Sofa Documentation

The Sofa Team

August 12, 2008

Abstract

This is the Sofa documentation

Contents

Chapter 1

Introduction to SOFA

François Faure

1.1 Brief overview

SOFA is an open-source C++ library for physical simulation, primarily targeted to medical simulation. It can be used as an external library in another program, or using one of the associated GUI applications.

The main feature of SOFA compared with other libraries is its high flexibility. It allows the use of multiple interacting geometrical models of the same object, typically, a mechanical model with mass and constitutive laws, a collision model with simple geometry, and a visual model with detailed geometry and rendering parameters. Each model can be designed independently of the others. During run-time, consistency is maintained using mappings.

Additionally, SOFA scenes are modeled using a data structure similar to hierarchical scene graphs commonly used in graphics libraries. This allows the splitting of the physical objects into collections of independent components, each of them describing one feature of the model, such as mass, force functions and constraints. For example, you can replace spring forces with finite element forces by simply replacing one component with another, all the rest (mass, collision geometry, time integration, etc.) remaining unchanged.

Moreover, simulation algorithms, such as time integration or collision detection and modeling, are also modeled as components in the scene graph. This provides us with the same flexibility for algorithms as for models.

Flexibility allows one to focus on its own domain of competence, while re-using the other's contributions on other topics. However, efficiency is a major issue, and we have tried to design a framework which allows both efficiency and flexibility.

1.2 Commented example

Figure 1.1 shows a simple scene composed of two different objects, one rigid body and one particle system, and linked by a spring. This scene is modeled and simulated in C++ as shown in section 1.6.1. The corresponding scene graph is shown in figure 1.2. Note that the graph in the left of figure 1.1 only displays a hierarchical view, while the whole graph includes additional pointers displayed as dashed arrows in figure 1.2.

The scene is modeled as a tree structure with four nodes:

- `root`
- `deformableBody` corresponds to the elastic string
- `rigidBody` corresponds to the rigid object
- `rigidParticles` corresponds to a set of particles (only one in this case) attached to the rigid body

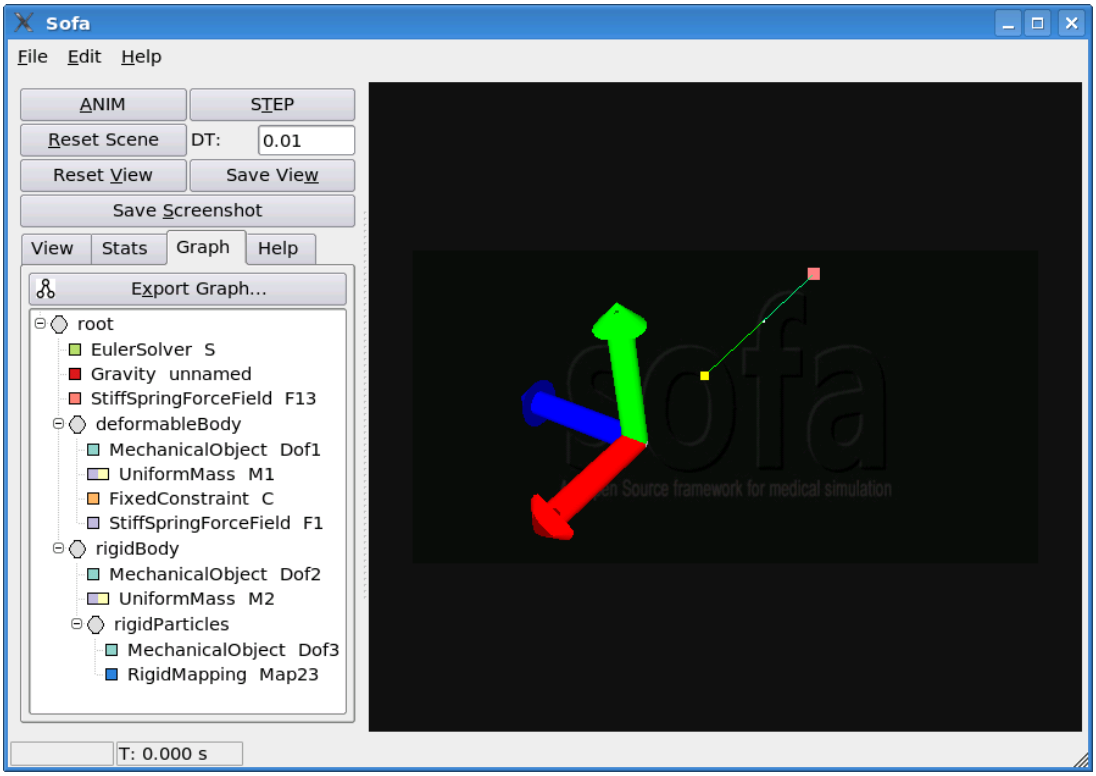

Figure 1.1: A pendulum composed of a rigid body (reference frame and yellow point) attached to an elastic string (green) fixed at one end (pink point). The corresponding scene graph is displayed on the left.

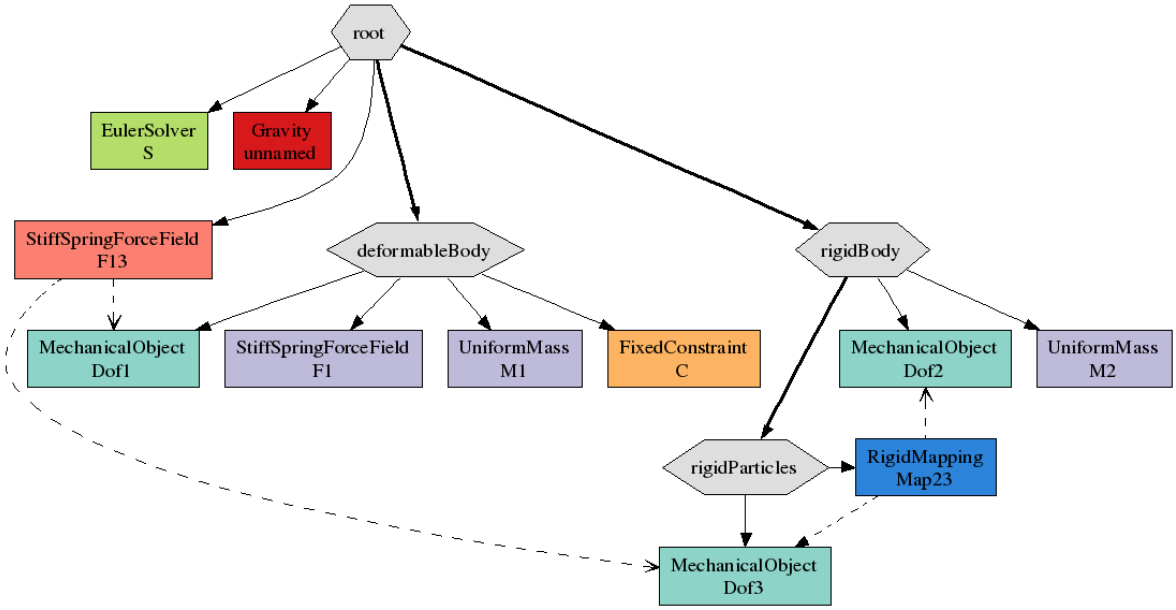

Figure 1.2: The scene graph of the mixed pendulum. The nodes are displayed as grey hexagons, while the components are displayed as rectangles with colors associated with their types or roles. The bold plain arrows denote node hierarchy, while the thin plain arrows point to the components attached to the nodes, and the dotted arrows denote pointers between components.

Each node can have children nodes and *components*. Each component implements a reduced set of functionalities.

One of the most important type of component is the `MechanicalObject`, which contains a list of *degrees of freedom* (DOF), i.e. coordinates, velocities, and associated auxiliary vectors such as forces and accelerations. All the coordinates in a `MechanicalObject` have the same type, e.g. 3D vectors for particles, or (translation, rotation) pairs for rigid bodies. `MechanicalObject`, like many other SOFA classes, is a generic (C++ template) class instantiated on the types of DOF it stores. The particle DOFs are drawn as white points, whereas the rigid body DOFs are drawn as red, green, blue reference frame axes. There can be at most one `MechanicalObject` attached to a given node. This guarantees that all the components attached to the same node process the same types of DOF. Consequently, the particles and the rigid body necessarily belong to different nodes.

In this example, the masses are stored in `UniformMass` components. The types of their values are related to the types of their associated DOF. `UniformMass` is derived from the abstract `Mass` class, and stores only one value, for the case where all the associated objects have the same mass. If necessary, it can replaced by a `DiagonalMass` instanciated on the same DOF types, for the case where the associated objects have different masses. This is an important feature of SOFA : each component can be replaced by another one deriving from the same abstract class and instantiated on the same DOF types. This results in a high flexibility.

The `FixedConstraint` component attached a particle to a fixed point in world space, drawn in pink. The constraints act as filters which cancel the forces and displacements applied to their associated particle(s). They do not model more complex constraints such as maintaining three points aligned.

The `StiffSpringForceField` stores a list of springs, each of them modeled by a pair of indices, as well as the standard physical parameters, stiffness, damping and rest length.

The rigid body is connected to the deformable string by a spring. Since this spring is shared by the two bodies, it is modeled in the `StiffSpringForceField` attached to a common ancestor, the graph root in this example. Our springs can only connect particles. We thus need to attach a particle to the rigid body. Since the particle DOFs types are different from the rigid body DOF types, they have to be stored in another `MechanicalObject`, called `rigidParticleDOF` in this example, and attached to a different node. However, `rigidParticleDOF` is not a set of independent DOF, since they are fixed in the reference frame of the rigid body. We thus attach it to a child node of the rigid body, and connect it to `rigidDOF` using a `RigidMapping`. This component stores the coordinates of the particle in the reference frame of the rigid body. Its task is to propagate the position, velocity and displacement of the rigid body down to the yellow particle, and conversely, to propagate the forces applied to the particle up to the rigid body.

Mappings are one of the major features of SOFA . They allow us to use different geometric models for a given body, e.g. a coarse tetrahedral mesh for viscoelastic internal forces, a set of spheres for collision detection and modeling, and a fine triangular mesh for rendering.

The gravity applied to the scene is modeled in the `Gravity` component near the root. It applies to all the scene, unless locally overloaded by another gravity component inside a branch of the tree.

The abstract component classes are defined in namespace `core::componentmodel`.

So far, we have discussed the physical model of the scene. To animate it, we need to solve an *Ordinary Differential Equation* (ODE) in time. There are plenty of ODE solvers, and SOFA allows the design and the re-use of a wide variety of them. Here we use a simple explicit Euler method, modeled using an `EulerSolver` component. It triggers computations such as force accumulation, acceleration computation and linear operations on state vectors. More sophisticated solvers are available in SOFA , and can be used by simply replacing the `EulerSolver` component by another one, e.g `RungeKutta4` or `CGImplicit`.

Other capabilities of SOFA , such as collision detection and response, will be discussed in subsequent sections.

1.3 Multi-model objects

An important feature of Sofa is the possibility of using different models of a single physical object. Figure 1.3 shows a scene graph representing a liver, and three different images of it. The liver exhibits three different geometries for mechanics, rendering and collision. The corresponding xml code is given in section 1.6.2.

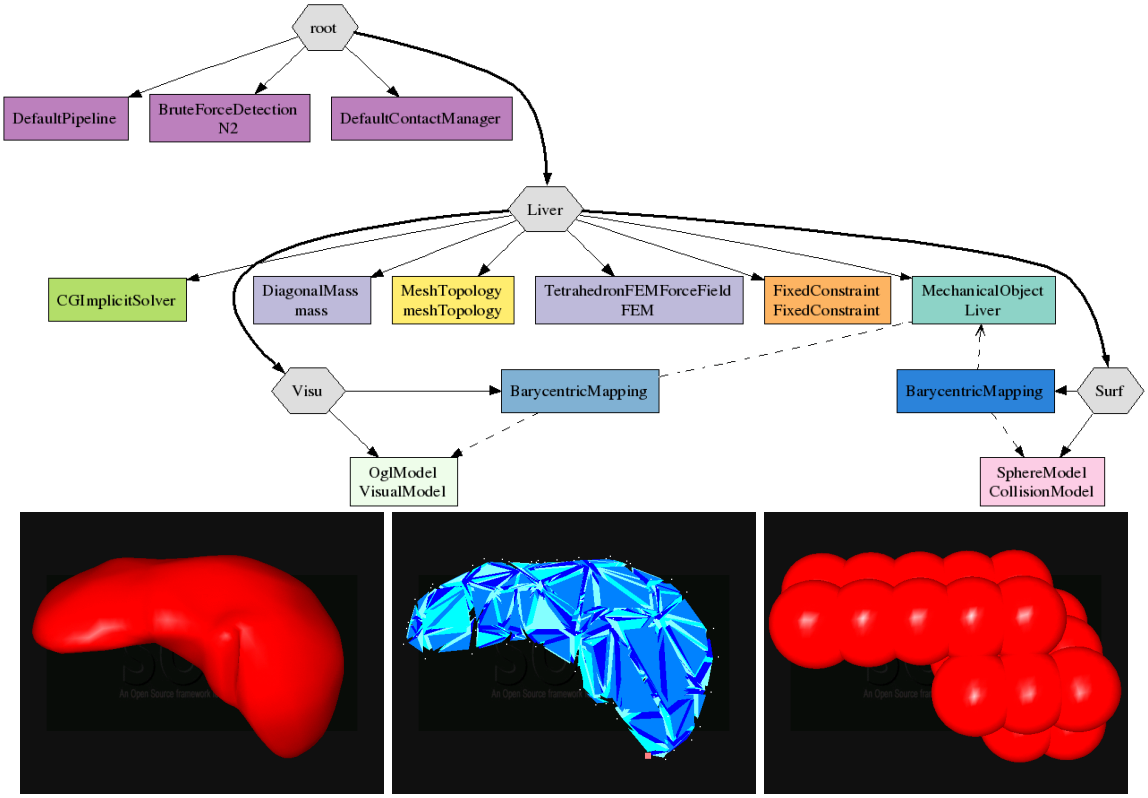

Figure 1.3: A liver. Top: scene graph. Bottom: visual model, mechanical model, collision model, respectively.

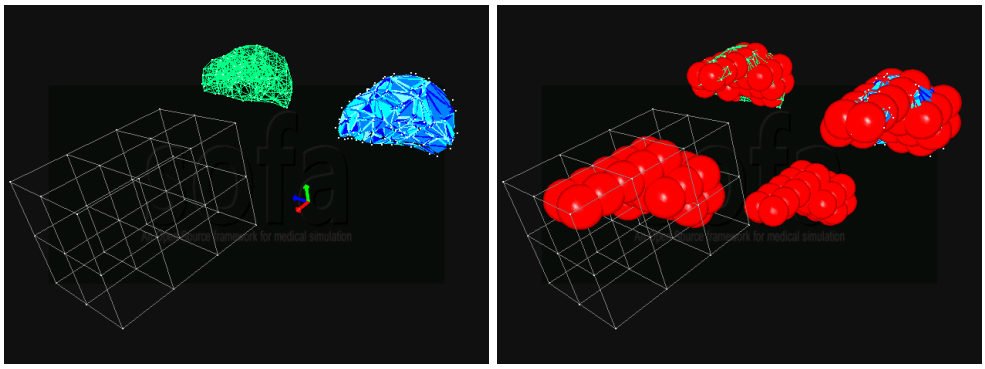

Figure 1.4: Left: four behavior models (from left to right: deformable grid, springs, rigid, tetrahedral FEM) combined with the same collision model (right).

On top of the scene, collision-related components allow a user to interact with the collision models using rays casted from the mouse pointer and hitting collision models.

The liver is modeled using three nodes, in two levels. The parent level contains the mechanical DOFs (particle positions and velocities) in a `MechanicalObject` component. These DOFs are the mechanically independent degrees of freedom of the object, in Lagrange's formalism. The node also contains components related to the dynamics of the particles, such as mass and internal forces. We call it the *behavior model*.

The two other nodes are in the lower level because during the simulation, their coordinates are totally defined by the coordinates of their parent node. Thus, they do not belong to the set of mechanically independent DOFs. *Mappings* are used to compute their positions and velocities based on their parent's, using the pointers represented as dashed arrows. Mappings are not symmetric. The motion of the parent DOFs is mapped to the children DOFs, whereas the motion of the children DOFs is not mapped to their parent. This ensures consistency.

The `VisualModel` has vertices which are used for rendering, along with other rendering data such as a list of polygons, normals, etc. The mapping is one-way and the mapped DOFs have no mechanical influence.

The `SphereModel` class derives from `MechanicalObject`, with an additional radius value. It also derives from `CollisionModel`, which allows it to be processed by the collision detection and modeling pipeline. When contact or mouse interaction forces are applied to the spheres, the forces are propagated bottom-up to their parent DOFs by the mapping (see section 2.1). This allows the contact forces to be taken into account in the dynamics equations. The mapping is thus two-ways and derives from `MechanicalMapping` instead of `Mapping`. This is why it has a different color in the image of the scene graph. Again, the mechanical mappings are not symmetric: the forces are propagated from the children to the parents, not the other way round.

Mappings only propagate positions top-down, whereas MechanicalMappings additionally propagate velocities top-down and forces bottom-up.

Mapped models can be designed independently of their parent models, provided that the adequate (mechanical) mapping is available. This results in a high flexibility. For example, collision spheres can be replaced by collision triangles without changing anything in the behavior model or in the visual model. Similarly, other visual models can be used without modifying the behavior and collision models, and different behavior models can be used with the same collision and visual models, as illustrated in figure 1.4.

1.4 Recursive data processing

A typical simulation program, controlled by an application such as the Graphics User Interface (GUI), looks like the one given in figure 1.5. In SOFA , each of the simulation methods is implemented as a

```
init();
repeat {
    animate();
    draw();
}
```

Figure 1.5: Pseudocode for a standard simulation program.

```
f = 0
accumulateForces(f,x,v);
a = f/M;
a = filter(a);
x += v * dt;
v += a * dt;
```

Figure 1.6: Pseudocode for explicit Euler integration.

recursive graph traversal, `InitVisitor`, `AnimateVisitor` and `VisualDrawVisitor`, respectively. Visitors are explained in the next section.

1.4.1 Visitors

The data structure is processed using objects called *visitors*. They recursively traverse the tree structure and call appropriate virtual methods to a subset of components during the *Top-Down Traversal* (TDT), using virtual method `Visitor::processNodeTopDown`, then during the *Bottom-Up Traversal* (BUT), using virtual method `Visitor::processNodeBottomUp`.

For example, the `VisualDrawVisitor` draws the `VisualModel` components during the TDT, and does nothing during the BUT. The `MechanicalComputeForceVisitor` accumulates the forces in the appropriate DOF vectors during the TDT, then propagates the forces to the parent DOFs using the mechanical mappings during the BUT.

When processed by a visitor a, a component can fire another visitor b through its associated sub-tree. Visitor a can continue once visitor b is finished. During the TDT, each traversed component decides whether the calling visitor continues, or prunes the sub-tree associated with the component, or terminates.

The components directly access their sibling components only, except for the mappings. A component traversed by a visitor can indirectly access the data in its associated sub-tree in read-write mode using visitors, whereas data in its parent graph is read-only and only partially accessible using method `getContext`. Sibling nodes of the same type can be traversed by visitors in arbitrary order.

The visitors belong to namespace `sofa::simulation`.

1.4.2 ODE Solvers

When an `AnimateVisitor` traverses a node with an `OdeSolver` component, the solver takes the control of its associated subtree and prunes the `AnimateVisitor`. The solver triggers visitors in its associated subtree to perform the standard mechanical computations and integrate time.

The simplest solver is the explicit Euler method, implemented in `EulerSolver`. The algorithm is shown as pseudocode in figure 1.6. Net force is computed in the first line. In the second line, the acceleration is deduced by dividing the force by the mass. Then the accelerations of the fixed points are canceled. Finally, position and velocity are updated.

This algorithm can not be directly implemented in SOFA because there are no state vectors x,v,f,a which gather the state values of all the objects in the scene. The solver processes an arbitrary number of objects, of possibly different types, such as particles and rigid bodies. Each physical object carries its state values and auxiliary vectors in its own `MechanicalObject` component, which is not directly accessible to the solver.

The solvers represent state vectors as `MultiVector` objects using symbolic identificators implemented in class `VecId`. There are four staticly predefined identificators: `VecId::position()`, `VecId::velocity()`, `VecId::force()` and `VecId::dx()`. A `Multivector` declared by a solver with a given VecId implicitly

refers to all the state vectors in the different MechanicalObject components with the same VecId in the solver's subtree.

Vector operations can be remotely triggered by a solver using a visitor of a given type, which defines the operator, and given VecIds, which define the operands. During the subtree traversal, the operator is applied to the given vectors of the traversed MechanicalObject components.

For example, let us comment the visitors performed by the EulerSolver shown in figure 1.1. Its implementation is in method component::odesolver::EulerSolver::solve(double). First, multivectors are declared.

Then method core::componentmodel::behavior::OdeSolver::computeForce(VecId) is called. It first fires a MechanicalResetForceVisitor to reset the force vectors of all the MechanicalObject components. It then fires a MechanicalComputeForceVisitor. During the TDT, each component derived from core::componentmodel::behavior::BaseForceField computes and accumulates its force in its sibling MechanicalObject. In the example shown in figure 1.1, F13 adds its contribution to Dof1 and Dof3, then F1 and M1 add their contributions to Dof1, then M2 to Dof2. Then during the BUT, the mechanical mappings sum up the forces of their chid DOF to their parent DOF, *i.e.*, the force in Dof3 to Dof2 through M23 in the same example. Note that branches deformableBody and rigidBody can be processed in parallel. At the end, the force vector in Dof1 contains the net force applied to the particles, and the force vector in Dof2 contains the net (six-dimensional) force applied to the rigid body.

Then method OdeSolver::accFromF(VecId,VecId) fires a MechanicalAccFromFVisitor. Each component derived from core::componentmodel::behavior::BaseMass computes the accelerations corresponding to the forces in its sibling MechanicalObject.

Then method OdeSolver::projectResponse fires a MechanicalApplyConstraintsVisitor. All the core::componentmodel::behavior::BaseConstraint components (component C in the example) filter the acceleration vector to maintain some points fixed.

Once the acceleration is computed, multivector methods are used to update the positions and velocities. Here again, visitors are used to perform the desired operation in each traversed MechanicalObject.

MultiVector operations are pruned at the first level for efficiency, because the solvers deal with the mechanically independent state variables rather than the mapped variables. Moreover, the mapped coordinates can not be assumed to vary linearly along with their parent variables. Applying a MechanicalPropagatePositionAn is thus necessary to update the mapped DOFs based on the mechanically independent DOFs. This visitor is automatically performed after time integration, as one can see in the code of method MechanicalIntegrationVisitor::fwdOdeSolver. It is also used by some solvers when auxiliary states are needed, as discussed in section 1.5, in order to update the mapped DOFs.

1.5 State vectors

The state vectors contain the coordinates, velocites, and other DOF-related values such as force and acceleration. They are stored in MechanicalObject components. This template class can be instanciated on a variety of types to model particles, rigid bodies or other types of bodies. The template parameter is a DataTypes class which describes data and data containers, such as the the type of coordinates and coordinate derivatives used. These two types are the same in the case of particles, but they are different in the case of rigid bodies.

Each MechanicalObject can represent a set of physical objects of the same type, such as particles. The coordinate state vectors are defined by the VecCoord type, while the derivatives (velocity, acceleration, force, small displacement) are defined by the VecDeriv type. Each MechanicalObject stores two arrays of state vectors, one for coordinates and the other for derivatives, as illustrated in figure 1.7.

Auxiliary vectors are necessary for complex solvers, such as RungeKutta2Solver. This solver first performs a half-length Euler step, then evaluates the derivative of this new state (called the *midpoint*), and finally uses this derivative to update the initial state over a whole time step.

To compute the forces at the midpoint while keeping the initial state for further use, we use the auxiliary vectors newX and newV. However, components such as forces and constraints use state vectors, and we have to make sure that they use the right ones. To ensure consistency and make the use of auxiliary states transparent, the other components get access to the state vectors using meth-

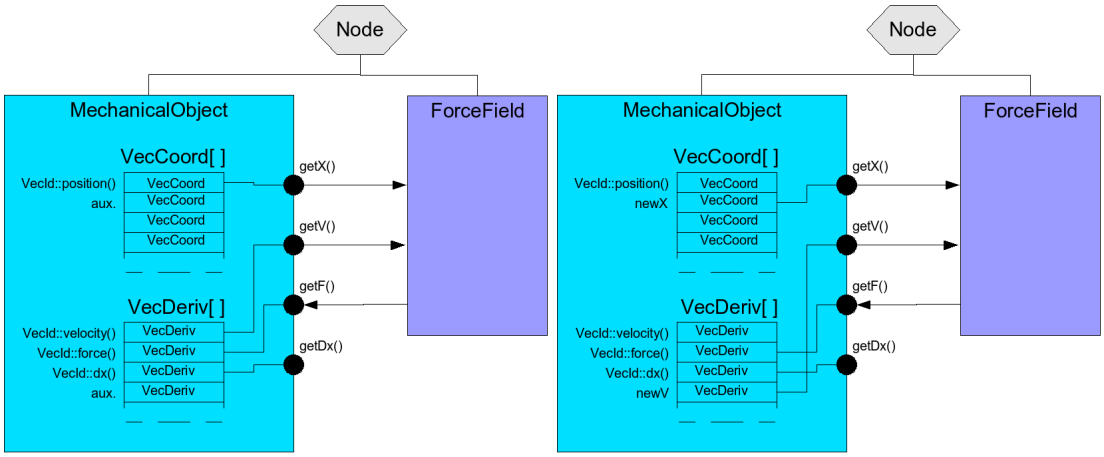

Figure 1.7: A `MechanicalObject` and a component addressing it. Left: using the default state vectors. Right: using auxiliary state vectors.

ods `MechanicalObject::getX()`, `getV()`, `getF()` and `getDx()`. These methods return pointers to the appropriate vectors, as illustrated in figure 1.7.

Internal `MechanicalObject` switches are performed by methods `MechanicalObject::setX()`, `setV()`, `setF()` and `setDx()`. These methods are applied by the visitors which take multivectors as parameters, before they use other components. See, for example, method `MechanicalPropagatePositionAndVelocityVisitor::fwdMechanicalState`.

Note that some constraint-based animation methods require large state vectors and matrices encompassing all the mechanical objects of the scene. Such methods are currently under development in SOFA , and they are not yet documented. They use visitors to count the total number of scalar DOFs and to gather them in large state vectors, as well as to build mechanical matrices such as mass, stiffness, damping and compliance etc.

1.5.1 Mechanical groups

During the simulation, each solver prunes the `AnimateVisitor` which traverses it and manages its associated subtree by itself using other visitors. The objects animated by a given solver are called a *mechanical group*. Each mechanical group corresponds to a subtree in the scene graph. In the example discussed in section 1.2, there is one mechanical group because a single solver located near the root manages the whole scene. However, using separate solvers for different objects can sometimes increase efficiency. In the example shown in figure 1.8, the same deformable body is animated using a `RungeKutta2Solver` while the rigid body is animated using an `EulerSolver`.

A mechanical group can include interaction forces between elements of the group, and such interaction forces are handled by the solver as expected. Interaction forces can also occur between objects which do not belong to the same group. In this case, the interaction force is located at a higher hierarchical level than the objects it applies to, as shown in figure 1.8. It can not be traversed by visitors fired by the solvers. Its evaluation is performed by the `AnimateVisitor`, and accumulated as external forces in the associated `MechanicalObject` components. Consequently, it acts as a constant constant force during each whole animation step. In a `RungeKutta2Solver`, during the force computation at midpoint, its value is the same as at the starting point. In a `CGImplicitSolver`, its stiffness is not taken into account, which may introduce instabilities if its actual stiffness is high.

The default collision manager of Sofa circumvents this problem by dynamically gathering the objects in contact in a common mechanical group.

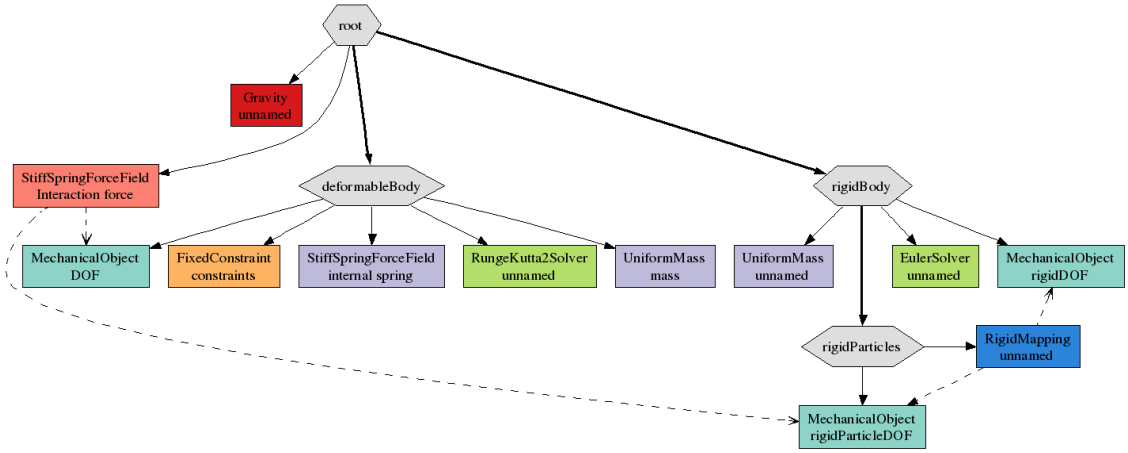

Figure 1.8: A scene graph with objects animated using different ODE solvers.

1.6 Code of the examples

1.6.1 The hybrid pendulum

This is the code of the example commented in section 1.2 :

```
/******************************************************************************
*       SOFA, Simulation Open-Framework Architecture, version 1.0 beta 3      *
*                (c) 2006-2008 MGH, INRIA, USTL, UJF, CNRS                    *
*                                                                             *
* This program is free software; you can redistribute it and/or modify it     *
* under the terms of the GNU General Public License as published by the Free  *
* Software Foundation; either version 2 of the License, or (at your option)   *
* any later version.                                                          *
*                                                                             *
* This program is distributed in the hope that it will be useful, but WITHOUT *
* ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or       *
* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for    *
* more details.                                                               *
*                                                                             *
* You should have received a copy of the GNU General Public License along     *
* with this program; if not, write to the Free Software Foundation, Inc., 51  *
* Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.                   *
*******************************************************************************
*                            SOFA :: Applications                             *
*                                                                             *
* Authors: M. Adam, J. Allard, B. Andre, P-J. Bensoussan, S. Cotin, C. Duriez,*
* H. Delingette, F. Falipou, F. Faure, S. Fonteneau, L. Heigeas, C. Mendoza,  *
* M. Nesme, P. Neumann, J-P. de la Plata Alcade, F. Poyer and F. Roy          *
*                                                                             *
* Contact information: contact@sofa-framework.org                             *
******************************************************************************/
/** A sample program. Laure Heigeas, Francois Faure, 2007. */
// scene data structure
#include <sofa/simulation/tree/Simulation.h>
#include <sofa/component/contextobject/Gravity.h>
#include <sofa/component/odesolver/CGImplicitSolver.h>
#include <sofa/component/odesolver/EulerSolver.h>
#include <sofa/component/odesolver/StaticSolver.h>
#include <sofa/component/visualmodel/OglModel.h>
// gui
#include <sofa/gui/SofaGUI.h>

#include <sofa/component/typedef/Sofa_typedef.h>

using namespace sofa::simulation::tree;
using sofa::component::odesolver::EulerSolver;

int main(int, char** argv)
{
    sofa::gui::SofaGUI::Init(argv[0]);
    //============================ Build the scene
    double endPos = 1.;
    double attach = -1.;
    double splength = 1.;

    //---------------------- The graph root node
    GNode* groot = new GNode;
    groot->setName( "root" );
    groot->setGravityInWorld( Coord3(0,-10,0) );

    // One solver for all the graph
    EulerSolver* solver = new EulerSolver;
    groot->addObject(solver);
    solver->setName("S");

    //---------------------- Deformable body
    GNode* deformableBody = new GNode;
    groot->addChild(deformableBody);
```

```
    deformableBody->setName( "deformableBody" );

    // degrees of freedom
    MechanicalObject3* DOF = new MechanicalObject3;
    deformableBody->addObject(DOF);
    DOF->resize(2);
    DOF->setName("Dof1");
    VecCoord3& x = *DOF->getX();
    x[0] = Coord3(0,0,0);
    x[1] = Coord3(endPos,0,0);

    // mass
    //     ParticleMasses* mass = new ParticleMasses;
    UniformMass3* mass = new UniformMass3;
    deformableBody->addObject(mass);
    mass->setMass(1);
    mass->setName("M1");

    // Fixed point
    FixedConstraint3* constraints = new FixedConstraint3;
    deformableBody->addObject(constraints);
    constraints->setName("C");
    constraints->addConstraint(0);


    // force field
    StiffSpringForceField3* spring = new StiffSpringForceField3;
    deformableBody->addObject(spring);
    spring->setName("F1");
    spring->addSpring( 1,0, 10., 1, splength );


    //---------------------- Rigid body
    GNode* rigidBody = new GNode;
    groot->addChild(rigidBody);
    rigidBody->setName( "rigidBody" );

    // degrees of freedom
    MechanicalObjectRigid3* rigidDOF = new MechanicalObjectRigid3;
    rigidBody->addObject(rigidDOF);
    rigidDOF->resize(1);
    rigidDOF->setName("Dof2");
    VecCoordRigid3& rigid_x = *rigidDOF->getX();
    rigid_x[0] = CoordRigid3( Coord3(endPos-attach+splength,0,0),
                    Quat3::identity() );

    // mass
    UniformMassRigid3* rigidMass = new UniformMassRigid3;
    rigidBody->addObject(rigidMass);
    rigidMass->setName("M2");


    //---------------------- the particles attached to the rigid body
    GNode* rigidParticles = new GNode;
    rigidParticles->setName( "rigidParticles" );
    rigidBody->addChild(rigidParticles);

    // degrees of freedom of the skin
    MechanicalObject3* rigidParticleDOF = new MechanicalObject3;
    rigidParticles->addObject(rigidParticleDOF);
    rigidParticleDOF->resize(1);
    rigidParticleDOF->setName("Dof3");
    VecCoord3& rp_x = *rigidParticleDOF->getX();
    rp_x[0] = Coord3(attach,0,0);

    // mapping from the rigid body DOF to the skin DOF, to rigidly attach the skin to the body
    RigidMechanicalMappingRigid3_to_3* rigidMapping = new
          RigidMechanicalMappingRigid3_to_3(rigidDOF,rigidParticleDOF);
    rigidParticles->addObject( rigidMapping );
    rigidMapping->setName("Map23");


    // ------------------ Interaction force between the deformable and the rigid body
    StiffSpringForceField3* iff = new StiffSpringForceField3( DOF, rigidParticleDOF );
    groot->addObject(iff);
    iff->setName("F13");
    iff->addSpring( 1,0, 10., 1., splength );




    //============================ Init the scene
    getSimulation()->init(groot);
    groot->setAnimate(false);
    groot->setShowNormals(false);
    groot->setShowInteractionForceFields(true);
    groot->setShowMechanicalMappings(true);
    groot->setShowCollisionModels(false);
    groot->setShowBoundingCollisionModels(false);
    groot->setShowMappings(false);
    groot->setShowForceFields(true);
    groot->setShowWireFrame(false);
    groot->setShowVisualModels(true);
    groot->setShowBehaviorModels(true);



    //============================ Run the main loop

    sofa::gui::SofaGUI::MainLoop(groot);
}
```

1.6.2 A liver

The XML code of the liver discussed in section 1.3 page 5 is in `../scenes/liver.scn`

Chapter 2

Design

2.1 Mappings

François Faure

2.1.1 Geometrical layers

Different geometrical models are used to model objects in contact. We organize them in a hierarchy of layers. An example is shown in figure 2.1, where a rigid object hits a shape embedded in deformable cells.

The state of a simulated system can be described by the values and time derivatives of its independent degrees of freedom (DOF) gathered in two vectors x_0 and v_0. The dynamics equation (Newton's law) relates the second time derivative a_0 of the DOF to the forces f_0 acting on them: $f_0 = Ma_0$, where M is a matrix modeling the mass of the system.

Geometry can be attached to the DOF for visualization or contact computation. We call it the *shape*. It is typically defined by points, such as triangle vertices or sphere centers, and additional data such as triangle connectivity or sphere radii. We call these points the *vertices*. Their positions, velocities and associated forces are stored in vectors x_1, v_1 and f_1, respectively. They are not independent variables, since the positions and velocities are bound to the DOF using kinematic operators which we call the *mappings*:

$$\begin{aligned} x_1 &= \mathcal{J}_1(x_0) \\ v_1 &= J_1 v_0 \end{aligned}$$

When the vertices and the DOF are the same, the mapping is the identity. This special case occurs for instance when we simulate cloth or rigid balls. More general cases include polygonal shapes attached to rigid bodies using local coordinates, or embedded in deformable cells using barycentric coordinates, as well as skin surrounding articulated bodies using vertex blending techniques. Matrix $J_1 = \frac{\partial x_1}{\partial x_0}$ encodes the linear relation between the DOF velocities and the shape velocities. Due to linearity, the same relation holds on elementary displacements dx. It also holds on accelerations, with an additional offset due to velocities when the position mapping $\mathcal{J}$is nonlinear. In most cases, operators $\mathcal{J}$ and J are the same, but in the case of rigid bodies, $\mathcal{J}$ is nonlinear with respect to x and it can not be written as a matrix. For surfaces embedded in deformable cells, matrix J contains the barycentric coordinates. For surfaces attached to rigid bodies, each row of the matrix encodes the usual relation $v = \dot{o} + \omega \times (x - o)$ for each vertex. Similarly, skins around articulated bodies involve, at each vertex, the weighted contributions of the rigid bodies.

When shapes collide, additional geometry can be necessary to model the contact. For instance, when an edge intersects another one, a contact force is applied to the intersection points. These points are defined by their barycentric coordinates with respect to their edge vertices. Other relations can be used, depending on the king of geometrical primitives in contact. This additional geometry requires another geometrical layer connected to the shape by a mapping, as illustrated in figure 2.1.

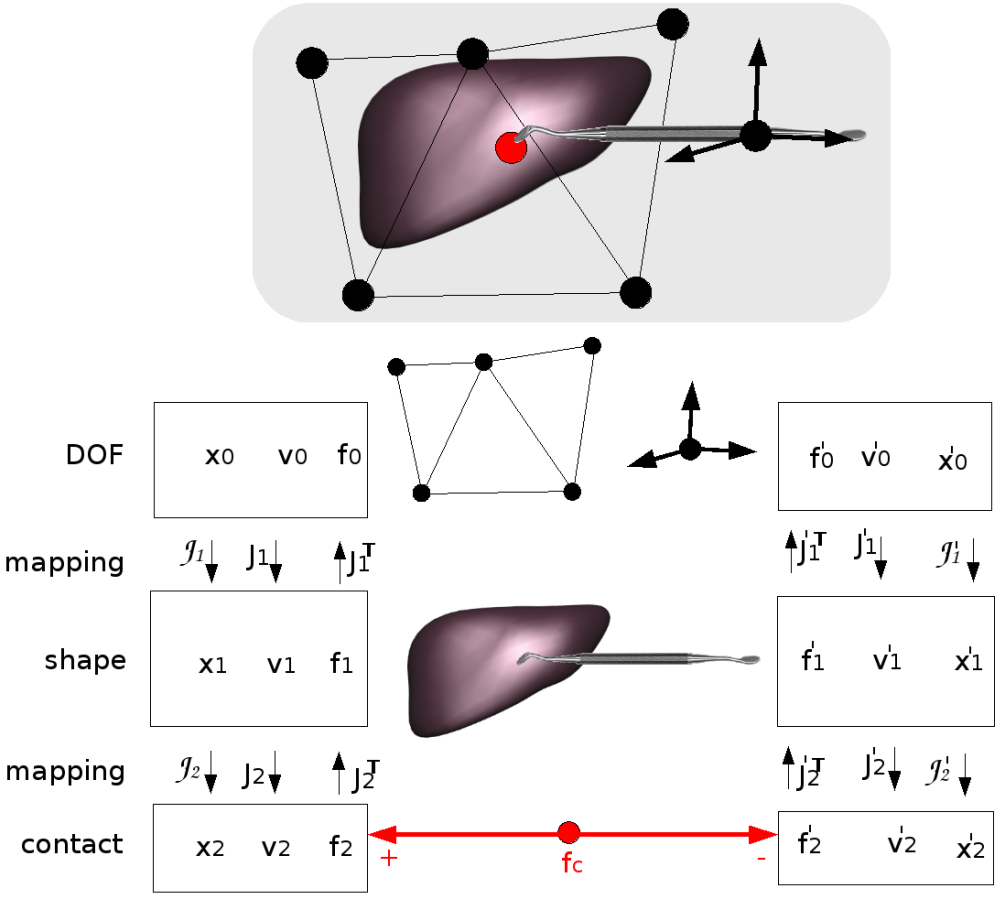

Figure 2.1: Mappings from the DOFs to a contact point. Top: two simulated objects in contact (red point). Bottom: hierarchy of geometrical layers. Positions and velocities are propagated top-down. The contact force f_c is accumulated in the contact layers. Forces are then propagated bottom-up.

We can straightforwardly extend this approach to tree-like hierarchies of geometries, with the DOF layer at the root. For instance, the DOF layer may have two independent children, one for collision using a coarse mesh, and the other for rendering using a finer mesh. The synchronization between these sibling layers is automatically guaranteed by their attachment to their common ancestor, the DOF layer. The hierarchy can also be deeper, using for instance a fine mesh for rendering and its convex hull for collision. This flexible framework gives us a great freedom for physical modeling.

2.1.2 Mechanical mappings

Positions and velocities can be propagated top-down through our layer hierarchy using the relations presented in the previous section. In order to take the contact forces into account in the dynamics equation, we have to convert the contact forces applied to the contact points to forces applied to the DOF, where Newton's law is applied. This requires an extension to the position and velocity amppings presented in section 2.1.1. We call it *mechanical mapping*.

We derive a new, general method to propagate forces bottom-up through the layers of the geometrical hierarchy. Given forces f_n applied to a geometry layer n, we derive the equivalent force applied to its parent layer $n-1$. Equivalent forces must have the same power. Thus, we have to compute f_{n-1} such that:

$$v_{n-1}^T f_{n-1} = v_n^T f_n$$

The relation $v_n = J_n v_{n-1}$ allows us to rewrite the previous equation as

$$v_{n-1}^T f_{n-1} = v_{n-1}^T J_n^T f_n$$

Since this relation must hold for any velocity v_{n-1}, we simplify it and get

$$f_{n-1} = J_n^T f_n \tag{2.1}$$

This corresponds to the principle of virtual work.

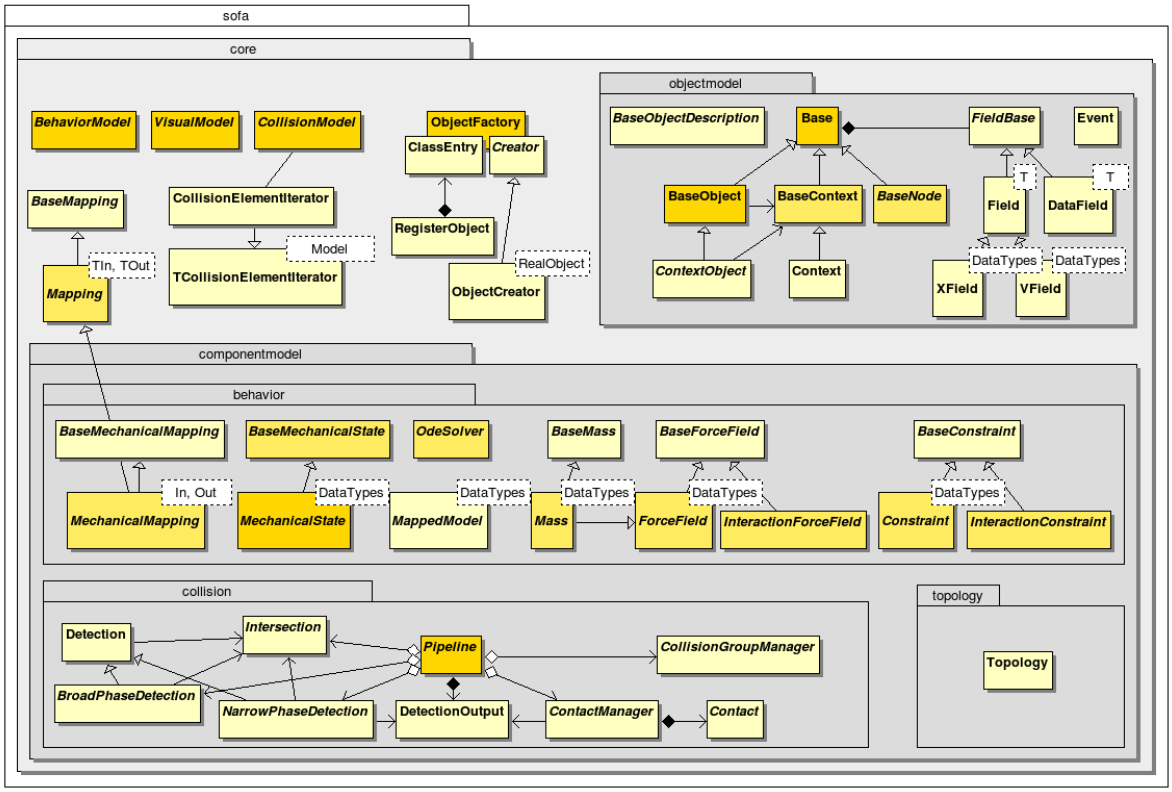

Figure 2.2: Classes of the sofa::core namespace.

2.2 Framework

2.2.1 UML Diagrams

Chapter 3

Modules

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

3.1 Collision Models

3.1.1 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 exemple 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.

3.2 Soft Articulations

3.2.1 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.

3.2.2 Realization

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

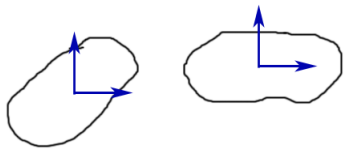

Figure 3.1: two bones

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

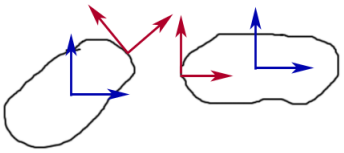

Figure 3.2: two bones (blue) with their articulation frames (red)

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*

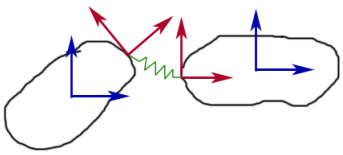

Figure 3.3: two bones linked by a joint-spring

3.2.3 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 softArticulations.scn shows a basic pendulum :

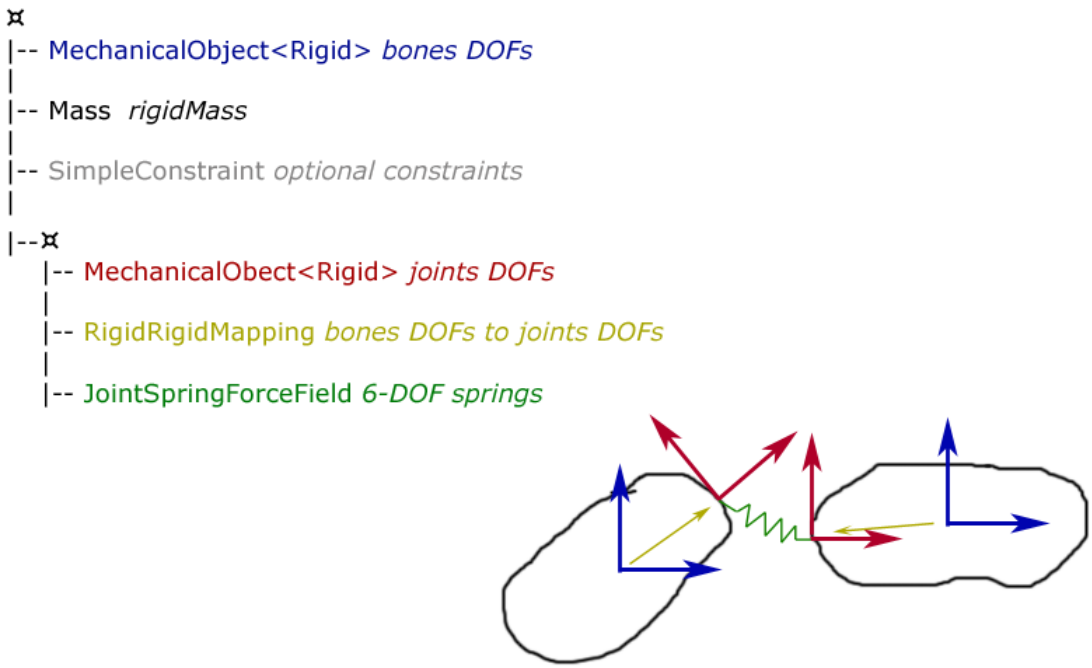

Figure 3.4: a simple articulated body scene

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

  <Node>
    <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>
```

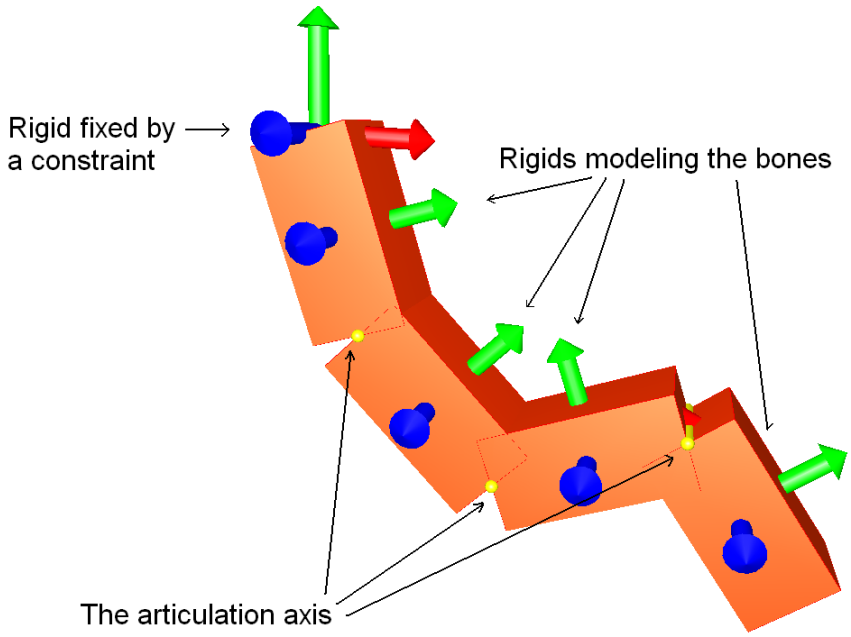

Figure 3.5: The pendulum is composed by 4 rigids linked one by one by articulations

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.

3.2.4 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.

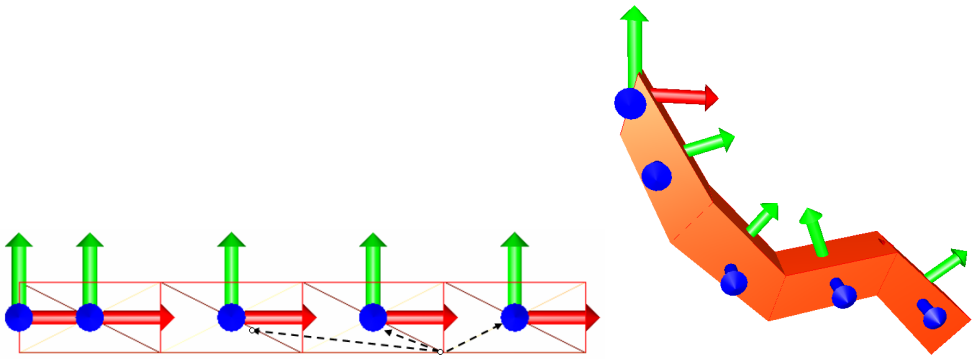

Figure 3.6: In the example "softArticulationsSkinned.scn" the external points compute their skinning weights from the 3 nearest DOFs

Figure 3.7: soft articulations coupled with skinning allow complexe model deformations

3.3 How to use mesh topologies in SOFA

H. Delingette, B. André

3.3.1 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 information are required on a computational mesh to perform :

- **Mesh Visualization**,
- **Collision detection** : some collision detection 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.

3.3.2 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. 3.8) 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. 3.8) 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. 3.9). 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.

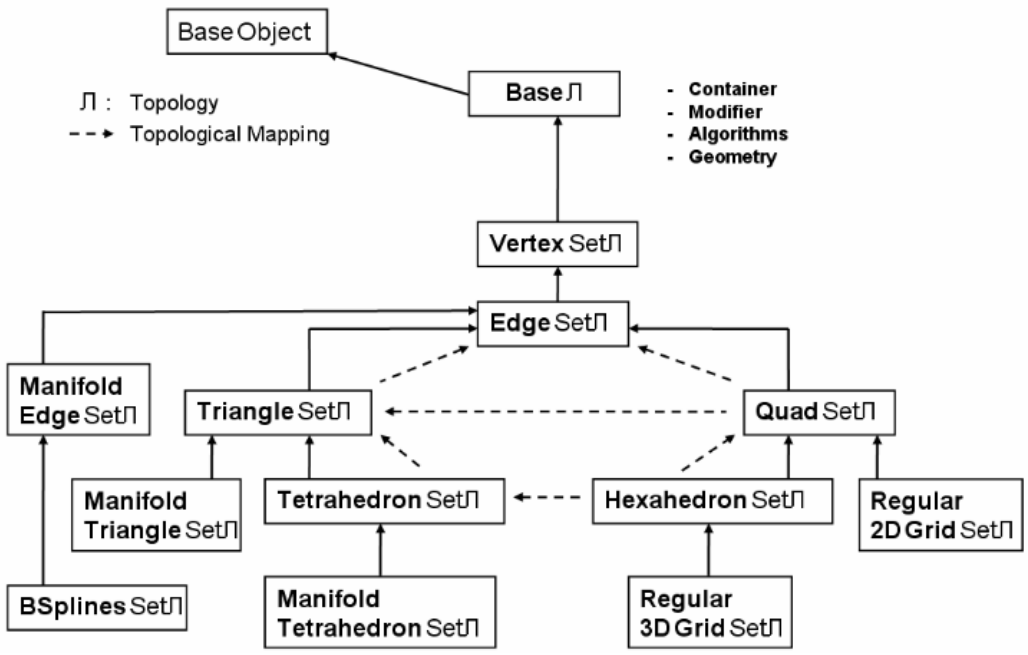

Figure 3.8: Family tree of topology objects. Dashed arrows indicate possible *Topological Mappings* from a topology object to another.

- 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.

3.3.3 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.

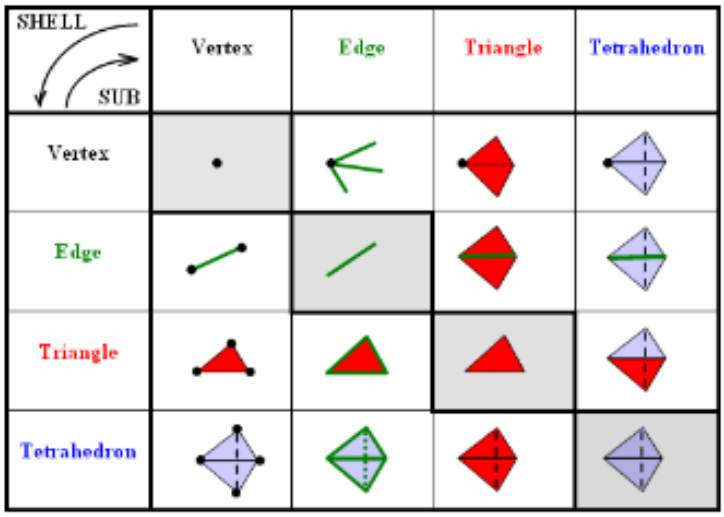

Figure 3.9: The two topological arrays stored in a *Container* correspond to the upper and lower triangular entries of this table. The upper entries provide the k-cells adjacent to a l-cell, $l < k$. The lower entries describe the l-cells included in a k-cell. Similar table exists for quad and hexahedron elements.

3.3.4 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. 3.11). 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. 3.12), 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. 3.10. 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.

3.3.5 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

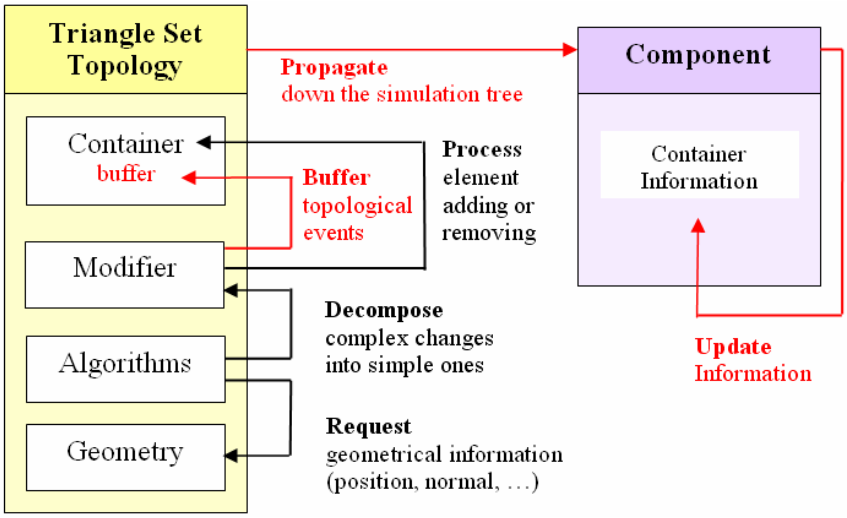

Figure 3.10: Handling topological changes, with the example of a *Triangle Set Topology*. *Component* corresponds to any component of the simulation which may need topological information to perform a specific task. Black features indicate the effective change process. Red features show the steps of 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 3.11: 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

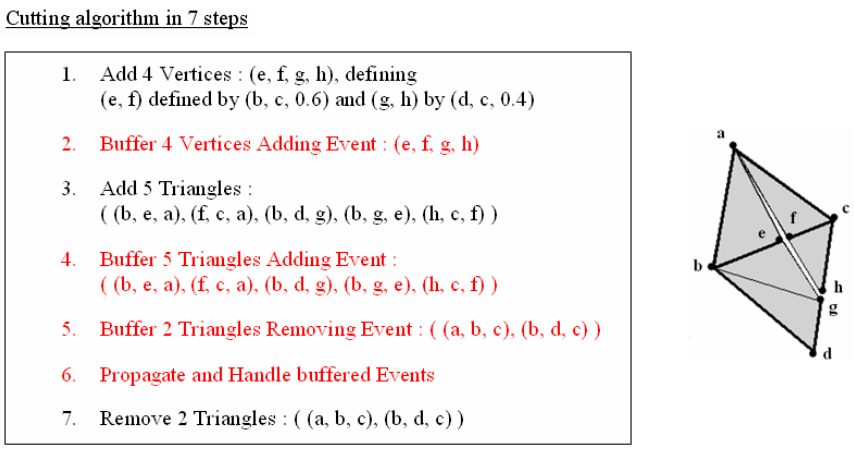

Figure 3.12: Seven steps to perform cutting along two triangles (these generic steps would be the same to cut an arbitrarily number of triangles). Black steps indicate the effective change process. Red steps show the steps of event notification.

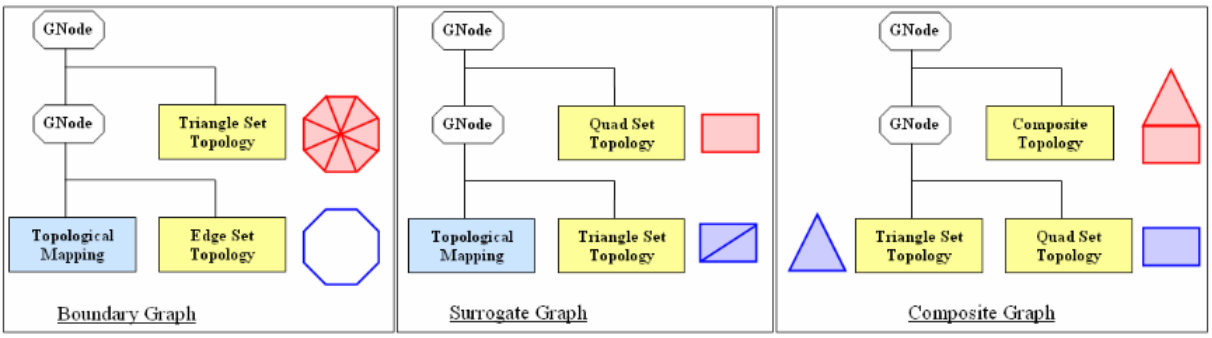

Figure 3.13: Three scenari examples to combine topologies. (*From left to right*) A *Boundary Graph* from a triangle set to an edge set, a *Surrogate Graph* from a quad set to an triangle set and a *Composite Graph* superseding a quad set and a triangle set.

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. 3.14 provides an example of a border scenario (triangulation as the border of a tetrahedralisation) while Fig. 3.13 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. 3.8.

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.

3.3.6 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 :

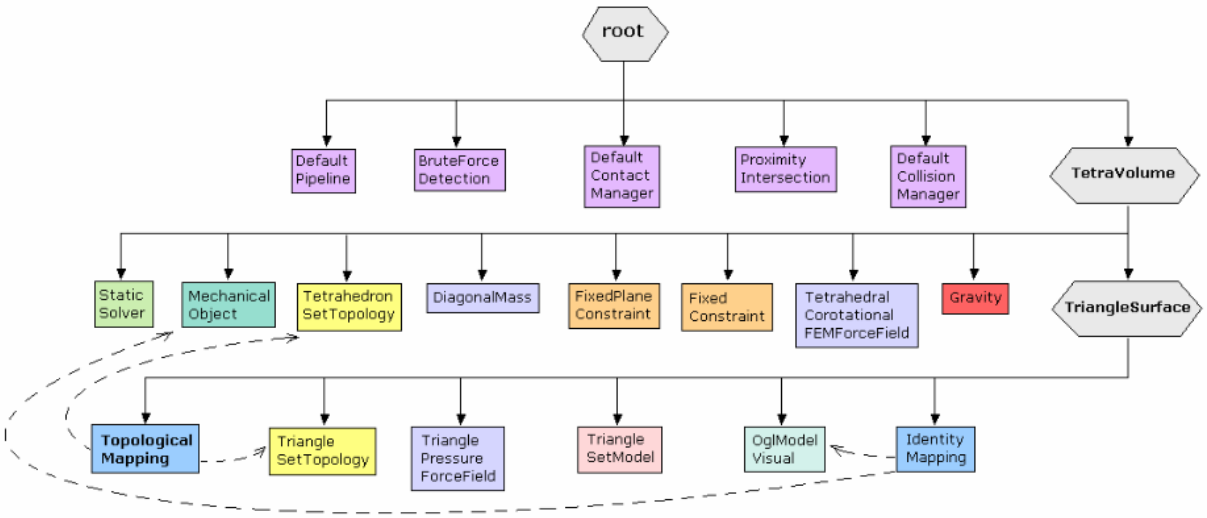

Figure 3.14: Scene Graph illustrating a TopologicalMapping from a TetrahedronSetTopology to a TriangleSetTopology.

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 TetrahedronTriangleShell 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.

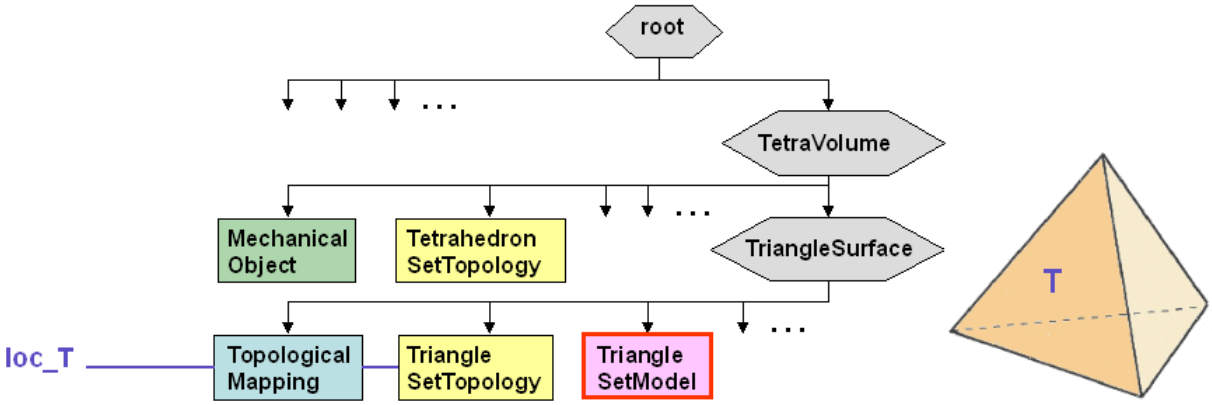

Figure 3.15: Scenario when the user wants to remove one tetraheron.- Step 1.

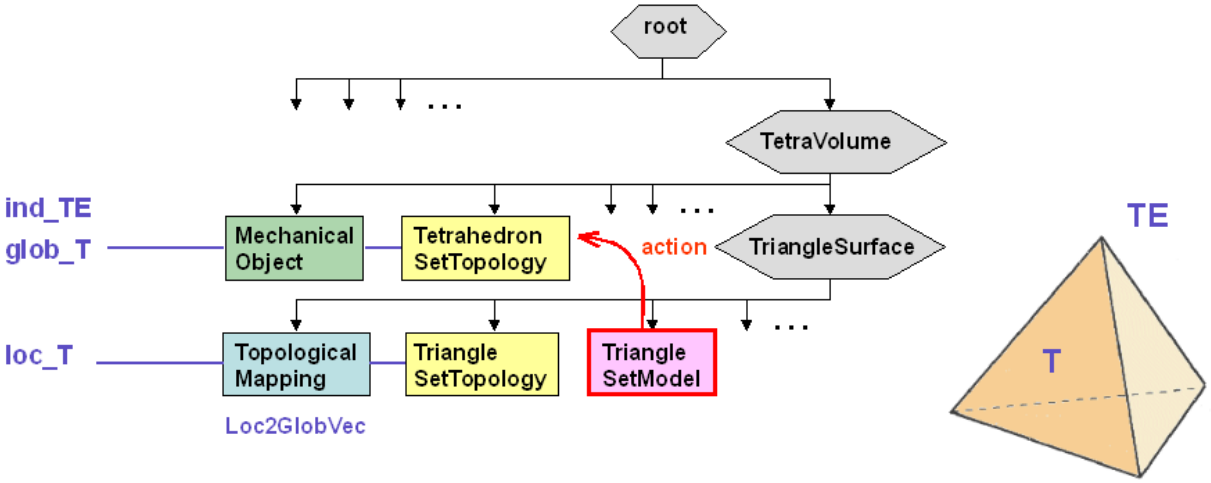

Figure 3.16: Scenario when the user wants to remove one tetraheron.- Step 2.

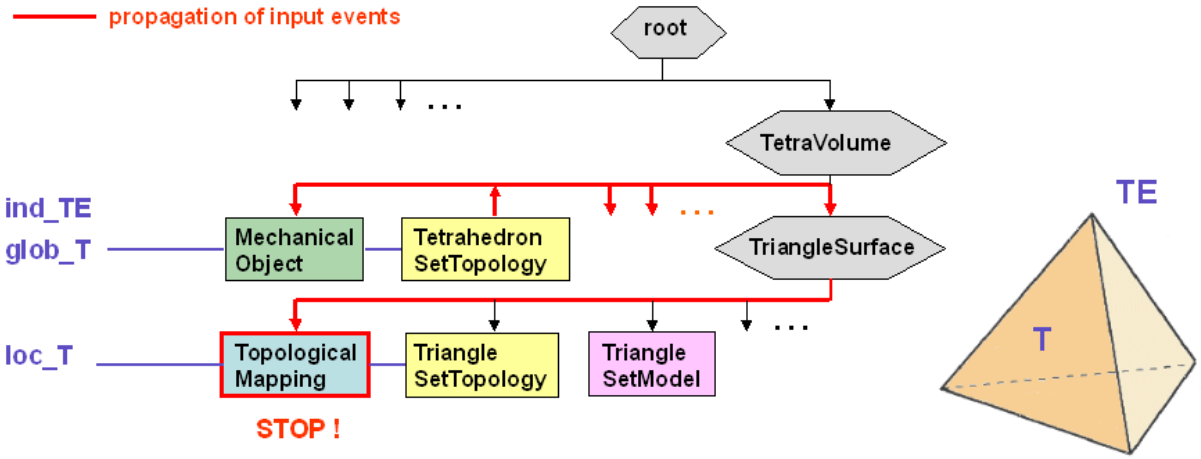

Figure 3.17: Scenario when the user wants to remove one tetraheron.- Step 3.

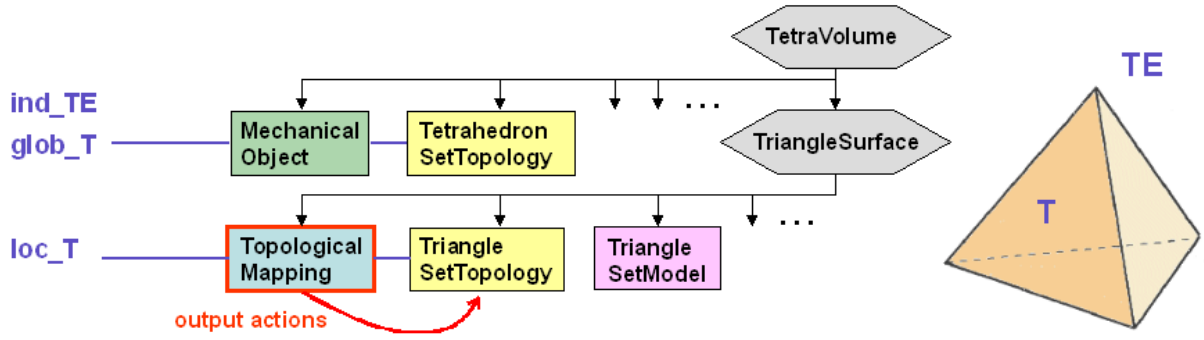

Figure 3.18: Scenario when the user wants to remove one tetraheron.- Step 4.

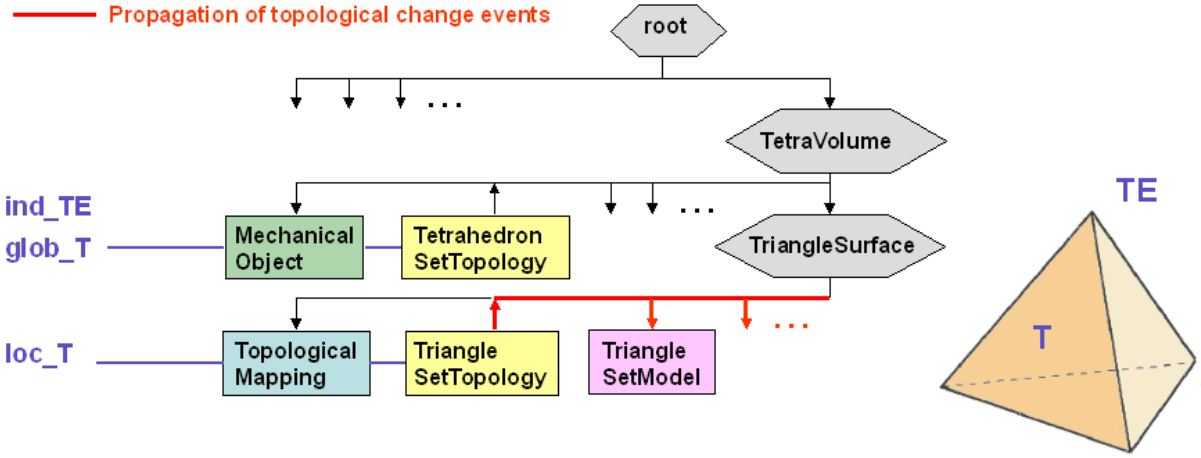

Figure 3.19: Scenario when the user wants to remove one tetraheron.- Step 5.

- **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.

3.3.7 Example of scene file with a topological mapping

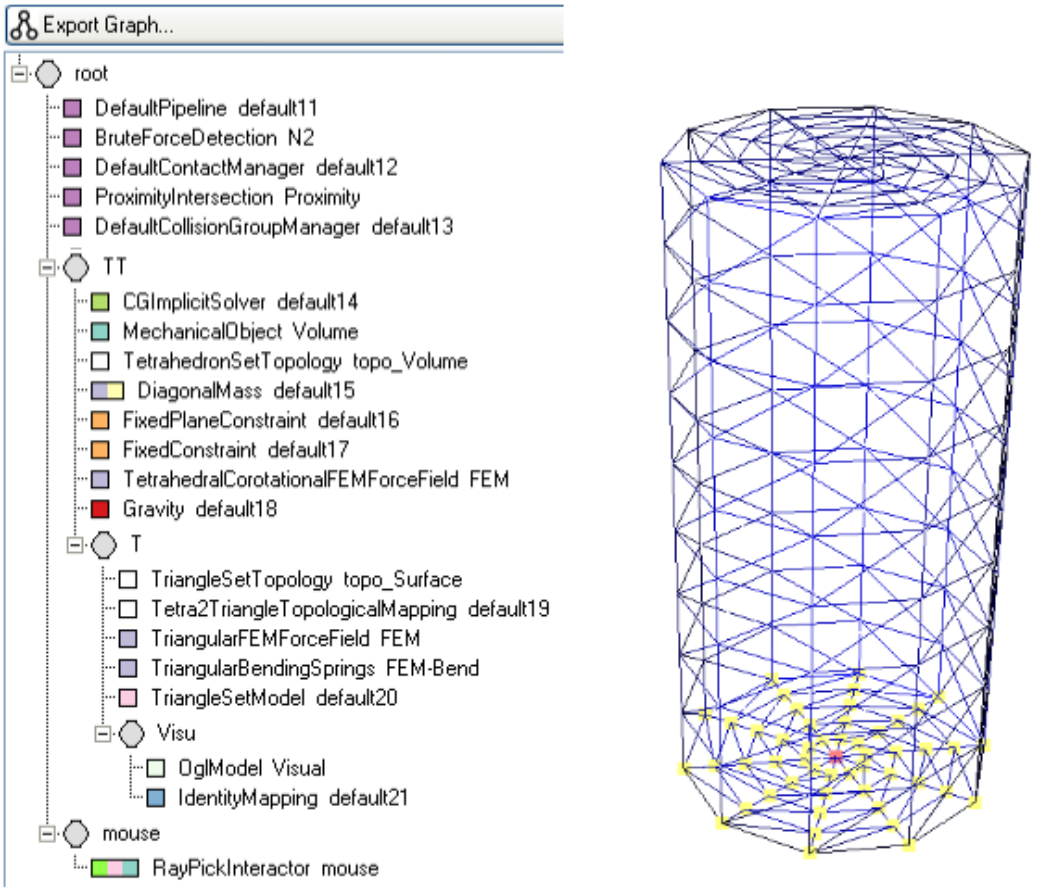

Figure 3.20: Scene Graph simulating a bending cylinder as a tetrahedral volume and as a triangular surface (the cylinder membrane)

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>
```

3.3.8 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 implement 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();
```

3.3.9 What happens when I split an Edge ?

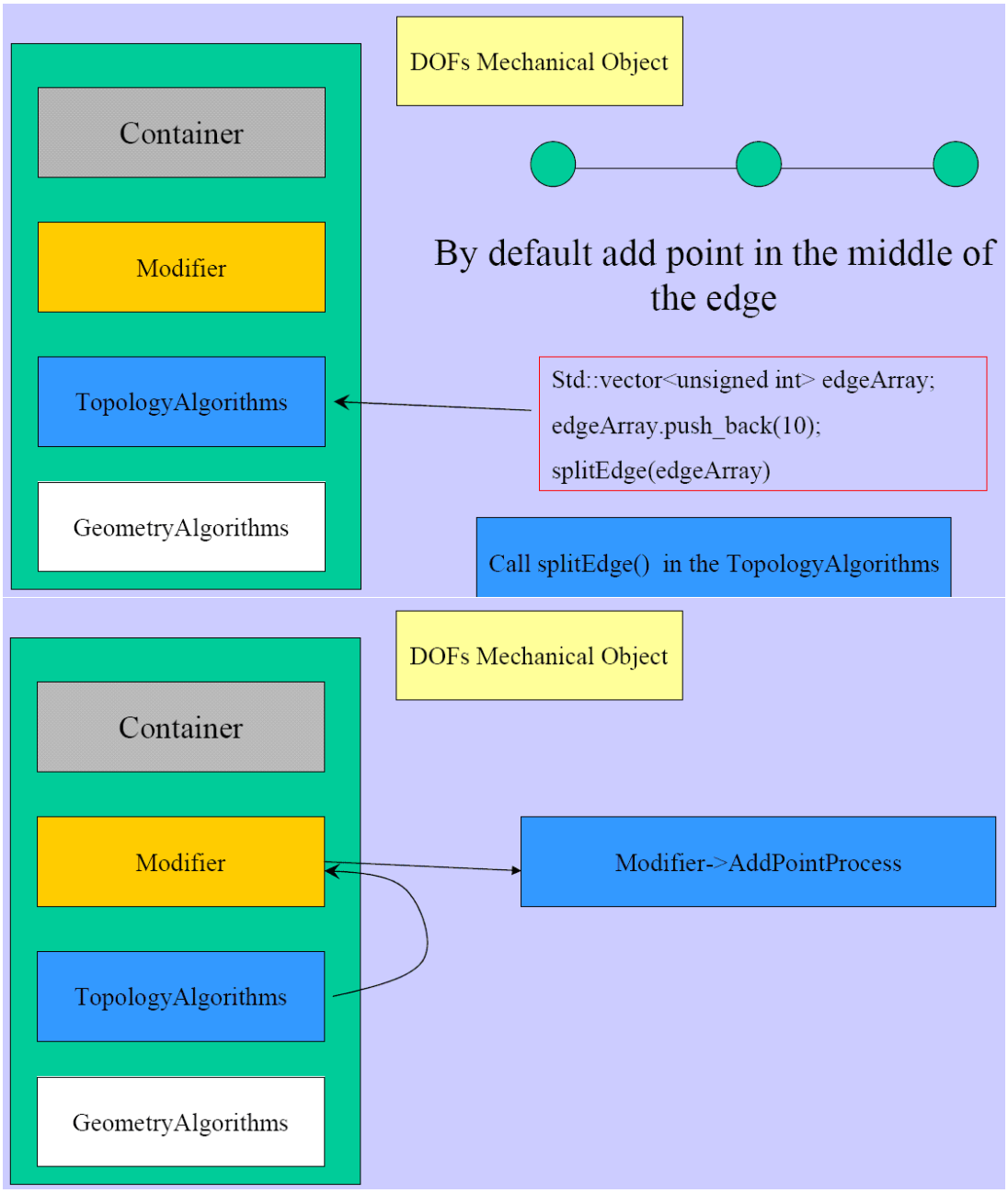

Figure 3.21: What happens when I split an Edge ? - Step 1. 2.

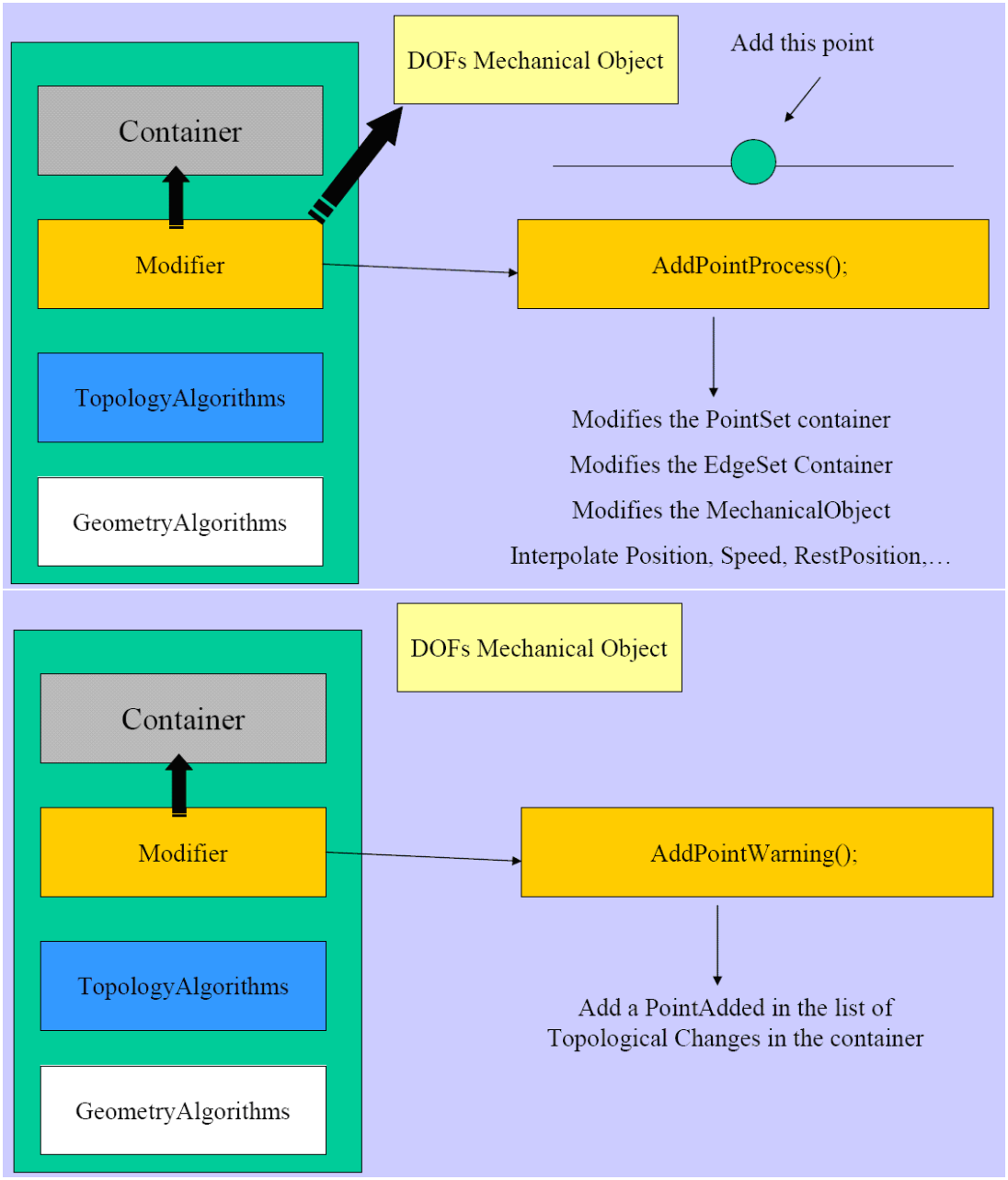

Figure 3.22: What happens when I split an Edge ? - Step 3. 4.

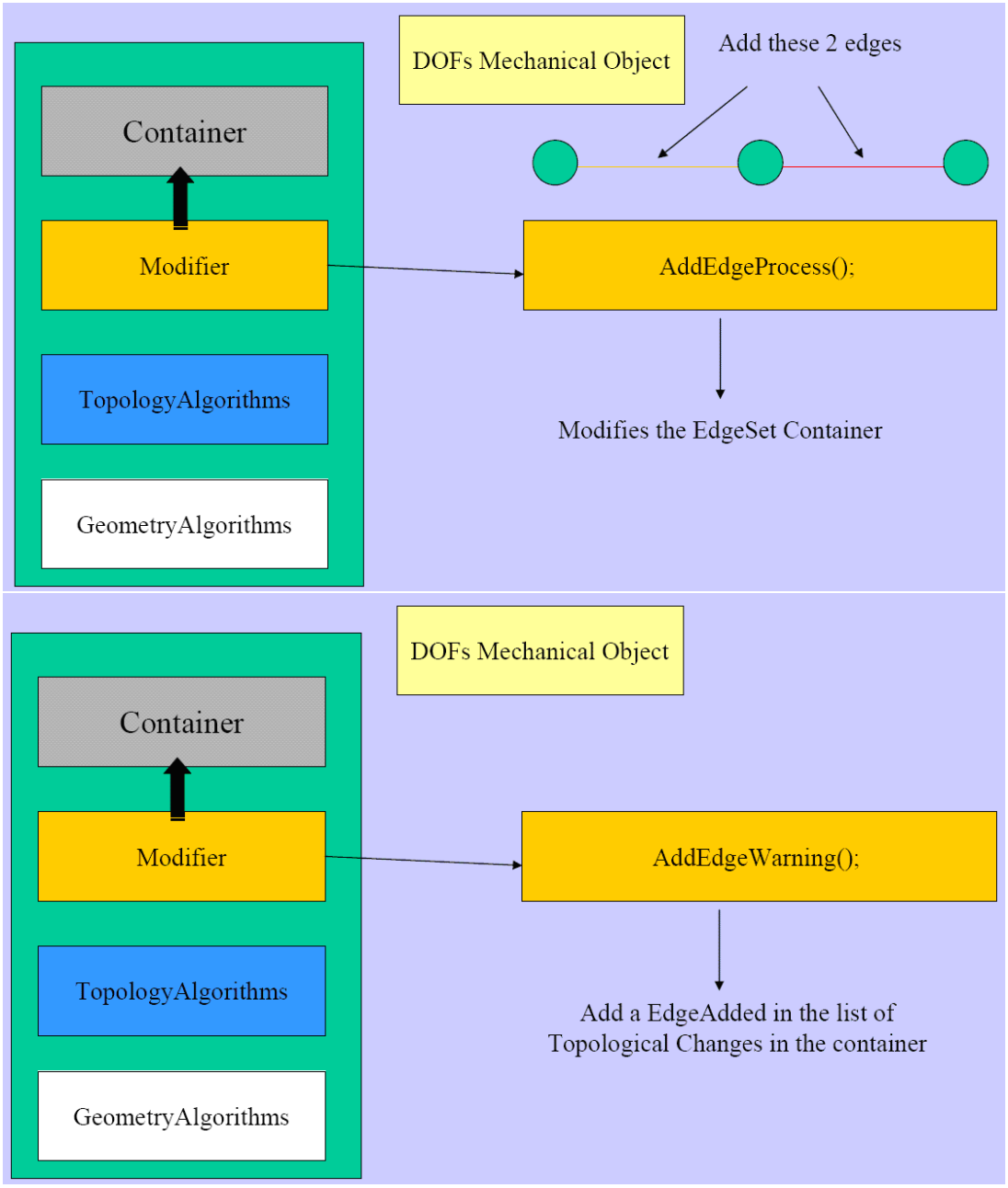

Figure 3.23: What happens when I split an Edge ? - Step 5. 6.

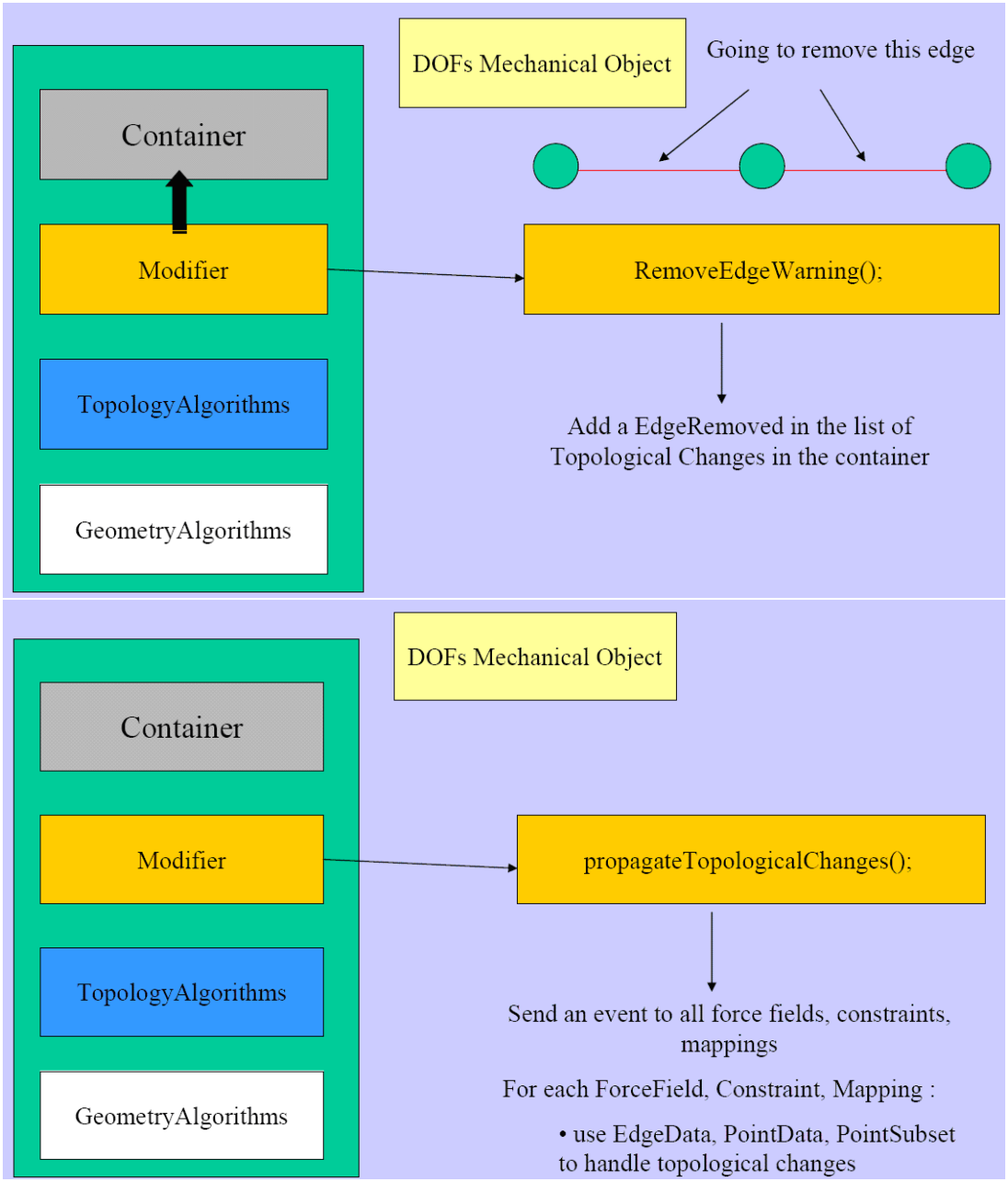

Figure 3.24: What happens when I split an Edge ? - Step 7. 8.

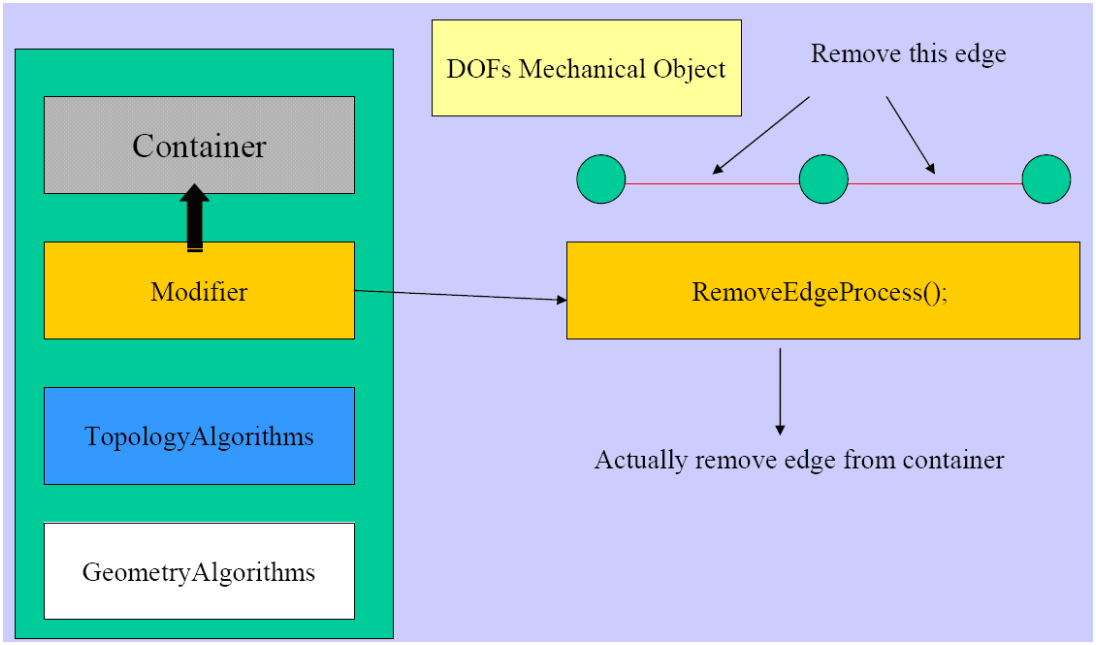

Figure 3.25: What happens when I split an Edge ? - Step 9.

3.4 Graphic User Interface

3.4.1 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.

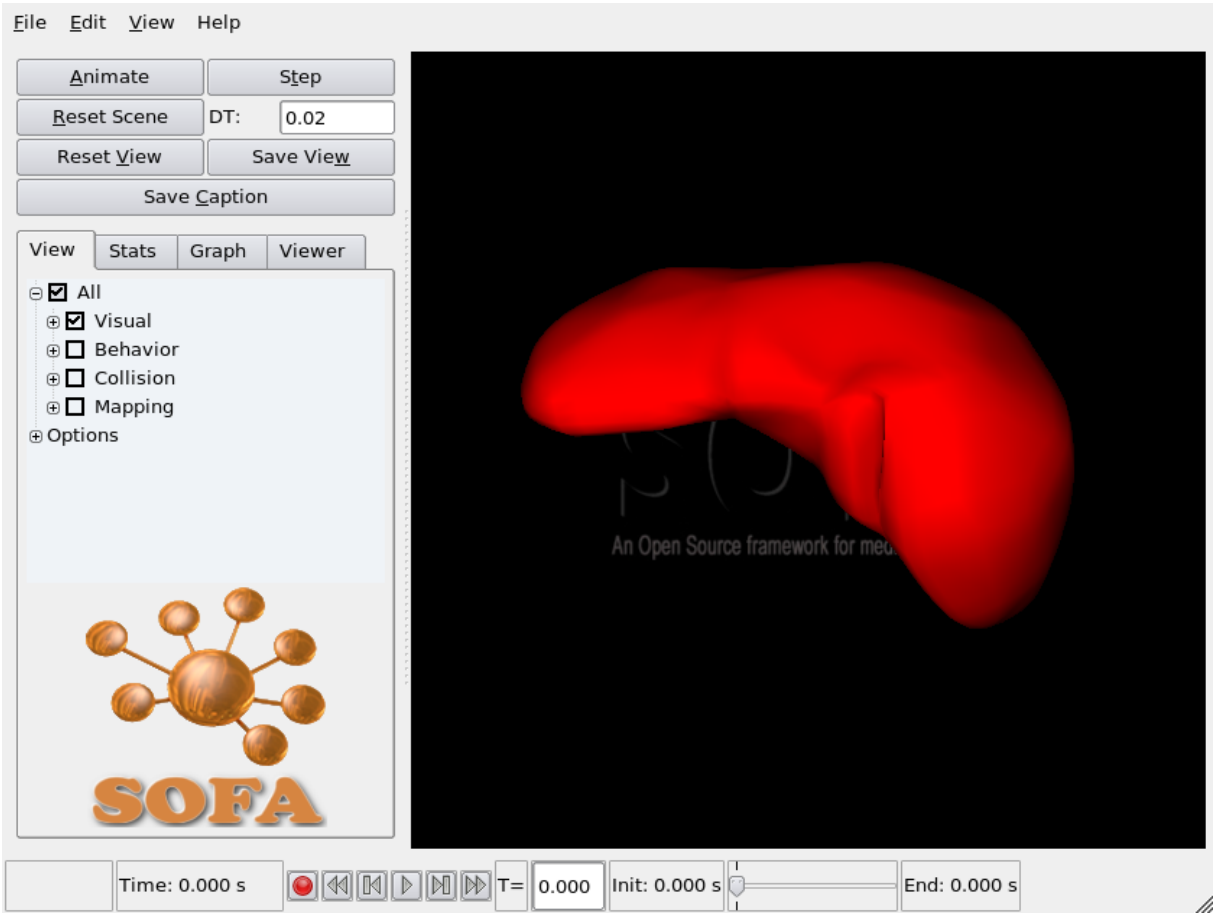

Figure 3.26: SOFA first-time

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

Figure 3.27: 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.

3.4.2 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.

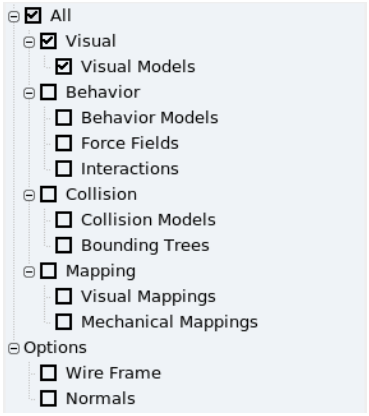

Figure 3.28: Basic Controls

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

3.4.3 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.

3.4.4 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 these datas. This kind of dialog windows give the possibility to modify directly some characteristics of your component. Take care of clicking 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, so that remain visible the other Nodes and the components right below
- **Expand**: Expand the graph, and open all the nodes below the clicked one
- **Desactivate**: Desactivate 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

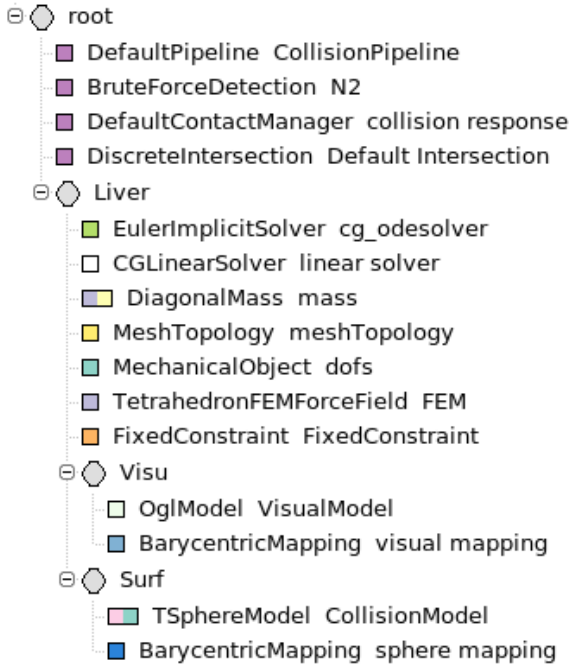

Figure 3.29: Scene Graph for the liver scene

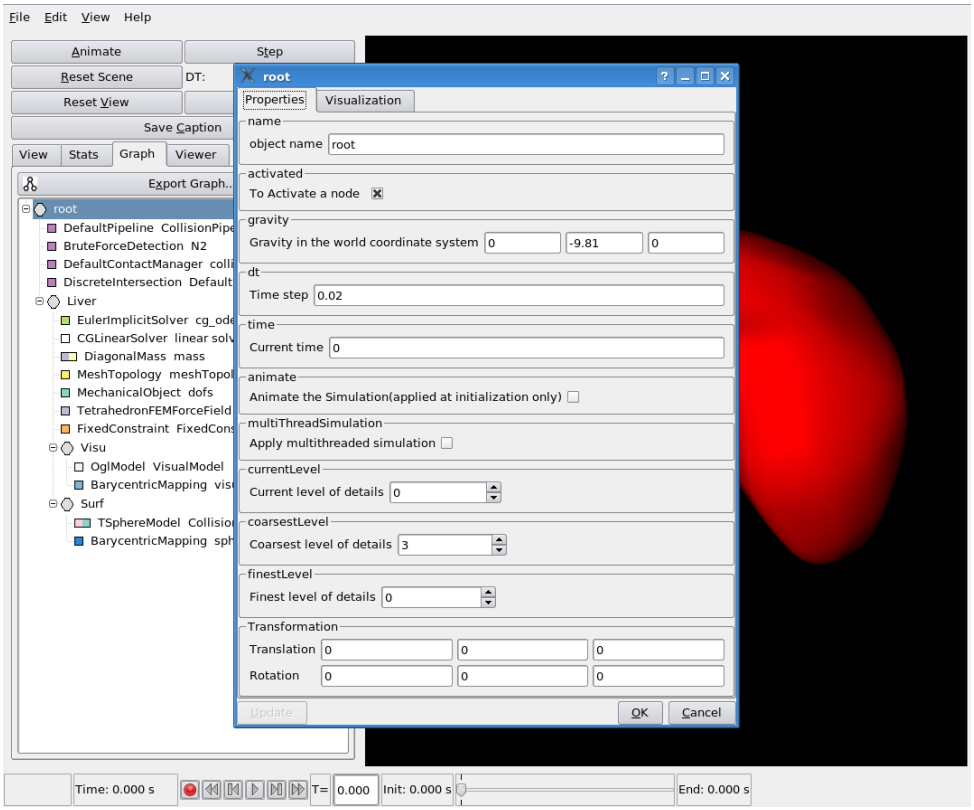

Figure 3.30: Double click on an item of the graph

- **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

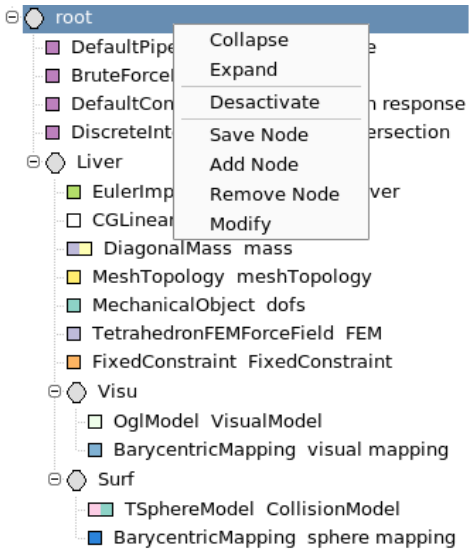

Figure 3.31: Right click on a node of the graph

3.4.5 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.

3.4.6 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.

3.4.7 Architecture

Fig. 3.32 gives an overview of the modular architecture of the GUI for SOFA.

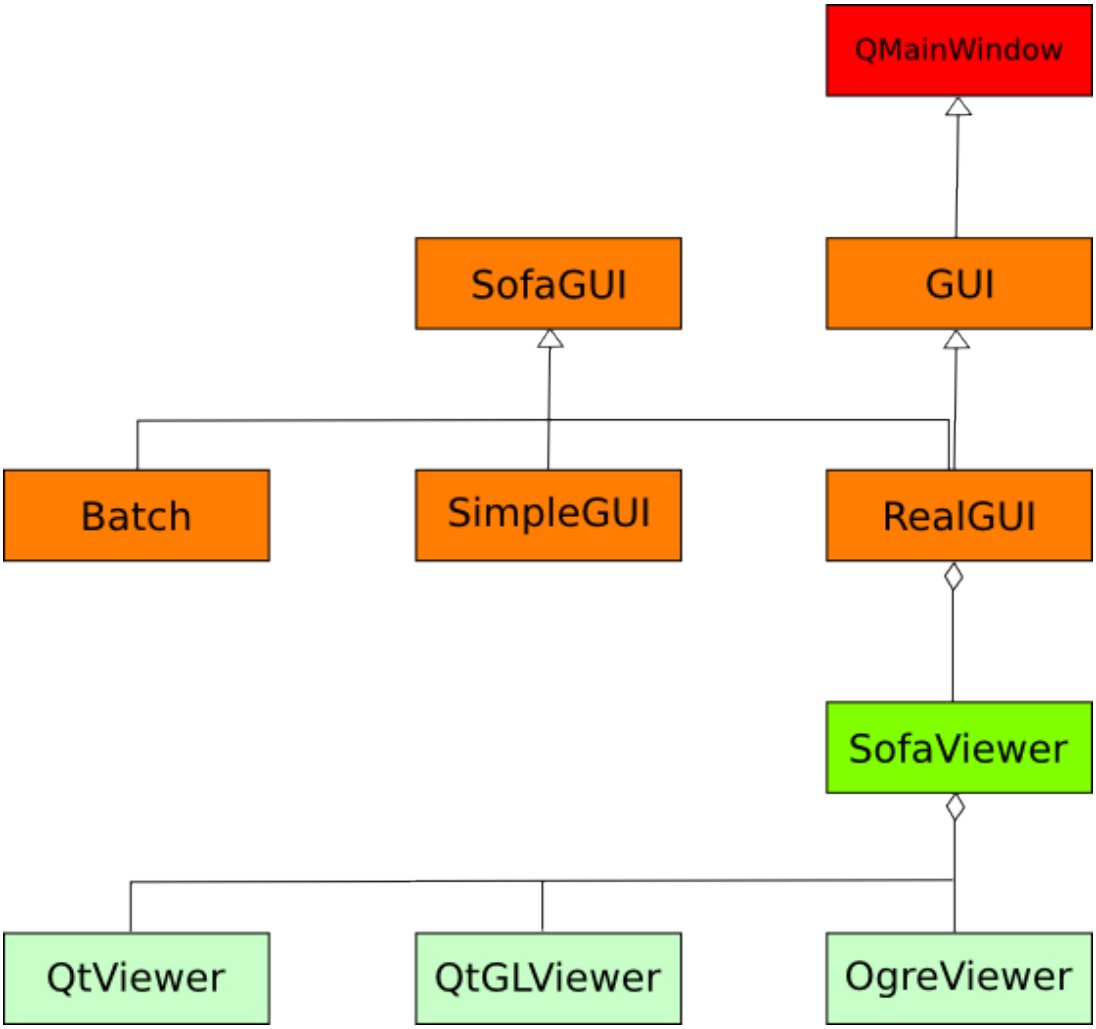

Figure 3.32: A modular Architecture of the GUI

3.4.8 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 functionalities
- **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

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.

3.4.9 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.

3.4.10 Player/Recorder

Figure 3.33: 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 animatethe 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.

3.5 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.

```
<Object type=''LightManager'' />
```

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

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

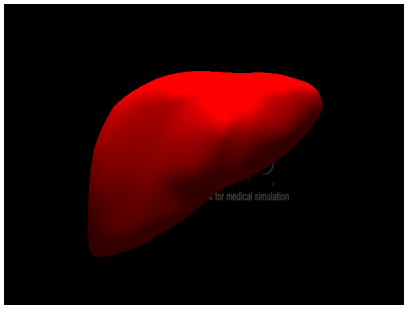

Figure 3.34: Positional Light

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

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

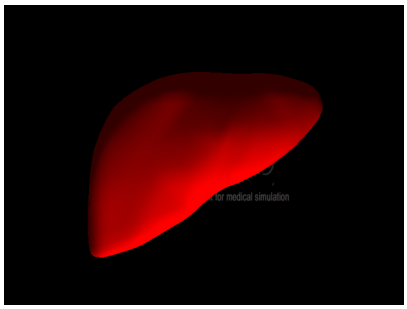

Figure 3.35: Directional Light

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

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

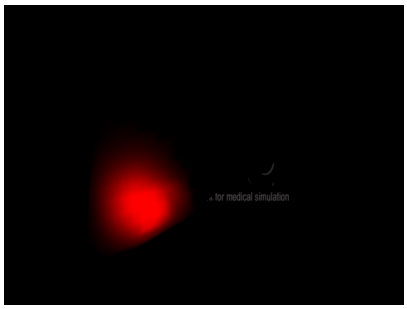

Figure 3.36: Spot Light

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

3.6 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.

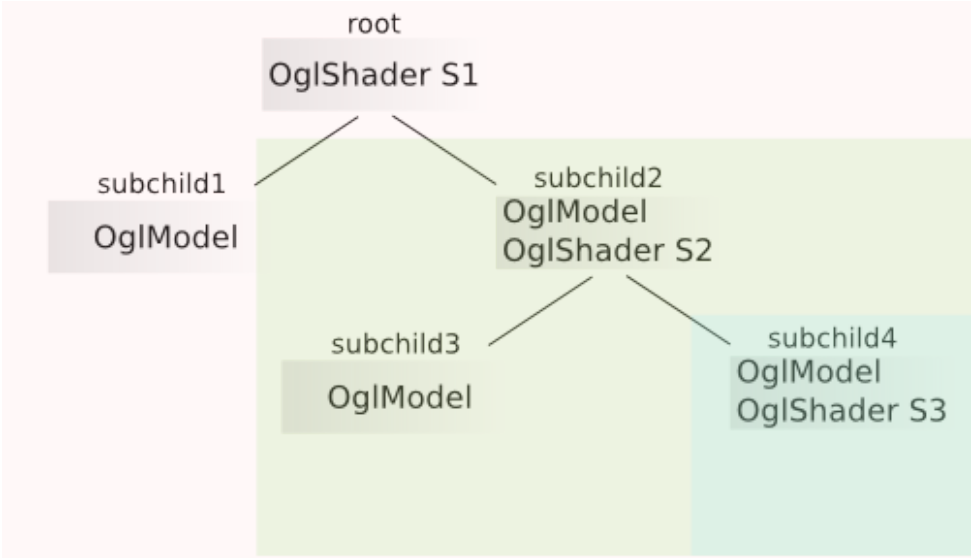

Figure 3.37: Example of shaders' area of effect

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

```
<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 varaibles 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" />
```

Chapter 4

How To

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

4.1 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.

4.1.1 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.

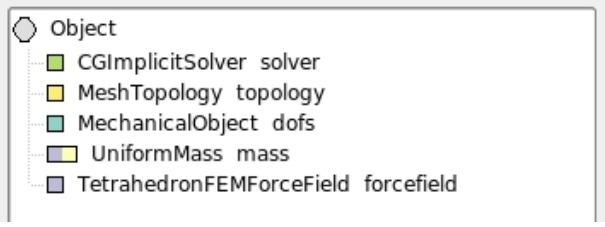

Figure 4.1: Basic example modelling a Finite Element Object

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 writting...
   - **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)

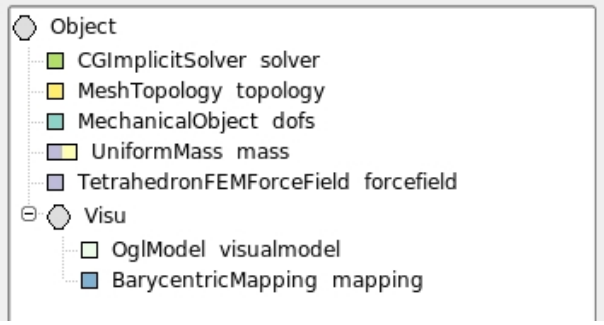

Figure 4.2: Basic example modelling a Finite Element Object with a Visual Model

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.

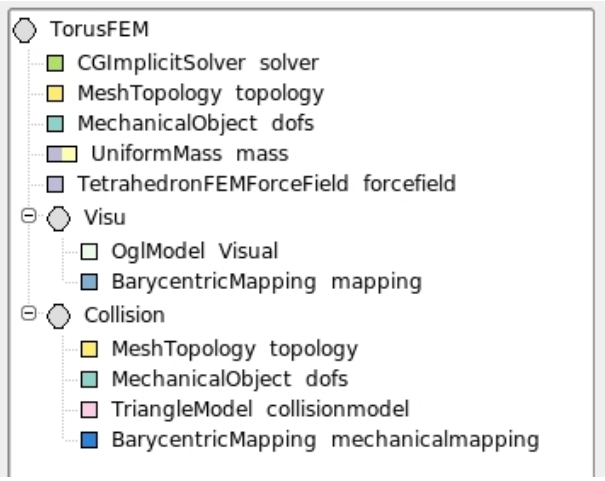

Figure 4.3: Basic example modelling a Finite Element Object with a Visual Model and CollisionModel

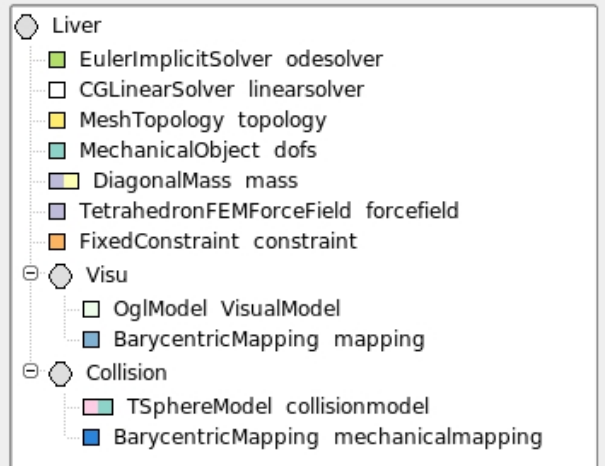

Figure 4.4: Modeling a liver with sphere collision model

4.1.2 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.

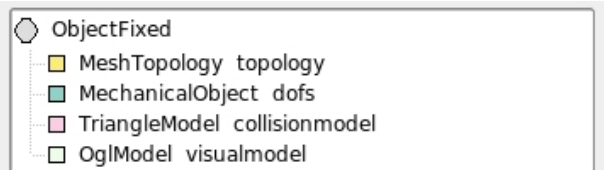

Figure 4.5: Modeling a Fixed object

4.1.3 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

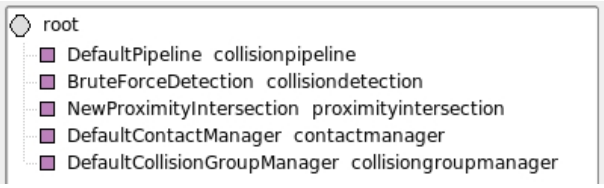

Figure 4.6: Collision Components

4.2 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 fonctions: addForce, getPotentialEnery. Others virtual functions exist like addDForce, addDForceV (if you want to make dynamics), and you should read the doxigen documentation about ForceField.
3. as 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/simulation/common/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.

4.3 How To make your Component modifiable

When you create your own component, it can be very convenient to display some internal datas, 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 one (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 call your class **foo**. You want to control a parameter of type boolean called **verbose**.
initData takes several parameters:

1. address of the Data
2. default value: it must be of the same type as your template(**OPTIONAL**)

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

Figure 4.7: Initialization of the object Data

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**)

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.

Chapter 5

How to contribute to this documentation

5.1 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.

5.2 Compiling the document

5.2.1 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.

5.2.2 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).

5.2.3 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.