Robotics Engineering course

University of Genoa

Simulation models

Student Chiara Saporetti **Year:** 2020 - 2021

Contents

1 Introduction

Here are reported the simulation models that I developed on Sofa Framework for my thesis. I tried to write a brief explanation of almost all parameters and why I chose something over something else. First there are the general sofa parameters, then skin models, and lastly the phantom and instruments models.

2 General parameters

Here are the general parameters that I used in my models. These must be defined prior to the 3D models, in the root node.

2.1 Animation Loop

Sofa provides:

- DefaultAnimationLoop
- MultiTagAnimationLoop
- MultiStepAnimationLoop
- FreeMotionAnimationLoop

I chose the animation loop that is usually chosen when using Lagrangian constraints: the **FreeMotionAnimationLoop**. This component divides the simulation into two main steps: a free motion and a correction step.

First, the free motion computes the projective constraints, the physics, and solves the resulting free linear system. Second, the correction step solves the constraints based on the Lagrange multipliers. It requires a *ConstraintSolver* and a *ConstraintCorrection*, which will be analyzed later. In code:

```
1 root.addObject('FreeMotionAnimationLoop')
```

2.2 Collision pipeline

The collision pipeline follows three steps:
1. reset of the collision
2. a collision detection: broad phase and narrow phase
3. a collision response

General collision pipeline definition in code:

```
root.addObject('CollisionPipeline', depth="6", verbose="0",
draw="0")
```

depth: how deep the BVH is searched for collisions. In all Geomagic examples it is 6, so this is the value I chose.

2.2.1 Broad phase

This phase quickly removes object pairs that are not in collision by checking the bounding boxes. It then returns the pairs that may potentially collide.
Sofa provides:

- brute force (based on BVH)
- ray tracing (based on octrees)
- sap (not common)

To implement it I used the **brute force** component. It is based on the comparison of the overall bounding volumes in the Bounding Volume Hierarcy to determine if they are in collision or not. In code:

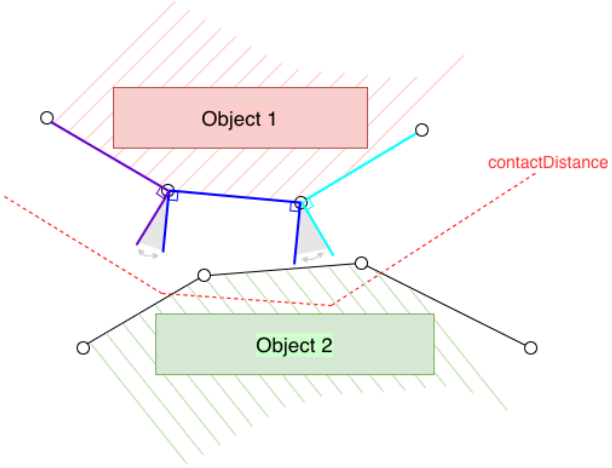
```
root.addObject('BruteForceDetection', name="detection")
```

2.2.2 Narrow phase

This phase checks the output of the broad phase and decides which elements are actually colliding. It returns pairs of geometric primitives, along with the associated contact points.
Sofa provides:

- min proximity intersection
- local min distance

Both of them detect a possible contact as soon as pair of collision elements are close to each other (distance smaller than the alarmDistance) and create contact when the distance is lower than contactDistance. However, in minProximityIntersection, the intersection might have a high number of contacts and, by using a response method based on Lagrange multipliers, there would be too many constranints generated: too computationally demanding.
So, to implement it I used the **local min distance**. To find an optimal number of contact points, the LocalMinDistance computes cones on all nodes of the collision model. A cone is the combination of the orthogonal directions/planes of the



neighboring lines/surfaces (see Fig ??). All contact outputs which are outside these cones will be invalidated (even if they are below the contactDistance). Thus, only the geometrically closest contacts remain: for convex surfaces: only one contact point.

```
root.addObject('LocalMinDistance', name="proximity", alarmDistance="1",
    contactDistance="0.05", angleCone="0.1")
```

It is generally better to set a contact distance that stays above zero.

2.2.3 Response

For this phase there is only one option available, the default contact manager. However, some parameters allow for a greater variation. I chose, for example, the friction contact.

```
root.addObject('DefaultContactManager',
name="CollisionResponse", response="FrictionContact")
```

response: This parameter is what allows to achieve great flexibility: used to specify which kind of contact should be simulated; responseParams: This is a list of parameters used by the specified response method (listed by separating them with) .

2.3 Integration scheme

The idea is to implement discretization of the temporal evolution of the simulation in small time intervals dt. Obviously, choosing a very small dt is more computationally demanding but also more accurate. At each dt, the system must solve the equation F=ma (or b=Ax) to find velocity and position of all the points in the simulation. Integration schemes are numerical methods that describe how to find the approximate

solution to the equation and define how the linear matrix system Ax=b must be built.
Sofa provides: two kinds of schemes in sofa: implicit (slower but more stable) and explicit (faster but need a very small dt to be stable).
I used an implicit Euler, which is a time integrator that uses backward Euler scheme for first- and second- order Ordinary Differential Equations.

```
epi.addObject('EulerImplicitSolver', name="ODE solver",
rayleighStiffness="0.01", rayleighMass="0.01")
```

The Rayleigh stiffness and mass refer to the Rayleigh damping. The Rayleigh damping is a numerical damping with no physical meaning. It corresponds to a damping matrix that is proportional to the mass or/and stiffness matrices using coefficients, respectively Rayleigh stiffness factor or Rayleigh mass factor. This numerical damping is usually used to stabilize or ease convergence of the simulation. The values should be tuned. Right now I just put the ones I found in the examples but I will need to vary them.

2.4 Solver

Solves the Ax=b problem. Sofa provides:

- CG linear solver: solves Ax=b without prior knowledge. Relies on conjugate gradient method: $r = b - Ax^k$, where r is the residual, which is used to compute mutually conjugate vectors to be used as a basis for the approximate solution.
- sparseLDLsolver: solves Ax=b without prior knowledge. Relies on the method of LDL decomposition. The system matrix will be decomposed as A=LDL, where L is the lower part of the matrix A and D is its is its diagonal. LDL simply computes the direct solution and, as a direct solver, it might be extremely time consuming for large system. However, it will always give you an exact solution, making the assumption that the system matrix A is symmetric.
- ShewchukPCGLinearSolver: similar to CG linear solver but allows to add a preconditioner. Preconditioners are used in cases where the convergence of the system is slow (usually a ill-conditioned system). To preserve accuracy while improving performance, preconditioning methods aims at projecting a matrix P on the linear system, in order to get closer to the solution. The efficiency of the preconditioner will depend on the choice of the P matrix.

I opted for a CG linear solver.

```
epi.addObject('CGLinearSolver', iterations="25", name="linear solver",
    tolerance="1.0e-9", threshold="1.0e-9")
```

iterations: maximum number of iterations after which the iterative descent of the CGLinearSolver must stop;
tolerance: defines the desired accuracy of the Conjugate Gradient solution (ratio of current residual norm over initial residual norm)
threshold: defines the minimum value of the denominator in the conjugate Gradient solution.

3 The first skin model: 3 layers of hexahedra

The first model is made of three different layers that are then geometrically attached one to the other. The structure of all the models is the same:

- Behavior model: hexahedra made with a regular grid: Fig 1.
- Collision model: triangles (created with hexa2quad and then quad2triangle mapping from behavior model): Fig 2.
- Visual model: triangles (identity mapping from collision): Fig 3.

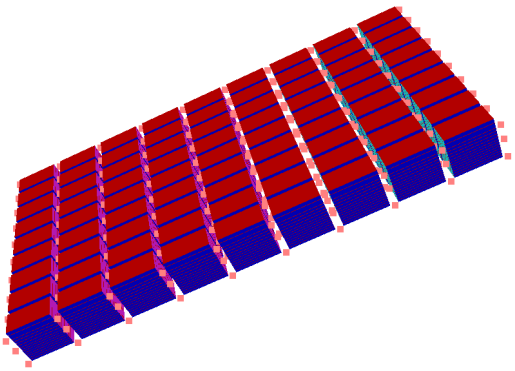


Figure 1: Skin1 behavior

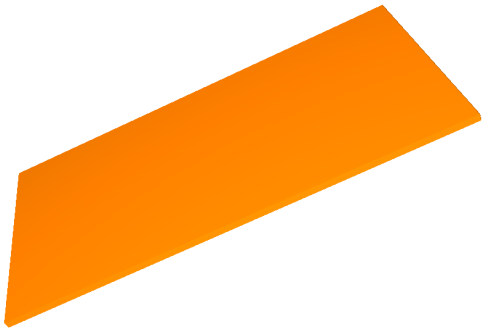


Figure 2: Skin1 collision

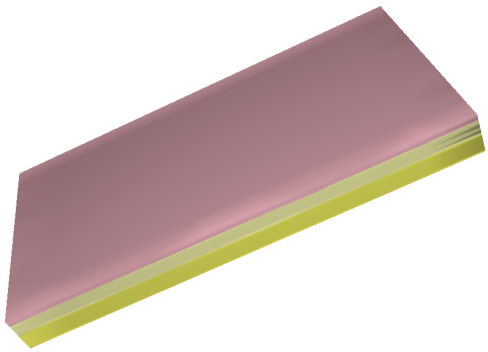


Figure 3: Skin1 visual

4 Behavior model

4.0.1 Topology

I used a Regular grid topology. This topology models the object as a regular 3D grid made of nx x ny x nz cells of hexahedral shape. I chose these values according to the information found in the literature, by at least trying to maintain the same proportions. Specifically:

- The epidermis layer has a thickness of approximately 0.1 mm.
- The derma layer has a thickness of approximately 1 mm .
- The hypodermis layer has a thickness of approximately 1.2 mm.

In code, the parameters:

```
z_epi=2
z_derma=11
z_hypo=13
```

Note: sofa sets one vertex less than the number that is specified.

In code, for the hypoderma layer:

```
hypo.addObject('RegularGridTopology', name="grid", nx=x_vertices, ny=
    y_vertices, nz=z_hypo, xmin=x_offset, xmax=x_length, ymin=y_offset,
    ymax=y_length, zmin=z_offset, zmax=(z_hypo+z_offset))
```

4.0.2 Mass

Sofa provides:

- UniformMass: Mass matrix is diagonal, no space integration is done: for rigid frames only, since topology is not accounted for..
- MeshMatrix: Performs integration of density over volume. Needs the total mass, how the total mass is distributed over the topology, and massDensity, needed for the integration.
- DiagonalMass: considers the mass matrix to be diagonal, but is still found through integration of density over volume. It is found with mass lumping, a numerical method and thus is less accurate method with respect to the MeshMatrix. It has the same parameters as the mesh matrix.

I used a diagonal mass with density set to 1, but this value must be changed accordingly to what I found in the literature. Values for density in the literature: *approximately 1116 kg mâ3 as the dermis density and approximately 971 kg mâ3 as the subcutaneous fat density.*

4.0.3 Forces

I chose to use hexaedron FEM force field since hexahedra are accurate to model soft tissue behaviour. The only topology that supports this is the Regular Grid Topology, described above. The force field is described with two components: the Young Modulus and the Poisson Ratio.
Young modulus is a mechanical property that measures the tensile stiffness of a solid material. It quantifies the relationship between tensile stress (force per unit area) and axial strain (proportional deformation) in the linear elastic region of a material. Poisson's ratio is a measure of the Poisson effect, the deformation (expansion or contraction) of a material in directions perpendicular to the specific direction of loading. From the literature, the values for the three skin layers I simulated are: The epidermis layer has the highest Young modulus (approx. 1 MPa) compared with the other skin layers. The dermis layer has its Young's modulus is much lower compared with the epidermis (from approx. 88 to 300 kPa). The subcutaneous fat layer has the lowest stiffness (around 34 kPa) among all the skin layers..
As with the width of the layers I traied to maintain comparable values of these values. In code:

```
epi.addObject('HexahedronFEMForceField', template="Vec3d", name="FEM",
    poissonRatio=0.49, youngModulus=200)
```

4.0.4 Mechanical object

The Mechanical Object is a 3d vector.

```
epi.addObject('MechanicalObject', template="Vec3d", name="Hexa_E",
    position="0 0 -8", showVectors="true", drawMode="2")
```

4.0.5 Topological mappings

It is possible to define a mesh topology from another mesh topology using the same degrees of freedom by using mappings. Mappings can be used either to go from one topology to a lower one in the topological hierarchy (from tetrahedra to triangles), or to split elements (quads into triangles). As usual mappings, forces applied on the slave topology are propagated onto the master one. Both topologies will therefore be assigned to the same MechanicalObject. as previously stated I used hexa2quad and quad2triangle.

4.0.6 Constraints

Sofa provides: projective and Lagrangian constraints.
Projective constraints allow to project the velocity of a constrained point to a desired

velocity. To do so they use a projection matrix.
Lagrangian constraints handle complex constraints such as contacts and joints between moving objects that can not be implemented using projection matrices.

4.0.7 Constraint solvers

The resolution of the constraint problem is done using the Gauss-Seidel algorithm. Sofa provides: two different implementations: LCP and Generic. The Geomagic device needs to solve the Linear Complementary Problem (LCP), so I chose this constraint solver.
In code:

```
root.addObject('LCPConstraintSolver', tolerance="0.001", maxIt="1000")
```

4.0.8 Constraint corrections

Constraint correction:

- UncoupledConstraintCorrection: makes the approximation that the compliance matrix Compliance matrix is diagonal. Strong assumption: constraints are independent.
- LinearSolverConstraintCorrection: computes the compliance matrix, by using A^-1 from a direct solver. This approach can therefore be very computationally-demanding if you have many constraints.
- recomputedConstraintCorrection: instead of computing Inverse of A at each time step, this constraint correction precomputes once the inverse of A at the initialization of the simulation and stores this matrix. Simualtion is faster but lacks accuracy if A changes during simulation.
- GenericConstraintCorrection

I chose UncoupledConstraintCorrection. In code:

```
epi.addObject('UncoupledConstraintCorrection')
```

Note: Before I had set LinearSolverConstraintCorrection. However I had this error: *[WARNING] [GraphScatterredType] set an element is not supported in MultiVector*
In Linear solver error I read that: *for the skin, you are using a LinearSolverConstraintCorrection. Such a constraint correction requires a direct linear solver to be able to compute the A-1 matrix, used for the computation of the compliance matrix W. Either you use a direct linear solver (like the LDL you commented) or you change for a UncoupledConstraintCorrection. Be careful, choosing one or the other are two*

different assumptions on your system. So I decided to use UncoupledConstraintCorrection, for the moment.

4.0.9 Constraint laws

- bilateral interaction: defines an holonomic constraint law between a pair of simulated body. Such a constraint is suited for attachment cases or sliding joints (paired objects).
- unilateral interaction: defines an non-holonomic constraint law between a pair of simulated body. defines an non-holonomic constraint law between a pair of simulated body. Must also use a ConstraintCorrection so that the corrective motion can be applied. Unlike other constraints, the UnilateralInteraction-Constraint is mostly used in SOFA for contact modeling (contact and collision cases)
- Sliding interaction: like a bilateral but only active for some vectors of the physics space (for instance only the x-direction)

I still need to understand how to actually implement constraint laws in my code. I think I will need to use them in the task simulations.

4.0.10 My projective constraints

Constraints allow to project the velocity of a constrained point to a desired velocity. To do so they use a projection matrix. *Sofa provides*:

- Fixed: point stays still
- Partial fixed: point doesn't move along certain degrees of freedom

And can be implemented as:

- Attach constraint: works with a pair of objects and it projects the degrees of freedom (e.g. position) and their derivatives (e.g. velocity), so that both objects are attached. Note: it ensures a geometrical connection between both objects at the end of the time step but it does not integrate the physics of both object!
- Fixed constraint.

In particular I put fixed constraints at the bottom of hypoderma. Indices are calculated by a simple function *computeIndices*. I know that during simulation, if you touch the constraints, the model deforms badly. The guy that worked on Sofa before

me put a plane of contraints and some random constraints in the internal part of the plane. For the moment I put constraints on all the indices at the bottom of the layer. In the future, I'd like to set a simple function that removes the outer annulus of the bottom layer (via matrices).
In code:

```
hypo.addObject('FixedConstraint', template="Vec3d", name="Fixed Dofs",
    indices=computeIndices("bottom", z_hypo))
```

I then used AttachConstraint component to attach the 3 layers of skin by connecting the bottom of each layer to the top of the underlaying layer.
In code:

```
root.addObject('AttachConstraint', name="lowerConstraint", object1="
    @hypo", object2="@derma", indices1=computeIndices("top", z_hypo),
    indices2=computeIndices("bottom", z_derma))
root.addObject('AttachConstraint', name="upperConstraint", object1="
    @derma", object2="@epi", indices1=computeIndices("top", z_derma),
    indices2=computeIndices("bottom", z_epi))
```

4.0.11 Compute indices

For the attached constraints I have written a very simple function that computes the indices. It uses the values along x,y,z of each layer and also needs the layer that it must consider (bottom or top). It defines a 2D array with values in range [0, x*y*z],and shape (z, x*y). Then it outputs the indexes of the desired layer of skin: bottom or top.

```
# Global variables
x_vertices=10
y_vertices=10
n_vertices_per_layer=x_vertices*y_vertices

# Funnction that computes indices
def computeIndices(layer, z):
    result=' '
    indices = np.array(range(z*n_vertices_per_layer)).reshape(z,
    n_vertices_per_layer)
    for i in range(n_vertices_per_layer):
        if layer=="bottom":
            result += str(indices[0,i]) + " "
        elif layer=="top":
            result += str(indices[-1,i]) + " "
```

Collision primitives/models: *Sofa provides*: PointCollisionModel, LineCollisionModel, TriangleCollisionModel, SphereCollisionModel, CylinderCollisionModel.

I used triangle, line and point collision, which are the most used to work with the Geomagic touch.

```
CollisionModel.addObject('PointCollisionModel')
CollisionModel.addObject('LineCollisionModel')
CollisionModel.addObject('TriangleCollisionModel')
```

Note: first I put collision on all three layers. However I had this error (only copied the beginning):

```
[WARNING] [IntersectorMap] Element Intersector SphereCollisionModel
<StdRigidTypes<3u, double> >-TriangleCollisionModel<StdVectorTypes<Vec<3u, double>,
Vec<3u, double>, double> > NOT FOUND within : OBBCollisionModel<StdRigidTypes<3u,
double> >-OBBCollisionModel<StdRigidTypes<3u, double> > TOBB<StdRigidTypes<3u,
double> >-TOBB<StdRigidTypes<3u, double> >
```

In error1Collision error they solved it by switching to v20.12.01. I solved it by removing collision from the second and third layer.

4.0.12 Visual

For the moment I simply set a different color for each layer.

```
visu.addObject('OglModel', template="Vec3d", name="Visual", color="1 1
    0.4", material="Default Diffuse 1 0 0 1 1 Ambient 1 0 0 0.2 1
    Specular 0 0 0 1 1 Emissive 0 0 0 1 1 Shininess 0 45 ")
visu.addObject('OglViewport', screenPosition="0 0", screenSize="250 250"
    , cameraPosition="-0.285199 -15.2745 16.7859", cameraOrientation="
    0.394169 0.0120415 0.00490558 0.918946" )
visu.addObject('IdentityMapping', template="Vec3d,Vec3d", name="
    default12", input="@..", output="@Visual" )
```

Conclusion I really liked this model. However, to modify a model, it is simpler to do it with a tetrahedral model, so I created a new model.

5 The second skin model: 1 layer of tetrahedra

The second model I did is made of tetrahedra, and I realized it by importing a cube with tetrahedral mesh. The parameters are mostly the same as the other model, but I reported the ones that varied. The structure the models is the following:

- Behavior model: tetrahedra: Fig 4
- Collision model: triangles (tetra2triangle): Fig 5
- Visual model: triangles (identity mapping from collision): Fig 6

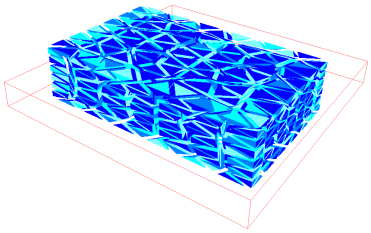


Figure 4: Skin2 behavior

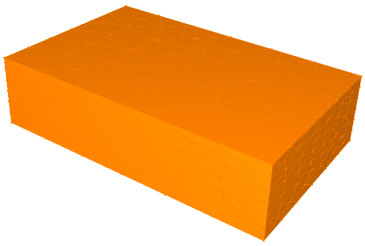


Figure 5: Skin2 collision

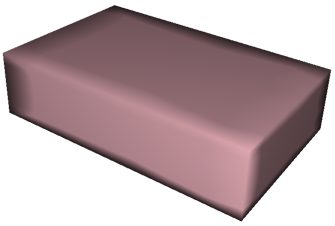


Figure 6: Skin2 visual

5.1 The gmsh model

To import a meshed model, *Sofa provides*: MeshGmshLoader, MeshVTKLoader, MeshObjLoader for volumes and surfaces, and MeshSTLLoader for surfaces only. I made a simple cube on gmsh, which is one of the simplest programes to create figures with meshes. I just learnt the super basic way to create a meshed cube, which is done by clicking: Modules $\longrightarrow$ Geometry $\longrightarrow$ Elementary entities $\longrightarrow$ add $\longrightarrow$ box. Then i added 2D meshes (Fig 7) and exported the file in stl format and msh format. Then i added 3D meshes (Fig 8) instead and saved the file in vtk format and msh format

(in msh: specify Version 2 ASCII, save all elements).

In the end I decided to work with the msh format and, since the surface mesh was

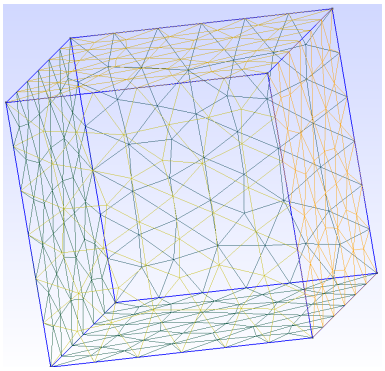


Figure 7: 2D mesh

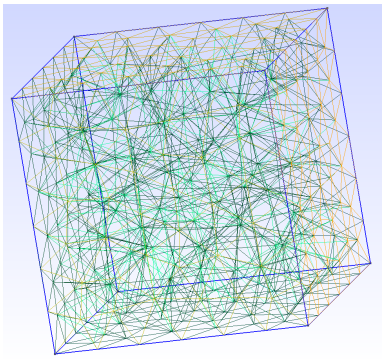


Figure 8: 3D mesh

not imported correctly (see Fig 9) and I didn't understand why, I decided to use a topological mapping from tetra to triangle for both collision and visual models, instead of importing a mesh. In code:

```
skin.addObject('MeshGmshLoader', name='volumeLoader', filename=
    skinVolume_fileName, scale3d=scale3d_std)
skin.addObject('TetrahedronSetTopologyContainer', src='@volumeLoader',
    name='TetraContainer', template='Vec3d')
skin.addObject('TetrahedronSetGeometryAlgorithms', template='Vec3d')
skin.addObject('TetrahedronSetTopologyModifier')
```

5.2 Using the model

Attaching this model with indices was way harder, since when you import a model from a file the indices are not regular as when creating it with RegularGrid, so I just added a BoxROI under the model. This component finds the primitives (vertex/edge/triangle/quad/tetrahedron/hexahedron) inside given boxes, and then these components can be fixed with a RestShapeSpringsForceField, that tends to bring them in position if they are moved.

```
skin.addObject('BoxROI', name='boxROI', box='-50 -50 0 50 50 10',
    drawBoxes='true')
skin.addObject('RestShapeSpringsForceField', points='@boxROI.indices',
    stiffness='1e12', angularStiffness='1e12')
```

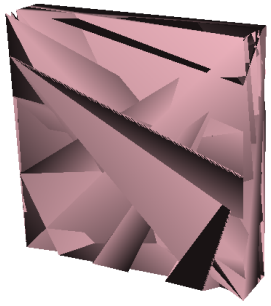


Figure 9: Ugly

6 The instrument

To interact with the skin I needed to simulate also the Geomagic Touch device. There are some parameters that must be set to work with it. First of all, examples divide the device from the instrument that it is going to use.

6.1 The device

In the root node:

```
root.addObject('GeomagicDriver', name="GeomagicDevice",
deviceName="Default Device", scale="1",
drawDeviceFrame="1", drawDevice="0",
positionBase="0 0 8", orientationBase="0.707 0 0 0.707")
```

Then the device node is created:

```
Omni.addObject('MechanicalObject', template="Rigid3",
name="DOFs", position="@GeomagicDevice.positionDevice")
Omni.addObject('MechanicalStateController',
template="Rigid3", listening="true",
mainDirection="-1.0 0.0 0.0")
```

6.2 The instrument connected to the device

In the examples, the instrument is usually the dental instrument. I will use needle/scalpel depending on the task that I want to implement, but at the beginning I just used the one from the examples.

6.2.1 Behavior

It first of all needs a integration scheme and a solver. I chose the same ones as for the skin.

```
instrument.addObject('EulerImplicitSolver', name="ODE solver",
    rayleighStiffness="0.01", rayleighMass="0.02")
instrument.addObject('CGLinearSolver', iterations="25", name="
    InstrumentLinearsolver", tolerance="1.0e-9", threshold="1.0e-9")
```

The mechanical object is positioned on the end effector of the Geomagic device via the GeomagicDevice.positionDevice variable.

```
instrument.addObject('MechanicalObject', template="Rigid3d",
name="instrumentState", position="@GeomagicDevice.positionDevice")
```

The mass can simply be a uniform mass, since it doesn't have to be deformed.

```
instrument.addObject('UniformMass', name="mass", totalMass="1" )
```

I then used another component (that I already used with the boxroi), which is RestShapeSpringsForceField. In fact, in haptics, the virtual environment compliance is modeled with a spring/damper, and this component simulates it. From the literature [2]:

In the direct rendering, the virtual probe directly follows the haptic device (map-map) and the collision forces are sent directly to the device. This approach provides high transparency of the haptic simulation, but less stability when two stiff materials collide in the virtual environment.

To solve the stability problem for direct rendering, a virtual coupling concept was introduced. This method is used for many 6-DOF haptic algorithms to ensure stability of the haptic simulation. The virtual coupling is a spring-damper force model between the device state and the tool object state in a haptic simulation. The virtual spring-damper system generates forces (and torques) that are sent to the haptic display and felt by the operator. The main disadvantage with virtual coupling is that the force sent back to operator can be felt as dampened and smoothened, thus reducing transparency. However, the spring-damper system of the virtual coupling often leads to heuristic optimization of the parameters for a given haptic display. *Stability issues in haptic rendering are related to the situation when two rigid objects are colliding in a virtual world. To be able to transfer a feeling of high stiffness it is important to have a stable haptic simulation. One absolute requirement for maintaining stability of a haptic simulation when two rigid objects are colliding is high haptic update rates of the force calculations (at least 1000 Hz for the collision of rigid bodies).*
The 6-DOF haptic rendering can be split into one of two main categories: direct rendering or virtual coupling. The categories differ in the way the virtual probe of the haptic device is connected to the real position of the haptic device.

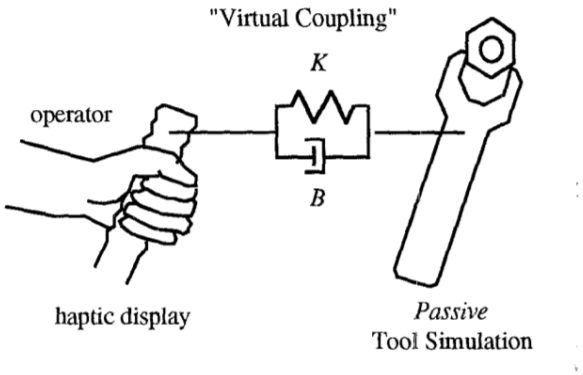


Figure 10: Caption

In the direct rendering, the virtual probe directly follows the haptic device (map-map) and the collision forces are sent directly to the device. This approach provides high transparency of the haptic simulation, but less stability when two stiff materials collide in the virtual environment.

To solve the stability problem for direct rendering, a virtual coupling concept was introduced. This method is used for many 6-DOF haptic algorithms to ensure stability of the haptic simulation. The virtual coupling is a spring-damper force model between the device state and the tool object state in a haptic simulation. The virtual spring-damper system generates forces (and torques) that are sent to the haptic display and felt by the operator. The main disadvantage with virtual coupling is that the force sent back to operator can be felt as dampened and smoothened, thus reducing transparency. However, the spring-damper system of the virtual coupling often leads to heuristic optimization of the parameters for a given haptic display.

To enter in detail, as it is written in [1]: Consider for example the haptic display of a rigid tool interacting with a rigid environment. This interaction is characterized by multiple unilateral constraints. A good simulation approach is to model each unilateral constraint as a spring-damper, and select the stiffness and damping coefficients to be as large as possible without compromising passivity. Because the number of parameters is quite large, and the system quite nonlinear, an analytical result is not feasible. Therefore, it will probably be necessary to use a trial-and-error approach to find appropriate values. See Fig 10.

To represent this in my code, I wrote

```
instrument.addObject('RestShapeSpringsForceField', stiffness='1000',
    angularStiffness='1000', external_rest_shape='@../Omni/DOFs', points
```

```
    ='0', external_points='0')
```

As previously said, the Geomagic wants the LCP force feedback, and I also set the uncoupled constraint correction.

```
instrument.addObject('LCPForceFeedback', name="LCPFF1",
activate="true", forceCoef="0.5")
instrument.addObject('UncoupledConstraintCorrection')
```

6.2.2 Collision

The collision loads the actual instrument object. In this case I took the values of rotation (rx, ry, rz) and position (dx, dy, dz) wrt the end effector from the sofa examples.

```
CollisionModel.addObject('MeshObjLoader', filename="Demos/Dentistry/data
    /mesh/dental_instrument_centerline.obj",
name="loader")
CollisionModel.addObject('MeshTopology',src="@loader", name="
    InstrumentCollisionModel" )
CollisionModel.addObject('MechanicalObject', src="@loader", name="
    instrumentCollisionState", ry="-180", rz="-90", dz="3.5", dx="-0.3")
```

Collision is then computed as a line and a point collision model.

```
CollisionModel.addObject('LineCollisionModel')
CollisionModel.addObject('PointCollisionModel')
```

Finally, mapping between the behavior and the collision is a rigid mapping.

```
CollisionModel.addObject('RigidMapping', name="CollisionMapping",  input
    ="@instrumentState",  output="@instrumentCollisionState")
```

6.2.3 Visual

The visual node loads the mesh again and sets a desired color. It has a rigid mapping with the behavior model.

```
VisualModel.addObject('MeshObjLoader', name="meshLoaderGeo", filename="
    Demos/Dentistry/data/mesh/dental_instrument.obj", handleSeams="1")
VisualModel.addObject('OglModel',  name="InstrumentVisualModel",
src="@meshLoaderGeo", color="1.0 0.2 0.2 1.0",
ry="-180", rz="-90", dz="3.5", dx="-0.3")
VisualModel.addObject('RigidMapping', name="MM->VM mapping",  input="
    @instrumentState",  output="@InstrumentVisualModel")
```

6.3 Using two haptic devices

I would like to later use two devices. To do so, one must simply define two GeomagicDrivers in the root node, and define two LCPForceFeedback with different names for them. Then the same nodes must be created for both (OmniLeft, OmniRight, InstrumentLeft, InstrumentRight). Note: I already wrote the code for this, the only problem is that I couldn't try with 2 devices yet.

Figure 11: Scalpel visual and collision model

7 Incision task models

For this task I needed the scalpel model, which I found online. I imported the obj model of the scalpel and then set a collision point on it. The result is in Fig 11. In particular this is the model with the scalpel without the geomagic. I just want to report the basic parameters. The ones I didn't report are the ones that stay the same from the dental model.

7.1 Behavior

First import the model.

```
scalpelNode.addObject('MeshObjLoader', name='instrumentMeshLoader',
    filename=scalpel_Instrument)
```

Then, to position that object that you loaded with a mesh: use translation and not position as parameter.

```
scalpelNode.addObject('MechanicalObject', src="@instrumentMeshLoader",
    name='mechObject', template='Rigid3d', translation="10 15 60 ",
    scale3d=scale3d_scalpel)
```

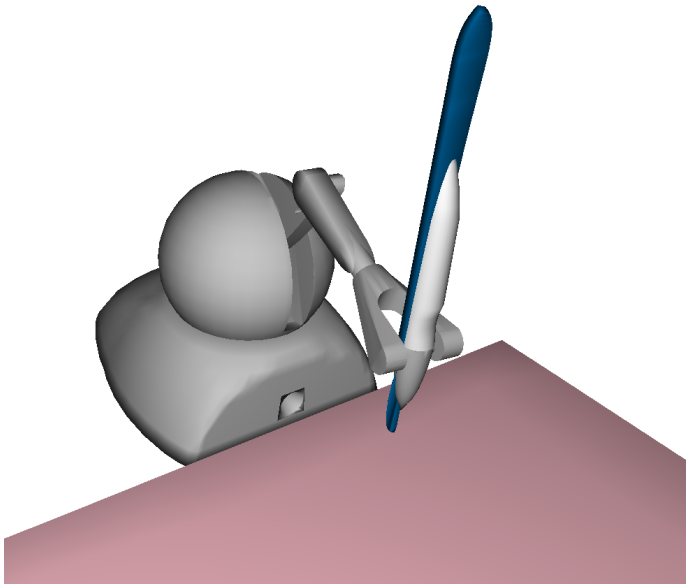


Figure 12: Scalpel attached to geomagic

7.2 Collision

Collision. Had to create multiple particles or they wouldn't move with the scalpel. They all have the following structure.

```
scalpelColNode.addObject('MechanicalObject', template="Vec3d", name="
    Particle", position=pointPosition_onscalpel1)
scalpelColNode.addObject('SphereCollisionModel', radius=r, name="
    ParticleModel", contactStiffness="2", tags="CarvingTool")
scalpelColNode.addObject('RigidMapping', template="Rigid3d,Vec3d", name=
    "MM->CM mapping",  input="@../mechObject",  output="@Particle")
```

Then, when attached to the Geomagic, the object must be positioned and rotated so that it is comfortable to move it with the end effector. A first trial is the one in Fig 12.

7.3 Using SofaCarving

To perform the incision taks the model must be modified, and to do so the SofaCarving plugin may be useful. The plugin provides one component, the CarvingManager. When added to a scene, it automatically looks for a collision model that will acts as the tool from removing tetrahedra, and a surface model that will have its tetrahedra removed by the tool. If your scene has many possible collision or surface models, you

can specify the ones that the CarvingManager will use by giving those components the tags CarvingTool and CarvingSurface.

Note: CarvingSurface: from the examples it must be a triangle/point collision model (also generated from tetra2triangletopologicalmapping, for example). Note: CarvingTool: from the examples it must be a point/sphere collision model.
In code:

```
    # In the root node
    root.addObject('CarvingManager', active="true", carvingDistance="0.1
    ")

    # In the skin node
    skinCollis.addObject('TriangleCollisionModel', tags="CarvingSurface"
    )
    skinCollis.addObject('LineCollisionModel')#, tags="CarvingSurface")
    skinCollis.addObject('PointCollisionModel')#, tags="CarvingSurface")

    # In the scalpel node
    scalpelColNode1.addObject('PointCollisionModel', name="
    ParticleModel1", contactStiffness="1", tags="CarvingTool")
```

Note that i chose to use point collision on the scalpel and triangle collision on the skin, since the other methods simply did not work.

Problem 1: with 4 different collision particles, sofa crashes. I reduced the number now, but I'll have to deal with this problem one day.
After 'solving' this problem, sofacarving worked, but visually it was not good, since tetrahedra were too big. For this reason I changed the model: I did a bigger box, and then scaled it smaller, so that tetrahedra are smaller. However, this model is visually bad too (Fig 13).

New idea: try to implement what I found in the literature, which is a mass spring model in which i remove springs after collision. The mss model is easy to implement:

```
skin.addObject('MeshSpringForceField', name="FEM-Bend", template="Vec3d
    ", stiffness="100", damping="1")
```

The difficulty is in the detection of the springs that have collisions, their removal and eventually remeshing. Look at link1 and link2

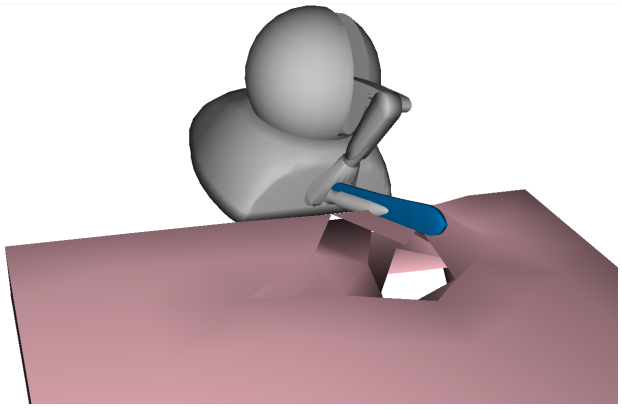


Figure 13: Incision with sofacarving

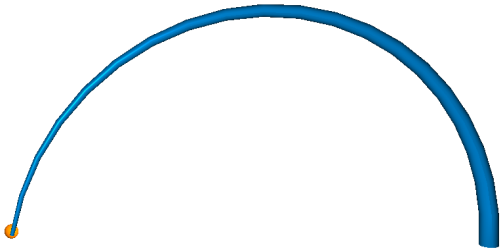


Figure 14: Needle visual and collision model

8 Suture task models

For this task I needed the needle model, which I found online. I imported the obj model of the needle and then set a collision point on it. The result is in Fig 14. In particular this is the model with the needle without the geomagic.

8.1 Collsion

To create only one point of collision: create a mechanical object with a desired position.

```
needleColNode = needleNode.addChild('CollisionModel')
needleColNode.addObject('MechanicalObject', template="Vec3d", name="
    Particle", position=pointPosition_onNeedle)
```

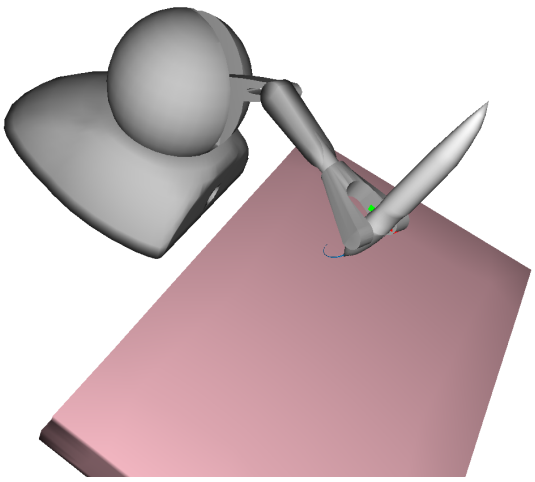


Figure 15: Needle attached to geomagic

```
needleColNode.addObject('PointCollisionModel', name="ParticleModel",
    contactStiffness="2")
needleColNode.addObject('RigidMapping', template="Rigid3d,Vec3d", name="
    MM->CM mapping",  input="@../mechObject",  output="@Particle")
```

Attaching the needle to the geomagic, what changes is the behavior model: the mechanical object must be positioned on the geomagic, and some more parameters must be added.

A first trial is the one in Fig 15. Note: when loading the model on Sofa, you can see the needle attached to the geomagic only in animation mode, I believe it is because, when the model is only leaded, the position parameter with value "@GeomagicDevice.positionDevice" is not evaluated.

9 Conclusion

This is a work in progress, now I'll concentrate on the tasks, starting from incision.

References

[1] *Issues in the haptic display of tool use*. 1995. URL: https://ieeexplore.ieee.org/document/525875.

[2] *Three 6-DOF Haptic Algorithms Compared for Use in a Milling Surgery Simulator Prototype*. 2012. URL: https://www.annualreviews.org/doi/abs/10.1146/annurev-control-060117-105043.