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

58

Euler integration methods

init state

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

one step

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

$$t = t + dt$$

59

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…

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

60

Forces

…

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

…

- Forces are often a function of current positions
  - But not always
- Examples:
  - Gravity
    - Constant, near the surface of a planet
    - But, function of positions in a space simulation
  - Wind
    - Depends on the area exposed in the wind direction
  - Electrical / magnetic forces
  - Archimede’s buoyancy
    - 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

61

Forces

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

- Forces are often a function of current positions
  - Not always
- Real-world forces can be modelled by things that aren't "forces":
  - Frictions
    - In reality: a force in the opposite direction of motion
    - Its magnitude is proportional to speed ($\vec{f}$ is a function of $\dot{\mathbf{p}}$: difficult to solve!)
    - Can be modelled with velocity drag / damp (see later)
  - Impacts & other violent things
    - In reality: very short, very strong forces
    - Duration << *dt*
    - Must be modelled with impulses (see later)
  - Resistance forces
    - 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
    - In reality, an internal force that contrast an external force (such as gravity)
    - Necessary to model "rigid bodies" and solid bodies
    - Must be modelled by positional constraints (see later)

62

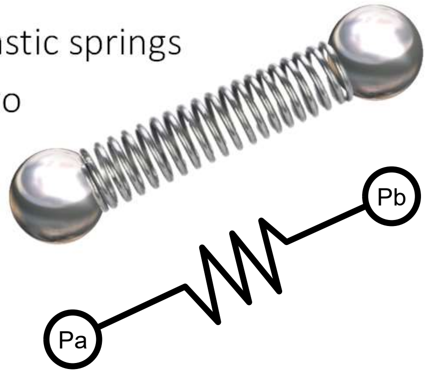
Forces: control forces

- 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

63

Forces: Springs (Hooke's law)

- Simplified model for elastic springs
- One spring connects two particles in Pa and Pb
- Characterized by:
  1. Rest length ℓ
  2. Stiffness k
- Srping force: counteracts stretching and compression



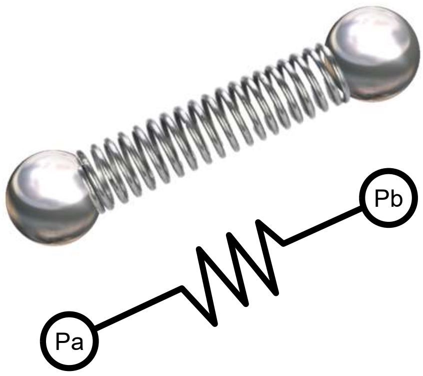
$$\vec{f_a} = k(\ell - \|\mathbf{p}_b - \mathbf{p}_a\|)\frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}$$

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

65

Forces: springs friction

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



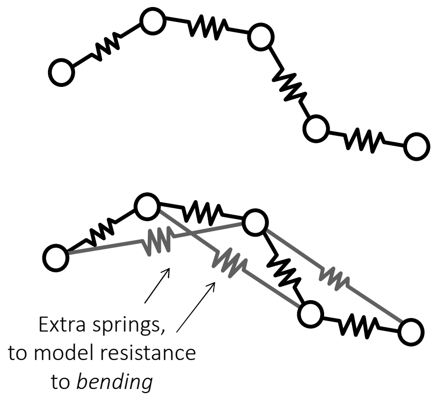
$$\hat{d} = \frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}$$

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

66

Mass and Spring systems

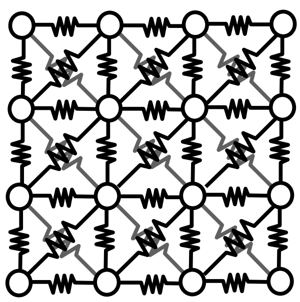
- Useful for deformable objects
- for instance: elasitic ropes (or hairs)

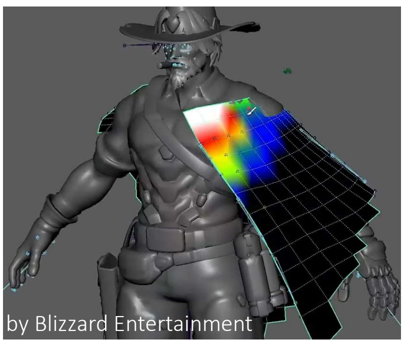


67

Mass and Spring systems

- For instance: cloth





68

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 dt needed
  - Doesn't work. How, then? see later

69

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)

70

Dynamics displacements VS kinematic

…
$$p = p + \vec{v} \cdot dt$$
…

aka dynamic displacements

(justified by the physics)

…
$$p = p + dp$$
…

aka Kinematic displacements

just "teleportation"

direct and discontinuous change of state (position)

71

Impulses VS Forces

a discontinuous change of state (velocity)!

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

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

- Forces (continuous)
  - Continuous application
  - every frame
- Impulses
  - Infinitesimal time
  - una tantum

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

72

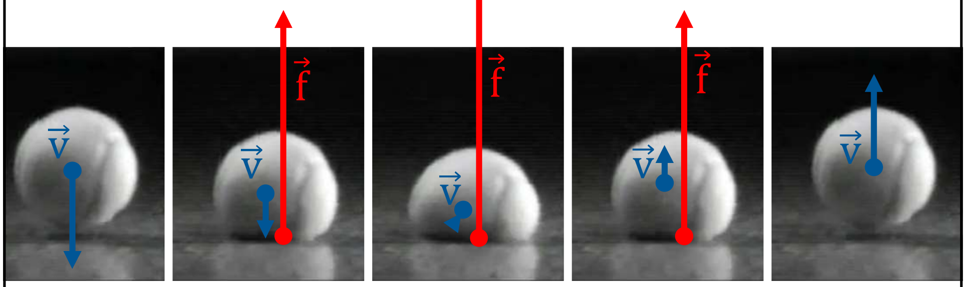
Impulses VS Forces

- 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

73

Impulses VS Forces

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



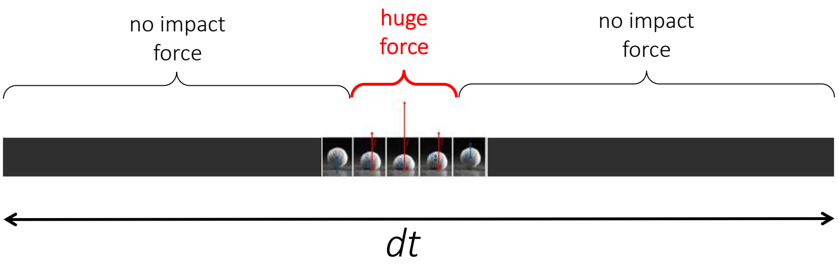
0 msec 1 msec 2 msec 3 msec 4 msec

- 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

75

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*

76

Effect of integration errors of System Energy

- Because integration errors:
  simulated solutions ≠ “real” solutions
- In a real system, the total energy cannot increase.
  - Usually, 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 faster and faster
- Particularly bad because:
  - Compromises stability
    (velocity = big, displacements = crazy, error = crazy)
  - Compromises plausibility
    (we can *see* it’s wrong)
- Therefore, a simple way to avoid this:
  make sure the simulation always includes attritions
  - makes simulation more stable + robust

77

Damping VS attrition forces

- We can include attrition as forces in our system
  - direction: opposite of current velocity direction
  - magnitude: proportional to a constant, and to speed (speed = magnitude of velocity vector)
  - note: so this force depends on velocity, not just positions.
  - This is the most correct way to model attrition
- Huge simplification: model attrition as "velocity damping"
  - simply, we reduce velocity vectors by a fixed proportion
  - e.g. reduce them all by 2% (drag = 0.02)
  - makes sense!
    Higher speed = more attrition = more loss of speed.
    Attrition = a "fixed tax" on speed.

78

Velocity Damping: the math

- I want to decrease velocity of a percentage for every second of (virtual) time
  - e.g.: if 2% then Drag = 0.02
- how should I update velocity for at every *dt* ? (1/FPS sec)

$$\vec{v} \longleftarrow \vec{v} \cdot (1 - Drag)^{dt}$$

- for small enough Drag, this is well approximated by

$$\vec{v} \longleftarrow \vec{v} \cdot (1 - dt \cdot Drag)$$

79

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 );
}
```

80

Velocity Damping: notes

- Velocity Damping is useful for robustness,
  - avoids energy to increase
- Problems of Velocity Damping
  - tends to exaggerate frictions;
    even when it makes sense, e.g. in space, no air
  - Crude approximation: attrition forces are not really linear with speed
  - It's attrition with everything…: air, soil.
  - Isotropic force: in reality, attrition force depends of velocity direction
- In practice:
  - low values: hardly noticeable (except in the long run)
  - high values: feels like everything is moving in molasses; (ita: *melassa*)
    everything quickly grinds to a halt
  - very high values: (e.g. 50% per frame) basically, no inertia anymore
    (useful to quickly converge to (local) minimal energy states:
    becomes basically a solver for static problems, not of dynamics)

81

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

- Some commonly used alternatives:
  - “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
  - E.g. order of accuracy
  - Note: a more accurate method is also more efficient
    (larger d*t* are possible, so fewer steps are necessary)

82

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)
- in practice, 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

83

Forward Euler

init state

$\mathbf{p} \leftarrow \cdots$

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

one step

$\vec{\mathrm{f}} \leftarrow fun(\mathbf{p}, \dots)$

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

$\mathbf{p} \leftarrow \mathbf{p} + \vec{\mathrm{v}} \cdot dt$

$\vec{\mathrm{v}} \leftarrow \vec{\mathrm{v}} + \vec{\mathrm{a}} \cdot dt$

$t = t + dt$

84

Symplectic Euler

init state

$\mathbf{p} \leftarrow \cdots$

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

one step

$\vec{\mathrm{f}} \leftarrow fun(\mathbf{p}, \dots)$

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

$\vec{\mathrm{v}} \leftarrow \vec{\mathrm{v}} + \vec{\mathrm{a}} \cdot dt$

$\mathbf{p} \leftarrow \mathbf{p} + \vec{v} \cdot dt$

$t = t + dt$

85

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…

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

86

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…

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

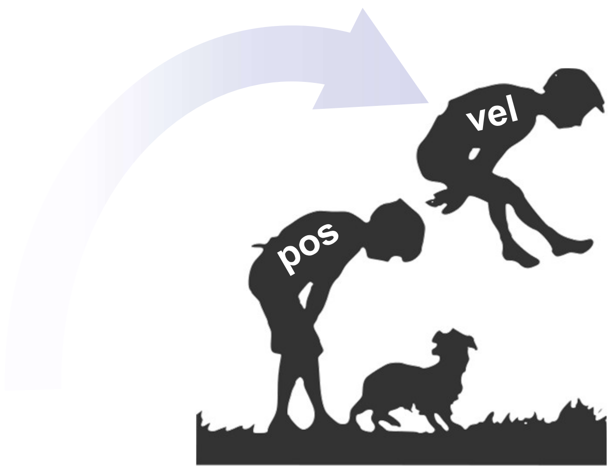
87

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 one ☹)
  - On average, better behavior
  - More stable, more accurate

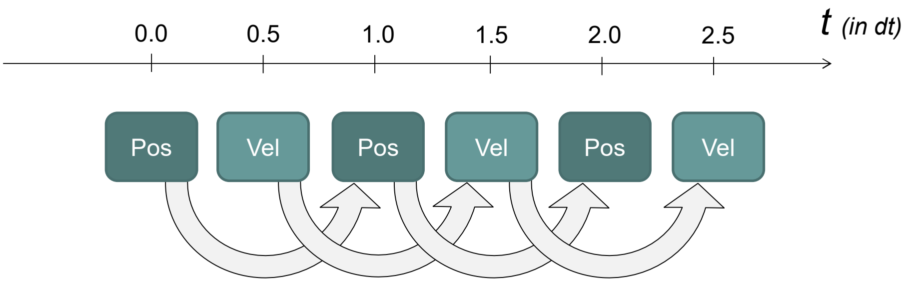
89

Leapfrog Integration



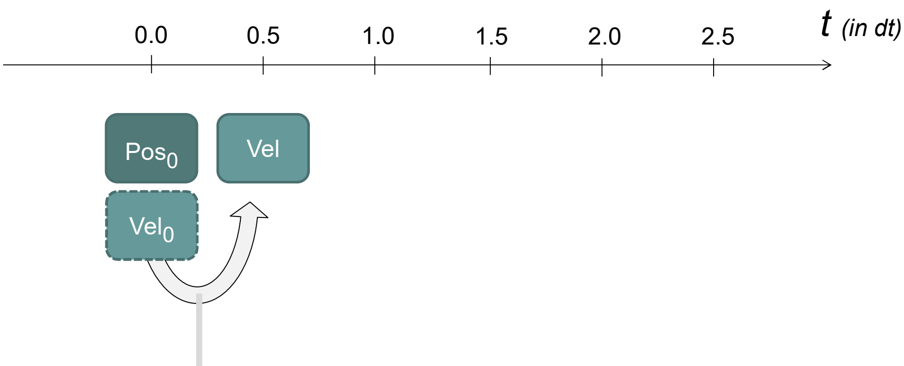
90

Leapfrog Integration



91

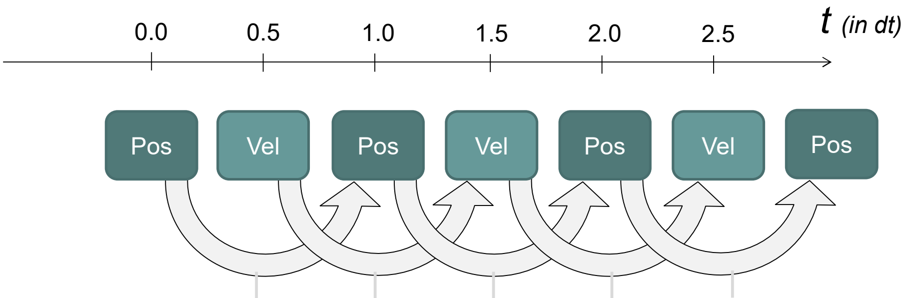
Leapfrog Integration first step



$$a = f(p_0\ ,\ \dots)$$
$$\vec{v}_{0.5} = \vec{v}_0 + \vec{a} \cdot dt / 2$$

92

Leapfrog Integration



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

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

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

93

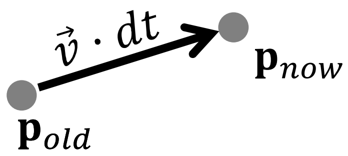
Leapfrog method: pros and cons

- 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 e.g. floating point errors)
- But: requires fixed *dt* during all the simulation
  - for the theory to work as advertised

94

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

95

Verlet integration method

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...

96

Verlet integration method

init state

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

one step

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

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

97

Verlet: characteristics

- Velocity is kept implicit
  - but that doesn't save RAM:
    we need to 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 from any state
  - only in theory… careful with implementation details

98

Verlet: *caveats*

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

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

⚠ 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)

99

Changing the value of ***dt*** in Verlet
(if it's 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

$$\Longrightarrow$$

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

100

Velocity damping in Verlet (way 2)

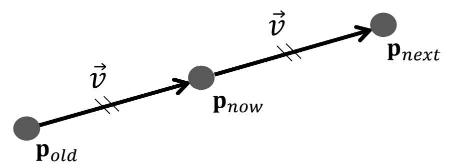
- Velocity at next frame: $\vec{v} = (\mathbf{p}_{next} - \mathbf{p}_{now})/dt$ (implicit)
- We want to multiply $\vec{v}$ a factor c_{damp} (e.g. 0.98 obtained as $(1\text{-}dt \cdot c_{DRAG})$)
  - before applying accelerations
- We can do that using a more general formula for $\mathbf{p}_{next}$

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

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

102

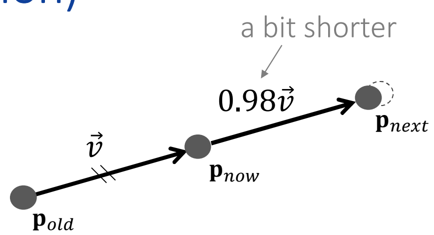
Velocity damping in Verlet (way 2) (geometric interpretation)



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

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 $\mathbf{p}_{now}$, $\mathbf{p}_{old}$:

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

103