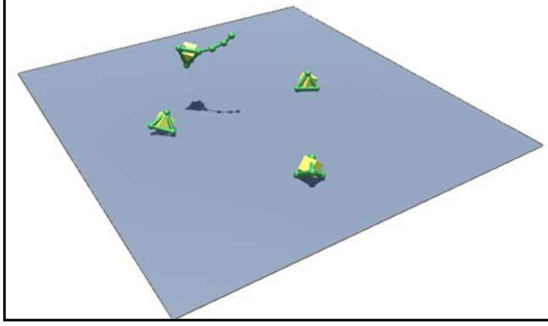
3D video games

Physics sandbox

Marco Tarini



Objective of this sandbox

Implement a simple Verlet based, PBD physics system on Unity

- Basic idea:
  - don’t enable default Unity physics system
  - instead, crudely implement phsyics in scripts by hand
  - *note*: in a normal project, there’s no reason to do this!
- How not to enable physics in Unity:
  - Just don’t add, to any GameObject, any “RigidBody” component (implemets dynamics) or any “Collider” component (implements collision handling)
- we will still use the normal Unity scene-graph support
  - GameObjects, and their associated Transforms

Background: “behaviors” in Unity

- In Unity, a behavior is a script associated to Game objects
- It is a C# class, with predefined methods used by the resto of the engine:
  - Start() – called at scene construction
  - FixedUpdate() – called for each fixed step
  - Update() – called before rendering this object (that is, at each rendering step)
- The value *dt* is exposed as Time.FixedDeltaTime

For details on methods used in this sandbox, refer to the implementation on the website!

Particles and Particle behavior

- Our particle is a game object
  - rendered as a small sphere
- Its associated behavior includes the fields:
  - pos, prevPos (points): for Verlet dynamics (pos is the current position)
  - mass, drag (scalars): constants (exposed to the interface)
- and the methods:
  - Start(): initializes Verlet
  - FixedUpdate(): performs a Verlet integration step

Implementation detail: pos VS transform.position

- For each particle, the current positions is stored twice:
  - The position according to our custom physics engine: pos – a custom field in the "behavior" of the particle
  - rendering position: the position used by the rendering engine transform.position, i.e. the position Unity uses for everything
- We keep them separated, just for code clarity
- At the beginning (start method)
  - physic position ← rendering position
    (so that the objects starts where we placed them in the GUI)
- Before each rendering (update method)
  - rendering position ← physic position
    (so that the object is rendered where the physics moved it)

Fixed-update of particles

- Basic Verlet integration
- Includes velocity dumping
  - see dump computation
- Includes addition of forces
  which depend only on this one particle
  - Such as gravity
- Includes enforcement of positional constraints
  which depend only on this one particle
  - Such as ground collision ("please stay above ground")

Adding positional constraint: stay “fixed”

- A Particle can simply “be asked” to stay fixed
- Implementation notes:
  - Added a Boolean field isFixed
  - Added the Vector3 field fixedPos , the pos where this particle is fixed in the scene (copied on Start)
  - Trivially imposed the constraint in the FixedUpdate()
- Small hack:
  - fixedPos is also updated at every frame,
    as the current rendering position
  - (so that we can move this particle from the GUI)

Adding rods

- Rods are GameObjects representing rigid rods connecting two particles
- Rendering:
  - A rod is rendered as a small cylinder
    (a cylinder mesh associated to the Game Object)
  - Before each rendering (update method)
    a transformation is computed so that the cylinder is scaled (on Y only), rotated, and translated
    to make it graphically connect the two particles
  - (therefore, it doesn't matter where we place them in the scene: they will teleport to the right location at each frame)

Rod behavior

- Fields:
  - Connected particles A and B
    It’s public: set them in the Unity GUI !
  - Rest length (computed on Start as the initial distance between particles A and B)
- Methods:
  - FixedUpdate: enforce the positional constraints, acting on the position of the two particles
  - Note: this take in account correctly of their mass

Sand box: results.

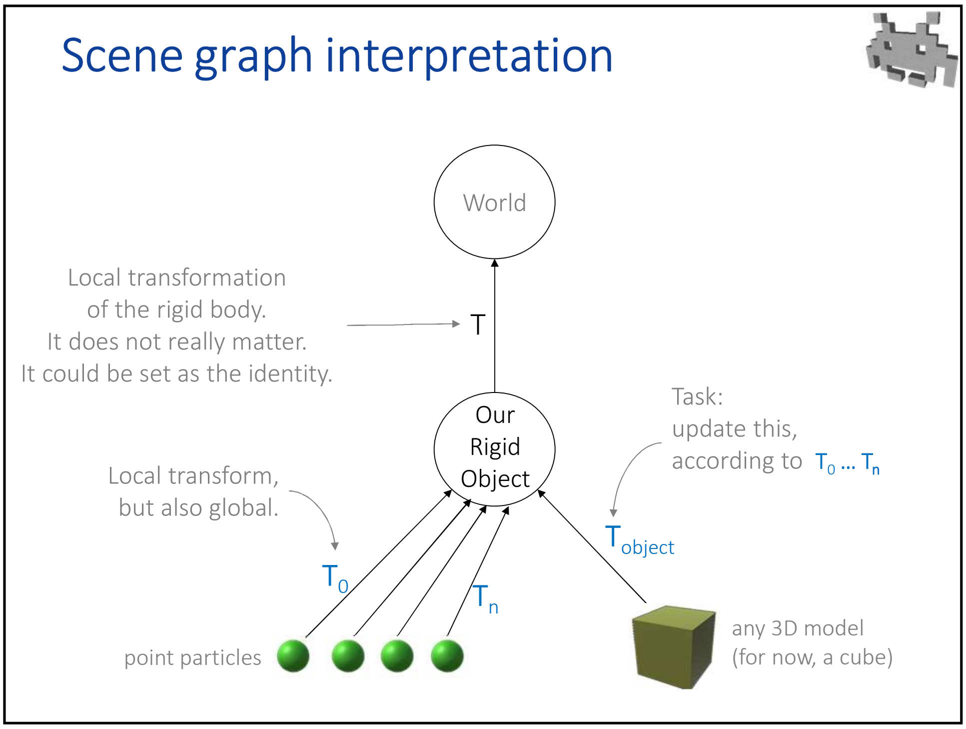
- Combining multiple particles and rods,
  we can construct meta-objects such as…
  - Ropes (a multiple-joint pendulum)
  - Rigid objects
  - Other articulated objects (todo!)
- For convenience, the sub-tree of the scene-graph making a meta-object can be stored as Prefabs (assets)
- Observe: rigid objects behave correctly, with plausible…
  - Effect of impact with the ground
  - Angular velocity
  - Angular momentum
  - Barycenter (try assigning a different mass to a rigid object hanged on a rope with an fixed end)

Sandbox: bonus results (overtime). A disclaimer.

- From this point on, the sandbox is *not* part of the official lectures of this course
- This is merely an occasion to have fun together using the conceptual tools seen at lectures
- No new topic that is relevant for the course is introduced from this point on
  - Any new topic encountered are *not* asked at exams

Rigid objects rendered as 3D models

- So far, a rigid object is just a subtree of the scene composed of many particles and rods
  - Note: they are not Unity “rigid objects”, but they act similarly
- Next step:
  we want to associate any custom 3D model to this object,
  - We want to render our rigid bodies as any single model,
    not as a collection of “balls and sticks”
  - For now, we will use just cubes as the models
- Basic idea: the “rigid body” meta-object is a two-level subtree
  - Children are:
    rods & particles,
    and the associated 3D model



Scene graph interpretation

World

T

Local transformation
of the rigid body.
It does not really matter.
It could be set as the identity.

Our
Rigid
Object

Task:
update this,
according to $T_0 \dots T_n$

T_{object}

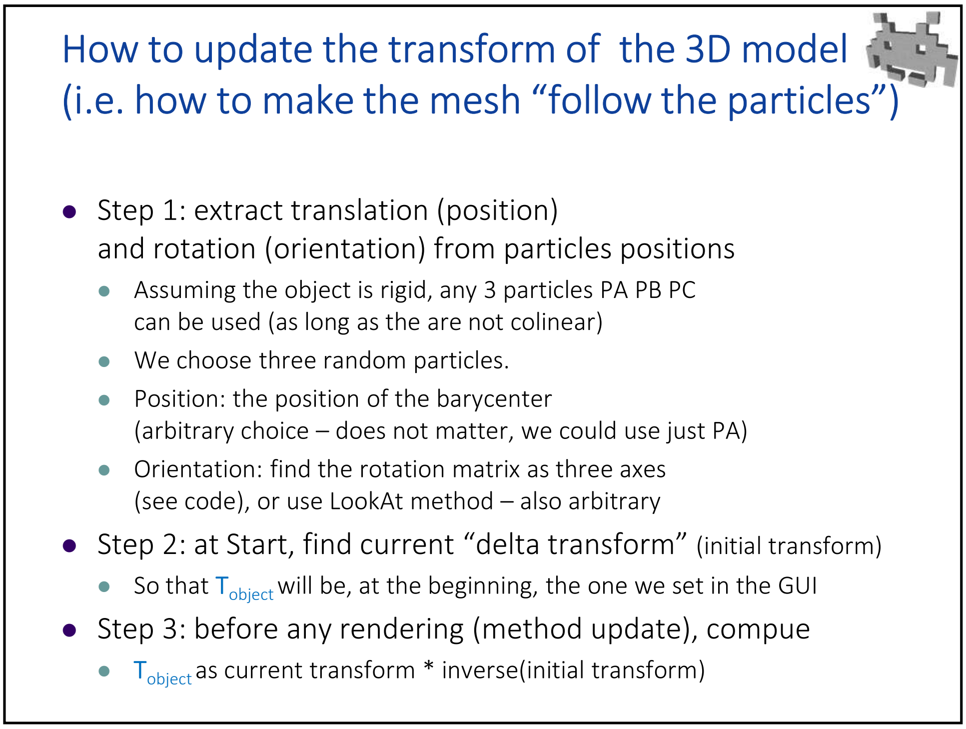
Local transform,
but also global.

T_0

T_n

point particles

any 3D model
(for now, a cube)



How to update the transform of the 3D model (i.e. how to make the mesh "follow the particles")

- Step 1: extract translation (position)
  and rotation (orientation) from particles positions
  - Assuming the object is rigid, any 3 particles PA PB PC can be used (as long as the are not colinear)
  - We choose three random particles.
  - Position: the position of the barycenter (arbitrary choice – does not matter, we could use just PA)
  - Orientation: find the rotation matrix as three axes (see code), or use LookAt method – also arbitrary
- Step 2: at Start, find current "delta transform" (initial transform)
  - So that T_{object} will be, at the beginning, the one we set in the GUI
- Step 3: before any rendering (method update), compue
  - T_{object} as current transform * inverse(initial transform)

How to update the transform of the 3D model (i.e. how to make the mesh “follow the particles”)

- Notes:
  - All computations happen in local space of the “my rigid body” node
  - Therefore we need to update the local Transform of the 3D model GameObject (i.e. its localRotation, localPosition…)
- About the bug which we had to fix:
  - We erroneously update its global rotation instead
  - In theory, it would have made no difference, if transform T was the identity (T = local transform of the rigid body)

Future work:
Idea for how to progress 1/4

- Current problem:
  - Each positional constraint is enforced only once per frame
- Fix it: make a global “behavior”
  - Associated to the root of the scene
  - instead of relying on Unity to execute fixed updates of every object, use only the fixed update of the global behavior, making a sequence of loops:
    - 1st loop: execute Verlet integration (loop over all particles)
    - 2nd loop: enforce all positional constraints
      (loop over all particle *and* over all rods in the scene)
    - Repeat 2nd loop multiple times

Future work:
Idea for how to progress: 2/4

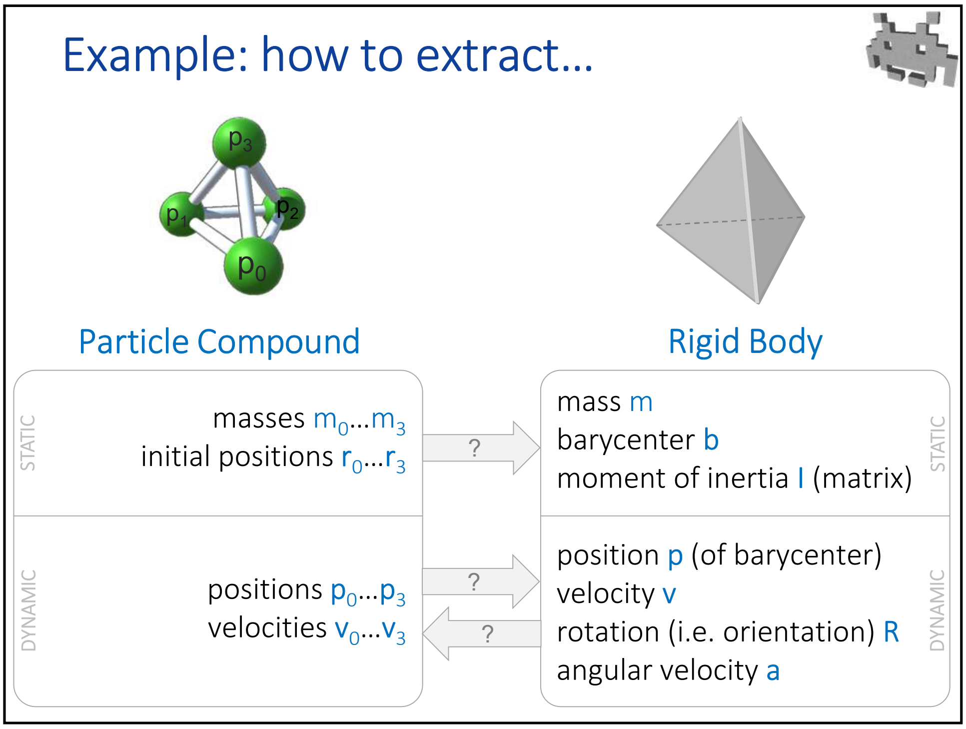
- Add springs
- How to: add spring object (similar to rods)
  - 1. Rest length: computed at start (like for rods)
  - 2. Particles at the extremes: a public field, just as for rods
  - 3. Elasitic constant *k*: a (public) scalar parameters
  - 4. Write fixed update(): add to forces of the two particles
  - 5. Profit! Add spring to your compound meta-objects
- Caveats:
  - Unless you use a global script, you will need to set forces to 0 (InitForces method) at the *end* of the FixedUpdate (not the beginning) and at initialization (why?)

Future work:
Idea for how to progress: 3/4

- Floor is lava (or water)
  - Instead of having a hard-granite floor, make it liquid
- How to:
  - 1. Remove the “stay above ground” constraint
  - 2. Add buoyancy (ita: forza di Archimede) to the particles
    - (as an approximation, you don’t need it for the rods or the rigid objects: just the particles)
    - Reminder: buoyancy is an upward force with a magnitude = mass of the submerged volume if it was made of water
    - Math task: compute the volume of the part of sphere (of a given radius) which has $y > 0$
  - 3. Profit! See how object float, or sink
    - (and which parts stays up if they float) – depends on masses are size

Future work:
Idea for how to progress: 4/4

- Extract emerging behavior of the meta object
- The meta object has its own…:
  - 1. Baricenter (done! – function currentBaricenter)
  - 2. Linear velocity (done! – function averageVelocity)
  - 3. Total mass
  - 4. Angular velocity
  - 5. Moment of inertia
- They are all implicitly updated (emerging behaviour)
- Can we make them explicit, extracting them?



Example: how to extract…

Particle Compound

Rigid Body

STATIC

masses m_0…m_3
initial positions r_0…r_3

mass m
barycenter b
moment of inertia I (matrix)

STATIC

DYNAMIC

positions p_0…p_3
velocities v_0…v_3

position p (of barycenter)
velocity v
rotation (i.e. orientation) R
angular velocity a

DYNAMIC