:: SOFA ::
Collision Pipeline

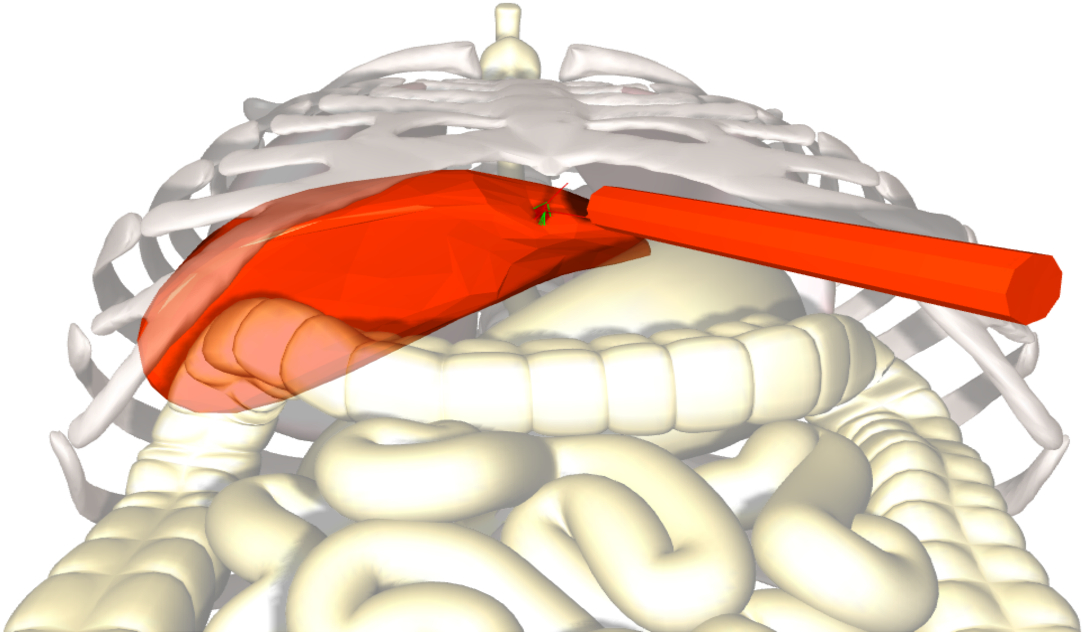
Jérémie Allard
INRIA - Alcove
http://sofa-framework.org

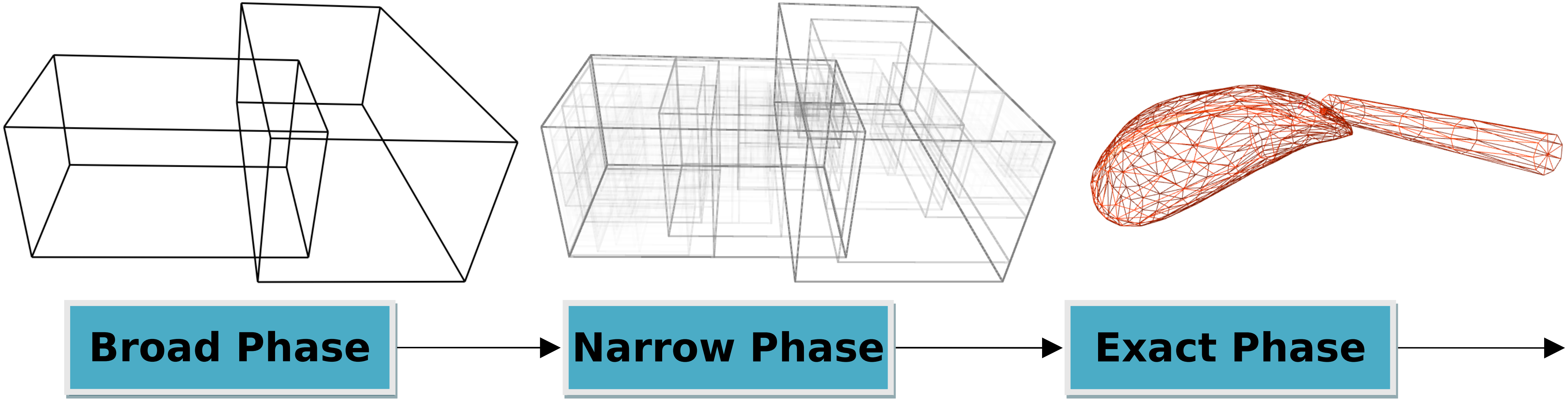
Outline

- Overview
- Current Design and Implementation
  - **Changes for SIGGRAPH**
- Current Issues
- Future Directions

Collision Pipeline Overview (1)

- Detection

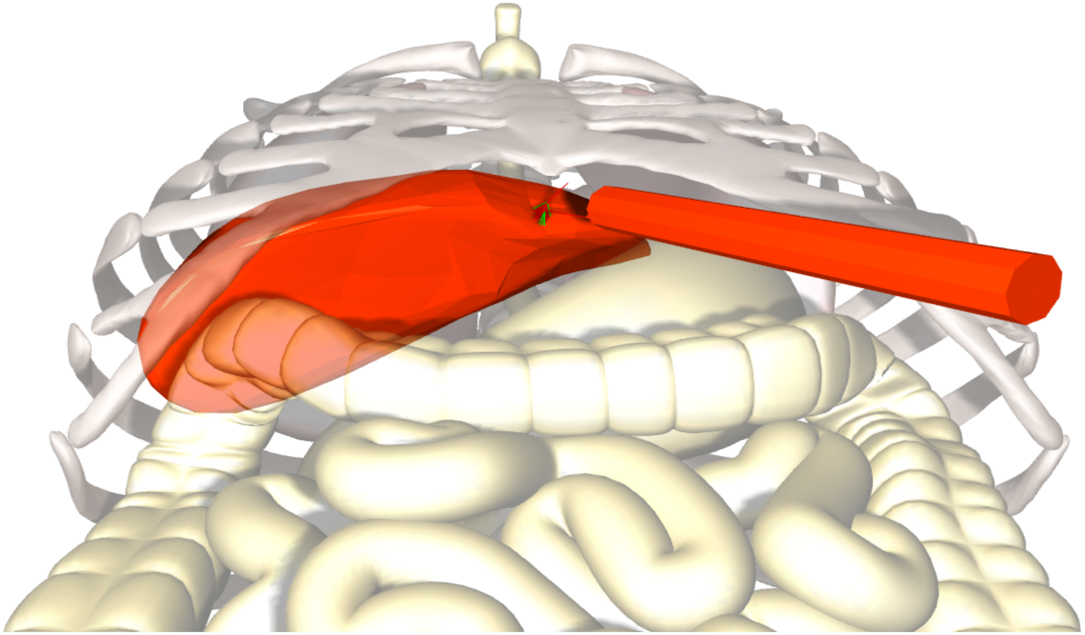


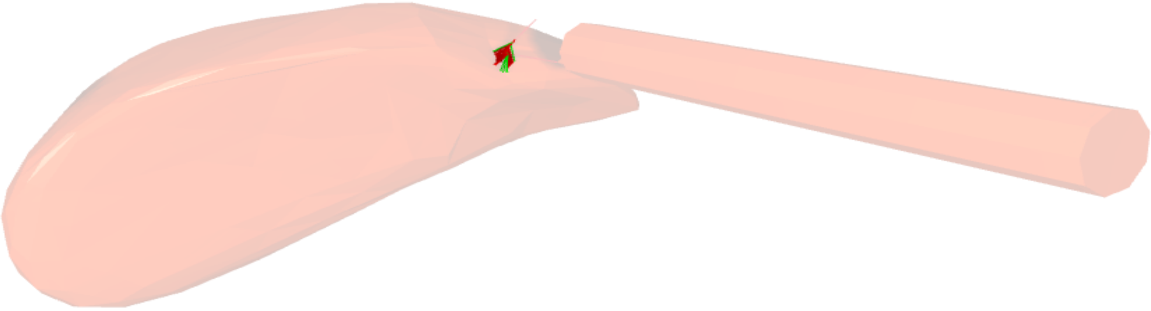


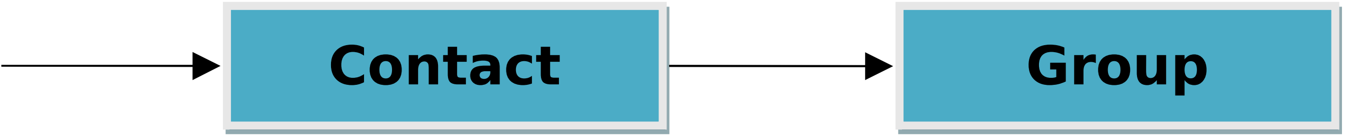
Broad Phase → Narrow Phase → Exact Phase →

Collision Pipeline Overview (2)

- Response







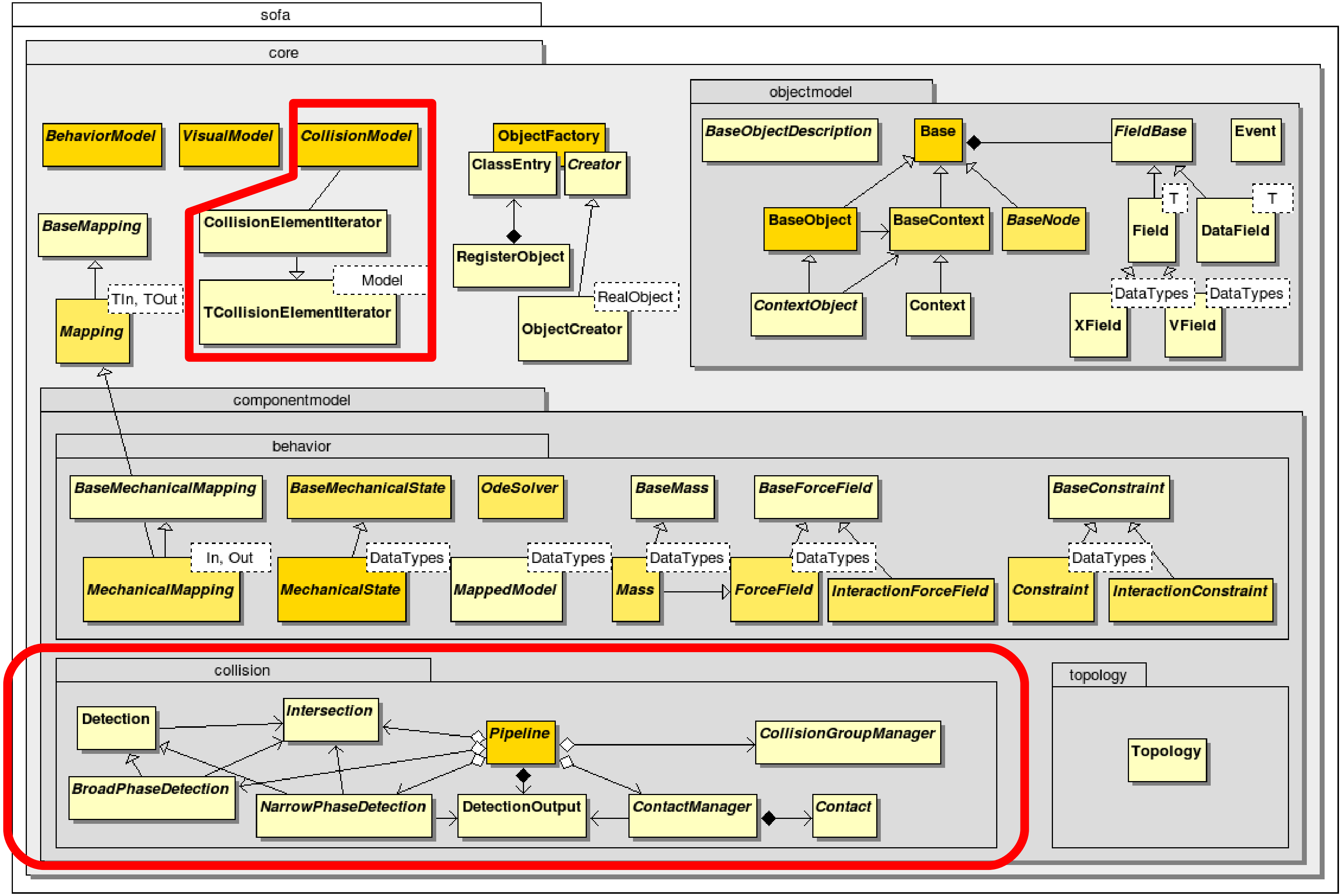
Contact

Group

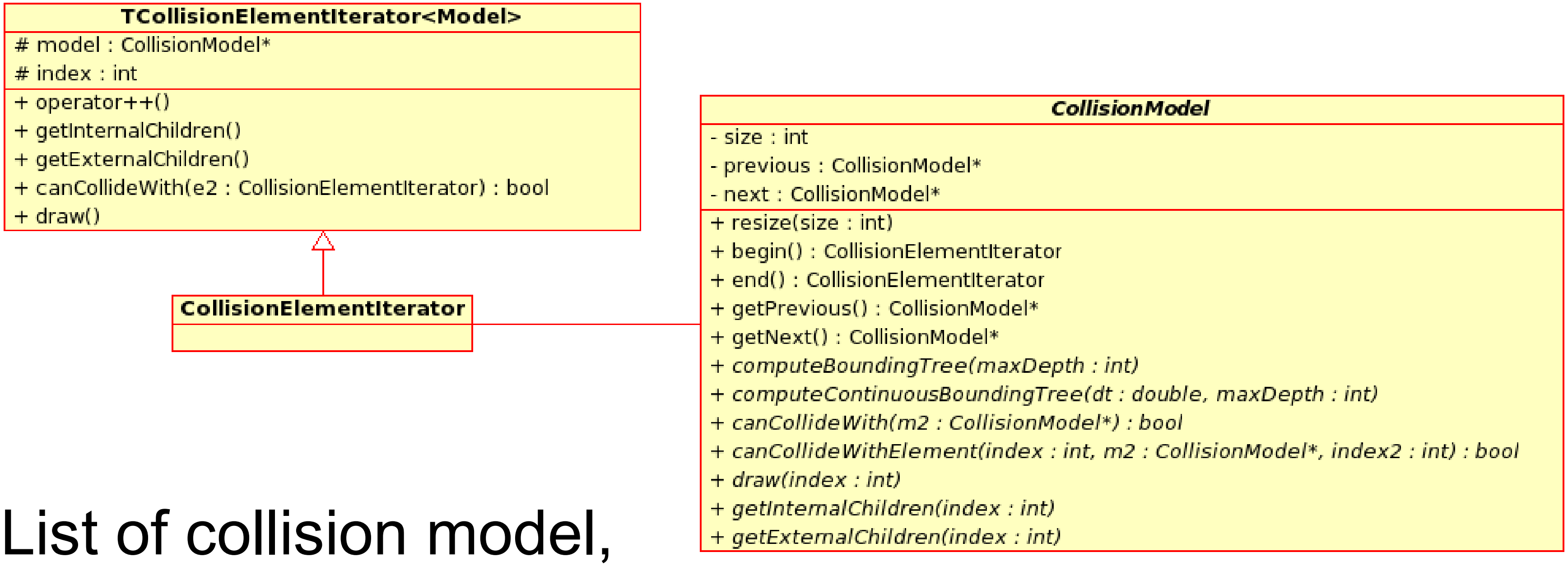
Collision Components

- Collision Models
  - Sphere, Triangle, Line, Point, **Distance Field**, ...
- Bounding Tree
  - AABB-Tree, Sphere-Tree
- Intersection Method
  - Discrete, Continuous, Proximity
- Detection Algorithm
  - Brute-Force, Hierarchical, **GPU-based**
- Contact Manager
  - Penality, Lagrange-Multiplier, Constraints, Custom Interactions

High-Level Design



Design : Collision Models



TCollisionElementIterator<Model>

model : CollisionModel*
index : int

+ operator++()
+ getInternalChildren()
+ getExternalChildren()
+ canCollideWith(e2 : CollisionElementIterator) : bool
+ draw()

CollisionElementIterator

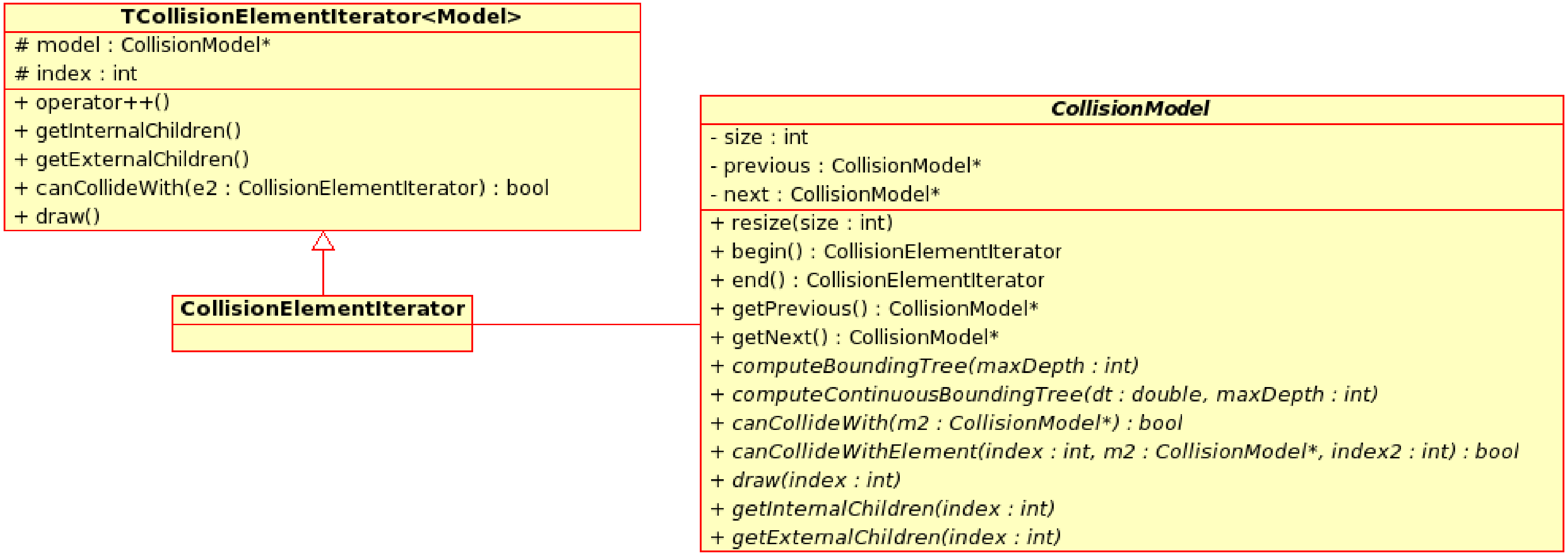
CollisionModel

- size : int
- previous : CollisionModel*
- next : CollisionModel*

+ resize(size : int)
+ begin() : CollisionElementIterator
+ end() : CollisionElementIterator
+ getPrevious() : CollisionModel*
+ getNext() : CollisionModel*
+ *computeBoundingTree(maxDepth : int)*
+ *computeContinuousBoundingTree(dt : double, maxDepth : int)*
+ *canCollideWith(m2 : CollisionModel*) : bool*
+ *canCollideWithElement(index : int, m2 : CollisionModel*, index2 : int) : bool*
+ *draw(index : int)*
+ *getInternalChildren(index : int)*
+ *getExternalChildren(index : int)*

- List of collision model, each corresponding to a certain level in the hierarchy
  - First : root of the hierarchy, contains only one element
  - Last : leaves of the hierarchy, final elements
- Each CollisionModel contains a list of elements accessible using CollisionElementIterators
  - begin(), end(), operator++()

Design : Collision Elements



TCollisionElementIterator<Model>

model : CollisionModel*
index : int

+ operator++()
+ getInternalChildren()
+ getExternalChildren()
+ canCollideWith(e2 : CollisionElementIterator) : bool
+ draw()

CollisionElementIterator

CollisionModel

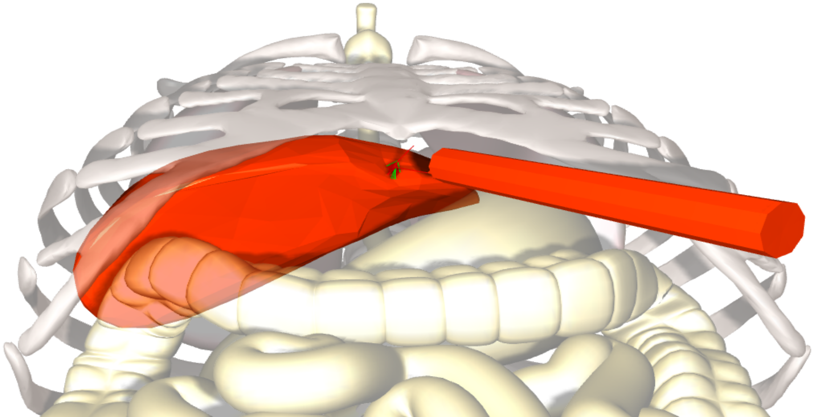
- size : int
- previous : CollisionModel*
- next : CollisionModel*

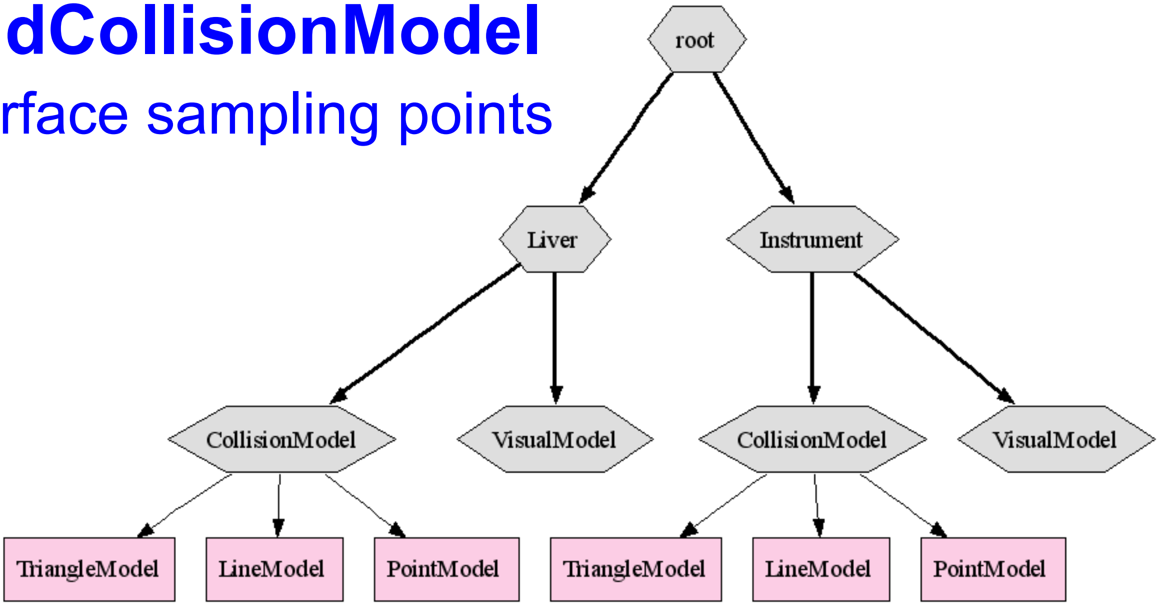
+ resize(size : int)
+ begin() : CollisionElementIterator
+ end() : CollisionElementIterator
+ getPrevious() : CollisionModel*
+ getNext() : CollisionModel*
+ *computeBoundingTree(maxDepth : int)*
+ *computeContinuousBoundingTree(dt : double, maxDepth : int)*
+ *canCollideWith(m2 : CollisionModel*) : bool*
+ *canCollideWithElement(index : int, m2 : CollisionModel*, index2 : int) : bool*
+ *draw(index : int)*
+ *getInternalChildren(index : int)*
+ *getExternalChildren(index : int)*

- Each non-final element can have a list of children :
  - Internal children: child elements of the same type as their parent (often corresponding to non-final elements)
  - External children: child elements of a different type (often corresponding to the final elements)
- Derived classes add methods to access other data fields

Implementation : Collision Models

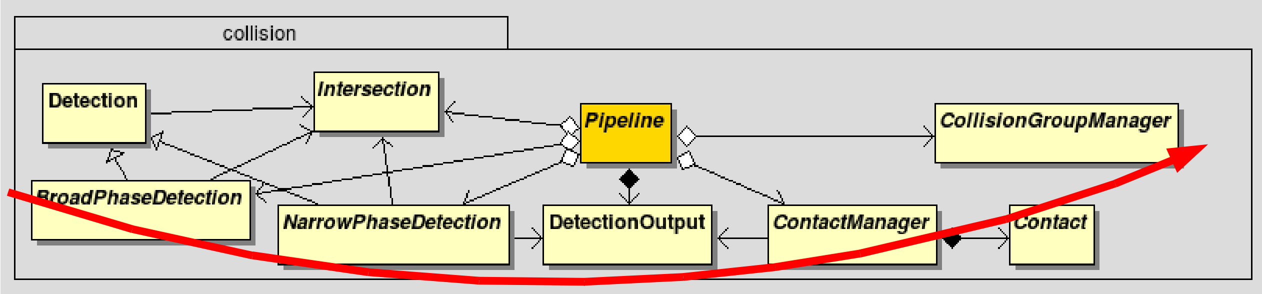
- **CubeModel** : AABB hierarchy, cannot be final
- **SphereModel**, (SphereTreeModel)
  - Derive from MechanicalObject
- **PointModel**, **LineModel**, **TriangleModel**
  - Pieces of a surface mesh, for simple Line/Line & Point/Triangle tests
  - Flags added in TriangleModel to remove redundant computations in Triangle/Triangle tests
- **(Rigid/FFD)DistanceGridCollisionModel**
  - Uniform distance fields + surface sampling points
  - GPU-accelerated

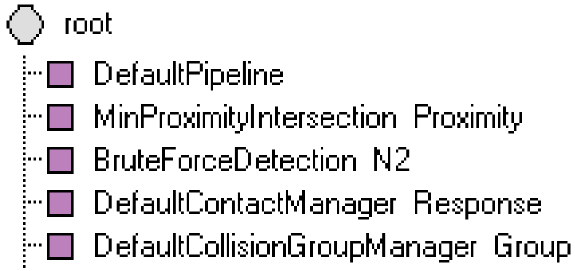




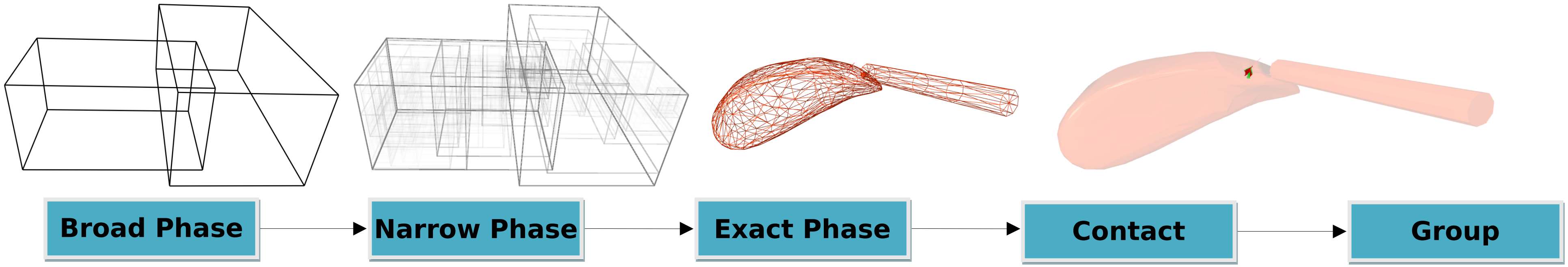
Design : Collision Pipeline

- Each piece of the pipeline is added to the scene root
  - No support for different algorithms in parts of the scene
- Pipeline component gather list of collision models and control the sequence of computations





root
- DefaultPipeline
- MinProximityIntersection Proximity
- BruteForceDetection N2
- DefaultContactManager Response
- DefaultCollisionGroupManager Group



Implementation : Collision Pipeline

- **DefaultPipeline**
  - The one and only

Design : Detection Outputs

- Describe results of collision detection
- **DetectionOutput** : Generic description of a contact point
  - ***elem***: pair of colliding elements
  - ***id***: unique id of the contact for the given pair of collision models.
  - ***point***: contact points on the surface of each model.
  - ***normal***: normal of the contact, pointing outward from the first model.
  - ***distance***: signed distance (negative if objects are interpenetrating).
  - ***deltaT***: estimated of time of contact.
- DetectionOutputVector : previously a *std::vector*
  now an abstract class
  - Support other contact point description (free motion, bary coords, ...)
  - Support other memory container (GPU)
  - Template TDetectionOutput<CM1,TM2> class specify which format is used for each combination of collision models

Design : Intersection Methods

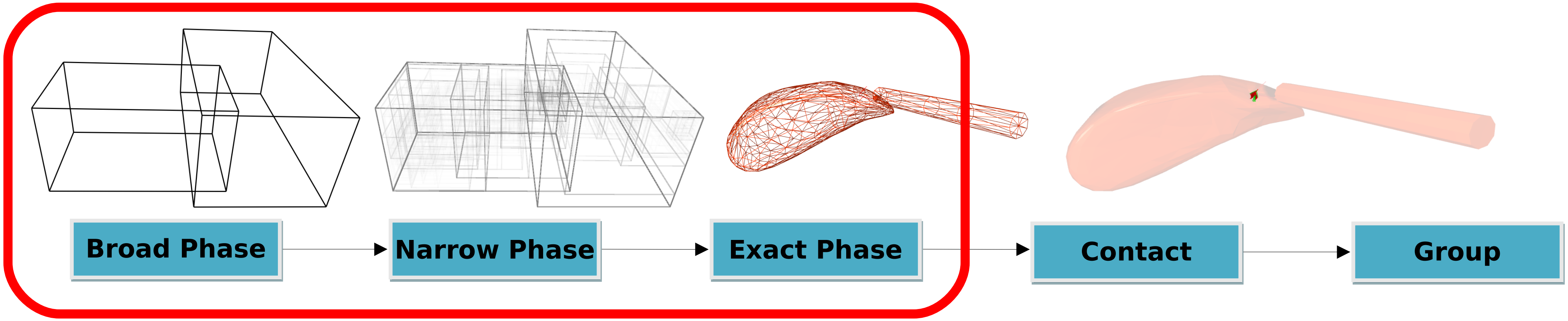
- Given 2 collision elements, test if an intersection is possible (for bounding volumes), or compute intersection points if any
- Implementation depends on the type of both collision models
- **Intersection::findIntersector()** allow to resolve once which implementation (**ElementIntersector**) to use and reuse it for all tests between a pair of models.
  - This method is now mandatory
- Results are stored in a given container, instead of dynamic memory allocations for each point
- Multiple contact points can now be returned by each test

Implementation : Intersection

- **DiscreteIntersection** : basic tests
  - Cube/Cube, Sphere/*, Ray/* (mouse), DistanceGrid/*
  - no mesh-mesh
- **ContinuousIntersection**
  - Triangle/Triangle only, currently no support for response
- **ProximityIntersection** : proximity based on LCP
  - All, including full Triangle/Triangle
- **MinProximityIntersection** : min proximity can only be between a Point/Triangle or a Line/Line
  - Faster
- **NewProximityIntersection** : use new flags to remove redondant tests in Triangle/Triangle, ignore Line/Line
  - Fastest, but only for well tesselated meshes
  - PointModel and LineModel no longer required, 3 times less BB trees

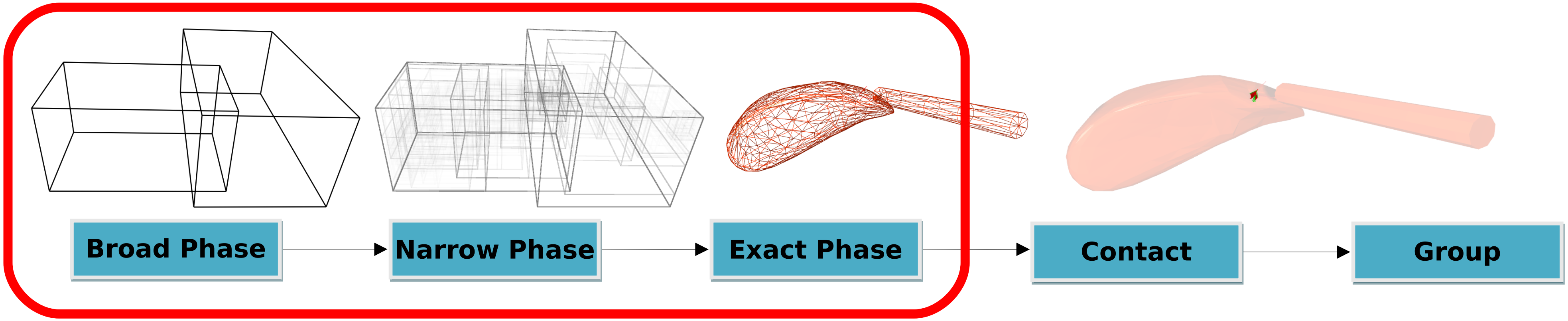
Design : Detection

- **BroadPhaseDetection** : given a set of root collision models, computes potentially colliding pairs
- **NarrowPhaseDetection** : given a set of potentially colliding pairs of models, compute set of contact points
  - Internally execute the "*Exact Phase*" instead of storing the potentially much bigger list of possible colliding element pairs.
- Only schedule / organize the tests, use an **Intersection** class for computations



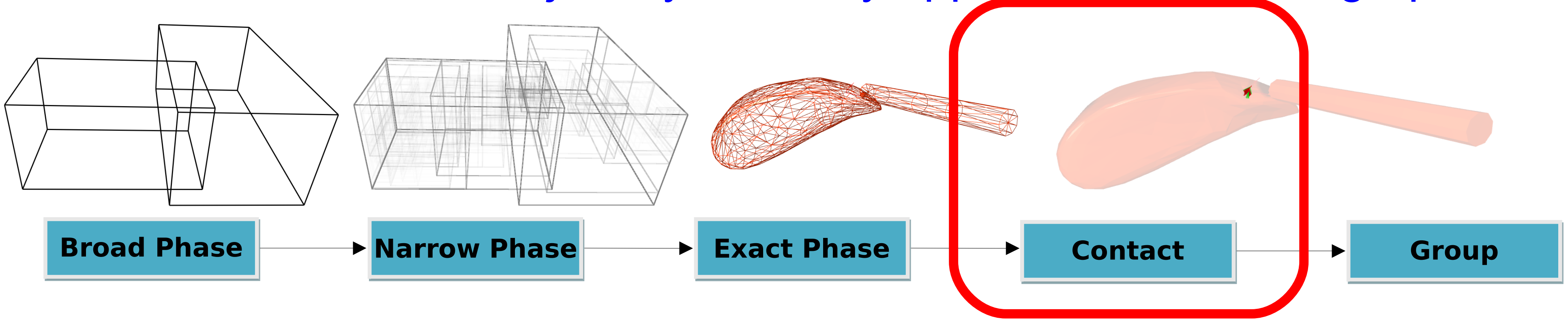
Implementation : Detection

- **BruteForceDetection** :
  - Brute force $O(n^2)$ for broad phase
  - Hierarchical pruning for narrow phase
- **CudaCollisionDetection** :
  - GPU support
  - Launch GPU tests in parallel to CPU tests
  - Only spheres and distance-grids are supported on the GPU for now



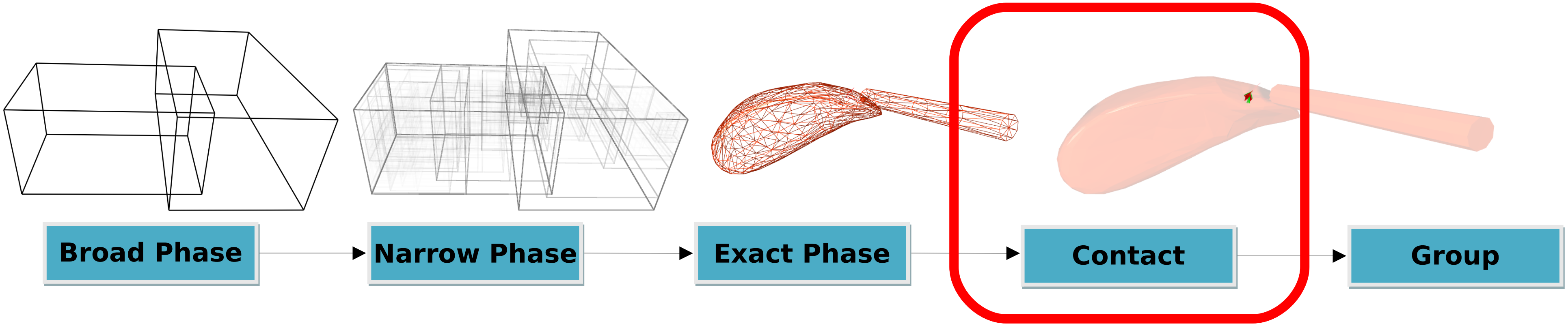
Design : Contacts

- **ContactManager** : given a set of detected contact points, create contact response components
- **Contact** : contact response component handling the response between a pair of models
  - Dynamically created by the ContactManager
    - Persistent between iterations
    - New id data in DetectionOutput allow to keep an history of a contact
  - In most cases : create and initialize the real response component
    - InteractionForceField, Constraint, ...
  - Now the Contact object dynamically appears in the scenegraph



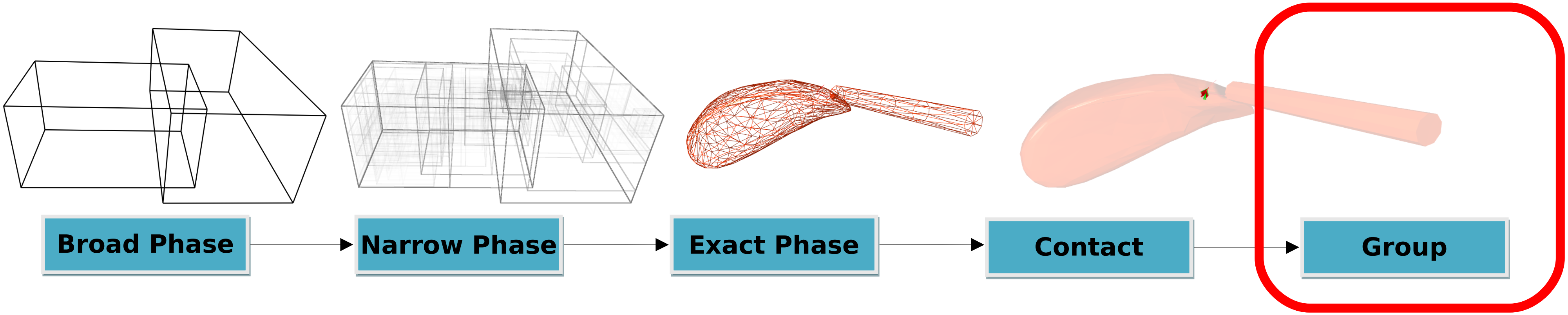
Implementation : Contacts

- **DefaultContactManager** : use a factory to create Contact instances given a pair of collision models and a string
  - “default” : penality forces
- **BarycentricPenalityContact**
  - Create repulsive springs
  - If necessary map the contact points to the original DOFs
    - Need a BarycentricContactMapper<> implementation for each possible CollisionModels
- **RayContact**
  - Handle mouse interactions



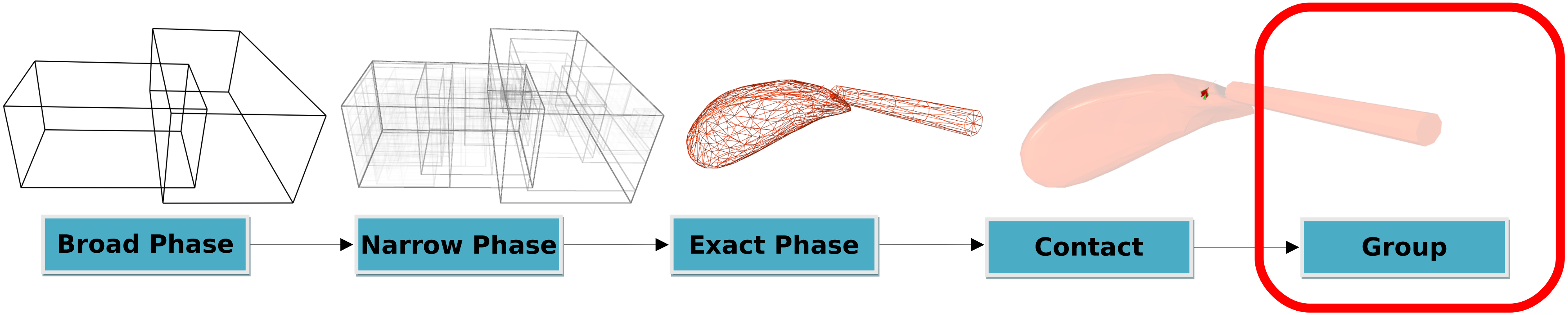
Design : Collision Groups

- **CollisionGroupManager** : given a set of contacts, create integration groups
  - Contacts between models defines a graph
  - “Simply” gather connected subgraphs
  - Decide which integrator/solver algorithms will be used



Implementation : Collision Groups

- **DefaultCollisionGroupManager**
  - For each pair of objects in contacts :
    - Look which mechanical integration algorithm is used
    - If they are “compatible”, create a algorithm merging them
      - Often simply the most stable of the two
        - Explicit Euler + Explicit Runge Kutta -> Explicit Runge Kutta
        - Explicit * + Implicit Euler -> Implicit Euler

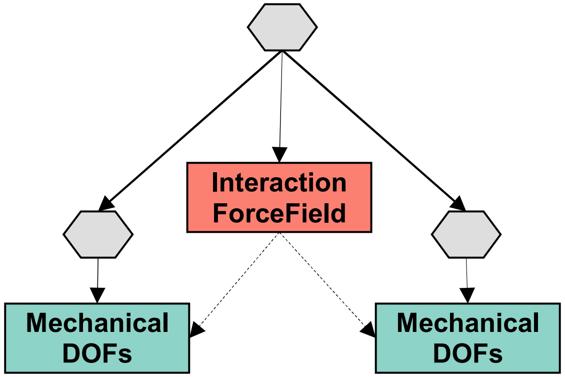


Finally

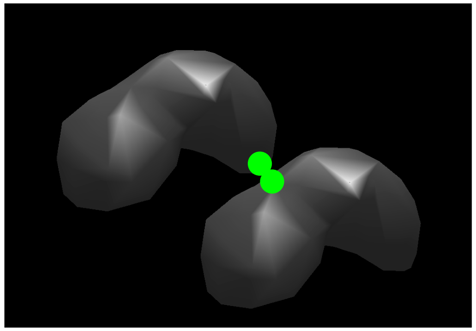
- The Pipeline is complete !
- Next is (usually) the mechanical integration step

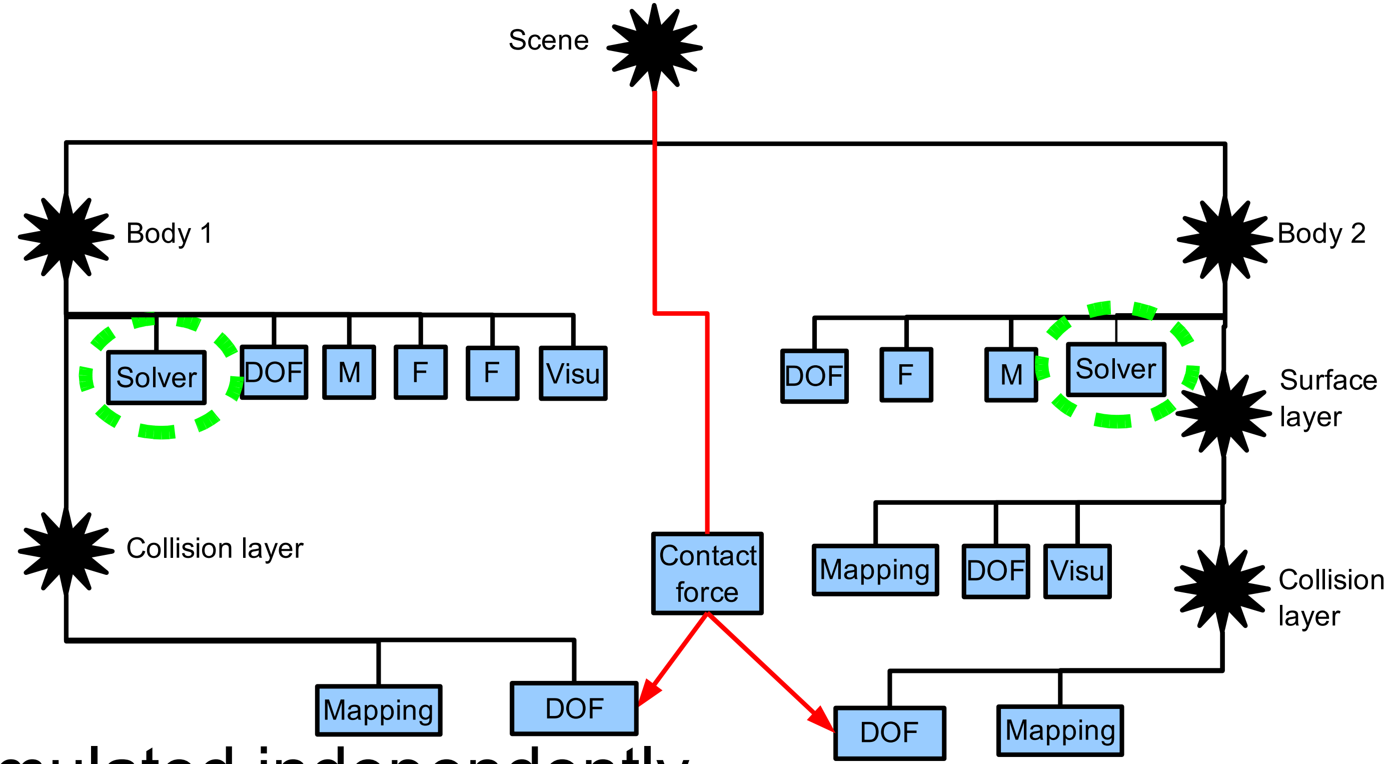
Contacts in the Simulation

- Contacts dynamically change the scene structure
  - Add new forcefields or constraints between objects
  - Change integration groups (implicit stiff interaction forces, global constraints solvers)
- Interactions create loops in the graph
  - InteractionForceFields point to the 2 involved MechanicalDOFs
  - Attached to the first common ancestor node



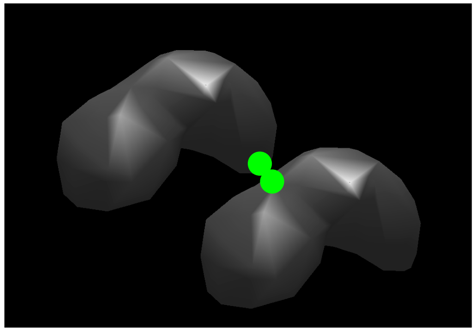
Contact Example

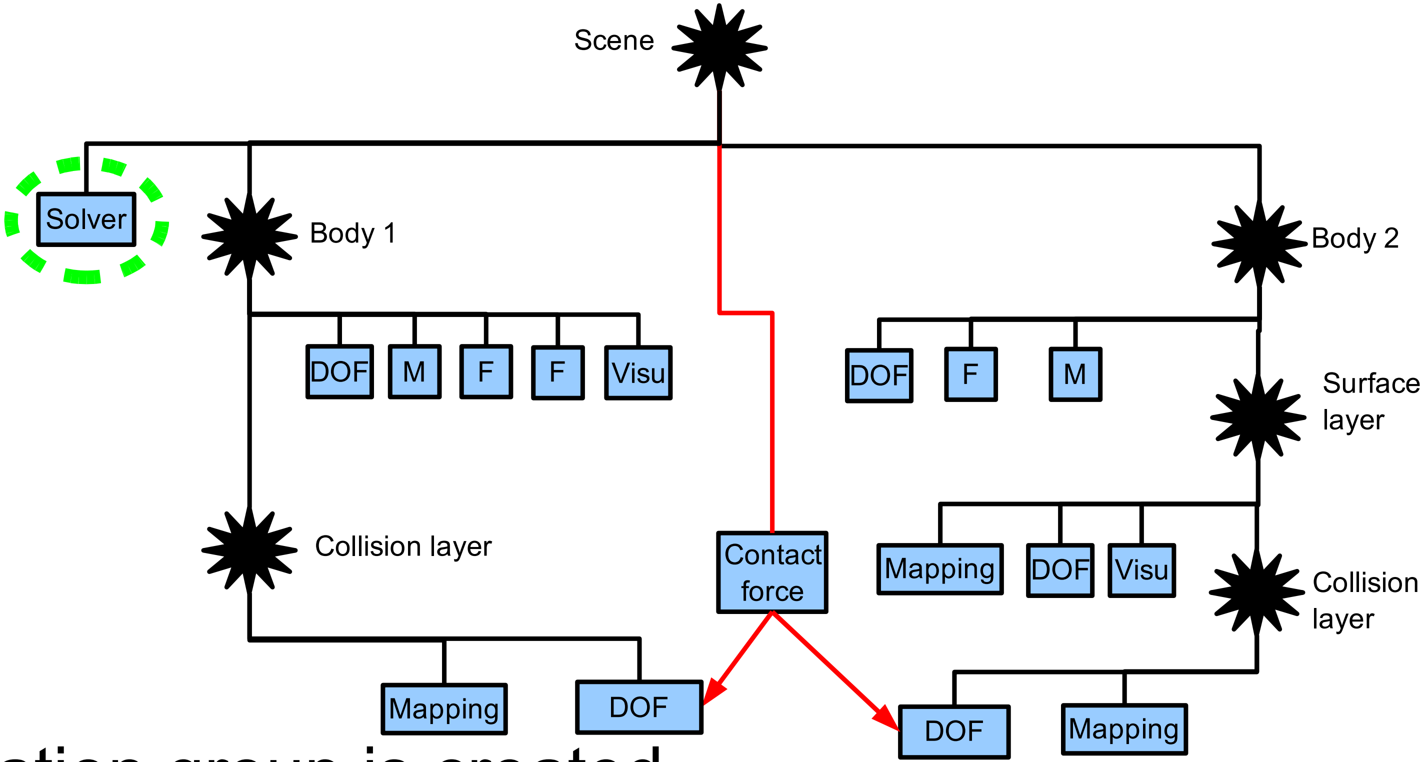




- Bodies are simulated independently
  - Contact force computed once per time step

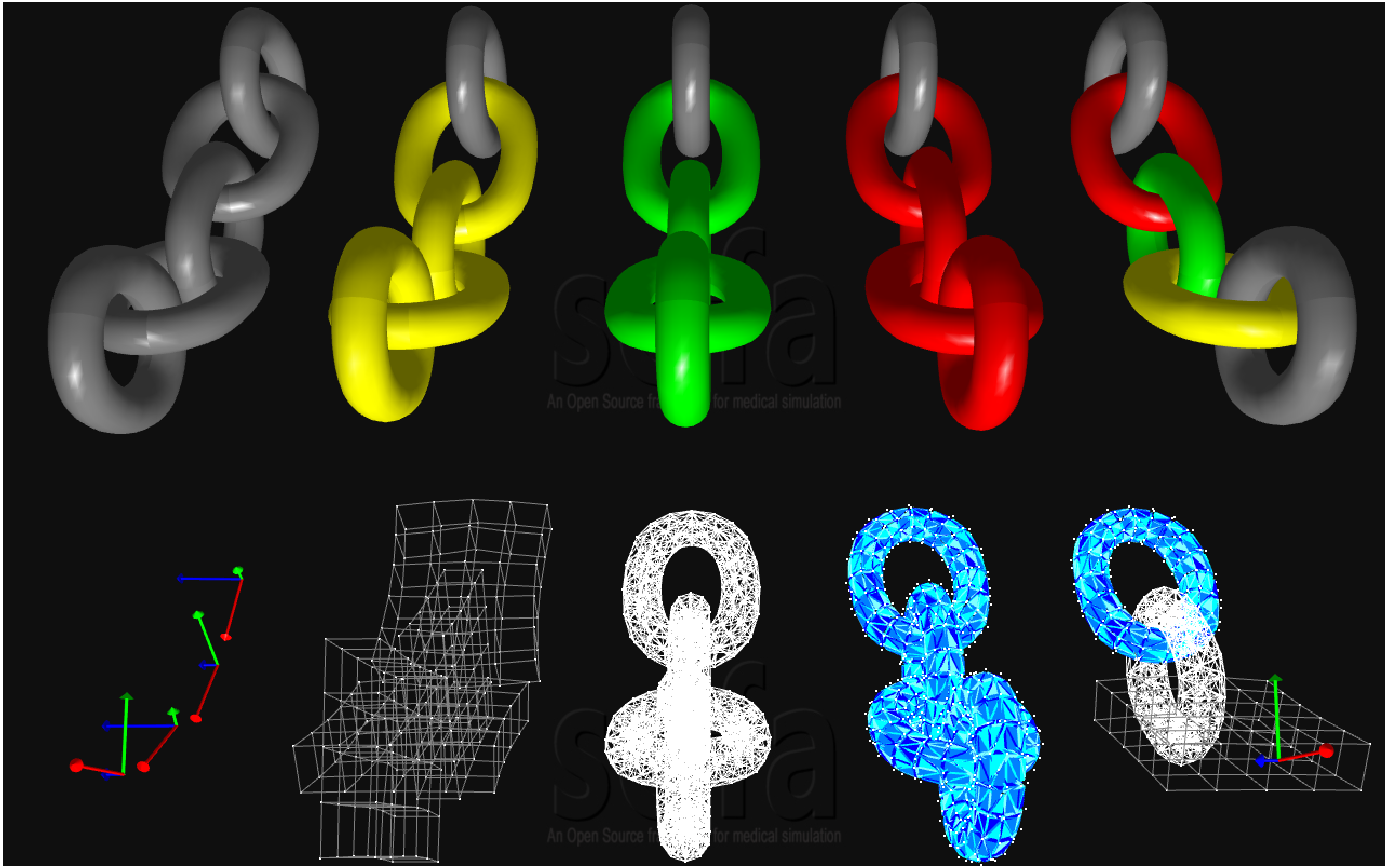
Stiff interactions





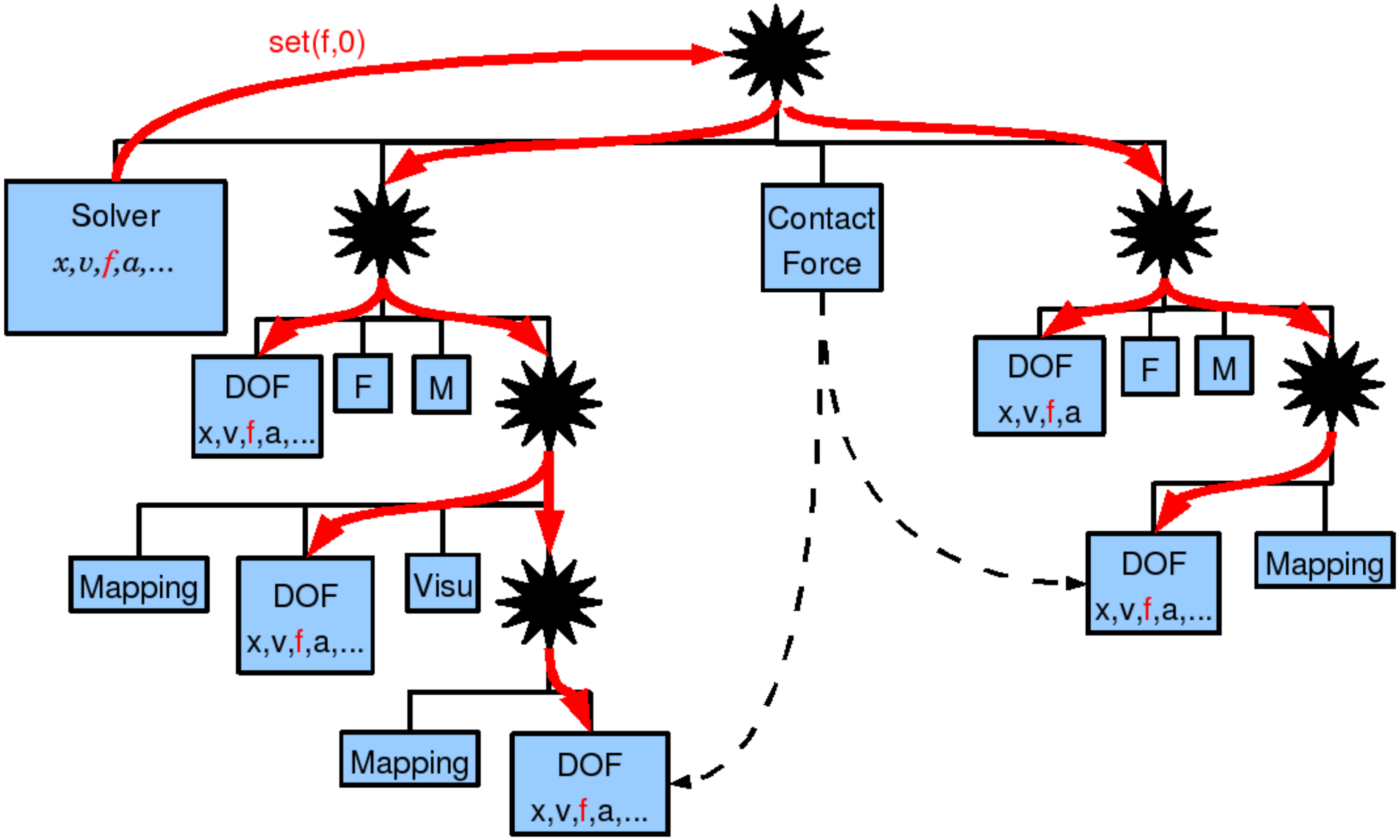
- A new integration group is created
  - Contact force can now be evaluated implicitly

Validation of implicit contacts



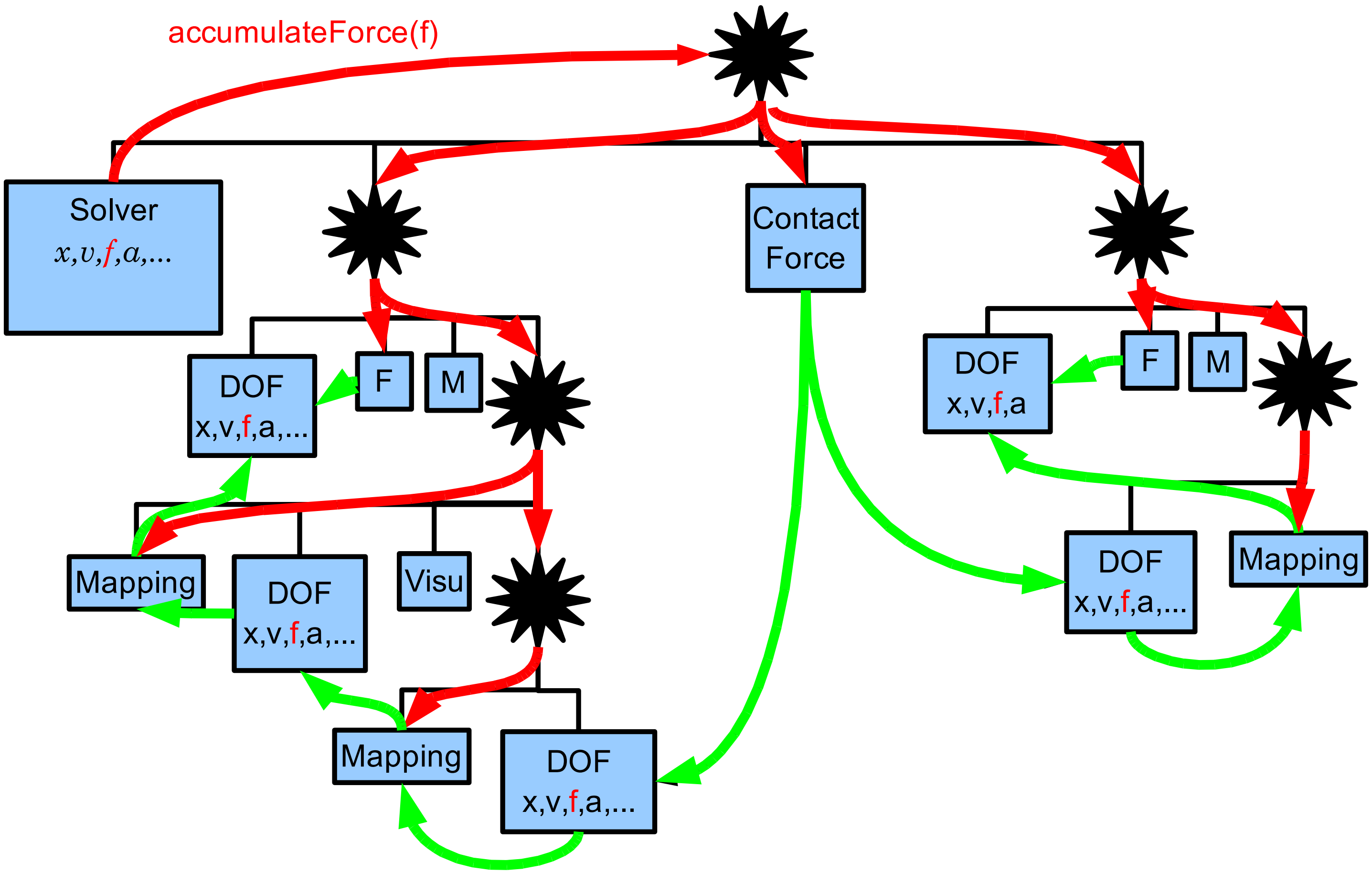
Interaction Example

- Computing forces: (1) f = 0



Interaction Example

- Computing forces: (2) accumulate forces



Current Issues

- Many stages have only one implementation
  - More are coming (see presentations on constraints and GPU)
  - Need to validate that we have enough flexibility
- Only one intersection/detection/... algorithm acting on the whole scene
- Everything is recomputed at each iteration
  - Need to be able to update a given subset of the contacts
  - Using time consistency could increase the speed
- Adding a new CollisionModel requires adding code in many places
  - Intersection algorithms, instantiation of Contact classes, ContactMapper specialization

Future Work

- Add support for different algorithms for subgroups
  - Rigids, Deformables, Rigids-Deformables, ...
  - Visible/near objects vs hidden/far objects
- Stabilize and validate API to describe contact points
- Parallelization
- Link to external “optimized” collision detection libraries