

# Game Physics

---

305890  
Spring 2013  
6/7/2013  
Kyoung Shin Park  
kpark@dankook.ac.kr

## Application in Video Games

---

- Racing games: Cars, snowboards, etc..
  - Simulates how cars drive, collide, rebound, flip, etc..
- Sports games
  - Simulates trajectory of soccer, basket balls.
- Increasing use in First Person Shooters: Unreal
  - Used to simulate bridges falling and breaking apart when blown up.
  - Dead bodies as they are dragged by a limb.
- Miscellaneous uses:
  - Flowing flags / cloth.
- Problem is that real time physics is very compute intensive. But it is becoming easier with faster CPUs.

## Definitions

---

- Kinematics (운동학)
  - Study of movement over time.
  - Not concerned with the cause of the movement.
- Dynamics (동역학)
  - Study of forces and masses that cause the kinematic quantities to change as time progresses.

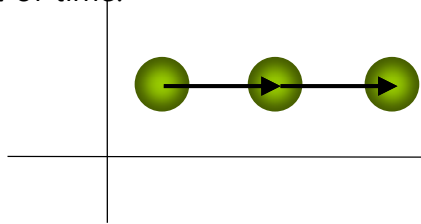
## Game Physics

---

- Motion (운동)
- Position (위치), Velocity (속도), Acceleration (가속도)
- Force (힘), Gravity (중력)
- Buoyancy (부력), Drag (저항력)
- Friction (마찰력)
  - Kinetic friction (운동마찰)
  - Static friction (정지마찰)
- Spring (스프링)

## Motion

- In physics, motion is a change in location or position of an object with respect to time.
- Object motion is represented with vectors
- Velocity is a vector
  - Vector direction is direction of movement
  - Vector magnitude is speed of movement
- Velocity vector corresponds to amount object will move in one unit of time.



## Basic Motion

- Displacement (변위) = velocity \* time
- If an object starts at position,  $P_0$  with velocity  $v$ , after  $t$  time units, its position  $P(t)$  is:

$$P(t) = P_0 + v t$$

- NOTE: choice of unit is arbitrary as long as things are consistent, e.g. meters for distance, seconds for time, meters/second for velocity

## Varying Velocity

- The previous formula only works if the object moves with a constant velocity.
- In many cases, object's velocities change over time.
- The velocity is defined by the derivative:

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

- Constant velocity:  $v(t) = v_0$
- Velocity change over time by constant acceleration:  $v(t) = v_0 + a t$
- The displacement is a function that we integrate velocity

$$displacement = \int_0^t velocity dt$$

## Acceleration

- The acceleration is the rate of change in velocity.
- The acceleration is defined by the derivative

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

- Velocity is the integral of acceleration

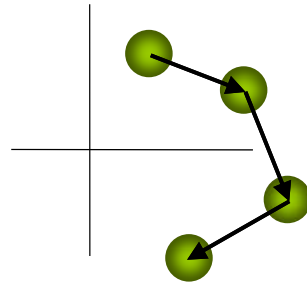
$$velocity = \int_0^t acceleration dt$$

## Euler Method (or Euler Integration)

- Euler method (or Euler Integration) approximates an integral by step-wise addition.
  - The most basic kind of explicit method for numerical integration of ordinary differential equations (ODE).
- At each time step, we move the object in a straight line using the current velocity:

$$dt = t_1 - t_0$$

$$P(t_1) = P(t_0) + v dt$$



## Euler Method (or Euler Integration)

- Applying Euler Integration to compute the position:

$$dt = t_1 - t_0;$$

$$Acc = \text{ComputeAcceleration}();$$

$$Vel = Vel + Acc * dt;$$

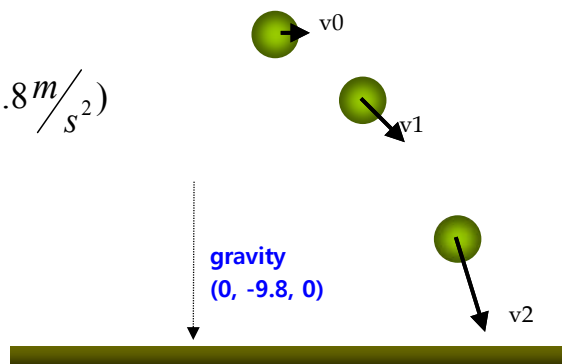
$$Pos = Pos + Vel * dt;$$

$$P(t) = \int_0^t (v_0 + at) dt = P_0 + v_0 t + \frac{1}{2} a t^2$$

## Gravity

- Gravity (지구 중력 가속도) near the Earth's surface produces a constant acceleration of  $9.8 \text{ meter/sec}^2$

$$F = mg \quad (g = -9.8 \text{ m/s}^2)$$



## Force

- Newton's second law of motion:

$$F = ma$$

$$\Rightarrow a = F / m$$

- If an object has mass M, and force F is applied to it, its motion can be calculated via Euler integration:

$$Acc = F/M;$$

$$Vel += Acc * dt;$$

$$Pos += Vel * dt;$$

Note that F, Acc, Vel, and Pos are all vectors. M is a scalar.

### 뉴턴 역학의 3법칙

- 관성의 법칙: 모든 물체는 다른 물체의 움직임의 영향을 받지 않는다고 할 때, 정지해 있었다면 계속 정지해 있을 것이고, 움직이고 있었다면 일정한 속도로 계속 운동할 것이다.
- 가속도의 법칙: 물체의 운동량의 변화율은, 크기와 방향에서, 그 물체에 작용하는 힘에 따른다.
- 작용, 반작용의 법칙: 모든 작용에는 그 반대방향으로 같은 크기의 반작용이 존재한다.

## Gravitational Force

- Gravitational force (중력)
  - The force of attraction between all masses in the universe; especially the attraction of the earth's mass for bodies near its surface.
  - The gravitation between two bodies is proportional to the product of their masses and inversely proportional to the square of the distance between them.

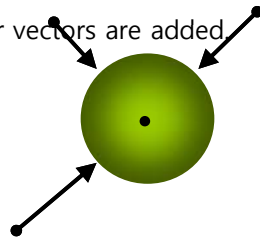
- For a complete simulation, we need to calculate the force on each object every frame.

- When multiple forces are applied, their vectors are added.

$$F_{gravity} = \frac{GM_1M_2}{d^2} \quad (G = 6.673 \times 10^{-11})$$

$M_1, M_2$  : mass(kg)

$d$  : distance (meter)



## Projectile Motion

- The projectile position,  $P_0$ , at  $t=0$  with the velocity  $v_0$ :

$$P(t) = P_0 + v_0t + \frac{1}{2}gt^2$$

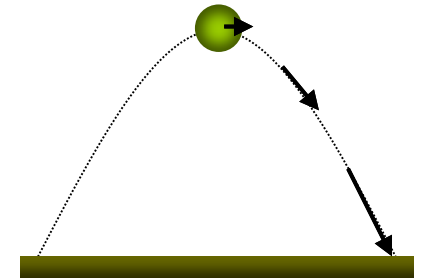
$$x(t) = x_0 + v_x t, \quad y(t) = y_0 + v_y t - \frac{1}{2}gt^2, \quad z(t) = z_0 + v_z t$$

- Time to reach the maximum height,  $t$ :

$$y(t) = v_y t - gt^2 = 0 \Rightarrow t = \frac{v_y}{g}$$

- Maximum height,  $h$ :

$$h = y_0 + \frac{v_y^2}{2g}$$



## Projectile Motion

- Maximum range,  $r$ :

$$y(t) = y_0 + v_y t - \frac{1}{2}gt^2 = y_0 \Rightarrow t=0 \text{ or } t = \frac{2v_y}{g}$$

$$x(t) = x_0 + v_x t \Rightarrow r = \frac{2v_x v_y}{g}$$

- Angle of elevation to reach the maximum height,  $\theta$ :

$$h = y_0 + \frac{v_z^2}{2g} \Rightarrow h = y_0 + \frac{(s \sin \theta)^2}{2g} \Rightarrow \theta = \sin^{-1} \left( \frac{1}{s} \sqrt{2g(h - y_0)} \right)$$

- Angle required to hit the target,  $\theta$ :

$$r = \frac{2v_x v_y}{g} \Rightarrow r = \frac{2(s \cos \theta)(s \sin \theta)}{g} = \frac{s^2}{g} \sin 2\theta \Rightarrow \theta = \frac{1}{2} \sin^{-1} \frac{rg}{s^2}$$

## Buoyancy

- Buoyancy force

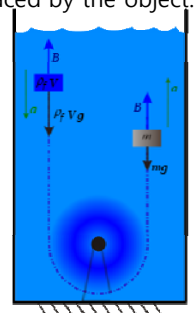
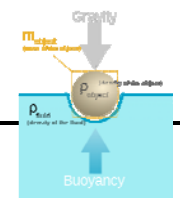
- Buoyancy is an upward acting force exerted by a fluid that oppose an object's weight.
- Archimedes' principle:
  - Any floating object displaces its own weight of fluid.
  - I.e., any object, wholly or partially immersed in a fluid, is buoyed up by a force equal to the weight of the fluid displaced by the object.

$$F_{buoyancy} = -\rho_f V g$$

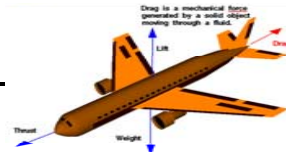
$\rho_f$  is the density of the fluid

$V$  is the volume of the displaced body of liquid

$g$  is the gravitational acceleration



## Drag



### □ Drag force

- In fluid dynamics, drag refers to forces that oppose the relative motion of an object through a fluid (a liquid or gas).

- Drag at low velocity (Stoke's drag):

$$F_d = -bv$$

$$b = 6\pi\eta r \quad (r : \text{small spherical object radius, } \eta : \text{viscosity})$$

- Drag at high velocity:

$$F_d = \frac{1}{2} \rho v^2 A C_d \frac{v}{\|v\|}$$

$F_d$  is the force vector of drag

$\rho$  is the density of the fluid

$v$  is the velocity of the object relative to the fluid

$A$  is the reference area

$C_d$  is the drag coefficient

## Kinetic Friction

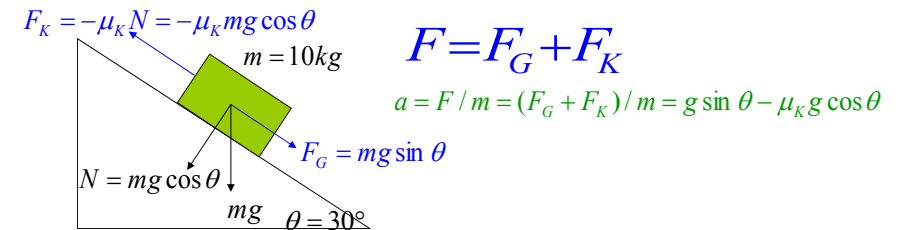
### □ Kinetic friction

- Kinetic (or dynamic) friction occurs when two objects are moving relative to each other and rub together (E.g., a sled on the ground).

$$F_K = -\mu_K N$$

$N$  is the normal force

$\mu_K$  is the coefficient of kinetic friction



## Static Friction

### □ Static friction

- Static friction is the friction between two solid objects that are not moving relative to each other (E.g., static friction can prevent an object from sliding down a sloped surface).

$$F_S = -\mu_S N$$

$N$  is the normal force

$\mu_S$  is the coefficient of static friction

- The maximum value of static friction,  $F_{\text{max}}$ , when motion is impending, is sometimes referred to as limiting friction.
- Any force larger than  $F_{\text{max}}$  overcomes the force of static friction and causes sliding to occur.

$$\mu_S mg \cos \theta = mg \sin \theta \Rightarrow \theta = \tan^{-1} \mu_S$$

## Momentum

- Momentum,  $P$ , is the product of the mass and velocity of an object.
- The rate of change of the momentum of a particle is proportional to the resultant force acting on the particle and is in the direction of that force.

$$P = mv$$

$$\Rightarrow \frac{dP}{dt} = m \frac{dv}{dt} = ma = F$$

Force = ComputeTotalForce();

Momentum += Force \* dt;

Velocity = Momentum / Mass;

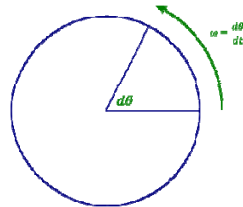
Position += Velocity \* dt;

## Angular Velocity

- Angular velocity (각속도) is the rate of change of angular displacement

- Angular velocity (radian/second):

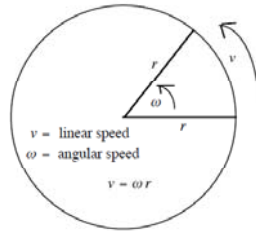
$$\omega(t) = \frac{d}{dt} \theta(t)$$



- Relationship between angular velocity,  $\omega$ , and linear velocity (선속도),  $v$

- Given a fixed speed  $v$  and radius  $r$ , then:

$$v(t) = \omega(t) \times r(t)$$



## Centrifugal Force

- Linear acceleration (선가속도)

$$a(t) = \omega'(t) \times r(t) + \omega(t) \times r'(t) \\ = \omega'(t) \times r(t) + \omega(t) \times [\omega(t) \times r(t)]$$

- If the angular velocity is constant:  $\omega'(t) = 0$

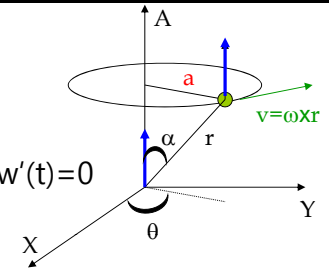
$$a(t) = \omega(t) \times [\omega(t) \times r(t)]$$

- Centrifugal force (원심력), equal and opposite to the tension (장력), drawing a rotating body away from the center of rotation.

$$F_c = -m(\omega(t) \times [\omega(t) \times r(t)])$$

- Centrifugal force (when  $r(t)$  and  $\omega(t)$  is perpendicular):

$$F_c = m\omega^2 r = \frac{mv^2}{r}$$



## Rigid Motion

- Rigid motion (강체운동)

- A rigid body is an idealization of **solid body (e.g. car)** of finite size in which deformation is neglected. (only translation & rotation possible)

- Rigid body dynamics (강체동역학)

- Linear & angular position, velocity, acceleration

Force = ComputeTotalForce();

Momentum += Force \* dt;

Velocity = Momentum / Mass;

Position += Velocity \* dt;

Torque = ComputeTotalTorque();

AngMomentum += Torque \* dt;

Matrix I = Matrix\*RotInertia\*Matrix.Inverse(); // tensor

AngVelocity = I.Inverse()\*AngMomentum;

Matrix.Rotate(AngVelocity\*dt);

## Integration Method

- Euler method

- $v = v_0 + a*dt$

- $P = P_0 + v*dt$

float t = 0; // 현재 시간

float dt = 1; // 시간 간격 (timestamp)

float velocity = 0; // 초기 속도

float position = 0; // 초기 위치

float force = 10;

float mass = 1;

float acceleration = force/mass;

while (t <= 10) {

position += velocity \* dt;

velocity += acceleration \* dt;

t += dt;

}

**Initial :**  $y'(t) = f(t, y(t)), y(t_0) = y_0$

**Euler Method :**  $y_{n+1} = y_n + hf(t_n, y_n)$

## Integration Method

### Runge-Kutta method

**Initial :**  $y' = f(t, y), y(t_0) = y_0$

**RK4:**  $y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

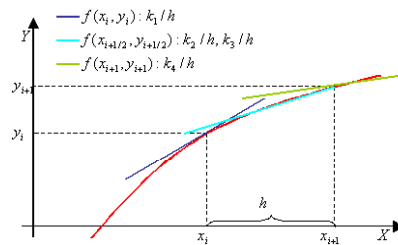
$k_1 = f(t_n, y_n)$

$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1)$

$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2)$

$k_4 = f(t_n + h, y_n + hk_3)$

$slope = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$



## Integration Method

```
void RK4Integration(vector3& pos, vector3& vel, float t, float dt) {
    vector3 k1Vel = vel;
    vector3 k1Acc = f(t, pos, vel);
    vector3 k2Vel = vel + 0.5f * dt * k1Acc;
    vector3 k2Acc = f(t + 0.5f * dt, pos + 0.5f * dt * k1Vel, k2Vel);
    vector3 k3Vel = vel + 0.5f * dt * k2Acc;
    vector3 k3Acc = f(t + 0.5f * dt, pos + 0.5f * dt * k2Vel, k3Vel);
    vector3 k4Vel = vel + dt * k3Acc;
    vector3 k4Acc = f(t + dt, pos + dt * k3Vel, k4Vel);
    pos += (dt / 6.0f) * (k1Vel + 2.0f * k2Vel + 2.0f * k3Vel + k4Vel);
    vel += (dt / 6.0f) * (k1Acc + 2.0f * k2Acc + 2.0f * k3Acc + k4Acc);
}
while (t <= 10) {
    RK4Integration(position, velocity, t, dt);
    t += dt;
}
```

## Springs

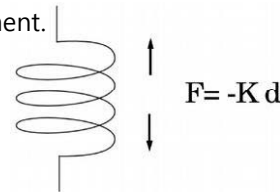
### Hooke's Law

- Spring force is proportional to displacement.

$$F = -K_s d$$

$K_s$  is the spring constant

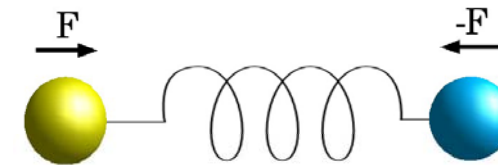
$d$  is the displacement from rest length



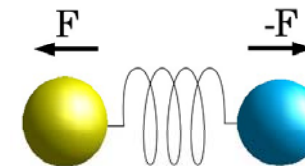
- Spring is modeled as two point masses, linked by the spring.
- Equal but opposite force is applied to each end.

## Springs

- When spring is stretched, spring force pulls masses together.



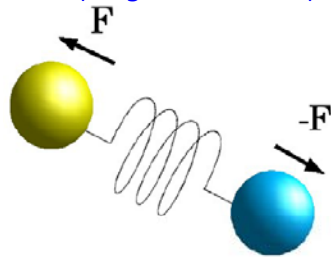
- When spring is compressed, spring force pushes masses apart.



## Springs

- Vector between the points is used to compute displacement and the direction of force:

```
Vector3 v = point1 - point0;  
float displacement = v.length() - restLength;  
v.normalize();  
Vector3 force = springConstant * displacement * v;
```



## Spring Classes

```
class PointMass  
{  
    float mass;  
    float position[3];  
    float velocity[3];  
    float acceleration[3];  
    void ClearForces();  
    void AddForce();  
    void Update();  
    void Freeze();  
}
```

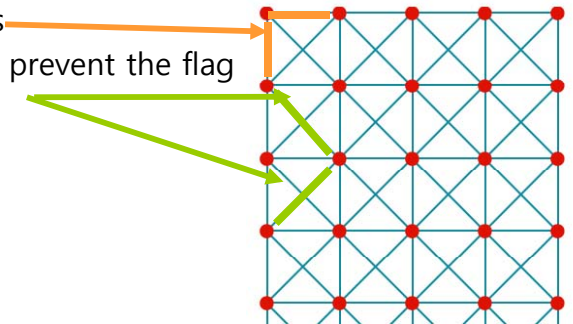
## Spring Classes

```
class Spring  
{  
    float pointMass[2];  
    float springConstant;  
    float restLength;  
    void Update();  
}
```



## Simulating Cloth

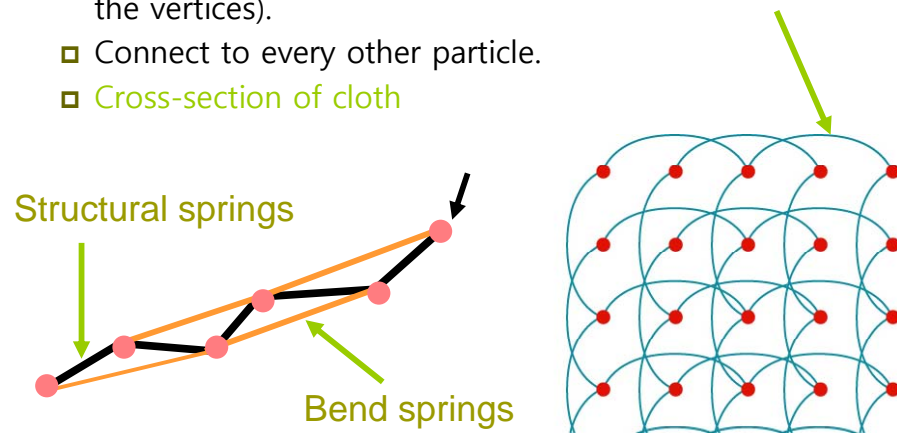
- Cloth can be simulated by a mesh of springs.
- Structural Springs
- Shear Springs (to prevent the flag from shearing)





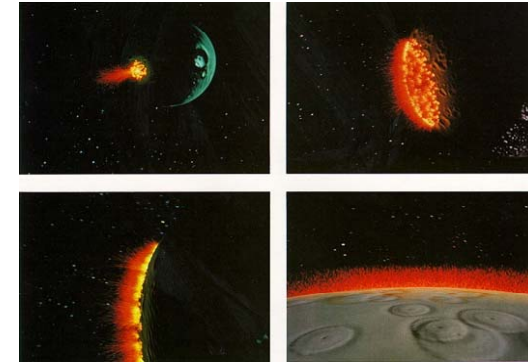
## Simulating Cloth

- Bend Springs (to prevent the flag from folding along the vertices).
- Connect to every other particle.
- Cross-section of cloth



## Particle Systems

- First used for graphics in Star Trek II (1983) "Genesis Effect"



## Particle Systems

- Particle systems simulate explosions, smoke, fire, spray.
- They are also useful for modeling non-rigid objects such as jelly or cloth (more later).
- Infinitely small objects that have **Mass, Position and Velocity**
- Motion of a Newtonian particle is governed by:
  - $F=ma$  ( $F$ =force,  $m$ =mass,  $a$ =acceleration)
  - $a=dv/dt$  (Change of velocity over time-  $v$ =velocity;  $t$ =time)
  - $v=dp/dt$  (Change of distance over time-  $p$ =distance or position)
  - So a basic data structure for a particle consists of:  $F, m, v, p$ .

## E.g. a 3D particle might be represented as:

- ```
class Particle
{
    float mass;
    float position[3]; // [3] for x,y,z components
    float velocity[3];
    float forceAccumulator[3];
}
```
- forceAccumulator is here because the particle may be acted upon by several forces- e.g. a soccerball is affected by the force of gravity and an external force like when someone kicks it. (see later)
  - Anything that will impart a force on the particle will simply ADD their 3 force components (force in X,Y,Z) to the forceAccumulator.

## E.g. 3D Particle System

---

```
class ParticleSystem
{
    particle *listOfParticles;
    int numParticles;
    void EulerStep();// Discussed later
}
```

## Particle Dynamics Algorithm

---

```
For each particle
{
    Compute the forces that are acting on the particle.
    Compute the acceleration of each particle:
        Since  $F=ma$ ;  $a=F/m$ 
    Compute velocity of each particle due to the
        acceleration.
    Compute the new position of the particle based on
        the velocity.
}
```

## How do you calculate velocity?

---

- Recall that:
  - $a = dv/dt$  (ie change in velocity over time)
  - $v = dp/dt$  (ie change in position over time)
- So to find velocity we need to find the integral of acceleration
- To find the position we need to find the integral of velocity
- A simple numerical integration method (**Euler's Method**):
  - $Q(t+dt) = Q(t) + dt * Q'(t)$
  - So in our case:
    - To find velocity at each simulation timestep:
      - $v(t+dt) = v(t) + dt * v'(t) = v(t) + dt * a(t)$  // we know  $a(t)$  from  $F=ma$
    - To find the position at each simulation timestep:
      - $p(t+dt) = p(t) + dt * p'(t) = p(t) + dt * v(t)$  // we know  $v(t)$

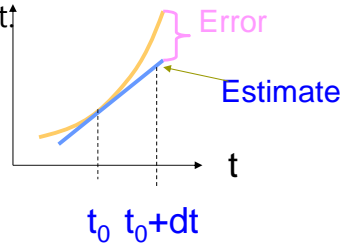
## E.g. Euler Integration EulerStep

---

- To find velocity at each simulation timestep:
  - $v(t+dt) = v(t) + dt * a(t)$  // we know  $a(t)$  from  $F=ma$
  - $v\_next[x] = v\_now[x] + dt * a[x];$
  - $v\_next[y] = v\_now[y] + dt * a[y];$
  - $v\_next[z] = v\_now[z] + dt * a[z];$
- To find the position at each simulation timestep:
  - $p(t+dt) = p(t) + dt * v(t)$  // we know  $v(t)$
  - $p\_next[x] = p\_now[x] + dt * v\_now[x];$
  - $p\_next[y] = p\_now[y] + dt * v\_now[y];$
  - $p\_next[z] = p\_now[z] + dt * v\_now[z];$
- Remember to save away  $v\_next$  for the next step through the simulation:
  - $v\_now[x] = v\_next[x]; v\_now[y] = v\_next[y]; v\_now[z] = v\_next[z];$

## Warning about Euler Method

- Big time steps causes big integration errors.
- You know this has happened because your particles go out of control and fly off into infinity!
- Use small time steps- but note that small time steps chew up a lot of CPU cycles.
- You do not necessarily have to *DRAW* every time step. E.g. compute 10  $\Delta t$  timesteps and then draw the result!
- There are other better solutions:
  - Adaptive Euler Method
  - Midpoint Method
  - Implicit Euler Method
  - Runge Kutta Method



## Adaptive Step Sizes

- Ideally we want the step-size ( $\Delta t$ ) to be as big as possible so we can do as few calculations as possible.
- But with bigger step sizes you incorporate more errors and your system can eventually destabilize.
- So small step sizes are usually needed. Unfortunately smaller step sizes can take a long time.
- You don't want to force a small step size all the time if possible.

## Euler with Adaptive Step Sizes

- Suppose you compute 2 estimates for the velocity at time  $t+\Delta t$ :
- So  $v_1$  is your velocity estimate for  $t+\Delta t$
- And  $v_2$  is your velocity estimate if you instead took 2 smaller steps of size  $\Delta t/2$  each.
- Both  $v_1$  and  $v_2$  differ from the true velocity by an order of  $\Delta t^2$  (because Euler's method is derived from Taylor's Theorem truncated after the 2nd term- see reference in the notes section of this slide)
- By that definition,  $v_1$  and  $v_2$  also differ from each other by an order of  $\Delta t^2$
- So we can write a measure of the current error as:  $E = |v_1 - v_2|$
- Let  $E_{\text{tolerated}}$  be the error that YOU can tolerate in your game.
- Adaptive step size  $\Delta t_{\text{adapt}}$  is calculated as approximately:
$$\Delta t_{\text{adapt}} = \text{Sqrt}(E_{\text{tolerated}} / E) * \Delta t$$
- So a bigger tolerated error would allow you to take a bigger step size. And a smaller one would force a smaller step size.

## Handling Collisions

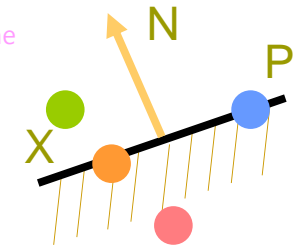
- Particles often bounce off surfaces.
  1. Need to detect when a collision has occurred.
  2. Need to determine the correct response to the collision.

## Detecting Collision

- General Collision problem is complex:
  - Particle/Plane Collision – we will look at this one coz it's easy way to start
  - Plane/Plane Collision
  - Edge/Plane Collision

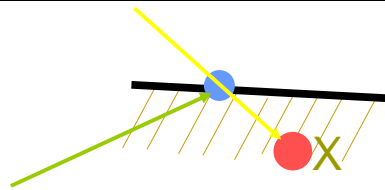
## Particle/Plane Collisions

- $P$ =any point on the plane
- $N$ =normal pointing on the "legal" side of the plane.
- $X$ =position of point we want to examine.
- For  $(X - P) \cdot N$ 
  - If  $> 0$  then  $X$  is on legal side of plane.
  - If  $= 0$  then  $X$  is on the plane.
  - If  $< 0$  then  $X$  is on the wrong side of plane



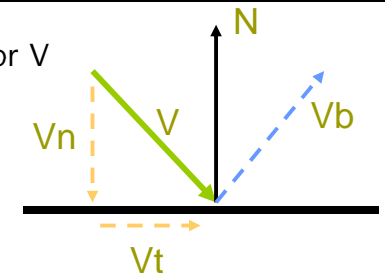
## Collision Response – dealing with the case where particle penetrates a plane (and it shouldn't have)

- If particle  $X$  is on the wrong side of the plane, move it to the surface of the plane and then compute its collision response.



## Collision Response

- $N$ =normal to the collision plane
- $V_n$ =normal component of a vector  $V$  is
 
$$V_n = (N \cdot V) N$$
- $V_t$ =tangential component is:
 
$$V_t = V - V_n$$
- $V_b$ =bounced response:
 
$$V_b = (1 - K_f) * V_t - (K_r * V_n)$$
- $K_r$ =coefficient of restitution: ie how bouncy the surface is. 1=perfectly elastic; 0=stick to wall.
- $K_f$ =coefficient of friction: ie how much the tangential vector is slowed down after the bounce. 1=particle stops in its tracks. 0=no friction.



## References

---

- <http://www.evl.uic.edu/spiff/class/cs426/Notes/physics.ppt>
- [http://en.wikipedia.org/wiki/Newton%27s\\_laws\\_of\\_motion](http://en.wikipedia.org/wiki/Newton%27s_laws_of_motion)
- [http://en.wikipedia.org/wiki/Equations\\_of\\_motion](http://en.wikipedia.org/wiki/Equations_of_motion)
- <http://en.wikipedia.org/wiki/Projectile>
- <http://en.wikipedia.org/wiki/Trajectory>
- <http://en.wikipedia.org/wiki/Buoyancy>
- [http://en.wikipedia.org/wiki/Drag\\_\(physics\)](http://en.wikipedia.org/wiki/Drag_(physics))
- [http://en.wikipedia.org/wiki/Euler\\_method](http://en.wikipedia.org/wiki/Euler_method)
- <http://en.wikipedia.org/wiki/RK4>
- <http://www.gaffer.org/game-physics/>