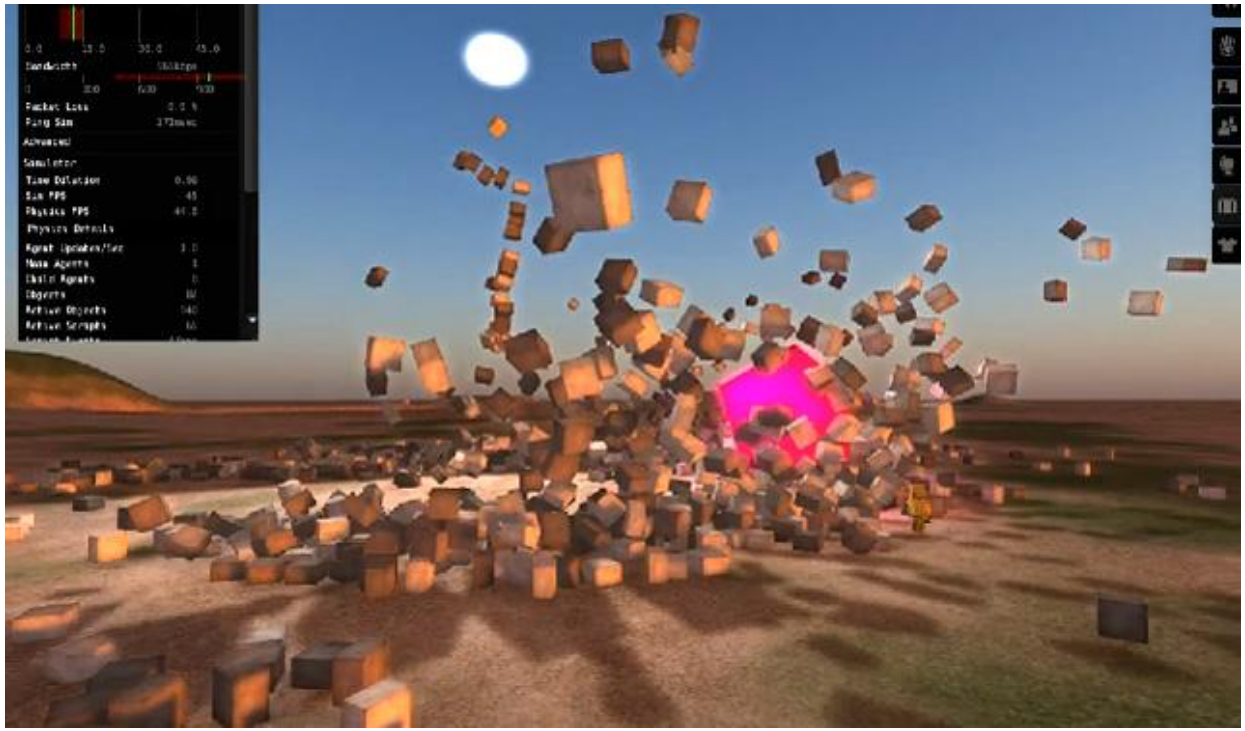
Physics Engines II

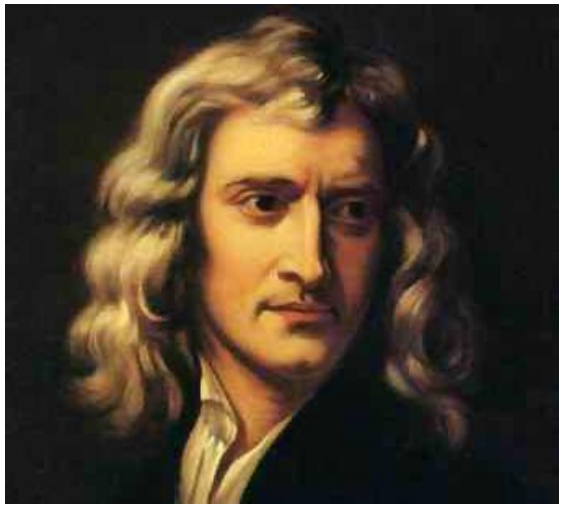
Dynamics



Stefan Bornhofen

Classical Mechanics

Three laws describing the relationship between the forces influencing a rigid body and the resulting motion of this body.



Newton's laws of motion:

1. **Law of inertia:**

   - A body in motion will remain in motion unless a net force is exerted upon it.

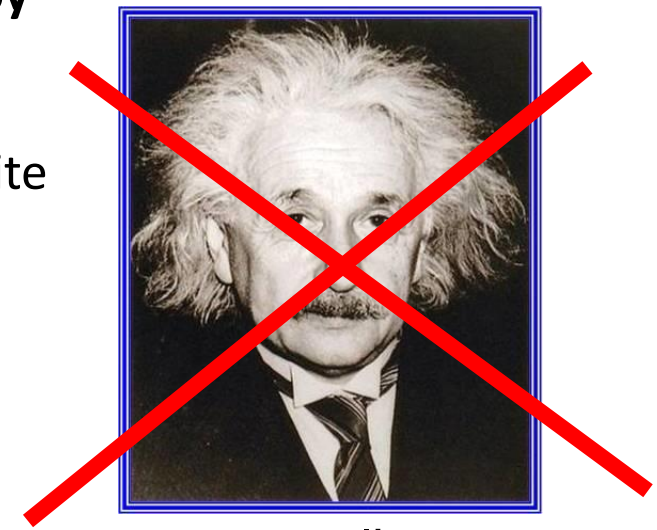
2. **Law of acceleration:**

   - The force on a body is its mass multiplied by its acceleration (F = ma).

3. Law of reciprocal actions:

   - To every action there is an equal and opposite reaction (not used in most physics engines).

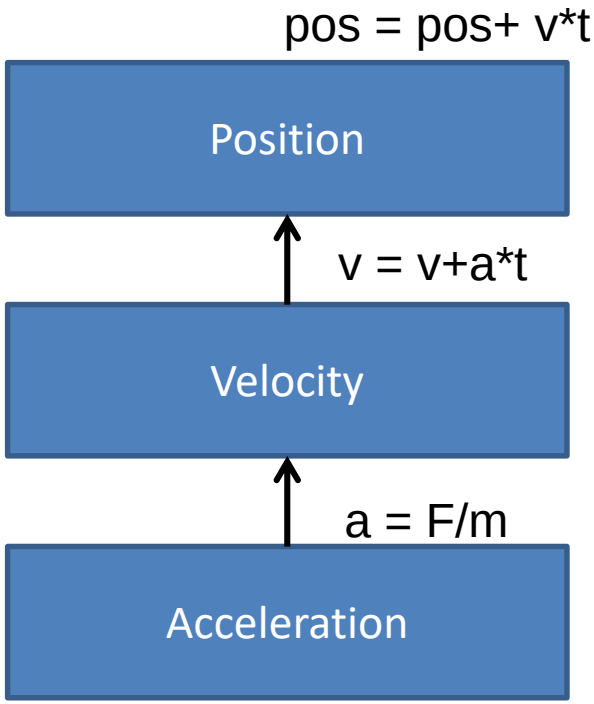
*Contemporary science shows that these laws break down when trying to describe the universe, but they are **good enough**.*



sorry, Albert.

Linear motion

Force can be
- Friction
- Drag: air or fluid
- Springs
- Electromagnetic
- Magic
- User interaction
- (Gravity: $f = G\frac{m_1 m_2}{r^2}$)

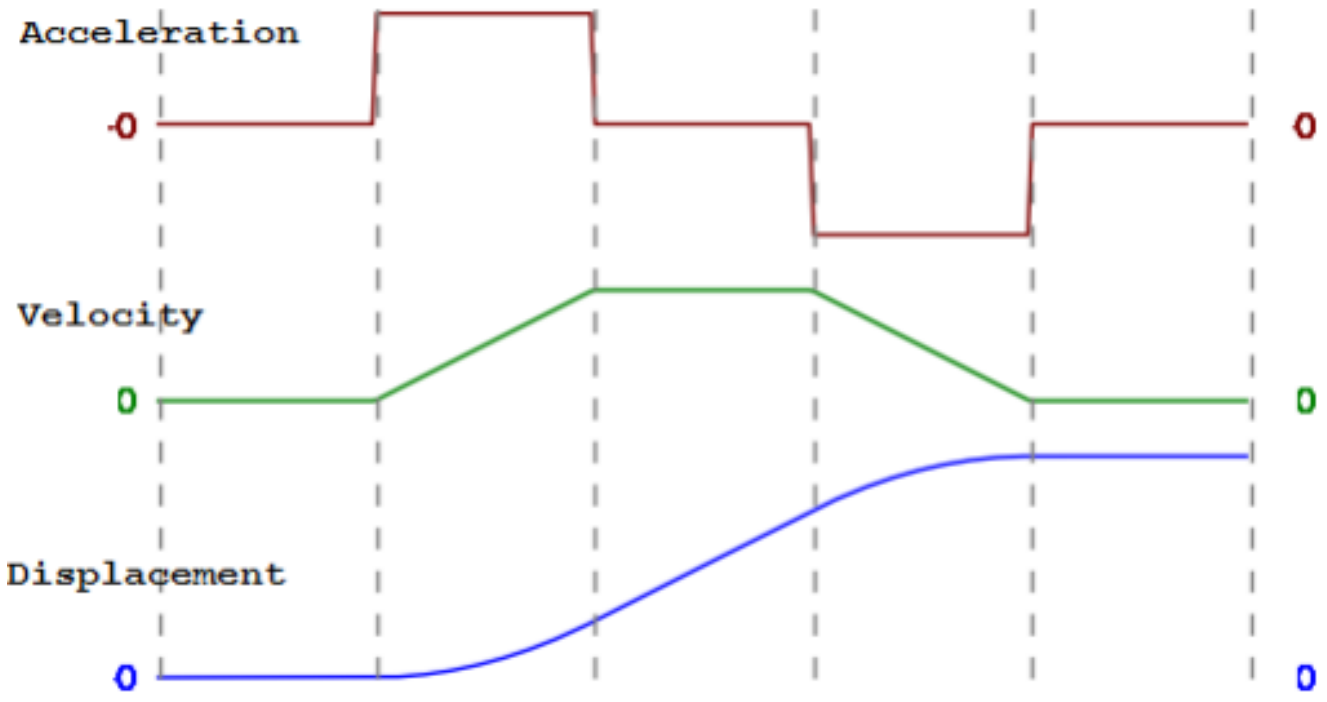


Cycle of motion:

1) force causes acceleration
2) acceleration causes change in velocity
3) velocity causes change in position

$$a(t) = \frac{d}{dt}v(t) = \frac{d^2}{dt^2}p(t)$$

Linear motion



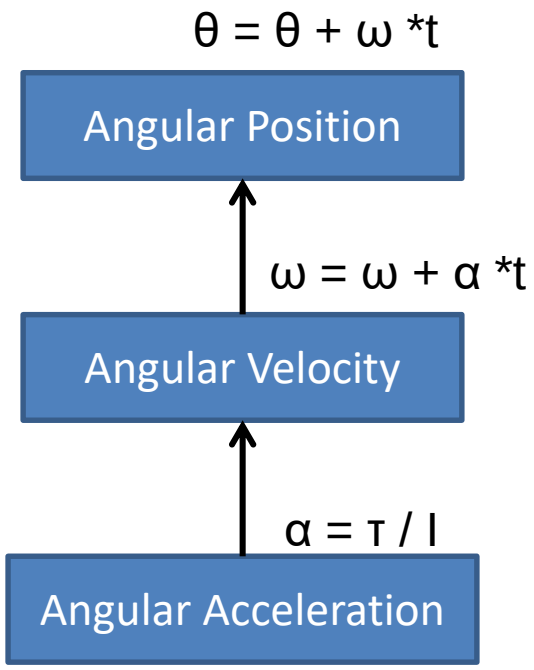
Relationship between acceleration, velocity and displacement

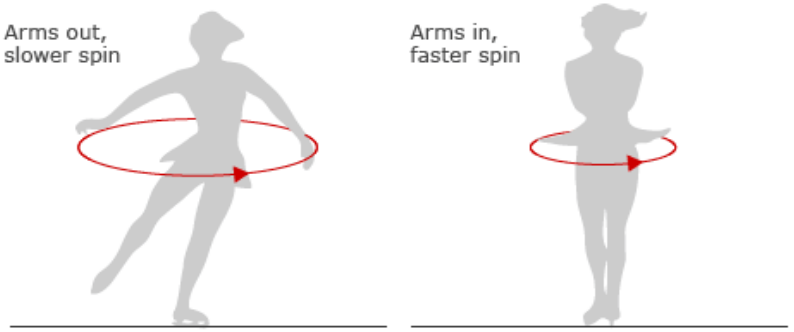
Angular motion

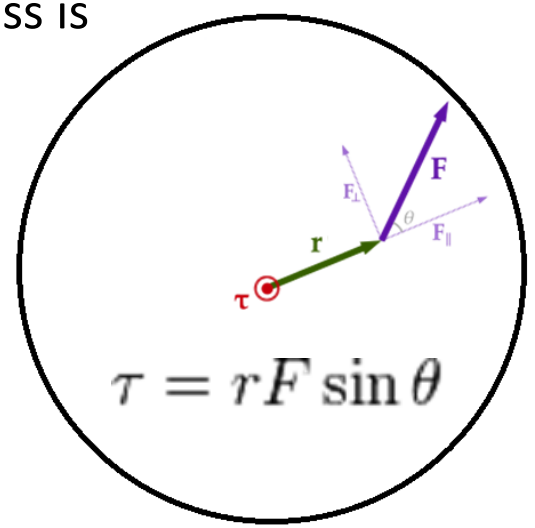
Same principle for angular movement

- θ : angular position
- ω : angular velocity
- α : angular acceleration
- τ : torque (« angular force »)
- I: moment of inertia (equivalent to mass):

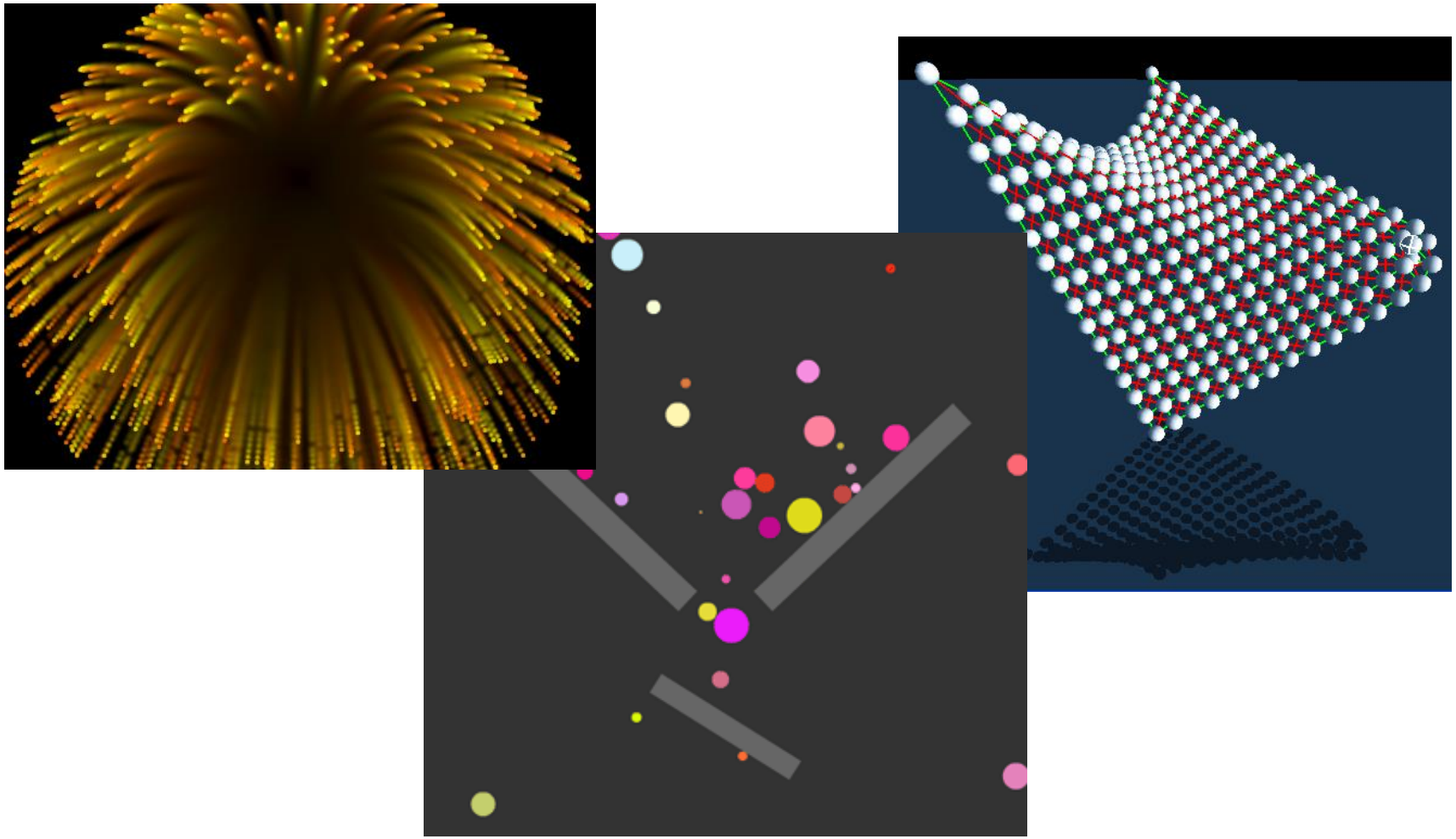
The torque will generate a rotation with respect to the mass center. The **moment of inertia** is a measure of how difficult it is to start it spinning, or to alter an object's spinning motion. It depends on the mass of an object, but it also depends on how that mass is distributed relative to the axis of rotation.





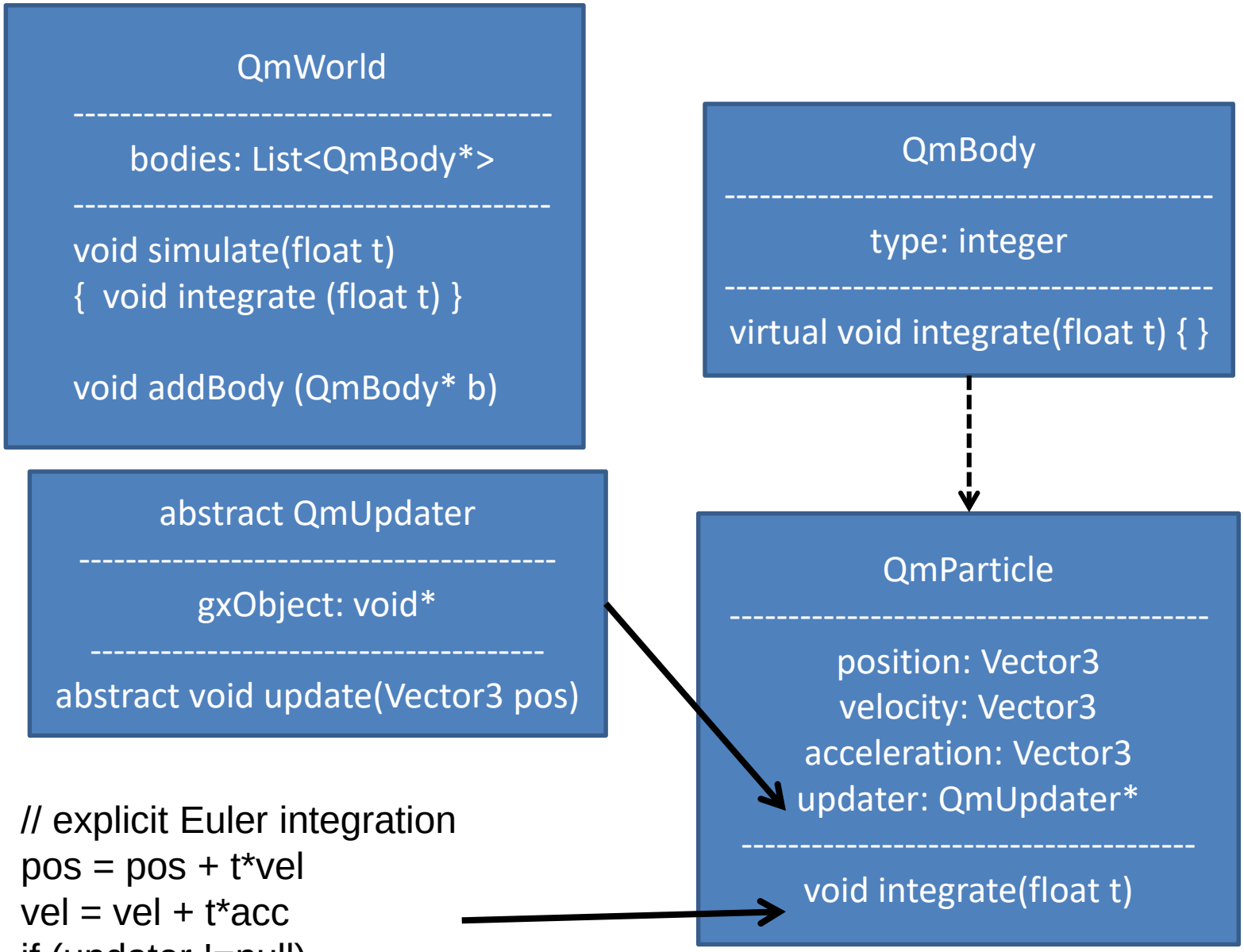


Our Physics Engine



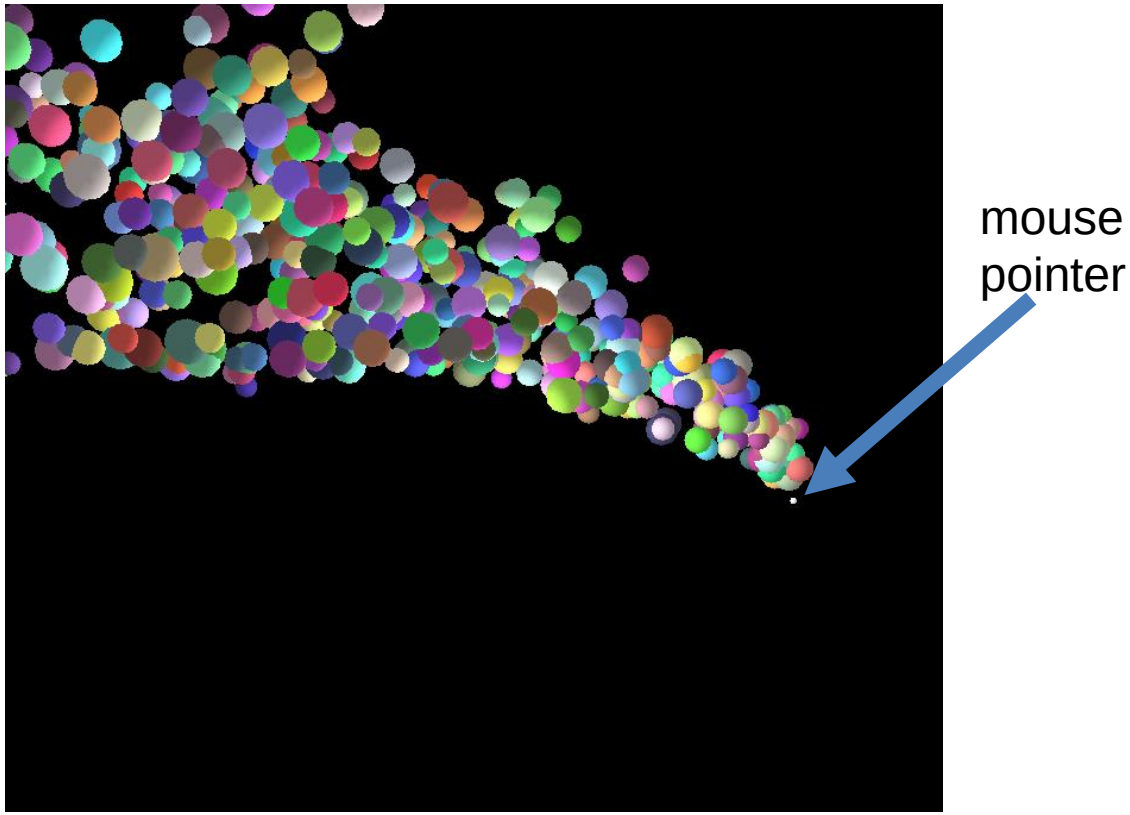
Visual Studio setup

- Unzip the starter kit
- Create a solution with two projects
  - Quantum (Win32 static library):
    your physics engine
    - Copy Quantum/*.* into project folder and add to project
    - Link with glm library
  - Application (Win32 console application):
    an executable using your physics engine
    - Copy Application/*.* into project folder and add to project
    - Add => Reference: Quantum
    - Project Properties => Additional Include Directory: ..\Quantum
    - Link with GLUT and glm libraries
- Define dependency
  - Solution Properties => Project Dependencies:
    Application depends on Quantum
- Set Application as your Startup project


QmWorld
bodies: List<QmBody*>
void simulate(float t)
{ void integrate (float t) }
void addBody (QmBody* b)
QmBody
type: integer
virtual void integrate(float t) { }
abstract QmUpdater
gxObject: void*
abstract void update(Vector3 pos)
QmParticle
position: Vector3
velocity: Vector3
acceleration: Vector3
updater: QmUpdater*
void integrate(float t)
// explicit Euler integration
pos = pos + t*vel
vel = vel + t*acc
if (updater !=null)
updater->update(pos)

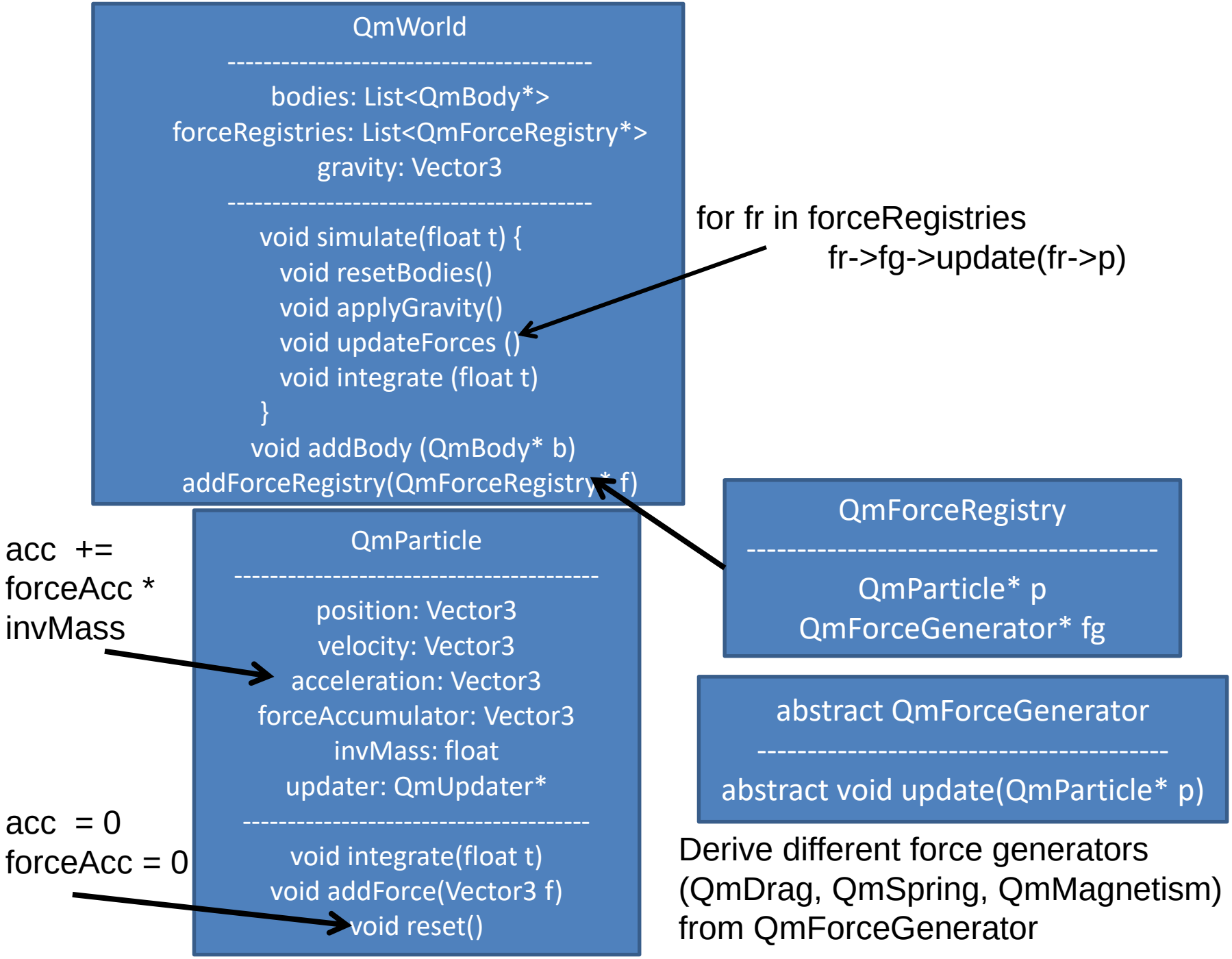
Scene 1

Implement explicit Euler integration for particles with initial position, initial velocity and gravity as constant acceleration. Create a particle fountain coming out of the mousepointer.



Forces

- Acceleration is a result of forces (F = m*a)
- Calculate the acceleration for every frame considering the **sum** of all applied forces and the mass of the particle (D'Alembert's principle)
- Some force generators
  - Drag: $|F| = k1*|v| + k2*|v|^2$
  - Springs: $|F| = K*(l - l0)$
  - Magnetism: $|\boldsymbol{F}| = k_e \frac{|q_1 q_2|}{r^2}$
- Gravity $|F| = m*g$ is a particular force that is optimized in most physics engines: g is directly applied to all objects (as acceleration).


QmWorld
bodies: List<QmBody*>
forceRegistries: List<QmForceRegistry*>
gravity: Vector3
void simulate(float t) {
void resetBodies()
void applyGravity()
void updateForces ()
void integrate (float t)
}
void addBody (QmBody* b)
addForceRegistry(QmForceRegistry* f)
for fr in forceRegistries
fr->fg->update(fr->p)
QmForceRegistry
QmParticle* p
QmForceGenerator* fg
QmParticle
position: Vector3
velocity: Vector3
acceleration: Vector3
forceAccumulator: Vector3
invMass: float
updater: QmUpdater*
void integrate(float t)
void addForce(Vector3 f)
void reset()
acc += forceAcc * invMass
acc = 0
forceAcc = 0
abstract QmForceGenerator
abstract void update(QmParticle* p)
Derive different force generators (QmDrag, QmSpring, QmMagnetism) from QmForceGenerator

Force Generator: QmDrag

QmDrag update (Particle* p):

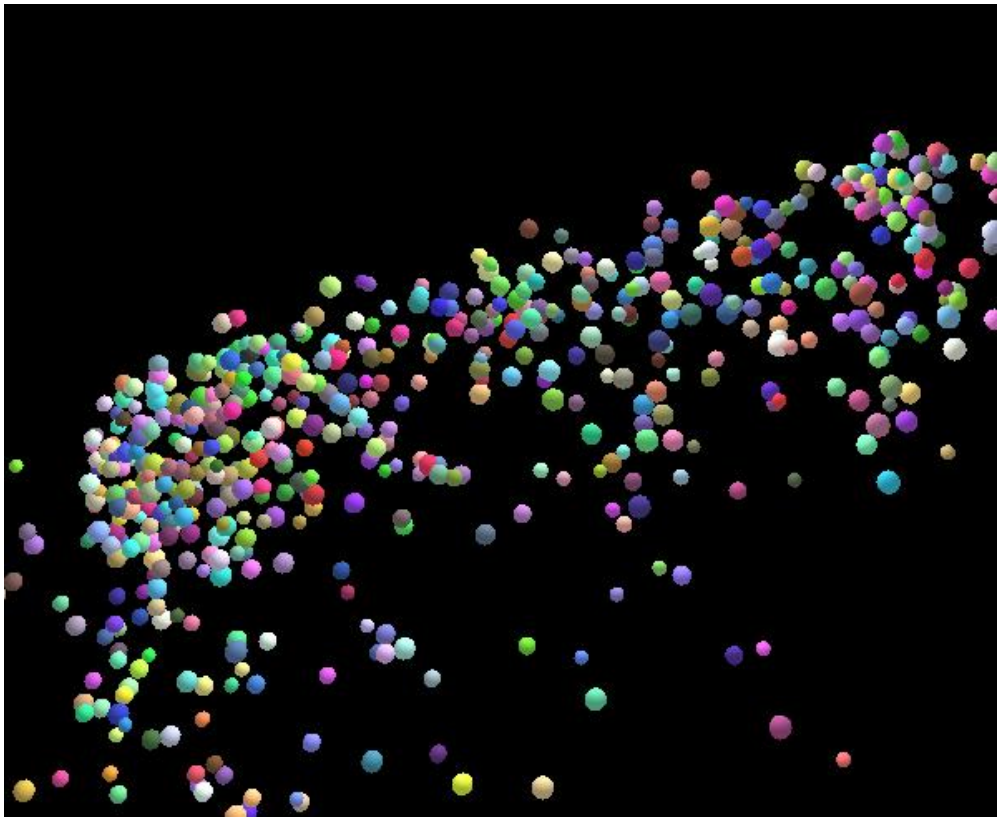
```
float coeff = - (K1*|p->vel| + K2*|p->vel|²)
p->addForce(normalize(p->vel)*coeff);
```

Scene 1

Add random drag forces to the particles.

Model gravity as a constant acceleration (and not as a force).

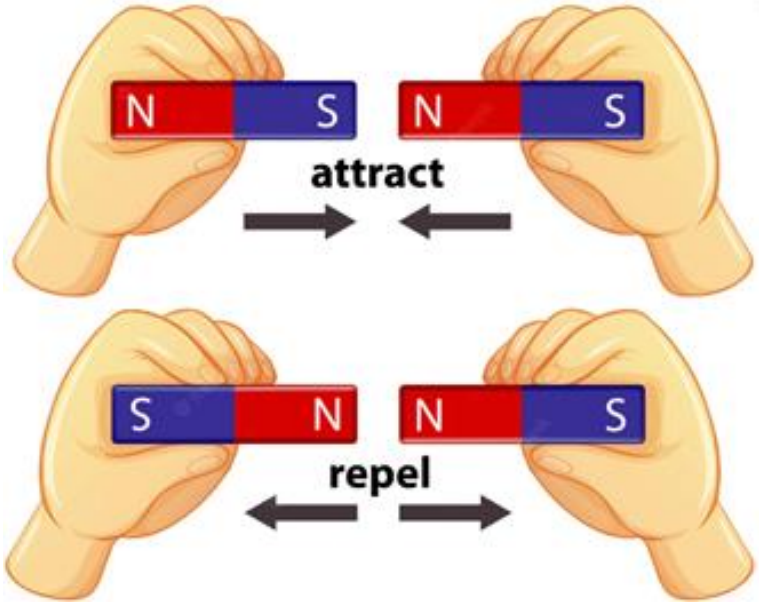
Toggle gravity on/off.



Force Generator: QmMagnetism

QmMagnetism update (Particle* p):

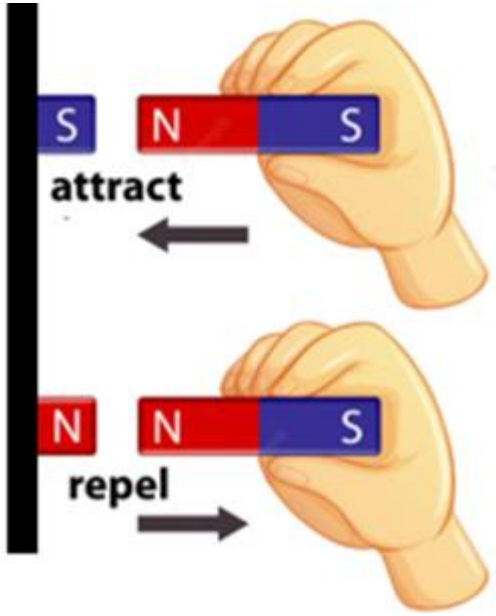
```
Vector3 d = p->pos – OTHERBODY->pos
float coeff = K*(p->charge*OTHERBODY->charge)
p->addForce(normalize(d)*coeff/(|d|²+ EPS));
```



Force Generator: QmFixedMagnetism

QmFixedMagnetism update (Particle* p):

```
Vector3 d = p->pos – FIXEDPOS
float coeff = K*(p->charge * FIXEDCHARGE)
p->addForce(normalize(d)*coeff/(|d|²+ EPS));
```

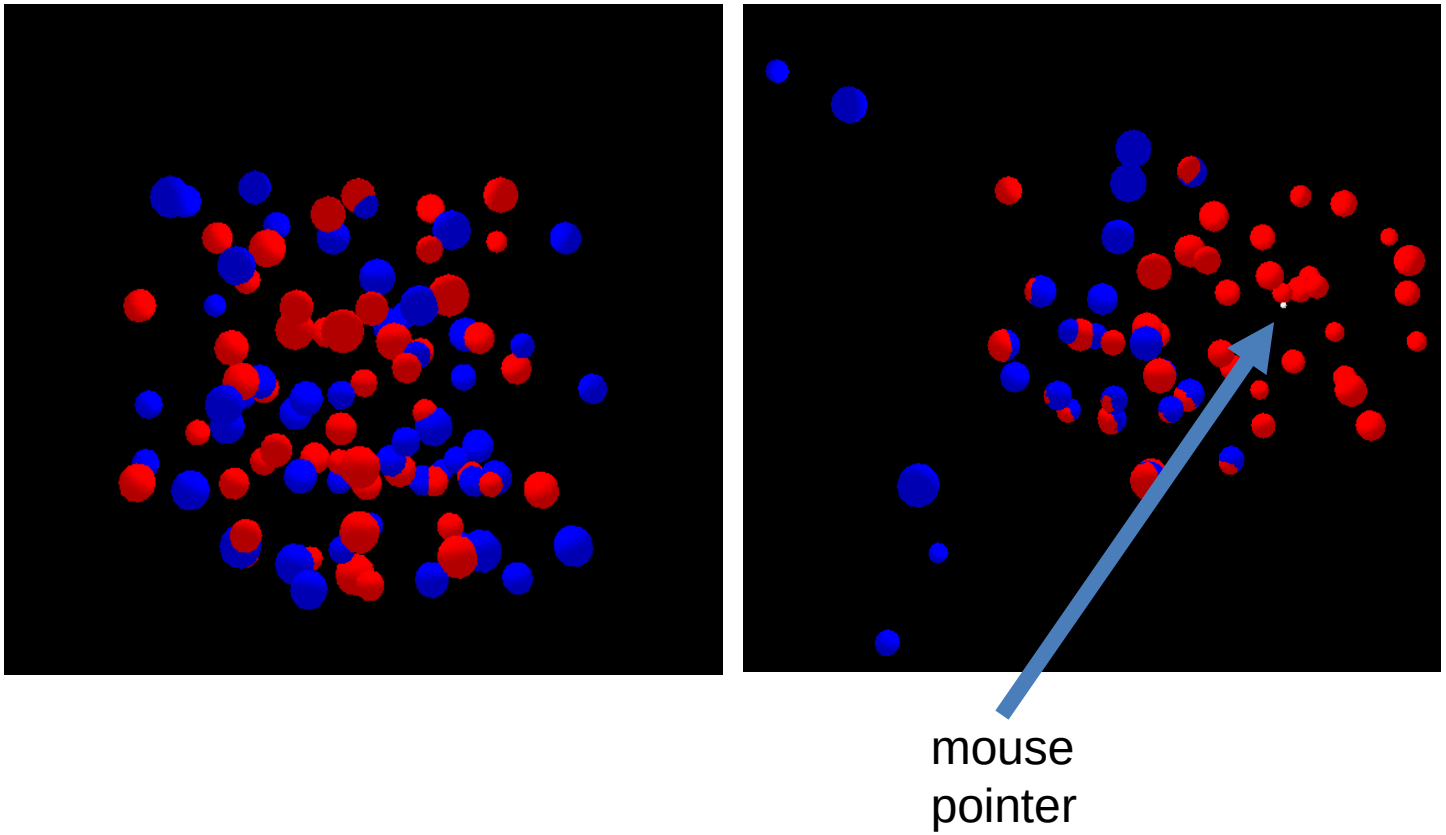


Scene 2

Create magnetic particles.

There are two types of particles with opposite charges.

Charge/uncharge the mousepointer.



Problem 1

The simulation does not converge.

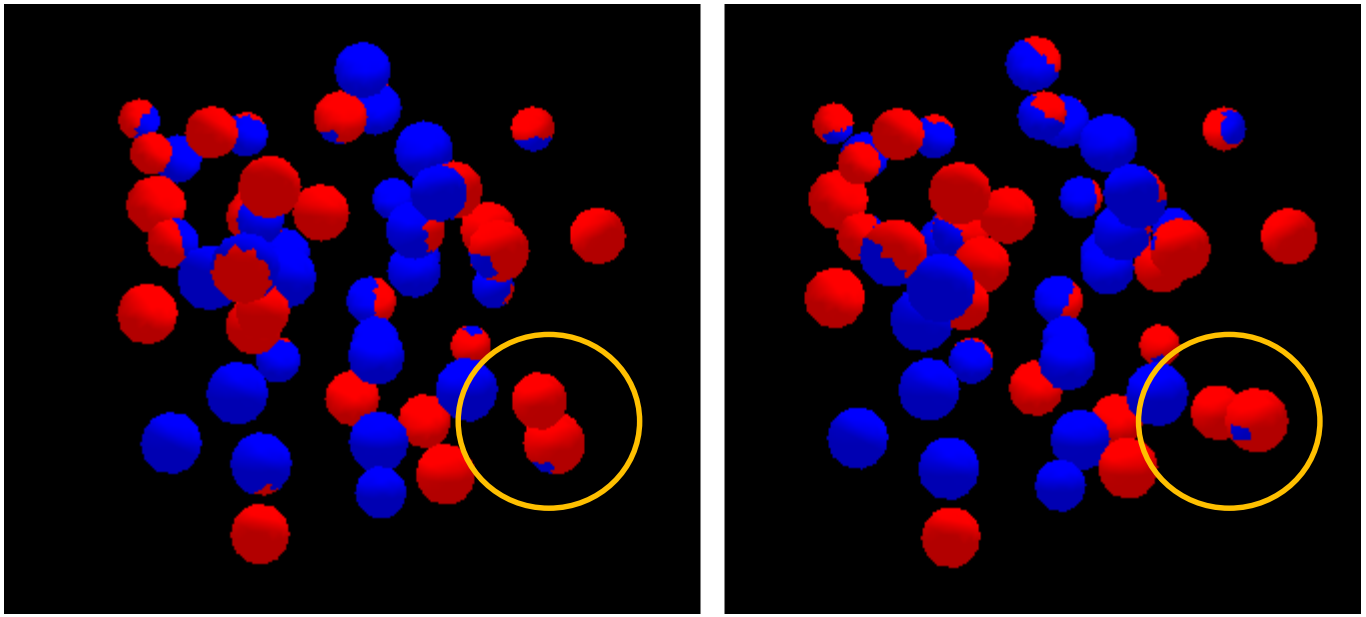
During the integration step, we add a damping coefficient to the velocity:

```
pos = pos + t*vel
vel = vel + t*acc
vel *= damping
```

Damping introduces a slight but steady deceleration of the particles, representing the unavoidable energy loss of the real world. At the same time it contributes to the stability of the simulation (cf. scene 3: springs).

Problem 2

From an identical initial configuration, each simulation run has a different outcome!



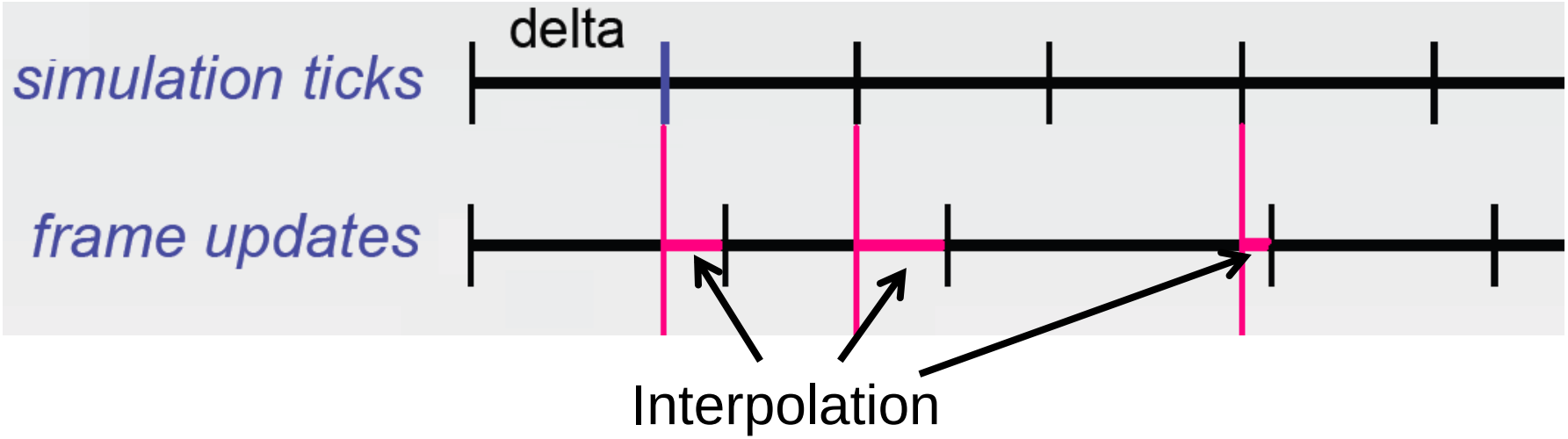
How do we achieve deterministic behavior?

We add framerate independence.

Frame Rate Independence

- Complex numerical simulations used in physics engines are sensitive to time steps
  (due to truncation error and other numerical effects)
- However in many applications the results should be repeatable

Solution: control simulation interval separately



Simulation Ticks

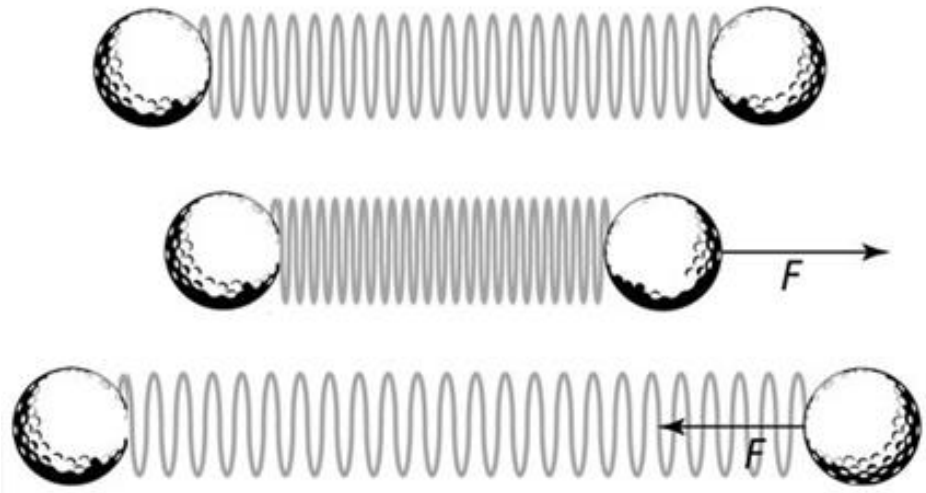
```
world::time = 0 // current simulation time
world::ticktime = 0 // timestamp of last tick

function world::tick(float t): float
    // former world::simulate
    resetBodies(); applyGravity(); updateForces(); integrate(t)
    ticktime += t
    return time-ticktime // the remaining time interval
end
procedure world::interpolate(float dt)
    for p in particles
          p->updater->update(p->pos + dt*p->vel)
end
procedure world::simulate(float t)
    time += t
    dt = time-ticktime
    if (useDELTA) { // deterministic framerate-independent simulation
          while (dt >= DELTA)
                dt = tick(DELTA)
          interpolate(dt)
    } else { // old fashioned all-frame non-deterministic simulation
          tick(t)
          interpolate(0)
    }
end
```

Force Generator: QmSpring

QmSpring update (Particle* p):

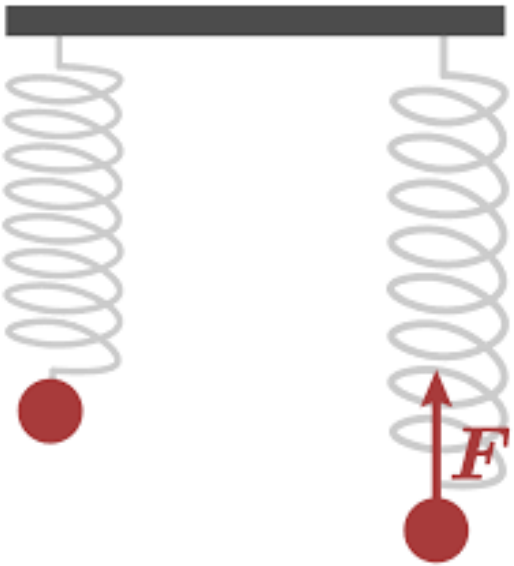
```
Vector3 d = p->pos – OTHERPARTICLE->pos
float coeff = -(|d|-RESTLENGTH)*SPRINGCONSTANT
p->addForce(normalize(d)*coeff);
```



Force Generator: QmFixedSpring

QmFixedSpring update (Particle* p):

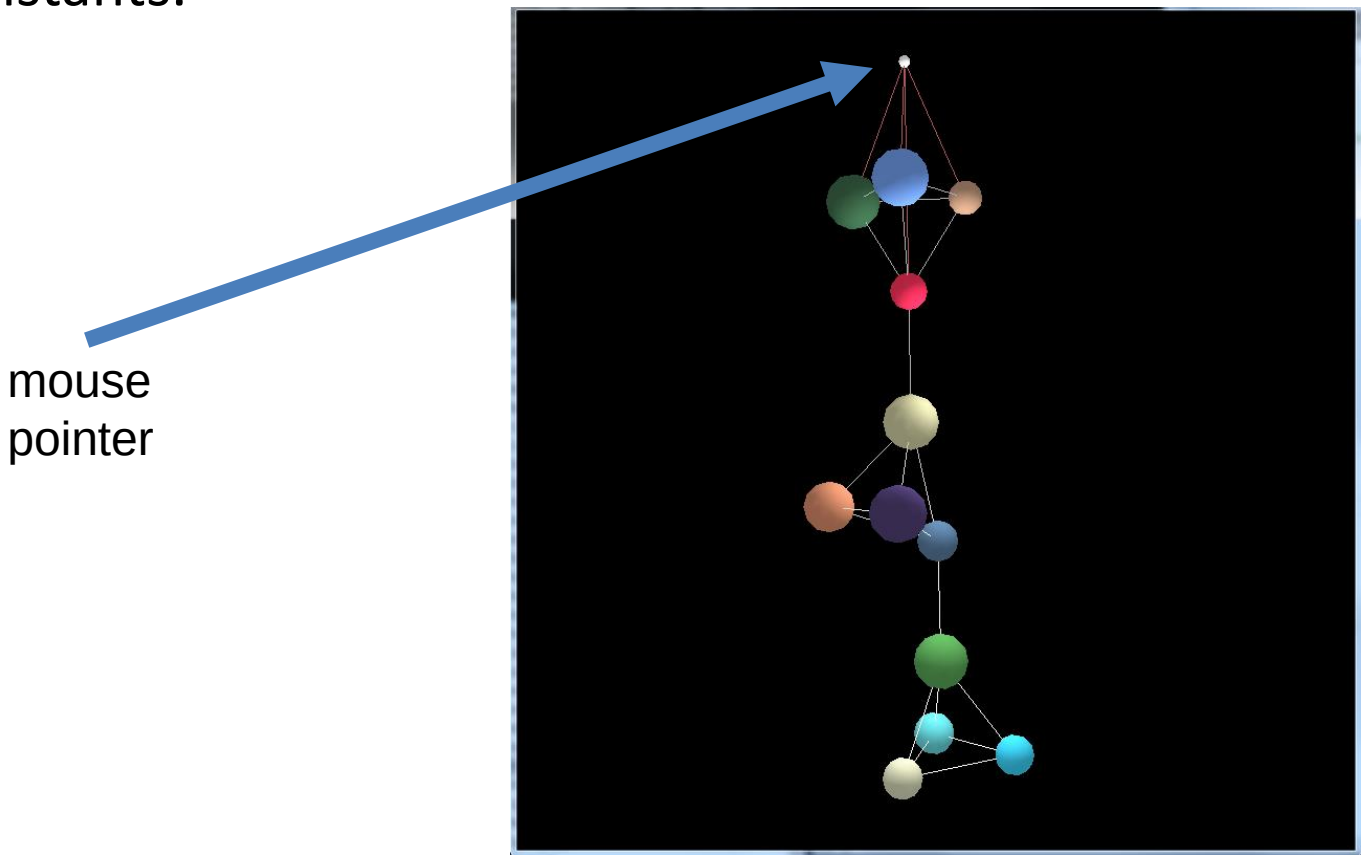
```
Vector3 d = p->pos – FIXEDPOS
float coeff = -(|d|-RESTLENGTH)*SPRINGCONSTANT
p->addForce(normalize(d)*coeff);
```



Scene 3

Create a spring network with particle-particle springs and particle-fixed point springs.

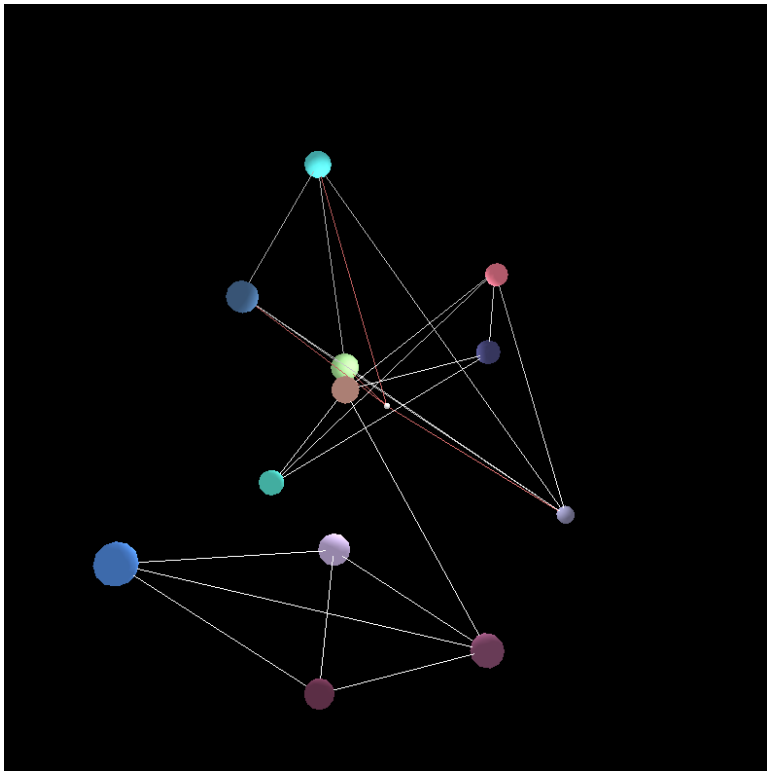
The user can change the particle damping and the spring constants.



Problem

Even with damping, the simulation gets unstable for high spring constants.

We implement a better numerical integrator.

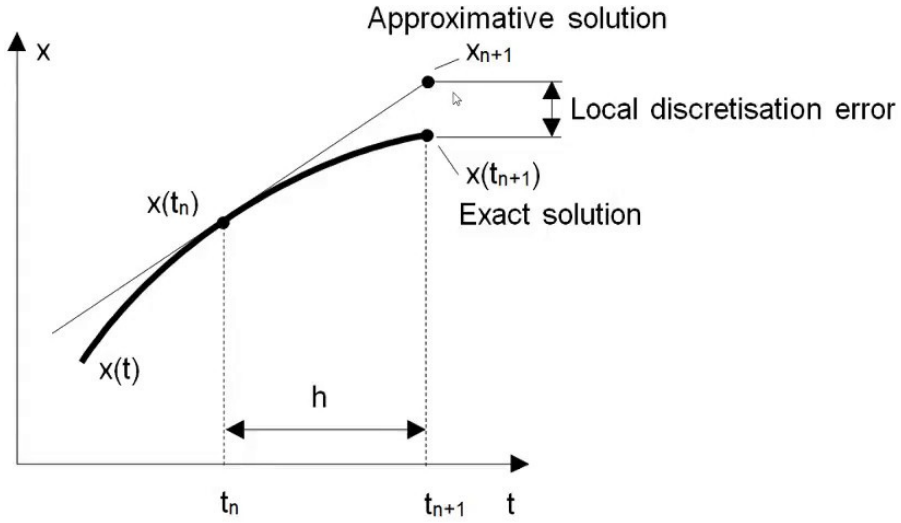


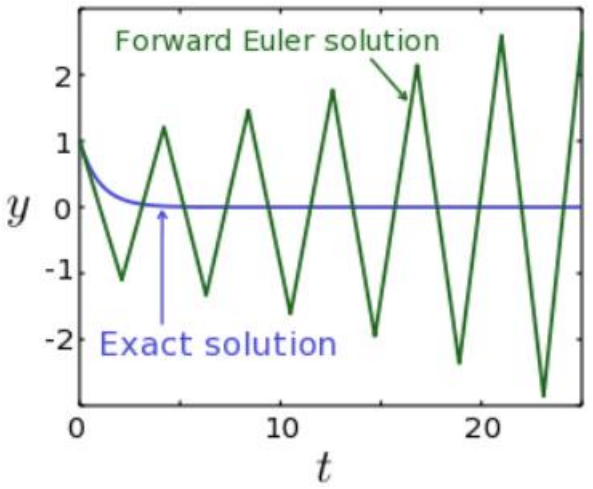
Numerical Integrators

In computer simulations using time steps, continuous ODEs are replaced with discrete equations. **Numerical integration** comprises a broad family of algorithms to solve these ODE.

The **discretization error** is the difference between the exact analytical solution of the ODE and the approximative numerical solution.

Stability is a fundamental problem for the integration of ODEs. The equations which tend to destabilize numerical ODE solvers are often those containing spring constants which are large compared to the time step. Such equations are called **stiff equations**.



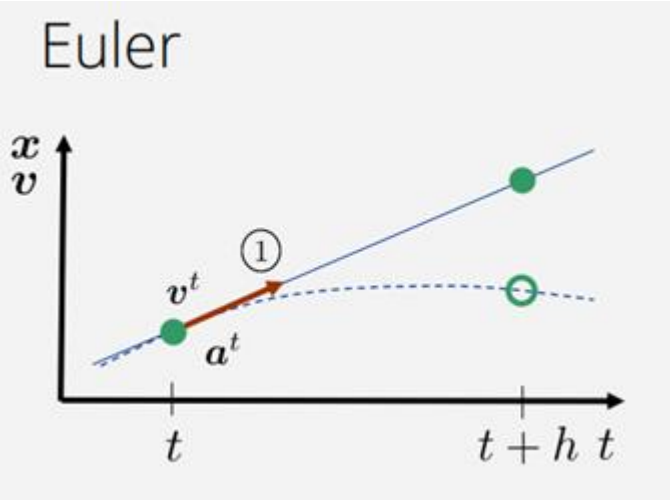


First Order Integrators

Forward (explicit) Euler

```
position += velocity*dt;
velocity += acceleration*dt;
```

In many systems, the energy increases steadily when the forward Euler method is applied (“instability”).



Semi-Implicit Euler

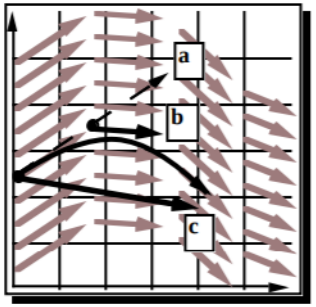
```
velocity += acceleration*dt;
position += velocity*dt;
```

The semi-implicit Euler method conserves the energy in many systems (“symplectic integrator”). This simple method works acceptably well if the step is not too big and the accelerations not too high.

The local discretization error at each step of the Euler methods is $O(h^2)$, and the global error $O(h)$.

Second Order Integrators

- Midpoint



a. Compute an Euler step

$$\Delta \mathbf{x} = \Delta t\, \mathbf{f}(\mathbf{x},t)$$

b. Evaluate f at the midpoint

$$\mathbf{f}_{mid} = \mathbf{f}\left(\frac{x + \Delta x}{2}, \frac{t + \Delta t}{2}\right)$$

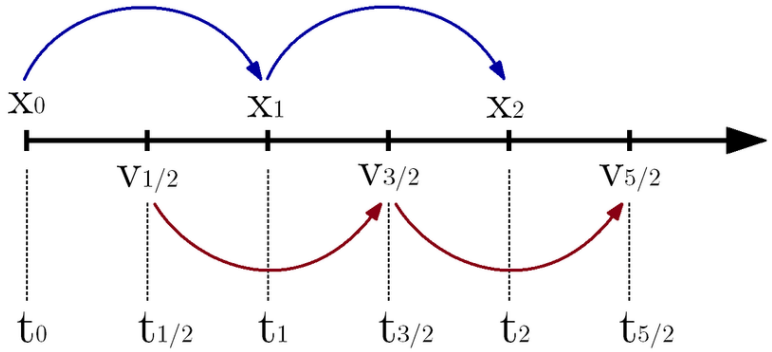
c. Take a step using the midpoint value

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \mathbf{f}_{mid}$$

- Velocity Verlet

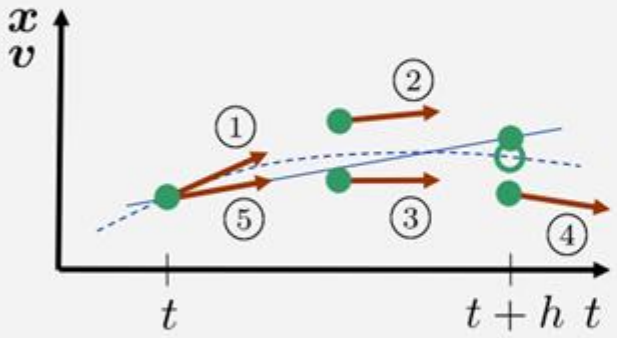
1. Calculate $\mathbf{v}\left(t + \frac{1}{2}\Delta t\right) = \mathbf{v}(t) + \frac{1}{2}\mathbf{a}(t)\,\Delta t.$
2. Calculate $\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}\left(t + \frac{1}{2}\Delta t\right)\,\Delta t.$
3. Derive $\mathbf{a}(t + \Delta t)$ from the interaction potential using $\mathbf{x}(t + \Delta t)$.
4. Calculate $\mathbf{v}(t + \Delta t) = \mathbf{v}\left(t + \frac{1}{2}\Delta t\right) + \frac{1}{2}\mathbf{a}(t + \Delta t)\Delta t.$

- Leapfrog



The local discretization error of second order methods is $O(h^3)$, and the global error $O(h^2)$.

Runge Kutta 4th Order (RK4)

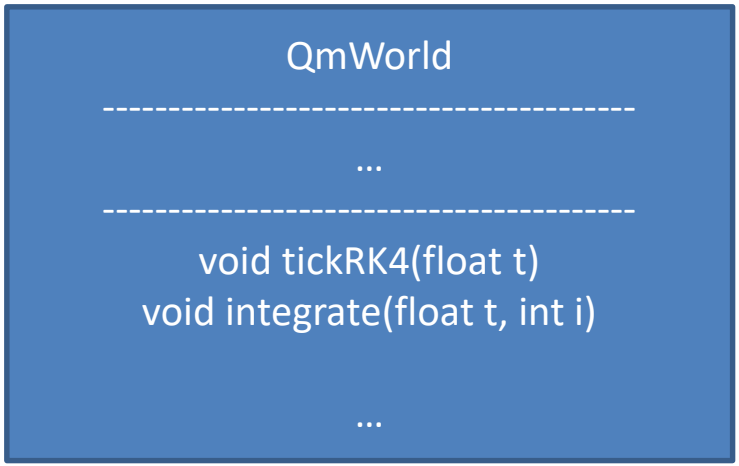


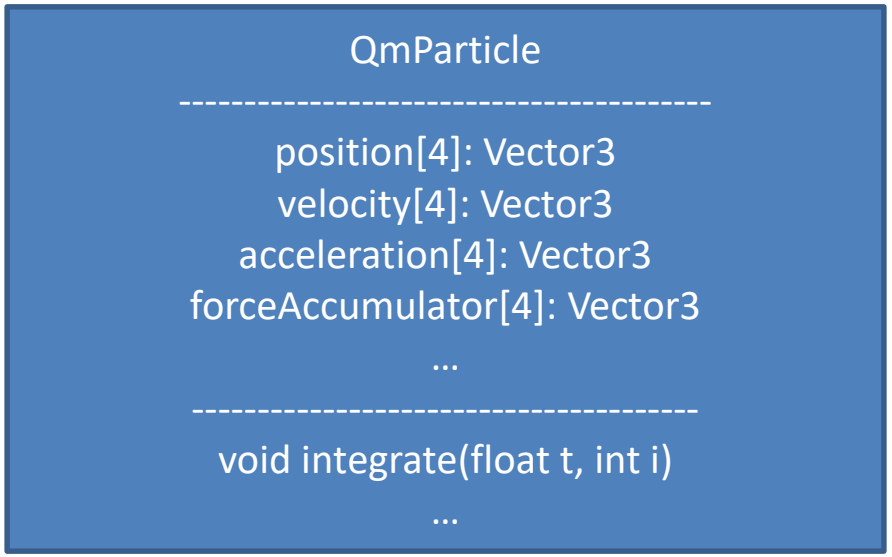
- $\boldsymbol{x}^t, \boldsymbol{v}^t$
- $\boldsymbol{a}^t(\boldsymbol{x}^t, \boldsymbol{v}^t)$ ①
- $\boldsymbol{x}^* = \boldsymbol{x}^t + \frac{h}{2}\boldsymbol{v}^t \qquad \boldsymbol{v}^* = \boldsymbol{v}^t + \frac{h}{2}\boldsymbol{a}^t$
- $\boldsymbol{a}^*(\boldsymbol{x}^*, \boldsymbol{v}^*)$ ②
- $\boldsymbol{x}^{**} = \boldsymbol{x}^t + \frac{h}{2}\boldsymbol{v}^* \qquad \boldsymbol{v}^{**} = \boldsymbol{v}^t + \frac{h}{2}\boldsymbol{a}^*$
- $\boldsymbol{a}^{**}(\boldsymbol{x}^{**}, \boldsymbol{v}^{**})$ ③
- $\boldsymbol{x}^{***} = \boldsymbol{x}^t + h\boldsymbol{v}^{**} \qquad \boldsymbol{v}^{***} = \boldsymbol{v}^t + h\boldsymbol{a}^{**}$
- $\boldsymbol{a}^{***}(\boldsymbol{x}^{***}, \boldsymbol{v}^{***})$ ④
- $\boldsymbol{x}^{t+h} = \boldsymbol{x}^t + h\frac{\boldsymbol{v}^t + 2\boldsymbol{v}^* + 2\boldsymbol{v}^{**} + \boldsymbol{v}^{***}}{6}$
- $\boldsymbol{v}^{t+h} = \boldsymbol{v}^t + h\frac{\boldsymbol{a}^t + 2\boldsymbol{a}^* + 2\boldsymbol{a}^{**} + \boldsymbol{a}^{***}}{6}$

The local discretization error of RK4 is $O(h^5)$, and the global error $O(h^4)$.

The trade-off between the number of slope evaluations and the obtained accuracy makes the Runge-Kutta of order 4 the most popular numerical integrator.

RK4: Howto






```
void tickRK4 (float t)
{
  computeAccelerations(0); // compute acc[0] considering pos[0]/vel[0]
  integrate(t/2,1); // forward-Euler integrate t/2 using vel[0]/acc[0] => pos[1]/vel[1]
  computeAccelerations(1); // compute acc[1] considering pos[1]/vel[1]
  integrate(t/2,2); // forward-Euler integrate t/2 using vel[1]/acc[1] => pos[2]/vel[2]
  computeAccelerations(2); // compute acc[2] considering pos[2]/vel[2]
  integrate(t,3);   // forward-Euler integrate t   using vel[2]/acc[2] => pos[3]/vel[3]
  computeAccelerations(3); // compute acc[3] considering pos[3]/vel[3]
  integrateRK4(t);  // forward-Euler integrate t using weighted average of vel[0..3]/acc[0..3]
  // => pos[0]/vel[0]
}
```