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) \( d = ut + \frac{1}{2}at^2 \)
  4) \( v^2 = u^2 + 2ad \)

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

Example:
- \( d = ut \)
- \( d = ut + \frac{1}{2}at^2 \)
- \( u = 20 \, \text{m/s}, \, t = 300 \, \text{s} \)
- \( d = 20 \times 3000 = 60000 \, \text{m} \)
- \( d = 0 \times 3 + 0.5 \times 4 \times 9 = 18 \, \text{m} \)

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?

\[ v^2 = u^2 + 2ad \]

\( u = 10 \, \text{m/sec} \) (initial speed upward)
\( a = -10 \, \text{m/sec}^2 \) (approx gravity)
\( v = 0 \, \text{m/sec} \) (at top of flight)

\[ 0 = 10^2 + 2(-10)d \]
\[ d = 5 \, \text{meters} \]

(Note, answer independent of mass of ball)

Doing It In 3D

- Mathematically, consider all quantities involving position to be vectors:
  - \( d = vt \)
  - \( v = u + at \)
  - \( d = ut + \frac{1}{2}at^2 \)
- 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.

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)

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/sec²

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

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

Computing Kinematics in Real Time

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

function update () { // in game loop
    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
}
```

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)

Outline

- Introduction (done)
- Kinematics (done)
- Rigid Body Simulation (next)
- The Firing Solution
- Collision Detection
- Ragdoll Physics
- PhysX
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., \( d = ut + \frac{1}{2}at^2 \))
  
  - 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? \( \rightarrow \) Numerical simulation

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

- Suppose know position \( (p) \), velocity \( (v) \) and acceleration \( (a) \) at time \( (t) \) and frame time \( (\Delta t) \)
  
  - Want position at next frame time \( (t_{n+1}) \)
    
    - Don’t know exactly how forces are affecting (wind, friction, gravity…)
  
  - Can compute:
    
    \[
    F_{n+1} = F_{n} + a \times \Delta t
    \]
    
    \[
    v_{n+1} = v_{n} + a \times \Delta t
    \]
  
  - Can do beyond, but always higher terms

- What is this doing? \( \rightarrow \) Instead of integrating a curve, sum over discrete time-slices

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 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 \( t+\Delta t \)
  
- View: \( S(t+\Delta t) = S(t) + \Delta t \frac{dS}{dt} S(t) \)

Explicit Euler Integration Example (1 of 2)

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

Explicit Euler Integration (2 of 2)

- Write numerical integrator over arbitrary properties as change over time
  
  - For single particle, \( S = (mV, p) \) and \( \frac{dS}{dt} S = (F, V) \)
    
    - Derivative of position \( (p) \) is velocity \( (V) \)
    
    - Derivative of momentum (mass \( m \times V) \) is force \( (F) \)
    
    - i.e., how momentum changes with respect to time
  
  - Derivative of linear momentum \( (m \times V) \) is force \( (F) \)

- Integrate state vector of length \( N \)
  
  - Vector ExplicitEuler(int \( N \), Vector prior_S, Vector deriv_S, float \( \Delta t \))
    
    - Vector new_S[N];
    
    for (int \( i = 0 \); \( i < N; i++ \))
      
      new_S[i] = prior_S[i] + deriv_S[i] * delta_t;
    
    return new_S;

- Note, for 3D, \( m \) and \( p \) have 3 values each
  
  - \( S(t) = (m_{V_x, V_y, V_z}, p_{x, p_y, p_z}) \)
  
  - \( \frac{dS}{dt} S(t) = (F_{x, V_x}, F_{y, V_y}, F_{z, V_z}) \)
In other words, unstable simulations behave

typically, more state kept. Stable within bounds.

Reduce time step, \( \Delta t \) (next slide)

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 (Verlet, Runge-Kutta or others).
  - Typically, more state kept. Stable within bounds.

Truncation Error Example

\[ (VX(t)) - Truncation Error \]

Trade-off: truncation error and computation interval

Guidelines: Step more often than frame rate (otherwise, no update?)

\[ \Delta t \text{ under 30 ms} (20 \text{ ms a good choice}) \]
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

```
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()
    while (lag > delta)          // repeat until caught up
        lag += (now - previous)   // additional lag
    t += delta
    simulate(t)                 // note: kinematics. If dynamic, use delta
    lag -= delta
    previous = now              // simulation caught up to current time
}
```

Outline

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

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.
    - Distance (d) = velocity (u) * time (t) + (1/2) * acceleration (a) * time^2
      - u = muzzle velocity vector
      - a = g = -9.8 m/sec^2
      - d = ut + (1/2)at^2

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
  - 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 (angle solution)
  - Target is closer than maximum range (two possible solutions)

The Firing Solution (2 of 3)

- Usually choose short time trajectory
  - Gives target less time to escape
  - Unless shooting over wall, etc.
The Firing Solution (3 of 3)

Function firingSolution (start, target, muzzle_v, gravity) {
  // Calculate vector back from target to start
  delta = target - start
  // The only solution for quadratic equation:
  a = gravity * gravity
  b = -2 * ( gravity + delta + muzzle_v )
  c = delta * delta + delta
  // Check for real solutions
  if ( a == 0 ) return null
  // Find short and long times to target
  t1 = sqrt ( ( -b + disc ) / 2 * a )
  t2 = sqrt ( ( -b - disc ) / 2 * a )
  // Pick shortest valid time to target (real valued)
  if ( t1 < 0 ) && ( t2 < 0 ) return null
  // Real valued solutions
  if ( t1 < 0 ) return t2
  // No valid times
  if ( t1 < 0 ) return null
  // Calculate vector back from target to start
  delta = target - start
  // Find overlap vector
  return ( delta - gravity * ttt ) / ( 2 * muzzle_v )
}

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: 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

Overlap Testing

- 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

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)

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)
- 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

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

Speeding Up Collision Detection
- Bounding Volumes
  - Oriented
  - Hierarchical
- Partitioning
- Plane Sweep

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


Diablo 3 Ragdolls

“How to Smack a Demon”

Erin Catto

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

A ragdoll is a collection of collision shapes connected to bones
Physics joints connect two bones

- Cone Joint (like shoulder)
- Spherical Joint (for chandeliers)
- Revolute Joint (like elbow)
- Weld Joint (locks two bodies, for advanced)

We use the ragdoll bodies to adjust the pose

- Model space keyed animation
- World space ragdoll

Update the actor bounding sphere using the bone transforms

Partial ragdolls add flavor to living characters

- Not just for death and destruction

Tech artist connects bones with Physics joints.
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]

Outline

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

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

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 box trees)
  – Constraints: hinges, springs, etc.
  – and lots of other hairy stuff, see https://devtalk.nvidia.com/default/board/66/physx-and-physics-modeling/

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 PhysX

• General documentation NVIDIA® PhysX® SDK Documentation
  http://dev.nvidia.com/gameworks/content/gameworkslibrary/physxguide/index.html

• UE4 guide – PhysX, Integrating PhysX Code into Your Project (by Rama)
  https://wiki.unrealengine.com/PhysX_Integrating_PhysX_Code_into_Your_Project