PHYSICALLY BASED MODELING AND SIMULATION FOR VIRTUAL ENVIRONMENT BASED SURGICAL TRAINING

by

SURIYA NATSUPAKPONG

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Dissertation Advisor: M. Cenk Çavuşoğlu, Ph.D.

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

January, 2010

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Suriya Natsupakpong

candidate for the Doctor of Philosophy degree *.

(signed) M. Cenk Çavuşoğlu
(chair of the committee)

Guo-Qiang Zhang

Vira Chankong

Shudong Jin

Alan R. Cohen

(date) July 30, 2009

*We also certify that written approval has been obtained for any proprietary material contained therein.

To My Family

Contents

List of Tables

List of Figures

Acknowledgements

The completion of this research could not have been possible without the kind assistance and cooperation of a number of people.

My most profound gratitude goes to my advisor Cenk Çavuşoğlu. He not only provided me immense academic supports, but he also gave me the valuable suggestions and comments on this dissertation.

I am also grateful for the other members of my committee, Guo-Qiang Zhang, Shudong Jin, Vira Chankong, and Alan R. Cohen, for their valuable suggestions and comments on this dissertation.

I am also thank to Vincenzo Liberatore and Qingbo Cai for collaboration in GiPSiNet.

I am also indebted to the Development and Promotion of Science and Technology Talents Project (DPST), Royal Government of Thailand scholarship, for supporting me from high school until completeness of this dissertation, without the support of DPST, I would not have had wonderful opportunities.

I am grateful for my parents, my sister and brother, and my relations for their support throughout my long education life.

Especially, I would like to sincerely thank to Chansiri Singhtaun, whose encouragement, and love enabled me to complete this work.

This research was done in the Medical Robotics and Computer Integrated Surgery Laboratory at Case Western Reserve University with partial support from National

Science Foundation Grants CNS-0423253, IIS-0805495, US Department of Commerce grant TOP-39-60-04003, Rainbow Foundation, and Cleveland Foundation.

Physically Based Modeling and Simulation for Virtual Environment based Surgical Training

Abstract

by

SURIYA NATSUPAKPONG

Traditional medical education has relied on training with real patients in actual clinical setting under the supervision of an experienced surgeon. Novice surgeons can make mistakes that result in risks to patient safety. Computer simulation-based training has been proposed to complement traditional training to improve patient safety and surgeon efficiency and reduce cost and time. Surgical simulation allows surgeons to learn, practice and repeat surgical procedures to gain experience in a realistic and safe environment.

This dissertation focuses on the development of computer-based surgical simulations. Physically based modeling is used to model deformable objects to mimic human organs in simulation. Such a simulation is composed of many simulation objects whose behaviors are represented by differential equations. The system of differential equations can be solved by using numerical integration algorithms. Moreover, physical intersections between the objects require the computation of collision detection and response between objects. This dissertation studies the determination of elasticity parameters in lumped element models, the trade-offs in numerical integration algorithms for finding the suitable numerical integration algorithms and time step size of simulation objects, and the collision detection and response algorithms for deformable objects. Improvements and extensions of the open source/open architecture GiPSi surgical simulation framework are also presented. An endoscopic third ventriculostomy simulator is constructed using the GiPSi framework as a test bed of the specific tools and methods developed.

Chapter 1

Introduction

Traditional medical education has relied on training with real patients in actual clinical setting under the supervision of an experienced surgeon. Novice surgeons can make mistakes that result in risks to patient safety. Moreover, the traditional apprenticeship-based training is time consuming and costly. Simulation-based training has been proposed to complement traditional apprenticeship-based training in surgery to improve patient safety, reduce cost and improve efficiency. Surgical simulation allows surgeons to learn, practice and repeat surgical procedures to gain experience in a realistic and safe environment. Moreover, surgical simulation provides an opportunity to practice technical and problem-solving skills in a short period of time.

Surgical simulation can be classified into three groups [5], which are model-based simulation, computer-based simulation, and hybrid simulation. In model-based simulation, special materials, such as latex and silicon, are used to reproduce a part of human body to simulate the specific surgical procedure. This kind of simulation is used in many hospitals for practicing simple surgical procedures. The drawback of model-based simulation is that it is not interactive and hence does not give the surgeon feedback like real procedures. In computer-based simulation, virtual envi-

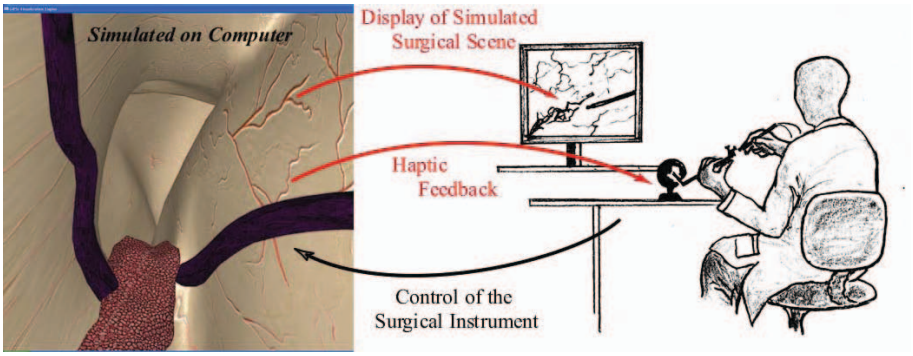


Figure 1.1: Computer-based surgical simulator concept.

ronments that mimic human organ behavior, along with the force feedback from the haptic devices, are used. Computer-based simulation is an active research area, e.g. [6, 7, 8, 9, 10, 11]. Hybrid simulation uses a combination of physical model-based and computer-based to make a complete simulation with look and feel like a real surgery. The example of hybrid simulation is the ProMIS system [12].

This study focuses on a development of computer-based surgical simulations, which are physically based modeling to model deformable objects used in simulation to mimic human organs. Such a simulation is composed of many simulation objects whose behaviors are represented by differential equations. The system of differential equations can be solved by using numerical integration. Moreover, the intersection between the objects in the virtual environment requires modeling and computation of collision detection and response between objects. This work is based on GiPSi framework and extends the functionality into GiPSi as well as developing the endoscopic surgical simulator as the test bed.

1.1 Surgical Simulator Concept

The objective of a computer-based surgical simulator is to help novice surgeons to practice surgical procedures to improve their skills (Figure 1.1). This means that the computer-based surgical simulator should have the following requirements:

(a) The simulator should give the realistic behaviors. The simulation contains many simulation objects and their interactions. Therefore, each simulation object should mimic the natural behaviors of real biological tissue. Physically based models use the laws of physics to model and simulate deformable object behaviors. It is also necessary to efficiently and realistically simulate the physical interactions between objects by employing suitable collision detection and response algorithms.

(b) The simulator should give the tactile feedback. By using a haptic technology, the simulator can calculate a force feedback from simulation object interaction. And then, send a result to haptic device to render force to a surgeon to sense the tool-tissue forces.

(c) The simulator should give the realistic visualization. By using computer graphic technology, the simulator can create a realistic environment in simulation.

(d) The simulator should maintain the numerical stability. The simulation uses numerical integration methods to simulate the behavior of physical models. Ensuring numerical stability of the integration methods employed is an essential requirement as numerical instability is a common problem encountered in practice.

1.2 Simulator Architecture

The presented research extensively uses the GiPSi (General Interactive Physical Simulation Interface) [8] framework, which is an open source/open architecture framework for developing surgical simulations (Figure 1.2). The general simulator architecture contains five main components, which are simulation objects, numerical integration

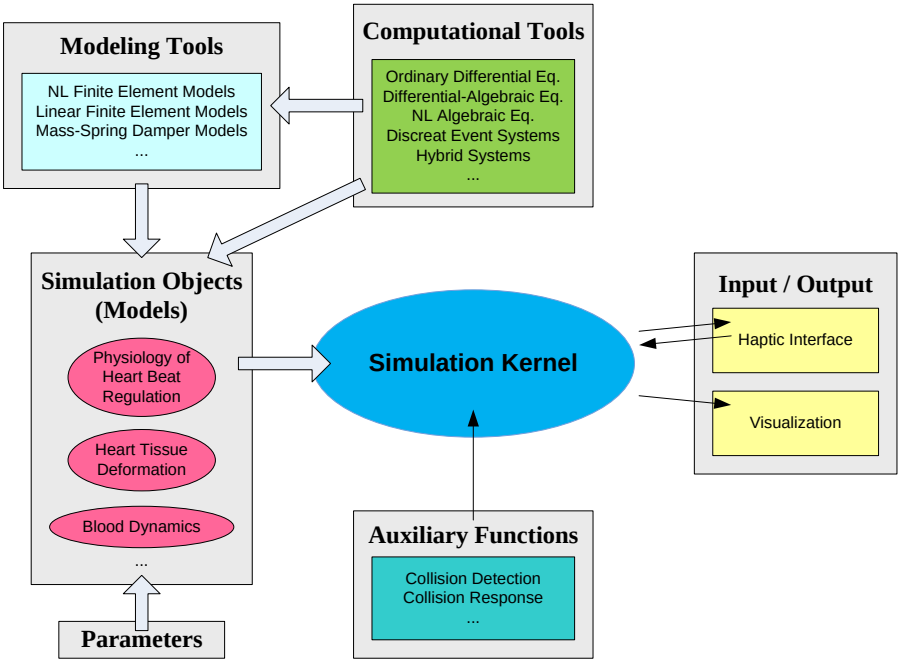


Figure 1.2: GiPSi: an open source/open architecture framework.

tools, collision detection and response tools, input/output modules, and simulation kernel.

Simulation objects. A scenario of surgical simulation contains many human organs, which are represented by simulation objects. Each simulation object has two main components: geometric model and physical model. The geometric model represents the information related to visualizing of the object to user via the display output, where as the physical model represents the dynamic behavior of the simulation object. High fidelity computer-based surgical simulations are made possible by creating simulation of physical models which capture the dynamic behavior of biological tissues. To achieve an accurate physical model, the study of biological properties is required (e.g. [13]). Biological tissues are anisotropic, inhomogeneous, nonlinear materials, and large deformations. Realistic simulation of biological behavior is one of the most challenging research areas. As surgical simulations need to run at interactive speeds, computationally efficient approximation is commonly employed. There are many approaches used to model the deformation object; for example, finite element

models, and lumped element models.

Numerical integration tools. The physical model representing a biological tissue is governed by the fundamental laws of physics and the relationship between these laws which apply universally to the simulation. Combining the laws of physics and their relations produce complex systems of mathematical equations, which usually are partial differential equations. Because the system of equations is complex and cannot be solved by using analytical methods, numerical methods are used instead, which require discretization and computation of the physical values at precise points in space and time.

Collision detection and response. A surgical simulation scene is composed of many simulation objects representing human organs and virtual surgical instruments. They need to interact with each other when their boundaries are in contact, otherwise the simulation objects could go through other simulation objects. To produce the interaction between simulation objects with realistic behaviors, collision detection algorithms need to compute the overlapping space and collision response algorithms need to separate the simulation objects by specific some constrain to that overlapped simulation objects.

Input/output modules. A user interfaces interacts with a user via the input/output subsystems which are haptic and visualization tools. The haptic tool is an input/output module that can retrieve positions of the surgical instrument (haptic device) from the user into the simulation and render force feedback to user to feel the interaction. The visualization tool is an output module that provides the graphic representation of simulation objects and the scenario of the simulation in three-dimensional space.

Simulation Kernel. A simulation kernel is a heart of simulator to manage simulation objects, communicate between the components, specific order of execution, and control simulator state.

1.3 Contributions

This thesis has five primary contributions. We proposed a novel method to determine elasticity parameters in lumped element models by using a finite element model as a reference model and using optimization techniques to determine the parameters. We studied in depth several numerical integration methods used in simulation to identify underlying trade-offs as a function of material properties. Third, we designed and implemented a novel collision detection and response method that is specifically suitable to be integrated into the GiPSi framework as an auxiliary function component. We extended the functionality of the GiPSi open source/open architecture surgical simulation framework to introduce networked simulation functionality. Finally, we developed an endoscopic third ventriculostomy simulator as a test bed platform.

1.4 Thesis Outline

The remainder of this thesis is organized as follows: In Chapter 2, the determination of elasticity parameters is presented. The in depth study and comparison of numerical integration algorithms used in simulation to identify underlying trade-offs as a function of material properties are presented in Chapter 3. In Chapter 4, the presentation of the algorithms for collision detection and response in deformable objects is presented. The improvement the design of the GiPSi framework architecture and the extension of the GiPSi framework to networked operation are presented in Chapter 5. Followed by the endoscopic third ventriculostomy simulator as a test bed is presented in Chapter 6. Finally, Chapter 7 discusses the conclusions and the future research direction.

Chapter 2

Elasticity Parameters Determination in Lumped Element Models

2.1 Introduction

Dynamic simulation of deformable objects in real-time for interactive virtual environments is an active area of research. The application that motivates this study is the virtual environment-based surgical training simulator, where real-time deformable tissue simulation is one of the enabling technologies. Virtual environments provide a complementary approach to the traditional method of training in surgery, which is primarily done through apprenticeship. The idea behind using surgical training simulators is similar to using flight simulators to train pilots. Virtual environments provide an environment where there is no risk to a patient and therefore less stressful. They are interactive and three-dimensional in contrast to study from textbooks, and they are relatively inexpensive compared to training in the operating room or animal labs. Virtual environments also give a unique advantage, as it is possible to

generate arbitrary anatomies and pathologies with which the surgeons can be trained for cases that they will encounter only a few times during their whole career but nonetheless must be trained for. However, effective virtual surgical environments require an interactive three-dimensional simulation environment, where the surgeons, using a haptic interface, can manipulate, cut, or suture dynamically and geometrically correct models of organs and tissues simulated on a computer.

The deformable tissue modeling approaches in the literature can be grouped in the following four broad categories: lumped element models (also known as mass-spring-damper models), finite element models (linear and nonlinear), particle based models, and parametric models.

Lumped element models (LEM) are meshes of mass, spring and damper elements [13, 14, 15]. Lumped element models are the most popular models for real time surgical simulators, because they are natural extensions of other deformable models used in computer animation. Lumped element models are conceptually simple, and possible to construct models, which can be simulated at interactive speeds. There are many applications used lumped element model, for example, Provot [16] used a mass and spring system to model a deformable cloth. Bourguignon and Cani [17] presented a method to controlling anisotropy of mass spring system in volumetric deformable models, such as human organs.

Finite element models (FEM) are used as a step to get closer to using models with physically based parameters [18, 19, 20]. Linear finite element models are computationally attractive as superposition can be used, and possible to perform extensive off-line calculations to significantly reduce the real-time computational burden. However, linear models are based on the assumption of small deformation, typically less than 1%, which is not valid for much of the soft tissue manipulation during surgery. These models cannot handle rigid motions either [21]. Linear models lose their computational advantage under topology changes, e.g., as a result of cutting, as the

off-line calculations cannot be used. To address this last problem, Delingette et al. [22] proposed to use lumped element models locally where there is topological change (such as cutting) and use a linear finite element model for the rest. Nonlinear finite element models are highly accurate models, which take into account nonlinear constitutive behavior of the materials as well as large deformation effects. They are generally regarded as the gold standard for high accuracy computation. However, these models are computationally very intensive and therefore not suitable for real-time simulation in their basic form [23, 24, 21]. Recently, a lot of research effort has been focusing on improving computational performance of these models. For example, Wu et al. proposed to use mass lumping and adaptive mesh refinement [25], and multigrid simulation [26] to achieve higher performance with nonlinear FEM. Müller et al. [27, 28] proposed a corotation-based approach for finite element models to improve the artifacts from large deformation. Nesme et al. [29] presented a FEM-based physically plausible modeling method. As well as, Irving et al. [30] presented an invertible finite element algorithm for simulating large deformations.

Particle based models model the deformable object continuum as a collection of loosely coupled finite volume particles, such as simulating a deformable object as a collection of elastic spheres (for example, as proposed by Conti [31]). The interaction of these particles between themselves and with the external forces determine the behavior of the deformable object. These models are not really intended to accurately model real tissue behavior, but to have a plausible looking tissue behavior achieved through minimal computation.

Parametric models include commonly used free form [32] and spline based [33] deformable models where location of some control points determines locally the shape of the deformable object. Another method for parametric modeling of deformable objects was proposed by Metaxas [15] where a very small number of parameters characterized globally the shape of a large geometric model of a deformable body,

e.g. using the semi-axis lengths and principal-axes directions to parameterize an ellipsoid. Parametric models are not physically based, and like particle based models, not intended to accurately model with real tissue behavior, but to have fast and plausible looking at interactive responses.

The two most commonly used deformable object models are finite element models and lumped element models. As mentioned earlier, lumped element models are conceptually simple and computationally more efficient compared to finite element models. A common problem with the lumped parameter models used in literature is the selection of component parameters: spring constants, damper constants, and nodal mass values. There is no general physically based or systematic method in the literature to determine the element types or parameters from physical data or known constitutive behavior. The typical practice in the literature is somewhat *ad hoc*, the element types and connectivities are empirically assumed, usually based on the structure of the geometric model, and the element parameters are either hand tuned to get a reasonable looking behavior or estimated by a parameter optimization method to fit the model response to an experimentally measured response. For example, Joukhadar et al. [34, 35] used a predefined mesh topology and then determined the element parameters with a genetic algorithm search technique. Bianchi et al. [36, 37] used a genetic algorithm based method to determine the mesh topology and stiffness of mass-spring models by using finite element models as reference. Deussen et al. [38] used a search method based on simulated annealing algorithm to determine optimum mass-spring parameters in two dimensions.

There are several studies in the literature which determine lumped element model parameters by using continuum mechanics, elasticity and finite element theory. Van Gelder [39] proposed a formulation to approximate the spring constants in triangular mesh of isotropic, linearly elastic materials. He also extended to three dimensions. The experimental results showed that his model can approximate the deformation of

an isotropic elastic membrane with limit condition of Poisson's ratio of zero. Maciel et al. [40] proposed techniques to model a soft tissue from real biological tissue properties by using a generalized mass spring model (molecular model) with four different methods; however, none of these methods worked unconditionally. Baudet et al. [41, 42] proposed an approach similar to Van Gelder but for rectangular and hexahedral elements. They also introduced a correction force orthogonal to the elongation force to correct the effects of non-zero Poisson's ratio and compared results with finite element model simulations. In 2000, Cavusoglu [43], proposed a method to determine elasticity parameters of a lumped element (mass-spring) model by approximating the stiffness matrix of the finite element model with the stiffness matrix of the lumped element model. More recently, Lloyd et al. [44] introduced a method for identification of spring constants of lumped element models from the finite element models in triangular, rectangular, and tetrahedral meshes. Their method produced the better approximative results when the spring constants had been calculated for the specific value of Poisson's ratio with pre-strained springs in two-dimensional element (rectangular meshes), and with volume preserving forces in three-dimensional element (tetrahedral meshes). Wang and Devarajan [45, 46] presented mass-spring models for one and two dimensions derived from explicit continuum expressions with a preloaded spring model to improve the accuracy.

In this chapter, a method to determine the mass and spring constants of lumped element models is presented. The proposed method to determine component parameters is based on approximating the input-output relations of finite element model "elements" with lumped element model "elements." The spring constants are determined through an optimization that minimizes the matrix norm of the error between the stiffness matrices of the lumped element model and a corresponding finite element model of the same object. Our method uses an approach similar to the methods of [43] and [44]. However, the method proposed in this study is uses an optimization

to find the lumped element model parameters that best fit the finite element model response, while the method presented in [44] calculated the lumped element model parameters from equating the stiffness matrices from lumped element and finite element model. Furthermore, in the present study, the method has also been developed for tetrahedral and hexahedral elements, whereas [44] is limited only to tetrahedral elements.

The proposed method is developed for two dimensions with triangular and quadrilateral elements and for three-dimensional volumetric objects with tetrahedral and hexahedral (brick) elements. In the next section, the basic continuum theory of elastic bodies and the finite element and lumped element models are summarized in order to formulate the problem and introduce the notations. The formulation of the soft object deformation using these two models are also compared to make some basic observations in section 2.2. Then, the proposed method is presented in detail in section 2.3. After that, the experimental results are presented in section 2.4, followed by concluding remarks in section 2.5.

2.2 Background and Formulation

2.2.1 Continuum Equations for Elastic Bodies

In order to be able to systematically study the methods mentioned above, we first formulate the underlying physical problem. Consider the deformable body $\mathcal{B}$, which is a regular region in $\Re^3$. $\mathcal{B}$ is also called the reference configuration. $\mathbf{p} \in \mathcal{B}$ are the body coordinates of the material points. A deformation $\mathbf{f} : \mathcal{B} \rightarrow \Re^3$ of a body is a one-to-one smooth mapping that maps each material point $\mathbf{p}$ to a point $\mathbf{x} = \mathbf{f}(\mathbf{p})$ in a spatial frame. A motion $\mathsf{x} : \mathcal{B} \times \Re \rightarrow \Re^3$ of a body is a $\mathcal{C}^3$ function where for each t, $\mathsf{x}(\mathbf{p}, t)$ is a deformation. (Figure 2.1)

The total Lagrangian form of the field equations that govern the dynamic behavior

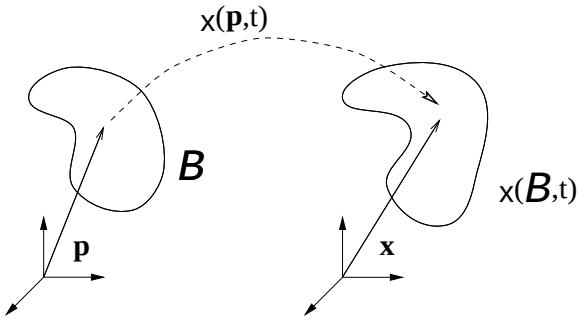


Figure 2.1: Motion of a body.

of elastic bodies are given by [47, 48, 49]:

$$\begin{aligned} \mathbf{S} &= \mathbf{F}\bar{\mathbf{S}}(\mathbf{C}), \\ \mathbf{C} &= \mathbf{F}^{\mathrm{T}}\mathbf{F}\,,\ \mathbf{F} = \nabla \mathsf{x}, \\ \mathrm{Div}\,\mathbf{S} + \mathbf{b}_0 &= \rho_0 \ddot{\mathsf{x}} \ \text{ in } \ \mathcal{B}, \end{aligned} \tag{2.1}$$

where $\mathbf{F}$ is the deformation gradient, $\mathbf{S}$ is the first Piola-Kirchhoff stress tensor, $\bar{\mathbf{S}}$ is the second Piola-Kirchhoff stress tensor, $\mathbf{C}$ is the right Cauchy-Green strain tensor, $\mathbf{b}_0$ is the body force, ρ_0 is the mass density at the reference configuration, $\ddot{\mathsf{x}}$ is an acceleration of x, and ∇ and Div are respectively the gradient and divergence operators in body coordinates. These field equations are derived from the empirical physical laws, such as conservation of mass and momentum, and the independence of the response from observer. The first equation is where the material properties are included.

The boundary value problems in finite elasticity are obtained by combining the basic system of field equations given by equation (2.1) with suitable initial and boundary conditions. Initial conditions are usually specified by the initial motion and velocity. The type of boundary value problem typically encountered in our application is specified with:

$$\mathsf{x}(\mathbf{p}, 0) = \mathsf{x}_0(\mathbf{p})\,,\ \dot{\mathsf{x}}(\mathbf{p}, 0) = \mathbf{v}_0(\mathbf{p}), \tag{2.2}$$

where x_0 and $\mathbf{v}_0$ are prescribed functions on $\mathcal{B}$. For boundary conditions, two

complimentary regular subsets $\mathcal{S}_1$ and $\mathcal{S}_2$ of $\partial\mathcal{B}$, and interior $\mathcal{S}^o$ of $\mathcal{B}$, with $\partial\mathcal{B} = \mathcal{S}_1 \cup \mathcal{S}_2$, $\mathcal{S}_1^o \cap \mathcal{S}_2^o = \emptyset$ are given, where the motion is prescribed on $\mathcal{S}_1$ and the surface traction is prescribed on $\mathcal{S}_2$:

$$\mathsf{x} = \bar{\mathsf{x}} \ \text{ on } \mathcal{S}_1 \times [0,\infty)\,, \quad \mathbf{S}\mathbf{n} = \bar{\mathbf{s}} \ \text{ on } \mathcal{S}_2 \times [0,\infty). \tag{2.3}$$

The boundary value problem that needs to be solved or simulated in real time in order to model the deformation of elastic tissue is given by the system of equations (2.1, 2.2, 2.3). For the solution of the boundary value problem specified by (2.1, 2.2, 2.3), the partial differential equation (PDE) needs to be spatially and temporally discretized. Typically, the PDE is first discretized in space, using a method such as finite element [50, 51] or finite difference [52] to construct a large system of ordinary differential equations, in the form of an initial value problem. The resulting ordinary differential equation is then approximately solved in time by numerical integration methods [53, 51].

2.2.2 Linear Elastic Material Properties

If the displacement gradient is small and the residual stress in reference configuration vanishes, then the system of field equations gives by (2.1) can be approximated by linearization. Specifically, if we define displacement

$$\mathbf{u}(\mathbf{p},t) = \mathsf{x}(\mathbf{p},t) - \mathbf{p}. \tag{2.4}$$

Then (2.1) can be linearized as [47]

$$\begin{aligned} \mathbf{S} &= \mathsf{C}[\mathbf{E}], \\ \mathbf{E} &= \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^{\mathrm{T}}), \\ \operatorname{Div}\mathbf{S} + \mathbf{b}_0 &= \rho_0\ddot{\mathbf{u}}, \end{aligned} \tag{2.5}$$

where $\mathbf{E}$ is the infinitesimal strain, and C is the elasticity tensor. When the material is isotropic, then (2.5) can be further simplified by

$$\mathbf{S} = \bar{\mathbf{S}} = 2\mu\mathbf{E} + \lambda(\operatorname{tr} E)\mathbf{I}, \tag{2.6}$$

where λ and μ are the Lame's constants. In matrix notation

$$\mathbf{S} = \begin{bmatrix} \sigma_{\mathrm{xx}} & \sigma_{\mathrm{xy}} & \sigma_{\mathrm{xz}} \\ \sigma_{\mathrm{yx}} & \sigma_{\mathrm{yy}} & \sigma_{\mathrm{yz}} \\ \sigma_{\mathrm{zx}} & \sigma_{\mathrm{zy}} & \sigma_{\mathrm{zz}} \end{bmatrix} \tag{2.7}$$

and

$$\mathbf{E} = \begin{bmatrix} \varepsilon_{\mathrm{xx}} & \varepsilon_{\mathrm{xy}} & \varepsilon_{\mathrm{xz}} \\ \varepsilon_{\mathrm{yx}} & \varepsilon_{\mathrm{yy}} & \varepsilon_{\mathrm{yz}} \\ \varepsilon_{\mathrm{zx}} & \varepsilon_{\mathrm{zy}} & \varepsilon_{\mathrm{zz}} \end{bmatrix} \tag{2.8}$$

which are symmetric matrices.

The elastic properties of linear homogeneous isotropic material are more commonly represented by two constants, namely Young's modulus (E) and Poisson's ratio (ν). The strain is the change of dimension divided by the dimension itself; $\varepsilon = \frac{\Delta L}{L}$. The stress is the force divided by the area on which it acts; $\sigma = \frac{F}{A}$. The Poisson's ratio is the ratio of transverse contraction strain to longitudinal extension strain in the direction of stretching force; $\nu = -\frac{\varepsilon_{transverse}}{\varepsilon_{longitudinal}}$. The modulus is the ratio of stress to strain, which is a constant depending on the material. The Young's modulus is the ratio of tensile stress to tensile strain; $E = \frac{\sigma}{\varepsilon} = \frac{F/A}{\Delta L/L} = \frac{FL}{\Delta LA}$. The shear modulus which is a commonly used parameter is the ratio of shear stress to shear strain. The deformation occurs when a force is applied parallel to one face of the object while the opposite face is fixed; $G = \frac{\sigma}{\tau} = \frac{F/A}{\Delta x/L} = \frac{FL}{\Delta xA} = \frac{E}{2(1+\nu)}$. The bulk modulus which is another commonly used parameter is the ratio of stress to change in volume of object; $K = \frac{P}{\Delta V/V} = \frac{PV}{\Delta V} = \frac{E}{3(1-2\nu)}$. The Lame's constants are directly related to the modulus of elasticity and Poisson's ratio as: $\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$, $\mu = \frac{E}{2(1+\nu)}$.

Two-dimensional elasticity is categorized into two cases: plane strain and plane stress. Plane strain is used when the thickness of an object is large, while plane stress is used when the thickness of an object is small compared to its overall dimensions. Both cases are subset of general three-dimensional elasticity problems. The stress

strain relation in plane stress case is given by:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = \begin{bmatrix} \frac{4\mu(\lambda+\mu)}{\lambda+2\mu} & \frac{2\lambda\mu}{\lambda+2\mu} & 0 \\ \frac{2\lambda\mu}{\lambda+2\mu} & \frac{4\mu(\lambda+\mu)}{\lambda+2\mu} & 0 \\ 0 & 0 & \mu \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{bmatrix}, \tag{2.9}$$

and for the plane strain case, it is given by:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = \begin{bmatrix} \lambda+2\mu & \lambda & 0 \\ \lambda & \lambda+2\mu & 0 \\ 0 & 0 & \mu \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{bmatrix}. \tag{2.10}$$

For three-dimensional elasticity, six components of stress and strain exist and the stresses are related to the strains by Hooke's law as follows:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{xz} \\ \sigma_{yz} \end{bmatrix} = \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ 2\varepsilon_{xy} \\ 2\varepsilon_{xz} \\ 2\varepsilon_{yz} \end{bmatrix}. \tag{2.11}$$

2.2.3 Finite Element Models (FEM)

Finite element method is a systematic technique for obtaining the spatial discretization of the PDE describing the continuum behavior of deformable objects. In this section, we introduce the basic formulation of the finite elements method. Please refer to [50] and [51] for detailed treatments of the finite elements method. Variational form of the PDE (2.1, 2.2, 2.3) is

$$\int_{\mathcal{B}} \xi \cdot (\rho_0 \ddot{\mathbf{x}} - \mathbf{b}_0 - \operatorname{Div} S) dV + \int_{\mathcal{S}_2} \xi \cdot (S\mathbf{n} - \bar{\mathbf{s}}) dA = 0, \tag{2.12}$$

where ξ is the variation of deformation. Divergence theorem gives

$$\begin{aligned} \int_{\mathcal{B}} \xi \cdot (\operatorname{Div} S) dV &= \int_{\mathcal{S}} \xi \cdot S\mathbf{n} dA - \int_{\mathcal{B}} S \cdot \nabla \xi dV \\ &= \int_{\mathcal{S}_2} \xi \cdot S\mathbf{n} dA - \int_{\mathcal{B}} S \cdot \nabla \xi dV. \end{aligned} \tag{2.13}$$

Substituting this in the variational form, we get

$$\int_{\mathcal{B}} \xi \cdot \rho_0 \ddot{\mathbf{x}} dV + \int_{\mathcal{B}} S \cdot \nabla \xi dV = \int_{\mathcal{B}} \xi \cdot \mathbf{b}_0 dV + \int_{\mathcal{S}_2} \xi \cdot \bar{\mathbf{s}} dA, \tag{2.14}$$

for the weak form of the PDE over the whole body.

Now consider the following finite element approximation on each element

$$\mathsf{x}(\mathbf{p},t) = \sum_{i=1}^{n} N_i^e \mathbf{x}_i^e(t), \tag{2.15}$$

$$\xi(\mathbf{p},t) = \sum_{i=1}^{n} N_i^e \xi_i^e(t), \tag{2.16}$$

where N_i^e, $i = 1..n$, is the isoparametric set of approximation functions (shape functions), and n is the number of node in element. We can write these in the matrix form as

$$\begin{aligned}
\mathsf{x}^e &= \begin{bmatrix} N_1^e & N_2^e & \cdots & N_n^e \end{bmatrix} \begin{bmatrix} \mathbf{x}_1^e \\ \mathbf{x}_2^e \\ \vdots \\ \mathbf{x}_n^e \end{bmatrix} \\
&= N^e \hat{\mathbf{x}}^e, \qquad (2.17)
\end{aligned}$$

$$\xi^e = N^e \hat{\xi}^e, \tag{2.18}$$

$$\nabla\xi \rightarrow \begin{bmatrix} \xi_{1,1}^e \\ \xi_{2,2}^e \\ \xi_{3,3}^e \\ \xi_{1,2}^e \\ \xi_{2,3}^e \\ \xi_{3,1}^e \\ \xi_{1,3}^e \\ \xi_{2,1}^e \\ \xi_{3,2}^e \end{bmatrix} = B^e \hat{\xi}^e, \tag{2.19}$$

$$B^e = \begin{bmatrix} B_1^e & B_2^e & \cdots & B_n^e \end{bmatrix}, \tag{2.20}$$

$$B_i^e = \begin{bmatrix} N_{i,1}^e & 0 & 0 \\ 0 & N_{i,2}^e & 0 \\ 0 & 0 & N_{i,3}^e \\ N_{i,2}^e & 0 & 0 \\ 0 & N_{i,3}^e & 0 \\ 0 & 0 & N_{i,1}^e \\ N_{i,3}^e & 0 & 0 \\ 0 & N_{i,1}^e & 0 \\ 0 & 0 & N_{i,2}^e \end{bmatrix}, \tag{2.21}$$

where B^e is the strain-displacement matrix. Here we have used the subscript notation, for example, $\xi_{3,1}^e$ is the partial derivative of the third component of ξ^e with respect

to its first variable. Substituting all of the above in the weak form, we obtain

$$\hat{\xi}^{eT}\left[\int_{\Omega_0^e} N^{eT}\rho_0 N^e dV\right]\ddot{\hat{\mathbf{x}}}^e + \hat{\xi}^{eT}\left[\int_{\Omega_0^e} B^{eT}S(N^e\hat{\mathbf{x}}^e)dV\right] = \hat{\xi}^{eT}\left[\int_{\Omega_0^e} N^{eT}\mathbf{b}_0 dV\right] + \hat{\xi}^{eT}\left[\int_{\partial\Omega_0^e} N^{eT}\bar{\mathbf{s}}dA\right], \quad (2.22)$$

which can be written compactly as

$$\hat{\xi}^{eT}\left[M^e\ddot{\hat{\mathbf{x}}}^e + R^e(\hat{\mathbf{x}}^e) - F^e\right] = \hat{\xi}^{eT}\int_{\partial\Omega_0^e - \mathcal{S}_{20}} N^{eT}\bar{\mathbf{s}}dA, \quad (2.23)$$

where

$$\begin{array}{ll} M^e = \int_{\Omega_0^e} N^{eT}\rho_0 N^e dV & \text{is the element mass matrix,} \\ R^e = \int_{\Omega_0^e} B^{eT}S(N^e\hat{\mathbf{x}}^e)dV & \text{is the stress divergence term,} \\ F^e = \int_{\Omega_0^e} N^{eT}\mathbf{b}_0 dV + \int_{\partial\Omega_0^e\cap\mathcal{S}_{20}} N^{eT}\bar{\mathbf{s}}dA & \text{is the external force vector.} \end{array} \quad (2.24)$$

As $\hat{\xi}^e$ is arbitrary, at the element level we have

$$M^e\ddot{\hat{\mathbf{x}}}^e + R^e(\hat{\mathbf{x}}^e) = F^e + \int_{\partial\Omega_0^e - \mathcal{S}_{20}} N^{eT}\bar{\mathbf{s}}dA. \quad (2.25)$$

After the element level equations are assembled, the resulting system is in the form

$$M\ddot{\hat{\mathbf{x}}} + R(\hat{\mathbf{x}}) = F, \quad (2.26)$$

which is a system of ordinary differential equations.

For the topology of FEM equations, the matrix M^e is dense since the element shape functions N_I^e are not typically mutually orthogonal. The matrix M^e is sometimes approximated with a diagonal matrix by using nodal quadrature to decrease computational cost, but this is by no means inherent to the finite element method. The same is true for the function R^e, i.e. the "force" on any node depends on the nodal variables of all the other nodes within the element, as given above in (2.24) and (2.25). Therefore, in FEM formulation, the degrees of freedom are fully connected within an element.

For the assembled set of equations, the variables for the elements are connected only by the degrees of freedom shared between elements. This results in a typical banded structure for the matrix M and a similar dependence in the function R.

2.2.4 Lumped Element Models (LEM)

Lumped element models are meshes of mass, spring and damper elements. Lumped masses at the nodes of the mesh are interconnected by spring and damper elements. The equations of motion are the collection of the Newton's equations written for the individual nodal masses.

For each nodal mass, the equation of motion is in the form

$$m_i \ddot{\mathbf{x}}_i + K_i(\mathbf{x}) = F_i \tag{2.27}$$

with F_i being the external force on the node, such as gravity, and

$$K_i(\mathbf{x}) = \sum_{\{i,j \text{ connected}\}} \mathbf{f}(\mathbf{x}_i, \mathbf{x}_j) + \sum_{\{i,j,k \text{ connected}\}} \mathbf{g}(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k), \tag{2.28}$$

where $f(\cdot,\cdot)$ is the force from a linear spring and the $g(\cdot,\cdot,\cdot)$ is the force from an angular spring. A typical expression used for linear springs is

$$\mathbf{f}(\mathbf{x}_1, \mathbf{x}_2) = k(\|\mathbf{x}_1 - \mathbf{x}_2\| - L_0)\frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|}. \tag{2.29}$$

For the angular springs, the force expression is in the form

$$\mathbf{g}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = k(\theta - \theta_0)\left(\frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|} \times \frac{\mathbf{x}_2 - \mathbf{x}_3}{\|\mathbf{x}_2 - \mathbf{x}_3\|}\right) \times \frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|}. \tag{2.30}$$

These expressions are for negative force acting on node $\mathbf{x}_1$, due to a spring between $\mathbf{x}_1, \mathbf{x}_2$ and an angular spring between $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$. L_0 is the rest length of the linear spring and θ_0 is the rest angle of the angular spring. The angular springs are typically used to enforce $\mathcal{C}^1$ continuity in the mesh. In this work, we use only linear spring for simplicity.

For the topology of LEM equations, the connectivity in LEM depends on the types of the springs used. The force on any node depends on the nodes that connected to through springs. This results in a sparse system of equations, similar to the finite element models.

2.2.5 Boundary Conditions

Consider a single deformable body placed on a rigid surface, being manipulated by a position controlled instrument, as seen in Figure 2.2. The typical boundary conditions for this case are as follows: At the interface between the deformable body and a position controlled object (such as ground, which has fixed location, or the instrument, whose position is specified through the haptic interface), normal displacement of the nodes is specified as boundary condition. In the tangent directions, the traction is specified as zero for the frictionless case or proportional to normal force when there is friction. For the parts of the body that are glued to a location or grabbed by the instrument, the displacement is specified in all directions. All other boundary conditions are given as $\bar{\mathbf{s}} = 0$ (zero traction).

Position boundary conditions can easily be prescribed in all of the modeling methods previously presented, through the nodal variables in lumped element models, and through the positional degrees of freedom of the elements in finite element models.

Enforcement of traction boundary conditions is more difficult. In the finite element method, the traction boundary conditions enter through the $\int_{\partial\Omega_0^e \cap \mathcal{S}_{2\,0}} N^{e\,T} \bar{\mathbf{s}} dA$ term of F^e. In the lumped element method, the traction boundary conditions need to be somehow converted into nodal forces. However, there is no systematic way to do this. As there is no counterpart of the approximation functions of FEM in LEM, the way F^e term is systematically calculated in (2.24) cannot be transferred to LEM. When the type of traction boundary conditions is simple, i.e. zero if there is no friction, it is possible to get away with this important deficiency of the LEM.

2.3 Determination of Parameters

In finite element models, the parameters of the elements are determined from the constitutive properties of the material of the object being simulated. For the lumped

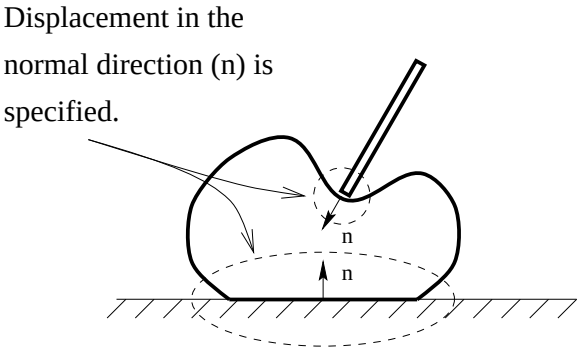


Figure 2.2: Typical boundary conditions of a deformable body being manipulated.

element models, there is no intrinsic method to determine the element parameters since the models are not actually motivated from approximating the physical behavior of the object.

As mentioned in the discussion above, one of the main problems of LEM is the lack of a systematic way to determine element parameters. In the literature, the parameters of the LEM models are determined through parameter estimation, to fit the response of the model to an experimentally measured response. If the structure allows, it may be possible to isolate effects of some parameters or do some approximations to isolate these parameters and therefore simplify parameter estimation [54]. Otherwise, this can be a very complex optimization problem depending on the number of parameters used.

Here, we establish a parallelism between the elements in FEM and LEM, and explore methods for setting up of the LEM mesh and selection of its parameters as a way to approximate FEM. In the discussion below, without loss of generality, we look at the two-dimensional case (plane strain*) in the absence of external forces as an illustrative example, in order to simplify the notation and equations.

Consider a planar 4 node with $\mathcal{C}^0$ continuous isoparametric element for the FEM, and a 4 mass configuration for the LEM (Figure 2.3). The masses of the LEM mesh

*Plane strain analysis is used to solve deformation in infinitely long structures, which are uniform in the third dimension.

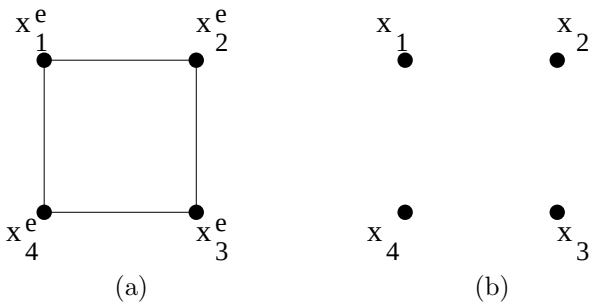


Figure 2.3: 4 node FEM (a) and LEM (b) elements.

are located at the same spatial locations as the nodes of the FEM element. This configuration of the LEM masses, with the interconnection springs and dampers, is used as the building block elements of the LEM mesh. At this point, we are not yet specifying the spring connections between the nodes.

The equations of motion for the FEM and LEM elements are respectively

$$\begin{aligned} M^e\ddot{\hat{\mathbf{x}}}^e + R^e(\hat{\mathbf{x}}^e) &= 0, \\ \begin{bmatrix} m_1 & & 0 \\ & \ddots & \\ 0 & & m_4 \end{bmatrix} \ddot{\hat{\mathbf{x}}} + K(\hat{\mathbf{x}}) &= 0. \end{aligned} \tag{2.31}$$

The matrix M^e is dense, but if we use nodal quadrature, it is possible to get a diagonal approximation for the M^e in FEM. This is a commonly used approximation in FEM to improve computational efficiency (e.g [25]). For the LEM, we can choose

$$m_i = m^e_{ii}, \tag{2.32}$$

therefore getting a physically based value for the nodal mass. Then, if we can approximate $R^e(\hat{\mathbf{x}}^e)$ with $K(\hat{\mathbf{x}})$, we can use the LEM for approximating FEM, avoiding the parameter determination problem. If we observe at the structure of the function $R^e(\hat{\mathbf{x}}^e)$,

$$R^e(\hat{\mathbf{x}}^e) = \begin{bmatrix} R^e_1(\mathbf{x}^e_1, \mathbf{x}^e_2, \mathbf{x}^e_3, \mathbf{x}^e_4) \\ R^e_2(\mathbf{x}^e_1, \mathbf{x}^e_2, \mathbf{x}^e_3, \mathbf{x}^e_4) \\ R^e_3(\mathbf{x}^e_1, \mathbf{x}^e_2, \mathbf{x}^e_3, \mathbf{x}^e_4) \\ R^e_4(\mathbf{x}^e_1, \mathbf{x}^e_2, \mathbf{x}^e_3, \mathbf{x}^e_4) \end{bmatrix}, \tag{2.33}$$

and comparing the FEM equations with the LEM equations,

$$R_i^e(\hat{\mathbf{x}}^e) = \int_{\Omega_0^e} B_i^{eT} S(\hat{\mathbf{x}}^e) dV,$$
$$K_i(\hat{\mathbf{x}}) = \sum_{\{i,j \text{ connected}\}} \mathbf{f}(\mathbf{x}_i, \mathbf{x}_j), \tag{2.34}$$

we observe that the nonlinear functions $R_i^e(\hat{\mathbf{x}}^e)$ needs to be approximated by the function $K_i(\hat{\mathbf{x}})$, which is the sum of the spring forces on node i. Performing an optimization over the nonlinear functions $R_i^e(\hat{\mathbf{x}}^e)$ and $K_i(\hat{\mathbf{x}})$ would be computationally complex. Instead, we linearize the two models, and perform an optimization to identify the LEM parameters that most closely match the linearization. This actually has the effect of matching the tangent behavior of the two original nonlinear models. Also, the linear case enables us to make some basic observation, which gives us important insights.

For the LEM element, we need to linearize expression for the spring forces by using Taylor series expansion. The result is

$$\begin{bmatrix} \Delta\mathbf{f}_i \\ \Delta\mathbf{f}_j \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{f}(\mathbf{x}_i,\mathbf{x}_j)}{\partial \mathbf{x}_i} & \frac{\partial \mathbf{f}(\mathbf{x}_i,\mathbf{x}_j)}{\partial \mathbf{x}_j} \end{bmatrix} \begin{bmatrix} \mathbf{u}_i \\ \mathbf{u}_j \end{bmatrix}, \tag{2.35}$$

where $\mathbf{u} = \mathbf{x} - \mathbf{x}(0)$ is the displacement,

$$\frac{\partial \mathbf{f}(\mathbf{x}_i,\mathbf{x}_j)}{\partial \mathbf{x}_i} = -\frac{\partial \mathbf{f}(\mathbf{x}_i,\mathbf{x}_j)}{\partial \mathbf{x}_j} = \begin{bmatrix} A_1^{\mathbf{u}_i,\mathbf{u}_j} & B_{1,2}^{\mathbf{u}_i,\mathbf{u}_j} \\ B_{2,1}^{\mathbf{u}_i,\mathbf{u}_j} & A_2^{\mathbf{u}_i,\mathbf{u}_j} \end{bmatrix}, \tag{2.36}$$

$$A_p^{\mathbf{u}_i,\mathbf{u}_j} = k^{i,j}\left(1 - \frac{L_0}{\|\mathbf{x}_j - \mathbf{x}_i\|}\frac{\|\mathbf{x}_j - \mathbf{x}_i\|^2 - (\mathbf{x}_{j_p} - \mathbf{x}_{i_p})^2}{\|\mathbf{x}_j - \mathbf{x}_i\|^2}\right), \tag{2.37}$$

$$B_{p,q}^{\mathbf{u}_i,\mathbf{u}_j} = k^{i,j}\left(\frac{L_0}{\|\mathbf{x}_j - \mathbf{x}_i\|}\frac{(\mathbf{x}_{j_p} - \mathbf{x}_{i_p})(\mathbf{x}_{j_q} - \mathbf{x}_{i_q})}{\|\mathbf{x}_j - \mathbf{x}_i\|^2}\right). \tag{2.38}$$

Define

$$K_{i,j} = -\begin{bmatrix} A_1^{\mathbf{u}_i,\mathbf{u}_j} & B_{1,2}^{\mathbf{u}_i,\mathbf{u}_j} \\ B_{2,1}^{\mathbf{u}_i,\mathbf{u}_j} & A_2^{\mathbf{u}_i,\mathbf{u}_j} \end{bmatrix} \tag{2.39}$$

to simplify the notation. Note that $K_{i,j} = K_{j,i}$. Then, the linearized equations for LEM is

$$K(\hat{\mathbf{x}}) \approx K\hat{\mathbf{u}} \tag{2.40}$$

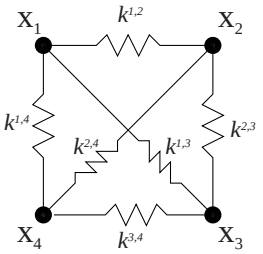


Figure 2.4: A fully connected 4 node LEM element.

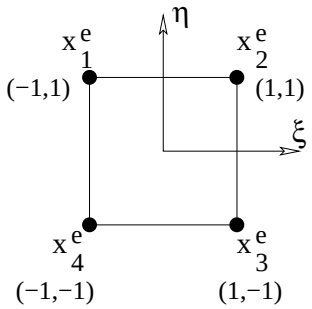


Figure 2.5: 4 node master FEM element.

and the K matrix has entries for each of the springs. For example, if we consider the LEM element of Figure 2.4 we get

$$K^e = \begin{bmatrix} K_{1,1} & K_{1,2} & K_{1,3} & K_{1,4} \\ K_{2,1} & K_{2,2} & K_{2,3} & K_{2,4} \\ K_{3,1} & K_{3,2} & K_{3,3} & K_{3,4} \\ K_{4,1} & K_{4,2} & K_{4,3} & K_{4,4} \end{bmatrix}, \tag{2.41}$$

where $K_{i,i} = -\sum_{j=1,j\neq i}^{4} K_{i,j}$.

For the FEM element

$$R^e(\hat{\mathbf{x}}^e) \approx R^e \hat{\mathbf{u}}, \tag{2.42}$$

$$R^e = \int_{\Omega_0^e} B^{e\,T} D B^e dV, \tag{2.43}$$

where D is the matrix which transforms strain vector to stress vector ($\sigma = D\varepsilon$). For brevity, we are using the same symbol for the nonlinear function and matrix for the linear case, since they are distinguishable from the context.

At this point, to simplify the calculations, we further assume that the element in the reference configuration is the same as the master element $\hat{\Omega}$ (Figure 2.5) and the

deformable object is a homogeneous linear isotropic material. Then,

$$\begin{aligned} R^e &= \int_{\Omega_0^e} B^{eT} D B^e dx dy \\ &= \int_{\hat{\Omega}} B^{eT} D B^e |J| d\xi d\eta \\ &= \int_{-1}^{1}\int_{-1}^{1} B^{eT} D B^e |J| d\xi d\eta, \end{aligned} \tag{2.44}$$

$$R^e_{i,j} = \int_{-1}^{1}\int_{-1}^{1} B^{eT}_i D B^e_j |J| d\xi d\eta, \tag{2.45}$$

where J is the Jacobian operator relating the natural coordinate derivatives to the local coordinate derivatives. For an isoparametric element, the shape functions in the natural coordinate are

$$\begin{aligned} N^e_1 &= \frac{(1-\xi)(1+\eta)}{4}, \\ N^e_2 &= \frac{(1+\xi)(1+\eta)}{4}, \\ N^e_3 &= \frac{(1+\xi)(1-\eta)}{4}, \\ N^e_4 &= \frac{(1-\xi)(1-\eta)}{4}, \end{aligned} \tag{2.46}$$

and for an isotropic plane strain

$$D = \begin{bmatrix} \lambda+2\mu & \lambda & 0 \\ \lambda & \lambda+2\mu & 0 \\ 0 & 0 & \mu \end{bmatrix}, \tag{2.47}$$

$$B^e_i = \begin{bmatrix} N^e_{i,1} & 0 \\ 0 & N^e_{i,2} \\ N^e_{i,2} & N^e_{i,1} \end{bmatrix}, \tag{2.48}$$

where λ and μ are the Lame's constants of the material. If we evaluate the equation (2.45), we get

$$R^e = \begin{bmatrix} \frac{\lambda}{3}+\mu & \frac{-\lambda-\mu}{4} & \frac{-\lambda}{3}-\frac{\mu}{2} & \frac{-\lambda}{4}+\frac{\mu}{4} & \frac{-\lambda}{6}-\frac{\mu}{2} & \frac{\lambda}{4}+\frac{\mu}{4} & \frac{\lambda}{6} & \frac{\lambda-\mu}{4} \\ \frac{-\lambda-\mu}{4} & \frac{\lambda}{3}+\mu & \frac{\lambda}{4}-\frac{\mu}{4} & \frac{\lambda}{6} & \frac{\lambda}{4}+\frac{\mu}{4} & \frac{-\lambda}{6}-\frac{\mu}{2} & \frac{-\lambda+\mu}{4} & \frac{-\lambda}{3}-\frac{\mu}{2} \\ \frac{-\lambda}{3}-\frac{\mu}{2} & \frac{\lambda}{4}-\frac{\mu}{4} & \frac{\lambda}{3}+\mu & \frac{\lambda}{4}+\frac{\mu}{4} & \frac{\lambda}{6} & \frac{-\lambda}{4}+\frac{\mu}{4} & \frac{-\lambda}{6}-\frac{\mu}{2} & \frac{-\lambda}{4}-\frac{\mu}{4} \\ \frac{-\lambda}{4}+\frac{\mu}{4} & \frac{\lambda}{6} & \frac{\lambda}{4}+\frac{\mu}{4} & \frac{\lambda}{3}+\mu & \frac{\lambda}{4}-\frac{\mu}{4} & \frac{-\lambda}{3}-\frac{\mu}{2} & \frac{-\lambda}{4}-\frac{\mu}{4} & \frac{-\lambda}{6}-\frac{\mu}{2} \\ \frac{-\lambda}{6}-\frac{\mu}{2} & \frac{\lambda}{4}+\frac{\mu}{4} & \frac{\lambda}{6} & \frac{\lambda}{4}-\frac{\mu}{4} & \frac{\lambda}{3}+\mu & \frac{-\lambda}{4}-\frac{\mu}{4} & \frac{-\lambda}{3}-\frac{\mu}{2} & \frac{-\lambda}{4}+\frac{\mu}{4} \\ \frac{\lambda}{4}+\frac{\mu}{4} & \frac{-\lambda}{6}-\frac{\mu}{2} & \frac{-\lambda}{4}+\frac{\mu}{4} & \frac{-\lambda}{3}-\frac{\mu}{2} & \frac{-\lambda}{4}-\frac{\mu}{4} & \frac{\lambda}{3}+\mu & \frac{\lambda}{4}-\frac{\mu}{4} & \frac{\lambda}{6} \\ \frac{\lambda}{6} & \frac{-\lambda+\mu}{4} & \frac{-\lambda}{6}-\frac{\mu}{2} & \frac{-\lambda}{4}-\frac{\mu}{4} & \frac{-\lambda}{3}-\frac{\mu}{2} & \frac{\lambda}{4}-\frac{\mu}{4} & \frac{\lambda}{3}+\mu & \frac{\lambda+\mu}{4} \\ \frac{\lambda-\mu}{4} & \frac{-\lambda}{3}-\frac{\mu}{2} & \frac{-\lambda}{4}-\frac{\mu}{4} & \frac{-\lambda}{6}-\frac{\mu}{2} & \frac{-\lambda}{4}+\frac{\mu}{4} & \frac{\lambda}{6} & \frac{\lambda+\mu}{4} & \frac{\lambda}{3}+\mu \end{bmatrix}. \tag{2.49}$$

We can make one observation here, on how to determine the required connectivity of the LEM elements. The matrix R^e does not have a zero block. This is because the degrees-of-freedom in this FEM element are all coupled. Therefore, for the LEM element to be able to have a behavior similar to the FEM element, it needs to be fully connected, as shown in Figure 2.4.

Since the material is assumed to be isotropic, the LEM element has to be symmetric and K^e has only two independent parameters, k^{edge} and k^{diag}, as the following

$$k^{edge} = k^{1,2} = k^{2,3} = k^{3,4} = k^{4,1}, \tag{2.50}$$

$$k^{diag} = k^{1,3} = k^{2,4}. \tag{2.51}$$

If we evaluate (2.39), we get

$$K^e = \begin{bmatrix} k^{aa} & -\frac{k^{dd}}{2} & -k^{ee} & 0 & -\frac{k^{dd}}{2} & \frac{k^{dd}}{2} & 0 & 0 \\ -\frac{k^{dd}}{2} & k^{aa} & 0 & 0 & \frac{k^{dd}}{2} & -\frac{k^{dd}}{2} & 0 & -k^{ee} \\ -k^{ee} & 0 & k^{aa} & \frac{k^{dd}}{2} & 0 & 0 & -\frac{k^{dd}}{2} & -\frac{k^{dd}}{2} \\ 0 & 0 & \frac{k^{dd}}{2} & k^{aa} & 0 & -k^{ee} & -\frac{k^{dd}}{2} & -\frac{k^{dd}}{2} \\ -\frac{k^{dd}}{2} & \frac{k^{dd}}{2} & 0 & 0 & k^{aa} & -\frac{k^{dd}}{2} & -k^{ee} & 0 \\ \frac{k^{dd}}{2} & -\frac{k^{dd}}{2} & 0 & -k^{ee} & -\frac{k^{dd}}{2} & k^{aa} & 0 & 0 \\ 0 & 0 & -\frac{k^{dd}}{2} & -\frac{k^{dd}}{2} & -k^{ee} & 0 & k^{aa} & \frac{k^{dd}}{2} \\ 0 & -k^{ee} & -\frac{k^{dd}}{2} & -\frac{k^{dd}}{2} & 0 & 0 & \frac{k^{dd}}{2} & k^{aa} \end{bmatrix}, \tag{2.52}$$

where $k^{ee} = k^{edge}$, $k^{dd} = k^{diag}$, and $k^{aa} = \frac{k^{diag}}{2} + k^{edge}$. R^e for the FEM element also has two independent parameters, namely, Lame's constants, λ and μ. Then, at first we may think that it should be possible to construct the LEM element which has the same input-output behavior as the FEM element. However, it is not too difficult

to see that this is not true, if we look at the individual terms of the matrices K^e and R^e. Each subblock $R^e_{i,j}$ depends on both of the parameters (λ, μ), but this is not the case for $K^e_{i,j}$, which depends only on single parameter $(k^{i,j})$. Therefore, in block by block sense, the LEM element cannot represent the Poisson's ratio and bulk modulus simultaneously. It is also informative to note that there are some structural differences between these blocks as well, which can be observed from (2.49). $K^e_{i,j}$ is always symmetric (see (2.39)), but same is not true for $R^e_{i,j}$, whose corresponding off-diagonal terms may or may not have the same sign. Also, the diagonal terms of $K^e_{i,j}$ are always positive, but the diagonal terms of $R^e_{i,j}$ may be positive or negative.

The difference between the behavior of the FEM and LEM elements comes from the fact that the interaction between the nodes is restricted to be some form of spring-like behavior in LEM, whereas there is freedom in FEM. This restricts the physical material behavior that LEM models can represent.

It is also possible to consider adding angular springs within the LEM element. This would enrich the behavior of the LEM element. However, addition of angular springs would decrease the computational attractiveness of the LEM because of the increased computational complexity. We leave this for future work.

To approximate FEM element behavior with an LEM element, at least for the linear case, we need to perform the following optimization

$$(k^{edge}, k^{diag}) = \arg \inf_{k^{edge}, k^{diag}} \left\| R^e(\lambda, \mu) - K^e(k^{edge}, k^{diag}) \right\| \tag{2.53}$$

in some norm. There are two different norms used in this study, which are the Frobenius norm and induced 2-norm. The Frobenius norm is the square root of the sum of the absolute squares of its elements, which the elastic parameters can determine by using algebra solver. The induce 2-norm is the square root of the maximum eigenvalue of the matrix conjugate transpose and matrix itself, which the elastic parameters can determine by using optimization technique. The most natural choice would be the induced 2-norm, which looks at the input-output behavior of the

two matrices.

For this particular example, if we use the Frobenius norm, the blocks will be decoupled, and we can get a simple closed form solution for the k^{edge} and k^{diag} values that minimize the error:

$$R^e_{1,2} = \begin{bmatrix} -\frac{\lambda}{3} - \frac{\mu}{2} & -\frac{\lambda}{4} + \frac{\mu}{4} \\ \frac{\lambda}{4} - \frac{\mu}{4} & \frac{\lambda}{6} \end{bmatrix} \quad K^e_{1,2} = \begin{bmatrix} -k^{edge} & 0 \\ 0 & 0 \end{bmatrix} \tag{2.54}$$

gives

$$k^{edge} = \frac{\lambda}{3} + \frac{\mu}{2} \tag{2.55}$$

and

$$R^e_{1,3} = \begin{bmatrix} -\frac{\lambda}{6} - \frac{\mu}{2} & \frac{\lambda}{4} + \frac{\mu}{4} \\ \frac{\lambda}{4} + \frac{\mu}{4} & -\frac{\lambda}{6} - \frac{\mu}{2} \end{bmatrix} \quad K^e_{1,3} = \frac{1}{2} \begin{bmatrix} -k^{diag} & k^{diag} \\ k^{diag} & -k^{diag} \end{bmatrix} \tag{2.56}$$

gives

$$k^{diag} = \left(\frac{\lambda}{6} + \frac{\mu}{2}\right) + \left(\frac{\lambda}{4} + \frac{\mu}{4}\right) = \frac{5\lambda}{12} + \frac{3\mu}{4} \tag{2.57}$$

for the element configurations we assumed.

At this point, it is important to observe that, even though it is possible to find a unique (k^{edge}, k^{diag}) pair for every (λ, μ), it is impossible to make the error equal to zero. This is because of the fact that the stiffness matrices of FEM "elements" and LEM "elements" are structurally different. Therefore, it is not possible to select LEM parameter to exactly match the FEM behavior even though they may have same number of parameters.

If we use the 2-norm or consider different element geometries, the unknown parameters will not be decoupled, and we cannot get a closed form solution; therefore, optimization techniques need to be used to find the best solution. This optimization to determine LEM parameters need to be performed for every different element configuration, since the linearization depends on the geometry of the elements.

We can summarize the proposed method as follows:

(1) We construct the LEM of the deformable object as composed of "elements" or building blocks that are fully connected, rather than having an arbitrarily con-

nected mass-spring mesh. These building block LEM "elements" are used to approximate the stiffness characteristic of FEM "elements" of same size and geometry.

(2) For each of the LEM "elements":

a) A linearized LEM "element" stiffness matrix K^e is calculated using (2.35)-(2.40). This stiffness matrix is parameterized by the (unknown) spring constants of the LEM, and gives the tangent behavior of the LEM.

b) A FEM element with same geometry as the LEM "element" is constructed. The stiffness matrix R^e for the FEM element is calculated with (2.24), using a linear elastic model (e.g. (2.11)). This FEM stiffness matrix is parameterized by the (known) constitutive parameters of the material, and gives the tangent behavior of the FEM.

c) An optimization is performed to identify the LEM element parameters that minimize the error $\|R^e - K^e\|$, similar to (2.53), using a suitable matrix norm.

d) The nodal mass values of the elements are calculated using (2.32), where m^e_{ii} are given by (2.24).

(3) The LEM is "assembled" by adding the nodal mass values and spring constants for overlapping lumped masses and springs from neighboring "elements".

As these optimizations to identify LEM component parameters are conducted off-line when the object model is constructed, it is not impact the on-line computational efficiency of the LEM. This optimization can also be performed at the whole object level instead of per element. However, this would not be practical as its computation cost will be prohibitively large and it will be prone to local minima problems. Finally, the optimization in step 2(c) can also be performed using the nonlinear forms of

$R^e(\hat{\mathbf{x}}^e)$ and $K^e(\hat{\mathbf{x}})$ by evaluating these at a collection of p values, $\hat{\mathbf{x}}_i^e = \hat{\mathbf{x}}_i, i = 1, ..., p$, and minimizing a cost function $\sum_{i=1}^{p} \|R^e(\hat{\mathbf{x}}_i^e) - K^e(\hat{\mathbf{x}})\|^2$, defined with a suitable vector norm. However, this optimization would also be computationally intensive, and therefore has not been pursued in this study.

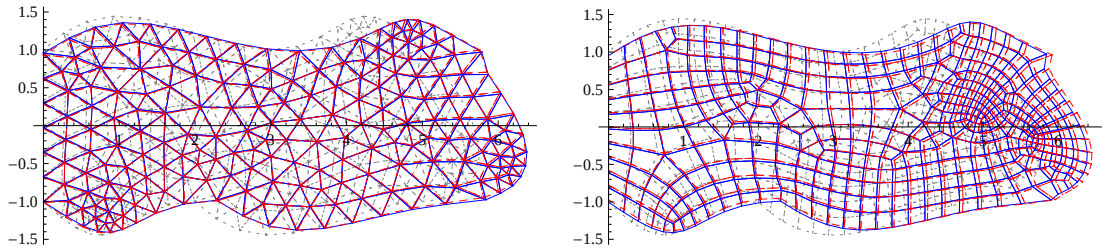
2.4 Simulation Results

In this section, the simulation results were presented to validate and demonstrate our proposed method. The simulation experiments were implemented by using Mathematica, MATLAB, and C++ with GiPSi framework [8] environments. In our simulation, the simulated objects had the Young's modulus of 10 kPa and the Poisson's ratio of 0.3, unless otherwise stated. In two-dimensional examples, a plane stress case was used. Four different configurations of LEM "element" meshes were considered. In the planar examples, triangular and quadrilateral meshes were used and for three-dimensional volumetric object examples, tetrahedral and hexahedral meshes were used. In order to provide quantitative results, the percentage of root mean square error, $\%e_{rms}$, and the percentage of maximum error, $\%e_{max}$, of Euclidean distance between FEM and LEM nodes were calculated by $\%e_{rms} = \frac{e_{rms}}{o_{max}} * 100$ and $\%e_{max} = \frac{e_{max}}{o_{max}} * 100$, where $e_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i^{FEM} - x_i^{LEM})^2}$, $e_{max} = \max|x_i^{FEM} - x_i^{LEM}|$, o_{max} is the maximum Euclidean distance between the undeformed configuration and the deformed configuration of FEM, n is the number of nodes in the models, and x_i^{FEM} and x_i^{LEM} denote the positions of corresponding nodes of the FEM and LEM. In the figures showing simulation results, the color and type of line identify the object configuration. The original configuration of object is shown with dotted gray lines, the deformed configuration of the FEM is shown with dashed blue lines and the deformed configuration of the LEM is shown with solid red lines. Simulation results which compared the deformation of various two and three-dimensional test

objects were reported in section 2.4.1. The use of two different matrix norms during the optimization in step 2(c) of the algorithm were also compared in this section. The norms used in experiments were the Frobenius norm and the induced 2-norm. The elastic parameters were calculated symbolically in the Frobenius norm case using Mathematica and numerically in the induced 2-norm case using MATLAB. In section 2.4.2, specially designed specimen were used to determine the mechanical properties of object using the proposed method. In section 2.4.3, if the Young's modulus and Poisson's ratio can be independently set was explored. In section 2.4.4, the comparison of the torsion loading effect in linear elastic and nonlinear lumped element models was observed. In section 2.4.5 and 2.4.6, the effect of resolutions and dimensions in LEM were observed. Finally in section 2.4.7, simulation results which compared our method to other similar methods in the literature by using tension, shearing, and torsion tests were presented.

2.4.1 Test Objects

We demonstrated our proposed method with variety of test objects. In the two-dimensional case, we used the object shown in Figure 2.6, which represented an arbitrarily shaped soft tissue approximately 2x6 cm^2 in size. In the three-dimensional case, three objects were used. The first object was a cylindrical object with a radius of 1 cm and a height of 2 cm shown in Figures 2.7a and 2.7b. This object was discretized with tetrahedral (Figure 2.7a) and hexahedral (Figure 2.7b) elements. The second object was a cylinder with the same dimension as the first object but with an empty spherical hole of radius 0.6 cm inside (Figure 2.7c), which was discretized with tetrahedral elements. The third object was a cube with dimensions of 2x2x2 cm^3 (Figure 2.8), which was discretized with tetrahedral (Figure 2.8a) and hexahedral (Figure 2.8b) elements. In each object, the boundary was kept fixed on one side and the tension force was applied in opposite side to make the object deformed at least



(a) (b)

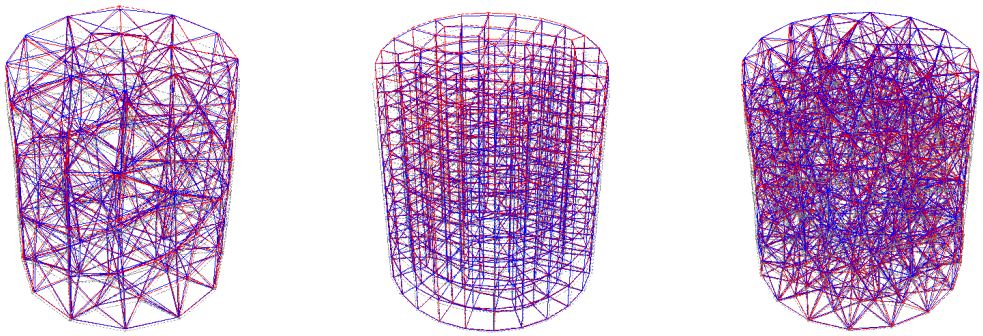
Figure 2.6: Two-dimensional test object results: (a) Mesh with triangular elements calculated using 2-norm; and (b) Mesh with quadrilateral elements calculated using with F-norm.

10% of original configuration. The simulation results were shown in Figures 2.6, 2.7 and 2.8 and quantitative results were summarized in Table 2.1. When we compared the effect of the matrix norm used in the optimization, we observed that the 2-norm had yielded better results in triangular and hexahedral elements, whereas the F-norm had yielded better results in quadrilateral and tetrahedral elements.

Table 2.1: Simulation results of test objects.

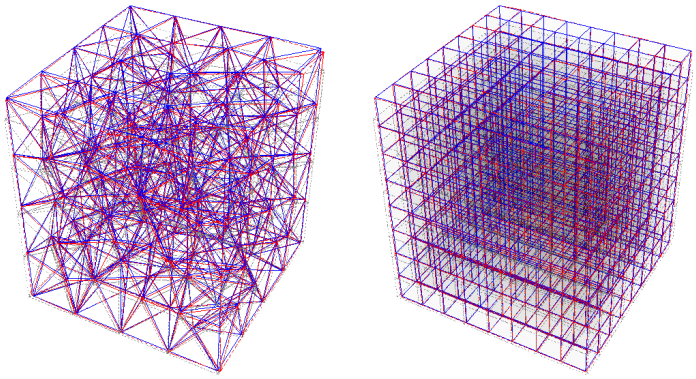
test cases	elements	norm	$\%e_{rms}$	$\%e_{max}$	#node	#element
Tissue	tri	2-norm	2.26	4.99	210	366
		F-norm	3.46	6.42		
	qua	2-norm	5.55	9.65	445	405
		F-norm	5.14	8.44		
Cylinder	tet	2-norm	22.89	43.73	124	381
		F-norm	11.00	24.29		
	hex	2-norm	5.63	18.49	675	496
		F-norm	6.15	18.00		
Cylinder with hole	tet	2-norm	20.53	38.96	404	1,559
		F-norm	5.91	13.77		
Cube	tet	2-norm	22.52	53.51	170	592
		F-norm	8.28	25.29		
	hex	2-norm	2.91	6.92	729	512
		F-norm	14.98	25.19		

In order to illustrate the time complexity of our proposed parameter estimation method, the computation time for each element type and matrix norm was recorded. The experiment was conducted by using Mathematica 7.0 and MATLAB R2007b and tested on the Microsoft Windows Server 2008™ 64-bit based workstation with the



(a) (b) (c)

Figure 2.7: Three-dimensional test object results: (a) Cylindrical object with tetrahedral elements calculated using F-norm; (b) Cylindrical object with hexahedral elements calculated using 2-norm; and (c) Cylindrical object with a hole with tetrahedral elements calculated using F-norm.



(a) (b)

Figure 2.8: Three-dimensional test object results: (a) Cube object with tetrahedral elements calculated using F-norm; and (b) Cube object with hexahedral elements calculated using 2-norm.

Intel Pentium Dual-Core E5200 2.50 GHz, 3.24 GB of RAM to collect computation time benchmarks. The computation time was calculated by averaging the time used in finding the solution for 100 randomly constructed elements. Both 2-norm and F-norm used Mathematica to build the element. However, 2-norm used the Mathematica solver whereas F-norm used the MATLAB solver called from Mathematica. The results were shown in Table 2.2. The computation time of F-norm was sig-

nificantly less than the computation time of 2-norm. Because the computation for determination by using F-norm can be solved with algebra solver (as illustrated in section 2.3), whereas the computation for determination by using 2-norm is performed numerically to find the best solution. The number of spring constants that have to be determined also effects the computational time.

Table 2.2: Time complexity of each element type with F-norm and 2-norm.

		time used/element(s)	
element	#spring constant	F-norm	2-norm
Triangular	3	0.08	1.73
Quadrilateral	6	0.32	2.26
Tetrahedral	6	0.32	2.33
Hexahedral	28	3.04	29.01

2.4.2 Mechanical Tests on Specimens

It was also valuable to measure the resulting Young's modulus and Poisson's ratio of the constructed simulation models of objects to evaluate how well they approximated the original values used. In order to accurately measure these parameters, we had conducted mechanical tests on specially designed specimens, following the experimental material characteristic literature. Specifically, the specimen shape chosen was an optimal shape of thin tensile test specimen from [1]. The specimen shape shown in Figure 2.9 with B=1 cm. The thickness of the three-dimensional specimen was 1 cm. The boundary on the left side was kept fixed and the tension force for loading was applied on the right side. The experiment procedure was as follows: First, Young's modulus of 10 kPa and Poisson's ratio of 0.3 were selected. Then, the spring constants of the specimens were determined using the proposed method. The boundary conditions and the loading forces were applied to the lumped element model and the displacements of the straight-sided section L_s were collected. These values were then used to calculate the corresponding Young's modulus and Poisson's ratio of specimen.

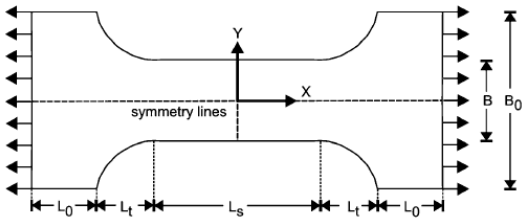


Figure 2.9: Specimen shape [1] with $L_s = 2B$, $L_t = 4B$, $L_0 = B_0$, and $B_0 = 3B$.

The experimental results shown in Table 2.3. The results shown that the proposed method can represent the Young's modulus close to the desired value in meshes with triangular (2D) and hexahedral (3D) elements with both F-norm and 2-norm and in meshes with quadrilateral and tetrahedral elements with F-norm. The quadrilateral and tetrahedral elements with 2-norm resulted in error in the Young's modulus more than 10%.

Table 2.3: Tension tests on specimen.

element	norm	E_{out}	ν_{out}	#node	#element
Triangular	2-norm	10,300.70	0.25	125	192
	F-norm	9,461.44	0.26		
Quadrilateral	2-norm	11,169.70	0.45	329	266
	F-norm	9,839.55	0.43		
Tetrahedral	2-norm	12,165.80	0.09	538	1,500
	F-norm	9,697.49	0.09		
Hexahedral	2-norm	9,804.64	0.29	1,396	843
	F-norm	10,228.00	0.29		

2.4.3 Control of Young's Modulus and Poisson's Ratio

In order to explore the relationship between the Young's modulus and Poisson's ratio of lumped element models and how independently they can be controlled, we conducted another set of tests. Specifically, we used a lumped element model with only one quadrilateral element (with size of 2x2 cm^2) and varied the desired elastic properties (E_{in} = 1 Pa–10 kPa, ν_{in} = 0–0.5). Then, the elastic properties (E_{out}, ν_{out}) of the resulting model were compared with the desired values. The results, shown in

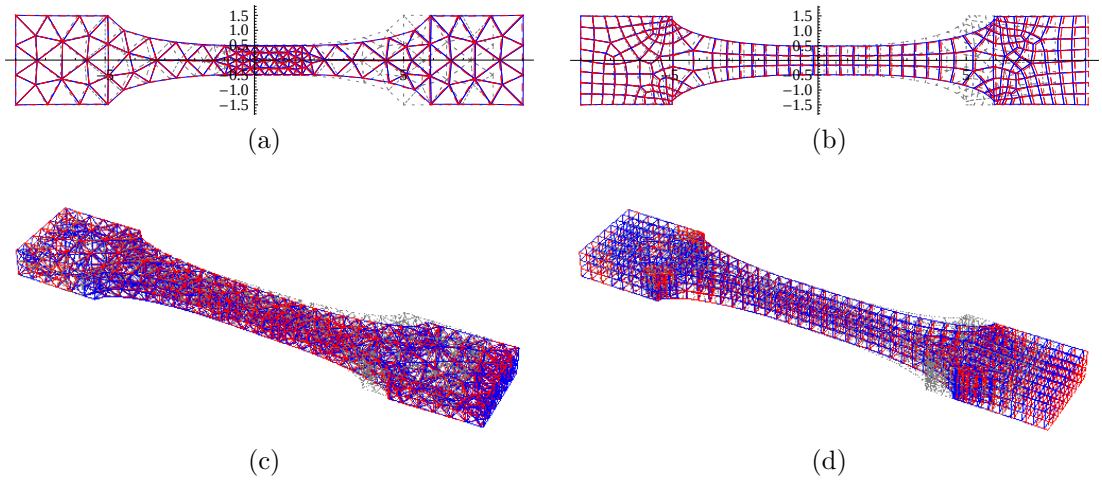


Figure 2.10: Tension test on specimens: (a) Triangular element; (b) Quadrilateral element; (c) Tetrahedral element; and (d) Hexahedral element.

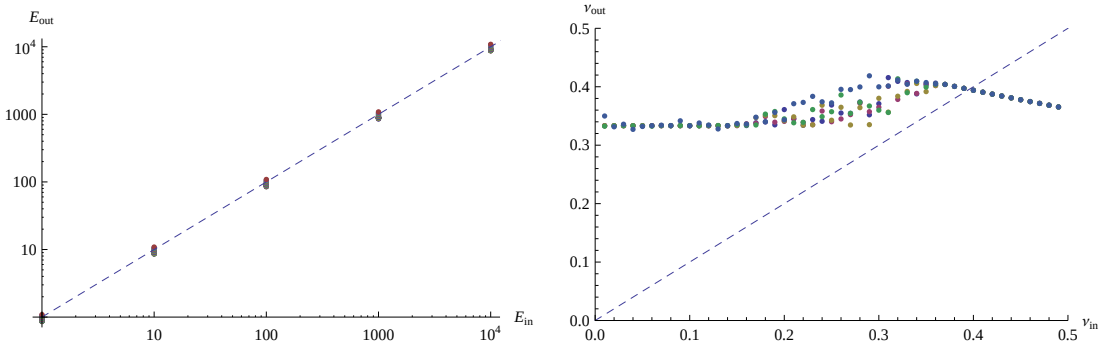
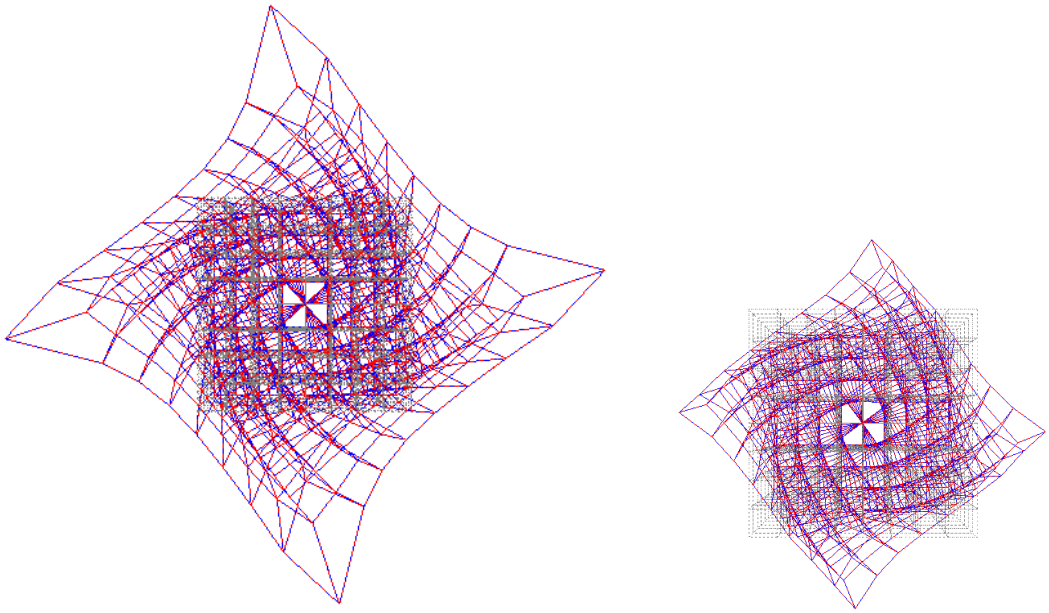


Figure 2.11: Young's modulus (a) and Poisson's ratio (b) of lumped element model with quadrilateral element.

Figure 2.11, revealed that, although it was possible to accurately control the Young's modulus value of the model, it was not possible to independently control the Poisson's ratio. Such a behavior was also observed by other researchers [41, 42, 44].

2.4.4 Torsion Loading Effect

In order to explore the torsion loading effect between the linear elastic model and lumped element model, we also did another experiment by applying the torsion force on top surface of cube object with fixed boundary at the bottom of object. The



(a) (b)

Figure 2.12: The top view of cube with torsion loading: (a) linear elastic model; and (b) nonlinear lumped element method.

cube object used in this experiment was the same as cube object in test case with hexahedral element. The experiment result shown that the linear elastic model was not suitable for torsion loading but it was not the problem of nonlinear lumped element models (Figure 2.12). When the torsion loading applied to model to make the large deformation by rotating on top surface and fixed position of base. The result of elastic linear model behaved unnatural behavior. While lumped element model behaved like nonlinear model because of the nonlinearity of lumped element model. Therefore, we can use lumped element model to imitate the nonlinear object to produce the natural behavior.

2.4.5 Resolution Effect

To test the consistency and accuracy of our method when the resolutions of object were varied, the objects were discretized at different resolutions of 1x1, 2x2, 4x4, 8x8 in two-dimensional square object with dimension of 2x2 cm^2 and different resolutions

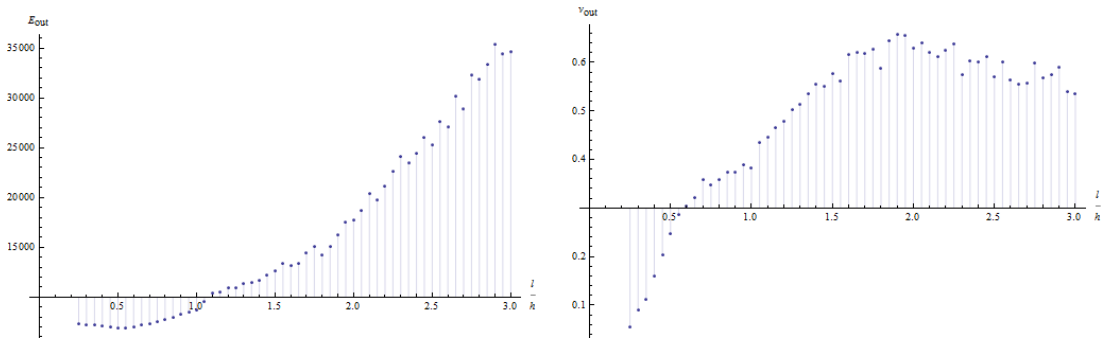
of 1x1x1, 2x2x2, 4x4x4, 8x8x8 in three-dimensional cube object with dimension of 2x2x2 cm^3. The experimental results shown in Table 2.4. The results shown that the more detail resolution, the closer behavior of lumped element model to finite element model. Another advantage of using identical element was the reduction of time used for determination the spring constants of the object.

Table 2.4: Resolution effect.

		$\%e_{rms}$	
objects	resolutions	2-norm	F-norm
Square	1x1	11.64	16.35
	2x2	10.27	14.48
	4x4	9.36	13.44
	8x8	9.08	12.92
Cube	1x1x1	8.68	18.00
	2x2x2	7.13	16.38
	4x4x4	3.14	15.50
	8x8x8	2.90	15.03

2.4.6 Element Shape Effect

To test the ability to represent the elastic properties at different dimensions by using our method to determine the spring constants, the experiment used only one quadrilateral element with fixed height of 2.0 cm and varied length of element from 0.5 cm to 6.0 cm with the same desired elastic properties (E=10 kPa, ν=0.3). The elastics properties were compared the desired values and output values to see the effect of dimensions. The boundary condition was fixed on left side and the tension loading force was applied on right side to deform the object about 10%. After applied loading, the Young's modulus and Poisson's ratio were calculated from the deformed object. The results in Figure 2.13 shown that the Young's modulus was closed to 10 kPa when the dimension ratio (l/h) of element was about 1.0, and Poisson's ratio output was about 0.3 when the dimension ratio (l/h) of element was about 0.6. In another word, a square element was the best for representing the Young's modulus, while a



(a) (b)

Figure 2.13: Young's modulus and Poisson's ratio output from element shape effect experiment.

rectangle element of dimension ratio 0.6 was the best for representing the Poisson's ratio.

2.4.7 Comparison with Other Methods

As discussed in section 2.1, there were several other studies in the literature that proposed methods to determine elasticity parameters in lumped element models. In the two-dimensional case, we compared our proposed method with the methods proposed by Van Gelder [39] and Lloyd et al. [44] for meshes with triangular elements (test object shown in Figure 2.10a), and the methods proposed by Baudet et al. [41] and Lloyd et al. [44] for the meshes with rectangular elements (test object shown in Figure 2.10b). We also compared our proposed method in three-dimensional case with the method proposed by Lloyd et al. [44] for meshes with tetrahedral elements (test object shown in Figure 2.10c), and with the method proposed by Baudet et al. [42] for meshes with hexahedral elements (test object shown in Figure 2.8b) [†]. The comparison results were shown in Table 2.5. The proposed method gave the best result except for the case with quadrilateral elements in two dimensions, when it gave

[†]In the case for meshes with hexahedral elements, the object in Figure 2.8b was used instead of the test specimen in Figure 2.10d, since the method by Baudet et al., was only for uniform (i.e. cubic) shaped elements

the second best result, and the case with tetrahedral element in three dimensions. For the quadrilateral mesh case, it should be noted that the method proposed by Baudet et al. introduced a correction force which was not from a spring, and hence it was not a pure lumped element model, and it was applicable only for "cubic" elements. Also, for the tetrahedral mesh case our proposed method resulted in an effective Young's modulus value which was close to the desired value than the method by Lloyd et al., even though the actual $\%e_{rms}$ value was slightly higher.

Table 2.5: Comparison with other methods.

element	method	E	ν	$\%e_{rms}$	$\%e_{max}$
Triangular	Van Gelder	15,010.80	0.27	20.08	34.58
	Lloyd et al.	10,855.80	0.29	5.61	10.88
	Our (2-norm)	10,300.70	0.25	3.02	5.65
Quadrilateral	Baudet et al.	10,330.50	0.34	1.06	2.07
	Lloyd et al.	13,097.10	0.44	4.84	8.20
	Our (2-norm)	11,169.70	0.45	3.36	5.50
Tetrahedral	Lloyd et al.	8,671.48	0.03	7.75	14.65
	Our (f-norm)	9,697.49	0.09	9.37	15.72
Hexahedral	Baudet et al.	10,712.50	0.12	3.67	6.99
	Our (2-norm)	10242.00	0.14	2.88	6.05

We also did the experiments on shearing and torsion tests and comparison with other methods. In the two-dimensional case, we used a 2x2 cm^2 square object with triangular elements (101 nodes, 168 elements) and with quadrilateral element (121 nodes, 100 elements). In the three-dimensional case, we used a 2x2x2 cm^3 cube object with tetrahedral elements (170 nodes, 592 elements) and with hexahedral elements (729 nodes, 512 elements). The results of these test were given in Table 2.6. Our proposed method in shearing test gave the best result in triangular and tetrahedral elements, and comparable results in hexahedral element. In torsion test, the linear FEM reference did not work well with torsion test, whereas the LEM can handle the torsion test better. The result when compares with FEM gave the large error on all methods.

Table 2.6: Shearing and torsion tests in comparison with other methods.

		Shearing		Torsion	
element	method	$\%e_{rms}$	$\%e_{max}$	$\%e_{rms}$	$\%e_{max}$
Tri	Van Gelder	17.25	34.17	-	-
	Lloyd et al.	5.04	10.73	-	-
	Our 2-Norm	3.68	8.60	-	-
Qua	Baudet et al.	3.60	8.28	-	-
	Lloyd et al.	3.64	8.85	-	-
	Our 2-Norm	3.82	11.28	-	-
Tet	Lloyd et al.	5.68	15.16	7.80	26.40
	Our F-Norm	5.30	14.55	8.59	34.69
Hex	Baudet et al.	3.86	9.36	8.58	44.61
	Our 2-Norm	3.96	9.37	8.36	43.91

2.5 Conclusion

A systematic method to determine mass and spring constants of lumped element models of deformable objects was presented. The lumped element model parameters were determined using a finite element model as a reference model by minimizing the error the stiffness matrices of the finite element and lumped element models through an optimization. The proposed method was demonstrated by several test objects in two and three dimensions with triangular, quadrilateral, tetrahedral, and hexahedral elements. Using 2-norm yielded better results in triangular and hexahedral elements, whereas using F-norm yielded better results in quadrilateral and tetrahedral elements. The time complexity by using F-norm was shown to be an order-of-magnitude less than using 2-norm. It was also shown that, with the proposed method, the Young's modulus of the objects was well approximated. However, it was impossible to control the Poisson's ratio of the lumped element model independently as it was also observed by other researchers [41, 42] and [44]. Moreover, the proposed method was also compared with existing methods in the literature [39, 41, 42, 44].

Chapter 3

Numerical Integration Methods for Deformable Object Models

3.1 Introduction

A surgical simulation contains many simulation objects, which mimic the human body organs or living tissues. Most simulation objects are deformable objects, which can change the internal structure by the external forces. The deformable objects are modeled by using the physically based models which define the behavior of the objects using mathematical equations and the laws of physics. These models are expressed in the form of differential equations. During simulation, numerical integration is used to find the positions and velocities of the deformable object from the system of equations defining the model. There are many numerical integration methods for solving differential equations. These algorithms can be broadly divided into two categories: the explicit and implicit numerical integration methods. Each method has advantages and disadvantages depending on the model and the simulated system. Therefore, the comparison of the stability and efficiency of numerical integration methods is useful for the user to select an efficient numerical method and suitable simulation time

step to guarantee the stability of physically based model of the deformable object in simulation.

In this chapter, we present the comparison of the numerical integration methods in a lumped element based model as a physically based deformable object model. However, the results are generalizable to simulation of other types of physically based deformable object model. The remainder of this chapter is organized as follows: In the next section, the related studies in the literature on numerical integration methods for deformable object simulation are discussed in section 3.2. The physically based deformable object model is presented in section 3.3. The numerical integration methods, which include both explicit and implicit numerical integration methods, are presented in detail in section 3.4. Then, the implementations of implicit numerical integration methods are presented in section 3.5. After that, the experimental results are presented in section 3.6, followed by concluding remarks in section 3.7.

3.2 Related Works

Terzopoulos et al. [14] pioneered using elasticity properties to model deformable objects and used an implicit integration method to solve the resultly system of equations. Later many researchers used explicit integration methods to create real-time simulations because the explicit integration method is easy to implement and results in less computation per simulation time step. For example, Provot [16] used an explicit Euler integration method, Volino et al. [55] used an explicit Midpoint integration method, and Eberhardt et al. [56] used the forth order of Runge-Kutta integration method. Implicit integration method became popular again after the work of Baraff and Witkin [57] which used mass-spring models and implicit integration in cloth simulation to avoid instability of explicit integration method. Implicit integration methods require significant computation for each time step and a large

storage space. In order to address this issue, Eberhardt et al. [58] developed the implicit-explicit (IMEX) integration method. The idea of this method is to separate the stiff parts of the system of equations for implicit method and the non stiff part for explicit method. Volino and Magnenat-Thalmann [59] compared different numerical integration methods in cloth simulation. Hauth [60] reviewed numerical integration techniques for cloth simulation and compared different numerical integration methods in three simple examples.

In surgical simulations, deformable objects are modeled by using physical properties, which is called the physically based modeling. There are many physically based models, such as lumped element models, finite difference models, finite element models, and mesh free models. Nealen et al. [61] provided a recent review of the physically based deformable model used in computer graphics. The lumped element models (LEM) or mass spring damper models (MSD) are popular because they are easy to implement. However, the lumped element models cannot represent the real physical properties of real deformable objects. The finite different, finite element, and mesh free models are used to simulate the physical properties of real deformable objects but their time complexity is higher than the lumped element models.

3.3 Physically Based Deformable Object Model

The physically based deformable object model used in this study is a lumped element model. A lumped element model is a system of masses connected with springs. Each spring has physical properties, which are stiffness constant (k), and damping constant (b). For two masses connected with one spring (Figure 3.1), the spring force of node p is

$$\mathbf{f}_s(\mathbf{p},\mathbf{q}) = k\left(\|\mathbf{p}-\mathbf{q}\| - L_0\right)\frac{(\mathbf{q}-\mathbf{p})}{\|\mathbf{p}-\mathbf{q}\|}, \tag{3.1}$$

where k is the spring constant, $\mathbf{p}$ and $\mathbf{q}$ are the position of nodes p and q, and L_0 is the rest length of spring. The damping force of node p is

$$\mathbf{f}_d(\mathbf{p},\mathbf{q},\mathbf{v},\mathbf{w}) = b\left((\mathbf{w}-\mathbf{v})^T\frac{(\mathbf{q}-\mathbf{p})}{\|\mathbf{p}-\mathbf{q}\|}\right)\frac{(\mathbf{q}-\mathbf{p})}{\|\mathbf{p}-\mathbf{p}\|}, \tag{3.2}$$

where b is the damping constant, $\mathbf{p}$ and $\mathbf{q}$ are the position of nodes p and q, and $\mathbf{v}$ and $\mathbf{w}$ are the velocity of nodes p and q. We define $\mathbf{x}$ as the position vector of all nodes in the lumped element model, and $\dot{\mathbf{x}}$ as the corresponding velocity vector. If we consider a simple one-dimensional mass spring damper system with a single mass, the equation of motion is in the form

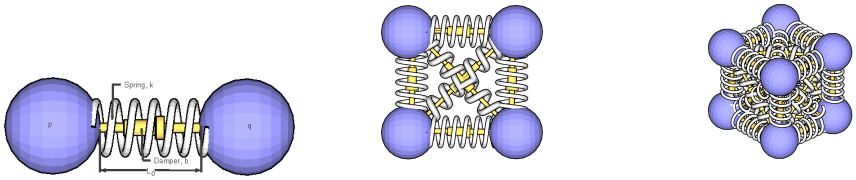
$$m\ddot{x} + b\dot{x} + kx = f_{ext}, \tag{3.3}$$

where m is the mass, b is the damping constant, k is the spring constant, and f_{ext} is an external force, such as the gravity force or pushing force from user. The equation of motion can be simplified by defining the natural frequency: $\omega_0 = \sqrt{k/m}$, the damping coefficient: $\zeta = b/2\sqrt{km}$. When no external force applies, the equation of motion becomes:

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2 x = \mathbf{0}, \tag{3.4}$$

and the solution assumes in the form $x = e^{\gamma t}$, where $\gamma = \omega_0(-\zeta \pm \sqrt{\zeta^2-1})$. The behavior of the system is depended on the natural frequency (ω_0) and the damping coefficient (ζ). When $\zeta = 1$, the system is critical damped and the solution is in the form $x = (C_1 + C_2 t)e^{-\omega_0 t}$. When $\zeta > 1$, the system is over damped and the solution is in the form $x = C_1 e^{\gamma_1 t} + C_2 e^{\gamma_2 t}$. And when $\zeta < 1$, the system is under damped and the solution is in the form $x = e^{-\omega_0\zeta t}(C_1 cos(\omega_d t) + C_2 sin(\omega_d t))$ and $\omega_d = \omega_0\sqrt{1-\zeta^2}$.

The deformable object is modeled by using masses connected with springs (Figure 3.1). The input parameters of lumped element model are the density (ρ) with unit g·cm^{-3}, Young's modulus (E) with unit g·cm^{-1}·s^{-2}, and damping coefficient (ζ) without unit. In the analysis and experiments presented in section 3.6, the mass (m) with unit g, stiffness constant (k) with unit g·s^{-2} and damping constant (b)



(a) (b) (c)

Figure 3.1: Lumped element models: (a) One dimension; (b) Two dimensions; and (c) Three dimensions.

with unit $\mathrm{g{\cdot}s^{-1}}$ are calculated by assuming that the object is an isotropic material as follows. The dimensions of object are dx, dy and dz. For two-dimensional model, the thickness is 0.1 cm. The resolutions of object are ex, ey, and ez. The mass of object is $m = \rho \cdot dx \cdot dy \cdot dz$. The Young's modulus equals to the stress over the strain. If the force is applied in z direction, the stiffness constant is $k = E\frac{dx \cdot dy}{dz}$. The damping constant is $b = 2\zeta\sqrt{km}$. After all of physical properties are known, the mass is distributed to each node and spring properties are set equally for every spring element.

3.4 Numerical Integration

The system state contains positions and velocities. The system state is defined as the following:

$$\mathbf{y}(t) = \begin{bmatrix} \dot{\mathbf{x}}(t) \\ \mathbf{x}(t) \end{bmatrix}, \tag{3.5}$$

where $\mathbf{y}(t)$ is the system state at time t, which contains the velocity $\dot{\mathbf{x}}(t)$ and position $\mathbf{x}(t)$. The second derivative of position, $\ddot{\mathbf{x}}(t)$ is acceleration. The dynamics in the simulation usually use the fundamental laws of mechanics to describe the behavior of the simulation objects. The Newton's second law is stated as follows:

$$\mathbf{f}(t) = m\ddot{\mathbf{x}}(t) = m\mathbf{a}(t), \tag{3.6}$$

where $\mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}$ are the vector of position, velocity and acceleration of mass nodes and $\mathbf{f}$ is the sum of the forces applied to the mass nodes.

Combining the equations from all node masses yields a system of differential equations. The implementation of efficient numerical integration methods to solve these differential equations is a key for efficient simulation systems. The issue of numerical integration is the numerical error which accumulates in each simulation time step. These numerical errors may instability for large time steps. The following sections are the detail and comparison of each numerical integration method used in this study.

3.4.1 Explicit Integration Methods

Explicit integration methods are the simple methods to solve the differential equations. Explicit integration methods calculate the next system state information from the current system state information. The examples of explicit integration methods are Euler, Midpoint, Heun and Runge-Kutta methods. The explicit integration method can be derived from Taylor series,

$$y(t+h) = y(t) + h \cdot y^{(1)}(t) + \frac{h^2}{2!} \cdot y^{(2)}(t) + \frac{h^3}{3!} \cdot y^{(3)}(t) + \frac{h^4}{4!} \cdot y^{(4)}(t) + \ldots, \quad (3.7)$$

where y is the system state, $y^{(i)}$ is the derivative order i of system state, and h is time step size.

Explicit Euler Method

Explicit Euler method or forward Euler method is simplest and easiest numerical integration method to numerically solve the ordinary differential equation. This method is derived from Taylor series expansion (3.7) through the term $O(h^2)$ which are

$$y_{n+1} = y_n + h \cdot f(t_n, y_n), \quad (3.8)$$

where y is the system state which consists of positions and velocities of the system, h is time step size, and $f(t_n, y_n)$ is the derivative of the system state which is velocities

and accelerations. $O(h^2)$ is the simulation error which is proportional to h^2. The explicit Euler method is an explicit one-step formula because it uses the system state at current time step to calculate the next system state. To reduce the error and maintain the stability of this method, the step size should be decreased, which proportionally increases the computation time.

Explicit Midpoint Method

Explicit midpoint method is an explicit two-step formula because it uses the system state in two past time steps to calculate the next system state. This method uses a centered difference estimation between n and $n+1$. This method is derived from Taylor series expansion (3.7) through the term $O(h^3)$ which are

$$\begin{aligned} k_1 &= y_n + \frac{h}{2} \cdot f(t_n, y_n), \\ y_{n+1} &= y_n + h \cdot f(t_n + \frac{h}{2}, k_1), \end{aligned} \tag{3.9}$$

where k_1 is an estimation of the state by using an explicit Euler step with a half of time step size.

Heun Method

Heun method is an explicit three-step formula. The error per step is $O(h^4)$. The equations for computing the next system state are

$$\begin{aligned} k_1 &= y_n + \frac{h}{3} \cdot f(t_n, y_n), \\ k_2 &= y_n + \frac{2h}{3} \cdot f(t_n + \frac{h}{3}, k_1), \\ y_{n+1} &= y_n + \frac{h}{4} \left(f(t_n, y_n) + 3f(t_n + \frac{2h}{3}, k_2) \right). \end{aligned} \tag{3.10}$$

Runge-Kutta Method

The forth order of Runge-Kutta method is an explicit four-step formula. It is a popular method for integration to achieve a high accuracy. The error per step is

$O(h^5)$. The equations for computing are

$$
\begin{aligned}
k_1 &= h \cdot f(y_n, t_n), \\
k_2 &= h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}), \\
k_3 &= h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}), \\
k_4 &= h \cdot f(t_n + h, y_n + k_3), \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).
\end{aligned} \tag{3.11}
$$

Semi-Explicit Euler Method

Semi-explicit Euler method is an integration method that updates the velocity by using the acceleration at current time step and updating the position by using the velocity at the next time step. The equations are

$$
\begin{aligned}
v_{n+1} &= v_n + h \cdot f(x_n), \\
x_{n+1} &= x_n + h \cdot v_{n+1}.
\end{aligned} \tag{3.12}
$$

3.4.2 Implicit Integration Methods

In explicit integration methods, a small time step is typically required to maintain numerical stability. To avoid this problem, implicit integration methods are used. The implicit integration method calculates the next system state information from the current and the next system state information. Therefore, implicit integration methods require to solve the system of nonlinear algebraic equations to find the value of the system state in the next time step.

Implicit Euler Method

Implicit Euler or backward Euler integration method can use the larger time steps than the explicit integration methods. However, the space and time complexity of

implicit Euler methods are very intensive because it requires to solve a system of nonlinear algebraic equations. The implicit Euler equation is

$$y_{n+1} = y_n + h \cdot f(t_{n+1}, y_{n+1}), \tag{3.13}$$

where y is the system state, h is time step size, and $f(t_{n+1}, y_{n+1})$ is the derivative of the system state at the next time step. The function in the next time step can be approximated by linearizing the nonlinear system of algebraic equations by using Taylor series expansion,

$$f(t_{n+1}, y_{n+1}) \cong f(t_n, y_n) + \frac{\partial f(t_n, y_n)}{\partial y} \triangle y, \tag{3.14}$$

where $\triangle y$ is the difference of system states between next and current time step, $y_{n+1} - y_n$. Replacing Eq. (3.14) with Eq. (3.13), the resulting linearized system of equation is

$$\left(\mathbf{I} - h \cdot \frac{\partial f(t_n, y_n)}{\partial y} \right) \triangle y = h \cdot f(t_n, y_n), \tag{3.15}$$

which is in the form $\mathbf{A} \triangle y = \mathbf{b}$. This linear equation can be solved using, such as the conjugate gradient method, to find the solution of $\triangle y = y_{n+1} - y_n$. The system state at the next time step is then calculated as $y_{n+1} = \triangle y + y_n$. If the state vector is defined as in (3.5), the linearized system of equations become

$$\begin{bmatrix} \mathbf{I} - \frac{h}{m} \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} & -\frac{h}{m} \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \\ -h\mathbf{I} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \triangle \dot{\mathbf{x}} \\ \triangle \mathbf{x} \end{bmatrix} = h \begin{bmatrix} \frac{\mathbf{f}_0}{m} \\ \dot{\mathbf{x}}_0 \end{bmatrix}, \tag{3.16}$$

where $\frac{\partial f}{\partial \dot{\mathbf{x}}}$ is Jacobian respect to velocity ($\dot{\mathbf{x}}$), $\frac{\partial f}{\partial \mathbf{x}}$ is Jacobian respect to position ($\mathbf{x}$), h is step size, m is mass, $\mathbf{I}$ is an identity matrix. Instead of solving the linearized system, it is also possible to improve the accuracy of the solution by iteratively solving the original nonlinear system of equation using the Newton's method. The algorithm for an implicit Euler integration using the Newton's method is as follows:

Algorithm 1 Implicit Euler integration algorithm with Newton's method

```
λ ← 1
while (‖b⃗‖ ≥ ε) or (‖x⃗‖ ≥ ε) do
    x⃗ ← A^{-1} b⃗                          ▷ by using conjugate gradient
    x⃗ ← x⃗ + λx⃗
    b⃗ ← b⃗ − λx⃗
    if ‖b⃗‖ ≥ ε then
        λ ← 0.5λ
    end if
end while
```

Implicit Midpoint Method

The implicit Midpoint method is a second order accurate implicit numerical integration method. The implicit Midpoint equations are,

$$k_1 = h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}),$$
$$y_{n+1} = y_n + k_1. \tag{3.17}$$

The resulting linearized system of equation is given by

$$\left(\mathbf{I} - \frac{h}{2} \cdot \frac{\partial f(t_n, y_n)}{\partial y}\right)(y_{n+1/2} - y_n) = \frac{h}{2} \cdot f(t_n, y_n), \tag{3.18}$$
$$\left(\mathbf{I} - \frac{h}{2} \cdot \frac{\partial f(t_n, y_n)}{\partial y}\right)(\triangle y_n - \triangle y_{n-1/2}) = \frac{h}{2} \cdot f(t_{n+1/2}, y_{n+1/2}) - \triangle y_{n-1/2}, \tag{3.19}$$

where $\triangle y_{n-1/2} = y_{n+1/2} - y_n$ and $\triangle y_n = y_{n+1} - y_{n+1/2}$.

3.5 Implementation

Numerical integration algorithms have been implemented and integrated into GiPSi [8] since the first release in 2002. However, the original version of GiPSi (version 1.0) only included explicit numerical integration methods (Figure 3.2), which were Euler, Midpoint, Heun, and forth order of Runge-Kutta methods. As part of this research, several new numerical integration methods are added into GiPSi, including semi-explicit Euler, implicit Euler, implicit Euler with Newton's method, and implicit

Midpoint methods (Figure 3.3). The implementations of explicit numerical integration methods are straightforward, which calculate the system state from the previous system state as shown in section 3.4. The semi-explicit Euler integration method is also implemented in the same way as explicit integration method, but using the new velocity instead of the previous velocity to calculate the new position. The implementation of implicit numerical integration is more involved because the next system state is calculated from the current system state and the next system state itself. This, after linearization, creates the system of linear algebraic equations that need to be solved by a matrix solver. We use the conjugate gradient algorithm to solve the system of linear equations in each simulation time step of implicit integration method.

`Integrator` provided in GiPSi version 1.0 is the based integrator class using `S` as a class template for any simulation object. Each simulation object provides the specific functions to allocate system state memory, calculate the system state derivative, and calculate the system state accumulation. The `Integrator` based class has a virtual `Integrate()` function for integrating the system state in one simulation time step. Every derived `Integrator` class needs to implement this function. Using the provided API, the implementation of `Integrate()` function in each integration method is simple as the formula provided in section 3.4 because they use these generic functions to perform the integration for any simulation object. The difficult part is the implementation of the generic functions in each simulation object. Specifically, the simulation object needs to provide `AllocState()` function for allocating the memory for the system state, `GetState()` function for getting the system state, `DerivState()` function for computing the derivative of the system state, and `AccumState()` function for computing the accumulation of the system state.

In GiPSi version 2.0, we add `ImplicitIntegrator` as a derived integrator class for implicit integration methods by adding more specific functions, which is necessary for

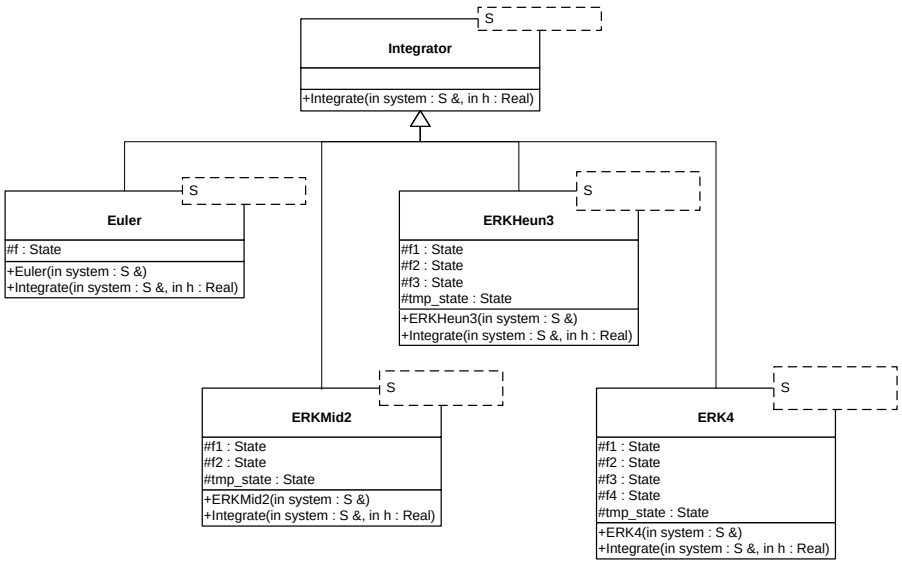


Figure 3.2: Original integrator class diagram.

solving the system of linear equations in implicit integration algorithms. The auxiliary functions for manipulating the system state are inherited from the based `Integrator` class. The `ImplicitIntegrator` introduces the auxiliary function related to implicit integration, namely `AllocJacobian()` function for allocating the memory for Jacobian matrix, `IdentityMinusJacobian()` function for computing identity matrix minus the simulation time step multiplied with the Jacobian matrix, `ScaleState()` function for scaling the system state, and `AddState()` function for adding the system state. The `ImplicitEuler`, `ImplicitEulerNT`, and `ImplicitMidPoint` are derived from the `ImplicitIntegrator` class.

In `IdentityMinusJacobian()` function, the matrix in (3.16) is computed. This matrix contains the Jacobian matrix with respect to velocity and position, which are computed as the follows. From the spring force equation given in (3.1), a linearization is calculated as:

$$\mathbf{f}_s \approx \mathbf{f}_{s_0} + \frac{\partial \mathbf{f}_s}{\partial \mathbf{p}} \triangle \mathbf{p} + \frac{\partial \mathbf{f}_s}{\partial \mathbf{q}} \triangle \mathbf{q}, \tag{3.20}$$

where $\mathbf{f}_s$ is a spring force, $\mathbf{p}$ and $\mathbf{q}$ are the position of nodes p and q. This equation

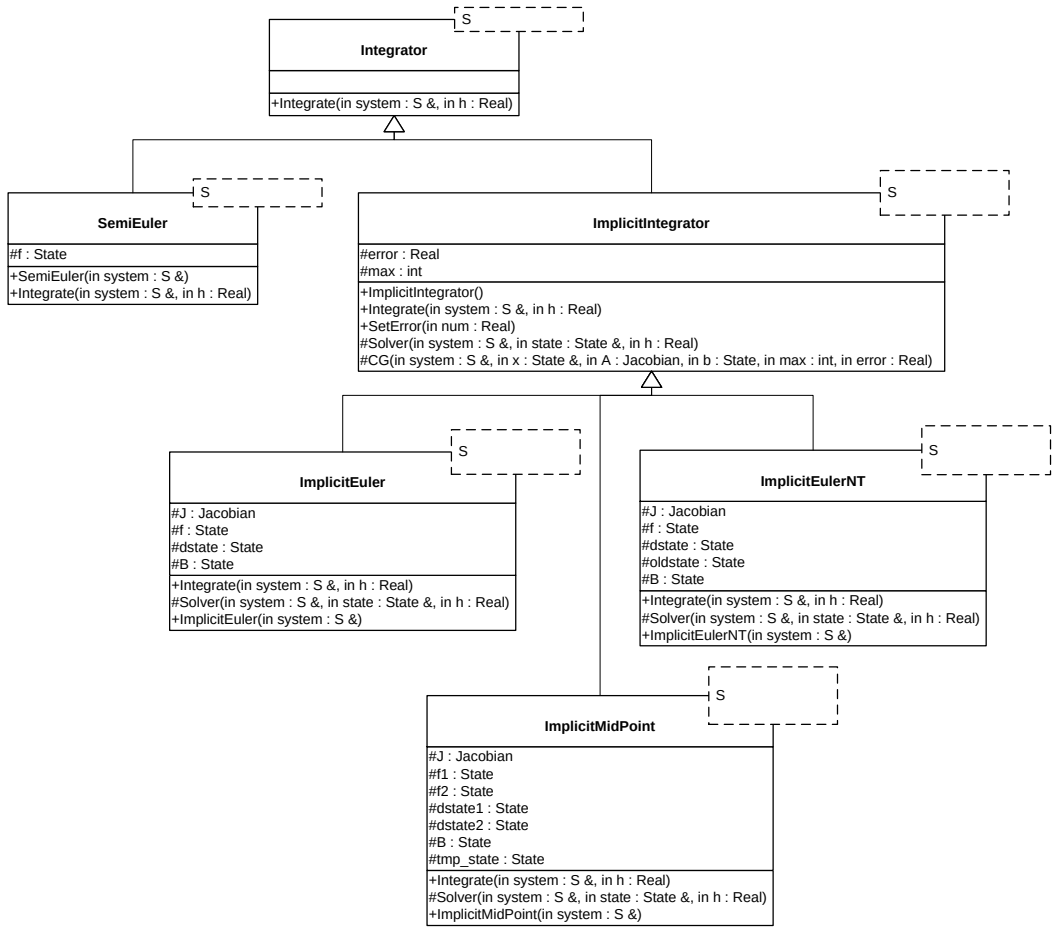


Figure 3.3: New integrator class diagram which were added implicit Euler, implicit Midpoint and semi-explicit Euler methods into the original integrator class.

can be represented in matrix form as:

$$\mathbf{f}_s \approx \mathbf{f}_{s_0} + \begin{bmatrix} \dfrac{\partial \mathbf{f}_s}{\partial \mathbf{p}} & \dfrac{\partial \mathbf{f}_s}{\partial \mathbf{q}} \end{bmatrix} \begin{bmatrix} \triangle \mathbf{p} \\ \triangle \mathbf{q} \end{bmatrix}. \tag{3.21}$$

Consider the derivative $\mathbf{f}_s$ respect to $\mathbf{p}$, which can be represented in each component as:

$$\frac{\partial \mathbf{f}_s}{\partial \mathbf{p}} = \begin{bmatrix} \dfrac{\partial \mathbf{f}_{s1}}{\partial \mathbf{p}_1} & \dfrac{\partial \mathbf{f}_{s1}}{\partial \mathbf{p}_2} & \dfrac{\partial \mathbf{f}_{s1}}{\partial \mathbf{p}_3} \\ \dfrac{\partial \mathbf{f}_{s2}}{\partial \mathbf{p}_1} & \dfrac{\partial \mathbf{f}_{s2}}{\partial \mathbf{p}_2} & \dfrac{\partial \mathbf{f}_{s2}}{\partial \mathbf{p}_3} \\ \dfrac{\partial \mathbf{f}_{s3}}{\partial \mathbf{p}_1} & \dfrac{\partial \mathbf{f}_{s3}}{\partial \mathbf{p}_2} & \dfrac{\partial \mathbf{f}_{s3}}{\partial \mathbf{p}_3} \end{bmatrix}. \tag{3.22}$$

Define

$$A_i^{pq} = -\frac{\partial \mathbf{f}_{si}}{\partial \mathbf{p}_i} = k\left[1 - \frac{L_0}{\|\mathbf{p}-\mathbf{q}\|}\frac{\|\mathbf{p}-\mathbf{q}\|^2 - (q_i - p_i)^2}{\|\mathbf{p}-\mathbf{q}\|^2}\right], \tag{3.23}$$

$$B_{i,j}^{pq} = -\frac{\partial \mathbf{f}_{si}}{\partial \mathbf{p}_j} = k\left[\frac{L_0}{\|\mathbf{p}-\mathbf{q}\|}\frac{(q_i - p_i)(q_j - p_j)}{\|\mathbf{p}-\mathbf{q}\|^2}\right], \tag{3.24}$$

and

$$K^{pq} = \begin{bmatrix} -A_1^{pq} & -B_{1,2}^{pq} & -B_{1,3}^{pq} \\ -B_{2,1}^{pq} & -A_2^{pq} & -B_{2,3}^{pq} \\ -B_{3,1}^{pq} & -B_{3,2}^{pq} & -A_3^{pq} \end{bmatrix}. \tag{3.25}$$

Now, consider the derivative $\mathbf{f}_s$ respect to $\mathbf{q}$, which can also be represented in each component as:

$$\frac{\partial \mathbf{f}_s}{\partial \mathbf{q}} = \begin{bmatrix} \frac{\partial \mathbf{f}_{s1}}{\partial \mathbf{q}_1} & \frac{\partial \mathbf{f}_{s1}}{\partial \mathbf{q}_2} & \frac{\partial \mathbf{f}_{s1}}{\partial \mathbf{q}_3} \\ \frac{\partial \mathbf{f}_{s2}}{\partial \mathbf{q}_1} & \frac{\partial \mathbf{f}_{s2}}{\partial \mathbf{q}_2} & \frac{\partial \mathbf{f}_{s2}}{\partial \mathbf{q}_3} \\ \frac{\partial \mathbf{f}_{s3}}{\partial \mathbf{q}_1} & \frac{\partial \mathbf{f}_{s3}}{\partial \mathbf{q}_2} & \frac{\partial \mathbf{f}_{s3}}{\partial \mathbf{q}_3} \end{bmatrix}. \tag{3.26}$$

Define

$$A_i^{qp} = -\frac{\partial \mathbf{f}_{si}}{\partial \mathbf{q}_i} = -k\left[1 - \frac{L_0}{\|\mathbf{p}-\mathbf{q}\|}\frac{\|\mathbf{p}-\mathbf{q}\|^2 - (q_i - p_i)^2}{\|\mathbf{p}-\mathbf{q}\|^2}\right], \tag{3.27}$$

$$B_{i,j}^{qp} = -\frac{\partial \mathbf{f}_{si}}{\partial \mathbf{q}_j} = -k\left[\frac{L_0}{\|\mathbf{p}-\mathbf{q}\|}\frac{(q_i - p_i)(q_j - p_j)}{\|\mathbf{p}-\mathbf{q}\|^2}\right]. \tag{3.28}$$

We can see that $A_i^{qp} = -A_i^{pq}$ and $B_{i,j}^{qp} = -B_{i,j}^{qp}$. Therefore, $K^{qp} = -K^{pq}$. Then (3.21) becomes

$$\mathbf{f}_s \approx \mathbf{f}_{s_0} + \begin{bmatrix} K^{pq} & -K^{pq} \end{bmatrix} \begin{bmatrix} \triangle \mathbf{p} \\ \triangle \mathbf{q} \end{bmatrix}. \tag{3.29}$$

The damper force in (3.2) can also be linearized similar to the spring force. The linearization of the damper force in the matrix form is given by

$$\mathbf{f}_d \approx \mathbf{f}_{d_0} + \begin{bmatrix} \frac{\partial \mathbf{f}_d}{\partial \mathbf{p}} & \frac{\partial \mathbf{f}_d}{\partial \mathbf{q}} \end{bmatrix} \begin{bmatrix} \triangle \mathbf{p} \\ \triangle \mathbf{q} \end{bmatrix} + \begin{bmatrix} \frac{\partial \mathbf{f}_d}{\partial \mathbf{v}} & \frac{\partial \mathbf{f}_d}{\partial \mathbf{w}} \end{bmatrix} \begin{bmatrix} \triangle \mathbf{v} \\ \triangle \mathbf{w} \end{bmatrix}, \tag{3.30}$$

where $\mathbf{f}_d$ is a damper force, $\mathbf{v}$ and $\mathbf{w}$ are the velocity of nodes p and q. Consider the derivative $\mathbf{f}_d$ respect to $\mathbf{p}$, which can be represented in each component as,

$$\frac{\partial \mathbf{f}_d}{\partial \mathbf{p}} = \begin{bmatrix} \frac{\partial \mathbf{f}_{d1}}{\partial \mathbf{p}_1} & \frac{\partial \mathbf{f}_{d1}}{\partial \mathbf{p}_2} & \frac{\partial \mathbf{f}_{d1}}{\partial \mathbf{p}_3} \\ \frac{\partial \mathbf{f}_{d2}}{\partial \mathbf{p}_1} & \frac{\partial \mathbf{f}_{d2}}{\partial \mathbf{p}_2} & \frac{\partial \mathbf{f}_{d2}}{\partial \mathbf{p}_3} \\ \frac{\partial \mathbf{f}_{d3}}{\partial \mathbf{p}_1} & \frac{\partial \mathbf{f}_{d3}}{\partial \mathbf{p}_2} & \frac{\partial \mathbf{f}_{d3}}{\partial \mathbf{p}_3} \end{bmatrix}. \tag{3.31}$$

Define

$$\begin{aligned} C_i^{pq} &= -\frac{\partial \mathbf{f}_{di}}{\partial \mathbf{p}_i} \\ &= -b\left[(w_i - v_i)\frac{(q_i - p_i)}{\|\mathbf{p}-\mathbf{q}\|^2} + ((\mathbf{w}-\mathbf{v})\cdot(\mathbf{q}-\mathbf{p})\frac{\|\mathbf{p}-\mathbf{q}\|^2 + 2(q_i-p_i)^2}{\|\mathbf{p}-\mathbf{q}\|^4}\right], \end{aligned} \tag{3.32}$$

$$\begin{aligned} D_{i,j}^{pq} &= -\frac{\partial \mathbf{f}_{di}}{\partial \mathbf{p}_j} \\ &= -b\left[(w_j - v_j)\frac{(q_i - p_i)}{\|\mathbf{p}-\mathbf{q}\|^2} + ((\mathbf{w}-\mathbf{v})\cdot(\mathbf{q}-\mathbf{p}))\frac{2(q_i-p_i)(q_j-p_j)}{\|\mathbf{p}-\mathbf{q}\|^4}\right], \end{aligned} \tag{3.33}$$

and

$$V^{pq} = \begin{bmatrix} -C_1^{pq} & -D_{1,2}^{pq} & -D_{1,3}^{pq} \\ -D_{2,1}^{pq} & -C_2^{pq} & -D_{2,3}^{pq} \\ -D_{3,1}^{pq} & -D_{3,2}^{pq} & -C_3^{pq} \end{bmatrix}. \tag{3.34}$$

Now, consider the derivative $\mathbf{f}_d$ respect to $\mathbf{v}$, which can be represented in each component as:

$$\frac{\partial \mathbf{f}_d}{\partial \mathbf{v}} = \begin{bmatrix} \frac{\partial \mathbf{f}_{d1}}{\partial \mathbf{v}_1} & \frac{\partial \mathbf{f}_{d1}}{\partial \mathbf{v}_2} & \frac{\partial \mathbf{f}_{d1}}{\partial \mathbf{v}_3} \\ \frac{\partial \mathbf{f}_{d2}}{\partial \mathbf{v}_1} & \frac{\partial \mathbf{f}_{d2}}{\partial \mathbf{v}_2} & \frac{\partial \mathbf{f}_{d2}}{\partial \mathbf{v}_3} \\ \frac{\partial \mathbf{f}_{d3}}{\partial \mathbf{v}_1} & \frac{\partial \mathbf{f}_{d3}}{\partial \mathbf{v}_2} & \frac{\partial \mathbf{f}_{d3}}{\partial \mathbf{v}_3} \end{bmatrix}. \tag{3.35}$$

Define,

$$E_i^{pq} = -\frac{\partial \mathbf{f}_{di}}{\partial \mathbf{v}_i} = -b\frac{(q_i - p_i)^2}{\|\mathbf{p}-\mathbf{q}\|^2}, \tag{3.36}$$

$$F_{i,j}^{pq} = -\frac{\partial \mathbf{f}_{di}}{\partial \mathbf{v}_j} = -b\frac{(q_i - p_i)(q_j - p_j)}{\|\mathbf{p}-\mathbf{q}\|^2}, \tag{3.37}$$

and

$$B^{pq} = \begin{bmatrix} -E_1^{pq} & -F_{1,2}^{pq} & -F_{1,3}^{pq} \\ -F_{2,1}^{pq} & -E_2^{pq} & -F_{2,3}^{pq} \\ -F_{3,1}^{pq} & -F_{3,2}^{pq} & -E_3^{pq} \end{bmatrix}. \tag{3.38}$$

The derivative $\mathbf{f}_d$ respect to $\mathbf{q}$ and $\mathbf{w}$ are constructed in the same way as the derivative $\mathbf{f}_d$ with respect to $\mathbf{p}$ and $\mathbf{v}$, respectively. But, the results have the opposite sign. Finally, (3.30) becomes

$$\mathbf{f}_d \approx \mathbf{f}_{d_0} + \begin{bmatrix} K^{pq} + V^{pq} & -K^{pq} - V^{pq} \end{bmatrix} \begin{bmatrix} \triangle \mathbf{p} \\ \triangle \mathbf{q} \end{bmatrix} + \begin{bmatrix} B^{pq} & -B^{pq} \end{bmatrix} \begin{bmatrix} \triangle \mathbf{v} \\ \triangle \mathbf{w} \end{bmatrix}. \tag{3.39}$$

We calculate the matrix $K^{pq}+V^{pq}$ and B^{pq} for all pairs of mass nodes connected with springs, and then fill in the Jacobian matrix with respect to position and velocity, respectively. And finally, the Jacobian matrix is used to fill the matrix in (3.16). Each of the $K^{pq} + V^{pq}$ and B^{pq} matrices calculated is used to fill out four locations according to the symmetry of the lumped element model. For example, when the matrix $K^{pq} + V^{pq}$ is computed, we place it at locations (p,p) and (q,q). And then put the negative of this matrix at locations (p,q) and (q,p). This completes the `IdentityMinusJacobian()` function.

3.6 Experiment and Discussion

The implicit integration algorithms were implemented and integrated into GiPSi. The experiments had been performed on the Microsoft Windows XP™ 32-bit based workstation with the Intel Pentium 4 2.53 GHz, 512 MB of RAM and a PCI Express RADEON 9500 Pro Graphics Card with 128 MB of memory. The experiments compared the maximum time step and total time used of the numerical integration algorithms that still maintained the stability of simulation by varying the Young's modulus and damping coefficient. The experiments objective was finding the best numerical integration algorithm and time step used for specific Young's modulus and damping coefficient values of deformable object.

The experiments to compare the numerical integration methods were conducted in two- and three-dimensional lumped element models as shown in Figure 3.5. The procedure to find the maximum simulation time step for specific numerical integra-

tion method, Young's modulus and damping coefficient was shown in Figure 3.4. The lumped element model was loaded with specific numerical integration method, Young's modulus and damping coefficient. The procedure started with simulation time step; h of 1, and the system was simulated from $t = 0$ to $t = 10$ second. If the system was stable, the new simulation time step was set to double value of current simulation time step. Otherwise, the new simulation time step was set to half value of current simulation time step. Then, the system was simulated again with the new simulation time step. If the simulation time step was less than 0.1 and the total energy was stable, then recorded that simulation time step as the maximum time step of stable system. The system stability was calculated from the total energy of lumped element model which included the kinetic and potential energies. If the system was stable, the total energy should be in steady state and total energy should converge and trend to be zero. Therefore, the system stability was determined by comparing the previous total energy and the current total energy with ratio of current and previous total energies was less than the specific error from the script file provided by the user. The system stability was also determined by comparing the initial total energy and the current total energy.

The experiment for two-dimensional lumped element model was conducted with a deformable object which had dimension of 5.0x5.0x0.1 cm^3 with a discretization of 5x5 elements. The model had the density of 1.1415 $g{\cdot}cm^{-3}$, the Young's modulus from 1 to 1,000 $g{\cdot}cm^{-1}{\cdot}s^{-2}$, the damping coefficient from 0.1 to 1.0. The simulation model was fixed at one side and a tension force was applied the opposite side starting at initial time ($t = 0$). For each stiffness and damping coefficient values, the maximum simulation time step that maintained numerical stability, time used per simulation time step and total running time were collected for 10 seconds of simulation time. The goal was to find the optimal simulation time step at each value of Young's modulus and damping coefficient. The experimental results of two-dimensional LEM were

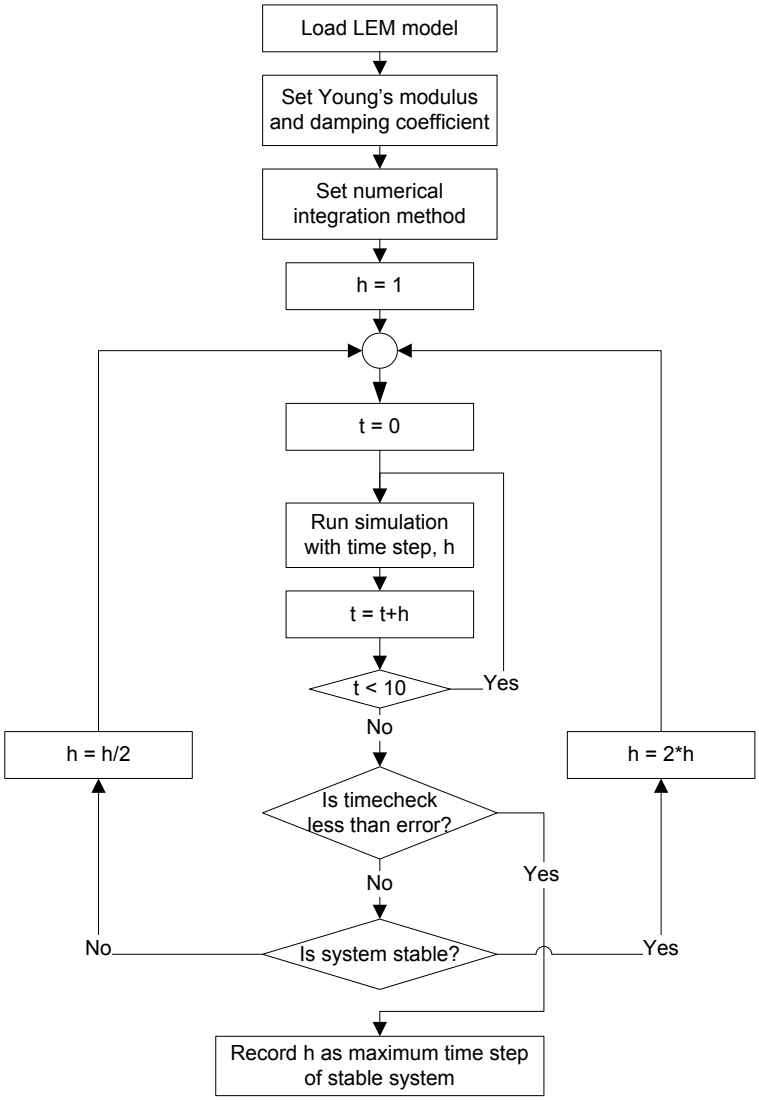
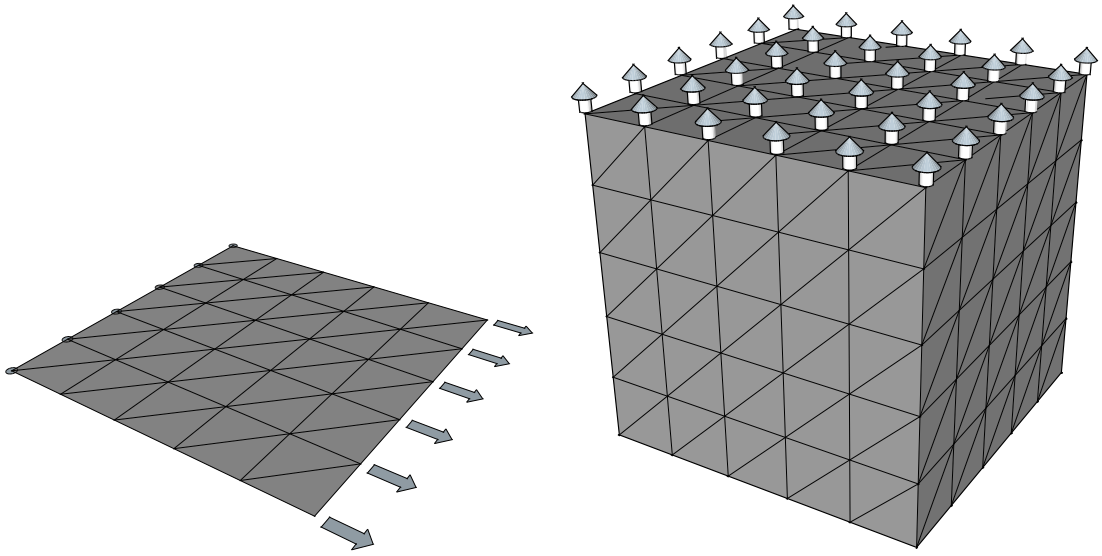


Figure 3.4: Flow chart of the procedure to find the maximum simulation time step for specific integration method, Young's modulus and damping coefficient.

shown in Figure 3.6.

In three-dimensional lumped element model, the experiment was conducted with a deformable object which had dimension of 5.0x5.0x5.0 cm^3 with a discretization of 5x5x5 elements. The physical properties of lumped element model in three dimensions were the same configuration as in two dimensions. The model was fixed at the bottom surface and a normal tension force was applied at the top surface starting at initial time ($t = 0$). The experimental results of three-dimensional LEM were shown in



(a) (b)

Figure 3.5: Experimental configurations of lumped element model: (a) In two dimensions; and (b) In three dimensions.

Figure 3.7.

The trend of the experimental results was the same for both two- and three-dimensional LEM experiments. The total time used in simulation for the explicit integration methods ordering from minimum to maximum were Euler, semi-explicit Euler, Midpoint, Heun, and forth order of Runge-Kutta corresponding to the order of simulation error in each numerical integration method. The total time used in simulation for the implicit integration methods ordering from minimum to maximum were implicit Euler, implicit Euler with Newton's method, and implicit Midpoint. Because the implicit Euler solved a linear system of equations in one conjugate gradient step, but the implicit Euler with Newton's method employed several of conjugate gradient steps until the error of solution was less than the error criteria. Whereas the implicit Midpoint solved two systems of equations, the maximum time step of stabilized system in implicit Midpoint method was highest. The total time used in implicit Midpoint depended on the size of simulation object. If the simulation object had a lot of elements, the size of system of equation increased in polynomial time.

The time to solve the system of equation also increased as well.

It was important to observe the cross-over between the explicit and implicit methods. For example, at damping coefficient 0.1, the intersection was at the Young's modulus value of about 30.0 - 250.0 $g{\cdot}cm^{-1}{\cdot}s^{-2}$ in two dimensions (Figure 3.6e) and about 0.8 - 3.0 $g{\cdot}cm^{-1}{\cdot}s^{-2}$ in three dimensions (Figure 3.7e). These cross-over points indicated the set of parameters where integration algorithm became more efficient compared to the other algorithms. The intersection lines between explicit and implicit moved to the lower Young's modulus values in three-dimensional object model compared to two-dimensional object model because the stiffness in three-dimensional object had the thickness 5.0 cm while the thickness in two-dimensional object was 0.1 cm. The stiffness was calculated from $k = EA/x$. Therefore, the three-dimensional object had higher stiffness than the two-dimensional object for the same Young's modulus value. These results indicated that the explicit method was unsuitable for the stiff equation, as the results shown the lowest step size and the highest total time used to complete simulation. It was also important to note that although the implicit methods were stable for all simulation time steps, but the accuracy still decreased as simulation time step increased because the implicit methods had a strongly damped when the simulation time step was increased.

3.7 Conclusion

In this chapter, the implicit numerical integration algorithms, which were implicit Euler, implicit Euler with Newton's method, and implicit Midpoint, were implemented and integrated in GiPSi framework. The comparisons of the numerical integration algorithms were presented, which were explicit Euler, explicit Midpoint, Heun, forth order of Runge-Kutta, semi-explicit Euler, implicit Euler, implicit Euler with Newton's method, and implicit Midpoint. This study helped a model developer choose

the best numerical integration method with suitable simulation time step for specific simulation object model. The maximum simulation time step and the total time used per simulation time step depended on the physical properties of lumped element model. The higher damping coefficient, the lower maximum simulation time step and the higher total time used. The same situation occurred, when the spring stiffness was increased. The decreasing of simulation time step made the system more stable. The time used per simulation time step in each numerical integration method depended on the complexity of lumped element model and the integration algorithm itself. The Young's modulus of lumped element models used in this study about 1 - 1,000 Pa was significantly lower than the real physical properties of deformable object, such as soft tissue, which had the Young's modulus approximately in order of 10 kPa to 1,000 kPa [62]. The Young's modulus of soft tissues was higher than the Young's modulus used in this study. It meant that if the lumped element model had the exact physical properties of soft tissue, the implicit integration algorithms were needed for the stability of the system. There are also other popular numerical integration algorithms, such as the Newmark and Simplectic algorithms, which have not been included in this study and may have advantage over the numerical integration algorithms used in this study.

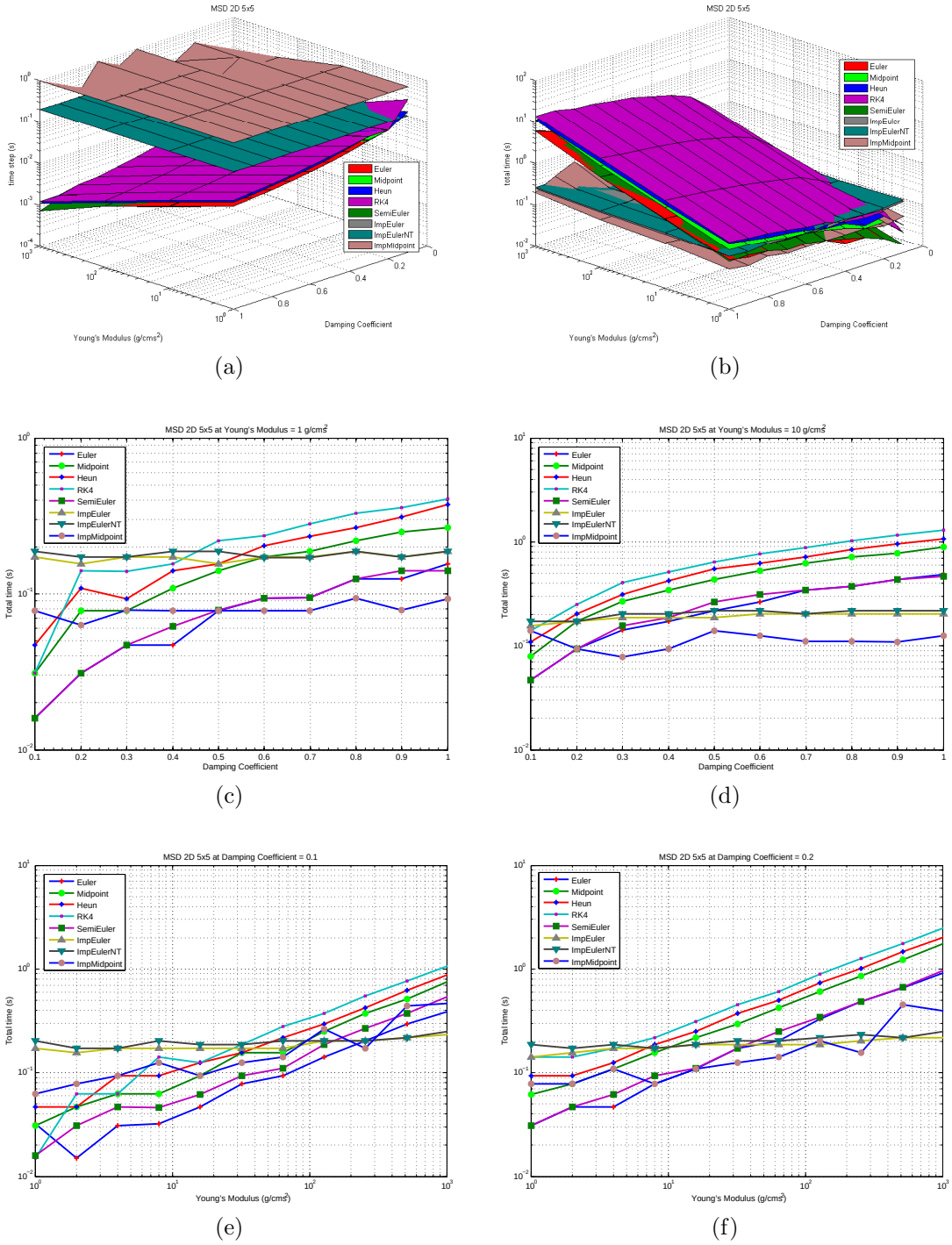


Figure 3.6: The experimental results of two-dimensional LEM: (a) The maximum time step of each method that still maintain numerical stability; (b) The total time used per time step; (c) - (f) The cross-over points between explicit and implicit integration methods.

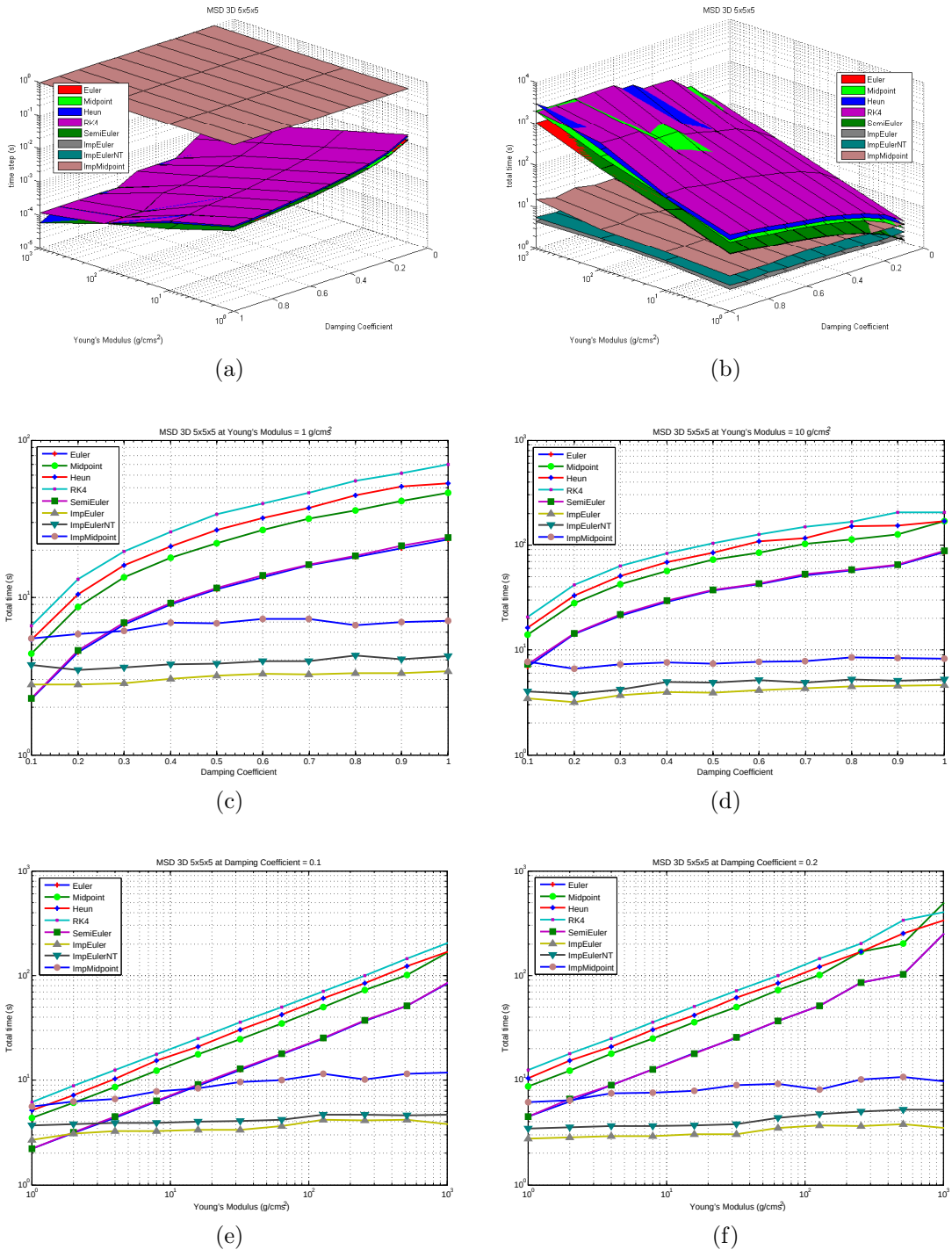


Figure 3.7: The experimental results of three-dimensional LEM: (a) The maximum time step of each method that still maintain numerical stability; (b) The total time used per time step; (c) - (f) The cross-over points between explicit and implicit integration methods.

Chapter 4

Collision Detection and Response for Deformable Object Models

4.1 Introduction

A surgical simulator deals with simulation objects which are interacting with each other. Collision detection and collision response play an important role in producing realistic behaviors in interactive simulations when simulation objects collided.

Collision is a configuration of two objects occupying the same space at the same time. Collision detection is the procedure to find the overlapping parts of objects, and collision response is the procedure to resolves the overlapped parts between the objects. In the simulation, the collision detection checks intersections between all pairs of objects at each simulation time step and provides collision information for collision response function to separate collided objects by specified reaction effects to them. In surgery simulation, the most simulation objects are deformable objects, which may change internal structure, in contrast to rigid bodies, which have no change in internal structure. Therefore, collision detection and response algorithms for rigid objects are not fully compatible with simulations involving deformable objects. Rigid

object collision detection algorithms need to be modified with additional geometric refitting at each simulation time step in order to be able to handle object deformations. The collision response algorithm needs to apply the reaction effects locally, on the overlapped parts of the collided objects.

In this chapter, a collision method integrated into GiPSi [8] is presented. As such, the method complies with the requirement of the GiPSi API. Specifically, the GiPSi API requires the collision detection and response to be through simulation object boundaries. The geometric representation of simulation object boundaries in GiPSi is triangular surface. Therefore, the collision detection method needs to be suitable for GiPSi geometric primitive structure. Furthermore, the collision response for the simulation objects also needs to be applied through boundary conditions of the simulation objects.

The proposed collision detection method is a two-phase algorithm which employs a hierarchical axis aligned bounding box algorithm in the board phase, and a triangle-triangle intersection test algorithm in the narrow phase for finding exact overlapped part of collided objects. The proposed collision response method is based on a penetration depth estimation algorithm for separating overlapped parts of collided objects.

The remainder of this chapter is organized as follows: In the next section, the related studies in the literature on collision detection and response methods are discussed (section 4.2). The proposed algorithm including the data structures used and the details of collision detection and response algorithms are presented in section 4.3. Experimental results to verify and validate algorithms are presented in section 4.4, followed by concluding remarks in section 4.5.

4.2 Related Works

A collision contains two different problems: collision detection and collision response. The former is basically related to objects' geometry and how to find overlapping objects in space, whereas the latter is related to the mechanical modeling and how to separate the collided parts of objects. In collision detection the main problem is computation complexity, whereas in collision response the main problem is how to generate and apply the mechanical effects (force, displacement, acceleration, impulse) to simulation objects to produce the realistic result.

Collision detection algorithms for deformable objects were reviewed by Teschner et al.[63]. A naive collision detection algorithm checks all simulation objects in the simulation scene. If the simulation scene has n simulation objects, the collision detection is performed in $\frac{n(n-1)}{2}$ times. The complexity of this situation is in $O(n^2)$. Moreover, the naive collision detection procedure needs to do an intersection test between geometrical primitives of two simulation objects. If each simulation object contains m geometrical primitives, which usually is triangle or rectangle, the time complexity is multiplied with $O(m^2)$. This results in an overall complexity of order $O(n^2m^2)$. Many research groups have proposed collision detection algorithms to reduce this complexity. There are many approaches of collision detection algorithms. Most of them depend on the geometrical attributes. Therefore, collision detection algorithms can be classified into four groups [64]:

(a) In bounding volumes methods, complex objects or object groups are enclosed within simple volumes that can be easily tested for collisions;

(b) In projection methods, possible collisions are evaluated by considering the projections of the scene separately along several axes or surfaces;

(c) In subdivision methods, complex objects decompose into smaller space volumes or object regions based either on the scene space or on the objects to be evalu-

ated through bounding volume technique;

(d) In proximity methods, the scene objects are arranged according to their geometrical neighborhoods and the collision between these objects are detected based on the neighborhood structure.

The collision detection algorithm proposed here is based on the (well-known) hierarchical bounding volume method. In hierarchical bounding volumes, the collision detection complexity is reduced by representing the objects using a hierarchy of simple bounding volumes. The collision detection starts by performing the overlap test at the root of bounding volume tree which represents the whole object. If they overlap, the collision tests are performed at the child nodes recursively until the leaf nodes, which contain a geometrical primitive of the object, are reached. The examples of bounding volume methods are bounding spheres [65], axis aligned bounding boxes (AABBs) [66], oriented bounding boxes (OBBs) [67], and discrete orientation polytopes (k-DOPs) [68]. Example of different types of bounding volumes are shown in Figure 4.1.

The bounding spheres are represented by centers and radii of spheres that bound an entire object. The overlap test of bounding spheres compares the distance between their centers and the sum of their radii. This overlap test is very fast and easy to implement. The AABBs are rectangular bounding boxes that align to the axes of the object's local coordinate system and all boxes in a tree have the same orientation. The AABBs can fit an object more tightly than the bounding spheres. However, a drawback of AABBs is a fitting problem when object rotates after building a bounding box. The OBBs are rectangular bounding boxes at an arbitrary orientation in three-dimensional space. The OBBs are oriented to enclose an object as tightly as possible. The OBBs generally allow geometries to be bounded more tightly with a few numbers of boxes. However, the expense of overlap computations is more costly, when compared with AABBs. The k-DOPs are convex polytope bounding volumes

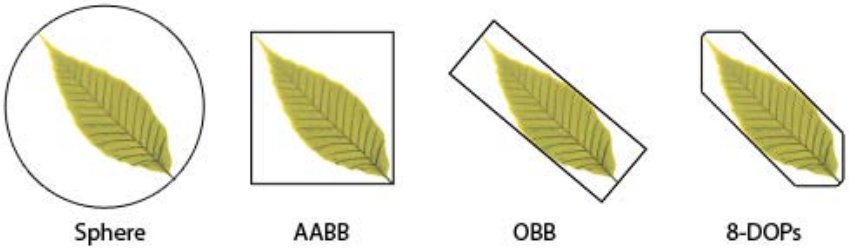


Figure 4.1: Example of bounding volumes from left to right: Sphere, AABB, OBB, and k-DOPs.

whose facets are defined by $\frac{k}{2}$ orientations (where k is even). The AABBs in three dimensions are 6-DOPs. The larger value of k can make the bounding volume close to the object, but it increases the time complexity. In three dimensions, 14-DOPs are defined by 7 axes and use the 6 half spaces of the AABBs facets and 8 additional diagonal half spaces that cut off 8 corners of AABBs. Therefore, the k-DOPs can bound objects tighter than AABBs. Klosowski et al. [68] experimentally found that $k = 18$ yields the best execution times for several different test cases. OOBs provide the best fit for object follow by k-DOPs, AABBs, and bounding spheres. Building an AABBs tree of a given object is faster than building an OBBs tree. Van Den Bergen [66] found that building an OBBs tree takes about three times as much time as building an AABBs tree. The OBBs are represented using 15 scalars (9 scalars for 3x3 matrix representing the orientation, 3 scalars for position, and 3 for extent), whereas the AABBs only requires 6 scalars (3 scalars for position and 3 for extent). Therefore, an AABB tree requires roughly half of the storage required by an OBB tree of the same object. AABBs provide a fast overlap test while OBBs and k-DOPs provide slower overlap test. The comparison of the intersection testing performance between the AABBs and OBBs by using separating axis theorem (SAT) has shown that the OBBs perform faster than the AABBs [66, 69].

There are many collision detection libraries publicly available. Examples include, RAPID (using OBBs), SOLID (using AABBs), and QuickCD (using k-DOPs). Van

Den Berger [69] claims that the AABBs can be used with deformable objects by refitting the AABBs, which is about ten times faster than recalculating the AABBs. For real-time geometric refitting to deformation objects, James and Pai [70] uses a bounding sphere hierarchy called a bounded deformation tree or BD-Tree. This method does not update the bounding spheres from the deformed geometric object, but instead, uses the average motions associated with the displacement fields to update the bounding spheres.

The proposed collision detection methods for use in GiPSi employs hierarchical axis aligned bounding boxes (AABBs) for broad phase and the triangle-triangle intersection test for narrow phase collision detection.

The collision response specifies how the collided objects are separated by applying reaction effects, such as forces, displacements, velocities or impulses. The collision response methods can be classified into four groups:

(a) Analytical methods, use analytical constrains in the equations of motion of colliding objects and produces differential-algebraic equations to find the exact solutions.

(b) Impulse-based methods, use collision impulses computed from the new center of mass and angular velocities for each object.

(c) Penalty force methods, compute response forces based on penetration depths of colliding objects, and then apply those penalty forces back to colliding objects.

(d) Penetration depth methods, compute penetration depths of colliding objects, and then apply those displacements back to colliding objects

Moore and Wilhelms [71] initiated an algorithm to calculate the forces between collided rigid bodies at a single contact point. After that Baraff [72] proposed an analytical method for finding forces between contacting objects based on linear programming techniques, and later they also proposed an analytical method for finding

the contact forces between curved surfaces [73]. Analytical methods are difficult to implement and time consuming as they require solving equations during simulation. Therefore, Mirtich [74] introduced an impulse-based method as the new approach for solving collision response in dynamic simulation. The impulse-based method is simpler and faster than analytical constrains. However, both analytical and impulse-based methods are only work for rigid body objects. In the case of deformable objects, the collision response occurs at the area of collided parts, which are suitable for penalty force and penetration depth methods. Moore and Wilhelms [71] introduced a penalty force method based on spring forces to prevent objects in resting contact from penetrating. The penalty force method is easy to implement and computationally efficient. However, the penalty force method approximates the collision and allows some interpenetration between objects. This also yields stiff equations, which can result in numerical stability problems and requires small time step. Penetration depth method works well on deformable object by only applying the displacement on the collied parts which prevents the cause of stiff equation from spring force approximation in the penalty force method.

The collision response algorithm used in this work employs a penetration depth technique. The proposed method computes the minimal translation displacement of the collided objects to separate them. There are many algorithms to calculate the penetration depth of collided objects. An exact penetration depth algorithm is based on Minkowski sums [75], whereas an approximate penetration depth algorithm is based on the Gilbert-Johnson-Keerthi (GJK) algorithm [76]. Kim et al. [77] proposed the fast penetration depth algorithm using object-space and image-space techniques. These approaches did not address the inconsistency problem in large penetration depth (Figure 4.2). In the left of Figure 4.2, the non-plausible penetration depth occurs when the approximation strictly computing a minimal distance from object surface, whereas our proposed method computes consistent penetration

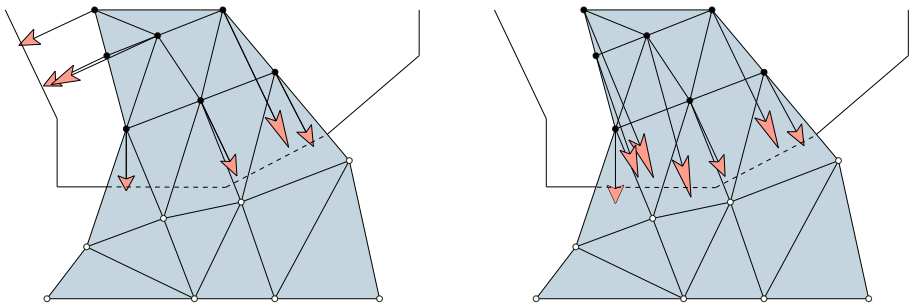


Figure 4.2: Problem of non-plausible penetration depth estimation by using minimal penetration distance (left figure), by using consistent penetration distance (right figure).

distance as in the right of Figure 4.2. Heidelberger at el. [78] proposed a consistent penetration depth estimation algorithm for volumetric models to eliminate the inconsistency artifact inherent in existing penetration depth algorithm.

The collision response algorithm developed in this study is based on the consistent penetration depth estimation originated by Heidelberger at el. [78]. The algorithm of Heidelberger at el. is designed to work with tetrahedral meshes and a collision detection based on hash function [79]. In the present study, this algorithm has been extended to work with the triangle surface geometry representation of objects, and linked list data structure type of collision information from collision detection algorithms as required by GiPSi API.

4.3 Algorithm

This section provides an overview of the proposed algorithm followed by a detailed of each step. Figure 4.3 shows the flow diagram of the proposed algorithm. First, the algorithm checks the value of the `changing` flag of simulation object geometry to determine the bounding box of the objects should be updated or not. The adjustment of the bounding box could be necessary as a deformable object can change internal structure at every simulation time step when an external force is applied. The collision

detection step is then divided into two phases: broad phase and narrow phase. The broad phase is executed to find the collided bounding volumes by using an overlap test. The narrow phase uses a primitive intersection test for each pair of collided bounding volumes from the board phase. After that the collision information result feeds into the collision response algorithm. The collision response algorithm is based on a penetration depth approximation method, which contains four steps. In step 1, collision vertex identification identifies all colliding vertices that are adjacent to one or more non-colliding vertices as border vertices. The intersection points and surface normal vectors are computed for border vertices. In step 2, the penetration depth approximation calculates the penetration depth and direction for each border vertex based on the adjacent intersection points and the surface normal vectors from step 1. In step 3, the penetration depth propagation propagates the penetration depth and direction from border vertices to all colliding vertices that are not border vertices. In step 4, the artificial border investigation finds vertices in intersecting faces that do not contain any border vertices. This step is a correction step for the case that a face is identified as intersecting with the other object, but none of its vertices are identified as border vertices. Following this the penetration depth and direction for those vertices are calculated until no more colliding vertices remain. Finally, the geometries of the collided objects are updated using a penetration depth-based collision response method and the `changing` flag is set to false.

4.3.1 Data Structures

The data structures used in collision algorithm is shown in Figure 4.4. Collision information is stored in a list of `CollisionInfo` structures. The collision information list contains an entry of `CollisionInfo` for each pair of colliding objects. The `CollisionInfo` stores, for a pair of boundary of colliding objects, a pair of face collision information arrays, a pair of vertex collision information lists. Moreover, the

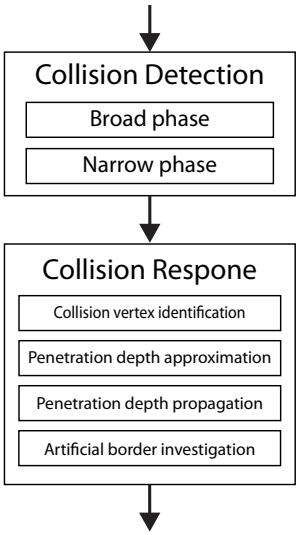


Figure 4.3: Flow diagram of collision detection and response.

sets of current and next face to be processed for each of the colliding objects are also stored in the collision information structure to keep track of the faces, which have intersect state, are currently being processed, and the faces that will be processed in the next iteration. The face collision information contains face state, vertex indices, and array of collided faces of other object. There are four face states, which are: '`unknown`', '`intersect`', '`inside`', and '`outside`'. The face state is initially '`unknown`'. If a face is determined to intersect other faces during narrow phase of the collision detection, then the state of the face is set '`intersect`'. A face is assigned the '`inside`' state if it is determined to be inside the colliding object. And, a face is assigned the '`outside`' state if it is determined to be outside the colliding objects. The vertex collision information contains vertex position, vertex index, vertex state, penetration depth, penetration direction, penetration force, array of normal vectors, and array of intersection points. There are five vertex states, which are: '`unknown`', '`non-colliding`', '`border`', '`processing`', and '`artificial border`'. The '`unknown`' state is an initial vertex state. A vertex is assigned the '`non-colliding`' state if it is outside other object. A vertex is assigned the '`border`' state if it is adjacent to one or more '`non-colliding`' vertices. A vertex is assigned the '`processing`' state if it is

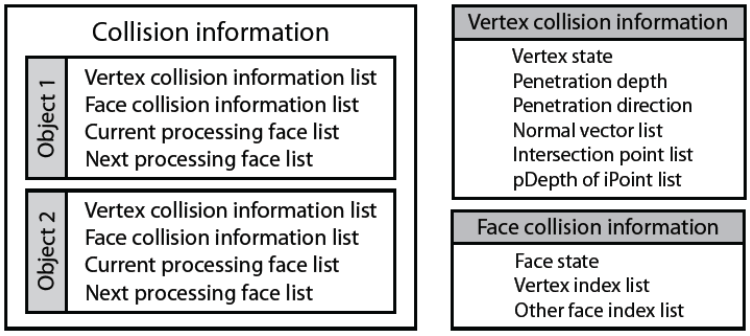


Figure 4.4: Data structure in collision detection and response algorithm.

being propagated. A vertex is assigned the '`artificial border`' state if it belongs to the face with '`intersect`' state and no other vertices with '`border`' state exist in that face.

4.3.2 Collision Detection

The basic collision detection is an algorithm to test intersections for all possible pairs of objects in a virtual environment. The collision detection algorithm implemented has two phases: broad phase and narrow phase. The broad phase is an overlap test between two bounding volume hierarchies. If an overlap is found, the narrow phase will be performed by using a primitive intersection test algorithm to retrieve the exact collision information.

Broad Phase

The broad phase is a bounding volume overlap test, which uses the bounding volume hierarchy to reduce the time complexity by testing collisions only for the pair of intersected bounding volumes. The bounding volume hierarchy is a model partitioning which subdivides an object into geometrically coherent subsets and computes a tight-fitting bounding volume for each subset of object. When the intersection tests are performed on the bounding volumes, subsets of objects can be quickly excluded from intersection testing depending on whether their bounding volumes overlap. A

bounding volume of a model is a primitive shape that encloses the model such that it fits the model as tightly as possible, it is cheap to test overlap between bounding boxes, it requires a small amount of memory storage, and it is fast to calculate the bounding box of a given model. The broad phase is based on the OPCODE [80] collision detection library. The OPCODE is a small collision detection library which is fast and memory efficient. The OPCODE library uses hierarchical axis aligned bounding boxes. A bounding volume hierarchy is calculated for each object by using binary tree, top-down construction approach. In the resulting bounding volume hierarchy, the bounding volumes may overlap but each leaf node contains primitive data (face list) with no duplication of primitives in the leaf nodes. The procedure for the intersection test in the broad phase is as follows: Bounding volume overlap test is performed for each pair of objects in the collision object pool using the binary trees that represent their bounding volume hierarchies. If the bounding volumes of the nodes do not intersect, then the next pair of simulation objects are tested. If both nodes are leaves, then primitive intersection test is executed and the result is returned. If one of the nodes is a leaf and the other is an internal node, then leaf node is tested for primitive intersection with each of the children of the internal node. If both nodes are internal nodes, then the node with smaller volume is tested for primitive intersection with the children of the node with the larger volume.

Narrow Phase

The narrow phase is a primitive intersection test, which uses a triangle-triangle intersection test algorithm developed by Möller [81]. The algorithm is adapted to provide more information on the collision results which is necessary for the collision response algorithm used in GiPSi. Our modified algorithm adds three more vertex information after testing triangle-triangle intersections. The first additional collision information is `codeT1` and `codeT2` to classify which vertices are inside or outside referenced by the

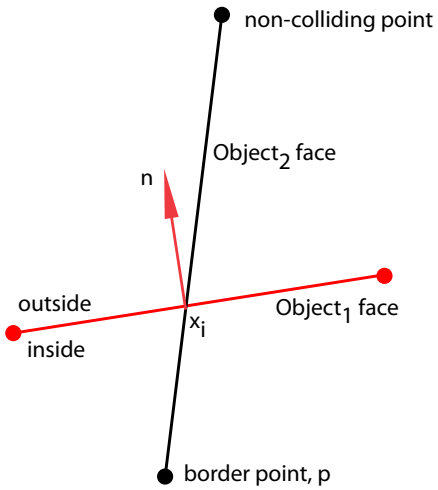


Figure 4.5: Triangle-triangle intersection test with vertex information `codeT1` and `codeT2` to classify inside or outside vertex referenced by the triangle plane of other colliding object.

triangle plane of other colliding object, as shown in Figure 4.5. The output represents the vertices in triangle in binary format, $u_2u_1u_0$. Each bit has a value of 1 for inside other object and a value if 0 for outside other object, where the normal vector of each face points the outward direction of an object.

The second additional collision information is `collidedCode` to classify where intersection points $\mathbf{x}_1, \mathbf{x}_2$ are located. Consider the example shown in the left side of Figure 4.6, when two triangles, $T_1(\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2)$ and $T_2(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$, are intersecting at points $\mathbf{x}_1, \mathbf{x}_2$. The intersection line $\overline{\mathbf{x}_1\mathbf{x}_2}$ is extended and intersects with the edges of each triangle named as $\mathbf{e}_{T_11}, \mathbf{e}_{T_12}$ for triangle T_1, and $\mathbf{e}_{T_21}, \mathbf{e}_{T_22}$ for triangle T_2. The intersection line between two triangles can be classified into 9 cases, shown in the right side of Figure 4.6, based on where the end points of the intersection line $\overline{\mathbf{x}_1\mathbf{x}_2}$, the end points of the line $\overline{\mathbf{e}_{T_11}\mathbf{e}_{T_12}}$ of triangle T_1, and the end points of the line $\overline{\mathbf{e}_{T_21}\mathbf{e}_{T_22}}$ of triangle T_2 are located. The points, $\mathbf{x}_1, \mathbf{x}_2, \mathbf{e}_{T_11}, \mathbf{e}_{T_12}, \mathbf{e}_{T_21}, \mathbf{e}_{T_22}$ are on the extended intersection line. Consider the location of the intersection point $\mathbf{x}_1$. The first three cases are the intersection points $\mathbf{x}_1$ are equal to $\mathbf{e}_{T_11}$ and less than $\mathbf{e}_{T_21}$. Another three cases are the intersection points $\mathbf{x}_1$ are equal to $\mathbf{e}_{T_21}$ and greater than

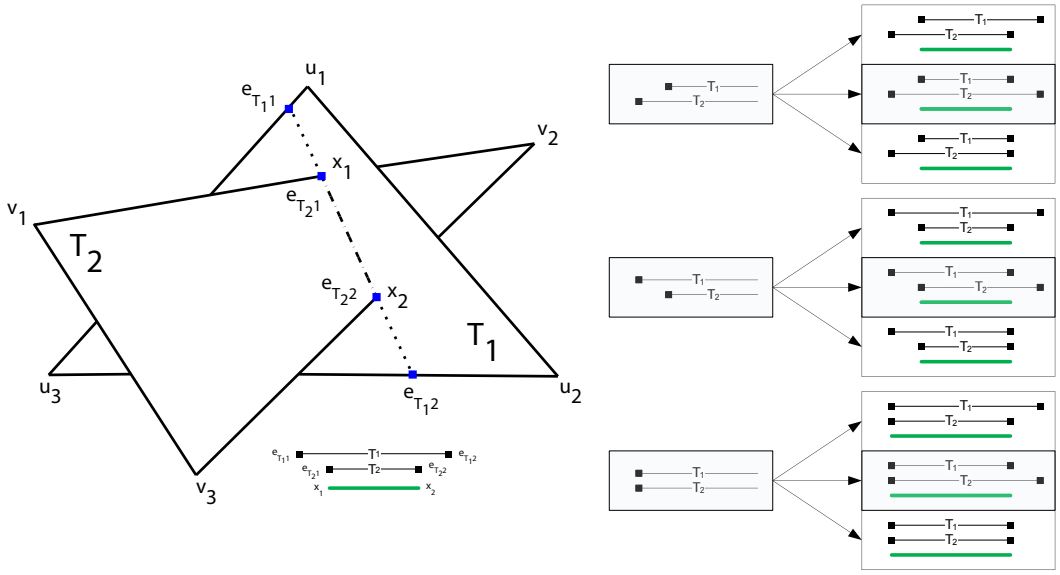


Figure 4.6: Triangle-triangle intersection test with collision information. Example of triangle-triangle intersection (left figure), and 9 cases of `collidedCode` (right figure)

$\mathbf{e}_{T_1 1}$. The last three cases are the intersection points $\mathbf{x}_1$ are equal to $\mathbf{e}_{T_1 1}$ and $\mathbf{e}_{T_2 1}$. Each of three cases has sub three cases by considering the intersection point $\mathbf{x}_2$. First, the intersection point $\mathbf{x}_2$ is equal to $\mathbf{e}_{T_2 2}$ and less than $\mathbf{e}_{T_1 2}$. Second, the intersection point $\mathbf{x}_2$ is equal to $\mathbf{e}_{T_1 2}$ and less than $\mathbf{e}_{T_2 2}$. Third, the intersection point $\mathbf{x}_2$ is equal to $\mathbf{e}_{T_1 2}$ and $\mathbf{e}_{T_2 2}$. The states of the collided faces are set to the '`intersect`'.

The last additional collision information is `codeTT1` and `codeTT2` to classify which vertices of triangle are neither '`border`' nor '`non-colliding`' vertices in intersecting triangles T_1 and T_2. The vertices are identified by finding the vertices that are belong to the edges of triangle and these edges do not intersect with intersection line $\overline{\mathbf{x}_1\mathbf{x}_2}$.

At the end of collision detection algorithm, the collision detection provides collision information for collision response algorithm, which consists of the intersection line defined by the points $\mathbf{x}_1$ and $\mathbf{x}_2$, '`codeT1`', '`codeT2`', '`collidedCode`', '`codeTT1`', '`codeTT2`', the vertices of each colliding face, normal vectors of the colliding faces, face states, and vertex states.

4.3.3 Collision Response

After collision detection is performed, the collision information list is sent to the collision response function. The collision response contains four steps: collision vertex identification, penetration depth approximation, penetration depth propagation, and artificial border investigation. After all four steps are executed, the approximate penetration depth of each vertex in the list of vertex collision information structure, `VertexCollisionInfo`, are applied back to the simulation object boundaries. The details of each step are explained below.

Collision Vertex Identification

The collision information structure for each pair of colliding objects consists of the list of face and vertex collision information structures of both objects. At the end of narrow phase collision detection, the vertices of the two colliding object are classified into '`unknown`', '`non-colliding`' or '`border`' states in the vertex collision information structure. The initial state of each vertex is '`unknown`'. The intersection points of colliding objects, $\mathbf{x}_1$ and $\mathbf{x}_2$, are located on the surface of colliding faces, T_1 and T_2. The vertices of these faces are identified as '`border`' or '`non-colliding`'. The vertices which are inside the other colliding object are identified as '`border`' vertices $\mathbf{p}$. Other vertices are outside the other colliding object and are identified as '`non-colliding`' vertices. The inside/outside test is performed by the checking the dot product between $\mathbf{p} - \mathbf{x}_i$ and the unit normal vector of other colliding object face, $\hat{\mathbf{n}}$. It is assumed that the face unit normal vector $\hat{\mathbf{n}}$ points outward. Then, a vertex $\mathbf{p}$ is inside when $\hat{\mathbf{n}} \cdot (\mathbf{p} - \mathbf{x}_i) < 0$. At this step, the list of vertex collision information structure is filled with the vertices from the collided faces.

Penetration Depth Approximation

The penetration depths and directions are calculated for all vertices which are marked as '`border`' in the list of vertex collision information structure. The weight function $\omega(\mathbf{x}_i, \mathbf{p})$, the penetration depth $d(\mathbf{p})$, the penetration direction $\mathbf{r}(\mathbf{p})$, and the normalized penetration direction $\hat{\mathbf{r}}(\mathbf{p})$ are calculated as the follows:

$$\omega(\mathbf{x}_i, \mathbf{p}) = \frac{1}{\|\mathbf{x}_i - \mathbf{p}\|^2}, \tag{4.1}$$

$$d(\mathbf{p}) = \frac{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p}) \cdot (\mathbf{x}_i - \mathbf{p}) \cdot \mathbf{n}_i}{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p})}, \tag{4.2}$$

$$\mathbf{r}(\mathbf{p}) = \frac{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p}) \cdot \mathbf{n}_i}{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p})}, \tag{4.3}$$

$$\hat{\mathbf{r}}(\mathbf{p}) = \frac{\mathbf{r}(\mathbf{p})}{\|\mathbf{r}(\mathbf{p})\|}, \tag{4.4}$$

where $\mathbf{x}_i$ is the list of intersection points of vertex $\mathbf{p}$ in the vertex collision information structure, and $\mathbf{n}_i$ is the list of other object's face normal corresponding to intersection point of vertex $\mathbf{p}$ in the vertex collision information structure.

Penetration Depth Propagation

In this step, the penetration depths and penetration directions are propagated to all the other colliding vertices that are not '`border`' vertices. The idea of the propagation is to avoid non-plausible penetration depth in case of large penetrations as shown in Figure 4.2. The propagation (Figure 4.7) is an iterative process including of two steps. In the first step, the vertices which are marked as '`border`' vertices are set to the '`processing`' state. In the second step, the vertices which neighbor the '`processing`' vertices are set to '`border`' state. The penetration depths and directions of new '`border`' vertices then are calculated based on the information from all adjacent '`processing`' vertices. If there are no more colliding vertices, the iteration is terminated. The vertices that connect to '`border`' vertices are identified as '`processing`' vertices by retrieving vertices from a set of current processing faces.

Then, the faces that are connected to the '`processing`' vertices are identified, then face is set to '`inside`' state, and added to the next processing faces. Following this, the vertices are assigned to '`border`' state if the vertices are not '`processing`' vertex and in the '`inside`' face. Then, the '`processing`' vertices are added to the list of intersection points and the normalized penetration directions are added to the list of face normals.

The calculations of penetration depths and directions of '`border`' vertices in penetration depth propagation step are similar to the calculations of penetration depth and direction of '`border`' vertices. The weight function $\omega(\mathbf{q}_i, \mathbf{p})$ is calculated from the adjacent '`processing`' vertices $\mathbf{q}_j$ of the current '`border`' vertex $\mathbf{p}$,

$$\omega(\mathbf{q}_j, \mathbf{p}) = \frac{1}{\|\mathbf{q}_j - \mathbf{p}\|^2}. \tag{4.5}$$

The penetration depth $d(\mathbf{p})$ of the current '`border`' vertex $\mathbf{p}$ is calculated as,

$$d(\mathbf{p}) = \frac{\sum_{j=1}^{l} \omega(\mathbf{q}_j, \mathbf{p}) \cdot ((\mathbf{q}_j - \mathbf{p}) \cdot \hat{\mathbf{r}}(\mathbf{q}_j) + d(\mathbf{q}_j))}{\sum_{j=1}^{l} \omega(\mathbf{q}_j, \mathbf{p})}, \tag{4.6}$$

where $\hat{\mathbf{r}}(\mathbf{q}_j)$ is the normalized penetration direction of the '`processing`' vertex $\mathbf{q}_j$ and $d(\mathbf{q}_j)$ is the penetration depth of the '`processing`' vertex $\mathbf{q}_j$ and l is the number of the '`processing`' vertices adjacent to the current '`border`' vertex $\mathbf{p}$. The penetration direction $\mathbf{r}(\mathbf{p})$ is calculated as a weighted average of the penetration direction of the '`processing`' vertices adjacent to the current '`border`' vertex, $\mathbf{p}$,

$$\mathbf{r}(\mathbf{p}) = \frac{\sum_{j=1}^{l} \omega(\mathbf{q}_j, \mathbf{p}) \cdot \hat{\mathbf{r}}(\mathbf{q}_j)}{\sum_{j=1}^{l} \omega(\mathbf{q}_j, \mathbf{p})}, \tag{4.7}$$

$$\hat{\mathbf{r}}(p) = \frac{\mathbf{r}(p)}{\|\mathbf{r}(p)\|}, \tag{4.8}$$

where $\hat{\mathbf{r}}(\mathbf{p})$ is the normalized penetration direction.

Artificial Border Investigation

In some cases, it is possible to have faces which are in the '`intersect`' state, but do not have any '`border`' state vertices. These cases are handled separately. Specifically, the

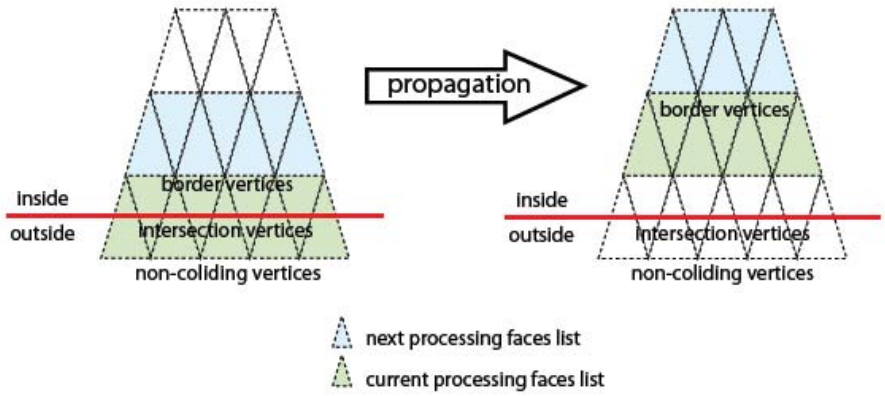


Figure 4.7: Penetration depth propagation step.

vertices of such faces are identified as '`artificial border`' state. An '`artificial border`' vertex $\mathbf{a}$ has no collision information. Therefore, the penetration depths and directions are computed from the other colliding object as follows: The '`border`' vertices in '`intersect`' faces of other colliding object are retrieved and add those vertices to the list of intersection vertices $\mathbf{b}_i$ in $\mathbf{a}$. Then the penetration depth of those vertices also add to the list of the face normal $\mathbf{n}_i$ in $\mathbf{a}$.

In the artificial border investigation, the calculations of penetration depths and directions in '`artificial border`' vertices are similar to the calculations of penetration depths and directions in penetration depth approximation step. For a specific '`artificial border`' vertex $\mathbf{a}$:

$$\omega(\mathbf{b}_i, \mathbf{a}) = \frac{1}{\|\mathbf{b}_i - \mathbf{a}\|^2}, \tag{4.9}$$

$$\mathbf{r}(\mathbf{a}) = \frac{\sum_{i=I} \omega(\mathbf{b}_i, \mathbf{a}) \cdot \mathbf{n}_i}{\sum_{i=I} \omega(\mathbf{b}_i, \mathbf{a})}, \tag{4.10}$$

$$\hat{\mathbf{r}}(\mathbf{a}) = \frac{\mathbf{r}(\mathbf{a})}{\|\mathbf{r}(\mathbf{a})\|}, \tag{4.11}$$

where I are the indices of the intersection vertices, $\mathbf{b}_i$, that $(\mathbf{b}_i - \mathbf{a}) \cdot \mathbf{n}_i > 0$.

After all vertices are identified, the penetration depths and directions are calculated for all collided vertices, and all artificial borders are investigated. The displacements are then applied to the current position of all collided vertices of simulation

object boundaries:

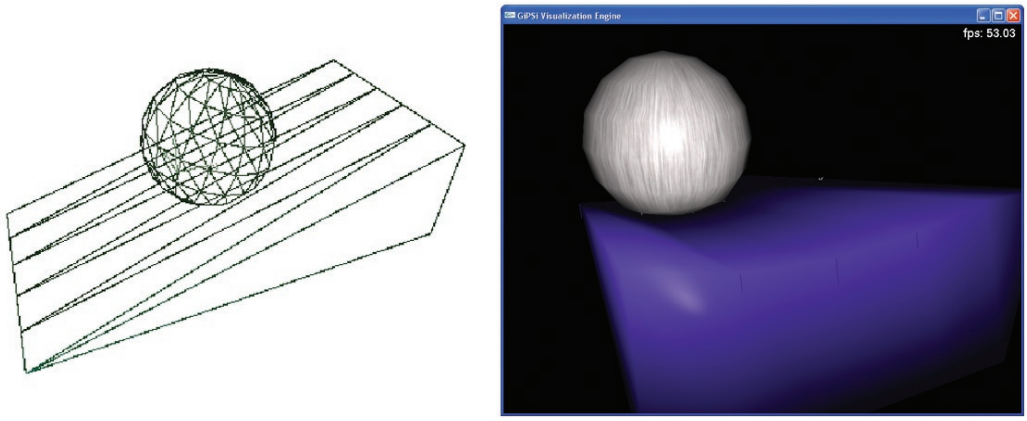
$$\mathbf{p}_{new} = \mathbf{p}_{old} + d(\mathbf{p}) \cdot \hat{\mathbf{r}}(\mathbf{p}) \tag{4.12}$$

4.4 Experiments and Discussion

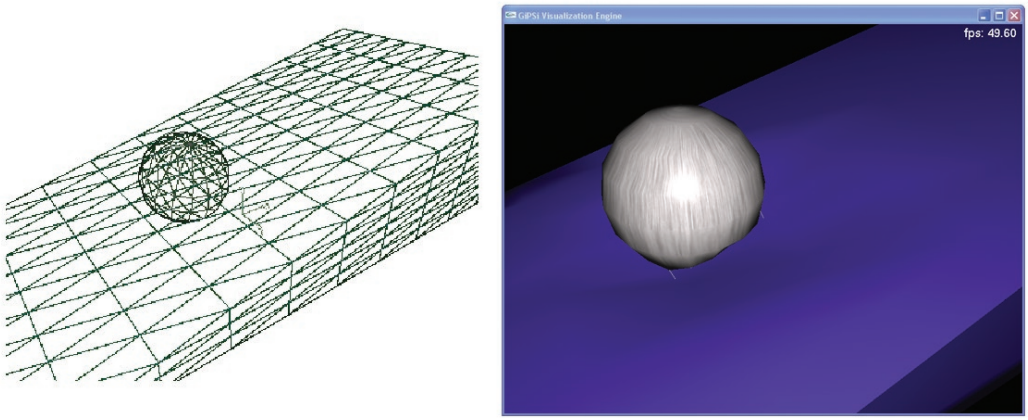
The described collision detection and response algorithms were implemented and integrated into GiPSi framework [8]. Various test cases and experiments were performed to validate the collision detection and response algorithms. All experimental scenarios were performed on the Microsoft Windows XP^{TM} 32-bit based workstation with the Intel Pentium D 2.80 GHz, 1 GB of RAM and a PCI Express NVidia GeForce 6800 Graphics Card. Six static and two dynamic test cases were used. The static test cases included objects that overlapped with each other (Figures 4.8 - 4.13). Two dynamic test cases contained dynamic objects falling down to a membrane (Figures 4.14 - 4.15).

In the static test cases 1a and 1b, a sphere object (114 vertices, 224 faces) was overlapped with a prism object in low resolution (18 vertices, 32 faces), as shown in Figure 4.8a, and in high resolution (431 vertices, 794 faces), as shown in Figure 4.8b. These static cases were used to test the functionality of the collision response steps. The high resolution case verified that the algorithm can handle intersection of two objects which had comparable resolutions (i.e., constructed with primitives of similar sizes), whereas the low resolution case verified that the algorithm can handle a low resolution object intersecting with a high resolution object.

In the static test cases 2a and 2b, a sphere object (114 vertices, 224 faces) was overlapped with a concave object at two separate locations in low resolution (36 vertices, 68 faces), as shown in Figure 4.8a and high resolution (583 vertices, 1,094 faces), as shown in Figure 4.9b. These cases verified that the proposed method can handle the collision response when there were two separated collision areas between two over-



(a)



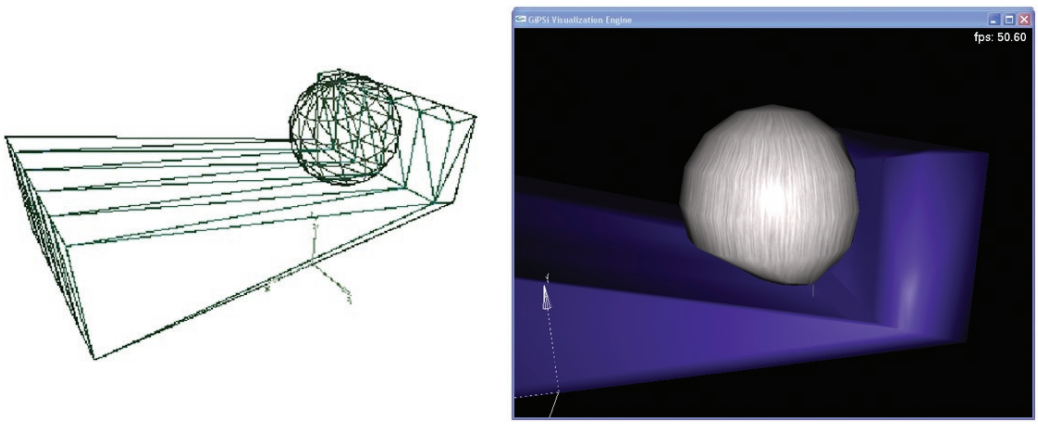
(b)

Figure 4.8: Static test case 1: (a) low resolution; (b) high resolution.

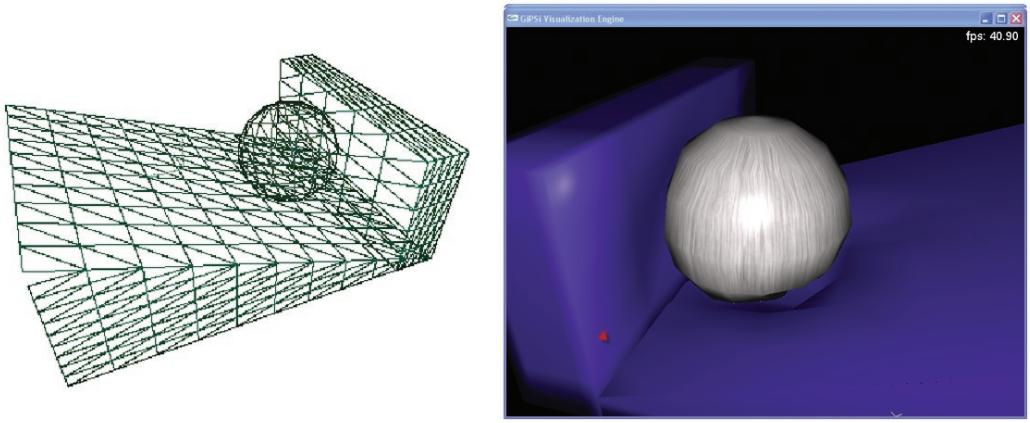
lapping objects. The low and high resolution cases verified that the algorithm can handle colliding objects with similar and different primitive sizes.

In the static test case 3, a sphere object (114 vertices, 224 faces) was overlapped with the corner of box (152 vertices, 300 faces), as shown in Figure 4.10. In this test case, there was one overlapped area but part of the box, which was inside the sphere, had three orthogonal surface normals resulted in three different penetration directions of the box inside the sphere. This case verified that collision configuration resulted in discontinuous penetration depths and directions worked correctly.

In the static test case 4, a sphere object (114 vertices, 224 faces) was overlapped



(a)



(b)

Figure 4.9: Static test case 2: (a) low resolution; (b) high resolution.

with two boxes (each with 152 vertices, 300 faces), as shown in Figure 4.11. In this test case, there were several overlapped areas. This case verified the correctness of the penetration depth and direction calculation when there were multiple collision areas.

In the static test case 5, a small bar (152 vertices, 300 faces) was overlapped with one triangular surface of a large sphere (114 vertices, 224 faces), as shown in Figure 4.12. This case verified the correctness of the artificial border investigation step. When the small bar intersected at the center of triangular surface, the vertices of collided triangle should be identified as artificial border states. Otherwise, the vertices would

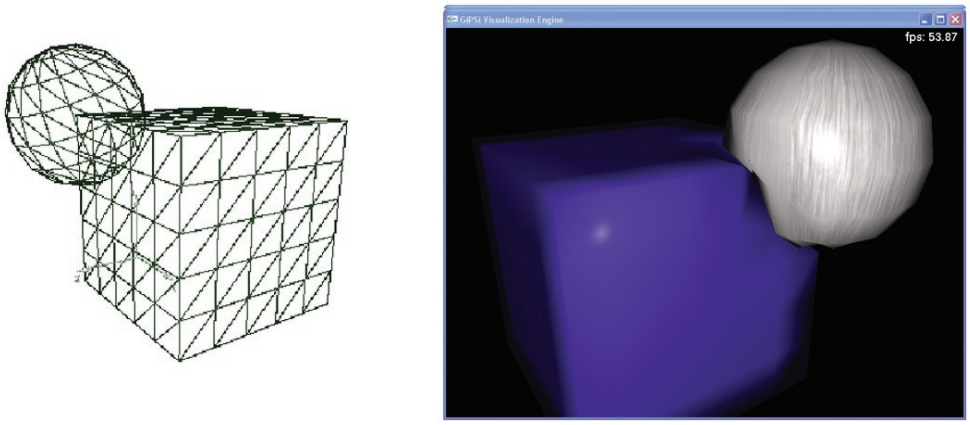


Figure 4.10: Static test case 3.

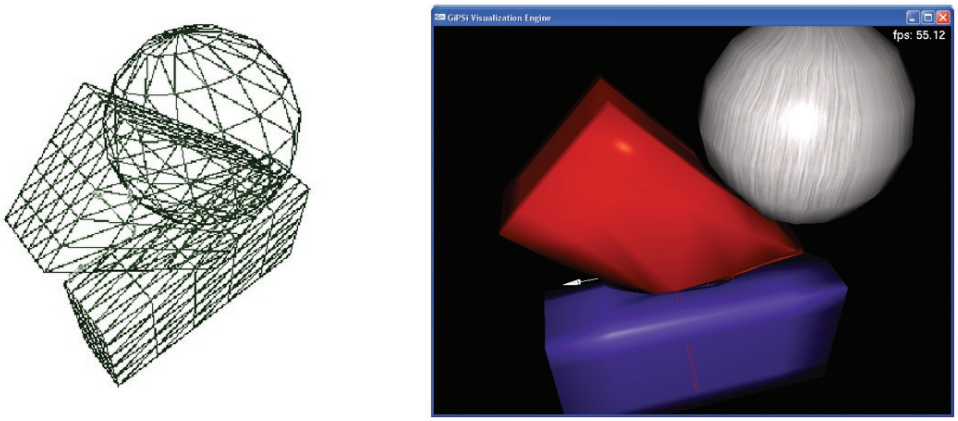


Figure 4.11: Static test case 4.

be identified as non collided vertex state and would not be used in penetration depth calculations.

In the static test case 6, an arbitrary object (686 vertices, 1368 faces) was overlapped with a long bar (152 vertices, 300 faces), as shown in Figure 4.13. There were three overlapped areas between the two objects. This case verified that our proposed algorithm can detect and resolve multiply collided areas correctly.

The experimental results of static test cases were shown in Figures 4.8 to 4.13 and the average computational times of the collision algorithm in each of the static test cases were shown in Table 4.1. The left side of all static test case figures shown

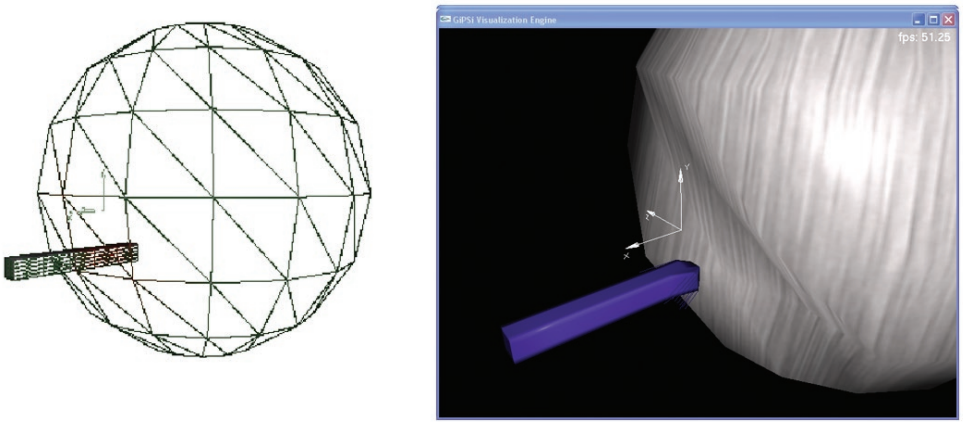


Figure 4.12: Static test case 5.

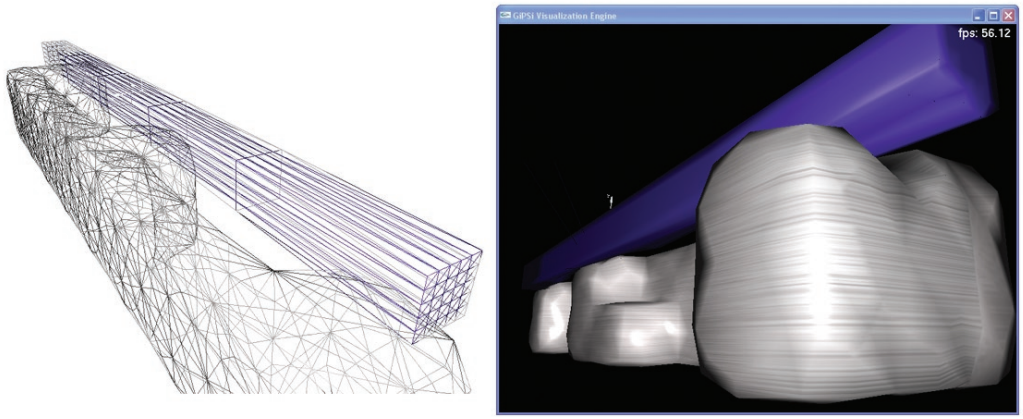


Figure 4.13: Static test case 6.

the initial configuration of objects in wireframe view and the right side of the figures shown the final configuration of objects drawn with texture. The experimental results of all static test cases shown that our proposed algorithm can detect the collisions and correctly separate the overlapping areas. Most of the time in the collision algorithms was spent on the collision detection procedure. The average time used in collision response procedure per vertex was about 0.24 ms. The penetration depth calculation from our proposed algorithm resulted in gaps between collided objects. Those gaps came from the calculation of the penetration depth approximation of collided vertices and distribution of the penetration depths among the collided vertices. It was necessary to adjust the parameter to select how much the displacement or force

of collided vertices should be applied.

Table 4.1: Average computational time of collision algorithm in static test cases.

Static	Collision time (ms)		Total number of
test cases	Detection	Response	collided vertices
1a	15.69	8.05	23
1b	21.18	6.75	27
2a	15.91	8.71	36
2b	25.86	7.04	37
3	8.41	2.99	26
4	23.05	8.86	45
5	16.63	8.88	24
6	25.04	9.44	51

Two dynamic test cases were also performed. The lumped element model (LEM) was used in all simulation objects with simulation time step of 0.001 second and using the 4^{th} order Runge-Kutta numerical integration method. The first dynamic test case simulated five boxes falling down on the membrane under gravitational force (Figure 4.14). Each box was 2x2x2 elements in three dimensions. The boxs had physical models of 27 nodes and 158 springs with elasticity of 78.1 kPa. The surface geometry of each box has 26 vertices and 48 faces. The membrane has a physical model with 155 nodes and 422 springs with elasticity of 21.3 kPa. The spring constants were determined from the procedure provided in Chapter 2. The surface geometry of the membrane had 155 vertices and 268 faces.

The second dynamic test case simulated five boxes falling down to a membrane with a hole under gravitational force (Figure 4.15). The boxes had the same configuration as the boxes in the first dynamic test case with different initial positions. The membrane with hole had a physical model with 88 nodes and 217 springs with elasticity of 21.3 kPa. The spring constants were determined from the procedure provided in Chapter 2. The surface geometry of the membrane had 88 vertices and 129 faces.

The experimental results of dynamic test cases were shown in Figures 4.14 and 4.15. The average computation times in the collision detection and response algorithms in each dynamic test case were shown in the Table 4.2. The left side of all

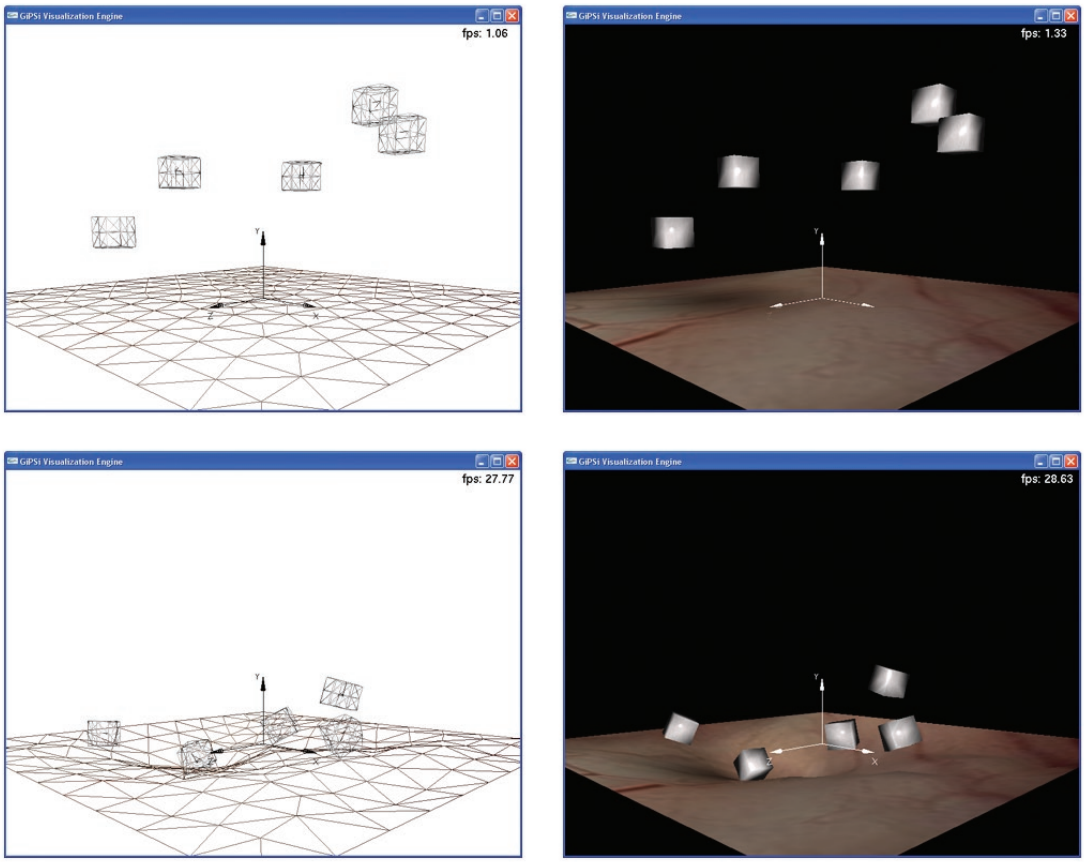


Figure 4.14: The first dynamic test case: Boxes falling down on a membrane. The initial configuration (upper figures), and a snapshot configuration (lower figures).

dynamic test case figures shown the simulation objects in wireframe view and the right figures shown the scene rendered with texture. The upper figures shown the initial configuration of simulation objects and the lower figures shown a snapshot configuration of the simulation. The experimental results of all dynamic test cases shown that our proposed algorithm can detect the collisions and separate the overlapping areas in the dynamic situation. However, it was necessary to maintain stability by adjusting the simulation time step and selecting an appropriate numerical integration method that were suitable for the parameters of physical model (LEM) to maintain numerical stability of the simulation. In dynamic scenario, the mapping function between the surface object and the physical model was needed to transfer the positions,

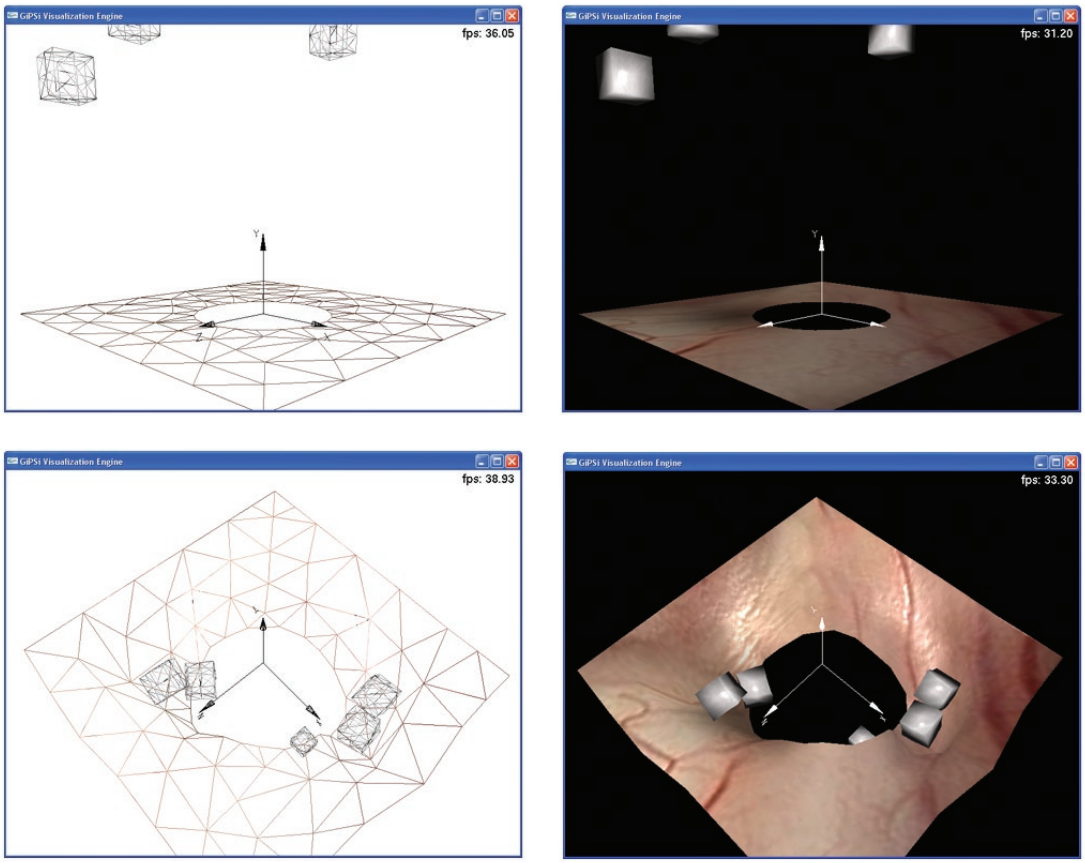


Figure 4.15: The second dynamic test case: Boxes falling down on a membrane with a hole. The initial configuration (upper figures), and a snapshot configuration (lower figure) are shown.

velocities, and forces from the boundary surface to the physical model and vice versa. The experimental results shown that our proposed algorithm was suitable for surface geometric objects. However, it was easy to adapt our proposed algorithm to handle the volume geometric objects by providing connectivity of volume elements.

Table 4.2: Average computational time of collision algorithm in dynamic test cases.

Dynamic	Collision time (ms)	
test cases	Detection	Response
1	8.92	4.34
2	4.30	2.14

4.5 Conclusions

The proposed collision detection and response algorithms were developed and integrated into GiPSi. The collision detection algorithm used a hierarchical axis aligned bounding boxes (AABB) method for board phase and a triangle-triangle intersection test for narrow phase, while the collision response algorithm used the penetration depth approximation method. Scenarios when one object completely crosses or completely penetrates a second object within a single time step cannot be handled by the proposed algorithm. The penetration depth approximation can be improved by adding more step with bisection search for finding the closet position between colliding objects at final step to minimum the gap between colliding objects. Moreover, checking the penetration force before applying back to the colliding object boundaries is helpful to maintain the stability of simulation. The collision algorithm can speed up by using a graphics processing unit (GPU) to test the collision and leave the CPU to do other tasks in the simulation.

Chapter 5

Improvements to the Design of the GiPSi Simulation Framework Architecture

5.1 Introduction

GiPSi (General Interactive Physical Simulation Interface) is an open source/open architecture framework for developing open-level surgical simulations, such as interactive surgical training and planning system. The GiPSi has been initiated by Çavuşoğlu et al.[8]. The main goal of the GiPSi framework is to facilitate shared development of reusable models and simulations among multiple research groups. To this end, GiPSi framework provides an application programming interface (API) for interfacing dynamic models defined over spatial domains, and input/output (I/O) interfaces for visualization and haptics for real-time interactive applications. The framework focuses on addressing the technical issues in accommodating different levels and types of model abstractions, supporting heterogeneous models of computation, and providing mechanisms for interfacing multiple heterogeneous models. GiPSi is

specifically designed to be independent of the specifics of the modeling methods used, and therefore facilitates seamless integration of heterogeneous models and processes.

In this chapter, we present the GiPSi and improvement to the design of GiPSi beyond the first release. In the next section, the GiPSi architecture is introduced in section 5.2. Following by the design improvements of the GiPSi in section 5.3. The network extension to the GiPSi framework called GiPSiNet are presented in section 5.4. The experimental evaluations for GiPSiNet are presented in section 5.5, followed by concluding remarks in section 5.6.

5.2 GiPSi Architecture

The overall system architecture of GiPSi [8] is shown in Figure 5.1. The models of physical processes, such as muscle mechanics of the heart are represented as "Simulation Objects". Each simulation object can be derived from a specific computational model contained in "Modeling Tools", such as finite elements, finite differences, lumped elements, etc. The "Computational Tools" provide a library of numerical methods for low-level computation of the simulation objects dynamics. These tools include explicit/implicit ordinary differential equation (ODE) solvers, linear and nonlinear algebraic system solvers, and linear algebra support. The simulation objects are created and maintained by the "Simulation Kernel" which arbitrates their communication to other simulation objects and components of the system. One such component is the "Input/Output" (I/O) subsystem which provides basic user input provided through the haptic interface tools and basic output through the visualization tools. There are also "Auxiliary Functions" that provides application-dependent support to the system, such as collision detection and collision response tools that are widely used in interactive applications.

An important difference of GiPSi from earlier efforts at developing open simula-

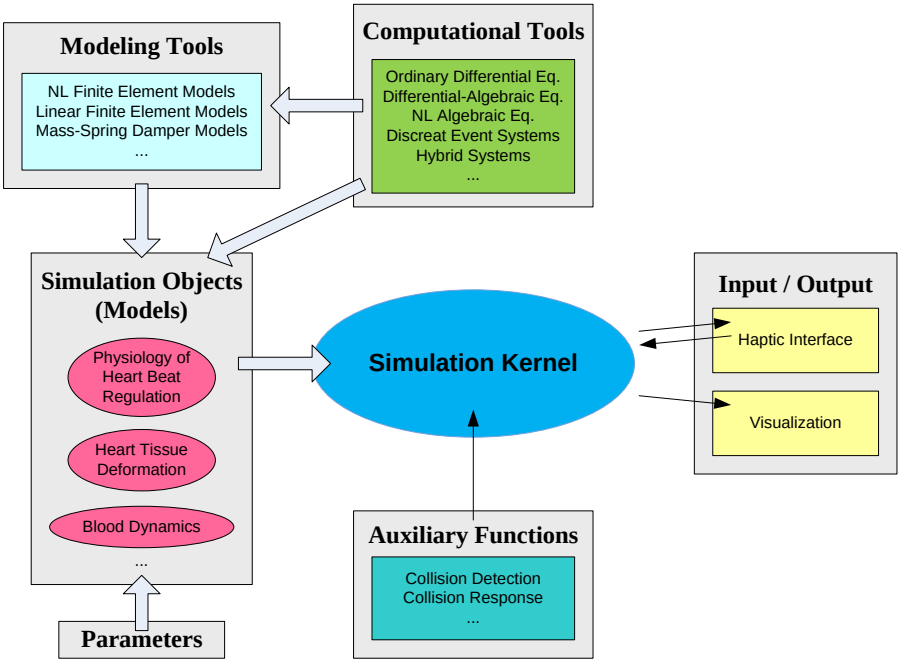


Figure 5.1: The architecture of a GiPSi-based simulation system.

tion toolkits for surgical simulations (e.g., SPRING [6], AlaDyn-3D [35], CAML [20], VRASS [82]) is that the APIs in GiPSi were designed with a special emphasis on being general and independent of the specifics of the implemented modeling methods. In earlier dynamic modeling frameworks, such as SPRING [6] or AlaDyn-3D [35], the underlying models used were woven into the specifications of the overall frameworks developed. This allows GiPSi to seamlessly integrate heterogeneous models and processes, which is not possible with the earlier dynamic modeling frameworks [20]. SOFA is a recent open source framework being developed, primarily targeted at real-time simulation with an emphasis on medical simulation [10]. SOFA provides a modular and flexible software framework, and aims to allow independently developed algorithms to interact together within a common simulation while minimizing the development time required for integration.

5.3 Design Improvements

The first release of GiPSi version 1.0 (2002) provided the basic functionality to verify the concepts of GiPSi. As part of the research presented here, the design of GiPSi has been significantly extended and partially revised to add more functionality that is suitable with our research objectives. The following of design improvements and extensions have been made to the first release of GiPSi framework.

Real-time Simulation Kernel

A simulation kernel acts as the central core to connect the GiPSi components together. The simulation kernel represents the application itself which is constructed by the application developer. The GiPSi only provides the simulation kernel for a typical interactive simulator. The simulation kernel is the running loop to execute simulation objects, connectors, collision detection and collision response, display functions, and user interface command execution. Each of the simulation objects needs to evaluate its system state at the next simulation time step by using numerical integration methods. Each of the simulation objects may also use different simulation time step values depending on the physical properties of the object. Moreover, collision detection and collision response procedures require some processing time to find and resolve overlapping objects. The simulation kernel also records the time stamp for all simulation processes, namely, the time spent on each of the simulation objects and connectors, and time spent in the collision detection and response, display function, user interface command execution, and time step adjustment.

The simulation object has two time steps; `simTimestep` and `clockTimestep` which are corresponding to the simulation time step and the exact time spending on computational time used in simulation step. The total time used in each simulation step is the sum of all computation times in the individual functions which may be over or under the real-time system. However, a real-time simulator requires the

simulation time step to match the clock time step. Therefor, for every simulation object, the simulation time step needs to be adjusted by increasing or decreasing simulation time step to match the clock time step.

An interactive simulation can be classified into three categories based on the relative values of the simulation and clock time steps. These categories are supra real-time, real-time, and sub real-time (Figure 5.2). A supra real-time system is a system where the clock time step is less than the simulation time step. A real-time system is a system where the clock time step is equal to the simulation time step. A sub real-time system is a system where the clock time step is greater than the simulation time step. For example, if the `clockTimestep` in simulation object is 0.05 ms and the `simTimestep` is set up to 0.05 ms, then the simulation runs in the real-time, i.e., the simulation time and actual clock time go at the same speed. If the `clockTimestep` is less than the `simTimestep`, then the simulation runs faster than the actual clock time. If the `clockTimestep` is greater than the `simTimestep`, then the simulation runs slower than the actual clock time. The supra real-time case is the ideal case for interactive simulation because the difference between the `simTimestep` and the `clockTimestep` can be used for other simulation tasks. A sub real-time system needs to be adjusted so that the `simTimestep` matches the `clockTimestep` by increasing the `simTimestep` or changing the numerical integration method to reduce the necessary computation time, and therefore the `clockTimestep`. Increasing the `simTimestep` may introduce instability to the simulation object, and changing the numerical integration method to an algorithm which allows longer stable `simTimeste` increases computation time and therefore `clockTimestep`. In this operation condition, the numerical stability of the simulation object would be maintained if the minimum simulation time step required for every simulation object is longer than the total processing time required for a single kernel execution, i.e. clock time step. The stability of numerical integration methods in deformable object is discussed in detail

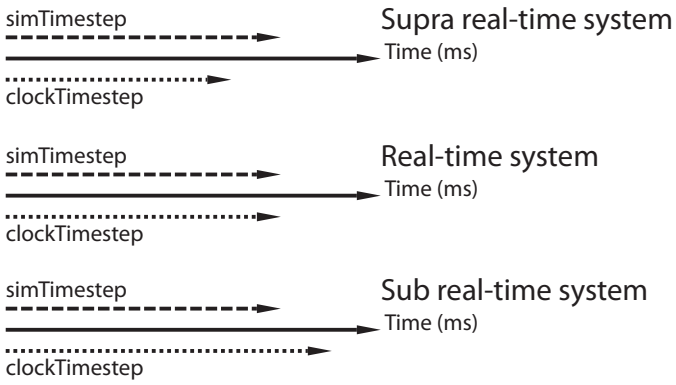


Figure 5.2: Supra real-time, real-time, and sub real-time systems

in Chapter 3.

Simulation Order

The main tasks of simulation kernel are the management of object interactions and the communication between the components. These involve establishing the execution order of simulation objects and other components, which include connectors, collision detection and response, display function, and user interface command execution. Different execution orders produce the different simulation results depending on the model and connector semantics. The execution order of simulation objects and other components inside simulation kernel is left to the application developer because the simulation order is not explicitly specified as part of GiPSi. However, the design, integration and implementation of generic simulation order into GiPSi is helpful for application developer to speed up the application development process.

The simulation order is divided into two parts, which are in project file and in the simulation kernel. This helps the application developer to order the execution of simulation object and other components in project file with no need to change the code in the simulation kernel. The GiPSi project file uses the XML (Extensible Markup Language) format as the description information about simulation objects, connectors, visualization, and collision detection and response. Adding the simulation order

element in project file speeds up the application developer to build the simulator. The new element node `<simulationorder>` is added into the project file. The contents inside this element node are the list of `<object>` in the execution order. The `<object>` element has two elements inside which are `<name>` for the name of simulation object or connector, and `<type>` for identification of what type of `simObject` or `connector` is presented. If the simulation object or connector is not provided in this list, the simulation kernel automatically adds the simulation objects and connectors into the list in the order of appearance in project file. Assuming the project file has simulation objects: S1, S2, S3 and connectors: C12, C23 in order of appearance in project file. The example of `<simulationorder>` that produces the execution order: S3, S1, C23, S2, C12 is the following:

```
<simulationorder>
  <object>
    <name>S3</name>
    <type>simObject</type>
  </object>
  <object>
    <name>S1</name>
    <type>simObject</type>
  </object>
  <object>
    <name>C23</name>
    <type>connector</type>
  </object>
</simulationorder>
```

The pseudo codes for the `CreateSimulationOrder()` and `Simulate()` are give in algorithm 2 and 3.

Implicit Numerical Integration

The physically based deformable model results in a system of equations which cannot be solved analytically; therefore, a numerical integration method needs to be used.

Algorithm 2 Create simulation order procedure

```
1: procedure SimulationKernel::CreateSimulationOrder(XMLNode)
2:     numSimOrder = numSimObject + numConnector
3:     create normalSimOrder[numSimOrder]                    ▷ order from project file
4:     index = 0
5:     for all object ∈ <simobject> do
6:         normalSimOrder[index] = <simobject>
7:         normalSimOrder[index].type = 1
8:         normalSimOrder[index].flag = false
9:         index = index + 1
10:    end for
11:    for all object ∈ <connector> do
12:        normalSimOrder[index] = <connector>
13:        normalSimOrder[index].type = 2
14:        normalSimOrder[index].flag = false
15:        index = index + 1
16:    end for
17:    create simOrder[numSimOrder]                ▷ order from <simulationorder>
18:    index = 0
19:    for all object ∈ <simulationorder> do
20:        object.name = <name>
21:        object.type = <type>
22:        if object.name ∈ normalSimOrder then
23:            simOrder[index] = object
24:            set object.flag in normalSimOrder to true
25:            index = index + 1
26:        end if
27:    end for
28:    for all object ∈ normalSimOrder do
29:        if ¬ object.flag then
30:            simOrder[index] = object
31:            set object.flag in normalSimOrder to true
32:            index = index +1
33:        end if
34:    end for
35: end procedure
```

Algorithm 3 Simulate procedure

```
1: procedure SimulationKernel::Simulate
2:     for all object ∈ simOrder do
3:         if object.type=1 then                    ▷ object is simulation object
4:             (SimObject*)simOrder.simulate()
5:         else                                     ▷ object is connector
6:             (Connector*)simOrder.process()
7:         end if
8:     end for
9:     collision.detection()
10:    collision.response()
11:    time = time + timestep
12: end procedure
```

The key issue of numerical integration is the numerical error which accumulates in each time step, and causes the instability after several time steps. The first release of the GiPSi implementation only supported the explicit numerical integration algorithms. As part of the research, several implicit numerical integration algorithms are implemented and added into GiPSi. The details of the algorithms are discussed is in Chapter 3.

Simulation Object

The simulation objects represent organs and the physical processes associated with them. The `SIMObject` class diagram of GiPSi has been revised to categorize the type of simulation objects. Figure 5.3 shows the class diagram of `SIMObject` and derived classes. The main derived class is `SolidObject` for solid simulation object. The `SolidObject` also has two derived classes, which are `DeformableSolidObject` and `RigidSolidObject`. The `DeformableSolidObject` is a deformable solid simulation object which contains finite element object (`FEM_3LMObject`), lumped element object (`MSDObject`), quasi-static decoupled spring object (`QSDSObject`), and balloon object (`BalloonObject`). The `RigidSolidObject` is a rigid solid simulation object which is an object related to surgery instrument. In the simulator, the surgery instrument

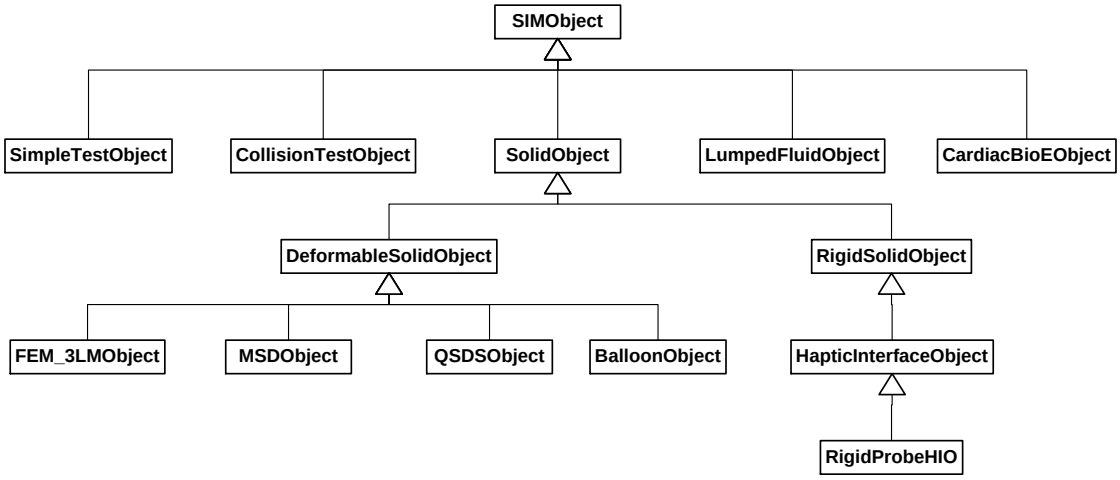


Figure 5.3: Class diagram of `SIMObject` and derived classes.

is represented by haptic device and captured by `HapticInterfaceObject` which is derived from `RigidSolidObject`. `SimpleTestObject` is a simple simulation object which does not do anything other than displaying itself. The `SimpleTestObject` is suitable for test objects or static simulation objects. `CollisionTestObject` is a simulation object for testing the collision detection and response. `LumpedFluidObject` is a lumped fluid model and `CardiacBioEObject` is a simple cardiac bioelectricity model.

Extended Geometric Data Structure

The `SIMObject` is composed of `Geometry` class for display the simulation object geometry. The `Geometry` is the geometry based class and the derived class of `Geometry` class is used for storing information related to visualization of simulation object. As part of research, an extended geometric data structure is implemented and added into GiPSi.

The class diagram of `Geometry` and its derived classes shows in Figure 5.4. The new derived classes are `VectorField` and `ExtendedTriSurface`. The `VectorField` geometry uses for display the vector field of each vertex in simulation object, which represent the normal of vertex or the force feedback applied to vertex. It is useful

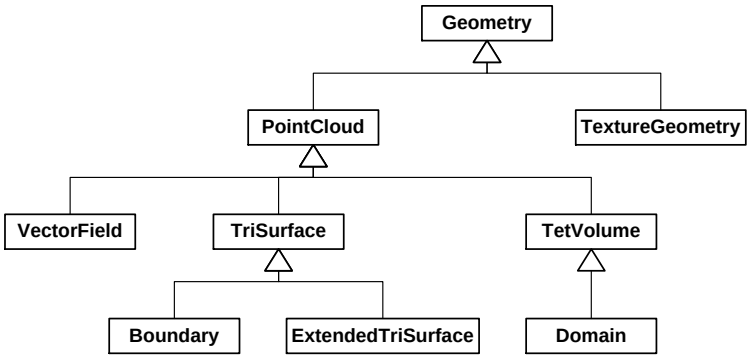


Figure 5.4: Class diagram of `Geometry` and derived classes.

in debugging process for checking collision detection and response algorithms when the developer wants to visualize the forces feedback acted on collided vertices. The current implementation of `TriSurface` geometric data structure in the GiPSi stores only vertices and face-to-vertex table which does not support the geometric manipulation primitives in effective way, such as adding and deleting vertices or faces. The `ExtendedTriSurface` is created to solve this issue by storing the adjacency information. In order to manipulate geometry in efficiency, the reverse table of face-to-vertex or vertex-to-face table should be stored which can retrieve neighboring faces of vertex quickly. Moreover, the edge-to-face table is also stored to keep track of relationship between the edge and faces that belong to this edge.

The vertex-to-face and edge-to-face tables are dynamic, growable data structure associated with each vertex, and each edge, respectively. The linked list of indices of faces in `TriList` and `EdgeList` are maintained for each vertex and edge in geometry, respectively. In `ExtendedTriSurface`, the linked list `TriList` and `EdgeList` are added. When the geometry is loaded, the vertex array, face-to-vertex table, and edge-to-face table are created from file. The vertex-to-face table is created by looping all faces and inserting a face index into the linked list `TriList` of those vertices. The edge-to-face table is created by looping all faces and their three edges. If the edge does not exist in the linked list `EdgeList`, then adds this edge into the linked list with storing the vertices and faces of this edge.

Memory management for resizing allocation memory of vertices and faces is also implemented in `ExtendedTriSurface`. When the size of vertices or faces is changed, the memory management allocates the new memory size of vertices or faces by double in the size of original and then copies the data to the new location. After that the old data location is deleted. The functionality of memory management is a useful utility for `ExtendedTriSurface` to provide the efficiently functions of adding new vertex or face.

Multi-Display Manager

The original GiPSi display manager handles only one geometry at a time. As part of research in some situation one geometry per display manager is not enough to display the simulation object. For example, in collision debugging environment, the developer needs to see the visualize of the overlapping area of collided object. The multi-display manager can handle this situation by adding another geometry for only collided faces and display them on top of original geometry. Another situation when the developer needs the detail in some part of simulation object, such as the hole from the poking action. The multi-display manager handles the issue by replace the specific area of interest with high resolution geometry on top of the original location. The `MultiTriSurfaceDisplayManager` is derived from `DisplayManager` based class and uses dynamic array to store multi-geometry. The idea of multi-display manager is using only one big display array and index array to storing all geometries by expansion the arrays to fit the need at run-time. When the `Display()` function is called, all geometries in display manager write the information to display array and index array by topping up the size of display array and index array of each geometry to make array continuer. This display manager can also use as the normal display manager with only one geometry per simulation object. The extended geometry also supports the original mechanism for display the simulation object with OpenGL can handle

the same way as the original display manager.

Collision Detection and Response

In the first release of the GiPSi, the collision detection and response as the auxiliary functions were not implemented. As the part of research the collision detection and response algorithms are implemented and integrated into GiPSi, as presented in Chapter 4. The collision rule is also designed and integrated into GiPSi. The collision rule allows the application developer to select which simulation objects should perform the collision detection and response.

Haptic Interaction

Haptic interaction is a common interaction interface in virtual environment simulation. Haptic device interfaces a user with the sense of touch by capturing movements of the user and rendering force, vibration, and motion to user depending on appropriate situation. Haptic interaction with deformable objects is an increasingly used in many simulations, such as virtual clay [83], cloth design system [84], and especially in surgical simulations. An important issue in haptic interaction with deformable object simulation that has implications in the design of the haptics middleware and the related APIs is difference between simulation and haptic interface sampling rates. The simulation typically runs at graphical update rate of 10 Hz order of magnitude. However, haptic interface requires update rate in the order of 1 kHz. It is not possible to increase the update rate of the physical model to match the update rate of haptic interface due to computational limitations. Multi-rate simulation approaches are used to address the sampling rate mismatch issue. Çavuşoğlu and Tendick [85] proposed a multi-rate simulation approach which uses a local linear approximation to handle the difference between the update rate requirements for the physical model and haptic interface. This method was adopted in GiPSi.

Linearized Low-Order Approximations

A linearized low-order approximation is an order reduction of the lumped element model in a single contact with haptic device [85, 86]. The linear low-order approximation is constructed and sent to the local model to generate the force feedback to the haptic device at haptic update rate. This method is integrated into the GiPSi framework by `MSDModel` and `MSDModelBuilder` classes. The `MSDModel` is the container class to store the information of masses and springs for building a linearized low-order approximations of lumped element model. The `MSDModelBuilder` fills the connectivity information into the `MSDModel`. When the haptic device contacts the node in the lumped element model, the `ReturnHapticModel()` function in `MSDObject` class is called to create the linearized low-order approximations model at a contact node. The `MSDModelBuilder` retrieves the connectivity around the contact node, such as the neighbor nodes and springs and identifies what type of springs. There are three types of spring, which are contact, internal, and boundary springs. The contact spring is the spring that has one contact mass node. The boundary spring is the spring that has one boundary mass node. The internal spring is the spring that has neither contact nor boundary mass node. The boundary of model is defined by the depth of neighbor contact mass node. After that, the low-order linear model is stored in `GiPSiLowOrderLinearHapticModel` structure and sent to the haptic device to generate the force feedback in haptic callback.

User Interface

The GiPSi user interface is the control sequences of command that user employs to control the state of GiPSi simulation, such as the keystrokes from the computer keyboard. The `UserInterface` based class contains the constructor and destructor, the simulation kernel pointer and the function to set the simulation kernel pointer. The `EndoSimUserInterface` class is derived from the user interface class for managing

the key presses from user. This derived class does the mapping between key press character and user interface command string. The `EndoSimUserInterface` searches the user interface command string corresponding to the key press character when user presses a keyboard. After that the `EndoSimUserInterface` sends the user interface command string to simulation kernel buffer and simulation kernel executes the user interface command string. The user interface is implemented in simulation kernel to get and execute the user interface command. The `setUICommand()` is added to simulation kernel to get the user interface command string controlling simulation kernel from `EndoSimUserInterface` and save it in buffer, while the user interface command string controlling visualization is executed immediately in visualization engine. The `executeUICommand()` is also added to simulation kernel to get the user command string from buffer and execute it in simulation kernel. After that the user command string buffer is cleared.

5.4 GiPSiNet

The first release of the GiPSi framework provided an API between the simulation kernel and input/output interfaces (haptic, visualization). By expanding and adding a network extension to GiPSi provides benefit to user to use surgical simulation over the network. Users can access and perform the simulation any time from any appropriate network access point. Network extension of GiPSi involves a middleware module (GiPSiNet) to improve the lack of network QoS and to enhance the user-perceived quality of a networked simulation. In order to extend the kernel-I/O API to a network environment, a middleware layer (GiPSiNet) is added between the simulation kernel and the input/output subsystem; such a middleware encapsulates all networking functionalities (Figure 5.5). The resulting simulator involves two communication endpoints, which are client and server. The client interacts with the end-user

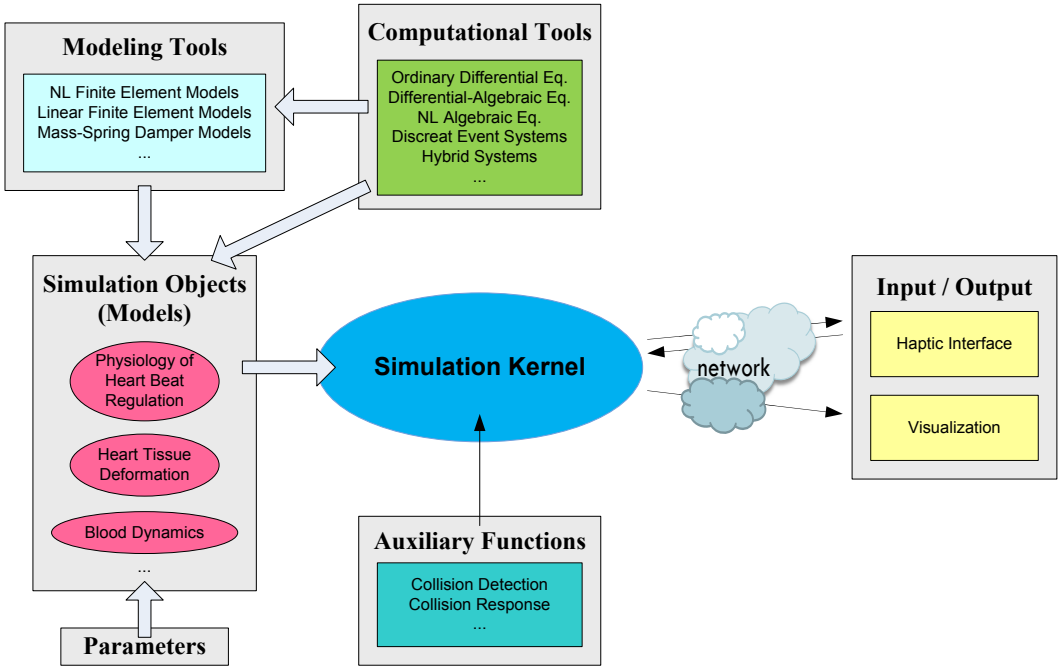


Figure 5.5: The GiPSiNet software architecture. Shaded modules were part of the existing GiPSi platform, clear modules were part of the GiPSiNet middleware [2].

(e.g., surgeon, trainee) through the haptic and visualization interfaces, and the server is the host for running simulation kernel, simulating the physical models, and collision detection and response. The middleware transparently incorporates the complex adaptive methods that are described in [2]. Moreover, the networked framework inherits from GiPSi its flexibility and potential for extensibility to a variety of surgical simulations. The design of the middleware layer (including the design of the client and server side proxies for visualization engine and the haptic manager) have been done by Dr.Cai, and has been explained in detail in Appendix A.1 for completeness.

GiPSiNet is the extension of GiPSi [8] to networked virtual environments via the Internet by using real-time CORBA with TAO [87, 88]. CORBA is a middleware technology that supports the construction and integration of client-server application in heterogeneous distributed environment. CORBA is stand for the Command Object Request Broker Architecture which is the result of a standardization consortium, called OMG (Object Management Group), involving more than six hundred interna-

tional software companies. The ACE ORB (TAO) is an open source implementation of a standard CORBA Object Request Broker (ORB) which based on the standard Object Management Group (OMG) CORBA reference model, and is constructed using s software concepts and frameworks provided by the Adaptive Communications Environment (ACE). The ACE/TAO is implemented by using C and C++ languages and developed by the researchers at Washington University, St.Louis, University of California, Irvine and Vanderbilt University. The TAO is an open-source available online at http://www.cs.wustl.edu/∼schmidt/TAO.html.

The GiPSiNet middleware between the client and the server uses the *remote proxy design pattern* [89], which provides a surrogate or placeholder for another object to control access to it. The remote proxy pattern uses a proxy to hide a service object, which is located on a different machine from the client objects. The client objects request the service from the service proxy, which is located on the same machine. The service object and the service proxy are transferring the information transparently like the service and the client are in the same machine. In order to provide a network functionality with transparency to the original GiPSi API, the simulation kernel proxy (SKP), haptic manager proxy (HMP), and visualization engine proxy (VEP) are created to be the proxy of simulation kernel, haptic manager, and visualization engine, respectively. In GiPSi data flow, the simulation kernel communicates with the haptic manager and visualization engine directly. In GiPSiNet data flow (Figure 5.6), the HMP and VEP communicate with the SK at the server side, and the SKP communicates with the HM and VE at the client side, instead of communication directly with the original objects.

5.4.1 GiPSiNet Mechanism

GiPSiNet is the network extension of GiPSi which contains server and client shown in Figure 5.7. In the networked environment, the server contains the simulation kernel,

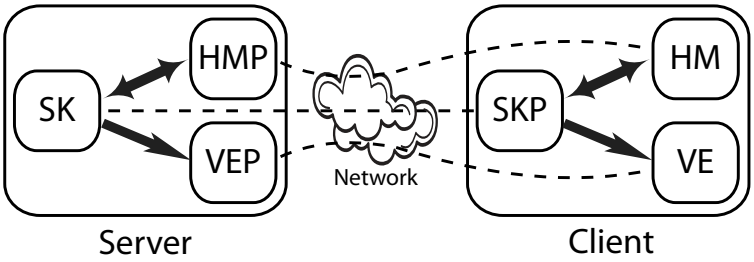


Figure 5.6: GiPSiNet middleware by using remote proxy pattern.

and the client contains user interface, haptic manager and visualization engine. The user interface manages user inputs, such as, key presses on the keyboard, opening of the project file which contains the parameters to set up a simulation, closing of the connection between the client and the server. The haptic manager manages haptic devices which capture input position from user and render force feedback to user. The visualization engine manages display output from simulation. Proxies between GiPSiNet server and client, specifically between simulation kernel at the server and the haptic manager, visualization engine and user interface at the client, are implemented using CORBA/TAO middleware.

From the view point of CORBA/TAO middleware, the haptic manager and visualization engine are the services running in the client side and the user interface is embedded in the visualization engine. When a user starts the client, the user interface connects to the server and selects the project file to load. Client sends selected filename to the server to retrieve the project file and send that project file back to the client. Then, both the server and the client loads this project file. After that the client starts the haptic manager and visualization engine services and sends a signal to the server to notify that the haptic manager and visualization engine services are started in the client. The server then starts haptic manager and visualization engine proxies. These let the visualization engine and haptic manager connect to the simulation kernel via the visualization engine and haptic manager proxies. Two communication services, which are simulation kernel-haptic manager and simulation

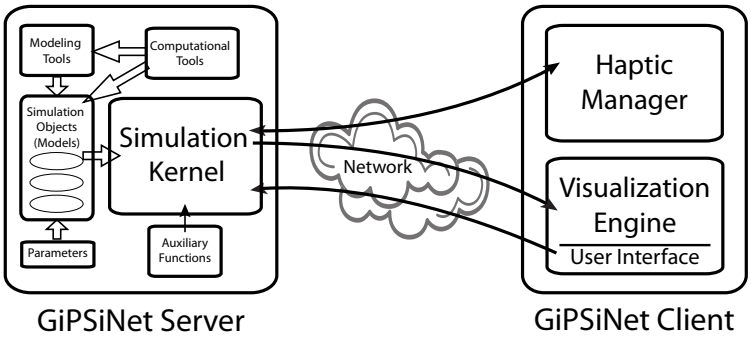


Figure 5.7: GiPSiNet Client - Server Architecture

kernel-visualization engine, are created separately to transfer the data independently to the simulation kernel.

Figure 5.8 shows the communication flow chart between the client and the server. The start and stop processes are described in detail below (Figure 5.8). The CORBA naming service is required for mapping CORBA objects, and needs to run all the time in GiPSiNet communication. The server initially starts and waits for connection of the client by using TAO with that naming service. When the client starts, the client makes a connection with the server with the same naming service. The client then retrieves the XML data from the server by using the `getProjectXML()` function. The server opens the project file and sends the XML data back to the client. The client loads this project and sends an acknowledgment signal to the server by using `loadedProject()` function. When the server gets the signal, the server loads the project file. Then, the client starts the haptic manager service and sends an acknowledgment signal to the server by using `startedService()` function with "HM" parameter. When the server gets the signal, the server starts the haptic manager proxy. The same events occur for the visualization engine service. The client starts the visualization engine service and sends an acknowledgment signal to the server. When the server gets the signal it starts the visualization engine proxy. After that the client starts the visualization engine with the graphical user interface (GUI) loop and the server starts the visualization engine proxy loop at the same time. In these loops, the

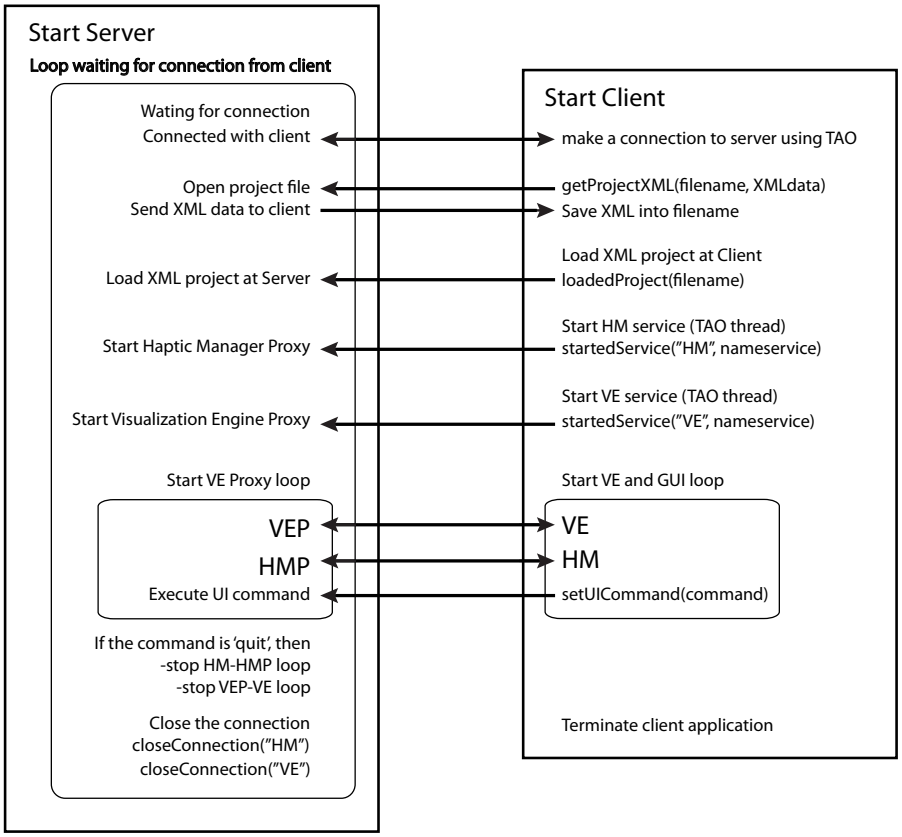


Figure 5.8: GiPSiNet communication flow chart

communication between VEP-VE and HMP-HM are established to send data between the server and the client. The HM sends the data of the haptic device position and receives the data of the force feedback to generate the feedback to the user. The VEP sends the visualization buffer to the client for display of the simulation objects at the user terminal. Moreover, the user interface communication is also captured in these loops. The details of visualization engine proxy and haptics manager proxy, which are collaborated with Prof.Liberatore and Dr.Cai are described in Appendix A.1. The detail of user interface is described in the next section. The stop process occurs when the user presses "quit", then the `setUICommand()` function is called and sends the string "exit" to the server. The server executes the command by terminating the simulation and calls the `closeConnection()` function, which stops the haptic manager proxy and visualization engine proxy loops.

5.4.2 GiPSiNet User Interface

The user interface is a way to control the system by user by using keyboard, mouse or other input devices. The simple user interface is a command line interface that controls the system by pressing a keyboard. Later, a graphical user interface (GUI) was introduced to control the system with richly graphical design for easy using. The GiPSi framework does not provide the user interface itself, because the user interface is dependent on the developed application. GiPSi only provides the necessary API for user interface to control the state of simulation kernel.

The user interface in GiPSi is based on the transmission of command from visualization engine to simulation kernel (Figure 5.9a). A user interface string buffer is introduced between visualization engine and simulation kernel to store the commands that need to be processed (Figure 5.9b). There are two sets of user interface commands, which are for controlling the simulation kernel and controlling the visualization engine. The user interface command strings for controlling the simulation kernel from the visualization engine are sent to the simulation kernel and saved in a buffer, while the user interface command strings for controlling the visualization are executed immediately by the visualization engine.

For remote user interface, the user command buffer is transmitted over the network by implemented the `setUICommand()` function in simulation kernel proxy, which derived from simulation kernel in Figure 5.9c. This function sends the user interface command from simulation kernel proxy at the client to simulation kernel at the server and executes the user command in the same way as GiPSi does. This means that only user interface commands controlling simulation kernel are transmitted over the network.

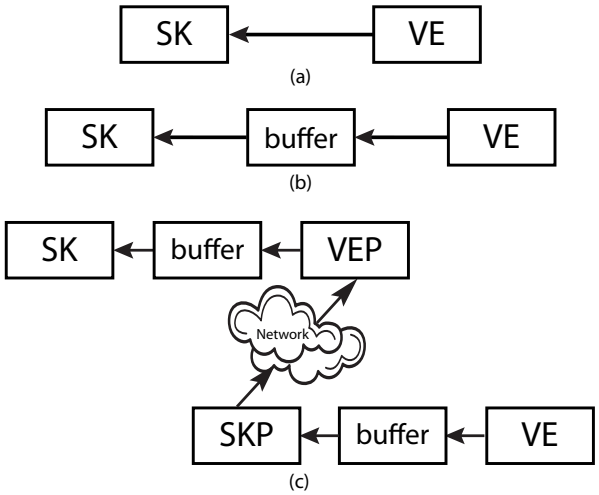


Figure 5.9: User interface in GiPSi (a,b) and remote user interface (c).

5.5 Experimental Evaluation

The simulation was tested and benchmarked on the Microsoft Windows XP^{TM} 32-bit based workstation with the Intel Pentium D 2.80 GHz, 1 GB of RAM and a PCI Express NVidia GeForce 8800 Ultra Graphics Card with 768 MB of memory to collect performance benchmarks for non-networked environment. For the networked environment configuration was the following. The server was the Microsoft Windows XP^{TM} 64-bit based workstation with dual Intel XEON 2.8 GHz, 8 GB of RAM used for GiPSi core computations, which were simulation kernel, simulation objects evaluating with numerical integration, and collision detection and response. The client was the workstation with the Intel Pentium D 2.80 GHz, 1 GB of RAM and a PCI Express NVidia GeForce 8800 Ultra Graphics Card with 768 MB of memory connected with the PHANTOM Ommi® haptic device. Both workstations installed CORBA/TAO environment and had a gigabit network card connected with AT-GS916GB 16 port 10/100/1000T unmanaged switch. The network connection used was a low latency dedicated network connection without significant cross traffic. The naming service was running on the client machine.

We implemented an endoscopic neurosurgery training simulator (Figure 5.10) as

a test bed to evaluate the GiPSi/GiPSiNet functionality. The simulation was composed of geometric models of the anatomy obtained from a commercial company. The simulation consisted of 20 simulation objects and 13 texture objects. The complete list of simulation objects and texture objects was in the Appendix A.2.1. For example, two choroid plexus models were composed of 6,000 nodes and 11,996 faces in each model. The ventricle system model was composed of 9,145 nodes and 17,942 faces with quasi-static decoupled spring model which every node in model anchored to the original position with springs. The ventricle floor model was composed of 233 nodes and 286 faces with lumped element model of 74 masses and 249 springs. The implicit Euler numerical integration method with simulation time step of 0.01 second was used for lumped element models in simulation. The simulator was run 10 times and the results averaged. The visualization engine was operating at 43 frames per second in non-networked environment. In networked environment, the visualization engine was operating at 54 frames per second. This because the GiPSi/GiPSiNet was designed with multi-threading processes. In non-networked environment all threading processes ran on same machine which decreased the visualization performance from intensive computation in simulation kernel, while in networked environment visualization engine and simulation kernel are separated in different machine.

We also measured and compared the time spent in the main five communication functions, which were related to the communication of the project file, user interface command, display arrays, linearized low-order approximation haptic models and haptic device configurations. The turnaround times from invocation to completion of the network communication functions in networked environments were run 10 times and the average results were shown in Table 5.1. The size of project file was 63.38 KB. The total data transfer for the visualization per scene was 8.44 MB. The maximum data transfer of a linearized low-order approximation of haptic model was 12.10 KB per haptic update. The complete list of linearized low-order approximation models

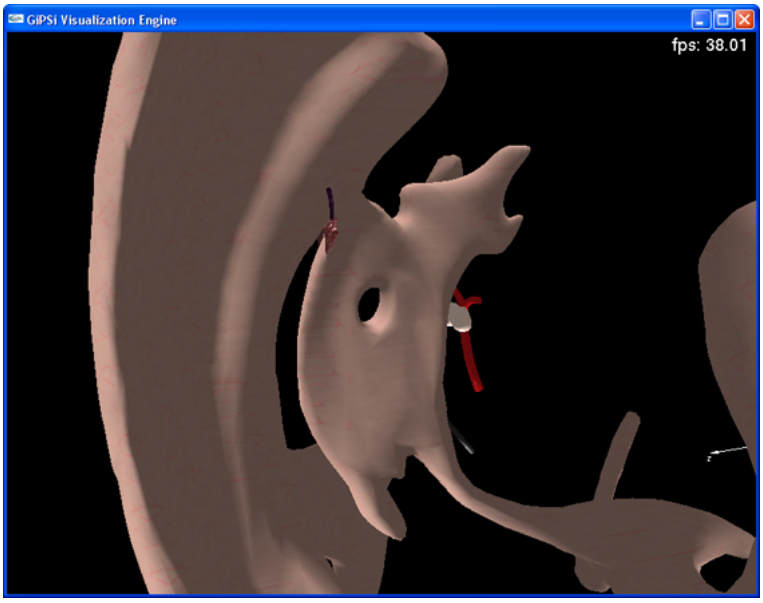


Figure 5.10: The endoscopic third ventricle simulation.

Table 5.1: Transfer time of functions in endoscopic simulator with networked environments. The average turnaround time of visualization and haptic communications reported per frame. Project file transmission occurred once per simulation. User interface commands were issued on demand and repeated per transmission.

	Data transfer (bytes)	Average transfer time (ms)
Transfer project file	63,382	4.26
Set user interface command	4-17	3.14
VEP-setArray()	8,440,428	283.64
HMP-UseHapticModel()	12,102	4.19
HMP-ReadConfiguration()	100	7.41

was in the Appendix A.2.2. The data transfer of haptic device configuration was 0.1 KB per haptic update.

We also implemented a simple test bed (Figure 5.11) with soft tissue model with haptic device to evaluate the network capability. The simulation consisted of 3 simulation objects and 5 texture objects. The soft tissue model had two square geometric models connected together and for each geometric model was composed of 121 nodes and 200 faces. One of the objects was embedded with lumped element model composed of 121 masses and 420 springs and the other object was embedded with quasi-static decoupled spring model also composed of 121 masses and 121 springs an-

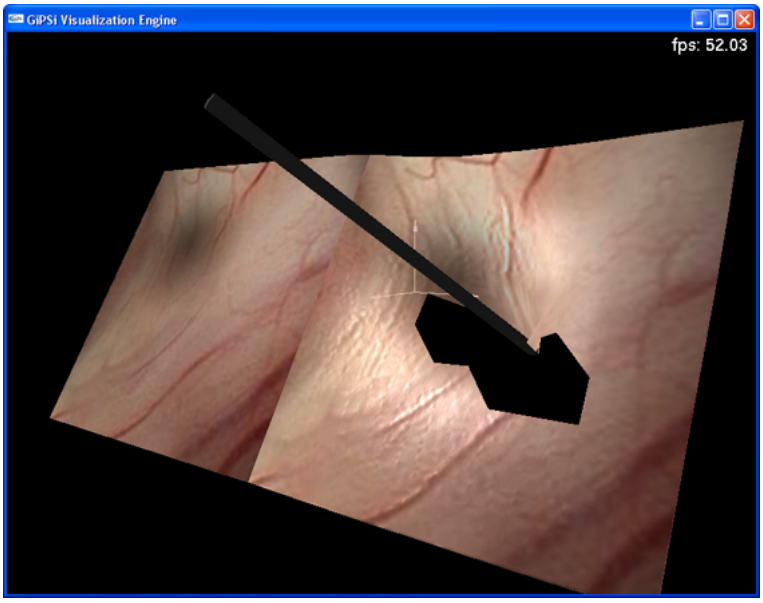


Figure 5.11: The tissue embedded with different physical models, the quasi-static decoupled spring model on the left and the lumped element model on the right.

Table 5.2: Transfer time of functions in simple test bed with networked environments. The average turnaround time of visualization and haptic communications reported per frame. Project file transmission occurred once per simulation. User interface commands were issued on demand and repeated per transmission.

	Data transfer (bytes)	Average transfer time (ms)
Transfer project file	12,265	2.69
Set user interface command	4-17	2.29
VEP-setArray()	3,166,872	92.99
HMP-UseHapticModel()	12,102	2.91
HMP-ReadConfiguration()	100	7.60

chor to the original position. The simulation was run 10 times and results averaged. The visualization engine was operating at 56 frames per second in non-networked environment and 57 frames per second in networked environment. The turnaround times from invocation to completion of the network communication functions were shown in Table 5.2. The size of project file was 12.27 KB. The total data transfer for visualization per scene was 3.17 MB. The maximum data transfer of a linearized low-order approximation of haptic model was 12.10 KB per haptic update. The data transfer of haptic device configuration was 0.1 KB per haptic update.

A detailed profiling of the simulation using the Intel VTune^{TM} performance analyzer was used to measure the sources of overhead resulting from using the GiPSi/ GiPSiNet API in non-networked and networked environments. The sources of overhead in non-networked operation were the data translations between the model state and the display, boundary, and domain geometries, and the data translations occurring inside the connectors. The additional sources of overhead in networked operation were the data transfers occurring between the GiPSi modules and the proxies, and the overhead resulting from the use of TAO functions. The percentage of overhead running time was calculated from the overhead running time divided by the total running time and multiplied by 100%. The double-chamber heart model simulation [8] (Figure 5.12) was tested and benchmarked in the stand alone environment. The visualization engine was operating at 58 frames per second. The profiling analysis revealed that the overhead accounted was 6.18% of the overall application. In network environment, the profiling analysis revealed that the overhead accounted were 7.31% and 4.62% for the client and the server, respectively. For the endoscopic neurosurgery training simulator (Figure 5.10), the profiling analysis in the stand alone environment revealed that the overhead accounted was 15.22% of the overall application. In network environment, the profiling analysis revealed that the overhead accounted were 13.55% and 5.86% for the client and server, respectively. For the simple test bed (Figure 5.11), the profiling analysis in stand alone environment revealed that the overhead accounted was 12.97% of the overall application. In network environment, the profiling analysis revealed that the overhead accounted were 8.49% and 7.15% for the client and server, respectively. The overhead was relatively higher for the endoscopic training simulator because the endoscopic simulator contained a lot of complex geometry objects which the simulation kernel spent a lot of time to transfer the data between the model state and the display, boundary, and domain geometries.

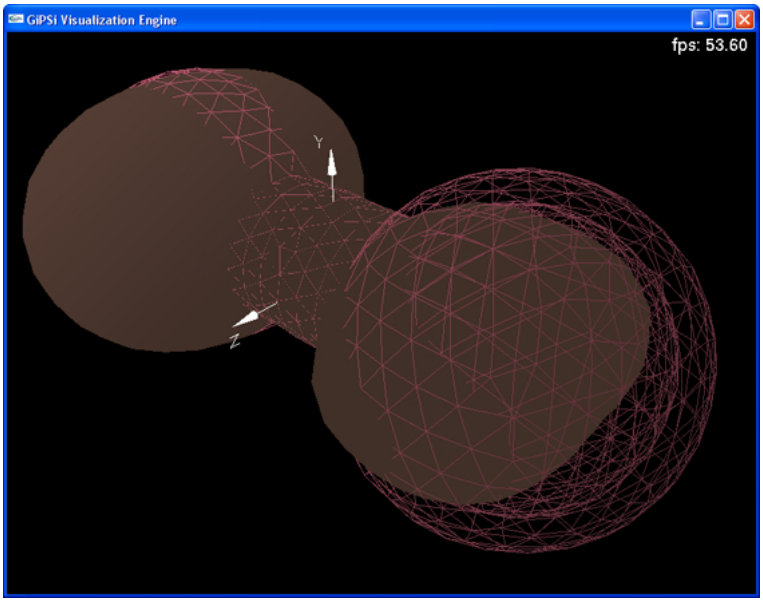


Figure 5.12: The double-chamber heart model simulation.

5.6 Conclusion

In this chapter we presented the GiPSi, an evolving open source/open architecture software framework for developing organ-level surgical simulations, and the extensions and improvements to GiPSi that were done as part of the research. The design improvements include the development of the real-time simulation kernel, execution order in simulation kernel, implicit numerical integration algorithms, revision of simulation object class hierarchy, extended geometry data structure, multi-display manager, collision detection and response algorithm, collision rules, haptic interaction and linearized low-order approximation model, and user interface. Moreover the GiPSiNet, an open source/open architecture networked simulation framework for surgical simulation, is introduced. The GiPSiNet extends GiPSi with network environment and enhances the quality of networked surgical virtual simulation by using CORBA middleware. The GiPSiNet mechanism is presented along with the communications between the client and the server which are the GiPSiNet user command interface, GiPSiNet visualization, and GiPSiNet haptic.

We used an endoscopic simulator and simple test bed as the test applications for GiPSi/GiPSiNet. We captured the data transfer and time used between the client and the server to benchmark the performance of GiPSiNet. As well as the detailed profiling of the simulation was measured by using the Intel VTune^{TM} performance analyzer.

Chapter 6

Endoscopic Third Ventriculostomy Surgery Simulator

6.1 Introduction

Endoscopic surgery is a minimally invasive surgical technique that involves the use of an endoscope, a special viewing instrument which allows a surgeon to see images of the internal body's structures through very small incisions. Endoscopic surgery has been used for decades in a number of different procedures, such as gallbladder removal. The first endoscopic neurosurgery was performed by Lespinasse in 1910 [90]. He performed endoscopy through burr holes with choroid plexus coagulation in two hydrocephalic infants. The first endoscopic third ventriculostomy was performed by Mixter in 1923 [91] in a child with hydrocephalus by using endoscopic guidance by bypassing an obstruction in the ventricular system. Nowadays, endoscopic surgery is applied into many surgery procedures, for example, endoscopic sinus surgery, cholecystectomy (gall bladder removal), endoscopic spine surgery, endoscopic third ventriculostomy surgery, etc.

An endoscope consists of two basic parts: a tubular probe with a tiny camera

and light, which is inserted through a small incision; and a viewing screen, which magnifies the transmitted images of the body's internal structures. During surgery, the surgeon watches the screen while moving the tube of the endoscope through the surgical area. It's important to understand that the endoscope functions as a viewing device only. To perform the surgery, a separate surgical instrument, such as a scalpel, scissors, or forceps, must be inserted through a different point of entry and manipulated within the patient's body. Surgeons should be well-trained and experienced to perform these surgeries. There are many approaches to practice those surgical skills. Computer-based surgical simulation offers the trainers to practice with variety of surgical skills in a controlled, risk-free environment before surgeon performs in actual operation. The benefits of computer-based surgical simulation are reducing cost and time compared with traditional surgical training and increasing the patient safety and surgeon self-confidence.

This chapter introduces a virtual environment-based training simulator for endoscopic third ventriculostomy as a test bed of surgery simulator. The GiPSi [8] framework is used to develop the third ventriculostomy simulator. The remainder of this chapter is organized as follows: In the next section, the symptoms and treatments of Hydrocephalus are described in section 6.2, including the third ventriculostomy surgery, which is an endoscopic surgical technique for treatment of Hydrocephalus. In section 6.3, the third ventricle surgery simulator is described. The elastoplasticity lumped element model which used to model the ventricle floor is presented in section 6.4. Then, the implementations of simulator which are endoscopic surgery instruments and hole poking procedure are presented in section 6.5 and 6.6. After that, the experimental results are presented in section 6.7, followed by concluding remarks in section 6.8.

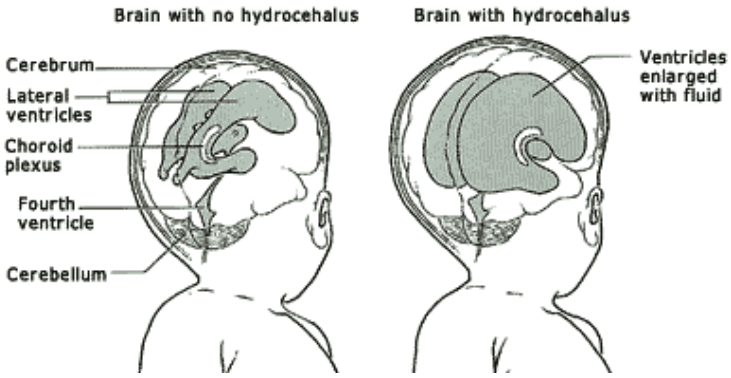


Figure 6.1: A brain without and with hydrocephalus [3].

6.2 Hydrocephalus and Third Ventriculostomy

Hydrocephalus, or called water on the brain, is a condition that the amount of cerebrospinal fluid (CSF) excesses the capacity of the brain and puts pressure to the tissue of the brain causing brain damage as well as causing the head enlargement. Hydrocephalus occurs in one of 1,000 live births. There are two treatments of hydrocephalus, which are shunting procedure and third ventriculostomy procedure. The shunting procedure is introduced by Holter and colleagues in the early 1950s. The shunt procedure diverts the flow of CSF from the Central Nervous System (CNS) to another area of the body by inserting a shunt system. The shunt system consists of a catheter, and a valve. The catheter is a flexible tube made of sturdy plastic. One end of catheter is placed within a ventricle inside the brain and the other end is usually placed within the abdominal cavity. The small valve is located between two ends of catheter to maintain one-way flow and control the amount of CSF flow. An example of the shunt is shown in Figure 6.2. The shunting procedure is not the perfect treatment in long term because shunt may malfunction or cause infection. It needs operative revision several times. The third ventriculostomy is introduced as an alternative hydrocephalus treatment. The third ventriculostomy was performed before shunting procedure by Mixter [91]. However, this procedure was not widespread because the instruments used in surgery at that time were not as good as the instru-

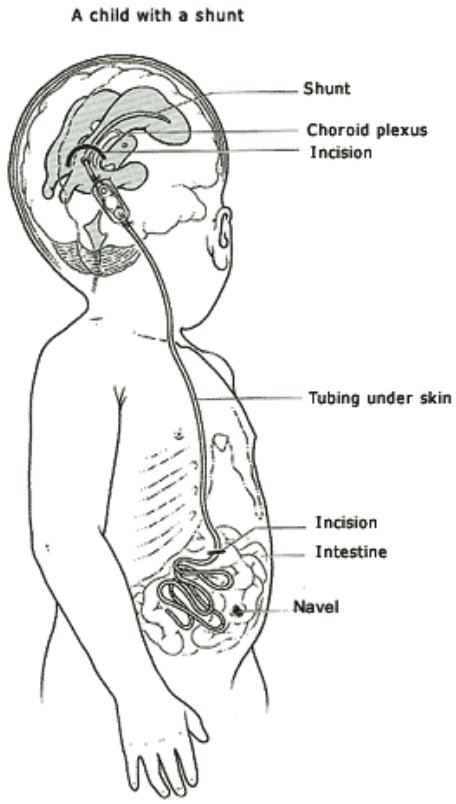


Figure 6.2: An example of a shunt in place [3].

ment used today. In this procedure, surgeon inserts an endoscope into the patient's head to reach the surgical area which is the third ventricular floor and uses a small tool to make a hole in the floor. This hole poking at third ventricular floor allows the CSF to flow normally in the system and reduce the pressure in the brain.

The third ventriculostomy procedure or endoscopic third ventriculostomy (ETV) from Farina at el. [4] is described as follows: A burr hole is made through the skull (Figure 6.3). The endoscope trajectory is aimed to the foramen of Monro and floor of the third ventricle from the burr hole (Figure 6.3). Then, an endoscopic instrument (Figure 6.4) is inserted into the lateral ventricle, and the choroid plexus. The foramen of Monro is located between septal and thalamostriate veins (Figure 6.5). The endoscope is advanced into the third ventricle. The mamillary bodies are the posterior landmark of the third ventricle (Figure 6.6). An opening in the floor

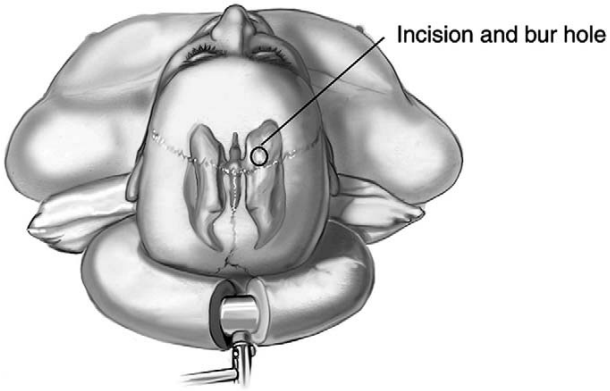


Figure 6.3: Positioning the patient and planning the incision [4].

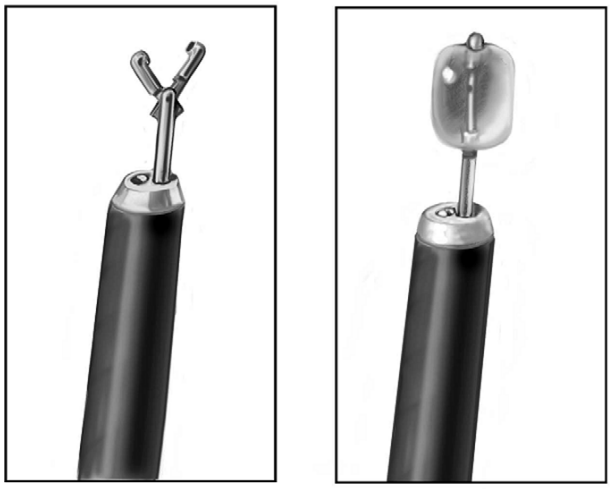


Figure 6.4: Endoscopic instrument with forceps (left) and with balloon catheter (right) [4].

of the third ventricle is initially made with neuroendoscopic instruments and then inflated using a balloon catheter.

6.3 Third Ventricle Surgery Simulator

A third ventriculostomy procedure is a endoscopic surgery which is difficult for surgeon to operate because endoscopic surgery has restricted vision, restricted movement of instruments, difficult to handle instruments. Manipulation of endoscopic instruments with indirect vision through a monitor requires extensive training. A computer-

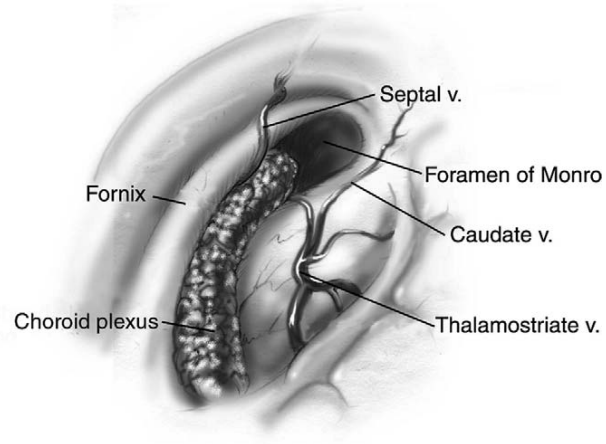


Figure 6.5: Endoscopic intraventricular anatomy [4].

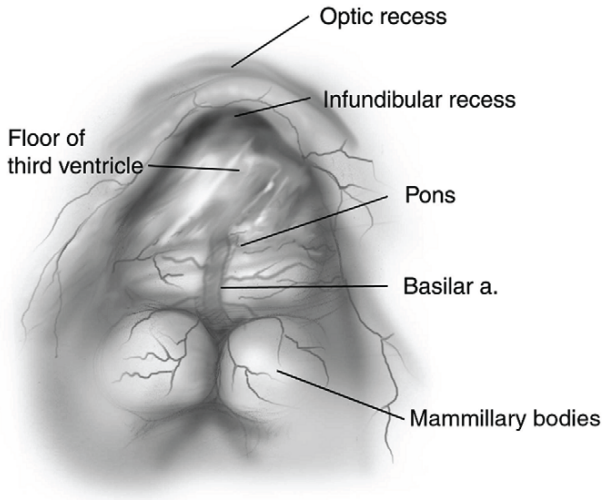


Figure 6.6: Viewing through foramen of Monro targeting the floor of the third ventricle [4].

based simulation training environment is the best solution for training surgeon in the endoscopic surgery procedure.

The third ventriculostomy simulator has been developed by using GiPSi framework. The geometric models of the anatomy used in the simulator were obtained from a commercial company. Then, the area of interest, the third ventricle floor, is extracted for creating the lumped element model as physical model by using parameters determination method as describe in Chapter 2. The third ventricle model building procedure is shown in Figure 6.7.

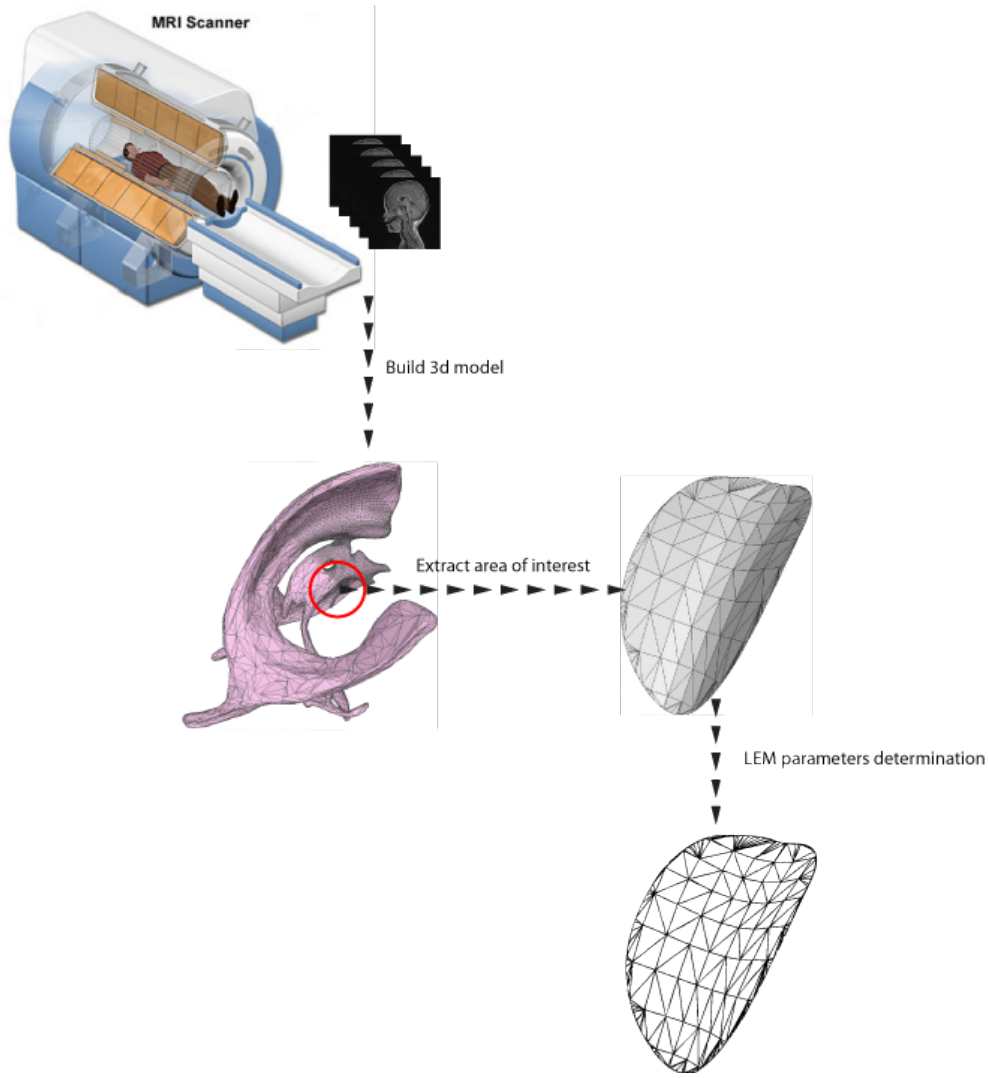


Figure 6.7: Third ventricle model building procedure.

The endoscopic surgery instruments consist of an endoscope, and a balloon catheter. The endoscopic surgery instruments are controlled by using the PHANTOM Ommi® haptic device (SensAble Technologies). An elastoplasticity lumped element model, which is the physically based model, is used to model the third ventricle floor which interacts with the endoscopic instrument. The third ventricle floor can also create a hole when an endoscopic instrument contacts the third ventricle floor model and applies forces in excess of the limits of the physical model.

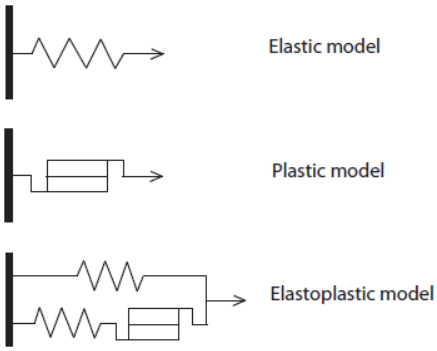


Figure 6.8: Elastoplastic lumped element model.

6.4 Elastoplasticity Lumped Element Model

A soft tissue is not perfectly elastic. A real tissue may exhibit a variety of inelastic properties. For example, the tissue may not restore to its initial shape after the force removal. A deformable object with inelastic properties does not obey the Hooke's law when the forces exceed certain limited values, and the object does not reverse into its initial shape. The effect of permanent deformation is called plasticity. The combination between elasticity and plasticity is called the elastoplasticity shown in Figure 6.8. The elastoplasticity is extended from the normal elastic spring behavior with the plasticity property by adding more parameters, which are the spring constant of plasticity, elasticity limited percentage, and the initial rest length. If the strain exceeds the elastic limitation, the spring rest length is modified and spring constant of plasticity is introduced into the original spring [14].

$$l = \|x_1 - x_2\| \tag{6.1}$$

$$f_s(l) = k_e\,(l - L_0) + k_p\,(L - L_0) \tag{6.2}$$

$$L_{new} = L_{old} + dL \tag{6.3}$$

$$dL = \begin{cases} 0; & L - RL_0 < l < L + RL_0 \\ dl; & l > L + RL_0 \text{ and } dl > 0 \\ 0; & l > L + RL_0 \text{ and } dl < 0 \\ dl; & l < L - RL_0 \text{ and } dl < 0 \\ 0; & l < L - RL_0 \text{ and } dl > 0 \end{cases} \tag{6.4}$$

where l is the displacement between node 1 and node 2, f_s is the spring force from elasticity and plasticity, k_e is the elasticity spring constant, k_p is the plasticity spring constant, L is the rest length of spring, L_0 is the initial rest length of spring, R is the elasticity limited percentage.

The elastoplasticity is implemented into `Spring` class. To validate the elastoplastic mechanism, the simple mass-spring model with two masses connected with one spring is created by applied the external sinusoidal displacement at node 2 and fixed position at node 1. The mass is 0.5 units. The spring has a damping constant of 0.86, elastic constant of 15000 units, plastic spring constant of 1/10 of elastic spring constant. The initial rest length is 2.0 units, the elasticity limited percentage is 0.2. The result of relationship between the force and displacement is shown in Figure 6.9. The spring starts from the rest length at 2.0 units and applies sinusoidal displacement at node 2. When the spring extends to $L + RL_0$ = 2.0+0.2*2.0 = 2.4 units, which reaches the elasticity limit, the plastic spring force is added into the spring force while the elastic spring force maintains the same value from this point. The displacement is continued to be applied until it reaches to 3.0 units. At this point, $L = L_0 + dl$ = 2+0.6 = 2.6 units. Then the displacement is decreased. When the spring length is less than 2.2 units, which is less than $L - RL_0$ = 2.6-0.2*2.0 = 2.2 units, the plastic spring force starts to decrease. The displacement continues to be applied until it reaches to 1.0 unit. At this point, $L = L_0 + dl$ = 2.6-1.2 = 1.4 units. Then the displacement is increased again, until the spring displacement is greater than 1.8 units, which is more than $L + RL_0$ = 1.4 + 0.2*2.0 = 1.8 units. Then ,the plastic spring force starts to increase again.

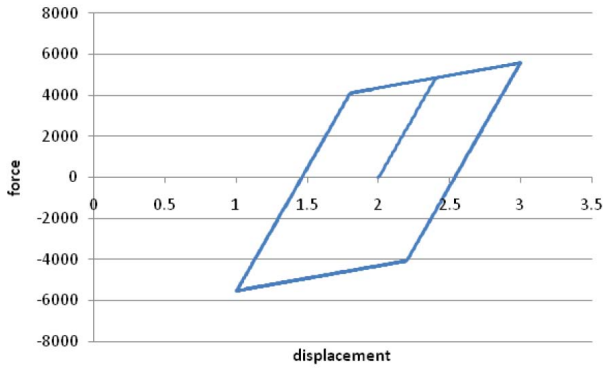


Figure 6.9: Displacement vs. force of lumped element model at node 2.

6.5 Endoscopic Surgery Instruments

Endoscopic surgery is a minimally invasive surgical technique that involves the use of an endoscope, a special viewing instrument which allows a surgeon to see images of the body's internal structures through very small incisions. Endoscopic surgery instruments as simulation objects are designed and implemented as described below.

6.5.1 Endoscope

An endoscope consists of two basic parts: a tubular probe with a tiny camera and light, which is inserted through a small incision; and a viewing screen, which magnifies the transmitted images of the body's internal structures. During surgery, the surgeon watches the screen while moving the tube of the endoscope through the surgical area.

The `RigidProbeHIO` class is used for endoscope which captures the movement of the haptic interface and renders the endoscope in the simulator. The transformation matrix from endoscope to world coordinates (g_{WE}) is used in the simulation by equation:

$$g_{WE} = g_{WL_H} \cdot g_{L_H H} \cdot g_{HE} \tag{6.5}$$

where g_{WL_H} is the transformation matrix from local coordinates of haptic interface to the world coordinates which is defined by user in the project file in `<HIOParameters>`

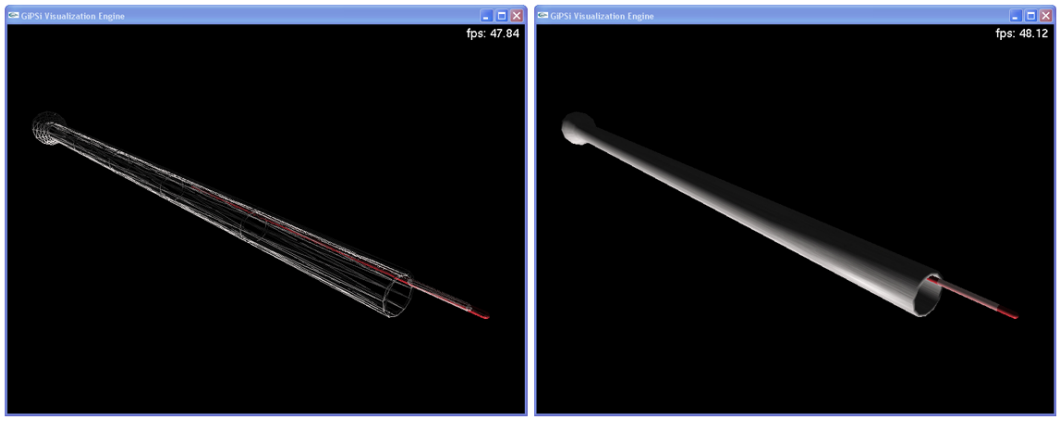


Figure 6.10: Endoscope in simulator using the GiPSi framework.

element, $g_{L_H H}$ is the transformation matrix from haptic interface to the local coordinates of haptic interface which is the position and orientation of haptic device, and g_{HE} is the transformation matrix from endoscope to haptic interface which is defined by user in the project file. The connector between endoscope and catheter is introduced because the catheter is constrained with the tube of endoscope. When the endoscope moves, the catheter should move with the same position and orientation of endoscope. The transformation matrix from the local coordinates of catheter to the world coordinates (g_{WL_C}) is the following:

$$g_{WL_C} = g_{WE} \cdot g_{EL_C} \tag{6.6}$$

where g_{EL_C} is the transformation matrix from local coordinates of catheter to endoscope which is defined by user in the project file. The result of transformation matrix transfers to the catheter. Figure 6.10 shows the endoscope in GiPSi simulator. The left side figures show the objects drawn with wireframe and the right side figures show the objects drawn with texture.

6.5.2 Balloon Catheter

A balloon catheter is a soft catheter with an inflatable balloon at the tip which is used during a catheterization procedure to enlarge a narrow hole. In endoscopic

third ventriculostomy surgery, the balloon catheter is used to poke a hole in third ventricular floor to make the cerebrospinal fluid circulate. The surgeon uses the endoscope as a guide into the brain and inserts the balloon catheter through the hole of endoscope. The tip of the balloon pokes a small hole and then the balloon is inflated to enlarge the hole.

The design of balloon catheter model contains the catheter and balloon objects. The balloon catheter can be moved in and out of endoscope device by the mouse movement. The balloon inflation and deflation can be controlled by the keyboard input from the user which sends user interface command string to simulation kernel. The balloon catheter object is modeled by simple mass spring damper model which can interact with the other simulation objects. The force feedback can be generated and send back to the haptic device.

The balloon is the simulation object separated from the catheter because the balloon can inflate and deflate. The connector between catheter and balloon is introduced to attach these two simulation objects together when the user moves the catheter, the balloon is moved relative to the moving catheter. The transformation matrix from catheter to world coordinates (g_{WC}) is used in the simulation by equation:

$$g_{WC} = g_{WE} \cdot g_{EL_C} \cdot g_{L_CC} = g_{WL_C} \cdot g_{L_CC} \tag{6.7}$$

where g_{L_CC} is the transformation matrix from catheter to the local coordinates of catheter which is defined by user in the project file. The transformation matrix from the local catheter to the world coordinates (g_{WL_C}) can get from the connector between the endoscope and catheter. The connector between catheter and balloon is introduced because the balloon is at the tip of the catheter. When the catheter moves, the balloon should move with the same position and orientation of catheter. The transformation matrix from the local coordinates of balloon to the world coordinates

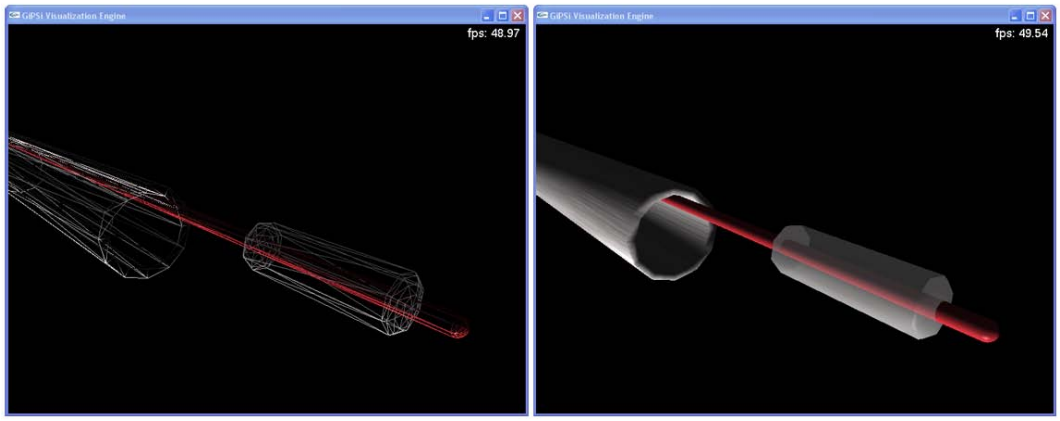


Figure 6.11: Balloon catheter in simulator using the GiPSi framework.

(g_{WL_B}) is calculated as the following:

$$g_{WL_B} = g_{WC} \cdot g_{CL_B} \tag{6.8}$$

where g_{CL_B} is the transformation matrix from local coordinates of balloon to catheter which is defined by user in the project file. The result of transformation matrix is sent to the balloon object.

The transformation matrix from balloon object to world coordinates (g_{WB}) is used in the simulation by equation:

$$g_{WB} = g_{WC} \cdot g_{CL_B} \cdot g_{L_B B} = g_{WL_B} \cdot g_{L_B B} \tag{6.9}$$

where $g_{L_B B}$ is the transformation matrix from balloon object to the local coordinates of balloon which is defined by user in the project file. The transformation matrix from the local coordinates of balloon to the world coordinates (g_{WL_C}) can get from the connector between the catheter and balloon. Figure 6.11 shows the balloon catheter in GiPSi simulator. The left side figures show the objects drawn with wireframe and the right side figures show the objects drawn with texture.

6.5.3 Camera Attached Haptic Device

The camera acts as the eye of surgeon to view the working area of surgery. The camera is inserted into the tube of endoscope to the end of endoscope, normally at the tip of endoscope; therefore, the position of attached camera locates at the tip of endoscope. When the haptic device moves, the viewing of camera moves with the same amount of the movement of the haptic interface. The transformation matrix from viewing of camera to world coordinates (g_{WV}) is used in the simulation by equation:

$$g_{WV} = g_{WL_H} \cdot g_{L_H H} \cdot g_{HV} \tag{6.10}$$

where g_{HV} is the transformation matrix from viewing of camera to haptic interface which is defined by user in the project file in `<cameraToHapticTransformation>` sub-element in `<camera>` element and g_{WL_H} is also defined by user in the project file in `<baseToWorldTransformation>` sub-element in `<camera>` element.

6.5.4 Light Attached Camera

The light illuminates the area of surgery which attaches with camera. The transformation matrix from light source to world coordinates (g_{WL}) equals to the transformation matrix from viewing of camera to world coordinates (g_{WV}) as the equation:

$$g_{WL} = g_{WV} \tag{6.11}$$

6.6 Poking a Hole

When an endoscopic instrument reaches the third ventricle, the floor of the third ventricle is opened by a catheter and enlarged the hole with balloon inflation. The movement of endoscopic instrument is captured by haptic device and the movement of catheter and balloon are captured by the mouse. The ventricle floor is modeled by lumped element model with elastoplasticity. The endoscopic instrument, catheter and

balloon movement interacts with the ventricle floor. The collision detection detects the overlap parts among simulation objects and the collision response with penetration depth approximation method resolves the collided object by separating parts of collided area. The hole at the third ventricle floor occurs when the catheter contacts the third ventricle floor hard enough to break the tissue strength. We designed and implemented the algorithm for creating hole by breaking spring.

The basic idea of this algorithm comes from the nature of how tissue is broken when the external force is large enough to break the tissue structure. The haptic device captures the movement of the user controlling endoscopic instrument. When the tip of catheter contacts the tissue at the third ventricle floor, a linearized low-order approximation model is created at the contact node. This model generates the force feedback to the haptic device. Springs with elastoplasticity can change their properties depending on the how large of a force is acting on them. This force is compared with the threshold value. If the spring force reaches the breaking point, then spring is broken and removed from the simulation. The `Hole` class is created and associated in `MSDObject`. `Hole` stores the contact node index, contact node force. When the haptic device contacts the node in lumped element model and the force feedback is large enough to make a hole. The node index and the force are stored in `Hole`. `Hole` is updated in `Display()` function with removing the broken spring as well as broken geometry. The implementation and the result shows in Figure 6.12. The left side figures show the objects drawn with wireframe and the right side figures show the objects drawn with texture.

This algorithm needs the neighbor connectivity information tables which are provided by the `ExtendedTriSurface` data structure. The neighbor connectivity information tables are created after the simulator loads a geometry model from file. The extended geometry creates a vertex-to-face and edge-to-face tables. When the user move the haptic device and touch a node at the third ventricle floor, the user can

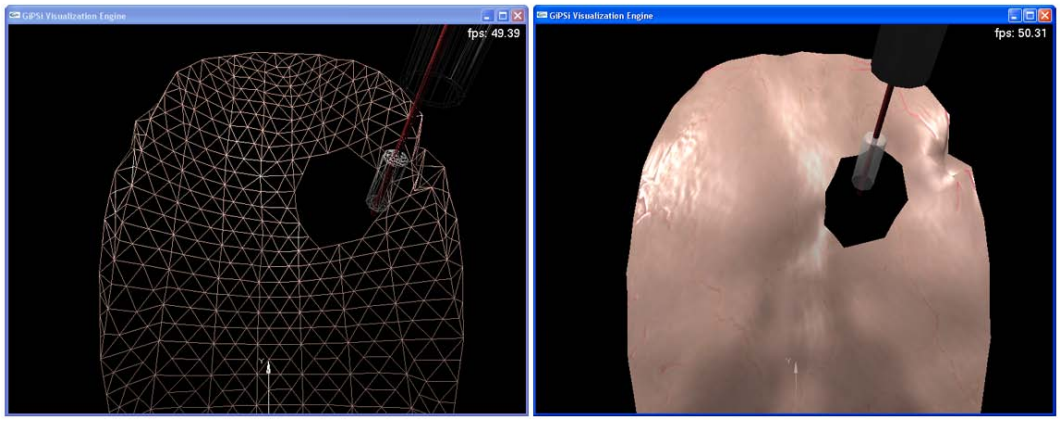


Figure 6.12: Result of hole poking by breaking spring algorithm.

touch the node harder until the length of spring connected with that node exceeds the limitation of elastoplasticity. The springs are broken by setting a spring rest length to zero and set `broken` flag to true. At this point, the vertices related to nodes belonged to broken spring can retrieve from mapping function. After the two vertices are retrieved, the edge is identified along with the faces that belong to that edge. The faces are then removed from the geometry. For simplicity and efficiency, the vertices and faces in `ExtendedTriSurface` are not physically deleted, but instead a flag is set to exclude them during processing.

6.7 Experiment and Discussion

The endoscopic third ventriculostomy simulator was implemented by using GiPSi/GiPSiNet framework and tested on the Microsoft Windows XP^{TM} 32-bit based workstation with Intel Pentium D 2.80 GHz, 1 GB of RAM and a PCI Express NVidia GeForce 8800 Ultra Graphics Card with 768 MB of memory to collect performance benchmarks. The hole poking by breaking spring algorithm was implemented and integrated into the simulator. The simulator consisted of 22 simulation objects, 13 texture objects and 3 connectors. The ventricle system model was composed of

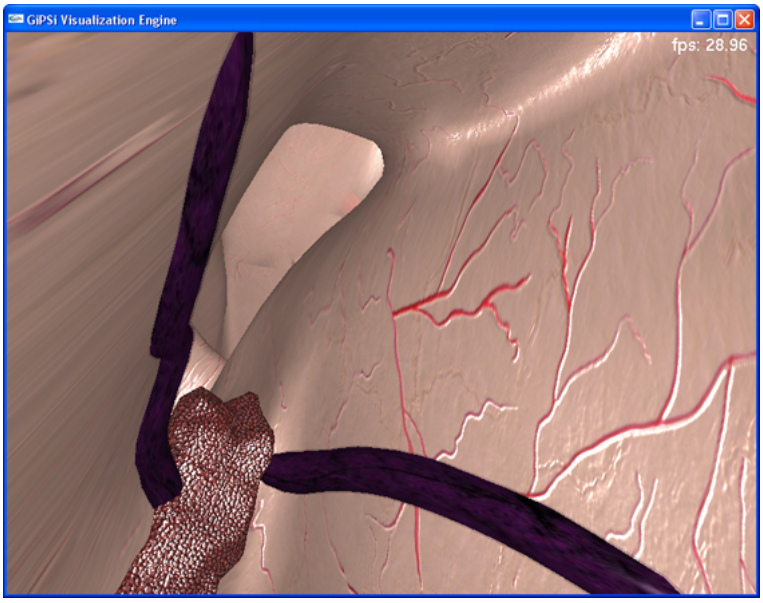


Figure 6.13: Result of endoscopic third ventriculostomy simulator viewing at intraventricular between lateral ventricle and the choroid plexus.

9,145 nodes and 17,942 faces with quasi-static decoupled spring model. The ventricle floor model was composed of 558 nodes and 1,050 faces with lumped element model of 558 masses and 1,607 springs. The implicit Euler numerical integration method with simulation time step of 0.01 second was used for lumped element model in simulation. The simulator operated at about 27 frames per second. The visualization results were shown in Figures 6.13, 6.14, 6.15, and 6.16.

6.8 Conclusion

We developed the endoscopic third ventriculostomy simulator as a test bed of surgery simulator by using GiPSi/GiPSiNet framework. The third ventriculostomy procedure is the treatment of hydrocephalus, which needs the surgeon to train and get familiar with endoscopic surgery. The computer-based simulator is a new and excited training method. Since the GiPSi is a general interactive physical simulation interface, which provides only the core of simulator, the specific endoscopic simulator applica-

Figure 6.14: Result of endoscopic third ventriculostomy simulator viewing at third ventricle floor and the mamillary bodies are the posterior landmark of the third ventricle.

tion needs to implement specific components into the GiPSi, such as, springs with elastoplasticity, specific tools, such as, an endoscope with a balloon catheter, and related connectors between endoscope, catheter and balloon, and hole poking procedures. The result shown that the GiPSi/GiPSiNet framework enabled the application developer to build the specific simulator with speed and easiness.

To produce the realistic look of the hole, the implementation of hole by patching area algorithm needs to be done. Using GPU also enables a realistic look with speed up of the simulator. In order to complete the third ventriculostomy simulator for use in actual training, a graphic user interface, specific training scenario, methods for measuring performance metrics, and a database for storing the training outcomes also need to be designed and implemented.

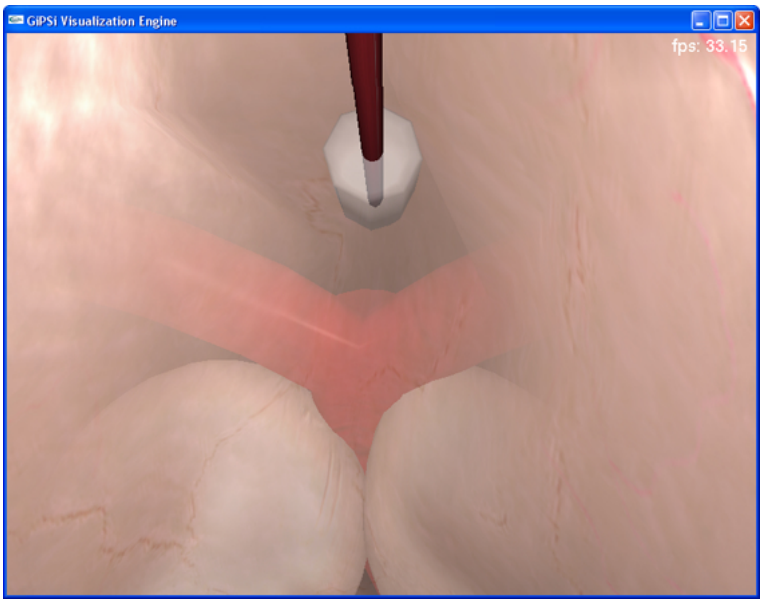


Figure 6.15: Result of endoscopic third ventriculostomy simulator viewing at the third ventricle floor with balloon catheter performing the hole poking.

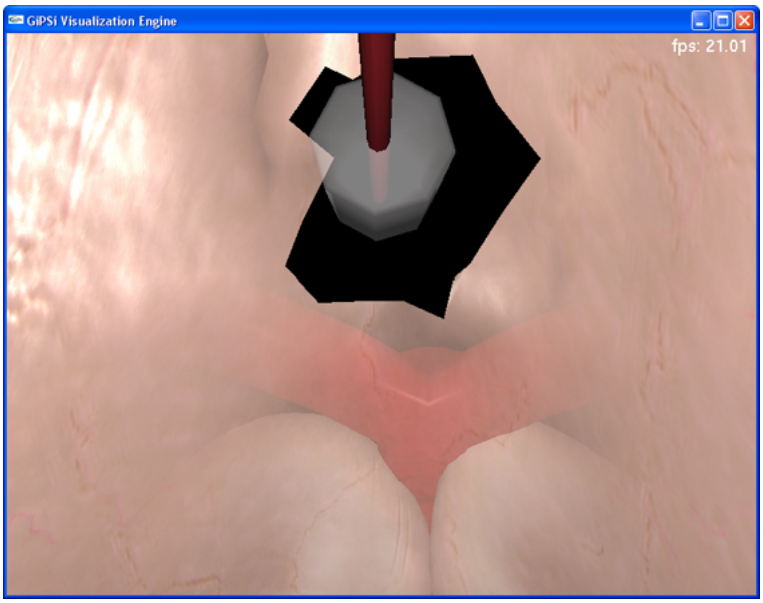


Figure 6.16: Result of endoscopic third ventriculostomy simulator viewing at the third ventricle floor with balloon catheter and the hole.

Chapter 7

Conclusion

This chapter concludes the dissertation. The first section summaries the main research results and contributions of this thesis, followed by a discussion of possible future research related to this thesis.

7.1 Conclusion

The main contributions of this thesis are distributed into five primary areas. The first is a novel method to determine the elasticity parameters in lumped element models by approximating elements of lumped element model with elements of finite element models. Lumped element models have advantages over the highly accurate finite element model in terms of computational complexity. However, there are no well-established methods to determine the elasticity parameters of lumped element model for a specific deformable object. Here, we propose the method where the elasticity parameters are determined through an optimization that minimizes the matrix norm of the error between the stiffness matrices of the lumped element model with a corresponding finite element model. The resulting deformable object models using lumped element model are shown to have behavior very close to finite element models.

The second is as in depth study and comparison of numerical integration algorithms used in simulation to identify underlying trade-offs as a function of material properties. The results of this study can be used to determine a suitable numerical integration algorithms and simulation time step size to be used in simulations to maintain system stability.

The third is the design and implementation of collision detection and response algorithms for deformable objects used in the GiPSi framework. The design of collision detection and response algorithms are based on hierarchical axis aligned bounding boxes and a penetration depth approximation method, respectively. The proposed algorithms are suitable for deformable objects. The developed algorithms are incorporated into the GiPSi framework and employed in the developed test bed surgical simulation.

The forth area is the improvement the design of the GiPSi simulation framework architecture and the extension of the GiPSi framework to networked operation. As parts of this research, several improvements to the GiPSi framework are developed for improving the functionality of GiPSi framework including: the addition of a real-time simulation kernel, ability to explicitly specify object order of execution, addition of implicit numerical integration algorithms, extension of geometry data structures, multi-display manager, development of collision detection and response algorithms, implementation of haptic interaction algorithm which uses linearized low-order approximations, and the addition of a user interface. The network extension of GiPSi (GiPSiNet) to provide remote operation of GiPSi over the Internet is also implemented in collaboration with Prof.Vincenzo Liberatore and Dr.Qingbo Cai.

The fifth area is the development of a prototype endoscopic third ventriculostomy simulator as a test bed using the GiPSi/GiPSiNet framework. The resulting of the test bed application shows the capability of GiPSi/GiPSiNet framework for developing the specific simulator application.

7.2 Future Work

The following are the possible future directions of research related to this study. The elasticity parameters determination for lumped element model can be more precisely determined by adding more spring parameters, such as angular springs to the lumped element model. Otherwise, the use of different optimization methods can produce the difference elasticity parameters which may be suitable for specific deformable objects or applications.

In numerical integration algorithms for deformable object, the study and implementation of new numerical integration algorithms, such as Newmark and Simplectic methods, may increase the performance and stability of the system. Moreover, the new physical based models can also improve physically based models, such as the finite element model, mesh free model or point associated finite field (PAFF) [92].

The collision detection and response algorithms use a large portion of the computation time in the simulation kernel. We can improve the computation time by using the compute unified device architecture (CUDA) developed by NVidia which performs computing on GPU. We can also use CUDA to improve performance of other computations by implementing a lumped element method, as well as implementing numerical integration algorithms. Moreover, the implementation of finite element model in CUDA can speed the application to real-time system.

The endoscopic third ventriculostomy simulator has many possible future research directions. The hole puncturing in the simulator can be designed and implemented by using other alternative approaches, which may produce better visualization of the hole. For example, hole by patching area algorithm places a virtual hole at the contact point with a specified radius and using subdivision to generate additional geometry around the hole and attach into the original physical model. Moreover, the complete simulator including the graphic user interface, database, and training sets can be developed to be an usable application for surgeon.

Another interesting topic is an augmented reality technology which is an active research area. The augmented reality can apply on surgical simulation by producing augmented video on top of the tangible physical surfaces or artificial tissues interact with the real surgical instruments. This technique should produce the realistic look over the augmented video and realistic feel from artificial tissue.

Appendix A

GiPSiNet Visualization and Haptics

A.1 GiPSiNet Design

GiPSiNet is in collaboration with Prof.Liberatore and Dr.Cai from network research laboratory at Case Western Reserve University. The GiPSiNet visualization and haptic design originate by Dr.Cai and Prof.Liberatore. The detail of GiPSiNet is in [2].

A.1.1 GiPSiNet Visualization

Visualization in surgical simulation is an output to screen display representing the physical simulation objects and their interactions that communicates to user via the visual perception. Especially, in minimally-invasive surgery using endoscopic device which the procedures in the real world, surgeon interacts with an image on a screen. The key requirement is the development of an API such that the actual mechanics of the display specific to a given visualization library are transparent to the model developer. Therefore, the API needs to separate the specifics of what needs to be

displayed, which is determined by the model developer, from the specifics of how the actual display happens.

Visualization in GiPSi is based on a stream of frames. Each frame represents a snapshot of the video. The video is displayed by rendering the continuous sequence of frames in a stream. Frames are currently represented with a format specific to GiPSi, and have variable size, i.e., different frames in a stream can have different length. GiPSi frames are composed of frame elements. Frame elements belong to simulation objects, and each simulation object can have 0, 1, or more frame elements. Each object can dynamically change the size and number of its frame elements. The GiPSi implementation breaks up the storage of each frame into the frame elements that belong to each object. In the basic GiPSi concept, the simulation kernel (SK) generates the frame stream, which is consumed by visualization engine (VE) for rendering as shown in Figure A.1a. In GiPSi, the SK and VE are two separate threads of control (e.g., threads or processes) that communicate the stream over shared memory. The rendering engine is currently implemented with OpenGL. The SK constructs each frame by iterating through the simulation objects and generating their frame elements. The VE implicitly reconstructs each frame by iterating through the frame elements that belong to the simulation objects. The SK can produce frames at a higher or lower rate than the rate at which the VE can render them. Hence, a buffer is introduced between the SK and the VE to match the different processing speeds as shown in Figure A.1b. The buffer is lossy in the sense that the SK can generate frames that never reach the VE. In effect, buffer losses are used to match the slower VE with a faster SK. The buffer consists of an input buffer, a queue, and an output buffer as shown in Figure A.1c.

The SK repeatedly writes a frame in the input buffer, which is then enqueued. The VE repeatedly copies the queue head-of-line into the output buffer and renders its contents. In GiPSi, the queue has length one. Its drop policy is "drop-head": if the

queue is full and a new frame arrives, the previously enqueued frame is dropped. The motivation of the drop-head policy is that newer frames should be rendered rather than older ones. The buffer management requires repeated copies of the input buffers into the queue and of the queue head into the output buffer. Frame copying is fast in shared memory since it only requires pointer management and locking (triple-buffer implementation). Since frames can have different lengths, the buffer elements might have to be resized over the course of the execution.

The objective of remote visualization is to implement the SK and VE on different machines with no changes to their code. In particular, the SK and VE must have the appearance of communicating with each other through a shared memory buffer as discussed above. Remote visualization is accomplished with an engine that matches the output buffer at the SK side with the input buffer at the VE side.

The engine consists of a SK-side thread of control called the visualization engine proxy (VEP) and a VE-side thread of control called the simulation kernel proxy (SKP). The VEP (SKP) is intended to interface with the SK (VE) exactly as the VE (SK) would. In particular, the VEP communicate to the SK via shared memory and copies frames out of the queue via locking and pointer management. Analogously, the SKP communicates with the VE via shared memory. The VEP repeatedly dequeues a frame into the SK-side output buffer and sends it to the SKP. The SKP repeatedly fills the VE-side input buffer with received frames and enqueues them. The format of frame elements is a GiPSi API standard, and can be converted into IDL. The VEP and SKP implementation must deal correctly with variable-size frames.

Although in non-distributed visualization, the rendering speed is limited by the VE throughput, in distributed visualization the visualization speed is determined by the minimum of VE throughput and network goodput. Regardless, the SK-side buffer can match a higher-speed SK with a slower (networked) visualization system.

The main issues that arise in remote visualization is transport reliability. In

a reliable transport, the frame is copied exactly and reliably across sites. In an unreliable transport, the frame can be lost. In an unreliable transport, the SKP should drop a whole frame even if only a part was lost. Moreover, frame losses should not jeopardize rendering correctness.

Although correctness can be preserved even with an unreliable transport, performance could differ significantly. Furthermore, it may be advantageous to implement an unreliable transport that makes multiple attempts at receiving a frame. Transport performance is a major open issue, and hence to be postponed to future cycles.

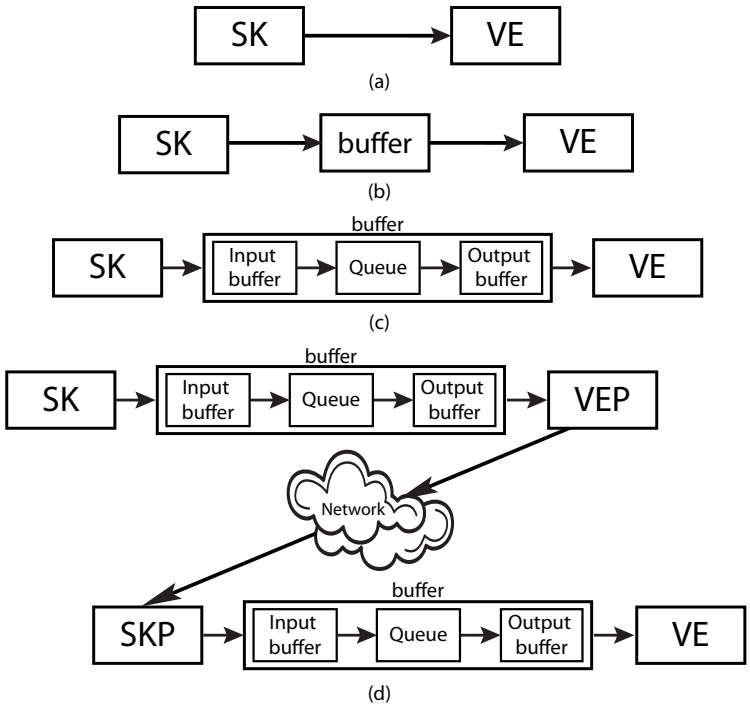


Figure A.1: Visualization in GiPSi (a,b,c) and remote visualization (d).

A.1.2 GiPSiNet Haptics

GiPSi uses a multi-rate simulation scheme proposed by Çavuşoğlu and Tendick in [85]. In this method, each simulation object in haptic interaction provides local dynamic and geometric models for the haptic interface. The local dynamic model is a low-order linear approximation of the full deformable object model, constructed

by the simulation object from the full model at its update intervals, and the local geometric model is a planar approximation of the local geometry of the simulation object at the haptic interfacing location. These local models are used by the haptic interface, running at a significantly higher update rate than the dynamic simulations, for estimating the inter-sample interaction forces and inter-sample collisions. The haptics in GiPSi is based on a closed loop control of a haptic interface: the simulation kernel (the controller) repeatedly queries the current configuration (i.e., position, orientation, stylus button state) of the haptic interface, and generates force feedback to apply to the haptic interface. In current GiPSi, the haptic Interface (HI) is managed by a haptic manager (HM) thread as shown in Figure A.2a, which in turn is launched by the project loader. The actual haptic interface is represented by haptic interface object (HIO) in the simulation, and the HIOs are created and maintained by the simulation kernel (SK); the HM instantiates HIOs and attaches them to the correct HIs according to the XML data in project file. The HM and SK threads communicate with each other using the process owned memory (variables).

The objective of remote haptics is to implement the SK and HM on different machines with minimum changes to GiPSi code as shown in Figure A.2b. In GiPSiNet, the SK-HM inter-process communications done by method invocations on remote objects. For example, the HI proxy is implemented and called by the SK to delegate all operations on the HI, which is managed by the HM and is a remote object to the SK. Meanwhile, to incur minimum code change to extend GiPSi to GiPSiNet, the HI proxy (HIP) is implemented as the derived class of HI, and consequently the HIP can be used where the HI is used in GiPSi. Moreover, the HIP encapsulates the details of invocations on remote objects. Therefore, the SK can invoke methods on remote HI objects via HIP objects as if the HIs were local objects. The haptic attachment occurs in simulation kernel by using the HM. The HM gets a haptic interface pointer through a device identifier and attaches to the HIO. In GiPSiNet, four proxy classes

are added: HI proxy (HIP), PhantomHI proxy (PhantomHIP), HM proxy (HMP), and OpenHM proxy (OpenHMP). The method invocation mechanism on remote objects is a typical use of CORBA/TAO, and consequently the main effort to implement this class is to do the argument transformations between the GiPSi types and TAO types so that the TAO implementation is encapsulated in GiPSiNet and is hidden from GiPSi.

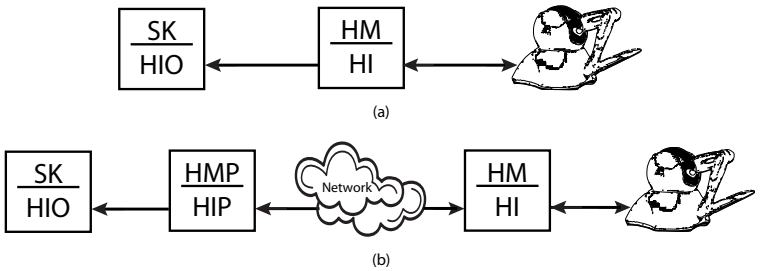


Figure A.2: Haptics in GiPSi (a) and remote haptics (b).

A.2 GiPSiNet Bandwidth

This section shows the bandwidth of data transfer between the client and the server. The data is captured into four locations which are visualization engine proxy, haptic manager proxy, user interface command, project file transfer.

A.2.1 GiPSiNet Visualization Bandwidth

The bandwidth of visualization data is calculated by the summation of `dA_size` and `iA_size` of all geometry objects and the summation of height*width of all textures. The visualization bandwidth includes the size of geometry and texture shown in Tables A.1 and A.2 for endoscopic neurosurgery training simulator. Tables A.3 and A.4 are geometry and texture data for simple test bed. The number of vertex can calculated from `dA_size` divided by `DISPARRAY_NODESIZE` which is 15 expect the

haptic geometry is divided by 12. The number of face can calculated from `iA_size` divided by 3 for triangle face.

Table A.1: Geometry data in endoscopic simulator

Geometry Object	dA_size	iA_size	Total (byte)
ANOTHER_VEIN	6,330	2,520	35,400
ANOTHER_VEIN_DISPLACED	6,330	2,520	35,400
ANTERIOR_SEPTAL_VEIN	6,330	2,520	35,400
ANTERIOR_SEPTAL_VEIN_DISPLACED	6,330	2,520	35,400
BASILAR_ARTERY	4,530	1,800	25,320
BLACK_COVER	3,240	1,284	18,096
CHOROID_PLEXUS	90,000	35,988	503,952
Haptic	1,212	579	7,164
LEFT_ANOTHER_VEIN	6,330	2,520	35,400
LEFT_ANOTHER_VEIN_DISPLACED	6,330	2,520	35,400
LEFT_ANTERIOR_SEPTAL_VEIN	6,330	2,520	35,400
LEFT_ANTERIOR_SEPTAL_VEIN_DISPLACED	6,330	2,520	35,400
LEFT_CHOROID_PLEXUS	90,000	35,988	503,952
LEFT_THALAMOSTRIATE_VEIN	6,330	2,520	35,400
LEFT_THALAMOSTRIATE_VEIN_DISPLACED	6,330	2,520	35,400
MAMMILLARY_BODIES	14,460	5,760	80,880
THALAMOSTRIATE_VEIN	6,330	2,520	35,400
THALAMOSTRIATE_VEIN_DISPLACED	6,330	2,520	35,400
VENTRICLE_FLOOR	3,495	858	17,412
VENTRICLE_SYSTEM	137,175	53,826	764,004

Table A.2: Texture data in endoscopic simulator

Texture data	Height	Width	Total(byte)
AnteriorSeptalVeinBase	192	192	147,456
AnteriorSeptalVeinHeightMap	192	192	147,456
BasilarArteryBase	192	192	147,456
BasilarArteryHeightMap	192	192	147,456
CatheterBase	256	256	262,144
ChoroidPlexusBase	256	256	262,144
ChoroidPlexusHeightMap	256	256	262,144
MammillarryBodiesBase	256	256	262,144
MammillarryBodiesHeightMap	256	256	262,144
VentricleFloorBase	512	512	1,048,576
VentricleFloorHeightMap	512	512	1,048,576
VentricleSystemBase	512	512	1,048,576
VentricleSystemHeightMap	512	512	1,048,576

A.2.2 GiPSiNet Haptic Bandwidth

`GiPSiLowOrderLinearHapticModel` structure stores a linearized low-order approximation as the following:

Table A.3: Geometry data in simple test bed

Geometry data	dA_size	iA_size	Total (byte)
Haptic	312	144	1,824
SHEET1	1,815	600	9,660
SHEET2	1,815	600	9,660

Table A.4: Texture data in simple test bed

Texture data	Height	Width	Total(byte)
BlueBase	512	512	262,144
OrganBase	256	256	65,536
OrganHeight	256	256	65,536
TissueBase	256	256	65,536
TissueHeight	256	256	65,536

```
typedef struct {
  unsigned int _n;          // number of states
  unsigned int _m;          // number of inputs
  unsigned int _k;          // number of outputs
  Matrix<Real> *A11;        // _n/2 x _n/2 matrix
  Matrix<Real> *A12;        // _n/2 x _n/2 matrix
  Matrix<Real> *B1;         // _n/2 x _m matrix
  Matrix<Real> *C11;        // _k x _n/2 matrix
  Matrix<Real> *C12;        // _k x _n/2 matrix
  Matrix<Real> *D;          // _k x _m matrix
  Vector<Real> *f_0;        // _k x 1 vector
  Vector<Real> *zdot_0;     // _n x 1 vector
  Vector<Real> *normal;     // _k x 1 vector
} GiPSiLowOrderLinearHapticModel;
```

The bandwidth of linearized low-order approximation model is calculated by summation of all data type in structure. Assuming an `unsigned int` is 4 bytes and `Real` is 8 bytes. The `Matrix` and `Vector` are approximated only by the size of matrix and vector with no other overhead.

$$bandwidth_{haptic} = 12 + 8\left(\frac{n^2}{2} + \frac{nm}{2} + km + kn + 2k + n\right)$$

where n is the number of states, m is the number of inputs, and k is the number of outputs of linearized low-order approximation. Tables A.5 and A.6 are linearized

low-order approximation model data for endoscopic simulator and simple test bed, respectively.

Table A.5: A linearized low-order approximation model in endoscopic simulator

n	m	k	Data(byte)
0	6	3	204
6	6	3	684
48	6	3	12,108

Table A.6: A linearized low-order approximation model in simple test bed

n	m	k	Data(byte)
0	6	3	204
6	6	3	684
30	6	3	5,484
48	6	3	12,108

The `ReadConfiguration` function transfers the state of haptic device from haptic manager in the client to haptic manager proxy in the server. The data transfer contains

```
double       Position[3];
double       Orientation[9];
unsigned int ButtonState;
```

The data transfer size is 24+72+4=100 bytes.

Bibliography

[1] T. Klemensø, E. Lund, and B. F. Sørensen, "Optimal shape of thin tensile test specimen," *Journal of the American Ceramic Society*, vol. 90, pp. 1827–1835, 2007.

[2] V. Liberatore, Q. Cai, and M. C. Çavuşoğlu, "GiPSiNet: An open source/open architecture network middleware for surgical simulations," in *Medicine Meets Virtual Reality 14: Accelerating Change in Healthcare: Next Medical Toolkit*, 2006, pp. 316–321.

[3] "Spina bifida and hydrocephalus explained." [Online]. Available: http://www.asbha.org.au/SpinaBifidaandHydrocephalusExplained.htm

[4] A. Farina, H. E. Aryanb, B. M. Ozgurb, A. T. Parsad, and M. L. Levy, "Endoscopic third ventriculostomy," *Journal of Clinical Neuroscience*, vol. 13, no. 7, pp. 763–770, 2006.

[5] R. Kneebone, "Simulation in surgical training: educational issues and practical implications," *Medical Education*, vol. 37, no. 3, pp. 267–277, 2003.

[6] K. Montgomery, C. Bruyns, J. Brown, S. Sorkin, F. Mazzella, G. Thonier, A. Tellier, B. Lerman, and A. Menon, "Spring: A general framework for collaborative, real-time surgical simulation," in *Medicine Meets Virtual Reality (MMVR 2002)*. IOS Press, 2002, pp. 23–26.

[7] A. Liu, F. Tendick, K. Cleary, and C. Kaufmann, "A survey of surgical simulation: Applications, technology, and education," *Presence: Teleoperators & Virtual Environments*, vol. 12, no. 6, pp. 599–614, 2003.

[8] M. C. Çavuşoğlu, T. Goktekin, and F. Tendick, "GiPSi: A framework for open source/open architecture software development for organ level surgical simulation," *IEEE Transactions on Information Technology in Biomedicine*, vol. 10, no. 2, pp. 312–321, 2006.

[9] P. Lamata, E. Gómez, F. Bello, R. Kneebone, R. Aggarwal, and F. Lamata, "Conceptual framework for laparoscopic VR simulators," *IEEE Computer Graphics and Applications*, vol. 26, no. 6, pp. 69–79, 2006.

[10] J. Allard, S. Cotin, F. Faure, P. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. Grisoni, "SOFA - an open source framework for medical simulation," in *Medicine Meets Virtual Reality (MMVR 2007)*, 2007, pp. 13–18.

[11] P. Lamata, E. J. Gómez, F. M. Sánchez-Margallo, í. López, C. Monserrat, V. García, C. Alberola, M. A. R. Florido, J. Ruiz, and J. Usón, "SINERGIA laparoscopic virtual reality simulator: Didactic design and technical development," *Computer Methods and Programs in Biomedicine*, vol. 85, no. 3, pp. 273–283, 2007.

[12] G. Lacey, D. Ryan, D. Cassidy, and D. Young, "Mixed-reality simulation of minimally invasive surgeries," *IEEE Multimedia*, vol. 14, no. 4, pp. 76–87, 2007.

[13] Y. C. Fung, *Biomechanics: mechanical properties of living tissues.* New York, NY, USA: Springer-Verlag, 1993.

[14] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically deformable models," in *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 205–214.

[15] D. N. Metaxas, *Physics-Based Deformable Models: Applications to Computer Vision, Graphics, and Medical Imaging.* Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[16] X. Provot, "Deformation constraints in a mass-spring model to describe rigid cloth behavior," in *Proceedings of Graphics Interface*, 1995, pp. 147–154.

[17] D. Bourguignon and M. Cani, "Controlling anisotropy in mass-spring systems," in *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*, ser. Springer Computer Science, aug 2000, pp. 113–123.

[18] M. Bro-Nielsen, "Finite element modeling in surgery simulation," *Proceedings of the IEEE*, vol. 86, no. 3, pp. 490–503, 1998.

[19] D. L. James and D. K. Pai, "ArtDefo: Accurate real time deformable objects," in *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 1999, pp. 65–72.

[20] S. Cotin, H. Delingette, and N. Ayache, "Real-time elastic deformations of soft tissues for surgery simulation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 1, pp. 62–73, 1999.

[21] Y. Zhuang and J. Canny, "Haptic interaction with global deformations," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2000)*, 2000, pp. 2428–2433.

[22] H. Delingette, S. Cotin, and N. Ayache, "Efficient linear elastic models of soft tissues for real-time surgery simulation," in *Studies in Health Technology and Informatics*, vol. 62, 1999, pp. 100–101.

[23] K. D. Costa, P. J. Hunter, J. M. Rogers, J. M. Guccione, L. K. Waldman, and A. D. McCulloch, "A three-dimensional finite element method for large elastic

deformations of ventricular myocardium: Part I - Cylindrical and spherical polar coordinates," *ASME Journal of Biomechanical Engineering*, vol. 118, pp. 452–463, 1996.

[24] ——, "A three-dimensional finite element method for large elastic deformations of ventricular myocardium: Part II - Prolate spherical coordinates," *ASME Journal of Biomechanical Engineering*, vol. 118, pp. 464–472, 1996.

[25] X. Wu, M. S. Downes, T. Goktekin, and F. Tendick, "Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes," in *Proceedings of the Eurographics*, vol. 20, no. 3, 2001, pp. 349–358.

[26] X. Wu and F. Tendick, "Multigrid integration for interactive deformable body simulation," in *Procedding of International Symposium on Medical Simulation (ISMS 2004)*, ser. Lecture Notes in Computer Science, vol. 3078, 2004, pp. 92–104.

[27] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler, "Stable real-time deformations," in *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. New York, NY, USA: ACM, 2002, pp. 49–54.

[28] M. Müller and M. Gross, "Interactive virtual materials," in *GI '04: Proceedings of Graphics Interface 2004*. Canadian Human-Computer Communications Society, 2004, pp. 239–246.

[29] M. Nesme, Y. Payan, and F. Faure, "Efficient, physically plausible finite elements," in *Eurographics (short papers)*, 2005, pp. 77–80.

[30] G. Irving, J. Teran, and R. Fedkiw, "Invertible finite elements for robust simulation of large deformation," in *SCA '04: Proceedings of the 2004 ACM*

SIGGRAPH/Eurographics symposium on Computer animation. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 131–140.

[31] F. Conti, "Tissue modeling via space filling elastic spheres," 2001, presented at the Stanford Workshop on Surgical Simulation.

[32] T. W. Sederberg and S. R. Parry, "Free-form deformation of solid geometric models," in *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, 1986, pp. 151–160.

[33] B. Barsky, *Computer Graphics and Geometric Modeling Using Beta-Splines*. New York: Springer-Verlag, 1988.

[34] A. Joukhadar, F. Garat, and C. Laugier, "Parameter identification for dynamic simulation," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '97)*, 1997, pp. 1928–1933.

[35] A. Joukhadar and C. Laugier, "Dynamic simulation: Model, basic algorithms, and optimization," in *Algorithms For Robotic Motion and Manipulation*, 1997, pp. 419–434.

[36] G. Bianchi, M. Harders, and G. Székely, "Mesh topology identification for mass-spring models," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI 2003)*, ser. Lecture Notes in Computer Science, vol. 2878, 2003, pp. 50–58.

[37] G. Bianchi, B. Solenthaler, G. Székely, and M. Harders, "Simultaneous topology and stiffness identification for mass-spring models based on FEM reference deformations," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI 2004)*, vol. 2, 2004, pp. 293–301.

[38] O. Deussen, L. Kobbelt, and P. Tucke, "Using simulated annealing to obtain good nodal approximations of deformable bodies," in *In Sixth Eurographics Workshop on Simulation and Animation*, 1995, pp. 30–43.

[39] A. V. Gelder, "Approximate simulation of elastic membranes by triangulated spring meshes," *Journal of Graphics Tools*, vol. 3, no. 2, pp. 21–42, 1998.

[40] A. Maciel, R. Boulic, and D. Thalmann, "Deformable tissue parameterized by properties of real biological tissue," in *Surgery Simulation and Soft Tissue Modeling*, ser. Lecture Notes in Computer Science, vol. 2673, 2003, pp. 74–87.

[41] V. Baudet, M. Beuve, F. Jaillet, B. Shariat, and F. Zara, "New mass-spring system integrating elasticity parameters in 2D," LIRIS, Universit Lyon 1, Tech. Rep. RR-LIRIS-2007-003, February 2007.

[42] ——, "Integrating tensile parameters in 3D mass-spring system," LIRIS, Universit Lyon 1, Tech. Rep. RR-LIRIS-2007-004, February 2007.

[43] M. C. Çavuşoğlu, "Telesurgery and surgical simulation: Design, modeling, and evaluation of haptic interfaces to real and virtual surgical environments," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2000.

[44] B. A. Lloyd, G. Székely, and M. Harders, "Identification of spring parameters for deformable object simulation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 5, pp. 1081–1094, 2007.

[45] X. Wang and V. Devarajan, "1D and 2D structured mass-spring models with preload," *The Visual Computer*, vol. 21, no. 7, pp. 429–448, 2005.

[46] ——, "Improved 2D mass-spring-damper model with unstructured triangular meshes," *The Visual Computer*, vol. 24, no. 1, pp. 57–75, 2007.

[47] M. E. Gurtin, *An Introduction to Continuum Mechanics*. New York, NY, USA: Academic Press, 1981.

[48] ——, *Topics in Finite Elasticity*. Philadelphia, PA, USA: Society for Industrial and Applied Mechanics, 1981.

[49] J. E. Marsden and T. J. R. Hughes, *Mathematical foundations of elasticity*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1983.

[50] J. N. Reddy, *An Introduction to the Finite Element Method*, 2nd ed. McGraw-Hill, Inc., 1993.

[51] K. Bathe, *Finite Element Procedures*. Englewood Cliffs, NJ, USA: Prentice Hall, Inc., 1996.

[52] J. C. Strikwerda, *Finite difference schemes and partial differential equations*. New York, NY: Chapman & Hall, 1989.

[53] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C : the Art of Scientific Computing*, 2nd ed. Cambridge, NY: Cambridge University Press, 1992.

[54] D. d'Aulignac, C. Laugier, and M. C. Çavuşoğlu, "Towards a realistic echographic simulator with force feedback," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robotics and Systems (IROS '99)*, 1999, pp. 727–732.

[55] P. Volino, M. Courchesne, and N. Magnenat-Thalmann, "Versatile and efficient techniques for simulating cloth and other deformable objects," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 1995, pp. 137–144.

[56] B. Eberhardt, A. Weber, and W. Strasser, "A fast, flexible, particle-system model for cloth draping," *IEEE Computer Graphics and Applications*, vol. 16, no. 5, pp. 52–59, 1996.

[57] D. Baraff and A. Witkin, "Large steps in cloth simulation," in *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998, pp. 43–54.

[58] B. Eberhardt, O. Etzmu, and M. Hauth, "Implicit-explicit schemes for fast animation with particle systems," in *Eurographics Computer Animation and Simulation Workshop*. Springer, 2000, pp. 137–151.

[59] P. Volino and N. Magnenat-Thalmann, "Comparing efficiency of integration methods for cloth simulation," *Computer Graphics International Conference*, vol. 0, p. 0265, 2001.

[60] M. Hauth, "Numerical techniques for cloth simulation," SIGGRAPH 2003 Course Notes, 2003.

[61] A. Nealen, M. Muller, R. Keiser, E. Boxerman, and M. Carlson, "Physically based deformable models in computer graphics," *Computer Graphics Forum*, vol. 25, no. 4, pp. 809–836, 2006.

[62] V. Egorov, S. Tsyuryupa, S. Kanilo, M. Kogit, and A. Sarvazyan, "Soft tissue elastometer," *Medical Engineering & Physics*, vol. 30, no. 2, pp. 206–212, 2008.

[63] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M. P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino, "Collision detection for deformable objects," *Compututer Graphics Forum*, vol. 24, no. 1, pp. 61–81, 2005.

[64] P. Volino and N. Magnenat-Thalmann, *Virtual Clothing Theory and Practice*. Springer-Verlag, 2000.

[65] I. J. Palmer and R. L. Grimsdale, "Collision detection for animation using sphere-trees," *Computer Graphics Forum*, vol. 14, no. 2, pp. 105–116, 1995.

[66] G. van den Bergen, *Collision Detection in Interactive 3D Environments*, ser. The Morgan Kaufmann Series in Interactive 3D Technology. San Francisco, CA: Morgan Kaufmann, 2003.

[67] S. Gottschalk, M. C. Lin, and D. Manocha, "OBBTree: a hierarchical structure for rapid interference detection," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 171–180.

[68] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.

[69] G. van den Bergen, "Efficient collision detection of complex deformable models using AABB trees," *Journal of Graphics Tools*, vol. 2, no. 4, pp. 1–14, 1997.

[70] D. L. James and D. K. Pai, "BD-Tree: Output-sensitive collision detection for reduced deformable models," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 393–398, 2004.

[71] M. Moore and J. Wilhelms, "Collision detection and response for computer animationr3," *ACM SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 289–298, 1988.

[72] D. Baraff, "Analytical methods for dynamic simulation of non-penetrating rigid bodies," in *Proceedings of the 1989 ACM SIGGRAPH conference*, vol. 23, no. 3, 1989, pp. 223–232.

[73] ——, "Curved surfaces and coherence for non-penetrating rigid body simulation," *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 19–28, 1990.

[74] B. Mirtich, "Impulse-based dynamic simulation of rigid body systems," Ph.D. dissertation, University of California, Berkeley, December 1996.

[75] S. A. Cameron and R. K. Culley, "Determining the minimum translational distance between two convex polyhedra," in *Proceedings of International Conference on Robotics and Automation*, 1986, pp. 591–596.

[76] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal of Robotics and Automation*, vol. 4, no. 2, pp. 193–203, 1988.

[77] Y. J. Kim, M. A. Otaduy, M. C. Lin, and D. Manocha, "Fast penetration depth computation for physically-based animation," in *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2002, pp. 23–31.

[78] B. Heidelberger, M. Teschner, R. Keiser, M. M uller, and M. Gross, "Consistent penetration depth estimation for deformable collision response," in *Proceedings of Vision, Modeling, Visualization (VMV 2004)*, 2004, pp. 339–346.

[79] M. Teschner, B. Heidelberger, M. M uller, D. Pomeranets, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects," in *Proceedings of Vision, Modeling, Visualization (VMV 2003)*, 2003, pp. 47–54.

[80] P. Terdiman, "OPCODE: Optimized collision detection," 2003. [Online]. Available: http://www.codercorner.com/Opcode.htm

[81] T. Möller, "A fast triangle-triangle intersection test," *Journal of Graphics Tools*, vol. 2, no. 2, pp. 25–30, 1997.

[82] M. Kawasaki, M. Rissanen, N. Kume, Y. Kuroda, M. Nakao, T. Kuroda, and H. Yoshihara, "VRASS (virtual reality aided simulation)." [Online]. Available: http://www.kuhp.kyoto-u.ac.jp/~{}mi/en/index.php?_research__med_vr

[83] K. T. McDonnell, H. Qin, and R. A. Wlodarczyk, "Virtual clay: a real-time sculpting system with haptic toolkits," in *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, 2001, pp. 179–190.

[84] K. Miyahara, Y. Okada, and K. Niijima, "A cloth design system by intuitive operations," in *CGIV '06: Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation*, 2006, pp. 458–463.

[85] M. C. Çavuşoğlu and F. Tendick, "Multirate simulation for high fidelity haptic interaction with deformable objects in virtual environments," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2000)*, 2000, pp. 2458–2465.

[86] P. Jacobs and M. C. Çavuşoğlu, "High fidelity haptic rendering of stick-slip frictional contact with deformable objects in virtual environments using multi-rate simulation," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2007)*, 2007, pp. 117–123.

[87] D. G. Schmidt and F. Kuhns, "An overview of the real-time CORBA specification," *Computer*, vol. 33, no. 6, pp. 56–63, 2000.

[88] D. C. Schmidt, B. Natarajan, A. Gokhale, and N. W. C. Gill, "TAO: A pattern-oriented object request broker for distributed real-time and embedded systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, 2002.

[89] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[90] V. L. Lespinasse, "Hydrocephalus and spina bifida," *in Lea & Febigers Principles of Neurological Surgery*, pp. 438–447, 1992.

[91] W. J. Mixter, "Ventriculoscopy and puncture of the floor of the third ventricle," *Boston Medical and Surgical Journal*, vol. 188, pp. 277–278, 1923.

[92] S. De, Y. Lim, M. Manivannan, and M. Srinivasan, "Physically realistic virtual surgery using the point-associated finite field (PAFF) approach," *Presence: Teleoperators & Virtual Environments*, vol. 15, no. 3, pp. 294–308, 2006.