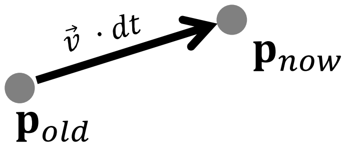
Course Plan

lec. 1: Introduction ●
lec. 2: Mathematics for 3D Games ●●●●●
lec. 3: Scene Graph ●
lec. 4: Game 3D Physics ●●● + ●●
lec. 5: Game Particle Systems ◖
lec. 6: Game 3D Models ●◖
lec. 7: Game Textures ●●
lec. 8: Game 3D Animations ●●●
lec. 9: Game 3D Audio ●
lec. 10: Networking for 3D Games ●
lec. 11: Artificial Intelligence for 3D Games ●
lec. 12: Game 3D Rendering Techniques ●●

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



$$\mathbf{p}_{now} = \mathbf{p}_{old} + \vec{v} \cdot dt$$

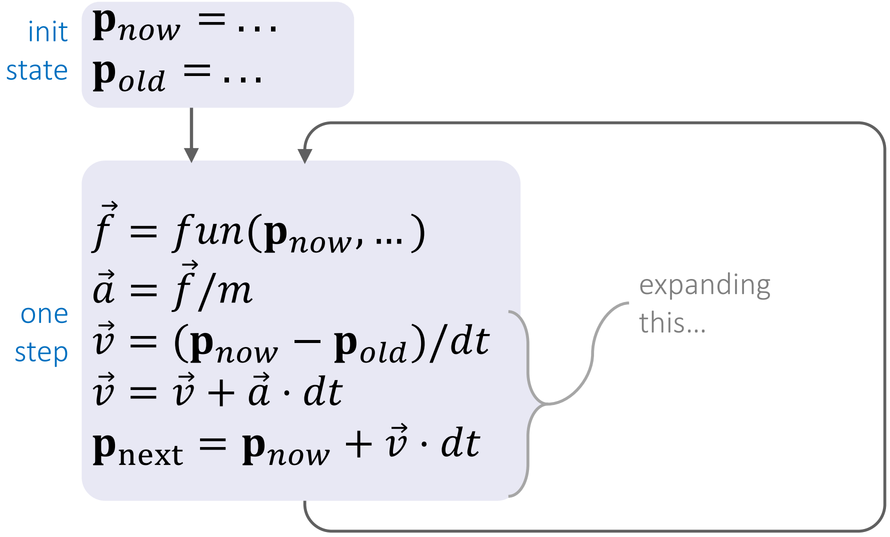
⇦ Euler & friends

$$\Longrightarrow$$

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

⇦ Verlet

Verlet integration method



init state

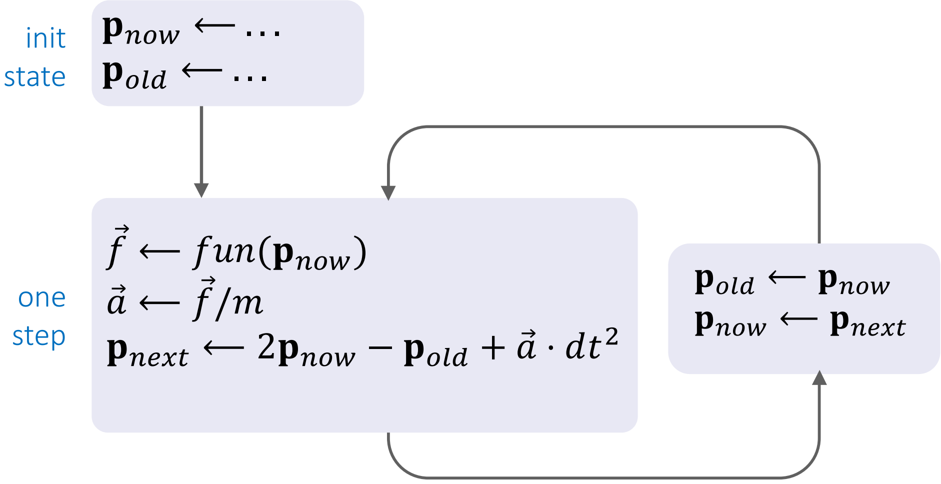
$$\mathbf{p}_{now} = \ldots$$
$$\mathbf{p}_{old} = \ldots$$

one step

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

expanding this...

Verlet method



init state

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

one step

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

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

Verlet: characteristics

- Velocity is kept implicit
  - but that doesn't save RAM:
    we need so store previous position instead
- 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
  - that, in theory… careful with implementation details

Verlet: *caveats*

⚠ it assumes a constant *dt* (time-step duration)

- if it varies: corrections are needed! (which ones?)

⚠ Q: how to act on velocity (which is now implicit)?

- e.g. to apply impulses
- A: change $\mathbf{p}_{old}$ instead

⚠ Q: how to act of positions w/o impacting velocity?

- e.g. to apply teleports / kinematic motions
- A: displace both $\mathbf{p}_{new}$ and $\mathbf{p}_{old}$ by the same amount

⚠ Q: how to apply velocity damps?

- A: act on $\mathbf{p}_{old}$ or $\mathbf{p}_{next}$ (see below)

dt updates in Verlet

(if they are not constant)

Problem:

if dt now changes to a new dt'

then, all $\mathbf{p}_{old}$ must be updated to some $\mathbf{p}'_{old}$

Find $\mathbf{p}'_{old}$:

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

current velocity $\vec{v}$ and position $\mathbf{p}_{now}$ must not change

$$\Longrightarrow$$

$$\mathbf{p}'_{old} = \mathbf{p}_{now} \cdot (dt - dt')/dt + \mathbf{p}_{old} \cdot dt'/dt$$

Velocity damping in Verlet

- Velocity at next frame: $\vec{v} = (\mathbf{p}_{next} - \mathbf{p}_{now})/dt$

implicit

- We want to multiply $\vec{v}$ a factor c_{damp}
  - before applying accelerations

e.g. 0.98 obtained as (1-$dt \cdot c_{DRAG}$)

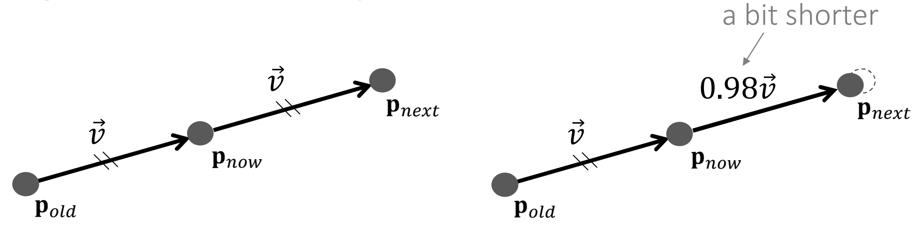
- We can do that using a more general formula for $\mathbf{p}_{next}$

$$\mathbf{p}_{next} = \quad 2 \quad \cdot \mathbf{p}_{now} \quad - \quad 1 \quad \cdot \mathbf{p}_{old} \quad + dt^2 \cdot \vec{a}$$

$$\Downarrow$$

$$\mathbf{p}_{next} = \left(1 + c_{\text{damp}}\right) \cdot \mathbf{p}_{now} - c_{\text{damp}} \cdot \mathbf{p}_{old} + dt^2 \cdot \vec{a}$$

Velocity damping in Verlet (geometric interpretation)



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

That is ,
$\mathbf{p}_{next}$ is an extrapolation
of $\mathbf{p}_{now}$, $\mathbf{p}_{old}$:

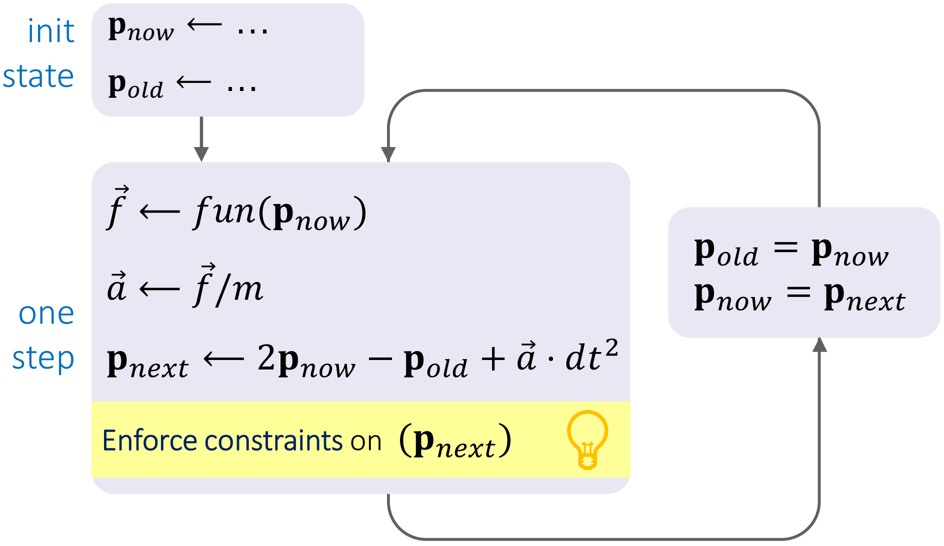
$$\mathbf{p}_{next} = mix(\ \mathbf{p}_{old},\ \mathbf{p}_{now},\ 2)$$

$$\mathbf{p}_{next} = 1.98\cdot\mathbf{p}_{now} - 0.98\cdot\mathbf{p}_{old}$$

That is ,
$\mathbf{p}_{next}$ is a different extrapolation
of $\mathbf{p}_{now}$, $\mathbf{p}_{old}$:

$$\mathbf{p}_{next} = mix(\ \mathbf{p}_{old},\ \mathbf{p}_{now},\ 1.98)$$

Verlet with PBD “Position Based Dynamics”



init state

$\mathbf{p}_{now} \leftarrow \dots$

$\mathbf{p}_{old} \leftarrow \dots$

one step

$\vec{f} \leftarrow fun(\mathbf{p}_{now})$

$\vec{a} \leftarrow \vec{f}/m$

$\mathbf{p}_{next} \leftarrow 2\mathbf{p}_{now} - \mathbf{p}_{old} + \vec{a}\cdot dt^2$

Enforce constraints on $(\mathbf{p}_{next})$

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

Position Based Dynamics

a formula
with '=' '>' '<' etc.

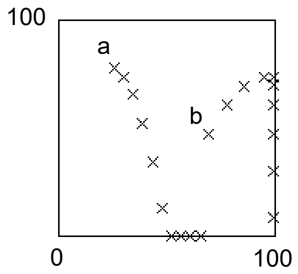
- A positional constraint is an equation/inequality involving the *positions* of particles.
  - Useful, for example, to model consistency conditions
  - Like "*solid objects don't compenetrate each other*", or "*steel bars won't bend*"
  - We will see specific examples soon
- We enforce (impose) positional constraint directly by displacing the *positions* of particles
  - Thanks to Verlet: this displacement automatically cause some appropriate update of the velocity!
  - not necessarily the correct one, but a plausible one

Verlet + Position Based Dynamics. Advantages

- flexibility: different constraints can be used to model many different phenomena
  - Useful constraints are straightforward to define
  - They are easy to impose (they involve only few particles)
  - They can be used to model many possible phenomena
  - Esamples: see following slides
- robustness : plausibility is ensured by explicitly enforced the conditions we want to see
  - For exampe: a ball won't ever be seen outside the box containing it (at lest, not for long)
- Bypasses the need to using forces / impulses to enforce the same consistency condition
  - Which is much more difficult to enforce

Example of positional constraint

«Particles must stay within [0 – 100] x [0 – 100] *»*



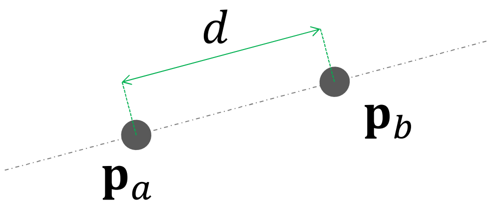
Imposing constraint: simple clamp !
ex:

```
for(int i=0; i<NUM_PARTICLES; i++)
{
  p[i].x = clamp( p[i].x, 0, 100 );
  p[i].y = clamp( p[i].y, 0, 100 );
}
```

⚠ Imposing constraints like this is a first part of collision response.
For re-bounces, impulses must still be added (see collisions).

Example of positional constraint: equidistance constraint

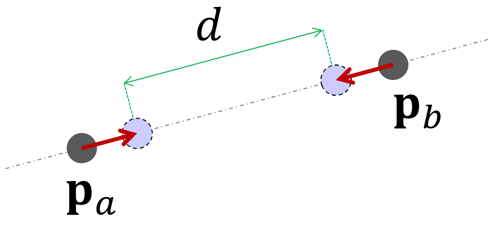
*«Particles **a** and **b** must stay at distance **d** »*



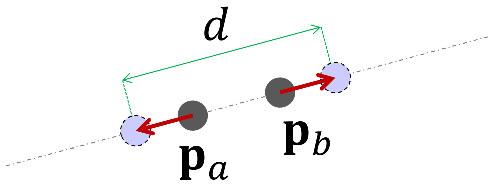
$$\|\mathbf{p}_a - \mathbf{p}_b\| = d$$

Enforce equidistance constraints

if $\|\mathbf{p}_a - \mathbf{p}_b\| > d$



if $\|\mathbf{p}_a - \mathbf{p}_b\| < d$



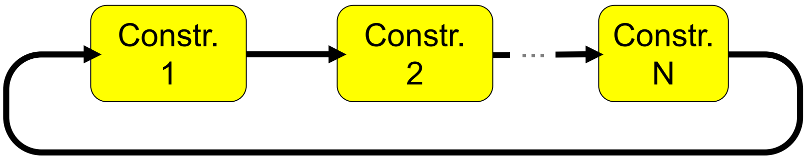
Enforce equidistance constraints: pseudo code

```
Vector3 pa, pb; // curr positions of a,b
float d;        // distance (to enforce)

Vector3 d = pa - pb;
float currDist = v.length;

d /= currDist;  // normalization of d

float delta = currDist - d ;

pa += ( 0.5 * delta) * d;
pb -= ( 0.5 * delta) * d;
```

assuming equal mass, each particle moves *half the way*
(see later for the general case)

Enforcing sets of constraints

- Many constraints to impose:
  when you solve one → you break another one!
- Simultaneous enforcement: computationally expensive
- Practical solution: enforce them in cascade
  (Gauss-Seidel fashon):



Constr. 1

Constr. 2

…

Constr. N

Repeat until convergence (= max error below threshold)
…but at most for *N* times! even 1 (remember: *soft* real time)

Enforcing sets of constraints

- Note:
  - The whole loop for imposing the constraints happen in the constraint enforcement phase on one physics step!
- Convergence:
  - if constraints are not contradictory
  - if convergence not reached (or solution doesn't exist):
    never mind, next frames will fix it (it's fairly robust)
  - needed iterations (typically): 1 ~ 10 (efficient!).
  - Optimization (to decrease number of needed iterations):
    solve the most unsatisfied constraints first

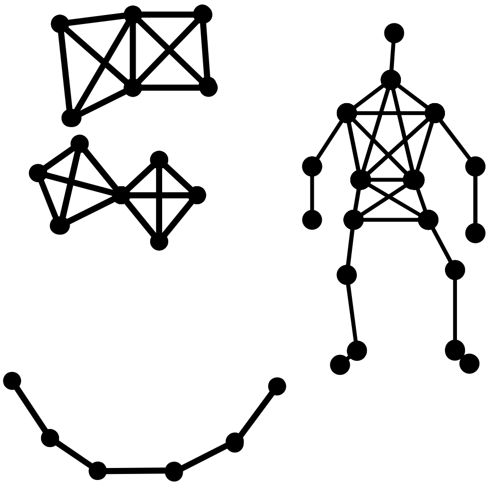
⚠ Problem: it's a sequential approach! ☹
  - parallelized versions (similar to Jacobi) are possible
  - they have a worse convergence in practice
    (they require more iterations)

Equidistance constraints VS springs

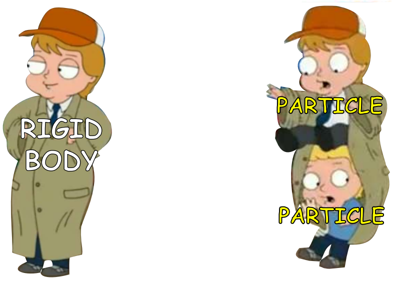
- They are similar
  - they both mean:
    these 2 particles "want to be" at *this* distance (not more, not less)
- Differences:
  - equidistance constraint:
    - applied during constraint enforcement
      - directly affecting positions
    - models a rigid rod between the two particles
      - of a given length
    - sometimes called an "HARD" constraint
  - spring:
    - applied during force evaluation step
      - affecting forces, therefore accelerations
    - models a deformable spring between the two particles
      - of a given length
    - sometimes called a "SOFT" constraint
- A physic engine can combine them in one object!

We can combine equidistance constraints to obtain…

- Rigid bodies
- Articulated bodies
- Ragdolls
- Cloth
- Non-elastic ropes
- And more

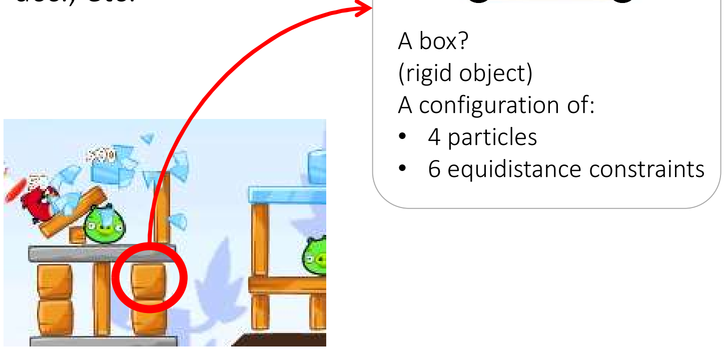


Compounds of particles disguised as rigid bodies

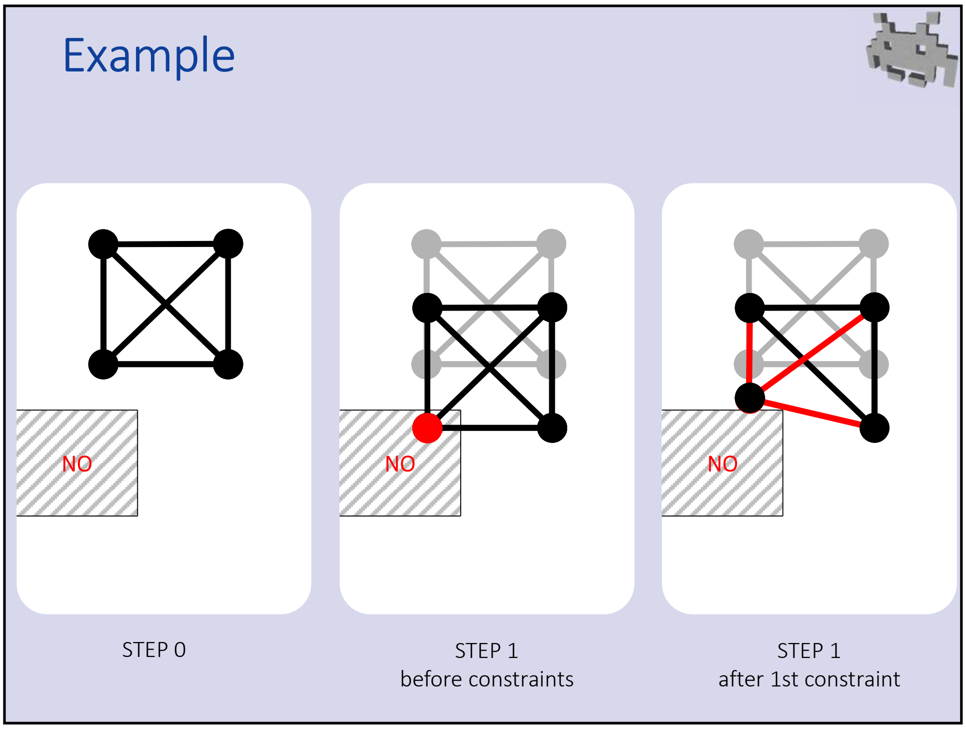


Combining equidistance constraints we obtain rigid objects

- Rigid body dynamics as emerging behavior
  - without explicitly updating their orientation, angular vel, angular acc., etc.



A box?
(rigid object)
A configuration of:
- 4 particles
- 6 equidistance constraints

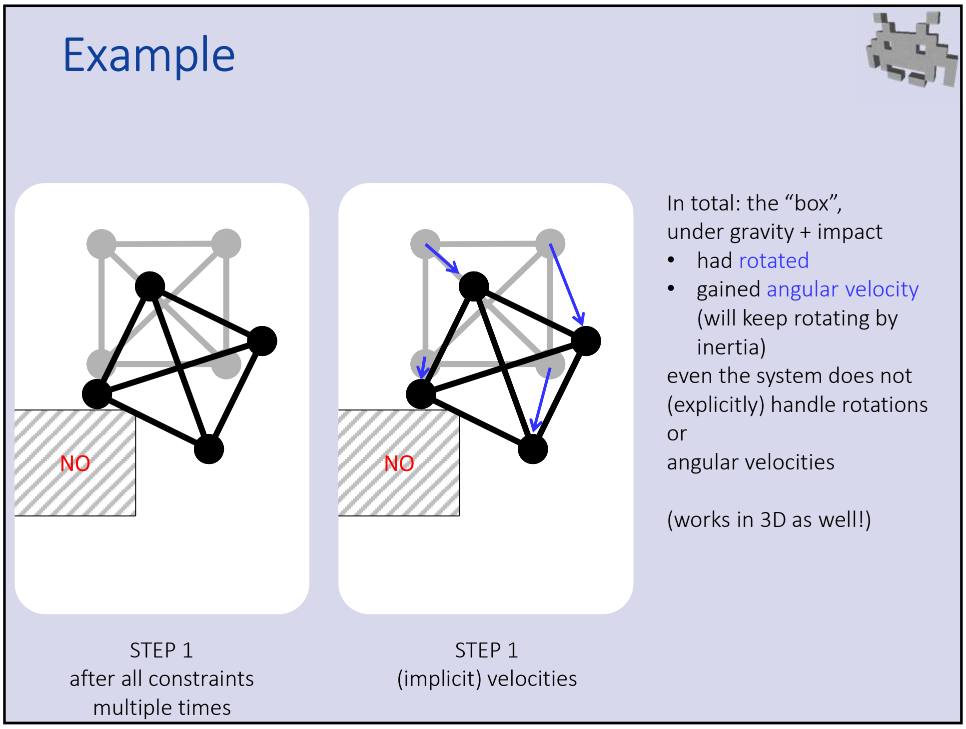


Example

STEP 0

STEP 1
before constraints

STEP 1
after 1st constraint



Example

STEP 1
after all constraints
multiple times

STEP 1
(implicit) velocities

In total: the “box”,
under gravity + impact

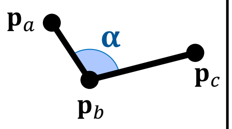
- had rotated
- gained angular velocity (will keep rotating by inertia)

even the system does not (explicitly) handle rotations or
angular velocities

(works in 3D as well!)

More examples of positional constraints

- Preserve volume of some object: *«Volume is* v_C *»*
  - How to impose it:
    1. Estimate current total volume v
    2. uniform scaling of the entire object of $\sqrt[3]{v_C / v}$
- Fixed positions: *«particle* a *stays in* $\mathbf{p}_\mathrm{a}$ *»*
  - particles «pinned in position»
  - trivial to impose, but useful!
- Angle constraints, e.g. $\boldsymbol{\alpha} < \boldsymbol{\alpha}_\mathrm{max}$
  - e.g. on joints: *«elbows cannot bend backward»*



- Coplanarity / collinearity
- Non interpenetration
  - this is part of collision handling – see collisions later

Enforcing a positional constraint: the general case.

- Check: does the equation/inequality hold?
- If so, nothing to do!
- Else:
  - All positions must be displaced a bit so that it does
  - Infinite ways to achieve this. Which one to pick?
  - Answer:
    minimize the sum of *squared* displacements
    (with respect to current position)
    weighted by particle masses
  - Find minimizer by analytically solving simple problems
    (in closed form, “analytically” = “on paper”)

Enforcing a positional constraint the general case: formally problem

- We want to enforce a constraint $\mathcal{C}$ on particles a , b , c,…
  with have mass m_a, m_b, m_c …
- $\mathcal{C}$ defined as an equation/inequality of their positions $\mathbf{p}_a$, $\mathbf{p}_b$, $\mathbf{p}_c$, …
- We must apply the displacements $\vec{d_a}$, $\vec{d_b}$, $\vec{d_c}$
  which minimize:

$$\underset{\vec{d_a},\,\vec{d_b},\,\vec{d_c},\ldots}{\operatorname{argmin}} \left(m_a \left\| \vec{d_a} \right\|^2 + m_b \left\| \vec{d_b} \right\|^2 + m_c \left\| \vec{d_c} \right\|^2 + \cdots \right)$$

$$\text{such that } \mathcal{C}\left(\mathbf{p}_a + \vec{d_a} \;, \mathbf{p}_b + \vec{d_b} \;, \mathbf{p}_c + \vec{d_c} \;, \ldots \right)$$

among all the choices that satisfy this,

we want the one which minimizes this

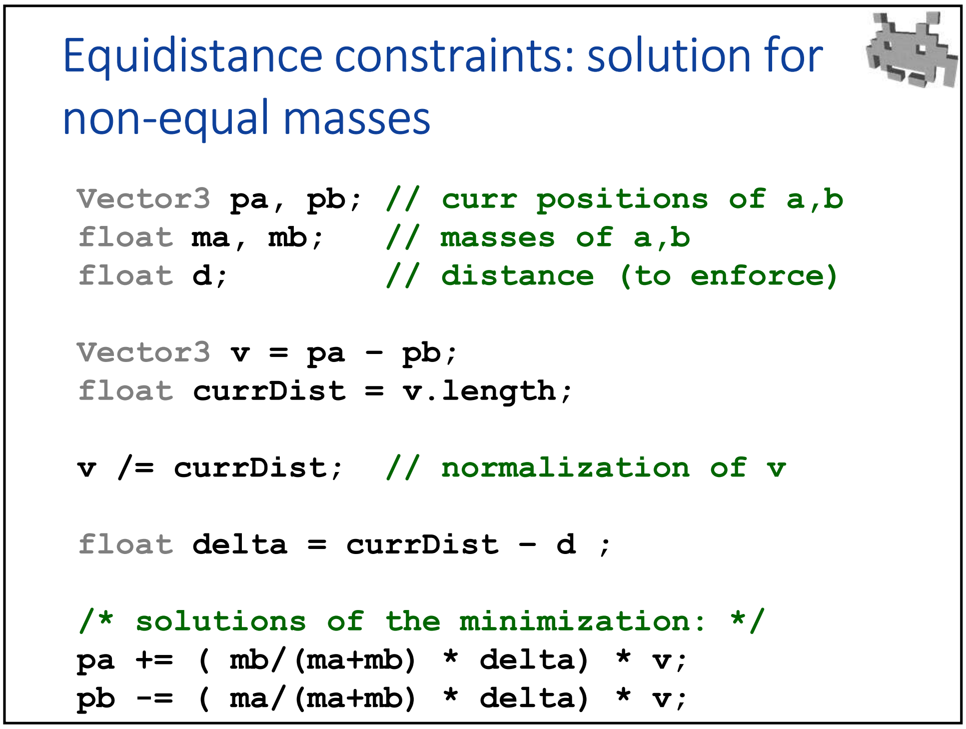
Enforcing positional constraint Example: equidistance constraint

- To enforce the constraint
  "particles a and b must stay at distance k"
  - input: current positions $\mathbf{p}_a$, $\mathbf{p}_b$
  - input: masses m_a, m_b
- We need to the the displacements $\vec{d_a}$, $\vec{d_b}$
  found by minimizing:

$$\underset{\vec{d_a},\,\vec{d_b}}{\operatorname{argmin}} \left(\mathrm{m}_a \left\| \vec{d_a} \right\|^2 + \mathrm{m}_b \left\| \vec{d_b} \right\|^2 \right)$$

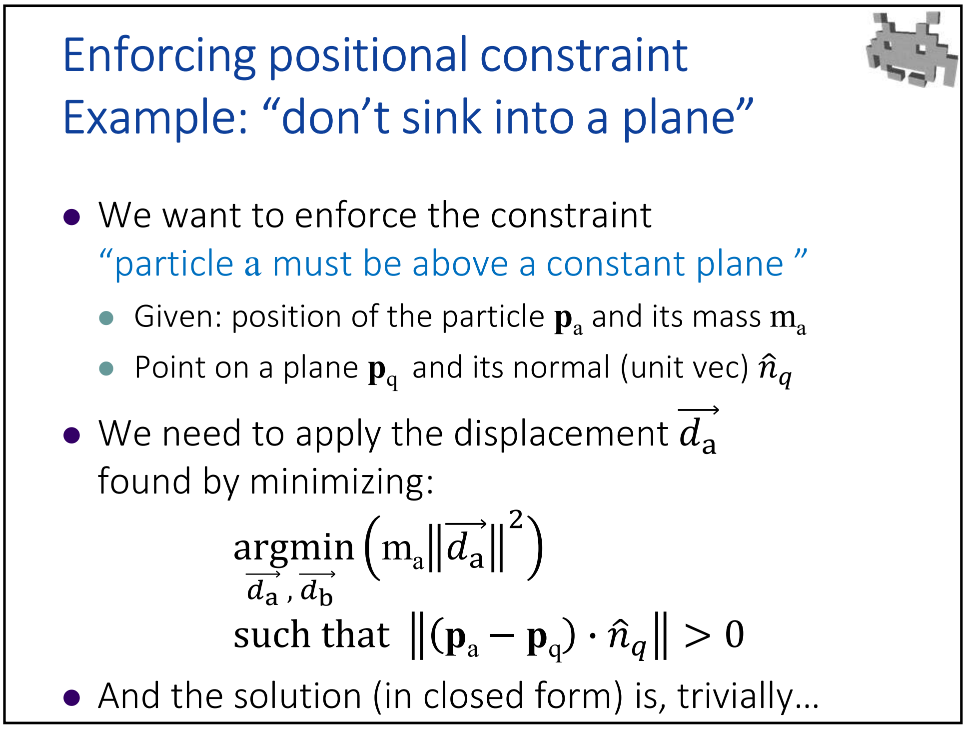
$$\text{such that } \left\| \left(\mathbf{p}_a + \vec{d_a} \right) - \left(\mathbf{p}_b + \vec{d_b} \right) \right\| = k$$

- And the solution (in closed form) is…



Equidistance constraints: solution for non-equal masses

```
Vector3 pa, pb; // curr positions of a,b
float ma, mb;   // masses of a,b
float d;        // distance (to enforce)

Vector3 v = pa - pb;
float currDist = v.length;

v /= currDist;  // normalization of v

float delta = currDist - d ;

/* solutions of the minimization: */
pa += ( mb/(ma+mb) * delta) * v;
pb -= ( ma/(ma+mb) * delta) * v;
```



Enforcing positional constraint Example: “don’t sink into a plane”

- We want to enforce the constraint
  “particle a must be above a constant plane ”
  - Given: position of the particle $\mathbf{p}_\mathrm{a}$ and its mass m_a
  - Point on a plane $\mathbf{p}_\mathrm{q}$ and its normal (unit vec) $\hat{n}_q$
- We need to apply the displacement $\overrightarrow{d_\mathrm{a}}$ found by minimizing:

$$\underset{\overrightarrow{d_\mathrm{a}},\,\overrightarrow{d_\mathrm{b}}}{\operatorname{argmin}} \left(\mathrm{m}_\mathrm{a} \left\| \overrightarrow{d_\mathrm{a}} \right\|^2 \right)$$

$$\text{such that } \left\| (\mathbf{p}_\mathrm{a} - \mathbf{p}_\mathrm{q}) \cdot \hat{n}_q \right\| > 0$$

- And the solution (in closed form) is, trivially…

In pseudocode

```
Vector3 pa; // curr positions of a
float ma;   // mass (no effect here)
Vector3 pq; // point on the plane
Vector3 nq; // normal of the plane (unit)

Vector3 v = pa – pq;
float currDist = Vector3.dot( v , n );

if (currDist < 0.0)
   pa -= currDist * n; // just project!
else {} // constrain holds, do nothing
```

Rigid objects as compounds of constrained particles: advantages

- Interesting/rich/useful set of “emerging behaviors” (i.e. effects with “just automatically happens”) :
  - rigid, deformable, jointed objects
    - made of particles + hard constraints
  - their angular velocities
    - rotation around proper axis
  - their barycenter
  - their momentum of inertia
    - angular velocity is maintained
  - somewhat believable bounces on “impacts”
    - for more control: impact impulses can be added (see collisions)

you don’t need to compute or store these

consequence of constraints disallowing compenetration

Particles + constraint, or rigid bodies?

- Rigid-body based systems:
  - explicitly compute dynamics for rigid bodies
  - updating their rotation, angular speed,…
- Particles-based systems:
  - only compute dynamics for particles
  - rigid (or deformable, or jointed) bodies as an emerging behavior
- Mixed systems:
  - use both
  - may even dynamically swap between the two representations for rigid bodies

Rigid body as particles + constraints: Challenges

- Approximations are introduced
  - e.g.: mass is concentrated in a few locations
- Scalability issues
  - many constraints to enforce, many particles to track
- Some of the info which is kept *implicit* is needed by the rest of the game engine
  - and must therefore be extracted ☹
  - example: the transform (position + orientation) of the “rigid body” is needed to render the associated mesh
  - similarly: angular speed, barycenter pos, velocity…