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 particle system consisting of:
  1. A set of point masses $m_i$ with positions $x_i$ and velocities $v_i$, $i = 1...n$;
  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 $k_d$.

• Typical spring topology:
Some Properties

• 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
Forces in a Single Spring (Plus Damper)

• Given: masses $m_i$ and $m_j$ with positions $x_i$, $x_j$

• Let $r_{ij} = \frac{x_j - x_i}{\|x_j - x_i\|}$

• The force between particles $i$ and $j$:

1. Force exerted by spring (Hooke's law):
   $$f_{s}^{ij} = k_s r_{ij}(\|x_j - x_i\| - l_0)$$
   acts on particle $i$ in the direction of $j$

2. Force exerted on $i$ by damper: $f_{d}^{ij} = -k_d ((v_i - v_j) \cdot r_{ij}) r_{ij}$

3. Total force on $i$:
   $$f^{ij} = f_{s}^{ij} + f_{d}^{ij}$$

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

• A spring-damper element in reality:

• Alternative spring force: \( f_{s}^{ij} = k_{s} r_{ij} \frac{\| x_{j} - x_{i} \| - l_{0}}{l_{0}} \)

• Notice: from (4) it follows that the total momentum is conserved

  • Momentum \( p = \mathbf{v} \cdot m \)

  • Fundamental physical law (follows from Newton's laws)

• Note on terminology:

  • English "momentum" = German "Impuls" = velocity \( \times \) mass

  • English "Impulse" = German "Kraftstoß" = force \( \times \) time
Simulation of a Single Spring

• From Newton’s law, we have:  \( \ddot{x} = \frac{1}{m} f \)

• Convert this differential equation (ODE) of order 2 into ODE of order 1:
  \[
  \dot{x}(t) = v(t) \\
  \dot{v}(t) = \frac{1}{m} f(t)
  \]

• Initial values (boundary values):  \( v(t_0) = v_0, \ x(t_0) = x_0 \)

• By Taylor expansion we get:  \( x(t + \Delta t) = x(t) + \Delta t \dot{x}(t) + O(\Delta t^2) \)

• Analogously:  \( v(t + \Delta t) = v(t) + \Delta t \dot{v}(t) \)

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

• "Simulation" = "Integration of ODE's over time"
The Algorithm for a Mass-Spring System

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

loop forever:
forall particles $i$ :

\[
f_i \leftarrow f^g + f^\text{coll} + \sum_{j,(i,j) \in S} f(x_i, v_i, x_j, v_j)
\]

forall particles $i$ :

\[
v_i + = \Delta t \cdot \frac{f_i}{m_i}
\]

\[
x_i + = \Delta t \cdot v_i
\]

render the system every $n$-th time

\[
f^g = \text{gravitational force}
\]

\[
f^\text{coll} = \text{penalty force exerted by collision (e.g., from obstacles)}
\]
• 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 $\Delta t > \frac{1}{k}$: $x^{t+1} = x^t \left(1 - k\Delta t\right)$
  
  $< 0$

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

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

- Terminology: if \( k \) is large \( \rightarrow \) the ODE is called "stiff"
  - The stiffer the ODE, the smaller \( \Delta t \) has to be!
Visualization of Error Accumulation

• Consider this ODE:
  \[ \dot{x}(t) = \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix} \]

• Exact solution:
  \[ x(t) = \begin{pmatrix} r \cos(t + \phi) \\ r \sin(t + \phi) \end{pmatrix} \]

• The solution by Euler integration moves in spirals outward, no matter how small \( \Delta t \)!

• Conclusion: Euler integration accumulates errors, no matter how small \( \Delta t \)!
Visualization of Differential Equations

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

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

- 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 \( x( t + \frac{1}{2} \Delta t ) \)
  • The integrator (w/o proof):
    \[
    a_1 = v^t \\
    a_2 = \frac{1}{m} f(x^t, v^t) \\
    b_1 = v^t + \frac{1}{2} \Delta t a_2 \\
    b_2 = \frac{1}{m} f(x^t + \frac{1}{2} \Delta t a_1, v^t + \frac{1}{2} \Delta t a_2) \\
    x^{t+1} = x^t + \Delta t b_1 \\
    v^{t+1} = v^t + \Delta t b_2
    \]

• 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: \( e(\Delta t) = O(\Delta t^4) \)
• Runge-Kutta of order 2:

\[ y = y(x) \]

\[ x_0 \]

\[ x_n \]

\[ x_{n+1} = x_n + h \]

\[ y = y(x) \]

\[ x_n \]

\[ x_{n+1} \]

Euler

• 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 $x(t - \Delta t)$

- Derivation:
  - Develop the Taylor series in both time directions:

\[
x(t + \Delta t) = x(t) + \Delta t \ddot{x}(t) + \frac{1}{2} \Delta t^2 \dddot{x}(t) + \frac{1}{6} \Delta t^3 \ddddot{x}(t) + O(\Delta t^4)
\]

\[
x(t - \Delta t) = x(t) - \Delta t \ddot{x}(t) + \frac{1}{2} \Delta t^2 \dddot{x}(t) - \frac{1}{6} \Delta t^3 \ddddot{x}(t) + O(\Delta t^4)
\]
• Add both:

\[ x(t + \Delta t) + x(t - \Delta t) = 2x(t) + \Delta t^2 \ddot{x}(t) + O(\Delta t^4) \]

\[ x(t + \Delta t) = 2x(t) - x(t - \Delta t) + \Delta t^2 \ddot{x}(t) + O(\Delta t^4) \]

• Initialization:

\[ x(\Delta t) = x(0) + \Delta t v(0) + \frac{1}{2} \Delta t^2 \left( \frac{1}{m} f(x(0), v(0)) \right) \]

• 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

\[ \| \mathbf{x}_1 - \mathbf{x}_2 \| = l_0 \]

1. Perform one Verlet integration step \( \rightarrow \ddot{x}^{t+1} \) (tentative)

2. Enforce the constraint:

\[
d = \frac{1}{2} (\| \ddot{x}_2^{t+1} - \ddot{x}_1^{t+1} \| - l_0)
\]

\[
\mathbf{x}_1^{t+1} = \ddot{x}_1^{t+1} + d \mathbf{r}_{12}
\]

\[
\mathbf{x}_2^{t+1} = \ddot{x}_2^{t+1} - d \mathbf{r}_{12}
\]

• 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 $\Delta t$'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:

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

  • Divide the expansions by $\Delta t_i$ and $\Delta t_{i-1}$, respectively, then add both, like in the derivation of the basic Verlet

  • Rearranging and omitting higher-order terms yields:

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

• 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 $\Delta t$
  • This range depends on the stiffness of the springs

• Goal: unconditionally stability

• One option: implicit Euler integration

  explicit
  \[ x_i^{t+1} = x_i^t + \Delta t v_i^t \]
  \[ v_i^{t+1} = v_i^t + \Delta t \frac{1}{m_i} f(x^t) \]

  implicit
  \[ x_i^{t+1} = x_i^t + \Delta t v_i^{t+1} \]
  \[ v_i^{t+1} = v_i^t + \Delta t \frac{1}{m_i} f(x^{t+1}) \]

• Now we've got a system of non-linear, algebraic equations, with $x_i^{t+1}$ and $v_i^{t+1}$ as unknowns on both sides → implicit integration
Solution Method

• Write the whole spring-mass system with vectors ($n = \#\text{mass points}$):

$$
\mathbf{x} = \begin{pmatrix}
    x_0 \\
    x_1 \\
    \vdots \\
    x_{n-1} \\
    x_{3n-1}
\end{pmatrix}, \quad
\mathbf{v} = \begin{pmatrix}
    v_0 \\
    v_1 \\
    \vdots \\
    v_{n-1} \\
    v_{3n-1}
\end{pmatrix}, \quad
\mathbf{f}(\mathbf{x}) = \begin{pmatrix}
    f_0(\mathbf{x}) \\
    \vdots \\
    f_{n-1}(\mathbf{x})
\end{pmatrix}
$$

\[
\mathbf{f}_i = \begin{pmatrix}
    f_{3i+0}(\mathbf{x}) \\
    f_{3i+1}(\mathbf{x}) \\
    f_{3i+2}(\mathbf{x})
\end{pmatrix}, \quad
\mathbf{M}_{3n \times 3n} = \begin{pmatrix}
    m_0 & m_0 & m_0 \\
    m_0 & m_1 & m_1 \\
    m_0 & m_1 & \ddots \\
    & m_{n-1} & m_{n-1} & m_{n-1}
\end{pmatrix}
\]
• Write all the implicit equations as one big system of equations:

\[ Mv^{t+1} = Mv^t + \Delta tf(x^{t+1}) \]  \hspace{1cm} (1)

\[ x^{t+1} = x^t + \Delta t v^{t+1} \]  \hspace{1cm} (2)

• Plug (2) into (1):

\[ Mv^{t+1} = Mv^t + \Delta t f(x^t + \Delta tv^{t+1}) \]  \hspace{1cm} (3)

• Expand f as Taylor series:

\[ f(x^t + \Delta t v^{t+1}) = f(x^t) + \frac{\partial}{\partial x} f(x^t) \cdot (\Delta t v^{t+1}) + O((\Delta t v^{t+1})^2) \]  \hspace{1cm} (4)
Plug (4) into (3): \[ Mv^{t+1} = Mv^t + \Delta t \left( f(x^t) + \frac{\partial}{\partial x} f(x^t)(\Delta tv^{t+1}) \right) \]

\[ = Mv^t + \Delta tf(x^t) + \Delta t^2 K v^{t+1} \]

- \( K \) 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)
• Now reorder terms:

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

• Now, this has the form:

\[A \mathbf{v}^{t+1} = \mathbf{b}\]

mit \(A \in \mathbb{R}^{3n \times 3n}, \quad \mathbf{b} \in \mathbb{R}^{3n}\)

• 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 of all, understand the anatomy of matrix $K$:
  - A spring $(i,j)$ adds the following four $3 \times 3$ block matrices to $K$:

- Block matrix $K_{ij}$ arises from the derivation of $f_i = (f_{3i}, f_{3i+1}, f_{3i+2})$ wrt. $x_j = (x_{3j}, x_{3j+1}, x_{3j+2})$:

- In the following, consider only $f^s$ (spring force)
• First of all, compute $K_{ii}$:

$$K_{ii} = \frac{\partial}{\partial x_i} f_i(x_i, x_j)$$

$$= k_s \frac{\partial}{\partial x_i} \left( (x_j - x_i) - l_0 \frac{x_j - x_i}{\|x_j - x_i\|} \right)$$

$$= k_s \left( -I - l_0 \frac{-I \cdot \|x_j - x_i\| - (x_j - x_i) \cdot \frac{(x_j - x_i)^T}{\|x_j - x_i\|}}{\|x_j - x_i\|^2} \right)$$

$$= k_s \left( -I + l_0 \frac{1}{\|x_j - x_i\|} I + \frac{l_0}{\|x_j - x_i\|^3} (x_j - x_i)(x_j - x_i)^T \right)$$
• Reminder:

\[
\left( \frac{f}{g} \right)' = \frac{f'g - fg'}{g^2}
\]

\[
\frac{\partial}{\partial \mathbf{x}} \|\mathbf{x}\| = \frac{\partial}{\partial \mathbf{x}} \left( \sqrt{x_1^2 + x_2^2 + x_3^2} \right) = \frac{\mathbf{x}^T}{\|\mathbf{x}\|}
\]
• From some symmetries, we can analogously derive:

• \( K_{ij} = \frac{\partial}{\partial x_j} f_i(x_i, x_j) = -K_{ii} \)

• \( K_{jj} = \frac{\partial}{\partial x_j} f_j(x_i, x_j) = \frac{\partial}{\partial x_j} (-f_i(x_i, x_j)) = K_{ii} \)

• \( K_{ji} = K_{ij} \)
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 into $K$ at the right places

- 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 time steps needed, b/c 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 \times 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 outside the surface
  • Connect the points by Delaunay triangulation (see my course "Computational Geometry")

• 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

inconsistent  consistent  inconsistent  consistent
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 \( x_i \)
   - For each intersection point, compute a normal \( n_i \)
     - 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
- Approximate the penetration depth for point \( p \) by

\[
d(p) = \frac{\sum_{i=1}^{k} \omega(x_i, p) (x_i - p) \cdot n_i}{\sum_{i=1}^{k} \omega(x_i, p)}
\]

where \( x_i = \) point of the intersection of an edge incident to \( p \) with surface \( Q \),
\( n_i = \) normal to surface of \( Q \) at point \( x_i \),
and \( \omega(x_i, p) = \frac{1}{\|x_i - p\|} \)
• Set the direction of the penalty force on border points:

\[
    r(p) = \frac{\sum_{i=1}^{k} \omega(x_i, p) n_i}{\sum_{i=1}^{k} \omega(x_i, p)}
\]

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

\[
    d(p) = \frac{\sum_{i=1}^{k} \omega(p_i, p)((p_i - p) \cdot r_i + d(p_i))}{\sum_{i=1}^{k} \omega(x_i, p)}
\]

where \( p_i \) = points of \( P \) that have been visited already, \( p \) = point not yet visited, \( r_i \) = direction of the estimated penalty force in point \( p_i \).
Video

http://cg.informatik.uni-freiburg.de
Art with Mass-Spring Systems