Sofa Documentation

Also See **http://www.sofa-framework.org/documentation**

The Sofa Team

October 12, 2011

Abstract

SOFA (Simulation Open Framework Architecture) is an open-source C++ library primarily targeted at interactive 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 algorithms 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 elastic forces, collision detection, haptics or visual display. To ensure a consistent simulation, 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. As a result of this flexible and 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.

Contents

Part I

Foundation of SOFA

Chapter 1

Introduction to SOFA

1.1 Why SOFA?

Programming interactive physical simulation of rigid and deformable objects requires multiple skills in geometric modeling, computation 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.

1.2 The "philosophy" of SOFA

SOFA introduces the concept of multi-model representation to easily build simulations composed of complex anatomical structures. The pool of simulated objects and algorithms used in a simulation (also called a scene) is described using a hierarchical data structure similar to scene graphs used in graphics libraries. 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 force model and modify the solver and state vectors in order to compute the model 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 multiple partial models, each optimized for a given type of computation. Typically, a physical object in SOFA is described using three partial models: a mechanical model with mass and 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 coupling two different physical objects. During run-time, the models are synchronized using a generic mechanism called *mapping* to propagate forces and displacements. The user can interact in real-time with the mechanical models simulated in SOFA, using the mouse but also using other type of input device. Haptic rendering is also supported.

1.3 Why should I contribute to SOFA?

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. The code is open-source and the license is LGPL. You can use this code to build your own medical simulations needs or for other applications. You can also include this code in a commercial product. The only requirement is that if you modify the code for a commercial product you need to share this modification with your client.

Moreover, you can build upon SOFA using the plug-in system. Your plug-in can have an other license than LGPL. Consequently there is a considerable freedom for you to use SOFA for your research, your developments or your products !

Finally, SOFAis also intended for the research community to help the development and the sharing of newer algorithms and models. So, do not hesitate to share your experience of SOFA, your code and your results with the SOFAcommunity !!!

Chapter 2

Multi-Model approach

Consider the deformable model of a liver shown in the left of Figure 2.1. It is surrounded by different anatomical structures (including the diaphragm, the ribs, the stomach, the intestines...) and is also in contact with a grasper (modeled as an articulated rigid chain). In SOFA, this liver is simulated using three different representations: the first is used to model its internal mechanical behavior, which may be computed using Finite Element Method (FEM) or other models. The geometry of the mechanical model is optimized for the internal force computations, e.g. one will try to use a reduced number of well-shaped tetrahedra for speed and stability. However, we may want to use different geometrical models for visualization or contact computation. The second representation is used for collision detection and response, while the third is dedicated to the visual rendering process. This sections presents these representations and their connections.

2.1 Solid Mechanics

Different models can be employed to discretize a deformable solid continuum as a dynamic or quasi-static system of particles (also called simulation nodes). The node coordinates are the independent degrees of freedom (DOFs) of the object, and they are typically governed by equations of the following type:

$$\mathbf{a} = \mathbf{P}\mathbf{M}^{-1} \sum_i \mathbf{f}_i(\mathbf{x}, \mathbf{v}) \tag{2.1}$$

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 graph node, as shown in the right of Figure 2.2. Each component is responsible for a small number of tasks implemented using virtual functions in an object-oriented approach. Each operator in Equation 2.1 corresponds to a component. *MeshLoader* is used to read the topology and the geometry. 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 *MechanicalObject*, which is the core component of the mechanical model. A tetrahedral connectivity is stored in *TetrahedronSetTopologyContainer*, and made available to other components such as *TetrahedralCorotationalFEMForceField*, which accumulates one of the terms of the force sum using the Finite Element method. The two other terms come from *SpringForceField*, which accumulates the forces generated by the 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 squared particles. *EulerSolver* implements the logic of time integration.

This approach is highly modular because the components are completely independent of each other and are implemented using C++ classes with a reduced number of abstract functions. For instance, in the example of figure 2.2, if one want to use a FEM for the membrane force instead of the spring based

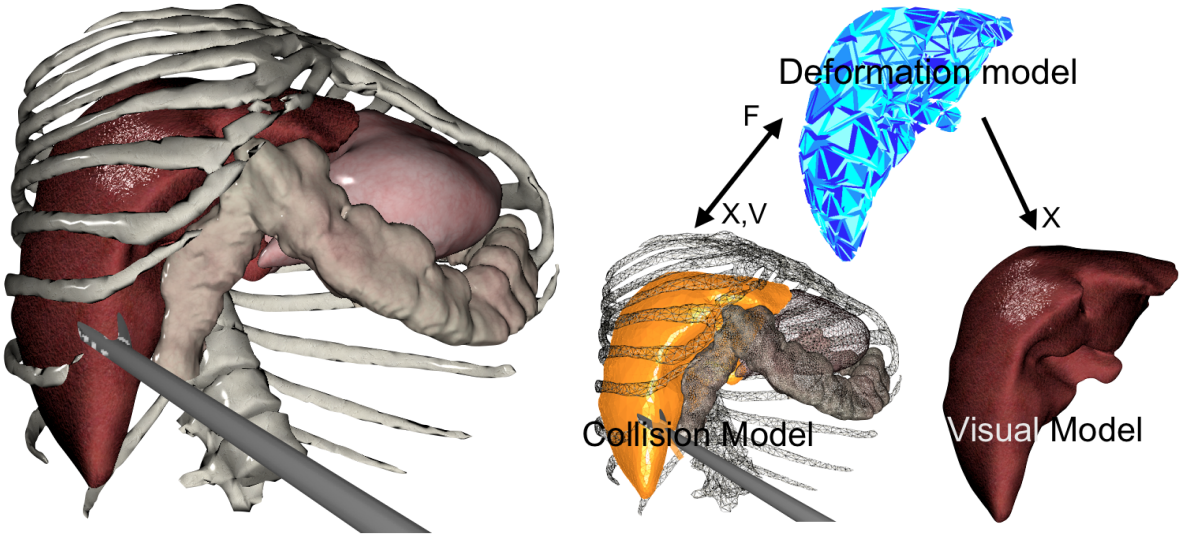


Figure 2.1: A simulated Liver Left: The simulation of the liver (dataset from IRCAD, France). Right: Three representations are used for the liver: one for the internal mechanics, one for the collisions, and one for the visualization. These representations are linked using mappings (black arrows).

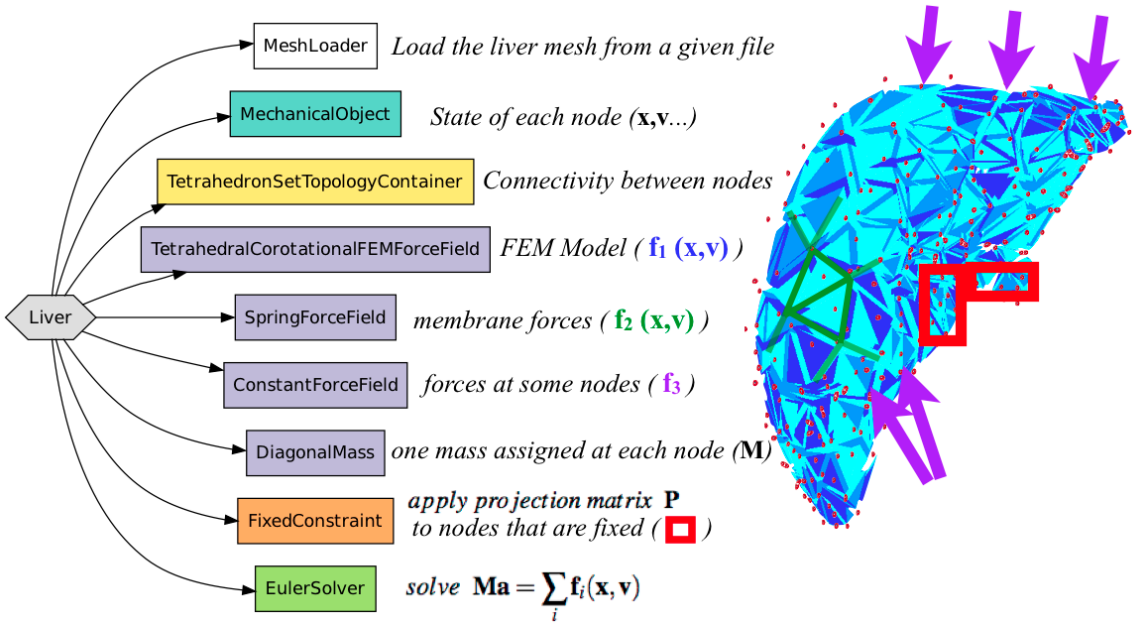


Figure 2.2: Mechanical model of a liver. In order to facilitate the combination of models and algorithms, the liver is described as a composition of specialized components.

computation, only *SpringForceField* has to be changed for *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 solver for instance.

For efficiency, each mechanical state vector contains the values of all the simulation nodes, to avoid multiple call of virtual function resolutions. The vector size is basically the number of particles times the number of space dimensions. We use C++ templates to avoid code redundancy between scalar types (float, double), the types of degrees of freedom (particles, frames, generalized coordinates), and the number of space dimensions. All the particles in a vector have the same type known at compile time. Degrees of freedom of different types must be grouped in different objects, possibly connected with interaction forces, as discussed in Section 5.2. This greatly simplifies the design and allows aggressive compiler optimizations.

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 springs 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 2.4.

2.2 Collision models

When a lot of primitives comes 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-tracing [12], and intersection volume using images [2] The collision pipeline is described in section **??** with more details.

In order to adapt the models to the data structure of the different collision algorithms, we have defined a *collision model*. This model is similar to a mechanical 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 the collision of a given simulation takes too much time, or to reduce the number of collision points, the meshes used for collision detection can be chosen less detailed than the mechanical ones. In the opposite, if precise collision detection and response is needed with smooth surfaces, it is sometimes suitable to use more detailed mesh for collision detection.

2.3 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 disconnected from the models used for the simulation. The mappings described in section 2.4 maintain the coherency between them. Hence, SOFA simulation results can easily be displayed using models much more detailed than used for internal mechanics, and rendered using external libraries such as OGRE[1] and Open Scene Graph[2].

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

[1] www.ogre3d.org
[2] www.openscenegraph.org

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.

2.4 Mappings

As previously discussed, objects simulated in SOFA, like the liver in Figure 2.1, 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 2.3 illustrates the hierarchies of two objects. The visual models, in additional branches, are omitted for clarity. When contact models collide, additional geometry is necessary to model the contacting points. This additional geometry is represented in an additional level connected to the models, as depicted in the figure.

The positions $\mathbf{x}_c$ of a child model are computed by the mapping based on the positions $\mathbf{x}_p$ of the master using a function $\mathcal{J}$.

$$\mathbf{x}_c = \mathcal{J}(\mathbf{x}_p) \tag{2.2}$$

The velocities can be mapped in a similar way:

$$\mathbf{v}_c = \mathbf{J}\mathbf{v}_p \tag{2.3}$$

The Jacobian matrix $\mathbf{J} = \frac{\partial \mathbf{x}_c}{\partial \mathbf{x}_p}$ encodes the linear relation between the parent and child velocities. It also holds on accelerations, with an additional offset due to velocities when the position mapping $\mathcal{J}$ is nonlinear. In linear mappings, operators $\mathcal{J}$ and $\mathbf{J}$ are the same, otherwise $\mathcal{J}$ is nonlinear with respect to x_p and it can not be written as a matrix. For surfaces embedded in deformable cells, matrix $\mathbf{J}$ contains the barycentric coordinates (it corresponds to linear interpolation in FEM). 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, up to the independent DOFs, where Newton's law $\mathbf{f} = \mathbf{Ma}$ is applied. Given forces $\mathbf{f}_c$ applied to a child model, the mapping computes and accumulates the equivalent forces $\mathbf{f}_p$ applied to its parent. Since equivalent forces must have the same power, the following relation holds:

$$\mathbf{v}_p^T \mathbf{f}_p = \mathbf{v}_c^T \mathbf{f}_c$$

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

$$\mathbf{v}_p^T \mathbf{f}_p = \mathbf{v}_p^T \mathbf{J}^T \mathbf{f}_c$$

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

$$\mathbf{f}_p = \mathbf{J}^T \mathbf{f}_c \tag{2.4}$$

When a model has several children, each child accumulates its contribution to the parent forces 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 be used to connect generalized coordinates, such as joint angles, to world-space geometry, as in the grasper of Figure 2.3.

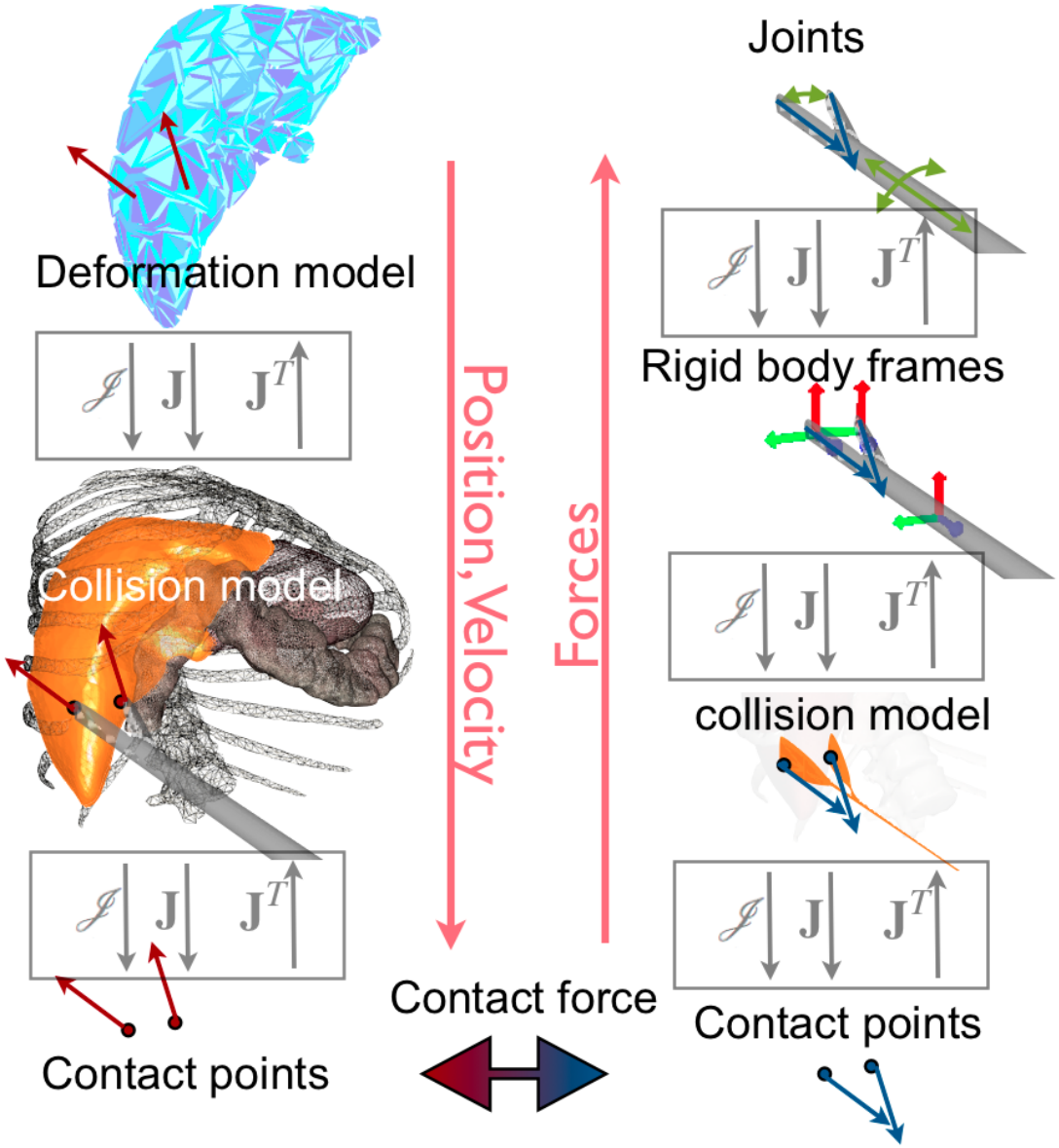


Figure 2.3: Mappings from the DOFs to the contact point. Right (top to bottom): The Mechanical model of the liver is based on Finite Element model. A triangular mesh is mapped for collision detection with the surface. The two contact points found by the collision detection (with the grasper) are mapped on the collision model. Left (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.

Chapter 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 coarsest structure is the scenegraph, used to hierarchically organize the groups of objects and their multi-models. At a finer scale, the attributes of the components can be linked by relations. Finally, the geometrical models and the topological changes deserve a special attention.

3.1 Scene-Graph

When the simulation involves several objects, we model them as different branches in a scenegraph data structure. In the example shown in Figure 3.1, the scene contains two objects animated using different time integrators, collision detection components (discussed in Section **??**), an interaction force, and a camera to display the objects. The root node does not contain DOFs. It is used to contain the components which are common to its child nodes.

Scenegraphs are popular in Computer Graphics due to their versatility. The data structure is processed using visitors (discussed in Section 3.1) which apply virtual functions to each node they traverse, which in turn apply virtual functions to the components they contain. In basic scenegraph frameworks, the visitors are exclusively 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 2.1, in components. This neatly decouples the physical model from the simulation algorithms, in sharp contrast with dataflow graphs which intricate data and operators 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 may 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 component being shared between the two objects, it is located in their common ancestor node. The coupling created by penalty forces should be seen as soft or stiff, depending on the stiffness and the size of time step [3]. 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 together. 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.

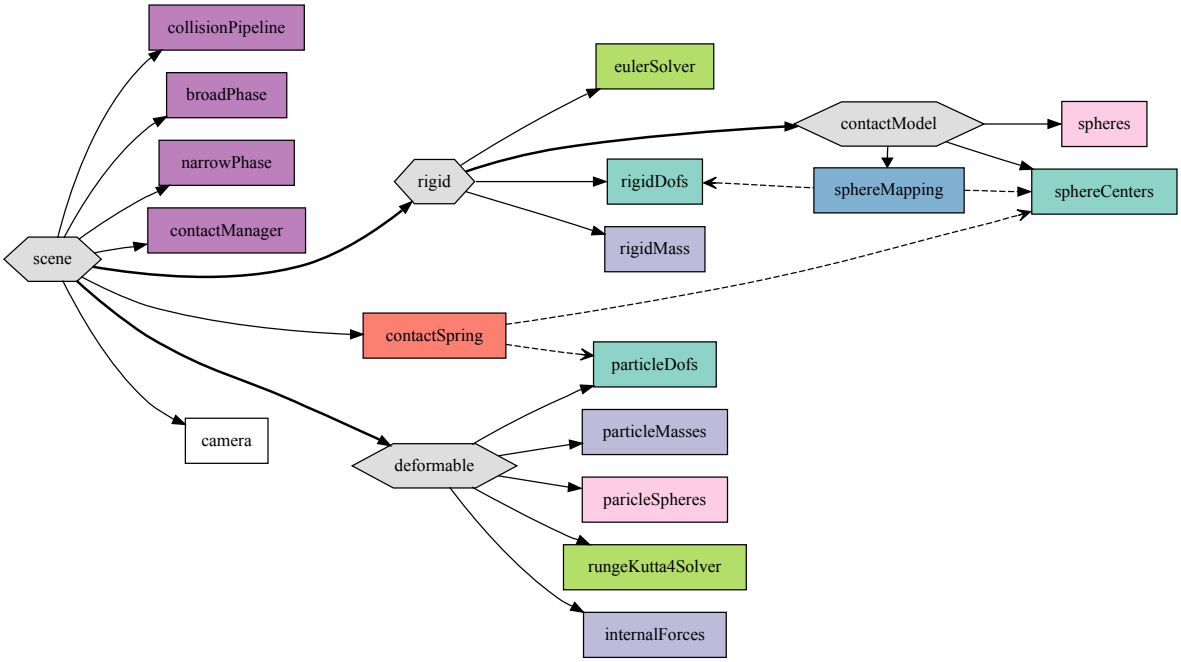


Figure 3.1: A scenegraph with collision detection and two independent objects interacting through a spring.

Visitors

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 3.2. 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. The data structure can easily be generalized from strict hierarchies to directed acyclic graphs for more general kinematic dependencies. Moreover, various parallelism strategies can be applied independently of the mechanical computations performed at each node.

Forward time stepping is implemented using the *AnimateVisitor* traversal method, shown in the right of Figure 3.2. Applied to the simple scene in Figure 3.3, 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 a ODE solver is encountered. This allows the ODE solver to take control of its subgraph and to overload lower-level solvers, which are not reached by the AnimateVisitor. In the more complex scene shown in Figure 3.1, the solver 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, a *MasterSolver* 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.

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

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

Figure 3.2: Left: a recursive implementation of the visitor traversal. Right: the AnimateVisitor.

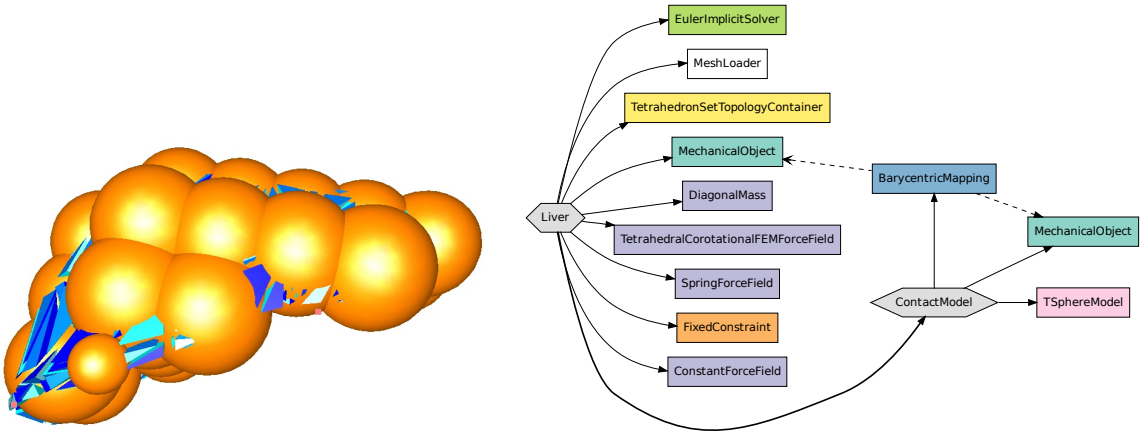


Figure 3.3: Left: simple mechanical (in blue) and collision (in yellow) models of a liver. Right: the corresponding scene graph. The plain arrows denote hierarchy, while the stippled arrows represent connections.

3.2 Data, Engines and Tags

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 **??**. We can create connection 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 Data. 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 scene graph. This two-graph framework is used in other graphics software such as OpenInventor and Maya, where engines are used to generate the animation, by periodically updating the state vectors using time as input. However, while this approach works well for straightforward computation pipelines, such as keyframe interpolation, it does not easily allow the branching and loops control structures used in 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 instead determined by combining several components, communicating through scenegraph visitors, as explained in Section **??**.

Objects and node tagging (Tag and TagSet).

The goal of the introduction of tags is to provide one of the pieces necessary to support non-mechanical states (electrical potentials, constrast agent concentrations) as well as cleaner non-geometrical mechanical states (fluid dynamics, reduced-coordinate articulations). For example, in a simulation involving blood in deformable vessels, we would use two tags to distinguish the different states : mechanical, fluid. These tags will be used to easily work with only a subset of the components, so that the mechanical solver works on positions and forcefields but don't interferes with blood flow and pressure, and inversely for the fluid solver (see [1]). We decided on using there tags instead of extending the class hierarchy as was done before with the State and MechanicalState classes. A hierarchy is fine when we have only one feature that we want to differentiate on (such as base vs mechanical vs electrical), but when we add other criteria (lagrangian geometry vs eulerian vs reduced generalized coordinates, velocity vs vorticity, independent vs mapped DOFs) it is no longer manageable as specialized classes. A secondary use of these tags is to replace existing subsets mechanisms within CollisionModels (r2441) and Constraints (r3121). The design is based on the following elements. Tags are added to BaseObject, as a list of string (internally converted to a list of unique ids for faster processing). All visitors now filter the objects they process based on their list of tags. All solvers by default copy their own list of tags to the visitors they execute, so that they only affect the objects with the same tags as they have (TODO: this is currently broken).

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.

[1] http://wiki.sofa-framework.org/tdev/wiki/Notes/ProposalGenericStates

Keeping a modular design implies that mesh related information (such as mechanical or visual properties) is not centralized in a single mesh data structure but is instead spread out in the software components that are using this information. 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. 12.9) 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. 12.9) 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.

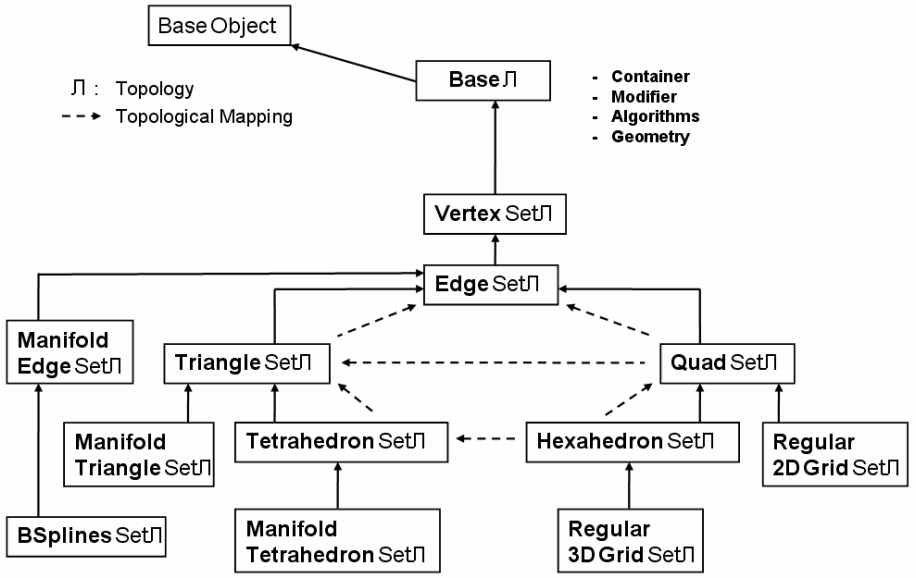


Figure 3.4: Hierarchy of mesh topology. Dashed arrows indicate possible *Topological Mappings* from a topology object to another.

Topology objects are composed of four functional members: *Container*, *Modifier*, *Algorithms* and *Geometry*. The *Container* member creates and updates when needed two complementary arrays. The former describes the l-cells included in a single k-cell, $l < k$, while the latter gives the k-cells adjacent to a single l-cell. From these arrays, generic methods give access to both full topological element description and complete adjacency information. The *Modifier* member provides low-level methods that implement elementary topological changes such as the removal or addition of an element. The *Algorithms* member provides high-level topological modification methods (cutting, refinement) which decompose complex tasks into low-level ones). The *Geometry* member provides geometrical information about the mesh (e.g. length, normal, curvature, ...) and requires the knowledge of the vertex positions stored in the *Degrees of Freedom* component.

Another important concept introduced in SOFA is the notion of *Topological Mapping*. Those mappings define a mesh topology in a from another mesh topology using the same DOFs. For instance, one may

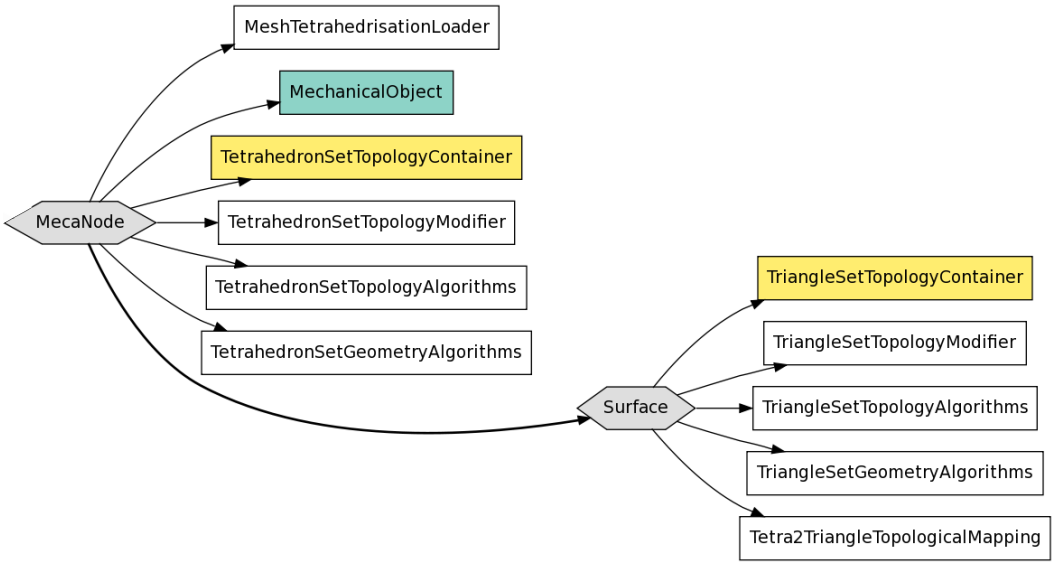


Figure 3.5: Scene Graph of a simple tetrahedral mesh where a topological mapping is defined to have the set of triangles located at the surface of the volumetric mesh.

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 the list of triangles on the border of a tetrahedral surfaces (see Figure 3.5). 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 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 an tetrahedral mesh from a set of hexahedra or to create triangular meshes from quads.

Mesh Data Structure

Containers storing mesh information (material stiffness, list of fixed vertices, nodal masses, ...) are stored in *components* and are spread out in the simulation tree. Most containers are simple arrays with contiguous memory storage and a 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 container consistent with topological changes is automated.

There are as many containers 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. Finally, two other types of containers with similar characteristics have been defined. The former stores a data structure for a subset of topological elements (for instance pressure on surface triangles in a tetrahedralisation) while the latter stores only a subset of element indices.

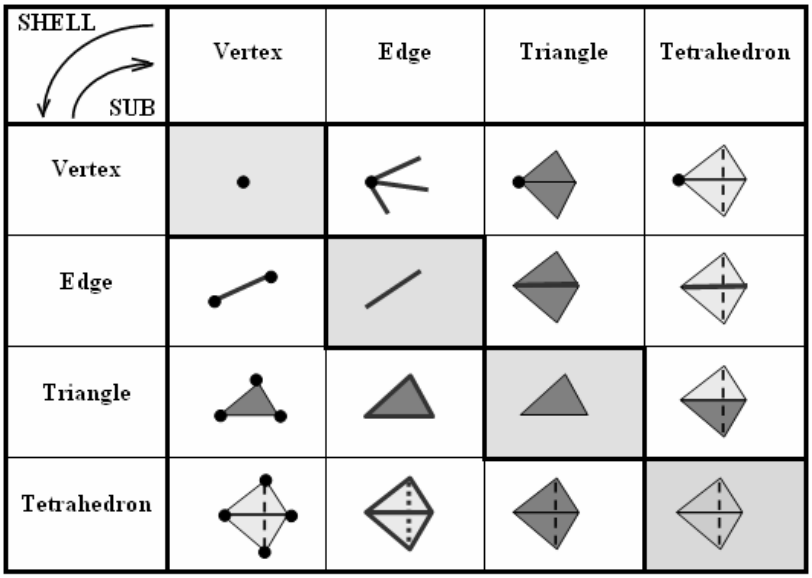


Figure 3.6: The two topological arrays stored in a *Container* correspond to the upper and lower triangular entries of this table. The upper entries provide the k-cells adjacent to a l-cell, $l < k$. The lower entries describe the l-cells included in a k-cell. Similar table exists for quad and hexahedron elements.

Chapter 4

States

4.1 States

The State component is used to store the states vector of a physical object. For a mechanical object, it would be the position, velocity, forces, rest position... Depending on the type of solvers (see chapter 7), additional informations can be stored inside the states, for instance a *mid-step* position if needed in the time integration scheme, or a *free position* that correspond to the position without any constraints if needed during the constraint solving process.

.. to be completed...

4.2 Degrees of Freedom using template

4.3 Data manipulation

Chapter 5

Mechanical forces

to be completed

5.1 ForceField components

5.2 Interaction ForceField

5.3 Mass and inertial forces

Chapter 6

Mappings

6.1 Mapping functions

As seen in section **??**, the mappings propagate positions, velocities, displacements and accelerations top-down, and they propagate forces bottom-up. The top-down propagation methods are:

- `apply (const MechanicalParams*, MultiVecCoordId outPos, ConstMultiVecCoordId inPos )` for positions,
- `applyJ(const MechanicalParams*, MultiVecDerivId outVel, ConstMultiVecDerivId inVel )` for velocities and small displacements,
- `computeAccFromMapping(const MechanicalParams*, MultiVecDerivId outAcc, ConstMultiVecDerivId inVel, ConstMultiVecDerivId inAcc )` for accelerations, taking into account velocity-dependent accelerations in nonlinear mappings.

The bottom-up propagation methods are:

- `applyJT(const MechanicalParams*, MultiVecDerivId inForce, ConstMultiVecDerivId outForce )` for child forces or changes of child forces,
- `applyDJT(const MechanicalParams*, MultiVecDerivId parentForce, ConstMultiVecDerivId childForce )` for changes of parent force due to a change of mapping with constant child force,
- `applyJT(const ConstraintParams*, MultiMatrixDerivId inConst, ConstMultiMatrixDerivId outConst )` for constraint Jacobians,

The name of the methods used to propagate velocities or small displacements top-down contain J, which denotes the kinematic matrix, while the names of the methods used to propagate forces or constraint Jacobians bottom-up contain JT, which denotes the transpose of the same. Method `applyJT(const MechanicalParams*, MultiVecDerivId inForce, ConstMultiVecDerivId outForce )` is used to accumulate forces from a child model to its parent. It performs a cumulative write (+=) since a model may have several children:

$$f_p += J^T f_c \tag{6.1}$$

Some differential equation solvers need compute the change of force df, given a change of position dx. The displacement dx, considered small, is propagated top-down using the linear operator `applyJ(const MechanicalParams*, MultiVecDerivId outVel, ConstMultiVecDerivId inVel )`, then the force changes are accumulated bottom-up. Differentiating eq.6.1, we get:

$$\delta f_p += J^T \delta f_c + \delta J^T f_c \tag{6.2}$$

Once the change of child force δf_c is computed (see the section on force fields), method `applyJT(const MechanicalParams*, MultiVecDerivId inForce, ConstMultiVecDerivId outForce )` is used to accumulate it in the parent, corresponding to the first term in the right of eq.6.2. Method `applyDJT(const`

`MechanicalParams*, MultiVecDerivId parentForce, ConstMultiVecDerivId childForce )` is used to accumulate the second term, which is due to the change of matrix J due to a displacement. It is null in linear mappings, such as BarycentricMapping. This method queries the last displacement propagated and the child force using the MechanicalParams.

Constraints enforced using Lagrange multipliers are represented using linear equations. If a linear constraint on the child DOFS in expressed as $L_c v_c = a$, where L_c is the Jacobian of the constraint, then the equivalent constraint at the parent level is: $L_p v_p = a$, where $L_p = J^T L_c$. Method applyJT(const ConstraintParams*, MultiMatrixDerivId inConst, ConstMultiMatrixDerivId outConst) is used to compute L_p. Since the Jacobians are generally sparse, they are encoded in sparse matrices instead of the dense vectors used for forces.

6.2 Barycentric Mapping

6.3 Rigid Mapping

6.4 Identity Mapping

6.5 Skinning Mapping

Chapter 7

Solvers

7.1 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. They apply abstract mechanical operations to state vectors represented by IDs, as illustrated in the algorithm shown in Figure 7.1. Each mechanical operation, such as allocating a state vector or accumulating the forces, is implemented using a specialized visitor parameterized on vector IDs or control values such as dt. This allows to implement the solvers completely independenly of the physical model. Each vector used by a solver ID is actually scattered over all the state vector containers in the different nodes in the scope of the solver. Some vector operations such as the dot product apply only to the independent DOFs, stored in the state vectors not attached to a parent by a mapping. Notice that this design avoids the 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.

We have identified two families of ODE solvers. The first contains the explicit solvers, which compute the derivative at the beginning of the time step. They are variants of the Euler explicit solver presented in Figure 7.1, and are easily implemented in Sofa using the same operators. The second family contains the implicit solvers, which consider the derivative at the end or somewhere in the middle of the time step. They typically require the solution of equation systems such as:

$$\underbrace{(\alpha\mathbf{M} + \beta\mathbf{B} + \gamma\mathbf{K})}_{\mathbf{A}}\,\delta\mathbf{v} = \mathbf{b} \tag{7.1}$$

where $\mathbf{M}$ is the mass matrix, $\mathbf{K} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ 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

```
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
```

Figure 7.1: Euler's explicit time integration.

```
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
mapping.applyJT(this.df)
```

Figure 7.2: Computing *df* given *dx* using a visitor.

matrix $\mathbf{P}$ can be used, and the system becomes $\mathbf{P}^T\mathbf{AP}\delta\mathbf{v} = \mathbf{P}^T\mathbf{b}$ [3]. Implicit integration has the advantage of being more stable for stiff forces or large time steps. However, solving 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.

7.2 Linear solvers

7.2.1 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 7.2 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:

$$\mathbf{df} = \sum_i \left(\prod_{j \in path(i)} \mathbf{J}_j \right)^T \mathbf{K}_i \left(\prod_{j \in path(i)} \mathbf{J}_j \right) \mathbf{dx} \tag{7.2}$$

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 nodes from the solver to the node the force applies to. This complex product is 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. This method allows us to trade-off accuracy for speed by limiting the number of steps of the iterative solution.

7.2.2 Direct Solvers

Direct solvers are also available in SOFA. They can be used as preconditionners of the conjugate gradient algorithm [4] or for directly solving equation 7.1. 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, like 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.

7.2.3 From ODE solver to linear solver

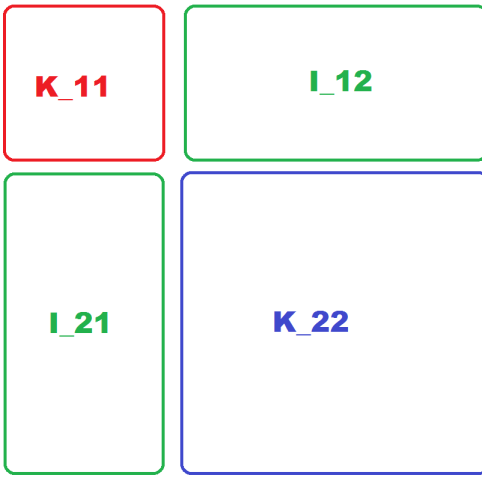
In SOFA, all states of a mechanical object is described by its degree of freedom. The main works for the simulation are filling and inverting a matrix system in order to find the states of mechanical objects by steps of time. This system matrix can be described as below :

$$[\mathbf{MBK}]\,.a = f \text{ , or at time n+1 : } [\mathbf{MBK}]\,.a_{n+1} = f_{n+1}$$

where,

$$\begin{cases} \mathbf{M} : \text{the mass matrix} \\ \mathbf{B} : \text{the damping matrix} \\ \mathbf{K} : \text{the stiffness matrix} \\ [\mathbf{MBK}] : \text{is a linear combination of MBK (not multiplication)} \\ a_{n+1} : \text{accelerator field} \\ f_{n+1} : \text{force field} \end{cases}$$

Usually, **M** is filled by **mass** components, **K** is filled by **forcefield** or **interactionforcefield** components, **B** (often $\alpha\mathbf{M} + \beta\mathbf{K}$) and [**MBK**] are computed by **odesolver** components. The works left to invert the matrix system are done by **linearsolver** components.



On the case for example when there are two mechanical objects, we can see a global stiffness matrix describing the two mechanical states, composed diagonal blocs and non-diagonal blocs. The diagonal blocs are filled by **mass**,**forcefield** components, and the non-diagonal ones are filled by **interactionforcefield** if existed.

Mapping matrix contribution : When existe a mapping on the simulation scene, the states of two mechanical objects are relied by :

$$\begin{array}{rl} \mathbf{x}_2 & = \Im\left(\mathbf{x}_1\right) \text{ , mapping::apply} \\ \mathbf{v}_2 & = [\mathbf{J}]\,\mathbf{v}_1 \text{ , mapping::applyJ} \end{array}$$

The [**J**] matrix is derivative of $\Im$ operator and is defined by the **mapping** components. The dynamic and matrix system of the two objects are relied by :

$$\begin{array}{rl} \mathbf{f} & += \mathbf{f}_1 + [\mathbf{J}]^t\,\mathbf{f}_2 \text{ , mapping::applyJT} \\ [\mathbf{MBK}] & += [\mathbf{MBK}]_1 + [\mathbf{J}]^t\,[\mathbf{MBK}]_2\,[\mathbf{J}] \end{array}$$

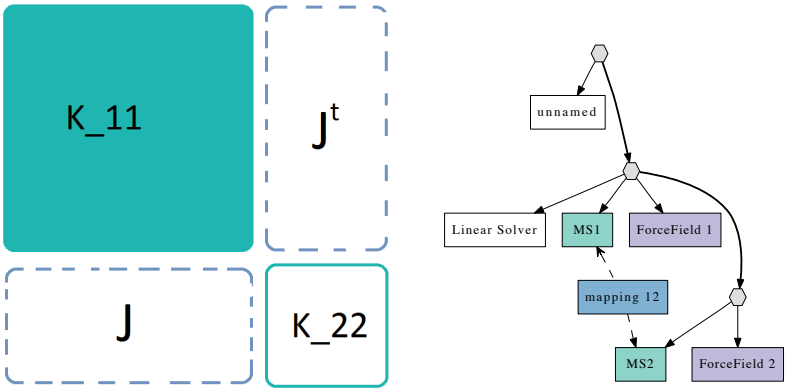
The resolution of the system with the mapping is done in general :

$$\begin{cases} \mathbf{a}^{n+1} & = [\mathbf{MBK}]^{-1}\,\mathbf{f}^{n+1} \\ \mathbf{v}_1^{n+1} & = \mathbf{v}_1^n + dt.\mathbf{a}^{n+1} \\ \mathbf{x}_1^{n+1} & = \mathbf{x}_1^n + dt.\mathbf{v}^{n+1} \\ \mathbf{x}_2^{n+1} & \text{ , mapping::apply} \\ \mathbf{v}_2^{n+1} & \text{ , mapping::applyJ} \end{cases}$$

7.2.4 Particular implementation in SOFA

The direct solver demands to build explicitly the matrix, and invert this matrix after every step of time in order to solve the mechanical response after a solicitation. In SOFA, there are a little more complicated component called mapping, relying geometrical and mechanical properties by master-slave (DOF-mapped object) relation. All changes of geometry or solicitation to one object interfere to other and vice versa. If the mapped object have its own mechanical behavior, it must be counted on the mechanical propagation by the mapping.

self-stiffness propagation



In the simple simulation scene, **MS2** is a mapped object to the **MS1** mechanical object by the mapping. The matrix to be inverted for all mechanical response is the filled colorized one (K_{11}), the matrix K_{22} describing mechanical properties of the second objects must contribute to K_{11} by the formula :

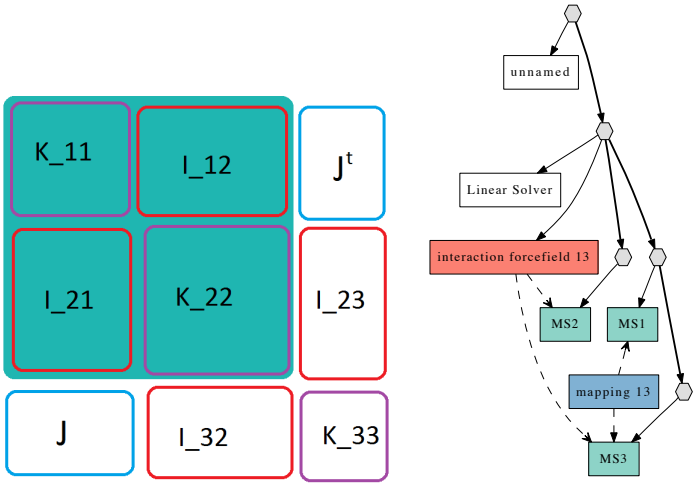
$$\begin{cases} K_{11} & += J^t * K_{22} * J \\ \text{or,} & \\ K_{tempo} & = J^t * K_{22} \\ K_{11} & += K_{tempo} * J \end{cases}$$

By doing this computation, we propagate the stiffness of the mapped mechanical object to its root mechanical object.

interaction-stiffness propagation

In the general case, the may have a simulation scene where there are many level of mapped mechanical states (mapped of mapped state ...) and many interaction forcefield interacting between them. Therefor the stiffness of interaction forcefield and the mapped mechanical state need to be propagated through the mappings. We can imagine for one propagation, there are two simple cases.

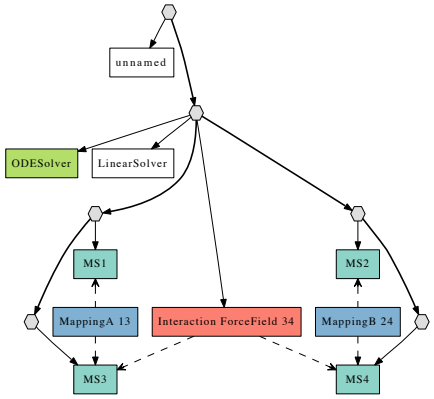
Interaction beweent Real Mechanical Object and Mapped Mechanical Object



In the case where one of the two mechanical states in interaction is non-mapped, the propagation can be computed directly by the formula :

$$\begin{cases} K_{11} & += J^t * K_{33} * J \\ \text{or,} & \\ K_{tempo} & = J^t * K_{33} \\ K_{11} & += K_{tempo} * J \\ \text{and,} & \\ I_{12} & += J^t * I_{32} \\ I_{21} & += I_{23} * J \end{cases}$$

Interaction beweent Mapped Mechanical Object and Mapped Mechanical Object



In the case where the two mechanical states in interaction are mapped, the propagation can be computed by two steps. The first consist to propagate the interaction I_{34} to the interation I_{14} :

$$\begin{cases} K_{11} & += J^t * K_{33} * J \\ \text{or,} & \\ K_{tempo} & = J^t * K_{33} \\ K_{11} & += K_{tempo} * J \\ \text{and,} & \\ I_{14} & += J_A^t * I_{34} \\ I_{41} & += I_{43} * J_A \end{cases}$$

The following step can compute as the one of above paragraph, propagating the interation I_{14} to I_{12}.

7.3 Constraint solvers

To handle different kinds of interactions (contact, friction, joints between particles..) between the simulated objects, SOFA allows the use of Lagrange multipliers [8]. 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.)[1]:

$$\begin{array}{l} \Phi(\mathbf{x}_1, \mathbf{x}_2, ...) = 0 \\ \Psi(\mathbf{x}_1, \mathbf{x}_2, ...) \geq 0 \end{array} \tag{7.3}$$

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 (7.1):

$$\begin{array}{l} \mathbf{A}_1 \delta \mathbf{v}_1 = \mathbf{b}_1 + \mathbf{H}_1^T \lambda \\ \mathbf{A}_2 \delta \mathbf{v}_2 = \mathbf{b}_2 + \mathbf{H}_2^T \lambda \end{array} \tag{7.4}$$

where

$$\mathbf{H}_1 = [\frac{\delta \Phi}{\delta \mathbf{x}_1} \ ; \ \frac{\delta \Psi}{\delta \mathbf{x}_1}] \qquad \mathbf{H}_2 = [\frac{\delta \Phi}{\delta \mathbf{x}_2} \ ; \ \frac{\delta \Psi}{\delta \mathbf{x}_2}]. \tag{7.5}$$

Matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ are stored in the mechanical state component of each node. Thus, when the constraint applies to a model that is mapped (see section 2.4), the constraints are recursively mapped upward like forces to be applied to the independent degrees of freedom [7]. 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 (7.4) with $\lambda = 0$ independently using a dedicated solver.

Step 2, Constraint Solving: The constrained equations can be linearized and linked to the dynamics (see [6] for details).

$$\begin{bmatrix} \Phi(\mathbf{x}_1, \mathbf{x}_2) \\ \Psi(\mathbf{x}_1, \mathbf{x}_2) \end{bmatrix} = \begin{bmatrix} \Phi(\mathbf{x}_1^{\mathrm{f}}, \mathbf{x}_2^{\mathrm{f}}) \\ \Psi(\mathbf{x}_1^{\mathrm{f}}, \mathbf{x}_2^{\mathrm{f}}) \end{bmatrix} + \underbrace{h\mathbf{H}_1 \delta \mathbf{v}_1^{\mathrm{c}} + h\mathbf{H}_2 \delta \mathbf{v}_2^{\mathrm{c}}}_{h\left[\mathbf{H}_1 \mathbf{A}_1^{-1} \mathbf{H}_1^T + \mathbf{H}_2 \mathbf{A}_2^{-1} \mathbf{H}_2^T\right]\lambda} \tag{7.6}$$

With $\delta \mathbf{v}^{\mathrm{c}} = \delta \mathbf{v} - \delta \mathbf{v}^{\mathrm{f}}$. Together with equation (7.3), 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 [9].

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

$$\begin{array}{l} \mathbf{x}_1^{t+h} = \mathbf{x}_1^{\mathrm{f}} + h\delta \mathbf{v}_1^{\mathrm{c}} \ \text{ with } \ \delta \mathbf{v}_1^{\mathrm{c}} = \mathbf{A}_1^{-1} \mathbf{H}_1^T \lambda \\ \mathbf{x}_2^{t+h} = \mathbf{x}_2^{\mathrm{f}} + h\delta \mathbf{v}_2^{\mathrm{c}} \ \text{ with } \ \delta \mathbf{v}_2^{\mathrm{c}} = \mathbf{A}_2^{-1} \mathbf{H}_2^T \lambda \end{array} \tag{7.7}$$

A Master Solver, which is generally placed at the top of the graph of SOFA has the role of imposing this new scheduling to the rest of the graph.

Compliance computation : Equations 7.6 and 7.7 involve the inverse of matrix $\mathbf{A}$ (called compliance matrix), which changes at every time step namely 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 hole matrix, or a precomputed inverse [14], or an asynchronous factorization on the GPU [5]. These strategies are implemented in a category of components, called *ConstraintCorrections* that provide different ways of computing $\delta \mathbf{v}^{\mathrm{c}}$ given a value of λ. Given a simulation, it is very easy to make tests and chose the better solution.

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

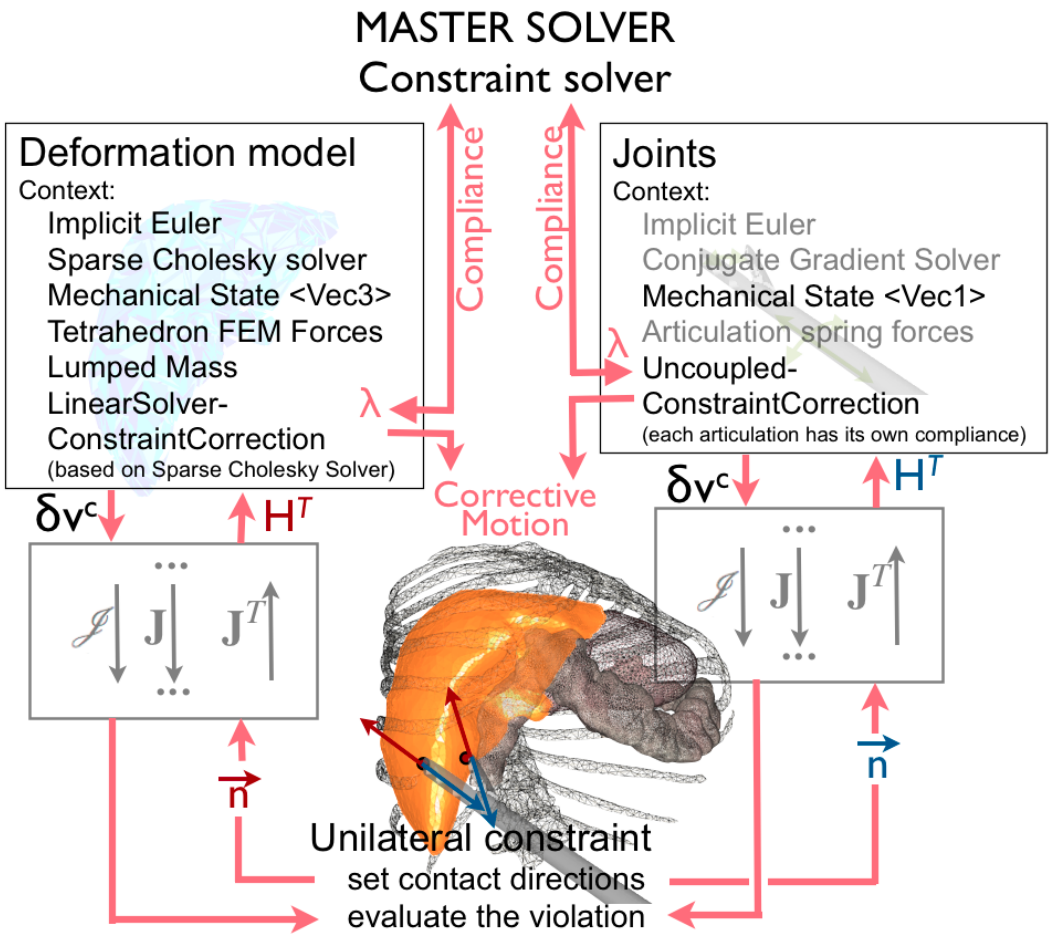


Figure 7.3: Contact process using constraints: A unilateral constraint is placed at the level of the contact points. The constraint direction is mapped to the degrees of freedom of the objects to obtain matrix $\mathbf{H}^T$. The *ConstraintCorrections* components compute the compliance to obtain equation 7.6. The Constraint solver found a new value of λ which is sent to the *ConstraintCorrections* to compute an adequate corrective motion. The Master Solver is placed at the root of the simulation graph to impose the steps of the simulation process.

Chapter 8

Collision detection

Collision detection is split in several phases, each implemented in a different component, and organized using a collision pipeline 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 boxes (currently, axis-aligned bounding boxes). Based on this, the narrow phase component returns pairs of geometric primitives with the corresponding collision 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 at the next time step. This framework has allowed us to efficiently implement popular proximity-based repulsion methods as well as novel approaches based on ray-casting [12] or surface rasterization [11, 2]. Its main limitation is that the contacts can be mechanically processed only after they all have 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.

When stiff contact penalties or contact constraints are created by the contact manager, an additional *GroupManager* component is used to create interaction groups handled by a common solver, as discussed in Section **??**. 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.

to be completed

Chapter 9

Visual Rendering

Chapter 10

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.

There are two main issues for a platform like SOFA for providing haptic: 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 can add artificially some energy inside the simulation and can create instabilities, if the control is not *passive*.

Thus two different approaches are currently implemented into SOFA. The first one is the *Virtual Coupling* technique and the other, more advanced, allows for rendering the constraints presented in section 7.3.

10.1 Virtual Coupling

Plugging 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 Fig10.1, could have been used. In this *Direct coupling* case, the simulation would play the role of a controller in an open loop.

Such 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 destabilize the system [10]. 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 Fig10.2). 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.

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 such case, the haptic feedback

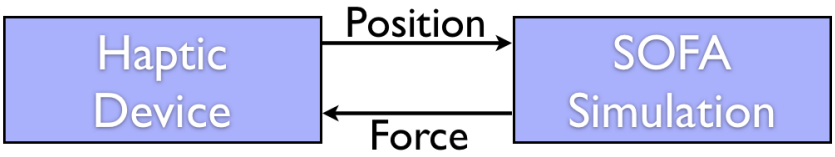


Figure 10.1: Direct coupling

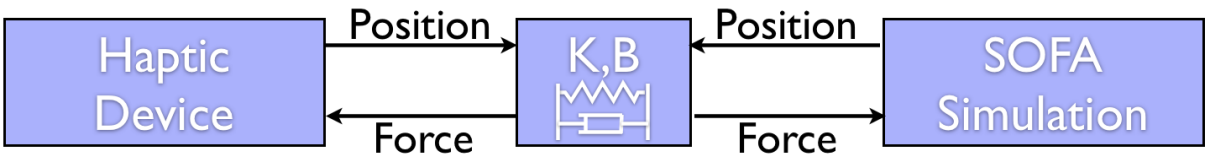


Figure 10.2: Virtual coupling technique. A 6 DoFs Damped spring is placed between the haptic loop and the simulation.

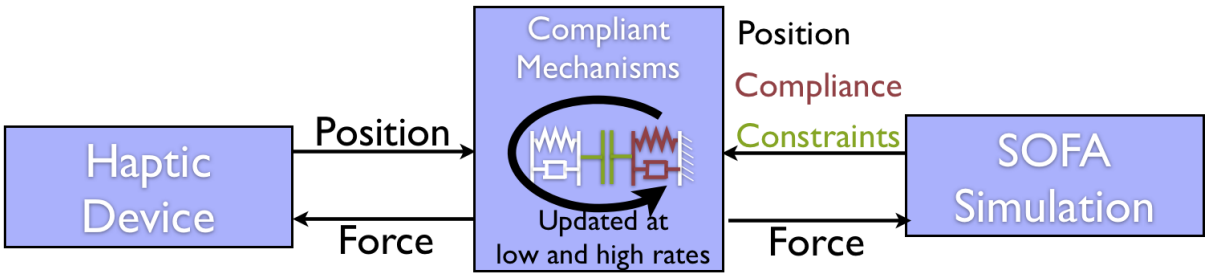


Figure 10.3: Compliant mechanisms technique. The simulation shares the mechanical compliance of the objects and the constraints between them. The constraint response is being computed at low rate within the simulation and at high rates within a separate haptic thread. A 6 DoFs Damped spring is still used to coupled the position of the device to its position in the simulation

remains stable but the delay between the two loop is creating an 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.

10.2 Constraint-based rendering

An innovative way of dealing with haptic rendering for medical simulation has been proposed in the context of SOFA (see[1] [14] and [13]). The approach deals with the mechanical interactions using appropriate force and/or motion transmission models named *compliant mechanisms* (see Fig10.3). These mechanisms are formulated as a constraint-based problem (like presented in section 7.3) 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.

10.3 How to use it in SOFA ?

Please see the web page: http://wiki.sofa-framework.org/wiki/Haptic and use the tutorial "dentistry".

[1]The implementation of [14] is available in open-source, for the implementation of [13], please contact: christian.duriez@inria.fr

Part II

Design and Development

Chapter 11

Design

11.1 Introduction

This chapter present the design of SOFA, as well as its recent evolutions. It also includes discussions justifying some of the decision that are behind this design.

The conventions used in UML class diagrams are presented in Fig 11.1.

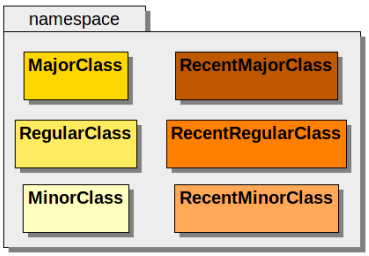


Figure 11.1: Color conventions in UML diagrams.

11.2 Core Class Diagrams

11.2.1 Object Model (sofa::core::objectmodel)

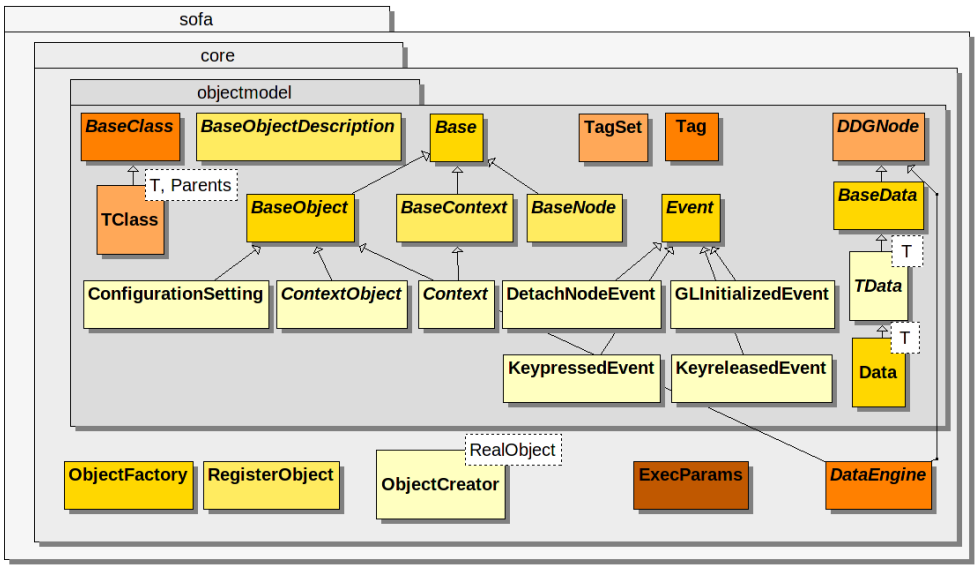


Figure 11.2: Classes of the sofa::core::objectmodel namespace.

Changes compared to the 1.0 beta 4 version

New class description system. It is based on the BaseClass and TClass classes, and requires to add the SOFA_CLASS macro in the declaration of all classes deriving from Base. The benefits are that it is now possible to follow the full hierarchy of classes from the final components, instead of having just a fixed set of categories. This macro is also necessary in classes with a templated parent class to be able to use methods and member variables defined in Base such as initData or sout. This removed all the previously redundant direct heritage to BaseObject that was previously required.

Objects and node tagging (Tag and TagSet). The goal of the introduction of tags is to provide one of the pieces necessary to support non-mechanical states (electrical potentials, constrast agent concentrations) as well as cleaner non-geometrical mechanical states (fluid dynamics, reduced-coordinate articulations). For example, in a simulation involving blood in deformable vessels, we would use two tags to distinguish the different states : mechanical, fluid. These tags will be used to easily work with only a subset of the components, so that the mechanical solver works on positions and forcefields but don't interferes with blood flow and pressure, and inversely for the fluid solver (see [1]). We decided on using there tags instead of extending the class hierarchy as was done before with the State and MechanicalState classes. A hierarchy is fine when we have only one feature that we want to differentiate on (such as base vs mechanical vs electrical), but when we add other criteria (lagrangian geometry vs eulerian vs reduced generalized coordinates, velocity vs vorticity, independent vs mapped DOFs) it is no longer manageable as specialized classes. A secondary use of these tags is to replace existing subsets mechanisms within CollisionModels (r2441) and Constraints (r3121). The design is based on the following elements. Tags are added to BaseObject, as a list of string (internally converted to a list of unique ids for faster processing). All visitors now filter the objects they process based on their list of tags. All solvers by default copy their own list of tags to the visitors they execute, so that they only affect the objects with the same tags as they have (TODO: this is currently broken).

Dependencies between Data (DDGNode and DataEngine). The goal is to be able to specify simple links between datas or through computation engines. To function correctly, the methods getValue() or beginEdit()/endEdit() are required to be called in all codes accessing values contained in Data instances. To enforce this, the class DataPtr was removed. Note also that it is very inefficient to call getValue() repetively within computation loops. Instead, the recommanded method is to use the helper classes ReadAccessor and WriteAccessor that can hold a reference to a Data value, provides the same API as regular vectors, and automatically calls endEdit() at the end of the call block in case of WriteAccessor.

TODO: TData<T> *was useful as a common parent for* Data<T> *and* DataPtr<T>*, but now that* DataPtr *no longer exist, it would simplify the hierarchy to merge* TData<T> *and* Data<T>*.*

***Copy-on-Write* (CoW) mechanism (DataContainer).** The goal is to reduce copies of datas when using engines and/or multi-threading. It is completly transparent to other codes, since everything should now respect the data access API (see above). There is however one important side-effect that may break some existing code: the pointer to the value of a Data can now change during a call to beginEdit(). This means that it is now illegal to retrieve the pointer to the value in a Data at init time and then reuse it after edits might have been made.

Support for asynchronous multi-threading (ExecParams). This feature is an important goal of the new design, however it is not yet completely functionnal and thus may change shortly.

11.2.2 Physical Behavior (sofa::core::behavior)

Changes compared to the 1.0 beta 4 version

VecId is replaced by templated classes to specify vector types and read/write access. Previously, the VecId class was used by solvers and visitors to specify on which state vectors should an

[1] http://wiki.sofa-framework.org/tdev/wiki/Notes/ProposalGenericStates

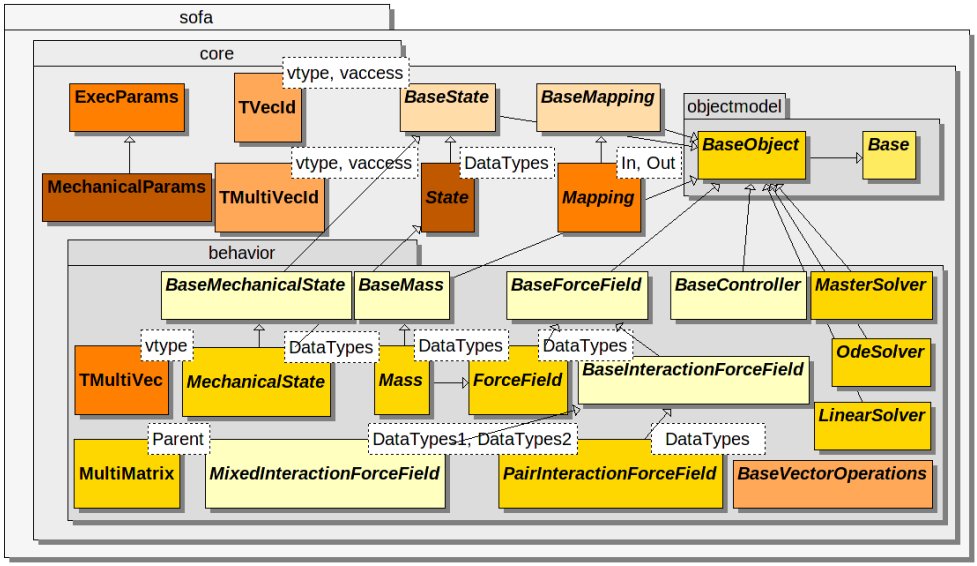


Figure 11.3: Classes of the sofa::core::behavior namespace.

operation be applied. However, this API did not express the requirements for these vectors (which type, i.e. Coord, Deriv, or MatDeriv), nor if they were supposed to be read or written. Now, methods and visitors can use the following classes instead (all typedefs from a common TVecId templated class), so that developers will now explicitly what to expect:

```
/// Identify one vector stored in State
/// A ConstVecId only provides a read-only access to the underlying vector.
typedef TVecId<V_ALL, V_READ> ConstVecId;

/// Identify one vector stored in State
/// A VecId provides a read-write access to the underlying vector.
typedef TVecId<V_ALL, V_WRITE> VecId;

/// Typedefs for each type of state vectors
typedef TVecId<V_COORD, V_READ> ConstVecCoordId;
typedef TVecId<V_COORD, V_WRITE> VecCoordId;
typedef TVecId<V_DERIV, V_READ> ConstVecDerivId;
typedef TVecId<V_DERIV, V_WRITE> VecDerivId;
typedef TVecId<V_MATDERIV, V_READ> ConstMatrixDerivId;
typedef TVecId<V_MATDERIV, V_WRITE> MatrixDerivId;
```

Also, a new class TMultiVecId is introduced to be able to specify different IDs for specific states or groups of states. Similar typedefs are defined as above, replacing Vec with MultiVec.

New State API and class hierarchy. The previous State API was based on getX()/getV()/... methods that returned pointers to the vectors last specified (using setX()/setV()/... methods) as the current position, velocity, ... This is now replaced by read(ConstVec*TYPE*Id) and write(Vec*TYPE*Id) methods, where *TYPE* is either Coord, Deriv, or MatDeriv. These methods return a pointer to a Data instance containing the vector, instead of the vector itself. This was necessary to respect the Data access API (see section 11.2.1). For compatibility with existing codes (especially draw() methods), the read-only getX()/getV()/... methods still exists but they are deprecated (and there are strictly equivalent to calling read() with the default ID for position, velocity, ...). The non-const versions however are removed, so all codes modifying state vectors will have to use the new Data-based API. Note that with this change, the state components no longer store the information about which vectors should be considered as the current position, velocity, ... So another mechanism is required to specify these associations (MechanicalParams, see below).

A new state class hierarchy is also defined, introducing a new parent class BaseState, common to all states (mechanical, visual, ...). Methods allowing to read and write vectors given their IDs are specified

in State<DataTypes>. The MappedModel class is removed, non-mechanical state components (such as visual models) now directly derive from State<DataTypes>.

Simplified Mapping class hierarchy. Previously, a different base class (Mapping< State<TIn>, MappedModel<TOut> > or MechanicalMapping< MechanicalState<TIn>, MechanicalState<TOut> >) where used for mechanical versus non-mechanical (i.e. visual or one-way) mappings. This introduced complexities and redundancies in the code of the final components (they where templated by the type of their parent class and compiled twice for each pair of input/output datatypes). Now, a single base class is used (Mapping<TIn,TOut>, templated only by the input and output datatypes. Components are also templated by the same types, similarly to other classes such as forcefields. A new set of flags is used to check if a given mapping is mechanical or not. Different flags are used to activate the mapping of forces, masses, and constraints separatly (although by default they have the same value). Components that do not implement applyJT (i.e. visual mappings) can force them to false to indicate they do not support mechanical computations.

New mechanical component API based on Data and MechanicalParams. This is the most important change from the previous design. To provide all the required vector IDs that where previously stored within each MechanicalState by calls to setX()/... methods, we define a MechanicalParam class. This class is given to all mechanical-related methods, specified by OdeSolvers and transmitted by MechanicalVisitors. It hides the VecId system from most component codes, providing the same abstraction of accessing the current position and velocity vectors as was previously handled within MechanicalState. For example, where a component such as a ForceField implementation used :

```
1 const VecCoord* x = this->mstate->getX();
```

It will now use MechanicalParams as follows :

```
1 helper::ReadAccessor<VecCoord> x = *mparams->readX(this->mstate);
```

This API is preferable to directly manipulating VecIds (such as in this->mstate->read(mparams->getXId(this->mstate))) because :

- The code is very similar to the previous version.
- If the API of the VecId or MechanicalState class is further changed, the MechanicalParam::readX() method can handle it.
- Different VecIds for different states can be chosen within MechanicalParam (such as what is currently used for free vs mapped states, and animated vs external obstacle state).

However it is still possible to get the ids (using mparams->x().getId(mstate)).

All vector accesses given though MechanicalParams are read-only. Vectors actually modified by each method are given as separate MultiVecId parameters. This is useful to make the API clearer (we can see explicitly what each method is supposed to write) and safer (no accidental modifications to X or V within addForce() or draw() for example). Finally, other interesting pieces of information can be queried though MechanicalParams. For example, a ForceField can know as soon as the addForce() call if the current solver is implicit or explicit, if the kinematic and potential energy should be computed, ...

New solver hierarchy and vector manipulation API (TMultiVec and BaseVectorOperations). Previously, the solver classes had a complex hierarchy, with several parent classes (SolverImpl, OdeSolverImpl, MasterSolverImpl) that where used to launch each type of visitors (mechanics, linear algebra, ...). In the new design, these classes are replaced by separate helper classes (VectorOperations, MechanicalOperations) that are instanciated at runtime and allow to launch each type of visitors while holding and updating the persistant parameters (in the form of a MechanicalParams for mechanical visitors for example, or ExecParams for simple vector operations).

11.2.3 Topology (sofa::core::topology)

The design of this namespace is still a work in progress, so major changes should still be expected...

Changes compared to the 1.0 beta 4 version

To be documented...

11.2.4 Collision (sofa::core::collision)

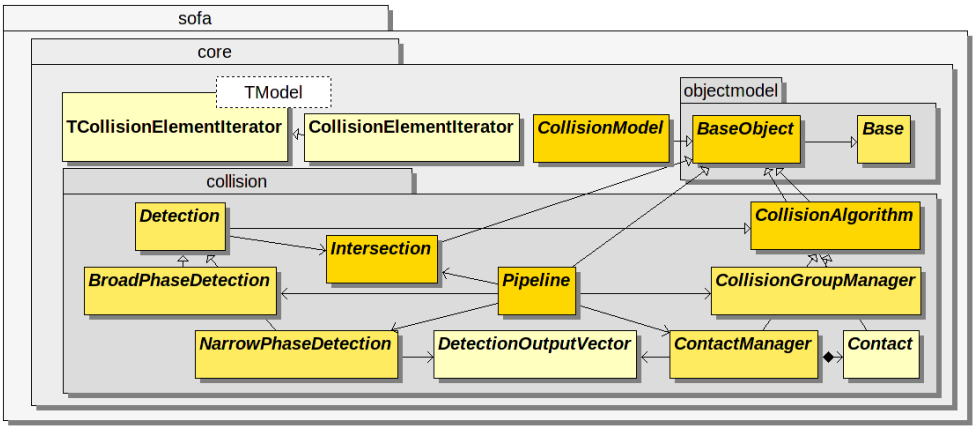


Figure 11.4: Classes of the sofa::core::collision namespace.

Changes compared to the 1.0 beta 4 version

No major changes.

11.2.5 Mesh and Image Loaders (sofa::core::loader)

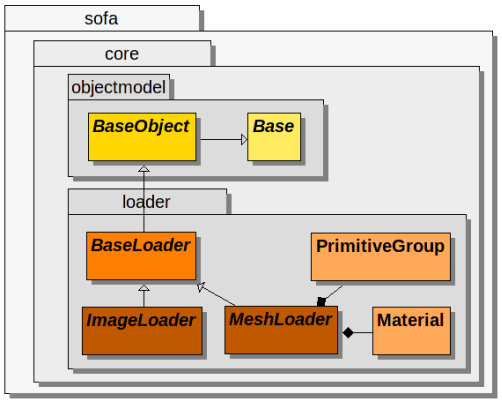


Figure 11.5: Classes of the sofa::core::loader namespace.

Changes compared to the 1.0 beta 4 version

This namespace is completely new.

To be documented...

11.3 Main classes

11.3.1 sofa::core::objectmodel

Base

```
{

/**
 * \brief Base class for everything
 *
 * This class contains all functionnality shared by every objects in SOFA.
 * Most importantly it defines how to retrieve information about an object (name, type, data fields).
 * All classes deriving from Base should use the SOFA_CLASS macro within their declaration (see BaseClass.h).
 *
 */
class SOFA_CORE_API Base
{
public:
    typedef TClass< Base, void > MyClass;
    static const MyClass* GetClass() { return MyClass::get(); }
    virtual const BaseClass* getClass() const { return GetClass(); }

    Base();
    virtual ~Base();

private:
    /// Copy constructor is not allowed
    Base(const Base& b);

public:



    /// Accessor to the object name
    std::string getName() const;

    /// Set the name of this object
    void setName(const std::string& n);

    /// Get the type name of this object (i.e. class and template types)
    virtual std::string getTypeName() const;

    /// Get the class name of this object
    virtual std::string getClassName() const;

    /// Get the template type names (if any) used to instantiate this object
    virtual std::string getTemplateName() const;

    /// @name fields
    ///   Data fields managment
    /// @{

    /// Parse the given description to assign values to this object's fields and potentially other parameters
    virtual void parse ( BaseObjectDescription* arg );

    /// Assign the field values stored in the given list of name + value pairs of strings
    void parseFields ( std::list<std::string> str );

    /// Assign the field values stored in the given map of name -> value pairs
    virtual void parseFields ( const std::map<std::string,std::string*>& str );

    /// Write the current field values to the given map of name -> value pairs
    void writeDatas (std::map<std::string,std::string*>& str);

    /// Write the current Node values to the given XML output stream
    void xmlWriteNodeDatas (std::ostream& out, unsigned level);

    /// Write the current field values to the given XML output stream
    void xmlWriteDatas (std::ostream& out, unsigned level, bool compact);

    /// Find a field given its name. Return NULL if not found. If more than one field is found (due to aliases), only the first
        is returned.
    BaseData* findField( const std::string &name ) const;

    /// Find fields given a name: several can be found as we look into the alias map
    std::vector< BaseData* > findGlobalField( const std::string &name ) const;

    /// Helper method used to initialize a field containing a value of type T
```

```
73      template<class T>
74      BaseData::BaseInitData initData( Data<T>* field, const char* name, const char* help, bool isDisplayed=true, bool
            isReadOnly=false )
75      {
76          BaseData::BaseInitData res;
77          this->initData0(field, res, name, help, isDisplayed, isReadOnly);
78          return res;
79      }
80
81      /// Helper method used to initialize a field containing a value of type T
82      template<class T>
83      typename Data<T>::InitData initData( Data<T>* field, const T& value, const char* name, const char* help, bool
            isDisplayed=true, bool isReadOnly=false )
84      {
85          typename Data<T>::InitData res;
86          this->initData0(field, res, value, name, help, isDisplayed, isReadOnly);
87          return res;
88      }
89
90      /// Add an alias to a Data
91      void addAlias( BaseData* field, const char* alias);
92
93      /// Accessor to the vector containing all the fields of this object
94      const std::vector< std::pair<std::string, BaseData*> >& getFields() const { return m_fieldVec; }
95      /// Accessor to the map containing all the aliases of this object
96      const std::multimap< std::string, BaseData* >& getAliases() const { return m_aliasData; }
97
98      /// @}
99
100     /// @name tags
101     ///  Methods related to tagged subsets
102     /// @{
103
104     /// Represents the subsets the object belongs to
105     const sofa::core::objectmodel::TagSet& getTags() const { return f_tags.getValue(); }
106
107     /// Return true if the object belong to the given subset
108     bool hasTag( Tag t ) const;
109
110     /// Add a subset qualification to the object
111     void addTag(Tag t);
112     /// Remove a subset qualification to the object
113     void removeTag(Tag t);
114
115     /// @}
116
117     /// @name logs
118     ///  Messages and warnings logging
119     /// @{
120
121     mutable sofa::helper::system::SofaOStream<Base> sendl;
122     mutable std::ostringstream                      serr;
123     mutable std::ostringstream                      sout;
124
125     const std::string& getWarnings() const;
126     const std::string& getOutputs() const;
127
128     void clearWarnings();
129     void clearOutputs();
130
131     void processStream(std::ostream& out);
```

BaseObject

```
1
2
3   namespace objectmodel
4   {
5
6   class Event;
7
8   /**
9    * \brief Base class for simulation objects.
10   *
11   * An object defines a part of the functionnality in the simulation
12   * (stores state data, specify topology, compute forces, etc).
13   * Each simulation object is related to a context, which gives access to all available external data.
14   * It is able to process events, if listening enabled (default is false).
15   *
```

```
16  */
17 class SOFA_CORE_API BaseObject : public virtual Base
18 #ifdef SOFA_SMP
19 , public BaseObjectTasks
20 #endif
21 {
22 public:
23     SOFA_CLASS(BaseObject, Base);
24
25     BaseObject();
26
27     virtual ~BaseObject();
28
29     /// @name Context accessors
30     /// @{
31
32     void setContext(BaseContext* n);
33
34     const BaseContext* getContext() const;
35
36     BaseContext* getContext();
37
38     /// @}
39
40     /// @name control
41     ///  Basic state control
42     /// @{
43
44     /// Pre-construction check method called by ObjectFactory.
45     template<class T>
46     static bool canCreate(T*& /*obj*/, BaseContext* /*context*/, BaseObjectDescription* /*arg*/)
47     {
48         return true;
49     }
50
51     /// Construction method called by ObjectFactory.
52     template<class T>
53     static void create(T*& obj, BaseContext* context, BaseObjectDescription* arg)
54     {
55         obj = new T;
56         if (context) context->addObject(obj);
57         if (arg) obj->parse(arg);
58     }
59
60     /// Parse the given description to assign values to this object's fields and potentially other parameters
61     virtual void parse ( BaseObjectDescription* arg );
62
63     ///  Initialization method called at graph creation and modification, during top-down traversal.
64     virtual void init();
65
66     ///  Initialization method called at graph creation and modification, during bottom-up traversal.
67     virtual void bwdInit();
68
69     /// Update method called when variables used in precomputation are modified.
70     virtual void reinit();
71
72     /// Save the initial state for later uses in reset()
73     virtual void storeResetState();
74
75     /// Reset to initial state
76     virtual void reset();
77
78     /// Called just before deleting this object
79     /// Any object in the tree bellow this object that are to be removed will be removed only after this call,
80     /// so any references this object holds should still be valid.
81     virtual void cleanup();
82
83     /// @}
84
85     /// Render internal data of this object, for debugging purposes.
86     virtual void draw(const core::visual::VisualParams*)
87     {
88 #ifndef SOFA_DEPRECATE_OLD_API
89         draw();
90 #endif
91     }
92     ///@}
93 #ifndef SOFA_DEPRECATE_OLD_API
94     virtual void draw() {}
95 #endif
96
```

```
97      /// @name data access
98      ///   Access to external data
99      /// @{
```

11.3.2 sofa::core

BaseState

```
1  /**
2   * \brief Component storing position and velocity vectors.
3   *
4   * This class define the interface of components used as source and
5   * destination of regular (non mechanical) mapping. It is then specialized as
6   * MechanicalState (storing other mechanical data) or MappedModel (if no
7   * mechanical data is used, such as for VisualModel).
8   */
9  class SOFA_CORE_API BaseState : public virtual objectmodel::BaseObject
10 {
11 public:
12     SOFA_CLASS(BaseState, objectmodel::BaseObject);
13
14     virtual ~BaseState() { }
15
16     /// Current size of all stored vectors
17     virtual int getSize() const = 0;
18
19     /// Resize all stored vector
20     virtual void resize(int vsize) = 0;
```

State

```
1  {
2
3  /**
4   * \brief Component storing position and velocity vectors.
5   *
6   * This class define the interface of components used as source and
7   * destination of regular (non mechanical) mapping. It is then specialized as
8   * MechanicalState (storing other mechanical data) or MappedModel (if no
9   * mechanical data is used, such as for VisualModel).
10  *
11  * The given DataTypes class should define the following internal types:
12  * \li \code Real \endcode : scalar values (float or double).
13  * \li \code Coord \endcode : position values.
14  * \li \code Deriv \endcode : derivative values (velocity).
15  * \li \code VecReal \endcode : container of scalar values with the same API as sofa::helper::vector.
16  * \li \code VecCoord \endcode : container of Coord values with the same API as sofa::helper::vector.
17  * \li \code VecDeriv \endcode : container of Deriv values with the same API as sofa::helper::vector.
18  * \li \code MatrixDeriv \endcode : vector of Jacobians (sparse constraint matrices).
19  *
20  */
21 template<class TDataTypes>
22 class State : public virtual BaseState
23 {
24 public:
25     SOFA_CLASS(SOFA_TEMPLATE(State,TDataTypes), BaseState);
26
27     typedef TDataTypes DataTypes;
28     /// Scalar values (float or double).
29     typedef typename DataTypes::Real Real;
30     /// Position values.
31     typedef typename DataTypes::Coord Coord;
32     /// Derivative values (velocity, forces, displacements).
33     typedef typename DataTypes::Deriv Deriv;
34     /// Container of scalar values with the same API as sofa::helper::vector.
35     typedef typename DataTypes::VecReal VecReal;
36     /// Container of Coord values with the same API as sofa::helper::vector.
37     typedef typename DataTypes::VecCoord VecCoord;
38     /// Container of Deriv values with the same API as sofa::helper::vector.
39     typedef typename DataTypes::VecDeriv VecDeriv;
40     /// Vector of Jacobians (sparse constraint matrices).
41     typedef typename DataTypes::MatrixDeriv MatrixDeriv;
42
43     virtual ~State() { }
44
```

```
45      /// @name New vectors access API based on VecId
46      /// @{
47
48      virtual Data< VecCoord >* write(VecCoordId v) = 0;
49      virtual const Data< VecCoord >* read(ConstVecCoordId v) const = 0;
50
51      virtual Data< VecDeriv >* write(VecDerivId v) = 0;
52      virtual const Data< VecDeriv >* read(ConstVecDerivId v) const = 0;
53
54      virtual Data< MatrixDeriv >* write(MatrixDerivId v) = 0;
55      virtual const Data< MatrixDeriv >* read(ConstMatrixDerivId v) const = 0;
56
57      /// @}
58
59      /// @name Old specific vectors access API (now limited to read-only accesses)
60      /// @{
61
62      /// Return the current position vector.
63      /// @deprecated use read(ConstVecCoordId::position()) instead.
64      virtual const VecCoord* getX() const
65      {
66          const Data<VecCoord>* v = read(ConstVecCoordId::position());
```

BaseMapping

```
1   /**
2    * \brief An interface to convert a model to an other model
3    *
4    * This Interface is used for the Mappings. A Mapping can convert one model to an other.
5    * For example, we can have a mapping from a BehaviorModel to a VisualModel.
6    *
7    */
8   class SOFA_CORE_API BaseMapping : public virtual objectmodel::BaseObject
9   {
10  public:
11      SOFA_CLASS(BaseMapping, objectmodel::BaseObject);
12
13      /// Constructor
14      BaseMapping();
15
16      /// Destructor
17      virtual ~BaseMapping();
18
19      Data<bool> f_mapForces;
20      Data<bool> f_mapConstraints;
21      Data<bool> f_mapMasses;
22      Data<bool> f_mapMatrices;
23
24      /// Apply the transformation from the input model to the output model (like apply displacement from BehaviorModel to
            VisualModel)
25      virtual void apply (const MechanicalParams* mparams /* PARAMS FIRST = MechanicalParams::defaultInstance()*/,
            MultiVecCoordId outPos = VecCoordId::position(), ConstMultiVecCoordId inPos = ConstVecCoordId::position() ) = 0;
26      virtual void applyJ(const MechanicalParams* mparams /* PARAMS FIRST = MechanicalParams::defaultInstance()*/,
            MultiVecDerivId outVel = VecDerivId::velocity(), ConstMultiVecDerivId inVel = ConstVecDerivId::velocity() ) = 0;
27
28      /// Accessor to the input model of this mapping
29      virtual helper::vector<BaseState*> getFrom() = 0;
30
31      /// Accessor to the output model of this mapping
32      virtual helper::vector<BaseState*> getTo() = 0;
33
34      // BaseMechanicalMapping
35      virtual void applyJT(const MechanicalParams* mparams /* PARAMS FIRST */, MultiVecDerivId inForce, ConstMultiVecDerivId
            outForce) = 0;
36      virtual void applyDJT(const MechanicalParams* mparams /* PARAMS FIRST */, MultiVecDerivId inForce,
            ConstMultiVecDerivId outForce) = 0;
37      virtual void applyJT(const ConstraintParams* mparams /* PARAMS FIRST */, MultiMatrixDerivId inConst,
            ConstMultiMatrixDerivId outConst) = 0;
38      virtual void computeAccFromMapping(const MechanicalParams* mparams /* PARAMS FIRST */, MultiVecDerivId outAcc,
            ConstMultiVecDerivId inVel, ConstMultiVecDerivId inAcc) = 0;
39
40      virtual bool areForcesMapped() const;
41      virtual bool areConstraintsMapped() const;
42      virtual bool areMassesMapped() const;
43      virtual bool areMatricesMapped() const;
44
45      virtual void setForcesMapped(bool b);
46      virtual void setConstraintsMapped(bool b);
47      virtual void setMassesMapped(bool b);
```

```
48
49      virtual void setNonMechanical();
50
51      /// Return true if this mapping should be used as a mechanical mapping.
52      virtual bool isMechanical() const;
53
54      /// Return true if the destination model has the same topology as the source model.
55      ///
56      /// This is the case for mapping keeping a one-to-one correspondance between
57      /// input and output DOFs (mostly identity or data-conversion mappings).
58      virtual bool sameTopology() const { return false; }
59
60      /// Get the (sparse) jacobian matrix of this mapping, as used in applyJ/applyJT.
61      /// This matrix should have as many columns as DOFs in the input mechanical states
62      /// (one after the other in case of multi-mappings), and as many lines as DOFs in
63      /// the output mechanical states.
64      ///
65      /// applyJ(out, in) should be equivalent to $out = J in$.
66      /// applyJT(out, in) should be equivalent to $out = J^T in$.
67      ///
68      /// @TODO Note that if the mapping provides this matrix, then a default implementation
69      /// of all other related methods could be provided, or optionally used to verify the
```

Mapping

```
1  /**
2  * \brief Specialized interface to convert a model of type TIn to an other model of type TOut
3  *
4  * This Interface is used for the Mappings. A Mapping can convert one model to an other.
5  * For example, we can have a mapping from a BehaviorModel to a VisualModel.
6  *
7  */
8
9  template <class TIn, class TOut>
10 class Mapping : public BaseMapping
11 {
12 public:
13     SOFA_CLASS(SOFA_TEMPLATE2(Mapping,TIn,TOut), BaseMapping);
14
15     /// Input Data Type
16     typedef TIn In;
17     /// Output Data Type
18     typedef TOut Out;
19
20     typedef typename In::VecCoord InVecCoord;
21     typedef typename In::VecDeriv InVecDeriv;
22     typedef typename In::MatrixDeriv InMatrixDeriv;
23     typedef Data<InVecCoord> InDataVecCoord;
24     typedef Data<InVecDeriv> InDataVecDeriv;
25     typedef Data<InMatrixDeriv> InDataMatrixDeriv;
26
27     typedef typename Out::VecCoord OutVecCoord;
28     typedef typename Out::VecDeriv OutVecDeriv;
29     typedef typename Out::MatrixDeriv OutMatrixDeriv;
30     typedef Data<OutVecCoord> OutDataVecCoord;
31     typedef Data<OutVecDeriv> OutDataVecDeriv;
32     typedef Data<OutMatrixDeriv> OutDataMatrixDeriv;
33
34 protected:
35         /// Input Model, also called parent
36     State< In >* fromModel;
37         /// Output Model, also called child
38     State< Out >* toModel;
39 public:
40     /// Name of the Input Model
41     //Data< std::string > object1;
42     objectmodel::DataObjectRef m_inputObject;
43     /// Name of the Output Model
44     //Data< std::string > object2;
45     objectmodel::DataObjectRef m_outputObject;
46
47     Data<bool> f_applyRestPosition;
48     Data<bool> f_checkJacobian;
49
50     /// Constructor, taking input and output models as parameters.
51     ///
52     /// Note that if you do not specify these models here, you must called
53     /// setModels with non-NULL value before the intialization (i.e. before
54     /// init() is called).
```

```
55      Mapping(State< In >* from=NULL, State< Out >* to=NULL);
56      /// Destructor
57      virtual ~Mapping();
58
59      /// Specify the input and output models.
60      virtual void setModels(State< In > * from, State< Out >* to);
61
62      /// Set the path to the objects mapped in the scene graph
63      void setPathInputObject(const std::string &o){m_inputObject.setValue(o);}
64      void setPathOutputObject(const std::string &o){m_outputObject.setValue(o);}
65
66      /// Return the pointer to the input model.
67      State< In >* getFromModel();
68      /// Return the pointer to the output model.
69      State< Out >* getToModel();
70
71      /// Return the pointer to the input model.
72      helper :: vector<BaseState*> getFrom();
73      /// Return the pointer to the output model.
```

Chapter 12

Modules

In this document we explain the usage and the functionalities of the modules developped using the Core Sofa Framework.

12.1 Collision Models

12.1.1 Ray Traced Collision Detection

This module implements the algorithm described in the paper entitled "Ray-traced collision detection for deformable bodies" by E.Hermann F.Faure and B.Raffin. When two objects are in collision, a ray is shot from each surface vertex in the direction of the inward normal. A collision is detected when the first intersection belongs to an inward surface triangle of another body. A contact force between the vertex and the matching point is then created. Experiments show that this approach is fast and more robust than traditional proximity-based collisions.

To speedup the searching of elements that cross the ray, we stored all the triangles of each colliding objects in an octree. Therefore we can easily navigate inside this octree and efficiently find the points crossing the ray. The octree structure allow us to have a satisfying performance independently from the size of the triangles used, which is not the case for a regular grid.

Using this module

An example showing the usage of the Ray Traced collision detection can be found in the RayTraceCollision.scn file in the *scene* directory. The collision detection mechanism must be set as **RayTraceDetection**, and instead of using a TriangleModel one must use a **TriangleOctreeModel**. The TriangleOctreeModel will create an Octree that contains all the Triangles from the collision model.

12.2 Forces

12.2.1 NonUniformHexahedronFEMForceFieldAndMass

Concepts

This force field implement the article :

```
@InProceedings{NPF06,
author       = "Nesme, Matthieu and Payan, Yohan and Faure, Fran\c{c}ois",
title        = "Animating Shapes at Arbitrary Resolution with Non-Uniform Stiffness",
booktitle    = "Eurographics Workshop in Virtual Reality Interaction and Physical Simulation (
month        = "nov",
year         = "2006",
organization = "Eurographics",
```

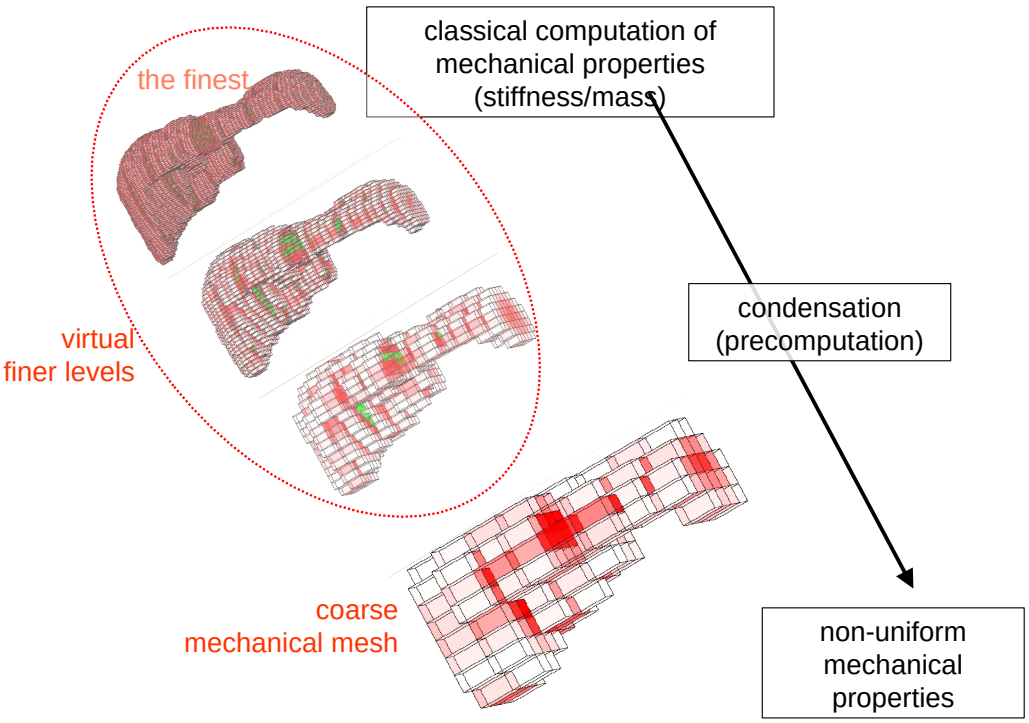


Figure 12.1: The condensation principle.

```
address      = "Madrid",
url          = "http://www-evasion.imag.fr/Publications/2006/NPF06"}
```

The basic idea, illustrated in figure 12.1, is :

- the use of finer virtual levels of SparseGrid
- the computation of classical mechanical matrices (mass and stiffness) at the finest resolution and the condensation of theses matrices to the current coarse mechanical resolution

Warning : actually the NonUniformHexahedronFEMForceFieldAndMass is only working with SparseGridTopology, and need enough finer virtual levels to compute the condensation.

Data Fields

- From HexahedronFEMForceFieldAndMass
  - method (char) : large/polar, the corotationnal method (default = large)
  - poissonRatio, youngModulus, density (float) : mechanical properties (density = volumetric mass in english $kg.m^{-3}$)
  - assembling (bool) : assembling the global system matrix ? (default = false)

- Specific to NonUniformHexahedronFEMForceFieldAndMass
  - nbVirtualFinerLevels (int) : how many finer virtual levels are employed in the condensation stage ? (default = 0)
- A hack on masses (for debugging)
  - useMass (bool) : are the condensated mass matrices are used ? (if not, scalar masses concentrated on particles are used) (default = 0)
  - totalMass (float) : if useMass=false, the scalar mass of the object

Example

```
<Node name="non uniform">
   <Object type="SparseGrid"
                   n="4 4 4"
                   filename="mesh/mymesh.obj"
                   nbVirtualFinerLevels="2"  />
   <Object type="MechanicalObject"/>
   <Object type="NonUniformHexahedronFEMForceFieldAndMass"
                   nbVirtualFinerLevels="2"
                   youngModulus="20000"
                   poissonRatio="0.3"
                   density="10" />
</Node>
```

Important : note that the SparseGrid has nbVirtualFinerLevels=2 in order to built enough finer virtual levels. This SparseGrid− >nbVirtualFinerLevels has to be greater or equal to the NonUniformHexahedronFEMForceF

A more complex example can be found in : examples/Components/forcefield/NonUniformHexahedronFEMForceFieldAndMass.scn where a comparison with a classical HexahedronFEMForceFieldAndMassForceField is done.

12.3 Soft Articulations

12.3.1 Concepts

The objective of this method is to use stiff forces to simulate joint articulations, instead of classical constraints.

To do this, a joint is modeled by a 6 degrees of freedom spring. By the way, the user specify a stiffness on each translation and rotation.

- A null stiffness defines a free movement.
- A huge stiffness defines a forbidden movement.
- All nuances are possible to define semi constrained movements.

2 main advantages can be extracted from this method :

- A better stability. As we don't try to statisfy constraints but only apply forces, there is always a solution to resolve the system.
- more possibilities to model articulations are allowed. As the stiffnesses define the degrees of freedom of the articulations, a better accuracy is posssible to simulate free movements as forbidden movements, i.e. an articulation axis is not inevitably totally free or totally fixed.

12.3.2 Realization

To define physically an articulated body, we first have a set of rigids (the bones). *cf fig. 1*

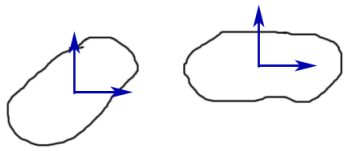


Figure 12.2: two bones

Each of these bones contains several articulations points, also defined by rigids to have orientation information. *cf fig. 2*

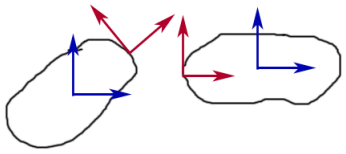


Figure 12.3: two bones (blue) with their articulation frames (red)

As seen previously, a joint between 2 bones is modeled by a 6-DOF spring. These springs are attached on the articulations points. *cf fig. 3*

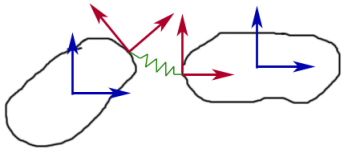


Figure 12.4: two bones linked by a joint-spring

12.3.3 Sofa implementation

To simulate these components in Sofa, we first need 2 mechanical objects : one for the bones (independent DOFs), and an other for the articulation points (mapped DOFs). Each of them contains a list of rigid DOFs (respectively all the bones and all the articulations of the articulated body). A mapping performs the link between the two lists, to know which articulations belong to which bones.

Corresponding scene graph

Example

The example `../examples/Components/forcefield/JointSpringForceField.scn` shows a basic pendulum :

```
<Node>
  <Object type="BruteForceDetection"/>
  <Object type="DefaultContactManager"/>
  <Object type="DefaultPipeline"/>
  <Object type="ProximityIntersection"/>

  <Node>
```

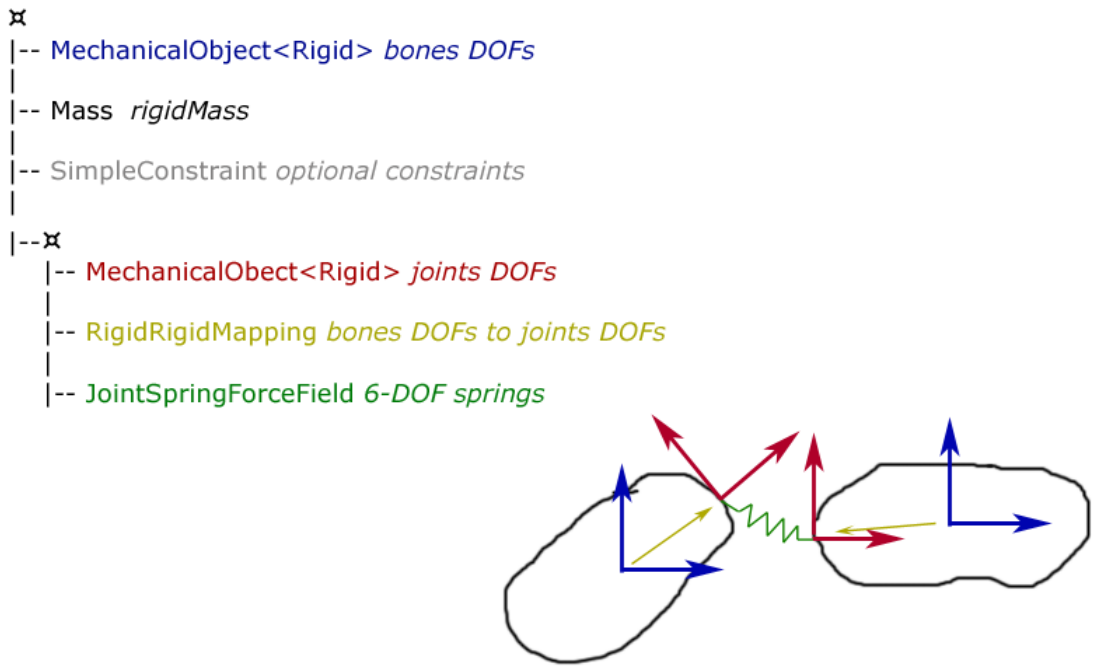


Figure 12.5: a simple articulated body scene

```
<Object type="CGImplicitSolver" />
<Object type="MechanicalObject" template="Rigid" name="bones DOFs"
        position="0 0 0  0 0 0 1
                  1 0 0  0 0 0 1
                  3 0 0  0 0 0 1
                  5 0 0  0 0 0 1
                  7 0 0  0 0 0 1" />
<Object type="UniformMass" template="Rigid" name="bones mass"
        mass="1 1 [1 0 0,0 1 0,0 0 1]" />
<Object type="FixedConstraint" template="Rigid" name="fixOrigin"
        indices="0" />

<Node>
  <Object type="MechanicalObject" template="Rigid" name="articulation points"
          position="0 0 0  0.707914 0 0 0.707914
                   -1 0 0  0.707914 0 0 0.707914
                    1 0 0  0.707914 0 0 0.707914
                   -1 0 0  0.707914 0 0 0.707914
                    1 0 0  0.707914 0 0 0.707914
                   -1 0 0  0.707914 0 0 0.707914
                    1 0 0  0.707914 0 0 0.707914
                   -1 0 0  0.707914 0 0 0.707914
                    1 0 0  0.707914 0 0 0.707914" />
  <Object type="RigidRigidMapping"
          repartition="1 2 2 2 2" />
  <Object type="JointSpringForceField" template="Rigid" name="joint springs"
          spring="BEGIN_SPRING 0 1  FREE_AXIS 0 0 0 0 1 0  ......  END_SPRING
                  BEGIN_SPRING 2 3  FREE_AXIS 0 0 0 0 1 0  ......  END_SPRING
                  BEGIN_SPRING 4 5  FREE_AXIS 0 0 0 0 1 0  ......  END_SPRING
```

```
                    BEGIN_SPRING 6 7  FREE_AXIS 0 0 0 0 1 0  ......  END_SPRING " />
    </Node>
    <Node>
      <Object type="MechanicalObject" template="Vec3d"
              position="-1 -0.5 -0.5  -1 0.5 -0.5 ..." />
      <Object type="MeshTopology"
              lines="0 1  1 2  ..."
              triangles="3 1 0  3 2 1  ..." />
      <Object type="TriangleModel"/>
      <Object type="LineModel"/>
      <Object type="RigidMapping"
              repartition="0 8 8 8 8" />
    </Node>
  </Node>
</Node>
```

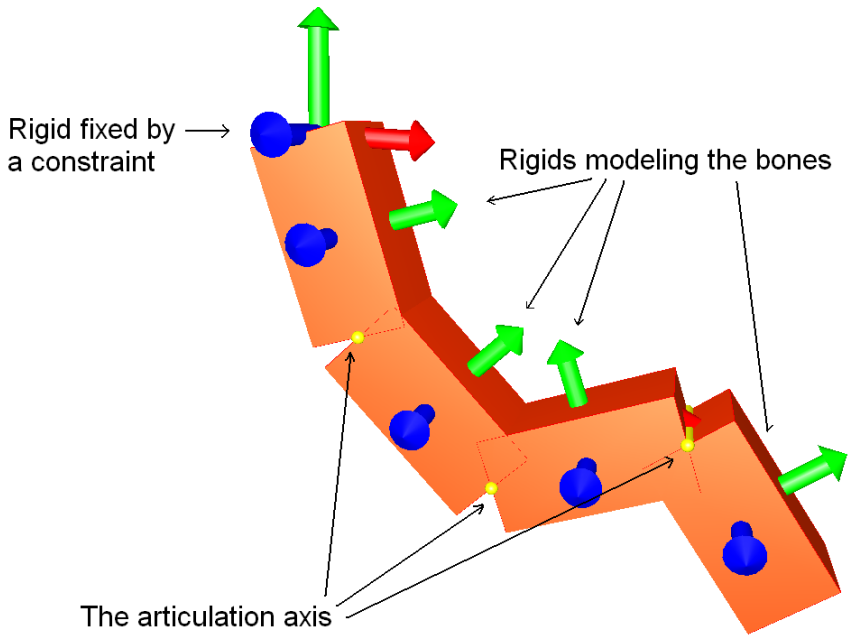


Figure 12.6: The pendulum is composed by 4 rigids linked one by one by articulations

In this example, we have under the first node the components to manage collisions, as usual. Under the second node, we have :

- the solver,
- the mechanical object modeling the independent rigid DOFs (5 rigids here),
- the rigid mass,
- a constraint, to fix the first rigid.

The third node (a child of the previous one) contains the components relative to the articulations :

- the mechanical object modeling articulation points. Positions and orientations are relative to their parents.

- the mapping to link the two mechanical objects, as explained before. To know which articulations belong to which bones, a repartition vector is used. Several cases for this vector are possible :
  - no value specified : every articulations belong to the first bone (classic rigid mapping).
  - one value specified (ex: repartition="2") : each bone has the same number of articulations.
  - number of bones values (like here, repartition="1 2 2 2 2") : the number of articulations is specified for each bone. For instance, here the first bone has 1 articulation, the next has 2 articulations, the next 2, Etc.
- the JointSpringForceField containing the springs (4 springs here). Each spring is defined by a list of parameters, separated by tag names. Each spring is defined between the tags BEGIN_SPRING and END_SPRING. For instance here we have : "BEGIN_SPRING 0 1 FREE_AXIS 0 0 0 0 1 0 KS_T 0.0 30000.0 KS_R 0.0 200000.0 KS_B 2000.0 KD 1.0 R_LIM_X -0.80 0.80 R_LIM_Y -1.57 1.57 R_LIM_Z 0.0 0.0 END_SPRING".
  - "0 1" are the indices of the two articulations the spring is attached to. They are the only compulsory parameters, the others are optional and take a default value if they are not specified.
  - "FREE_AXIS 0 0 0 0 1 0" design the free axis for the movements. it contains 6 booleans, one for each axis."0 0 0" mean that the 3 translation axis are constrained, and "0 1 0" mean that only the Y rotation axis is free.
  - "KS_T 0.0 30000.0" specify the stiffnesses to apply respectively for free translations and constrained translations.
  - "KS_R 0.0 200000.0" specify the stiffnesses to apply respectively for free rotations and constrained rotations.
  - "KS_B 2000.0" specify the stiffnesses to apply when an articulation is blocked, i.e. when a rotation exceeds the limit angle put on one axis.
  - "KD 1.0" is the damping factor
  - "R_LIM_X -0.80 0.80" design the limit angles (min and max) on the x axis.
  - "R_LIM_Y -1.57 1.57" design the limit angles (min and max) on the y axis.
  - "R_LIM_Z 0.0 0.0" design the limit angles (min and max) on the z axis.
  - It is also possible to specify "REST_T x y z" and "REST_R x y z t", which design the initial translation and rotation of the spring (in rest state).

The last node contains the collision model. Nothing special here.

12.3.4 Skinning

The articulated body described previously models the skeleton of an object. To have the external model (for the visual model or the collision model), which follows correctly the skeleton movements, it has to be mapped with the skeleton. A skinning mapping allows us to do this link. The external model is from this moment to deform itself smoothly, i.e. without breaking points around the articulations.

The influence of the bones on each point of the external model is given by skinning weights. 2 ways are possible to set the skinning weights to the mapping :

- Either the user gives directly the weights list to the mapping. It is useful if good weights have been pre computed previsouly, like in Maya for instance.
- Else, the user defines a number of references n that will be used for mapped points. Then, each external model point will search its n nearest bones (mechanical DOFs), and then compute the skinning weights from the relation :

$$W = \frac{1}{d^2}$$

with d : the distance between the external point and the rigid DOF.

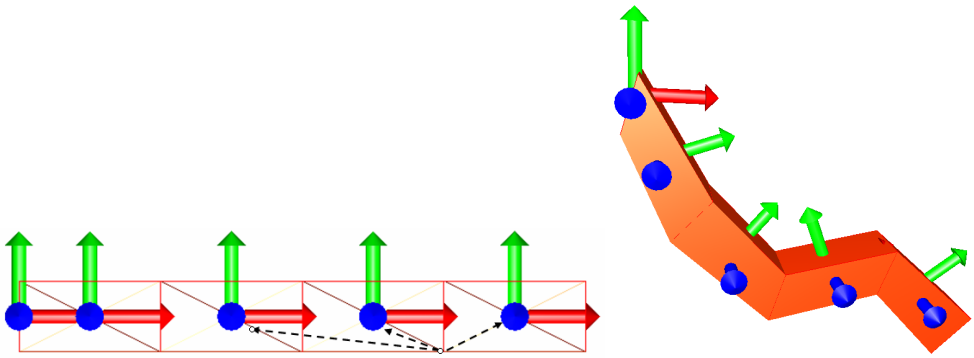


Figure 12.7: In the example `../examples/Components/mapping/SkinningMapping.scn` the external points compute their skinning weights from the 3 nearest DOFs

Figure 12.8: soft articulations coupled with skinning allow complexe model deformations

12.4 How to use mesh topologies in SOFA

H. Delingette, B. André

12.4.1 Introduction

While mesh geometry describes where mesh vertices are located in space, mesh topology tells how vertices are connected to each other by edges, triangles or any type of mesh element. Both bits of information are required on a computational mesh to perform :

- **Mesh Visualization**,
- **Collision detection** : some collision detection methods are mesh based (e.g. triangles or edges),
- **Mechanical Modeling** : deforming a mesh also requires to the knowledge of a mesh topology. For instance a spring mass model requires knowing about the edges that connects pair of vertices,
- **Haptic rendering**,
- **Description of scalar** (temperature, electric potential, etc.) or vectorial fields (speed, fiber orientation, etc.)

Since topological changes are essential for surgery simulators, a common difficulty when designing those simulators is to ensure that the visual, mechanical, haptic and collision behavior of all meshes stay valid and consistent upon any topological change.

Our approach to handle topological changes is modular since each software component (collision detection, mechanical solver...) may be written with little knowledge about the nature of other components. It is versatile because any type of topological changes can be handled with the proposed design.

Our objective to keep a modular design implies that mesh related information (such as mechanical or visual properties) is not centralized in the mesh data structure but is stored in the software components that are using this information. Furthermore, we manage an efficient and direct storage of information into arrays despite the renumbering of elements that occur during topological changes.

12.4.2 Family of Topologies

We focus the topology description on 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. 12.9) 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 derived types. For instance a spring-mass mechanical component only requires the knowledge of a list of edges (an *Edge Set Topology* as described in Fig. 12.9) to be effective. With the proposed design, this component can be used with no changes on triangulation or hexahedral meshes.

The proposed hierarchy makes also a distinction between conformal and manifold meshes. While most common FEM components require a mesh to be conformal (but not necessarily manifold), many high-level software components (such as cutting, contact, haptic feedback algorithms) require the mesh to be a manifold where a surface normal is well-defined at each vertex.

Topology objects are composed of four functional members: *Container*, *Modifier*, *Algorithms* and *Geometry*.

- The *Container* member creates and updates when needed two complementary arrays (see Fig. 12.10). The former describes the l-cells included in a single k-cell, $l < k$, while the latter gives the k-cells adjacent to a single l-cell.
- The *Modifier* member provides low-level methods that implement elementary topological changes such as the removal or addition of an element.

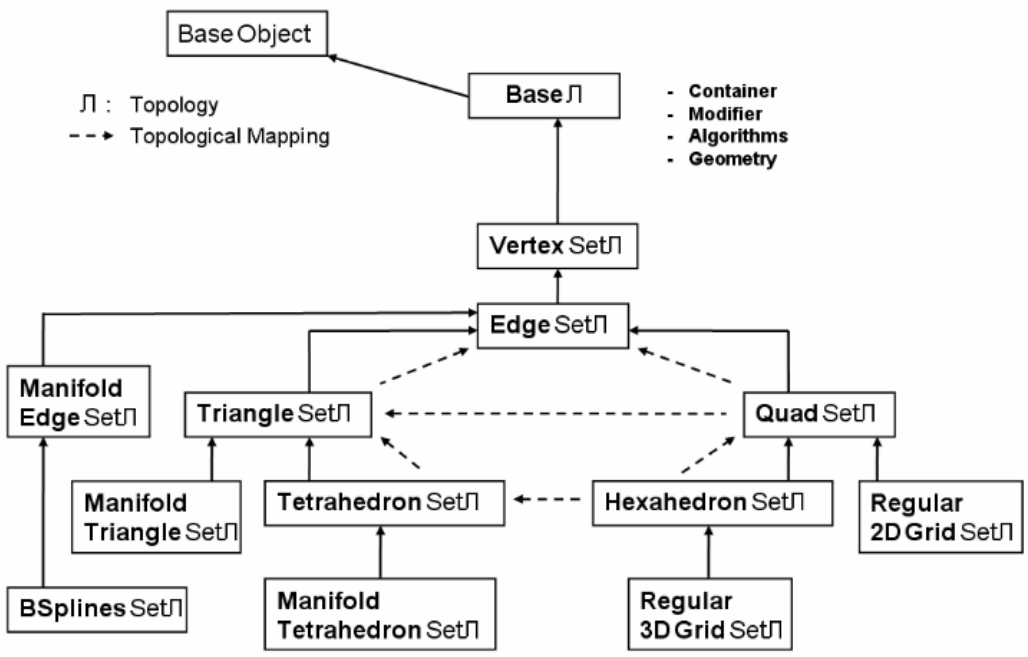


Figure 12.9: Family tree of topology objects. Dashed arrows indicate possible *Topological Mappings* from a topology object to another.

- The *Algorithms* member provides high-level topological modification methods (cutting, refinement) which decompose complex tasks into low-level ones).
- The *Geometry* member provides geometrical information about the mesh (*e.g.* length, normal, curvature, ...) and requires the knowledge of the vertex positions stored in the *Degrees of Freedom* component.

12.4.3 Component-Related Data Structure

A key feature of our design is that containers storing mesh information (material stiffness, list of fixed vertices, nodal masses, ...) are stored in *components* and spread out in the simulation tree. This modular approach is in sharp contrast with a centralized storage of information in the mesh data structure through the use of generic pointers or template classes.

Another choice is that most containers are simple arrays with contiguous memory storage and a 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. For instance, when a single element is removed, the last array element is renumbered such that the array stays contiguous. Fortunately, all renumbering tasks that maintain consistent arrays can be automated and hidden to the user when topological changes in the mesh arise. Besides, time to update data structures does not depends on the total number of mesh elements but only on the number of modified elements. Therefore, in our framework, mesh data structures are stored in simple and efficient containers, the complexity of keeping the container consistent with topological changes being automated.

There are as many containers as topological elements: vertices, edges, triangles, ... 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. Finally, two other types of containers may be used when needed. The former stores a data structure for a subset of topological elements (for instance pressure on surface triangles in a tetrahedralisation) while the latter stores only a subset of element indices.

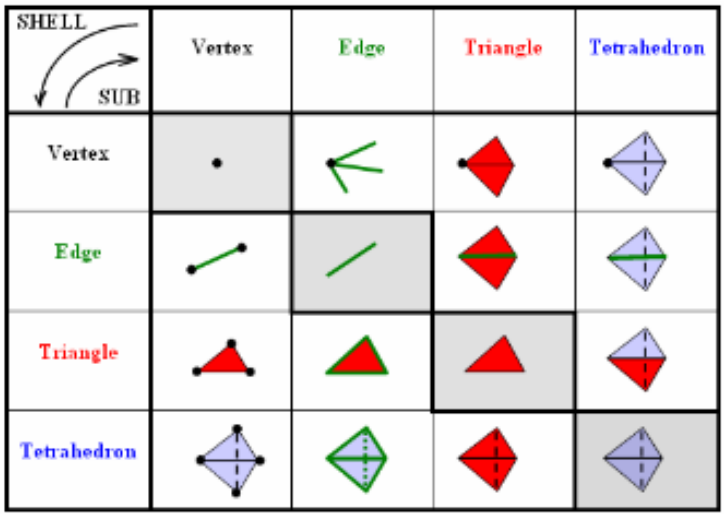


Figure 12.10: The two topological arrays stored in a *Container* correspond to the upper and lower triangular entries of this table. The upper entries provide the k-cells adjacent to a l-cell, $l < k$. The lower entries describe the l-cells included in a k-cell. Similar table exists for quad and hexahedron elements.

12.4.4 Handling Topological Changes

Surgery simulation involves complex topological changes on meshes, for example when cutting a surface along a line segment, or when locally refining a volume before removing some tissue. However, one can always decompose these complex changes into a sequence of elementary operations, such as adding an element, removing an element, renumbering a list of elements or modifying a vertex position.

Our approach to handle topological changes makes the update of data structures transparent to the user, through a mechanism of propagation of topological events. A topological event corresponds to the intent to add or to remove a list of topological elements. But the removal of elements cannot be tackled in the same way as the addition of elements. Indeed, the element removal event must be first notified to the other components before the element is actually removed by the *Modifier*. Conversely, element addition is first processed by the *Modifier* and then element addition event is notified to other components (see Fig. 12.12). Besides, the events notifying the creation of elements also include a list of ancestor elements. Therefore, when splitting one triangle into two sub-triangles (see Fig. 12.13), each component-related information (*e.g.* its Young modulus or its mass density) associated with a sub-triangle will be created knowing that the sub-triangle originates from a specific triangle. Such mechanism is important to deal with meshes with non-homogeneous characteristics related to the presence of pathologies.

The mechanism to handle topological changes is illustrated by Fig. 12.11. The notification step consists in accumulating the sequence of topological events involved in a high-level topological change into a buffer stored in the *Container*. Then the event list is propagated to all its neighbors and leaves beneath by using a visitor mechanism, called a *Topology Visitor*. Once a given component is visited, the topological events are actually processed one by one and the data structure used to store mesh related information are automatically updated.

In practice, for each specific component (*e.g.* spring-mass mechanical component), a set of callback functions are provided describing how to update the data structure (*e.g.* spring stiffness and damping values) when adding or removing an element (*e.g.* the edges defined by the two extremities of the springs). We applied the observer design pattern so that component-related data structures update themselves automatically.

12.4.5 Combining Topologies

Handling a single mesh topology in a surgery simulation scene is often too restrictive. There are at least three common situations where it is necessary to have, for the same mesh, several topological

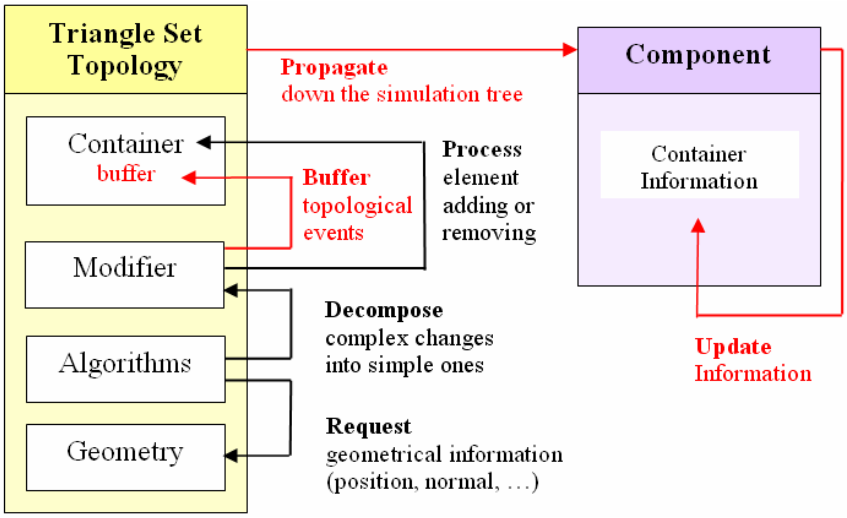


Figure 12.11: Handling topological changes, with the example of a *Triangle Set Topology*. *Component* corresponds to any component of the simulation which may need topological information to perform a specific task. Black features indicate the effective change process. Red features show the steps of event notification.

Adding a list of items	Removing a list of items
1. ADD	1. WARN
2. WARN	2. PROPAGATE
3. PROPAGATE	3. REMOVE

Figure 12.12: Order to respect when adding or removing an item. WARN means : add the current topological change (add or delete a list of items) in the list of TopologyChanges. PROPAGATE means : traverse the simulation tree with a TopologyChangeVistor to send the current topological change event to all force fields, constraints, mappings, etc.

descriptions sharing the same degrees of freedom: the *boundary*, *composite* and *surrogate* cases. In the *boundary* scenario, specific algorithms may be applied to the boundary of a mesh, the boundary of tetrahedral mesh being a triangulated mesh and that of a triangular mesh being a polygonal line. For instance, those algorithms may consist of applying additional membrane forces (*e.g.* to simulate the effect of the Glisson capsule in the liver) or visualizing a textured surface. Rather than designing specific simulation components to handle triangulations as the border of tetrahedrisations, our framework allows us to create a triangulation topology object from a tetrahedrisation mesh and to use regular components associated with triangulations.

The *composite* scenario consists in having a mesh that includes several types of elements: triangles with quads, or hexahedra with tetrahedra. Instead of designing specific components for those composite meshes, it is simpler and more versatile to reuse components that are dedicated to each mesh type. Finally the *surrogate* scenario corresponds to cases where one topological element may be replaced by a set of elements of a different type with the same degrees of freedom. For instance a quad may be split into two triangles while an hexahedron may be split into several tetrahedra. Thus a quad mesh may also be viewed as a triangular mesh whose topology is constrained by the quad mesh topology.

These three cases can be handled seamlessly by using a graph of multiple topologies, the topology

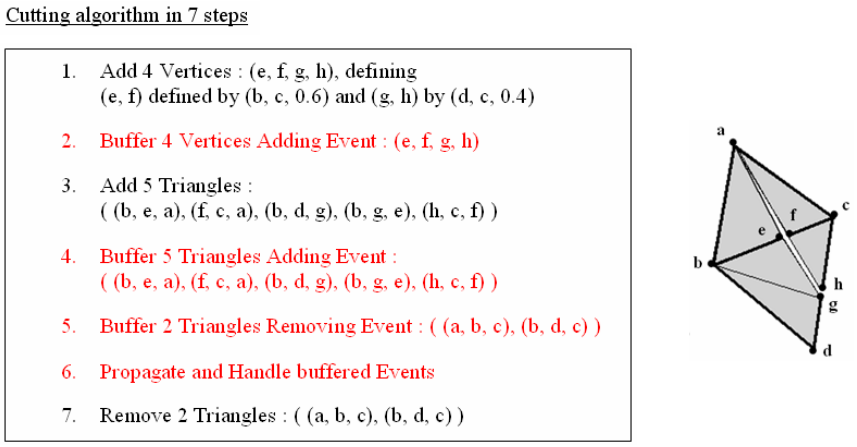


Figure 12.13: Seven steps to perform cutting along two triangles (these generic steps would be the same to cut an arbitrarily number of triangles). Black steps indicate the effective change process. Red steps show the steps of event notification.

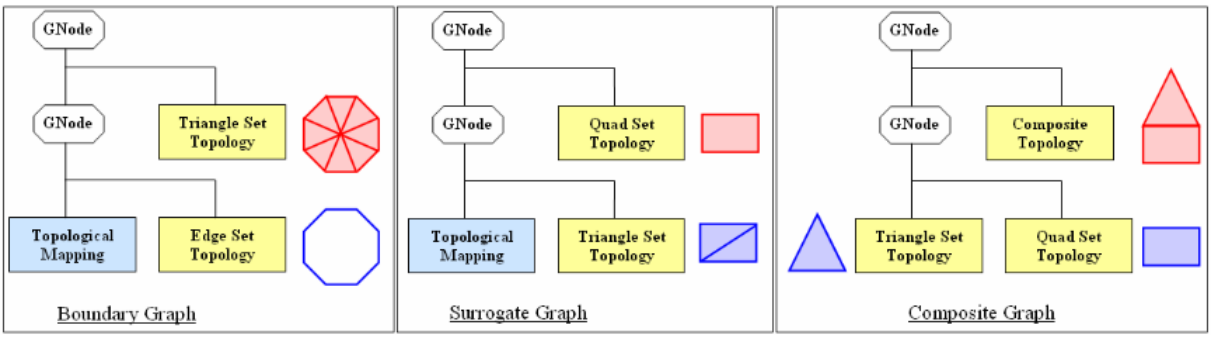


Figure 12.14: Three scenari examples to combine topologies. (*From left to right*) A *Boundary Graph* from a triangle set to an edge set, a *Surrogate Graph* from a quad set to an triangle set and a *Composite Graph* superseding a quad set and a triangle set.

object being at the top node having the specific role of controlling the topologies below. Although we chose to duplicate topological information in memory, it has no effect on the time required to compute the forces. Fig. 12.15 provides an example of a border scenario (triangulation as the border of a tetrahedralisation) while Fig. 12.14 shows general layout of topology graphs in the three cases described previously. In any cases, those graphs include a dedicated component called a *Topological Mapping* whose objectives are twofold. First, they translate topological events originating from the master topology (*e.g.* remove this quad) into topological actions suitable for the slave topology (*e.g* remove those two triangles for a surrogate scenario). Second, they provide index equivalence between global numbering of elements in the master topology (*e.g* a triangle index in a tetrahedralisation topology) and local numbering in the slave topology (*e.g* the index of the same triangle in the border triangulation). Possible *Topological Mappings* from a topology object to another have been represented by blue arrows in Fig. 12.9.

Note that those topology graphs can be combined and cascaded, for instance by constructing the triangulation border of a tetrahedrisation created from an hexahedral mesh. But only topology algorithms of the master topology may be called to simulate cutting or to locally refine a volume. By combining topology graphs with generic components one can simulate fairly complex simulation scenes where topological changes can be seamlessly applied.

12.4.6 An example of Topological Mapping : from TetrahedronSetTopology to TriangleSetTopology

A TopologicalMapping is a new kind of Mapping which converts an input topology to an output topology (both topologies are of type BaseTopology).

It first initializes the mesh of the output topology from the mesh of the input topology, and it creates the two Index Maps that maintain the correspondence between the indices of their common elements.

Then, at each propagation of topological changes, it translates the topological change events that are propagated from the input topology into specific actions that call element adding methods or element removal methods on the output topology, and it updates the Index Maps.

So, at each time step, the geometrical and adjacency information are consistent in both topologies.

Here is the scene-graph corresponding to the simulation of an object which can be represented as a tetrahedral volume (on which one volume force is applied) or as a triangular surface (on which two surface forces are applied). Note that the Visual Model and the Collsion Model are attached to the surface mesh of the object :

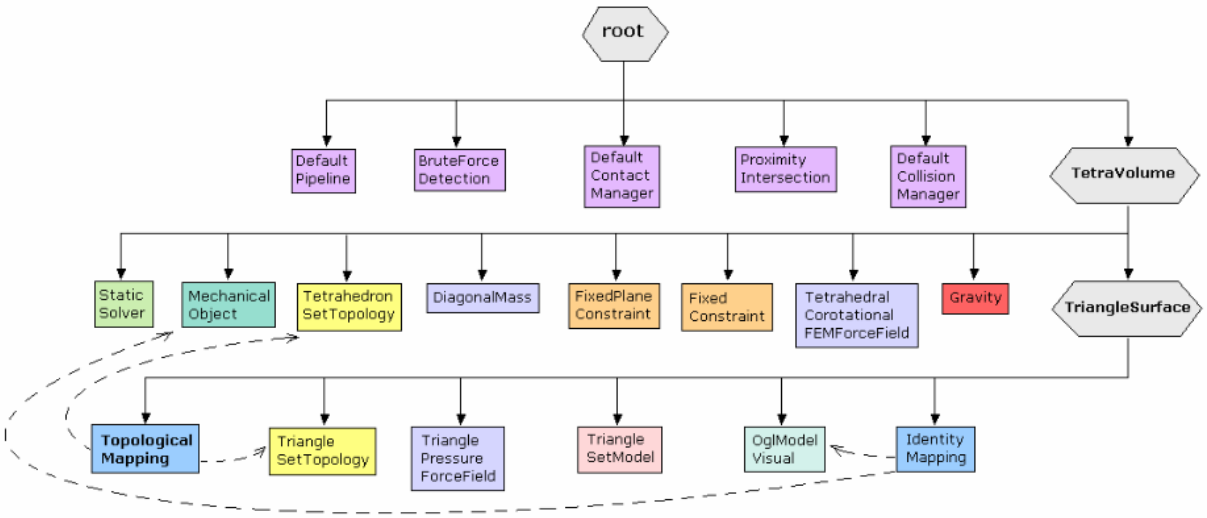


Figure 12.15: Scene Graph illustrating a TopologicalMapping from a TetrahedronSetTopology to a TriangleSetTopology.

Let us consider an example where the user wants to remove one tetraheron (whose one triangle at least is visible) from the tetrahedral volume. The component Tetra2TriangleSetTopology handles this topological change by following five steps :

- **Step 1.** The user right-clicks on visible triangle T in the scene, which is detected by the Collision Model and indexed by loc_T in the triangular surface mesh.
- **Step 2.** If a Topological Mapping of type (input = TetrahedronSetTopology, output = TriangleSetTopology) does exist, the index map $Loc2GlobVec$ is requested to give the index $glob\_T$ which is indexing the triangle T in the tetrahedral volume mesh. The TetrahedraAroundTriangle gives then the index ind_TE which is indexing the unique tetrahedron TE containing T. We call the action RemoveTetrahedra($< ind\_TE >$) on the input topology.
- **Step 3.** The TetrahedronSetTopology notifies all the removal events, that are successively concerning one tetrahedron, one or more isolated triangles, the possibly isolated edges and the possibly isolated points.

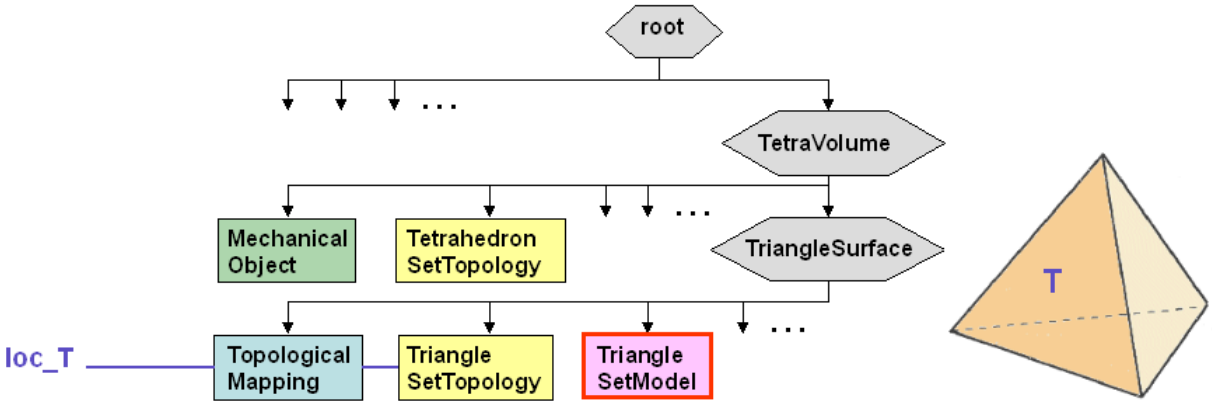


Figure 12.16: Scenario when the user wants to remove one tetraheron.- Step 1.

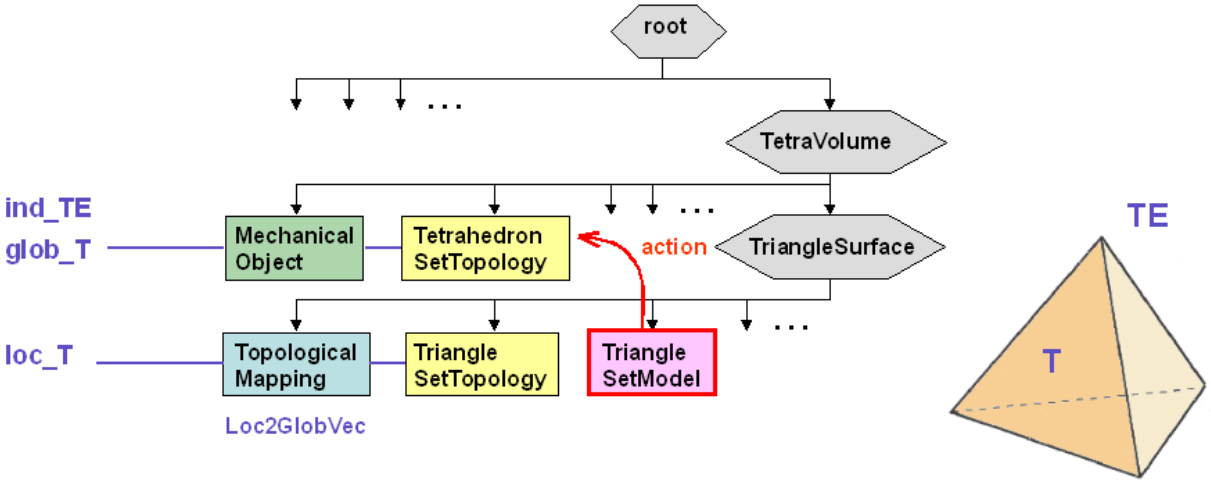


Figure 12.17: Scenario when the user wants to remove one tetraheron.- Step 2.

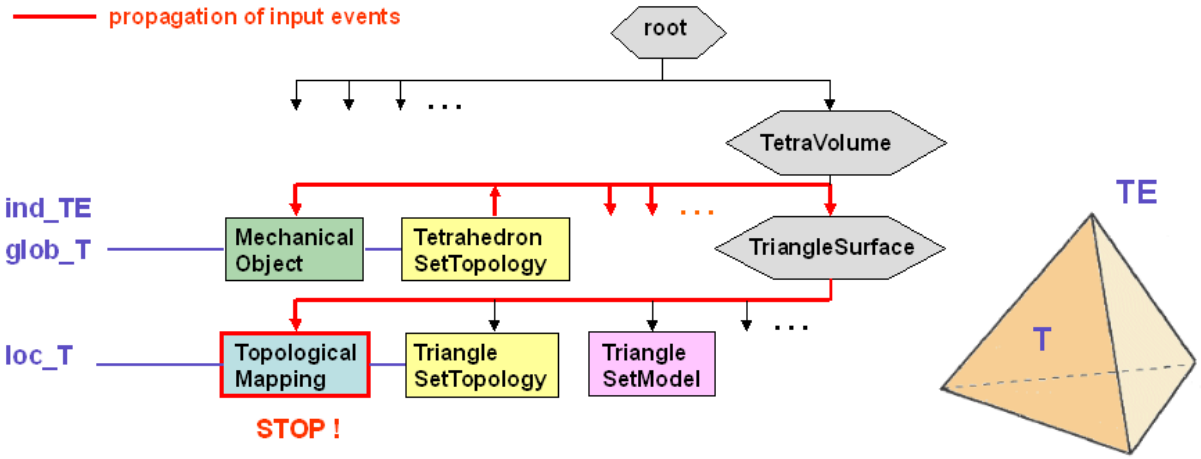


Figure 12.18: Scenario when the user wants to remove one tetraheron.- Step 3.

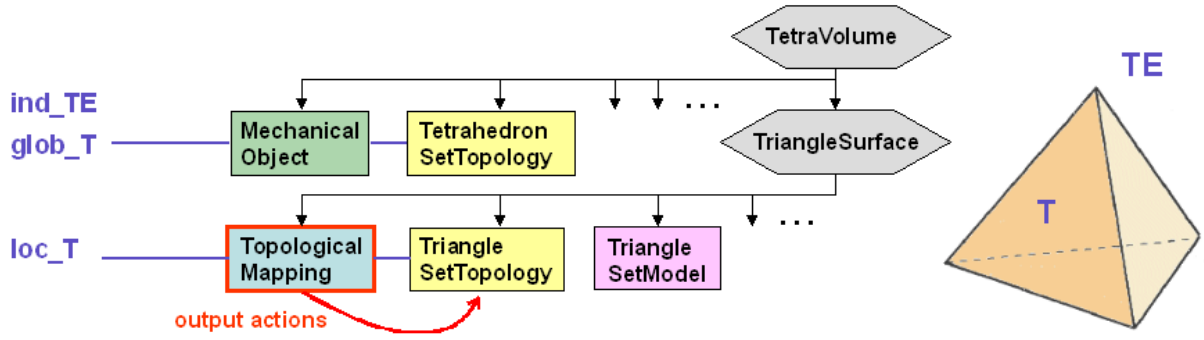


Figure 12.19: Scenario when the user wants to remove one tetraheron.- Step 4.

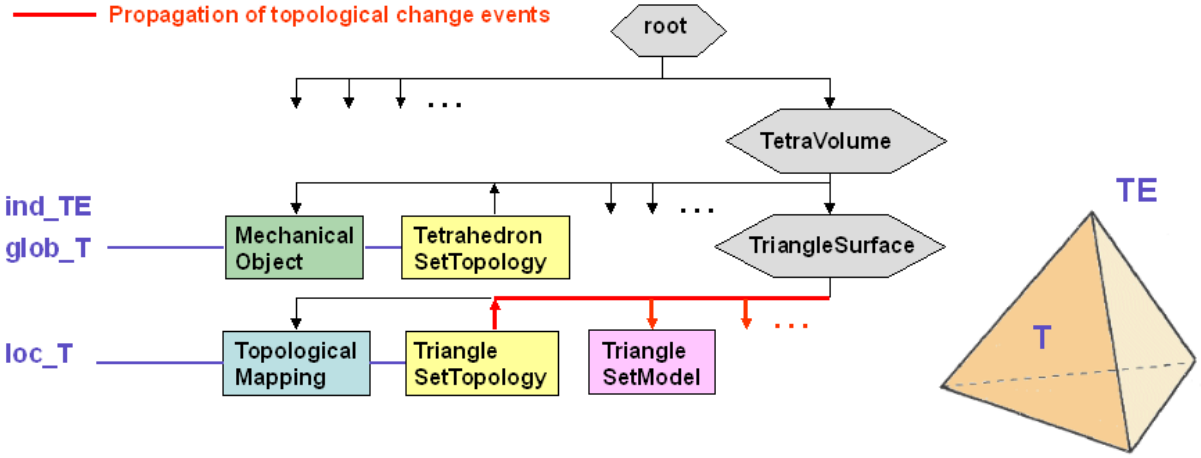


Figure 12.20: Scenario when the user wants to remove one tetraheron.- Step 5.

- **Step** 4. The propagation of topological events reaches a Topological Mapping which is strictly lower in the scene graph, then it stops. The Tetra2TriangleTopologicalMapping translates the input events into output actions on the TriangleSetTopology. The event Tetrahedron removal is translated into the action AddTriangles (< indices of new visible triangles >). The event Triangle removal is translated into the action RemoveTriangles (< indices of destroyed triangles >, removeDOF = false), where (removeDOF = false) indicates that the DOFs of the isolated points must not be deleted because they have already been removed by the input topology. The Index maps (Loc2GlobVec, Glob2LocMap, In2OutMap) are requested and updated to maintain the correspondence between the items indices in input and output topologies.
- **Step** 5. The adding events concerning one or more new visible triangles and the removal events concerning one or more isolated triangles are notified from the TriangleSetTopology.

12.4.7 Example of scene file with a topological mapping

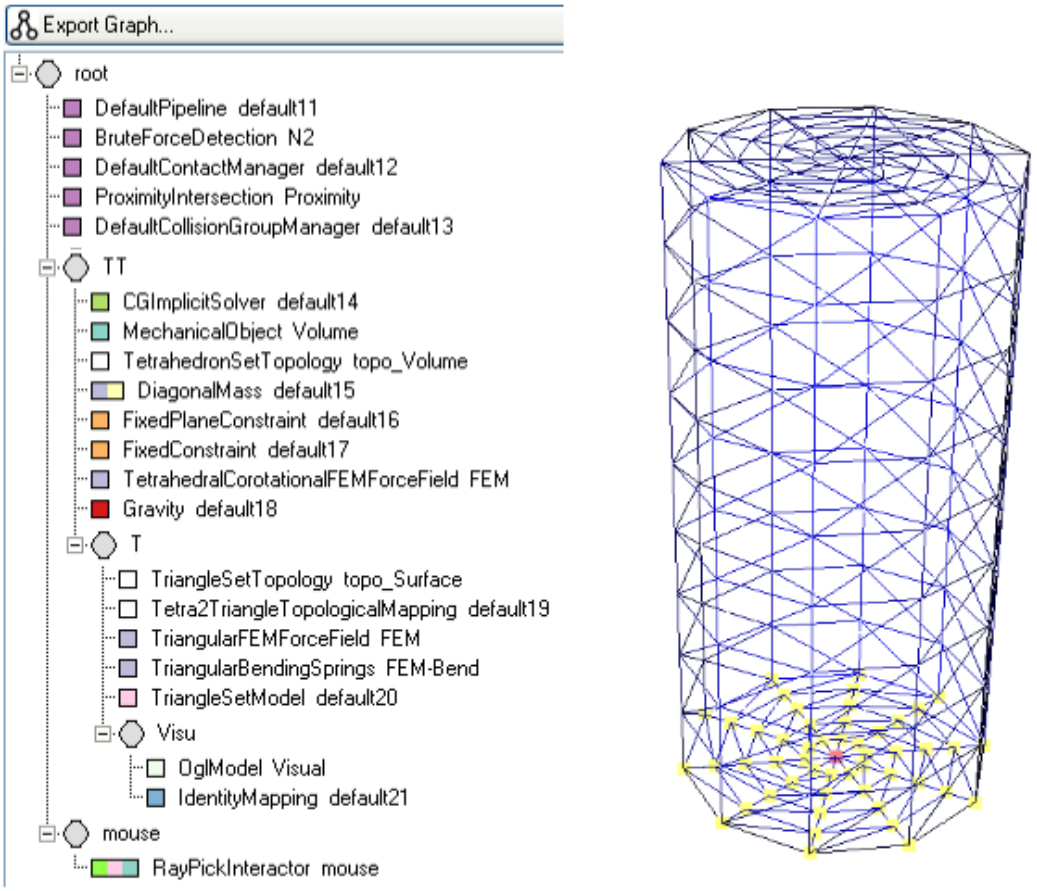


Figure 12.21: Scene Graph simulating a bending cylinder as a tetrahedral volume and as a triangular surface (the cylinder membrane)

Here is the scene file corresponding to the example :

```
<Node name="root" dt="0.05" showBehaviorModels="1" showCollisionModels="0" showMappings="0"
 showForceFields="0" showBoundingTree="0" gravity="0 0 0">

<Object type="CollisionPipeline" verbose="0" />
<Object type="BruteForceDetection" name="N2" />
<Object type="CollisionResponse" response="default" />
<Object type="MinProximityIntersection" name="Proximity" alarmDistance="0.8"
 contactDistance="0.5" />

<Object type="CollisionGroup" />

<Node name="TT">

<Object type="EulerImplicit" name="cg_odesolver" printLog="false"/>
<Object type="CGLinearSolver" iterations="25" name="linear solver" tolerance="1.0e-9"
 threshold="1.0e-9" />

<Object type="MeshLoader" name="meshLoader" filename="mesh/cylinder.msh" />
<Object type="MechanicalObject" name="Volume" />
<include href="Objects/TetrahedronSetTopology.xml" />

    <Object type="DiagonalMass" massDensity="0.5" />
<Object type="FixedPlaneConstraint" direction="0 0 1" dmin="-0.1" dmax="0.1"/>
<Object type="FixedConstraint" indices="0" />
    <Object type="TetrahedralCorotationalFEMForceField" name="FEM" youngModulus="60"
     poissonRatio="0.3" method="large" />

    <Object type="Gravity" gravity="0 0 0"/>

<Node name="T">

<include href="Objects/TriangleSetTopology.xml" />
<Object type="Tetra2TriangleTopologicalMapping" object1="../../Container" object2="Container"/>

      <Object type="TriangularFEMForceField" name="FEM" youngModulus="10" poissonRatio="0.3"
       method="large" />

<Object type="TriangularBendingSprings" name="FEM-Bend" stiffness="300" damping="1.0"/>

<Object type="TriangleSet"/>

<Node name="Visu">

<Object type="OglModel" name="Visual" color="blue" />
<Object type="IdentityMapping" object1="../../../Volume" object2="Visual" />

</Node>

</Node>

</Node>
```

```
</Node>
```

Here is the example of the included TriangleSetTopology.xml file, where the four members of the *TriangleSetTopology* are defined :

```
<Node name="Group">
  <Object type="TriangleSetTopologyContainer"  name="Container" />
  <Object type="TriangleSetTopologyModifier"   name="Modifier" />
  <Object type="TriangleSetTopologyAlgorithms" name="TopoAlgo"   template="Vec3d" />
  <Object type="TriangleSetGeometryAlgorithms" name="GeomAlgo"   template="Vec3d" />
</Node>
```

12.4.8 How to make a component aware of topological changes ?

There are actually a few generic lines of code to add in a component for it to handle topological changes.

If the component is based on topological elements like edges, the main idea is to introduce an object *edgeinfo* of type *EdgeData* and to template it by a data structure *EdgeInformation* that is attached to each edge (this data structure has to be defined by the component, to which it is specific). By calling the method *handleTopologyEvents* on the object *edgeinfo* (of type *EdgeData <EdgeInformation>*), the component-related data structure is automatically updated (code has been implemented in file *EdgeData.inl*).

Here : *edgeinfo*[i] is the data structure attached to the edge indexed by i in the topological component *EdgeSetTopology*.

In SOFA, among the existing ForceField component able to handle topological changes there are for example : *BeamFEMForceField* (for the simple case) and *TriangularBendingSprings* (for the complicated case : data structure attached to each edge need to contain indices of points, which must also be updated by the method *handleTopologyChange*).

For the simple case, here is how to adapt a Force Field component called *totoForceField* (for instance based on elements of type edge) to topological changes :

Modifications in header file *totoForceField.h* :

- Include the following header :

  ```
  #include <sofa/component/topology/EdgeData.h>
  ```

- Define a structure *EdgeInformation* which describes the information attached to each edge (for example *spring stiffness*, *spring length*, ...)

- Add an object *edgeinfo* of type *EdgeData* templated by the type *EdgeInformation* :

  ```
  topology::EdgeData<EdgeInformation> edgeinfo;
  ```

- Declare the virtual method *handleTopologyChange* :

  ```
  virtual void handleTopologyChange();
  ```

- Add the callback function for the creation of edges :

```
static void EdgeCreationFunction(
    int edgeIndex,
    void* param,
    EdgeInformation &ei,
    const topology::Edge& ,
    const sofa::helper::vector< unsigned int > &,
    const sofa::helper::vector< double >&
);
```

Modifications in inline file *totoForceField.inl* :

- Add at the end of the method *init()* to specify the callbacks :

```
edgeinfo.setCreateFunction(EdgeCreationFunction);
edgeinfo.setCreateParameter( (void *) this );
edgeinfo.setDestroyParameter( (void *) this );
```

- Implement the method *handleTopologyChange()* :

```
template <class DataTypes>
void totoForceField<DataTypes>::handleTopologyChange()
{

    std::list< const sofa::core::componentmodel::topology::TopologyChange* >
::const_iterator itBegin = _topology->firstChange();

    std::list< const sofa::core::componentmodel::topology::TopologyChange*>
::const_iterator itEnd = _topology->lastChange();

    edgeinfo.handleTopologyEvents(itBegin,itEnd);

}
```

- Implement the method *EdgeCreationFunction* (usefull for the initialization of the data attacheed to a new edge) :

```
template<class DataTypes>
void totoForceField<DataTypes>::EdgeCreationFunction(

int edgeIndex, void* param, EdgeInformation &ei,
const topology::Edge& e,  const sofa::helper::vector< unsigned int > &a,
const sofa::helper::vector< double >&
)

{

    totoForceField<DataTypes> *ff= (totoForceField<DataTypes> *)param;
```

```
        if (ff) {

          ei.champ = value; // initialiser tous les champs de "edgeinfo[i]"

          (...)

        }
    }
```

At any time, it is possible to access the information from the container member of the topology (so as to get some neighborhood information) by using a pointer to the *BaseMeshTopology* API :

```
sofa::core::componentmodel::topology::BaseMeshTopology* _topology;
_topology = getContext()->getMeshTopology();
```

12.4.9 What happens when I split an Edge ?

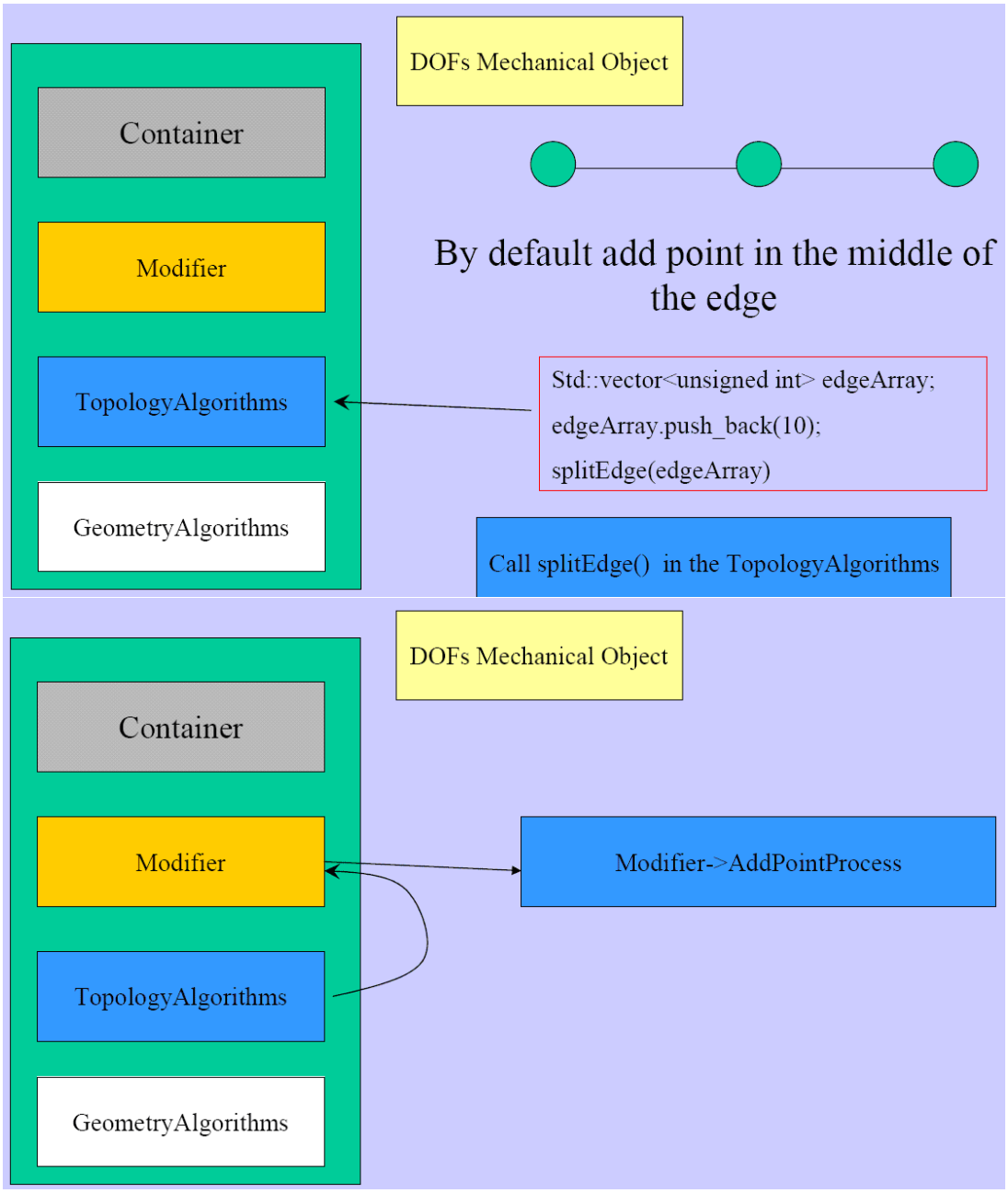


Figure 12.22: What happens when I split an Edge ? - Step 1. 2.

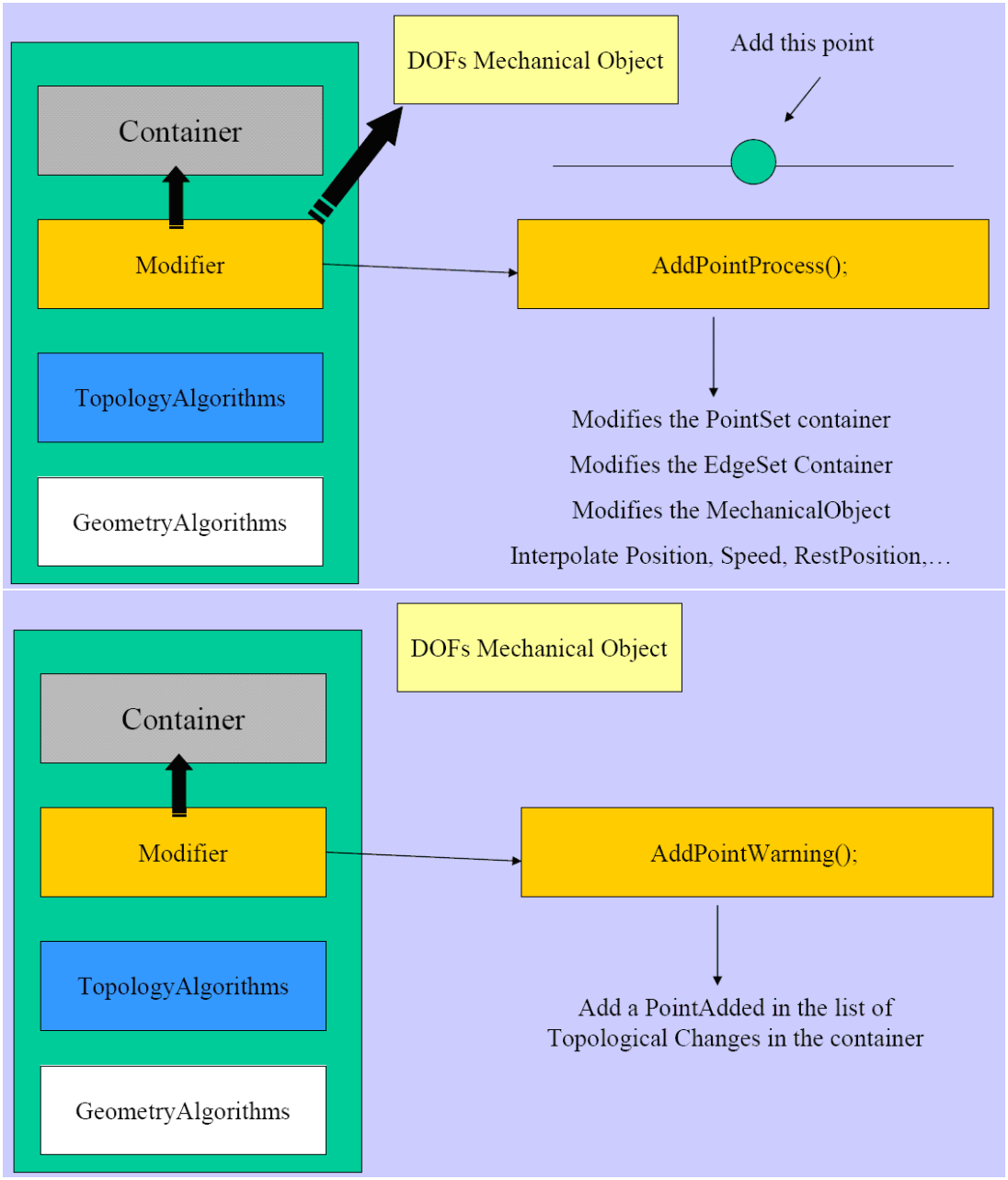


Figure 12.23: What happens when I split an Edge ? - Step 3. 4.

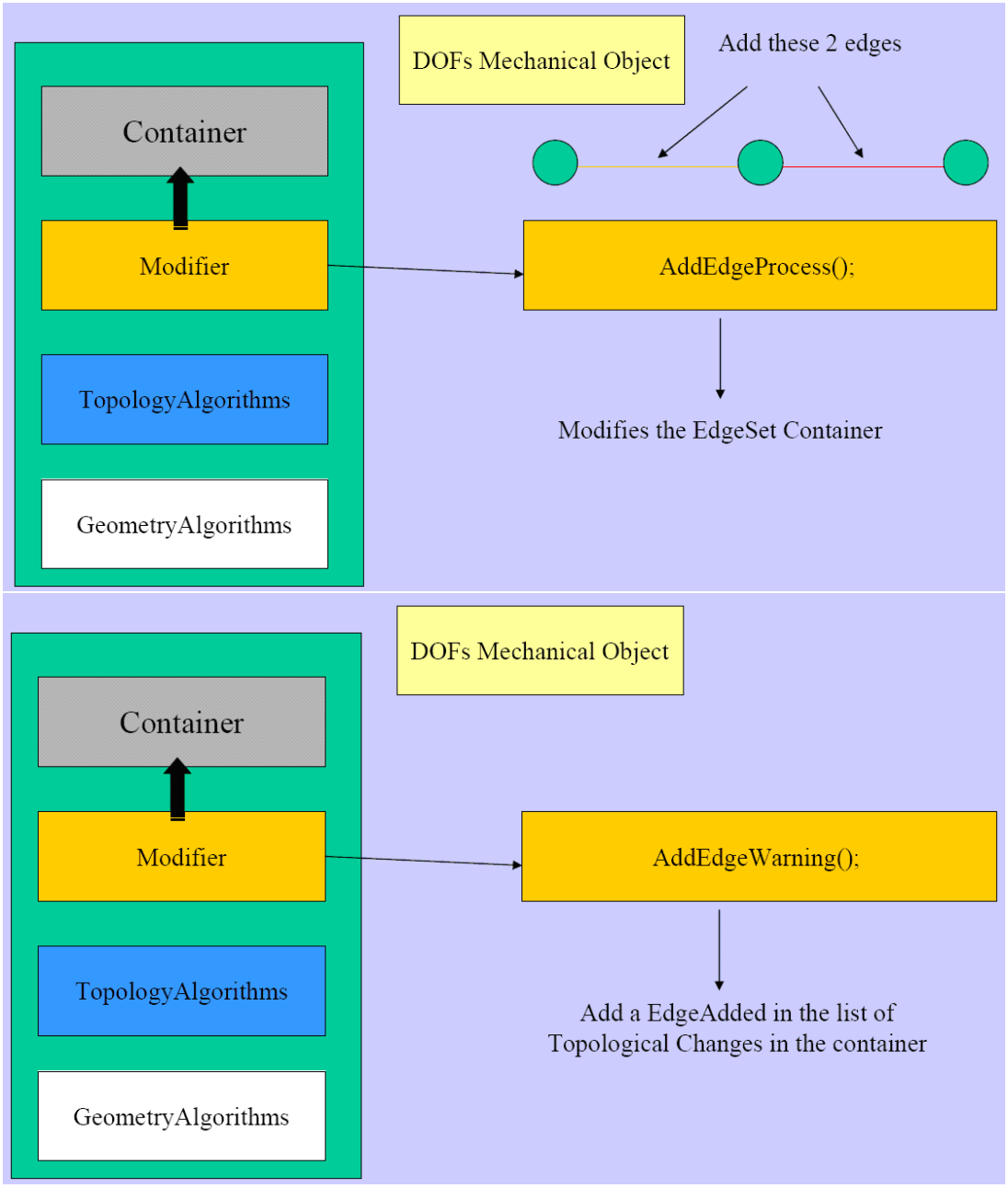


Figure 12.24: What happens when I split an Edge ? - Step 5. 6.

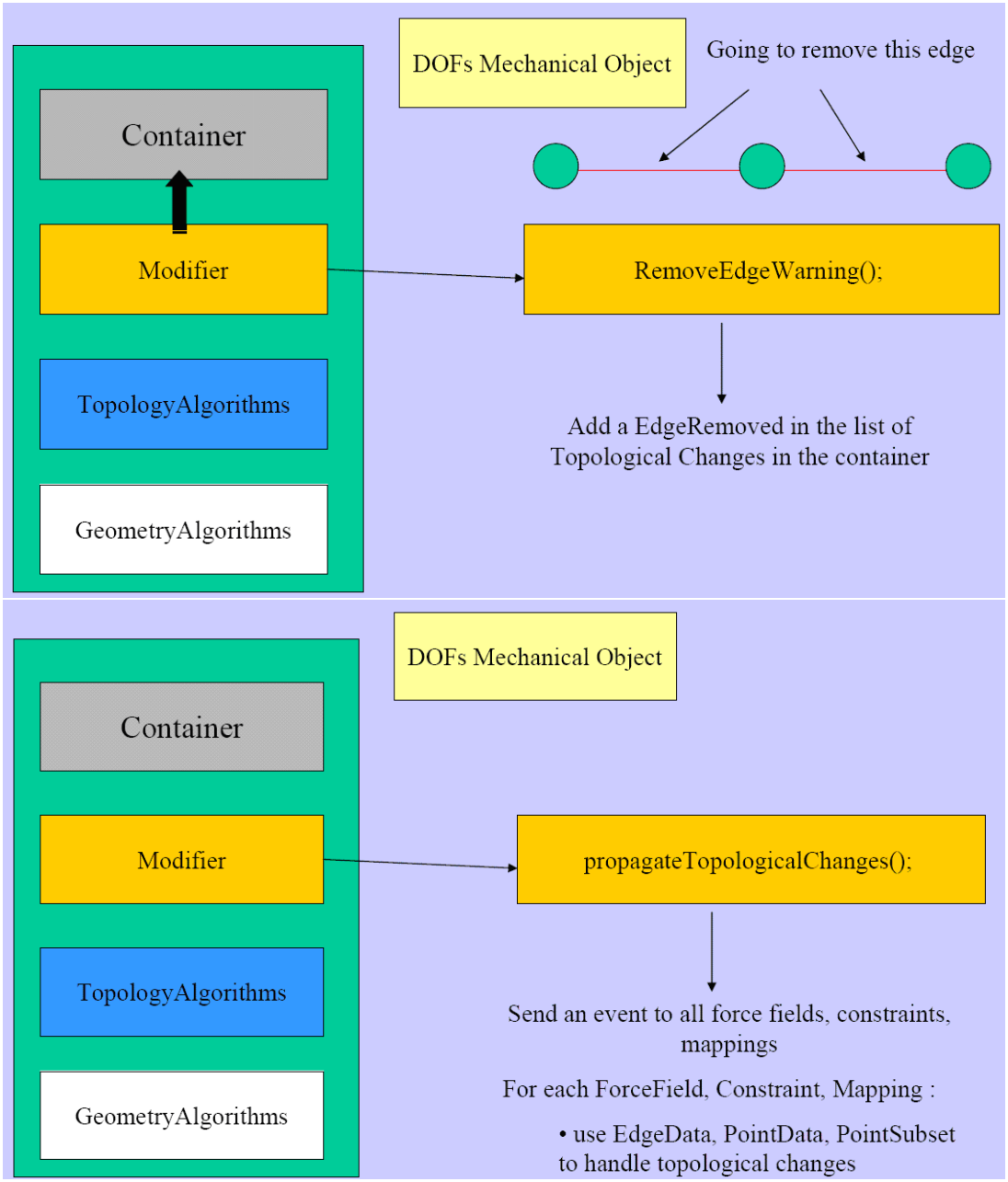


Figure 12.25: What happens when I split an Edge ? - Step 7. 8.

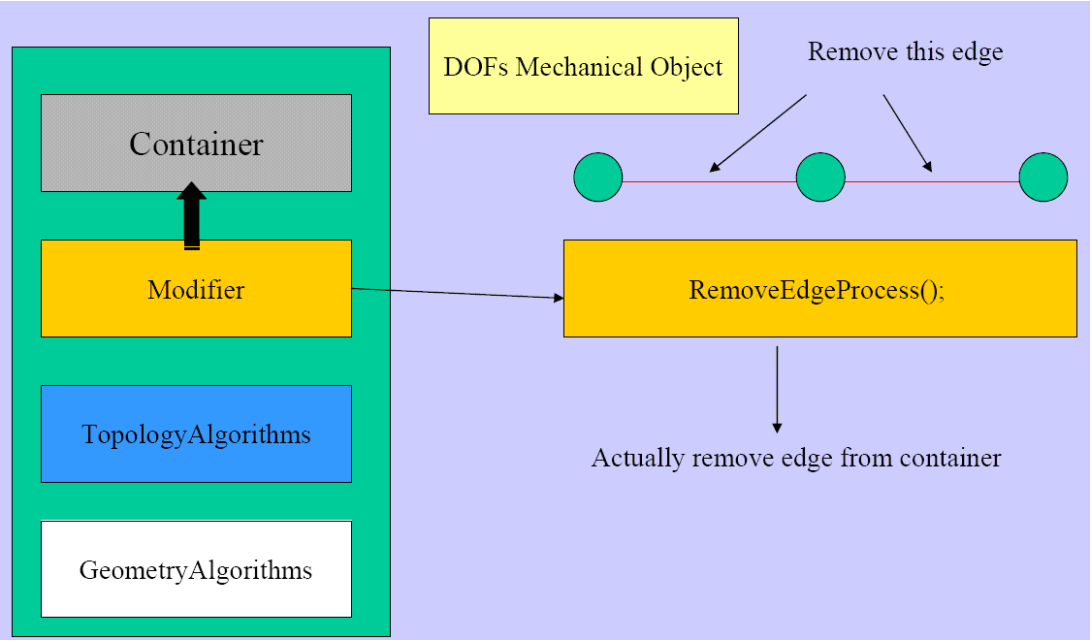


Figure 12.26: What happens when I split an Edge ? - Step 9.

12.5 Graphic User Interface

12.5.1 First steps

Once SOFA is compiled, in the directory bin will be placed an executable called runSofa (or runSofad if you are using the debug version). The first time you will run SOFA, a GUI using Qt will appear. By default a simulation modeling a liver with some fixed points will be displayed. Simulations must be written in a xml format, generally, Sofa scenes have the extension ".scn", and Sofa objects directly the extension ".xml". You can load both of them using the file menu, or drag & dropping them in the interface.
Basically the GUI is divided in two:

- a control panel subdivided in several tabulations, giving the user the possibility to display various information about the simulation, and even modifying it interactively
- a viewer: by default, you will be using our hand-made viewer, using OpenGL. Others are available, and below, we describe how to create your own viewer, if you desire to insert a more powerful rendering engine.

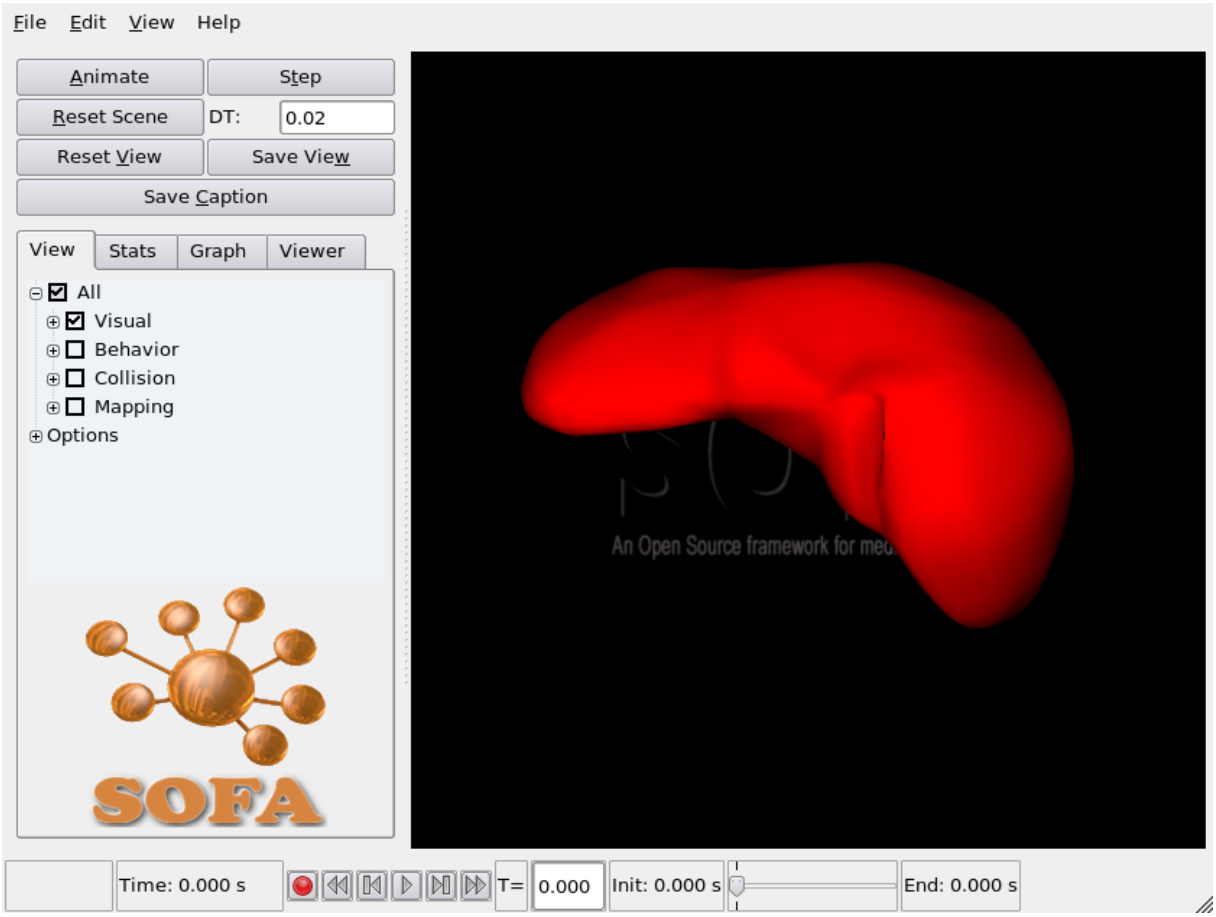


Figure 12.27: SOFA first-time

At any time, you can hide the control panel by moving its right border to the left.

Figure 12.28: Basic Controls

The basics controls are :

- **Animate** : launch the simulation. The simulation won't stop until you press Animate.
- **Step**: Process only one step of the simulation.
- **Reset Scene**: Reset all the components to their initial values.
- **Reset View**: Reset the camera to its original place.
- **Save View**: Save the position and orientation of the camera. Next time you will load your scene, these information will be used.
- **Save Caption**: Take a screenshot of the current simulation.

DT corresponds to the time step used in the computation of the simulation. It can be changed interactively.

12.5.2 View Tab

The "View Tab" is the default tab, you can filter the information you want to be displayed by your viewer. It is quite useful to have a fast and global control.

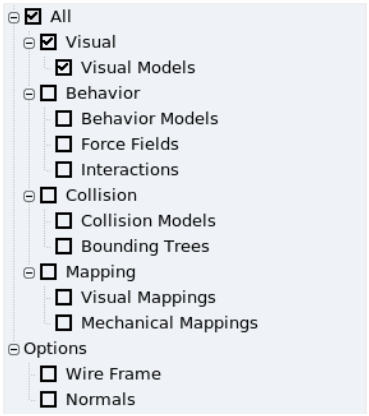


Figure 12.29: Basic Controls

The options are:

- **All**: Enable or disable the display of all the visual information available in SOFA
- **Visual Models**: the graphic representation of the objects
- **Behavior Models**: the mechanical DOFs of the simulation
- **Collision Models**: the models used to perform the collision detection

- **Bounding Tree**: the hierarchical bounding boxes of the collision models
- **Mappings**: All the non-mechanical mappings (for instance the visual mappings that link a mechanical object to its visual representation)
- **Mechanical Mappings**: All the mechanical mapping that propagates the forces and position from a mechanical object to another
- **Interactions**: Interactions of all kind between objects. Some are created by the collision pipeline when a penalty response is used
- **Wire Frame**: Change the way 3D models(visual, and collision) are displayed
- **Normals**: Normals of the visual models

12.5.3 Stats Tab

The "Stats Tab" is a tab displaying an inventory of the collision models present in the scene(how many triangles, lines, points, spheres, are used to perform the collision detection). You can also output some information about the current simulation.

- Dump State: export in "dumpState.data" the state of the simulation
- Log Time: display in the console, the time spent in each step of the simulation. Useful to do some monitoring
- Gnuplot: export gnuplot files. It will export positions, velocities, energies. Files will have the same name as the Object associated in the simulation, following by:
  - **"_x"** for the positions
  - **"_v"** for the velocities
  - **"_Energy"** for the Energies (contains kinetic, potential and mechanical)

  You can specify the directory where you want the files to be saved in the menu Edit.

12.5.4 Graph Tab

The "Graph tab" is certainly the most important tab. It displays the scene graph of the simulation. You can quickly see all the components used in the current simulation. The "Export Graph..." button gives a graphic representation of the inter-dependencies of the objects.

This graph can dynamically change during the simulation: collisions can create new nodes, new components in case of contacts. But you can directly interact with it too. Double clicking on an item of the graph will make appear a small window displaying important data.

To know how to display the information of your brand new Sofa component, please refer to the section "How to configure your Component". Only objects of type "sofa::core::objectmodel::Data" or "sofa::core::objectmodel::DataPtr" can be displayed. It is important to understand that only these information will be kept if you decide to save the simulation. Loading a saved simulation, will just fill the components with this data. This kind of dialog windows give the possibility to modify directly some characteristics of your component. Take care to click on the button "Update" once you have completed your modifications.

A Node gives access to much more interactions: right clicking on one of them makes appear a small context window.

- **Collapse**: Collapse the graph from the current node, close all the nodes below the clicked one
- **Expand**: Expand the graph, and open all the nodes below the clicked one
- **Desactivate**: Deactivate a part of the scene. Everything below this node won't be anymore taken into account. BUT it remains in the scene, and can be Activated again at any time

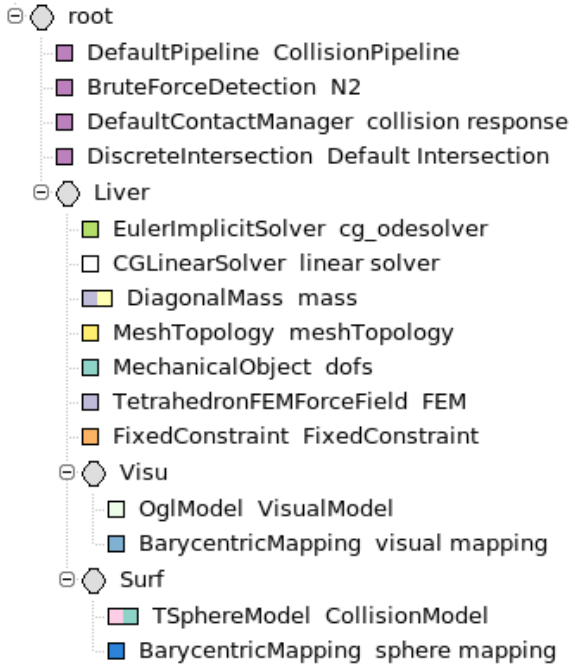


Figure 12.30: Scene Graph for the liver scene

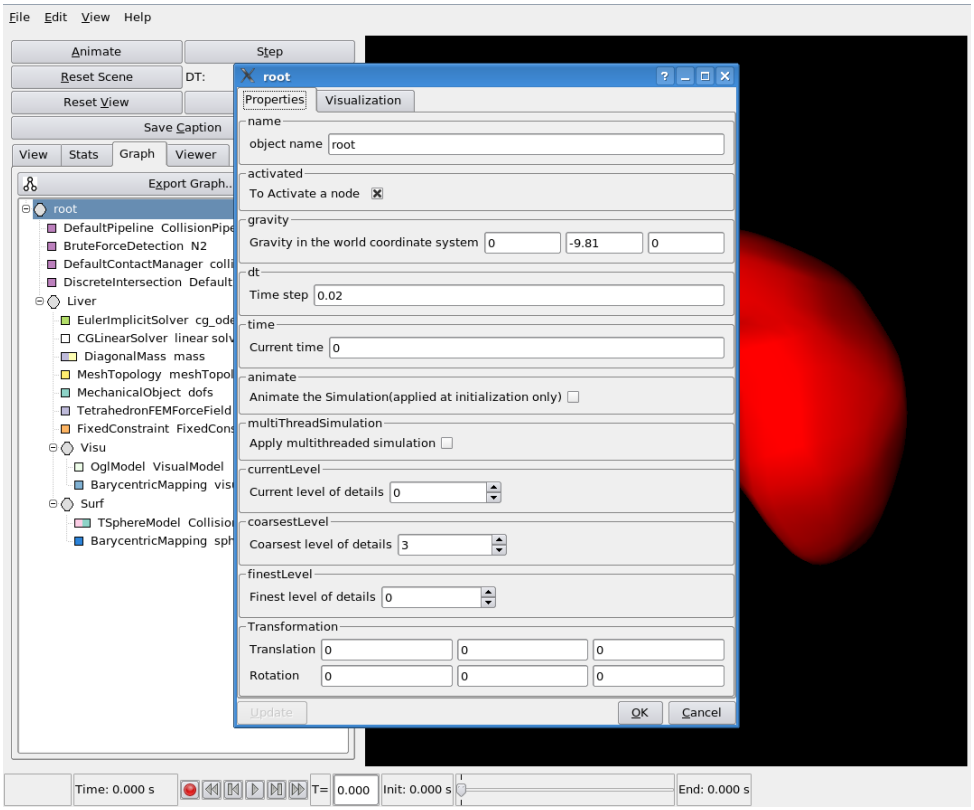


Figure 12.31: Double click on an item of the graph

- **Save Node**: Export in a XML files a part of the simulation
- **Add Node**: Read a XML files describing an object or a whole scene, and put it right below the clicked node. An interesting feature to note, is when you might be always using, and adding the same set of objects, you will find it convenient to add in the file scenes/object.txt the path to them. Like this, they will directly appear in the dialog window by default.
- **Remove Node**: Remove from the simulation everything within the clicked node. You won't be able anymore to make it appear unless you proceed to a restart or reload of the scene.
- **Modify**: Open the same dialog window as might do a double click: this action is common to all the items of the graph

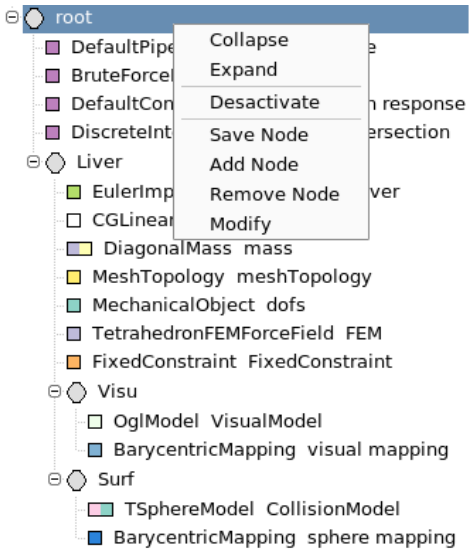


Figure 12.32: Right click on a node of the graph

12.5.5 Viewer Tab

The "Viewer tab" describes all the keyboard shortcuts available for the current viewer.

The last useful option of this tab is the possibility to re-size your viewer, which can be very helpful to record a video at a given resolution.

12.5.6 Interactions

You can interact with the simulation using the mouse with SHIFT + Right Click. A ray will be cast, and if it intersects one collision model of the scene, a spring will be created, allowing you to pull on some elements of the scene.

12.5.7 Architecture

Fig. 12.33 gives an overview of the modular architecture of the GUI for SOFA.

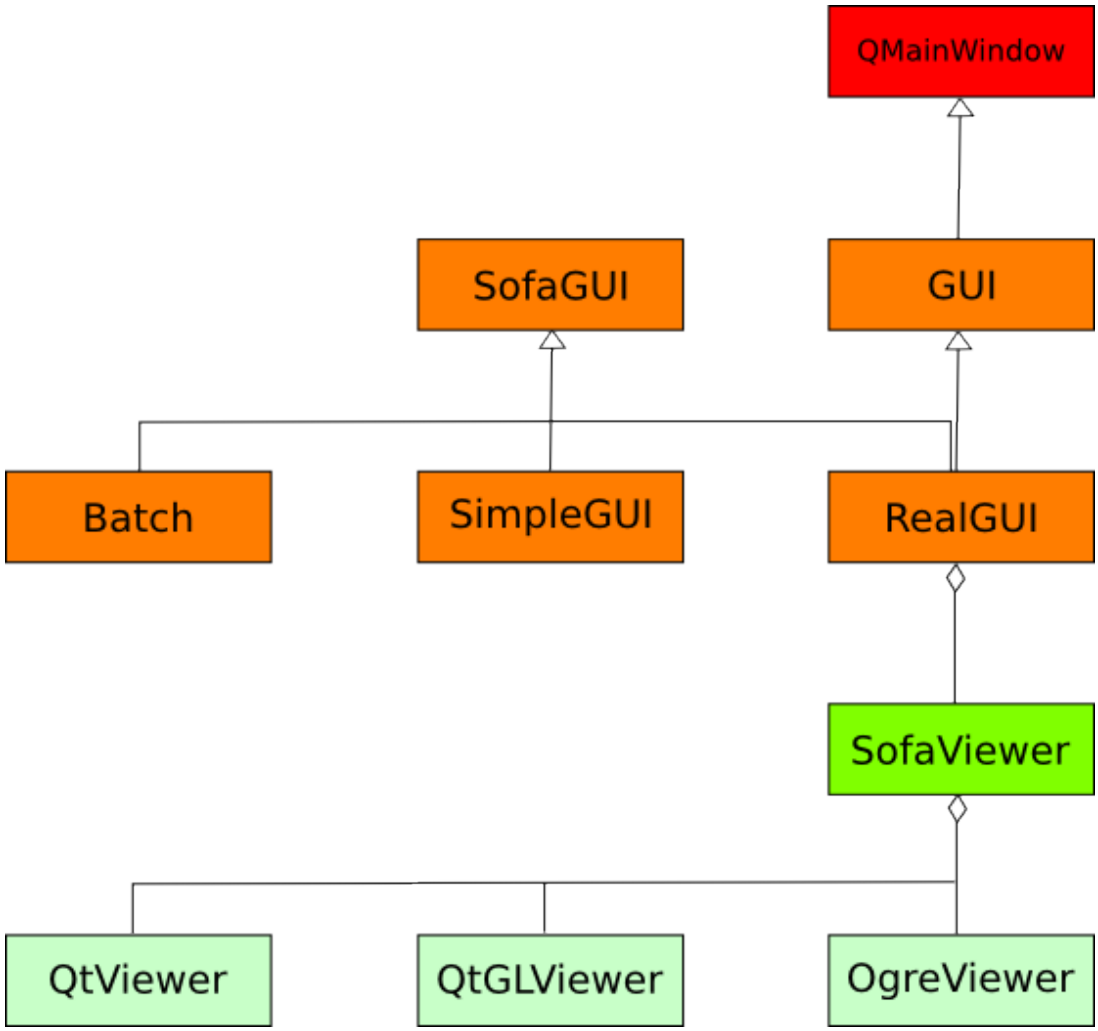


Figure 12.33: A modular Architecture of the GUI

12.5.8 Change the viewer

By default, SOFA provides three viewers, that can be integrated easily to the Qt interface.

- **QtViewer**: a hand made viewer using OpenGL functionality
- **QtGLViewer**: a viewer using the library QGLViewer created by Gilles Debunne. It is distributed directly with SOFA, but you can also download it at: **http://artis.imag.fr/Members/Gilles.Debunne/QGLViewer/**
- **OgreViewer**: this viewer remains experimental, but shows how it is possible to integrate a powerful rendering engine such as OGRE3D. You need to install Ogre. You can download it at **http://www.ogre3d.org**.

To use them, you have to edit the file sofa-default.cfg located in your SOFA directory, and uncomment the lines corresponding to the viewer you want. If you have several viewers activated, you can launch SOFA with a specific one by using the option:

- "runSofa -g qt" : for QtViewer
- "runSofa -g qglviewer" : for QtGLViewer
- "runSofa -g ogre" : for OgreViewer

You can also dynamically change the viewer when SOFA is running. The menu View of the main window displays all the viewer available and let you switch at any time.

If you desire to create a new viewer, the first step is to make it derive from SofaViewer.

12.5.9 Choose the GUI

By default, Sofa provides three GUIs.

- **Batch**: no gui, proceeds to 1000 iterations and then stops
- **GLUT**: GLUT window, implementing only the basic functions. To start the animation, you have to pass the option "-s" to your runSofa
- **Qt**: the default GUI, already described above

The Batch GUI is always available. To use GLUT or Qt interface, you have to uncomment in sofa-default.cfg the corresponding lines. To use them

- "runSofa -g batch" : for no GUI
- "runSofa -g glut" : for GLUT window
- for Qt GUI, please refer to the section above. By default, Sofa is using Qt interface with QtViewer.

If you desire to create a new GUI, the first step is to make it derive from SofaGUI.

12.5.10 Player/Recorder

Figure 12.34: Player/Recorder in Sofa

Sofa provides with the Qt Graphic interface, a compact Player/Recorder of simulation. When you want to record a simulation, press the red button (record). Automatically, some components will be added to your graph, and will create files to save the position, and velocities of all your mechanical elements. To stop recording, press again on the record button. A file with the same name as your simulation will be created, but with the extension ".simu". Sofa is able to read these files, and will initialize correctly the Player.

To readback a recorded simulation, you can process to a step by step(forward, and backward), or a continuous play. You can jump to a specific moment of your recording. You can even change the Dt of the recording, if you want to accelerate, or reduce the velocity of the playing. At any time, you can animate the scene (by pressing the Animate button), to compute the simulation.

The files will be stored by default in the directory scenes/simulation of your SOFA. Nevertheless, you can change this directory by clicking on the menu Edit of the main window.

12.6 Modeler

A modeler for Sofa has been recently created. The purpose of this tool is both to help the new-comers in Sofa to get an overview of the whole framework, and accelerate the process of creation and configuration of a complex simulation for expert users.

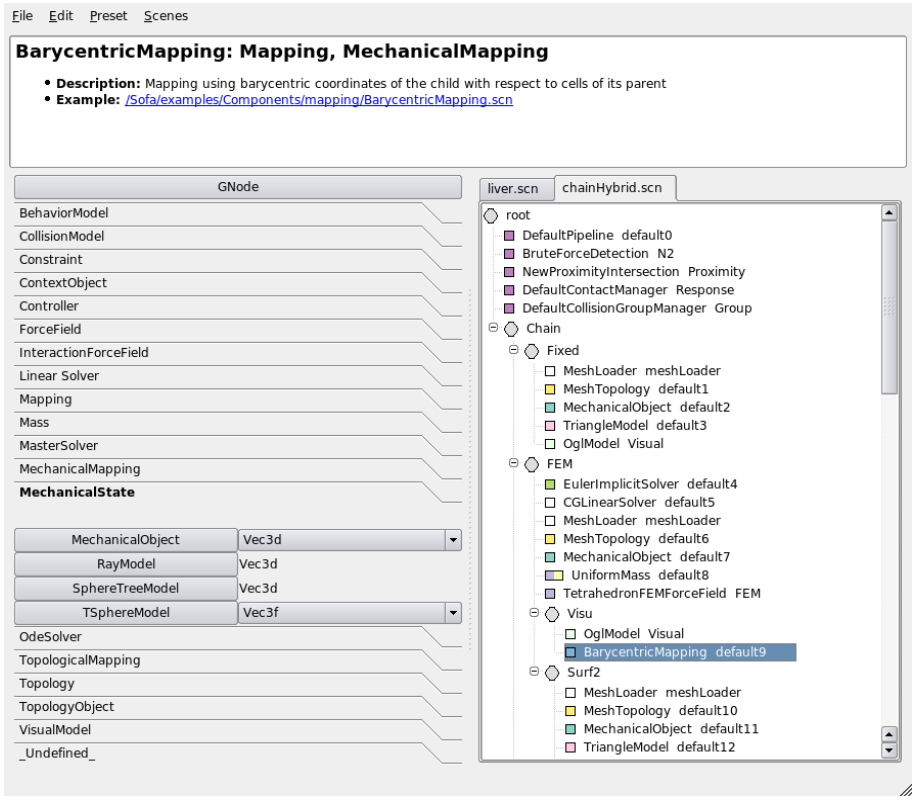


Figure 12.35: Main Window of the Modeler

12.6.1 Library

The left part of the application displays a library of all the different components available in Sofa, sorted by the name of the Base Class. Clicking on any of them will display information about the usage, authors, licence if any, and sometimes a link to an example. This link will open a new tab in the modeler displaying a simulation. At any time, you can launch Sofa from the Modeler, hitting CTRL+R, or going to File ... Run In Sofa.

12.6.2 Graph Editor

The right part seems very similar to the scene graph displayed in Sofa. The behavior is the same. Double clicking on a component will open a dialog, in which you can define all the parameters of the object. But, as you are in an editor, you can easily remove everything you have added. This editor is very convenient to test things. Try to create your object, launch into sofa, modify some components, or parameters, using the documentation displayed each time you select one item.

12.6.3 Modeling

To model a new scene, just make a series of drag and drop of the components you desire from the Library to the Graph Editor. By default, when opening a new tab, all the default components to perform the collision detection are added. If you don't want them, just clear the tab (CTRL+N).
To accelerate the process of creation, preset objects are available. You can build automatically :

- deformable objects
  - in a grid
  - using a tetrahedral mesh

- rigid objects
  - simulated
  - not simulated: they will perform as obstacle like floors, or walls

12.7 Light management

One white global light illuminates the scene by default. This can be changed through a light manager object and a certain number of lights (limited by OpenGL).

The first step is to add the object called "LightManager", preferably at the top of the scene file.

```
1    <Object type="LightManager" />
```

After that, we can add 3 different kinds of lights :

- a positional light (parameters : color, position) ;

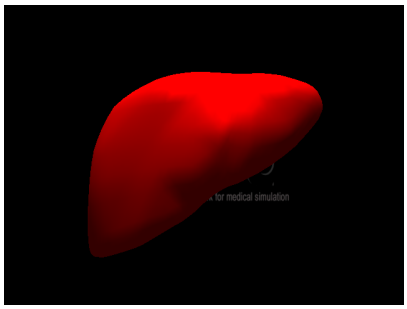


Figure 12.36: Positional Light

```
1        <Object type="PositionalLight" position="0 -5 10" />
```

- a directional light (parameters : color, direction) ;

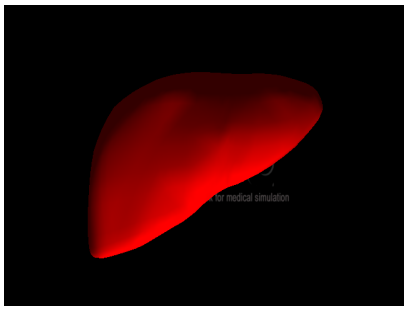


Figure 12.37: Directional Light

```
1        <Object type="DirectionalLight" direction="0 5 0" />
```

- and a spotlight (parameters : color, position, direction, cut off, exponent, attenuation)

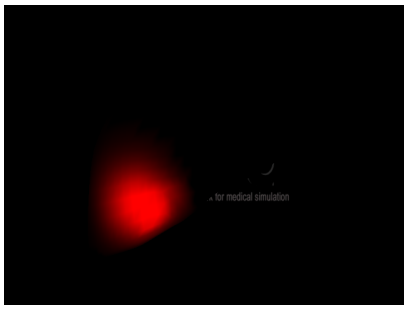


Figure 12.38: Spot Light

```
1        <Object type="SpotLight" position="-3 2 5" direction="0 0 -1" />
```

12.8 Shader management

A complete set of tools about using shaders is implemented into SOFA. The three kinds of shaders (vertex and fragments (mandatory), geometry (optionally)) are available. Shader is used only for Visual Model as OglModel.
The effects of the shader is spread to the associated subtree. Finally, there is only one shader activated for each visual model : if two shaders are present in the same node, only the second will be effective.

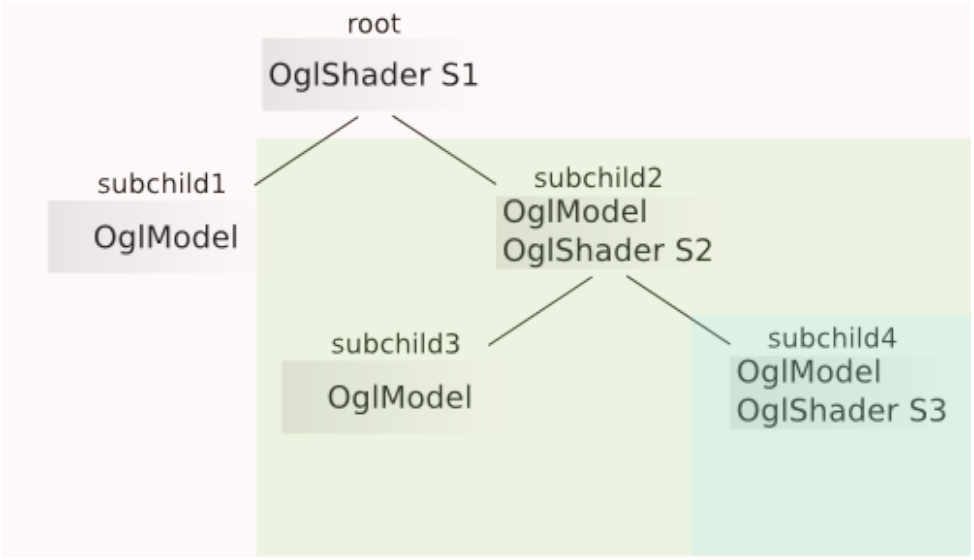


Figure 12.39: Example of shaders' area of effect

To simply include a shader, add this into your node :

```
1        <Object type="OglShader" vertFilename="test.vert" fragFilename="test.frag" />
```

vertFilename and *fragFilename* are the only mandatory parameters. Other optional parameters are about geometry shader : *geoFilename*, *geometryInputType*, *geometryOutputType* and *geometryVerticesOut*. A last parameter, *turnOn*, is for debugging purpose, when you want to disable shader without restarting the scene.

If you want to send values to uniform variables defined into the shader, a certain number of objects is available :

- OglIntVariable,OglInt2,3,4Variable : for int and ivec2,3,4
- OglFloatVariable,OglFloat2,3,4Variable : for float and vec2,3,4
- OglIntVectorVariable, OglIntVector2,3,4Variable : for arrays of int and ivec2,3,4
- OglFloatVectorVariable, OglFloatVector2,3,4Variable : for arrays of float and vec2,3,4

Their parameters are *id* for their name into the shader and *value* (single type) or *values* (array type). Example :

```
<Object type="OglFloat3Variable" id="fragmentColor" value="1.0 0.0 0.0" />
<Object type="OglFloatVariable" id="fragmentOpacity" value="2.0"/>
<Object type="OglFloatVector4Variable" id="MappingTable" values="1.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0" />
```

2D texture can be added with OglTexture2D object.Its parameters are *id*, *texture2DFilename* and *textureUnit*.

```
<Object type="OglTexture2D" texture2DFilename="textures/lights4-small-noise.bmp" textureUnit="1"
    id="planeTexture" />
```

The last object about shaders is a partial support of macro processing in GLSL. It's possible to define macro variable if a part of code is enabled or not. For example, this can be very useful if there is a common code for two 3D objects, one with a texture, and the other with simple colors. You define the macro :

```
#define HAS_TEXTURE
    ...;
    //code about textured 3D object
#else
    ...;
    //code about colored 3D object
#endif
```

and put the following object into the scene file, at the same node as the OglShader used by the textured 3D object:

```
<Object type="OglShaderDefineMacro" id="HAS_TEXTURE" />
```

Part III

Practical guide

Chapter 13

How To

In this document we will try to give small tutorials on various topics you should encounter during your experience with SOFA.

13.1 How To create a simulation

To create your own simulation, from a xml description, or a c++ file, you have to respect some rules. The Modeler can be used to have a quick view of all the components already available in Sofa.

13.1.1 Model a dynamic object

To model a dynamic object, you have to follow that steps:

Mechanical

1. **GNode**: Generally, we give it the name of the whole object
2. **Solver**: choose the solver you want to resolve this part of the simulation (you might need two components actually, a OdeSolver followed by a LinearSolver)
3. **Topology**: describes how the dofs will be connected
4. **MechanicalState**: the degrees of freedom (dofs) of your object. It is the heart of the simulation
5. **Mass**: the mass attached to each dofs of the object
6. **ForceField**: describes the behavior of your object, how it will interact. If you don't specify one, your model won't be deformable
7. **Constraint**: optional

After these steps, you will have a mechanical model, that can be integrated in a Sofa Simulation. Nevertheless, you won't have any visual model, only points representing your dofs.

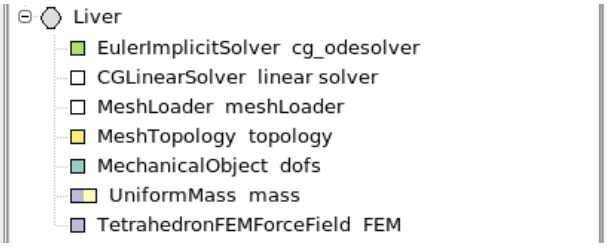


Figure 13.1: Basic example modelling a Finite Element Object

Visual

Using the previously described mechanism of Mapping, you can attach a visual model, of any kind, to represent your mechanical object.

1. **GNode**: add a GNode inside your current object. It will contain the components necessary to do the visual mapping
2. **VisualModel**: this component contains the mesh representing your object
3. **Mapping**: a non-mechanical mapping will connect your mesh to the dofs. This mapping won't transmit forces from your visual model to the dofs. If you are writing...
   - **a c++ file**, take good care of using a non-mechanical template: The second object should be a template of ExtVec3Types
   - **a xml file**, you have to specify the path to the two models to be mapped:
     - object1="../.." : meaning the dofs are located one level below
     - object2="Visual" : where "Visual" is the name of your VisualModel (as described in this example, it is located at the same level as your mapping)

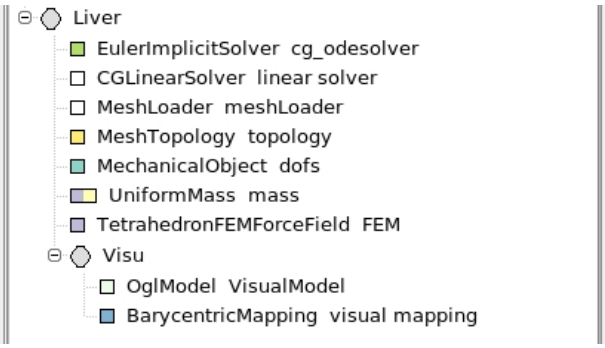


Figure 13.2: Basic example modelling a Finite Element Object with a Visual Model

Collision

If you need to simulate interactions between objects, you will need another node, a Collision Node. In the example we describe, we will use a Triangle Model as collision model. We chose it because, it behaves like most of our collision models, needing a topology and dofs to behave properly. But if you use the simple SphereCollisionModel, this component already contains a topology, dofs and collision model. So you will just have to create a mechanical mapping.

1. **GNode**: add a GNode inside your current object. It will contain the components necessary to do the mechanical mapping
2. **Topology**
3. **MechanicalState**: the dofs of your collision model. They will be used to transmit the forces they receive from the interactions to the real mechanical dofs of your object
4. **CollisionModel**: the model of collision, a sequence of them can be specified (for example, TriangleModel, then LineModel, then PointModel).
5. **MechanicalMapping**: for a XML description of your object, you don't need to specify who is object1 or object2

Your object is now ready to be inserted in a Sofa simulation.
Another example of a full object, using SphereModels.

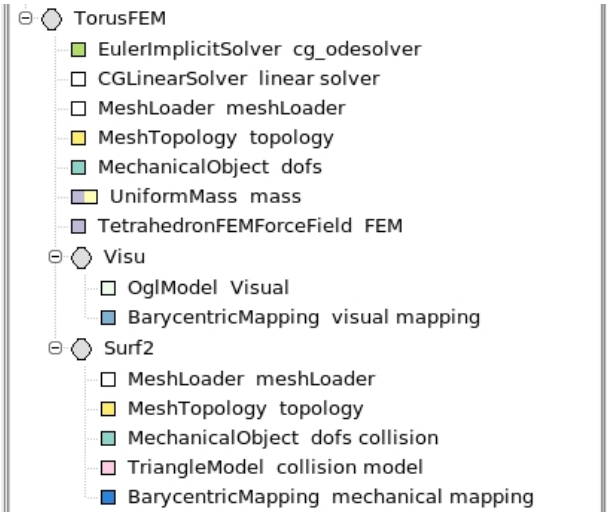


Figure 13.3: Basic example modelling a Finite Element Object with a Visual Model and CollisionModel

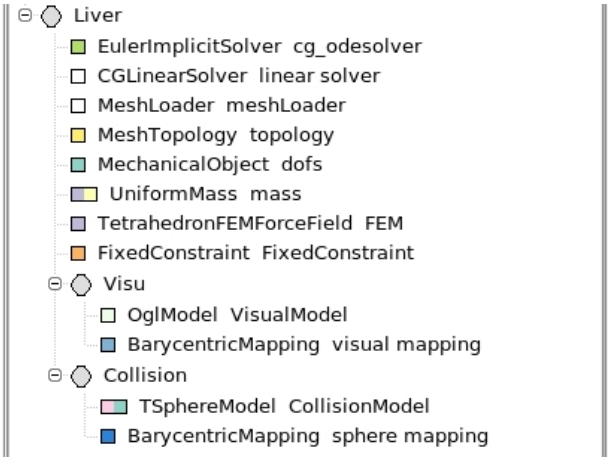


Figure 13.4: Modeling a liver with sphere collision model

13.1.2 Model a static object

Fixed object, like floors, walls, or objects that only must be used as obstacle are easier to model.

1. **GNode**: Generally, we give it the name of the whole object
2. **Topology**: describes how the dofs will be connected
3. **MechanicalState**: the degrees of freedom (dofs) of your object
4. **CollisionModel**: the model of collision, a sequence of them can be specified (for example, TriangleModel, then LineModel, then PointModel). You have to specify the fact that your object is fixed by setting some flags.
   - **moving**: if your object can be displaced. You can think of an external interaction, using an haptic device for instance
   - **simulated**: if you object is controlled by a simulation. Generally, a fixed object is not simulated.
5. **VisualModel**: this component contains the mesh representing your object

No need of any mapping as no forces, or modifications of position will be transmitted.

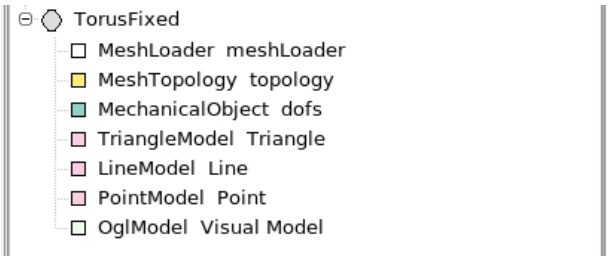


Figure 13.5: Modeling a Fixed object

13.1.3 Include Collisions

To perform collision detection, as you have seen, the objects of the scene must have a or several collision models. But, you will have to set up several components performing the collision detection, and response.

1. **CollisionPipeline**: currently, only our default collision pipeline is available.
2. **CollisionDetection**: method to detect collisions
3. **IntesectionMethods**: depending on the collision detection algorithm, you may have to specify some components to perform the proximity intersection test for example.
4. **ContactManager**: receiving the collisions found, it will generate a response. You can chose the response you want by filling the field "response". By default, we use a penalty response.
5. **CollisionGroupManager**: manages collisions between different kind of simulated objects. It avoids explosions of your simulation by changing the graph dynamically, and putting an appropriated solver above the objects in interaction

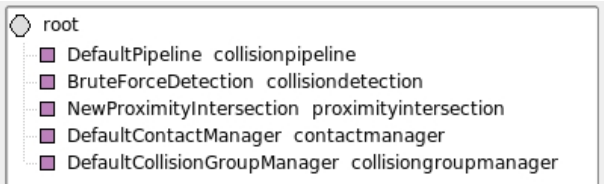


Figure 13.6: Collision Components

13.2 How To create a new Force Field

In SOFA, the Force Field already existing are located in the namespace sofa::component::forcefield. They derive from the core class sofa::core::componentmodel::behavior::ForceField. It is templated by the type of elements you want to model. It can be a deformable object of 1,2,3 or 6 dimensions, or rigid bodies of 2 and 3 dimensions.
The simplest way to implement your own ForceField is :

1. make it derive from sofa::core::componentmodel::behavior::ForceField
2. implements the following virtual functions: addForce, getPotentialEnery. Others virtual functions exist like addDForce, addDForceV (if you want to make dynamics), and you should read the doxygen documentation about ForceField.
3. as with all the others component you might create, you have to add it to the project.
   Edit $SOFA_DIR/modules/sofa/component/component.pro, and add the path to new files in the section HEADERS and SOURCES.
4. Edit $SOFA_DIR/modules/sofa/component/init.cpp and add your new forcefield to the list. This step is compulsory for Windows system, and does the linking of a new component to the factory. If you forget it, your component won't be created at initialization time.

The method addForce computes and accumulates the forces given the positions and velocities of its associated mechanical state.
If the ForceField can be represented as a matrix, this method computes

$$f+ = Bv + Kx$$

This method is usually called by the generic ForceField::addForce() method.

13.3 How To include objects in a XML simulation

If you have the same object appearing several times in your simulation, you may find convenient to be able to describe it only once, and then, only include this description. But you may need to specify parameters of this object, or apply basic transformations (translation, rotation, scale).

13.3.1 Include an object

The basic command to include an object to your scene is:

```
<include name="YourObjectName" href="PathToYourXMLFile/YourFile.xml" />
```

This will load the content of the xml file you specified in **href** under a new node called **YourObjectName**.

Now, you may want to modify some parameters of this special object. Simply add the name of the parameter followed by its value. When you main scene will be loaded, it will replace all the occurrences of this parameter by the value you wrote. **Remember:** in the object description must appear all the parameters your want to modify! The only parameter that can't be modified is **type**, specify the nature of a component.

```
<include name="YourObjectName1" href="PathToYourXMLFile/Object.xml" color="red"/>
<include name="YourObjectName2" href="PathToYourXMLFile/Object.xml" color="blue"/>
```

Two entities of the same object will be created under two different nodes, **YourObjectName1** and **YourObjectName2**. In Object.xml, the visual model has a default value for the parameter **color**. Then, when you will launch your scene, the first object will appear in red, and the second in blue, but at the same position.

To apply some basic transformations, translation, rotation, scale, simply specify in you Object.xml default values for translation, rotation and scale. Then, you will be able to include your object specifying new configurations.
In Object.xml, typically express:

- the MechanicalObjects :

```
<Object type="MechanicalObject"
        dx="0" dy="0" dz="0" rx="0" ry="0" rz="0" scale="1.0"/>
```

- the VisualModels

```
<Object type="OglModel" fileMesh="YouMesh.obj"
        color="white"
        dx="0" dy="0" dz="0" rx="0" ry="0" rz="0" scale="1.0"/>
```

Now, to include two object from the same file description, with a different color (or other parameter) and different position, orientation, and scale

```
<include name="YourObjectName1" href="PathToYourXMLFile/Object.xml"
         color="red" dx="1" ry="90" scale="0.5"/>
<include name="YourObjectName2" href="PathToYourXMLFile/Object.xml"
         color="blue" />
```

This will translate the red object along the X axis, and do a 90 degrees rotation along the Y axis, reducing its scale with a factor 0.5. The blue object will be loaded with no modifications.

A problem may appear if several parameters have the same name, and you only want to modify a special one. For instance, you have two VisualModels in your **Object.xml** with the parameter color.

```
<Object type="OglModel" name="visual1" fileMesh="YouMesh1.obj"
        color="white" />
        ...
<Object type="OglModel" name="visual2" fileMesh="YouMesh2.obj"
        color="white" />
```

When you will include it, you want **visual1** to be red, and **visual2** to be blue. Simply specify before the name of the parameter, the name of the component followed by two _ :

```
<include name="YourObjectName1" href="PathToYourXMLFile/Object.xml"
         visual1__color="red"  />
<include name="YourObjectName2" href="PathToYourXMLFile/Object.xml"
         visual2__color="blue" />
```

13.3.2 Including a set of components

The restriction of this mechanism if that the loaded file will be placed under a node. If you simply want to load one, or several components, that you would like to place inside the current node, you have to specify one keyword. In you XML description of the components, specify as the name of the root node **Group** . When this file will be loaded, the contents will be directly placed inside the current node. One benefit would be to describe the components needed to perform the collision detection and response only one, in a **Group** XML file, and simply include it at the beginning of the scene files.

```
<Node name="Group">
        <Object type="CollisionPipeline" name="DefaultCollisionPipeline" depth="6"/>
        <Object type="BruteForceDetection" name="Detection" />
        <Object type="MinProximityIntersection" name="Proximity"
                alarmDistance="0.3" contactDistance="0.2" />
        <Object type="CollisionResponse" name="Response" response="default" />
        <Object type="CollisionGroup" name="collisionGroup" />
</Node>
```

And to include it, just use the same mechanism but **WITHOUT** specifying a name.

```
<include href="PathToYourXMLFile/Components.xml" />
```

If you decide that you want to place these components under a new node, simply specify a name, this will overwrite the keyword **Group**.

```
<include href="PathToYourXMLFile/Components.xml" name="UnderNode"/>
```

13.3.3 Commented examples

The scene **Sofa/examples/Demos/chainHybrid.scn** has been created using the include mechanism. We described several kind of Torus, and they are included defining a new position, and orientation. The scene **Sofa/examples/Components/forcefield/StiffSpringForceField.scn** is using the **Group** keyword to only include a special component. The include mechanism is highly used for the topologies. To have dynamic topologies, several components are needed, a TopologyContainer, a TopologyModifier, a TopologyAlgorithms, a GeometryAlgorithms. We have already created these set of components for the EdgeSetTopology, ManifoldEdgeSetTopology, QuadSetTopology, TriangleSetTopology, HexahedronSetTopology, PointSetTopology, TetrahedronSetTopology. Several examples in **Sofa/examples/Components/topology** are using them.

13.4 How To make your Component modifiable

When you create your own component, it can be very convenient to display some internal data, or be able to modify its behavior by modifying a few values. It is made possible by the usage of two objects:

- sofa::core::objectmodel::Data
- sofa::core::objectmodel::DataPtr

They are templated with the type you want. It can be "classic" types, bool, int, double (...), or more complex ones (your own data structure). You only have to implement the stream operators "«" and "»". In the constructor of your object, you have to call the function initData, or initDataPtr.
for instance, let's call your class **foo**. You want to control a parameter of type boolean called **verbose**. You want it to be displayed

```
foo(): verbose(initData(&verbose, false, "verbose", "Helpful comments", true, false)){}
```

initData takes several parameters:

1. address of the Data
2. default value: it must be of the same type as your template(**OPTIONAL**)
3. name of your Data: it will appear in your XML file
4. description of your Data: it will appear in the GUI
5. boolean to know whether or not it has to be displayed in the GUI(**OPTIONAL**, default value true: always displayed)

6. boolean to know whether or not your Data will be ONLY readable in the GUI(**OPTIONAL**, default value false: always readable and writable)

Once you have modified your Datas in the GUI, pressing the button "Update" will call the virtual method "void reinit()" inherited by all the objects. It is up to you to implement it in your component if the change of one field requires some computations or actualization. You can chose to hide a specified Data from the GUI at any time by using the method setDisplayed(bool). You can chose to enable or disable the write access of a specified Data from the GUI at any time by using the method setReadOnly(bool).

13.5 How to use the carving manager

CarvingManager uses a sphere model to remove elements from a surface model. When the sphere collides with the surface model, all the elements which are in contact with the sphere are removed.

NOTE: To be able to detect the proximity between the models, the surface model must be mapped into a triangle model using a topological mapping, like in the example showed below:

```
<Node name="Liver" >
    <Object type="MeshLoader" filename="mesh/liver.msh" />
    <Object type="MechanicalObject" />
    <include href="Objects/TetrahedronSetTopology.xml" />
    <Node name="CollisionModel" >
        <include href="Objects/TriangleSetTopology.xml" />
            <Object type="Tetra2TriangleTopologicalMapping" object1="../../Container"
            object2="Container"/>
        <Object type="TriangleSet" contactStiffness="100" />
    </Node>
</Node>
```

The different Data's of this object are:

- **modelTool**: This data determines the name of the sphere model. If empty, a sphere model will be searched in the node of the CarvingManager. If it is not found, an error will occur.
- **modelSurface**: It determines the surface model. If empty, a surface model (more exactly a triangle model mapped to another geometrical model using a topological mapping) will be searched in the whole scene. If not found, an error occurs.
- **active**: activate/deactivate the object.
- **key**: activate the object only when an event using this key is caught.
- **keySwitch**: activate this object when an event using this key is caught and deactivate it when the event is caught.

An example of how to use this object can be found in *examples/component/collision/CarvingManager.scn*

Chapter 14

How to contribute to this documentation

14.1 Document structure

This document gathers the content of other documents located in different subdirectories. These documents can also be compiled as stand-alone documents. The structure can be illustrated as follows:

- `sofaDocumentation/sofadocumentation.tex` : the root document.
- `macros_docu.tex` : custom commands and macros. This file is included by the root document.
- `introduction` : a subdirectory containing a chapter of this document.
  - `introduction.tex` : a stand-alone article containing the same. This file may include `../macros_docu.tex` to use the common custom commands.
  - `introduction_body.tex` : the text of the chapter/article. This file is included as a chapter by `sofadocumentation.tex`, and included as full article text by `introduction.tex`.
- there could (and hopefully will !) be other subdirectories with a similar structure.

14.2 Compiling the document

14.2.1 File formats

The graphics are handled by: `\usepackage[pdftex]{graphicx}`. This allows the inclusion of `.png` images rather than `.eps`, which makes the image files much smaller, and compilation of `html` faster. The result of the compilation is a `.pdf` rather than a `.dvi` file.

14.2.2 Include paths

The root document includes files in the subdirectories, which in turn include files too. The path from the subdirectory (used when compiling a stand-alone article) is the same as from the parent directory (used when compiling the whole report).

14.2.3 HTML

HTML can be generated using the following command:
`latex2html sofadocumentation.tex -mkdir -dir ./html -show_section_numbers -split 1`
Currently, the listings do not appear in html.

Bibliography

[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, 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.

[3] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *SIGGRAPH '98*, pages 43–54. ACM Press, 1998.

[4] 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.

[5] Hadrien Courtecuisse, Jérémie Allard, Christian Duriez, and Stéphane Cotin. Preconditioner-based contact response and application to cataract surgery. In *MICCAI 2011*. Springer, September 2011.

[6] 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.

[7] Christian Duriez, Hadrien Courtecuisse, Juan-Pablo de la Plata Alcalde, and Pierre-Jean Bensoussan. Contact skinning. In *Eurographics conference (short paper)*, 2008.

[8] 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.

[9] 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.

[10] Richard J. Adams et Blake Hannaford. Stable haptic interaction with virtual environments. *IEEE Transactions on Robotics and Automation*, pages 465–474, 1999.

[11] 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.

[12] 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.

[13] 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.

[14] 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.