Particle system dynamics

Jiří Chmelík, Marek Trtík

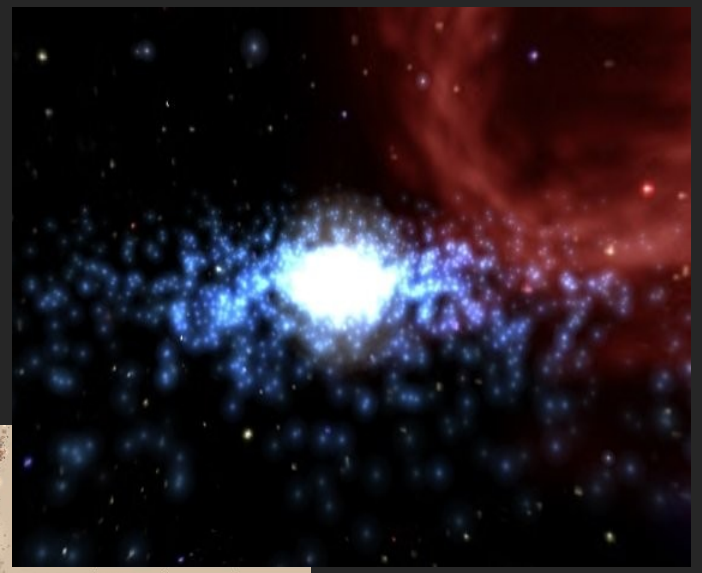
PA199

Outline

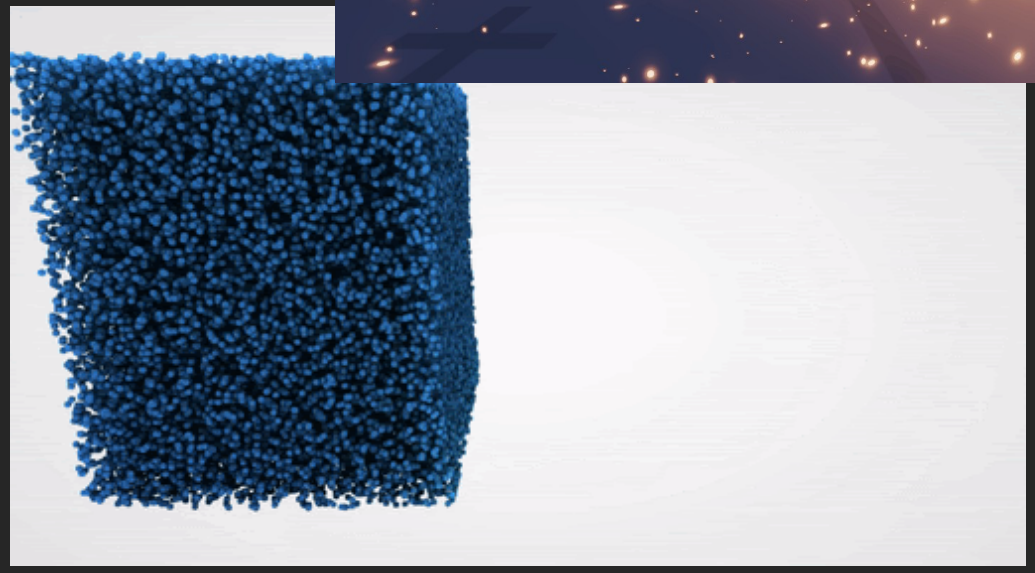
- Motivation
- Motion of a single particle: Equations of motion
  - Use of an ODE solver
- Motion of many particles
- Forces
  - Gravity, drag, spring, local interaction
- Collision: particle vs. plane
  - Detection, response, simple friction

Motivation

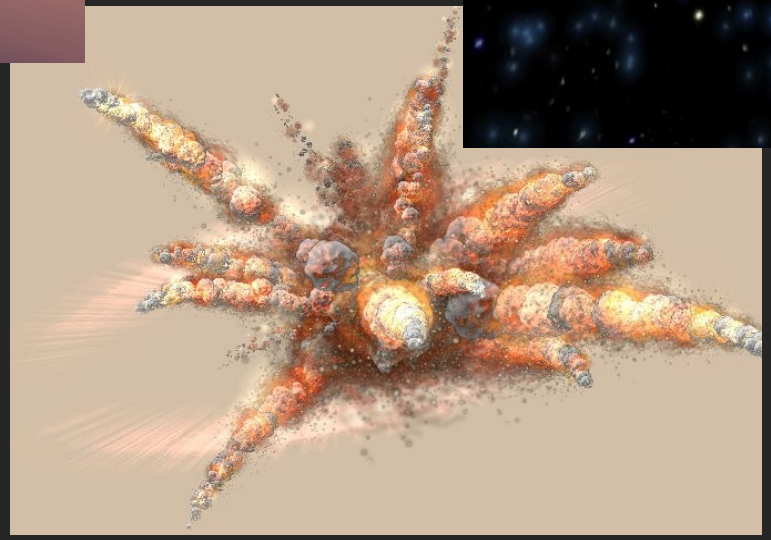
https://github.com/Lakshitha Maadushan/Unity-Particle-System



https://en.wikipedia.org/wiki/Particle_system



https://experiments.withgoogle.com/fluid-particles



Particle definition

- Particle = an abstract object with these properties:
  - No spatial extent - it is just a point in 3D space
  - Velocity
  - Respond to forces (e.g., gravity)
  - Mass - resistance to changes in motion state
- Particle in math: $P = (x, v, F, m)$.
- Particle in C++:

```
struct Particle {
        Vector3 position;
        Vector3 velocity;
        Vector3 force;
        float mass;
};
```

Particle equations of motion

- Motion of a particle $\mathcal{P}$ in space is given by a function of time:
  - $\mathcal{P}(t) = (\mathbf{x}, \mathbf{v}, \mathbf{F}, m)(t) = (\mathbf{x}(t), \mathbf{v}(t), \mathbf{F}, m)$
    - m is constant (not dependent on time).
    - $\mathbf{F}$ is total external force (not updated by the particle system).
- To compute $\mathcal{P}(t)$ we need to know how it changes in time.
  ⇒ We need to compute $\mathcal{P}(t) = (\mathbf{x}(t), \mathbf{v}(t))$.
  - Newton's second law of motion: $\mathbf{F} = m\mathbf{a}$
  - Important relations: $\mathbf{v} = \dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt}$, $\mathbf{a} = \dot{\mathbf{v}} = \frac{\partial \mathbf{v}}{\partial t}$.
- So, $\mathcal{P}(t)$ is a solution of Newton's equations of motion:

$$\dot{\mathbf{x}}(t) = \mathbf{v}(t), \qquad \dot{\mathbf{v}}(t) = \mathbf{a} = \frac{\mathbf{F}}{m}.$$

Solving equations of motion

- There is 6 ordinary differential equations (ODE) of the 1st order in the Newton's equations of a single particle.
  - $\mathbf{x}(t)$ and $\mathbf{v}(t)$ are 3D vector functions.
- In general, a system of n 1st order ODEs has the form:

$$\dot{y} = \mathbb{F}(y, t)$$

where $y(t) = (y_0(t), \ldots, y_{n-1}(t))^T$ and

$$\mathbb{F}(y, t) = (F_0(y_0(t), \ldots, y_{n-1}(t), t), \ldots, F_{n-1}(y_0(t), \ldots, y_{n-1}(t), t))^T.$$

Therefore, we have a system:

$$\dot{y}_0 = F_0(y_0, \ldots, y_{n-1}, t), \quad \ldots, \quad \dot{y}_{n-1} = F_{n-1}(y_0, \ldots, y_{n-1}, t)$$

- At each simulation time t_0 we know $\mathbf{x}(t_0) = \mathbf{x}_0$ and $\mathbf{v}(t_0) = \mathbf{v}_0$.
- Therefore, we solve the initial value problem of 1st order ODEs:

$$\dot{y} = \mathbb{F}(y, t), \quad y(t_0) = y_0$$

Solving equations of motion

- We are given a black-box function ODE solving the initial value problem of 1st order ODEs: $\dot{y} = f(y, t)$, $y(t_0) = y_0$:

```
using F_y_t = std::function<float(float, std::vector<float>&, float const, float const)>;

void ODE(
        float const
        float const
    float
    float const
        float
    );
```

NOTE: Implementation of ODE is the topic of next lecture.

Building initial state for ODE

```
void getState(Particle const& p, std::vector<float>& y0) {
        y0.push_back(p.position.x);
        y0.push_back(p.position.y);
        y0.push_back(p.position.z);

        y0.push_back(p.velocity.x);
        y0.push_back(p.velocity.y);
        y0.push_back(p.velocity.z);
}
```

Building derivatives for ODE

```
void getDerivative(Particle const
                                    float const float return
                                    float const float return
                                    float const float return

                                    float const float return
                                    float const float return
                                    float const float return
}
```

- Observation: Parameters of lambda functions are not used.
  - Our functions $f(y, t)$ are simple; ODE solver handles general cases.

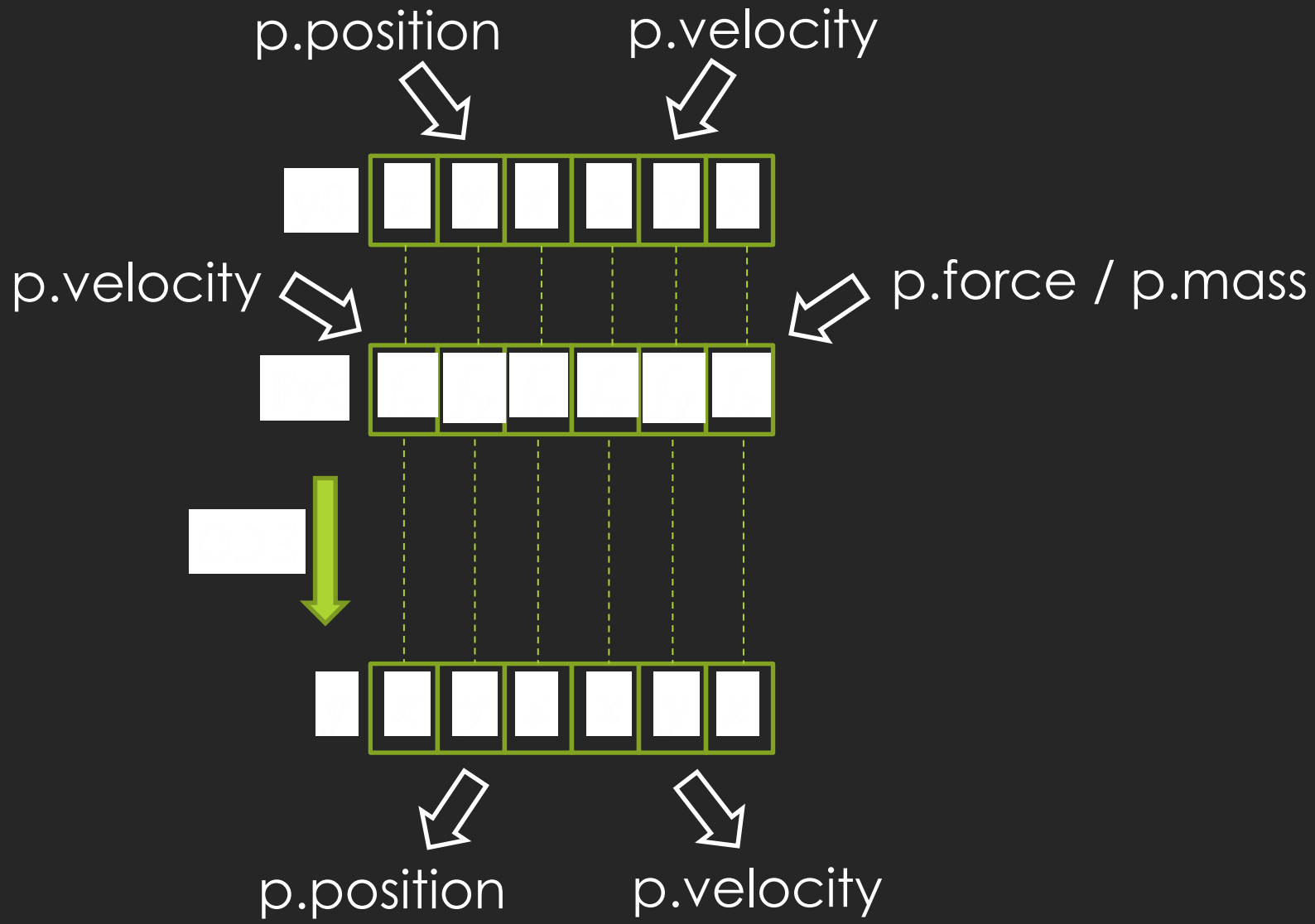
Simulation step for single particle

```
void doSimulationStep(Particle& p, float& t, float const dt) {
        UpdateForce(p,t,dt);       // Applies external forces and impulses.

        std::vector<float> y0, y;
        std::vector<F_y_t> Fyt;
        getState(p, y0);
        getDerivative(p, Fyt);
        ODE(y0, Fyt, t, dt, y);       // Computes y and updates t (t += dt).
        setState(p, y.begin());
}
```

Saving ODE results

```
void setState(Particle& p, std::vector<float>::const_iterator& it) {
        p.position.x = *it; ++it;
        p.position.y = *it; ++it;
        p.position.z = *it; ++it;

        p.velocity.x = *it; ++it;
        p.velocity.y = *it; ++it;
        p.velocity.z = *it; ++it;
}
```

Data flow in simulation step



Particle system

- It is a system consisting of n particles.
- Particle system in math:

$$p^n = [p_0, p_1, \dots, p_{n-1}] = [(x_0, v_0, F_0, m_0), (x_1, v_1, F_1, m_1), \dots, (x_{n-1}, v_{n-1}, F_{n-1}, m_{n-1})].$$

- Particle system in C++:

```
using ParticleSystem = std::vector<Particle>;
```

ODE helper functions

```
void getState(ParticleSystem const& ps, std::vector<float>& y0) {
        for (Particle const& p : ps) getState(p,y0);
}

void getDerivative(ParticleSystem const& ps, std::vector<F_y_t>& Fyt) {
        for (Particle const& p : ps) getDerivatives(p, Fyt);
}

void setState(ParticleSystem& ps, std::vector<float>::const_iterator& it) {
        for (Particle& p : ps) setState(p, it);
}
```

Simulation step for whole system

```
void doSimulationStep(ParticleSystem& ps, float& t, float const dt) {
        UpdateForce(ps,t,dt);     // Applies external forces and impulses.

        std::vector<float> y0, y;
        std::vector<F_y_t> Fyt;
        getState(ps, y0);
        getDerivative(ps, Fyt);
        ODE(y0, Fyt, t, dt, y);       // Computes y and updates t (t += dt).
        setState(ps, y.begin());
}
```

Data flow in simulation step

NOTE: for p^0 we have a system of $6n$ equations

Forces

```
void UpdateForce(ParticleSystem& ps, float const t, float const dt) {
        clearForce(ps);
        applyForce(ps,t,dt);  // Add all forces and impulses to all particles.
}

void clearForce(ParticleSystem& ps) {
        for (Particle& p : ps) p.force = Vector3(0,0,0);
}
```

- Next we discuss what forces we can add to particles inside the function applyForce().

Gravity

- Homogeneous field:
  - For each particle we add the force vector $\vec{F} = m\vec{g}$ where
    - m is the mass of the particle.
    - $\vec{g}$ is a constant vector, e.g., $\vec{g} = \text{Vector3}(0,0,-10)$.
- Radial field:
  - There is a center of gravity S of mass M (it can be one of the particles).
  - For each particle we add the force vector

$$\vec{F} = G\frac{Mm}{|S-x|^2}\frac{S-x}{|S-x|} = G\frac{Mm}{|S-x|^3}(S-x), \quad \text{where}$$

    - G is the gravitational constant.
    - m is the mass of the particle.
    - x is the position of the particle.
  - We can handle cases when $|S-x|$ is small by not applying the force.

Viscous Drag

- A force of the environment making a particle decrease its velocity relative to the environment.
- A drag force can also enhance numerical stability of simulation.
- For each particle we add the force vector $\vec{F} = k_d(\vec{v}_e - \vec{v})$, where
  - k_d is the coefficient of drag.
  - $\vec{v}_e$ is the velocity of the environment (often $\vec{v}_e = \vec{0}$).
  - $\vec{v}$ is the velocity of the particle.

Spring

- It is a force between two particles P_i and P_j given by Hook's law:

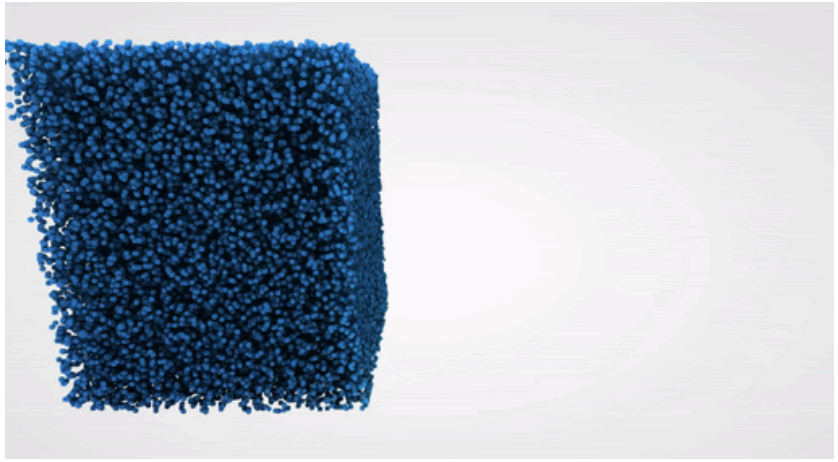
$$\mathbb{F}_i = -\left(k_s(|\vec{d}| - d_0) + k_d \vec{d} \cdot \frac{\vec{d}}{|\vec{d}|}\right)\frac{\vec{d}}{|\vec{d}|}$$

$$\mathbb{F}_j = -\mathbb{F}_i \quad (3^{rd} \text{ Newton's law – action and reaction})$$

  where
  - k_s is the spring constant.
  - k_d is the damping constant.
  - $\vec{d} = x_i - x_j$ is the distance vector between the particles.
  - d_0 is the rest length between the particles.
  - $\dot{\vec{d}} = v_i - v_j$ is the relative velocity between the particles.

Local interaction

- Particles start to interact when they come close.
- Particles stop to interact when they move apart.
- Example: Particle-based fluid simulation.
- Computationally expensive task:
  - $O(n^2)$ – all pairs of particles are checked.
  - Space partitioning methods (e.g., octree) are essential for performance.



Collision: particle vs. plane

- We often want particles to collide with the ground or a wall. These boundaries can be approximated by planes.

https://github.com/LakshithaMadushan/Unity-Particle-System

- The process consists of two parts:
  - Detection of a collision.
  - Response to the collision.

Collision detection

- Let us consider a particle $\mathcal{P} = (x, v, F, m)$.
- The plane is represented by the equation $N \cdot (X - P) = 0$, where
  - N is the unit normal vector pointing "outside" (above the ground).
  - P is some point in the plane.
  - X is a tested point.
- The particle collides with the plane only if $N \cdot (x - P) \leq 0$.
  - Only in that case we proceed to the collision response.

Collision response

- If the particle increases the penetration with the plane, i.e., when $\mathbf{N} \cdot \mathbf{v} < 0$, then we change the component of $\mathbf{v}$ orthogonal to the plane:
  - The component of $\mathbf{v}$ orthogonal to the plane is $\mathbf{v}^{\perp} = (\mathbf{N} \cdot \mathbf{v})\mathbf{N}$.
  - The velocity change is then is $\Delta\mathbf{v} = -(1+r)\mathbf{v}^{\perp} = -(1+r)(\mathbf{N} \cdot \mathbf{v})\mathbf{N}$, where
    - $r \in (0,1)$ is the coefficient of restitution.
  - We update $\mathbf{v}$ to be $\mathbf{v} + \Delta\mathbf{v}$.
    - NOTE: Formally, we apply an impulse $\mathbf{I} = m\Delta\mathbf{v}$ to the particle.
- If $\mathbf{N} \cdot \mathbf{F} < 0$, then we cancel the component of $\mathbf{F}$ orthogonal to the plane:
  - We compute $\Delta\mathbf{F} = -\mathbf{F}^{\perp}$, where $\mathbf{F}^{\perp} = (\mathbf{N} \cdot \mathbf{F})\mathbf{N}$.
  - We update $\mathbf{F}$ to be $\mathbf{F} + \Delta\mathbf{F}$.
  - NOTE: This step should be applied after all external forces (gravity, etc.) were added to the $\mathbf{F}$ field of the particle.

Simple friction

- We build a simplified friction model for particle system:
  - We do not distinguish static and dynamic friction.
  - We ignore variable changes caused by interactions with other particles.
- If $N \cdot F < 0$, then a friction force F_f is acting on the particle.
  - $|F_f|$ is proportional to $|F^{\perp}|$, where $F^{\perp} = (N \cdot F)N$.
  - The direction of F_f is opposite to the component $v^{\perp}$ of v parallel with the plane, where $v^{\perp} = \frac{N \times v \times N}{|N \times v \times N|}$.
- Therefore, we define the friction force as $F_f = k_f(N \cdot F)v^{\perp}$, where
  - k_f is a friction coefficient.
- Note: We should apply the friction before the collision response.

Summary

- We defined particle and particle system.
- We learned Newton's equations of motion for a particle, i.e., a system of 1st order ODEs.
- We learned how to use ODE solver for the simulation.
- We learned several kinds of forces which we can apply to particles.
- We know how to compute and respond to collision of a particle with a plane, including application of a friction force.

References

- [1] *Andrew Witkin*; Physically Based Modeling: Principles and Practice Particle System Dynamics; Robotics Institute, Carnegie Mellon University, 1997.