol{x}$ due to the different grid spacings. The construction is described by the following equations:

$$ {}^{h}\boldsymbol{B}_{\boldsymbol{i}}^{\boldsymbol{x}} = \sum_{\substack{\boldsymbol{k}=-\boldsymbol{1} \\ |i_j-k_j|\le 1\,,\, j=1,2,3}}^{\boldsymbol{1}} w_{\boldsymbol{k}}\ {}^{h}\boldsymbol{A}_{\boldsymbol{i}-\boldsymbol{k}}^{2\boldsymbol{x}+\boldsymbol{k}}\ , \qquad \boldsymbol{i} = -\boldsymbol{2},\ldots,\boldsymbol{2}, \tag{3.6} $$

$$ {}^{2h}\boldsymbol{A}_{\boldsymbol{i}}^{\boldsymbol{x}} = \sum_{\substack{\boldsymbol{k}=-\boldsymbol{1} \\ |2i_j+k_j|\le 2\,,\, j=1,2,3}}^{\boldsymbol{1}} w_{\boldsymbol{k}}\ {}^{h}\boldsymbol{B}_{2\boldsymbol{i}+\boldsymbol{k}}^{\boldsymbol{x}}\ , \qquad \boldsymbol{i} = -\boldsymbol{1},\ldots,\boldsymbol{1}. \tag{3.7} $$

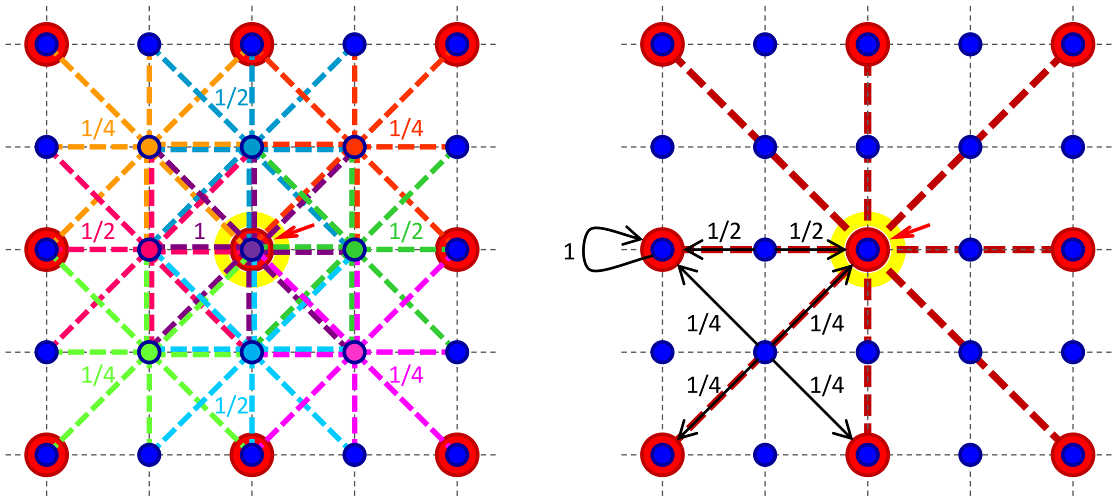


Figure 3.3: *Illustration of the construction of the coarse grid equation for a specific vertex (center vertex marked with a red arrow). In the first step (left), a linear combination of the per-vertex equations (their stencils and the used weights are shown in different colors) in the 3^3 fine grid (blue vertices) neighborhood of the considered vertex is computed. The resulting equation resides on a 5^3 stencil on the fine grid. In the second step (right), this equations is restricted to a 3^3 stencil (shown in red color) on the coarse grid (red vertices) by substituting the unknowns at the fine grid vertices by interpolation from the unknowns at the coarse grid vertices, corresponding to a distribution of the respective coefficients from the fine grid to the coarse grid vertices (black arrows and weights). Note that the weights shown in the figure correspond to the 2D case; the weights for the 3D case are given in the text.*

In these equations, $w_{\boldsymbol{k}} = (2 - |k_1|)\,(2 - |k_2|)\,(2 - |k_3|)/8$ are the weights used for restriction and interpolation. The additional conditions for the summation index variables ensure that no coefficients are fetched outside the valid ranges ($-\mathbf{1}, \ldots, \mathbf{1}$ for coefficients $\boldsymbol{A}$, and $-\mathbf{2}, \ldots, \mathbf{2}$ for coefficients $\boldsymbol{B}$).

Multigrid V-Cycle The linear system of equations is solved by performing multigrid V-cycles, each consisting of the following steps:

1. Gauss-Seidel relaxation of the per-vertex equations (Equation 3.5) (n_1 steps).

2. Computation of the residual $\boldsymbol{r}^h_{\boldsymbol{x}}$:

$$\boldsymbol{r}^h_{\boldsymbol{x}} = \boldsymbol{b}^h_{\boldsymbol{x}} - \sum_{\boldsymbol{i}=-\mathbf{1}}^{\mathbf{1}} {}^h\boldsymbol{A}^{\boldsymbol{x}}_{\boldsymbol{i}}\, \boldsymbol{u}^h_{\boldsymbol{x}+\boldsymbol{i}}. \tag{3.8}$$

3. Restriction of the residual to the next coarser grid:

$$\boldsymbol{r}_{\boldsymbol{x}}^{2h} = \sum_{\boldsymbol{i}=-\boldsymbol{1}}^{\boldsymbol{1}} w_{\boldsymbol{i}}\, \boldsymbol{r}_{2\boldsymbol{x}+\boldsymbol{i}}^{h}. \tag{3.9}$$

4. On the next coarser grid, the error $\boldsymbol{e}_{\boldsymbol{x}}^{2h}$ corresponding to the residual is determined by the residual equation

$$\sum_{\boldsymbol{i}=-\boldsymbol{1}}^{\boldsymbol{1}} {}^{2h}\boldsymbol{A}_{\boldsymbol{i}}^{\boldsymbol{x}}\, \boldsymbol{e}_{\boldsymbol{x}+\boldsymbol{i}}^{2h} = \boldsymbol{r}_{\boldsymbol{x}}^{2h}. \tag{3.10}$$

   The residual equation is solved by applying this scheme recursively ($\boldsymbol{e}_{\boldsymbol{x}}^{2h} \equiv \boldsymbol{u}_{\boldsymbol{x}}^{2h}$, $\boldsymbol{r}_{\boldsymbol{x}}^{2h} \equiv \boldsymbol{b}_{\boldsymbol{x}}^{2h}$) or solved directly if the coarsest level is reached.

5. Interpolation of the error back from the next coarser grid:

$$\boldsymbol{e}_{\boldsymbol{x}}^{h} = \sum_{\substack{\boldsymbol{i}=-\boldsymbol{1} \\ \boldsymbol{x}+\boldsymbol{i}\equiv\boldsymbol{0}\ (\mathrm{mod}\ 2)}}^{\boldsymbol{1}} w_{\boldsymbol{i}}\, \boldsymbol{e}_{(\boldsymbol{x}+\boldsymbol{i})/2}^{2h}\ . \tag{3.11}$$

   The condition $\boldsymbol{x} + \boldsymbol{i} \equiv \boldsymbol{0} \pmod 2$ ensures that only coarse grid locations are considered.

6. Coarse grid correction: $\boldsymbol{u}_{\boldsymbol{x}}^{h} \leftarrow \boldsymbol{u}_{\boldsymbol{x}}^{h} + \boldsymbol{e}_{\boldsymbol{x}}^{h}$.

7. Gauss-Seidel relaxation of the per-vertex equations (Equation 3.5) (n_2 steps).

In our implementation, we use $n_1 = 2$ pre-smoothing and $n_2 = 1$ post-smoothing decoupled Gauss-Seidel relaxation steps per V-cycle ($\omega = 1$). On the coarsest level, a conjugate gradient solver is employed.

3.4 CUDA Implementation

The CUDA implementation of the multigrid finite hexahedra method consists of two parts: a preprocess for creating the finite element model, including the multigrid hierarchy, and the real-time simulation of the deformable model. In the preprocess, the finite element model is constructed and packed into several index arrays, which are stored in GPU memory. Then, the finite element model is used for the real-time simulation of the object's deformations due to applied forces.

In the following, we first explain the data structures used to represent the finite element model in GPU memory (Section 3.4.1). We then show how the computations are parallelized and mapped onto the CUDA threading model (Section 3.4.2). The description of our CUDA implementation is completed by presenting a memory layout that enables coalesced memory accesses and thus facilitates using the full memory bandwidth available on the GPU (Section 3.4.3).

3.4.1 Data Structures

The finite element model, including the multigrid hierarchy, is stored on the GPU using an indexed representation, i.e., finite elements and vertices[1] are addressed via indices. These indices are determined by enumerating the finite elements and the vertices in a specific order that will be explained in Section 3.4.3. The indices are counted from 0 and represented as 32-bit integer values. When referencing neighbors, parents, etc., a special index value of -1 is used to specify that an element or vertex is not existing.

For each finite element, we store its incident vertices, yielding an array with eight indices per element. *For each vertex in the multigrid hierarchy,* we store its neighbor vertices, i.e., the vertices in the 3^3 domain of the numerical stencil (array with 27 indices per vertex), the vertices on the next finer level that restrict to the considered vertex (array with 27 indices per vertex), as well as the vertices on the next coarser level which the considered vertex interpolates from (array with eight indices per vertex). Note that only up to 8 of the potential 27 indices in Equation 3.11 are required due to the condition that vertices have to lie on the coarse grid. If less vertices are required, we store -1 to mark invalid indices. *For each vertex on the simulation level,* we additionally store its incident finite elements (array with eight indices per vertex), as well as its initial position in the undeformed state (array with three scalars per vertex). Note that these arrays are read-only, i.e., do not change during runtime.

Furthermore, *for each finite element,* we store the elastic modulus and density (two arrays, each with one scalar per element), and for *each vertex on the simulation level,* we store the external force vector acting at that vertex (array with three scalars per vertex) and whether the vertex is free or fixed (array with one bool per vertex). These three arrays can be written during runtime, for example to interactively change the applied forces or to adapt the stiffness of the finite elements.

The numerical simulation requires further arrays: *For each finite element,* we store a

[1]Note that the term ‘finite element’ only refers to the grid cells on the simulation level (i.e., the finest level) of the multigrid hierarchy, but ‘vertices’ refers to the vertices on *all* levels of the multigrid hierarchy, unless noted otherwise.

rotation matrix according to the corotational strain formulation (array with nine scalars per element). *For each vertex in the multigrid hierarchy,* we allocate memory for the per-vertex equations, i.e., the 3×3-matrix coefficients ${}^{h}\boldsymbol{A}_{i}^{\boldsymbol{x}}$ (array with $27 \cdot 9$ scalars per vertex), the right-hand side vectors $\boldsymbol{b}_{\boldsymbol{x}}^{h}$ (array with three scalars per vertex), the displacement vectors $\boldsymbol{u}_{\boldsymbol{x}}^{h}$ (array with three scalars per vertex), and the residual vectors $\boldsymbol{r}_{\boldsymbol{x}}^{h}$ (array with three scalars per vertex). *For each vertex on the simulation level,* we furthermore store the displacement vectors $\boldsymbol{u}_{\boldsymbol{x}}^{\mathrm{old}}$ and their first and second time derivatives $\dot{\boldsymbol{u}}_{\boldsymbol{x}}^{\mathrm{old}}$, $\ddot{\boldsymbol{u}}_{\boldsymbol{x}}^{\mathrm{old}}$ of the previous time step for Newmark time integration (three arrays, each with three scalars per vertex).

It is worth noting that the index-based representation of the finite element model—in contrast to an index-free representation based on a rectangular domain with implicit neighborhood relationships—has the advantage of requiring significantly less memory. This is due to the fact that the memory overhead induced by the index structures is small compared to the memory that would have to be allocated for the per-vertex equations for void regions outside of the object. Another advantage of the index-based representation is that it yields compact lists of elements and vertices (no void ranges), which greatly simplifies an efficient mapping of the computation onto the CUDA threading model, as shown in the next section. Despite of its slightly more irregular nature, we will show in Section 3.4.3 that the index-based representation nevertheless allows for CUDA-friendly memory layouts and can thus exploit the full bandwidth provided by the GPU.

3.4.2 Parallelization

In the following we discuss how the substeps of one simulation time step are parallelized and mapped onto the CUDA threading model. Note that we are using the corotational strain formulation, which requires to update the simulation level equations as well as the coarse grid equations in every time step to consider the current element rotations.

Computation of the Element Rotations The element rotations are computed by polar decomposition [Hig86] of the elements' average deformation gradients. To parallelize these computations, we assign one CUDA thread to each finite element. Each thread fetches the current displacement vectors at the element's vertices from global GPU memory, determines the average deformation gradient, and iteratively computes its polar decomposition using five iteration steps. The resulting rotation matrix is stored in global memory.

Assembly of Simulation Level Equations For the assembly of the per-vertex equations on the simulation level, we assign one CUDA thread to each vertex. Each thread fetches the indices of the incident elements, and then loads the elements' density and elastic modulus values as well as the elements' current rotations. Furthermore, the thread fetches the external force applied at its vertex, the displacement vector at this vertex and its first and second time derivatives of the previous time step, as well as the initial positions and the fixation status of the vertices in the 3^3 neighborhood. Using the precomputed element matrices of a single, generic element, which are stored in cached constant memory, the thread assembles the per-vertex equation consisting of the 27 3×3-matrix coefficients as well as the right-hand side vector. Trading using an excessive number of registers per thread for an increase of memory traffic, the coefficients are assembled in global memory using read-modify-write operations.

Assembly of Coarse Grid Equations The levels of the multigrid hierarchy are assembled successively with one kernel call per level. We assign 9 CUDA threads to each vertex of the current level. The 9 threads first load the vertex indices of the corresponding 3^3 neighborhood on the previous finer level into shared memory. They then assemble the 27 3×3-matrix coefficients of the per-vertex equation. Each thread computes one of the 9 scalar components of every coefficient. To avoid costly read-modify-write operations to global memory, each thread uses 27 registers to accumulate the coefficients. To ensure lock-step execution, we map each group of 32 vertices onto 9 warps (9×32 threads) such that the i-th thread of each warp is assigned to the i-th vertex.

Gauss-Seidel Relaxation The sequential version of the Gauss-Seidel algorithm traverses the vertices at a particular level and successively relaxes each per-vertex equation. For the relaxation of an equation, the *updated* displacement vectors at the previously visited vertices are used. To parallelize the Gauss-Seidel algorithm, these dependencies have to be considered. We employ the so-called multi-color Gauss-Seidel algorithm, which partitions the set of vertices into multiple subsets such that the vertices within each subset can be relaxed in parallel. The subsets, however, have to be processed sequentially. For the numerical stencil in our application, 8 subsets are required. They are defined by $\{\boldsymbol{x} \mid x_1 \bmod 2 = i_1,\ x_2 \bmod 2 = i_2,\ x_3 \bmod 2 = i_3\}$, $\boldsymbol{i} \in \{0,1\}^3$, i.e., when dividing the domain into blocks of 2^3 vertices, the k-th vertex ($k = 1, \ldots, 8$) of each block belongs to the k-th subset.

To process the subsets sequentially, we issue one CUDA kernel call per subset. To compensate the reduced parallelism, which is especially important for medium-resolution finite element models having only a moderate number of vertices, we assign 13 CUDA threads to each vertex. Each thread computes two summands of the sum in Equation 3.5. For each summand, the thread first fetches the index of the respective neighbor vertex, and then fetches the corresponding displacement vector. The respective 3×3-matrix coefficients are also loaded from global memory. The sum is then computed by a logarithmic reduction operation. The first thread finally computes the new displacement vector and writes it back into global memory. To ensure lock-step execution, we map each group of 32 vertices onto 13 warps (13×32 threads) such that the i-th thread of each warp is assigned to the i-th vertex.

Computation of Residual For the computation of the residual, we assign one CUDA thread to each vertex. The computation is similar to the computation performed in the Gauss-Seidel relaxation step. In contrast, however, the residual computation does not exhibit any data dependencies, thus all vertices can be processed in parallel.

Transfer Operators For the transfer operators, we again assign one CUDA thread to each vertex. For the restriction operator, each thread iterates over the corresponding neighborhood on the next finer level to compute a weighted average of the residual vectors. For each neighbor, the thread first fetches the respective vertex index, and then loads the corresponding residual vector. The weighted average is finally written back into global memory, constituting the right-hand side vector of the per-vertex equation of the thread's vertex. Additionally, the thread initializes the displacement vector at its vertex with $\mathbf{0}$.

The interpolation operator is implemented in a similar way. Each thread iterates over the corresponding neighborhood on the next coarser level and computes a weighted average of the coarse grid correction vectors. This vector is added to the displacement vector at the thread's vertex.

Note that the transfer operators have to be implemented as gathering operations. Scattering would require atomic read-modify-write accesses to global GPU memory, since multiple threads might scatter to the same memory location. Furthermore, read-modify-write operations would increase memory traffic.

Conjugate Gradient Solver on the Coarsest Level Considering that the number of vertices on the coarsest level is too small to fully exploit the parallelism offered by the GPU, and that global synchronization via multiple kernel calls is rather expensive, we run the conjugate gradient solver for the coarsest level on a single multiprocessor. We assign one CUDA thread to each vertex. The maximum number of vertices on the coarsest level is thus limited by the maximum number of threads per thread block (1024 on the Fermi GPU). To obtain an efficient multigrid V-cycle we use as many multigrid levels as are required to reduce the number of vertices on the coarsest level to less than 512.

In the CUDA threading model, threads are organized in larger groups called thread blocks. All threads in a thread block are scheduled on the same multiprocessor and can cooperate via on-chip shared memory. For our implementation, experiments have shown that using a thread block of minimum size (32 elements/vertices per thread block) yields the best performance.

3.4.3 CUDA-friendly Memory Layout

The simulation of deformable objects comes with high memory requirements. Due to the underlying finite element discretization, the numerical stencil of a single vertex consists of 27 coefficients, with each coefficient being a 3×3-matrix. Moreover, the stencil is not constant for all vertices, but it varies due to different material parameters associated with each element and due to the corotational strain formulation. The numerical stencil leads to memory requirements of about 1 KB per vertex using 32-bit single floating point precision (2 KB per vertex using double precision).

A primary goal of our CUDA implementation thus is to effectively exploit the high memory bandwidth available on the GPU. In contrast to CPUs, maintaining data locality is not the main criterion for optimizing memory throughput on the GPU. Due to the specific hardware architecture of CUDA-enabled GPUs, it is mandatory to thoroughly coordinate memory access operations of parallel running threads in such a way that multiple memory accesses can be coalesced into single memory transactions. The Fermi GPU performs memory accesses at a fix granularity of 128 bytes, i.e., the GPU reads and writes entire blocks of 128 bytes that are aligned at 128-byte boundaries. Parallel memory accesses of the threads of a warp (consisting of 32 threads running in lock-step) that lie in the same 128-byte segment are coalesced into one single transaction. To maximize memory throughput, data thus should be organized in such a way that the i-th thread of a warp (half warp) accesses the i-th 32-bit (64-bit) word of a 128-byte segment. It is worth noting that without coalesced memory accesses, the effective memory throughput can decrease down to $1/32$ of the GPU's memory bandwidth (in the case of requiring just a single 32-bit word of a 128-byte segment). Furthermore, since all threads in a warp are blocked until all memory access operations of the warp are finished, considerably higher latencies are introduced.

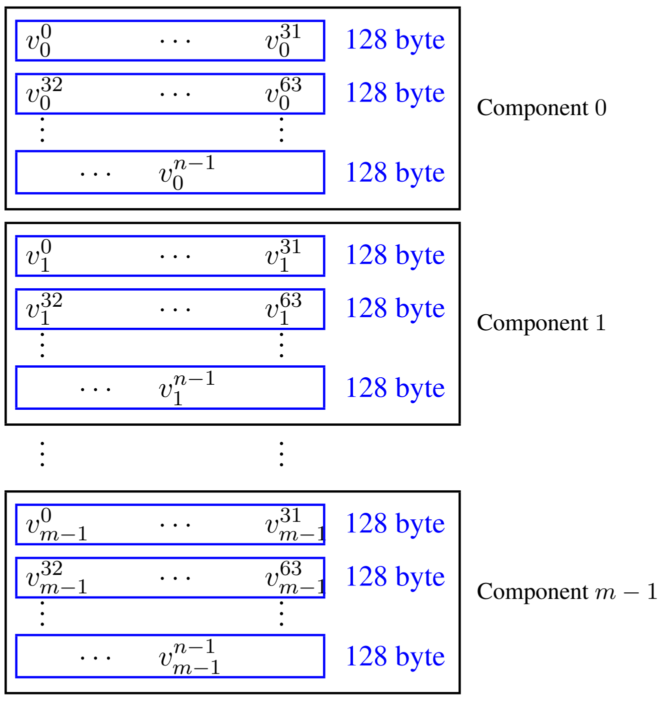


Figure 3.4: *Memory layout for a generic array $\boldsymbol{v}$ with n elements, each consisting of m 32-bit scalar components. v_j^i denotes the j-th component of the i-th element of the array. The j-th components of all elements are stored sequentially in a separate memory block. If the i-th thread accesses the i-th element, the memory accesses can be optimally coalesced into 128-byte memory transactions (blue).*

In the following, we describe how the data used in our application are stored in memory to allow for coalesced memory access operations. The main principle is to store each array of vectors or matrices such that their scalar components are grouped into separate memory blocks, i.e., the j-th components of all vectors/matrices of the array are sequentially stored in the j-th block. For the assignment of indices to the finite elements and vertices, we enumerate the elements as well as the vertices of each subset (for the multi-color Gauss-Seidel algorithm) in lexicographical order according to their 3D integer position (z first, y second, x third), with the vertices being enumerated continuously over all subsets and multigrid levels.

To align the memory accesses of warps at multiples of 128 bytes, the number of vertices per subset are rounded up to multiples of 32, i.e, the index of the first vertex of each subset is a multiple of 32. (The additional dummy vertices are marked as invalid by storing -1 in all index structures corresponding to these vertices.) In Figure 3.4, we illustrate this memory layout for a generic array consisting of n elements with m scalar

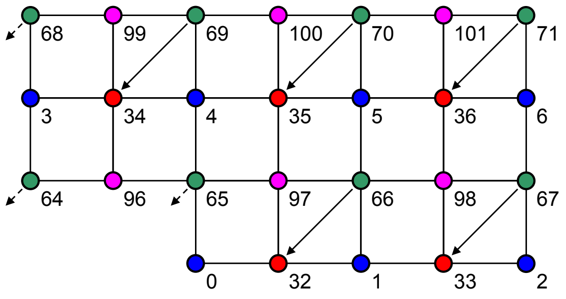


Figure 3.5: *Illustration of the efficiency for accessing data at the neighboring vertices. The colors of the vertices correspond to the subsets used for multi-color Gauss-Seidel relaxation. The numbers denote the indices of the vertices and correspond to the relative location of the per-vertex data in memory. In the example, the green vertices access data stored at their lower-left neighbors, which are all in the same subset (red vertices). Note that if the threads are sequentially assigned to the green vertices, the threads access the data at the neighbor vertices in contiguous memory blocks (except at the object's boundary).*

components per element. For each kernel call, we map each contiguous block of 32 indices onto a warp (or several warps) of 32 threads. If the i-th thread accesses the i-th element of the array, the memory accesses are maximally coalesced and yield optimal memory throughput.

In our application, this setting is always met when a thread assigned to a specific element or vertex accesses data that are specific to that element or thread. However, when a thread accesses data belonging to a neighboring vertex (for example, when accessing the displacement vectors $\boldsymbol{u}$ in the Gauss-Seidel relaxation step), the situation is slightly different. In this case, the threads belonging to a warp still read a contiguous block of memory (except at the object's boundary), as illustrated in Figure 3.5. However, since this block in general is not aligned at a 128-byte boundary, the hardware can only coalesce these memory accesses into two instead of one memory transaction.

3.5 Rendering

Even though the proposed CUDA implementation of elasticity simulation allows using model discretizations at reasonable resolution, a high-resolution render surface is required to achieve a visually continuous representation. To maintain and render such a representation efficiently on the GPU, we use CUDA and the graphics API Direct3D in combination.

We use the triangle surface mesh that is initially used to build the finite element model by voxelization. This surface mesh is stored in GPU memory, represented as

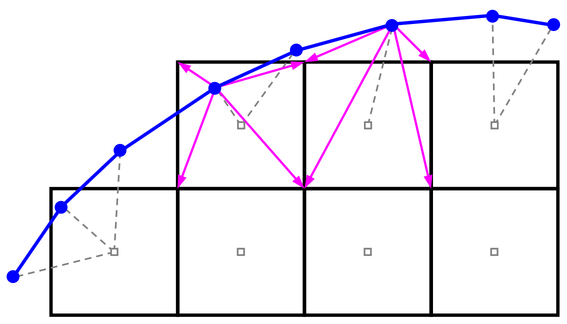


Figure 3.6: *Binding of a high-resolution render surface (blue) to the hexahedral simulation grid. Each render surface vertex is bound to the closest hexahedron with respect to the center of the elements (gray dashed lines). Magenta arrows indicate the element vertices used for trilinear interpolation/extrapolation (for simplicity shown only for two selected vertices).*

an Direct3D index array that contains for every triangle references into a shared vertex array with associated per-vertex attributes. The shared vertex array is stored on the GPU as an Direct3D buffer object. Notably CUDA can directly write into Direct3D resources, thereby avoiding any copying operations.

The vertices of the render surface are bound to the vertices of the simulation grid via interpolation weights as illustrated in Figure 3.6. For every render vertex, we determine the simulation element closest to this vertex—by using the distance between the vertex and the element center—and compute the trilinear interpolation/extrapolation weights of the element vertices. These weights, together with respective references to the simulation vertices, are computed in the preprocess and stored in GPU memory.

At runtime, the displacement vectors at the simulation vertices are computed via CUDA as proposed in the previous Section, and the render surface vertices are updated according to these displacements using the precomputed weights. Note that also the last step is performed via a CUDA compute kernel, which directly updates the Direct3D vertex array. Finally, the render surface is displayed using triangle rasterization.

3.6 Results

We analyze the performance of our CUDA-based multigrid finite element approach for simulating deformable objects using the Stanford bunny model at different resolutions, ranging from 12,000 to 269,000 hexahedral finite elements (see Figure 3.13). All of our experiments were run on a high-end workstation, equipped with two quad core Intel Xeon X5560 processors (based on the Nehalem microarchitecture) running at 2.8 GHz, 48 GB of DDR3 1333 MHz RAM, and an NVIDIA GeForce GTX 480 graph-

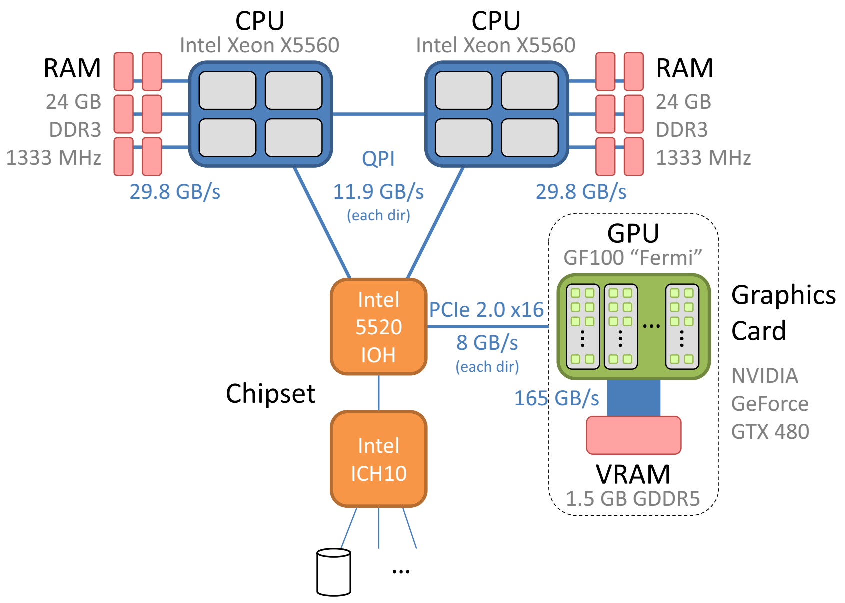


Figure 3.7: *Illustration of the architecture of the workstation used for performance measurements.*

ics card with 1.5 GB of video memory (see Figure 3.7). Each CPU core can issue a SIMD floating point ADD and a SIMD floating point MUL operation simultaneously in every clock cycle (with a SIMD width of 4 for single and 2 for double floating point precision). Each CPU with 4 cores thus exhibits a theoretical peak performance of 89.6 GFLOPS for single and 44.8 GFLOPS for double precision. Respectively half of the RAM is attached to each CPU's internal memory controller via triple channel, yielding an overall theoretical memory bandwidth of 29.8 GB/s per CPU. The two CPUs and the northbridge communicate via point-to-point quick path interconnect (QPI), which provides a theoretical bandwidth of 11.9 GB/s in each direction. Note that our system has a non-uniform memory access (NUMA) architecture, since accesses to a CPU's respective local memory are faster than accesses to its respective remote memory, which have to be performed via quick path interconnect.

The Fermi GPU on the graphics card is equipped with 480 scalar CUDA cores, running at 1401 MHz. Each of these cores is capable of performing one fused multiply-add (FMA) operation per clock at single floating point precision, or one FMA operation every two clocks at double precision. However, on the consumer-level NVIDIA GTX 480 graphics card, only 1/4 of the double precision floating point performance of the Fermi GPU is available. This corresponds to a theoretical peak performance of 1.34 TFLOPS for single and 168 GFLOPS for double precision. The theoretical memory bandwidth between the GPU and the video memory is 165 GB/s.

Model		#MG	**GPU**			CPU, 1 Core			CPU, 2 Cores		
#Hex.	#Vert.	Levels	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s
11,900	14,600	4	120	33.3	45.6	7.92	2.21	3.02	14.9	4.16	5.68
33,300	38,700	4	61.9	44.6	60.5	2.99	2.15	2.92	5.76	4.15	5.63
94,300	105,000	5	27.5	52.7	71.3	1.09	2.09	2.83	2.13	4.07	5.51
269,000	291,000	5	10.8	56.2	75.7	0.396	2.05	2.79	0.77	4.02	5.41

Model		#MG	CPU, 4 Cores			CPU, 8 Cores		
#Hex.	#Vert.	Levels	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s
11,900	14,600	4	26.0	7.24	9.89	41.1	11.5	15.7
33,300	38,700	4	10.2	7.36	9.99	17.2	12.4	16.8
94,300	105,000	5	3.78	7.24	9.80	6.67	12.8	17.3
269,000	291,000	5	1.39	7.24	9.74	2.43	12.7	17.1

Table 3.1: *Simulation performance on the GPU and CPU for different finite element models using* single *floating point precision. For each resolution, we first specify the number of hexahedral elements and the number of vertices (on the simulation level), as well as the number of multigrid levels. We then specify the simulation time steps per second, the sustained rate of floating point operations per second in GFLOPS, and the sustained effective memory throughput in GB/s achieved on the GPU and on the CPU using 1, 2, 4, and 8 cores, respectively.*

Model		#MG	**GPU**			CPU, 1 Core			CPU, 2 Cores		
#Hex.	#Vert.	Levels	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s
11,900	14,600	4	91.9	25.6	67.7	6.93	1.93	5.11	12.8	3.56	9.41
33,300	38,700	4	41.4	29.8	78.3	2.58	1.86	4.88	4.78	3.44	9.03
94,300	105,000	5	17.1	32.7	85.4	0.951	1.82	4.76	1.77	3.40	8.88
269,000	291,000	5	6.54	34.2	88.8	0.345	1.80	4.68	0.64	3.33	8.67

Model		#MG	CPU, 4 Cores			CPU, 8 Cores		
#Hex.	#Vert.	Levels	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s
11,900	14,600	4	20.6	5.73	15.1	31.6	8.81	23.3
33,300	38,700	4	7.70	5.54	14.5	12.9	9.32	24.5
94,300	105,000	5	2.85	5.47	14.3	4.75	9.09	23.8
269,000	291,000	5	1.02	5.34	13.9	1.69	8.81	22.9

Table 3.2: *Simulation performance using* double *floating point precision. The columns are analogous to Table 3.1.*

CUDA Kernel	FLOPs	%	Bytes R/W		%	% GPU Time	
			Single	Double		Single	Double
Computation of element rotations	470	2	160	300	1	0	1
Assembly of sim. level equations	10000	51	6600	13000	23	33	30
Assembly of coarse grid equations	3200	17	5100	10000	18	24	21
Gauss-Seidel relaxation	6×660	20	6×1800	6×3500	39	27	33
Computation of residual	2×620	6	2×1800	2×3500	13	11	12
Restriction of residual	2×210	2	2×580	2×1000	4	1	1
Interpol. of error and coarse grid corr.	2×39	0	2×200	2×350	1	2	1
CG solver on coarsest level	2×3	0	2×1	2×1	0	1	1
Total (per finite element per time step)	19000		28000	54000			

Table 3.3: *Detailed analysis of the costs per finite element per time step (total costs per time step divided by the number of finite elements) for each individual CUDA kernel. The analysis is based on the bunny model with 269,000 elements. From left to right, the columns contain the kernel, the number of FLOPs, the respective percentage of the total number of FLOPs, the number of bytes read and written for single and double floating point precision, the respective percentage of the total number of bytes read and written, and finally the measured percentage of GPU time spent for each individual kernel using single and double precision, respectively. The factors 6× and 2× correspond to performing 2 V-cycles per time step, each with 2 pre- and 1 post-smoothing Gauss-Seidel steps.*

Tables 3.1 and 3.2 show the performance of our implementation using single and double floating point precision, respectively. The construction of the simulation model is performed in a preprocess on the CPU. Each time step includes the computation of the element rotations, the assembly of the per-vertex equations on the simulation level and on the coarse grids of the multigrid hierarchy, as well as two multigrid V-cycles, each with two pre-smoothing and one post-smoothing Gauss-Seidel relaxation steps. The sustained rate of floating point operations performed per second in GFLOPS as well as the sustained memory throughput in GB/s are obtained by manually counting the number of floating point operations performed by each kernel, as well as the number of bytes read and written by each kernel (see Table 3.3). The statistics thus report the *effective* memory throughput, i.e., the transfer of unnecessary data due to the fix memory transaction size of 128 bytes and cache hits are not considered. Note, however, that due to our optimized memory layout that facilitates coalescing of memory accesses and due to the fact that almost all data—the coefficients of the per-vertex equations—are not accessed repeatedly in a kernel, the *effective* memory throughput should be close to the *physical* memory throughput. On the GPU using single floating point precision, we achieve 120 time steps per second for the 12,000 element model and 11 time steps per second for the 269,000 element model. For double precision, the update rates are 92 and 6.5 time steps per second, respectively.

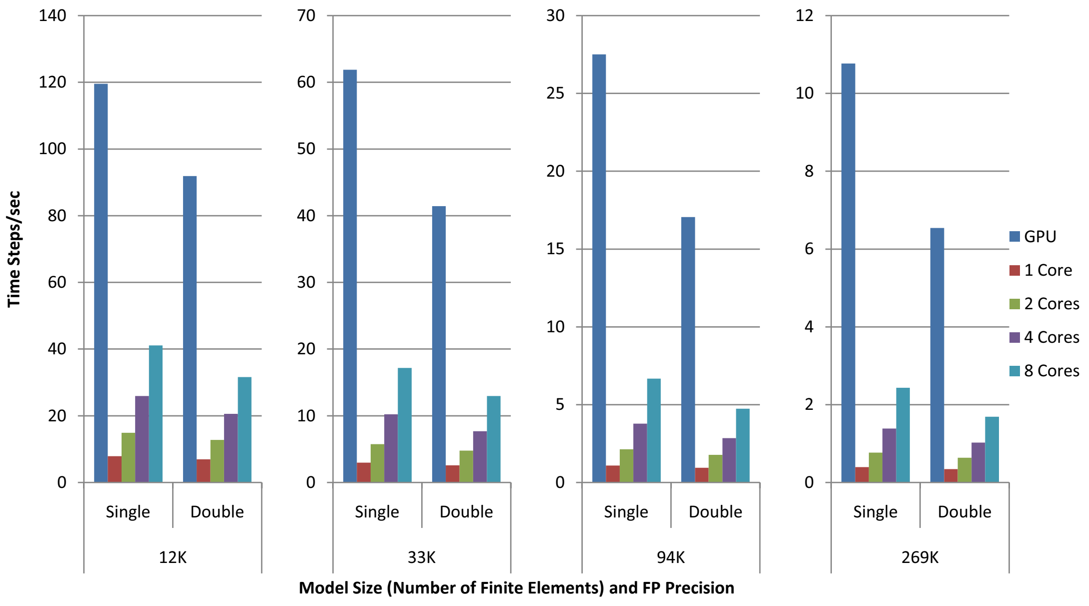


Figure 3.8: *Simulation time steps per second achieved for different model sizes as well as single and double floating point precision. Each group of five columns shows the performance obtained on the GPU and on the CPU using 1, 2, 4, and 8 cores. Each time step includes the re-assembly of the system of equations as well as two multigrid V-cycles.*

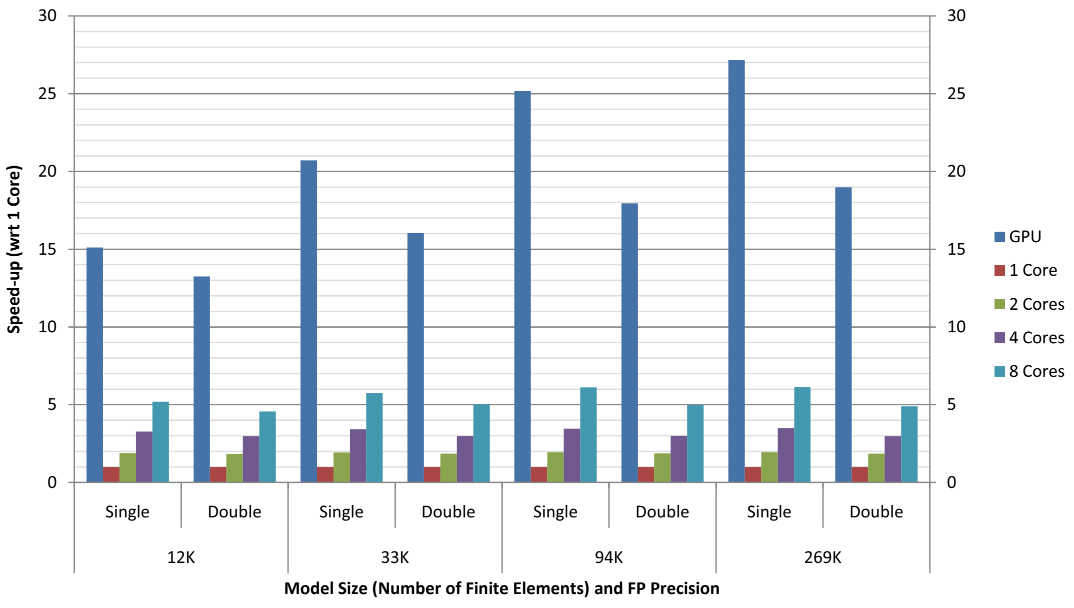


Figure 3.9: *Speed-ups achieved by parallelization on the GPU and on the CPU. All speed-ups are calculated with respect to a single CPU core.*

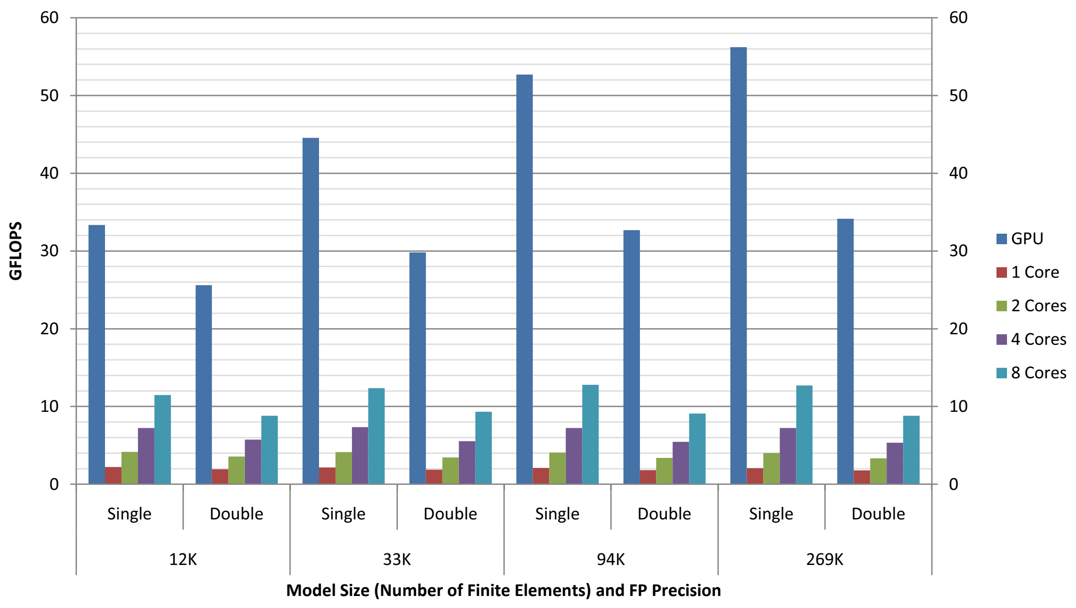


Figure 3.10: *Floating point performance (in GFLOPS) achieved on the GPU and on the CPU.*

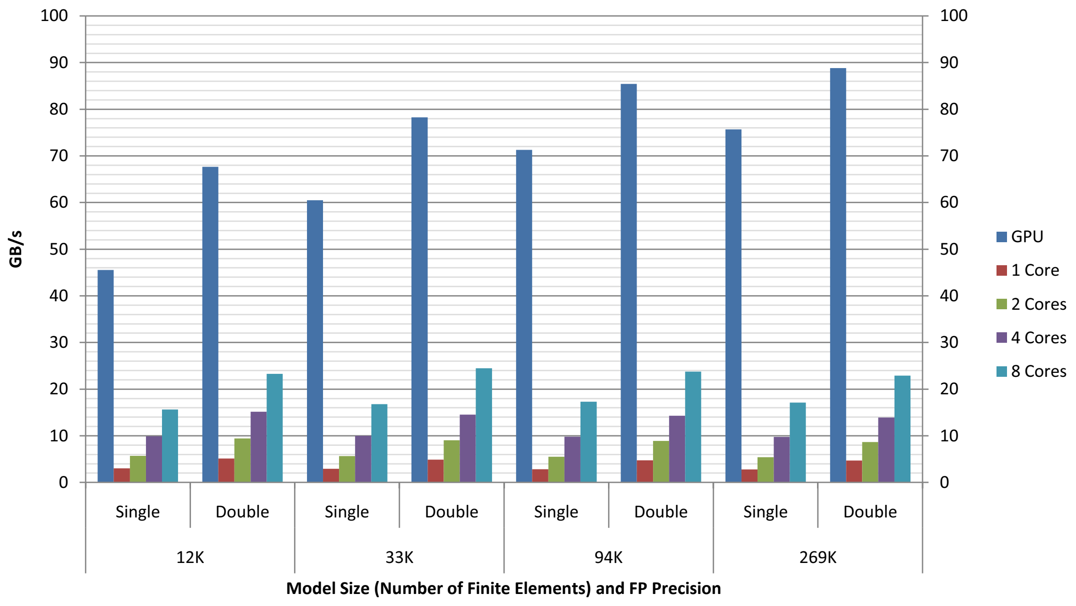


Figure 3.11: *Memory throughput (in GB/s) achieved on the GPU and on the CPU.*

Updating and rendering a high-resolution render surface (70,000 triangles) that was bound to the simulation grid took less than 6.5 ms in all examples. This time is not included in the timings in Tables 3.1 and 3.2.

In order to analyze the performance gain that is achieved by our GPU-based implementation, we compare it to an optimized CPU-based implementation that has been parallelized using OpenMP. The CPU implementation is very similar to the GPU implementation in that we use the same data structures and algorithms. The differences are as follows: On the CPU, we use one thread per core (with a fix assignment of threads to cores), and each thread processes a block of elements or vertices—in contrast to the GPU, where we use one or several threads per element or vertex. Corresponding to the different mapping of the work onto threads, we use different memory layouts on the CPU and GPU. On the CPU, we store the scalar components of each element of an array directly one after another, whereas on the GPU the elements are stored interleaved by grouping corresponding scalar components of the array elements into separate memory blocks, as described in Section 3.4.3.

To achieve optimal performance on our NUMA CPU target architecture operated under Windows 7, we store the thread-local per-element and per-vertex data in the respective CPU's *local* memory. The operating system's default strategy for assigning a memory page to a memory frame is to choose a page from the local memory of that CPU which performs the first read/write access to the frame ('first touch' strategy). Thus, by initializing the respective memory addresses *from within the respective thread* directly after virtual memory allocation, the intended memory assignment is obtained.

We compare the GPU-based implementation to the CPU-based implementation running on 1, 2, 4, and 8 CPU cores, respectively. When using 1, 2, and 4 cores, we use cores belonging to the same CPU. The time steps per second, GFLOPS, and memory throughputs are listed in Tables 3.1 and 3.2. Figure 3.8 shows the simulation time steps per second on the GPU and on the CPU. The respective speed-up factors, measured with respect to a single CPU core, are given in Figure 3.9.

The diagram shows that the speed-up on the GPU increases with the model resolution due to a better utilization of the GPU's massively parallel architecture with increasing model size. For the largest model consisting of 269,000 finite elements, with respect to a single CPU core we achieve a speed-up of 27 for single and of 19 for double floating point precision. Even with respect to 8 CPU cores, the GPU is still a factor of about 4 faster for single and double precision. Figures 3.10 and 3.11 show the floating point performance (in GFLOPS) and the memory throughput (in GB/s), respectively.

For the largest model we achieve 56 GFLOPS (single precision) and 34 GFLOPS

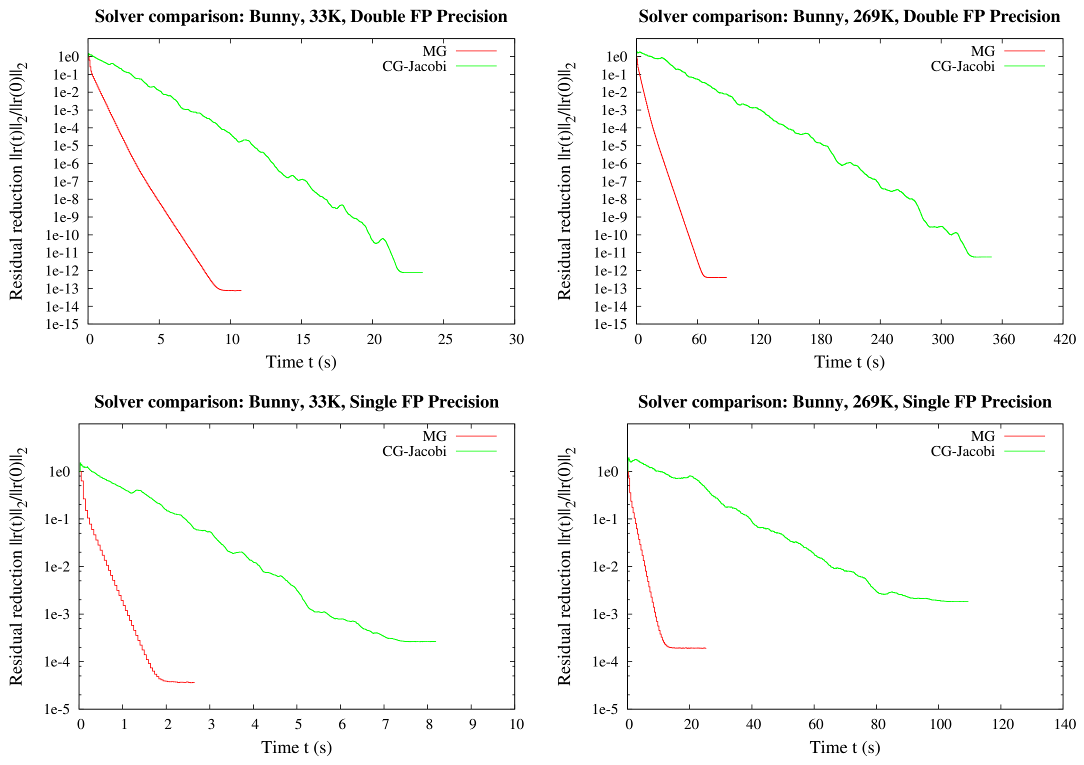


Figure 3.12: *Convergence of our multigrid solver (red) and a conjugate gradient solver with Jacobi preconditioner (green) with respect to computing time on a* single CPU core, *using two different model sizes as well as single and double floating point precision. The ordinate shows the reduction of the norm of the residual with respect to the start of the solver. ($E = 10^6\,\mathrm{Pa}$, $\nu = 0.3$, $\rho = 10^5\,\mathrm{kg/m^3}$, $dt = 50\,\mathrm{ms}$, edge length of hexahedral elements is 2.8 mm and 1.4 mm, respectively)*

(double precision) on the GPU. Since these values are clearly below the theoretical arithmetic throughput of the GPU, we assume that our GPU implementation is memory-bound. This is confirmed by the statistics on memory throughput, which report sustained rates of 76 GB/s for single and 89 GB/s for double precision, which is about half of the theoretical memory bandwidth. The decrease of performance when switching from single to double precision thus results from the doubled memory size of double precision values compared to single precision values. On the CPU using 1 core, we achieve about 2 GFLOPS for both single and double precision. 2 cores almost yield twice the performance of 1 core. Since the two cores are on the same CPU and thus use the same memory connection, this indicates that our CPU implementation is compute-bound on a single core. The statistics further show a better scalability in the number of cores for single precision than for double precision, and further report an increasing im-

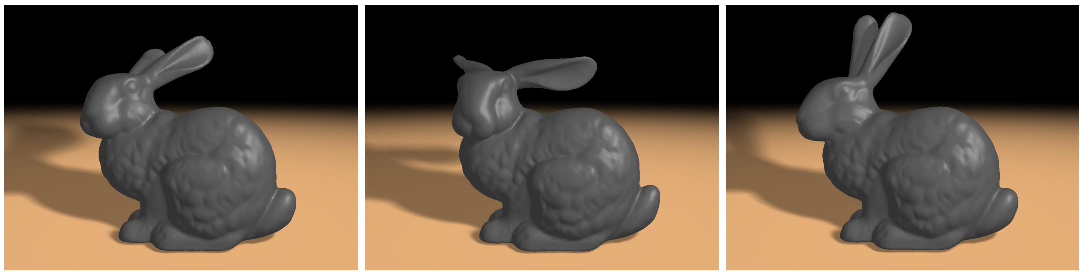


Figure 3.13: *Deformations of the Stanford bunny model (94,300 elements). The left image shows the model in the undeformed state. Simulation runs at 28 (17) time steps per second using single (double) floating point precision.*

pact on performance when switching from single to double precision with an increasing number of cores. This indicates that for double precision the CPU implementation is becoming memory-bound as more and more CPU cores are used. For the 269K element model, the CPU implementation achieves speed-ups of 1.76 and 1.65 using single and double precision, respectively, when going from 1 to 2 CPUs (4 to 8 cores).

In the future, we will also investigate the parallelization of our implementation on multiple GPUs. Since on current architectures GPU-to-GPU communication has to be initiated by the CPU and performed via PCI Express, we expect the scalability to be limited by the high latencies that are introduced.

In summary, the speed-ups achieved by the GPU compared to the CPU result from *both* the higher floating point performance and memory bandwidth on the GPU. Since for our application the limiting factor on the GPU is memory throughput, we expect the performance on future GPU architectures to be strongly related to the available memory bandwidth.

Finally, we also analyze the convergence behavior of our multigrid solver to demonstrate its suitability for the simulation on complicated domains. Figure 3.12 shows the reduction of the norm of the residual achieved by the multigrid solver (red curves) with respect to computing time. The convergence behavior of a conjugate gradient (CG) solver with Jacobi preconditioner (green curves) is shown for comparison. Note that the timings were performed on the CPU (using 1 core) to allow for a precise measurement of the individual solver cycles. As can be seen, the multigrid solver exhibits a constant rate of residual reduction over time, which is characteristic for this solver type. Furthermore, the multigrid solver is significantly more efficient than a CG solver, in that in a given time budget it reduces the residual by a much higher factor than the CG solver. Note that we simulate the dynamics of deformable objects, thus the solution from the previous time step provides a very good initial guess to start with in the current time step. Due to this, we achieve high simulation accuracy using only 2 V-cycles per time step.

3.7 Conclusion and Future Work

In this chapter, we have presented a real-time method for physics-based elasticity simulation using CUDA on NVIDIA's Fermi GPU. The method employs the power of numerical multigrid schemes for the efficient solution of the governing system of partial differential equations based on a finite element discretization. Underlying our approach is a hexahedral discretization of the simulation domain, giving rise to efficient algorithms for model construction and parallel simulation. By introducing a discretization-specific restructuring on the algorithmic level, the multigrid simulation scheme can efficiently be mapped onto the GPU via the CUDA computing API. In this way, significant performance gains can be achieved compared to our optimized parallel CPU implementation.

Our method also opens a number of areas for future research. Since our application is memory-bound, a mixed floating point precision approach as proposed in [CBB$^+$10] might alleviate memory bandwidth limitations and further increase simulation performance. Another interesting question is how to parallelize the method on GPU clusters. Parallelization strategies similar to the one proposed in [SB10] will be considered, with the focus on minimizing inter-GPU communication. An additional challenge is to integrate GPU-based collision detection and handling in real-time deformable body simulations. Image-based techniques as proposed in [GKW07] will be investigated for this purpose.

Chapter 4

A Hexahedral Multigrid Approach for Simulating Cuts in Deformable Objects

In this chapter, we present a hexahedral finite element method for simulating cuts in linear elastic bodies using the corotational formulation of strain at high computational efficiency. Key to our approach is a novel embedding of adaptive element refinements and topological changes of the simulation grid into a geometric multigrid solver. Starting with a coarse hexahedral simulation grid, this grid is adaptively refined at the surface of a cutting tool until a finest resolution level, and the cut is modeled by separating elements along the cell faces at this level. To represent the induced discontinuities on successive multigrid levels, the affected coarse grid cells are duplicated and the resulting connectivity components are distributed to either side of the cut. Drawing upon recent work on octree and multigrid schemes for the numerical solution of partial differential equations, we develop efficient algorithms for updating the systems of equations of the adaptive finite element discretization and the multigrid hierarchy. To construct a surface that accurately aligns with the cuts, we adapt the splitting cubes algorithm to the specific linked voxel representation of the simulation domain we use. The chapter is completed by a convergence analysis of the finite element solver and a performance comparison to alternative numerical solution methods. These investigations show that our approach offers high computational efficiency and physical accuracy, and that it

This chapter is based on material that has been originally published in C. Dick, J. Georgii, and R. Westermann, *A hexahedral multigrid approach for simulating cuts in deformable objects*, IEEE Transactions on Visualization and Computer Graphics 17 (2011), no. 11, 1663–1675 [DGW11a]. The copyright for this material is owned by the IEEE.

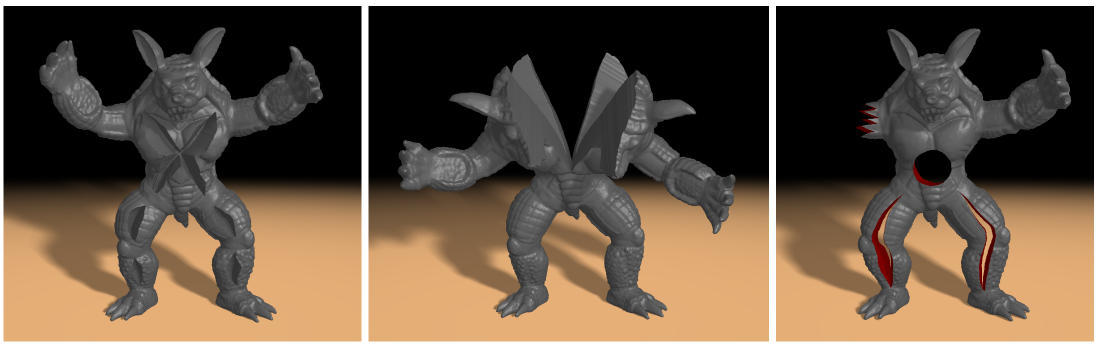


Figure 4.1: *Cuts in the Stanford Armadillo model. An adaptive finite hexahedra model consisting of 493K simulation elements is used. Adaptive refinements of the simulation mesh along the cuts result in 76K, 150K, and 153K additional elements, from left to right. Cutting and simulation is performed at 9 to 11 seconds per time step.*

enables cutting of deformable bodies at very high resolutions.

4.1 Introduction

In this chapter, we propose a method for realistically simulating complicated cuts in linear elastic deformable objects. Our approach is different from previous approaches in that it does not treat the cutting procedure and the numerical solution scheme independently, but intertwines both procedures in a way that enables high computational efficiency. We achieve this by developing a novel embedding of adaptive finite element refinements and topological changes of the simulation grid into a geometric multigrid method [Bra77, Hac85, BHM00]. Adaptivity enables representing complicated cuts at very high resolution, and the multigrid method achieves optimal computational complexity that is linear in the number of simulation elements. Figure 4.1 shows some cuts that have been performed using our approach.

Underlying the basic multigrid idea is the principle of coupling multiple scales, for instance, by using a geometric model hierarchy equipped with transfer operators to propagate quantities across the scales. The use of such a hierarchy, in general, imposes performance limitations when using multigrid schemes in combination with cutting schemes based on tetrahedral [BMG99, BG00, MK00] or polyhedral [WBG07] finite elements. Since element subdivision generates unstructured meshes in general, there are no canonical coarse versions of the mesh and the construction of a geometric model hierarchy becomes very complicated. Due to this reason, incorporating subdivision based cutting into geometric multigrid schemes has not yet been considered.

Instead of explicitly modeling the boundary induced by a cut in the finite element discretization, this boundary can also be incorporated into the basis functions of the finite dimensional solution spaces [BM97, SCB01]. This enables using a coarse simulation grid that does not depend on the shape of the object. The same principle is underlying the extended finite element method (XFEM) [BB99], which enriches the finite element spaces by employing additional step functions to represent material discontinuities. XFEM has mainly been used to accurately simulate material interfaces and crack propagation [MDB99, SMMB00], and just recently its potential for cutting and fracturing deformable objects in graphics applications has been recognized [AH08, JK09, KMB$^+$09].

Since the XFEM method uses a static simulation grid for which a hierarchy can be constructed in a preprocess, its embedding into a multigrid solver is possible in principle. However, the modeling of high resolution cuts, for instance via enrichment textures [KMB$^+$09], requires large systems of equations to be solved, and it comes at the expense of increasing the computational cost of element integration.

4.2 Contribution

We propose a novel algorithm for physics-based cutting of linear elastic deformable bodies using hexahedral finite elements. Simulation elements that are touched by the cutting tool are recursively subdivided using a regular octree refinement. This results in an adaptive finite hexahedron approximation of the cut object. An example is given in Figure 4.2.

The octree is refined until a sufficient approximation is reached, and on this subdivision level cutting is performed along the element faces. The adaptive refinement allows arbitrarily thin and complicated structures to be sliced, and it can be employed to adapt the octree to material jumps in heterogeneous materials. Examples demonstrating these possibilities are shown in Figure 4.13. Since all elements have the same shape the method only requires scaled instances of the precomputed element matrices of one generic element to accommodate whatever deformation is applied. This eliminates the need for element integration at runtime and significantly reduces storage requirements.

The deformable object has to be discretized on the adaptive octree that is generated by the cutting algorithm. Since none of the previous multigrid approaches considers cuts of the simulation grid, we propose a novel algorithm to embed the induced discontinuities in the geometric multigrid hierarchy, including fast algorithms for updating the resulting systems of equations. On the coarse resolution levels the algorithm dupli-

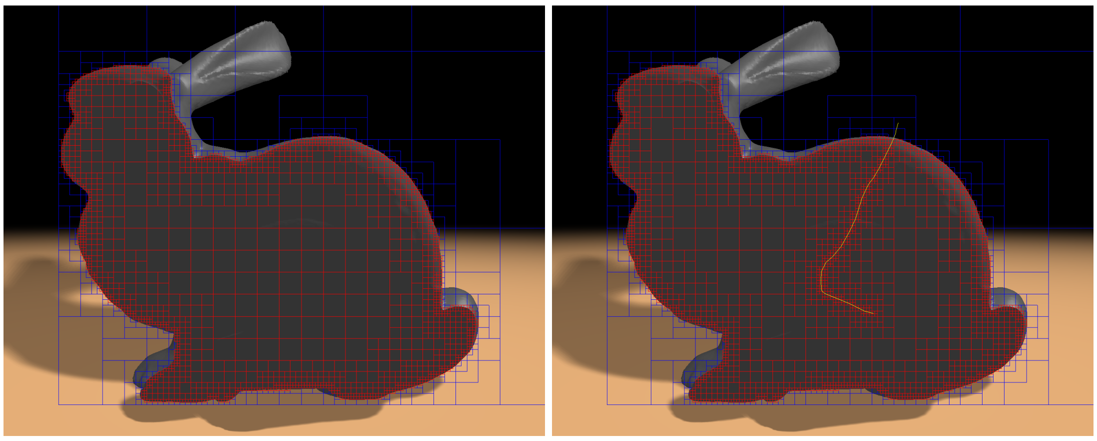


Figure 4.2: *Cross-section through an adaptive octree grid without (left) and with (right) a cut.*

cates respective cells and distributes the per-cell equations to the duplicates according to the separated material components. We present a detailed convergence analysis of our solver and a thorough comparison to alternative solution methods.

Rendering a smooth polygon surface that aligns with the cuts is very difficult since it can undergo complicated topological changes. In particular, a render surface that is bound to the initial simulation grid as proposed in [DDBC99, MG04, GW05, BPWG07] cannot easily be employed for this purpose. Due to this reason we use the splitting cubes algorithm [PGCS09] on a dual grid to compute a watertight boundary surface directly from the 3D simulation grid. By computing separate per-vertex normals for each individual triangle, high quality rendering is achieved.

To clearly focus our work on the efficient cutting and simulation of a finite element model, collision detection and response is not considered. Please note that this is *not* a limitation of the presented approach. In fact, any state of the art collision handling algorithm that uses surface meshes for collision detection and external forces for collision response could be directly integrated.

The remainder of this chapter is as follows: In the next section we review work that is related to ours. Then we describe the cutting algorithm from a purely geometric perspective. Section 4.5 discusses the embedding of adaptive finite element refinements and topological changes of the simulation grid into the multigrid scheme. Section 4.6 presents the specific adaptations to the splitting cubes algorithm to reconstruct an accurate boundary surface from the simulation grid. A detailed analysis of the proposed algorithm, both with respect to performance and convergence is given in Section 4.7. We conclude the chapter with a comparison to alternative approaches and some ideas on future challenges in the field.

4.3 Related Work

Starting with the seminal work of Terzopoulos and co-workers [TPBF87, TF88], physics-based methods for simulating deformable models have been researched extensively in computer graphics for the last two decades. A good overview of the multitude of methods for realistically simulating deformable bodies can be found in [NMK$^+$06]. For example, boundary element models [JP99], adaptive and multiresolution approaches [DDCB01, CGC$^+$02, GKS02], grid-less techniques [MHTG05, SSIF07], and finite element methods [BNC96, WDGT01] have been proposed. The simulation of brittle fracture based on finite elements was described by O'Brien and Hodgins [OH99] and later extended to ductile fracture [OBH02]. [NKJF09] proposed a composite element formulation that considers varying material properties within a coarse element.

Tetrahedral subdivision methods for cutting deformable objects were introduced in [BMG99, MK00]. To reduce the number of ill-shaped elements, Nienhuys and van der Stappen proposed cutting along the element faces [NvdS00]. Cotin et al. and Forest et al. deleted elements that were cut [CDA00, FDA02]. Smooth cuts that also reduce the number of ill-shaped elements were achieved by adaptively aligning mesh edges and faces with the cutting surface [NvdS01, SHS01, SOG06]. By restricting subdivisions to a few refinement patterns [BG00, BGTG03] the number of additional simulation elements caused by a cut can be reduced. A multi-resolution approach for this method was presented by Ganovelli et al. [GCMS00]. The virtual node algorithm [MBF04] avoids ill-shaped elements by duplicating simulation elements and re-assigning material components on both sides of a cut.

Wicke et al. and Kaufmann et al. introduced polyhedral subdivision [WBG07, MKB$^+$08], which splits initial tetrahedra into polyhedra and then subdivides these elements further. Extended finite element methods [BB99] enrich a finite element model with specific basis functions to capture discontinuities in the simulation elements. The use of XFEM for virtual surgery simulation [AH08, JK09] and cutting in 2D thin shells [KMB$^+$09] has been demonstrated. Sifakis et al. clipped a high-resolution material boundary surface mesh against a coarse simulation mesh to consider fine material components in a coarse elasticity simulation [SDF07].

Octree-based physical simulation of fluids and gases was shown in [Pop03, SY04, LGF04]. Both restricted and unrestricted octrees were used. To achieve high resolution of small scale details, one focus was on deriving adaptive finite difference discretizations of the governing equations. Finite element discretizations for the numerical solution of partial differential equations on restricted octrees were introduced in

[HH07, SB10].

Multigrid approaches have recently gained much attention in the computer graphics community due to their optimal computational complexity. The applications range from fluid simulation [BFGS03] and deformable body simulation [WT04, GW06, SYBF06] to image processing [KH08] and texture synthesis [JCM07]. Interactive multigrid approaches for simulating linear elastic materials on hexahedral grids were presented in [DGBW08, ZSTB10]. Since it is well known that the multigrid convergence is lowered in case of complicated material boundaries, [ZSTB10] proposed a numerical boundary smoother for finite difference schemes. An adaptation of the basis functions on the coarser levels to more accurately represent the covered boundary was proposed in [SW06, PRS07, LPR$^+$09].

4.4 Cutting Algorithm

To enable the efficient embedding of the cutting algorithm into a geometric multigrid scheme, we avoid any unstructured grid refinement and instead cut along the element faces in a hexahedral simulation grid that adaptively refines along the cut. This results in an adaptive octree grid. The octree's leaf cells represent the simulation elements and store the corresponding vertices. At the cells on the finest level we store links that are marked as connected or disconnected depending on whether the corresponding elements have been detached by the cut. The links that are cut by the object's boundary are also marked as disconnected. Each octree cell is equipped with memory references to its child cells, its parent cell, and the neighboring cells on the same level.

For the sake of simplicity, we first describe the cutting algorithm in a uniform grid before we introduce the extensions that are necessary to perform a cut in an adaptive octree grid.

4.4.1 Cuts—Uniform Grid

In the following we assume that the object to be cut has been discretized into a uniform hexahedral (voxel) grid. Discretization means building a binary representation, where every voxel is classified as inside or outside depending on whether its center is in the interior of the object or not. Inside voxels represent the finite elements that are used in the physics-based simulation.

We leverage a linked volume representation [FG99] in which the centers of face adjacent elements are connected via links. Initially, each 3D element has six links, and

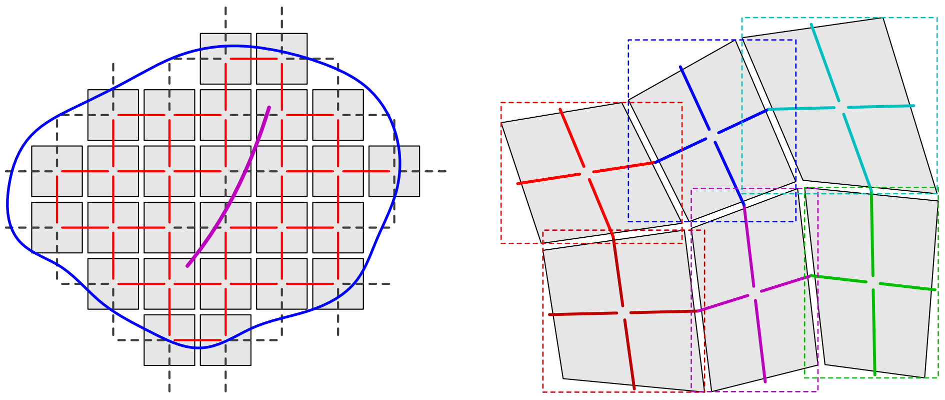


Figure 4.3: *Left: 2D illustration of the linked volume representation, consisting of a set of finite elements (gray) that are connected via links (red). These links are disconnected (dashed, gray) at the object boundary (blue) and when cut by a cutting surface (violet). Right: Bounding boxes (dashed) of deformed finite elements used to accelerate the link/blade intersection test. The links are deformed according to the (trilinear) deformation of the elements. Each element's bounding box covers half of each link emanating from this element.*

the links that are intersected by the surface of the object are marked as disconnected. Cutting the FE-model is performed by disconnecting the links that are cut by the cutting blade (see Figure 4.3 (left) for an illustration).

The blade is realized as a triangle mesh, which is built from consecutive cut-lines that are induced via a cutting tool. To find the links that are cut, all links in the vicinity of the cutting front, i.e., the surface spanned by the current and the last cut-line, are tested for an intersection with the triangles representing this front.

Since the intersection test has to be performed in the deformed object state, in general the cutting blade needs to be tested against all connected links. To prune as many links as possible before testing against the blade, the blade's axis-aligned bounding box is tested first. In addition, every finite element stores its axis-aligned bounding box (see Figure 4.3 (right)), and a bounding box hierarchy is created to prune needless tests. By using this information the broad phase intersection test reduces to simple bounding box tests, and only a few link-triangle tests are required in the narrow phase.

4.4.2 Cuts—Octree Grid

In order to avoid using a uniform finite element representation at the finest resolution level, we employ an adaptive object discretization, i.e., an octree grid, where the octree's leaf cells represent the finite elements. An adaptive representation allows using the fine level simulation elements at the locations where they are needed to resolve the

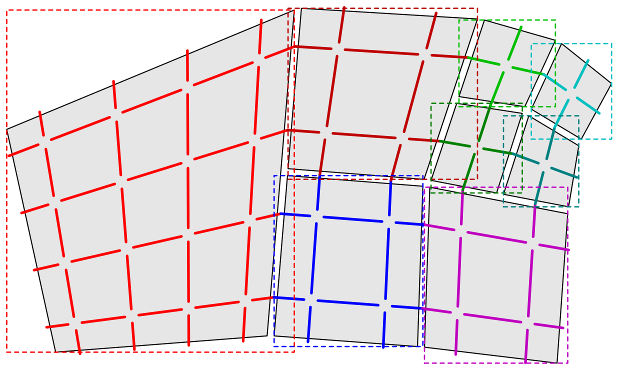


Figure 4.4: *Illustration of the bounding boxes (dashed) of the octree finite elements (gray) used in the link/blade intersection test.*

object boundary accurately. Thus, it can reduce the number of simulation elements significantly, and therefore improves the performance of the finite element simulation.

The octree grid is built from an initial uniform object discretization in a coarse-to-fine procedure. The resolution of this discretization is chosen such that it can adequately model the physical behavior of the object's inner part. In an interactive application, it can be set to a resolution that allows simulating the deforming body at reasonable speed.

Starting with this discretization, at each level the cells containing at least one link that would be cut by the object boundary are refined regularly into 2^3 smaller cells. Finally, all cells that do not contain at least one finest level cell center that is in the object's interior are deleted from the octree structure. Note that the cell centers at the finest level, and thus also the link positions, can be computed without that a cell has to be refined explicitly down to the finest resolution.

The octree refines adaptively along the object's boundary while it models the object's interior away from the boundary at the selected coarser resolution. It is restricted to not have level differences between adjacent elements—sharing a vertex, an edge, or a face—of more than one (see Figure 4.2). In this way abrupt changes in the structural material representation are avoided.

Cutting is performed by traversing the octree and performing a regular 1:8 split of the leaf cells to be refined. The refinement criterion is the same as is used to initially refine the octree grid along the object boundary. On the finest level no split is performed but the links that are cut are disconnected. To accelerate the intersection test between the blade and the elements, each element stores its axis-aligned bounding box. Figure 4.4 illustrates the bounding boxes for adjacent elements at different resolution levels.

The refinement procedure creates additional elements, whose number increases proportional to the object's surface increase. Upon refining an element, the octree is updated to ensure only level transitions of at most one. The bounding boxes of new elements are computed on-the-fly.

4.5 Discontinuous Multigrid Solver

Our simulation approach is based on linear elasticity, combined with the corotational formulation of strain to accurately simulate deformations with large rotations. The governing system of partial differential equations is discretized by using a hexahedral finite element discretization based on the adaptive octree grid described in the previous section and the implicit Newmark time integration scheme, leading to a linear system of equations $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$ in each time step. For details, we refer the reader to Chapter 2.

To solve this linear system of equations, we have developed a computationally efficient geometric multigrid solver. This solver is based on a novel strategy to incorporate complex topology changes induced by cuts into the coarse grid hierarchy in order to avoid a severe decrease of the solver's convergence rate when cuts are introduced. For an introduction to geometric multigrid solvers, we refer the reader to Section 2.3.

We use trilinear interpolation operators $\boldsymbol{I}_{2h}^{h}$, and the restriction operators $\boldsymbol{R}_{h}^{2h}$ and coarse grid operators $\boldsymbol{A}^{2h}$ are chosen according to the variational properties of multigrid, i.e., $\boldsymbol{R}_{h}^{2h} = \left(\boldsymbol{I}_{2h}^{h}\right)^{\mathrm{T}}$ and $\boldsymbol{A}^{2h} = \boldsymbol{R}_{h}^{2h}\,\boldsymbol{A}^{h}\,\boldsymbol{I}_{2h}^{h}$. We employ V-cycles with 1 pre- and 1 post-smoothing decoupled Gauss-Seidel relaxation steps ($\omega = 1.7$). On the coarsest level, a Cholesky solver [TCRM03] is used. Our implementation is based on a matrix-free formulation of all multigrid components (see Chapter 3).

4.5.1 Hierarchy Construction—Uniform Grid

The challenge in developing a multigrid approach that supports topological changes of the simulation grid lies in constructing the coarse grid hierarchy. Since cuts on the finest level also have to be modeled on the coarse levels, cells cannot simply be merged based on their spatial relationship. This would correspond to merging physically and mechanically disconnected parts, and it would result in low approximation quality on the coarse grids and very slow convergence of the solver. In the following we describe the novel principle underlying our hierarchy construction. For the sake of clarity we first restrict the discussion to a uniform hexahedral grid with cuts.

The common approach to build a hexahedral multigrid hierarchy is to merge blocks of 2^3 cells into one coarse grid cell. Note that cells are allowed to be only partially

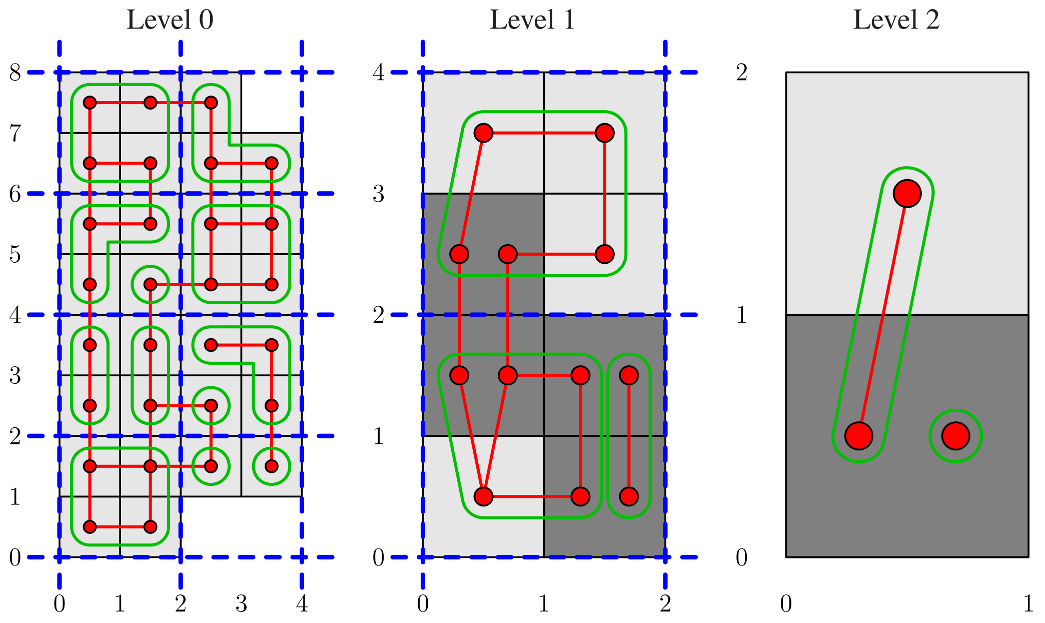


Figure 4.5: *Construction of the coarse multigrid levels based on the undirected graph representation (red) in a uniform grid. The blocks of double cell size, in which connected components (green) are searched, are marked in blue. Duplicated cells are indicated in dark gray. The numbers denote integer coordinates in the underlying lattice.*

filled with cells at the previous level, for example at the object's boundary. To respect physically disconnected parts of the simulation domain, we perform a similar merging strategy, but we explicitly consider the connectivity between the cells to possibly generate more than one coarse grid cell at the same position. A similar strategy has been pursued by Aftosmis et al. [ABA00] to handle complex embedded object boundaries in a multigrid solver for computational fluid dynamics.

The basic idea underlying our construction is to interpret each grid of the hierarchy as an undirected graph $(\mathcal{C}^\ell, \mathcal{E}^\ell)$. The level number $\ell = 0$ denotes the finest level of the hierarchy, and ascending numbers denote successively coarser levels. The nodes $\mathcal{C}^\ell$ of each graph represent the cells of the respective grid, and the edges $\mathcal{E}^\ell \subseteq \big\{\{c_1, c_2\} \mid c_1, c_2 \in \mathcal{C}^\ell, c_1 \neq c_2\big\}$ describe the connectivity between face-adjacent cells. Connected cells share their vertices. In the following, a cell $c \in \mathcal{C}^\ell$ with domain $[x_c, x_c + 1] \times [y_c, y_c + 1] \times [z_c, z_c + 1]$ in the underlying lattice (with a spacing of 2^ℓ relative to the spacing of the finest level) is associated with position (x_c, y_c, z_c).

At the finest level, the nodes $\mathcal{C}^0$ correspond to the finite elements, and the edges $\mathcal{E}^0$ correspond to the links introduced in Section 4.4.1. Starting on the finest level and proceeding to the second coarsest level, the following two steps are performed

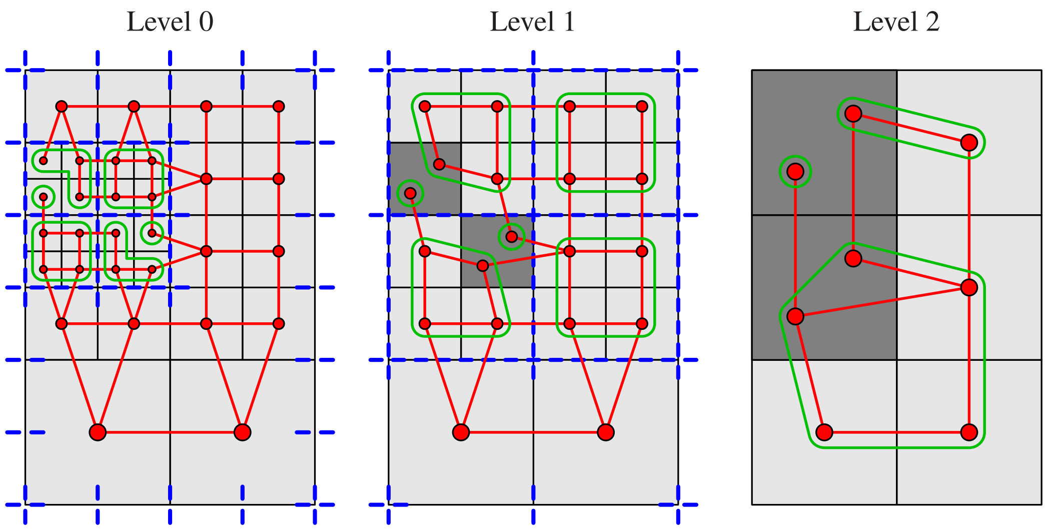


Figure 4.6: *Construction of the coarse multigrid levels based on the undirected graph representation in an octree grid. The notation is analogous to Figure 4.5.*

subsequently at each level $\ell = 0, 1, \ldots$ to build the coarse grid hierarchy (see also the 2D example in Figure 4.5):

Step 1: Analogous to the common approach, blocks of double grid cell size are considered. Let $\mathcal{C}^{\ell}_{(x,y,z)} \subseteq \mathcal{C}^{\ell}$, $x, y, z \in \mathbb{Z}$ denote the set of nodes corresponding to the cells of such a block, defined by $\mathcal{C}^{\ell}_{(x,y,z)} = \big\{c \in \mathcal{C}^{\ell} \mid \lfloor x_c/2 \rfloor = x, \lfloor y_c/2 \rfloor = y, \lfloor z_c/2 \rfloor = z\big\}$. By performing a depth first search on the subgraph of $(\mathcal{C}^{\ell}, \mathcal{E}^{\ell})$ induced by the nodes $\mathcal{C}^{\ell}_{(x,y,z)}$, the connected components of this subgraph are determined. For each connected component, a coarse grid cell $C \in \mathcal{C}^{\ell+1}$ at position $(x_C, y_C, z_C) = (x, y, z)$ is created, which subsumes the fine grid cells belonging to this component.

Step 2: The connectivity between cells on the coarse grid is determined from the connectivity on the fine grid. Two coarse grid cells C_1 and C_2 are connected iff there exist two connected fine grid cells c_1 and c_2 that are merged into C_1 and C_2, respectively. I.e., it is $\{C_1, C_2\} \in \mathcal{E}^{\ell+1}$ iff there exist c_1 in C_1 and c_2 in C_2 such that $\{c_1, c_2\} \in \mathcal{E}^{\ell}$. The notation c in C denotes that the fine grid cell c is belonging to the connected component corresponding to the coarse grid cell C.

The hierarchy that is constructed in this way has the property that small cuts disappear at coarser resolution levels. This is in-line with the multigrid idea that the coarse grid levels describe the macroscopic behavior of the body. Small cuts do not strongly affect this behavior and can therefore be neglected at one of the coarser levels.

4.5.2 Hierarchy Construction—Octree Grid

Building the hierarchy on the octree grid proceeds similar to the uniform case, with the exception that on each multigrid level it must be considered that cells with different size exist. At the transition from multigrid level ℓ to level $\ell + 1$, only cells of size 2^ℓ are merged. All other cells are passed on to the next coarser level. In Figure 4.6 a 2D example of the construction process is given.

4.5.3 Construction of Coarse Grid Equations

Once the cell hierarchy has been constructed, the coarse grid equations have to be built correspondingly. Since the corotational strain formulation is used, the equations have to be rebuilt in every time step to account for the element rotations.

Since all simulation elements have the same shape and only differ in size, we only need to precompute the element matrices of a single, generic element. The element matrices of an arbitrary element can be obtained from these matrices by appropriately scaling with the element's side length, density, and Young's modulus (see 2.2.3).

The restricted octree discretization leads to hanging vertices lying in the interior of other cells' edges or faces. To obtain a continuous discretization of the displacement (finest grid) or the error (coarse grids) using trilinear shape functions, a hanging vertex does *not* represent a degree of freedom but the value at this vertex is determined by linear (along edges) or bilinear (on faces) interpolation. Thus, we eliminate unknowns at hanging vertices via interpolation from unknowns at non-hanging vertices. Each cell finally depends on eight non-hanging vertices (which are not necessarily the geometric vertices of the cell), and we associate the cell with these vertices. This approach corresponds to the special finite elements introduced by Wang [Wan01] and extended to 3D by Sampath and Biros [SB10].

Computing the Galerkin coarse grid operators $\boldsymbol{A}^{2h} = \boldsymbol{R}_h^{2h} \boldsymbol{A}^h \boldsymbol{I}_{2h}^h$ means distributing the equations at the fine grid vertices to the respective coarse grid vertices (restriction $\boldsymbol{R}_h^{2h}$), at the same time substituting the unknowns at the fine grid vertices by interpolation from the unknowns at the coarse grid vertices (interpolation $\boldsymbol{I}_{2h}^h$). The respective weights of the trilinear interpolation operator $\boldsymbol{I}_{2h}^h$ and the restriction operator $\boldsymbol{R}_h^{2h} = \left(\boldsymbol{I}_{2h}^h\right)^{\mathrm{T}}$ are illustrated in Figure 4.7.

We compute the equations on each multigrid level per *cell*, since this leads to constant memory requirements per cell to store the equations and to constant memory access patterns during equation construction. In contrast, constructing the equations per shared *vertex* would yield varying memory requirements and access patterns, since

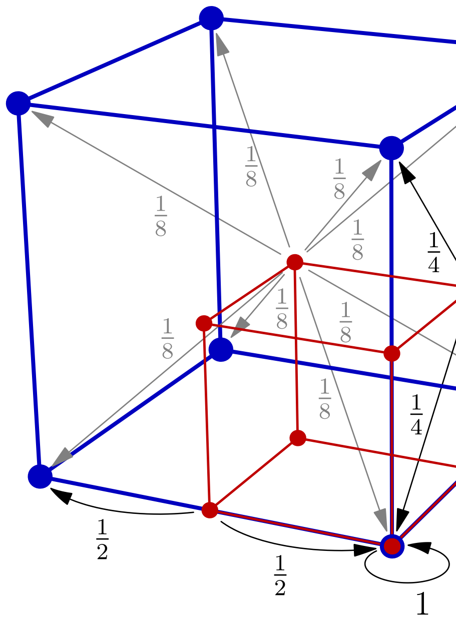


Figure 4.7: *Trilinear weights used to distribute the per-cell equations of a fine grid cell (red) to its corresponding coarse grid cell (blue) (restriction). For the interpolation of the unknowns from the coarse grid, the same weights are used. For simplicity reasons, the distribution is only illustrated for selected vertices.*

vertices can have a varying number of adjacent vertices due to the irregular topology induced by cutting.

At a specific level, the left-hand sides of the equations for each cell c are stored as rows of an 8×8-matrix $\boldsymbol{A}^c$, with each entry being itself a 3×3-matrix. The matrix $\boldsymbol{A}^C$ for a coarse grid cell C on the next coarser level is then computed from the matrices $\boldsymbol{A}^c$ of those cells c which are merged into cell C as follows:

$$\boldsymbol{A}^C_{[mn]} = \sum_{c \text{ in } C} \sum_{i=1}^{8} \left(\underbrace{w^{c \to C}_{i \to m}}_{\text{Restr.}} \sum_{j=1}^{8} \underbrace{w^{C \to c}_{n \to j}}_{\text{Interp.}} \; \boldsymbol{A}^c_{[ij]} \right) , \quad m, n = 1, \ldots, 8. \tag{4.1}$$

In this equation, i, j, m, n are cell-local vertex indices, and $w^{c \to C}_{i \to m}$ and $w^{C \to c}_{n \to j}$ are the corresponding weights for restriction and interpolation between the cells c on the fine grid and the coarse grid cell C, respectively (see Figure 4.7). Since all element matrices are symmetric (considering 24×24-matrices with scalar entries, i.e., $A^c_{pq} = A^c_{qp}$, $p, q = 1, \ldots, 24$), we only have to compute and store the matrices' lower triangular parts. To simulate very large models consisting of millions of elements, memory requirements can be reduced significantly by not storing the equations on the finest level, since they can be built on-the-fly from the generic element matrices and the element rotations.

To finally solve the system of equations, in the relaxation step and the residual computation step of the multigrid V-cycle equations per shared vertex have to be considered. These per-vertex equations are assembled from the per-cell equations as follows: At a shared vertex V, described by the set of cell vertices $V \subset \mathcal{C}^\ell \times \{1, \ldots, 8\}$ that are coalesced into this vertex, we accumulate the per-cell equations at these cell vertices. The resulting per-vertex equation at the shared vertex V then is

$$\sum_{(c,i)\in V} \sum_{j=1}^{8} \boldsymbol{A}^c_{[ij]} \boldsymbol{u}^{V(c,j)} = \boldsymbol{b}^V. \tag{4.2}$$

Here, $c \in \mathcal{C}^\ell$ denotes a cell incident to the shared vertex and $i, j \in \{1, \ldots, 8\}$ denote cell-local vertex indices. $V(c, j)$ is the shared vertex corresponding to vertex j of cell c, and $\boldsymbol{u}^{V(c,j)}$ is the displacement/error at this vertex. $\boldsymbol{b}^V$ is the right-hand side/residual at vertex V.

4.6 Surface Construction

To reconstruct a smooth polygonal surface that is aligned with the separated object parts we use the splitting cubes algorithm [PGCS09]. In a hexahedral simulation grid the splitting cubes algorithm constructs the surface topology in each cell depending on the patterns of the edges that are cut. The algorithm introduces additional points in the interior of the cells to construct a smooth surface representation. The placement of these points is driven by the normals of the cut surface at the intersection points (see Figure 4.8).

In order to use the splitting cubes algorithm in our approach, we have to adapt the algorithm to the particular simulation data structure we use, i.e., the linked voxel representation. Therefore, we consider the dual grid representation that is built from the links between the simulation elements. These links define the connectivity between the simulation elements, and exactly these links are cut by the cutting tool. For each link that is cut we store two distances from the link end points to the respective nearest cut point on that link. In this way, the cubic cells of the dual grid are cut at their edges, which results in a representation that allows us to directly apply the splitting cubes algorithm on these cells.

To let the reconstructed surface move according to the object deformations we bind the surface to the simulation vertices. For vertices on the links that are cut, the corre-

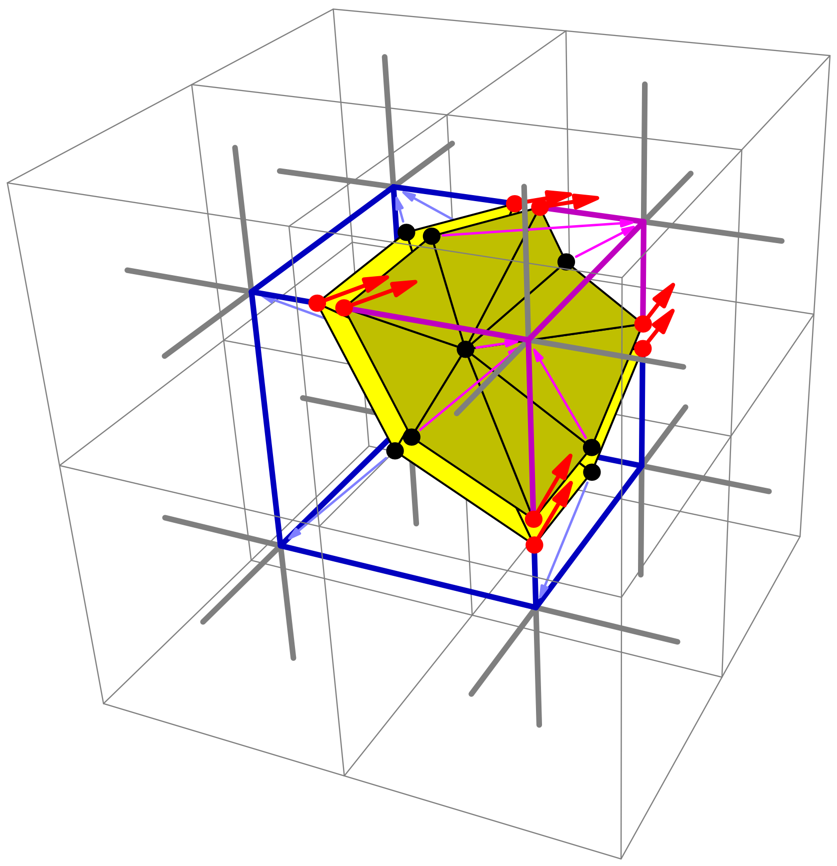


Figure 4.8: *Dual grid (bold lines) consisting of the links between the elements. For each cell of the dual grid a local render surface is built via the splitting cubes algorithm. The surface is spanned by the intersection points (red) between the cutting blade and the element links, and by the normal of the cutting tool at these points (red arrows). Vertices are bound to the nearest element of the respective connected component (thin blue and magenta arrows to the respective element centers).*

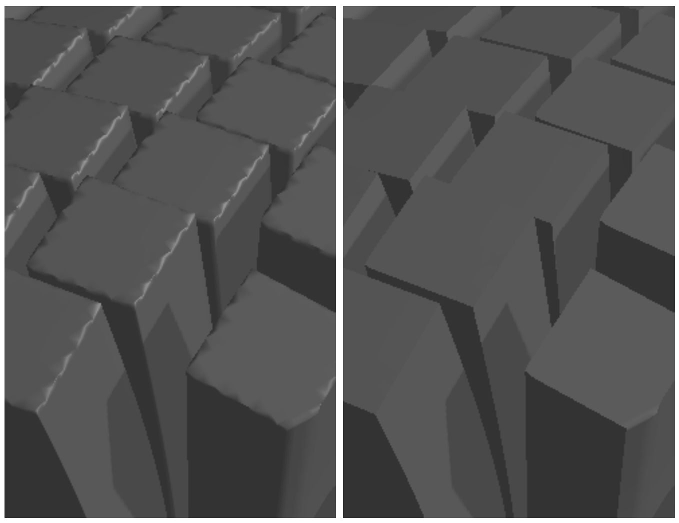


Figure 4.9: *Left: Standard per-vertex normals computed by averaging the face normals of all incident triangles lead to rendering artifacts at cutting edges. Right: Only face normals of triangles with the same cut ID are averaged, enabling smooth rendering of cutting surfaces while preserving sharp cutting edges.*

sponding simulation element is found by following the links to the left or to the right, respectively. To handle interior vertices we utilize the fact that each interior vertex is associated to a unique surface patch in the splitting cubes approach. Therefore, for an interior vertex the respective hexahedra can be found by following the links of the edges that are cut by the respective surface patch. From the determined set of hexahedra we choose the one that has the shortest distance to the surface vertex. The interior vertex is then bound to this hexahedron by means of trilinear interpolation or extrapolation.

Using standard per-vertex normals, which are computed by averaging the face normals of all triangles incident to the vertex, leads to rendering artifacts at cutting edges as shown in Figure 4.9. To achieve a visually pleasant rendering of the splitting cubes surface, we render both sharp cutting edges as well as smooth cutting surfaces by employing separate per-vertex normals for each individual triangle. To compute these normals, we make use of a cut surface ID, which is incremented for each new cut and then stored at the links intersected by this cut together with the distances. These IDs are distributed to the incident triangles of the splitting cubes surface, and for each triangle vertex the normal is computed by averaging the face normals of the incident triangles with the same ID as the considered triangle. The cut surface IDs can also be used to color the arising cutting surfaces differently (see Figure 4.1 right).

	#Hexahedra		#MG	Time [sec]	
Model	Initial	Cut	Levels	Cutting	Simulation
Bunny	118K (755K)	197K	5	0.693	2.42 (9.31)
Cube	83.3K (512K)	230K	4	0.781	3.13 (7.05)
Armadillo	493K (3717K)	643K	6	2.24	8.24 (47.2)
Bunny 2	26.1K (94.3K)	42.4K	4	0.143	0.498 (1.12)
Bunny 3	5.48K (11.9K)	8.27K	3	0.0286	0.0928 (0.130)

Table 4.1: *Timing statistics for cutting different deformable models using our adaptive finite element approach. Numbers in parentheses show respective measures for a uniform grid at the finest resolution level.*

4.7 Results and Analysis

In the following we analyze our method and show results that have been produced using this method. The analysis includes performance measures, a detailed evaluation of the convergence rates of the multigrid solver, and a comparison to alternative numerical solvers. All of our experiments were run on a desktop PC, equipped with an Intel Xeon X5560 2.8 GHz processor (we use a single core), 8 GB of RAM, and an NVIDIA GeForce GTX 280 graphics card.

4.7.1 Performance

Table 4.1 lists the deformable models we used and gives timings for cutting and simulation. The first three entries correspond to Figures 4.14 (left), 4.11 (right), and 4.1 (middle). The last two entries also correspond to the model shown in Figure 4.14 (left), but using an adaptive octree discretization reduced by one (Bunny 2) or two (Bunny 3) octree levels. The second and third column give the initial number of simulation elements and the number of elements due to adaptive refinements along the cuts. The fourth column specifies the number of levels of the multigrid hierarchy. The next column shows the time spent on adaptively refining the octree and rebuilding the finite element model. Note that these steps are only required when the cutting tool is moved. Finally, the time required to perform one time step is given. It includes the times it takes to recompute the equations on the simulation level and all coarse grids (as shown in Equation 4.1) and to perform three multigrid V-cycles with one pre- and one post-smoothing Gauss-Seidel step. In a dynamic simulation this is a reasonable choice since the solution from the last time step typically gives a good initial guess to start with in the current time step. These times are measured for the adaptively refined models. Times in parentheses are for a uniform grid at the finest resolution level.

Model	MG Time [sec]	MG Error Reduction	CG Error Reduction	CG Time [sec]
Bunny	1.30	0.100	0.935	9.76
Cube	1.94	0.0145	0.870	31.9
Armadillo	4.35	0.0190	0.973	101
Bunny 2	0.270	0.107	0.864	1.22
Bunny 3	0.0532	0.0734	0.655	0.172

Table 4.2: *Comparison of the adaptive MG solver to a CG solver with Jacobi preconditioner. Solver times are given for one time step. The error reduction is computed as $\|e_3\|_2 \,/\, \|e_0\|_2$, where e_0 and e_3 are the linearized error vectors before and after the time step, respectively. The last column shows the time that is required by CG to achieved the same error reduction as MG.*

Table 4.1 demonstrates the advantages of an adaptive octree discretization over a uniform discretization. It shows that a considerable amount of elements can be saved, yet modeling the induced discontinuities at equal resolution. For the higher resolution models only $1/6$ of the number of elements of the uniform grid are required. In general, this allows resolving the boundaries at a significantly higher resolution than would be possible in a uniform setting, giving the possibility to apply very complicated and thin cuts. Remarkably, even though the numerical simulation on an octree discretization is much more complicated than on a uniform discretization, the reduced number of elements also results in a significant performance gain over the uniform setting. In our scenarios, speed-ups between a factor of 4 and 5 are achieved. Furthermore, the statistics show that the computation time per simulation step depends linearly on the number of finite elements.

To demonstrate the computational efficiency and accuracy of the adaptive MG approach, in Table 4.2 we show a comparison of the experiments in Table 4.1 to the very same setup using a CG solver with Jacobi preconditioner. In this table only the solver times, i.e., the times required for solving the equations on the finest level, are considered. For the MG solver this means that the given times (2nd column) also include the construction of the equations on the coarse multigrid levels. The 3rd column gives the error reduction that was achieved by the adaptive MG solver. In the 4th column we show the error reduction by the CG solver within the same period of time as given in the 2nd column. Finally, the 5th column shows the times required by the CG solver to achieve the same error reduction as the MG solver.

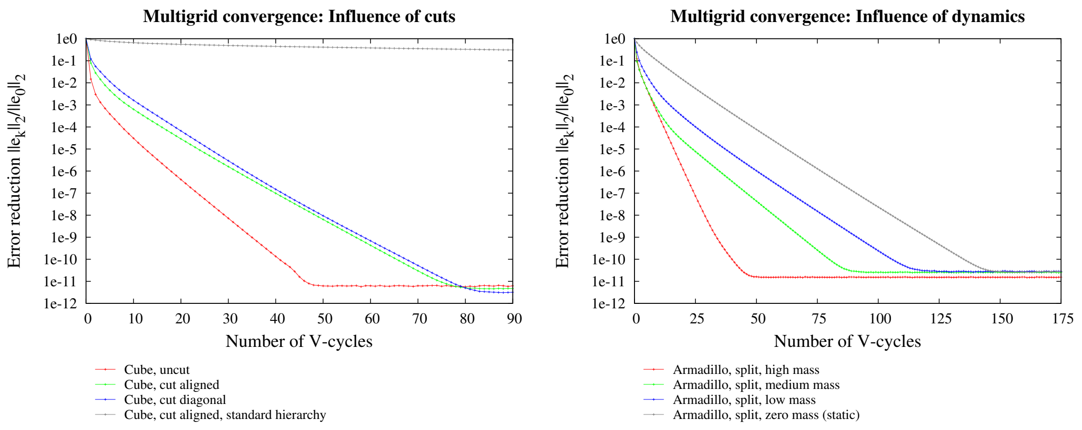


Figure 4.10: *Multigrid convergence. Left: Differently oriented cuts through a cube (Figure 4.11, right). Right: The Armadillo model (Figure 4.1, middle) having different mass.*

4.7.2 Multigrid Convergence Analysis

To demonstrate the effectiveness of the multigrid solver for simulating objects with complicated boundaries as induced by a cut, in Figure 4.10 (left) we show the solver's convergence for the scenario in Figure 4.11 (right). The curves show the total error reduction $\|e_k\|_2 \,/\, \|e_0\|_2$ dependent on the number k of V-cycles, with e_k denoting the error after the k-th V-cycle. The error is determined by using the solution obtained by a direct solver (Cholesky) as ground truth.

Three different tests were performed, all of them using a cube model aligned with the axes of the simulation grid: a) no cuts were performed, b) a grid-like cutting tool consisting of square blades aligned with the simulation grid was moved into the cube for a distance of half the cube's edge length, c) the cutting tool was rotated against the axes of the simulation grid to create cuts that are not aligned with the simulation grid. In each of the experiments the cuts were performed at a single point in time, and the solver's performance was measured for simulating the next time step right after the model was cut.

In experiment a) the grids of all multigrid levels exactly cover the domain of the cube, giving rise to an optimal multigrid hierarchy. A convergence rate $\rho = 0.66$ is achieved. The convergence rate is computed as $\rho = (\|e_{k_2}\|_2 \,/\, \|e_{k_1}\|_2)^{1/(k_2-k_1)}$ and denotes the average error reduction per V-cycle for those cycles k_1 through k_2 which exhibit a stable convergence rate. Experiments b) and c) demonstrate the influence of complicated boundaries on the convergence. In b) and c) the same spacing between the individual blades is used, but in contrast to b) a jagged object boundary is generated in

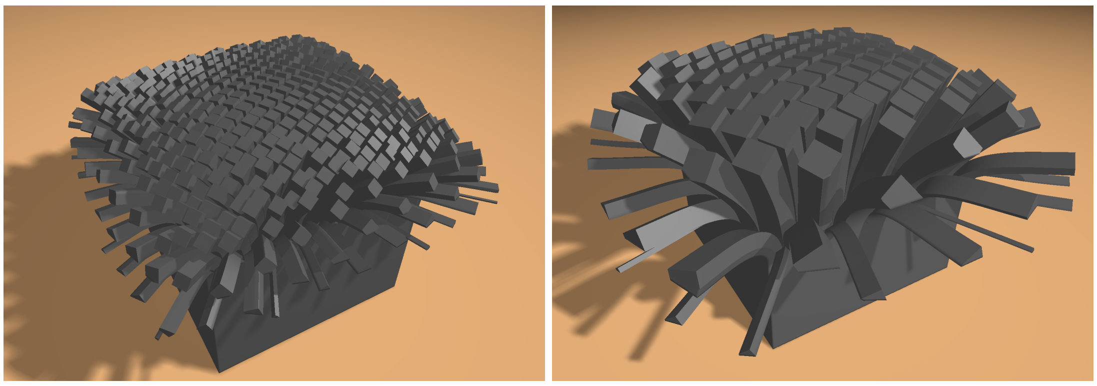


Figure 4.11: *In one time step a cube model (83K hexahedral elements) is cut using a grid-like cutting tool consisting of square blades, resulting in 204K (left) and 230K (right) elements. A radial force field is applied to fan out the resulting rods. Note that rods with different size bend differently. Cutting and simulation take less than 3.5 seconds (left) and 3.9 seconds (right) per time step.*

c) due to the cut orientation.

In both scenarios the convergence rate decreases to 0.75. This decrease is in line with previous findings in the context of geometric multigrid schemes, which have indicated decreasing convergence in case the coarse grid cells can no longer approximate the object's domain accurately, which is the case if the material is cut into disconnected parts. Notably, however, the convergence behavior does not depend on the smoothness of the boundary of the object.

To clearly demonstrate the effectiveness of our novel coarse grid hierarchy, Figure 4.10 (left) also shows the solver's convergence for experiment b) when a standard hierarchy is used, i.e., when cells are merged solely based on their spatial location without considering the connectivity between cells. In this case, the convergence rate degenerates from 0.66 for experiment a) (for a convex object without cuts, our novel hierarchy is identical to a standard hierarchy) to 0.99 for experiment b).

In another example we analyze the influence of the model dynamics on the solver's convergence. Typically, the convergence rate of an iterative solver increases with the relative magnitude of the main diagonal of the system matrix. Since the values on the main diagonal of our system matrix depend on the mass of the object, higher element masses yield better convergence rates, as indicated in Figure 4.10 (right) for the model shown in Figure 4.1 (middle). In comparison we have also simulated the static problem where the dynamics is not considered (gray curve). This curve indicates that independent of the object's mass a convergence rate of at least 0.85 can be achieved for this model by the multigrid solver.

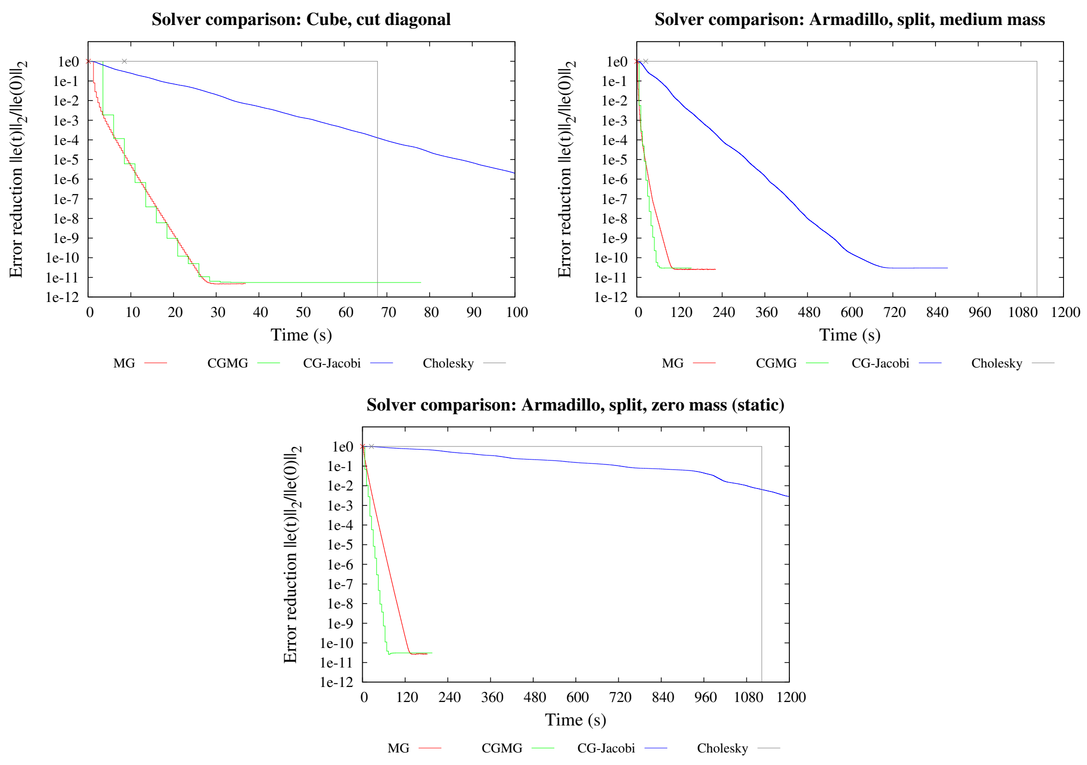


Figure 4.12: *Computational efficiency of different numerical solvers for deformable object simulation. The corresponding models are shown in Figures 4.11 (right) and 4.1 (middle). For the second model, medium and zero mass is used.*

4.7.3 Solver Comparison

In the following we compare the performance of the multigrid solver to alternative solvers that are widely used for simulating deformable objects. One important aspect in this analysis is the potential of the solver to reduce the error at most in a given period of time. This is required, for instance, to guarantee a given response time in interactive scenarios.

For the models shown in Figures 4.11 (right), and 4.1 (middle) with medium and zero mass, Figure 4.12 shows the error reduction of different numerical solvers over time for the solution of the first simulation time step, i.e., the cuts are performed to the object in its initial position, and an initial guess of $\mathbf{0}$ is used for the displacements. Here, $e(t)$ is the error after the respective solver has run for time t. In this comparison the respective initialization times of the solvers are included, i.e., each solver starts solely from the finest level equations and all computations specific to the solver—in case of the multigrid solver the construction of the coarse grids, the assembly of the coarse grid

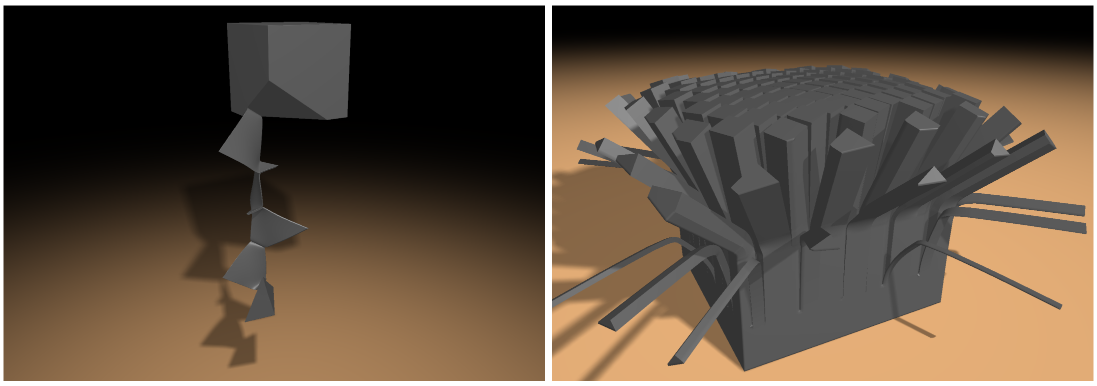


Figure 4.13: *Left: Cutting of thin slices (initially 345,000 hexahedra, 429,000 hexahedra after cutting). Cutting and simulation take 7.3 seconds per time step. Right: Cutting of a heterogeneous model (initially 83,000 hexahedra, 321,000 hexahedra after cutting). The cube consists of very stiff material, except of a thin horizontal layer at half height, leading to buckling of the rods. Cutting and simulation are performed at 5.5 seconds per time step.*

equations, and the initialization of the direct solver on the coarsest level—are included in the timings.

We analyze the Cholesky solver of the TAUCS library [TCRM03], a CG solver with Jacobi preconditioner, and the proposed multigrid solver. The multigrid solver is either used directly (MG) or as a preconditioner for the conjugate gradient method (CGMG). For each solver a cross on the respective curve highlights the structural initialization time that is only required in case of topological changes. For the Cholesky solver, structural initialization consists of the symbolical factorization of the system matrix. For the multigrid solver, structural initialization refers to the construction of the coarse grid hierarchy. (For MG and CGMG the crosses fall onto the same position.) CG does not have a structural initialization time.

Besides needing a vast amount of time to solve the system of linear equations, Cholesky turns out to be impractical for applications requiring a first approximation of the solution within a short time interval. CG-Jacobi shows a significantly slower convergence rate than MG and CGMG. Both multigrid solvers converge much faster towards the solution than their competitors and, in particular, they are able to provide good approximations in a significantly shorter time. CGMG increases the error reduction per time slightly, but this benefit only pays off if a large number of cycles is performed.

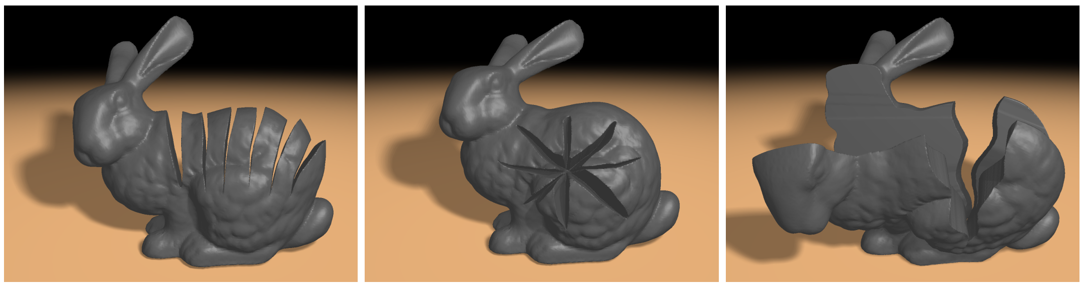


Figure 4.14: *A potpourri of different cuts is applied to the Stanford bunny model to demonstrate the flexibility of our approach. Multiple overlapping cuts (middle) can be handled accurately. From left to right, the cut models consist of 197,000, 166,000, and 177,000 hexahedral elements. Cutting and simulation are performed at approximately 3.1, 2.6, and 2.8 seconds per time step, respectively.*

4.8 Conclusion and Future Work

In this chapter, we have proposed an efficient approach for physics-based cutting of deformable objects. This approach employs an adaptive octree grid to represent cuts at very fine scales. The cutting algorithm is incorporated into a multigrid scheme, giving rise to a numerical solver that can handle topological changes in the simulation grid at high computational efficiency. To reconstruct a smooth surface from the disconnected object parts, an extension of the splitting cubes approach has been proposed. This extension uses the dual simulation grid to build a boundary surface that is consistent with the cuts in the simulation grid.

It is worth noting that in a particular scenario, the unrestricted refinement along a cut may result in more elements than would actually be required to solve accurately. For instance, homogenization approaches [NKJF09, KMOD09] could possibly reduce the number of elements by identifying the material properties at coarser scales from those of their constituents and using only the respective coarse grid cells in the simulation. Thus, even with less efficient numerical solvers homogenization approaches can often simulate very fast. In the general case, however, such approaches yield an approximation to the numerical solution on a finer discretization using ‘non-homogenized’ finite elements, since they reduce the number of degrees of freedom to solve for. The principle underlying our approach is to simulate on a finite element model that has as many degrees of freedom as given by the initial discretization and to achieve high speed by employing a computationally efficient numerical solver. Thus, our approach always simulates accurately at the possible expense of a higher number of simulation elements.

The proposed method opens a number of future research directions. Since in the current approach we fix the resolution of the simulation grid in the interior of the deforming object, the resulting number of degrees of freedom might not be sufficient to accurately represent the induced deformations. Therefore, a dynamic adaptation of the finite element discretization in the object's interior is desirable. This can directly be integrated into our approach, but it first requires to develop an a priori oracle to decide where to refine.

Another interesting question is how to efficiently adapt a high resolution render surface to the topological changes of the simulation mesh. In the current approach the resolution of the finite element model and the render surface are coupled due to the use of the splitting cubes algorithm. Even though it is possible to bind a higher resolution render surface to the vertices of the initial simulation grid, it is unclear how to adapt this surface to the induced topological changes at a speed comparable to that of the simulation.

Furthermore, we will also integrate collision handling into our approach. In particular we plan to use the deforming splitting cubes surface to detect self-collisions [TKH$^+$05] and to propagate the collision response to the respective simulation elements.

Chapter 5

Computational Steering for Implant Planning in Orthopedics

Fast and reliable methods for predicting in-vivo bone stresses are of great importance for implant planning in orthopedics. To avoid adaptive remodeling with cortical thinning and increased porosity of the bone due to stress shielding, in a preoperative planning process the optimal implant shape, size, and position has to be determined. This process involves interactive implant positioning within the bone as well as simulation and visualization of the stresses within bone and implant. In this chapter, we present a prototype of such a visual analysis tool, which provides the first computational steering environment for optimal implant selection and positioning. This prototype considers patient-specific biomechanical properties of the bone to select the optimal implant shape, size, and position according to the prediction of the individual load transfer from the implant to the bone. We employ our GPU-based multigrid finite element solver for real-time elasticity simulation to compute the bone stress distribution at interactive update rates. By utilizing a real-time GPU-method to detect elements that are covered by the movable implant, we can automatically generate computational models from patient-specific CT scans in real-time, and we can instantly feed these models into the simulation process. In combination with GPU-based visualization techniques for the resulting stress tensor fields, this enables to interactively investigate the effect of different implant shapes, sizes, and positions on the stress distribution in the patient-specific bone, and thus provides a new quality of orthopedic surgery planning.

This chapter is based on material that has been originally published in C. Dick, J. Georgii, R. Burgkart, and R. Westermann, *Computational steering for patient-specific implant planning in orthopedics*, Proc. Eurographics Workshop on Visual Computing for Biomedicine, 2008, pp. 83–92 [DGBW08]. The copyright for this material is owned by Eurographics.

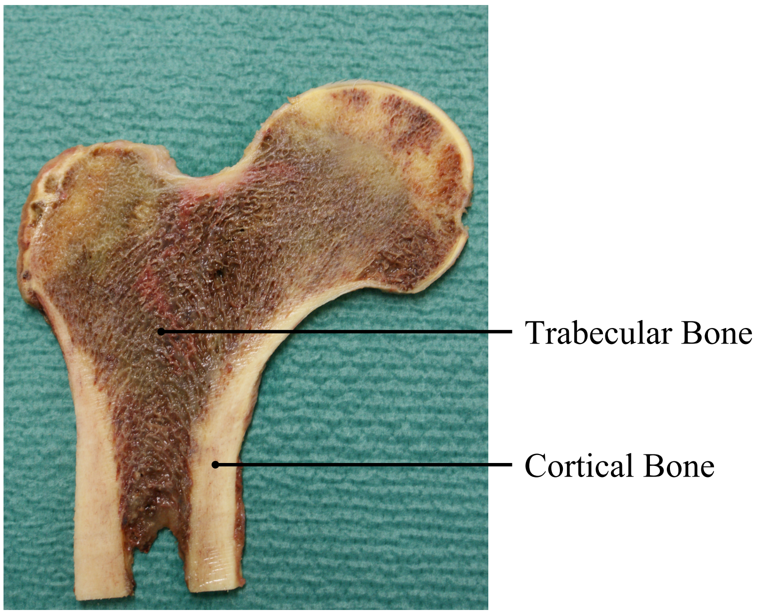


Figure 5.1: *Cross section through a human femur, showing the bone shell consisting of stiff cortical bone and the bone interior consisting of spongious trabecular bone.*

5.1 Introduction and Related Work

Methods for predicting in-vivo bone stresses are of great importance for clinical applications such as fracture fixation or total hip joint replacement. Such procedures call for highly efficient and reliable analysis tools that allow the surgeon during an interactive, preoperative design loop to find the optimal shape, size, and position of an implant by matching its mechanical properties with those of the individual bone. The clinical relevance of such a planning approach is due to the well known fact that an essential determinant factor for the long term stability of an implant is a physiological load transmission to the adjacent bone stock.

A human bone consists of two types of tissue (see Figure 5.1): Stiff cortical bone, forming the outer shell, as well as spongious trabecular bone in the interior. These two types of bone tissue are the consequence of a natural optimization process. Bone is a living tissue and adapts to changes of the mechanical load situation (for example due to bone growth) by bone formation or resorption, dependent on whether the load has increased or decreased, respectively. In this way, the bone continuously provides high mechanical stability at light weight. Paradoxically, bone resorption also occurs if the load exceeds a certain critical magnitude.

The insertion of an implant changes the stress distribution in the bone in that stresses

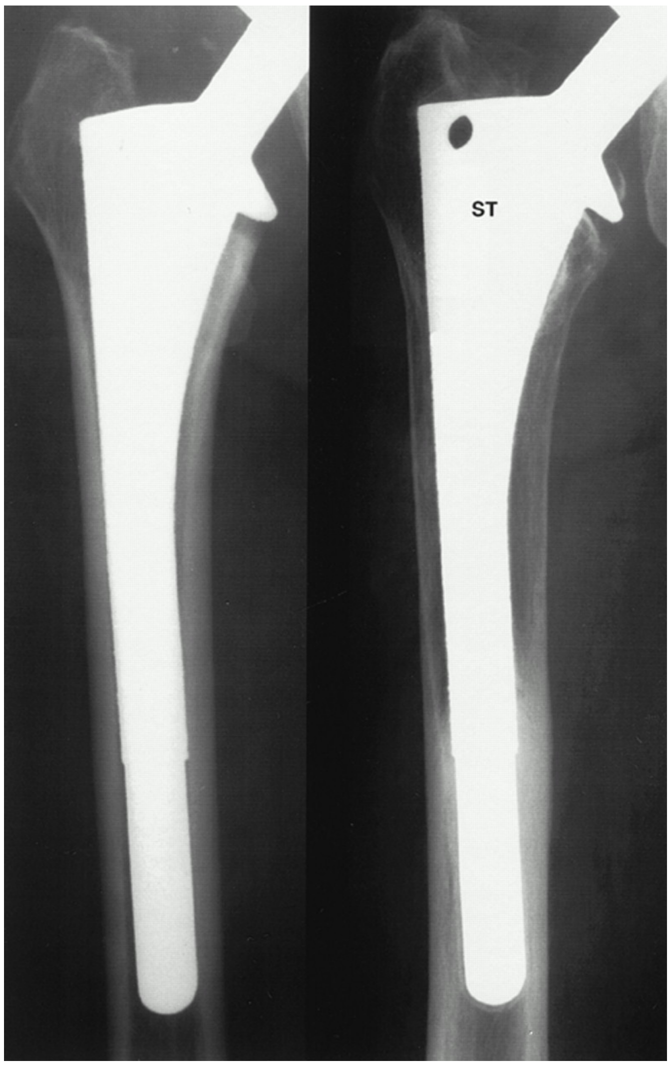


Figure 5.2: *X-rays of a patient's proximal femur, taken immediately (left) and two years after the insertion of an implant (right). In the right X-ray, a significant reduction of the density of the cortical bone around the upper part of the implant stem is visible. This degeneration of bone tissue is a consequence of stress shielding.*
Image reprinted from W. D. Bugbee, W. J. Culpepper, II, C. A. Engh, Jr., and C. A. Engh, Sr., Long-term clinical consequences of stress-shielding after total hip arthroplasty without cement, *Journal of Bone and Joint Surgery, American Volume 79 (1997), no. 7, 1007–1012 [BCEE97], with kind permission from The Journal of Bone and Joint Surgery.*

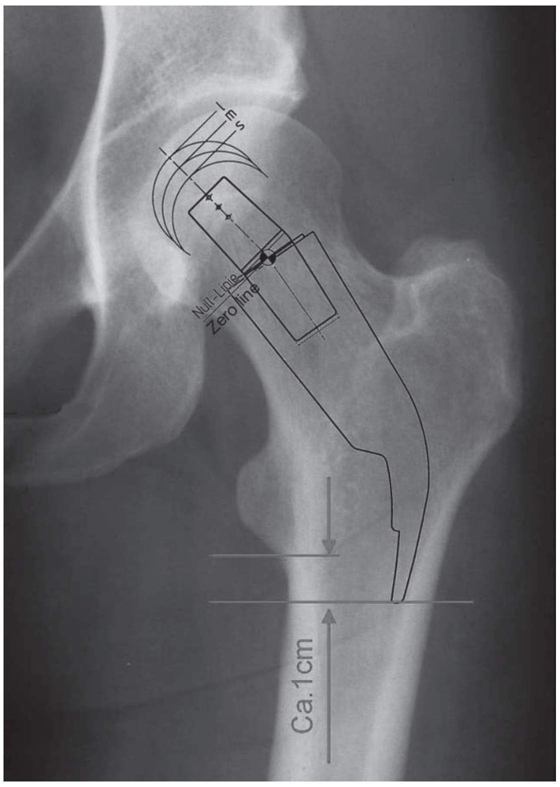


Figure 5.3: *Preoperative implant planning based on an X-ray of the patient's hip joint and a set of transparent 2D template sheets with the outlines of the implants.*
Image reprinted from M. Rudert, U. Leichtle, C. Leichtle, and W. Thomas, Implantation technique for the CUT-type femoral neck endoprosthesis, *Operative Orthopädie und Traumatologie 19 (2007), no. 5/6, 458–472 [RLLT07], with kind permission from Springer Science + Business Media.*

are bypassed by the implant, which leads a reduction of stresses in certain regions of the bone. This effect is referred to as *stress shielding*. As a consequence of the bone's adaptation to changes in the stress patterns, stress shielding causes a degeneration of bone tissue (see Figure 5.2), which can finally lead to osteopenia, bone fracture, or aseptic loosening of the implant [OH78, SV02].

The main objective is thus to simulate the mechanical response of the patient-specific bone and the implant to an applied load, and to find of all possible implant shapes, sizes, and positions the one that results in the most physiological stress distribution, i.e., minimizes stress shielding. The challenge in developing such an analysis tool results from the complexity of simulating stresses in a physically correct way and at interactive update rates, and from the difficulty of considering a movable implant with specific mechanical properties in such a simulation. Furthermore, such a tool is highly demanding on advanced visualization technology because it requires simultaneous visualization of dynamically changing surface and volume structures at rates that allow for interactive monitoring and analysis.

Due to the aforementioned challenges, to the best of our knowledge, so far no surgical planning system incorporates the biomechanical data about the patient-specific bone and the implants into the planning process. In the majority of clinical centers worldwide, the preoperative planning for the selection of an implant, e.g., an endoprosthesis for total hip joint replacement, is performed on an X-ray of the patient's bone, using a set of transparent template sheets with the outlines of the implants [Wix08] (see Figure 5.3). The drawbacks and limitations of such a two-dimensional approach are obvious, especially because rotational misalignment cannot be controlled and the position of the endoprosthesis can only be revised in 2D planes.

To overcome these limitations, 3D planning systems have been developed in the last years that provide a 3D visualization of the implant position using patient-specific CT data [HEPP01, KOO$^+$09, STWH00, VCT$^+$04]. These systems are used for implant selection from a set of standard prostheses as well as for designing custom-made implants for abnormal anatomies. To accurately reproduce the planned implant position during surgery, computer-assisted navigation systems have been developed [AP04, WG04, Wix08]. Since the existing systems are purely geometry-based, stress shielding is so far only considered according to the surgeon's subjective medical experience. However, to allow for a precise and objective assessment of an implant with respect to stress shielding, the prediction of the stress distribution in the patient-specific bone and the implant is mandatory. This information is still missing in current 3D planning systems.

5.2 Contribution

We present a prototypical 3D implant planning system that addresses the aforementioned requirements (see Figure 5.4). Following previous work in orthopedics [BOM$^+$07, TSH$^+$07, YTM07], we use 3D finite element analysis to simulate the stresses in the bone and implant due to applied loads. The physical model underlying our approach is based on linear elasticity and thus mimics the behavior of the bone at the macro-level during normal movements [KGW$^+$94]. The mechanical properties of the bone are derived directly from the Hounsfield units of the voxels in a measured CT scan as proposed in [KLS94, KF03]. In particular, we use our GPU-based finite element method for the simulation of elastic objects (see Chapter 3), reduced to the

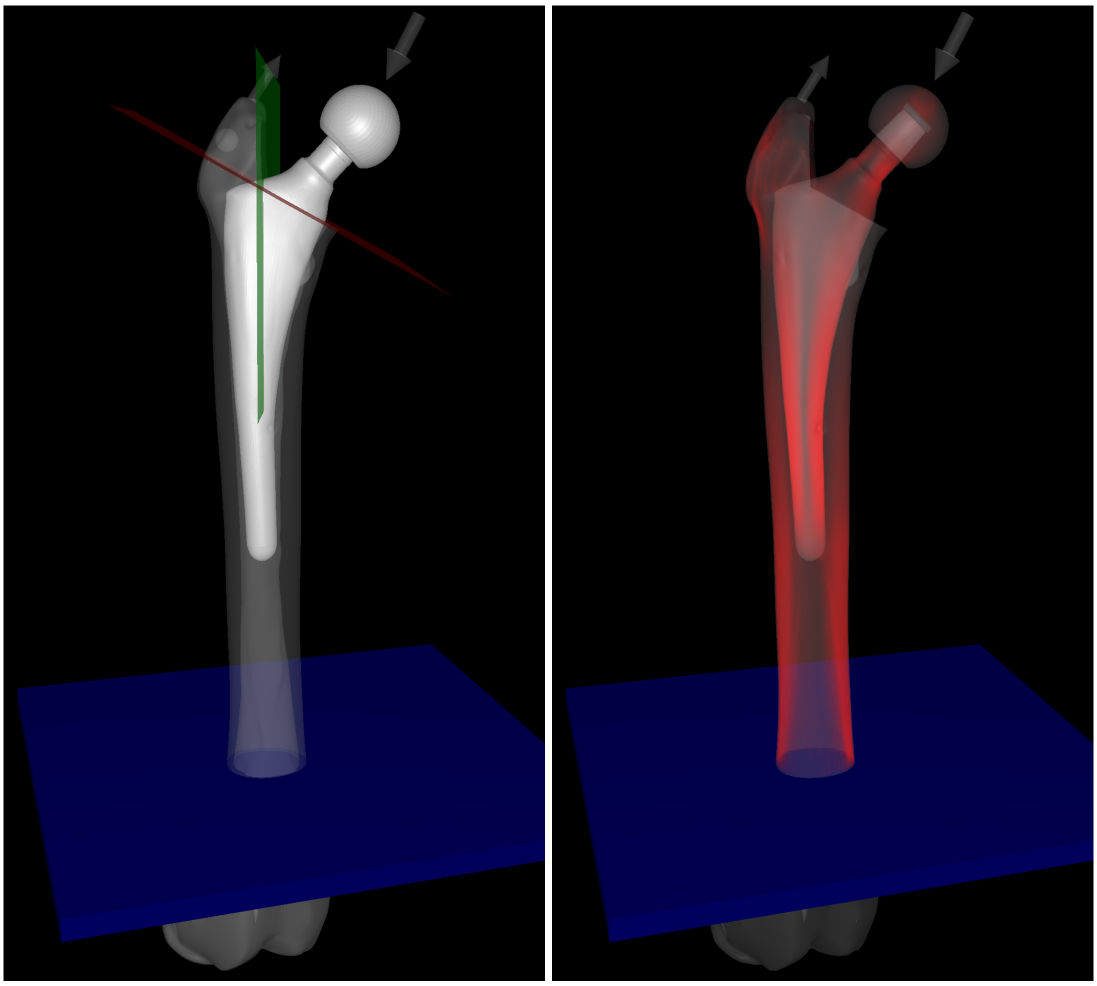


Figure 5.4: *Screenshots of our simulation environment: Left: Semi-transparent rendering of the femur, including the boundary between cortical and trabecular bone, supports the interactive positioning of the implant. To specify the removal of the femoral head, two cutting planes (red and green) are used. The directions and positions of the applied forces are indicated by the arrows and the small spheres. Right: The simulated stresses in the bone and implant are visualized by using volume rendering.*

simulation of the static behavior of linear elastic material. This method enables to simulate the stresses in the bone and implant in quasi real-time on a standard desktop PC. In particular, we achieve simulation rates of more than 13 updates per second using a finite element model at half the resolution of the CT scan (i.e., one hexahedral finite element per 2^3 CT voxels), and still more than 2 updates per second at the resolution of the CT scan (i.e., one hexahedral finite element per CT voxel).

To further improve the proposed system towards a computational steering environment, we have integrated 3D visualization and interaction mechanisms. Specifically, we enable the user to interactively place implants of different shape and size in the patient-specific bone and to apply loads on the bone and the implant, and we provide immediate visual feedback of simulated stresses via volume ray-casting based on the scalar von Mises stress. In addition, we provide the option to visualize the absolute von Mises stress values via color coding on a set of parallel, axial slices through the bone.

To the best of our knowledge, this is the first time that changes in bone stresses due to variations in implant shape, size, and position and external loads can be monitored and analyzed in quasi real time. This has been made possible by a fully automatic approach for the generation of a computational model based on the initial CT scan. A fast voxelization technique is employed to determine all CT voxels that are covered by the implant, which then get assigned the material properties of this implant. The FE analysis considers these voxels and simulates their interaction with the surrounding bone voxels under load. As the voxelization process is performed entirely on the GPU, it does not impose any performance constraints.

In our work, we exemplarily focus on implant planning for total hip joint replacement. Considering that alone this particular medical procedure is performed more than one million times per year worldwide, and 135,000 times in Germany [GG06], demonstrates that our tool is of high clinical relevance. Even though the proposed advanced surgery planning based on CT scans—considering the additional effort of acquiring these scans as well as the additional radiation exposure of the patients—might not be required for every standard surgery, it is of particularly high importance for complicated anatomies and revision cases.

The remainder of this chapter is organized as follows: In the following section, we give an overview of our prototypical 3D planning system. Next, we address application-specific extensions to our GPU-based finite element simulation method, including the generation of the computational model of the bone and the implant. The following section discusses the visualization of the virtual 3D environment and the simulated stress distributions. We then explain the integration of the simulation and visualization com-

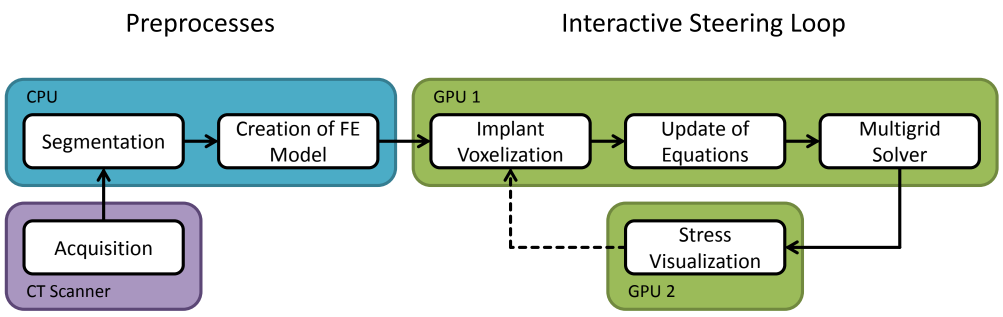


Figure 5.5: *Overview of the proposed surgery support system.*

ponents into the overall system for the realization of a computational steering environment. Finally, we demonstrate the application of our system to a real-world scenario, and give a detailed performance analysis of the individual components.

5.3 System Overview

Figure 5.5 illustrates the different components of the 3D planning system for hip joint replacement. The system is divided into preprocesses and the interactive steering loop. The preprocesses are performed once before interactive implant selection and positioning and stress monitoring starts, and include the acquisition of the CT scan, the segmentation of the bone voxel model from this scan, as well as the creation of the finite element model. The preprocesses are performed on the CPU. The interactive steering loop comprises the simulation and visualization of the stress distribution in the bone and implant. For the stress simulation, we first voxelize the implant surface mesh in its current position to determine the CT voxels covered by the implant, which then get assigned the material properties of this implant. We then update the underlying linear system of equations to consider the modified material properties of the finite elements, and solve this system by using our efficient multigrid solver. The simulated stresses are visualized. Simulation and visualization are each running on a separate GPU.

5.4 Simulation

In this section, we describe the simulation component of our computational steering environment, including the creation of the computational model of the bone from a patient-specific CT scan. According to the widely accepted assumption of linear elastic response of the bone under normal load, our model is based on linear elasticity using heterogeneous material properties described by the Young's modulus E. We assume the material to be isotropic, which is a common abstraction from the real bone behavior due to the difficulty in identifying the anisotropic material parameters.

The stress computation is based on our GPU-based finite element method for the simulation of elastic objects (see Chapter 3). However, we now simulate the static deformation behavior of a linear elastic body, i.e., dynamics and the corotational formulation of strain have been deactivated. Using a finite element discretization, this behavior is described by the linear system of equations (2.73).

One central aspect of our computational steering approach is the possibility to obtain immediate visual feedback of the stress patterns while the surgeon interactively positions the implant in the simulation environment. Therefore, we provide a novel approach to handle a movable implant in the numerical simulation. This approach is described in Section 5.4.2.

5.4.1 Finite Element Model Creation

The finite element model of the bone is generated from a patient-specific, high-resolution CT scan. The axial scan was performed on a Siemens Sensation Cardiac 64 CT scanner with 1.0 mm slice thickness and 0.75 mm pixel size. The samples are given on a $256 \times 256 \times 470$ lattice. In a preprocess, the bone is segmented from the CT data, i.e., the voxels are classified into bone voxels and exterior voxels, and the exterior voxels are removed from the voxel model. Then, for each voxel, we derive the Young's modulus value E (in MPa) from the Hounsfield unit HU according to [KLS94, KF03]

$$E(HU) = \begin{cases} 33900\,(8.2106 \cdot 10^{-4} HU + 0.057663)^{2.20} & HU \leq 320 \\ 10200\,(8.2106 \cdot 10^{-4} HU + 0.057663)^{2.01} & HU \geq 660 \\ 5307\,(8.2106 \cdot 10^{-4} HU + 0.057663) + 469 & 320 < HU < 660 \end{cases} . \quad (5.1)$$

From the bone voxel model, we generate a finite element model consisting of hexahedral finite elements aligned on a 3D Cartesian grid, such that each hexahedron subsumes a block of n^3 CT voxels. A finite element is created if at least one voxel of the respective block exists. The Young's modulus values of the finite elements are obtained by averaging the Young's modulus values of the respective voxels. In order to obtain a well-posed static elasticity problem, we use a minimum Young's modulus value for the finite elements of 10 MPa. For the Poisson's ratio ν a value of 0.3 is used.

In our experiments, we use finite element models at the resolution of the CT scan, i.e., one hexahedron per CT voxel ($n = 1$), and at half the resolution of the CT scan, i.e., one hexahedron per 2^3 CT voxels ($n = 2$).

The distal part of the femur is fixed (indicated by the horizontal plane (blue color) in Figure 5.4). Since in our current implementation the fixation is not changed during the interactive steering of the simulation, we remove all hexahedra that are completely fixed (i.e, that have 8 fixed vertices) from the finite element model.

5.4.2 Modeling of the Implant

During a total hip joint replacement surgery, the femoral head is removed, and the inner trabecular bone is partially removed in order to be able to insert the implant (for details about the medical procedure we refer the reader to [RLLT07]).

One possible approach for obtaining a computational model reflecting this medical procedure would be to delete the bone voxels corresponding to the parts of the bone that are removed by the surgeon, then to build a new finite element model of the bone, and finally to explicitly simulate the interaction of the bone model with a separate finite element model of the implant. Since these are very difficult and time-consuming operations, we have developed a different approach that enables us to generate the computational model as well as to perform the simulation based on this model both in real time.

The basic idea of our approach is to directly incorporate the implant into the finite element model of the bone by assigning the material properties of the implant to the bone voxels that are covered by the implant in its current position. To determine these voxels, we compute a voxelization of the implant surface triangle mesh with respect to the CT voxel grid, which directly classifies the voxels into bone and implant voxels. Since the voxelization has to be recomputed whenever the implant is moved, we compute the voxelization directly on the GPU to avoid any performance constraints (see Section 5.4.3 for a detailed description of the voxelization algorithm). By using this approach, we simulate a non-slip boundary between the bone and the implant, which is a reasonable approximation since the implant is fixed in the bone. For the implant voxels, we use a Young's modulus of 110 GPa, corresponding to the respective value for titanium alloys.

The removal of the femoral head region is modeled by assigning a Young's modulus of 0 to the respective bone voxels. By using two cutting planes (see Figure 5.4), this region is specified as the intersection of the positive half spaces spanned by these planes.

After updating the voxels' material properties, we recompute the material properties of the finite elements as described in the previous section, and we update the underlying linear system of equations to reflect the modified material properties of the finite elements. It is worth noting that in our approach the finite element model is not changed,

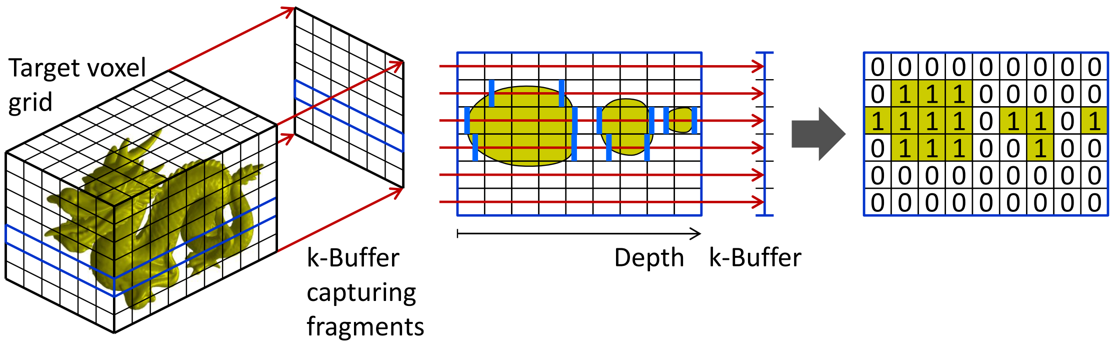


Figure 5.6: *GPU-based voxelization of a triangle surface mesh using two rendering passes. In the first pass, the depth layers are captured by using the stencil-routed k-buffer. The depth information is used in the second pass to create the voxelization.*

because neither are hexahedral cells removed nor added to the model. Therefore, we only have to recompute the coefficients of the per-vertex equations. The index structures used to represent the finite element model and the multigrid hierarchy, which are computed on the CPU in a preprocess (see Chapter 3), remain unchanged during the entire runtime of the simulation.

5.4.3 GPU-Based Voxelization

In our computational steering approach, the implant is modeled by adapting the Young's modulus values of the bone voxels that are covered by the implant. By using a GPU-based voxelization method, and thus by exploiting the rasterization and parallel processing capabilities available on recent GPUs, we can accurately determine these voxels in each simulation frame without performance constraints.

Our voxelization method uses two rendering passes (see Figure 5.6). In the first pass, the implant mesh is rendered into a stencil routed k-buffer [MB07] to obtain its depth layers. In the second pass, the depth information is used to build a 3D binary volume representing the voxelization of the implant.

The stencil routed k-buffer allows capturing of multiple fragments per texel in a single rendering pass. When writing to a multisampled texture while multisample antialiasing is disabled, an incoming fragment is spread to all subsamples of the respective texel, but the stencil is tested individually for each subsample. At the beginning, the stencil buffer is initialized with the values $2, 3, 4, \ldots, 9$ for the eight subsamples of each texel. The stencil test is set to "passing if equal to 2", and the stencil fail and pass operation is set to "decrementing". Depth testing is disabled. With respect to a specific texel, for the first incoming fragment the stencil test passes exactly for the first subsample, and thus the fragment is written into this subsample. After execution of the stencil operation, the resulting stencil values are $1, 2, 3, \ldots, 8$. For the second incoming fragment the stencil test thus passes exactly for the second subsample. In this way, the incoming fragments for a texel are successively routed to different subsamples, i.e., the i-th fragment is stored in the i-th subsample. This functionality allows us to capture multiple fragments per texel in a single rendering pass.

In the first rendering pass, we render the implant's surface mesh into the k-buffer to capture the depth layers of the implant. The only attribute being associated with the fragments is the voxel space depth, thus the k-buffer consists of a single component floating point texture. Current graphics hardware supports up to eight subsamples per texel, and thus enables capturing of up to eight depth layers in a single rendering pass—to voxelize objects with higher depth complexity further rendering passes would be required [MB07]. We use an orthographic projection with a view frustum matching the bounding box of the voxel volume, and we choose a view port that aligns the texels in the k-buffer with the voxel grid. Front/back face culling and depth testing are disabled. By rendering the implant's mesh, each texel captures the voxel space depth values of the entry and exit points of an imaginary ray through the implant.

In the second pass, we build the 3D binary volume representing the implant. This volume is stored in a four component unsigned integer 3D texture (R32G32B32A32), with each voxel being encoded into one bit. Thus, 128 voxel slices are stored in one 3D texture slice. By adding the SV_RenderTargetArrayIndex semantic to the geometry shader output declaration, which enables the geometry shader to specify the respective target slice within the 3D texture, the entire volume can be created in a single rendering pass.

The bit patterns representing the voxelization are created in the fragment shader. By using up to 8 render targets and thus simultaneously accessing 8 texture slices, the fragment shader can output a vector of up to 1024 voxels at once. First, the k-buffer entry corresponding to the respective ray is read and the depth values are sorted in ascending order. Each consecutive pair of depth values then represents an entry and an exit point into and from the implant. If $z_{\text{entry}}, z_{\text{exit}}$ denote the voxel space depth values, the entry and exit voxel indices $k_{\text{entry}}, k_{\text{exit}}$ are determined by $k_{\text{entry}} = \lceil z_{\text{entry}} - \frac{1}{2} \rceil$ and $k_{\text{exit}} = \lfloor z_{\text{exit}} - \frac{1}{2} \rfloor$. The corresponding voxel block between the entry and the exit voxel is created by adding $\sum_{i=k_{\text{entry}}}^{k_{\text{exit}}} 2^i = 2^{k_{\text{exit}}+1} - 2^{k_{\text{entry}}}$ to the bit pattern. Note that due to the triangle rasterization rules and due to the specific rounding in the computation of k_{entry} and k_{exit}, a voxel is created iff the voxel center is lying in the interior of the surface mesh.

5.4.4 Stress Calculation

After the per-vertex displacement vectors $\boldsymbol{u}_k$ are computed according to Equation (2.73), the stresses acting in the bone and the implant are determined according to Equations (2.35) and (2.58) as

$$\begin{aligned} \overline{\boldsymbol{\sigma}}(\boldsymbol{x}) &= \overline{\boldsymbol{C}}^e \overline{\boldsymbol{\varepsilon}}(\boldsymbol{x}) \\ &= \overline{\boldsymbol{C}}^e \boldsymbol{B}^e(\boldsymbol{x}) \underline{\boldsymbol{u}}^e \end{aligned} \quad , \qquad \boldsymbol{x} \in \Omega^e. \tag{5.2}$$

In this chapter, the stress visualization is based on the scalar von Mises stress [Bat02], which is computed from the stress tensor $\boldsymbol{\sigma}$ according to

$$\sigma_{\text{Mises}} = \sqrt{\sigma_{11}^2 + \sigma_{22}^2 + \sigma_{33}^2 - \sigma_{11}\sigma_{22} - \sigma_{11}\sigma_{33} - \sigma_{22}\sigma_{33} + 3(\sigma_{12}^2 + \sigma_{13}^2 + \sigma_{23}^2)}. \tag{5.3}$$

For the GPU-based visualization of the stress tensor field, we sample this field at the centers $\boldsymbol{x}_C^e$ of the finite elements to obtain a per-element stress tensor $\boldsymbol{\sigma}^e$, i.e.,

$$\begin{aligned} \overline{\boldsymbol{\sigma}}^e &= \overline{\boldsymbol{\sigma}}(\boldsymbol{x}_C^e) \\ &= \underbrace{\overline{\boldsymbol{C}}^e \boldsymbol{B}^e(\boldsymbol{x}_C^e)}_{\boldsymbol{S}^e} \underline{\boldsymbol{u}}^e \\ &= \boldsymbol{S}^e \underline{\boldsymbol{u}}^e. \end{aligned} \tag{5.4}$$

Since all elements have the same shape, and since the per-element elasticity tensor $\boldsymbol{C}^e$ scales linearly with the Young's modulus value, we only need a single, precomputed instance of the matrix $\boldsymbol{S}^e$.

The corresponding von Mises stress values σ_{Mises}^e are stored in a 3D texture with the resolution of the hexahedral finite element grid. This texture is sampled by means of trilinear interpolation during the visualization process.

5.5 Visualization

For the rendering of the virtual 3D environment and the visualization of the simulation results, we use GPU-based rendering techniques. In particular, we employ semi-transparent rendering of surface meshes for the bone and the implant, and volume ray-casting for the von Mises stress scalar field. These techniques enable us to simultaneously render the simulation objects as well as the simulation results and thus to show the results in their respective context, without limiting perception due to occlusions. To render opaque and semi-transparent geometry as well as the volume in correct visibility order, we use a multi-pass approach, which again utilizes the stencil-routed k-buffer.

Note that in the considered application the occurring displacements are small and do not lead to a significant deformation of the bone and implant geometry. Therefore, the deformation of the bone and implant geometry is neglected in the visualization, i.e., we render this geometry in the undeformed state.

First, we render all opaque geometry into the frame buffer with enabled depth testing. The content of the depth buffer is later used during ray-casting to correctly handle occlusions of semi-transparent geometry and the volume rendering by opaque objects. Then, we render all semi-transparent geometry into an off-screen k-buffer, with depth testing as well as front/back face culling being disabled. By means of the k-buffer we can capture for each pixel up to eight incoming fragments, independently of the rendering order of the geometry. In our current implementation, we store with each fragment its depth value and its surface normal (needed for diffuse lighting) in camera space as well as an object ID, which is later used to access a small GPU lookup table storing the material colors of the respective object. Since these values are encoded into 2×32 bits, we choose a two component unsigned integer texture format (R32G32) for the k-buffer.

We then use a full-screen rendering pass to ray-cast the volume as well as to simultaneously render the semi-transparent geometry. Our ray-casting approach is based on the technique proposed by Krüger and Westermann [KW03]. For each pixel, we first analytically compute the corresponding ray's entry and exit point into and from the volume. Furthermore, we fetch the pixel's k-buffer entry, i.e., the fragments of the semi-transparent geometry for that pixel, and sort them with respect to ascending camera space depth values. We also fetch the pixel's depth value from the depth buffer and back-project the depth value into camera space. The depth value is used to handle occlusions by opaque objects.

Along the ray, we accumulate the color-contributions of the semi-transparent fragments and of the volume by using front-to-back blending. We first accumulate the semi-transparent fragments that are lying in front of the volume. We then ray-cast the volume by sampling the volume along the ray. The depth of the next sampling position is determined as the minimum of the depth of the previous sampling position plus the depth increment corresponding to the given step size (which is chosen as half the voxel size), and the depth of the next semi-transparent fragment. The semi-transparent geometry is incorporated into the ray-casting process by blending a fragment when the depth of the fragment is equal to the depth of the current sampling position. After ray-casting the volume, we accumulate the semi-transparent fragments that are lying behind the volume. Whenever the ray hits the opaque geometry, it is terminated. The accumulated color is finally written into the frame buffer by using alpha blending with the opaque geometry.

For rendering the von Mises stress on a set of parallel, axial slices through the bone, we proceed analogously. However, we sample the volume only at the intersection points of the ray with the slices. The respective sampling positions are determined by using the analytical description of the slices and the ray. The von Mises stress values are mapped onto colors by means of a rainbow color map.

Due to the hexahedral finite element discretization of the bone, the von Mises stress field slightly sticks out from the smooth bone surface. To achieve a more appealing visualization of the von Mises stress volume, we clip its rendering at the bone surface, i.e., we only sample the von Mises stress volume when the sampling position is lying in the interior of the bone mesh. This is determined by using a boolean flag during ray-casting that is toggled whenever a bone surface fragment is encountered. The slices are clipped analogously.

5.6 System Integration

To enable a smooth interaction with the system, we use two separate threads, one for the simulation (referred to as the simulation thread), and one for the visualization and the handling of the user input (referred to as the render thread). Each thread operates on a thread-local set of buffers storing the simulation parameters and the simulation results. The simulation parameters include the implant position, the cutting planes, and the forces that are applied to the bone and the implant. The simulation results comprise the per-vertex displacement vectors, as well as the Young's modulus values of the finite elements (the latter information is necessary to compute the stresses from the displacements). For the communication between the two threads, another set of buffers, referred to as the intermediate buffers, is used. Accesses to the intermediate buffers must be performed mutually exclusive, i.e., critical section synchronization primitives must be used to prevent simultaneous accesses.

The threads continuously execute a simulation loop and a rendering loop, with each iteration being referred to as a frame. In each simulation frame, the simulation thread 1.) copies the current simulation parameters from the intermediate buffers into his local buffers, 2.) runs the simulation, and 3.) finally copies the new simulation results from his local buffers into the intermediate buffers. In each render frame, the render thread 1.) handles the user input, 2.) copies the current simulation parameters from his local buffers into the intermediate buffers, 3.) if new simulation results are available, copies the new results from the intermediate buffers into his local buffers, and 4.) generates the visualization.

Simulation and visualization are each performed on a separate GPU. To minimize GPU-to-GPU traffic, which has to be performed via PCI Express buses, we only transfer the (densely packed) per-vertex displacement vectors and the (densely packed) Young's modulus values of the finite elements. The von Mises stress field is then directly computed on the visualization GPU by employing a dedicated CUDA kernel, and stored in a 3D texture for visualization (see Section 5.4.4). The latter step is based on the interoperability between CUDA and Direct3D 10, which enables writing into 3D textures from CUDA kernels.

In the following, we describe how the individual simulation and the visualization components described in the previous sections are integrated into the system. In each simulation frame, we first voxelize the implant surface mesh into a 3D texture using our GPU-based voxelization algorithm (see Section 5.4.3). We then compute the Young's modulus values of the finite elements, using the Young's modulus values of the bone voxels (stored in a 3D texture), the voxelization of the implant, and the analytical description of the cutting planes specifying the part of the femoral head that is to be removed (see Sections 5.4.1 and 5.4.2). The Young's modulus values of the finite elements are downloaded to the CPU, to be later transferred to the visualization GPU for the computation of the stresses.

We then update the per-vertex equations on the simulation level, as well as on the coarse grids of the multigrid hierarchy. To reassemble only those per-vertex equations which are affected by the changes of the Young's modulus values, we employ a boolean flag at each vertex that specifies whether the respective equation needs to be reassembled (a 'changed' vertex) or not. On the simulation level, the 'changed' vertices are exactly those vertices which are incident to a hexahedral finite element with changed Young's modulus. On the coarse grid levels, the 'changed' vertices are exactly those vertices which interpolate from a 'changed' vertex on the previous finer level. The flags are successively propagated from the finest to the coarsest level. Note that the voxelization of the implant and the update of the equations are completely skipped if the implant and cutting plane configuration has not changed with respect to the previous simulation frame.

We then upload the per-vertex forces to the GPU, which constitute the right hand side of the underlying linear system of equations, and approximately solve this system by performing two multigrid V-cycles, each with two pre-smoothing and one post-smoothing decoupled Gauss-Seidel relaxation steps ($\omega = 1$) (see Chapter 3). On the coarsest level, we employ the Cholesky solver from the TAUCS library [TCRM03]. Even though this solver is running on the CPU—which requires to download the coarsest level linear system of equations to the CPU, as well as to upload its solution back to the GPU—our experiments have shown that for the considered application this strategy is faster than employing a conjugate gradient solver running entirely on the GPU. The reason is that due to the inhomogeneous material properties of the bone, a high number of CG cycles is required to achieve a sufficient correction from the coarsest grid. The computed per-vertex displacement vectors are downloaded to the CPU, to be later uploaded to the visualization GPU.

The iterative nature of the employed multigrid solver enables providing a fast and reasonably accurate feedback in a very short time interval, which is crucial in the context of a computational steering environment. Since we use the results from the previous simulation frame as initial values for the current frame, the multigrid solver is successively improving the solution with each further frame when the simulation parameters remain unchanged. After 4 to 8 V-cycles, the solution is already visually indistinguishable from the exact solution, as we will demonstrate in the following section.

For the visualization, the per-vertex displacements and the Young's modulus values of the finite elements are uploaded to the visualization GPU. We then compute the von Mises stress scalar field, which is stored in a 3D texture to enable GPU-based trilinear interpolation in this field (see Section 5.4.4). The von Mises stress field is finally visualized using the techniques described in Section 5.5.

5.7 Results

In this section, we give a detailed analysis of the performance and accuracy of our surgery support system. All benchmarks were run on a high-end workstation, equipped with two quad core Intel Xeon X5560 processors running at 2.8 GHz, 48 GB of DDR3 1333 MHz RAM, and two NVIDIA Tesla C2070 graphics and computing cards, each with 6 GB of video memory.

Figures 5.7 to 5.10 show the simulation results obtained by our virtual 3D planning system for a real-world scenario. The finite element model of the femur has been derived from a clinical CT scan of the patient. In the figures, we show both the physiological stress distribution in the intact femur, as well as the stress distribution after insertion of a specific implant. In particular, we simulate the stress distribution for a classical G2 implant with a long stem, as well as for a modern CUT implant with a short stem. For the latter implant, we perform the simulation using both a straight as

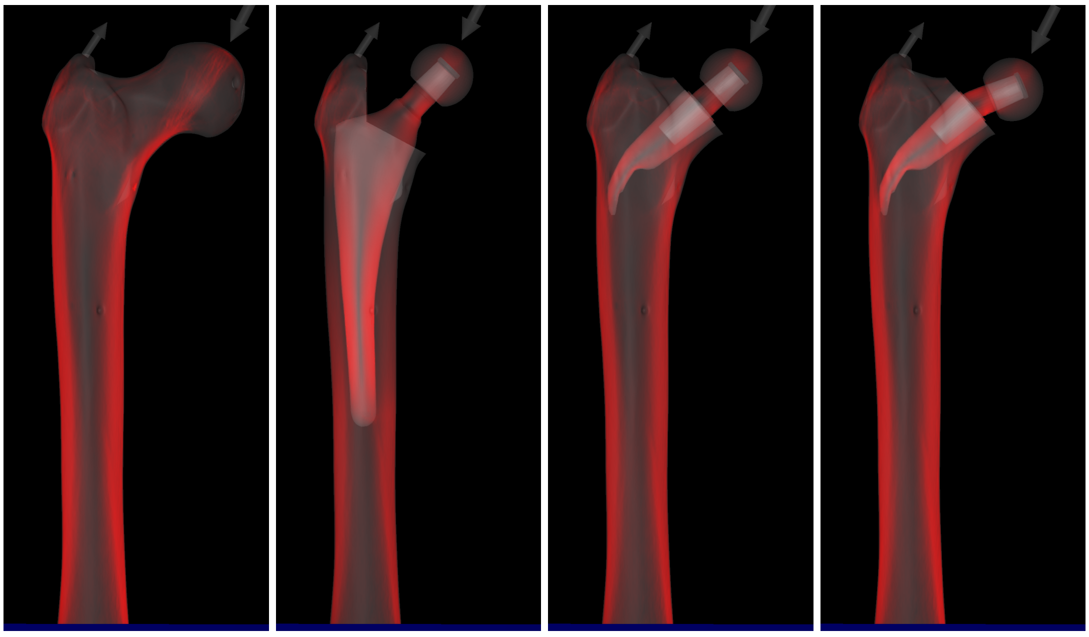


Figure 5.7: *From left to right: Volume rendering of the simulated von Mises stresses in the intact bone, and after the insertion of a G2 implant, a CUT implant with straight conus, and a CUT implant with angled conus, respectively. Simulation was performed at a resolution of one hexahedral finite element per CT voxel.*

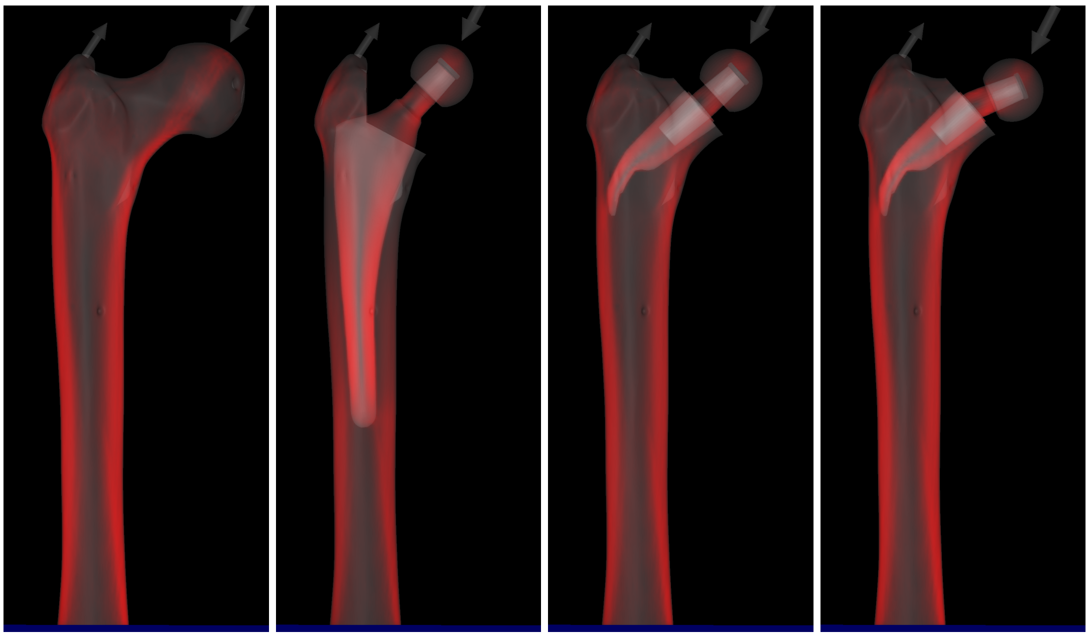


Figure 5.8: *Volume rendering of the simulated von Mises stresses (setup analogous to Figure 5.7). Simulation was performed at a resolution of one hexahedral finite element per 2^3 CT voxels.*

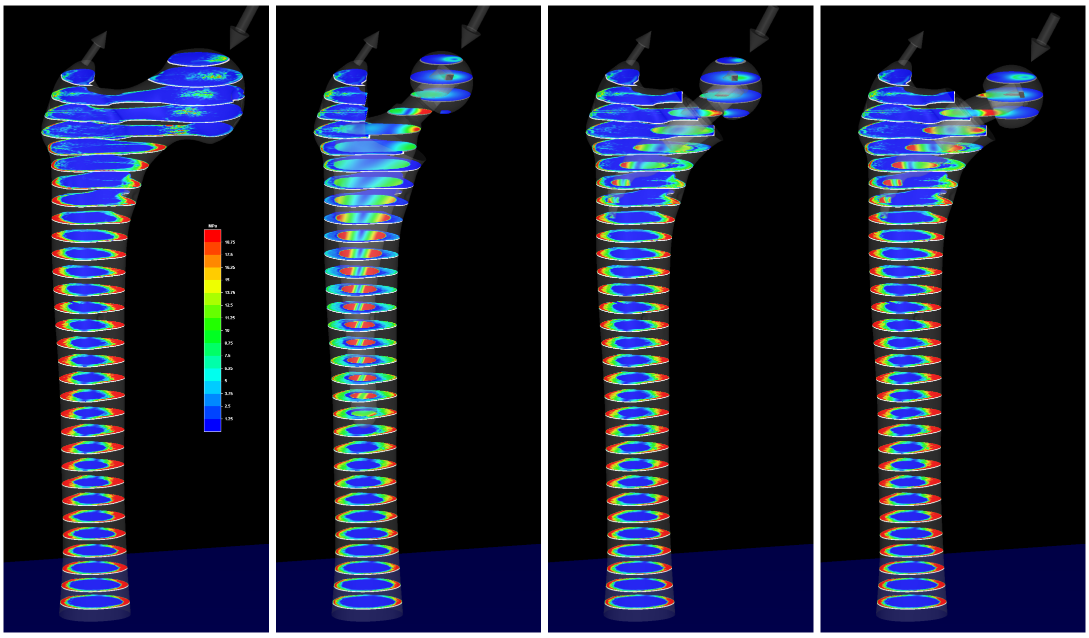


Figure 5.9: *Color coding of simulated von Mises stresses on a set of parallel, axial slices (setup analogous to Figure 5.7). Simulation was performed at a resolution of one hexahedral finite element per CT voxel.*

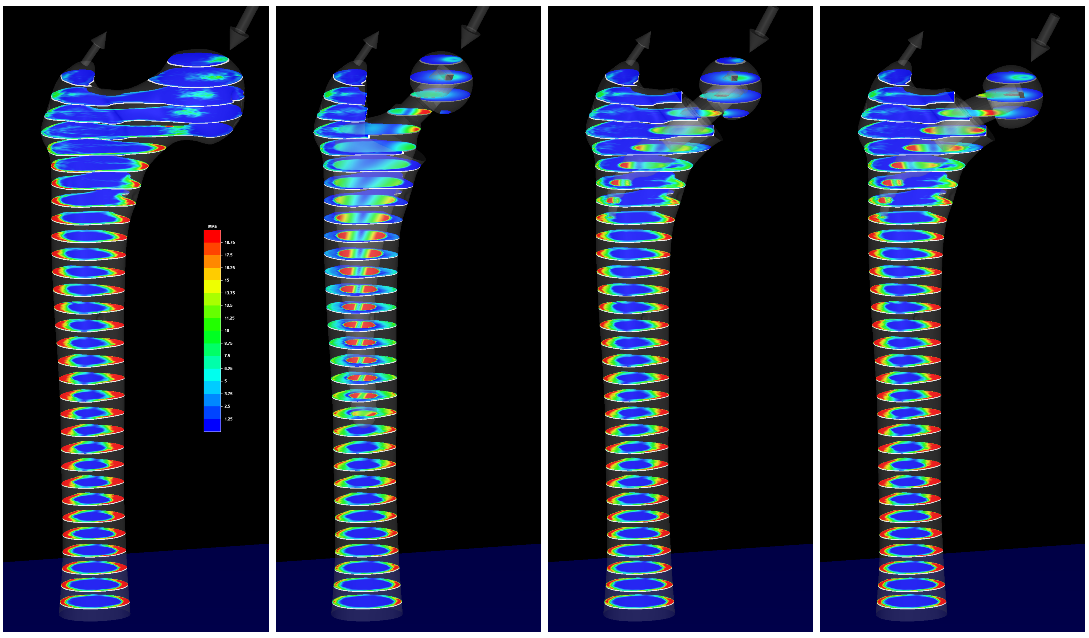


Figure 5.10: *Color coding of simulated von Mises stresses on a set of parallel, axial slices (setup analogous to Figure 5.7). Simulation was performed at a resolution of one hexahedral finite element per 2^3 CT voxels.*

well as an angled conus. The modular assembly of the implant (consisting of stem, conus, and ball) enables to accurately reproduce the hip joint rotation center.

In our simulation, we apply two force vectors to the bone and the implant. These force vectors represent the accumulated muscular force of the hip abductors acting at the trochanter major, and the contact force in the hip joint resulting from the muscular forces of the abductors and the body weight. In the figures, we have applied a joint contact force of 1500 N and a muscular force of 1125 N, corresponding to the load situation of a healthy hip joint in the standing position for a body mass of 75 kg (muscular force : joint contact force : half body weight = 3 : 4 : 1 [Mül71]). In addition to the manual specification of the forces, we provide the option to apply the specific forces occurring over an entire motion cycle, such as of normal walking or stair climbing (see Figures 6.8 and 6.9 on pages 148 and 149 for an example). The respective contact forces were measured in-vivo by Bergmann et al. [BDH+01] using instrumented implants, and the respective muscular forces were computed by Heller et al. [HBD+01] using a musculoskeletal model of the human lower extremity. The data were published by Bergmann on the HIP 98 CD [Ber01].

In the figures, we show the simulation results for two different resolutions of the finite element model, i.e., one hexahedron per CT voxel, and one hexahedron per 2^3 CT voxels. The figures demonstrate a high agreement between the stress distributions obtained for the two resolutions. The finer resolution model even resolves the trabecular structures in the bone, which become indirectly visible in the stress patterns. When comparing the stress patterns in the intact bone, and those resulting from the insertion of a specific implant, the figures show that the insertion of the classical long-stemmed G2 implant leads to a significant reduction of the stresses in the region around the implant stem. These results are confirmed by X-rays showing the postoperative long-term change of bone density for this implant type (see Figure 5.2). There, a significant degeneration of bone tissue in exactly these regions due to stress shielding can be observed. The figures also clearly demonstrate that the stress distribution resulting from the insertion of a modern short-stemmed CUT implant much better resembles the physiological stress state.

Table 5.1 shows the performance of our simulation support system for the two finite element model resolutions (first row). The second, third, and fourth row list the number of hexahedral elements, the number of vertices, as well as the number of multigrid levels for the respective FE model resolution. The next three rows give the time for a single simulation frame, split into the time for the voxelization of the implant, the time for the assembly of *all* per-vertex equations on the simulation level and the coarse

CT Voxels per Hexahedron		2^3	1^3
#Hexahedra		97,700	734,000
#Vertices		110,000	782,000
#MG Levels		5	6
Voxelize Implant	[ms]	0.872	0.872
Assemble Equations	[ms]	41.7	266
Solve	[ms]	32.7	208
Total	[ms]	75.2	475
Simulation Update Rate	[1/s]	13.3	2.10

Table 5.1: *Timing statistics of our surgery support system. Note that these timings correspond to the worst-case in that we measure the time to reassemble* all *per-vertex equations, even though the majority of equations are unchanged with respect to the previous simulation cycle. In particular, if the implant configuration has not changed with respect to the previous simulation cycle, the implant voxelization and the equation assembly step are completely skipped.*

grids of the multigrid hierarchy (this time includes the computation of the Young's modulus values of the finite elements, and the download of these values to the CPU), as well as the time to approximately solve the underlying linear system of equations by performing two multigrid V-cycles (this time includes the upload of the per-vertex force vectors to the GPU, as well as the download of the per-vertex displacement vectors to the CPU). The time for the implant voxelization (22,500 triangles) is the same for both finite element model resolutions, since the voxelization is always performed at the resolution of the CT voxel grid. The following two rows contain the total time of a single simulation frame, as well as the number of simulation frames per second.

Note that the specified simulation frame times and numbers of simulation frames per second correspond to the worst case. Due to our optimized implementation, in each frame only a small number of per-vertex equations has to be reassembled in practice. In particular, if the implant configuration has not changed with respect to the previous frame, for example during interactive monitoring of the change of stresses due to different loading conditions, the implant voxelization and the update of equation step are completely skipped.

With update rates of more than 13 simulation frames per second at half CT resolution, and still more than 2 frames per second at CT resolution, fast response times are guaranteed, which enables an interactive, visually-guided steering of the simulation.

Since we use two separate threads for the simulation and the visualization, the rendering frame rate is decoupled from the simulation frame rate, which enables a smooth interaction with the system even for the highest finite element model resolution. On a 1920×1200 view port, the visualization is running at a frame rate of more than 60 fps.

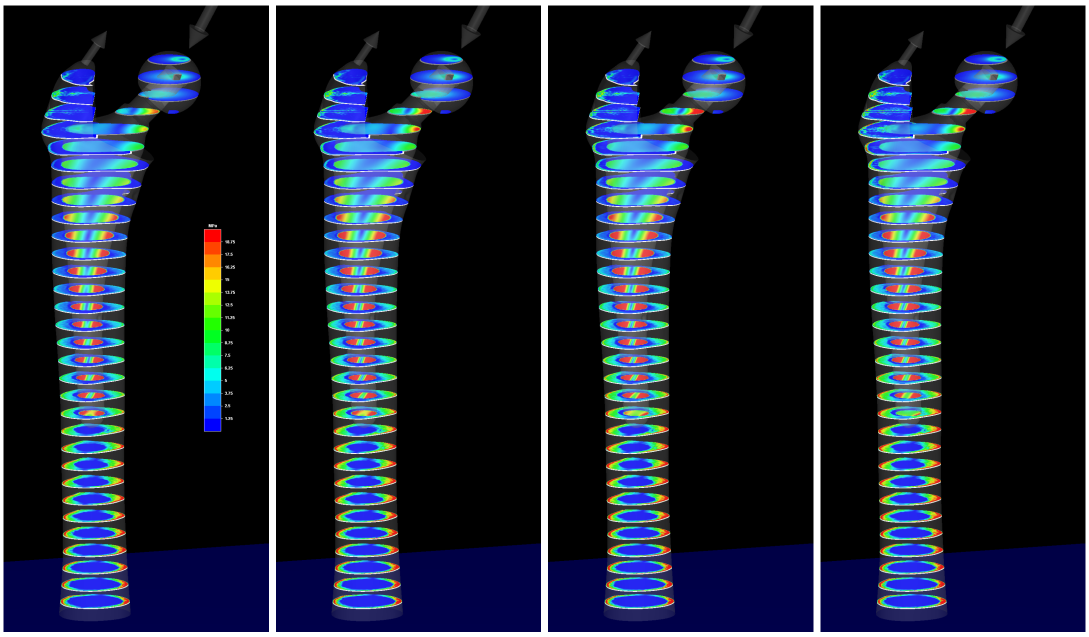


Figure 5.11: *Multigrid convergence study, using a resolution of one hexahedral finite element per CT voxel, and starting with an initial value of* $\mathbf{0}$ *for the per-vertex displacement vectors. From left to right, the figures show the simulated von Mises stresses after 2, 4, and 8 V-cycles, and after full convergence of the solver.*

In Figure 5.11 we finally demonstrate the convergence behavior of our multigrid solver for the particular application. For this experiment, we start with an initial value of $\mathbf{0}$ for the per-vertex displacement vectors $\boldsymbol{u}_k$, and show the von Mises stress distribution after 2, 4, and 8 V-cycles, as well as the solution after the solver has reached convergence. The figure demonstrates that the solution after 2 V-cycles already is in close visual agreement with the fully converged solution, and that the solution after 4 to 8 V-cycles is virtually indistinguishable from this solution.

5.8 Conclusion and Future Work

In this chapter, we have presented the first computational steering environment for optimal implant selection and positioning. Our results demonstrate that by using advanced numerical schemes for finite element analysis, interactive yet highly accurate simulations are possible today on desktop PC systems. Combined with efficient visualization schemes including surface and volume rendering, a powerful visual computing tool for implant planning in orthopedics has been developed.

In the future, we will validate the results of our simulation with respect to real-world experiments. Further directions of research are the integration of anisotropic material laws into the simulation, with the directions of anisotropy being derived from the trabecular structures in a high-resolution CT scan, as well as the explicit modeling of the contact zone between implant and bone.

Chapter 6

Stress Tensor Field Visualization for Implant Planning in Orthopedics

We demonstrate the application of advanced 3D visualization techniques to determine the optimal implant design and position in hip joint replacement planning. Our methods take as input the physiological stress distribution inside a patient's bone under load and the stress distribution inside this bone under the same load after a simulated replacement surgery. The visualization aims at showing principal stress directions and magnitudes, as well as differences in both distributions. By visualizing changes of normal and shear stresses with respect to the principal stress directions of the physiological state, a comparative analysis of the physiological stress distribution and the stress distribution with implant is provided, and the implant parameters that minimize stress shielding, i.e., most closely replicate the physiological stress state, can be determined. Our method combines volume rendering for the visualization of stress magnitudes with the tracing of short line segments for the visualization of stress directions. To improve depth perception, semi-transparent, shaded, and antialiased lines are rendered in correct visibility order, and they are attenuated by the volume rendering. We use a focus+context approach to visually guide the user to relevant regions in the data, and to support a detailed stress analysis in these regions while preserving spatial context information. Since all of our techniques have been realized on the GPU, they can immediately react to changes in the simulated stress tensor field and thus provide an effective means for optimal implant selection and positioning in a computational steering environment.

This chapter is based on material that has been originally published in C. Dick, J. Georgii, R. Burgkart, and R. Westermann, *Stress tensor field visualization for implant planning in orthopedics*, IEEE Transactions on Visualization and Computer Graphics 15 (2009), no. 6, 1399–1406 [DGBW09b]. The copyright for this material is owned by the IEEE.

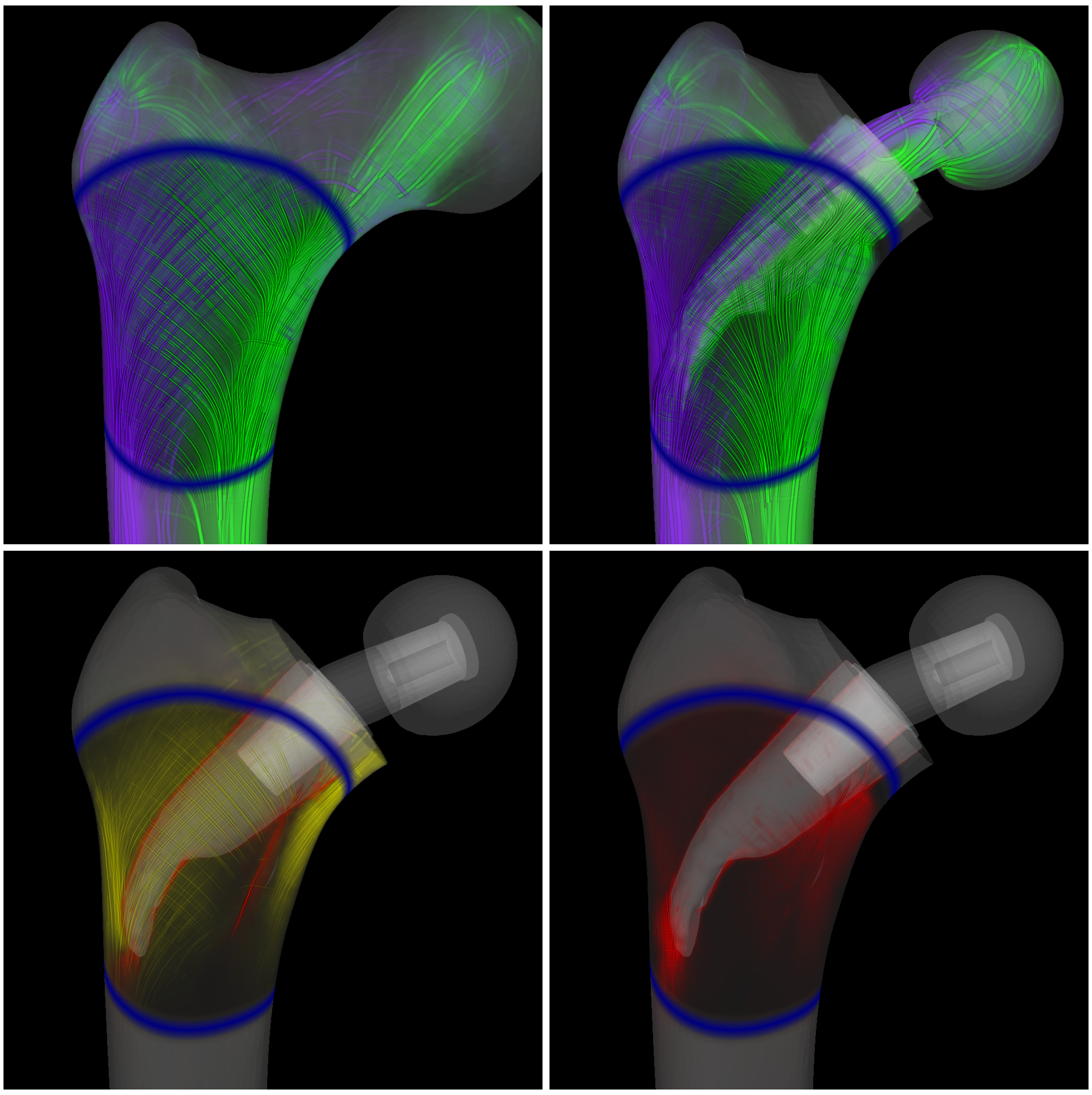


Figure 6.1: *Interactive visualizations ($<$ 50 ms) of simulated stress tensor fields for a human femur under load, using our methods. Top left: Principal stress directions and magnitudes in the physiological state (violet = tension, green = compression). Top right: Principal stresses after a simulated implant surgery. Bottom left: Change of normal stresses with respect to the principal stress directions of the physiological state (red = increase, yellow = decrease). Bottom right: Change of shear stresses.*

6.1 Introduction

3D simulation and visualization methods for in-vivo bone stresses are of great importance in hip joint replacement planning, since they support the surgeon in finding the optimal shape, size, and position of an implant during a preoperative design loop. This is due to the fact that an essential factor for the long-term stability of an implant is the load transmission to the adjacent bone stock. In particular, an optimized femoral stem should minimize stress shielding, i.e., provide bone stress patterns that are close to the physiological stress state, in order to avoid a degeneration of bone tissue with the consecutive effects of osteopenia, fracture and aseptic loosening [BWW+08, KOO+09, OH78, RSG+08, SV02].

Physically-based simulation of the mechanical response of a bone to an applied load (without or with an inserted implant) has been addressed in a number of research papers [BWW+08, BOM+07, TSH+07, YPJM07]. In Chapter 5 we have shown the advantage of real-time simulation, which provides the possibility to obtain immediate feedback to changes of the implant shape, size, and position as well as the exerted forces in the context of a computational steering environment. So far, these approaches visualize simulation results only by rendering scalar stress tensor norms on surfaces, or by using volume rendering of the respective scalar fields (see Figure 6.3). However, for medical purposes this kind of visualization is limited by the fact that it neglects important directional information in the simulated stress tensor fields. Furthermore, since an optimal implant should provide a bone stress distribution close to the physiological stress state, the surgeon needs a comparative visualization of the two stress distributions. To the best of our knowledge, such a comparative visual analysis of the stress patterns resulting from a simulated implant surgery to the physiological stress state has not yet been reported in the literature.

6.2 Contribution

To the best of our knowledge, we present the first interactive visualization approach for dynamically changing 3D stress tensor fields, and we demonstrate the application of this approach to find the most optimal implant shape, size and position in hip joint replacement planning. We use volume rendering to show stress magnitudes, and we combine it with semi-transparent, shaded, and antialiased lines to indicate stress directions. To improve depth perception, these lines are rendered in correct visibility order, attenuated by the volume rendering. Since the proposed visualization techniques run entirely on the GPU, high frame rates can be achieved even for dynamically changing stress tensor fields. Therefore, the techniques provide an effective means for analyzing changing stress patterns as they are simulated in computational steering environments.

The user can flexibly change a number of visualization parameters such as transparency and color to explore the underlying 3D stress tensor field. Furthermore, changes of normal and shear stresses with respect to the principal stress directions of the physiological state can be visualized, thus allowing for an immediate recognition of the regions where stress shielding occurs. To restrict the user's attention to the relevant regions and to enable a precise, yet context-preserving analysis of the stress directions in these regions, a focus+context technique is used. Only in the focus region fine details of the stress directions are shown, while the stress directions in the context region are visualized with lines on a much coarser scale.

A first qualitative evaluation of our visualization techniques shows the importance of the proposed methodologies for preoperative implant selection and positioning. Even though the visualizations differ vastly from the current state-of-the-art, practitioners have indicated an immediate medical benefit from these visualizations.

6.3 Related Work

In this section, we survey related work in the area of the visualization of second-order, symmetric tensor fields, with the most prominent examples of stress/strain as well as diffusion tensor fields. A 3D second-order, symmetric tensor is represented by a symmetric 3×3-matrix (six independent scalar values), and is thus uniquely characterized by its three eigenvalues and three mutually orthogonal eigenvectors. By sorting the three eigenvalues, the corresponding eigenvectors can be classified into the major/medium/minor eigenvector, resulting in three eigenvector fields. Most of the proposed visualization methods make use of the eigendecomposition.

Previous work can be classified into glyph-based approaches, methods originating from vector field visualization based on line/surface tracing or line integral convolution, direct volume rendering techniques, topology-based visualization methods, and approaches based on the visualization of the physical effect of the tensor field on the underlying media.

Glyph based approaches map the tensor field's local properties onto the shape and visual appearance of graphical icons. Simple glyph-based visualization techniques are hedgehogs, which represent the three eigenvectors by short lines, the absolute eigenvalues by the line length, and the sign of the eigenvalue by the line color, or ellipsoids, with half axes aligned to the eigenvectors and scaled according to the (normalized) eigenvalues [LAK$^+$98]. To avoid visual ambiguities, more advanced primitives are used, for example composite shapes [WMM$^+$02] or superquadric tensor glyphs [Kin04].

A broad class of methods adopts approaches from vector field visualization. These approaches can be divided into line and surface tracing methods, and techniques that are based on line integral convolution (LIC). Examples for the first category are hyperstreamlines [DH93], which are traced in the major eigenvector field, and which have an ellipsoidal cross section that is determined by the medium and minor eigenvalues, as well as streamlines, streamtubes and streamsurfaces, which are used in the context of diffusion tensor field visualization [ZDL03, ZB02]. To visualize the differences between two diffusion tensor fields, da Silva et al. [dSZDL01] proposed color-coding of streamlines to visualize the distance between corresponding fibers. To allow for the interactive visualization of static tensor fields based on the tracing approaches, several GPU-based rendering methods were presented. This includes the ray-casting of thin threads splatted into a volume [WKZL04], stress nets, which are limited to 2D tensor fields [WB05], ray-casting of hyperstreamlines [RBE$^+$06], rendering of streamtubes by textured triangle strips [MSE$^+$06], and the rendering of streamtubes combined with a level of detail technique [PFK07]. Kondratieva et al. [KKW05] presented a GPU-based particle tracing method for diffusion tensor fields that even performs the tracing directly on the GPU.

LIC-based approaches are based on the adaptation of the filter kernel parameters [SEHW02] or of the shape of the filter kernel domain [ZP03] to the local tensor field, or they apply the LIC to each eigenvector field separately, and then overlay the resulting images [HFH$^+$04]. Tensor field visualization techniques based on direct volume rendering [BW03, KWH00] use a complex mapping to obtain a color volume from the tensor field. Topology-based approaches [DH94, HLL97, ZPP05] compute and visualize the degenerated points/lines and the separating lines/surfaces of a tensor field. Another method interprets the tensor field as a stress tensor field, and visualizes its deforming effect on the underlying media [BP98, ZP02].

Each of the proposed visualization methods has advantages and disadvantages. Due to the complex nature of a second-order, symmetric 3D tensor field—at each point, three directions and three scalar values have to be visualized—most of the proposed techniques suffer from occlusion and visual cluttering problems, which often reduces their application to the 2D case, and/or they are difficult to interpret, at least for the non-expert.

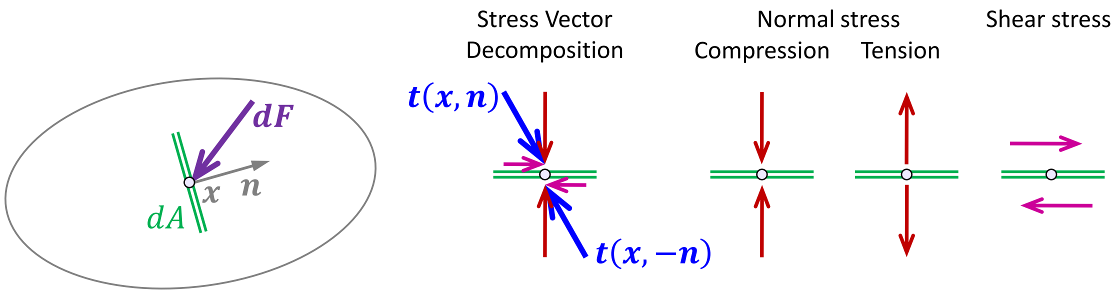


Figure 6.2: *Quantities used in the definition of the stress vector, and illustration of the decomposition of the stress vector into normal and shear stress components.*

6.4 Stress Tensor Field Visualization

Stress is a measure of the internal forces acting within a deformable body. Given a point $\boldsymbol{x}$ as well as a normal vector $\boldsymbol{n}$ specifying the orientation of an infinitely small, *imaginary* area element passing through that point, the stress vector $\boldsymbol{t}(\boldsymbol{x}, \boldsymbol{n})$ is defined as a force density (force per unit area) $\boldsymbol{t}(\boldsymbol{x}, \boldsymbol{n}) = \frac{\boldsymbol{dF}}{dA}$, where $\boldsymbol{dF}$ denotes the force vector that the material on the positive side of the normal exerts on the material on the negative side via the area element dA (see Figure 6.2 and Section 2.1.4).

At each point, the state of stress is fully described by the stress vectors for three mutually orthogonal orientations of the area element. In particular, the stress tensor $\boldsymbol{\sigma}$ contains the stress vectors for the three orientations corresponding to the axes of a Cartesian coordinate system. For an arbitrary orientation of the area element specified by its normal vector $\boldsymbol{n}$, the stress vector is determined by $\boldsymbol{\sigma n}$. This vector can be decomposed into a normal stress and a shear stress component, acting orthogonally and tangentially on the area element, respectively. For each stress tensor, there are three mutually orthogonal orientations of the area element where the shear stress components vanish. For these orientations, the normal stresses are called the *principal stresses* of the stress tensor. Mathematically, the *principal stress magnitudes* are the eigenvalues of the stress tensor, and they are independent of the coordinate system in which the stress tensor is given. The associated eigenvectors are the *principal stress directions*, and they are given with respect to the current coordinate system. The sign of the principal stress magnitudes classifies the stresses into tension (positive sign) or compression (negative sign). However, since there are three principal stresses acting at each point, the classification is with respect to a specific direction.

Our computational steering environment for implant planning in orthopedics computes the stress tensor field in quasi real-time whenever the exerted external forces or the shape or position of the implant are changed. The stress computation is based on a finite element model consisting of hexahedral finite elements aligned on a 3D Cartesian grid, which is built from patient-specific CT data in a preprocess. For the GPU-based visualization of the stress tensor field, we sample this field at the centers of the finite elements, and store the stress tensor components in a set of 3D textures. These textures are sampled by means of componentwise trilinear interpolation. It is worth noting that in our application the occurring displacements are typically so small that—from a visualization point of view—they do not result in a significant change of the body's shape and thus can be ignored in the visualization.

6.4.1 Principal Stresses

A human femur is made of two different types of bone tissue—compact *cortical bone* forming its outer shell, and spongious *trabecular bone* occupying its interior (see top left image in Figure 6.3). These two different types of bone tissue result from a natural optimization process, which allows the femur to be load-bearing and light-weight at the same time. Bone is living tissue that is subject to a life-long remodeling process, consisting of bone resorption and bone formation, and driven by the mechanical load situation on the bone [Kum05]. For an unchanging load situation, bone resorption and formation are at equilibrium. This equilibrium is disturbed, however, if the mechanical load situation is changed (for example due to bone growth). A larger (smaller) deformation causes an increase of bone formation (resorption), which will decrease (increase) the deformation, until the equilibrium is restored. In this way, the bone adapts itself to changes in the load situation to provide optimal mechanical stability at light weight. As a consequence of this natural adaptation process, the trabeculae of the spongy bone are aligned along the principal stress directions (Wolff's law [Wol92]). This can be clearly seen in the top left and bottom right image of Figure 6.3, which show a cut through a real femur and the visualization of the principal stresses in a simulated physiological stress distribution, respectively. Since the trabecular structures—and thus the principal stress directions in the physiological state—are well-known by orthopedic surgeons from their education and daily work with X-rays (see top right image in Figure 6.3, which is taken from medical literature and which shows a 2D sketch of the principal stress directions in a human femur), visualizing principal stresses gives the surgeon an intuitive way to judge the stress distribution resulting from a simulated implant surgery. In particular, this approach is much more intuitive than previous approaches based on the scalar von Mises norm [Bat02], which is used in the bottom left image of Figure 6.3.

Following these ideas, we combine volume ray-casting for the visualization of stress

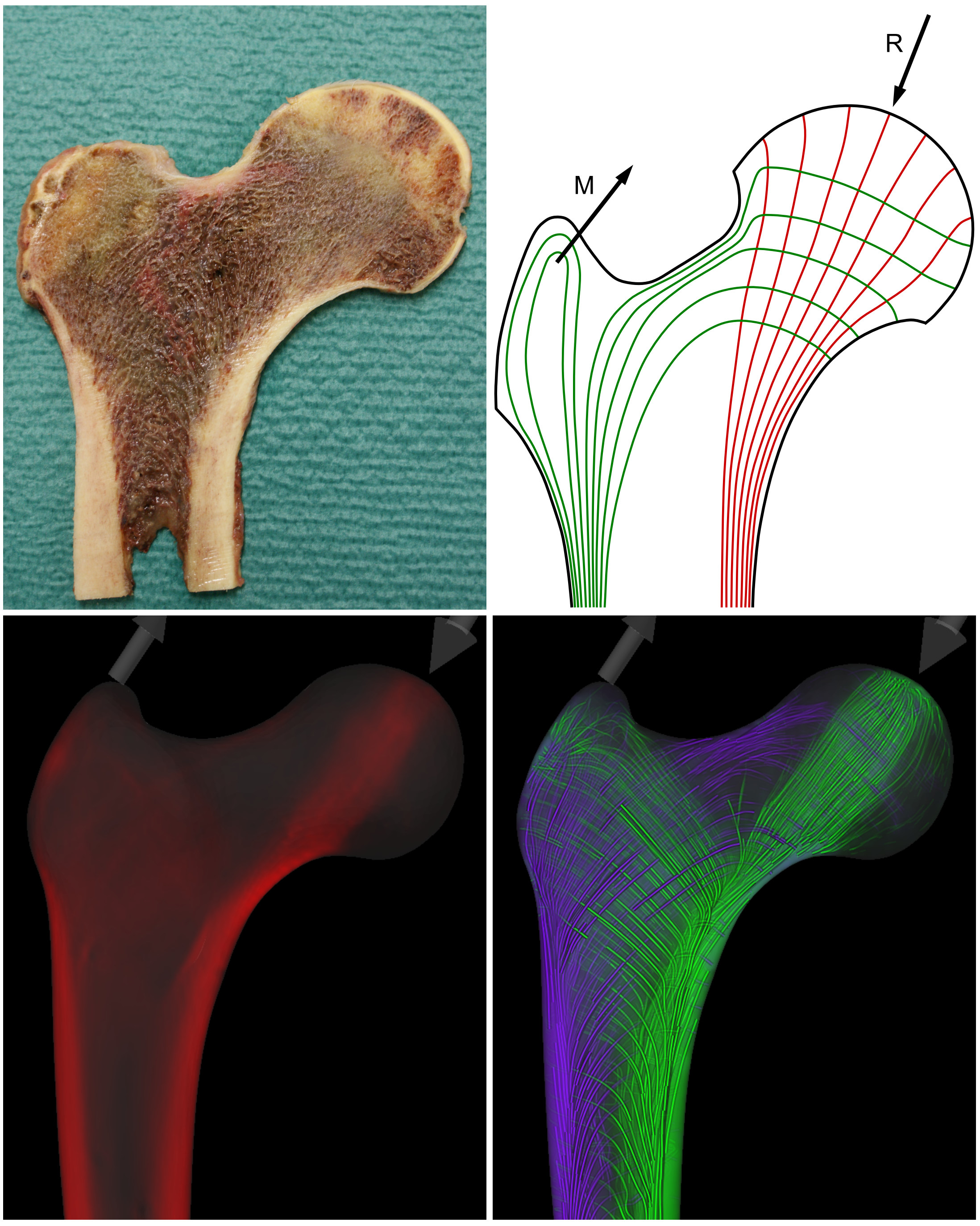


Figure 6.3: *Top left: Cross section of a human femur showing the cortical and trabecular structures. Top right: Schematic overview of the principal stress directions in 2D (according to Pauwels [Pau73]). Bottom left: 3D volume rendering of the scalar von Mises stress norm. Bottom right: Visualization of the principal stresses using the method proposed in this work.*

magnitudes, color-coded according to tension and compression, with the rendering of semi-transparent lines for the visualization of stress directions.

For the volume rendering of the principal stress magnitudes, we use the following mapping to assign a color to the stress tensor at a given point. First, we compute the tensor's eigenvalues and eigenvectors. We then determine for each principal stress magnitude $\sigma_i,\ i \in \{1, 2, 3\}$ a color contribution (RGB_i, α_i) as follows (using non-associated colors): The color RGB_i is determined by the classification into tension (violet) and compression (green), and the respective opacity α_i is proportional to the absolute stress magnitude, i.e.,

$$RGB_i = \begin{cases} \text{violet} & \text{if } \sigma_i \geq 0 \\ \text{green} & \text{if } \sigma_i < 0 \end{cases}, \qquad \alpha_i = \operatorname{saturate}\left(c \cdot |\sigma_i|\right), \tag{6.1}$$

where c is a user-specified scaling factor. In this way, regions with low stresses are almost transparent and do not occlude regions with high stresses. Since stress magnitude is finally represented by the brightness of the associated color $\alpha_i \cdot RGB_i$, the violet and the green base color are selected such that they have the same luminance. The three color contributions at a point are accumulated to get the final color (RGB, α) for the point according to

$$RGB = \frac{\sum_{i=1}^{3} \alpha_i \cdot RGB_i}{\sum_{i=1}^{3} \alpha_i}, \qquad \alpha = \operatorname{saturate}\left(\sum_{i=1}^{3} \alpha_i\right). \tag{6.2}$$

To add directional information to the visualization, we trace lines along the three eigenvector fields, with six traces originating from each seed-point (two for each principal stress direction and its opposite direction). We choose the seed-points on a regular Cartesian grid that is restricted by the bone surface, and we slightly jitter each seed position. To accurately integrate the lines of principal stress, we apply a fourth-order Runge-Kutta scheme with fixed step size. During tracing, the eigenvector-decomposition is computed on-the-fly for each sample point. To ensure directional consistency within a trace (note that the eigenvector direction is not uniquely determined), we flip computed eigenvectors if the angle to the current line direction is larger than $90°$. We restrict the length of the traces to a user-specified limit, since the principal stress directions are a local property of the tensor field. The line is colored according to the principal stress magnitude in the trace direction, using the same color mapping as is used for the volume rendering. In this way, stress lines representing low stresses are almost transparent and thus do not occlude stress lines with high stresses.

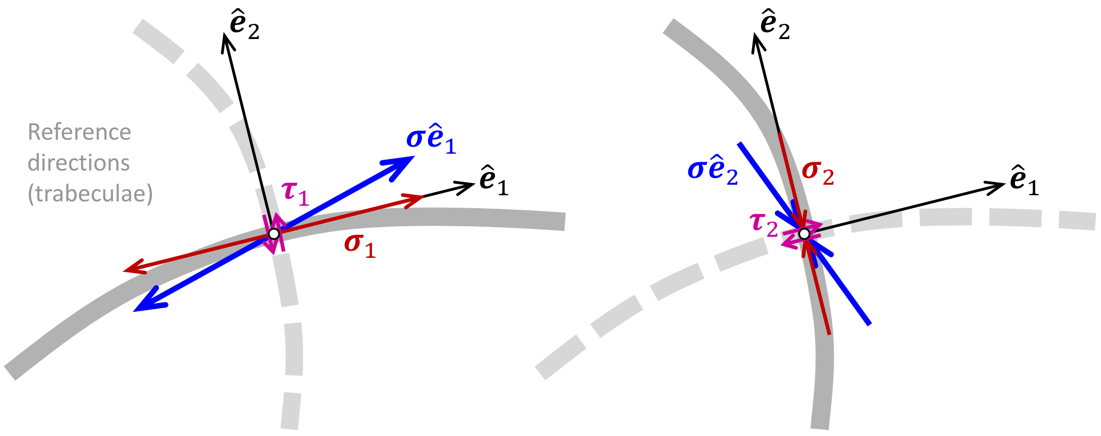


Figure 6.4: *Decomposition of the stress tensor $\boldsymbol{\sigma}$ with respect to the reference directions $\widehat{e}_1$ (left) and $\widehat{e}_2$ (right) into normal stresses σ_1 and σ_2 and shear stresses τ_1 and τ_2. The figure shows the 2D case for simplicity.*

6.4.2 Comparison to Physiological Stress State

One major problem in implant surgery is the change of the bone stress distribution due to the insertion of the implant. In particular, the stiffening of the bone by the implant as well as an unphysiological load transmission from the implant to the bone lead to a removal of stress from certain regions in the bone, which is called *stress shielding*. Due to the bone's adaptation to changed stress patterns, stress shielding causes degeneration of bone tissue, which finally may lead to fracture or loosening of the implant. Paradoxically, degeneration of bone tissue also occurs when the stress exceeds a certain critical stress magnitude [Kum05]. Therefore, the ultimate goal of the medical procedure is to keep the stress distribution after insertion of an implant as close as possible to the physiological state. We address this requirement by providing a novel method to visualize differences between the simulated stress tensor field resulting from a virtual implant surgery and the physiological stress distribution. Our idea is to use the directions of the trabecular structures—which correspond to the principal stress directions in the physiological state—as a reference frame at each point for decomposing the stress tensors of both fields into normal stress and shear stress components, and then to visualize differences with respect to these components. It is worth noting that by using these reference frames, in the physiological state the normal stresses are principal stresses and the shear stresses vanish, corresponding to an optimal loading of the trabecular structures.

As a first step, our approach requires a model of the trabeculae. While in principle the main trabeculae are visible in high-resolution CT scans, we choose a model-driven approximation here. We define the reference directions by means of the principal stress directions that arise from the default loading of the intact bone (standing position). This model can be obtained by using the available simulation back end. It is worth noting that the intent of the proposed comparative visualization method is to show changes of normal and shear stresses *with respect to the trabecular directions*. Since the visualization's physical significance thus depends on the accuracy of the reference frames, let us mention here that we could employ any other, more accurate trabeculae model, too.

Using the trabeculae model, we can decompose any stress tensor $\boldsymbol{\sigma}$ at a given sample point with respect to the local reference frame $\widehat{\boldsymbol{e}}_i,\ i \in \{1,2,3\}$, i.e., the trabecular directions, yielding a stress vector for each direction as illustrated in Figure 6.4. Each stress vector is then further decomposed into the normal stress component (magnitude σ_i) along the direction $\widehat{\boldsymbol{e}}_i$ and the orthogonal shear stress component (magnitude τ_i) by

$$\sigma_i = \widehat{\boldsymbol{e}}_i^{\mathrm{T}}\, \boldsymbol{\sigma}\, \widehat{\boldsymbol{e}}_i\ , \qquad i \in \{1,2,3\}, \tag{6.3a}$$

$$\tau_i = \|\boldsymbol{\sigma}\, \widehat{\boldsymbol{e}}_i - \sigma_i\, \widehat{\boldsymbol{e}}_i\|_2\ , \qquad i \in \{1,2,3\}. \tag{6.3b}$$

The visualization maps the differences of the absolute normal/shear stress magnitudes onto colors of the respective principal stress lines in the physiological stress distribution, using a similar color mapping as described in the previous section, but with different base colors. Analogously to the visualization of the principal stresses, these lines are attenuated by the volume rendering of the accumulated values for all directions. Since changes of normal/shear stress magnitudes with respect to the physiological stress distribution are highly important for the surgeon, we use highlight colors red and yellow to visualize the increase and decrease of the stress magnitudes, respectively. Due to the semi-transparent rendering of the lines with their alpha value depending on the magnitude of the differences, only those lines are visible which reflect a significant change in the load transmission and thus are of relevance for the surgeon. Since stresses within the implant are not of primary interest in the preoperative planning phase, these stresses can be hidden in the visualization as it is shown in Figure 6.5.

We provide four different visualization modes in order to reduce the information shown to the surgeon at once, which are shown in Figure 6.5:

Tension view: Change of normal stresses that are classified as tension in the physiological state.

Compression view: Change of normal stresses that are classified as compression in the physiological state.

Normal view: Change of normal stresses (tension and compression).

Shear view: Change of shear stresses.

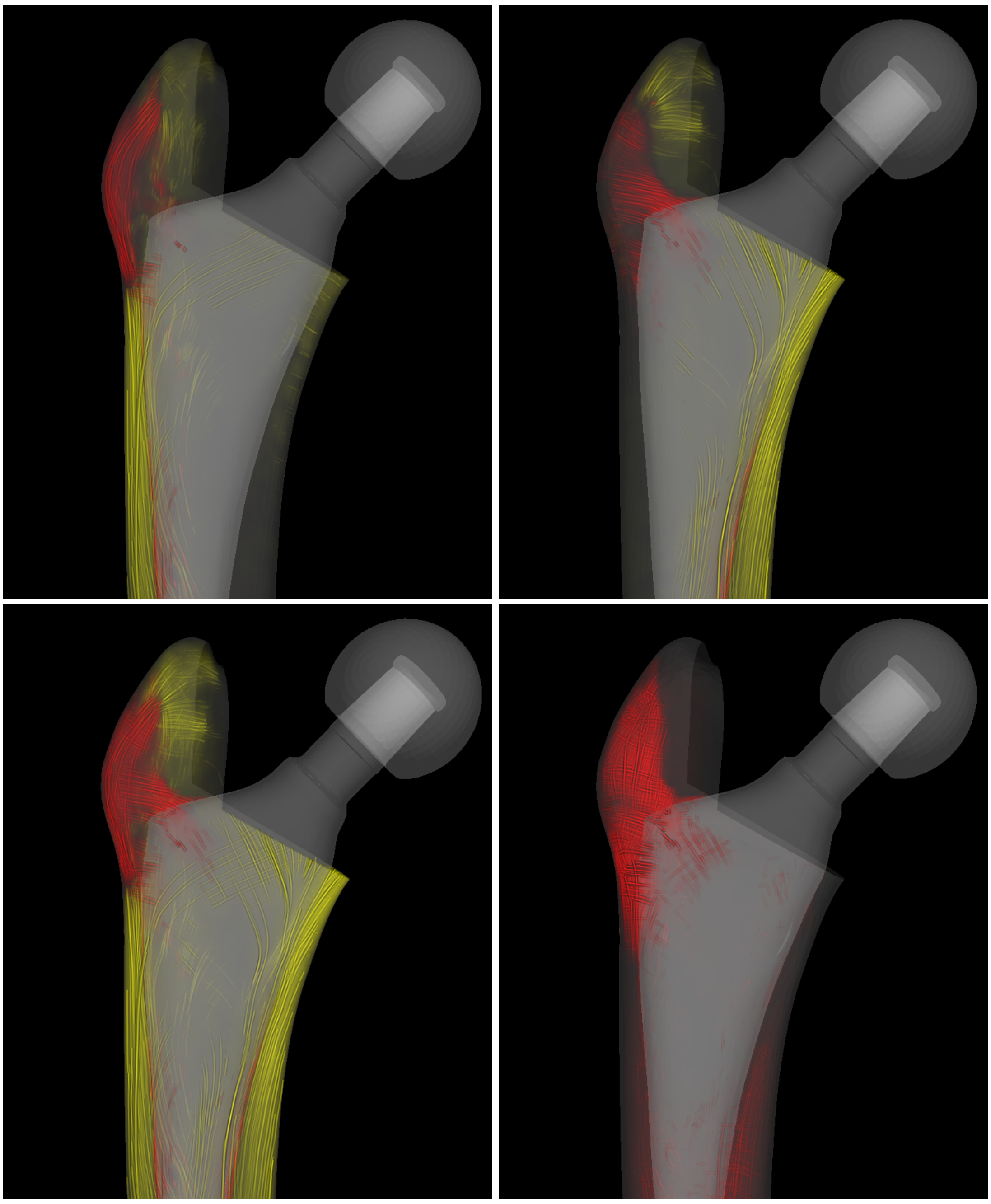


Figure 6.5: *Different options for the comparative visualization of the stress distribution after a simulated implant surgery and the physiological stress distribution (red = increase, yellow = decrease). Top left: Change of normal stresses, tension only. Top right: Change of normal stresses, compression only. Bottom left: Change of normal stresses, both tension and compression. Bottom right: Change of shear stresses.*

6.5 GPU-Based Implementation

To ensure immediate updates of the visualization whenever visualization parameters or the underlying stress tensor field are changed, we exploit computational, bandwidth, and rendering capacities on the GPU for stress tensor field visualization. In combination with a finite element simulation method that provides interactive update rates, the GPU implementation enables the surgeon to interactively explore the effects of different implant shapes, sizes, and positions as well as varying loads on the stress distribution inside the patient-specific bone.

6.5.1 Eigendecomposition

As described in Section 6.4.1, our visualization method requires the computation of the principal stresses of a given stress tensor, i.e., its eigenvalues and eigenvectors. Our implementation uses the eigensolver algorithm proposed by Hasan et al. [HBPA01], which allows for the analytical computation of the eigenvalues and eigenvectors of a symmetric, positive definite 3×3-matrix. For a GPU implementation, an analytical solver is preferable to an iterative algorithm, since a constant runtime better fits to the GPU's lock-step execution of parallel threads. On the GPU, Hasan et al.'s algorithm has recently been used for eigendecomposition in the context of DT-MRI visualization [KKW05], but in contrast to diffusion tensors, which are positive definite, i.e., which only have positive eigenvalues, stress tensors have eigenvalues of arbitrary sign. Fortunately, from the proof given by Hasan et al. it can be deduced that the algorithm is also applicable to non-positive definite, symmetric 3×3-matrices. In this case, however, the algorithm does not implicitly return the eigenvalues in ascending order so that an explicit sorting of the three eigenvalues is required.

6.5.2 Line Tracing

The GPU-based computation of the stress lines is similar to the particle advection method presented in [KKKW05]. We use a fixed number of vertices per line, as well as a fixed step size. First, the seed-points are computed on the CPU and are uploaded into a buffer on the GPU. Then, starting from the seed-points, the lines are successively traced on the GPU by using a multi-pass "ping-pong" technique, which in each rendering pass simultaneously performs one trace step for each line.

The lines' vertices are stored in two texture sets, which are alternately used as input and output. For each vertex, we store its position, the tangential direction of the stress line at that position (i.e., the eigenvector), the stress magnitude (i.e., the eigenvalue), as well as a number encoding which eigenvector field (major/medium/minor) the respective line follows. Therefore, each texture set consists of two four-component, floating-point textures. The stress tensor field is stored in two three-component, floating point 3D textures. This enables us to directly use the GPU's texture sampling capabilities for trilinear, componentwise interpolation within the tensor field. The eigendecomposition of the interpolated tensor is computed on-the-fly by employing the method described in the previous section.

Tracing lines with a length of N vertices requires N rendering passes. In pass 0, the seed-point of each line is fetched from the buffer and written into texture set 0. In all other passes $i = 1, 2, ..., N - 1$, each line's previous vertex $i - 1$ is fetched from texture set $(i - 1) \bmod 2$, and each line's new vertex i is computed using fourth-order Runge-Kutta integration. The new vertices are stored in texture set $i \bmod 2$.

6.5.3 Rendering

For the GPU-based visualization of the 3D stress tensor field, we combine volume raycasting and semi-transparent line rendering. While the volume rendering is used to simultaneously visualize the three stress magnitudes at each point, the lines are used to visualize selected stress directions as well as the stress magnitudes along these directions. The semi-transparent lines and the volume are rendered in correct visibility order, which considerably improves depth perception, since the lines are attenuated with increasing depth (see Figure 6.6 for a comparison of different rendering techniques).

The correct visibility order is achieved by using the stencil-routed k-buffer proposed by Myers and Bavoil [MB07], in a similar way as it was used in Chapter 5 for the simultaneous rendering of semi-transparent surfaces and volumes. The stencil-routed k-buffer can capture up to eight incoming fragments per pixel in a single rendering pass. It is thus significantly faster than depth-peeling, which can only capture one fragment per pixel in each pass. In our application, four rendering passes, i.e., up to 32 fragments per pixel, are typically sufficient.

The k-buffer consists of a multisampled texture accompanied by a stencil buffer, which is used to route the fragments falling into one pixel to individual subsamples in the texture. To capture more than eight fragments per pixel, the geometry has to be rendered successively into multiple k-buffers using different initial stencil values for each buffer. Since the maximum number of incoming fragments to a pixel, i.e., the number of required rendering passes, is not known a priori, we employ occlusion queries to detect at runtime whether an overflow of the k-buffer has occurred, i.e, whether further ren-

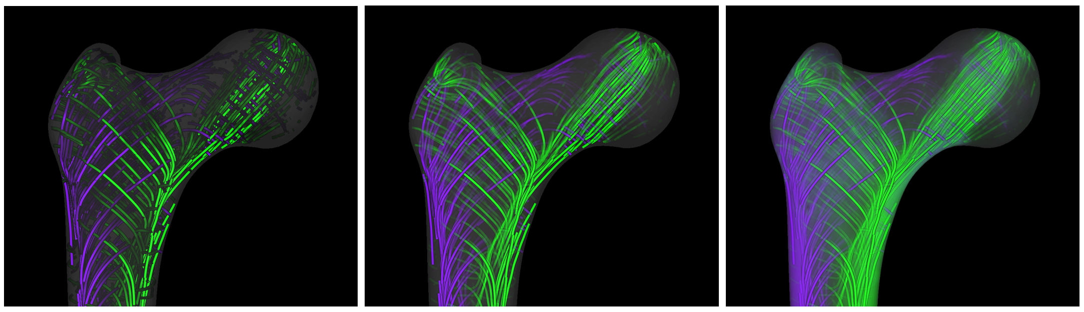


Figure 6.6: *Comparison of different rendering techniques: Left: Opaque lines without antialiasing. Middle: Semi-transparent lines with antialiasing. Right: Semi-transparent antialiased lines with volume attenuation (proposed method).*

dering passes are required. For details, we refer the reader to the original work [MB07]. In our application, where we use the k-buffer to render semi-transparent geometry in correct visibility order, we reduce the maximum number of incoming fragments per pixel by discarding all line fragments with an alpha value < 0.02 during rendering, since these fragments would only have an insignificant contribution to the final image.

The visualization is generated in multiple passes. First, all opaque scene geometry is rendered into the frame buffer, with depth testing being enabled. The depth buffer content is later used during ray-casting to correctly clip semi-transparent geometry and the volume at the opaque geometry. Then, all semi-transparent geometry is rendered into the screen-aligned k-buffers, with depth testing as well as front/back face culling being disabled, so that all fragments are captured. The semi-transparent geometry consists of the bone and the implant surface mesh, as well as the semi-transparent stress lines. For each bone and implant fragment, we store its depth and normal in camera space, as well as an object ID, which is later used to assign material colors. For the lines, we store the depth in camera space as well as the RGBA color of the fragment. All of these values are encoded into 2×32 bits, requiring a two-component unsigned integer texture format.

To render the stress lines, which have been traced using the method described in Section 6.5.2 and stored in two textures, the lines' vertices are fetched from these textures in the vertex shader. We render shaded and antialiased lines to improve perception. This is achieved by employing the geometry shader to expand each line segment into a screen aligned quad, which is conceptually similar to the method proposed by Merhof et al. [MSE+06]. The expansion is performed in screen space, so that the lines are rendered with a specific pixel width. We apply transparency-based antialiasing using a box filter kernel [CD05]. Employing non-associated colors, this is implemented

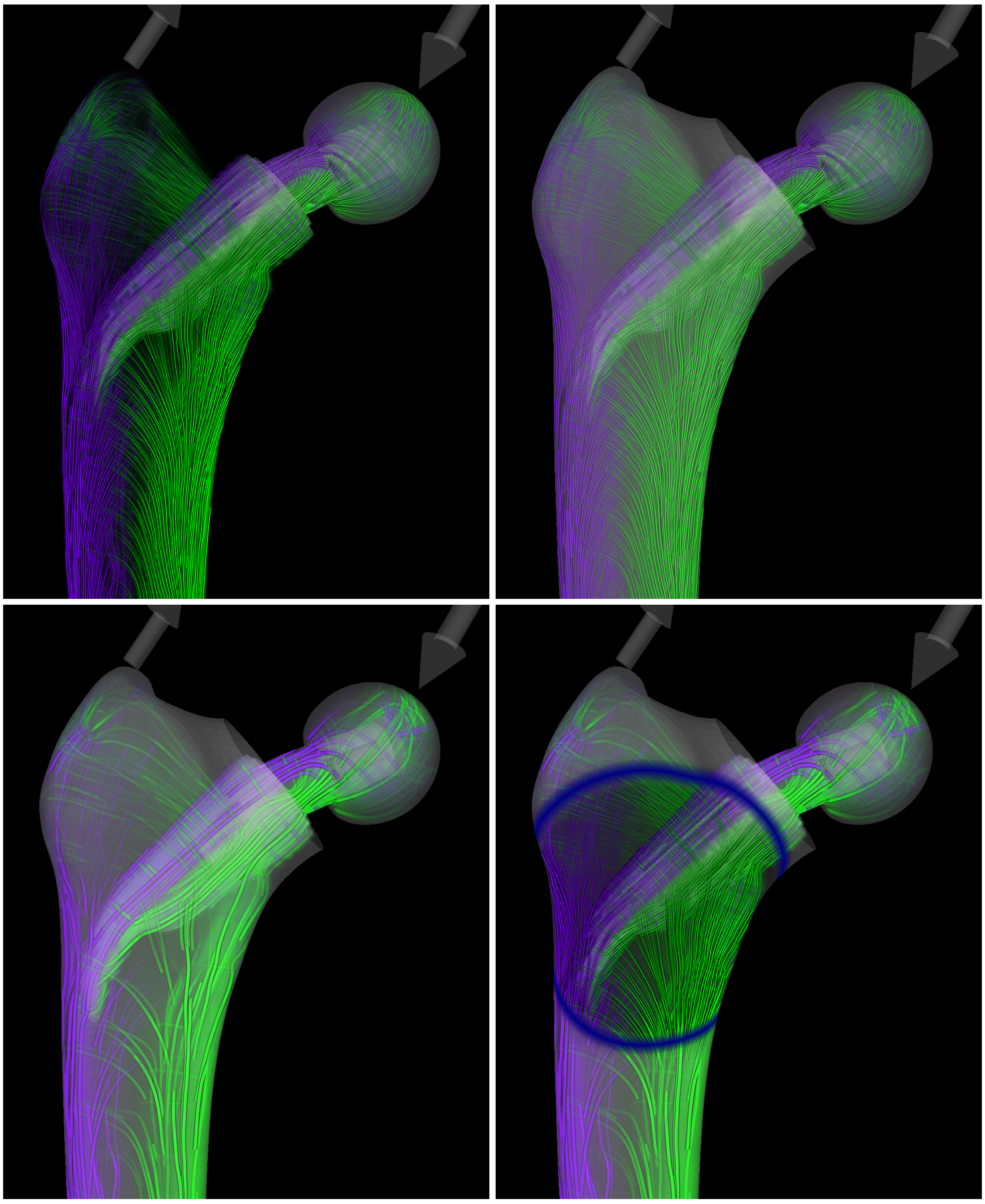


Figure 6.7: *Focus and context visualization of 3D stress tensor fields. Top left: Spatial context is lost if only the stress lines are shown. Top right: Increasing the opacity of the bone's surface improves the spatial perception but washes out the lines. Bottom left: Fewer and thicker lines improve the perception of the global stress distribution but do not allow for a detailed analysis. Bottom right: Focus+context visualization provides the spatial context and the global stress distribution, and allows for a detailed stress analysis in the focus region.*

by multiplying a fragment's alpha value with $\mathrm{saturate}(w - d)$ in the fragment shader, where $w > 1.5$ denotes half the line width and d the distance of the fragment's center from the line's centerline, both values measured in pixels. To enhance the lines' silhouettes, we multiply the fragment's RGB color components by $\mathrm{saturate}\left((w - 1.5) - d\right)$, which introduces thin black borders along the lines.

After all semi-transparent fragments have been captured, we use a full-screen render pass to perform the volume rendering as well as to simultaneously create the final image. For each pixel, in the fragment shader we first fetch the semi-transparent fragments corresponding to that pixel and sort them according to ascending depth. Furthermore, we clip at the opaque geometry by using the pixel's depth buffer value. If the view ray corresponding to the pixel does not intersect the volume, the fragments lying in front of the opaque geometry are blended front-to-back, and the result is blended with the frame buffer content. Otherwise, we ray-cast the volume along the ray and simultaneously incorporate the semi-transparent fragments in correct depth order. At each sample point, two texture fetches are performed to obtain the componentwise interpolated stress tensor, and its eigenvalues are computed to derive the sample's color. This color is accumulated along the ray with the semi-transparent fragments via front-to-back blending. If the ray reaches the depth of the opaque geometry it is terminated, and the accumulated color value is blended with the frame buffer content.

Due to the discrete tensor field voxelization, the volume and the stress lines can appear slightly outside of the bone surface mesh. We therefore improve the visualization by clipping the volume and line rendering at the bone surface. This is implemented by using a flag that specifies whether the current sampling/fragment position is inside or outside the bone surface. The flag is toggled whenever a semi-transparent bone mesh fragment is blended. Using this flag, the color contribution of the volume or a semi-transparent line fragment is only blended if the respective position is lying inside of the bone. The removal of the stress visualization within the implant mesh is implemented analogously.

6.6 Focus+Context

To emphasize important regions in the data and to support a detailed stress analysis in these regions, we employ a lens-like focus+context metaphor as proposed by Krüger et al. [KSW06] for scalar volume rendering. We also use a sphere-shaped focus region that snaps onto the bone surface, but in contrast to showing different structures in the context and the focus region, we show the same structures at different resolutions. In the context region, the density and thickness of the stress lines is decreased and increased, respectively, while the opacity of the bone's surface is increased.

This particular visualization approach is motivated by the observation that a detailed stress analysis requires fine details of the stress directions, which can be achieved by using a high seeding density of the stress lines. Furthermore, the bone's surface has to be completely transparent to optimally reveal these details. By only showing the stress lines, however, spatial perception is significantly reduced, especially if the user navigates around the bone. As a consequence, we increase the surface's opacity in the context region, yielding a considerably improved perception of the spatial relationships. The problem now is that in the context region the blending of the bone's surface with the finely detailed directional structures results in a rather flat visualization showing low contrast and little structural information. To circumvent this drawback we reduce the seeding density of the stress lines in the context region, at the same time making them thicker and thus more distinguishable. By doing so, we provide important spatial context information of the bone as well as the global stress distribution, at the same time enabling a precise analysis of the focus region as demonstrated in Figure 6.7.

Increasing the seeding density of the stress lines is implemented by increasing the spacing of the regular Cartesian grid that is used for the placement of the seed lines as described in Section 6.4.1. Only within the focus region we insert additional seed-points located in the middle between the existing ones. To achieve a visually continuous visualization, i.e., to avoid that lines suddenly pop up or disappear during movements of the focus region, we proceed as follows: The additional seed-points are placed within a sphere region located at the focus region center, but with a radius of the sum of the radius of the focus region plus the trace length. For the stress lines originating from the *original* seed-points, we use a total line width of 10 pixels, with the exception of line segments located within the focus region, for which we use a line width of 6 pixels (with a smooth transition at the border of the focus region). For the stress lines originating from the *additional* seed-points, we always use a total line width of 6 pixels, but for line segments located outside of the focus region, we set the alpha value to 0 (again with a smooth transition at the border of the focus region).

6.7 Results and Evaluation

We have integrated the proposed visualization techniques for 3D stress tensor fields into the virtual implant planning tool presented in Chapter 5, which simulates the stresses in a human femur under load without and with an inserted implant. The tool allows the user to interactively adapt the shape, size, and position of the implant as well as exerted forces.

All images were created in less than 50 ms on a standard desktop PC, equipped with an Intel Core 2 Quad Q9450 2.66 GHz processor, 8 GB of RAM, and an NVIDIA GeForce GTX 280 graphics card with 1024 MB of local video memory. The view port size was 1280×1024. The resolution of the tensor field is $89 \times 83 \times 182$ voxels at a spacing of $1.5\,\text{mm} \times 1.5\,\text{mm} \times 2\,\text{mm}$. The trace length is 40 mm, the step size is 0.5 mm, and the seed points are located on a Cartesian grid with a spacing of 12 mm in the context region and 6 mm in the focus region.

In Figure 6.1, we demonstrate the effectiveness of our visualization techniques for the precise analysis of 3D stress tensor fields. First, we visualize tension (violet) and compression (green) as they arise in a healthy femur under a typical load. Next, we show the same data set after a virtual implant surgery. It is clearly visible how the implant changes the stress distribution in the surrounding bone tissue. To allow for a more precise comparison, we apply our novel comparative visualization technique in the next image. We use colors red and yellow to visualize the increase and decrease of the normal stresses. In this image, the stress shielding in the lower cortical region of the femoral head becomes apparent (yellow region). The last image visualizes the shear stresses that arise mainly around the implant. In addition, in Figures 6.8 and 6.9, we show the principal stresses in the intact bone and after insertion of an implant over an entire motion cycle of stair climbing.

The relevance of the visualization techniques for orthopedic surgeons is demonstrated in Figures 6.10 and 6.11. We show comparative stress visualizations for two implant types and positions. It is clearly visible that the common G2 implants yield to stress shielding in large areas of the femur, especially in the cortical region close to the cut surface. The images demonstrate that modern shorter CUT implants yield a better physiological load transmission. However, these implants are more sensitive to correct positioning. If the implant does not have direct contact to the cortical region, the load transmission exhibits higher stress shielding and higher shear stresses, which increases the risk of fracture or aseptic loosening. These examples demonstrate that our visualization techniques help the surgeon to understand and evaluate the complex mechanical situation arising in hip joint replacement surgery. Consequently, the surgeon can optimize the shape, size, and position of the implant to ensure the best possible long-term prognosis for the patient.

To obtain a feedback on the importance of the proposed methodologies to practitioners and the additional benefits they can derive, we have pursued a first qualitative

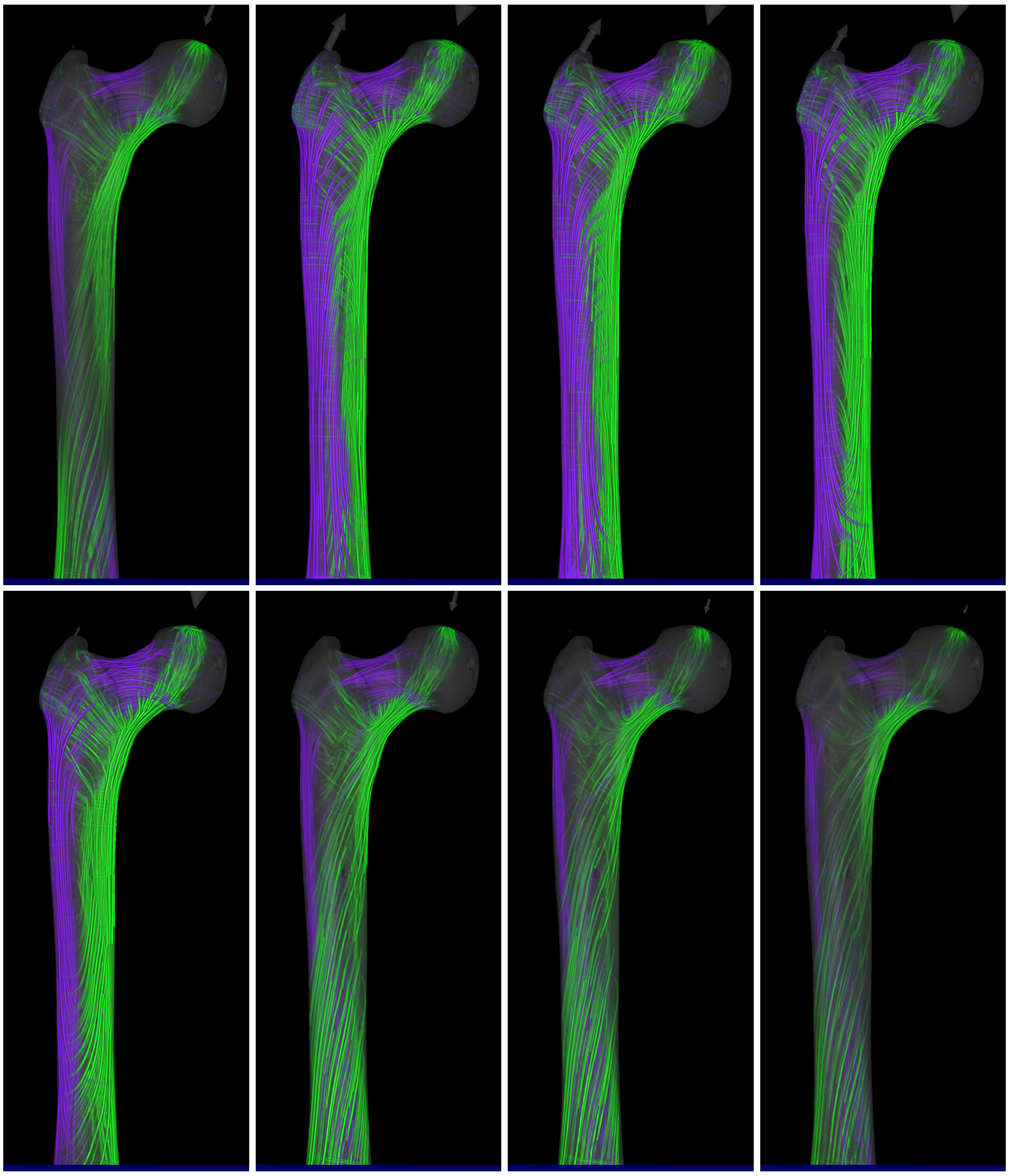


Figure 6.8: *Simulation of the physiological stress distribution in the intact bone during stair climbing. The eight images show an entire motion cycle with a time interval of 0.2 s between two successive images. The muscular and joint contact forces were taken from [Ber01].*

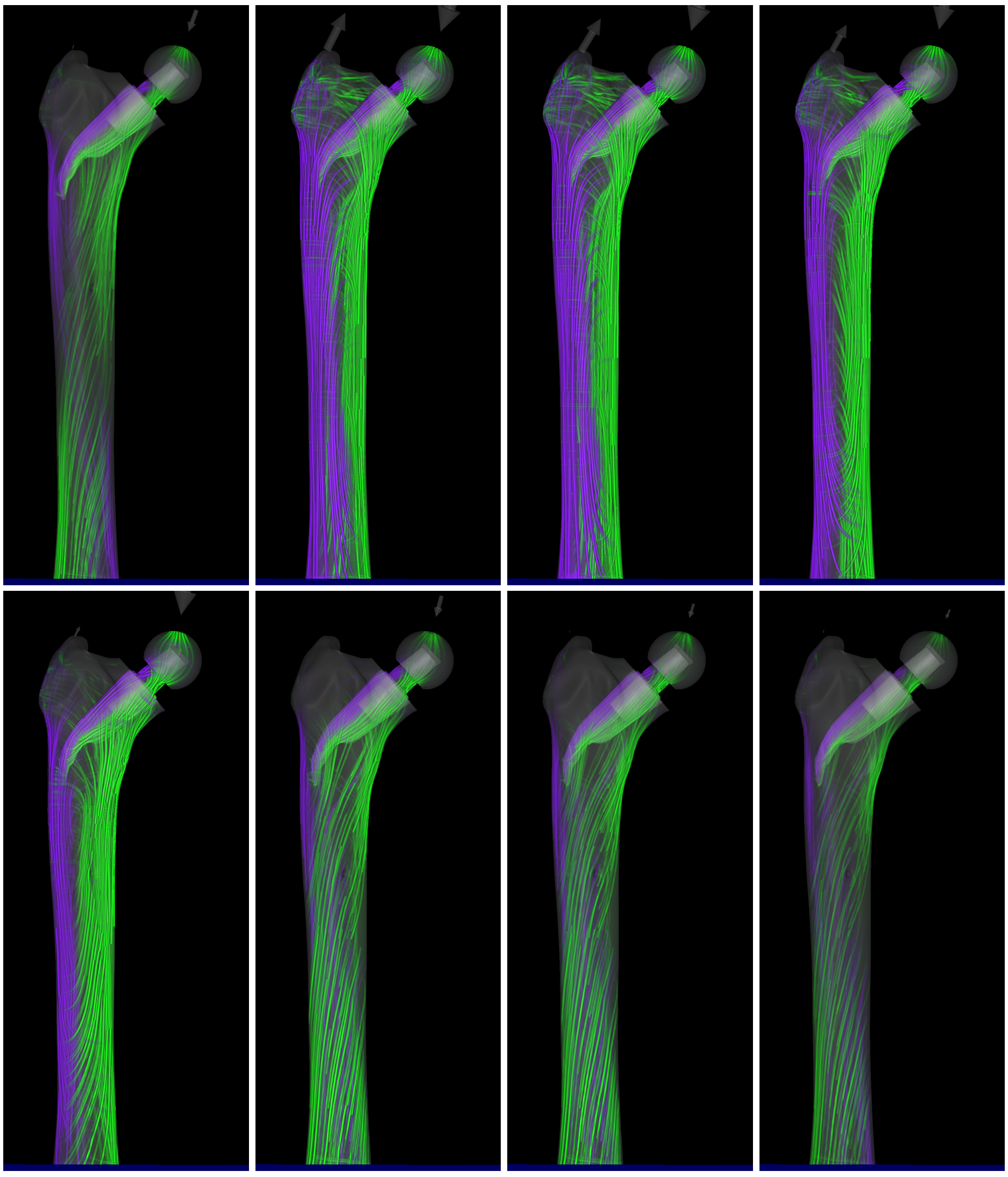


Figure 6.9: *Simulation of the stress distribution during stair climbing (analogous to Figure 6.8) after insertion of the CUT implant.*

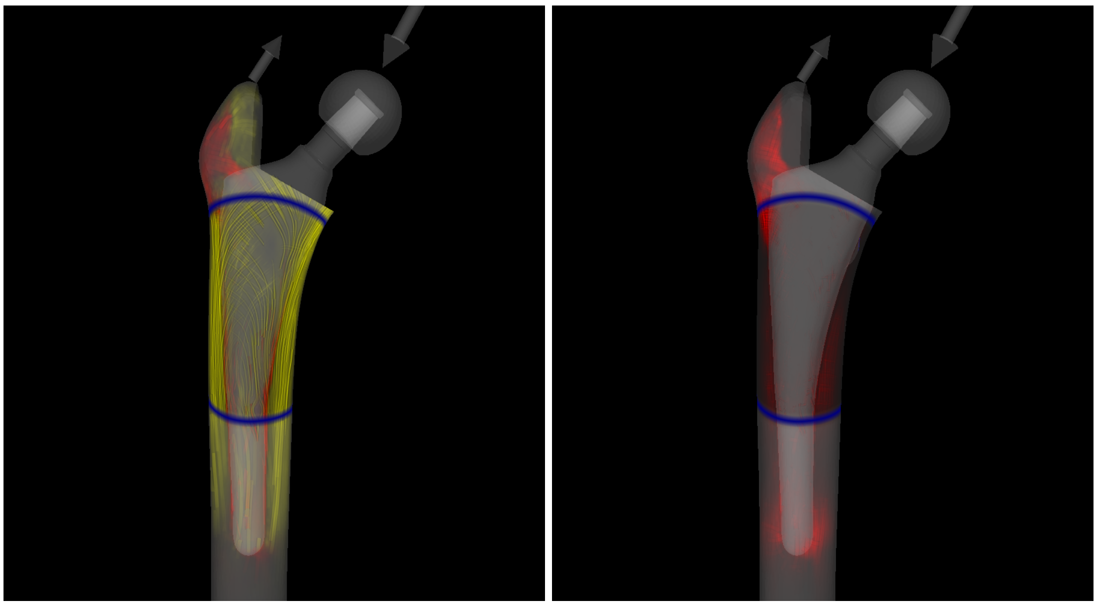
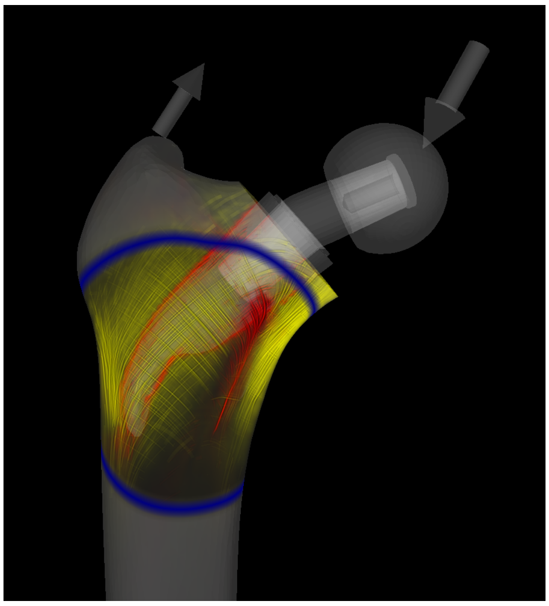
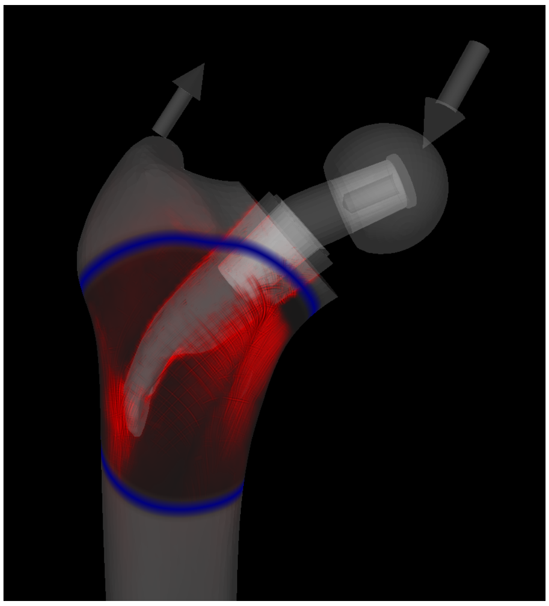


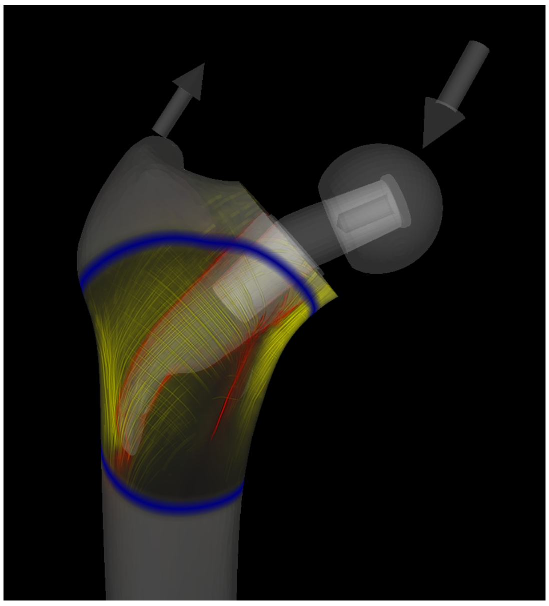
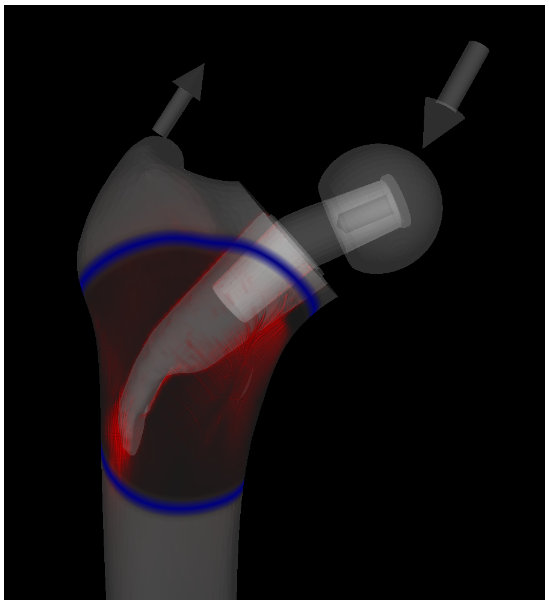
Figure 6.10: *Visualization of the changes of the stress distribution for the G2 implant.*

evaluation of the orthopedic implant planning tool. We have shown the tool to five experienced orthopedic surgeons (assistant or associate professors) at the Klinikum Rechts der Isar, Technische Universität München. Specifically, we have shown to them visualizations of the stress distribution in two patient-specific data sets without and with three different implants, each inserted at two slightly different positions. We have asked them to judge whether the results can be easily understood and are in line with their intuitive expectation, in particular with respect to an immediate assessment of the patient-specific differences in the stress distributions induced by different implant types and positions.

All surgeons rated the proposed stress visualization methods as highly relevant in practice, especially because an intuitive perception of the different kinds of stresses, i.e., tension and compression, as well as their main directional structure in combination with the stress magnitudes is given. All of them stated that the comparative visualization of the stress distributions provides an intuitive and clearly visible rating of different implants, and that the use of such a support tool for preoperative implant selection can significantly reduce the risk of stress shielding. An even stronger statement has been made by all surgeons with respect to the use of the tool in the preoperative planning phase for patients with anatomical anomalies, e.g., pronounced hip-joint dysplasia and bone deformation, or in the context of revision interventions. All five surgeons have

CUT (non-optimal pos.) – normal stresses CUT (non-optimal pos.) – shear stresses

CUT (optimal pos.) – normal stresses CUT (optimal pos.) – shear stresses

Figure 6.11: *Visualization of the changes of the stress distribution for the CUT implant at different positions. It is clearly visible that the CUT implant reduces the stress shielding in the cortical region and yields a more physiological load transmission than the G2 implant (Figure 6.10). Optimal positioning of the CUT implant yields less stress shielding as well as reduced shear stresses around the implant (top vs. bottom row of images).*

indicated strong interests to use the tool for this purpose, even though it causes an additional temporal expense. Due to the more precise and predictable operative care, this temporal expense would be accepted. Two surgeons were pointing out the value of the tool in education, because all variants and results thereof can be tested, compared, and analyzed virtually.

The focus+context technique was considered to be an interesting feature of the system. However, there were some concerns with respect to its usability in clinical practice, since the focus region was placed manually in the demonstration. The majority of the surgeons requested that the focus region should be automatically placed at a medical point of interest (e.g., at the calcar region at the proximal medial femur), a feature we will integrate in the future. Two surgeons were critical concerning the current realization of the tool, because it does not yet support a default mechanism to automatically place the implant into a physiologically meaningful initial position. It was also seen problematic, whether the tool can really be used to precisely analyze the most optimal position of a selected implant. Here it was conjectured that the fine-granular adjustment as provided by the tool might be difficult to be mimicked in reality.

6.8 Conclusion

In this chapter, we have presented advanced, yet interactive, visualization techniques for 3D stress tensor fields. To the best of our knowledge, this is the first approach that allows for the interactive visual exploration of dynamically changing 3D stress tensor fields. Due to the sophisticated visualization options provided, surgeons get deep insights into the highly complex stress tensor data sets arising in preoperative implant planning environments. Especially, we have demonstrated the importance of comparing the stress distribution after an implant has been inserted with the physiological stress state, and we have provided powerful novel visualization techniques for this purpose. These techniques allow the surgeon to optimize the shape, size, and position of the implant with respect to the specific patient, which finally improves the long-term prognosis of the patient.

Chapter 7

Distance Visualization for Implant Planning in Orthopedics

An instant and quantitative assessment of spatial distances between two objects plays an important role in interactive applications such as virtual model assembly, medical operation planning, or computational steering. While some research has been done on the development of distance-based measures between two objects, only very few attempts have been reported to visualize such measures in interactive scenarios. In this chapter we present two different approaches for this purpose, and we investigate the effectiveness of these approaches for intuitive 3D implant positioning in a medical operation planning system. The first approach uses cylindrical glyphs to depict distances, which smoothly adapt their shape and color to changing distances when the objects are moved. This approach computes distances directly on the polygonal object representations by means of ray/triangle mesh intersection. The second approach introduces a set of slices as additional geometric structures, and uses color coding on surfaces to indicate distances. This approach obtains distances from a precomputed distance field of each object. The major findings of the performed user study indicate that a visualization that can facilitate an instant and quantitative analysis of distances between two objects in interactive 3D scenarios is demanding, yet can be achieved by including additional monocular cues into the visualization.

This chapter is based on material that has been originally published in C. Dick, R. Burgkart, and R. Westermann, *Distance visualization for interactive 3D implant planning*, IEEE Transactions on Visualization and Computer Graphics 17 (2011), no. 12, 2173–2182 [DBW11]. The copyright for this material is owned by the IEEE.

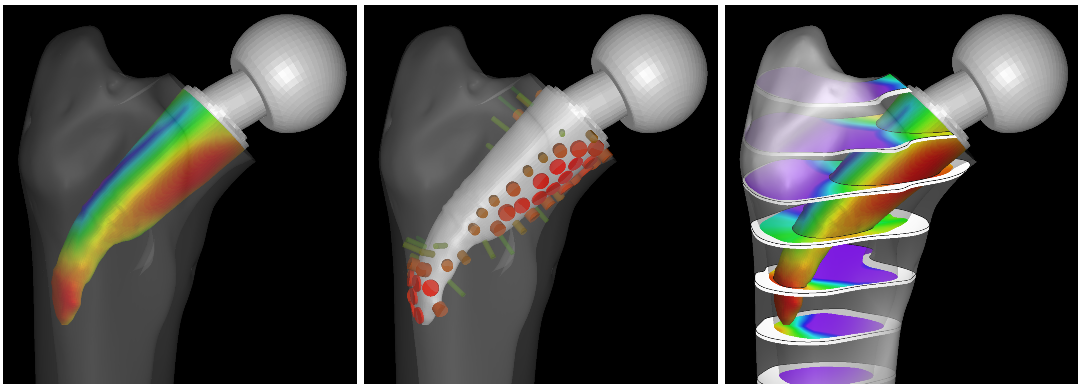


Figure 7.1: *Distance visualization for interactive (> 30 fps) preoperative implant planning using color coding on the implant surface (left), oriented distance glyphs (middle), and color coding on the implant surface and on a set of additional axial slices (right).*

7.1 Introduction and Related Work

Whenever a user in an interactive computer environment is faced with the problem of positioning an object relative to another object, additional cues are required to support an instant communication of the absolute spatial distance of the controlled object to the other object. Even though a number of such cues can be used in general, for example, audio cues or haptic feedback, in most environments only binocular or monoscopic views and mouse-based interaction is available. Furthermore, as for spatial stimuli the visual sense predominates, followed by audition and touch [WW80], binocular or monocular visual cues are usually the preferred indicators.

Psychovisual experiments, however, have indicated that stereopsis provides only relative depth information, which allows inferring on the location of an object relative to another object rather than the absolute distance between them. Monocular cues, such as motion effects or additional geometric structures embedded into a visualization, on the other hand, can effectively indicate absolute distances. Without such direct cues, the estimation of absolute spatial distance between objects becomes very difficult, and typically requires a mental indirection to infer an accurate estimate. Let us refer to the textbooks by Steinman et al. [SSG00] and Schwartz [Sch04] for a scientific review of these arguments.

An example where color is used to indicate distance is given in Figure 7.1 (left): The minimum distances between points on an implant and the surface of a bone into which the implant should be placed are color-coded on the implant. In previous work, color (and iso-contours in color distributions) has been shown to be quite effective at indicating where distance falls below a critical value, and in combination with color legends distance-based coloring even supports quantitative investigations [DMA+01, MLD+04, SPP+10]. As can be seen, however, such visualizations fail in communicating contextual information that is required to effectively interpret distance. Regardless of the specific distance measure that is used, such as shortest distances via distance fields [Gué01, JBS06, RCD+10], spatial distance always 'connects' two reference points, and the human is used to interpret distance in the context of such references. Due to this observation, distance coding in still images via connecting structures such as lines or glyphs has been shown to be very effective. Pang et al. [PWL97] used line primitives of varying length to indicate distances between two surfaces, and in [PTSP02, RCD+10] point-to-point distances between two selected anatomical features were visualized via arrow glyphs accompanied by textual annotations. Without such connecting structures the user has difficulties to associate the points the color is referring to and to use her experience for interpretation.

Apart from the question for an adequate visual encoding of absolute distances in still images, another aspect becomes very important in interactive environments where the user's eyes fixate on the controlled objects and track their motion and orientation. In this case, distances to other objects have to be communicated via peripheral vision, which is the ability to gather information from the environment other than the point of focus. Peripheral vision directs our attention to slight movements in the environment around us, and it is processed significantly faster than vision requiring color. To the best of our knowledge, an investigation of the potential of glyphs for revealing distances between two moving objects in interactive scenarios—taking into account the possibility to dynamically change their shape, size, color, and density—has not been performed so far.

Based on the aforementioned observations, in this chapter we provide novel monocular cues for effectively revealing point-to-point distances between two objects in interactive environments. Based on the mentioned previous techniques using static lines and glyphs we propose a number of extensions to exploit the human's peripheral vision, and we investigate the effectiveness of these extensions in an application-specific user study. Specifically, we address an application in which the user has to position an object with respect to another object that is fixed in space. The underlying application is implant planning for hip joint replacements, where in a preoperative planning phase a surgeon tries to find the patient-specific optimal implant shape, size, and position. In this application, the user can interactively translate and rotate both models together or only the implant relative to the bone.

The monocular cues we present indicate distance via additional geometric structures that smoothly adapt their shape, appearance, or position to changing distance distributions. Two different kinds of monocular cues are provided: For interactive scenarios we have developed dynamically changing and colored distance glyphs to effectively employ peripheral vision for gathering distance information. An example demonstrating the visual sensation produced by these glyphs is given in Figure 7.1 (middle). Note that even in a still image the absolute distances can be revealed much more intuitively compared to pure color coding as in Figure 7.1 (left). One important reason is that the glyphs serve as 'bridging' structures between the implant and the bone surface, enabling the user to associate the distance values to geometric points on the surfaces.

For still images, where peripheral vision is not relevant for perceiving distances, we embed slices into the 3D visualization, and we further augment these slices by colors indicating distance (see Figure 7.1, right). The advantage of this kind of visualization is that it allows for a better observation of the implant due to the sparsity of the used geometric elements, yet providing visual indicators of the spatial relationships between the two objects.

For both types of visual cues, by carefully selecting the shape and appearance attributes of the used graphical primitives, critical regions where the moved implant comes close to the bone surface are identified preattentively, i.e., they are detected very rapidly and accurately by the low-level visual system. The low-level visual system is associated with the lateral geniculate nucleus, or simply called visual thalamus, which is part of the thalamus and responsible for reducing the optical input that is further processed by the visual cortex. In our application, the preattentive visual features we are exploiting to accommodate a rapid analysis of distances are hue, shape, and density of the additional geometric structures. For a more elaborate discussion of the preattentive mechanisms of the low-level visual system let us refer to [Hea99, SW00].

Throughout this chapter we will present the methodology underlying the two new approaches for visualizing spatial distances, and we will analyze the particular strengths as well as the differences and limitations of these approaches in the context of preoperative implant planning. To the best of our knowledge, this is the first time that spatial distance visualization between two objects in interactive environments is addressed other than by color.

The remainder of this chapter is as follows: In the next section we present the medical application for which our distance visualization techniques are developed. Section 7.3 outlines the specific distance measures we use and describes how these measures are mapped onto meaningful graphical representations, such as glyphs and slices. The dedicated GPU implementation of all approaches, by which it is possible to achieve

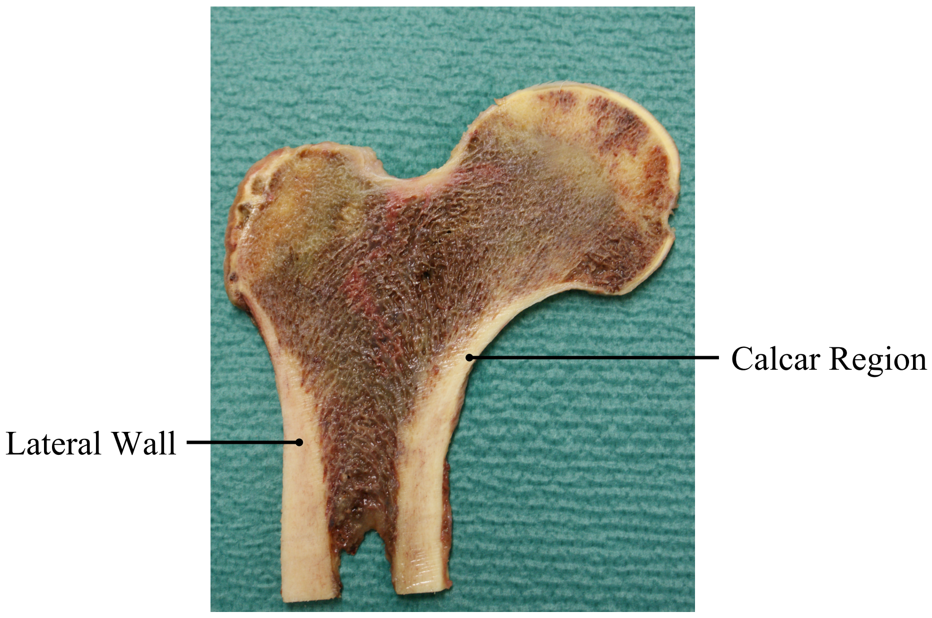


Figure 7.2: *Cross section through a human femur, showing the bone shell consisting of stiff cortical bone and the bone interior consisting of spongious trabecular bone.*

interactive frame rates even for large and complicated data sets, is discussed in Section 7.4. The result section demonstrates the application of our techniques and compares the results to each other. We conclude the chapter with a summary of the basic advantages and disadvantages of both techniques.

7.2 Medical Background

A human femur consists of two types of tissue: The outer shell of the bone consists of stiff cortical bone, whereas the interior consists of spongious trabecular bone (see Figure 7.2). For total hip joint replacement, the femoral head is removed, and the inner trabecular bone has to be partially removed in order to be able to insert the implant.

During preoperative planning, the patient-specific optimal implant shape, size, and position has to be determined according to certain geometrical as well as biomechanical criteria. The geometrical criteria include that the implant has to fit into the interior of the cortical bone shell, without penetrating this shell, and that the joint rotation center has to be preserved. The main biomechanical criterion is related to the stress distribution in the loaded femur, in that the stress distribution after insertion of the implant should closely match the preoperative, physiological stress distribution. For an optimal load transmission from the implant to the adjacent bone stock, it is important that the implant has tight press fit contact to the remaining bone shell, in particular in the calcar region and—especially for short stemmed implant designs—the lateral wall of the femur. If the implant has not sufficient contact in these regions, the stress distribution is changed significantly in that stresses are moved almost completely from regions with no contact to regions with contact. An unphysiological stress distribution can lead to a substantial resorption of bone tissue, with the consequences of fracture or loosening of the implant.

Our computational steering environment for implant planning in orthopedics (see Chapter 5) so far uses semi-transparent rendering of the bone and implant surfaces to visualize the implant position three-dimensionally within the bone, which only allows for a very limited perception of the geometric relationships and absolute distances, i.e., the distances between bone and implant. In the considered application, however, the perception of the geometric relationships and absolute distances is highly important for a precise and intuitive navigation of the implant in the virtual 3D environment, for validating if the current implant configuration meets the described geometrical criteria, as well as for analyzing if a particular unphysiological stress distribution results from the implant not having sufficient contact with the bone, and in this case, how the implant has to be moved or changed in size in order to establish this contact. Therefore, the goal of the work presented in this chapter was to develop dedicated visualization methods that allow for a rapid perception of the geometric relationships and absolute distances between implant and bone. In the computational steering environment, the surgeon can then flexibly switch between the distance and the stress visualization (or use both visualizations simultaneously on a split screen) in order to find the patient-specific optimal implant shape, size, and position.

7.3 Distance Visualization

For the particular application of preoperative implant planning, we employ a triangle mesh of the outer bone surface, a triangle mesh enclosing the trabecular region of the bone and thus separating the cortical and trabecular regions, and a triangle mesh of the implant surface. The first two meshes, which are in the following referred to as the outer and inner bone mesh, are obtained by segmentation from a CT scan. The distances that are of primary relevance for preoperative planning are those between the inner bone mesh and the implant mesh. To simplify the notation, we typically omit the differentiation between inner and outer bone mesh in the following.

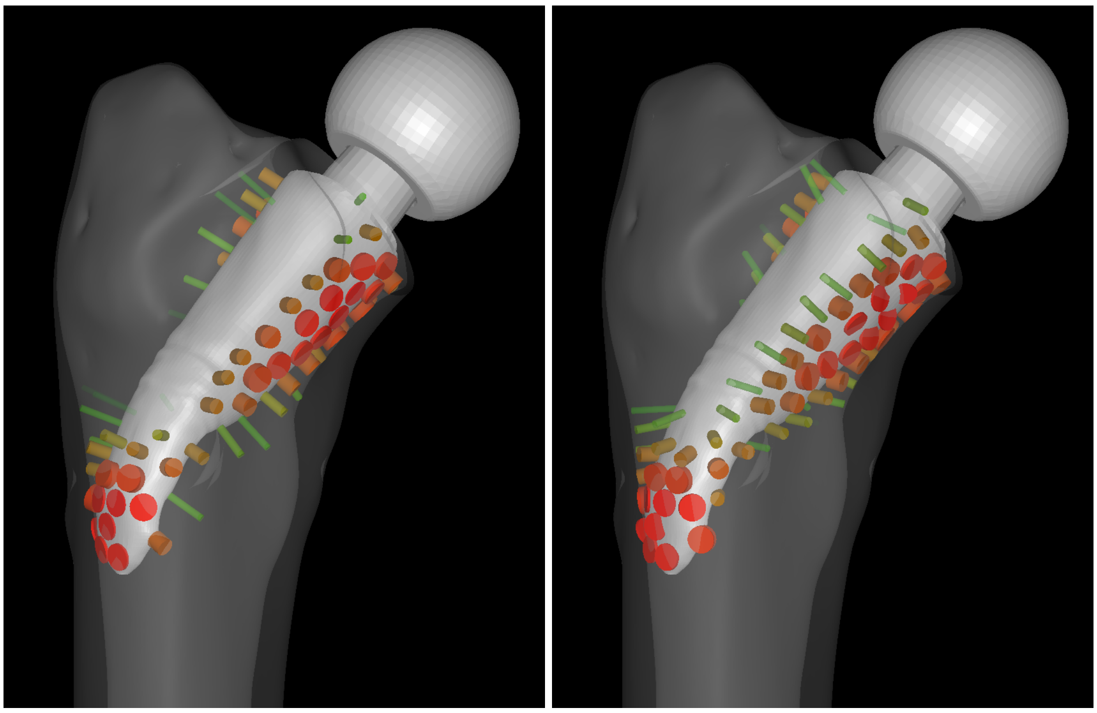


Figure 7.3: *Distance rendering using oriented glyphs, located at certain seed points on the implant surface: Left: Orienting the glyphs orthogonally to the implant surface enables an effective perception of the spatial relationships. Right: Orienting the glyphs toward the respective closest point on the bone surface makes perception of the spatial relationships rather difficult.*

7.3.1 Glyph-based Distance Visualization

Our first approach for the visualization of distances between bone and implant mesh is based on rendering cylindrical glyphs at selected seed points on the implant surface.

We use cylindrical glyphs since these provide an intuitive means to depict distances: The height of the cylinder visualizes the represented distance, and its axis specifies the direction along which the distance is measured. There are distinct strategies to choose the radius of the cylinder: For example, by using a radius proportional to the cylinder's height, the glyph is scaled isotropically with a scaling factor proportional to distance. In contrast, by using a radius inverse proportional to the height (i.e., the product of radius and height is constant), the radius is increasing if the distance is decreasing, and vice versa. This suggests that the glyphs are clamped between the two surfaces, and that they are elastically deforming according to the relative movements of the surfaces, i.e., the glyphs are squeezed with decreasing distances. We use the latter strategy due to this intuitive—since physically plausible—relation between changes of distances and effects on the glyphs. In addition, for the considered medical application, small distance values are of high relevance. By increasing the radius with decreasing height, it is ensured that the glyphs representing small distances remain visible.

For the glyph-based approach, distances between the two surfaces are measured orthogonally to the implant surface and are determined by means of ray/triangle mesh intersection. By orienting the glyphs orthogonally to the implant surface, their orientations are changing smoothly with the implant surface, which enables an effective perception of the spatial relationships. In contrast, orienting the glyphs toward the respective closest point on the bone surface leads to abrupt changes of the glyphs' orientations, which introduces visual clutter and makes perception of the spatial relationships rather difficult (see Figure 7.3).

Since large distances are of minor relevance, we only render glyphs representing distances below a certain maximum value $d_{\max}$. In this way, we also avoid visual cluttering of the glyphs (see Figure 7.4). To avoid that glyphs suddenly pop up or disappear when the represented distance crosses $d_{\max}$, we fade out the glyphs by rendering them as semi-transparent objects. In particular, we linearly interpolate a glyph's alpha value from 1 to 0 for distances from $d_{\max}/2$ to $d_{\max}$. Furthermore, to avoid that the glyphs' radii become arbitrary large, we clamp the radii at a certain maximum value $r_{\max}$. To ensure that glyphs do not overlap, we prescribe a minimum distance between each pair of seed points of $\delta = 2r_{\max}$.

We only render glyphs on those parts of the implant surface which are lying in the interior of the bone. To avoid that the glyphs are hidden by the bone surface, we render the bone meshes semi-transparently. If the implant sticks out of the bone surface, the implant is in a geometrically invalid position. Therefore, at the transition from bone interior to exterior and vice versa, we abruptly remove the glyphs (no fading out) to attract the attention of the user. In addition, we colorize the parts of the implant surface that are lying outside of the bone mesh by using blue color, which is in high contrast to the other colors used for the bone and implant surfaces and the glyphs.

Besides adapting the shape of the glyphs, we additionally colorize the glyphs to encode the represented distance. We linearly interpolate between colors green and red, where green color is used for a distance of $d_{\max}$, and the highlighting red color (corresponding to the importance of small distances) for a distance of 0. This smoothly varying color map was chosen to avoid a rapid change of colors when the glyphs' sizes are changing, which would otherwise disturb the perception of size changes.

The glyph seed points are computed once when a particular implant mesh is selected. In particular, we use spatially fixed seed points and a uniform seeding density. Besides the important goal of achieving visual coherence, these choices are motivated

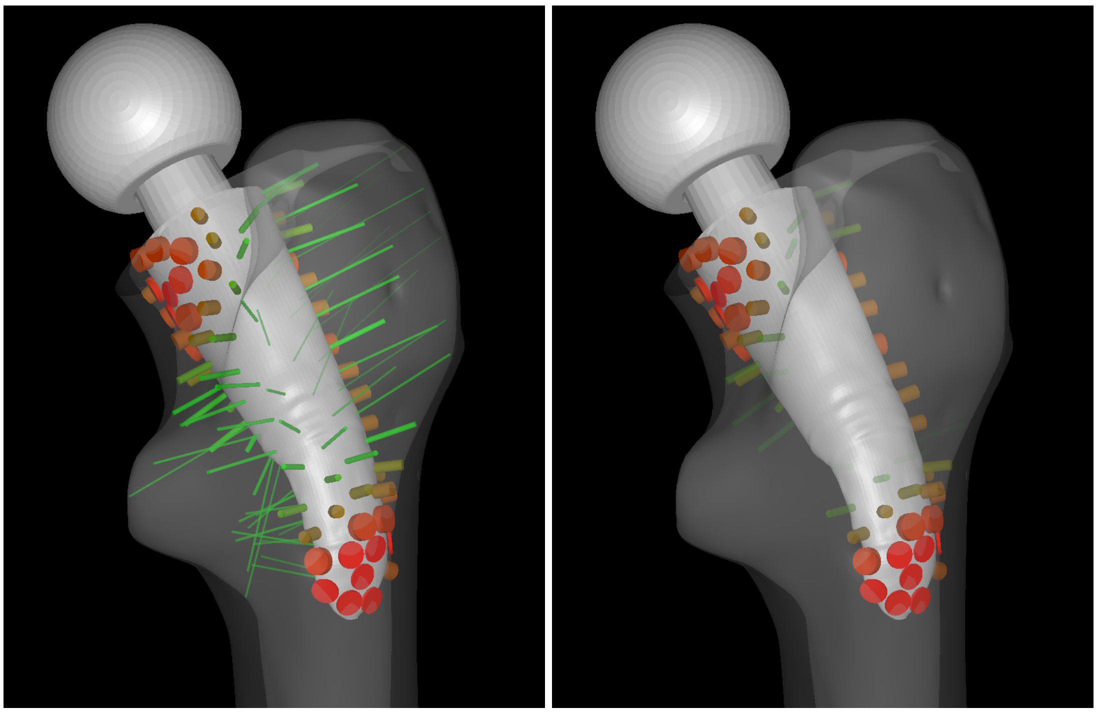


Figure 7.4: *Distance rendering using oriented glyphs: Left: Rendering glyphs for all distances leads to visual cluttering. Right: By rendering glyphs only for distances up to a certain maximum value, visual cluttering can be avoided.*

as follows: By using spatially fixed seed points, we avoid that changes of visualized distances due to moving of the implant are overlaid by changes of visualized distances due to moving of the seed points, which would distract from precisely monitoring the implant's position. Furthermore, by using a fixed seeding density, increasing distances (corresponding to decreasing radii of the glyphs) lead to a visual thinning out of the glyphs. For the considered application, this is in accordance with the fact that large distances are of minor interest.

To determine the seed points, we proceed as follows: In the first step, we build a list of potential seed points L' by iterating over the individual triangles of the implant surface mesh. For each triangle, we test if all edge lengths are at most δ. If this is the case, we add the triangle's barycenter to L'. Otherwise, we perform a 1:4 split of the triangle, and proceed recursively on the newly created triangles. In the second step, we build the list of seed points L by iterating over L'. For each potential seed point P, we test if all seed points already contained in L have at least a distance of δ from P (measured in 3D space). If this is the case, P is added to L, otherwise it is discarded. Since this algorithm considers the individual triangles of the implant mesh separately, it is very robust in that it is independent of the mesh quality.

7.3.2 Slice-based Distance Visualization

One approach for the visualization of the distances between two surfaces is to display at each surface point the distance to the nearest point on the respective other surface via color coding [DMA+01, MLD+04, SPP+10]. This can be realized by using a 3D distance field [JBS06] for each of the two surfaces. Given a set of feature points $\Sigma \subset \mathbb{R}^3$, a distance field is a mapping $d_\Sigma : \mathbb{R}^3 \to \mathbb{R}$ that specifies for each point $\boldsymbol{x}$ in three-dimensional space the distance to the nearest feature point: $d_\Sigma(\boldsymbol{x}) = \inf_{\boldsymbol{y}\in\Sigma} \|\boldsymbol{y} - \boldsymbol{x}\|_2$. A distance field for a particular surface is obtained by choosing Σ to be the set of points on this surface. To create the described visualization, for the first surface, the second surface's distance field is sampled at the points on the first surface, and vice versa. Algorithmically, a distance field can be obtained by means of a distance transform [JBS06].

For our application of interactive 3D implant planning, this visualization approach is shown in Figure 7.1 (left). Here, the distance volume of the bone surface is visualized on the surface of the implant. To provide a large number of distinguishable colors for conveying absolute distance values effectively, a rainbow color map ranging from red (small distance) to violet (large distance) is used. This choice was motivated by our observation that the known shortcomings of rainbow color maps, as for instance discussed in [BTI07], are much less severe in our application scenario. Since here the user's perception is supported by 3D geometry that provides additional depth cues, and the color variation on the implant changes dynamically according to the issued movements, the rainbow color map can convey a very accurate and intuitive image of absolute distances.

Figure 7.1 (left), however, demonstrates that the spatial relationships between the nested meshes along the viewing direction cannot be recognized. As a consequence, it is not intuitively obvious how the implant has to be moved in order to optimize the distances between bone and implant with respect to the specific medical requirements, which makes navigation in an interactive setting rather difficult.

To address this issue, we propose to augment the visualization by rendering additional geometric structures that bridge the volume between implant and bone. Since slicing planes are still the predominant tool for analyzing 3D data in medical diagnosis, and thus are graphical elements to which doctors are accustomed through experience, we render a set of parallel, axial slices passing through the bone and implant, as shown in Figure 7.6 (top left). From the intersection pattern of the slices with the bone and

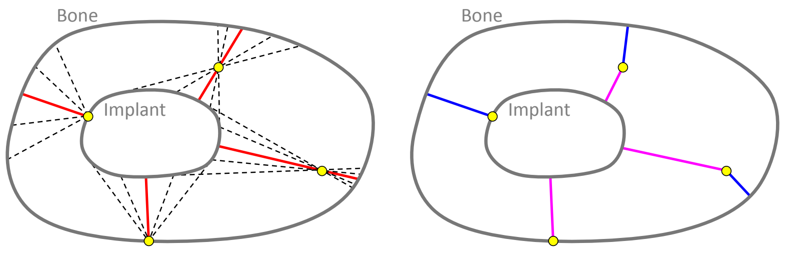


Figure 7.5: *Left: Distance measure (red) at points (yellow) in the volume between bone and implant (for simplicity, the 2D case is shown). Right: Approximation of this distance measure using distance fields (blue: bone distance field, magenta: implant distance field).*

implant the spatial relationships become directly visible. By using an orthographic projection, leading to a parallel rendering of the slices in screen space, occlusions introduced by the slices can be kept at a minimum.

However, it is still difficult to relate the color-coded distances on the implant to the spatial relationships. This can be improved by visualizing color-coded distances also on the slices. Rendering the distance field of the bone not only on the implant but also on the slices results in ring-shaped structures, as illustrated in Figure 7.6 (top right), with the number of rings between implant and bone being a visual clue for the distance. However, on the slices the color coding then only provides the distance of the respective point to the bone, without incorporating the distance to the implant. In particular, the colors on the slices do not change upon movements of the implant, rendering this approach as rather non-intuitive in an interactive setting.

To enable a direct visualization of the distances between bone and implant on the slices, we first have to define a reasonable distance measure at the points in the volume between implant and bone. Considering a point $\boldsymbol{x}$, a reasonable choice would be the minimum length of all line segments passing through $\boldsymbol{x}$ and connecting a point on the implant and bone surface, respectively (see Figure 7.5). On the bone and implant surfaces, this definition reduces to the distance to the nearest point on the respective other surface. In case of parallel or concentric surfaces (as is the case in the current application), this measure can be well approximated by $d_{\text{Bone}}(\boldsymbol{x}) + d_{\text{Implant}}(\boldsymbol{x})$, where d_{Bone} and d_{Implant} denote the distance fields of the bone and implant, respectively. These distance fields have to be computed only once when the respective mesh is changed. The result is illustrated in Figure 7.6 (bottom left).

To achieve a high-quality visualization, we render slices of a certain thickness (1 mm) instead of infinitely thin slices (as is illustrated in Figure 7.6, bottom right for comparison). If the view direction is falling into the slice plane, infinitely thin slices

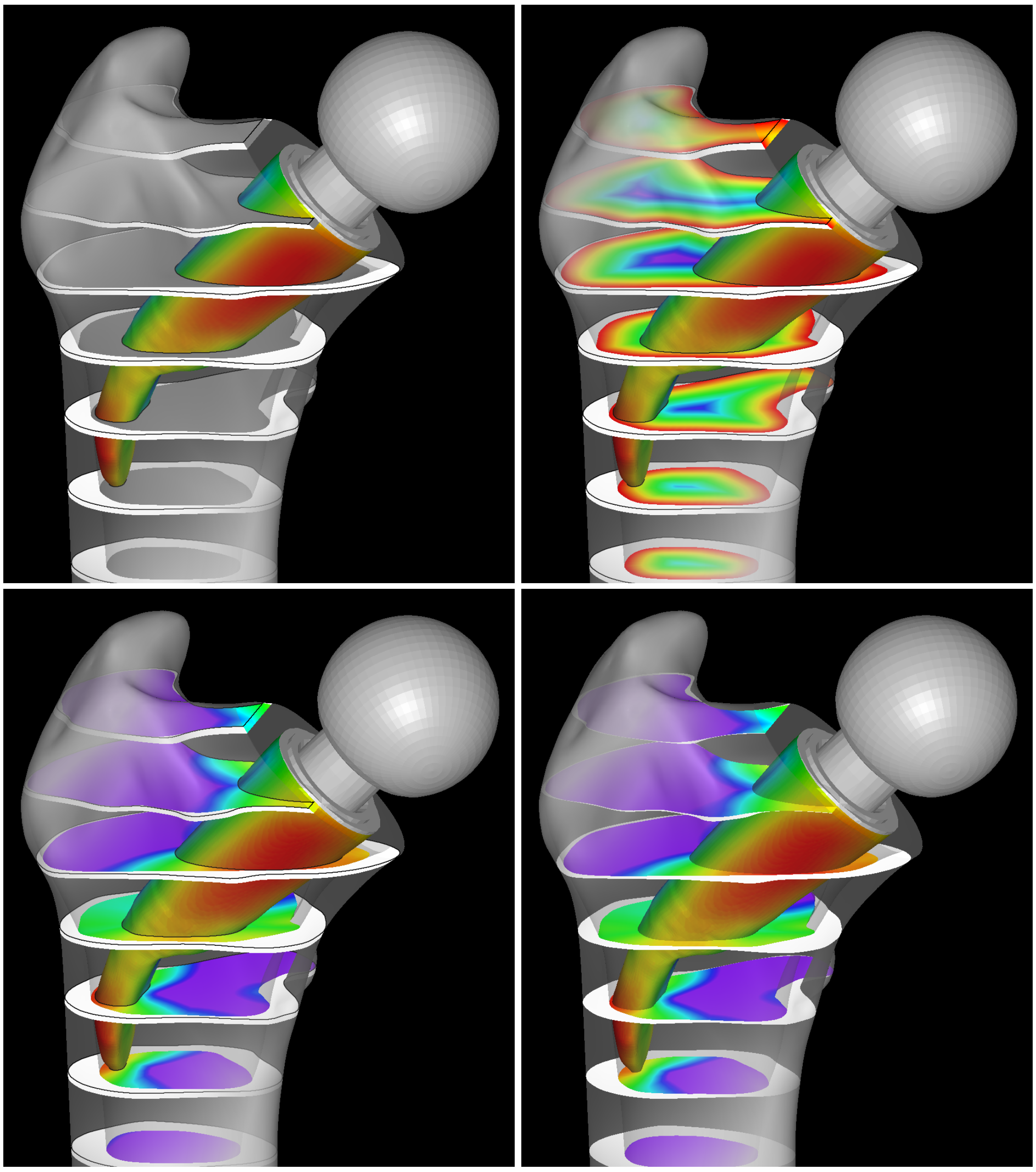


Figure 7.6: *Color coding of distances on the slices: Top left: No color coding of distances. Top right: Distance to the bone. Bottom left: Sum of the distance to the bone and the distance to the implant. Bottom right: Infinitely thin slices without contours.*

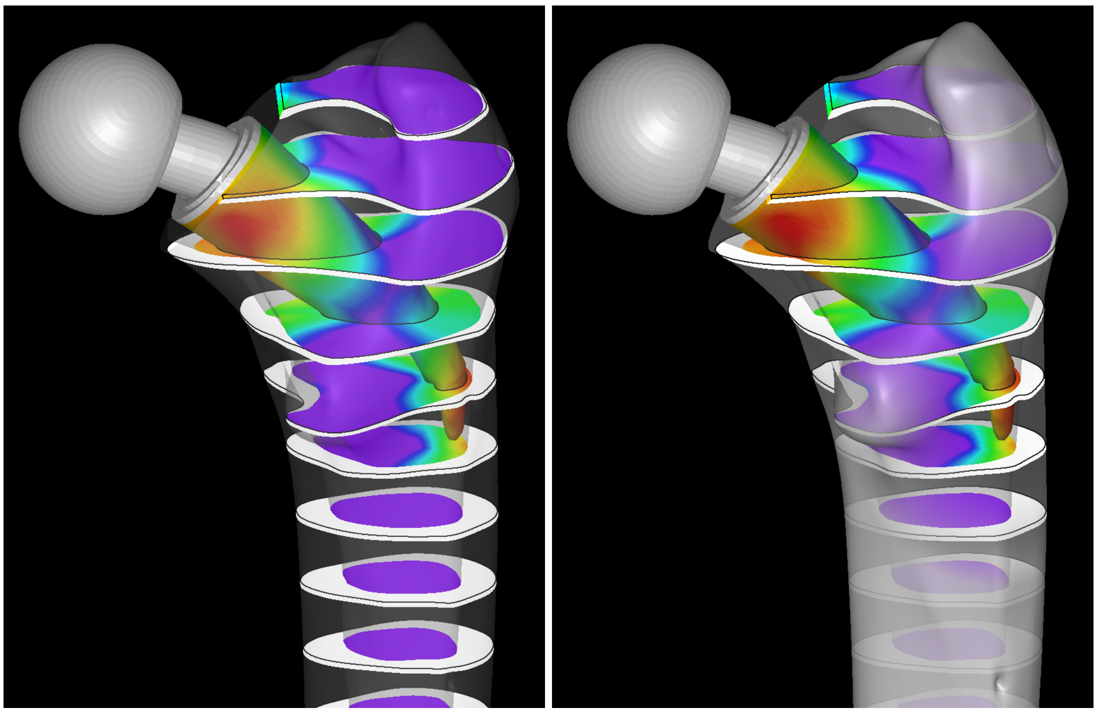


Figure 7.7: *Focus+context visualization: Left: To avoid that colors are washed out by the bone surface, the surface's opacity is reduced. However, this reduces the spatial context information. Right: By using a focus+context approach, the colors are saturated in the important regions, and at the same time the spatial context information is preserved.*

would become invisible. Furthermore, we render black contours along the intersection lines of the slices with the bone and the implant. In this way, partially overlapping projections of slices can be clearly distinguished.

To color-code the distances, we employ a rainbow color scale that is obtained from the HSV color space, using a hue from $0°$ = red to $270°$ = violet. The red end of the scale (highlighting color) is used to encode the distance value 0, the violet end to encode a certain maximum value $d_{\max}$. Analogous to the glyph-based approach, to clearly indicate when the implant has left the interior of the bone, i.e., a geometrically invalid positioning of the implant, the parts of the implant that have left the interior are rendered in a distinct color. Here, we use white color, which is in high contrast to the red color used to encode the smallest distances. In this way geometrically invalid positions are immediately visible.

For the 3D visualization, we use semi-transparent rendering of the bone surfaces. To hide non-important regions with large distances between bone and implant from the visualization, we use a focus+context approach (see Figure 7.7) by scaling the opacity of the fragments of the *outer* bone surface with the factor $(\mathrm{saturate}(d_{\mathrm{Implant}}(\boldsymbol{x})/r))^2$, where r denotes the radius of the focus region, and $\boldsymbol{x}$ is the world space position of the respective fragment. This leads to high and low opacities of the outer bone surface in regions with large and small distances between bone and implant, respectively. In this way, the important regions with small distances between bone and implant, especially the calcar region and the lateral femoral wall, are clearly visible, and at the same time, the perception of the outer shape of the bone and thus of the spatial relationships is improved.

Additionally, we provide the possibility to render the implant as a semi-transparent surface. This enables the surgeon to monitor the distances between bone and implant also on the back side of the implant, which are otherwise hidden by an opaque implant (see Figure 7.11, right, and Figure 7.12, right). Since the user can rotate the entire scene consisting of bone and implant, note however that a clear view on the relevant parts can even be obtained when a color-coded, opaque implant surface is used.

7.4 GPU-Based Implementation

To achieve interactive update rates, we have realized our visualization methods on the GPU. Our implementation is based on the stencil-routed k-buffer [MB07], which enables to render semi-transparent surfaces in correct visibility order. Considering the complex spatial situation with multiple nested meshes, semi-transparent rendering is essential to avoid occlusions. In contrast to z-buffer-based rendering of opaque geometry, where of the set of fragments incoming to the same pixel only the fragment with the smallest depth is captured, the k-buffer allows to capture up to 8 fragments incoming to a pixel. To capture up to 32 fragments per pixel, we employ 4 k-buffer slices, which are filled by rendering the semi-transparent geometry 4 times. For each fragment, we store 2×32 bits to encode an object ID, the fragment's depth in camera space, as well as the fragment's RGBA color value.

Using the k-buffer, rendering of the scene is performed in three steps: In the first step, all opaque geometry is rendered into the standard frame buffer. In the second step, all semi-transparent geometry is rendered into the k-buffer. The semi-transparent geometry consists of the bone surface meshes, the implant surface mesh, as well as the glyphs for our glyph-based distance visualization method. In the third step, a full screen ray-casting pass is performed. First, for each pixel the fragments stored in the k-buffer are fetched. These fragments are then sorted according to ascending camera space depth and blended using front-to-back blending, up to the camera space depth of the opaque geometry, which is obtained by accessing the frame buffer and projecting the clip space depth back into camera space. The accumulated color value is finally blended with the color value of the opaque geometry stored in the frame buffer.

For our glyph-based visualization approach, the glyphs have to be updated whenever the implant position is changed. This update is performed on the CPU. For each glyph seed point, we determine the intersection point of the ray emanating from the seed point in the direction of the outward implant surface normal with the inner bone surface. If no intersection is found, the seed point is lying outside of the inner bone surface, and the seed point is skipped. The distance value is then obtained from the distance between the seed point and the intersection point. To achieve interactive update rates, a kd-tree acceleration structure is used to determine ray/mesh intersections. The glyphs' seed points, normals, and heights are then uploaded into a buffer on the GPU, and by using a single generic cylinder triangle mesh, the glyphs are rendered as semi-transparent geometry using instanced rendering.

For the slice-based distance visualization, the slices are rendered on-the-fly during the ray-casting process by analytically computing the intersections between the rays and the slicing planes. To generate the distance fields of the bone and implant used by this approach, we first create a voxelization for each of the two meshes, using the GPU-based algorithm described in [DGBW08]. We then compute a distance transform of each voxel volume, using the GPU-based algorithm described in [SKW09]. By utilizing the GPU, these computations can be performed in less than 2 seconds for a voxel resolution of 0.25 mm, allowing the user to almost instantaneously switch between different implant shapes and sizes. The distance fields are stored in GPU memory as 3D textures and are sampled using trilinear interpolation.

During the ray traversal of each pixel's stack of fragments, we maintain three flags specifying whether the current position of the ray is lying inside the outer bone mesh, inside the inner bone mesh, and inside the implant mesh, respectively. These flags are toggled whenever a fragment of the respective mesh (identified by the fragment's object ID) is encountered. The flags are used to clip the rendering of the slices at the bone and implant surfaces, to distinctly colorize the parts of the implant lying outside of the inner bone mesh, and to achieve a high-quality rendering of the inner and outer bone mesh: Since the inner and the outer bone mesh are generated in separated processes, it is possible that the inner mesh slightly sticks out of the outer mesh. This is fixed during rendering by clipping the inner mesh at the outer mesh, and by creating additional inner bone mesh fragments on-the-fly during ray-casting to close the resulting holes.

The rendering of contours along the intersection lines of the slices with the bone and the implant is performed in screen space. Whenever a slice fragment is created, we test in a 5×5 neighborhood of adjacent pixels whether the (potential) slice fragment corresponding to the respective neighboring pixel is lying outside of the outer bone surface or within the implant. If such a pixel is found, the center fragment is a border fragment and is colored accordingly, taking into account the number of neighboring 'outside' fragments to obtain antialiased contours.

7.5 Results and Evaluation

All images shown in this chapter were rendered in less than 30 ms on a standard desktop PC, equipped with an Intel Core 2 Quad Q9450 2.66 GHz processor, 8 GB of RAM, and an NVIDIA GeForce GTX 480 graphics card with 1536 MB of local video memory. The view port size was 1920×1200.

To validate the effectiveness of the developed distance visualization methods, we have performed a user study. In this study, the images were created using the following visualization parameters: General parameters: $\alpha_{\text{Outer Bone}} = 0.2$ for focus+context disabled, $\alpha_{\text{Outer Bone}} = 0.9$ for focus+context enabled, $\alpha_{\text{Inner Bone}} = 0.2$, $\alpha_{\text{Implant}} = 1.0$ or 0.35, focus radius $r = 30\,\text{mm}$. Parameters for glyph-based visualization: $r \cdot d = 5\,\text{mm}^2$, $r_{\max} = 2.5\,\text{mm}$, $d_{\max} = 10\,\text{mm}$ (i.e., red corresponds to a distance of 0 mm, green to 10 mm). Parameters for slice-based visualization: Resolution of distance fields 0.25 mm, slice spacing 15 mm, slice thickness 1 mm, $d_{\max} = 16\,\text{mm}$ (i.e., red corresponds to a distance of 0 mm, violet to 16 mm).

7.5.1 User Study

For the user study we recruited 30 participants, comprised of 21 computer science and 6 medical students as well as 3 experienced orthopedic surgeons. The participants were selected to be right-handed, and to have no color vision deficiency. All participants were daily users of computers. The students were exposed to the application for the first time. The three surgeons knew the underlying application including the stress visualization, but also used the distance visualization for the first time. The study was performed using the desktop PC described above with a standard 24 inch monitor. The position of the implant was alternatively controlled using the mouse or a Sensable Phantom Omni haptic device [Sen] (force feedback was not implemented). Simultaneous rotations and translations of the bone and implant model were controlled by mouse or keyboard input, respectively. For each user, an individual session of about 50-60 minutes was

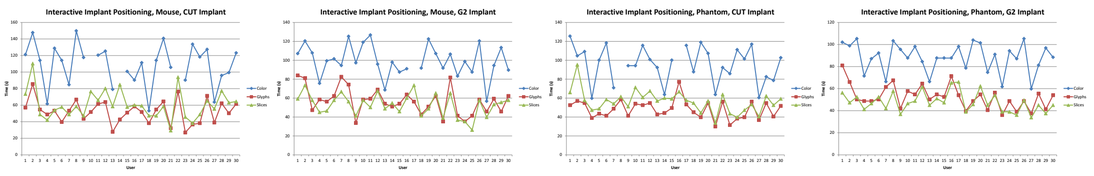


Figure 7.8: *Time (in seconds) needed by the individual users for interactive implant positioning (Experiment 1) dependent on the input device and the distance visualization method, as well as on the implant shape.*

performed. Each session consisted of a demonstration and training phase, and three experiments.

Demonstration and Training Phase. The goal of the demonstration and training phase was to acquaint the participants with the visualization methods and the interaction mechanisms for controlling the implant position using the mouse and the Phantom device. To accelerate the learning process, in this phase we used a simplified geometrical setup consisting of two cylinders of different size, with the larger one (resembling the bone) being fixed. Initially, the smaller cylinder was located outside of the larger cylinder, and its main axis was rotated against the main axis of this cylinder. The task was to move the smaller cylinder (resembling the implant) such that it is concentrically centered inside the larger cylinder.

Firstly, the task was demonstrated to each participant individually using all of the three distance visualization methods, i.e., color coding on the implant surface, the glyph-based approach, and the slice-based approach, as well as both the mouse and the Phantom device. During the demonstration, the meaning of the graphical representations employed in our distance visualization methods, as well as the specific mapping of mouse and Phantom inputs onto movements of the implant, was verbally explained to the participant. After the demonstration, which took about 5 minutes, the participants were asked to perform the task. For each combination of visualization method and input device, about 2 minutes of training time were granted. The entire demonstration and training phase took about 20 minutes.

Experiment 1. In the first experiment, we quantitatively and qualitatively evaluated the three distance visualization methods within the interactive 3D environment. In contrast to the training phase, this experiment was performed using the real bone and implant geometry. The participants were asked to place the implant in a certain position within the bone. This target position was explained to the participants by means of a sketch showing a 2D cross section of the bone and the implant in the target position. In particular, the target position was precisely specified by the following criteria: a) the

	Mouse			Phantom		
Implant	Color	Glyphs	Slices	Color	Glyphs	Slices
CUT	109	52	61	96	48	56
G2	99	57	53	88	53	48

Table 7.1: *Average time (in seconds) needed by the users for interactive implant positioning (Experiment 1) dependent on the input device and the distance visualization method (columns), as well as on the implant shape (rows).*

Implant	Color	Glyphs	Slices
CUT	0/0/30	19/11/0	11/19/0
G2	0/0/30	5/25/0	25/5/0

Table 7.2: *Rankings of the distance visualization methods for interactive implant positioning (Experiment 1). The table shows the numbers of users that ranked the respective method first/second/third.*

implant must be located inside the cortical bone shell without penetrating this shell, b) the distances between implant and bone in the calcar region and at the lateral wall of the femur must be minimized, and c) the rotation center of the joint must be preserved (i.e., the ball of the implant must be centered within the femoral head).

The participants were asked to perform this task a total of 12 times, using the three distance visualization methods, two different implant shapes (CUT and G2), and the two input devices. The implant was always placed at the same starting position outside of the bone, with a different orientation than in the target position, and always the same initial view point was selected. To avoid biasing due to learning effects, with each participant we permuted the order in which the individual visualization methods, implants, and input devices were tested. During the experiment, we measured the time that was needed by the participants to navigate the implant into the target position, allowing a tolerance of 3 mm wrt translation and 5° wrt rotation. If the participant was not able to reach the target position within a 3 minute time interval, we proceeded with the next configuration. The times measured for each individual participant are shown in Figure 7.8, the average times are given in Table 7.1.

In addition, for each of the two implant shapes we asked the participants for a subjective ranking (first, second, third) of the three visualization methods according to their capability to communicate spatial relationships and distances. The rankings are shown in Table 7.2.

The results clearly demonstrate the effectiveness of our novel glyph- and slice-based distance visualization methods. Compared to pure color coding of distances, these methods allow for a significantly faster navigation of the implant into the target posi-

Implant	Color	Glyphs	Slices
CUT	0/0/90	27/63/0	63/27/0
G2	0/0/90	6/84/0	84/6/0

Table 7.3: *Rankings of the distance visualization methods in still images (Experiment 2). The table shows the numbers of times each method was ranked first/second/third.*

tion. Notably, by using color only the task could not be accomplished by 6 of the users within the 3 minute time limit.

For the positioning of the CUT implant (see Figure 7.1), the majority of the users preferred the glyph-based approach over the slice-based approach. This is mainly due to the continuous movement of the glyphs, generating an additional visual attraction that can be processed instantly by the visual system. Especially the use of ‘marshmallow’-like cylindrical glyphs, which simulate quite plausibly the physical deformations of elastic glyphs under pressure, was received very positively by the users. The good matching of this approach with the users’ expectation and experience in reality was given the main reason for this positive response. Considering the slice-based approach, the users criticized the fixation of the slicing planes to a few positions in the 3D domain. Due to this, a non-uniform distance distribution across the implant surface cannot easily be perceived during interactive placement of the implant. In addition, as the distance values within a slice are referring to 3D points that are not in the slicing plane, with increasing distances no immediate spatial correspondence is given between what is seen in the slice and the objects in the scene.

In contrast, for the positioning of the significantly larger G2 implant (see Figures 7.11 and 7.12), the majority of the users indicated a strong preference of the slice-based approach over the glyph-based approach. Due to its large size, this implant has to be navigated in a very narrow surrounding, giving a rather homogeneous distance distribution on the implant. Here, it turns out that in situations were the distances between the implant and the bone are uniformly low, glyph-based approaches fail in clearly depicting these distances. In such cases the very fine differences in the distance values cannot be perceived effectively anymore, and the large number of glyphs everywhere on the implant induces visual clutter.

Experiment 2. In the second experiment, we qualitatively evaluated the three distance visualization methods within 2D still images. We prepared 6 sets of images, showing the two implants in 3 different (optimal and non-optimal) positions and from different view points. Each set consists of three images, each image showing the same implant in the same position and from the same view point, but each using a different distance visualization method. The three images of a set are simultaneously shown to

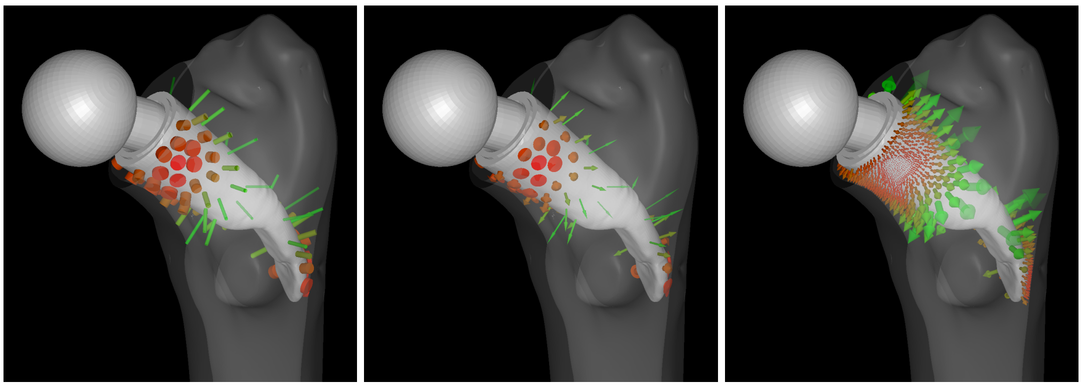


Figure 7.9: *Comparison of different glyph types and different strategies to adapt glyphs to changing distances: Left: Squeezed cylindrical glyphs, uniform seeding density. Middle: Squeezed arrow glyphs, uniform seeding density. Right: Scaled arrow glyphs, adaptive seeding density.*

the user on the monitor. With each user, we permuted the order in which the sets are presented, as well as the order of the images of each set on the screen. For each set, we asked the participant for a subjective ranking of the three visualization methods according to their capability to communicate spatial relationships and distances, analogous to Experiment 1. The rankings are given in Table 7.3.

Interestingly, a different feedback was given than in Experiment 1. Still, both glyph-based and slice-based approaches are rated superior over the pure color coding approach. However, the results show that for 2D still images the slice-based approach is preferred by the users for both the CUT and the G2 implant. Since the slices, and the coloring on them, can very effectively depict the implant and the bone structure, in case of a 2D still image it seems very easy for our visual system to infer on the spatial relationships between the implant and the bone also in the regions between the slices. Furthermore, since slicing planes restrict the visualization to only a few regions in the domain, visual clutter is significantly reduced, giving a more undisturbed view on the relevant structures.

Experiment 3. In the third experiment, we quantitatively and qualitatively evaluated the effect of differently shaped glyphs on the perception of spatial relationships and distances. Besides the ‘marshmallow’-like cylindrical glyphs, we provided arrow glyphs where—analogous to the cylinders—the length of the arrow represents the distance, and the axis represents the directions along which the distance is measured. The arrow glyphs were scaled either in the same way as for the cylindrical glyphs (a smaller distance results in a larger arrow width) or were scaled isotropically (a smaller distance results in a smaller arrow width). In the latter case, the seeding density was increased

	Cylinders Squeezed	Arrows Squeezed	Arrows Scaled
Interactive	22/8/0	0/0/30	8/22/0
Still Image	28/62/0	0/0/90	62/28/0

Table 7.4: *Rankings of different glyph shapes as well as different strategies to adapt glyphs to changing distances for interactive implant positioning and in still images (Experiment 3). The table shows the numbers of times each method was ranked first/second/third.*

	Cylinders Squeezed	Arrows Squeezed	Arrows Scaled
Time	6.4	7.2	2.3
Accuracy	15%	27%	19%

Table 7.5: *Average time (in seconds) needed by the users to estimated the distance represented by a glyph, dependent on the glyph shape and the strategy to adapt glyphs to changing distances (Experiment 3). The second row shows the achieved estimation accuracy.*

with decreasing distance to avoid that small distances become invisible. In Figure 7.9, the different options are shown next to each other. The idea behind using arrows instead of cylinders was to improve the perception of the spatial orientation of the glyphs, since an arrow head (cone) projects onto two clearly distinguishable shapes when seen from the side (triangle) or from the top (circle).

Using an adaptive seeding density of the glyphs is implemented as follows: We first generate a set of uniformly distributed seed points with a minimum spacing of $\delta = 0.5\,\mathrm{mm}$, using the strategy described in Section 7.3.1. In contrast to the case of using a uniform seeding density, we now create glyphs only at a subset of the seed points. In particular, we iterate over the set of seed points in a fix order, and create a glyph at the currently considered seed point if that glyph does not overlap with any previously created glyph. More precisely, a glyph with radius r at a seed point $\boldsymbol{x}$ is created, if for all of the previously created glyphs with respective radius r' and seed point $\boldsymbol{x}'$ the condition $\|\boldsymbol{x} - \boldsymbol{x}'\|_2 \geq r + r'$ is satisfied. The radius r of the glyph (determined by the arrow head) is computed as $r = 0.2d$, where d denotes the height of the glyph (length of the arrow), i.e., the represented distance.

Firstly, we asked the users to move the CUT implant in the interactive 3D environment and to monitor the change of distances between bone and implant, using the three different glyph options. We then asked the users for a ranking of these options according to their capability to communicate spatial relationships and distances and for comments. The rankings are given in Table 7.4.

For interactive positioning there was a preference of all users for squeezed cylindrical glyphs over squeezed arrow glyphs. As the key reasons for this judgment it was reported that the squeezing of arrows was observed a rather unrealistic and thus non-intuitive effect, and especially when arrows become extremely flat or thin, their usual and expected potential to indicate spatial orientation was lost, making the rendering of the arrow head unnecessary.

Comparing squeezed cylindrical glyphs and isotropically scaled arrow glyphs, cylindrical glyphs were favored by over 70% of the users. Considering the arrow glyphs, many users were distracted by glyphs popping up or disappearing (this is a consequence of the adaptation of the seeding density, when larger glyphs are replaced by smaller glyphs, and vice versa), and particularly appreciated the visual coherence of the 'marshmallow'-like cylindrical glyphs. In addition, a small number of users criticized the visual clutter introduced by the increasing number of arrow glyphs when distances become small.

We then asked the users to rank the different glyph options in still images. This was performed similarly as in Experiment 2 by using 3 sets of images showing the CUT implant. The rankings are shown in Table 7.4. The results show a clear preference of the users for the isotropically scaled arrow glyphs over the 'marshmallow'-like cylindrical glyphs. In contrast to the dynamic view, where the arrow glyphs suffer from a limited visual coherence, in the static view the strength of arrows to better depict the glyphs' orientation was observed a clear advantage over cylinders.

Finally, we analyzed the different glyph options with respect to their capability to communicate absolute distances. We prepared three images, showing the CUT implant in different positions and from different view points. For each image, a different glyph option was used to depict the distances. In each image, 3 glyphs were labeled with the respective distances represented by these glyphs. The color scale was hidden from the participants. For each image, we then asked the participants to estimate the distances represented by another selected 5 glyphs, and we measured the time that was needed by the participants. The average times (per glyph) and estimation accuracies can be found in Table 7.5. Using isotropically scaled arrow glyphs, over 85% of the users could give the same quantitative estimate of absolute distance in one third of the time that was needed when cylindrical glyphs were used.

7.5.2 Technical Improvements

After the study was finished, the users were asked for modifications or extensions they could imagine for improving the proposed distance visualization. A small number of users observed the change in saturation on the cap of the cylindrical glyphs when par-

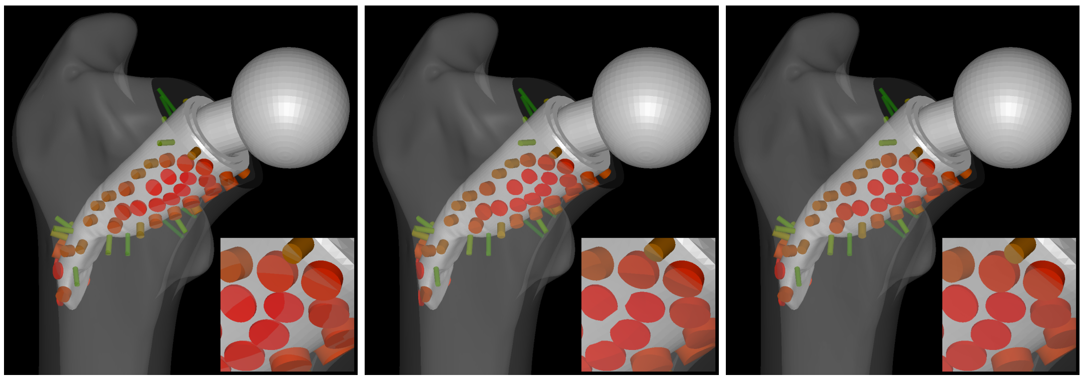


Figure 7.10: *Left: Rendering artifacts resulting from the glyphs partially sticking out of the bone surface. Middle: Clipping the glyphs at the bone surface does not resolve these artifacts. Right: By reordering fragments along the view rays the artifacts are removed.*

tially penetrating the white, semi-transparent bone surface mesh. Even though none of the users considered this effect to have any negative influence on the distance perception, we integrated a rendering option into our tool to reduce this effect.

This rendering option is based on reordering the fragments of the semi-transparent geometry along each view ray, which is implemented using the k-buffer as described in Section 7.4. In our initial implementation, the fragments along each ray are sorted according to ascending camera space depth, and then blended using front-to-back blending. The reordering is now performed on the sorted stack of fragments, before front-to-back blending is applied.

Conceptually, if a ray enters a glyph before entering the bone, or leaves a glyph after leaving the bone (in both cases the glyph penetrates the bone surface), we move the corresponding bone surface fragments in front of or behind the glyph surface fragment, respectively. Entering and leaving of closed surfaces is determined by using flags that are toggled whenever a fragment of the respective surface is encountered. Technically, the reordering of fragments is implemented as follows: If the ray encounters an 'entering' glyph fragment (i.e., a glyph fragment at the transition from the outside to the inside of a glyph), starting from the current position, we search for 'entering' bone fragments (from both the inner and outer bone surface mesh) along the ray within a certain distance from this position. These fragments are moved in front of the glyph fragment, without changing the relative order of the 'entering' bone fragments. Analogous, if the ray encounters a 'leaving' glyph fragment, starting from the current position, we search for 'leaving' bone fragments *backwards* along the ray within a certain distance from this position. These fragments are moved behind the glyph fragment. In our implementation the search distance is set to 5 mm.

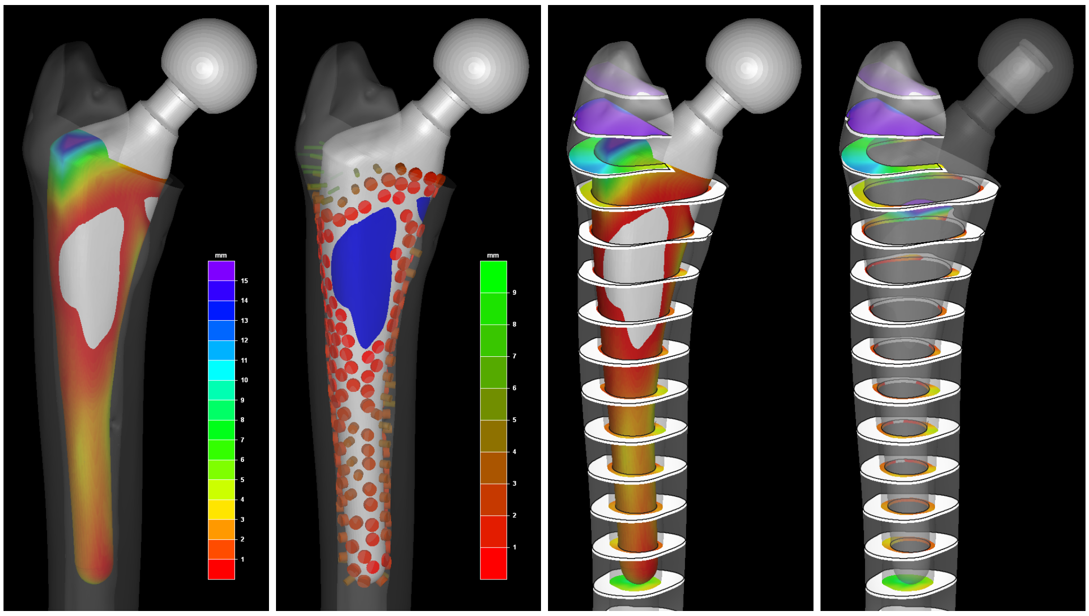


Figure 7.11: *G2 implant, front view: Distance visualization using color coding on the implant surface (left), oriented distance glyphs (middle left), and color coding on the implant surface and on a set of axial slices (middle right). Additionally, we provide the possibility to render the implant semi-transparently (right).*

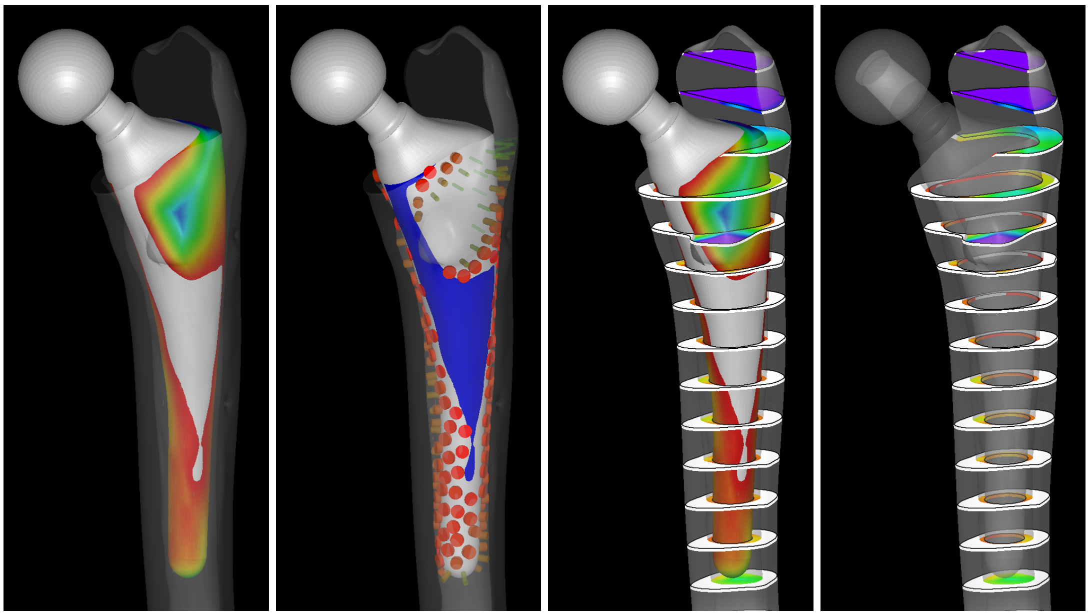


Figure 7.12: *G2 implant from Figure 7.11, back view.*

The front-to-back blending is then performed on the reordered stack of fragments. Even though it is clear that this approach cannot always guarantee that all transitions are fixed and depends on the specified search distance, as demonstrated in Figure 7.10 it can very effectively eliminate virtually all of the transitions.

7.6 Conclusion and Future Work

In this chapter, we have presented two different approaches for visualizing distances between two objects in 3D space. These approaches are specifically tailored to support the user in the positioning of objects in interactive environments. We have introduced novel GPU-based visualization techniques using dynamic glyphs to reveal distance, depth, and directional information, and slicing planes on which distance values are instantly updated during navigation. These approaches have been developed to support 3D implant positioning in a medical planning system, and their strengths and potential limitations in this particular application have been analyzed in a number of user experiments.

In the future, we will analyze the capabilities of our approaches in other interactive applications like virtual part assembly. The investigation of possibilities to automatically integrate distance visualization techniques into medical and technical illustrations will be another focus. From a perceptual point of view, we are very interested in the analysis of the perceptual effects that are caused by the combination of stereo rendering and the proposed distance visualization techniques. A more elaborate study of alternative or supplementary techniques for revealing distance, such as density volumes or textual annotations, in interactive environments is mandatory.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented novel, sophisticated simulation and visualization techniques for the realization of the first computational steering environment for preoperative implant planning in orthopedics. This environment simulates and visualizes the internal stresses in the patient-specific bone dependent on the selected implant shape, size, and position, and thus allows the surgeon to interactively determine an implant configuration that minimizes stress shielding, i.e., closely replicates the preoperative stress state.

To this end, we have presented the first multigrid approach for the physics-based real-time simulation of deformable objects that is implemented entirely on the GPU, and we have developed novel real-time techniques for the visualization of the internal stresses in elastic bodies, and for the visualization of the spatial distances between two objects. In addition, we have presented the first approach for the simulation of cuts in deformable objects that combines cutting with a computationally efficient geometric multigrid solver.

Our techniques are based on three key ingredients. First, we employ a hexahedral discretization of the simulation domain, which can be created and adapted in a very efficient and robust way. Second, our methods are based on efficient multigrid schemes to solve the arising linear systems of equations. And third, we exploit the GPU's massive computational power and memory bandwidth for 3D graphics rendering and general purpose computing. Our results clearly demonstrate that real-time simulation and computational steering are possible on today's desktop PCs at least for selected applications, even when using discretizations at high resolutions.

We would like to emphasize that the applicability of the presented methods is not limited to the described computational steering environment, but that our methods can be used for all kinds of applications that require the real-time simulation of deformable objects and/or the visualization of the internal stresses in elastic bodies, such as virtual surgery simulations or computational steering environments in mechanical engineering.

In the future, we will continue our research in physics-based real-time simulation. First, we will use our GPU-based multigrid solver for real-time fluid simulation. Our goal is to realize a virtual wind tunnel that allows the engineer to interactively study the air flow around an object in a virtual 3D environment. In this application, complex fluid domain boundaries around the object can occur. To accurately resolve these boundaries, we will use an adaptive octree grid, and to improve the multigrid convergence rate, we will use our novel approach for representing complicated topologies on the coarse grids. Since the resulting semi-regular grid leads to varying shapes of the numerical stencil at each vertex, the challenge is to find a mapping of the computations onto the GPU that enables parallel processing of data elements using the same execution paths as well as coalescing of memory accesses, and thus to effectively exploit the GPU's computational power and memory bandwidth.

In addition, we will work on the parallelization of our GPU-based multigrid solver on multiple GPUs. Due to the inherently sequential structure of the multigrid V-cycle (despite of the fact that the individual computational steps of the V-cycle are massively parallel), a rather high number of synchronization operations is required. Therefore, the challenge is to effectively hide latencies resulting from GPU-to-GPU communication via PCI Express buses and InfiniBand network links.

Furthermore, we will pursue research on advanced multigrid schemes with strong smoothers in the context of real-time elasticity and fluid simulation. Here, the goal is to even further increase the accuracy of the approximate solution that is computed in a given time interval. In particular for problems with rough coefficients, advanced multigrid schemes have shown to provide better convergence rates than the standard multigrid scheme with Gauss-Seidel smoother (see [TOS01] for an introduction). A better convergence rate typically leads to performance advantages when solving a linear system until convergence and when solving a system for multiple right-hand sides. However, for our application of real-time simulation it is important to note that we are interested in computing only an approximate solution in a given time interval and that we have to deal with a linear system of equations that changes in every simulation update step. Many existing advanced schemes introduce a significant amount of precomputations, significantly increase the computing time per cycle, and/or cannot be parallelized efficiently on multi-/many-core architectures. For our application of real-time simulation, these precomputations cannot be amortized over a large number of cycles, and the given time interval limits the maximum cycle time. Therefore, advanced multigrid schemes have to be developed that are particularly tailored to the requirements in real-time simulation.

Another research direction will be the on-the-fly adaptation of the simulation grid during runtime of the simulation. This provides improvements with respect to both simulation speed and accuracy, since degrees of freedom are exactly used where they are needed. However, this requires the development of effective a priori oracles that decide where the mesh has to be adapted.

Bibliography

[ABA00] M. J. Aftosmis, M. J. Berger, and G. Adomavicius, *A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries, AIAA 2000-0808*, Proc. 38th AIAA Aerospace Sciences Meeting and Exhibit, 2000.

[AD99] Mark Adams and James W. Demmel, *Parallel multigrid solver for 3D unstructured finite element problems*, Proc. ACM/IEEE Supercomputing, 1999.

[AH08] Yazid Abdelaziz and Abdelmadjid Hamouine, *A survey of the extended finite element*, Computers & Structures **86** (2008), no. 11-12, 1141–1151.

[AP04] Louis-Philippe Amiot and François Poulin, *Computed tomography-based navigation for hip, knee, and spine surgery*, Clinical Orthopaedics and Related Research **421** (2004), 77–86.

[Bat02] Klaus-Jürgen Bathe, *Finite element procedures*, Prentice Hall, 2002.

[BB99] T. Belytschko and T. Black, *Elastic crack growth in finite elements with minimal remeshing*, International Journal for Numerical Methods in Engineering **45** (1999), no. 5, 601–620.

[BCEE97] William D. Bugbee, William J. Culpepper, II, C. Anderson Engh, Jr., and Charles A. Engh, Sr., *Long-term clinical consequences of stress-shielding after total hip arthroplasty without cement*, Journal of Bone and Joint Surgery, American Volume **79** (1997), no. 7, 1007–1012.

[BDH$^+$01] G. Bergmann, G. Deuretzbacher, M. Heller, F. Graichen, A. Rohlmann, J. Strauss, and G. N. Duda, *Hip contact forces and gait patterns from routine activities*, Journal of Biomechanics **34** (2001), no. 7, 859–871.

[Ber01] G. Bergmann, *Hip98*, 2001, http://www.biomechanik.de/.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, ACM TOG **22** (2003), no. 3, 917–924.

[BG00] Daniel Bielser and Markus H. Gross, *Interactive simulation of surgical cuts*, Proc. Pacific Graphics, 2000, pp. 116–125.

[BGTG03] Daniel Bielser, Pascal Glardon, Matthias Teschner, and Markus Gross, *A state machine for real-time cutting of tetrahedral meshes*, Proc. Pacific Graphics, 2003, pp. 377–386.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick, *A multigrid tutorial*, 2 ed., SIAM, 2000.

[BM97] I. Babuška and J. M. Melenk, *The partition of unity method*, International Journal for Numerical Methods in Engineering **40** (1997), no. 4, 727–758.

[BMG99] Daniel Bielser, Volker A. Maiwald, and Markus H. Gross, *Interactive cuts through 3-dimensional soft tissue*, Computer Graphics Forum **18** (1999), no. 3, 31–38.

[BNC96] Morten Bro-Nielsen and Stephane Cotin, *Real-time volumetric deformable models for surgery simulation using finite elements and condensation*, Computer Graphics Forum **15** (1996), no. 3, 57–66.

[BOM$^+$07] Masahiko Bessho, Isao Ohnishi, Juntaro Matsuyama, Takuya Matsumoto, Kazuhiro Imai, and Kozo Nakamura, *Prediction of strength and strain of the proximal femur by a CT-based finite element method*, Journal of Biomechanics **40** (2007), no. 8, 1745–1753.

[BP98] Ed Boring and Alex Pang, *Interactive deformations from tensor fields*, Proc. IEEE Visualization, 1998, pp. 297–304.

[BPWG07] Mario Botsch, Mark Pauly, Martin Wicke, and Markus Gross, *Adaptive space deformations based on rigid cells*, Computer Graphics Forum **26** (2007), no. 3, 339–347.

[Bra77] Archi Brandt, *Multi-level adaptive solutions to boundary-value problems*, Mathematics of Computation **31** (1977), no. 138, 333–390.

[Bra07] Dietrich Braess, *Finite elements: Theory, fast solvers, and applications in solid mechanics*, 3 ed., Cambridge University Press, 2007.

[BTI07] David Borland and Russell M. Taylor II, *Rainbow color map (still) considered harmful*, IEEE Computer Graphics and Applications **27** (2007), no. 2, 14–17.

[BW03] Abhir Bhalerao and Carl-Fredrik Westin, *Tensor splats: Visualising tensor fields by texture mapped volume rendering*, Proc. MICCAI, 2003, pp. 294–302.

[BWW$^+$08] B.-A. Behrens, C. J. Wirth, H. Windhagen, I. Nolte, A. Meyer-Lindenberg, and A. Bouguecha, *Numerical investigations of stress shielding in total hip prostheses*, Journal of Engineering in Medicine **222** (2008), no. 5, 593–600.

[CBB$^+$10] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, *Solving lattice QCD systems of equations using mixed precision solvers on GPUs*, Computer Physics Communications **181** (2010), no. 9, 1517–1528.

[CD05] Eric Chan and Frédo Durand, *Gpu gems*, vol. 2, ch. Fast Prefiltered Lines, Addison-Wesley, 2005.

[CDA99] Stéphane Cotin, Hervé Delingette, and Nicholas Ayache, *Real-time elastic deformations of soft tissues for surgery simulation*, IEEE TVCG **5** (1999), no. 1, 62–73.

[CDA00] ———, *A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation*, The Visual Computer **16** (2000), no. 8, 437–452.

[CGC$^+$02] Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović, *A multiresolution framework for dynamic deformations*, Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2002, pp. 41–47.

[Cia88] Philippe G. Ciarlet, *Mathematical elasticity, volume I: Three-dimensional elasticity*, Studies in Mathematics and Its Applications, vol. 20, Elsevier, 1988.

[CTA$^+$08] Olivier Comas, Zeike A. Taylor, Jérémie Allard, Sébastien Ourselin, Stéphane Cotin, and Josh Passenger, *Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework SOFA*, Proc. International Symposium on Biomedical Simulation, Lecture Notes in Computer Science, vol. 5104, 2008, pp. 28–39.

[DBW11] Christian Dick, Rainer Burgkart, and Rüdiger Westermann, *Distance visualization for interactive 3D implant planning*, IEEE TVCG **17** (2011), no. 12, 2173–2182.

[DDBC99] Gilles Debunne, Mathieu Desbrun, Alan H. Barr, and Marie-Paule Cani, *Interactive multiresolution animation of deformable models*, Proc. 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*, Proc. ACM SIGGRAPH, 2001, pp. 31–36.

[DGBW08] Christian Dick, Joachim Georgii, Rainer Burgkart, and Rüdiger Westermann, *Computational steering for patient-specific implant planning in orthopedics*, Proc. Eurographics Workshop on Visual Computing for Biomedicine, 2008, pp. 83–92.

[DGBW09a] ______, *A 3D simulation system for hip joint replacement planning*, Proc. World Congress on Medical Physics and Biomedical Engineering, IFMBE Proceedings, vol. 25/IV, 2009, pp. 363–366.

[DGBW09b] ______, *Stress tensor field visualization for implant planning in orthopedics*, IEEE TVCG **15** (2009), no. 6, 1399–1406.

[DGW11a] Christian Dick, Joachim Georgii, and Rüdiger Westermann, *A hexahedral multigrid approach for simulating cuts in deformable objects*, IEEE TVCG **17** (2011), no. 11, 1663–1675.

[DGW11b] ______, *A real-time multigrid finite hexahedra method for elasticity simulation using CUDA*, Simulation Modelling Practice and Theory **19** (2011), no. 2, 801–816.

[DH93] Thierry Delmarcelle and Lambertus Hesselink, *Visualizing second-order tensor fields with hyperstreamlines*, IEEE Computer Graphics and Applications **13** (1993), no. 4, 25–33.

[DH94] ______, *The topology of symmetric, second-order tensor fields*, Proc. IEEE Visualization, 1994, pp. 140–147.

[DMA$^+$01] Ç. Demiralp, G. E. Marai, S. Andrews, D. H. Laidlaw, J. J. Crisco, and C. Grimm, *Modeling and visualization of inter-bone distances in joints*, Proc. IEEE Visualization, Work in Progress, 2001, pp. 24–25.

[dSZDL01] M. J. da Silva, S. Zhang, Ç. Demiralp, and D. H. Laidlaw, *Visualizing diffusion tensor volume differences*, Proc. IEEE Visualization, Work in Progress, 2001, pp. 16–17.

[EKS03] Olaf Etzmuß, Michael Keckeisen, and Wolfgang Straßer, *A fast finite element solution for cloth modelling*, Proc. Pacific Graphics, 2003, pp. 244–251.

[FDA02] C. Forest, H. Delingette, and N. Ayache, *Removing tetrahedra from a manifold mesh*, Proc. Computer Animation, 2002, pp. 225–229.

[FG99] Sarah F. Frisken-Gibson, *Using linked volumes to model object collisions, deformation, cutting, carving, and joining*, IEEE TVCG **5** (1999), no. 4, 333–348.

[GCMS00] Fabio Ganovelli, Paolo Cignoni, Claudio Montani, and Roberto Scopigno, *A multiresolution model for soft objects supporting interactive cuts and lacerations*, Computer Graphics Forum **19** (2000), no. 3, 271–281.

[GG06] H. Gollwitzer and R. Gradinger, *Hüfte: Standardimplantat*, Ossäre Integration, Springer, 2006, pp. 99–109.

[GKS02] Eitan Grinspun, Petr Krysl, and Peter Schröder, *CHARMS: A simple framework for adaptive simulation*, ACM TOG **21** (2002), no. 3, 281–290.

[GKW07] Joachim Georgii, Jens Krüger, and Rüdiger Westermann, *Interactive collision detection for deformable and GPU objects*, IADIS International Journal on Computer Science and Information Systems **2** (2007), no. 2, 162–180.

[GSMY$^+$08] Dominik Göddeke, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Hilmar Wobker, Christian Becker, and Stefan Turek, *Using GPUs to improve multigrid solver performance on a cluster*, International Journal of Computational Science and Engineering **4** (2008), no. 1, 36–55.

[Gué01] André Guéziec, *"Meshsweeper": Dynamic point-to-polygonal-mesh distance and applications*, IEEE TVCG **7** (2001), no. 1, 47–61.

[GW05] Joachim Georgii and Rüdiger Westermann, *Interactive simulation and rendering of heterogeneous deformable bodies*, Proc. Vision, Modeling and Visualization, 2005, pp. 383–390.

[GW06] ______, *A multigrid framework for real-time simulation of deformable bodies*, Computer & Graphics **30** (2006), no. 3, 408–415.

[GW08] ______, *Corotated finite elements made fast and stable*, Proc. Workshop in Virtual Reality Interactions and Physical Simulation, 2008, pp. 11–19.

[Hac85] Wolfgang Hackbusch, *Multi-grid methods and applications*, Springer Series in Computational Mathematics, Springer, 1985.

[HBD$^+$01] M. O. Heller, G. Bergmann, G. Deuretzbacher, L. Dürselen, M. Pohl, L. Claes, N. P. Haas, and G. N. Duda, *Musculo-skeletal loading conditions at the hip during walking and stair climbing*, Journal of Biomechanics **34** (2001), no. 7, 883–893.

[HBPA01] Khader M. Hasan, Peter J. Basser, Dennis L. Parker, and Andrew L. Alexander, *Analytical computation of the eigenvalues and eigenvectors in DT-MRI*, Journal of Magnetic Resonance **152** (2001), no. 1, 41–47.

[Hea99] Christopher G. Healey, *Fundamental issues of visual perception for effective image generation*, ACM SIGGRAPH Course, 1999.

[HEPP01] H. Handels, J. Ehrhardt, W. Plötz, and S. J. Pöppl, *Simulation of hip operations and design of custom-made endoprostheses using virtual reality techniques*, Methods of Information in Medicine **40** (2001), no. 2, 74–77.

[HFH$^+$04] Ingrid Hotz, Louis Feng, Hans Hagen, Bernd Hamann, Kenneth Joy, and Boris Jeremic, *Physically based methods for tensor field visualization*, Proc. IEEE Visualization, 2004, pp. 123–130.

[HG07] Mike Houston and Naga Govindaraju, *GPGPU: General purpose computation on graphics hardware*, ACM SIGGRAPH Courses, 2007.

[HH07] Eldad Haber and Stefan Heldmann, *An octree multigrid method for quasi-static Maxwell's equations with highly discontinuous coefficients*, Journal of Computational Physics **223** (2007), no. 2, 783–796.

[Hig86] Nicholas J. Higham, *Computing the polar decomposition—with applications*, SIAM Journal on Scientific and Statistical Computing **7** (1986), no. 4, 1160–1174.

[HLL97] Lambertus Hesselink, Yuval Levy, and Yingmei Lavin, *The topology of symmetric, second-order 3D tensor fields*, IEEE TVCG **3** (1997), no. 1, 1–11.

[HS04] Michael Hauth and Wolfgang Strasser, *Corotational simulation of deformable solids*, Journal of WSCG **12** (2004), no. 1, 137–144.

[JBS06] Mark W. Jones, J. Andreas Bærentzen, and Milos Sramek, *3D distance fields: A survey of techniques and applications*, IEEE TVCG **12** (2006), no. 4, 581–599.

[JCM07] Xiaogang Jin, Shaochun Chen, and Xiaoyang Mao, *Computer-generated marbling textures: A GPU-based design system*, IEEE Computer Graphics and Applications **27** (2007), no. 2, 78–84.

[JK09] Lenka Jeřábková and Torsten Kuhlen, *Stable cutting of deformable objects in virtual environments using XFEM*, IEEE Computer Graphics and Applications **29** (2009), no. 2, 61–71.

[JP99] Doug L. James and Dinesh K. Pai, *ArtDefo: Accurate real time deformable objects*, Proc. ACM SIGGRAPH, 1999, pp. 65–72.

[KF03] Joyce H. Keyak and Yuri Falkinstein, *Comparison of in situ and in vitro CT scan-based finite element model predictions of proximal femoral fracture load*, Medical Engineering and Physics **25** (2003), no. 9, 781–787.

[KGW$^+$94] Tony M. Keaveny, Edward Guo, Edward F. Wachtel, Thomas A. McMahon, and Wilson C. Hayes, *Trabecular bone exhibits fully linear elastic behavior and yields at low strains*, Journal of Biomechanics **27** (1994), no. 9, 1127–1136.

[KH08] Michael Kazhdan and Hugues Hoppe, *Streaming multigrid for gradient-domain operations on large images*, ACM TOG **27** (2008), no. 3, 21:1–21:10.

[Kin04] Gordon Kindlmann, *Superquadric tensor glyphs*, Proc. Joint Eurographics - IEEE TCVG Symposium on Visualization, 2004, pp. 147–154.

[KKKW05] Jens Krüger, Peter Kipfer, Polina Kondratieva, and Rüdiger Westermann, *A particle system for interactive visualization of 3D flows*, IEEE TVCG **11** (2005), no. 6, 744–756.

[KKW05] Polina Kondratieva, Jens Krüger, and Rüdiger Westermann, *The application of GPU particle tracing to diffusion tensor field visualization*, Proc. IEEE Visualization, 2005, pp. 73–78.

[KLS94] J. H. Keyak, I. Y. Lee, and H. B. Skinner, *Correlations between orthogonal mechanical properties and density of trabecular bone: Use of different densitometric measures*, Journal of Biomedical Materials Research **28** (1994), no. 11, 1329–1336.

[KMB$^+$09] Peter Kaufmann, Sebastian Martin, Mario Botsch, Eitan Grinspun, and Markus Gross, *Enrichment textures for detailed cutting of shells*, ACM TOG **28** (2009), no. 3, 50:1–50:10.

[KMOD09] Lily Kharevych, Patrick Mullen, Houman Owhadi, and Mathieu Desbrun, *Numerical coarsening of inhomogeneous elastic materials*, ACM TOG **28** (2009), no. 3, 51:1–51:8.

[KOO$^+$09] Kenji Kawate, Yutaka Ohneda, Tetsuji Ohmura, Hiroshi Yajima, Kazuya Sugimoto, and Yoshinori Takakura, *Computed tomography-based custom-made stem for dysplastic hips in japanese patients*, Journal of Arthroplasty **24** (2009), no. 1, 65–70.

[KSW06] Jens Krüger, Jens Schneider, and Rüdiger Westermann, *ClearView: An interactive context preserving hotspot visualization technique*, IEEE TVCG **12** (2006), no. 5, 941–948.

[Kum05] Benno Kummer, *Biomechanik: Form und Funktion des Bewegungsapparates*, Deutscher Ärzte-Verlag, 2005.

[KW03] Jens Krüger and Rüdiger Westermann, *Acceleration techniques for GPU-based volume rendering*, Proc. IEEE Visualization, 2003, pp. 287–292.

[KWH00] Gordon Kindlmann, David Weinstein, and David Hart, *Strategies for direct volume rendering of diffusion tensor fields*, IEEE TVCG **6** (2000), no. 2, 124–138.

[LAK$^+$98] David H. Laidlaw, Eric T. Ahrens, David Kremers, Matthew J. Avalos, Russell E. Jacobs, and Carol Readhead, *Visualizing diffusion tensor images of the mouse spinal cord*, Proc. IEEE Visualization, 1998, pp. 127–134.

[LGF04] Frank Losasso, Frédéric Gibou, and Ron Fedkiw, *Simulating water and smoke with an octree data structure*, ACM TOG **23** (2004), no. 3, 457–462.

[LJWD08] Youquan Liu, Shaohui Jiao, Wen Wu, and Suvranu De, *GPU accelerated fast FEM deformation simulation*, Proc. IEEE Asia Pacific Conference on Circuits and Systems, 2008, pp. 606–609.

[LPR+09] Florian Liehr, Tobias Preusser, Martin Rumpf, Stefan Sauter, and Lars Ole Schwen, *Composite finite elements for 3D image based computing*, Computing and Visualization in Science **12** (2009), no. 4, 171–188.

[MB07] Kevin Myers and Louis Bavoil, *Stencil routed A-buffer*, Proc. ACM SIGGRAPH Technical Sketch Program, 2007, p. 21.

[MBF04] Neil Molino, Zhaosheng Bao, and Ron Fedkiw, *A virtual node algorithm for changing mesh topology during simulation*, ACM TOG **23** (2004), no. 3, 385–392.

[MDB99] Nicolas Moës, John Dolbow, and Ted Belytschko, *A finite element method for crack growth without remeshing*, International Journal for Numerical Methods in Engineering **46** (1999), no. 1, 131–150.

[MDM+02] Matthias Müller, Julie Dorsey, Leonard McMillan, Robert Jagnow, and Barbara Cutler, *Stable real-time deformations*, Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2002, pp. 49–54.

[MG04] Matthias Müller and Markus Gross, *Interactive virtual materials*, Proc. Graphics Interface, 2004, pp. 239–246.

[MHS05] Jesper Mosegaard, Peder Herborg, and Thomas Sangild Sørensen, *A GPU accelerated spring mass system for surgical simulation*, Studies in Health Technology and Informatics **111** (2005), 342–348.

[MHTG05] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross, *Meshless deformations based on shape matching*, ACM TOG **24** (2005), no. 3, 471–478.

[Mic10] Microsoft, *Windows DirectX graphics documentation (part of the DirectX Software Development Kit)*, June 2010, http://www.microsoft.com/directx.

[MJLW07] Karol Miller, Grand Joldes, Dane Lance, and Adam Wittek, *Total lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation*, Communications in Numerical Methods in Engineering **23** (2007), no. 2, 121–134.

[MK00] Andrew B. Mor and Takeo Kanade, *Modifying soft tissue models: Progressive cutting with minimal new element creation*, Proc. MICCAI, Lecture Notes in Computer Science, vol. 1935, 2000, pp. 598–608.

[MKB+08] Sebastian Martin, Peter Kaufmann, Mario Botsch, Martin Wicke, and Markus Gross, *Polyhedral finite elements using harmonic basis functions*, Computer Graphics Forum **27** (2008), no. 5, 1521–1529.

[MLD+04] G. Elisabeta Marai, David Laidlaw, Çağatay Demiralp, Stuart Andrews, Cindy M. Grimm, and Joseph J. Crisco, *Estimating joint contact areas and ligament lengths from bone kinematics and surfaces*, IEEE Transactions on Biomedical Engineering **51** (2004), no. 5, 790–799.

[MSE$^+$06] Dorit Merhof, Markus Sonntag, Frank Enders, Christopher Nimsky, Peter Hastreiter, and Günther Greiner, *Hybrid visualization for white matter tracts using triangle strips and point sprites*, IEEE TVCG **12** (2006), no. 5, 1181–1188.

[Mül71] Maurice Edmond Müller, *Die hüftnahen Femurosteotomien unter Berücksichtigung der Form, Funktion und Beanspruchung des Hüftgelenkes*, G. Thieme, 1971.

[NKJF09] Matthieu Nesme, Paul G. Kry, Lenka Jeřábková, and François Faure, *Preserving topology and elasticity for embedded deformable models*, ACM TOG **28** (2009), no. 3, 52:1–52:9.

[NMK$^+$06] Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman, and Mark Carlson, *Physically based deformable models in computer graphics*, Computer Graphics Forum **25** (2006), no. 4, 809–836.

[NvdS00] Han-Wen Nienhuys and A. Frank van der Stappen, *Combining finite element deformation with cutting for surgery simulations*, Proc. Eurographics - Short Presentations, 2000, pp. 43–52.

[NvdS01] ______, *A surgery simulation supporting cuts and finite element deformation*, Proc. MICCAI, Lecture Notes in Computer Science, vol. 2208, 2001, pp. 145–152.

[NVI09] NVIDIA, *NVIDIA's next generation CUDA compute architecture: Fermi*, 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[NVI10a] ______, *NVIDIA CUDA C best practices guide version 3.2 (part of the CUDA Toolkit 3.2)*, August 2010, http://www.nvidia.com/cuda.

[NVI10b] ______, *NVIDIA CUDA C programming guide version 3.2 (part of the CUDA Toolkit 3.2)*, November 2010, http://www.nvidia.com/cuda.

[OBH02] James F. O'Brien, Adam W. Bargteil, and Jessica K. Hodgins, *Graphical modeling and animation of ductile fracture*, ACM TOG **21** (2002), no. 3, 291–294.

[OH78] Indong Oh and William H. Harris, *Proximal strain distribution in the loaded femur*, Journal of Bone and Joint Surgery, American Volume **60** (1978), no. 1, 75–85.

[OH99] James F. O'Brien and Jessica K. Hodgins, *Graphical modeling and animation of brittle fracture*, Proc. ACM SIGGRAPH, 1999, pp. 137–146.

[OHL$^+$08] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, *GPU computing*, Proceedings of the IEEE **96** (2008), no. 5, 879–899.

[OLG$^+$07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell, *A survey of general-purpose computation on graphics hardware*, Computer Graphics Forum **26** (2007), no. 1, 80–113.

[Pau73] F. Pauwels, *Atlas zur Biomechanik der gesunden und kranken Hüfte*, Springer, 1973.

[PFK07] Vid Petrovic, James Fallon, and Falko Kuester, *Visualizing whole-brain DTI tractography with GPU-based tuboids and LoD management*, IEEE TVCG **13** (2007), no. 6, 1488–1495.

[PGCS09] Nico Pietroni, Fabio Ganovelli, Paolo Cignoni, and Roberto Scopigno, *Splitting cubes: a fast and robust technique for virtual cutting*, The Visual Computer **25** (2009), no. 3, 227–239.

[PH90] I. D. Parsons and J. F. Hall, *The multigrid method in solid mechanics: Part I–Algorithm description and behaviour*, International Journal for Numerical Methods in Engineering **29** (1990), no. 4, 719–737.

[Pop03] Stéphane Popinet, *Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries*, Journal of Computational Physics **190** (2003), no. 2, 572–600.

[PRS07] Tobias Preusser, Martin Rumpf, and Lars Ole Schwen, *Finite element simulation of bone microstructures*, Proc. 14th Workshop on the Finite Element Method in Biomedical Engineering, Biomechanics and Related Fields, 2007, pp. 52–66.

[PTSP02] Bernhard Preim, Christian Tietjen, Wolf Spindler, and Heinz-Otto Peitgen, *Integration of measurement tools in medical 3d visualizations*, Proc. IEEE Visualization, 2002, pp. 21–28.

[PWL97] Alex T. Pang, Craig M. Wittenbrink, and Suresh K. Lodha, *Approaches to uncertainty visualization*, The Visual Computer **13** (1997), no. 8, 370–390.

[RB86] C. C. Rankin and F. A. Brogan, *An element independent corotational procedure for the treatment of large rotations*, ASME Journal of Pressure Vessel Technology **108** (1986), no. 2, 165–174.

[RBE$^+$06] G. Reina, K. Bidmon, F. Enders, P. Hastreiter, and T. Ertl, *GPU-based hyperstreamlines for diffusion tensor imaging*, Proc. Eurographics/IEEE-VGTC Symposium on Visualization, 2006, pp. 35–42.

[RCD$^+$10] Ivo Rössling, Christian Cyrus, Lars Dornheim, Andreas Boehm, and Bernhard Preim, *Fast and flexible distance measures for treatment planning*, International Journal of Computer Assisted Radiology and Surgery **5** (2010), no. 6, 633–646.

[RLLT07] Maximilian Rudert, Ulf Leichtle, Carmen Leichtle, and Wolfram Thomas, *Implantation technique for the CUT-type femoral neck endoprosthesis*, Operative Orthopädie und Traumatologie **19** (2007), no. 5/6, 458–472.

[RNS06] J. Rodriguez-Navarro and A. Susin, *Non structured meshes for cloth GPU simulation using FEM*, Proc. Workshop in Virtual Reality Interactions and Physical Simulation, 2006, pp. 1–7.

[RSG$^+$08] Tobias Renkawitz, Francesco S. Santori, Joachim Grifka, Carlos Valverde, Michael M. Morlock, and Ian D. Learmonth, *A new short uncemented, proximally fixed anatomic femoral implant with a prominent lateral flare: design rationals and study design of an international clinical trial*, BMC Musculoskeletal Disorders **9** (2008), 147.

[SB10] Rahul S. Sampath and George Biros, *A parallel geometric multigrid method for finite elements on octree meshes*, SIAM Journal on Scientific Computing **32** (2010), no. 3, 1361–1392.

[SCB01] T. Strouboulis, K. Copps, and I. Babuška, *The generalized finite element method*, Computer Methods in Applied Mechanics and Engineering **190** (2001), no. 32-33, 4081–4193.

[Sch04] Steven H. Schwartz, *Visual perception: A clinical orientation*, McGraw-Hill, 2004.

[SDF07] Eftychios Sifakis, Kevin G. Der, and Ronald Fedkiw, *Arbitrary cutting of deformable tetrahedralized objects*, Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2007, pp. 73–80.

[SEHW02] Andreas Sigfridsson, Tino Ebbers, Einar Heiberg, and Lars Wigström, *Tensor field visualisation using adaptive filtering of noise fields combined with glyph rendering*, Proc. IEEE Visualization, 2002, pp. 371–378.

[Sen] Sensable, http://www.sensable.com/.

[SHS01] D. Serby, M. Harders, and G. Székely, *A new approach to cutting into finite element models*, Proc. MICCAI, Lecture Notes in Computer Science, vol. 2208, 2001, pp. 425–433.

[SKW09] Jens Schneider, Martin Kraus, and Rüdiger Westermann, *GPU-based real-time discrete Euclidean distance transforms with precise error bounds*, Proc. VISAPP, 2009, pp. 435–442.

[Sla02] William S. Slaughter, *The linearized theory of elasticity*, Birkhäuser, 2002.

[SM99] Heidrun Schumann and Wolfgang Müller, *Visualisierung: Grundlagen und allgemeine Methoden*, 1 ed., Springer, 1999.

[SM06] Thomas Sangild Sørensen and Jesper Mosegaard, *An introduction to GPU accelerated surgical simulation*, Proc. International Symposium on Biomedical Simulation, Lecture Notes in Computer Science, vol. 4072, 2006, pp. 93–104.

[SMMB00] N. Sukumar, N. Moës, B. Moran, and T. Belytschko, *Extended finite element method for three-dimensional crack modelling*, International Journal for Numerical Methods in Engineering **48** (2000), no. 11, 1549–1570.

[SOG06] Denis Steinemann, Miguel A. Otaduy, and Markus Gross, *Fast arbitrary splitting of deforming objects*, Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2006, pp. 63–72.

[SPP$^+$10] Jochen Süßmuth, Wassilios-Daniele Protogerakis, Alexander Piazza, Frank Enders, Ramin Naraghi, Günther Greiner, and Peter Hastreiter, *Color-encoded distance visualization of cranial nerve-vessel contacts*, International Journal of Computer Assisted Radiology and Surgery **5** (2010), no. 6, 647–654.

[SSG00] Scott B. Steinman, Barbara A. Steinman, and Ralph Philip Garzia, *Foundations of binocular vision: A clinical perspective*, McGraw-Hill, 2000.

[SSIF07] Eftychios Sifakis, Tamar Shinar, Geoffrey Irving, and Ronald Fedkiw, *Hybrid simulation of deformable solids*, Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2007, pp. 81–90.

[STWH00] M. Starker, P. Thümler, A. Weipert, and S. Hanusek, *Computer-assisted prosthesis selection and implantation control*, Orthopäde **29** (2000), no. 7, 627–635.

[SV02] J. A. Simões and M. A. Vaz, *The influence on strain shielding of material stiffness of press-fit femoral components*, Journal of Engineering in Medicine **216** (2002), no. 5, 341–346.

[SW00] Katrin Suder and Florentin Wörgötter, *The control of low-level information flow in the visual system*, Reviews in the Neurosciences **11** (2000), no. 2-3, 127–146.

[SW06] S. A. Sauter and R. Warnke, *Composite finite elements for elliptic boundary value problems with discontinuous coefficients*, Computing **77** (2006), no. 1, 29–55.

[SY04] Lin Shi and Yizhou Yu, *Visual smoke simulation with adaptive octree refinement*, Proc. Computer Graphics and Imaging, 2004, pp. 13–19.

[SYBF06] Lin Shi, Yizhou Yu, Nathan Bell, and Wei-Wen Feng, *A fast multigrid algorithm for mesh deformation*, ACM TOG **25** (2006), no. 3, 1108–1117.

[TCO08] Zeike A. Taylor, Mario Cheng, and Sébastien Ourselin, *High-speed nonlinear finite element analysis for surgical simulation using graphics processing units*, IEEE Transactions on Medical Imaging **27** (2008), no. 5, 650–663.

[TCRM03] Sivan Toledo, Doron Chen, Vladimir Rotkin, and Omer Meshar, *TAUCS: A library of sparse linear solvers*, 2003, http://www.tau.ac.il/∼stoledo/taucs.

[TF88] Demetri Terzopoulos and Kurt Fleischer, *Modeling inelastic deformation: Viscoelasticity, plasticity, fracture*, Proc. ACM SIGGRAPH, 1988, pp. 269–278.

[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.

[TOS01] Ulrich Trottenberg, Cornelis W. Oosterlee, and Anton Schüller, *Multigrid*, Elsevier Academic Press, 2001.

[TPBF87] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer, *Elastically deformable models*, Proc. ACM SIGGRAPH, 1987, pp. 205–214.

[TSH$^+$07] Fulvia Taddei, Enrico Schileo, Benedikt Helgason, Luca Cristofolini, and Marco Viceconti, *The material mapping strategy influences the accuracy of CT-based finite element models of bones: An evaluation against experimental measurements*, Medical Engineering and Physics **29** (2007), no. 9, 973–979.

[VCT$^+$04] Marco Viceconti, Alessandro Chiarini, Debora Testi, Fulvia Taddei, Barbara Bordini, Francesco Traina, and Aldo Toni, *New aspects and approaches in pre-operative planning of hip reconstruction: a computer simulation*, Langenbeck's Archives of Surgery **389** (2004), no. 5, 400–404.

[Wan01] Weigang Wang, *Special bilinear quadrilateral elements for locally refined finite element grids*, SIAM Journal on Scientific Computing **22** (2001), no. 6, 2029–2050.

[WB05] Andrew Wilson and Rebecca Brannon, *Exploring 2D tensor fields using stress nets*, Proc. IEEE Visualization, 2005, pp. 11–18.

[WBG07] Martin Wicke, Mario Botsch, and Markus Gross, *A finite element method on convex polyhedra*, Computer Graphics Forum **26** (2007), no. 3, 355–364.

[WDGT01] Xunlei Wu, Michael S. Downes, Tolga Goktekin, and Frank Tendick, *Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes*, Computer Graphics Forum **20** (2001), no. 3, 349–358.

[WG04] Karl-Heinz Widmer and Paul Alfred Grützner, *Joint replacement—total hip replacement with CT-based navigation*, Injury **35** (2004), no. 1, S-A84–S-A89.

[WH04] Wen Wu and Pheng Ann Heng, *A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting*, Computer Animation and Virtual Worlds **15** (2004), no. 3-4, 219–227.

[Wix08] R. L. Wixson, *Computer-assisted total hip navigation*, Instructional Course Lectures **57** (2008), 707–720.

[WKZL04] Andreas Wenger, Daniel F. Keefe, Song Zhang, and David H. Laidlaw, *Interactive volume rendering of thin thread structures within multivalued scientific data sets*, IEEE TVCG **10** (2004), no. 6, 664–672.

[WMM$^+$02] C.-F. Westin, S. E. Maier, H. Mamata, A. Nabavi, F. A. Jolesz, and R. Kikinis, *Processing and visualization for diffusion tensor MRI*, Medical Image Analysis **6** (2002), no. 2, 93–108.

[Wol92] Julius Wolff, *Das Gesetz der Transformation der Knochen*, Hirschwald, 1892.

[WT04] Xunlei Wu and Frank Tendick, *Multigrid integration for interactive deformable body simulation*, Proc. International Symposium on Medical Simulation, Lecture Notes in Computer Science, vol. 3078, 2004, pp. 92–104.

[WW80] Robert B. Welch and David H. Warren, *Immediate perceptual response to intersensory discrepancy*, Psychological Bulletin **88** (1980), no. 3, 638–667.

[YPJM07] Zohar Yosibash, Royi Padan, Leo Joskowicz, and Charles Milgrom, *A CT-based high-order finite element analysis of the human proximal femur compared to in-vitro experiments*, Journal of Biomechanical Engineering **129** (2007), no. 3, 297–309.

[YTM07] Zohar Yosibash, Nir Trabelsi, and Charles Milgrom, *Reliable simulations of the human proximal femur by high-order finite element analysis validated by experimental observations*, Journal of Biomechanics **40** (2007), no. 16, 3688–3699.

[ZB02] Leonid Zhukov and Alan H. Barr, *Oriented tensor reconstruction: Tracing neural pathways from diffusion tensor MRI*, Proc. IEEE Visualization, 2002, pp. 387–394.

[ZDL03] Song Zhang, Çagatay Demiralp, and David H. Laidlaw, *Visualizing diffusion tensor MR images using streamtubes and streamsurfaces*, IEEE TVCG **9** (2003), no. 4, 454–462.

[ZP02] Xiaoqiang Zheng and Alex Pang, *Volume deformation for tensor visualization*, Proc. IEEE Visualization, 2002, pp. 379–386.

[ZP03] ______, *HyperLIC*, Proc. IEEE Visualization, 2003, pp. 249–256.

[ZPP05] Xiaoqiang Zheng, Beresford N. Parlett, and Alex Pang, *Topological lines in 3D tensor fields and discriminant Hessian factorization*, IEEE TVCG **11** (2005), no. 4, 395–407.

[ZSTB10] Yongning Zhu, Eftychios Sifakis, Joseph Teran, and Achi Brandt, *An efficient multigrid method for the simulation of high-resolution elastic solids*, ACM TOG **29** (2010), no. 2, 16:1–16:18.