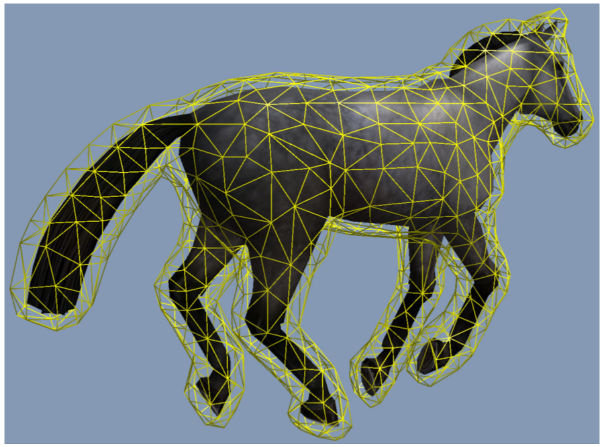
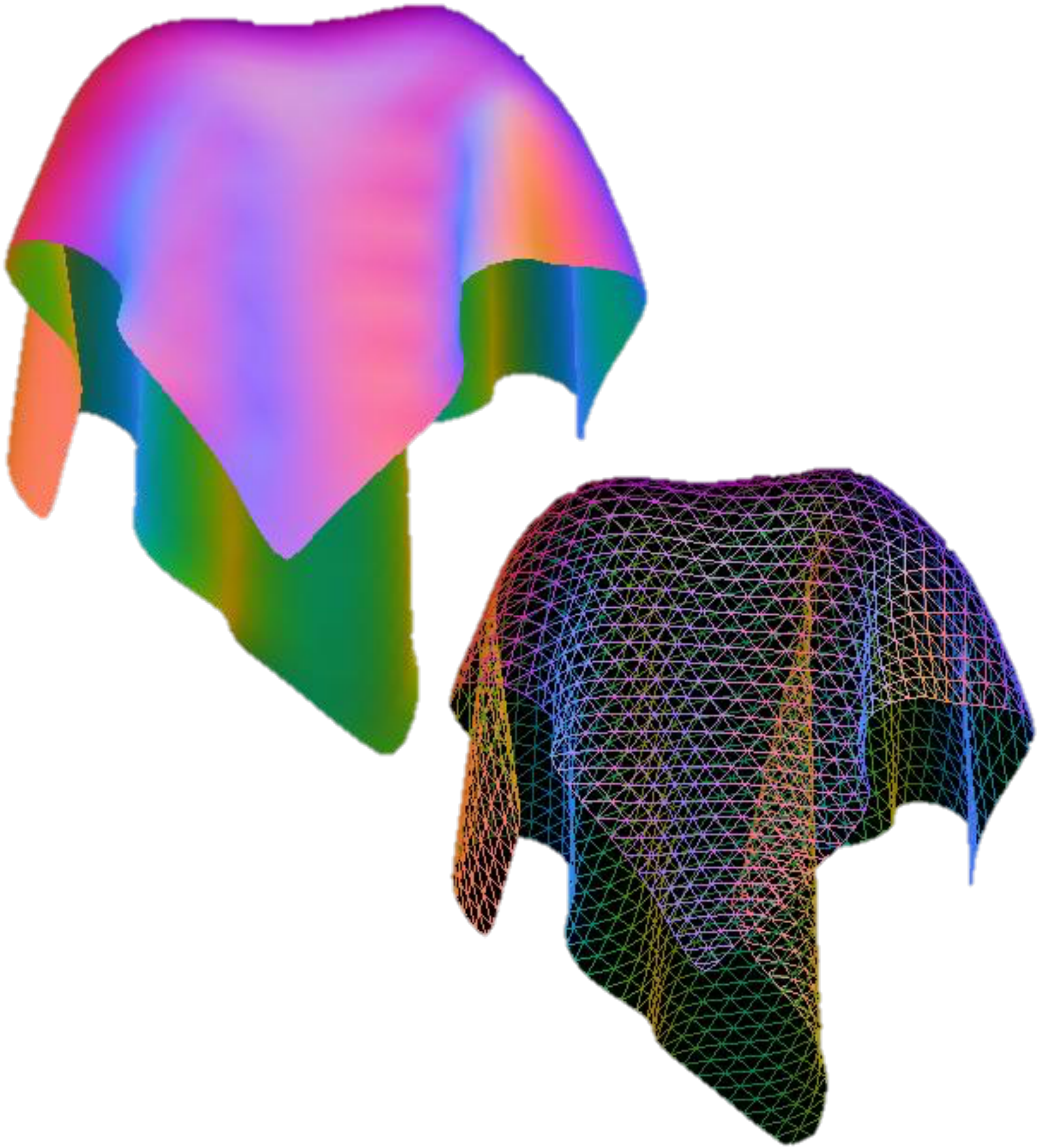
Virtual Reality & Physically-Based Simulation
Mass-Spring-Systems

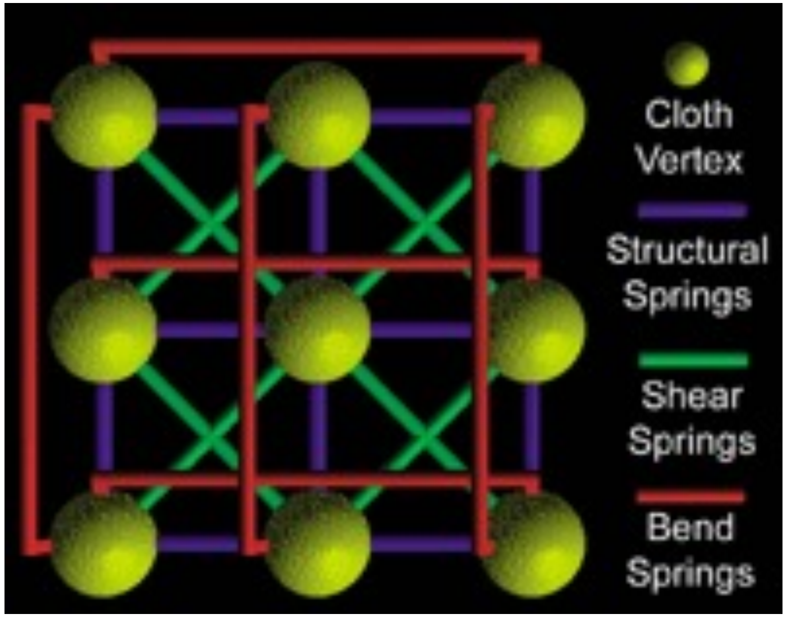


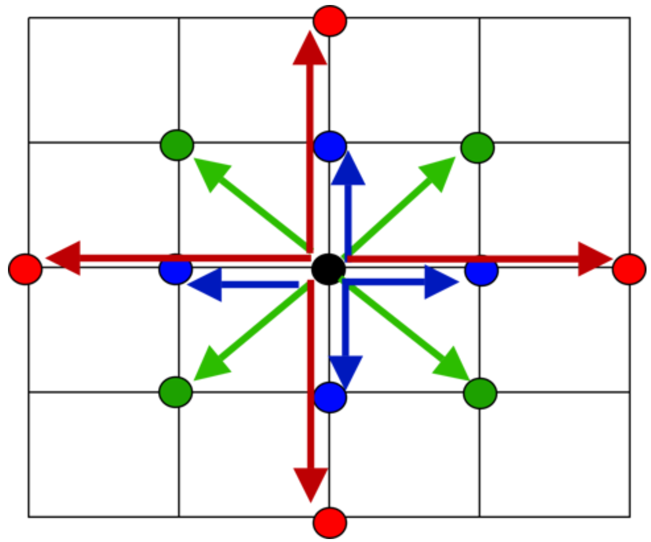
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

- Definition:
  A mass-spring system is a system consisting of:
  1. A set of point masses m_i with positions $\mathbf{x}_i$ and velocities $\mathbf{v}_i$, i = 1…n ;
  2. A set of springs $s_{ij} = (i, j, k_s, k_d)$, where s_{ij} connects masses *i* and *j*, with rest length l_0 , spring constant k_s (= stiffness) and the damping coefficient k_d
- Advantages:
  - Very easy to program
  - Ideally suited to study different kinds of solving methods
  - *Ubiquitous in games* (cloths, capes, sometimes also for deformable objects)
- Disadvantages:
  - Some parameters (in particular the spring constants) are not obvious, i.e., difficult to derive
  - No built-in volumetric effects (e.g., preservation of volume)

Example Mass-Spring System: Cloth

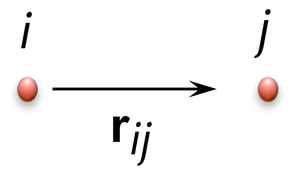




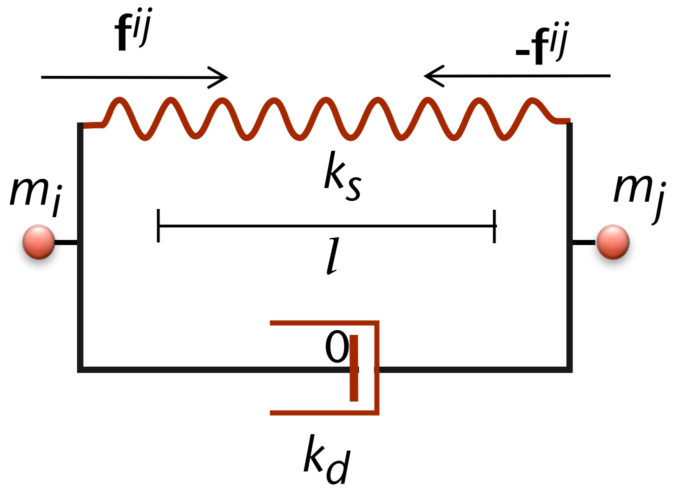


A Single Spring (Plus Damper)

- Given: masses m_i and m_j with positions $\mathbf{x}_i$, $\mathbf{x}_j$



- Let $\mathbf{r}_{ij} = \dfrac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|}$
- The force between particles *i* and *j* :



1. Force exerted by spring (Hooke's law):

$$\mathbf{f}_s^{ij} = k_s \mathbf{r}_{ij}(\|\mathbf{x}_j - \mathbf{x}_i\| - l_0)$$

   acts on particle *i* in the direction of *j*

2. Force exerted on *i* by damper: $\mathbf{f}_d^{ij} = -k_d((\mathbf{v}_i - \mathbf{v}_j)\cdot\mathbf{r}_{ij})\mathbf{r}_{ij}$

3. Total force on *i* : $\mathbf{f}^{ij} = \mathbf{f}_s^{ij} + \mathbf{f}_d^{ij}$

4. Force on m_j : $\mathbf{f}^{ji} = -\mathbf{f}^{ij}$

Remarks

- A spring-damper element in reality:

- Alternative spring force:

$$\mathbf{f}_s^{ij} = k_s \mathbf{r}_{ij} \frac{\|\mathbf{x}_j - \mathbf{x}_i\| - l_0}{l_0}$$

- Notice: from (4) it follows that the total momentum is conserved
  - Momentum $\mathbf{p} = \mathbf{v} \cdot m$
  - Fundamental physical law (follows from Newton's laws)
- Note on terminology:
  - English "momentum" = German "Impuls" = velocity × mass
  - English "Impulse" = German "Kraftstoß" = force × time

Simulation of a Single Spring

- From Newton's law, we have: $\ddot{\mathbf{x}} = \frac{1}{m}\mathbf{f}$
- Convert differential equation (ODE) of order 2 into ODE of order 1:

$$\dot{\mathbf{x}}(t) = \mathbf{v}(t)$$

$$\dot{\mathbf{v}}(t) = \frac{1}{m}\mathbf{f}(t)$$

- Initial values (boundary values): $\mathbf{v}(t_0) = \mathbf{v}_0$, $\mathbf{x}(t_0) = \mathbf{x}_0$
- "Simulation" = "Integration of ODE's over time"
- By Taylor expansion we get:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \dot{\mathbf{x}}(t) + O\left(\Delta t^2\right)$$

- Analogously: $\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \Delta t\, \dot{\mathbf{v}}(t)$

→ This integration scheme is called **explicit Euler integration**

The Algorithm for a Mass-Spring System

```
forall particles i :
   initialize x_i, v_i, m_i

loop forever:
   forall particles i :
```

$$\mathbf{f}_i \leftarrow \mathbf{f}^g + \mathbf{f}_i^{coll} + \sum_{j,\,(i,j)\in S} \mathbf{f}(\mathbf{x}_i, \mathbf{v}_i, \mathbf{x}_j, \mathbf{v}_j)$$

```
   forall particles i :
```

$$\mathbf{v}_i \mathrel{+}= \Delta t \cdot \frac{\mathbf{f}_i}{m_i}$$

$$\mathbf{x}_i \mathrel{+}= \Delta t \cdot \mathbf{v}_i$$

```
   render the system every n-th time
```

$\mathbf{f}^{g}$ = gravitational force

$\mathbf{f}^{coll}$ = penalty force exerted by collision (e.g., from obstacles)

- Advantages:
  - Can be implemented very easily
  - Fast execution per time step
  - Is "trivial" to parallelize on the GPU (⟶ "Massively Parallel Algorithms")
- Disadvantages:
  - Stable only for very small time steps
    - Typically $\Delta t \approx 10^{-4} \ldots 10^{-3}$ sec!
  - With large time steps, additional energy is generated "out of thin air", until the system explodes ☺
  - Example: overshooting when simulating a single spring
  - Errors accumulate quickly

Example for the Instability of Euler Integration

- Consider the differential equation

$$\dot{x}(t) = -kx(t)$$

- The exact solution:

$$x(t) = x_0\, e^{-kt}$$

- Euler integration does this:

$$x^{t+1} = x^t + \Delta t(-kx^t)$$

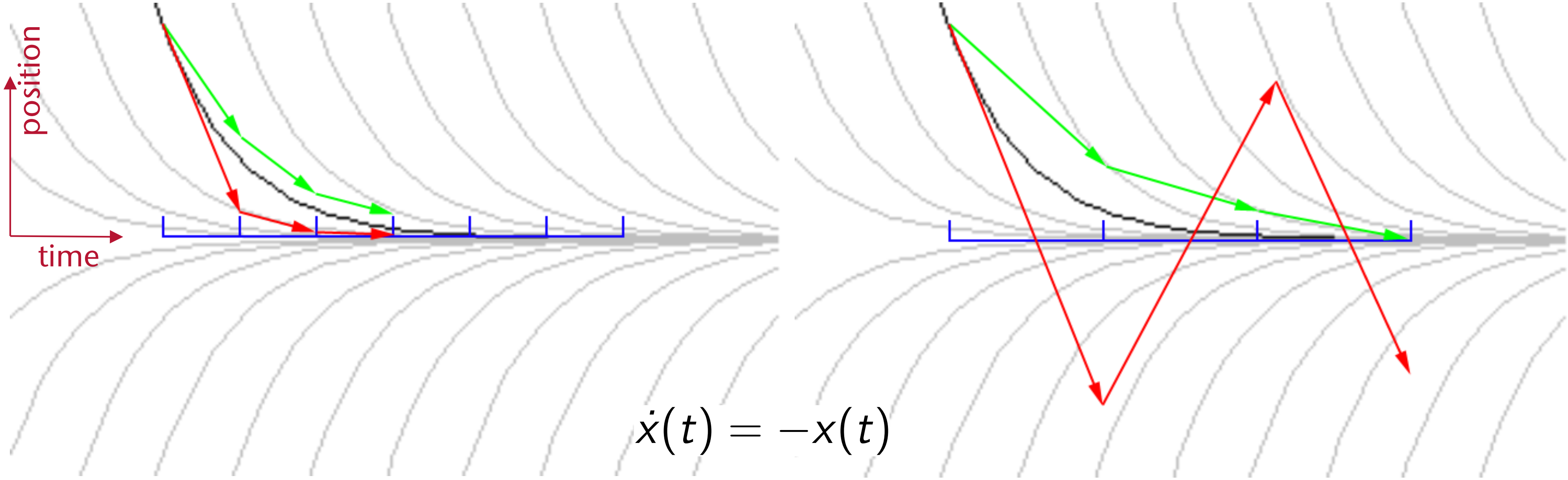
- Case $\Delta t > \frac{1}{k}$:

$$x^{t+1} = x^t \underbrace{(1 - k\Delta t)}_{<0}$$

$\Rightarrow$ x^t oscillates about 0, but approaches 0 (hopefully)

- Case $\Delta t > \frac{2}{k}$: $\Rightarrow$ $x^t \rightarrow \infty$!

- Visualization:



$$\dot{x}(t) = -x(t)$$

- Terminology: if *k* is large → the ODE is called *"stiff "*
  - The stiffer the ODE, the smaller Δt has to be

Visualization of Error Accumulation

- Consider this ODE:

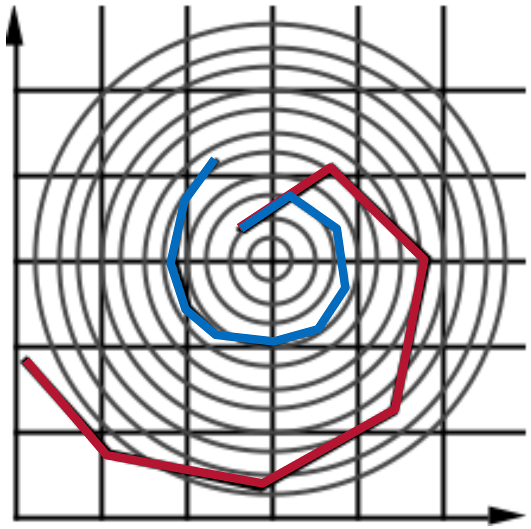
$$\dot{\mathbf{x}}(t) = \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix}$$

- Exact solution:

$$\mathbf{x}(t) = \begin{pmatrix} r\cos(t+\phi) \\ r\sin(t+\phi) \end{pmatrix}$$

- The solution by Euler integration moves in spirals outward, no matter how small Δt!

- Conclusion: Euler integration accumulates errors, no matter how small Δt!

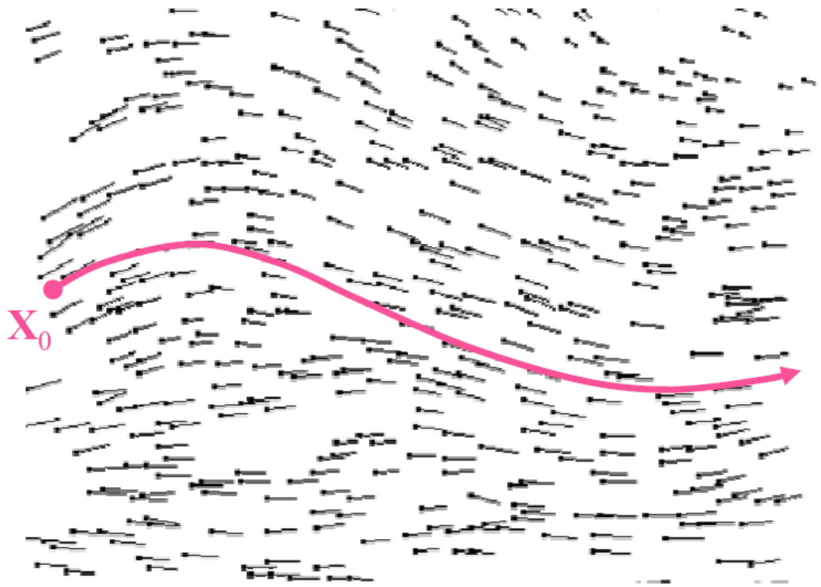


Visualization of Differential Equations

- The general form of an ODE (ordinary differential equation):

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\,\mathbf{x}(t), t\,)$$

- Visualization of **f** as a vector field:



  - Notice: this vector field can vary over time!

- Solution of a boundary value problem = path through this field

Other Integrators

- Runge-Kutta of order 2:
  - Idea: approximate **f**(**x**(*t*), *t*) by using the derivative at positions **x**(*t*) and **x**(*t*+ ½Δ*t*)
  - The integrator (w/o proof):

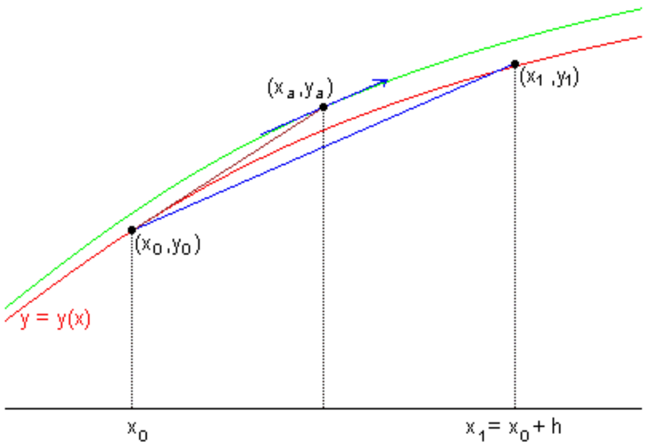
$$\mathbf{a}_1 = \mathbf{v}^t \qquad \mathbf{a}_2 = \frac{1}{m}\mathbf{f}(\mathbf{x}^t, \mathbf{v}^t)$$

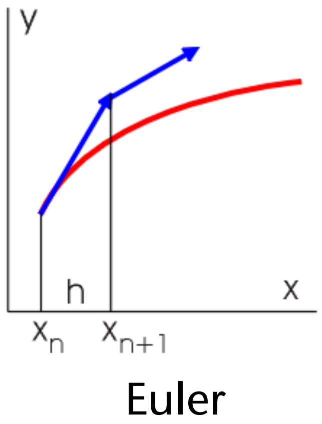
$$\mathbf{b}_1 = \mathbf{v}^t + \frac{1}{2}\Delta t \mathbf{a}_2 \qquad \mathbf{b}_2 = \frac{1}{m}\mathbf{f}\left(\mathbf{x}^t + \frac{1}{2}\Delta t \mathbf{a}_1, \mathbf{v}^t + \frac{1}{2}\Delta t \mathbf{a}_2\right)$$

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \Delta t \mathbf{b}_1 \qquad \mathbf{v}^{t+1} = \mathbf{v}^t + \Delta t \mathbf{b}_2$$

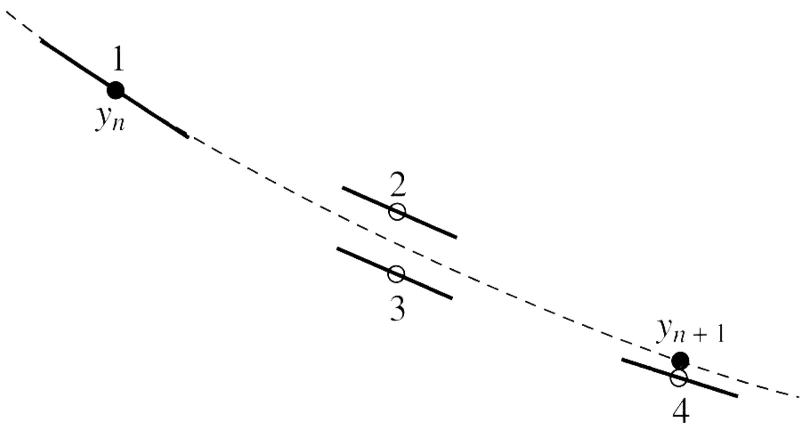
- Runge-Kutta of order 4:
  - The standard integrator among the explicit integration schemata
  - Needs 4 function evaluations (i.e., force computations) per time step
  - Order of convergence is: $e(\Delta t) = O\left(\Delta t^4\right)$

- Runge-Kutta of order 2:





- Runge-Kutta of order 4:



Verlet Integration

- A general, alternative idea to increase the order of convergence: utilize values from the past
- Verlet integration = utilize $\mathbf{x}(t - \Delta t)$
- Derivation:
  - Develop the Taylor series in both time directions:

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \Delta t \dot{\mathbf{x}}(t) + \frac{1}{2}\Delta t^2 \ddot{\mathbf{x}}(t) + \frac{1}{6}\Delta t^3 \dddot{\mathbf{x}}(t) + O(\Delta t^4)$$

$$\mathbf{x}(t-\Delta t) = \mathbf{x}(t) - \Delta t \dot{\mathbf{x}}(t) + \frac{1}{2}\Delta t^2 \ddot{\mathbf{x}}(t) - \frac{1}{6}\Delta t^3 \dddot{\mathbf{x}}(t) + O(\Delta t^4)$$

- Add both:

$$\mathbf{x}(t + \Delta t) + \mathbf{x}(t - \Delta t) = 2\mathbf{x}(t) + \Delta t^2\, \ddot{\mathbf{x}}(t) + O(\Delta t^4)$$

$$\mathbf{x}(t + \Delta t) = 2\mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \Delta t^2\, \ddot{\mathbf{x}}(t) + O(\Delta t^4)$$

- Initialization:

$$\mathbf{x}(\Delta t) = \mathbf{x}(0) + \Delta t \mathbf{v}(0) + \frac{1}{2}\Delta t^2 (\frac{1}{m}\mathbf{f}(\mathbf{x}(0), \mathbf{v}(0)))$$

- Remark: the velocity does not occur any more! (at least, not explicitly)

Constraints

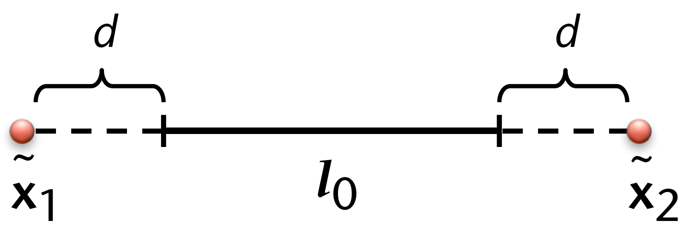
- Big advantage of Verlet over Euler & Runge-Kutta: it is very easy to handle constraints
- Definition: constraint = some condition on the position of one or more mass points
- Examples:
  1. A point must not penetrate an obstacle
  2. The distance between two points must be constant, or distance must be $\leq$ some maximal distance

- Example: consider the constraint

$$\|\mathbf{x}_1 - \mathbf{x}_2\| \stackrel{!}{=} l_0$$

1. Perform one Verlet integration step → $\tilde{\mathbf{x}}^{t+1}$
2. Enforce the constraint:

$$d = \frac{1}{2}(\|\tilde{\mathbf{x}}_2^{t+1} - \tilde{\mathbf{x}}_1^{t+1}\| - l_0)$$



$$\mathbf{x}_1^{t+1} = \tilde{\mathbf{x}}_1^{t+1} + d\mathbf{r}_{12}$$

$$\mathbf{x}_2^{t+1} = \tilde{\mathbf{x}}_2^{t+1} - d\mathbf{r}_{12}$$

- Problem: if several constraints are to constrain the *same* mass point, we need to employ constraint satisfaction algorithms

Time-Corrected Verlet Integration

- Big assumption in basic Verlet: time-delta's are *constant*!
- Solution for non-constant Δt's:
  - Time steps are: $t_i = t_{i-1} + \Delta t_{i-1}$ and $t_{i+1} = t_i + \Delta t_i$
  - Expand Taylor series in both directions:

$$\mathbf{x}(t_i + \Delta t_i) \quad \text{and} \quad \mathbf{x}(t_i - \Delta t_{i-1})$$

  - Divide the expansions by Δt_i and Δt_{i-1}, respectively, then add both, like in the derivation of the basic Verlet
  - Rearranging and omitting higher-order terms yields:

$$\mathbf{x}(t_i + \Delta t_i) = \mathbf{x}(t_i) + \frac{\Delta t_i}{\Delta t_{i-1}}\left(\mathbf{x}(t_i) - \mathbf{x}(t_i - \Delta t_{i-1})\right) + \ddot{\mathbf{x}}(t_i)\frac{\Delta t_i + \Delta t_{i-1}}{2} \cdot \Delta t_i$$

- Note: basic Verlet is a special case of time-corrected Verlet

Implicit Integration (a.k.a. Backwards Euler)

- All explicit integration schemes are only *conditionally stable*
  - I.e.: they are only stable for a specific range for Δt
  - This range depends on the stiffness of the springs
- Goal: *unconditionally stability*
- One option: implicit Euler integration

explicit

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \Delta t \mathbf{v}_i^t$$

$$\mathbf{v}_i^{t+1} = \mathbf{v}_i^t + \Delta t \frac{1}{m_i} \mathbf{f}(\mathbf{x}^t)$$

implicit

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \Delta t \mathbf{v}_i^{t+1}$$

$$\mathbf{v}_i^{t+1} = \mathbf{v}_i^t + \Delta t \frac{1}{m_i} \mathbf{f}(\mathbf{x}^{t+1})$$

- Now we've got a system of non-linear, algebraic equations, with $\mathbf{x}^{t+1}$ and $\mathbf{v}^{t+1}$ as unknowns on both sides → implicit integration

Solution Method

- Write the whole spring-mass system with vectors (n = #mass points):

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_{n-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{3n-1} \end{pmatrix} , \quad \mathbf{v} = \begin{pmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_{n-1} \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_{3n-1} \end{pmatrix} , \quad \mathbf{f}(\mathbf{x}) = \begin{pmatrix} \mathbf{f}_0(\mathbf{x}) \\ \vdots \\ \mathbf{f}_{n-1}(\mathbf{x}) \end{pmatrix}$$

$$\mathbf{f}_i = \begin{pmatrix} f_{3i+0}(\mathbf{x}) \\ f_{3i+1}(\mathbf{x}) \\ f_{3i+2}(\mathbf{x}) \end{pmatrix} , \quad M_{3n\,x\,3n} = \begin{pmatrix} m_0 & & & & & & & & \\ & m_0 & & & & & & & \\ & & m_0 & & & & & & \\ & & & m_1 & & & & & \\ & & & & m_1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & m_{n-1} & & \\ & & & & & & & m_{n-1} & \\ & & & & & & & & m_{n-1} \end{pmatrix}$$

- Write all the implicit equations as one big system of equations :

$$M\mathbf{v}^{t+1} = M\mathbf{v}^t + \Delta t\mathbf{f}(\mathbf{x}^{t+1}) \quad (1)$$

$$\mathbf{x}^{t+1} = \mathbf{x}^t + \Delta t\,\mathbf{v}^{t+1} \quad (2)$$

- Plug (2) into (1) :

$$M\mathbf{v}^{t+1} = M\mathbf{v}^t + \Delta t\,\mathbf{f}(\,\mathbf{x}^t + \Delta t\mathbf{v}^{t+1}\,) \quad (3)$$

- Expand **f** as Taylor series:

$$\mathbf{f}(\mathbf{x}^t + \Delta t\,\mathbf{v}^{t+1}) = \mathbf{f}(\,\mathbf{x}^t) + \frac{\partial}{\partial\mathbf{x}}\mathbf{f}(\,\mathbf{x}^t) \cdot \left(\Delta t\,\mathbf{v}^{t+1}\right) + O\left((\,\Delta t\,\mathbf{v}^{t+1})^2\right) \quad (4)$$

- Plug (4) into (3):

$$M\mathbf{v}^{t+1} = M\mathbf{v}^t + \Delta t\Big(\mathbf{f}(\mathbf{x}^t) + \underbrace{\frac{\partial}{\partial \mathbf{x}}\mathbf{f}(\mathbf{x}^t)}_{K}\cdot\big(\Delta t\mathbf{v}^{t+1}\big)\Big)$$

$$= M\mathbf{v}^t + \Delta t\mathbf{f}(\mathbf{x}^t) + \Delta t^2 K\mathbf{v}^{t+1}$$

- K is the Jacobi-Matrix, i.e., the derivative of **f** wrt. **x**:

$$K = \begin{pmatrix} \frac{\partial}{\partial x_0} f_0 & \frac{\partial}{\partial x_1} f_0 & \cdots & \frac{\partial}{\partial x_{3n-1}} f_0 \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_0} f_{3n-1} & \cdots & \cdots & \frac{\partial}{\partial x_{3n-1}} f_{3n-1} \end{pmatrix}$$

  - K is called the tangent stiffness matrix
    - (The normal stiffness matrix is evaluated at the equilibrium of the system: here, the matrix is evaluated at an arbitrary "position" of the system in phase space, hence the name "*tangent* ...")

- Reorder terms :

$$\left(M - \Delta t^{\,2} K\right) \mathbf{v}^{t+1} = M\,\mathbf{v}^{t} + \Delta t\,\mathbf{f}(\mathbf{x}^{t})$$

- Now, this has the form:

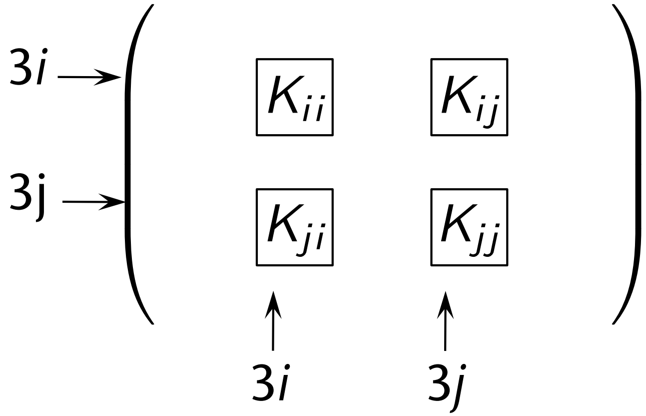
$$A\,\mathbf{v}^{t+1} = \mathbf{b}$$

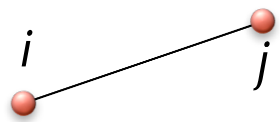
$$\text{mit} \quad A \in \mathbb{R}^{3n \times 3n}, \quad b \in \mathbb{R}^{3n}$$

- Solve this system of linear equations with any of the standard iterative solvers
- Don't use a non-iterative solver, because
  - *A* changes with every simulation step
  - We can "warm start" the iterative solver with the solution as of last frame
    - Incremental computation

Computation of the Stiffness Matrix

- First, understand the anatomy of matrix K :
  - A spring (i, j) adds the following four 3x3 block matrices to K :





  - Matrix K_{ij} arises from the derivation of $\mathbf{f}_i = (f_{3i}, f_{3i+1}, f_{3i+2})$ wrt. $\mathbf{x}_j = (x_{3j}, x_{3j+1}, x_{3j+2})$:

$$K_{ij} = \begin{pmatrix} \frac{\partial}{\partial x_{3j}} f_{3i} & \frac{\partial}{\partial x_{3j+1}} f_{3i} & \frac{\partial}{\partial x_{3j+2}} f_{3i} \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_{3j}} f_{3i+2} & \cdots & \frac{\partial}{\partial x_{3j+2}} f_{3i+2} \end{pmatrix}$$

  - In the following, consider only f^S (spring force)

- First of all, compute K_{ii}:

$$K_{ii} = \frac{\partial}{\partial \mathbf{x}_i} f_i(\mathbf{x}_i, \mathbf{x}_j)$$

$$= k_s \frac{\partial}{\partial \mathbf{x}_i} \left((\mathbf{x}_j - \mathbf{x}_i) - l_0 \frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|} \right)$$

$$= k_s \left(-I - l_0 \frac{-I \cdot \|\mathbf{x}_j - \mathbf{x}_i\| - (\mathbf{x}_j - \mathbf{x}_i) \cdot \frac{(\mathbf{x}_j - \mathbf{x}_i)^\top}{\|\mathbf{x}_j - \mathbf{x}_i\|}}{\|\mathbf{x}_j - \mathbf{x}_i\|^2} \right)$$

$$= k_s \left(-I + l_0 \frac{1}{\|\mathbf{x}_j - \mathbf{x}_i\|} I + \frac{l_0}{\|\mathbf{x}_j - \mathbf{x}_i\|^3} (\mathbf{x}_j - \mathbf{x}_i)(\mathbf{x}_j - \mathbf{x}_i)^\mathsf{T} \right)$$

- Reminder:
  - $$\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$$
  - $$\frac{\partial}{\partial \mathbf{x}} \|\mathbf{x}\| = \frac{\partial}{\partial \mathbf{x}} \left(\sqrt{x_1^2 + x_2^2 + x_3^2} \right) = \frac{\mathbf{x}^\mathsf{T}}{\|\mathbf{x}\|}$$

- From some symmetries, we can analogously derive:
  - $K_{ij} = \frac{\partial}{\partial \mathbf{x}_j} f_i(\mathbf{x}_i, \mathbf{x}_j) = -K_{ii}$
  - $K_{jj} = \frac{\partial}{\partial x_j} f_j(\mathbf{x}_i, \mathbf{x}_j) = \frac{\partial}{\partial \mathbf{x}_j}(-\mathbf{f}_i(\mathbf{x}_i, \mathbf{x}_j)) = K_{ii}$
  - $K_{ji} = K_{ij}$

Overall Algorithm for Solving Implicit Euler Integration

- Initialize $K = 0$
- For each spring (i, j) compute $K_{ii}, K_{ij}, K_{ji}, K_{jj}$ and accumulate it to K at the right places

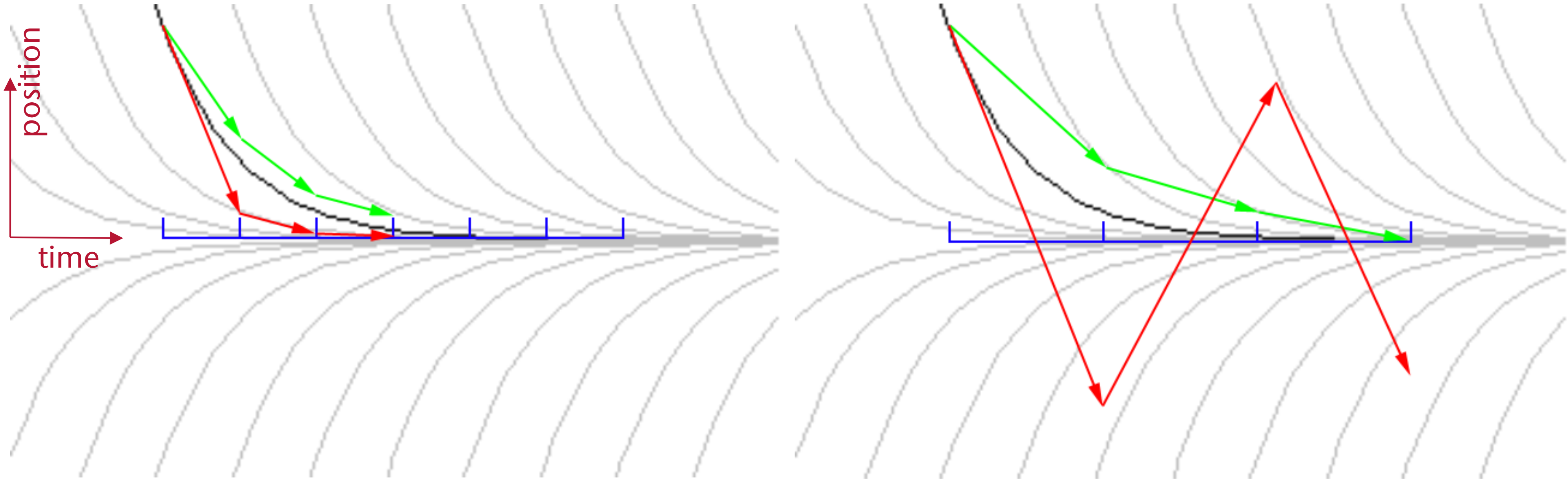
$$\begin{pmatrix} \boxed{K_{ii}} & \boxed{K_{ij}} \\ \boxed{K_{ji}} & \boxed{K_{jj}} \end{pmatrix}$$

- Compute $\mathbf{b} = M\mathbf{v}^t + \Delta t \mathbf{f}(\mathbf{x}^t)$
- Solve the linear equation system $A\mathbf{v}^{t+1} = \mathbf{b} \rightarrow \mathbf{v}^{t+1}$
- Compute $\mathbf{x}^{t+1} = \mathbf{x}^t + \Delta t\, \mathbf{v}^{t+1}$

Advantages and Disadvantages

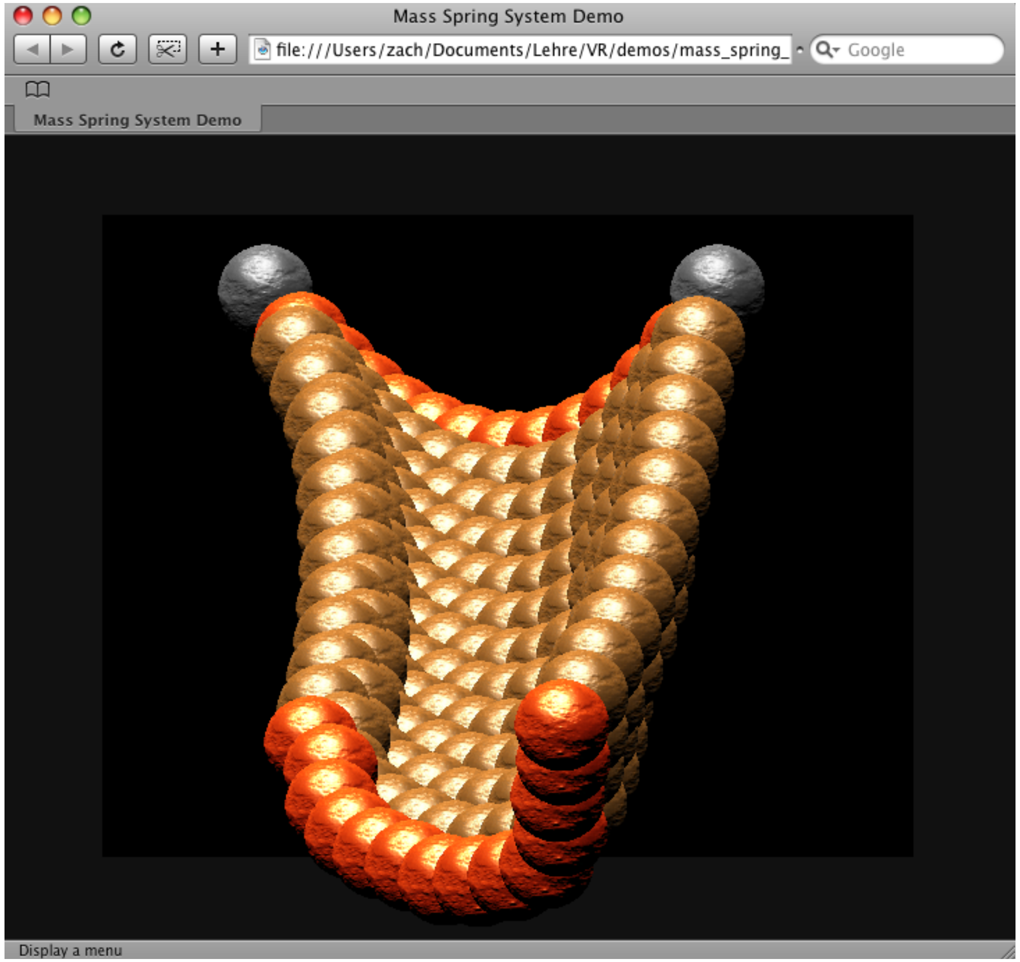
- Explicit integration:
  + Very easy to implement
  - Small step sizes needed
  - Stiff springs don't work very well
  - Forces are propagated only by one spring per time step
- Implicit Integration:
  + Unconditionally stable
  + Stiff springs work better
  + Global solver → forces are being propagated throughout the whole spring-mass system within one time step
  - Large stime steps are needed, because one step is much more expensive (if real-time is needed)
  - The integration scheme introduces damping by itself (might be unwanted)

- Visualization of: $\dot{x}(t) = -x(t)$



- Informal Description:
  - Explicit jumps forward blindly, based on current information
  - Implicit tries to find a future position and a backwards jump such that the backwards jump arrives exactly at the current point (in phase space)

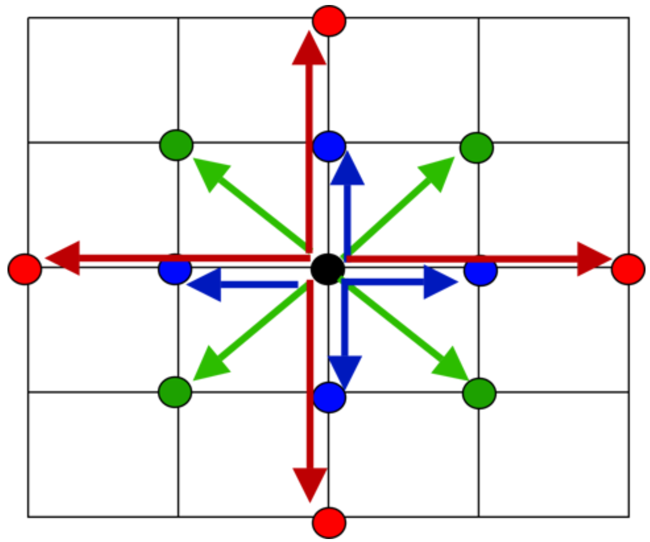
Demo



http://www.dhteumeuleu.com/dhtml/v-grid.html

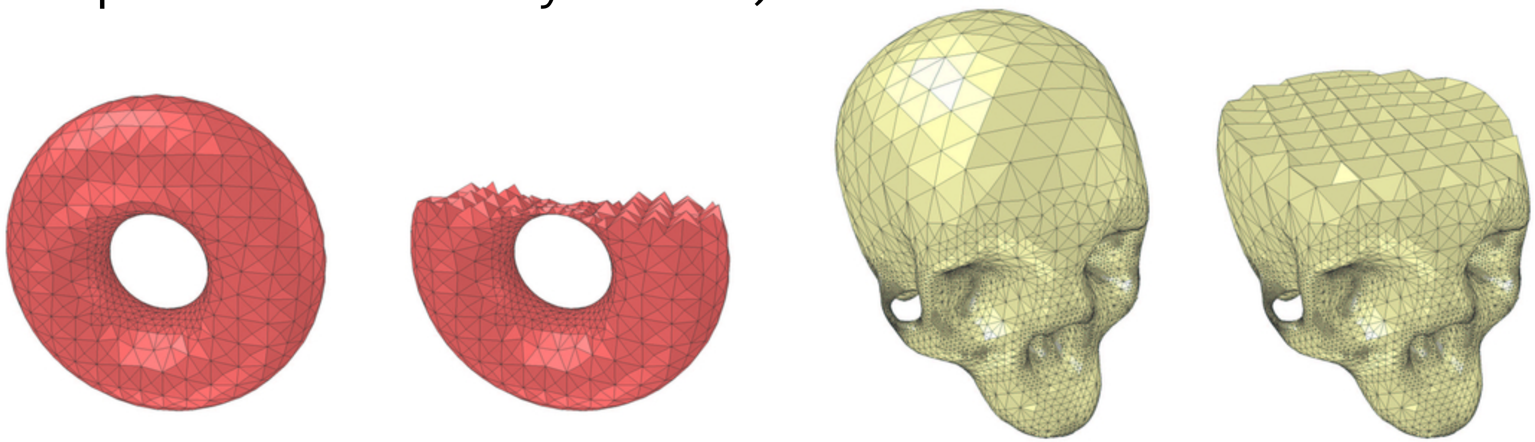
Mesh Creation for Volumetric Objects

- How to create a mass-spring system for a volumetric model?
  - Challenge: volume preservation!
- Approach 1: introduce additional, volume-preserving constraints
  - Springs to preserve distances between mass points
  - Springs to prevent shearing
  - Springs to prevent bending
- No change in model & solver required
- You could also introduce "angle-preserving springs" that exert a torque on an edge



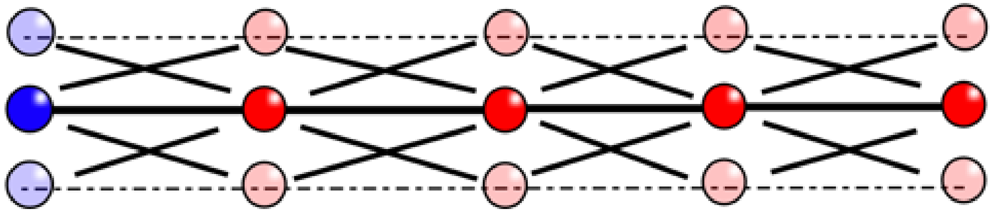
- Approach 2 (and still simple): model the inside volume explicitly
  - Create a tetrahedron mesh out of the geometry (somehow)
  - Each vertex (node) of the tetrahedron mesh becomes a mass point, each edge a spring
  - Distribute the masses of the tetrahedra (= density × volume) equally among the mass points

- Generation of the tetrahedron mesh (simple method):
  - Distribute a number of points uniformly (perhaps randomly) in the interior of the geometry (so called "Steiner points")
  - Dito for a sheet/band above the surface
  - Connect the points by Delaunay triangulation (see my course "Computational Geometry for CG")



- Anchor the surface mesh within the tetrahedron mesh:
  - Represent each vertex of the surface mesh by the *barycentric combination* of its surrounding tetrahedron vertices

- Approach 3: kind of an "in-between" between approaches 1 & 2
  - Create a virtual shell around the two-manifold mesh
  - Connect the shell with the "real" mesh by diagonal springs



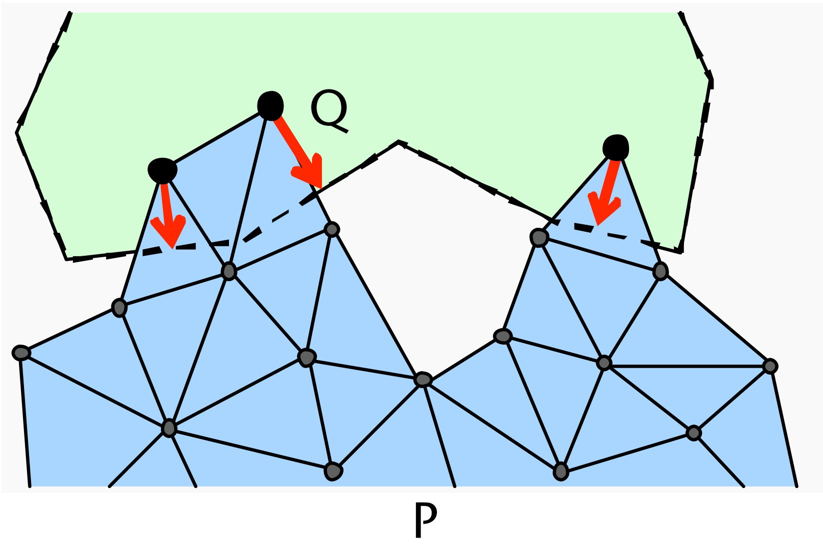
- Video:
  1. no virtual shells,
  2. one virtual shell,
  3. several virtual shells

Collision Detection for Mass-Spring Systems

- Put all tetrahedra in a 3D grid (use a hash table!)
- In case of a collision in the hash table:
  - Compute exact intersection between the 2 involved tetrahedra

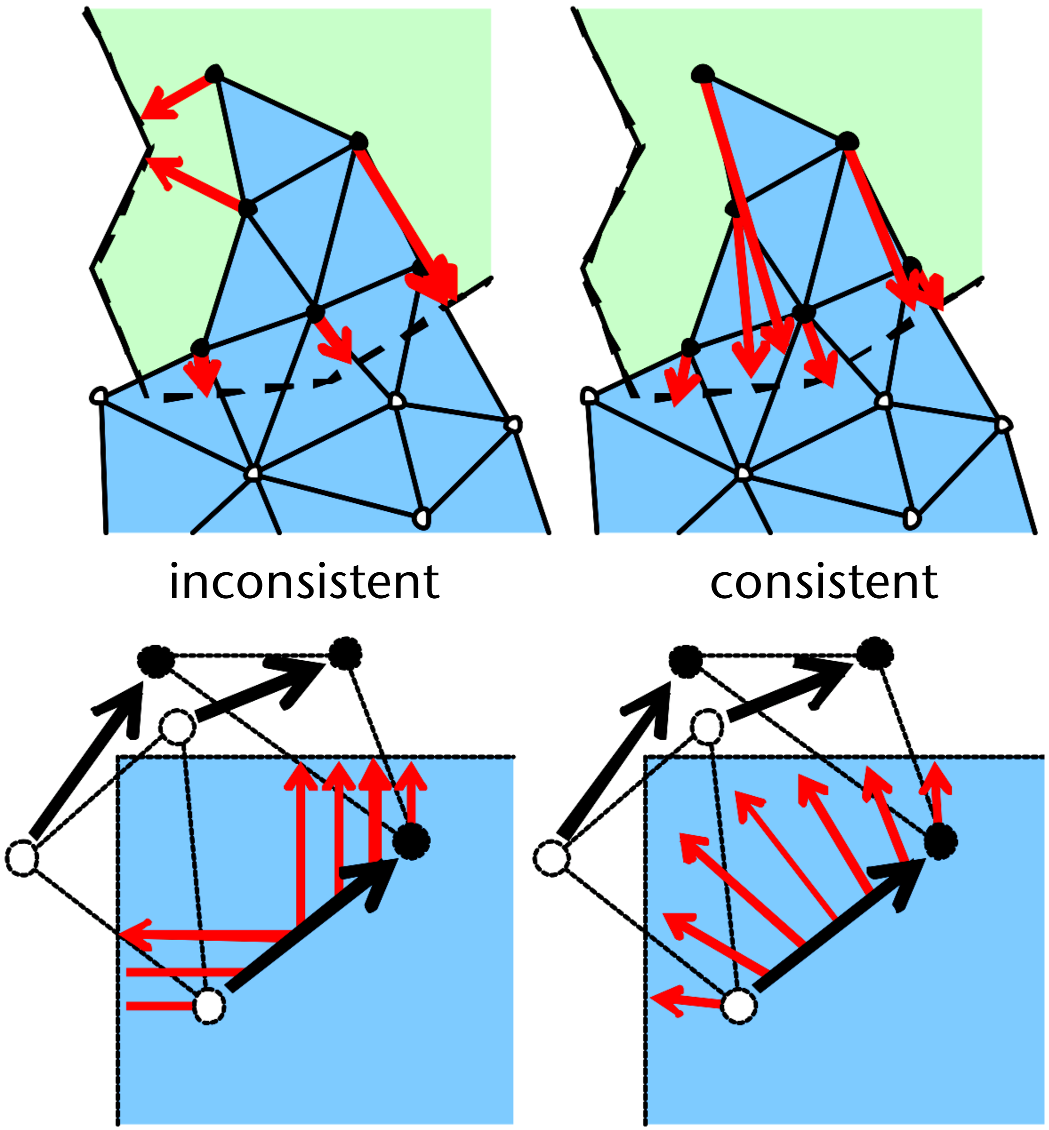
Collision Response

- Given: objects P and Q (= tetrahedral meshes) that collide
- Task: compute a penalty force
- Naïve approach:
  - For each mass point of P that has penetrated, compute its closest distance from the surface of Q → force = amount + direction



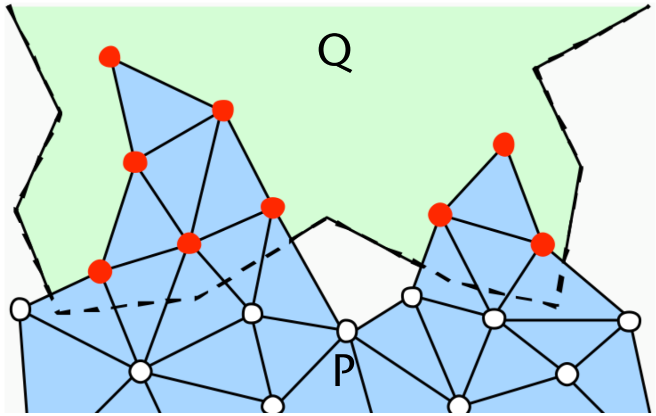
- Problem:
  - Implausible forces
  - "Tunneling" (s. a. the chapter on force-feedback)

- Examples:

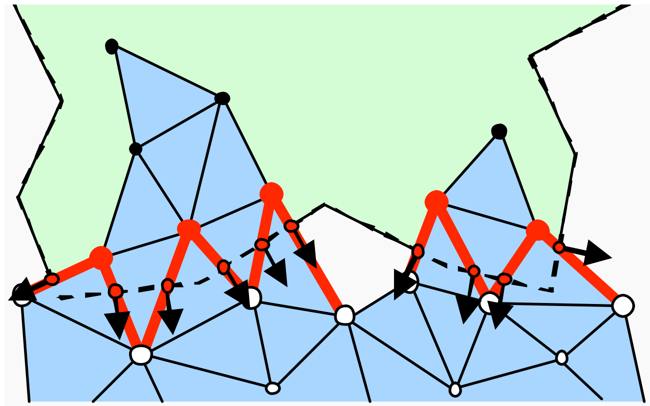


Consistent Penalty Forces

1. Phase: identify all points of P that penetrate Q

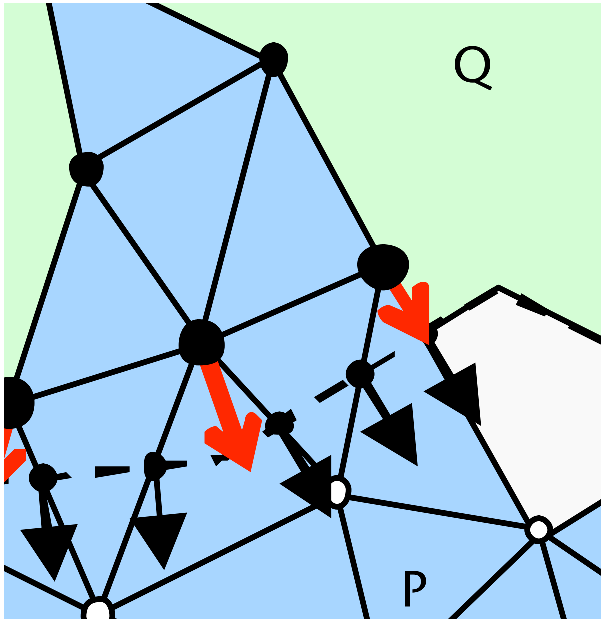


2. Phase: determine all edges of P that intersect the surface of Q
   - For each such edge, compute the exact intersection point $\mathbf{x}_i$
   - For each intersection point, compute a normal $\mathbf{n}_i$
     - E.g., by barycentric interpolation of the vertex normals of Q



3. Phase: compute the approximate force for border points

- Border point = a point **p** that penetrates Q and is incident to an intersecting edge
- Observation: a border point can be incident to several intersecting edges
- Set the penetration depth for point p to



$$d(\mathbf{p}) = \frac{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p}) \, (\mathbf{x}_i - \mathbf{p}) \cdot \mathbf{n}_i}{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p})}$$

where $d(\mathbf{p})$ = approx. penetration depth of mass point **p**, $\mathbf{x}_i$ = point of the intersection of an edge incident to **p** with surface Q, $\mathbf{n}_i$ = normal to surface of Q at point $\mathbf{x}_i$,

and $\omega(\mathbf{x}_i, \mathbf{p}) = \frac{1}{\|\mathbf{x}_i - \mathbf{p}\|}$

- Direction of the penalty force on border points:

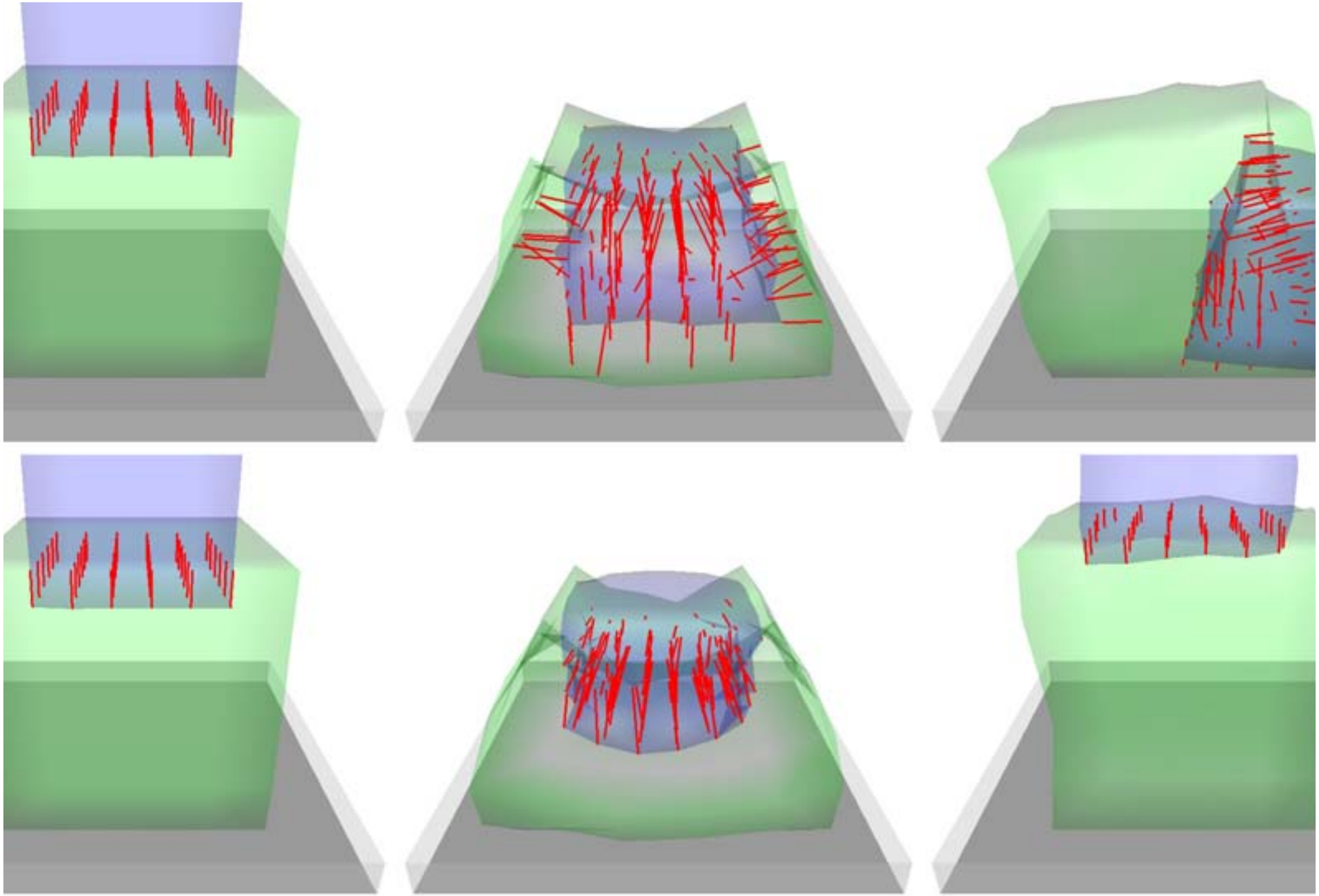
$$\mathbf{r}(\mathbf{p}) = \frac{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p})\, \mathbf{n}_i}{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p})}$$

4. Phase: propagate forces by way of breadth-first traversal through the tetrahedron mesh

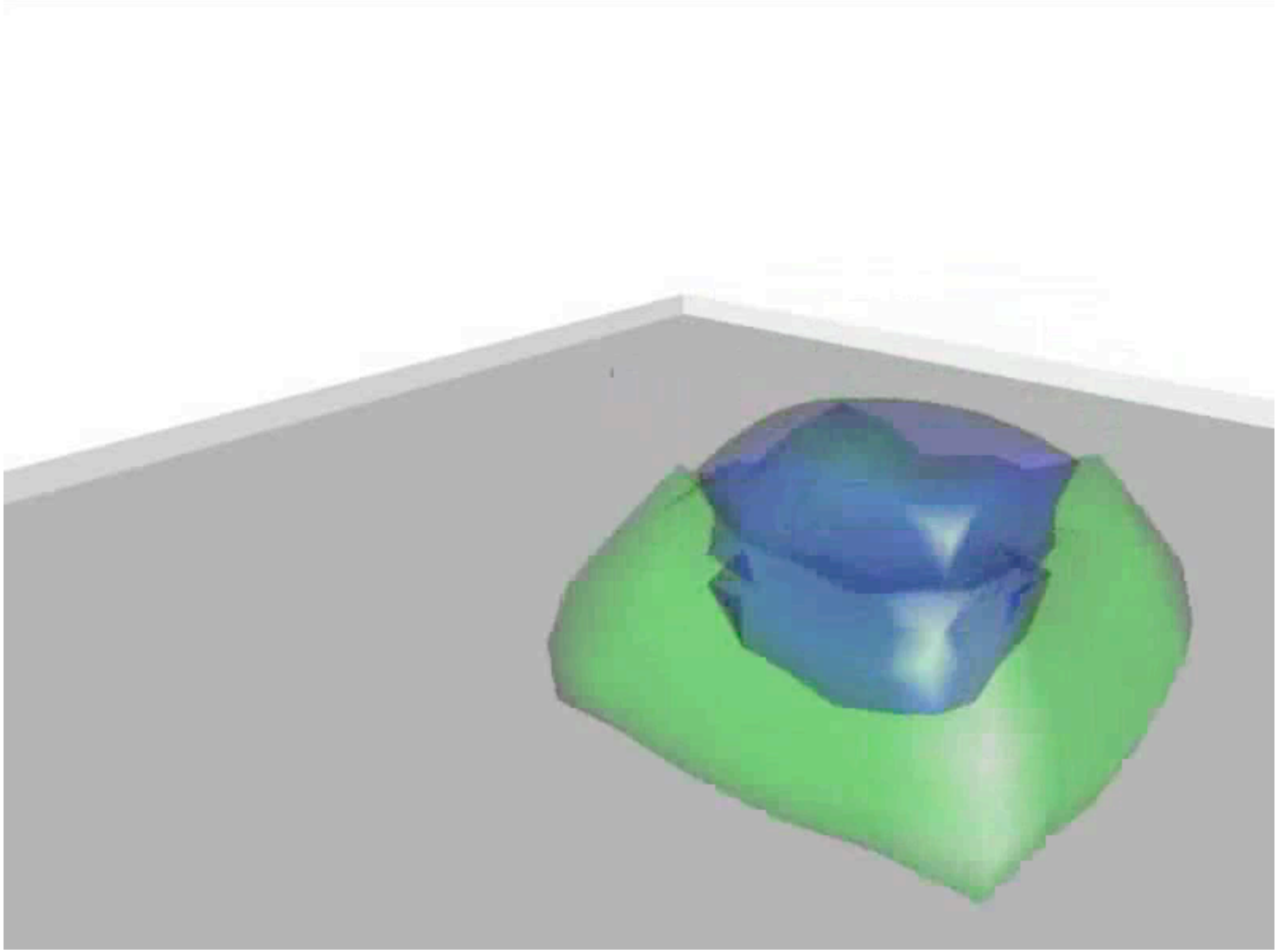
$$d(\mathbf{p}) = \frac{\sum_{i=1}^{k} \omega(\mathbf{p}_i, \mathbf{p})\big((\mathbf{p}_i - \mathbf{p})\cdot\mathbf{r}_i + d(\mathbf{p}_i)\big)}{\sum_{i=1}^{k} \omega(\mathbf{x}_i, \mathbf{p})}$$

where $\mathbf{p}_i$ = points of P that have been visited already, $\mathbf{p}$ = point not yet visited, $\mathbf{r}_i$ = direction of the estimated penalty force in point $\mathbf{p}_i$.

Visualization



Video



http://cg.informatik.uni-freiburg.de