An Introduction to Sofa

François Faure[1]

[1]Grenoble Universities
INRIA - Evasion

September 11th 2007

SOFA - Simulation Open Framework Architecture

- Open Source framework primarily targeted at real-time simulation
  - create complex and evolving simulations by combining new algorithms with algorithms already included in SOFA
  - modify most parameters of the simulation: deformable behavior, surface representation, solver, constraints, collision algorithm, etc. by simply editing an XML file
  - build complex models from simpler ones using a scene-graph description
  - efficiently simulate the dynamics of interacting objects using abstract equation solvers
  - reuse and easily compare a variety of available methods
- Currently developed by four research teams in two institutes: INRIA(Alcove, Asclepios, Evasion), CIMIT(SimGroup,MIT/Harvard/MGH)

Main Features

- High modularity using a scene graph
- Multiple models of the same object can be synchronized using mappings
- Current implementations:
  - deformable objects (springs, FEM)
  - rigid bodies (6-dof)
  - fluids (SPH, Eulerian)
  - collision detection: spheres, triangles
  - explicit and implicit time integration

Using Sofa

- In your application, as an external simulation library (C++, LGPL license)
- As a stand-alone application with one of the default user interfaces (Qt,Glut) and renderers (OpenGL, Ogre)

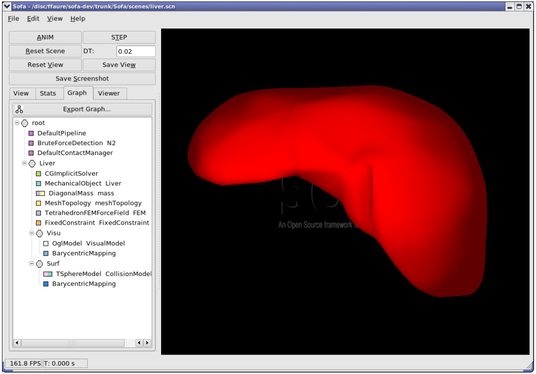


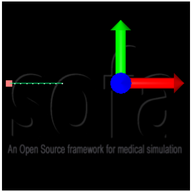
Figure: Sofa stand-alone with Qt GUI and OpenGL rendering.

Data structure

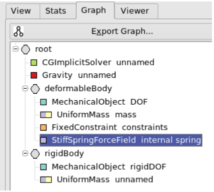
Scene graph with three levels of hierarchy

- Nodes contains Nodes and Components.
- Components are leaves of the scene graph. They implement algorithms. They contain DataFields.
- DataFields contain raw data (positions, masses, ...)

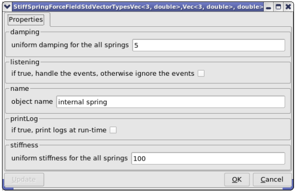
Example: A mass-spring string and a rigid body



the objects:
one string
on rigid



scene graph:
3 nodes
8 components



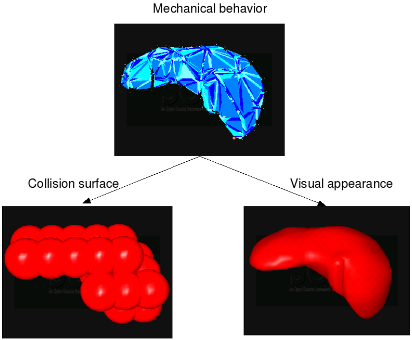
the datafields of a selected component

Why using scene graphs

- standard graphics tool
  - modeling languages: VRML1, VRML2, X3D...
  - libraries: OpenInventor, Java3D, OpenSG, OpenSceneGraph,...
- simple structure (Directed Acyclic Graph): no cyclic dependencies
- abstraction and modularity
- dynamically add or remove objects in the scene
- I/O file format

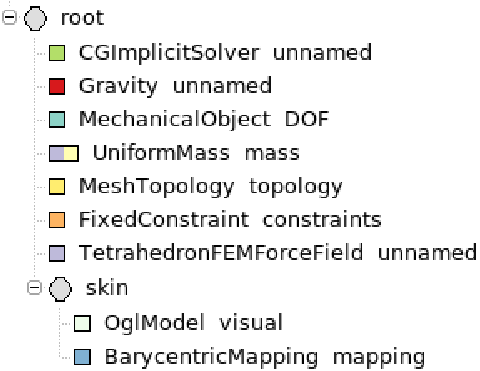
Multiple models of a given object

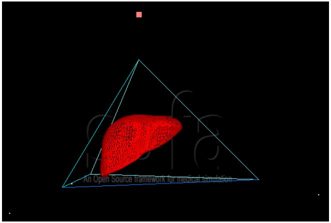
- different geometries for different purposes
- one master geometry (mechanical behavior) contains the independent degrees of freedom (dof)
- implemented as a node hierarchy
- non-independent dofs updated using *mappings*

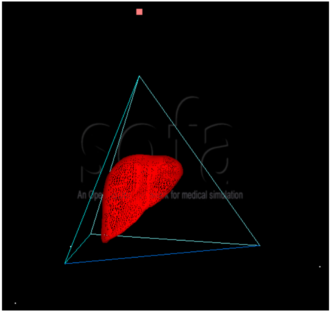


Example of mapping

- 4 mechanical control points
- one finite element (tetrahedron)
- hundreds of vertices in the visual model, all totally defined by the mechanical control points

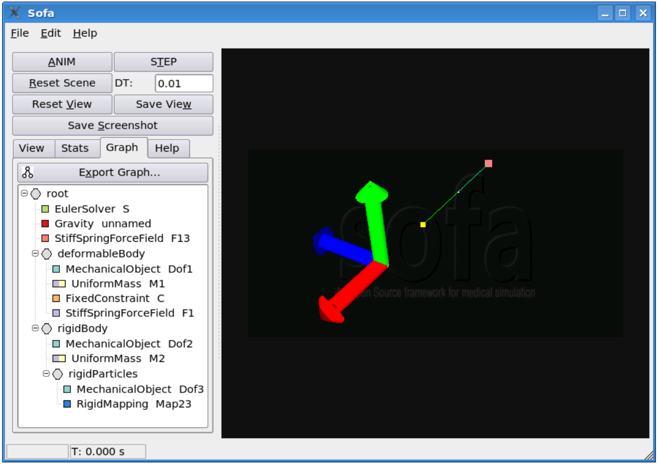






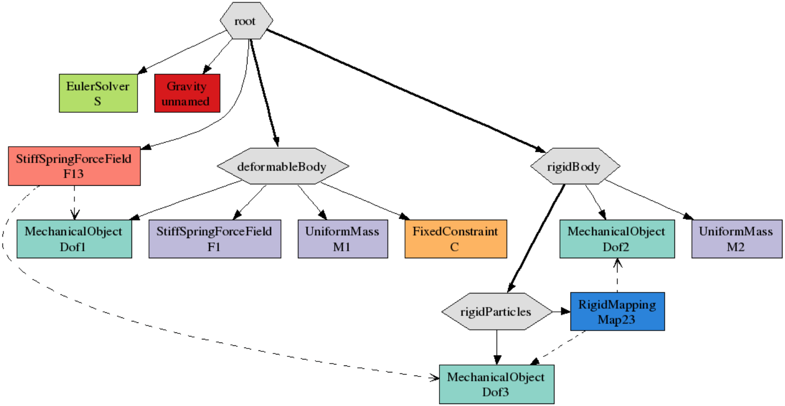
Interacting bodies

- We attach a point to the rigid body using a mapping
- We insert a spring between this point and the string



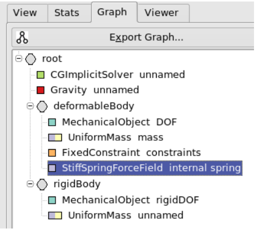
A more complete picture of the scene graph

- The graph displayed in the GUI shows only the hierarchy
- Additional pointers are used:

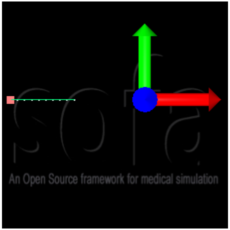


Interacting bodies

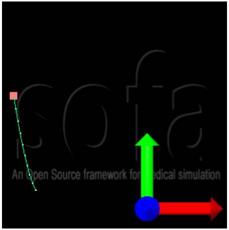
Without interaction force



scene graph



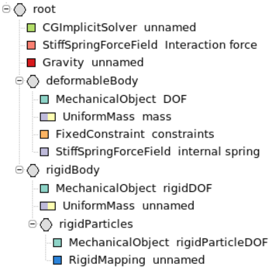
t=0



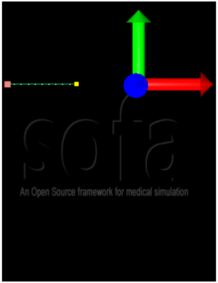
t = ...

Interacting bodies

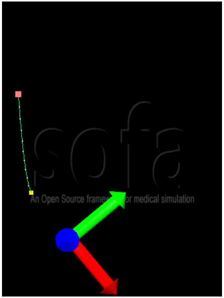
With interaction force



scene graph



t=0



t = ...

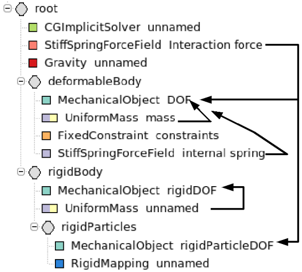
Data processing

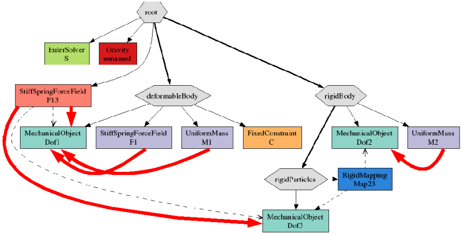
Sofa Actions:

- The scene graph is recursively traversed top-down and bottom-up
- Callbacks are applied to components of certain classes
- Used for all operations: physical computations, vector operations, collision detection,...
- Example: accumulate forces
  - top-down: masses and force fields accumulate force
  - bottom-up: mappings dispatch forces to their parents

Example of action

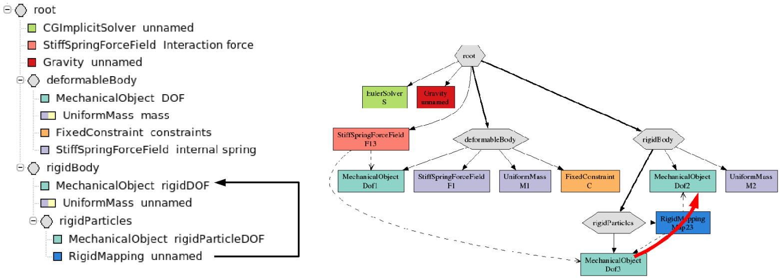
- Accumulate forces, top-down callbaks:
  - masses accumulate their weight
  - force fields (springs) accumulate their force
- each force is applied to the local Degrees of Freedom (DOF)
- except the interaction force





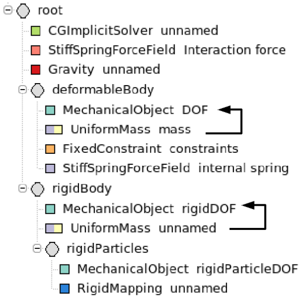
Example of action (continued)

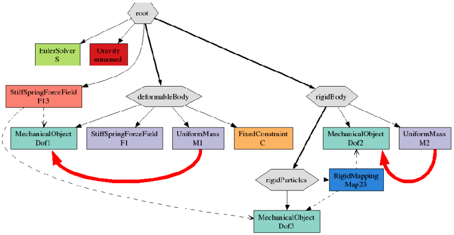
- Accumulate forces, bottom-up callbacks:
  - mappings accumulate force from child to parent
- At the end, the net force applied to the independent (i.e. non-mapped) DOF is computed.



Another action

- Compute acceleration, top-down callbacks:
  - masses compute $M^{-1}f$
- modularity: uniform or diagonal (or matrix-) masses





Degrees of Freedom (DOF)

- can be of different types (particles/rigids, float/double, ...)
- are stored in template class `MechanicalObject<DataType>`
- example of DataType (simplified):

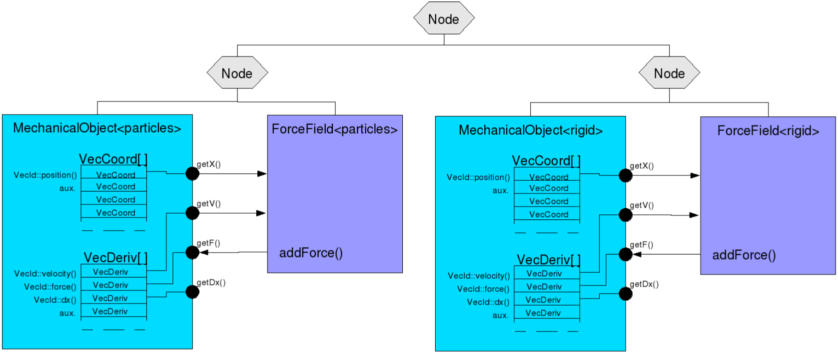
```
class Vec3fTypes
{
  public:
  // Basic data
      typedef float          Real;
      typedef Vec<3,float> Coord; // coordinates
      typedef Vec<3,float> Deriv; // all the rest: velocities, forces,...

  // Data containers
      typedef vector<Real>  VecReal;
      typedef vector<Coord> VecCoord;
      typedef vector<Deriv> VecDeriv;
};
```

- `Coord` differs from `Deriv` in 6-dof rigid bodies

Abstract state vectors

- Due to different DataTypes, the state vectors (x,v,f,a,aux,...) are scattered in the MechanicalObjects



- A MultiVector is a union of state vectors with a common index
- Standard vector mathematics is implemented (sums, dot products...)
- But no [],size(),begin(),end()

Using state vectors

- Reserved indices are used to denote x,v,f,dx
- Auxiliary vectors can be used, e.g. in the beginning of Runge-Kutta2:

```
// Allocate auxiliary vectors
MultiVector acc(this, VecId::V_DERIV);
MultiVector newX(this, VecId::V_COORD);
MultiVector newV(this, VecId::V_DERIV);

// Compute state derivative. vel is the derivative of pos
computeAcc (startTime, acc, pos, vel); // acc is the derivative of vel

// Perform a dt/2 step along the derivative
newX = pos;
newX.peq(vel, dt/2.); // newX = pos + vel dt/2
newV = vel;
newV.peq(acc, dt/2.); // newV = vel + acc dt/2
```

Algorithms

At each time step:

1. Collisions are modeled by springs using a CollisionAction
2. Differential Equation solvers update x and v
3. Mappings propagate the new state to all the layers and external buffers (rendering)

ODE solvers and collision modeling modules are easily customizable.

Conclusion

Strengths:

- Interaction of multiple models using mappings
- Efficient simulation of deformable bodies using implicit time integration
- Highly customizable

Weaknesses:

- Documentation is not complete
- Main loop is not yet customizable
- Constraint-based methods are difficult to implement
- Still a lot to do: friction, haptics, faster collision detection...

Future work:

- get rid of weaknesses
- parallelisation (clusters, and more GPU)
- easy debugging