fvlib release 4

http://prototy.blogspot.com

Description

fvlib is a minimalistic point-spring solver library, built with high-performance in mind. It uses Verlet Integration for the calculation of point trajectories and Dynamic Relaxation for the calculation of constraints.

Release 2 added support for parallel processing of points and links, via the *java.util.concurrent* framework. Using the 'fast' option, the library has been tested on a set of 9800 points and 58779 links. Using 15 steps, each iteration was measured to 15-16 msec (using java's *System.currentTimeMillis()*), on a 2.2GHz Intel Core 2 Duo. In other words, more than a million point and spring loops can be performed in realtime on a typical contemporary computer.
Furthermore, compared to the performance index of 20-24 msec of the first release of fvlib, release 2 of the library essentially increases performance by about 50% on a dual-core computer. Expected performance may be higher for computers with more than two processors.

In release 3, the library's structure was completely rewritten. This resulted in a highly modular class structure that will allow the easy expansion of the library's functionality. Some extra functionality has been also implemented, namely the simulation of "charged" Particles Interaction Solvers.
Furthermore, most property-setting methods of classes implement the return of the instance itself so that adjustments can be chained together in one line, therefore saving some typing and promoting ease of reading.

Release 4 fixes a few bugs and implements a couple of new features, namely Unyielding points of controllable slip and Constant Force Modifier/Solver. The implementation of point weight in this release has been removed, will be properly re-introduced in a future release.

Installation

Unzip and move the extracted folder to Processing's libraries folder. Use Sketch>Import Library>fvlib from within Processing.

Classes/Constructors

Solvers

- **Solver**: Abstract class that lays the foundation for parallel processing of solver algorithms and implements a few useful functions. It is not necessary to instantiate this class under normal usage of the library.

  ```
  Solver()
  ```

- **IntegratorVerlet**: An integrator based on the Verlet Integration scheme, as described by Thomas Jakobsen in "Advanced Character Physics".

  ```
  IntegratorVerlet()
  IntegratorVerlet(ArrayList<Point> points)
  IntegratorVerlet(Point[] points)
  ```
  Note: If the second constructor is used (Arrays), the class retains a reference to the arrays supplied themselves; if you perform a change in the arrays it will be automatically seen by the solver.

- **SolverRelaxation**: A relaxation solver using the Link class.

  ```
  SolverRelaxation()
  SolverRelaxation(ArrayList<Link> links)
  SolverRelaxation(Link[] links)
  ```
  Note: If the second constructor is used (Arrays), the class retains a reference to the arrays supplied themselves; if you perform a change in the arrays it will be automatically seen by the solver.

- **SolverStaticSingle**: A solver used for the simulation of similar to electrostatic forces within a group of points.

```
SolverStaticSingle()
SolverStaticSingle(ArrayList<Point> points)
SolverStaticSingle(Point[] points)
```

Note: If the second constructor is used (Arrays), the class retains a reference to the arrays supplied themselves; if you perform a change in the arrays it will be automatically seen by the solver.

- **SolverStaticPair**: A solver used for the simulation of similar to electrostatic forces between two groups of points (no forces between points of the same group are calculated).

```
SolverStaticPair()
SolverStaticPair(ArrayList<Point> points1, ArrayList<Point> points2)
SolverStaticPair(Point[] points1, Point[] points2)
```

Note: If the second constructor is used (Arrays), the class retains a reference to the arrays supplied themselves; if you perform a change in the arrays it will be automatically seen by the solver.

- **ConstantForce**: Adds a constant force specified by a PVector to the specified array of Points

```
ConstantForce()
ConstantForce(ArrayList<Point> points)
ConstantForce(Point[] points)
```

Note: If the second constructor is used (Arrays), the class retains a reference to the arrays supplied themselves; if you perform a change in the arrays it will be automatically seen by the solver.

Actor Objects

- **Point**: The point class. Has properties to define weight and fix it to a position(unyielding). Extends the *PVector* class.

```
Point(float x, float y, float z, float weight)
Point(PVector v, float weight)
Point(float x, float y, float z, float weight, float charge, float range)
Point(PVector v, float weight, float charge, float range)
```

- **Link**: The link class. Takes two points as arguments. Properties are it's length and it's stiffness.

```
Link(Point p1, Point p2, float stiffness)
```

Public Fields

In Point:

```
public float x,y,z
```
Current point coordinates.

```
public PVector sforce
```
Force vector and accumulator. Is reset every time IntegratorVerlet.step() is called.

In Link:

```
public Point p1,p2
```
Points associated with the link.

```
public float L,C,S
```
Current Length(L), Original Length(C), Stifness(S).

Methods

In Solver:

```
void step()
```
Steps the simulation by one iteration. Inherited to all specific solver classes (described below).

In IntegratorVerlet:

`IntegratorVerlet setF(float f)`
Set the Friction constant (0-1, default 0.99).

`float getF()`
Return the current Friction constant.

`IntegratorVerlet setP(Point[] points)`
Re-set the array of points in use.

`IntegratorVerlet setP(ArrayList<Point> points)`
Re-set the array of points in use.

`Point[] getP()`
Return the array of points in use.

In SolverRelaxation:

`SolverRelaxation setFast(boolean fast)`
Set whether to use the fast (but less accurate) solver. Default false.

`SolverRelaxation setL(Link[] links)`
Re-set the array of links in use.

`SolverRelaxation setL(ArrayList<Link> links)`
Re-set the array of links in use.

`Link[] getL()`
Return the array of links in use.

In SolverStaticSingle:

`SolverStaticSingle setP(Point[] points)`
Re-set the array of points in use.

`SolverStaticSingle setP(ArrayList<Point> points)`
Re-set the array of points in use.

`Point[] getP()`
Return the array of points in use.

`SolverStaticSingle setBias(float bias)`
Set the bias (Number added to the distance when calculating reaction forces).

`SolverStaticSingle setInv(boolean inv)`
Set whether to invert forces (normally: ++ repel , +- attract).

`float getBias()`
Return the bias.

`boolean getInv()`
Return whether forces are inverted.

In SolverStaticPair:

`SolverStaticPair setP(Point[] points1, Point[] points2)`
Re-set the two arrays of points in use.

`SolverStaticPair setP(ArrayList<Point> points1, ArrayList<Point> points2)`
Re-set the two arrays of points in use.

`Point[][] getP()`
Return the arrays of points in use.

`SolverStaticPair setBias(float bias)`
Set the bias (Number added to the distance when calculating reaction forces).

`SolverStaticPair setInv(boolean inv)`
Set whether to invert forces (normally: ++ repel , +- attract).

`float getBias()`
Return the bias.

`boolean getInv()`
Return whether forces are inverted.

In ConstantForce:

`ConstantForce setP(Point[] points)`
Re-set the two array of points in use.

`ConstantForce setP(ArrayList<Point> points)`
Re-set the two array of points in use.

`Point[] getP()`
Return the array of points in use.

`ConstantForce setF(PVector f)`
Set the constant force vector.

`PVector getF()`
Get the constant force vector.

In Point:

`Point copy()`
Return a copy of this point.

`Point setW(float weight)`
Set the weight of this point. Default 1.

`float getW()`
Return the weight of this point.

`Point setC(float charge)`
Set the charge of this point. Used for electrostatic interaction calculations. Default 0.

`float getC()`
Return the charge of this point.

`Point setR(float range)`
Set the range of this point. Used for electrostatic interaction calculations. Default 0.

`float getR()`
Return the range of this point.

`Point setU(boolean u)`
Set whether this point is Unyielding.

`boolean getU()`
Return whether this point is Unyielding.

`Point setUMult(PVector uMult)`
Set the multiplier for Unyielding points. This can be used to restrain points in a single axis (e.g. by using PVector (0,1,1), the point is restrained in the X-axis). For stiff points just leave as it is.

`PVector getUMult()`
Get the unyielding multiplier.

`Point setPos(PVector pos)`
Set the position of the point to the specified value. This modifies both current (this) and previous(this.old) positions.

`PVector getV()`
Get the velocity vector (current-old position).

In Link:

`float updateC()`
Update original length (C) to reflect the current distance between points.

`Link setC(float cin)`
Set the original length.

`float getC()`
Get the original length.

`float getL()`
Get the current length.

`float updateL()`
Update the current length(L) to reflect the current distance between points.
This should be used whenever the current length should be queried, since during normal operation the current length is not updated.

`Link setP1(Point p)`
Set Point 1.

`Link setP2(Point p)`
Set Point 2.

`Point getP1()`
Get Point 1.

`Point getP2()`
Get Point 2.

License