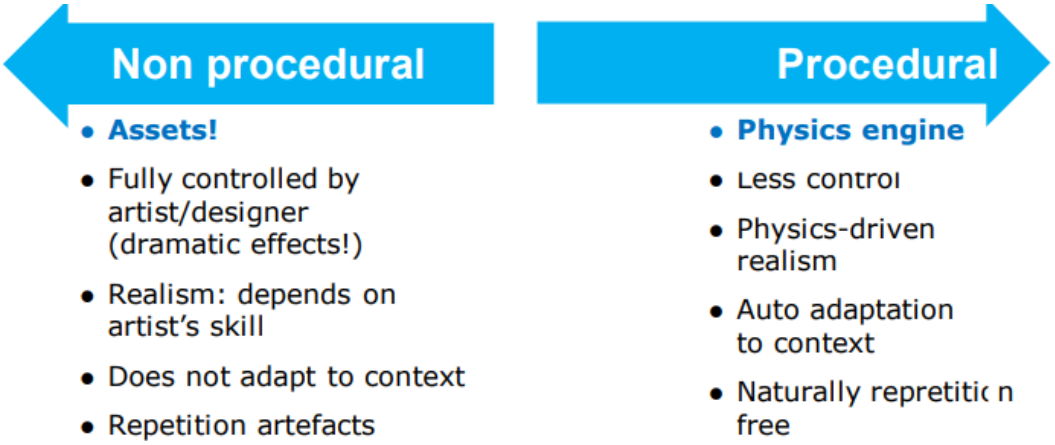
Lezione 15 15/05/2025

Le animazioni possono essere fatte con:

- **metodi non procedurali**, ovvero fatte da un artista, che sposta i punti manualmente (tramite dei tools). Il risultato dipende dall'abilità dell'artista.
- **metodi procedurali**, seguono la fisica, o un'approssimazione della fisica. Sono effettuate tramite l'engine fisico. Non si può quindi determinare dove finiranno i punti, dipende dalla fisica.



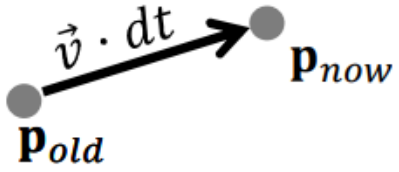
Le simulazioni fisiche ci sono sia nel 3D che nel 2D, sono in real time. Hanno la caratteristica di essere percepite come plausibili, ma non descrivono la fisica vera.

In molti casi la fisica è parte del gioco

L'engine fisico si occupa di fare quei calcoli (molto velocemente) che riguardano gli aspetti fisici.

Verlet integration method

- Idea: remove velocity from state
  Instead, store previous position
- Velocity is now implicit
- It's defined by:
  - current pos $\mathbf{p}_{now}$
  - last pos $\mathbf{p}_{old}$ which we need to record



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

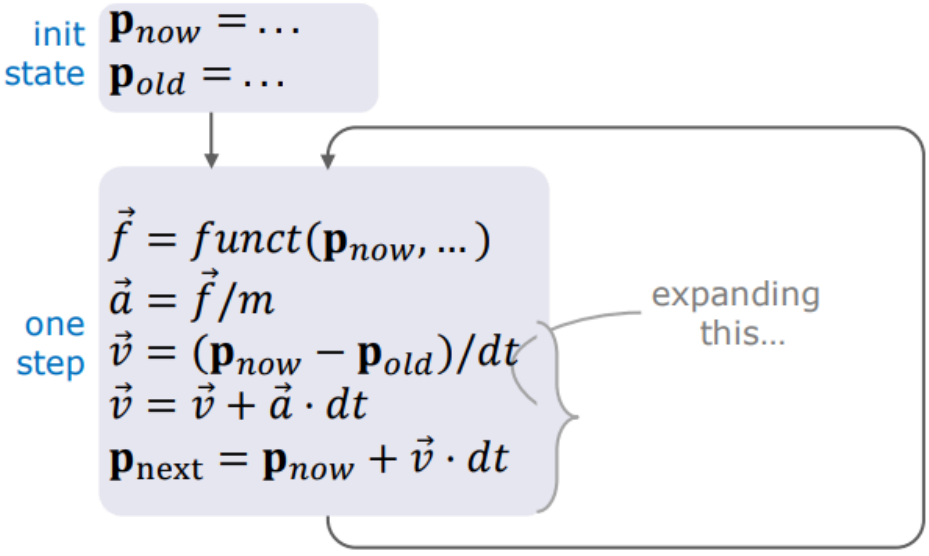
$$\Longrightarrow$$

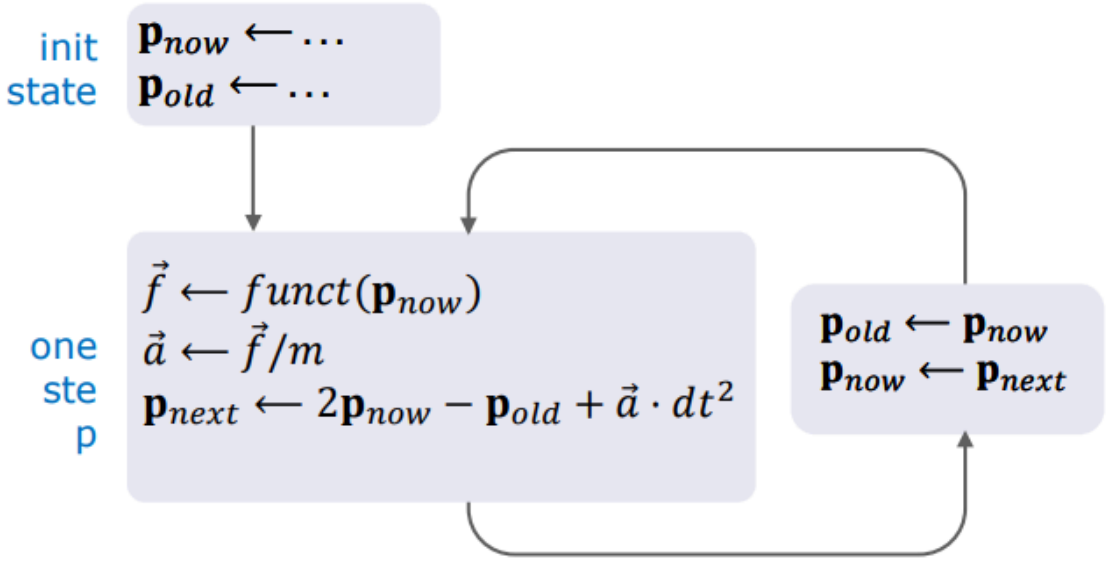
$$\vec{v} = (\mathbf{p}_{now} - \mathbf{p}_{old})/dt \quad \leftarrow \text{Verlet}$$

L'idea è di togliere la velocità dalla parte destra dell'equazione, in questo modo la velocità diventa una funzione delle posizioni.

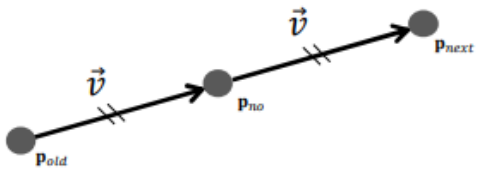
In questo modo v dipende solo dalla posizione attuale e quella precedente.

Quindi se ho come input pnow e pold posso fare una serie di operazioni per avere la forza (non la facciamo), l'accelerazione, ...





Se l'accelerazione è nulla:



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

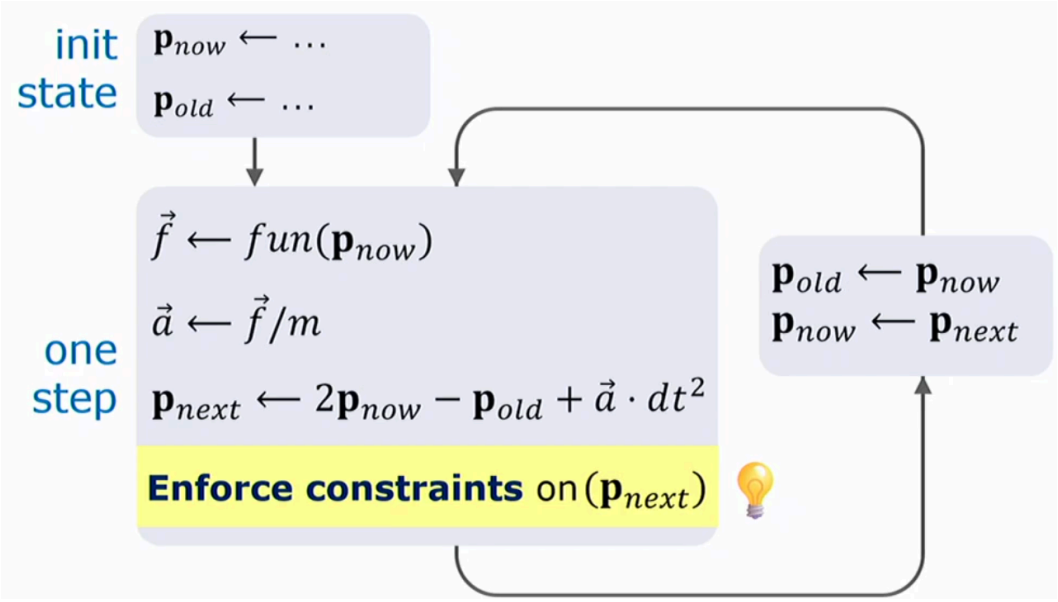
$\mathbf{p}_{next}$ can be written as an **extrapolation** of $\mathbf{p}_{now}$, $\mathbf{p}_{old}$:

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

In questo modo abbiamo un buon compromesso tra efficienza e accuratezza

- Velocity is kept implicit
  - but that doesn't save RAM: we need to store previous position instead
  - (a point instead of a vector: same memory)
- Good efficiency / accuracy ratio
  - 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

Quindi usando verlet abbiamo la "position based dynamics" cioè una dinamica che dipende dai punti (posizioni).



Nei videogiochi si usa questo metodo basato sulle posizioni, perchè è più efficiente.

- A **positional constraint** is an equality/inequality involving the *positions* of particles.
  - Useful, for example, to model consistency conditions
  - Like "*solid objects don't compenetrate each other*", or "*steel bars won't become shorter or longer than they are*"
  - We will see many examples

a formula with '=' '>' '<' etc.

- We **enforce** (impose) positional constraint directly by displacing the *positions* of particles
  - Thanks to Verlet: this displacement automatically causes some appropriate update of the velocity!
  - it's not necessarily correct, but it's plausible and robust

Collision handling

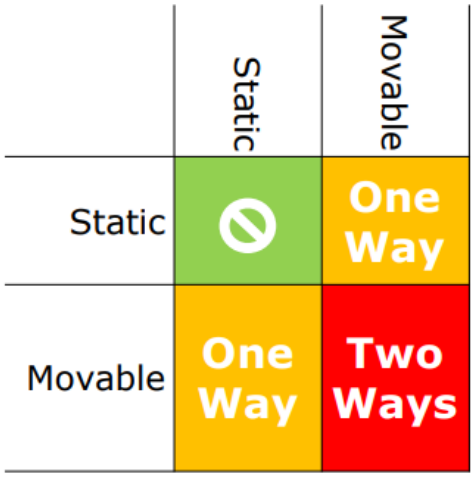
L'altra parte dei motori fisici serve a gestire le collisioni.

Due oggetti che sono nel gioco si scontrano, occupano (parzialmente) lo stesso spazio 3D. Abbiamo 2 tipologie di oggetti che si possono trovare: statici e dinamici.

Quelli **statici** sono quelli che hanno velocità 0, non si muovono. Hanno un effetto sugli altri oggetti, ma gli altri oggetti non hanno effetto su questi.

Quelli **non statici** si possono muovere.

Abbiamo quindi 2 tipi di collisioni:



By labelling every object as static or movable, we reduce the computation considerably!

E.g., if 50% static, 50% movable then…

- 1/4 of the potential collisions cease to exists (*).
  Of the rest:
- 2/3 are **one ways** (easier to handle)
- Only 1/3 are **two-ways**

(*) No collision handling for Static VS static. That's not just an "optimization", but a feature:

- Wall models can compenetrate, to build a house (no collision!)
- Buildings can sink into the terrain (no collision!)
- Etc.

Le collisioni bisogna determinare dove avvengono e stabilire il loro effetto.

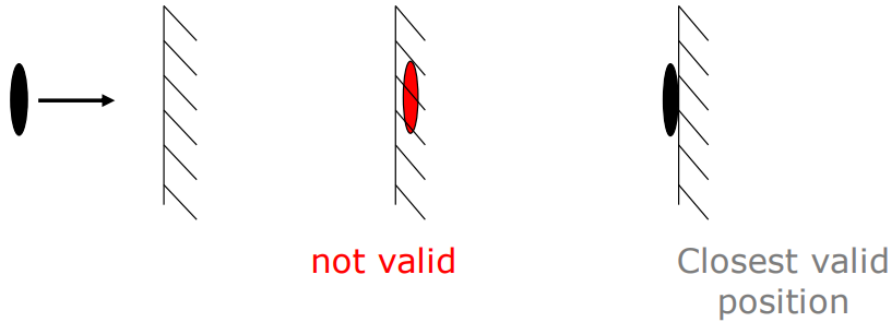
- Enforce **non-penetration**
  - objects must be placed in valid positions — *when to:* **always**
- **Impacts**
  - with impulses (bounces) — *when to:* collision occurred now, but not before
- **Frictions** between the two objects
  - energy dissipation — *when to*: from 2° consecutive step of collision
- **Ad-hoc effects**
  - breaking objects, gameplay effects — *when to - if at all:* entirely gameplay dependent

By scripts

Se 2 oggetti finiscono per occupare lo stesso spazio, si dice che occupano una posizione non valida

◆ Invalid position?

- **strategy 1:** revert to last valid pos (easy to do, not ideal)
- **strategy 2:** project to closest valid pos (necessary, in Position Based dynamics)



◆ In Point Based Dynamics:
just another **positional constraint** ← Note: asymmetrical constraint (> not =)

- bonus: velocity updates (similar to inelastic impacts)
- but we will need to explicitly compute impacts if we want a better control of the behavior (see later)

A practical problem: the existence of the constraint it is **not** known a-priori.

◆ **How to enforce** this constraint:

- *two-ways* :
  displace both of them, minimizing the summed squared displacements × the mass
- *one-way* :
  only displace the one movable objects by the minimal amount
  (equivalent to the above, when fixed object mass → ∞)

Quando due oggetti si scontrano, dobbiamo spostare entrambi, minimizzando la somma dello spostamento dei due per la massa dei due, non vogliamo un effetto dove si genera nuova energia.

Per quanto riguarda la frizione, se ci sono situazioni di collisione che si prolungano nel tempo (più di un frame) allora rappresentiamo la frizione come perdita di velocità dell'oggetto. Per esempio con la collisione con un oggetto statico, l'oggetto perde man mano velocità.

- Apply it on prolonged contact
  - collision with an object that was colliding last frame too
- Affects component of velocity parallel to **contact plane**
- Can be implemented with:
  (1) forces, or (2) velocity damping
- Forces:
  - Opposite to current velocity,
    projected on contact plane (note: I need its normal)
  - Magnitude: proportional to the speed

- **Sudden** velocity change ← so, it's the effect of an **impulse**
  - resolve the impact = determine the new velocities $\vec{v}_{new}$
  - ***equivalently***, determine the impulses $\vec{\imath} = (\vec{v}_{new} - \vec{v}_{old}) \cdot m$

  We will write formulas for whichever is easier to write

  And, for rigid body dynamics: also new angular velocity

- *All* impacts preserve total **momentum** $m \cdot \vec{v}$
  - *Always*, no matter what

  a vector (in italian: *«quantità di moto»*)

- To resolve the impact,we need further assumptions, different for each type of the impact:
  - **elastic**
  - **inelastic**

(completely)
elastic
impact

(completely)
inelastic
impact

Si può modellare l'elasticità tramite la "bounciness"

"Bounciness" = 1.0

...

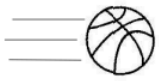
"Bounciness" = 0.5

...

"Bounciness" = **0.0**

Se l'impatto è elastico, non vogliamo perdere energia (se settata a 1).

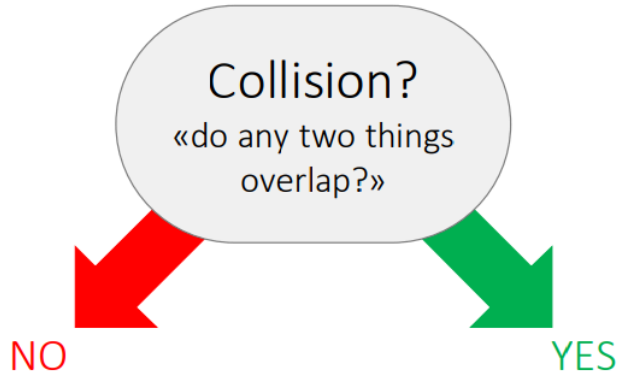
E se invece c'è un impatto?




"Bounciness" = ???

- Practical solution:
  adopt some formula between the
  bounciness values associated to the two objects
  - For example: **avg**, **min**, **max**
  - It's a choice of the game engine
  - (can be hard-wired in the physics engine, or exposed to the users)

Collision detection



Produce:
- a hit **positions**
- a hit **normal**
  - orientation of the impacted place

Dobbiamo dire sia dov'è, ma anche che tipo di collisione è (oggetti statici, non statici, elastici...)

Quali sono i problemi nel collision detection? Deve essere efficiente:

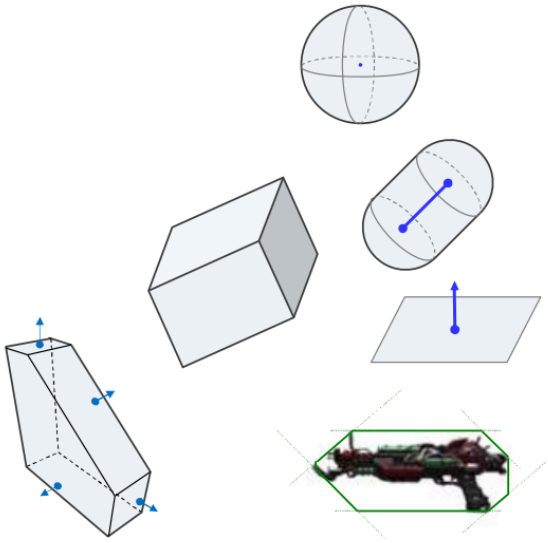
- Observation:
  - almost 100% of the object pairs, almost 100% of the times, **do NOT collide**.
  - for efficiency, the «no-collision» case needs to be optimized
  - «early reject» of the collision test

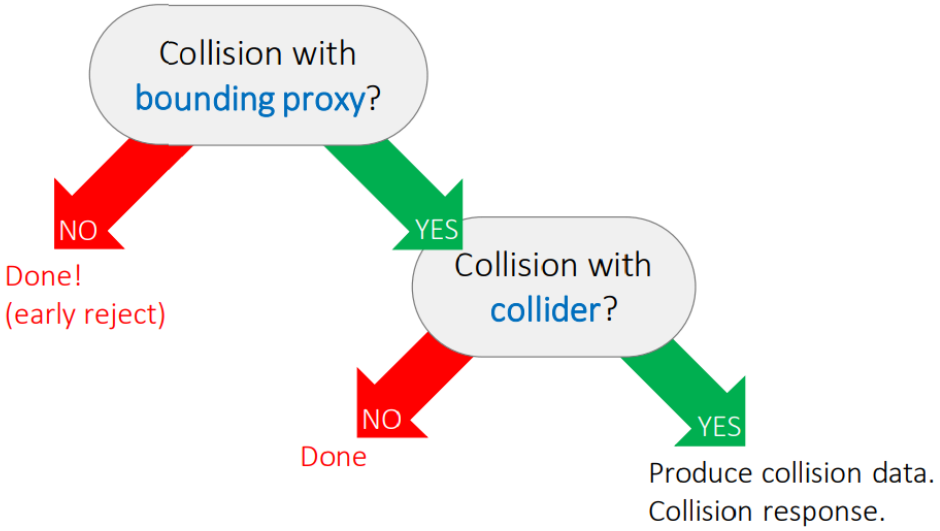
Quindi consideriamo come se non ci sia nessuna collisione di base.

Se ho n oggetti in scena, ho n al quadrato oggetti che possono collidere. Come facciamo a controllare quando due elementi collidono, senza controllare tutte le coppie?

Approssimiamo gli oggetti con dei proxy (bounding volumes, colliders).

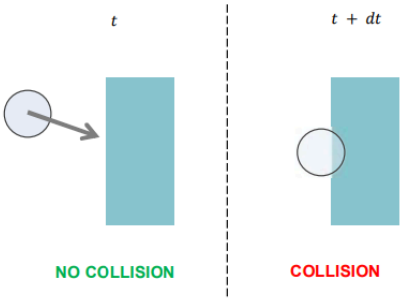
- Spheres
- Capsules
- Half-spaces
- Axis Aligned (Bounding) Boxes
  - aka AABB
- Generic Boxes
- Discrete Oriented Polytopes
  - aka DOP
- Ellipsoids
  - axis aligned or not
- Cylinders
- Convex polyhedrons
- Non-convex polyhedrons
  - Meshes



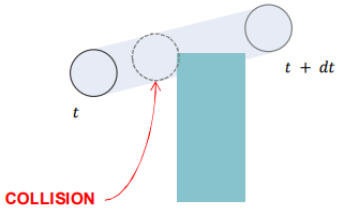


Ci sono diversi tipi di collision detection:

◆**Static** Collision detection
- ("a posteriori", "discrete")
- approximated
- simple + quick



◆**Dynamic** Collision detection
- ("a priori", "continuous")
- accurate
- resource consuming



VR

Nella realtà virtuale vogliamo sostituire gli stimoli percepiti dalla persona con stimoli virtuali che siano realistici.