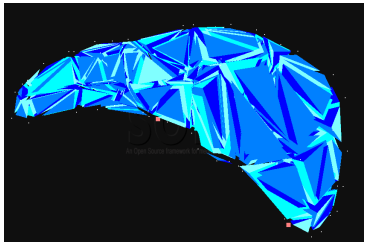
SOFA :
Design for Parallel Computations

Jérémie Allard

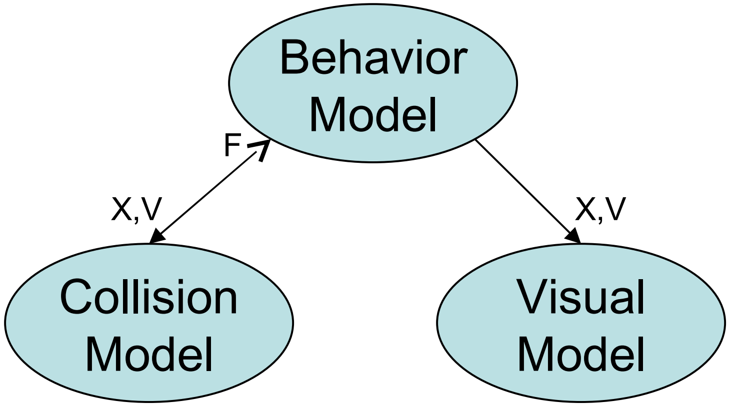
SOFA

- Goal: interactive deformable objects simulation platform
- Integrate as many simulation algorithms as possible
  - Rigid bodies, mass-springs, finite-element models, fluids, articulated bodies, …
  - Implicit/explicit/static solvers, penality/constraint collision response, stiff interactions, …

SOFA: Basic Principles



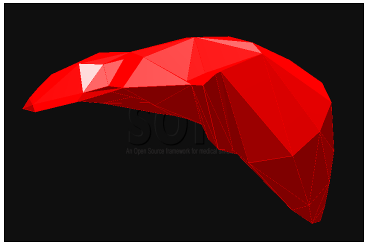
- Each object has several aspects
  - Behavior Model
  - Collision Model
  - Visual Model

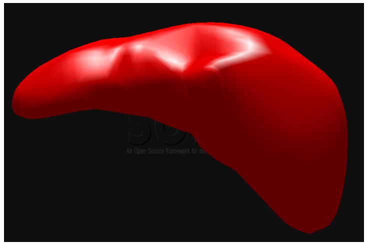


Behavior Model

Collision Model

Visual Model





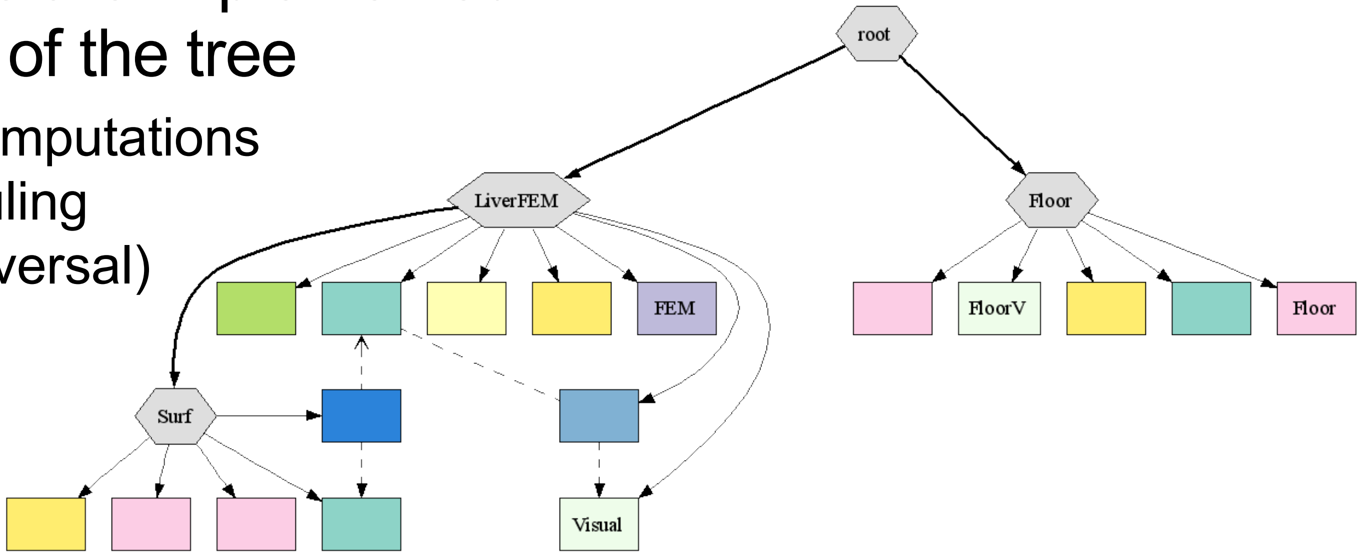
- *Mappings* are used to link them
  - BM → VM/CM : propagate positions and velocities
  - CM → BM : send back forces

Behavior Model

- 2 possible designs
  - “black-box” single element
    - No knowledge of the internal algorithm of each object
    - Exchange interaction forces at each time-step
    - Similar to *FlowVR Interact*
  - “white-box” aggregation
    - *MechanicalModel : Degree-of-freedoms (DOF)*
    - *Attached elements : Mass, ForceField, Constraint, Solver, …*
    - Internal *MechanicalMappings* to map new representations to the original DOFs
      - Attach a set of points on a rigid body as if it was a mass-spring object
      - Embed a surface inside a deformable FFD grid
      - Implement interactions between 2 types of objects by mapping them to a common representation whenever possible

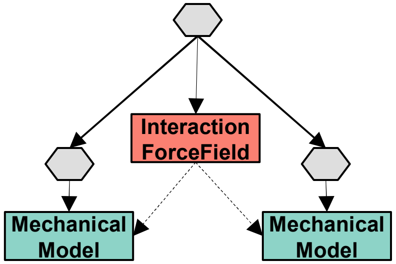
Scene structure

- Scene-graph design
- All data are in leaf elements (*Objects*)
  - Internal elements (*Nodes*) contains only pointers to attached objects and child nodes
- Computations are implemented as traversals of the tree
  - Separate computations from scheduling (order of traversal)



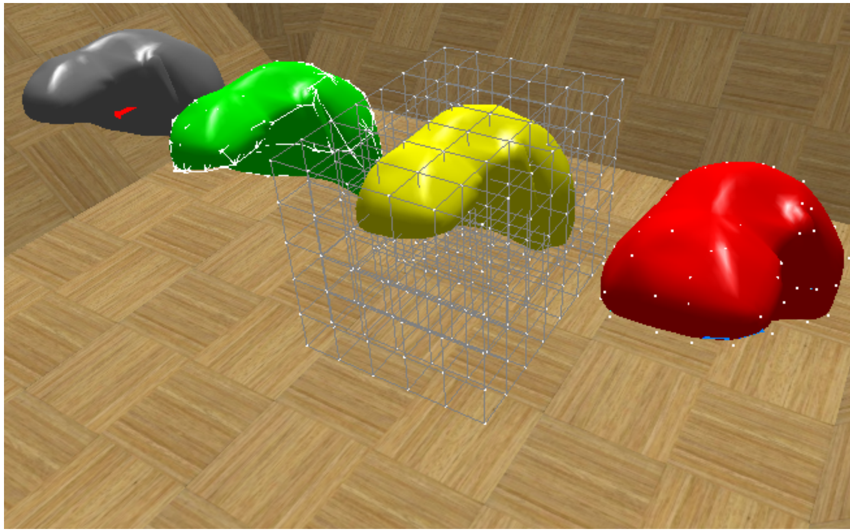
Collisions and Interactions

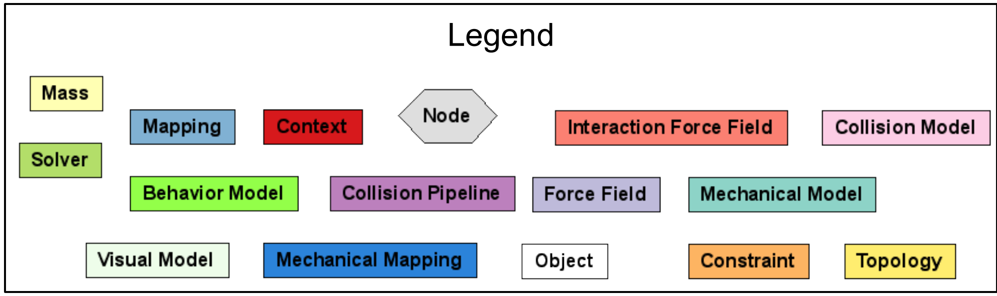
- Generic collision pipeline
  - Compute set of contacts between collision models
  - Change the scene structure dynamically
    - Add new forcefields or constraints between objects
    - Change integration groups (implicit stiff interaction forces, global constraints solvers)
- Interactions create loops in the graph
  - *InteractionForceFields* point to the 2 MechanicalModels involved
  - Attached to the first common ancestor node
    - *Except if one of the model is immobile (such as static obstacles), in which case it is attached to the other model*

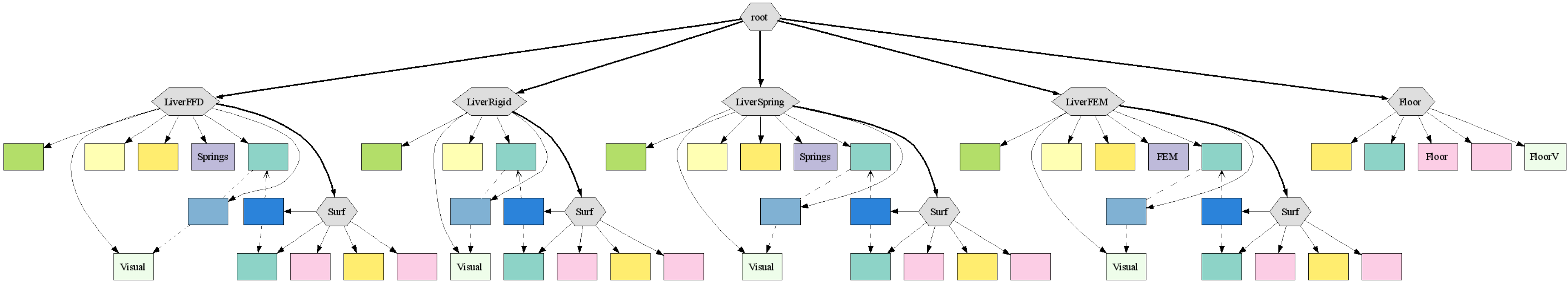


Example

- 4 objects falling on the floor
  - 1 Rigid
  - 1 Mass-spring
  - 1 FFD spring grid
  - 1 FEM
- Each have an mapped collision surface

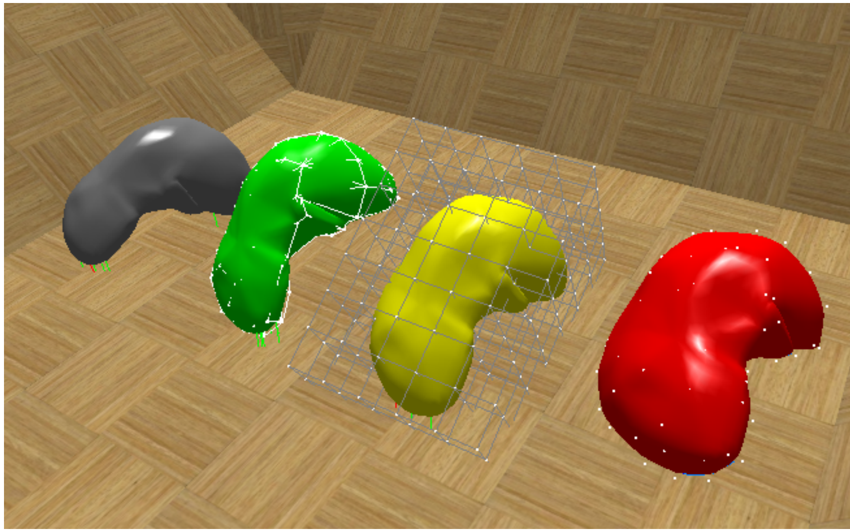


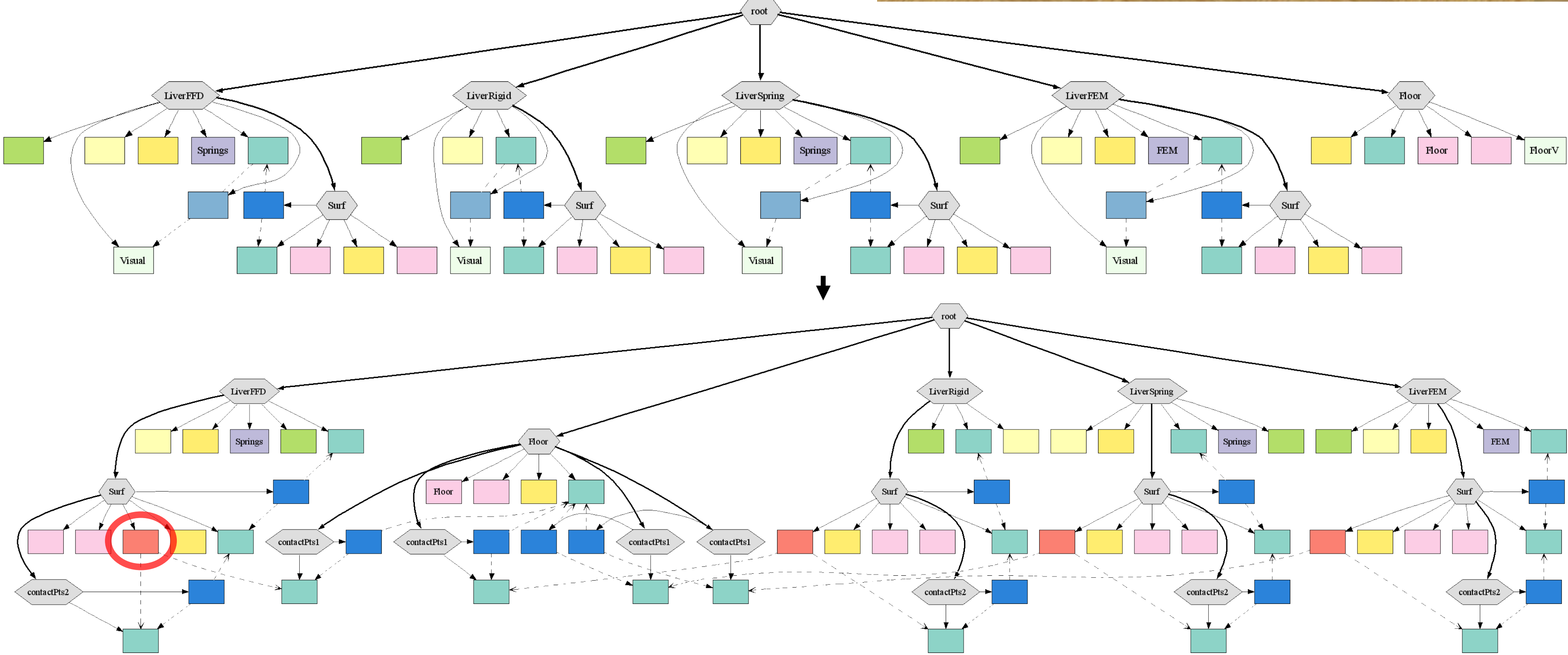




Example (2)

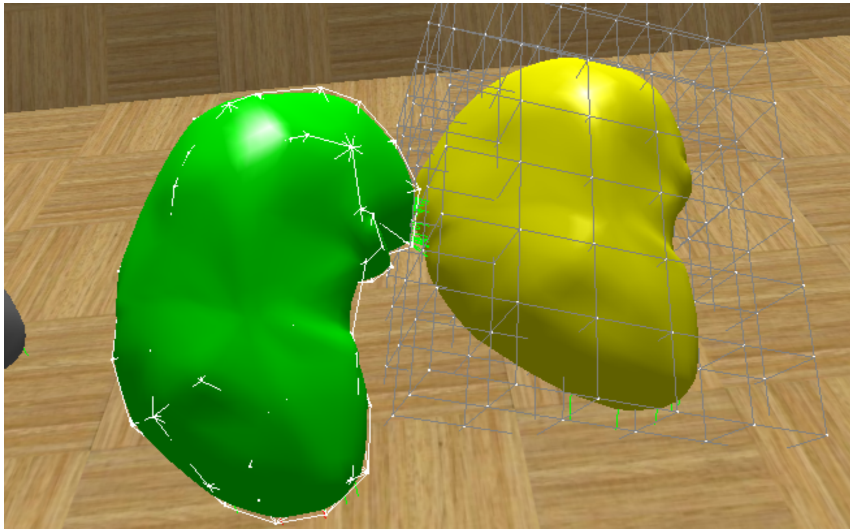
- Contacts with the floor
  - New nodes containing contact points
  - New InteractionForceFields

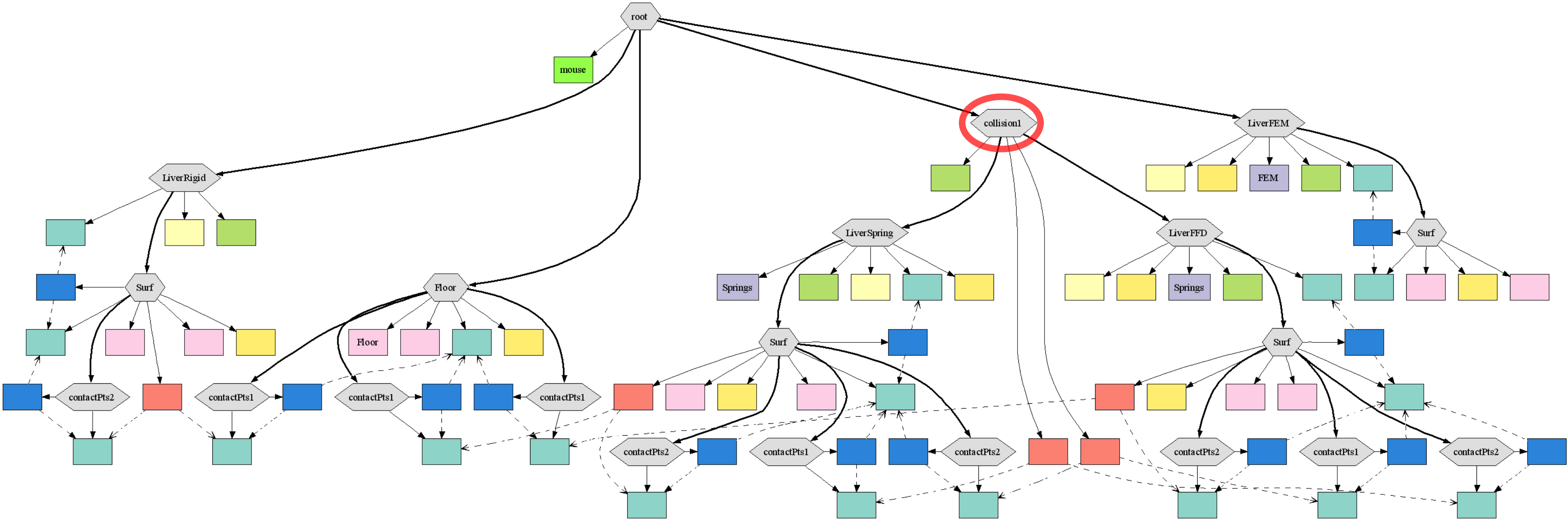




Example (3)

- Contacts between objects
  - Hierarchy changed to group connected objects under one solver





Computations

- Each computation is implemented as an *Action* executed from a given graph node
  - called recursively for each node
    - *processNodeTopDown* called before recursion to child nodes
    - *processNodeBottomUp* after
  - At each node, it can:
    - Ask to be called recursively on each child
    - Stop the recursion
    - Execute other actions from that node

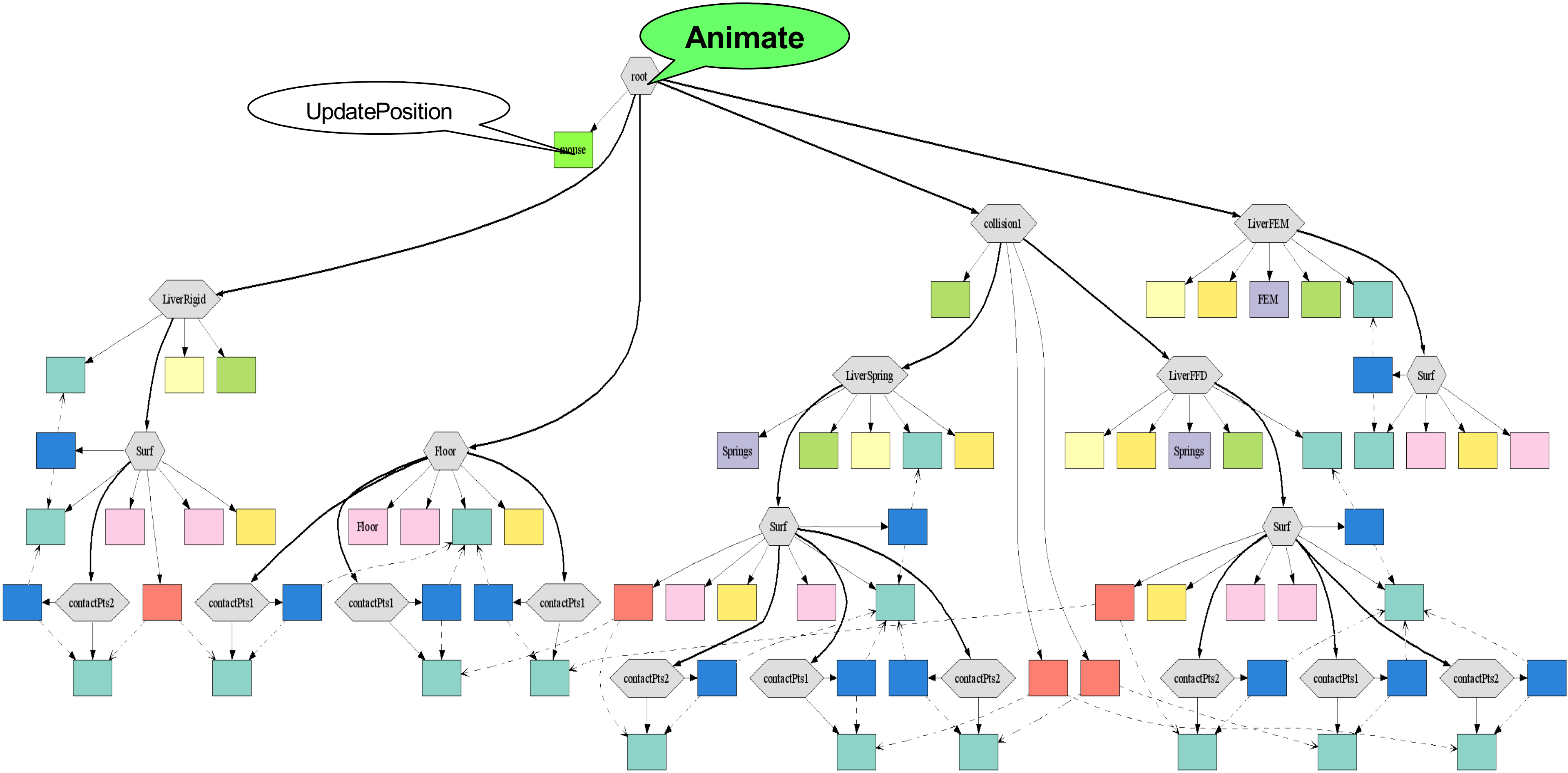
Computations (2)

- Data dependencies rules:
  - processNodeTopDown: read access to all parent nodes, read/write access to current and all child nodes
  - processNodeBottomUp: read access to all parent nodes, read/write access to current and all child nodes **and parent node**

Computing Animation

- Animate Action Algorithm:
  - Call updatePosition() for each *BehaviorModel*
  - If there is a *Solver* :
    - Call solver->solve(dt) (which will execute mechanical actions)
    - Stop the recursion
  - Else
    - Continue the recursion
- Mechanical Actions:
  - PropagatePositionAndVelocity: set new position and velocity and apply mappings to propagate them downward
  - ComputeForce: call all ForceFields to compute the current force and apply mappings to accumulate it upward
  - AccFromF: use Mass to compute accelerations from force
  - $V = A + B . f$: linear vector operation (i.e. $x \mathrel{+}= v.dt$)

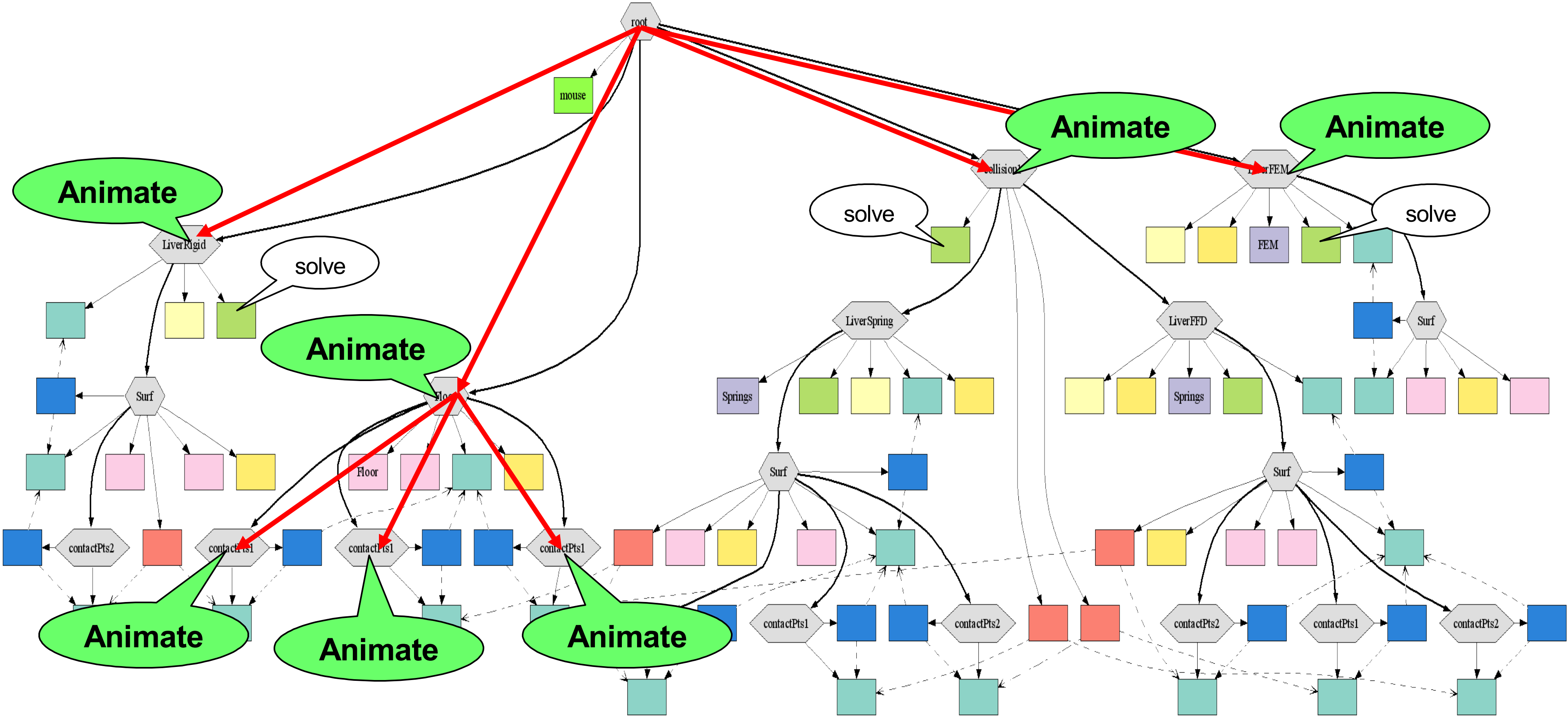
Computing Animation: Step 1



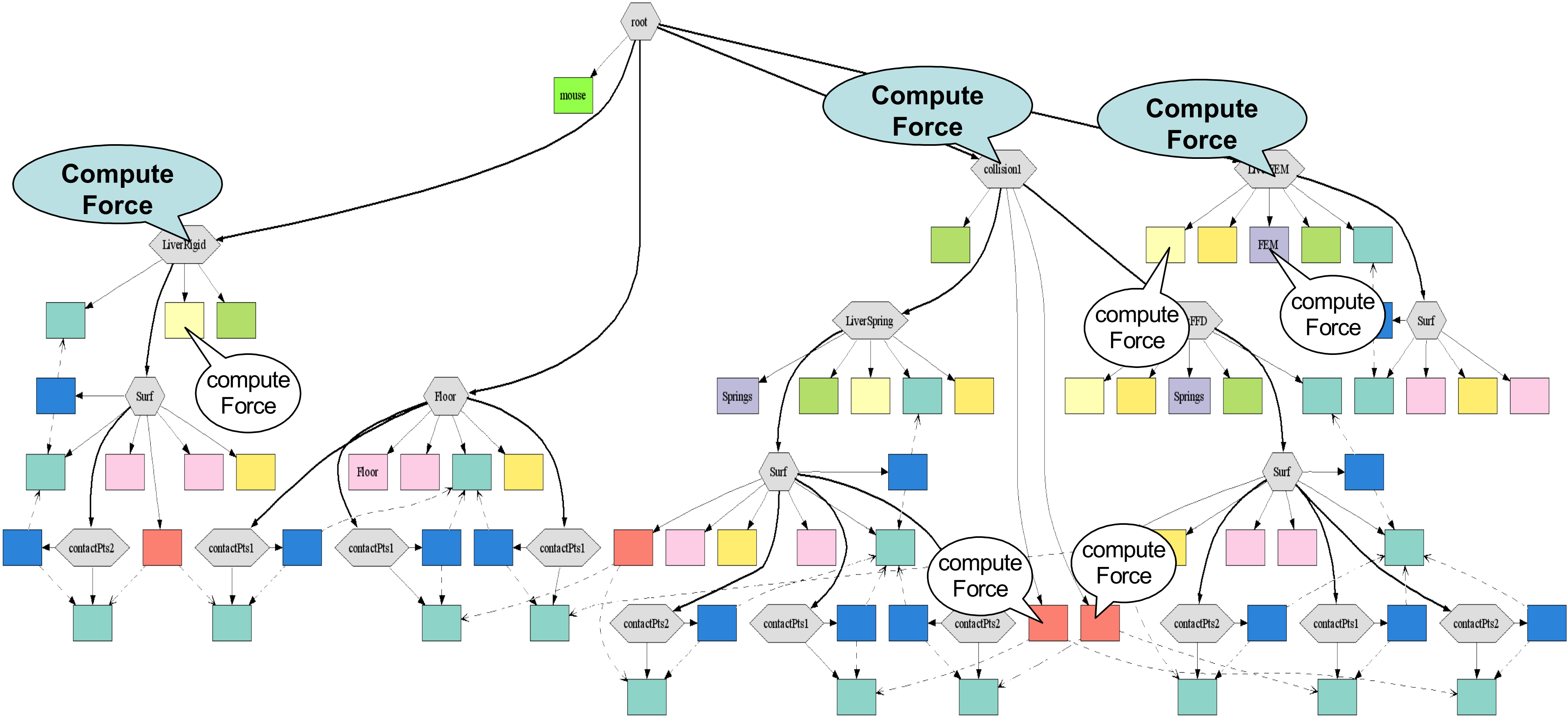
Animate

UpdatePosition

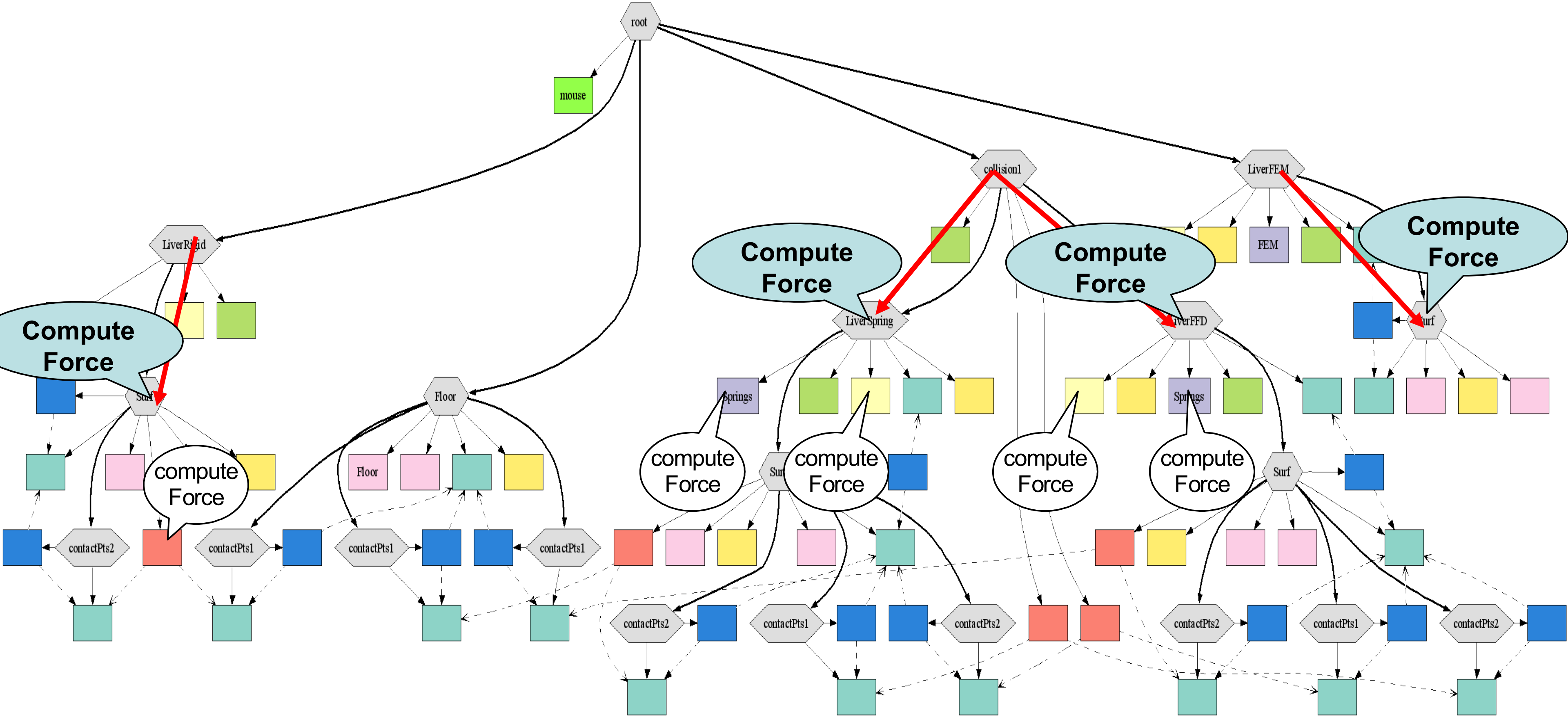
Computing Animation: Step 2



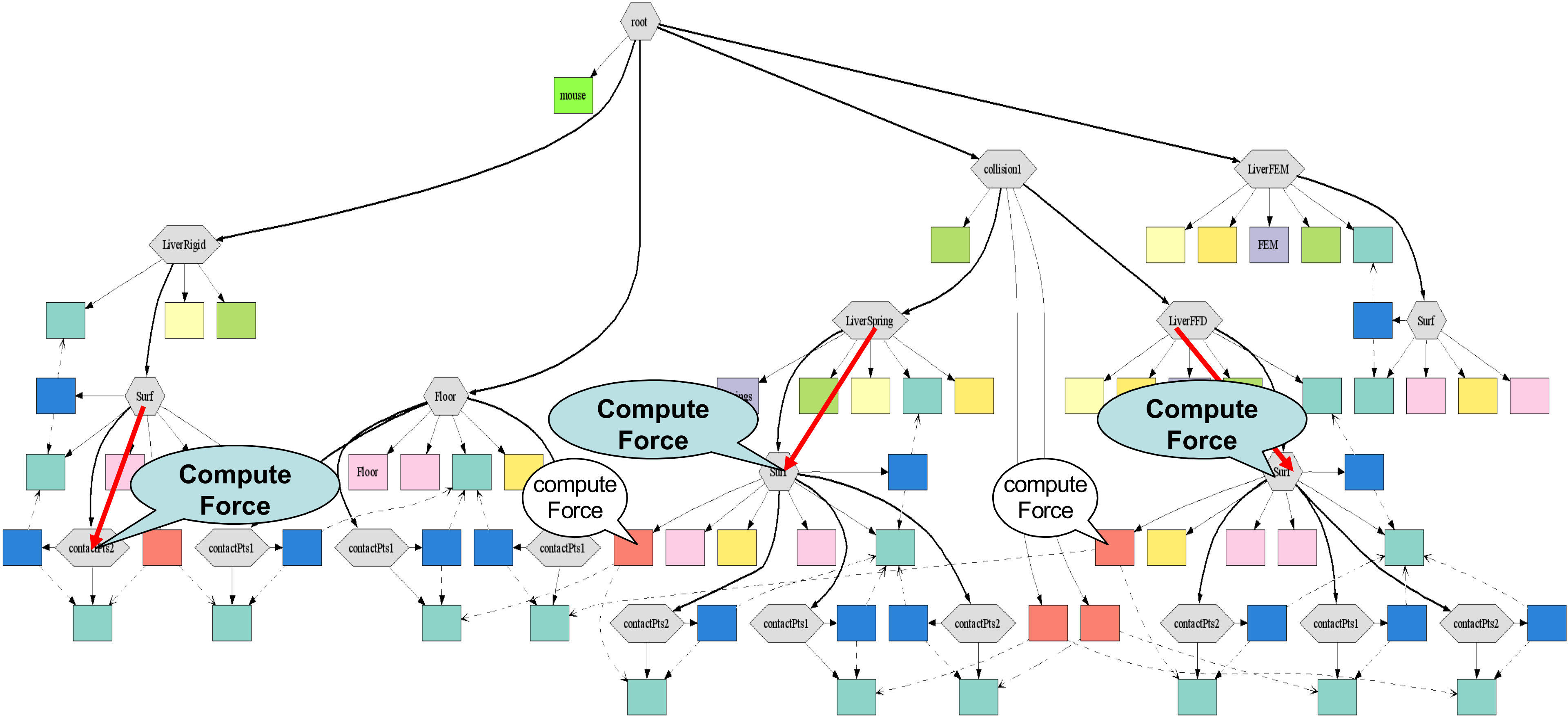
Computing Animation: Step 3



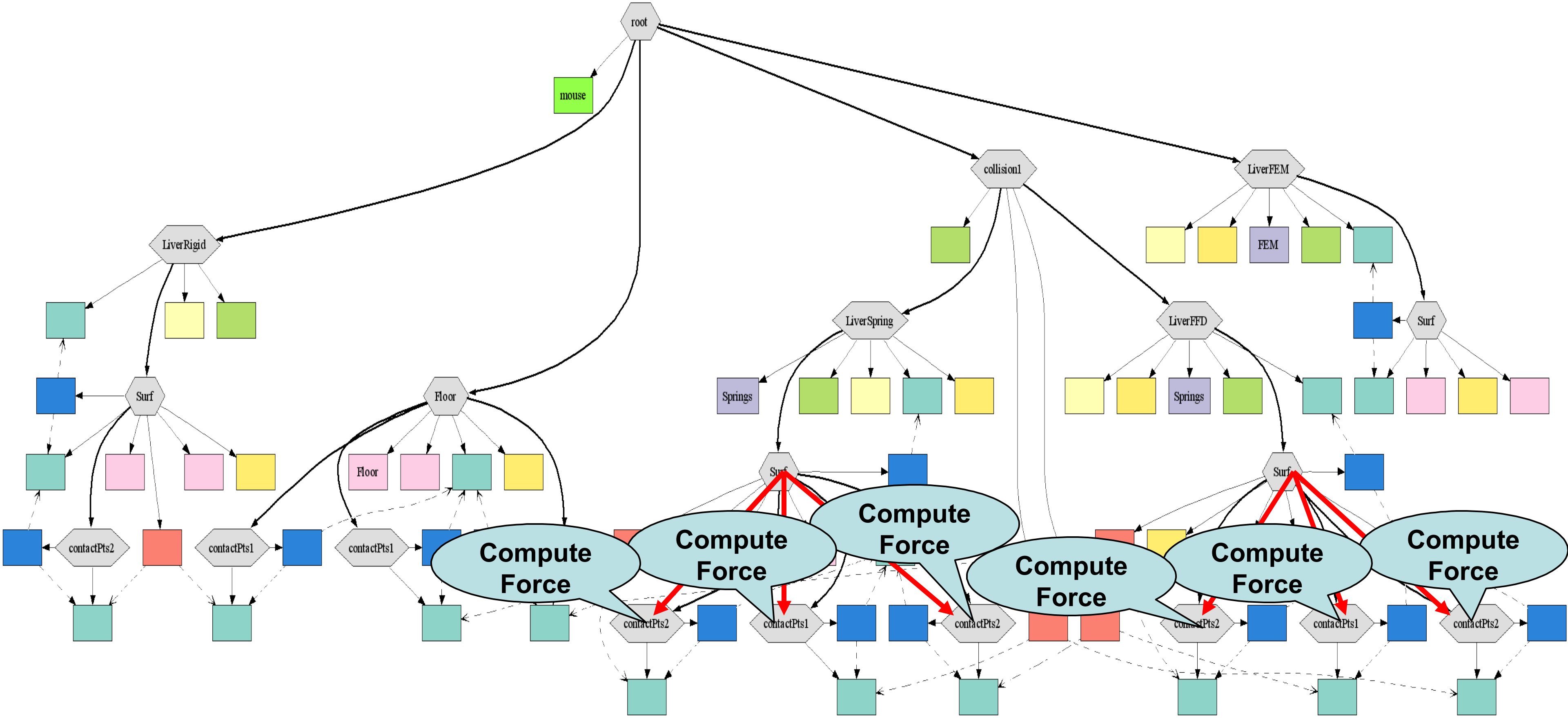
Computing Animation: Step 4



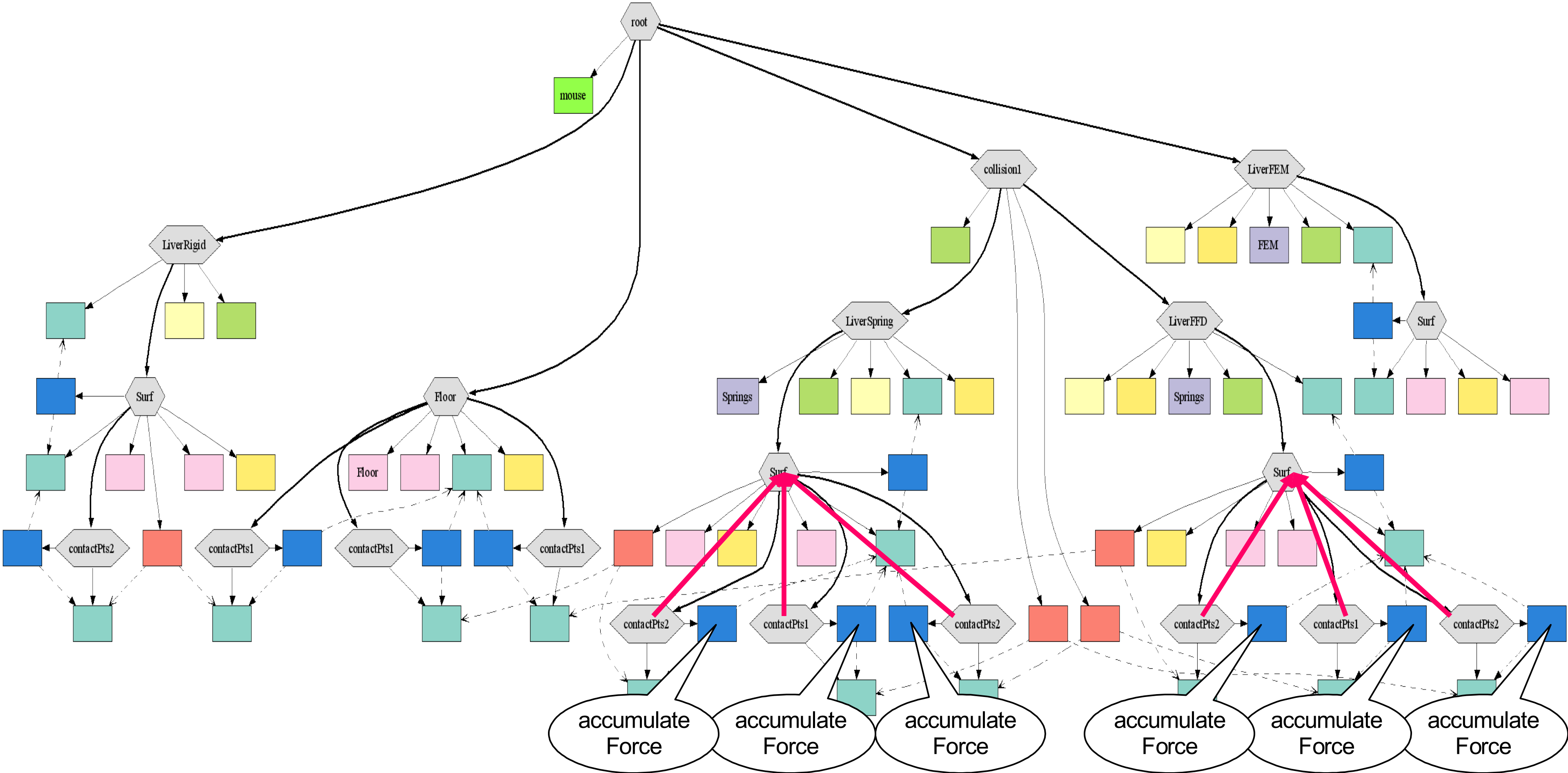
Computing Animation: Step 5



Computing Animation: Step 6



Computing Animation: Step 7



accumulate Force

accumulate Force

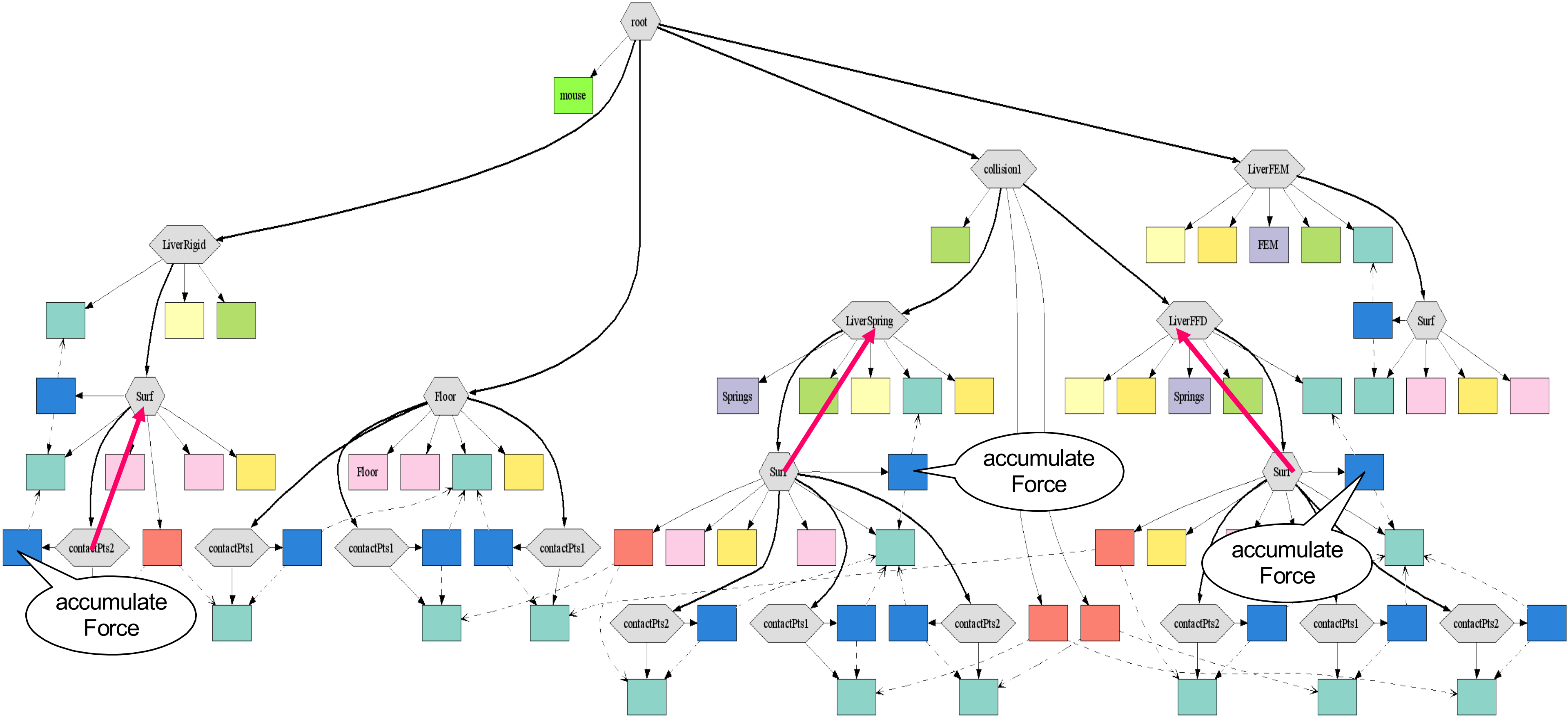
accumulate Force

accumulate Force

accumulate Force

accumulate Force

Computing Animation: Step 8

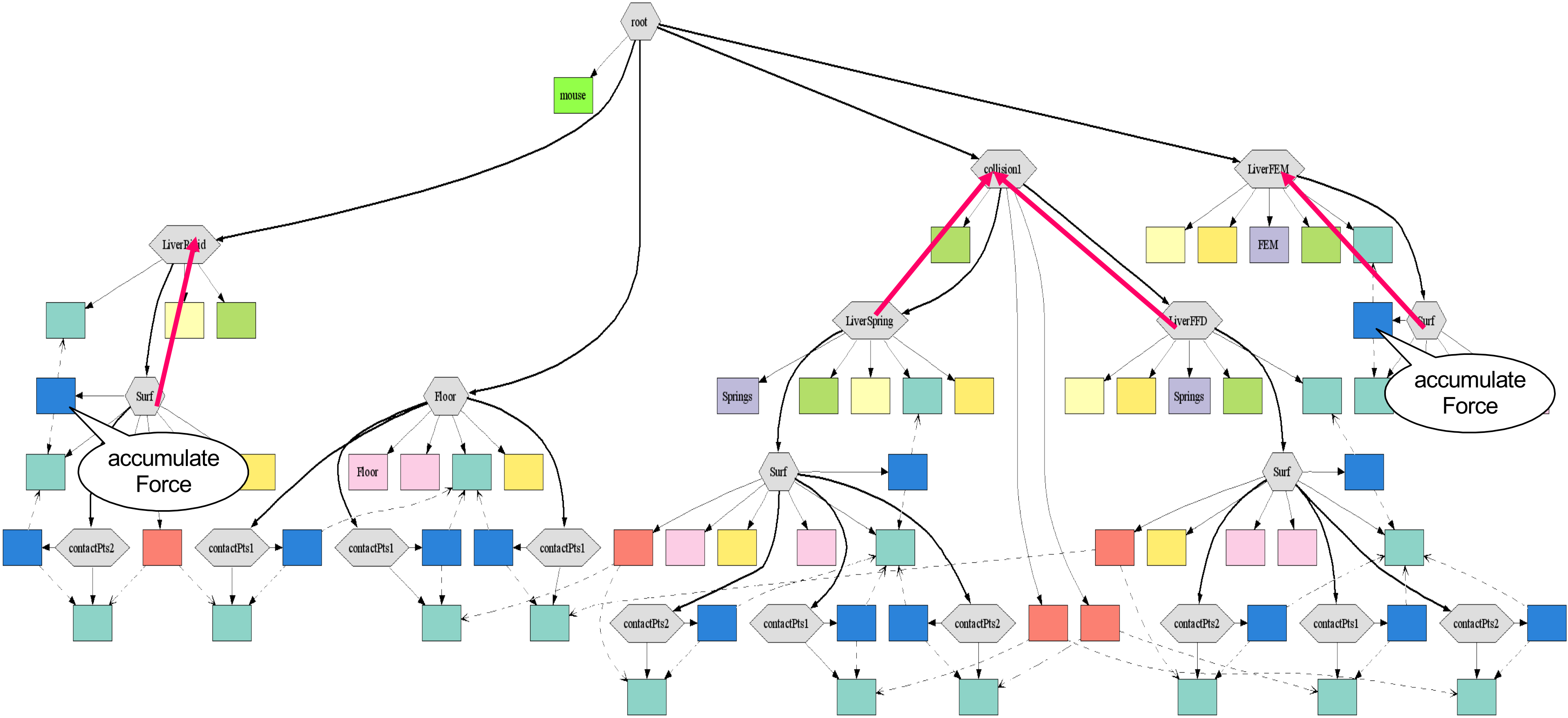


accumulate Force

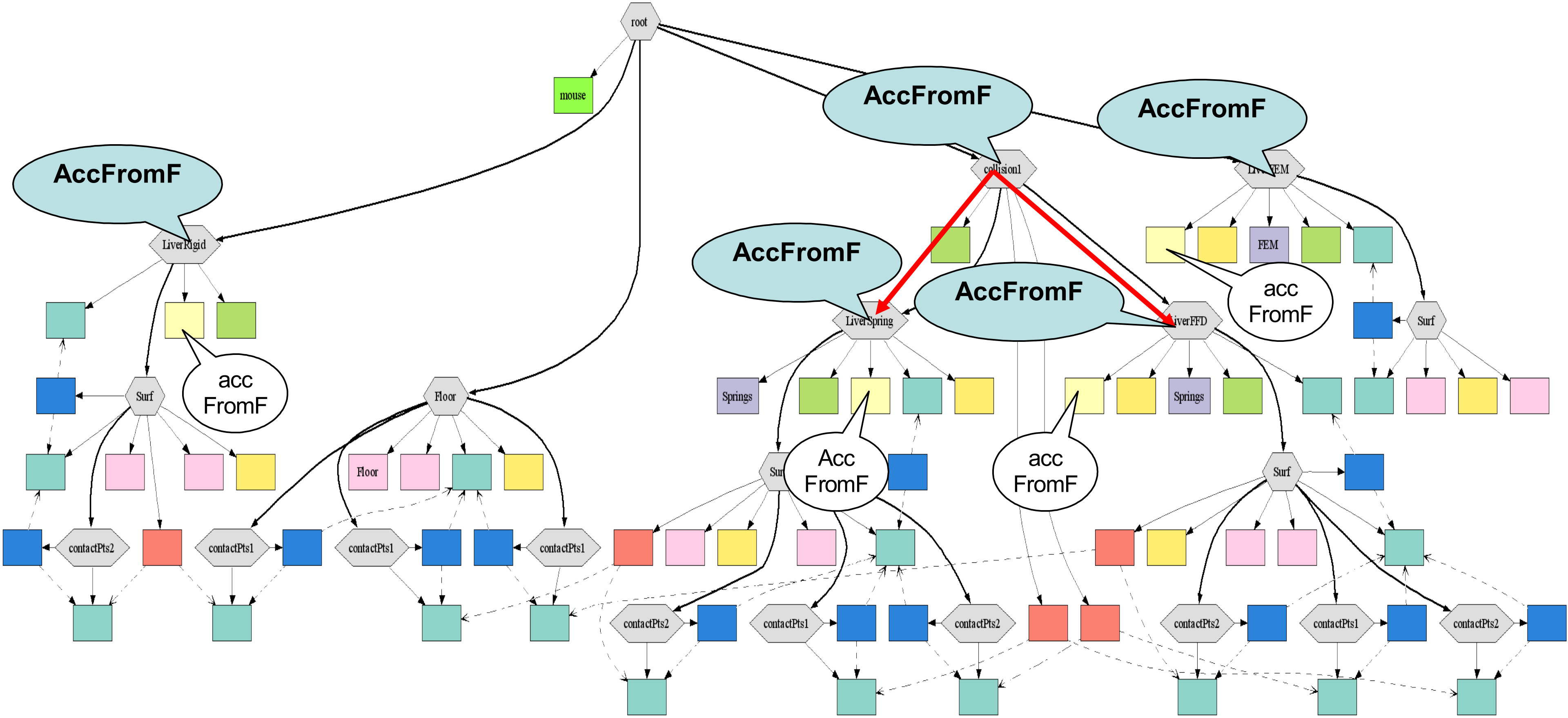
accumulate Force

accumulate Force

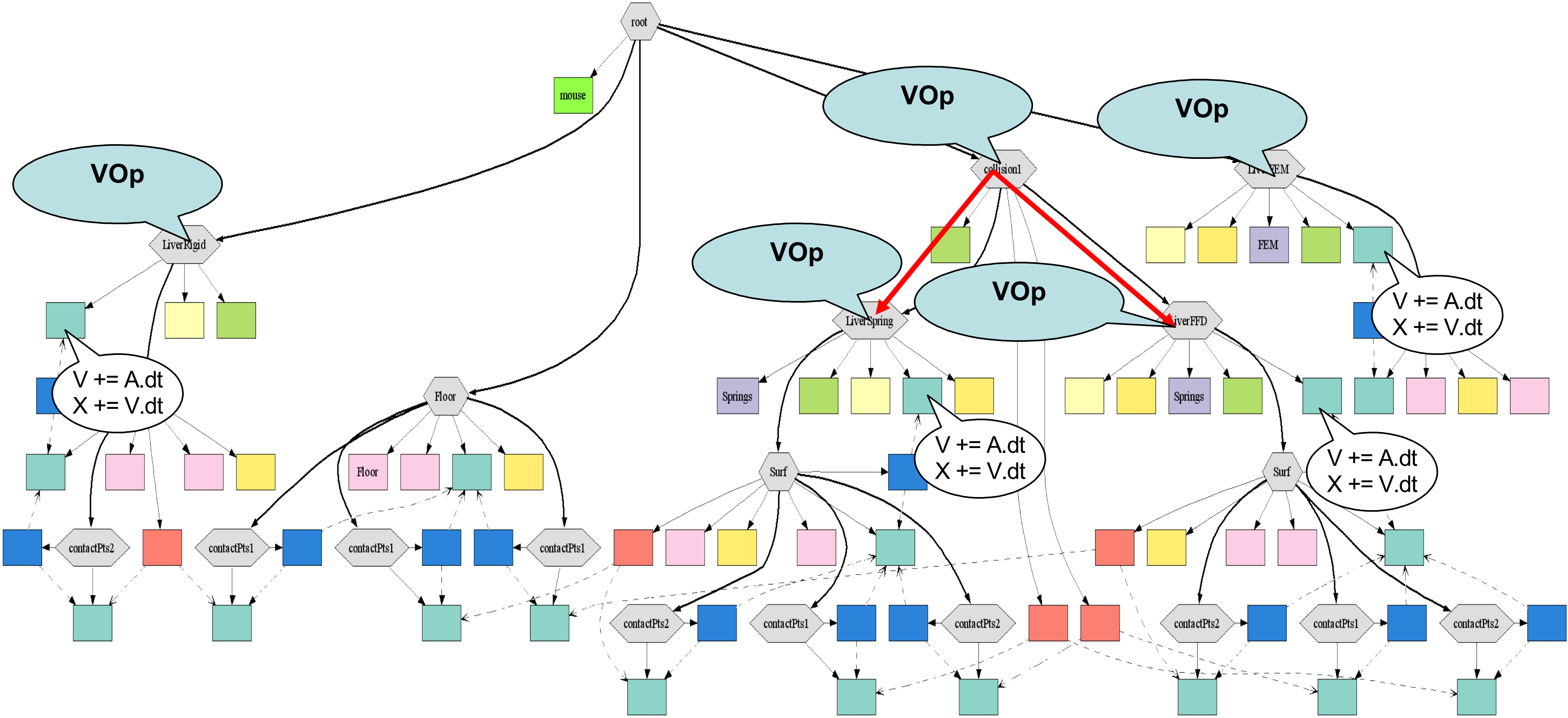
Computing Animation: Step 9



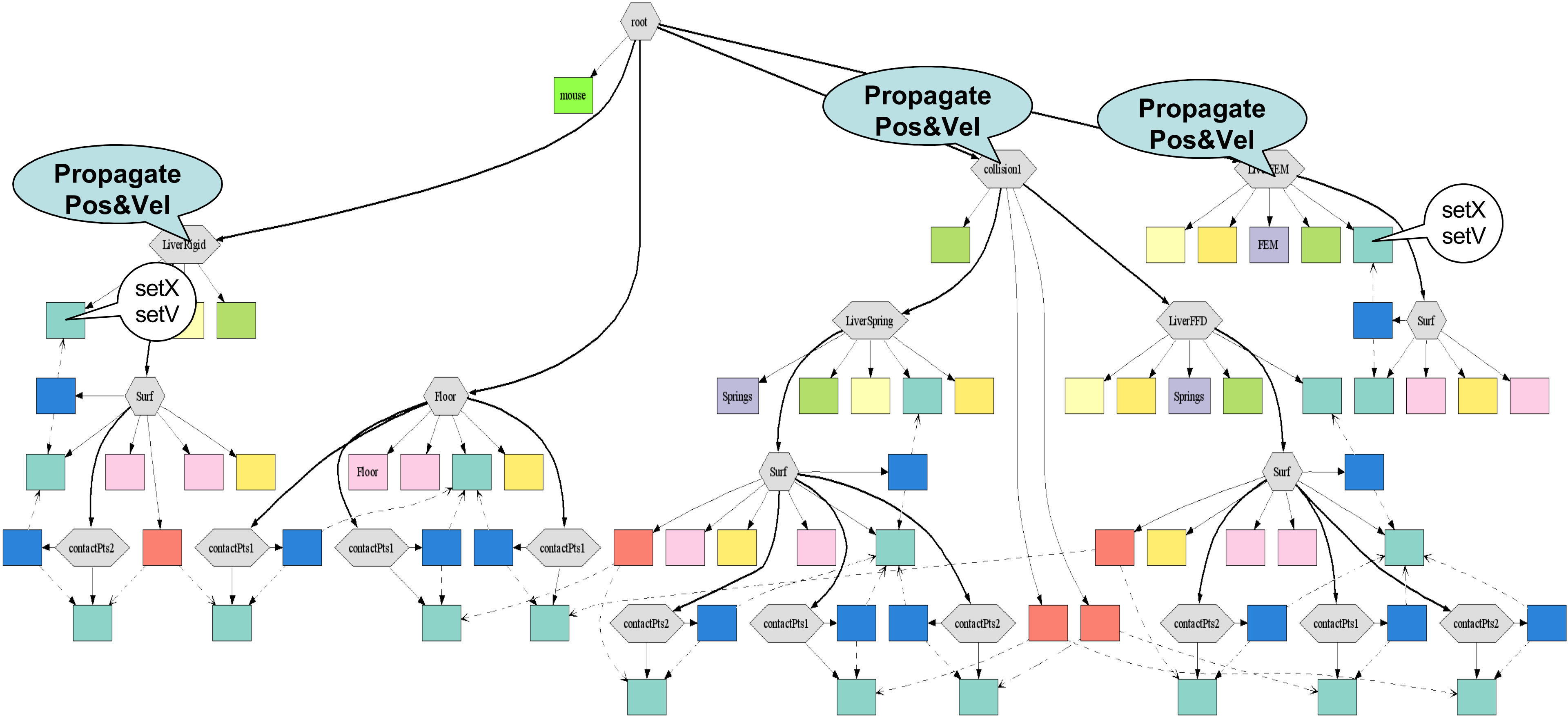
Computing Animation: Step 10



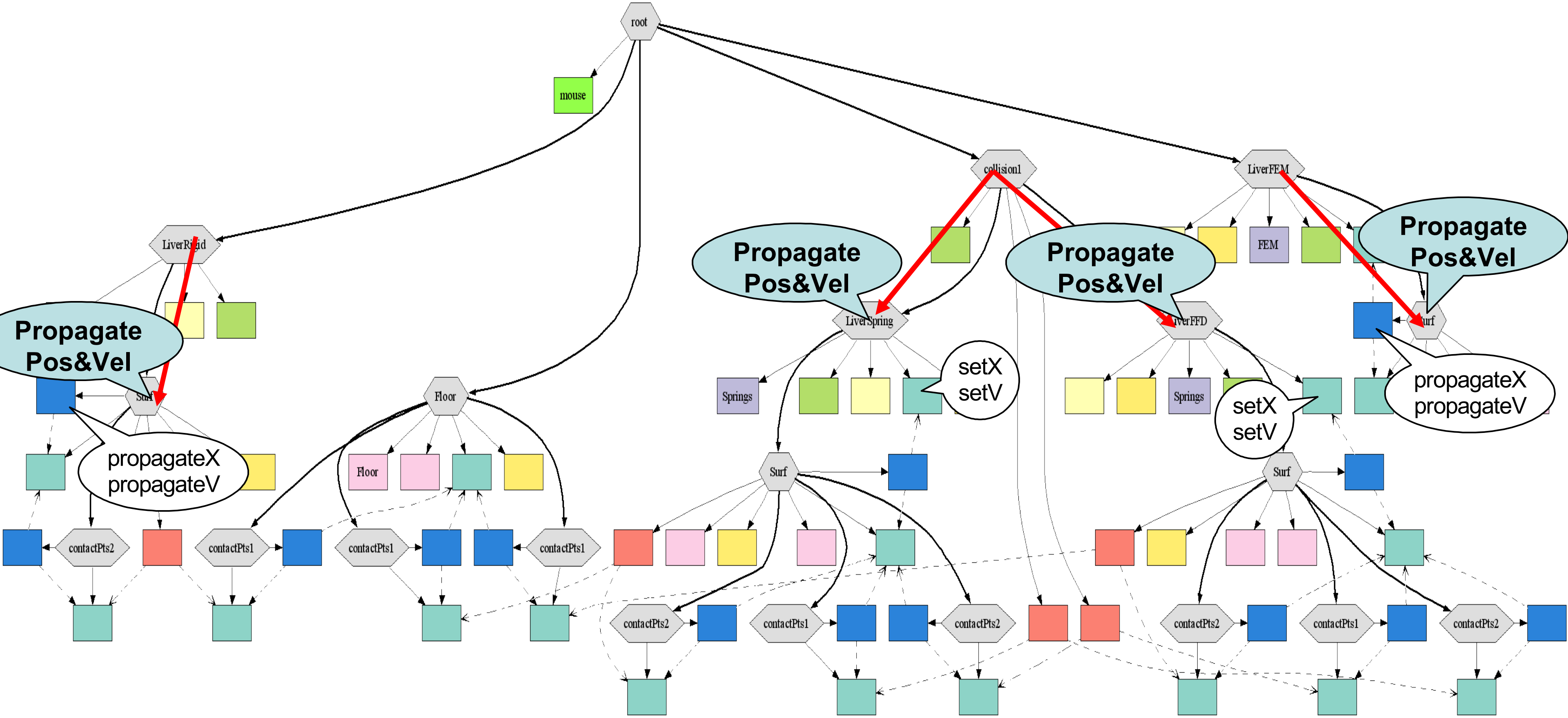
Computing Animation: Step 11



Computing Animation: Step 12



Computing Animation: Step 13



Propagate Pos&Vel

propagateX
propagateV

Propagate Pos&Vel

setX
setV

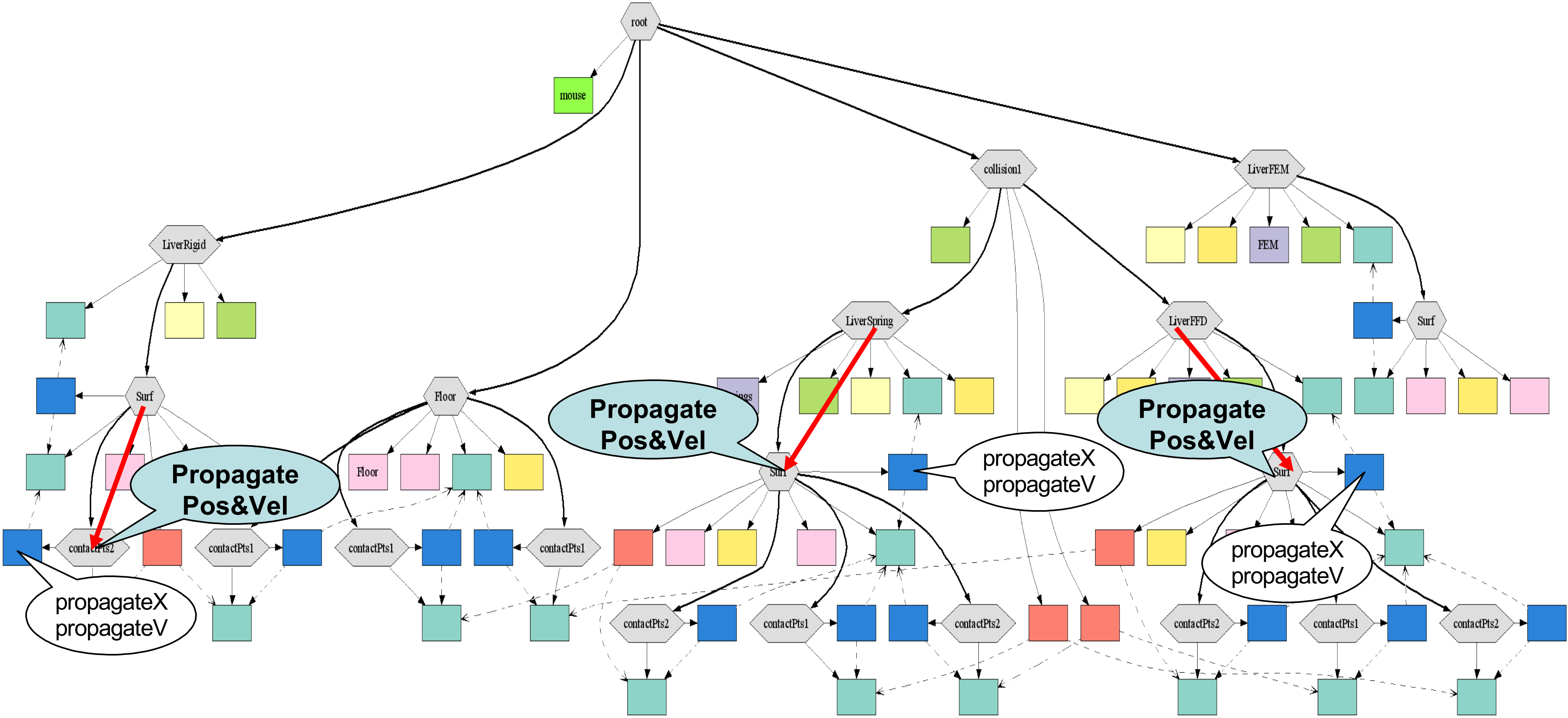
Propagate Pos&Vel

setX
setV

Propagate Pos&Vel

propagateX
propagateV

Computing Animation: Step 14



Propagate Pos&Vel

Propagate Pos&Vel

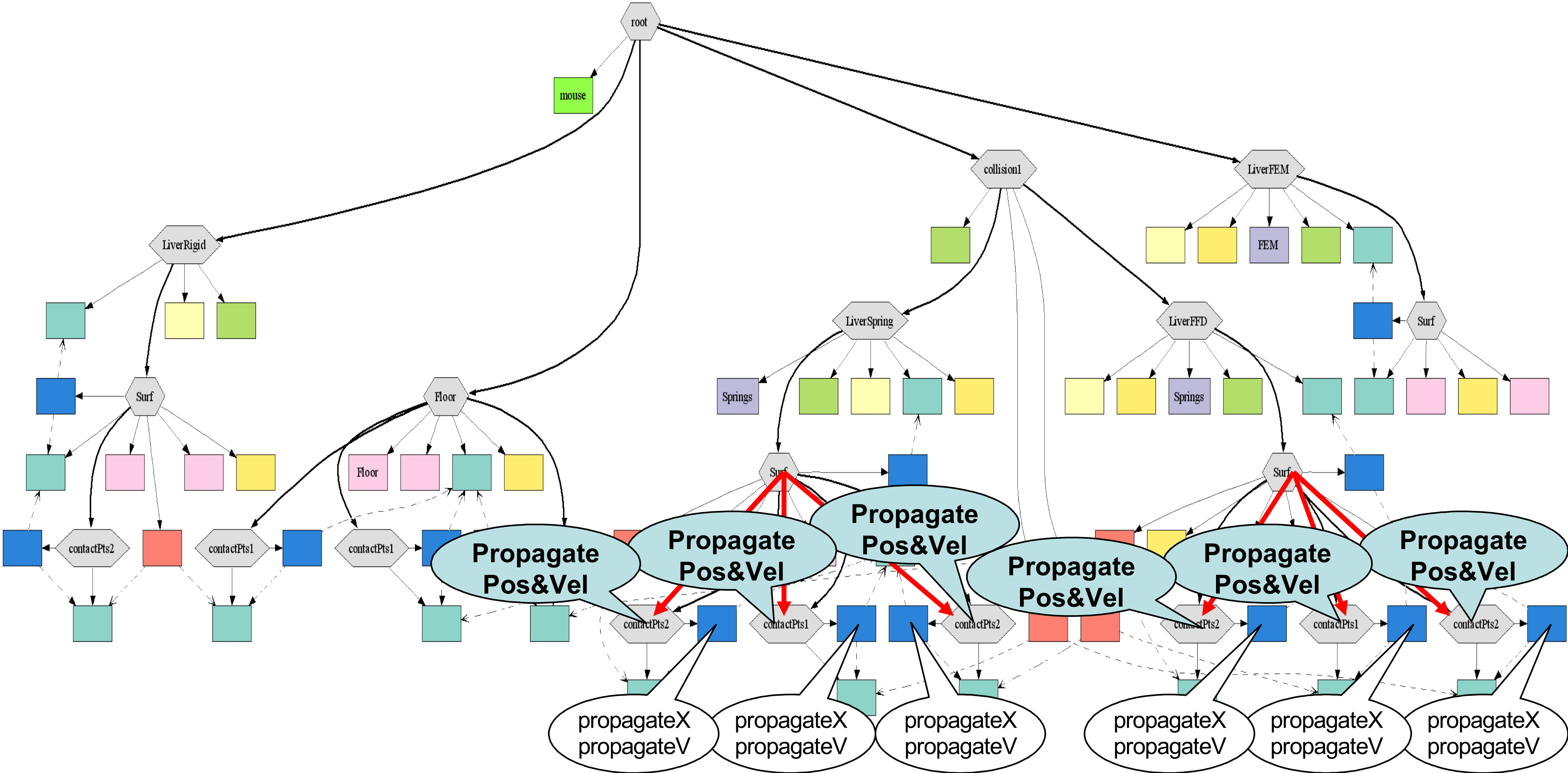
Propagate Pos&Vel

propagateX propagateV

propagateX propagateV

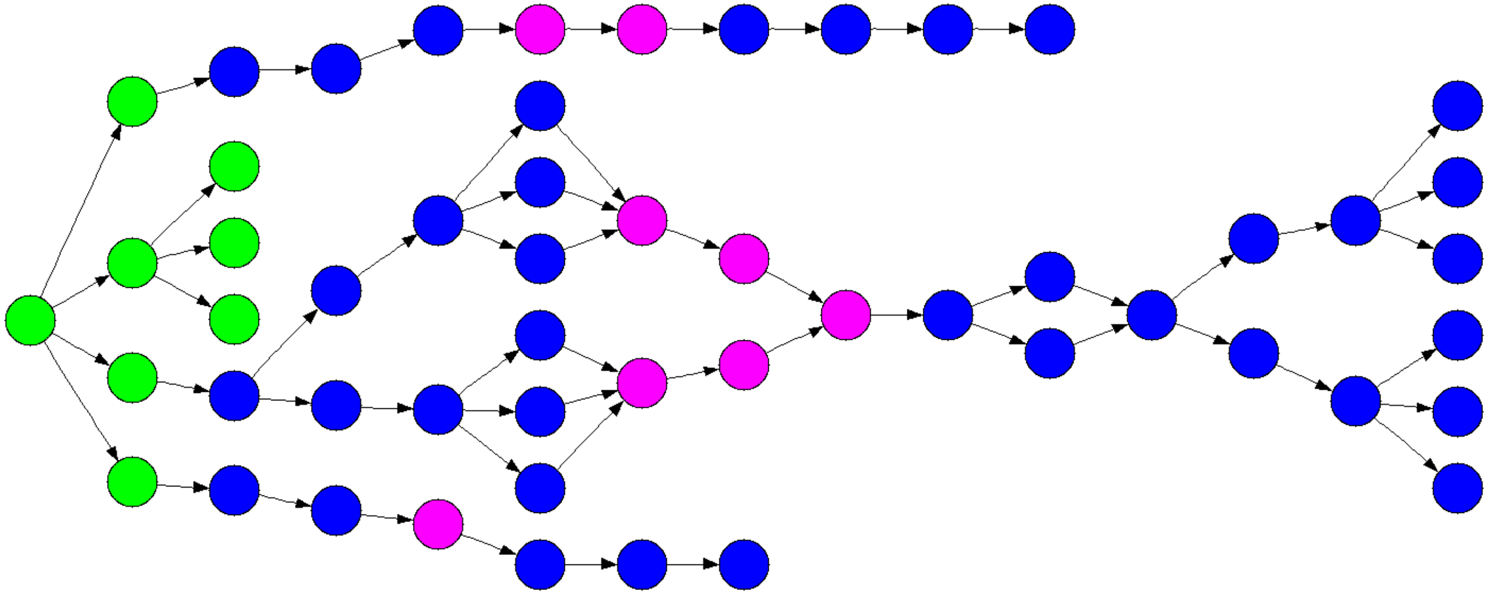
propagateX propagateV

Computing Animation: Step 15

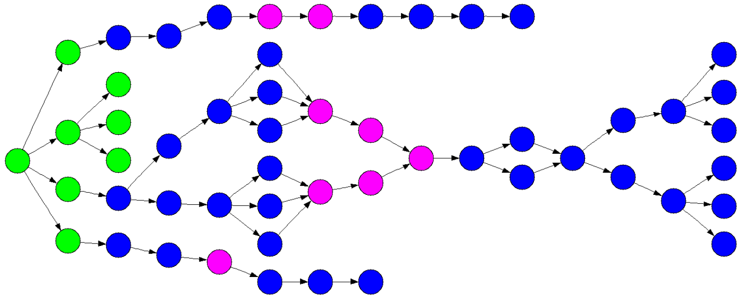


Task dependency graph

- Determined by the scene tree
- More advanced solvers requires more actions
  - varying number for iterative solvers (CG)



Parallel Scheduler



- Coarse grained:
  - Schedule Animate actions (green tasks)

    # thread ≤ # integration groups ≤ # objects
- Fine grained:
  - Schedule all tasks

    Cost of parallelization increase with the size of the tree
- Adaptive:
  - Work stealing

    Costly only when necessary (when one idle thread *steals* a task)

Work Stealing Scheduler

- Requirements
  - Handle > 1000 tasks per iteration (with > 30 it/sec)
  - Support Linux and Windows

    Linux-only is fine for initial tests
- Several possibilities
  - KAAPI ?
  - Cilk ?
  - Custom code ? (Luciano’s octree work)