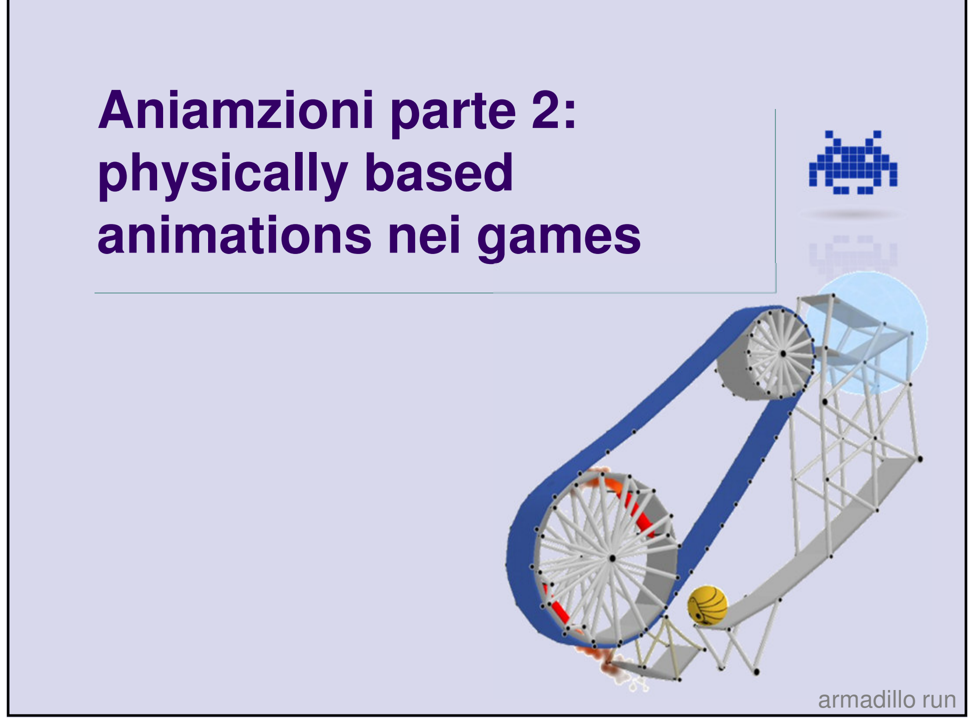
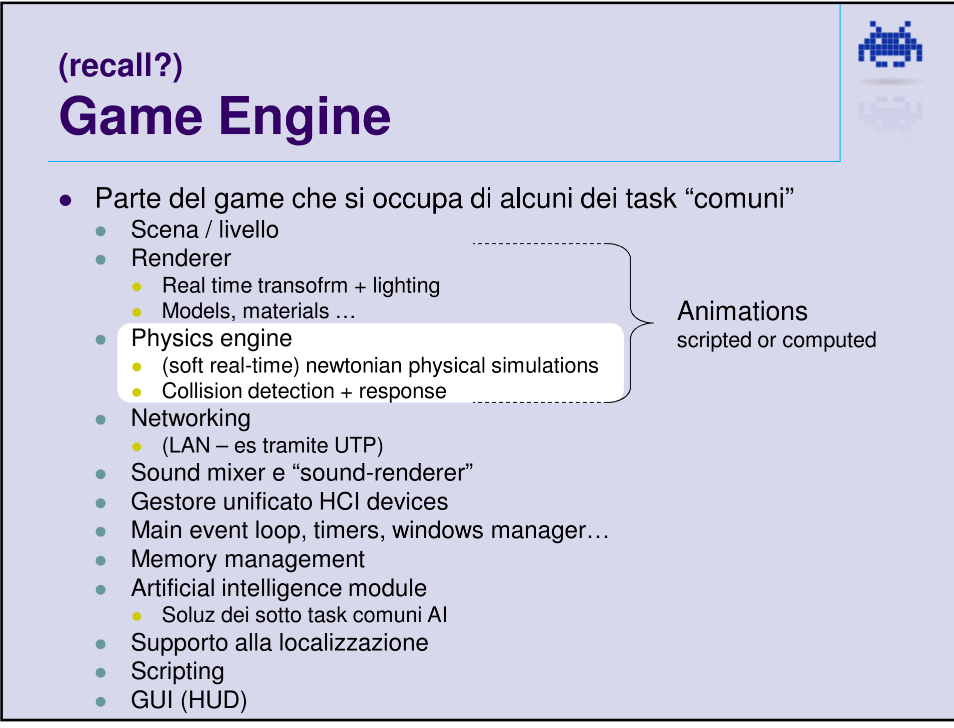


Animazioni nei games

- Scripted
  - Parte degli Assets!
  - Controllo da parte degli artisti (dramatic effects!)
  - Non interattiva
  - Realismo… dipende dall'artista
  - Poca customizzabilità

- Computed
  - Physics engine
  - Poco controllo
  - Interattiva
  - Realismo come prodotto collaterale del rispetto leggi fisiche
  - Si autoadatta



Simulazione evoluz fisica nei video games

- 3D, oppure 2D
- “soft” real-time
- efficienza
  - 1 frame = 33 msec (a 30 FPS)
  - fisica = 5% - 30% max del tempo di computaz
- plausibililità
  - (ma non necessariamente “realismo”)
- robustezza
  - (non deve “scoppiare”… quasi mai)

Physics engine: intro

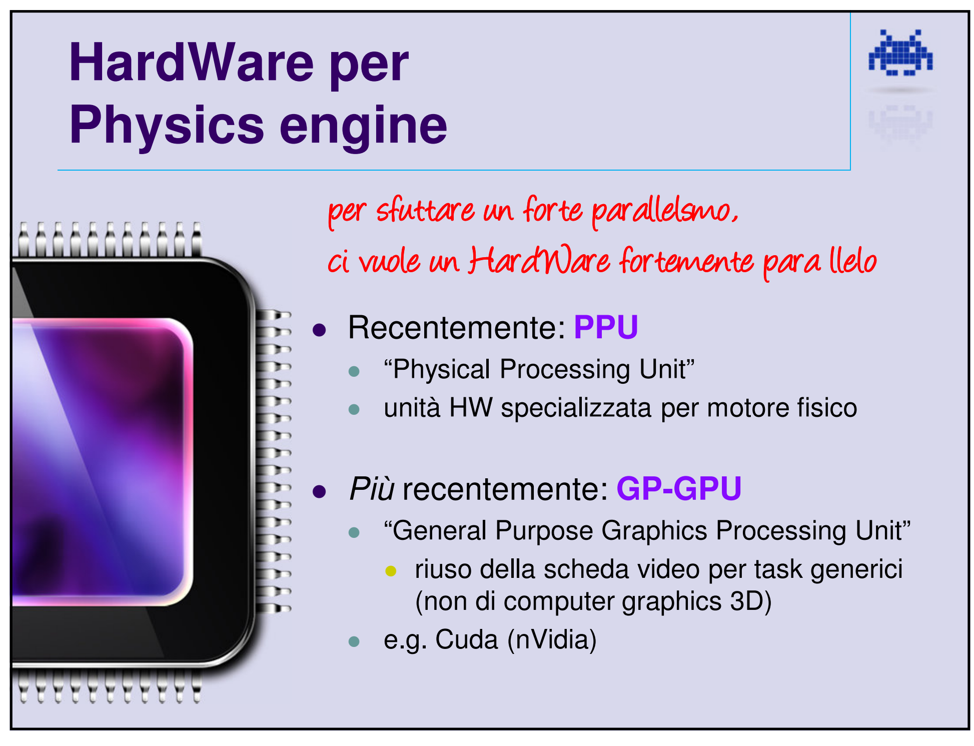
- Modulo del game engine
  - → esegue a tempo di esecuz del game
- Computazione high-demanding
  - (su low-time budgets!)
- Ma, altamente parallelizzabile
  - "embarassignly parallel" ;)
  - → hardware support

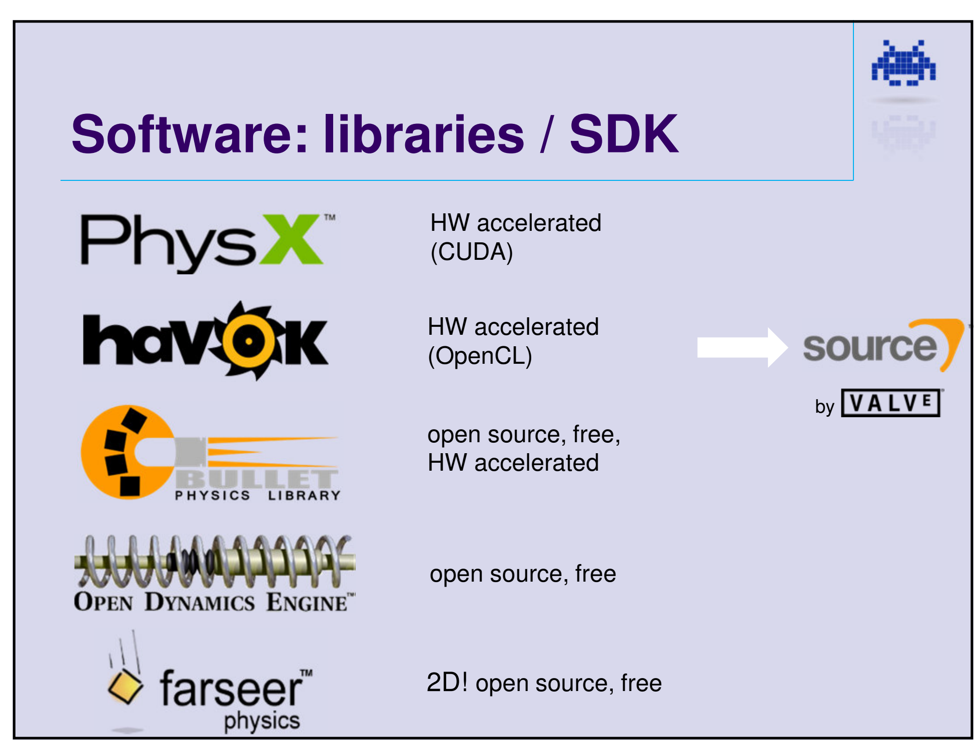
(come il Rendering Engine, del resto)

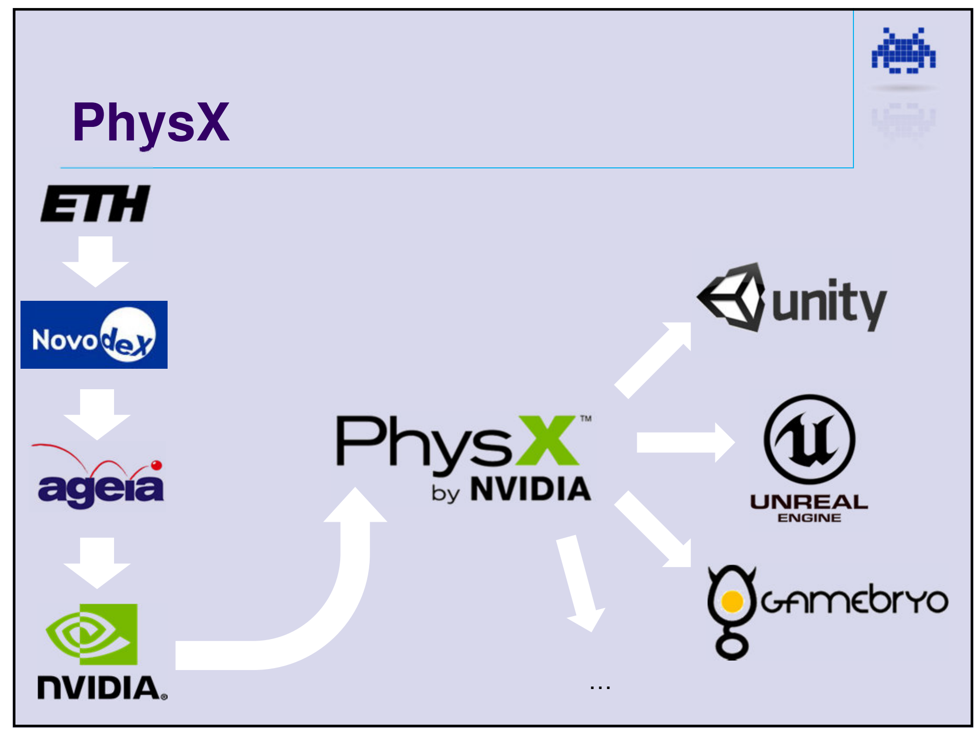
Motore fisico: tasks

Simulatore evoluz fisica del sistema:

- Dynamics (Newtoniana). Classi di oggetti come:
  - Particelle (puntiformi)
  - Corpi rigidi (rigid bodies)
    - Pezzo unico
    - Con giunture
  - Corpi deformabili (soft bodies)
    - Generici
    - Corde
    - Tessuti
    - Molle
    - …
  - Fluidi
  - …
- Collisions
  - Collision detection
  - Collision response


HardWare per
Physics engine
per sfuttare un forte parallelsmo,
ci vuole un HardWare fortemente para llelo
Recentemente: PPU
"Physical Processing Unit"
unità HW specializzata per motore fisico
Più recentemente: GP-GPU
"General Purpose Graphics Processing Unit"
riuso della scheda video per task generici (non di computer graphics 3D)
e.g. 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

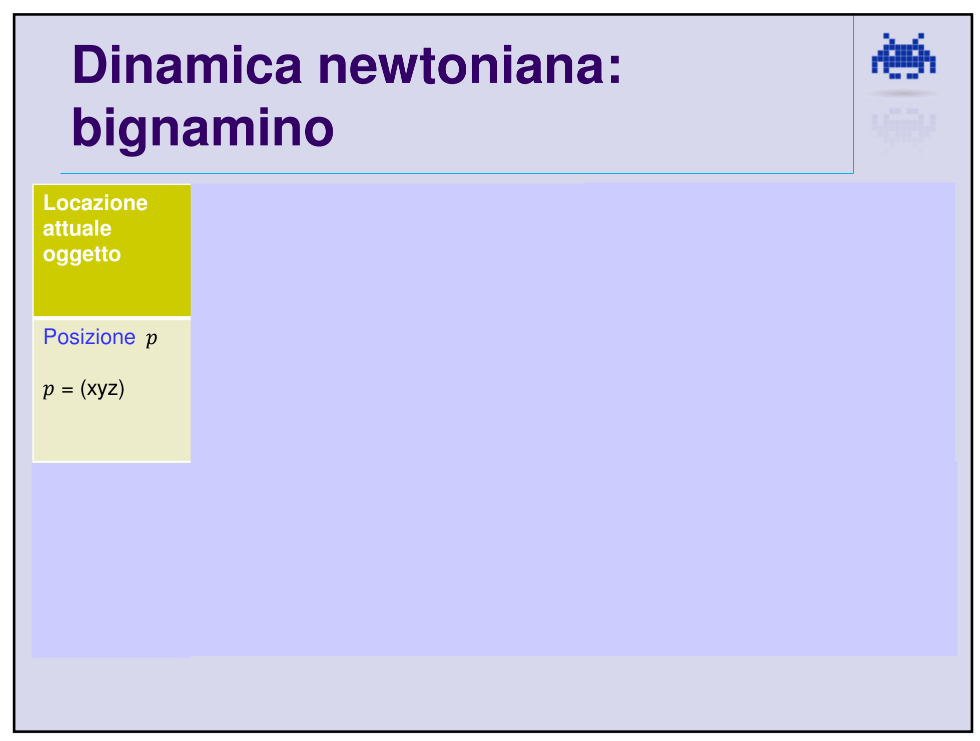

PhysX
ETH
Novodex
ageia
NVIDIA
PhysX by NVIDIA
unity
UNREAL ENGINE
Gamebryo
...

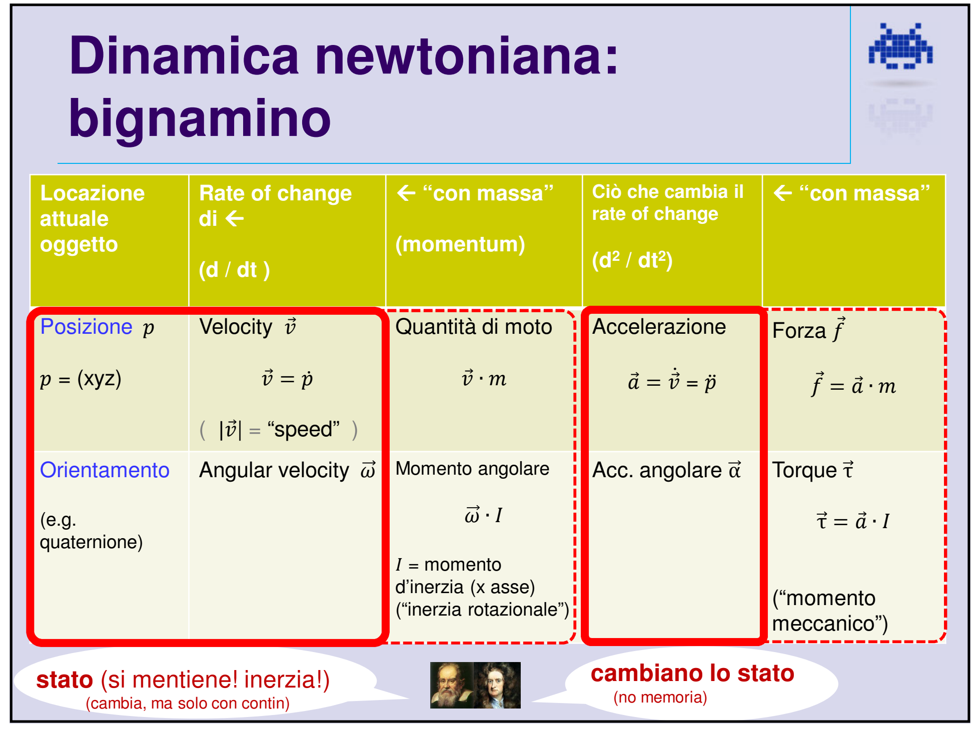
Fisica nei games: cosmesi o gameplay?
Solo un accessiorio grafico? (x realismo!)
es:
particle effects (senza feedback)
“secondary” animations e.g.
ragdolling
o parte del gameplay?
e.g. phyisic based puzzles
in 2D, approccio popolare (da sempre!)

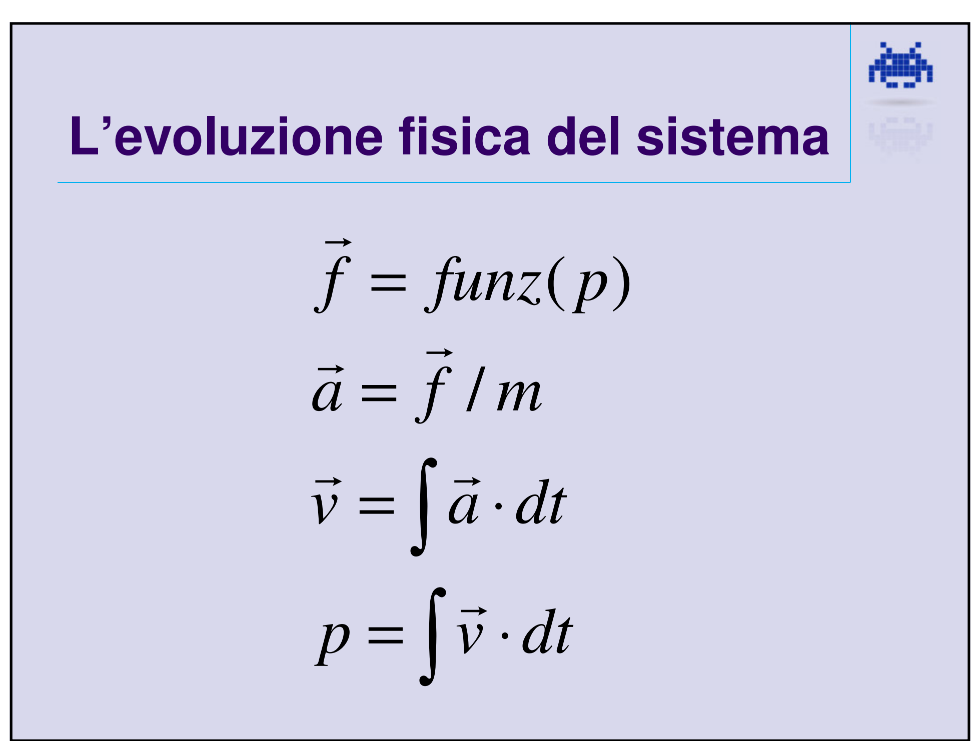
Fisica nei games: cosmesi o gameplay?

- Solo un accessiorio grafico?
  (x realismo!)
  - es:
    - particle effects (senza feedback)
    - “secondary” animations e.g.
    - ragdolling
- o parte del gameplay?
  - e.g. phyisic based puzzles
  - in 3D : tendenza in aumento

Reminder:
Posizione di un oggetto
Fisica 2D
Posizione:
(x,y)
Orientamento:
(α) – angolo (scalare)
Fisica 3D
Posizione:
(x,y,z)
Orientamento:
quaternione oppure
asse,angolo oppure
asse*angolo oppure
matrice 3x3 oppure
angoli Eulero oppure …
non molto adatti


Dinamica newtoniana:
bignamino
Locazione attuale oggetto
Posizione p
p = (xyz)


Dinamica newtoniana: bignamino
Locazione attuale oggetto
Rate of change di ← (d / dt)
← "con massa" (momentum)
Ciò che cambia il rate of change (d² / dt²)
← "con massa"
Posizione p
p = (xyz)
Velocity $\vec{v}$
$\vec{v} = \dot{p}$
($|\vec{v}|$ = "speed")
Quantità di moto
$\vec{v} \cdot m$
Accelerazione
$\vec{a} = \dot{\vec{v}} = \ddot{p}$
Forza $\vec{f}$
$\vec{f} = \vec{a} \cdot m$
Orientamento
(e.g. quaternione)
Angular velocity $\vec{\omega}$
Momento angolare
$\vec{\omega} \cdot I$
I = momento d'inerzia (x asse) ("inerzia rotazionale")
Acc. angolare $\vec{\alpha}$
Torque $\vec{\tau}$
$\vec{\tau} = \vec{a} \cdot I$
("momento meccanico")
stato (si mentiene! inerzia!)
(cambia, ma solo con contin)
cambiano lo stato
(no memoria)


L'evoluzione fisica del sistema
$\vec{f} = funz(p)$
$\vec{a} = \vec{f} / m$
$\vec{v} = \int \vec{a} \cdot dt$
$p = \int \vec{v} \cdot dt$

Computare l'evoluzione fisica

- Modelli analitici:

  stato = funz(t)

- Metodi numerici

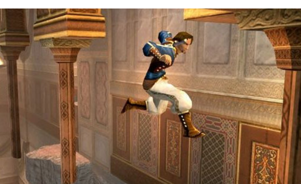
  1. $\text{stato}_{(t=0)}$ ← init
  2. $\text{stato}_{(t+1)}$ ← evolve(stato_t)
  3. goto 2

ESEMPI ALLA LAVAGNA

Un (ovvio) inciso

- t = tempo virtuale != tempo reale
  - es:
    - gioco in pausa → t costante.
    - Fast forward, replay, rallenty, time all'indietro → cambia ritmo di scorrimento di t

e, occasionalmente,
nel **gameplay** si gioca su questa differenza in modo spettacolare!



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

Braid (Jonathan Blow, 2008)

Computare l’evoluzione fisica

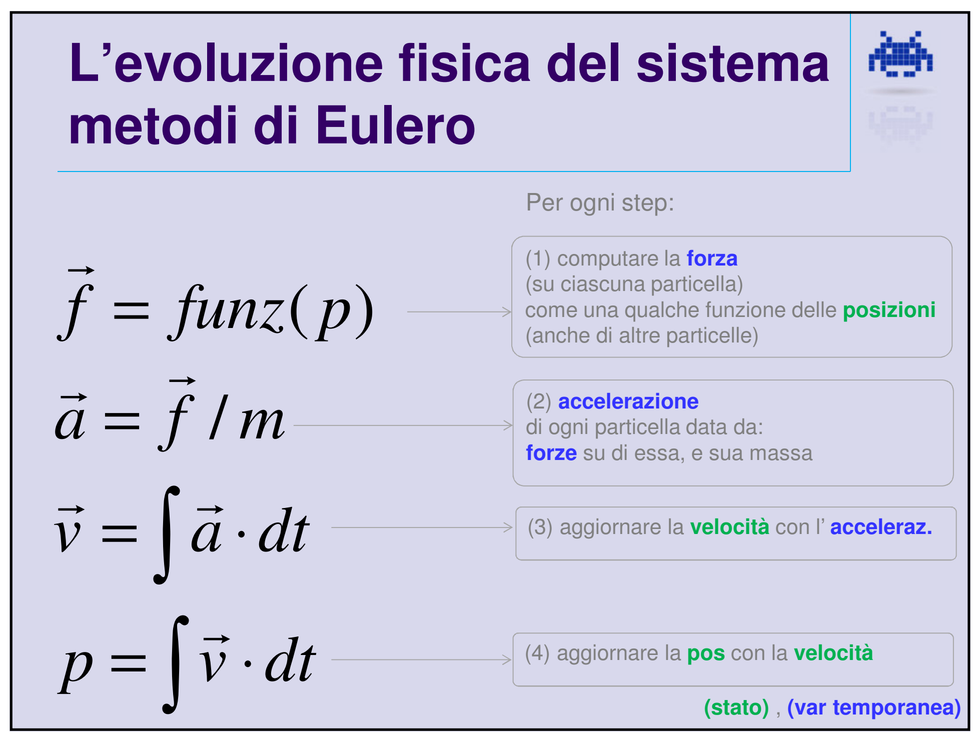
- Modelli analitici:
  - efficientissimi!
    - soluz forma chiusa
  - accurati
  - solo sistemi semplici
  - formule ad-hoc caso per caso
  - NO (ma, per es, utili per fare predirre le cose all’AI)
- Metodi numerici:
  - dispendiosi (iterativi)
    - ma, *interattivi*
  - errori di integrazione
  - flessibili
  - generici
  - SI

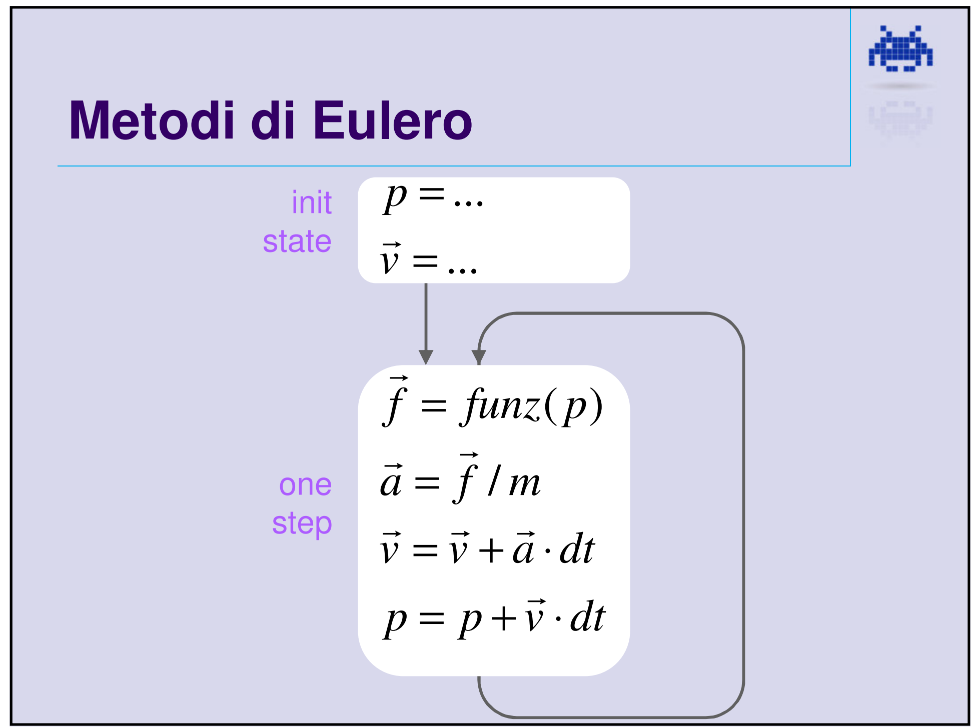
Alcuni modelli numerici

- Metodi di Eulero
  - (semplici e diretto. Ma…)
- Metodi “leapfrog”
  - (*it:* a “cavallina”)
- Metodi di Verlet
  - (position based dynamics)

Caratteristiche di un metodo numerico

- Quanto è efficiente / oneroso
  - serve che sia almeno <u>soft</u> real-time
    - (se ogni tanto computaz rimadata al prox frame, ok)
- Quanto è accurato
  - serve che sia almeno <u>plausibile</u>
    - (se rimane plausibile, discrepanze dalla realtà, ok)
- Quanto è robusto
  - serve che risultati del tutto sballati siano molto rari
    - (e, <u>mai</u> crash)
- Quanto è generico
  - quali fenomeni / vincoli / tipo di oggetti è in gradi di riprodurre?
  - necessità dipendenti dal contesto (es, gameplay)

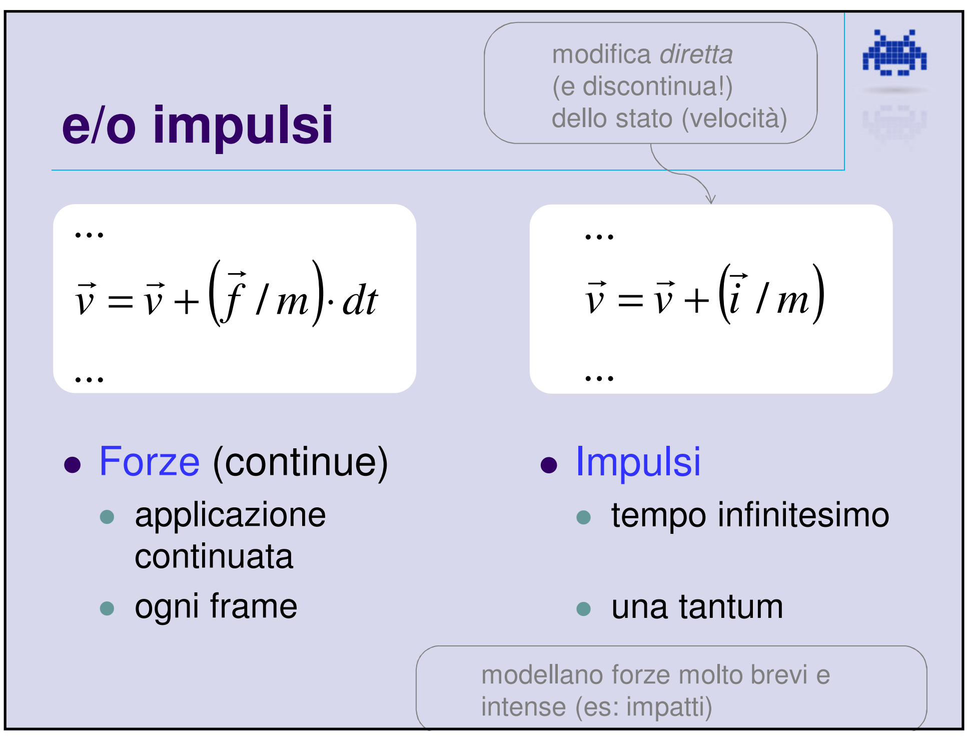


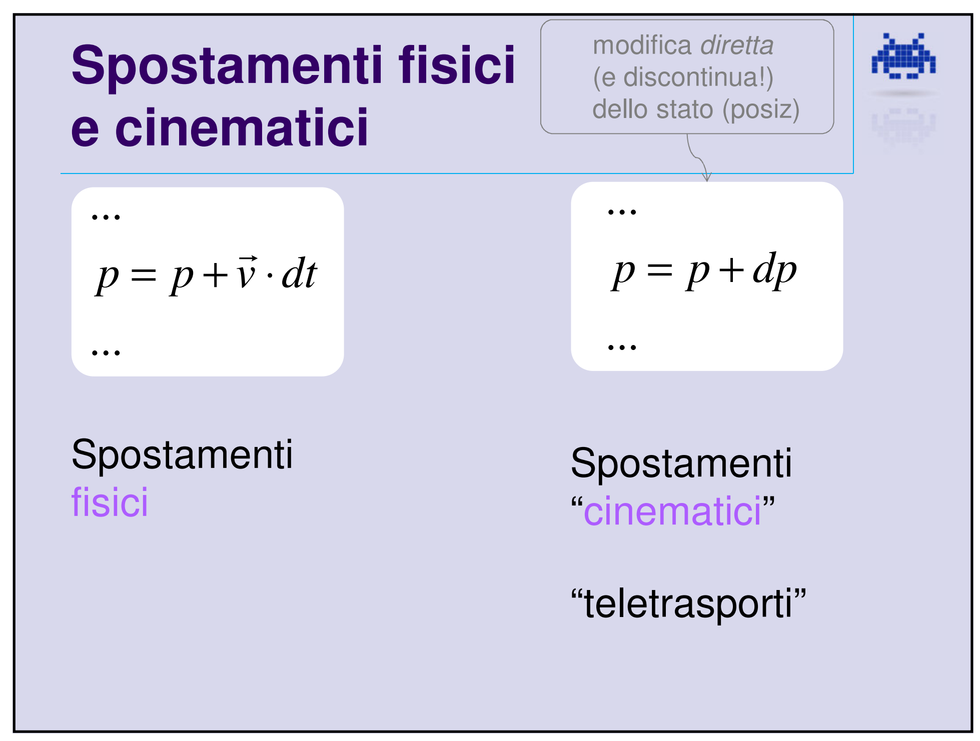


Forze

- In generale, funzione delle pos attuali
  - Gravità
  - Attriti
    - (ma, questi simulate bene anche da damping)
  - Opposizione dei materiali
    - (ma, queste simulate bene anche da vincoli… vedi poi)
  - Vento, elettrica, magentica, tensione superficiale, molle, onde d'urto (esplosioni), etc etc…
  - Forze "posticcie"
    - (aggiunte x controllare l'evoluzione)

$$\ldots$$
$$\vec{f} = funz(p)$$
$$\ldots$$

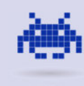

e/o impulsi
modifica diretta
(e discontinua!)
dello stato (velocità)
...
$\vec{v} = \vec{v} + (\vec{f} / m) \cdot dt$
...
...
$\vec{v} = \vec{v} + (\vec{i} / m)$
...
Forze (continue)
applicazione continuata
ogni frame
Impulsi
tempo infinitesimo
una tantum
modellano forze molto brevi e intense (es: impatti)


Spostamenti fisici e cinematici
modifica diretta
(e discontinua!)
dello stato (posiz)
...
$p = p + \vec{v} \cdot dt$
...
...
$p = p + dp$
...
Spostamenti fisici
Spostamenti "cinematici"
"teletrasporti"

Euler *pseudo code*

```
Vec3 position = …
Vec3 velocity = …

void initState(){
   position = …
   velocity = …
}

void physicStep( float dt )
{
   position += velocity * dt;
   Vec3 acceleration = force( positions ) / mass;
   velocity += acceleration * dt;

}

void main(){
  initState();
  while (1) do physicStep( 1.0 / FPS );
}
```

Metodi numerici: passo di integrazione



dt : delta di tempo virtuale dall'ultimo step

la "risuluzione temporale" della simulazione!

- se **piccolo**: poca efficienza
  - più steps per simulare stessa quantità tempo virtuale
- se **grande**: poco accuratezza
  - soprattutto in presenza di forze e/o vel grandi

- valori tipici: 1 / 60 sec -- 1 / 30 sec
  - nota: non necess. lo stesso refresh rate del rendering

ordine della simulazione:
quanto cresce l'errore al crescere di dt

Metodo Eulero: limitazioni

- cattivo rapporto efficienza / accuratezza
  - errore accumulato nel tempo = lineare con dt
- persino la plausibilità può essere compromessa
  - non mantenimento energia !
  - es: oscillazioni divergenti !
  - patch: damping della velocità (in ogni step)

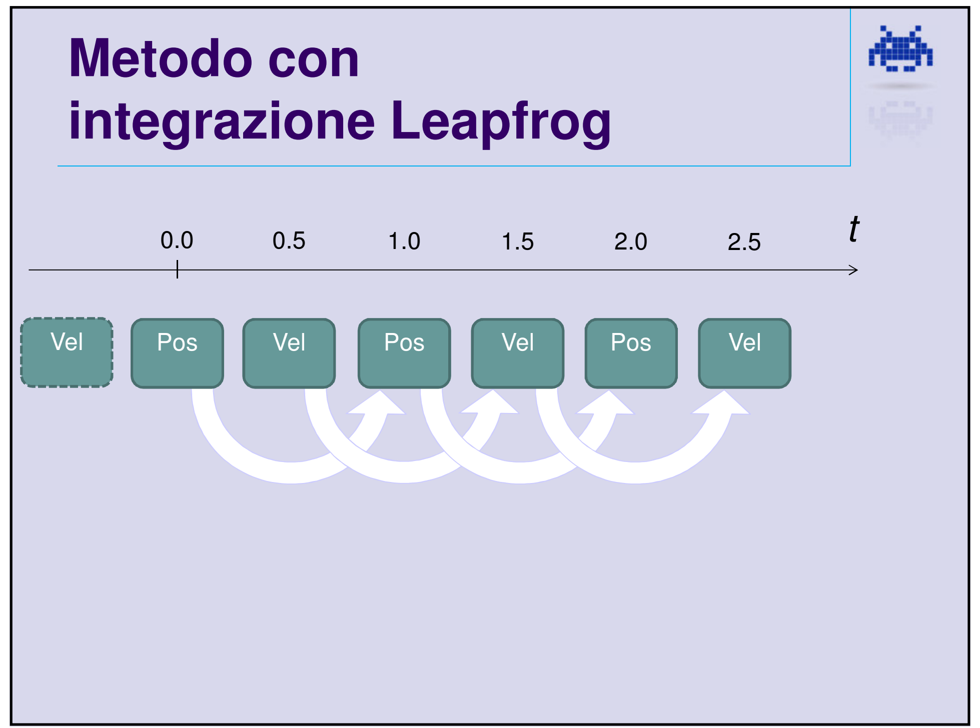
```
float damp = 1.0 – CONST * dt;
velocity *= damp;
```

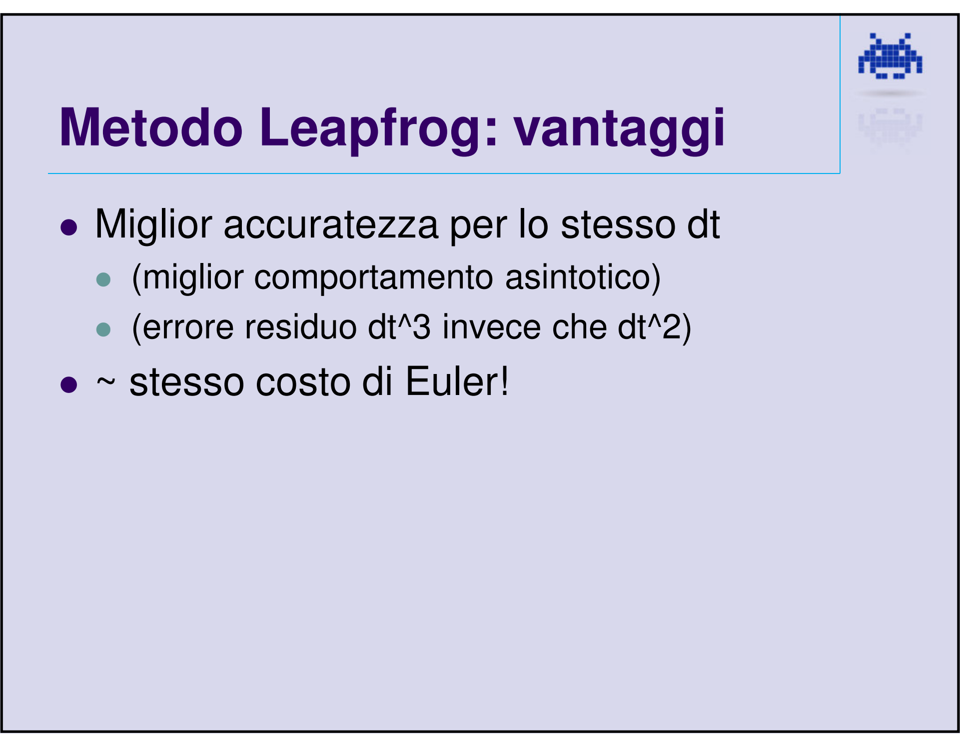
    - perdita continua dell'energia cinetica
    - scusa ufficiale: "sono gli attriti con… tutto (aria compresa)"

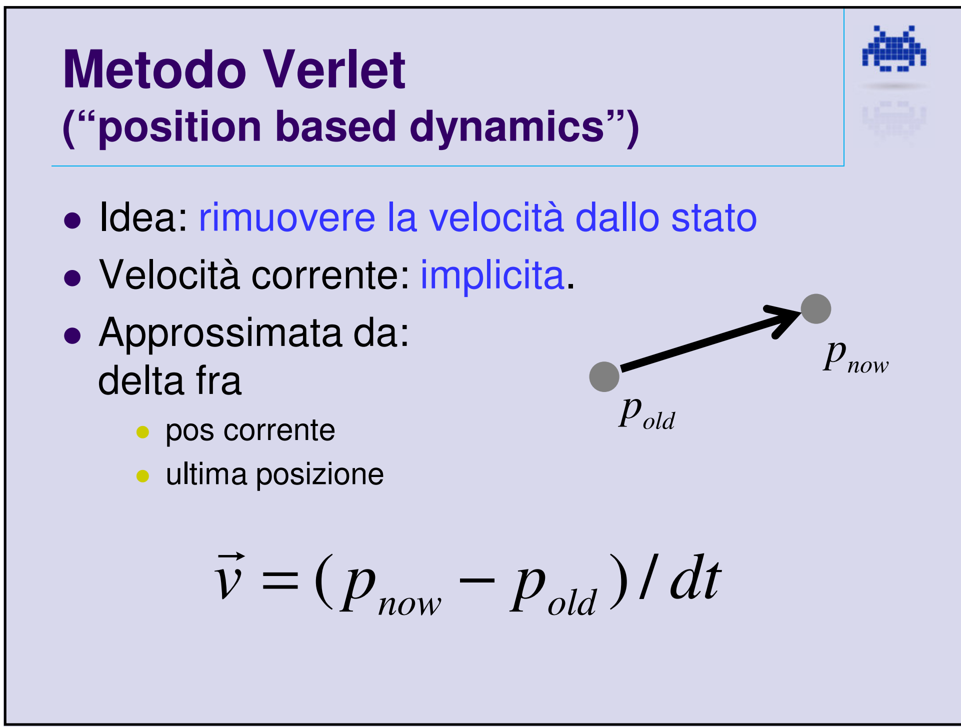
Trick: velocity «Damping»
(o «Drag», resistenza dell'aria)

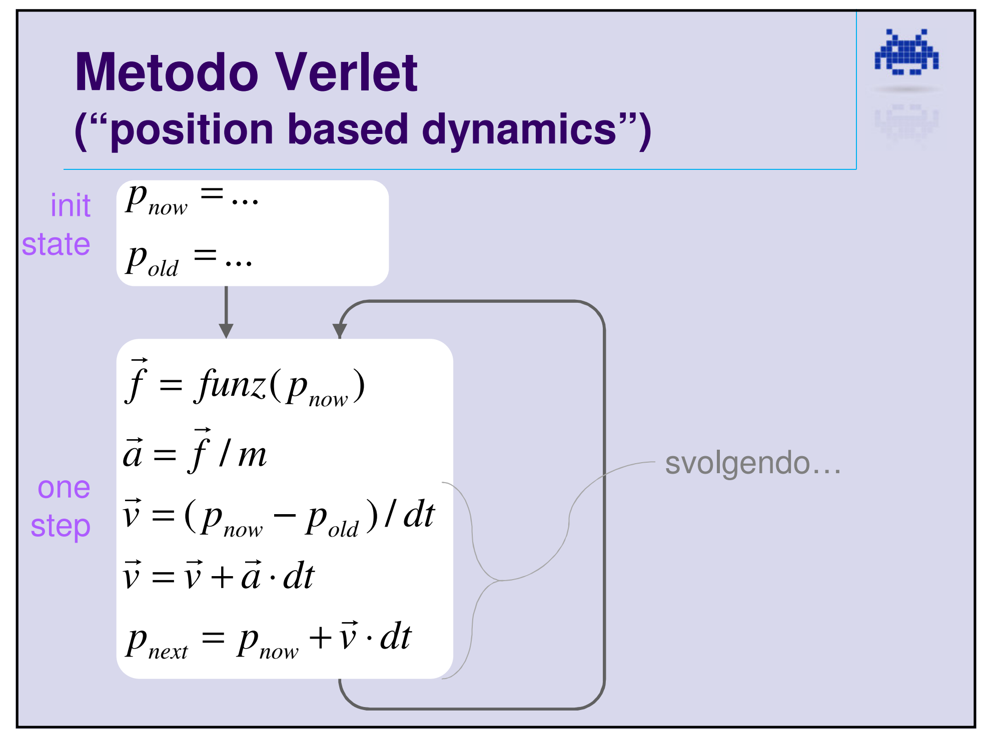
```
Vec3 position = …
Vec3 velocity = …

void initState(){
   position = …
   velocity = …
}

void physicStep( float dt )
{
   position += velocity * dt;
   Vec3 acceleration = force( positions ) / mass;
   velocity += acceleration * dt;
   velocity *= (1.0 – DAMP * dt);
}

void main(){
  initState();
  while (1) do physicStep( 1.0 / FPS );
}
```

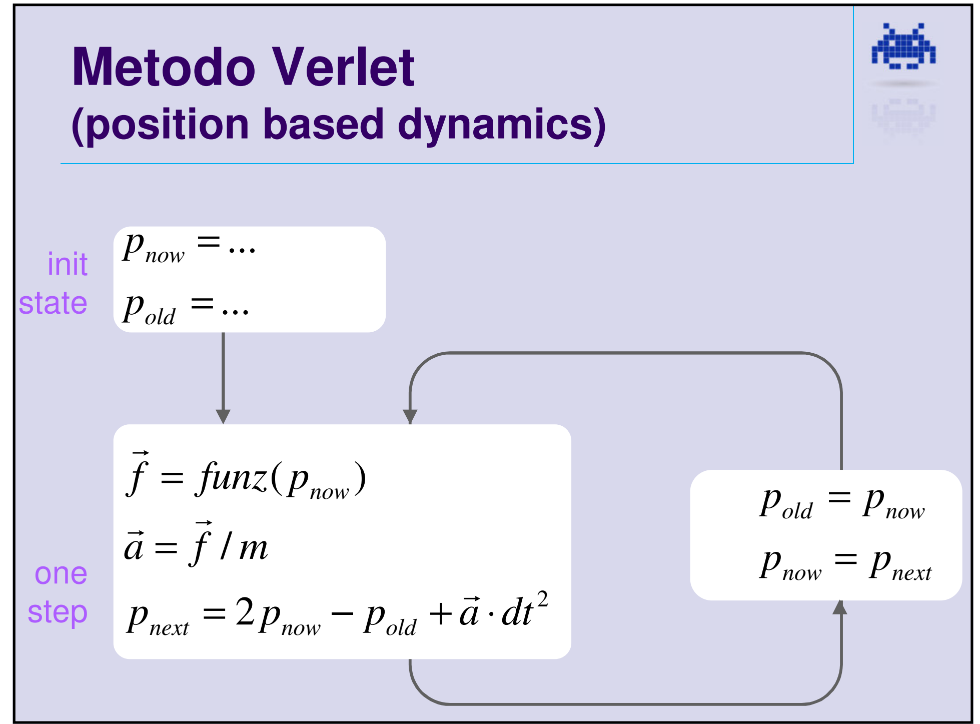
decurtamento aritificiale energia cinetica:
- robustezza
  (evita aumeto energia x errori numerici)
  (es. evita oscillazioni divergenti)
- dispersioni non modellate
  (attriti vari)
- dump alto = come muoversi nella melassa
- aka DRAG


Metodo con
integrazione Leapfrog
0.0
0.5
1.0
1.5
2.0
2.5
t
Vel
Pos
Vel
Pos
Vel
Pos
Vel


Metodo Leapfrog: vantaggi
Miglior accuratezza per lo stesso dt
(miglior comportamento asintotico)
(errore residuo dt^3 invece che dt^2)
~ stesso costo di Euler!


Metodo Verlet
("position based dynamics")
Idea: rimuovere la velocità dallo stato
Velocità corrente: implicita.
Approssimata da: delta fra
pos corrente
ultima posizione
p_{now}
p_{old}
$\vec{v} = (p_{now} - p_{old}) / dt$


Metodo Verlet
("position based dynamics")
init state
$p_{now} = \ldots$
$p_{old} = \ldots$
one step
$\vec{f} = funz(p_{now})$
$\vec{a} = \vec{f} / m$
$\vec{v} = (p_{now} - p_{old}) / dt$
$\vec{v} = \vec{v} + \vec{a} \cdot dt$
$p_{next} = p_{now} + \vec{v} \cdot dt$
svolgendo…



Verlet: caratteristiche

- Implicit velocity!
  - ma, lavorare direttamente sulla velocità?
    - es: damping
    - es: impulsi
  - ancora possible: modificare p_{old}
- Buon rapporto efficienza / accuratezza
  - errore accumulato: ordine di dt^2
- Bonus extra: invertibilità del sistema
  - (volendo, posso percorrere l'evoluzione all'indietro nel t e tornare nello stato iniziale corretto)
  - (stando attenti con dettagli implementativi)

Verlet: caratteristiche

- Vantaggio principale: genericità
  - possibile imporre molti vincoli diversi
  - basta risolvere per posizione:
    - cambiamento della velocità "automatico" (non corretto ma plausibile)
  - *torneremo su questo in seguito…*

Motore fisico: COLLISIONI

L'altra meta' del motore fisico…

- **Collision detection**
  - scoprire quando avvengono
- **Collision response**
  - includerne gli effetti nella dinamica
    - garantire non compenetrazione
    - rimbalzi
    - attriti

Collision detection

- Problemi di efficienza:
  a) evitare esplosione quadratica dei test
    - N oggetti → N^2 tests ?
  b) test fra due oggetti:
    - Come renderlo efficace?

- Soluzioni:
  strutture dati apposite…

Rendere efficiente la collision detection

a) evitare esplosione quadratica dei test
  - N oggetti → N^2 tests ?

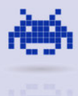
- *Soluzione*:
  strutture di indicizzazione spaziale
  - statiche!
  - necessità ricomputazione / aggiornamento, se si spostano
    - preprocessing sulla parte statica della scena
    - (che va indentificata)

Rendere efficiente la collision detection

b) test fra due oggetti:
- Come renderlo efficace?

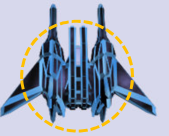
- *Soluzione*:
  geometric proxy…

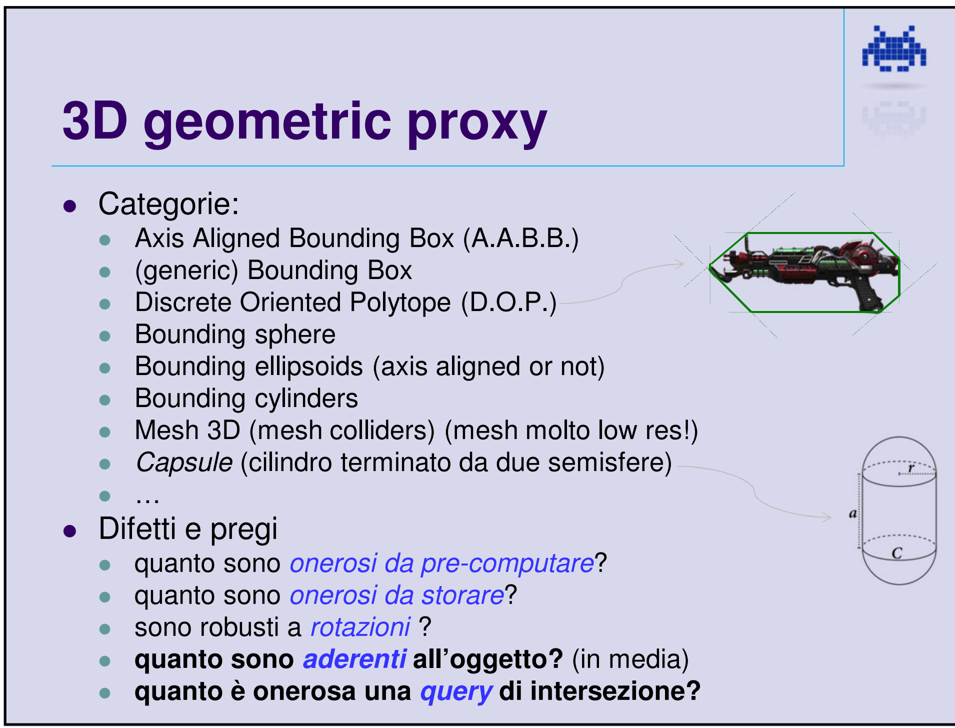
Geometric proxy

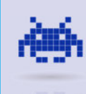


- Rappresentazione molto semplificata degli oggetti usata (anche) dal motore fisico
  - (*molto* più semplice e approssimato del modello usato nel rendering!)
- Bounding volume
  - limite sup all'estensione dell'oggetto (oggetto "tutto dentro" il bounding volume)
  - ➔ test *conservativi*

- Collision object (o "hit-box")
  - approx dell'estensione dell'oggetto
  - ➔ test *approssimati*



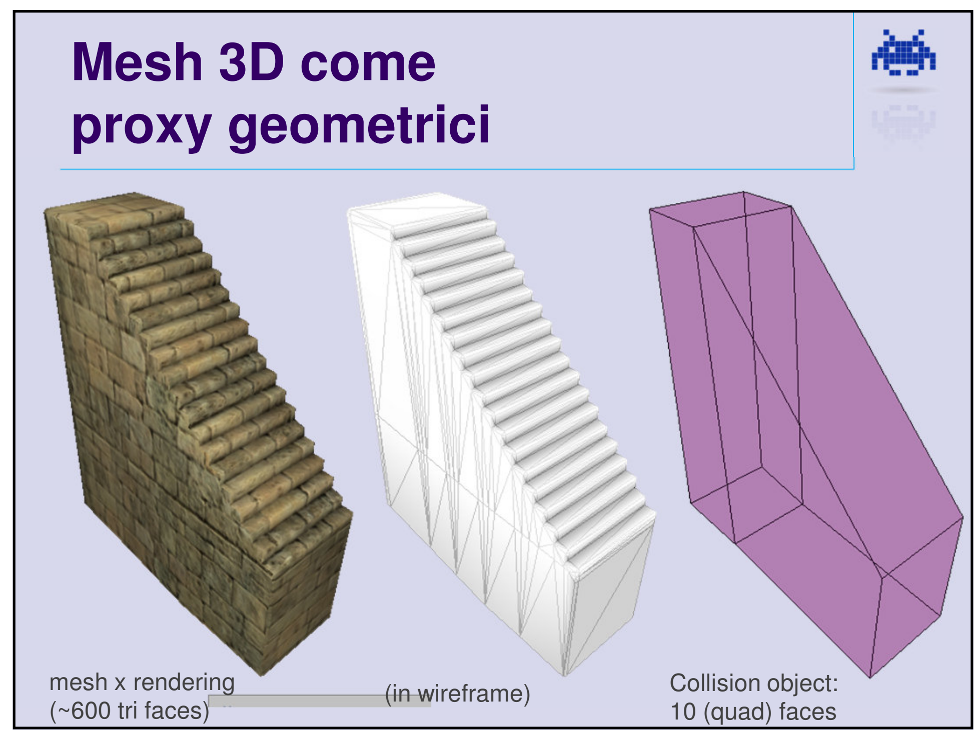


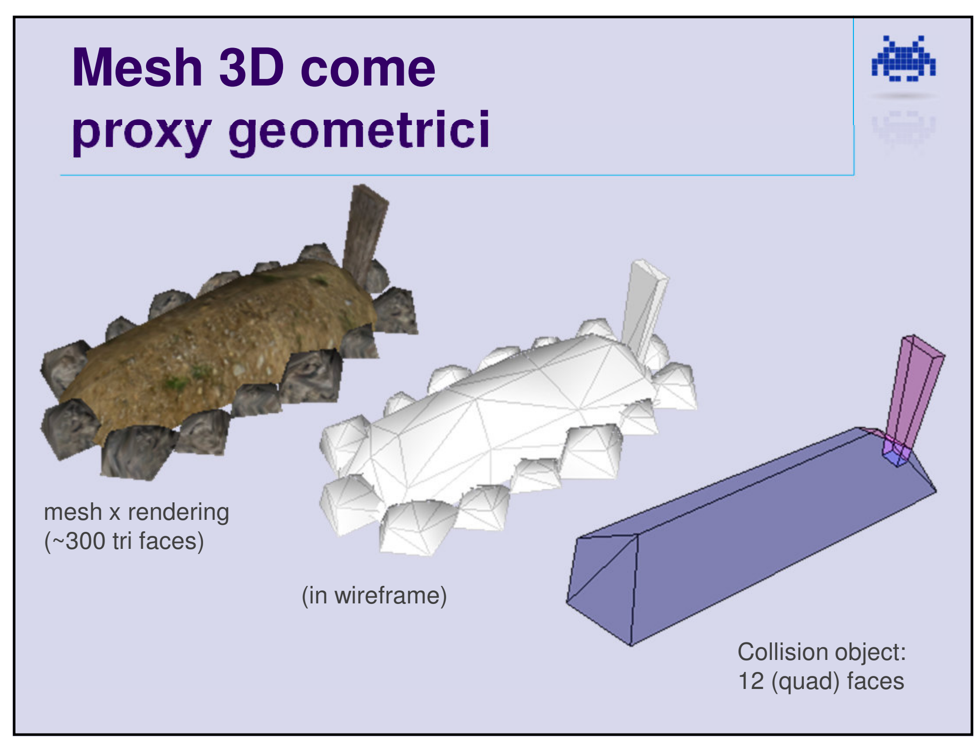


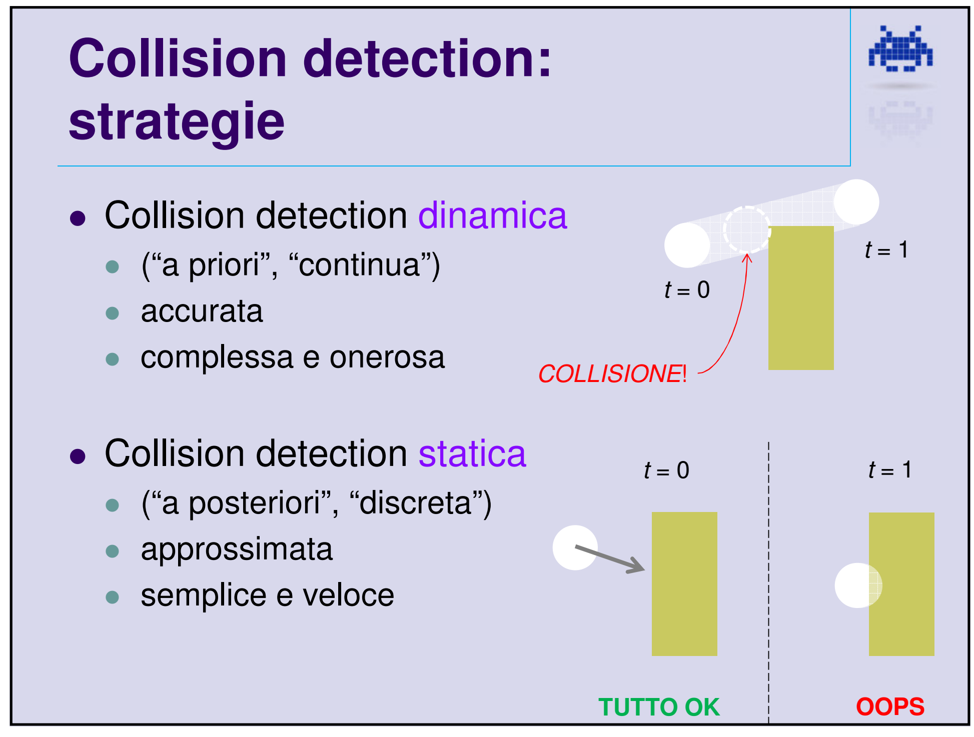
Mesh 3D come proxy geometrici

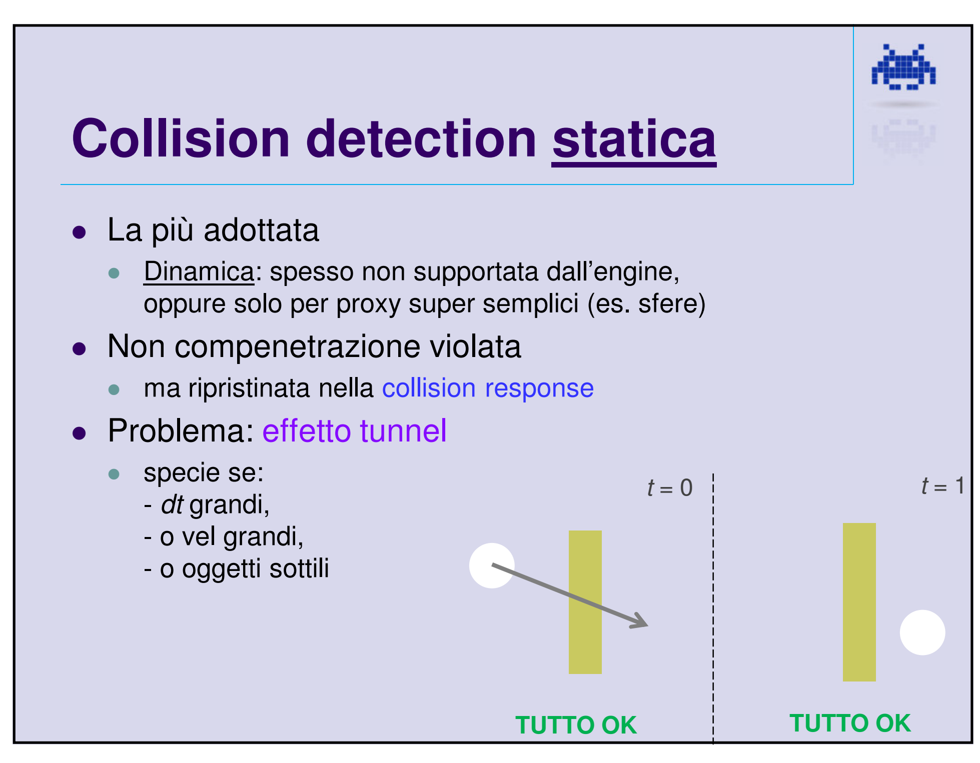
- Sono assets !
  - a volte, da sempl automatica
  - altre, modellati da artisti

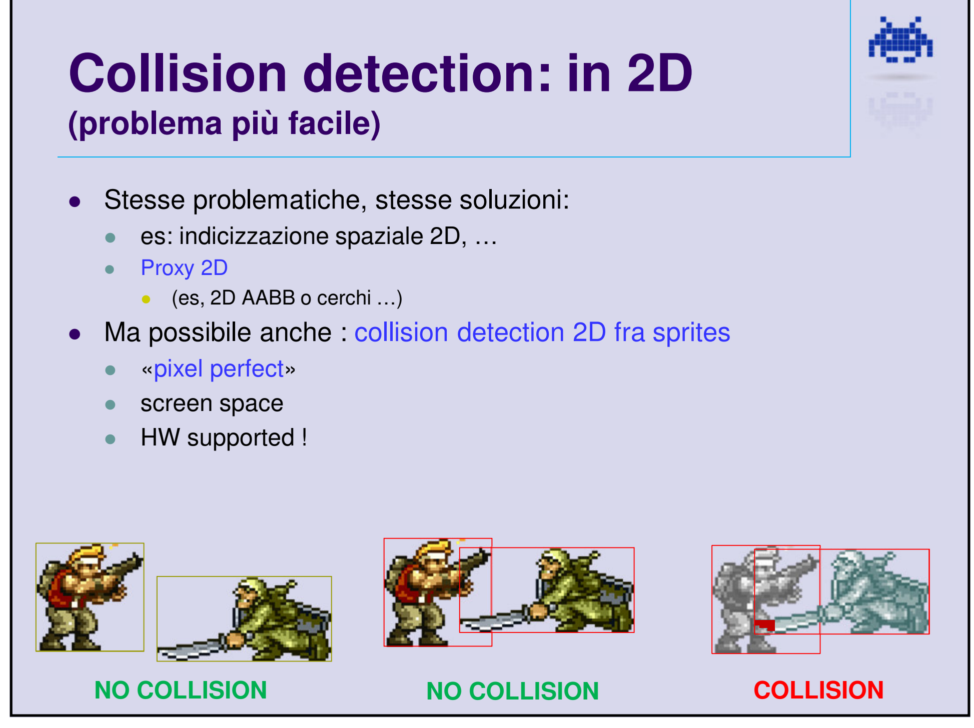
  quelli usati per rendering
- Differenze con modelli 3D «normali» :
  - molto lower res (~ $O(10^2)$)
  - no attributi (no uv-mapping, etc)
  - poligoni non tri: ok
  - sempre chiuse, e two manifold (dentro != fuori)
  - a volte: solo convesse
- Preprocessing:
  - collision trees


Mesh 3D come
proxy geometrici
mesh x rendering
(~600 tri faces)
(in wireframe)
Collision object:
10 (quad) faces


Mesh 3D come
proxy geometrici
mesh x rendering
(~300 tri faces)
(in wireframe)
Collision object:
12 (quad) faces


Collision detection: strategie
Collision detection dinamica
("a priori", "continua")
accurata
complessa e onerosa
Collision detection statica
("a posteriori", "discreta")
approssimata
semplice e veloce
t = 0
t = 1
COLLISIONE!
t = 0
t = 1
TUTTO OK
OOPS


Collision detection statica
La più adottata
Dinamica: spesso non supportata dall'engine, oppure solo per proxy super semplici (es. sfere)
Non compenetrazione violata
ma ripristinata nella collision response
Problema: effetto tunnel
specie se:
- dt grandi,
- o vel grandi,
- o oggetti sottili
t = 0
t = 1
TUTTO OK
TUTTO OK



Motore fisico: COLLISIONI

L’altra meta’ del motore fisico…

- Collision detection
  - scoprire quando avvengono
- Collision response
  - includerne gli effetti nella dinamica

Collision response

- Includerne nella dinamica gli effetti della collisione:
  (1) garantire **non-compenetrazione**
  - *quando:* sempre
  - *come:* “teletrasporti”, oppure vincoli posiz (vedi poi)

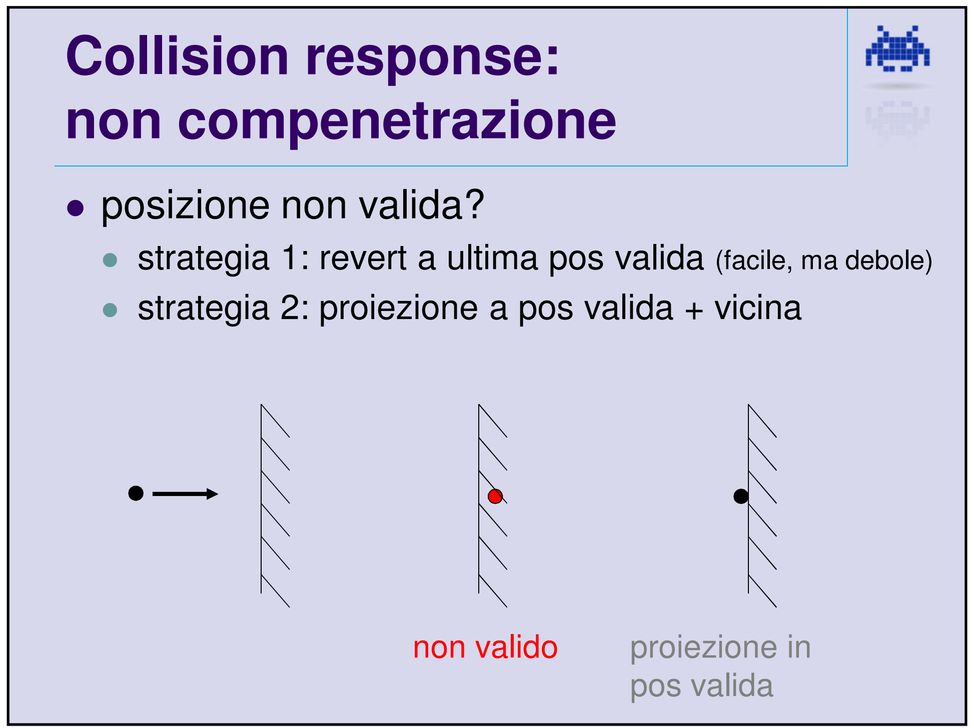
  (2) **rimbalzi**
  - *quando:* al momento dell’impatto
  - *come:* impulsi (oppure anche nulla, in Verlet, vedi poi)

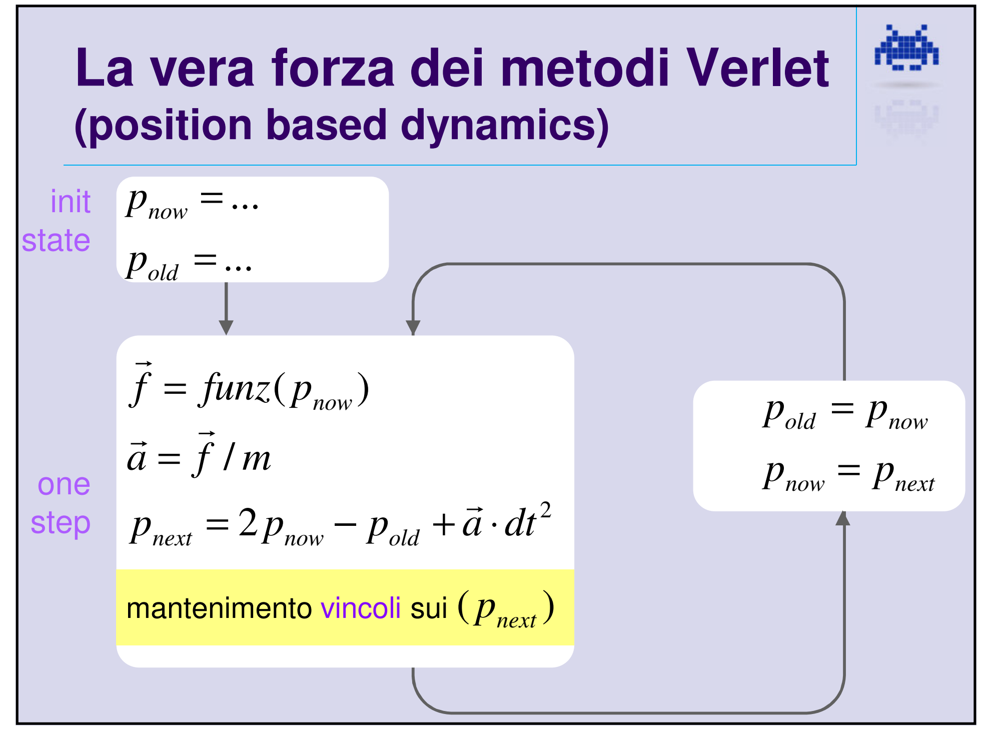
  (3) **attriti**
  - *quando:* nei contatti prolungati
  - *come:* forze che si oppongono al moto
    oppure (+ semplice): incrementare DAMP

Esito di una collision detection

- Collisione SI / NO ?
  - c’è intersezione o no
- Se SI:
  - dati necessari alla collision response:
    - posizione(-i) della collisione?
    - normale?
    - posizione “valida” più vicina?

calcolati dai proxy!


Collision response: non compenetrazione
posizione non valida?
strategia 1: revert a ultima pos valida (facile, ma debole)
strategia 2: proiezione a pos valida + vicina
non valido
proiezione in pos valida

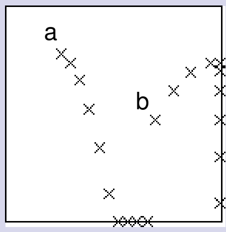

La vera forza dei metodi Verlet
(position based dynamics)
init state
$p_{now} = \dots$
$p_{old} = \dots$
one step
$\vec{f} = funz(p_{now})$
$\vec{a} = \vec{f} / m$
$p_{next} = 2p_{now} - p_{old} + \vec{a} \cdot dt^2$
mantenimento vincoli sui (p_{next})
$p_{old} = p_{now}$
$p_{now} = p_{next}$

Vincoli su posizione
(positional constraints)

- generici e espressivi
  - molti fenomeni
  - non solo (ma anche) “no compenetrazioni”
- semplici da definire
- facili da imporre (vedi poi)
- e, con Verlet:
  - aggiornamento velocità: *conseguenza automatica*!
  - senza passare dalle forze / impulsi
    - (quelle che nella realtà impongono il vincolo)
  - → approssimazione cruda, ma risultati plausibili!

Esempio di vincolo posizionale

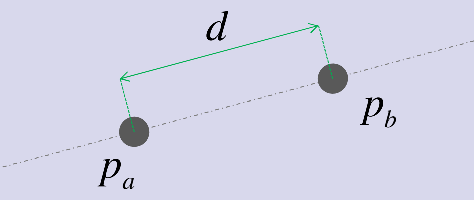
“Le particelle devono stare dentro [0 – 1000] x [0 – 1000]”




```
// for each particle
for(int i=0; i<NUM_PARTICLES; i++)
{
  p[i].x = clamp( p[i].x, 0, 1000 );
  p[i].y = clamp( p[i].y, 0, 1000 );
}
```

Vincoli di equidistanza

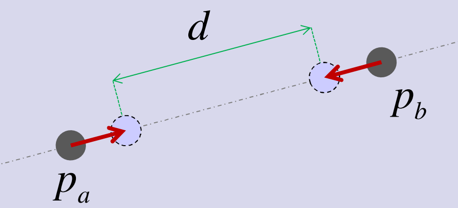
- "Particelle *a* e *b* devono stare a distanza *d* "



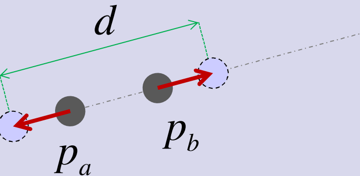
$$|\, p_a - p_b \,| = d$$

Imporre i vincoli di equidistanza

se $|\, p_a - p_b \,| > d$



se $|\, p_a - p_b \,| < d$

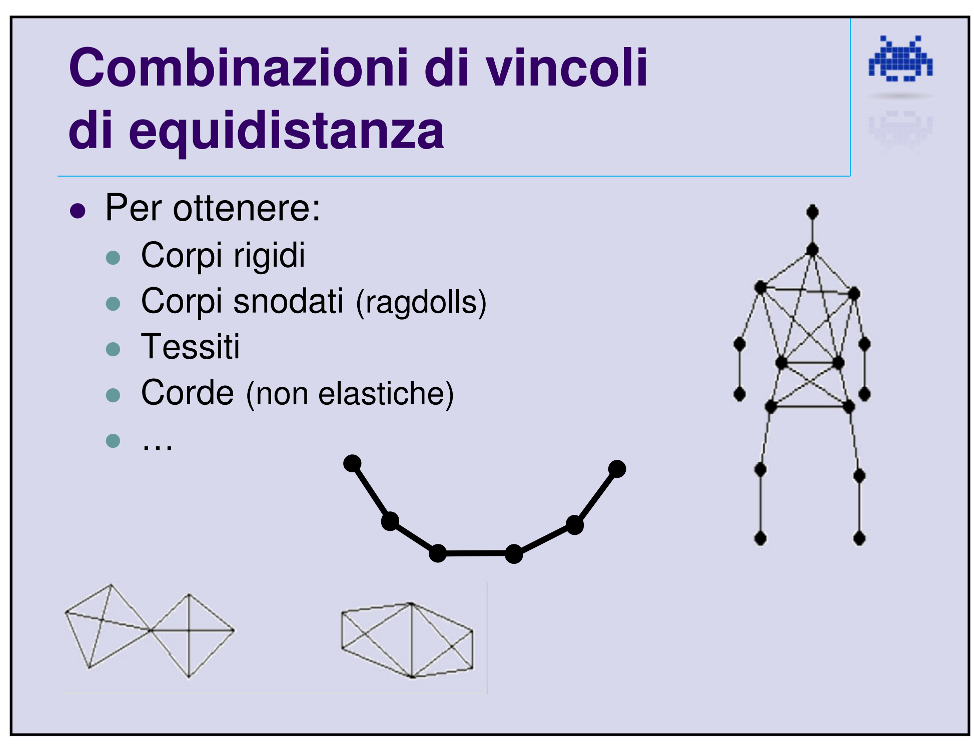


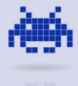
Vincoli di equidistanza: pseudocodice

```
Point3 delta = p[b] - p[a];

float curr_dist = delta.length;

Point3 diff = (curr_dist - d ) / curr_dist;

p[a] += delta * (0.5 * diff);
p[b] -= delta * (0.5 * diff);
```

Combinazioni di vincoli di equidistanza

- Per ottenere:
  - Corpi rigidi
  - Corpi snodati (ragdolls)
  - Tessiti
  - Corde (non elastiche)
  - …



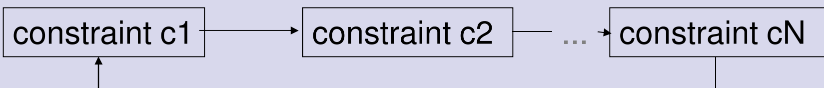


Altri vincoli posizionali

- Mantenimento area / volume
- Posizioni fisse
  - particelle «inchiodate sul posto»
  - (banale, ma utile!)
- Vincoli su angoli
  - (per es, sui giunti)
  - (come: «i gomiti non si piegano all'indietro»)
- Coplanarità / colinearità
- …

Soddisfare constraints tramine Rilassamento

- Molti constraint
- Risolverne uno → romperne un altro
- Soluzione simultanea: computazionalmente difficile
- Soluzione pratica:

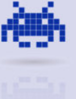


Ripetere fino a soddisfacimento (= errore max sotto soglia) …ma al più *N* volte!

Constraint Satisfaction by Relaxation

- Relaxation method
  - (simile a metodo di Jacobi)
  - alternativa: Gauss-Seidel
- Convergenza
  - se vincoli non contraddittori
  - itreazioni richieste (di solito): 1 ~ 10 (efficiente!).
  - e se non converge subito, pazienza:
    frame successivi provvederanno (abb. robusto)

Per approfondire



- Müller-Fischer et al.
  Real-time physics
  (Siggraph course notes, 2008)
  http://www.matthiasmueller.info/realtimephysics/