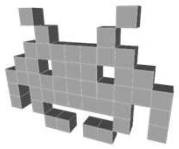
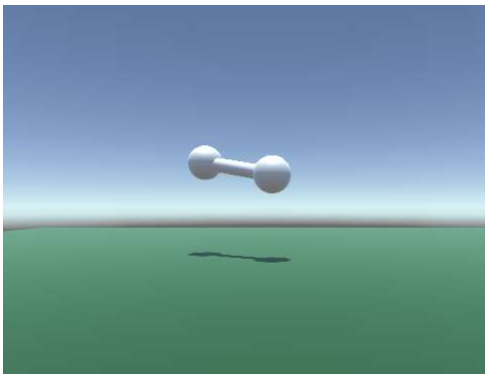
3D video games

notes on the sand-box coding done in class



Marco Tarini



181

Objective of this sandbox

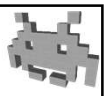
Implement a PBD system

(particle based, with Verlet integration) on Unity

- Plan:
  - we will NOT enable the default Unity physics system
  - instead, implement our ad-hoc physics "by hand", by scripting
  - *note*: in a normal project, there's no good reason to do that!
- How to ***NOT*** enable physics in Unity:
  - Simply keep your "GameObject"s free from:
    any "RigidBody" component (they implement *dynamics*) and
    any "Collider" component (they implement *collision handling*)
- we will still use the Graphics engine of Unity, its GUI, etc
  - and scene-graph support: GameObjects, their Transforms, etc

182

Background: "behaviors" in Unity



- In Unity, a behavior is a script associated to a Game-Object
- It's a C# class (or other languages) with predefined methods employed by the rest of the engine, such as:
  - Start() – called at start at before the first rendering
  - FixedUpdate() – called at every physics steps, just before the hard-wired physics step
  - Update() – called before rendering this object (*if* it is rendered)
- The value *dt* is exposed as Time.FixedDeltaTime

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

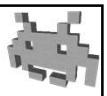
183

Our Particles and their behavior

- Our particle is a game-object
  - an element of the scene graph (1st level – local transf = global tr.)
  - it's rendered as a small sphere
- Its associated behavior class includes fields to record the state:
  - p_old , p_old (two point): state in Verlet dynamics
- and invariants (not constats, we can change them):
  - mass (scalar): a constant
  - drag (another scalar): % of speed lost per second (TODO! A constant for now)
  - making the public, we can manipulate them from the interface
- and the methods:
  - Start(): sets p_old
  - enforceConstraints(): enforces all the constraints affecting this particle alone (stay in box, don't be below ground)
  - oneStep(): performs a Verlet integration step

184

Implementation detail: p_now VS transform.position



- For each particle, the current position is already kept by Unity as its transform.position :
  - Reminder: it's the translation/position component of the **global** transformation
  - (BTW it's not really a field, but it pretends to be – C# "property")
    (for example, Unity will update it from local transforms)
  - Reminder: physical simulation always acts in *world space*!
  - That value is used by the rest of unity, rendering engine, the GUI, etc.
- For clarity, we use a variable p_now instead but we will keep it in sync with transform.position
  - at the beginning of each physics step (method fromUnity):
    p_now ← transform.position
  - at the end (method toUnity):
    transform.position ← p_now

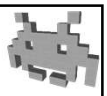
185

Implementation detail: pNow VS transform.position

- For each particle, the current positions is stored twice:
  - The position according to our custom physics engine:
    pNow – 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)

186

Implementation detail: pNow VS transform.position



- When to synchronize graphics (transform.position) and our physics (pNow)?
- Best solution we found in class:
  - In method `Start()`: pNow ← transform.position
    - This makes the physics be initialized with the position set from the Unity GUI
  - At end of `FixedUpdate()`: transform.position ← pNow
    - this makes the engine show the particle on screen at it's correct physics position
  - In `LateUpdate()`: pNow ← transform.position
    - this allows us to control the (physics) position from the GUI
    - Observe: because it's Verlet, by changing the position we control also the velocity. E.g. we can "toss" objects

187

oneStep() method of particles (Verlet integration step)

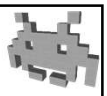
Called by fixedUpdate() (that is, once per frame)

Basic Verlet integration is called here. In it:

- We add forces *that depends only on this one particle*
  - Such as gravity
- We include velocity dumping
  - see dump computation in prev slides

188

OneStep() method of particles (Verlet integration step)



- Called by fixedUpdate() (that is, once per frame)
- We add forces
  that depend on this one particle only
  - That is, gravity
- We include enforcement of positional constraints
  which depend only on this one particle
  - ground collision ("please stay above ground")
  - box collision ("please stay inside this 10x10 box")
- Includes velocity dumping
  - see dump computation in prev slides

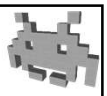
189

Adding "sticks"

- Sticks are GameObjects representing rigid "rods" connecting two particles
- Rendering (just for the looks):
  - A stick is rendered as a small cylinder
    (a cylinder mesh associated to the Game Object)
  - Before each rendering (so, in the Update() method)
    its (global) transformation is computed anew,
    so that the cylinder is scaled, rotated, and translated
    to make it graphically connect the two particles
  - This new transformation replaces the old at every frame
  - (therefore, it doesn't matter where we place them in the scene: they will teleport to the right location at each frame)

190

Adding “sticks”



- Fields:
  - References to connected particles A and B
    This is a public field: that way, we will set them in the Unity GUI !
  - Rest length (scalar)
    This is automatically computed on Start as
    the initial distance (in the scene graph) between particles A and B
- Methods:
  - enforceConstraints():
    forces the positional equidistance constraint
    (on the p_now of the two particles)
  - See slides or code for how this is to be computed from their current positions

191

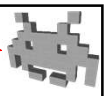
Leaving Unity scheduling: from scripts run in each object…

- In our first version, we executed the entire physical step in the fixedUpdate method of the particle:
  - fromUnity *for that particle* — inherit its position from the rest of the engine
  - physic step *for that particle* — Verlet integration includes gravity force and vel damping
  - constraints enforcing *for that particle* — Single particle constraint
  - toUnity *for that particle* — makes the rest of the engine get our position
- This allowed for a quick start!
  - It’s convenient to let Unity run scripts for each “gameObject” for us
- However, we need to control better the global order of operation

192

Leaving Unity scheduling: …to a global script

a common practice in Unity projects…



- let's add a global script ("PhysicEngine")
  - associated to a dummy object in the scene
  - at starts, collects all particles and sticks etc (in vectors)
- its fixedUpdate method calls, in succession:
  - fromUnity *for **all** particles*
  - cumulate forces *in **all** particles, from any source*
  - physic step *for **all** particles*
  - constraints enforcing *for **all** particles & sticks, etc*
  - toUnity *for **all** particles*
- this allows us to execute things in the correct order & more control
  - We can leave invoking "update" methods before every rendering by Unity

TODO? E.g. spring

potentially, multiple cycles (was impossible before)

193

Sand-box project: results (so far).

- Combining just to particle and a sticks, we constructed a simple Rigid objects that already exhibits a somewhat plausible…
  - Angular velocity
  - Angular momentum
  - Reaction of impacts with the ground / walls (bounces)

…without having explicitly coded any of that.

Plus, it's very stable (constraints enforced every frame)

- Potential todo list:
  - Construct more complex rigid bodies / articulated bodies (composing particles)
  - Account for masses in positional constraints
  - Add forces from other sources

194