Introduction to SOFA

François Faure

2007

Abstract

This document is targeted to the new SOFA users who want to understand the main principles of this library. More experienced users might learn things too. It is a mix of a high-level presentation and a guided tour of the code. Comments and suggestions are welcome on the SOFA users mailing list.

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.

2 Commented example

Figure 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 6.1. The corresponding scene graph is shown in figure 2. Note that the graph in the left of figure 1 only displays a hierarchical view, while the whole graph includes additional pointers displayed as dashed arrows in figure 2.

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

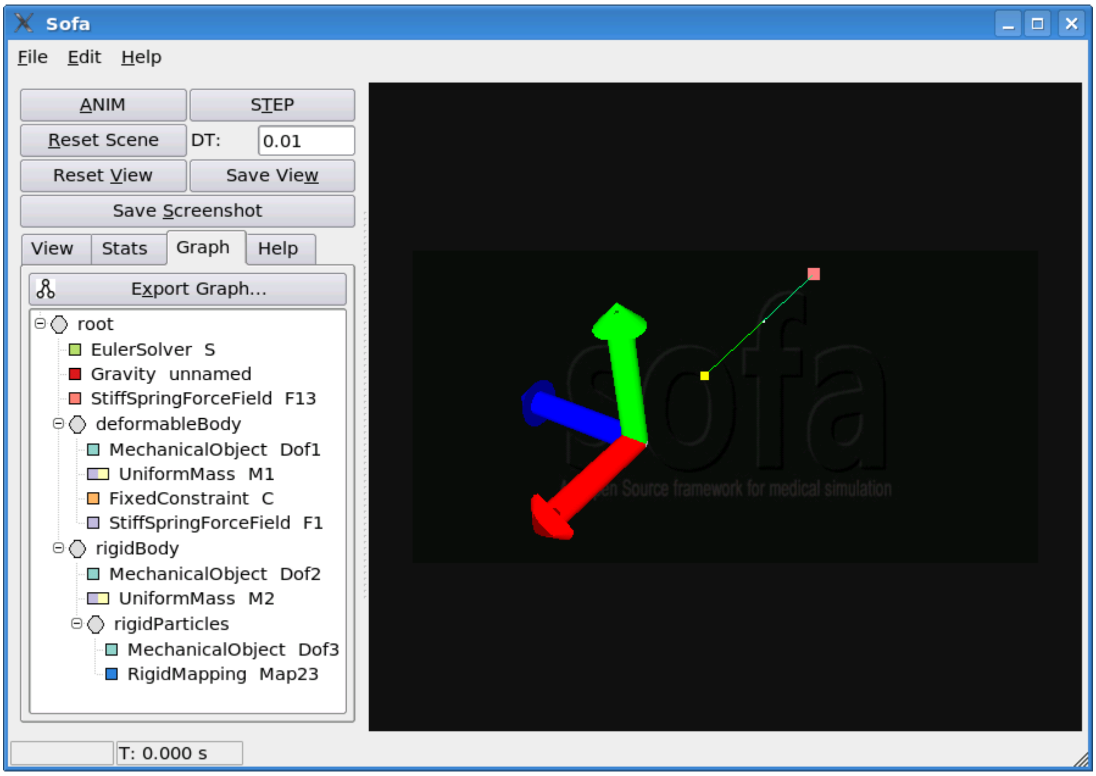

Figure 1: A pendulum composed of a rigid body (reference frame and yellow point) attached to an elastic string (green) fixed at one end (pink point). The corresponding scene graph is displayed on the left.

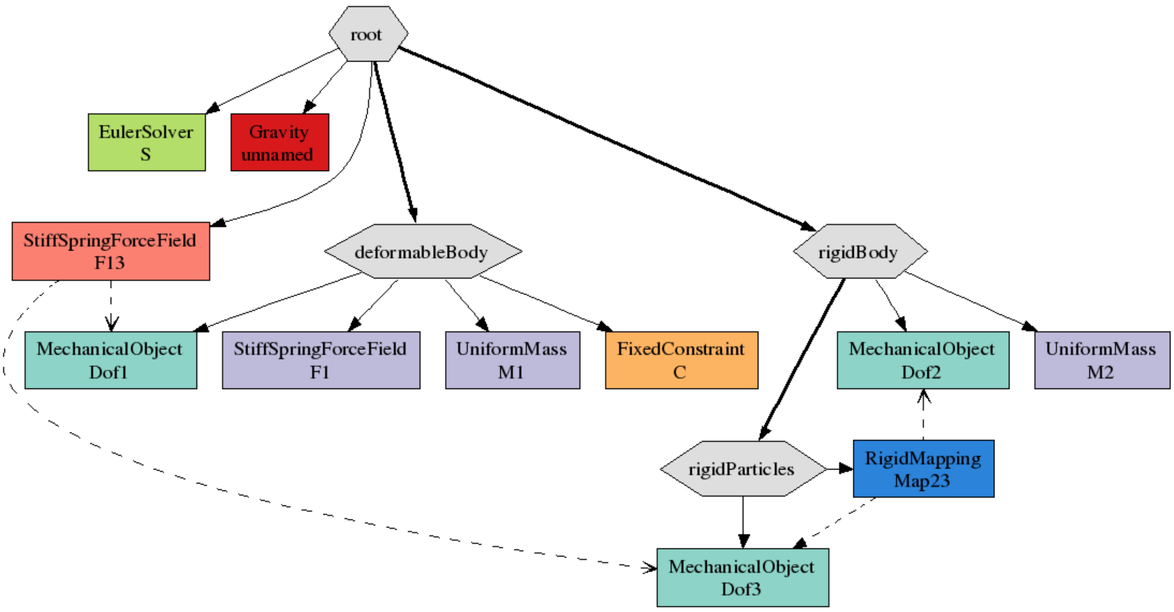

Figure 2: The scene graph of the mixed pendulum. The nodes are displayed as grey hexagons, while the components are displayed as rectangles with colors associated with their types or roles. The bold plain arrows denote node hierarchy, while the thin plain arrows point to the components attached to the nodes, and the dotted arrows denote pointers between components.

- `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

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 be replaced by a `DiagonalMass` instantiated 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.

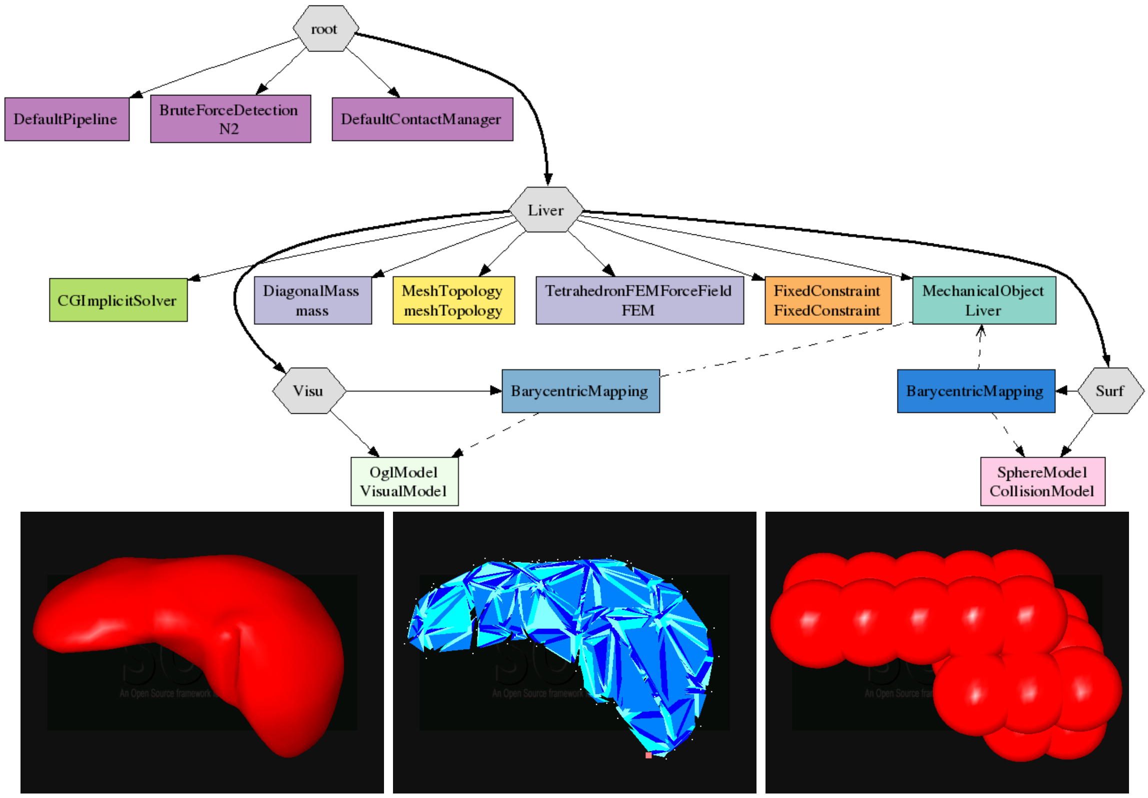

Figure 3: A liver. Top: scene graph. Bottom: visual model, mechanical model, collision model, respectively.

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.

3 Multi-model objects

An important feature of Sofa is the possibility of using different models of a single physical object. Figure 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 6.2.

On top of the scene, collision-related components allow a user to interact with the collision

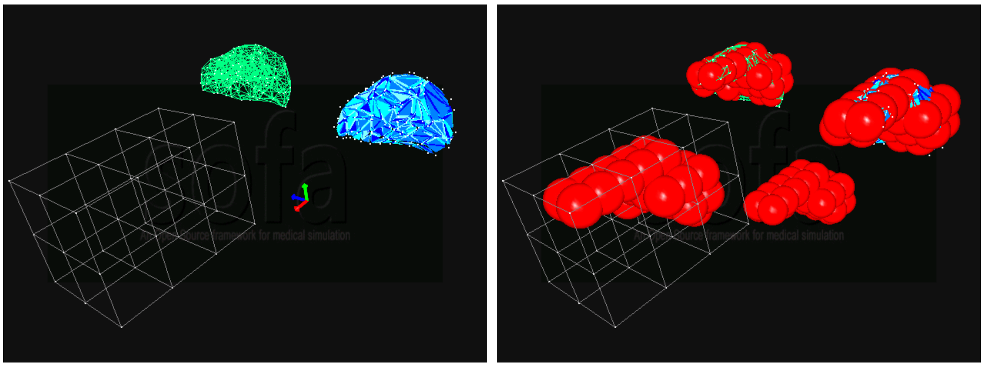

Figure 4: Left: four behavior models (from left to right: deformable grid, springs, rigid, tetrahedral FEM) combined with the same collision model (right).

models using rays cast 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 **??**). 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 4.

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

Figure 5: Pseudocode for a standard simulation program.

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 5. In SOFA, each of the simulation methods is implemented as a recursive graph traversal, `InitVisitor`, `AnimateVisitor` and `VisualDrawVisitor`, respectively. Visitors are explained in the next section.

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`.

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 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.

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

Figure 6: Pseudocode for explicit Euler integration.

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 `VecId`s, 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. 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, `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 child 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 `MechanicalPropagatePositionAndVelocityVisitor` 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`.

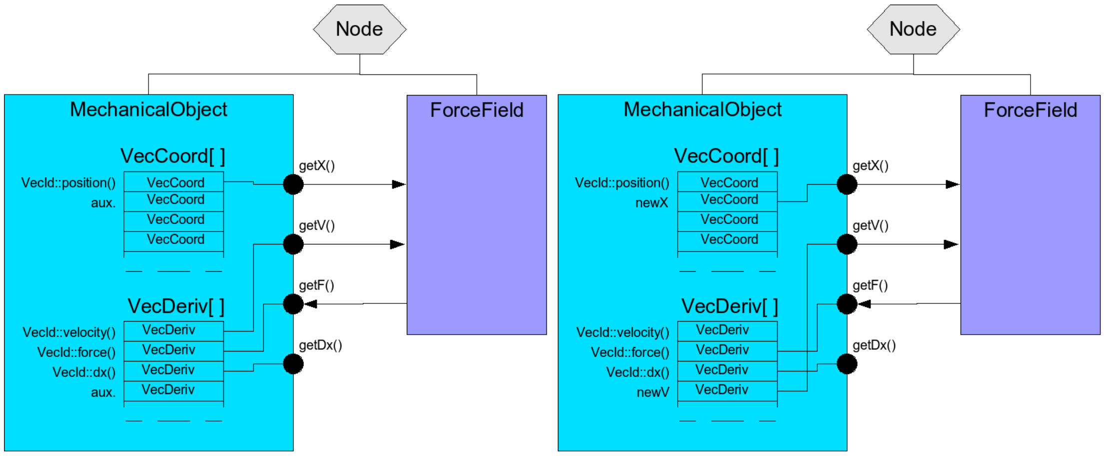

Figure 7: A `MechanicalObject` and a component addressing it. Left: using the default state vectors. Right: using auxiliary state vectors.

It is also used by some solvers when auxiliary states are needed, as discussed in section 5, in order to update the mapped DOFs.

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

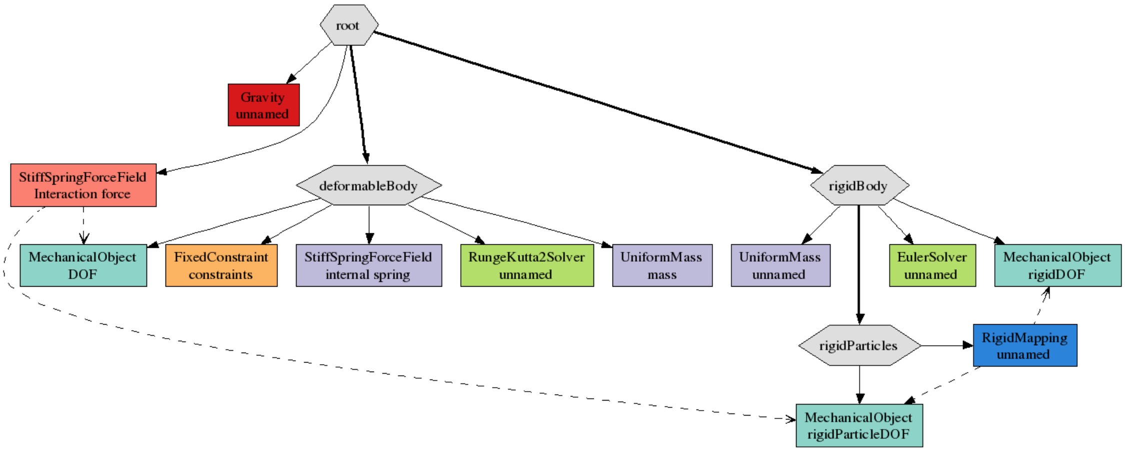

Figure 8: A scene graph with objects animated using different ODE solvers.

opment 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.

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 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 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 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 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.

6 Code of the examples

6.1 The hybrid pendulum

This is the code of the example commented in section 2 :

```
/******************************************************************************
*       SOFA, Simulation Open-Framework Architecture, version 1.0 beta 4      *
```

```
*                   (c) 2006-2009 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/GNode.h>
#include <sofa/simulation/tree/TreeSimulation.h>
#include <sofa/component/contextobject/Gravity.h>
#include <sofa/component/odesolver/EulerSolver.h>
#include <sofa/component/odesolver/StaticSolver.h>
#include <sofa/component/visualmodel/OglModel.h>
// gui
#include <sofa/gui/GUIManager.h>
#include <sofa/core/VecId.h>
#include <sofa/core/objectmodel/Data.h>
#include <sofa/helper/accessor.h>

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

using namespace sofa::simulation::tree;
typedef sofa::component::odesolver::EulerSolver Solver;
using sofa::core::objectmodel::Data;
using sofa::helper::ReadAccessor;
using sofa::helper::WriteAccessor;
using sofa::core::VecId;

int main(int, char** argv)
{
    sofa::gui::GUIManager::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->setGravity( Coord3(0,-10,0) );

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

    //-------------------- Deformable body
    GNode* deformableBody = new GNode("deformableBody", groot);

```

```
// degrees of freedom
MechanicalObject3* DOF = new MechanicalObject3;
deformableBody->addObject(DOF);
DOF->resize(2);
DOF->setName("Dof1");
WriteAccessor< Data< VecCoord3 > > x = *DOF->write(VecId::position() );
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, 100., 1, splength );


//-------------------- Rigid body
GNode* rigidBody = new GNode("rigidBody",groot);

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

// mass
UniformMassRigid3* rigidMass = new UniformMassRigid3;
rigidBody->addObject(rigidMass);
rigidMass->setName("M2");
UniformMassRigid3::MassType* m = rigidMass->mass.beginEdit();
m->mass=0.3;
UniformMassRigid3::MassType::Mat3x3 inertia;
 inertia.fill(0.0);
float in = 0.1f;
 inertia[0][0] = in;
 inertia[1][1] = in;
 inertia[2][2] = in;
m->inertiaMatrix = inertia;
m->recalc();
rigidMass->mass.endEdit();


//-------------------- the particles attached to the rigid body
GNode* rigidParticles = new GNode("rigidParticles",groot);

// degrees of freedom of the skin
MechanicalObject3* rigidParticleDOF = new MechanicalObject3;
 rigidParticles->addObject(rigidParticleDOF);
rigidParticleDOF->resize(1);
rigidParticleDOF->setName("Dof3");
WriteAccessor< Data< VecCoord3 > > rp_x = *rigidParticleDOF->write(VecId::position() );
```

```
    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
    RigidMappingRigid3_to_3* rigidMapping = new RigidMappingRigid3_to_3(rigidDOF,rigidParticleDOF);
    std::string pathobject1("@"+rigidBody->getName()+"/"+rigidDOF->getName());
    std::string pathobject2("@"+rigidParticles->getName()+"/"+rigidParticleDOF->getName());
    rigidMapping->setPathInputObject(pathobject1);
    rigidMapping->setPathOutputObject(pathobject2);
     rigidParticles ->addObject( rigidMapping );
    rigidMapping->setName("Map23");


    // ---------------- Interaction force between the deformable and the rigid body
     StiffSpringForceField3 * iff = new StiffSpringForceField3( DOF, rigidParticleDOF );
     iff->setPathObject1(deformableBody->getName()+"/"+DOF->getName());
     iff->setPathObject2(rigidParticles->getName()+"/"+rigidParticleDOF->getName());
    groot->addObject(iff);
     iff->setName("F13");
     iff->addSpring( 1,0, 100., 1., splength );




    //========================== Init the scene
    sofa::simulation::tree::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::GUIManager::MainLoop(groot);
}
```

6.2 A liver

The XML code of the liver discussed in section 3 page 4 is in `../examples/Demos/liver.scn`