## Course Plan



72

## Forces: examples

...
$$\vec{f} = \text{function}(\mathbf{p}, \dots)$$
...

Primarily, a function of the positions

But not always, and sometimes not only of positions (also: velocities? Global time?)

- Gravity
  - Constant · *m*, near the surface of a planet
  - Function of positions in a space simulation
- Wind pressure
  - Depends on the area exposed in the wind direction
- Electrical / magnetic forces
- Buoyancy *(ita: forza di Archimede)*
  - Depends on the weight of the submerged volume
- Mechanical springs
  - simple model: Hooke's law – see later
- Shock waves (explosions)
- Fake / "Magic" control forces
  - added for controlling the evolution of the system, not physically justified

73

## Non-forces: examples

$$\dots$$
$$\vec{f} = \text{function}(\mathbf{p}, \dots)$$
$$\dots$$

- Real-world forces can be simulated by things that aren't technical "forces":
  - Frictions
    - Can be simulated by: **drag** (see later)
  - Impacts & other violent things
    - In reality: very short, very strong forces
    - Duration << *dt*
    - Must be simulated by: **impulses** (see later)
  - Resistance forces
    - In reality, an internal force that contrast an external force (such as gravity)
      E.g.: what prevents your computer to fall through the table
    - E.g.: what prevents a pencil to contract when you push it on the paper
    - Necessary to simulated the solidity bodies
    - Can be simulated by: **positional constraints** (see later)

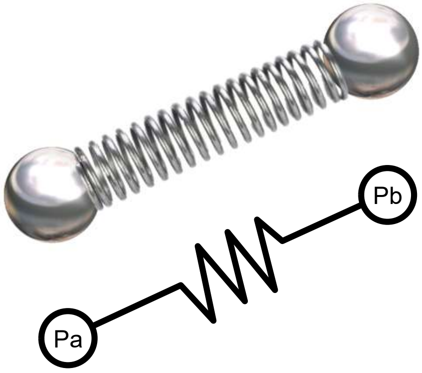
74

## Forces: control forces

- Example: the player pressing the forward button
  ⇒ a forward force is applied to his/her avatar
  - no physical justification
  - "Don't ask questions, physics engine"
- According to many:
  it's better when that's not done much
  - the more physically justified the forces, the better
  - for example: does the car accelerate…
    because a torque is appied to its two traction wheels VS
    because a force is applied to its body
  - usually much harder to cortrol
  - see also: gameplay VS cosmetics, control VS realism, emerging behaviours

75

## Forces: Springs



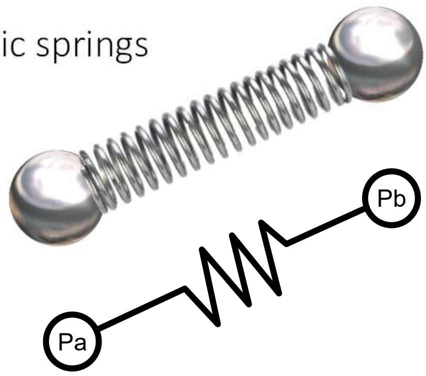

Hooke's law:

$$\vec{f_a} = k(\ell - \|\mathbf{p}_b - \mathbf{p}_a\|)\underbrace{\frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}}_{\text{force direction (versor)}}$$

76

## Forces: Springs (Hooke's law)

- Simplified model for elastic springs
- One spring connects two particles in $\mathbf{p}_a$ and $\mathbf{p}_b$
- Characterized by:
  1. Rest length $\ell$
  2. Stiffness $k$
- Spring force: counteracts expansion and compression




$$\vec{f_a} = k(\ell - \|\mathbf{p}_b - \mathbf{p}_a\|)\frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}$$

$$\vec{f_b} = - \vec{f_a}$$

77

## Forces: Springs (Hooke's law)

$$\vec{f_a} = \underbrace{k(\overbrace{\ell - \|\mathbf{p}_b - \mathbf{p}_a\|}^{\text{elongation / compression}})}_{\substack{\text{force magnitude}\\ \text{(scalar)}\\ \text{(positive or negative)}}} \underbrace{\frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}}_{\substack{\text{force direction}\\ \text{(versor)}}}$$

⇐ force to be applied to particle a

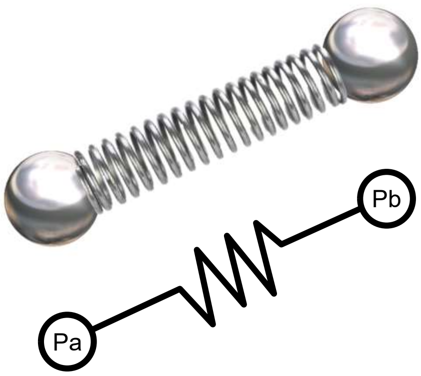
$$\vec{f_b} = -\vec{f_a}$$

⇐ force to be applied to particle b

78

## Forces: springs friction

- A dissipative force
  - Damping factor $k_D$
- Wants to slow down elongation / shrinking



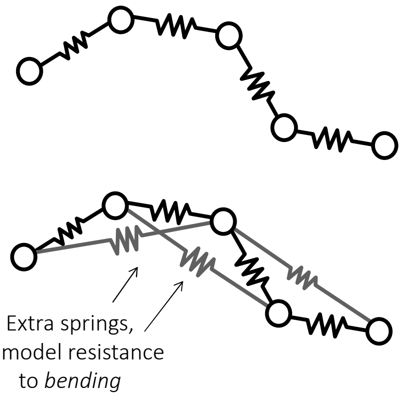

$$\hat{d} = \frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}$$

$$\vec{f_a} = k_D\big(\hat{d} \cdot (\vec{v}_b - \vec{v}_a)\big)\,\hat{d}$$

79

## Mass and Spring systems

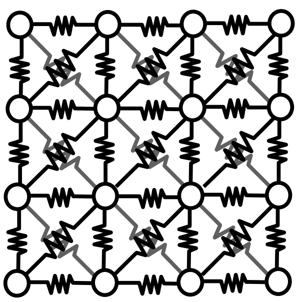
- Useful for deformable objects
- for instance: elasitic ropes (or hairs)

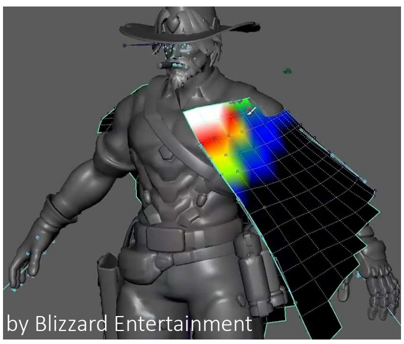



80

## Mass and Spring systems

- For instance: cloth






81

# Mass and Spring systems can model…

- Elastic deformable objects (aka "soft bodies")
  - Elastic = go back to original shape
  - Easily modelled as compositions of (ideal) springs.
- Plastic deformable objects? (yes, but not easy)
  - Plastic = assume deformed pose permanently
  - Dynamically change rest-length $L$ in response to large compression/stretching, in certain conditions (not easy)
- Rigid bodies / inextensible ropes ? (they can't)
  - Increase spring stiffness? $k \to \infty$
  - Makes sense, physically, but…
  - Large $k \Rightarrow$ large f $\Rightarrow$ instability $\Rightarrow$ unfeasibly small d$t$ needed
  - Doesn't work. How, then? see later

82

# Example of forces: gravitational force on a plantet surface

- Given a particle with (gravitational) mass $m$

some global constant dependent on… the planet

$$\vec{f_a} = g\, m \;\; \hat{\mathrm{d}}_{\mathrm{DOWN}}$$

force magnitude (positive scalar)

force direction (versor)

Notes:
- does not depend on position, only on mass
- will produce a constant acceleration (regardless of mass!) when divided by (inertial) mass $m$

83

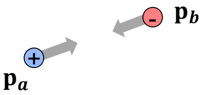
# Example of forces: gravitational forces in open space

- Given two charged particles in $\mathbf{p}_a$ and $\mathbf{p}_b$ with (gravitational) masses $m_a$ and $m_b$

some global constant

$$\vec{f_a} = \underbrace{\frac{G\, m_a\, m_b}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}}_{\text{force magnitude (positive scalar)}} \underbrace{\frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}}_{\text{force direction (versor)}} = \frac{-K\, q_a\, q_b}{\|\mathbf{p}_b - \mathbf{p}_a\|^3} (\mathbf{p}_b - \mathbf{p}_a)$$

$$\vec{f_b} = -\vec{f_a}$$



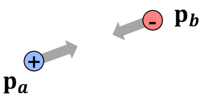



# Example of forces: electric forces

- Given two charged particles in $\mathbf{p}_a$ and $\mathbf{p}_b$ with positive or negative charges $q_a$ and $q_b$

some global constant

$$\vec{f_a} = \underbrace{\frac{-K\, q_a\, q_b}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}}_{\text{force magnitude (scalar) positive or negative}} \underbrace{\frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}}_{\text{force direction (versor)}} = \frac{-K\, q_a\, q_b}{\|\mathbf{p}_b - \mathbf{p}_a\|^3} (\mathbf{p}_b - \mathbf{p}_a)$$

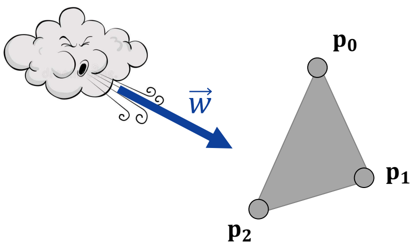





Marco Tarini
Università degli studi di Milano

# Example of forces: wind pressure

- Wind is a force acting on surfaces
- The larger the exposed surface to the wind, the larger the force
- The more orthogonal the surface to the wind direction, the larger the force
- The stronger the wind pressure $\vec{w}$ (a vector), the larger the force




$$\vec{f} = \left|\underbrace{\overbrace{\frac{1}{2}(\mathbf{p}_1 - \mathbf{p}_0)\times(\mathbf{p}_1 - \mathbf{p}_0)}^{\text{area vector}}\cdot\vec{w}}_{\text{force magnitude (scalar)}}\right| \underbrace{\frac{\vec{w}}{\|\vec{w}\|}}_{\text{force direction (versor)}}$$

(apply 1/3 of $\vec{f}$ on each particle)

86

# Example of forces: etc

- Remember all forces acting on a particle add up! (vector summatory)

one step

$$\vec{\mathrm{f}} \leftarrow fun(\mathbf{p}, \ldots)$$
$$\vec{\mathrm{a}} \leftarrow \vec{\mathrm{f}}\,/m$$
$$\mathbf{p} \leftarrow \mathbf{p} + \vec{\mathrm{v}} \cdot dt$$
$$\vec{\mathrm{v}} \leftarrow \vec{\mathrm{v}} + \vec{\mathrm{a}} \cdot dt$$

87

## Attrition (or friction) forces

- Isotropic friction forces :
  - a force that oppose any motion, regardless of its direction
  - direction: always opposite of current velocity direction
  - magnitude: proportional to the speed (= magnitude of velocity vector)
  - note: this force depends on velocity, not just positions.
  - models the effect of the medium where the motion happens (air, water, thin space…)
  - the denser the medium, the stronger the force (water >> air >> thin space)
- Planar friction forces:
  - A force that happens when things slide against each other
  - Always parallel to the contact plane (orthogonal to the normal)

88

## Attrition (or friction) forces by velocity dumping

- A useful trick to quickly simulate isotropic friction: “velocity damping”
  - we simply reduce all velocity vectors by a fixed proportion
  - for example: scale velocity down by 2% per second (drag = 0.02 / sec) (that is, scale velocity vector by a factor 0.98)
  - It makes sense! Higher speed = more attrition = more loss of speed. Attrition = a “fixed tax” on speed.

89

# Velocity Damping: pseudo-code

```
Vec3 position = …
Vec3 velocity = …

void initState(){
   position = …
   velocity = …
}

void physicStep( float dt )
{
   Vec3 acceleration = force( positions ) / mass;
   position += velocity * dt;
   velocity += acceleration * dt;
   velocity *= (1.0 – DRAG * dt);
}

void main(){
  initState();
  while (1) do physicStep( 1.0 / FPS );
}
```

90

# Velocity Damping: notes

- Velocity Damping helps for robustness,
  - avoids energy to ever increase
- Problems of Velocity Damping
  - it tends to exaggerate frictions of, e.g., air, especially in absence of contacts
  - crude approximation: attrition forces are not really *linear* with speed
- In practice:
  - low drag: hardly noticeable (except in the long run)
  - high drag: everything feels like to be moving in molasses; (ita: *melassa*); everything quickly grinds to a halt
  - super high drag: (e.g. 50% per sec) basically, no inertia anymore useful to converge to (local) minimal energy states: simulation turns into a solver for statics

91

## Continuity of pos and vel

- In real Newtonian physics the state (pos and vel) can only change *continuously*
  - No sudden jump!
- In practice, sometimes is useful to artificially break continuity in the simulations
- Discontinuous changes:
  - in positions: "teleports"
  - in velocity: "impulses"
  - (those are not necessary variations justified by forces)

92

## Dynamics displacements VS kinematic

a discontinuous change of state (position)

...
$$p = p + \vec{v} \cdot dt$$
...

aka dynamic displacements

Justified by physics

...
$$p = p + dp$$
...

aka Kinematic displacements

Just "teleportation"

93

## Impulses VS Forces

a discontinuous change of state (velocity)

…

$$\vec{v} = \vec{v} + \left(\vec{f} / m\right) \cdot dt$$

…

- Forces (continuous)
  - Continuous application
  - every frame

…

$$\vec{v} = \vec{v} + \left(\vec{i} / m\right)$$ ~~$\cdot dt$~~

…

- Impulses
  - Infinitesimal time
  - una tantum

they model very intense but short forces (such as impacts)

94

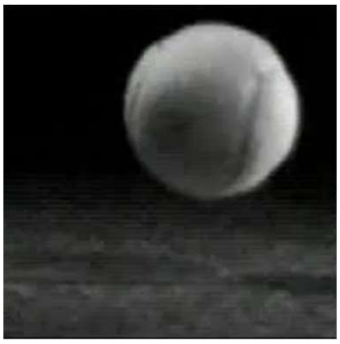
## Impulses VS Forces

- Force :
  - it determines an acceleration
  - acc determines a (continuous!) change of vel
  - physically correct
- Impulse :
  - a (discontinuous!) change of vel
  - useful to control a simulation (direct change of velocity)
  - a physical interpretation: a force with:
    - application time approaching **zero**
    - magnitude approaching **infinity**
  - Useful to model phenomena with a time scale << *dt*
    - ex: a tennis ball rebounding against a tennis racket

95

## Impulses VS Forces

- what does *truly* happen when it bounces off the ground?

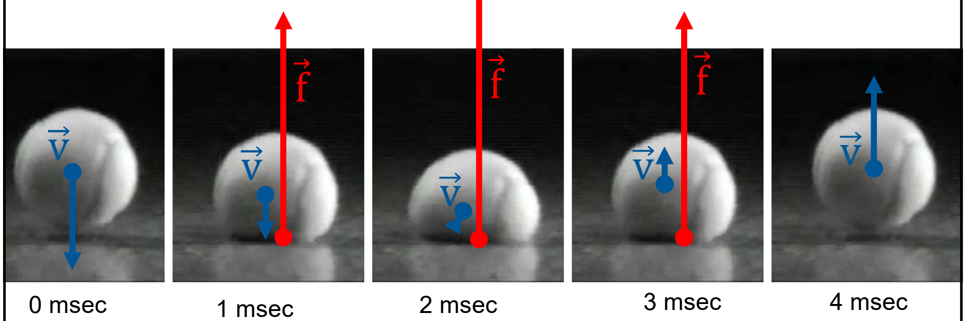


  - very strong forces (but not infinite)
  - applied for a very short time (but not instantaneous)
  - see *collision response* later for details
    about the impulse-based approximations

96

## Impulses VS Forces

- what does *truly* happen when it bounces off the ground?

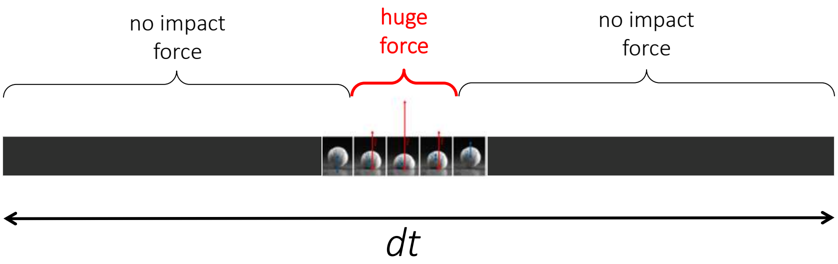



  - very strong forces (but not infinite)
  - applied for a very short time (but not instantaneous)
  - see *collision response* later for details
    about the impulse based approximations

97

## Impulses VS Forces

- what does *truly* happen when it bounces off the ground?




- This can only be modelled as an *impulse*, not a force
- *See also collision response, later*

98

## Effect of integration errors of System Energy

- Because of integration errors:
  simulated solutions ≠ "real" solutions
- In a real system, the total energy can never increase
  - typically, it *decreases* over time, due to dissipations
  - that is, attrition turns *dynamic energy* into *heat*
- Therefore, a particularly nasty integration error is when the total energy of the system *increases* over time
  - e.g.: a pendulum swings wider and wider
- Particularly bad because:
  - compromises stability
    (velocity = big, displacements = crazy, error = crazy)
  - compromises plausibility
    (we can *see* it's wrong)
- A simple way to avoid this:
  make sure the simulation always includes attritions
  - makes simulation more stable + robust

99

# Other numerical integrators
## (“numerical ways to compute integrals”)

- Some commonly used alternatives (among MANY!):
  - “Forward” Euler method (the one seen so far)
  - Symplectic Euler method
  - Leapfrog method
  - Verlet method
- These are just variants of each other – let’s see them!
  - From the code point of view, no big change
  - They can differ in accuracy / behavior
  - They can have different “orders of accuracy”
  - Note: a more accurate method is also more efficient (larger $dt$ are possible, so fewer steps are necessary)

100

# Forward Euler Method: limitations

- efficiency / accuracy: not too good
  - error accumulated over time = linear in dt
  - it’s only a “first order” method
  - Doubles the steps = halve the *dt* , only halves the errors (can be better, but no guarantees)
- scarce stability for large *dt*
- minor problem: no reversibility, *even in theory*
  - real Newtonian Physics is reversible: flip all velocities and forces ⇒ go backward in time.
  - In our simulation (with Euler): this doesn’t work exactly
  - Ability to go reverse a simulation would be useful in games! E.g. replays in a soccer game ?
  - Pro tip: basically, reverse time direction never done like this To go backward in time accurately, store states

101

## Forward Euler

init state

$$\mathbf{p} \leftarrow \cdots$$
$$\vec{\mathrm{v}} \leftarrow \cdots$$

one step

$$\vec{\mathrm{f}} \leftarrow fun(\mathbf{p}, \dots)$$
$$\vec{\mathrm{a}} \leftarrow \vec{\mathrm{f}}/m$$
$$\mathbf{p} \leftarrow \mathbf{p} + \vec{\mathrm{v}} \cdot dt$$
$$\vec{\mathrm{v}} \leftarrow \vec{\mathrm{v}} + \vec{\mathrm{a}} \cdot dt$$

$$t = t + dt$$

102

## Symplectic Euler

init state

$$\mathbf{p} \leftarrow \cdots$$
$$\vec{\mathrm{v}} \leftarrow \cdots$$

one step

$$\vec{\mathrm{f}} \leftarrow fun(\mathbf{p}, \dots)$$
$$\vec{\mathrm{a}} \leftarrow \vec{\mathrm{f}}/m$$
$$\vec{\mathrm{v}} \leftarrow \vec{\mathrm{v}} + \vec{\mathrm{a}} \cdot dt$$
$$\mathbf{p} \leftarrow \mathbf{p} + \vec{v} \cdot dt$$

$$t = t + dt$$

103

# Forward Euler *pseudo code*

```
Vec3 position = …
Vec3 velocity = …

void initState(){
   position = …
   velocity = …
}

void physicStep( float dt )
{
   Vec3 acceleration = compute_force( position ) / mass;
   position += velocity     * dt;
   velocity += acceleration * dt;
}

void main(){
  initState();
  while (1) do physicStep( 1.0 / FPS );
}
```

Equivalent to…

$$\vec{f}_i \leftarrow function(p_i, \dots)$$
$$\vec{a}_i \leftarrow \vec{f}/m$$
$$\vec{v}_{i+1} \leftarrow \vec{v}_i + \vec{a}_i \cdot dt$$
$$p_{i+1} \leftarrow p_i + \vec{v}_i \cdot dt$$

104

# Symplectic Euler *pseudo code* (aka semi-implicit Euler)

```
Vec3 position = …
Vec3 velocity = …

void initState(){
   position = …
   velocity = …
}

void physicStep( float dt )
{
   Vec3 acceleration = compute_force( position ) / mass;
   velocity += acceleration * dt;
   position +=     velocity * dt;
}

void main(){
  initState();
  while (1) do physicStep( 1.0 / FPS );
}
```

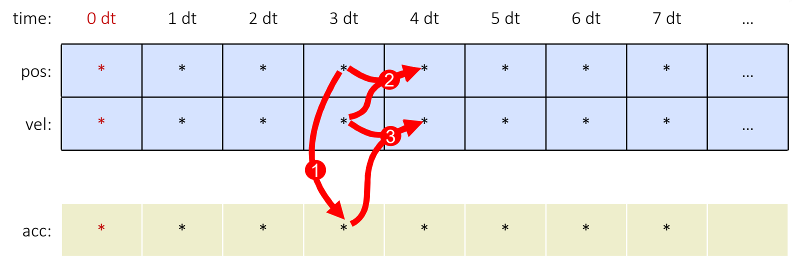
just flip the order

Equivalent to…

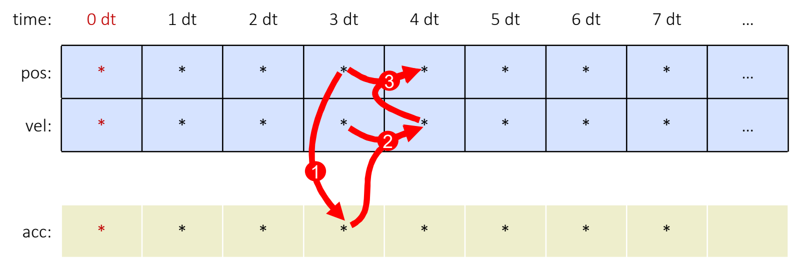
$$\vec{f}_i \leftarrow function(p_i, \dots)$$
$$\vec{a}_i \leftarrow \vec{f}/m$$
$$\vec{v}_{i+1} \leftarrow \vec{v}_i + \vec{a}_i \cdot dt$$
$$p_{i+1} \leftarrow p_i + \vec{v}_{\mathbf{i+1}} \cdot dt$$

105

## Forward Euler:




## Symplectic Euler:




106

```
acceleration = compute_force( position ) / mass;
velocity += acceleration * dt;
position += velocity * dt;

acceleration = compute_force( position ) / mass;
velocity += acceleration * dt;
position += velocity * dt;

acceleration = compute_force( position ) / mass;
velocity += acceleration * dt;
position += velocity * dt;


position += velocity * dt;
acceleration = compute_force( position ) / mass;
velocity += acceleration * dt;

position += velocity * dt;
acceleration = compute_force( position ) / mass;
velocity += acceleration * dt;

position += velocity * dt;
acceleration = compute_force( position ) / mass;
velocity += acceleration * dt;
```

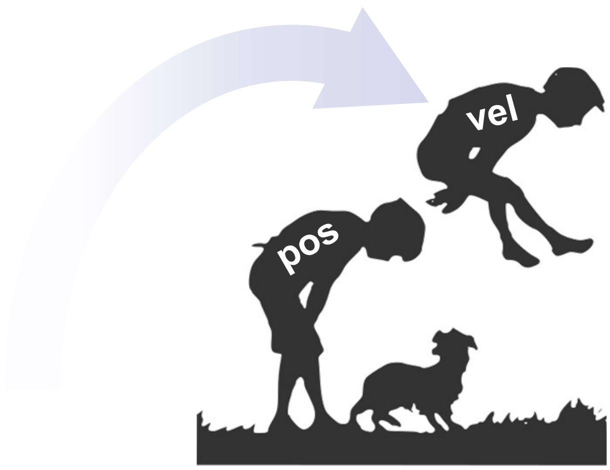
107

## Forward Euler VS Symplectic Euler (warning: over-simplifications)

- From the code point of view, they are very similar
- The semantics changes:
  - in Symplectic Euler
    the position altered using *next frame* velocity
  - (it's "wrong", in a sense – but works better)
- Similar properties, but better in practice
  - Same order of convergence (still just 1 ☹)
  - On average, better behavior:
    more stable and accurate

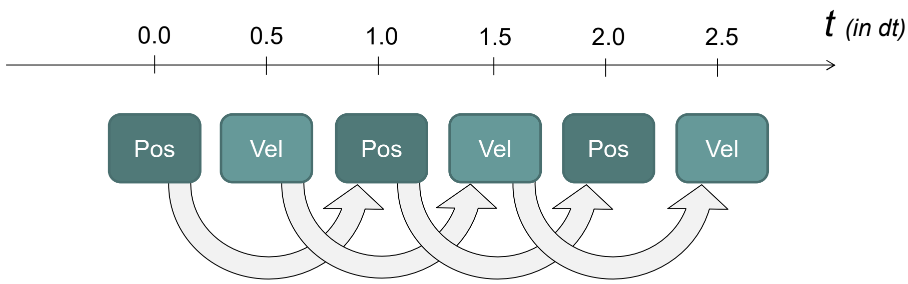
108

## Leapfrog Integration



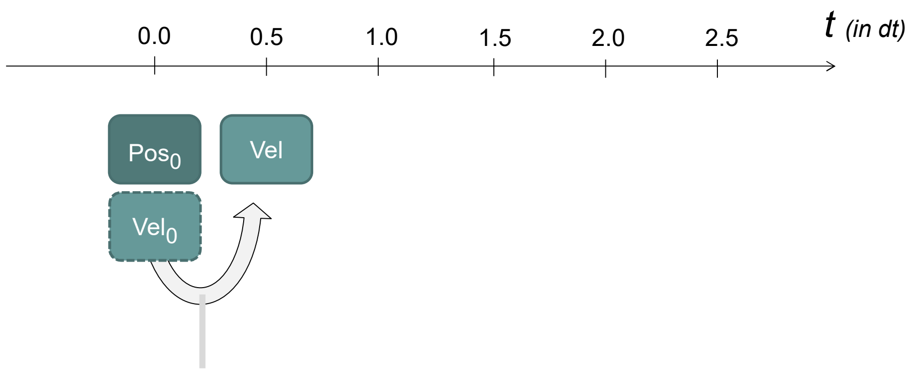

109

# Leapfrog Integration




110

# Leapfrog Integration first step

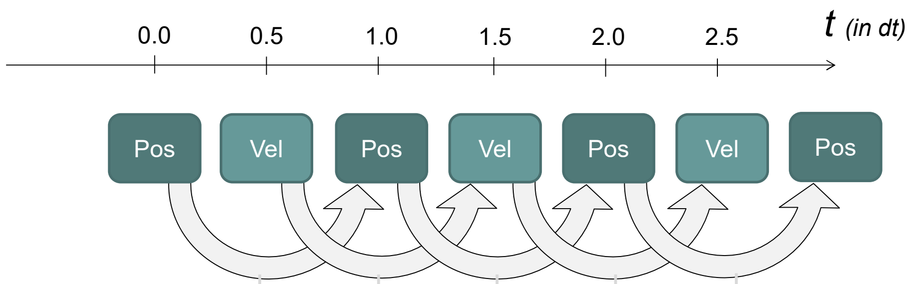



$$a = f(p_0\ ,\ \ldots)$$

$$\vec{v}_{0.5} = \vec{v}_0 + \vec{a} \cdot dt / 2$$

111

# Leapfrog Integration




$$p_1 = p_0 + \vec{v}_{0.5} \cdot dt \qquad p_2 = p_1 + \vec{v}_{1.5} \cdot dt \qquad p_3 = p_2 + \vec{v}_{2.5} \cdot dt$$

$$a = f(p_1\ ,\ \ldots) \qquad a = f(p_2\ ,\ \ldots)$$

$$\vec{v}_{1.5} = \vec{v}_{0.5} + a \cdot dt \qquad \vec{v}_{2.5} = \vec{v}_{1.5} + a \cdot dt$$

112

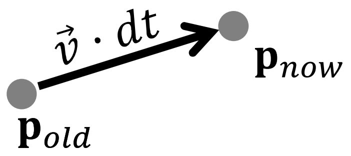
# Leapfrog method: pros and cons

- Same cost as Euler – and basically same code
  - Velocity stored in status = velocity “half a *dt* ago” (and after updating it: “half a frame in the future”)
  - Only real difference: the initialization of speed
- Better theorical accuracy, for the same *dt*
  - better asymptotic behavior: it’s a second order instead of first!
  - cumulated error: proportional to $dt^2$ instead of $dt$
  - error per frame: proportional to $dt^3$ instead of $dt^2$
- Bonus: fully reversible!
  - in theory only. Beware numerical errors.
- But: requires fixed *dt* during all the simulation
  - for the theory to work as advertised

113

## Verlet integration method

- Idea: remove velocity from state
- Current velocity is implicit
- It's defined from:
  - current pos $\mathbf{p}_{now}$
  - last pos $\mathbf{p}_{old}$
    which we need to record




$$\mathbf{p}_{now} = \mathbf{p}_{old} + \vec{v} \cdot dt$$ ⇐ Euler & variants

$$\Longrightarrow$$

$$\vec{v} = (\mathbf{p}_{now} - \mathbf{p}_{old})/dt$$ ⇐ Verlet

114

## Verlet integration method

init state:

$$\mathbf{p}_{now} = \ldots$$
$$\mathbf{p}_{old} = \ldots$$

one step:

$$\vec{f} = funct(\mathbf{p}_{now}, \ldots)$$
$$\vec{a} = \vec{f}/m$$
$$\vec{v} = (\mathbf{p}_{now} - \mathbf{p}_{old})/dt$$
$$\vec{v} = \vec{v} + \vec{a} \cdot dt$$
$$\mathbf{p}_{\text{next}} = \mathbf{p}_{now} + \vec{v} \cdot dt$$

expanding this...

115

## Verlet integration method

init state:

$\mathbf{p}_{now} \leftarrow \ldots$
$\mathbf{p}_{old} \leftarrow \ldots$

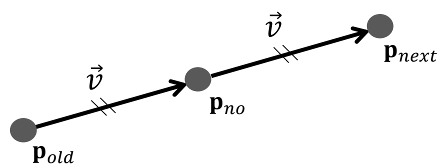
one step:

$\vec{f} \leftarrow funct(\mathbf{p}_{now})$
$\vec{a} \leftarrow \vec{f}/m$
$\mathbf{p}_{next} \leftarrow 2\mathbf{p}_{now} - \mathbf{p}_{old} + \vec{a} \cdot dt^2$

$\mathbf{p}_{old} \leftarrow \mathbf{p}_{now}$
$\mathbf{p}_{now} \leftarrow \mathbf{p}_{next}$

116

## Verlet integration method: geometric interpretation




$\mathbf{p}_{next} = 2 \cdot \mathbf{p}_{now} - 1 \cdot \mathbf{p}_{old}$

$\mathbf{p}_{next}$ can be written as an extrapolation of $\mathbf{p}_{now}$, $\mathbf{p}_{old}$ :

$\mathbf{p}_{next} = mix(\ \mathbf{p}_{old},\ \mathbf{p}_{now},\ 2)$

117

## Verlet: characteristics

- Velocity is kept implicit
  - but that doesn't save RAM:
    we need to store previous position instead
  - (a point instead of a vector: same memory)
- Good efficiency / accuracy ratio
  - Per-step error: linear with *dt*
  - accumulated error: order of $dt^2$ (second order method)
- Extra bonus: reversibility
  - it's possible to go backward in *t* and
    reach the initial state from any state
  - only in theory… careful with implementation details

118

## Verlet: *caveats*
(see next lecture for solutions)

⚠ it assumes a constant *dt* (time-step duration)
- if *dt* varies: corrections are needed! (how?)

⚠ Q: how to act on velocity (which is now implicit)?
- for example, how to apply impulses ?
- A: change $\mathbf{p}_{old}$ instead (how?)

⚠ Q: how to act of positions w/o impacting velocity?
- for example, to apply teleports / kinematic motions ?
- A: change both $\mathbf{p}_{new}$ and $\mathbf{p}_{old}$ (how?)

⚠ Q: how to apply velocity damps?
- A: act on $\mathbf{p}_{old}$ or $\mathbf{p}_{next}$ (how?)

119