GiPSi: An Open Source/Open Architecture Software Development Framework for Surgical Simulation

Tolga G. Göktekin[2], M. Cenk Çavuşoğlu[1]

in collaboration with

Frank Tendick[3], Shankar Sastry[2]

[1] :Dept. of Electrical Eng. and Computer Sci., Case Western Reserve University
cavusoglu@case.edu
[2] :Dept. of Electrical Eng. and Computer Sci., University of California, Berkeley
goktekin,sastry@eecs.berkeley.edu
[3] :Dept. of Surgery, University of California, San Francisco
tendick@eecs.berkeley.edu

March 12, 2004

Abstract

In this paper we propose an open source/open architecture framework for developing organ level surgical simulations. Our goal is to facilitate shared development of reusable models, to accommodate heterogeneous models of computation, and to provide a framework for interfacing multiple heterogeneous models. The framework provides an intuitive API for interfacing dynamic models defined over spatial domains. It is specifically designed to be independent of the specifics of the modeling methods used and therefore facilitates seamless integration of heterogeneous models and processes. Furthermore, each model has separate geometries for visualization, simulation, and interfacing, allowing the model developer to choose the most natural geometric representation for each case. I/O interfaces for visualization and haptics for real-time interactive applications have also been provided.

Chapter 1

Introduction

Computer simulations have become an important tool for medical applications, such as surgical training, pre-operative planning, and biomedical research. However, the current state of the field of medical simulation is characterized by scattered research projects using a variety of models that are neither inter-operable nor independently verifiable models. Individual simulators are frequently built from scratch by individual research groups without input and validation from a larger community. The challenge of developing useful medical simulations is often too great for any individual group since expertise is required from different fields. The motivation behind this study is our prior experience in surgical training simulators and physically based modeling [11, 12].

The open source, open architecture software development model provides an attractive framework to address the needs of interfacing models from multiple research groups and the ability to critically examine and validate quantitative biological simulations. Open source models ensure quality control, evaluation, and peer review, which are critical for basic scientific methodology. Furthermore, since subsequent users of the models and the software code have access to the original code, this also improves the reusability of the models and inter-connectibility of the software modules. On the other hand, an open architecture simulation framework allows open source or proprietary third party development of additional models, model data, and analysis and computation modules.

The key technical issues that need to be addressed in the development of an open source, open architecture simulation framework are:

1. **Abstraction:** In the context of surgical simulation, model abstraction is an important consideration. Within a general modeling and simulation framework, different applications and different problems require different types or levels of abstraction for the each of the processes and components in the model. For example in a heart model, the beating of the heart can be modeled as an electro-chemically activated mechanical process, or it can be modeled as a finite state machine, with each state corresponding to a discrete phase of the heart cycle. These are different types of abstractions for the same process. It is also possible to have different level of abstractions: by modeling the heart muscle contraction starting at the individual cell level, or at tissue level or at the level of the whole organ. In surgical simulations, it is most important to have an accurate model of the mechanical manipulation of the heart. Hence, when model-

ing effects of medications in a heart surgery simulation it may be sufficient to use an abstraction which include the aggregate physiological effects of different medications used during the procedure to the extent they effect the electrical and mechanical activity of the heart, instead of modeling all the processes going on at the cell or tissue level. However, in a simulation to study the effects of an experimental drug, it would be necessary to have a more detailed and accurate model of these processes. Therefore, the simulation framework developed needs to be able to accommodate different types and levels of abstraction for each of the different subcomponents in the model hierarchy, without artificially limiting the possibilities based on the requirements of a specific application.

2. **Heterogeneous physical mechanisms and models of computation:** Another issue that arises with the varying types of abstractions is the requirement on the simulation engine to be able to handle heterogeneous physical mechanisms (e.g. solid mechanics, fluid mechanics and bioelectricity) and models of computation (e.g. differential equations, finite state machines and hybrid systems). The simulator kernel and the application interfaces need to have support for hybrid models of computation, i.e. computation of continuous and discrete deterministic processes, and stochastic processes, which can be used to model basic biological functions.

3. **Interfacing models of different physical processes:** In order to simulate a complex biological system, models of different physical processes, which may even use different models of computation, need to be coupled together. Therefore, it is necessary to develop standard model interfaces in the form of software APIs for interconnection of these models. There are two key aspects of this API: 1) interfacing multiple physical processes at the semantic level, 2) coupling multiple models of computation at the structural level. The semantic level specifies how the different physical quantities are coupled at the interface, including the semantics of the coupling, and the structural level specifies how the interfacing is achieved at the specifics of the individual computational models used.

4. **Modularity through encapsulation and data hiding:** The API and the overall framework needs to be able to support hierarchical models and abstraction of the input-output behavior of individual layers or subsystems for the level of detail desired from the simulation model. The object oriented programming concepts of encapsulation and data hiding facilitates the modularity of the components. This also provides mechanisms to interface and embed the constructed models and other computational modules to a larger, more sophisticated model.

5. **Validation:** Validation of the models and the underlying empirical data is a basic requirement for reusability of the models. It is also important to have mechanisms to track the assumptions of the individual models and model data within a complex simulation environment, to ensure that the aggregate assumptions behind the models and the abstractions satisfy the requirements of the application at hand.

6. **Customization to patient specific physiology:** In surgical planning and preoperational rehearsals, it is necessary to use the patient specific models during simulation.

Therefore the models in the simulation need to be customizable. This actually ties to the open architecture design of the simulation framework. The open architecture approach should allow loading and working with custom data sets generated by third parties.

In this report we describe GiPSi (General Interactive Physical Simulation Interface), an open source/open architecture framework for developing surgical simulations such as interactive surgical training and planning systems. The main goal of this framework is to facilitate shared model development and simulation of organ level processes as well as data sharing among multiple research groups. To address these, we focused on the first four of the technical aspects mentioned above: Model abstraction, support for heterogeneous models of computation, APIs for interfacing various heterogeneous physical processes, and modularity. In addition, I/O interfaces for visualization and haptics for real-time interactive applications have been provided. The implementation of the framework is done using C++ and it is platform independent.

An important difference of GiPSi from earlier object-oriented tools and languages for modeling and simulation of complex physical systems, such as Modelica [8], Matlab Simulink [7], and Ptolemy [3], is its focus on representing and enforcing time dependent spatial relationships between objects, especially in the form of boundary conditions between interfaced and interacting objects. The APIs in GiPSi are also being designed with a special emphasis on being general and independent of the specifics of the implemented modeling methods, unlike earlier dynamic modeling frameworks such as SPRING [9] or AlaDyn-3D [6], where the underlying models used in these physical modeling tools are 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 [5].

1.1 Overview

One of the major goals of GiPSi is to provide a framework that facilitates shared development that would encourage the extensibility of the simulation framework and the generality of the interfaces allowing components built by different groups and individuals to plug together and reused. Therefore, modularity through encapsulation and data hiding between the components should be enforced. In addition, a standard interfacing API facilitating communication among these components needs to be provided.

We are developing our tools on a specific test-bed application: the construction of a heart model for simulation of heart surgery. This test-bed model captures the most important aspects of the general problem we are trying to address: *i)* multiple heterogeneous processes that need to be modeled and interfaced, and *ii)* different levels of abstraction possible for the different processes. In the heart surgery simulation, several different processes, namely physiology, bioelectrical activity, muscle mechanics, and blood dynamics, need to modeled. Physiological processes regulate the bioelectrical activity, which, in turn, drives the mechanical activity of the heart muscle. Muscle dynamics, coupled with the fluid dynamics of the blood, determine the resulting motion of the heart [2]. Models for all these processes need to be intimately coupled: the mechanical and fluid models through

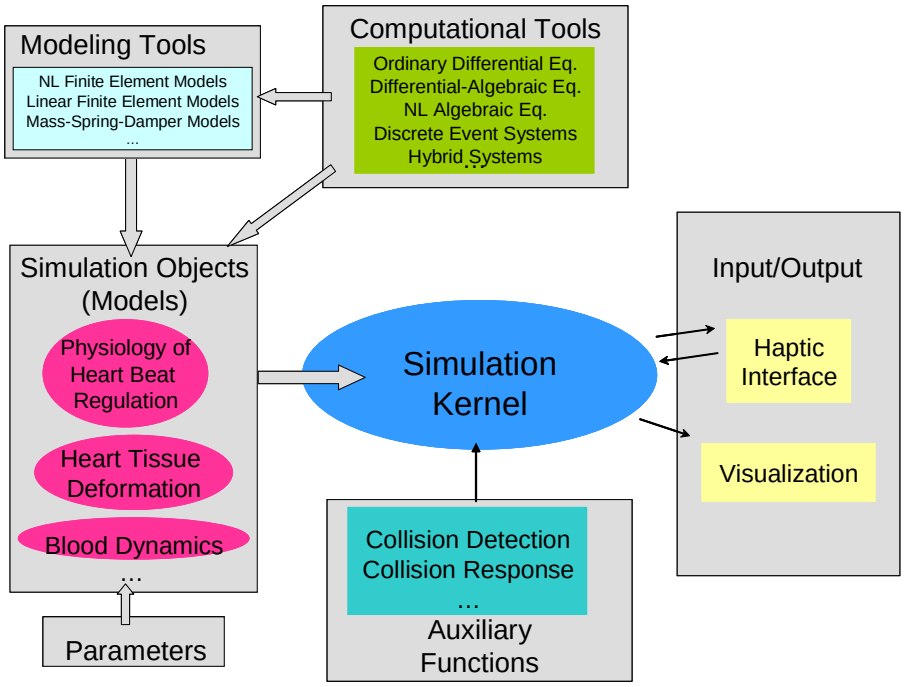

Figure 1.1: The architecture of a GiPSi-based simulation system

a boundary interaction, and the electrochemical and mechanical models through a volume interaction.

The overall system architecture of GiPSi is shown in Fig. 1.1. The models of physical processes such as muscle mechanics of the heart are represented as Simulation Objects (Sect. 2.1). Each simulation object can be derived from a specific computational model contained in Modeling Tools such as finite elements, finite differences, lumped elements etc (Sect. 3.1.2). The Computational Tools provide a library of numerical methods for low level computation of the object's dynamics (Sect. 3.1.1). These tools include explicit/implicit ordinary differential equation (ODE) solvers, linear and nonlinear algebraic system solvers, and linear algebra support. The objects are created and maintained by the Simulation Kernel which arbitrates their communication to other objects and components of the system (Sect. 3.2.2). One such component is the I/O subsystem which provides basic user input provided through the haptic interface tools and basic output through visualization tools (Sect. 2.2). There are also Auxiliary Functions that provide application dependent support to the system such as collision detection and collision response tools that are widely used in interactive applications (Sect. 3.2.1).

It is important to note that GiPSi is intended to be a general software development framework rather than a complete simulation engine. The framework consists of the Simulation Object API, which also includes the object interfacing API, the Visualization API and the Haptics API. The implemented Modeling Tools and Computational Tools form an initial set of GiPSi compliant libraries to support development of GiPSi based simulations. The Auxiliary Functions and the Simulation Kernel are completely application dependent, and can not be specified as part of the API.

Chapter 2

Core GiPSi API

2.1 Simulation Objects

In this framework, organs and physical processes associated with them are represented as Simulation Objects. These objects define the basic API for simulation, interfacing, visualization and haptics (see Fig. 2.1a).

Each Simulation Object can be a single level object implementing a specific physical process or can be an aggregate of other objects creating a hierarchy of models depending on the level of abstraction desired. For example, if we were interested only in muscle model of a beating heart, then we would define the heart as a single object that simulates the muscle mechanics. However, if we were to model a more sophisticated heart with both muscle and blood models, then our heart object would be an aggregate of two objects, one implementing the muscle mechanics and the other implementing the blood dynamics. The specific coupling of these muscle and blood objects would be implemented at their aggregate heart object (see Fig. 2.1c).

The majority of the models in organ level simulations involve solving multiple time varying PDEs that are defined over spatial domains and are coupled via boundary conditions, e.g. a structural model representing the heart muscles coupled with a fluid model representing the blood which share the inner surface of the heart wall as their common boundary. Our goal is to design a flexible API that facilitates the shared development and reuse of models based on these PDEs. Therefore it is necessary to provide: *i)* a *Simulation API iii)* an *Interfacing API.*

2.1.1 Simulation API

There are many different techniques for simulating a given physical process which impose different requirement on the simulation API. In its most general form the simulation API can only be abstracted to a single method Simulate(). Instead of enumerating through each of the techniques and defining a special API for it, we focused on providing one that facilitates development of the most challenging and widely used class of models, namely systems of differential equations, in particular PDEs.

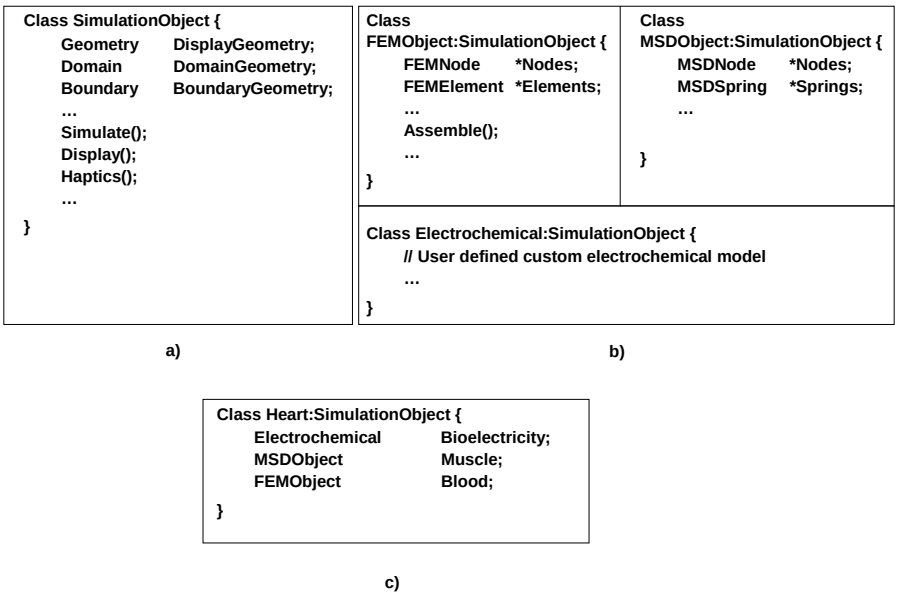

Figure 2.1: a) Simulation Object, b) Examples of modeling tool and user defined objects, c) Heart object

Simulation API for PDE Based Models

The first step in solving a continuous PDE is to discretize the spatial domain it is defined on. Therefore, every object must contain a proper geometry that describes its discretized domain, called the *Domain Geometry*. The definition of this geometry is flexible enough to accommodate the traditional mesh based methods as well as point based (mesh free) formulations. GiPSi defines a set of geometries that can be used as a domain including but not limited to polygonal surface and polyhedral volume meshes. In our current implementation we provide geometries for triangular and tetrahedral meshes.

Second, a method for solving a PDE should be employed such as Finite Element Methods (FEM), Finite Difference Methods (FDM) or Lumped Element (e.g. Mass-Spring-Damper (MSD)) methods. This numerical computation is performed inside the Simulate() method. In the current implementation, basic general purpose objects that implement some of these methods are provided as Modeling Tools, e.g. there is a general customizable FEM object that implements a basic non-linear finite element method for solid mechanics (see Fig. 2.1b). GiPSi also provides a library of numerical analysis tools in the Computational Tools that can be used to solve these discretized equations. Our current implementation provides explicit and implicit integrators, some popular direct and iterative linear system solvers and C++ wrappers around a subset of BLAS and LAPACK functions [1].

2.1.2 Interfacing API

The simulation API also needs to provide a standard means to interface multiple objects. In the models mentioned above, the basic coupling of two objects are defined via the boundary conditions between them. Therefore, we need to provide an API to facilitate the passing

of boundary conditions between different models. First, we need a common definition of the boundary, i.e. each object needs to have a specific *Boundary Geometry*. In our current implementation, we chose triangular surfaces as our standard boundary geometry. Even though the type of the boundary geometry is fixed for every object, the values that can be set at the boundary and their semantics are up to the model developer and should be well documented. Based on this documentation, it is the application developer's task to interface two objects with different semantics on the boundary. For example, a generic fluid object can compute velocities and pressures on its boundary. In order to interface it with a structural object that requires forces on its boundary as boundary conditions, the application developer needs to convert the boundary pressure values to boundary forces by integrating the pressure on the boundary.

Use of boundary conditions is not the only interfacing scheme for objects. For example, the coupling between the electrochemical and mechanical models (excitation-contraction coupling) in the heart is through the commonly occupied volume rather than a shared boundary. Therefore we need a common definition of the domain, i.e. each object needs to have a specific *Domain Geometry*. In our current implementation, we chose 3D point clouds and tetrahedral meshes as our standard domain geometries. A more general information passing over the domain is provided by a simple point and element-wise Get/Set scheme, i.e. an object can read and write values inside another object by simply using Get(value) and Set(value) methods provided by the object respectively. The set of values that can be get and set by other objects and their semantics are again left to the model developer. In the above example, the electrochemical model sets the internal stress values of mechanical model based on the excitation level which in turn result in the contraction of the muscles.

Both interfacing through a surface via boundary conditions and interfacing through a volume (domain) via the Get/Set scheme are achieved by the use of the *Connector* classes. Since the connection of two arbitrary models is application dependent, it is the application developer's task to construct these connectors. Fig. 2.2 shows two connector classes that interface three basic models contained in the aggregate Heart model. The first connector class provides basic communication between the Bioelectrical and Muscle models through their volumetric domain. It basically *gets* the excitation levels from the Bioelectric models (Domain 1), converts them to stress and *sets* the stress tensor values in the Muscle model (Domain 2). The second connector interfaces the Lumped Fluid Blood model with the Muscle model through their surfaces via boundary conditions. In this example the communication is in both ways. The connector class reads the displacement values on the Muscle boundary (Boundary 1), converts them into velocity and passes the velocities to Fluid model (Boundary 2) as boundary conditions. Similarly it receives the boundary pressure values from Boundary 2, converts them into forces and passes them to Boundary 1 as traction values on the boundary.

2.2 Input/Output Subsystem

The Input/Output subsystem provides basic tools for visualizing and interacting with the objects. Currently, GiPSi provides haptics tools for input and visualization tools for output. These tools provide modularity and encapsulation of data, and define a standard API for model developers.

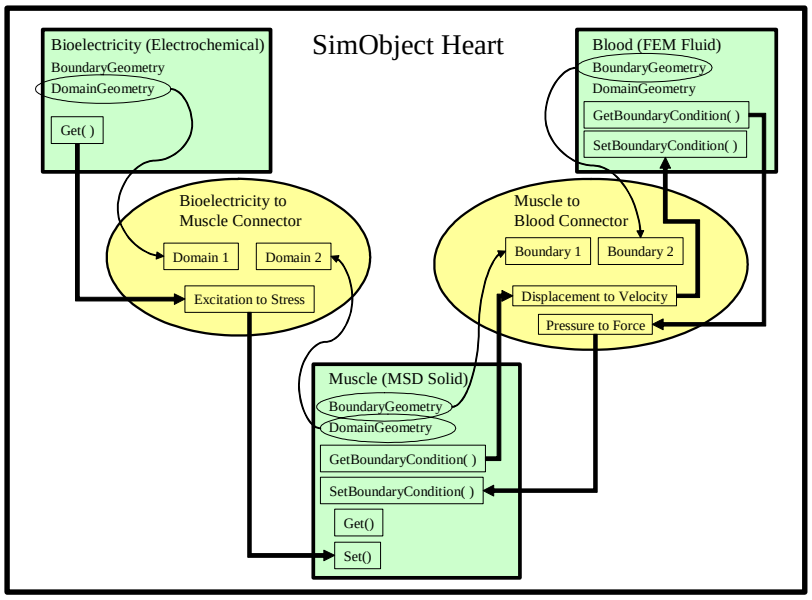

Figure 2.2: Connector class example

2.2.1 Visualization API

Visualization of an object involves displaying the geometry of the object on the screen using a visualization library such as OpenGL, VTK, DirectX etc. The key requirement is development of an API, such that, the actual mechanics of the display specific to a given visualization library is 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.

In order to display an object we need a geometry dedicated for visualization. This geometry is called the *Display Geometry* and can be of any type of geometry defined in GiPSi. However, for modularity, these geometries need to be converted into a standard form. This is done by the *Display Managers* associated with each display geometry. Display managers convert the data in geometries into a standard format used by the visualization module where the actual display takes place. Then the visualization tool accesses this data through the object pool maintained by the simulation kernel and displays it. This makes the development of visualization tools and development of models mutually exclusive and ensures the modularity and the flexibility of the system. In our current design, the standard format used is simply the list of vertex positions, vertex normals, vertex colors and connectivity information. In our current implementation we use OpenGL for actual display.

2.2.2 Haptics API

Haptic interfaces require significantly higher update rates, usually in the order of 1 kHz, than are possible for the rest of the physical models, which are typically run at update

rates in the order of 10 Hz. It is not possible to increase the update rate of the physical models to the haptic rate with their full complexity due to computational limitations, or to decrease the haptic update rate to physical model update rates due to stability limitations. GiPSi handles this conflicting requirements using a multi-rate simulation scheme proposed by Çavuşoğlu in [4]. 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 Haptic I/O module completely encapsulates the haptic interface and its real-time update rate requirements, and provides a standard API for all of the simulation objects which will be haptically interactive. The interface between the haptic I/O module and the simulation objects is through the local dynamic and geometric models provided by the simulation objects, and the haptic instrument location and interaction forces provided by the haptic I/O module. The instrument-object interaction forces are applied to the objects through the object boundary conditions and the instrument-object collision detections are handled no differently than the regular object-object collisions.

Chapter 3

Other Components of GiPSi

3.1 GiPSi Toolset

3.1.1 Computational Tools

GiPSi implementation provides a set of computational tools to support the simulation of algebraic and differential equation based models. The computational tools include basic linear algebra operations on vectors and matrices, explicit numerical integrators, some popular direct and iterative linear system solvers, and C++ wrappers around a subset of BLAS and LAPACK functions [1].

Basic vector and matrix operations are the backbones of any simulation framework. GiPSi provides a C++ based matrix and vector operations toolbox. In this toolbox, basic vector and matrix classes implement the vector-scalar, vector-vector, vector-matrix, matrix-scalar, and matrix-matrix algebraic operations, basic matrix inversions, and simple I/O functions. Another important enabling tools required for development of numerical simulations is the linear algebraic system solvers. Pseudo-inverse and LU decomposition techniques are provided as direct linear system solvers by means of C++ wrappers around LAPACK functions. Conjugate Gradient, Jacobi, and Successive Over-Relaxation algorithms are the iterative linear solvers implemented. Finally, GiPSi also provides a suite of numerical integrators: a number of popular single and multi-step explicit methods, including Runge-Kutta and Adams-Bashford algorithms, are implemented.

3.1.2 Modeling Tools

GiPSi provides two sample modeling tools in the current implementation, namely Nonlinear Finite Elements based solid mechanics model (FEM_Object), and a Lumped Element solid mechanics model (MSD_Object). The FEM_Object is a basic geometrically nonlinear FEM model which uses linear tetrahedral elements to model linear viscoelastic solid materials. The MSD_Object is a simple Mass-Spring-Damper based geometrically nonlinear lumped element model which can be used to model deformable solids.

FEM_Object

For the FEMObject we followed the formulation in [10] which uses an explicit discretization of a geometrically nonlinear linear elasticity model.

MSD_Object

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. 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

$$m_i \ddot{\mathbf{x}}_i \quad = \quad F_i(\mathbf{x}, \dot{\mathbf{x}}) + F_e \tag{3.1}$$

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

$$F_i(\mathbf{x}) = \sum_{\{i,j \text{ connected}\}} \mathbf{f}(\mathbf{x}_i, \mathbf{x}_j) + \sum_{\{i,j \text{ connected}\}} \mathbf{g}(\mathbf{x}_i, \mathbf{x}_j, \dot{\mathbf{x}}_i, \dot{\mathbf{x}}_j) \tag{3.2}$$

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

$$\mathbf{f}(\mathbf{x}_1, \mathbf{x}_2) = k(\|\mathbf{x}_1 - \mathbf{x}_2\| - L_0)(\mathbf{x}_2 - \mathbf{x}_1)/(\|\mathbf{x}_1 - \mathbf{x}_2\|). \tag{3.3}$$

For the linear damper, the force expression is in the form

$$\mathbf{g}(\mathbf{x}_1, \mathbf{x}_2, \dot{\mathbf{x}}_i, \dot{\mathbf{x}}_j) = b\left((\dot{\mathbf{x}}_1 - \dot{\mathbf{x}}_2) \cdot \frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|}\right) \frac{\mathbf{x}_2 - \mathbf{x}_1}{\|\mathbf{x}_1 - \mathbf{x}_2\|}. \tag{3.4}$$

These expressions are for force acting on node $\mathbf{x}_1$, due to the a spring and damper between $\mathbf{x}_1, \mathbf{x}_2$. L_0 is the rest length of the linear spring.

3.2 Other Functionality Needed for Interactive Simulation System Development

3.2.1 Collision Detection/Collision Response

In interactive surgical simulations one needs to detect collisions to prevent penetration between objects in the system, such as organ models and tools used during surgery. Therefore collision detection (CD) and collision response (CR) play an important role. In our framework, CD module detects the collisions between boundary geometries of different models and the CR module computes the required response to resolve these collisions in terms of displacements and/or penalty forces and communicates the result to the models as displacement or force based boundary conditions. The models process these boundary conditions if necessary and iterate. As a result, the mechanics of contact detection and resolution is done by the application developer, and therefore becomes transparent to the model developer. Hence, the framework is flexible enough to accommodate a wide variety of CD/CR schemes.

3.2.2 Simulation Kernel

The simulation kernel acts as the central core where everything above comes together. Since the simulation kernel completely represents the application itself, it needs to be specified entirely by the application developer. Its tasks include the management of the top level object pool, coordination of the object interactions, and arbitration of the communication between the components. This involves establishing the execution order of the models and the specific interfacing between them, allowing the application developer to properly specify the semantics of the individual top level objects and the interfacing between them, based on the specific application that the simulation is being developed for.

Acknowledgements

This research was supported in part by National Science Foundation under grants CISE IIS-0222743, CDA-9726362 and BCS-9980122, and US Air Force Research Laboratory under grant F30602-01-2-0588. We also would like to thank Xunlei Wu for his valuable discussions and feedback.

Chapter 4

Appendix

4.1 Class Hierarchy

The class hierarchy is shown in figures 4.1-4.7.

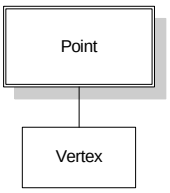

Figure 4.1: Primitive class hierarchy

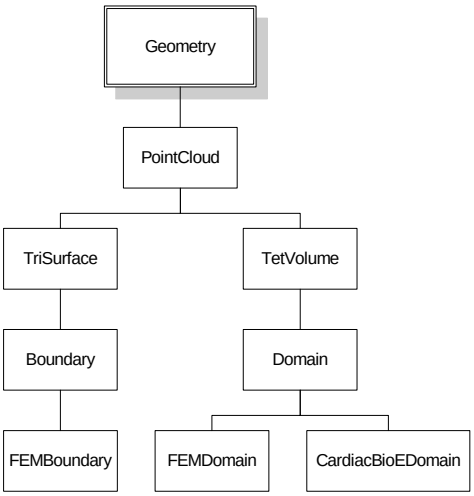

Figure 4.2: Geometry class hierarchy

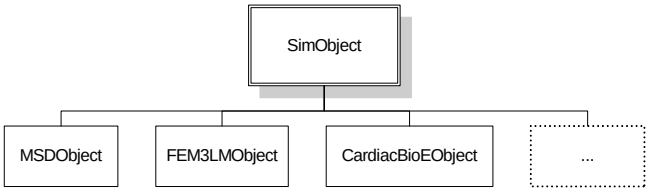

Figure 4.3: SimObject class hierarchy

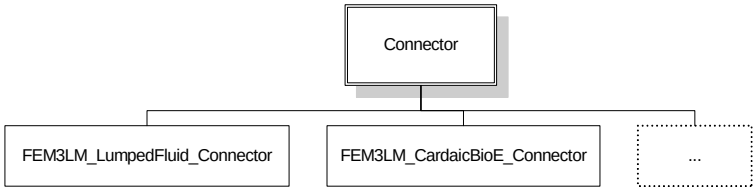

Figure 4.4: Connector class hierarchy

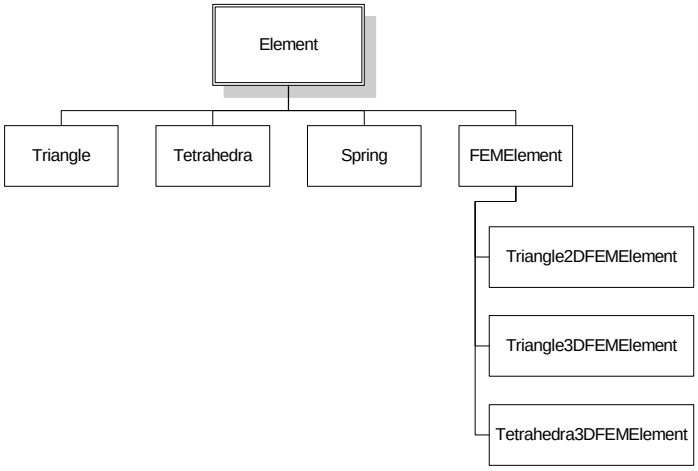

Figure 4.5: Connector class hierarchy

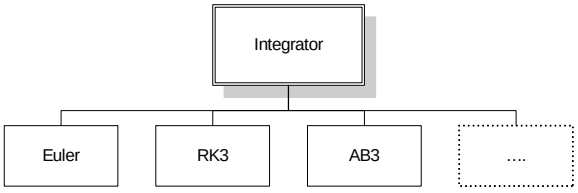

Figure 4.6: Integrator class hierarchy

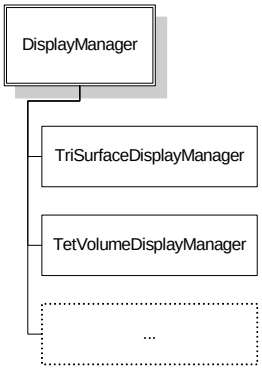

Figure 4.7: Display Manager class hierarchy

4.2 Core GiPSi API Specification

4.2.1 Geometric Primitives

	Name	Type	Description
Class Name	**Point**		Base class for all point like primitives, e.g. vertex, node
Parent Class			
Public Fields	refid pos	unsigned int Vector<Real>	Reference ID number Position
Protected Fields			

Constructors	`Point`()
Public Methods	
Protected Methods	

	Name	Type	Description
Class Name	**Vertex**		Base class for vertex primitive
Parent Class	Point		
Public Fields	n valence color texcoord	Vector<Real> int Vector<Real> Vector<Real>	Normal Degree Color Texture coordinates
Protected Fields			

Constructors	`Vertex`()
Public Methods	Initialize void `init` (unsigned int refid, Real *pos, Real *n, Real *color)
Protected Methods	

	Name	Type	Description
Class Name	**Element**		Base Element Class
Parent Class			
Public Fields	refid	unsigned int	Reference id
Protected Fields			

Constructors	`Element`()
Public Methods	
Protected Methods	

	Name	Type	Description
Class Name	**Triangle**		Basic triangular element
Parent Class	Element		
Public Fields	n	Vector<Real>	Normal vector
	vertex[3]	Vertex*	Vertices
Protected Fields			

Constructors	`Triangle`()
Public Methods	Initialize void `init`(unsigned int refid, Real *n, Vertex **vertex)
Protected Methods	

	Name	Type	Description
Class Name	**Tetrahedra**		Simple Tetrahedral element
Parent Class	Element		
Public Fields	vertex[4]	Vertex*	Vertices
Protected Fields			

Constructors	`Tetrahedra`()
Public Methods	Initialize void `init`(unsigned int refid, Vertex **vertex)
Protected Methods	

4.2.2 Geometry API

	Name	Type	Description
Class Name	**PointCloud**		Base class for point cloud geometry
Parent Class	Geometry		
Public Fields	vertex	Vertex *	Vertex array
	num_vertex	int	Number of vertices
Protected Fields			

Constructors	`PointCloud`(float r, float g, float b, float a)
Public Methods	Loader for .obj files void `Load`(char*)
	Translate object in space void `Translate`(float tx, float ty, float tz)
	Rotate object in space void `Rotate`(Real angle, Real ax, Real ay, Real az)
	Scale dimensions of the object void `Scale`(float sx, float sy, float sz)
Protected Methods	

	Name	Type	Description
Class Name	**TriSurface**		Base class for simple triangular geometry class
Parent Class	PointCloud		
Public Fields	face	Triangle *	Face array
	num_face	int	Number of faces
Protected Fields			

Constructors	`TriSurface`(float r, float g, float b, float a)
Public Methods	Loader for .obj files void `Load`(char*)
	Calculate surface normals void `calcNormals`(void)
Protected Methods	

	Name	Type	Description
Class Name	**TetVolume**		Tetrahedral volume geometry
Parent Class	PointCloud		
Public Fields	face	Triangle *	Face array
	num_face	int	Number of faces
	tet	Tetrahedra *	Tetrahedra Array
	num_tet	int	Number of tetrahedra
Protected Fields			

Constructors	`TetVolume`(float r, float g, float b, float a)
Public Methods	Initialize void `init`(int num_vertex, int num_face, int num_tet)
	Loader for .obj files void `Load`(char*)
	Loader for Pyramid files void `Load`(LoadData*)
	Calculate Normals void `calcNormals`(void)
Protected Methods	

	Name	Type	Description
Class Name	**Boundary**		Base Boundary class
Parent Class	TriSurface		
Public Fields			
Protected Fields			

Constructors	
Public Methods	
Protected Methods	

	Name	Type	Description
Class Name	**Domain**		Base Domain class
Parent Class	TetVolume		
Public Fields			
Protected Fields			

Constructors	
Public Methods	
Protected Methods	

4.2.3 Explicit Numerical Integrators API

	Name	Type	Description
Class Name	**Integrator**		Basic explicit numerical integration class (Templated with system class)
Parent Class			
Public Fields			
Protected Fields			

Constructors	
Public Methods	Integrate one time step virtual void `Integrate`(S &system, Real h)
Protected Methods	

4.2.4 Simulation Object API

	Name	Type	Description
Class Name	**SIMObject**		Base Simulation Model class
Parent Class			
Public Fields	displayMngr	DisplayManager *	Display manager associated with this model
Protected Fields	name	char *	Name of the model
	geometry	Geometry *	Display geometry
	boundary	Boundary *	Boundary geometry
	domain	Domain *	Domain geometry
	time	Real	Local time
	timestep	Real	Local time step

Constructors	`SIMObject`(char* name, Real time, Real timestep)
Public Methods	Get name of the model char* `GetName`(void)
	Set name of the model void `SetName`(char* newname)
	Get models's local time step Real `GetTimestep`(void)
	Set models' local time step void `SetTimestep`(Real dt)
	Get model's local time Real `GetTime`(void)
	Set model's local time void `SetTime`(Real t)
	Get pointer to the display geometry Geometry* `GetGeometryPtr`(void)
	Get pointer to the boundary geometry Boundary* `GetBoundaryPtr`(void)
	Get pointer to the domain geometry Domain* `GetDomainPtr`(void)
	Load model virtual void `Load`(char* filename)
	Display model virtual void `Display`(void)
	Setup display maqnager and data for the model virtual void `SetupDisplay`(void)
	Simulate the model virtual void `Simulate`(void)
Protected Methods	

4.2.5 Connector API

	Name	Type	Description
Class Name	**Connector**		Base connector class
Parent Class			
Public Fields			
Protected Fields	boundaries	Boundary *	List of connected boundaries
	domains	Domain *	List of connected domains

Constructors	
Public Methods	Perform communication between the connected models virtual void `process`(void)
Protected Methods	

4.2.6 Display Manager API

	Name	Type	Description
Class Name	**DisplayManager**		Base display manager class
Parent Class			
Public Fields			
Protected Fields	display	DisplayArray	Standard display data filled in by the manager

Constructors	`DisplayManager`(const DisplayHeader &inheader)
Public Methods	Get header information void `GetDisplayHeader`(DisplayHeader &outheader)
	Set header information void `SetDisplayHeader`(const DisplayHeader &inheader)
	Get pointer to display data DisplayArray* `GetDisplay`(void
	Display (i.e update the display array) virtual void `Display`(void)
Protected Methods	

	Name	Type	Description
Class Name	**TriSurfaceDisplayManager**		Display manager for the triangular surface geometry
Parent Class	DisplayManager		
Public Fields			
Protected Fields	geometry	TriSurface *	The triangular surface geometry to display

Constructors	`TriSurfaceDisplayManager` (TriSurface* ingeometry, DisplayHeader &inheader)
Public Methods	Display (i.e update the display array) void `Display`(void)
Protected Methods	

	Name	Type	Description
Class Name	**TetVolumeDisplayManager**		Display manager for the tetrahedral volume geometry
Parent Class	DisplayManager		
Public Fields			
Protected Fields	geometry	TetVolume *	The tetrahedral volume geometry to display

Constructors	`TetVolumeDisplayManager` (TetVolume* ingeometry, DisplayHeader &inheader)
Public Methods	Display (i.e update the display array) void `Display`(void)
Protected Methods	

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (3rd ed.)*. SIAM, 1999.

[2] R. M. Berne and M. N. Levy, editors. *Principles of Physiology*. Mosby, Inc., St. Louis, MO, third edition, 2000.

[3] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation special issue on Simulation Software Development*, 1994.

[4] 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)*, pages 2458–2465, April 2000.

[5] S. Cotin, D. W. Shaffer, D. A. Meglan, M. P. Ottensmeyer, P. S. Berry, and S. L. Dawson. CAML: A general framework for the development of medical simulations. In *Proceedings of SPIE Vol. 4037: Battlefield Biomedical Technologies II*, 2000.

[6] A. Joukhadar and C. Laugier. Dynamic simulation: Model, basic algorithms, and optimization. In J.-P. Laumond and M. Overmars, editors, *Algorithms For Robotic Motion and Manipulation*, pages 419–434. A.K. Peters Publisher, 1997.

[7] Mathworks, Inc. Simulink. http://www.mathworks.com/products/simulink/.

[8] *Modelica — A Unified Object-Oriented Language for Physical Systems Modeling; Language Specifications 2.0*. The Modelica Association, 2002. http://www.modelica.org/.

[9] K. Montgomery, C. Bruyns, J. Brown, S. Sorkin, F. Mazzella, G. Thonier, A. Tellier, B. Lerman, and A. C. Menon. Spring: A general framework for collaborative, real-time surgical simulation. In J. Westwood et al., editor, *Medicine Meets Virtual Reality (MMVR 2002)*, Amsterdam, 2002. IOS Press.

[10] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 137–146. ACM Press/Addison-Wesley Publishing Co., 1999.

[11] F. Tendick, M. Downes, T. Goktekin, M. C. Çavuşoğlu, D. Feygin, X. Wu, R. Eyal, M. Hegarty, and L. W. Way. A virtual environment testbed for training laparoscopic surgical skills. *Presence*, 9(3):236–255, June 2000.

[12] 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 2001*, September 2001.