SOFA, a Multi-Model Framework for Interactive Physical Simulation

François Faure and Christian Duriez and Hervé Delingette and Jérémie Allard and Benjamin Gilles and Stéphanie Marchesseau and Hugo Talbot and Hadrien Courtecuisse and Guillaume Bousquet and Igor Peterlik and Stéphane Cotin

Abstract SOFA (Simulation Open Framework Architecture) is an open-source C++ library primarily targeted at interactive computational medical simulation. SOFA facilitates collaborations between specialists from various domains, by decomposing complex simulators into components designed independently and organized in a scenegraph data structure. Each component encapsulates one of the aspects of a simulation, such as the degrees of freedom, the forces and constraints, the differential equations, the main loop algorithms, the linear solvers, the collision detection al-

François Faure
University Joseph Fourier, INRIA, LJK-CNRS, Grenoble e-mail: Francois.Faure@imag.fr

Christian Duriez
INRIA, University of Lille e-mail: Christian.Duriez@inria.fr

Hervé Delingette
INRIA, Sophia-Antipolis e-mail: Herve.Delingette@inria.fr

Jérémie Allard
INRIA, University of Lille e-mail: Jeremie.Allard@inria.fr

Benjamin Gilles
INRIA, Montpellier e-mail: Benjamin.Gilles@inria.fr

Stéphanie Marchesseau
INRIA, Sophia-Antipolis e-mail: Stephanie.Marchesseau@inria.fr

Hugo Talbot
INRIA, University of Lille e-mail: Hugo.Talbot@inria.fr

Hadrien Courtecuisse
INRIA, University of Lille e-mail: Hadrien.Courtecuisse@inria.fr

Guillaume Bousquet
University Joseph Fourier, INRIA, LJK-CNRS, Grenoble e-mail: Guillaume.Bousquet@inria.fr

Igor Peterlik
INRIA, University of Lille e-mail: Igor.Peterlik@inria.fr

Stéphane Cotin
INRIA, University of Lille e-mail: Stephane.Cotin@inria.fr

gorithms or the interaction devices. The simulated objects can be represented using several models, each of them optimized for a different task such as the computation of internal forces, collision detection, haptics or visual display. These models are synchronized during the simulation using a mapping mechanism. CPU and GPU implementations can be transparently combined to exploit the computational power of modern hardware architectures. Thanks to this flexible yet efficient architecture, SOFA can be used as a test-bed to compare models and algorithms, or as a basis for the development of complex, high-performance simulators.

1 Introduction

Programming interactive physical simulations of rigid and deformable objects requires multiple skills in geometric modeling, computational mechanics, numerical analysis, collision detection, rendering, user interface and haptics feedback, among others. It is also challenging from a software engineering standpoint, with the need for computationally efficient algorithms, multi-threading, or the deployment of applications on modern hardware architectures such as the GPU. The development of complex medical simulations has thus become an increasingly complex task, involving more domains of expertise than a typical research and development team can provide. The goal of SOFA is to address these issues within a highly modular yet efficient framework, to allow researchers and developers to focus on their own domain of expertise, while re-using other expert's contributions.

SOFA introduces the concept of scenegraph-based multi-model representation to easily build simulations composed of an arbitrary number of objects. The pool of simulated objects and algorithms used in a simulation (also called a scene) is described using a hierarchical data structure similar to scenegraphs used in graphics libraries. The simulated objects are decomposed into collections of independent components, each of them describing one feature of the model, such as state vectors, mass, forces, constraints, topology, integration scheme, and solving process. As a result, switching from internal forces based on springs to a finite element approach can be done by simply replacing one component with another, all the rest (mass, collision models, time integration, ...) remaining unchanged. Similarly, it is possible to keep the same solver and modify other components to compute the forces on the GPU instead of the CPU. Moreover, the simulation algorithms, embedded in components, can be customized with the same flexibility as the physical models.

In addition to this first level of modularity, it is possible to go one step further and decompose simulated objects into a set of specialized models, each optimized for a given type of computation. A physical object in SOFA is typically described using three models: an internal model with the independent degrees of freedom (DOFs), the mass and the constitutive laws, a collision model with contact geometry, and a visual model with detailed geometry and rendering parameters. Each model can be designed independently of the others, and more complex combinations are possible, for instance for the coupling of two different physical objects. During run-time, the

models are synchronized using a generic mechanism called *mapping* to propagate forces and displacements.

SOFA was first released in 2007 [1]. Since then, it has evolved toward a comprehensive, high-performance library used by an increasing number of academics and commercial companies.

This chapter is organized as follows. Section 2 introduces the multi-model framework of SOFA and the component-based architecture. Section 3 details the data structures used to represent complex scenes, and the way data is stored and propagated. The high-level simulation algorithms, including ODE solution and collision detection algorithms, are presented in Section 4, while input-outputs and the user interface are sketched in Section 5. Complex simulation examples are shown in Section 6, and a conclusion and perspectives are briefly drawn is Section 7.

2 Multi-model representation

Consider the deformable model of a liver shown in the left of Figure 1. It is surrounded by different anatomical structures (including the diaphragm, the ribs, the stomach, the intestines, etc.) and it is also in contact with a grasper (modeled as an articulated rigid chain). In SOFA, this liver can be simulated using three different

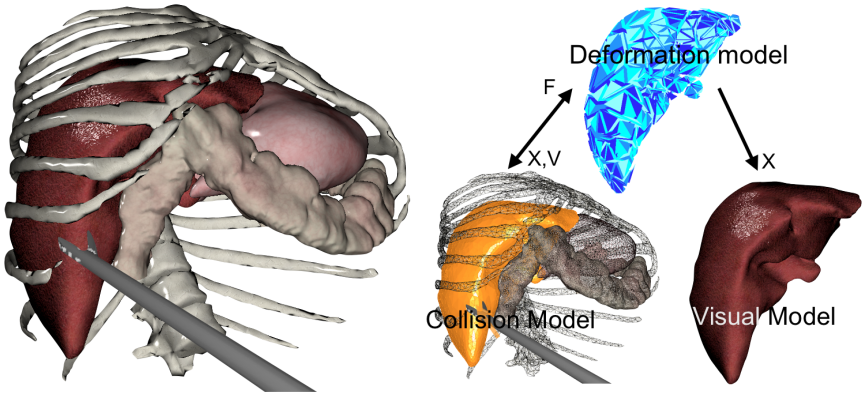

Fig. 1 A simulated Liver. Left: The liver displayed in its environment. Right: Three representations are used for the liver: one master model for the internal deformable mechanics, one for the collisions, and one for the visualization. Mappings (black arrows) are used to propagate positions (X) and velocities (V) from master to slaves, while forces (F) are propagated in the opposite direction.

models. The first is used to represent its internal mechanical behavior, which may be computed using Finite Element Method (FEM) or other models. The geometry of this model is optimized for the computation of internal forces, typically using a reduced number of well-shaped tetrahedra for speed and stability. However, the best trade-off between precision and speed in collision detection may require another

geometrical model, while the realistic visualization certainly requires a smoother and more detailed geometry. We thus use a second model for collision detection and response, while a third one is dedicated to the visual rendering process. This section presents these models and their connections.

2.1 Solid Mechanics

The deformable solid continuum shown in Figure 2 is modeled using a dynamic or quasi-static system of particles (also called simulation nodes). The node coordinates are the independent DOFs of the object, and they are typically governed by equations of the following type:

$$\mathbf{a} = \mathbf{P}\mathbf{M}^{-1}\sum_i \mathbf{f}_i(\mathbf{x},\mathbf{v}) \tag{1}$$

where $\mathbf{x}$ and $\mathbf{v}$ are the position and velocity vectors, the $\mathbf{f}_i$ are the different force functions (volume, surface and external forces in this example), $\mathbf{M}$ is the mass matrix and $\mathbf{P}$ is a projection matrix to enforce boundary conditions on displacements. Note that the modeling of rigid body dynamics leads to the same type of equations.

The corresponding model in SOFA is a set of components connected to a common scenegraph node. Scenegraph nodes, not to be misunderstood as simulation nodes, are discussed in Section 3. Each component is responsible for a reduced set of tasks implemented using virtual functions in an object-oriented approach. Each

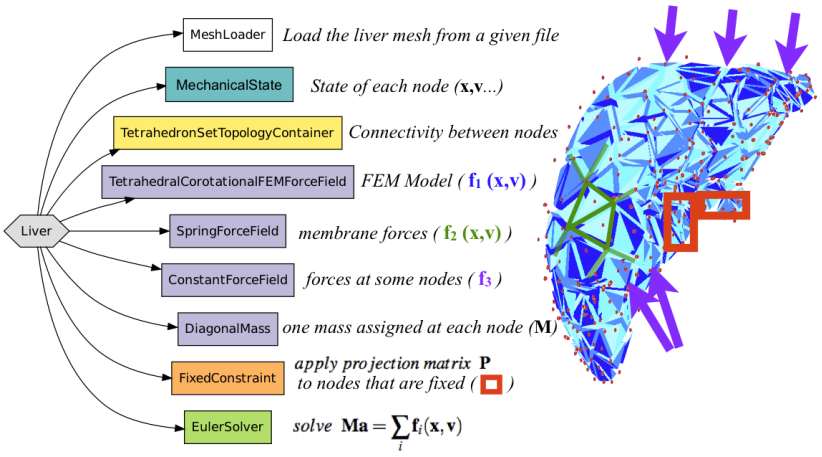

Fig. 2 Mechanical model of a liver. Boxes highlight fixed particles, while arrows denote external forces. In order to facilitate the combination of models and algorithms, the liver is described as a composition of specialized components.

operator in Equation 1 corresponds to a component. *MeshLoader* is used to read the topology and the geometry from a file. The coordinate vector $\mathbf{x}$ of the mesh nodes and all the other state vectors (velocity $\mathbf{v}$, net force $\sum \mathbf{f}$, etc.) are stored in a *MechanicalState*, which is the core component of the mechanical model. The tetrahedral connectivity is stored in *TetrahedronSetTopologyContainer*, and made available to other components such as *TetrahedralCorotationalFEMForceField*, which accumulates force based on the Finite Elements. An arbitrary number of force functions can be attached to the scenegraph node, such as *SpringForceField*, which accumulates the forces generated by the external surface membrane, and *ConstantForceField*, which accumulates external forces to a given subset of simulation nodes (for instance the pressure exerted by the diaphragm on the liver). *DiagonalMass* is used to implement the product with matrix $\mathbf{M}^{-1}$. *FixedConstraint* implements the product with matrix $\mathbf{P}$ to cancel the displacements of the particles depicted in squares in the figure. *EulerSolver* implements the logic of time integration. In this example, the connections between the components need not be represented explicitly. Each component can query its parent node to get access to the local *MechanicalState* and topology. High level algorithms, such as time integration, are implemented using visitors traversing the data structure, as explained in sections 3 and 4.

This design is highly modular because the components are completely independent of each other. In the example of Figure 2, replacing springs with FEM for the membrane force only requires to replace *SpringForceField* with *TriangleFEMForceField*. Similarly, the mass matrix, stored as diagonal matrix in this example, can be stored as a single scalar value (*UniformMass*) if less accuracy but faster computation is sought, in combination with an iterative implicit solver for instance.

For efficiency, the *MechanicalState* contains the state vectors of all the simulation nodes of the object, to avoid multiple calls of virtual functions. The vector size is the number of nodes, and each vector entry has the size of the node type, such as 3 for 3d particles. We use C++ templates to avoid code redundancy between scalar types (float, double) and between node types (particles or frames, in 1d, 2d or 3d, or generalized coordinates). In this document, the type instances are shown in the scenegraph figures when necessary, and omitted most of the time. All the nodes in a vector have the same type, known at compile time, to allow aggressive compiler optimizations. Simulation nodes of different types must be gathered in different *MechanicalStates* attached to different scenegraph nodes, possibly connected with interaction forces, as discussed in Section 3.

More than 30 classes of forces are implemented in SOFA, including springs, FEM for volumetric (tetrahedron or hexahedron) or surface (triangular shell and membrane) deformable objects using corotational or hyperelastic formulations, and for wire or tubular object (beam models meshed with segments), have been implemented. Different types of elastic forces allow for easy and fast modeling of the deformations (bending, compression/traction, volume, interactions between two bodies, joints...). In rigid objects, the main components are the degrees of freedom (a single frame with 3 rotations and 3 translations) and the mass matrix that contains the inertia of the object. Surfaces can be attached to objects using *mappings*, as discussed in Section 2.5.

2.2 Other physical models

Research on interactive medical simulation is often limited to (bio-) mechanical aspects. However, an important step needs to be accomplished to better capture the physiology of the patient. This involves integrating into the simulation more information relevant to the procedure, which can be of different nature such as electrical or fluid.

In order to simulate fluids in a free environment (like blood leaks) Smoothed-Particle Hydrodynamics (SPH) models can be employed. This Lagrangian model is similar to the deformable models described previously: a set of particles, with a given mass, are linked by a force function. The method is mesh-free: at each step of the simulation, particles are grouped by neighborhoods and attraction-repulsion forces are computed between them. As the method relies on particles, the coupling with deformable model can be easily done by repulsion forces (see Figure 3). More advanced fluid models, based on Eulerian approaches, were also implemented in SOFA (see [29] for instance).

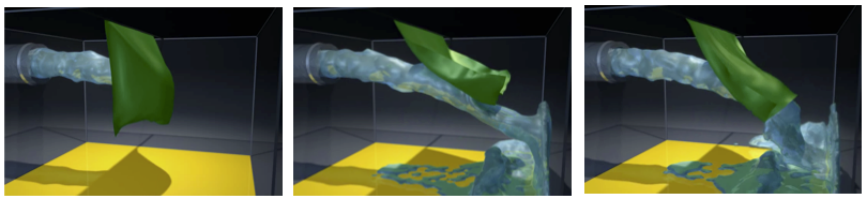

Fig. 3 Simulation of interaction between a fluid model and a cloth model in SOFA.

The flexible structure of Sofa allows the simulation of non-mechanical phenomena such as electrical waves in cardiac electrophysiology. While this topic is a work in progress in Sofa, different ways of modeling these electrical waves have been implemented, namely with an eikonal approach and with monodomain cell models, see Section 6.1.

When simulating the evolution of a physical field defined on a mesh, the state vectors of the electric potential are placed in a new component named *ElectricalState* (see Figure 4). A further extension of the framework has been to allow the coupling between mechanical and physical model evolution.

2.3 Collision models

When a lot of primitives come into contact, collision detection and response can become the bottleneck of a simulation. Several collision detection approaches have been implemented: distances between pairs of geometric primitives (triangles and spheres), points in distance fields, distances between colliding meshes using ray-

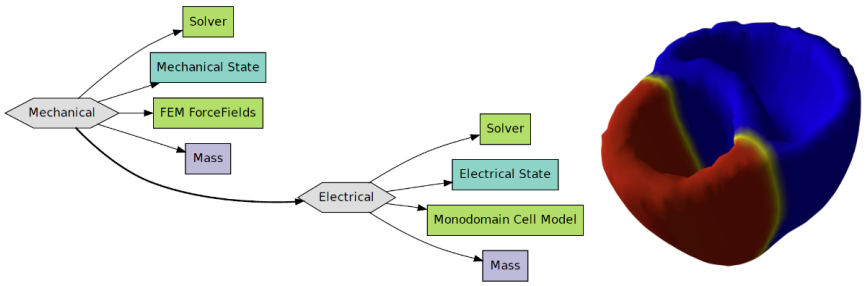

Fig. 4 Electrophysiology model: a dedicated component implements the propagation of the potential field, based on the model of Monodomain Cell.

tracing [18], and intersection volume using images [3]. The collision pipeline is described in section 4.4 with more detail.

In order to adapt the models to the data structure of the different collision algorithms, a separate *collision model* is employed. This model is similar to an internal model, except that its topology and its geometry are of its own and can be stored in a data structure dedicated to collision detection. For instance, the component *TriangleModel* is the interface for the computation of collision detection on a triangular mesh surfaces.

If collision detection takes too much time, or if we wish to model contacts using a small number of points, the collision mesh can be set coarser than the internal mesh. Conversely, if precise collision detection and response between detailed surfaces is needed, it is sometimes suitable to use more detailed mesh for collision detection.

2.4 Visual models

In the context of surgical simulation for training, to reach the state of what is often called *suspension of disbelief* i.e. when the user forgets that he or she is dealing with a simulator, there are other factors than the mechanical behavior. Realistic rendering is one of them. It involves visually recreating the operating field with as much detail as possible, as well as reproducing visual effects such as bleeding, smoke, lens deformation, etc. The main feature of the visual model of SOFA is that the meshes used for the visualization can be different from the models used for the simulation. The mappings described in section 2.5 maintain the coherency between them. Hence, SOFA simulation results can easily be displayed using models much more detailed than used for internal mechanics. They can also be rendered using external libraries such as OGRE[1] and Open Scene Graph[2].

[1] www.ogre3d.org

[2] www.openscenegraph.org

We have also implemented our own rendering library based on openGL. This library allows for modeling and render the visual effects that occurs during an intervention or the images that the surgeon is watching during the procedure. For instance, in the context of interventional radiology simulator, we have developed a dedicated interactive rendering of X-ray and fluoroscopic images.

2.5 Mappings

As previously discussed, objects simulated in SOFA, like the liver in Figure 1, typically rely on several models: one for the internal model, one for collision, and one for the visual rendering. To enforce consistency, one of them, typically the internal model, acting as the master, imposes its displacements to slaves (typically the visual model and the collision model), using *mappings*. Mapped model can be masters of other models in turn, creating a hierarchy whith the independent DOFs at the root. Figure 5 illustrates the hierarchies of two objects. The visual models, in additional branches, are omitted for clarity. The independent DOF of the objects, on top, are the masters of contact models based on triangle vertices. When the contact models collide, pairs of contact points are created, each point a slave of a contact model.

Let $\mathscr{J}$ be the function used to map the positions $\mathbf{x}_m$ of a master model to the positions $\mathbf{x}_s$ of a slave:

$$\mathbf{x}_s = \mathscr{J}(\mathbf{x}_m) \tag{2}$$

The velocities are mapped in a similar way:

$$\mathbf{v}_s = \mathbf{J}\mathbf{v}_m \tag{3}$$

The Jacobian matrix $\mathbf{J} = \frac{\partial \mathbf{x}_s}{\partial \mathbf{x}_m}$ encodes the linear relation between the master and slave velocities. Accelerations can be mapped using:

$$\mathbf{a}_s = \mathbf{J}\mathbf{a}_m + \frac{\partial \mathbf{J}}{\partial \mathbf{x}_m}\mathbf{v}_m \tag{4}$$

In linear mappings, operators $\mathscr{J}$ and $\mathbf{J}$ are the same, otherwise $\mathscr{J}$ is nonlinear with respect to x_m and it can not be written as a matrix. For surfaces embedded in deformable cells, matrix $\mathbf{J}$ contains the barycentric coordinates. For surfaces attached to rigid bodies, each row of the matrix encodes the usual relation $v = \dot{o} + \omega \times (x - o)$ for each vertex.

The positions and the velocities are propagated top-down in the hierarchy. Conversely, the forces are propagated bottom-up to the independent DOFs, where Newton's law $\mathbf{f} = \mathbf{M}\mathbf{a}$ is applied. Given forces $\mathbf{f}_s$ applied to a slave model, the mapping computes and accumulates the equivalent forces $\mathbf{f}_m$ applied to its master. Since equivalent forces must have the same power, the following relation holds:

$$\mathbf{v}_m^T \mathbf{f}_m = \mathbf{v}_s^T \mathbf{f}_s$$

The kinematic relation $\mathbf{v}_s = \mathbf{J}\mathbf{v}_m$ allows us to rewrite the previous equation as

$$\mathbf{v}_m^T \mathbf{f}_m = \mathbf{v}_m^T \mathbf{J}^T \mathbf{f}_s$$

Since this relation holds for all possible velocities $\mathbf{v}_m$, the principle of virtual work allows us to simplify the previous equation to obtain:

$$\mathbf{f}_m = \mathbf{J}^T \mathbf{f}_s \tag{5}$$

When a model has several slaves, each slave accumulates its contribution to the forces on the master using its mapping. This hierarchical kinematic model allows us to compute displacements and to apply forces at all levels. So far, 22 variants of mappings have been implemented to attach models to rigid objects and deformable primitives such as tetrahedra, hexahedral grids, splines, blended frames, flexible beams and scalar fields. Mappings are also used to connect generalized coordinates, such as joint angles, to world-space geometry, as in the grasper of Figure 5.

3 Data structure

The organization of simulation data is a complex issue. We have identified three relevant levels, and proposed different solutions for each of them. The main structure is a scenegraph, used to hierarchically organize the groups of objects and their different models (Section 3.1). Additionally, a network of dependencies between component attributes can be created (Section 3.2). Finally, the geometrical models and the topological changes deserve a special attention (Section 3.3).

3.1 Scenegraph and visitors

The main structure of the scene is defined by the scenegraph nodes, which serve different purposes. They are used to gather the components associated with the same DOFs or topology. DOFs connected by a mapping within a kinematic hierarchy must be located in different nodes, and the master must be placed as parent or higher than the slave in the hierarchy, to ensure that it is traversed first during visitor top-down traversals. Scenegraph nodes can also be used to represent arbitrarily nested object groups, with group elements set as children of the group node. To apply a simulation algorithm, implemented in a high level component (Section 4), to a list of objects, it is necessary to gather the objects in a group and to attach the component at the root of the group or higher in the hierarchy. The nesting of sub-groups does not impact the behavior of the algorithms, as long as masters are higher than slaves in the hierarchy. In the example shown in Figure 6, the scene contains two objects animated using different time integrators, collision detection components (discussed in Section 4.4), an interaction force, and a camera to display the objects. The root

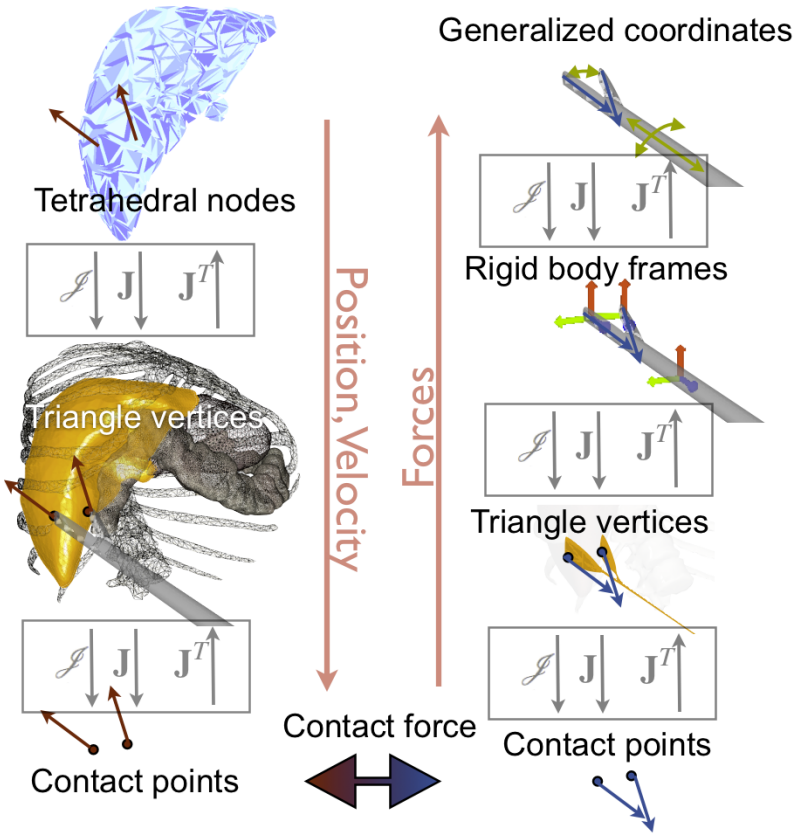

Fig. 5 Mappings between the DOFs and the contact points. Left (top to bottom): the internal model of the liver is based on Finite Element model. A triangular mesh is mapped for collision detection with the surface. The two contact points found by the collision detection (with the grasper) are mapped on the collision model. Right (bottom to top): the contact points are also mapped on the collision model of the grasper. This collision model is a simplification of the grasper shape and is mapped on the rigid body frames. The motion of these frame is mapped on the state of the joints which are the independent DOFs of the grasper.

node represents the whole simulation. It contains the two simulated objects, each in a child node, and components applied to these objects. The rigid object node contains the independent degrees of freedom of the rigid object, a single moving frame in this case, and the components which process the associated state vectors (positions, forces, *etc.*), here only the mass. Collision spheres are attached to the rigid body using a *RigidMapping* called *sphereMapping*, as illustrated in Figure 8. A child node is required for the sphere centers, first because they are not independent DOFs, but also due to the different types, frame and points. The deformable object is based on a single set of simulation nodes, thus only one scenegraph node is necessary to model it.

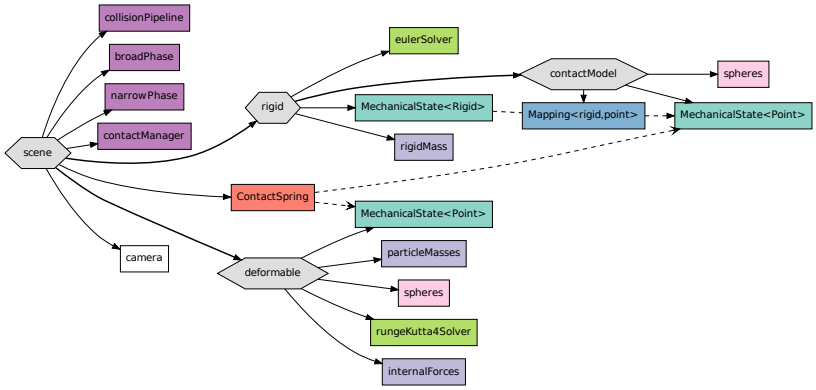

Fig. 6 A scenegraph with collision detection and two independent objects interacting through a spring. Graph nodes and components are represented by hexahedra and boxes, respectively. The plain arrows represent the scenegrap structure, while dashed lines are pointers between components attached to different nodes.

Connections between non-sibling components such as mappings or inter-object force fields require explicit references, shown as dashed arrows in Figure 6. Components shared by two objects are attached to their common group node. Most of them process the children by sending visitors. However, since the interaction force field is specific to a given pair of objects, it requires pointers to their *MechanicalStates*.

The data structure is processed using visitors, discussed below, which apply virtual functions to each node they traverse, which in turn apply virtual functions to the components they contain. In simple scenegraph frameworks, the visitors are only fired from an external control structure such as the main loop of the application. In SOFA, the components are allowed to suspend the current traversal to send an arbitrary number of other visitors, then to resume or to prune the suspended visitor. This allows us to implement global algorithms (typically ODE solution or collision detection), such as the explicit Euler velocity update of Equation 1, in components which fire lower-level visitors. The visitors are implemented in separate classes which are available to all the components. The scenegraph-visitor approach neatly decouples the physical model from the simulation algorithms, in sharp contrast with dataflow graphs which intricate data and algorithms in the same graph. Replacing a time integrator requires the replacement of one component in our scenegraph, whereas the corresponding dataflow graph would have to be completely rewritten.

Interactions between objects can be handled using penalty forces or Lagrange multipliers. In all cases, a component connected to the two objects is necessary to geometrically model the contact and compute the interaction forces. This shared component is located in their common ancestor node. The coupling created by penalty forces should be considered soft or stiff, depending on the stiffness and the size of time step [4]. A soft coupling can be modeled by an interaction force constant

during each time step. In this case, each object can be animated using its own, possibly different, ODE solver. The assumption of constant interaction force during each time step is compatible with all explicit time integration methods. However, when the interaction forces are stiff, implicit integration is necessary to apply large time steps without instabilities. This requires the solution of an equation system involving the two objects as well as their interaction force. In this case, the ODE solver is placed in the common ancestor node, at the same level as the interaction component. This is also true for constraint-based interaction which requires the computation of Lagrange multipliers based on interaction Jacobians. Due to the superlinear time complexity of equation solvers, it is generally more efficient to process independent interaction groups using separated solvers rather than a unique solver.

We implement the simulation using **visitors** which traverse the scene top-down and bottom-up, and call the corresponding virtual functions at each graph node traversal. A possible implementation of the traversal of a tree-like graph is shown in the left of Figure 7. Algorithmic operations on the simulated objects are implemented by deriving the Visitor class and overloading its virtual functions *topDown( )* and *bottomUp( )*. This approach hides the scene structure (parent, children) from the components, for more implementation flexibility and a better control of the execution model. Moreover, various parallelism strategies can be applied independently of the mechanical computations performed at each node. The data structure is actually extended from strict hierarchies to directed acyclic graphs to handle more general kinematic dependencies. The top-down node traversals are pruned unless all the parents of the current node have been traversed already, so that nodes with multiple parents are traversed only once all their parents have been traversed. The bottom-up traversals are made in the reverse order.

```
void Visitor::traverse(Node n)
bool continue = this.topDown( n )
if continue then
   for all c child of n do
      this.traverse( c )
   end for
   this.bottomUp( n )
end if
```

```
bool AnimateVisitor::topDown(Node n)
if n.animationLoop then
   n.animationLoop.animate(this.dt)
   return false
end if
if n.collisionPipeline then
   n.collisionPipeline.modelContacts()
end if
if n.odeSolver then
   n.odeSolver.solve(this.dt)
   return false
end if
for all InteractionForce n.f do
   n.f.apply()
end for
return true
```

Fig. 7 Left: a recursive implementation of the visitor traversal. Right: the *AnimateVisitor*.

An example of visitor is *AnimateVisitor*, whose traversal method triggers forward time stepping, as shown in the right of Figure 7. Applied to the simple scene

in Figure 8, it triggers the ODE solver, which in turn applies its algorithm using visitors for mechanical operations such as propagating states through the mappings or accumulating forces. Note that the traversal of the *AnimateVisitor* is pruned when an ODE solver is encountered. This allows the ODE solver to take control of its subgraph, overriding the solvers lower in the hierarchy. In the more complex scene shown in Figure 6, the *AnimateVisitor* triggers the collision detection, which may create a contact between the chidren, such as *contactSpring*. The visitor then triggers the computation of the interaction force, which will be seen by the objects as a constant, external force during the time step. The visitor then continues the traversal and triggers each object ODE solver. The default behavior is to model the contacts prior to applying time integration. To implement other strategies, an *AnimationLoop* can be used to prune the visitor and apply time integration and collision detection in a different order, possibly looping until all collisions are solved.

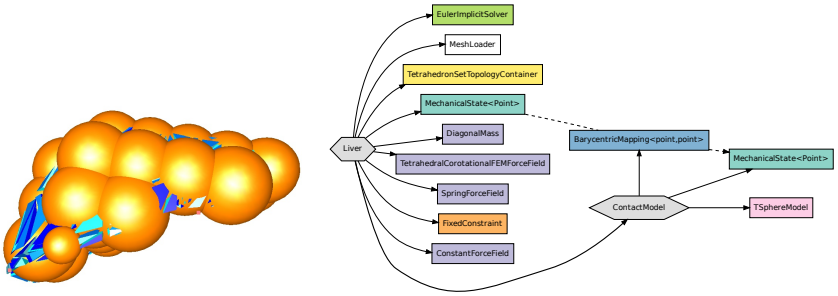

Fig. 8 Left: simple internal (blue) and collision (yellow) models of a liver. Right: the corresponding scenegraph. The plain arrows denote hierarchy, while the stippled arrows represent connections.

3.2 Data and Engines

Component parameters are stored in member objects using *Data* containers, templated on the type of attribute they represent. For instance, the list of particle indices constrained by a FixedConstraint is stored in a Data< vector<unsigned> >. These containers provides a reflective API, used for serialization in XML files and the automatic creation of input/output widgets in the user interface, as discussed in Section 5. Additionally, we can create connections between Data instances to keep their value synchronized. This is used for instance when a *Loader* component loads several attributes from a file (such as topology, positions, stiffnesses, boundary conditions) which are then connected to one or more components using it as input. In some cases we need to not simply copy an existing value but compute it from one or several others. This feature is provided by *Engine* components. Engines contain input

and output Data, and their update method computes the output based on the input. A mechanism of lazy evaluation is used to recursively flag Data values that are not up-to-date, but they are recomputed only when necessary. For instance, based on a bounding box and a vector of coordinates, a BoxROI engine computes the list of indices of the coordinates inside the box. These indices can then be used as input of a FixedConstraint to define a fixed boundary condition. With this design, the simulation can transparently be setup either from data stored in static files, or generated automatically with engines.

The network of interconnected Data objects defines a data dependency graph, superimposed on the scenegraph. This two-graph framework has been used in other graphics software such as OpenInventor and Maya, where the engines are used to generate the animation, by periodically updating the state vectors using time as input, while the scenegraph represents the frame hierarchy. This approach works well for straightforward animation pipelines, such as keyframe interpolation, but it does not easily allow the branching and loop control structures used in more sophisticated physical simulation algorithms. It is also a rather low-level representation, essentially encoding every computation steps required to compute a given Data. Consequently, we only use engines to implement straightforward relations between the parameters of the model, which may remain unchanged during the simulation. In SOFA, the state update algorithms are implemented in components communicating using scenegraph visitors, as explained in Section 4.

3.3 Topology and Geometry

While mesh geometry describes the location of mesh vertices in space, mesh topology indicates how vertices are connected to each other by edges, triangles or any type of mesh element. Both information are required on a computational mesh to perform mesh visualization, mechanical modeling, collision detection, haptic rendering, scalar or vectorial field description. We consider meshes that are cellular complexes made of k-simplices (triangulations, tetrahedralisation) or k-cubes (quad or hexahedron meshes). These meshes are the most commonly used in real-time surgery simulation and can be hierarchically decomposed into k-cells, edges being 1-cells, triangles and quads being 2-cells, tetrahedron and hexahedron being 3-cells. To take advantage of this feature, the different mesh topologies are structured as a family tree (see Fig. 9) where children topologies are made of their parent topology.

This hierarchy makes the design of simulation components very versatile since a component working on a given mesh topology type will also work on its derived types. For instance a spring-mass mechanical component only requires the knowledge of a list of edges (an *EdgeSetTopology* as described in Fig. 9) to be effective. With this design, a spring-mass component can be used at no additional cost on triangulation or hexahedral meshes that derive from an *EdgeSetTopology* mesh.

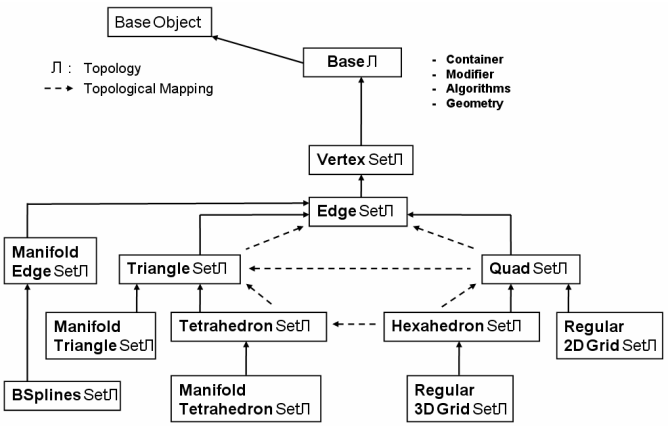

Fig. 9 Hierarchy of mesh topology. Dashed arrows indicate possible *Topological Mappings* from a topology object to another.

Topology objects are composed of four functional members(*Container*, *Modifier*, *Geometry* and *Algorithms*) to create, modify the topologies arrays, or give access to geometrical and adjacent information.

Another important concept introduced in SOFA is the notion of *Topological Mapping*. Those mappings define a mesh topology from another mesh topology using the same DOFs. These topologies will therefore be assigned to the same *MechanicalState*. For instance, one may need to apply specific forces on the surface bounding a volume (for instance to model the Glisson capsule surrounding the liver parenchyma). In this context, a *Tetra2TriangleTopologicalMapping* may be used to generate in a subnode (for instance the node *Surface* in Figure 10) the list of triangles on the border of a tetrahedral surface. Similarly, one may obtain the set of edges bordering a triangular mesh or the set of quads at the surface of an hexahedral mesh. Topological mapping may also be used to split topological cells into other types of cells. Thus, a quad may be split into 2 triangles and an hexahedron into 5 or 6 tetrahedra. Specific mapping components exist to create a tetrahedral mesh from a set of hexahedra or to create triangular meshes from quads.

Mesh Data Structure

Specific data structure storing mesh information (material stiffness, list of fixed vertices, nodal masses, ...) are stored in *components* and are spread out in the simulation tree. They consist of simple arrays with contiguous memory storage and a

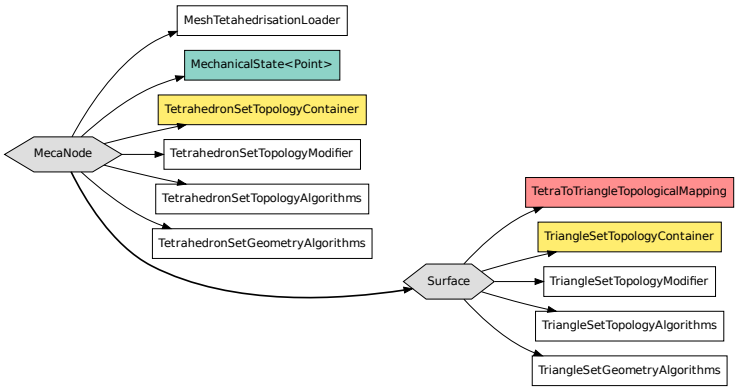

Fig. 10 Scenegraph of a simple tetrahedral mesh where a topological mapping is defined to have the set of triangles located at the surface of the volumetric mesh, with their DOF in the parent scene node.

short direct access time. This is important for real-time simulation, but bears some drawbacks when elements of these arrays are being removed since it entails the renumbering of elements. Fortunately, all renumbering tasks that maintain consistent arrays can be automated and hidden to the user when topological changes in the mesh arise. Therefore, efficient access of mesh data structures is granted while the complexity of keeping the arrays consistent with topological changes is automated.

There are as many specific data structures as topological elements, currently: vertices, edges, triangles, quads, tetras, hexas. These containers are similar to the STL *std::vector* classes and allow one to store any component-related data structure. A typical implementation of spring-mass models would use an edge container that stores for each edge, the spring stiffness and damping value, the i^{th} element of that container being implicitly associated with the i^{th} edge of the topology.

4 Simulation Algorithms

While typically most components in a scene implement low-level methods implying a small number of other components, such as accumulating force or mapping state vectors, some of them perform more abstract operations to implement simulation algorithms applied to arbitrary scenes, by overloading visitor traversals and firing their own visitors. These include ODE integration, linear equation solution, complex constraints, and collision detection, and may also involve components implemented on the GPU.

4.1 ODE solvers

ODE solvers implement animation algorithms applied at each time step to integrate time and compute positions and velocities one time step forward in time. The solvers do not directly address the physical models. Each state vector used by a solver (such as position or force) is actually scattered over all the *MechanicalStates* in the different scenegraph nodes in the scope of the solver. The state vectors are thus denoted by symbolic identificators, called VecIds. Each mechanical operation, such as allocating a state vector or accumulating the forces, is implemented using a specialized visitor parameterized on VecIds and on control values such as the time step. A given VecId uniquely identifies the corresponding state vector in each *MechanicalState*. This allows one to implement the solvers completely independenly of the physical model, as illustrated in the algorithm shown in Figure 11. This design avoids the

```
void ExplicitEulerSolver::solve(VecId x, VecId v, double dt)
create auxiliary vectors a,f
resetForce(f)
accumulateForce(f,x,v)
computeAcceleration(a,f)
project(a,a)
v += a * dt
x += v * dt
```

Fig. 11 Euler's explicit time integration. Each statement is implemented using a visitor.

assembly of global state vectors (i.e. copying Vec3 and quaternions to and from vectors of scalars). Moreover, the virtual function calls are resolved at the granularity of the state vectors (i.e. all the particles together, and all the moving frames together) rather than each primitive (i.e. each particle and each frame independently), and allow to optimize each implementation independently. There is thus virtually no loss of efficiency when mixing arbitrary types in the same simulation.

Explicit ODE solvers are variants of the Euler explicit solver presented in Figure 11, and are easily implemented in Sofa using the same operators. Implicit solvers, which consider the derivative at the end or somewhere in the middle of the time step, typically require the solution of equation systems such as:

$$\underbrace{(\alpha\mathbf{M}+\beta\mathbf{B}+\gamma\mathbf{K})}_{\mathbf{A}}\delta\mathbf{v}=\mathbf{b} \tag{6}$$

where $\mathbf{M}$ is the mass matrix, while $\mathbf{K}=\frac{\partial\mathbf{f}}{\partial\mathbf{x}}$ and $\mathbf{B}=\frac{\partial\mathbf{f}}{\partial\mathbf{v}}$ respectively are the stiffness and damping matrices (the method is explicit if β and γ are null). In order to apply simple displacement constraints, a projection matrix $\mathbf{P}$ can be used [4], and the system becomes $\mathbf{P}^T\mathbf{A}\mathbf{P}\delta\mathbf{v}=\mathbf{P}^T\mathbf{b}$. Implicit integration has the advantage of being more stable for stiff forces or large time steps. The solution of these equation systems requires linear solvers, discussed in the next section. Currently, eight ODE solvers

have been implemented, including symplectic Euler and explicit Runge-Kutta4, implicit Euler and statics solution.

4.2 Linear solvers

Conjugate Gradient An interesting feature of visitor-based mechanical computations is their ability to efficiently and transparently compute matrix products. Thus, we have proposed in SOFA an implementation of the Conjugate Gradient, based on the graph traversal. The visitor shown in Figure 12 computes the force change *df* based on a given displacement *dx*, as repeatedly performed in Conjugate Gradient algorithm. An arbitrary number of forces and projections may be present in all the nodes, resulting in a complicated stiffness matrix, as shown in the following equation:

$$\mathbf{df} = \sum_i \left(\prod_{j \in path(i)} \mathbf{J}_j \right)^T \mathbf{K}_i \left(\prod_{j \in path(i)} \mathbf{J}_j \right) \mathbf{dx} \tag{7}$$

where $\mathbf{K}_i$ is the stiffness matrix of force i, matrix $\mathbf{J}$ encodes the first-order mapping relation of a node with respect to its parent, and $path(i)$ is the list of mappings from the independent DOFs to the node the force applies to. This complex product is

```
bool ComputeDfVisitor::topDown():
dof.resetF(this.df)
if mapping then
   mapping.applyJ(this.dx)
end if
return true
```

```
void ComputeDfVisitor::bottomUp():
for all forceField F do
   F.addDF( this.df,this.dx )
end for
if mapping then
   mapping.applyJT(this.df)
end if
```

Fig. 12 Computing `df` given `dx` using a visitor. The top-down visitor propagates the given displacement and clears the force vectors, while the bottom-up visitor accumulates the forces and maps them up to the independent DOFs.

computed using only matrix-vector products and with optimal factoring thanks to the recursive implementation. It allows us to efficiently apply implicit time integration to arbitrary scenes using the Conjugate Gradient, and to trade-off accuracy for speed by limiting the number of steps of the iterative solution.

Direct Solvers Direct solvers are also available in SOFA. They can be used as preconditionners of the conjugate gradient algorithm [8] or for directly solving equation 6. Their implementation are based on external libraries such as Eigen, MKL and Taucs. When dealing with Finite Element Models, the matrices are generally very sparse and efficient implementations based on sparse factorizations allow for fast computations. Moreover, when dealing with specific topologies, such as wire-like structures, tri-diagonal band solvers can be used for extremely fast results

in $\mathcal{O}(n)$ These different linear solvers address matrices which can be stored in different formats, adapted to the numerical library. The type of matrix is a parameter of the linear solver, and of the visitors the solver uses. Ten linear solvers have been implemented in SOFA. They can be interchanged to compare their efficiency.

4.3 Constraint solvers

SOFA allows the use of Lagrange multipliers [12] to handle complex constraints, such as contacts and joints between moving objets that can not be straightforwarly implemented using projection matrices as in Section 4.1. They may be combined with explicit or implicit integration. Each constraint depends on the relative position of the interacting objects, and on optional parameters (such as a friction coefficient, etc.)[3]:

$$\begin{array}{c} \Phi(\mathbf{x}_1, \mathbf{x}_2, ...) = 0 \\ \Psi(\mathbf{x}_1, \mathbf{x}_2, ...) \geq 0 \end{array} \tag{8}$$

where Φ represents the bilateral interaction laws (attachments, sliding joints, etc.) whereas Ψ represents unilateral interaction laws (contact, friction, etc.). These functions can be non-linear. The Lagrange multipliers are computed at each simulation step. They add force terms to Equation (6):

$$\begin{array}{c} \mathbf{A}_1 \delta \mathbf{v}_1 = \mathbf{b}_1 + \mathbf{H}_1^T \lambda \\ \mathbf{A}_2 \delta \mathbf{v}_2 = \mathbf{b}_2 + \mathbf{H}_2^T \lambda \end{array} \tag{9}$$

where

$$\mathbf{H}_1 = [\frac{\delta \Phi}{\delta \mathbf{x}_1} \; ; \; \frac{\delta \Psi}{\delta \mathbf{x}_1}] \qquad \mathbf{H}_2 = [\frac{\delta \Phi}{\delta \mathbf{x}_2} \; ; \; \frac{\delta \Psi}{\delta \mathbf{x}_2}]. \tag{10}$$

Matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ are stored in the *MechanicalState* of each node. Thus, when the constraint applies to a model that is mapped (see section 2.5), the constraints are recursively mapped upward like forces to be applied to the independent degrees of freedom [11]. Solving the constraints is done by following these steps:

Step 1, Free Motion: interacting objects are solved independently while setting $\lambda = 0$. We obtain what we call a *free motion* $\delta \mathbf{v}_1^{\mathrm{f}}$ and $\delta \mathbf{v}_2^{\mathrm{f}}$ for each object. After integration, we obtain $\mathbf{x}_1^{\mathrm{f}}$ and $\mathbf{x}_2^{\mathrm{f}}$. During this step, each object solves equation (9) with $\lambda = 0$ independently using a dedicated solver.

Step 2, Constraint Solving: The constrained equations can be linearized and linked to the dynamics (see [10] for details).

$$\begin{bmatrix} \Phi(\mathbf{x}_1, \mathbf{x}_2) \\ \Psi(\mathbf{x}_1, \mathbf{x}_2) \end{bmatrix} = \begin{bmatrix} \Phi(\mathbf{x}_1^{\mathrm{f}}, \mathbf{x}_2^{\mathrm{f}}) \\ \Psi(\mathbf{x}_1^{\mathrm{f}}, \mathbf{x}_2^{\mathrm{f}}) \end{bmatrix} + \underbrace{h\mathbf{H}_1 \delta \mathbf{v}_1^{\mathrm{c}} + h\mathbf{H}_2 \delta \mathbf{v}_2^{\mathrm{c}}}_{h\left[\mathbf{H}_1 \mathbf{A}_1^{-1} \mathbf{H}_1^T + \mathbf{H}_2 \mathbf{A}_2^{-1} \mathbf{H}_2^T\right] \lambda} \tag{11}$$

[3] For simplicity, we present the equations for two interacting objects (rigid or deformable) 1 and 2, but the solution applies to an arbitrary number of interacting bodies.

With $\delta \mathbf{v}^{\mathrm{c}} = \delta \mathbf{v} - \delta \mathbf{v}^{\mathrm{f}}$. Together with equation (8), these equations compose a Mixed Complementarity Problem that can be solved by a variety of solvers. We compute the value of λ using a projected Gauss-Seidel algorithm that iteratively checks and projects the various constraint laws contained in Φ and Ψ [13].

Step 3, Corrective Motion: when the value of λ is available, the corrective motion is computed as follows:

$$\begin{aligned} \mathbf{x}_1^{t+h} &= \mathbf{x}_1^{\mathrm{f}} + h\delta \mathbf{v}_1^{\mathrm{c}} \quad \text{with} \quad \delta \mathbf{v}_1^{\mathrm{c}} = \mathbf{A}_1^{-1}\mathbf{H}_1^T\lambda \\ \mathbf{x}_2^{t+h} &= \mathbf{x}_2^{\mathrm{f}} + h\delta \mathbf{v}_2^{\mathrm{c}} \quad \text{with} \quad \delta \mathbf{v}_2^{\mathrm{c}} = \mathbf{A}_2^{-1}\mathbf{H}_2^T\lambda \end{aligned} \tag{12}$$

An *AnimationLoop*, typically placed at the top of the graph of SOFA, has the role of imposing this new scheduling to the rest of the graph.

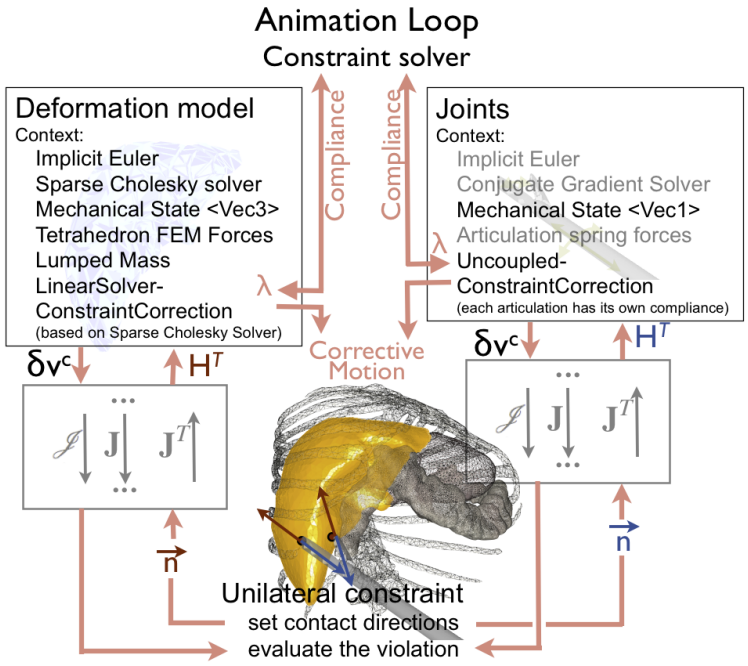

Fig. 13 Contact process using constraints: A unilateral constraint is placed at the level of the contact points. The constraint direction is mapped to the degrees of freedom of the objects to obtain matrix $\mathbf{H}^T$. The *ConstraintCorrections* components compute the compliance to obtain equation 11. The Constraint solver found a new value of λ which is sent to the *ConstraintCorrections* to compute an adequate corrective motion. The AnimationLoop is placed at the root of the simulation graph to impose the steps of the simulation process.

Compliance computation : Equations 11 and 12 involve the inverse of matrix $\mathbf{A}$ (called compliance matrix), which changes at every time step in case of a non-linear model. Depending on the simulation case, computing this inverse could be time

consuming for real-time simulation. When this is too time-consuming, we propose several strategies to improve the speed of the algorithm such as using the diagonal of $\mathbf{A}$ instead of the whole matrix, or a precomputed inverse [27], or an asynchronous factorization on the GPU [9]. These strategies are implemented in a category of components, called *ConstraintCorrections* that provide different ways of computing $\delta \mathbf{v}^c$ given a value of λ. Given a simulation, it is very easy to make tests and chose the best solution.

4.4 Collision detection and response

Collision detection is split in several phases, each implemented in a different component, and scheduled by a *CollisionPipeline* component. Each potentially colliding object is associated with a collision geometry based on or mapped from the independent DOFs. The broad phase component returns pairs of colliding bounding volumes (currently, axis-aligned bounding boxes). Based on this, the narrow phase component returns pairs of geometric primitives, along with the associated contact points. This is passed to the contact manager, which creates contact interactions of various types based on customizable rules. Repulsion has been implemented based on penalties or on constraints using Lagrange multipliers, and is processed by the solvers together with the other forces and constraints. When stiff contact penalties or contact constraints are created by the contact manager, an optional *GroupManager* component is used to create interaction groups handled by a common solver, as discussed in Section 3. When contacts disappear, interaction groups can be split to keep them as small as possible. The scenegraph structure thus changes along with the interaction groups.

This framework has allowed us to efficiently implement popular proximity-based repulsion methods as well as novel approaches based on ray-casting [18] or surface rasterization [15, 3]. Its main limitation is that the contacts can be mechanically processed only after they all have been modeled by the collision pipeline. This does not allow to mechanically react to a collision as soon as it is detected, possibly avoiding further collisions between primitives of the same objects.

4.5 GPU support

By targeting complex interactive simulations, SOFA needs to achieve high computational performances. For this reason, we have extended the architecture and functionalities to handle GPU-based computations.

Thanks to the scene-graph design, components such as ODE and linear solvers can be applied to both CPU and GPU models, as they do not directly manipulate the state vectors. Other components such as force fields and constraints needs to be specifically adapted. However, most of their code is reused from the CPU version

thanks to templated generic programming. As explained in Section 2.1, mechanical components (but also mappings and some collision models) are templated by the type of the state variables they manipulate. This template parameter (*DataTypes*) specify the type of DOFs in use, but also their containers, which on CPU are simple vectors. To implement GPU support, an hybrid CPU/GPU vector container was created, replacing standard vectors by adding new methods and functionnalities to transparently provide a GPU-side version of the contained data. This is necessary because currently GPUs use their own memory. The hybrid vector container provides methods to access the data either on the CPU or GPU, for *read* or *write* operations. Flags are used internally to execute allocation and transfer operations when necessary. Standard operators such as random access are also provided, so that existing CPU codes can be used.

As a consequence, to implement GPU support in an existing SOFA component, we first need to instanciate the existing code to a new hybrid *DataTypes* template. Then, only the few computationally intensive methods that are used during the simulation loop will need to be specialized and rewritten to be executed on GPU. All initialization, debugging and validation codes can be reused, and data transfers happen transparently. Below are two important examples of components where the use of GPU-based computation has proved very beneficial.

4.5.1 Physical models with implicit time integration

We implemented FEM using a Conjugate Gradient-based implicit time integration scheme on GPU. In contrast with existing GPU-based sparse solvers [19, 5], we do not explicitly build the system matrix, but instead parallelize the vector operations and matrix-vector products based on topological elements. The parallelization strategy relies on first computing the contribution of mesh elements using one thread per tetrahedron, followed by a parallel gather to accumulate contributions at vertices. This considerably reduces the number of operations required, and more importantly the consumed bandwidth, enabling the method to be fast enough for interactive simulations of soft bodies. Further optimizations include mesh ordering, compact data structures, memory layout, and changing sequences of operations to reduce synchronization points. More detail can be found in [2]. A deformable object with 45k tetrahedral elements is simulated at 212 FPS on a Nvidia GeForce GTX 480, 18$\times$ faster than our most optimized sequential implementation on an Intel Core i7 975 3.33GHz CPU.

The conjugate gradient solver does not access the objects directly, and stays on the CPU. The only values it gathers from the simulated objects are dot products, which are easily read back from the GPU and summed up to the dot products of other objects. This allows one to transparently combine objects simulated on the CPU and on the GPU in the same equation system , including simple and complex constraints discussed in sections 4.1 and 4.3, using mechanical state containers to encapsulate the data transfers.

In addition to FEM, several components in SOFA have a GPU-based implementation (using CUDA for most of them, and OpenCL in a few instances), such as linear springs, and a fluid simulation based on Smoothed Particle Hydrodynamics (SPH) [22], allowing more than 32k particles to be simulated at 25Hz on a GeForce GTX 280 GPU.

4.5.2 Image-based Collisions

Collision detection can also benefit from GPU-based computation. We have integrated image-based collision and response methods [15, 3] which are well suited for handling complex deformable objects. They compute intersection volume gradients which are discretized on pixels and accumulated on vertices. Applied to complex geometries, this results in dramatically simpler equation systems than those of traditional mesh contact models. Contact between highly detailed meshes can be simplified to a single unilateral constraint equation, or accurately processed at arbitrary geometry-independent resolution with simultaneous sticking and sliding across contact patches. Complex contacts involving both rigid and deformable objects can be detected and their response computed at interactive rates and without precomputations, making the method suitable for large deformations and cutting.

5 Interface

SOFA is a library which can be called from any external C++ program. The distribution comes with an executable providing a batch execution mode, a simple Glut window and a more sophisticated Qt-based graphics user interface (*GUI*). The scenes can be built procedurally or read from XML files, as presented in the following section. Gnuplot files can be exported during the simulation for replaying the animation or plotting trajectory curves. Interactive visualization can be performed using OpenGL, Ogre or OpenSceneGraph. Sequences of geometry files can also be exported, to create high quality images and videos using state-of-the-art renderers.

5.1 Scene files

An interesting feature of scenegraphs is their ability to be read from and written to text files. The following code corresponds to the scene shown in Figure 6.

```
1 <?xml version="1.0"?>
2 <Node  name="scene" showVisualModels="1" showBehaviorModels="1" showCollisionModels="1"
        showMappings="0" showForceFields="0" >
3     <DefaultPipeline name="collisionPipeline" />
4     <BruteForceDetection name="broadPhase" />
5     <NewProximityIntersection name="narrowPhase" />
6     <DefaultContactManager name="contactManager" response="default" />
```

```
    <Node  name="rigid"  >
        <EulerSolver name="eulerSolver" />
        <MechanicalState template="Rigid" name="rigidDofs" position="0 0 0 0 0 0 1"  />
        <UniformMass template="Rigid" name="rigidMass" />
        <Node  name="contactModel" >
            <MechanicalState template="Vec3d" name="sphereCenters" position="0 0 0" />
            <SphereModel template="Vec3d" name="spheres" fileSphere="mesh/liver.sph" />
            <RigidMapping name="sphereMapping" input="@../rigidDofs" output="@sphereCenters" />
        </Node>
    </Node>
    <Node  name="deformable" >
        <RungeKutta4Solver name="rungeKutta4Solver" />
        <MechanicalState template="Vec3d" name="particleDofs" position="2 0 0 3 0 0"  />
        <DiagonalMass template="Vec3d" name="particleMasses" massDensity="1" />
        <SphereModel template="Vec3d" name="paricleSpheres" />
        <SpringForceField template="Vec3d" name="internalForces" />
    </Node>
    <SpringForceField template="Vec3d" name="contactSpring"
            object1="rigid/contactModel/sphereCenters" object2="deformable/particleDofs" />
</Node>
```

5.2 User interface

The GUI is shown in Figure 14. It is mainly composed of a graphics viewer, and tabs to tune the viewing or to display alternative views, like the scenegraph shown in this example. Selecting a component in the scenegraph window allows to open its GUI and to interactively edit the parameters using specialized widgets. The edition of parameters directly in the graphic window is not yet implemented. An application included in the distribution, *Modeler*, allows the addition and removal of nodes and components in a scenegraph, which can be loaded from and exported to XML files. The Data objects used to store the attributes of the components are used to automatically create the component GUIs. These can be customized to display additional features such as the convergence of the linear solver shown in the right of the figure. The GUI can also display a graphical view of the computation time measured hierarchically per visitor or per object, as shown in Figure 15. This is also useful to trace the call graph. The tree view shows the two visitors fired during one time step, along with the visitors they have triggered, recursively.

5.3 Haptic Rendering

The main interest of interactive simulation is that the user can modify the course of the computations in real-time. This is essential for surgical simulation : during a training procedure, when a virtual medical instrument comes into contact with some models of a soft-tissue, *instantaneous* deformations must be computed. This visual feedback of the contact can be enhanced by haptic rendering so that the surgeon can really *feel* the contact.

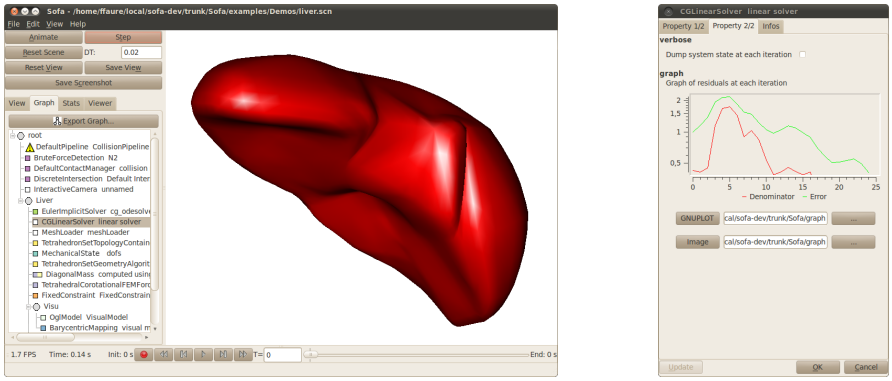

Fig. 14 SOFA's default user interface. Left: the main window, with the scenegraph on the left. Right: the GUI of the CGLinearSolver selected in the scenegraph.

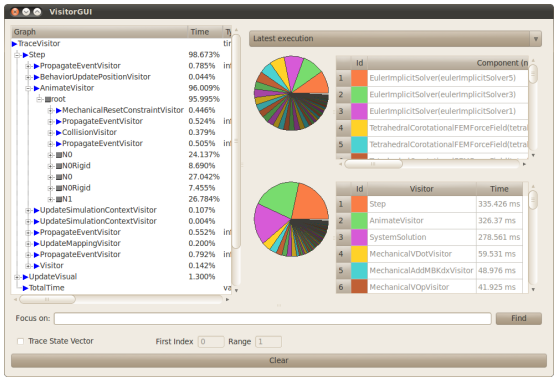

Fig. 15 The visual display of computation times for each visitor. In the left, the triangles denote visitors, while squares denote traversed nodes. In the right, the computation time is displayed per component (top) and per visitor (bottom).

There are two main issues for a platform like SOFA for providing haptics: the first is that haptic forces need to be computed at $1kHz$ whereas real-time visual feedback (without haptic) is obtained at $30Hz$. The second is that haptic feedback could artificially add some energy inside the simulation that creates instabilities, if the control is not *passive*.

Thus two different approaches are currently implemented in SOFA. The first one is the *Virtual Coupling* technique and the other, more advanced, allows for rendering the constraints presented in section 4.3.

Virtual Coupling: the coupling of a haptic device is bidirectional: the user applies some motions or some forces on the device and this device, in return, applies forces and/or motions to the user. The majority of the haptic devices propose a *Impedance* coupling: the position of the device is provided by the API and this

API asks for force values from the application. A very simple scheme of coupling, presented in Figure 16, could have been used. In this *direct coupling* case, the simulation would play the role of a controller in an open loop.

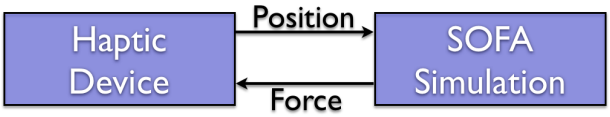

Fig. 16 Direct coupling

Such a design is not suitable when stable and robust haptic feedback on a virtual environment is desired. Indeed some combination of the environment impedance and human user reactions can generate instabilities [14]. To avoid this, a virtual mechanical coupling is set. It corresponds to the use of a damped stiffness between the position measured on the device and the simulated position in the virtual environment (see Fig17). If very stiff constraints are being simulated, then the stiffness perceived by the user will not be infinite but will correspond to the stiffness of this virtual coupling. Hence, a compromise between stability and performance must be found by tuning the stiffness value of the coupling.

Fig. 17 Virtual coupling technique. A 6-DoF damped spring is placed between the haptic loop and the simulation.

The damped spring is simulated two times. One time in the haptic loop and one time in the simulation loop. If the two loops are synchronized, then the result is the same. But it can also be used in asynchronous mode: fast update of the haptic loop and low rates in the simulation. In this case, the haptic feedback remains stable but the delay between the two loops creates artificial damping. There is an option to cancel this artificial damping if no contact is detected in the simulation. However, this option can create a sensation of sticking contacts. The main advantage of the virtual coupling technique is that it can be easily employed with every simulation of SOFA. The main drawback is that the haptic rendering is not transparent (i.e. the haptic interaction does not feel the same as the real interaction it is reproducing).

Constraint-based rendering: A novel way of dealing with haptic rendering for medical simulation has been proposed in the context of SOFA (see [27] and [24]). The approach deals with the mechanical interactions using appropriate force and/or motion transmission models named *compliant mechanisms* (see Fig18). These mechanisms are formulated as a constraint-based problem (like presented in section 4.3) that is solved in two separate threads running at different frequencies.

The first thread processes the whole simulation including the soft-tissue deformations, whereas the second one only deals with computer haptics. With this approach, it is possible to describe the specific behavior of various medical devices while relying on a unified method for solving the mechanical interactions between deformable objects and haptic rendering.

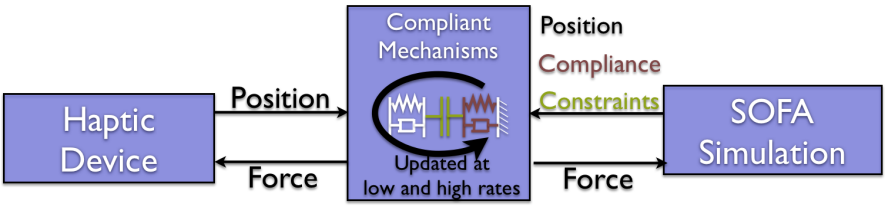

Fig. 18 Compliant mechanisms technique. The simulation shares the mechanical compliance of the objects and the constraints between them. The constraint response is being computed at low rate within the simulation and at high rates within a separate haptic thread. A 6-DoF damped spring is still used to coupled the position of the device to its position in the simulation

6 Examples

In this section, we detail three examples of advanced use of SOFA. The first, in Section 6.1, presents a multiphysics model coupling. The second, in Section 6.2, studies the introduction of a novel deformable model in the SOFA framework. Finally, Section 6.3 details a complete simulation of liver resection.

6.1 Cardiac modeling

The heart is a complex machine that is controlled by an electro-mechanical coupling: an electrical wave propagates through the heart and depolarizes the cardiac cells or muscle fibers (see Figure 19) leading to the contraction of ventricles which then eject the blood in the aorta and pulmonary arteries. The fiber relaxation then follows and the ventricles are filled again with incoming blood from the atria.

Modeling the heart

As for the modeling of skeletal muscles, cardiac mechanics is based on separated components : a passive part that deals with hyper-elasticity and viscosity, and an active part during the contraction and relaxation phases corresponding to the binding and unbinding of the actine-myosine bridges in the sarcomeres. We chose to implement the electro-mechanical coupling model proposed by Bestel-Clement-Sorine (BCS) [6]. It is schematically represented in Figure 20.

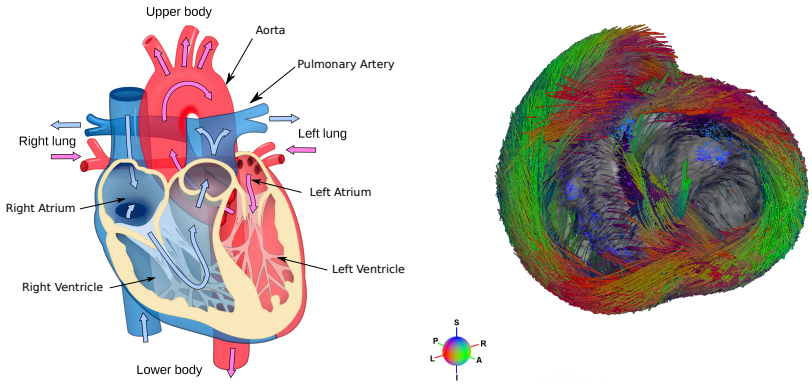

Fig. 19 (Left) Heart Anatomy showing the right and left heart that work synchronously to pump the blood in the pulmonary and general system, from *Wikipedia*. (Right) Representation of the fibers where the colors describe the orientation of the primary eigenvector according to the color sphere, from [26].

To describe the blood flow that goes in and out of the heart cavities, we chose to apply blood pressures as constraints. A valve model was therefore implemented in order to apply alternatively different boundary conditions depending on the four phases of the cycle [28]: filling, isovolumetric contraction, ejection and isovolumetric relaxation.

Simulating the heart in SOFA

The geometry of the heart muscle was first segmented from images then meshed with tetrahedra, and the fibers were extrapolated from an atlas (more details about fibers and so on can be found in [26]). Fiber directions are crucial for both the electrophysiological resolution and the mechanical resolution since the wave follows those fibers and the muscle is stiffer in the fiber directions.

Each mechanical or electrophysiological component was implemented separately and independently. We used *MultiTagAnimationLoop* to deal with the electromechanical coupling. This solver enables to solve two different systems in the same scene, at each time step. The complete model of the heart is represented through the diagram Fig. 20. We can therefore find, like in all SOFA scenes, solvers, topological components, a loader, a Mass and forcefields components. In this case, the passive part of the model is the hyperelastic forcefield *MooneyRivlin*. The active part is more complicated since it needs the resolution of a electrophysiological system done in ElecNode, which gives a 1D potential for each element of the mesh. This potential is then used in *ContractionForceField* for the unidirectional coupling. Each step of the electromechanical model first requires the solution of the ElecNode to estimate the wave potential u which is necessary to the *ContractionForceField* component to obtain τ_c and finally update the mechanical system. Boundary conditions are im-

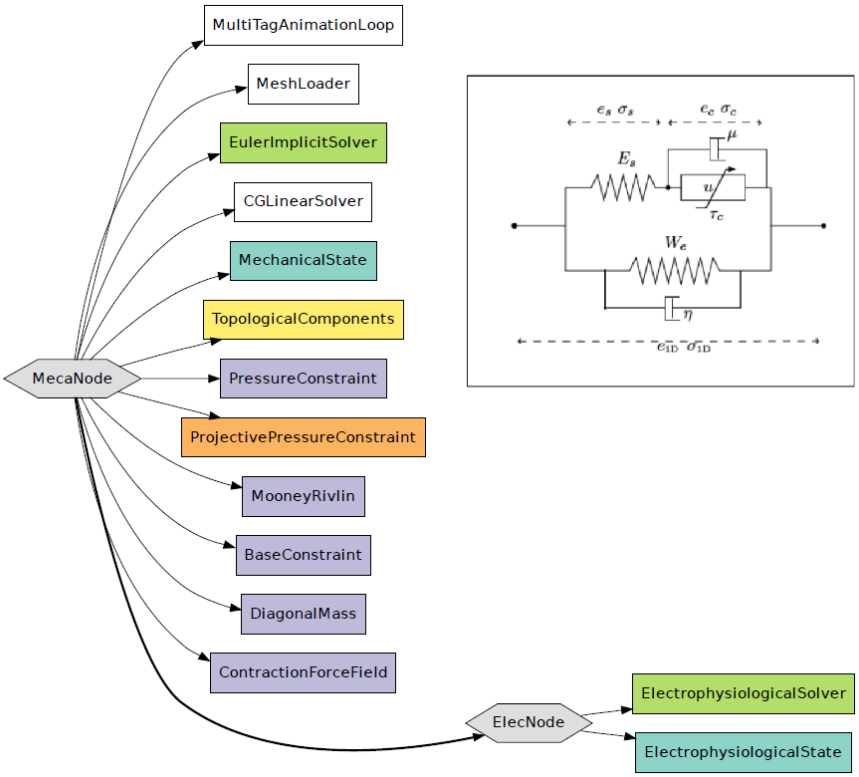

Fig. 20 (Left) SOFA graph with the different components to simulate the complete model of the Heart. (Right) Complete rheological model. W_e is the strain energy of the chosen hyperelastic material, here Mooney Rivlin. η represents the viscosity of the passive part. u is the electrophysiological potential that controls the contraction stress τ_c. μ deals with active relaxation in forms of damping, and E_s is a linear spring to enforce stress in the fiber direction. The indices $s, c, 1D$ respectively refer to the contraction terms, the linear spring in series and the projection along the fiber direction

plemented through the *PressureConstraint* and *ProjectivePressureConstraint* components which apply forces in the endocardium due to blood pressure and projects the velocity field to satisfy the valve model. Finally the base of the heart is attached with springs (*BaseConstraint*) which is equivalent to add extra diagonal terms in the stiffness matrix of the basal nodes.

Results of the simulation

This complex simulation takes around 3 minutes for a full cycle (representing on average 0.8s), with a time step of 10ms, with 14000 mesh nodes, on a normal laptop. More than 95% of the cost is due to the mechanical solution. An example of resulting geometry during a cycle is shown in Figure 21. We also managed to recover the volume and pressure curves that clearly contain the four phases, as shown

in Fig. 21.

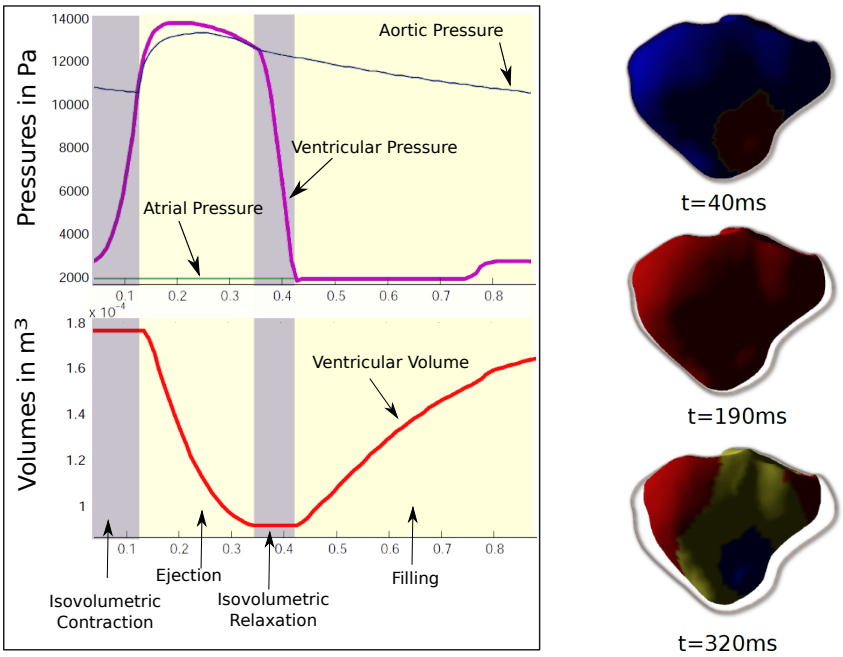

Fig. 21 (Left) Resulting pressure and volume curves for the left ventricle, for one heart cycle. (Right) Resulting geometry at different times of the cycle overlaid with the initial mesh (in shadow). The color map indicates the potential wave, solutions of the electrophysiological node.

The complete multi-physics model can therefore be successfully implemented on SOFA, the simulation time is short enough so that numerous parameter estimation techniques can be though of. Moreover, several cardiac therapy simulations are made possible with SOFA interactivity, such as Cardiac Resynchronization Therapy or cardiac Radio-Frequency Ablation [20],[23].

6.2 Knee joint mechanics

In this section, we show an example of leveraging the versatility of the framework to create new physical deformable models, while re-using available high performance components for collision and time integration. The knee is a complex joint including four bones connected by ligaments with complex shapes (fig. 23). Its physical simulation has been difficult to achieve, because an important limitation of FEM comes from the necessity of partitioning the objects in elementary volumes, each of

them filled with a uniform material, resulting in high computation times which do not allow interactive applications.

Noticing that the deformations are typically smooth, we model the deformation by blending a small number of rigid displacements, to dramatically reduce the number of independent DOFs and the associated computation time. This frame-based, mesh-free deformation function alleviates the sampling issues of the traditional FEM and particle-based approaches, and it is straightforwardly applied to arbitrarily detailed geometry. The deformation energy is computed at integration points, summed up and differentiated with respect to the independent DOFs, as usual in Continuum Mechanics. The frames and their distribution, the computation of their relative influence, the choice of blending functions, the distribution of the integration points and the computation of their associated energy are detailed in [17] and [16].

All the steps involved in this pipeline are modeled using the SOFA components shown in Figure 22. The *MechanicalState<Frame>* component contains the independent DOFs (frames with 6, 12 or 30 DOFs per frame were implemented). To measure local deformations in the solid, the spatial derivatives of the displacement function (*i.e.*, the deformation gradient, a 3×3 matrix) are necessary. Therefore, a mapping computes the deformation gradients and deformation gradient rates stored in the *MechanicalState<DeformationGradient>*, based on the frame positions and velocities. The deformation gradients are converted to strains by a *ForceField* using the Green-Lagrange strain (other measures where implemented), and the corresponding stresses are computed using a material law defined in *MaterialMap*. The *VolumePreservationForceField* additionally penalizes the volume change based on the determinant of the deformation gradient. The stresses are converted to deformation gradient forces, which are generalized forces represented by 3×3 matrices accumulated in the *MechanicalState*, and which are then mapped up to frame forces. External forces acting on the collision model are incorporated through a

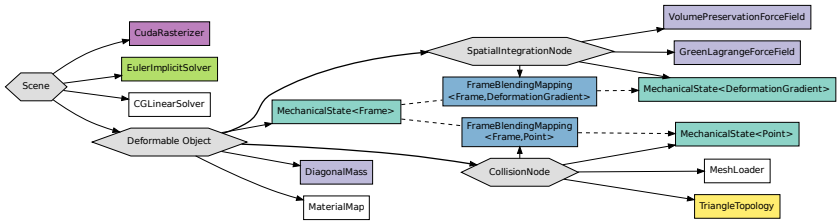

Fig. 22 SOFA graph of a frame-based simulation.

low order specialization of the mapping, where only 3d points are mapped instead of deformation gradients. After summation of DOF forces, a standard implicit ODE solver updates the positions and velocities according to the generalized frame stiffness matrix, mass and damping coefficients. GPU collision detection and response is applied using the available component *CudaRasterizer* [3]. In summary,

the component-based architecture of SOFA is used to customize each stage of the simulation pipeline: the DOF type, the kinematic interpolation method, the strain measure, the material constitutive law, the spatial integration method and the degree of integration samples.

The framework structure provided us with implementation guidelines, and available components for implicit integration and gpu collision detection where directly reused.

The frame-based deformation framework produces interactive simulations of complex heterogeneous objects, such as the knee shown in Figure 23. The simulation is one to three orders of magnitude faster than using FEM, thanks to the reduced number of DOFs, and this allows us to interactively extend the joint by pulling the upper ligament. We emphasize that the joint behavior is entirely created by the tissue elasticity and the contact forces between the bones. Though this simple model needs improvements to meet the precision requirements of biomedical applications, it illustrates the efficiency and the extensibility of SOFA.

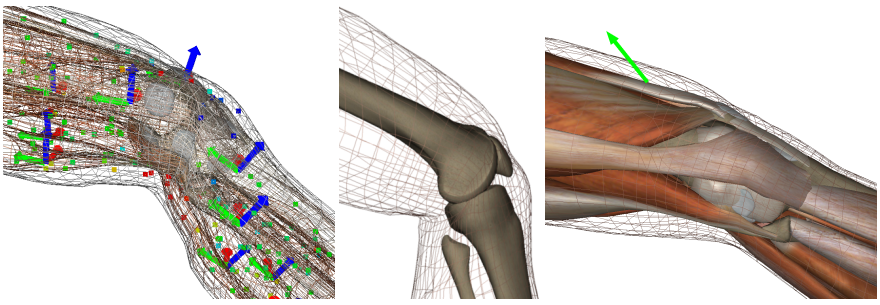

Fig. 23 Interactive simulation of a knee joint. Left: the moving frames (arrows) and the integration points (squares). Middle: the contact between the bones. Right: the user stretches the knee by pulling a ligament.

6.3 Simulation of hepatic resection

The liver is one of the major organs in the human body and is in charge of more than hundred vital functions. For this reason, its pathologies are varied, numerous and often lethal. Nevertheless, surgery is not always performed due to several limitations, in particular the pre-operative estimation of the liver volume remaining after resection. This volume highly depends on the choice of the operative strategy as well as anatomical constraints defined by the vascular network (see figure 24). Such limitations could be overcome by improving the quality of the planning, which relies on a combination of components, mostly image processing, registra-

tion, and biomechanical modelling. In the context of the PASSPORT project[4] we have developed a simulation in SOFA as a proof of concept towards this goal. This work covers four main areas: a model of the whole liver, composed of its three main components: parenchyma, vascular network, and Glisson capsule; collision detection and response adapted to complex contact configurations; topological changes for simulating the resection of a part of the liver.

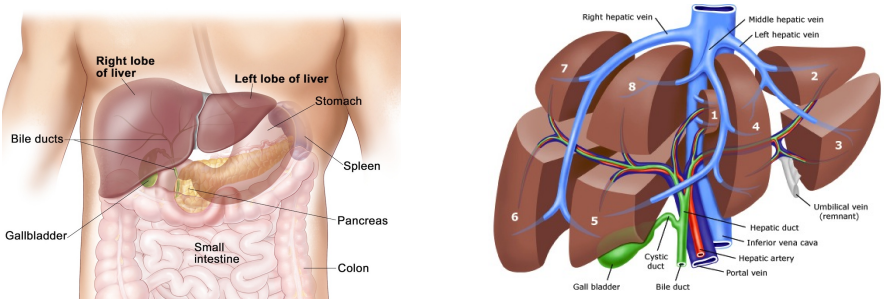

Fig. 24 Liver anatomy: (left) with an average weight of about 1.5 kg, the human liver is the largest internal organ. It is located in the right upper quadrant of the abdominal cavity, resting just below the diaphragm. It lies to the right of the stomach and overlies the gallbladder. (Right): the liver is connected to the hepatic artery and the portal vein. The hepatic artery carries blood from the aorta, whereas the portal vein carries blood containing digested nutrients from the entire gastrointestinal tract and also from the spleen and pancreas. The vascular structure of the liver provides a functional subdivision of the liver into eight subsegments.

6.3.1 Liver biomechanical model

In spite of various results in the area of liver simulation and modeling, a key element is required for providing an accurate hepatic resection system: an accurate biomechanical model of the liver, compatible with (near) real-time simulations to enable augmented reality approaches. An important body of work exists regarding the biomechanical model of the liver and its mechanical properties, and several works have addressed the issue of real-time simulation. Yet, none of the existing approaches take into account the biomechanical influence of the vascular structures of the liver nor its capsule. To address this limitation, we have developed a vascularized model of the liver, which takes into account separate constitutive laws for the parenchyma and vessels, and defines a coupling mechanism between these two entities. Similarly, a capsule model can be linked to this vascularized liver model to build a complete liver model.

[4] PAtient Specific Simulation and PreOperative Realistic Training for liver surgery – www.passport-liver.eu

Parenchyma model: multiple biomechanical studies concerning the liver have reported constitutive models and parameters for the parenchyma, with an overall agreement on a viscoelastic behaviour. Yet, depending on the application, a simpler non-linear elastic model can be sufficient. This is typically the case when the transient part of the deformation may not be of interest, but rather the static equilibrium under some specific loading conditions. SOFA makes it easy to choose between different constitutive models, and different implementation of these models. In a variety of examples we have used a finite element co-rotational method (*TetrahedralCorotationalFEMForceField* to simulate the deformation of the parenchyma. This method allows for large displacements or rotations in the model, while relying on a linear expression of the stress-strain relationship. As such it offers a very good tradeoff between realism of the behavior and computational efficiency. However, this approach does not enforce incompressibility. We have also experimented with the MJED [21] model. The Multiplicative Jacobian Energy Decomposition (MJED) is a method for discretizing hyperelastic materials on linear tetrahedral meshes which leads to faster computation than the standard Finite Element Method and enables this way to reach near real-time simulations. In the remainder of this section, we will illustrate results based on the co-rotational method (*TetrahedralCorotationalFEMForceField* in SOFA).

Vascularized liver model: blood vessels are modeled using serially linked beam elements (*BeamFEMForceField*). In this model, each beam element is flexible (i.e. handles stretching, bending and torsion), and can take into account the particular nature of vessels through specific cross section profiles and moments of inertia. A beam element is defined by two nodes, each described by six degrees of freedom, three of which correspond to the spatial position, and three to the angular position of the node in a global reference frame. The resulting representation allows for geometrically non-linear deformations. The process of converting a patient specific surface model of the vascular system into a set of beam elements is done automatically through centerline extraction of the vessels and conversion into a set of Bézier curves. This continuous parametric representation of the vessels can then be sampled (according to various criteria) to generate series of beam elements. During the process, parameters such as the minimal radius of selected vessels or density of the discretization points can be adjusted.

To create the vascularized liver model, we used a dedicated *mapping* between the mesh nodes of the vessels and the volumetric elements of the FEM model for the parenchyma. This particular mapping links vessel nodes (with 6 DOFs) to parenchyma nodes (with 3 DOFs). Since no relative motion between the vessels and parenchyma is observed in reality, the mapping can be implemented as a position constraint. At each step of the simulation, the actual displacements of the parenchyma mesh nodes are mapped to the vessel nodes and reciprocally, the force contribution due to the deformation of the vessel is propagated back to the parenchyma.

Complete liver model: A complete mechanical model of the liver should include the influence of the Glisson capsule surrounding the liver parenchyma. This membrane can be described using shell elements [7] or by using angular springs *Triangu-*

larBendingSprings, both models being available in SOFA. The resulting combined model can be simulated in real-time thanks to an advanced solver using an asynchronous preconditioning strategy (see section 4), particularly well-suited for this case where different material stiffness parameters of the constituents of the liver lead to an ill-conditioned system matrix (this bad conditioning is due to the important difference in stiffness between parenchyma and vessels, the stiffness in the direction of the main axis of the vessel being several orders of magnitude higher than the parenchyma).

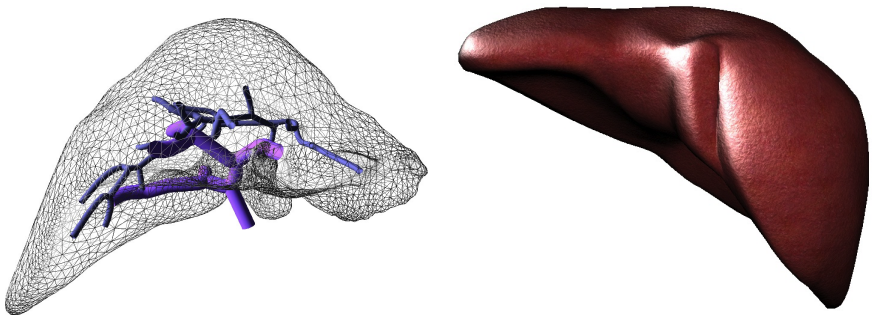

Fig. 25 Deformable liver simulation based on components available in SOFA: non-linear finite element model with implicit time-integration implemented on GPU, mechanical coupling between vessels, parenchyma and capsule based on multi-model representation, efficient numerical solver based on asynchronous preconditioning, and improvements in visual rendering. The liver geometry is based on meshes generated automatically from abdominal CT scan images.

6.3.2 Complex contact handling and haptic feedback

Collision detection and collision response: collision detection is based on the LDI method introduced in section 4.5.2. Using this approach, fast contact response can be provided using constraints with friction, even when relying on detailed surface meshes. Figure 26 illustrates a real-time simulation involving contacts between surgical instruments and the liver, as well as between the liver and surrounding organs.

Haptic feedback: the motion of the laparoscopic instrument or camera can be controlled using an haptic interface. SOFA incorporates drivers for different haptic interfaces, and provides haptic rendering algorithms. In particular, we have developed a generic formalism for solving complex interactions between various medical devices and anatomical structures, and for computing the associated haptic rendering. The proposed approach models the interactions using virtual mechanisms (that are extended here for deformable objects) and solved using a constraint-based process. With this approach, it is possible to describe the specific behavior of various medical devices while relying on a unique method for solving the interactions and computing haptic feedback [25]. In this approach, the dynamics of the virtual ob-

Fig. 26 Real-time complex interactions inside the abdominal cavity. Left: visualization of the collision models. The green surface is the diaphragm, which motion is controlled to reproduce the respiratory cycle. Right: final rendered image of the tissue interactions, involving tissue-tool contacts but also contacts between multiple anatomical structures (liver, stomach, ribs).

jects is computed at low-rate in the simulation and the mechanical interaction forces are modeled and solved using constraints. But these forces are re-computed at high-rate, in the haptic loop based on an intermediate representation shared between the two loops. This intermediate representation includes (I) a physics based measure of the mechanical coupling between the different interactions and (II) a set of kinematic links that capture the interactions' behavior and which are modeled by using constraint laws). As such, this intermediate representation is then analogous to the concept of virtual mechanisms.

6.3.3 Simulation of hepatic resection

Simulation of liver resection requires to take topological changes into account. Several approaches for cutting volumetric meshes have been developed within the SOFA framework. The first one relies on triangular or tetrahedral meshes that are locally remeshed to simulate the contact with a scalpel or with a bipolar cautery device (see Figure 27). The second approach is based on hierarchical hexahedral meshes inside which surfaces are embedded, in which case topological changes can be simulated by splitting the hexahedral meshes and modifying the embedded meshes. Each approach has benefits and drawbacks, and a trade-off between complexity, stability, and computational efficiency has to be found for each application. As a result, a combination of several advanced components available in SOFA made it possible to develop a convincing proof of concept of hepatic surgery simulation, involving complex interactions with multiple organs in the abdominal cavity while providing an improved level of realism compared to previous systems.

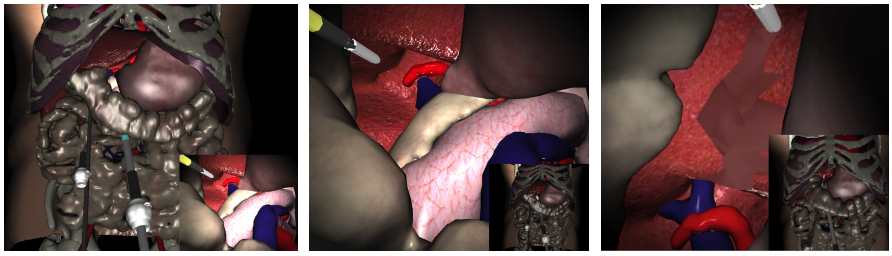

Fig. 27 Interactive simulation of hepatic resection. Left: global view of the operative field, with the bottom-right insert showing the actual view from the laparoscopic camera. Middle: laparoscopic view showing instrument interactions with the liver. Right: local resection of the liver using volumetric topological changes.

7 Conclusion

The need for reusability, versatility and performance has driven the design of SOFA, which has reached a decent level of usability. The extension to a multi-thread framework and the easier management of topological changes are work in progress, as well as the improvement of the learning curve through better documentation and cleaner code. In future work, we plan to enhance the multi-physics capabilities and the deployment on the GPU, and to make collision detection and response more easily customizable and intertwinable with ODE and constraint solutions.

Within five years, SOFA has become one of the major open-source medical simulation softwares, with more than 100,000 downloads of the public version, and an increasing number of research teams and industrials using it. We believe that the versatile yet efficient design of the framework make it a good basis for long-term developments. The project has received an important funding by INRIA, and it has to evolve toward a more sustainable model. The foundation of an international consortium would be an interesting move, and we are open to proposals from academic institutes and companies.

References

1. Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. SOFA - an open source framework for medical simulation. In Medicine Meets Virtual Reality, MMVR 15, February, 2007, pages 1–6, Long Beach, California, Etats-Unis, 2007.
2. Jérémie Allard, Hadrien Courtecuisse, and François Faure. Implicit FEM solver on GPU for interactive deformation simulation. In GPU Computing Gems Jade Edition, chapter 21. Elsevier, 2011.
3. Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, and Paul G. Kry. Volume contact constraints at arbitrary resolution. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010), 29(3), August 2010.

4. David Baraff and Andrew Witkin. Large steps in cloth simulation. In SIGGRAPH '98, pages 43–54. ACM Press, 1998.
5. Luc Buatois, Guillaume Caumon, and Bruno Lévy. Concurrent number cruncher - a GPU implementation of a general sparse linear solver. Int J Parallel Emerg. Distrib. Syst., 24(3):205–223, 2009.
6. D. Chapelle, P. Le Tallec, P. Moireau, and M.Sorine. An energy-preserving muscle tissue model: formulation and compatiblediscretizations. IJMCE, 2010.
7. Olivier Comas, Christian Duriez, and Stéphane Cotin. Shell model for reconstruction and real-time simulation of thin anatomical structures. In Tianzi Jiang, Nassir Navab, Josien Pluim, and Max Viergever, editors, Medical Image Computing and Computer-Assisted Intervention – MICCAI 2010, volume 6362 of Lecture Notes in Computer Science, pages 371–379. Springer Berlin / Heidelberg, 2010.
8. Hadrien Courtecuisse, Jérémie Allard, Christian Duriez, and Stéphane Cotin. Asynchronous preconditioners for efficient solving of non-linear deformations. In Proceedings of Virtual Reality Interaction and Physical Simulation (VRIPHYS), November 2010.
9. Hadrien Courtecuisse, Jérémie Allard, Christian Duriez, and Stéphane Cotin. Preconditioner-based contact response and application to cataract surgery. In MICCAI 2011. Springer, September 2011.
10. Hadrien Courtecuisse, Hoeryong Jung, Jérémie Allard, Christian Duriez, Doo Yong Lee, and Stéphane Cotin. Gpu-based real-time soft tissue deformation with cutting and haptic feedback. Progress in Biophysics and Molecular Biology, 103(2-3):159–168, December 2010. Special Issue on Soft Tissue Modelling.
11. Christian Duriez, Hadrien Courtecuisse, Juan-Pablo de la Plata Alcalde, and Pierre-Jean Bensoussan. Contact skinning. In Eurographics conference (short paper), 2008.
12. Christian Duriez, Frederic Dubois, Claude Andriot, and Abderrahmane Kheddar. Realistic haptic rendering of interacting deformable objects in virtual environments. IEEE Transactions on Visualization and Computer Graphics, 12(1):36–47, 2006.
13. Christian Duriez, Christophe Guébert, Maud Marchal, Stéphane Cotin, and Laurent Grisoni. Interactive simulation of flexible needle insertions based on constraint models. In Guang-Zhong Yang, David Hawkes, Daniel Rueckert, Alison Noble, and Chris Taylor, editors, Proceedings of MICCAI 2009, volume 5762, pages 291–299. Springer, 2009.
14. Richard J. Adams et Blake Hannaford. Stable haptic interaction with virtual environments. IEEE Transactions on Robotics and Automation, pages 465–474, 1999.
15. François Faure, Sébastien Barbier, Jérémie Allard, and Florent Falipou. Image-based collision detection and response between arbitrary volumetric objects. In ACM Siggraph/Eurographics Symposium on Computer Animation, SCA 2008, July, 2008, Dublin, Irlande, July 2008.
16. François Faure, Benjamin Gilles, Guillaume Bousquet, and Dinesh K. Pai. Sparse meshless models of complex deformable solids. ACM Transactions on Graphics, 2011.
17. Benjamin Gilles, Guillaume Bousquet, François Faure, and Dinesh K. Pai. Frame-based elastic models. ACM Transactions on Graphics, 30(2), April 2011.
18. Everton Hermann, François Faure, and Bruno Raffin. Ray-traced collision detection for deformable bodies. In 3rd International Conference on Computer Graphics Theory and Applications, GRAPP 2008, January, 2008, Funchal, Madeira, Portugal, January 2008.
19. Jens Krüger and Rüdiger Westermann. A GPU framework for solving systems of linear equations. In GPU Gems 2, chapter 44, pages 703–718. Addison-Wesley, 2005.
20. Tommaso Mansi, Barbara André, Michael Lynch, Maxime Sermesant, Hervé Delingette, Younes Boudjemline, and Nicholas Ayache. Virtual pulmonary valve replacement interventions with a personalised cardiac electromechanical model. In Recent Advances in the 3D Physiological Human, pages 201–210. Springer, November 2009.
21. Stéphanie Marchesseau, T. Heimann, Simon Chatelin, Rémy Willinger, and Hervé Delingette. Multiplicative jacobian energy decomposition method for fast porous visco-hyperelastic soft tissue model. In Proc. Medical Image Computing and Computer Assisted Intervention (MICCAI'10), LNCS. Springer, 2010.
22. J.J Monaghan. An introduction to sph. Computer Physics Communications, 48,(1):88–96, 1988.

23. E. Pernod, M. Sermesant, E. Konukoglu, J. Relan, H. Delingette, and N. Ayache. A multi-front eikonal model of cardiac electrophysiology for interactive simulation of radio-frequency ablation. Computers and Graphics, 35:431–440, 2011.
24. Igor Peterlick, Mourad Nouicer, Christian Duriez, Stephane Cotin, and Abderrahmane Kheddar. Constraint-based haptic rendering of multirate compliant mechanisms. IEEE Transactions on Haptics, Accepted with minor rev.
25. Igor Peterlik, Mourad Nouicer, Christian Duriez, Stephane Cotin, and Abderrahmane Kheddar. Constraint-based haptic rendering of multirate compliant mechanisms. Transactions on Haptics - Special Issue on Haptics in Medicine and Clinical Skill Acquisition, to appear, September 2011.
26. Jean-Marc Peyrat, Maxime Sermesant, Xavier Pennec, Hervé Delingette, ChenYang Xu, Eliot R. McVeigh, and Nicholas Ayache. A computational framework for the statistical analysis of cardiac diffusion tensors: Application to a small database of canine hearts. IEEE Transactions on Medical Imaging, 26(11):1500–1514, November 2007. PMID: 18041265.
27. Guillaume Saupin, Christian Duriez, and Stephane Cotin. Contact model for haptic medical simulations. In ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation, pages 157–165, Berlin, Heidelberg, 2008. Springer-Verlag.
28. M Sermesant, J M Peyrat, P Chinchapatnam, F Billet, T Mansi, K Rhode, H Delingette, R Razavi, and N Ayache. Toward patient-specific myocardial models of the heart. Heart Failure Clinics, 4(3):289–301, July 2008.
29. Yiyi Wei, Stephane Cotin, Le Fang, Jeremie Allard, Chunhong Pan, and Songde Ma. Toward real-time simulation of blood-coil interaction during aneurysm embolization. In Medical Image Computing and Computer-Assisted Intervention, MICCAI 2009, volume 5761, pages 198–205. Springer Berlin / Heidelberg, 2009.