GiPSi: A Framework for Open Source/Open Architecture Software Development for Organ-Level Surgical Simulation

M. Cenk Çavuşoğlu, *Member, IEEE*, Tolga G. Göktekin, and Frank Tendick, *Member, IEEE*

***Abstract*—This paper presents the architectural details of an evolving open source/open architecture software 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 application programming interface 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. Input/output interfaces for visualization and haptics for real-time interactive applications have also been provided.**

***Index Terms*—Open architecture framework, shared development, surgical simulation, virtual environments.**

I. Introduction

COMPUTER simulations have become an important tool for medical applications such as surgical training, preoperative 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 interoperable nor independently verifiable. 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 from different fields is required. The motivation behind this study is our prior experience in surgical training simulators and physically based modeling [1]–[4].

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 provide means for 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 interconnectibility 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.

In this paper, we describe general interactive physical simulation interface (GiPSi), 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. GiPSi focuses on addressing technical issues in accommodating different levels and types of model abstractions, supporting heterogeneous models of computation, developing application programming interfaces (APIs) for interfacing various heterogeneous physical processes, and achieving modularity. In addition, input/output (I/O) interfaces for visualization and haptics for real-time interactive applications have been provided. The latest release of the GiPSi framework is available as open source at the project web site, http://gipsi.case.edu/.

An important difference of GiPSi from earlier object-oriented tools and languages for modeling and simulation of complex physical systems, such as Modelica [5], Matlab Simulink [6], and Ptolemy [7], 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 [8] or AlaDyn-3D [9], 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 [10]. SCIRun [11] is a general-purpose problem solving environment developed for scientific computation. As such, it focuses on integrating stand-alone modules and programs involved in the general scientific workflow into a single framework, using a data-flow architecture to integrate the steps of preparing, executing, and visualizing simulations of physical and biological

Manuscript received May 1, 2005; revised August 31, 2005. This work was supported in part by the National Science Foundation under Grants CISE CDA-9726362, BCS-9980122, IIS-0222743, EIA-0329811, and CNS-0423253, in part by the U.S. Air Force Research Laboratory under Grant F30602-01-2-0588, and in part by the U.S. Department of Commerce under Grant TOP-39-60-04003.

M. C. Çavuşoğlu is with the Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH 44106 USA.

T. G. Göktekin is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA.

F. Tendick is with the Department of Surgery, University of California, San Francisco, CA 94143 USA.

Digital Object Identifier 10.1109/TITB.2005.864479

systems. It is designed as a problem solving environment, rather than a framework to construct real-time interactive simulations like GiPSi. FEMLab [12] (by COMSOL, Inc.) is a commercial toolset for modeling and simulation of scientific and engineering problems based on partial differential equations. Notably, it focuses on models and simulations involving heterogeneous physical processes. However, as a framework, it is fundamentally based on a single class of models, namely, finite element models.

It is also important to note several other relevant ongoing open source efforts in the general area of medical modeling. 3DSlicer [13] is an open source medical visualization and processing environment for visualization, registration, segmentation, and quantification of medical data. The Insight Segmentation and Registration Toolkit (ITK) [14] is an open source software system for segmentation and registration of (medical) image datasets, which started as a toolkit for supporting the Visible Human Project [15].

The rest of the paper starts with a discussion of the technical issues that need to be considered in the design of an open architecture simulation framework for medical simulation. This is followed by an overview of GiPSi, presentation of the core GiPSi API and other components of GiPSi, and a discussion of the implementation of a simple heart model composed of heterogeneous submodels.

II. Technical Issues in Development of an Open Architecture Framework for Medical Simulation

At this point, it valuable to discuss the key technical issues that need to be addressed in the development of open source/open architecture medical simulation frameworks.

A. 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 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 electrochemically 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 levels of abstraction: the heart muscle contraction can be modeled starting at the individual cell level, at the tissue level, or at the level of the whole organ. In surgical simulations, it is important to have an accurate model of the mechanical manipulation of the heart. Hence, when modeling effects of medications in a heart surgery simulation, it may be sufficient to use an abstraction that includes the aggregate physiological effects of different medications used during the procedure to the extent they affect 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.

B. 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.

C. 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 the API: 1) interfacing multiple physical processes at the semantic level and 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.

D. Modularity Through Encapsulation and Data Hiding

Maintaining the integrity of the data of the individual models in an open architecture simulation is an important requirement. Moreover, the API and the overall framework also need 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 facilitate the modularity of the components while maintaining the data integrity. It also provides mechanisms to interface and embed the constructed models and other computational modules to a larger, more sophisticated model.

E. Support for Parallel and Distributed Simulation

In a surgical simulation, software modules numerically simulate the physics of a target environment. Highly accurate simulations for surgical planning and compelling virtual environments for training typically require extensive computation available beyond basic desktop computers or single processor workstations. It is therefore necessary for the simulation framework to support parallel and distributed simulations. Beyond just parallel processing, development of network-enabled virtual environments is desirable to extend the accessibility of surgical

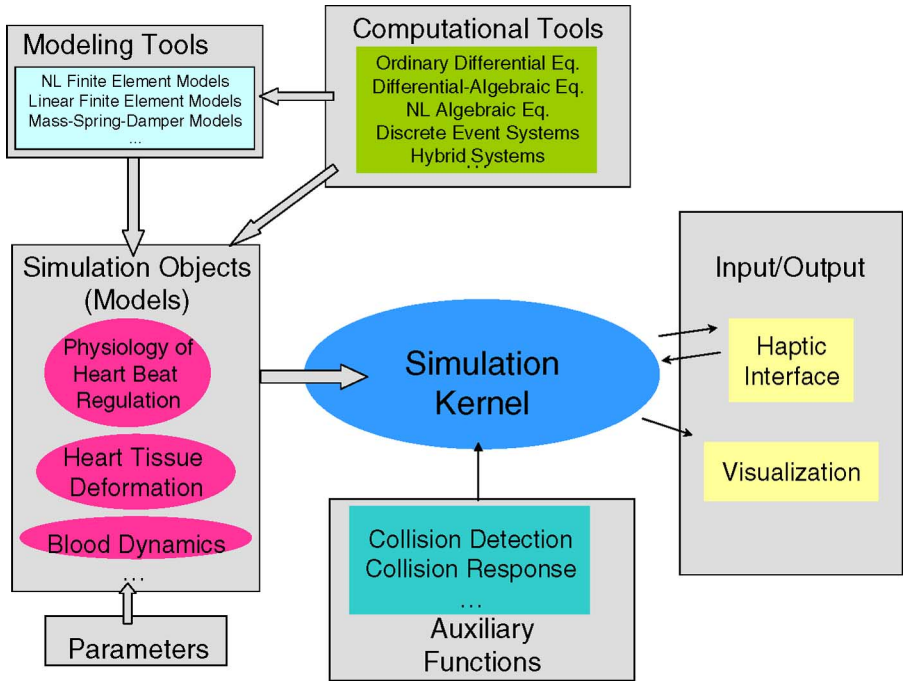

Fig. 1. Architecture of a GiPSi-based simulation system.

simulations, and to allow computation to take place in existing computing facilities while supporting planning and training from a variety of locations. This would allow sharing of computational resources and ease the logistical requirements for deploying virtual environment-based simulators.

F. 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.

G. Customization to Patient-Specific Physiology

In surgical planning and preoperational rehearsals, it is necessary to use patient-specific models during simulation. Therefore, the models in the simulation need to be customizable. This 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.

The main goal of the GiPSi framework is to facilitate shared model development and simulation of organ-level processes, as well as data sharing among multiple research groups. In addressing these, we explicitly 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. Basic support for parallel simulations has been included in the form of support for separating main computation, visualization, and haptics as separate threads, as discussed in Section V, on a multiprocessor computer using the open source pthreads libraries. Including further support for parallel and distributed simulations into GiPSi is part of our ongoing and future work, as discussed in Section VIII. Issues of model validation have not been explicitly addressed in GiPSi, and are left to the application developer, which is possible, as GiPSi is an open source framework. Customization of the simulations with patient-specific data is implicitly supported by the GiPSi architecture (Fig. 1), and explicit support depends on the specifics of the models used.

III. GIPSI 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 interoperate and be reused. Therefore, modularity through encapsulation and data hiding between the components are enforced. In addition, a standard interfacing API facilitating communication among these components is provided.

We developed our tools in the context of 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: 1) multiple heterogeneous processes that need to be modeled and interfaced and 2) 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 be 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 [16]. Models for all these processes need to be intimately coupled: the mechanical and

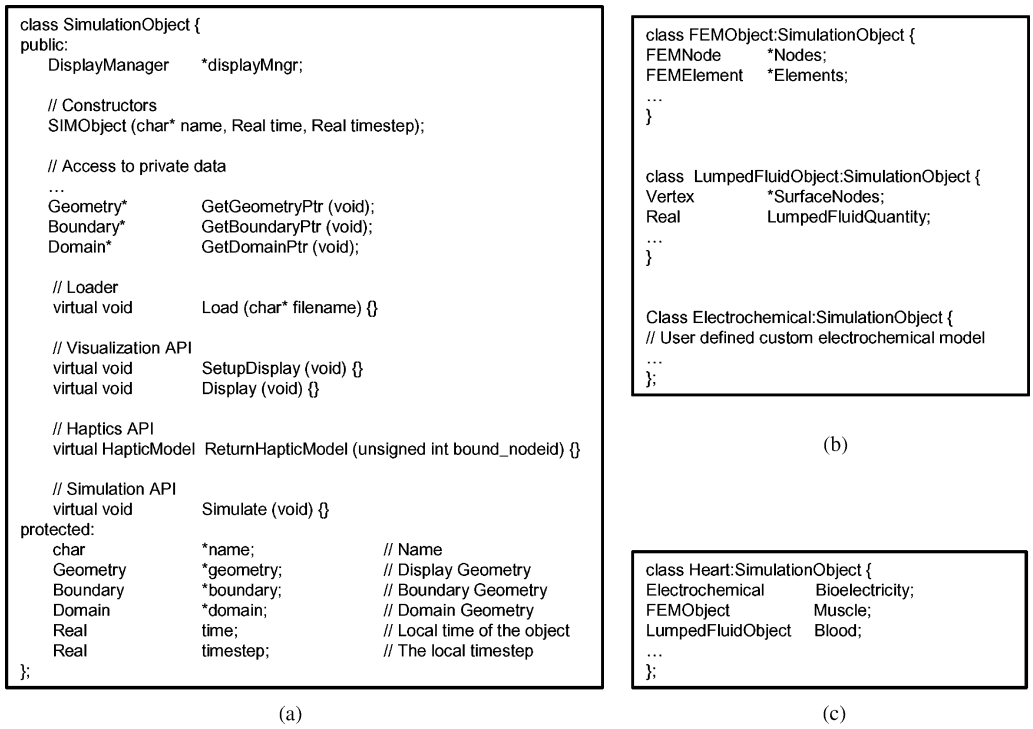

Fig. 2. (a) Simulation Object. (b) Examples of modeling tool and user-defined objects. (c) Heart object.

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

The overall system architecture of GiPSi is shown in Fig. 1. The models of physical processes such as muscle mechanics of the heart are represented as *Simulation Objects* (Section IV). Each simulation object can be derived from a specific computational model contained in *Modeling Tools* such as finite elements, finite differences, lumped elements, etc. (Section VI-A2). The *Computational Tools* provide a library of numerical methods for low-level computation of the object's dynamics (Section VI-A1). 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 (Section VI-B2). 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 (Section V). 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 (Section VI-B1). The class hierarchy of GiPSi is shown in Fig. 7. The implementation of the framework is done using C++ and it is platform-independent.

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 includes the Simulation API and 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 cannot be specified as part of the API. The specific models discussed in the paper are for illustration of the APIs and the framework architecture, and are not intended to be sophisticated or comprehensive.

IV. Simulation Object API

In this framework, organs and the 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(a)].

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 one were interested only in a muscle model of a beating heart, then one would define the heart as a single object that simulates the muscle mechanics. However, if one were to model a more sophisticated heart with both muscle and blood models, then the 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(c)].

The majority of the models in organ-level simulations involve solving multiple time-varying partial differential equations (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: 1) a *Simulation API* and 2) an *Interfacing API*.

A. Simulation API

There are many different techniques for simulating a given physical process, each of which impose different requirements on the simulation API. In its most general form, the simulation API can only be abstracted to a single method, named, 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.

1) Simulation of 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. The model developer can also define new geometry, derived from the base geometry class, if desired. In our current implementation geometries for triangular and tetrahedral meshes, and point clouds are predefined. Second, a method for solving a PDE should be employed such as finite element methods (FEMs), 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 (Section VI-A2); e.g., there is a general customizable FEM object that implements a basic nonlinear finite element method for solid mechanics [see Fig. 2(b)]. GiPSi also provides a library of numerical analysis tools in the Computational Tools (Section VI-A1) that can be used to solve these discretized equations. Our current implementation provides a collection of explicit integrators, some popular direct and iterative linear system solvers, and C++ wrappers around a subset of BLAS and LAPACK functions [17].

Existence of separate geometries for display, computation domain, and boundaries is an important feature of the GiPSi framework. This allows the framework to be model-independent; i.e., not be tied to a specific class of models such as spring-based models or, more generally, mesh-based models. Different representations of the same object can, therefore, be used for computation, display, and interfacing (including collision detection) purposes. Performance considerations have not been ignored in the design. Our object-oriented design approach allows a single data structure to be used for all these purposes, eliminating data translation overhead, if different geometric representations are not needed.

B. Object Interfacing API

The simulation object API also needs to provide a standard mechanism to interface multiple objects. In the models mentioned above, the basic coupling of two objects is 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 done through the commonly occupied volume rather than a shared boundary. Therefore, for this case, we need to communicate through the Domain Geometry. 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 preceding example, the electrochemical model sets the internal stress values of the mechanical model based on the excitation level which, in turn, result in the contraction of the muscles.

In the definition of the interfacing API, point-based geometry was chosen for its generality, and triangle based polygonal surfaces were chosen for their popularity. It is true that using point clouds or surface triangulations for interfacing multiple implicit surface based models is not optimal. However, we believe that the overhead is not prohibitive and does not outweigh the flexibility. On a practical note, almost every type of geometric representation (e.g., implicit, parametric, etc.) needs to be triangulated for fast display on conventional graphics hardware. Therefore, many applications already perform the conversion, but perhaps not as frequently.

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. 3 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 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

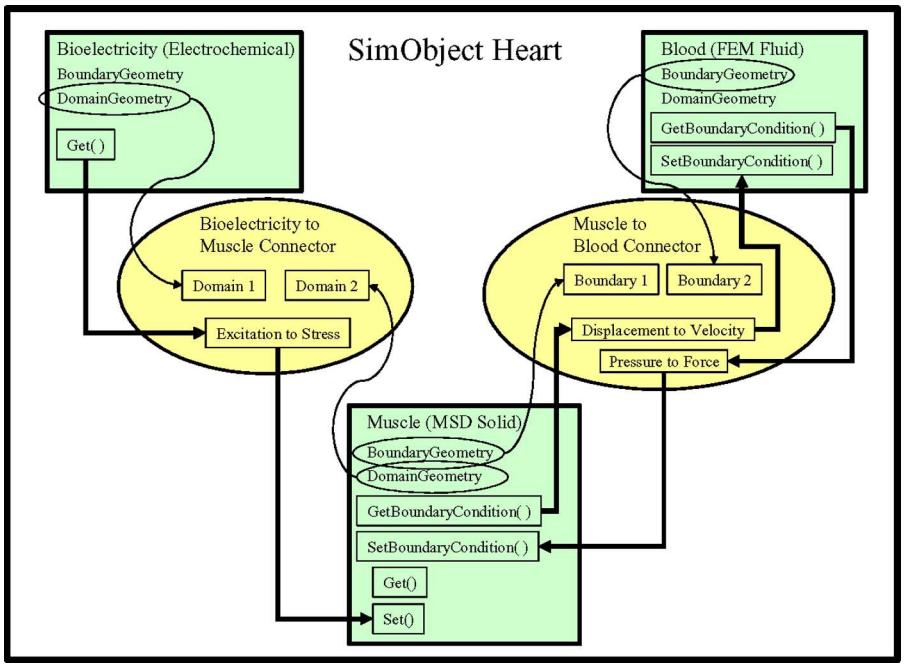

Fig. 3. Connector class example.

the lumped fluid blood model with the muscle model through their surfaces via boundary conditions. In this example the communication is in both directions. The connector class reads the displacement values on the muscle boundary (Boundary 1), converts them into velocities and passes the velocities to the 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.

For most natural and flexible interfacing functionality, the fundamental solid mechanics boundary conditions have been used as the boundary interfacing semantics for deformable solid objects. As part of the deformable solid object API, displacement (Dirichlet), force (Neumann), and mixed force-displacement type boundary conditions have been used, and the mechanisms to specify them have been defined.

V. 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.

A. 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 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.

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 a visualization engine based on OpenGL for actual display.

B. 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 feasible 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 these conflicting requirements using a multirate simulation scheme proposed by Çavuşoğlu [18]. 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 intersample interaction forces and intersample 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.

VI. Other Components of GiPSi

A. GiPSi Toolset

1) Computational Tools: The 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 [17].

Basic vector and matrix operations are the backbone 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. Other important enabling tools required for the development of numerical simulations are linear algebraic system solvers. Pseudoinverse 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. Featuring a number of popular single and multistep explicit methods, including Runge–Kutta and Adams–Bashford algorithms.

2) Modeling Tools: GiPSi provides two sample modeling tools in the current implementation: a nonlinear finite element-based solid mechanics model (FEM_Object), and a lumped element solid mechanics model (MSD_Object). These models are derived from a common deformable solid object API, which is, in turn, derived from the simulation object class, as shown in Fig. 7(c). 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.

a) FEM_Object: For the FEMObject we followed the formulation in [19], which uses an explicit discretization of a geometrically nonlinear, linear elasticity model.

b) 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. The equations of motion are the collection of Newton's equations written for the individual nodal masses.

B. Other Functionality Needed for Interactive Simulation System Development

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, the 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 iterates. 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. There are no preimplemented CD/CR modules included in the current release of GiPSi.

2) Simulation Kernel: The simulation kernel acts as the central core where everything described previously 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 for which the simulation is being developed. The simulation kernel of a typical interactive simulator implemented using GiPSi would have the functionality shown in Fig. 4.

The simulation kernel encapsulates an important aspect of the simulation; namely, the model and interconnection semantics. The execution order and the synchronization of the individual models and connectors within the simulation have not been explicitly specified as part of GiPSi, since this requires knowledge of the semantics of the models, and the context in which they

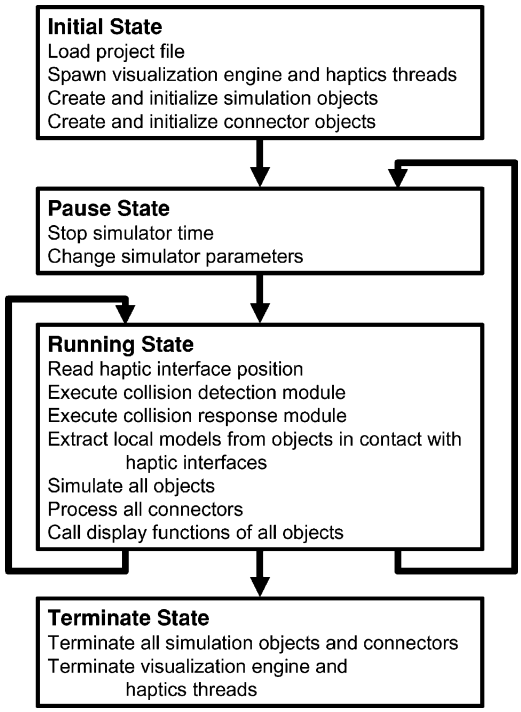

Fig. 4. Simulation kernel state diagram for a typical interactive simulator implemented using GiPSi.

are integrated; i.e., the application. As the specifics of these are application dependent, it is not desirable to constrain the model and connector semantics as part of the simulation framework. Within GiPSi, the specification of the execution order of the individual models and connectors inside the simulation kernel are left to the application developer, in order to provide a general purpose framework, and give flexibility to the application developer in determining the model and interconnection semantics in the context of the specific simulation being developed.

The sample simulation kernel diagrammatically shown in Fig. 4 demonstrates the range of functionality that needs to be implemented inside the simulation kernel of a typical simulation. The specifics of the execution order of the individual models and connectors are application dependent and needs to be explicitly specified.

VII. Case Study

We implemented a double chamber heart model as a test-bed model to evaluate the functionality of the API [Fig. 5]. The model is composed of three independent heterogeneous models [Fig. 2(c)] coupled together using the APIs and mechanisms provided by GiPSi, as shown in Fig. 3. The first is a three-dimensional nonlinear geometry linear viscoelastic-material finite element model of the cardiac muscle. The second model is a lumped fluid model of the blood dynamics, which can optionally be coupled to a simplified Windkessel model of the systemic circulation. The third model is a simplified discrete event model of cardiac bioelectricity.

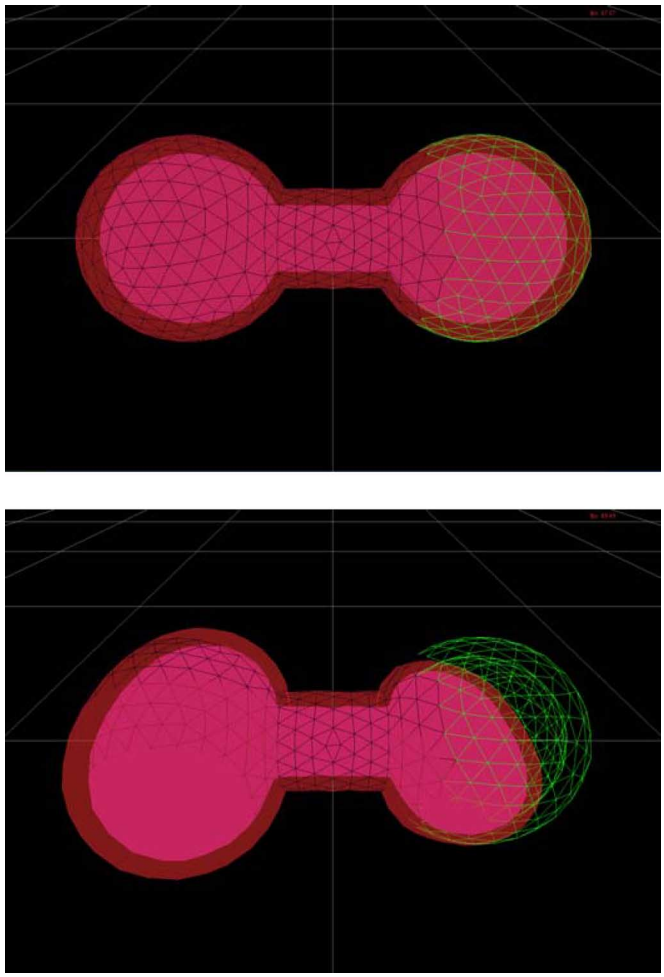

Fig. 5. Simulation of the double-chamber heart model implemented in GiPSi. The opaque core is the surface of the fluid, and the translucent volume is the heart muscle. The wireframe mesh corresponds to the undeformed shape of the heart muscle, with the bright areas corresponding to the part of the muscle being excited by the bioelectrical model.

The blood and muscle models are coupled through their shared boundary, the inner surface of the chamber. The interaction between the two models is obtained with a connector class which accesses the boundary interface API of the two models. This connector handles the structure and semantic interfacing between the models by converting the boundary displacements of the muscle model to velocity boundary conditions to the blood fluid model, and converting the pressure on the surface of the blood to force (Neumann-type) boundary conditions on the muscle surface.

The muscle and bioelectricity models are coupled through a cooccupied volume. The interaction between the two models is obtained with a connector class which accesses the domain interface API of the two models. This connector mainly implements a basic excitation-contraction coupling by converting the excitation level at a given element of the bioelectricity model to an internal stress in the corresponding element of the finite element model of the heart muscle.

The simulation was tested and benchmarked on a Microsoft Windows XP-based workstation with a single 2.8-GHz Intel

Xeon Processor, 1 GB of RAM, and a PCI-express NVidia Quadro FX 1300 Graphics Card to collect performance benchmarks. The finite element-based heart muscle model was composed of 1415 nodes and 4237 tetrahedral elements, which resulted in a 8490th-order ordinary differential equation system. The lumped fluid model of the blood was a 611th-order model, with 610 states for tracking the fluid surface, and 1 state for lumped blood quantity. The bioelectrical model was a second-order nonlinear oscillator driving a discrete event model. All the models were updated with the same update rate. A third-order Heun predictor-corrector numerical integration algorithm, which is a three-step explicit integration algorithm (requiring three accumulations and three computations of the integrand for every time step), was employed in the numerical simulation of all the models. The resulting simulation executed a single simulation time step, updating all the models, in 246 ms (average value with standard deviation 1.3 ms). The graphics display inside the visualization engine was simultaneously operating at 60 frames/s in a separate thread.

A detailed profiling of the simulation using the Intel VTune performance analyzer was used to measure the sources of overhead resulting from using the GiPSi API, which are the data translations between the model state and the display, boundary, and domain geometries, and the data translations occurring inside the connectors. The profiling analysis was performed using the call graph profiling functionality provided by VTune, measuring the total time spent inside the functions performing the core computations and functions responsible for API overhead operations mentioned previously. The profiling analysis revealed that the overhead accounted for less than 7% of the overall computation in the main simulation thread. There is no overhead inside the visualization engine resulting from the GiPSi API, as the data translation from state to display geometry occurs inside the main simulation thread.

VIII. Conclusion

We presented the architectural details of GiPSi, an evolving open source/open architecture software framework for developing organ-level surgical simulations. In this framework we provide a *Simulation Object API* to encapsulate the organs and the physical processes associated with them. This API enables the shared development and reusability of these models. A hierarchy of different Simulation Objects can be created to facilitate different levels of abstraction. The framework also provides a *Simulation API* which supports a set of *Modeling Tools* in order to accommodate heterogeneous models of computation. Finally, we provide an *Object Interfacing API* for interfacing dynamic models defined over spatial domains through boundary conditions and more general Get/Set mechanisms.

We used a virtual single-chamber heart simulation as a test application. In order to support interactive applications, we also developed APIs for visualization and haptics. To test the modeling API to its full extent, we modeled the heart as a composition of three closely coupled heterogeneous models including cardiac muscle, cardiac bioelectricity, and blood.

Fig. 6. Snapshot from the virtual environment-based endoscopic neurosurgery training simulator testbed being developed using the GiPSi framework.

The GiPSi framework is also being used in house as the main development framework in an ongoing project (at Case Western Reserve University) to develop and validate a virtual environment-based simulator for skill and task training for endoscopic neurosurgery, demonstrating the general purpose nature of the framework. As part of this project, an endoscopic third ventriculostomy simulator testbed is being constructed (Fig. 6). The virtual environment used in the simulation is based on geometric models of the anatomy constructed from patient-specific medical imaging data. Heterogeneous physical models are used for simulation of the deformable tissue and cerebrospinal fluid present in the surgical site, and heterogeneous models of computation are employed to manage computational complexity while maintaining realism.

Our future work on GiPSi will focus on development of more modeling tools, including a fully nonlinear solid FEM implementation and a finite differences-based fluid model implementation, and implementation of more detailed models of biological systems. Another ongoing research track we are pursuing is the development of network middleware for GiPSi to enable seamless development of networked virtual environments for simulation. We are also planning to port the multigrid finite element model of deformable solids developed by Wu and Tendick [20], and its parallel implementation developed by Wu *et al.* [21], in order to be able to more effectively simulate larger scale deformable solid models.

Appendix

Class Hierarchy

The class hierarchy of GiPSi is shown in Fig. 7(a)–(g).

Acknowledgment

The authors would like to thank X. Wu for his valuable discussions and feedback.

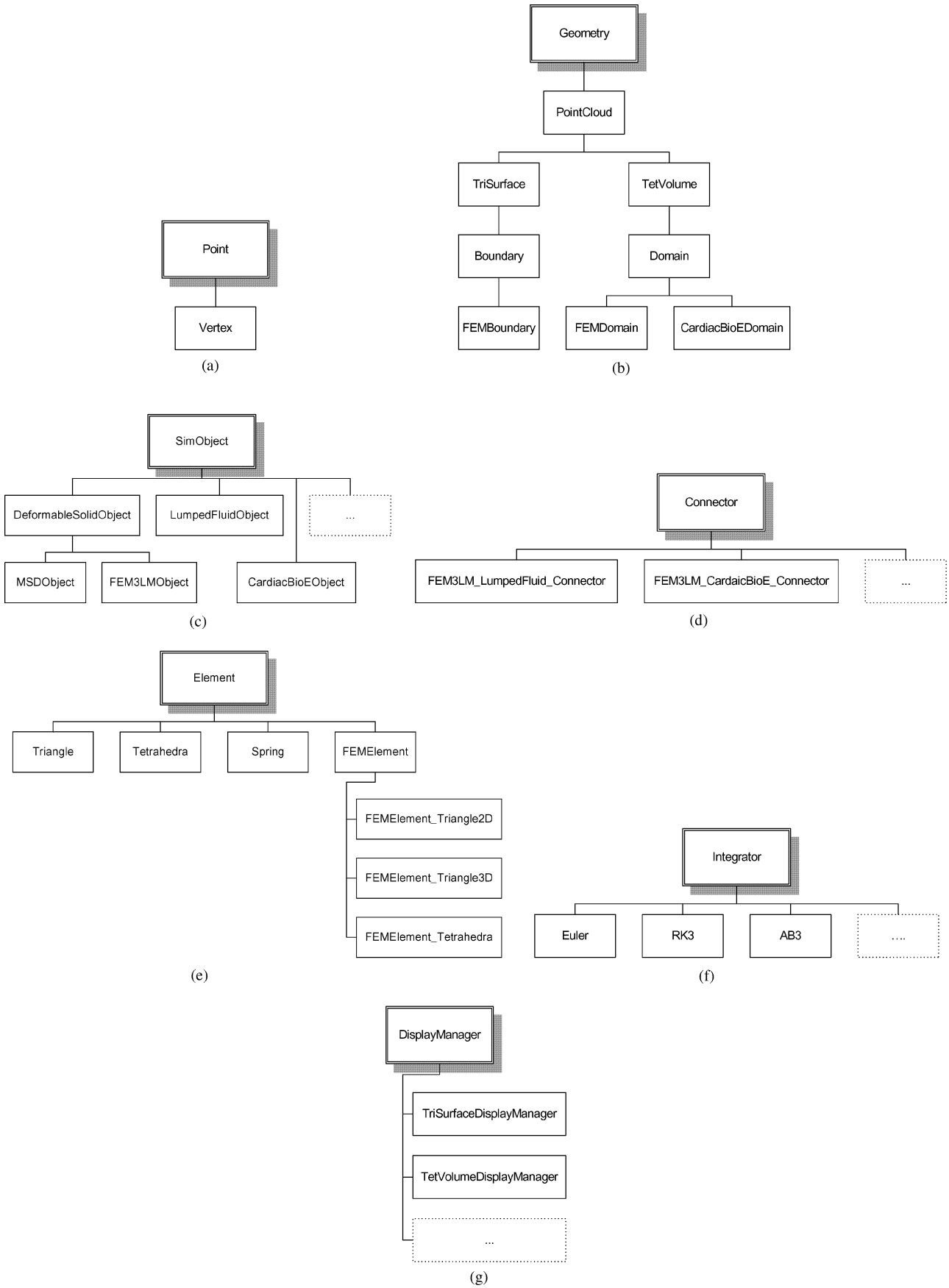

Fig. 7. GiPSi class hierarchies.

REFERENCES

[1] M. S. Downes, M. C. Çavuşoğlu, W. Gantert, L. W. Way, and F. Tendick, "Virtual environments for training critical skills in laparoscopic surgery," in *Proc. Medicine Meets Virtual Reality VI (MMVR'98)*, Jan. 1998, pp. 316–322.

[2] 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*, vol. 9, no. 3, pp. 236–255, Jun. 2000.

[3] M. C. Çavuşoğlu, F. Tendick, and S. Sastry, "Telesurgery and surgical simulation: Haptic interfaces to real and virtual surgical environments," in *Touch in Virtual Environments*, Ser, IMSC Series in Multimedia, M. L. McLaughlin, J. P. Hespanha, and G. Sukhatme, Eds. Englewood Cliffs, NJ: Prentice-Hall, 2001, pp. 217–237.

[4] X. Wu, M. S. Downes, T. Goktekin, and F. Tendick, "Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes," in *Proc. EUROGRAPHICS 2001*, Sep. 2001.

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

[6] (1996). Simulink. [Online]. Available: http://www.mathworks.com/ products/simulink/

[7] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simul.*, vol. 4, no. 2, pp. 155–182, 1994.

[8] 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 *Medicine Meets Virtual Reality (MMVR 2002)*. J. Westwood, *et. al.* Eds. Amsterdam, The Netherlands: IOS, 2002, pp. 296–303.

[9] A. Joukhadar and C. Laugier, "Dynamic simulation: model, basic algorithms, and optimization," in *Algorithms For Robotic Motion and Manipulation*: J. P. Laumond and M. Overmars, Eds. Wellesley, MA: A. K. Peters, 1997, pp. 419–434.

[10] 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 *Proc. SPIE Battlefield Biomedical Technology Conf.*, vol. 4037, 2000, pp. 294–300.

[11] Scientific Computing and Imaging Institute (SCI). SCIRun: A scientific computing problem solving environment, 2002. [Online]. Available: http://software.sci.utah.edu/scirun.html

[12] FEMLab Multiphysics Modeling Software, COMSOL, Inc. 2005. [Online]. Availavle: http://www.comsol.com/

[13] 3DSlicer. [Online]. Available: http://www.slicer.org/

[14] ITK insight segmentation and registration toolkit. [Online]. Available: http://www.itk.org/

[15] R. A. Banvard, "The visible human project® image data set from inception to completion and beyond," *Proc. CODATA 2002: Frontiers of Scientific and Technical Data,* Track I-D-2, Medical and Health Data, Montreal, ON, Canada, Oct. 2002.

[16] R. M. Beme and M. N. Levy Eds. *Principles of Physiology*, 3rd. ed. St. Louis, MO: Mosby, 2000.

[17] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra , J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: SIAM, 1999.

[18] M. C. Çavuşoğlu and F. Tendick, "Multirate simulation for high fidelity haptic interaction with deformable objects in virtual environments," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA 2000)*, Apr. 2000, pp. 2458–2465.

[19] J. F. O'Brien and J. K. Hodgins, "Graphical modeling and animation of brittle fracture," in *Proc. 26th Annu. Conf. Computer Graphics and Interactive Techniques*, 1999, pp. 137–146.

[20] X. Wu and F. Tendick, "Multigrid integration for interactive deformable body simulation," in *Proc. Int. Symp. Medical Simulation (ISMS 2004),* Lecture Notes in Computer Science, vol. 3078, New York: Springer-Verlag, 2004, pp. 92–104.

[21] X. Wu, T. G. Goktekin, and F. Tendick, "An interactive parallel multigrid FEM simulator," *Proc. Int. Symp. Medical Simulation (ISMS 2004),* Lecture Notes in Computer Science, vol. 3078, New York: Springer-Verlag, 2004, pp. 124–133.

M. Cenk Çavuşoğlu (S'92–M'01) received the B.S. degree in electrical and electronic engineering from the Middle East Technical University, Ankara, Turkey, in 1995, and the M.S. and Ph.D. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 1997 and 2000, respectively.

He was a Postdoctoral Researcher and Lecturer at the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, from 2000 to 2002. He is now an Assistant Professor of electrical engineering and computer science at Case Western Reserve University, Cleveland, OH.

His research interests include robotics (medical robotics and haptics), virtual environments and computer graphics (surgical simulation and physical modeling), and systems and control theory.

Dr. Çavuşoğlu is an Associate Editor of the IEEE TRANSACTIONS ON ROBOTICS.

Tolga G. Göktekin received the B.S. degree in computer engineering from Bogazici University, Istanbul, Turkey, in 1994, and the M.S. degree in computer science from the University of Florida, Gainseville, in 1997. He is currently a Ph.D. candidate at the University of California, Berkeley.

He is a Technical Director Trainee at Pixar Animation studios. His research interests are physically based computer animation, medical simulations, and virtual environments for training and control.

Frank Tendick (S'85–M'94) received the B.S. degree in aeronautics and astronautics from the Masschusetts Institute of Technology, Cambridge, the M.S. degree in mechanical engineering from the University of California, Berkeley, and the Ph.D. degree from the Joint Graduate Group in Bioengineering at the University of California, San Francisco (UCSF) and Berkeley campuses.

He is currently an Assistant Professor and Director of the Surgical Skills Center at UCSF. His interests are in human–machine interfaces, teleoperation, and virtual environments, especially as applied to surgery and medicine.