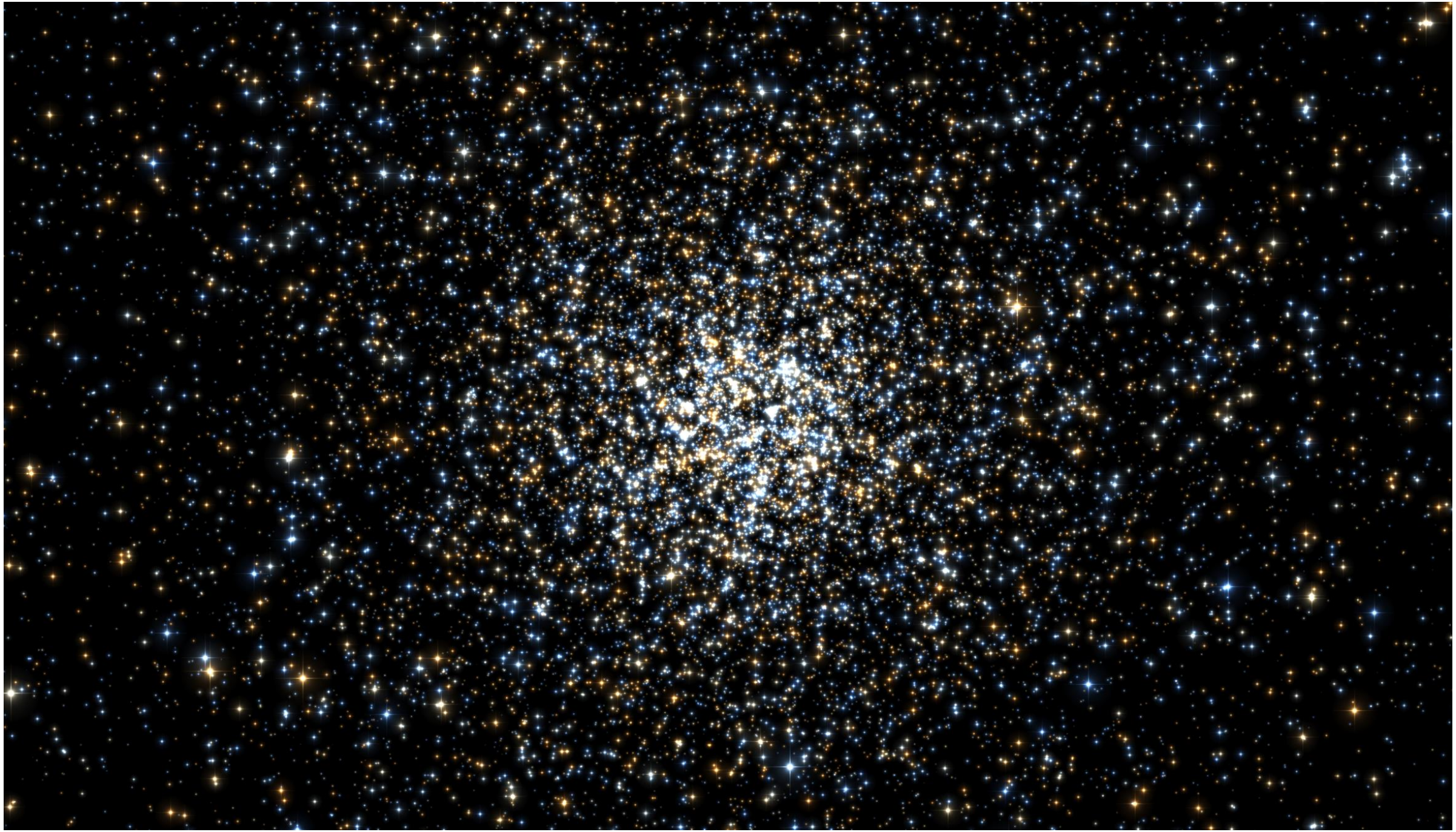
Particle Systems

Particle Systems

- Particle systems have been used extensively in computer animation and special effects since their introduction in the early 1980's
- The rules governing the behavior of an individual particle can be relatively simple – and the complexity comes from having many particles
- Usually, particles will follow some combination of physical and non-physical rules, depending on the exact situation



Physics

Kinematics of Particles

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

$$\mathbf{v}(t) = \frac{d\mathbf{r}}{dt}$$

- And acceleration is the second derivative:

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

Uniform Acceleration

- How does a particle move when undergoing a **constant** acceleration?

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

Uniform Acceleration

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

- This shows us that a particle undergoing a constant acceleration will follow a parabola
- Keep in mind that this is a 3D vector equation and there is potentially a parabola equation in each dimension. Together, they will 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

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:

$$\mathbf{a} = 0$$

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

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

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 $\mathbf{p}$, which is the product of scalar mass and vector velocity

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

Force

- Force is defined as the rate of change of momentum

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

- If we assume that mass *m* is constant, we can expand this to:

$$\mathbf{f} = \frac{d\mathbf{p}}{dt} = \frac{d(m\mathbf{v})}{dt} = m\frac{d\mathbf{v}}{dt} = m\mathbf{a}$$

$$\boxed{\mathbf{f} = m\mathbf{a}}$$

Newton's Second Law

- Newton's Second Law says:

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

- This relates the *kinematic* quantity of acceleration $\mathbf{a}$ to the *physical* quantity of force $\mathbf{f}$

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

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

- 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 Law of Conservation of Momentum

Conservation of Momentum

- Remember that a force is a rate of change of momentum
- If Newton's Third Law says that the forces in a system cancel out, then the total change of momentum of the system must be 0
- Therefore, the total momentum in a system must remain constant
- This is the Law of Conservation of Momentum

Conservation of Momentum

- Conservation of Momentum is an important property to preserve in a physical simulation
- However, for non-physical effects, we are allowed to break the rules
- Even for physically valid simulations, we will sometimes only implicitly follow this rule. For example, we can explicitly apply an aerodynamic drag force to a particle but only implicitly apply the equal and opposite force to the air itself, as we aren’t actually simulating the air

Forces on a Particle

- A particle may 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:

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

Newtonian Mechanics

- Newton's Second Law relates the property of force to the kinematic property of acceleration through a measureable constant mass:

$$\mathbf{f} = m\mathbf{a}$$

- Forces are a very useful quantity to work with because of Newton's Third Law:

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

- And because they add up in a very simple way:

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

- These principles form the foundation of all Newtonian based simulations, including solid dynamics, rigid body dynamics, and fluid dynamics

Integration

Integration

- Newtonian simulation involves working with forces
- As we've seen, forces relate to accelerations through Newton's Second Law $f=ma$
- Ultimately however, we need to compute positions in order to advance the simulation forward and visualize what's happening
- Position is the integral of velocity and velocity is the integral of acceleration
- Therefore, the process of integration will be central to physics simulation

Analytical (Symbolic) Integration

- If we have a relatively simple mathematical function, we can usually compute an analytical integral
- For example, if our function is a polynomial like:

$$f(t) = 3t^2 + 4t + 5$$

- then we can compute the integral as

$$\int f(t)dt = t^3 + 2t^2 + 5t + c$$

- Analytical integration calculates an exact solution to the integral.

Numerical Integration

- However, many mathematical functions can't be integrated analytically, and this applies to most of the situations we'll be interested in.
- Therefore, we will accept that we will rely on *numerical integration* techniques throughout the entire quarter.
- Numerical integration uses iteration and approximation to compute "brute force" results, and so can suffer from problems with accuracy and stability.
- Still, we have little choice if we want to simulate complex problems, so we must understand these properties in order to make use of numerical integration techniques.

Forward Euler Integration

- The forward Euler method uses the derivative at the start of the time step to advance the simulation forward
- For example, to compute the new velocity at time step *i*+1, we use the acceleration computed at time step *i* and assume it holds constant for the duration of Δt

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \Delta t$$

- For the position integration, we do essentially the same thing, except we use the new velocity instead of the previous velocity

$$\mathbf{r}_{i+1} = \mathbf{r}_i + \mathbf{v}_{i+1} \Delta t$$

Particle Simulation

- Dynamics simulations generally follow a pattern like this:

  Specify initial conditions for time t_0
  while (not finished) {
    - Evaluate all forces in current configuration at time t_n and use these to compute all accelerations
    - Integrate accelerations over some finite time step Δt to advance everything to new positions (and velocities) at new time t_{n+1}
    - Display/store/analyze results

  }

- Sure, the details will get more complicated, but in general, we are taking finite steps forward in time and evaluating forces at each step

Particle Example

```
class Particle {
public:
        void ApplyForce(vec3 &f)        {Force+=f;}
        void Integrate(float deltaTime) {
                vec3 accel=(1/Mass) * Force;
                Velocity += accel*deltaTime;
                Position += Velocity*deltaTime;
                Force=vec3(0);
        }
private:
        vec3 Position;
        vec3 Velocity;
        vec3 Force;
        float Mass;
};
```

Energy

- The kinetic energy of a particle is $\frac{1}{2}m|\mathbf{v}|^2$
- There are also various forms of potential energy such (gravity, springs, etc. can store energy as a potential)
- Energy in a system may convert between different forms (kinetic, potential, thermal, electromagnetic…) but the total energy in a system remains constant
- The subject of energy is important in physics, but Newtonian formulations of the equations rarely make direct use of it
- We will therefore not discuss it much today, but it will come back once or twice in later lectures

Basic Forces

Uniform Gravity

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

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

$$\mathbf{g}_0 = [0 \quad -9.8 \quad 0]\frac{m}{s^2}$$

- It assumes that we are near the surface of the Earth and we can approximate the gravity as constant in both magnitude and direction
- 9.8 m/s^2 is a reasonable approximation, as it actually ranges from roughly 9.76 to 9.83 around the world due to variations in altitude and local density (there are detailed maps of this available)

Inverse-Square Gravity

- If we are modeling orbital mechanics, planetary systems, or galaxies, we need to consider the full inverse-square law of gravity acting between two bodies

$$\mathbf{f}_{gravity} = G\frac{m_1 m_2}{d^2}\mathbf{e}$$

- Where *G* is the universal *gravitational constant* (2014 version):

$$G = 6.67408 \times 10^{-11}\ \frac{m^3}{kg{\cdot}s^2}$$

- *d* is the distance between the two bodies: $d = |\mathbf{r}_1 - \mathbf{r}_2|$

- And **e** is a unit length vector pointing in the direction of gravitational attraction (i.e., towards the other body)

Inverse-Square Gravity

$$\mathbf{f}_{gravity} = G\frac{m_1 m_2}{d^2}\mathbf{e}$$

- We need to consider the gravitational force acting on every *pair* of bodies
- In a system of n bodies, this means we need to compute a gravitational force $n(n-1)/2$ times
- In terms of algorithm performance, this implies O(n^2) performance, which is potentially slow for large values of n
- It turns out that for galactic simulations with millions of particles, we can actually achieve O(n log n) performance using some octree techniques, and for some more limited cases, we can even achieve O(n) performance using some Fourier techniques

Aerodynamic Drag

- Aerodynamic interactions are very complex and difficult to model accurately
- For particles, we can use a reasonable simplification to describe the total aerodynamic drag force on an object:

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

- Where ρ is the density of the surrounding fluid (air, water, etc.), c_d is the coefficient of drag for the object, a is the cross sectional area of the object, and $\mathbf{e}$ is a unit vector in the opposite direction of the velocity:

$$\mathbf{e} = -\frac{\mathbf{v}}{|\mathbf{v}|}$$

- Also, keep in mind that we really want the relative velocity, which is the different between the particle velocity and the average velocity of the surrounding fluid

$$\mathbf{v} = \mathbf{v}_{particle} - \mathbf{v}_{fluid}$$

Fluid Density: ρ

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

- The fluid density ρ of air at 15^o C and a pressure of 101.325 kPa (14.696 psi) is 1.225 kg/m^3 and is used as a common default value
- The fluid density ρ of liquid water is 999.8 kg/m^3 at 0^o C and 997.0 kg/m^3 at 25^o C at sea level

Drag Coefficient: c_d

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

- The aerodynamic drag force uses a unit-less constant c_d called the drag coefficient
- This number effectively quantifies the aerodynamic drag of the particular shape, and typically ranges from around 0.01 (very streamlined) to 1.5 (bluff body)
- A sphere has a c_d around 0.47 and a cube has a c_d around 1.05
- The Tesla Model 3 has a c_d of 0.23 and a Jeep Wrangler has a c_d of 0.58

Cross Sectional Area: a

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

- In the aerodynamic drag equation above, *a* refers to the *cross sectional area* of the object moving through the surrounding fluid
- This means the area when viewed from the direction of motion
- For a spherical object of radius *r*, it would be πr^2

Aerodynamic Drag

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

- Most of the values here are constants (ρ, c_d, a), and **e** is just used to specify the direction the force acts
- Therefore, when we really boil this down, we see that the aerodynamic drag force is proportional to velocity squared

$$f_{drag} \propto v^2$$

Springs

- We can use Hooke's Law to model simple linear spring forces:

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

- Where k_s is the *spring constant* describing the stiffness of the spring and **x** is a vector describing the displacement
- The spring force is therefore going to work against the displacement
- The direction of the force will be along the axis of the spring and will pull if the spring is extended and push if the spring is compressed

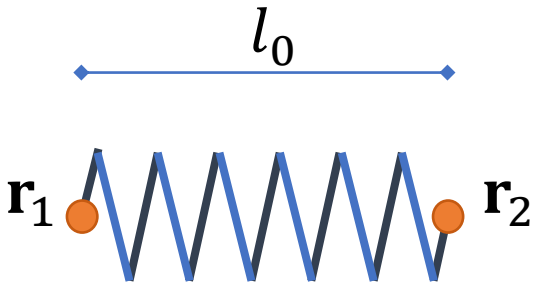
Springs

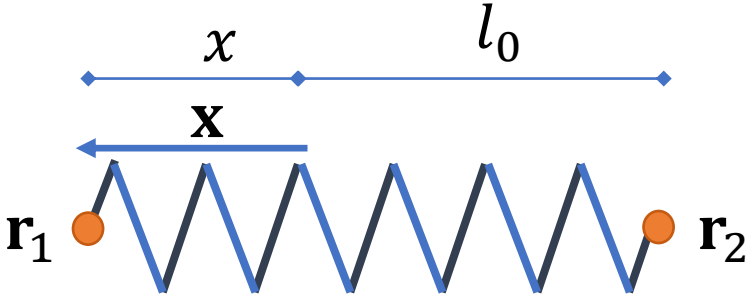
- In practice, it's nice to define a spring as connecting two particles and having a *rest length* l_0 where the spring force is 0
- This gives us:

$$x = |\mathbf{r}_1 - \mathbf{r}_2| - l_0$$

$$\mathbf{e} = \frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|}$$

$$\mathbf{x} = x\mathbf{e}$$





Springs

- A spring applies equal and opposite forces to two particles, and therefore explicitly obeys Newton's Third Law
- They should also obey the Laws of Conservation of Momentum and Conservation of Energy
- In practice however, how well they obey these laws is due to the numerical integration scheme used

Dampers

- We can apply damping forces between particles:

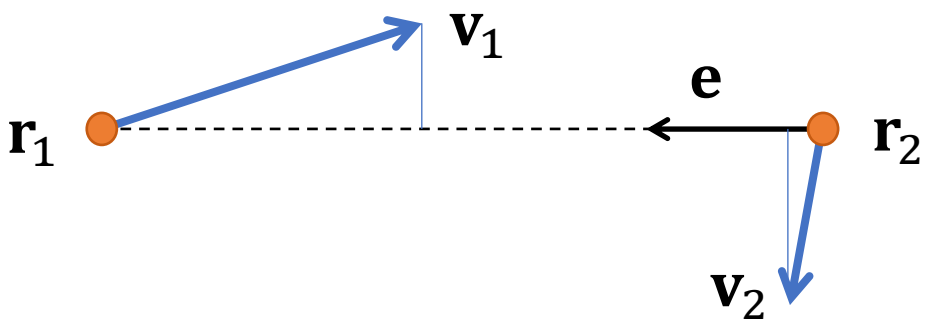
$$\mathbf{f}_{damp} = -k_d v_{close} \mathbf{e}$$

- k_d is the damping constant, v_{close} is the closing velocity, and $\mathbf{e}$ is a unit vector that provides the direction ($\mathbf{e}$ works the same as with springs)
- Dampers will oppose any difference in velocity between particles
- The damping forces are equal and opposite, so they should conserve momentum, but they will remove energy from the system by design

Closing Velocity

- To calculate the damping force, we need to calculate the *closing velocity* between two particles
- This is the rate that the two particles are approaching each other

$$v_{close} = (\mathbf{v}_2 - \mathbf{v}_1) \cdot \mathbf{e}$$



Combining Forces

- All of the different forces we've examined can be combined by simply adding them together
- The total force on a particle is just the sum of all of the individual forces

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

- In each step of the simulation, we compute all of the forces in the entire system at the particular instant
- We then use those forces to integrate the accelerations to compute new velocities and positions at some finite time step later

Particle Systems

Particle Properties

- Particles will always have fundamental properties like position and velocity and mass
- In addition, they can have other useful properties:
  - Color
  - Size (radius?)
  - Life span
  - Other stuff…
- All of these properties are assigned to some initial value when a particle is created and some or all may change over the life of the particle according to various rules

Randomness

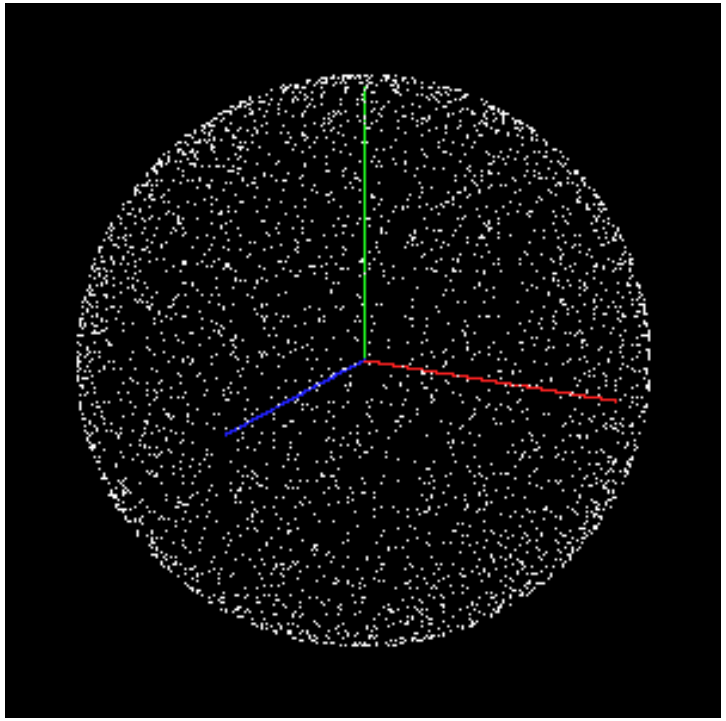
- Randomness is a key aspect of particle systems
- Usually, initial properties are given a range of values, and individual particles are assigned a random value within that range
- The distribution over the range could be uniform or non-uniform (such as Gaussian, etc.)
- It can be nice to specify a range as a min/max, or possibly as a mean/variance

Random Direction Vector

- If we want to generate a random velocity, it is nice to be able to produce a uniform spherical distribution of directions
- To generate a random direction vector $\mathbf{d}$, we start with two random numbers s and t in the [0…1] interval:

$$u = 2\pi s$$
$$v = \sqrt{t(1-t)}$$

$$d_x = 2v \cdot \cos(u)$$
$$d_y = 1 - 2t$$
$$d_z = 2v \cdot \sin(u)$$



Creation & Destruction

- Sometimes, we'll have a particle system with a fixed number of particles, but most often, we will want to create and destroy particles on the fly
- It is nice to have various creation & destruction rules that can be applied to a particle system to customize its behavior
- This implies that we need an efficient method of adding and removing particles from the system

Efficient Particle Management

- In most situations, we can put some kind of upper limit on the number of particles that a particular effect requires and pre-allocate a buffer to support that
- If not, we can use a re-sizeable array such as a std::vector and dynamically add new particles as needed, but they should be added as a single block per frame, not one at a time
- To remove a single particle from the middle of the array, a good strategy is to just replace it with the last particle and decrease the particle count. This has the cost of copying a few bytes, rather than collapsing down the entire array. This assumes of course, that the order of the particles is not relevant, which might not always be the case…

Creation Rules

- When a particle is created, one must set its initial position, velocity, and other attributes
- It is often nice to be able to specify some type of geometry of the particle source, along with a particle creation rate
- For example, the source geometry could be a point, line, curve, triangle, or even a full triangle mesh
- This source geometry could also animate over time…

Creating Particles from Triangle Meshes

- For example, let’s say we are given a triangle mesh and we want to create particles from it
- We want to create new particles that are equally distributed over the total surface area
- This means that when creating a new particle, we need to choose a random triangle weighted by area so that large triangles are more likely than small ones (this isn’t hard, but isn’t exactly trivial either)
- Once we’ve chosen a triangle, we need to choose a random point within the triangle and use that as the initial position
- We can base the initial velocity off of the triangle normal, plus some randomness

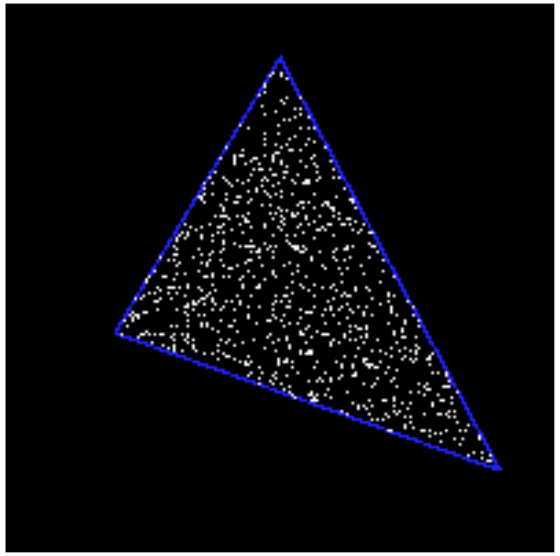
Random Point on a Triangle

- There are a variety of reasons why we might want to choose a random point on a triangle
- Given a triangle defined by three vertices **a**, **b**, and **c**, and two random numbers *s* and *t* in the [0…1] interval, we proceed by generating random barycentric coordinates α and β:

$\alpha = \text{sqrt}(s) * t$

$\beta = 1 - \text{sqrt}(s)$

$\mathbf{p} = \mathbf{a} + \alpha(\mathbf{b}-\mathbf{a}) + \beta(\mathbf{c}-\mathbf{a})$



Creation Rate

- For any particular particle source, we want to be able to specify its creation rate in particles per second (PPS)
- If we are running a simulation at 60 frames per second and creating particles at 1000 PPS, we can just create 1000/60 particles per frame which is around 16 or 17
- However, if we want to create 5 per second, we need to create one particle every 11 frames or so
- This means that we should keep track of the roundoff error over time to ensure an accurate creation rate

Creation Rate

```
void ParticleSource::Update(float deltaTime) {
        // Determine how many new particles to create this frame
        float num=deltaTime*CreationRate+RoundOffError;
        int newParticles=int(num);
        RoundOffError=num-float(newParticles);

        // Create particles
        …
}
```

First Frame Adjustments

- If we are creating a fast stream of particles at 100 per frame, we want them to appear uniformly distributed, rather that appearing as lots of little puffs of 100 particles
- To fix this, we can assume the particle was created at some point *within* the frame, rather than exactly at the beginning/end of the frame
- This means that we should update newly created particles by a random number times the time step, thus distributing them within the frame

Destruction

- It is common to assign a particle a fixed life span at the time it is created. Each frame, the life span decreases until it gets to 0 and the particle is removed
- One can also add other destruction rules such as:
  - Falls below some height value
  - Outside of bounding region
  - Collision with an object

- Note that it can also be nice to create new particles upon the destruction of an old one. For example, fireworks can be done by launching an initial particle and when it dies, it creates a bunch of new particles that inherit its position and velocity for their initial conditions…

Particle Rendering

- Rendering is another key issue of particle systems
- There are various options:
  - Points
  - Line from last position to current position
  - ‘Sprites’
  - Mesh geometry (spheres, etc.)