# SOFA

The SOFA team

2007

**Abstract**

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

## 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.

### 1.1 Model a dynamic object

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

#### 1.1.1 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.

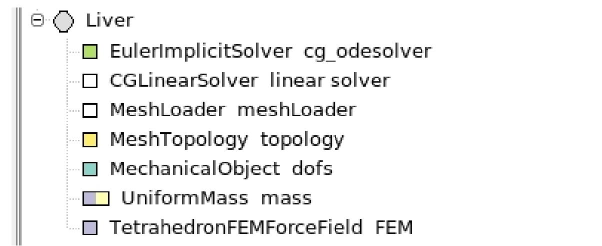


Figure 1: Basic example modelling a Finite Element Object

### 1.1.2 Visual

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

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

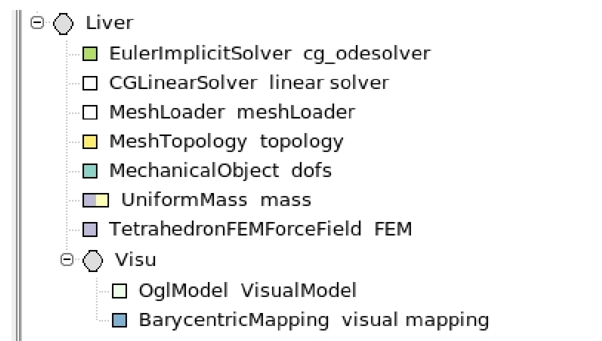


Figure 2: Basic example modelling a Finite Element Object with a Visual Model

### 1.1.3 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

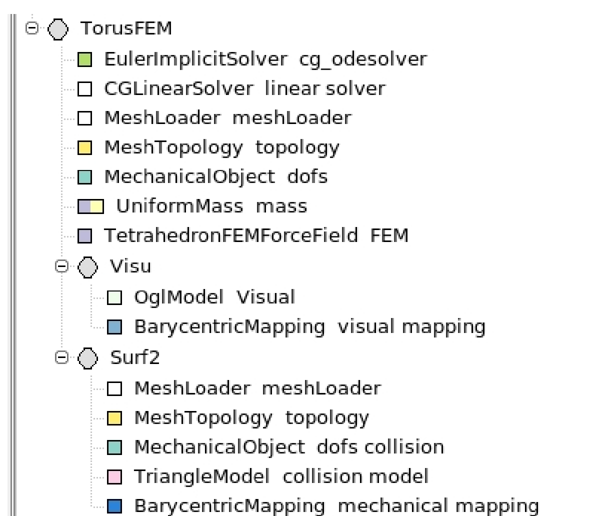


Figure 3: Basic example modelling a Finite Element Object with a Visual Model and Collision-Model

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

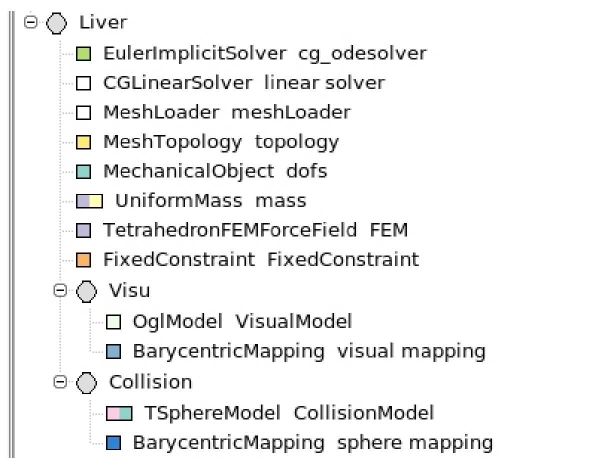


Figure 4: Modeling a liver with sphere collision model

## 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.

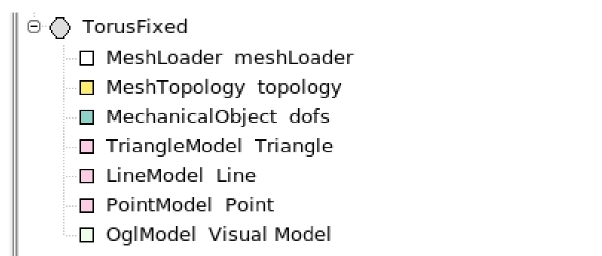


Figure 5: Modeling a Fixed object

## 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

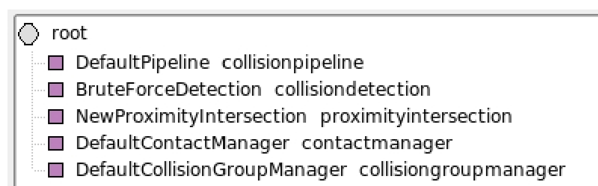


Figure 6: Collision Components

# 2 How To create a new Force Field

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

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

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

$$f+ = Bv + Kx$$

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

# 3 How To include objects in a XML simulation

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

## 3.1 Include an object

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

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

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

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

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

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

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

- the MechanicalObjects :

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

- the VisualModels

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

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

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

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

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

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

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

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

### 3.2 Including a set of components

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

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

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

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

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

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

### 3.3 Commented examples

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

# 4 How To make your Component modifiable

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

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

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

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

initData takes several parameters:

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

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

# 5 How to use the carving manager

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

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

```
1    <Node name="Liver" >
2        <Object type="MeshLoader" filename="mesh/liver.msh" />
3        <Object type="MechanicalObject" />
4        <include href="Objects/TetrahedronSetTopology.xml" />
5        <Node name="CollisionModel" >
6            <include href="Objects/TriangleSetTopology.xml" />
```

```
7                <Object type="Tetra2TriangleTopologicalMapping" object1="../../Container"
8                object2="Container"/>
9            <Object type="TriangleSet" contactStiffness="100" />
10        </Node>
11    </Node>
```

The different Data's of this object are:

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

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