Fundamentals of Computer Graphics

Lecture 8
Particle Systems

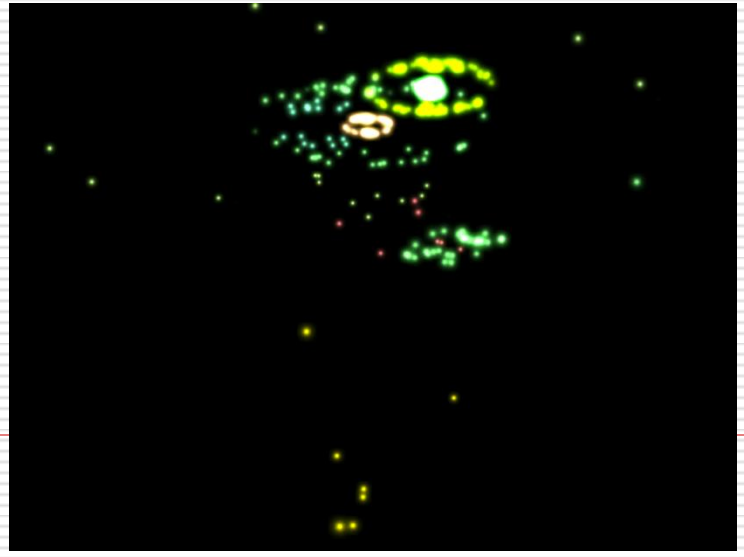
Yong-Jin Liu
liuyongjin@tsinghua.edu.cn

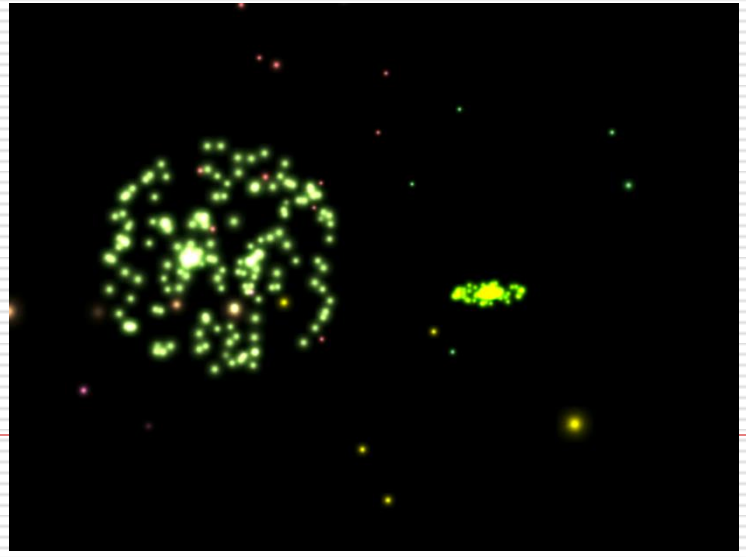
Candidate course projects

5. Chinese firework

- Key points:

  Physical based modeling;
  Particle system





Candidate course projects

6. Simple physics engine

- Key points:

  Physical-based modeling;
  Collision detection;
  Physical law simulation

Candidate course projects

7. Game design

- Key points:

  Design “Asteroids” game with game engine Irrlicht

  Using everything you know

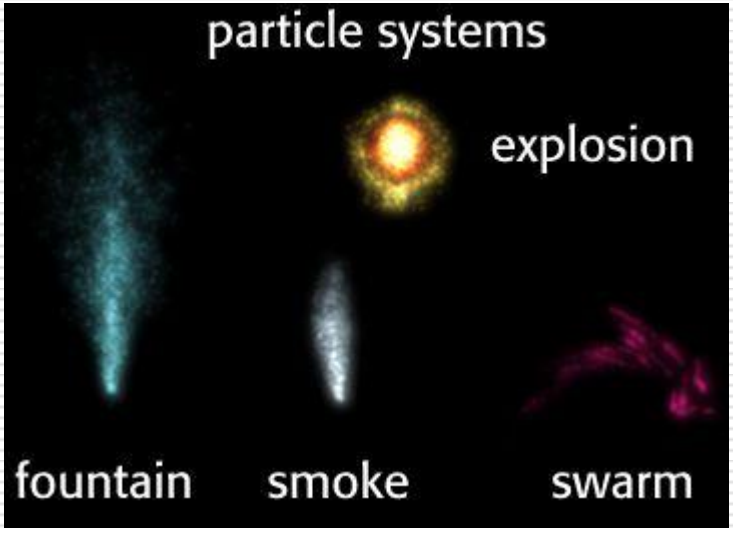
  http://irrlicht.sourceforge.net

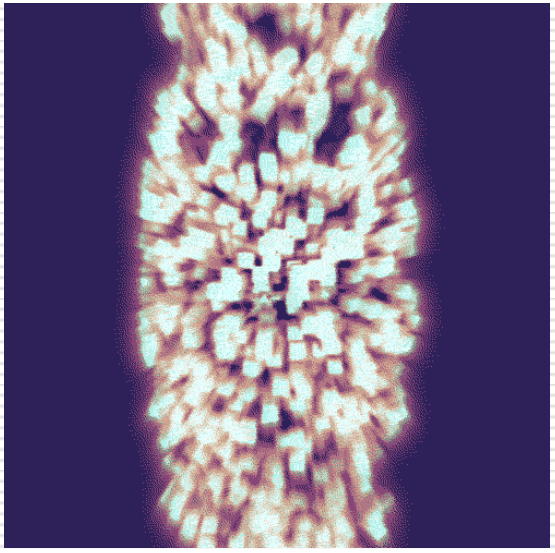
Overview

- Particle systems refers to a computer graphics technique that uses a large number of very **small sprites** or other graphic objects to simulate certain kinds of "fuzzy" phenomena, which are otherwise very hard to reproduce with conventional rendering techniques - usually highly chaotic systems, natural phenomena, and/or processes caused by chemical reactions.

Particle Systems

- Particle systems offer a solution to modeling amorphous, dynamic and fluid objects like clouds, smoke, water, explosions and fire.



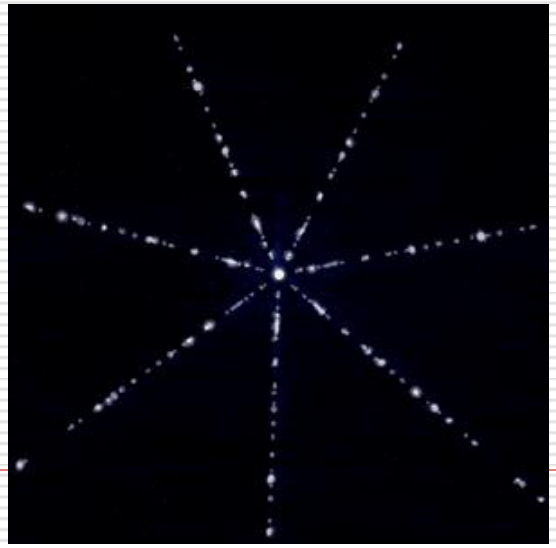


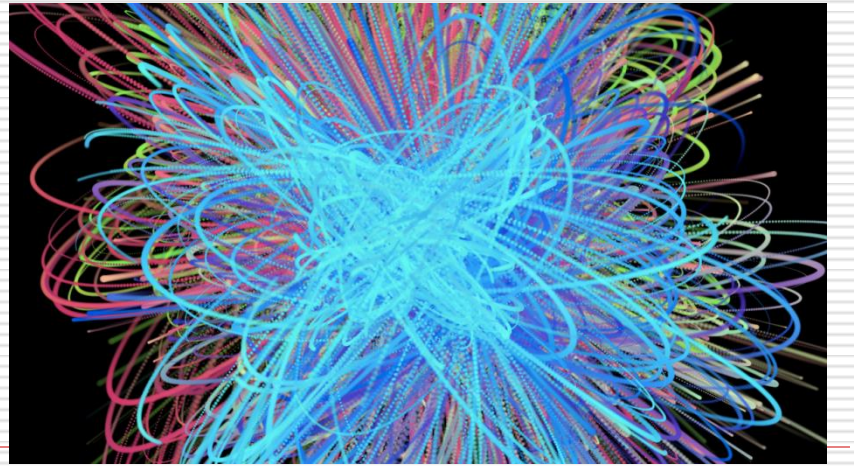
Particle System (PS) Gallery

Particle System (PS) Gallery

Particle Systems

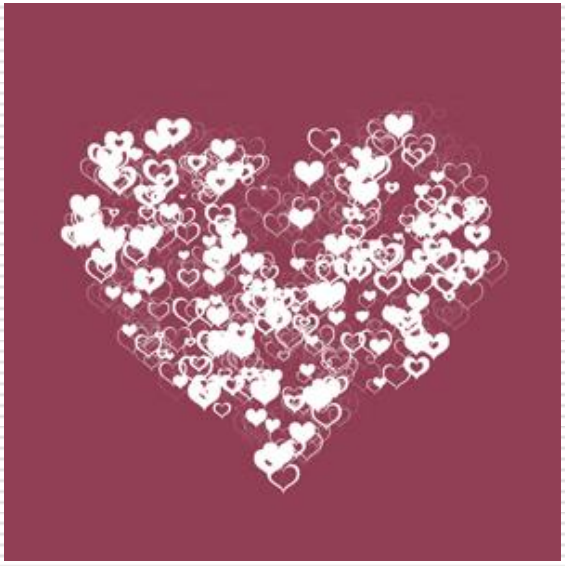
- Particle systems have been used extensively in computer animation and special effects since their introduction to the industry in the early 1980's





Particle Systems

- The rules governing the behavior of an individual particle can be relatively simple, and the complexity comes from having lots of particles
- Usually, particles will follow some combination of physical and non-physical rules, depending on the exact situation

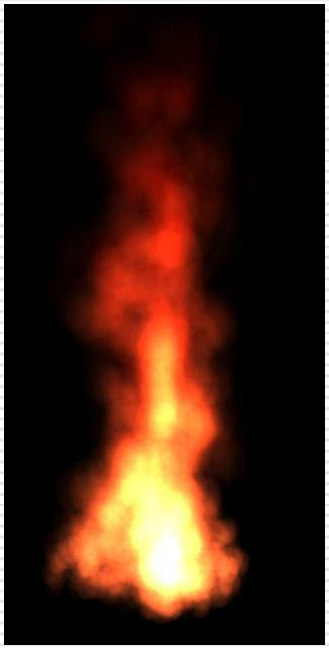


Particles

- Can either be 2D or 3D
- Geometry of a particle can be a plane or a cube
- Parameters of a particle: position, velocity, acceleration, texture, possibility for randomness, etc.

Representing Objects with Particles

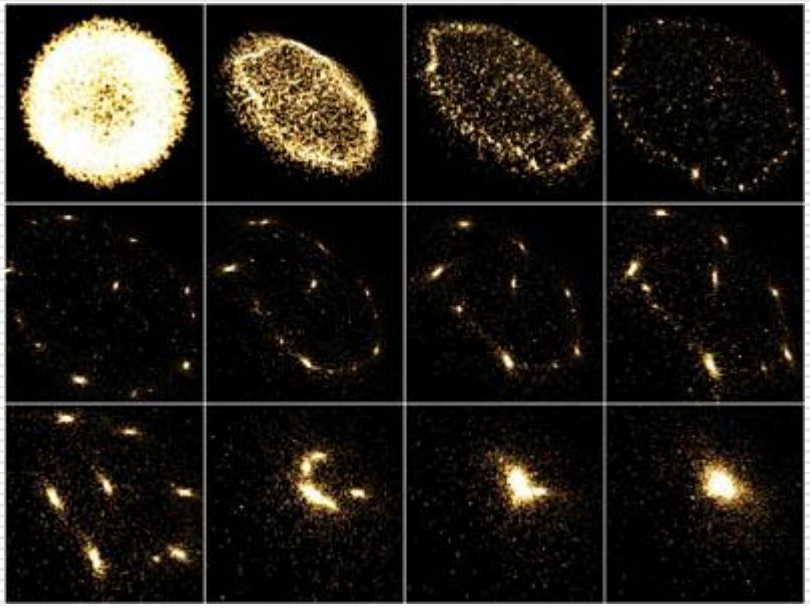
- An object is represented as clouds of primitive particles that define its volume rather than by polygons or patches that define its boundary.
- A particle system is dynamic, particles changing form and moving with the passage of time.
- Object is not deterministic, its shape and form are not completely specified. Instead



System framework

Modules in framework

- Physics
- Forces
- Integration
- Particle Systems
- Particle Rendering



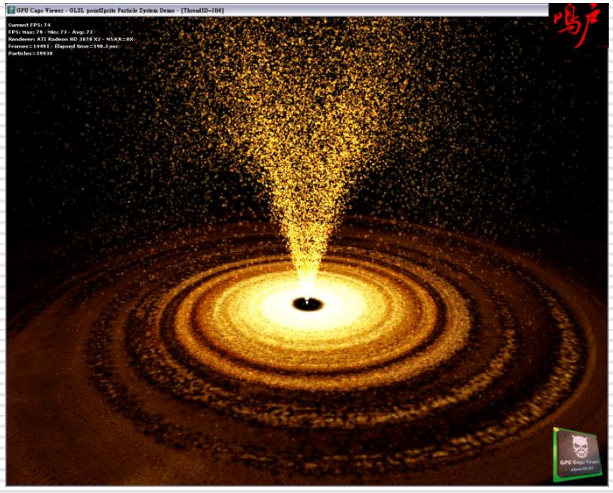
Kinematics of Particles

- We will define an individual particle's 3D position over time as $\mathbf{r}(t)$
- By definition, the velocity is the first derivative of position, and acceleration is the second

$$\mathbf{r} = \mathbf{r}(t)$$

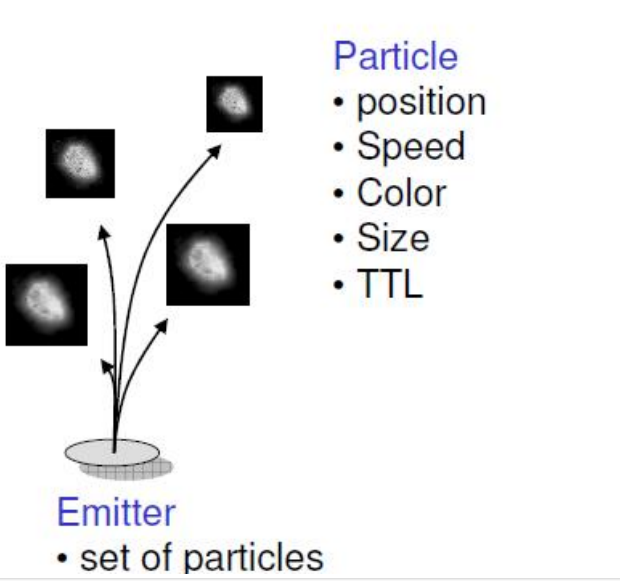
$$\mathbf{v} = \frac{d\mathbf{r}}{dt}$$

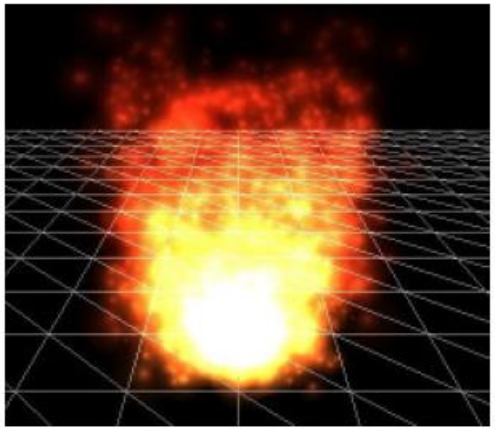
$$\mathbf{a} = \frac{d\mathbf{v}}{dt} = \frac{d^2\mathbf{r}}{dt^2}$$

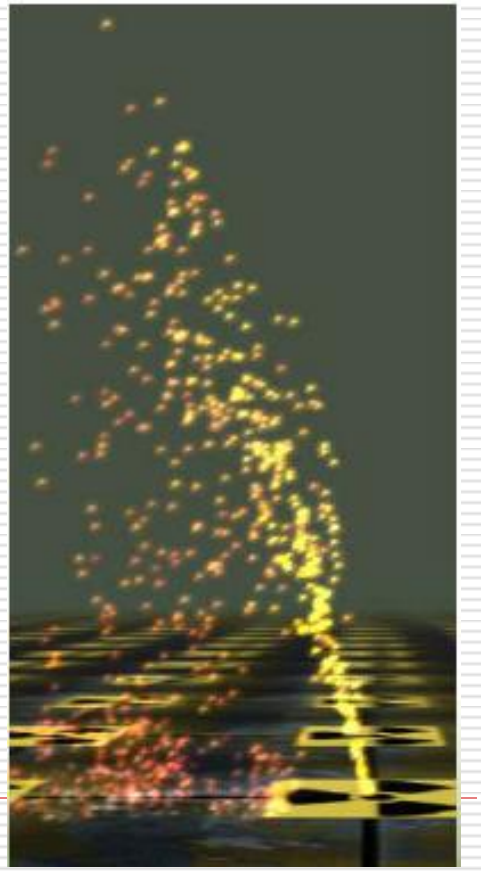


Kinematics of Particles

- To render a particle, we need to know it's position **r**.







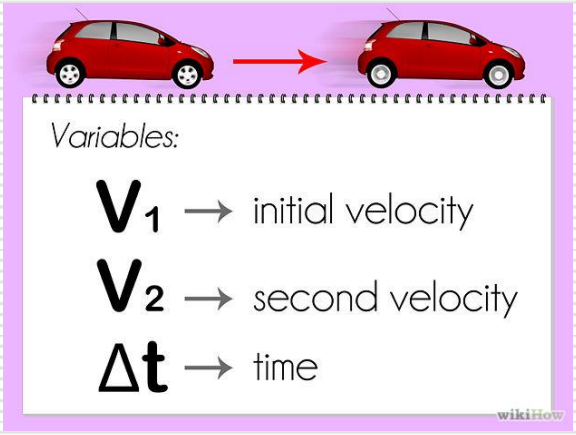
Uniform Acceleration

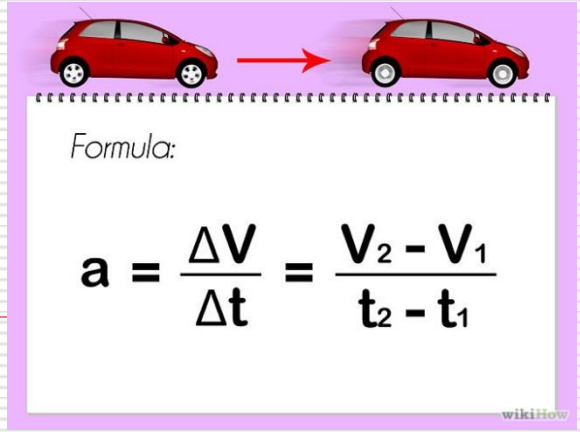
- How does a particle move when subjected to a constant acceleration?

$$\mathbf{a} = \mathbf{a}_0$$

$$\mathbf{v} = \int \mathbf{a}\,dt = \mathbf{v}_0 + \mathbf{a}_0 t$$

$$\mathbf{r} = \int \mathbf{v}\,dt = \mathbf{r}_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}_0 t^2$$





Uniform Acceleration

$$\mathbf{r} = \mathbf{r}_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}_0 t^2$$

- This shows us that the particle's motion will follow a parabola
- Keep in mind, that this is a 3D vector equation, and that there is potentially a parabolic equation in each dimension. Together, they form a 2D parabola oriented in 3D space
- We also see that we need two additional vectors $\mathbf{r}_0$ and $\mathbf{v}_0$ in order to fully specify the equation. These represent the initial position and velocity at time t=0

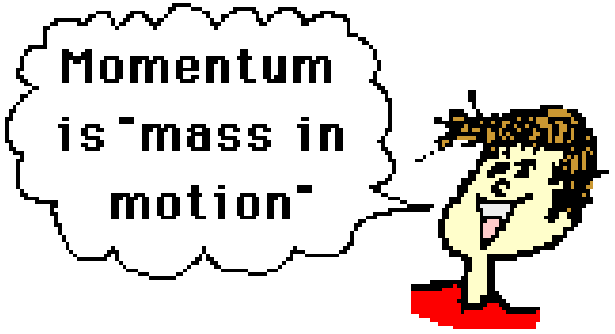
Mass and Momentum

- We can associate a mass m with each particle. We will assume that the mass is constant

$$m = m_0$$

- We will also define a vector quantity called momentum (**p**), which is the product of mass and velocity

$$\mathbf{p} = m\mathbf{v}$$



Newton's First Law

- Newton's First Law states that a body in motion will remain in motion and a body at rest will remain at rest- unless acted upon by some force
- This implies that a free particle moving out in space will just travel in a straight line

$$\mathbf{a} = 0$$

$$\mathbf{v} = \mathbf{v}_0$$

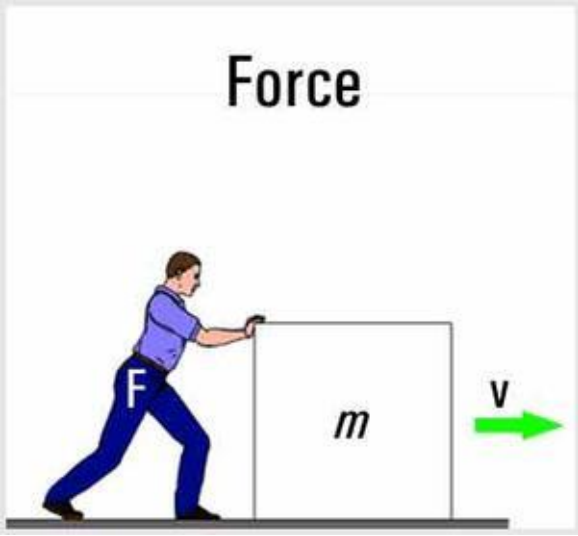
$$\mathbf{r} = \mathbf{r}_0 + \mathbf{v}_0 t$$

$$\mathbf{p} = \mathbf{p}_0 = m\mathbf{v}_0$$

Force

- Force is defined as the rate of change of momentum

$$\mathbf{f} = \frac{d\mathbf{p}}{dt}$$



- We can expand this out:

$$\mathbf{f} = \frac{d(mv)}{dt} = \frac{dm}{dt}v + m\frac{dv}{dt} = m\frac{dv}{dt}$$

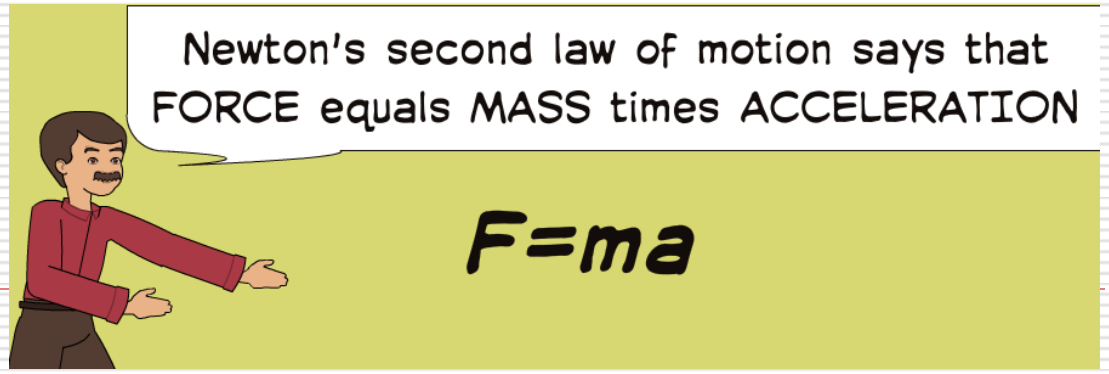
$$\mathbf{f} = ma$$

Newton's Second Law

- Newton's Second Law says:

$$f = \frac{dp}{dt} = ma$$

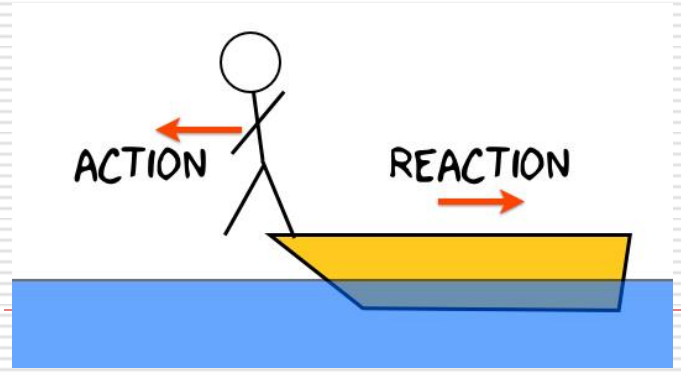
- This relates the kinematic quantity of acceleration to the physical quantity of force

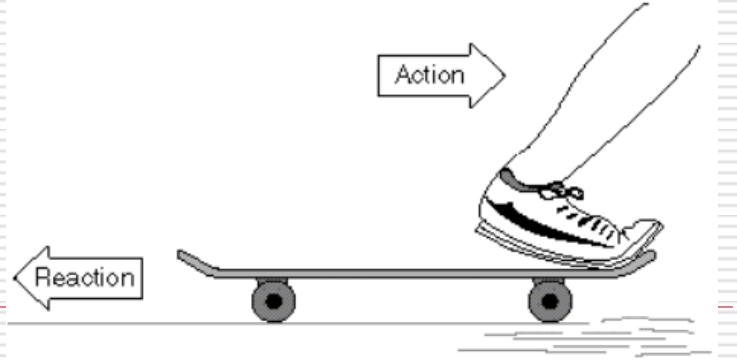


Newton's Third Law

- Newton's Third Law says that any force that body A applies to body B will be met by an equal and opposite force from B to A

$$\mathbf{f}_{AB} = -\mathbf{f}_{BA}$$





Criteria

- Conservation of Momentum
- Energy
- Forces on a Particle

$$\mathbf{f}_{total} = \sum \mathbf{f}_i$$

Particle Simulation

- Basic kinematics
- Newton's laws
- A general scheme for simulating particles

Particle Simulation

- Compute all forces acting within the system in the current configuration (making sure to obey Newton's third law)
- Compute the resulting acceleration for each particle ($a=f/m$) and integrate over some small time step to get new positions

- Repeat

- This describes the standard 'Newtonian' approach to simulation. It can be extended to rigid bodies, deformable bodies, fluids, vehicles, and more

Particle Example

```
class Particle {
        float Mass;             // Constant
        Vector3 Position;       // Evolves frame to frame
        Vector3 Velocity;       // Evolves frame to frame
        Vector3 Force;          // Reset and re-computed each frame
public:
        void Update(float deltaTime);
        void Draw();
        void ApplyForce(Vector3 &f)       {Force.Add(f);}
};
```

Particle Example

```
class ParticleSystem {
        int NumParticles;
        Particle *P;
public:
        void Update(deltaTime);
        void Draw();
};
```

Particle Example

```
ParticleSystem::Update(float deltaTime) {
        // Compute forces
        Vector3 gravity(0,-9.8,0);
        for(i=0;i<NumParticles;i++) {
                Vector3 force=gravity*Particle[i].Mass;   // f=mg
                Particle[i].ApplyForce(force);
        }

        // Integrate
        for(i=0;i<NumParticles;i++)
                Particle[i].Update(deltaTime);
}
```

Particle Example

```
Particle::Update(float deltaTime) {
        // Compute acceleration (Newton's second law)
        Vector3 Accel=(1.0/Mass) * Force;

        // Compute new position & velocity
        Velocity+=Accel*deltaTime;
        Position+=Velocity*deltaTime;

        // Zero out Force vector
        Force.Zero();
}
```

Particle Example

- With this particle system, each particle keeps track of the total force being applied to it
- This value can accumulate from various sources, both internal and external to the particle system
- The example just used a simple gravity force, but it could easily be extended to have all kinds of other possible forces
- The integration scheme used is called ‘forward Euler integration’ and is about the simplest method possible

System framework

Modules in framework

- Physics
- Forces
- Integration
- Particle Systems
  (Particle Rendering)

Uniform Gravity

- A very simple, useful force is the uniform gravity field:

$$\mathbf{f}_{gravity} = m\mathbf{g}_0$$

$$\mathbf{g}_0 = \begin{bmatrix} 0 & -9.8 & 0 \end{bmatrix} \quad \frac{m}{s^2}$$

Newton's Law of Gravitation

- If we are far away enough from the objects such that the inverse square law of gravity is noticeable, we can use Newton's Law of Gravitation:

$$\mathbf{f}_{gravity} = \frac{Gm_1m_2}{d^2}\mathbf{e} \qquad \mathbf{e} = \frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|}$$

$$G = 6.673 \times 10^{-11} \qquad \frac{m^3}{kg \cdot s^2}$$

Gravity

- The equation describes the gravitational force between two particles
- To compute the forces in a large system of particles, every pair must be considered
- This gives us an N^2 loop over the particles
- Actually, there are some tricks to speed this up, but we won't look at those

Aerodynamic Drag

- Aerodynamic interactions are actually very complex and difficult to model accurately
- A reasonable simplification it to describe the total aerodynamic drag force on an object using:

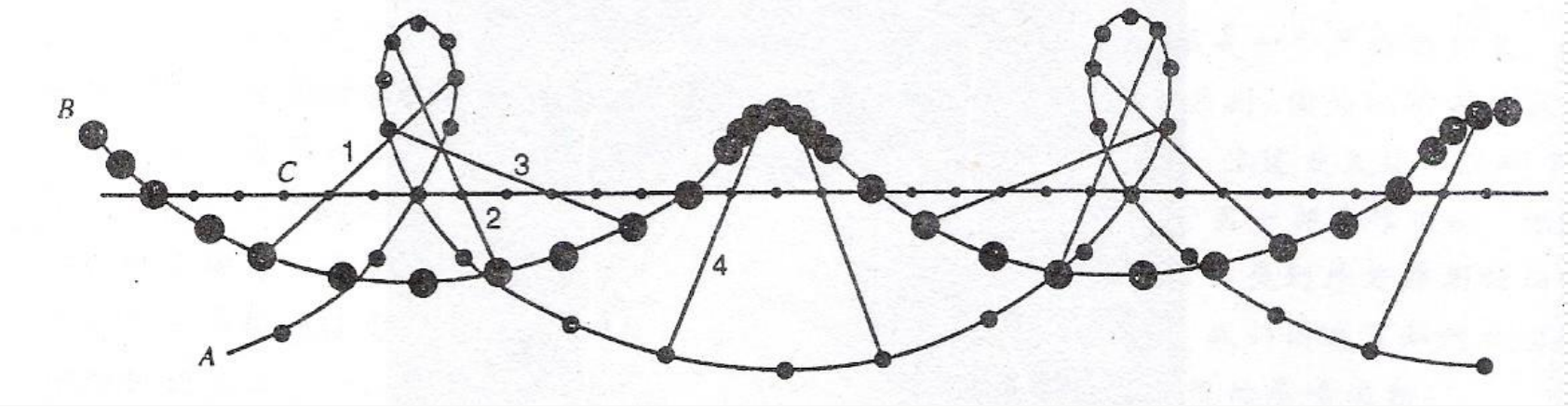
$$\mathbf{f}_{aero} = \frac{1}{2}\rho|\mathbf{v}|^2 c_d a\mathbf{e} \qquad \mathbf{e} = -\frac{\mathbf{v}}{|\mathbf{v}|}$$

- Where ρ is the density of the air (or water…), c_d is the coefficient of drag for the object, a is the cross sectional area of the object, and **e** is a unit vector in the opposite direction of the velocity

Aerodynamic Drag

- Like gravity, the aerodynamic drag force appears to violate Newton's Third Law, as we are applying a force to a particle but no equal and opposite force to anything else
- We can justify this by saying that the particle is actually applying a force onto the surrounding air, but we will assume that the resulting motion is just damped out by the viscosity of the air

We give the rod a push along the frictionless horizontal surface and examine its motion.



Snapshots of the locations of points A and B at successive intervals of time.

Springs

- A simple spring force can be described as:

$$\mathbf{f}_{spring} = -k_s \mathbf{x}$$

- Where k is a ‘spring constant’ describing the stiffness of the spring and x is a vector describing the displacement

Dampers

- We can also use damping forces between particles:

$$\mathbf{f}_{damp} = -k_d \mathbf{v}$$

- Dampers will oppose any difference in velocity between particles
- The damping forces are equal and opposite, so they conserve momentum, but they will remove energy from the system
- In real dampers, kinetic energy of motion is converted into complex fluid motion within the damper and then diffused into random molecular motion causing an increase in temperature. The kinetic energy is effectively lost

Dampers

- Dampers operate in very much the same way as springs, and in fact, they are usually combined into a single spring-damper object
- A simple spring-damper might look like:

```
class SpringDamper {
        float SpringConstant, DampingFactor;
        float RestLength;
        Particle *P1,*P2;
public:
        void ComputeForce();
};
```

Force Fields

- We can also define any arbitrary force field that we want. For example, we might choose a force field where the force is some function of the position within the field

$$\mathbf{f}_{field} \propto \mathbf{f}(\mathbf{r})$$

- We can also do things like defining the velocity of the air by some similar field equation and then using the aerodynamic drag force to compute a final force
- Using this approach, one can define useful turbulence fields, vortices, and other flow patterns

Collisions & Impulse

- A collision is assumed to be instantaneous
- However, for a force to change an object's momentum, it must operate over some time interval
- Therefore, we can't use actual forces to do collisions
- Instead, we introduce the concept of an impulse, which can be though of as a large force acting over a small time

Impulse

- An impulse can be thought of as the integral of a force over some time range, which results in a finite change in momentum:

$$\mathbf{j} = \int \mathbf{f}\, dt = \Delta \mathbf{p}$$

- An impulse behaves a lot like a force, except instead of affecting an object's acceleration, it directly affects the velocity
- Impulses also obey Newton's Third Law, and so objects can exchange equal and opposite impulses
- Also, like forces, we can compute a total impulse as the sum of several individual impulses

Impulse

- The addition of impulses makes a slight modification to our particle simulation:

// Compute all forces and impulses

$$\mathbf{f} = \sum \mathbf{f}_i$$

$$\mathbf{j} = \sum \mathbf{j}_i$$

// Integrate to get new velocity & position

$$\mathbf{v}' = \mathbf{v}_0 + \frac{1}{m}\left(\mathbf{f}\Delta t + \mathbf{j}\right)$$

$$\mathbf{r}' = \mathbf{r}_0 + \mathbf{v}'\Delta t$$

Collisions

- Today, we will just consider the simple case of a particle colliding with a static object
- The particle has a velocity of $\mathbf{v}$ before the collision and collides with the surface with a unit normal $\mathbf{n}$
- We want to find the collision impulse $\mathbf{j}$ applied to the particle during the collision

Elasticity

- There are a lot of physical theories behind collisions
- We will stick to some simplifications
- We will define a quantity called elasticity that will range from 0 to 1, that describes the energy restored in the collision
- An elasticity of 0 indicates that the closing velocity after the collision is 0
- An elasticity of 1 indicates that the closing velocity after the collision is the exact opposite of the closing velocity before the collision

Combining Forces

- All of the forces we've examined can be combined by simply adding their contributions
- Remember that the total force on a particle is just the sum of all of the individual forces
- Each frame, we compute all of the forces in the system at the current instant, based on instantaneous information (or numerical approximations if necessary)
- We then integrate things forward by some finite time steps

System framework

Modules in framework

- Physics
- Forces
- Integration
- Particle Systems
  (Particle Rendering)

Integration

- Computing positions and velocities from accelerations is just integration
- If the accelerations are defined by very simple equations (like the uniform acceleration we looked at earlier), then we can compute an analytical integral and evaluate the exact position at any value of t
- In practice, the forces will be complex and impossible to integrate analytically, which is why we automatically resort to a numerical scheme in practice
- The Particle::Update() function described earlier computes one iteration of the numerical integration. In particular, it uses the ‘forward Euler’ scheme

Forward Euler Integration

- Forward Euler integration is about the simplest possible way to do numerical integration

$$x_{n+1} = x_n + x'_n \Delta t$$

- It works by treating the linear slope of the derivative at a particular value as an approximation to the function at some nearby value
- The gradient descent algorithm we used for inverse kinematics used Euler integration

Forward Euler Integration

- For particles, we are actually integrating twice to get the position

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_n \Delta t$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_{n+1} \Delta t$$

which expands to

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \left(\mathbf{v}_n + \mathbf{a}_n \Delta t\right) \Delta t$$

$$= \mathbf{r}_n + \mathbf{v}_n \Delta t + \mathbf{a}_n \left(\Delta t\right)^2$$

Forward Euler Integration

- Note that this:

is very similar to the result we would get if we just assumed that the particle is under a uniform acceleration for the duration of one frame:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_n \Delta t + \frac{1}{2}\mathbf{a}_n (\Delta t)^2$$

Forward Euler Integration

- Actually, it will work either way
- Both methods make assumptions about what happens in the finite time step between two instants, and both are just numerical approximations to reality
- As Δt approaches 0, the two methods become equivalent
- At finite Δt, however, they may have significant differences in their behavior, particularly in terms of accuracy over time and energy conservation
- As a rule, the forward Euler method works better
- In fact, there are lots of other ways we could approximate the integration to improve accuracy, stability, and efficiency

Forward Euler Integration

- The forward Euler method is very simple to implement and if it provides adequate results, then it can be very useful
- It will be good enough for lots of particle systems used in computer animation, but it's accuracy is not really good enough for 'engineering' applications
- It may also behave very poorly in situations where forces change rapidly, as the linear approximation to the acceleration is no longer valid in those circumstances

Forward Euler Integration

- One area where the forward Euler method fails is when one has very tight springs
- A small motion will result in a large force
- Attempting to integrate this using large time steps may result in the system diverging (or ‘blowing up’)
- Therefore, we must use lots of smaller time steps in order for our linear approximation to be accurate enough
- This resorting to many small time steps is where the computationally simple Euler integration can actually be slower than a more complex integration scheme that costs more per iteration but requires fewer iterations
- We will look at more sophisticated integration schemes in future lectures

System framework

Modules in framework

- Physics
- Forces
- Integration
- Particle Systems
  (Particle Rendering)

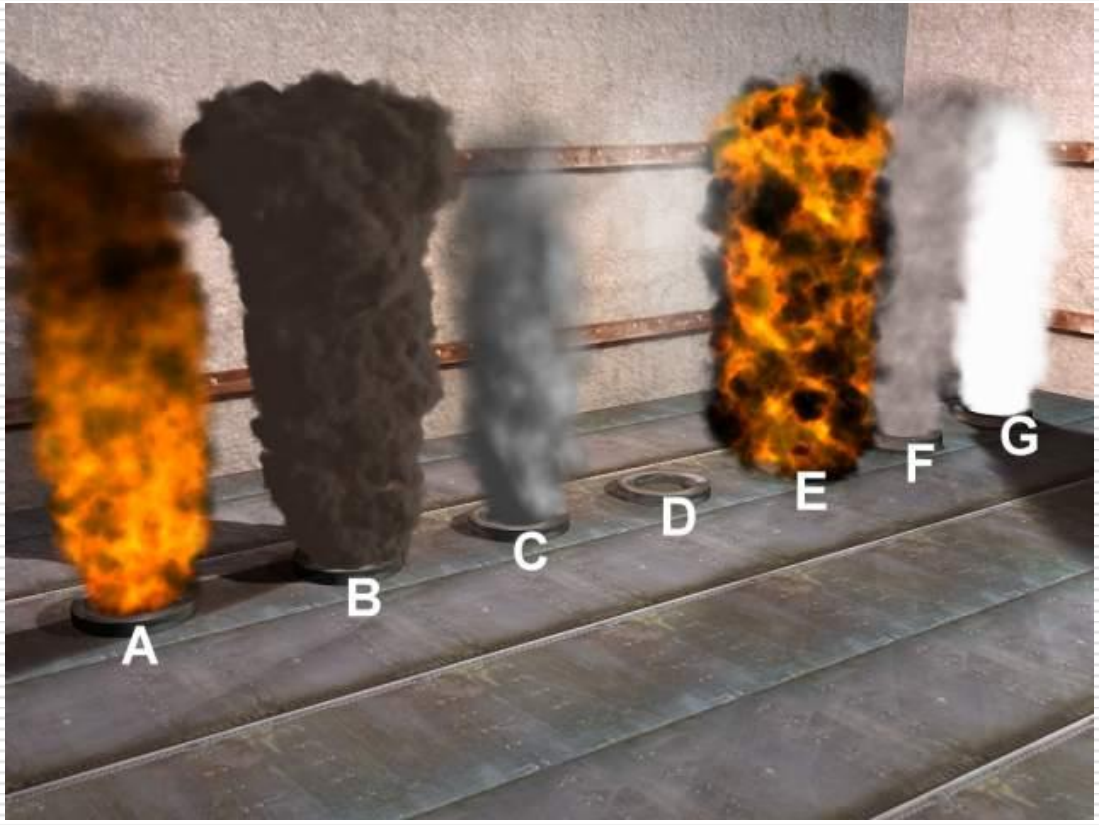
Particle Systems

- In computer animation, particle systems can be used for a wide variety of purposes, and so the rules governing their behavior may vary
- A good understanding of physics is a great place to start, but we shouldn't always limit ourselves to following them strictly
- In addition to the physics of particle motion, several other issues should be considered when one uses particle systems in computer animation

Particles

- In physics, a basic particle is defined by it’s position, velocity, and mass
- In computer animation, we may want to add various other properties:
  - Color
  - Size
  - Life span
  - Anything else we want…

A particle's position is found by simply adding its velocity vector to its position vector. This can be modified by forces such as gravity.



Other attributes can vary over time as well, such as color, transparency and size. These rates of change can be global or they can be stochastic for each particle.

Particle Extinction --- life span

- When generated, given a lifetime in frames.
- Lifetime decremented each frame, particle is killed when it reaches zero.
- Kill particles that no longer contribute to image (transparency below a certain threshold, etc.).

Creation & Destruction

- The example system we showed at the beginning had a fixed number of particles
- In practice, we want to be able to create and destroy particles on the fly
- Often times, we have a particle system that generates new particles at some rate
- The new particles are given initial properties according to some creation rule
- Particles then exist for a finite length of time until they are destroyed (based on some other rule)

Creation & Destruction

- This means that we need an efficient way of handling a variable number of particles
- For a realtime system, it's usually a good idea to allocate a fixed maximum number of particles in an array, and then use a subset of those as active particles
- When a new particle is created, it uses a slot at the end of the array (cost: 1 integer increment)
- When a particle is destroyed, the last particle in the array is copied into its place (cost: 1 integer decrement & 1 particle copy)
- For a high quality animation system where we're not as concerned about performance, we could just use a big list or variable sized array

Creation Rules

- It's convenient to have a 'CreationRule' as an explicit class that contains information about how new particles are initialized
- This way, different creation rules can be used within the same particle system
- The creation rule would normally contain information about initial positions, velocities, colors, sizes, etc., and the variance on those properties
- A simple way to do creation rules is to store two particles: mean & variance (or min & max)

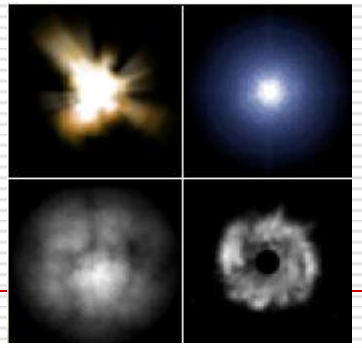
Creation Rules

- In addition to mean and variance properties, there may be a need to specify some geometry about the particle source
- For example, we could create particles at various points (defined by an array of points), or along lines, or even off of triangles
- One useful effect is to create particles at a random location on a triangle and give them an initial velocity in the direction of the normal. With this technique, we can emit particles off of geometric objects

Destruction

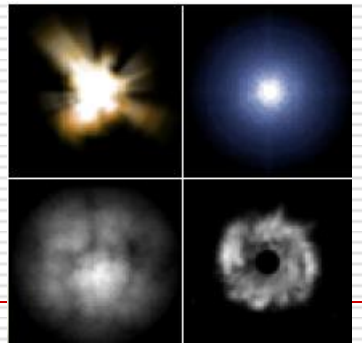
- Particles can be destroyed according to various rules
- A simple rule is to assign a limited life span to each particle (usually, the life span is assigned when the particle is created)
- Each frame, it's life span decreases until it gets to 0, then the particle is destroyed
- One can add any other rules as well
- Sometimes, we can create new particles where an old one is destroyed. The new particles can start with the position & velocity of the old one, but then can add some variance to the velocity. This is useful for doing fireworks effects…

Particle Rendering



- Particles can be rendered using various techniques
  - Points
  - Lines (from last position to current position)
  - Sprites (textured quad's facing the camera)
  - Geometry (small objects…)
  - Or other approaches…
- For the particle physics, we are assuming that a particle has position but no orientation. However, for rendering purposes, we could keep track of a simple orientation and even add some rotating motion, etc…

Particle Rendering



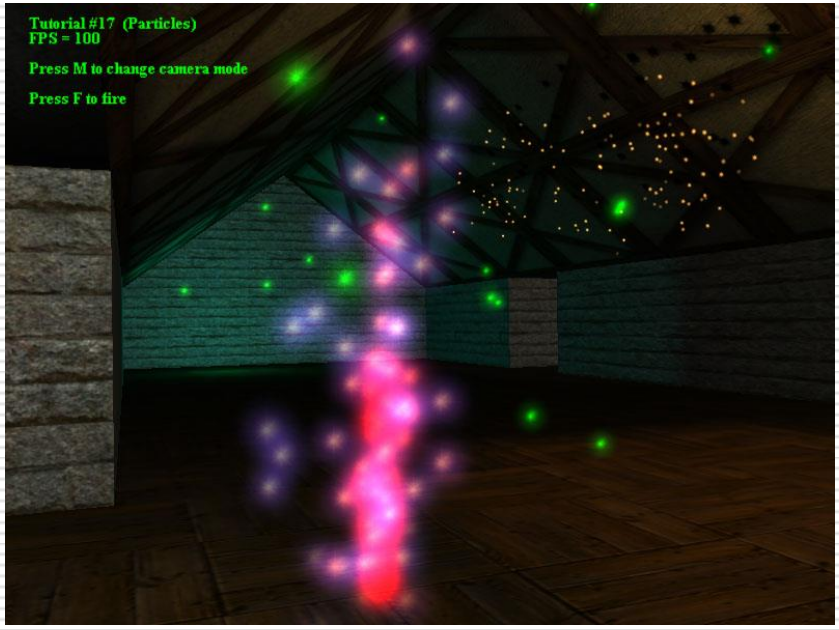
- Particles display as(OpenGL points, (Textured) billboarded quads, Point sprites[6])
- Sample code:

```
glTexEnvf(GL_POINT_SPRITE, GL_COORD_REPLACE,
GL_TRUE);
glEnable(GL_POINT_SPRITE);
glBegin(GL_POINTS);
        glVertex3f(position.x, position.y, position.z);
glEnd();
glDisable(GL_POINT_SPRITE);
```

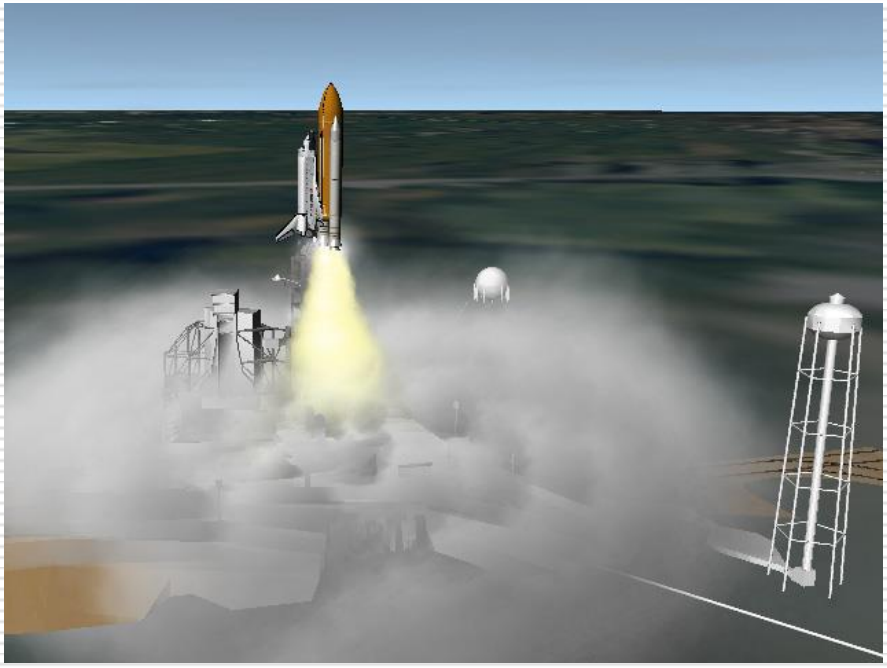
Reference

- Reeves W.: "Particle Systems -- A Technique for Modelling a Class of Fuzzy Objects", Computer Graphics, 17(3), pp. 359-376, 1983
- [Building a Million Particle System Lutz Latta Massive Development GmbH] http://www.2ld.de/gdc2004/MegaParticlesSlides.pdf
- OpenGL Extensions & Advanced rendering https://home.zhaw.ch/~frp/CGr/Unterlagen/12_OpenGL_Advanced.pdf]
- Implementing Particle Systems in OpenGL SPSU (southern polytechnic state university)
- http://www.lri.fr/~mbl/ENS/IG2/devoir2/files/docs/fuzzyParticles.pdf
- http://www.naturewizard.com/tutorial08.html
- J ü rgen P. Schulze, Ph.D.University of California, San Diego Fall Quarter 2011, Lecture #18
- Computer Graphics Lecture Notes University of Toronto November 24, 2006
- lecture on Advanced Computer Graphics by Bas Zalmstra and Marries van de Hoef
- Procedural Smoke Particle System with OpenGL 2.0 Tommy Hinks – tomhi761@student.liu.se
- http://en.wikipedia.org/wiki/Particle_system
- http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter06.html
- Particle Animation and Rendering Using Data Parallel Computation Karl Sims
- http://en.wikipedia.org/wiki/Euler_equations_(fluid_dynamics)
- http://en.wikipedia.org/wiki/Verlet_integration

Particle System (PS) Gallery



Particle System (PS) Gallery



End.
Thanks