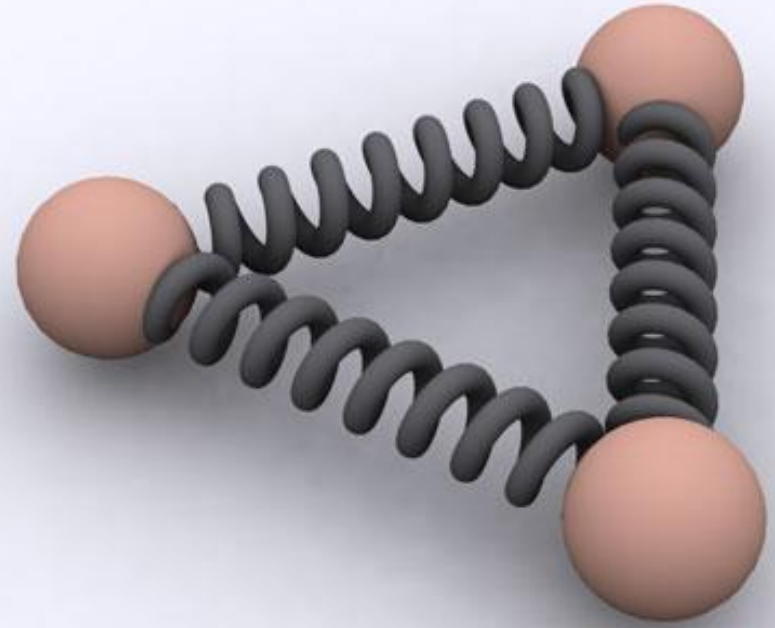
Cloth Simulation with Particle Mechanics

#11

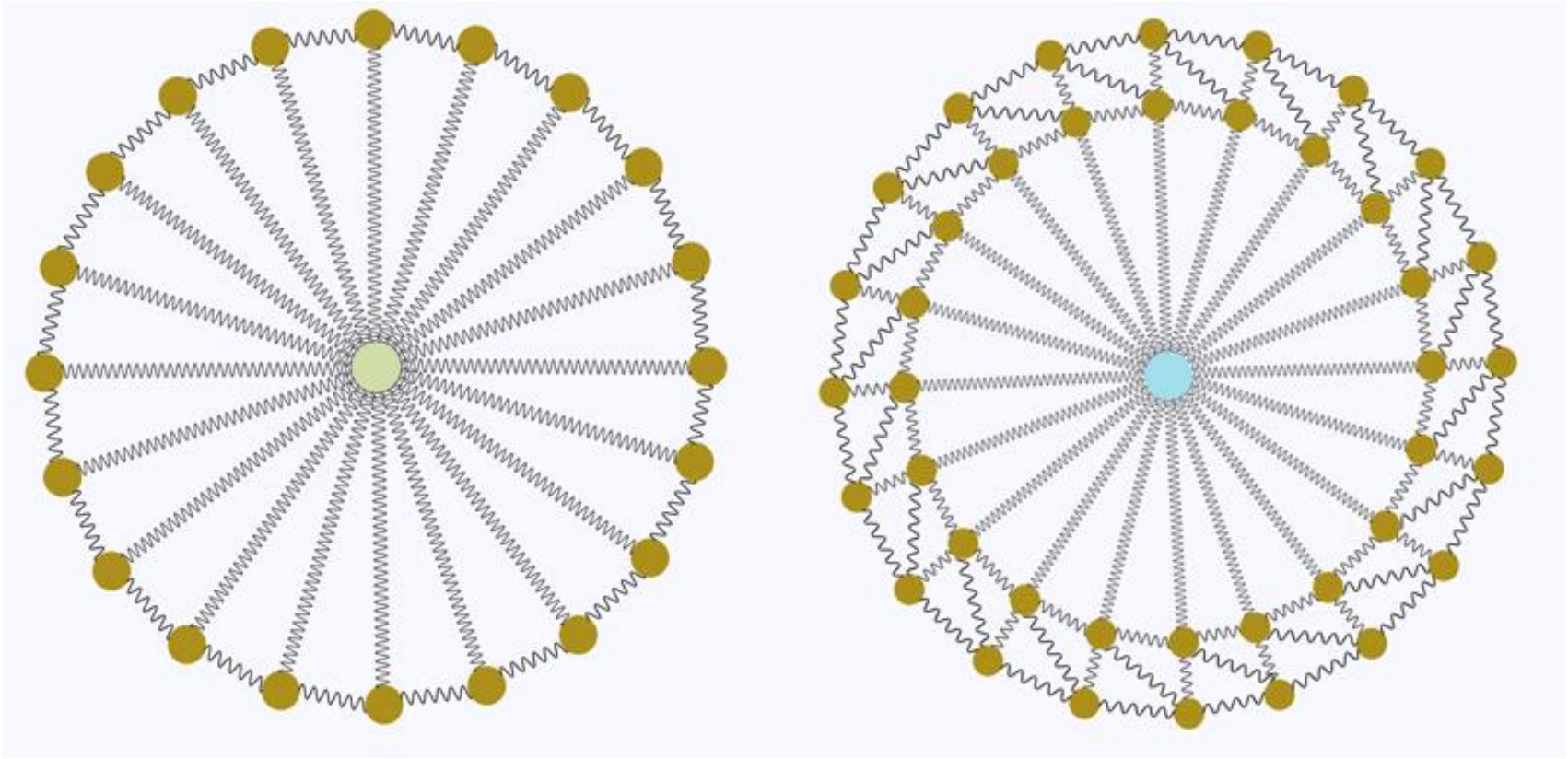
Serdar ARITAN

Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

MASS SPRING SYSTEM



PHYSICS in COMPUTER ANIMATIONS and GAMES



Cloth Animation

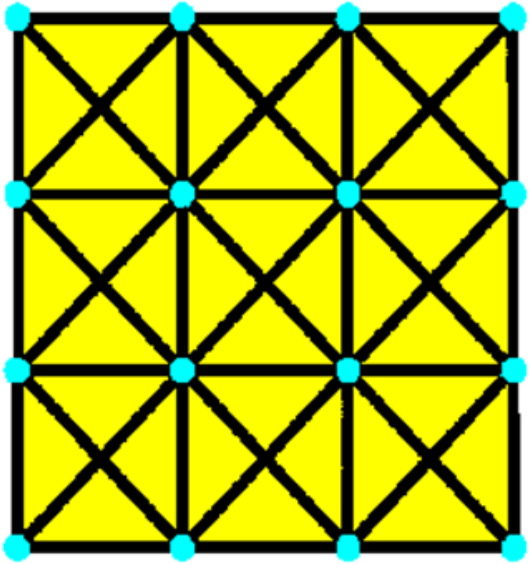
Cloth modeling is the term used for simulating cloth within a computer program, usually in the context of 3D computer graphics. The main approaches used for this may be classified into three basic types: geometric, physical, and particle/energy.

Most models of cloth are based on "particles" of mass connected in some manner of mesh. Newtonian Physics is used to model each particle through the use of a "black box" called a physics engine. This involves using the basic law of motion (Newton's Second Law)

Cloth Animation

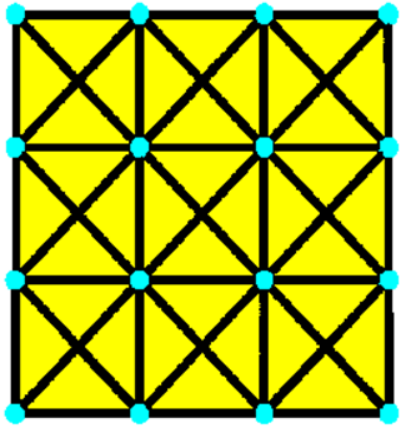
Choose Underlying Model :Mass-Spring

- Easy to understand and implement
- Not as physically accurate as other models

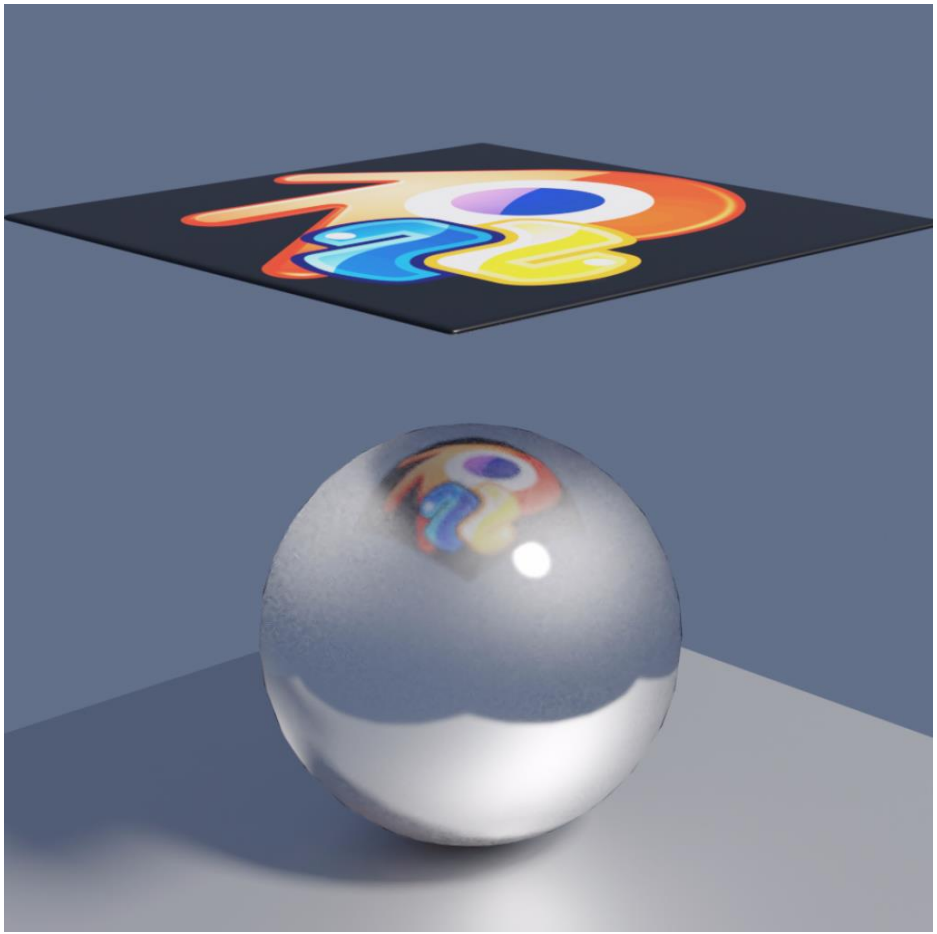


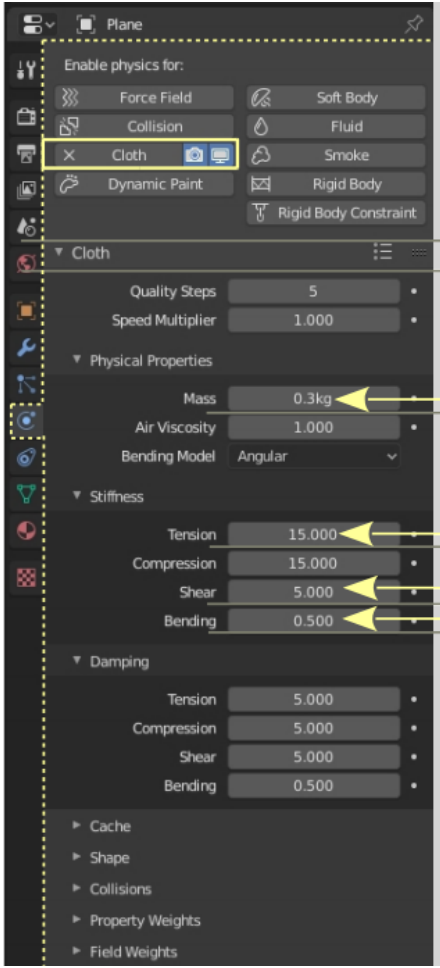
Cloth Animation

- Consider the sheet of cloth.
- Divide it up into a series of approximately evenly spaced masses M.
- Connect nearby masses by a spring, and use Hooke's Law and Newton's 2nd Law as the equations of motion.
- Various additions, such as spring damping or angular springs, can be made.
- A mesh structure proves invaluable for storing the cloth and performing the simulation directly on it.
- Each vertex can store all of its own local information (velocity, position, forces, etc.) and when it comes time to render, the face information allows for immediate rendering as a normal mesh.



Cloth Animation



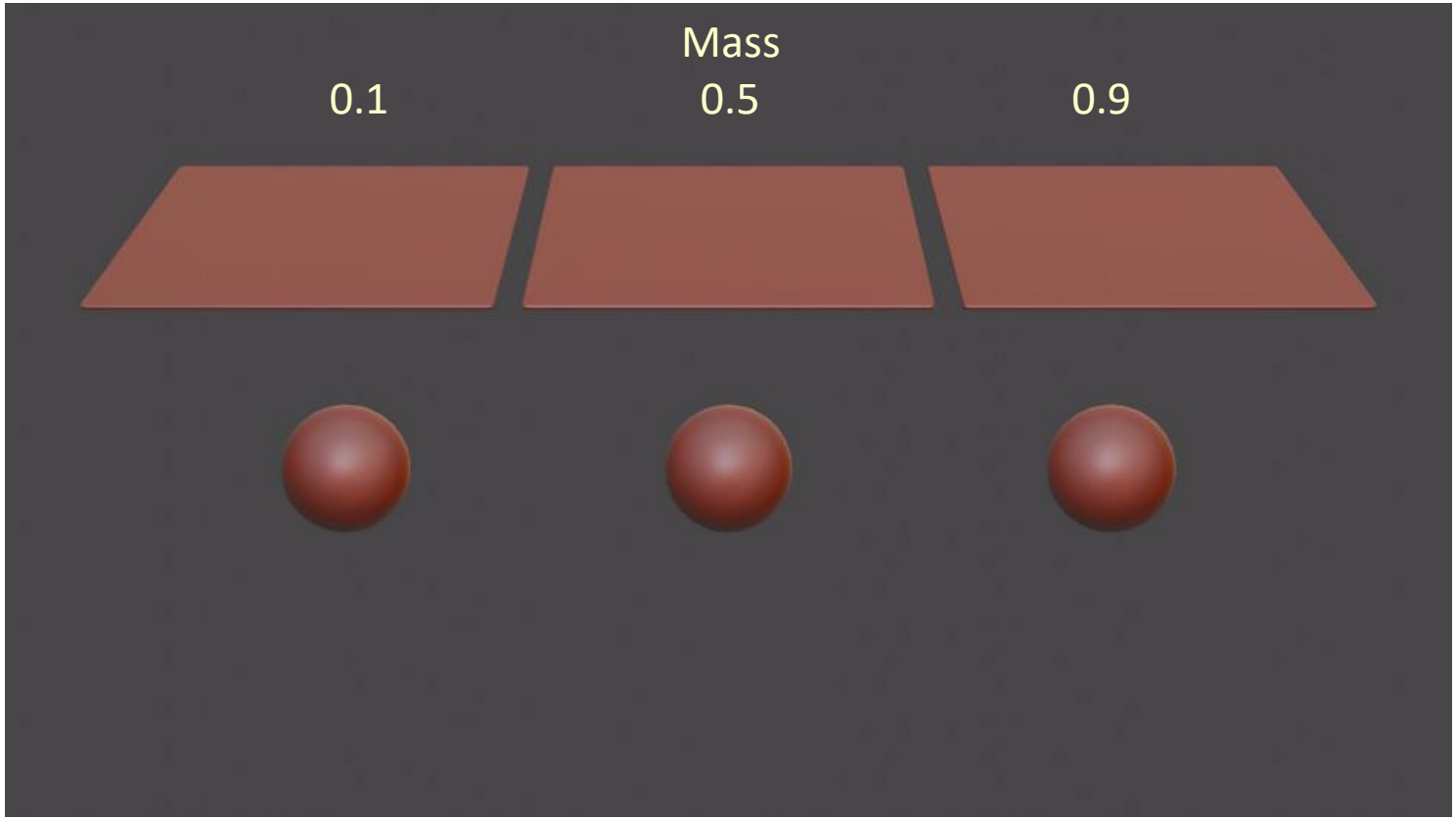


fabric materials.

	Default	Cotton	Leather	Rubber	Denim	Silk
Mass	0.3kg	0.300	0.4	3.000	1.000	0.150
Tension	15.000	15	80	15	40	5
Shear	5.000	5	25	25	25	0.0
Bending	0.500	0.5	150	25	10	0.05

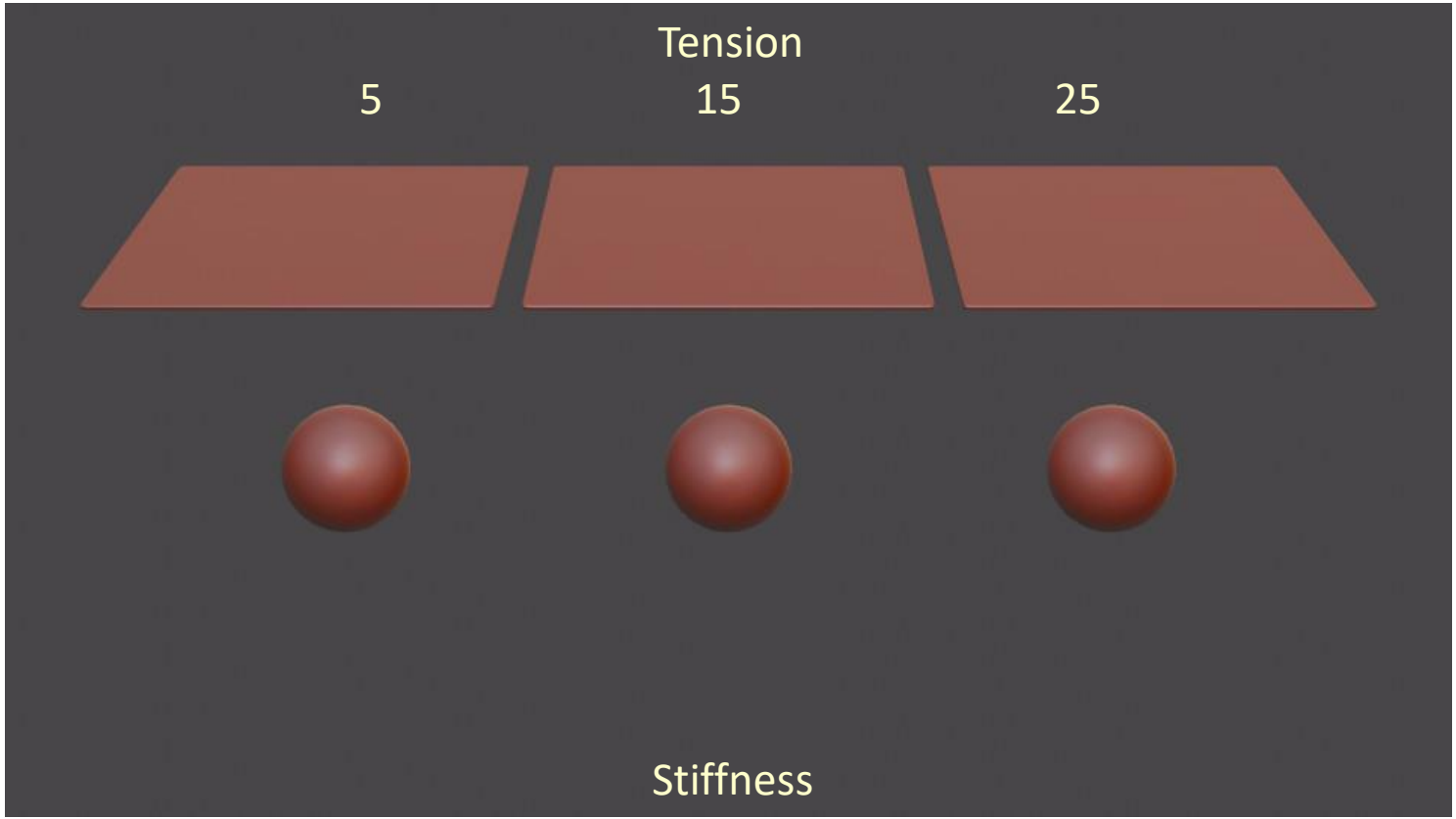
Blender/Python API

Cloth Simulation



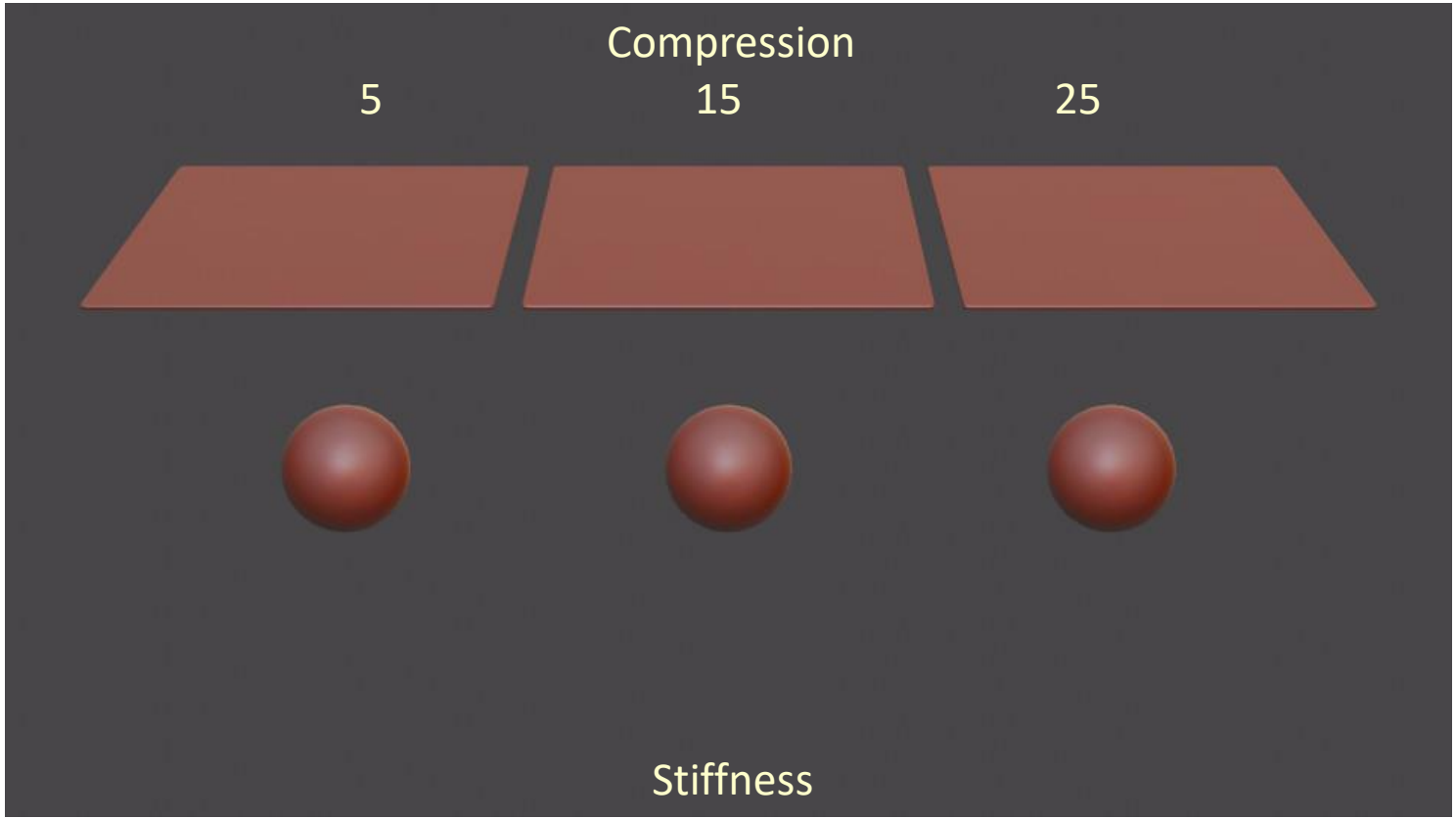
Blender/Python API

Cloth Simulation



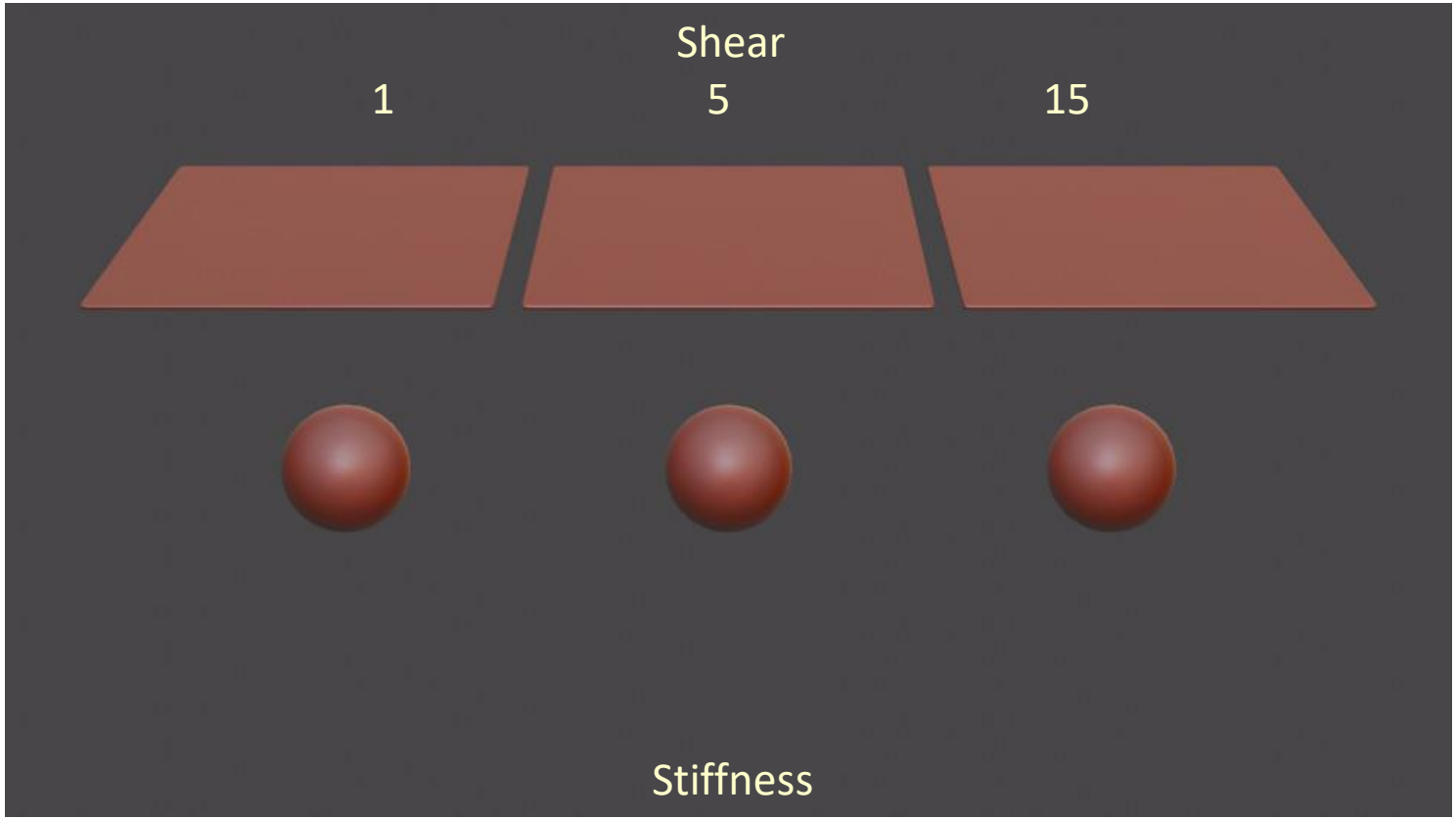
Blender/Python API

Cloth Simulation



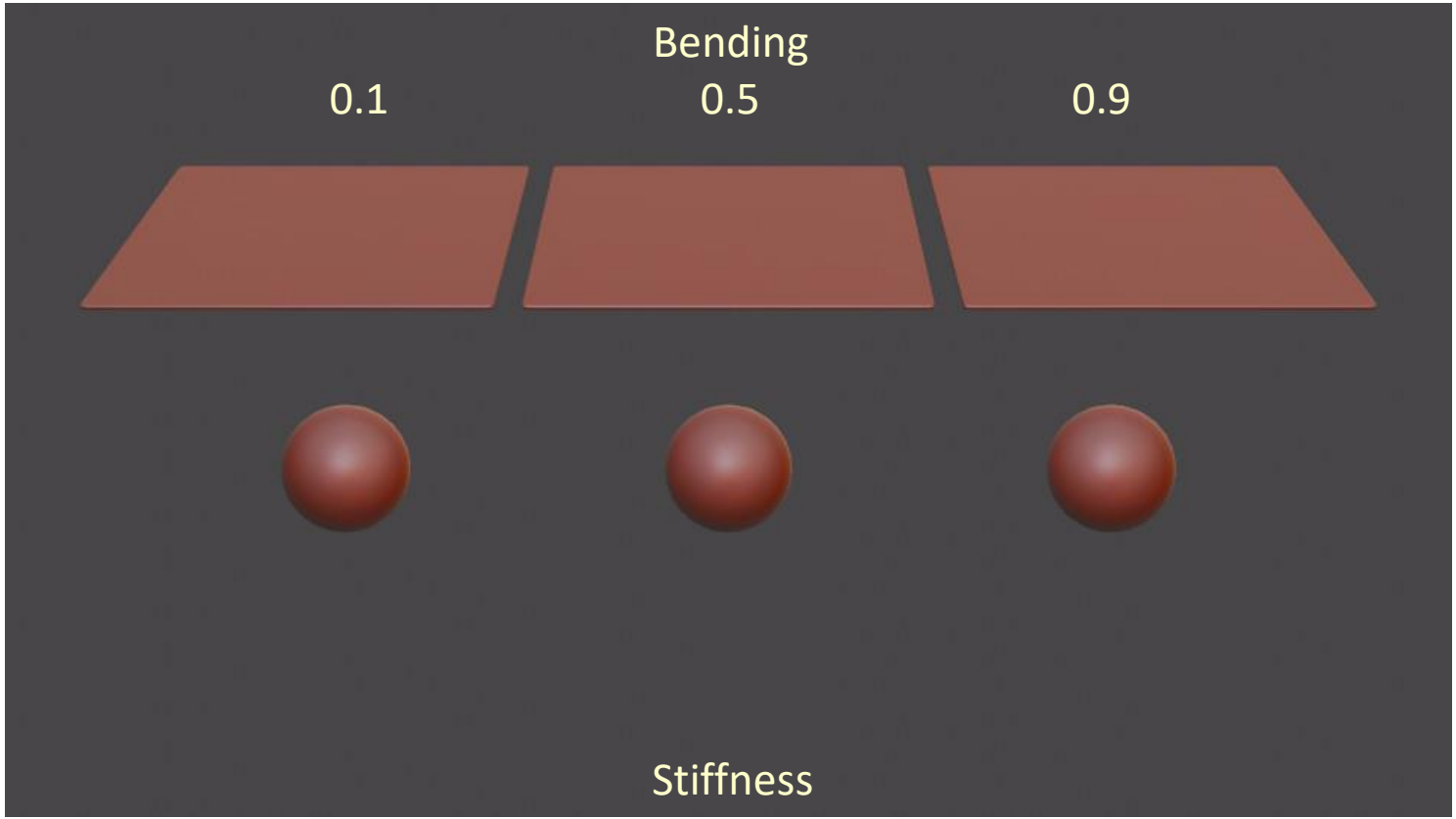
Blender/Python API

Cloth Simulation



Blender/Python API

Cloth Simulation



Cloth Animation

- Minimize Strain Energy
- Elasticity-based forces

$$\mathbf{F}_{net} = \left\{ \frac{\partial E(S)}{\partial x}, \frac{\partial E(S)}{\partial y}, \frac{\partial E(S)}{\partial z} \right\}$$

$$\frac{\partial E(x)}{\partial x} = \frac{E(x + \Delta x) - E(x)}{\Delta x}$$

- Generally this derivative must be computed analytically. Suppose we attempted to compute the derivative numerically; we consider the state variable constant, reducing our energy E(s) to E(x). Evaluating the energy E(S) takes a long time; we must iterate over all the vertices, faces, and edges, summing the energy of each one.

Cloth Animation

- Define overall motion of the system
- Given a state vector at a given time representing all relevant physical quantities (position, velocity) return the change in these variables w. r. t. time
- In our case we have simple Newtonian equations:

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = \frac{F}{m}$$

Cloth Animation

The cyan nodes are vertices, and the blue and pink lines are springs. The diagonal springs are necessary to resist collapse of the face; it ensures that the entire cloth does not decompose into a straight line.

Equations of State: Force

$$F_{net}(v) = Mg + F_{wind} + F_{air\ resistance} - \sum_{Springs \in v} k(x_{current} - x_{rest}) = Ma$$

M = mass of vertex

g = gravity vector = $(0, -9.8, 0)$

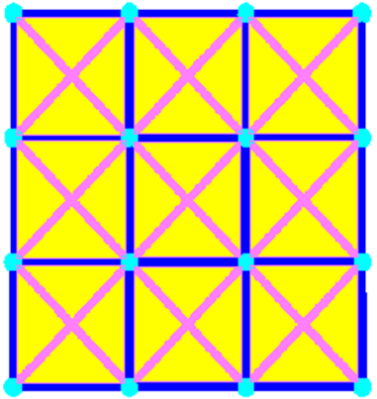
k = spring constant

$x_{current}$ = current length of spring

x_{rest} = rest (initial) length of spring

F_{wind} = wind vector

$F_{air\ resistance} = -a * velocity(v)^2$



Cloth Animation

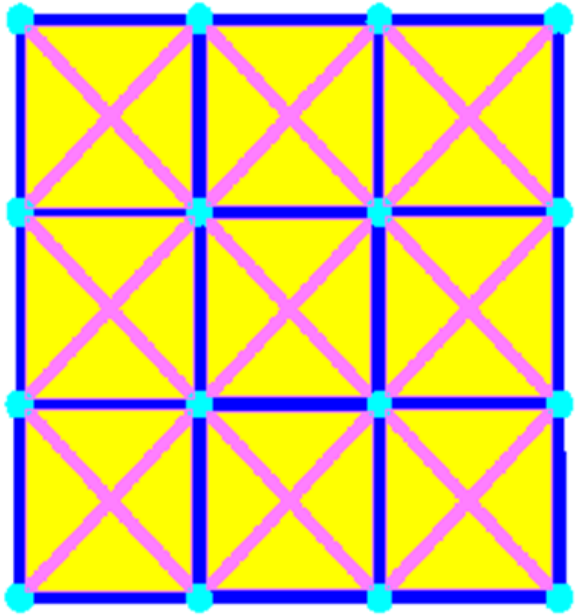
To determine M, a simple constant (assume is 1) is fine for all vertices. To be more accurate, you should compute the area of each triangle, and assign 1/3rd of it towards the mass of each incident vertex; this way the mass of the entire cloth is the total area of all the triangles times the mass density of the cloth. The gravity vector can also be an arbitrary vector; if all distance units were meters, time was measured in seconds, and we were on the surface of the earth and "y" was the "up/down" vector, (**0, -9.8, 0**) would be the correct "g". X(current) is just the current length of the spring, and X(rest), the spring's rest length, needs to be stored in each spring structure. F(wind) can just be some globally varying constant function, say (***sin(x*y*t), cos(z*t), sin(cos(5*x*y*z)***). a is a simple constant determined by the properties of the surrounding fluid (usually air,) but it can also be used to achieve certain cloth effects, and can help with the numeric stability of the cloth. k is a very important constant; if too low, the cloth will sag unrealistically:

Cloth Animation

Damping Springs: Springs resist relative, not absolute, changes in velocity

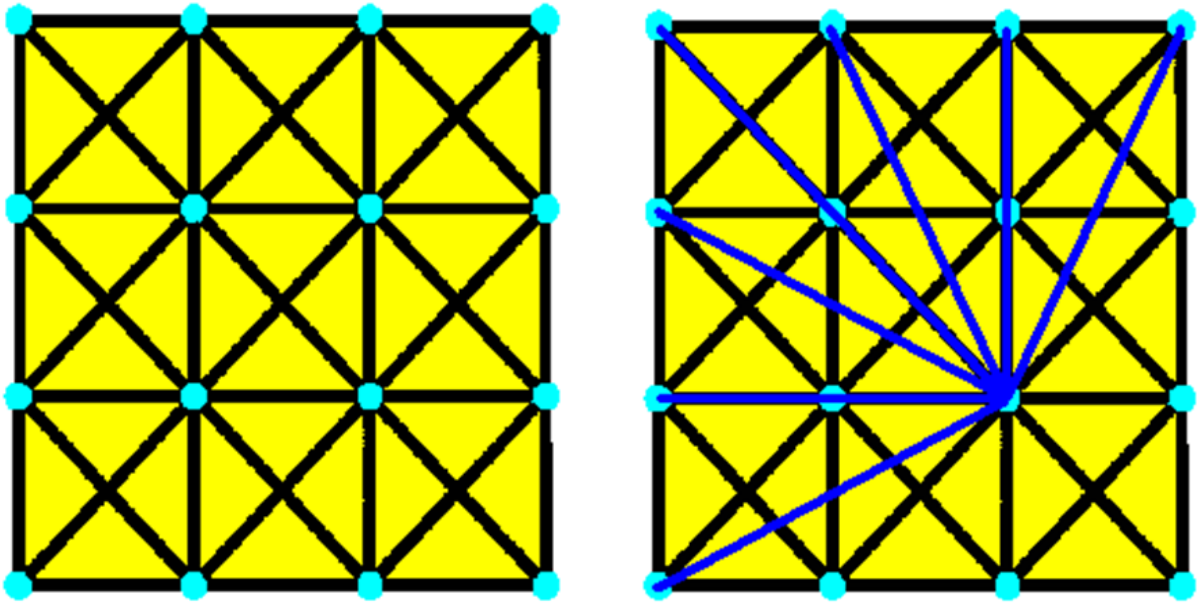
$F_{damp} = k_{damp}(\text{velocity}(v1) - \text{velocity}(v2))$

- Diagonal springs
  resist changes in shear
- Horizontal / Vertical
  springs resist compression

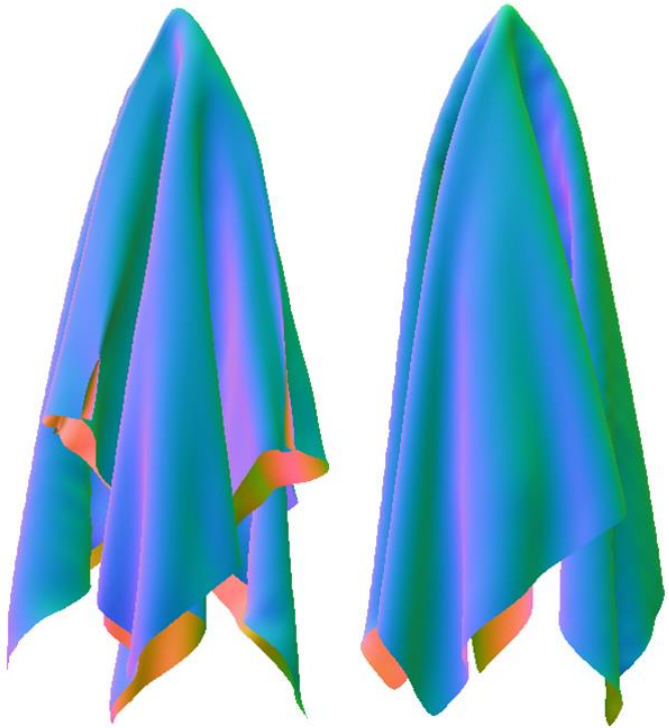


Cloth Animation

Bending forces: cloth resists high curvature
This can simulated well with bending springs

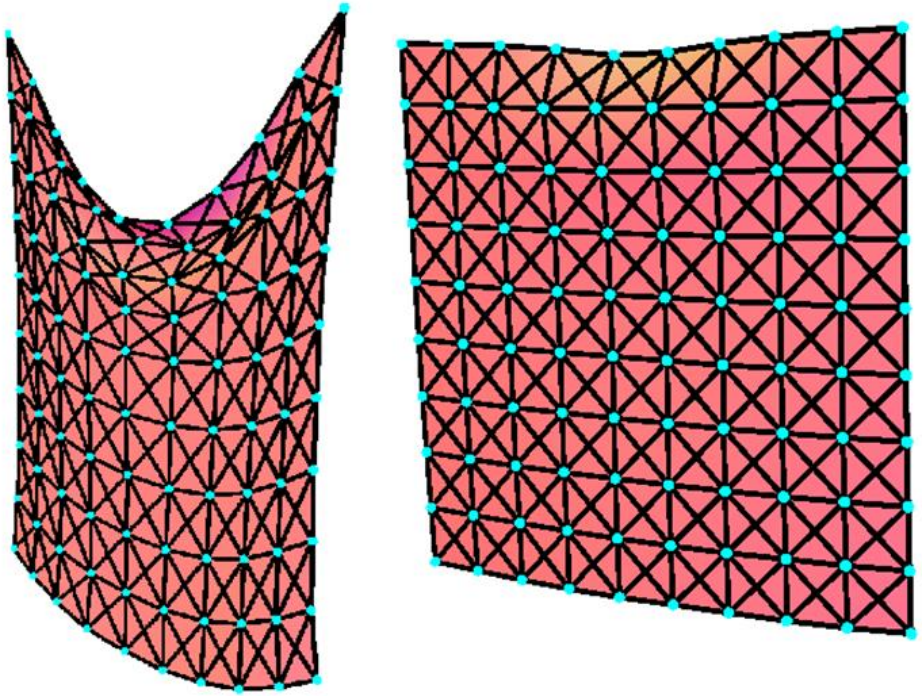


Cloth Animation



No bending springs Bending springs

Cloth Animation



Low **k** – sagging

High **k** - stiff

Cloth Animation

Integrating Equations of State
Explicit vs. Implicit vs. Symplectic

Euler's Method (1st order) [**Explicit**]

$$v_{t+\Delta t} = v_t + \Delta t \left(\frac{dv}{dt}\right)_t$$

$$x_{t+\Delta t} = x_t + \Delta t\, v_t$$

Runge Kutta (4th order) [**Explicit**]

Verlet Algorithm [**Explicit**]

$$x_{t+\Delta t} = 2\, x_t - x_{t-\Delta t} + \left(\frac{dv}{dt}\right)_t (\Delta t)^2$$

- **Implicit integrators** are stable but slow and tedious to implement
- *Symplectic integrators are fast but hard to generalize
- **Explicit integrators** are easy to implement but unstable

**Symplectic is an infrequently used mathematical term that describes objects joined together smoothly.*

Explicit and Implicit Methods

Explicit and implicit methods are approaches used in numerical analysis for obtaining numerical approximations to the solutions of time-dependent ordinary and partial differential equations, as is required in computer simulations of physical processes.

The explicit method calculates the system status at a future time from the currently known system status.

The implicit method calculates the system status at a future time from the system statuses at present and future times.

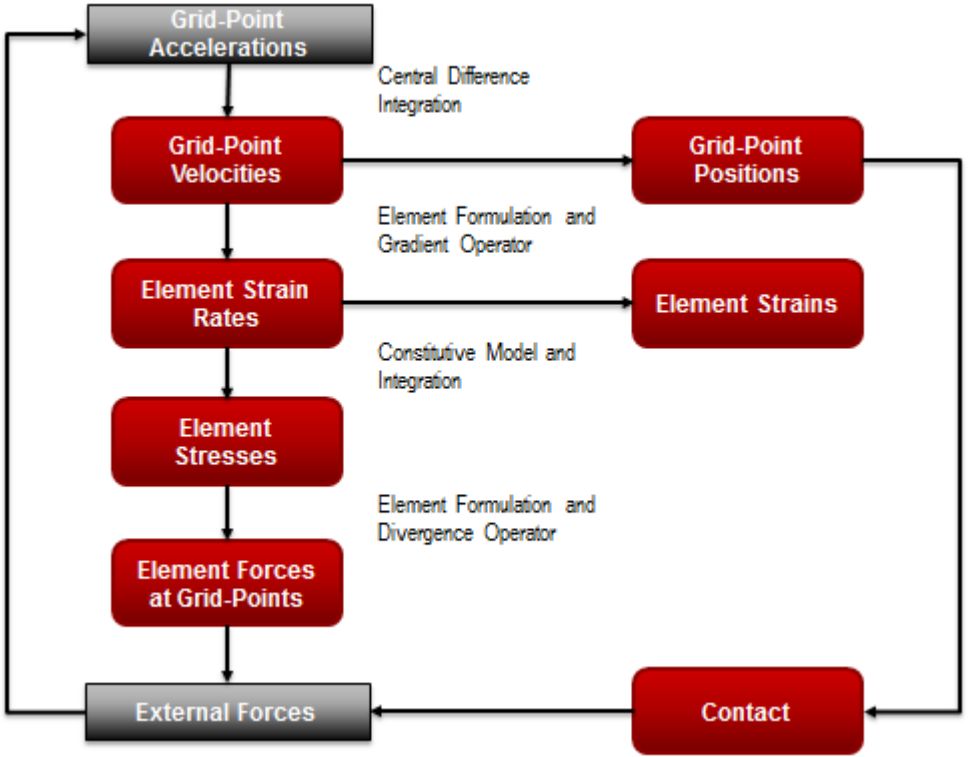
Explicit and Implicit Methods

The explicit method is easier to program and can be calculated within a shorter time.

But its stability is so low that you need to use a step size small enough to prevent divergence. On the contrary, the implicit method has high stability and converges if you set proper parameters. But, as you need to solve an equation at every step, it takes a long time to calculate.

As the implicit method can use a sufficiently large step size, it is suitable for solving equations that involve a long time. Also, in non-linear equations such as contact, **it is difficult to predict a future from the past state**. So, in these cases, it is recommended that you use the implicit method rather than the explicit method.

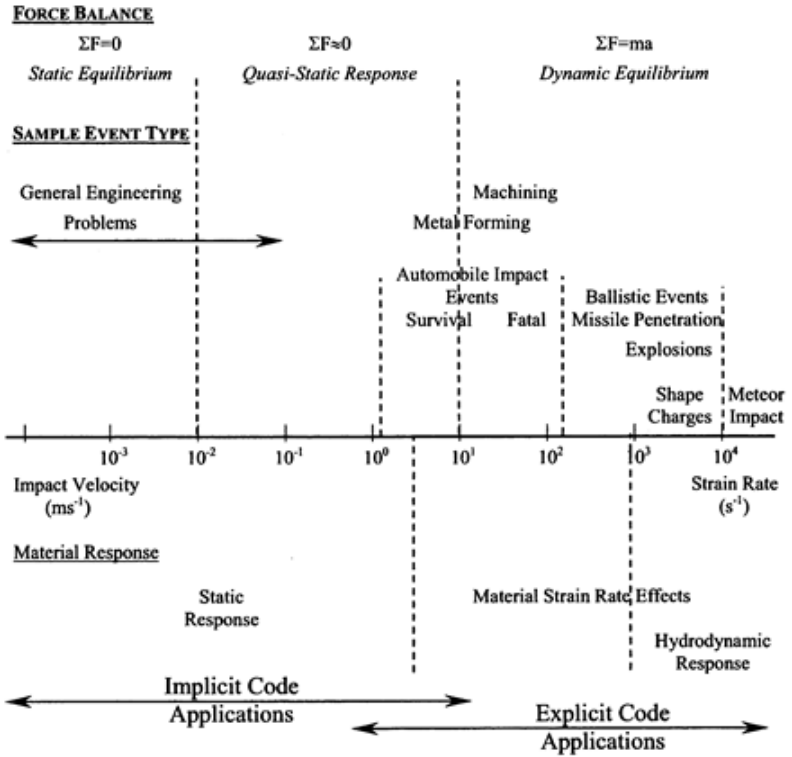
Explicit and Implicit Methods



Choosing between Implicit and Explicit Methods

Implicit is generally most efficient in solving for static and quasi-static equilibrium, therefore long duration nonlinear events would be suitable.

Explicit is more appropriate for high speed events, because the time step constrained by the event itself and the assumption of lumped mass. The use of reduced integration elements also mean that each step is considerably faster than implicit. A benefit of the small time step approach is that extreme non-linearities can be handled by virtue of the relatively small change in state between each time step.



Cloth Animation

Verlet Algorithm (Explicit)

- The Verlet integration algorithm is such an explicit model with the very interesting property that it does not need to know anything about the velocity; it computes this internally via looking at the position at both the current and previous time step.

- Another wonderful aspect of this algorithm is that like 4^{th} order Runge-Kutta, it is 4th order accurate. Because it is quite accurate, easy to implement, and does not need the velocity terms, it is usually favorite explicit model used in all cloth models.

Cloth Animation

- Verlet integration is frequently used to calculate trajectories of particles in molecular dynamics simulations and computer graphics. The algorithm was first used in **1791 by Delambre** and has been rediscovered many times since then, most recently by Loup Verlet in the 1960s for use in molecular dynamics. It was also used by Cowell and Crommelin in 1909 to compute the orbit of Halley's Comet.
- Where Euler's method uses the forward difference approximation to the first derivative in differential equations of order one, Verlet integration can be seen as using the central difference approximation to the second derivative:

$$\frac{\Delta^2 \vec{x}_n}{\Delta t^2} = \frac{\frac{\vec{x}_{n+1} - \vec{x}_n}{\Delta t} - \frac{\vec{x}_n - \vec{x}_{n-1}}{\Delta t}}{\Delta t} = \frac{\vec{x}_{n+1} - 2\vec{x}_n + \vec{x}_{n-1}}{\Delta t^2}$$

$$\vec{a}_n \Delta t^2 = \vec{x}_{n+1} - 2\vec{x}_n + \vec{x}_{n-1} \qquad \vec{x}_{n+1} = 2\vec{x}_n - \vec{x}_{n-1} + \vec{a}_n \Delta t^2$$

Euler integration

The heart of the simulation is a particle system. Typically, in implementations of particle systems, each particle has two main variables: Its position $\boldsymbol{r}(t_0)$ and its velocity $\boldsymbol{v}(t_0)$. Then in the time-stepping loop, the new position $\boldsymbol{r}(t_0 + \Delta t)$ and velocity $\boldsymbol{v}(t_0 + \Delta t)$ are often computed by applying the rules

$$\boldsymbol{a}(t_0) = \frac{F}{m}$$

$$\boldsymbol{v}(t_0 + \Delta t) \approx \boldsymbol{v}(t_0) + \boldsymbol{a}(t_0, \boldsymbol{r}(t_0), \boldsymbol{v}(t_0))\Delta \boldsymbol{t}$$

$$\boldsymbol{r}(t_0 + \Delta t) \approx \boldsymbol{r}(t_0) + \boldsymbol{v}(t_0 + \Delta t)\Delta t$$

where $\boldsymbol{\Delta t}$ is the time step, and $\boldsymbol{a}$ is the acceleration computed using Newton's second law.

Verlet integration

$$v(t_0 + \Delta t) \approx v(t_0) + a(t_0, r(t_0), v(t_0))\Delta t$$

$$r(t_0 + \Delta t) \approx r(t_0) + v(t_0 + \Delta t)\Delta t$$

Euler integration

$$r(t_0 + \Delta t) \approx r(t_0) + v(t_0) + a(t_0)\Delta t\,\Delta t$$

$$r(t_0 + \Delta t) \approx r(t_0) + r(t_0) - r(t_0 - \Delta t) + a(t_0)\Delta t\,\Delta t$$

$$r(t_0 + \Delta t) \approx 2r(t_0) - r(t_0 - \Delta t) + a(t_0)\Delta t^2$$

Verlet integration

```
verlet(x, x0, F, dt, m)
euler(x, v, F, dt, m)
```

Verlet integration

In Verlet integration, however, we choose a **velocity-less** representation and another integration scheme: Instead of storing each particle's position and velocity, we store its current position $\boldsymbol{r}(t_0)$ and its previous position $\boldsymbol{r}(t_0 - \Delta t)$. Keeping the time step fixed, the update rule (or integration step) is then

$$\boldsymbol{r}(t_0 + \Delta t) \approx \mathbf{2}.\boldsymbol{r}(t_0) - \boldsymbol{r}(t_0 - \Delta t) + a(t_0)\Delta t^2$$

$$\boldsymbol{r}(t_0 - \Delta t) \approx \boldsymbol{r}(t_0 + \Delta t)$$

It works due to the fact that

$$\mathbf{2}.\boldsymbol{r}(t_0) - \boldsymbol{r}(t_0 - \Delta t) = \boldsymbol{r}(t_0) + \big(\boldsymbol{r}(t_0) - \boldsymbol{r}(t_0 - \Delta t)\big)$$

is an approximation of the current velocity. (actually, it's the distance traveled last time step)

Verlet integration

It is not always very accurate (energy might leave the system, i.e., dissipate) but it's **fast** and ***stable***. By lowering the value **2** to something like **1.99** a small amount of drag can also be introduced to the system.

At the end of each step, for each particle the current position $\boldsymbol{r}(t_0)$ gets stored in the corresponding variable $\boldsymbol{r}(t_0 - \Delta t)$. Note that when manipulating many particles, a useful optimization is possible by simply swapping array pointers.

A Newtonian Particle

- Differential equation: f = ma
- Forces can depend on:
  Position x, Velocity v, Time t

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

The second order ODE

$$\frac{d^2x}{dt^2} = \frac{f\left(x, {}^{dx}/_{dt}, t\right)}{m}$$

The first order ODE

$$\frac{dv}{dt} = \frac{f(r, v, t)}{m}$$

To handle a second order ODE, we convert it to a first-order one by introducing extra variables. Here we create a variable ***v*** to represent velocity, giving us a pair of coupled first-order ODE's $\dot{v} = {}^{f}/_{m}$, $\dot{x} = v$

$$a = \frac{f(r, v, t)}{m}$$

A Newtonian Particle

The position and velocity **x** and **v** can be concatenated to form a 6-vector. This position/velocity product space is called ***phase space***. In components, the phase space equation of motion is;

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{v}_1 \\ \dot{v}_2 \\ \dot{v}_3 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ f_1/m \\ f_2/m \\ f_3/m \end{bmatrix}$$

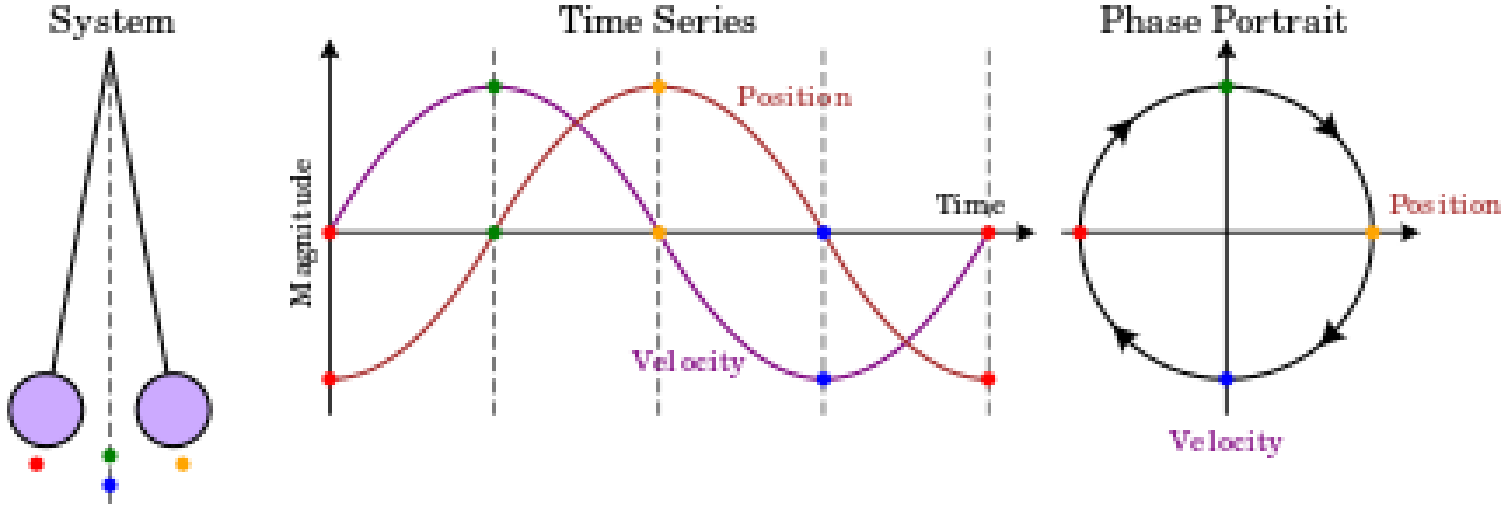
system of n particles is described by n copies of the equation, concatenated to form a 6n-long vector. Conceptually, the whole system may be regarded as a point moving through 6n-space.

the phase space

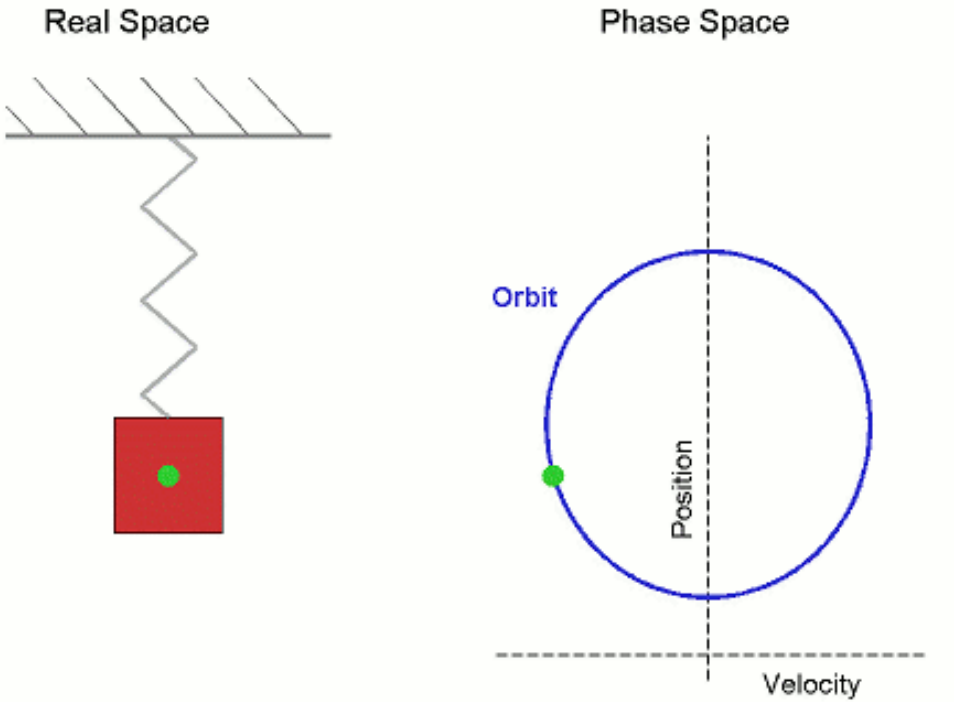
In newtonian mechanics, the phase space is the space of all possible states of a system; the state of a mechanical system is defined by the constituent positions **x** and velocity **v**. **x** and **v** together determine the future behavior of that system. In other words if you know **x** and **v** at time **t** you will be able to calculate the **x** and **v** at time **t+1.**

To describe the motion of a single particle you will need 6 variables, 3 positions and 3 velocities. You can imagine a 6 dimensional space; three positions and three velocities. Each point in this 6 dimensional space is a possible description of the particles' possible states, of course constraint by the laws of classical mechanics.

the phase space

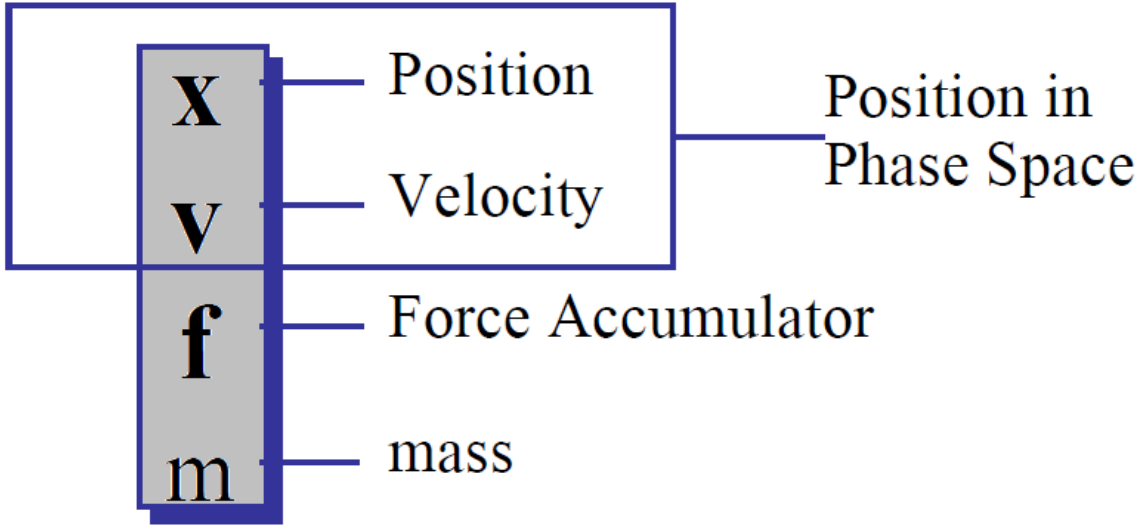


the phase space



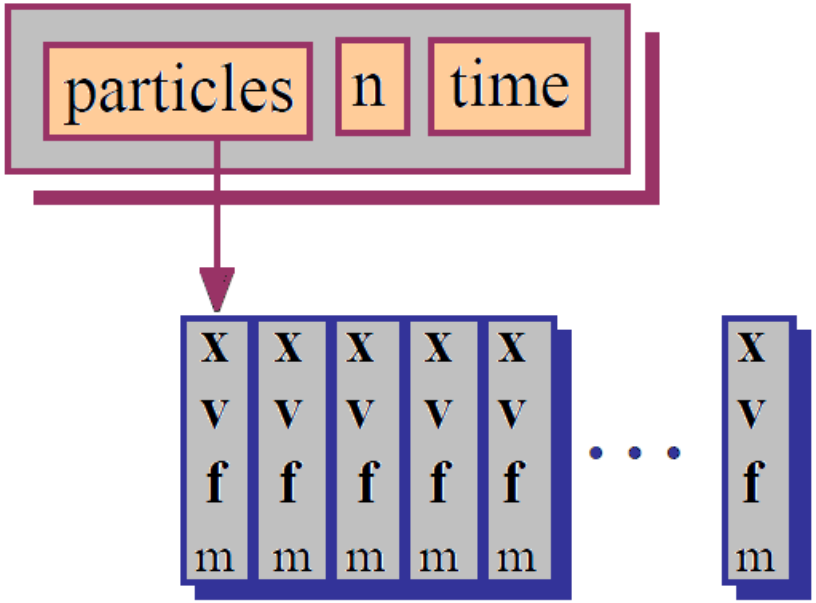
Particle Structure

A particle may be represented by a structure containing its **position**, **velocity**, **force**, and **mass**. The six-vector formed by concatenating the position and velocity comprises the point's position in phase space.



Particle Systems

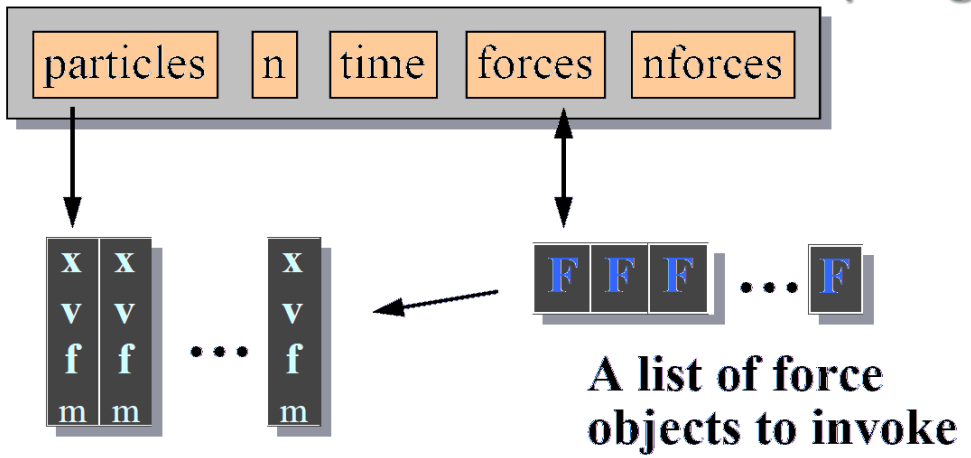
A particle system is essentially just a [list of particles].

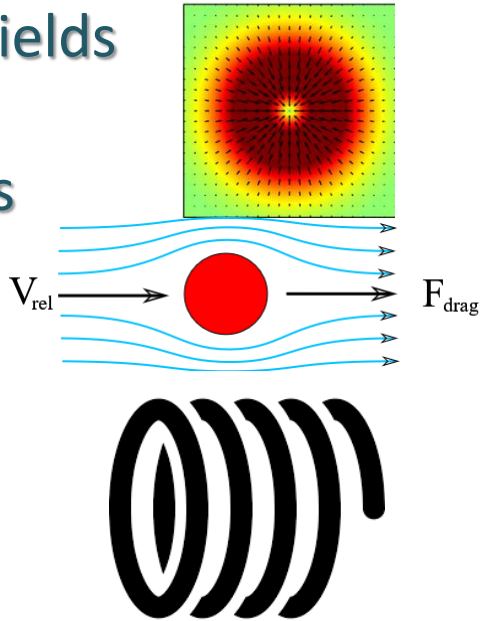


```
class Particle(object):
    """
    Stores position, previous position, and where it is in the grid.
    """
    def __init__(self, screen, currentPos, gridIndex):
        # Current Position : m_x
        self.currentPos = Vec2d(currentPos)
        # Index [x][y] of Where it lives in the grid
        self.gridIndex = gridIndex
        # Previous Position : m_oldx
        self.oldPos = Vec2d(currentPos)
        # Force accumulators : m_a
        self.forces = GRAVITY
        # Should the particle be locked at its current position?
        self.locked = False
        self.followMouse = False
```

Forces

- **Constant** — gravity
- **Position/time dependent** — force fields
- **Velocity-Dependent** — drag
- **n-ary** — springs





Forces

All particles are essentially alike. In contrast, the objects that give rise to forces are heterogeneous. As a matter of implementation, we would like to make it easy to extend the set of force-producing objects without modifying the basic particle system model.

Forces can be grouped into three broad categories:

- **Unary forces**, such as gravity and drag, that act independently on **each particle**, either exerting a constant force, or one that depends on one or more of particle position, particle velocity, and time.
- **n-ary forces**, such as springs, that apply forces to a fixed set of particles.
- Forces of **spatial interaction**, such as attraction and repulsion, that may act on any or all pairs of particles, depending on their positions.

Unary forces: Gravity

Global earth gravity is trivial to implement. The gravitational force on each particle is $\mathbf{f} = m\mathbf{g}$, where $\mathbf{g}$ is a constant vector (presumably pointing down) whose magnitude is the gravitational constant. If all particles are to feel the same gravity, which they need not in a simulation, then gravitational force is applied simply by traversing the system's particle list, and adding the appropriate force into each particles force accumulator. Gravity is basic enough that it could reasonably be wired it into the particle system, rather than maintaining a distinct "gravity object".

Unary forces: Viscous Drag

Ideal viscous drag has the form $\mathbf{f} = -\mathbf{k_d}\mathbf{v}$, where the constant $\mathbf{k_d}$ is called the coefficient of drag. The effect of drag is to resist motion, making a particle gradually come to rest in the absence of other influences. It is highly recommended that at least a small amount of drag be applied to each particle, if only to enhance numerical stability. Excessive drag, however, makes it appear that the particles are floating in molasses. Like gravity, drag can be implemented as a wired-in special case.

n-ary forces

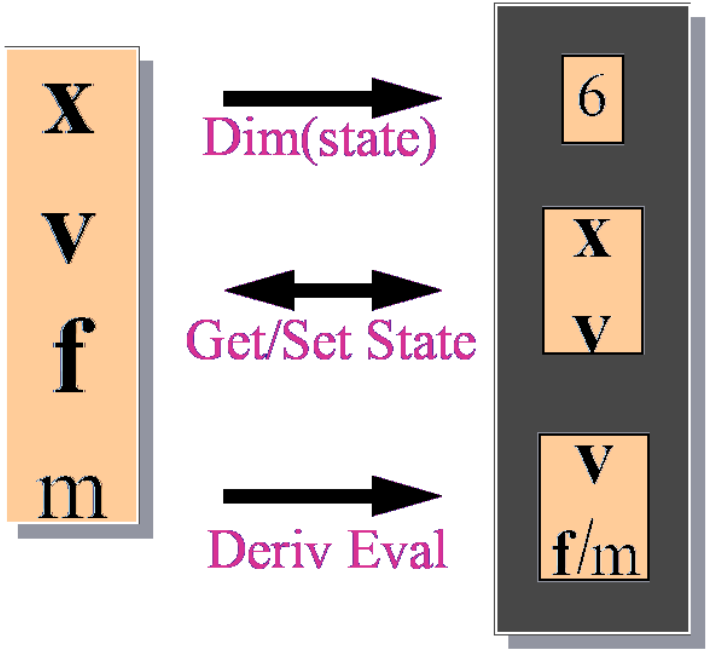
An example of a binary force is a Hook's law spring. In a basic **spring-damper** simulation, the springs are the structural elements that hold everything together. The spring forces between a pair of particles at positions a and b are

$$f_a = -[k_s(|r_a - r_b| - l) + k_d(v_a - v_b)^2] \quad f_b = -f_a$$

the spring force magnitude is proportional to the difference between the actual length and the rest length, while the damping force magnitude is proportional to a and b's relative speed.

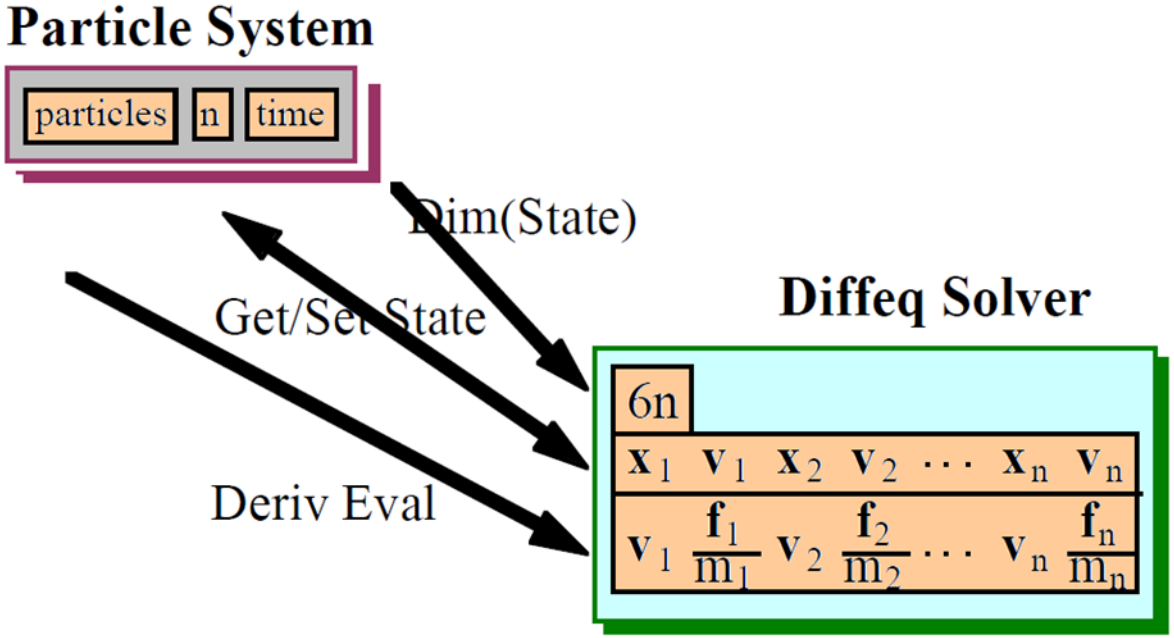
Solver Interface

The relation between a particle system and a differential equation solver.



Solver Interface

The relation between a particle system and a differential equation solver.



PHYSICS in COMPUTER ANIMATIONS and GAMES

```
class ParticleSystem(Grid):
    """
    Implements the verlet particles physics on the encapsulated Grid object.
    """
    def __init__(self, screen, rows=16, columns=16, step=PSTEP, offset=OFFSET):
        super(ParticleSystem, self).__init__(screen, rows, columns, step, offset)

    def verlet(self):
        # Verlet integration step:
        for p in self:
            if not p.locked:
                # make a copy of our current position
                temp = Vec2d(p.currentPos)
                p.currentPos += p.currentPos - p.oldPos + p.forces * TSTEP**2
                p.oldPos = temp
            elif p.followMouse:
                temp = Vec2d(p.currentPos)
                p.currentPos = Vec2d(pygame.mouse.get_pos())
                p.oldPos = temp
```

Derivation Evaluation Loop

- Clear forces

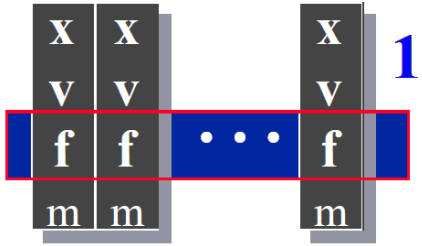
  Loop over particles, zero force accumulators.

- Calculate forces

  Sum all forces into accumulators.

- Gather

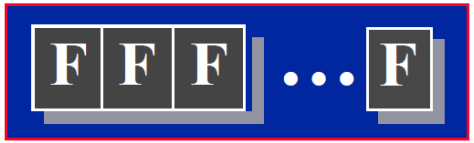
  Loop over particles, copying v and $\mathrm{f/m}$ into destination array.



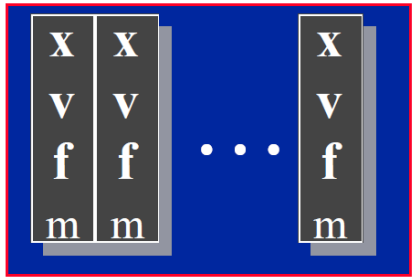
1

Clear Force Accumulators

2



Invoke `accumulateForces` functions

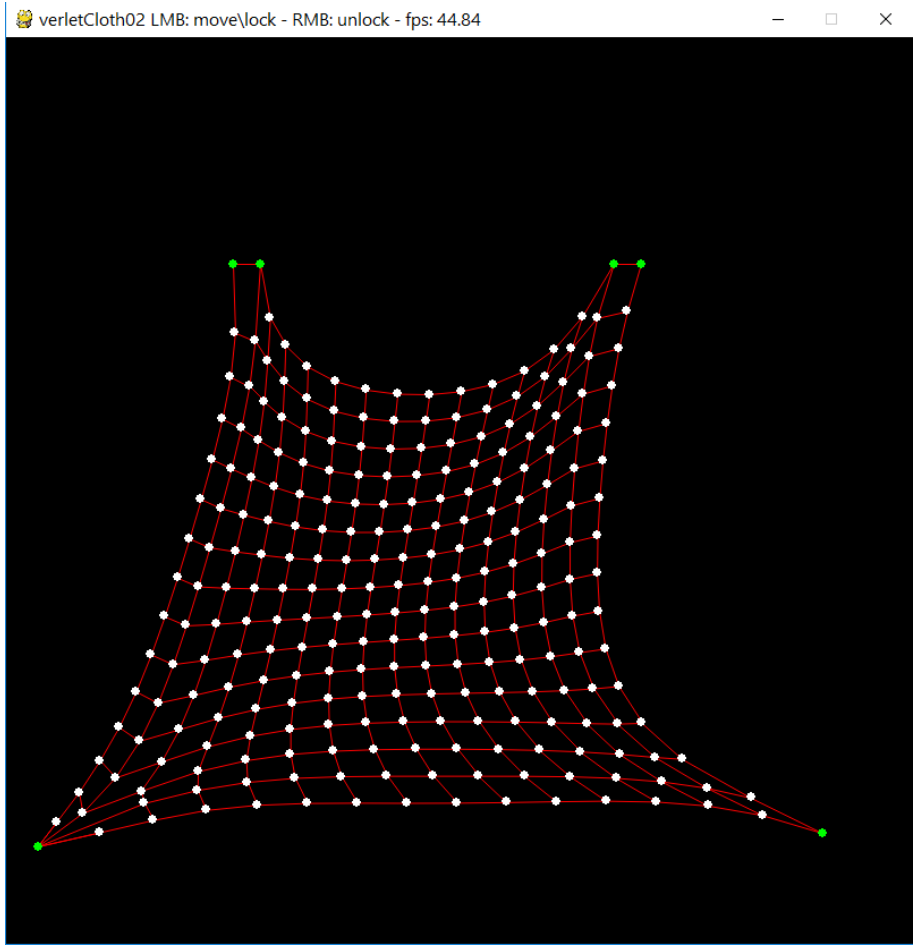


3

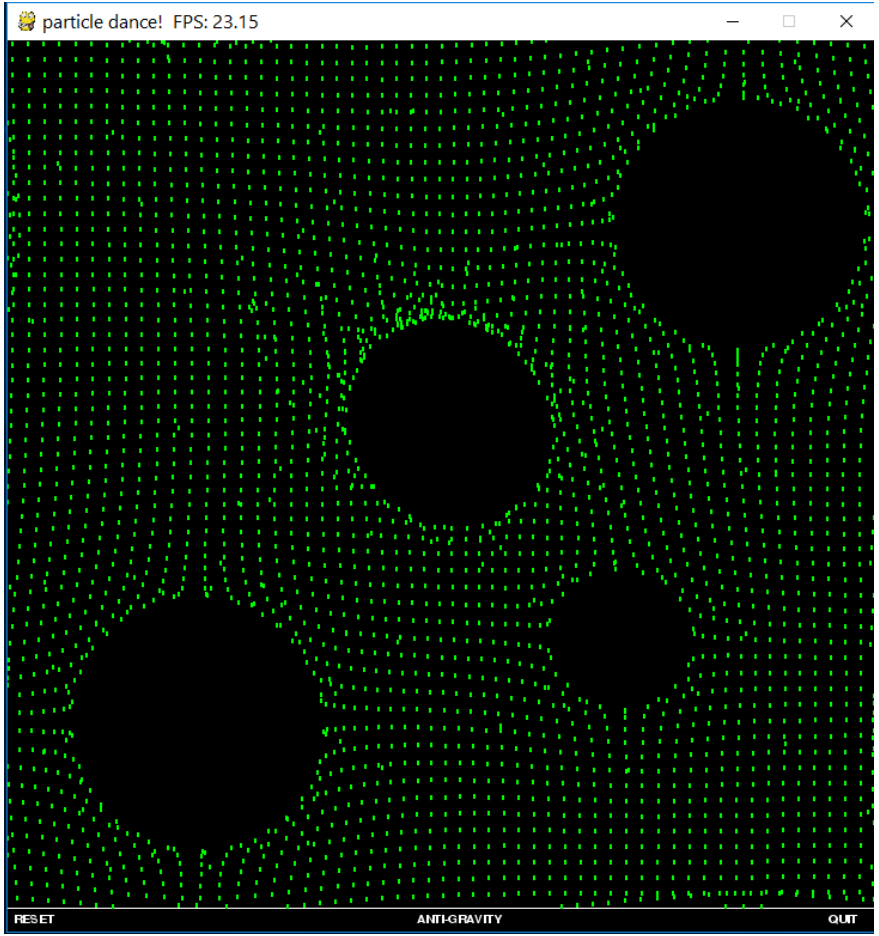
Return [v, f/m,…] to solver.

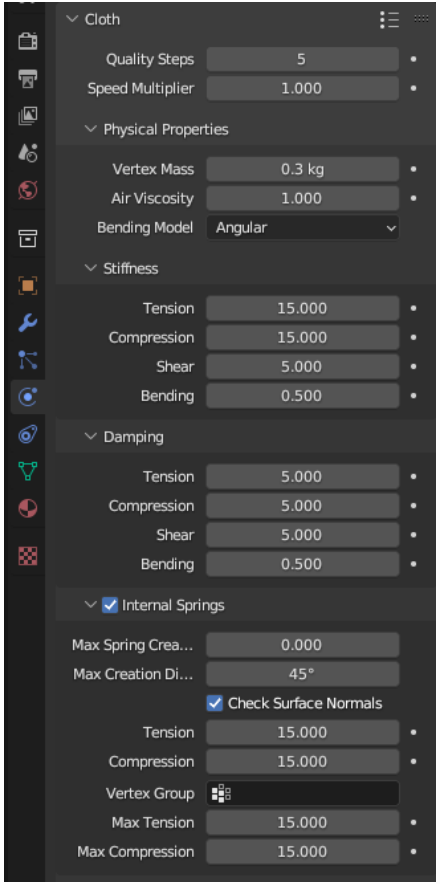
PHYSICS in COMPUTER ANIMATIONS and GAMES

```
class ParticleSystem(Grid):
    """
    Implements the verlet particles physics on the encapsulated Grid object.
    """
    def __init__(self, screen, rows=16, columns=16, step=PSTEP, offset=OFFSET):
        super(ParticleSystem, self).__init__(screen, rows, columns, step, offset)

    def accumulateForces(self):
        # This doesn't do much right now, other than constantly reset the
        # particles 'forces' to be 'gravity'.  But this is where you'd implement
        # other things, like drag, wind, etc.
        for p in self:
            p.forces = GRAVITY

    def timeStep(self):
        # This executes the whole shebang:
        self.accumulateForces()
        self.verlet()
        for i in range(ITERATE):
            self.satisfyConstraints()
```


verletCloth02 LMB: move\lock - RMB: unlock - fps: 44.84

PHYSICS in COMPUTER ANIMATIONS and GAMES





Integration

Euler

Also known as “Forward Euler”. Simplest integrator. Very fast but also with less exact results. If no dampening is used, particles get more and more energy over time. For example, bouncing particles will bounce higher and higher each time. Should not be confused with “Backward Euler” (not implemented) which has the opposite feature, the energy decrease over time, even with no dampening. Use this integrator for short simulations or simulations with a lot of dampening where speedy calculations are more important than accuracy.

Verlet

Very fast and stable integrator, energy is conserved over time with very little numerical dissipation.

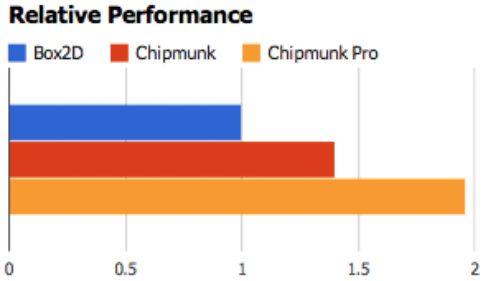
Midpoint

Also known as “2nd order Runge-Kutta”. Slower than Euler but much more stable. If the acceleration is constant (no drag for example), it is energy conservative. It should be noted that in example of the bouncing particles, the particles might bounce higher than they started once in a while, but this is not a trend. This integrator is a generally good integrator for use in most cases.

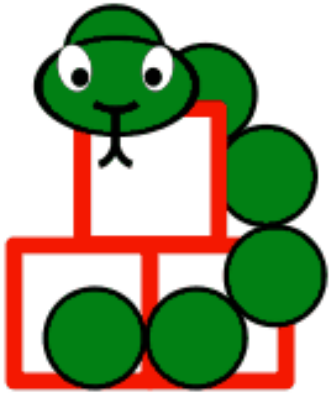
RK4

Short for “4th order Runge-Kutta”. Similar to Midpoint but slower and in most cases more accurate. It is energy conservative even if the acceleration is not constant. Only needed in complex simulations where Midpoint is found not to be accurate enough.

Chipmunk Game Dynamics



- Designed specifically for 2D video games.
- Circle, convex polygon, and beveled line segment collision primitives.
- Multiple collision primitives can be attached to a single rigid body.
- Fast broad phase collision detection by using a bounding box tree with great temporal coherence or a spatial hash.
- Extremely fast impulse solving by utilizing Erin Catto's contact persistence algorithm.
- Supports sleeping objects that have come to rest to reduce the CPU load.
- Support for collision event callbacks based on user definable object types types.
- Flexible collision filtering system with layers, exclusion groups and callbacks.
- Supports nearest point, segment (raycasting), shape and bounding box queries to the collision detection system.
- Collision impulses amounts can be retrieved for gameplay effects, sound effects, etc.
- Large variety of joints – easily make vehicles, ragdolls, and more.
- Joint callbacks.
  Can be used to easily implement breakable or animated joints.
- Maintains a contact graph of all colliding objects.
- Lightweight C99 implementation with no external dependencies outside of the Std. C library



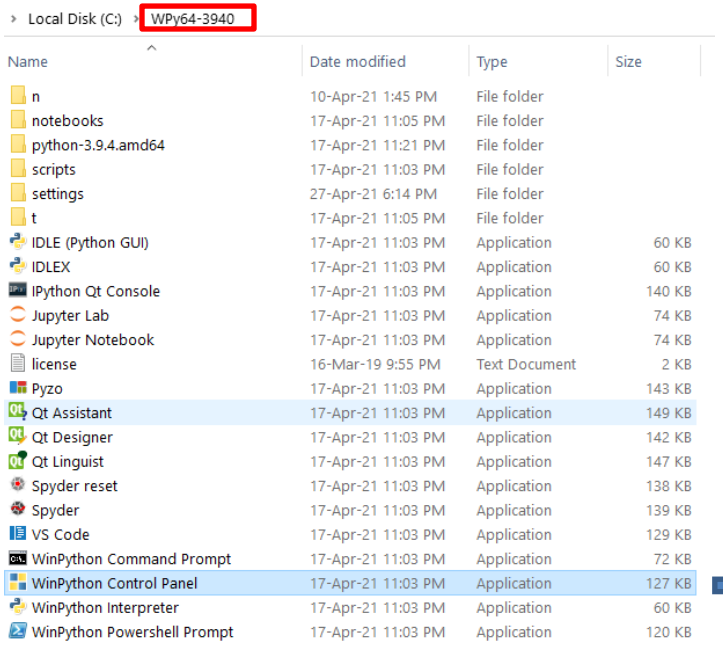
pymunk

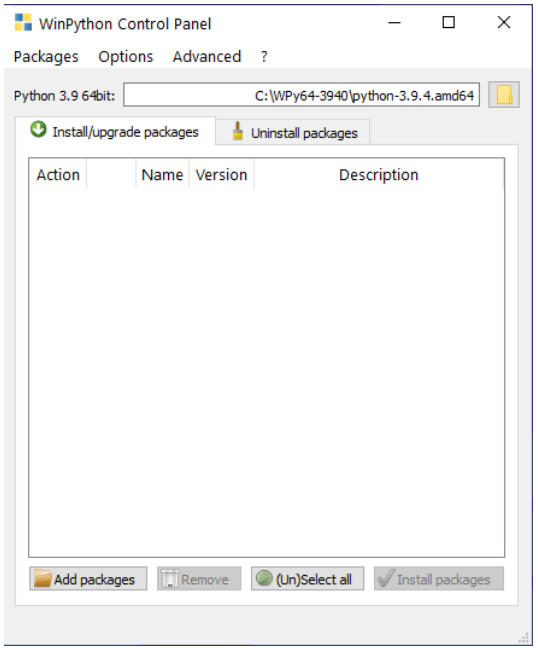
Pymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python. Perfect when you need 2d physics in your game, demo or other application! It is built on top of the very capable 2d physics library ***Chipmunk***.

In the normal case pymunk can be installed with pip:

```
> pip install pymunk
```

Pymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python. Perfect when you need 2d physics in your game, demo or other application! It is built on top of the very capable 2d physics library Chipmunk.





Pymunk is a easy-to-use pythonic 2d physics library that can be used whenever you need 2d rigid body physics from Python. Perfect when you need 2d physics in your game, demo or other application! It is built on top of the very capable 2d physics library ***Chipmunk***.

In the normal case pymunk can be installed with pip:

```
> pip install pymunk
```

pymunk 6.7.0

Latest version

```
pip install pymunk
```

Released: May 1, 2024

Pymunk is a easy-to-use pythonic 2D physics library

Navigation

- Project description
- Release history
- Download files

Verified details

These details have been verified by PyPI

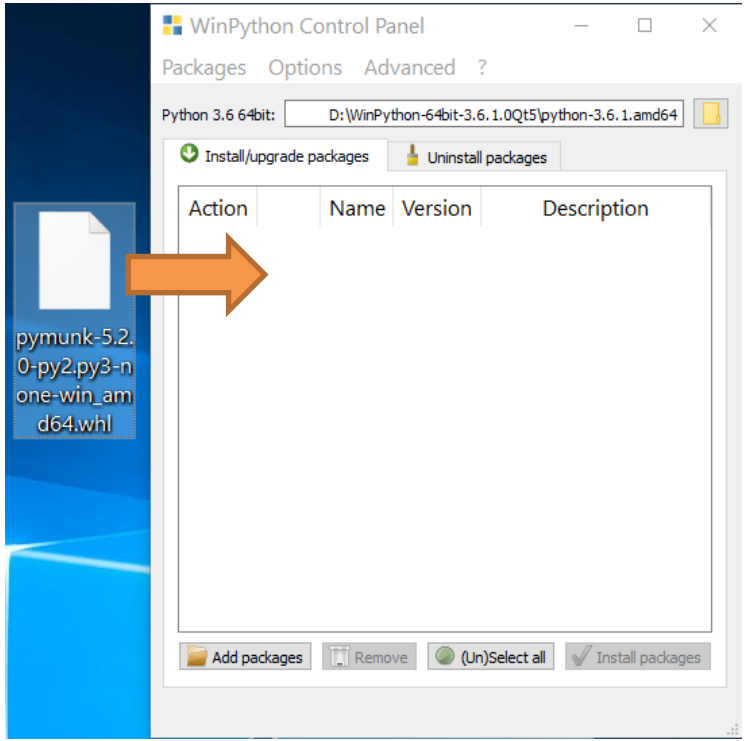
Maintainers

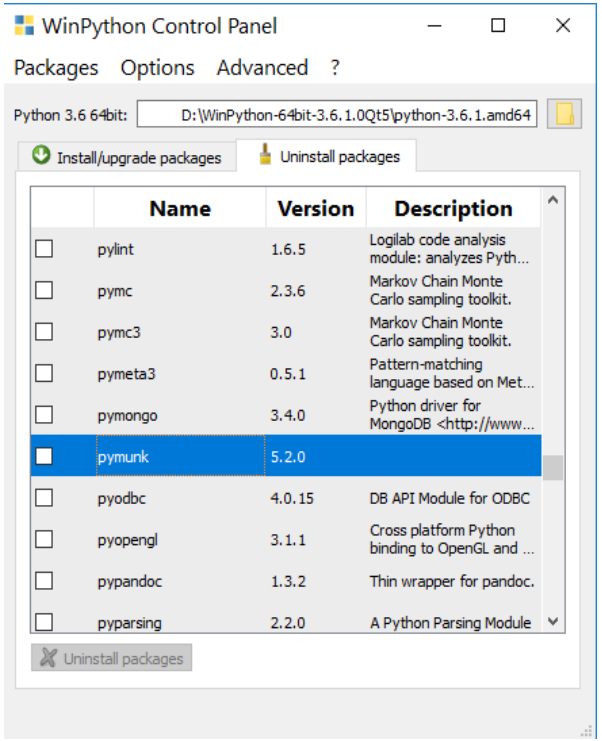
viblo

Project description

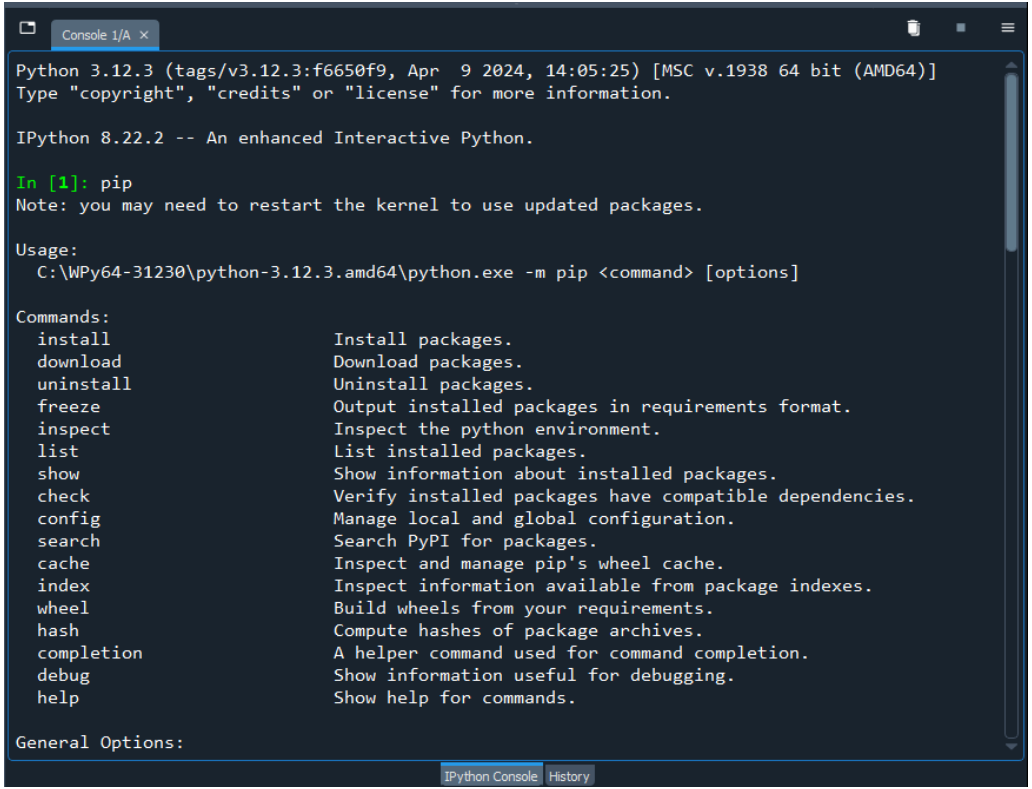
Pymunk is an easy-to-use pythonic 2D physics library that can be used whenever you need 2D rigid body

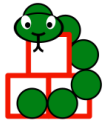
pymunk


WinPython Control Panel
Packages Options Advanced ?
Python 3.6 64bit: D:\WinPython-64bit-3.6.1.0Qt5\python-3.6.1.amd64
Install/upgrade packages
Uninstall packages
Action
Name
Version
Description
pymunk-5.2.0-py2.py3-none-win_amd64.whl
Add packages
Remove
(Un)Select all
Install packages


WinPython Control Panel
Packages Options Advanced ?
Python 3.6 64bit: D:\WinPython-64bit-3.6.1.0Qt5\python-3.6.1.amd64
Install/upgrade packages
Uninstall packages
Name
Version
Description
pylint 1.6.5 Logilab code analysis module: analyzes Pyth...
pymc 2.3.6 Markov Chain Monte Carlo sampling toolkit.
pymc3 3.0 Markov Chain Monte Carlo sampling toolkit.
pymeta3 0.5.1 Pattern-matching language based on Met...
pymongo 3.4.0 Python driver for MongoDB <http://www...
pymunk 5.2.0
pyodbc 4.0.15 DB API Module for ODBC
pyopengl 3.1.1 Cross platform Python binding to OpenGL and ...
pypandoc 1.3.2 Thin wrapper for pandoc.
pyparsing 2.2.0 A Python Parsing Module
Uninstall packages

pymunk


Console 1/A
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr 9 2024, 14:05:25) [MSC v.1938 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 8.22.2 -- An enhanced Interactive Python.
In [1]: pip
Note: you may need to restart the kernel to use updated packages.
Usage:
C:\WPy64-31230\python-3.12.3.amd64\python.exe -m pip <command> [options]
Commands:
install Install packages.
download Download packages.
uninstall Uninstall packages.
freeze Output installed packages in requirements format.
inspect Inspect the python environment.
list List installed packages.
show Show information about installed packages.
check Verify installed packages have compatible dependencies.
config Manage local and global configuration.
search Search PyPI for packages.
cache Inspect and manage pip's wheel cache.
index Inspect information available from package indexes.
wheel Build wheels from your requirements.
hash Compute hashes of package archives.
completion A helper command used for command completion.
debug Show information useful for debugging.
help Show help for commands.
General Options:
IPython Console
History



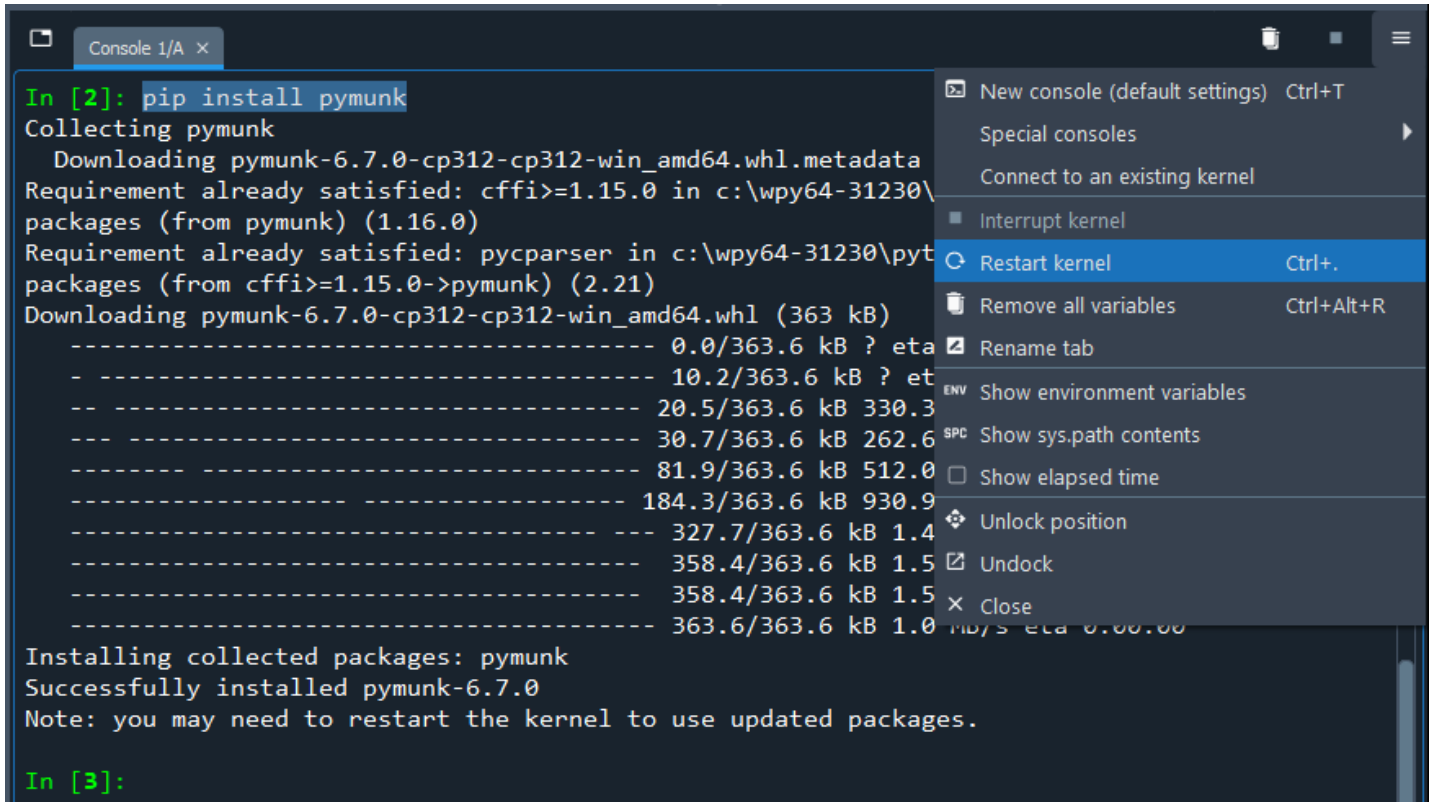
pymunk

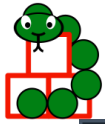
pip install pymunk

Console 1/A

```
In [2]: pip install pymunk
Collecting pymunk
  Downloading pymunk-6.7.0-cp312-cp312-win_amd64.whl.metadata (6.9 kB)
Requirement already satisfied: cffi>=1.15.0 in c:\wpy64-31230\python-3.12.3.amd64\lib\site-
packages (from pymunk) (1.16.0)
Requirement already satisfied: pycparser in c:\wpy64-31230\python-3.12.3.amd64\lib\site-
packages (from cffi>=1.15.0->pymunk) (2.21)
Downloading pymunk-6.7.0-cp312-cp312-win_amd64.whl (363 kB)
   ---------------------------------------- 0.0/363.6 kB ? eta -:--:--
   - -------------------------------------- 10.2/363.6 kB ? eta -:--:--
   -- ------------------------------------- 20.5/363.6 kB 330.3 kB/s eta 0:00:02
   --- ------------------------------------ 30.7/363.6 kB 262.6 kB/s eta 0:00:02
   -------- ------------------------------- 81.9/363.6 kB 512.0 kB/s eta 0:00:01
   ------------------ ------------------ 184.3/363.6 kB 930.9 kB/s eta 0:00:01
   ---------------------------------- --- 327.7/363.6 kB 1.4 MB/s eta 0:00:01
   ----------------------------------------  358.4/363.6 kB 1.5 MB/s eta 0:00:01
   ----------------------------------------  358.4/363.6 kB 1.5 MB/s eta 0:00:01
   ---------------------------------------- 363.6/363.6 kB 1.0 MB/s eta 0:00:00
Installing collected packages: pymunk
Successfully installed pymunk-6.7.0
Note: you may need to restart the kernel to use updated packages.

In [3]:
```


```
Installing collected packages: pymunk
Successfully installed pymunk-6.7.0
Note: you may need to restart the kernel to use updated packages
```



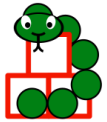


pymunk

pip show pymunk

Console 1/A

```
Restarting kernel...

---


In [1]: pip show
Note: you may need to restart the kernel to use updated packages.
WARNING: ERROR: Please provide a package name or names.

In [2]: pip show pymunk
Name: pymunk
Version: 6.7.0
Summary: Pymunk is a easy-to-use pythonic 2D physics library
Home-page: http://www.pymunk.org
Author: Victor Blomqvist
Author-email: vb@viblo.se
License: MIT License
Location: C:\WPy64-31230\python-3.12.3.amd64\Lib\site-packages
Requires: cffi
Required-by:
Note: you may need to restart the kernel to use updated packages.

In [3]: import pymunk
```



pymunk

Rigid bodies

A rigid body holds the physical properties of an object. (mass, position, rotation, velocity, etc.) It does not have a shape by itself. If you've done physics with particles before, rigid bodies differ mostly in that they are able to rotate.

Collision shapes

By attaching shapes to bodies, you can define the a body's shape. You can attach many shapes to a single body to define a complex shape, or none if it doesn't require a shape.

Constraints/joints

You can attach joints between two bodies to constrain their behavior.

Spaces

Spaces are the basic simulation unit in Chipmunk. You add bodies, shapes and joints to a space, and then update the space as a whole.

pymunk

```
import sys
import pygame
from pygame.locals import *
import pymunk #1

def main():
    pygame.init()
    screen = pygame.display.set_mode((600, 600))
    pygame.display.set_caption("Joints. Just wait and the L will tip over")
    clock = pygame.time.Clock()
    space = pymunk.Space() #2
    space.gravity = (0.0, 980.0)

    try:
        while True:
            for event in pygame.event.get():
                if event.type == QUIT:
                    sys.exit(0)
                elif event.type == KEYDOWN and event.key == K_ESCAPE:
                    sys.exit(0)

            screen.fill((127,127,255))
            space.step(1/50.0) #3
            pygame.display.flip()
            clock.tick(50)
    finally:
        pygame.quit()
        sys.exit()

if __name__ == '__main__':
    main()
```

The code will display a blank window, and will run a physics simulation of an empty space.

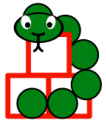
#1 We need to import pymunk in order to use it...

#2 We then create a space and set its gravity to something good. Remember that what is important is what looks good on screen, not what the real world value is. -980 will make a good looking simulation, but feel free to experiment.

#3 In our game loop we call the step() function on our space. The step function steps the simulation one step forward in time.

Note

It is best to keep the **step size constant** and **not adjust** it depending on the **framerate**. The physic simulation will work much better with a constant step size.

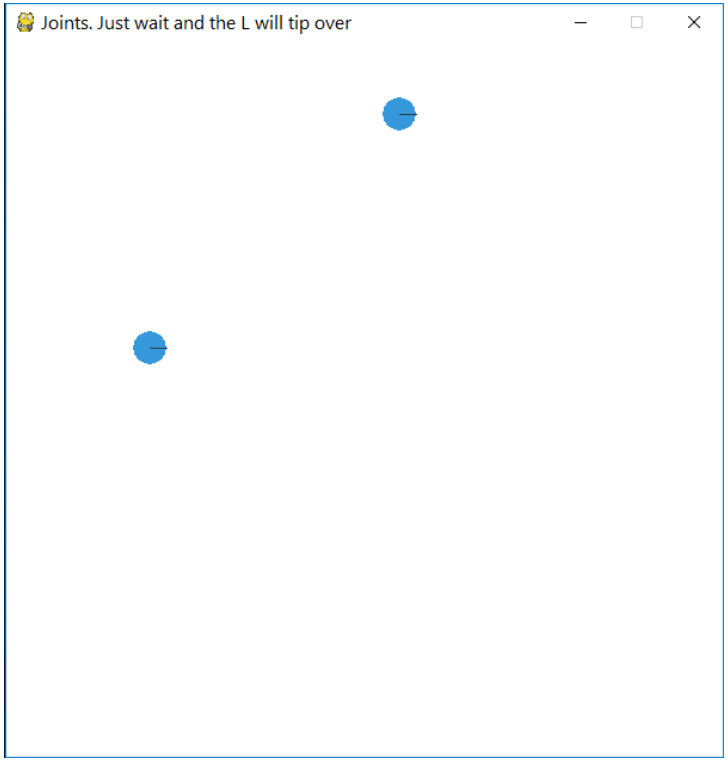


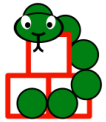
pymunk

```
def add_ball(space):
    """Add a ball to the given space at a random position"""
    mass = 1
    radius = 14
    inertia = pymunk.moment_for_circle(mass, 0, radius, (0,0))
    body = pymunk.Body(mass, inertia)
    x = random.randint(120,380)
    body.position = x, 550
    shape = pymunk.Circle(body, radius, (0,0))
    space.add(body, shape)
    return shape
```

1. All bodies must have their moment of inertia set. If our object is a normal ball we can use the predefined function **`moment_for_circle`** to calculate it given its mass and radius. However, you could also select a value by experimenting with what looks good for your simulation.
2. After we have the inertia we can create the body of the ball.
3. And we set its position
4. And in order for it to collide with things, it needs to have one (or many) collision shape(s).
5. Finally we add the body and shape to the space to include it in our simulation.

pymunk





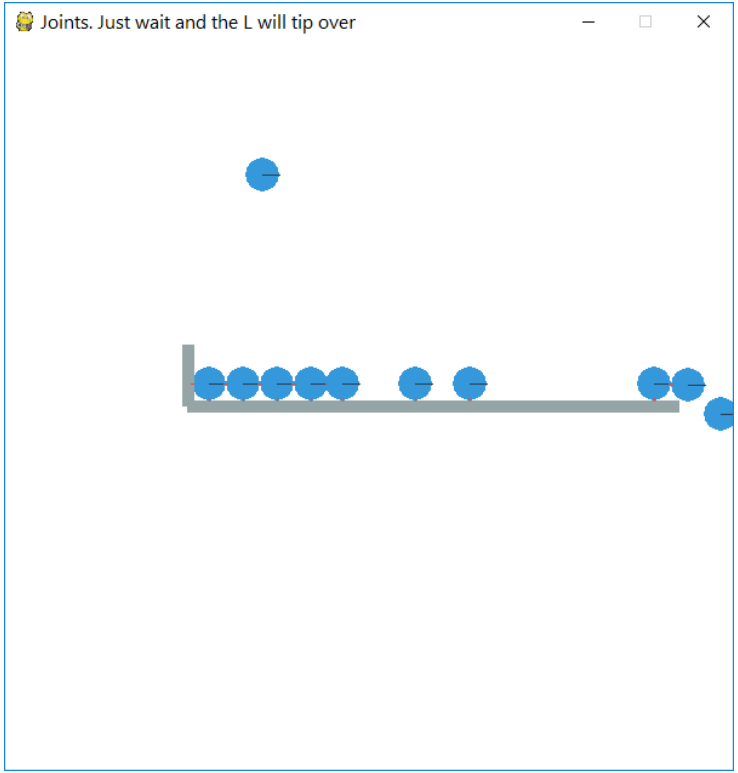
pymunk

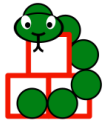
So lets add something the balls can land on, two static lines forming an L. As with the balls we start with a function to add an L to the space:

```
def add_static_L(space):
    body = pymunk.Body(body_type = pymunk.Body.STATIC) # 1
    body.position = (300, 300)
    l1 = pymunk.Segment(body, (-150, 0), (255, 0), 5) # 2
    l2 = pymunk.Segment(body, (-150, 0), (-150, 50), 5)
    space.add(l1, l2) # 3
    return l1,l2
```

1. We create a "**static**" body. The important step is to never add it to the space like the dynamic ball bodies. Note how static bodies are created by setting the **body_type** of the body.
2. A line shaped shape is created here.
3. Again, we only add the segments, not the body to the space.

PHYSICS in COMPUTER ANIMATIONS and GAMES





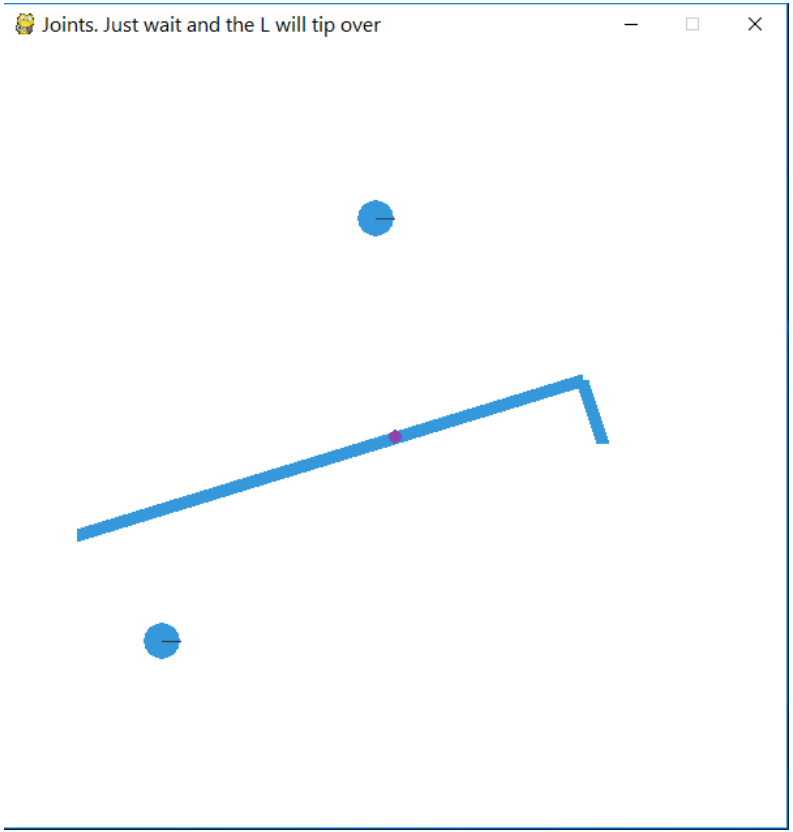
pymunk

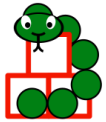
A static L shape is pretty boring. So lets make it a bit more exciting by adding **two joints**, one that it can **rotate around**, and one that **prevents it from rotating** too much.

```
def add_L(space):
    rotation_center_body = pymunk.Body(body_type = pymunk.Body.STATIC) # 1
    rotation_center_body.position = (300, 300)

    body = pymunk.Body(10, 10000) # 2
    body.position = (300, 300)
    l1 = pymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
    l2 = pymunk.Segment(body, (-150.0, 0), (-150.0, 50.0), 5.0)

    rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0)) # 3

    space.add(l1, l2, body, rotation_center_joint)
    return l1,l2
```

1. This is the rotation center body.
2. The L shape will now be moving in the world, and therefor it can no longer be a static body.
3. A pin joint allow two objects to pivot about a single point.

pymunk





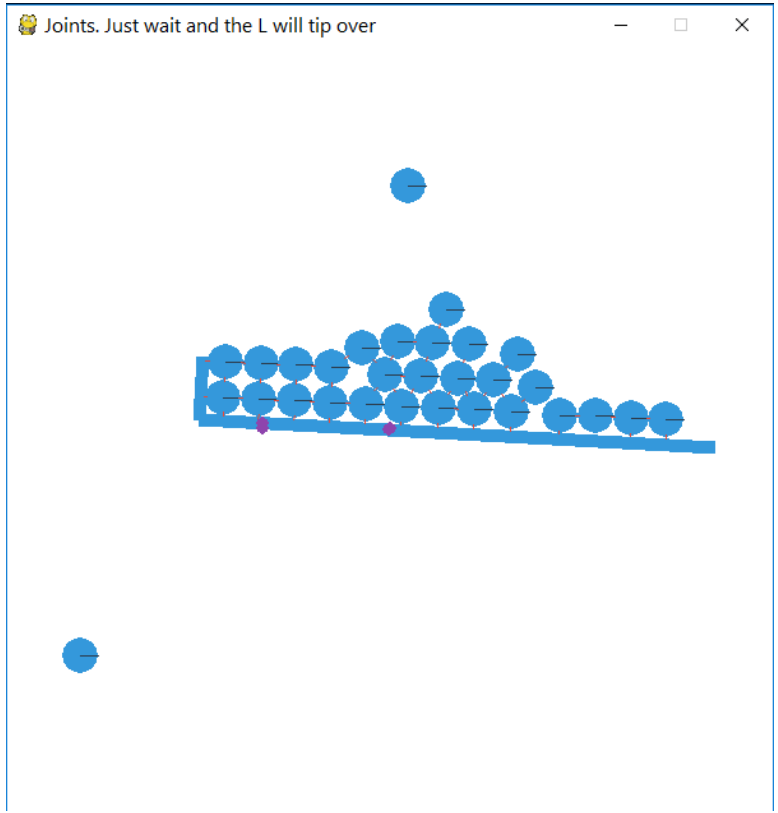
pymunk

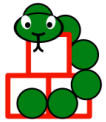
To constrain the rotating L shape to create a more interesting simulation.

```
def add_L(space):
    rotation_center_body = pymunk.Body(body_type = pymunk.Body.STATIC)
    rotation_center_body.position = (300,300)

    rotation_limit_body = pymunk.Body(body_type = pymunk.Body.STATIC) # 1
    rotation_limit_body.position = (200,300)

    body = pymunk.Body(10, 10000)
    body.position = (300,300)
    l1 = pymunk.Segment(body, (-150, 0), (255.0, 0.0), 5.0)
    l2 = pymunk.Segment(body, (-150.0, 0), (-150.0, 50.0), 5.0)
    rotation_center_joint = pymunk.PinJoint(body, rotation_center_body, (0,0), (0,0))
    joint_limit = 25
    rotation_limit_joint = pymunk.SlideJoint(body, rotation_limit_body, (-100,0), (0,0), 0, joint_limit) # 2
    space.add(l1, l2, body, rotation_center_joint, rotation_limit_joint)
    return l1,l2
```

1. We add a body.
2. Create a slide joint. It behaves like pin joints but have a minimum and maximum distance. The two bodies can slide between the min and max.

pymunk





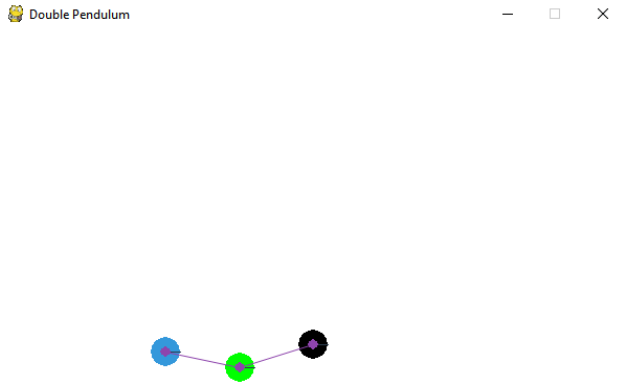
pymunk

You might notice that we never delete balls. This will make the simulation require more and more memory and use more and more cpu.

```
balls_to_remove = []

for ball in balls:
    if ball.body.position.y < 0:        # 1
        balls_to_remove.append(ball)    # 2

for ball in balls_to_remove:
    space.remove(ball, ball.body)       # 3
    balls.remove(ball)                  # 4
```

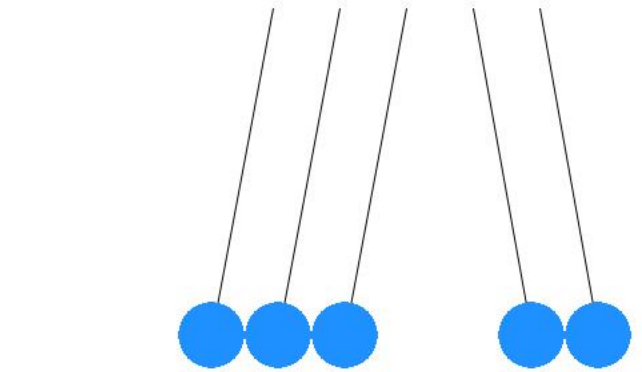
1. Loop the balls and check if the body.position is less than 0.
2. If that is the case, we add it to our list of balls to remove.
3. To remove **an object** from the **space**, we **need to remove its shape and its body**.
4. And then we remove it from our list of balls.

pymunk



test_10_doublePendulum.py

Classwork: Newton's Cradle



hint: shape.elasticity = 0.9999999