

Basic Game Physics

Technical Game Development II

Professor Charles Rich
Computer Science Department
rich@wpi.edu

[some material provided by Mark Claypool]

Introduction

- *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 (cont'd)

- *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*
 - integrated/native, e.g., C4
 - pluggable, e.g.,
 - C4+PhysX
 - **jME+ODE** (via jME Physics)



IMGD 4000 (D 10)

5

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
- *Examples*
 - kinematics and dynamics
 - projectile motion
 - collision detection and response



IMGD 4000 (D 10)

6

Kinematics

- Study of the motion of objects *without* taking into account mass or force
- Basic quantities: **position, time**
- Basic equations:

$$d = vt$$

$$v = u + at$$

$$d = ut + at^2/2$$

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

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

Kinematics (cont'd)

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?

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

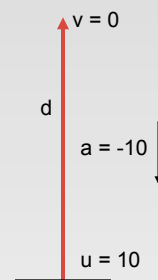
$$u = 10 \text{ m/sec}$$

$$a = -10 \text{ m/sec}^2 \text{ (approx due to gravity)}$$

$$v = 0 \text{ m/sec (at top of flight)}$$

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

$$d = 5 \text{ m}$$



(note answer independent of mass of ball)

Computing Kinematics in Real Time

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

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

function simulate (t) {
  d = (u + (0.5 * a * t)) * t
  move object to p + d
}
```



$$d = ut + at^2/2$$

Problem: Number of calls and time values to `simulate` depend on (changing) **frame rate**



IMGD 4000 (D 10)

9

Frame Rate Independence

- Complex numerical simulations used in physics engines are very 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
- **Solution:** control simulation interval separately



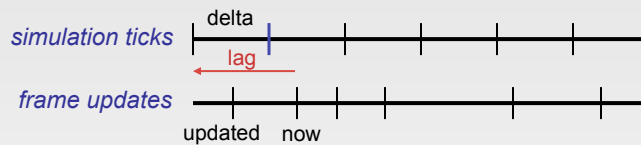
IMGD 4000 (D 10)

10

Frame Rate Independence

```
delta = 0.02 // physics simulation interval (sec)
lag = 0      // physics lag
updated = 0  // time of last update

function update () { // in render loop
  now = getTime()
  t = (updated - start) - lag
  lag = lag + (now - updated)
  while ( lag > delta )
    simulate(t)
    t = t + delta
    lag = lag - delta
  updated = now
}
```



IMGD 4000 (D 10)

11

Doing It In 3D

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

$$\mathbf{d} = \mathbf{v}t$$

$$\mathbf{v} = \mathbf{u} + \mathbf{a}t$$

$$\mathbf{d} = \mathbf{u}t + \mathbf{a}t^2/2$$

(Note these are all scalar products, so essentially calculations are performed independently in each dimension.)

- Computationally, using appropriate 3-element vector datatype



IMGD 4000 (D 10)

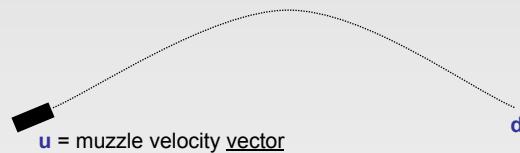
12

The Firing Solution

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

$\mathbf{a} = [0, 0, -9.8] \text{ m/sec}^2$
(but typically use higher value, e.g. -18)

$$\mathbf{d} = \mathbf{u}t + \mathbf{a}t^2/2$$



Given \mathbf{d} , solve for \mathbf{u} .



IMGD 4000 (D 10)

13

The Firing Solution

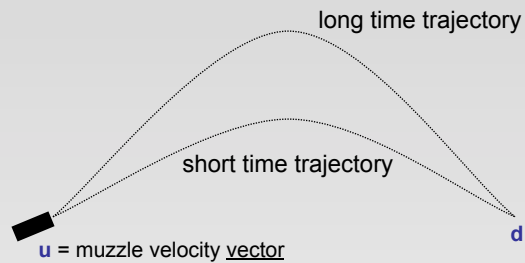
- In most typical game situation, the *magnitude* of \mathbf{u} is fixed and we only need to know its relative components (orientation)
- After a lot of hairy math [see Millington 3.5.3], it turns out there are three relevant cases:
 - target is out of range (no solutions)
 - target is at exact maximum range (single solution)
 - target is closer than maximum range (two possible solutions)



IMGD 4000 (D 10)

14

The Firing Solution



- Usually choose short time trajectory
 - gives target less time to escape $u = (2d - at^2) / 2xt$
 - unless shooting over wall, etc. *where x = max muzzle speed*

```
function firingSolution (d, x, gravity) {  
    // real-valued coefficients of quadratic  
    a = gravity * gravity  
    b = -4 * (gravity * d + x*x)  
    c = 4 * d * d  
  
    // check for no real solutions  
    if ( 4*a*c > b*b ) return null  
  
    // find short and long times  
    disc = sqrt(b*b - 4*a*c)  
    t1 = sqrt((-b + disc) / 2*a)  
    t2 = sqrt((-b - disc) / 2*a)  
    if ( t1 < 0 )  
        if ( t2 < 0 ) return null  
        else t = t2  
    else if ( t2 < 0 ) t = t1  
    else t = min(t1, t2)  
  
    // return firing vector  
    return (2*d - gravity*t*t) / (2*x*x)  
}
```

Note scalar product of two vectors using $$, e.g.,*

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

Dynamics

- Notice that the 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
- fine for some casual games (esp. with appropriate animations)

Newton's Second Law

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

at each moment in time:

\mathbf{F} = force vector, Newton's

m = mass (intrinsic property of matter), kg

\mathbf{a} = acceleration vector, m/sec²

This equation is the fundamental driver of all physics simulations:

- force causes acceleration
- acceleration causes change in velocity
- velocity causes change in position

How Are Forces Applied?

- Without contact
 - gravity
 - wind (if not modeling air particles)
 - magic
- Usually involves contact
 - collision (rebound)
 - friction (rolling, sliding)
- Dynamic (force) modeling also used for autonomous steering behaviors



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 a collision will occur in the future

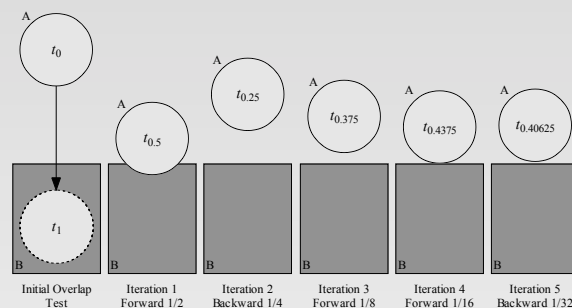


Overlap Testing

- 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

Overlap Testing: Finding Collision Time

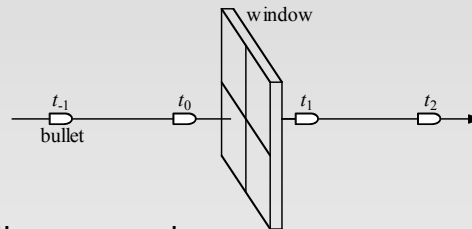
- Calculated by doing “binary search” in time, moving object back and forth by 1/2 steps (bisections)



- 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 tick than *thinnest object*
 - may require reducing simulation step size (adds computation overhead)

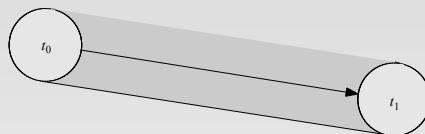


IMGD 4000 (D 10)

25

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 (no searching)
 - resolve collision
 - simulate remaining time step(s)
 - works for bullet/window example



IMGD 4000 (D 10)

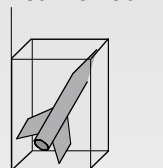
26

Speeding Up Collision Detection

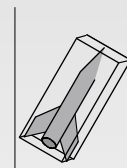
- Bounding Volumes
 - Oriented
 - Hierarchical
- Partitioning
- Plane Sweep

Bounding Volumes

- 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
- 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

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
- Techniques can be combined
 - e.g., hierarchical oriented bounding boxes (OBBTree) in jME

Oriented Bounding Box Tree

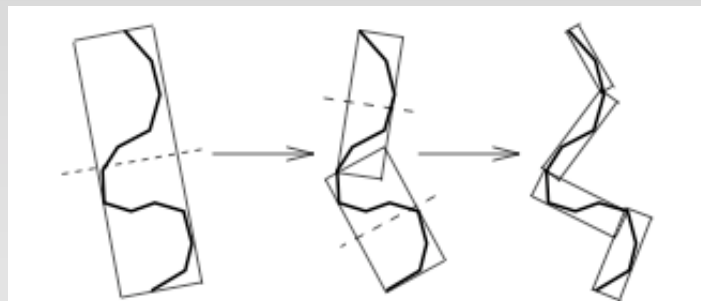
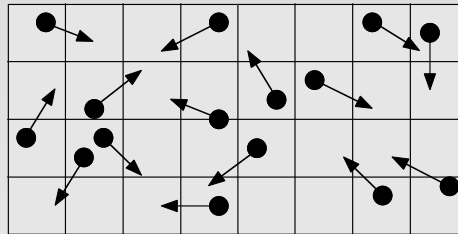


Figure 1: Building the OBBTree: recursively partition the bounded polygons and bound the resulting groups.

[Gottschalk, Lin, Minocha, SIGGRAPH '96]

Partitioning for Collision Testing

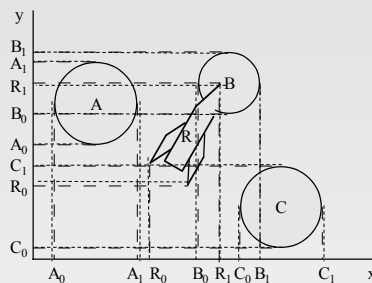
- To address the n^2 problem...
- Partition space so only test objects in same cell



- In **best case** (uniform distribution) reduces n^2 to linear
- In **worst case** (all objects in same cell) no improvement

Plane Sweep for Collision Testing

- **Observation:** a lot of objects stay in one place
- **Sort** bounds along axes (expensive to do 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)



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)
- Soft body simulation
 - spring-mass-damper dynamics

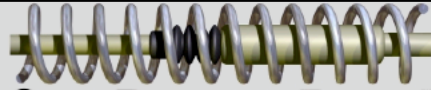
[see excellent recent book by Millington, "Game Physics Engine Development", MK, 2007]



Open Dynamics Engine

- Brief case study of a complete physics engine
- Overview from author (Russell Smith)
- jME Physics interface

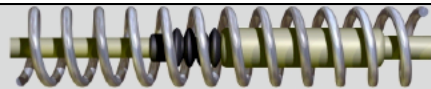




OPEN DYNAMICS ENGINE™

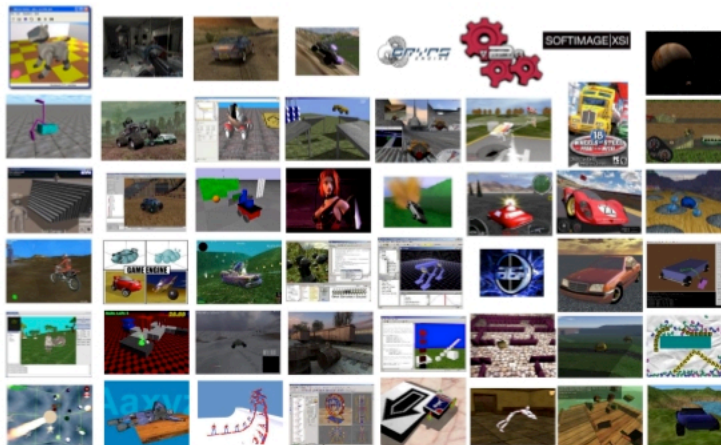
- My rigid body simulation library.
 - Many others: Havok, PhysX (Novodex, Meqon), SD/Fast...
- A platform for research.
 - Simulation algorithms, simulation applications.
- Open source (BSD license).
 - Dynamics should be ubiquitous: encourage innovation.
 - Closed source libraries constrain users: endless customization and integration hassles.
 - Why customization: ODE → SoftImage XSI → ILM
 - Used in "Eternal Sunshine of the Spotless Mind".

Copyright ©2004 Russell Smith



OPEN DYNAMICS ENGINE™

- Over 1000 users: widely used in games, game engines, robot simulation, 3D animation.



Copyright ©2004 Russell Smith

Why Simulation is Hard

- Modeling real-world mechanisms is hard.
- Unexpected behavior.
 - Hard-to-debug numerical explosions, jitter, poor contact behavior and general unexpected weirdness.
- APIs force the user to learn arcane concepts.
 - Many simulation primitives not intuitive – angular velocity, inertia tensors.
- Too slow for big models.
- Force-based modeling is tricky.
- Too many numerical parameters to tune.
 - Many modeling / numerical approximations used, all with their own tradeoff parameters. Little guidance available, need to experiment.

Copyright ©2004 Russell Smith

API Issues

- In the old days it was harder:
 - MDH parameters, weird reference frames, text file configuration, poor documentation, implementation exposed.
- Now we think about the user experience.
 - Absolute positions, utility functions (e.g. for rotation), interactive setup (3D tools), documentation, API consistency, only essential concepts in the API.
- Still lots of room for improvement.
 - Constructive modeling: glue, split, clone, deform, etc.
 - Dynamics debuggers – identify model physical / numerical errors.
 - Standardized data formats.

Copyright ©2004 Russell Smith

Speed

- Higher speed → real time simulation of more complex worlds.
- Big-matrix methods need lots of Optimization.
 - Coding tricks: minimize memory traffic, cache-friendly algorithms, pipelining, SIMD.
 - Parameterized code, search for efficient parameters (ATLAS).
- CPU budgeting.
 - Iterative methods allow us to cap effort per frame.
 - But accuracy is an issue.
- Parallelization.
 - Problem is not coarse grained – so clusters don't work well.
 - Parallel direct factorization – only for *large* problems.
 - Iterative techniques the easiest to parallelize.
 - ODE QuickStep inner loop: 3x speedup using 6 CPUs – [SGI Altix].

Copyright ©2004 Russell Smith

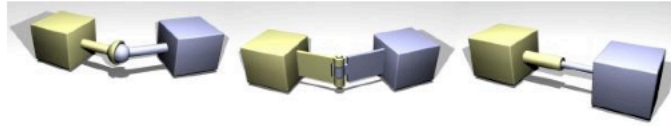
Speed

- Higher speed → real time simulation of more complex worlds.
- Big-matrix methods need lots of Optimization.
 - Coding tricks: minimize memory traffic, cache-friendly algorithms, pipelining, SIMD.
 - Parameterized code, search for efficient parameters (ATLAS).
- CPU budgeting.
 - Iterative methods allow us to cap effort per frame.
 - But accuracy is an issue.
- Parallelization.
 - Problem is not coarse grained – so clusters don't work well.
 - Parallel direct factorization – only for *large* problems.
 - Iterative techniques the easiest to parallelize.
 - ODE QuickStep inner loop: 3x speedup using 6 CPUs – [SGI Altix].

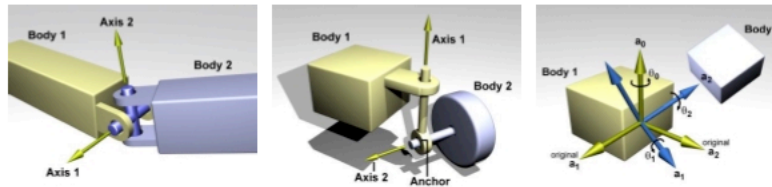
Copyright ©2004 Russell Smith

Joints are Constraint

ODE's "robot" joints:



ODE's special purpose joints:



Copyright ©2004 Russell Smith

Constraint Based Modeling (good!)

- Velocity / acceleration [in]equality constraints (LCP):

$$f_u(\mathbf{v}) = 0, f_v(\mathbf{v}) \geq 0 \quad \text{or} \quad f_a(\mathbf{a}) = 0, f_b(\mathbf{a}) \geq 0$$

- Contacts and friction.
 - Relative velocity & force normal to contact surface ≥ 0 .
 - Tangential forces limited by Coulomb friction (various models).
 - Constraint modeling is now commonplace for contacts.
- Actuators and brakes.
 - Joint velocity = v , but don't apply too much force.
- Simulator enforces constraints automatically
 - No parameters to tune.
 - Integration problems hidden away.
- Can also model stiff springs, e.g. suspensions.

Copyright ©2004 Russell Smith

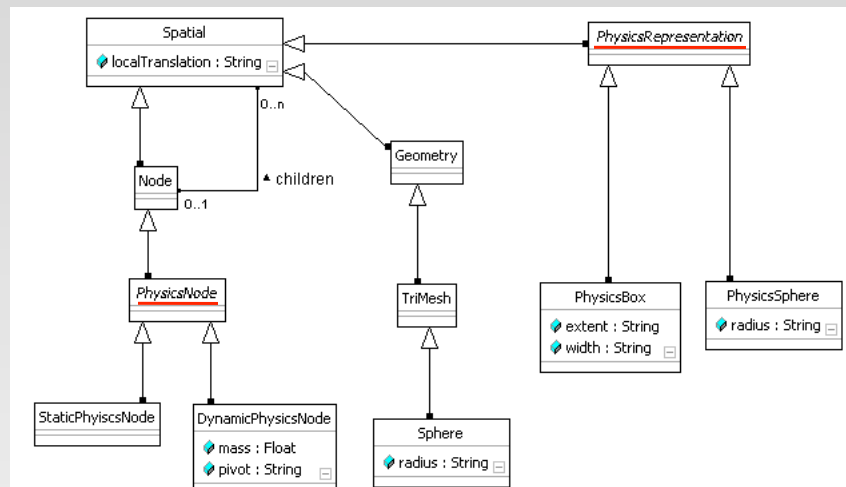
Constraint Also Good For:

- **Mechanisms**
 - Gears, linked platforms, steering geometry, suspensions, roller coasters, weird joints (e.g. screw joints), etc.
- **Modeling**
 - Contact geometry, various kinds of friction, various actuators, spongy / flexible joints, etc.
- **Disadvantages:**
 - More expensive than forces.
 - Must factor a matrix of constraint information.
 - More mathematically difficult to formulate.
 - But “intuitive” guidance available, see Game Gems IV ch3.4.



Copyright ©2004 Russell Smith

jME/ODE Integration (jME Physics 2.1)



<http://code.google.com/p/jmephysics/>

jME Physics (2.1)

