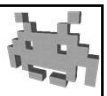
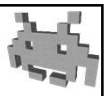
Course Plan



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 Materials
lec. 9: Game 3D Animations
lec. 10: 3D Audio for 3D Games
lec. 11: Networking for 3D Games
lec. 12: Interactive Agents for 3D Games
lec. 13: Rendering Techniques for 3D Games

87

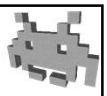
Attrition (or friction) forces



- *Isotropic* friction force : $\vec{f} = -k\,\vec{v}$
  - a force that opposes motion, dissipating energy
  - isotropic = magnitude independent on the direction of motion
  - direction of force: always opposite of current velocity direction
  - Magnitude of force: proportional to the speed = $||\vec{v}||$
  - note: this force depends on velocity, not position(s).
  - models the resistence of the medium where the motion happens (air, water, thin space…)
  - the denser the medium, the largest k, 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!)

88

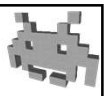
Attrition (or friction) forces: 💡simulate them with *velocity damping* !



- A simpler way to simulate (isotropic) friction: "velocity damping" (or "viscous 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

89

Velocity Damping: how to (as an example)

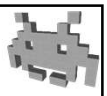


- Objective: "reduce speed by 1.5% every second"
  Drag factor: 0.015
- So:
  - After a second: $\vec{v} \leftarrow (1.0 - 0.015)\,\vec{v}$
  - After 2 seconds? $\vec{v} \leftarrow (1.0 - 0.015)^2\,\vec{v}$
  - After k seconds? $\vec{v} \leftarrow (1.0 - 0.015)^k\,\vec{v}$
  - After dt seconds? $\vec{v} \leftarrow (1.0 - 0.015)^{dt}\,\vec{v}$

    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

90

Equivalent formulas for friction
(controlled by different constants k_0 , k_1 , k_2)



- $\vec{f} = -k_0\vec{v}$ => $\vec{a} = -\frac{k_0}{m}\vec{v}$ => $\vec{v} \leftarrow \vec{v} - \frac{k_0}{m}\vec{v}\,dt$ =>

  update: $\vec{v} \leftarrow \left(1 - \frac{k_0}{m}\right)^{dt}\vec{v}$
- update: $\vec{v} \leftarrow (1 - k_1)^{dt}\vec{v}$
- update: $\vec{v} \leftarrow e^{-k_2 \cdot dt}\vec{v}$

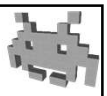
91

Equivalent formulas for friction
(controlled by different constants k_0 , k_1 , k_2) - notes

- 1st way: k_0 is the ratio force magnitude / speed.
  Physically based, and auto-adapts with mass,
  but difficult to tune in your game?
- 2nd way: k_1 (i.e., $\frac{k_0}{m}$) is the velocity loss per second (%).
  Intuitive to set! (here, we will adopt this way).
- 3rd way: k_2 (i.e., $-\log(1 - k_1)$)
  is the exponential decay.
  Also quite used in videogames
  (e.g., in Unity, Godot, Unreal)

92

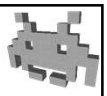
Velocity Damping: pseudo-code



```
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 );
}
```

93

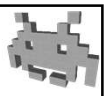
Velocity Damping: notes



- 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 now basically a solver for statics, not dynamics

94

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

95

Dynamics displacements VS kinematic

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

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

aka dynamic displacements

Justified by physics

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

aka Kinematic displacements

Just "teleportation"

96

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

…

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

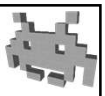
…

- Impulses
  - Infinitesimal time
  - una tantum

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

97

Impulses VS Forces

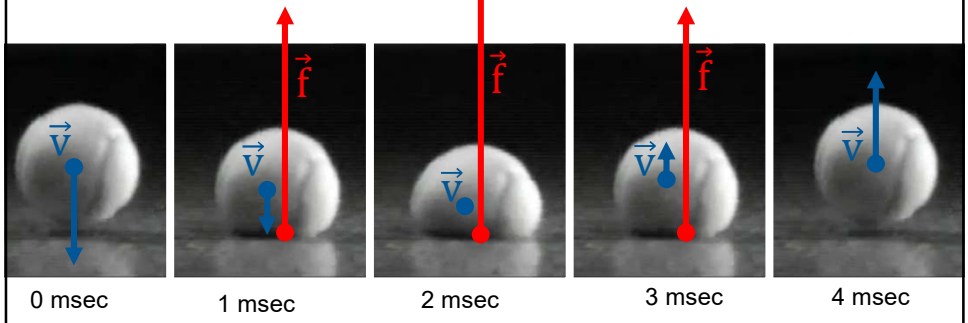


- Force :
  - it determines an acceleration
  - acc determines a (continuous!) change of vel
  - it's sustained over meaningful periods of time
- Impulse :
  - a (discontinuous!) change of vel
  - in reality: just a force with
    - very large magnitude
    - very short application times
  - we model 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)

98

Impulses VS Forces

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

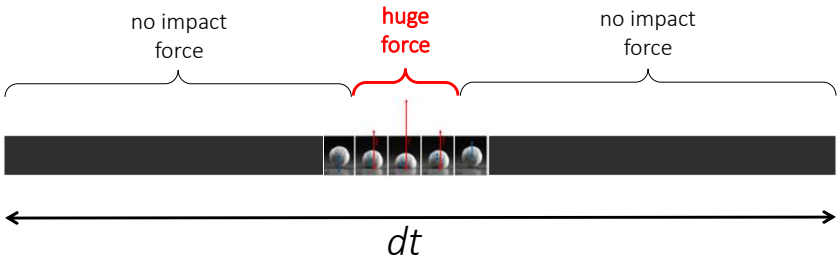


- very strong forces (but not infinite)
- applied for a very short time (but not instantaneous)
- see *collision response* later for details about the impulse based approximations

100

Impulses VS Forces

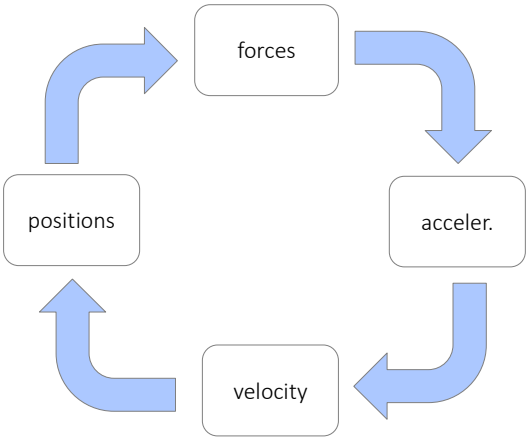
- what does *truly* happen when it bounces off the ground?



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

101

Next: better integration methods for (Newtonian) dynamics



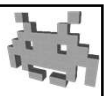
102

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)

103

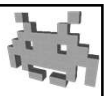
Forward Euler Method: limitations

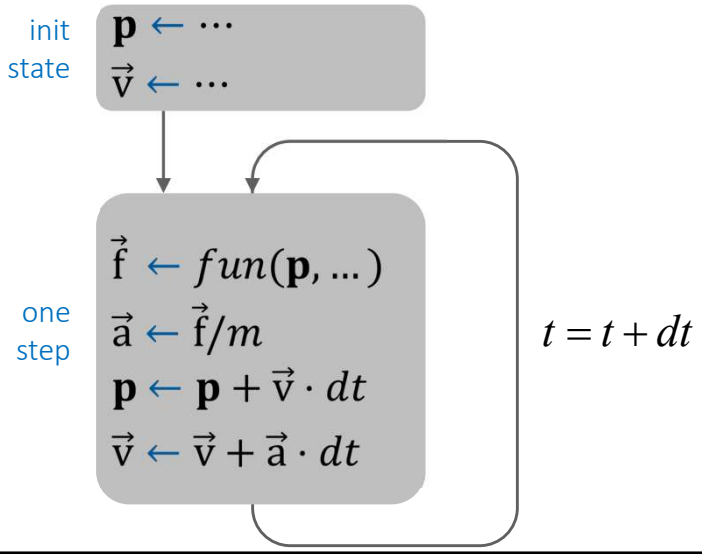


- efficiency / accuracy: not too good
  - error accumulated over time = linear in dt
  - it's only a "first order" method
  - Doubles the steps = halve the *dt* , only halves the errors (can be better, but no guarantees)
- scarce stability for large *dt*
- minor problem: no reversibility, *even in theory*
  - 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

104

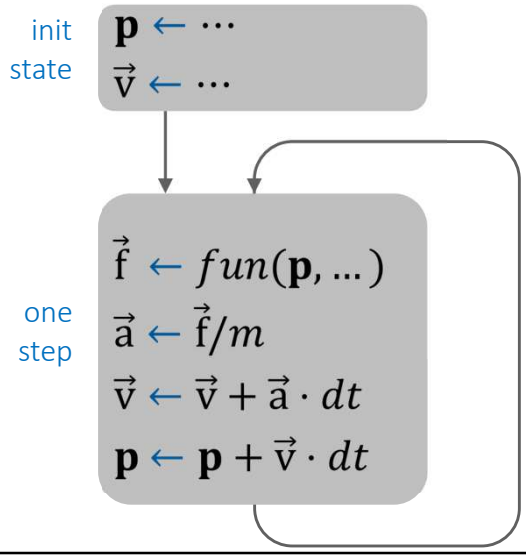
Forward Euler





106

Symplectic Euler



107

Forward Euler *pseudo code*

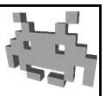
```
Vec3 position = …
Vec3 velocity = …

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…

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

108

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



Equivalent to...

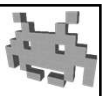
$$\vec{f}_i \leftarrow function(p_i, \ldots)$$
$$\vec{a}_i \leftarrow \vec{f}/m$$
$$\vec{v}_{i+1} \leftarrow \vec{v}_i + \vec{a}_i \cdot dt$$
$$p_{i+1} \leftarrow p_i + \vec{v}_{i+1} \cdot dt$$

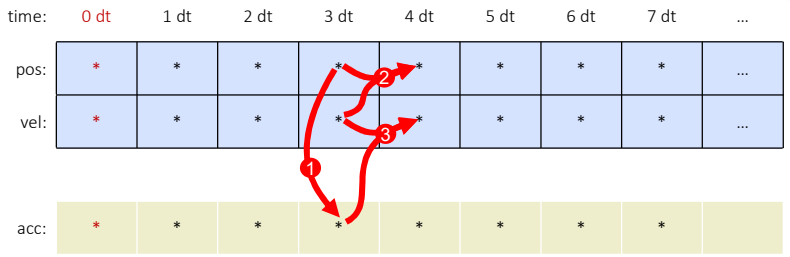
```
Vec3 position = …
Vec3 velocity = …

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

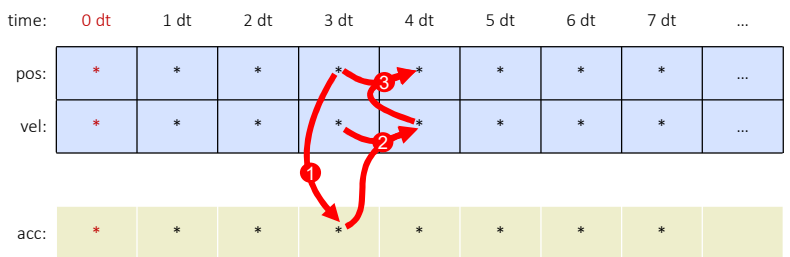
109

Forward Euler:



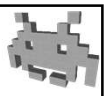


Symplectic Euler:



110

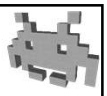
Forward Euler VS Symplectic Euler (warning: over-simplifications)

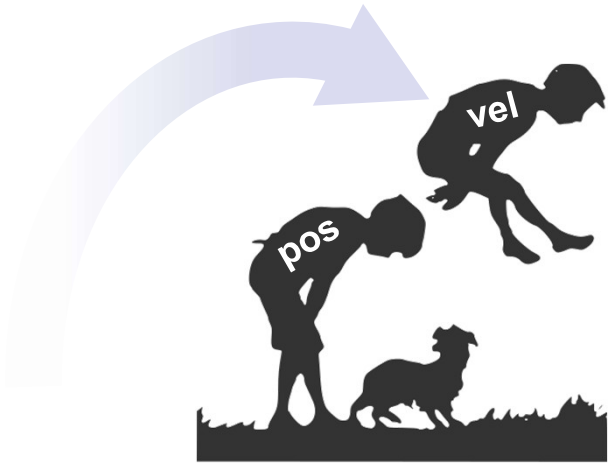


- From the code point of view, they are very similar
- The semantics changes:
  - in Symplectic Euler
    the position altered using *next frame* velocity
  - (it's "wrong", in a sense – but tends to work better)
- Similar properties, but better in practice
  - Same order of convergence (still just 1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

112

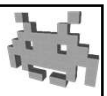
Leapfrog Integration Method





113

Leapfrog Integration Method

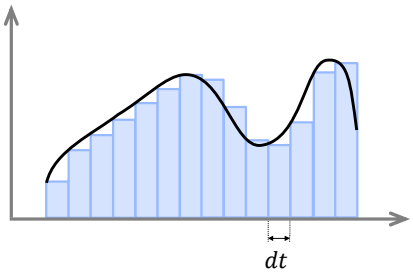


- Basic idea:
  store positions at time $k \cdot dt$
  (that is, at $t = 0, 1dt, 2dt, 3dt$...)
  but store velocities at time $k \cdot dt + ½\, dt$
  (that is, at $t = 0.5\, dt, 1.5\, dt, 2.5dt, 3.5dt$...)

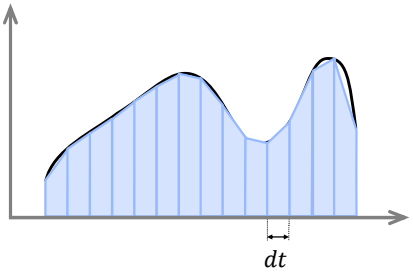
- (roughly) equivalent to use a summatory of
  the areas of trapezoids,
  (having base dt and average height $v(t + ½\, dt)$)
  not rectangles, to compute the integral

114

Better numerical integration (concept)



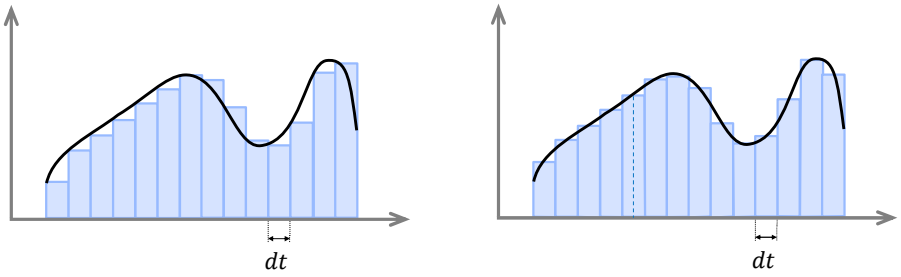
with Euler



with Leapfrog

115

Better numerical integration (concept)

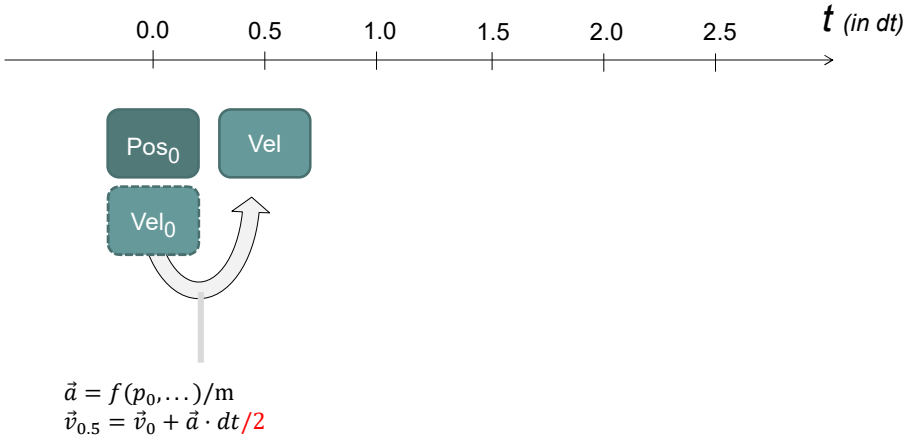


with Euler

with Leapfrog

116

Leapfrog Integration first step

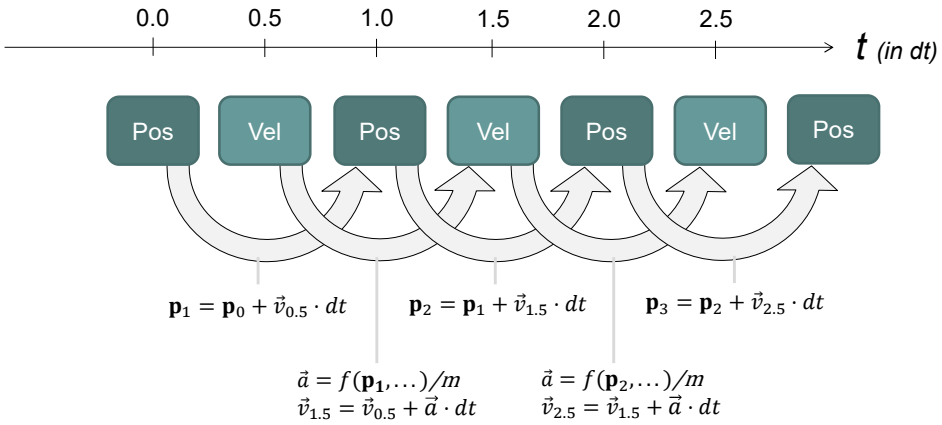


$\vec{a} = f(p_0, \ldots)/\mathrm{m}$

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

118

Leapfrog Integration



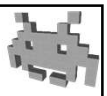
119

Leapfrog method: pros and cons

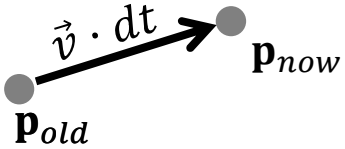
- Same cost as Euler – and basically same code
  - Velocity stored in status = velocity "half a *dt* ago" (and after updating it: "half a frame in the future")
  - Only real difference: the initialization of velocities
- Better theorical accuracy, for the same *dt*
  - better asymptotic behavior: it's 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

120

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



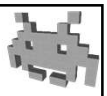
$$\mathbf{p}_{now} = \mathbf{p}_{old} + \vec{v} \cdot dt$$ ← Euler & variants

$$\Longrightarrow$$

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

121

Verlet integration method: (modifying Euler integration…)



init state

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

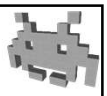
one step

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

expanding this…

122

Verlet integration method (modifying Euler integration…)



init state

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

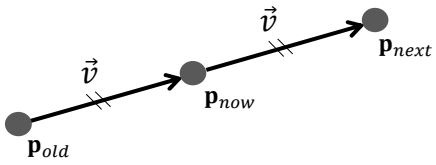
one step

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

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

123

Verlet integration method: geometric interpretation



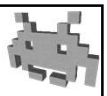
$\mathbf{p}_{next} = 2 \cdot \mathbf{p}_{now} - 1 \cdot \mathbf{p}_{old}$

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

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

124

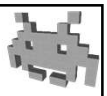
Verlet: characteristics

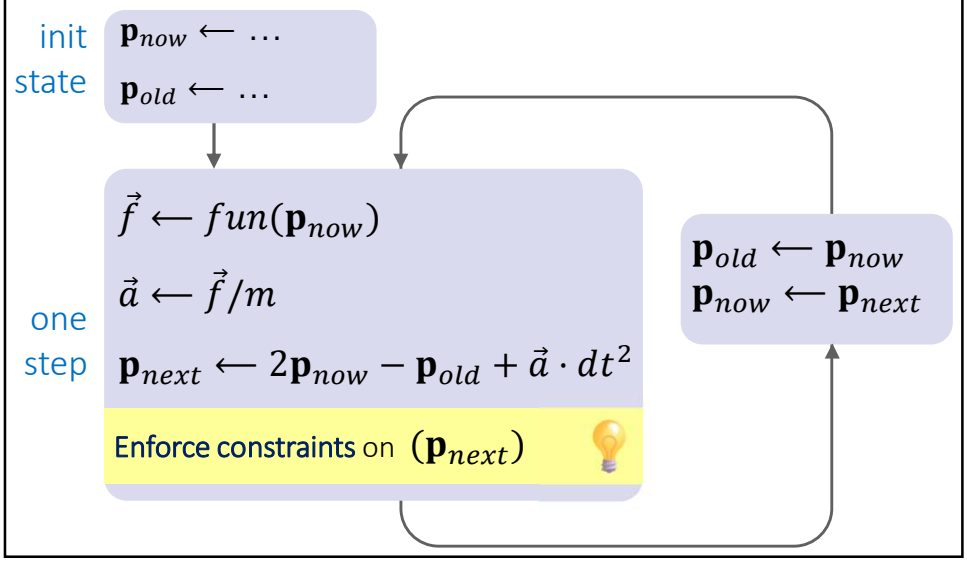


- Velocity is kept implicit
  - but that doesn't save RAM:
    we need to store previous position instead
  - (a point instead of a vector: same memory)
- Good efficiency / accuracy ratio
  - accumulated error: order of dt^2 (second order method)
- Extra bonus: reversibility
  - it's possible to go backward*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

125

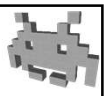
Verlet integration + "Position Based Dynamics" (PBD)





126

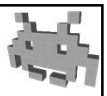
Position Based Dynamics (PDB)



- 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

127

Example of a positional constraint



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

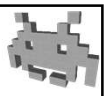
Enforce this constraint: trivial!

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

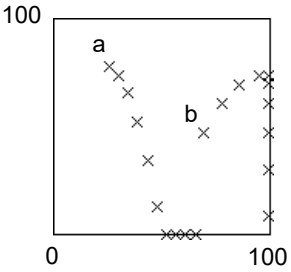
Imposing constraints like this one is a first part of collision response. For re-bounces, impulses must still be added (see collisions).

128

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



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



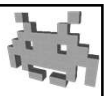
Enforce this constraint: simple clamp!

```
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).

129

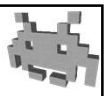
Verlet + Position Based Dynamics. Advantages



- 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

130

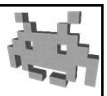
Verlet: *caveats*
(see next slides for solutions)



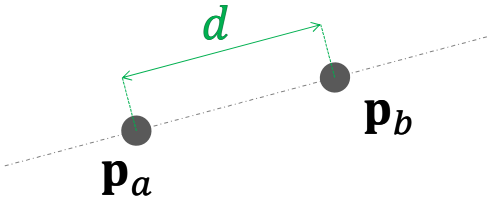
- ⚠ it assumes a constant *dt* (time-step duration)
  - if *dt* varies: corrections are needed! (how?)
- ⚠ Q: how to act on velocity (which is now implicit)?
  - for example, how to apply impulses ?
  - A: change $\mathbf{p}_{old}$ instead (how?)
- ⚠ Q: how to act of positions w/o impacting velocity?
  - for example, to apply teleports / kinematic motions ?
  - A: change both $\mathbf{p}_{new}$ and $\mathbf{p}_{old}$ (how?)
- ⚠ Q: how to apply velocity damps?

131

Example of positional constraint: equidistance constraint



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

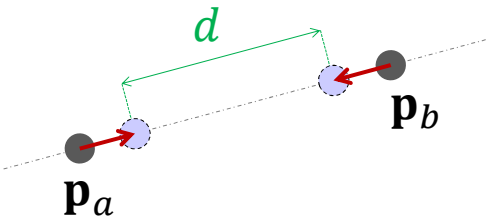


I require that... $\|\mathbf{p}_a - \mathbf{p}_b\| = d$

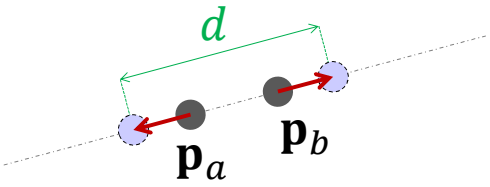
132

Enforce equidistance constraints
(assuming equal masses for now)

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



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



133

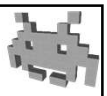
Enforce equidistance constraints: pseudo code

```
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)

134

Compare: equidistance constraints vs. springs



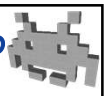
- Similar
  - they both mean: these 2 particles "want to be" at *this* distance (not more, not less)

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!

135

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



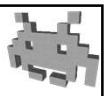
When enforcing constraints…

- 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?

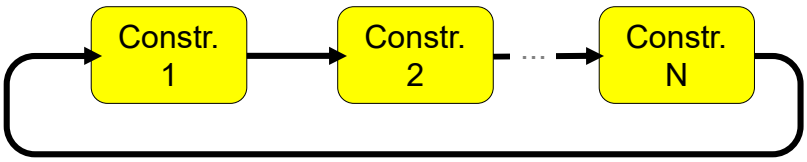
136

Enforcing a set of constraints



- There are many constraints to impose:
  when you solve one → maybe you break another!
- Simultaneous enforcement: computationally expensive

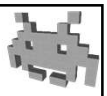
- Practical & easy solution: just enforce them in cascade
  (similar in concept to Gauss-Seidel solvers):



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

137

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

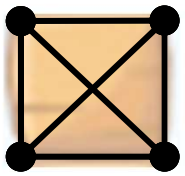
138

Compounds of particles disguised as rigid bodies

139

We can combine equidistance constraints to obtain rigid objects!

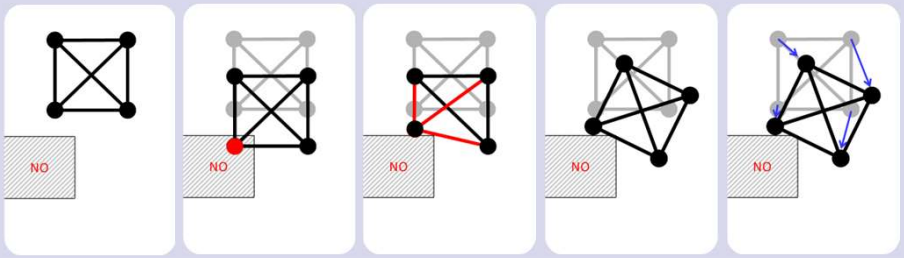
- Rigid body dynamics as emerging behavior
  - without explicitly keeping track their orientation, angular vel, angular acc., etc.



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

140

Example



FRAME 0

FRAME 1
dynamics

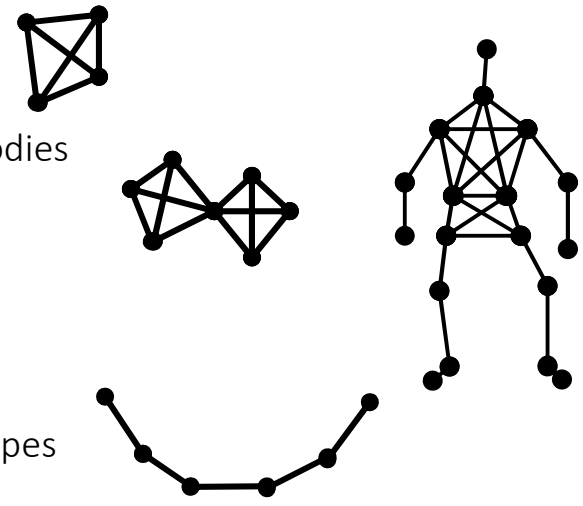
FRAME 1
constraints solving
(repeat until
convergence)

FRAME 1
final
-
implicit velocities
shown in blue

143

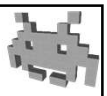
We can combine equidistance constraints to obtain…

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



144

Positional constraints (in general terms, and more formally)



- A predicate defined on the position(s) of a number of particles (usually, a small number: 1 - 4)

$$\mathcal{C}: (\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c, \dots) \rightarrow \{ true, false \}$$

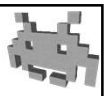
- For example, the equidistance constraints is

$$\mathcal{C}(\mathbf{p}_a, \mathbf{p}_b) \Leftrightarrow \|\mathbf{p}_a - \mathbf{p}_b\| = k_{CONST}$$

- They can be an equality ($=$) or an inequality ($\leq$ or $\geq$)

145

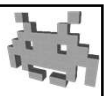
Equality positional constraints: examples



- Equidistance constraint (the one we have seen):
  «these 2 particles must stay at distance k»
  - E.g: because they are linked by a metal rod of length k
- Fixed positions:
  «this particle must stay in position $\mathbf{p}_a$ »
  - the particle is “pinned” in position
  - trivial to impose, but still useful!
- Coplanarity / collinearity:
  «these N particles must stay on a line / on a plane»

146

Equality positional constraints: other examples

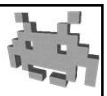


- Volume preservation:
  "The volume delimited by the squishy ballon defined by these particles is a constant k_{CONST} *"*
  (e.g., because it's filled with water)

- How to impose it (approximation):
  1. Estimate current total volume v
  2. uniform scale the entire object by factor $\sqrt[3]{f_{\text{CONS}} \ / v}$ around its barycenter

147

Inequality positional constraints: example



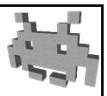
- "please don't sink below the ground"
  assuming the ground is the plane Y = 0

$$C(\mathbf{p}_a) \Leftrightarrow \mathbf{p}_a.y \geq 0$$

- Trivial to impose:
  just set the y to 0, if it is < 0

148

Inequality positional constraints: example



- "this particle must stay above this fixed (and arbitrary) plane"
  - For example, because the plane is a solid unmovable slab
  - The plane is given by a point on it $\mathbf{p}_\mathrm{q}$ and its normal $\hat{n}_q$

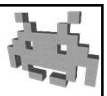
$$\mathrm{C}(\mathbf{p}_\mathrm{a}) \Leftrightarrow (\mathbf{p}_\mathrm{a} - \mathbf{p}_\mathrm{q}) \cdot \hat{n}_q \geq 0$$

- To enforce it (when it's not already true) enforce

$$(\mathbf{p}_\mathrm{a} - \mathbf{p}_\mathrm{q}) \cdot \hat{n}_q = 0$$

149

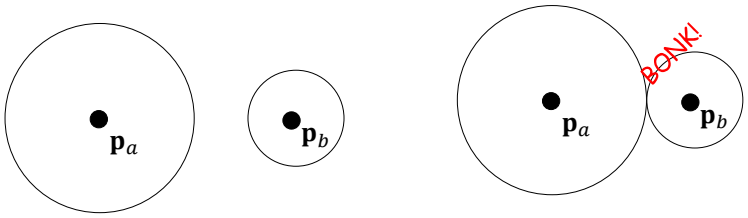
Inequality positional constraints: example



- These two particles must be *at least* k_{CONST} apart

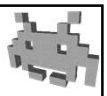
$$\mathcal{C}(\mathbf{p}_\mathrm{a}, \mathbf{p}_\mathrm{b}) \Leftrightarrow \|\mathbf{p}_\mathrm{a} - \mathbf{p}_\mathrm{b}\| \geq k_{CONST}$$

  - For example, because they are the centers of two rigid spheres and k_{CONST} is the sum of their radii
  - part of "collision handling" (a topic for later)



150

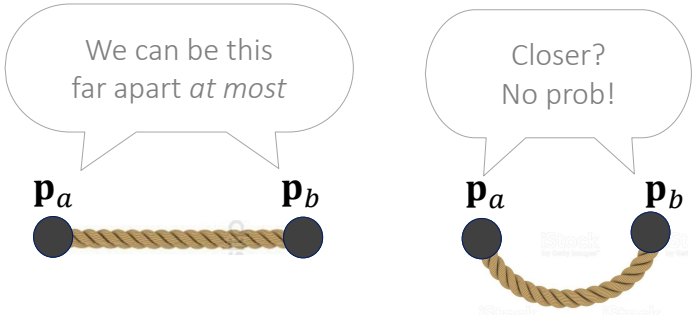
Inequality positional constraints: example



- These two particles must be *at most* k_{CONST} apart

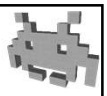
$$\mathcal{C}(\mathbf{p}_a, \mathbf{p}_b) \Leftrightarrow \|\mathbf{p}_a - \mathbf{p}_b\| \leq k_{CONST}$$

  - For example, because they are tied by an inextensible rope that has length k_{CONST} (but can fold)

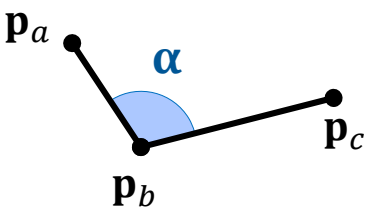


151

Inequality positional constraints: example

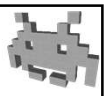


- Angle constraints, e.g. $\boldsymbol{\alpha} < \boldsymbol{\alpha}_{\max}$ with $\boldsymbol{\alpha}$ the angle between $\mathbf{p}_a, \mathbf{p}_b$ and $\mathbf{p}_b, \mathbf{p}_c$
  - e.g., on joints: *«elbows cannot bend backward»*
  - (a constraint between three particles!)



152

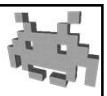
Enforcing one positional constraint (in general terms)



- Inequality constraint:
  1. *Test:* does the inequality already hold?
  2. If so: do nothing
  3. If not: enforce it as an equality (=) instead (see below)
- Equality constraint:
  - All involved particles must be displaced from that current position, so that it now holds
  - There can be infinite ways to achieve this!
    Question: Which one to pick?

153

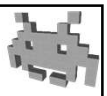
Enforcing one equality constraint: (assuming for now all particles have same mass)



- Answer:
  minimize the sum of *squared* displacements
  (with respect to current position)
- For each kind of constraint, we need to find the minimizer analytically
  ("analytically" = in closed form = "with formulas" = "solving a simple math problem on paper")
  - That's what we did for the equality constraint

154

Enforcing one equality constraint
(assuming for now all particles have same mass)



- We want to enforce a constraint $\mathcal{C}$ on particles $\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots$ currently in positions $\mathbf{p}_\mathrm{a}, \mathbf{p}_\mathrm{b}, \mathbf{p}_\mathrm{c}$ …

$$\mathcal{C}: (\mathbf{p}_\mathrm{a}, \mathbf{p}_\mathrm{b}, \mathbf{p}_\mathrm{c}, \ldots) \rightarrow \{ true, false \}$$

- We must apply the displacements $\overrightarrow{d_\mathrm{a}}$, $\overrightarrow{d_\mathrm{b}}$, $\overrightarrow{d_\mathrm{c}}$ that are the

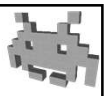
$$\underset{\overrightarrow{d_\mathrm{a}}, \overrightarrow{d_\mathrm{b}}, \overrightarrow{d_\mathrm{c}}, \ldots}{\operatorname{argmin}} \left(\|\overrightarrow{d_\mathrm{a}}\|^2 + \|\overrightarrow{d_\mathrm{b}}\|^2 + \|\overrightarrow{d_\mathrm{c}}\|^2 + \cdots \right)$$

$$\text{such that } \mathcal{C}\left(\mathbf{p}_\mathrm{a} + \overrightarrow{d_\mathrm{a}}, \mathbf{p}_\mathrm{b} + \overrightarrow{d_\mathrm{b}}, \mathbf{p}_\mathrm{c} + \overrightarrow{d_\mathrm{c}}, \ldots\right)$$

among all the choices that satisfy this,
we want the one which minimizes this

155

Enforcing one equality constraint
(in general, for particles with different mass)



- We want to enforce a constraint $\mathcal{C}$ on particles $\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots$ in positions $\mathbf{p}_\mathrm{a}, \mathbf{p}_\mathrm{b}, \mathbf{p}_\mathrm{c}$ … and with masses $m_\mathrm{a}, m_\mathrm{b}, m_\mathrm{c}, \ldots$

$$\mathcal{C}: (\mathbf{p}_\mathrm{a}, \mathbf{p}_\mathrm{b}, \mathbf{p}_\mathrm{c}, \ldots) \rightarrow \{ true, false \}$$

- We must apply the displacements $\overrightarrow{d_\mathrm{a}}$, $\overrightarrow{d_\mathrm{b}}$, $\overrightarrow{d_\mathrm{c}}$ which are the:

$$\underset{\overrightarrow{d_\mathrm{a}}, \overrightarrow{d_\mathrm{b}}, \overrightarrow{d_\mathrm{c}}, \ldots}{\operatorname{argmin}} \left(m_\mathrm{a}\|\overrightarrow{d_\mathrm{a}}\|^2 + m_\mathrm{b}\|\overrightarrow{d_\mathrm{b}}\|^2 + m_\mathrm{c}\|\overrightarrow{d_\mathrm{c}}\|^2 + \cdots \right)$$

$$\text{such that } \mathcal{C}\left(\mathbf{p}_\mathrm{a} + \overrightarrow{d_\mathrm{a}}, \mathbf{p}_\mathrm{b} + \overrightarrow{d_\mathrm{b}}, \mathbf{p}_\mathrm{c} + \overrightarrow{d_\mathrm{c}}, \ldots\right)$$

among all the choices that satisfy this,
we want the one which minimizes this

156