Basic Game Physics

IMGD 4000

With material from: *Introduction to Game Development*, Second Edition. Edited by Steve Rabin. Cengage Learning, 2009. (Chapter 4.3)

Introduction (1 of 2)

- *What is game physics?*
  - Computing *motion* of objects in virtual scene
    - Including player avatars, NPC's, inanimate objects
  - Computing *mechanical interactions* of objects
    - Interaction usually involves contact (collision)
  - Simulation must be real-time (versus high-precision simulation for CAD/CAM, etc.)
  - Simulation may be very realistic, approximate, or intentionally distorted (for effect)

Introduction (2 of 2)

- *And why is it important?*
  - Can improve immersion
  - Can support new gameplay elements
  - Becoming increasingly prominent (expected) part of high-end games
  - Like AI and graphics, facilitated by hardware developments (multi-core, GPU)
  - Maturation of physics engine market

Physics Engines

- Similar *buy* vs. *build* analysis as game engines
  - Buy:
    - Complete solution from day one
    - Proven, robust code base (hopefully)
    - Feature sets are pre-defined
    - Costs range from *free* to *expensive*
  - Build:
    - Choose exactly features you want
    - Opportunity for more game-specification optimizations
    - Greater opportunity to innovate
    - Cost guaranteed to be expensive (unless features extremely minimal)

Physics Engines

- Open source
  - Box2D, Bullet, Chipmunk, JigLib, ODE, OPAL, OpenTissue, PAL, Tokamak, Farseer, Physics2d, Glaze
- Closed source (limited free distribution)
  - Newton Game Dynamics, Simple Physics Engine, True Axis, PhysX
- Commercial
  - Havok, nV Physics, Vortex
- Relation to Game Engines
  - Native, e.g,. C4
  - Integrated, e.g., UE4 + PhysX
  - Pluggable, e.g., C4 + PhysX, jME + ODE (via jME Physics)

Basic Game Physics Concepts

- *Why are we studying this?*
  - To use an engine effectively, you need to understand something about what it's doing
  - You may need to implement small features or extensions yourself
  - Cf., owning a car without understanding anything about how it works (possible, yes, but ideal?, no)
- Examples
  - Kinematics and dynamics
  - Projectile motion
  - Collision detection and response

Outline

- Introduction (done)
- Kinematics (next)
- Rigid Body Simulation
- The Firing Solution
- Collision Detection
- Ragdoll Physics
- PhysX

Kinematics (1 of 3)

- Study of motion of objects *without* taking into account mass or force
- Basic quantities: position, time
- ... and their derivatives: velocity, acceleration
- Basic equations:

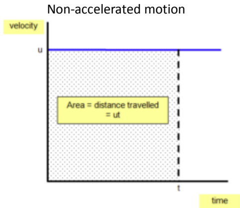
```
1) d = vt
2) v = u + at
3) d = ut + ½at²
4) v² = u² + 2ad
```

Where:
t - (elapsed) time
d - distance (change in position)
v - (final) velocity (change in distance per unit time)
a - acceleration (change in velocity per unit time)
u - (initial) velocity

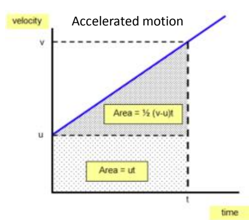
Note, equation #3 is the integral of equation #2 with respect to time (see next slide). Equation #4 can be useful.

Kinematics (2 of 3)

Non-accelerated motion



Accelerated motion

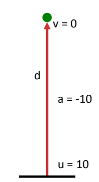



```
d = ut
```
Example:
```
u = 20 m/s, t = 300 s
d = 20 x 3000 = 6000 m
```

```
d = ut + ½at²
```
Example:
```
u=0 m/s, a=4m/s², t=3s
d = 0x3 + 0.5x4x9 = 18m
```

Kinematics (3 of 3)

Prediction Example: If you throw a ball straight up into the air with an initial velocity of 10 m/sec, how high will it go?



$v^2 = u^2 + 2ad$

u = 10 m/sec (initial speed upward)
a = -10 m/sec² (approx gravity)
v = 0 m/sec (at top of flight)

$0 = 10^2 + 2(-10)d$
$d = 5$ meters

(Note, answer independent of mass of ball!)

Doing It In 3D

- Mathematically, consider all quantities involving position to be vectors:

$d = vt$
$v = u + at$
$d = ut + at^2/2$

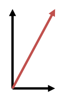
- Computationally, using appropriate 3-element vector datatype

Dynamics

- Notice that preceding kinematic descriptions say nothing about *why* an object accelerates (or why its acceleration might change)
- To get a full “modern” physical simulation you need to add two more basic concepts:
  - force
  - mass
- Discovered by Sir Isaac Newton
- Around 1700 ☺

Newton's Laws

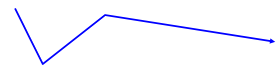
1. A body will remain at rest or continue to move in a straight line at a constant speed unless acted upon by a *force*.
2. The acceleration of a body is *proportional* to the resultant force acting on the body and is in the same direction as the resultant force.
3. For every action, there is an equal and opposite reaction.



13

Motion Without Newton's Laws

- Pac-Man or early Mario style
  - Follow path with *instantaneous changes* in speed and direction (velocity)



  - Not physically possible
- Note - fine for some casual games (especially with appropriate animations)

14

Newton's Second Law
F = ma

At each moment in time:

F = force vector, in Newton's

m = mass (intrinsic property of matter), in kg

a = acceleration vector, in m/sec^2

Player cares about state of world (position of objects). Equation is fundamental driver of all physics simulations.

- Force causes acceleration (**a** = **F**/m)
- Acceleration causes change in velocity
- Velocity causes change in position

15

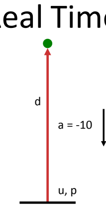
How Are Forces Applied?

- May involve contact
  - Collision (rebound)
  - Friction (rolling, sliding)
- Without contact
  - Rockets/Muscles/Propellers
  - Gravity
  - Wind (if not modeling air particles)
  - Magic
- Dynamic (force) modeling also used for autonomous steering behaviors

16

Computing Kinematics in Real Time

```
start = getTime()      // start time
p = 0                  // initial position
u = 10                 // initial velocity
a = -10

function update () {   // in game loop
   now = getTime()
   t = now - start
   simulate(t)
}

function simulate (t) {
   d = (u + (0.5 * a * t)) * t
   move object to p + d  // move to loc. computed since start
}
```



d = ut + at^2/2

Note! Number of calls and time values to `simulate()` depend on (changing) ***game loop time*** (frame rate)

Is this a problem? It can be! For rigid body simulation with colliding forces and friction (e.g., many interesting cases) …

17

Outline

- Introduction (done)
- Kinematics (done)
- Rigid Body Simulation (next)
- The Firing Solution
- Collision Detection
- Ragdoll Physics
- PhysX

18

Rigid-Body Simulation

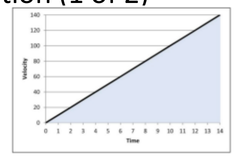
- If no rotation, only gravity and occasional frictionless collision, basic Kinematic equations are fine
  - Closed form solution can be integrated
    (e.g., d = ut + ½at²)
- But in many games (and life!), interesting motion involves non-constant forces and collision impulse forces
  - Unfortunately, often no closed-form solutions
- *What to do?* → *Numerical simulation*

Numerical Simulation techniques for incrementally solving equations of motion when forces applied to object are not constant, or when otherwise there is no closed-form solution

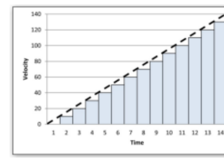
Numerical Simulation of Newtonian Equation of Motion (1 of 2)

- Suppose know position (p), velocity (v) and acceleration (a) at time (t_i) and frame time (Δt)
- Want position at next frame time (t_{i+1})
  - Don't know *exactly* how forces are affecting (wind, friction, gravity …)
- Can compute:
  - $p_{i+1} = s_i + v_i * \Delta t$
  - $v_{i+1} = v_i + a * \Delta t$

What is this doing? → Instead of integrating a curve, sum over discrete time-slices



vs.



Numerical Simulation of Newtonian Equation of Motion (2 of 2)

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

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

(Taylor series are used to estimate unknown functions)

$$S(t+\Delta t) = S(t) + \Delta t\, d/dt\, S(t) + (\Delta t)^2/2!\, d^2/dt\, S(t) + \ldots$$

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

$$S(t+\Delta t) = S(t) + \Delta t\, d/dt\, S(t) + O(\Delta t)^2$$

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

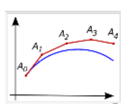


Illustration of the Euler method. The unknown curve is in blue, and its polygonal approximation is in red.

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+Δt
  - S(t+Δt) is only unknown value so can solve without solving system of simultaneous equations
  - Every term on right side is evaluated at t, right before new time t+Δt
- View: $S(t+\Delta t) = S(t) + \Delta t\, d/dt\, S(t)$

new state prior state state derivative

Explicit Euler Integration (2 of 2)

- Write numerical integrator over arbitrary properties as change over time
- For single particle, S = (mV, p) and d/dt S = (F, V)
  - Derivative of position (p) is velocity (V)
    - i.e., how position changes with respect to time
  - Derivative of momentum (mass m * V) is force (F)
    - i.e., how momentum changes with respect to time
- Integrate state vector of length N

```
Vector ExplicitEuler(int N, Vector prior_S, Vector deriv_S, float delta_t)
  Vector new_S[N];
  for (i=0; i<N; i++)
     new_S[i] = prior_S[i] + delta_t * deriv_S[i];
  return new_S;
```

- Note, for 3D, mV and p have 3 values each
  - $S(t) = (m_1V_1, p_1, m_2V_2, p_2, \ldots, m_NV_N, p_N)$
  - $d/dt\, S(t) = (F_1, V_1, F_2, V_2, \ldots, F_N, V_N)$

Explicit Euler Integration Example (1 of 2)

V_{init} = 30 m/s
Launch angle: 75.2 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} $\mathbf{p}_{init}$ $m\mathbf{V}_{init}$ **F=Weight** $= m\mathbf{g}$ $\mathbf{V}_{init}$

$\mathbf{S} = \langle m\mathbf{V}_{init}, \mathbf{p}_{init} \rangle$

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

Explicit Euler Integration Example (2 of 2)

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

$\Delta t = .2\ s$	$\Delta t = .1\ s$	$\Delta t = .01\ s$
$= \begin{bmatrix} 19.2025 \\ 0.0 \\ 67.5951 \\ 11.5362 \\ 0.0 \\ \boxed{7.8000} \end{bmatrix}$	$= \begin{bmatrix} 19.2025 \\ 0.0 \\ 72.0476 \\ 10.7681 \\ 0.0 \\ \boxed{4.9000} \end{bmatrix}$	$= \begin{bmatrix} 19.2025 \\ 0.0 \\ 72.2549 \\ 10.0768 \\ 0.0 \\ \boxed{2.2900} \end{bmatrix}$

Exact, Closed - form Solution

$\Delta t = .2\ s$	$\Delta t = .1\ s$	$\Delta t = .01\ s$
$= \begin{bmatrix} 19.2 \\ 0.0 \\ 67.5951 \\ 11.5362 \\ 0.0 \\ \boxed{7.6038} \end{bmatrix}$	$= \begin{bmatrix} 19.2 \\ 0.0 \\ 72.0476 \\ 10.1536 \\ 0.0 \\ \boxed{4.8510} \end{bmatrix}$	$= \begin{bmatrix} 19.2 \\ 0.0 \\ 72.2549 \\ 10.0768 \\ 0.0 \\ \boxed{2.2895} \end{bmatrix}$

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); // update state of all particles
        for (i=0; i<N; i++) {
                render particle i at position cur_S[2*i+1];
        }
   }
```

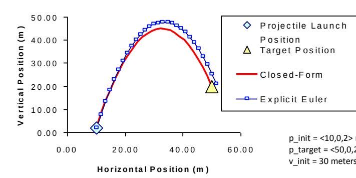

Pseudo Code for Numerical Integration (2 of 2)

```
// Update state of all particles based on physics
void doPhysicsSimulationStep(float 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;
}
```


Computing Position Over Time

- Solution proceeds step-by-step, each time integrating from 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

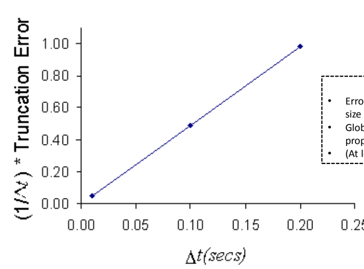


p_init = <10,0,2> meters
p_target = <50,0,20> meters
v_init = 30 meters per second

Truncation Error

- Numerical simulation can be different from exact, closed-form solution
  - Difference primarily truncation error
- Truncation error can accumulate causing instability
  - Ultimately produces floating point overflow
  - Unstable simulations behave unpredictably (not same each time)
- Sometimes, truncation error can become zero
  - In other words, produces exact, correct result
  - For example, when zero force is applied or frictionless and constant force
- But, more often truncation error is non-zero. Control by:
  - Select different numerical integrator (*Vertlet*, *Runge-Kutta* or others). Typically, more state kept. Stable within bounds.
  - Reduce time step, Δt (next slide)

Truncation Error Example



Euler
- Error per step proportional to step size squared
- Global error (error at given time) proportional to step size
- (At lower limit, error goes to 0)

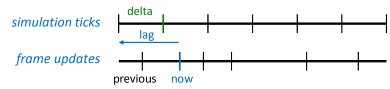
Trade-off: truncation error and computation interval
Guidelines? Step more often than frame rate (otherwise, no update!)
→ Δt under 30 ms (20 ms a good choice)

Frame Rate Independence

- Complex numerical simulations used in physics engines are sensitive to time steps (due to truncation error and other numerical effects)
- But results need to be repeatable regardless of CPU/GPU performance
  - for debugging
  - for game play
- So, if frame rate drops (game loop can't keep up), then physics will change
- Solution: Control physics simulation interval independently of frame rate

31

Frame Rate Independence




```
start = ...
delta = 0.02       // physics simulation interval (sec)
lag = 0            // time since last simulated
previous = 0       // time of previous call to update

function update() { // in game loop
     now = getTime()
     t = (previous - start) – lag   // previous simulate()
     lag = lag + (now - previous)   // additional lag
     while ( lag > delta )          // repeat until caught up
         t = t + delta
         simulate(t)               // note: kinematics.  If dynamic, use delta
         lag = lag - delta
     previous = now                // simulation caught up to current time
}
```

32

Outline

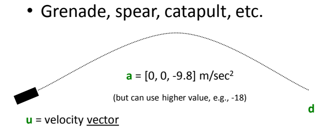
- Introduction (done)
- Kinematics (done)
- Rigid Body Simulation (done)
- The Firing Solution (next)
- Collision Detection
- Ragdoll Physics
- PhysX

33

The Firing Solution (1 of 3)

- How to hit target
  - Beam weapon or high-velocity bullet over short ranges can be viewed as traveling in straight line
  - But projectile travels in parabolic arc
    - Grenade, spear, catapult, etc.

$d = ut + at^2/2$



Most typical game situations, *magnitude* of u fixed. We only need to know relative components (orientation) → **Challenge:**

- Given d, solve for u

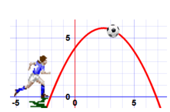
34

Remember Quadratic Equations?

- Make nice curves
  - Like firing at target!
- Solutions are where equals 0. E.g., when firing with gun on ground:
  - At gun muzzle
  - At target
- But unlike in algebra class, not just solving quadratic but finding angle with y = gun, y = target
- Angle changes speed in x-direction, but also time spent in air
- After hairy math [Millington 3.5.3], three relevant cases:
  - Target is out of range (no solution)
  - Target is at exact maximum range (single solution)
  - Target is closer than maximum range (two possible solutions)

this makes it Quadratic

$5x^2 - 3x + 3 = 0$

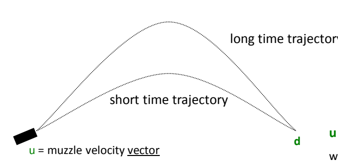


- Solve with quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

35

The Firing Solution (2 of 3)



[Millington 3.5.3]

$u = (2\Delta - at^2) / (2 \text{ (muzzle\_v) } t)$

where muzzle_v = max muzzle speed

Δ is difference vector from d to u

a is gravity

- Usually choose short time trajectory
  - Gives target less time to escape
  - Unless shooting over wall, etc.

Project Firing

36

The Firing Solution (3 of 3)

```
function firingSolution (start, target, muzzle_v, gravity) {

   // Calculate vector back from target to start
   delta = target - start

   // Real-valued coefficents of quadratic equation
   a = gravity * gravity
   b = -4 * (gravity * delta + muzzle_v * muzzle_v)
   c = 4 * delta * delta

   // Check for no real solutions
   if ( 4 * a * c > b * b ) return null

   // Find short and long times to target
   disc = sqrt (b * b - 4 * a *c)
   t1 = sqrt ( ( -b + disc ) / (2 * a ))
   t2 = sqrt ( ( -b - disc ) / (2 * a ))

   // Pick shortest valid time to target (ttt)
   if ( t1 < 0 ) && ( t2 < 0) return null   // No valid times
   if ( t1 < 0 ) ttt = t2 else
   if ( t2 < 0 ) ttt = t1 else
   ttt = min ( t1, t2 )

   // Return firing vector
   return ( 2 * delta - gravity * ttt * ttt ) / ( 2 * muzzle_v * ttt )
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note, a, b and c are scalars so a normal square root.

*Note scalar product of two vectors using *:*

*[a,b,c] * [d,e,f] = a*d + b*e + c*f*

[Millington 3.5.3]

Outline

- Introduction (done)
- Kinematics (done)
- Rigid Body Simulation (done)
- The Firing Solution (done)
- Collision Detection (next)
- Ragdoll Physics
- PhysX

Collision Detection

- Determining when objects collide is not as easy as it seems
  - Geometry can be complex
  - Objects can be moving quickly
  - There can be *many* objects
    - naive algorithms are $O(n^2)$
- Two basic approaches:
  - Overlap testing
    - Detects whether collision has already occurred
  - Intersection testing
    - Predicts whether collision will occur in future

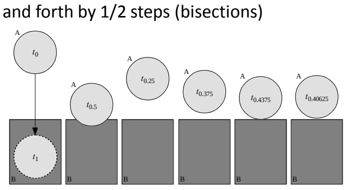
Overlap Testing

Basics – discussed and implemented in IMGD 3000!

- Most common technique used in games
- Exhibits more error than intersection testing
- Basic idea:
  - at every simulation step, test *every pair* of objects to see if overlap
- Easy for simple volumes (e.g., spheres), harder for polygonal models
- Results of test:
  - collision normal vector (useful for reaction)
  - time that collision took place

Overlap Testing: Finding Collision Time

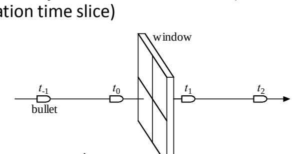
- Calculated by doing "binary search" in time, moving object back and forth by 1/2 steps (bisections)



- In practice, five iterations usually enough

Limitations of Overlap Testing

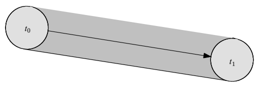
- Fails with objects that move too fast (no overlap during simulation time slice)



- Solution approach:
  - constrain game design so that fastest object moves smaller distance in one physics "tick" (delta) than thinnest object
  - may require reducing simulation step size (adds computation overhead)

Intersection Testing

- Predict future collisions
- Extrude geometry in direction of movement
  - e.g., “swept” sphere turns into capsule shape



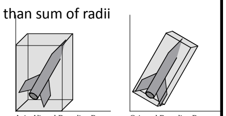
- Then, see if extruded shape overlaps objects
- When collision found (predicted)
  - Move simulation to time of collision (have collision point)
  - Resolve collision
  - Works for bullet/window example (bullet becomes line segment)

Speeding Up Collision Detection

- Bounding Volumes
  - Oriented
  - Hierarchical
- Partitioning
- Plane Sweep

Bounding Volumes

- Commonly used volumes
  - sphere - distance between centers less than sum of radii
  - boxes
    - axis aligned (loose fit, easier math)
    - oriented (tighter fit, more expensive)



Axis-Aligned Bounding Box   Oriented Bounding Box

- If bounding volumes don’t overlap, then no more testing is required
  - If overlap, more refined testing required
  - Bounding volume alone may be good enough for some games

Complex Bounding Volumes

- Multiple volumes per object
  - e.g., separate volumes for head, torso and limbs of avatar object
- Hierarchical volumes
  - e.g., boxes inside of boxes

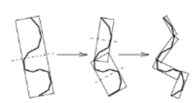
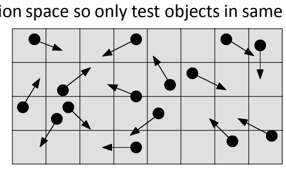


Figure 1: Building the OBBTree: recursively partition the bounded polygons and bound the resulting groups.

[Gottschalk, Lin, Minocha, SIGGRAPH '96]

Partitioning for Collision Testing

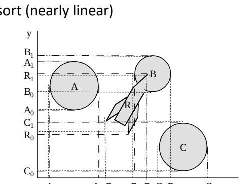
- To address the n^2 problem...
- Partition space so only test objects in same cell



- In best case (uniform distribution) reduces n^2 to linear
  - Can happen for uniform size, density objects (e.g., cloth/fluids)
- In worst case (all objects in same cell) no improvement

Plane Sweep for Collision Testing

- *Observation: many* moveable objects stay in one place most of the time
- *Sort* bounds along axes (expensive to do, so do just once!)
- Only adjacent sorted objects which overlap on all axes need to be checked further
- Since many objects don’t move, can keep sort up to date very cheaply with bubblesort (nearly linear)



Outline

- Introduction (done)
- Kinematics (done)
- Rigid Body Simulation (done)
- The Firing Solution (done)
- Collision Detection (done)
- Ragdoll Physics (next)
- PhysX

What is Ragdoll Physics?

- Procedural animation often used as replacement for traditional (static) death animation
  - Generated by code, not hand
  - Using physics constraints on body limbs & joints in real-time

Still from early animation using ragdoll physics

https://en.wikipedia.org/wiki/Ragdoll_physics

http://www.freeonlinegames.com/game/ragdoll-physics-2

Diablo 3 Ragdolls

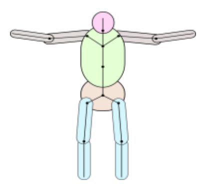
"How to Smack a Demon"

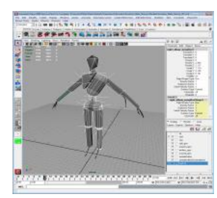
Erin Catto

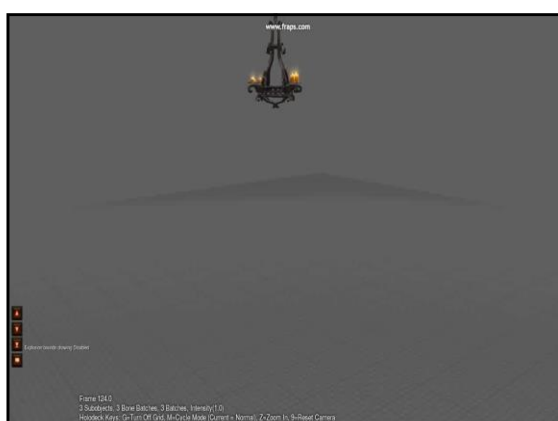
(*Game Developer's Conference*, San Francisco, California, USA, 2013)

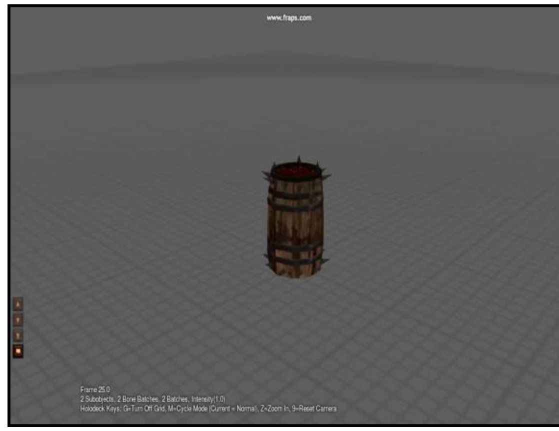
Physics Programmer for Diablo 3 (Blizzard, 2012)

A ragdoll is a collection of collision shapes connected to bones

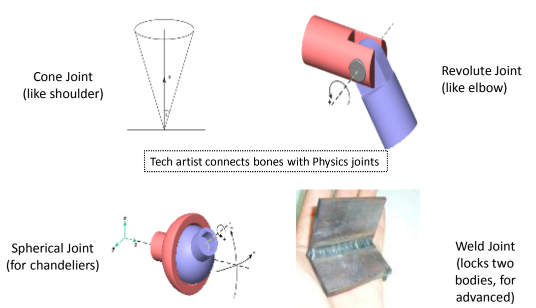




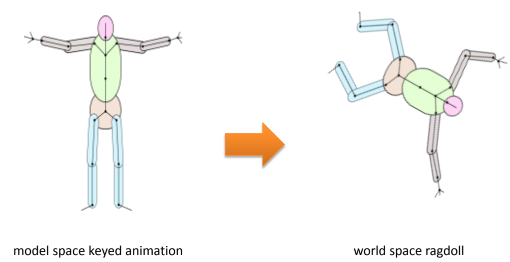




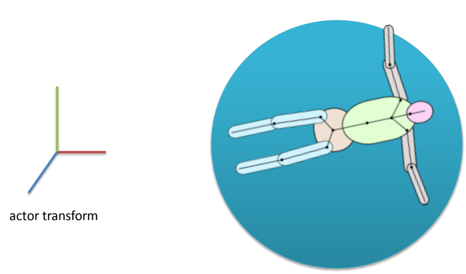
Physics joints connect two bones



We use the ragdoll bodies to adjust the pose



Update the actor bounding sphere using the bone transforms



Partial ragdolls add flavor to living characters

Not just for death and destruction

More Physics We Are Not Covering

- Collision response
- Conservation of momentum
- Elastic collisions
- Non-elastic collisions – coefficient of restitution
- Rigid body simulation (vs. point masses)
- Joints as constraints to motion
- Soft body simulation

[see excellent book by Millington, "Game Physics Engine Development", MK, 2007]

62

Outline

- Introduction (done)
- Kinematics (done)
- Rigid Body Simulation (done)
- The Firing Solution (done)
- Collision Detection (done)
- Ragdoll Physics (done)
- PhysX (next)

63

PhysX Overview

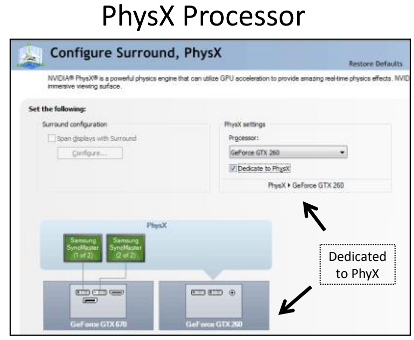
- Developed by NVIDIA for C++ applications
- Windows, Mac, Linux, Playstation, Xbox, Android, Apple iOS and Wii
- Simulate
  - Fluids
  - Soft bodies (cloth, hair)
  - Rigid bodies (boxes, bones)

Why Does NVIDIA Make Physics Software?

- NVIDIA is mainly known as a developer and manufacturer of graphics hardware (GPU's)
- So taking advantage of GPU for hardware acceleration of their physics engine
  - Algorithms can be tuned to their hardware
  - Giving a competitive advantage over other GPU manufacturers

65

Configure Video Card as Dedicated PhysX Processor



What Algorithms Does PhysX Use?

- Hard to know exactly, because algorithm details are NVIDIA's intellectual property (IP)
- However from various forums and clues, it is clear PhysX uses:
  - Both sweep and overlap collision detection
  - AABB and OBBT and (both axis-aligned and oriented bounding bounding box trees)
  - Constraints: hinges, springs, etc.
  - and lots of other hairy stuff, see

https://devtalk.nvidia.com/default/board/66/physx-and-physics-modeling/

67

Rocket Sled

CES 2010, **Rocket Sled** demonstrates both graphics and physics computing capabilities of new **GF100** (Fermi) GPUs.

Raging Rapids Ride

Graphics ok, but with **intensive and complex real-time fluid** simulation

Havok Cloth

PhysX competitor bought by Microsoft

How to Use PhsyX

- General documentation NVIDIA® PhysX® SDK Documentation
  http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Index.html
- UE4 guide – PhysX, Integrating PhysX Code into Your Project (by Rama)
  https://wiki.unrealengine.com/PhysX,_Integrating_PhysX_Code_into_Your_Project

```
#include "YourGame.h"
#include "HappyDestructible.h"

//~~~~~~~~~~~~~~~~~~~~~~~~
//        PhysX
#include "PhysXIncludes.h"
//~~~~~~~~~~~~~~~~~~~~~~~~

AHappyDestructible::AHappyDestructible(const class FPostConstructInitializeProperties& PCIP) :
{

}

//Tick
void AHappyDestructible::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
	//~~~~~~~~~~~~


	//Draw All Centers to the screen!
	#if WITH_PHYSX
	ScreenMsg("Got into PhysX!!!");
```