


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** ●●

56

Integration errors



- A numerical integrator only approximates the real value of the integrals
- The discrepancy (simulation errors) accumulate with virtual time during all the simulation
- How much error is accumulated?
- It depends on dt !
 - Small $dt \Rightarrow$ more steps needed (for same virtual time) \Rightarrow more computationally expensive, but smaller errors, i.e. more accurate simulation

57

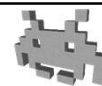
Order of convergence



- How much does the total error decrease as dt decreases?
 - That's called the Order of the simulation
 - 1st order: the total error can be as large as $O(dt^1)$
 - "if the number of physics steps doubles (physical computation effort doubles) dt becomes halves and errors can be expected to halve"
 - The error introduced by each single step is $O(dt^2)$,
 - Euler is 1st order
 - This is not too good, we want better
 - Note: The error is usually not that bad as linear with dt , but they *can* be

58

The integration steps dt of any numerical methods (summary)



- dt : delta of **virtual time** from last step
- the "temporal resolution" of the simulation!
 - if **large**: more efficiency
 - fewer steps to simulate same amount of virtual time
 - if **small**: more accuracy
 - especially with strong forces and/or high velocities
 - Common values: 1 sec / 60 ... 1 sec / 30
 - i.e. a step simulates around 16 ... 32 msec. of virtual time
 - note: it's not necessarily the same refresh rate of rendering (FPS of rendering \neq FPS of physics. Rendering can be *less!*)
 - note: dt is not necessarily the same in all physics steps (need more accuracy *now*? Decrease dt)

number of physics steps per sec, or «physics FPS»

59

Forces

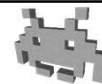
$$\vec{f} = \text{function}(p, \dots)$$



- In general, forces are a function of current positions
- Examples
 - Gravity
 - Wind, electrical, magnetic, Archimede's buoyancy, mechanical springs, shock waves (explosions), etc ...
 - Fake / "Magic" control forces
 - added for controlling the evolution, not physically justified
 - Frictions
 - They also depend on speed
 - But, they can be accounted for using *damping* – see later
 - And more (resistance forces of solid objects?)

60

Forces: control forces

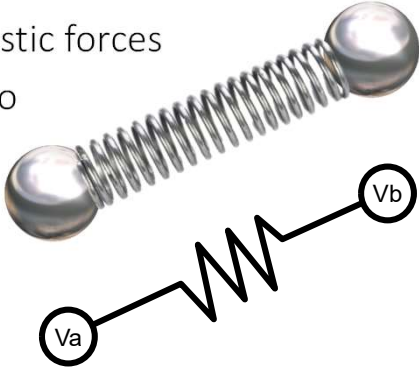


- Example: the player pressing the forward button
⇒ a forward force is applied to his/her avatar
 - no physical justification
 - "Don't ask questions, physics engine"
- According to many:
it's better when that's not done much
 - the more physically justified the forces, the better
 - for example: does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

61

Forces: Springs

- Simplified model for elastic forces
- One spring connects two particles Va and Vb
- Characterized by:
 1. Rest length L
 2. Elastic constant K
- Force: counteracts stretching and compression



The force f exerted by spring on Va is:

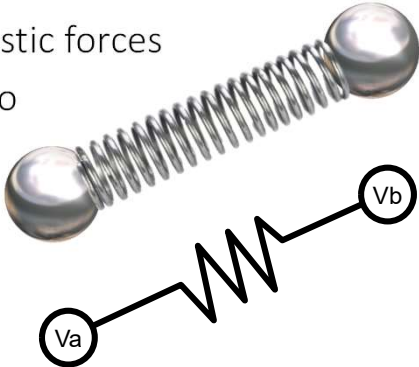
- direction: versor from Vb to Va
- magnitude: $K (L - \text{dist}(Va, Vb))$

The force f exerted on Vb is $-f$

62

Forces: Springs

- Simplified model for elastic forces
- One spring connects two particles Va and Vb
- Characterized by:
 1. Rest length ℓ
 2. Elastic constant k
- Force: counteracts stretching and compression



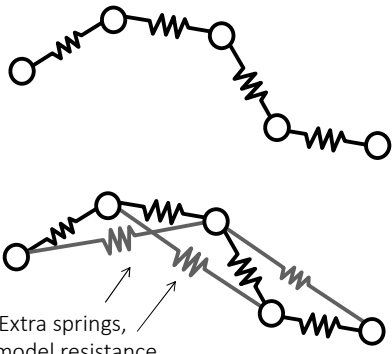
$$\vec{f}_a = k(\|v_a - v_b\| - \ell) \cdot \frac{(v_a - v_b)}{\|v_a - v_b\|}$$

$$\vec{f}_b = -\vec{f}_a$$

63

Mass and spring systems

- Useful for deformable objects
- for instance elastic ropes (or hairs)



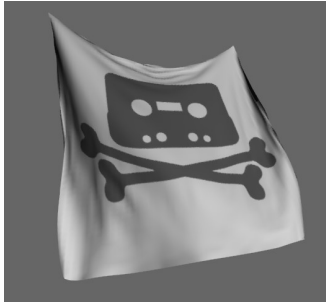
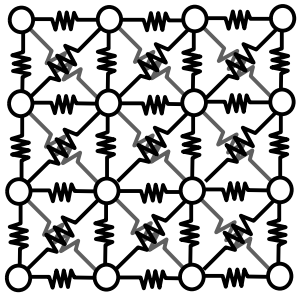
Extra springs, to model resistance to bending

The diagram shows two configurations of mass-spring systems. The top configuration is a simple chain of five masses connected by four springs, forming a curved shape. The bottom configuration is a more complex system with six masses and several springs, including two additional springs that cross between non-adjacent masses to model bending resistance. Arrows point to these extra springs with the text 'Extra springs, to model resistance to bending'.

64

Mass and spring systems

- For instance cloth



img by msqrt (pauli kemppinen)

The diagram on the left shows a 4x4 grid of masses connected by springs in a square lattice. The diagram on the right is a 3D rendering of a cloth simulation, showing a white rectangular piece of fabric with a black skull and crossbones logo, draped and waving.

65

Mass and Spring systems can model...



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

66

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)

67

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)

68

Impulses VS Forces

a discontinuous change of state (velocity)!

...

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

...

- **Forces** (continuous)
 - Continuous application
 - every frame

...

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

...

- **Impulses**
 - Infinitesimal time
 - una tantum

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

69

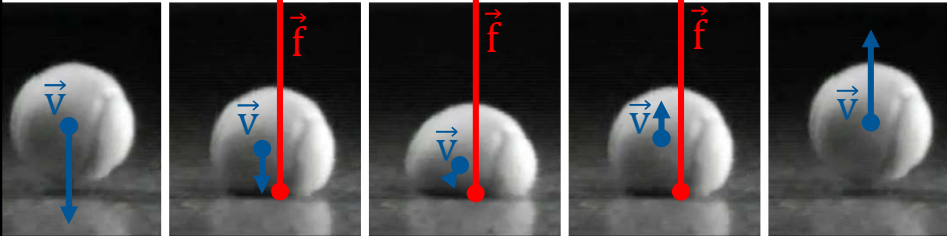
Impulses VS Forces

- **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 $\ll dt$
 - ex: a tennis ball rebounding against a tennis racket

70

Impulses VS Forces

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

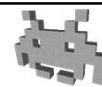


0 msec 1 msec 2 msec 3 msec 4 msec

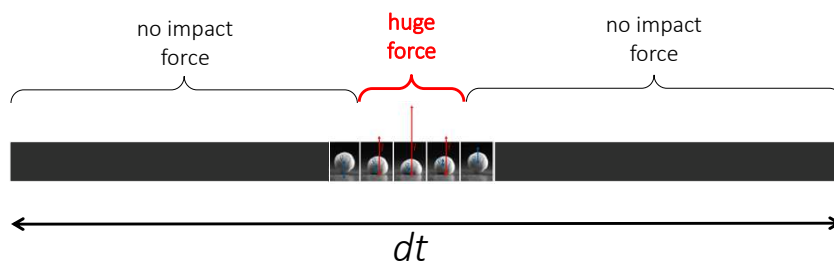
- very strong forces (but not infinite)
- applied for a very short time (but not instantaneous)
- see *collision response* later for details about the impulse based approximations

72

Impulses VS Forces



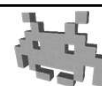
- what does *truly* happen when it bounces off the ground?



- This can only be modelled as an *impulse*, not a force
(and maybe a small *teleport* of the ball to move it outside the table)

73

Effect of integration errors of System Energy



- Because integration errors:
simulated solutions \neq "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

74

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.

75

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

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

- for small enough Drag, this is well approximated by

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

76

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 );  
}
```

77

Velocity Damping: notes



- Problem of Velocity Damping
 - tends to exaggerate frictions;
it is needed for robustness, no or almost no damping = bad idea 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
- Practical effects:
 - 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 minimal energy state: becomes basically a solver for of statics, not of dynamics)

78

Different numerical integrators ("numerical ways to compute integrals")



- Some commonly used alternatives in games:
 - "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
 - There can be semantic difference (the variables stand for slightly different things)
 - They can differ in accuracy / behaviour
 - Note: a more accurate method is more efficient (larger dt are possible, so fewer steps are necessary)

79

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 \Rightarrow 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

80

Forward Euler *pseudo code*



<pre>Vec3 position = ... Vec3 velocity = ... void initState(){ position = ... velocity = ... } void physicsStep(float dt) { Vec3 acceleration = compute_force(position) / mass; position += velocity * dt; velocity += acceleration * dt; } void main(){ initState(); while (1) do physicsStep(1.0 / FPS); }</pre>	<p>Equivalent to...</p> $\vec{f}_i = \text{function}(p_i, \dots)$ $\vec{a}_i = \vec{f}/m$ $\vec{v}_{i+1} = \vec{v}_i + \vec{a}_i \cdot dt$ $p_{i+1} = p_i + \vec{v}_i \cdot dt$
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

81

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



<pre>Vec3 position = ... Vec3 velocity = ... void initState(){ position = ... velocity = ... } void physicsStep(float dt) { Vec3 acceleration = compute_force(position) / mass; velocity += acceleration * dt; position += velocity * dt; } void main(){ initState(); while (1) do physicsStep(1.0 / FPS); }</pre>	<p>Equivalent to...</p> $\vec{f}_i = \text{function}(p_i, \dots)$ $\vec{a}_i = \vec{f}/m$ $\vec{v}_{i+1} = \vec{v}_i + \vec{a}_i \cdot dt$ $p_{i+1} = p_i + \vec{v}_{i+1} \cdot dt$ <div style="text-align: center; margin-top: 5px;"> </div>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

just flip the order

82

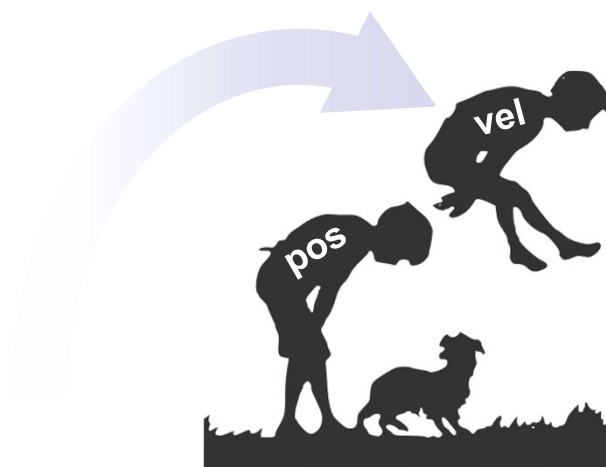
Forward Euler VS Symplectic Euler (warning: over-simplifications)



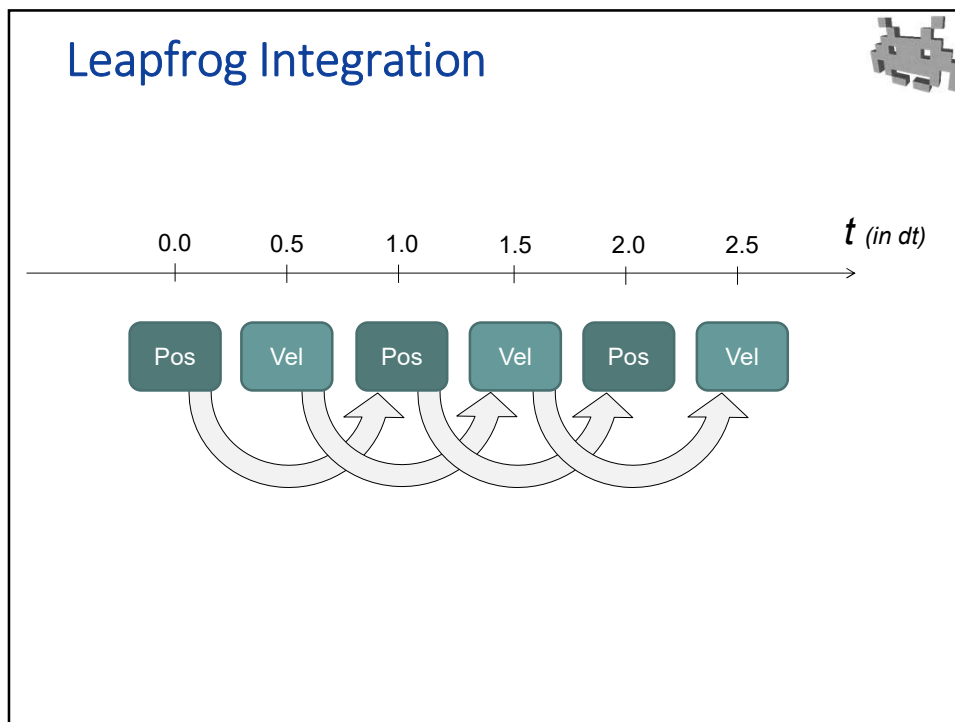
- From the code point of view, they are very similar
- The semantics changes:
 - in Symplectic Euler
the position altered using *next frame* velocity
 - (it's "wrong", in a sense – but works better)
- Similar properties, but better in practice
 - Same order of convergence (still just one ☹)
 - On average,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

84

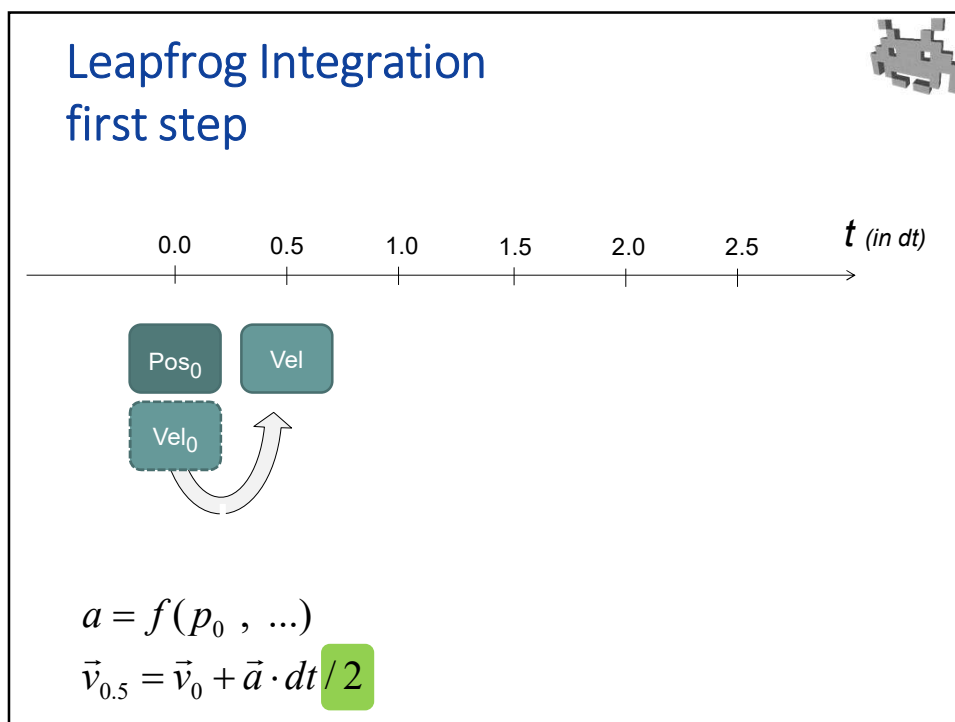
Leapfrog Integration



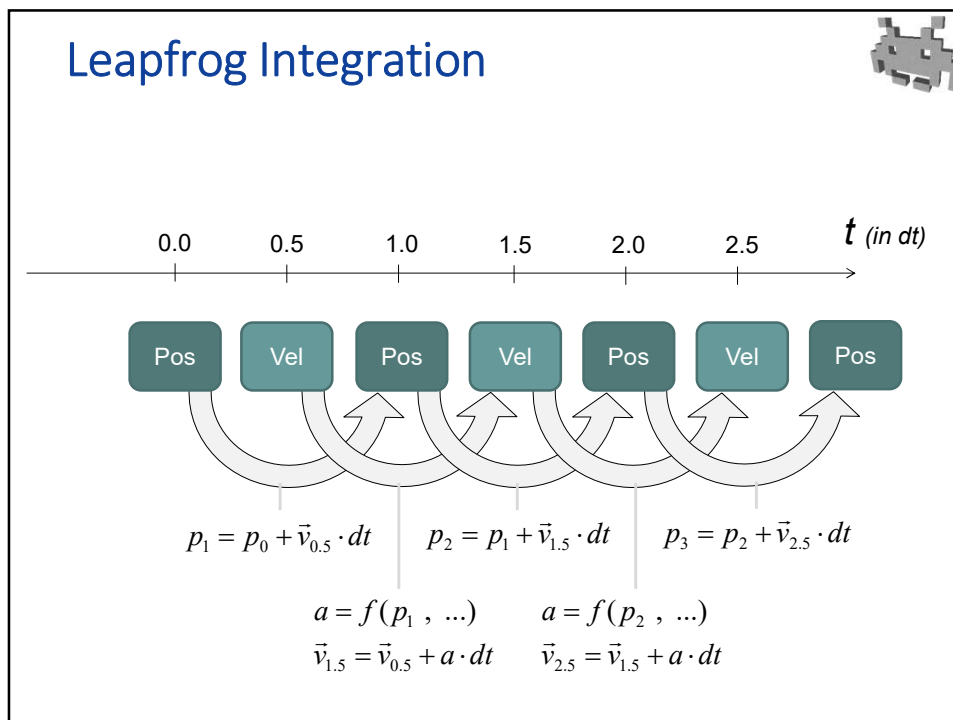
85



86



87



88

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 theoretical 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

89