Techniques for Single System Integration of Elastic Simulation Features

by

Nathan M. Mitchell

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 11/22/2016

The dissertation is approved by the following members of the Final Oral Committee:

Eftychios Sifakis, Assistant Professor, University of Wisconsin - Madison, Computer Sciences

Michael Gleicher, Professor, University of Wisconsin - Madison, Computer Sciences

Bilge Mutlu, Associate Professor, University of Wisconsin - Madison, Computer Sciences

Court Cutting M.D., Professor of Plastic Surgery, New York University - Langone Medical Center

Mark Hill, Professor, University of Wisconsin - Madison, Computer Sciences

For my grandfather,

whose lifelong passion for teaching others

inspired me to pursue my own goals and education.

ACKNOWLEDGMENTS

I would like to thank all the people who have supported me and my work over the years. In particular, I would like to thank my family who have shown me unwavering support and my friends who have kept me sane. Also, I would like to thank Ben Recht, Ronald Fedkiw, Matthew Cong for their advice and discussions on several of the contributions made in this dissertation. Additionally, I wish to extend my thanks to Dr. Timothy King, the medical residents of the Plastic & Reconstructive Surgery program, and the School of Medicine and Public Health at the University of Wisconsin-Madison for their involvement and support with the virtual plastic surgery platform, as well as acknowledge Aaron Oliker and BioDigital Systems for supplying the models used in this project.

This work was generously supported by the following grants and funding sources: NSF grants IIS-1253598, IIS-1407282, CCF-1423064, CCF-1533885, CNS-1218432, CCF-1438992.

CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ALGORITHMS

Page

ABSTRACT

Techniques for simulating the behavior of elastic objects have matured considerably over the last several decades, tackling diverse problems from non-linear models for incompressibility to accurate self-collisions. Alongside these contributions, advances in parallel hardware design and algorithms have made simulation more efficient and affordable than ever before. However, prior research often has had to commit to design choices that compromise certain simulation features to better optimize others, resulting in a fragmented landscape of solutions. For complex, real-world tasks, such as virtual surgery, a holistic approach is desirable, where complex behavior, performance, and ease of modeling are supported equally. This dissertation caters to this goal in the form of several interconnected threads of investigation, each of which contributes a piece of an unified solution. First, it will be demonstrated how various non-linear materials can be combined with lattice deformers to yield simulations with behavioral richness and a high potential for parallelism. This potential will be exploited to show how a hybrid solver approach based on large macroblocks can accelerate the convergence of these deformers. Further extensions of the lattice concept with non-manifold topology will allow for efficient processing of self-collisions and topology change. Finally, these concepts will be explored in the context of a case study on virtual plastic surgery, demonstrating a real-world problem space where these ideas can be combined to build an expressive authoring tool, allowing surgeons to record procedures digitally for future reference or education.

1 INTRODUCTION

This document describes a set of techniques and design choices dealing with the processes of modeling, discretization, and simulation of elastic deformable solids on data structures related to regular Cartesian grids. The decision to use Cartesian grid representations for deformable bodies was made due to their ease of use and the substantial potential for performance optimizations. This document describes a series of contributions which enable the joint support of a number of desirable simulation capabilities within this design regime, and allow the performance potential to materialize in the context of real-world applications.

We use the task of crafting instructional animations of reconstructive plastic surgery as the driving application and source of motivation for this work. This domain was used to define the scope of the research presented in this document and the virtual surgery systems developed within were leveraged as testing grounds for the algorithmic and data structure contributions described in this dissertation.

Finally, this dissertation explores the practical challenges of deploying these algorithmic contributions in the context of a practical and usable interactive system, addressing engineering and deployment issues that extend beyond the details of the constituent core algorithms.

1.1 Thesis

This dissertation aims to support the following statement: Creating virtual simulators for soft tissue reconstructive plastic surgery has reached the point of technical feasibility. This capability is demonstrated using soft tissue reconstructive surgery as the benchmark, both for its intrinsic value and the degree of complexity and challenges it exemplifies.

In pursuit of this goal, this document will describe how current practices for simulating elastic materials can be combined in a holistic fashion to optimize for performance and practical usability. In the process, limitations with the current approaches will be explored and, in some cases, alternative techniques will be proposed to solve technical challenges that

our benchmark application exposes.

1.2 Motivation

Physicians have been seeking better methods to capture human anatomy and function, both normal and pathological, for the purpose of healing since the earliest days of modern surgical theory. From the anatomical drawings of Da Vinci, to more modern practices of constructing realistic simulacra, surgeons, and their students, have been pursuing tools that allow them to practice their skills before operating on real patients. Existing research shows the benefits of engaging in these practice sessions in a virtual, non-invasive setting [Gallagher et al., 2005]. Practiced surgeons make fewer mistakes and can use preparation sessions to plan new approaches safely.

This general philosophy, which can be summed up with the classic proverb of "measure twice, cut once", is practiced by many high risk professions. From flight school to driving simulators, computer constructed virtual environments have become an integral part of training highly skilled professionals. The reasoning is three-fold: computer simulations are relatively low cost and are easy to reset and reconfigure quickly, unforeseen parameters and situations can be introduced more easily than in physical environments, and a trainee's progress can be easily recorded for later review. With these advantages over purely physical training environments, why are surgeons still using aids such as diagrams, physical mannequins, and cadavers?

Part of the answer involves the reality that better, technology-assisted alternative educational aids are largely scarce and mature solutions are narrowly scoped. Performing surgery is a complex task involving a combination of dexterous and cognitive, often spatial reasoning, skills [Gallagher et al., 2005]. Tools that support all of these areas are difficult to get right, and most attempts to build technological aids have focused on subsets of the skills required. Historically, these have been the dexterous skills, which many authors have

tried to solve with a variety of haptic simulation techniques [Mendoza and Laugier, 2003, Lindblad and Turkiyyah, 2007]. While these surgical simulation philosophies are useful, and have been used in commercial products [Simbionix USA Corporation, 2002–2014a], they don't really meet the need of training cognitive skills. This need varies across surgical specialties - reconstructive plastic surgery, which is highlighted in this document, requires the surgeon to have internalized geometrical intuitions in order to manipulate tissue in the three dimensional space of the human body. The lack of tools supporting this type of knowledge has kept traditional, less technological aids as the core of many plastic surgery training programs.

In comparison to internal surgery, plastic surgery is also challenged by the practical reality that the results of any operation will be visible to others. This fact adds an additional constraint onto practitioners; not only must their work be as technically correct as before, but they must also be considering the final aesthetics of their procedures. It follows then that a simulator for plastic surgery operations must provide an environment for surgeons to freely practice design, as well as correctly display the outcomes.

1.3 Contributions

General Material Support We present techniques for accommodating general classes of materials, including nonlinear mechanical properties, and anisotropic media such as muscles, in the context of a Cartesian grid-based discretization. This generality is supported despite the challenge of simultaneously accommodating additional simulation constraints such as parallelism, topology change, and collision handling.

Hybrid Grids for Non-Manifold Embedding We demonstrate an augmented data structure that enables the resolution of thin, sub-voxel material features within an otherwise standard hexahedral grid embedding context. This is accomplished by combining the implicit topology of a grid and the flexibility of an explicit mesh structure, creating a hybrid data structure that has large performance potential and modeling versatility. We demonstrate this

data structure in the context of surgical operations with complex, thin incisions created by user input.

A Thread- and Vector-conscious Parallelization Framework We developed a programming paradigm and an object-oriented code infrastructure for bridging the performance divide between hand-optimized numerical kernels and what compiler optimizations were able to provide in the context of complex simulation tasks. This framework drastically simplifies the developer's effort, generating highly optimized SIMD code while presenting a API resembling scalar-style semantics. We demonstrated this framework by constructing kernels for elastic simulations, showing how even large kernels can be successfully vectorized without suffering inefficiencies of automatic compiler vectorization.

Non-Manifold Level Sets We propose a data structure for discretizing a level set over a non-manifold domain, allowing the capture of implicit geometry with zero width incisions and overlapping regions. Additionally, we provide algorithms for important tasks, such as locating the nearest surface location from an interior point, enabling the use of the data structure in self collision scenarios for elastic simulation.

Macroblock Solver Design We designed a hybrid iterative-direct solver for elastic materials defined over hexahedral grids, which divides the domain into self-contained abstractions of simulation elements, labeled macroblocks. The interior of each macroblock is solved in a direct fashion, using a cache-friendly, hierarchical factorization approach, while the interfaces between macroblocks are solved iteratively. This technique provides excellent convergence for non-linear materials, inheriting robustness properties of direct solvers, while remaining fast and tunable, like iterative solvers.

Deployment Methodology for Remote Simulation In order to facilitate easy and cost-effective deployment and collaboration, we developed a prototype surgical simulation

system that combines lightweight front-end clients with specialized remote simulation servers. By employing modern web technologies, we are able to support cross-platform, multi-user shared simulation environments over the network. This approach provides good scalability across multiple clients, reduced infrastructure costs, and better long term maintenance options.

1.4 Cartesian Grids as Model Representations

Capturing the shape of deformable models in visual computing applications has been accomplished in conjunction with a variety of geometry representations, including tetrahedral meshes, point clouds, cages, and grids. This last method, which includes Cartesian grid representations, has a number of benefits, including a simple and procedurally defined topology along with an excellent potential for performance optimizations. Yet, these advantages come with important caveats:

- Grid based representations form only an approximation of the object's surface.
- Although the use of grids provides regularity at the data structure level, irregularity can also manifest in other ways, such as the heterogeneity of material properties, especially in models inspired by anatomy.
- Many of the ways that model topology might be required to change in scenarios of cutting or fracture can jeopardize the regular implicit topology of the grid data structure.
- Several established methods for collision handling are not optimized for grid-based representations, and even less so for the circumstances that can emerge from topology change.
- Finally, translating the *potential* for performance in grid based representations into *practical gains* in a fully-featured interactive system is a nontrivial proposition.

Handling all of these concerns simultaneously is the essence of the challenge at hand. The work presented in this dissertation describes methods for addressing these issues. In particular, techniques for infusing additional topological flexibility into grid representations, allowing them to effectively capture thin, sub-cell material, while not giving up on performance opportunities will be covered in Chapters 5 and 6; these chapters also address collision processing within the same framework. In order to capitalize on the performance potential, the regularity of the grid based representation will be used to build efficient streaming kernels in Chapter 6. Additionally, higher level abstractions, described in Chapter 7, are possible: grouping together multiple cells and creating larger macroblocks for improved performance and to more effectively capture nonlinear behaviors.

1.5 Parallelism Concerns with Modern Hardware

Modern hardware and modern simulation techniques are currently intersecting with a high level of maturity on both sides. This brings the possibility of being able to run large simulations on commodity hardware at near real-time rates, something once considered impractical. However, simple reuse of existing simulation implementations on current hardware does not always result in competitive performance. Algorithms and techniques must be adapted to the underlying performance mechanisms in modern computational hardware architecture - namely thread and vector based parallelism. Each of these mechanisms carries its own caveats and idiosyncrasies which must be accounted for in order to gain the most benefit. Unfortunately, despite the advances in simulation capability and behaviors, less attention has been paid to these parallelism concerns, often leading to algorithms which are poorly structured to take advantage of both forms of parallelism simultaneously. Although for some application domains the performance divide between platform-specific and platform-agnostic algorithmic development might be limited, in the case of performance-conscious iteractive applications, such as the clinical domain we target, the wasted performance opportunities are

too severe to ignore.

This concern is only complicated by the fact that current arithmetic bandwidth, expressed in operations per second, is about two orders of magnitude greater than the available memory bandwidth on most platforms. This impacts a large majority of low-level numerical algorithms that deformable simulations are built on; as a consequence, the source of inadequate performance in interactive simulation scenarios is often traced in our work to imbalances in the memory-computation mix of the underlying algorithms. Thus, many of our interventions that prove most effective in restoring interactive performance focus on the efficiency of low-level throughput-sensitive kernels.

The work presented in this document attempts to address these issues in two key aspects. First, a framework for building vectorized and multithreaded numerical kernels is presented in Chapter 6. This approach delivers a scalar-like API to the developer, while being able to generate efficient parallelized code for multiple architectures and architectural widths automatically behind the scenes. This allows developers to focus on algorithmic correctness, while not loosing the benefits of vectorization. The second intervention is the development of a solver for local neighborhoods of grid cells (macroblocks) as discussed in Chapter 7, which uses a delicate mirroring pattern to both expose large amounts of vectorization friendly computation and to keep the memory bounds of the operation within first level processor caches. This creates a lower demand for memory bandwidth, at the trade-off of additional computation, improving upon the memory-computation imbalance found in modern hardware.

1.6 Practical Deployment of Surgical Simulations

Surgical simulation has been an active area for deformable solid research due to its intrinsic value as well as its potential as a catalyst for algorithmic innovation. Unfortunately, some of the most promising results from basic research are confronted with complex challenges upon attempted integration into a comprehensive real-world system. Commercial surgical

simulation tools have fared better, but they are often restricted to expensive and bulky workstations, only support a small number of simultaneous users, and provide a limited feature set.

The work in this document attempts to address these issues by exploring methods for practically deploying surgical simulation software to a wider audience of users. This problem has been tackled along two fronts. First, we have attempted to build fast and interactive simulations using commodity hardware, while not compromising on simulated features like collisions and nonlinear materials. This allows for a wider range of platforms to be used and not restricting the system to exotic, specific requirements. Second, we have explored a wide range of implementations and approaches for delivering software to users. These include both local and remote simulation designs, making use of modern web technologies, and taking careful look at third party library implications. This approach was demonstrated in a live pilot deployment of the surgical simulation software for medical students. The details of this development strategy are covered in Chapter 8.

1.7 Outline

What follows is a short outline of the remaining chapters within this dissertation. In Chapter 2, the motivation for this document is dissected in more detail, exploring both technical domain considerations and design philosophies. A technical deconstruction of basic deformable solid simulation practices is covered in Chapter 3, along with an introduction to notable technical challenges. An examination of related work is covered in Chapter 4, placing the contributions of this document in a broader context. Chapter 5 demonstrates how the regularity of Cartesian grids can be combined with desired amounts of topological flexibility, both for capturing model geometry and for contact scenarios. Chapter 6 continues with a discussion around thread and vector based parallelism in the context of Cartesian grids. Chapter 7 describes a new method for solving for elastic deformations which constructs larger macroblocks for

better convergence behavior. Chapter 8 describes practical deployment concerns and delivers a critique of the various implementation options for surgical simulation systems. Chapter 9 concludes this document with a final look back at completed work and considers future work along with current shortcomings.

2 MOTIVATION

Computer aided tools for plastic surgery serve as the external motivation for the work presented in this document. In this chapter, this idea will be explored more thoroughly. The goals of this chapter are twofold: first, we will formalize and review the concept of plastic surgery simulation by studying the various aspects and requirements of such systems. As we will see, there exist many subtleties of both plastic surgery training itself, and the task of simulating it on a computer. Second, this chapter will attempt to bring these ideas into a general conceptual framework, showing how they can be used in more areas beyond surgical simulation and how these systems can drive research. Combined, these two discussions will motivate the importance and utility of the technical contributions presented in later chapters.

2.1 Medical Simulation: Requirement Specification

When exploring computer science research with a domain specific focus, such as we are doing here with plastic surgery simulation, it is critical to understand the expectations of domain experts. Not only do these expectations guide the development of software, but they can help triage research goals and serve as domain appropriate metrics for the evaluation of final results. In this section, we will explore the specific requirements of our domain, specifically soft tissue plastic surgery, while keeping practical engineering considerations in mind. In the process, we will begin developing a framework in which these requirements can be placed and will be explored further in later sections.

To accomplish this goal, we need to answer a series of questions that define the scope of our proposed tools, as well as its metrics for success: What is the potential real-world utility of the tool? How well does the system accomplish the task it was designed for? What are the fine grained aspects of the required tasks? What features need to be implemented (or perhaps more interestingly, not implemented)? Who are its users? We will answer these questions by looking more narrowly at the problem of medical simulation for soft tissue operations which

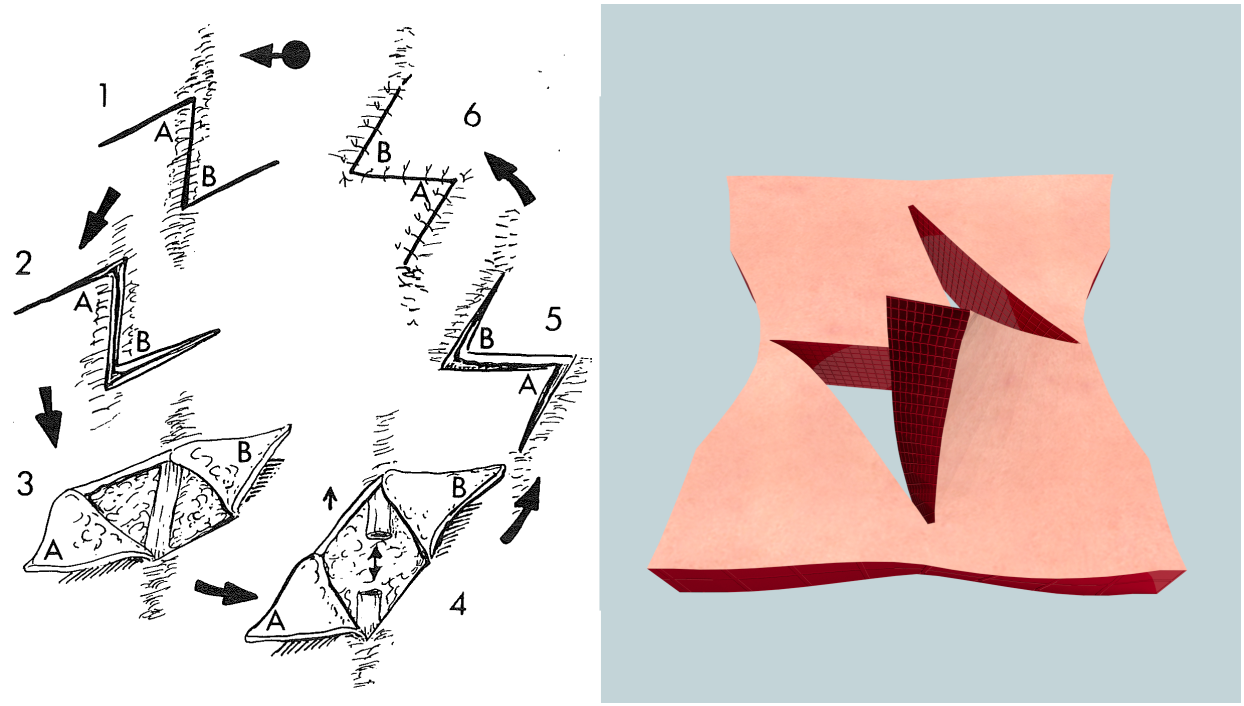


Figure 2.1: Z-plasty: Comparison between textbook illustration and simulation
A comparison between the classic *Z-plasty* operation from a standard textbook [McCarthy, 1990] (left) and a simulated version of the procedure in three dimensions (right). The simulation allows for more of the three dimensional shape of the procedure to be appreciated.

we have chosen for our benchmark.

For the purposes of this document, the range of soft tissue operations under consideration will be restricted to "local flap" procedures [Baker, 2014]. These procedures locally alter the geometry of skin tissue, but do not include any alterations to underlying bones or muscle layers. Nor do they involve any remote effects, such as tissue grafts. These procedures are fundamental building blocks of more complex operations, but, due to their often non-intuitive geometry, can be difficult to learn. In fact, these procedures often have natural analogs in computer graphics and geometry.

Graphics practitioners would associate these elementary actions with geometric transforms: shear, rotation, uniform or anisotropic scaling. Of course, applying such transformation on live tissue is very different than their application on a geometric model. When stretching a tissue patch in one direction to 125% of its original length while contracting it in the

transverse direction down to 80%, squeezing-and-stretching in-place is typically *not* the desired way to execute this transform. Real skin might not stretch that far or buckle in the transverse direction during the process. A maneuver called the *Z-plasty* (see Figure 2.1, right) achieves the same net effect, with a more graceful stress distribution and a smooth blend to the surrounding tissues. In plastic surgery, *cognitive training* addresses the mental challenge of how these elemental surgical puzzle-pieces are individually designed and how they can be assembled into larger, more complex operations. Unfortunately, 3D computer-based cognitive training solutions for facial reconstructive surgery are virtually nonexistent; common educational materials are limited to 2D sketches (see Figure 2.1, left) and still photographs of procedures.

Surgical skills targeted by computer-based training solutions have been classified Gallagher et al. [2005] in two major categories. Psychomotor skills refer to the dexterous use of the surgeon's hands to manipulate instruments in the course of an operation. In plastic surgery, psychomotor training involves mechanical aspects of surgical tasks, such as the "feel" of tissue being cut or the nuances of manipulating a scalpel to enact a curved incision. For example, training for laparoscopic procedures requires a clinician to be familiar with the tactile response of pushing and pulling on organs and to practice coordination skills required for suturing and cauterization. A number of computer-based solutions focus on psychomotor training, including the works of Mendoza and Laugier [2003], De et al. [2005], Kim et al. [2007], Lindblad and Turkiyyah [2007]. In contrast to psychomotor training, *cognitive* skills and training are largely mental rather than dexterous exercises. For example, in the procedure shown in Figure 2.2, the surgeon needs to contemplate how to best repair a large square skin defect (i.e. area of excised tissue) by making auxiliary incisions that create properly shaped "puzzle pieces" which can be sutured together without creating excessive stress. As an example of a cognitive training system, Chentanez et al. [2009] described a cognitive training system for steerable needle insertion, where the mental challenge lies in planning a sequence of actions involving needle flexion and torsion, in order to achieve a desired insertion

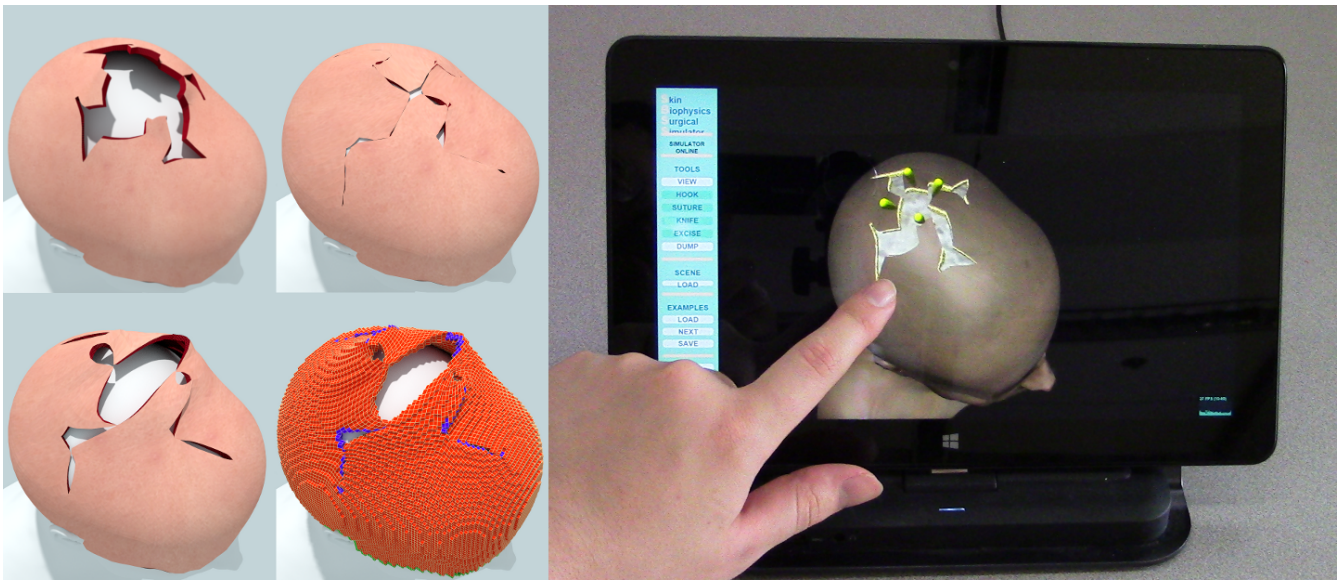


Figure 2.2: Simulation of an advanced surgical repair at the top of the scalp
Simulated *Dufourmentel-Mouly* repair (see Baker [2014]) for a large gap of excised tissue on the scalp. From top to bottom, left to right: Rendered stages of procedure, embedding lattice, real time demo on a tablet running in a web browser.

trajectory.

Focusing on any individual type or aspect of skill training affects the design decisions of a proposed tool. For instance, building a system for psychomotor training likely places a greater emphasis on the interactivity aspects of the system, potentially requiring haptic feedback mechanisms to provide tactile sensations. Likewise, a cognitive training aid might require more involved reactivity and dynamism, such as a more accurate representation of tissue behaviors. Ultimately, our goal should be to construct a virtual aid which resembles reality as much as possible, in order to support all training goals. In our pursuit of this gold standard, we need to take the opportunities as they arrive to target particular aspects of surgical training. In this case, we find ourselves in a position to handle important cognitive training tasks, even if we have to compromise on psychomotor skills.

This document will present techniques for supporting a cognitive simulation for plastic surgery. In particular, the goal is to support an authoring tool which supports the following basic tasks: design local flap operations on interactive, virtual tissue models, record and playback operations for the purpose of knowledge retention and review, and support limited

training potential. It is not the immediate goal of this work to present a training tool in which user's actions are graded, nor do we claim a predictive model, where all tissue responses are accurate enough to use as a basis for real surgery decisions. Instead, the tool should simply allow an experienced practitioner to capture their knowledge of procedures in an interactive, visual environment and to pass that knowledge onto others.

In order to further refine the ultimate shape of this proposed simulation aid, we will now look at five important aspects as they relate to plastic surgery simulation: Geometry, Reactivity, Dynamics, Interactivity, and Utility. In the following section, we will see how these ideas can be generalized.

Geometry Surgery in general, plastic surgery in particular, are areas in medicine which depend strongly on a practitioner's intuitive spatial reasoning processes. At a high level, the task of a plastic surgeon is to manipulate the geometry of the human body into a new configuration. The reasons for this style of intervention are numerous and range from cosmetic procedures, post-operation repairs, and fixing congenital deformities. Contrary to popular public perception, plastic surgery is often employed to enact functional repairs. Those living with injuries and deformities often have difficulties because the specific geometry of their condition adversely affects the proper functioning of their body. A surgeon must understand the geometry of their patient in order to produce both functionally correct, but also aesthetically pleasing, outcomes. Since the results of these operations are often highly visible, failures or mistakes can be extremely costly in an emotional and social sense for the patient, let alone a continued disability. Any tool designed for this domain must take these concerns seriously and present a compelling visual representation.

The work presented in this document achieves this goal by describing a tool for visually authoring plastic surgery operations in a three dimensional virtual environment. However, it is worth taking time to consider some potential alternatives. For instance, a two dimensional sketching interface could be considered, drawing on the rich history of surgical instructional

diagrams that exist in the field. Certainly such approaches have been used for decades in surgical textbooks, but they fall short of giving the viewer a comprehensive perspective of the inherent three dimensional nature of human anatomy. Alternatively, one might consider using video of real operations, annotated with information describing what is happening. While this idea certainly leaves no question about the three dimensionality of the problem, it does so by sacrificing the clarity a rendered computer model can provide and the ability to alter the geometry easily.

A note should also be made on display technology. Here the word "display" should be treated loosely and should be taken to mean any physical channel which conveys information from the tool to the user. Commonly, this is restricted to visual interfaces, such as computer monitors or head mounted displays (HMDs). However, it could also refer to force feedback devices, where a user receives tactile responses from their actions in the system. Ultimately, the proper device choice for a surgical simulation aid depends on its intended purpose. Many commercial tools for training surgeons on laparoscopic equipment use a tightly coupled visual and tactile feedback system. The goal of these tools is to replicate operating room conditions exactly, hopefully instilling into physicians the tacit knowledge required.

Reactivity and User Interaction The display of three dimensional, static tissue geometry is an improvement over the two dimensional illustrations commonly employed in training literature. However, in order to provide surgeons more realism, we also need to have the models react to their inputs. In real surgeries, there exist many ways a surgeon might interact with human tissue: injecting drugs, cutting it, applying suction to remove blood, and suturing it together are all candidate interactions. For the purposes of this document, and to focus on the cognitive aspects of manipulating tissue geometry, we will be focusing on three types of user interactions: pulling on tissue, joining tissue together, and changing its topology. These concepts map directly to surgical tasks, namely using surgical hooks to pull tissue around, adding sutures to close wounds, and cutting into tissue with a scalpel. This

subset of interactions, while small, is still capable of expressing many interesting operations, particularly in regard to the cognitive problem solving scenarios we wish to target. The ability for users to cut, pull, and join enables them to explore arranging the complex tissue geometries required for local flap procedures. We then use physical simulation to compute a mechanical response to these user inputs, which can be visualized by deforming the geometry of the tissue model.

By creating a cause and effect relationship between the user's input and the tissue's shape through simulation, we can provide users an environment in which to learn more about how their proposed procedures might perform in reality. By leaving the simulation flexible, we can also allow for later integration of realistic biomaterial behaviors. These materials are extremely complex and often behave in non-intuitive, time dependent fashions. While their accurate simulation is beyond the scope of this document, we want to not exclude the possibility of including them later.

Dynamics It is worth discussing at this point the nature of *time* with respect to our proposed tool, which can often be non-intuitive. The real-world phenomena we are representing are heavily dependent on time in two major ways: sequencing and lag-based effects. Sequencing refers to the nature of cause and effect, specifically the relative ordering of events and the time taken between them. In the real world, we can generally think of the allowed user actions (i.e. cutting, pulling, joining) as being naturally ordered and smoothly continuous in time. For our purposes, we will relax this concept in our benchmark tool to allow for discrete events. While we maintain that user actions should remain ordered, as we wish to track the steps of an operation, we will record these actions as a series of discrete states. This is not fundamentally different than the idea of discrete stages of an illustrated operation, as we saw in Figure 2.1 earlier. As we transition between input states, the simulation will gradually converge to a new mechanical response, creating the illusion of dynamic motion.

However, we need to be careful to distinguish between this illusion of motion and real

dynamic lag effects. These effects are commonly expressed in real materials as jiggling, but can also be seen in surgical contexts in the form of swelling or other physiological reactions to tissue damage. We will be specifically avoiding these types of dynamic behavior in our tool for several reasons. First, the models describing these behaviors are extremely complex and delicate. Moreover, the right answers to these problems are not clear yet in all cases. Second, even if we had a well defined model, computing dynamic effects is computationally expensive, which could harm the interactivity of our tool. Finally, we don't really need perfect resolution of these effects to initially accomplish our goal of cognitive training, which involves more careful planning rather than immediate reactions to dynamic effects.

Utility The final topic to discuss is the planned utility of the tool, specifically the expected use cases and deployment concerns. As mentioned previously, our goal is to support users in authoring scenarios for local flap operations. The planned modes of interaction are two-fold. First is the authoring mode, where an experienced user will interact with a three dimensional, simulated tissue model to sequentially plan a procedure. This will be accomplished by using the mouse and keyboard to select tools (i.e. cutting, pulling, and joining) and apply them to the surface of the model. As each tool is applied and recorded, the simulation will update providing the user with the results of their last input. The second mode is the playback, or viewing, mode. In this mode, the user, potentially a student, will select a pre-authored procedure from an available database. At this point, the user will be allowed to advance through the procedure one step at a time, where the steps are the prerecorded inputs from the previous authoring mode. During both modes of operation, the user will have freedom to move the model around in virtual space, allowing them to see all aspects of the manipulation at every stage.

In addition to the two modes of operation, we want the tool to support a multi-user shared environment. This will allow the tool to support scenarios such as classroom settings or shared presentations. Under this concept, multiple users will be able to connect clients

to a remote simulation server and observe the same scene. Each client will be permitted their own independent view point, allowing the same freedom to look at the deforming model as before. However, the simulation will be shared and the effects of any input entered by any user will be propagated to every other connected client. The remote simulation server should be designed to run as one of the clients or as a wholly separate device, allowing it to be customized for additional performance or upgraded later. In comparison, the user clients should be lightweight, allowing for easy deployment across a wide range of available platforms.

2.2 Simulation Assisted Visual Systems

While assisting the plastic surgery community, by improving their available methods for teaching and preparing for surgery, through computer simulation is certainly valuable, additional value can be gained by understanding how the previous section's concepts generalize to a larger class of applications. This generalized class can be referred to broadly as Simulation Assisted Visual Systems (SAVS). These systems are characterized by their *use of both physical simulation and interactive visuals to support practical tasks.* Such visual systems are commonplace in our modern society, though we do not often think of them in these terms. Concrete examples include video games, animation tools, and virtual avatar systems, though many more could be named. The goal of this section is to define the basic concepts underpinning these systems, particularly in respect to physics based simulation. This deconstruction will be followed by a examination of the challenges that exist when developing a such a system, along with a short defense of their worth beyond their targeted domains.

SAVS Deconstruction

The core of a SAVS is a virtual world, or environment. When discussing a SAVS, we are talking about artificially constructed settings, often completely described by a computer program and displayed via some device. This is contrast to other simulated environments which are built entirely in the real-world, such as police training courses for example. However, a SAVS is not completely disconnected from reality. In order to support a user in a particular task, a SAVS is often designed to mimic real life situations. Returning to the definition above, is it appropriate to declare any virtual environment a SAVS? Not exactly - building on their proposed definition, we can structure a SAVS according to a series of important properties, or *aspects*. These aspects should be familiar from the previous section: Geometry, Reactivity, Dynamics, Interactivity, and Utility. However, now we will demonstrate how these aspects are expressed in a more generic sense instead of the surgical context from earlier.

Geometry The Geometry of a SAVS refers to anything within the system dealing with shape or visual appearance. This is an intentionally broad concept, including geometry representation, texturing, and how these topics can be used functionally within the system. Ultimately, the greatest challenge with the geometry aspect is our own desires. Like the legend of King Midas, we must be careful to reign in our never ending need for more detail, in the form of more refined meshes, higher resolution textures, or higher resolution discretizations, or we run the risk of negatively impacting the overall system. Visual detail comes with steep costs, whether in the form of computational time required to render images, the storage space hold the information, or in the time required by artists to manually create assets. However, improved visuals are not simply an optional feature. If we give them up for the sake of performance or simplicity, we risk being unable to appreciate the results of the system in a satisfying manner. Worse, we might adversely affect the system's simulation components by producing discrete models too coarse to capture the important continuous mechanical properties. In the end, visual detail needs to be balanced between our wants and what we can

afford, while paying close attention to what we choose to comprise as the result of the choice.

Reactivity Reactivity describes the ability of a SAVS to hold state and allow its users to change this state meaningfully via interactions with the system. In terms of simulation, reactivity also captures the concept of boundary conditions, material models, and contact behaviors. In the benchmark application described earlier, our boundary conditions were limited to user defined hooks or point-to-point joins. In general, the types of boundary conditions vary - from spatial spline curves [Setaluri et al., 2014b] to attaching virtual bones to simulated muscles [Patterson et al., 2012, Mitchell et al., 2015]. The primary challenge with boundary conditions is determining how to best translate the intuitive control we are used to in the real-world into simulation friendly alternatives. It is easy to demand the ability to push on a squishy simulated object, but reconciling that specification with the specific discretizations employed or remaining robust throughout the process can be challenging. Similarly complex are the available material models for simulated objects. While a fairly standard set of materials were chosen for the surgical application, in order to facilitate performance and robustness, more exotic models exist that exhibit properties like plasticity and viscoelasticity [Wojtan and Turk, 2008]. Additionally, contact handling is a large part of reactivity - the choice of including it or not, and whether or not collision free guarantees are required can dramatically affect the results of a simulation. The choice of including some or all of these features into a system can provide more control on the part of the user, but it is important to avoid adding so much that the performance or robustness of the system is negatively affected. Instead, these reactivity features should be selected for the utility they bring to the system.

Dynamics As we discussed earlier, Dynamics refers to the effects of time in a system. For our surgical simulation benchmark, we have chosen to do without many dynamic behaviors to improve robustness and performance. However, dynamics are often an important property to retain in other use cases. For instance, dynamic jiggling of soft material can be highly desirable

in character animation, where lifelike motion is often closely linked with this highly visual behavior. Advanced animation use cases might require non-linear or anisotropic damping models, depending on the underlying material composition and character design [Xu and Barbič, 2017].

Interactivity Interactivity refers to two concepts: the overall speed or performance of the system and the methods that a user has to interact with it. With regards to simulation, we can talk about the concept of a time-sensitive simulation. Under this regime, we have a limited budget of time before we must display a new visual frame to the user. Video games are a classic example of this idea, where high frame rates are desirable to provide a smooth and immersive experience. Simulations used in video games are under extremely tight constraints in terms of time per simulation step [Parker and O'Brien, 2009]. This problem also be thought of as lag, or the time between when a user performs and action and when they see a change from the simulation. This issue only becomes exaggerated when introducing remote simulations, where the network becomes an additional lag source in the system. Video games have generally handled this problem by locally predicting game state into the future and interpolating new information as it arrives, but this technique is difficult for simulation based systems as good future predictions are on the same order of complexity as the simulation itself.

The other side of interactivity is the user controls. For simulation, control is generally achieved via user manipulable boundary conditions, potentially creating additional reactivity requirements within the system. But in general, user controls exist to either serve a technical function, such as the abstract manipulators in geometric modeling environments, or to improve the immersiveness, such as the hand and body tracking technologies being explored in video games. The proper choice of input mechanisms depends on the intended use of the system and can greatly shape a user's impressions of the system.

Utility We touched on some high level examples of Utility earlier, citing video games and animation tools, but in general the reason we use simulation in a SAVS is to impart some aspect of mechanical realism into a virtual environment. The term mechanical is being used here to refer to objects which behave according to physical laws, not that we are restricting ourselves to only simulating machinery. And certainly there are many applications which benefit from these behaviors. Video games can use simulation to create procedurally destructible environments [Parker and O'Brien, 2009] and animators use modeling tools to create lifelike character motion through flesh simulation [McAdams et al., 2010]. More complex simulations are regularly used to predict mechanical behaviors in advance of manufacturing, saving both time and money during development.

SAVS Challenges

Designing a Simulation Assisted Visual System comes with a wide array of challenges, which is not especially surprising when considering how many components can be included into their framework. In this section, some of these challenges will be reviewed, hopefully to provide better context for the design decisions that were made for the rest the work presented in this document.

No One Size Fits All?

Reusability is commonly described in software engineering as an important design concept, but comes with a particular set of challenges in a SAVS. In general, the idea is that reusable components in a software system are desirable as the work required for their creation can be amortized among multiple client systems. Designing a SAVS imposes some interesting roadblocks for the principle of reusability.

The first issue that often comes up is that the requirements for a SAVS, while they can appear similar in structure (e.g. display a 3D environment, respond to user input, simulate materials, etc.), are often implemented with specific optimizations due to the tight restrictions

placed on such systems (e.g. near real-time performance, extremely complex environments, etc.). Developers faced with these issues can easily fall into the trap of *blind optimization*. This is a form of anti-pattern [Rising, 1998], where they optimize the implementation, often quite expertly, for some aspect but without considering the rest of the system, or later reusability. It is important to distinguish this from *premature optimization*, where the developer spends time optimizing an implementation before knowing if such effort is required. Blind optimization is performed under local justification. It is only with a broader context that it can be determined to be a wise course of action. Premature optimization may end up being wasted effort at best, detrimental at worst. In a simulation context, these activities can result in rigid components, such as material models or solvers, which deliver high performance but are not adaptable or easy to swap out for new functionality. These are issues the work in this document has attempted to avoid.

Even if we ignore optimization, the fundamental design choices baked into simulation systems can make them difficult to reuse. Often, in the pursuit of supporting a particular physical property (e.g. strong incompressibility), choices are made in the mathematical formulation or data structures that preclude other properties from easily coexisting. The challenge here is in identifying these restricting choices early, either to avoid them or to understand their effect on potential future development activities. Ignoring the consequences can easily lead to situations where an implementation is abandoned simply because it will not function well with other techniques, instead of any inherent mistake or flaw in its own design.

Designing general purpose SAVS platforms is not impossible. Game developers, over a long developmental history, have created many excellent general platforms for game development, referred to as game engines. But increasingly, these platforms, and the developers who design them, are becoming a field in their own right. In the past, a game developer might have done everything from writing low level graphics code to higher level game logic. In contrast, modern games are often written by developers who know little about the low level optimizations required to reach the fidelity and performance expected by current audiences.

Instead, these skills are expressed in game engines - highly tuned, carefully optimized systems which are not a game per se, but act as a solid foundation for games written on top of them. They provide *services*: rendering, resource management, network support, user interface toolkits, and much more. In essence, this divide is not dissimilar to that of applications and operating systems.

The work completed in this document generally follows this philosophy. As will be described in later sections, the systems presented in this work adhere to two general principles.

1. **Avoid Uncalled for Optimization** During implementation of techniques presented in this document, care was taken to avoid optimizing too soon. By doing so, we avoided excessive specialization until the design goals could be cleared stated. In fact, many design goals are still unclear, so a maintainable, if not fully optimized, system can be an advantage.

2. **Enforce Clean Separation** As will be discussed in more detail in Chapter 8, the domain specific motivation, plastic surgery, was separated as much as possible from the underlying enabling technologies. This allowed a coupled, but functionally isolated, system design, where the components needed to build a plastic surgery simulation were isolated from the components needed to build a high performance finite element simulator.

Realism and Cruciality

For the simulation underlying the virtual environment of a SAVS, one primary concern is the balance between the concepts of realism and cruciality. Realism is a direct measure of how much a user of a system sees what is being presented as an accurate simulacra, in terms of the mechanical behavior, of a real object or environment. Here we are making an assumption, which should be intuitively justifiable, that real world mechanics are the highest standard of correctness possible. If we were capable of simulating these behaviors

with perfect accuracy, all users should be satisfied with the results. And yet, the nature of our simulations fundamentally requires that they are an approximation of reality. One goal of simulation research and building a SAVS is dealing with the issue of cruciality, the process of identifying which features, or behaviors, of reality are most crucial to supporting the primary, user-oriented tasks of the system. Every application has a different set of crucial features which help define it uniquely. For the purposes of cognitive surgical training, we are interested in simulated behaviors including smoothness of motion, maintaining surface details, supporting contact scenarios, and inextensiblity of materials. Real tissue, in contrast, supports many more behaviors such as incompressibility, viscoelasticity, and material failure under high strain - but these are not crucial to our task. Attempting to support them in order to improve realism both takes away developmental resources and unnecessarily increases complexity.

Design Conflicts

A major problem facing the construction of any SAVS is that of design goal conflicts. This is a common software development problem, where supporting one feature or aspect of a system, such as performance, comes into direct conflict with implementing another feature. Continuing with the performance example, suppose we wanted to impose a strict amount of time between visual updates to the user. By doing so, we have restricted our updates with an upper bound on the maximum amount of work they can complete at any one interval due to these time restrictions. This choice may bias further choices towards the use of other techniques, not because of any technical merit, but due to computational complexity.

Many of these conflicting goals exist, some of which are well known in general software design circles. In the context of physical simulation, there are several conflicts we need to be especially aware of.

Lag Vs. Accuracy Before, we touched on the ideas of reactivity and interactivity when discussing the nature of cause and effect within a SAVS, but related idea is known as lag. Lag is defined here to be the time between when a user applies some change to a simulated system (a force impulse, a constraint change, etc...) and when the user sees the result of the action[1]. The smaller this time, the more reactive the simulation *feels*. We can see this when comparing the simulation to real materials. For real objects, the lag is effectively zero.

Of course, real materials have an advantage that simulated materials do not. Because they are composed of individual atoms, real objects act as a nearly continuous finite element simulation, where the elements are almost infinitely small and operate completely in parallel to each other. Computer simulated materials are much coarser in their resolution and, despite great advances in parallel processing, do not come close to that naturally available in real materials. As we increase a simulated object's resolution in order to capture more and more detail, or use more complex elements that capture more interesting macroscopic effects, the overall lag of the simulation increases as more effort is spent resolving each user action.

The challenge is to find the appropriate balance between the desired lag of a simulation and the accuracy of the simulation. A major focus of this work has been to explore how both of these aspects can be tackled simultaneously, both by exploiting underutilized parallelism opportunities and by looking at novel data structures to extract additional effective resolution without significantly doing so.

Domain Utility Vs. Generality Another two ideas that often find themselves in conflict for physical simulation systems are the concepts of domain utility and generality. Lets look at domain utility first, as it's the more straightforward of the two. In the simplest terms, domain utility refers to design choices in a system that primary serve the specific task, or domain, that it is currently being built for. This may refer to choosing or discarding certain features, deciding what API best suits the current task, or making optimization along critical

[1]This should be contrasted with the concept of hysteresis, where the effects of a stimulus naturally lag behind the cause. What we are talking about with lag is an *artificial* delay which stems from algorithmic or computational artifacts.

paths for the client application. All of these choices can be reasonable, even correct, as long as you never intend to reuse the system for any other purpose.

Generality, on the other hand, asks what is the commonality of different tasks and guides design choices along this route. A general design should be flexible to different and changing requirements. Such a system typically eschews APIs built for specific tasks and instead tries to distill out the fundamental building blocks that any potential client may need from the system. The difficulties with this philosophy are two-fold. First, it isn't always obvious what the fundamental interfaces are, partially because designers by necessity must look at past applications to define them and can only make educated guesses about future ones. But secondly, general designs often cannot make the simplifying assumptions allowed by having domain specific knowledge. These designs are often left with an awkward choice between overly complex code that tries to optimize for every use case or simpler universal code which doesn't optimize anything.

On the surface, general systems seem harder to build effectively and often don't result in well optimized solutions, impacting other aspects such as interactivity, potentially leading people to question the entire effort. The short answer in response is *flexibility*. For a well studied domain, where every last detail is known and accounted for, a specialized system is probably the best choice. But in order to answer challenging research questions, tools and problems often have to change quickly and in unexpected directions as researchers adapt to new findings and explore new ideas. Medical simulation is very much one of these areas, where new questions are constantly arising and old preconceptions are abandoned. As such, during the implementation of the systems described in this document, considerable effort was spent on creating generalized simulation systems, and attempting to identify which areas are ready for optimization and which were not.

A Rationale for Developing a SAVS

Despite the complexity involved in designing and implementing SAVS type systems, the practice of developing them has significant benefits both inside and outside of academic practice. This can be seen when comparing them with the traditional academic development model. Designing isolated experiments for academic pursuits in computer science is a time tested approach for performing research. This methodology is designed to control for unknown factors during experiments - certainly this is what other scientific disciplines teach as the proper approach. However, there can be significant value in building a large system as the primary research platform. In order to support this claim, let us look at the three avenues by which a SAVS creates value: As a Catalyst, By Filling a Need, and Intrinsically.

Catalyst for Advances

Reasoning about a SAVS is a complex task, as is the subsequent process of implementation. In going about this process however, we have the potential to discover the unexpected. Anytime we have to adapt a SAVS to a new domain or integrate new functionality, we will ultimately generate questions and hopefully new answers. In this way, SAVS implementations are a generator for new research. Whether it is answering questions about rendering, human-computer interaction, software design, optimization, or in the case of this document, physical simulation, a SAVS acts as fertile soil within which researchers are able to experiment in many areas. But more than simply providing a platform to test isolated ideas, the integrated nature of SAVS style systems encourage a global perspective. Every change affects and is affected by everything else, forcing researchers to take in and understand the big picture around their work and where it fits into the whole.

Utility Gap

The second reason that building a SAVS is often worthwhile is to fill a need. A SAVS is, at its core, software with a purpose. In some cases, such as game development, many

implementations exist, making the bar to justify development high. But in other areas, such as medical simulation, the gaps in functionality coverage are more severe. To use the benchmark described in this document as an example, there have been many projects developing systems for simulated organ surgery, but few systems for performing simulated plastic surgery. Filling this gap is extremely valuable, as without it practitioners are being left behind while their colleagues are more and more enjoying the benefits of modern computing technology.

Intrinsic Value

The development of SAVS itself is valuable, even if we ignore the added value of a SAVS in fulfilling its specified purpose, or the additional research that can be spawned as a result of its construction. Implementing a SAVS requires time, dedication, and skill - but no one enters and leaves such a project unchanged. Simply being a developer on a SAVS helps a one become a better software engineer, through the long hours of practice they will spend on it. Beyond individual developers, building a SAVS is important to the community at large for the reason that its existence demonstrates that such a project can be completed. Like all large pieces of software, sometimes the most important idea they can convey is that such a project is even feasible at all.

3 ENGINEERING DECONSTRUCTION

Given the design challenge of creating a simulation assisted visual system for authoring plastic surgery procedures, the next problem is determining by what methods can we best accomplish this task. Commercially, there exist examples of surgical simulation products based on both procedural animations and physical simulations. As discussed in the previous chapter, a detailed procedural animation can accommodate some, but not all, of the goals for a surgical authoring system. Procedural animation methods can provide realistic geometry, respond to user inputs by advancing down pre-scripted event chains, and serve as a functional replacement for traditional illustrated procedure guides. However, procedural approaches fall short when assisting in cognitive training tasks.

Under cognitive training, the surgeon is expected to be developing a mental intuition for how operations work at a fundamental level and be capable of adapting this knowledge to new situations. These requirements go beyond a simple recall of a "correct" approach, but to understand why these techniques work as they do. Traditionally, this level of understanding is gained through years of experience working with live patients or animal models. Through this process, surgeons experiment with designs (guided by experienced mentors) and observe the results in order to better understand what works and what does not. Supporting this style of intellectual freedom, the ability to deviate from diagrams in textbooks, is the goal of the plastic surgery simulation system we wish to build. To address these concerns, we turn away from procedural animations to physical simulation. Physical simulation can address this problem by providing correct, real-time responses to a user's actions, instead of forcing them into pre-scripted routines.

The remaining chapters in this document will discuss technical contributions that form the foundation of the plastic surgery benchmark application described in the previous chapter. The goal of this chapter is to deconstruct the basic engineering challenges found in this space. This deconstruction will frame the work presented in subsequent chapters, which can

otherwise feel somewhat disconnected from the topic of plastic surgery.

3.1 Background

To start, let us bring the topic of discussion down from the general specifications discussed in the previous chapter and provide a concrete goal for what we want to provide. At the core, our proposed surgical application can be described as a system which instills a reactive mechanical response into three dimensional models of tissue to static and animated constraints. Let us break this goal down further and look at each of these parts in more detail.

To begin, let us start with the concept of mechanical response. Mechanical response of soft bodies, such as human tissue, can be described as a series of relationships between shape, forces, and energy. At a very high level, when forces act on an object and alter its shape, this requires a proportional amount of energy, which is then stored by the object. In real materials, this potential energy can be dissipated in many ways, such as heat or *by further changes in shape*. Our goal in the simulation of elastic materials is to minimize the energy of our virtual materials by this latter outcome. In other words, we attempt to compute new shapes, which minimize the energy of the object, in response to external conditions and forces. The precise form of these shapes depends on the material of the object. For our purposes, we are interested in a family of materials known as *hyperelastic* materials. The energy of these materials depends only on the object's current shape and do not consider the history of the object's shape, only its initial and final ones. While these materials are idealized, they are most similar to rubbers or other materials that "bounce back" after being deformed.

If mechanical response is the relationship between shape, energy, and forces, shape itself is the physical extents of the object. If we consider the virtual tissue models we are interested in simulating, they can be described geometrically as a closed boundary in a three dimensional coordinate system and the region the boundary encloses, which we represent by Ω. For the moment we will consider this domain to be a continuous volume. In a later section, we will

discuss how this domain can be stored discretely on a computer.

The last part, relating to constraints, we will hold off on for now. We will return to this idea at the end of the chapter, once we have covered the mathematical and computational details surrounding mechanical response and shape, which will be the topic of the next two sections.

3.2 Continuous Formulation of Deformation

In this section, we will expand on the relationship between shape, energy, and force from the previous section and provide a mathematical foundation which we will later use for discrete computations. The first idea we need to explore is that of shape change itself. When an object changes shape from one configuration to another, we refer to this as a *deformation*. To keep track of the deformation, we record it relative to the object's reference configuration. This is an arbitrarily chosen shape in the coordinate system of the object from which all other deformations are measured. Mathematically, we can define the deformation as a mapping between the locations in the domain of the object and $\mathbf{R}^3$:

$$\phi : \Omega \subset \mathbf{R}^3 \to \mathbf{R}^3 \tag{3.1}$$

Under this regime, we consider the domain of ϕ to be points in Ω, often referred to as material space locations as they can conceptually be considered infinitesimal blobs of undeformed material. We simultaneously locate and identify them by a spatial vector: $\vec{X} = (X, Y, Z)$. The points in R^3 which make up the image of ϕ are the corresponding deformed locations $\vec{x} = (x, y, z)$. This relationship can be seen in Figure 3.1.

Now that we have defined for what it means for an object to be deformed and the concept of a deformation map, we can return to the relationship between shape, force, and energy. In order to relate shape, or deformation, of an object to its energy, we need to define a way to measure how the deformation varies throughout the object. We facilitate this by defining the

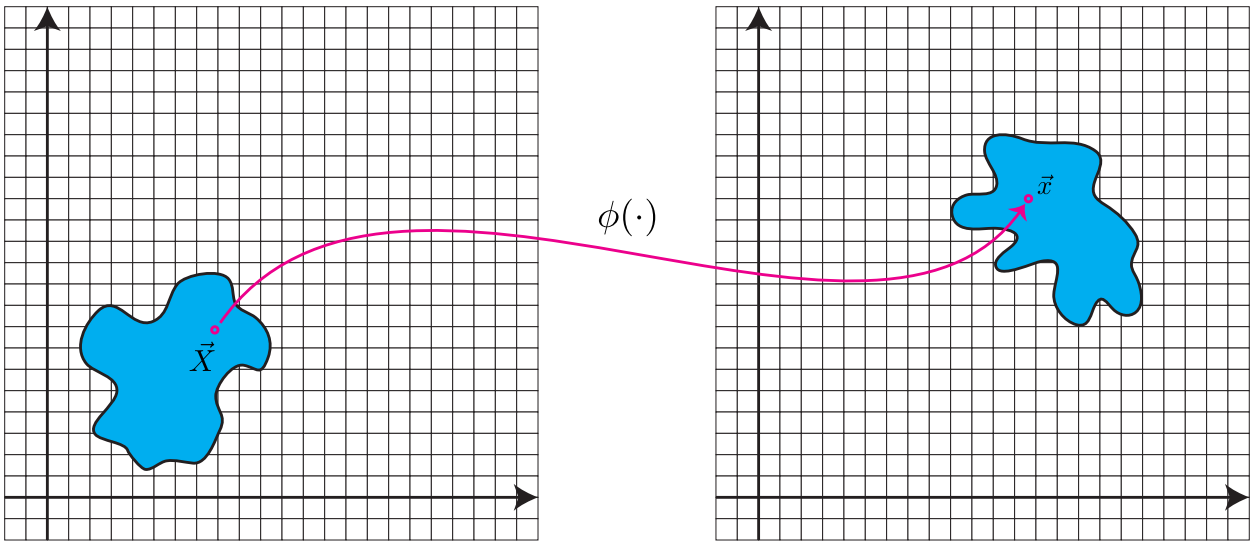


Figure 3.1: Example Deformation Map
The deformation map ϕ is illustrated here mapping between locations $\vec{X}$ in the object's reference configuration and corresponding deformed locations $\vec{x}$ in the object's deformed configuration.

quantity called the deformation gradient $\mathbf{F}$, the jacobian of the deformation map.

$$\mathbf{F}(\vec{X}) = \frac{\partial \phi(\vec{X})}{\vec{X}} \tag{3.2}$$

From the deformation gradient, we can extract metrics of deformation, known as strain measures. The strain measure is directly used to compute the strain energy, the potential energy of the material due to deformation. In the reference configuration, we want the strain measure to be evaluate to zero. Additionally, if the object's deformation is purely translational, we would also like the strain measure to be zero. A simple strain measure that meets these goals might be:

$$s = \mathbf{F} - \mathbf{I} \tag{3.3}$$

While this evaluates to zero under translation and no deformation, this simple measure is non-zero under rotation, which can result in inaccuracies under large deformation. We can treat this issue by employing a different strain measure, the Green strain $\mathbf{E}$:

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^\mathsf{T}\mathbf{F} - \mathbf{I}) \tag{3.4}$$

The Green strain is more robust to large deformations by virtue of it being rotation invariant, but is non-linear and can behave non-intuitively around extreme deformations. As an example, consider a deformation that simply inverts an object along one axis. The Green strain of this deformation would be zero in this case despite a significant deformation. If we take a linear approximation around the undeformed configuration of the Green strain, we arrive at the *small strain* $\boldsymbol{\epsilon}$:

$$\boldsymbol{\epsilon} = \frac{1}{2}(\mathbf{F} + \mathbf{F}^\mathsf{T}) - \mathbf{I} \tag{3.5}$$

This formula is linear and under small deformations is approximate to the Green strain, making it useful for scenarios with limited deformation, such as structural simulations for buildings or other near rigid objects. As the deformation increases however, the small strain suffers from the similar problems as the simple strain presented earlier (Equation 3.3). In order to get a strain measure that is rotationally invariant and is more predictable than the Green strain, one possibility is to apply a polar decomposition to the deformation gradient in order to extract the rotational components:

$$\begin{aligned} \mathbf{F} &= \mathbf{RS} \\ \boldsymbol{\epsilon}_\mathbf{c} &= \mathbf{S} - \mathbf{I} \end{aligned} \tag{3.6}$$

While this strain measure is still non-linear, it behaves similarly to the small strain measure, including a linear (not quadratic) relationship to axis aligned stretch, while maintaining rotational invariance.

Once we have chosen our strain measure, we can use it to define a strain energy density $\Psi(\mathbf{F})$. This is a measure of the potential energy per unit volume of the material as a consequence of undergoing deformation. Similar to the strain measure, there are several

popular energy density functions which can describe generic deformable materials. Linear elasticity (Equation 3.7), derived from the small strain tensor, is easy and uncomplicated to compute due to its linear relationship to the deformation gradient. However, since it is based on the small strain tensor, it is not rotationally invariant and can exhibit physically inaccurate behaviors under large deformations.

$$\Psi(\mathbf{F}) = \mu\boldsymbol{\epsilon} : \boldsymbol{\epsilon} + \frac{\lambda}{2}\mathrm{tr}^2(\boldsymbol{\epsilon}) \tag{3.7}$$

If we replace the strain measure used in the Linear elasticity model with the Green strain, we get what is known as the St. Venant-Kirchhoff energy density function (Equation 3.8).

$$\Psi(\mathbf{F}) = \mu\mathbf{E} : \mathbf{E} + \frac{\lambda}{2}\mathrm{tr}^2(\mathbf{E}) \tag{3.8}$$

While this model is more accurate under large deformations due to the Green strain, it has the unfortunate property of not penalizing significant compression and inversion. In the film production industry, another nonlinear energy function is often used instead, called Co-rotated elasticity, which is based off the fourth strain measure detailed in Equation 3.6:

$$\begin{aligned}\Psi(\mathbf{F}) &= \mu\boldsymbol{\epsilon}_\mathbf{c} : \boldsymbol{\epsilon}_\mathbf{c} + \frac{\lambda}{2}\mathrm{tr}^2(\epsilon_\mathbf{c}) \\ &= \mu\|\mathbf{S} - \mathbf{I}\|_\mathbf{F}^2 + \frac{\lambda}{2}\mathrm{tr}^2(S - \mathrm{I})\end{aligned} \tag{3.9}$$

Finally, we can calculate the energy of the entire object by integrating the energy density over the domain:

$$\mathsf{E}(\phi) = \int_\Omega \Psi(\mathbf{F})\, \mathrm{d}\vec{X} \tag{3.10}$$

In the next section, we will convert these continuous equations of deformation and energy into discrete forms, allowing us to solve for our object's deformations. Up until now, the

equations we have been using have assumed a continuous deformation field, free of any particular discretization. The expressions in the next section will replace these continuous equations with those appropriate for the lattice based discretization we use in our application.

3.3 Discrete form of Elastic Deformation

With these equations in hand, we can now reason about how to produce realistic mechanical responses from our virtual tissue models on a computer. To do so, we need to change how we have been treating the expressions from the previous section from a continuous representation to a discrete representation suitable for computation. To do this, we will be dividing our continuous domain in to smaller regions, or elements. Each element represents a finite quanta of material, thus giving rise to the name of the approach we will be using throughout this document: Finite Element Method (FEM).

In our new discrete world, we must first translate our former continuous domain Ω into a data structure representable on a computer. Here we have a number of choices to pick from. FEM literature has described many approaches for discretization over the years, ranging from volumetric meshes to particle based approaches. These designs can be evaluated over several categories: regularity, conformity, and ease of use. Regularity refers to the extent that the technique uses repeating data structures or provides implicit internal relationships that can be predicted, which is often extremely beneficial for performance optimization. Conformity refers to how the data structure represents the object's boundary shape, either attempting to directly replicate the object's domain or acting as a scaffolding around it, which can affect the accuracy of the simulation. Finally, ease of use is a catch all term that includes any property which makes the data structure easy or troublesome to include in higher level pipelines. In this document, the approach we will be using is a mesh discretization. In particular, we will be using a design referred to as an embedding lattice. Seen in Figure 3.2, we have an example mesh of a lion and its embedding lattice. This data structure is extremely regular

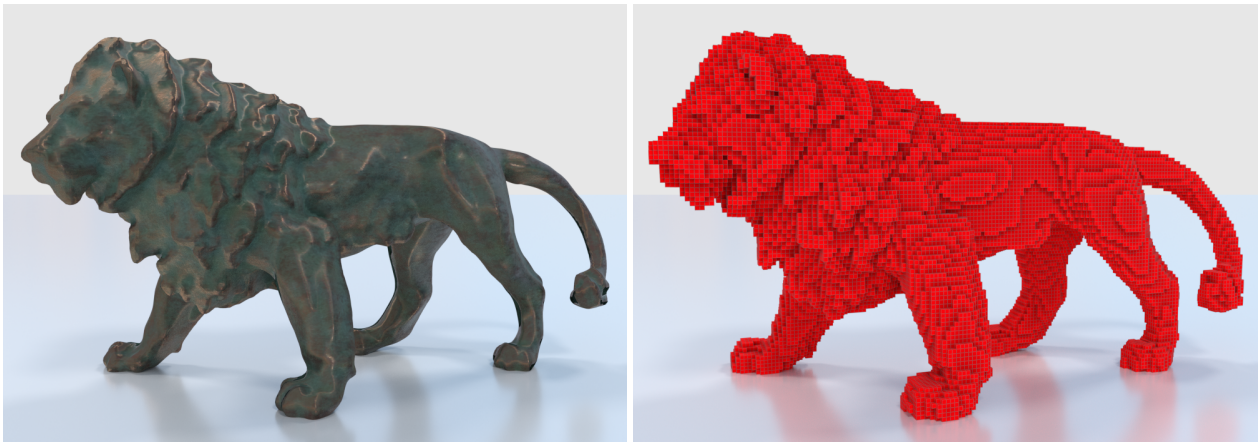


Figure 3.2: Lattice Embedding
Show here is a three dimensional lion model[1](left) and the corresponding embedding lattice (right). Each vertex of the lion model is embedded into a cell of the lattice, which acts as a deformable scaffold around the model.

and generally easy to construct, though it can suffer from a lack of conformity.

Under this representation, our material will be sampled at Cartesian *nodes* of a regular hexahedral lattice and our elements of computation will be its *cells*. We refer to these nodes as containing degrees of freedom, as they represent the discrete points where deformation can occur. To deal with the issue of conformity, we utilize a technique known as lattice embedding - whereby the discrete boundary of our object, represented by a separate surface mesh, is embedded in cells of the lattice via a weighted interpolation of its nodes. Later in this document, additional techniques will be presented to further increase the accuracy of the lattice's boundary conformity. Since the terms presented in regards to embedding can be confusing, we will stick to the following terminology:

lattice - The simulation lattice is the discrete representation of our simulation domain and is composed of elements of material known as cells. Through the simulation lattice, we compute deformation gradients, energy, and ultimately forces.

node - A node is a topological point in our simulation lattice. Each node is identified by a spatial location as a label and stores three degrees of freedom, or DOFs. These degrees

[1]The lion model was rendered with a shader created by Anthony Pilon.

of freedom are represented by the discrete deformation values. Additionally, nodes store the discrete forces resulting from this deformation.

cell - A cell is the basic computational unit of our simulation lattice. It is represented as a regular hexahedral volume and connects the eight nodes at its corners.

mesh - Our tissue material surface is represented by a mesh. This mesh is conforming, i.e it attempts to match the shape of the continuous domain as closely as possible. The deformed material surface is the desired visual result of our physical simulation.

vertex - A vertex is a topological point in our material mesh. Its position in space is wholly determined by its embedding relationship, typically bi-linear, to its parent cell in the deformation lattice.

Using this simulation lattice as a discrete approximation of our continuous domain, we can now translate the concepts from the previous section. Instead of material points, we have nodes. Each node is labeled by an unique identifier. This identifier can be as simple as an integer, e.g. Node 52, but since we are using a regular lattice, we will continue with the previous spatial labeling, although in this case we will now use integer coordinates. Our former concept of a reference configuration will remain, by assigning each node a reference position in space. Defining our deformation map ϕ then becomes as simple as:

$$\phi(\vec{X}; \mathbf{x}) = \sum_{i} x_i \mathcal{N}_i(\vec{X}) \tag{3.11}$$

We define $\mathbf{x} = x_1, x_2, x_3, \ldots, x_n$ to be our discrete deformation state, which includes the displacement of every node in our lattice from its reference location. In this expression $\mathcal{N}_i$ is the *shape function* for each node in the cell. This shape function is used to interpolate the discrete nodal displacements into a continuous deformation deformation field. For the hexahedra we are using, our shape functions are derived from trilinear interpolation. Using

this new description of ϕ, we can derive a continuous definition of the deformation gradient for any point $\vec{X}$ inside the cell just as before:

$$\mathbf{F}(\vec{X};\mathbf{x}) = \frac{\partial \phi(\vec{X};\mathbf{x})}{\partial \vec{X}} \tag{3.12}$$

From this expression, we can integrate over the volume of a cell to arrive at an expression for a cell's energy.:

$$\mathsf{E}^e(\mathbf{F}) = \int_{\Omega_e} \Psi^e(\mathbf{F}(\vec{X};\mathbf{x}))\, d\vec{X} \tag{3.13}$$

In order to evaluate the integral over the volume of the cell, we employ a numerical quadrature approach. This discrete approximation of the cell's energy is expressed in Equation 3.14:

$$\begin{aligned} \mathsf{E}^e(\mathbf{x}) &= \int_{\Omega_e} \Psi^e(\mathbf{F}(\vec{X};\mathbf{x}))\, d\vec{X} \\ &\approx \|\Omega_e\| \sum^{\mathcal{Q}} w_q \Psi^e(\mathbf{F}(\vec{X}_q;\mathbf{x})) = \hat{\mathsf{E}}^e(\mathbf{x}) \end{aligned} \tag{3.14}$$

Here, we are evaluating $\mathcal{Q} = \{\vec{X}_1, \vec{X}_2, ..., \vec{X}_q\}$ quadrature points, each with a individual weight w_q, such that $\sum_q w_q = 1$. $\|\Omega_e\|$ corresponds to the volume of an individual cell. Our choice of quadrature points greatly affects the accuracy of the approximation. Using an eight point Gauss quadrature, while expensive, is third-order accurate for regular, axis-aligned hexahedra. For cells which are only partially covered with material, Patterson et al. [2012] demonstrated an alternative quadrature scheme that second order accurate and uses four points. Finally, one point quadrature is possible [McAdams et al., 2011] if an additional stabilization energy term is included to compensate for unpenalized deformation modes.

At this point, it is straightforward to define the global discrete energy to be the sum of every cell's energy:

$$E(\mathbf{x}) = \sum_e \hat{E}^e \tag{3.15}$$

From these expressions, we can derive an expression for forces at each node by simply taking the derivative of the systems energy at each cell and summing its contribution onto the node:

$$\vec{f}_i(\mathbf{x}) = \sum_e \vec{f}_i^{(e)}(\mathbf{x}), \text{ where } \vec{f}_i^{(e)} = -\frac{\partial \hat{E}^e(\mathbf{x})}{\partial x_i} \tag{3.16}$$

We define $\mathbf{f} = \vec{f}_1, \vec{f}_2, \vec{f}_3, \ldots, \vec{f}_n$ as our force state, which is similar to our deformation state $\mathbf{x}$. Using this relationship between nodal displacements and nodal forces, we can finally describe the process for computing deformations.

Solving for Deformations

When a real object is exposed to external forces and constraints, it will naturally attempt to assume a shape that equilibrates its internal energy with its external constraints. We will seek the same situation in our case by solving for a deformation which corresponds to a local minimum in the object's energy E. This is equivalent to solving for a deformation state $\mathbf{x}$ such that $\mathbf{f}(\mathbf{x}) = 0$, as minimums in E correspond to zero values in its first derivative, i.e. the discrete force state $\mathbf{f}$.

Since the relationship between forces and deformation is generally non-linear, unless we use the linear elasticity model for small deformation situations, we employ the standard Newton-Rhapson method to determine the solution. This method uses a series of linear approximations to determine the roots of our function $\mathbf{f}(\mathbf{x})$. Given some initial configuration $\mathbf{x}_0$, which will typically be the last equilibrium shape, our goal is to find a $\delta\mathbf{x}$ such that $\mathbf{f}(\mathbf{x}_0 + \delta\mathbf{x}) = 0$. To solve for $\delta\mathbf{x}$, consider the Taylor expansion of this expression:

$$\begin{aligned} \mathbf{f}(\mathbf{x}_0 + \delta\mathbf{x}) &= \mathbf{f}(\mathbf{x}_0) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\delta\mathbf{x} + O(\delta\mathbf{x}^2) \\ \mathbf{f}(\mathbf{x}_0 + \delta\mathbf{x}) &\approx \mathbf{f}(\mathbf{x}_0) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\delta\mathbf{x} \end{aligned} \tag{3.17}$$

If we assume that $\mathbf{f}(\mathbf{x}_0 + \delta\mathbf{x}) = 0$, since this is our goal, we can solve for an approximate solution to $\delta\mathbf{x}$:

$$\begin{aligned} 0 &\approx \mathbf{f}(\mathbf{x}_0) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\delta\mathbf{x} \\ -\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\delta\mathbf{x} &\approx \mathbf{f}(\mathbf{x}_0) \\ \delta\mathbf{x} &\approx (-\frac{\partial \mathbf{f}}{\partial \mathbf{x}}^{-1})\mathbf{f}(\mathbf{x}_0) \end{aligned} \tag{3.18}$$

Which suggests an update formula for $\mathbf{x}$:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \underbrace{\left(-\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\bigg|_{\mathbf{x}_n}\right)}_{\mathbf{K}(\mathbf{x}_n)}^{-1}\mathbf{f}(\mathbf{x}_n) \tag{3.19}$$

We refer to the matrix $\mathbf{K}$ as the *stiffness matrix*. In order to arrive at a solved equilibrium configuration, for each iteration n our task becomes the following steps:

1. Update the stiffness matrix $\mathbf{K}$ for the current configuration $\mathbf{x}_n$.
2. Compute forces $\mathbf{f}_n$ corresponding to $\mathbf{x}_n$ for our right hand side of the update expression.
3. Terminate if we are within tolerance to zero net forces.
4. Solve $\mathbf{K}^{-1}$ for a displacement update $\delta\mathbf{x}$.
5. Update our current deformation state $\mathbf{x}_{n+1} = \mathbf{x}_n + \delta\mathbf{x}$
6. Repeat

At the end of this process, our final deformation shape will be the one that equilibrates the internal and external forces of the object. We can use this process to create animations by solving for a series of *quasi-static* poses. These poses are controlled by time varying constraints which are successively applied to the object between solves. If moved in smooth trajectories, these constraints can create the appearance of smooth deformation, despite the

simulation converging to a static pose at every iteration. This can be thought of as taking snapshots of an object's motion, similar to how film is captured, though it does not include dynamic effects such as jiggling or damped motion. However, this approach is much more robust and can handle rather sudden and dramatic departures from prior constraints more readily than techniques that allow more dynamic motions.

Unfortunately, the solution to $\mathbf{K}^{-1}$ is not without significant challenges. Some of the major difficulties include:

Memory Footprint Each entry in $\mathbf{K}$ is a relation between force along one axis and displacement along another between every connected degrees of freedom. Quantitatively, this means we are faced with a matrix which has 27 non-zero entries on each row, where each entry is a dense three by three matrix. To simply store $\mathbf{K}$ for a modest sized domain of 128^3 degrees of freedom, it would require 500 million scalar values at just under two gigabytes of storage if 32-bit floating point representation is used[2]. Since we are required to rebuild and then solve this matrix repeatedly, this footprint represents a significant bottleneck on most platforms where available compute bandwidth exceeds memory bandwidth by an order of magnitude.

Solver Choice Since our goal is to solve a large linear system, we have several options including both direct and iterative style solvers. However, both general techniques have different and significant drawbacks. Direct solvers are generally robust even when the matrix is poorly conditioned due to stiff constraints or non-uniform materials. Unfortunately, direct solvers typically work via a factorization and backward substitution solution, which both requires storing the entire matrix and then performing relatively little computation over its entries compared to the memory reads, making their performance poor on very large systems. Iterative solutions in contrast can do away with storing the entire matrix, instead only requiring that its multiplicative effect

[2]This is not including any additional storage cost overheads for representing the sparse matrix itself. This number is simply the space required to store only the non-zero coefficients alone.

can be provided. This approach is known as a *matrix-free* solution. Iterative methods are potentially faster and allow early termination for approximate solutions, but they are sensitive to matrix conditioning. Iterative solvers can also be slow to propagate effects throughout the domain, focusing on local errors before global smoothness.

Performance Concerns When computing the solution to $\mathbf{K}^{-1}$, attention needs to be paid to how parallelization through multithreading and vectorization can be used to increase performance. What makes this task particularly tricky is that by being more efficient with computation only makes the divide between available memory and computation bandwidth even worse. Careful management of access patterns and data organization needs to be done in order to avoid starving the processor and undoing any advantage parallelization can bring. An additional concern is that if explicit construction of $\mathbf{K}$ is required, considerable expense must be paid for building a data structure used for a single Newton step. We can avoid this issue by sticking with solutions which don't require explicit construction.

Discrete Formulas

In order to invert $\mathbf{K}$ without forming an explicit copy of the matrix, we can use algorithms which are referred to as matrix-free, such as Conjugate Gradients (CG). These techniques operate by taking products of the form $\mathbf{K}(\mathbf{x}_n)\delta\mathbf{x}$, where $\delta\mathbf{x}$ is an arbitrary vector used by the specific algorithm being employed (for convenience, we label this vector as $\delta\mathbf{x}$, though it does not necessarily correspond to a deformation or displacement).

$$\mathbf{K}(\mathbf{x}_n)\delta\mathbf{x} = -\left(\frac{\delta\mathbf{f}}{\delta\mathbf{x}}\bigg|_{\mathbf{x}_n}\right)\delta\mathbf{x} \tag{3.20}$$

The product of $(\frac{\delta\mathbf{f}}{\delta\mathbf{x}}|_{\mathbf{x}_n})\delta\mathbf{x}$ is equivalent to the *differential* $\delta\mathbf{f}[\delta\mathbf{x};\mathbf{x}_n] = (\frac{\delta\mathbf{f}}{\delta\mathbf{x}}|_{\mathbf{x}_n})\delta\mathbf{x}$. In this section, we'll cover the process of computing discrete values for this differential, along with the corresponding forces (the right hand side of Equations [3.18,3.19]).

Algorithm 3.1: Algorithm for computing elemental elastic force and force differentials

1: **procedure** FELASTIC($\mathbf{x}^e$, $\mathcal{Q}$, params $\mathfrak{p}$)
2: **reshape** $\mathbf{x}^e \rightarrow \mathbf{D}_s$
3: **for** $\mathfrak{q} \in \mathcal{Q}$ **do**
4: **compute** $\mathbf{G} \leftarrow$ GRADIENT($X_\mathfrak{q}$)
5: **compute** $\mathbf{F} \leftarrow \mathbf{D}_s\mathbf{G}^T$
6: **compute** $\mathbf{P} \leftarrow \mathrm{P}(\mathbf{F}, \mathfrak{p})$
7: **accumulate** $\mathbf{H} \leftarrow \frac{V_0}{w_\mathfrak{q}}\mathbf{PG}$
8: **reshape** & **accumulate** $\mathbf{H} \rightarrow \mathbf{f}^e$

1: **procedure** δFELASTIC($\delta\mathbf{x}^e$,$\mathbf{x}^e$,$\mathcal{Q}$, params $\mathfrak{p}$)
2: **reshape** $\mathbf{x}^e \rightarrow \mathbf{D}_s$, $\delta\mathbf{x}^e \rightarrow \delta\mathbf{D}_s$
3: **for** $\mathfrak{q} \in \mathcal{Q}$ **do**
4: **compute** $\mathbf{G} \leftarrow$ GRADIENT($X_\mathfrak{q}$)
5: **compute** $\mathbf{F} \leftarrow \mathbf{D}_s\mathbf{G}^T$, $\delta\mathbf{F} \leftarrow \delta\mathbf{D}_s\mathbf{G}^T$
6: **compute** $\mathcal{T} \leftarrow$ DPDF($\mathbf{F}, \mathfrak{p}$)
7: **compute** $\delta\mathbf{P} \leftarrow \mathcal{T} : \delta\mathbf{F}$
8: **accumulate** $\delta\mathbf{H} \leftarrow \frac{V_0}{w_\mathfrak{q}}\delta\mathbf{PG}$
9: **reshape** & **accumulate** $\delta\mathbf{H} \rightarrow \delta\mathbf{f}^e$

These two algorithms for elemental forces and force differentials, respectively, receive as input values from each of the eight nodes of a cell e, indicated by x^e, and accumulate forces back onto these nodes. The functions P and DPDF can be user-defined to implement arbitrary isotropic materials, controlled by material parameters $\mathfrak{p}$. The **reshape** operation concatenates eight nodal vectors in a cell (positions, forces, etc) into a 3×8 matrix and vice versa. $\mathcal{Q}$ is the set of quadrature points used to approximate the volume integral. $\mathbf{G}$ is a 3×8 gradient matrix encoding the derivative of the shape functions. $V_0 = h^3$ is the volume of the cartesian cell. $\mathcal{T}$ is a sparse 4th order tensor as defined in Teran et al. [2005b]

Trilinear elements and gradients We start by deriving a concise expression for the deformation gradient $\mathbf{F}(\vec{\mathbf{X}}; \mathbf{x})$. We use the notation $\{\vec{\mathbf{X}}_{\mathtt{i}_1\mathtt{i}_2\mathtt{i}_3}\}$ s.t. $\{\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3\} \in \{0, 1\}$ for the eight vertices of a given cell with $\vec{\mathbf{X}}_{\mathtt{i}_1\mathtt{i}_2\mathtt{i}_3} = \vec{\mathbf{X}}_0 + (\mathtt{i}_1, \mathtt{i}_2, \mathtt{i}_3)\mathtt{h}$, where $\mathtt{h}$ is the diameter of a cell. Their respective interpolation basis functions are $\mathcal{N}_{\mathtt{i}_1\mathtt{i}_2\mathtt{i}_3}(\vec{\mathbf{X}}) = \prod_{\mathtt{k}}(\xi_{\mathtt{k}})^{\mathtt{i}_{\mathtt{k}}}(1-\xi_{\mathtt{k}})^{1-\mathtt{i}_{\mathtt{k}}}$ where

$$\begin{aligned}\vec{\xi}(\vec{\mathbf{X}}) &= (\xi_1, \xi_2, \xi_3) \\ &= (\mathtt{X} - \mathtt{X}_0, \mathtt{Y} - \mathtt{Y}_0, \mathtt{Z} - \mathtt{Z}_0)/\mathtt{h}\end{aligned}$$

are the trilinear coordinates of $\vec{\mathbf{X}}$. Partial derivatives of the interpolating functions are readily computed as $\partial_{\mathtt{j}}\mathcal{N}_{\mathtt{i}_1\mathtt{i}_2\mathtt{i}_3}(\vec{\mathbf{X}}) = (-1)^{1-\mathtt{i}_{\mathtt{j}}}\prod_{\mathtt{k}\neq\mathtt{j}}(\xi_{\mathtt{k}})^{\mathtt{i}_{\mathtt{k}}}(1-\xi_{\mathtt{k}})^{1-\mathtt{i}_{\mathtt{k}}}$. From this point, we shall refer to the vertices of a specific cell simply as $\vec{\mathbf{X}}_1 ... \vec{\mathbf{X}}_8$, and $\mathcal{N}_1(\vec{\mathbf{X}}) ... \mathcal{N}_8(\vec{\mathbf{X}})$ for the respective trilinear basis functions, with the understanding that we know how to relate to the prior indexing convention. By equations (3.11,3.12) the deformation gradient at a location $\vec{\mathbf{X}}_*$ is

$$\begin{aligned}\mathbf{F}(\vec{\mathbf{X}}_*; \mathbf{x}) &= \sum_{\mathtt{i}} \vec{\mathbf{x}}_{\mathtt{i}} \nabla\mathcal{N}_{\mathtt{i}}(\vec{\mathbf{X}}_*)^{\mathsf{T}} \\ &= \mathbf{D}_{\mathbf{s}}(\mathbf{x})\mathbf{G}(\vec{\mathbf{X}}_*)^{\mathsf{T}}\end{aligned} \tag{3.21}$$

where $\mathbf{D}_{\mathbf{s}}(\mathbf{x}) = [\vec{\mathbf{x}}_1\ \vec{\mathbf{x}}_2\ \ldots\ \vec{\mathbf{x}}_8] \in \mathbf{R}^{3\times 8}$ is the cell shape matrix. The matrix $\mathbf{G}(\vec{\mathbf{X}}_*) \in \mathbf{R}^{3\times 8}$ with $\mathbf{G}_{\mathtt{i}\mathtt{j}}(\vec{\mathbf{X}}_*) = \partial_{\mathtt{i}}\mathcal{N}_{\mathtt{j}}(\vec{\mathbf{X}}_*)$ will be referred to as the gradient matrix at $\vec{\mathbf{X}}_*$. Note that for any material point $\vec{\mathbf{X}}_*$, $\mathbf{G}(\vec{\mathbf{X}}_*)$ can be precomputed as its value is independent of any deformation.

Force computation We mimic the elastic force derivation from McAdams et al. [2011], referring to Equations [3.14,3.16], for an individual force components $\mathtt{f}_{\mathtt{i}}^{(\mathtt{j})}$, where $\mathtt{i}$ indicates which node and $\mathtt{j} \in \{1, 2, 3\}$ indicates the axis:

$$
\begin{aligned}
f_i^{(j)} &= -\frac{\partial E^e}{\partial x_i^{(j)}} = -h^3 \frac{\partial \Psi(\mathbf{F}^e)}{\partial x_i^{(j)}} \\
&= -h^3 \sum_{k,l} \left. \frac{\partial \Psi(\mathbf{F}^e)}{\partial F_{kl}} \right|_{\mathbf{F}^e} \frac{\partial F^e_{kl}}{\partial x_i^{(j)}} \\
&= -h^3 \sum_{k,l} [\mathbf{P}(\mathbf{F}^e)]_{kl} G^e_{li} \delta_{ik} \\
&= -h^3 \sum_{l} [\mathbf{P}(\mathbf{F}^e)]_{jl} G^e_{li}
\end{aligned}
\tag{3.22}
$$

Continuing from this expression, we can pack, or *reshape*, the resulting forces into a unified matrix: $\mathbf{H}_e(\mathbf{x}) = [\vec{f}_1 \ \vec{f}_2 \ \ldots \ \vec{f}_8]$, which is a 3×8 matrix containing all nodal elastic forces in a cell Ω_e. The force matrix $\mathbf{H}_e$, for a set of quadrature points $\mathcal{Q}$, is assembled as:

$$
\mathbf{H}_e(\mathbf{x}) = -h^3 \sum_{\mathcal{Q}} w_q \mathbf{P}(\mathbf{F}(\vec{\mathbf{X}}_q; \mathbf{x})) \mathbf{G}(\vec{\mathbf{X}}_q) \tag{3.23}
$$

$$
\text{where} \quad \mathbf{P}(\mathbf{F}) = \partial \Psi(\mathbf{F}) / \partial \mathbf{F}.
$$

In this last expression, $\mathbf{P}$ is the 1st Piola-Kirchhoff stress tensor.

Force differentials The Conjugate Gradients (CG) solver used to solve equation (3.18) does not require the matrix $\mathbf{K}(\mathbf{x}_n)$ to be explicitly constructed, as long as its action on an input vector $\delta\mathbf{x}$ can be evaluated. The result of this implicit matrix-vector multiplication are the force differentials $\delta\mathbf{f}$. We perform this matrix-free operation on an element-by-element basis, adding the contribution of each Ω_e to the aggregate differentials. The force differentials can be collectively computed as $\delta\mathbf{H}_e(\delta\mathbf{x}; \mathbf{x}) = [\delta\vec{f}_1 \ \delta\vec{f}_2 \ \ldots \ \delta\vec{f}_8]$, described in McAdams et al. [2011] and Patterson et al. [2012], as follows:

$$
\delta\mathbf{H}_e(\delta\mathbf{x}; \mathbf{x}) = -h^3 \sum_{\mathcal{Q}} w_q \delta\mathbf{P}(\delta\mathbf{F}; \mathbf{F}(\vec{\mathbf{X}}_q; \mathbf{x})) \mathbf{G}(\vec{\mathbf{X}}_q) \tag{3.24}
$$

$$
\text{where} \quad \delta\mathbf{P}(\delta\mathbf{F}; \mathbf{F}) = \mathcal{T}(\mathbf{F}) : \delta\mathbf{F},
$$

$$\text{and} \quad \delta\mathbf{F} = \mathbf{D_s}(\delta\mathbf{x})\mathbf{G}(\vec{\mathbf{X}}_\mathbf{q})^\mathsf{T}.$$

In this expression, the fourth order tensor $\mathcal{T} = \partial^2\Psi/\partial\mathbf{F}^2$ is the stress derivative. We refer the reader to Teran et al. [2005b] for a discussion of how this tensor can be constructed via the SVD for isotropic materials.

3.4 Constraints

Until this point, the discussion of materials and mechanical responses has not mentioned the last component of the initial problem statement: constraints. This is a complex topic and it is best to approach it after having a basic understanding of how simulated materials are solved in general. At basic level, a constraint is a condition we are either forcing or strongly encouraging the simulation to meet. For our purposes, we will consider two general categories of constraints, defined by their use cases: animated user interaction constraints and static scene constraints.

User Interactions

User interaction constraints are the mechanism through which users are able to control and direct the simulation. In the real world, we interact with objects intuitively - we push and pull things, glue them in place, or join them via mechanical connectors. For simulated objects, we can perform many of the same conceptual tasks by mapping their high level descriptions into the fundamental forces behind them. Our benchmark application will support two primary types of user interaction constraints: hook constraints and suture constraints.

A hook constraint acts to pull material locations towards user specified points in space. This constraint is treated as an influence, not a hard requirement. During the simulation, a user is allowed to place an arbitrary number of hooks, move their target locations, and remove them. We consider these hooks to be animated, as the user is allowed to dynamically alter them over the course of the simulation. However, as was discussed before, our simulation is

quasi-static - meaning that changes to the hooks can only occur in between solve steps. While this prevents a user from having smooth control, we can create its illusion by decreasing the time of each solve step. In later chapters we'll look at techniques for improving raw performance, but we can also stop the Newton method early - displaying partially converged deformations to the user, but allowing them the opportunity to engage in further control actions to refine the partial results.

Suture constraints are similar to hook constraints, in that they pull material towards other locations, except sutures pull two material locations towards one another instead of arbitrary spatial locations. Users are allowed to place and remove sutures during the simulation, just like hook constraints. However, they are not allowed to move them. Once created, a suture acts as a semi-permanent influence between two material locations, attempting to pull them together.

Both hooks and sutures are allowed to be placed in arbitrary material locations and are not restricted to discrete locations, like nodes. This is accomplished by embedding them, similar to how we embed vertices of the surface mesh. However, in this case we are embedding a constraint point which imparts external forces on its embedding cell, rather than the cell imparting a deformation on the point, as would be the case for a vertex. To generate these forces, we implement hooks and sutures as as zero-rest length springs, which obey the simple one dimensional form of Hooke's law:

$$\vec{f} = -k\|x_a - x_b\|_2 \tag{3.25}$$

In this relationship, the restorative forces (forces acting in a direction to return the spring to its resting length) $\vec{f}$ are proportional to the amount the spring has been stretched, represented by the distance between its two end points x_a and x_b, all of which is modulated by a spring constant k. In our case, the endpoints x_a and x_b might be absolute positions, or they may be relative to a cell, defined by a set of trilinear interpolation weights. The forces, similarly,

are applied by distributing themselves to their surrounding nodes through the inverse of the interpolation operation.

Challenges In terms of the equations we've looked at before, force constraints modify both the stiffness matrix $\mathbf{K}$ and the right-hand side $\mathbf{f}$ in Equation 3.19. This immediately suggests several technical challenges. First, there is a question of the strength of the spring constant k. This value directly controls how much influence the constraint exerts on the material to which it is attached. If it is made too weak, the hooks and sutures will not look realistic, not moving the material enough. If there are too many springs embedded in a cell, there may be insufficient degrees of freedom available to satisfy the constraints, creating locking behaviors. Strong springs can also cause iterative methods like CG to spend large amounts of effort correcting their local effects, leading to slow global convergence. Hooks and sutures additionally disrupt the regularity aspects of the deformation lattice, making the force computations in some cells different from others. Sutures are more challenging, as they additionally introduce non-local effects - forces in one cell depend on the deformation of a potentially far away cell, instead of its immediate neighbors.

Scene Constraints

The second category of constraints are the static scene constraints. In this case the term static is referring to the notion that the user does not have control over the creation of these constraints and that they are instead initialized once for each surgical model. This is a subtle distinction, which will be more clear during the discussion about collision handling a little later. There are two types of scene constraints that are employed in our application: positional constraints and contact constraints.

Arguably the easiest constraint type, positional constraints can be thought of as holding a part of an object fixed in space with infinitely strong glue. Mathematically speaking, these constraints are referred to as *Dirichlet*, or boundary conditions which are defined by enforcing

a specific value at a location. Numerically, Dirichlet conditions are applied by enforcing a specific deformation on constrained nodes, usually being no deformation. For virtual surgery, these constraints could serve to pin the edges of tissue down, or act as a transition between the simulated and non-simulated regions, preventing separation. For our application, we enforce positional constraints only at node granularity as this is relatively easy and satisfies the use cases we have.

Contact handling, or collision handling, is the constraint which exists to prevent one object from non-physically penetrating another object. Collision is a particularly tricky property to support in elastic simulation, especially self collision. Collision can be broken into two distinct steps: collision detection and collision response. *Collision detection* is the aptly named process of determining whether or not collision has occurred at any particular point in time. For volumetric objects, which possess distinct inside and outside regions, collision detection typically takes the form of testing for surface penetration. This task, in a general sense, is rather expensive, so we reduce the computational load by restricting our tests to specific locations, or proxies. Proxies are points distributed (and embedded into the simulation lattice) over the surface of the object, creating a discrete stand-in for the true surface. The next step is determining which proxies are currently in states of collision or interpenetration.

For rigid body collisions, i.e. collisions between the soft body and an external rigid object, level sets have been employed quite successfully for collision detection purposes [Teran et al., 2005b, McAdams et al., 2011] with complexities in the order $\mathsf{O}(1)$ for any point of interest. The general idea is to define a level set over a rigid body and then between each quasi-static solution check which proxies are in a state of penetration. For each proxy that is colliding, the *collision response* we employ is a penalty force. This takes the form of an additional spring constraint we introduce between the proxy and the closest location on the surface of the object, which can be determined via the level set. This spring, over the course of subsequent solves, acts to push (or pull) the material into a non-colliding state. Self collision

scenarios work similarly, but with an additional complication that both surfaces involved are potentially deforming. This topic will be covered in more detail in Chapter 5, where a full collision processing algorithm with level sets will be described.

Challenges The primary difficulties with this approach are due to geometry, robustness, and performance. The first problem that comes up is that our proxies need to be a good representation of our object's surface. Too few and we can miss substantial penetrations. But too many can be needlessly expensive to handle during the collision response phase. Distribution is important as well. For instance, choosing to place a proxy at the barycenter of each embedded mesh triangle is only reasonable if the mesh is uniformly discretized - uneven coverage of large and small triangles can lead to uneven collision detection. Additionally, since our penalty forces are implemented via spring constraints, we suffer from the same robustness issues discussed earlier for user constraints. Only now we have many more such constraints, potentially orders of magnitude more, making the problem more delicate. This approach can also suffer from instabilities relating to the discrete nature of the penalty forces - as proxies return to non-colliding states, other forces acting on the object can easily push them back, creating a pseudo-vibratory behavior as penalty forces are removed and reinstated. Finally, we need a way of handling the response in a performant manner. Before we mentioned that springs can break the regularity of the stiffness matrix making parallelization more difficult, but now we have lots of springs which only intensifies the potential problems. Maintaining reasonable performance in the face of these challenges will be touched on in Chapter 6.

3.5 Topology Change

The last topic that needs to be discussed in relation to simulation and constraints is the concept of topology change. Apart from physically moving tissue around or joining it with sutures, this interaction represents a user's ability to incise material with some type of virtual cutting implement. As such, supporting topology change is very important as without support

most surgical procedures would be impossible to perform. It is worth considering the ways that topology change in simulated objects can be done. Initially, it might be tempting to say that topology change needs to mimic the actual physics of a scalpel cutting tissue. In other words, track the physical forces at the tip of the blade, the strain limit of the tissue, and cause the material of the simulated object to separate as the blade traverses the surface. This approach is appealing due to the naturalness of the effect - as the blade moves, tissue is cleanly cut and slides away from it on either side, just as one might expect in a real life procedure.

Unfortunately, this form of *online* cutting, where the topology change and deformation solution are fully coupled, is very complicated. It requires careful maintenance of multiple data structures and the stiffness matrix, all while ensuring the problem remains robust and avoids spurious forces. It is also not, strictly speaking, necessary. If we were approaching the problem from a psychomotor perspective, where we were interested in training the user on how it would feel to cut tissue, this type of cutting would be required to correctly inform haptic feedback devices. However, as stated in the previous chapter, we are mostly interested in the cognitive aspects of plastic surgery. Under this regime, online cutting is less important than providing users a clear interface in which to plan and enact cuts with precision.

To provide for this need, our system defines cuts via user traced paths. Working in the undeformed, or reference, configuration, users are allowed to trace cutting paths on the model's surface. These paths are then extruded into triangularized cutting surfaces which follow the normal of the surface. We allow users to enact these cuts in discrete points in time, creating a sequence of cut operations. These cutting surfaces are then used to geometrically divide the underlying surface mesh, simulating the separation of tissue. While this approach does not support certain types of operations, such as cleft lip repair or operations on highly volumetric regions like the human breast, due to its inability to cut deformed material, it easily supports the local flap style operations we are targeting.

Challenges Even this restricted description of cutting has an important challenge to overcome, namely how we embed the cut surface mesh into our lattice. Simply embedding the mesh into a lattice would almost certainly place both sides of the cut into the same cell, effectively joining the material as if the cut never happened. We could simply increase the resolution of the lattice, making the cells small enough to resolve the cut - but this doesn't work for zero thickness cuts. If we had replaced our lattice with an explicit simulation discretization, perhaps a conforming tetrahedral mesh, we could consider re-meshing the simulation discretization to conform to the cut. However, a generic lattice doesn't have that kind of flexibility, as its topology is implicit. The best approximation to the tetrahedral case would be to simply remove entire cells, leading to extremely coarse, axis-aligned cuts. In Chapter 5, we'll show how these problems can be overcome with the introduction of a non-manifold topology design (as previewed in Figure 3.3), in Chapter 6 we'll look at dealing with the regularity challenges due to cutting, and in Chapter 7 we will cover an advanced solver approach that can more capably handle cells with partial material coverage.

3.6 Engineering A Solid Foundation

The following chapters will discuss in more detail some of the important engineering challenges and present techniques to overcome them. The primary goal is to demonstrate a set of related approaches that work well with each other and can be used to construct a highly performant,

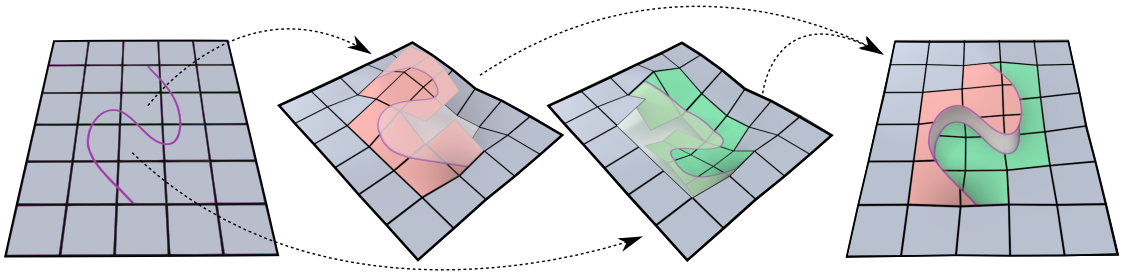


Figure 3.3: Illustration of non-grid aligned topology change
A cutting path (far left) gives rise to new cells on either side of the cut (center), allowing the lattice to be divided below cell resolution (far right).

yet flexible foundation for a plastic surgery Simulation Assisted Visual System (SAVS). After a short related work section, highlighting some informative prior work, there will be several technical chapters dealing with the topics below. Finally, there will be a brief discussion section which will touch briefly on remaining philosophical issues and technical conclusions.

Thin Feature Support Due to the nature of the application, ensuring our simulated tissue can support thin geometric features, arising from incising tissue or simply the original tissue itself, is critical. What do we mean by support? In particular, our simulations need to be able to represent this geometry correctly within an embedded lattice deformer without compromising performance and accommodating behaviors like self collision. The current danger we face is that by embedding geometry in a lattice, instead of making our simulation elements conforming, we will not only loose accuracy around the boundary but potentially create unrealistic behavior by incorrectly connecting material across gaps unresolved by the lattice. In Chapters 5 and 6, we'll look at solutions to these concerns via a process of non-manifold embedding.

Optimized Lattice Deformers We defined a SAVS to be an interactive system, and in order to use elastic simulation as a supporting technology we must ensure it can maintain sufficient levels of interactivity. Thus simulation performance is a key engineering challenge for us. Several times in this chapter we have mentioned that embedding lattice deformers were picked as the discretization model of choice over competing solutions, such as conforming mesh designs, due to their performance opportunities. These opportunities stem from their implicit data structures and regularity. These qualities reduce the amount of information than needs to read in order to access the store simulation data and allow for simplified computational designs. However, these advantages do not come for free. In Chapters 6 and 7, we'll show how these properties can be exploited to create hardware aware algorithms for the solutions to the elasticity equations presented in this chapter.

Support for Non-linearity While linear elastic materials are popular and easy to simulate, they fall short of the complexities observed in real biological material. While this document doesn't make any claim at delivering such materials, which require significant testing and validation against real tissue properties, we do wish to support these materials effectively. Moreover, even with relatively simple material models, adding additional constraints (either from user interactions or from contact) can make the problem non-linear. In Chapter 7, we demonstrate a powerful generic technique for solving these highly non-linear problems, allowing for rapid convergence onto a correct solution.

Deployment While often overlooked, it is equally important to deliver a finished simulation to a user as it is to perform the simulation quickly and correctly. In Chapter 8, we'll explore solutions for presenting simulated surgery operations to a multi-user environment. In particular, we'll look at issues such as available operating environments, suitability of cloud services, and maintainability.

4 RELATED WORK

Faced with the challenge of providing an interactive virtual environment for authoring plastic surgery simulations, the field of computer graphics research has generated many potential solutions and techniques to solve this problem. Procedural techniques [Joshi et al., 2007, Wang and Phillips, 2002, Kavan et al., 2008, Vaillant et al., 2013] offer real-time performance for certain animation tasks but lack the physical accuracy needed in surgical simulations. Consequently, some research ventures into surgical simulation turned to elastic deformation models [Terzopoulos et al., 1987] that responded more realistically to scenarios of probing and cutting [Bro-nielsen and Cotin, 1996, Mendoza and Laugier, 2003, Nienhuys and van der Stappen, 2001]. However, these early works were limited in their scope and effectiveness due to computational cost, geometric constraints and oversimplified material models. In this chapter we outline prior contributions that help address these limitations, and review a number of existing surgical simulation systems.

Simulation of Elastic Materials

Simulation of elastic deformable models is ubiquitous in computer graphics and remains a vibrant area of research. Algorithmic techniques for deformable body simulation, pioneered by Terzopoulos et al. [1987] have attained a significant level of maturity, leading to broad adoption in visual effects, games, virtual environments and biomechanics applications. Over the years, a number of different approaches have been explored for the simulation of deformable objects, each with different strengths.

Lattice-based volumetric deformers are popular components in both physics-based and procedural animation techniques. In the case of physics-based simulation, one of their key advantages is that they avoid having to construct a simulation-ready conforming volume mesh, which is a delicate preprocessing task often requiring supervision and fine-tuning. Another crucial benefit is that the regularity of such data structures enables aggressive performance

optimizations as vividly demonstrated by shape matching techniques [Rivers and James, 2007]. Cartesian lattices have also been leveraged to accelerate performance in physics-based approaches, albeit predominantly for simple models such as linear or corotated elasticity [Müller et al., 2004, Georgii and Westermann, 2008, McAdams et al., 2011]. Prior graphics work, however, has not demonstrated such aggressive performance gains from lattice-based discretizations when highly nonlinear, anisotropic or incompressible materials are involved. In part, this is attributed to the fact that simulation of complex materials commands an increased level of attention to issues of robust convergence. Mature solutions to these concerns have predominantly been demonstrated in the context of specific discretizations (e.g. explicit tetrahedral meshes) where regularity of data structures, compactness of memory footprint and parallelization/vectorization potential were not inherently emphasized. Furthermore, as applications requiring the use of complex materials are also likely to emphasize geometric accuracy, they often opt for conforming mesh discretizations due to their superior performance in capturing intricate boundary features, even if their computational cost is higher.

Despite the popularity and feature set of conforming discretizations, many researchers have explored improving embedded techniques for reasons of simplicity and performance. Embedding has been combined with homogenization [Nesme et al., 2006, Kharevych et al., 2009] to resolve sub-element variation of material parameters, optionally with the use of non-manifold embedding lattices to support objects with a branching structure [Nesme et al., 2009]. Jerabkova et al. [2010] employed a method similar to the one presented in Chapter 5, using a finer voxel grid to capture material topology to be embedded in a coarser, non-manifold voxel grid. Finally, Zhao and Barbič [2013] demonstrated the use of multiple voxel grid domains to segment a model hierarchically, which they used to simulate plants at interactive rates.

In addition to classical FEM approaches, some authors have achieved success with more exotic variations. Extended FEM (XFEM) formulations have also been explored [Jeřábková and Kuhlen, 2009], where discontinuities are introduced into the element's shape functions, to

model cutting. In a similar vein, Kaufmann et al. [2009b] used discontinuous Galerkin FEM formulations. Others have dispensed with mesh based discretizations completely, preferring meshless methods [De and Bathe, 2000] which were also used for surgical simulation [De et al., 2005].

Anatomical Modeling

Approaches based on the FEM have been particularly popular in the medical simulation community [Marchal et al., 2008] where the need for biologically accurate materials is more pronounced. In one of the earliest uses of advanced materials in computer animation, Chen and Zeltzer [1992] focused on anatomical structures such as muscles. FEM techniques were further leveraged in the animation literature for the discretization of linear elasticity for fracture modeling in a small-strain regime [O'Brien and Hodgins, 1999]. Highly nonlinear materials such as active musculature Teran et al. [2003] exposed challenges in robustness and numerical stability of FEM discretizations. Invertible FEM [Irving et al., 2004] improved simulation robustness in scenarios involving extreme compression, while modified Newton methods [Teran et al., 2005b] reduced the cost of implicit schemes with large time steps. Several of these algorithms have been incorporated in open-source modeling and simulation packages [Sin et al., 2013]. Solutions have also been proposed for material behaviors such as incompressibility [Irving et al., 2007] and viscoelasticity [Goktekin et al., 2004, Wojtan and Turk, 2008], both of which can be found in typical biomaterials. Recent results in coupled Lagrangian-Eulerian simulation of solids have also facilitated the inclusion of intricate contact and collision handling in biomechanical modeling tasks [Sueda et al., 2008, Li et al., 2013, Fan et al., 2014].

Topology Change

A number of techniques have targeted topology change during simulation, due to cutting or fracture. Early work [Terzopoulos and Fleischer, 1988] resorted to breaking connectivity of

elements when stress limits were exceeded. Later methods [Nienhuys and van der Stappen, 2001] split tetrahedra near cut boundaries and then used vertex snapping to more accurately approximate the cut. Steinemann et al. [2006] used a similar approach in the context of surgical simulation, where a combination of node-snapping and edge cuts on tetrahedra were used to avoid thin elements, while still remaining close to the user's specified cuts. Local remeshing was also employed to simulate cracks in brittle materials [O'Brien and Hodgins, 1999]. An issue with such subdivision schemes is the possible creation of poorly conditioned elements, which prompted a number of authors to pursue embedded simulation schemes [Molino et al., 2004, Teran et al., 2005a]. These techniques use non-conforming meshes with elements which are only partially covered by material, in lieu of conforming remeshing. Embedded simulation can provide a great degree of flexibility in cutting and fracture scenarios [Sifakis et al., 2007], although cutting meshes along arbitrary surfaces requires delicate book-keeping and careful handling of degeneracies.

Surgical Simulation

While much of the previously discussed work is geared towards general elastic body simulation in computer graphics, many relevant results originated in surgery-specific work. Pieper et al. [1995] demonstrated a very early surgical simulation platform for facial procedures, using FEM elastic shells. Many surgical simulation projects focus on the mechanical manipulation of organs and other soft internal objects [Nienhuys and van der Stappen, 2001, Kim et al., 2007]. Even expensive commercial simulators like the Lap Mentor and GI Mentor primarily focus on pushing and cutting simulated internal organs [Simbionix USA Corporation, 2002–2014b,-]. These types of simulations are so common that several open source frameworks have been built to specifically support further development [Allard et al., 2007, Cavusoglu et al., 2006]. These provide easy access to common components like haptic feedback and APIs to connect multiple simulated components. Certain surgical simulation systems are tailored to specific skills, including interactive simulations of needle insertion [Chentanez et al., 2009].

Performance Optimization

Improving simulation rates is a common challenge for many interactive modeling tasks, and even more so for accuracy-conscious applications such as virtual surgery. Attempts to improve performance have either relied on new data structures, faster solvers, or aggressive use of parallelization. The Boundary Element Method [James and Pai, 1999] has been used to achieve interactive deformation rates for objects manipulated via their surface. Other authors have employed similar formulations that abstract away interior degrees of freedom to accelerate collision processing [Gao et al., 2014]. Grid-based, embedded elastic models [Müller et al., 2004, Nesme et al., 2006, McAdams et al., 2011, Patterson et al., 2012, Mitchell et al., 2015] have been very popular due to their inherent potential for performance optimizations, and can also be used with shape-matching approaches [Rivers and James, 2007]. They form the foundation for a class of highly efficient, multigrid-based numerical solution techniques [Zhu et al., 2010, Georgii and Westermann, 2008, Dick et al., 2011]. Regular discretizations have also been coupled with multigrid solvers to facilitate GPU accelerations for elastic skinning techniques [McAdams et al., 2010]. However, in spite of the efficiency of multigrid schemes, adapting them to the presence of incisions or other intricate topological features can be a nontrivial proposition.

Hermann et al. [2009] analyzed data flow in their simulations to inform a parallel scheduler for multicore systems. To avoid write hazards during parallel code execution, Kim and Pollard [2011] proposed a system of computation phases with coalesced memory writes, which allowed them to parallelize force computation. Related efforts by Courtecuisse and Allard [2009], developed a parallel version of the Gauss-Seidel algorithm that can run on GPUs. Finally, optimized direct solvers have been shown to be very effective [Sin et al., 2013] and have employed techniques such as delayed updates to factorization approaches [Hecht et al., 2012] for improved efficiency. The approach outlined in Chapter 7 is related to these approaches, as well as the general class of Schur complement methods [Quarteroni and Valli, 1999].

Level Sets and Collision Handling

Level set methods were first introduced by Osher and Sethian [1988] for tracking moving interfaces in the context of Hamilton-Jacobi equations. Subsequently, Adalsteinsson and Sethian [1994] proposed substantial runtime savings by restricting computations to a thin band of active voxels near the interface. Sethian [1998] proposed fast marching methods for monotonically advancing fronts as well as for redistancing the level set using values seeded only on the narrow band. Besides fast computation, a number of methods have also been proposed for efficiently storing level sets including octrees [Losasso et al., 2004], RLE representations [Houston et al., 2006, Irving et al., 2006, Chentanez and Müller, 2011], the VDB data structure [Museth, 2013] which evolved from Dynamic Tubular Grids [Nielsen and Museth, 2006] and the DB+Grid data structure [Museth, 2011], and the virtual-memory based SPGrid data structure [Setaluri et al., 2014a].

Methods have been proposed for computing implicit representations of non-manifold surfaces [Bloomenthal and Ferguson, 1995, Yuan et al., 2012]. Similar ideas were used for simulating bubbles [Zheng et al., 2006] and multiphase fluids [Losasso et al., 2006]. The work in Chapter 5 diverges from these approaches as we enhance the expressive capability of a *single* level set by embedding signed distance values on an explicit mesh. Our work is related to the practice of embedding high-resolution geometry in regular meshes, a concept that was first leveraged by Muller et al. [2004] for deformable body simulations and fracture. In addition to hexahedral embeddings, methods such as the virtual node algorithm [Molino et al., 2004] have been used to create non-manifold tetrahedral lattices that correspond to thin topological features in the embedding geometry. Virtual node concepts are also similar to XFEM methods which were used for crack modeling [Moës et al., 1999] and for cutting and fracturing thin shells [Kaufmann et al., 2009a]. This principle has continued to evolve with many of the topological limitations in prior approaches being raised by Sifakis et al. [2007] and has been successfully used in production tools as well [Hellrung et al., 2009].

Our non-manifold level set approach in Chapter 5 is inspired by these methods, but

it needs to be made cognizant of further topological limitations that the signed distance field imposes on our representation (see Section 5.3). Notably, when dealing with collisions near thin features, all of the aforementioned approaches employed detection and response techniques based on surface meshes [Bridson et al., 2002] that rely on the availability of good surface meshes, are computationally expensive, presume collision-free history or use impulses which makes implicit integration challenging. To accelerate collision detection and response while allowing for implicit integration, methods have been proposed using implicit surface representations [McAdams et al., 2011] which work even in near-interactive settings, but require enough level set resolution to avoid any non-manifold features altogether. Recently, image-based techniques [Faure et al., 2008, Wang et al., 2012] have been proposed which provide an interesting alternative. Finally, implicit surfaces have also been recently used in real-time skinning applications [Vaillant et al., 2013, 2014].

5 NON-MANIFOLD EMBEDDING FOR GEOMETRY AND CONTACT

5.1 Non-manifold Embedding

We employ an embedded simulation similar to other authors, who used regular lattice embeddings for performance [Müller et al., 2004, Rivers and James, 2007, McAdams et al., 2011]. However, due to the presence of extremely thin incisions common in surgical models, standard lattice embedding would not be able to resolve the tissue topology, unless an extremely high resolution embedding was used. We thus adopt a non-manifold lattice-derived embedding discretization in the spirit of Virtual Node or XFEM methods [Molino et al., 2004, Sifakis et al., 2007, Nesme et al., 2009]. In this chapter, we describe how these non-manifold embedding structures can be easily constructed and how they can be used for handling contact scenarios in addition to geometry representation. The following chapter will discuss how these structures can be further optimzed for parallel processing.

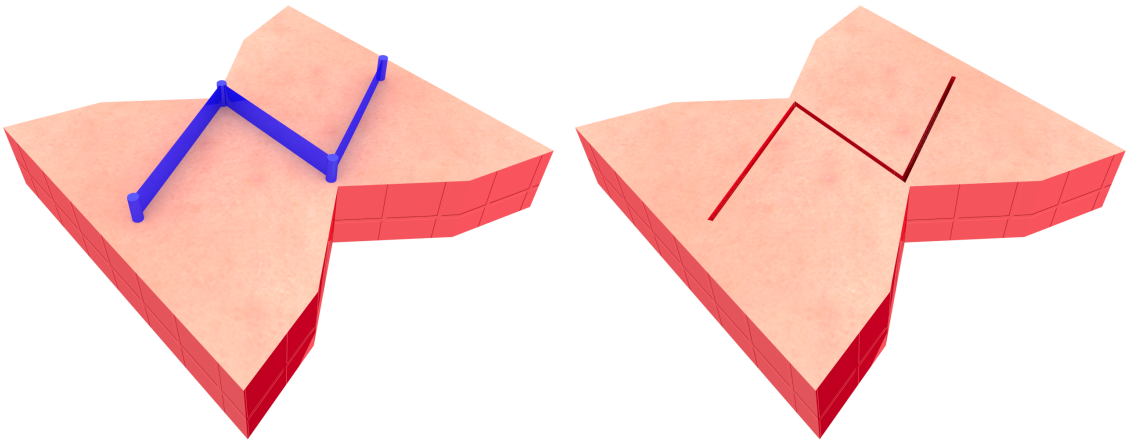


Figure 5.1: Illustration of incision technique
Incisions in the flesh surface model are created by extruding and thickening user specified line segments.

Surface Model

Prior to elasticity discretization, a watertight surface model of the flesh, including any incisions, must be created. The method choosen to generate these models is immaterial to our embedding algorithm, but for completeness we present our solution for incising surgical tissue models. In our system, incisions are generated from user specified line segment curves, which guide constructive solid geometry (CSG) difference operations to produce cut surface meshes. We begin from a user specified line segment curve from which we construct prisms by thickening the line segments tangentially and perpendicularly along the surface normal. We then apply these prisms in a subtraction operation with the surface, resulting in a slightly thickened incision (Figure 5.1). Disconnected regions produced during this step can be marked and removed by the user. Note that for scenarios involving malignant tissue, discarding of excised tissue is commonplace.

Rasterization

Given a cut surface mesh, we first create a fine rasterization of the surface. The resolution of the rasterization is selected to capture all desired topological detail (typically an order of magnitude finer than simulation resolution). In Figure 5.2, it is possible to see the fine rasterization grid in contrast to the coarser simulation resolution. The rasterization is performed by detecting all voxels intersected by the object surface and flood-filling to mark the volumetric material region. Once the rasterization is complete, subsequent embedding operations are purely combinatorial and not sensitive to poor conditioning of surface mesh elements. Additionally, this fine-grid embedding can also act as an interface layer to more complex embedding schemes, such as the non-manifold approach described next. We leverage this by translating any deformation results back to the fine-grid embedding prior to rendering, to hide non-manifold embedding or numerical solution details from the visual front-end.

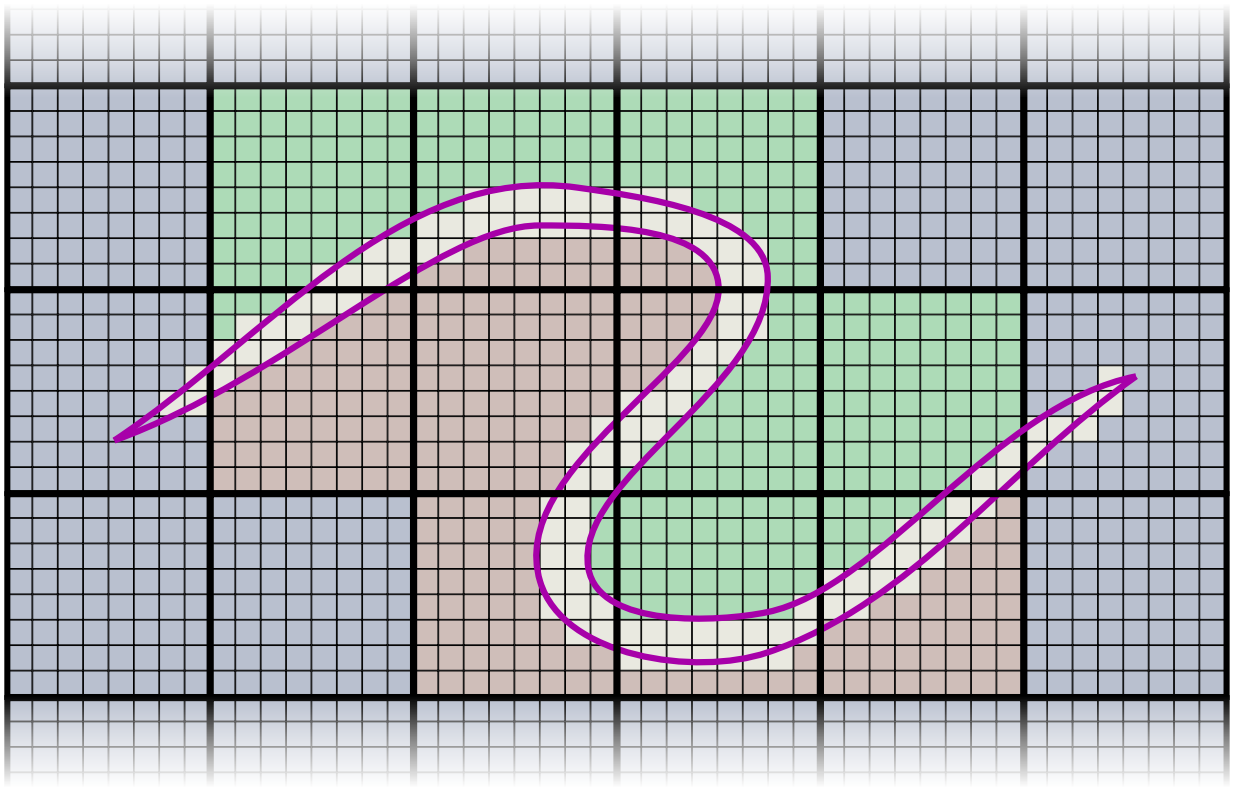


Figure 5.2: Illustration of the fine grid rasterization of a cut
Fine cells within the cut are empty, and colored to show material continuity.

Non-Manifold mesh generation

We now seek to construct a coarser resolution explicit mesh discretization, which is allowed to be non-manifold in regions, as shown in Figure 5.3. We will extend the paradigm of non-manifold embedding proposed by Teran et al. [2005a] and Sifakis et al. [2007] using the precomputed fine grid rasterization to answer material connectivity predicates. Our non-manifold mesh generation process is outlined in Algorithm 5.1, and illustrated intuitively in Figure 5.3. Note that, in the pseudocode provided, there are two geometric predicates being used: (a) Determination of material components (line 3) requires the identification of all disconnected components of material present in the intersection of our domain with a given lattice cell. (b) Adjacent-element material continuity (line 10) is a predicate invoked to determine if two material fragments, associated with adjacent lattice cells, exhibit material

Algorithm 5.1: Non-Manifold Simulation Mesh Construction

This algorithm describes the steps required to identify material connectivity across voxel boundaries, and generate appropriate voxel topologies which respect this underlying material toplogy. Material continuity across voxel faces results in vertex collapse. While this can lead to loss of simulation resolution near incision points, it doesn't affect the embedding process and is less complicated to simulate.

Input: Coarse Resolution

```
1:  function GENERATE_NONMANIFOLD_MESH
2:    for all Coarse Cells: i do
3:      C ← DETERMINE_MATERIAL_COMPONENTS(i)
4:      for all Components in C do
5:        Instance separate copy of i
6:        Generate unique, separate DOFs
7:        Assign descriptor of material content
8:    for all Geometrically adjacent cell pairs: (i, j) do
9:      for all Pairs of duplicates from i and j: (h, k) do
10:       if MATERIAL_IS_CONTINUOUS(h, k) then
11:         Mark shared vertices as equivalent
12:   for all Coarse Cells: i do
13:     Compare all duplicates of i
14:     Collapse duplicates with equivalent DOF's
```

Output: An explicit, possibly non-manifold mesh

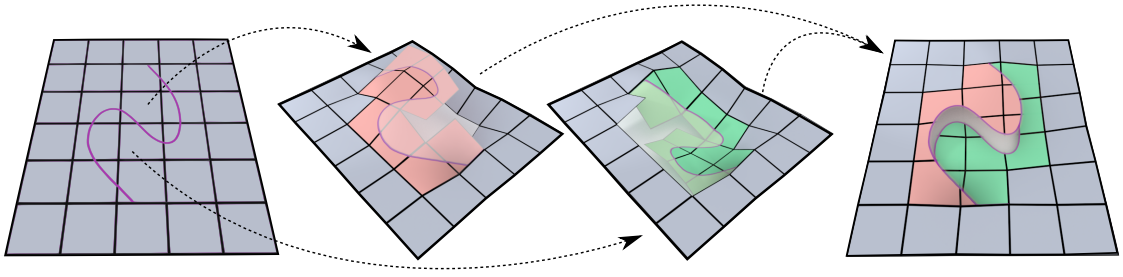


Figure 5.3: Illustration of a cut generating a non-manifold lattice
(a) A cut passing through the grid. (b) Mesh cells generated for the top half of the cut. (c) Mesh cells generated for the bottom half of the cut. (d) Cut surface is colored to show cell assignment.

continuity across their common face. These two geometric predicates are expressed in a fashion that is agnostic to the underlying geometric representation of material; in Teran et al. [2005a] the assumption is that a tetrahedralized model of the material is available, while Sifakis et al. [2007] define material fragments indirectly, by specifying cutting surfaces instead. In our case, the availability of the fine-grid rasterization makes both such operations purely combinatorial in nature. Material fragments within a coarse cell are computed via flood-fill, and fragments on adjacent cells are continuous if they contain adjacent *fine* cells on their rasterization. At the conclusion of this step, we have produced a coarse mesh (with explicitly stored connectivity), whose topology is as close as possible to the embedded geometry.

This generated mesh is now suitable for simulation, allowing us to correctly embed opposite sides of a cut or thin feature in topologically disconnected cells. However, beyond simulation, this approach can also be valuable for detecting and handling contact scenarios under the context of soft-body self collisions. The remaining sections in this chapter will demonstrate how this embedding technique can be adapted to this purpose via a technique we call non-manifold level sets.

5.2 Level Sets & Collision Processing

Before we discuss the details of non-manifold levelsets, it is important to review the basic concept of a level set and how it can be used for collision handling. Nearly three decades after their introduction [Osher and Sethian, 1988], level sets have evolved into one of the most widely used representations of geometry, alongside traditional alternatives such as meshes, splines and subdivision surfaces. A level set implicitly represents a domain boundary $\Gamma = \partial\Omega$ as the zero-value isosurface (i.e. zero *level set*)

$$\Gamma = \{\vec{x} \in \mathbf{R}^n \mid \phi(\vec{x}) = 0\} \tag{5.1}$$

of a scalar field $\phi(\vec{x})$[1] measuring signed distances to the boundary of the object $\Omega \subset \mathbf{R}^n$. Level sets allow for fast $O(1)$ time point-object intersection queries or point projections to the object surface. Model deformation is also possible, including topological split and merge operations, simply by varying the underlying scalar field. Level sets are used in a diverse range of applications including surface editing [Museth et al., 2002], tetrahedral meshing [Labelle and Shewchuk, 2007], scattered point interpolation [Zhao et al., 2001], fluid simulation [Osher and Fedkiw, 2002] and collision processing for deformable solids [Gascuel, 1993], rigid bodies [Guendelman et al., 2003] and skinning animations [Vaillant et al., 2013].

Level set collisions for volumetric solids

Self-collision processing is paramount in generating visually attractive and realistic shapes, as is evident in character skinning pipelines [McAdams et al., 2011, Vaillant et al., 2013]. Handling collisions in volumetric solids can be quite different than the typical cloth collision pipeline. With volumetric solids there is a clear distinction between an *inside* and an *outside* region, making it possible to process collisions in a single time instance of a simulated deformation. In contrast, if collisions are detected in a cloth simulation, we need to rely on deformation history to determine how the cloth surface is to be untangled (unless global intersection analysis [Baraff et al., 2003] is performed). While cloth simulations typically strive for a collision-free state at all times, commonly enforced via impulses in semi-implicit integration schemes [Bridson et al., 2003], volumetric objects can tolerate occasional, limited interpenetration, and respond to collision with penalty forces which are more easily coupled with explicit integration schemes. In this section, we review a level-set assisted technique that capitalizes on these opportunities to handle volumetric object collisions in large time-step implicit integration schemes, without requiring a collision-free deformation history. In section 5.3 we explain when standard level sets are inadequate for this task, and use this as motivation for our non-manifold variant.

[1] It should be noted that the function $\phi(\vec{x})$ used here is different than the deformation map $\phi(\vec{X})$ described in Chapter 3, despite the use of the same symbol.

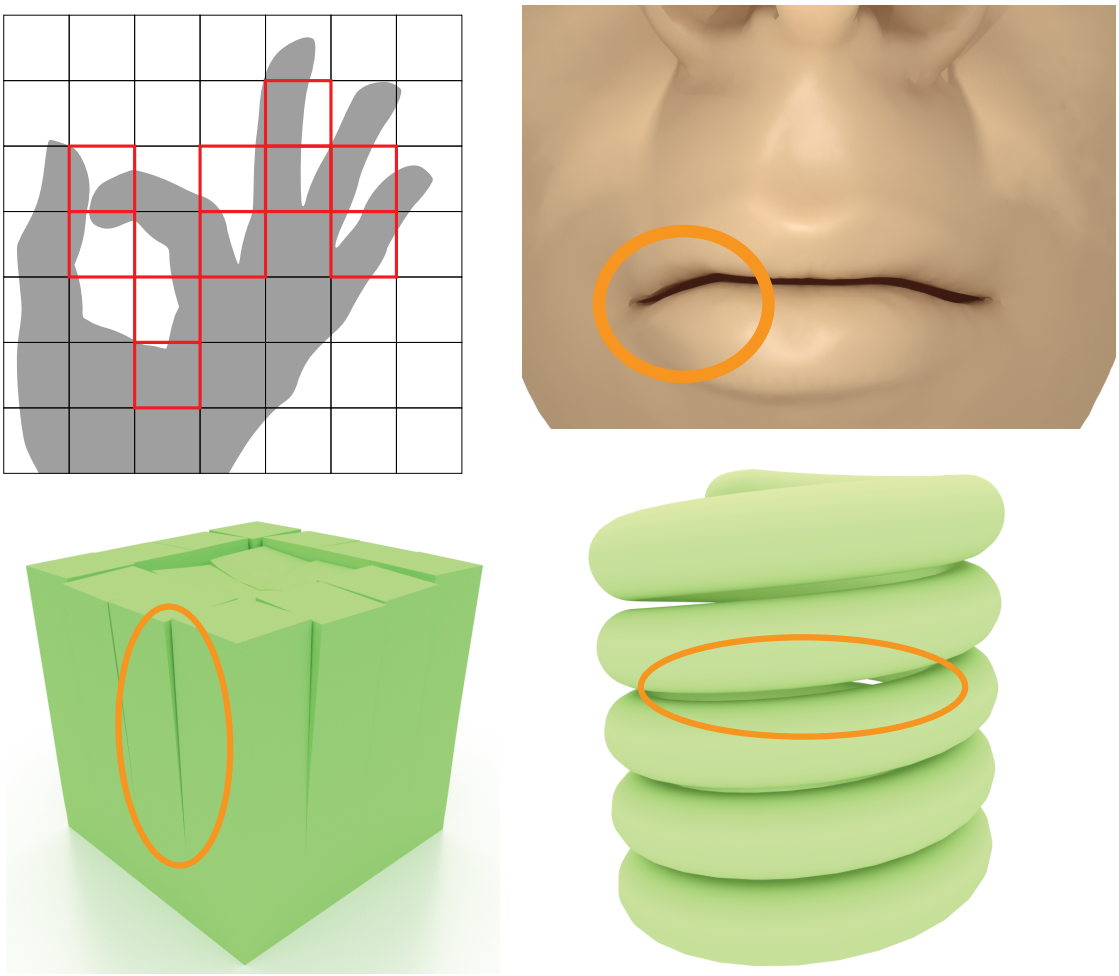


Figure 5.4: Illustration of cases poorly handled by conventional level set discretizations Scenarios where standard Cartesian grid-based level sets lack the expressive ability to resolve thin features. For the vector art hand (top left), the cells highlighted in red show features that will not be resolved. While many cases can be resolved with fine enough resolutions, the fractured cube (bottom left) is an instance that cannot be resolved with conventional level sets.

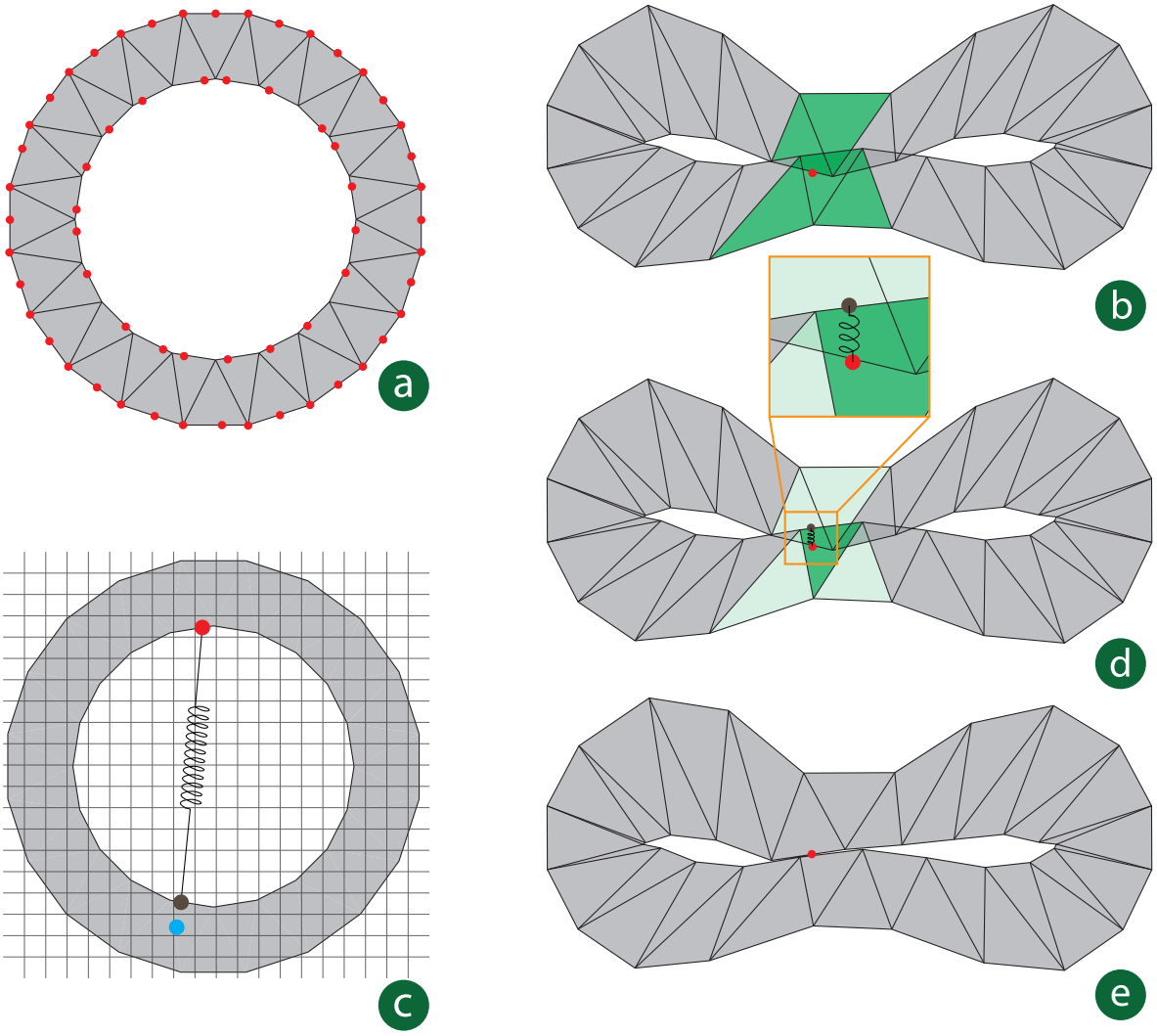


Figure 5.5: Illustration of the self-collision pipeline
(a) a triangulated torus model is pictured in its undeformed configuration. Collision proxies on the surface shown in red. (b) The torus is deformed into a self-colliding state. A bounding box hierarchy yields initial candidates of triangles colliding with the proxy. (c) After pruning false positives, the material location that the proxy collided onto is identified, and mapped back to the undeformed configuration (blue dot). The level set (stored on the pictured grid) is used to project to the closest surface point (brown dot). (d) A zero-rest spring is initialized between the proxy and its surface-projected target. (e) The deformed torus after the self-collision is resolved.

Our collision pipeline consists of two stages: In the detection stage, discrete material points (labeled *collision proxies*) are checked for collision against the object interior. In the response stage, we use a spring-like penalty force to push each colliding proxy to the object surface [Teran et al., 2005b, McAdams et al., 2011]. Since simulated solids are typically endowed with elastic material models that prevent (or discourage) inversion, in all our examples we chose to only seed collision proxies on the object surface, as internal non-inversion combined with boundary non-collision would imply a globally non-intersecting state. Interior collision proxies can also be used, if desired, with no algorithmic change. Figure 5.5 illustrates the detection and response process on an elastic torus model squished into self-collision. For any colliding proxy, we identify the offending (internal) material location that the proxy collided with. The closest surface point to that material location is calculated, and a zero rest-length spring is introduced between that surface location and the original proxy. This spring remains active just until the collision detection phase is repeated; typically for one step of the time integration method employed, or for just a single Newton iteration in an implicit scheme. The most costly predicates in this process are (i) detecting whether a proxy intersects the object interior, and (ii) projecting the offending location to the model surface. Both predicates could be answered in $\mathsf{O}(1)$ if a level set representation of the model was available; unfortunately, the continuous deformation makes updating an implicit representation impractical. Hence, we opt for an *approximate* algorithm [McAdams et al., 2011] that only relies on a level set representation of the *undeformed* model.

For simplicity, let us assume that the deformable volumetric solids are tetrahedralized (we can make this choice without loss of generality - the following algorithm applies equally well to hexahedral discretizations). Let E_i denote the i-th simulation element in the undeformed configuration and e_i denote the same element in the deformed configuration. Similarly, let P_i denote the location of the i-th collision proxy in the undeformed configuration and p_i denote its deformed counterpart. Collision proxies can be regularly sampled on the *surface* of the simulated object, or the surface vertices of the embedded object themselves can be used

as proxies, if their distribution is reasonably regular[2]. Let ϕ denote the level set function for the simulated volume in the undeformed configuration. The collision handling routine performs the following steps for each proxy $\mathtt{p}_\mathtt{i}$:

Step 1 The set of (deformed) elements $\mathcal{E} = \{\mathtt{e}_{\mathtt{i}_1}, \mathtt{e}_{\mathtt{i}_2}, \ldots, \mathtt{e}_{\mathtt{i}_k}\}$ are checked against $\mathtt{p}_\mathtt{i}$ for intersection. This is performed as follows:

(a) We use an axis-aligned bounding box hierarchy, defined over all deformed elements, to identify all elements whose bounding box intersects $\mathtt{p}_\mathtt{i}$, i.e. $\mathcal{E}_{\text{int}} = \{\mathtt{e}_\mathtt{k} \in \mathcal{E} | \mathtt{Box}(\mathtt{e}_\mathtt{k}) \cap \mathtt{p}_\mathtt{i} \neq \emptyset\}$.

(b) We identify the element $\mathtt{E}_\mathtt{i}$ that contains the proxy $\mathtt{P}_\mathtt{i}$ in the undeformed configuration. This may be more than one element, e.g. if $\mathtt{P}_\mathtt{i}$ was a mesh vertex. We trivially have that $\mathtt{e}_\mathtt{i} \in \mathcal{E}_{\text{int}}$, as $\mathtt{p}_\mathtt{i}$ is embedded in it. Similar to McAdams et al. [2011], we prune $\mathtt{e}_\mathtt{i}$ along with all of its immediate topological neighbors from $\mathcal{E}_{\text{int}}$, since collision response between primitives that share embedding parents can be problematic (instead, we rely on elasticity to discourage extreme cases of local collision).

(c) We perform an exact intersection test between any elements $\mathtt{e}_\mathtt{t}$ that have not been already pruned. We do so by computing the barycentric coordinates of $\mathtt{p}_\mathtt{i}$ with respect to $\mathtt{e}_\mathtt{t}$, and discard elements if those coordinates are out of bounds.

Step 2 For every colliding proxy, we identify the location $\mathtt{X}_\mathtt{t}$ in the *undeformed* configuration of the material point the proxy impacted[3]. We do so using the barycentric coordinates computed in step 1(c) to interpolate $\mathtt{X}_\mathtt{t}$ from the *undeformed* colliding element $\mathtt{E}_\mathtt{t}$.

[2]Proxy spacing is an important task, though somewhat tangential to this conversation. Too few proxies in a region can lead to poor collision response or excessive penetration. Too many proxies can lead to an over-constrained problem, resulting in slow convergence or other unfortunate artifacts. For our system, a Poisson-disk sampling technique [Corsini et al., 2012, Devroye, 1986] was used to generate a blue noise pattern of proxies over the model surfaces.

[3]Note that the location $\mathtt{X}_\mathtt{t}$ is unique if the element $\mathtt{E}_\mathtt{t}$ is a triangle in two spatial dimensions or a tetrahedron in three spatial dimensions, but this may not be the case for polygonal or polyhedral elements such as squares, hexahedra, etc. We first triangulate or tetrahedralize such elements, which can result in several locations $\{\mathtt{X}_{\mathtt{t}_1}, \mathtt{X}_{\mathtt{t}_2}, \ldots, \mathtt{X}_{\mathtt{t}_r}\}$ in the rest configuration corresponding to each of these several triangles/tetrahedrons. In this case, we initialize the spring between $\mathtt{p}_\mathtt{i}$ and the point $\mathtt{y}_{\mathtt{t}_j}$, where $\mathtt{X}_{\mathtt{t}_j}$ is *closest* to the surface.

Step 3 Elements E_t with $\phi(\mathsf{X}_t)>0$ are dismissed as non-colliding (this could be due to discretization discrepancy between mesh and level set, or if an embedded simulation approach is used where elements in $\mathcal{E}$ reach beyond the extent of the simulated model).

Step 4 Using the level set, point X_t is projected to the surface point $\mathsf{Y}_t = \mathsf{X}_t - \phi(\mathsf{X}_t)\nabla\phi(\mathsf{X}_t)$, for all elements E_t, where $\mathsf{e}_t \in \mathcal{E}_{\text{int}}$.

Step 5 In the deformed configuration, a zero rest-length spring is initialized between points p_i and y_t to resolve the collision.

In step 5, y_t corresponds to the point Y_t in the deformed configuration. Note that our algorithm, in steps 3 and 4, relied upon a level set representation of the *undeformed* shape of the simulated model. The cost paid for this convenience is that the surface location y_t (the collision target) is only an approximate surface projection in the *deformed* configuration; nevertheless, this disparity vanishes as the effect of the collision springs progressively brings the penetration depth closer to zero.

Figure 5.5 illustrates the individual steps of the algorithm on a torus in two spatial dimensions.

5.3 Non-manifold Level Sets

In principle, based on equation (5.1) a level set could represent any object $\Omega \subset \mathbf{R}^n$. In practice, however, the scalar field $\phi(\vec{x})$ is never provided analytically, but instead sampled at discrete points in $\mathbf{R}^n$. As a consequence, the expressive ability of discrete level sets is limited by the sampling resolution and the interpolation scheme used. In the common practice where ϕ values are sampled on the nodes of a uniform Cartesian grid, and trilinear interpolation is used to define a continuous scalar field, models with multiple boundary crossings per grid edge (near narrow gaps or strips, see Figure 5.4) cannot be represented. These issues can be alleviated to some extent by using adaptive schemes [Losasso et al., 2004, Museth, 2013] to concentrate resolution near fine features, or hybridizing with point-based methods [Enright

et al., 2002] to capture details at a sub-cell level. Nevertheless, gratuitously increasing the level set sampling resolution is a brute-force remedy which quickly becomes impractical if the thickness of topological gaps approaches zero, as is commonly the case with geometries arising from cutting and fracture modeling pipelines (see Figure 5.11(top)). It is also unfortunate that even though level sets are perfectly capable of localizing the implicit surface to sub-cell resolution (trilinearly interpolated level sets on Cartesian grids converge quadratically to surfaces of bounded curvature) they cannot resolve multiple interface crossings within a single cell.

Furthermore, the self-collision algorithm outlined in Section 5.2 works well only when a good quality level set can be computed from the model's undeformed configuration. In such cases, it provides the opportunity for excellent performance, even allowing interactive simulation for highly detailed models [McAdams et al., 2011], as it allows very aggressive integration time steps (tolerating occasional mild inter-penetration) and exploits the fast intersection/projection level set queries. The approach breaks down, however, in cases where the object cannot be resolved by the level set resolution. As a brute-force remedy, it might be possible to pose a model in a *reference configuration* that avoids thin features (e.g. modeling a hand such that the fingers are generously separated [McAdams et al., 2011]). However, this pre-processing can be tedious (e.g. for faces with narrow clearance between the lips), unnatural (if the "reference pose" is not really a *rest* pose, see the elastic coil in Figure 5.4), or impossible to perform a priori if the thin features arise during simulation (e.g. cracks and cuts). We propose a principled remedy, designing a new implicit geometry data structure that fully supports the necessary geometric predicates, but accommodates models with narrow gaps or even material overlap.

We argue that these apparent limitations of level sets are not intrinsic defects of the implicit representation (equation 5.1), but consequences of the data structure (e.g. Cartesian grid) conventionally used to store the signed distance values. Instead of using $\mathbf{R}^n$ as the domain of $\phi(\vec{x})$, we propose to define this scalar field over an explicit quadrilateral (2D) or

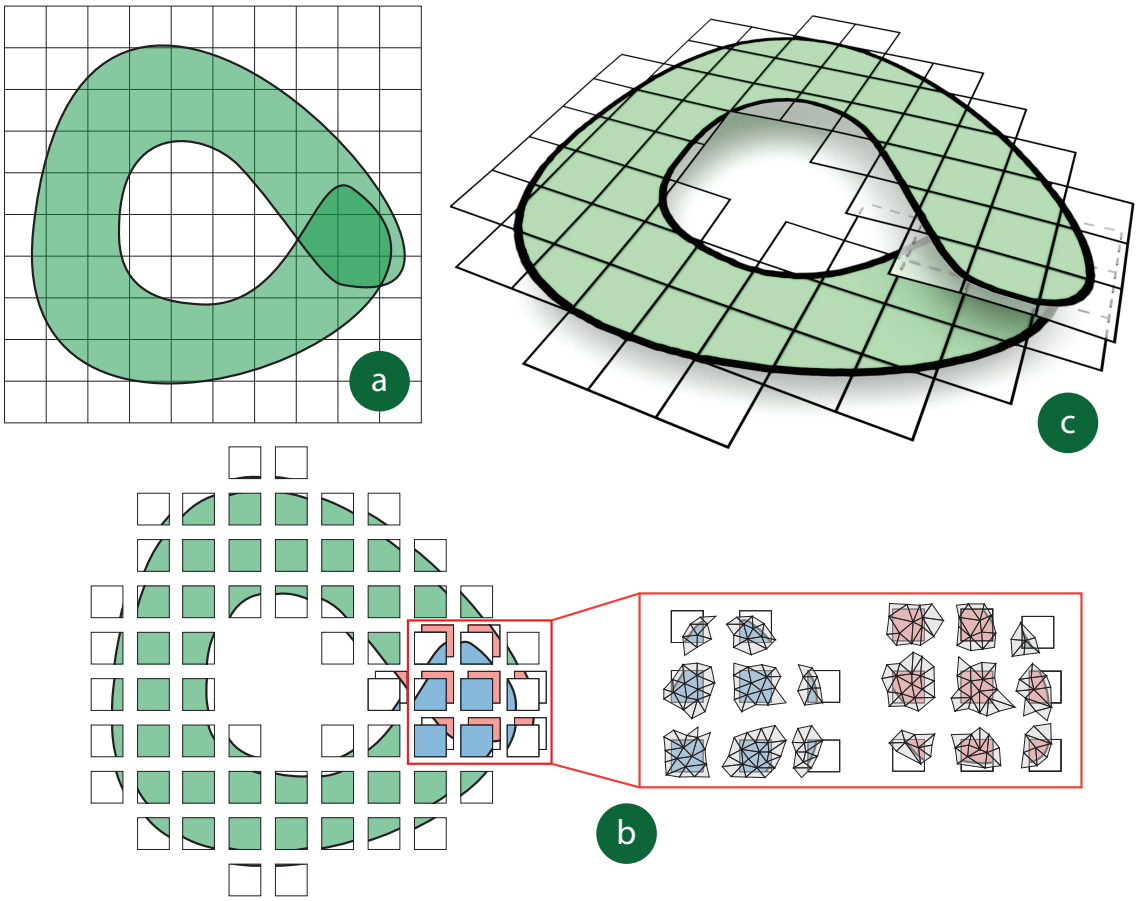


Figure 5.6: Illustration of non-manifold mesh construction
(a) A self-overlapping 2D model with template mesh overlaid. (b) Duplicate elements created during non-manifold embedded mesh generation, along with their associated material fragments. (c) Final non-manifold embedding mesh.

hexahedral (3D) mesh. We use regular (square or cube) elements in these meshes, identical in shape to the cells of a conventional Cartesian grid. However, the explicit connectivity in our mesh allows us to have multiple overlapping elements associated with geodesically distant regions (see Figure 5.6). Furthermore, this enables us to introduce non-manifold connectivity to capture topological bifurcation at the tip of a crack or incision, or in the vicinity of highly concave regions (see Figure 5.7).

Review: Basic non-manifold embedding

Models such as the ones illustrated in Figure 5.4 have been known to pose challenges not just for level set generation, but also for certain dynamic simulation techniques even in the absence of collision processing. Of course, simulation of elastic deformation is a straightforward proposition, e.g. using the Finite Element Method [Sifakis and Barbic, 2012], if an explicit tetrahedral mesh representation of the model is available. However, if we wish to use lattice deformer techniques, capable of high degrees of performance optimization [Rivers and James, 2007, McAdams et al., 2011, Mitchell et al., 2015], we run the risk of "tying" together disconnected material regions if they are separated by a distance smaller than the embedding mesh resolution (e.g. adjacent helices of the coil, or the two lips of the face model pictured in Figure 5.4). Fortunately, as we saw earlier in this chapter, we have algorithms designed for constructing embedding lattices capable of handling these thin incisions and fine features. These methods add non-manifold connectivity to the embedding mesh, duplicating elements and degrees of freedom as necessary to best capture the embedded model topology.

Before discussing the details behind our non-manifold level sets, we will take a moment to review a common formalism of the non-manifold embedding process [Sifakis et al., 2007]. The algorithm is illustrated in Figure 5.6. This algorithm is very similar to the one presented in Algorthim 5.1, except here we are assuming a material predicate defined by a triangulation (or tetrahedralization in 3D). We do this partially to show how the algorithm adapts to multiple material description predicates, but also to support models with zero width cuts and overlapping geometry. These later aspects are not supported by the discrete fine grid as presented earlier in the chapter. The choice of material description is somewhat arbitrary - a decision made around what geometric features one needs to capture.

Input: (a) A geometric description of the shape to be embedded (the green-shaded area in Figure 5.6). For simplicity, we may assume the geometry is given as a triangulated model, which allows us to express the self-overlap in our specific example. (b) A mesh which will be used as a *template* for our embedding process. In Figure 5.6(a) this is the regular quadrilateral

mesh pictured in the foreground.

Step 1 [**Element separation**] We separate each element of the template (quadrilateral) mesh, keeping track of the subset of our material model that is contained in each such element (e.g. taking note of all material triangles that intersected each quadrilateral).

Step 2 [**Element duplication for disconnected components**] For each embedding element, we identify all disjoint connected components of material contained therein (e.g. by checking connectivity of the respective material triangles). We generate a *duplicate* embedding (quadrilateral) element for each material component. Note that, at this point, all embedding elements are still disconnected.

Step 3 [**Restoring connectivity**] For any pair of *geometrically* adjacent embedding elements, we check if there is material continuity across their common face (e.g. by checking if they both intersect the same material triangle on that face). If such continuity exists, we collapse all vertices along their common face. This collapse is transitive; in the example of Figure 5.7(left) all three elements near a convex material region have acquired a common face (with non-manifold connectivity) due to transitive pair-wise vertex collapses.

The result is shown in Figure 5.6(c); after discarding embedding elements with no material content, the final embedding mesh has been fully assembled, with overlapping duplicates of elements properly connected, respecting the topology of the embedded material.

Mesh bifurcation and transition faces

The intent of our proposed level set data structure would be to store signed distance values on the *nodes* of the embedding mesh produced by the algorithm just described (to our knowledge, these *non-manifold* embedding meshes have only been previously used to store deformation data, not level set values). Of course, such signed distances would be computed geodesically, along the embedding mesh, rather than in the Euclidean sense. Subsequently, a continuous signed distance field would be computed on the embedding mesh via standard bilinear (2D) or trilinear (3D) interpolation. We note that in "simple" cases such as the example of

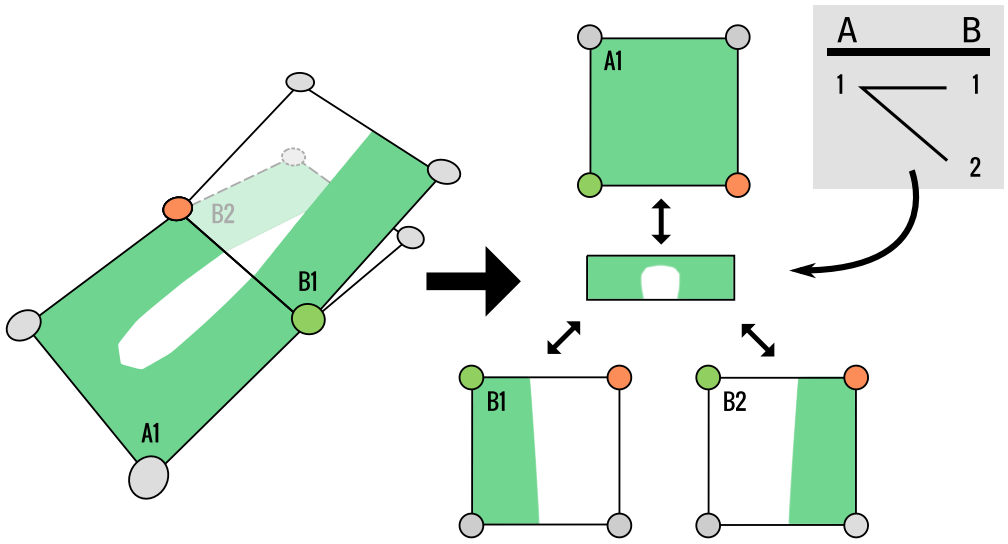


Figure 5.7: Illustration of handling material bifurcations
(Left) An example where material bifurcates at an edge in a non-manifold Cartesian embedding. (Right) Level sets can only store a single interface transition at an edge. In the non-manifold level set bifurcations are explicitly recorded in *transition faces* that record a connectivity graph between all cells on the left and right.

Figure 5.6 (where we have element overlap, but no non-manifold connectivity) this approach would have been fully sufficient. Unfortunately, scenarios such as the one illustrated in Figure 5.7(left) reveal a newfound challenge: elements hinged in a non-manifold configuration on a common face may disagree on the *sign* of the signed distance value stored on one of their common vertices. In Figure 5.7(left), elements A1 and B2 record the vertex in orange as being inside the material domain (hence carrying a negative level set value), while the same vertex is outside the embedded domain (with a positive level set value) as far as element B1 is concerned. At this point, we should emphasize that any discrete level set is an inherently approximate representation of geometry, as it depends on interpolation of signed distance value samples. The severity of this phenomenon, however, is much greater as it carries the risk of eliminating parts of the model boundary, or forcing it to spuriously appear in parts of the embedding mesh that it did not originally traverse. Note that this behavior does not affect non-manifold embedding for simulation purposes, since such techniques explicitly track the material embedded in each element, rather than using interpolated vertices.

We posit that, for the proper resolution of non-manifold connectivity, the Algorithm

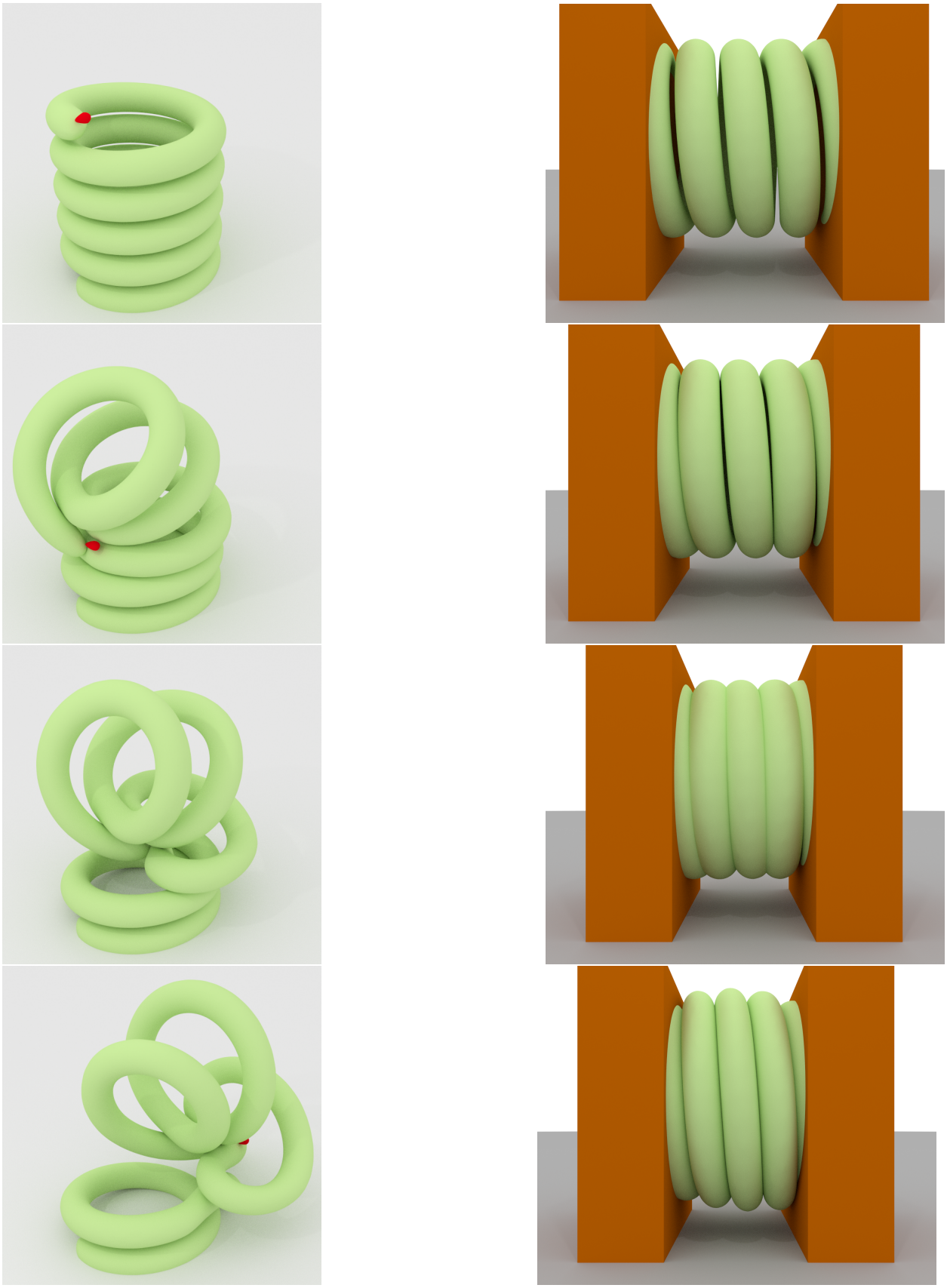


Figure 5.8: A non-manifold level set is used to correctly track self-collision of a coil (Top) A volumetric coil self-collides under user manipulation. (Bottom) A coil is compressed against two walls. Subsequently, collision handling is disabled and the geometry self-intersects (third frame in row). Self-collisions are turned back on and the coil recovers.

5.1 presented in Section 5.1 and Section 5.3 cannot be allowed to indiscriminately collapse vertices (in Step 3) based solely on material continuity if this yields a contradiction in the nodal signed distance values across connected elements. Thus, we introduce the concept of a *transition face* which encodes connectivity between incompatible (in terms of the signs of nodal distance values) materially connected elements. This construct is illustrated in Figure 5.7(right). The transition face is envisioned as an infinitesimally thin connective strip between between elements $\mathsf{A1}$, $\mathsf{B1}$ and $\mathsf{B2}$ with the appropriate internal structure as to connect the material of each element *as reconstructed from their nodal values* via bilinear interpolation. For example, we see that element $\mathsf{A1}$ is considered to be fully interior to the domain, once described by the signed distance values stored at its nodes. We explicitly store a transition face as a *connected* bipartite graph as seen in Figure 5.7, which records pairwise material continuity of elements on either side, which would normally be lost once only nodal level set values are retained for each element.

Non-manifold level set mesh algorithm

Using the transition face mechanism, we can now describe our new algorithm for generating the embedded mesh whose nodes will be used to store the signed distance values of our non-manifold level set. The entire process is outlined in Algorithm 5.2. The first phase of the algorithm is identical to the initial phase of the stock embedding algorithm 5.1. As before, given a geometric material description $\mathcal{M}$ (e.g. a tessellation of the model) we identify the material region $\mathcal{M} \cap \mathsf{E}_\mathsf{i}$ contained within the embedding element E_i from an embedding "template" mesh $\mathcal{T}$. We identify connected components $\{\mathfrak{m}_\mathsf{j}\}_\mathsf{j}$ in this set, and create a duplicate embedding element $\mathsf{D}_{\mathsf{i},\mathsf{j}}$ associated with each material component. As before, all duplicate elements $\mathsf{D}_{\mathsf{i},\mathsf{j}}$ are completely disconnected at this point.

Subsequently, we analyze material continuity on adjacent embedding elements, with the goal of reconnecting the previously separated elements into the final embedding mesh. For any two elements E_k, E_l that were adjacent in the template mesh $\mathcal{T}$, we identify the

Algorithm 5.2: Non-Manifold Level Set Mesh Construction: This algorithm is a modification of the procedure to generate a basic non-manifold embedding (Algorithm 5.1) shown previously. The modfications here account for the addition of transition faces to track the interface near material bifurcations instead of simply collapsing vertices greedily.

Input: Template Embedding Mesh $\mathcal{T}$, Material Description $\mathcal{M}$

procedure CONSTRUCT_NONMANIFOLD_LEVELSET_MESH
  ▷ **Phase 1: Duplicate elements by connected components**
  for all Elements in $\mathcal{T}$: E_i **do**
    $\mathcal{C} \leftarrow$ CONNECTED_COMPONENTS($\mathcal{M} \cap \mathsf{E}_i$)
    for all Components in $\mathcal{C}$: m_j **do**
      $\mathsf{D}_{i,j} \leftarrow$ CREATE_DUPLICATE(T_i , m_j)
  ▷ **Phase 2: Reconnect or build transition faces**
  for all Geometrically adjacent element pairs: $(\mathsf{E}_k, \mathsf{E}_l)$ **do**
    $\mathbf{G} \leftarrow$ INITIALIZE_BIPARTITE_GRAPH($\mathcal{D}_k, \mathcal{D}_l, \{\}$)
    for all Duplicates from E_k and E_l: $(\mathsf{D}_{(k,q)}, \mathsf{D}_{(l,r)})$ **do**
      if MATERIAL_CONTINUOUS($\mathsf{D}_{(k,q)}$,$\mathsf{D}_{(l,r)}$) **then**
        INSERT_EDGE($\mathbf{G}, \mathsf{D}_{(k,q)}, \mathsf{D}_{(l,r)}$)
    for all Connected subgraphs of $\mathbf{G}$: C_i **do**
      if #EDGES(C_i) $= 1$ **then** ▷ Face is Manifold
        COLLAPSE(*Vertices on common face*)
      if #EDGES(C_i) > 1 **then** ▷ Face is Non-Manifold
        REGISTER_TRANSITION_FACE(C_i)

sets $\mathcal{D}_k = \{\mathsf{D}_{k,q}\}_q$ and $\mathcal{D}_l = \{\mathsf{D}_{l,r}\}_r$ of duplicate elements that were respectively spawned from them. We examine each possible pair $(\mathsf{D}_{k,q}, \mathsf{D}_{l,r})$ drawn from these sets for material continuity across their common face. At this point, however, instead of collapsing vertices on the common face of such pairs that are found to be materially connected, we simply record this connectivity with an edge in a bipartite graph $\mathbf{G}$ defined over the sets $\mathcal{D}_k$ and $\mathcal{D}_l$. Once all pairs from $\mathcal{D}_k$ and $\mathcal{D}_l$ have been processed, we proceed to split the graph $\mathbf{G}$ into its connected components (in terms of graph connectivity, not material connectivity as in Phase 1). For every connected component (subgraph) of $\mathbf{G}$, we proceed as follows:

- If a connected subgraph contains *exactly* one edge, the duplicate elements $\mathsf{D}_{k,q}$ and $\mathsf{D}_{l,r}$ connected by that edge are guaranteed to be compatible relative to the sign of the distance value stored on their nodes, since they agree exactly on the material intersecting their (geometrically) common face. This is a consequence of this edge

being a connected component of $\mathbf{G}$, indicating that no other element is independently connected to either $\mathsf{D}_{k,q}$ or $\mathsf{D}_{l,r}$. In this case, we are free to collapse the vertices of the two duplicate elements across their common face, exactly as we did in section 5.3.

- If a connected subgraph contains two or more edges (see Figure 5.7(right)), we cannot collapse all vertices on the duplicate elements' common face, since some of these elements may disagree on the sign of the distance field stored on their nodes. In this case, we simply generate a transition face, which is encoded using the same connected subgraph, allowing the duplicate elements that are juxtaposed on that transition face to retain independent signed distance values on their nodes. As we will see in the next sections, a transition face is semantically equivalent to a "hard" topological connection (a collapsed face) for operations that traverse the final embedding mesh, with the exception of its ability to allow separate signed distance values on each duplicate element it connects.

For example, Figure 5.9 demonstrates an elastic model being cut by a user-specified fracture surface. For this, we leveraged the method of Sifakis et al. [2007] which explicitly subdivides each element of the template mesh $\mathcal{T}$ into disjoint polyhedra (a 2D analogue of this process is shown in the inset image to the right, where a "cutting curve" of line segments is used to section square cells into polygonal regions). This decomposition natively provides connectivity information, and can easily detect material continuity across embedding elements by checking adjacency of polyhedral material regions. Finally, the transitive Collapse operation (line 15) is simply implemented using a Union-Find structure which records the equivalences of vertex identifiers.

Level set operations on nonmanifold meshes

Initialization of signed distances Once the topology of the embedding mesh has been constructed, including the creation of the necessary transition faces, the embedding mesh nodes must be populated with the proper signed distance values. We start by explicitly

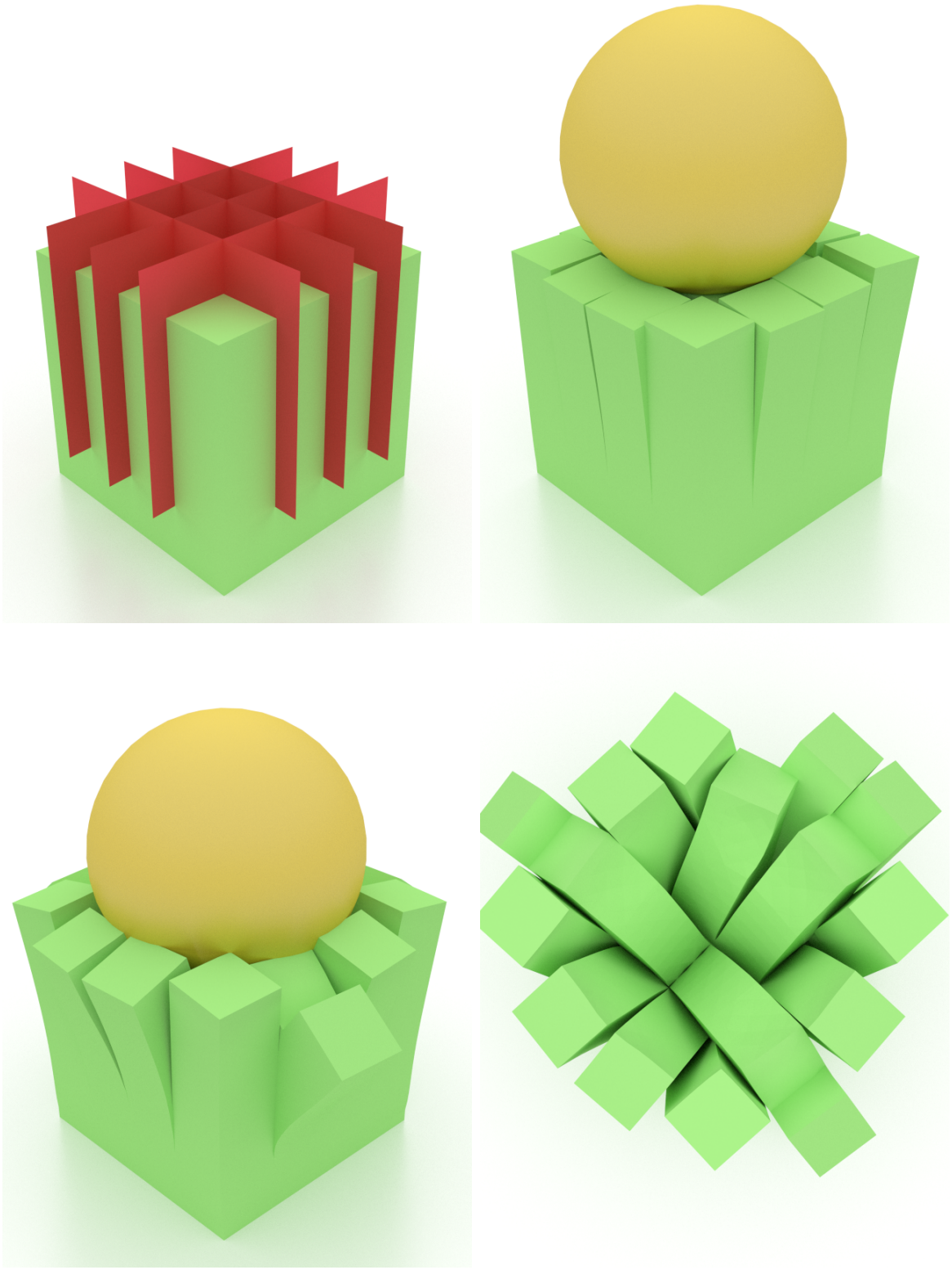


Figure 5.9: Non-manifold level sets can correctly handle zero width cuts
A cube is partially sliced by 6 planes and a ball subsequently squashes it to push the resulting 16 fingers apart. Our non-manifold level set can robustly resolve zero width cuts which could not be resolved with standard Cartesian grid-based level sets.

computing such distances on embedding elements that intersect the object boundary. Since we possess an explicit description of the material contained in each element, for each of their nodes we compute the minimum (absolute) distance from all material contained in that element. We also compute the sign depending on whether the node is inside or outside the embedded material component. Adjoining elements that have had common vertices collapsed (topologically; not connected via transition faces) will agree on the sign of the signed distance field at shared nodes, but not necessarily the magnitude. We retain the distance value with the *minimum* magnitude, across all elements incident to this node. Of course, no such reduction is performed on nodes connected via transition faces. Subsequently, we propagate the signed distance field in the interior of the object using the $O(n \log n)$ Fast Marching Method [Sethian, 1998], with the only modification that this Dijkstra-type algorithm is allowed to propagate through transition faces in exactly the same fashion as through explicitly connected nodes. While we only compute a scalar signed distance field, it would be straightforward to also compute a normal field [Kobbelt et al., 2001] to support higher quality reconstructions.

Distance queries and surface projection The basic level set predicates required in the collision pipeline of section 5.2 include a lookup of the signed distance value $\phi(\vec{x})$ at an arbitrary location $\vec{x}$ in space, and the projection of a material point to the closest location $\mathsf{Proj}(\vec{x}; \Gamma)$ on the model surface Γ. Since our embedding mesh may contain several overlapping elements, it is no longer sufficient to define such predicates as functions of just the spatial location $\vec{x}$ being queried; we also need to identify the appropriate branch of material being referred to. Thus, we reformulate these predicates as $\phi(\vec{x}, D_i)$ and $\mathsf{Proj}(\{\vec{x}, D_i\}; \Gamma)$, where the element D_i embeds the material point $\vec{x}$ in the non-manifold level set mesh. Subsequently, the result of the projection operator is also a tuple $(\vec{x}^\star, D_j)$ denoting a material point $\vec{x}^\star$ and its respective embedding element D_j.

Given an embedding element D_l and a location $\vec{x}$ embedded in it, level set value and

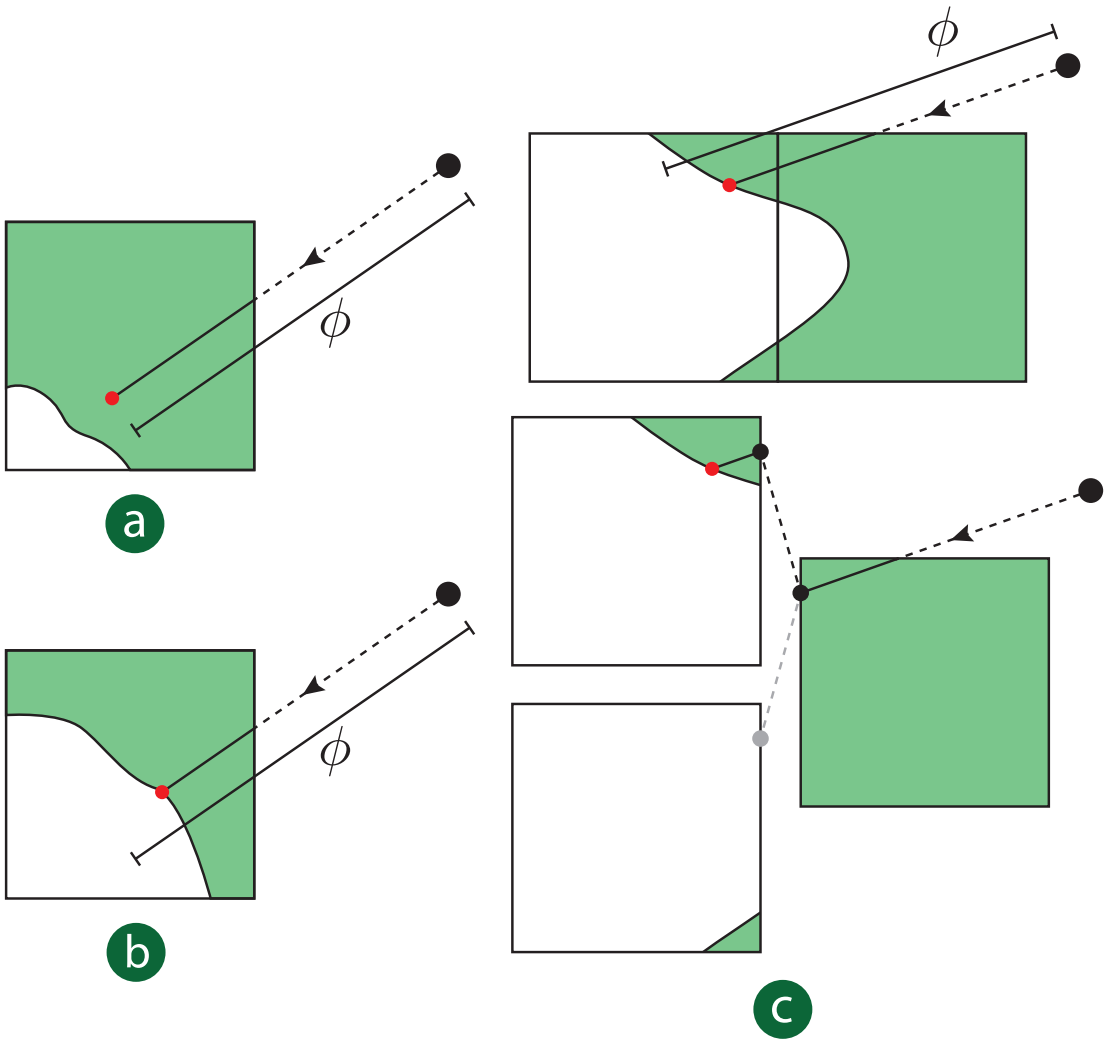


Figure 5.10: Illustration of the backtrace procedure to determine surface crossings Different surface projection scenarios. (a) Backtracing terminates after covering a distance of ϕ without intersecting the interface. (b) Backtracing stops at the first interface crossing. (c) Backtracing hits a transition face and continues into the connected neighbor that has the negative ϕ value with the smallest magnitude.

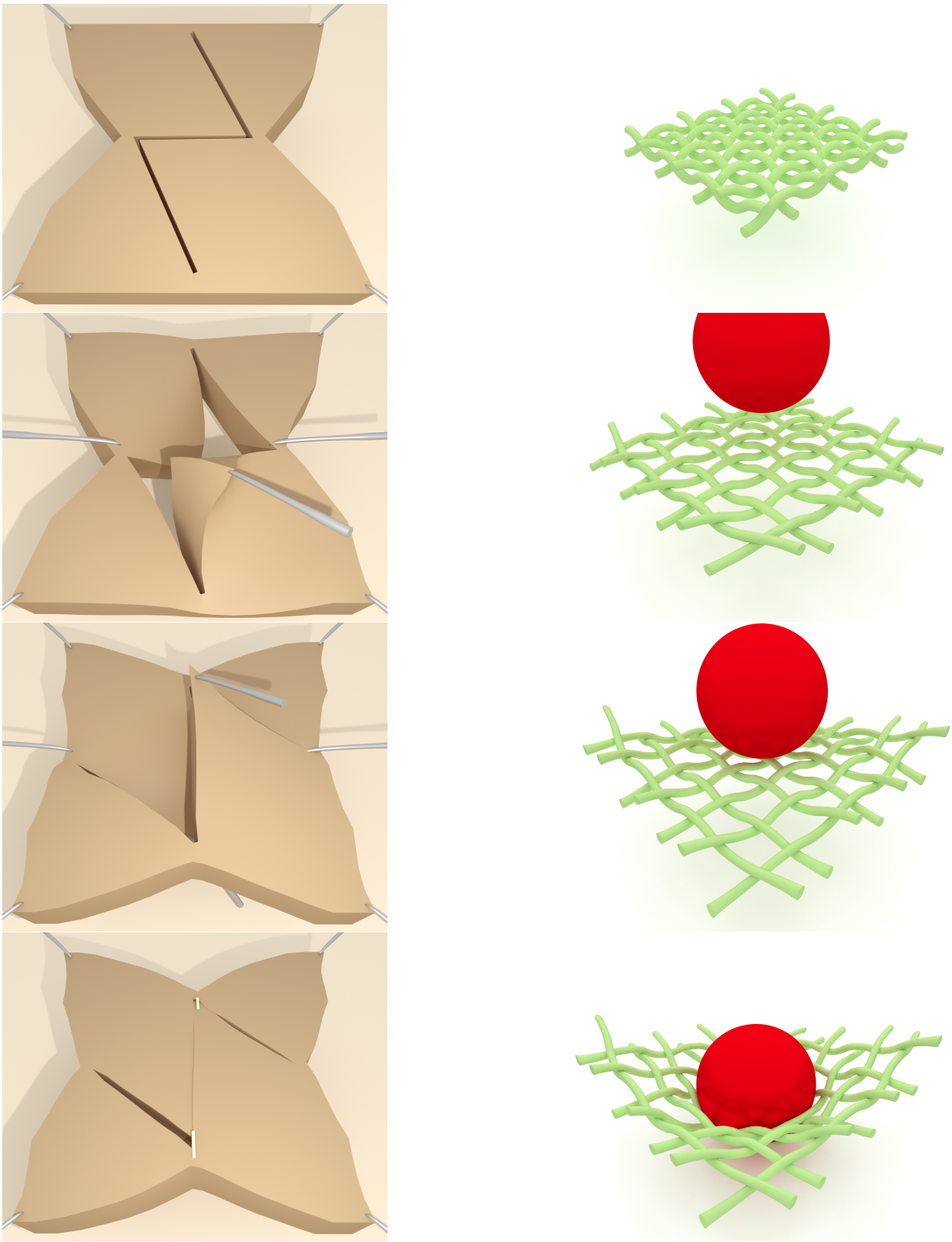


Figure 5.11: Non-manifold level sets handle surgical scenarios and complex woven geometry (Left) Surgical simulation of a z-plasty procedure, with self-collision processing. (Right) A net is stretched out, twisted to a saddle configuration and a ball is subsequently dropped on it. A single level set is used for the entire net during self-collision processing.

gradient are computed via trilinear interpolation:

$$\phi(\vec{x}, \mathsf{D}_l) = \sum_{i,j,k=0}^{1} \mathcal{N}_{ijk}(\vec{x})\phi_{ijk}, \quad \nabla\phi(\vec{x}, \mathsf{D}_l) = \sum_{i,j,k=0}^{1} \nabla\mathcal{N}_{ijk}(\vec{x})\phi_{ijk}$$

where $\mathcal{N}_{ijk}$ denotes the trilinear basis functions and ϕ_{ijk} are the signed distance values at the nodes of D_l. This formula for the gradient can be used throughout the embedding mesh, and has been fully adequate for our collision processing application. However, should higher accuracy be desired, a higher order finite difference scheme [Osher and Fedkiw, 2002] can be optionally substituted for cells exibiting manifold connectivity with all their neighbors. It is known that the gradient of the level set function, i.e. the steepest ascent direction of the distance field, is a unit normal which points in the direction of the closest point on the surface. Thus, the closest point to $\vec{x}$ on the model surface is to be found in the direction of $\vec{n} = \nabla\phi(\vec{x}, \mathsf{D}_l)$, at a distance of $|\phi(\vec{x}, \mathsf{D}_l)|$. Thus, analytically:

$$\mathsf{Proj}(\{\vec{x}, \mathsf{D}_l\}) = \vec{x} - \phi(\vec{x}, \mathsf{D}_l)\nabla\phi(\vec{x}, \mathsf{D}_l)$$

To compute the projection $\mathsf{Proj}(\{\vec{x}, \mathsf{D}_l\}; \Gamma)$ we topologically backtrace the non-manifold level set mesh along $\vec{n}$ element-by-element, to ensure that we follow a geodesic path along the embedding mesh, as shown in Figure 5.10. If while traversing a distance $\phi(\vec{x}, \mathsf{D}_l)$ along $\vec{n}$ we land in an element D_m that is crossed by the interface, then we use bisection search to compute the interface point $\vec{x}^\star$ and return the tuple $(\vec{x}^\star, \mathsf{D}_m)$ (Figure 5.10(b)). If we have traversed a distance equal to $\phi(\vec{x}, \mathsf{D}_l)$ without crossing any interface, we stop the backtrace operation and report the location reached after the requisite distance has been traveled (Figure 5.10(a)); we do so to avoid grazing by a nearby interface without actually stopping there. Finally, if the backtracing process crosses a transition face f, then we compute the point $\vec{x}_\mathsf{f}$ where the ray from $\vec{x}$ along $\vec{n}$ crosses f. We then compute the value of ϕ at $\vec{x}_\mathsf{f}$ for all elements on the other side (connected through the transition face) and choose the one with a *negative* value but with the *minimum* magnitude (to approach the surface as soon as

possible). If no such element is present, then we assume that the interface lies *exactly* at $\vec{x}_f$ and return this point along with the cell from which we entered the transition face as the result of the projection. We note that although this projection is approximate, the error is comparable with conventional, grid-based level sets, and is acceptable for collision handling.

5.4 Examples

We simulated a number of examples to demonstrate the efficacy of our method in several challenging scenarios. Figure 5.8(top) shows a user pulling a three dimensional volumetric coil at the red handle creating complex self-collisions. Figure 5.8(bottom) shows the same coil being compressed against two moving walls. Self-collisions are turned off at some point to make the geometry self-intersecting, and subsequently turned back on again resulting in the coil bulging outwards. This example shows that our method does not require any history information for resolving self-collisions. Figure 5.11(top) shows a simulation of the *Z-plasty* operation (as described in Chapter 2), while Figure 5.11(bottom) shows a ball dropping on a net that has been stretched outwards and twisted into a saddle configuration. Our method uses a single level set for the entire net during self-collision processing, obviating the need for multiple collision level sets and circumventing the complexity in bookkeeping associated with such scenarios. Figure 5.12(b) shows an example where the lower jaw of a face model is pulled down and subsequently pushed back up, opening and closing the mouth in the process. Note the slight bulge in the cheeks due to self-collisions at the lips when the mouth is closed because the jaw is pushed further up compared to the rest state. Figure 5.12(c) shows a user moving around two points on the lips (shown in orange) to demonstrate complex self-collisions that our method can resolve. Finally, Figure 5.9 shows an example where a cube is partially sliced by six planes using the method of [Sifakis et al., 2007]. This results in sixteen fingers which are pushed apart when squashed by a ball from the top. Note that a standard Cartesian grid-based level set cannot be used for resolving this structure irrespective

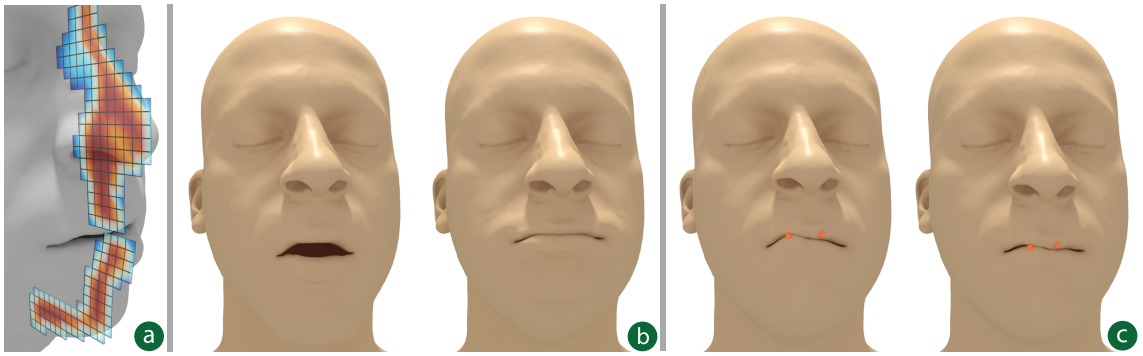


Figure 5.12: Non-manifold level sets are applicable to simulating small facial features (a) Cutaway view of the non-manifold level set generated on a face model. (b) The lower jaw is displaced vertically, opening and closing the mouth. Note the small bulges in the cheek due to self-collisions at the lips because the jaw is pushed further up compared to the rest state. (c) A user moves around two points on the lips (orange) to demonstrate the robustness of our method in resolving self-collisions.

of its resolution.

Model	Level set Gen. (s)	Solve (s)	Collision Proc. (s)	Backtrace Total (ms)	Proxy Count
Z-plasty	222.6	1.961	0.0671	0.0479	7121
Coil	580.5	13.22	0.4651	14.8	31126
Net	524.0	23.47	0.4123	5.38	48042
Face	271.0	24.46	0.2977	0.620	48851

Table 5.1: Performance results for non-manifold level set generation and collision processing Example timing comparing the cost of solving elasticity equations to running collision detection using our non-manifold level set data structure. Compared to the cost of simulation per frame, collision processing is generally insignificant. Level set generation, while currently expensive, is performed only as a pre-processing step.

This table captures the performance impact of our collision methodology. The first column lists the computation times for generating the non-manifold level set mesh; we emphasize that this is a one-time precomputation cost, before dynamic simulation even starts. The following columns list the cost for each step of our Backward Euler implicit integration scheme, divided into the solution of the linearized equations, the cost of collision processing, and specifically the aggregate cost of all backtracing operations for projecting proxies to the object surface. It can be seen that the cost of collision processing is a minute fraction of the overall simulation.

This stems from the fact that we do not require a history of collision-free states, and can take more aggressive steps than semi-implicit schemes that disallow interpenetration [Bridson et al., 2002].

6 PARALLELIZATION TECHNIQUES FOR LATTICE DEFORMERS

In this chapter, we will explore techniques for optimizing the force computation procedures for lattice deformers. Recall from Chapter 3, that our goal in solving elastic deformations is to compute forces and force differentials from current nodal positions. We codified this process in Algorithm 3.1, where we detailed the computational steps required for converting nodal positions into the corresponding nodal forces according to a material specific energy function. The important take away from this algorithm is that we are able to compute these forces on a per cell basis. On the surface, this would seem to be an excellent opportunity for thread-based parallelism: divide all the cells among available processor cores and have each core operate on its cells in isolation. Unfortunately, this is where we run into problems.

The first problem is that while all cells are functionally independent, the nodes themselves are not. When we compute forces on nodes, we are actually producing aggregate forces. That is, for any node in the lattice, we are interested in sum of all forces from all cells it is connected with. By using thread-based parallelism, we encounter a significant problem. In the context of a single thread, the processing of cells is serial. But when two or more threads, each operating on different cells, try to accumulate to a single node we encounter a serious *write hazard*. In this case, the hazard is that we don't have any guarantees that our final result will be the sum of all forces from all involved cells. Due to the behaviors of the processor caches and system memory, we might have a result equal to any one single cell's result, any combination of the cells combined, or some unrelated value. We could attempt to remove this confusion by adding a locking protocol around each node, but this would introduce significant performance penalties.

The second problem we encounter is when reading and writing nodal information. In Chapter 5, we demonstrated how non-manifold embedding meshes could be constructed to represent material geometry with thin features or incisions. However, this approach comes with a significant drawback: The explicit topology required to define the mesh removes much

of the regularity we could otherwise depend on for performance. One nice benefit of using an implicit topology for our lattices is that the memory locations of all cells and nodes can be quickly computed via a function of their geometric positions. This allows reading and writing to these locations to be done via a single memory access: the exact storage location. In contrast, the explicit topology we constructed previously has no such guaranteed relationship between storage locations and geometric positions. Instead it uses an explicit record of pointers for each cell that informs us where the memory is stored. This can hurt performance in two ways: first, there is no guarantee that memory is well ordered. Neighboring geometric nodes might be arbitrarily far apart in memory. Since modern processors load memory in linear strips, known as cache lines, this distance might require multiple loads, where a single load might have sufficed if they were closer. Second, by using a collection of pointers per cell, we are actually reading twice for every entry. The first read is to load the value of the pointer, followed by the data itself. Combined, these memory access issues can be extremely detrimental to performance, especially since modern processors operate in a regime of roughly two orders of magnitude more available computational resources than memory access rate. Despite the significant number of numerical steps required to process each cell, any delays in memory access could lead to the computation becoming memory bound.

We can deal with these problems by more carefully arranging our data for computation. Our proposed solution makes use of two concepts: hybrid grids and blocking. Along the way, we will also show how C++ templates can be employed for guiding SIMD vectorization of blocks. The following sections will cover these concepts in more detail.

6.1 A hybrid embedding lattice structure

We build on the non-manifold embedding mesh concepts discussed in Chapter 5, but now we seek to optimize these data structures for computational performance. Although the use of non-manifold embedding meshes recovers much of the topological expressive ability of

conforming meshes, it jeopardizes one of the most attractive features of regular embedding lattices, the fact that connectivity is implicit in the lattice structure as opposed to explicitly stored in a mesh. The performance impact of implicitly defined topology can be profound; the memory footprint of explicitly stored connectivity information can easily exceed the state variables themselves (e.g. nodal positions) and reduce effective memory bandwidth by necessitating indirect memory access. In this section, we strive to leverage the best of both worlds: We use an (implicit topology) Cartesian grid to capture the majority of the embedded model in regions where non-manifold duplication is not needed. We retain the topological flexibility of non-manifold embedding lattices by hybridizing this grid with an (explicit topology) hexahedral mesh used to describe regions in the vicinity of narrow slits and incisions.

Reduction and Remapping

To transform an explicit non-manifold mesh into a hybrid grid, we attempt to map as much of the explicit-connectivity mesh as possible back onto an implicit-connectivity grid to recover regularity. From the explicit mesh, we have two geometric primitives to consider for remapping: nodes and cells. For simplicity, we will first consider nodes and then cells. Figure 6.1 illustrates the results of the remapping rules below:

- Nodes are mapped to the grid if and only if they possess no duplicates.
- Cells are mapped to the grid if and only if all of their vertices have been mapped to the grid.

Each mesh cell remaining in the hybrid lattice is associated with a coordinate from the grid. This mapping will become important later when we discuss the block-based acceleration structures. After these rules are applied, the new structure must adhere to several post-conditions.

- All grid cells are composed only of grid nodes.

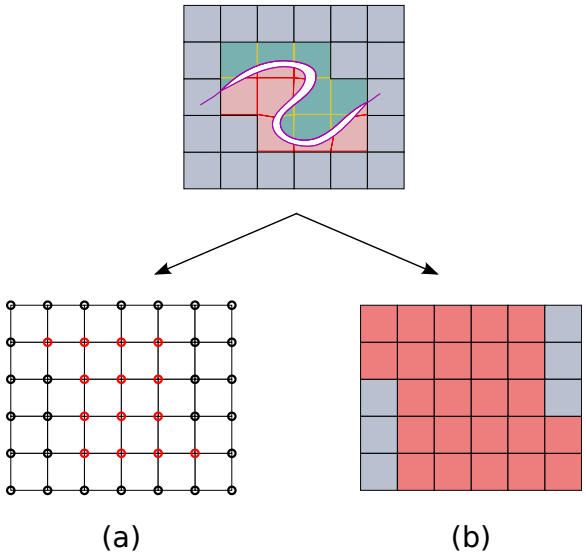


Figure 6.1: Illustration of cell type categorization
From an explicit mesh (top), we generate (a) mesh mapped nodes in red, grid mapped nodes in black and (b) mesh mapped cells in red, grid mapped cells in gray.

- Mesh cells contain one or more mesh nodes.

It should be noted that this set of rules is not strictly optimal in the sense of mapping the most elements into the grid. A more aggressive strategy would be to select one element from each set of geometrically co-located items and map it to the grid. We defer investigation of similar compaction heuristics to future work. **Note:** In these surgical examples (Figure 6.2) mesh-mapped embedding cells are indicated with blue color, grid-mapped ones in red. Typically, only a minority of cells is mesh-mapped, allowing us to retain the bulk of performance benefits of implicit grid-mapped embeddings.

6.2 Parallelization

Algorithm 6.1: General Parallelization Design Strategy

1: **for** $t = 0 \ldots N$ **do**
2: $\{o_1(t), o_2(t), \ldots, o_m(t)\} = \text{KERNEL}(\{i_1(t), i_2(t), \ldots, i_n(t)\})$

Our framework relies on both multithreading and vectorization (SIMD) to obtain the best possible performance. The fact that our Cartesian-based discretization consists of identically

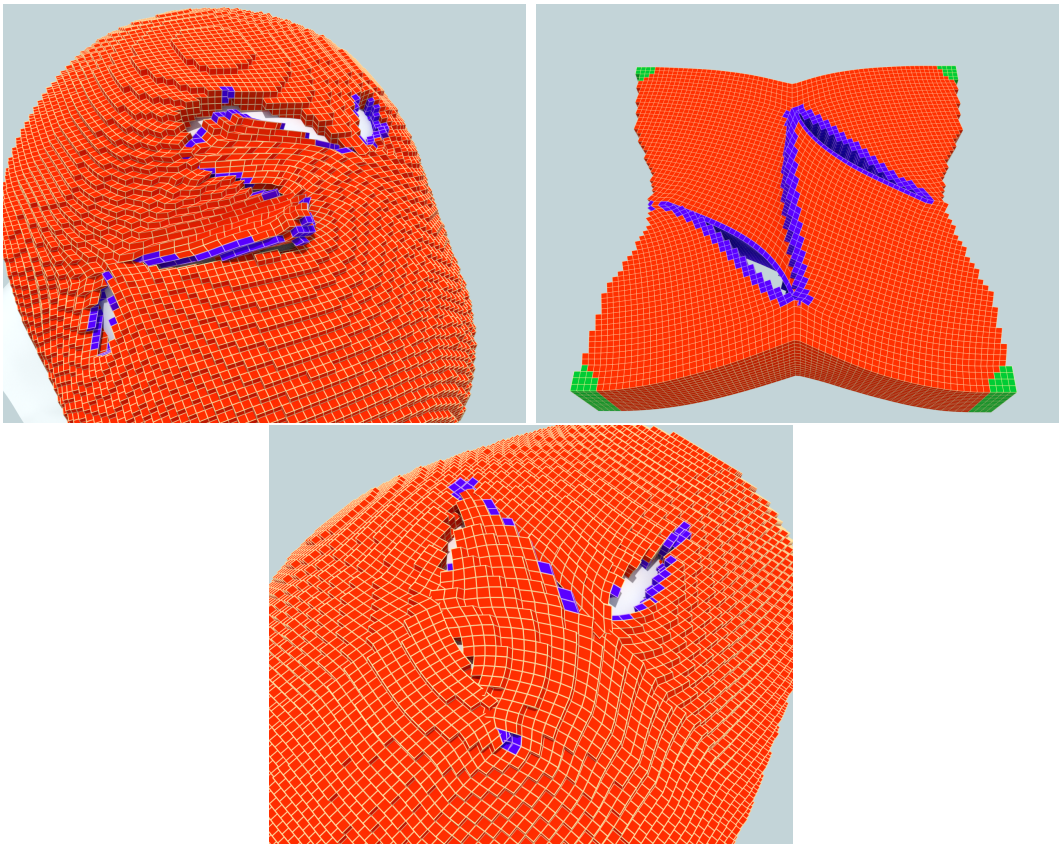


Figure 6.2: Embedding discretizations of surgical operations
Three different surgical operations, S-Plasty (Top Left), Z-Plasty (Top Right), and a Rhomboid Flap (Bottom), are shown with their embedding hybrid grids. Cells in red are part of the grid region, while blue cells are mesh mapped. Green cells mark Dirichlet regions.

shaped elements offers a great opportunity to leverage both thread-level and data-level parallelism, due to the inherent regularity of the simulation kernels. Our general strategy is to design operations that resemble the form shown in Algorithm 6.1. Under this approach, our numerical kernels for computing nodal values operate on multiple streams of input data, $\{\mathsf{i}_1, \mathsf{i}_2, \ldots, \mathsf{i}_n\}$, and produce multiple streams of output data, $\{\mathsf{o}_1, \mathsf{o}_2, \ldots, \mathsf{o}_m\}$. Collections of streams can be grouped together logically into vectors or matricies. By structuring the computation in this form, we can clearly see where multithreading and vectorization apply: Each thread will be assigned a partition over N, while vectorization can be used to execute multiple instances of KERNEL by stepping through the partition in strides. This also suggests a data structure design: Arrays of Structs of Arrays (AoSoA), where each struct contains the

data for each computational stride and the array of structs can be divided evenly between threads. In the next couple of sections, we will look at how the simulation data, currently arranged in a hybrid grid, can be repackaged according to the AoSoA methodology, a process called blocking.

Algorithm 6.2: SIMD Compatible Block Construction

Input: Block region `i`

```
1: function GenerateBlocks
2:   for all  Cell c in i  do
3:     Create new empty block
4:     Copy c into new block
5:   Build connectivity graph between blocks
6:   repeat
7:     for all Symmetric connected block pairs do
8:       Find pair with fewest neighbor mismatches
9:     if Suitable pair found then
10:      Collapse, merging block contents
11:  until No further collapses occurred
12:  return All remaining blocks
```

Output: A collection of one or more manifold Blocks

Blocking

As described in Section 3.3 and Algorithm 3.1, forces are computed on a per cell basis. A naive multithreaded port would result in write hazards at nodal positions, unless expensive synchronization was used. Simple partitioning would eliminate this issue, but would not make efficient use of modern SIMD-enabled processors. Instead, we employ a blocking scheme to avoid write hazards while retaining a memory layout favorable for vectorization. Our objective is to redefine our "quantum" of computation from a single lattice cell, to a geometric neighborhood (or *block*) that is processed concurrently using vector operations. We adopt a block size of eight cells arranged as a $2 \times 2 \times 2$ cube. This formation allows us to fit blocks into eight-wide vectors and later we will demonstrate how we can adapt to larger and smaller vector widths. In this way, each cell in the block can be considered a "channel" in the vector.

Blocks are tiled together to cover the extent of the lattice. However, restricting the contents of a non-manifold hybrid lattice to the spatial extent of a single block could easily yield more than one cell at each position in the block, as illustrated in Figure 6.3. To create blocks without overlapping cells, we employ a greedy algorithm which collects cells into manifold groupings along block boundaries, as seen in (Figure 6.3c). The full algorithm for this process is described in Algorithm 6.2.

We use the partitioning of our lattice into blocks to circumvent write dependencies during multithreaded execution. Our approach is illustrated in figure 6.4. Prior to the execution of any kernel involving force computation, we copy the state variables from either the grid or mesh structures that natively store them, into duplicate copies for every block. We label this process a **Compaction** step, which is essentially a *gather* operation that yields a representation of the state variables into a flattened array of blocks (with shared variables duplicated across blocks). Of course, this step entails creating multiple copies of data, but is not as expensive as if a separate copy of all nodal data was made for every individual voxel (the practical data overhead is $< 3\mathsf{x}$ for this scheme, compared to $8\mathsf{x}$ for a replication of all nodes for all cells). The cost of this data duplication is reduced by the fact that additional simulation meta-data (material parameters, precomputed stress derivatives and Singular Value/Polar decompositions, if needed) which are conceptually cell-centered can be stored *persistently* in a flattened array of blocks. Once this translation is completed, fully balanced multi-threading is possible by simply subdividing the processing of this flattened array across computing threads. Within each thread, we leverage the 2^3 multiplicity of each block to compute differentials with SIMD instructions. The blocks of nodal and cell data are first copied from the heap-allocated flat arrays onto a stack-allocated copy. Then, we perform a final separation of cell and node data, creating one fully separate copy for each of the 8 voxels. Note that this operation does not incur memory bandwidth expense, since this local stack-allocated copy (typically less than $6 - 8\mathsf{KB}$ in size) is expected to be cache-resident for the duration of the computation. We have leveraged the AVX instruction set available in modern Intel CPU architectures to

process all 8 voxels of the block simultaneously. Subsequently, force computation can be executed in parallel on each block, without write dependencies, by allowing each block to record its own force contribution to the lattice nodes it touches. Upon completion of the local computation, the reverse operation, labeled **Uncompaction**, scatters and accumulates the contents of the per-block forces back to their native (non-duplicated) grid or mesh storage.

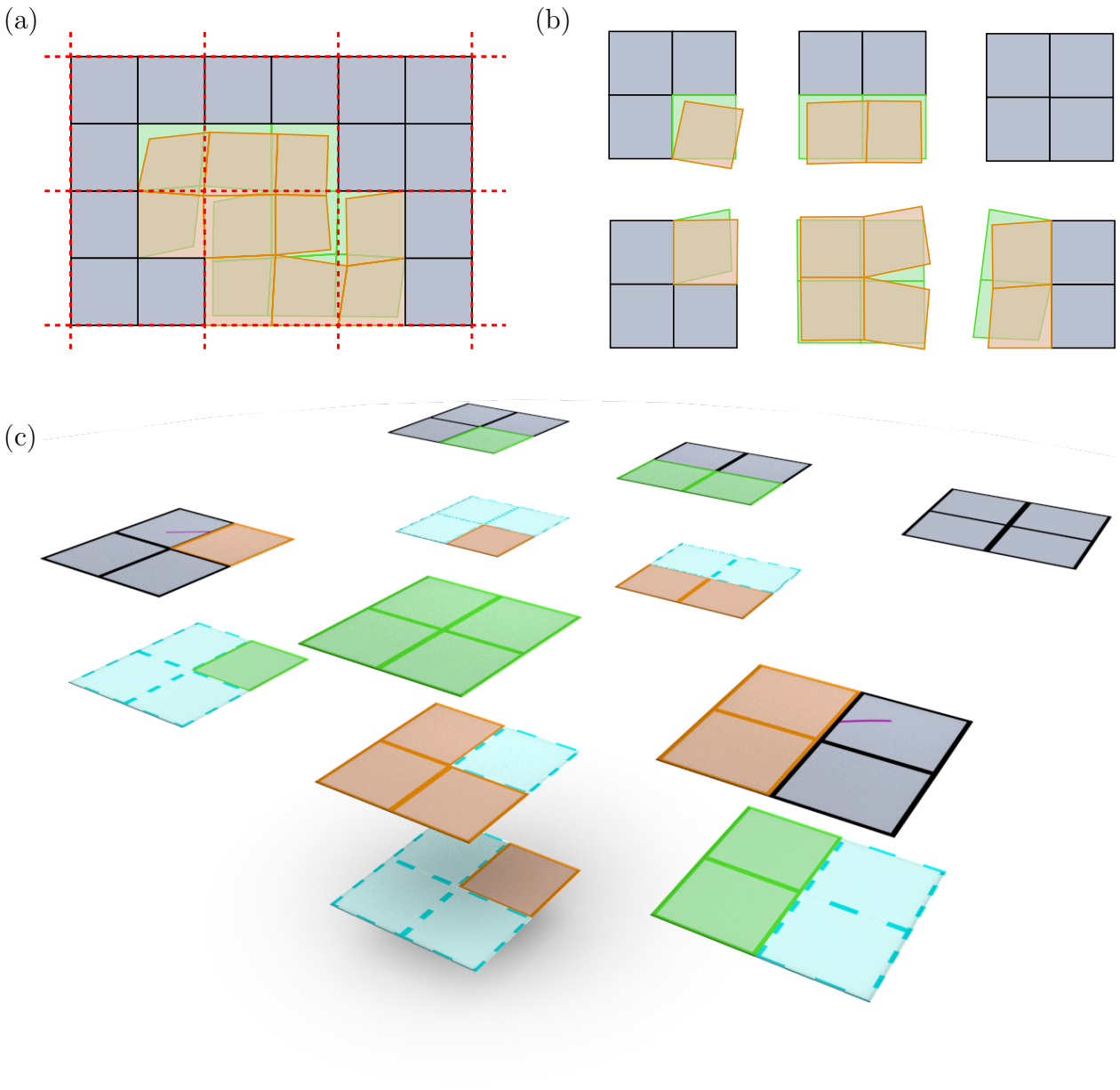


Figure 6.3: Illustration of blocks formed from regions of manifold connectivity Generating Blocks. (a) Block boundaries superimposed over hybrid lattice. (b) Non-manifold contents of each block region. (c) Final manifold blocks for each region.

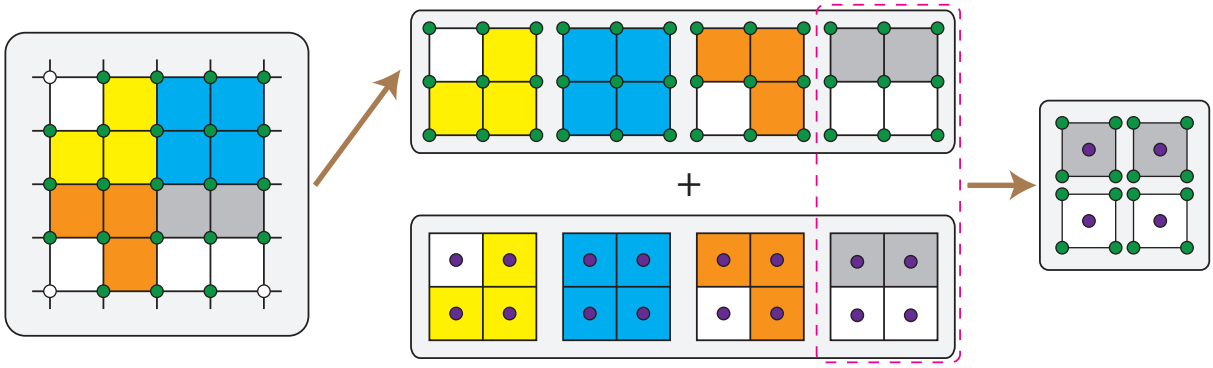


Figure 6.4: Illustration of data structure optimized for vector hardware
A 2D illustration of our simulation data structures. Left: Nodal (deformation) data stored on a grid. Middle: On demand, nodal data is copied to an array of 2^d-sized blocks and combined with cell-based data which are persistently stored in arrays of blocks. Right: Nodal and cell-centered data for a single block are copied to a stack-allocated structure, and duplicated for each voxel for SIMD computation.

Write dependencies can be avoided at this stage by partitioning this parallel operation on the grid or mesh variables that collect the per-block contributions. As a result, complex force computations can fully enjoy the benefits of thread- and SIMD-parallelism, without being concerned with data dependencies arising from the non-manifold mesh structure.

Supporting Irregular Cells

In simple cases, where all cells are composed of material or not, the previous approach for data organization works well. Even for cases where cells are allowed to be practically full of material, we can simply adjust the force computation [Patterson et al., 2012]. While this increases the complexity, it does so in a uniform fashion - all cells become more complex. The true enemy of vectorization is irregularity. Unfortunately, when we are faced with features like point spring constraints and optional material layering, useful in adding non-uniform, local anisotropic behaviors like muscle fiber effects, we quickly encounter cells which require more or less computation than their neighbors. Fortunately, we can adapt the blocking approach to handle this situation with a few minor modifications.

The primary idea we will use in this situation is the concept of a *block overlay*. A block

overlay is additional metadata applied to each block to handle optional force generating components. For each block which contains any irregular cells, we can build a block overlay data structure which contains additional per cell data. The exact description of each overlay varies, depending on its reason for existence. For instance, a spring constraint overlay would contain embedding weights and a spring stiffness coefficient for each cell with an embedded spring. By building these overlay structures at the level of whole blocks, we can easily integrate them with the thread-parallelized loop over all blocks. In order to handle cells with more than one special feature (e.g. it is reasonable to have more than one spring constraint per cell), we can repeat the same idea we used earlier when constructing the block layouts. In this case, we attempt to pack block overlays as full as possible, as long as each cell in the overlay has at most one special feature. Thus, we will generate as many block overlays as the most complex cell in the block, where the worst case is that only one cell in the overlay has non-null data. The drawback of this approach is that we have now have some amount of variable processing per block, given an arbitrary number of overlays, but computing each overlay's contribution can be done in a vectorized fashion. The entire breakdown of the process, over multiple blocks with overlays, can be seen in Figure 6.5.

Guided Vectorization

The high degree of regularity exposed by our blocking procedure naturally suggests using modern processor's SIMD capabilities to compute on all cells of a given block simultaneously. Although the performance potential is undeniable, porting code from a scalar implementation to a SIMD platform is a tedious task, one that auto-vectorization features of compilers have been traditionally ineffective in providing automatically (especially for large kernels, as the ones in our solver, which might contain thousands of machine instructions for processing forces on a single cell). An example is the highly optimized SVD routines, published with the work of McAdams et al. [2011] which replicates almost instruction-by-instruction identical SIMD intrinsics to implement scalar, SSE and AVX versions; it can be easily verified that

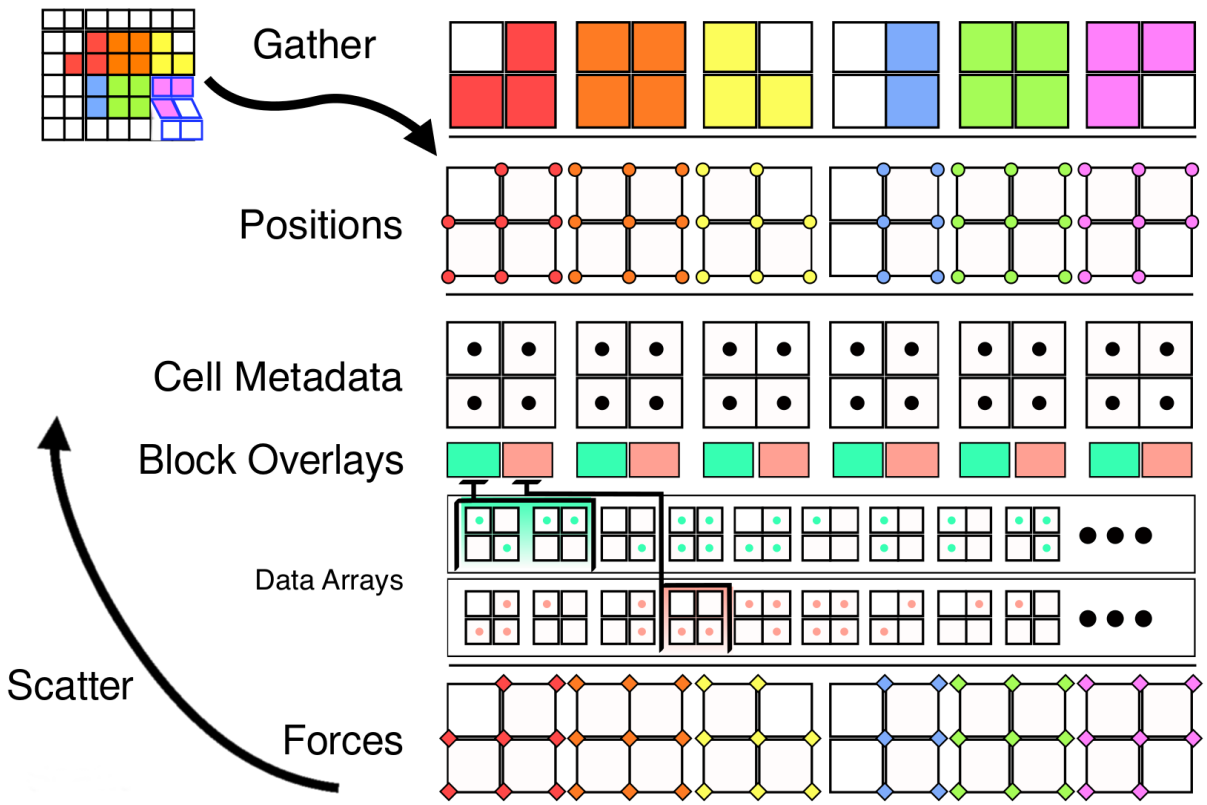


Figure 6.5: Illustration of data structure optimized for vector hardware, with overlays A complete 2D illustration of the block based vectorization computation of elastic forces, including irregular cell data overlays. From top to bottom: Gathering positional data from hybrid grid cells, combining with cell centered metadata for elastic force computation, adding block dependent overlays for force constraints and local material mix-ins, final forces are scattered back to hybrid mesh.

compiler auto-vectorization cannot provide competitive performance with these tediously hand-optimized kernels.

We have designed a programming paradigm called *guided vectorization*, with which we practically achieve the performance of hand-vectorized kernels, while only providing a single specification for scalar *and* vector variants. Our solution is object-oriented and based on the observation that the semantics of fundamental data types are very similar across scalar/vector platforms, even if the interface differs. Our system is rooted on two templatized C++ classes:

```
   template<class scalar_arch> class Number;
```

```
   template<class boolean_arch> class Mask;
```

Class `Number` is an abstraction of a single floating point number in a scalar platform (`scalar_arch==float`) or of a 4/8/16-wide vector register in SSE/AVX/Xeon Phi platforms (`scalar_arch:=__mm128|__mm256|__mm512`). Similarly, class `Mask` is an abstraction of the result of a comparison operation, in a form that can be used to perform a conditional assignment; thus `Mask<bool>` encapsulates a single C++ boolean variable, `Mask<__mm256>` captures a 256-bit mask usable in AVX `BLEND` instructions, while `Mask<__mmask16>` encapsulates the special concept in Intel Xeon Phi of a 16-bit *mask register* that is used in comparisons and conditional assignments. We provide enough overloaded operators in the interface of these classes to allow them to be used in algebraic expressions regardless of the encapsulating vector width. Ultimately we use them to construct macroscopic kernels of the form:

```
   template<class scalar_arch,class T_DATA>
void Add_Force_Differential(
    const T_DATA (&dx)[3][8], ...
    const T_DATA (&V)[9],
    const T_DATA (&dPdF)[12],
    T_DATA (&df)[3][8]);
```

These kernels are broken out in Figures 6.6, 6.7, and 6.8.

In this paradigm, we have separated the programmatic data width (type `T_DATA`, which could be `float`, for scalar code that computes forces on individual cells, or `float[8]`, for the force computation of all 8 cells of a block at once) from the architectural vector width. This allows us to design all of these kernels with the same semantics that would be followed

for scalar execution, and automatically generate code that works on geometric blocks of any size, and vector architectures of different vector widths. For example the function call `Add_Force_Differential<__m128,float[16]>(....)` would use SSE instructions to compute force differentials of geometric blocks containing 16 cells each (e.g. blocks shaped like $4 \times 2 \times 2$ grid cells).

Performance Results

This particular strategy for performance optimization for lattice deformers was tested in a prototype surgical simulation tool (For more details, see Chapter 8 on its deployment). In addition to the surgical models simulated with this tool, we also benchmarked our system with a high resolution human body model with anisotropic active musculature. Detailed timings, including time taken at the individual kernels of our solver, can be seen in Table 6.1.

Figure 6.6: Kernel Components for Update Position Based State

Diagram demonstrating the sub-kernel components for the Update Position Based State kernel. This kernel is executed once per Newton step and computes the deformation gradient $\mathbf{F}$, the Singular Value Decomposition of $\mathbf{F}$, and the stress tensor $\mathcal{T}$, as of Teran et al. [2005b], of the current configuration. These values can be reused in subsequent force differential computations as they do not change between Newton steps.

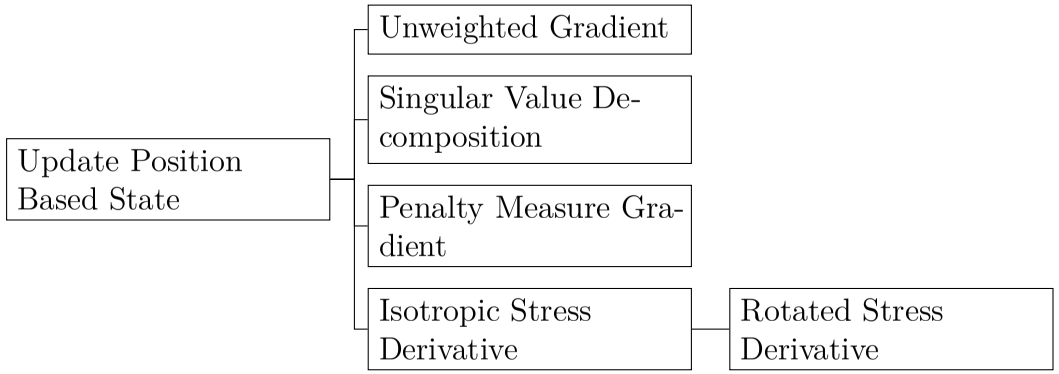


Figure 6.7: Kernel Components for Add Force

Diagram demonstrating the sub-kernel components for the Add Force kernel. This kernel produces nodal forces due to a single cell's nodal deformation. It is called once per Newton step as the right hand side of the update expression 3.19.

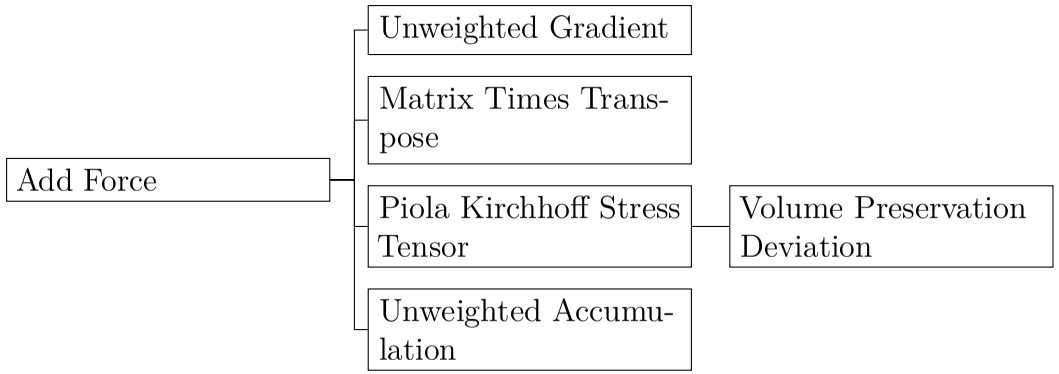
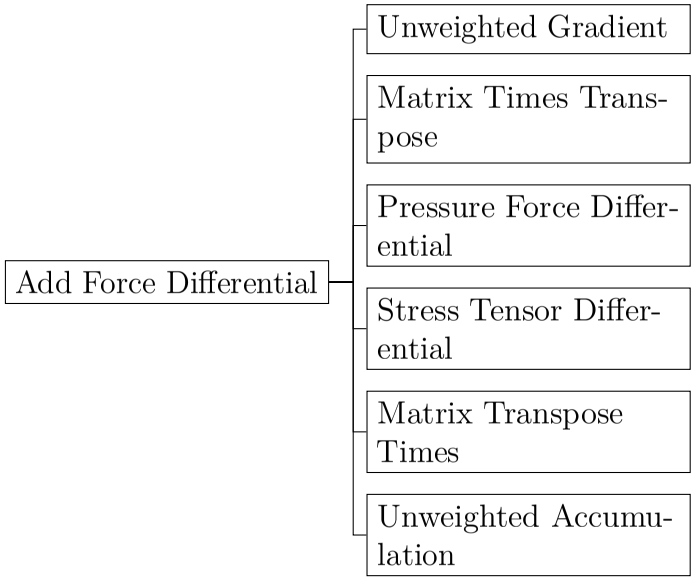


Figure 6.8: Kernel Components for Add Force Differential

Diagram demonstrating the sub-kernel components for the Add Force Differential kernel. This kernel is called repeatedly during each Newton step, acting as the effect of multiplying by the stiffness matrix $\mathbf{K}$. This kernel consumes the deformation gradient, SVD components, and the stress tensor $\mathcal{T}$ produced from the Update Position Based State kernel.



Example	**Plat.**	**Total Voxels**	**Grid Voxels**	**Mesh Voxels**	**Blocks**	**Newton Iteration** (s)	**Update State** (ms)	**Add Force** (ms)	**Add Differ.** (ms)	**Compact** (ms)	**UnCompact** (ms)
Dufourmentel Mouly	Xeon	47689	47206	483	8821	0.3630	1.5	1.4	1.6	0.4	0.6
		294248	294015	233	47681	2.0648	7.8	5.1	4.2	2.1	3.0
	Phi	47689	47206	483	8821	0.4259	1.3	1.5	0.9	0.1	2.0
		294248	294015	233	47681	0.7349	2.8	3.3	2.2	0.7	4.7
Rhomboid Flap	Xeon	48529	47203	1326	6909	0.1620	1.1	0.9	0.7	0.2	0.4
		879571	877362	2209	112302	2.4131	18.4	9.6	9.0	3.2	4.0
	Phi	48529	47203	1326	6909	0.3735	1.3	1.4	1.0	0.1	1.5
		879571	877362	2209	112302	1.1323	5.3	6.2	3.9	1.2	9.6
ZPlasty	Xeon	54003	49306	3697	6943	0.2293	1.2	0.5	0.9	0.3	0.5
		960810	957908	2902	127070	2.7707	20.5	9.3	10.1	3.6	4.5
	Phi	54003	49306	3697	6943	0.3814	1.3	1.4	1.1	0.1	1.5
		960810	957908	2902	127070	1.2401	5.9	6.8	4.3	1.3	10.8
Human	Xeon	2006903	2006903	0	265078	12.112	35.9	21.5	20.8	8.4	12.0
	Phi	2006903	2006903	0	265078	3.9608	11.4	16.3	7.8	3.3	16.2

Table 6.1: Performance results for surgical and animation examples.
Examples run on the Xeon platform used a 6-core Intel Xeon CPU E5-1650 machine with 64 GB of memory, while the Phi examples ran on a 60-core Xeon Phi 5110P card with 8GB of memory. All surgical examples were run with 50 conjugate gradient iterations while the human example was run with 100 iterations.

7 MACROBLOCK TECHNIQUE FOR HYBRID SOLVERS

In this chapter, we will explore an alternative approach for solving elastic deformation problems. Compared to Chapter 6, the approach outlined in this part of the document will be focusing on the challenge of poor convergence when using complex, non-linear materials. We will also be focusing on how careful attention to the interplay between modern compute ability verses memory availability can help us create more optimized solvers for elastic materials. To do this, we will be targeting the common approach used by many non-linear elastic simulations, namely the combination of an outer Newton method around an inner linear solver.

The Newton method has largely been the golden standard for the simulation of nonlinear elastic bodies, although a number of interesting deviations from this standard approach have garnered attention in the graphics literature (e.g., nonlinear multigrid cycles [Zhu et al., 2010], projective and position-based dynamics [Müller et al., 2007, Bouaziz et al., 2014, Wang, 2015] and shape matching [Rivers and James, 2007]). In a typical Newton scheme, once a linear approximation to the governing equations is computed, most practitioners will either employ a direct method or select a technique from a spectrum of iterative methods in order to solve the resulting system.

Direct solvers are perhaps the safest and most straightforward way to solve the system that results from the linearization of the governing equations. These methods can be quite practical for relatively small problems when direct algebra is not very expensive. Additionally, these techniques are quite resilient to the conditioning of the underlying problem. Even for large models, high quality parallel implementations such as the Intel MKL PARDISO library are available. Despite such advantages, direct methods suffer from inherently superlinear computational complexity. Even with the benefit of parallelism, direct methods will typically be more expensive than several iterative schemes, especially if few number of iterations are performed. Additionally, direct methods are inherently memory bound; at the core of direct

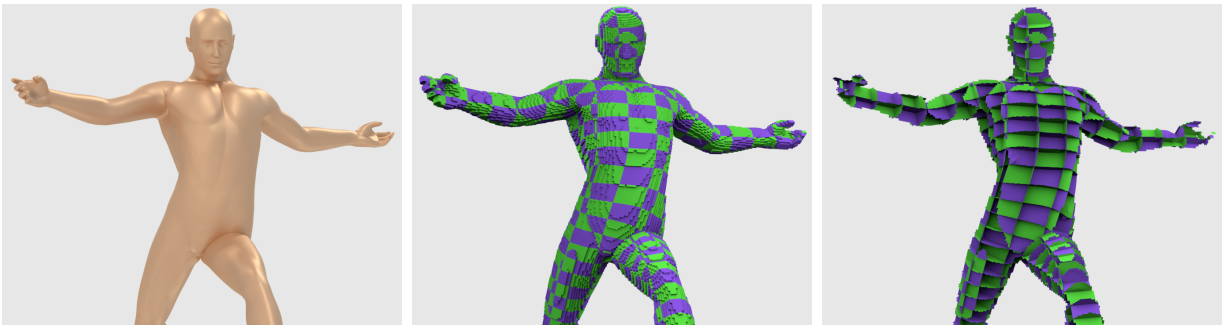


Figure 7.1: Deformed model alongside illustration of its constitutive macroblocks (Left) High resolution human mesh posed quasistatically by a skeleton with soft spring constraints. (Center) Embedding lattice divided into macroblocks (shown as alternating regions of green and purple). (Right) Illustration of the degrees of freedom along the macroblock boundaries. Conjugate Gradients is applied to a system with the size of these interface nodes. The model has 286K grid cells.

solvers are forward and backward substitution routines that carry out a very small number of arithmetic operations for each memory access required. This often results in grossly memory-bound execution profiles on modern hardware. This drawback is even more heavily felt for large models that do not fit in cache. Finally, each iteration of the Newton method is inherently inexact, providing only a step towards the converged solution. With direct methods we often find ourselves perfectly solving an inaccurate linearized approximation.

With iterative solvers, we can aim for an approximate solution to the linearized problem with the understanding that with each Newton iteration the problem itself will change. These methods include Krylov methods like Conjugate Gradient, Multigrid, and fixed-point iterations such as Jacobi, Gauss-Seidel and SOR. The primary benefit of iterative techniques is that each individual iteration is relatively cheap; this allows users the option to either iterate as much as they can afford, or alternatively truncate the iterative process when the approximate solution is acceptable. Also, many iterative methods are assembly-free, alleviating the need to construct or store the stiffness matrix. In fact, some of the most efficient techniques go to great lengths to minimize memory footprint [McAdams et al., 2011] while leveraging SIMD and multithreading.

Iterative solvers often have to cope with challenges of their own. Local methods like Jacobi,

GS, and SOR are slow to capture global effects, as they propagate information at a limited speed across the mesh. Krylov methods will typically prioritize the most important modes that contribute to a high residual; for example, consider a system with a few tangled elements that create large local forces. Elements suffering from small errors will be relatively neglected by a method like Conjugate Gradients, while the solver focuses on the highly tangled elements before turning its attention to the bigger picture. Multigrid is an interesting alternative that often emerges as the performance champion; however, it can often be tricky to get to work robustly, and might be less appropriate for thin elastic objects, such as a thin flesh layer on a simulated face. Preconditioning can accelerate the convergence of iterative solvers but, in contrast to certain fluids simulation scenarios, the accelerated convergence might not always justify the increased per-iteration cost. Preconditioners based on incomplete factorizations are memory bound as they require matrix assembly, and generally require an expensive re-factorization step at each Newton iteration. We note that the same factorization overhead would be incurred even when the Newton method is nearly converged, where just a handful of iterations would suffice to solve the linearized equations. Multigrid-based preconditioners might achieve more competitive performance, but such approaches have been primarily tested in the area of fluid simulation [Ferstl et al., 2014] and not so much in nonlinear deformable solids.

We propose a hybrid method that balances certain advantages of both direct and iterative schemes. Specifically we endeavor to achieve a good compromise between memory and compute load, reduce the memory footprint whenever possible, while significantly reducing iteration count. We pursue these goals while being competitive with the per-iteration cost of unpreconditioned CG. We employ a grid-based discretization, and aggregate rectangular clusters of cells into "macroblocks" with a proposed size of $16 \times 8 \times 8$ cells. These clusters essentially act as composite elements the same way that a typical hexahedral element can be thought of as a black box that takes displacements as inputs and produces nodal forces as output. However, our composite elements only take in displacements on the nodes of their

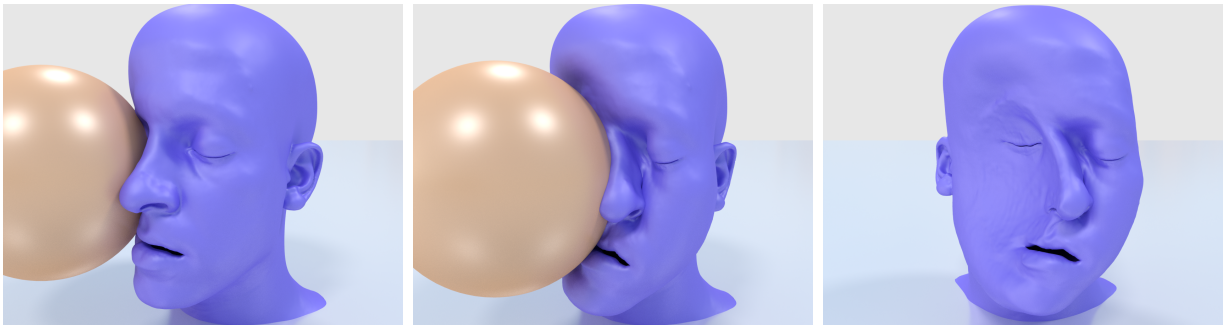


Figure 7.2: Macroblock solver used to handle rigid-elastic collision scenario
A kinematic rigid sphere collides against a high-resolution embedded face model. The relatively small thickness of the elastic flesh, in addition to the topological features near the nose and mouth regions, would complicate the use of a typical multigrid solver [McAdams et al., 2011] .

periphery and return forces on those same boundary nodes. Using this construct we obtain an equivalent linear system with degrees of freedom only on cluster boundaries.

Scope This chapter is an exploration of the performance potential offered by composite "macroblock" elements, initially focusing on the well-established simulation paradigm of a Newton-type scheme for solving a nonlinear system of governing equations. Thus, we only focus on grid-based discretizations of elasticity, and forgo the exploration of different simulation paradigms (e.g., multigrid, projective dynamics) where our formulation might still have a viable role. Finally, we consciously restrict our investigation to grid-based models that do not exhibit non-local interactions, such as spring-based constraints or penalty-based self-collision resolution mechanisms (one-sided collisions between the elastic body and kinematic objects are supported).

7.1 Macroblock-based discretization and numerical solution

We start by reviewing the equations that our method targets, and detailing how our proposed *macroblock* concept can reformulate them into an equivalent but more efficiently solvable

form. This process will necessitate the exact solution of several smaller systems of equations, each in the order of a couple thousand of unknowns. In this section we will simply assume that a highly efficient *direct* solver for those systems is available. Section 7.2 will provide the implementation details of this highly optimized solver.

The governing equations describing the deformation of an elastic nonlinear solid depend on the time integration scheme employed. For example, in quasistatic simulation we have to solve the nonlinear equilibrium equation $\mathbf{f}(\mathbf{x};\mathbf{t}) = 0$ at any time instance $\mathbf{t}$. Using an initial guess $\mathbf{x}_n$ of the solution, Newton's method computes a correction $\delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n$ by solving the linearized system:

$$\underbrace{\left(-\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\bigg|_{\mathbf{x}_n} \right)}_{\mathbf{K}(\mathbf{x}_n)} \delta\mathbf{x} = \mathbf{f}(\mathbf{x}_n) \tag{7.1}$$

If an implicit Backward Euler scheme was used, a system with similar structure would form the core of Newton's method [Sifakis and Barbic, 2012]:

$$\left[(1+\frac{\gamma}{\Delta t})\mathbf{K}(\mathbf{x}_n) + \frac{1}{\Delta t^2}\mathbf{M} \right] \delta\mathbf{x} = \frac{1}{\Delta t}\mathbf{M}(\mathbf{v}_p - \mathbf{v}_n) + \mathbf{f}(\mathbf{x}_n, \mathbf{v}_n) \tag{7.2}$$

where $\mathbf{M}$ is the mass matrix, γ is the Rayleigh coefficient, $\mathbf{v}_p$ the velocities at the previous time step, and $\mathbf{f}$ now includes both elastic and damping forces (see [Sifakis and Barbic, 2012] for further details).

Despite the semantic differences, the linear systems in equations (7.1) and (7.2) are very similar from an algebraic standpoint:

- Their coefficient matrices are both symmetric positive definite.
- Their coefficient matrices have the same sparsity pattern.
- In a grid-based discretization, their coefficient matrices can be assembled from the contributions of individual grid cells.

We note that in order for this last property to hold true, we have assumed that our

elastic model does not have any interactions between remote parts of its domain, such as penalty forces used to enforce self-collision (which we consciously excluded from our scope). Incidentally, penalty forces used to enforce collisions with external kinematic bodies are allowed, since their point of application on the elastic body can be embedded in a single grid cell. For brevity, we will write any linear system that shares the three properties above using the simplified notation $\mathbf{Kx} = \mathbf{f}$, without individual emphasis on whether the system originated from a quasistatic, or a dynamic implicit scheme as in equations (7.1) and (7.2), respectively.

The crucial next step in our proposed approach is a partitioning of the active grid cells into *macroblocks*, which are grid-aligned rectangular clusters of a predetermined size, as illustrated in figure 7.1. In our implementation we use macroblocks with dimensions of $16 \times 8 \times 8$ grid cells, although the formulations in this section are largely independent of the macroblock size. Section 7.3 provides the reasoning behind the choice of this particular size of a macroblock.

Each macroblock $\mathcal{B}_\mathtt{i}$ consists of up to $16 \times 8 \times 8 = 1024$ grid cells $\mathsf{C}_{\mathtt{i}_1}, \mathsf{C}_{\mathtt{i}_2}, \ldots, \mathsf{C}_{\mathtt{i}_M}$; note that in some cases this maximum number of constituent cells will not be reached, if the macroblock overlaps with the boundary of the elastic object, or if "gaps" of empty grid cells are present within its extent. Similarly, up to $17 \times 9 \times 9$ nodal degrees of freedom will be present in the region spanned by $\mathcal{B}_\mathtt{i}$. Up to $15 \times 7 \times 7$ of them will be on the *interior* of $\mathcal{B}_\mathtt{i}$ and thus will not be touched by any other macroblock; we will denote this interior node set with $\mathrm{I}_\mathtt{i}$. The remaining nodes, located on the *boundary* of $\mathcal{B}_\mathtt{i}$ are potentially shared by neighboring macroblocks; we will call these *interface nodes* (as they reside at the interface between macroblocks) and denote their set with $\Gamma_\mathtt{i}$. All sets $\mathrm{I}_\mathtt{i}$ are clearly disjoint, and we will denote their union by $\mathrm{I} = \cup \mathrm{I}_\mathtt{i}$. The interface sets $\Gamma_\mathtt{i}$ do overlap with one another, and we denote their union by $\Gamma = \cup \Gamma_\mathtt{i}$. For large enough models, we expect around 72% of grid nodes to lie in some interior set, and approximately 28% on the interface set Γ, using the aforementioned macroblock size.

Our objective will be to replace the linear system $\mathbf{Kx} = \mathbf{f}$ with an equivalent system,

Figure 7.3: Macroblock solver used to handle basic quasistatic pose scenario Armadillo model deforming as a result of kinematically animated Dirichlet constraints. Embedding lattice shown on the right.

which only includes the interface nodes in Γ as unknowns. To do so, we first write the system in block form, by separating interior and interface variables as follows:

$$\begin{pmatrix} \mathbf{K}_{II} & \mathbf{K}_{I\Gamma} \\ \mathbf{K}_{\Gamma I} & \mathbf{K}_{\Gamma\Gamma} \end{pmatrix} \begin{pmatrix} \mathbf{x}_I \\ \mathbf{x}_\Gamma \end{pmatrix} = \begin{pmatrix} \mathbf{f}_I \\ \mathbf{f}_\Gamma \end{pmatrix}$$

Using block Gauss elimination, this system can be converted to the following equivalent block-triangular form:

$$\begin{pmatrix} \mathbf{K}_{II} & \mathbf{K}_{I\Gamma} \\ \mathbf{0} & \mathbf{K}_{\Gamma\Gamma} - \mathbf{K}_{\Gamma I}\mathbf{K}_{II}^{-1}\mathbf{K}_{I\Gamma} \end{pmatrix} \begin{pmatrix} \mathbf{x}_I \\ \mathbf{x}_\Gamma \end{pmatrix} = \begin{pmatrix} \mathbf{f}_I \\ \mathbf{f}_\Gamma - \mathbf{K}_{\Gamma I}\mathbf{K}_{II}^{-1}\mathbf{f}_I \end{pmatrix} \tag{7.3}$$

Equation (7.3) suggests the following algebraically equivalent method for solving the system $\mathbf{K}\mathbf{x} = \mathbf{f}$:

Step 1 Compute an interface-specific right hand side, from the bottom block of the right hand side of system (7.3):

$$\hat{\mathbf{f}}_\Gamma = \mathbf{f}_\Gamma - \mathbf{K}_{\Gamma I}\mathbf{K}_{II}^{-1}\mathbf{f}_I \tag{7.4}$$

Step 2 Solve the interface-specific system $\hat{\mathbf{K}}\mathbf{x}_\Gamma = \hat{\mathbf{f}}_\Gamma$ to compute the values $\mathbf{x}_\Gamma$ of all interface

nodes. Note that the matrix of the system

$$\hat{\mathbf{K}} = \mathbf{K}_{\Gamma\Gamma} - \mathbf{K}_{\Gamma\mathrm{I}}\mathbf{K}_{\mathrm{II}}^{-1}\mathbf{K}_{\mathrm{I}\Gamma} \tag{7.5}$$

is the Schur complement of the symmetric positive definite original matrix $\mathbf{K}$, hence it is symmetric and positive definite in its own right. We will solve this system, which only involves interface degrees of freedom, using Conjugate Gradients.

Step 3 Conclude the computation by solving for the interior nodal variables from the top block of system (7.3) as:

$$\mathbf{x}_{\mathrm{I}} = \mathbf{K}_{\mathrm{II}}^{-1}\left(\mathbf{f}_{\mathrm{I}} - \mathbf{K}_{\mathrm{I}\Gamma}\mathbf{x}_{\Gamma}\right) \tag{7.6}$$

In order to reproduce the exact solution of $\mathbf{Kx} = \mathbf{f}$, we would need to solve the interface problem $\hat{\mathbf{K}}\mathbf{x}_{\Gamma} = \hat{\mathbf{f}}_{\Gamma}$ in Step 2 exactly. However, given that we only use this solution as part of an iterative Newton update, there is nothing preventing us from stopping the Conjugate Gradients solver for the interface system short of full convergence. However, as we discuss in sections 7.4 and 7.4, the interface problem requires far fewer CG iterations to produce good quality results than the same Krylov method applied to $\mathbf{Kx} = \mathbf{f}$. Furthermore, the optimizations of the following section allow us to make the per-iteration cost of CG on the interface problem be comparable to each CG iteration on the original problem, resulting in a significant net performance gain. When assessing the cost of Steps 1-3, it is important to observe the following:

Inversion of $\mathbf{K}_{\mathrm{II}}$ is the main performance challenge. The most performance-sensitive component of this process is the multiplication with the inverse $\mathbf{K}_{\mathrm{II}}^{-1}$ of the matrix block corresponding to variables interior to macroblocks. Nevertheless, since there is no direct coupling (in $\mathbf{K}$) between interior variables of neighboring macroblocks, $\mathbf{K}_{\mathrm{II}}$ is a block diagonal matrix, comprised of decoupled diagonal components for each set of interior variables of each macroblock. We thus use multithreading to invert the interior of each macroblock in a parallel and independent fashion. Within each macroblock, we use the aggressively SIMD-optimized

direct solver detailed in section 7.2 to perform the inversion exactly and efficiently.

Multiplication with $\mathbf{K}_{I\Gamma}, \mathbf{K}_{\Gamma I}$ in Steps 1 & 3 is inexpensive. The off-diagonal blocks $\mathbf{K}_{I\Gamma}$ and $\mathbf{K}_{\Gamma I}$ appearing in Steps 1 and 3 are small and sparse sub-blocks of $\mathbf{K}$. In addition, they are only used in two matrix-vector multiplications across Steps 1 and 3 for an entire Newton iteration (we will address their role in Step 2, next). These matrices can be efficiently stored in sparse format, and their multiplication with vectors can be parallelized (in our implementation, via SIMD within macroblocks and multithreading across blocks). These matrices have minimal performance impact in our examples.

Conjugate Gradients does not need to construct $\hat{\mathbf{K}}$. The interface matrix $\hat{\mathbf{K}}$, being a Schur complement, is significantly denser than the original matrix $\mathbf{K}$; for example, any two nodal variables on the interface of the same macroblock would be coupled together. Fortunately, the Conjugate Gradients method does not need this matrix to be explicitly constructed. Instead, the only requirement is to be able to compute matrix-vector products of the form

$$\mathbf{s}_\Gamma = \hat{\mathbf{K}}\mathbf{p}_\Gamma = \left(\mathbf{K}_{\Gamma\Gamma} - \mathbf{K}_{\Gamma I}\mathbf{K}_{II}^{-1}\mathbf{K}_{I\Gamma}\right)\mathbf{p}_\Gamma$$

for any given input vector $\mathbf{p}_\Gamma$. In fact, we can compute such products on a per-macroblock basis. We start by computing the restriction of $\mathbf{p}_\Gamma$ to the boundary Γ_i of each macroblock $\mathcal{B}_i$, which we denote by $\mathbf{p}_{\Gamma_i}$. Subsequently, we compute a partial contribution to the matrix-vector product as

$$\mathbf{s}_{\Gamma_i} = \hat{\mathbf{K}}_i\mathbf{p}_{\Gamma_i} = \left(\mathbf{K}_{\Gamma_i\Gamma_i} - \mathbf{K}_{\Gamma_i I_i}\mathbf{K}_{I_i I_i}^{-1}\mathbf{K}_{I_i\Gamma_i}\right)\mathbf{p}_{\Gamma_i} \tag{7.7}$$

The highly efficient evaluation of the expression in equation (7.7) is precisely the focus of section 7.2. We compute the contributions of all macroblocks $\mathbf{s}_{\Gamma_i}$ in parallel, via multithreading, and reduce them all together in a final summation to produce the global result $\mathbf{s}_\Gamma$.

Finally, we point out a significant intuition behind the nature of the macroblock-local Schur complement $\hat{\mathbf{K}}_i$, defined via equation (7.7). Similar to how an elemental stiffness matrix maps nodal displacements to nodal force differentials for a tetrahedral or hexahedral element,

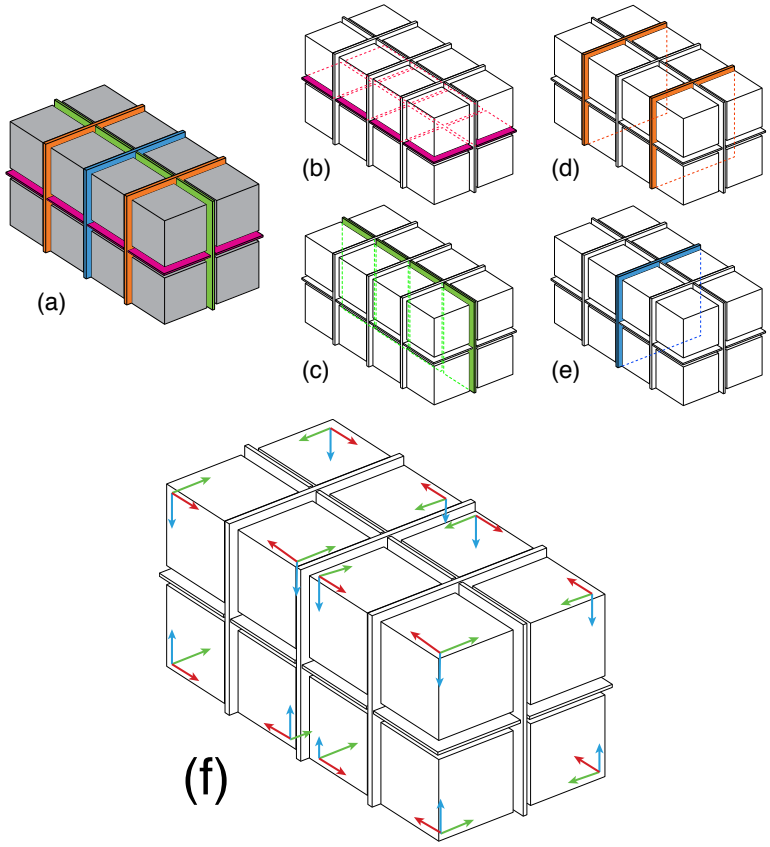


Figure 7.4: Illustration of the internal macroblock divisions and structure
The $15\times7\times7$ macroblock interior nodes are hierarchically subdivided, yielding (a) sixteen $3\times3\times3$ "subdomains" and (b,c,d,e) four "interface" layers. The first subdomain is reordered to maximize sparsity, and this ordering is mirrored (f) to the other 15 subdomains.

the *macroblock stiffness matrix* $\hat{\mathbf{K}}_\mathrm{i}$ directly maps displacements on the boundary to forces on the same boundary nodes, under the assumption that all interior nodes are functionally constrained to their exact solution subject to the boundary displacement values. We note the similarity of this concept to the work of Gao et al. [2014], although they used a Schur complement to abstract away the interior nodes of an entire model, rather than assembling an elastic solid from macroscopic cell blocks.

7.2 An optimized direct solver for macroblocks

As outlined in section 7.1, inverting $\mathbf{K}_{\mathrm{I_i I_i}}$ within each macroblock is the most performance-sensitive part of our numerical approach. In this section we explain how this operation can be performed with high efficiency, by reducing its memory footprint and aggressively leveraging instruction-level (SIMD) parallelism. We have designed a numerical data structure containing the appropriate metadata and computational routines to compute the matrix-vector product $\mathbf{s}_{\Gamma_\mathrm{i}}$ of equation (7.7), given the boundary values $\mathbf{p}_{\Gamma_\mathrm{i}}$ as input. This structure stores matrices $\mathbf{K}_{\Gamma_\mathrm{i}\Gamma_\mathrm{i}}$, $\mathbf{K}_{\Gamma_\mathrm{i}\mathrm{I_i}}$ and $\mathbf{K}_{\mathrm{I_i}\Gamma_\mathrm{i}}$ explicitly in compressed sparse format (with slight modifications to facilitate SIMD parallelism, as explained in section 7.2), as those are relatively compact and inexpensive to multiply with. In addition, we store just enough information to be able to multiply the interior inverse $\mathbf{K}^{-1}_{\mathrm{I_i I_i}}$ with input vectors, without storing this matrix explicitly. As this section focuses on a single macroblock $\mathcal{B}_\mathrm{i}$, we omit the macroblock index i, using the symbols I and Γ to denote its interior and interface nodes.

Given the sparsity and definiteness of $\mathbf{K}_{\mathrm{II}}$, one straightforward approach would be to compute its (exact) Cholesky factorization, under a sparsity optimizing variable reordering. This factorization would take place once per Newton iteration, while forward and backward substitution passes would be used to apply the inverse in every subsequent CG iteration based on equation (7.7). We do, in fact, compute exactly such a reordered Cholesky factorization; however, instead of forward/backward substitution, we leverage a hierarchical alternative (derived from the coefficients of the computed factorization) that achieves the same result in significantly less time, by reducing the required memory footprint.

Reordering

We utilize a custom reordering of the $15 \times 7 \times 7$ interior nodes of the macroblock, in order to optimize the sparsity of Cholesky factorization and expose repetitive regular patterns that can be matched with SIMD calculations. We define this reordering by means of a hierarchical

subdivision, as illustrated in figure 7.4. First, we subdivide the $15 \times 7 \times 7$ interior region into two $7 \times 7 \times 7$ subregions, separated by a $1 \times 7 \times 7$ interface layer, illustrated in blue color in figure 7.4(e). Each of these two regions is further subdivided into two $3 \times 7 \times 7$ parts, separated by $1 \times 7 \times 7$ interface layers, shown in orange in figure 7.4(d). Those $3 \times 7 \times 7$ regions are then split into two $3 \times 3 \times 7$ parts, separated by $3 \times 1 \times 7$ interfaces, shown in green in figure 7.4(c). A last subdivision results in two $3 \times 3 \times 3$ subdomains, on either side of a $3 \times 3 \times 1$ connector, drawn in magenta in figure 7.4(b). We refer to the resulting $3 \times 3 \times 3$ blocks as *subdomains*, and the connective regions in figures 7.4(b) through 7.4(e) as Level-1 through Level-4 *interfaces*. We then proceed to compute a minimum-degree reordering for one of the 16 resulting $3 \times 3 \times 3$ subdomains, and *mirror* this reordering across their hierarchical interfaces to enumerate the nodes of all remaining subdomains. This mirroring is essential in creating repetitive patterns in the Cholesky factors, on which SIMD optimizations are crucially dependent. The final overall reordering is formed by assembling a tree of this hierarchical subdivision (with interfaces on parent nodes, and the regions they separate as their children), and computing a reverse breadth-first tree traversal.

We have found this reordering to be optimal; it matches or outperforms any heuristics (e.g., minimum-degree reordering in Matlab) in the sparsity of the Cholesky factors. The resulting sparsity pattern is illustrated in figure 7.5. Matrix entries colored red are a subset (but not all) of the entries that were filled-in during the Cholesky process. As expected, forward and backward substitution on this matrix is a pronouncedly memory-bound operation; hence we propose a further algorithmic modification that produces the same result with approximately one-seventh of the memory footprint. This alternative approach will only need to store the number of coefficients corresponding to the *black-colored* entries in figure 7.5. The metadata for this alternative approach, detailed next, will be harvested from the Cholesky factorization just computed.

Hierarchical factorization

Consider the first hierarchical subdivision, illustrated in 7.4(e), which separated the $15 \times 7 \times 7$ block of interior nodes into two $7 \times 7 \times 7$ subregions, which we denote by I_1 and I_2, along with a $7 \times 7 \times 1$ connective region, denoted $\mathrm{I}_\mathbf{c}$ (drawn blue in the figure above). If we reorder the matrix $\mathbf{K}_{\mathrm{II}}$ to expose this partitioning, it assumes the following block form:

$$\begin{pmatrix} \mathbf{K}_{11} & & \mathbf{K}_{1\mathbf{c}} \\ & \mathbf{K}_{22} & \mathbf{K}_{2\mathbf{c}} \\ \mathbf{K}_{\mathbf{c}1} & \mathbf{K}_{\mathbf{c}2} & \mathbf{K}_{\mathbf{cc}} \end{pmatrix}$$

It can be easily verified that the *inverse* of this matrix can be written in the following Block-LDL form:

$$\begin{pmatrix} \mathbf{I} & & -\mathbf{K}_{11}^{-1}\mathbf{K}_{1\mathbf{c}} \\ & \mathbf{I} & -\mathbf{K}_{22}^{-1}\mathbf{K}_{2\mathbf{c}} \\ & & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{K}_{11}^{-1} & & \\ & \mathbf{K}_{22}^{-1} & \\ & & \mathbf{C}^{-1} \end{pmatrix} \begin{pmatrix} \mathbf{I} & & \\ & \mathbf{I} & \\ -\mathbf{K}_{\mathbf{c}1}\mathbf{K}_{11}^{-1} & -\mathbf{K}_{\mathbf{c}2}\mathbf{K}_{22}^{-1} & \mathbf{I} \end{pmatrix}$$

where $\mathbf{C} = \mathbf{K}_{\mathbf{cc}} - \mathbf{K}_{\mathbf{c}1}\mathbf{K}_{11}^{-1}\mathbf{K}_{1\mathbf{c}} - \mathbf{K}_{\mathbf{c}2}\mathbf{K}_{22}^{-1}\mathbf{K}_{2\mathbf{c}}$ is the Schur complement of $\mathbf{K}_{\mathbf{cc}}$. With this formulation, solving a problem $\mathbf{K}_{\mathrm{II}}\mathbf{x}_{\mathrm{I}} = \mathbf{f}_{\mathrm{I}}$ is equivalent to multiplying with the factorized version of $\mathbf{K}_{\mathrm{II}}^{-1}$ in the equation above. We make the following significant observations:

- Other than the (seemingly elusive) inverses $\mathbf{K}_{11}^{-1}, \mathbf{K}_{22}^{-1}$ and $\mathbf{C}^{-1}$, the factorization above does not incur any fill-in; factors such as $\mathbf{K}_{1\mathbf{c}}$, etc. have the original sparsity found in sub-blocks of $\mathbf{K}_{\mathrm{II}}$.

- We can prove that the lower-triangular Cholesky factor of the Schur complement $\mathbf{C}$ is *exactly* the bottom-rightmost (dense) diagonal block of the matrix shown in figure 7.5 (also more prominently colored blue in figure 7.6). Thus, multiplication with $\mathbf{C}^{-1}$ can be performed simply via forward and backward substitution.

- The inverses of the two subregions, $\mathbf{K}_{11}^{-1}$ and $\mathbf{K}_{22}^{-1}$ can be applied recursively using the

exact same decomposition and block-LDL factorization described here, by splitting each $7 \times 7 \times 7$ into two $7 \times 7 \times 3$ subregions and a $7 \times 7 \times 1$ connector as before. This recurrence can be unfolded until we arrive at the (sixteen) $3 \times 3 \times 3$ subdomains shown in figure 7.4. The Cholesky factors of those sixteen blocks are exactly the top-sixteen (sparse) diagonal blocks on the top-left of the Cholesky factorization in figure 7.5; thus those submatrices can be readily inverted without recursion.

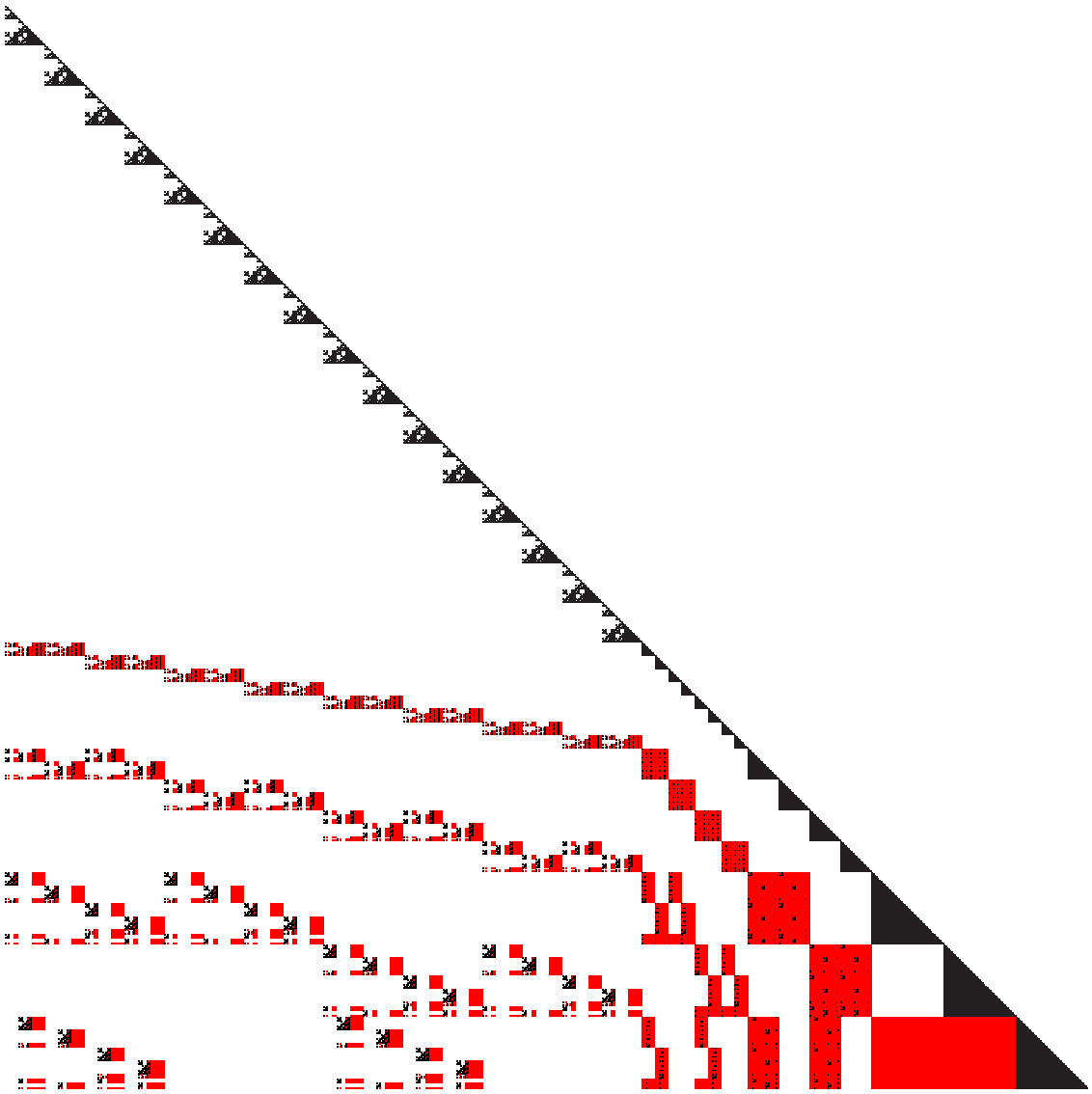


Figure 7.5: Illustration of macroblock sparsity patterns
Sparsity of Cholesky factorization (with our optimal reordering), shown with red **and** black colors. The memory footprint of our proposed solver only includes the black-colored coefficients.

We note that the Cholesky factors of the Schur complement matrices (**C**) that appear in deeper levels of this hierarchical solution scheme are similarly harvested from the (dense) diagonal blocks of the overall Cholesky factorization (highlighted in purple, green and orange color in figure 7.6, immediately above the blue block at the bottom-rightmost part which corresponds to the first hierarchical subdivision). At the final level of this hierarchical solution process, we need the inverses of the matrix blocks corresponding to the sixteen $3 \times 3 \times 3$ subdomains themselves. For those blocks, we employ directly their sparse Cholesky factorization, as seen in the top-sixteen (dark blue colored) diagonal blocks in figure 7.6, and solve using standard forward and backward substitution.

It would appear that the additional computation that this recursive solution entails would render it prohibitively expensive. However, the stock Cholesky forward and backward substitution are memory-bound by such a wide margin that our optimized recursive solution can afford to execute a significantly larger amount of arithmetic operations, while still being (barely, this time) bound by the time required to stream the requisite matrix coefficients from memory into cache. The not so obvious, but very significant, benefit is that the entire working set of this solver is less than 800KB per macroblock, allowing all subsequent memory accesses to occur exclusively in cache for every CPU core handling an individual macroblock. Note that, although the original reordered Cholesky factorization produces additional fill-in on the matrix entries colored red in figure 7.5, our recursive substitution process only touches a significantly sparser subset of entries (colored black), requiring about 27% of the entries and 15% of the storage footprint of the full, filled-in Cholesky (accounting for row/column indices of structurally sparse blocks). In section 7.4 we provide the effective memory bandwidth achieved by our macroblock solver, averaging between 13-18GB/s on a 10-core Haswell-EP Xeon processor.

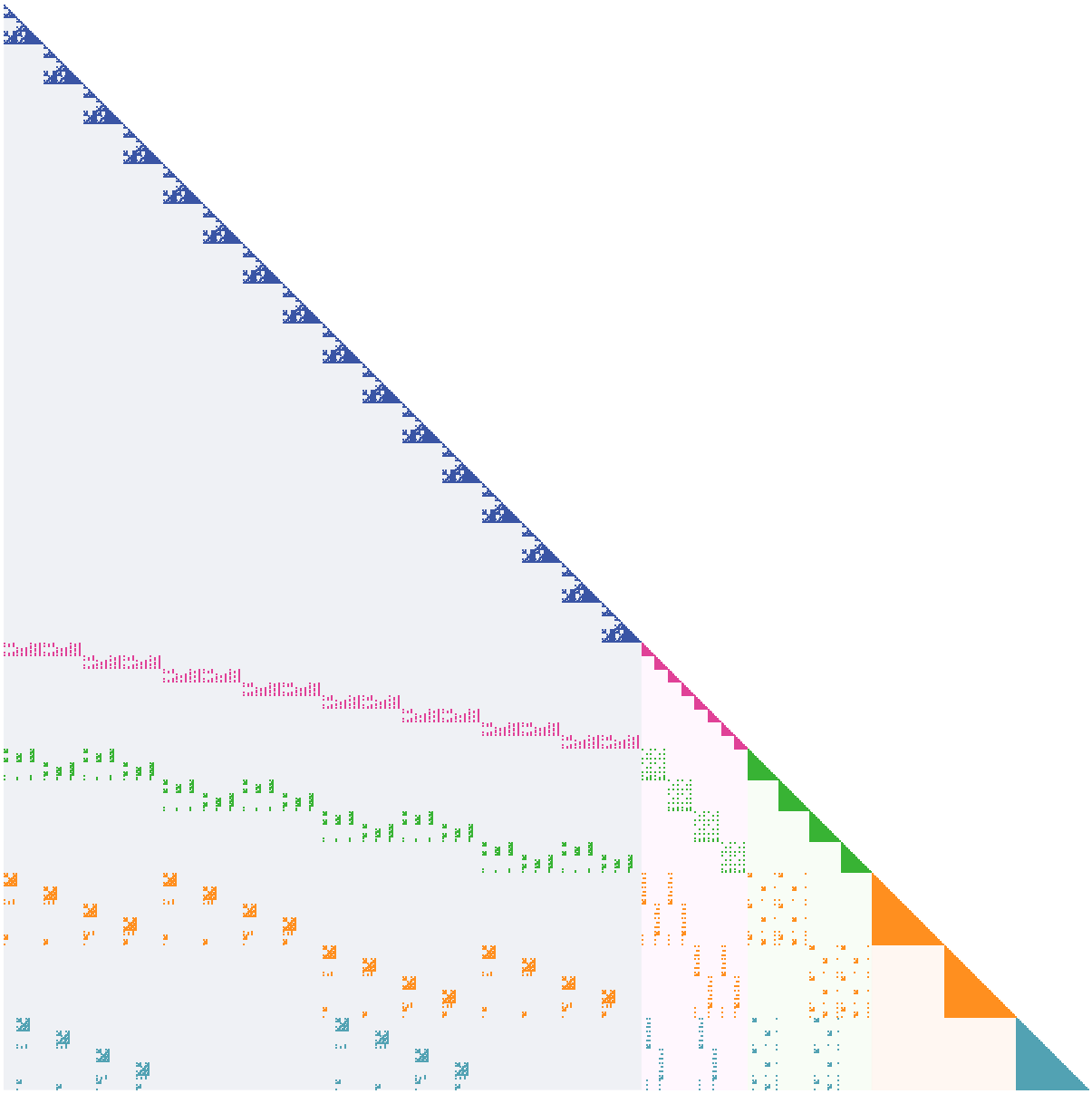


Figure 7.6: Illustration of SIMD-instruction groupings of a macroblock matrix
Our method reveals regular structures in the matrix sparsity pattern, exploiting them for vectorization. Same-color entries in the off-diagonal blocks can be processed with SIMD instructions.

Vectorization

The sparse matrix data used in our method, as seen in figure 7.6, is characterized by extensive regular and repetitive sparsity patterns that can facilitate computation using SIMD instructions. We have used color coding to indicate data used within a level of our hierarchical solution scheme, and to highlight such patterns of regularity. Those include the sixteen

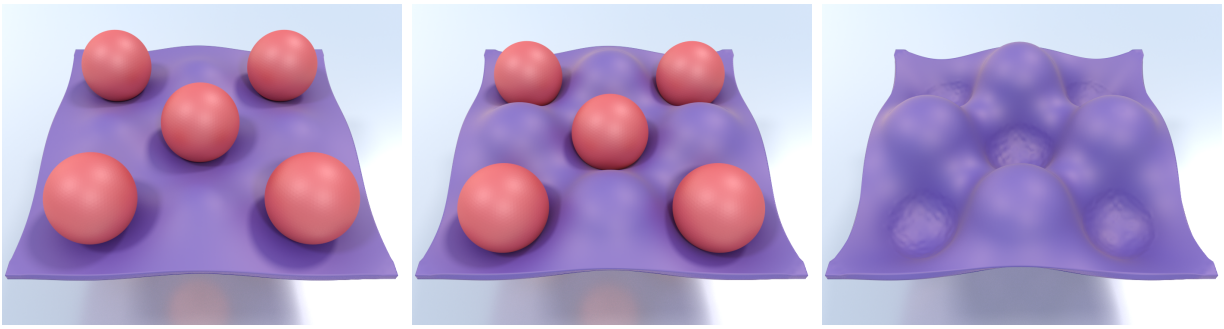


Figure 7.7: Macroblock collision scenario unsuitable for multigrid techniques
An array of 9 kinematic spheres, arranged in an alternating pattern across a thin volumetric sheet, are pressed against it. The limited thickness of this model would hinder applicability of stock geometric multigrid, in the absence of nonstandard coarsening strategies.

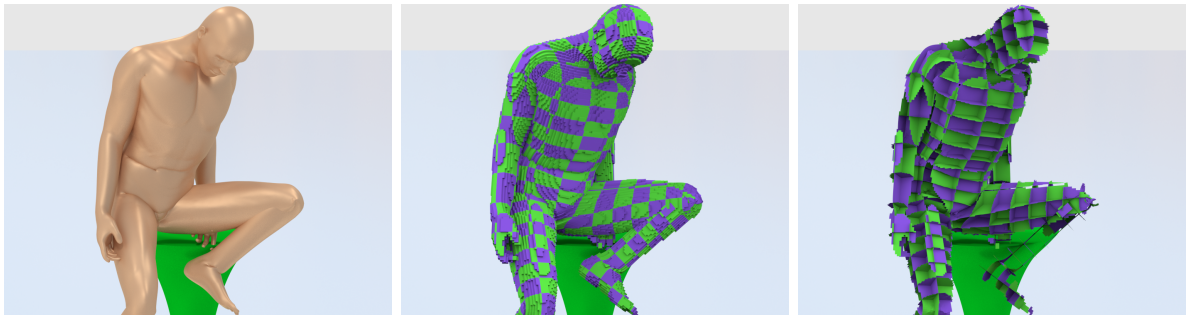


Figure 7.8: Skinning simulation example with spring-attached bones
An additional demonstration of a skinning simulation, driven by kinematic bones attached to the flesh via spring constraints.

sparse Cholesky factors corresponding to the interiors of the $3 \times 3 \times 3$ subdomains (colored as dark blue blocks, along the top-leftmost part of the matrix diagonal), the dense Cholesky factors of Schur complements at deeper levels (eight magenta, four green, two orange, and one cyan dense block, spanning the rest of the block-diagonal region of the matrix), and sparse submatrices on the block lower-triangular part of the matrix, corresponding to entries of the original stiffness matrix that touch an interface layer at a given level of the hierarchy and nodes on the two subregions that the interface layer separates.

Opportunities for aggressive vectorization directly emerge from such data regularities. For example, sparse forward and backward substitution on all sixteen $3 \times 3 \times 3$ subdomains can be done in tandem, with 16-way SIMD parallelism (e.g., using two 8-wide AVX instructions).

Repetitive sparsity patterns in the lower-triangular part of the matrix of figure 7.6 are used in vectorized matrix-vector multiplication operations. The *dense* nature of the blocks along the lower part of the block-diagonal allows fine-grain vectorization via standard practices. Furthermore, even matrix operations that connect the $15 \times 7 \times 7$ interior node set with the *boundary* of the macroblock, as the multiplication with matrices $\mathbf{K}_{\Gamma_i \Gamma_i}$, $\mathbf{K}_{\Gamma_i I_i}$ and $\mathbf{K}_{I_i \Gamma_i}$ defined in the beginning of this section, can be vectorized by splitting up such matrices in parts that correspond to the sixteen $3 \times 3 \times 3$ macroblocks at the interior of the macroblock boundary. Ultimately, about 96% of the requisite computations can accommodate 16-wide SIMD parallelism, and the majority of the remaining operations offer at least 8-wide SIMD parallelism potential. We have extensively leveraged these vectorization opportunities in our optimized implementation based on AVX compiler intrinsics.

7.3 Justification of macroblock size choice

Our choice for utilizing macroblocks of dimension $16 \times 8 \times 8$ was motivated by a number of factors. First, we wanted to provide the opportunity for at least 16-way SIMD-based parallelism, which is a future-safe choice given the upcoming availability of CPUs with the AVX-512 instruction set. The working set size associated with macroblocks of that size is conveniently approximately 800KB, which allows the entire macroblock solver to fit entirely in cache, even if all cores of a typical modern Xeon processor are processing independent macroblocks, in parallel. Using an even larger macroblock size would allow the dimensionality of the interface to be further reduced, but the increment in the working set would be disproportionately large, due to the size of the next-level interface (would be $15 \times 1 \times 7$) which would, at that point, yield an unattractively large dense Schur complement matrix for that interface level.

7.4 Examples and performance evaluation

We visually demonstrate the applicability of our solver to a number of simulation scenarios including constraint-driven deformations, skinning animations and elastic models colliding with kinematic rigid objects. We used a hexahedral finite element discretization of corotated linear elasticity, with the standard adjustments for robust simulation in the presence of inverted elements [Irving et al., 2004]. Given that our method uses a direct solver at the macroblock level, we opted to integrate the strain energy using the eight Gauss quadrature points for each hexahedron, as opposed to the one-point quadrature scheme that is often used [McAdams et al., 2011, Patterson et al., 2012]. This more accurate quadrature scheme does not require explicit stabilization, and adds no extra algorithmic effort in our solver other than a modest increase in the matrix construction cost.

In figure 7.3, we demonstrate an armadillo model being deformed as a result of specific lattice nodes animated as kinematic Dirichlet boundary conditions. In order to incorporate Dirichlet boundary conditions in the interior of a macroblock, we replace the equation associated with any such node with an explicit Dirichlet condition $\delta\mathbf{x}_\mathbf{i} = \mathbf{0}$ (the value can be set to zero without loss of generality, since equation (7.1) is solved for position corrections, which are zero for constraint nodes that have been already moved to their target locations). We restore symmetry of the overall matrix by zeroing out entries involving the Dirichlet node in the stencil of the elasticity operator of any neighboring node (again, a safe operation as the Dirichlet value is zero for the correction $\delta\mathbf{x}_\mathbf{i}$). Similarly, any nodes in a macroblock that are exterior to the simulated model are treated as zero-Dirichlet conditions, to maintain a constant matrix structure for all macroblocks.

In figures 7.2 and 7.7, we demonstrate the compatibility of our method with penalty-based collisions with kinematic objects. We use an implicit representation for the colliding bodies to enable fast detection of collision events between such bodies and embedded collision proxies on the surface of our model. When such an event occurs, a zero rest length penalty spring constraint is instantiated connecting the offending point on the embedded surface to the

nearest point on the surface of the collision object. Finally, figures 7.1 and 7.8 show two examples of a human character animated using embedded kinematic bone constraints. Skeletal motion data was drawn from the CMU motion capture database (http://mocap.cs.cmu.edu).

Performance benchmarks - Comparison to CG

Table 7.1 provides runtime details for individual solver components. The first two columns correspond to the models of figures 7.1 and 7.3, and have been processed with our proposed macroblock solver. In addition, we repeat the skinning simulation of figure 7.1 using this time a highly optimized and parallelized matrix-free implementation of unpreconditioned Conjugate Gradients, borrowed from the work of Mitchell et al. [2015]. While using this matrix-free CG solver, we consider two discretization alternatives: (a) a one-point quadrature scheme, with explicit stabilization [McAdams et al., 2011, Patterson et al., 2012], listed in the third column and (b) a more accurate 8-point quadrature scheme matching the one in our macroblock solver (fourth column). As mentioned, the quadrature scheme does not affect

Table 7.1: Performance results for the macroblock solver across several examples

Runtime details on a 10-core Xeon E5-2687W CPU. The benchmark in the first column is repeated in the last two columns using stock CG, with one and eight quadrature points respectively. Interface-Multiply is the multiplication with the Schur complement.

	Human	Armadillo	Human	Human
Solver	Macroblock	Macroblock	CG (1-QP)	CG (8-QP)
Active Cells	286K	24K	286K	286K
Macroblocks	642	95	*N/A*	*N/A*
Interface Multiply	27.6 ms (17 GB/s)	4.36 ms (16 GB/s)	*N/A*	*N/A*
CG Iteration	33.3 ms	5.22 ms	18.8 ms	88.3 ms
Factorization	291 ms	88.0 ms	*N/A*	*N/A*
Newton Iteration				
10 CG	791 ms	166 ms	269 ms	958 ms
20 CG	1.29 s	244 ms	462 ms	1.84 s
50 CG	2.79 s	479 ms	1.07s	4.47 s

the solve times of our method, once the matrix has been constructed; the construction cost is included in the Newton iteration runtimes, and was less than 10% of the overall runtime in all our experiments. We observe that, in spite of the up-front factorization cost that our method incurs, it typically stays within a factor of 2-3x of the cost of the single quadrature point CG scheme, for the same number of iterations. Further experiments have shown that the effect of as few as ten iterations of our macroblock scheme is commensurate with 5-10x more iterations of the stock CG method. Note that if the more accurate quadrature scheme is employed, our method outperforms the CG option even on a per-iteration basis.

Additional solver comparisons

We report some additional comparisons with other established numerical algorithms or software packages. All our comparisons are relative to the skinning example in the first column of Table 7.1.

Macroblock inversion via Cholesky/PARDISO As an alternative to our optimized macroblock solver of section 7.2, one could choose to directly compute *and* apply a stock Cholesky factorization per macroblock. We tested this using the PARDISO library, which yielded a factorization cost of 748ms (ours: 291ms) and a solve time of 93ms via forward/backward substitution (ours: 20.9ms; part of the Interface-Multiply cost). Solve time savings are due to our reduced memory demands. Faster factorization time is attributed to intrinsic knowledge about the constant sparsity pattern of each block, allowing us to optimally vectorize over multiple blocks without duplicating the data that captures their sparsity patterns.

Different solvers for Newton Step Three options were investigated *(a) Full Cholesky* – We experimented with using a direct (complete) Cholesky solve at each Newton step, via PARDISO. The resulting Newton iteration cost was 31.8s, more than three times the cost our method would require for 250CG iterations (9.36s) and near-perfect convergence. However, our method hardly needs that many CG iterations to achieve excellent Newton convergence, and in the long run easily outperformed full Cholesky by more than an order of magnitude.

(b) Incomplete Cholesky PCG – ICPCG performed very well in our examples, often requiring half (or less) of our CG iterations for comparable convergence. It is, however, in principle a serial algorithm. Our adequately optimized (albeit serial) implementation required 7.23s to factorize the preconditioner (ours: 291ms) and 422ms (ours: 33.3ms) for each CG iteration. *(c) Block Jacobi PCG* – A parallelism-friendly alternative to ICPCG was to compute a Block Jacobi Preconditioner, with block sizes comparable to our own macroblocks. Matrix entries that straddle blocks were discarded, and a standard Cholesky factorization of the resulting block-diagonal matrix computed via PARDISO. Convergence of this option was generally comparable, and at times slightly better than our solver. This parallel method required 1.24s for factorization (ours: 291ms) and yielded a CG iteration cost of 183ms (ours: 33.3ms).

8 PRACTICAL DEPLOYMENT FOR INTERACTIVE SIMULATIONS

This final technical chapter focuses on the issues surrounding the deployment of Simulation Assisted Visual Systems (SAVS). The organization of this chapter is divided into two main parts. First, there will be a short review of the high level technical concerns regarding deployment. Second, the developmental history of our benchmark surgical application will be described, highlighting the choices, and their rationals, made along the way.

8.1 Deployment Issues

At a high level, there are many decisions involving technology that need to be addressed when deploying large systems. In this section, we'll briefly look at three of the most critical for our benchmark application: Remote architectures, Platforms, and Maintenance.

Network Architectures One of the largest architectural questions we need to answer is whether we want a wholly local application or a networked application. The advantages of a local application include decreased complexity, potentially better access to client resources, and more control over data. Unfortunately, local applications are also completely dependent on local resources. For simulations, which are computationally intensive, a purely local application might be difficult to deploy at scale due to hardware costs. The addition of networking allows us to break many of these issues apart and distribute them more fairly across multiple machines. In such an architecture, servers capable of large computations could service multiple clients, enabling larger deployments with less capital investment. On the other hand, network designs become dependent on another system: The network itself. For locations with poor network connectivity, such as in the developing world, local installations might be more valuable, despite the up front costs.

Platforms Platform support is another important concern. Currently there are many popular consumer facing application platforms. The application platform, which is often simply the computer's operating system, defines what features and services are available to the application. We have many choices in choosing these platforms, depending on the device we which to run our simulation system on. For the personal computer, operating systems such as Microsoft's Windows, Apple's OSX, and multiple distributions of Linux, can fulfill the role of the application platform. On smaller devices, such as phones and tablets, Apple's iOS and Google's Android are extremely popular choices. Finally, the Internet itself, through the combination of HTML and Javascript on web browsers has become a de facto operating system for many people.

Interestingly, all of these platforms, despite their differences, support the user interface technologies (user input, 2D and 3D rendering) required for surgical simulation applications. Unfortunately, not all of these platforms (when considering the operating system and device combined) can support the computational requirements we require. This presents a curious situation where it is possible to develop cross platform user interfaces, but not necessarily cross platform simulations. As we'll see later, dividing these responsibilities allows for more flexible architectural solutions.

Maintenance One final technical issue is the concept of updating and maintenance. Despite the focus of deployment in this chapter, it is important to remember that there are plenty of ongoing research questions regarding simulation, surgical or otherwise. Investigations of these questions might result in new requirements for hardware or software environments. Inevitably, the simulation system we are attempting to deploy, and possibly the underlying platform itself, will need to be updated - whether to support a new simulation technique or to fix more mundane bugs. When this situation occurs, the method of deployment can greatly affect the ease of this process. Local applications can be the hardest to manage under these circumstances, since they are by nature decentralized and might require wide scale hardware

upgrades. Network based deployments, where the simulation is run on remote servers, offer more opportunities for more centralized upgrades.

8.2 System Architecture Comparisons

Over the several years of research leading up to this dissertation, multiple system designs for a surgical SAVS were developed and tested. In this section, each design will be covered in detail along with its particular benefits and problems. The goal of this section is to demonstrate how different system designs fit with different deployment concerns and how emerging technologies can support simulation deployments more easily than in the past.

The core feature set for all implementations, which was defined through conversations with our domain collaborators including Dr. Court Cutting, is described below.

1. **Real-time physics simulation** 3D models of tissue were simulated via an embedding lattice deformer design, allowing users interactive visualizations of physical responses. Multiple models were made available for simulation, ranging from more academic shapes designed to illustrate particular principles to a more realistic model of the human scalp.

2. **Keyboard and Mouse Interface** Users were presented with an open 3D space which contained the actively simulated model. They were allowed to move the virtual camera through the space using their keyboard and mouse. Additionally, tools (such as a virtual scalpel and virtual sutures) were made available via a simple "click and place" interface.

3. **Local Flap Operations** All models represented a 2.5D (generally a thin, possibly curved, sheet) region of tissue with a defined top and bottom side. Intended operations were "local flaps", where simulated tissue was cut and moved around in a local region for the purposes of closing holes or relieving stress patterns.

4. **Free Form Cutting** Incisions were placed into tissue via line segments drawn on the top surface of models. Cuts were allowed to be defined with arbitrary complexity, including the isolation and removal of regions, with the only restriction that they all be performed all at once, before simulation began.

5. **User Interactions** Users were allowed to interact with the running simulation via force constraints. These constraints took the form of "hooks", positional constraints that pulled simulated material towards a point in space, and "sutures" double ended constraints that pulled one region of simulated tissue towards another region.

6. **Operation Recording/Playback** Users were allowed to record sequences of operations and then play them back on subsequent run throughs. This functionality was designed to replace the more traditional approach of drawing operation steps in textbooks.

The tool's user interface was proscribed and prototyped by our primary domain collaborator, Dr. Court Cutting. The choices made in the process of designing the interface were made in response to the background and familiarity to three dimensional modeling tools of our expected user base, plastic surgeons.

Traditional Desktop Application: Building the Monolith

The first design approach followed that of many traditional native desktop applications. All components of the application were compiled together as a single binary executable which ran on a single machine. This initial design used C++ as the development language, both for the underlying simulation engine and for the graphical user interface. This approach had many positive aspects:

1. **Single Development Language** By using one language for all components, building APIs was made considerably easier, which facilitated faster development.

2. **Integrated Application** Since the entire application was linked as a self-contained binary, it was easy to setup and run. External dependencies amounted to several libraries required for the graphical interface and OpenGL graphics.

3. **Accelerator Access** By running entirely on a single machine, the simulation engine had access to all computational resources on the client, including all CPU cores and available accelerator cards. In the latter case, MPI was used to communicate with installed Intel Phi accelerators[1].

4. **Fully Featured Graphics Stack** Running as a native application, this design had access to the full OpenGL 4.0 specification. This allowed it to be developed with modern OpenGL techniques.

5. **Platform Control** Since everything was designed by hand, all aspects of the application could be tuned to desired functionality. While certain choices were constrained by selection of a GUI toolkit library [wxWidgets, 2017], the choice of the library itself was made freely.

In total, these aspects made for a well-performing client, capable of running optimized simulation code and displaying a functional visual interface for the surgical simulation. Unfortunately, it had several drawbacks that ultimately made it a poor choice for future development:

1. **Lack of Cross Platform** Due to it being designed purely with native C++ code, moving to new platforms was exceedingly difficult. For desktop operating systems, such as Windows and OSX, switching was a matter of recompiling and working out platform incompatibilities. The choice of wxWidgets as a GUI toolkit library helped in this regard, but for more mobile platforms, like iOS, Android, or HTML, there were no options.

[1]This unfortunately broke the "self-contained binary" aspect of this design, as the MPI routines required multiple processes running on different physical nodes. However, since the MPI enabled application could be launched via the standard "mpirun" command, it still resembled a self contained application.

2. **Complex GUI Development** While the GUI library, wxWidgets, was chosen for its cross platform capabilities, it was not especially friendly towards rapid development of useful user interfaces. As such, the GUI was rather difficult to use and hard to modify.

3. **Scalability Concerns** The design's primary strength - a self contained application, was also its largest drawback for deployment. By executing on a single workstation, it required an entire workstation per instance. This was unpalatable for larger installations, such as a classroom setting, where few institutions could or would be willing to buy high end workstations for each student.

It was this last drawback that pushed our development of the system in a new direction: network based simulation.

Network Simulation: A Three-Tiered Architecture

Unlike the previous design, the next iteration explored the potential of network, or cloud, based simulation. The idea was straightforward: it would be an architecture where simulation would be performed on a remote machine and its results displayed on a local client. These initial forays into this idea were built on top of the previous client and used an network interchange library from the Apache Project called Thrift [Apache Software Foundation, 2014]. Thrift allowed the creation of multi-language bindings to be delivered over network as RPC (Remote Procedure Call) APIs. Initial experiments involved splitting the former unified application into two parts: a visual frontend and a simulation backend. These parts were be joined together via Thrift, allowing them to communicate over the network.

Once this new organization was working, the frontend was rewritten as a web client, which used HTML, Javascript, and WebGL [Khronos Group, 2017] to replicate the functionality of the older C++ GUI. Further experimentation with this arrangement eventually showed a significant problem with Thrift: it made developing smooth, lag-free user interfaces somewhat tricky. The fault primarily laid with Thrift's RPC style conventions. Under this

communication style, a client would continuously poll the server for new information. Under a multiple client scenario, this method broke down, causing unnecessary network traffic and lag. This issue caused us to seek a new communication method, one that could support a "push" style communication. We finally settled on a newer Web standard: Websockets.

The Websocket standard [Fette and Melnikov, 2011] is a full-duplex communication protocol available in HTML5. The standard defines a custom protocol over TCP, where messages can be sent from either end and are guaranteed to arrive in order on the other side. The primary benefits of the protocol are easy bidirectional communication and low overhead. Since the protocol uses a custom TCP-based protocol instead of HTTP, it can operate without unnecessary headers and transmit data as byte streams. Replacing Thrift with Websockets was the final piece of what was later referred to as our Three-Tiered Architecture.

The three-tier design consisted of the following components: a web-based client (Tier 1), an SMP server for modeling (Tier 2), and a many-core accelerator for numerics (Tier 3). The *front-end client* served primarily as the user interface to the system. The client was responsible for acquiring user input and visualizing the simulation results. Communication is performed via the WebSockets standard, which allows the client to operate under a push data model. Thus, the remote server could send updates to clients when they become available instead of requiring clients to poll for updates.

We refer to the *second-tier platform* as the CPU host. Its primary task was to perform all non-simulation computation that can be offloaded from the client level. The host manages user sessions, stores and loads scene data from disk, performs geometric manipulations (non-manifold meshing and incision modeling) as a result of user actions and runs collision detection. This tier requires large amounts of memory, beyond what is natively offered on a GPU or Many-Core accelerator. The *third tier* is the numerical solver, which executes all low-level compute intensive kernels (but not combinatorial tasks such as mesh generation). We have implementations that allow this layer to either run on the same CPU platform as the 2nd-tier code or run natively on a Xeon Phi accelerator, interfacing with the host over MPI.

This later functionality is similar to how the initial monolithic system design functioned.

The specific benefits from this design approach were:

1. **Fast GUI Development** As the final implementation of the frontend was written in HTML and Javascript, iteration on the GUI was much faster due to visual debugging tools and more mature support libraries for web application development.

2. **Native Support for Network Simulation** Since the frontend was designed as a web application, by nessessity it needed to connect to the simulation engine via network protocols. By using Websockets, we were able to construct a server side implementation that could broadcast the current simulation state to multiple connected clients simultaneously. This allowed for centralized simulation and more scalable deployments.

3. **Cross Platform Support** While the simulation engine was still using native code, and required additional effort to port to other platforms, the HTML frontend was able to run on multiple platforms and form factors with minimal development effort. We were able to successfully deploy the client on desktops, tablets, and smartphones. Since the client's computational responsibilities were limited, it could be run a wide range of hardware as long as it had a reasonably moderate graphics accelerator.

4. **Service Integration** Finally, a side benefit of the Web application frontend was the possibility of integration with other web services. For production deployment, this would allow easier connection with existing web authentication platforms and online help desk services.

However, no design is perfect. Like the previous monolithic design, the three-tiered design had several drawbacks:

1. **Complex Infrastructure** While it was praised earlier, there is no denying there exists added difficultly when deploying and running a network application that must

coordinate between multiple independent nodes. This is the unfortunate double edged sword aspect of any network or cloud service.

2. **Network Bandwidth Dependency** Unlike video games, where visual assets are typically cached on clients and then modified according to predetermined scripts, the geometry of deformable simulations is inherently unpredictable by the client. As such, every frame of the simulation requires transmitting all of the deformation data down to the client. For low polygon models this is manageable, but highly detailed biological scenes can require significant amounts of network bandwidth, or they risk causing unacceptable lag for clients.

3. **Loss of Platform Control** While using a web client provides a large amount of cross platform support, it places the client at the mercy of the web browsers that are running it. One of the primary challenges facing web developers is the number of small differences between browsers (and versions) that can sometimes mean the difference between a working client and a failing one.

4. **WebGL Only** While the WebGL standard contains most of the features that the modern OpenGL standards support, there are gaps and differences in functionality. However, as more and more 3D applications find their way onto the Web, these standards are becoming more and more robust.

While this approach proved successful and provided many nice benefits for developers, it did so at the cost of added complexity nearly all cloud based systems create. The difficulties of managing a distributed system, both in development and for the potential to be deployed in areas with unreliable networks, inspired the third design approach, which attempted to combine the benefits of the two prior architectures.

Monolithic Web Design: Electron

The third design made use of a relatively recent framework called Electron. Electron [GitHub, Inc, 2016] is a framework for building traditional desktop applications by using web application development tools. It does so by combining Google's open source browser Chromium and NodeJS [Node.js Foundation, 2017], an interpreter and set of standard libraries for Javascript which allow more traditional system level calls, such as file access and process management. This framework provides developers access to a familiar web development environment, along with access to the underlying operating system, functionality normally inaccessible in the browser. Using this framework, it was possible to build the third design architecture: the monolithic web application.

The primary concern with using Electron as a the application framework is performance. With the original design, everything was written with native code, which allowed for optimized code to implemented with appropriate care. The second design ran a native code simulation server, providing effectively the same benefits over the network. For the Electron based design, a different approach was required. In order to maintain native performance where it was most needed, the simulation engine was repackaged into a native NodeJS extension. Using the API provided by V8, the underlying Javascript interpreter for both Chromium and NodeJS, the native C++ code was exposed to the higher level Javascript code. From the Javascript environment, the C++ code could be called just like other Javascript routines, except with native performance - they were able to use all of the multithreading and SIMD optimizations described in previous chapters.

With this native wrapping completed, the rest of the application could be ported over from the second architecture design in a straightforward fashion. The web client code required minimal changes, since it was still running in a web browser like before. The only new code required was a re-implementation of the connection between the client and the simulation engine. This was again done with Websockets, only this time as connections entirely internal to the application. This was a conscious choice - allowing for a potential client-server mode

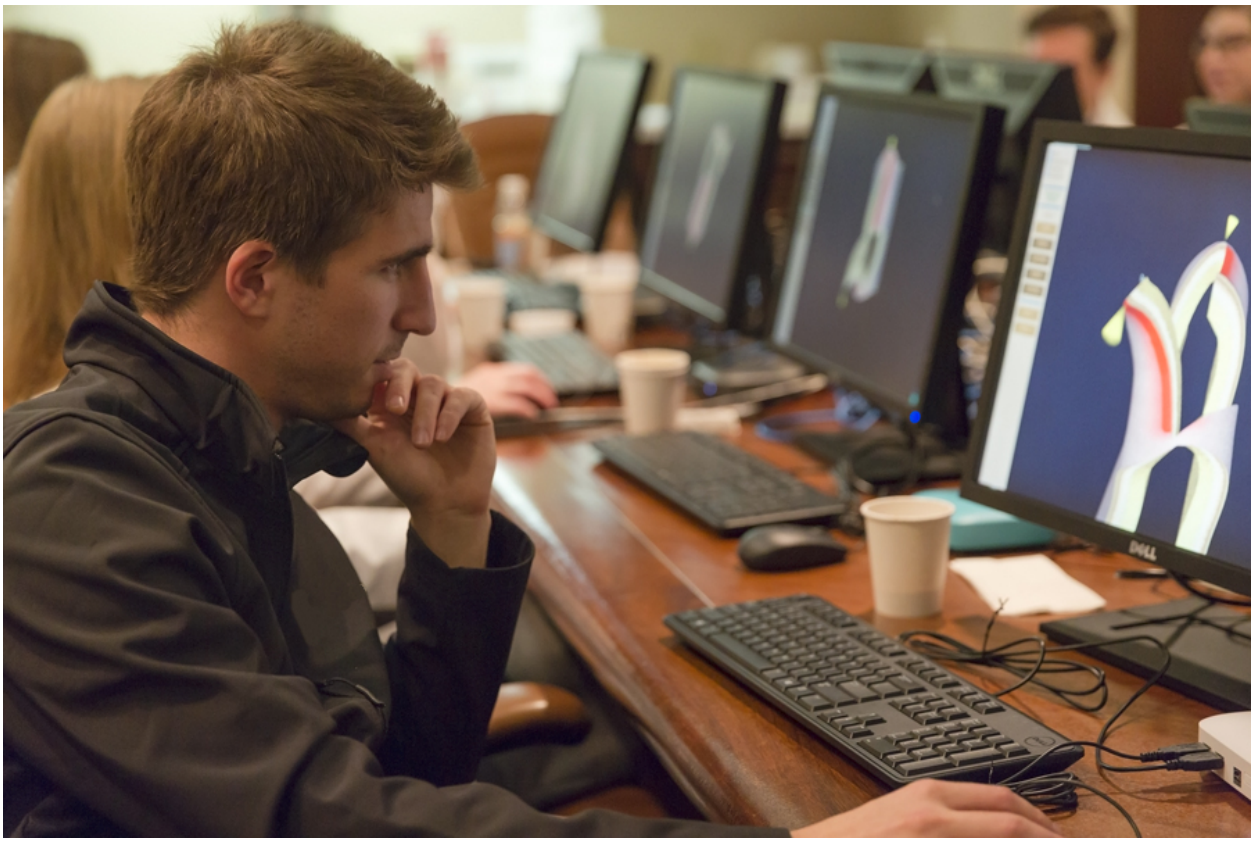


Figure 8.1: Pilot deployment of web-based simulator
Pilot deployment of our interactive web-based simulator with 14 Plastic & Reconstructive Surgery medical residents at the University of Wisconsin-Madison. This demo used the second architecture, the three-tiered design.

in the future. This design successfully merged the major positive characteristics from the previous two designs: an integrated desktop application, access to local computational resources, faster user interface development, and easy access to network simulation. The major drawbacks of this design were the added complexity of the multi-language bindings and a dependency on the Electron framework[2].

8.3 Deployment Study

Seeking to evaluate our system, we conducted a pilot deployment for the medical residents in the University of Wisconsin-Madison Plastic & Reconstructive Surgery program, as part of a workshop on craniofacial reconstruction techniques. This deployment used the second architecture described in Section 8.2, the three-tiered design. We set up a local switched network at the location of the workshop, with a portable desktop-grade computer (with a 4-core Intel 4770R processor and 16GB RAM) acting as the Tier 2/3 simulation server, and a collection of eight Chromebox web-based thin clients as the user stations (our setup can be seen in Figure 8.1). Although our modest portable server did not have the computing power available to our many-core accelerated server systems which we used for our large-scale offline simulations, its (AVX-accelerated) performance was more than adequate to deliver an interactive user experience. Total cost of the entire deployment, including server, networking and client stations was $4,000.

The primary goals of the test were twofold. First, we were interested in determining the technical feasibility of the system under active, real-world conditions. Second, we wanted to gather initial impressions from likely users in order to help guide future research. Participants were initially given an extremely brief orientation on visual navigation within the application and its interface. In first part of the deployment exercise, the workshop instructor (a seasoned user of the system) authored several different reconstructive procedures on different anatomical models, while the participants were invited to follow the manipulations as they were taking place (without affecting them) by adjusting the view of the dynamic model on their own station. Subsequently, individual participants who had no prior exposure to the system were invited to drive the authoring process, which their colleagues would virtually follow. The workshop instructor provided guidance on clinical aspects of the repair being authored, while questions about the user interface would be recorded and addressed by the system developers.

[2]However, since we are now only targeting a single web browser, the one bundled with Electron, we are freed from worrying about supporting multiple browser versions.

At the conclusion of the exercise, the participants were debriefed and given the opportunity to evaluate their experience and propose improvements.

We found that our participants who used our benchmark tool were extremely comfortable with aspects of 3D visual navigation, even with the very rudimentary orientation that was provided. Most participants found the visual examples of procedures demonstrated to be very enlightening, with many of them commenting that this visual illustration was the most informative exposure they had for procedures they only knew from reference literature (most of them had not witnessed these procedures in the operating room). Almost no participant volunteered the lack of self-collision processing as an observed omission, until the interviewer explicitly asked them about this aspect (all demonstrations in our workshop entailed full suturing of wound closure). On the contrary, several participants identified inaccuracies in the elastic behavior of the virtual tissues, finding that our models appeared to be more "permissive" to manipulation than real flesh tissue. An interface feature that was pointed out as lacking was the inability to appreciate (simply by looking at the final sutured result) the deformation patterns that have resulted from a certain repair; it was suggested that adding a texture (grid lines or checkerboard patterns) on the skin surface would be much more useful in evaluating tissue strain and deformation. We also received requests for biologically inspired aids - in particular a visualization of anatomical elements such as blood vessels in the tissue being cut. These additions were requested for practical reasons - sub-dermal blood vessels and nerves are important to preserve during operations. Several users requested more traditional animation features, such as timeline scrubbing and history undo, as well as side-by-side views of different repair approaches for visual comparison.

9 DISCUSSION

The conclusion of this document, presented in this chapter, will attempt to draw together concepts and lessons from previous chapters. Included among them are salient ideas that derive from the accumulated knowledge picked up from the experiences and research that produced this document, known and discovered limitations of the methods previously described, and potential avenues for future work. The structure of this chapter is as follows: the next several sections will touch on themes and concepts that precipitate from the earlier technical chapters, highlighting several important lessons learned. This will be followed by a section describing currently understood technical limitations of the techniques presented thus far, along with directions for future research.

9.1 Themes

Comments on Vectorization

Despite support for vector instructions existing in consumer grade hardware since the late 1990's, developing applications that successfully and efficiently use this computation model can still present serious complications. This difficulty arises from multiple directions, which is perhaps a reason for its continual presence. The major challenges encountered during the work that led up to this dissertation can be grouped into several categories: design, abstractions, and debugging.

Design Our experience with designing data structures and algorithms for use with vector instructions was centered primarily around need. While we would have preferred using existing automatic tools and middleware that could have abstracted away the challenges of directly writing vector-aware code, our experiences showed us that existing solutions were unsuitable for the strict optimization and performance requirements we operated under. We

were required to significantly restructure data structures and algorithms in order to extract vector friendly parallelism. At present, it seems clear to us that it remains a developer's responsibility to organize algorithms and data structures in vector-friendly ways, such as the blocking designs demonstrated in Chapter 6. And even with these invasive restructurings, modern compilers struggle to produce optimal vector code: minimizing the amount of register spills into main memory, inserting appropriate prefetching instructions, and other such optimizations can often must be completed by hand.

Abstractions At a level higher than designing a appropriate algorithm or data structure for vectorization, we also were faced with the challenge of making such designs portable to different vector architectures. Our desire to have our software be portable to multiple platforms required us to focus on a variety of situations, including closely related architectures, such as the multiple vector instruction sets in the x86 family of processors, and between architectures, such as between Intel Phi accelerator cards and GPUs. The work demonstrated in Chapter 6 on the vector library class was designed to assist with this problem, making it easier to port code between multiple vector instruction sets in the x86 architecture family. Other related work involving heterogeneous simulation of fluids [Liu et al., 2016] presents an example of the other problem, where an abstraction layer needed to be created to communicate with Intel Phi accelerators, GPUs, and standard CPUs on an equal basis, along with multiple implementations of the same algorithm tailored for each architecture. It remains an open question on how to best support such a heterogeneous architecture environment, especially when high level abstractions can easily prevent exploiting architecture specific properties required for optimal performance.

Debugging The last major issue that arose during our development process was how to successfully debug the vectorized code we were writing. This step proved more challenging than initially anticipated for several, not immediately obvious, reasons. Some of the problems were due to the structure of the code itself: By using vector instructions we are generally

breaking two commonly depended upon aspects of code. First, a vector instruction acts upon multiple, independent items of data at once. Because of this, identifying issues often required tracing multiple concurrent computations at the same time, as they are impossible to separate during execution. Second, due to the fact that vector code is typically branch-free (or close to it), we could not depend on tracing different code paths as a indication of problems. Instead of being able to compartmentalize the process along isolated branches, we were forced to look at the entire algorithm as one continuous flow when diagnosing issues. All of these problems would be challenging enough, but vector instructions provide one last lingering complication during debugging. Discovered during the creation of the vector kernels in Chapter 6, not all vector instruction sets, despite providing the same semantic instructions, will the same produce bit-identical output as each other for certain instructions. The error is generally low, small enough to not matter in most practical scenarios, but its existence makes checking numerical correctness difficult when exact answers cannot be relied upon.

Intuition: A Double Edged Sword

One of the big themes in this dissertation has been the use of data structures for non-manifold geometry. Our reasons for choosing this family of geometry representations were primarily based on our intuition that they would maintain many of the regularity features we enjoyed from more strict grid-based data structures, while improving upon the topological flexibility of those designs. For the most part, we were satisfied in this regard - somewhat ironically, one the biggest challenges we faced with this approach was not technical, but our own geometric intuitions. The major problem we discovered with non-manifold representations is that there are no real world analogs with which we could relate to. Real objects, unlike non-manifold geometry, can not self intersect - this is a purely non-physical concept that makes certain algorithms easier to develop. As a result, we struggled with these approaches as, unlike with many other topics in visual computing, they proved to be very difficult to visualize mentally. An example of this problem occurred during the development of the backtracing algorithm

for non-manifold level sets. Initially, due to our intuitions at the time, we focused most of our effort on defining the surface of the geometry and how the data structure would capture this information. In the end, we realized that this line of thinking, while it made sense from the perspective of visualizing our virtual object, did not actually help us answer the real question: where was the closest surface point. Refocused around this more narrowly scoped goal, the backtracing algorithm described in Section 5.3 came together much more naturally.

This problem with intuition is especially present as one transitions from two dimensional domains into three dimensional domains. Unlike other properties, such as those based on volume, whose complexity might only change from a quadratic scaling to a cubic scaling, topology is much more complex. As an example, consider the number of ways material can be connected in a non-manifold fashion across a hexahedral cell in two dimensions, if we think back to the idea of material continuity described in Chapter 5. To begin, at least one of the two shared nodes across the edge must share a material connection - which immediately leaves the other node to either be connected or not. If we restrict the question to representing non-overlapping geometry and simply those objects with bifurcations not resolvable by the grid, this leaves us with a total of two patterns: two cells attached across an edge to a third, each containing material on only one node, and its symmetric opposite. However, by introducing just one more dimension, changing edges to faces, the number of possible material patterns jumps to twenty nine symmetric pairs. The true danger is not that the potential patterns are hard to incorporate into an algorithm, but that a developer might miss them. The nature of these non-manifold geometric data structures, and associated algorithms, is that they are difficult to imagine. And what cannot be imagined is all too easy to overlook.

Software Engineering Lessons

A large amount of work completed in pursuit of this dissertation was dependent on good software engineering practices. From these experiences, we have distilled several important ideas that are worth further discussion. The first is a general statement on optimization

strategy. Often algorithmic optimization is judged by comparison: by what percentage does implementation A outperform implementation B? Instead, we find that a more absolute approach is preferable, allowing us to have a better sense of how well our algorithms perform and, often equally important, how much performance is being underutilized. Our approach, given the general problem of operating over large discrete domains, has been to consider the reading and writing of data as the primary limiting factor in algorithmic performance. This is justified on the following ideas: first, no work can be completed before all the inputs have been read and all the outputs have been written, and second, we recognize that in most modern systems have a imbalance of memory bandwidth and computational bandwidth (one of the motivating factors in Chapter 7), making data transfer generally the slowest component of any computation. By comparing the performance of an implementation to the theoretical time required to move its data set into and out of main memory[1], we can derive a measure of *algorithmic efficiency*. We find this method more useful, practically, than the common approach of comparing against existing implementations (either ours or someone else's). By comparing against an absolute reference point instead of a relative one, for a particular platform at least, we had confidence that we could not do better than we had done. This proved especially important when considering whether or not a particular intervention was successful *enough*.

Related to this approach, the next software engineering advice we have found valuable during our work is that memory bandwidth is extremely important and generally becomes a major consideration sooner than people expect. This is somewhat at odds with conventional knowledge that faster processors (i.e. improved computational bandwidth) is the primary route to improved performance. Instead, our experiences have shown that it is very easy to run into bottlenecks in terms of memory transfers, even when not using vector instructions. For instance, since memory is loaded by cache line, runs of sixty four bytes, even scalar

[1]We assume, during this computation, that we have perfect caching of all information read, prefetching and other memory access optimizations. We are purely interested in the amount of bytes read and written, divided by the amount of memory bandwidth provided by the hardware platform in question.

algorithms that load memory in unstructured ways can suffer from poor memory bandwidth utilization. Worse, these problems can be difficult to see - while analysis tools can report general issues, the causes are often subtle and structural in an implementation.

A final concept that we espouse is that understanding the particular nature of a problem can lead to more effective solutions than existing powerful, but generic, solutions. Our macroblock solver detailed in Chapter 7 is a great example of this philosophy. Existing algorithms for matrix reordering (which is recognized as NP-complete), such as Minimum Degree Ordering, designed for reducing fill-in, could not compete with our own reordering scheme because it was tailored for the specific sparsity pattern of our macroblocks. Similarly, despite attempting to use generic vectorization tools and libraries initially, we found that developing our own library, tailored for the kinds of computations we were performing, provided more control over the final generated machine instructions. While it might have been more expedient to use generic solutions, our benchmark application goals required more than these methods could provide. The trade-off between performance and development time was judged to be worth taking.

Domain Experiences

The experiences of working with domain experts during the research and development that went into the surgical simulation benchmark has left a lasting impression and driven home several important ideas. The most important among them is the sense that maintaining a broad attitude when approaching new projects is beneficial and in many cases necessary for their success. Commonly, researchers focus on narrowly defined problems, only reluctantly moving into higher or lower levels of abstraction. And while this focus can be desirable, potentially leading to great advances, had we restricted our research questions to simply pursuing theoretical improvements or algorithmic designs, we would never have arrived at a successful tool that was both usable and high performance. In the quest to support the needs of our domain experts, we saw no choice but to move down the levels of abstraction

to acquire additional performance by tackling questions all the way down to the hardware level. Likewise, as we wanted a system that was flexible to be used in a number of real-world scenarios, we moved up from pure simulation goals and looked at challenges in user interfaces and network service design. Being willing to move in this space did mean that less depth was covered, but it came at the benefit of seeing the system from a complete perspective, allowing for cross-cutting interventions that required a broad take on the situation. Moreover, it was important to step outside of the field of computer science to learn about the concepts and problems that were important in the field of plastic surgery in order to effectively converse with our colleagues in this domain. What resulted was a broad covering of many areas, with perhaps less depth in any one area, which produced a successful end product, filled with both compelling research results and solid engineering accomplishments.

9.2 Broader Applications

Before discussing specific limitations and future work, it is worth considering the broader applicability of the techniques described in this dissertation to areas outside of plastic surgery simulation. While the benchmark application discussed throughout this document was chosen for its intrinsic challenges and potential impact in the field of surgery, it also served as an exemplar of challenging problems in other domains. In this section, we will look at the major technical contributions covered in this document and how they apply to other tasks.

The first contribution area, non-manifold embedding and level sets, has a strong applicability elsewhere in computer graphics, including general simulation and geometric processing. Even with nothing else added, the ability to embed, simulate, and handle collisions between objects in a relatively coarse grid while maintaining topological separation of thin features is desirable. While we used these capabilities to embed thin incisions, classic simulation problems such as character animation and fracture modeling would all benefit from being able to embed sub-voxel resolution material, such as fingers and shards of a model, respec-

tively. Another interesting area is geometric model cleanup, where being able to construct a level set of a self-intersecting mesh could enable mesh detangling algorithms. Other areas where non-manifold embedding and non-manifold level sets could be very valuable would be multiphase liquid simulations. In these situations, multiple types of liquid need to be in very close contact with one another. Non-manifold level sets could be used to capture their interfaces, while still allowing multiple liquid interfaces to exist in extremely close contact.

The next contribution, the vectorization library and related performance optimizations, also has general applicability - here beyond just computer graphics. While the contributions of data layout for improving simulation performance are certainly transferable to other deformable solid simulation tasks, such as character animation, the core vectorization library could be applied much more broadly. The ability to easily write and maintain vectorized numerical kernels is valuable in any scientific or engineering effort where performance on modern computational hardware is critical. Moreover, the work presented in this document provides a roadmap and a list of hazards for future developers to use when developing their own multi-threaded and vectorized solutions. This last point should not be understated, as we found such guidelines often absent during the development of our solutions. Having them available to the general community is a contribution in its own right.

Likewise, the macroblock concept and associated solver also has applicability to general problems. These ideas were originally designed to solve issues we were experiencing with convergence in regions too thin (such as the scalp area) to handle with existing solutions like multigrid solvers. However, as we investigated the idea further, it became clear that the hybrid iterative-direct macroblock solver could be used successfully on a broad collection of linear algebra problems defined over a grid discretized domain. The type of connectivity pattern the technique depends on is fairly common, as grid discretized simulation and optimization problems are present in computer graphics, mechanical engineering, additive manufacturing (3D printing) research, and many other areas. One idea we are currently exploring is how a macroblock style solver could be used as a drop in replacement for existing numerical

packages such as PARDISO [Petra et al., 2014b,a], as long as the problem is defined over a grid.

Finally, the deployment investigation has revealed several important findings that could apply outside the scope of surgical simulation. The major points here are the practicality of remote simulation and the rise of web technologies as front ends for computationally intensive tasks. While there exist open questions concerning network bandwidth and lag, remote simulation for interactive applications is possible - which opens up a broad range of applications, including animation tools for the graphics professionals, to more consumer oriented applications such as video games and simulation assisted activities, like shopping for clothing virtually. Building tools for these tasks can be assisted by the use of web technologies for user interfaces and connectivity. These frameworks, including Electron, have shown themselves to be ready to handle complex graphics applications, including simulation, during the trials completed as a part of this document.

In the end, we feel strongly that using a surgical simulation as the benchmark to develop the contributions of this document was the correct choice. It presented numerous challenges that resulted in solutions that not only addressed the direct needs of our application, but also applied to a broader collection of problems in computer graphics, scientific computing, mechanical engineering, and more consumer focused fields. Far from being limiting or restricting our vision to a narrow problem, surgical simulation was a true springboard to an even larger world of challenges.

9.3 Limitations and Future Work

In the remainder of this chapter, we will explore some known limitations of the techniques presented in this document, along with some potential places where future work could provide improvements.

Non-manifold Level sets

Our pipeline for non-manifold level sets in Chapter 5 was been specifically created for self-collision processing in deformable body simulations. There are several diverse applications of the standard (manifold) level set concept such as representing geometry, tracking dynamically evolving interfaces, as a geometric query structure, etc. However, we believe that the current methodology may require application-specific embellishments to cater to broader tasks in modeling and simulation, beyond what was needed for our collision-oriented proof of concept. For example, when two dynamically evolving interfaces are brought together, they may either be allowed to merge (as is typical in fluid simulations) or overtake one another (e.g. the two separate branches of the lips during self-collision). Thus, application-specific semantics might be needed to use non-manifold level sets for dynamic interfaces. Finally, we showed two examples (Figures 5.11 and 5.9) where the simulated elastic model resulted from conceptual "cutting" operations; in both cases we simply generated the non-manifold level set from the final, cut geometry. If such cuts were progressively enacted during simulation, our present implementation would sustain a significant recompuation cost to rebuild the non-manifold implicit representation. Incremental update following localized topology change will be a significant direction of future investigation.

Currently, we do not extend the non-manifold level set values into a narrow band outside the interface because we used the same grid both for simulation and for storing level set values. If the extent is predetermined, then we could generate a coarse non-manifold level set mesh first and refine it as a post-process. Alternatively, one could also "grow" the non-manifold level set by following characteristics, but as mentioned above, one may or may not wish to merge overlapping characteristics depending on the context. This issue is also related to the question of "evolving" a non-manifold level set. In future work, we wish to address these questions targeting the specific applications of multiphase flow and crack propagation. For standard Cartesian grid-based level sets, there exists an established theoretical foundation for accurately computing high order gradients even in the vicinity of singularities. For

non-manifold level sets, an extra level of complexity is introduced because there can be a topological bifurcation in addition to a singularity. To obtain valid values in such complex situations, our current scheme reverts to first order element-wise trilinear interpolation. It would be interesting to explore in future work if there can be a more accurate representation for the interface in these cases. In all of our examples we generated the non-manifold level set at the same resolution as the underlying lattice-based elasticity discretization. This was motivated by the fact that the ability of the elastic model to respond to collision events is restricted by the resolution of the elastic model, limiting the hazards of interpolation errors near high-curvature areas. Although this has been a successful heuristic for our test cases, there is no guarantee that such resolution will always be accurate for proper collision detection and response. A higher resolution can be used for the level set, if desired; doubling the linear resolution, for example, would incur an 8x increase in level set construction, but no worse than a two-fold increase of the backtrace cost, in practice (assuming collisions remain relatively shallow).

Finally, the current implementation used Cartesian grids as template meshes for generating the non-manifold level set. In future work, an exploration of different representations such as octrees [Losasso et al., 2004], RLE representations [Houston et al., 2006, Irving et al., 2006, Chentanez and Müller, 2011], the VDB data structure [Museth, 2013] and SPGrid [Setaluri et al., 2014a] would be warranted.

Performance Optimization and Macroblocks

In Chapters 6 and 7, we looked at two different approaches for improving the performance of computing forces and solving elastic deformation problems. To some extent, the two approaches are the result of sequential development. While the blocking organization allowed us to compute forces for elasticity problems very efficiently, we found that the addition of large numbers of force constraints and complex materials, like muscle fibers, negatively impacted our ability to achieve reasonable convergence with Conjugate Gradients, the iterative solver

we used for the surgery simulation prototype. At the time, this was not a large problem, as we used the partially converged steps of the simulation as an approximation of dynamics. Moreover, the relatively rapid CG iterations allowed us to present more visual updates to users, giving the platform a more responsive feel. Nevertheless, we were hoping to improve upon the poor convergence we were seeing, a quest which ultimately led us to the macroblock idea. Unfortunately, despite achieving the goal of better convergence and a better ratio of memory to computational resources, this technique fell short in several important areas.

The most important limitations of our macroblock formulation are (a) the restriction of our scheme to Cartesian lattice-based discretizations of elasticity, and (b) the explicit lack of support for self collisions or other elastic interactions that would couple together disjoint parts of the mesh. We consciously limited our preliminary exploration to applications of macroblocks within a Newton-Raphson iterative solution scheme. In principle, there would have been an opportunity to also consider using macroblocks in the design of a highly efficient box smoother for multigrid, or as a replacement of the local optimization step in projective dynamics; we defer exploration of those interesting threads to future work. In terms of practical performance, the macroblock technique also includes an expensive factorization step. While this process should be vectorizable, and thereby mitigating this cost somewhat, it remains a significant bottleneck for interactive use cases at the moment. It is this issue, along with the lack of self collision support, which has prevented it from being included in the current iterations of the plastic surgery simulation tool.

Surgery Simulation Platform

As was mentioned in Chapter 8, the feature set of the surgical tool was relatively modest. These limitations were a conscious choice to allow us to focus on the platform itself, balancing performance and extensibility. From a research standpoint, our biggest perceived challenge is improving the accuracy of the constitutive models for the biomaterials involved. At the time of the pilot deployment described in Section 8.3, full self-collision support as per the work

described in Chapter 5 had not been integrated into the benchmark, but this support will be essential for extending our work to procedures that rely on contact, in addition to sutures, to model properly. Finally, a large class of reconstructive procedures cannot be modeled with all incisions performed at the beginning of time; a flexible topology change modeling system would be necessary to incorporate a seamless ability for topology change, concurrent with deformation.

In terms of technical feasibility, while we did not experience any major technical difficulties during the pilot deployment, the setup environment used was near optimal. Future testing and development will need to focus on robustness to less ideal settings. Some issues that will need to be addressed are improved handling of networks with variable latency and packet loss, protocol bandwidth usage, and general scalability beyond the eight stations used in the pilot deployment. While these are certainly major challenges, we feel they are well within our grasp as they have similarities with other fields such as multiplayer video games and media streaming.

A final note concerns an issue surrounding deployment: maintaining medical privacy. While supporting a surgical simulation tool that allows for remote collaboration might be extremely helpful, it is important to keep in mind the implication of medical privacy regulations, such as the HIPPA (Health Insurance Portability and Accountability Act) laws in the United States. Maintaining compliance with these regulations while still being able to transmit potentially patient specific pathology data to remote servers will be challenging to reconcile. There are interesting open questions about how this might be performed, given that the simulation requires access to the geometry of the model being simulated. One saving grace of our system is that the simulations are executed on an embedding lattice, which is only an approximation of a patient's tissue geometry. One might imagine a protocol that kept the actual geometry being manipulated local and transmitted some kind of obfuscated representation for remote computation. Or with potentially a dual simulation design, where the remote server would compute bulk updates on a generic tissue model and the local client

would use these updates as a highly effective preconditioner for a higher detail, patient specific model. These are questions than should receive more attention before these systems are ready for production use.

BIBLIOGRAPHY

D. Adalsteinsson and J. A. Sethian. A Fast Level Set Method for Propagating Interfaces. *JCP*, 118:269–277, 1994.

J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette, L. Grisoni, et al. SOFA - an Open Source Framework for Medical Simulation. In *MMVR 15*. IOS Press, 2007.

Apache Software Foundation. Apache Thrift. `http://thrift.apache.org/`, 2014.

S. R. Baker. *Local Flaps in Facial Reconstruction (3rd ed.)*. Saunders, 2014.

D. Baraff, A. Witkin, and M. Kass. Untangling Cloth. *ACM Trans. Graph.*, 22(3):862–870, 2003.

J. Bloomenthal and K. Ferguson. Polygonization of Non-manifold Implicit Surfaces. SIGGRAPH '95, pages 309–316, 1995.

S. Bouaziz, S. Martin, T. Liu, L. Kavan, and M. Pauly. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph.*, 33(4):154:1–154:11, July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601116.

R. Bridson, R. Fedkiw, and J. Anderson. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Trans. Graph.*, 21(3):594–603, 2002.

R. Bridson, S. Marino, and R. Fedkiw. Simulation of Clothing with Folds and Wrinkles. SCA '03, pages 28–36, 2003.

M. Bro-nielsen and S. Cotin. Real-time Volumetric Deformable Models for Surgery Simulation using Finite Elements and Condensation. In *Computer Graphics Forum*, pages 57–66, 1996.

M. C. Cavusoglu, T. G. Goktekin, and F. Tendick. GiPSi: A Framework for Open Source/Open Architecture Software Development for Organ-Level Surgical Simulation. *Information Technology in Biomedicine, IEEE Transactions on*, 10(2):312–322, 2006.

D. T. Chen and D. Zeltzer. Pump It Up: Computer Animation of a Biomechanically Based Model of Muscle Using the Finite Element Method. *SIGGRAPH Comput. Graph.*, 26(2): 89–98, July 1992. ISSN 0097-8930. doi: 10.1145/142920.134016.

N. Chentanez and M. Müller. Real-time Eulerian Water Simulation Using a Restricted Tall Cell Grid. SIGGRAPH '11, pages 82:1–82:10, 2011.

N. Chentanez, R. Alterovitz, D. Ritchie, L. Cho, K. K. Hauser, K. Goldberg, J. R. Shewchuk, and J. F. O'Brien. Interactive Simulation of Surgical Needle Insertion and Steering. *ACM Trans. Graph.*, 28(3):88:1–88:10, July 2009. ISSN 0730-0301. doi: 10.1145/1531326.1531394.

M. Corsini, P. Cignoni, and R. Scopigno. Efficient and Flexible Sampling with Blue Noise Properties of Triangular Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 18(6):914–924, June 2012. ISSN 1077-2626.

H. Courtecuisse and J. Allard. Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors. HPCC '09, pages 139–147. IEEE CS Press, June 2009. doi: 10.1109/HPCC.2009.51.

S. De and K. Bathe. The method of finite spheres. *Computational Mechanics*, 25:329–345, 2000.

S. De, J. Kim, Y.-J. Lim, and M. A. Srinivasan. The point collocation-based method of finite spheres (PCMFS) for real time surgery simulation. *Computers & Structures*, 83(17 - 18): 1515–1525, 2005. ISSN 0045-7949. doi: 10.1016/j.compstruc.2004.12.003.

L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

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(11):1663–1675, 2011. doi: 10.1109/TVCG.2010.268.

D. Enright, S. Marschner, and R. Fedkiw. Animation and Rendering of Complex Water Surfaces. *ACM Trans. Graph.*, 21(3):736–744, 2002.

Y. Fan, J. Litven, and D. K. Pai. Active Volumetric Musculoskeletal Systems. *ACM Trans. Graph.*, 33(4):152:1–152:9, July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601215.

F. Faure, S. Barbier, J. Allard, and F. Falipou. Image-based Collision Detection and Response Between Arbitrary Volume Objects. SCA '08, pages 155–162, 2008.

F. Ferstl, R. Westermann, and C. Dick. Large-Scale Liquid Simulation on Adaptive Hexahedral Grids. *IEEE Trans. Visualization & Computer Graphics*, 20(10):1405–1417, Oct 2014. doi: 10.1109/TVCG.2014.2307873.

I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011. ISSN 2070-1721. URL `https://www.rfc-editor.org/rfc/rfc6455.txt`. Updated by RFC 7936.

A. G. Gallagher, E. M. Ritter, H. Champion, G. Higgins, M. P. Fried, G. Moses, C. D. Smith, and R. M. Satava. Virtual Reality Simulation for the Operating Room: Proficiency-Based Training as a Paradigm Shift in Surgical Skills Training. *Annals of Surgery*, 241(2), 2005. ISSN 0003-4932. URL `http://journals.lww.com/annalsofsurgery/Fulltext/2005/02000/Virtual_Reality_Simulation_for_the_Operating_Room_.24.aspx`.

M. Gao, N. Mitchell, and E. Sifakis. Steklov-Poincarè Skinning. In V. Koltun and E. Sifakis, editors, *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. The Eurographics Association, 2014. ISBN 978-3-905674-61-3. doi: 10.2312/sca.20141132.

M.-P. Gascuel. An implicit formulation for precise contact modeling between flexible solids. In *SIGGRAPH '93*, pages 313–320, 1993.

J. Georgii and R. Westermann. Corotated Finite Elements Made Fast and Stable. VRIPHYS '08. The Eurographics Association, 2008. ISBN 978-3-905673-70-8. doi: 10.2312/PE/vriphys/vriphys08/011-019.

GitHub, Inc. Electron. `http://electron.atom.io/`, 2016.

T. G. Goktekin, A. W. Bargteil, and J. F. O'Brien. A Method for Animating Viscoelastic Fluids. *ACM Trans. Graph.*, 23(3):463–468, Aug. 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015746.

E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex Rigid Bodies with Stacking. *ACM TOG*, 22(3):871–878, 2003.

F. Hecht, Y. J. Lee, J. R. Shewchuk, and J. F. O'Brien. Updated Sparse Cholesky Factors for Corotational Elastodynamics. *ACM Trans. Graph.*, 31(5):123:1–123:13, 2012.

J. Hellrung, A. Selle, A. Shek, E. Sifakis, and J. Teran. Geometric Fracture Modeling in Bolt. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, pages 7:1–7:1, 2009.

E. Hermann, B. Raffin, and F. Faure. Interactive Physical Simulation on Multicore Architectures. EG PGV'09, pages 1–8. Eurographics Association, 2009. ISBN 978-3-905674-15-6. doi: 10.2312/EGPGV/EGPGV09/001-008.

B. Houston, M. B. Nielsen, C. Batty, O. Nilsson, and K. Museth. Hierarchical RLE Level Set: A Compact and Versatile Deformable Surface Representation. *ACM Trans. Graph.*, 25(1):151–175, 2006. ISSN 0730-0301. doi: 10.1145/1122501.1122508. URL `http://doi.acm.org/10.1145/1122501.1122508`.

G. Irving, J. Teran, and R. Fedkiw. Invertible Finite Elements for Robust Simulation of Large Deformation. SCA '04, pages 131–140. Eurographics Association, 2004. ISBN 3-905673-14-2. doi: 10.1145/1028523.1028541.

G. Irving, E. Guendelman, F. Losasso, and R. Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Trans. Graph.*, 25(3): 805–811, 2006. ISSN 0730-0301. doi: 10.1145/1141911.1141959.

G. Irving, C. Schroeder, and R. Fedkiw. Volume Conserving Finite Element Simulations of Deformable Models. *ACM Transactions on Graphics (SIGGRAPH Proc.)*, 26(3), 2007.

D. James and D. Pai. ArtDefo: accurate real time deformable objects. In *Proceedings of SIGGRAPH 99*, pages 65–72, 1999. doi: 10.1145/311535.311542.

L. Jerabkova, G. Bousquet, S. Barbier, F. Faure, and J. Allard. Volumetric modeling and interactive cutting of deformable bodies. *Progress in Biophysics and Molecular Biology*, 103(2-3):217–224, Dec. 2010. doi: 10.1016/j.pbiomolbio.2010.09.012. Special Issue on Biomechanical Modelling of Soft Tissue Motion.

L. Jeřábková and T. Kuhlen. Stable Cutting of Deformable Objects in Virtual Environments Using XFEM. *IEEE Comput. Graph. Appl.*, 29(2):61–71, 2009. ISSN 0272-1716. doi: 10.1109/MCG.2009.32.

P. Joshi, M. Meyer, T. DeRose, B. Green, and T. Sanocki. Harmonic Coordinates for Character Articulation. *ACM Trans. Graph.*, 26(3), July 2007. ISSN 0730-0301. doi: 10.1145/1276377.1276466.

P. Kaufmann, S. Martin, M. Botsch, E. Grinspun, and M. Gross. Enrichment Textures for Detailed Cutting of Shells. *ACM Trans. Graph.*, 28(3):50:1–50:10, 2009a.

P. Kaufmann, S. Martin, M. Botsch, and M. Gross. Flexible Simulation of Deformable Models Using Discontinuous Galerkin FEM. *Graph. Models*, 71(4):153–167, July 2009b. ISSN 1524-0703. doi: 10.1016/j.gmod.2009.02.002.

L. Kavan, S. Collins, J. Žára, and C. O'Sullivan. Geometric Skinning with Approximate Dual Quaternion Blending. *ACM Trans. Graph.*, 27(4):105:1–105:23, Nov. 2008. ISSN 0730-0301. doi: 10.1145/1409625.1409627.

L. Kharevych, P. Mullen, H. Owhadi, and M. Desbrun. Numerical Coarsening of Inhomogeneous Elastic Materials. *ACM Trans. Graph.*, 28(3):51:1–51:8, July 2009. ISSN 0730-0301. doi: 10.1145/1531326.1531357.

Khronos Group, 2017. WebGL Specification (Editors Draft). WebGL Specification (Editors Draft), May 2017. URL `https://www.khronos.org/registry/webgl/specs/latest/1.0`.

J. Kim and N. Pollard. Fast simulation of skeleton-driven deformable body characters. *ACM Transactions on Graphics (TOG)*, 30(5):121, 2011.

J. Kim, C. Choi, S. De, and M. A. Srinivasan. Virtual surgery simulation for medical training using multi-resolution organ models. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 3(2):149–158, 2007. ISSN 1478-596X. doi: 10.1002/rcs.140.

L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature Sensitive Surface Extraction from Volume Data. SIGGRAPH '01, pages 57–66, 2001.

F. Labelle and J. Shewchuk. Isosurface Stuffing: Fast Tetrahedral Meshes with Good Dihedral Angles. *ACM TOG*, 26(3), 2007.

D. Li, S. Sueda, D. R. Neog, and D. K. Pai. Thin Skin Elastodynamics. *ACM Trans. Graph.*, 32(4):49:1–49:10, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2462008.

A. Lindblad and G. Turkiyyah. A Physically-based Framework for Real-time Haptic Cutting and Interaction with 3D Continuum Models. SPM '07, pages 421–429. ACM, 2007. ISBN 978-1-59593-666-0. doi: 10.1145/1236246.1236307.

H. Liu, N. Mitchell, M. Aanjaneya, and E. Sifakis. A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Transactions on Graphics*, 35(6):1–12, nov 2016. doi: 10.1145/2980179.2982430. URL `https://doi.org/10.1145/2980179.2982430`.

F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 457–462, New York, NY, USA, 2004. ACM. doi: 10.1145/1186562.1015745.

F. Losasso, T. Shinar, A. Selle, and R. Fedkiw. Multiple interacting liquids. *ACM Trans. Graph.*, 25(3):812–819, 2006. ISSN 0730-0301. doi: 10.1145/1141911.1141960.

M. Marchal, J. Allard, C. Duriez, and S. Cotin. Towards a Framework for Assessing Deformable Models in Medical Simulation. In P. J. E. Fernando Bello, editor, *ISBMS '08*, volume 5104 of *Lecture Notes in Computer Science*, pages 176–184. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70521-5. doi: 10.1007/978-3-540-70521-5_19.

A. McAdams, E. Sifakis, and J. Teran. A parallel multigrid Poisson solver for fluids simulation on large grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 65–74, 2010.

A. McAdams, Y. Zhu, A. Selle, M. Empey, R. Tamstorf, J. Teran, and E. Sifakis. Efficient Elasticity for Character Skinning with Contact and Collisions. *ACM Trans. Graph.*, 30 (4):37:1–37:12, July 2011. ISSN 0730-0301. doi: 10.1145/2010324.1964932.

J. McCarthy. *Plastic Surgery General Principles.* Plastic Surgery: General Principles. Saunders, 1990. ISBN 9780721625423.

C. Mendoza and C. Laugier. Simulating Soft Tissue Cutting using Finite Element Models. volume 1 of *ICRA '03*, pages 1109–1114. IEEE, 2003.

N. Mitchell, C. Cutting, and E. Sifakis. GRIDiron: An Interactive Authoring and Cognitive Training Foundation for Reconstructive Plastic Surgery Procedures. *ACM Trans. Graph.*, July 2015. doi: 10.1145/2766918.

N. Moës, J. Dolbow, and T. Belytschko. A finite element method for crack growth without remeshing. *IJNME*, 46:131–150, 1999.

N. Molino, Z. Bao, and R. Fedkiw. A Virtual Node Algorithm for Changing Mesh Topology During Simulation. *ACM Trans. Graph.*, 23(3):385–392, Aug. 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015734.

M. Müller, M. Teschner, and M. Gross. Physically-Based simulation of Objects Represented by Surface Meshes. In *Proc. Computer Graphics International*, pages 156–165, June 2004.

M. Muller, M. Teschner, and M. Gross. Physically-Based Simulation of Objects Represented by Surface Meshes. CGI '04, pages 26–33, 2004.

M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007. doi: 10.1016/j.jvcir.2007.01.005.

K. Museth. DB+Grid: A Novel Dynamic Blocked Grid for Sparse High-resolution Volumes and Level Sets. SIGGRAPH '11, 2011. ISBN 978-1-4503-0974-5. doi: 10.1145/2037826.2037894. URL `http://doi.acm.org/10.1145/2037826.2037894`.

K. Museth. VDB: High-resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.*, 32(3):27:1–27:22, July 2013. ISSN 0730-0301. doi: 10.1145/2487228.2487235. URL `http://doi.acm.org/10.1145/2487228.2487235`.

K. Museth, D. Breen, R. Whitaker, and A. Barr. Level Set Surface Editing Operators. In *ACM TOG*, pages 330–338, 2002.

M. Nesme, Y. Payan, and F. Faure. Animating Shapes at Arbitrary Resolution with Non-Uniform Stiffness. In *Eurographics Workshop in Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, Madrid, nov 2006. URL `http://www-evasion.imag.fr/Publications/2006/NPF06`.

M. Nesme, P. Kry, L. Jeřábková, and F. Faure. Preserving topology and elasticity for embedded deformable models. In *ACM Transactions on Graphics (SIGGRAPH Proceedings)*, volume 28, page 52, 2009.

M. B. Nielsen and K. Museth. Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets. *J. Sci. Comput.*, 26(3):261–299, Mar. 2006. ISSN 0885-7474. doi: 10.1007/s10915-005-9062-8. URL `http://dx.doi.org/10.1007/s10915-005-9062-8`.

H.-W. Nienhuys and A. F. van der Stappen. A Surgery Simulation Supporting Cuts and Finite Element Deformation. MICCAI '01, pages 145–152. Springer, 2001.

Node.js Foundation. NodeJS. `http://nodejs.org`, 2017.

J. O'Brien and J. Hodgins. Graphical Modeling and Animation of Brittle Fracture. In *Proc. of SIGGRAPH 1999*, pages 137–146, 1999. doi: 10.1145/311535.311550.

S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.

S. Osher and J. Sethian. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *J. Comput. Phys.*, 79:12–49, 1988.

E. G. Parker and J. F. O'Brien. Real-Time Deformation and Fracture in a Game Environment. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 156–166, Aug. 2009. URL `http://graphics.berkeley.edu/papers/Parker-RTD-2009-08`.

T. Patterson, N. Mitchell, and E. Sifakis. Simulation of Complex Nonlinear Elastic Bodies Using Lattice Deformers. *ACM Trans. Graph.*, 31(6):197:1–197:10, Nov. 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366216.

C. G. Petra, O. Schenk, and M. Anitescu. Real-time stochastic optimization of complex energy systems on high-performance computers. *IEEE Computing in Science & Engineering*, 16 (5):32–42, 2014a.

C. G. Petra, O. Schenk, M. Lubin, and K. Gärtner. An augmented incomplete factorization approach for computing the schur complement in stochastic optimization. *SIAM Journal on Scientific Computing*, 36(2):C139–C162, 2014b.

S. D. Pieper, D. R. Laub Jr, and J. M. Rosen. A Finite-Element Facial Model for Simulating Plastic Surgery. *Plastic and Reconstructive Surgery*, 96(5):1100–1105, 1995.

A. Quarteroni and A. Valli. *Domain decomposition methods for partial differential equations*, volume 10. Clarendon Press, 1999.

L. Rising, editor. *The Patterns Handbook: Techniques, Strategies, and Applications*. SIGS, 1998. ISBN 0521648181.

A. Rivers and D. James. FastLSM: Fast lattice shape matching for robust real-time deformation. *ACM Trans. on Graphics (SIGGRAPH Proc.)*, 26(3), 2007. doi: 10.1145/1275808.1276480.

R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis. SPGrid: A Sparse Paged Grid Structure Applied to Adaptive Smoke Simulation. *ACM Trans. Graph.*, 33(6):205:1–205:12, 2014a.

R. Setaluri, Y. Wang, N. Mitchell, L. Kavan, and E. Sifakis. Fast Grid-Based Nonlinear Elasticity for 2D Deformations. The Eurographics Association, 2014b.

J. A. Sethian. Fast Marching Methods. *SIAM Review*, 41:199–235, 1998.

E. Sifakis and J. Barbic. FEM Simulation of 3D Deformable Solids: A Practitioner's Guide to Theory, Discretization and Model Reduction. In *ACM SIG. 2012 Courses*, SIGGRAPH '12, pages 20:1–20:50. ACM, 2012. ISBN 978-1-4503-1678-1. doi: 10.1145/2343483.2343501.

E. Sifakis, K. G. Der, and R. Fedkiw. Arbitrary Cutting of Deformable Tetrahedralized Objects. SCA '07, pages 73–80. Eurographics Association, 2007. ISBN 978-1-59593-624-0. URL http://dl.acm.org/citation.cfm?id=1272690.1272701.

Simbionix USA Corporation. Laparoscopic Simulator - LAP Mentor Simbionix. http://simbionix.com/simulators/lap-mentor, 2002–2014a.

Simbionix USA Corporation. Gastrointestinal Simulator - GI Mentor Simbionix. http://simbionix.com/simulators/gi-bronch-gi-mentor, 2002–2014b.

F. Sin, D. Schroeder, and J. Barbic. Vega: Non-Linear FEM Deformable Object Simulator. *Comput. Graph. Forum*, 32(1):36–48, 2013. doi: 10.1111/j.1467-8659.2012.03230.x.

D. Steinemann, M. Harders, M. Gross, and G. Szekely. Hybrid cutting of deformable solids. In *IEEE Virtual Reality Conference (VR 2006)*, pages 35–42, March 2006. doi: 10.1109/VR.2006.74.

S. Sueda, A. Kaufman, and D. K. Pai. Musculotendon Simulation for Hand Animation. *ACM Trans. Graph.*, 27(3):83:1–83:8, Aug. 2008. ISSN 0730-0301. doi: 10.1145/1360612.1360682.

J. Teran, S. Blemker, V. N. T. Hing, and R. Fedkiw. Finite Volume Methods for the Simulation of Skeletal Muscle. SCA '03, pages 68–74, 2003. ISBN 1-58113-659-5. URL http://dl.acm.org/citation.cfm?id=846276.846285.

J. Teran, E. Sifakis, S. S. Blemker, V. Ng-Thow-Hing, C. Lau, and R. Fedkiw. Creating and Simulating Skeletal Muscle from the Visible Human Data Set. *Visualization and Computer Graphics, IEEE Transactions on*, 11(3):317–328, 2005a.

J. Teran, E. Sifakis, G. Irving, and R. Fedkiw. Robust Quasistatic Finite Elements and Flesh Simulation. *Proc. of the 2005 ACM SIGGRAPH/Eurographics Symp. on Comput. Anim.*, pages 181–190, 2005b.

D. Terzopoulos and K. Fleischer. Deformable models. *The Visual Computer*, 4(6):306–331, 1988.

D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. *Computer Graphics (Proc. SIGGRAPH 87)*, 21(4):205–214, 1987.

R. Vaillant, L. Barthe, G. Guennebaud, M.-P. Cani, D. Rohmer, B. Wyvill, O. Gourmel, and M. Paulin. Implicit Skinning: Real-time Skin Deformation with Contact Modeling. *ACM Trans. Graph.*, 32(4):125:1–125:12, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461960.

R. Vaillant, G. Guennebaud, L. Barthe, B. Wyvill, and M.-P. Cani. Robust Iso-surface Tracking for Interactive Character Skinning. *ACM TOG*, 33(6):189:1–189:11, 2014.

B. Wang, F. Faure, and D. K. Pai. Adaptive Image-based Intersection Volume. *ACM Trans. Graph.*, 31(4):97:1–97:9, 2012.

H. Wang. A Chebyshev Semi-iterative Approach for Accelerating Projective and Position-based Dynamics. *ACM Trans. Graph.*, 34(6):246:1–246:9, Oct. 2015. ISSN 0730-0301. doi: 10.1145/2816795.2818063.

X. C. Wang and C. Phillips. Multi-weight Enveloping: Least-squares Approximation Techniques for Skin Animation. SCA '02, pages 129–138. ACM, 2002. ISBN 1-58113-573-4. doi: 10.1145/545261.545283.

C. Wojtan and G. Turk. Fast Viscoelastic Behavior with Thin Features. *ACM Trans. Graph.*, 27(3):47:1–47:8, Aug. 2008. ISSN 0730-0301. doi: 10.1145/1360612.1360646.

wxWidgets. wxWidgets - A Cross Platform GUI Library. `https://www.wxwidgets.org`, 2017.

H. Xu and J. Barbič. Example-Based Damping Design. *ACM Trans. on Graphics (SIGGRAPH 2017)*, 36(4), 2017.

Z. Yuan, Y. Yu, and W. Wang. Object-space Multiphase Implicit Functions. *ACM TOG*, 31 (4):114:1–114:10, 2012.

H.-K. Zhao, S. Osher, and R. Fedkiw. Fast Surface Reconstruction Using the Level Set Method. VLSM '01, pages 194–202, 2001.

Y. Zhao and J. Barbič. Interactive Authoring of Simulation-Ready Plants. *ACM Trans. on Graphics (SIGGRAPH 2013)*, 32(4):84:1–84:12, 2013.

W. Zheng, J.-H. Yong, and J.-C. Paul. Simulation of Bubbles. SCA '06, pages 325–333, 2006.

Y. Zhu, E. Sifakis, J. Teran, and A. Brandt. An Efficient Multigrid Method for the Simulation of High-resolution Elastic Solids. *ACM Trans. Graph.*, 29(2):16:1–16:18, Apr. 2010. ISSN 0730-0301. doi: 10.1145/1731047.1731054.