GEOMETRICALLY AND PHYSICALLY BASED MODELING FOR DEFORMABLE OBJECTS SIMULATING SOFT TISSUE INTERACTIONS IN SURGERY ENVIRONMENTS

By

SUKITTI PUNAK

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2007

To my soul mate, Pawinee; and to our kind and supportive parents.

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Dr. Jörg Peters, for his unending patience, guidance, and encouragement. I would also like to thank the other members of my committee; Dr. Paul Fishwick, Dr. Anand Rangarajan, Dr. Alper Üngör, and Dr. Carl Crane from department of mechanical and aerospace engineering (MAE). I thank Dr. Tim Davis for substituting Dr. Fishwick on my defense. I want to thank my colleagues in and out the SurfLab who all became great friends. I thank department of computer and information science and engineering (CISE) for its financial support. I also thank Sergei N. Kurenov for giving me the opportunity to work with pay on simulation projects for the Department of Surgery at UF. Without a doubt, Peter J. Dobbins, a CISE lecturer, helped me to develop my college teaching skills during my TA duties for his classes over the years. I am indebted to him for that.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, NOMENCLATURE, OR ABBREVIATIONS

3D Cartesian grid	A regular, axis aligned sampling of data in three dimensions where its topology is a unit cube (structured) lattice with clearly defined neighborhood voxels.
3D regular grid	A regular, axis aligned sampling of data in three dimensions where its topology is a rectangular (structured) lattice with clearly defined neighborhood voxels.
FFD	Free-form deformation
FEM	Finite element method
GPU	Graphics processing/processor unit
Home spring	A spring that connects a particle to its initial or rest position
Laparoscopic surgery	Laparoscopy or minimally invasive surgery (MIS) is a surgical technique that cuts small openings or holes in a patient's abdomen to pass surgical instruments to operate inside the patient.
Mass point	A particle, that has a point geometry and a finite mass, represents a node in a mass-spring system.
Mass-spring system	A particle system in which each node in the system is connected to some other nodes by springs.
MIS	Minimally invasive surgery (see laparoscopic surgery)
MSS	Mass-spring system
ODE	Ordinary differential equation
Plausible behavior	A behavior that could happen. It does not have to be realistic, but it appears worthy of belief.
Shader	A set of software instructions that is executed on GPU, usually for rendering-related computations.
Structured grid	A structured grid (such as Cartesian, regular, rectilinear, curvilinear, and block-structured grid) consists of hexahedral cells, in which connectivity is implicit ([2] on page 75).

Unstructured grid	An unstructured grid can be composed of, for example, tetrahedra, hexahedra, prisms, or pyramids, in which connectivity must be encoded explicitly ([2] on page 75).
Voxel	A volume element with its center serves as a grid point of a 3D grid ([3] on page 18).
Voxelization	The process of converting a geometric representation (polygonal surface) of a synthetic object into a set of voxels arranged in a volume data set (3D regular grid).

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

GEOMETRICALLY AND PHYSICALLY BASED MODELING
FOR DEFORMABLE OBJECTS SIMULATING SOFT TISSUE INTERACTIONS
IN SURGERY ENVIRONMENTS

By

Sukitti Punak

December 2007

Chair: Jörg Peters
Major: Computer Engineering

This dissertation contributes a method for modeling deformable objects that leverages computer graphics techniques and state-of-the-art hardware to simulate and visualize deformations, including topological transformations like cutting, in real time. The resulting models are suitable for interactive surgical simulation for illustration and teaching purposes. The approach has been tested, in particular, by modeling responsive fatty tissue. To support force feedback as well as visual feedback, the deformable object consists of three closely coupled representations for dynamic response, collision detection, and visualization. All three representations are implemented on the Graphics Processing Unit, rather than on the CPU, to leverage parallelism for speed.

CHAPTER 1
INTRODUCTION

This chapter discusses the importance of modeling deformable objects for surgery illustration and argues for simulating plausible behavior instead of realistic behavior.

1.1 How Accurate should Deformable Objects be?

The accurate mathematical modeling of deformable objects is central to many areas of engineering and science. For example in materials science, aerospace, and civil engineering, the strength and elasticity of deformable objects plays a crucial role. In graphics and haptic, force-feedback applications such as real-time computer animation and simulation, the focus is instead on fast performance and high visual quality where the computational resources are restricted. Rather than physical accuracy, visual and haptic plausibility [4], [5], [6] are considered acceptable.

Plausibility is off-hand a user-dependent, subjective notion. In the context of this dissertation, it is made concrete as constraints on robustness, flexibility, geometric resolution and bounds on physical accuracy within a particular application scenario, namely surgical procedure illustration. Besides reducing computational complexity, a second major reason for settling for a plausible model is that often an accurate model does not exist. This is due to the nonlinear nature of the deformable object lack of data or variability of human perception (see Section 1.2 (a)–(e)). Thirdly, in the surgical illustration and teaching environment [7], [8], physical and visual properties need to be user-adjustable, on-the-fly. They may even be chosen non-realistically to overstate a point. In the following, we therefore propose a framework for plausible modeling deformable objects optimizing in order of importance:

- Speed
- Robustness (stability and consistency)
- Flexibility (topology and material property changes)
- Geometry resolution
- Physical accuracy
- Memory usage

1.2 Deformable Objects in Surgical Simulation

Deformable objects are important in surgery illustration. The goal of surgery illustration is to provide a convenient, repeatable, and inexpensive supplement to traditional textbook learning and on-the-job "See one, do one, teach one" surgical training ([9] on page 146). Using virtual deformable objects and force-feedback devices for surgery illustration is especially well-matched with laparoscopic surgery, also called minimally invasive surgery (MIS) [10], [11], where the surgeon views the operating arena only via video and interacts via instruments without touching the surgical area by hand (Figure 1-1). A typical surgical illustration environment has the following components:

- Anatomical objects, e.g., liver, gallbladder, and blood vessels;
- Haptic device representing a surgery tool for user interaction;
- Visualization of the operation site, i.e., an endoscope for laparoscopic surgery.

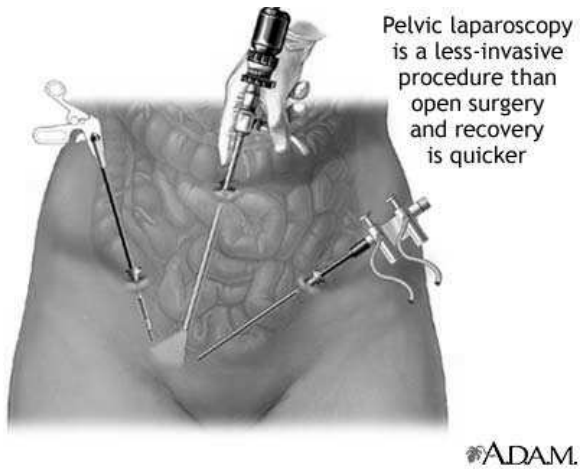


Figure 1-1. Pelvic laparoscopy (reprinted from [12], freely accessible from the Internet)

To be an effective alternative training system, the simulator has to be able to interact with the user in real time or at least at an acceptable interactive rate. Such an interactive simulation system needs to meet two main requirements. It needs to be fast enough to generate at least 10 (ideally 60) frames per second for visualization and at least 300 (ideally 1000) updates per second for haptic sensation [11].

However, speed is not the only requirement. The user should be allowed to drastically modify the topology, for example by cutting. Thus, stability, robustness, and consistency are just as important as the frame rate.

In particular, the intended level of accuracy of the model are influenced by the following observations.

a. **The Material Complexity of the Soft Tissue**
The mechanical behavior of soft tissues, such as the kidney, liver, stomach, and blood vessels, is extremely complex. The stress-strain or load-displacement relationship is highly nonlinear which means tissue becomes stiffer as it stretches. Tissue is also viscoelastic (i.e., the stress-strain relationship depends on the rate of deformation), heterogeneous (varies through the tissue volume), and anisotropic (varies with direction) [13]. Due to viscoelastic effects, as tissue is stretched repeatedly less load is typically necessary to produce the same elongation. With soft tissues such as skin, unlike typical engineering materials such as metals, very large deformations are possible. Hence, in the case of soft tissue modeling for a realistic surgical simulation, the simulator has to deal with large deformations of anisotropic, heterogeneous, and nonlinear viscoelastic materials.

b. **The Interaction**
The soft tissue in a surgery environment will interact with other tissues and surgical instruments. In particular, the deformable object is subject to being touched, pulled, cut and sutured by surgical tools.

c. **The Level of Accuracy**
Human ability to detect differences between objects is poor [14]. Research is necessary to determine what visual and haptic cues surgeons use, and how sensitive they are, for example, in detecting changes in tissue due to damage, hidden structures, or disease [14].

d. **The Complexity of the Computational Model**
Currently, in the scope of a surgical simulator, it is not possible to model the biomechanical complexity of living soft tissue realistically [15]. Therefore, the model has to be simplified in order to reduce the implementation complexity and to optimize computational efficiency. For example, to simplify viscoelasticity, Fung [13] proposed a model called the pseudo-elastic model that uses preconditioning of the tissue while using only two separate elastic materials, one during loading and another during unloading.

e. **The Analytical and Empirical Knowledge about the Soft Tissue**
With currently available methods, it is almost impossible to measure material parameters of human tissue completely and accurately. Therefore, even in the case

that an accurate model was available, the accuracy would still be controlled by the poor precision of the material parameters.

Plausible deformable objects for surgical illustration therefore follow a quote by Albert Einstein, "Everything should be made as simple as possible, but not one bit simpler."

1.3 Deformable Object and Simulation

In order to simulate the reaction of a deformable object in real time with plausible deformation, the simulation model must run fast enough and show plausible deformation with satisfactory rendering quality. Therefore, we chose a mixed, volumetric-and-surface representation consisting of three closely coupled components for dynamic response, collision detection, and visualization.

The dynamic response is based on a 3D mass-spring system that is simulated entirely on a GPU. It calculates the next position of each mass point in the mass-spring system of the deformable object. The dynamic response includes internal and external forces on the object. The collision detection adjusts the position of each mass point so that deformable objects and other objects do not penetrate another during the simulation environment. The computation is based on collision test and penetration distance between objects. Finally, the visualization is for rendering the deformable object.

With the latest graphics cards, we use GPU computations to accelerate the simulation by computing expensive steps in parallelism. Our deformable object runs in real time with a geometric resolution of a 40x40x40 grid for local haptic interaction. We tested the approach, by creating and tuning a deformable object that mimics soft tissue, in particular fatty tissue into which organs and vessels are embedded. The model supports changes of topology and material properties.

1.4 Contributions

The main contribution of this dissertation is a method for modeling deformable objects that is suitable for GPU implementation and thereby executes fast enough for

interactive surgical illustration and force feedback. In particular, it models responsive fatty tissue. The deformable object uses computer graphics techniques to simulate and visualize deformations, including topological transformations like cutting, in real time. To support force feedback as well as visual feedback, the deformable object consists of three closely coupled representations: dynamic response, collision detection, and visualization.

1.5 Outline

The dissertation is organized as follows. Chapter 2 reviews background and literature of modeling deformable objects. Chapter 3 introduces our framework for deformable object modeling, i.e., the dynamic response, collision detection and visualization representations. Chapter 4 discusses the creation of the deformable object simulator for our deformable object modeling. In this chapter, the implementation of the deformable object simulator using GPU shaders is explained. Chapter 5 describes application in the haptic-enabled toolkit for illustration of procedures in surgery (TIPS) [7], [8]. Chapter 6 concludes the dissertation.

CHAPTER 2
BACKGROUND

Three methods for modeling deformations are discussed in this chapter: free-form deformation (FFD), the mass-spring system (MSS), and the finite element method (FEM). Only the MSS and the FEM are suitable for modeling soft tissues. In Section 2.6, we argue why we opted for a mass-spring system to represent the physical component of our model. The advantages and disadvantages of the MSS over of the FEM are summarized in Tables 2-1 and 2-2. Table 2-3 compares the MSS to the FEM based on the order of importance listed on page 13.

2.1 Deformable Model Frameworks

There are two major approaches for deformable object simulation: namely, the geometrically based and the physically based approaches. Geometry based approaches such as free-form deformation (FFD) employ purely geometric techniques to model deformation. Physically based approaches employ techniques with parameters that simulate physical properties of the deformable object. Methods that are adapted or adopted from the mass-spring system (MSS) and the finite element method (FEM) and its variations, such as the boundary element method (BEM), belong to physically based approaches.

FFD, the MSS, and the FEM are the three most widely used methods for modeling deformable objects [10]:

- **Free-form deformation**. These techniques originated from computer graphics techniques such as spline and superquadric models. Models based on these techniques, however, are not easily modified to model soft tissue deformation with interaction.

- **The mass-spring system**. Mass-spring systems are simple, physically inspired approximations. However, adjusting parameter values of the system to achieve a desired behavior of the model is not easy, and model accuracy and stability depend highly on the parameter values such as spring stiffness constants.

- **The finite element method**. Based on continuum mechanics, the finite element method is one of the most accurate methods to model deformable objects. However,

for real-time applications its computational complexity restricts its application to either a geometrically complex model with simplified properties resulting in less accuracy or a simple geometric model resulting in less visual quality.

Free-form deformation are briefly presented. The mass-spring system and the finite element method are the main focus in this chapter. We should also mention early contributions to the simulation of deformable objects in computer graphics include Terzopoulos et al. [16], [17], [18], [19] who defined a physically based class of deformable objects (curves, surfaces, and solids).

2.2 Free-Form Deformation

Free-form deformation (FFD) is a general method for deforming objects that provides a higher and more powerful level of control than adjusting individual control points [20]. Basically, FFD changes the shape of an object by deforming the space in which the object is contained. The term free-form deformation, introduced by Sederberg and Perry [21], is a generalized Barr's approach [22] by embedding an object in a lattice of grid points of some standard geometric shape, such as a cube or cylinder. The basic FFD technique has been extended by several others: including extended FFD (EFFD) [23] for modeling; volume-preserving FFD [24], [25] with volume preservation property; dynamic FFD (DFFD) [26] with physical properties for animation purposes; and discontinuous FFD (DFFD) [27], [28] for incision simulation. These techniques are simple and fast since no physical considerations are directly involved. However, it is unclear how to incorporate physical properties to control the object's deformation reacting to forces. Hence, these techniques are hardly employed in surgical simulation.

2.3 Mass-Spring System

Non-physical methods, such as the FFD, for modeling deformation are limited by the expertise and patience of the user to animate and store the movement in advance for a visually plausible simulation. On the other hand, the mass-spring system consists of mass points with explicitly defined connectivity. It is a physically based technique that has been used widely and effectively for modeling deformable objects, such as clothes

[29], [30], [31], [32], ropes [33], [34], skin [35], muscles [36], and faces [37]. An object is modeled as a collection of mass points connected by springs in a lattice structure. The spring forces often obey Hooke's law, stating that "The power of any springy body is in the same proportion with the extension." [38]. However, nonlinear springs can be used to model tissues such as human skin that exhibit inelastic behavior.

In a dynamic system, Newton's second law of motion of a single mass point or particle is [20], [10], [39]:

$$m_i\ddot{\mathbf{x}}_i + c\dot{\mathbf{x}}_i + \sum_{j\in N_i} \mathbf{f}_{ij} = \mathbf{f}_i^{ext} \tag{2–1}$$

m_i is the constant mass of point i and $\mathbf{x}_i \in \mathbb{R}^3$ is its position. The acceleration and velocity of the point are represented by $\ddot{\mathbf{x}}_i$ and $\dot{\mathbf{x}}_i$, respectively. The second left-hand term is a velocity-dependent damping force where c is the damping constant. The third left-hand term is the summation of each $\mathbf{f}_{ij}$ which is the internal force exerting on mass point i by the spring connecting mass point i to mass point j. $\mathbf{f}_i^{ext}$ in the right-hand term is the sum of external forces, such as gravity or contact forces, exerting on mass point i.

The equations of motion for the entire system are assembled from the motions of all of the mass points composing the object. Concatenating the position vectors of the N individual masses into a single $3N$-dimensional position vector $\mathtt{x}$ gives:

$$\mathtt{M}\ddot{\mathtt{x}} + \mathtt{C}\dot{\mathtt{x}} + \mathtt{K}\mathtt{x} = \mathtt{f} \tag{2–2}$$

where $\mathtt{M}$, $\mathtt{C}$, and $\mathtt{K}$ are $3N \times 3N$ mass, damping, and stiffness matrices, respectively. The system is solved forward through time by changing it to the system of first-order differential equations:

$$\begin{aligned} \dot{\mathtt{v}} &= \mathtt{M}^{-1}(-\mathtt{C}\mathtt{v} - \mathtt{K}\mathtt{x} + \mathtt{f}) \\ \dot{\mathtt{x}} &= \mathtt{v} \end{aligned} \tag{2–3}$$

where $\mathtt{v}$ is the velocity vector of the system of mass points. A numerical integration technique [40], [41], such as implicit or explicit Euler integration method, is used to compute $\mathtt{x}$ and $\mathtt{v}$ as functions of time.

In a quasi-static system, the dynamic part of the dynamic equation is neglected ($\ddot{\mathtt{x}} = \dot{\mathtt{x}} = 0$) and the equation becomes:

$$\mathtt{Kx} = \mathtt{f} \tag{2–4}$$

2.3.1 Related Work on the Mass-Spring System

Keeve et al. [42] used a layered mass-spring system to model a patient's face for craniofacial surgery planning. The facial tissue was assumed to be homogeneous; We also use the same assumption for our deformable object. The non-linear stress-strain was modeled by using a biphasic spring constant. An additional force was added to preserve the volume of the tissue. The simulation ran offline to show and predict the preoperative and postoperative appearance of the patient after a craniofacial surgical procedure.

Nedel and Thalmann [36] used the mass-spring system to animate skeletal muscle deformations. They aligned springs with action lines of the muscle and used another set of springs to produce the muscle shape. A special spring was also added to account for the bending and twisting of the muscle surface. However, the method made no attempt to cut the modeled muscle. It simulated only muscle deformation which ran in real-time. The concept of multiple springs is also used in our deformable object.

Kühnapfel et al. [10] used a volumetric mass-spring system to model deformable objects, such as blood vessels and deformable tissue, in their minimally invasive surgery simulator called KISMET. Their KISMET also included deformable objects modeled by the finite element method. The paper did not described how cutting was implemented on the deformable objects and whether the cut could be performed only on the mass-spring based model or the finite element based model or both. Our deformable object is also

based on a volumetric mass-spring system, which is actually the 3D regular grid of the mass-spring system.

Bourguignon and Cani [43] derived an anisotropic model based on the mass-spring system. An object created from this model was either tiled with tetrahedral or hexahedral meshes. The axes of anisotropic directions were defined and set by the user. All internal forces were calculated from these axes instead of along the mesh edges. Our deformable object is also an anisotropic model, since the spring's rest length and/or stiffness for each x, y, and z direction can be set independently from one another.

Webster et al. [44] used an implicit prediction solver to solve the first order differential equations of a mass-spring model representing only the surface of an object. A precomputation of the solution of the linear system was required in order to speed up the computation. The simulated model had simple computation and responded well when interacting with a haptic device. However, a change in the topology of the object, such as by a cut, would require solving a very different linear system. Therefore, the method was not suitable for modeling a deformable object being cut.

Choi et al. define force propagation model (FPM) [45], which limited deformation of soft tissues to a local region of the model defined by the depth of force propagation in order to speed up the computation time. The FPM became a typical mass-spring model if the force propagation reached all of the nodes in the object. Later they tried to formulate the FPM by comparing and adjusting the spring stiffness and damping constant of each spring to emulate the deformation of a precomputed finite element model [46]. However, the deformation of the model with this technique could only emulate the deformation of the precomputed finite element model. It was unclear how the model would be deformed with different forces than the ones encountered in the setup phase.

The idea of using material properties to set spring properties was also used by Bruyns et al [47]. They measured and used physically derived stiffness constants of rat tissue from a finite element analysis to set the stiffness parameters of springs with the 2D Young's

Modulus for surface-based representations and the three-dimensional (3D) Young's Modulus for volumetric representations. The simulated object showed a more realistic deformation than the one with a uniform spring constant. However, no attempt on cutting the model was implemented by the method.

2.3.2 Mass-Spring Systems on the GPU

The GPU is a data-parallel processor. It can process a set of data in parallel. Since solving the ODE for each mass point in the mass-spring system with an explicit integration method can be done independently, the GPU can be used as the solver. A 3D Mass-spring system on the GPU was first implemented by Georgii and Westermann [39] followed by an analysis for spring-based computation vs. mass-point-based computation on the GPU for cloth simulation [48]. Their results showed that spring-based computation was slightly faster, e.g., 128^2 took 0.72 and 0.74 ms for spring-based and mass-point-based computation, respectively. Later they improved their system to include a multigrid implicit solver [49], and heterogeneous material [50]. Their 3D data set was a 3D unstructured grid consisting of tetrahedra. The simulated deformable object ran in real time or at an interactive rate depend on the resolution of the grid. The method did not suitable for topology changes, since a topology change requires the data on the unstructured grid to be changed. The change of the data structure would require the data stored on the GPU's texture memory to be rearranged. This operation would take as much time as the preprocessing time for loading the original unstructured grid to the GPU's memory. Hence, the method did not use for simulating a soft tissue that would be cut by a surgical tool.

Mosegaard and Sørensen [51], [52], [53] used GPU computation for simulating the deformations of a human heart during an open surgery. Their mass-spring system was a 3D Cartesian grid of a human heart. The data for the mass-spring system were obtained from subsampling the CT dataset of the heart. For visualization, they used the marching cubes algorithm to reconstruct a surface triangular mesh from the original CT dataset.

Each vertex of the mesh was mapped as an offset vector from the nearest mass-spring node. A deformation of the mass-spring system would result in a deformation of the surface mesh. The simulated object could be probed and cut by a surgical tool. However, the cut surface would generate a hole, since the cut vertex was split into two vertices that were separated apart and the object appeared hollow inside. This was fine for some parts of the human heart, since the heart had a lot of cavities. The other drawback was that the method relied on the implicit connectivity of the 3D Cartesian grid. To simulate topology changes, a GPU texture had to be set up for storing the topology changes. Therefore, each topology change would add a computation cost to the simulation. The lack of explicit connectivity and the fixed cell size from using the 3D Cartesian grid made material property adjustments less flexible, i.e., the method could not have different spring rest lengths for different axis directions.

To provide flexibility and steady performance (computation cost), our deformable object is based on the 3D regular grid and uses explicit connectivity. The explicit connectivity adds extra computation cost into the system, but it provides more flexible in connectivity setup. Besides, any topology change will not add extra cost to the simulation. To avoid creating an object that appears hollow after cut, we use the marching cubes algorithm to generate the object's surface in real time.

2.4 Finite Elements

The mass-spring system starts with discretizing a deformable object into mass points connected by springs. A more accurate physical model assumes the deformable object as a continuum solid [54], i.e., a deformable body with mass and energies distributed every where in the body that can support shear forces.

For tissue modeling, the focus is in modeling the displacement. The objective is to consider the equilibrium of the deformable object acted on by external forces. The object's deformation is a function of these acting forces and the object's material properties. Using the principle of minimum potential energy, the object reaches equilibrium when its

potential energy is at a minimum. The total potential energy of the object is denoted by Π, and is given by [20]:

$$\Pi = \mathrm{U} - \mathrm{W} \tag{2–5}$$

where U is the total strain energy of the object, and W is the potential energy of (or work done by) the external loads [55] (caused by concentrated, surface, and body forces) acting on the object.

To determine the equilibrium shape of the object, both U and W are expressed in terms of the object's deformation, which is represented by a function of the material displacement over the object. The total potential energy reaches a minimum when the derivative of Π with respect to the material displacement function is zero. This results in a continuous differential equilibrium equation that must be solved for the object's displacement. Since it is not always possible to find a closed-form analytic solution of this differential equation, a number of numerical methods, such as the finite element method (FEM), are used to approximate the object's deformation. Basic steps in using the FEM to compute the object's deformation [20] are as follows:

1. Derive an equilibrium equation from the potential energy equation in terms of material displacement of the continuum.

2. Choose suitable finite elements and associating interpolation functions for the specific problem. Divide the object into elements.

3. For each element, define the components of the equilibrium equation in terms of the interpolation functions and the element's node displacements.

4. Combine the set of equilibrium equations for all of the elements in the object into a single system. Solve the system for the node displacements over the whole object.

5. Use the node displacements and the interpolation functions of a particular element to compute displacements or other required quantities (such as internal stress or strain) for points inside the element.

More detailed treatment of the FEM can be found in textbooks such as [56], [57], or [58].

2.5 Related Work on the Finite Element Method

In order to reduce computation time of the linear FEM, Bro-Nielsen et al. used a condensation technique to partition the problem into interior and surface points and solve for deformations only at the surface points [59], [60]. This method improved the speed of the simulation but required preprocessing. It was, therefore, not suitable for tissues whose shape or topology could be changed due to cutting and suturing.

The KISMET software for a minimally invasive surgery (MIS) by Kühnapfel et al. [10] used their so-called FFEM (Fast FEM). The FFEM calculated the linear deformation of only the surface nodes in the volumetric 3D mesh, either by mesh sorting or by model condensation. The reasons were that only these nodes could interact with surgical instruments and only these nodes were visible. Thus, the method reduced memory usage and computational complexity by using mesh condensation and suffered the same drawbacks as of Bro-Nielsen et al.

Müller et al. [61] presented a technique called stiffness warping to solve the incorrect deformation problem of the linear FEM caused by large deformations. With a precomputation of the linear stiffness matrix, the time complexity of the method was similar to the linear FEM, which did not run in real time.

A parallel computing system was used to animate the finite element models in real time by Szekely et al [11]. They stated that the only way to have a complex finite element system running in real time was to use a parallel computer which supported fully parallel algorithms for the explicit time integration scheme. A software simulation of a 3D lattice of processors was implemented for the simulation. There were no follow-up implementation of a physical phototype of the proposed parallel computer system.

The FEM based on cubical elements of uniform size was used by Müller et al. [62] to generate a volumetric mesh of an object from its surface mesh. This technique allowed a surface mesh to possess a physically based elastic property, and could be deformed or fractured based on its association with its associated volumetric mesh. The

simulated object can be deformed and fractured in real time. However, the method did not implement the simulated model for cutting which required a straight line of fracture instead of a typical jagged fracture created by the simulation method.

Delingette and Ayache [15] described their work on soft tissue modeling for surgery simulations at INRIA[1] . They discussed quasi-static linear, dynamic linear, and quasi-static nonlinear elastic model based on the FEM, including some techniques to speed up the computation. The number of tetrahedral had to be kept small, so that the simulated model could run in real time. Their models deformed realistically, but a cut on the model produced jagged cut instead of a straight line due to the coarse subdivision (or splitting) of the tetrahedral mesh that represented the model.

Other methods such as modal models [4], [63], [64], boundary element models (BEM) [65], long elements models (LEM) [66], finite volume method (FVM) [67], and resolution adaptive models [68], are not reviewed here, since they were variations of the finite element method and suffered the same drawback that they were not suitable to model objects that underwent topology change at an interactive rate due to required preprocessing of their system.

2.6 Mass-Spring System vs. Finite Element Method

Both the mass-spring system and the finite element method have been developed to introduce physics into the simulation process [69]. The mass-spring system (MSS) reduces a volumetric 3D object to a network of mass nodes connected with 1D Hookean springs. Such simplification makes the MSS efficient to compute, easy to implement, and easy to modify the model's topology even on a parallel computing environment. On the downside, the simplification also makes the MSS less accurate and difficult to configure the model's behavior, since all the materials that make up the object are simply lumped together as discrete mass points connected to one another by springs. The MSS can exhibit oscillation

[1] The French national institute for research in computer science and control

or become unstable if the parameters are not carefully set up, such as increasing the time step and/or spring stiffness too high.

In contrast, the finite element method (FEM) applies continuum mechanics to every constituting element of the object. The stress-strain relationship that governs object deformation can be formulated by the generalized Hookes Law for the linear FEM. Therefore, more accurate results can be obtained from the FEM, than from the results obtained from the MSS. However, models built from an FEM based system are computationally more complex than models built from the MSS [46]. The FEM is unsuitable for use in real-time surgery simulation, due to the high computation time required by a numerical integration to solve the complex system.

Hence, the FEM is a method that is well suited for product design and testing, where a high computation time is justified by the accuracy of simulated models. With the availability of fast computers, there has been a trend in real-time animation to use more sophisticated and accurate models based on the FEM, rather than using simple models based on the mass-spring system. The object's deformation behavior can be specified using only a few parameters. However, some of computational complexity increases quadratically with the increasing number of elements [70]. In real-time application, either the resolution (number of elements) has to be kept small or preprocessing has to be applied, which results in the reduction of flexibility. For example, each topology change requires recalculation and assembly of the element mass and stiffness matrices, which is computationally intensive [70]. The MSS, on the other hand, is less accurate, but very efficient in computation time. A change of topology simply requires the corresponding change of the spring connectivity.

The advantages and disadvantages of the MSS over of the FEM are summarized in Tables 2-1 and 2-2. Table 2-3 compares the MSS to the FEM based on the order of importance listed on page 13. In summary, the MSS is more suitable than the FEM for modeling deformable objects in real-time application, even though it requires more care in

setting up parameter values to avoid unstable system behavior and to achieve a visually plausible behavior.

Table 2-1. Advantages of the mass-spring system over of the finite element method

Comparison	Mass-Spring System	Finite Element Method
Complexity	Low ✓	High
Complexity increment	Linear ✓	Quadratic
Implementation	Simple ✓	Complex
Parallelism	Each mass point ✓	Each element
Map to GPU-based computing	High ✓	Low
Cost of topology change	Low ✓	High
Overall real-time performance	**High**	**Low**

Table 2-2. Disadvantages of the mass-spring system compared to the finite element method

Comparison	Mass-Spring System	Finite Element Method
Stability	Unstable under conditions	Stable ✓
Accuracy	Low	High ✓
Property adjustment	Complicated	Simple ✓
Overall tweak requirement	**High**	**Low**

Table 2-3. Mass-spring system vs. finite element method (based on the order of importance listed on page 13)

Comparison	Mass-Spring System	Finite Element Method
Speed	✓	
Robustness		✓
Flexibility	✓	
Geometry resolution	✓	
Physical accuracy		✓
Memory usage		

CHAPTER 3
Deformable Object Modeling

The deformable objects encountered in surgery are mainly soft tissues. These objects are subject to being touched, pulled, and cut by surgical tools. In order to simulate the reaction of a deformable object in real time with plausible deformation, the simulation model must run fast enough and show plausible deformation with satisfactory rendering quality.

3.1 Deformable Object Modeling Framework

The framework for modeling a deformable object is composed of 2 preprocessing and 3 iterative steps:

1. Obtain a three-dimensional (3D) regular grid that represents the modeled object.
2. Set up the connectivities and properties, such as spring properties, of the object.
3. For each specific time step, simulate the object's deformation due to its own internal forces and external forces exerting on it.
4. For each specific time step, simulate the object's deformation due to the collision of the object with surgical tools .
5. For each specific time frame, render the object for user interaction.

Due to the intrinsically implicit connectivity of the 3D regular grid, which is a type of 3D structured grids, representing the modeled object (Fig 3-1). Elements and connections among the elements of the model can be added, changed, replaced, or removed to create a topology change. This topology change mimics a deformation resulting from an operation, such as cutting, suturing, or cauterizing.

To be able to trade speed for accuracy, the deformable object is separated into three representations: dynamic response, collision detection, and visualization. This way dynamic response (step 3), collision detection (step 4), and visualization (step 5) can be implemented and handled separately.

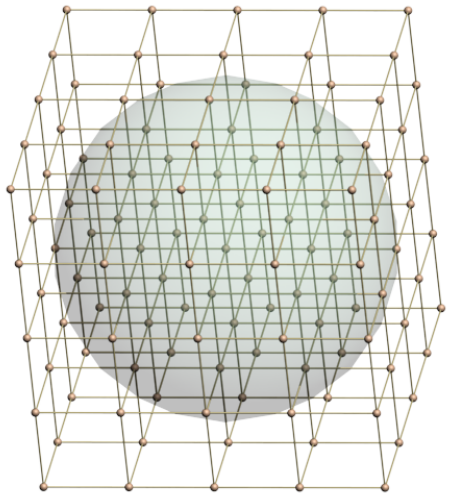


Figure 3-1. Volumetric model enclosed inside the 3D regular grid.

The dynamic response chosen for this framework is a volumetric mass-spring model arranged in a 3D regular grid. Each mass point in the model has spring connectivity of up to 26 neighboring lattice (Figure 3-2) plus 1 for the home spring, which is a spring connecting the mass point to its initial or rest position [71], [72]. To accommodate a change of topology in real time, the visualization representation is a level set surface approximated by a triangular mesh, generated in real time from the 3D regular grid by the marching cubes algorithm [73].

The following sections explain the framework steps mentioned above: transforming a surface to 3D regular grid data by voxelization, setting up a 3D mass-spring system, dynamic response, collision detection, and visualization.

3.2 Voxelization: Transforming a Surface to a 3D Regular Grid

Volumetric data representing a deformable object can be obtained by 3D measurement (e.g., from medical scanners, such as computerized tomography (CT) or magnetic resonance imaging (MRI)), by voxelizing a polygonal mesh, by generating procedural volume data, or by any combination of the three aforementioned methods [2], [3]. Also, data already contained in a 3D regular grid can be used directly. It can be either

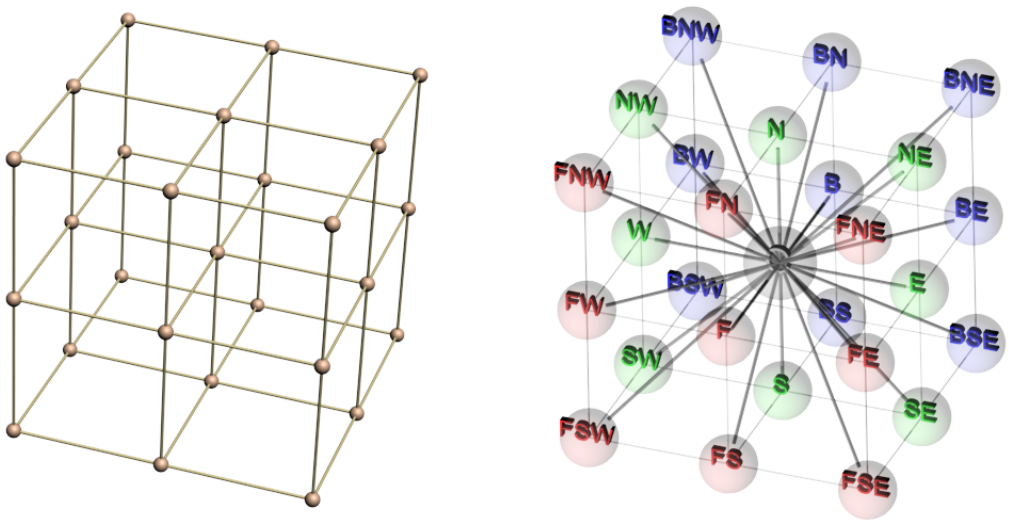


Figure 3-2. A node can be connected to its adjacent neighbors in the 3D regular grid by up to 26 connections + 1 for the home spring. Where F = Front, B = Back, N = North, S = South, E = East, W = West, and C for center, i.e., the home spring.

subsampled (or supersampled) to reduce (or increase) data size to the desired model data size.

Here, we focus on obtaining a 3D regular grid from voxelizing the input surface mesh for modeling a deformable object. Surface meshes are ubiquitous and can be generated quickly by an algorithm or 3D modeling software, such as Blender. The voxelization can be done quickly and efficiently on any polygonal surface mesh representing an arbitrary object. This advantage allows us to be able to model any solid object that can be represented by a polygonal mesh (Fig 3-3).

Notice that aliasing artifacts can be observed from the voxelized volume data. The higher the voxelized resolution, the finer the aliasing artifacts. This is due to the fact that the voxelized volume data are binary, i.e., each voxel is classified to be either 'inside' or 'outside' the modeled object. The aliasing artifacts occur when we try to extract the object's surface from the binary voxelized volume data. The original surface always intersects a specific space between a pair of 'inside' and 'outside' voxels, i.e., an edge of a cube. This information is not available with the binary volume data, i.e., the information

is loss when the object is voxelized. The interpolation value can be any where between 0 and 1, i.e., from the 'inside' voxel to the 'outside' voxel. The interpolation value is usually set to 0.5, which in this case the original surface is assumed to intersect the edges of the cube at the middle. The rendered image exhibits terracing where the original surface lie at a shallow angle to the sampling grid. This makes the sloped surface appear as a sequence of flat planes separated by sudden elevation changes [74].

To alleviate this problem, the exact intersections of the surface mesh with the edges of the cubes during the voxelization can be stored and later used for rendering the object. In our implementation, these data are stored as interpolation values in a GPU texture. Unfortunately, we cannot use the normals from the surface mesh for visualization, since the normals always vary with the deformation of the surface.

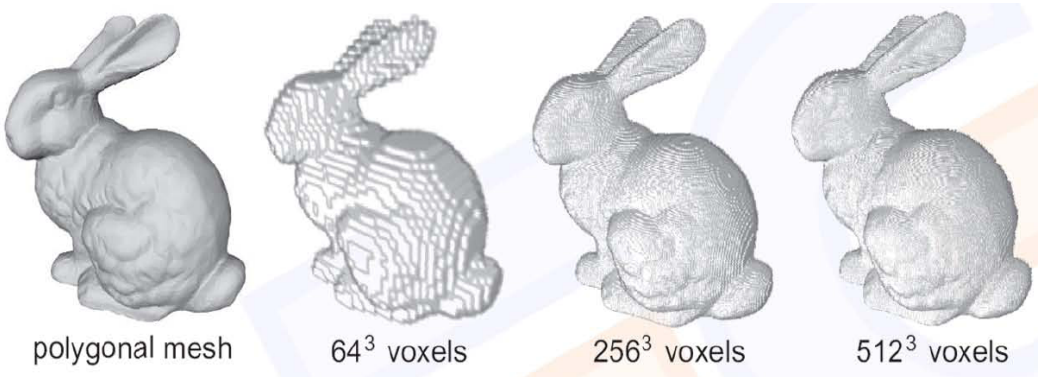
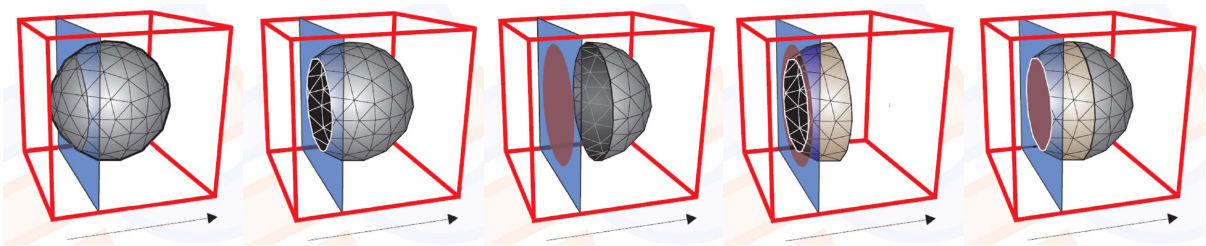


Figure 3-3. Creating a binary volume data set by voxelizating a polygonal surface mesh (reprinted from [3], freely accessible from the Internet)

A polygonal mesh can be voxelized by GPU or CPU computation. For obtaining the model data by GPU computation, the GPU is used to create the slice planes for the model data. A slice plane is a rectangular surface that contains scalar or color information based on the values of the volume data where the slice is positioned. The ingredients are the rendering of the front and back faces of the mesh and a clipping plane to clip the mesh into the clipped and unclipped part. The clipping plane is set up with the same position and orientation as the current slice plane. Without loss of generality the unclipped part is set to the space behind the clipping plane. Therefore, the first slice can be a clipped

plane located behind the mesh. An example for an intermediate slice plane is shown in Figure 3-4a. With back-face culling enabled (Figure 3-4b), the GPU renders the front faces of the mesh with foreground color to a framebuffer of desired rectangular size (Figure 3-4c). Then with front-face culling enabled, the GPU renders the back faces of the mesh with background color to the framebuffer (Figure 3-4d). The framebuffer now contains the cross section of the mesh (Figure 3-4e). The data in the framebuffer is then copied to the current slice. The next slice of the model data can be created by shifting the clipping plane to the next position and re-rendering the front and back faces of the mesh with aforementioned steps above. The process is repeated until the whole volume data is created, i.e., until the total number of slices is reached.



a) b) c) d) e)

Figure 3-4. Individual steps of the GPU voxelization algorithm (reprinted from [3], freely accessible from the Internet)

For the model data obtained by CPU computation, first the scalar values of all voxels are set to a value that represents an 'unset' state, then each triangle of the mesh is traversed. A bounding box is then created to cover the triangle. Inside and on the boundary of the created bounding box are the polygon and sampled voxels (Figure 3-5). Each sampled voxel is classified as lying in front of or behind the plane of the front face of the triangle. If the voxel is behind or on the plane, it is assumed to be inside the model and its scalar value is set to 'inside' the model. Otherwise, its scalar value is set to 'outside' the model. If the sampled voxel is already classified, then a decision has to be made to favor either the 'inside' or 'outside' element. In our implementation, if the voxel is found to lie in front of the plane and the voxel's value is already set to 'inside', then the

voxel value is changed to 'outside'. While if the voxel's value is already set to 'outside', then the voxel value is unchanged. This means our algorithm prefers the mesh model that is convex. Depending on the sampling size, the 3D lattice data set created from a mesh model that is not fully convex by this algorithm may not be correct. This traversal process runs for each triangle in the mesh model. After all triangles have been traversed, a flood fill algorithm ([75] on page 979) classifies the rest of the model data to either 'inside' or 'outside' the mesh based on the voxel values set up from the previous traversal process.

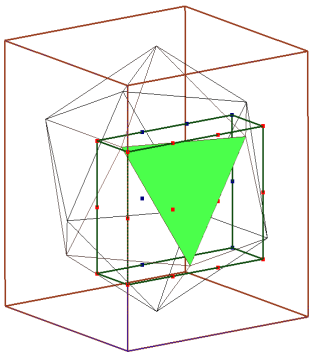


Figure 3-5. Example of voxelizing a triangle on a mesh model by a software algorithm.

3.3 Setting up a 3D Mass-Spring System

After obtaining the 3D regular grid data set, the next step is to set up a 3D mass-spring system. The setup mass-spring system will be run entirely on a GPU.

A current consumer graphics hardware or GPU can be used as a parallel execution engine, not only for graphical usages, but also for general-purpose computations [76], [77]. The GPU is designed for compute-intensive and highly parallel computations specifically for graphics rendering. Compared to CPU, most of transistors in the GPU are devoted to data processing rather than data caching and flow control [78]. Since the movement of each mass point in the mass-spring system caused by the internal and external forces acting on the system can be computed independently of one another, the computation can be mapped to a data-parallel computation. Therefore, the GPU can be used for dynamic response, collision detection, and visualization of our deformable object at a

much faster frame rate than the CPU. With a simple experiment on a 3D mass-spring system computation, the result was 10 to 15 times faster.

One reason for the speedup gain is the memory access provided by the GPU [79] (Figure 3-6). The memory access address for a CPU is sequential or '1D', while the

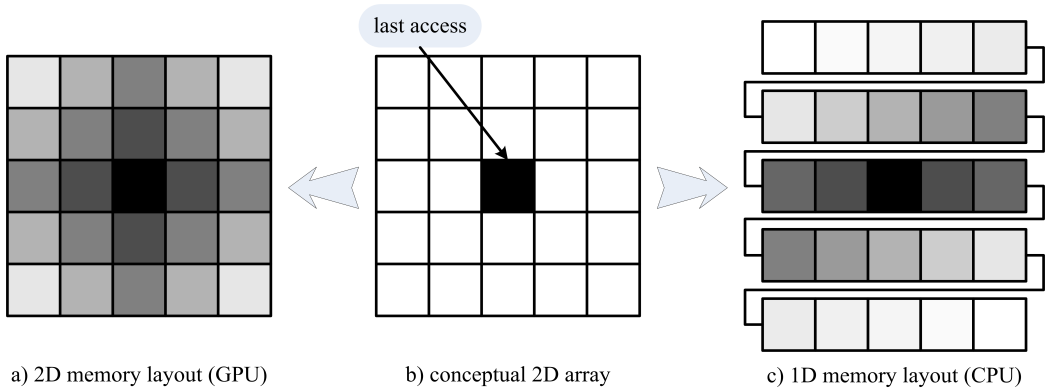


Figure 3-6. Example of memory layout for a) the GPU and b) CPU showing that the GPU has a faster memory access than of the CPU. Where the darker the gray, the faster the access [79].

memory access address for a GPU is parallel or '2D'. As volumetric simulation data are organized as a 3D array or a collection of 2D arrays, memory access times are optimal if the 2D memory access distance to the current memory location is small [79]. Therefore, this 2D memory access provided by the GPU fits well with our simulation data.

Volumetric data of our deformable object are arranged in a 3D regular grid, where each data element is called a voxel and eight voxels form a logical cube. Each voxel is mapped to a mass point with springs that connect its position to its neighboring mass points in the 3D regular grid. Hence, the total number of mass points is equal to the total number of voxels. Only the 'inside' and 'outside' voxels in the 3D regular grid will be processed by our deformable object simulator.

The volumetric data in the 3D regular grid maps naturally to a GPU 3D texture. This allows implicit access to spatial neighbor nodes (Figure 3-2 on page 32). However, another 3D data structure is set up for looking up connections explicitly. The idea behind this connectivity data structure is to provide flexibility in data setup. Different

connectivities for different regions in the 3D texture, for example, can allow multiple components to be simulated differently and simultaneously by one 3D texture, and can ease the connectivity change due to, for example, a cut by a surgical tool.

After the 3D mass-spring system is set up; the position and connectivity of each mass point in the system are initialized or defined, then the dynamic response of the system can be computed. The next subsections discuss the ingredients for implementing our dynamic response.

3.4 Dynamic Response

The mass-spring system is a dynamic system. The system is based on a group of finite particles, i.e., mass points, connected to a predefined set of other mass points by linear springs based on Hooke's (spring) law — the force is proportional to the extension of the spring. Therefore, the behavior of the system is controlled through the forces computed from the extensions of the springs in the system. The system represents a second-order ordinary differential equation (ODE). The ODE is solved by the Verlet integration discussed below.

3.4.1 Ordinary Differential Equation Solvers

Particles are one of the easiest objects to animate. Each particle has position, velocity, and mass, but no spatial extension, i.e., neither moment of inertia or torque. The behavior of these particles is governed by Newton's second law, which is a second-order differential equation on $\mathbf{x}$.

$$\mathbf{f} = m\ddot{\mathbf{x}}, \tag{3–1}$$

where $\mathbf{x}$ or $\mathbf{x}(t) = [x, y, z]^T$ is the position of the particle at time t, and $\ddot{\mathbf{x}} = \frac{d^2\mathbf{x}(t)}{dt^2}$ is the acceleration of the particle. This second-order differential equation can be converted to a system of coupled first-order differential equations (position and velocity) where $\dot{\mathbf{x}} = \mathbf{v} = \frac{d\mathbf{x}(t)}{dt}$:

$$\mathbf{p} = \begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \dot{\mathbf{x}} \end{bmatrix} \in \mathbb{R}^6 \tag{3–2}$$

Therefore,

$$\dot{\mathbf{p}} = \begin{bmatrix} \mathbf{v} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{x}} \\ \ddot{\mathbf{x}} \end{bmatrix} \tag{3–3}$$

In a simple explicit solution of this system of first-order differential equations, the velocity and the acceleration at time t are used to update the particle's position and velocity at time $t + h$, respectively. Hence, the effect of forces at time t will have an influence on the particle's position at time $t + 2h$. The problem is now in initial value form $\dot{\mathbf{p}}(t) = \mathbf{F}(\mathbf{p}(t))$ under the condition that $\mathbf{p}(t_0) = \mathbf{p}_0 = [\mathbf{x}_0, \dot{\mathbf{x}}_0]^T$. The equation is solved for $t > 0$. This initial value problem can be solved by discrete approximations, such as the explicit Euler integration or the Verlet integration method [80].

The derivations of the explicit Euler integration and the Verlet integration are described in the subsections below. Other explicit integration methods are not considered here, since a slight accuracy gain does not out weigh the speed loss due to higher time complexity that hinders the simulation speed. While any implicit integration method provides a much higher stability of the system, it requires a linear system to be solved at each time step, i.e., each mass point cannot be solved independently [41], [81], [82]. It means whenever there is a topological change of the model, the ODE system has to be reconstructed. Therefore, the same reason is applied for not choosing an implicit integration method. Hence, only the explicit Euler integration and the Verlet integration methods are suitable for our deformable object modeling framework.

3.4.1.1 Explicit Euler integration

In the explicit Euler integration, the time is discretized to $t_{i+1} = h + t_i$, where $i >= 0$ and $h > 0$. A second-order accurate approximation of the position at time t_{i+1} can be approximated by the first-order Taylor Series

$$\mathbf{p}(t_{i+1}) = \mathbf{p}(t_i) + h\dot{\mathbf{p}}(t_i) + \mathcal{O}(h^2) \tag{3–4}$$

Since the initial value of $\mathbf{p}(t_0)$ is assumed to be known or given and $\mathbf{F}(\mathbf{p}(t)) = [\dot{\mathbf{x}}, \mathbf{f}(t)/m]^T$, then the approximation of $\mathbf{p}(t_1)$ is

$$\mathbf{p}(t_1) \approx \mathbf{p}(t_0) + h\mathbf{F}(\mathbf{p}(t_0)) \tag{3–5}$$

Which becomes

$$\begin{bmatrix} \mathbf{x}_1 \\ \dot{\mathbf{x}}_1 \end{bmatrix} \approx \begin{bmatrix} \mathbf{x}_0 \\ \dot{\mathbf{x}}_0 \end{bmatrix} + h \begin{bmatrix} \dot{\mathbf{x}}_0 \\ \mathbf{f}(t_0)/m \end{bmatrix} \tag{3–6}$$

Hence, the calculation for each $i+1$ step is

$$\begin{bmatrix} \mathbf{x}_{i+1} \\ \dot{\mathbf{x}}_{i+1} \end{bmatrix} \approx \begin{bmatrix} \mathbf{x}_i \\ \dot{\mathbf{x}}_i \end{bmatrix} + h \begin{bmatrix} \dot{\mathbf{x}}_i \\ \mathbf{f}(t_i)/m \end{bmatrix} \tag{3–7}$$

So the next position and velocity, $i+1$, are solved simultaneously

$$\begin{bmatrix} \mathbf{x}_{i+1} \\ \mathbf{v}_{i+1} \end{bmatrix} \approx \begin{bmatrix} \mathbf{x}_i \\ \mathbf{v}_i \end{bmatrix} + h \begin{bmatrix} \mathbf{v}_i \\ \mathbf{a}_i \end{bmatrix} \tag{3–8}$$

This explicit Euler integration method is very simple to implement, and it is very fast to compute for each iteration. However, the solution may become unstable and the accuracy is worse with respect to each increasing time step [41].

3.4.1.2 Verlet integration

Verlet integration method was introduced in 1967 by Loup Verlet for computer experiments on classical fluids [80]. From the first-order Euler integration equation,

$$\begin{bmatrix} \mathbf{x}(t+h) \\ \mathbf{v}(t+h) \end{bmatrix} \approx \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{v}(t) \end{bmatrix} + h \begin{bmatrix} \mathbf{v}(t) \\ \mathbf{a}(t) \end{bmatrix} + \mathcal{O}(h^2) \tag{3–9}$$

Using the Taylor series to approximate $\mathbf{x}(t+h)$ and $\mathbf{x}(t-h)$,

$$\begin{aligned} \mathbf{x}(t+h) &= \mathbf{x}(t) + h\dot{\mathbf{x}}(t) + \frac{1}{2}h^2\ddot{\mathbf{x}}(t) + \frac{1}{6}h^3\dddot{\mathbf{x}}(t) + \mathcal{O}(h^4) \\ \mathbf{x}(t-h) &= \mathbf{x}(t) - h\dot{\mathbf{x}}(t) + \frac{1}{2}h^2\ddot{\mathbf{x}}(t) - \frac{1}{6}h^3\dddot{\mathbf{x}}(t) + \mathcal{O}(h^4) \end{aligned} \tag{3–10}$$

Adding the above equations results in

$$\mathbf{x}(t+h) + \mathbf{x}(t-h) = 2\mathbf{x}(t) + h^2\ddot{\mathbf{x}}(t) + \mathcal{O}(h^3) \tag{3–11}$$

Rearranging the equation to

$$\mathbf{x}(t+h) = 2\mathbf{x}(t) - \mathbf{x}(t-h) + h^2\ddot{\mathbf{x}}(t) + \mathcal{O}(h^3) \tag{3–12}$$

This result in the velocity is represented implicitly by the previous and current positions. By subtracting the two Taylor series approximation equations above and rearranging the result equation, the velocity is

$$\mathbf{v}(t) = \frac{\mathbf{x}(t+h) - \mathbf{x}(t-h)}{2h} + \mathcal{O}(h^2). \tag{3–13}$$

Since the Verlet integration does not need to keep track of velocity, the next time step solution can be computed in one step by Equation 3–12. The absence of the velocity term makes the implementation of constraints between particles simpler, such as handling a collision response. Whenever a particle penetrates an object, the particle is simply moved out of the object via the shortest path. The subsequent new implicit velocity will be the subtraction of the new position from the previous position that touches the object's surface. Hence, the implicit velocity will be slid along the surface. Also whenever a particle's position is adjusted, its velocity will be taken care of implicitly by the Verlet integration. The implicitly approximated velocity in the Verlet integration results in more damping in the particle system. This extra damping makes the Verlet integration method have better stability than the explicit Euler integration method.

Another advantage of the Verlet integration method over the explicit Euler integration method is the size of storage. The Verlet integration needs the previous, current and next positions, while the explicit Euler integration needs the current and next positions and the current and next velocities. These advantages are enough for choosing the Verlet integration over the explicit Euler integration for our deformable object simulator.

3.4.2 Data Structure on the GPU

The data structure on the GPU simulates as a CPU-based 3D array by a GPU-based 3D texture. To keep the GPU's shaders running in parallel, a texture of the GPU cannot be read and write to at the same time. Hence, two textures are needed for current and next stages, respectively. The explicit Euler integration requires current position and velocity of a mass point for computing the next position and velocity of the mass point. Therefore, four textures are needed. Two for (read) current position and (write to) next position, and the other two for (read) current velocity and (write to) next velocity. The Verlet integration requires previous and current positions of the mass point for computing its next position. Therefore, three textures are needed. The Verlet integration does not need to know the current velocity of a mass point, since the velocity of a mass point is approximated by the difference of the current and previous position. This difference shows that the Verlet integration requires less memory and computation time and memory for solving the ODE system on the GPU than that of the explicit Euler integration.

3.4.3 Force Computation

After the data structure is set up, the force computation can be processed. The forces in the dynamic response consist of internal forces and external forces, $\mathbf{F}^{tot} = \mathbf{F}^{int} + \mathbf{F}^{ext}$. The total force that acts on mass point i in the system is the sum of internal forces from connecting springs and external forces from outside interactions, such as a gravitational field.

$$\mathbf{F}_i^{tot} = \mathbf{F}_i^{int} + \mathbf{F}_i^{ext} = \sum_{j \in \text{neighbors of } i} k_{ij}(l_{ij} - ||\mathbf{x}_i - \mathbf{x}_j||)\frac{\mathbf{x}_i - \mathbf{x}_j}{||\mathbf{x}_i - \mathbf{x}_j||} + \mathbf{F}_i^{ext} \tag{3–14}$$

The internal force is computed by a GPU fragment shader. The shader looks up the previous and current positions of each mass point from the 3D textures based on the specific input texture coordinate. It also iteratively looks up the connectivity of the mass point to the adjacent neighbors. If a connection exists, the shader fetches the current position of the connected neighbor and computes the internal force of the spring

connecting the current position of the mass point to the current position of the neighbor. These internal forces are added based on the number of connected neighbors to the mass point. An external force, e.g., a gravitational force, can be added into this computation pass. The shader then uses the Verlet integration to find the next position of the mass point based on the total force acting on the mass point. For solving a collision with other objects, constraint-based displacement is used instead of the exact collision force (see Section 3.5 Collision Detection, Topology Changes, and Haptic Sensation).

3.4.4 Parameter Adjustment and Constraints

The stability of a mass-spring system notoriously depends on its parameters. Where one change of its parameter value can cause its dynamic response to diverge. Also if its parameters are not set properly, its dynamic response will diverge. With the requirement that the system must run in real time, there is no easy way to automatically setup parameters for each generated model size and shape.

In the deformable object simulator, the movement (farther away or closer) between two connected particles represents the elasticity of the deformable object. Most soft organic tissues have higher elastic deformation than most solid objects, such as aluminum, plastic, or cloth, but much less than elastic material, such as rubber. The stiffness of the deformable object can be adjusted by changing the stiffness of springs. If the stiffness of springs is set too low, then the model appears rubber-like. However, if the stiffness is set too high the mass-spring system will not be stable.

In order to make the simulated deformable object look more stiff without resorting to a more complex computation that hinders the performance or risks creating an unstable system, a constraint is added to adjust the degree of movement of a particle away from or toward its connected particles. If a spring is compressed or elongated more than the threshold, then an offset is applied to set the distance between the two mass points to be within the threshold, hence, preserving the length of the spring to within the threshold.

This relaxation constraint acts as an invisible barrier that block the particle from moving too close or too far from its connected particles [29]. Overall, this constraint makes the model appear less elastic and speeds up the convergence of the dynamic response of the system toward an equilibrium stage. It fits well with the Verlet integration due to the absence of the explicit velocity term in the Verlet integration.

Hence, stable default parameters are set for initializing the deformable object, and the user can empirically adjust the parameters of the mass-spring model to change its dynamic behavior.

3.5 Collision Detection, Topology Changes, and Haptic Sensation

The simulated deformable object must be able to interact with other objects, such as surgical tools and other objects, in the simulation environment. The collision detection will determine whether each mass point and/or each of its connection collides with the objects in the environment. There are three ways to resolve the collision without changing the resolution of the 3D regular grid, in which the last two ways can change the topology of the mass-spring system:

- Move the colliding mass point away from the object
- Remove the colliding mass point from the mass-spring system
- Remove the colliding connection from the mass-spring system

For collision detection and response of each mass point with a virtual tool, we compute the distance of the mass point from the bounding primitive (or the set of bounding primitives) that represents the (part of) tool. If the distance is less than a threshold, i.e., the mass point is considerably inside the bounding primitive, then the mass point is either removed or moved to outside the bounding primitive via the shortest path.

[**Topology Change by Removing Mass Points**] Removing the mass points in contact with the cutting part of a surgical tool is the same as voxel popping (Figure 3.5). This is similar to cautery — the process of destroying (abnormal) tissue by burning or searing using an agent or instrument.

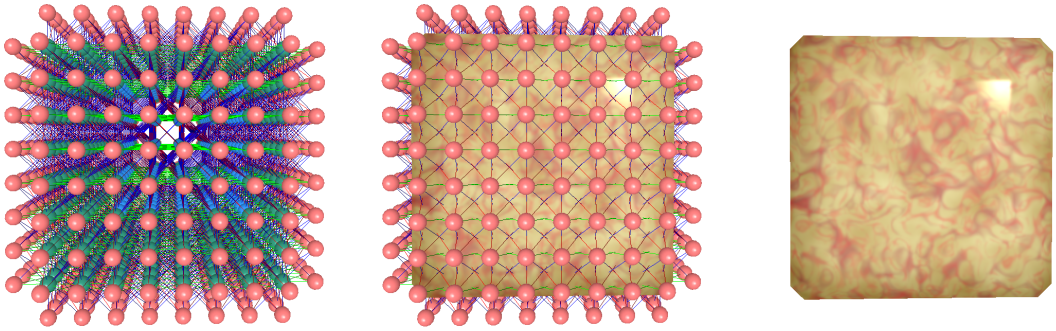
[**Topology Change by Removing Connections**] Removing the connections between the mass points in contact with the sharp edge of a cutting tool is shown in Figure 3.5). Each connection is treated as a line segment. The collision detection finds whether the line segment intersects with the bounding primitive (or the set of bounding primitives) that represents the sharp part of the tool. If the intersection occurs, the connection is removed from the mass-spring system.

[**Haptic Sensation**] To provide haptic sensation to the user, the physical device representing the virtual surgical tool that is manipulated by the user must be able to provide force feedback. We uses a PHANTOM®Omni™haptic device [1] (Figure 3-8), which provides six degree-of-freedom for tool manipulation and three degree-of-freedom (along x, y, and z direction) for force feedback. The collision detection will record penetration distance of each mass point in the mass-spring system with the virtual surgical tool in a GPU 3D texture. The data are read from the texture by the CPU. The CPU calculates the average sum of the data and uses it to set the force feedback on the haptic device to provide the user with sense of touch.

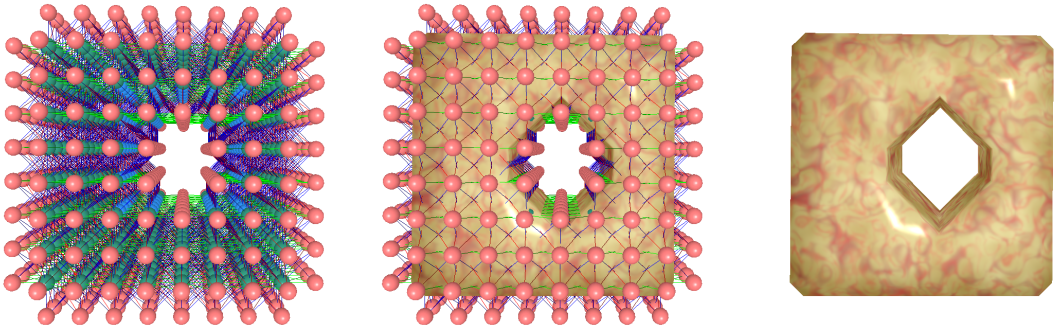
After the dynamic response and collision detection have run for a number of fixed time steps to advance the deformable object's dynamic behavior through time, the visualization can display the deformable object to the screen.

3.6 Visualization

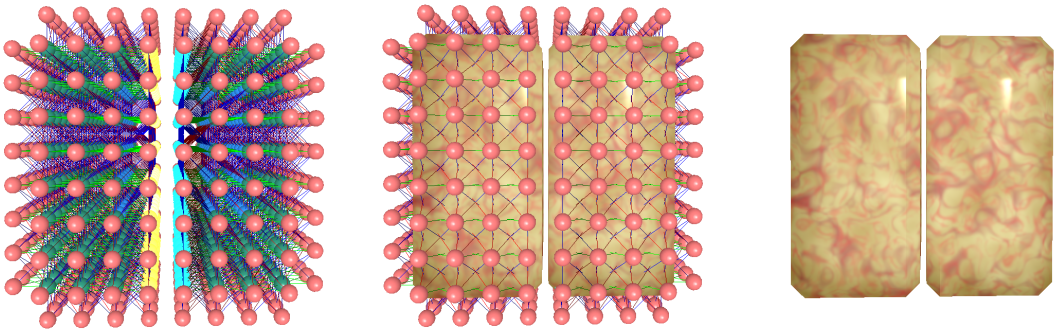
The goal is to display a volumetric model to the screen. The visualization data can be generated from the same source used to create the model data for the simulation. The data size of the visualization can be smaller than, the same as, or larger than the size of the simulation data. For our deformable object, we use the same data size for both simulation (dynamic response and collision detection) and visualization. The visualization is usually updated at a rate of 30Hz. The higher the simulation update rate, the better the simulation of the object's dynamic behavior. That means the simulation update rate should be higher or at least equal to the visualization update rate.



a) deformable cube: mass points, surface with mass points, and only surface



b) removing mass points change the surface



c)) removing connections change the surface

Figure 3-7. Simulate cuts and cauteries of a cube object. a) A deformable cube is cautery by b) removing mass points and cut by c) removing connections.

Figure 3-8. A PHANTOM®Omni™haptic device (reprinted from [1], freely accessible from the Internet)

Three options for visualization were considered: 3D texture, ray casting, and mesh approximation. Ray casting is the most popular image-order method for direct volume rendering [3]. It generates an image containing both the outside and inside of a volumetric object by tracing a ray of light from a light source to the object's surface and, if applicable, through the inside of the object that is reflected to the viewer's eyes, for example, like a camera lens. A lot of ray casting techniques have been developed for real-time volume rendering [2], [3]. However, with interactions of deformable objects and surgical instruments in a surgical simulation, the ray casting (or direct volume rendering) method is still too computationally expensive and hinders the simulation from running in real time.

Visualizing a volumetric model with both the inside and outside part can be accomplished by rendering a 3D texture that is mapped onto the volumetric model [2], [3]. Each data of the volumetric object is mapped to a texel in the 3D texture via texture coordinates. Using trilinear interpolation, the visualization renders each pixel on the screen from a combination of the texels and the data in the 3D regular grid [83, 84]. The major draw back of the 3D texture visualization is that the smoothness of the rendering depends on the size of the 3D regular grid (Figure 3-9). A deformation of the model can be simulated by adjusting the texture coordinate associated with the deformed

data (Figure 3-10). Also, the object's deformation space is limited by the dimension of the 3D texture in the world space.

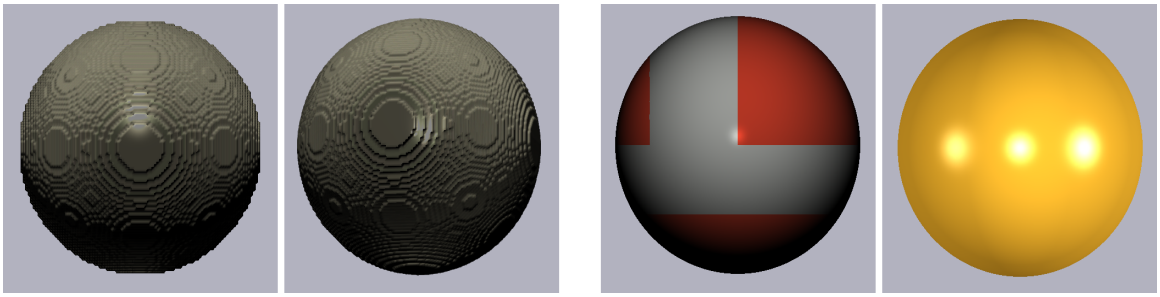


Figure 3-9. Sphere rendered by a (100x100x100) 3D Texture visualization compared with a sphere rendered by a (25002 vertices and 50000 faces) mesh visualization

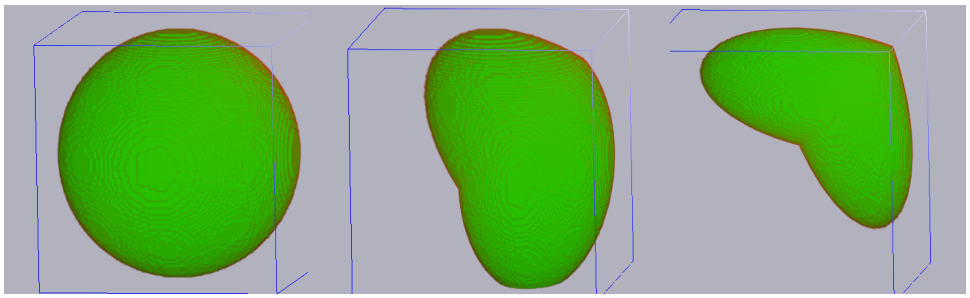


Figure 3-10. Sphere deformation visualized by a (100x100x100) 3D Texture visualization

The third option was to associate a surface mesh with the volumetric object. Each vertex of the mesh was mapped to a boundary data of the volumetric object. The surface deformation was then a transformation of the boundary data to the vertex [85], [52]. Mesh rendering gave a good smooth approximation of the volumetric model, but it was hollow inside. A cut on the model would result in a cut on the mesh [86]. This cut would expose the model as a hollow object. One possibility was to superimpose both on the screen; the mesh as the skin of the model and 3D texture as the material inside the model. Our study did not test on this combination. Instead, we resorted to another mesh approximation option.

Since the visualization must be able to modify its topology which is associated with the simulated object's topology that undergoes changes due to, for example, cutting, a

real-time generation of an approximated mesh from the volumetric data can be used. The marching cubes algorithm is used in this study for generating a surface triangular mesh for visualizing the volumetric data in real time.

3.6.1 Marching Cubes

The original Marching cubes algorithm [73], [87] (and its variations) is the most widely used method for the extraction of isosurfaces from 3D volumetric data [2]. The algorithm generates polygonal representations of 3D volumetric data directly from a 3D volumetric data set by tessellating a logical cube constructed from eight voxels in the data set. The surface boundary of the model contained inside the data set can intersect a cube according to a classification of each cube vertex into inside or outside the surface. The classification of the eight vertices of the cube results in 256 possible configurations, but only 15 topologically distinct patterns exists from the 256 configurations (Figure 3-11).

To shade the surface of the 3D mesh projected onto the view plane for display, an intensity is calculated from the component of the unit normal vector parallel to the view direction

[87]. The gradient vector of the intensity 3D data estimates the surface normal direction, since the gradient is perpendicular to surfaces of constant density.

$$
\begin{aligned}
g_x &= \frac{f(x_0 + a, y_0, z_0) - f(f(x_0 - a, y_0, z_0)]}{2a} \\
g_y &= \frac{f(x_0, y_0 + b, z_0) - f(f(x_0, y_0 - b, z_0)]}{2b} \\
g_z &= \frac{f(x_0, y_0, z_0 + c) - f(f(x_0, y_0, z_0 - c)]}{2c}
\end{aligned}
\tag{3–15}
$$

Equation 3–15 is a gradient estimation by central differences of the three components of the intensity function. In visualization, the gradients are used to provide contrast that depends on the surface orientation to the rendered image.

However, the 3D regular grid data set obtained from voxelization of an input mesh model does not have gray-scale intensity. Instead it is a binary data set. This is a major disadvantage of any discrete representations, such as our voxel-based data, since its

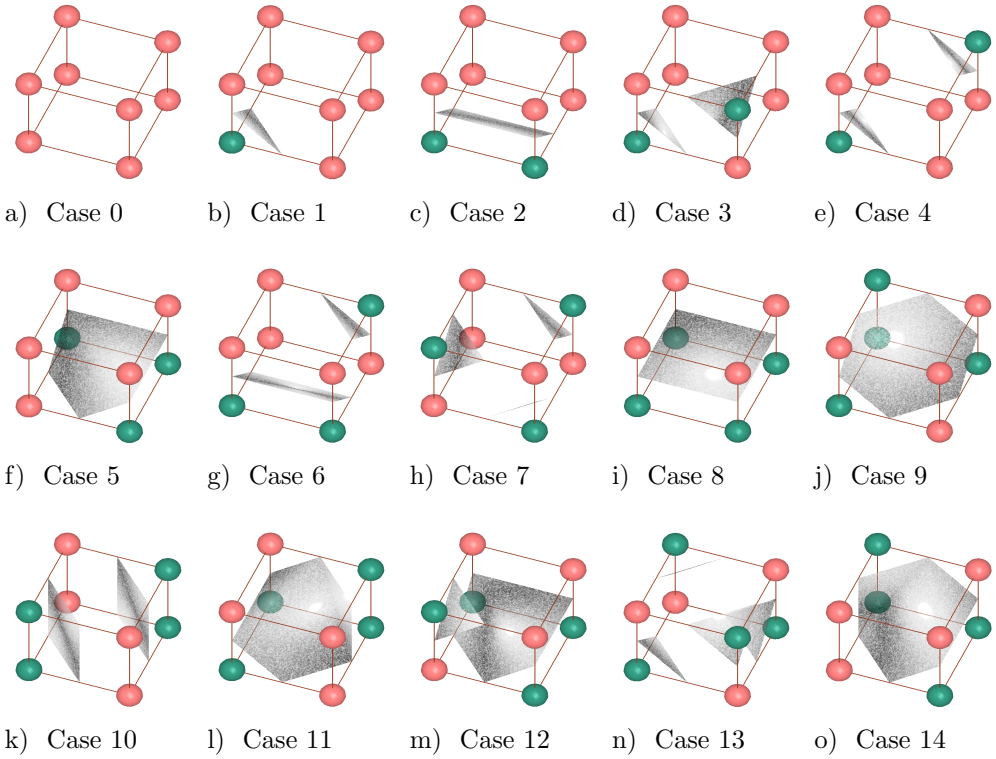


Figure 3-11. Fifteen topologically distinct patterns are used to tesselate a surface intersecting a cube. The green vertices are inside the surface, while the red vertices are outside. Each pattern separates the vertices inside the surface from outside the surface by a triangular mesh intersecting the edges [73], [87].

resolution is finite. This artifact is obvious when such a surface model is viewed from up close.

A higher-order interpolation could be used to improve the object's surface smoothness, so that the surface looked smooth even with close-up views. One of such higher-order interpolation methods was a tricubic interpolation based algorithm [88]. This method used a tricubic reconstruction to create a smooth surface from a binary discrete 3D data set, where a sampled voxel was only classified as either lying 'inside' or 'outside' the object. The tricubic interpolation function was based on a univariate cubic Hermite interpolation, where each vertex v of a cube in the discrete 3D data set was assigned 8 values: a distance value and 7 derivative values. The computation cost of the algorithm was that of 12 evaluations of univariate cubic polynomials and of 21 inner products of vectors of length

4 ([89] on page 76). Due to the computation complexity, this algorithm could not execute fast enough for an interactive application with deformable objects.

Instead of locally smooth the surface, other methods created a globally smooth surface from the binary volume data. These methods, such as ones based on the technique of constrained elastic surface nets [74], [90], were either iterative methods or required preprocessing [91] on the binary volume data, which have to be static. They did not run fast enough for a real-time application featuring unscripted deformable objects without special hardware, such as a special high-end graphics card or a parallel computer system.

In our study, the central difference method is used to estimate the gradients of the binary voxel data. Next we explains how our gradient estimation is computed.

3.6.2 Gradient Estimation

Since the voxel-based surface has no parameterization, the derivatives need to be approximated by a discrete method. The common gray-level computes the gradients of a given voxel based on the gray values of its local neighboring voxels by standard central difference. When applying the central difference method to our 3D regular grid data which are binary voxels that contain a value of either '0' or '1', the result is a poor quality approximation of derivatives. However, since the input mesh is a boundary representation, the exact interpolation value representing the intersection of each pair of the inside and outside voxels to the surface can be found. These data can be used to improve the geometry of the representation. The data are preprocessed and stored in the GPU's memory and later used by a visualization shader that uses the marching cubes to create and render the surface on the screen.

Based on the central difference method, the gradient of a voxel is approximated by the normalized sum of all vectors originating from the voxel's neighbors that are inside the model; while voxels that are outside the model do not contribute to the gradient

estimation (Figure 3-12). A similar gradient approximation method was also used by [52].

$$g_i = normalize(\sum_{j \in \text{neighbors of } i} v_i - v_j) \tag{3–16}$$

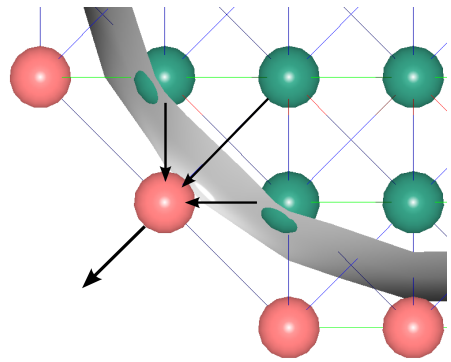


Figure 3-12. Gradient estimation for a voxel at the boundary

The gradient estimator suggested in the original marching cubes algorithm [87], uses only 3 directions (for a grey-scale intensity 3D data set), i.e., the axis aligned directions or face directions involving 6 neighbor voxels. However, every voxel, not on the boundary, is adjacent to 26 other voxels in the 3D regular grid data set. The gradient estimator, therefore, can be classified into the face directions (6 voxels), the vertex directions (8 voxels), and the edge directions (12 voxels). Our study found that the combination of all gradient estimators gave the best result (Figure 3-13).

3.6.3 Improving the Surface Smoothness

The smoothness of the surface obtained from the marching cubes method depends on the size of the 3D regular grid used. An algorithm for smoothing meshes called "curved PN triangles" was tested to improve the smoothness of the visualization. Curved point-normal triangles or PN triangles [92] is a subdivision-based algorithm for improving the visual quality of a triangular mesh. The algorithm took each triangle of the mesh and replaced the triangle with a curved patch and a higher-order normal variation. The flat geometry (vertex positions) of the triangle was substituted by the geometry of a three-sided cubic Bézier patch. The vertex normals of the triangle were replaced by the

normals of a three-sided quadratic Bézier patch. This algorithm was used to generate a smooth bumpy surface from the triangular mesh generated by the marching cubes algorithm (Figure 3-14);. The result showed that the algorithm improved the visualization of small 3D data sets. However, it degraded the visualization of bigger 3D data sets. This was caused by the triangular mesh generated from the marching cubes that contained irregular triangles. Therefore, in our deformable object modeling the curved PN triangles algorithm is only suitable for small 3D regular grid data sets.

Based on our study and design discussed above, we can start creating our deformable object simulator. Chapter 4 explains how to implement our deformable object simulator.

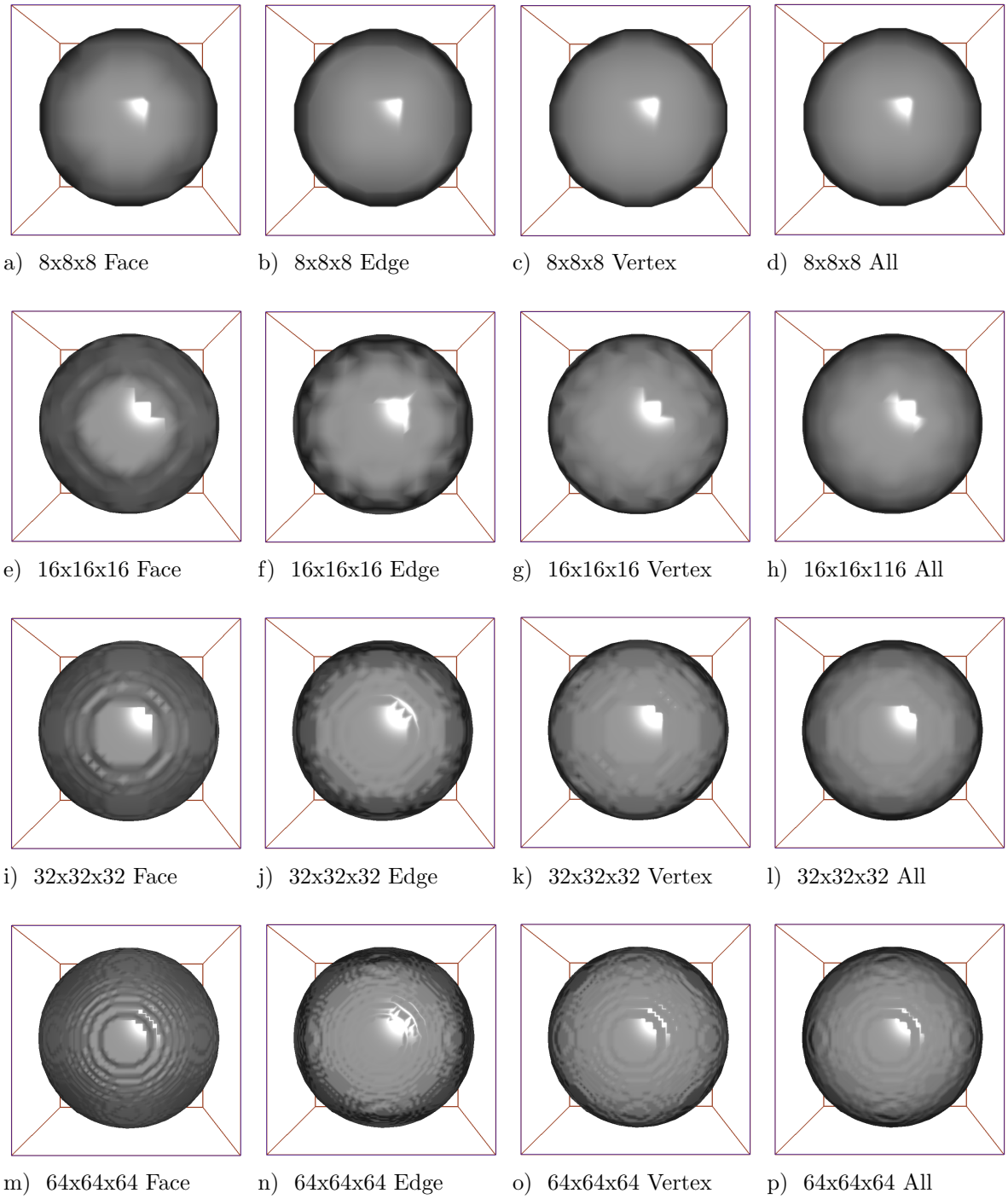
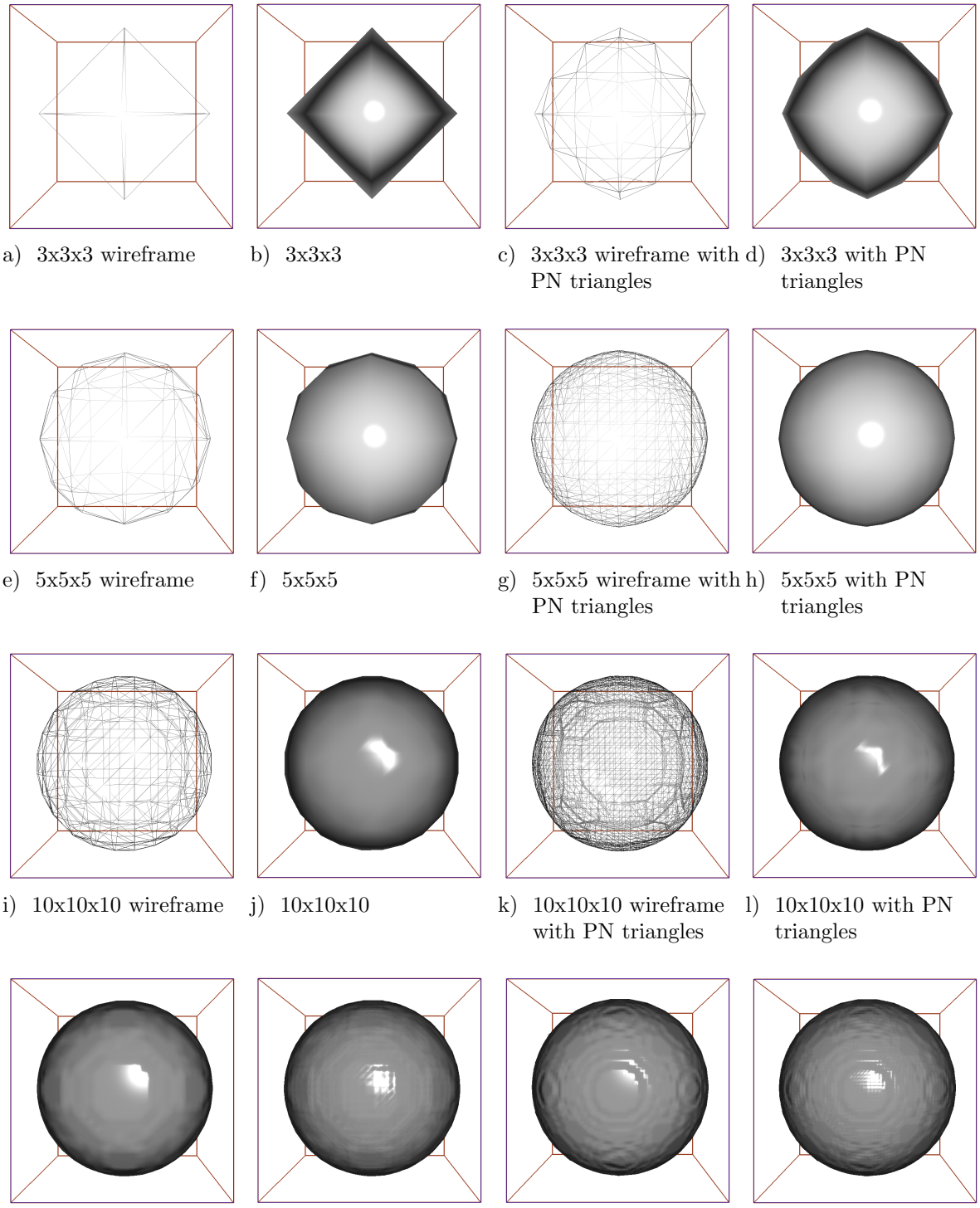


Figure 3-13. Comparison of gradient estimations; by the face, edge, vertex, and all directions.



a) 3x3x3 wireframe
b) 3x3x3
c) 3x3x3 wireframe with PN triangles
d) 3x3x3 with PN triangles
e) 5x5x5 wireframe
f) 5x5x5
g) 5x5x5 wireframe with PN triangles
h) 5x5x5 with PN triangles
i) 10x10x10 wireframe
j) 10x10x10
k) 10x10x10 wireframe with PN triangles
l) 10x10x10 with PN triangles
m) 32x32x32
n) 32x32x32 with PN triangles
o) 64x64x64
p) 64x64x64 with PN triangles

Figure 3-14. Visualization results from applying the curved PN triangles algorithm after the marching cubes algorithm.

CHAPTER 4
IMPLEMENTATION

Our deformable object simulator utilizes GPU computations to improve the performance. It uses all currently available shader types: vertex, geometry, and fragment shaders; for computing dynamic response, collision detection and visualization (Figure 4-1). The implementation of our deformable object simulator is in C++ with OpenGL 2.1 and OpenGL shading language (GLSL) 1.2. The simulator consists of three main representations: dynamic response, collision detection, and visualization explained in chapter 3. The dynamic response interacts with inside and outside simulative factors, such as connecting springs in the system and a gravitational force, involving both internal and external forces. The collision detection constrains the mass points in the system from penetrating other objects, such as a surgery tool, in the simulation environment. While the visualization interacts with the viewing environment, such as rendering specular highlights on the deformable object illuminated by a light source (Figure 4-2). Each representation, except the collision detection, consists of two modules. There are two collision detection representations: one for deformation by collision and another one for topology change by collision. First our data structure on the GPU is discussed followed by the implementation of each representation.

4.1 Data Structure on the GPU

With the Verlet integration, three 3D textures are needed for keeping the previous (input), current (input), and next (output) positions of the mass points. These textures are used as a circular buffer in each iteration. Even though, the connectivity of a mass point to its (adjacent) neighbors is implicitly defined in the 3D regular grid, an explicit connectivity is defined and kept in another 3D texture. This connectivity provides flexibility in data setup and ease in changing connectivity due to cutting by a surgical tool. Hence, at least four textures are needed: three for (previous, current, and next) positions and one for connectivity.

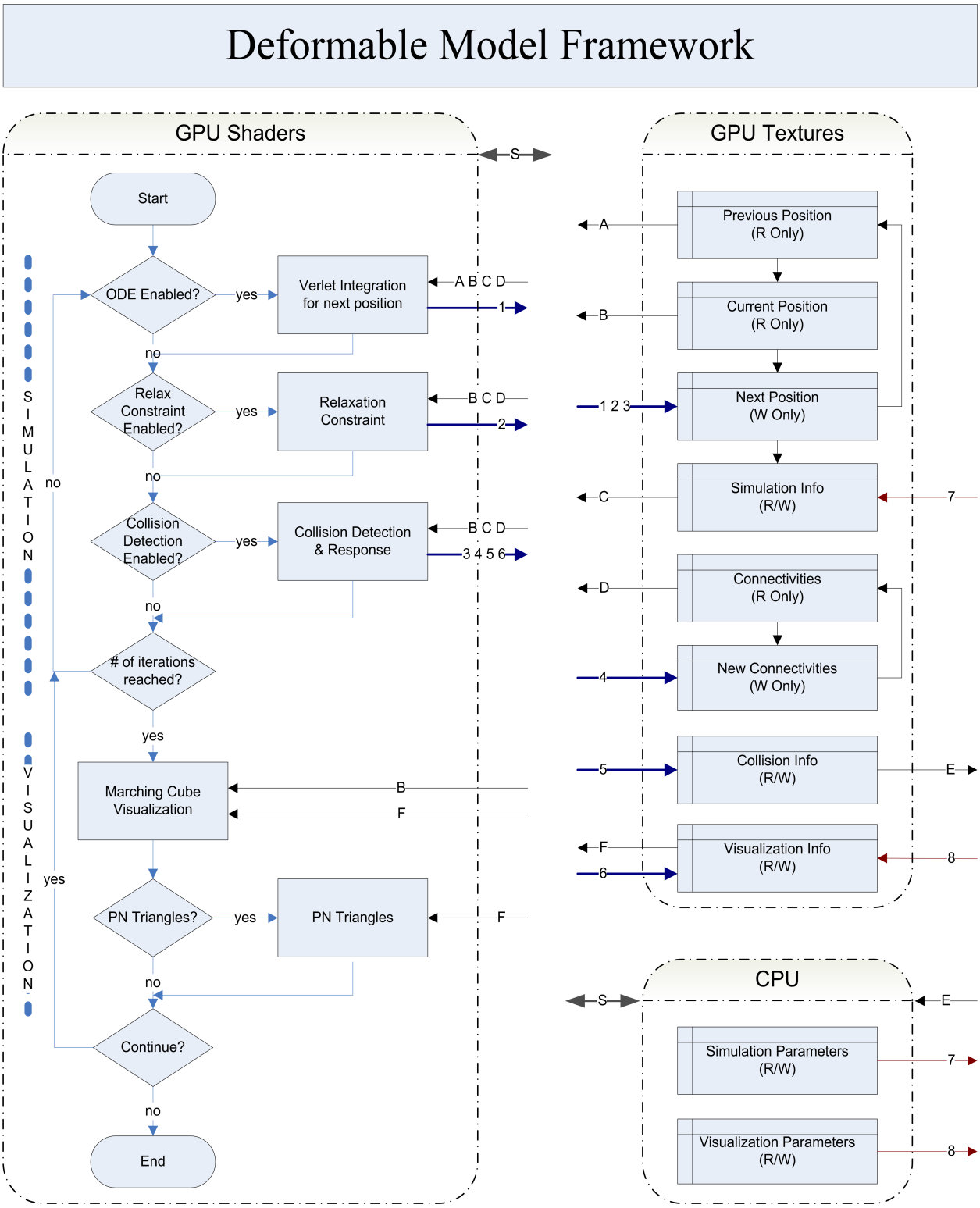
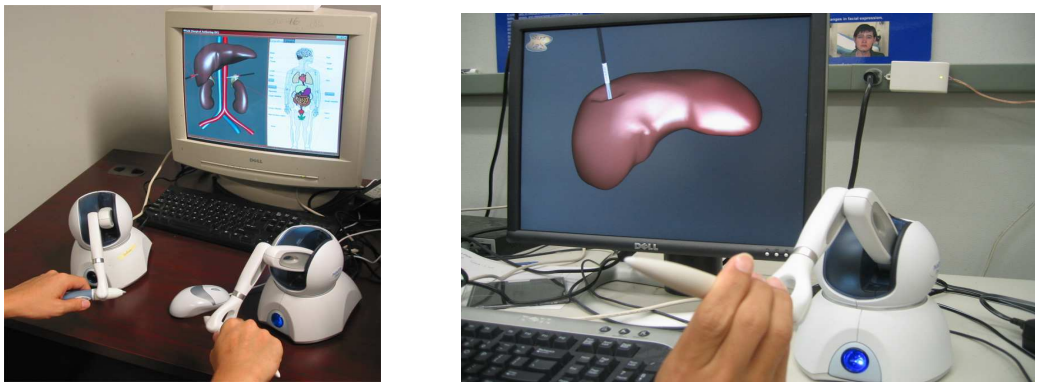
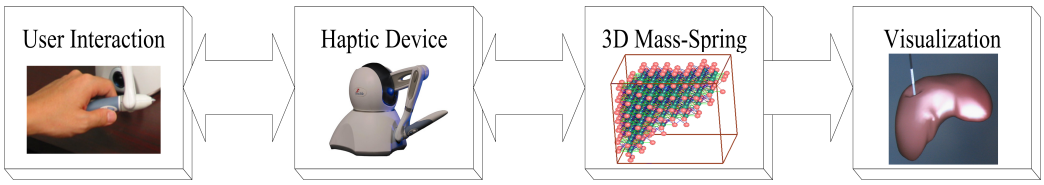


Figure 4-1. Deformable object framework diagram



a) user interaction b) visualization



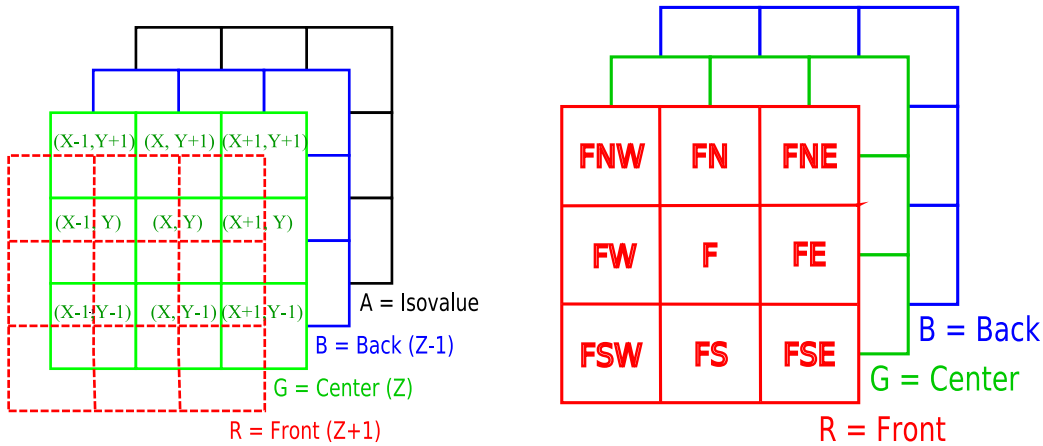
c) Abtraction of modeling process

Figure 4-2. Modeling process. User interaction and visualization are shown by a) and b).

Since each mass point can have up to 26 adjacent neighbors plus 1 for the home spring (Figure 4-3a), the mass point connectivity to its neighbor can be kept in a texture of size 3x3 texels (Figure 4-3b). Therefore, the width and height of the connectivity texture is three times bigger than a position texture. Each location, say FNW (Front-North-West), can be used for storing spring stiffness (and damping). Where a zero value represent no connection, and positive values represent spring stiffness. Therefore, this connectivity texture provides high flexibility for setting (3x3x3 = 27, last three is from RGB per texel) spring parameters per each mass point. This allows for different spring stiffnesses to be used for different regions in the model.

While the 3x3 connectivity data structure provides high flexibility for setting spring parameters, it is hardly needed in simulating a deformable object. Therefore, a more practical choice is to use only one (RGBA) texel for encoding the connectivity. Each component (R, G, B, and A) of the texel is a single 8-bit unsigned integer in the range of 0 to 255 (Figure 4-4). Each bit represents a flag bit for a connection; '0' or '1' for no

connection or connection, respectively. For example, bit 7 of the R component is set if the mass point is connected to its front-north-west (FNW) neighbor. In our implementation, this texture is used for keeping the connectivity of the mass-spring system. Next we can start creating our deformable object with this data structure.



a) position texture b) 3x3 connectivity texture

Figure 4-3. A mass point connectivity by 3x3 texels; a) the center mass point (x, y) of the position texture is associated with b) the 3x3 connectivity texture that provides nine RGB texels for looking up the connectivity of the center mass point to its 26 neighbors plus 1 for the home spring. Where C := Center, F := Front, B := Back, N := North, S := South, E := East, and W := West.

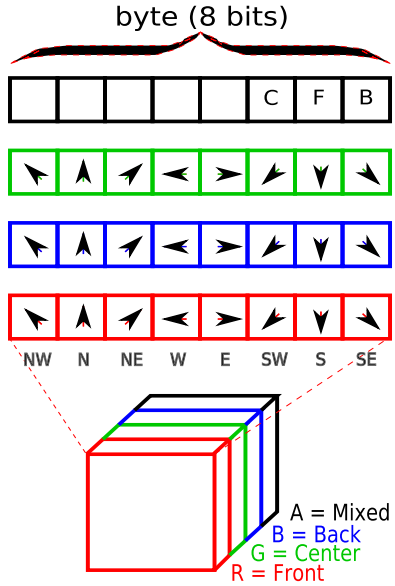


Figure 4-4. An RGBA texel for connectivities of a mass point to its neighbors.

To initialize our deformable object simulator, first the 3D voxel data representing a deformable object has to be loaded into the GPU's texture memory (or texture). Three textures are required for previous, current, and next positions, and one texture for connectivity among the mass-points in the system. These textures are needed by the dynamic response, consisting of the ODE solver and relaxation constraint adjustor, and the collision detection. Other textures are also needed: a texture for supporting collision detection, a texture for supporting simulation, and a texture for supporting visualization (Figure 4-1). Since a GPU texture cannot be bound as read and write at the same time, another texture for changing connectivity is needed for simulating a cut on the model. The simulator checks each connection from the read-only connectivity texture and removes the connection from the write-only connectivity texture if the connection is cut. The textures are then swapped, so that the new connectivity can take effect.

The same read/write situation is also needed to read the current and previous positions of each mass-point stored in a read-only texture and to write out its new position to a write-only texture. The previous and current positions are needed because the simulator's ODE solver is based on the Verlet integration. These three textures are, therefore, arranged as a circular buffer, cycling from the previous position texture to the current position texture, from the current position texture to the next position texture, and from the next position texture back to the previous position texture.

After the 3D data are loaded to the GPU's memory, the CPU can control the simulation, performing tasks such as to enable or disable the collision detection, and change the parameters of the mass-spring system, such as spring stiffness and the time step. In the next section we will show how each representation works.

4.2 Dynamic Response

In our deformable object simulator, the dynamic response is composed of two sequential modules: an ordinary differential equation (ODE) solver followed by a relaxation constraint adjustor.

4.2.1 Ordinary Differential Equation Solver

Our deformable object simulator is a mass-spring system based on the assumption that the forces are given or computable for all relevant interactions and positions. The initial position and velocity for each mass point in the system are also assumed to be known. The Verlet integration [80] is used in our deformable object simulator for solving the ODEs for the dynamic behavior of the 3D mass-spring system.

The Verlet integration equation (Equation 3–12 on page 40) for each mass-point in the mass-spring system is

$$\mathbf{x}_i(t+\Delta t) = \mathbf{F}_i \frac{\Delta t^2}{m_i} + 2\mathbf{x}_i(t) - \mathbf{x}_i(t-\Delta t) - k_d(\mathbf{x}_i(t) - \mathbf{x}_i(t-\Delta t)) \tag{4–1}$$

where $\mathbf{x}_i(t+\Delta t)$ is the new position of mass-point i resulting from the total force $\mathbf{F}_i$ applying to the mass-point. $\mathbf{x}_i(t)$ and $\mathbf{x}_i(t-\Delta t)$ are the current and previous position of mass-point i, respectively. Δt is the time step and m_i is the mass of mass-point i. k_d is a damping constant contained in the last term. The last term, $k_d(\mathbf{x}_i(t) - \mathbf{x}_i(t-\Delta t)$, is for damping the mass-point's movement to reduce an oscillation from applying a high force to the mass-point, which may cause the system dynamic to diverge.

4.2.2 Force Computation

In the simulation, the forces applied on a mass-point consist of internal $\mathbf{F}^{int}$ and external $\mathbf{F}^{ext}$ forces. The total force $\mathbf{F}$ exerted on mass point i is the sum of internal forces from connecting springs and external forces from outside interactions. An example of an external force is the force of gravity. The internal force acting on a mass-point in our mass-spring system is the summation of each force produced by the stretching or compressing of the spring that connects the mass-point to another mass-point in the system:

$$\mathbf{F}_i^{int} = \sum_{j \in \text{neighbors of } i} k_{ij}(l_{ij} - ||\mathbf{x}_i - \mathbf{x}_j||)\frac{\mathbf{x}_i - \mathbf{x}_j}{||\mathbf{x}_i - \mathbf{x}_j||} \tag{4–2}$$

Where k_{ij} is the stiffness of the spring connecting mass-point i to mass-point j. l_{ij} is the natural or rest length of the spring.

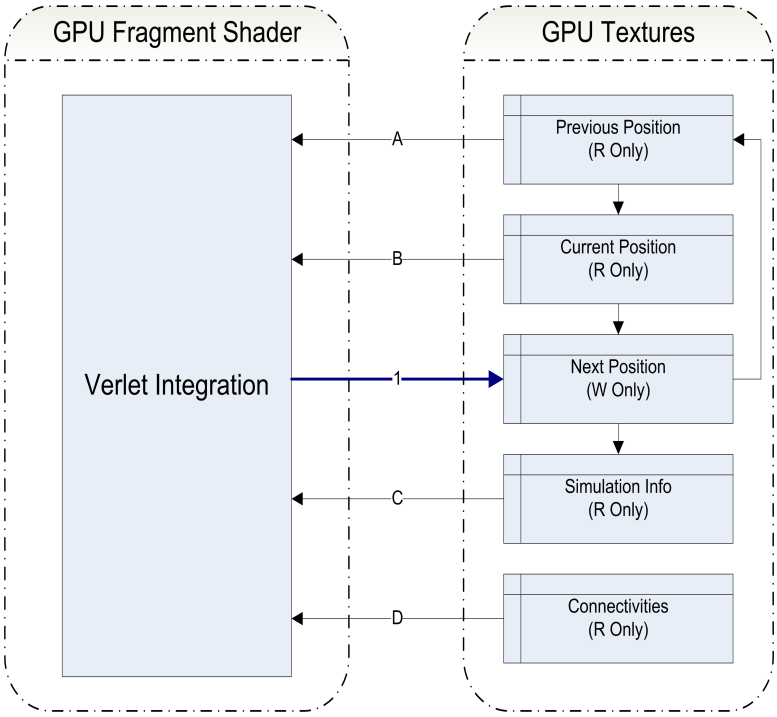


Figure 4-5. Simulation: ODE solver shader

A GPU fragment shader is set up to calculate the next position of each mass-point parallelly on the GPU (Figure 4-5). It computes both the internal and external force being exerted on each mass-point. The shader looks up the previous (-A-) and current (-B-) positions of a mass point from the 3D textures based on a texture coordinate identifying the point's logical location in the 3D lattice grid. The simulation parameters, such as spring stiffness constant and time step, are read (-C-) from Simulation Info texture. It also iteratively looks up the connectivity (-D-) of the mass point to the mass point's adjacent neighbors provided by the connectivity texture. If a connection exists, the shader fetches the (current) position of the connected neighbor and computes the internal force of the spring connecting the mass-point's current position to the neighbor. These internal forces are added based on the number of connected neighbors to the mass point. An external force, e.g., a gravitational force, can be added into this computation pass. For a collision with other objects, constraint-based displacement is used instead of estimating

the collision force (see Section 4.3 Collision Detection and Response). The next position of mass point i is computed by Equation 4–2 and written to (-1-) Next Position texture.

4.2.3 Relaxation Constraint Adjustor

After finding the new position of each mass-point, the simulator uses another shader to enforce a relaxation constraint on the distance of each mass-point to its neighbors in the system. The relaxation constraint shader (Figure 4-6) is similar to the ODE shader above (Figure 4-5). However, it does not need to know the previous position of the mass-point, only the current position is needed. The relaxation constraint is to limit all spring lengths to be within a min/max length threshold [29]. If a spring is extended too far, an offset vector is applied to both mass-points connected to each other by the spring to move them closer in order to reduce the length of the spring to within the maximum length threshold. If a spring is too compressed, an offset vector is applied to both mass-points connected to each other by the spring to move them farther apart in order to increase the length of the spring to within the minimum length threshold.

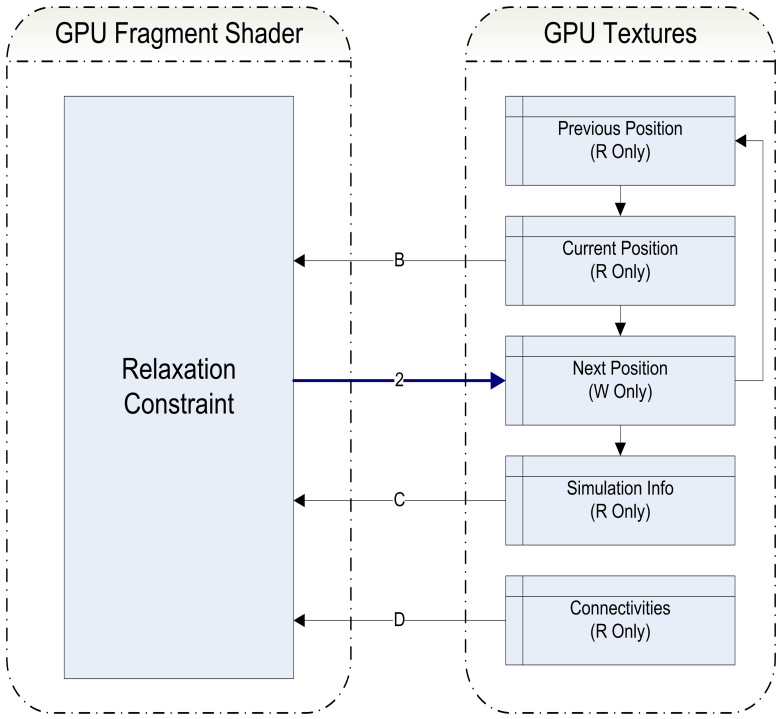


Figure 4-6. Simulation: Relaxation constraint shader

Our relaxation constraint shader does not process each spring. Instead, like the ODE shader, it processes each mass point. That means the current length of each spring is calculated twice, since two mass points are connected together by the spring. In order to process each spring only once on the GPU, the computation has to separate into two passes, since the GPU does not allow scatter write, i.e., the location to write to is fixed, The first computation pass computes each spring length, calculates the offset vector, and stores the offset vector in a texture. The second computation pass then moves each mass point according to the average sum of the associated offset vectors stored in the texture by the first computation pass.

To avoid the two computation passes, our relaxation constraint shader processes each mass point and finds the length of each spring that connects the mass point to another mass point. The shader sets an offset vector for each spring length and adds the offset vectors together, then averages the summation with the number of springs that connect to the mass point. The shader then uses the average offset vector to move the mass point in order to approximately preserve the length of all springs that connect to the mass point to the desired maximum or minimum length.

4.3 Collision Detection and Response

A surgical tool manipulated by a user will frequently collide with the simulated deformable object. Our collision detection is based on bounding volume intersections. Each mass point in the system is represented by a bounding sphere. The surgical tool can be represented by one or more bounding geometries. A GPU fragment shader is set up to perform this collision detection (Figure 4-7).

Without loss of generality, we assume the surgical tool is represented by a bounding sphere. The shader reads the current (-B-) position from Current Position Texture. It also reads the simulation parameters (-C-) from Simulation Info texture. The shader computes the intersection distance d between the bounding sphere S_i of the mass point with the

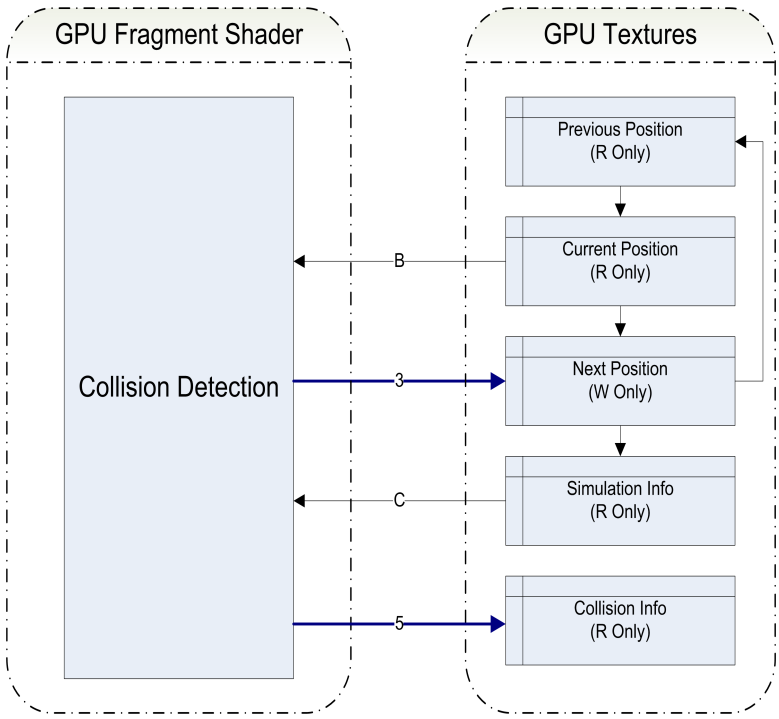


Figure 4-7. Simulation: Collision detection shader

bounding sphere S_t of the surgical tool.

$$d = R(S_i) + R(S_t) - ||C(S_i) - C(S_t)|| \tag{4–3}$$

where $C(S_n)$ and $R(S_n)$ are the center and radius of S_n, respectively. If d is less than zero, then the mass point intersects the surgical tool. The intersecting mass point is moved away from the surgical tool via the shortest distance $\vec{v}$ from the bounding sphere of the surgical tool.

$$\vec{v} = d\frac{C(S_i) - C(S_t)}{||C(S_i) - C(S_t)||} \tag{4–4}$$

The new position of the mass point ($\mathbf{x}_{t+t}$) is written to (-3-) Next Position texture. The change of the mass point's position ($\vec{d}$) is written to (-5-) Collision Info texture.

$$\vec{d} = \mathbf{x}_{t+t} - \mathbf{x}_t \tag{4–5}$$

where ($\mathbf{x}_t$) is the current position of the mass point. These data stored in Collision Info texture are for computing force feedback to the haptic device representing the surgical tool manipulated by user as described in Section 3.5 on page 43.

Another shader is needed for collision detection that create topology changes to the deformable object. This shader is more complicated than the previous collision detection shader (Figure 4-8). The shader needs to know the connectivity (-D-) of the mass point, so that it can change the connectivity (-4-) of the mass point. The shader is used for removing the mass points and/or the connections that intersect with the bounding sphere of the surgical tool.

The topology change on the mass-spring system will affect how the object is rendered by the visualization representation. The topology change due to collision response along with how to visualize it are described in Section 4.5 Topology Changes, after the implementation of the visualization has been described.

4.4 Visualization

In the deformable object simulator, the visualization is composed of two sequential modules: marching cubes followed by curved PN triangles (Figure 4-9).

4.4.1 Marching Cubes

Marching cubes [73] is a widely used algorithm for extracting a surface triangular mesh from a 3D grid of scalar data sampled from a scalar field. To implement the algorithm, a triangle and edge tables, for looking up the list of triangles that needs to be rendered according to the cube configuration value (Figure 3-11 on page 49), have to be set. These tables are obtained from Paul Bourke's Polygonising A Scalar Field [93].

The triangle and edge tables are kept in (-F-) Visualization Info texture. For each cube in the 3D regular grid, the shader reads eight voxels of the cube from (-B-) Current Position texture. A GPU geometry shader is used to determines the cube configuration value and uses the value to read the list of triangle from (-F-) the tables. For each triangle in the list, the shader computes the vertex positions, vertex estimated normals

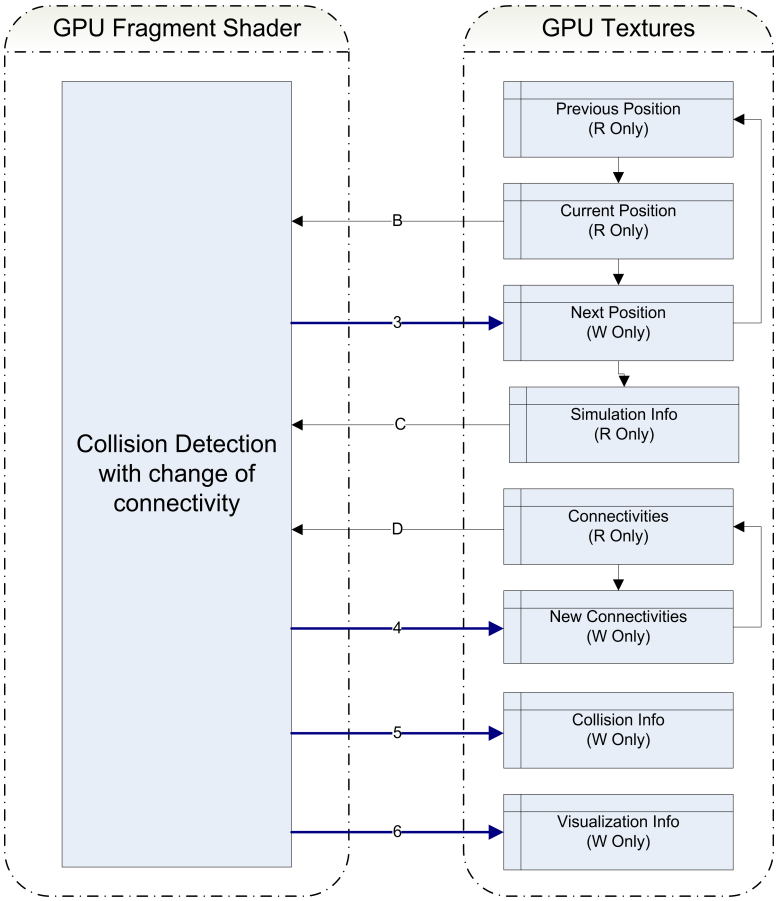


Figure 4-8. Simulation: Collision detection shader with the capability of making connectivity changes to the deformable object

(Equation 3–16 on page 51), and 3D texture coordinates for 3D texture mapping. The shader then renders the triangle to the screen.

For visualizing a logical cube that has been cut by a virtual tool, the marching cubes algorithm is modified to visualize the cut cube. The modified visualization is described in Section 4.5 Topology Changes.

4.4.2 Curved PN Triangles

As discussed in Section 3.6.3 on page 51, for small 3D regular grid data sets PN triangles [92] can be used to replace each triangle generated from the marching cubes algorithm with a three-sided cubic Bézier patch with quadratic normals. The algorithm is added into the geometry shader as a visualization option (see [92] for the algorithm detail).

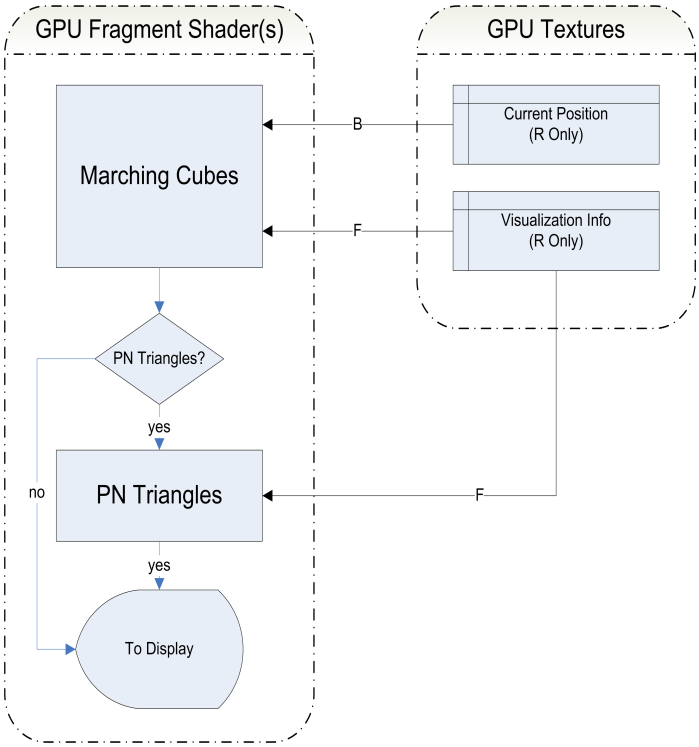


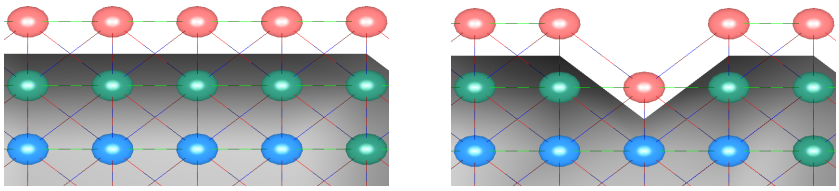
Figure 4-9. Visualization: Marching cubes and PN triangles shader

4.5 Topology Changes

As mentioned in Section 4.3, the topology of the object can be changed by the intersections of the virtual tool with the mass points and/or connections in the mass-spring system that represents the object.

For each removed mass point, each of its connections will be traversed in order to remove the connection from its neighbor, since the connection is symmetric (Figure 4-10). The removed mass point has no effect on the visualization by the marching cubes algorithm.

For each removed connection, the connection is removed from the connectivity of the two mass points that are connected by the connection. Since the removed connection represents a cut on the surface mesh, the visualization data has to be modified to display the cut. The cut is visualized by modifying the marching cubes algorithm to recognize the separation of the two mass points (Figure 4-11). In the dynamic responserepresentation,

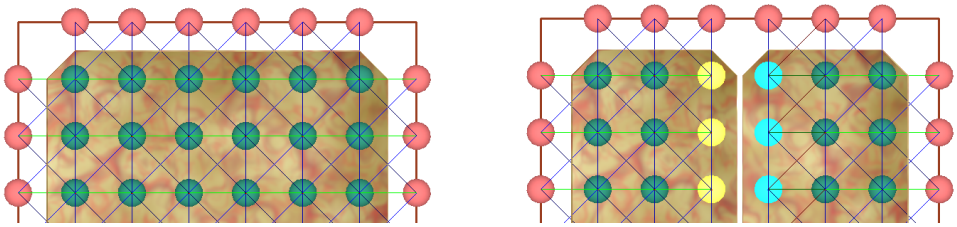


a) before the mass point is removed

b) after the mass point is removed

Figure 4-10. Removing mass points affect the surface visualization

the two mass points connected by a removed connection will be tagged with a 'left' and 'right' values (-6-). Assume the geometry of the cut is a rectangle. A connection is removed if it is intersected by the rectangle. The mass point connected by the connection that lies on the left (right) side of the rectangle will be tagged with 'left' ('right') value. The marching cubes is modified to create the triangles from the cube twice. The first (second) one treats all 'left' ('right') mass points as voxels inside the object and all 'right' ('left') points as voxels outside the object. The second one treats all 'right' mass points as voxels inside the object and all 'left' points as voxels outside the object. A similar approach was also used by Triquet et al. [94]. The implementation of the shader for the marching cubes algorithm is explained below.



a) before the connections are removed

b) after the connections are removed

Figure 4-11. Removing connections affect the surface visualization

CHAPTER 5
APPLICATION SCENARIO

An application (Figure 5-1) is created for developing and testing our deformable object simulator. The resulting deformable object can be saved and deployed in another application that requires such a deformable object (Figure 5-2).

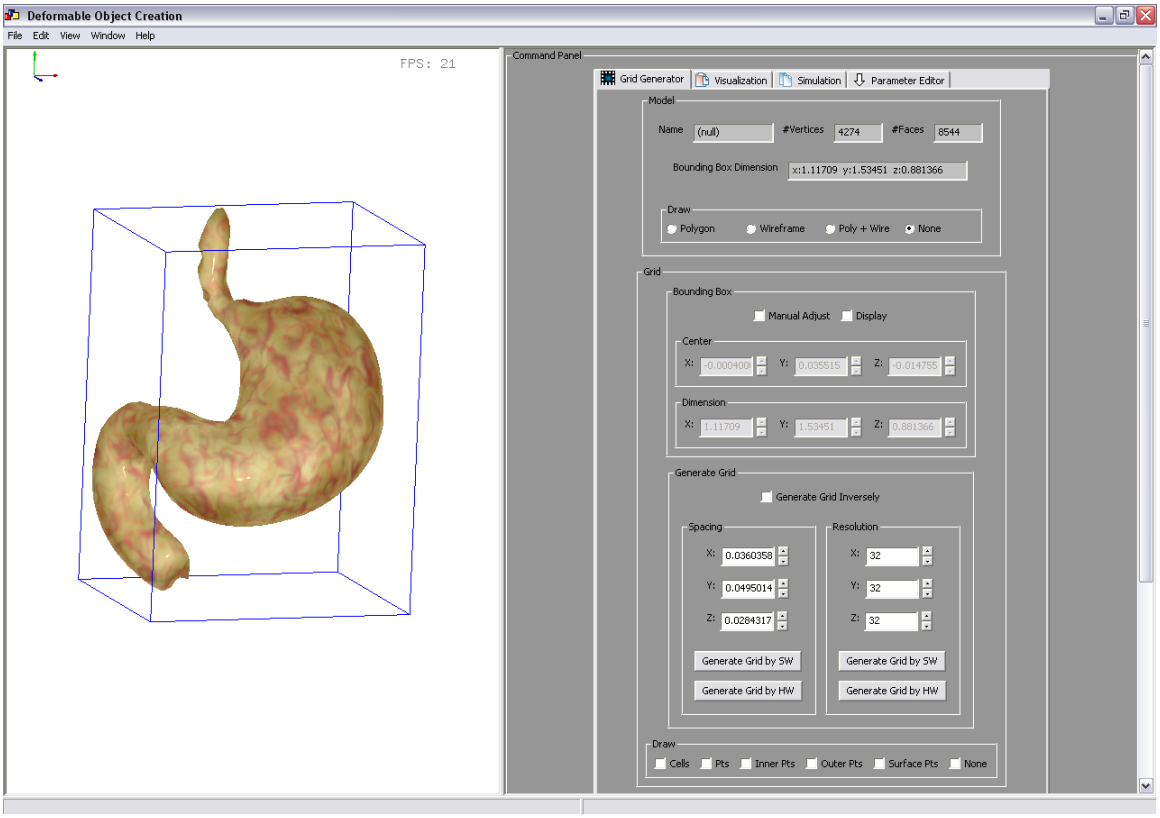
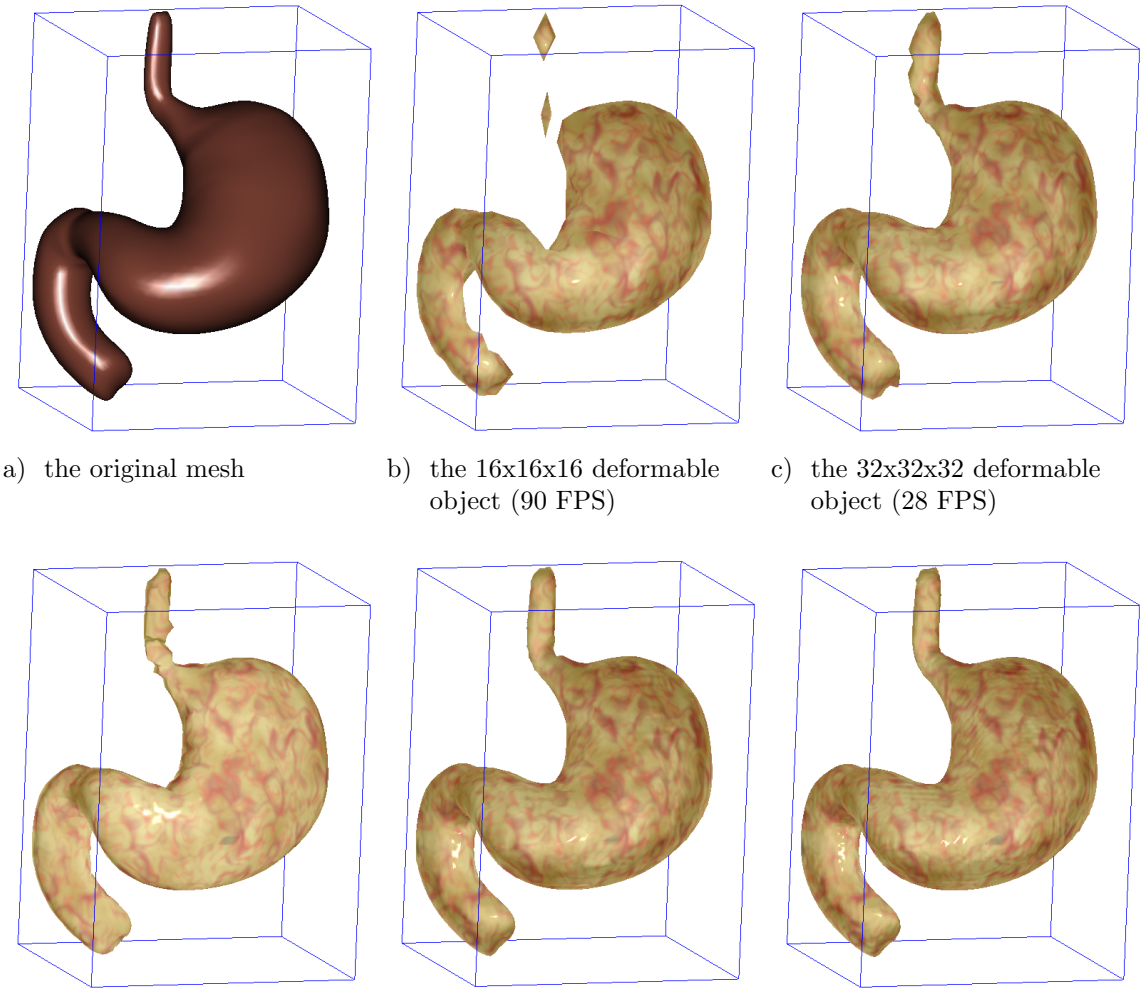


Figure 5-1. Application for creating deformable object

Currently our deformable object is used in TIPS. The haptic-enabled toolkit for illustration of procedures in surgery (TIPS) [7], [8] is a medical illustration and simulation for authoring a medical scene by an experienced surgeon to be repeatedly learned by medical students and residents. The application is created and developed by the SurfLab research group at the University of Florida (UF) in collaboration with other UF staff from the department of surgery.

The current case study under development in TIPS is the removal of the adrenal gland. This operation requires a safe dissection of the fatty tissue to expose the adrenal



a) the original mesh

b) the 16x16x16 deformable object (90 FPS)

c) the 32x32x32 deformable object (28 FPS)

d) the 40x60x20 deformable object (21 FPS)

e) the 64x64x64 deformable object (6.1 FPS)

f) the 100x100x100 deformable object (2.0 FPS)

Figure 5-2. Modeling a stomach

vein [8]. Our deformable object simulator is integrated into the software for simulating the fatty tissue (Figure 5-3).

In general, our deformable object can be used to model other soft tissue that need to be palpated, grasped, and cut by the virtual surgical tool manipulated by the user (Figure 5-4). The stomach model, with grid size of 32x32x32, runs at 40 fps on a computer equipped with an Intel Core 2 Duo 6600 2.4GHz, 1GB of memory, and a NVIDIA Geforce 8800 GTS 320MB graphics card.

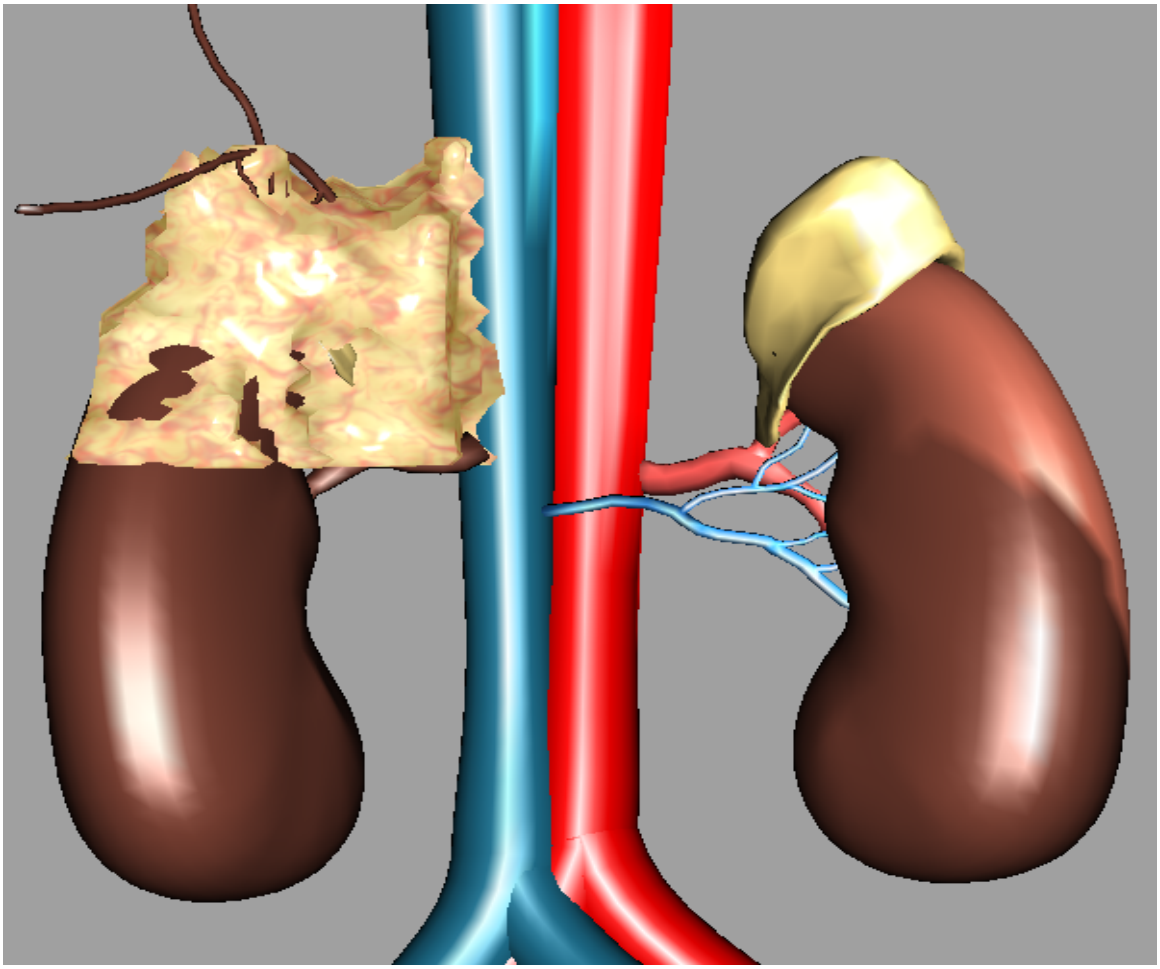
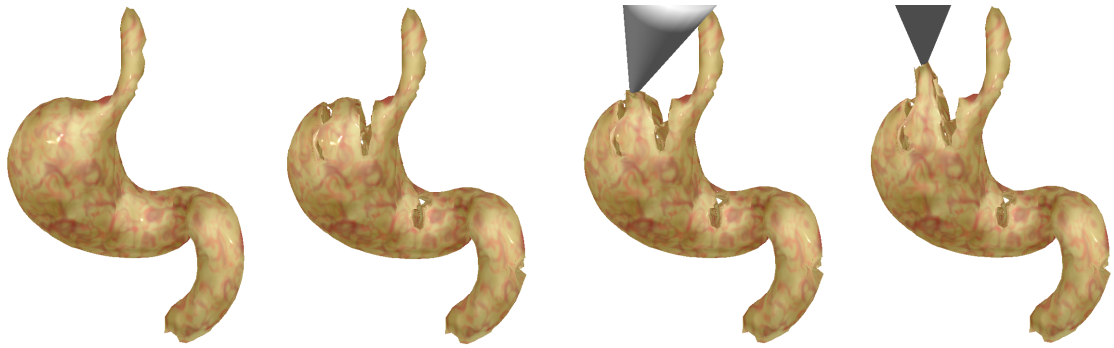


Figure 5-3. Fatty tissue on top of the left kidney is modeled by our deformable object



a) b) c) d)

Figure 5-4. Sequence of operations on the stomach; a) before cutting b) after a few cuttings c) initially grabbed d) grabbed and moved

CHAPTER 6
CONCLUSION

A deformable object simulator is implemented to simulate a dynamic behavior of a deformable object in real time. The deformable object simulator consists of three closely coupled models: dynamic response, collision detection, and visualization. The deformable object's dynamic response is modeled by a 3D regular grid of a mass-spring system. The interactions of the deformable object with other objects in the simulation environment are resolved by the collision detection model. The visualization of the deformable object is a surface triangular mesh generated in real time by a modified marching cubes algorithm. All three models are stored and computed in the GPU. All currently available GPU shaders, namely, vertex, geometry, and fragment shaders, are used to make the simulation and visualization of the deformable object run in real time. Trade-offs have to be made and are listed in order of importance from most to least important:

- Speed (computation cost and mathematical complexity),
- Robustness (stability and consistency)
- Flexibility (topology and material property changes),
- Geometry accuracy (resolution),
- Accuracy (realistic or plausible),
- Memory usage.

The deformable object is suitable, but not limited, for use in interactive surgical simulation. In TIPS [7], [8], the deformable object is used to simulate fatty tissues.

A number of improvements and extensions can be applied to the deformable object simulator. One extension is to make the deformable object simulator capable of modeling objects that are not fit well with the 3D regular grid, such as a thin torus. Instead of fitting a 3D regular grid to such an object, we can use a set of 3D regular grids to cover the object. For a thin torus four 3D regular grids can be used. This will reduce the memory size, since the set of 3D regular grids will take less space (i.e., memory) than one big 3D regular grid that fits the whole object. Each 3D regular grid will be fit with a mass-spring model. The deformable object simulator has to provide the interface among

the mass-spring models that are adjacent to one another. The extra computation cost will be compensated by the smaller number of mass points, in which translated into less pixels are executed in the GPU shaders.

REFERENCES

[1] PHANTOM®Omni™Haptic Device, http://www.sensable.com/haptic-phantom-omni.htm, 2007.

[2] C. R. Johnson and C. D. Hansen, *The Visualization Handbook*. Elsevier, 1st ed., 2004.

[3] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf, *Real-Time Volume Graphics*. A K Peters, 1st ed., 2006.

[4] A. Pentland and J. Williams, "Good Vibrations: Model Dynamics for Graphics and Animation," *Proc. 16th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '89)*, pp. 215–222, ACM Press, 1989.

[5] R. Barzel, J. F. Hughes, and D. N. Wood, "Plausible Motion Simulation for Computer Graphics Animation," *Computer Animation and Simulation '96*, pp. 183–197, 1996.

[6] R. Barzel, "Faking Dynamics of Ropes and Springs," *IEEE Computer Graphics and Applications*, vol. 17, pp. 31–39, May–June 1997. ISSN 0272-1716.

[7] M. Kim, S. Punak, J. Cendan, S. Kurenov, and J. Peters, "Exploiting Graphics Hardware for Haptic Authoring," *Medicine Meets Virtual Reality*, pp. 255–260, Studies in Health Technology and Informatics (SHTI), IOS Press, Amsterdam, 2006.

[8] M. Kim, T. Ni, J. Cendan, S. Kurenov, and J. Peters, "A Haptic-Enabled Toolkit for Illustration of Procedures in Surgery (TIPS)," *Medicine Meets Virtual Reality*, pp. 209–214, Studies in Health Technology and Informatics (SHTI), IOS Press, Amsterdam, 2007.

[9] N. J. Soper, L. L. Swanström, and W. S. Eubanks, *Mastery of Endoscopic and Laparoscopic Surgery*. Lippincott Williams & Wilkins, 2nd ed., 2005.

[10] U. G. Kühnapfel, H. K. Çakmak, and H. Maaß, "Endoscopic Surgery Training Using Virtual Reality and Deformable Tissue Simulation," *Computers and Graphics*, vol. 24, no. 5, pp. 671–682, 2000.

[11] G. Székely, C. Brechbühler, J. Dual, R. Enzler, J. Hug, R. Hutter, N. Ironmonger, M. Kauer, V. Meier, P. Niederer, A. Rhomberg, P. Schmid, G. Schweitzer, M. Thaler, V. Vuskovic, G. Tröster, U. Haller, and M. Bajka, "Virtual Reality-Based Simulation of Endoscopic Surgery," *Presence: Teleoperators and Virtual Environments*, vol. 9, pp. 310–333, June 2000.

[12] AllRefer.com, http://www.allrefer.com/, 2007.

[13] Y. C. Fung, *Biomechanics: Mechanical Properties of Living Tissues*. Springer, 2nd ed., Jan. 15 1993.

[14] A. Liu, F. Tendick, K. Cleary, and C. Kaufmann, "A Survey of Surgical Simulation: Applications, Technology, and Education," *Presence: Teleoperators and Virtual Environments*, vol. 12, no. 6, pp. 599–614, 2003.

[15] N. Ayache and H. Delingette, eds., *Surgery Simulation and Soft Tissue Modeling*, vol. 2673 of *Lecture Notes in Computer Science*, Springer-Verlag Inc., 2003. This book contains the written contributions to the Int'l Symp. on Surgery Simulation and Soft Tissue Modeling (IS4TM 2003).

[16] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically Deformable Models," *Proc. 14th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '87)*, pp. 205–214, ACM Press, 1987.

[17] D. Terzopoulos and K. Fleischer, "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture," *Proc. 15th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '88)*, pp. 269–278, ACM Press, 1988.

[18] D. Terzopoulos and A. Witkin, "Physically Based Models with Rigid and Deformable Components," *IEEE Computer Graphics and Applications*, vol. 20, issue 6, pp. 41–51, IEEE, Nov. 1988.

[19] D. Metaxas and D. Terzopoulos, "Dynamic Deformation of Solid Primitives with Constraints," *Proc. 19th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '92)*, pp. 309–312, ACM Press, 1992.

[20] S. F. F. Gibson and B. Mirtich, "A Survey of Deformable Modeling in Computer Graphics," tech. rep., Mitsubishi Electric Research Laboratories, Nov. 1997.

[21] T. W. Sederberg and S. R. Parry, "Free-Form Deformation of Solid Geometric Models," *Proc. 13th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '86)*, pp. 151–160, ACM Press, 1986.

[22] A. H. Barr, "Global and Local Deformations of Solid Primitives," *Proc. 11th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '84)*, pp. 21–30, ACM Press, 1984.

[23] S. Coquillart, "Extended Free-Form Deformation: a Sculpturing Tool for 3D Geometric Modeling," *Proc. 17th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '90)*, pp. 187–196, ACM Press, 1990.

[24] A. Rappoport, A. Sheffer, and M. Bercovier, "Volume-Preserving Free-Form Solids," *IEEE Trans. Visualization and Computer Graphics*, vol. 2, no. 1, pp. 19–27, 1996.

[25] G. Hirota, R. Maheshwari, and M. C. Lin, "Fast Volume-Preserving Free form Deformation Using Multi-Level Optimization," *Symp. Solid Modeling and Applications*, pp. 234–245, 1999.

[26] P. Faloutsos, M. van de Panne, and D. Terzopoulos, "Dynamic Free-Form Deformations for Animation Synthesis," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 3, pp. 201–214, 1997.

[27] S. Schein and G. Elber, "Discontinuous Free Form Deformations," *Pacific Conf. Computer Graphics and Applications*, pp. 227–236, 2004.

[28] G. Sela, S. Schein, and G. Elber, "Real-Time Incision Simulation Using Discontinuous Free Form Deformation," *ISMS*, pp. 114–123, 2004.

[29] X. Provot, "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior," *Proc. Graphics Interface Conf.*, pp. 147–154, May 1995.

[30] T. I. Vassilev, B. Spanlang, and Y. Chrysanthou, "Fast Cloth Animation on Walking Avatars," *Computer Graphics Forum*, vol. 20, no. 3, 2001.

[31] R. Bridson, R. Fedkiw, and J. Anderson, "Robust Treatment of Collisions, Contact and Friction for Cloth Animation," *Proc. 29th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '02)*, pp. 594–603, ACM Press, 2002.

[32] L. Chittaro and D. Corvaglia, "3D Virtual Clothing: from Garment Design to Web3D Visualization and Simulation," *Proc. Eighth Int'l Conf. 3D Web technology (Web3D '03)*, pp. 73–84, ACM Press, 2003.

[33] J. Lenoir, P. Meseure, L. Grisoni, and C. Chaillou, "Surgical Thread Simulation," *ESAIM: PROCEEDINGS*, vol. 12, pp. 102–107, November 2002.

[34] J. Lenoir, L. Grisoni, P. Meseure, Y. Rémion, and C. Chaillou, "Smooth Constraints for Spline Variational Modeling," *GRAPHITE*, pp. 58–64, 2004.

[35] F. Cordier, P. Volino, and N. Magnenat-Thalmann, "Integrating Deformations Between Bodies and Clothes," *J. Visualization and Computer Animation*, vol. 12, no. 1, pp. 45–53, 2001.

[36] L. P. Nedel and D. Thalmann, "Real Time Muscle Deformations Using Mass-Spring Systems," *Proc. Computer Graphics Int'l*, F.-E. Wolter and N. M. Patrikalakis, eds., pp. 156–165, IEEE Computer Society, June 22–26 1998.

[37] Y. Zhang, E. C. Prakash, and E. Sung, "Real-Time Physically-Based Facial Expression Animation Using Mass-Spring System," *Proc. Computer Graphics Int'l*, pp. 347–350, IEEE Computer Society, 2001.

[38] Y. C. Fung and P. Tong, *Classical and Computational Solid Mechanics*. World Scientific Publishing Company, 1st ed., Oct. 2001.

[39] J. Georgii and R. Westermann, "Mass-Spring Systems on the GPU," *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 693–702, 2005.

[40] P. Volino and N. Magnenat-Thalmann, "Comparing Efficiency of Integration Methods for Cloth Simulation," *Proc. Computer Graphics Int'l*, pp. 265–274, 2001.

[41] M. Hauth, O. Etzmuss, and W. Strasser, "Analysis of Numerical Methods for the Simulation of Deformable Models," *The Visual Computer*, vol. 19(7-8), pp. 581–600, Springer, Dec. 2003.

[42] E. Keeve, S. Girod, and B. Girod, "Craniofacial Surgery Simulation," *Proc. Fourth Int'l Conf. on Visualization in Biomedical Computing (VBC '96)*, pp. 541–546, 1996.

[43] D. Bourguignon and M.-P. Cani, "Controlling Anisotropy in Mass-Spring Systems," *Proc. Eurographics Workshop Computer Animation and Simulation (EGCAS '00)*, N. Thalman-Magnenat, D. Thalman, and B. Arnaldi, eds., pp. 113–124, Springer, Aug. 21–22 2000.

[44] R. Webster, R. Haluck, R. Ravenscroft, B. Mohler, E. Crouthamel, T. Frack, S. Terlecki, and J. Sheaffer, "Elastically Deformable 3D Organs for Haptic Surgical Simulation," *Medicine Meets Virtual Reality*, pp. 570–572, 2002.

[45] K. S. Choi, H. Sun, P. A. Heng, and J. C. Y. Cheng, "A Scalable Force Propagation Approach for Web-Based Deformable Simulation of Soft Tissues," *Proc. Seventh Int'l Conf. 3D Web technology (Web3D '03)*, pp. 185–193, ACM Press, Feb. 24–28 2002.

[46] K.-S. Choi, H. Sun, P.-A. Heng, and J. Zou, "Deformable Simulation Using Force Propagation Model with Finite Element Optimization," *Computers and Graphics*, vol. 28, no. 4, pp. 559–568, 2004.

[47] C. Bruyns and M. P. Ottensmeyer, "The Development of a Physically Based Virtual Animal Model Using Soft-Tissue Parameter Testing," Aug. 25 2002.

[48] J. Georgii, F. Echtler, and R. Westermann, "Interactive Simulation of Deformable Bodies on GPUs," *Proc. Simulation and Visualisation 2005*, pp. 247–258, 2005.

[49] J. Georgii and R. Westermann, "A Multigrid Framework for Real-Time Simulation of Deformable Volumes," *Proc. Second Workshop Virtual Reality Interaction and Physical Simulation*, pp. 50–57, 2005.

[50] J. Georgii and R. Westermann, "Interactive Simulation and Rendering of Heterogeneous Deformable Bodies," *Proc. Vision, Modeling and Visualization*, pp. 383–390, 2005.

[51] J. Mosegaard, P. Herborg, and T. S. Sørensen, "A GPU Accelerated Spring Mass System for Surgical Simulation," *Medicine Meets Virtual Reality*, vol. 111, pp. 342–348, January 2005.

[52] J. Mosegaard and T. S. Sørensen, "GPUAccelerated Surgical Simulators for Complex Morphology," *Proc. IEEE Conf. Virtual Reality (VR '05)*, pp. 147–154, 323, IEEE Computer Society, March 2005.

[53] T. S. Sørensen and J. Mosegaard, "An Introduction to GPU Accelerated Surgical Simulation," *Biomedical Simulation: Third Int'l Symp., (ISBMS '06)*, M. Harders and G. Szkely, eds., vol. 4072 of *Lecture Notes in Computer Science*, pp. 93–104, Springer Berlin / Heidelberg, 2006.

[54] T. M. Atanackovic and A. Guran, *Theory of Elasticity for Scientists and Engineers.* Birkhauser, 1st ed., May 2000.

[55] A. Alexandrou, *Principles of Fluid Mechanics.* Prentice Hall, 1st ed., Jan. 15 2001.

[56] P. Tong and J. N. Rossettos, *Finite-Element Method: Basic Technique and Implementation.* The MIT Press, 1st ed., Jan. 15 1977.

[57] T. J. R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis.* Prentice Hall, 1st ed., 1987.

[58] S. S. Rao, *The Finite Element Method in Engineering.* Butterworth-Heinemann, 3rd ed., 1999.

[59] M. Bro-Nielsen and S. Cotin, "Real-Time Volumetric Deformable Models for Surgery Simulation Using Finite Elements and Condensation," *Computer Graphics Forum*, vol. 15, pp. 57–66, Aug. 1996. Proc. Eurographics '96. ISSN 1067-7055.

[60] M. Bro-nielsen, "Finite Element Modeling in Surgery Simulation," Apr. 03 1998.

[61] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler, "Stable Real-Time Deformations," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation (SCA '02)*, pp. 49–54, ACM Press, 2002.

[62] M. Müller, M. Teschner, and M. Gross, "Physically-Based Simulation of Objects Represented by Surface Meshes," *Proc. Computer Graphics Int'l*, pp. 26–33, IEEE Computer Society, 2004.

[63] I. A. Essa, S. Scarloff, and A. Pentland, "A Unified Approach for Physical and Geometric Modeling for Graphics and Animation," Technical Report 179, M.I.T. Media Lab Vision and Modeling Group, Dec. 1991. 17 pages.

[64] D. L. James and D. K. Pai, "DyRT: Dynamic Response Textures for Real Time Deformation Simulation with Graphics Hardware," *Proc. 29th Ann. Conf. Computer Graphics and Interactive techniques (SIGGRAPH '02)*, pp. 582–585, ACM Press, 2002.

[65] D. L. James and D. K. Pai, "ArtDefo: Accurate Real Time Deformable Objects," *Proc. 26th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '99)*, pp. 65–72, ACM Press/Addison-Wesley Publishing Co., 1999.

[66] I. F. Costa and R. Balaniuk, "LEM - An Approach for Real Time Physically Based Soft Tissue Simulation," *Proc. IEEE Int'l Conf. Robotics and Automation*, pp. 2337–2343, 2001.

[67] J. Teran, S. Blemker, V. N. T. Hing, and R. Fedkiw, "Finite Volume Methods for the Simulation of Skeletal Muscle," *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation (SCA '03)*, pp. 68–74, Eurographics Association, 2003.

[68] G. Debunne, M. Desbrun, M.-P. Cani, and A. H. Barr, "Dynamic Real-Time Deformations Using Space & Time Adaptive Sampling," *Proc. 28th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '01)*, pp. 31–36, ACM Press, 2001.

[69] Y.-J. Lim, J. Hu, C.-Y. Chang, and N. Tardella, "Soft Tissue Deformation and Cutting Simulation for the Multimodal Surgery Training," *Proc. 19th IEEE Symp. Computer-Based Medical Systems (CBMS '06)*, pp. 635–640, IEEE Computer Society, 2006.

[70] C. Basdogan, M. Sedef, M. Harders, and S. Wesarg, "VR-Based Simulators for Training in Minimally Invasive Surgery," *IEEE Computer Graphics and Applications*, vol. 27, no. 2, pp. 54–66, 2007.

[71] M. LeDuc, S. Payandeh, and J. Dill, "Toward Modeling of a Suturing Task," *Proc. Graphics Interface Conf.*, pp. 273–279, CIPS, Canadian Human-Computer Commnication Society, A K Peters, June 2003. ISBN 1-56881-207-8, ISSN 0713-5424.

[72] H. Zhang, S. Payandeh, and J. Dill, "Simulation of Progressive Cutting on Surface Mesh Model," Oct. 07 2002.

[73] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Proc. 14th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '87)*, pp. 163–169, ACM Press, 1987.

[74] S. F. F. Gibson, "Constrained Elastic Surface Nets: Generating Smooth Surfaces from Binary Segmented Data," *Proc. First Int'l Conf. Medical Image Computing and Computer-Assisted Intervention (MICCAI '98)*, pp. 888–898, Springer-Verlag, 1998.

[75] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 1990.

[76] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *STAR Proc. Eurographics 2004*, C. Schlick and W. Purgathofer, eds., pp. 21–51, Eurographics Association, Sept. 2004.

[77] J. Krüger, "GPGPU and raytracing," *ACM SIGGRAPH 2007 courses (SIGGRAPH '07)*, p. 12, ACM Press, 2007.

[78] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 1.0*, June 2007.

[79] R. Strzodka, M. Doggett, and A. Kolb, "Scientific Computation for Simulations on Programmable Graphics Hardware," *Simulation Modelling Practice and Theory, Special Issue: Programmable Graphics Hardware*, vol. 13, pp. 667–680, Nov 2005.

[80] L. Verlet, "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules," *Physical Review*, vol. 159, p. 98, Jul 1967.

[81] L. Hilde, P. Meseure, and C. Chaillou, "A Fast Implicit Integration Method for Solving Dynamic Equations of Movement," *Proc. ACM Symp. Virtual Reality Software and Technology (VRST '01)*, pp. 71–76, ACM Press, 2001.

[82] D. Baraff and A. Witkin, "Large Steps in Cloth Simulation," *Proc. 25th Ann. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH '98)*, pp. 43–54, ACM Press, 1998.

[83] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner, "Fast Volumetric Deformation on General Purpose Hardware," *Graphics Hardware*, K. Akeley and U. Neumann, eds., pp. 17–24, Eurographics Association, 2001.

[84] R. Westermann and C. Rezk-Salama, "Real-Time Volume Deformations," *Computer Graphics Forum*, vol. 20, no. 3, pp. 443–451, 2001.

[85] K. Erleben, J. Sporring, K. Henriksen, and H. Dohlmann, *Physics-Based Animation*. Charles River Media, 1st ed., 2005.

[86] J. Mosegaard and T. S. Sørensen, "Real-Time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU," *Proc. Eurographics Workshop Virtual Environments*, vol. 11, pp. 105–111, Eurographics Association, 2005.

[87] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, and B. Teeter, "Two Algorithms for the Reconstruction of Surfaces from Tomograms," *Medical Physics*, vol. 15, pp. 320–327, 1988.

[88] A. Kadosh, D. Cohen-Or, and R. Yagel, "Tricubic Interpolation of Discrete Surfaces for Binary Volumes," *IEEE Trans. Visualization and Computer Graphics*, vol. 9(4), pp. 580–586, 2003.

[89] M. Chen, A. E. Kaufman, and R. Yagel, *Volume Graphics*. Springer, 1st ed., 2000.

[90] R. T. Whitaker, "Reducing Aliasing Artifacts in Iso-Surfaces of Binary Volumes," *Proc. IEEE Symp. Volume visualization (VVS '00)*, pp. 23–32, ACM Press, 2000.

[91] M. Shimizu and Y. Nakamura, "Constructing Detailed Solid and Smooth Surfaces from Voxel Data for Neurosurgical Simulation," *Computational Science and Its Applications (ICCSA '05)*, O. Gervasi, M. L. Gavrilova, V. Kumar, A. Laganà, H. P. Lee, Y. Mun, D. Taniar, and C. J. K. Tan, eds., vol. 3482 of *Lecture Notes in Computer Science*, pp. 1013–1022, Springer, 2005.

[92] A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell, "Curved PN Triangles," *Proc. Symp. Interactive 3D graphics (I3D '01)*, pp. 159–166, ACM Press, 2001.

[93] P. Bourke, "Polygonising a Scalar Field," http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/, 1994.

[94] F. Triquet, L. Grisoni, P. Meseure, and C. Chaillou, "Realtime Visualization of Implicit Objects with Contact Control," *Proc. First Int'l Conf. Computer Graphics and Interactive Techniques in Australasia and South East Asia (GRAPHITE '03)*, pp. 189–196, ACM Press, 2003.

BIOGRAPHICAL SKETCH

Sukitti Punak, a Bangkok native, received his vocational graduate certificate in electronics from King Mongkut's Institute of Technology North Bangkok (Bangkok, Thailand), B.E. in electrical engineering from King Mongkut's University of Technology Thonburi (Bangkok, Thailand), M.S. in computer science from Chulalongkorn University (Bangkok, Thailand), and M.S. in electrical engineering from Georgia Institute of Technology (Georgia, USA). He pursued his Ph.D. study to fulfill his career passion of becoming a computer graphics researcher. His main research interest is in computer graphics for animation, rendering, and visualization.