Everything you ever wanted to know about collision detection

(and as much about collision response
as I can figure out by Wednesday)

By Ryan Schmidt, ryansc@cpsc.ucalgary.ca

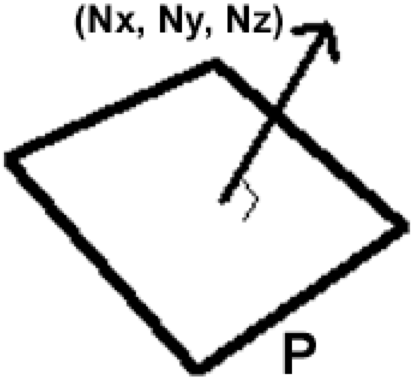
Where to find this on the Interweb

- http://www.cpsc.ucalgary.ca/~ryansc
- Lots of links to software, web articles, and a bunch of papers in PDF format
- You can also email me, ryansc@cpsc.ucalgary.ca - I'll try to help you if I can.

What you need to know

- Basic geometry
  - vectors, points, homogenous coordinates, affine transformations, dot product, cross product, vector projections, normals, planes
- math helps…
  - Linear algebra, calculus, differential equations

Calculating Plane Equations



- A 3D Plane is defined by a normal and a distance along that normal
- Plane Equation: $(Nx, Ny, Nz) \bullet (x, y, z) + d = 0$
- Find d: $(Nx, Ny, Nz) \bullet (Px, Py, Pz) = -d$
- For test point (x,y,z), if plane equation
  - \> 0: point on ‘front’ side (in direction of normal),
  - < 0: on ‘back’ side
  - = 0: directly on plane
- 2D Line ‘Normal’: negate rise and run, find d using the same method

So where do ya start….?

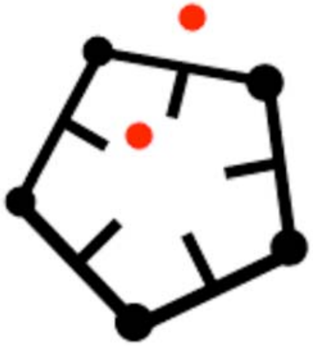
- First you have to detect collisions
  - With discrete timesteps, every frame you check to see if objects are intersecting (overlapping)
- Testing if your model's actual volume overlaps another's is too slow
- Use bounding volumes (BV's) to approximate each object's real volume

Bounding Volumes?

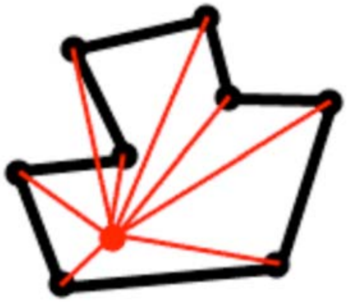
- Convex-ness is important*
- spheres, cylinders, boxes, polyhedra, etc.
- Really you are only going to use spheres, boxes, and polyhedra (…and probably not polyhedra)
- Spheres are mostly used for fast culling
- For boxes and polyhedra, most intersection tests start with point inside-outside tests
  - That's why convexity matters. There is no general inside-outside test for a 3D concave polyhedron.

* 40 Helens agree…

2D Point Inside-Outside Tests

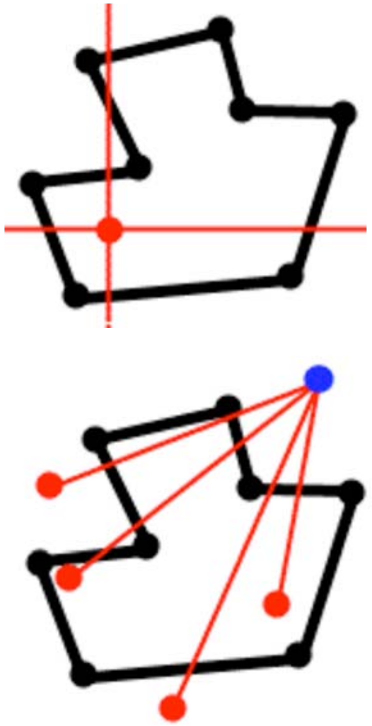


- Convex Polygon Test
  - Test point has to be on same side of all edges
- Concave Polygon Tests
  - 360 degree angle summation
  - Compute angles between test point and each vertex, inside if they sum to 360
  - Slow, dot product and acos for each angle!



More Concave Polygon Tests

- Quadrant Method (see Gamasutra article)
  - Translate poly so test point is origin
  - Walk around polygon edges, find axis crossings
  - +1 for CW crossing, -1 for CCW crossing,
  - Diagonal crossings are +2/-2
  - Keep running total, point inside if final total is +4/-4
- Edge Cross Test (see Graphics Gems IV)
  - Take line from test point to a point outside polygon
  - Count polygon edge crossings
  - Even # of crossings, point is outside
  - Odd # of crossings, point is inside
- These two are about the same speed



Closest point on a line

- Handy for all sorts of things…

$$A = P_2 - P_1$$

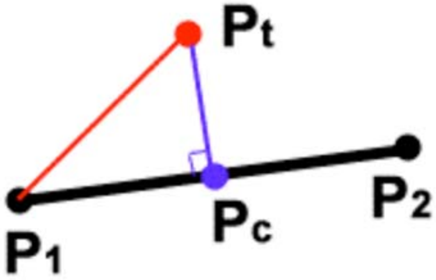
$$B = P_1 - P_t$$

$$C = P_2 - P_t$$

$$if\,(A \bullet B \leq 0) \quad P_c = P_1$$

$$else\,if\,(A \bullet C \leq 0) \quad P_c = P_2$$

$$else \quad P_c = P_1 + \frac{(P_1 - P_1) * (B \bullet A)}{(B \bullet A) + (C \bullet A)}$$



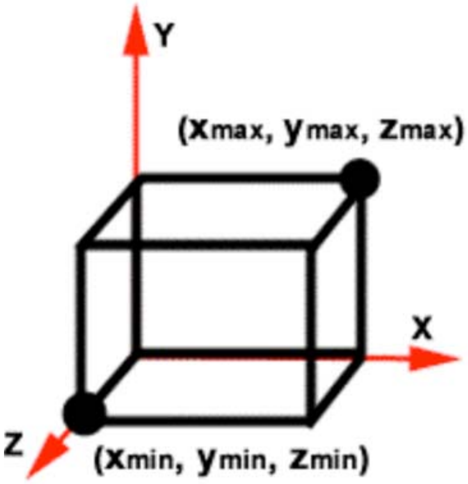
Spheres as Bounding Volumes

- Simplest 3D Bounding Volume
  - Center point and radius
- Point in/out test:
  - Calculate distance between test point and center point
  - If distance <= radius, point is inside
  - You can save a square root by calculating the squared distance and comparing with the squared radius !!!
  - (this makes things a lot faster)
- It is **ALWAYS** worth it to do a sphere test before any more complicated test. **ALWAYS**. I said **ALWAYS**.

Axis-Aligned Bounding Boxes

- Specified as two points:

$$(x_{\min}, y_{\min}, z_{\min}), (x_{\max}, y_{\max}, z_{\max})$$



- Normals are easy to calculate
- Simple point-inside test:

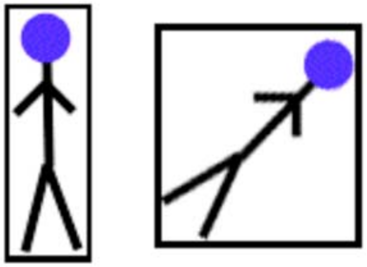
$$x_{\min} \leq x \leq x_{\max}$$

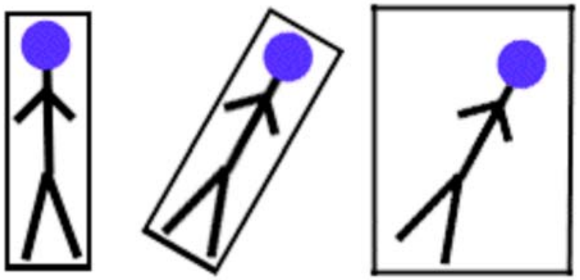
$$y_{\min} \leq y \leq y_{\max}$$

$$z_{\min} \leq z \leq z_{\max}$$

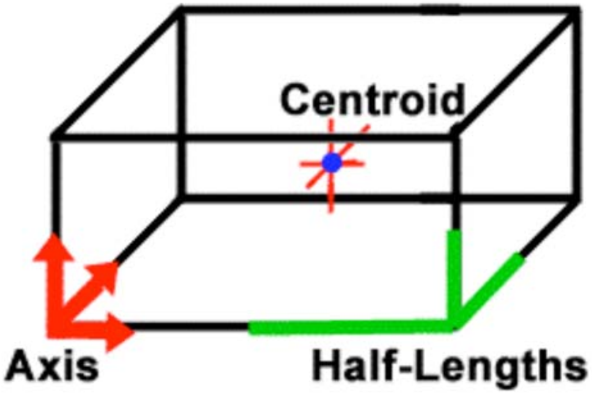
Problems With AABB's

- Not very efficient
- Rotation can be complicated
  - Must rotate all 8 points of box
  - Other option is to rotate model and rebuild AABB, but this is not efficient

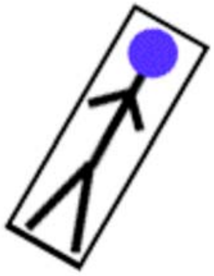




Oriented Bounding Boxes



- Center point, 3 normalized axis, 3 edge half-lengths
- Can store as 8 points, sometimes more efficient
  - Can become not-a-box after transformations
- Axis are the 3 face normals
- Better at bounding than spheres and AABB's



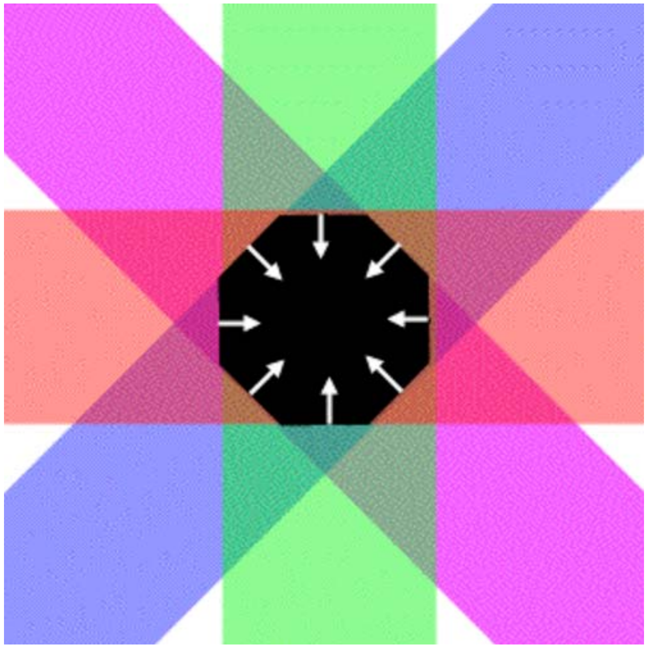
OBB Point Inside/Outside Tests

- Plane Equations Test
  - Plug test point into plane equation for all 6 faces
  - If all test results have the same sign, the point is inside (which sign depends on normal orientation, but really doesn't matter)
- Smart Plane Equations Test
  - Each pair of opposing faces has same normal, only d changes
  - Test point against d intervals – down to 3 plane tests
- Possibly Clever Change-of-Basis Test*
  - Transform point into OBB basis (use the OBB axis)
  - Now do AABB test on point (!)
  - Change of basis: $P' = B_{axis} \bullet P_{test}$

* This just occurred to me while I was writing,
So it might not actually work

k-DOP’s

- k-Discrete Oriented Polytype
- Set of k/2 infinite ‘slabs’
  - A slab is a normal and a d-interval
- Intersection of all slabs forms a convex polyhedra
- OBB and AABB are 6-DOP’s
- Same intersection tests as OBB
  - There is an even faster test if all your objects have the same k and same slab normals
- Better bounds than OBB



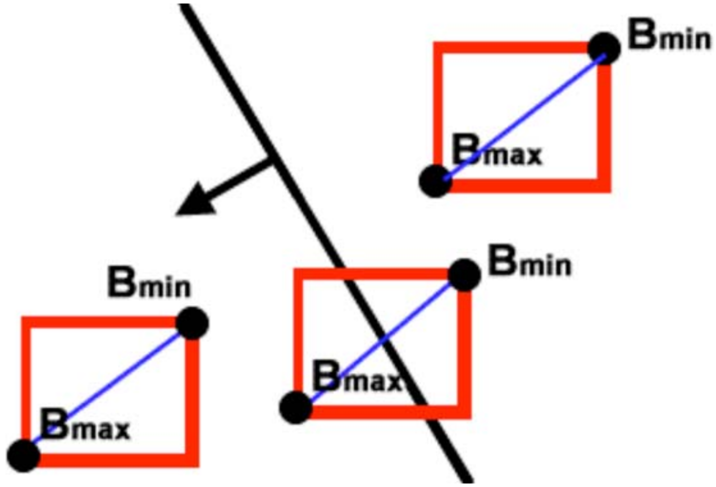
Plane Intersection Tests

- Planes are good for all sorts of things
  - Boundaries between level areas, ‘finish lines’, track walls, powerups, etc
- Basis of BSP (Binary Space Partition) Trees
  - Used heavily in game engines like Quake(1, 2,…∞)
  - They PARTITION space in half
  - In half…that’s why they’re binary…punk*

---

* Sorry, I had to fill up the rest of this slide somehow, and just making the font bigger makes me feel like a fraud…

AABB/Plane Test



- An AABB has 4 diagonals
- Find the diagonal most closely aligned with the plane normal
- Check if the diagonal crosses the plane
- You can be clever again…
  - If Bmin is on the positive side, then Bmax is guaranteed to be positive as well

OBB/Plane Test

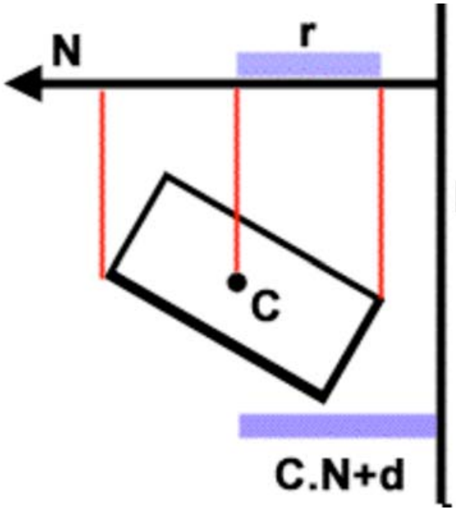
- Method 1: transform the plane normal into the basis of the OBB and do the AABB test on N`
  - $N' = (b_x, b_y, b_z) \cdot N$
- Method 2: project OBB axis onto plane normal

h_x, h_y, h_z are OBB half-spaces

b_x, b_y, b_z is OBB basis, C is centroid

$$r = \left|h_x N \bullet b_x\right| + \left|h_y N \bullet b_y\right| + \left|h_z N \bullet b_z\right|$$

if $\left|C \bullet N + d\right| > r$, no intersection



Other Plane-ish Tests

- Plane-Plane
  - Planes are infinite. They intersect unless they are parallel.
  - You can build an arbitrary polyhedra using a bunch of planes (just make sure it is closed….)
- Triangle-Triangle
  - Many, many different ways to do this
  - Use your napster machine to find code

Bounding Volume Intersection Tests

- Mostly based on point in/out tests
- Simplest Test: Sphere/Sphere test

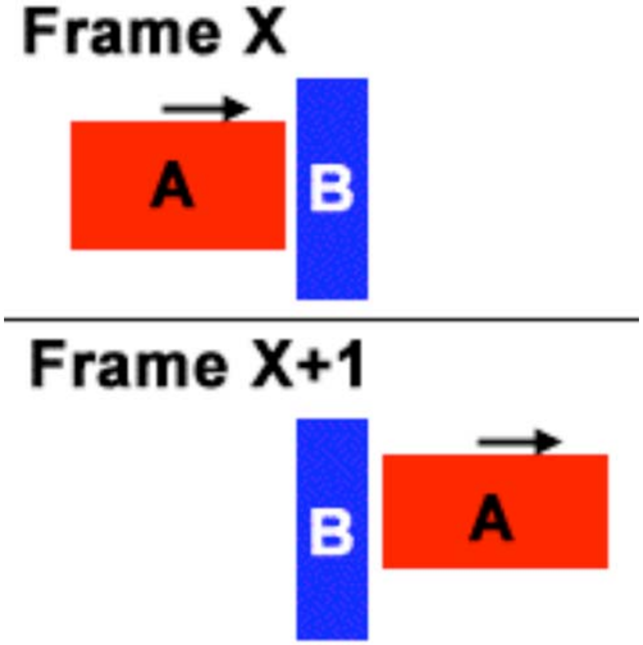
$$\text{Sphere 1}: \left[c_1(x, y, z), r_1\right]$$

$$\text{Sphere 2}: \left[c_2(x, y, z), r_2\right]$$

$$\text{if } \left\| c_1 - c_2 \right\|^2 < (r_1 + r_2)^2$$

$$\quad \text{spheres are intersecting}$$

A fundamental problem



- If timestep is large and A is moving quickly, it can pass through B in one frame
- No collision is detected
- Can solve by doing CD in 4 dimensions (4th is time)
  - This is complicated
- Can contain box over time and test that

Separating Axis Theorem

- For any two arbitrary, convex, disjoint polyhedra A and B, there exists a separating axis where the projections of the polyhedra for intervals on the axis and the projections are disjoint
- Lemma: if A and B are disjoint they can be separated by an axis that is orthogonal to either:
  1) a face of A
  2) a face of B
  3) an edge from each polyhedron

Sphere/AABB Intersection

- Algorithm:

```
$d = 0$
$\text{for}(i = 0; i < 3; ++i)$
    $\text{if } (c[i] < \min[i])$
        $d = d + (c[i] - \min[i])^2$
    $\text{else if } (c[i] > \max[i])$
        $d = d + (c[i] + \max[i])^2$
$\text{if } (d \leq r^2)$
    intersection
```

- For OBB, transform sphere center into OBB basis and apply AABB test

AABB/AABB Test

- Each AABB defines 3 intervals in x,y,z axis
- If any of these intervals overlap, the AABB’s are intersecting

OBB/OBB – Separating Axis Theorem Test

- Test 15 axis with with SAT
  - 3 faces of A, 3 faces of B
  - 9 edge combinations between A and B
- See OBBTree paper for derivation of tests and possible optimizations (on web)
- Most efficient OBB/OBB test
  - it has no degenerate conditions. This matters.
- Not so good for doing dynamics
  - the test doesn’t tell us which points/edges are intersecting

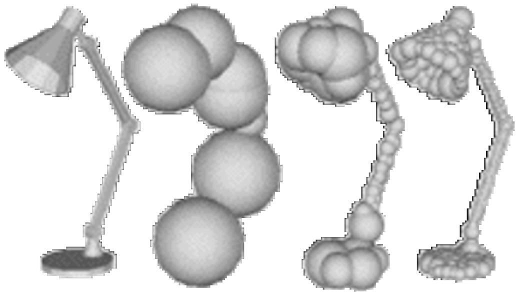
OBB/OBB – Geometric Test

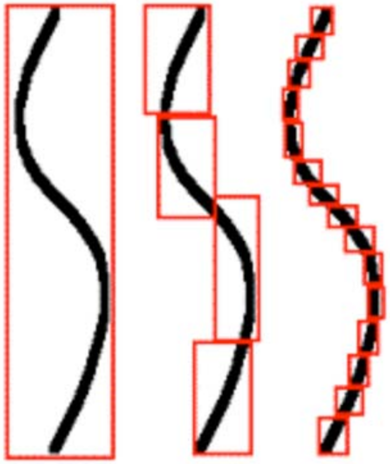
- To check if A is intersecting B:
  - Check if any vertex of A is inside B
  - Check if any edge of A intersects a face of B
- Repeat tests with B against A
- Face/Face tests
  - It is possible for two boxes to intersect but fail the vertex/box and edge/face tests.
  - Catch this by testing face centroids against boxes
  - Very unlikely to happen, usually ignored

Heirarchical Bounding Volumes

- Sphere Trees, AABB Trees, OBB Trees
  - Gran Turismo used Sphere Trees
- Trees are built automagically
  - Usually precomputed, fitting is expensive
- Accurate bounding of concave objects
  - Down to polygon level if necessary
  - Still very fast
- See papers on-line

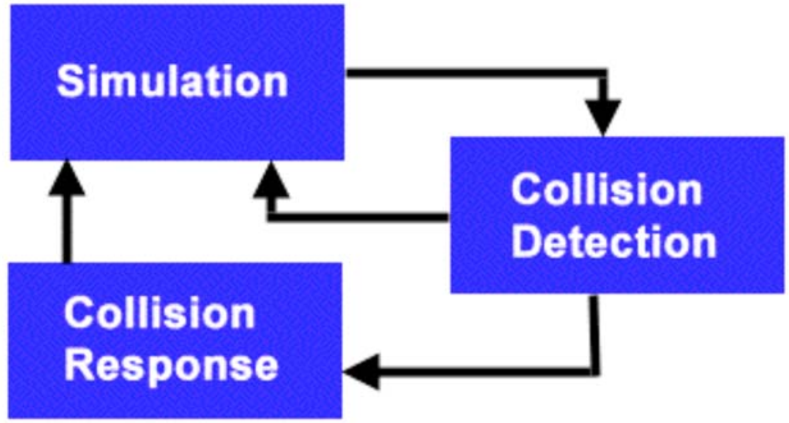
  Approximating Polyhedra with Spheres for Time-Critical Collision Detection, Philip M. Hubbard





Dynamic Simulation Architecture

- Collision Detection is generally the bottleneck in any dynamic simulation system



- Lots of ways to speed up collision-detection

Reducing Collision Tests

- Testing each object with all others is $O(N^2)$
- At minimum, do bounding sphere tests first
- Reduce to O(N+k) with sweep-and-prune
  - See SIGGRAPH 97 Physically Based Modelling Course Notes
- Spatial Subdivision is fast too
  - Updating after movement can be complicated
  - AABB is easiest to sort and maintain
  - Not necessary to subdivide all 3 dimensions

Other neat tricks

- Raycasting
  - Fast heuristic for ‘pass-through’ problem
  - Sometimes useful in AI
    - Like for avoiding other cars
- Caching
  - Exploit frame coherency, cache the last vertex/edge/face that failed and test it first
  - ‘hits’ most of the time, large gains can be seen

Dynamic Particle Simulation

- Simplest type of dynamics system
- Based on basic linear ODE:

Acceleration due to force is $f = ma$:

$$a = \frac{d^2x}{dt^2} \quad \therefore \quad \frac{f}{m} = \frac{d^2x}{dt^2}$$

velocity is derivative of position wrt time:

$$v = \frac{dx}{dt} \qquad \frac{dv}{dt} = \frac{f}{m}$$

Particle Movement

- Particle Definition:
  - Position x(x,y,z)
  - Velocity v(x,y,z)
  - Mass m
- For timestep Δt and force $f = (f_x, f_y, f_z)$

$$p_{t+\Delta t} = p_t + \Delta t v_t$$

$$v_{t+\Delta t} = v_t + \Delta t \frac{f}{m}$$

Example Forces

- gravity : $f = mg$

  drag : $f = -k_d v$ (k_d is drag coefficient)

  spring between $\mathrm{P_1}(\mathrm{x_1}, \mathrm{v_1}, \mathrm{m_1})$ and $\mathrm{P_2}(\mathrm{x_2}, \mathrm{v_2}, \mathrm{m_2})$:

  $$\Delta x = x_1 - x_2, \qquad \Delta v = v_1 - v_2$$

  $$f_{\mathrm{P_1}} = -\left[k_s\left(\|\Delta x\| - r\right) + k_d\left(\frac{\Delta v \cdot \Delta x}{\|\Delta v\|}\right)\right]\frac{\Delta x}{\|\Delta x\|}$$

  $$f_{\mathrm{P_2}} = -f_{\mathrm{P1}}$$

  (k_s is spring constant, k_d is damping constant)

Using Particle Dynamics

- Each timestep:
  - Update position (add velocity)
  - Calculate forces on particle
  - Update velocity (add force over mass)
- Model has no rotational velocity
  - You can hack this in by rotating the velocity vector each frame
  - Causes problems with collision response

Particle Collision System

- Assumption: using OBBs
- Do geometric test for colliding boxes
  - Test **all** vertices and edges
  - Save intersections
- Calculate intersection point and time-of-intersection for each intersection
- Move object back largest timestep
- Apply collision response to intersection point

Closed-form expression for point/plane collision time

- Point: position p, velocity v
- Plane: normal n, velocity v_p, point on plane x
- Point on plane if $(x - p) \cdot n = 0$
- With linear velocity normal is constant:

$$\left(\left(x + tv_p\right) - \left(p + tv\right)\right) \cdot n = 0$$

- now solve for t:

$$t = \frac{n \cdot p - n \cdot x}{n \cdot v_p - n \cdot v}$$

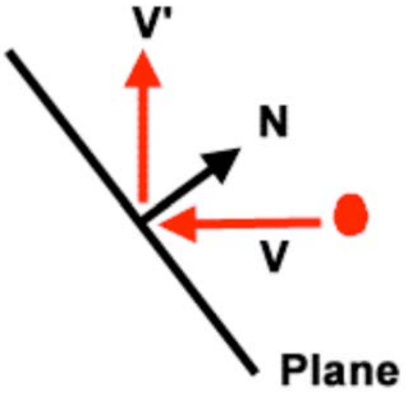
Problems with point/plane time

- Have to test point with 3 faces (why?)
- Time can be infinity / very large
  - Ignore times that are too big
  - Heuristic: throw away if larger than last timestep
- If the rotation hack was applied, can return a time larger than last timestep
  - This is why the rotation hack is bad
  - Can always use subdivision in this case
    - Have to use subdivision for edges as well, unless you can come up with a closed-form edge collision time (which shouldn't be too hard, just sub (p_i+tv_i) into line-intersection test)

Binary Search for collision time

- Move OBB back to before current timestep
- Run simulator for half the last timestep
- Test point / edge to see if they are still colliding
- Rinse and repeat to desired accuracy
  - 3 or 4 iterations is probably enough

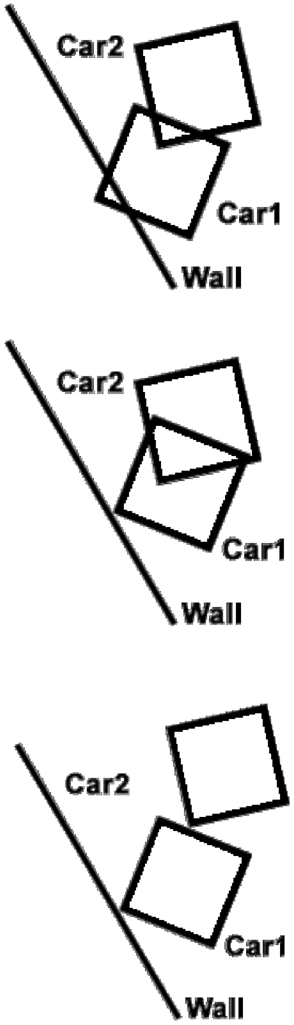
Particle Collision Response



- Basic method: vector reflection
- Need a vector to reflect around
  - Face normal for point/face
  - Cross product for edge/edge
  - Negate vector to reflect other object
- Vector Reflection: $V' = V - 2(N \cdot I)N$
- Can make collision elastic by multiplying reflected vector by elasticity constant
- You can hack in momentum-transfer by swapping the magnitude of each object’s pre-collision velocity vector
- This and the elastic collision constant make a reasonable collision response system

Multiple Collisions

- This is a significant problem
- For racing games, resolve dynamic/static object collisions first (walls, buildings, etc)
- Then lock resolved objects and resolve any collisions with them, etc, etc
- This will screw with your collision-time finding algorithms
  - Car2 may have to move back past where it started last frame
- The correct way to handle this is with articulated figures, which require linear systems of equations and are rather complicated (see Moore88)



Rigid Body Dynamics

- Now working with volumes instead of points
  - Typically OBB's, easy to integrate over volume
- Rotational/Angular velocity is part of the system
- A lot more complicated than the linear system
- SIGGRAPH 97 Physically Based Modelling course notes walk through the math and code for a Rigid Body Dynamics system. Far more useful than what I will skim over in these slides.

Center of Mass

- Also called the Centroid
- System is easiest to build if we place objects in local coordinate system
  - Want centroid at origin (0,0,0)
- x(t) is translation of origin in world coords
- R(t) rotates local reference frame (axis) into world axis
  - 3x3 rotation matrix

Linear Velocity and Momentum

- Linear velocity v(t) is just like particle velocity
- Linear momentum P(t) is velocity of mass of rigid body
- P(t) = Mv(t)
- More useful for solving motion equations

Force and Torque

- Force F(t) acts on the centroid, just like force for a particle
- Torque T(t) acts on a particle in the rigid body volume
- Force affects linear velocity, Torque affects angular velocity

Angular Velocity and Momentum

- Angular Velocity w(t) defines an axis object rotates around, magnitude is rotation speed
- Angular momentum L(t) is related to w(t) by inertia tensor I(t)
- L(t) = I(t)w(t)
- Angular Momentum is again more useful for solving dynamics equations

Inertia Tensor

- Relates angular velocity and angular momentum
- 3x3 matrix $I(t)$, entries are derived by integrating over object's volume
  - OBB is easy to integrate over
- Can be computed as $I(t) = R(t)I_{body}R(t)^T$, where I_{body} can be pre-computed
- Note: $I(t)^{-1} = R(t)I_{body}^{-1} R(t)^T$
- This is the most complicated part…

So how do we simulate motion?

- Rigid body is represented by:
  - Position x(t)
  - Rotation R(t)
  - Linear Momentum P(t)
  - Angular Momentum L(t)
- New Position:
  - $x`(t) = x(t) + v(t)$, where $v(t) = P(t)/Mass$
  - $R`(t) = w(t)*R(t)$, where $w(t) = I(t)^{-1}L(t)$
  - $P`(t) = P(t) + F(t)$
  - $L`(t) = L(t) + T(t)$
- Calculating F(t), T(t), and I_{body} are complicated, see the online SIGGRAPH 97 course notes

Rigid Body Collision Resolution

- Similar to particle collision resolution
- Finding collision point / time is difficult
  - No closed forms, have to use binary search
- Online SIGGRAPH 97 Course Notes have full explanation and code for how to calculate collision impulses
  - Rigid Body Simulation II – Nonpenetration Constraints
- Also describe resting contact collision, which is much more difficult

Helpful Books

- Real Time Rendering, chapters 10-11
  - Lots of these notes derived from this book
- Graphics Gems series
  - Lots of errors, find errata on internet
- Numerical Recipes in C
  - The mathematical computing bible
- Game Programming Gems
  - Not much on CD, but lots of neat tricks
- Lex & Yacc (published by O'reilly)
  - Not about CD, but useful for reading data files

What your mom never told you about PC hardware

- Cache Memory
  - Linear memory access at all costs!
  - Wasting cycles and space to get linear access is often faster. It is worth it to do some profiling.
  - DO NOT USE LINKED LISTS. They are bad.
  - STL vector<T> class is great, so is STL string
    - vector<T> is basically a dynamically-sized array
    - vector.begin() returns a pointer to front of array
- Conditionals are Evil
  - Branch prediction makes conditionals dangerous
  - They can trash the pipeline, minimize them if possible

What your mom never told you about C/C++ compilers

- Compilers only inline code in headers
  - The inline keyword is only a hint
  - If the code isn't in a header, inlining is impossible
- Inlining can be an insane speedup
- Avoid the temptation to be too OO
  - Simple objects should have simple classes
    - Eg: writing your own templated, dynamically resizable vector class with a bunch of overloaded operators is probably not going to be worth it in the end.

What your mom never told you about Numerical Computing

- Lots of algorithms have degenerate conditions
  - Learn to use isinf(), isnan(), finite()
- Testing for X = 0 is dangerous
  - If X != 0, but is really small, many algorithms will still degenerate
  - Often better to test fabs(X) < (small number)
- Avoid sqrt(), pow() – they are slow