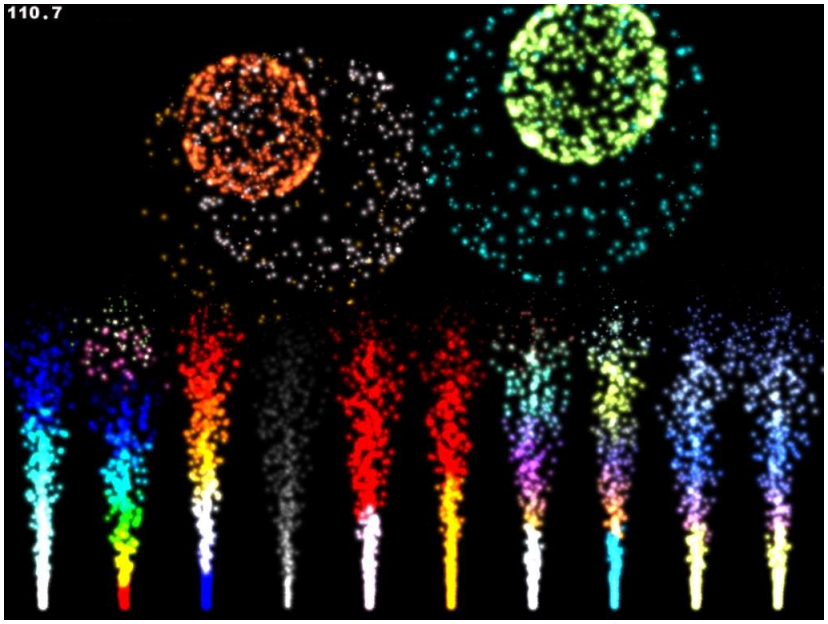
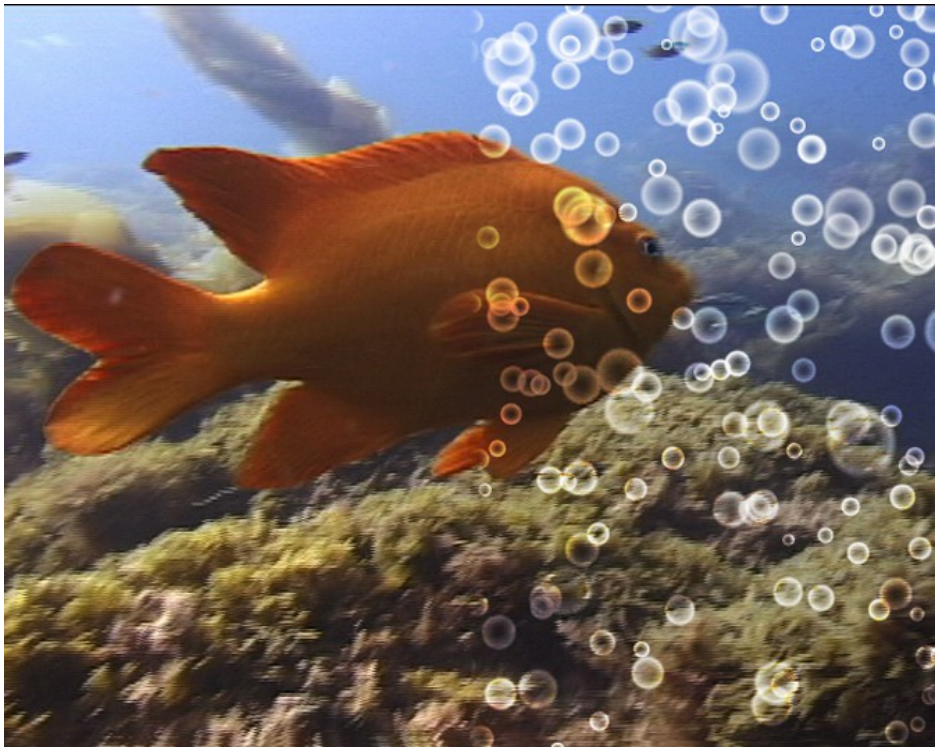
Particle Systems


110.7



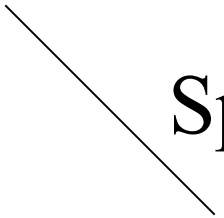
Particle Systems

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

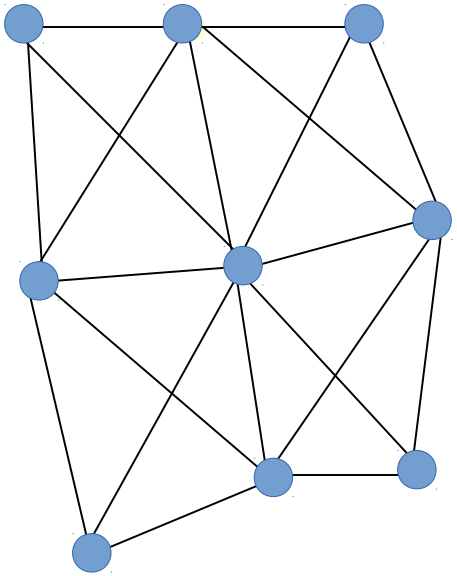


Cloth Simulation

Particle



Spring-damper



Physics

Kinematics of Particles

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

$$r = r(t)$$

$$v = \frac{dr}{dt}$$

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

Uniform Acceleration

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

$$a = a_0$$

$$v = \int_0^t a\,dt = v_0 + a_0 t$$

$$r = \int v\,dt = r_0 + v_0 t + \frac{1}{2} a_0 t^2$$

Uniform Acceleration

$$r = r_0 + v_0 t + \frac{1}{2} a_0 t^2$$

- This shows us that the particle's motion will follow a parabola
- Keep in mind, that this is a 3D vector equation (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

$$p = mv$$

Newton's First Law: momentum preservation

Force

- Force is defined as the rate of change of momentum

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

- We can expand this out:

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

$$f = ma$$

Newton's Second Law 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

$$f_{AB} = -f_{BA}$$

- Put another way: every action has an equal and opposite reaction
- This is very important when combined with the second law, as the two together imply the conservation of momentum

Forces on a Particle

- Usually, a particle will be subjected to several simultaneous vector forces from different sources
- All of these forces simply add up to a single total force acting on the particle

$$f_{total} = \sum f_i$$

Particle Simulation

- Basic kinematics allows us to relate a particle's acceleration to it's resulting motion
- Newton's laws allow us to relate acceleration to force
- This gives us a general scheme for simulating particles (and more complex things):

Particle Simulation

- Basic kinematics allows us to relate a particle's acceleration to it's resulting motion
- Newton's laws allow us to relate acceleration to force
- This gives us a general scheme for simulating particles (and more complex things):

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

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();
}
```

'forward Euler integration'

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

Forces

Uniform Gravity

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

$$f_{gravity} = mg_0$$

$$g_0 = \begin{bmatrix} 0 & -9.8 & 0 \end{bmatrix} \quad \frac{m}{s^2}$$

- It assumes that we are near the surface of a planet with a huge enough mass that we can treat it as infinite
- As we don't apply any equal and opposite forces to anything, it appears that we are breaking Newton's third law, however we can assume that we are exchanging forces with the infinite mass, but having no relevant affect on it

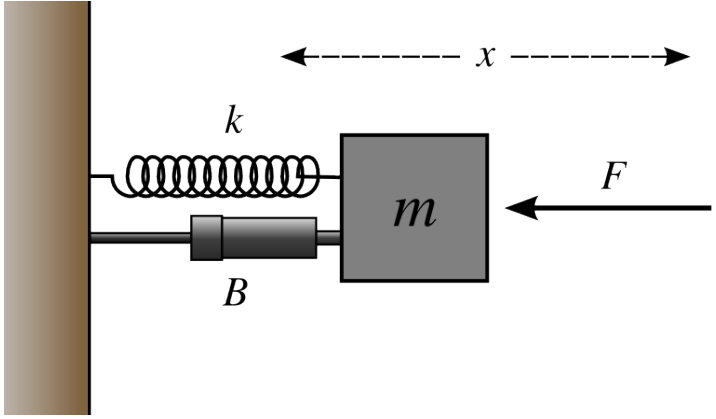
Gravity

- 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:

$$f_{gravity} = \frac{Gm_1 m_2}{d^2} e \qquad e = \frac{r_1 - r_2}{|r_1 - r_2|}$$

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

Springs

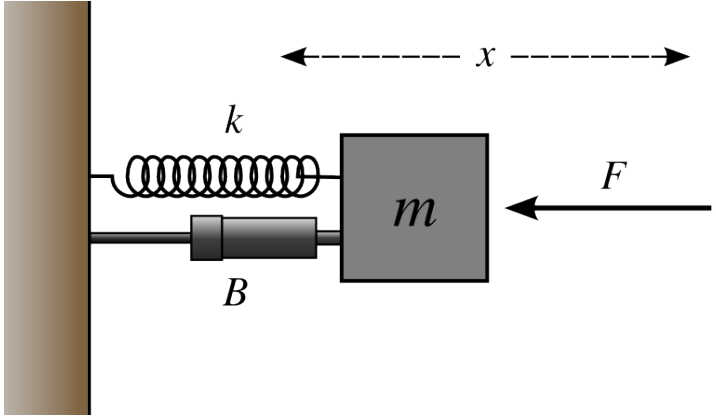


- A simple spring force can be described as:

$$f_{spring} = -k_s x \qquad x = r_1 - r_2$$

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

Springs



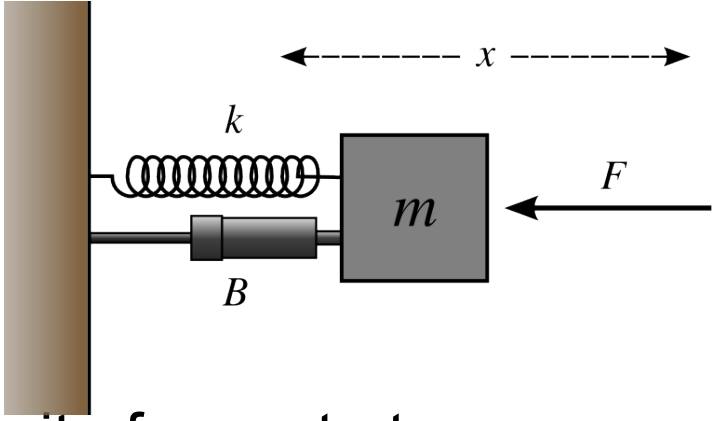
- In practice, it's nice to define a spring as connecting two particles and having some rest length *l* where the force is 0
- This gives us:

$$x = xe$$

$$x = |r_1 - r_2| - l \quad \text{(scalar displacement)}$$

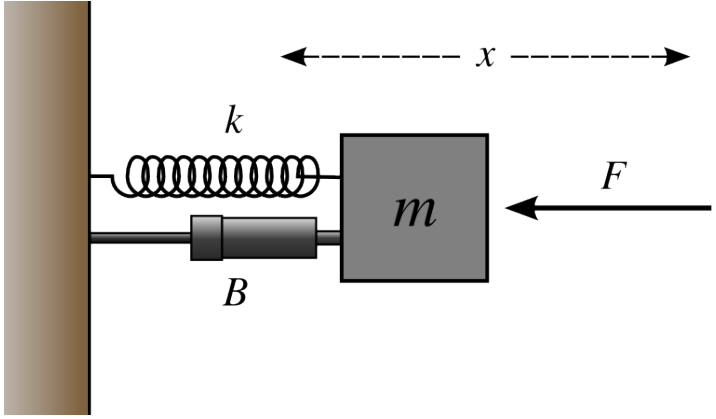
$$e = \frac{r_1 - r_2}{|r_1 - r_2|} \quad \text{(direction of displacement)}$$

Springs



- As springs apply equal and opposite forces to two particles, they should obey conservation of momentum
- It will not, however guarantee the conservation of energy, and in practice, we might see a gradual increase or decrease in system energy over time
- A gradual decrease of energy implies that the system damps out and might eventually come to rest. A gradual increase, however, is not so nice… (more on this later)

Dampers

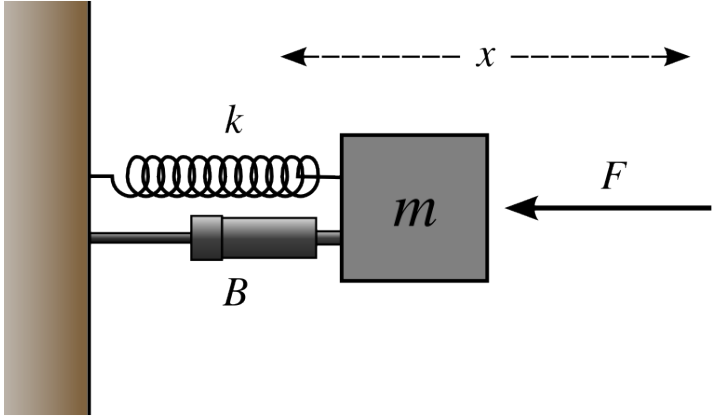


- We can also use damping forces between particles:

$$f_{damp} = -k_d v$$

- 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
- The kinetic energy is effectively lost.

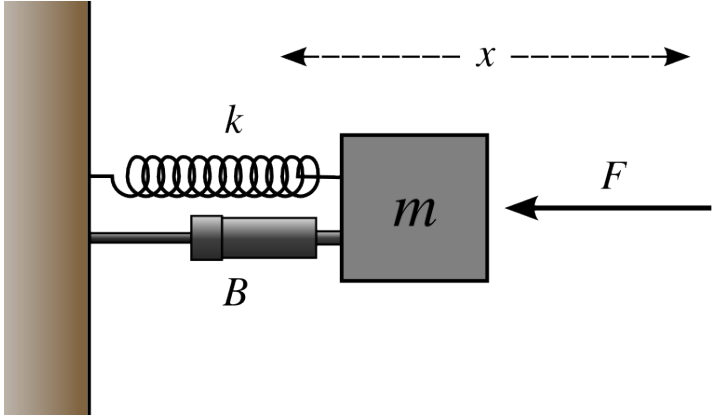
Dampers



- 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();
};
```

Dampers



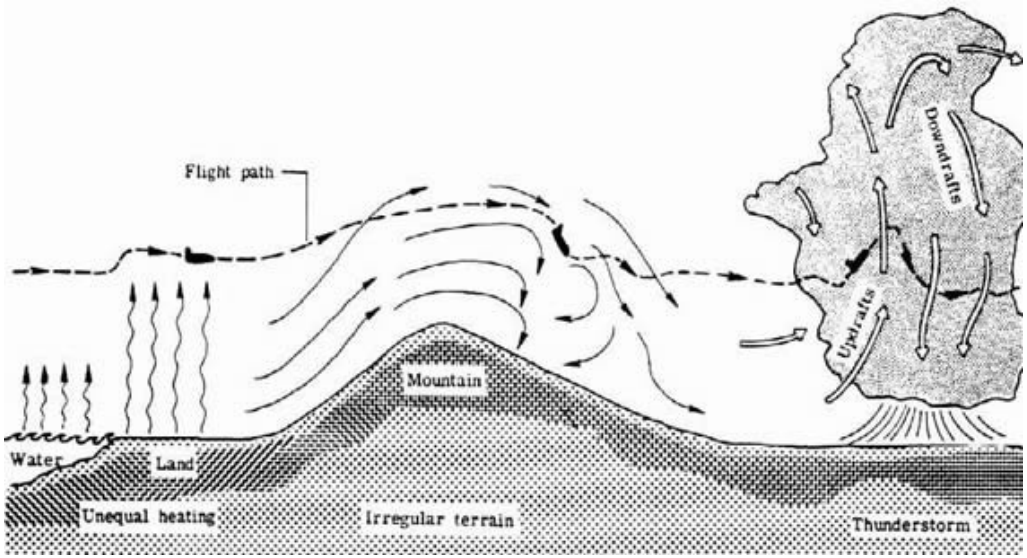
- To compute the damping force, we need to know the *closing* velocity of the two particles, or the speed at which they are approaching each other

$$e = \frac{r_1 - r_2}{|r_1 - r_2|}$$

$$v = v_1 \cdot e - v_2 \cdot e$$

- This gives us the instantaneous closing velocity of the two particles

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

$$f_{field} \propto f(r)$$

- 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, winds, vortices, and other flow patterns

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:

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

- 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

Integration

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

Forward Euler Integration

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

$$v_{n+1} = v_n + a_n \Delta t$$

$$r_{n+1} = r_n + v_{n+1} \Delta t$$

which expands to

$$r_{n+1} = r_n + \left(v_n + a_n \Delta t \right) \Delta t$$

$$= r_n + v_n \Delta t + a_n \left(\Delta t \right)^2$$

Forward Euler Integration

- Note that this:

$$r_{n+1} = r_n + v_n \Delta t + a_n (\Delta t)^2$$

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:

$$r_{n+1} = r_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2$$

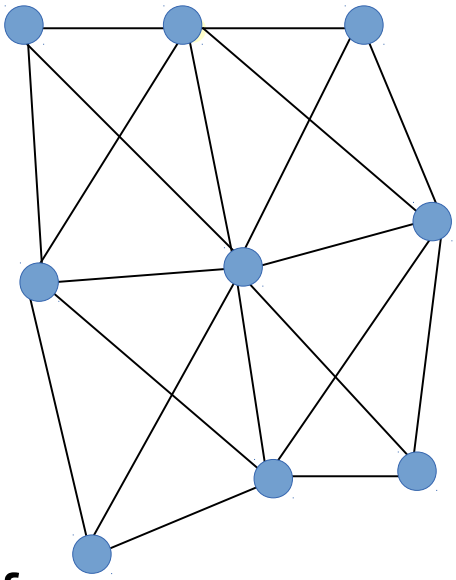
Forward Euler Integration

- 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

Cloth Simulation

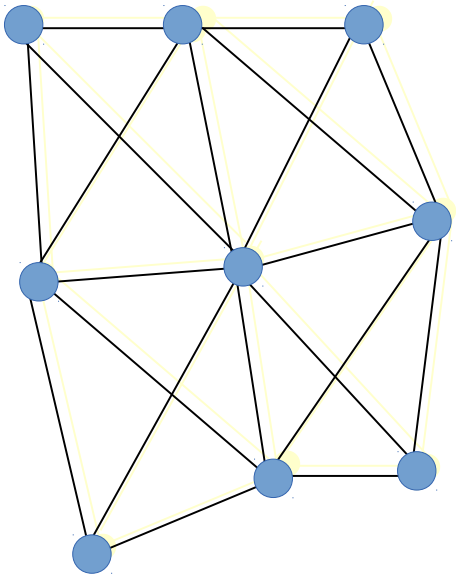


1. Compute Forces
   For each particle: Apply gravity
   For each spring-damper: Compute & apply forces
   For each triangle: Compute & apply aerodynamic forces

2. Integrate Motion
   For each particle: Apply forward Euler integration

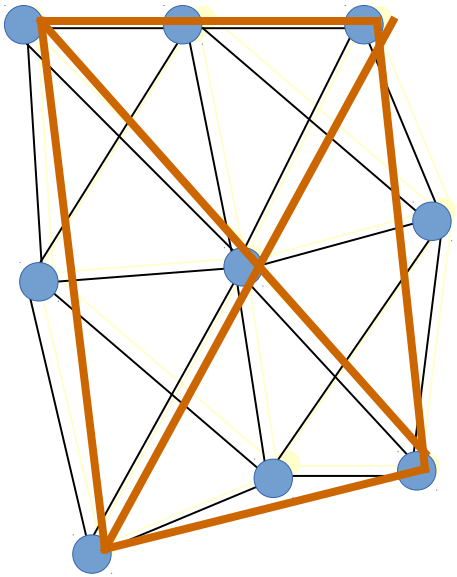
Bending Forces

- If we arrange our cloth springs as they are in the picture, there will be nothing preventing the cloth from bending

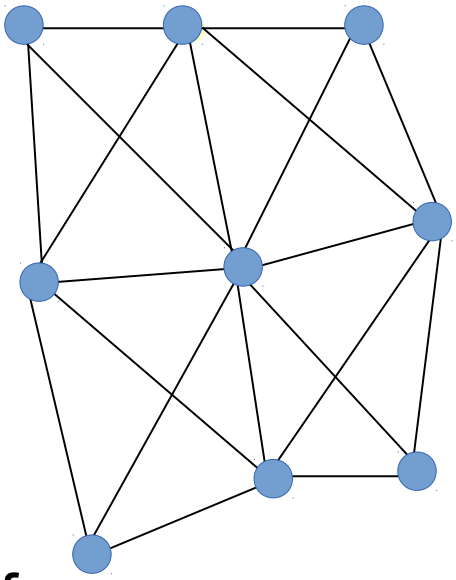


Bending Forces

- If we arrange our cloth springs as they are in the picture, there will be nothing preventing the cloth from bending
- A simple solution is to add more springs, arranged in various configurations, such as the one in the picture
- The spring constants of this layer might need to be tuned differently…



Cloth Simulation

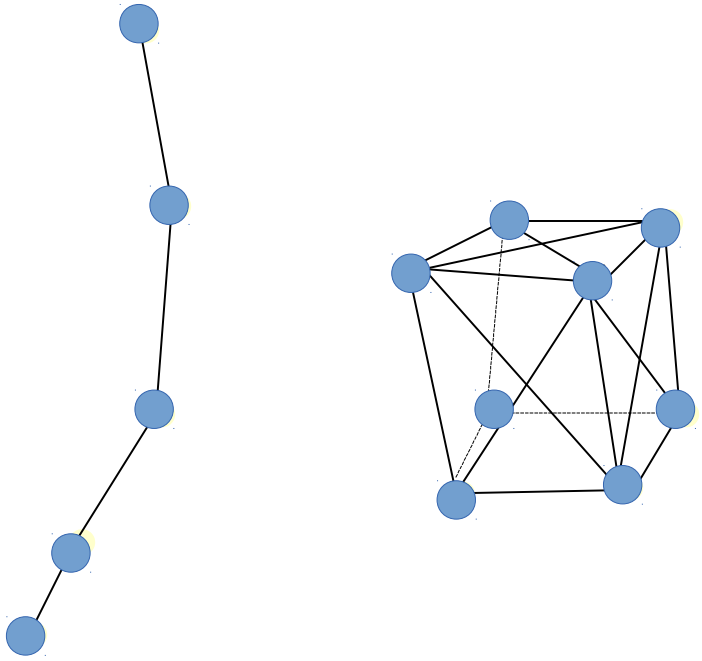


1. Compute Forces
   For each particle: Apply gravity
   For each spring-damper: Compute & apply forces
   For each triangle: Compute & apply aerodynamic forces

2. Integrate Motion
   For each particle: Apply forward Euler integration

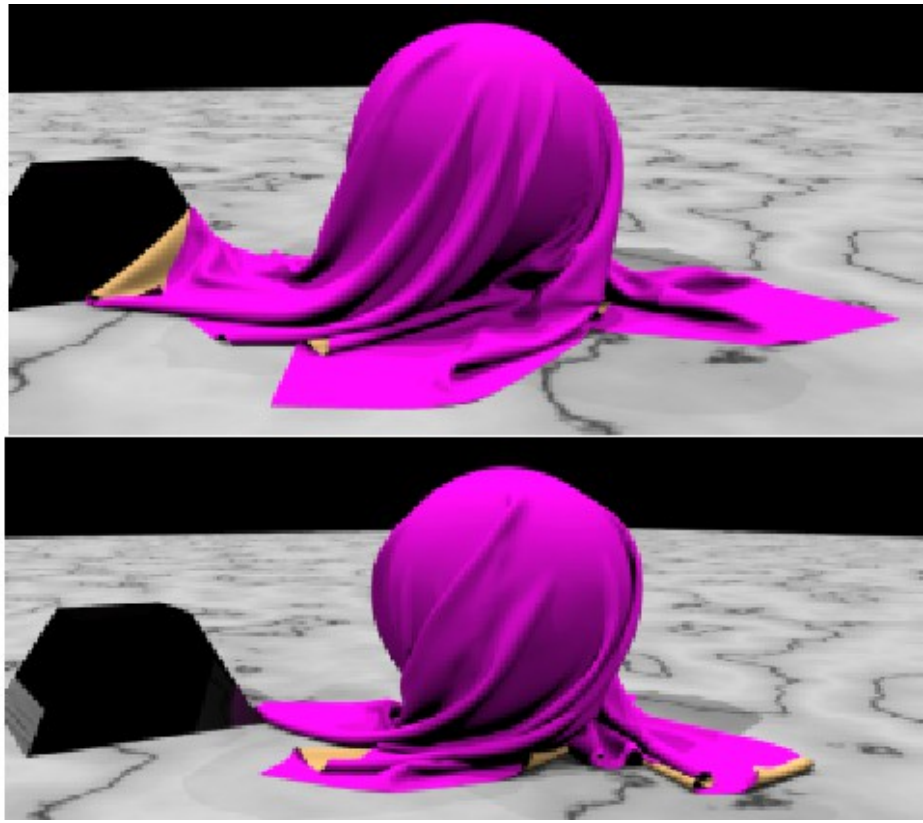
Ropes & Solids

- We can use this exact same scheme to simulate ropes, solids, and similar objects



Collision Detection & Response

- Cloth colliding with rigid objects is tricky
- Cloth colliding with itself is even trickier
- There have been several published papers on robust cloth collision detection and response methods



Collisions

- Here's a very basic way to collide with a $y=y_0$ plane

```
If(r.y < y0) {
    r.y= y0 - r.y;
    v.y= - elasticity * v.y;
    v.x= (1-friction) * v.x;          // cheezy
    v.z= (1-friction) * v.z;          // cheezy
}
```

Demo videos

Particle Systems

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…

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)

Randomness

- An important part of making particle systems look good is the use of randomness
- Giving particle properties a good initial random distribution can be very effective
- Properties can be initialized using uniform distributions, Gaussian distributions, or any other function desired

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…

Appendix

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, sometimes 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:

$$j = \int f\,dt = \Delta p$$

- 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

$$f = \sum f_i$$

$$j = \sum j_i$$

// Integrate to get new velocity & position

$$v' = v_0 + \frac{1}{m}\left(f\Delta t + j\right)$$

$$r' = r_0 + v'\Delta t$$

Collisions

- Today, we will just consider the simple case of a particle colliding with a static object
- The particle has a velocity of **v** before the collision and collides with the surface with a unit normal **n**
- We want to find the collision impulse **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

Collisions

- Let's first consider a collision with no friction
- The collision impulse will be perpendicular to the collision plane (i.e., along the normal)

$$v_{close} = v \cdot n$$
$$j = -(1+e) m v_{close} n$$

- That's actually enough for collisions today. We will spend a whole lecture on them next week.

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 step

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…