



Physics for Games

IMGD 4000



Topics

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



Introduction (1 of 2)


- 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



Introduction (2 of 2)

- Potential
 - New gameplay elements
 - Realism (ie- gravity, water resistance, etc.)
 - Particle effects
 - Improved collision detection
 - Rag doll physics
 - Realistic motion





Physics Engine - Build or Buy?

- 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



Newtonian Physics (1 of 3)

- Sir Isaac Newton (around 1700) described three laws, as basis for *classical mechanics*:
 1. A body will remain at rest or continue to move in a straight line at a constant velocity unless acted upon by another force
 - (So, Atari *Breakout* had realistic physics! ☺)
 2. The acceleration of a body is proportional to the resultant force acting on the body and is in the same direction as the resultant force.
 3. For every action, there is an equal and opposite reaction
- More recent physics show laws break down when trying to describe universe (Einstein), but good for computer games



Newtonian Physics (2 of 3)

- Generally, object does not come to a stop naturally, but forces must bring it to stop
 - Force can be friction (ie- ground)
 - Force can be drag (ie- air or fluid)
- Forces: gravitational, electromagnetic, weak nuclear, strong nuclear
 - But gravitational most common in games (and most well-known)
- From dynamics:
 - Force = mass x acceleration ($F=ma$)
- In games, forces often known, so need to calculate acceleration
$$a = F/m$$
- 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



Newtonian Physics (3 of 3)

- *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 to 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 (ie- 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
 - Generalized translation motion
- Soft Body Dynamic System
- Collision Detection



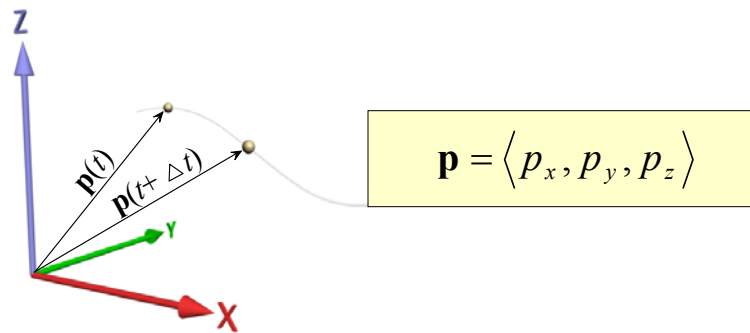
Point-Mass (Particle) Physics

- 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))



- 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):
 - $[\mathbf{p}(t + \Delta t) - \mathbf{p}(t)] / \Delta t$
 - But velocity may change in time Δt
- Instantaneous velocity is derivative of position:

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

(Position is the integral of velocity over time)

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



What is Physics Simulation?

- 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

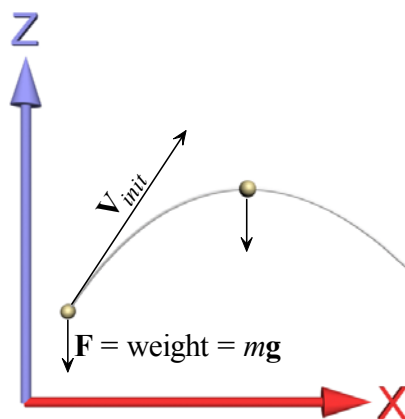


Topics

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



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.html
 - 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 (ie- gravity)
 - Force is *weight* of the projectile, $W = mg$
 - 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), $g_{Earth} = 9.81 \text{ m/s}^2$:

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

Note: the Moon 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



Frictionless Collision Response (1 of 4)

- *Linear momentum* - is the mass times the velocity
momentum = mV
 - (units are kilogram-meters per second)
- Related to the force being applied
 - 1st time derivative of linear momentum is equal to net force applied to object
$$d/dt (mV(t)) = F(t)$$
- Most objects have constant mass, so:
$$d/dt (mV(t)) = m d/dt (V(t))$$
 - Called the *Newtonian Equation of Motion*
 - Since when integrated over time it determines the motion of an object



Frictionless Collision Response (2 of 4)

- 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 (ex- pool balls)
- Integrate previous equation over duration of collision
$$m_1 V_1^+ = m_1 V_1^- + \Lambda \quad (\text{equation 1})$$
- $m_1 V_1^-$ is linear momentum of particle 1 just before collision
- $m_1 V_1^+$ is the linear momentum just after collision
- Λ is the linear impulse
 - Integral of collision force over duration of collision



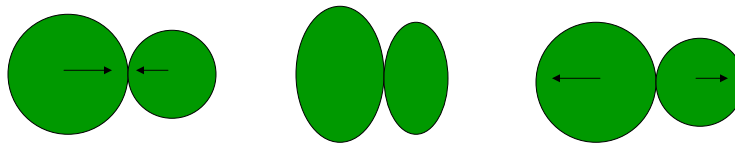
Frictionless Collision Response (3 of 4)

- 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, $-1*\Lambda$)
- Can solve these equations if know Λ
- Without friction, impulse force acts completely along unit surface normal vector at point of contact

$$\Lambda = \Lambda_s n \quad (\text{equation 2})$$
 - n is the unit surface normal vector (see collision detection for point of contact)
 - Λ_s 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^+, Λ_s).
 - Need third equation to solve for all



Frictionless Collision Response (4 of 4)



Period of deformation Period of restitution

- Third equation is approximation of material response to colliding objects

$$(V_1^+ - V_2^+) n = -\epsilon (V_1^- - V_2^-) n \quad (\text{equation 3})$$

- 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 racket, ϵ is 0.85 and deflated basketball with court ϵ is 0)
 - (Next slides have details)





Coefficient of Restitution (1 of 6)

- 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](#)



Coefficient of Restitution (2 of 6)

- Defined as the ratio of the differences in velocities before and after collision

$$\varepsilon = (V_1^+ - V_2^+) / (V_1^- - V_2^-)$$

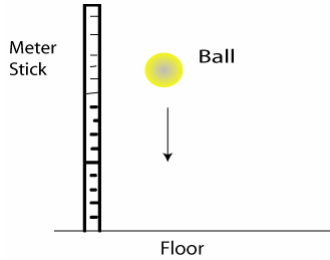
- For an object hitting an immovable object (ie- the floor)

$$\varepsilon = \text{sqrt}(h/H)$$

- Where h is bounce height, H is drop height



Coefficient of Restitution (3 of 6)

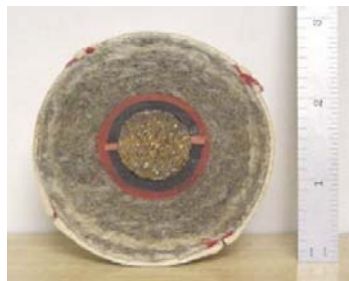


- Drop ball from fixed height (92 cm)
- Record bounce
- Repeat 5 times and average)
- Various balls

| 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 |



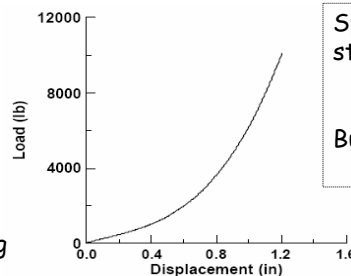
Coefficient of Restitution (4 of 6)



- 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:

$$F = kx$$

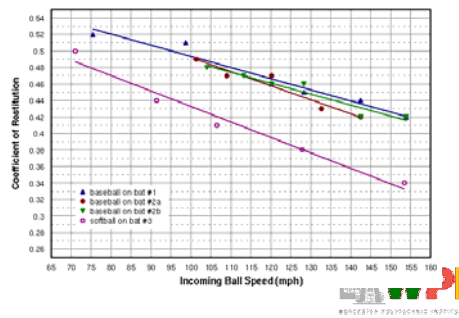
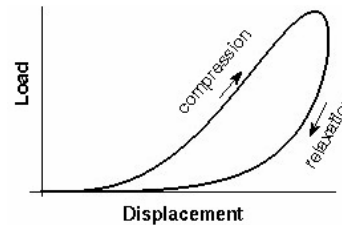
But is:

$$F = kx^p$$



Coefficient of Restitution (5 of 6)

- 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



Coefficient of Restitution (6 of 6)

- Last notes ...
- Technically
 - COR a property of a collision, not necessarily an object
 - 5 different types of objects \rightarrow 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

- Have 3 equations (equation 1, 1+ and 4) and 3 unknowns (V_1^+ , V_2^+ , Λ_s)

- Can then compute the linear impulse

$$\Lambda = - \left(\frac{m_1 m_2 (1 + \epsilon) (V_1^- - V_2^-) n}{m_1 + m_2} \right) n \quad (\text{equation 4})$$

- Can then apply Λ to previous equations:

- Equation 1 to get V_1^+ (and similarly V_2^+)
- ... and divide by m_1 (or m_2) to get after-collision velocities



The Story So Far

- Visited basic concepts in kinematics and Newtonian physics
- Generalized for 3 dimensions
- Ready to be used in some games!

- Show Pseudo code next
 - Simulating N Spherical Particles under Gravity with no Friction



Psuedocode (1 of 5)

```
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;
```



Psuedocode (2 of 5)

```
// 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
```



Pseudocode (3 of 5)

```
// for spherical particles, surface
// normal is just 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]);
```



Pseudocode (4 of 5)

```
// 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 (5 of 5)

```
// 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
 - Generalized translation motion
- Soft Body Dynamic System
- Collision Detection



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 Newtonian Equation of Motion

- Family of numerical simulation techniques called *finite difference methods*
 - The most common family of numerical techniques for rigid-body dynamics simulation
 - Incremental "solution" to equations of motion
- Derived from *Taylor series expansion* of properties we are interested in

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n. \quad \text{(Taylor series are used to estimate unknown functions)}$$

$$S(t+\Delta t) = S(t) + \Delta t \frac{d}{dt} S(t) + \frac{(\Delta t)^2}{2!} \frac{d^2}{dt^2} S(t) + \dots$$

- In general, not know values of any higher order. Truncate, remove higher terms

$$S(t+\Delta t) = S(t) + \Delta t \frac{d}{dt} S(t) + O(\Delta t)^2$$

- Can do beyond, but always higher terms
- $O(\Delta t)^2$ is called *truncation error*
- Can use to update properties (position)
 - Called "simple" or "explicit" *Euler integration*



Explicit Euler Integration (1 of 2)

- A "one-point" method since solve using properties at exactly one point in time, t , prior to update time, $t+\Delta t$.
 - $S(t+\Delta t)$ is the only unknown value so can solve without solving system of simultaneous equations
 - Important - every term on right side is evaluated at t , right before new time $t+\Delta t$
- View: $S(t+\Delta t) = S(t) + \Delta t \frac{d}{dt} S(t)$
 - new state prior state state derivative



Explicit Euler Integration (2 of 2)

- Can write numerical integrator to integrate arbitrary properties as change over time
- Integrate state vector of length N

```
void ExplicitEuler(N, new_S, prior_S, s_deriv, delta_t) {
  for (i=0; i<N; i++) {
    new_S[i] = prior_S[i] + delta_t * s_deriv[i];
  }
}
```
- For single particle, $S=(mV,p)$ and $d/dt S = (F,V)$
- Note, for 3D, mV and p have 3 values each:
 - $S(t) = (m_1 V_1, p_1, m_2 V_2, p_2, \dots, m_N V_N, p_N)$
 - $d/dt S(t) = (F_1, V_1, F_2, V_2, \dots, F_N, V_N)$



Explicit Euler Integration Example (1 of 2)

$V_{init} = 30 \text{ m/s}$
 Launch angle, ϕ : 75.2 degrees
 Launch angle, θ : 0 degrees (all motion in xz plane)
 Mass of projectile, m : 2.5 kg

| Time | Position (m) | | | Linear Momentum (kg-m/s) | | | Force (N) | | | Velocity (m/s) | | |
|------|--------------|-------|-------|--------------------------|--------|--------|-----------|-------|--------|----------------|-------|-------|
| | 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 |

t_{init} \mathbf{p}_{init} $m\mathbf{V}_{init}$ $\mathbf{F}=\text{Weight} = m\mathbf{g}$ \mathbf{V}_{init}

$\mathbf{S} = \langle m\mathbf{V}_{init}, \mathbf{p}_{init} \rangle$ $d\mathbf{S}/dt = \langle m\mathbf{g}, \mathbf{V}_{init} \rangle$



Explicit Euler Integration Example (1 of 2)

$$\mathbf{S}(t + \Delta t) = \mathbf{S}(t) + \Delta t \frac{d}{dt} \mathbf{S}(t) = \begin{bmatrix} 19.2 \\ 0.0 \\ 72.5 \\ 10.0 \\ 0.0 \\ 2.0 \end{bmatrix} + \Delta t \begin{bmatrix} 0.0 \\ 0.0 \\ -24.53 \\ 7.68 \\ 0.0 \\ 29.0 \end{bmatrix}$$

| $\Delta t = .2 \text{ s}$ | $\Delta t = .1 \text{ s}$ | $\Delta t = .01 \text{ s}$ |
|---------------------------|---------------------------|----------------------------|
| 19.2025 | 19.2025 | 19.2025 |
| 0.0 | 0.0 | 0.0 |
| 67.5951 | 72.0476 | 72.2549 |
| 11.5362 | 10.7681 | 10.0768 |
| 0.0 | 0.0 | 0.0 |
| 7.8000 | 4.9000 | 2.2900 |

Exact, Closed - form Solution

| | | |
|---------|---------|---------|
| 19.2 | 19.2 | 19.2 |
| 0.0 | 0.0 | 0.0 |
| 67.5951 | 72.0476 | 72.2549 |
| 11.5362 | 10.1536 | 10.0768 |
| 0.0 | 0.0 | 0.0 |
| 7.6038 | 4.8510 | 2.2895 |



Pseudo Code for Numerical Integration (1 of 2)

```
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];
        }
    }
}
```



Pseudo Code for Numerical Integration (2 of 2)

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

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

    // 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;
}
}
```





Collision Response in Simulation Loop

- Code can be used in game without collisions
- With collisions, need to modify
- If at beginning of step (at t before integration)
 - Resolve before copy `cur_S` to `prior_S`
 - For each colliding pair, use equation 4 to compute impulse and linear momentums as before
 - Replace `cur_S` with after collision momentums
 - When copy, `ExplicitEuler()` will use new after-collision velocities to resolve
- In general, can happen between t and Δt (and different for each pair!), say t_c
 - Split into two parts, t to t_c and then t_c to Δt
 - Integrate twice for each collision



Topics

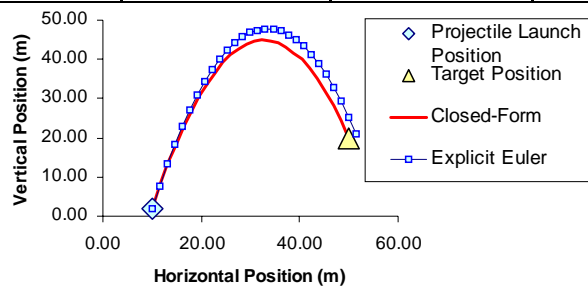
- Introduction
- Point Masses
 - Projectile motion
 - Collision response
- Rigid-Bodies
 - Numerical simulation
 - Controlling truncation error (next)
 - Generalized translation motion
- Soft Body Dynamic System
- Collision Detection



Explicit Euler Integration - Computing Solution Over Time

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

| Time | Position (m) | | | Linear Momentum (kg-m/s) | | | Force (N) | | | Velocity (m/s) | | |
|-------|--------------|-------|-------|--------------------------|--------|--------|-----------|-------|--------|----------------|-------|--------|
| | 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 (Verlet and others, not covered). Typically, more state kept. Stable within bounds.



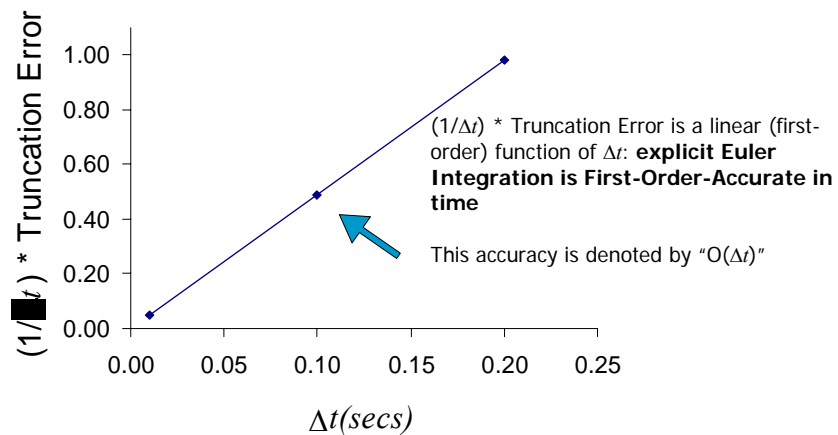
Truncation Error Example (1 of 2)

| | | | | | | |
|---|---|---|---|---|---|--|
| Truncation Error ($\Delta t = 0.2s$) | = | $\begin{bmatrix} 11.5362 \\ 0.0 \\ 7.800 \end{bmatrix}_{\text{numerical}}$ | - | $\begin{bmatrix} 11.5362 \\ 0.0 \\ 7.6038 \end{bmatrix}_{\text{exact}}$ | = | $\begin{bmatrix} 0.0 \\ 0.0 \\ 0.1962 \end{bmatrix}$ |
| Truncation Error ($\Delta t = 0.1s$) | = | $\begin{bmatrix} 10.1536 \\ 0.0 \\ 4.9000 \end{bmatrix}_{\text{numerical}}$ | - | $\begin{bmatrix} 10.1536 \\ 0.0 \\ 4.8510 \end{bmatrix}_{\text{exact}}$ | = | $\begin{bmatrix} 0.0 \\ 0.0 \\ 0.049 \end{bmatrix}$ |
| Truncation Error ($\Delta t = 0.01s$) | = | $\begin{bmatrix} 10.0768 \\ 0.0 \\ 2.2900 \end{bmatrix}_{\text{numerical}}$ | - | $\begin{bmatrix} 10.0768 \\ 0.0 \\ 2.2895 \end{bmatrix}_{\text{exact}}$ | = | $\begin{bmatrix} 0.0 \\ 0.0 \\ 0.0005 \end{bmatrix}$ |

Can only compare if normalize (divide by Δt)



Truncation Error Example (2 of 2)



Guidelines? Step less than frame rate (otherwise, no update)
 $\rightarrow \Delta t$ under 30 ms (20 ms good choice)



Frame Rate Independence

- Given numerical simulation sensitive to time step (Δt), important to create physics engine that is frame-rate independent
 - Results will be repeatable, every time run simulation with same inputs
 - Regardless of CPU/GPU performance
 - Maximum control over simulation
- Pseudo code next



Pseudo Code for Frame Rate Independence

```
void main() {
    float delta_t = 0.02;           // physics time
    float game_time;                // game time
    float prev_game_time;           // game time at last step
    float physics_lag_time=0.0;     // time since last update

    // simulation/render loop
    while(1) {
        update game_time; // could be take from system clock
        physics_lag_time += (game_time - prev_game_time);
        while (physics_lag_time > delta_t) {
            doPhysicsSimulation(delta_t);
            physics_lag_time -= delta_t;
        }

        prev_game_time = game_time;

        render scene;
    }
}
```



Topics

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



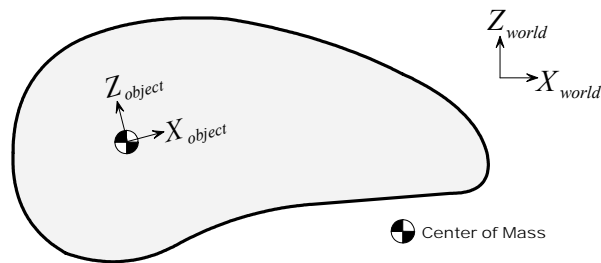
Generalized Translation Motion

- Have basic numerical Physics in place
- Consider variety of non-constant forces
 - Any combination act on object at any time
 - Apply in `calcForce(i)` code, previously
- Net force, F_{net} , by adding all applied forces
 - F_{net} exactly value to be used in state derivative vector for numerical integration
 - If F_{net} has zero magnitude, the object has said to be in *translational equilibrium*, although it may still have nonzero velocity



Generalized Translation Motion

- Previous equations work for objects of any size
 - But describe motion at a single point
- For rigid bodies, typically choose center of mass
- May be rotation



Generalized Motion Mini-Outline

- Linear Springs
- Viscous Damping
- Aerodynamic Drag
- Surface Friction
- Example



Linear Springs

- Spring connects end-points, p_{e1} and p_{e2}
- Has rest length, l_{rest}
 - Exerts zero force
 - Stretched longer than l_{rest} → attraction
 - Stretched shorter than l_{rest} → repulsion
- *Hooke's law*
 - $F_{spring} = k (l - l_{rest}) \mathbf{d}$
 - k is spring stiffness (in Newtons per meter)
 - l is current spring length
 - \mathbf{d} is unit length vector from p_{e1} to p_{e2} (provides direction)
- F_{spring} applied to object 1 at p_{e1}
- $-1 * F_{spring}$ applied to object 2 at p_{e2}



Viscous Damping

- Connects end-points, p_{e1} and p_{e2}
- Provides dissipative forces (reduce kinetic energy)
- Often used to reduce vibrations in machines, suspension systems, etc.
 - Called *dashpots*
- Apply damping force to objects along connected axis (put on the brakes)
 - Note, relative to velocity along axis
$$F_{damping} = c ((V_{ep2} - V_{ep1}) \mathbf{d}) \mathbf{d}$$
 - \mathbf{d} is unit length vector from p_{e1} to p_{e2} (provides direction)
 - c is *damping coefficient*
- $F_{damping}$ applied to object 1 at p_{e1}
- $-1 * F_{damping}$ applied to object 2 at p_{e2}



Aerodynamic Drag

- An object through fluid has drag in opposite direction of velocity
- Simple representation:
$$F_{\text{drag}} = -\frac{1}{2} \rho |V|^2 C_D S_{\text{ref}} \frac{V}{|V|}$$
- S_{ref} is front-projected area of object
 - Cross-section area of bounding sphere
- ρ is the mass-density of the fluid
- C_D is the drag co-efficient ([0..1], no units)
 - Typical values from 0.1 (streamlined) to 0.4 (not streamlined)



Surface Friction (1 of 2)

- Two objects collide or slide within contact plane \rightarrow friction
- Complex: starting (static) friction higher than (dynamic) friction when moving. Coulomb friction, for static:
 - F_{friction} is same magnitude as $\mu_s |F|$ (when moving $\mu_d |F|$)
 - μ_s static friction coefficient
 - μ_d is dynamic friction coefficient
 - F is force applied in same direction
 - (F_{friction} in opposite direction)
- Friction coefficients (μ_s and μ_d) depend upon material properties of two objects
 - Examples:
 - ice on steel has a low coefficient of friction (the two materials slide past each other easily)
 - rubber on pavement has a high coefficient of friction (the materials do not slide past each other easily)
 - Can go from near 0 to greater than 1
 - Ex: wood on wood ranges from 0.2 to 0.75
 - Must be measured (but many links to look up)
 - Generally, μ_s larger than μ_d





Surface Friction (2 of 2)

- If V is zero:
 - $F_{\text{friction}} = -[F_t / |F_t|] \min(\mu_s |F_n|, |F_t|)$
 - $\min()$ ensures no larger (else starts to move)
- If V is non-zero:
 - $F_{\text{friction}} = [-V_t / |V_t|] \mu_d |F_n|$
- Friction is dissipative, acting to reduce kinetic energy



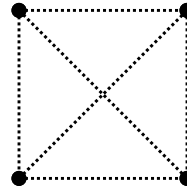
Topics

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



Simple Spring-Mass-Damper Soft-Body Dynamics System (1 of 3)

- Using results thus far, construct a simple soft-body dynamics simulator
- Create polygon mesh with interesting shape
- Use physics to update position of vertices
 - Create particle at each vertex
 - Assign mass
 - Create a spring and damper between unique pairs of particles
 - Spring rest lengths equal to distance between particles
- Code listing 4.3.8



..... 1 spring and 1 damper



Simple Spring-Mass-Damper Soft-Body Dynamics System (2 of 3)

```
void main() {  
    initialize particles (vertices)  
    initialize spring and damper between pairs  
    while (1) {  
        doPhysicsSimulationStep()  
        for each particle  
            render  
    }  
}
```

- Key is in `vector CalcForce(i)`
 - (Next)



Simple Spring-Mass-Damper Soft-Body Dynamics System (3 of 3)

```
vector CalcForce(i) {
    vector SForce /* spring */, Dforce /* damper */;
    vector net_force; // returns this

    // Initialize net force for gravity
    net_force = mass[i] * g;

    // compute spring and damper forces for each other vertex
    for (j=0; j<N; j++) {

        // Spring Force
        // compute unit vector from i to j and length of spring
        d = cur_S[2*j+1] - cur_S[2*i+1];
        length = d.length();
        d.normalize(); // make unit length

        // i is attracted if < rest, repelled if > rest (equation 20)
        SForce = k[i][j] * (length - lrest[i][j]) * d;

        // Damping Force
        // relative velocity
        relativeVel = (cur_s[2*j]/mass[j]) - (cur_S[2*i]/mass[i]);

        // if j moving away from i then draws i towards j, else repels i (equation 21)
        DForce = c[i][j] * relativeVel.dotProduct(d) * d;

        // increment net force
        net_force = SForce + DForce;
    }
    return (net_force);
}
```



Final Comments (1 of 2)

- Also rotational motion (*torque*), not covered
- Simple Games
 - Closed-form particle equations may be all you need
 - Numerical particle simulation adds flexibility without much coding effort
 - Works for non-constant forces
 - Provided generalized rigid body simulation
- Want more? Additional considerations
 - Multiple simultaneous collision points
 - Articulating rigid body chains, with joints
 - Rolling friction, friction during collision
 - Resting contact/stacking
 - Breakable objects
 - Soft bodies (deformable)
 - Smoke, clouds, and other gases
 - Water, oil, and other fluids



Final Comments (2 of 2)

- Commercial Physics Engines
 - Game Dynamics SDK (www.havok.com)
 - Renderware Physics (www.renderware.com)
 - NovodeX SDK (www.novdex.com)
- Freeware/Shareware Physics Engines
 - Open Dynamics Engine (www.ode.org)
 - Tokamak Game Physics SDK (www.tokamakphysics.com)
 - Newton Game Dynamics SDK (www.newtondynamics.com)
- Save time and trouble of own code
- Many include collision detection
- But ... still need good understanding of physics to use properly



Topics

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





Collision Detection

- Determining when objects collide not as easy as it seems
 - Geometry can be complex (beyond spheres)
 - Objects can move fast
 - Can be many objects (say, n)
 - Naïve solution is $O(n^2)$ time complexity, since every object can potentially collide with every other object
- Two basic techniques
 - *Overlap testing*
 - Detects whether a collision has already occurred
 - *Intersection testing*
 - Predicts whether a collision will occur in the future



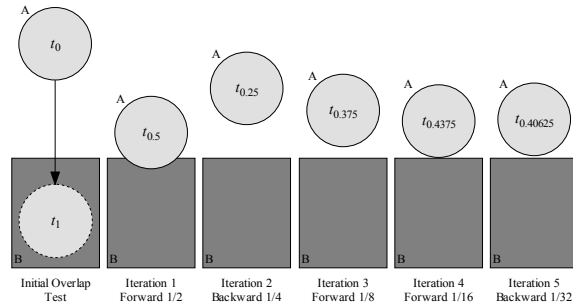
Overlap Testing

- Facts
 - Most common technique used in games
 - Exhibits more error than intersection testing
- Concept
 - For every simulation step, test every pair of objects to see if overlap
 - Easy for simple volumes like spheres, harder for polygonal models
- Useful results of detected collision
 - Collision normal vector (needed for physics actions, as seen earlier)
 - Time collision took place



Overlap Testing: Collision Time

- Collision time calculated by moving object back in time until right before collision
 - Move forward or backward $\frac{1}{2}$ step, called *bisection*

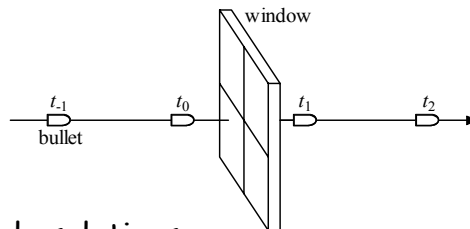


- Get within a delta (close enough)
 - With distance moved in first step, can know "how close"
- In practice, usually 5 iterations is pretty close



Overlap Testing: Limitations

- Fails with objects that move too fast
 - Unlikely to catch time slice during overlap

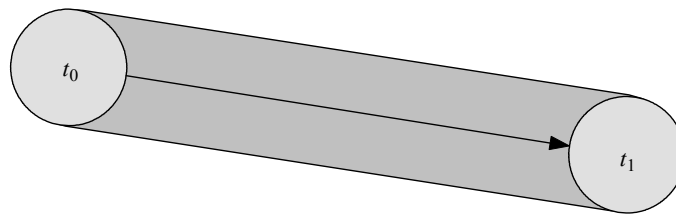


- Possible solutions
 - Design constraint on speed of objects (fastest object moves smaller distance than thinnest object)
 - May not be practical for all games
 - Reduce simulation step size
 - Adds overhead since more computation



Intersection Testing

- Predict future collisions
- Extrude geometry in direction of movement
 - Ex: *swept* sphere turns into a "capsule" shape
- Then, see if overlap
- When predicted:
 - Move simulation to time of collision
 - Resolve collision
 - Simulate remaining time step

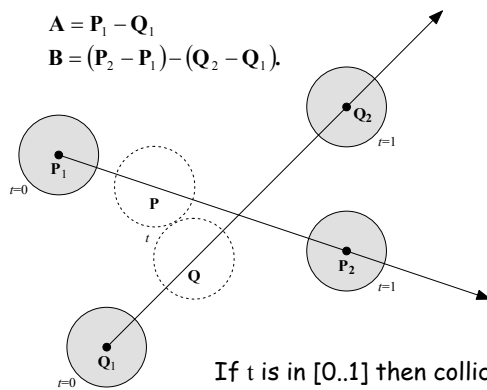


Intersection Testing: Sphere-Sphere Collision

$$t = \frac{-(\mathbf{A} \cdot \mathbf{B}) - \sqrt{(\mathbf{A} \cdot \mathbf{B})^2 - B^2(A^2 - (r_P + r_Q)^2)}}{B^2},$$

$$\mathbf{A} = \mathbf{P}_1 - \mathbf{Q}_1$$

$$\mathbf{B} = (\mathbf{P}_2 - \mathbf{P}_1) - (\mathbf{Q}_2 - \mathbf{Q}_1).$$



Or, simpler:

$$d^2 = A^2 - \frac{(\mathbf{A} \cdot \mathbf{B})^2}{B^2}$$

If distance large enough, no collision:

$$d^2 > (r_P + r_Q)^2$$



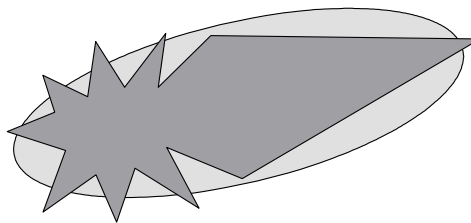
Dealing with Complexity

- Complex geometry must be simplified
 - Complex object can have 100's or 1000's of polygons
 - Testing intersection of each costly
- Reduce number of object pair tests
 - There can be 100's or 1000's of objects
 - If test all, $O(n^2)$ time complexity



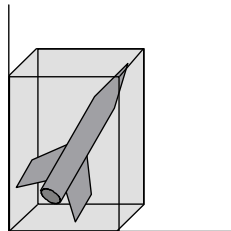
Complex Geometry - Bounding Volume (1 of 3)

- *Bounding volume* is simple geometric shape that completely encapsulates object
 - Ex: approximate spiky object with ellipsoid
- Note, does not need to encompass, but might mean some contact not detected
 - May be ok for some games

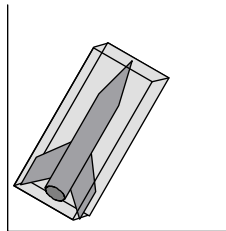


Complex Geometry - Bounding Volume (2 of 3)

- Testing cheaper
 - If no collision with bounding volume, no more testing is required
 - If there is a collision, then there *could* be a collision
 - More refined testing can be used
- Commonly used bounding volumes
 - *Sphere* - if distance between centers less than sum of Radii then no collision
 - *Box* - axis-aligned (loose fit) or oriented (tighter fit)



Axis-Aligned Bounding Box



Oriented Bounding Box



Complex Geometry - Bounding Volume (3 of 3)

- For complex object, can fit several bounding volumes around unique parts
 - Ex: For avatar, boxes around torso and limbs, sphere around head
- Can use hierarchical bounding volume
 - Ex: large sphere around whole avatar
 - If collide, refine with more refined bounding boxes

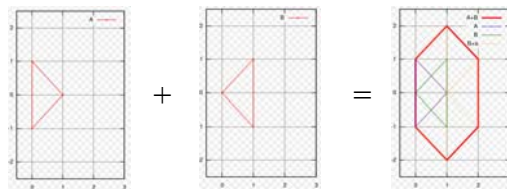
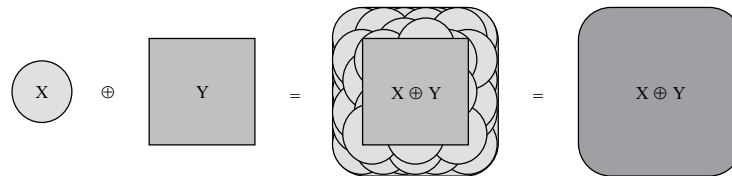


Complex Geometry - Minkowski Sum (1 of 2)

- Take sum of two convex volumes to create new volume

- Sweep origin (center) of X all over Y

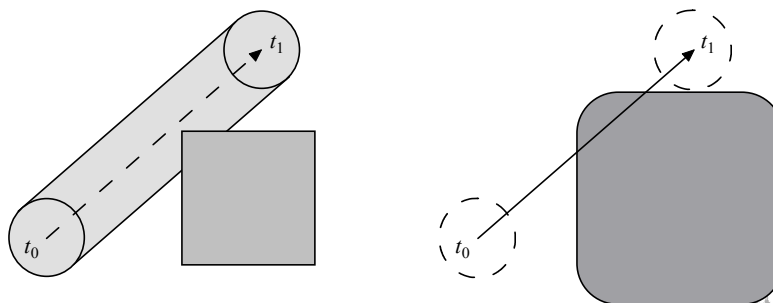
$$X \oplus Y = \{A + B : A \in X \text{ and } B \in Y\}$$



Complex Geometry - Minkowski Sum (2 of 2)

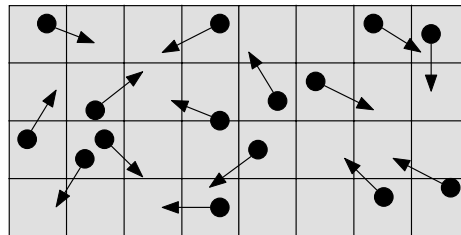
- Test if single point in X in new volume, then collide

- Take center of sphere at t_0 to center at t_1
- If line intersects new volume, then collision



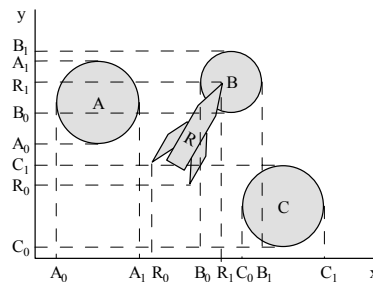
Reduced Collision Tests - Partitioning

- Partition space so only test objects in same cell
 - If N objects, then $\sqrt{N} \times \sqrt{N}$ cells to get linear complexity
- But what if objects don't align nicely?
- What if all objects in same cell? (same as no cells)



Reduced Collision Tests - Plane Sweep

- Objects tend to stay in same place
 - So, don't need to test all pairs
- Record bounds of objects along axes
- Any objects with overlap on *all* axes should be tested further
- Time consuming part is sorting bounds
 - *Quicksort* $O(n \log(n))$
 - But, since objects don't move, can do better if use *Bubblesort* to repair - nearly $O(n)$



Collision Resolution (1 of 2)

- Once detected, must take action to resolve
 - But effects on trajectories and objects can differ
- Ex: Two billiard balls strike
 - Calculate ball positions at time of impact
 - Impart new velocities on balls
 - Play "clinking" sound effect
- Ex: Rocket slams into wall
 - Rocket disappears
 - Explosion spawned and explosion sound effect
 - Wall charred and area damage inflicted on nearby characters
- Ex: Character walks through invisible wall
 - Magical sound effect triggered
 - No trajectories or velocities affected



Collision Resolution (2 of 2)

- Prologue
 - Collision known to have occurred
 - Check if collision should be ignored
 - Other events might be triggered
 - Sound effects
 - Send collision notification messages (OO)
- Collision
 - Place objects at point of impact
 - Assign new velocities
 - Using physics or
 - Using some other decision logic
- Epilog
 - Propagate post-collision effects
 - Possible effects
 - Destroy one or both objects
 - Play sound effect
 - Inflict damage
- Many effects can be done either in the prologue or epilogue



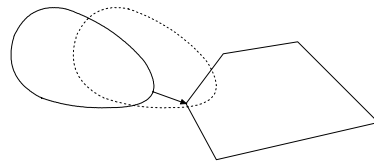
Collision Resolution - Collision Step

- For overlap testing, four steps
 - Extract collision normal
 - Extract penetration depth
 - Move the two objects apart
 - Compute new velocities (previous stuff)
- For intersection testing, two steps
 - Extract collision normal
 - Compute new velocities (previous stuff)

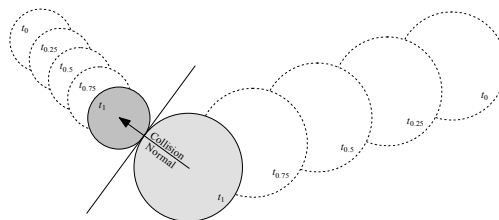


Collision Resolution - Collision Normal

- Find position of objects before impact
 - Use bisection
- Use two closest points to construct the collision normal vector (Case A)
- For spheres, normal is line connecting centers (Case B)



A



B





Collision Resolution - Intersection Testing

- Simpler than resolving overlap testing
 - No need to find penetration depth or move objects apart
- Simply
 1. Extract collision normal
 2. Compute new velocities



Collision Detection Summary

- Test via overlap or intersection (prediction)
- Control complexity
 - Shape with bounding volume
 - Number with cells or sweeping
- When collision: prolog, collision, epilog

