3D video games

3D Game Physics

Marco Tarini

2

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: Networking for 3D Games
lec. 11: 3D Audio for 3D Games
lec. 12: Rendering Techniques for 3D Games
lec. 13: Artificial Intelligence for 3D Games

computer animation

3

Animation in games

but, a note on terminology:
in some contexts, procedural means
"produced by a *simple* procedure"
as opposed to "physically simulated"

Non procedural	**Procedural**
Assets!	Physics engine
Fully controlled by artist/designer (dramatic effects!)	Less control
Realism: depends on artist's skill	Physics-driven realism
Does not adapt to context	Auto adaptation to context
Repetition artefacts	Naturally repretition free

4

Physics simulation in videogames

- 3D, or 2D
- "soft" real-time
- efficiency
  - 1 frame = 33 msec (at 30 FpS)
  - physics = 5% - 30% max of computation time
- plausibility
  - but not necessarily *accuracy*
- robustness
  - should almost never "explode"
  - it's tolerable to have inconsistency in a few frames, as long as it recovers in subsequent ones

6

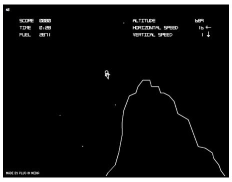
Physics in games: cosmetics or gameplay?

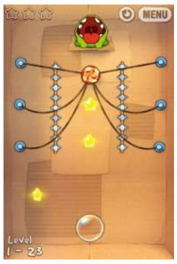
- Just a graphic accessory? (for realism!)
  - e.g.:
    - particle effects (w/o feedback)
    - secondary animations
    - Ragdolling
- Or a gameplay component?
  - e.g. physics based puzzles
  - Popular approach in 2D (since always!)

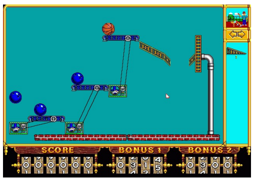
7

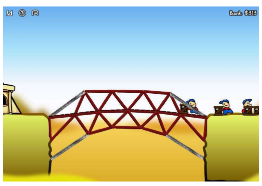
Physics in games: cosmetics or gameplay?

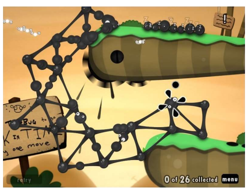
- Just a graphic accessory? (for realism!)
  - e.g.:
    - particle effects (w/o feedback)
    - secondary animations
    - Ragdolling
- Or a gameplay component?
  - e.g. physics based puzzles
  - Popular approach in 2D (since always!)







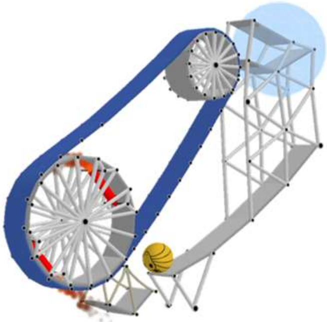


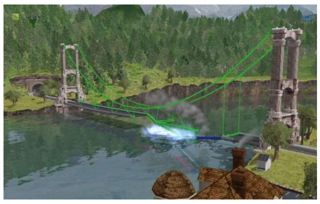


8

Physics in games: cosmetics or gameplay?

- Just a graphic accessory? (for realism!)
  - e.g.:
    - particle effects (w/o feedback)
    - secondary animations
    - Ragdolling
- Or a gameplay component?
  - e.g. physics based puzzles
  - Rising trend in 3D





9

Physics engine: intro

- Game engine module
  - executed in real time at game run-time
- A high-demanding computation
  - on a very limited time budget!
- ...but highly parallelizable
  - potentially, highly parallel

==> good fit for hardware support

(just like the Rendering Engine)

10

Hardware for Physics engine

To exploit a strong parallelism, you need a strongly parallel hardware!

- For a brief moment ~2006: PPU
  - "Physics Processing Unit"
  - HW unit specialized for physics
- After that: GP-GPU
  - "General Purpose Graphics Processing Unit"
    = Use of the graphics card for generic tasks
    (not related with 3D computer graphics)
  - or, Cuda (nVidia), OpenCL (openSource)

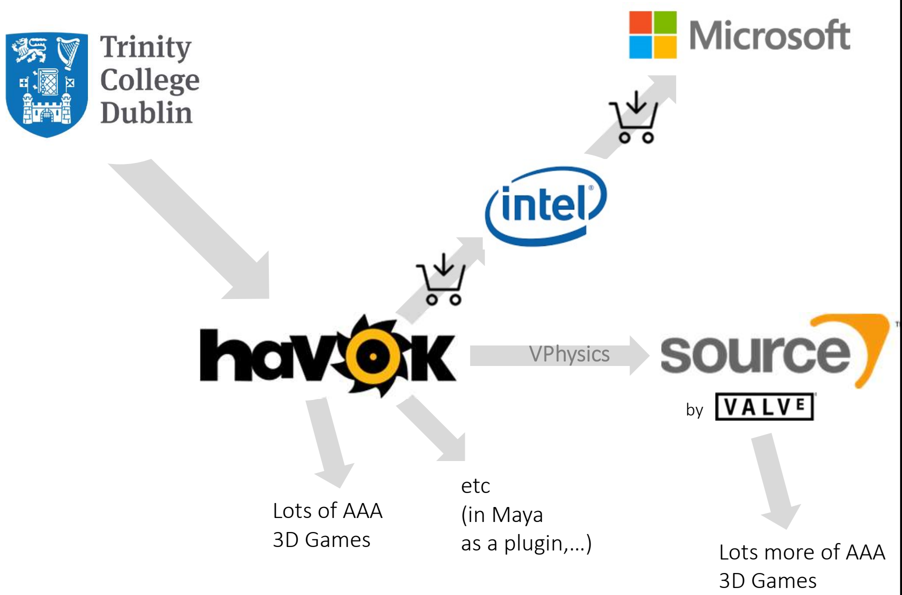
11

Main Software (libraries, SDK)

havok	mostly CPU (Microsoft)
PhysX™	CPU+GPU (CUDA) NVidia
BULLET PHYSICS LIBRARY	open source, free, HW accelerated (OpenCL) + CPU
OPEN DYNAMICS ENGINE™	open source, free
Box2D	2D, open source, free

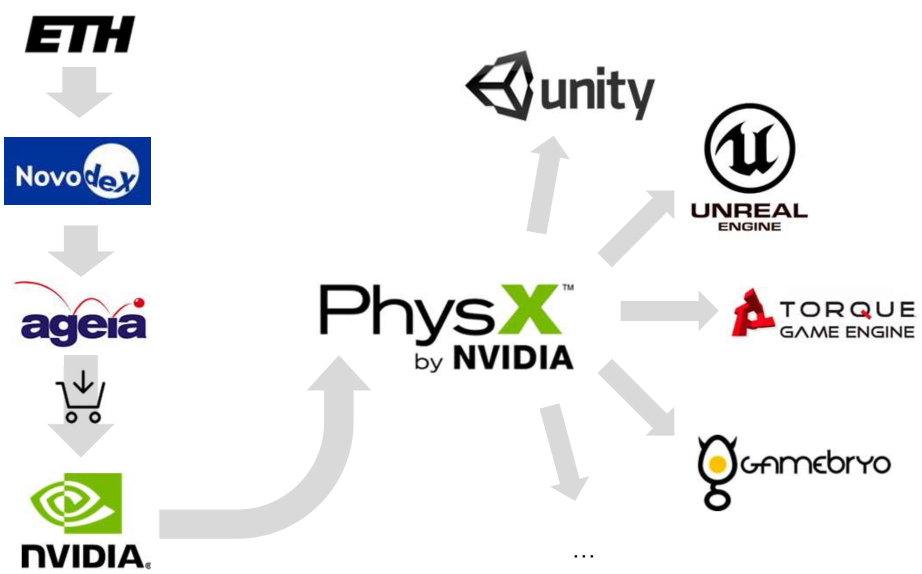
12

Brief history



13

Brief history



14

The 2 tasks of the Physics engine

1. Dynamics (Newtonian)

for objects such as:

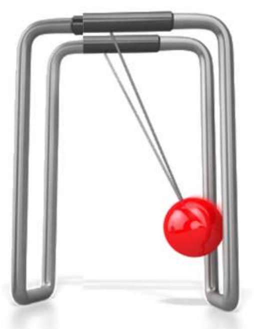
- Particles
- Rigid bodies
- Articulated bodies
  - E.g. "ragdolling"
- Soft bodies
  - Ropes (specific solutions)
  - Cloth (specific solutions)
  - Hair (specific solutions)
  - Free-form deformation bodies (general)
- Fluids
  - Expensive!

2. Collision handling

- Collision detection
- Collision response

15

Fields of study



Dynamics

The motion,
as a result of **forces**

Example:
"Subject to gravity, how will this pendulum swing?"

Statics

Equilibrium states,
minimal **energy** states

Example:
"In which state(s) can this pendulum be still?"

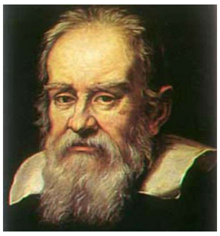
Kinematics

The **motion** itself, no matter why it moves

Example:
"If the angular speed of the pendulum is currently X, how fast is the ball moving?" (or vice versa)

17

Newtonian Dynamics



18

Physics and spaces (observation)

- The scene hierarchy, or the entire distinction between local and global space, its's entirely "in our mind"
  - It's a useful abstraction to control or code *scripted* animations
  - E.g., kinematics animations, skeletal animations…
- But physics *doesn't care* about any of it
  - Physics happens entirely in global (world) space
  - Persistent spatial relationships (e.g., between a car and its wheels) either exists due to physical constraints, or they are irrelevant
  - Even if they physically exists, they are still enforced in global space, like all the rest of the physics simulation
  - Physics simulation computes changes to objects states (position, orientation…) in global space
  - But, as we know, these updates can be converted/stored in local space

19

Spatial placement of a (rigid) object

2D Physics

- Position:
  (x,y)
- Orientation:
  (α) – angle (scalar)

3D Physics

- Position:
  (x,y,z)
- Orientation:
  quaternion or
  axis,angle or
  axis * angle or
  3x3 matrix or
  Euler angles

20

Newtonian dynamics: summary

Current object placement	Rate of change of ← (d / dt)	← "with mass" (momentum)	What changes the rate of change (d^2 / dt^2)	← "with mass"
Position p p = (x,y,z)	Velocity $\vec{v}$ $\vec{v} = \dot{p}$ ($\|\vec{v}\|$ = "speed")	Momentum $m\,\vec{v}$	Acceleration $\vec{a} = \dot{\vec{v}} = \ddot{p}$	Force $\vec{f}$ $\vec{f} = m\,\vec{a}$
Orientation (e.g. quaternion)	Angular velocity $\vec{\omega}$	Angular momentum $I\,\vec{\omega}$ I = moment of inertia	Angular acc. $\vec{\alpha}$	Torque $\vec{\tau}$ $\vec{\tau} = I\,\vec{\alpha}\cdot$ ("mechanic momentum")

state (is kept! inertia!)
(changes, but only continuously)

change the state
(no memory)

29

Per-object constant: mass & its distribution (for non point-shaped ones)

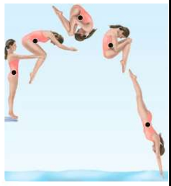
A few quantities associated to each rigid object

- constants: they don't (normally) change
- *input* of the physics dynamic simulation, not output

- Mass:
  - resistance to change of velocity

distribution of mass

- Moment of Inertia:
  - resistance to change of *angular* velocity
- Barycenter:
  - the center of mass



30

Mass: notes

- resistance to change of velocity
  - also called *inertial* mass
- also, incidentally:
  ability to attract every other object
  - also called *gravitational* mass
  - happens to be the same
- it's what you measure with a scale
- Unity of measure:
  kg, g, etc…

31

Barycenter: notes

- Aka the center of mass
  - it's a fixed position (for a rigid body)
- It's simply the *weighted average* of the positions of the subparts composing an object
  - literally "weighted": with their masses
- Does not necessarily coincide with the origin of the local frame of that object
  - but it can
  - otherwise, it's a fixed point (in local frame)
- In a physical simulation, the position of a rigid body is better described as the position of its barycenter
- In absence of forces, the object rotates (orbits, spin) around this position.

32

Moment of inertia: notes 1/3

- Resistance to change of angular velocity

- (an object rotates around its barycenter)

33

Moment of inertia: notes 2/3

- Scalar moment of inertia
  - Resistance to change of angular velocity
  - Depends on the total mass, and also on its *distribution*
    - the farthest one sub-mass from the axis, the > the resistance
- In 2D: it's a fixed value
  (for a given rigid object)
  - The object always spins around its barycenter

34

Moment of inertia: notes 3/3

- In 3D: the rigid objects spins around an axis passing through the barycenter
  - for any possible axis of rotation,
    you have a different *scalar* moment of inertia
  - for a given axis $\hat{a}$ the scalar moment is given by

    $$\hat{a}^{\mathrm{T}}\ \mathrm{M}\ \hat{a}$$

    where 3×3 matrix M is the «(moment of) inertia *matrix*»
    aka the «(moment of) inertia *tensor*»)
  - M can be computed for a given rigid object
    - how: that's beyond this course
    - in practice: use given formulas for common shapes
    - or, sum the contributions for each sub-mass
  - M describes the scalar moment of inertia for any possible axis or rotation

35

State of a rigid object in a physical simulation

```
Point position
Rotation orientation
```
current

```
Vector velocity
Vector angular_velocity
```
current rates of change

updated by physics (dynamics)

```
Scalar mass
Matrix moment_of_inertia
Point barycenter
Scalar drag
...
```
frictions; see later

constants

setup at initialization, rarely changes (e.g. by scripts)

*Note: acceleration/forces/torques are **not** part of the state*

36

In unity

```
Point position
Rotation orientation
```
part of **Transform** component

```
Vector velocity
Vector angular_velocity
Scalar mass
Matrix moment_of_inertia
Point barycenter
Scalar drag
bool isKinematic
```
the **RigidBody** component

Adding a “RigidBody” component to a Game Object is to say: *“please let the Physics engine take care of the dynamics of this object”*

37

In unity (using Unity terminology)

note: they are the components of the **global** transformation!

```
Vector3 position
Quaternion rotation
```

part of Transform component

note: speed = velocity.magnitude

```
Vector3 velocity
Vector3 angular_velocity
float mass
Vector3 inertiaTensor
Quaternion inertiaTensorRotation
Vector3 centerOfMass
float drag
bool isKinematic
```

the RigidBody component

per second
(not per frame!)

moment of inertia matrix

the Vector3 = a diagonal matrix D
by rotating it $R^T D R \rightarrow$ the final matrix

the barycenter (in local space)

if true: disable dynamics
(but keeps e.g. collisions)

38

The case of particles

- For now, we will study a simpler case:
  the dynamics of particles (and its simulation)
- Particle = ideal object shaped like a point,
  with all the mass concentrated in that point
- Particles-only is easier because the orientation (rotation) is irrelevant, and so the following are also irrelevant
  - the center of mass (it's the position of the particle itself);
  - the distribution of mass, i.e. the moment of inertia (there's none);
  - the torques (instead, there's only forces);
  - the angular velocity (instead, there's only linear velocities)
- These things are only relevant again for non-point sized (rigid) objects
- The basic algorithms, however, are the same.

39

State of a particle (point sized obj) in a physical simulation

```
Point position
~~Rotation orientation~~
Vector velocity
~~Vector angular_velocity~~
Scalar mass
~~Matrix moment_of_inertia~~
~~Point barycenter~~
Scalar drag
…
```

not used for point sized objects!

One possibility in a game phys engine is to only simulate point-particles.

Simpler: no rotation needed!

We will see later how to still get rigid bodies back.

For now, we focus on this simpler case.

40

Newtonian Dynamics (for particles)

describes the forces
given all the particle positions (and more)

$$\begin{cases} \vec{f}(t) = \text{function}(\,\mathbf{p}(t)\,, \ldots) \\[2ex] \vec{a}(t) = \dfrac{\vec{f}(t)}{m} \\[2ex] \vec{v}(t) = \vec{v}_0 + \displaystyle\int_{t'=0}^{t} \vec{a}(t') \cdot dt' \\[2ex] \mathbf{p}(t) = \mathbf{p}_0 + \displaystyle\int_{t'=0}^{t} \vec{v}(t') \cdot dt' \end{cases}$$

41

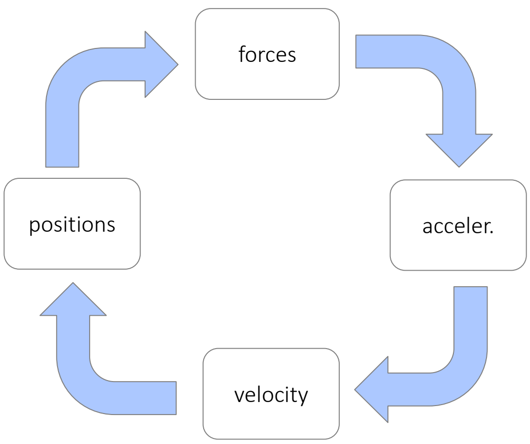
Newtonian Dynamics: equivalent formulation

derivative w.r.t. time

$$\begin{cases} \vec{f}(t) = \text{function}(\, \mathbf{p}(t)\, , \ldots) \\ \vec{v}(t) = \dot{\mathbf{p}}(t) \\ \vec{a}(t) = \ddot{\mathbf{p}}(t) = \dfrac{\vec{f}(t)}{m} \\ \dot{\mathbf{p}}(0) = \vec{\mathrm{v}}_0 \\ \mathbf{p}(0) = \mathrm{p}_0 \end{cases}$$

42

Dynamics (Newtonian)



43

An obvious remark, but

Simulation time ≠ Wall time

the t in all the slides

They are just artificially made to flow in sync… usually

- But (e.g.) not when:
  game is paused (t is constant), replays, fast forwards, reverses…

44

An obvious remark, but

Simulation time ≠ Wall time

the t in all the slides

Occasionally, the difference is spectacularly exploited by clever gameplay designs!

PoP – the sands of times
(Ubisoft, 2003)

Braid
(Jonathan Blow, 2008)

The longing
(Studio Seufz, 2020)

47

Computing physics evolution

- Analytical solutions:

  state = function(t)

  Given force functions (and acc), find the functions (pos, vel,...) in the specified relations:

$$\begin{cases} \vec{f}(t_C) = funz(p(t_C),\ldots) \\ \vec{a}(t_C) = \vec{f}(t_C)/m \\ \vec{v}(t_C) = \vec{v}_0 + \int_0^{t_C} \vec{a}(t)\cdot dt \\ p(t_C) = p_0 + \int_0^{t_C} \vec{v}(t)\cdot dt \end{cases}$$

- Numerical solutions:

  1. state$_{(t=0)}$ ← init
  2. state$_{(t+1)}$ ← do_1_step(state$_t$)
  3. goto 2

48

Analytical solutions

that is, a trajectory: a position over time

Find the positions as a function $\mathbf{p}(t)$ of time t such that...

a given function

derivative w.r.t. time

$$\begin{cases} \ddot{\mathrm{p}}(t) = \mathrm{forces}(\,\mathrm{p}(t),\ldots)/\mathrm{m} \\ \dot{\mathrm{p}}(0) = \vec{\mathrm{v}}_0 \\ \mathrm{p}(0) = \mathrm{p}_0 \end{cases}$$

sometimes, it's a function of other things too (e.g. velocity, time...). Harder to solve!

the initial conditions (for speed and position)

A system of ODE
(Ordinary Differential Equations)

51

Analytical solution

- Difficult to find
  - a function such that...
- Often, it doesn't even «exist»
  - in a form that we can write using common functions such as polinomials, algebraic functions, exponential trigonometry, etc
- When it exists, they are very convenient
  - we can find the position / the velocity for any given *t*
  - we can predict the status of the simulation for any given time
- Examples of systems that admit an analytical solution:
  - systems with a force function is constant w.r.t. positions & velocities (solution: just find its integral, twice)
  - two bodies (no more than two!), subject to reciprocal gravity force
  - a single pendulum, if one accepts an approximation (only good for small oscillations)
- Most other systems don't!

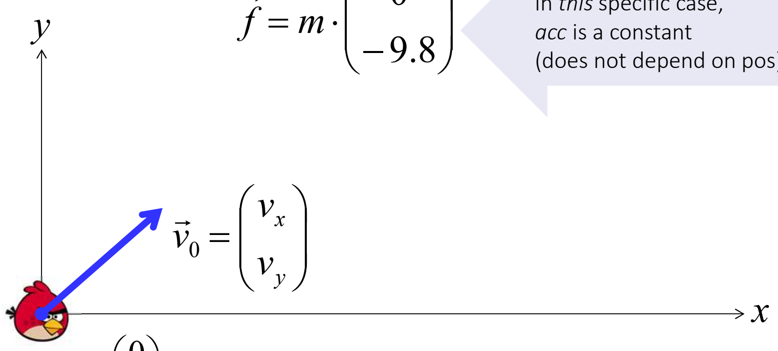
52

Simple example: analytical solution

«ballistic shooting» of a mass, in 2D, ignoring friction...

$$\vec{f} = m \cdot \begin{pmatrix} 0 \\ -9.8 \end{pmatrix}$$

in *this* specific case, *acc* is a constant (does not depend on pos)



$$\vec{v}_0 = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

$$p_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

53

Simple example: analytical solution

Solving…

$$\vec{f}(t_C) = fun(p(t_C),\ldots)$$
$$\vec{a}(t_C) = \vec{f}(t_C)/m$$
$$\vec{v}(t_C) = \vec{v}_0 + \int_0^{t_C} \vec{a}(t)\cdot dt$$
$$p(t_C) = p_0 + \int_0^{t_C} \vec{v}(t)\cdot dt$$

$$\vec{f}(t_C) = m \cdot \begin{pmatrix} 0 \\ -9.8 \end{pmatrix}$$

$$\vec{a}(t_C) = \vec{f}(t_C)/m = \begin{pmatrix} 0 \\ -9.8 \end{pmatrix}$$

$$\vec{v}(t_C) = \begin{pmatrix} v_x \\ v_y \end{pmatrix} + \int_0^{t_C} \begin{pmatrix} 0 \\ -9.8 \end{pmatrix} \cdot dt = \begin{pmatrix} v_x \\ v_y - 9.8 \cdot t_C \end{pmatrix}$$

$$p(t_C) = p_0 + \int_0^{t_C} \vec{v}(t)\cdot dt = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \int_0^{t_C} \begin{pmatrix} v_x \\ v_y - 9.8 \cdot t \end{pmatrix} \cdot dt = \begin{pmatrix} v_x \cdot t_C \\ v_y \cdot t_C - 9.8/2 \cdot t_C^{\,2} \end{pmatrix}$$

54

Simple example: analytical solution

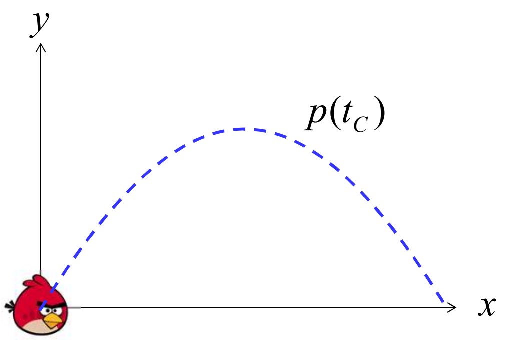
Final result:

$$\vec{f}(t_C) = m \cdot \begin{pmatrix} 0 \\ -9.8 \end{pmatrix}$$

$$\vec{a}(t_C) = \begin{pmatrix} 0 \\ -9.8 \end{pmatrix}$$

$$\vec{v}(t_C) = \begin{pmatrix} v_x \\ v_y - 9.8 \cdot t_C \end{pmatrix}$$

$$p(t_C) = \begin{pmatrix} v_x \cdot t_C \\ v_y \cdot t_C - 9.8/2 \cdot t_C^{\,2} \end{pmatrix}$$



55

Numerical integration

$$\vec{f}(t_C) = \text{function}(p(t_C), \dots)$$
$$\vec{a}(t_C) = \vec{f}(t_C)/m$$
$$\vec{v}(t_C) = \vec{v}_0 + \int_0^{t_C} \vec{a}(t) \cdot dt$$
$$p(t_C) = p_0 + \int_0^{t_C} \vec{v}(t) \cdot dt$$

It's our way to solve the ODE

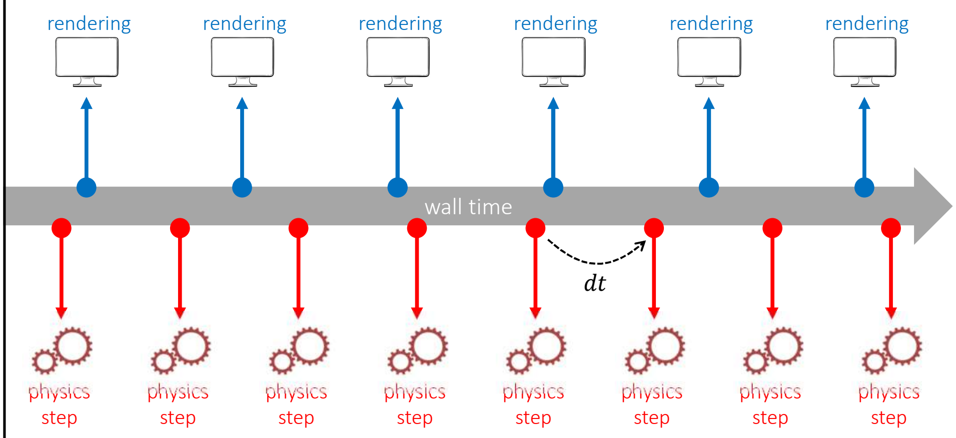
56

Numerical integration

- A numerical integrator computes the integral as summed area of small rectangles
  - For a physics engine, this means just updating velocity and positions at each physics step
- A crucial parameter is the width of the rectangles i.e. *dt* = the duration of the physics step (in virtual time)
  - If physics system perform *N* steps per second:
    dt = 1.0 sec / N
  - *N* is not necessarily same rendering frame rate
    e.g.: rendering 30 FPS but physics: 60 steps per seconds
  - *dt* is not necessarily constant during the simulation
    (but in most system, it is)

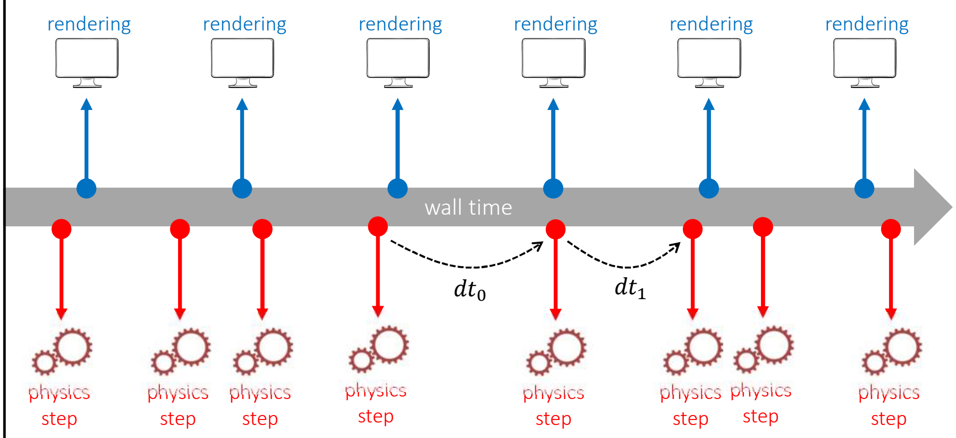
57

Rendering *Frames-per-Seconds* (FPS) vs Physics *Steps-per-Seconds*



58

Timesteps can be variable for each frame



60

Numerical methods: features

- How efficient / expensive
  - must be at least soft real-time
    - (if from time to time computation delayed to next frame, ok)
- How accurate
  - must be at least plausible
    - (if stays plausible, differences from reality are acceptable)
- How robust
  - rare completely wrong results
    - (and never crash)
- How generic
  - Which phenomena / constraints / object types is it able to recreate?
  - requirements depend on the context (ex: gameplay)

61

Euler integration methods

For each step:

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

(1) Evaluate the force
on each particle
as a function of positions
(of this and/or other particles)
and any other things needed things too

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

(2) acceleration
of each particle given by:
total force actiung on it divided by its mass

$$p = p_0 + \int \vec{v} \cdot dt$$

(3) Update position with velocity

$$\vec{v} = \vec{v}_0 + \int \vec{a} \cdot dt$$

(4) Update velocity with acceleration

green = state variables
blue = temp variables

62

Euler integration methods

init

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

one step

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

$$t = t + dt$$

63

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…

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

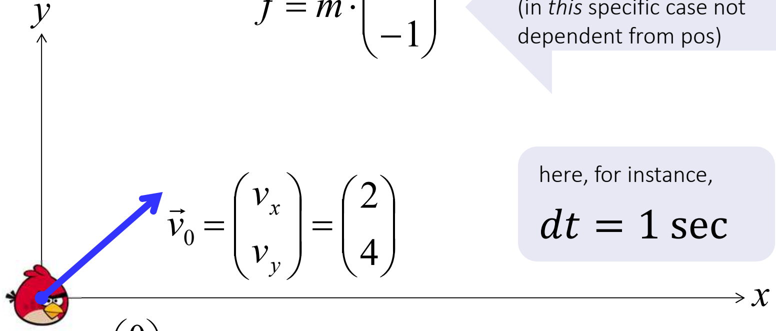
64

Simple example: numerical solution

Same phenomena
of previous example

$$\vec{f} = m \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

constant
(in *this* specific case not dependent from pos)



$$\vec{v}_0 = \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

$$p_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

here, for instance,

$$dt = 1 \text{ sec}$$

65

Simple example: numerical solution (with dt=1 sec)

init

Time:	0	1	2	3	4	5	6	7	...
vel:	(2,3)	(2,2)	(2,1)	(2,0)	(2,-1)	(2,-2)	(2,-3)	(2,-4)	...
pos:	(0,0)	(2,3)	(4,5)	(6,6)	(8,6)	(10,5)	(12,3)	(14,0)	...

step step step step step step step step

$$\vec{f} = m \cdot \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

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

$$\vec{v} = \vec{v} + \vec{a} \cdot dt$$

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

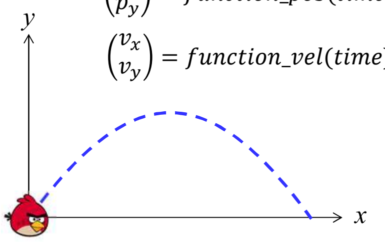
66

Physics evolution computation

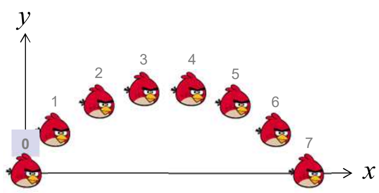
- Analytical solutions:

$$\begin{pmatrix} p_x \\ p_y \end{pmatrix} = function\_pos(time)$$

$$\begin{pmatrix} v_x \\ v_y \end{pmatrix} = function\_vel(time)$$



- Numerical solutions:



67

Physics evolution computation

- Analytical solutions:
  - Super efficient!
    - Close form solution
  - Accurate
  - Only simple systems
  - Formulas found case by case (often they don't even exist)
  - NOT USED (but, for instance, useful to to make predictions for, e.g. A.I.)
- Numerical solutions:
  - Expensive (iterative)
    - but *interactive*
  - Integration errors
  - Flexible
  - Generic
  - USED FOR DYNAMICS

68

Integration errors

- A numerical integrator only approximates the actual value of the integrals
- The discrepancy (simulation errors) accumulates with virtual time during all the simulation
- How much error is accumulated?
- It depends on dt
  - smaller $dt \Rightarrow$ smaller error (simulation is more accurate)

  but, clearly
  - smaller $dt \Rightarrow$ more steps are needed
    (for simulate the same virtual time)
    $\Rightarrow$ simulation is more computationally expensive,
    but smaller errors,

69

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),
  - The Euler seen 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

70

The integration step dt of any numerical methods (summary)

dt : delta of virtual time from last step

- the "temporal resolution" of the simulation!

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

- 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 ≠ FPS of physics. Rendering can be *less*!)
  - note: dt is not necessarily the same in all physics steps (need more accuracy *now*? Decrease *dt*

71

Effect of integration errors of System Energy

- Because of integration errors:
  simulated solutions ≠ "real" solutions
- In a real system, the total energy can never increase
  - typically, 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 wider and wider
- Particularly bad because:
  - compromises stability
    (velocity = big, displacements = crazy, error = crazy)
  - compromises plausibility
    (we can *see* it's wrong)
- A simple way to avoid this:
  make sure the simulation always includes attritions
  - makes simulation more stable + robust

72

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 (next lecture)
  - Verlet method (next lecture)
- 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)

73

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

74

Forward Euler

init state

$\mathbf{p} \leftarrow \cdots$

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

one step

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

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

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

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

$t = t + dt$

75

Symplectic Euler

init state

$\mathbf{p} \leftarrow \cdots$

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

one step

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

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

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

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

$t = t + dt$

76

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…

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

77

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 );
}
```

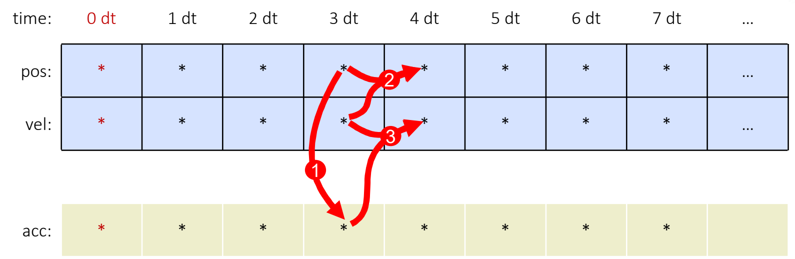
just flip the order

Equivalent to…

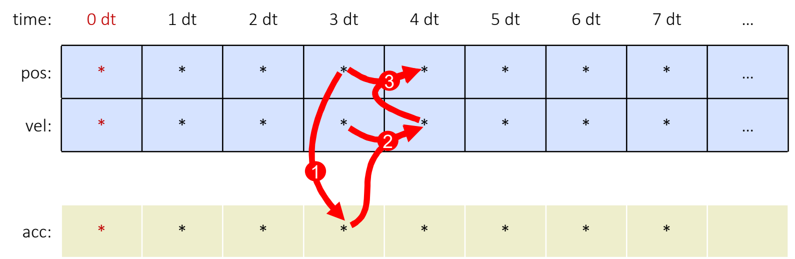
$$\vec{f}_i \leftarrow function(p_i, \ldots)$$
$$\vec{a}_i \leftarrow \vec{f}/m$$
$$\vec{v}_{i+1} \leftarrow \vec{v}_i + \vec{a}_i \cdot dt$$
$$p_{i+1} \leftarrow p_i + \vec{v}_{\boldsymbol{i+1}} \cdot dt$$

78

Forward Euler:



Symplectic Euler:



79

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 1 ☹)
  - On average, better behavior:
    more stable and accurate

81

Forces: examples

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

Primarily, a function of the positions

But not always, and sometimes not only of positions (also: velocities? Global time?)

- Gravity
  - Constant · *m*, near the surface of a planet
  - Function of positions in a space simulation
- Wind pressure
  - Depends on the area exposed in the wind direction
- Electrical / magnetic forces
- Buoyancy *(ita: forza di Archimede)*
  - 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

82

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

83

Example of forces: gravitational force on a plantet surface

- Given a particle with (gravitational) mass m

some global constant dependent on... the planet

$$\vec{f_a} = \underbrace{g\, m}_{\text{force magnitude (positive scalar)}} \; \hat{d}_{\text{DOWN}}$$

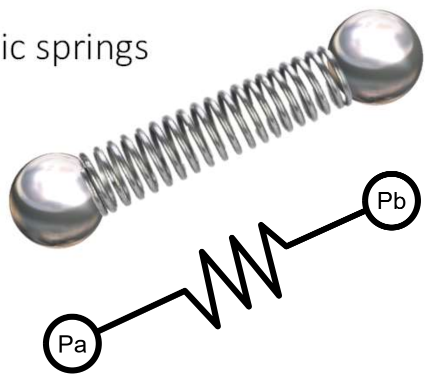
$\hat{d}_{\text{DOWN}}$: force direction (versor)

Notes:
- does not depend on position, only on mass
- will produce a constant acceleration (regardless of mass!) when divided by (inertial) mass m

84

Forces: Springs (Hooke's law)

- Simplified model for elastic springs
- One spring connects two particles in $\mathbf{p}_a$ and $\mathbf{p}_b$
- Characterized by:
  1. Rest length ℓ
  2. Stiffness k
- Spring force: counteracts expansion and compression



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

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

86

Forces: Springs (Hooke's law)

$$\vec{f_a} = \underbrace{k(\overbrace{\|\mathbf{p}_b - \mathbf{p}_a\| - \ell}^{\text{elongation / compression}})}_{\substack{\text{force magnitude} \\ \text{(scalar)} \\ \text{(positive or negative)}}} \underbrace{\frac{\mathbf{p}_b - \mathbf{p}_a}{\|\mathbf{p}_b - \mathbf{p}_a\|}}_{\substack{\text{force direction} \\ \text{(versor)}}}$$

force to be applied to particle a

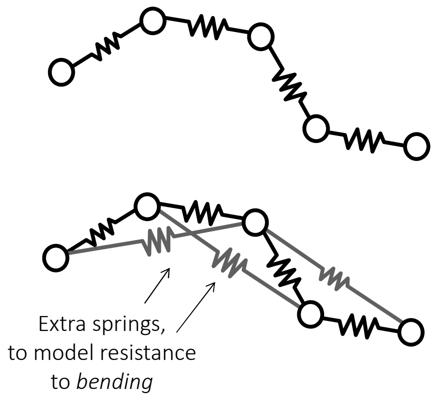
$$\vec{f_b} = - \vec{f_a}$$

force to be applied to particle b

87

Mass and Spring systems

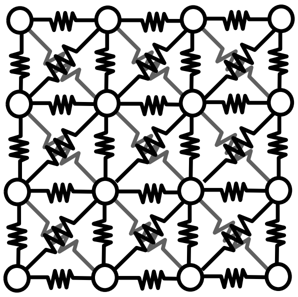
- Useful for deformable objects
- for instance: elasitic ropes (or hairs)

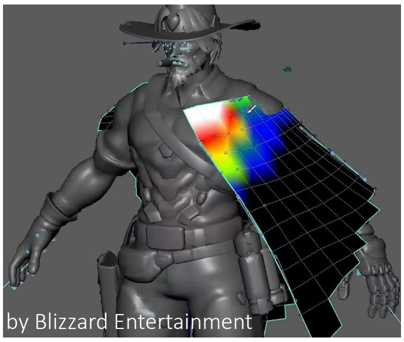


89

Mass and Spring systems

- For instance: cloth





90

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 ? (no they can't)
  - Increase spring stiffness? $k \to \infty$
  - Makes sense, physically, but…
  - Large $k \Rightarrow$ large $\mathbf{f} \Rightarrow$ instability $\Rightarrow$ unfeasibly small dt needed
  - Doesn't work. How, then? see later

91