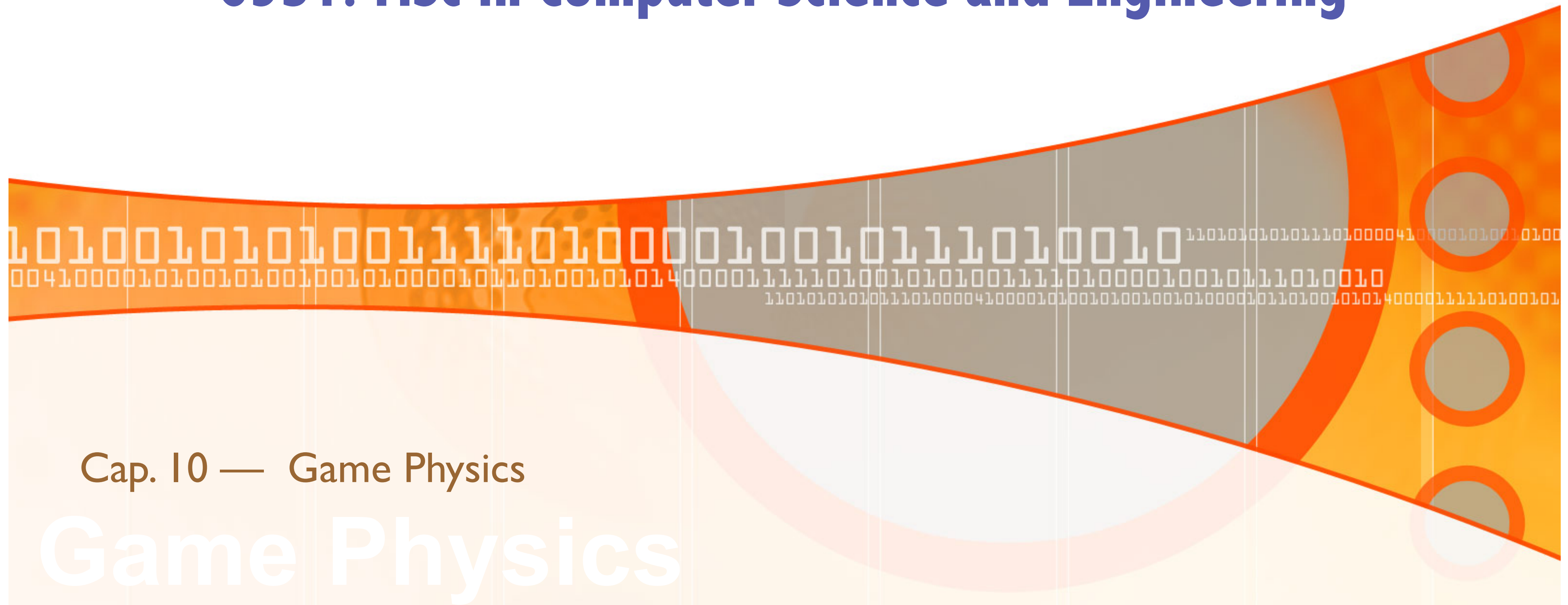
Some contents abusively taken from:
http://web.cs.wpi.edu/~imgd4000/d07
Author: Professor Mark Claypool

Video Game Technologies

6931: MSc in Computer Science and Engineering



Cap. 10 — Game Physics

Topics

- Introduction
- Point Masses
  - Projectile motion
  - Collision response
- Rigid-Bodies
  - Numerical simulation
  - Controlling truncation error
- Soft Body Dynamic System

http://en.wikipedia.org/wiki/Newton's_laws_of_motion

Physics is much about...

MOTION

Introduction

- Physics deals with motions of objects in virtual scene
  - and object interactions during collisions
- Physics increasingly (but only recently, last 3 years?) important for games
  - Similar to advanced AI, advanced graphics
- Enabled by more processing
  - Used to need it all for more core gameplay (graphics, I/O, AI)
  - Now have additional processing for more
    - Duo-core processors
    - Physics hardware (Ageia's Physx) and general GPU (instead of graphics)
    - Physics libraries (Havok FX) that are optimized
- Potential
  - New gameplay elements : Realism (i.e., gravity, water resistance, etc.)
  - Particle effects : Improved collision detection
  - Rag doll physics : Realistic motion

- Physics engine can be part of a game engine
- License middleware physics engine
  - Complete solution from day 1
  - Proven, robust code base (in theory)
  - Features are always a tradeoff
- Build physics engine in-house
  - Choose only the features you need
  - Opportunity for more game-specific optimizations
  - Greater opportunity to innovate
  - Cost can be easily be much greater

Newton's Three Laws of Motion (1 of 3)

- 1st Law (Law of Inertia)
  - Every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it.
- 2nd Law
  - The relationship between an object's mass *m*, its acceleration a, and the applied force *F* is

$$\mathbf{F} = m\mathbf{a}\,.$$

  - The acceleration ***a*** of a body is proportional to the resultant force ***F*** acting on the body and is in the same direction as the resultant force; in this law the direction of the force vector is the same as the direction of the acceleration vector.
- 3rd Law
  - For every action there is an equal and opposite reaction.

Newton's Three Laws of Motion (2 of 3)

- Generally, object does not come to a stop naturally, but forces must bring it to stop
  - Force can be friction (i.e., ground)
  - Force can be drag (i.e., air or fluid)
- Forces types: gravitational, electromagnetic, weak nuclear, strong nuclear
  - But gravitational most common in games (and most well-known)
- From dynamics:
  - Force = mass x acceleration: $\mathbf{F} = m\mathbf{a}$
- In games, forces often known, so need to calculate acceleration ($\mathbf{a} = \mathbf{F}/m$)
- Acceleration used to update velocity and velocity used to update objects position:
  - $x = x + (v + a * t) * t$ (t is the delta time)
  - Can do for (x, y, z) positions
  - (speed is just magnitude, or size, of velocity vector)
- So, if add up all forces on object and divide by mass to get acceleration

- *Kinematics* is study of motion of bodies and forces acting upon bodies
- Three bodies:
  - *Point masses* – no angles, so only linear motion (considered infinitely small)
    - Particle effects
  - *Rigid bodies* – shapes do not change, so deals with angular (orientation) and linear motion
    - Characters and dynamic game objects
  - *Soft bodies* – have position and orientation and can change shape (i.e., cloth, liquids)
    - Starting to be possible in real-time

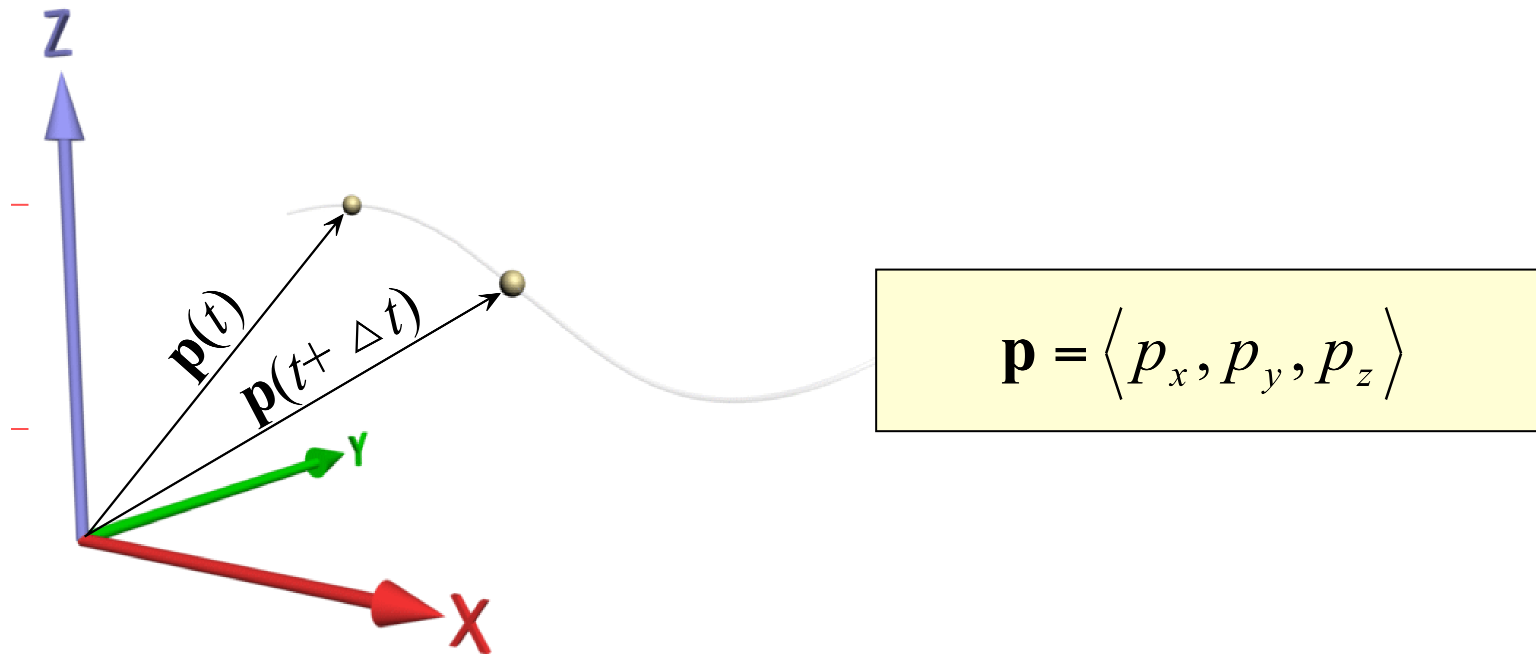
Topics

- Introduction
- Point Masses . . . . next!
  - Projectile motion
  - Collision response
- Rigid-Bodies
  - Numerical simulation
  - Controlling truncation error
- Soft Body Dynamic System

- What is a Particle?
  - A sphere of finite radius with a perfectly smooth, frictionless surface
  - Experiences no rotational motion
- Particle kinematics
  - Defines the basic properties of particle motion
  - Position, Velocity, Acceleration

Particle Kinematics: *Position*

- Location of Particle in World Space (units are meters (m))



$$\mathbf{p} = \langle p_x, p_y, p_z \rangle$$

  - Changes over time when object moves

Tip! Make sure consistent units used by all developers!

Particle Kinematics: *Velocity* and *Acceleration*

- Average velocity (units: meters/sec):
  - [p(t+Δt) - p(t)] / Δt
  - But velocity may change in time Δt
- Instantaneous velocity is derivative of position:

$$\mathbf{V}(t) = \lim_{\Delta t \to 0} \frac{\mathbf{p}(t+\Delta t) - \mathbf{p}(t)}{\Delta t} = \frac{d}{dt}\mathbf{p}(t)$$

(Position is the integral of velocity over time)

- Acceleration (units: m/s^2)
  - First-order derivative of velocity
  - Second-order derivative of position

$$\mathbf{a}(t) = \frac{d}{dt}\mathbf{V}(t) = \frac{d^2}{dt^2}\mathbf{p}(t)$$

Newton's 2nd Law of Motion

- Paraphrased – "An object's change in velocity is proportional to an applied force"
- The Classic Equation:

$$\mathbf{F}(t) = m\mathbf{a}(t)$$

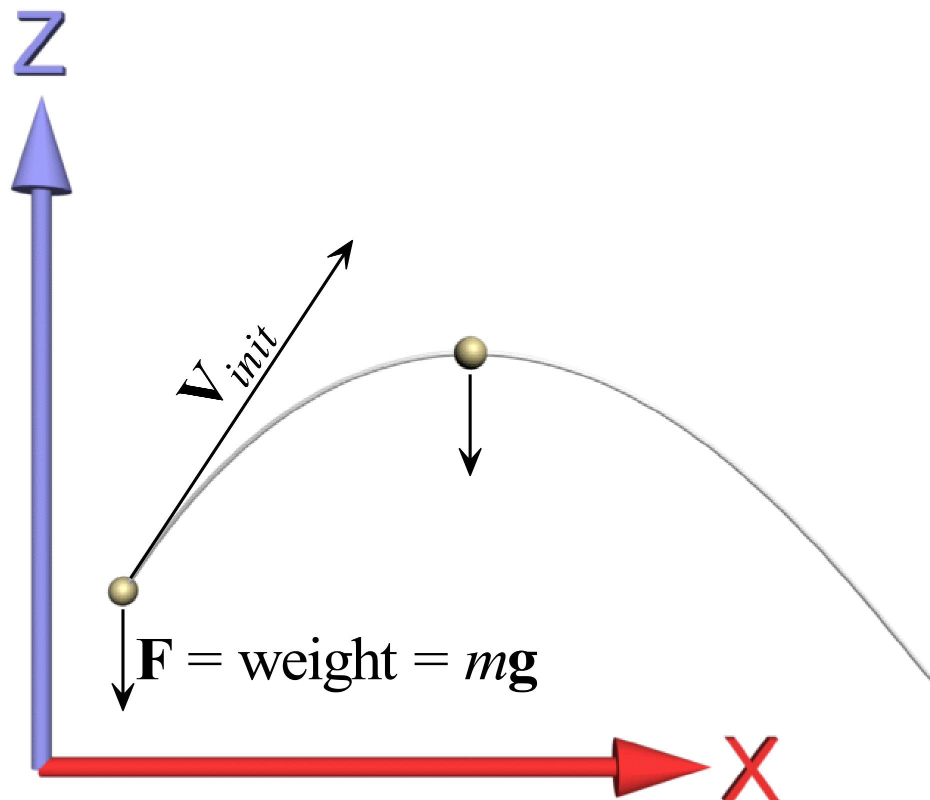
  - m = mass (units: kilograms, kg)
  - $\mathbf{F}(t)$ = force (units: Newtons)

- The Cycle of Motion:
  - Force, $\mathbf{F}(t)$, causes acceleration
  - Acceleration, $\mathbf{a}(t)$, causes a change in velocity
  - Velocity, $\mathbf{V}(t)$ causes a change in position
- Physics Simulation:
  - Solving variations of the above equations over time
  - Use to get positions of objects
  - Render objects on screen
  - Repeat to emulate the cycle of motion

Topics

- Introduction
- Point Masses
  - Projectile motion . . . . **next**!
  - Collision response
- Rigid-Bodies
  - Numerical simulation
  - Controlling truncation error
- Soft Body Dynamic System

Example: 3D Projectile Motion (1 of 3)



Basis for entire game!

- Eagle eye: http://www.teagames.com/games/eagleeye/play.php
  - Basic arrow projectile
- Fortress Fight: http://www.nick.com/games/nick_games/avatar/av_fortress.jhtml
  - Basic castle battle
- Castle Battle: http://www.freeonlinegames.com/play/1618.html
  - 3d perspective, physics on blocks

Example: 3D Projectile Motion (1 of 3)

- Constant Force (i.e., gravity)
  - Force is *weight* of the projectile, $\mathbf{W} = m\mathbf{g}$
  - g is constant acceleration due to gravity
    - On earth, gravity (g) is 9.81 m/s^2
- With constant force, acceleration is constant
- Easy to integrate to get closed form
- Closed-form "Projectile Equations of Motion":

$$\mathbf{V}(t) = \mathbf{V}_{init} + \mathbf{g}(t - t_{init})$$

$$\mathbf{p}(t) = \mathbf{p}_{init} + \mathbf{V}_{init}(t - t_{init}) + \frac{1}{2}\mathbf{g}(t - t_{init})^2$$

  - These closed-form equations are valid, and *exact**, for any time, t, in seconds, greater than or equal to t_{init} (Note, requires constant force)

Example: 3D Projectile Motion (2 of 3)

- For simulation:
  - Begins at time t_{init}
  - Initial velocity, $\mathbf{V}_{init}$ and position, $\mathbf{p}_{init}$, at time t_{init}, are known
  - Can find later values (at time t) based on initial values
- On Earth:
  - If we choose positive Z to be straight up (away from center of Earth), g_{Earth} = 9.81 m/s^2:

$$\mathbf{g}_{Earth} = -g_{Earth}\hat{k} = \langle 0.0, 0.0, -9.81 \rangle \text{ m/s}^2$$

Note: the Moon's gravity is about 1/6th that of Earth

Pseudo-code for Simulating Projectile Motion

```
void main()
{
    // Initialize variables
    Vector v_init(10.0, 0.0, 10.0);
    Vector p_init(0.0, 0.0, 100.0), p = p_init;
    Vector g(0.0, 0.0, -9.81);    // earth
    float t_init = 10.0;          // launch at time 10 seconds

    // The game sim/rendering loop
    while (1)
    {
      float t = getCurrentGameTime();                // could use system clock
      if (t > t_init) {
         float t_delta = t - t_init;
         p = p_init + (v_init * t_delta);            // velocity
         p = p + 0.5 * g * (t_delta * t_delta);      // acceleration
      }
      renderParticle(p);                     // render particle at location p
    }
}
```

Topics

- Introduction
- Point Masses
  - Projectile motion
  - Collision response . . . . **next**!
- Rigid-Bodies
  - Numerical simulation
  - Controlling truncation error
  - Generalized translation motion
- Soft Body Dynamic System
- Collision Detection

Linear Momentum

http://www.sparknotes.com/testprep/books/sat2/physics/chapter9.rhtml

- The concept of **linear momentum** is closely tied to the concept of force—in fact, Newton first defined his Second Law not in terms of mass and acceleration, but in terms of momentum.
- ***Linear momentum*** – is the mass times the velocity

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

  - Note that a body's momentum is always in the same direction as its velocity vector. The units of momentum are kg · m/s.
  - On the whole, it is useful to analyze systems in terms of energy when there is an exchange of potential energy and kinetic energy.
- Linear momentum, however, is useful in those cases where there is no clear measure for potential energy. In particular, we will use the law of **conservation of momentum** to determine the outcome of collisions between two bodies.
- Note that the word *momentum* in everyday life is consistent with the definition of momentum in physics. For example, we say that a BMW driving 20 miles per hour has less momentum than the same car speeding on the highway at 80 miles per hour. Also, if a large truck and a BMW travel at the same speed on a highway, the truck has a greater momentum than the BMW, because the truck has greater mass. Our everyday usage reflects the definition given above, that momentum is proportional to mass and velocity.

Linear Momentum and Newton's Second Law

- Using the concept of momentum, Newton's second can be expressed as follows:

$$\mathbf{F} = \frac{d\mathbf{M}}{dt}$$

- This formula is more flexible than $\boldsymbol{F} = m\boldsymbol{a}$ because it can be used to analyze systems where not just the velocity, but also the mass of a body changes, as in the case of a rocket burning fuel.
- Most objects have constant mass, so:

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

  - Called the *Newtonian Equation of Motion*
    - Since when integrated over time it determines the motion of an object

Impulse

- Impulse is a vector quantity defined as the product of the force acting on a body and the time interval during which the force is exerted.
- The impulse caused by a force during a specific time interval is equal to the body's change of momentum during that time interval: impulse, effectively, is a measure of change in momentum.
- Thus, by replacing the impulse **I** by the change of moment *d***M** in the Newton's Second Law, we have:

$$\mathbf{I} = \mathbf{F}dt$$

- ***Example***: A soccer player kicks a 0.1 kg ball that is initially at rest so that it moves with a velocity of 20 m/s. What is the impulse the player imparts to the ball? If the player's foot was in contact with the ball for 0.01 s, what was the force exerted by the player's foot on the ball?
  - Since impulse is simply the change in momentum, we need to calculate the difference between the ball's initial momentum and its final momentum. Since the ball begins at rest, its initial velocity, and hence its initial momentum, is zero. Its final momentum is:

$$M = m.v = 0.1 \times 20 = 2 \text{ kg} \cdot \text{m/s}$$

  - Because the initial momentum is zero, the ball's change in momentum, and hence its impulse, is 2 kg · m/s.

Frictionless Collision Response (1 of 3)

- Consider two colliding particles
- For the duration of the collision, both particles exert force on each other
  - Normally, collision duration is very short, yet change in velocity is dramatic (e.g., pool balls)
- Integrate previous equation over duration of collision

$$m_1 v_1^+ = m_1 v_1^- + I \qquad \text{(equation 1)}$$

  - $m_1 v_1^-$ is linear momentum of first particle just before collision
  - $m_1 v_1^+$ is the linear momentum just after collision
  - I is the linear impulse
    - Integral of collision force over duration of collision

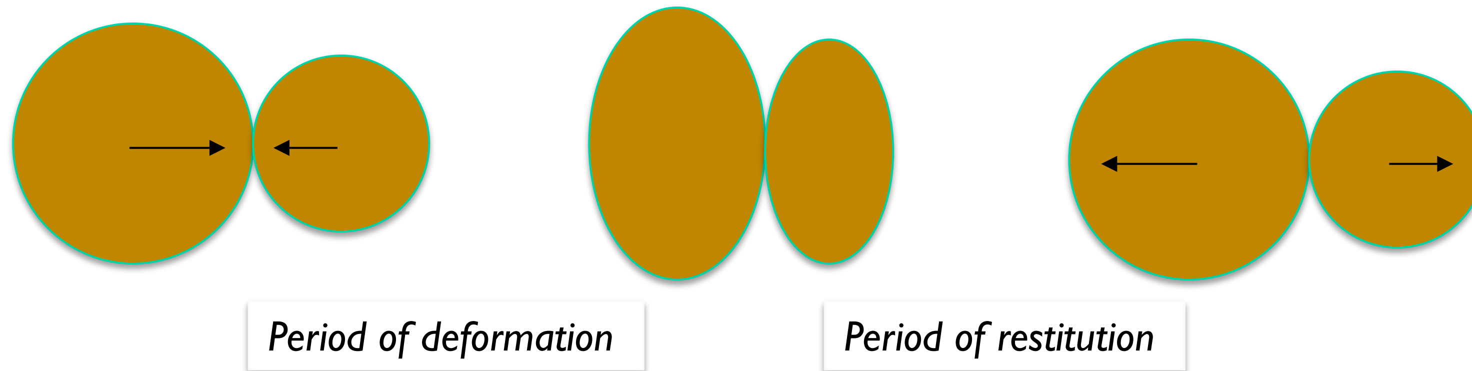
Frictionless Collision Response (2 of 3)

- Newton's third law of motion says for every action, there is an equal and opposite reaction
  - So, particle 2 is the same magnitude, but opposite in direction (so, -**I**)
- Can solve these equations if know **I**
- Without friction, impulse force acts completely along unit surface normal vector at point of contact

$$\mathbf{I} = i\mathbf{n} \quad \text{(equation 2)}$$

  - **n** is the unit surface normal vector (see collision detection for point of contact)
  - *i* is the scalar value of the impulse
    - In physics, *scalar* is simple physical quantity that does not depend on direction
- So, have 2 equations with three unknowns (v_1^+, v_2^+, i).
  - Need third equation to solve for all

Frictionless Collision Response (3 of 3)



- Third equation is approximation of material response to colliding objects:

$$(v_1^+ - v_2^+)\mathbf{n} = -\varepsilon(v_1^- - v_2^-)\mathbf{n} \qquad \text{(equation 3)}$$

  - Note, in general, can collide at angle
  - ε is coefficient of restitution
    - Related to conservation or loss of kinetic energy
    - ε is 1, totally elastic, so objects rebound fully
    - ε is 0, totally plastic, objects no restitution, maximum loss of energy
    - In real life, depends upon materials
      - Ex: tennis ball on raquet, ε is 0.85 and deflated basketball with court ε is 0)
      - (Next slides have details)

Coefficient of Restitution (1 of 5)

- A measure of the elasticity of the collision
  - How much of the kinetic energy of the colliding objects before collision remains as kinetic energy after collision
- Links:
  - Basic Overview : Wiki
  - The Physics Factbook : Physics of Baseball and Softball Bats
  - Measurements of Sports Balls
- Definition: ratio of the differences in velocities before & after collision

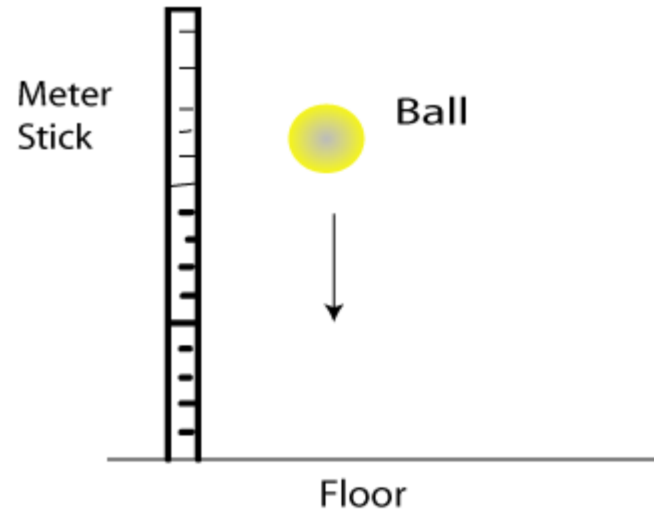
$$\varepsilon = (v_1^+ - v_2^+)/(v_1^- - v_2^-)$$

- For an object hitting an immovable object (i.e., the floor)

$$\varepsilon = \sqrt{h/H}$$

  - where h is bounce height, H is drop height

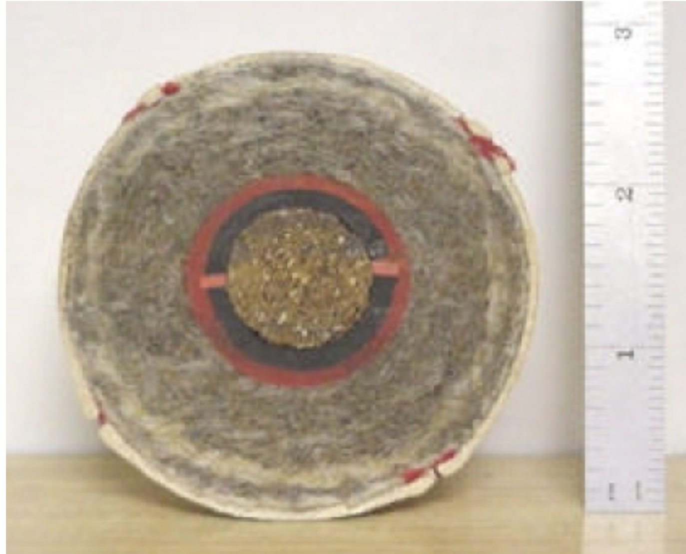
Coefficient of Restitution (2 of 5)



- *Drop ball from fixed height (92 cm)*
- *Record bounce*
- *Repeat 5 times and average*
- *Various balls*

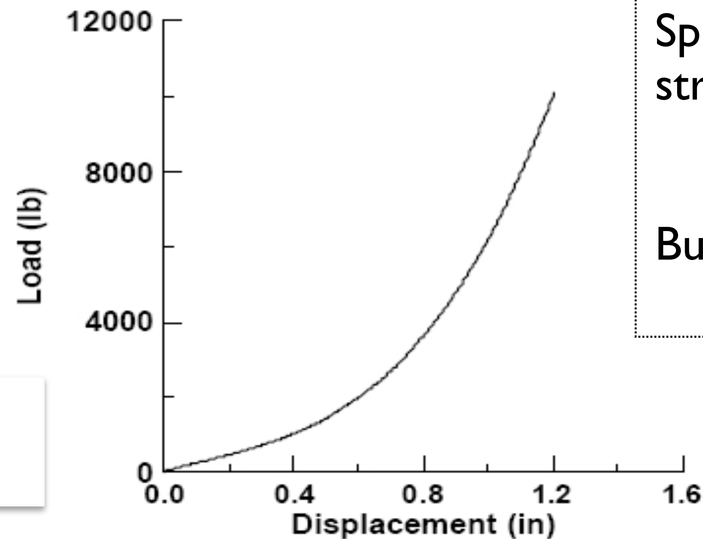
object	H (cm)	h_1 (cm)	h_2 (cm)	h_3 (cm)	h_4 (cm)	h_5 (cm)	h_{ave} (cm)	c.o.r.
range golf ball	92	67	66	68	68	70	67.8	0.858
tennis ball	92	47	46	45	48	47	46.6	0.712
billiard ball	92	60	55	61	59	62	59.4	0.804
hand ball	92	51	51	52	53	53	52.0	0.752
wooden ball	92	31	38	36	32	30	33.4	0.603
steel ball bearing	92	32	33	34	32	33	32.8	0.597
glass marble	92	37	40	43	39	40	39.8	0.658
ball of rubber bands	92	62	63	64	62	64	63.0	0.828
hollow, hard plastic ball	92	47	44	43	42	42	43.6	0.688

Coefficient of Restitution (3 of 5)



- Layers:
  - Cork and rubber (like a superball)
  - Tightly round yarn
  - Thin tweed
  - Leather
- (Softball simpler – just cork and rubber with leather)

More force needed to compress, sort of like a spring



Spring would be straight line:

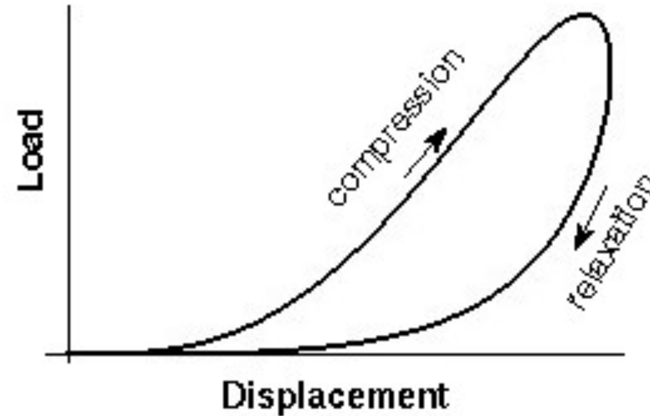
$F = k\ x$

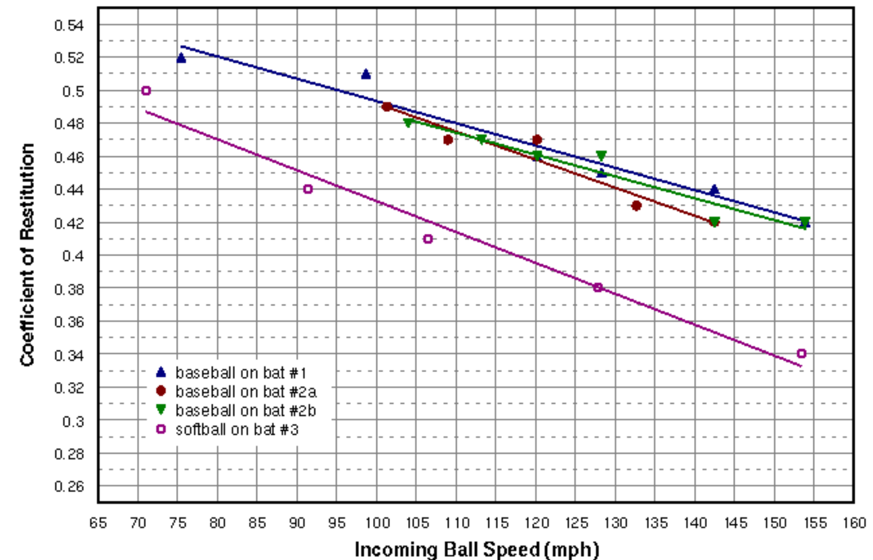
But is:

$F = k\ x^p$

Coefficient of Restitution (4 of 5)

- Plus, force-compression curve not symmetric
  - Takes more time to expand than compress
  - Meaning, for $F = kx^p$, p different during relaxation
- Area inside curve is energy that is lost to internal friction
- Coefficient of restitution depends upon speed
  - Makes it even more complicated





- Last notes …
- Technically
  - COR a property of a collision, not necessarily an object
    - 5 different types of objects → 10 (5 choose 2 = 10) different CORs
  - May be energy lost to internal friction (baseball)
  - May depend upon speed
  - All that can get complicated!
- But, for properties not available, can estimate
  - (ie- rock off of helmet, dodge ball off wall)
  - Playtest until looks “right”

Putting It All Together

- We have 3 equations and 3 unknowns ($\mathbf{v}_1^+$, $\mathbf{v}_2^+$, i)
- We can then compute the linear impulse

$$\mathbf{I} = -\frac{m_1 m_2 (1+\varepsilon)(\mathbf{v}_1^- - \mathbf{v}_2^-) \bullet \mathbf{n}}{m_1 + m_2}\mathbf{n} \qquad \text{(equation 4)}$$

- We can then apply $\mathbf{I}$ to previous equations:
  - Equation 1 to get $\mathbf{v}_1^+$ (and similarly $\mathbf{v}_2^+$)
- … and divide by m_1 (or m_2) to get after-collision velocities

Pseudocode (1 of 2)

```
void main() {
    // initialize variables
    vector v_init[N] = initial velocities;
    vector p_init[N] = initial positions;
    vector g(0.0, 0.0, -9.81); // earth
    float mass[N] = particle masses;
    float time_init[N] = start times;
    float eps = coefficient of restitution;
    // main game simulation loop
    while (1) {
          float t = getCurrentGameTime();
          detect collisions (t_collide is time);
          for each colliding pair (i,j) {
               // calc position and velocity of i
               float telapsed = t_collide - time_init[i];
               pi = p_init[i] + (V_init[i] * telapsed); // velocity
               pi = pi + 0.5*g*(telapsed*telapsed);      // accel
               // calc position and velocity of j
               float telapsed = tcollide - time_init[j];
               pj = p_init[j] + (V_init[j] * telapsed); // velocity
               pj = pj + 0.5*g*(telapsed*telapsed);      // accel
               // for spherical particles, surface normal is vector joining middle
               normal = Normalize(pj - pi);
               // compute impulse (equation 4)
               impulse = normal;
               impulse *= -(1+eps)*mass[i]*mass[j];
               impulse *=normal.DotProduct(vi-vj); //Vi1Vj1+Vi2Vj2+Vi3Vj3
               impulse /= (mass[i] + mass[j]);
               // Restart particles i and j after collision (eq 1); Since collision is
               // instant, after-collisions positions are the same as before
               V_init[i] += impulse/mass[i];
               V_init[j] -= impulse/mass[j];  // equal and opposite
               p_init[i] = pi;
               p_init[j] = pj;
               // reset start times since new init V
               time_init[i] = t_collide;
               time_init[j] = t_collide;
          } // end of for each
```

Pseudocode (1 of 2)

```
void main() {
    // initialize variables
    . . .
    // main game simulation loop
    while (1) {
          float t = getCurrentGameTime();
          detect collisions (t_collide is time);
          for each colliding pair (i,j) {
               . . .
          } // end of for each

          // Update and render particles
          for k = 0; k<N; k++){
               float tm = t - time_init[k];
               p = p_init[k] + V_init[k] + tm; //velocity
               p = p + 0.5*g*(tm*tm); // acceleration

               render particle k at location p;
          }
    }
```

Topics

- Introduction
- Point Masses
  - Projectile motion
  - Collision response
- Rigid-Bodies . . . . next!
  - Numerical simulation
  - Controlling truncation error
- Soft Body Dynamic System

Rigid-Body Simulation Intro

- If no rotation, only gravity and occasional frictionless collision, above is fine
- In many games (and life!), interesting motion involves non-constant forces and collision impulse forces
- Unfortunately, for the general case, often no closed-form solutions
- *Numerical simulation:*

> ***Numerical Simulation*** represents a series of techniques for incrementally solving the equations of motion when forces applied to an object are not constant, or when otherwise there is no closed-form solution

Numerical Integration of Newton's Equations: Finite Difference Method

Taylor series: $$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n.$$

- The first step in applying various numerical schemes that emanate from Euler method is to write Newton's equations of motion as two coupled first-order differential equations

$$a(t) = \frac{dv(t)}{dt} \qquad v(t) = \frac{dx(t)}{dt}$$

- We let be the time interval between successive time steps and , Δt , and a_n, v_n, and x_n be the values of acceleration a, velocity v, and particle position x at time $t_n = t_0 + n\Delta t$
- The goal of *finite difference methods* is to determine the value of x_{n+1} and v_{n+1} at time $t_{n+1} = t_n + \Delta t$.
- The nature of many integration algorithms can be understood by expanding $v_{n+1} = v(t_n + \Delta t)$ and $x_{n+1} = x(t_n + \Delta t)$ in a Taylor series:

$$v_{n+1} = v_n + a_n \Delta t + \mathbf{O}[(\Delta t)^2]$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + \mathbf{O}[(\Delta t)^3]$$

http://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/ode.html

Pseudo Code for Numerical Integration

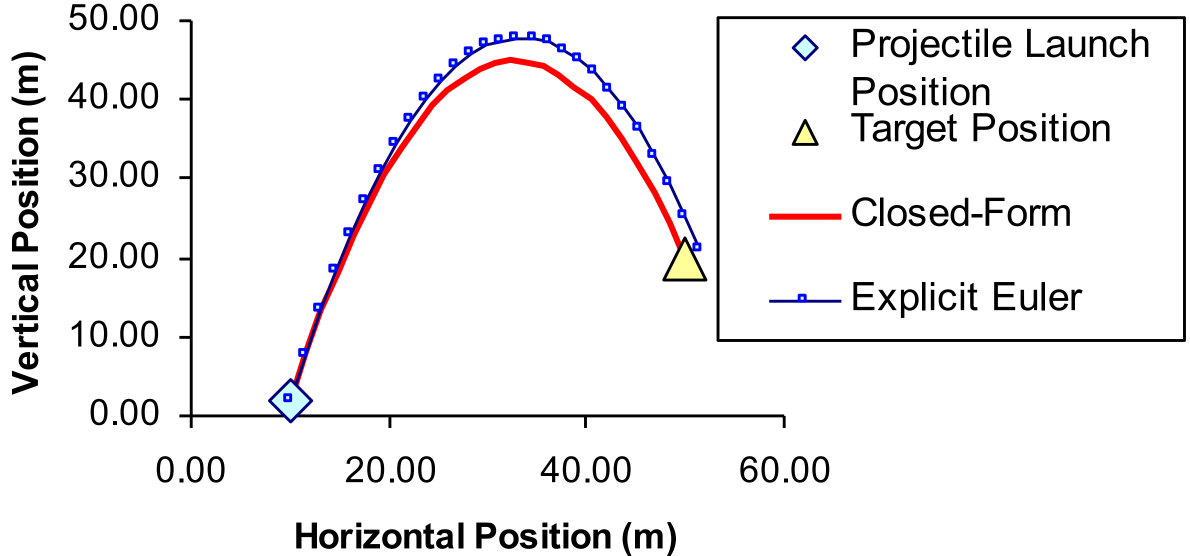
```
Vector cur_S[2*N];      // S(t+Δt)
Vector prior_S[2*N];    // S(t)
Vector S_deriv[2*N];    // d/dt S at time t
float mass[N];          // mass of particles
float t;                // simulation time t

void main()
{
    float delta_t;      // time step

    // set current state to initial conditions
    for (i=0; i<N; i++) {
       mass[i] = mass of particle i;
       cur_S[2*i] = particle i initial momentum;
       cur_S[2*i+1] = particle i initial position;
    }

    // Game simulation/rendering loop
    while (1)
    {
       doPhysicsSimulationStep(delta_t);
       for (i=0; i<N; i++) {
        render particle i at position cur_S[2*i+1];
    }
}
```

```
// update physics
void doPhysicsSimulationStep(delta_t)
{
    copy cur_S to prior_S;

    // calculate state derivative vector
    for (i=0; i<N; i++)
    {
       // could be just gravity
       S_deriv[2*i] = CalcForce(i);
       // since S[2*i] is mV → divide by m
       S_deriv[2*i+1] = prior_S[2*i]/mass[i];
    }

    // integrate equations of motion
    ExplicitEuler(2*N, cur_S, prior_S,
    S_deriv, delta_t);

    // by integrating, effectively moved
   // simulation time forward by delta_t
    t = t + delta_t;
}
```

Explicit Euler Integration: Computing Solution Over Time

The solution proceeds step-by-step, each time integrating from the prior state

	Position (m)			Linear Momentum (kg-m/s)			Force (N)			Velocity (m/s)		
Time	p_x	p_y	p_z	mV_x	mV_y	mV_z	F_x	F_y	F_z	V_x	V_y	V_z
5.00	10.00	0.00	2.00	19.20	0.00	72.50	0.00	0.00	-24.53	7.68	0.00	29.00
5.20	11.54	0.00	7.80	19.20	0.00	67.60	0.00	0.00	-24.53	7.68	0.00	27.04
5.40	13.07	0.00	13.21	19.20	0.00	62.69	0.00	0.00	-24.53	7.68	0.00	25.08
5.60	14.61	0.00	18.22	19.20	0.00	57.79	0.00	0.00	-24.53	7.68	0.00	23.11
⋮		⋮			⋮			⋮			⋮	
10.40	51.48	0.00	20.87	19.20	0.00	-59.93	0.00	0.00	-24.53	7.68	0.00	-23.97



Truncation Error

- Numerical solution can be different from exact, closed-form solution
  - Difference between exact solution and numerical solution is primarily *truncation error*
    - Equal and opposite to value of terms removed from Taylor Series expansion to produce finite difference equation

- Truncation error, left unchecked, can accumulate to cause simulation to become unstable
  - This ultimately produces floating point overflow
  - Unstable simulations behave unpredictably

- Sometimes, truncation error can become zero
  - In other words, finite difference equation produces exact, correct result
  - For example, when zero force is applied

- But, more often truncation error is nonzero. Control by:
  - Reduce time step, Δt (Next slide)
  - Select a different numerical integrator (Vertlet and others, not covered). Typically, more state kept. Stable within bounds.

- Euler-Cromer Method
  - http://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node2.html
- Midpoint and Half-Step Methods
  - http://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node3.html
- Euler-Richardson Method
  - http://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node4.html
- Verlet Method
  - http://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node5.html

Summary:

- Introduction
- Point Masses
  - Projectile motion
  - Collision response
- Rigid-Bodies
  - Numerical simulation
  - Controlling truncation error
- Soft Body Dynamic System

http://en.wikipedia.org/wiki/Newton's_laws_of_motion