3D video games

3D Game Physics

Marco Tarini

2

Course Plan

- lec. 1: Introduction
- lec. 2: Mathematics for 3D Games
- lec. 3: Scene Graph
- lec. 4: Game 3D Physics +
- lec. 5: Game Particle Systems
- lec. 6: Game 3D Models
- lec. 7: Game Textures
- lec. 8: Game 3D Animations
- lec. 9: Game 3D Audio
- lec. 10: Networking for 3D Games
- lec. 11: Artificial Intelligence for 3D Games
- lec. 12: Game 3D Rendering Techniques

3

Animation in games

but, a note on terminology:
in some contexts, procedural means
"produced by a *simple* procedure"
as opposed to "physically simulated"

Non procedural

- Assets!
- Fully controlled by artist/designer (dramatic effects!)
- Realism: depends on artist's skill
- Does not adapt to context
- Repetition artefacts

Procedural

- Physics engine
- Less control
- Physics-driven realism
- Auto adaptation to context
- Naturally repretition free

4

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 *accuracy*)
- robustness
  - (should almost never "explode")

6

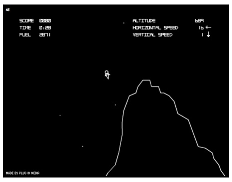
Physics in games: cosmetics or gameplay?

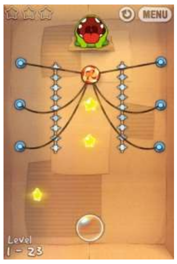
- Just a graphic accessory? (for realism!)
  - e.g.:
    - particle effects (w/o feedback)
    - secondary animations
    - Ragdolling
- Or a gameplay component?
  - e.g. physics based puzzles
  - Popular approach in 2D (since always!)

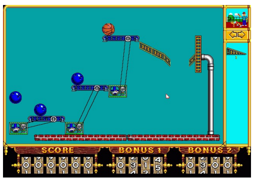
7

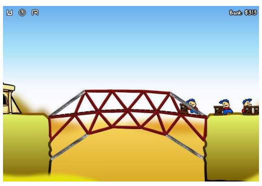
Physics in games: cosmetics or gameplay?

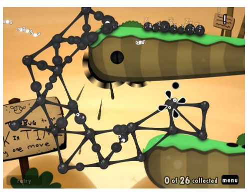
- Just a graphic accessory? (for realism!)
  - e.g.:
    - particle effects (w/o feedback)
    - secondary animations
    - Ragdolling
- Or a gameplay component?
  - e.g. physics based puzzles
  - Popular approach in 2D (since always!)







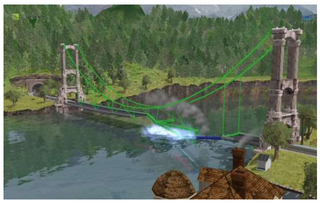




8

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



9

Physics engine: intro

- Game engine module
  - executed in real time at game run-time
- A high-demanding computation
  - on a very limited time budget!
- …but highly parallelizable
  - potentially, highly parallel

==> good fit for hardware support

(just like the Rendering Engine)

10

Hardware for Physics engine

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

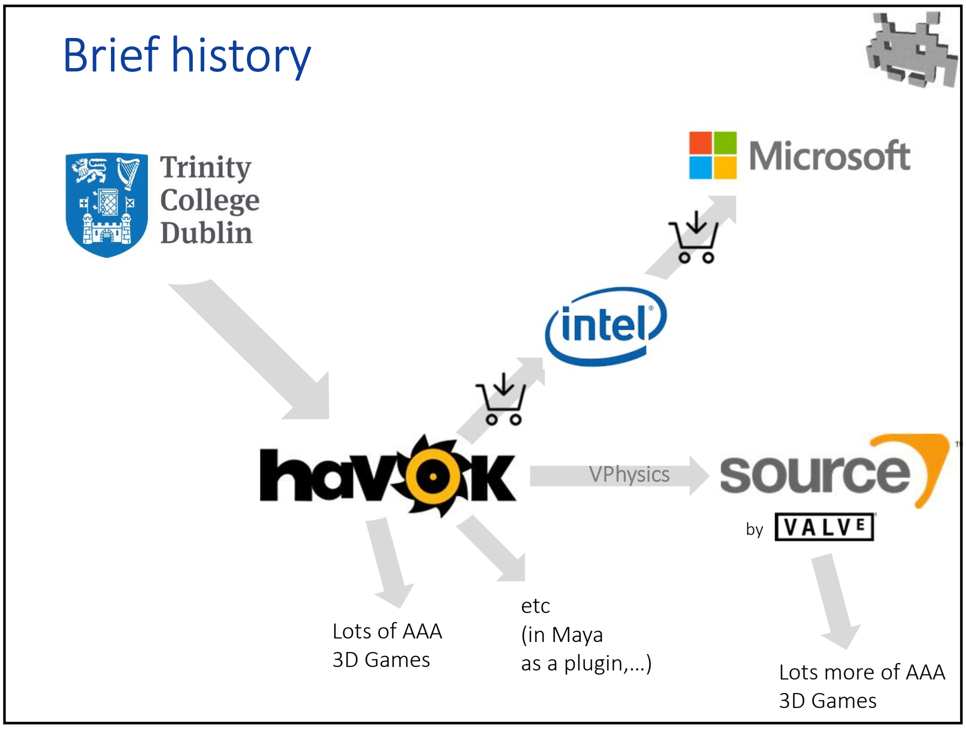
- For a brief moment ~2006: PPU
  - "Physics Processing Unit"
  - HW unit specialized for physics
- Then: GP-GPU
  - "General Purpose Graphics Processing Unit"
    - Use of the graphics card for generic tasks (not related with 3D computer graphics)
  - Ex.: Cuda (nVidia)

11

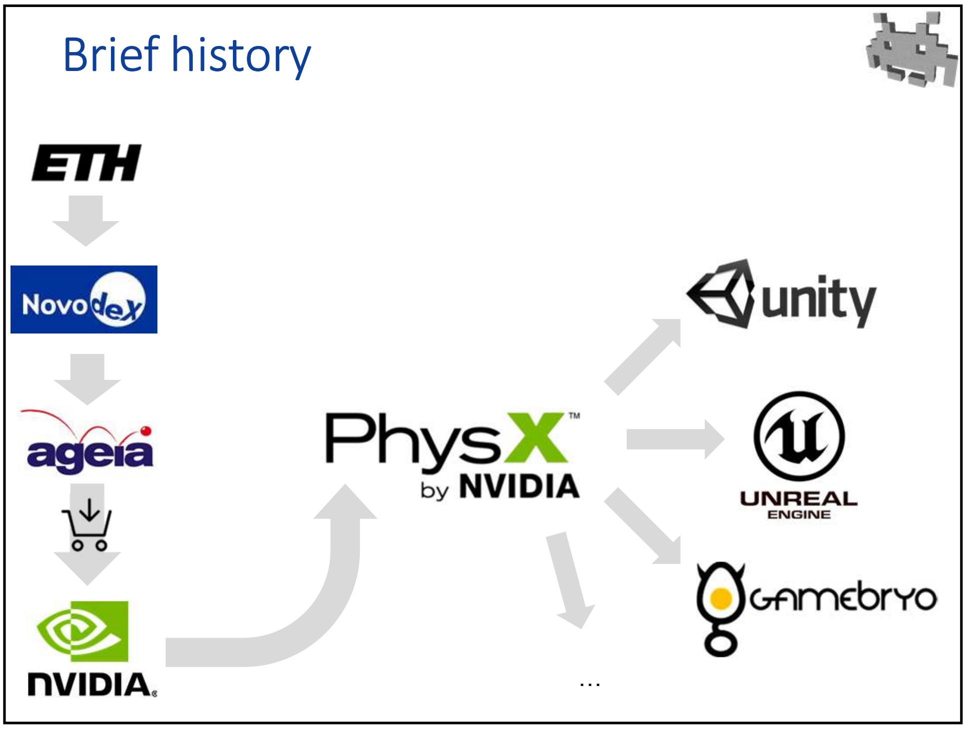
Main Software (libraries, SDK)

havok	mostly CPU (Microsoft)
PhysX	CPU+GPU (CUDA) NVidia
Bullet Physics Library	open source, free, HW accelerated (OpenCL) + CPU
Open Dynamics Engine	open source, free
Box2D	2D, open source, free

12


Brief history
Trinity College Dublin
Microsoft
intel
havok
VPhysics
source
by VALVE
Lots of AAA 3D Games
etc
(in Maya
as a plugin,...)
Lots more of AAA 3D Games

13


Brief history
ETH
NovodeX
ageia
NVIDIA
PhysX
by NVIDIA
unity
UNREAL ENGINE
Gamebryo
...

14

Fields of study

- Dynamics
  - The motion, as a result of forces
  - *"Subject to gravity, how will this pendulum swing?"*
- Statics
  - Equilibrium states, energy minimization states
  - *"In which state(s) can this pendulum be still?"*
- Kinematics
  - The motion itself, irrespective of why it's moving
  - *"If the angular speed of the pendulum is currently X, how fast is the tip moving?"* (or vice versa)

15

The 2 tasks of the Physics engine

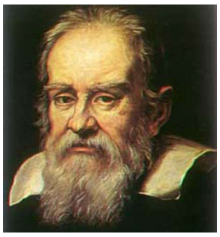
1. Dynamics (Newtonian)
for objects such as:
   - Particles
   - Rigid bodies
   - Articulated bodies
     - E.g. "ragdolling"
   - Soft bodies
     - Ropes (specific solutions)
     - Cloth (specific solutions)
     - Hair (specific solutions)
     - Free-form deformation bodies (general)
   - Fluids
     - Expensive!

2. Collision handling
   - Collision detection
   - Collision response

16

Newtonian Dynamics

- The one with:
  - Masses
  - position and its derivative: velocity
    - and momentum
  - direction and angular velocity
    - and angular momentum
  - forces acceleration…



17

Reminder: Spatial placement 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 * angle or
  3x3 matrix or
  Euler angles

18

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)

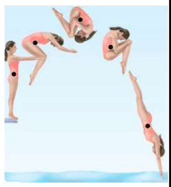
27

Per-object constants

A few quantities associated to each object

- constants: they don't (usually) change
- input of the physical dyniamics simulation, not output

- Mass:
  - resistance to change of velocity



- Moment of Inertia:
  - resistance to change of *angular* velocity
- Barycenter:
  - the center of mass

28

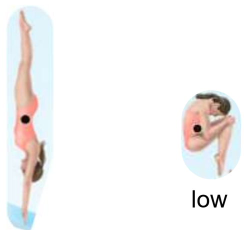
Mass: notes

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

29

Moment of inertia: notes 1/2

- Resistance to change of angular velocity



- (an object rotates around its barycenter)

30

Moment of inertia: notes 2/2

- 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: it's 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 that scalar, for any given axis
  - given an axis **a** (**a** = unit vector), the *moment of inertia* is

$$\mathbf{a}^T \mathbf{A} \, \mathbf{a}$$

  - 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

31

Barycenter: notes

- 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 and often will

32

State of a rigid object in a physical simulation

```
Point position
Rotation orientation
```
current

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

} updated by physics (dynamics)

```
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*

33

In unity

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

```
Vector velocity
Vector angular_velocity
Scalar mass
Matrix moment_of_inertia
Point barycenter
Scalar drag
bool isKinematic
```
the **RigidBody** component

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

34

In unity (using Unity terminology)

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

part of Transform component

```
Vector3 position
Quaternion rotation
```

the RigidBody component

```
Vector3 velocity
Vector3 angular_velocity
float mass
Vector3 inertiaTensor
Quaternion inertiaTensorRotation
Vector3 centerOfMass
float drag
bool isKinematic
```

note: speed = velocity.magnitude

per second (not per frame!)

moment of inertia matrix

the Vector3 = a diagonal matrix D by rotating it $R^T D R$ → the final matrix

the barycenter (in local space)

if true: disable dynamics (but keeps e.g. collisions)

35

State of a point-particle in a physical simulation

```
Point position
Rotation orientation
Vector velocity
Vector angular_velocity
Scalar mass
Matrix moment_of_inertia
Point barycenter
Scalar drag
…
```

(struck through: ~~Rotation orientation~~, ~~Vector angular_velocity~~, ~~Matrix moment_of_inertia~~, ~~Point barycenter~~)

not used !

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

Much simpler: no rotation needed!

We will see later how to still get rigid bodies back.

For now, we focus on this simpler case.

36

Dynamics (Newtonian)

describe the forces
given the particle positions (and more)

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

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

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

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

37

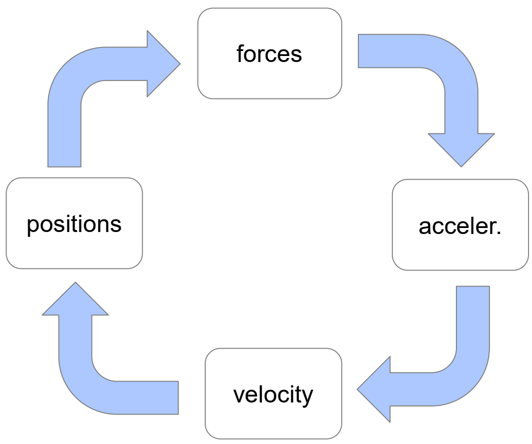
Dynamics (Newtonian)

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

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

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

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



38

An (obvious) precisation

- t_C = virtual time != real time (Wall 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)

39

Computing physics evolution

- **Analytical** solutions:

  state = function(t)

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

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

- **Numerical** solutions:

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

40

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

41

Analytical solutions

that is, find positions as functions p of time t such that…

a given function returning, the forces

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

given

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

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

sometimes, a function of other things too (e.g. velocity). Harder to solve!

the initial conditions (we want to find their evolution!)

42

Numerical integration

- A numerical integrator computes the integral as summed area of small rectangles
  - For a physics engine, this means just updating velocity and positions at each physics step
- A crucial parameter is the width of the rectangles i.e. *dt* = the duration of the physics step (in virtual time)
  - If physics system perform *N* steps per second:
    dt = 1.0 sec / N
  - *N* is not necessarily same rendering frame rate
    e.g.: rendering 30 FPS but physics: 60 steps per seconds
  - *dt* is not necessarily constant during the simulation
    (but in most system, it is)

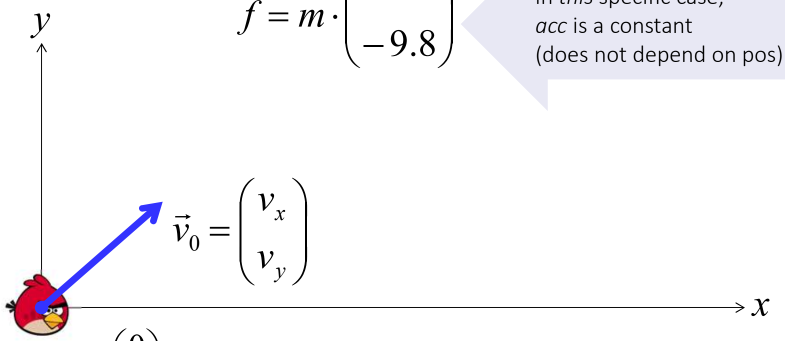
43

Simple example: analytical solution

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

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

in *this* specific case,
acc is a constant
(does not depend on pos)



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

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

44

Simple example: analytical solution

Solving…

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

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

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

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

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

45

Simple example: analytical solution

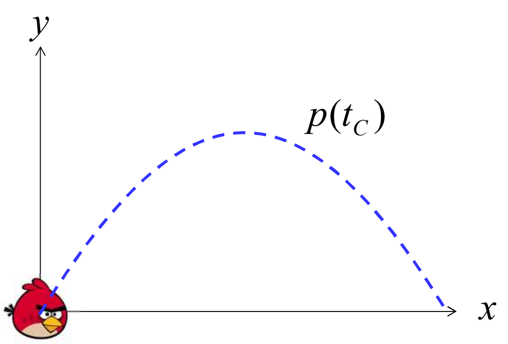
Final result:

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

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

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

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



46

Numerical methods: features

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

47

Euler integration methods

For each step:

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

(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)

Assumption: a

48

Euler integration methods

init state

$$\mathbf{p} \leftarrow \cdots$$
$$\vec{v} \leftarrow \cdots$$

one step

$$\vec{f} \leftarrow fun(\mathbf{p}, \ldots)$$
$$\vec{a} \leftarrow \vec{f}/m$$
$$\mathbf{p} \leftarrow \mathbf{p} + \vec{v} \cdot dt$$
$$\vec{v} \leftarrow \vec{v} + \vec{a} \cdot dt$$

$$t = t + dt$$

49

Forward Euler *pseudo code*

```
Vec3 position = …
Vec3 velocity = …

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

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

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

Equivalent to…

$$\vec{f}_i = function(p_i, \ldots)$$
$$\vec{a}_i = \vec{f}/m$$
$$\vec{v}_{i+1} = \vec{v}_i + \vec{a}_i \cdot dt$$
$$p_{i+1} = p_i + \vec{v}_i \cdot dt$$

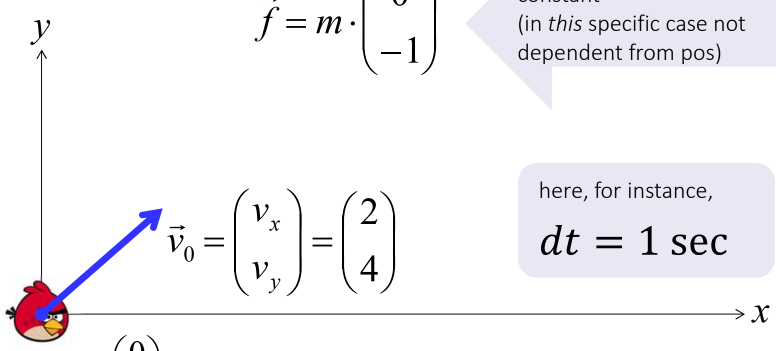
50

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,

$dt = 1$ sec

51

Simple example: numerical solution

init

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

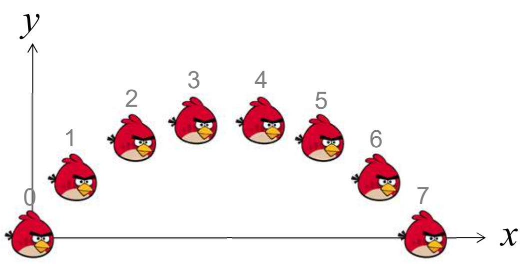
step step step step step step step step

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

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

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

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



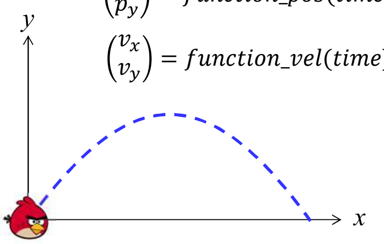
52

Physics evolution computation

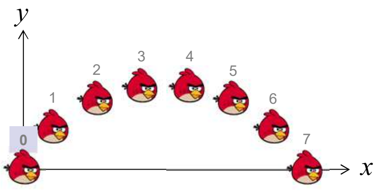
- Analytical solutions:

$$\begin{pmatrix} p_x \\ p_y \end{pmatrix} = function\_pos(time)$$

$$\begin{pmatrix} v_x \\ v_y \end{pmatrix} = function\_vel(time)$$



- Numerical solutions:



53

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

54

Integration errors

- A numerical integrator only approximates
  the real value of the integrals
- The discrepancy (simulation errors) accumulate
  with virtual time
  during all the simulation
- How much error is accumulated?
- It depends on *dt* !
  - Small *dt* ⇒ more steps needed (for same virtual time)
    ⇒ more computationally expensive,
    but smaller errors, i.e. more accurate simulation

55

Order of convergence

- How much does the total error decrease
  as *dt* decreases?
    - That's called the Order of the simulation
  - 1^{st} order: the total error can be as large as O(dt^1)
    - "if the number of physics steps doubles
      (physical computation effort doubles)
      dt becomes halves and errors can be expected to halve"
    - The error introduced by each single step is O(dt^2),
  - The Euler seen is 1^{st} order
    - This is not too good, we want better
    - Note: The error is usually not that bad as linear with *dt*,
      but they *can* be

56

The integration steps *dt* of any numerical methods (summary)

dt : delta of virtual time from last step

- the "temporal resolution" of the simulation!

- if large: more efficiency
  - fewer steps to simulate same amount of virtual time
- if small: more accuracy
  - especially with strong forces and/or high velocities
- Common values: 1 sec / 60 … 1 sec / 30
  - i.e. a step simulates around 16 … 32 msec. of virtual time
  - note: it's not necessarily the same refresh rate of rendering (FPS of rendering ≠ FPS of physics. Rendering can be *less*!)
  - note: di dt is not necessarily the same in all physics steps (need more accuracy *now*? Decrease *dt*

number of physics steps per sec, or «physics FPS»

57