3D video games

Game Physics

Marco Tarini

Animation in games

but, a caveat on terminology:
in some context 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 repetition free

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
  - (not necessarily *realism*)
- robustness
  - (should almost never "explode")

Physics engine: intro

- Game engine module
  - executed at game run time
- An high-demanding computation
  - on a very limited time budget!
- ...but highly parallelizable
  - "embarrassingly parallel"

==> good fit for hardware support

(just like the Rendering Engine)

Game engine tasks:
(Physics simulation!)

- Dynamics (Newtonian) for object types such as:
  - Rigid bodies
  - Soft bodies
    - "ragdolling"
  - Free-form deformation bodies: Specific solutions for
    - Ropes
    - Cloth
    - Hair…
  - Fluids
  - Air (e.g. wind) etc
- Collision handling
  - Collision detection
  - Collision response

Hardware for Physics engine

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

- Recently: PPU
  - "Physical Processing Unit"
  - HW unit specialized on physics
- *More* recently: GP-GPU
  - "General Purpose Graphics Processing Unit"
    - Use of the graphics card for generic tasks (not related with 3D computer graphics)
  - Ex.: Cuda (nVidia)

Software: libraries / SDK

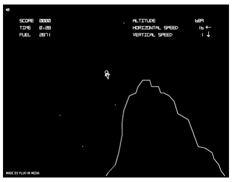
- PhysX — HW accelerated (CUDA)
- havok — HW accelerated (OpenCL) → source by VALVE
- BULLET PHYSICS LIBRARY — open source, free, HW accelerated
- OPEN DYNAMICS ENGINE — open source, free
- farseer physics — 2D! open source, free

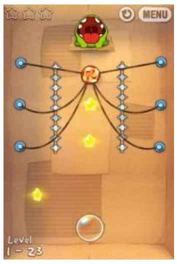
PhysX

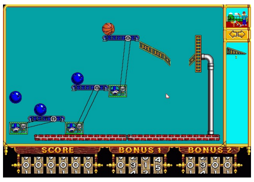
ETH → NovodeX → ageia → NVIDIA → PhysX by NVIDIA → unity, UNREAL ENGINE, Gamebryo, …

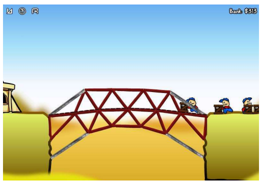
Physics in games: cosmetics or gameplay?

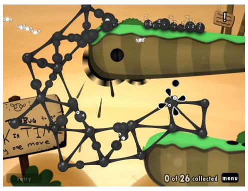
- Just a graphic accessory? (for realism!)
  - e.g.:
    - particle effects (w/o feedback)
    - secondary animations
    - Ragdolling
- Or a gameplay component?
  - e.g. physics based puzzles
  - Popular approach in 2D (since always!)





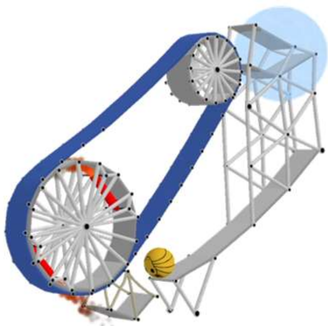


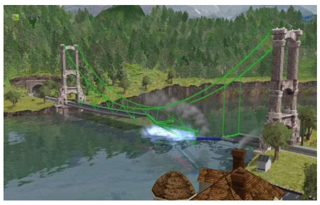




Physics in games: cosmetics or gameplay?

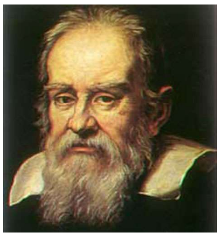
- Just a graphic accessory? (for realism!)
  - e.g.:
    - particle effects (w/o feedback)
    - secondary animations
    - Ragdolling
- Or a gameplay component?
  - e.g. physics based puzzles
  - Rising trend in 3D





Physics engine:
Dynamics

- Physics simulation (Newtonian)
  - Revision:
    - object = mass
    - Object state:
      - position and derivative: velocity
        - (and momentum)
      - direction and angular velocity
        - (and angular momentum)
    - State change:
      - forces => acceleration,
        torque



Reminder:
Spatial location of an object

2D Physics

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

3D Physics

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

Newtonian dynamics: summary

Actual object location	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 $\vec{v}$ $\vec{v} = \dot{p}$ ($\lvert\vec{v}\rvert$ = "speed")	Momentum $\vec{v} \cdot m$	Acceleration $\vec{a} = \dot{\vec{v}} = \ddot{p}$	Force $\vec{f}$ $\vec{f} = \vec{a} \cdot m$
Orientation (e.g. quaternion)	Angular velocity $\vec{\omega}$	Angular momentum $\vec{\omega} \cdot I$ I = moment of inertia (for axis) ("rotational inertia")	Angular acc. $\vec{\alpha}$	Torque $\vec{\tau}$ $\vec{\tau} = \vec{\alpha} \cdot I$ ("mechanic momentum")

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

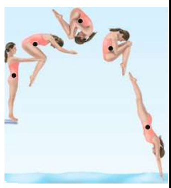
Change the state
(no memory)

A few constants per object

A few quantities associated to each object
- constants: they don't (usually) change
- input of the physical simulation, not output

- Mass:
  - resistance to change of velocity

- Moment of Inertia:
  - resistance to change of *angular* velocity



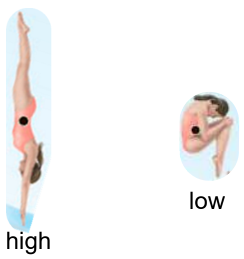
- Barycenter:
  - the center of mass

Mass

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

Moment of inertia

- Resistance to change of angular velocity



- (an object rotates around its barycenter)

Moment of inertia

- Scalar moment of inertia
  - Resistance to change of angular velocity
  - Depends on the mass, and on its *distribution*
    - the farthest one sub-mass from the axis, the > the resistance
  - In 3D: its different for each axis of rotation
    - It can be computed for any axis, thanks to…
- Moment of inertia as a 3x3 Matrix
  - a matrix **A** used to extract the scalar, for any given axis
  - given an axis **a** (**a** = unit vector), the *moment of inertia* is

$$a^T A a$$

  - matrix **A** can be computed once and for all for a rigid object
    - how: that's beyond this course
    - in practice: use given formulas for common shapes
    - or sum the contributions for each sub-mass

Barycenter

- Aka the center of mass
  - (a position)
- In the discrete setting:
  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)

State of an object

```
Point position
Rotation orientation
```
current

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

updated by physics

```
Scalar mass
Matrix moment_of_inertia
Point barycenter
Scalar drag
…
```
constants

frictions; see later (drag)

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

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

In unity

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

```
Vector velocity
Rotation angular_velocity
Scalar mass
Matrix moment_of_inertia
Point barycenter
Scalar drag
…
```
the **RigidBody** component

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

In Unity (using Unity terminology)

```
Vector3 position
Quaternion rotation
```

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

part of **Transform** component

```
Vector3 velocity
Quaternion angularVelocity
float mass
Vector3 inertiaTensor
Quaternion inertiaTensorRotation
Vector3 centerOfMass
float drag
…
```

note: speed = velocity.magnitude

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 object space)

State of a point-particle

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

not used !

One trend in game phys engines is to simulate point-particles only.

Much simpler!

E.g. no rotation needed!

We will see later how to still get complex objects back (with “PBD”)

For now,
we focus on this simpler case.

Dynamics (Newtonian)

describe the forces
given the particle positions (and more)

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

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

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

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

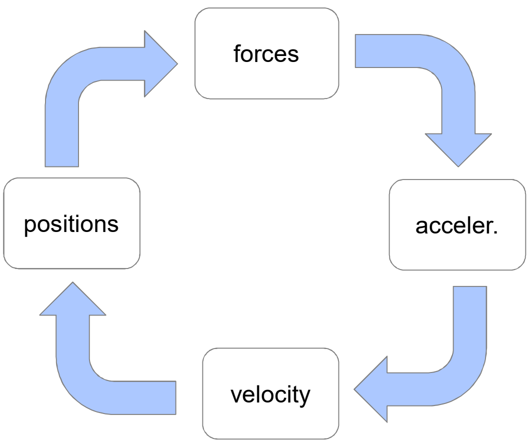
Dynamics (Newtonian)

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

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

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

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



An (obvious) precisation

- t_C = virtual time != real time
  - e.g.:
    - game paused → t costant.
    - Fast forward, replay, rallenty, reverse → change of speed/flow direction of t

occasionally,
gameplay exploit this difference in spectacular ways!


PoP – the sands of times serie (Ubisoft, 2003-…)


Braid (Jonathan Blow, 2008)

Computing physics evolution

- **Analytical** solutions:

  state = function(t)

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

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

- **Numerical** solutions:

  1. $\text{state}_{(t=0)}$ ← init
  2. $\text{state}_{(t+1)}$ ← evolve(state_t)
  3. goto 2

Analytical solutions

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

$$\vec{a}(t_C) = \vec{f}(t_C) / m$$

$$\vec{v}(t_C) = \vec{v}_0 + \int_0^{t_C} \vec{a}(t) \cdot dt$$

$$p(t_C) = p_0 + \int_0^{t_C} \vec{v}(t) \cdot dt$$

pos, acc, vel, forces: in function of current time t_C

Analytical solutions

that is, find position as function p of time s.t.

$$\ddot{p}(t) = \text{function}(\,p(t)\,) / \mathrm{m}$$

with

$$\dot{p}(0) = \vec{v}_0$$

$$p(0) = \mathrm{p}_0$$

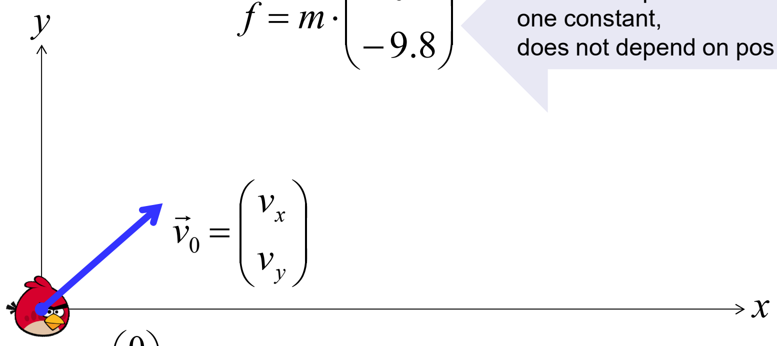
sometimes, of other things too (e.g. velocity). Even harder!

Simple example: analytical solution

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

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

a is in *this* specific case:
one constant,
does not depend on pos



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

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

Simple example: analytical solution

Solving...

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

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

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

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

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

Simple example: analytical solution

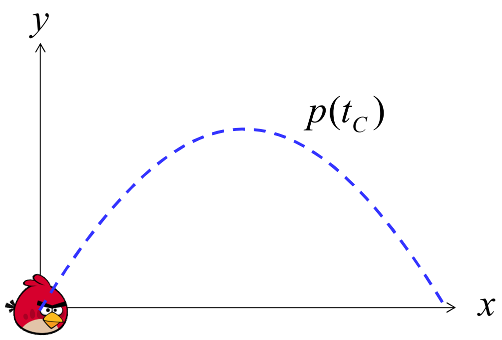
Final result:

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

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

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

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



Some numerical methods

- Forward Euler method
  - (simple and direct)
- Leapfrog method
- Verlet method
  - (position based dynamics)

Numerical method features

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

Euler method integration

For each step:

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

(1) Evaluate the **force** (on each particle) as a function of **position** (even of other particles)

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

(2) **acceleration** of each particle given by: **forces** on it and its mass

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

(3) Update **velocity** with **acceleration**

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

(4) Update **position** with **velocity**

(state / variables) , **(temp variables)**

Euler methods

init state:

$$p = \dots$$
$$\vec{v} = \dots$$

one step:

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

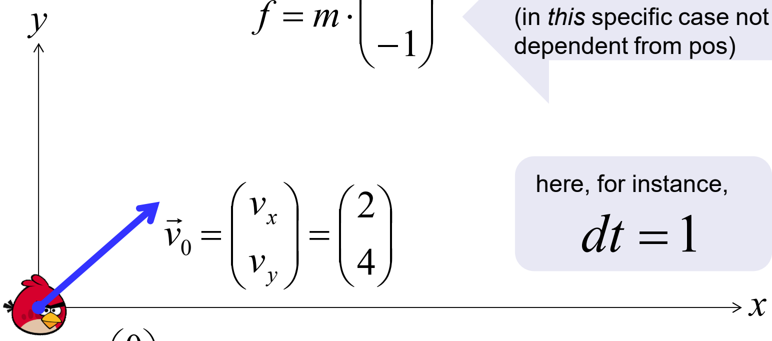
$$t = t + dt$$

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)



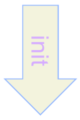
$$\vec{v}_0 = \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

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

here, for instance,

$$dt = 1$$

Simple example: numerical solution



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

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

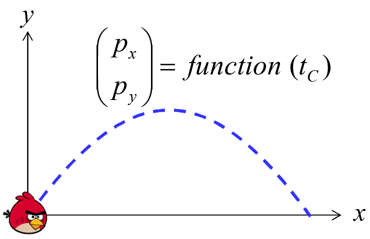
$$\vec{a} = \vec{f} \,/\, m$$

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

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

Physics evolution computation

- Analytical solutions:
- Numerical solutions:



Physics evolution computation

- Analytical solutions:
  - Super efficient!
    - Close form solution
  - Accurate
  - Only simple systems
  - formulas found case by case
    (often not existing!)
  - NO
    (but, for instance, useful to allow the AI to make predictions)
- Numerical solutions:
  - Expensive (iterative)
    - but *interactive*
  - Integration errors
  - Flexible
  - Generic
  - YES

Integration erros

- Depends on *dt*
  - Small *dt* ==> more steps needed (for same virtual time) ==> more computationally expensive, but smaller error, i.e. more accurate simulation
    (smaller difference with exact analytical solution)
  - *dt* = 1.0 sec / FPS of physics simulation
    - (recall: not necessarily same rendering frame rate)
  - How much does error decreasing when *dt* decreases?
    - that «Order» of the simulation
    - Euler is 1st order: the error can be as bad as $O(dt^1)$
      (but usually not that bad)
- Error keeps on accumulating with time
  - (dependent also from t_{tot})

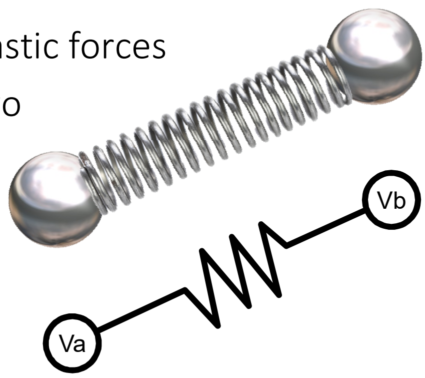
Forces

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

- In general, a function of current position(s)
  - Gravity
  - Resistance of solid materials
    - but, this can be accounted for using constraints… see later
  - 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
    - oops! also depend on speed
    - luckily, they can be accounted for using *damping* – see later

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 – dist(Va,Vb))

The force f exerted on Vb is –*f*