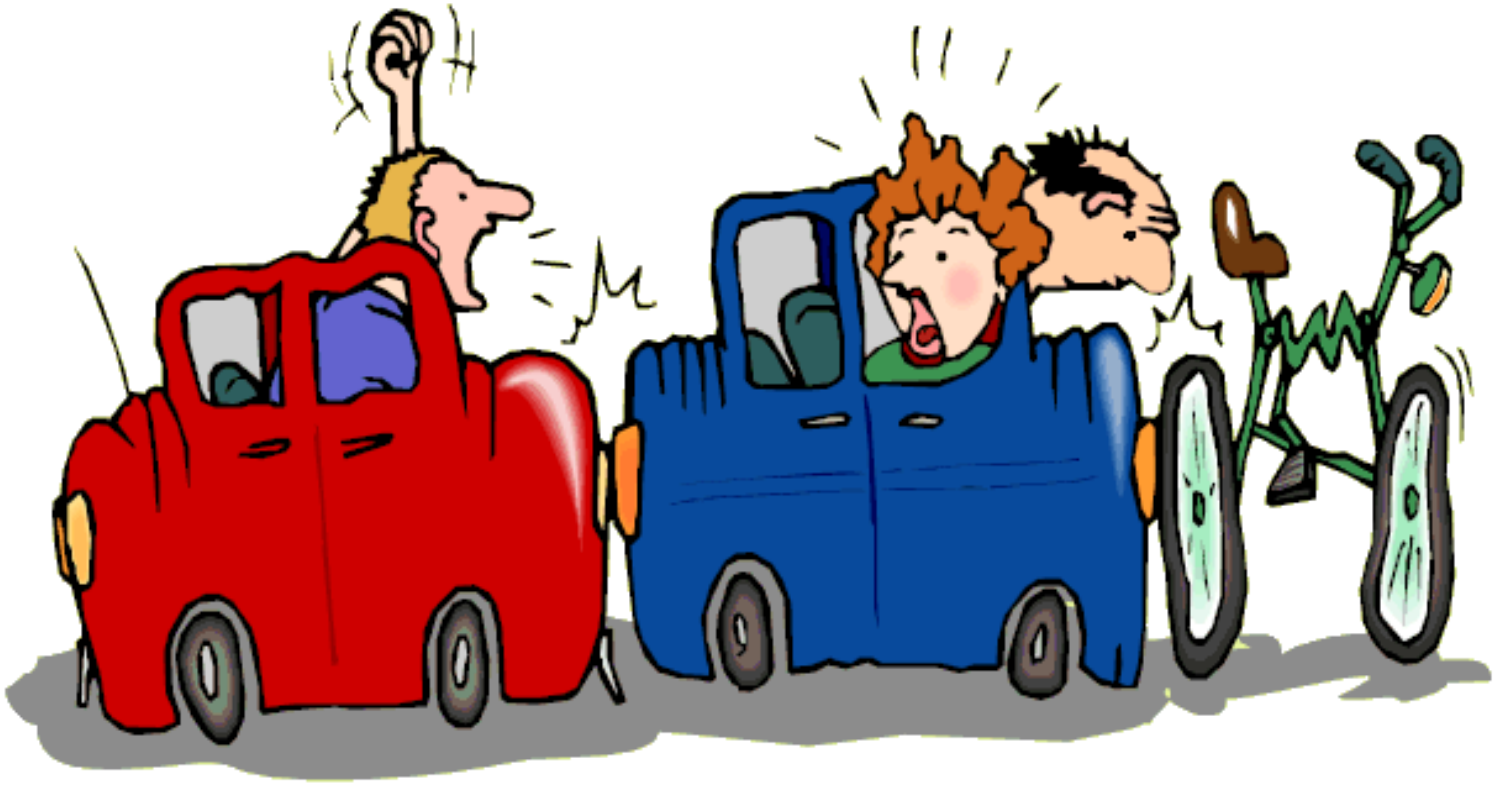
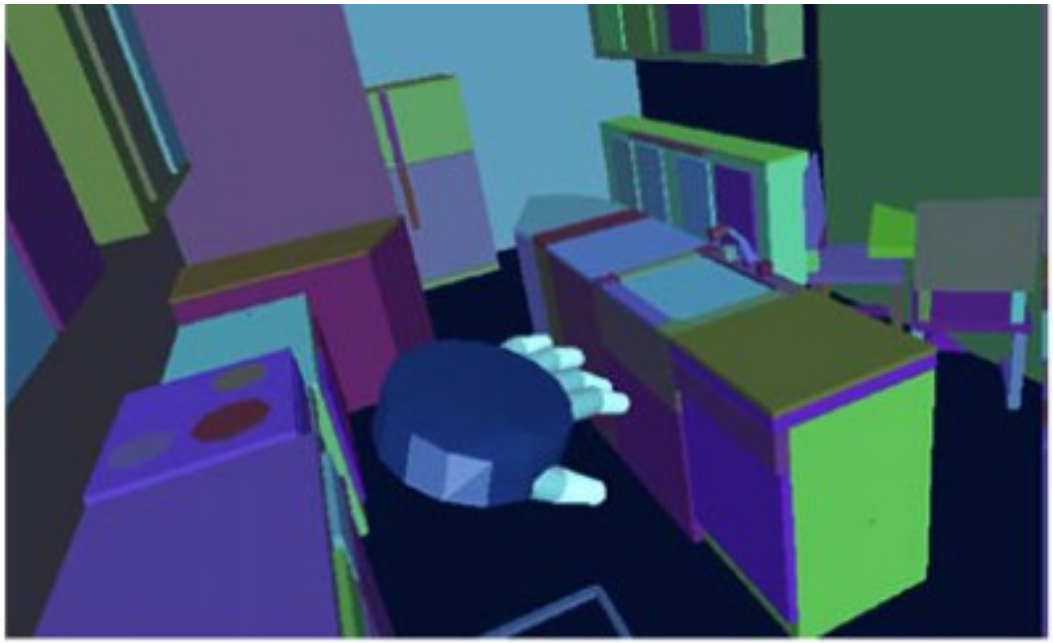
Detekcija trkov



Pomen detekcije trkov

- Especially important in Interactive Virtual Environments.



- No matter how good the VE look, the poor realism resulting from a lack of collision detection.
- However, there may be hundreds, even thousands of objects in the virtual world.

Pomen detekcije trkov

Uvod

Introduction

- Without collision detection (CD), moving bodies will pass through each other
- CD is very important in many applications areas
  - physically-based simulation
  - robotics
  - virtual reality
  - games

Kaj je odkrivanje trkov?

- Entities occupy, partially or fully, the same location at the same time.
- We term the above scenario a collision.
- Collision leads to event(s) in most cases –
  - Bullet hitting a box of grenades causing the event of explosion and possibly other events (e.g., reduction of life force for players stood near the grenades at time of explosion).
- However, this is not the full story –
  - Ideally we should foresee collision as immanent and time our event to occur at a deterministic point in time (true collision).
  - Q: How is this achieved while still maintaining performance?
  - A: With great difficulty when performance permits.

Collisions

Collision handling =

Collision detection (geometric problem) +

Collision response (dynamic problem)

Simple approach:

```
for all objects
  savePos
  updatePos
for all objects
  if collision
    restorePos
```

Odkrivanje trkov

- Essential for many games
  - Shooting
  - Kicking, punching, beating, whacking, smashing, hitting, chopping, dicing, slicing, julienne fries…
  - Car crashes
- Expensive
  - tests!!

$$O(N^2)$$

Demo

Problem N teles

An N-body problem

- Involves testing all pairs of polygons
- Inherent complexity is N^2

Strategies:

- Spatial partitioning
- Multiphase algorithms
- Categorize objects (static, dynamic, etc)
- Temporal cohesion

Multi-phase detection strategy

1. Detect possible collision between objects
   - i.e. cull things which can not collide
2. Detect colliding surfaces and determine point of contact

Različne vrste teles, ki trkajo

Different kinds of colliding bodies

Three levels of difficulty:

1. Convex rigid bodies
2. General rigid bodies
3. Deformable bodies

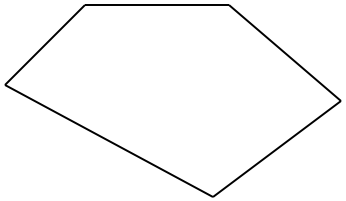
Uporaba odkrivanja trkov

- Determining if the player or character has a hit a wall or obstacle
  - To stop them walking through it
- Determining if a projectile (missile) has hit a target
- Detecting points at which behavior should change
  - A car in the air returning to the ground
- Cleaning up animation
  - Making sure a character’s feet do not pass through the floor
- Simulating motion of some form
  - E.g. cloth, or something else

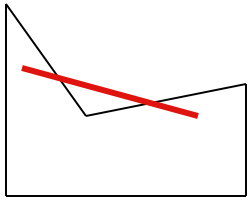
Odkrivanje trkov

- *Collision detection*, as used in the games community, usually means intersection detection of any form
  - *Intersection detection* is the general problem: find out if two geometric entities intersect – typically a static problem
  - *Interference detection* is the term used in the solid modeling literature – typically a static problem
  - *Collision detection* in the research literature generally refers to a dynamic problem – find out if and when moving objects collide
- Subtle point: Collision detection is about the algorithms for finding collisions *in time* as much as space

Terminologija

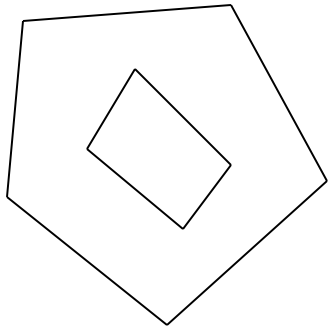


Convex

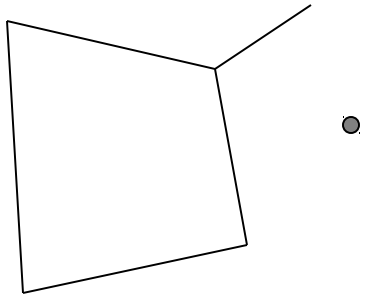


Concave

An object is convex if for every pair of points inside the object, the line joining them is also inside the object



Manifold



Non-Manifold

An surface is manifold if every point on it is homeomorphic to a disk.
Roughly, every edge has two faces joining it

Izbira algoritma

- The geometry of the colliding objects is the primary factor in choosing a collision detection algorithm
  - “Object” could be a point, or line segment
  - Object could be specific shape: a sphere, a triangle, a cube, …
  - Objects can be concave/convex, solid/hollow, deformable/rigid
- The way in which objects move is a second factor
  - Different algorithms for fast or slow moving objects
  - Different algorithms depending on how frequently the object must be updated
- Of course, speed, simplicity, robustness are all other factors

Strategije odkrivanja trkov

- There are several principles that should be considered when designing a collision detection strategy
- What might they be?
  - Say you have more than one test available, with different costs. How do you combine them?
  - How do you avoid unnecessary tests?
  - How do you make tests cheaper?

Strategije odkrivanja trkov

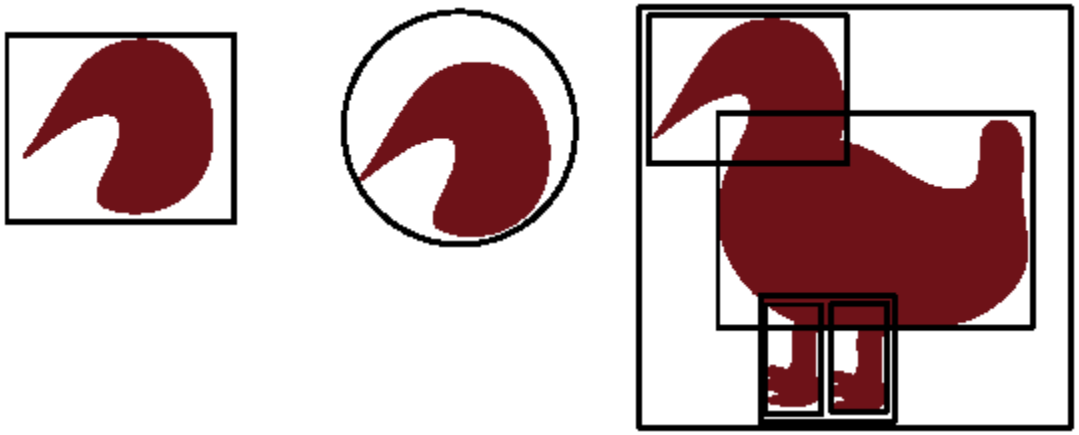
- There are several principles that should be considered when designing a collision detection strategy
- *Fast simple tests first* to eliminate many potential collisions
  - e.g.: Test bounding volumes before testing individual triangles
- Exploit *locality* to eliminate many potential collisions
  - e.g.: Use cell structures to avoid considering distant objects
- Use as much *information* as possible about the geometry
  - e.g.: Spheres have special properties that speed collision testing
- Exploit *coherence* between successive tests
  - Things don’t typically change much between two frames

Robustnost

- For our purposes, collision detection code is *robust* if it:
  - Doesn't crash or infinite loop on *any* case that might occur
    - Better if it doesn't crash on any case at all, even if the case is supposed to be "impossible"
  - Always gives some answer that is meaningful, or explicitly reports that it cannot give an answer
  - Can handle many forms of geometry
  - Can detect problems with the input geometry, particularly if that geometry is supposed to meet some conditions (such as convexity)
- Robustness is remarkably hard to obtain

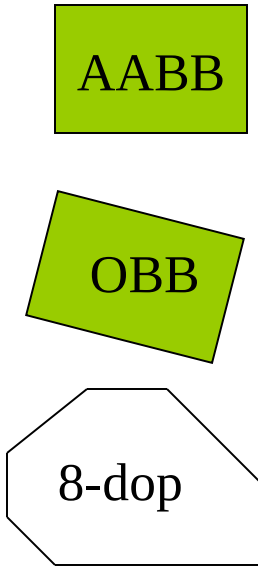
Odkrivanje trkov

- Efficiency hacks/cheats
  - Fewer tests: Exploit spatial coherence
    - Use bounding boxes/spheres
    - Hierarchies of bounding boxes/spheres



Tipi geometrije

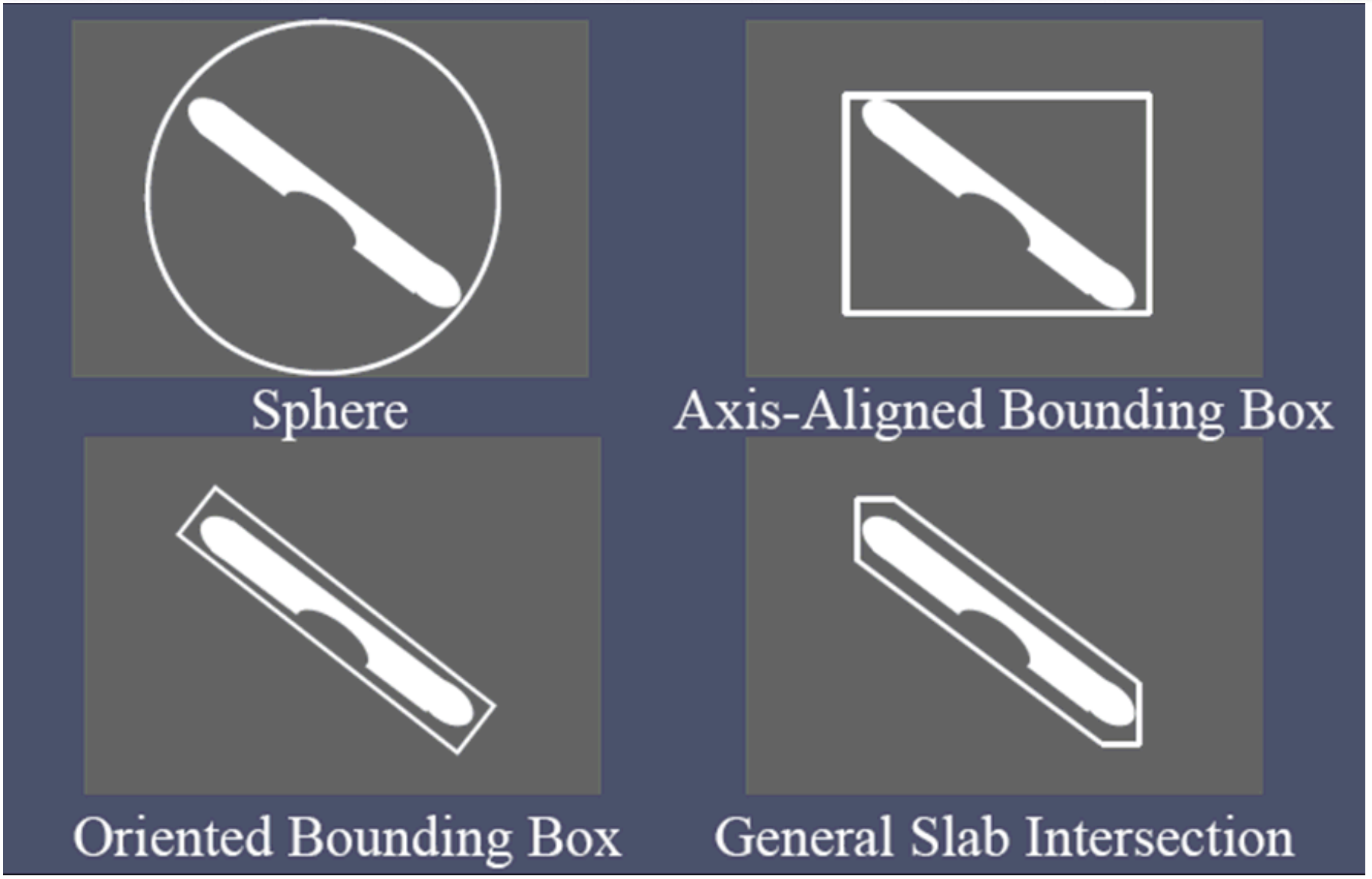
- Points
- Lines, Rays and Line Segments
- Spheres, Cylinders and Cones
- Cubes, rectilinear boxes - Axis aligned or arbitrarily aligned
  - AABB: Axis aligned bounding box
  - OBB: Oriented bounding box
- k-dops – shapes bounded by planes at fixed orientations
- Convex, manifold meshes – any mesh can be triangulated
  - Concave meshes can be broken into convex chunks, by hand
- Triangle soup
- More general curved surfaces, but not used (often) in games



Collision Detection using Bounding Spheres (Boxes)

- If the bounding spheres (boxes) of two objects don't intersect, there is no collision between the two object.
  - How to determine the intersection between spheres / boxes?
- If the bounding volumes intersect, there may be collision between the objects.
  - However, there may still be no collision.
  - Tight bounds performs better. We may treat two objects collided if the bounds are tight and intersect.
- Not every object has a good bounding sphere (box), e.g. planes.

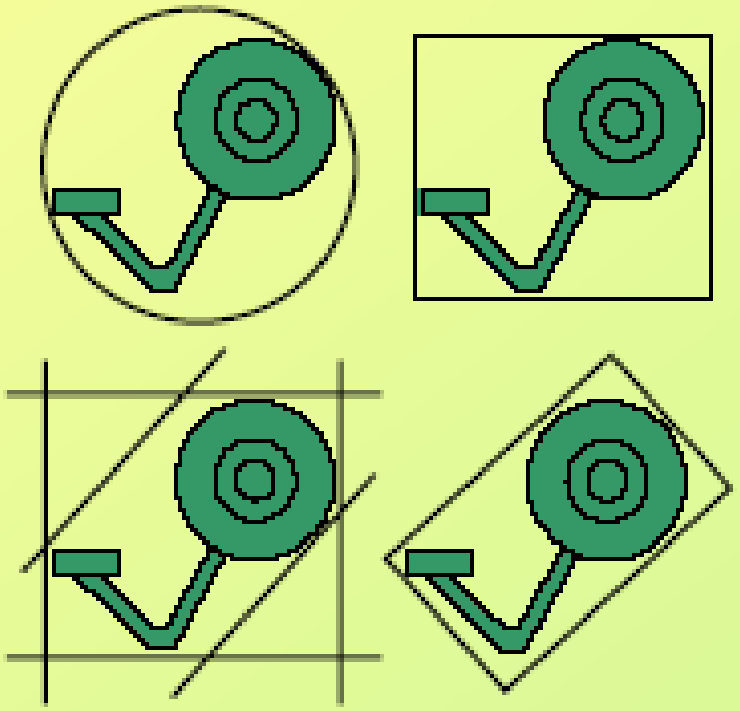
Primeri obsegajočih volumnov



Primeri obsegajočih volumnov

Bounding Volumes

- Examples
  - Spheres
  - AABBs
  - K-DOPs
  - OBBs
  - Sphere swept vol.
  - Convex hull



How do we compute these BVs?

Obsegajoča krogla in kvader

- Bounding sphere: the minimum sphere enclosing the object.
  - You may need to find the center of the object and the radius of the sphere. (can be done in the modeling phase)
  - Not accurate in may cases. But can be used as initial test before applying more complicated algorithms.
  - Not a good fit for narrow or long objects
- Bounding box (axis-aligned): the minimum and maximum coordinates of the object in x, y, z directions
  - Not transform-invariant. Need to be recomputed after a transformation, e.g. rotation.
  - Can fit odd shaped objects better

Obsegajoči kvader (bounding box)

- Place a box around an object
  - Box should completely enclose the object and be of minimal volume
  - Fairly simple to construct
- Test intersections between the boxes to find intersections
- Each box has 6 faces (planes) in 3D
  - Simple algebra to test for intersections between planes
  - If one of the planes intersects another, the objects are said to collide

Obsegajoči kvader

- Space complexity
  - Each object must store 8 points representing the bounding box
  - Therefore, space is O(8) and Ω(8)
- Time complexity
  - Each face of each object must be tested against each face of each other object
  - Therefore, $O((6n)^2) = O(n^2)$
    - n is the number of objects

Obsegajoči kvader

- Pro
  - Very easy to implement
  - Very little extra space needed
- Con
  - Very coarse detection
  - Very slow with many objects in the scene

Obsegajoče krogle (Bounding Spheres)

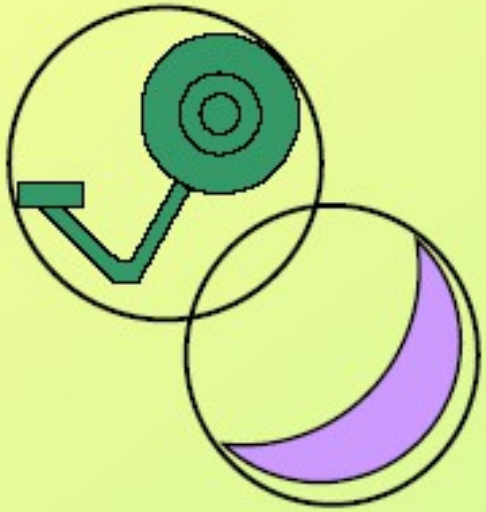
- Similar to bounding boxes, but instead we use spheres
  - Must decide on a “center” point for the object that minimizes the radius
  - Can be tough to find such a sphere that minimizes in all directions
  - Spheres could leave a lot of extra space around the object!

Obsegajoče krogle

- Each sphere has a center point and a radius
  - Can build an equation for the circle
  - Simple algebra to test for intersection between the two circles

Sphere/Sphere Overlap

- Sphere defined by
  - center, radius
- Very simple and fast overlap test
- Bad fit in many cases
- Fast to update for rigid body motion



Obsegajoče krogle

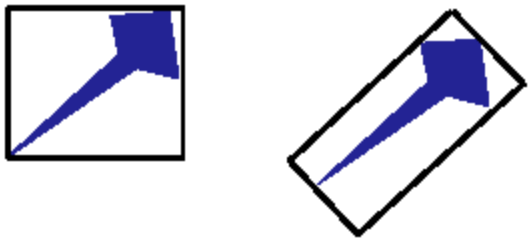
- Space complexity
  - Each object must store 2 values - center and radius - to represent the sphere
  - Therefore, space is O(2) and Ω(2)
  - Space is slightly less than bounding boxes
- Time complexity
  - Each object must test it's bounding sphere for intersection with each other bounding sphere in the scene
  - Therefore, $O(n^2)$
    - n is the number of objects
  - Significantly fewer calculations than bounding boxes!

Obsegajoče krogle

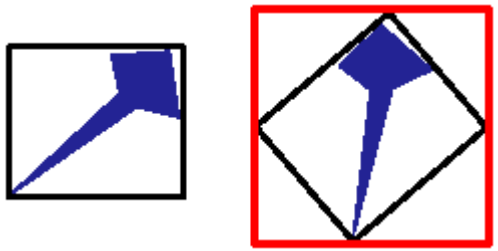
- Pro
  - Even easier to implement (than bounding boxes)
  - Even less space needed (than bounding boxes)
- Con
  - Still fairly coarse detection
  - Still fairly slow with many objects

Poravnani obsegajoči kvadri (Aligned Bounding Boxes)

- Axis-aligned vs. Object-aligned



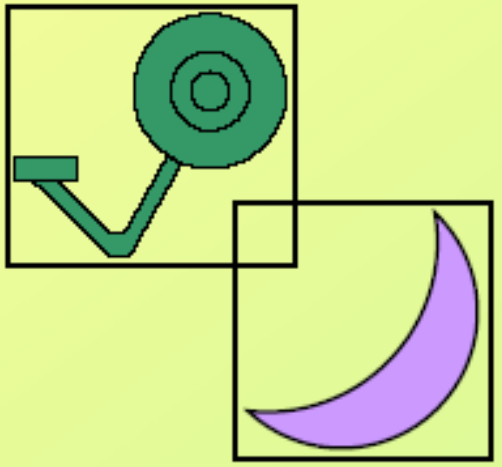
- Axis-aligned BBox change as object moves
- Approximate by rotating BBox



Swept volume

AABB/AABB Overlap

- Box defined by either
  - min, max
  - mid, ext
- Very simple and fast overlap test
  - It is also trivial to find the overlap region
- Bad fit in many cases



Oriented Bounding Boxes (OBBs)

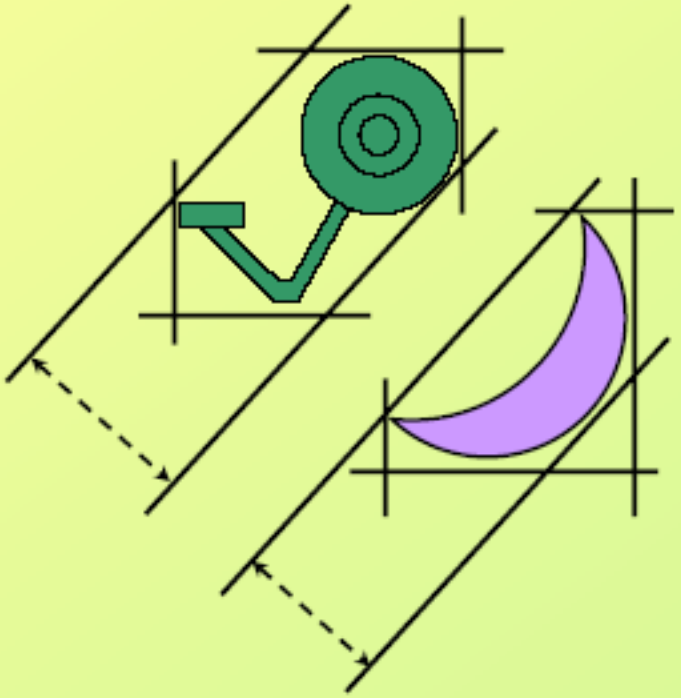
- These are more difficult to implement and result in greater effort to achieve collision detection. However, the collision detection is more accurate than AABBs.
- Furthermore, because the OBBs are expected to “fit” more tightly around an entity the tree structure is expected to be much reduced over that of AABBs.
- Checking for OBBs that are collided means ensuring that no separating axis exist (same as AABBs). However, in 3D there are 15 potential axes. Furthermore, the axes do not lie trivially on regular planes (X, Y, Z) but must be deduced from the orientation of the OBBs involved.
  - Handling OBBs in three dimensions are beyond the scope of this course.
  - AABBs are adequate for our purposes.

OBB/OBB Overlap

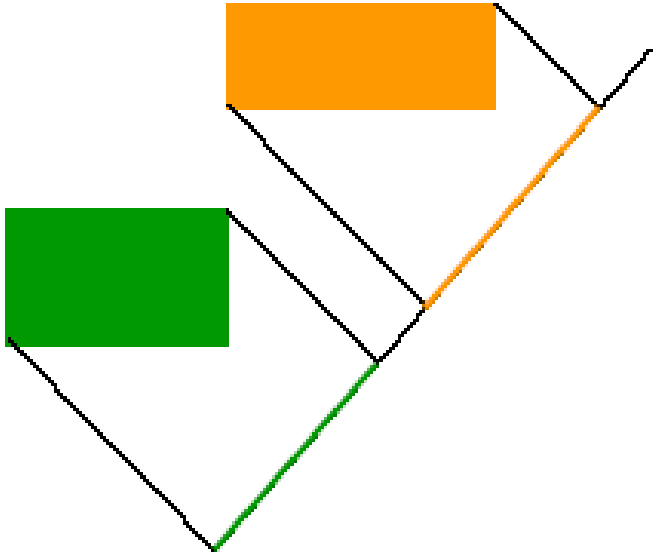
- OBB defined by
  - mid, ext
- Overlap test based on separating axes theorem
  - still rather expensive
- Good fit in many cases
- Fast to update for rigid body motion

k-DOP/k-DOP Overlap

- k-DOP defined by
  - k/2 normals
    - shared by all k-DOPs
  - k/2 1D intervals
    - Stored for each k-DOP
- Generalization of AABB
  - overlap test analogous to AABB/AABB test
- Good fit in many cases



Separating axis



Two convex objects are disjoint if and only if there exists an axis on which their projections are disjoint.

Corrolary: If there is no such axis, they intersect.

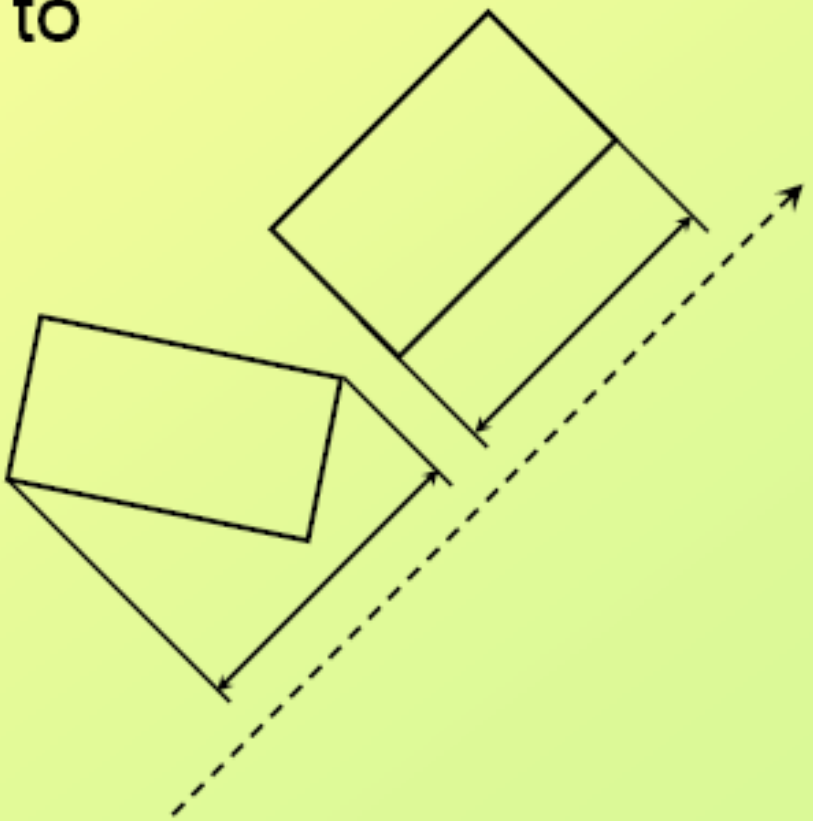
Separating axis theorem

If two convex polyhedra, A and B, are disjoint there exists a separating axis which is orthogonal to either:

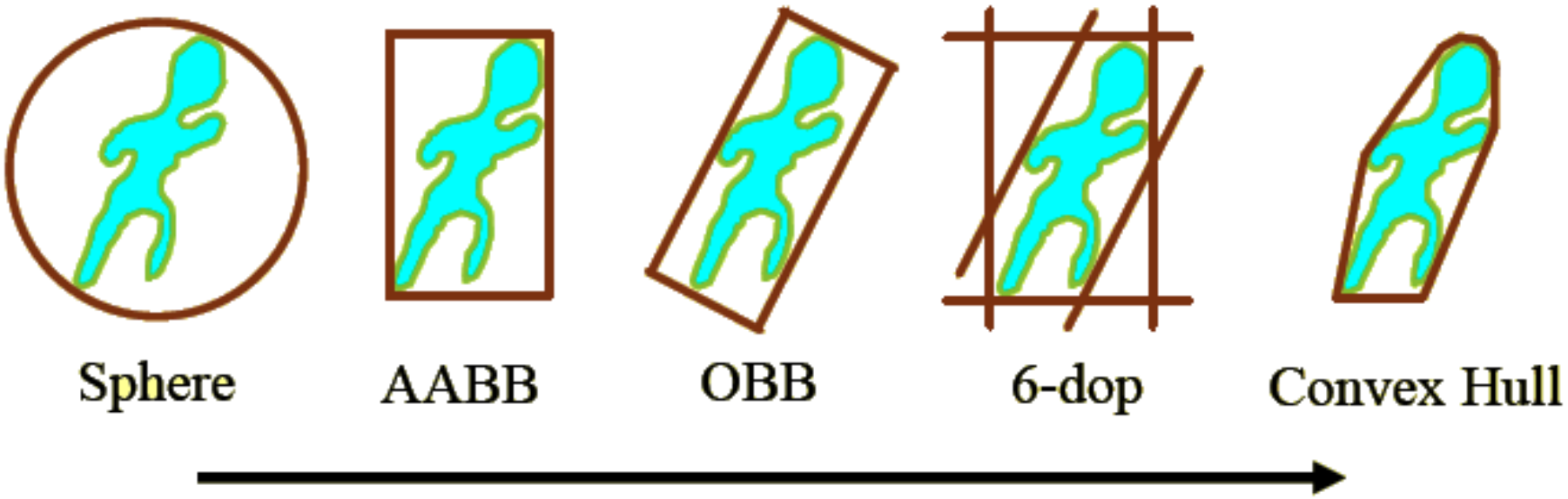
- a face of A
- a face of B
- an edge from A and an edge from B

Separating Axes Theorem

- OBB case: we need to test 15 axes (in 3D)
  - 3 face axes from each OBB
  - 9 edge/edge cross product axes



Trade-off in Choosing BV's



Increasing:

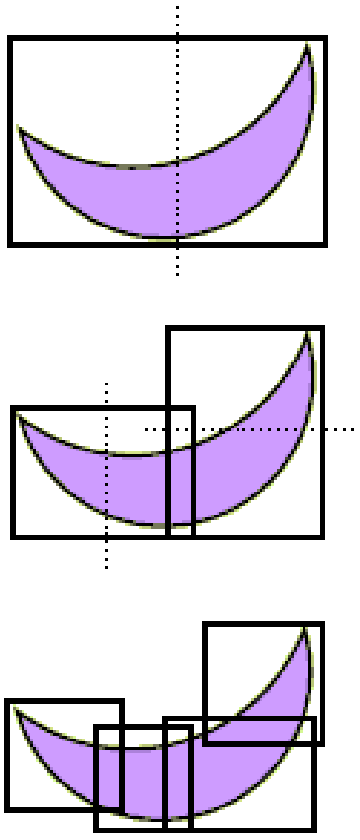
- Complexity
- Tightness of Fit
- Cost of overlap test

Hierarhije obsegajočih volumnov

- Test status of two close bodies
  - often called the narrow phase
- Standard CD method for complex bodies
- Why hierarchies?
  - Naive approach is $O(nm)$

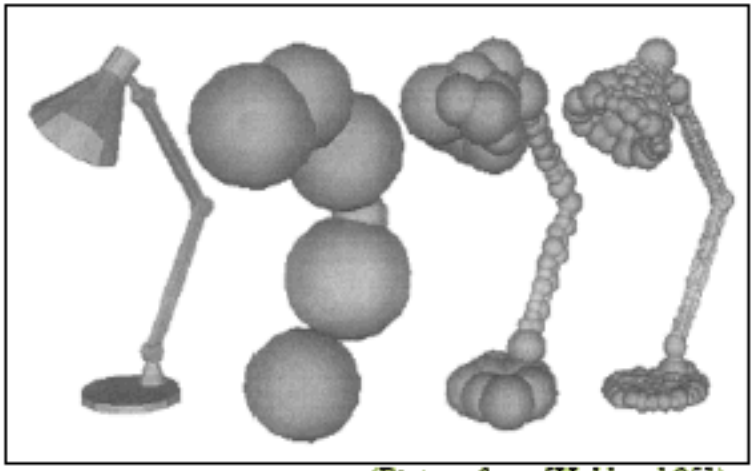
Izgradnja hierarhije

- Basic Approaches
  - Top-down
  - Bottom-up
  - Incremental insertion
- Other decision
  - BV type
  - Node degree
  - Primitives per leaf
  - Balanced tree?
    - Not best in all cases



Hierarhije za trdna telesa

- BV Tree Algorithms
  - SphereTree
  - OBBTree
  - k-DOPTree
  - QuOSPOTree
  - ConvexPieceTree



(Picture from [Hubbard 95])

Performance evaluation is extremely difficult:
- No algorithm performs best in all cases.
- Scenarios differs from application to application.

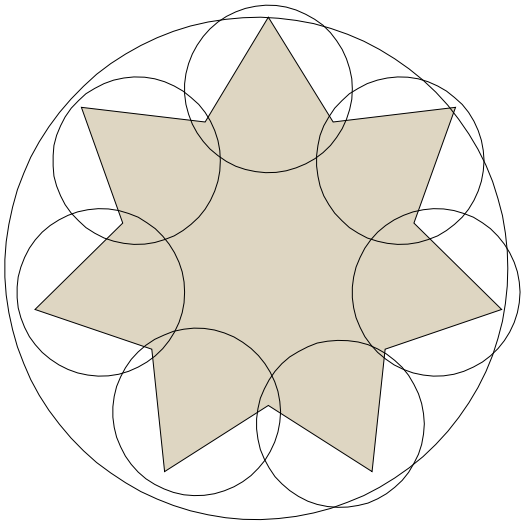
Optimization Structures

- BV, BVH (bounding volume hierarchies)
- Octree
- KD tree
- BSP (binary separating planes)
- OBB tree (oriented bounding boxes- a popular form of BVH)
- K-dop
- Uniform grid

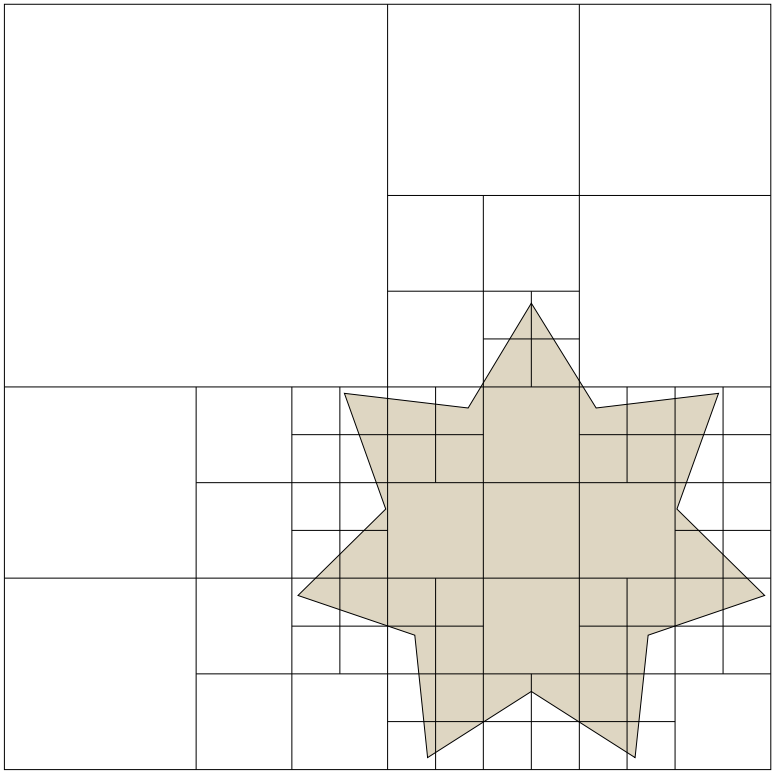
Testing BVH's

```
TestBVH(A,B) {
    if(not overlap(A_BV, B_BV) return FALSE;
    else if(isLeaf(A)) {
        if(isLeaf(B)) {
            for each triangle pair (T_a,T_b)
                if(overlap(T_a,T_b)) AddIntersectionToList();
        }
        else {
            for each child C_b of B
                TestBVH(A,C_b);
        }
    }
    else {
        for each child C_a of A
            TestBVH(C_a,B)
    }
}
```

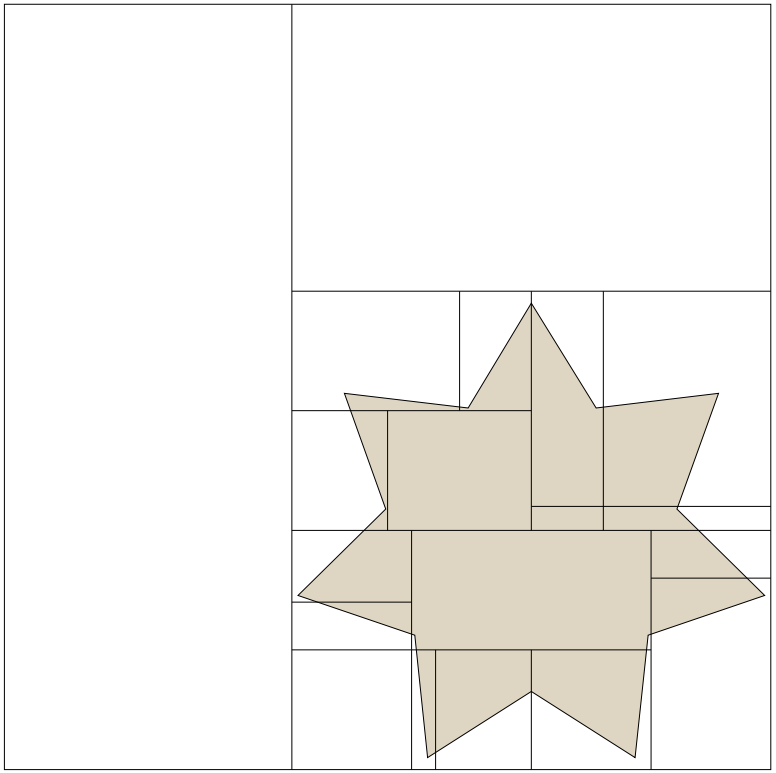
Bounding Volume Hierarchies



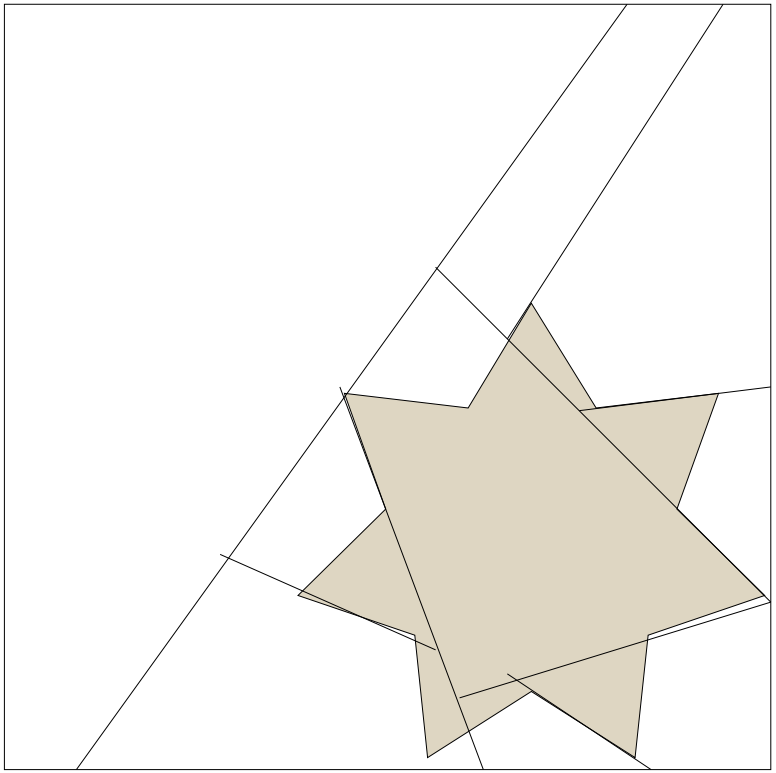
Octrees



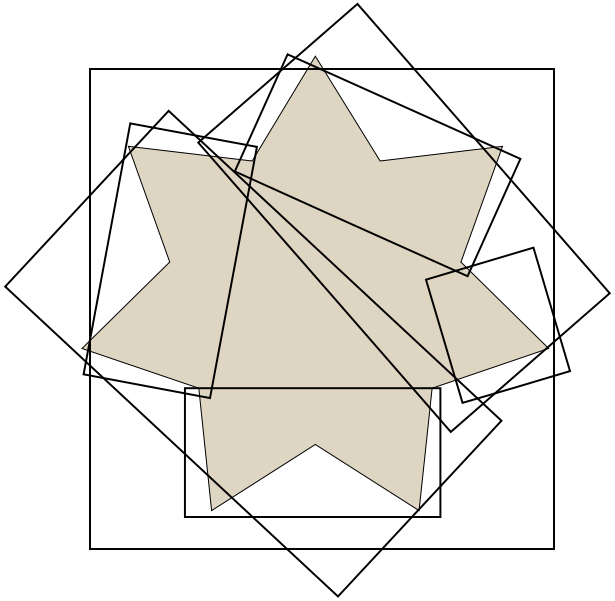
KD Trees



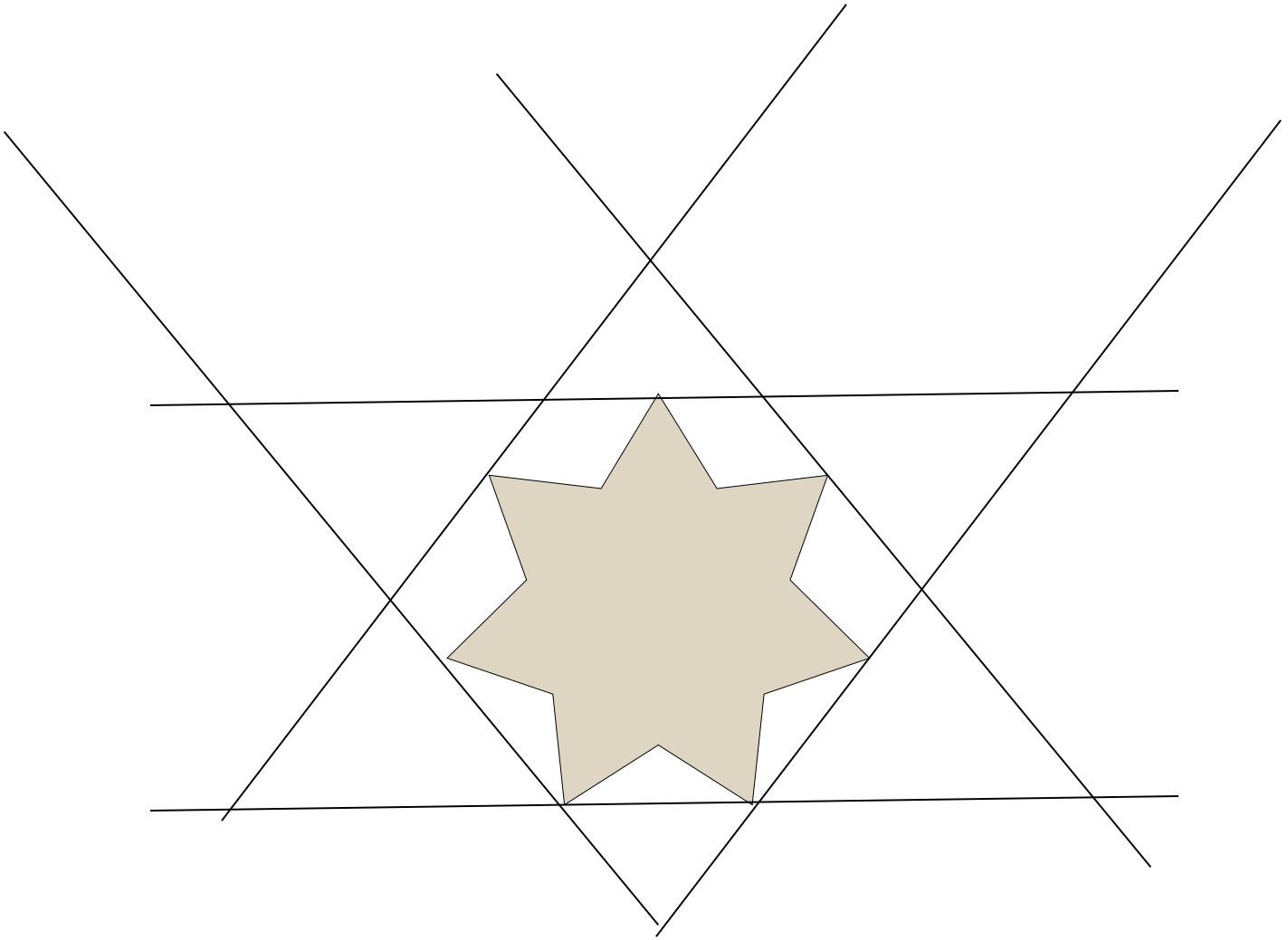
BSP Trees



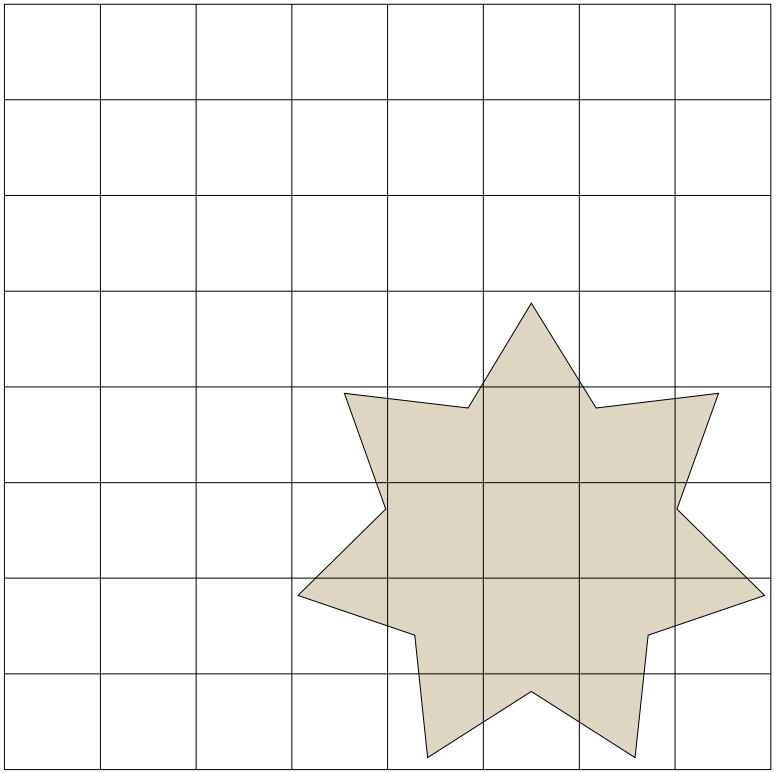
OBB Trees



K-Dops



Uniform Grids



BSP Trees

- BSP (Binary Space Partitioning) trees are used to break a 3D object into pieces for easier comparison
  - Object is recursively broken into pieces and pieces are inserted into the tree
  - Intersection between pieces of two object's spaces is tested

BSP Trees

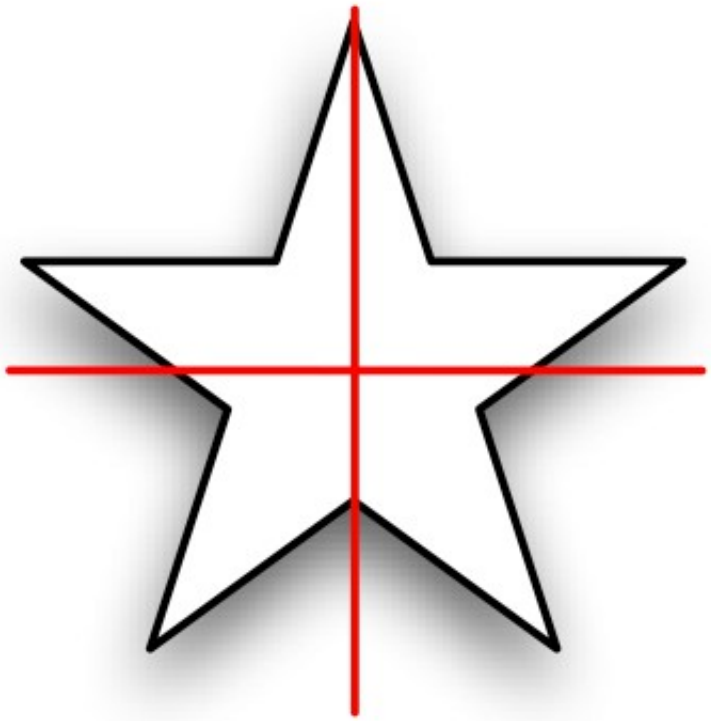
- We refine our BSP trees by recursively defining the children to contain a subset of the objects of the parent
  - Stop refining on one of a few cases:
    - Case 1: We have reached a minimum physical size for the section (ie: one pixel, ten pixels, etc)
    - Case 2: We have reached a maximum tree depth (ie: 6 levels, 10 levels, etc)
    - Case 3: We have placed each polygon in a leaf node
    - Etc… - Depends on the implementer

BSP Trees

- Example BSP Tree

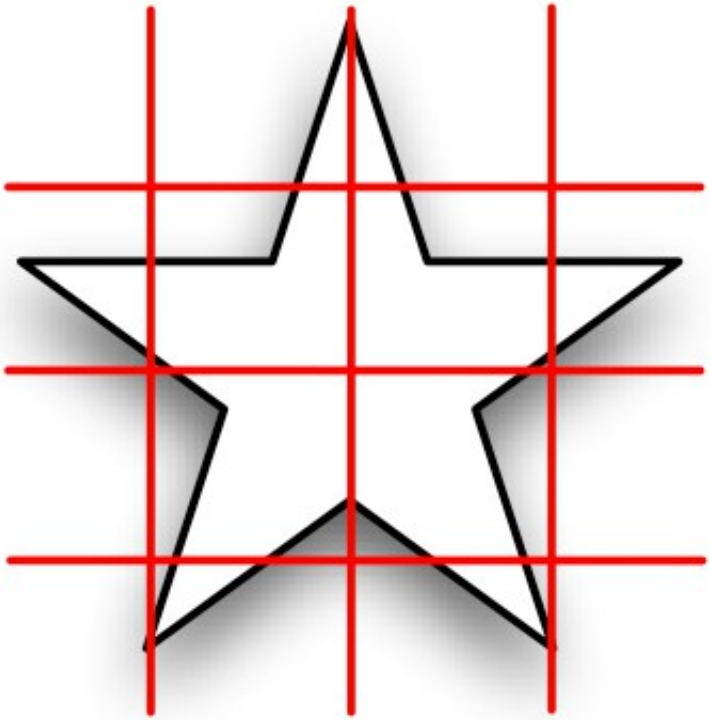
BSP Trees

- Example BSP Tree



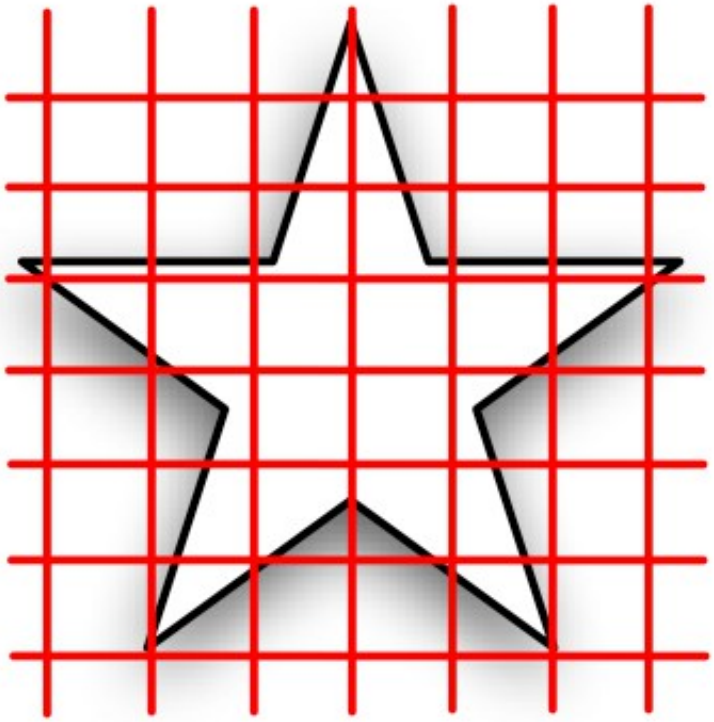
BSP Trees

- Example BSP Tree



BSP Trees

- Example BSP Tree



BSP Trees

- Collision Detection
  - Recursively travel through both BSP trees checking if successive levels of the tree actually intersect
    - If the sections of the trees that are being tested have polygons in them:
      - If inner level of tree, assume that an intersection occurs and recurse
      - If lowest level of tree, test actual intersection of polygons
    - If one of the sections of the trees that are being tested does not have polygons in it, we can surmise that no intersection occurs

BSP Trees

- Space Complexity
  - Each object must store a BSP tree with links to children
  - Leaf nodes are polygons with geometries as integer coordinates
  - Therefore, space depends on number of levels of tree, h, and number of polygons (assume convex triangles - most common), n
  - Therefore, space is $O(4^h + 3n)$ and $\Omega(4^h + 3n)$

BSP Trees

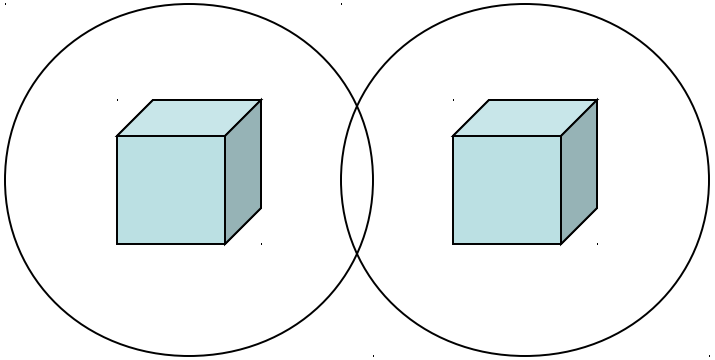
- Time Complexity
  - Each object must be tested against every other object - n^2
  - If intersection at level 0, must go through the tree - O(h)
    - Assume all trees of same height
  - Depends on number of intersections, m
  - Therefore, $O(n^2 + m*h)$ and $\Omega(n^2)$

BSP Trees

- Pros
  - Fairly fine grain detection
- Cons
  - Complex to implement
  - Still fairly slow
  - Requires lots of space

Using a sphere

- We surround our entity with a large enough sphere to ensure that collisions (in the most part) are detected.

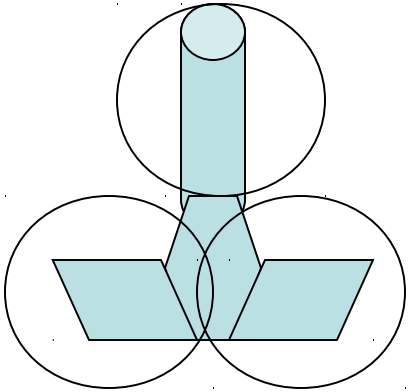


- When projected into consecutive frames, the spheres should overlap.
- This ensures we cover the whole (well nearly whole) scope of the possible collision.

- We check if the distance between the two centres of the spheres is less than the sum of the radii.
- However, while the calculations are simple, the results can look poor on screen (collision events may occur when the user actually views no collision).

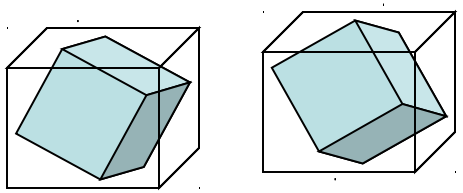
More precise detection

- When entities are an irregular shape, a number of spheres may be used.



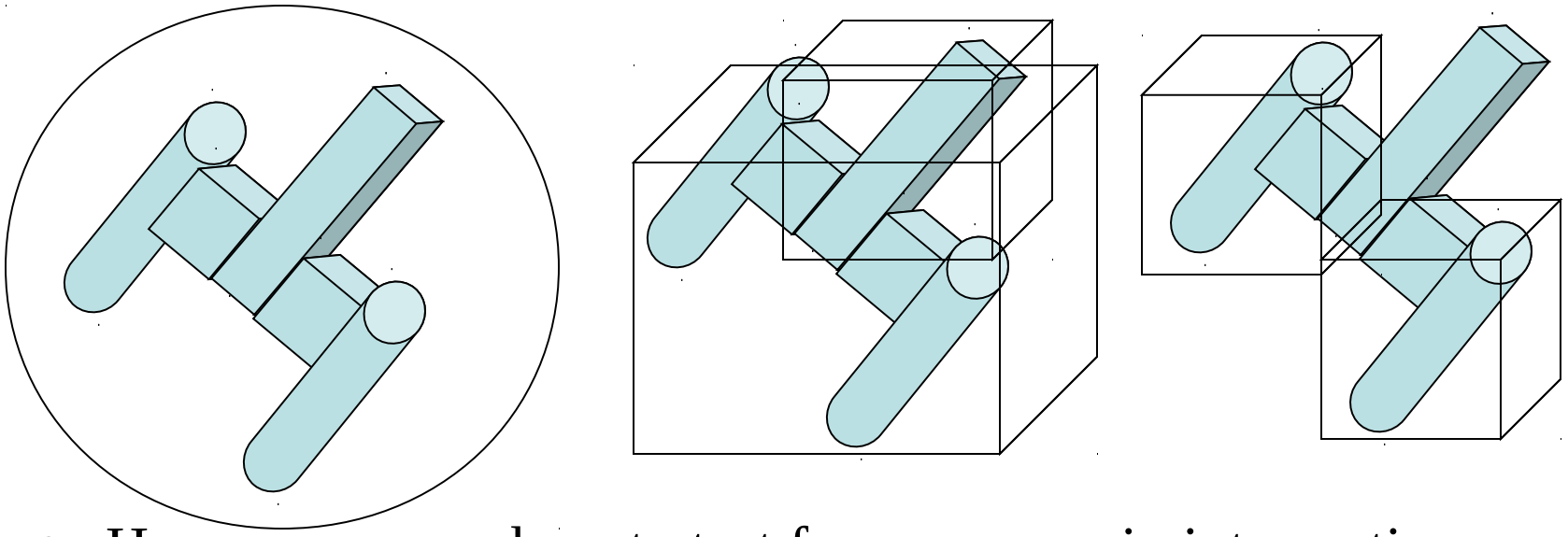
- In this diagram we use three spheres for greater accuracy.
- Geometry in games may be better suited to bounding boxes (not circles).

- To ease calculations, bounding boxes are commonly axis aligned (irrelevant of entity transformations they are still aligned to X, Y and Z coordinates in 3D) These are commonly called AABBs (Axis Aligned Bounding Boxes).



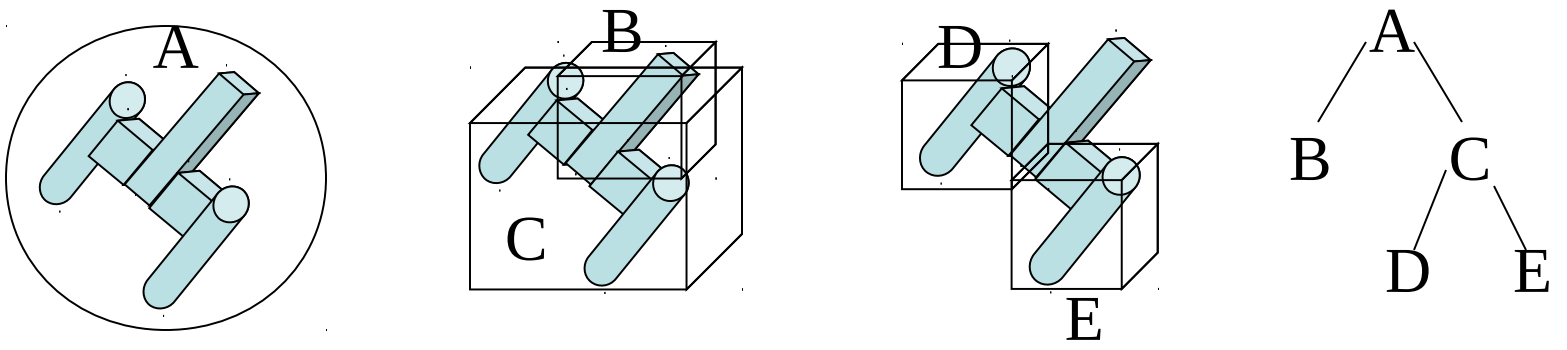
Recursive testing of bounding boxes

- We can build up a recursive hierarchy of bounding boxes to determine quite accurate collision detection.



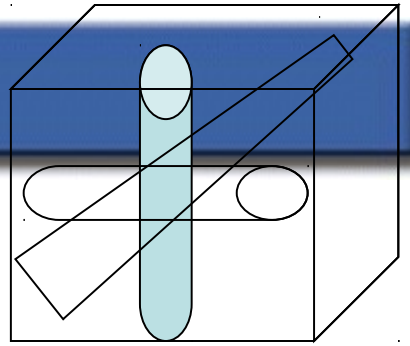
- Here we use a sphere to test for course grain intersection.
- If we detect intersection of the sphere, we test the two sub bounding boxes.
- The lower bounding box is further reduced to two more bounding boxes to detect collision of the cylinders.

Tree structure used to model collision detection



- A recursive algorithm can be used to parse the tree structure for detecting collisions.
- If a collision is detected and leaf nodes are not null then traverse the leaf nodes.
- If a collision is detected and the leaf nodes are null then collision has occurred at the current node in the the structure.
- If no collision is detected at all nodes where leaf nodes are null then no collision has occurred (in our diagram B, C or D must record collision).

Speed over accuracy



- The approach which offers most speed would be to have AABBs of a size fixed at entity initialization time.
  - That is, irrelevant of entity transformation the associated AABB does not change in size or shape.
- This is cumbersome for entities shaped like cylinders.
  - Plenty of “free space” will exist around the cylinder.
- For more realistic collision detection the bounding box may fit closely to the entity and rotate along with associated entity.
  - This requires the bounding box to be recomputed every time an entity is rotated.

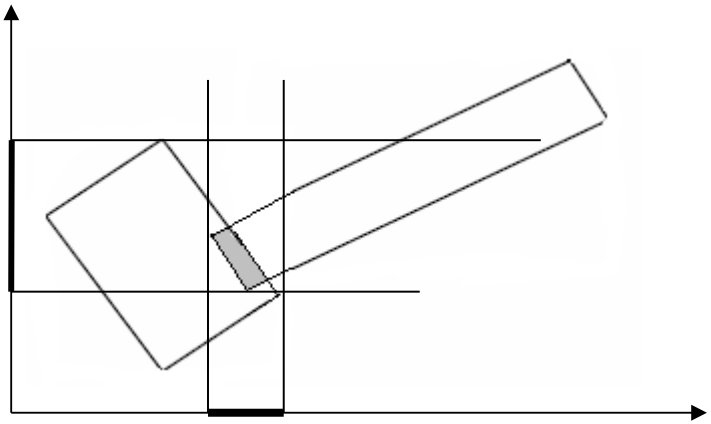
Optimization Structures

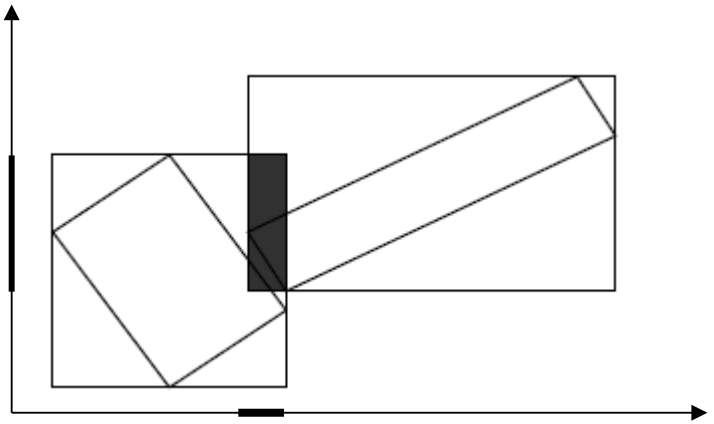
- All of these optimization structures can be used in either 2D or 3D
- Packing in memory may affect caching and performance

Collision Detection

- For each object ***i*** containing polygons ***p***
  - Test for intersection with object ***j*** with polygons ***q***
  - (***j*** > ***i***)
- For polyhedral objects, test if object ***i*** penetrates surface of ***j***
  - Test if vertices of ***i*** straddle polygon ***q*** of ***j***
    - If straddle, then test intersection of polygon ***q*** with polygon ***p*** of object ***i***

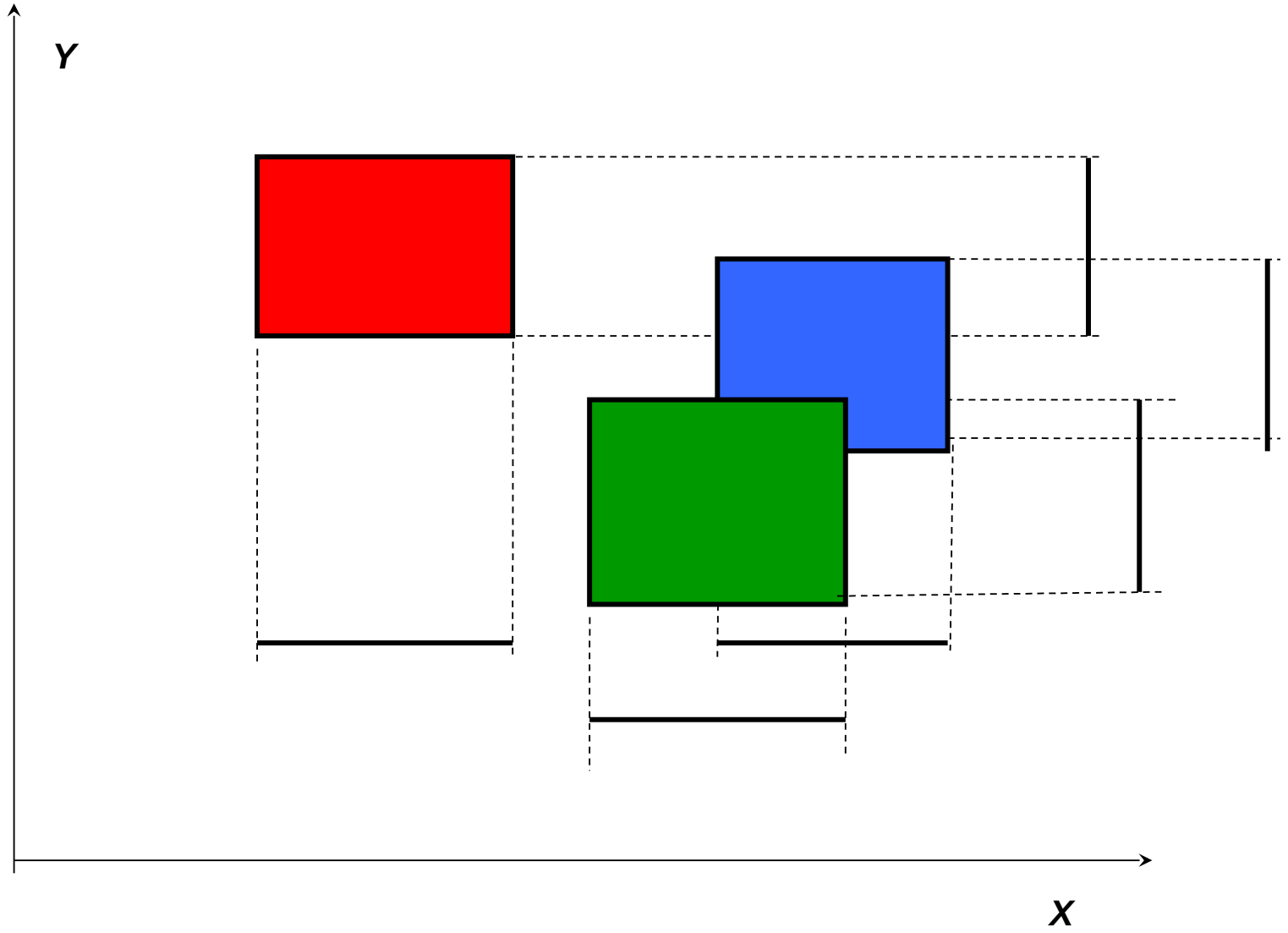
Dimension Reduction



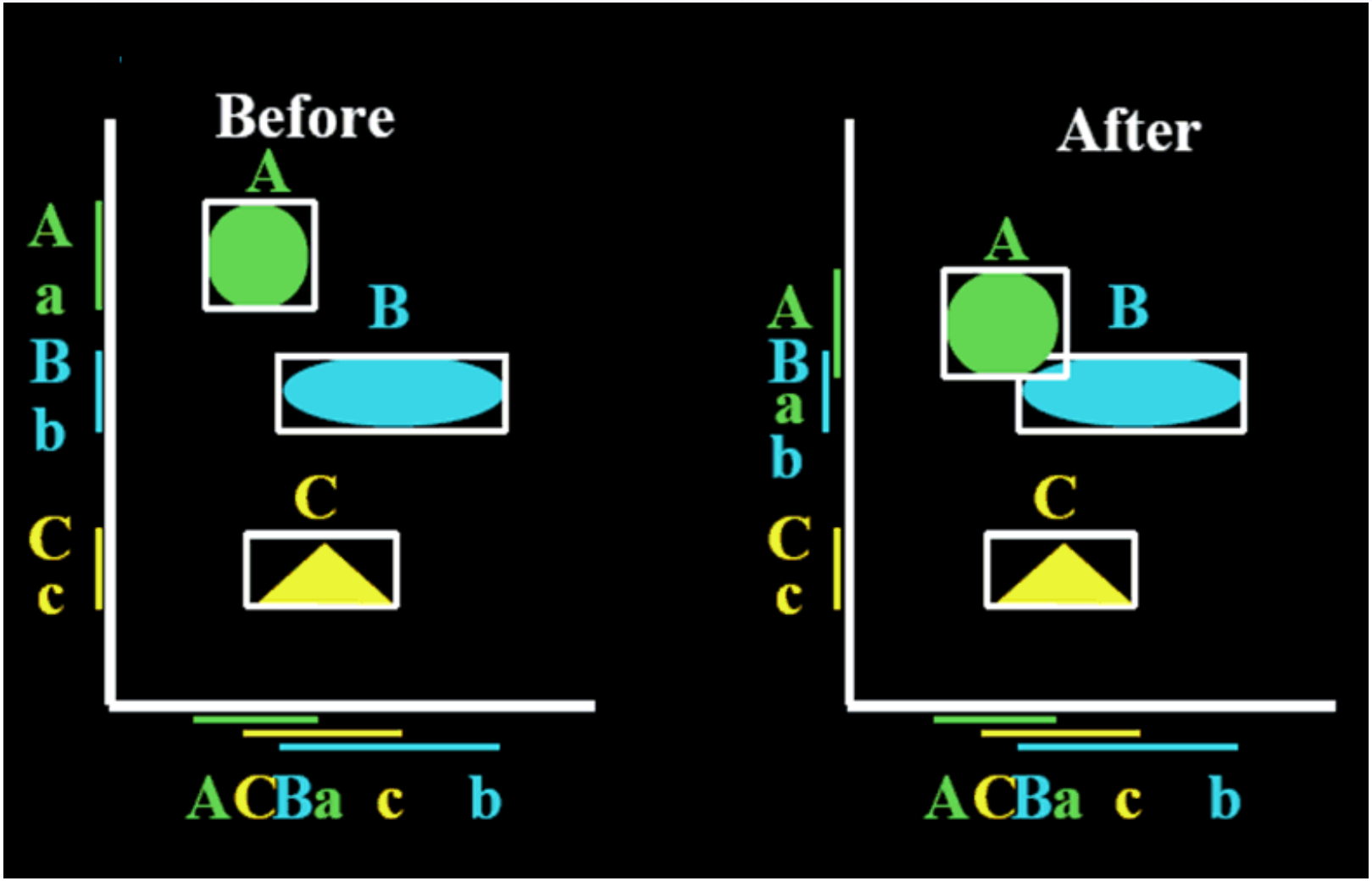


- If two bodies collide in 3-D space, their orthogonal projections onto the xy, yz, and xy planes, and x, y, and z axes must overlap.
- Two AABB intersect, if and only if their projections onto all three axis intersect.
  - This is why the bounding boxes are axis aligned.
- 1-D Sweep and Prune (projection onto x, y, z axes)
- 2D Intersection tests (projection onto xy, yz, xz planes)

Očrtani pravokotniki in odkrivanje trkov



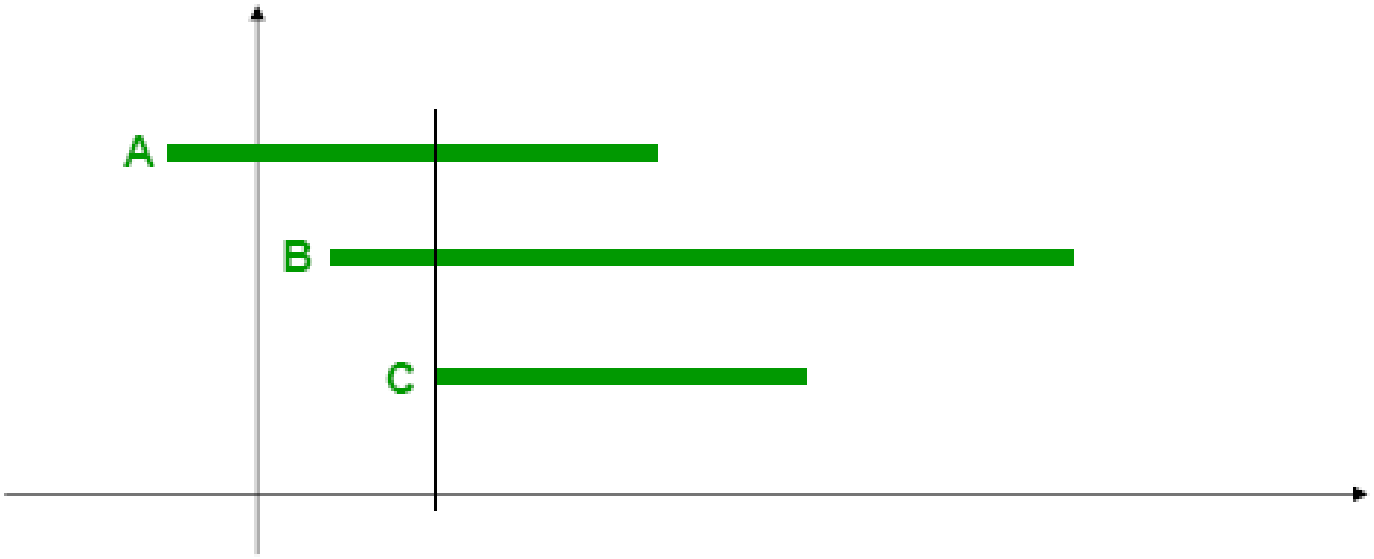
Dimension reduction



Box A moves to overlap box B

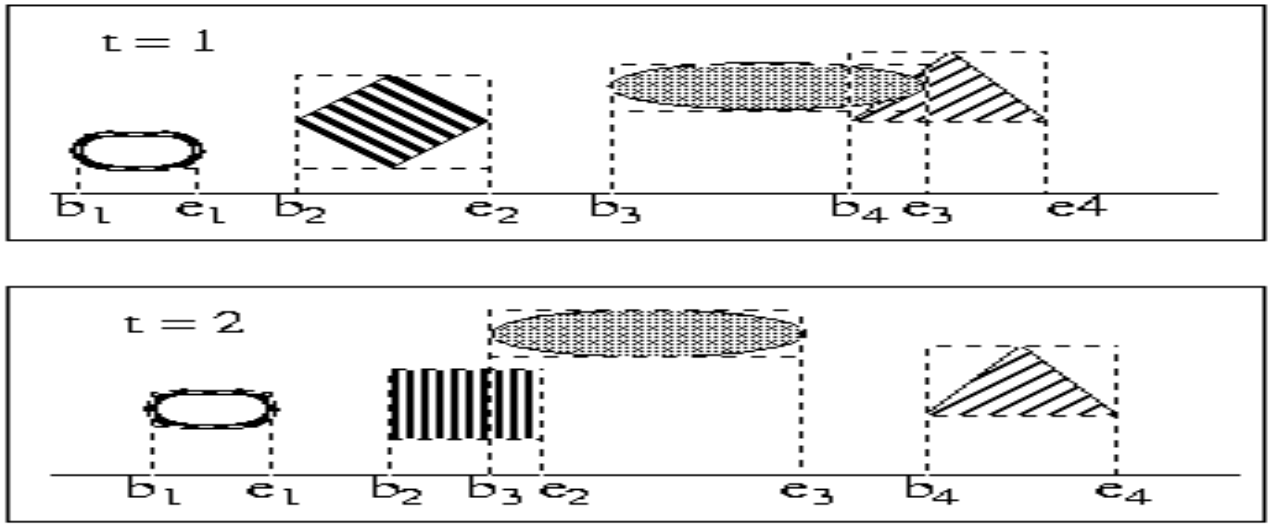
Broad AABB phase

- 3D interval overlap check
- Sweep a sorted interval list
- Maintain an ”active list”



1-D Sweep and Prune

- Bounding Volumes projected onto x, y, and z axes
- Have one sorted list for each dimension
- Maintain Boolean flag which only changes if swaps are made on sorted lists.
  - An upper-triangular $O(n^2)$ matrix



1-D Sweep and Prune

- If flags are true for all 3 dimensions,
  - we pass this pair on to exact collision detection.
- Due to temporal coherence, individual lists are likely almost sorted already.
- Insertion sort works well where large numbers of objects move locally.
  - O(n+s), where s is the number of pairwise overlaps

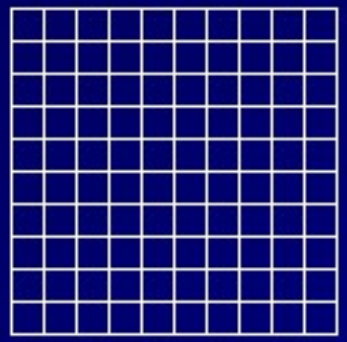
2D Intersection Tests

- Bounding Volumes projected onto xy, yz, and zx planes
  - Three 2D rectangles instead of 1D intervals
- Maintain the interval tree for performing 2D range queries [

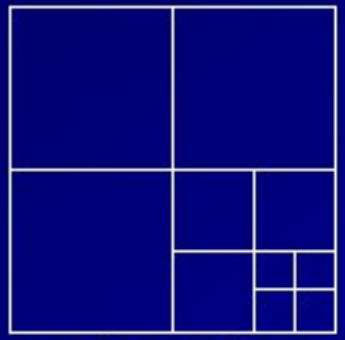
Multi-Body Problems

- Multi-body problems have many objects that could be colliding
  - Most common case in games is lots of players or agents
  - “Collision” may not mean intersecting, could just mean being close enough to react
- It is essential to avoid considering every pair (n^2 cost)
- Spatial subdivision schemes provide one solution
  - Object interactions are detected in a two step process: which cell am I in, then who else is in my cell?
  - Many possible spatial subdivision data structures: fixed grid, Octrees, Kd-trees, BSP trees, …
- Bounding volume schemes provide another solution
  - Hierarchies of bounding volumes

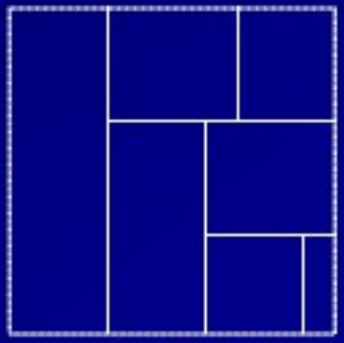
Spatial Data Structures & Subdivision



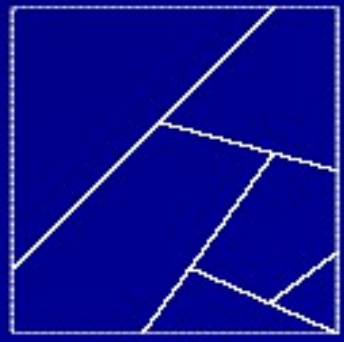
Uniform Spatial Sub



Quadtree/Octree



kd-tree

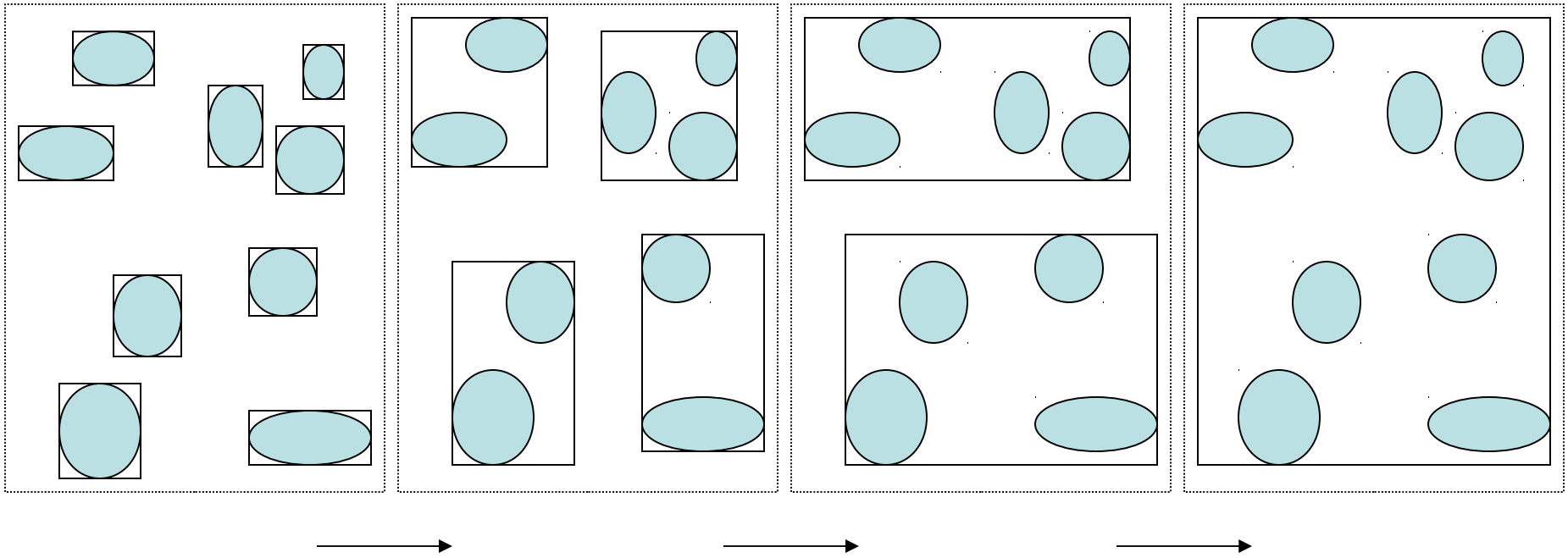


BSP-tree

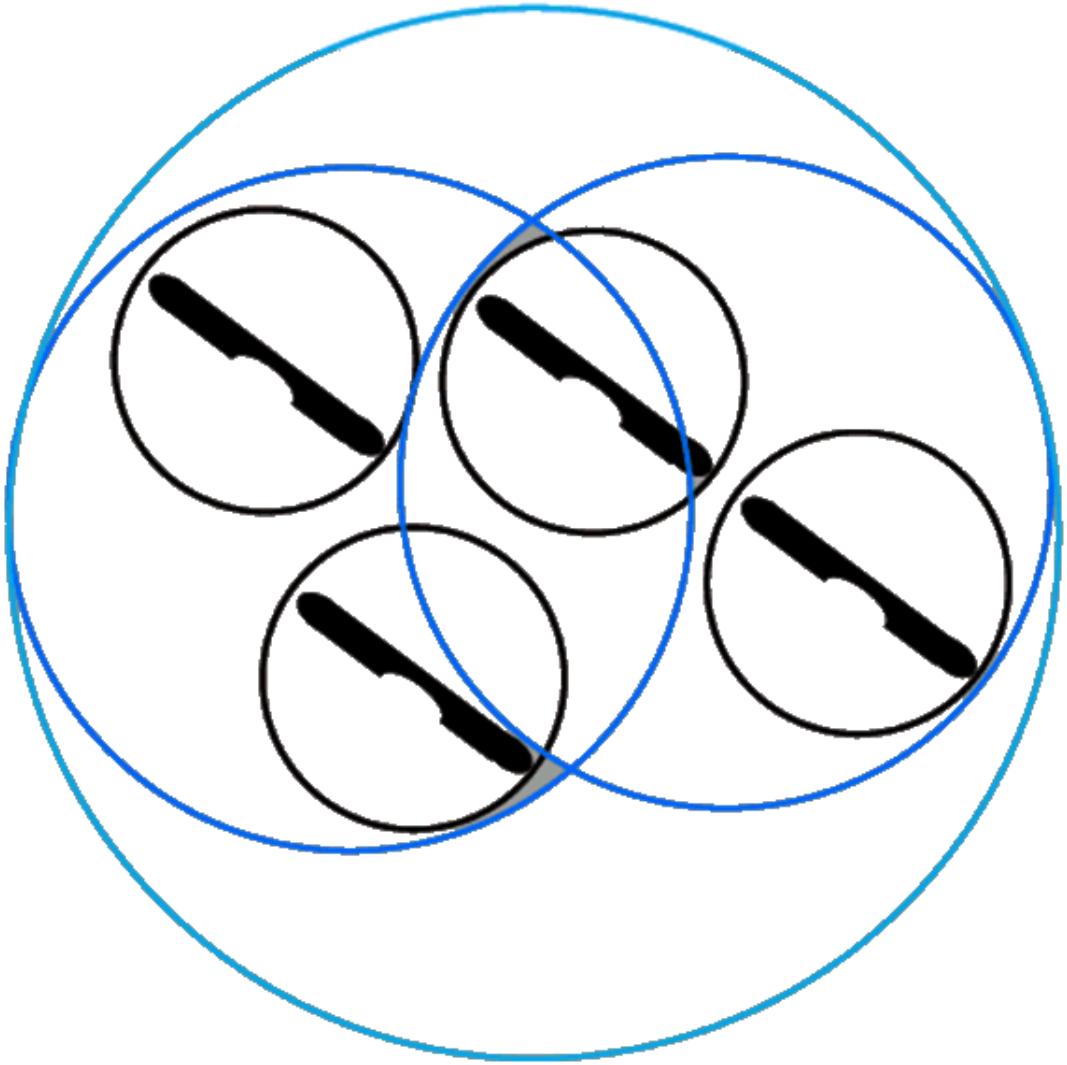
Bounding Volume Hierarchies

- Basic idea is to build a tree where each level of the tree bounds its children
  - Group objects in a hierarchical structure (tree), and recursively build bounding volume for each tree node to form a hierarchy of bounding volumes.
  - Most common bounding volumes are axis-aligned boxes or spheres
  - Generally used for large numbers of static obstacles and a few moving things
    - Put all the static obstacles into the tree
    - Test for intersection between moving object and tree
- Intersection test against a bounding volume hierarchy: recursively traverse the tree, skip the subtree if the bounding volumes don't intersect. A collision occurs if a leaf node's bounding volume intersect with the test object.
- The major difference is that a bounding volume hierarchy does not subdivide all of space, only those parts of space that are occupied

Bounding Tree Example



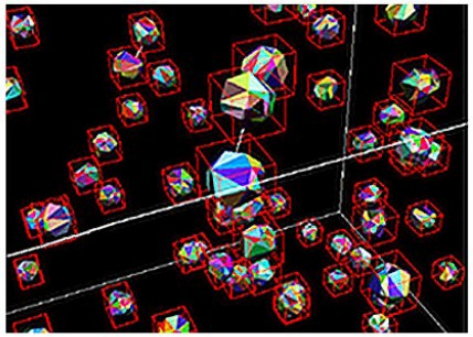
Bounding Volume Hierarchy Example

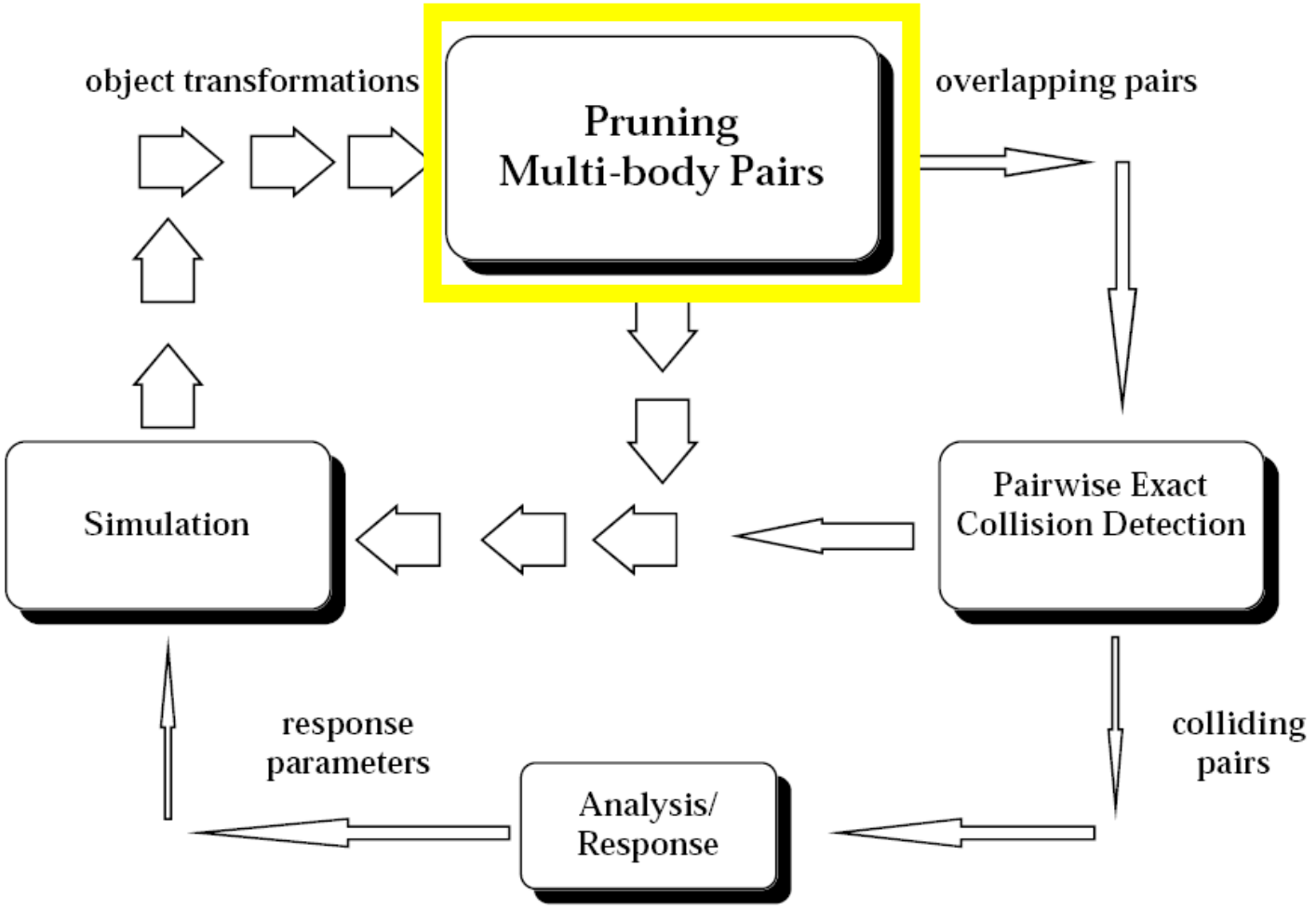


Hierarchical collision testing

```
colliding(a, b)
   if (not overlap (a.bv, b.bv)) return False
   if (isLeaf(a) && isLeaf(b))
      if (any polygon pair intersects) return True
   if (isLeaf(a))
      for each c <- b.children return colliding(a,c)
   for each c <- a.children return colliding(c,b)
```

Architecture for Multi-body Collision Detection





Pruning Multi-body Pairs

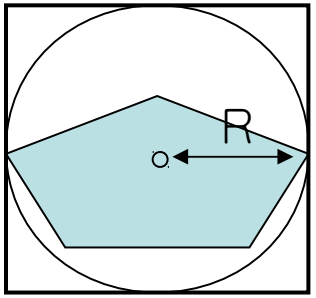
- N moving objects + M static objects

$$\binom{N}{2} + NM \text{ pairs}$$

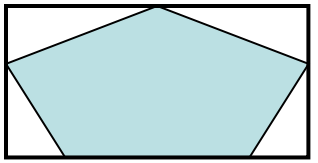
- Objective
  - to reduce the number of pairs of objects that are actually compared, to give O(N+M)
- Use coherence
  - By sorting objects in 3-space
  - Assumes each object is surrounded by a 3-D bounding volume (AABB)
- Use Dimension Reduction to sort the objects in 3-space

3-D Bounding Volumes

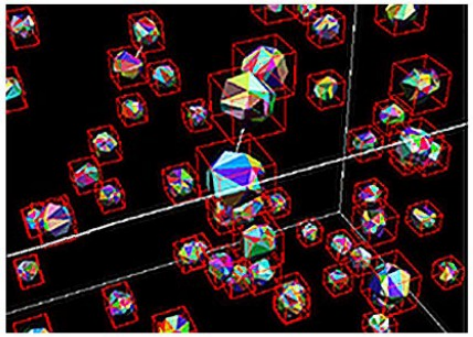
- Types
  - Fixed-Size Bounding Cubes
    - Recalculating fixed size bounding cubes add less overhead, as their size remains constant.

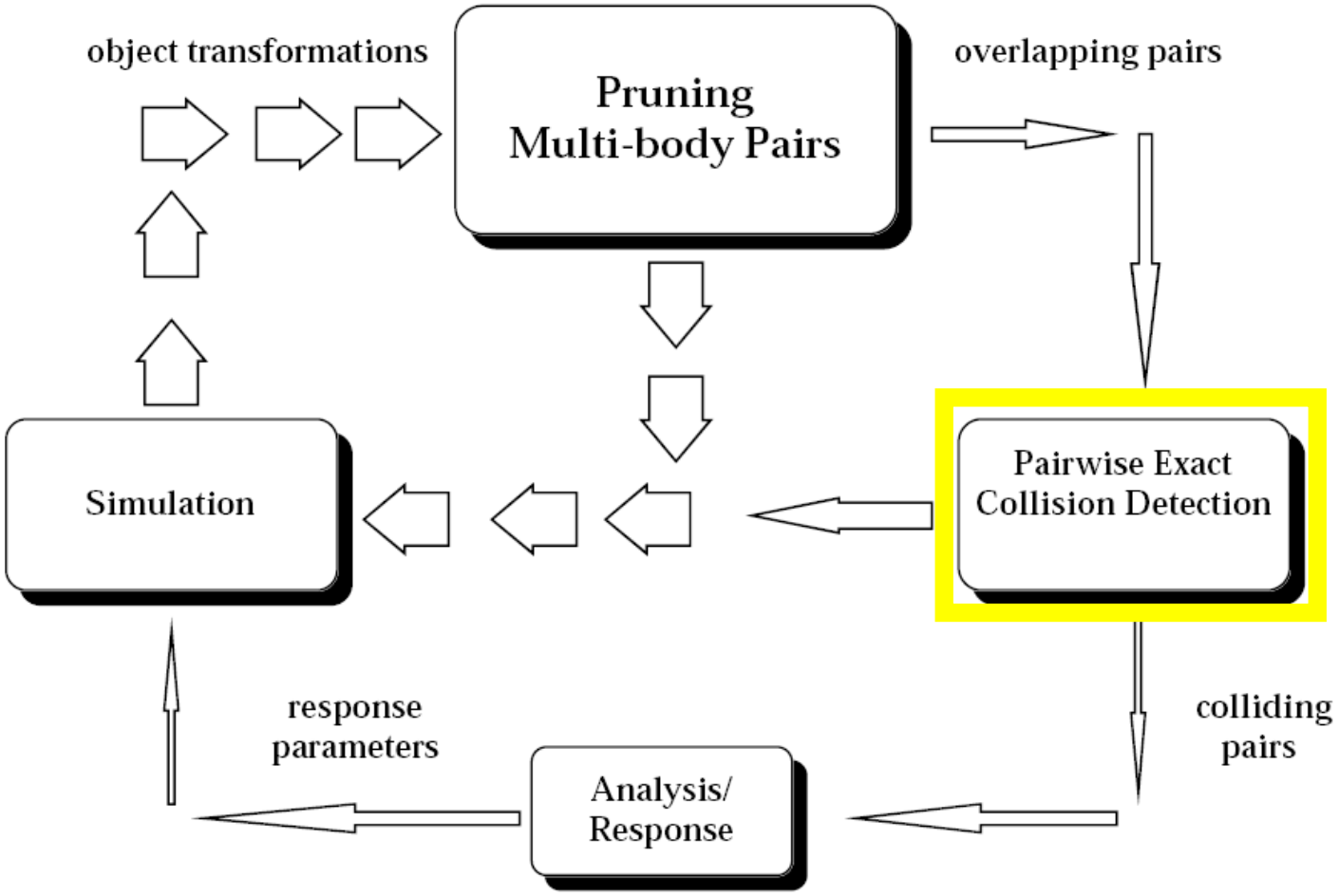


  - Dynamically-Resized Rectangular bounding bounding boxes.
    - Dynamically resized bounding boxes are the “tightest” axis-aligned box containing the object at a particular orientation.
    - More overhead due to recomputation of min, max x,y,z values



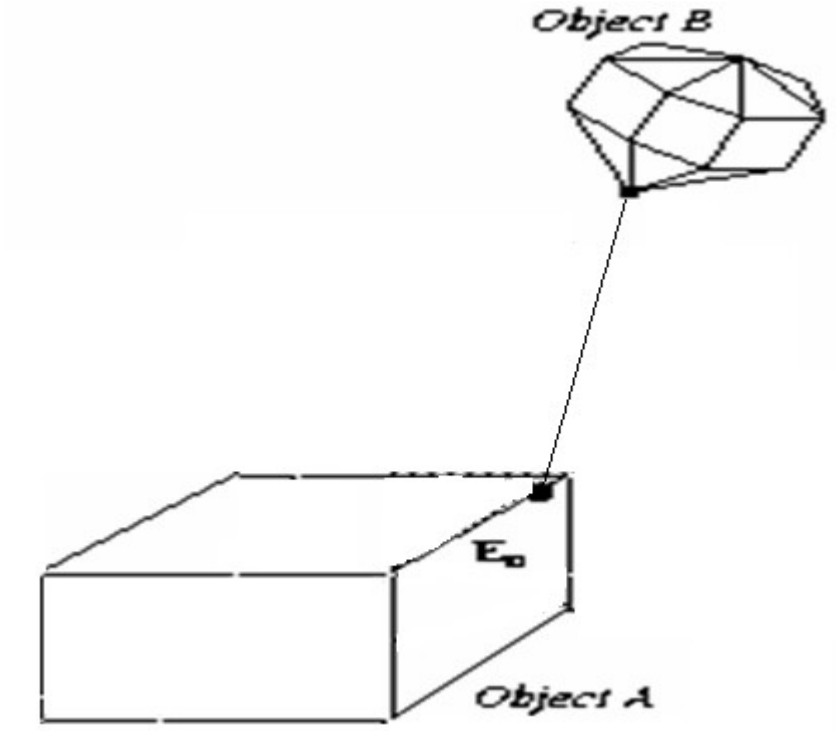
Architecture for Multi-body Collision Detection





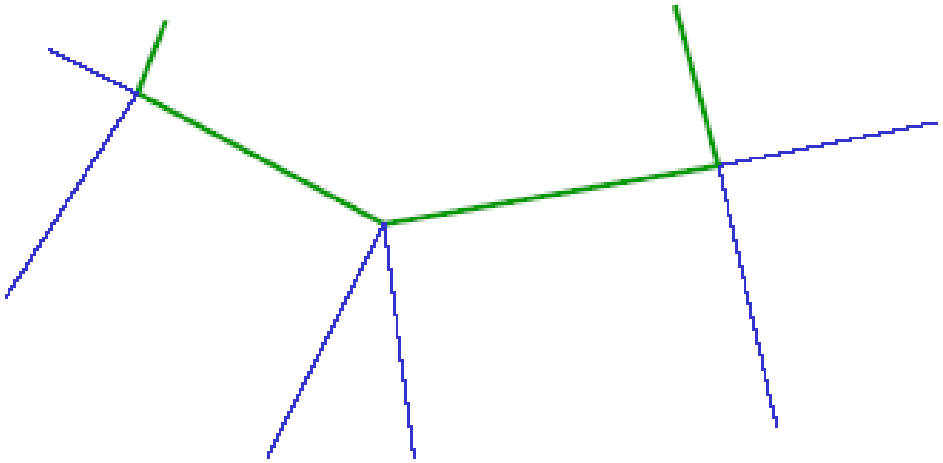
Pairwise Collision Detection for Convex Polytopes

- The Lin-Canny collision detection algorithm
  - Tracks closest points between pairs of convex polytopes
  - Gives closest features even if objects do not intersect



Lin-Canny algorithm

- Near constant time performance
- Actually proximity detection algorithm
- Incrementally tracking closest ”features”
  - features: vertices, edges, faces
- Constructs Voronoi regions in advance



Lin-Canny algorithm, contd

- compute pair of closest points of two objects
- determine which features they belong to
- for each time step and each feature pair do
  - compute closest points of current feature pair
  - if distance small, signal collision
  - check that points are still in respective regions
  - otherwise update closest feature pair

Collision Detection

- Discretization in 3D
- Create a voxel array
  - Store an ID in each voxel where an object is
  - Collision where voxel has an ID already
  - Difficult to determine good voxel size
  - Huge memory
  - Hash table -- gives constant time point queries

Inter-Object Distance

- A related problem:
  - *Collision Avoidance*
- Part of motion planning is to avoid collisions
- Many collision detection algorithms take time into account
- Estimate Collision Time, distance
- www.cs.unc.edu/~geom/collision.html

Collision Detection

- Cheaper distance calculation:

$$d = sqrt((x_1 - x_2)^2 + (y_1 - y_2)^2)$$

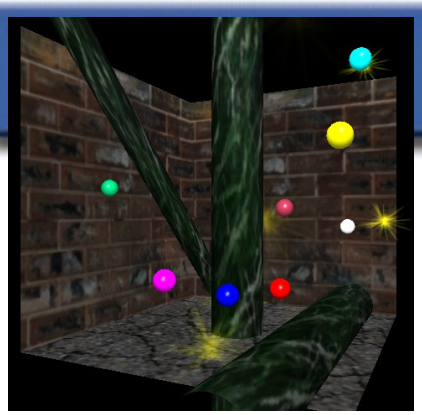
  - Compare against d^2

- Approximation: $d = abs(x_1 - x_2) + abs(y_1 - y_2) - \min(abs(x_1 - x_2), abs(y_1 - y_2))/2$

  - Manhattan distance - Shortest side/2

$$dx = 3, dy = 4 \longrightarrow d = 5$$

$$d' = 3 + 4 - 1.5 = 5.5$$



Bodies in Collision
Collisions and Contact

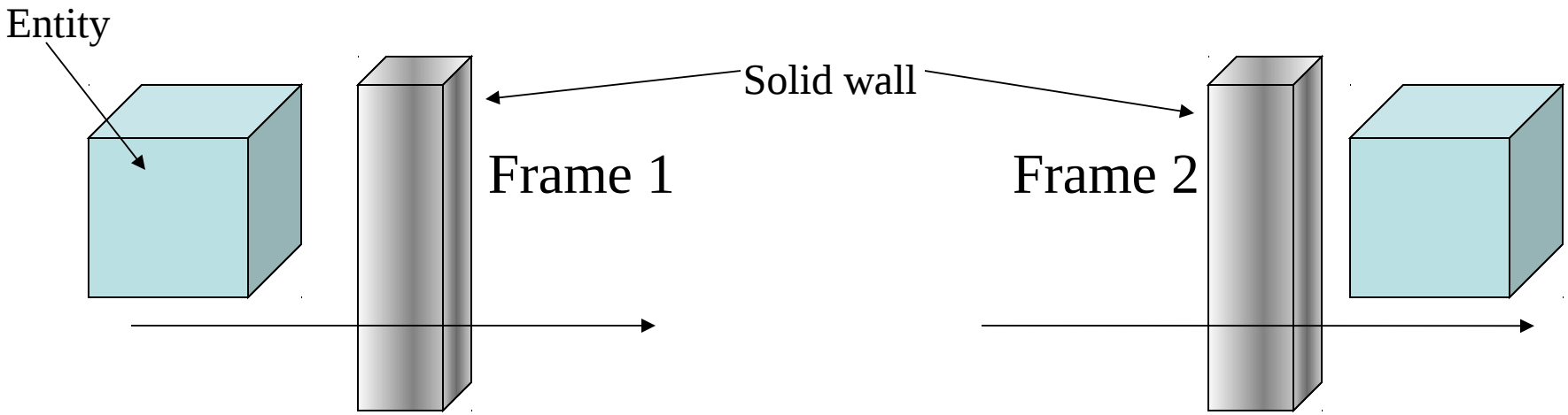
So far, no interaction between rigid bodies

Collision detection –
determining if, when and where a collision occurs

Collision response –
calculating the state (velocity, …) after the collision

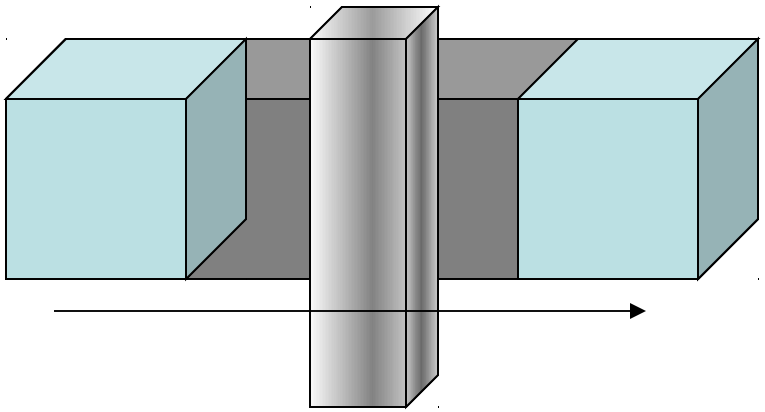
A problem with frame rate and collision

- An entity moves around the screen at a given speed. This speed may be increasing, decreasing (accelerating or decelerating) or may be static.
- If the speed of an entity is such that the distance moved on screen is sufficiently large per frame rate, then simply identifying if vertices cross is not an appropriate solution.
  - They may never cross!



Possible solutions – Projecting bounding box.

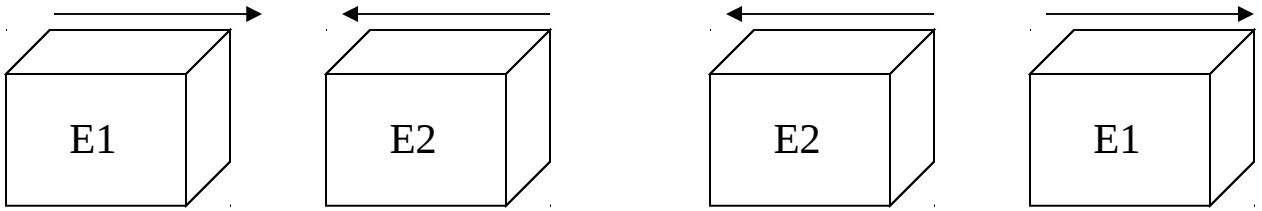
- Produce bounding box around an entity at current frame and next frame position (look ahead) and check collisions using this new shape.



- For this method to work we are actually carrying out twice as much detection as required. This is very time consuming and will possibly reduce performance (even slower frame rate).

Considering time in collision detection

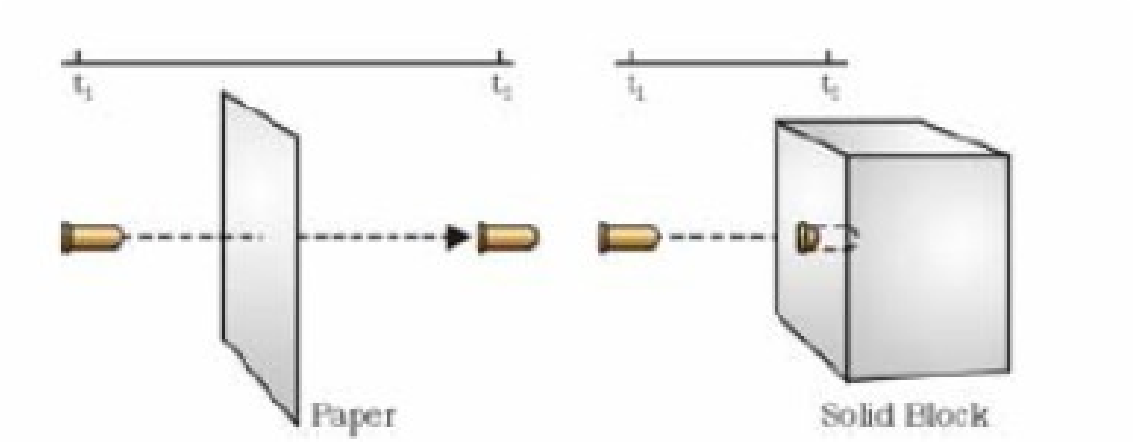
- Assume entities E1 and E2 are displayed in different positions at the following frame times t1, t2 (giving the illusion of motion).



- We know they collided, but as far as the display is concerned no collision has occurred. (they have passed through each other).
  - We need to introduce the notion of time into our equations to aid collision detection.

Missed collisions and interpenetration

- When collisions are detected too late
- Hard problem
  - Decrease time step
  - Avoid fast moving objects
  - Avoid thin objects

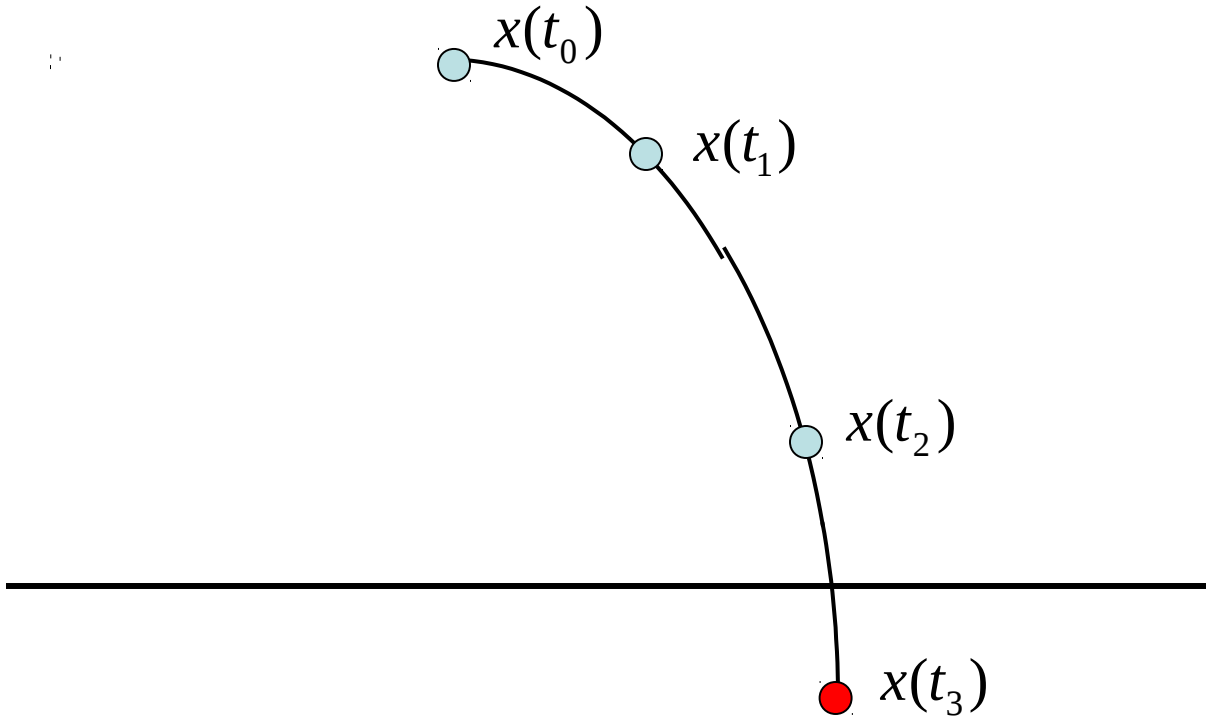


Possible solutions – Using time in our equations

- Use a recursive algorithm to check for intersection at lower time intervals (even though these will not be displayed on screen).
- Carry out our calculations in 4D (including time). This is complicated physics and computationally draining of valuable CPU time.
- In real-time games (such as first person shooters) these types of calculations are just too intensive.

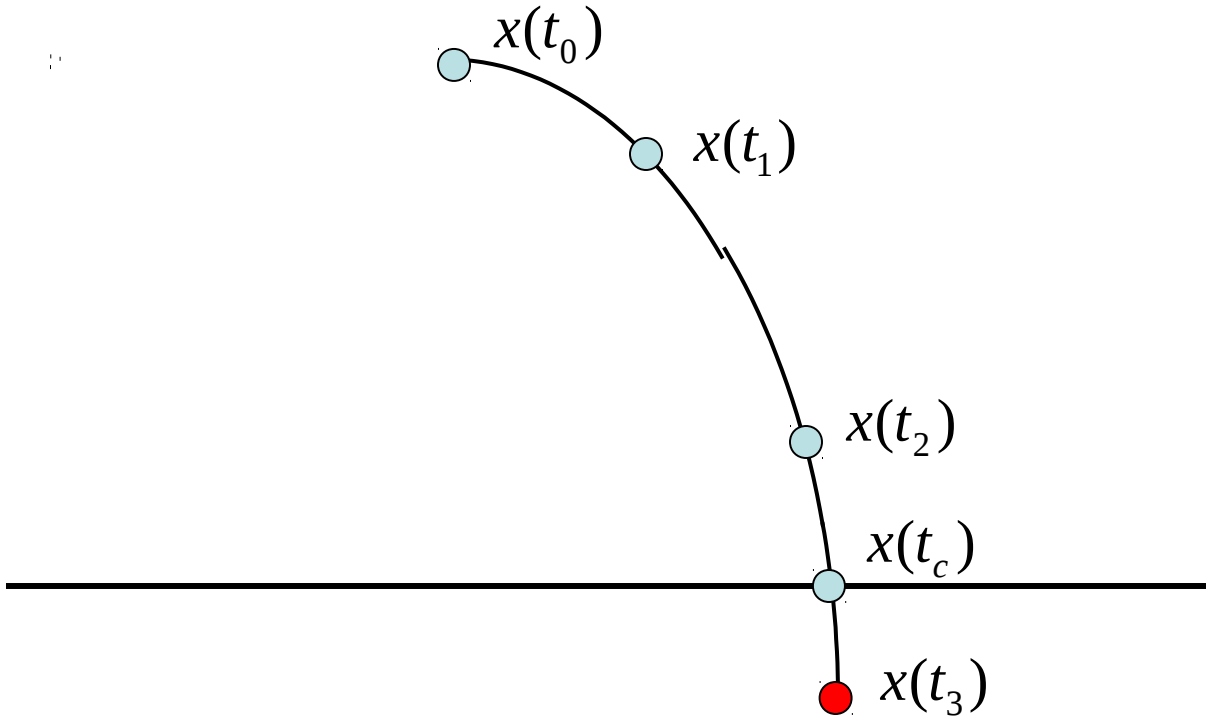
Collisions and Contact

What should we do when there is a collision?



Rolling Back the Simulation

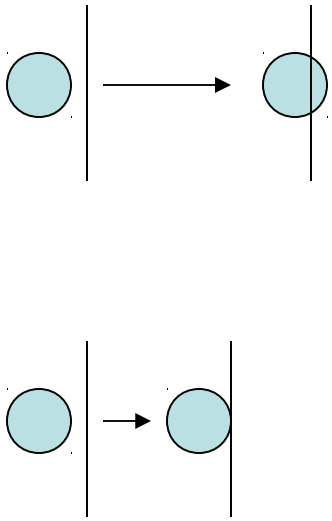
Restart the simulation at the time of the collision



Collision time can be found by bisection, etc.

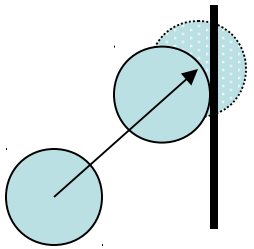
Precise Collision Times

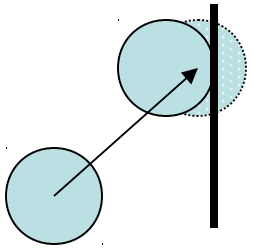
- Generally a player will go from not intersecting to interpenetrating in the course of a frame
- We typically would like the exact collision time and place
  - Better collision response calculation
  - Interpenetration may be algorithmically hard to manage
  - Interpenetration is difficult to quantify
  - A numerical root finding problem
- More than one way to do it:
  - Hacked (but fast) clean up
  - *Interval halving* (binary search)



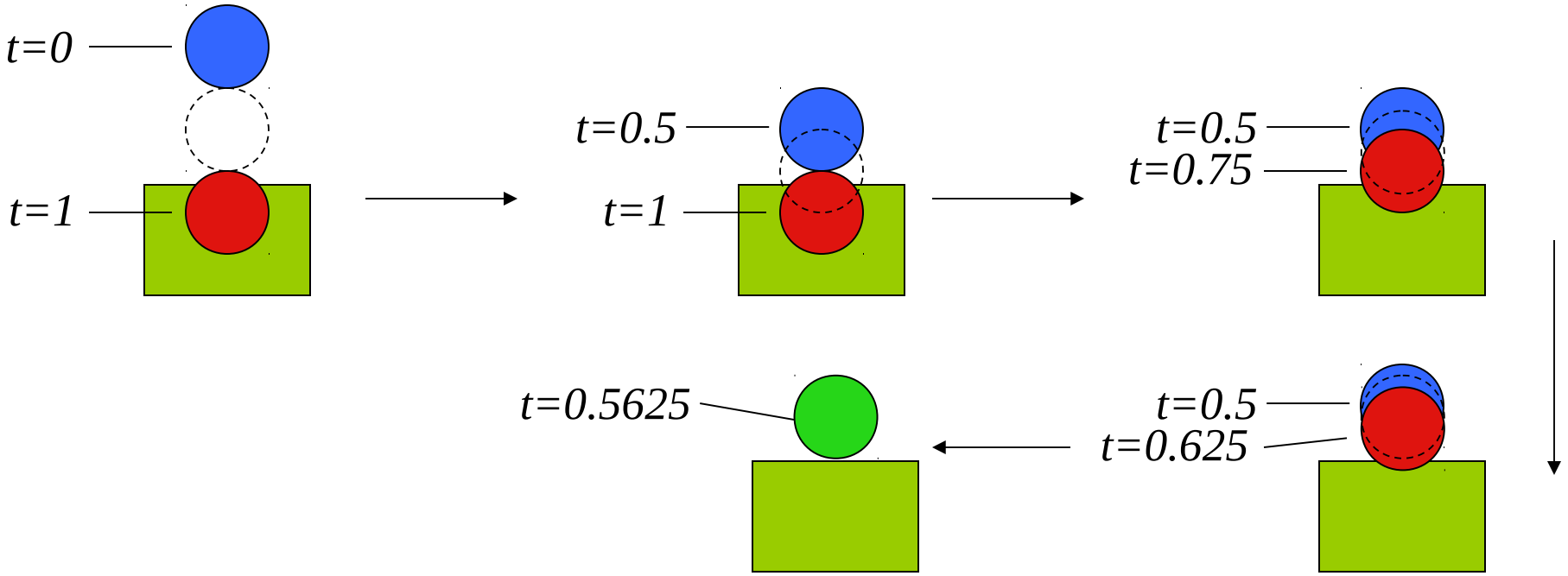
Collision Time

- Given a ball at initial position $p0 = (x1, x2, x3)$, with initial velocity $v = (v1, v2, v3)$, and a constant acceleration $a = (a1, a2, a3)$. When will it hit a plane $f(x, y, z) = 0$
  - $p = p0 + vt + ½\ at2$
  - Solve the equation $f(p) = 0$
- But it is the center of the ball that intersects the plane. To be more accurate, we can simply move the position so that the objects just touch, and leave the time the same. (Hacked clean up)
- Multiple choices for how to move:
  - Back along the motion path
  - Shortest distance to avoid penetration





Interval Halving Example



Interval Halving

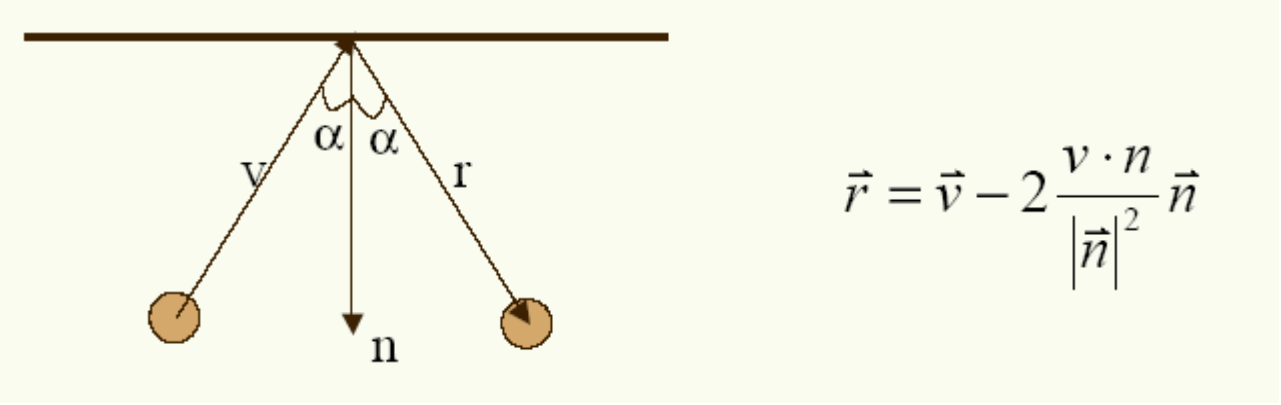
- Search through time for the point at which the objects collide
- You know when the objects were not penetrating (the last frame)
- You know when they are penetrating (this frame)
- So you have an upper and lower bound on the collision time
  - Later than last frame, earlier than this frame
- The aim is to do a series of tests to bring these bounds closer together
- Each test checks for a collision at the midpoint of the current time interval
  - If there is a collision, the midpoint becomes the new upper bound
  - If not, the midpoint is the new lower bound
- Keep going until the bounds are the same (or as accurate as desired)

Interval Halving Evaluation

- Advantages:
  - Finds accurate collisions in time and space, which may be essential
  - Not too expensive
- Disadvantages:
  - Takes longer than a hack (but note that time is bounded, and you get to control it)
  - May not work for fast moving objects and thin obstacles
    - Why not?
- Still, it is the method of choice for many applications

Collision Response

- What happens if there is a collision?
  - A complex issue, no simple and uniform solutions. It is game dependent.
  - Use game physics to find the answer in simple situations.
  - Example: in the game of pool, you can calculate the bounces of balls using Newtonian physics (momentum and kinetic energy conservation).



$$\vec{r} = \vec{v} - 2\frac{v \cdot n}{\left|\vec{n}\right|^2}\vec{n}$$

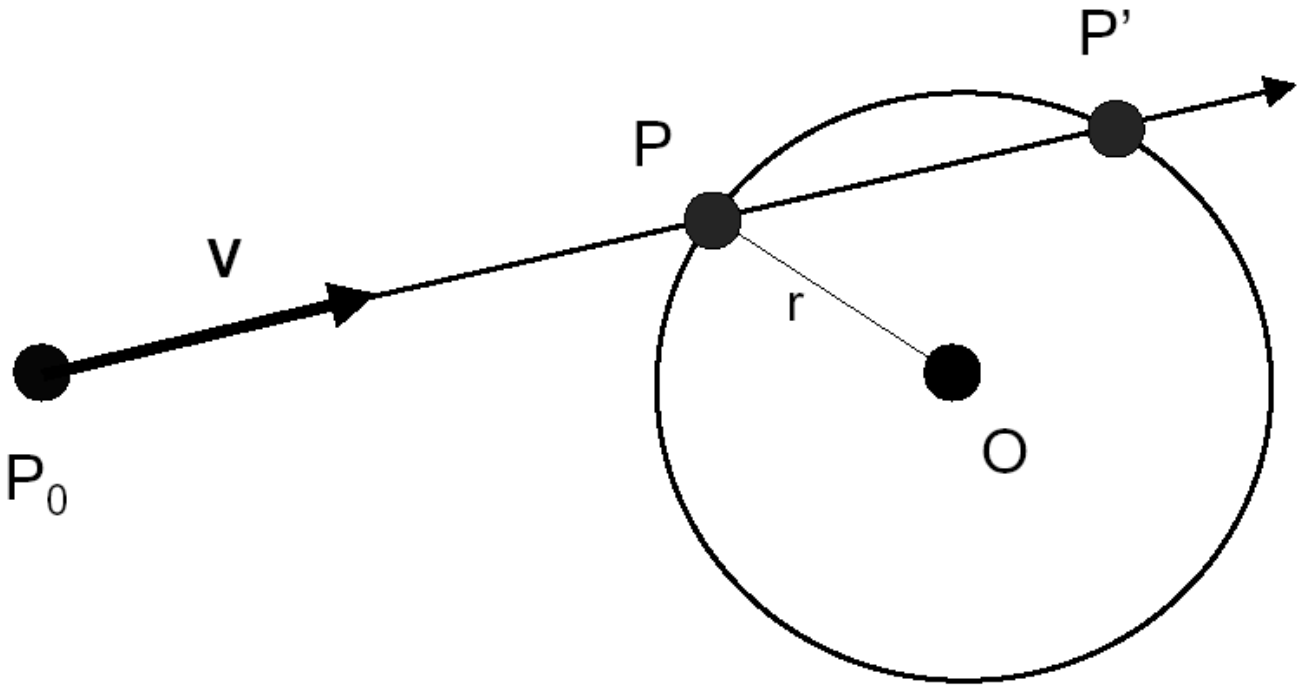
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees

Ray-Sphere Intersection

Ray: $P = P_0 + tV$

Sphere: $|P - O|^2 - r^2 = 0$



Ray-Sphere Intersection I

Ray: $P = P_0 + tV$

Sphere: $|P - O|^2 - r^2 = 0$

Algebraic Method

Substituting for P, we get:

$$|\mathbf{P_0} + \mathbf{tV} - O|^2 - r^2 = 0$$

Solve quadratic equation:

$$at^2 + bt + c = 0$$

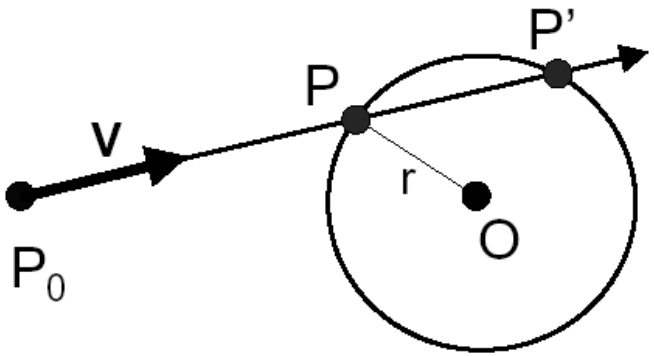
where:

$a = 1$

$b = 2\,V \bullet (P_0 - O)$

$c = |P_0 - C|^2 - r^2 = 0$

$P = P_0 + tV$



Ray-Sphere Intersection II

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

$L = O - P_0$

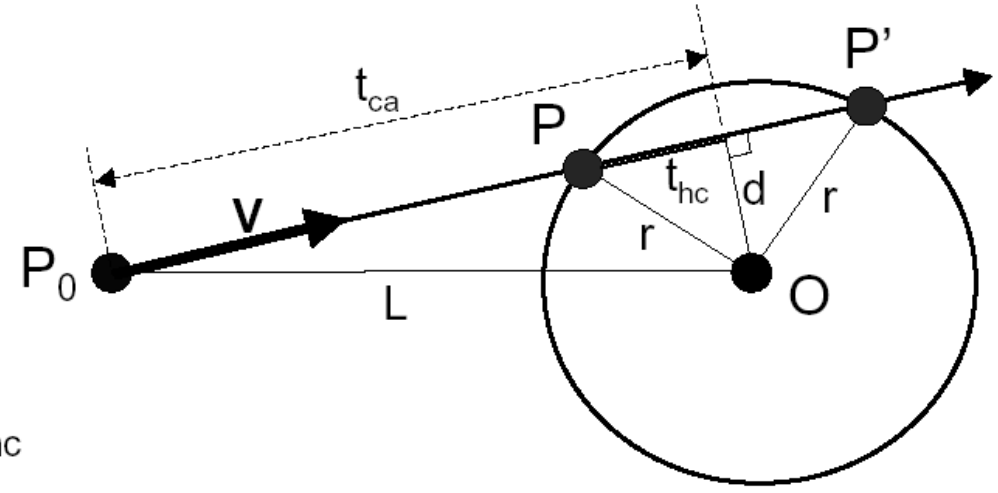
$t_{ca} = L \bullet V$
if ($t_{ca} < 0$) return 0

$d^2 = L \bullet L - t_{ca}^2$
if ($d^2 > r^2$) return 0

$t_{hc} = \text{sqrt}(r^2 - d^2)$
$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$

$P = P_0 + tV$

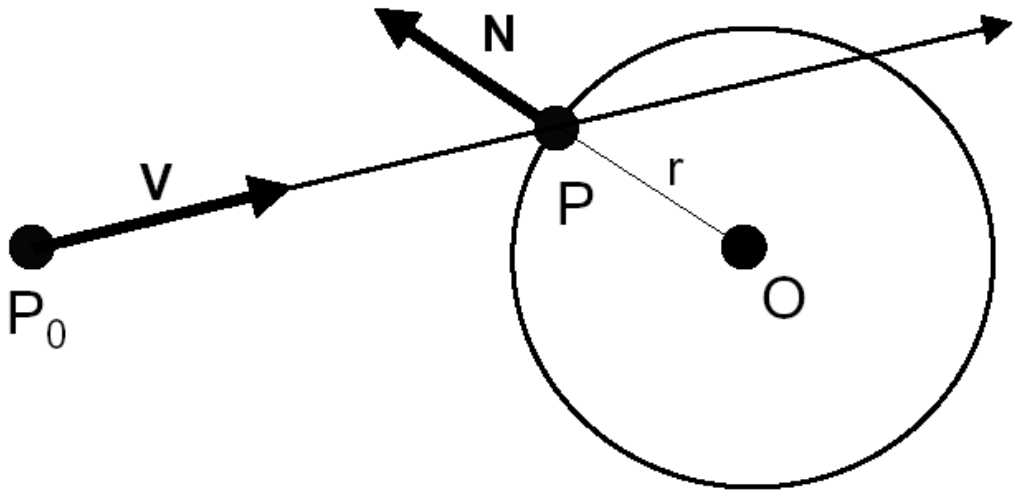
Geometric Method



Ray-Sphere Intersection

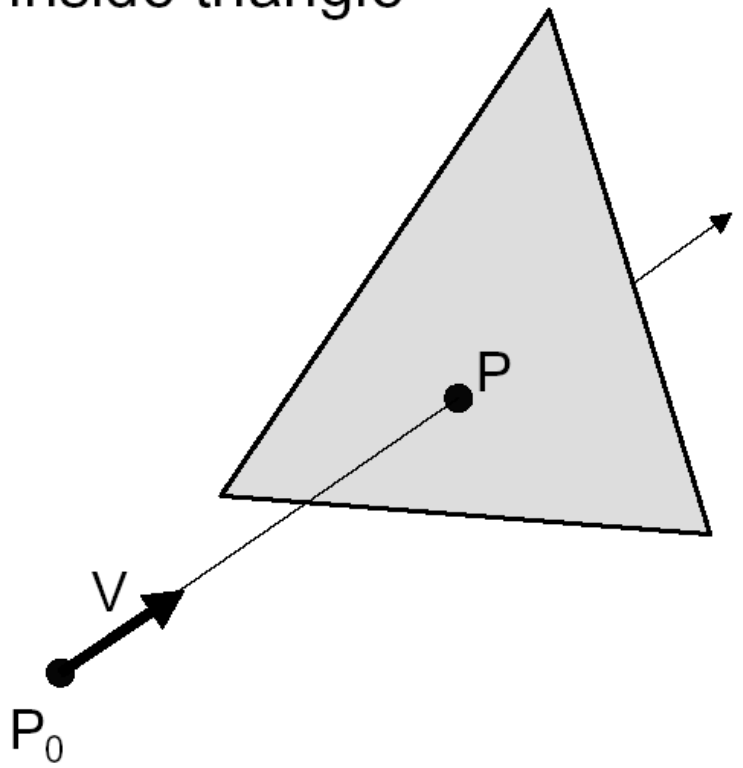
- Need normal vector at intersection for lighting calculations

$$N = (P - O) / ||P - O||$$



Ray-Triangle Intersection

- First, intersect ray with plane
- Then, check if point is inside triangle



Ray-Plane Intersection

Ray: $P = P_0 + tV$
Plane: $P \cdot N + d = 0$

Algebraic Method

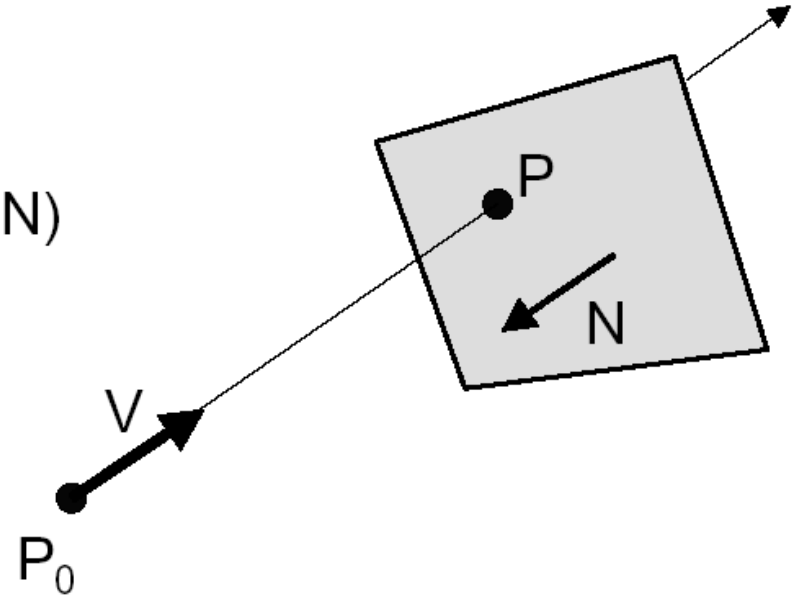
Substituting for P, we get:

$$(\mathbf{P_0 + tV}) \cdot N + d = 0$$

Solution:

$$t = -(P_0 \cdot N + d) / (V \cdot N)$$

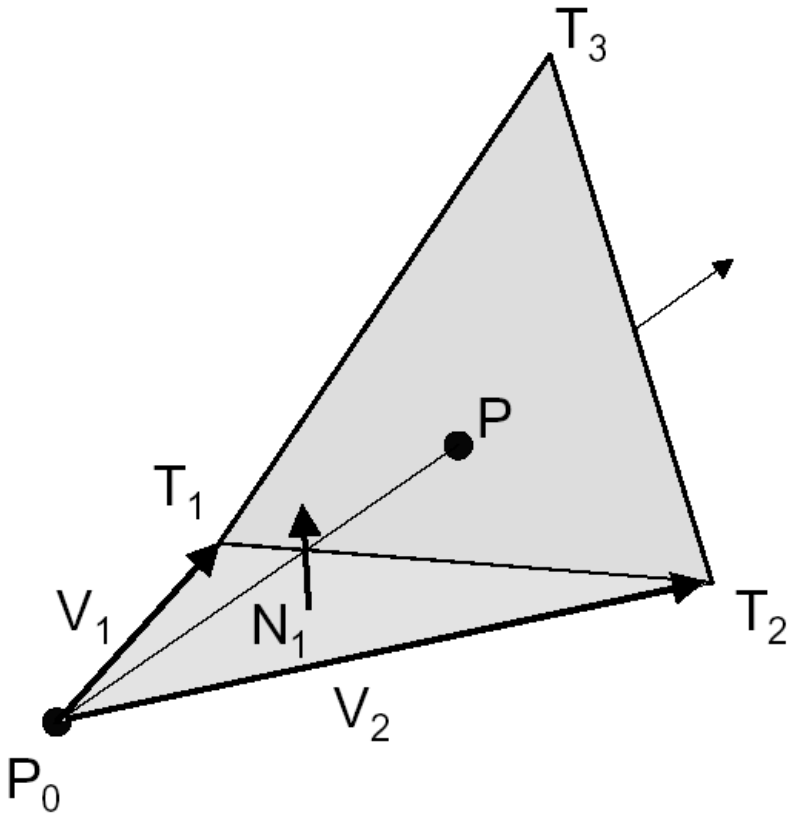
$$P = P_0 + tV$$



Ray-Triangle I Intersection

- Check if point is inside triangle algebraically

For each side of triangle
    $V_1 = T_1 - P$
    $V_2 = T_2 - P$
    $N_1 = V_2 \times V_1$
    Normalize N_1
    $d_1 = -P_0 \cdot N_1$
    if $((P \cdot N_1 + d_1) < 0)$
        return FALSE;
end



Ray-Triangle II Intersection

- Check if point is inside triangle parametrically

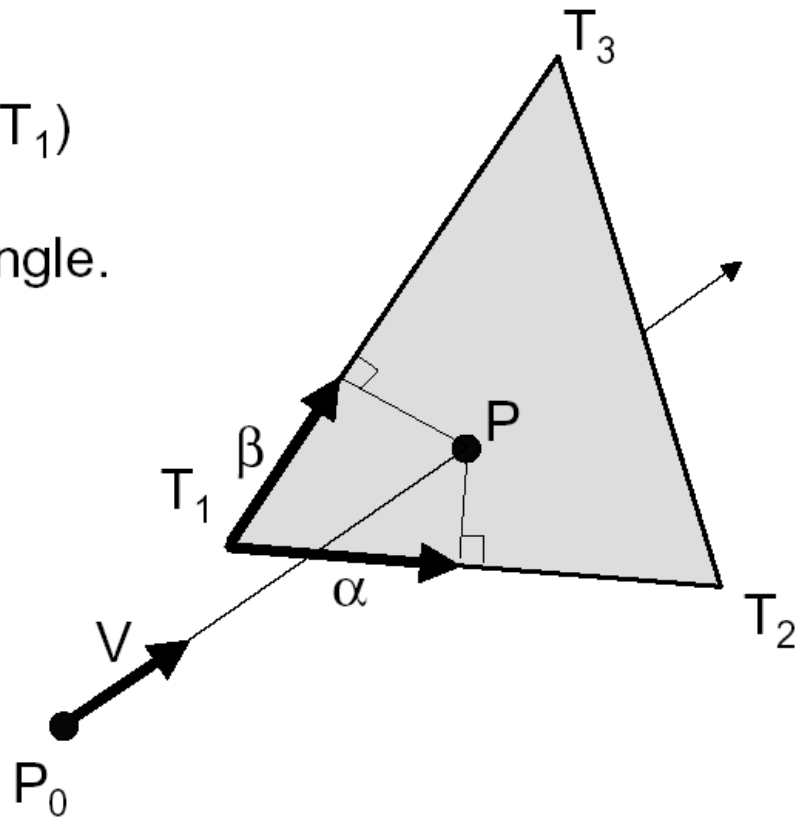
Compute α, β:

$$P = \alpha (T_2 - T_1) + \beta (T_3 - T_1)$$

Check if point inside triangle.

$$0 \le \alpha \le 1 \text{ and}$$

$$0 \le \beta \le 1$$



Other Ray-Primitive Intersections

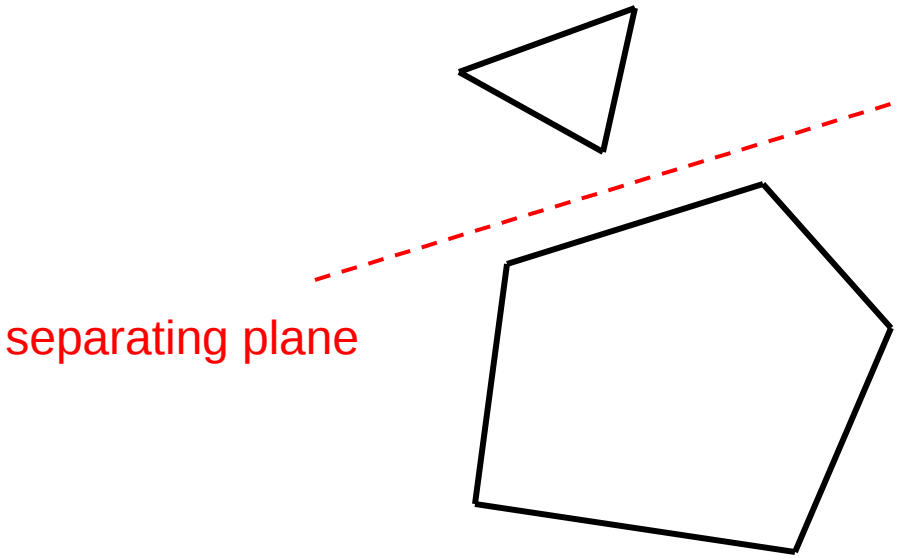
- Cone, cylinder, ellipsoid:
  - Similar to sphere
- Box
  - Intersect 3 front-facing planes, return closest
- Convex polygon
  - Same as triangle (check point-in-polygon algebraically)
- Concave polygon
  - Same plane intersection
  - More complex point-in-polygon test

Intersection Tests: How About Object-Object ?

- Methods for:
  - Spheres
  - Oriented Boxes
  - Capsules
  - Lozenges
  - Cylinders
  - Ellipsoids
  - Triangles
- But first let’s check some basic issues on collision/response…

Collision Detection

Exploit coherency through witnessing



Two convex objects are non-penetrating iff there exists a separating plane between them

First find a separating plane and see if it is still valid after the next simulation step

Speed up with bounding boxes, grids, hierarchies, etc.

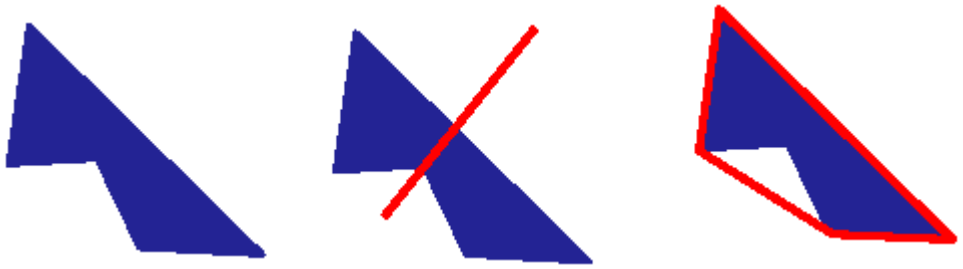
Collision Detection

- Convex objects
  - Look for separating plane
    - Test all faces

    - Test each edge from obj 1 against vertex of obj 2
  - Save separating plane for next animation frame

Collision Detection

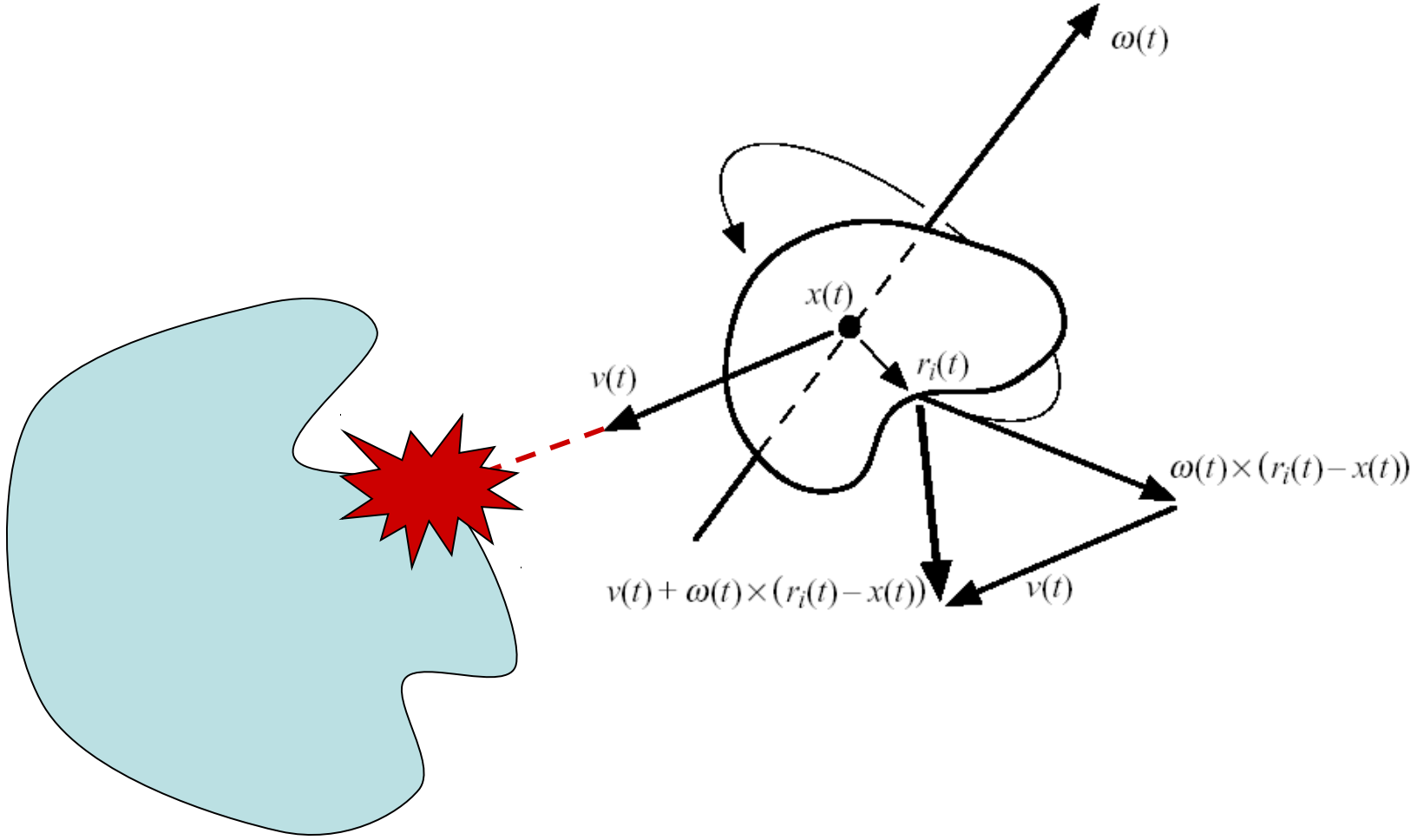
- Concave Objects
  - Break apart
  - Convex hull
    - Automatic or artist-created



Collision Detection

- To go faster
  - Sort on one dimension
    - Bucket sort (i.e. discretize in 1 dimension)
  - Exploit temporal coherence
    - Maintain a list of object pairs that are close to each other
    - Use current speeds to estimate likely collisions
  - Use cheaper tests

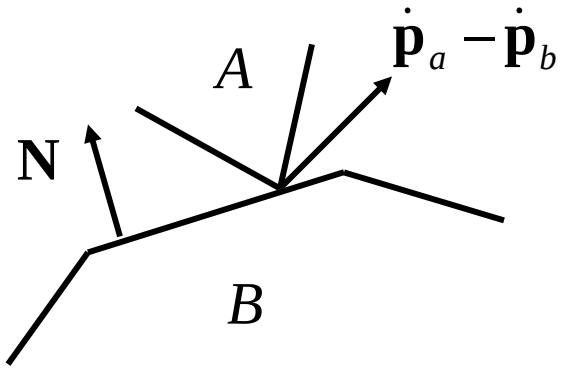
Collision Detection



Collision Detection

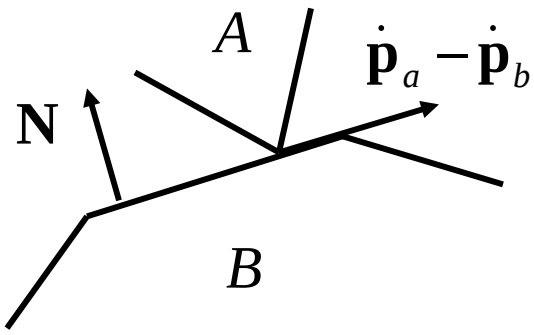
Conditions for collision

$$\dot{\mathbf{p}}_a(t) = \mathbf{v}_a(t) + \boldsymbol{\omega}_a(t) \times (\mathbf{p}_a(t) - \mathbf{x}_a(t))$$



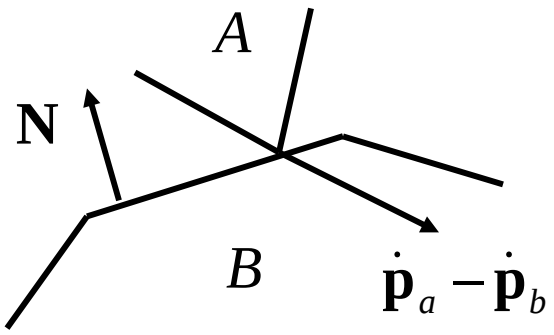
$\mathbf{N} \cdot (\dot{\mathbf{p}}_a(t) - \dot{\mathbf{p}}_b(t)) > 0$

separating



$\mathbf{N} \cdot (\dot{\mathbf{p}}_a(t) - \dot{\mathbf{p}}_b(t)) = 0$

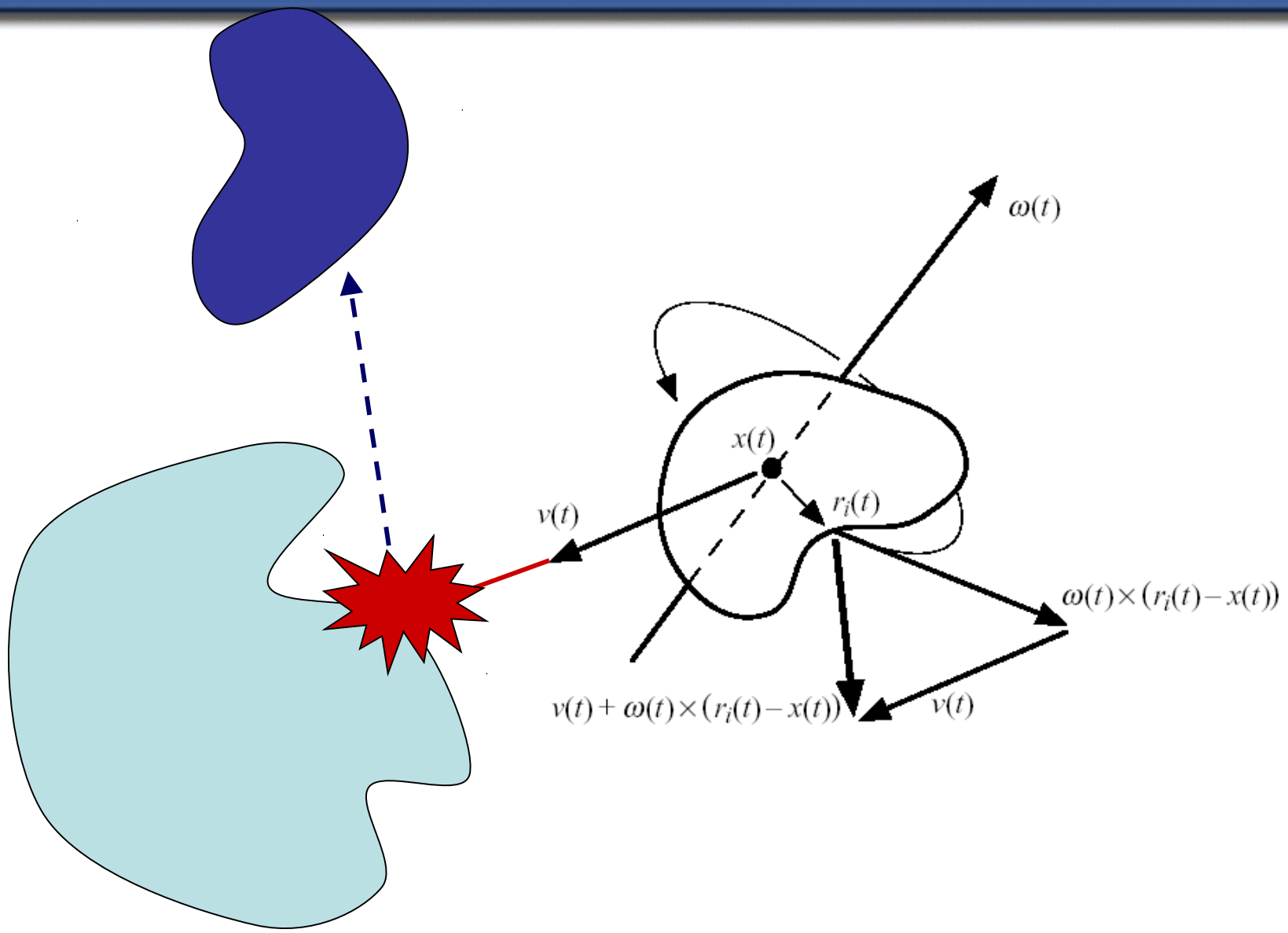
contact



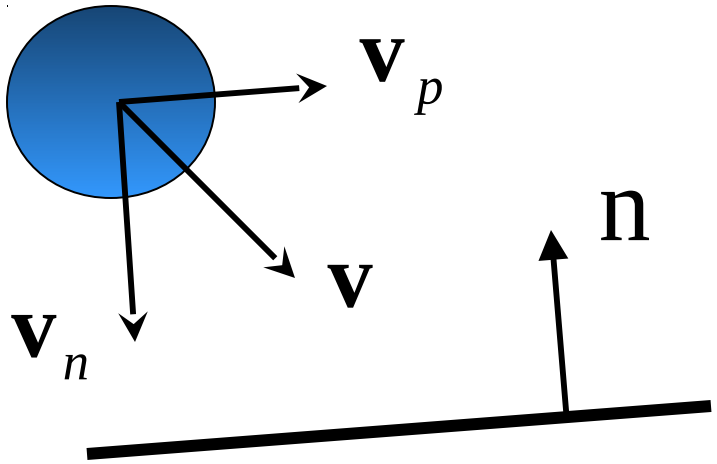
$\mathbf{N} \cdot (\dot{\mathbf{p}}_a(t) - \dot{\mathbf{p}}_b(t)) < 0$

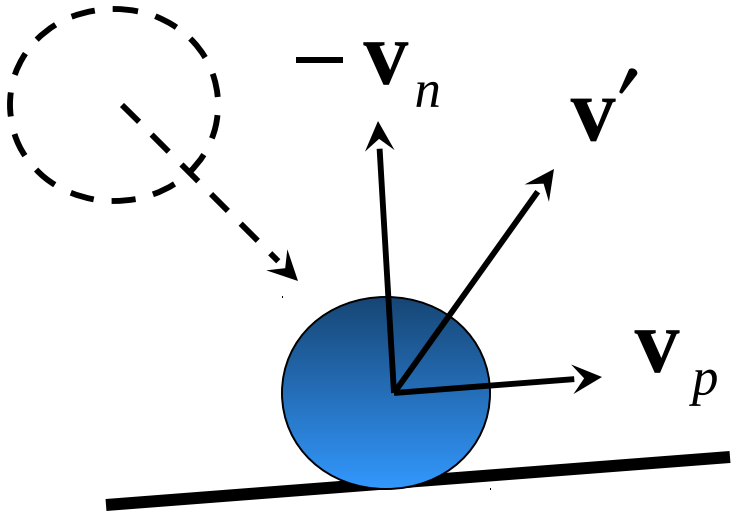
colliding

Collision Response



Collision Response → Sphere approaching a Plane





$$\mathbf{v} = \mathbf{v}_n + \mathbf{v}_p$$

$$\mathbf{v}_n = (\mathbf{v}.\mathrm{n})\mathrm{n}$$

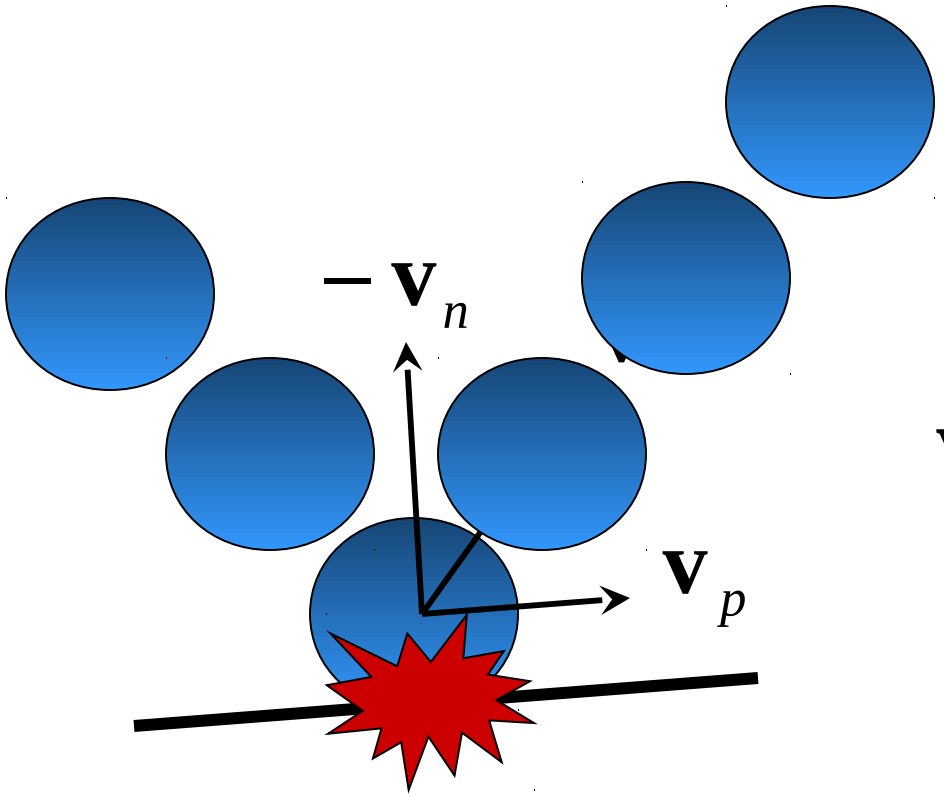
$$\mathbf{v}_p = \mathbf{v} - \mathbf{v}_n$$

$$\mathbf{v}' = \mathbf{v}_p - \mathbf{v}_n$$

What kind of response?

Totally elastic!
No kinetic energy is lost → response is “perfectly bouncy”

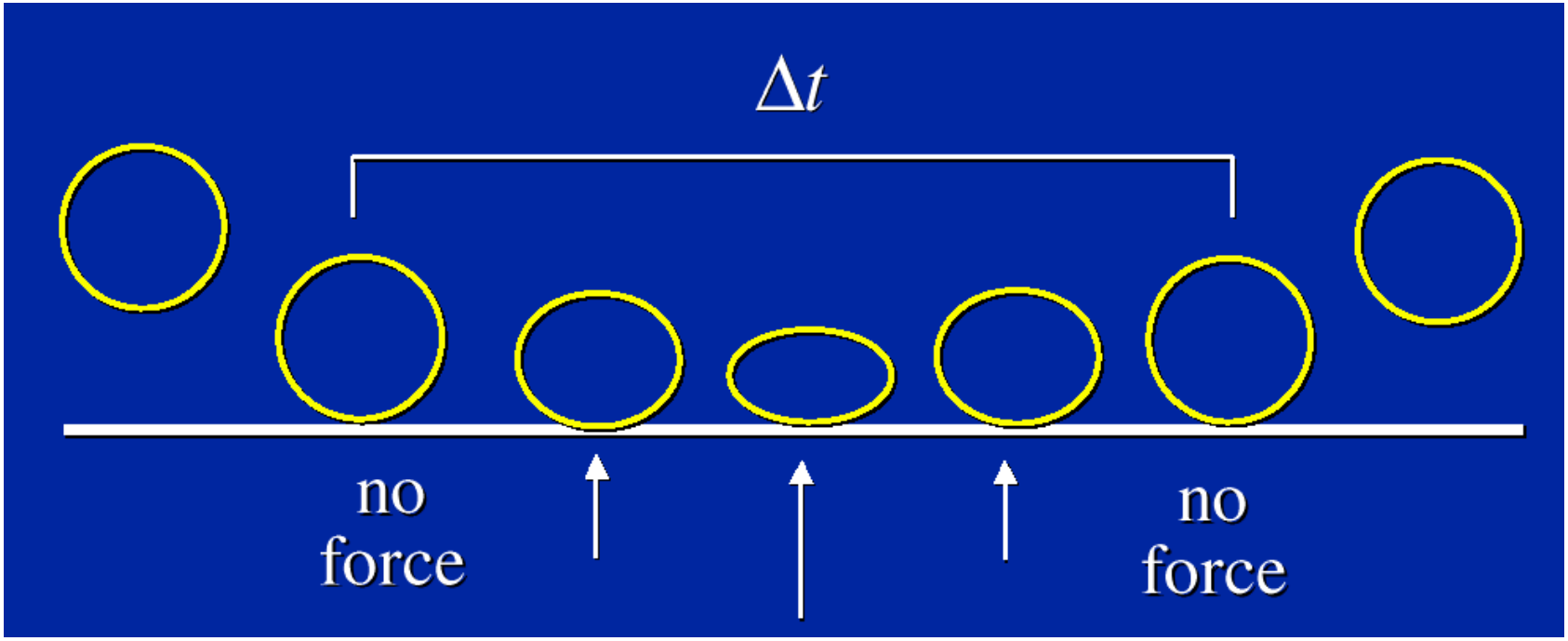
“Perfectly Bouncy” Response



$$\mathbf{v}' = \mathbf{v}_p - \mathbf{v}_n$$

Collision

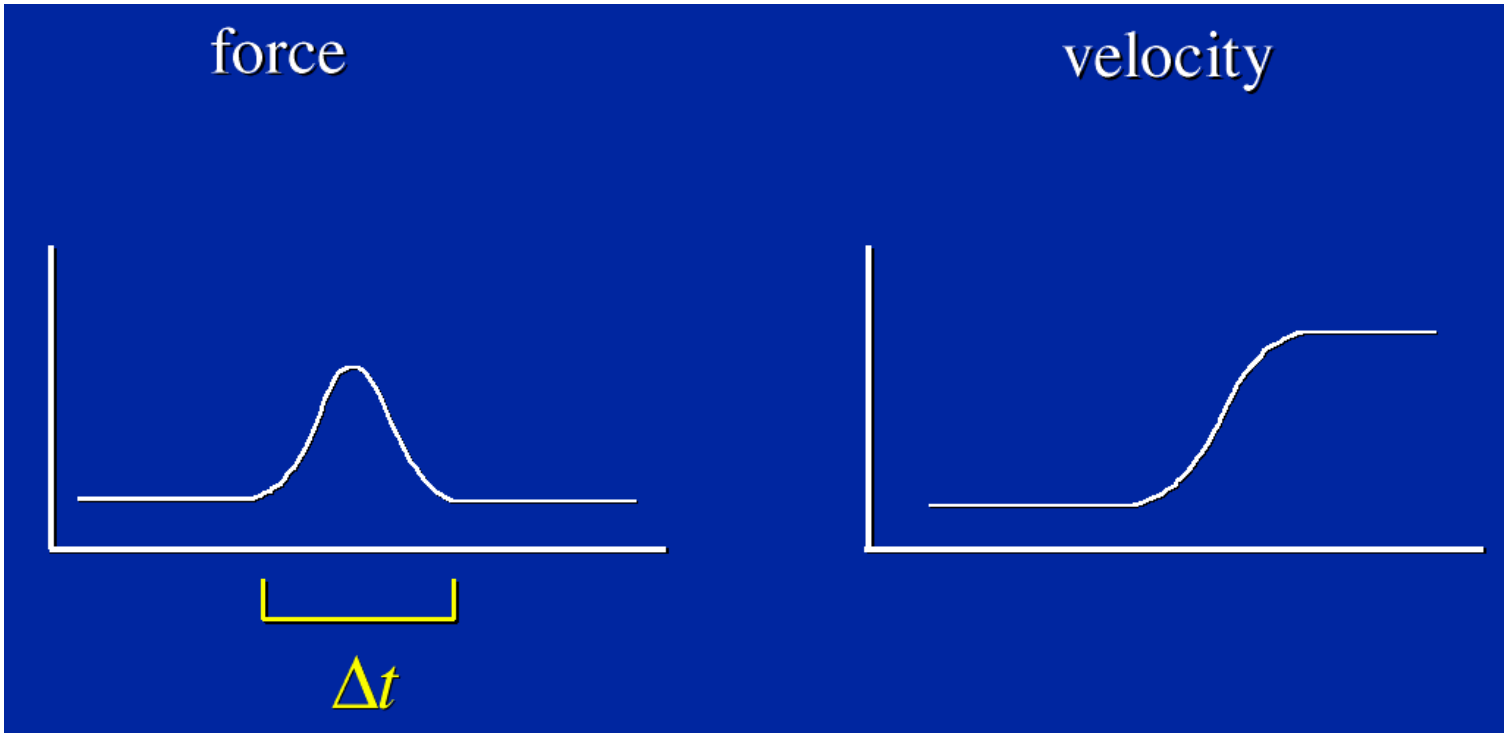
Soft Body Collision



Force is applied to prevent interpenetration

Collision

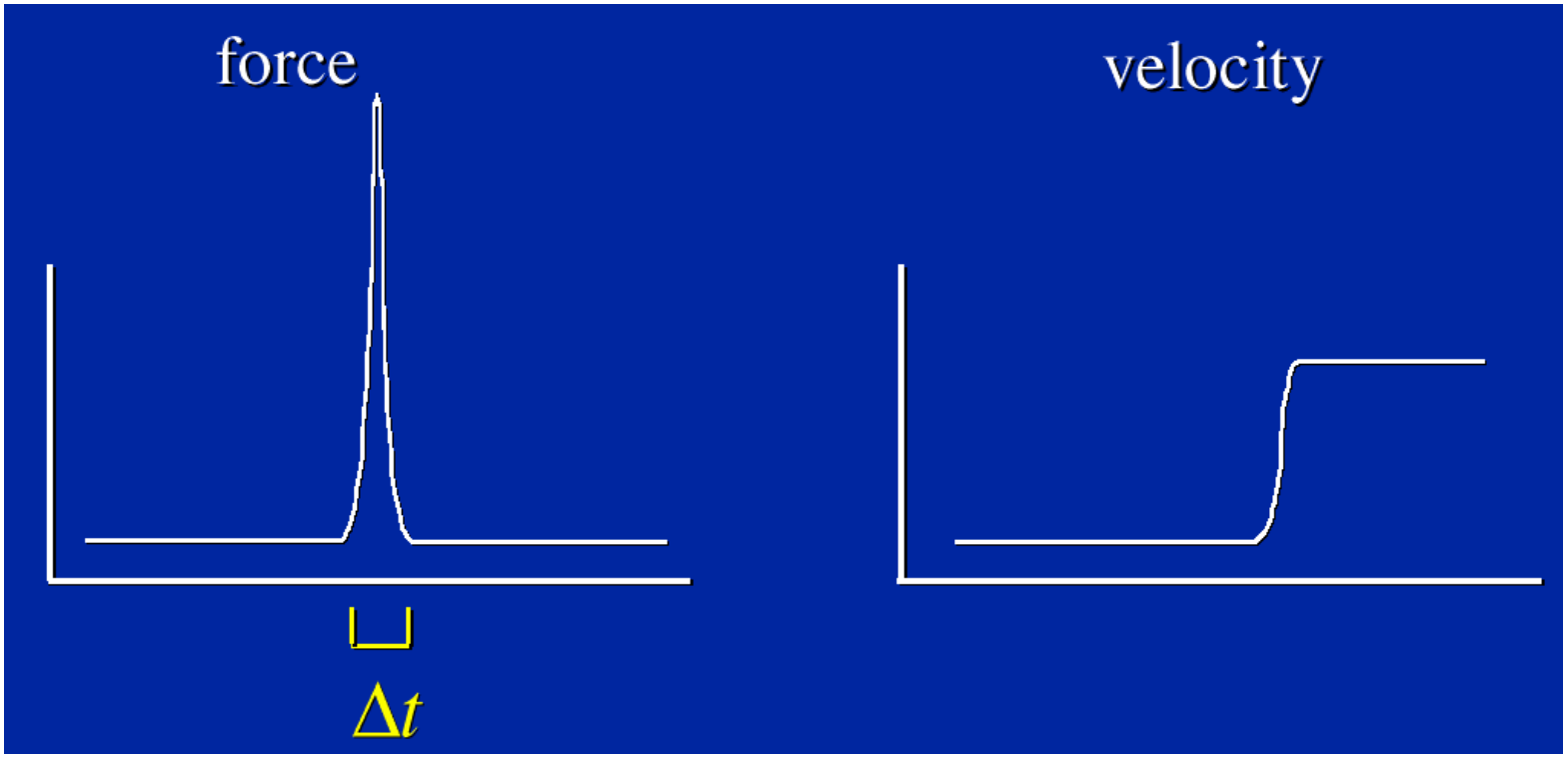
Soft Body Collision



Apply forces and change the velocity

Collision

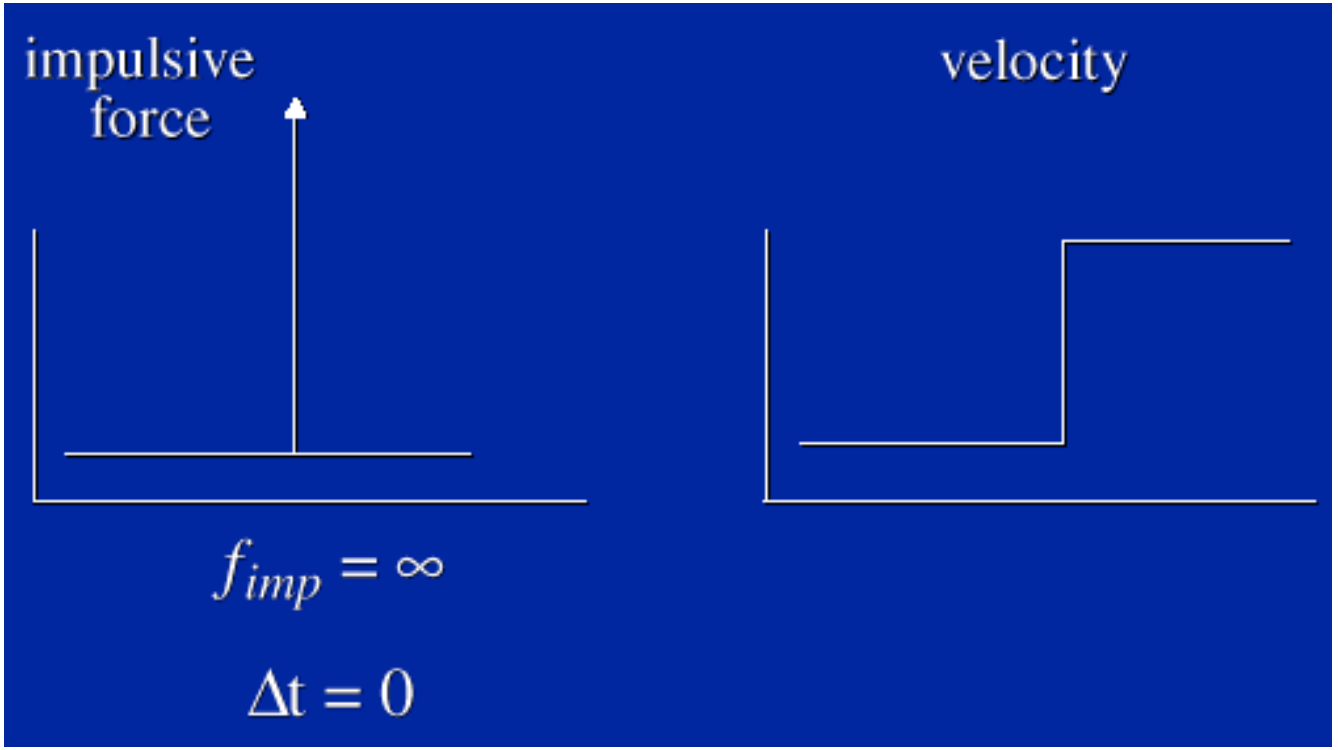
Harder Collision



Higher force over a shorter time

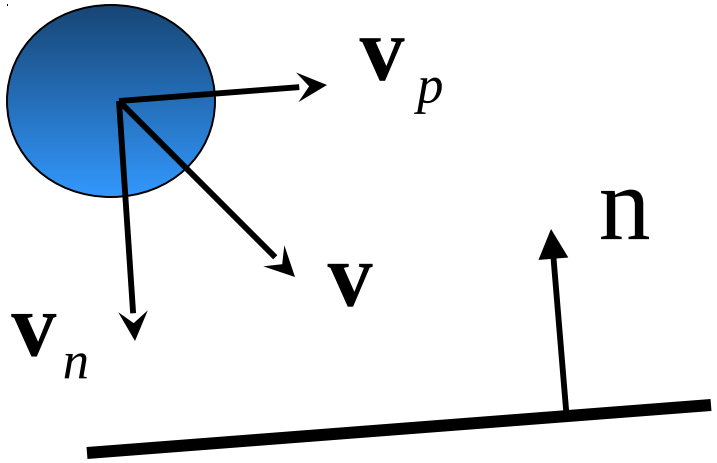
Collision

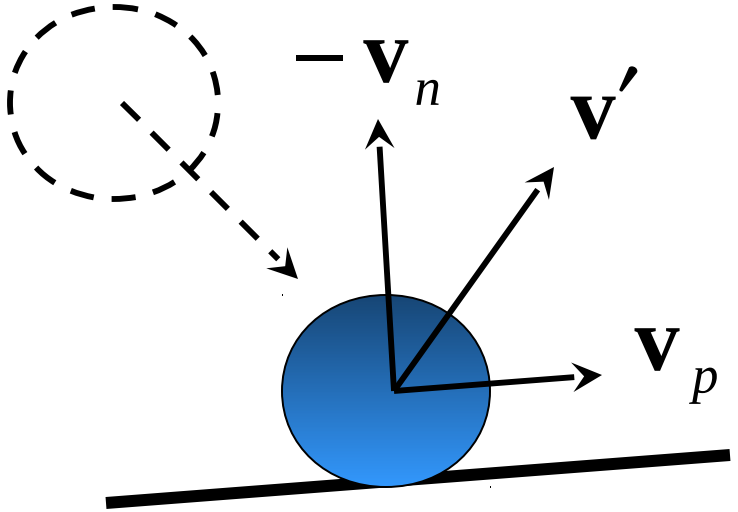
Rigid Body Collision



Impulsive force produces a discontinuous velocity

How to “Suggest” Sphere Deformation?





$$\mathbf{v} = \mathbf{v}_n + \mathbf{v}_p$$

$$\mathbf{v}_n = (\mathbf{v}.\mathrm{n})\mathrm{n}$$

$$\mathbf{v}_p = \mathbf{v} - \mathbf{v}_n$$

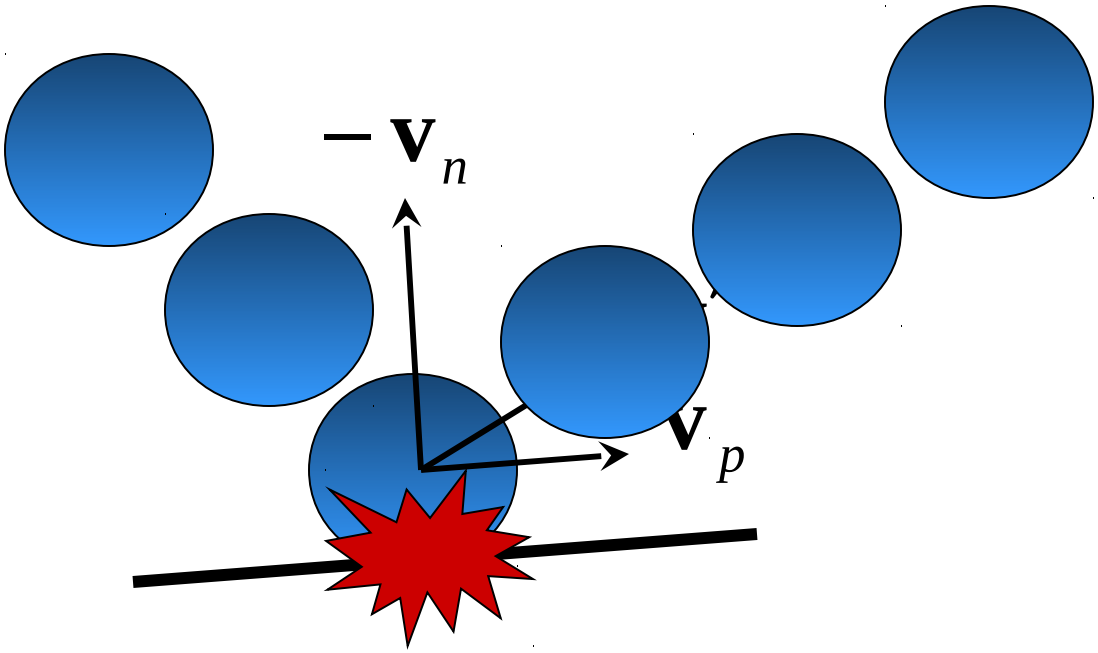
$$\mathbf{v}' = \mathbf{v}_p - k\mathbf{v}_n$$

k, in [0,1] → coefficient of *restitution*

Using Coefficient of Restitution:

As k gets smaller → more and more energy is lost → less and less bouncy

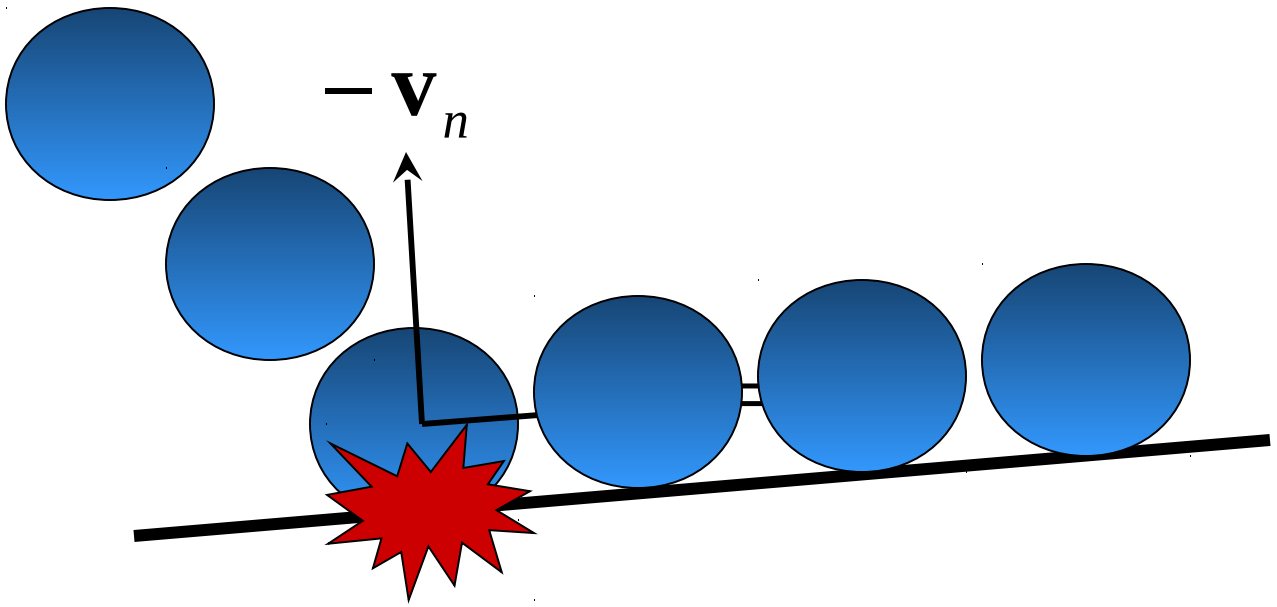
$$\mathbf{v}' = \mathbf{v}_p - 0.5\mathbf{v}_n$$



Using Coefficient of Restitution:

As k gets smaller → more and more energy is lost → less and less bouncy

$$\mathbf{v}' = \mathbf{v}_p - 0.0\mathbf{v}_n$$



Case Study 1: Player-Wall Collisions

- First person games must prevent the player from walking through walls and other obstacles
- In the most general case, the player is a polygonal mesh and the walls are polygonal meshes
- On each frame, the player moves along a path that is not known in advance
  - Assume it is piecewise linear – straight steps on each frame
  - Assume the player's motion could be fast

Stupid Algorithm

- On each step, do a general mesh-to-mesh intersection test to find out if the player intersects the wall
- If they do, refuse to allow the player to move
- What is poor about this approach? In what ways can we improve upon it?
  - In speed?
  - In accuracy?
  - In response?

Ways to Improve

- Even the seemingly simple problem of determining if the player hit the wall reveals a wealth of techniques
  - Collision proxies
  - Spatial data structures to localize
  - Finding precise collision times
  - Responding to collisions

Collision Proxies

- General mesh-mesh intersections are expensive
- Proxy: Something that takes the place of the real object
- A collision proxy is a piece of geometry used to represent a complex object for the purposes of finding a collision
- If the proxy collides, the object is said to collide
- Collision points are mapped back onto the original object
- What makes a good proxy?
- What types of geometry are regularly used as proxies?

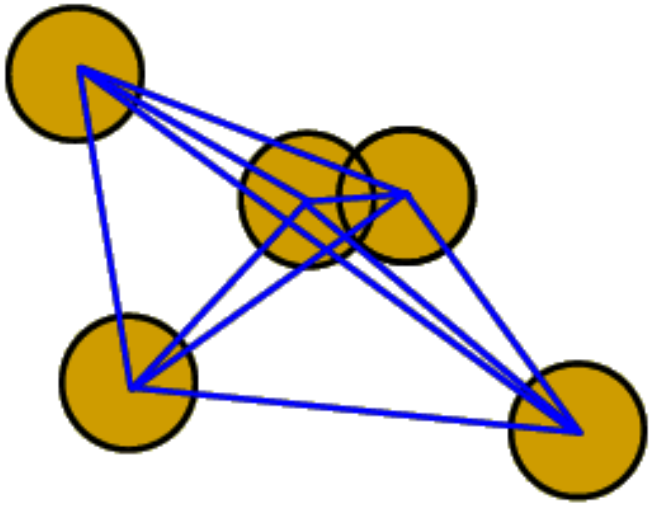
Proxy Properties

- A good proxy is cheap to compute collisions for and a tight fit to the real geometry
- Common proxies are spheres, cylinders or boxes of various forms, sometimes ellipsoids
  - What would we use for: A fat player, a thin player, a rocket, a car …
- Proxies exploit several facts about human perception:
  - We are extraordinarily bad at determining the correctness of a collision between two complex objects
  - The more stuff is happening, and the faster it happens, the more problems we have detecting errors
  - Players frequently cannot see themselves
  - We are bad a predicting what should happen in response to a collision

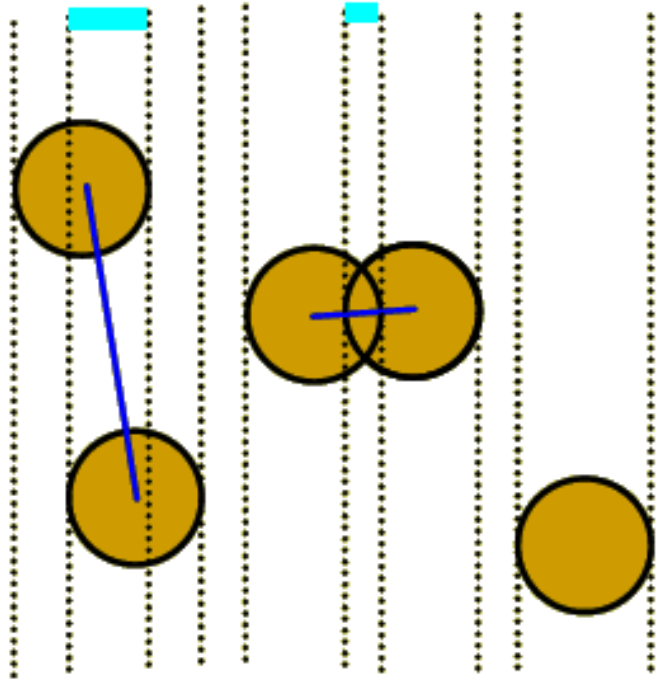
Spatial Data Structures

- You can only hit something that is close to you
- Spatial data structures can tell you what is close to an object
  - Fixed grid, Octrees, Kd-trees, BSP trees, …
- Recall in particular the algorithm for intersecting a sphere (or another primitive) with a BSP tree
  - Collision works just like view frustum culling, but now we are intersecting more general geometry with the data structure
- For the player-wall problem, typically you use the same spatial data structure that is used for rendering
  - BSP trees are the most common

Collision Detection:
Broad Phase vs Narrow Phase



BROAD PHASE:
Determine Which
Pairs to Check



NARROW PHASE :
Actually Calculate
Individual Pair Tests

Exploiting Coherence

- The player normally doesn’t move far between frames
- The cells they intersected the last time are probably the same cells they intersect now, or at least they are close
- The aim is to track which cells the player is in without doing a full search each time
- Easiest to exploit with a cell portal structure …

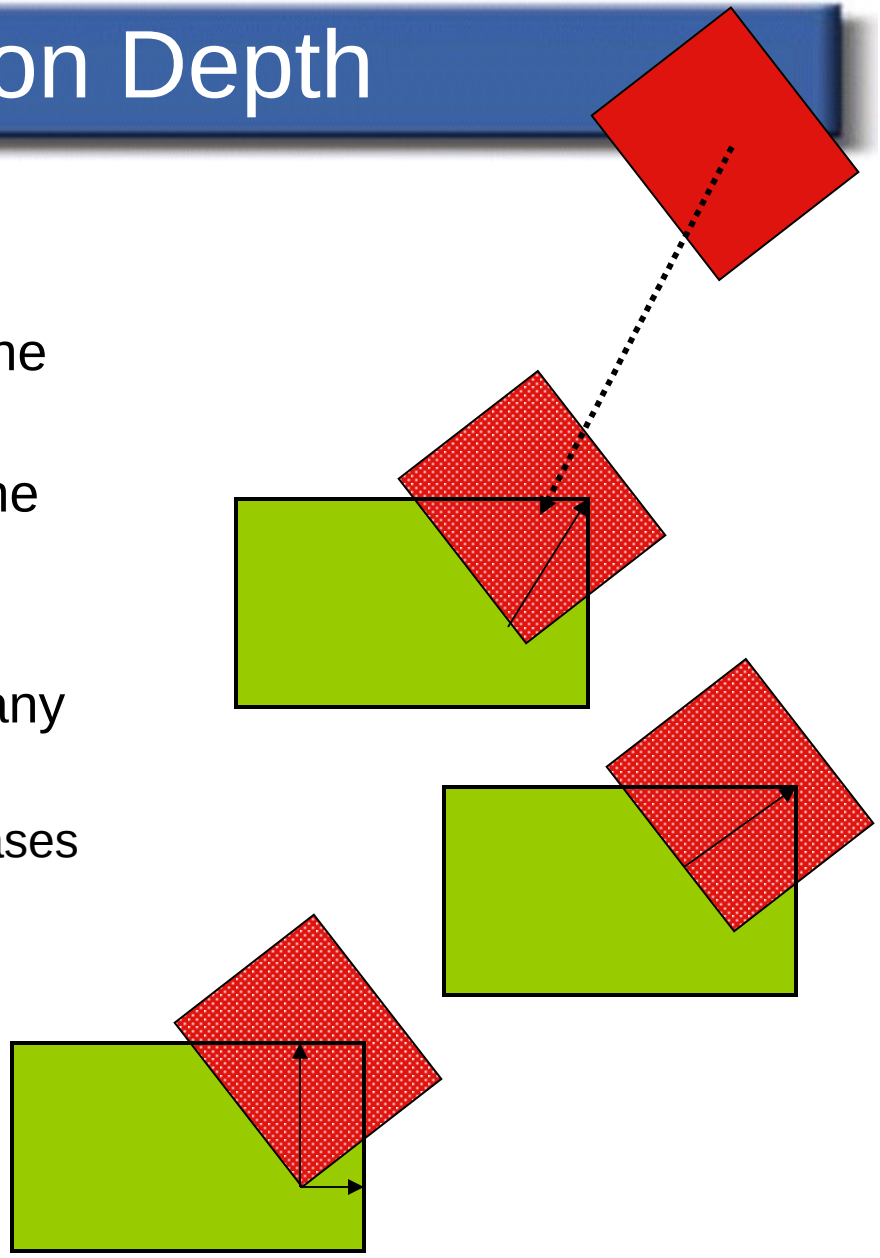
Cell-Portal Collisions

- Keep track which cell/s the player is currently intersecting
  - Can have more than one if the player straddles a cell boundary
  - Always use a proxy (bounding volume) for tracking cells
  - Also keep track of which portals the player is straddling
- The only way a player can enter a new cell is through a portal
- On each frame:
  - Intersect the player with the current cell walls and contents (because they're solid)
  - Intersect the player with the portals
  - If the player intersects a portal, check that they are considered "in" the neighbor cell
  - If the player no longer straddles a portal, they have just left a cell
- What are the implicit assumptions?

Defining Penetration Depth

- There is more than one way to define penetration depth

1. The distance to move back along the incoming path to avoid collision
   - But this may be difficult to compute
2. The minimum distance to move in any direction to avoid collision
   - Also difficult to compute in many cases
3. The distance in some particular direction
   - But what direction?
   - “Normal” to penetration surface

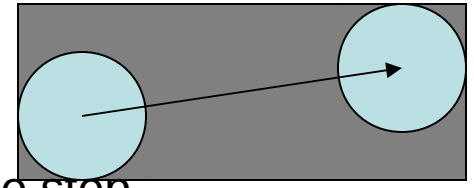


Managing Fast Moving Objects

- Several ways to do it, with increasing costs
- Test a line segment representing the motion of the center of the object
  - Pros: Works for large obstacles, cheap
  - Cons: May still miss collisions. How?
- Conservative prediction: Only move objects as far as you can be sure to catch the collision
  - Pros: Will find all collisions
  - Cons: May be expensive, and need a way to know what the maximum step is
- Space-time bounds: Bound the object in space and time, and check the bound
  - Pros: Will find all collisions
  - Cons: Expensive, and have to be able to bound motion

Prediction and Bounds

- Conservative motion:
  - Assume a maximum velocity and a smallest feature size
  - Largest conservative step is the smallest distance divided by the highest speed - clearly could be very small
  - Other more complex metrics are possible
- Bounding motion:
  - Assume linear motion
  - Find the radius of a bounding sphere
  - Build a box that will contain that sphere for the frame step
  - Also works for ballistic and some other predictable motions



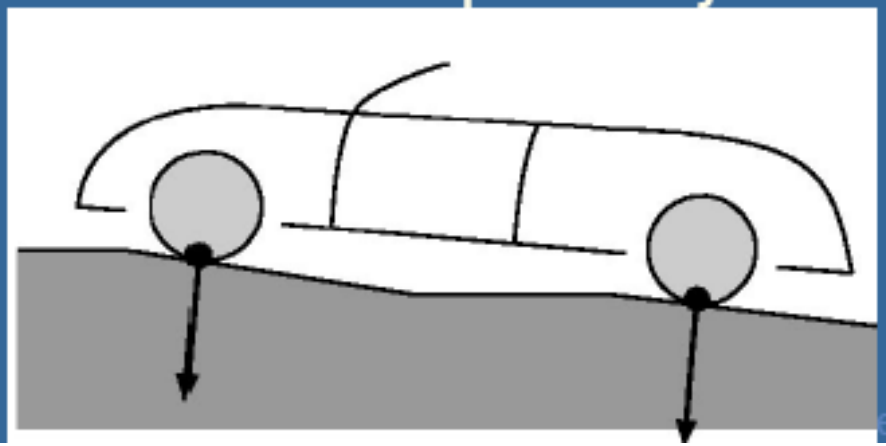
- Simple alternative: Just miss the hard cases - the player may not notice

Collision Response

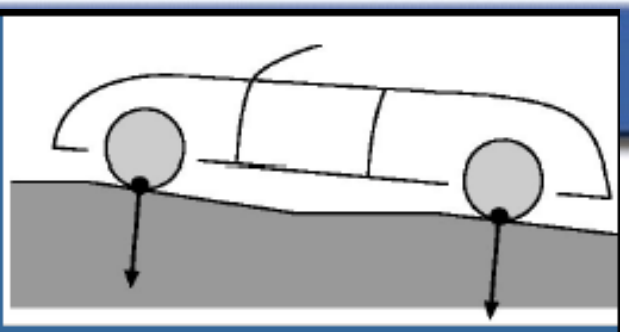
- For player motions, the best thing to do is generally move the player tangentially to the obstacle
- Have to do this recursively to make sure that all collisions are caught
  - Find time and place of collision
  - Adjust velocity of player
  - Repeat with new velocity, start time an start position (reduced time interval)
- Ways to handle multiple contacts at the same time
  - Find a direction that is tangential to all contacts
  - How do you do this for two planes?

Collision detection with rays

- Imagine a car is driving on a road sloping upwards
- Could test all triangles of all wheels against road geometry
- For certain applications, we can approximate, and still get a good result
- Idea: approximate a complex object with a set of rays



CD with rays, cont'd



- Put a ray at each wheel
- Compute the closest intersection distance, t, between ray and road geometry
- If t=0, then car is on the road
- If t>0, then car is flying above road
- If t<0, then car is ploughing deep in the road
- Use values of t to compute a simple collision response

Game Physics

Physics based game loop

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

$$\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt}$$

$$\boldsymbol{v} = \frac{d\boldsymbol{x}}{dt}$$

```
while true do
    update forces
    solve ODEs
    update objects
    render world
```

forces originate from user input, collisions, etc.

ODE solvers

A straightforward approach: The Euler method

$$F = ma$$

$$a = \frac{dv}{dt}$$

$$v = \frac{dx}{dt}$$

```
vNew = v + (F/m)*deltaT

xNew = x + (v+vNew)/2*deltaT
```

In general not accurate enough

Game Physics, contd

- Particle dynamics
- Rigid body dynamics
- Soft body dynamics
- (Fluid dynamics)

Particle systems

- Point masses - particles have mass but no extent
- But can be rendered as lines, polygons and objects
- Modelling of ”fuzzy objects”
  - Rain, snow, dust, wind
  - Fire, smoke, spray
  - Fountains, fireworks
  - Hair, fur

Particle parameters

Particle

- Position
- Velocity
- Geometry
- Mass
- Color
- Age
- Lifetime

Particle emitter

- Rate
- Position
- Movement
- Time, space and stochastic properties of the above

Particle system

- Forces
- Max no of particles

Exact collision detection algorithm

For two convex polyhedra Q and P:

```
for the plane defined by each face of Q:

    are all the vertices P on the outside of the
    plane defined by this face

for each face of P:

    are all the vertices Q on the outside of the
    plane defined by this face

for each edge of P each edge of Q:

    are all the vertices of P disjoint from all the
    vertices of Q with respect to the direction
    defined by the cross product of the vectors of
    the two edges.
```

Collision determination

For each colliding surface determine:

- Collision point
- Collision normal