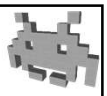
Course Plan



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. 9: Game **Materials**
lec. 8: Game **3D Animations**
lec. 10: **3D Audio** for 3D Games
lec. 11: **Networking** for 3D Games
lec. 12: **Artificial Intelligence** for 3D Games
lec. 13: **Rendering Techniques** for 3D Games

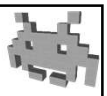
90

Forces (continued): control forces

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

91

Forces



- Remember all forces acting on a particle add up! (vector summatory)
- The resulting force is what counts

one step

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

$t = t + dt$

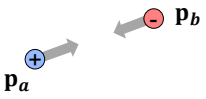
93

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

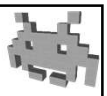
some global constant

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

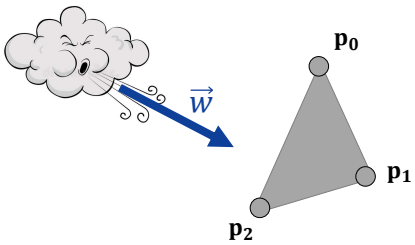


94

Example of forces: wind pressure



- Wind is a force acting on surfaces
- The larger the exposed surface to the wind, the STRONGER / MORE INTENSE 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



area vector

$$\vec{f} = \left\| \frac{1}{2} (\mathbf{p_1} - \mathbf{p_0}) \times (\mathbf{p_2} - \mathbf{p_0}) \cdot \vec{w} \right\| \frac{\vec{w}}{\|\vec{w}\|}$$

force magnitude (scalar)

force direction (versor)

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

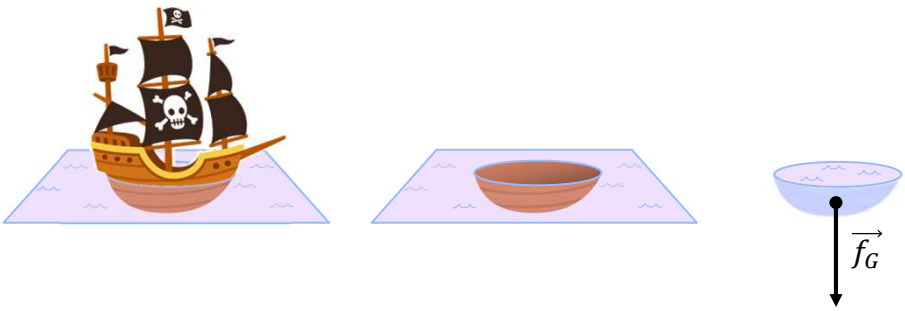
95

Example of forces: buoyancy

- Opposite of gravity force $\overrightarrow{f_G}$
  - of the submerged part... if it was made of water
  - mass of the submerged part = its volume *times* density of water

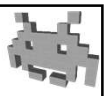
or whichever liquid

mass/volume
aka "specific mass"



96

Attrition (or friction) forces



- *Isotropic* friction forces :
  - a force that opposes 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 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)
  - Part of the collision response (see next topic!)

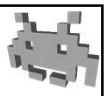
97

Attrition (or friction) forces:
simulate them with *velocity damping* !

- A useful trick to simulate isotropic friction:
  “velocity damping”
  - simply reduce all velocity vectors by a fixed proportion
  - for example: scale velocity down by 2% per second
    (“drag factor” = 0.02 / sec)
    (that is, scale velocity vectors by a factor 0.98)
  - Why it makes sense:
    Higher speed = more attrition = more loss of speed.
    So, attrition = a “fixed tax” (in %) on velocity.
- For planar friction:
  - Split velocity into parallel / orthogonal parts
  - Apply different Drag factors to each parts

98

Velocity Damping: how to (as an example)



- Objective: "reduce speed by 1.5% every second"
  - Drag factor: 0.015
- So:
  - After a second: $\vec{v} \leftarrow (1.0 - 0.015)\,\vec{v}$
  - After 2 seconds? $\vec{v} \leftarrow (1.0 - 0.015)^2\,\vec{v}$
  - After k seconds? $\vec{v} \leftarrow (1.0 - 0.015)^k\,\vec{v}$
  - After dt seconds? $\vec{v} \leftarrow (1.0 - 0.015)^{dt}\,\vec{v}$
    - e.g. 1/60 = 0.17 sec
  - Which can be approximated with $\vec{v} \leftarrow (1.0 - 0.015 \cdot dt)\vec{v}$
  - The approximation is good when *this* is small

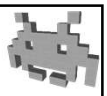
99

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 );
}
```

100

Velocity Damping: notes



- Velocity Damping is useful for robustness
  - Prevents the energy to ever increase
- Limitations:
  - it may exaggerate frictions of, e.g., air, especially in absence of contacts
  - it's a crude approximation: attrition forces are not really *linear* with speed
- In practice:
  - low drag: hardly noticeable (in the short run), increases robustness
  - high drag (e.g. 2% per sec): everything feels like to be moving in molasses (ita: *melassa*); everything quickly grinds to a halt
  - super high drag: (e.g. >25% per sec) basically, no inertia anymore. May be useful to converge to (local) minimal energy states: your simulator is basically a solver for statics, not dynamics

101

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:
  - for positions: "teleports"
  - for velocity: "impulses"
  - In the real world, those variations can well be consequences of forces, but these forces are not modelled as such, in our system

102

Dynamics displacements VS kinematic

a **discontinuous** change of state (position)

…
$\mathrm{p} = \mathrm{p} + \vec{\mathrm{v}} \cdot dt$
…

aka dynamic displacements

Justified by physics

…
$\mathrm{p} = \mathrm{p} + d\mathrm{p}$
…

aka Kinematic displacements

Just "teleportation"

103

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

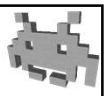
…
$\vec{v} = \vec{v} + \left(\vec{i} / m\right)$
…

- Impulses
  - Infinitesimal time
  - una tantum

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

104

Impulses VS Forces



- Force :
  - it determines an acceleration
  - acc determines a (continuous!) change of vel
  - it's sustained over meaningful periods of time
- Impulse :
  - a (discontinuous!) change of vel
  - in reality: just a force with
    - very large magnitude
    - very short application times
  - we model it as a force "pre-multiplied" with its application time
  - it's useful to:
    - model phenomena with a time scale << *dt*
      e.g. a tennis ball rebounding against a tennis racket
    - control the simulation (direct change of velocity)

105

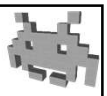
Impulses VS Forces

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

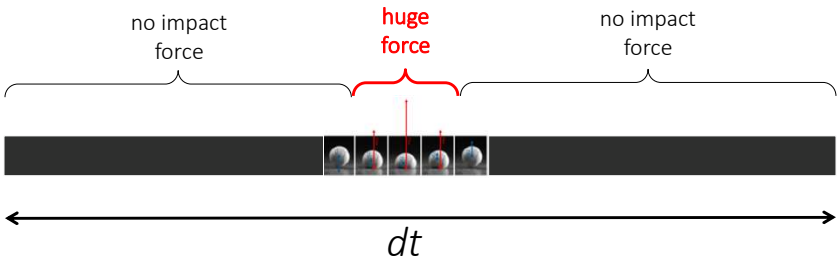
- very strong forces (but not infinite)
- applied for a very short time (but not instantaneous)
- see *collision response* later for details
  about the impulse based approximations

107

Impulses VS Forces



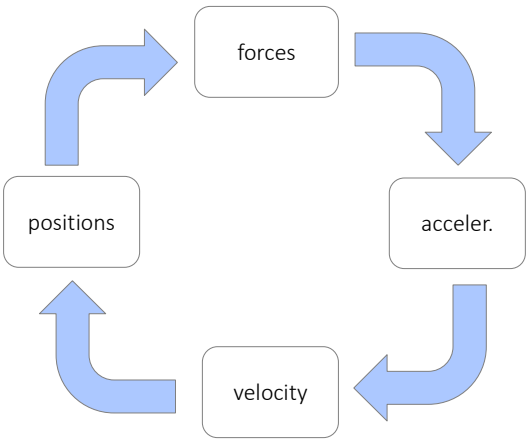
- what does *truly* happen when it bounces off the ground?



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

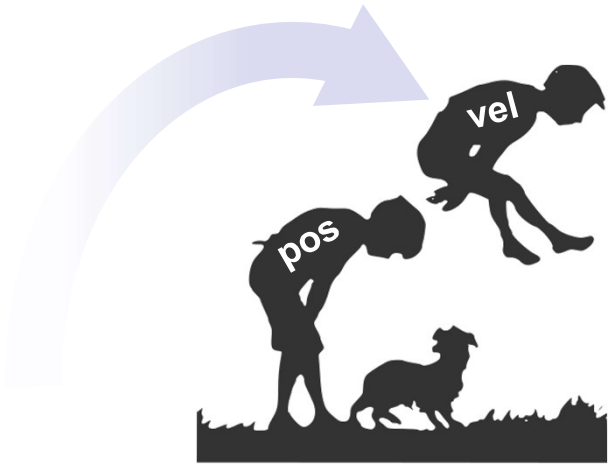
108

Next: better integration methods for (Newtonian) dynamics



109

Leapfrog Integration Method



110

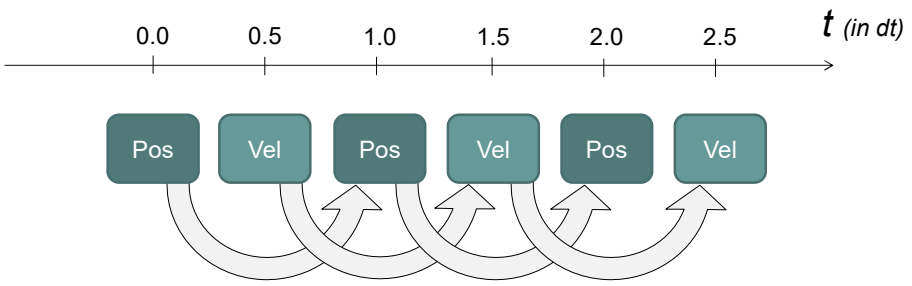
Leapfrog Integration Method

- Basic Idea:
  store positions at time $k \cdot dt$
  (that is, at t = 0, 1dt, 2dt, 3dt…)
  but store velocities at time $k \cdot dt + \frac{1}{2}\, dt$
  (that is, at t = 0.5dt, 1.5dt, 2.5dt, 3.5dt…)

- Equivalent to use a summatory of
  the areas of trapezoids,
  (having base dt and height $v(t)$)
  not rectangles, to compute the integral

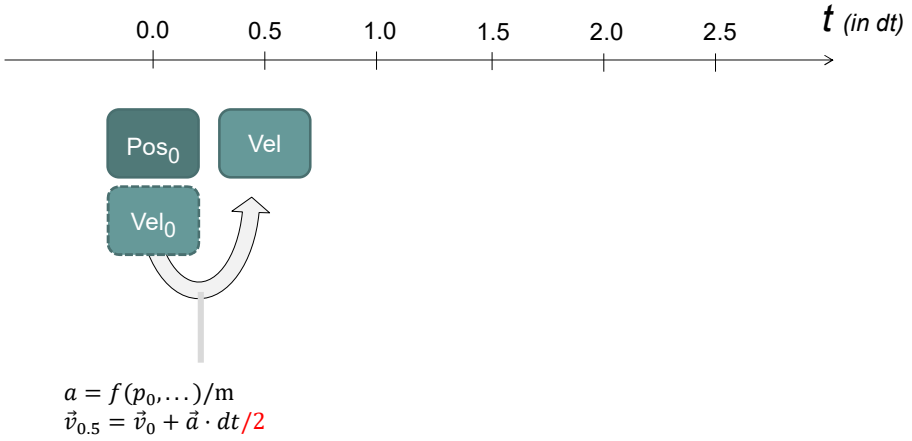
111

Leapfrog Integration Method



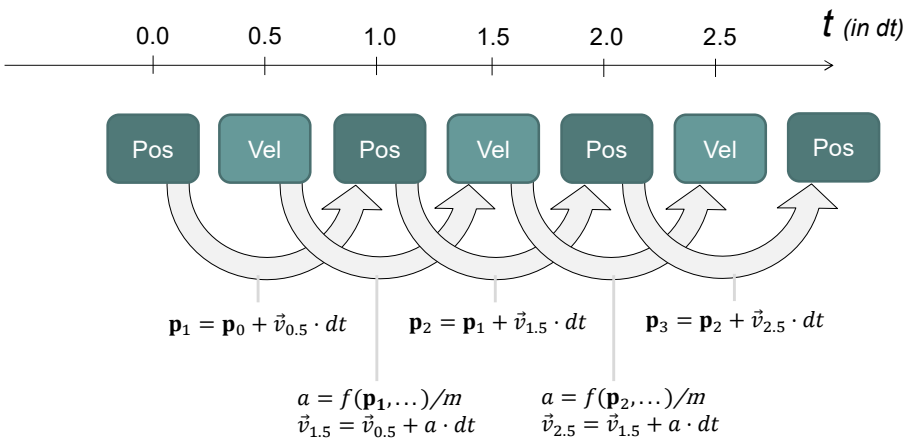
112

Leapfrog Integration first step



113

Leapfrog Integration



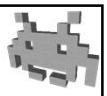
114

Leapfrog method: pros and cons

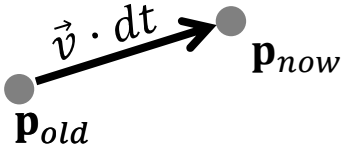
- Same cost as Euler – and basically same code
  - Velocity stored in status = velocity "half a *dt* ago" (and after updating it: "half a frame in the future")
  - Only real difference: the initialization of velocities
- Better theorical accuracy, for the same *dt*
  - better asymptotic behavior: it's a "second order" system 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 of numerical errors.
- But: requires fixed *dt* during all the simulation
  - Otherwise, updates of vel required in all particles

115

Verlet integration method



- Idea: remove velocity from state
  Instead, store previous position
- Velocity is now implicit
- It's defined by:
  - 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

116

Verlet integration method: (modifying Euler integration…)

init state

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

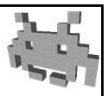
one step

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

expanding this…

117

Verlet integration method (modifying Euler integration…)



init state

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

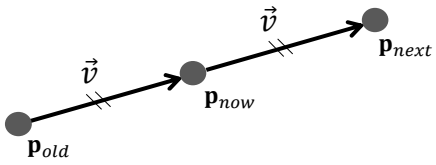
one step

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

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

118

Verlet integration method: geometric interpretation



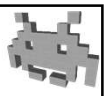
$\mathbf{p}_{next} = 2 \cdot \mathbf{p}_{now} - 1 \cdot \mathbf{p}_{old}$

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

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

119

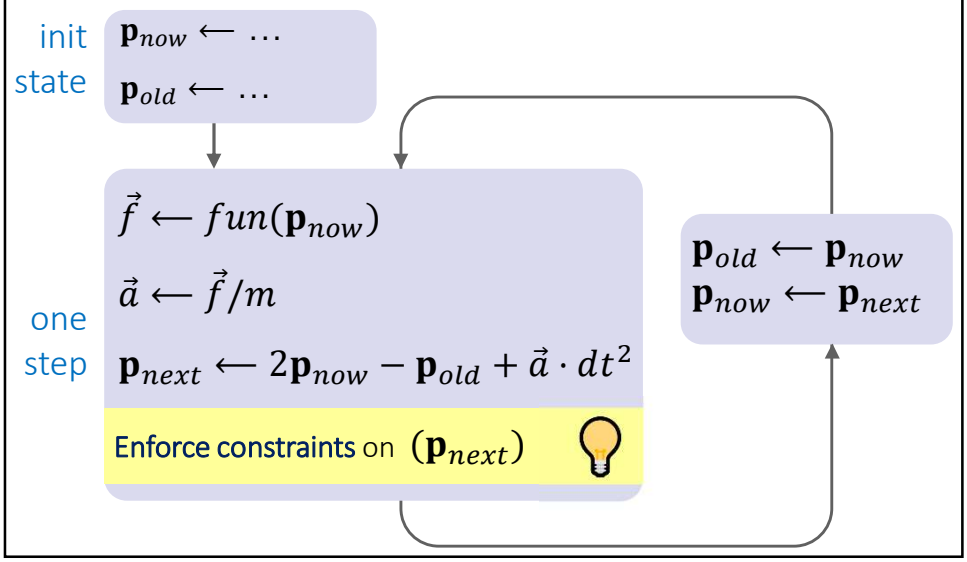
Verlet: characteristics



- 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
  - 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
    * (just swap p_now and p_old)
  - only in theory… careful with implementation details

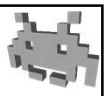
120

Verlet integration + "Position Based Dynamics" (PBD)



121

Position Based Dynamics (PDB)



- A positional constraint is an equality/inequality (a formula with '=' '>' '<' etc.) 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 become shorter or longer than they are*"
  - We will see many examples

We enforce (impose) positional constraint directly by displacing the *positions* of particles

- Thanks to Verlet: this displacement automatically causes some appropriate update of the velocity!
- it's not necessarily correct, but it's plausible and robust

122

Example of a positional constraint

«I want all particles to stay above ground (that is, their y must never be negative) »

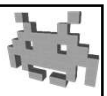
Enforce this constraint: trivial!

```
for (each particle i)
{
  if (p[i].y < 0) p[i].y = 0;
}
```

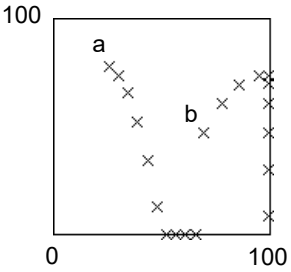
Imposing constraints like this one is a first part of collision response. For re-bounces, impulses must still be added (see collisions).

123

Example of a positional constraint (here, in 2D physics)



«I want particles to stay inside a 2D box [0 – 100] x [0 – 100] *»*



Enforce this constraint: simple clamp!

```
for (each particle i)
{
  p[i].x = clamp( p[i].x, 0, 100 );
  p[i].y = clamp( p[i].y, 0, 100 );
}
```

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

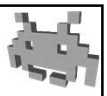
124

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
  - See following slides for examples
- robustness : plausibility is ensured by *explicitly* enforcing the conditions we want to see
  - For example: a ball won't ever be seen outside the box containing it – and it will also recover from mistakes
- No forces / impulses are needed to enforce any such consistency conditions

125

Verlet: *caveats*
(see next slides 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?

126

Changing the value of dt in Verlet
(whenever it's not constant, for any reason)

Problem:

if dt now changes to a new dt'

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

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

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

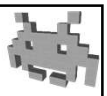
current velocity $\vec{v}$ and position $\mathbf{p}_{now}$ must *not* change, so we can only change $\mathbf{p}_{old}$

$$\Longrightarrow$$

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

127

Velocity damping in Verlet



implicit

- Velocity at next frame: $\vec{v} = (\mathbf{p}_{next} - \mathbf{p}_{now})/dt$
- We want to multiply $\vec{v}$ a factor c_{DAMP}
  - before applying accelerations

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

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

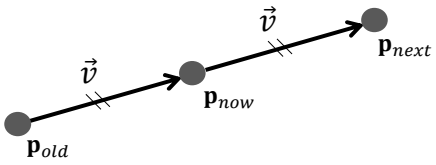
$$\mathbf{p}_{next} = 2 \cdot \mathbf{p}_{now} - 1 \cdot \mathbf{p}_{old} + dt^2 \cdot \vec{a}$$

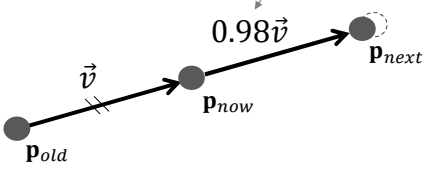
$$\mathbf{p}_{next} = (1 + c_{\text{DAMP}}) \cdot \mathbf{p}_{now} - c_{\text{damp}} \cdot \mathbf{p}_{old} + dt^2 \cdot \vec{a}$$

129

Velocity damping in Verlet (geometric interpretation)

a bit shorter





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

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

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

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

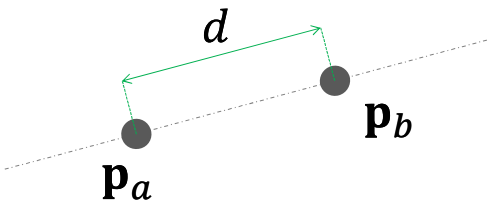
Equivalently,
$\mathbf{p}_{next}$ is a different extrapolation of $\mathbf{p}_{now}$, $\mathbf{p}_{old}$:

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

130

Example of positional constraint: equidistance constraint

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

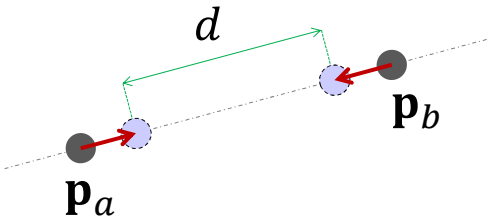


I want that... $\|\mathbf{p}_a - \mathbf{p}_b\| = d$

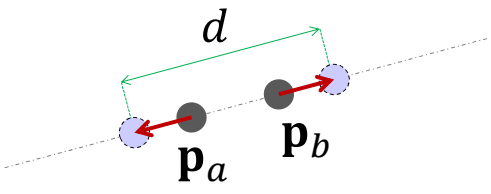
131

Enforce equidistance constraints
(assuming equal masses for now)

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

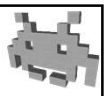


if $\|\mathbf{p}_a - \mathbf{p}_b\| < d$



132

Enforce equidistance constraints: pseudo code



```
Vector3 pa, pb; // curr positions 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 ;

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

we move each particle *half the way*
(it makes sense, but see later for why exactly)

133

Compare: equidistance constraints vs. springs

- Similar
  - they both mean:
    these 2 particles "want to be" at *this* distance (not more, not less)

some constant scalar parameter D

- but different
  - equidistance constraint:
    - applied during constraint enforcement
    - directly affects positions
    - models a rigid rod between the two particles
      - of a given length
    - sometimes called a "HARD" constraint
  - spring:
    - applied during force evaluation step
    - affects forces, therefore accelerations
    - models a deformable spring between the two particles
      - of a given length
    - sometimes called a "SOFT" constraint
- A physic engine can use both at the same time!

134

How to enforce a positional constraiont?
(see next lecture for the complete anwser)

When enforcing constraints…

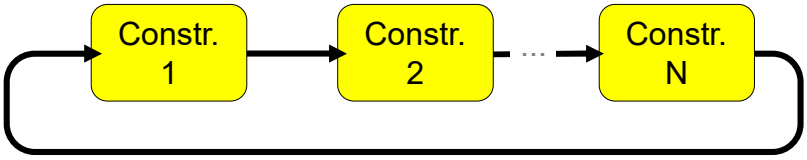
- If a constraint is valid, no problem.
- If it's not, change particle positions so that it does.
  - Problems: there's usually many possible ways to do

⚠ Which one to pick?

135

Enforcing a set of constraints

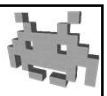
- There are many constraints to impose:
  when you solve one → maybe you break another!
- Simultaneous enforcement: computationally expensive
- Practical & easy solution: just enforce them in cascade
  (similar in concept to Gauss-Seidel solvers):



Repeat until convergence (= max error below threshold)
…but at most for N_{MAX} times! (remember: our simulation is *soft* real-time)

136

Enforcing a set of constraints one after the other (in cascade)



- The whole loop for imposing the constraints happen in the constraint enforcement phase on one physics step
- Notes about convergence:
  - needed iterations (typically) few: e.g. 1 ~ 10 (efficient!).
  - if convergence not reached within a given number of steps: never mind, next frames will fix it (so it's robust)
  - (it is never reached, if constraints are contradictory)
  - Optimization (to reduce the number of needed iterations): solve the most unsatisfied constraints first

⚠ Problem: it's a sequential approach! ☹

- parallelized versions (similar to Jacobi solvers) are possible
- they have a worse convergence in practice (they require more iterations), but each iteration is faster

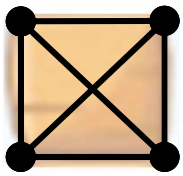
137

Compounds of particles disguised as rigid bodies

138

We can combine equidistance constraints to obtain rigid objects!

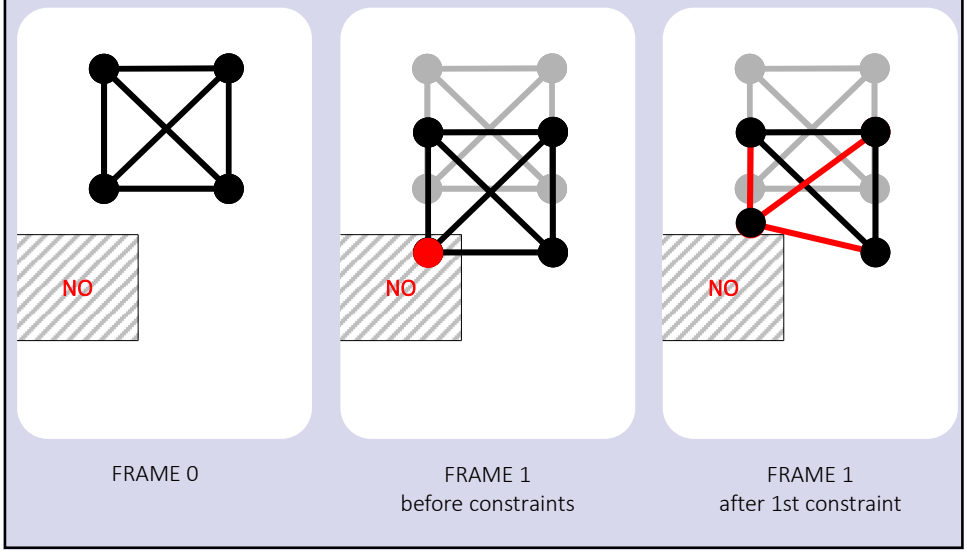
- Rigid body dynamics as emerging behavior
  - without explicitly keeping track their orientation, angular vel, angular acc., etc.



A box in 2D?
(rigid object)
A configuration of:
- 4 particles
- 6 equidistance constraints

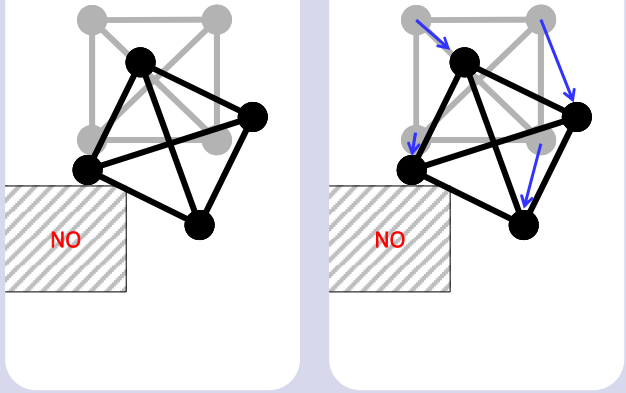
139

Example



140

Example



FRAME 1
after all constraints
multiple times

FRAME 1
resulting
(implicit) velocities

In total: the "box",
under gravity + collision
- has rotated
- has gained angular velocity (and will keep rotating by inertia)

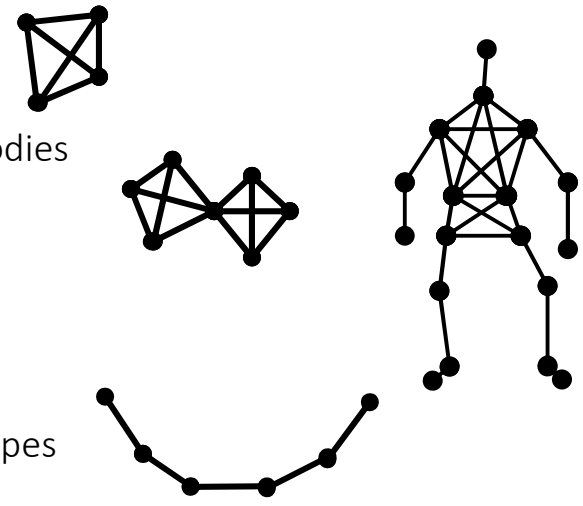
…even the system does not (explicitly) handle rigid-bodies (and they rotations, angular velocities, or angular momentum)

(works in 3D as well!)

141

We can combine equidistance constraints to obtain…

- Rigid bodies
- Articulated bodies
- Ragdolls
- Cloth
- Non-elastic ropes
- …and more



143