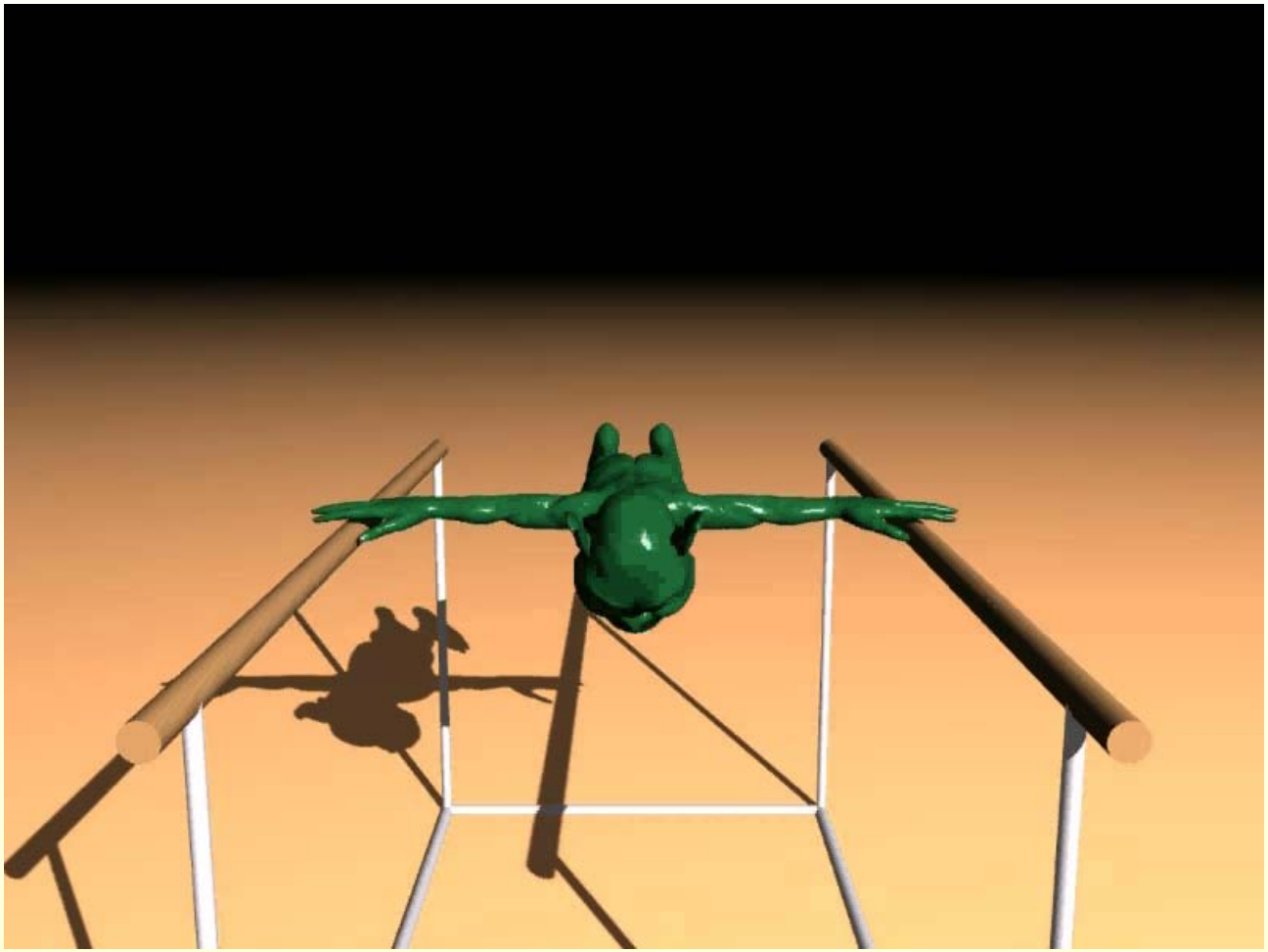
CS164: Moving Objects -- Collision Detection

Leonidas Guibas
Computer Science Dept.
Stanford University

Motion Simulation

- Motion simulation is essential to many parts of graphics and robotics:
  - special effects in movies
  - training simulators
  - interactive games
  - robot motion planning
- It has both continuous and discrete aspects
  - continuous: integration of the equations of motion ($F = ma$) – ODE solvers
  - discrete: collision detection and response

An Example



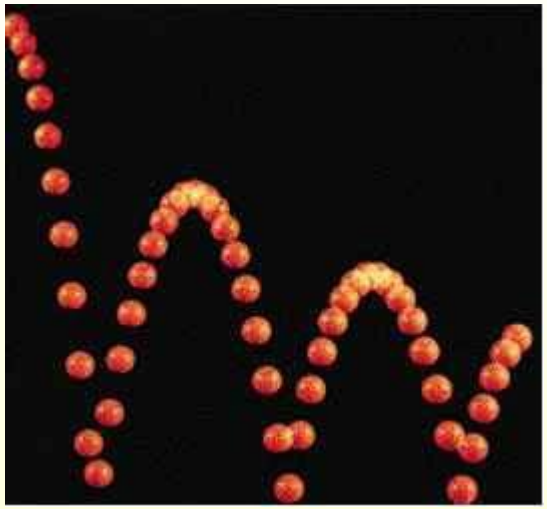
Collisions

- Collisions require detection and response
- Detection is a geometric problem
  - It boils down to repeated geometric intersection tests between virtual shapes
  - Thus it forms a good test-bed for different shape representations, and for coherence exploitation
- Response involves both geometry and physics – no time for it in this class

Collision Detection

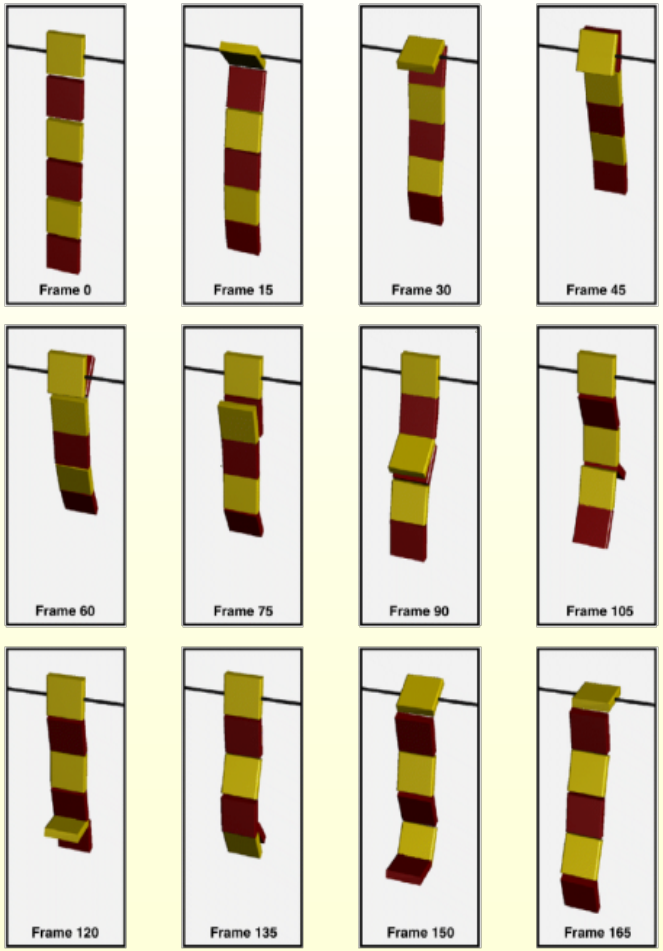
Collision Detection

- In a typical simulation or animation setting, the physics integrator advances the system state forward at certain time steps
- The motions or deformations generated by the integrator may cause collisions among the objects present in the scene
- These must be detected and avoided, or at least corrected after they happen



A Posterior vs. A Priori Collision Handling

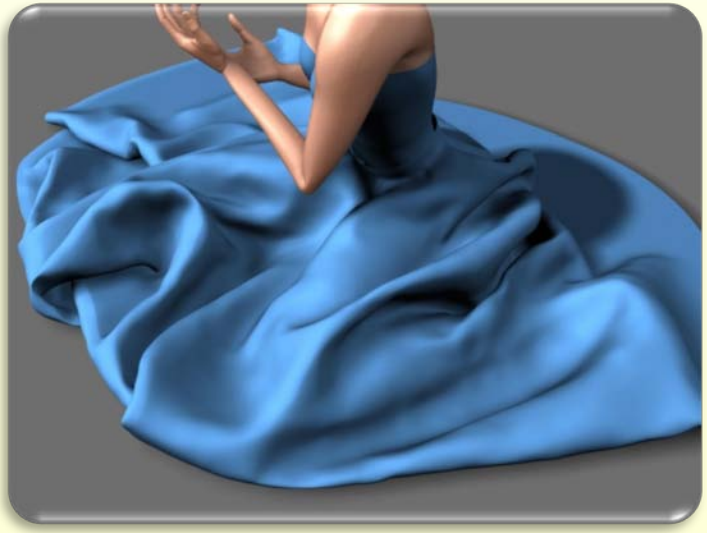
- A posteriori: The system is advanced by a small time step and a check made for all intersecting objects which are then “corrected” (updated position and/or trajectory). *A posteriori*, as instances of collisions may be“missed”.
- A priori: A collision detection algorithm is developed which can predict precise object trajectories. The exact instants of collision are calculated. *A priori*, as the instances of collision are calculated prior to collision.



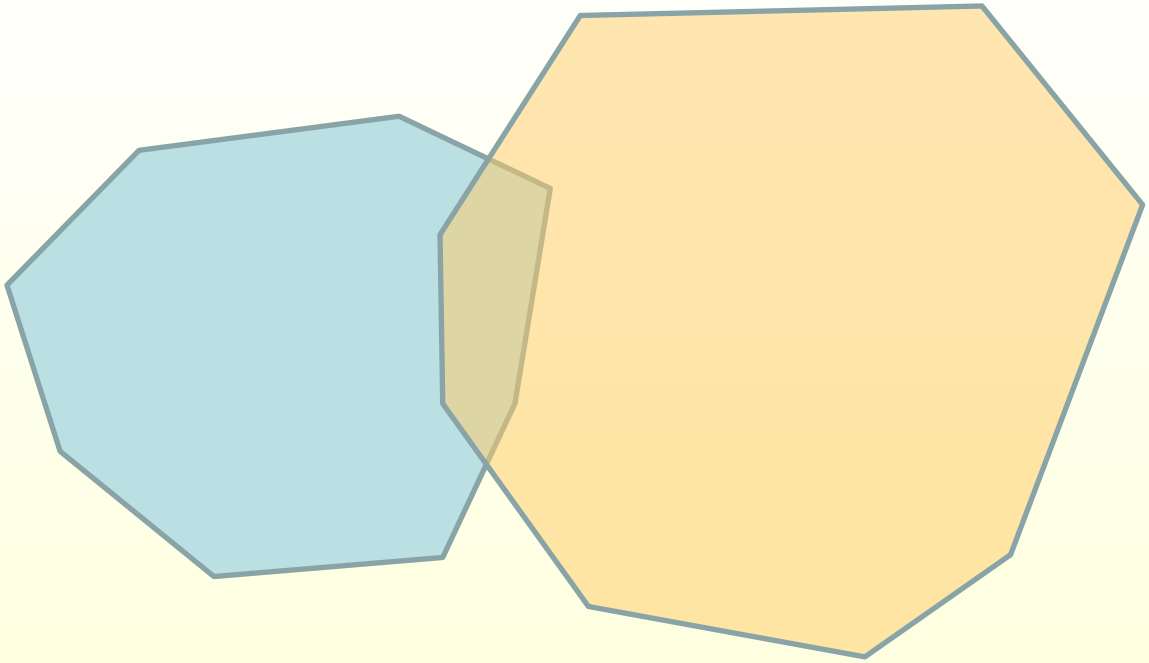
Trade-Offs

- Benefits of **a posteriori**: Potentially more simple collision detection (does not have to take into account friction, non-elastic collisions, deformable bodies, time, etc.)
- Downsides of **a posteriori**: the ability to accurately correct intersections (which are not physically correct)

- Benefits of **a priori**: increased fidelity and stability
- Downsides of **a priori**: highly complex algorithms (typically no closed form)



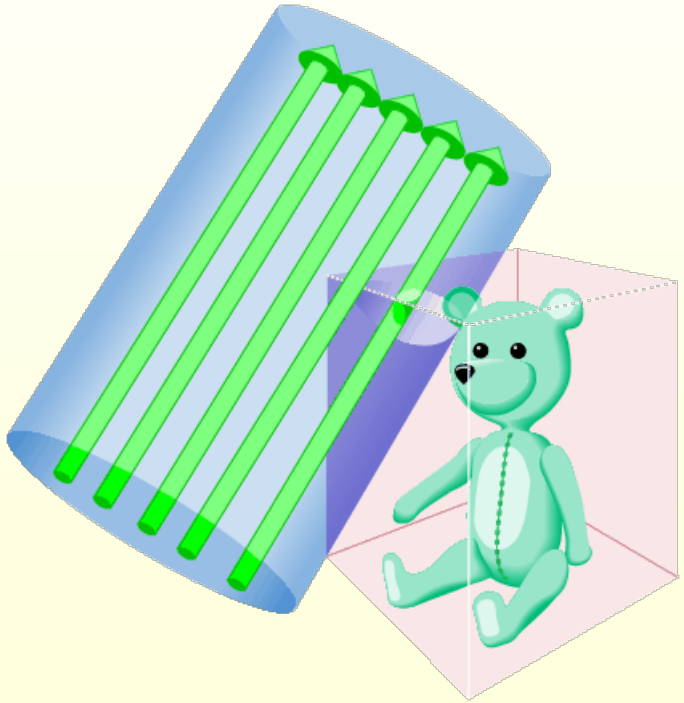
Collision Detection = Intersection Computation



A geometric intersection test

Collision Detection = Intersection Testing

- Given two objects, determine if the intersect (spatial collisions)
- Alternative: given two objects and two small-scale associated motions over an interval, determine in intersections occur during that interval (space-time collisions)
- Sometimes, we need to find all intersections. Other times, we just want the first one. Sometimes, we just need to know if the two objects intersect and don't need the actual intersection data.

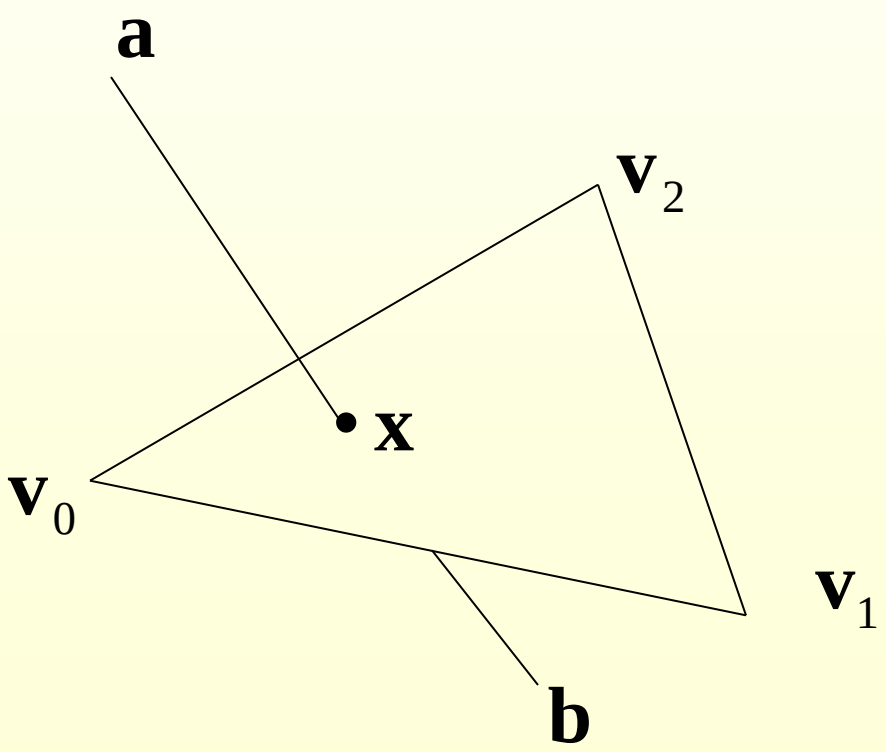


Primitives

- We often deal with various different ‘primitives’ that we describe our geometry with. Objects are constructed from these primitives
- For meshes, the standard primitive is a triangle. Boundaries of triangles, or line segments, are also treated as primitives.
- At the heart of the intersection testing are various primitive-primitive intersection tests

Segment vs. Triangle

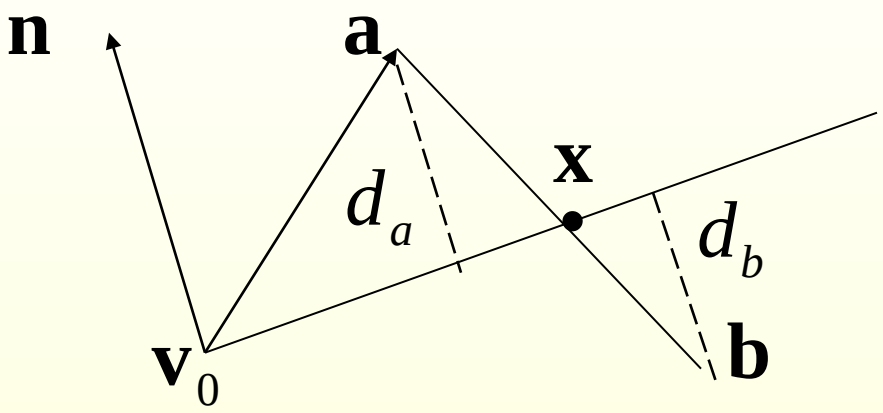
- Meshes are composed of triangles
- A basic test is “Does segment **ab** intersect triangle $\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$” ?



Segment vs. Triangle

- First, compute signed distances of a and b to triangle plane

$$d_a = (\mathbf{a} - \mathbf{v}_0) \cdot \mathbf{n}$$
$$d_b = (\mathbf{b} - \mathbf{v}_0) \cdot \mathbf{n}$$



- Reject if both are above or both are below triangle
- Otherwise, find intersection point **x**

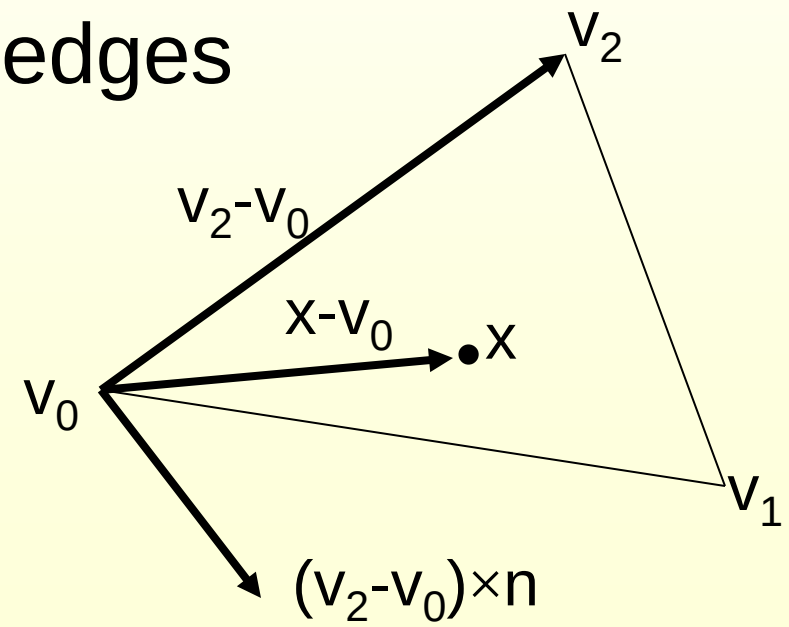
$$\mathbf{x} = \frac{d_a \mathbf{b} - d_b \mathbf{a}}{d_a - d_b}$$

Segment vs. Triangle

- Is point x inside the triangle?

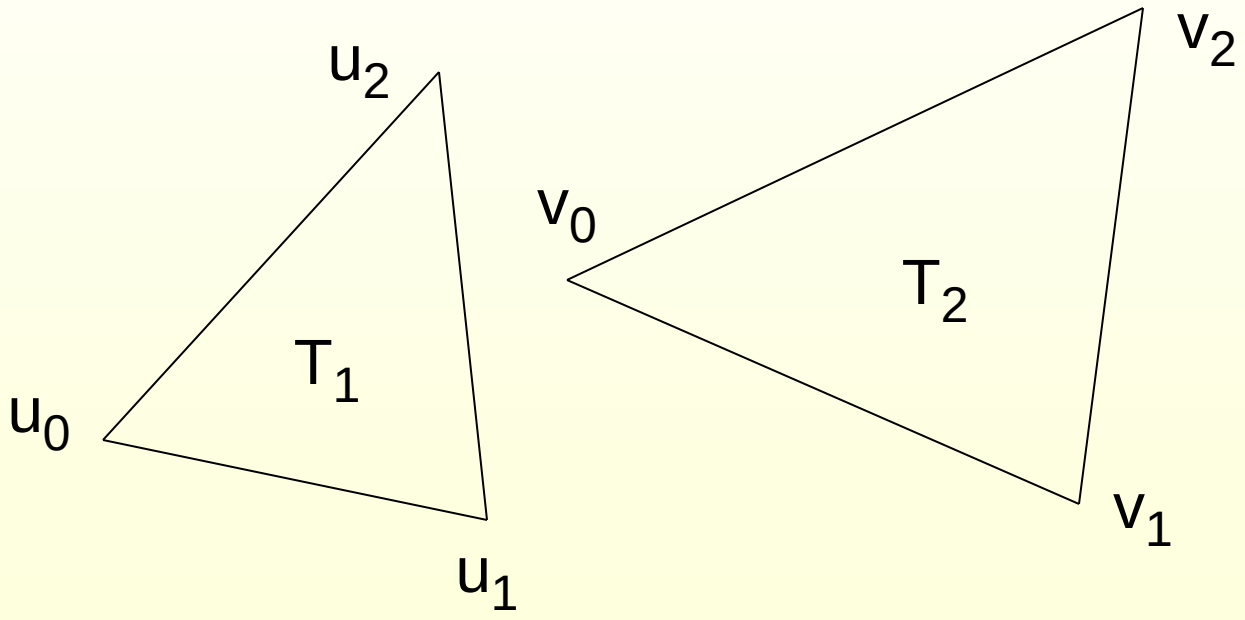
$$(x-v_0)\cdot((v_2-v_0)\times n) > 0$$

- Test all 3 edges



Triangle vs. Triangle

- Given two triangles: T_1 ($u_0u_1u_2$) and T_2 ($v_0v_1v_2$)

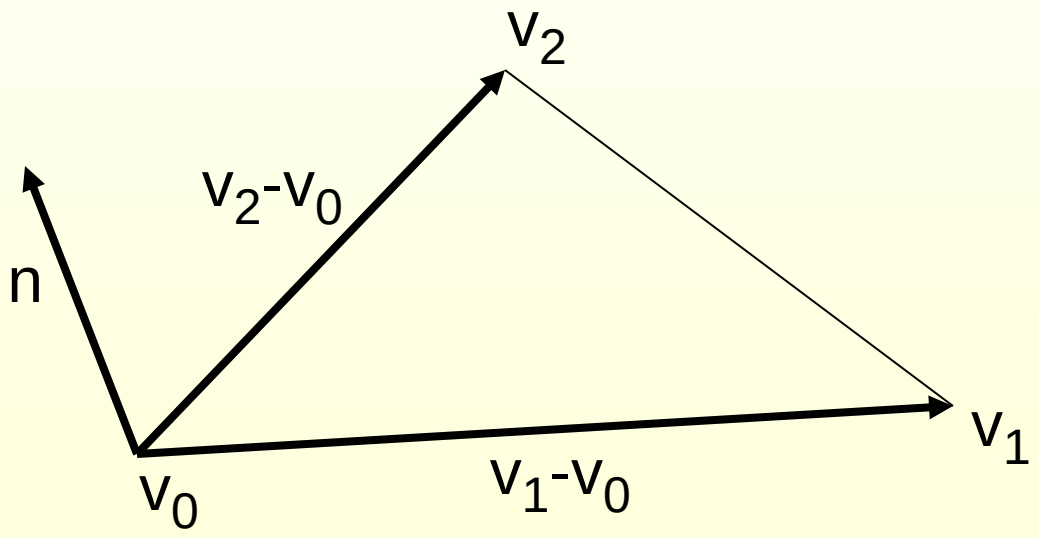


Triangle vs. Triangle

Step 1: Compute plane equations

$n_2=(v_1-v_0)\times(v_2-v_0)$

$d_2=-n_2\cdot v_0$

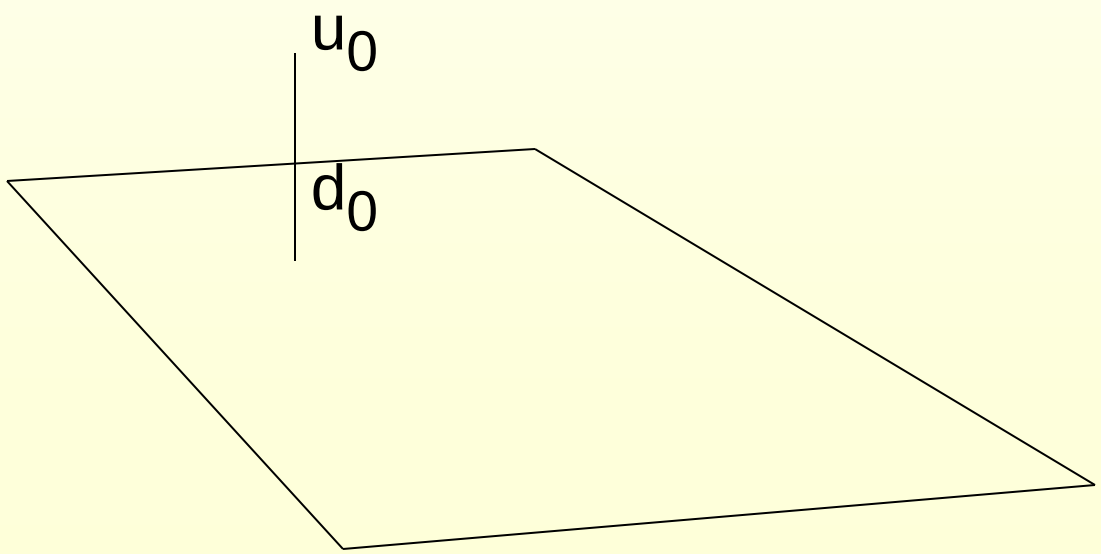


Triangle vs. Triangle

- Step 2: Compute signed distances of T_1 vertices to plane of T_2:

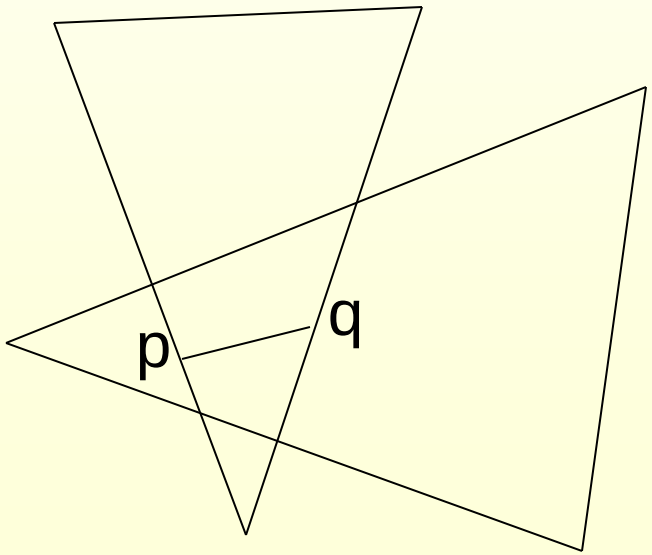
  $d_i = n_2 \cdot u_i + d_2 \qquad (i=0,1,2)$

- Reject if all $d_i<0$ or all $d_i>0$
- Repeat for vertices of T_2 against plane of T_1



Triangle vs. Triangle

- Step 3: Find intersection points
- Step 4: Determine if segment pq is inside triangle or intersects triangle edge

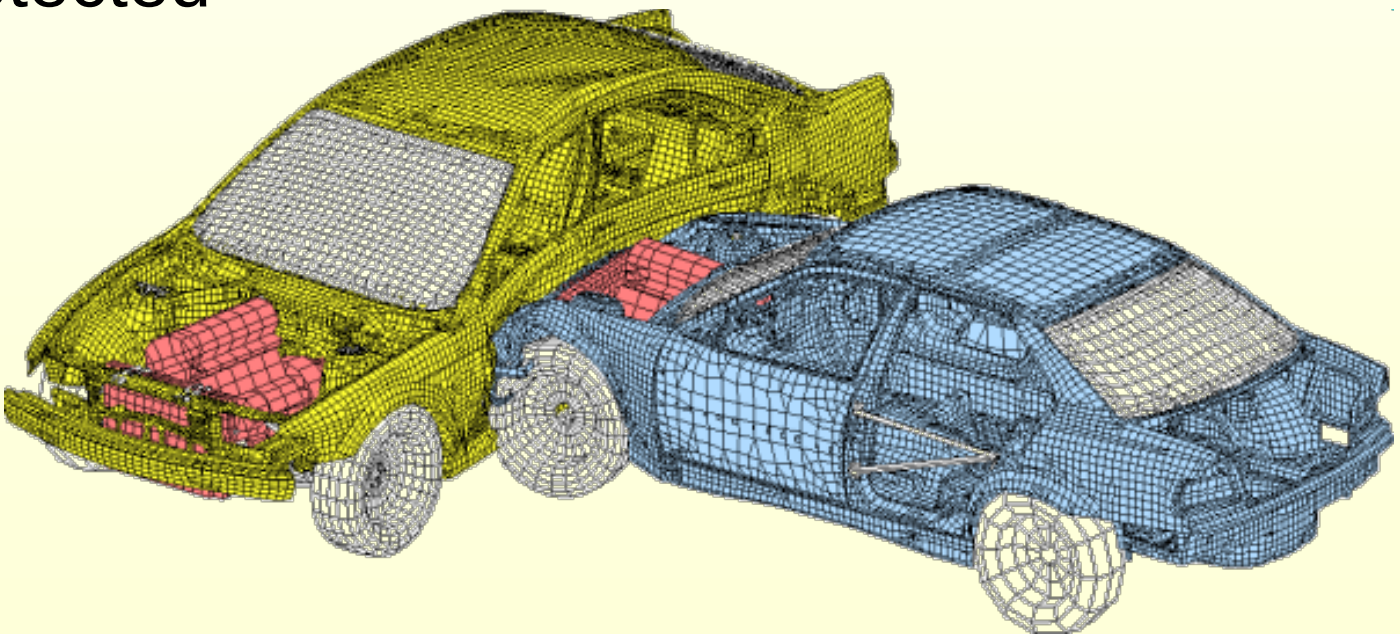


Intersection Issues

- Performance/speed
- Memory and precomputation
- Accuracy
- Floating point precision

Mesh vs. Mesh

- Geometry: points, edges, faces
- In principle, many collision types: p2p, p2e, p2f, e2e, e2f, f2f
- Multiple simultaneous collisions may be detected



Simple Geometry (Convexity)

+

Temporal Coherence

Exploiting Coherence

- Objects move only a little at each simulation time step
- Thus consecutive intersection problems are highly correlated
- This can be exploited by iterative collision detection algorithms which keep track of the closest pair of points between two objects

collision detection is closely related to distance computation

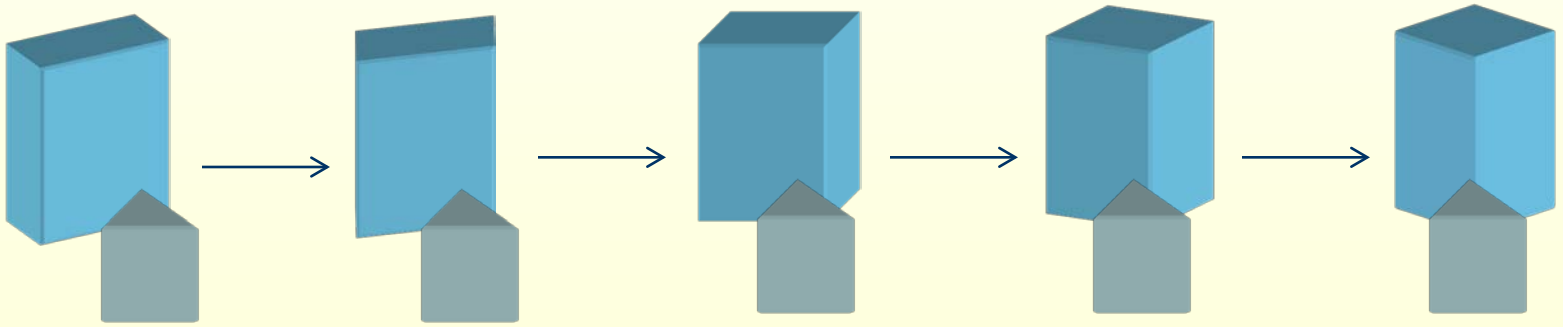
Lin-Canny algorithm for convex polyhedra…

an overview of the Lin-Canny algorithm

The Lin-Canny algorithm:

After some preprocessing (discussed later), the algorithm finds an initial pair of closest features between two polyhedra using an $O(n^2)$ search.

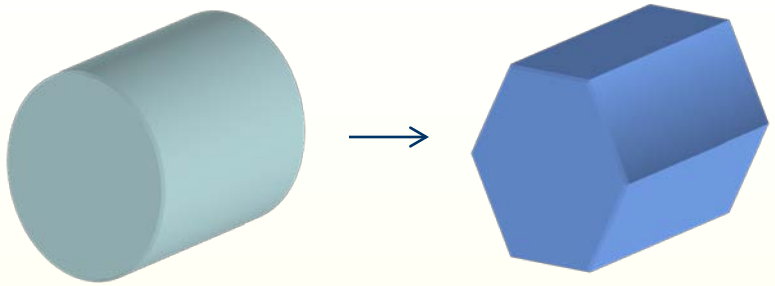
At each timestep, checks if the current closest feature pair is still the closest. If not, it finds the new closest pair.



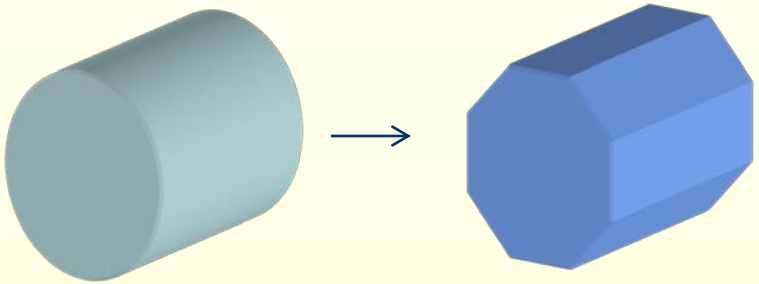
The Lin-Canny algorithm takes advantage of incremental motion, because closest features change infrequently.

implementating Lin-Canny in 4 easy steps…

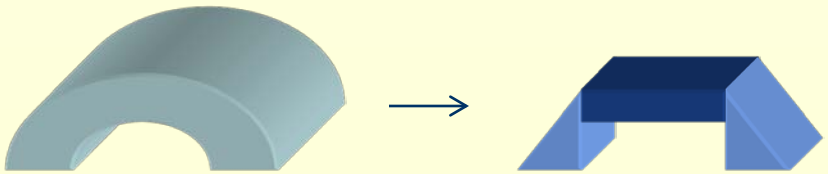
1. We first represent each object as a convex polyhedron, or a union of convex polyhedra.



We can improve the accuracy of the approximation by increasing the resolution of our representation.



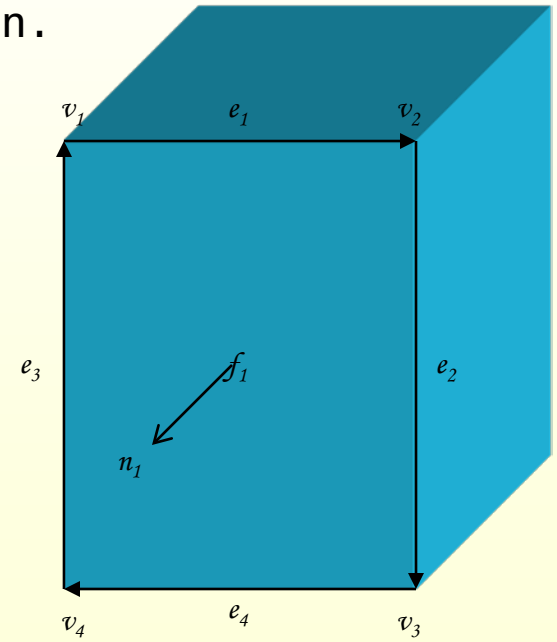
For non-convex objects we rely on subdivision into convex pieces (which can take up to quadratic time).



1. We first represent each object as a convex polyhedron, or a union of convex polyhedra.

2. **For each object we calculate the geometric parameters of each of its faces, edges, and vertices.**

A **FACE** f_i is parameterized by its outward normal and distance from the origin.

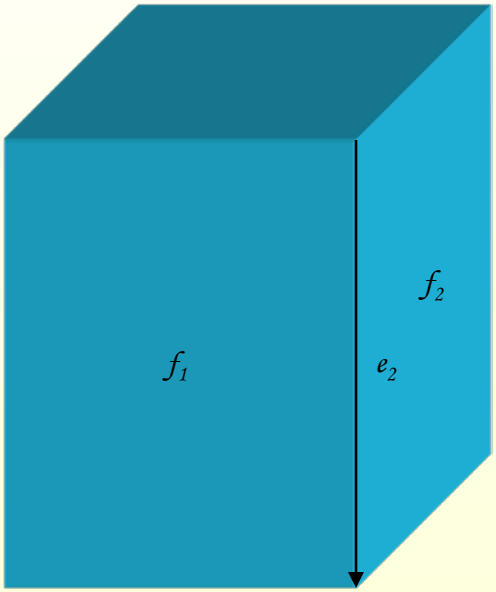


A **FACE** contains a list of **VERTICES** v_i which lie on its boundaries, and a list of **EDGES** e_i which bound the face.

1. We first represent each object as a convex polyhedron, or a union of convex polyhedra.

2. **For each object we calculate the geometric parameters of each of its faces, edges, and vertices.**

A **FACE** f_i is parameterized by its outward normal and distance from the origin.



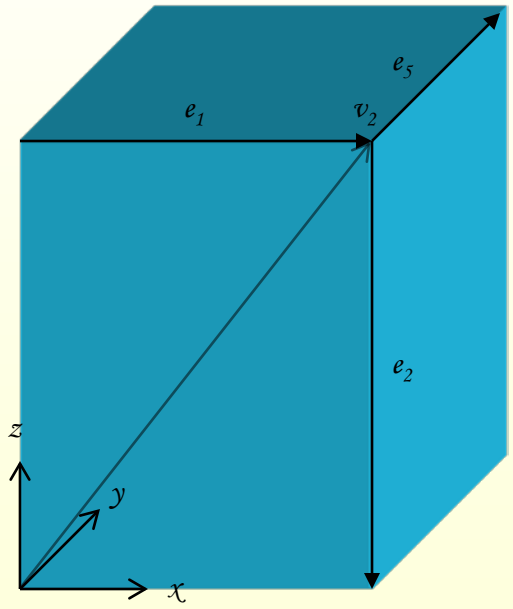
Each **EDGE** e_i is described by its head, tail, left face, and right face.

A **FACE** contains a list of **VERTICES** v_i which lie on its boundaries, and a list of **EDGES** e_i which bound the face.

1. We first represent each object as a convex polyhedron, or a union of convex polyhedra.

2. **For each object we calculate the geometric parameters of each of its faces, edges, and vertices.**

A **FACE** f_i is parameterized by its outward normal and distance from the origin.



Each **EDGE** e_i is described by its head, tail, left face, and right face.

A **FACE** contains a list of **VERTICES** v_i which lie on its boundaries, and a list of **EDGES** e_i which bound the face.

Each **VERTEX** v_i is described by its x,y,z coordinates and its **CO-BOUNDARY**, the set of edges that touch it.

1. We first represent each object as a convex polyhedron, or a union of convex polyhedra.

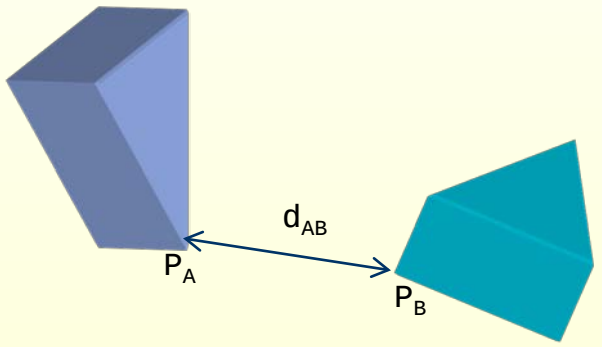
2. For each object we calculate the geometric parameters of each of its faces, edges, and vertices.

3. **For each pair of features between the two objects of interest, calculate the closest pair of points between those two features. Find the overall closest pair.**

Define P_A to be the closest point of $feature_A$ to $feature_B$

Define P_B to be the closest point of $feature_B$ to $feature_A$

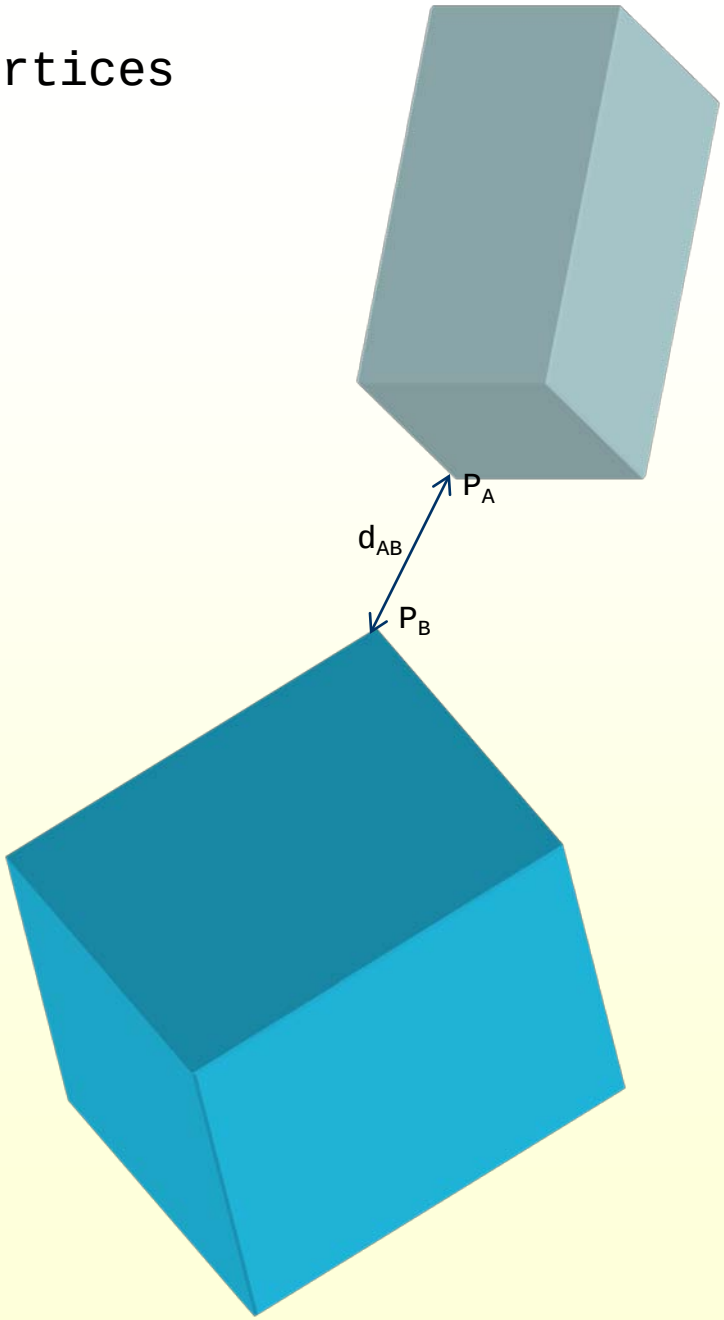
The distance d_{AB} is the Euclidean distance between P_A and P_B



There are six different possible feature pairs that we will need to analyze…

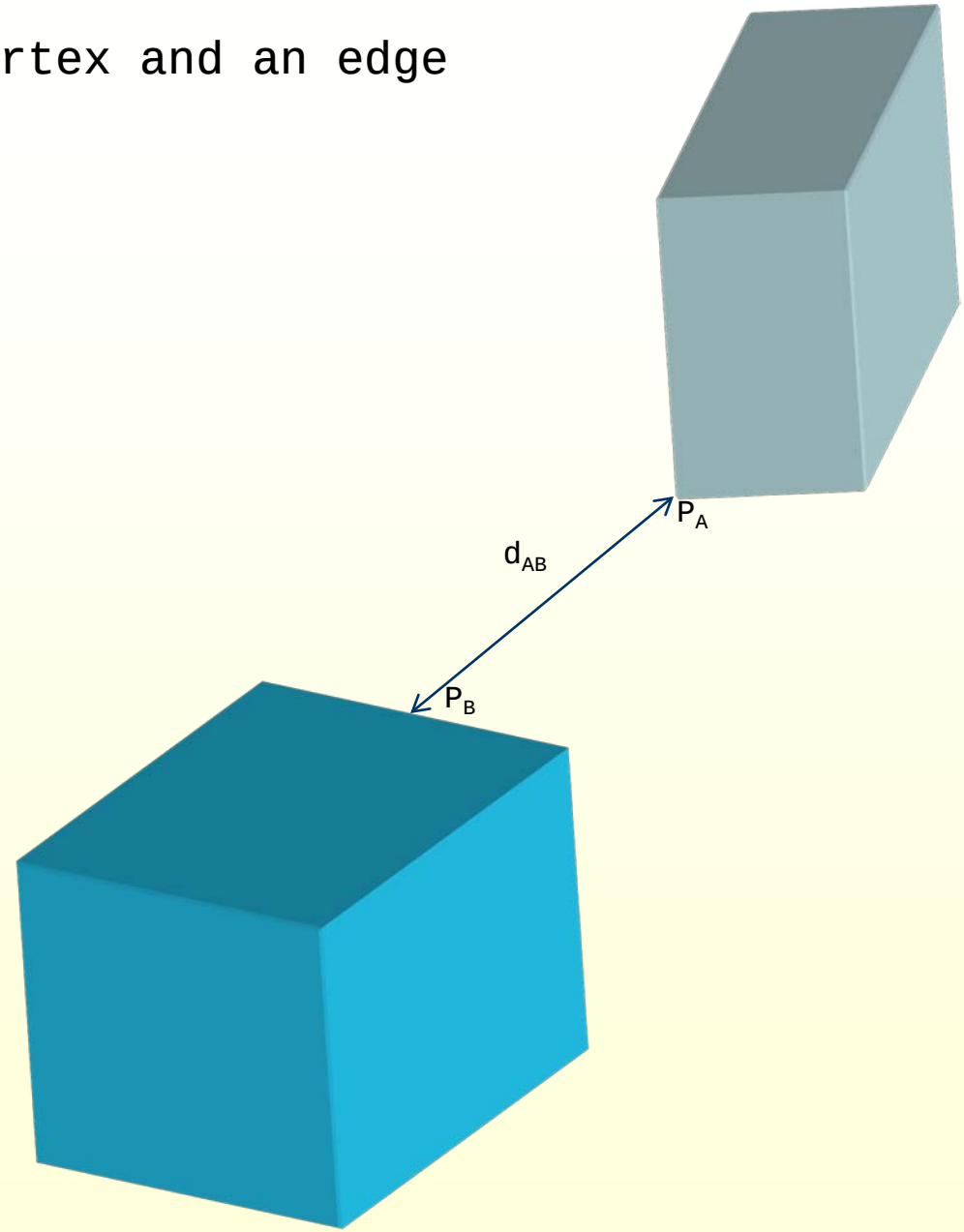
case one:

a pair of vertices



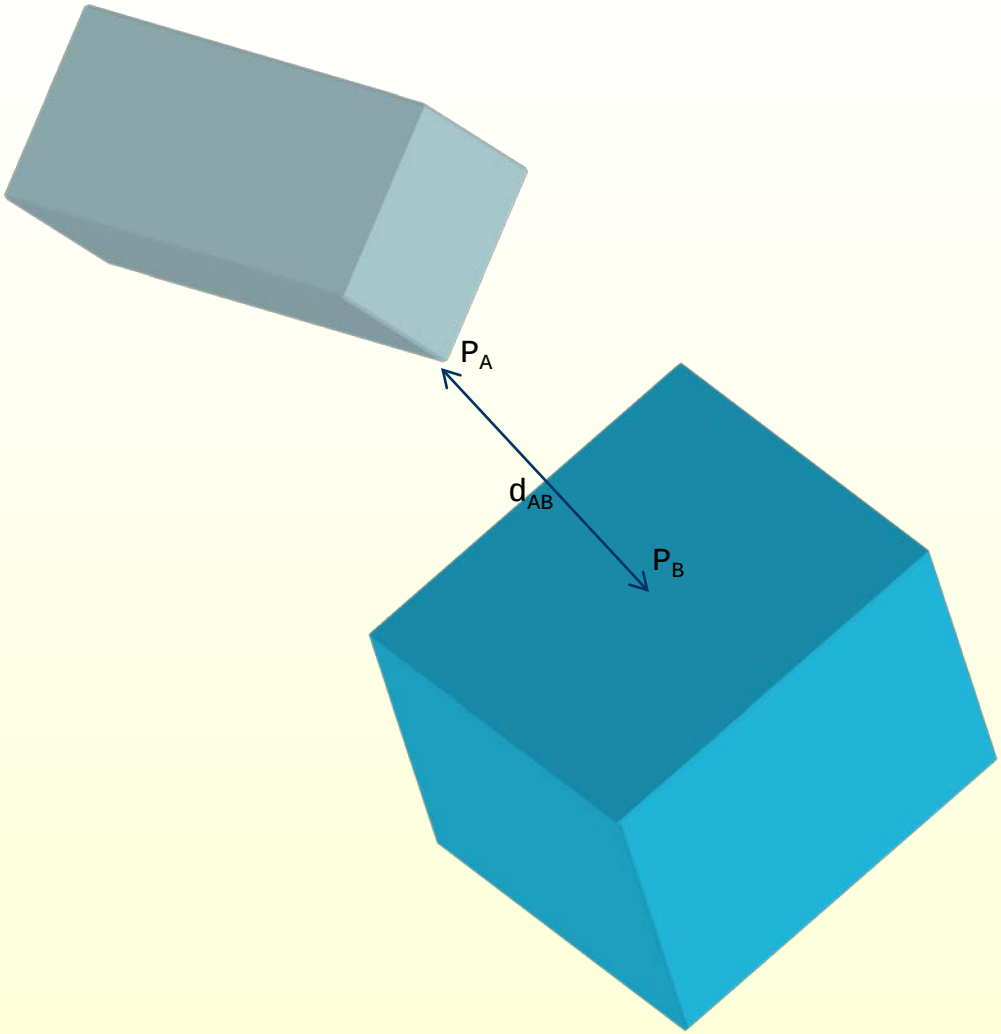
case two:

a vertex and an edge



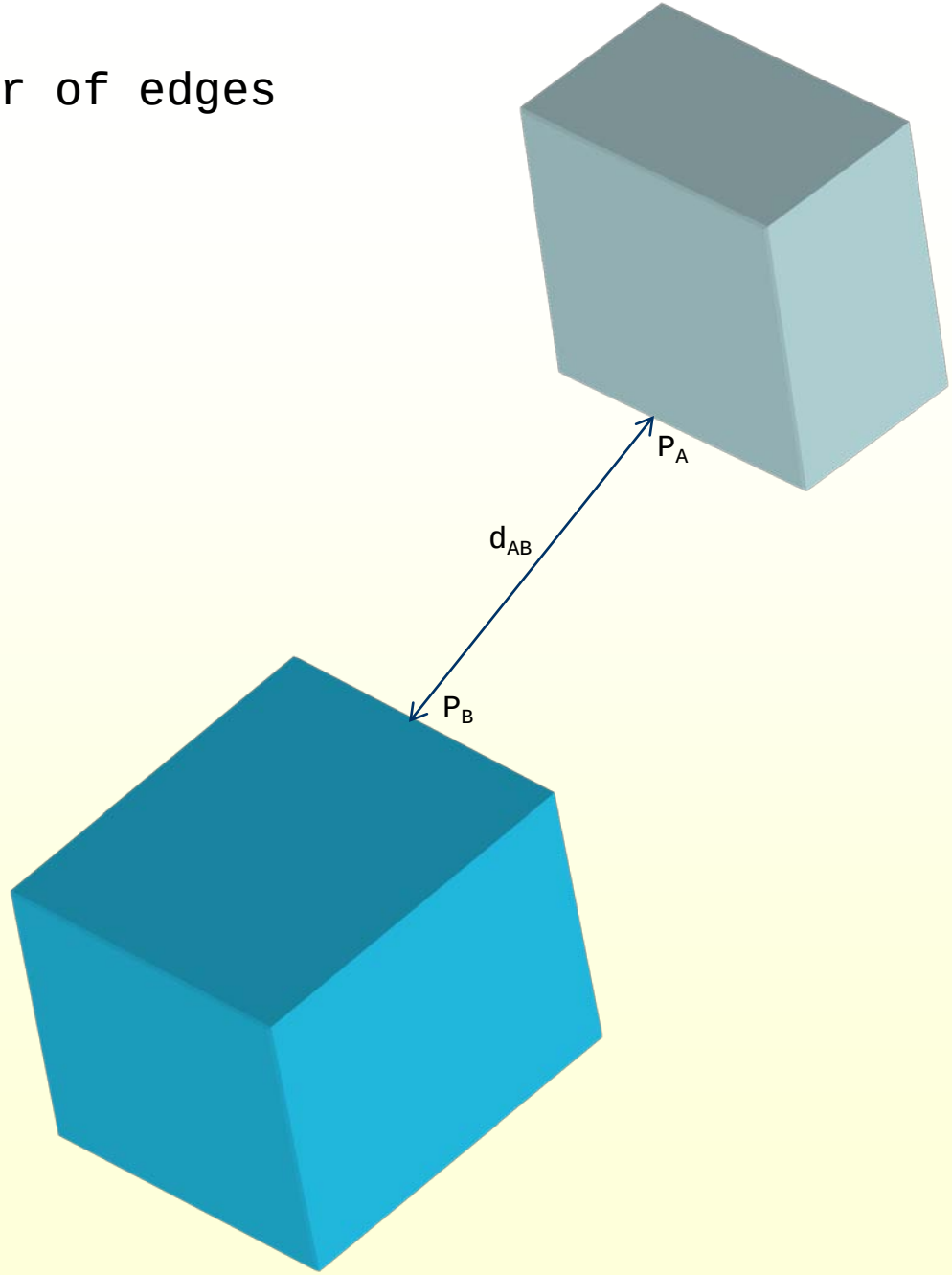
case three:

a vertex and a face



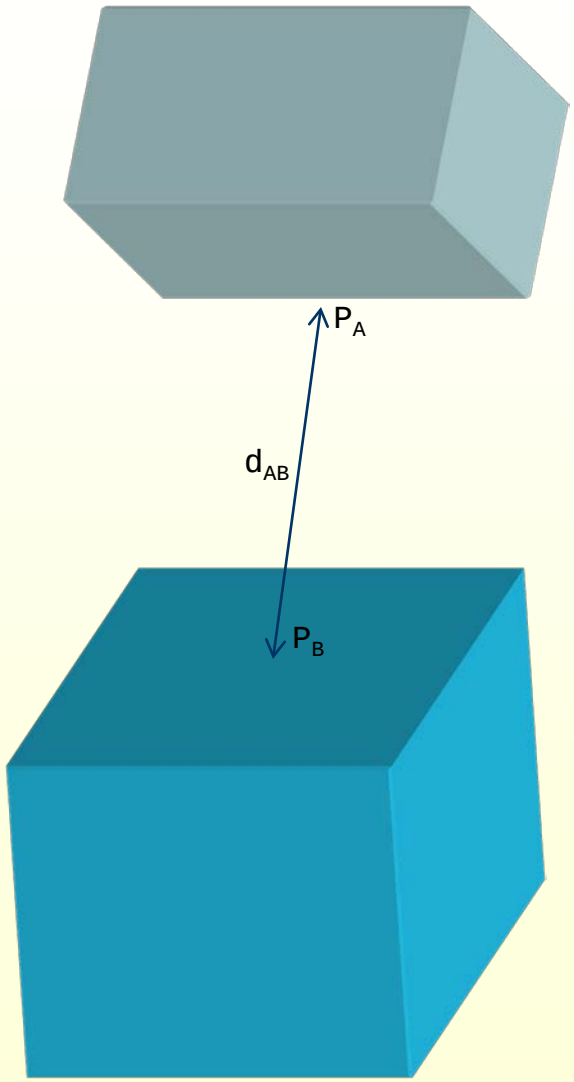
case four:

a pair of edges



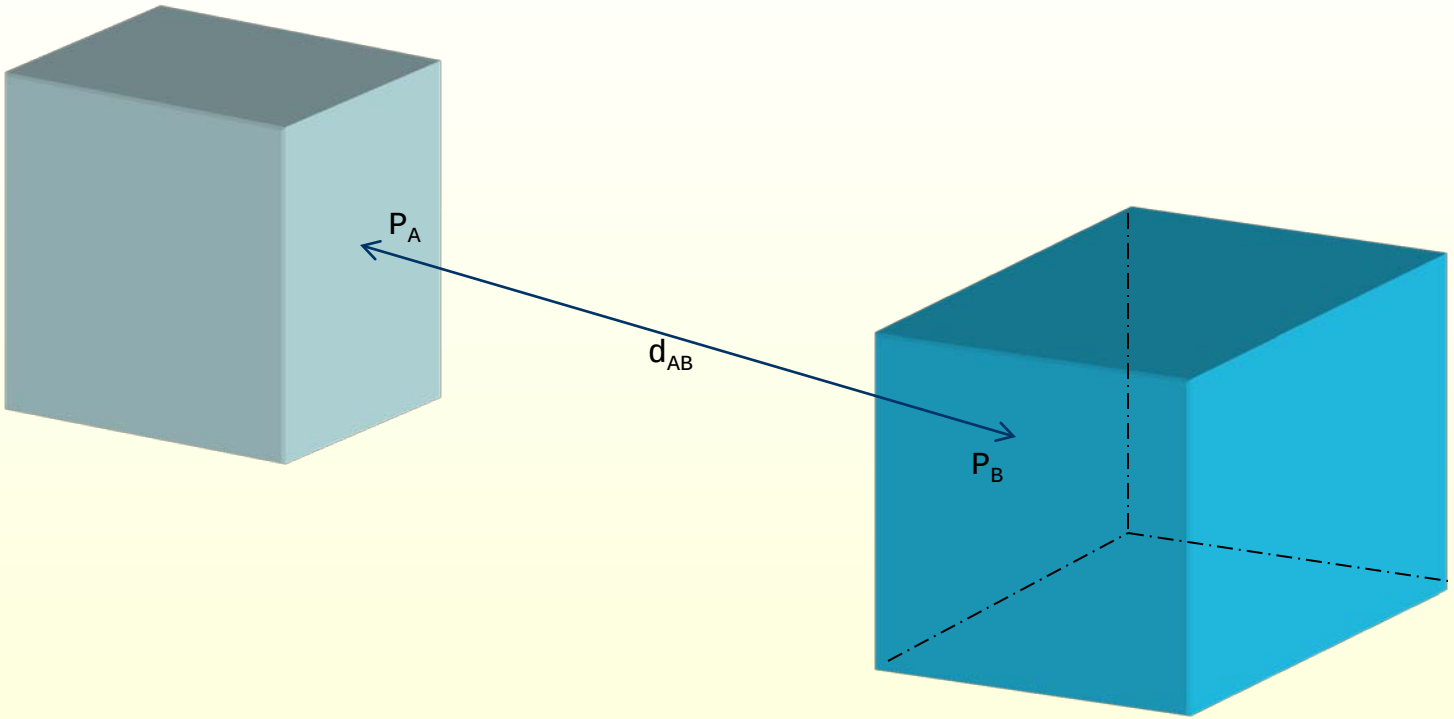
case five:

an edge and a face



case six:

a pair of faces (rare!)



1. We first represent each object as a convex polyhedron, or a union of convex polyhedra.

2. For each object we calculate the geometric parameters of each of its faces, edges, and vertices.

3. For each pair of features between the two objects of interest, calculate the closest pair of points between those two features. Find the overall closest pair.

4. **Incrementally update the closest feature pair.**

We utilize the following algorithm for updating:

1. Verify that P_A is the closest point of A to $feature_B$, and that P_B is the closest point of B to $feature_A$

2. If verification fails, choose a new feature pair and repeat step one

3. Eventually we will terminate on the closest feature pair

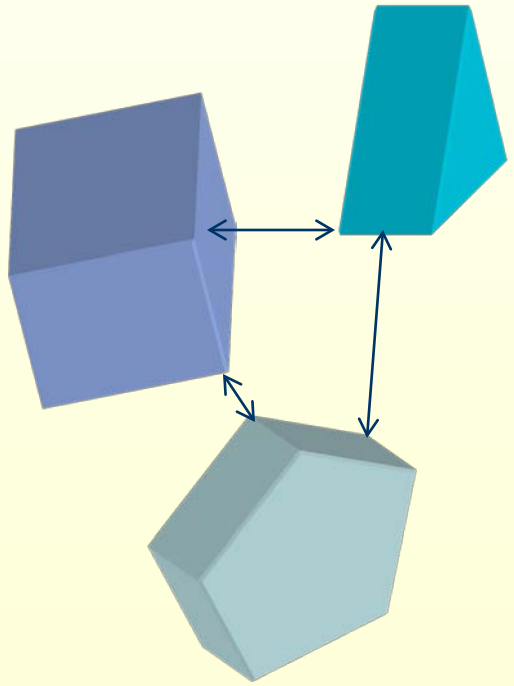
question: how can we tell if we've found a closest feature pair, or if we need to try a new pair?

answer: we have "applicability criteria" that each feature pair must satisfy in order to be the closest feature pair.

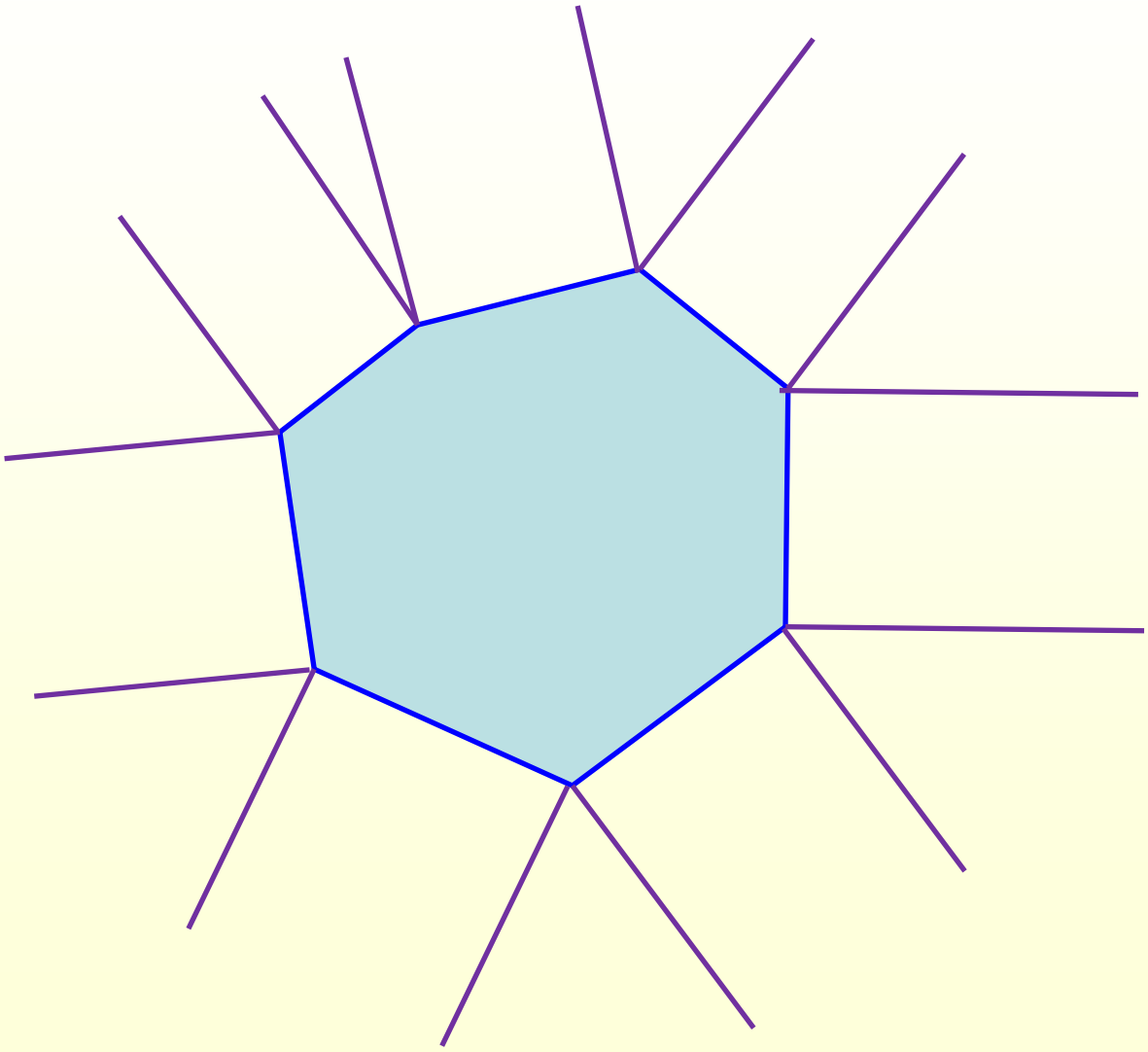
There are applicability criteria for the three of the feature pair combinations:

1. **Point-Vertex** (Vertex-Vertex)
2. **Point-Edge** (Vertex-Edge)
3. **Point-Face** (Vertex-Face)

These criteria can be used in dealing with **Edge-Edge**, **Edge-Face**, and **Face-Face** feature pairs as well (see paper).

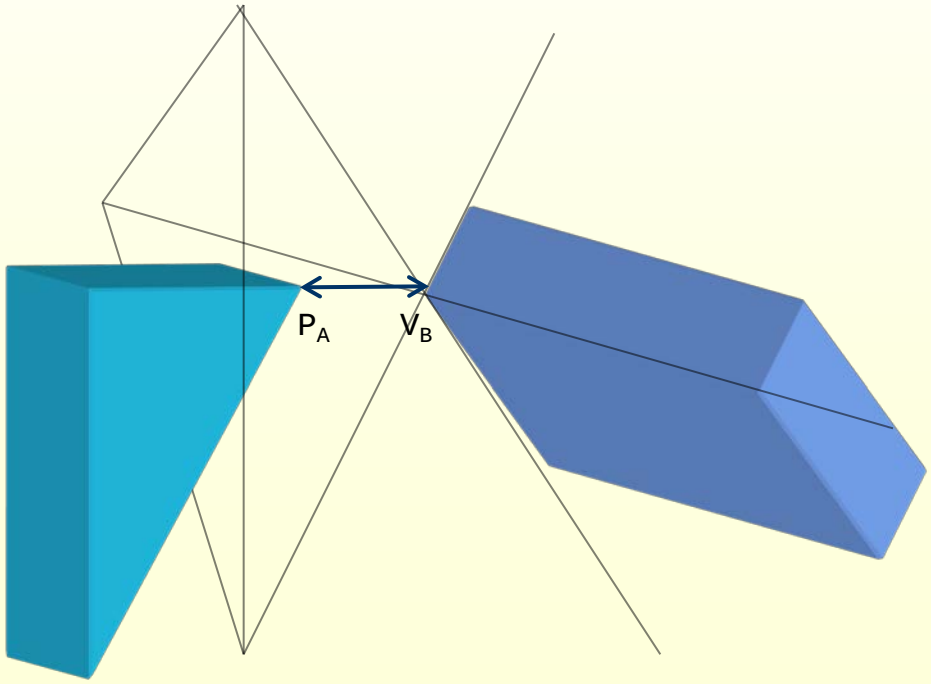


closest feature regions



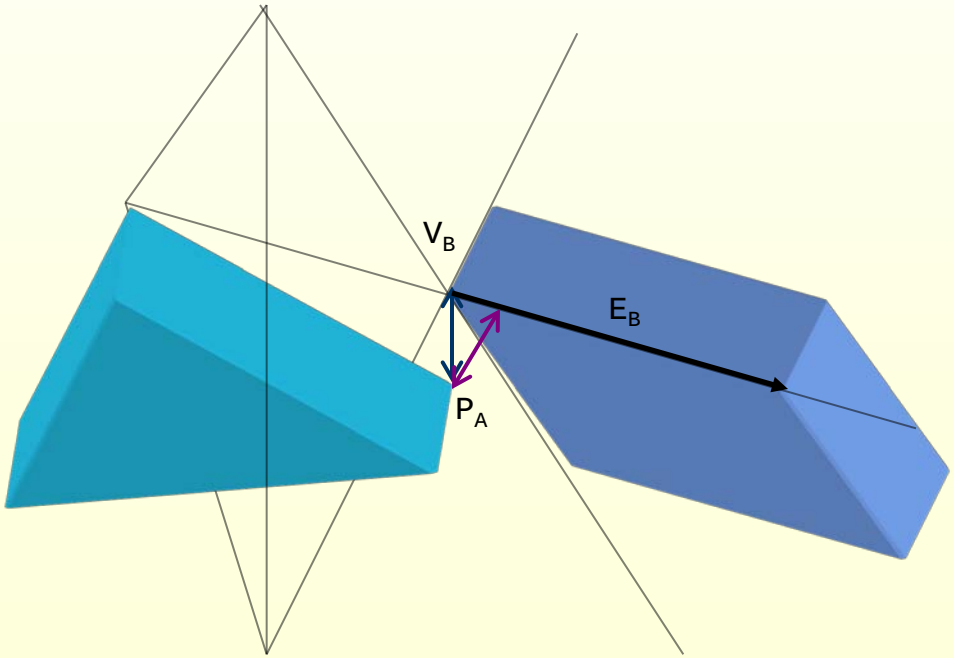
point-vertex applicability criteria

Point P_A and vertex V_B are the closest feature pair if P_A lies within the region bounded by the planes perpendicular to the edges touching V_B.



point-vertex applicability criteria

When P_A lies outside the planar boundaries of V_B (described previously), then some edge E_B is closer to P_A than V_B.

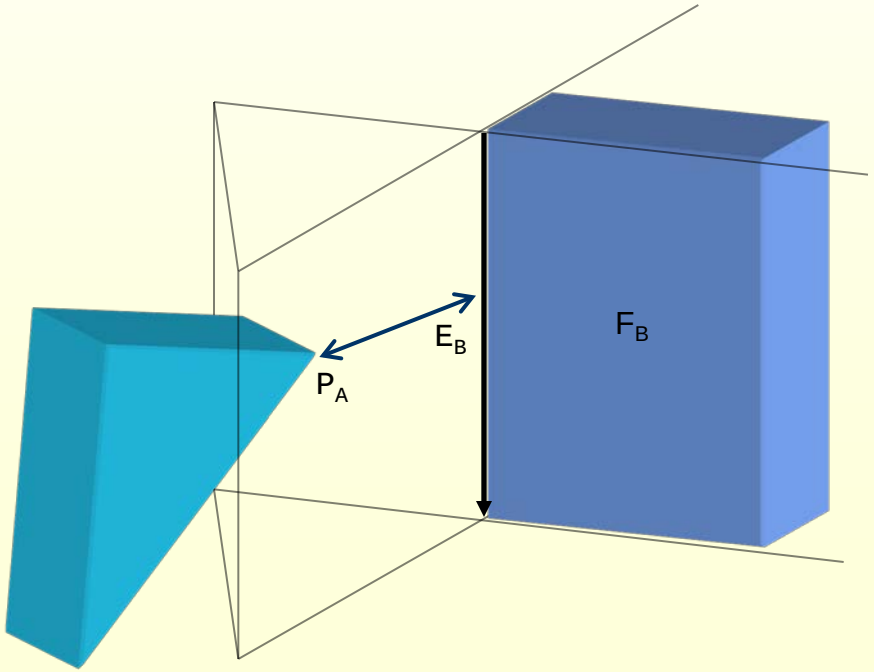


point-edge applicability criteria

Point P_A and edge E_B are the closest feature pair if P_A lies within the region bounded by:

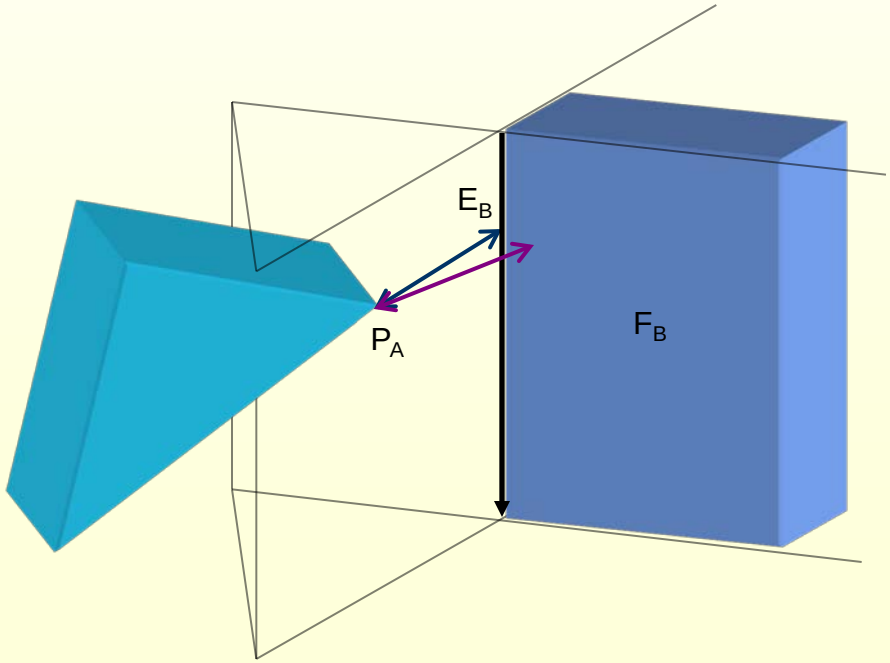
1. The two planes perpendicular to E_B, passing through its head and tail, *and*

2. The two planes perpendicular to the right and left faces of E_B.



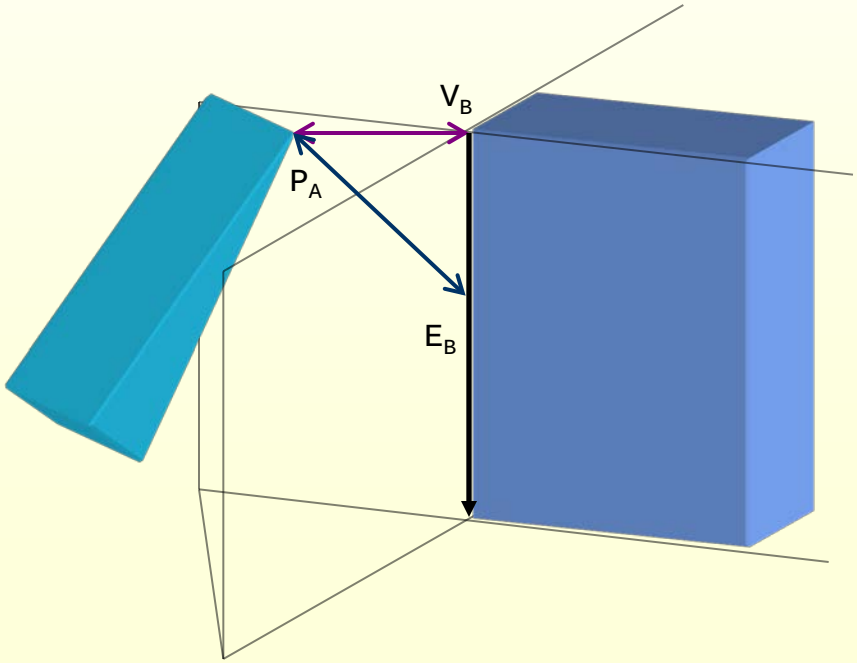
point-edge applicability criteria

When P_A lies outside one of the two planes perpendicular to the right and left faces of E_B, then some face F_B is closer to P_A than E_B.



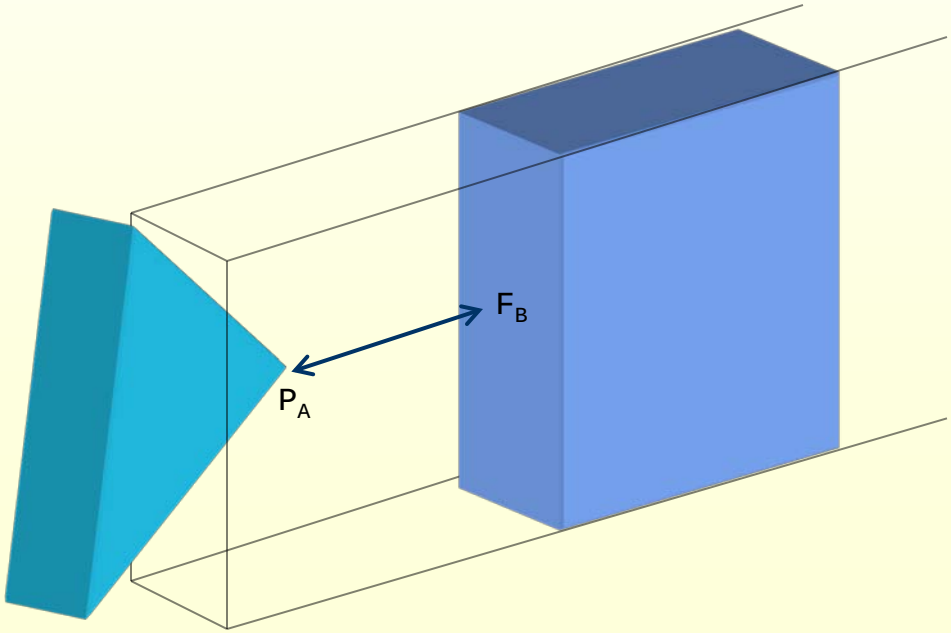
point-edge applicability criteria

When P_A lies outside one of the two planes perpendicular to E_B and passing through either its head or its tail, then some vertex V_B is closer to P_A than E_B.



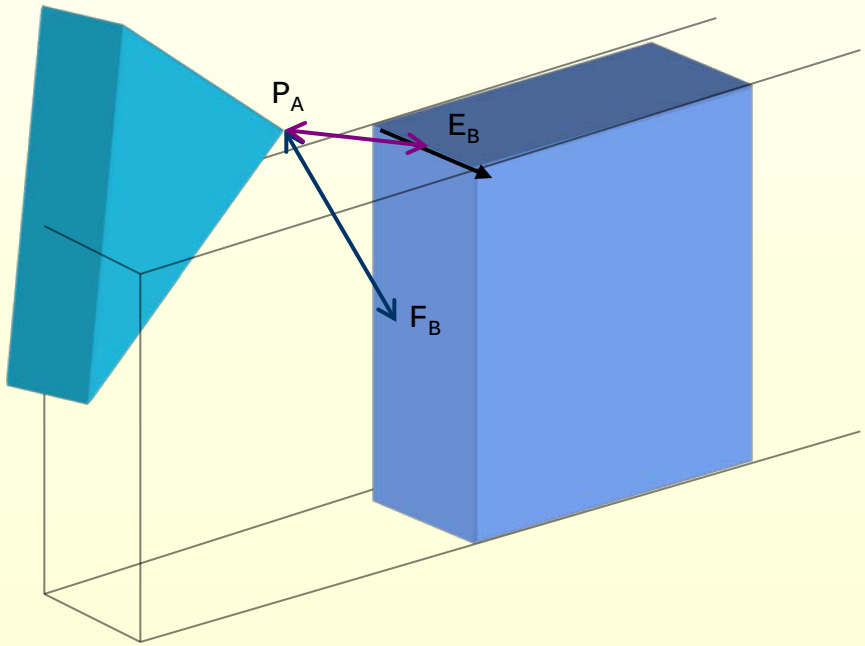
point-face applicability criteria

Point P_A and face F_B are the closes feature pair if P_A lies within the region bounded by the planes that are both perpendicular to F_B and contain the boundaries of F_B.



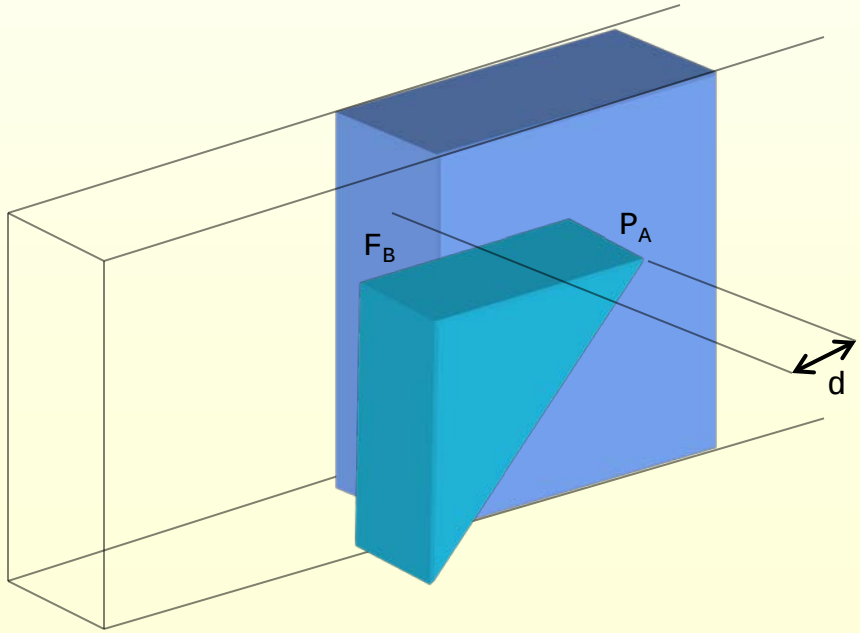
point-face applicability criteria

If P_A lies outside the planar boundaries of F_B (described previously) then some edge E_B is closer to P_A than F_B.



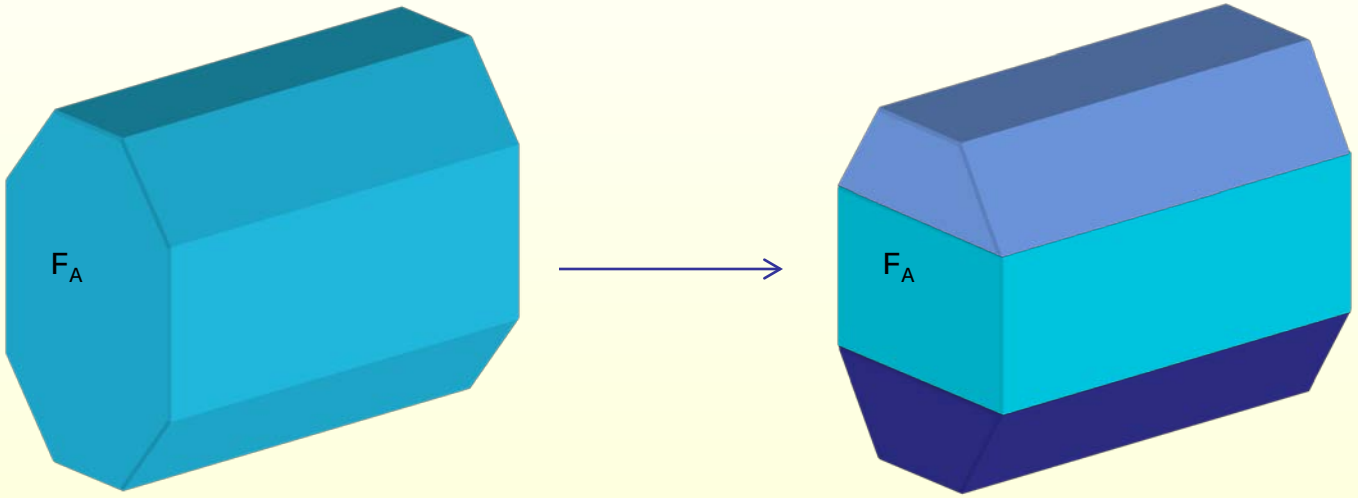
point-face applicability criteria

If P_A lies “below” F_B then some feature of B is closer to P_A than F_B (otherwise A would have collided into B). In this case, we must check every face of object B. This is NOT constant time. Note however that this can only happen during initialization; otherwise, the collision would have been detected.



preprocessing

Preprocessing steps ensure that objects have boundaries and co-boundaries of constant size (eg. limiting the number of edges going through each vertex and bounding each face to 4 or 5).



Constant sized boundaries allows constant time verification of a closest feature pair.

results and conclusions…

Lin-Canny runs in quadratic time when intially finding a closest pair

Finding the initial closest feature pair is in the worst case $O(n^2)$ in number of features per object.

Lin-Canny runs in constant time in an "incremental movement" framework

Once an initial closest feature pair is found, it takes on average constant time to keep track of that closest pair.

conclusion

Lin-Canny is a simple and efficient algorithm that is guaranteed to find the closest feature and point pair.

Results have applications in collision detection and motion planning.

Lin-Canny can be extended to handle non-convex objects with only modest increase in running time.

Complex Geometry

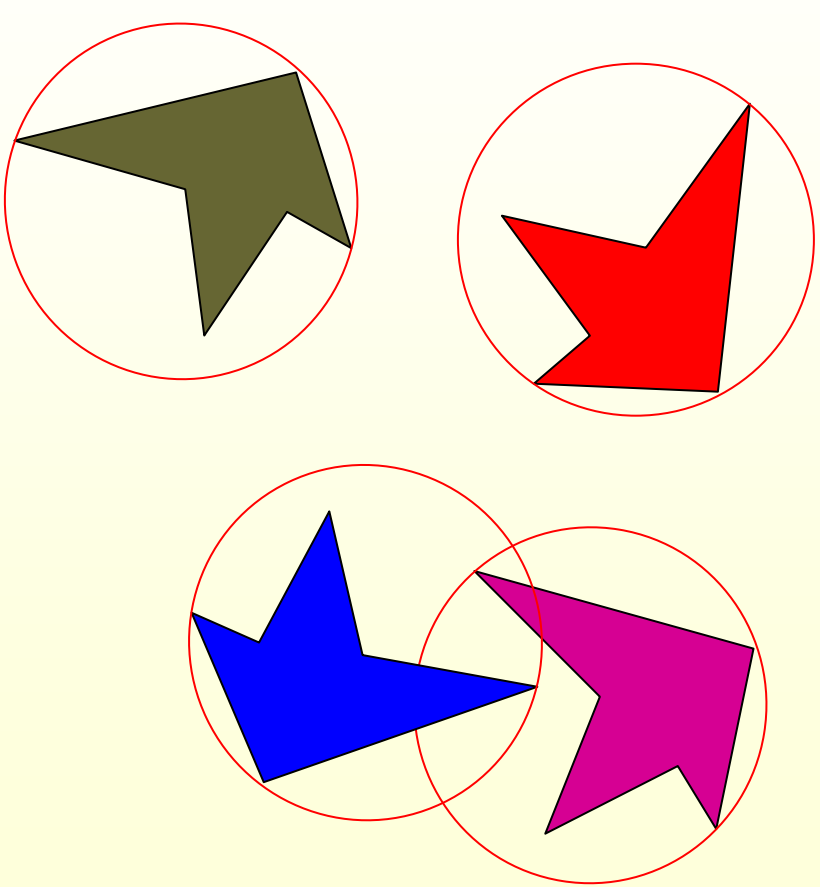
Collision Detection: Complex/Multiple Objects

- Many different methods to avoid quadratic overhead
- We will focus on two of them:
  - Grid methods: good for many simple moving objects of about the same size (e.g., many moving discs with similar radii). Generalizes to space partition methods.
  - Bounding Volume methods: good for few moving objects with complex geometry

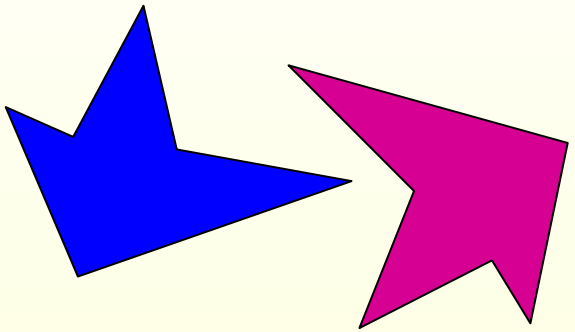
Hierarchical Representations

- Supported by both the above:
  - Spatial decompositions - BSP, K-d trees, octrees, MSP tree, R-trees, grids/cells, space-time bounds, etc.
  - Bounding volume hierarchies – trees of spheres, ellipses, cubes, axis-aligned bounding boxes (AABBs), oriented bounding boxes (OBBs), K-dop, SSV, etc.
- Do very well in “rejection tests”, when objects are far apart
- Performance may slow down, when the two objects are in close proximity and may have multiple contacts

Broad vs. Narrow Phase



Broad phase



Narrow phase

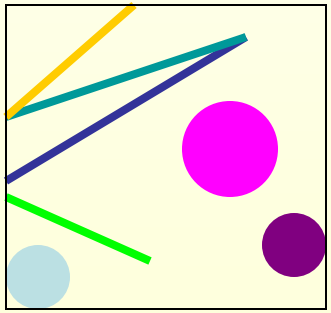
BVH vs. Spatial Partitioning

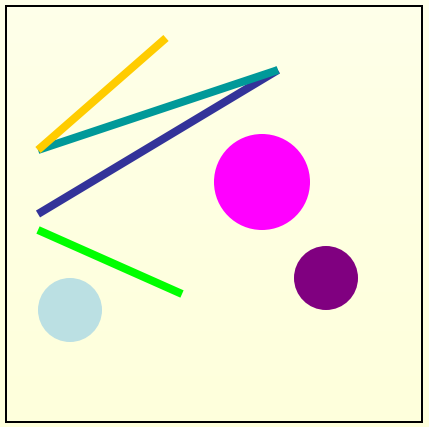
BVH:

- Object centric (Lagrangian)

SP:

- Space centric (Eulerian)





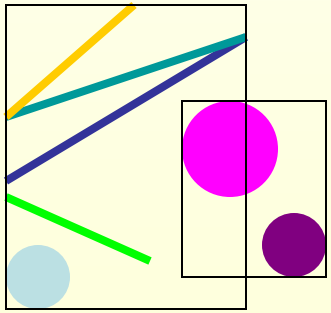
BVH vs. Spatial Partitioning

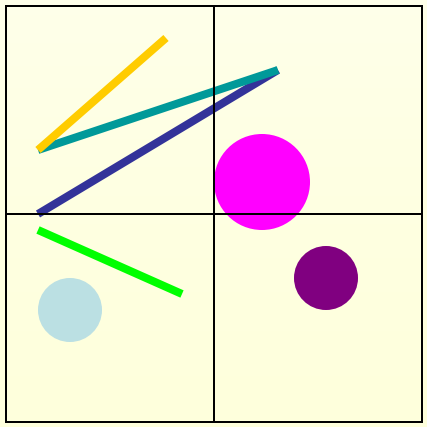
BVH:

- Object centric (Lagrangian)

SP:

- Space centric (Eulerian)





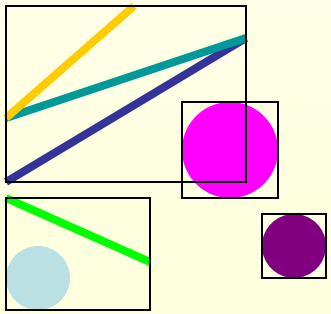
BVH vs. Spatial Partitioning

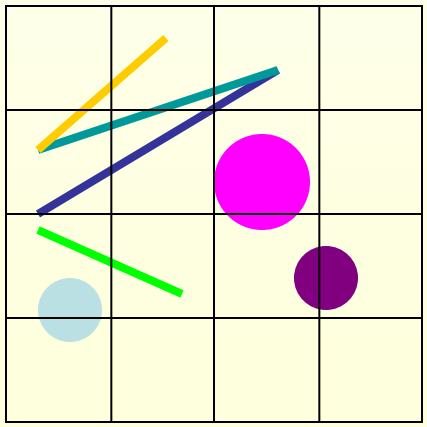
BVH:

- Object centric (Lagrangian)

SP:

- Space centric (Eulerian)





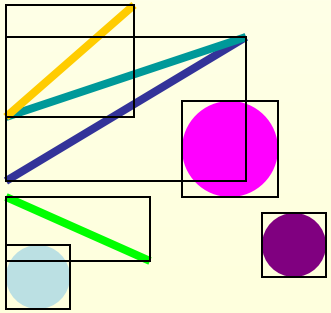
BVH vs. Spatial Partitioning

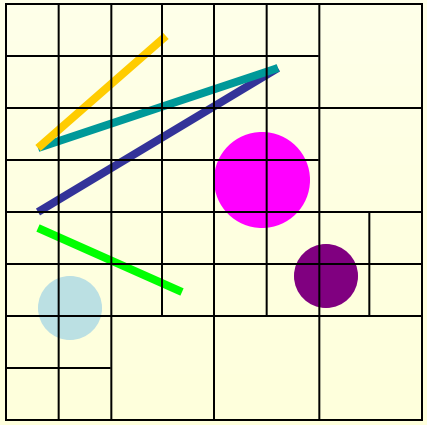
BVH:

- Object centric (Lagrangian)

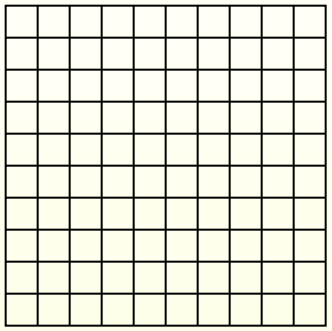
SP:

- Space centric (Eulerian)

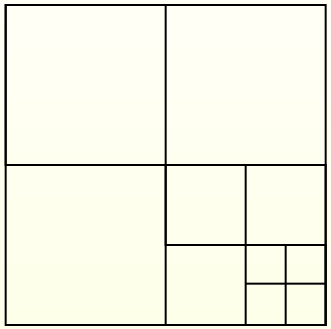




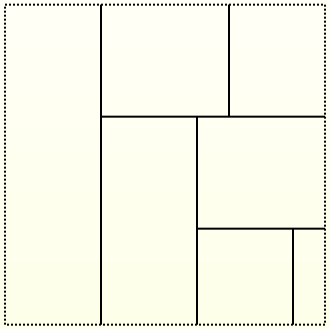
Spatial Data Structures & Subdivision



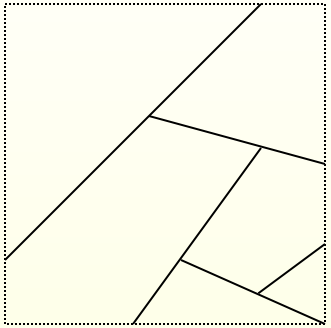
Uniform Spatial Sub



Quadtree/Octree



kd-tree

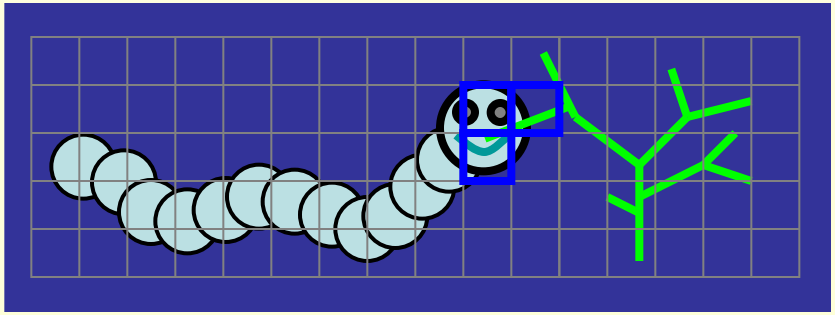


BSP-tree

- Many others……

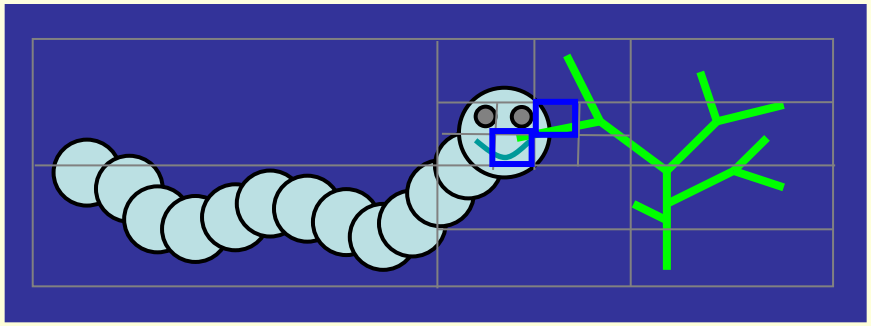
Uniform Spatial Subdivision

- Decompose the objects (the entire simulated environment) into identical cells arranged in a fixed, regular grids (equal size boxes or voxels)
- To represent an object, only need to decide which cells are occupied. To perform collision detection, check if any cell is occupied by two objects
- Storage: to represent an object at resolution of *n* voxels per dimension requires upto n^3 cells
- Accuracy: solids can only be “approximated”



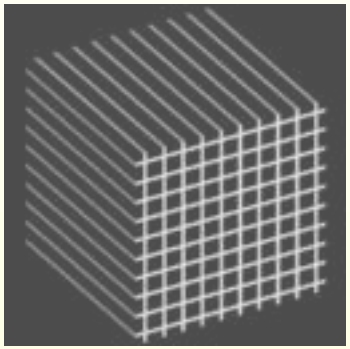
Octrees

- *Quadtree* is derived by subdividing a 2D-plane in both dimensions to form quadrants
- Octrees are a 3D-extension of quadtree
- Use divide-and-conquer
- Reduce storage requirements (in comparison to grids/voxels)



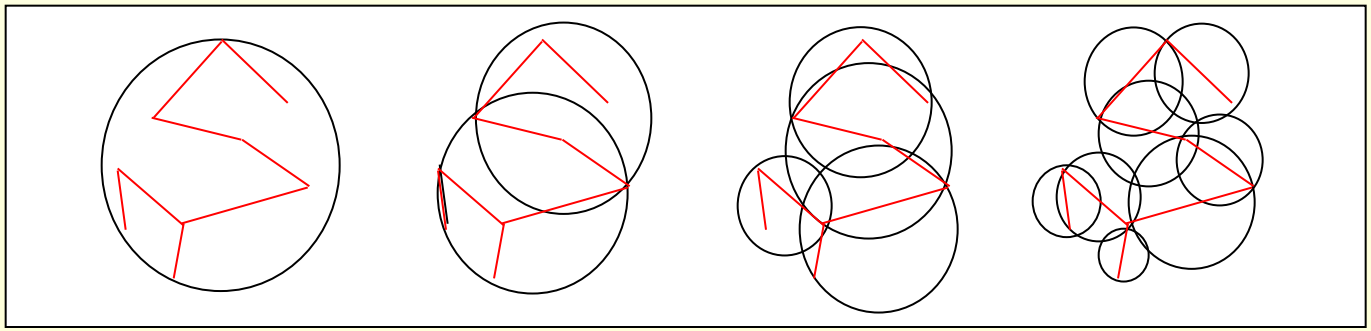
Hashed Voxel Grids

- Partition space into a (possibly hierarchical) voxel grid and apportion the objects into cells of that grid
  - Use a hash table, since many cells will be empty
  - To find the neighbors of a given object, search nearby cells
- Requires re-apportioning all the objects into cells after each simulation time step
  - Bad when system is moving but not deforming much
- Efficiency highly dependent on voxel grid size chosen



Bounding Volume Hierarchies

- Model Hierarchy:
  - each node has a simple volume that bounds a set of triangles
  - children contain volumes that each bound a different portion of the parent's triangles
  - the leaves of the hierarchy usually contain individual triangles
- A binary bounding volume hierarchy:

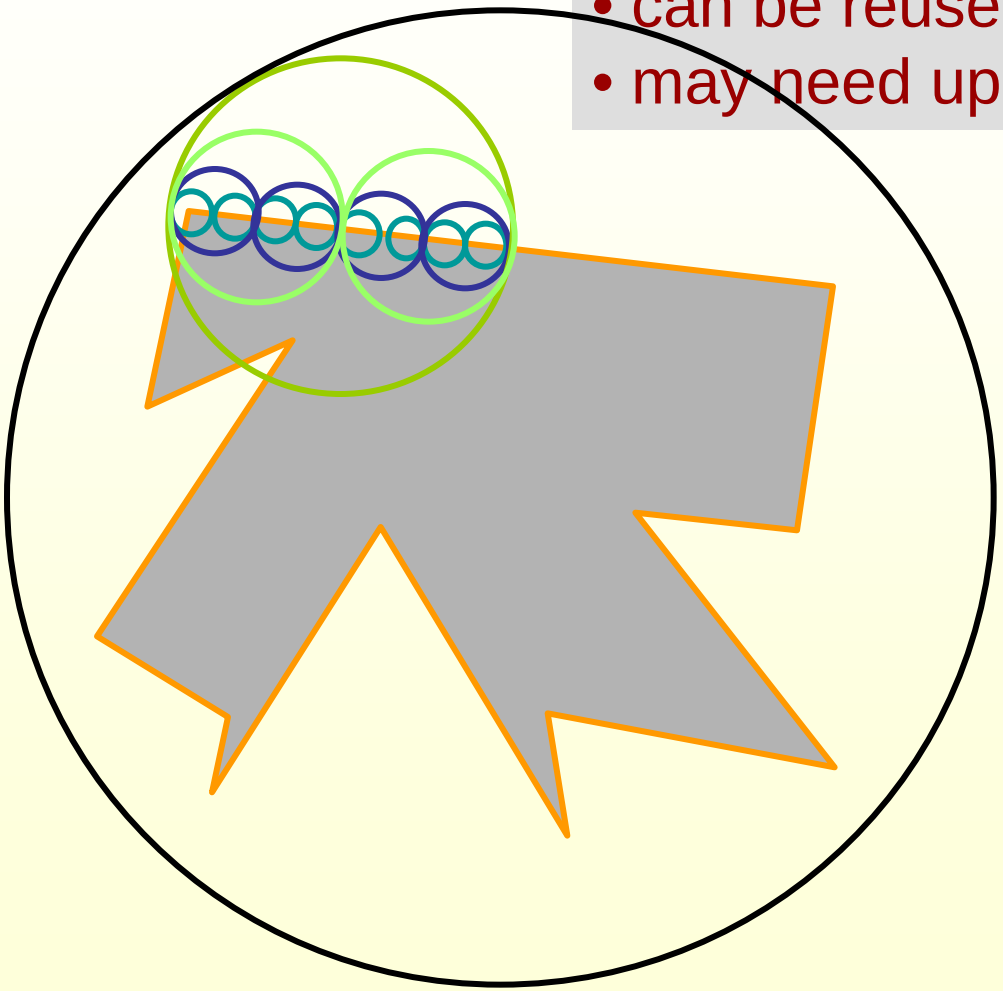


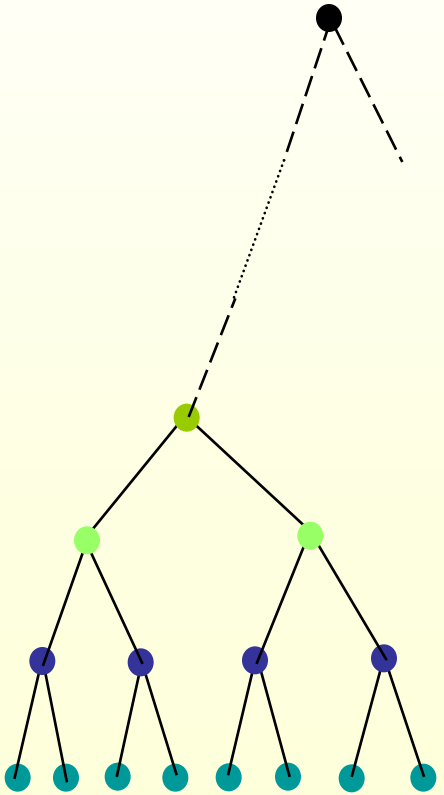
Type of Bounding Volumes

- Spheres
- Ellipsoids
- Axis-Aligned Bounding Boxes (AABB)
- Oriented Bounding Boxes (OBBs)
- Convex Hulls
- *k*-Discrete Orientation Polytopes (*k*-dop)
- Spherical Shells
- Swept-Sphere Volumes (SSVs)
  - Point Swept Spheres (PSS)
  - Line Swept Spheres (LSS)
  - Rectangle Swept Spheres (RSS)
  - Triangle Swept Spheres (TSS)

Bounding Volume Hierarchy Method

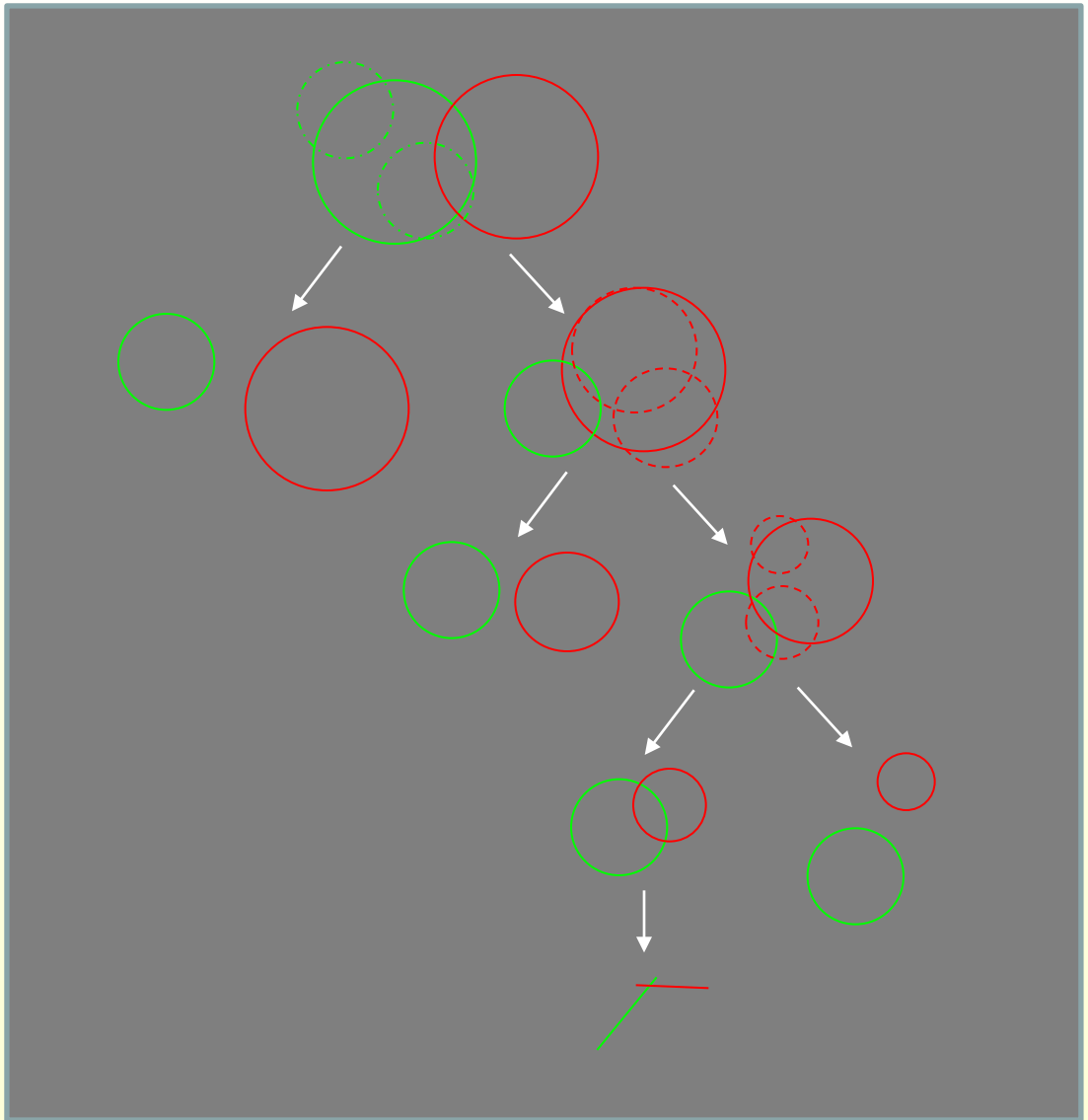
- BVH is pre-computed for each object
- can be reused for rigid objects
- may need update for deformable objects





BVH in 3D

BVH-Based Collision Detection



Collision Detection using BVH

1. Check for collision between two parent nodes (starting from the roots of two given trees)
2. If there is no interference between two parents,
3. Then stop and report “no collision”
4. Else all children of one parent node are checked against all children of the other node
5. If there is a collision between the children
6. Then If at leave nodes
7. Then report “collision”
8. Else go to Step 4
9. Else stop and report “no collision”

Evaluating Bounding Volume Hierarchies

Cost Function:

$$F = N_u \times C_u + N_{bv} \times C_{bv} + N_p \times C_p$$

F: total cost function for interference detection

N_u: no. of bounding volumes updated

C_u: cost of updating a bounding volume,

N_{bv}: no. of bounding volume pair overlap tests

C_{bv}: cost of overlap test between 2 BVs

N_p: no. of primitive pairs tested for interference

C_p: cost of testing 2 primitives for interference

Designing Bounding Volume Hierarchies

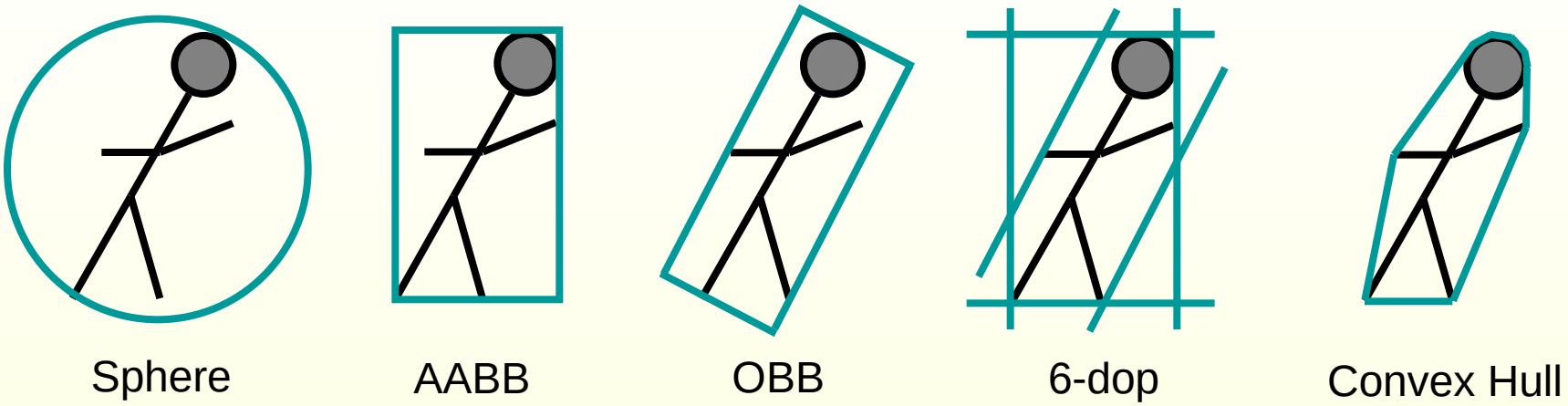
The choice governed by these constraints:

- It should fit the original model as tightly as possible (to lower N_{bv} and N_p)
- Testing two such volumes for overlap should be as fast as possible (to lower C_{bv})
- It should require the BV updates as infrequently as possible (to lower N_u)

Observation – Trade Offs

- Simple primitives (spheres, AABBs, etc.) do very well with respect to the second constraint. But they cannot fit some long skinny primitives tightly
- More complex primitives (minimal ellipsoids, OBBs, etc.) provide tight fits, but checking for overlap between them is relatively expensive
- Cost of BV updates needs to be considered

Trade-off in Choosing BV's



increasing complexity & tightness of fit

decreasing cost of (overlap tests + BV update)

Building Hierarchies

- Choices of Bounding Volumes
  - cost function & constraints
- Top-Down vs. Bottum-up
  - speed vs. fitting
- Depth vs. breadth
  - branching rules
  - Balancing considerations

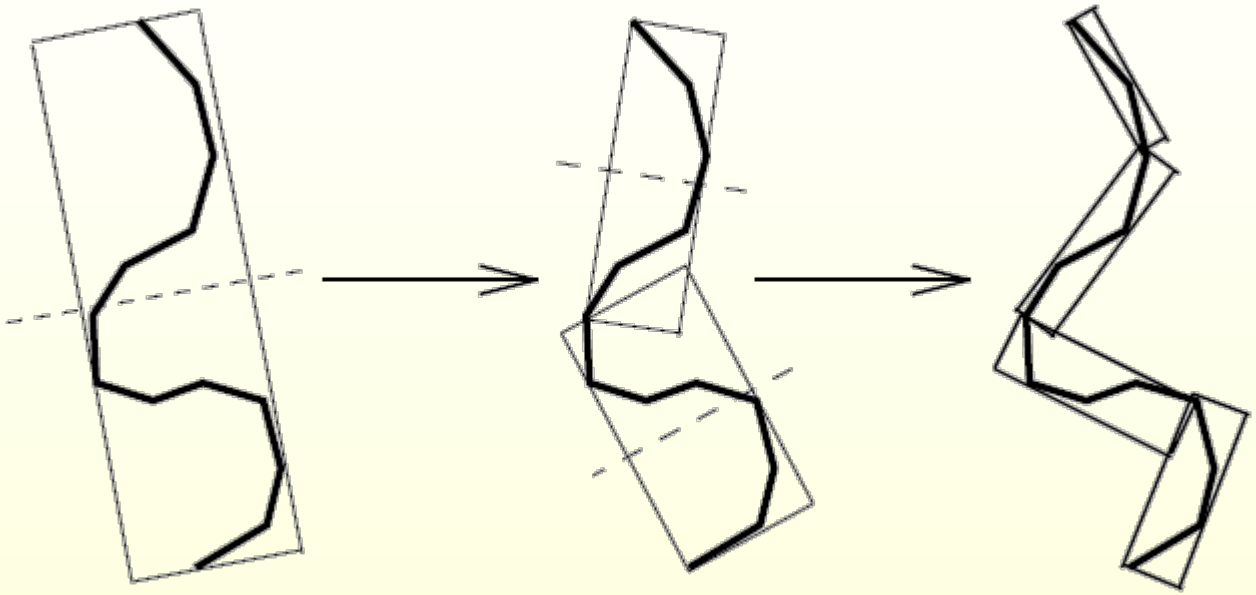
Sphere-Trees

- A *sphere-tree* is a hierarchy of sets of spheres, used to approximate an object
- Advantages:
  - Simplicity in checking overlaps between two bounding spheres
  - Invariant to rotations and can apply the same transformation to the centers, if objects are rigid
- Shortcomings:
  - Not always the best approximation (bad for long, skinny objects)
  - building good sphere-trees relatively complex

Methods for Building Sphere-Trees

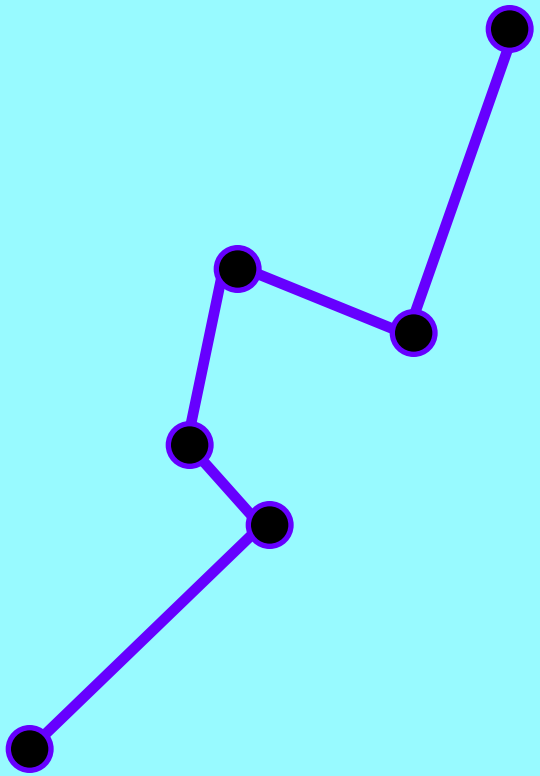
- start from a regular voxel grid and cover the geometry in each non-empty cell by a sphere
- aggregate nearby spheres into large spheres
  - fit spheres, or fit geometry
- Compute the medial axis and cover the shape with expanded medial balls
- Others……

Building an OBB Tree



Recursive top-down construction: partition and refit

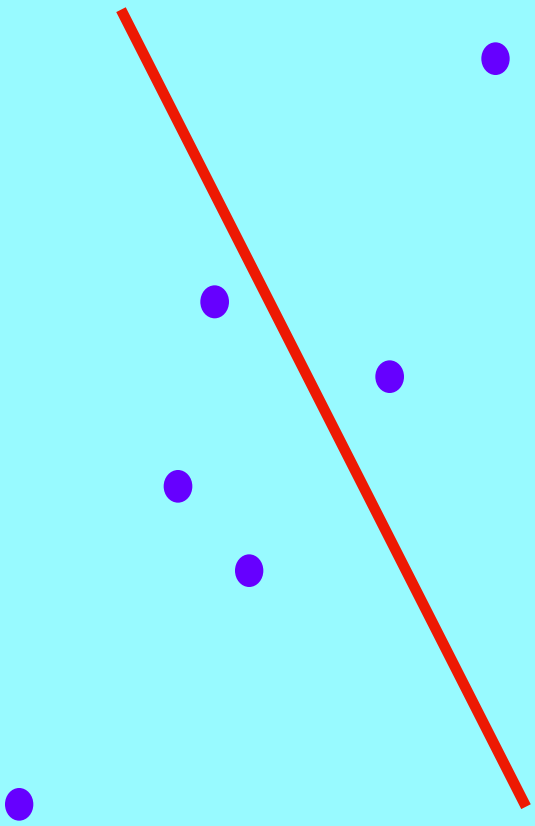
Building an OBB Tree



Given some polygons, consider their vertices...

Building an OBB Tree

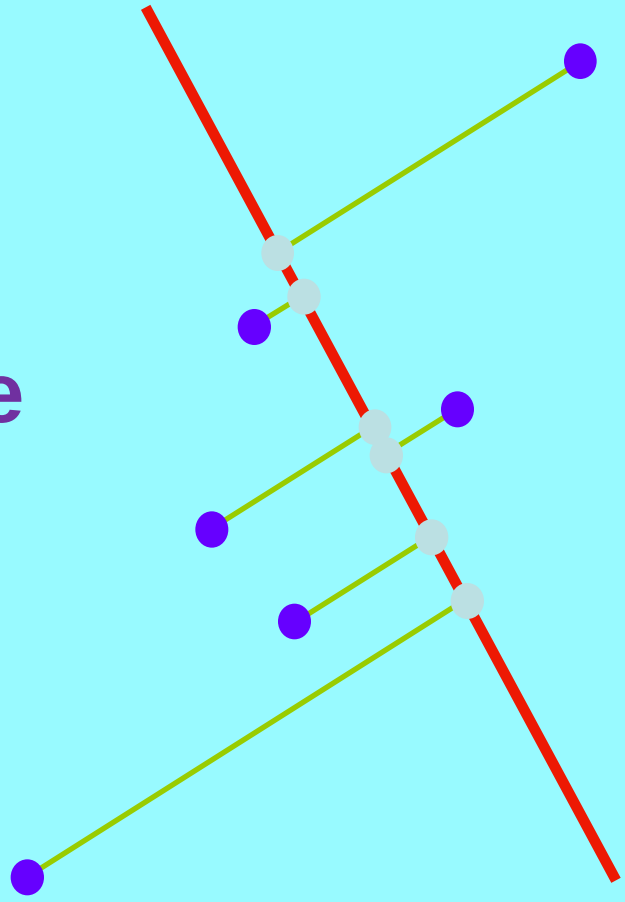
… and an arbitrary line



Building an OBB Tree

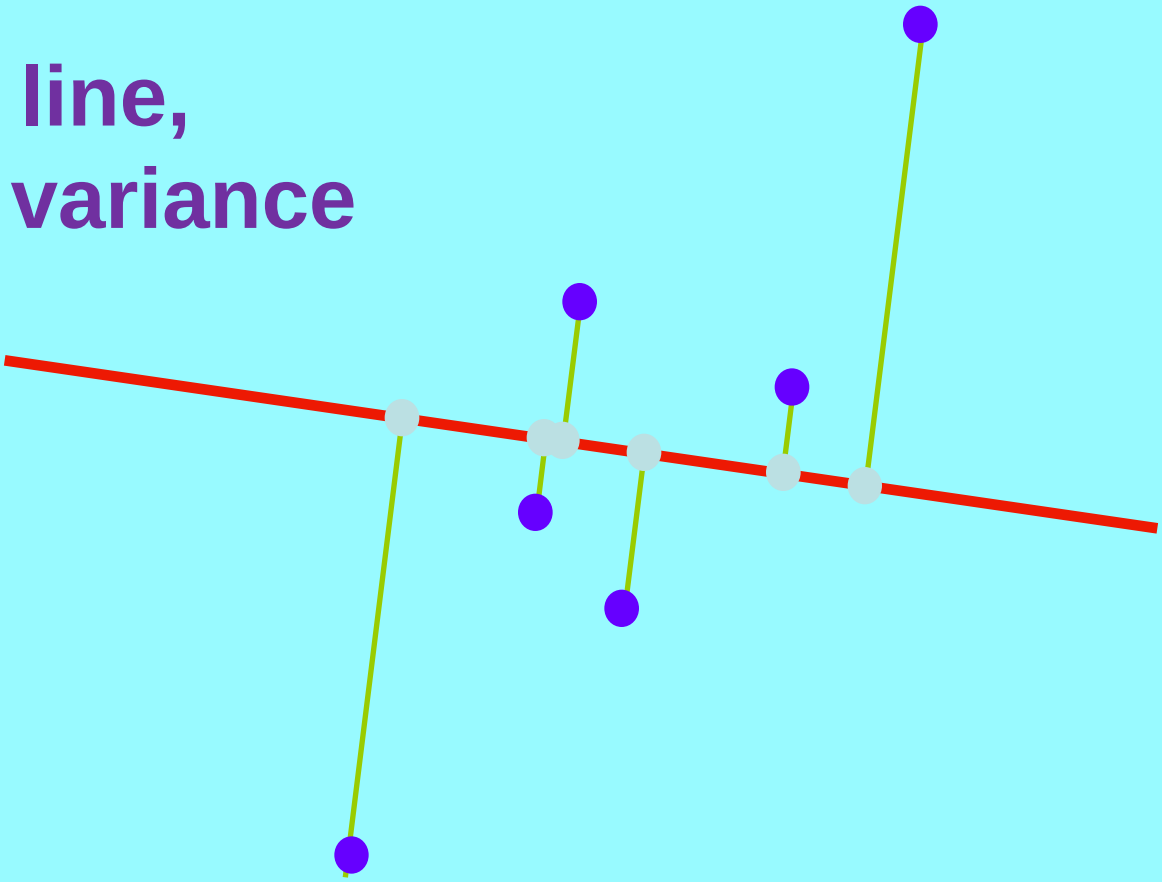
Project onto the line

Consider variance of distribution on the line



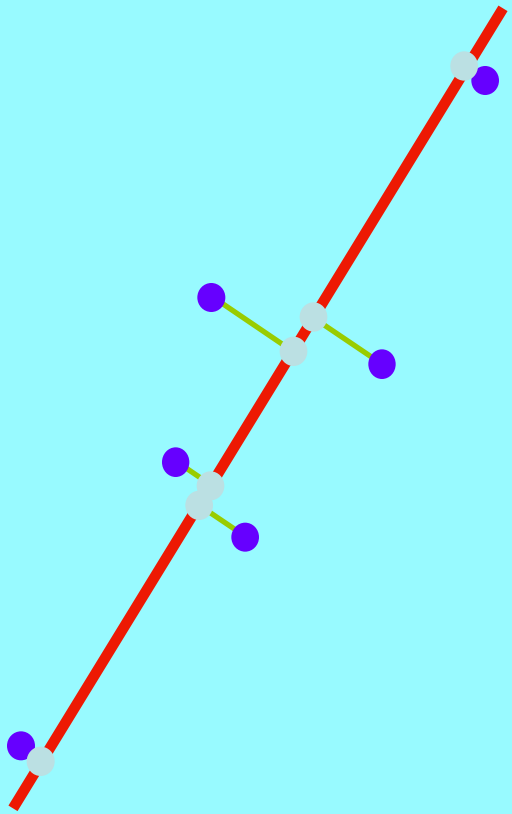
Building an OBB Tree

Different line,
different variance

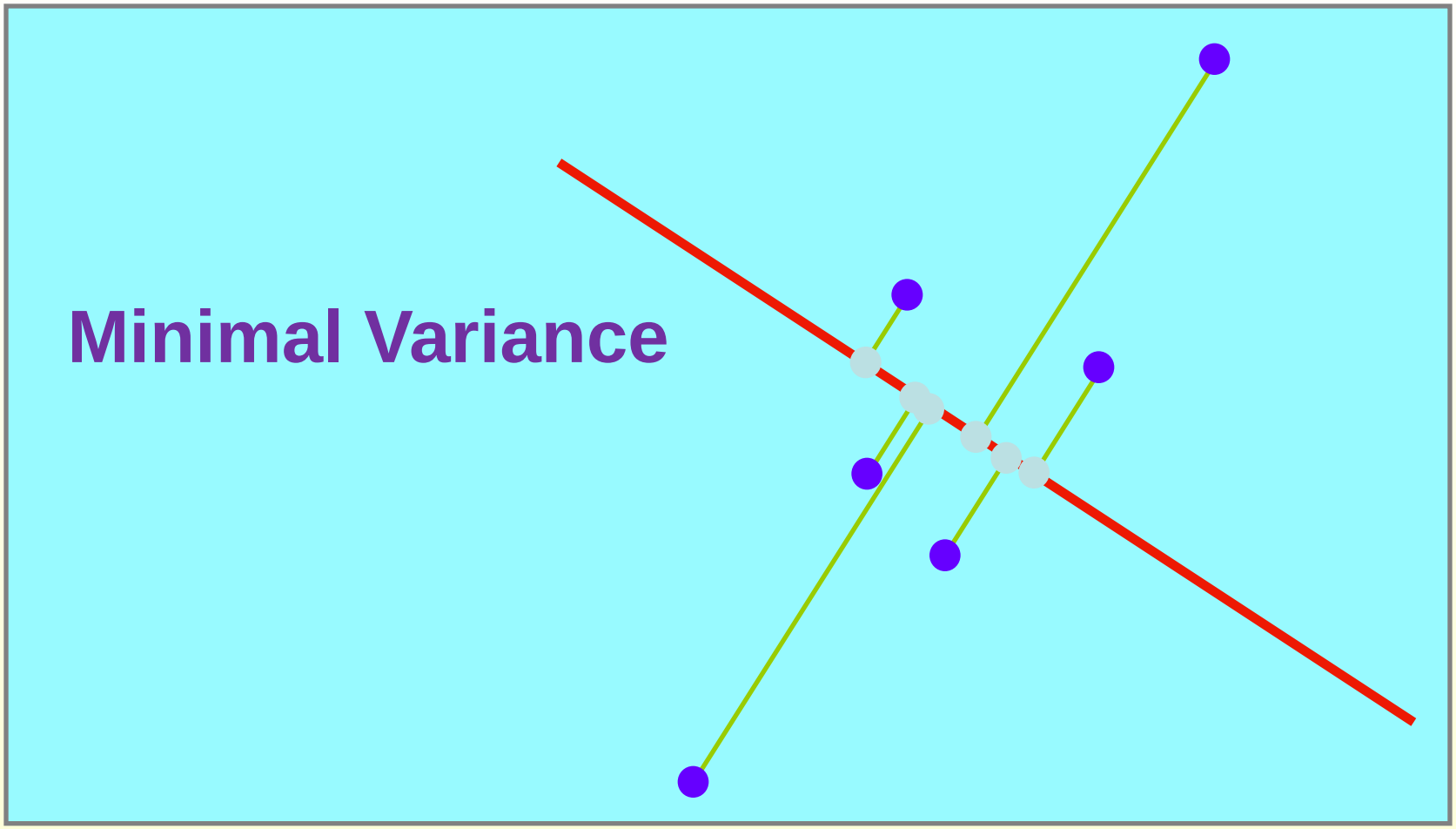


Building an OBB Tree

Maximum Variance

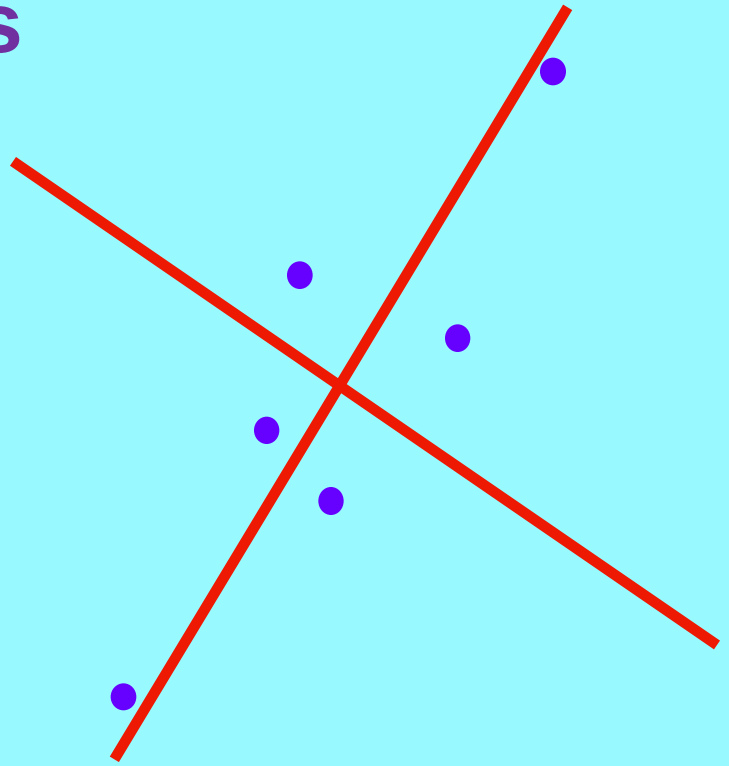


Building an OBB Tree



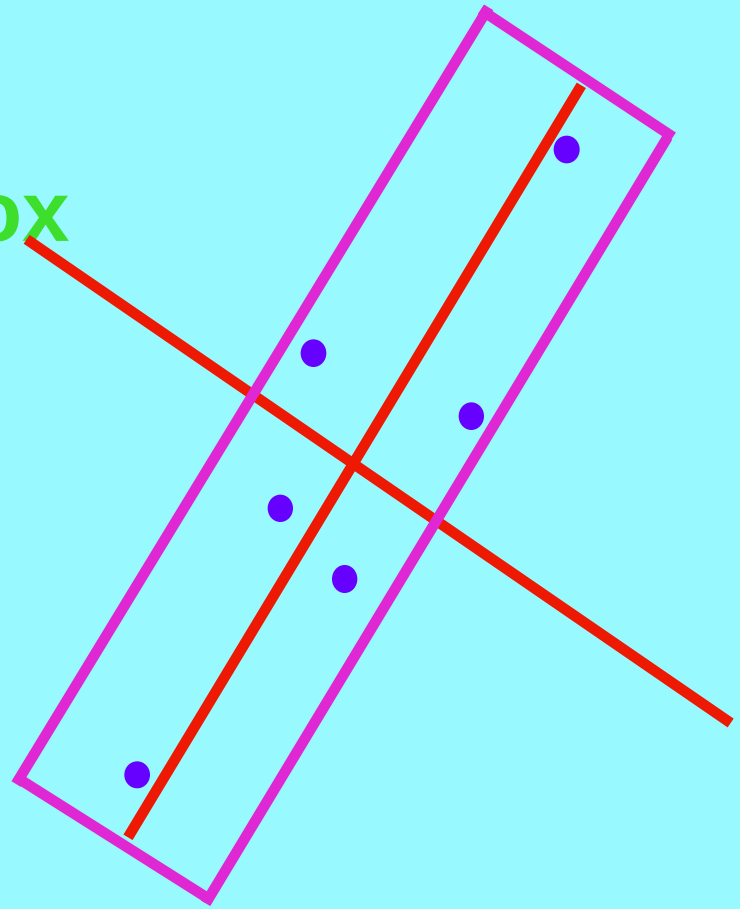
Building an OBB Tree

Given by eigenvectors
of covariance matrix
of coordinates
of original points



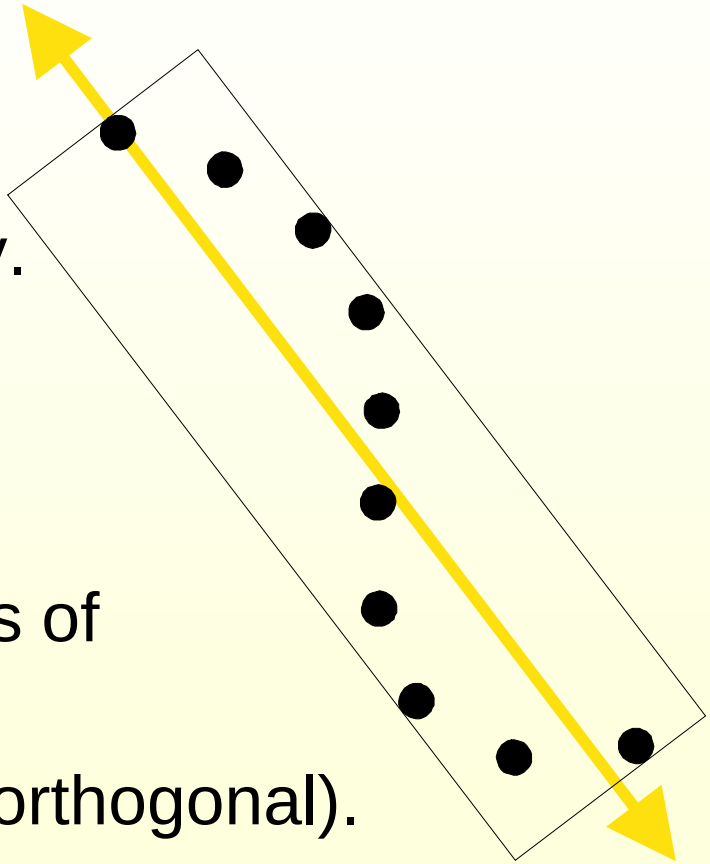
Building an OBB Tree

Choose bounding box oriented this way



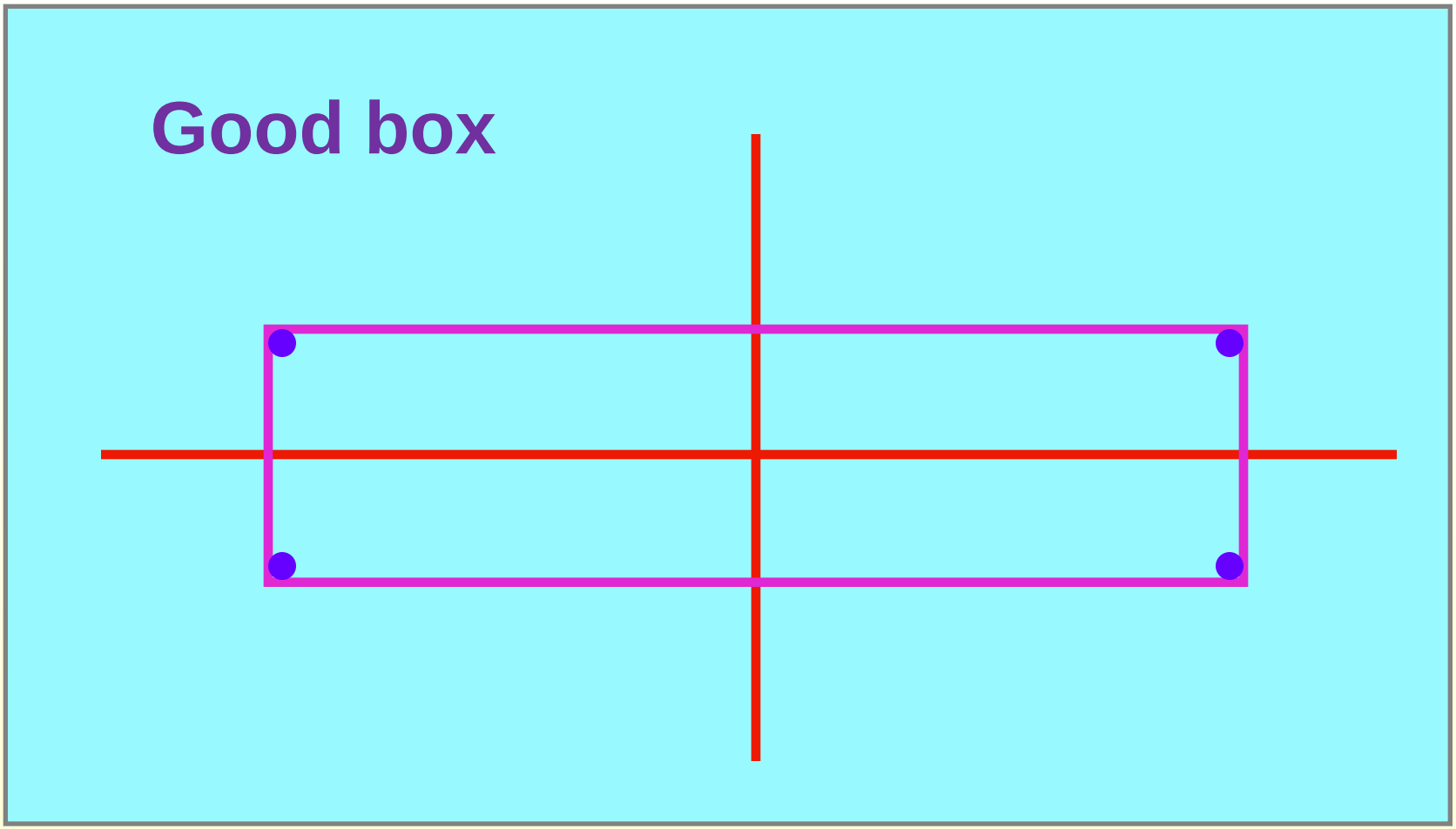
Building an OBB Tree: Fitting

Covariance matrix of
point coordinates describes
statistical spread of geometry.



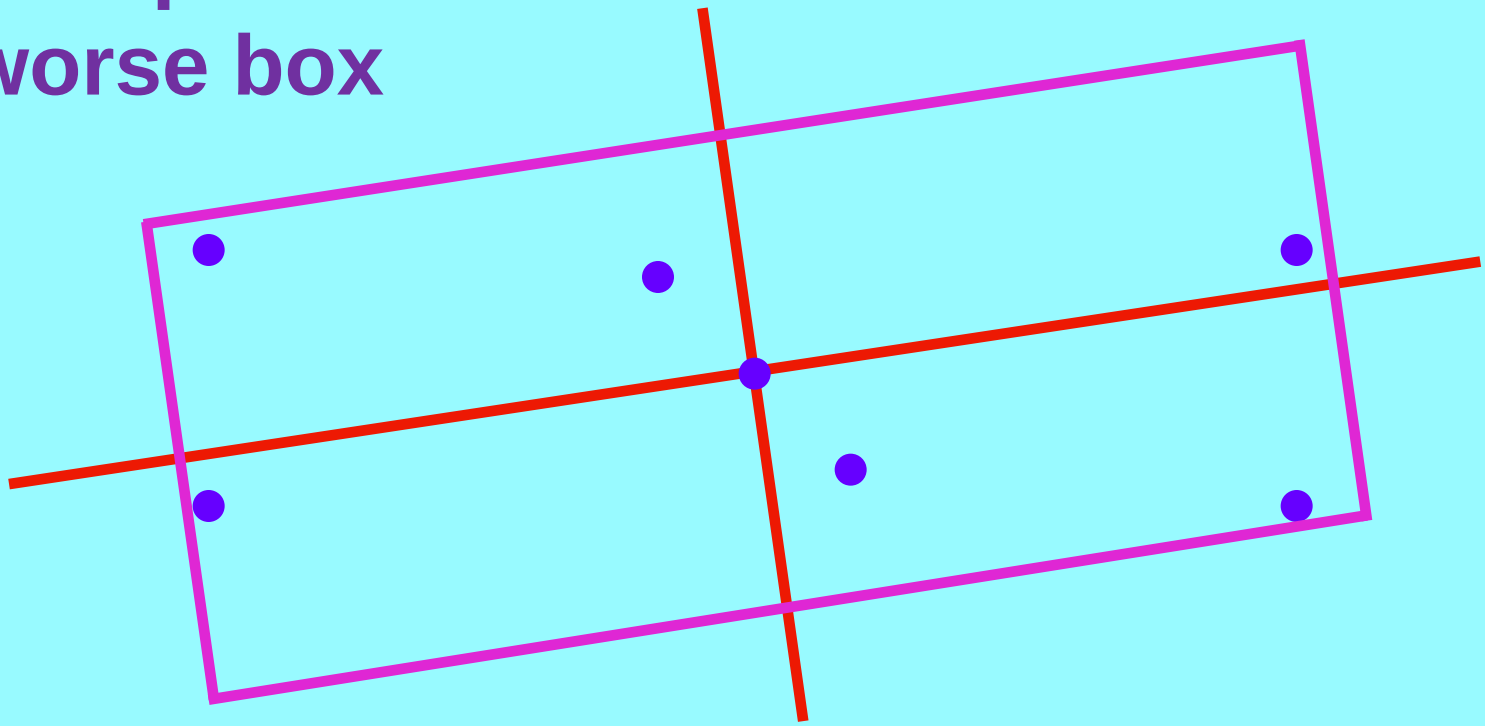
OBB is aligned with directions of
greatest and least spread
(which are guaranteed to be orthogonal).

Building an OBB Tree

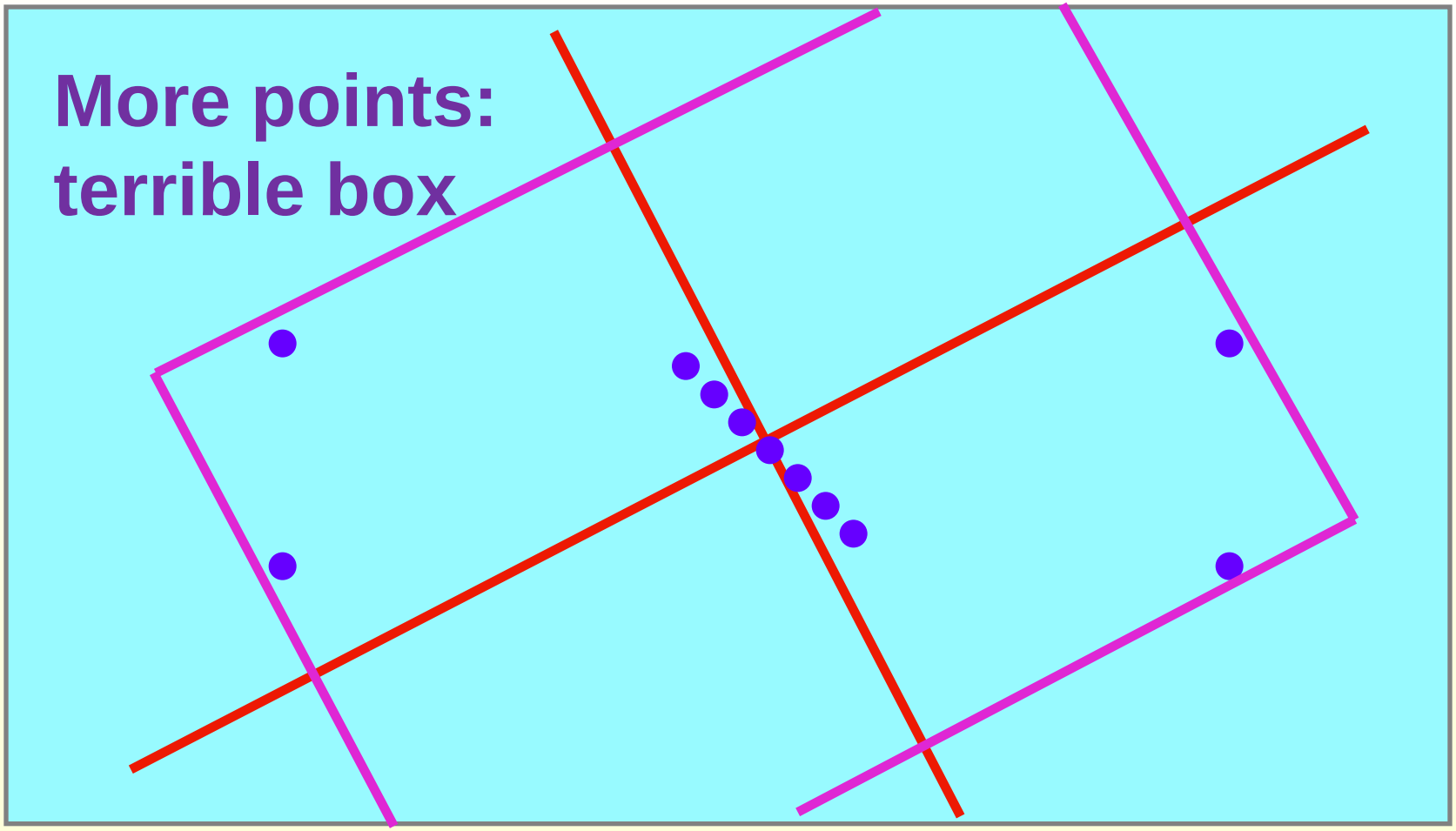


Building an OBB Tree

Add points:
worse box

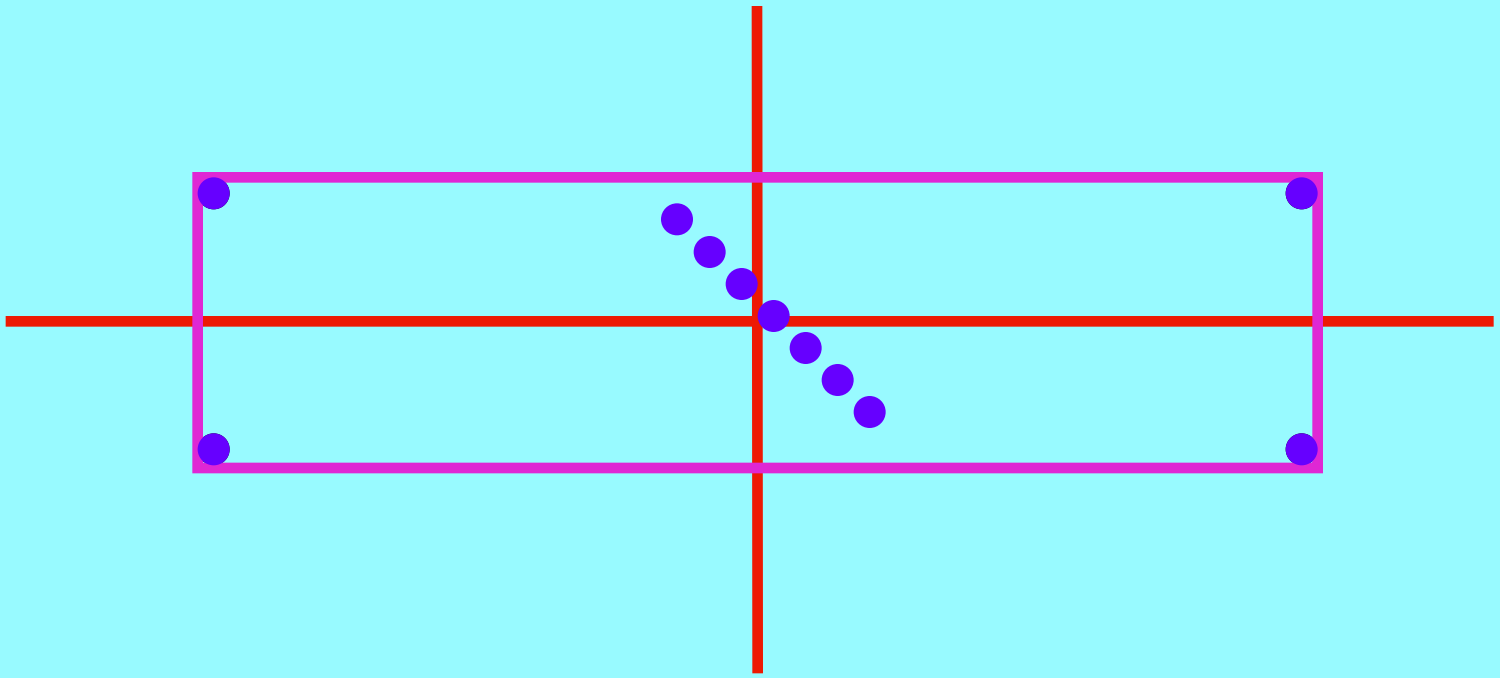


Building an OBB Tree



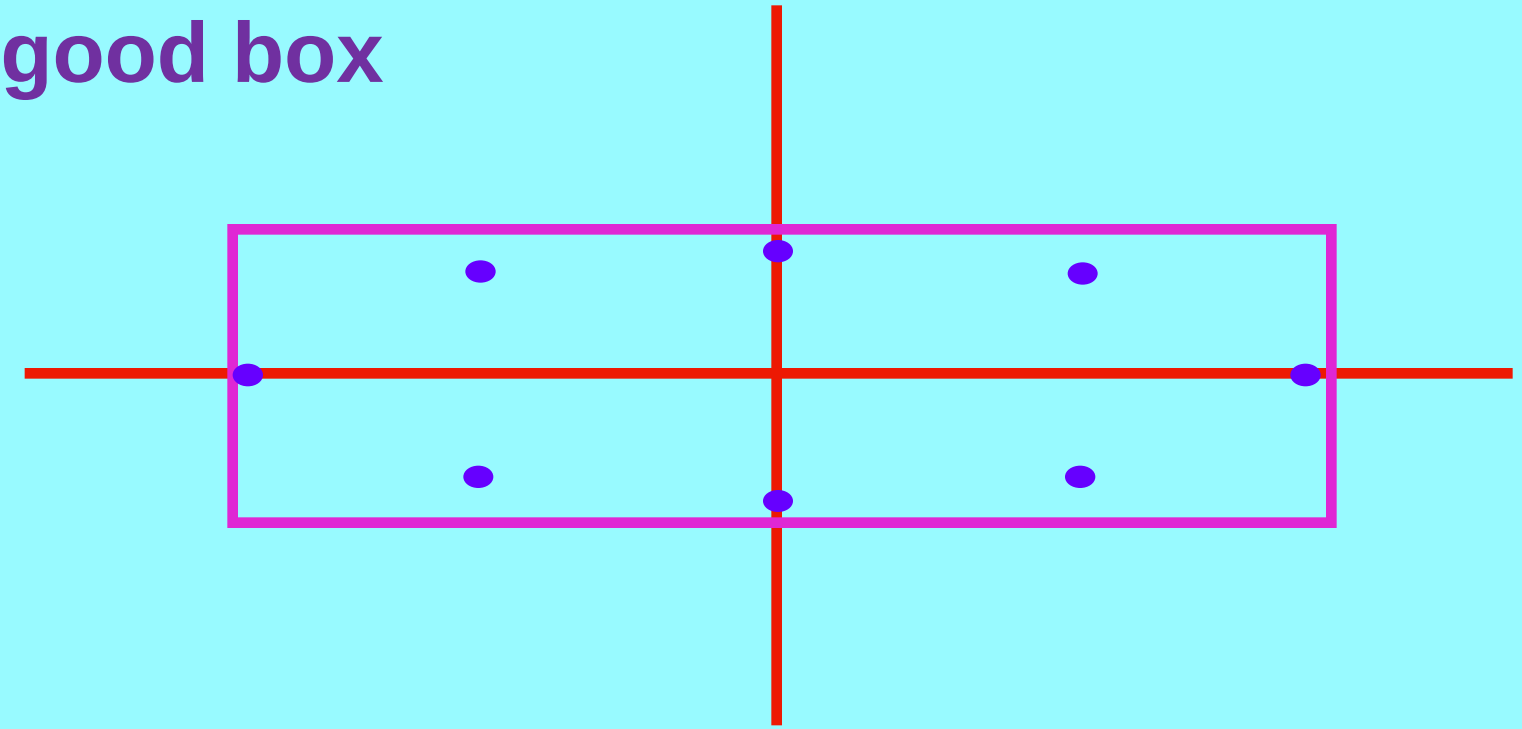
Building an OBB Tree

Compute with extremal points only



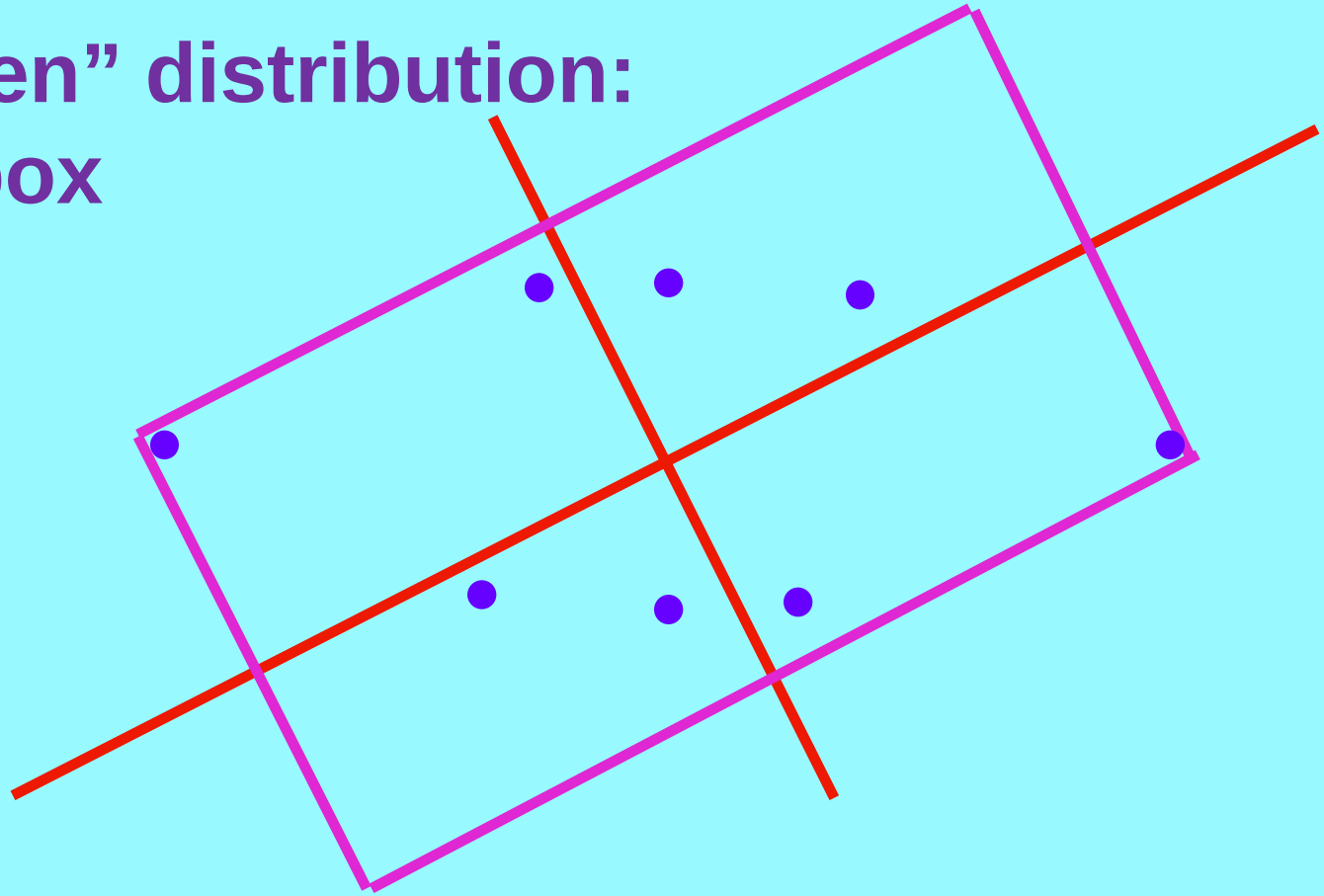
Building an OBB Tree

“Even” distribution:
good box



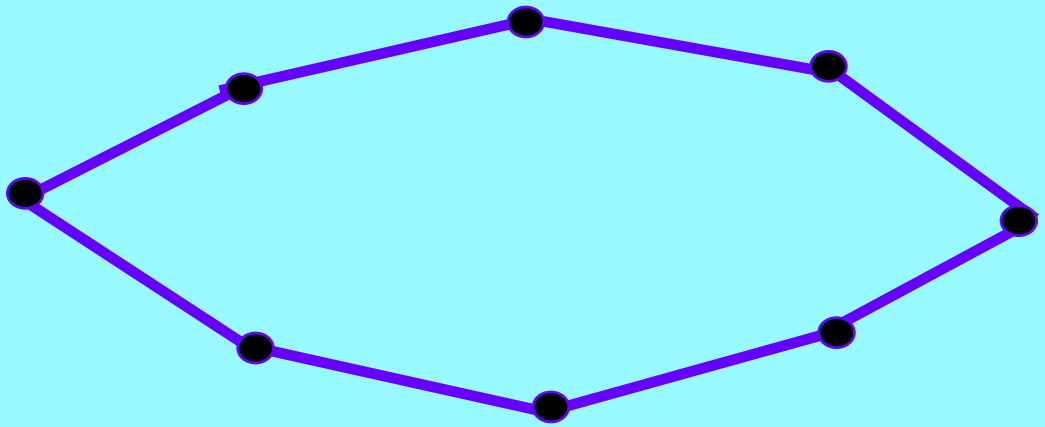
Building an OBB Tree

“Uneven” distribution: bad box



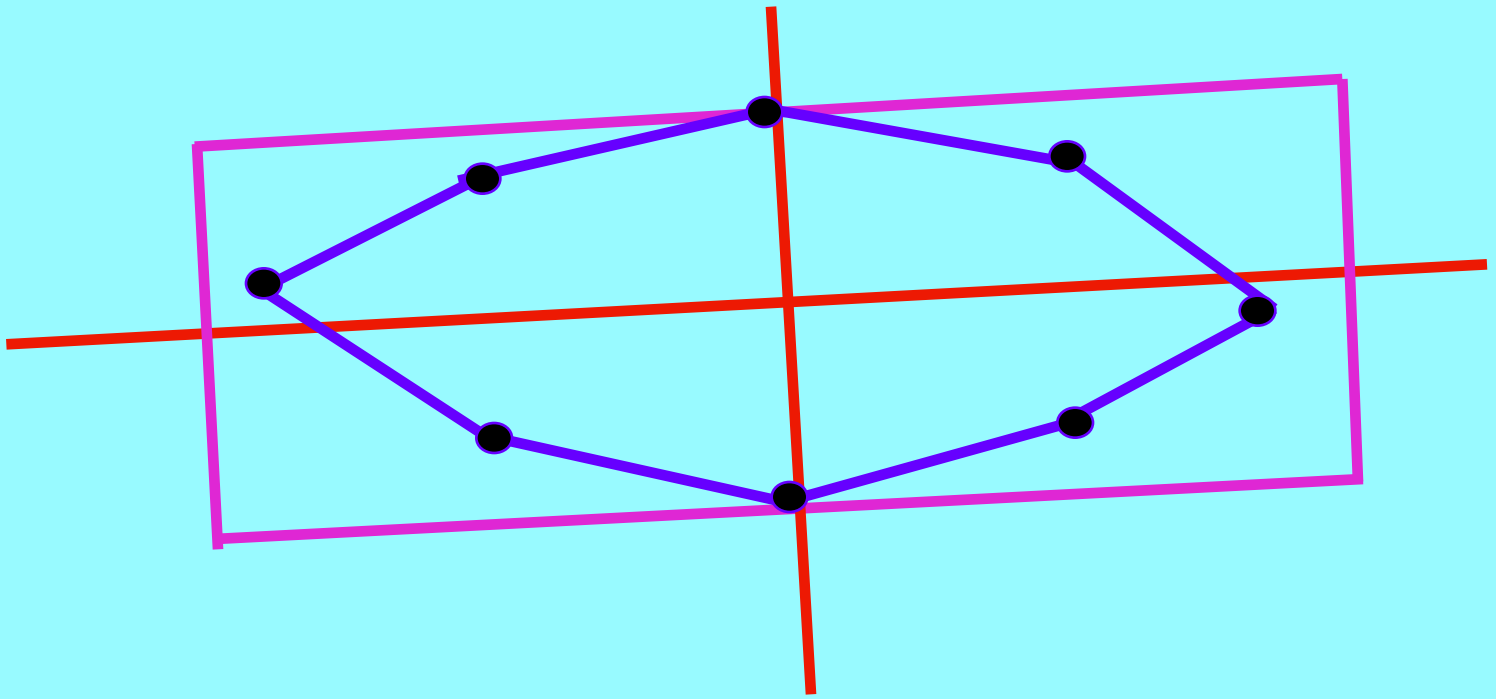
Building an OBB Tree

Fix: Compute facets of convex hull...



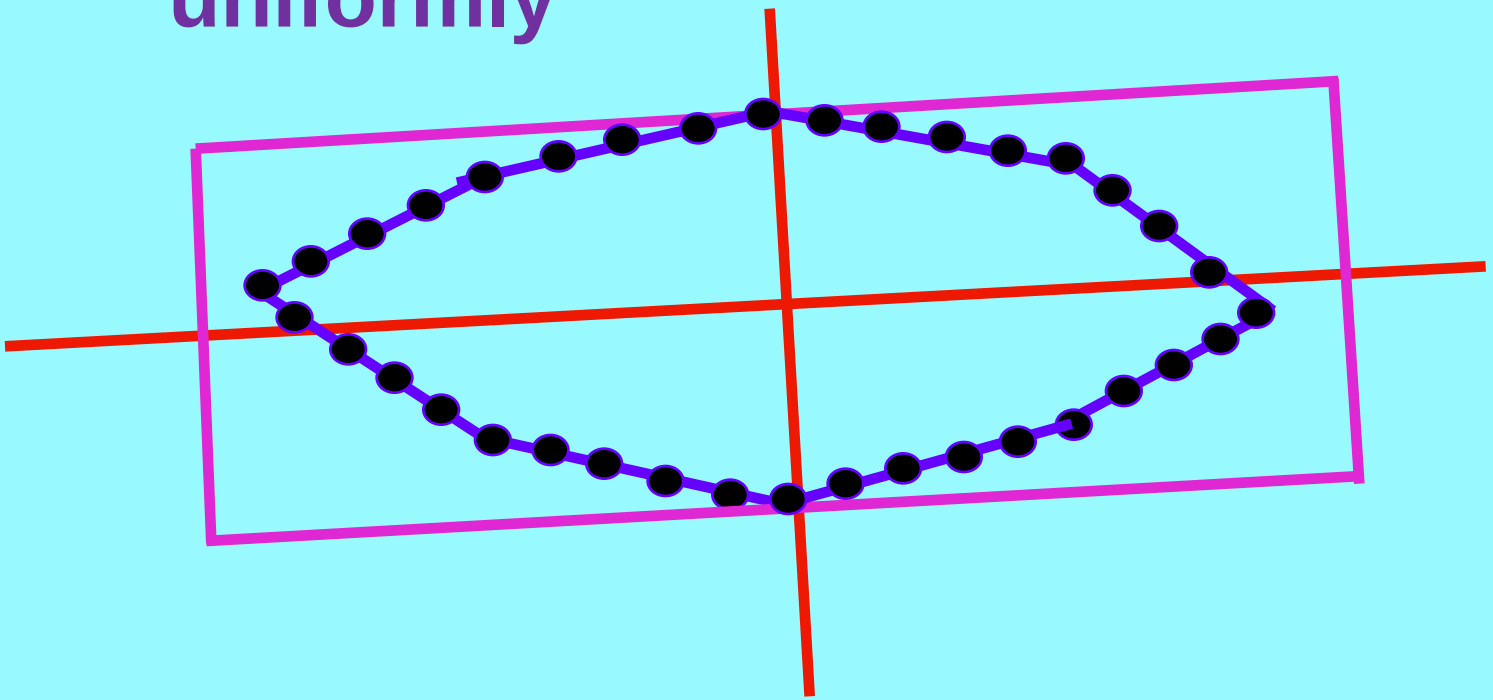
Building an OBB Tree

Better: Integrate over facets



Building an OBB Tree

… in practice, sample them uniformly



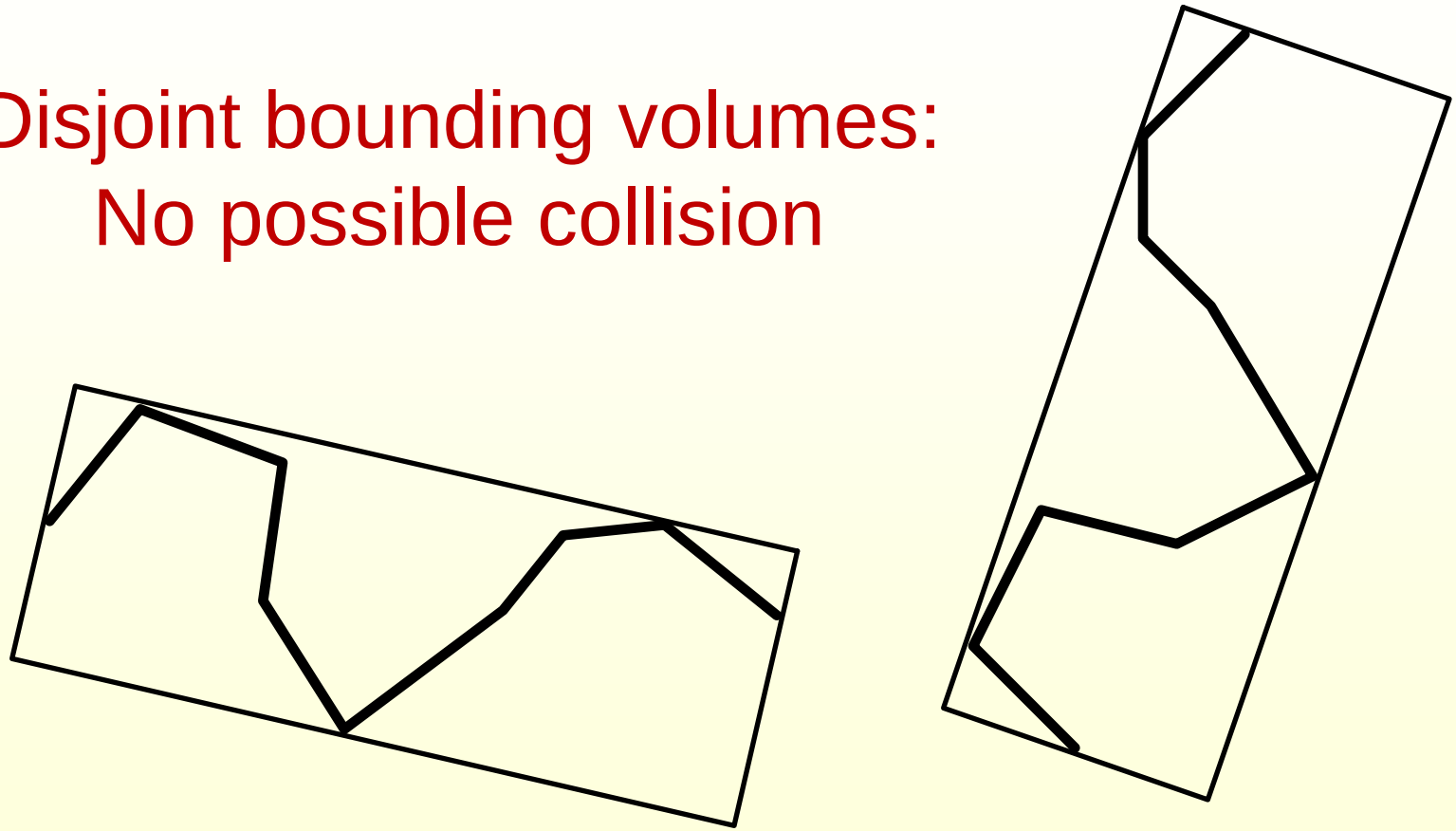
Building an OBB Tree: Summary

OBB Fitting algorithm:

- covariance-based
- use of convex hull
- not foiled by extreme distributions
- O(n log n) fitting time for single BV
- O(n $\log^2$ n) fitting time for entire tree

Tree Traversal

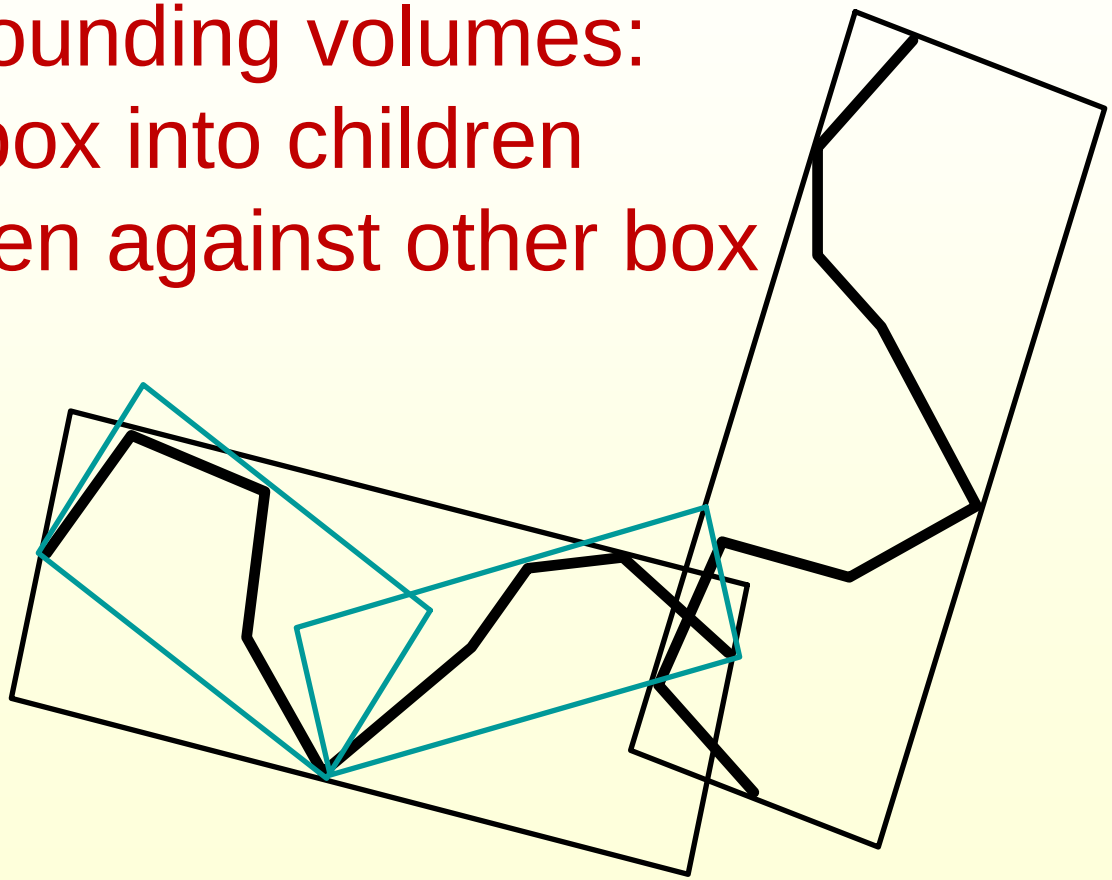
Disjoint bounding volumes:
No possible collision



Tree Traversal

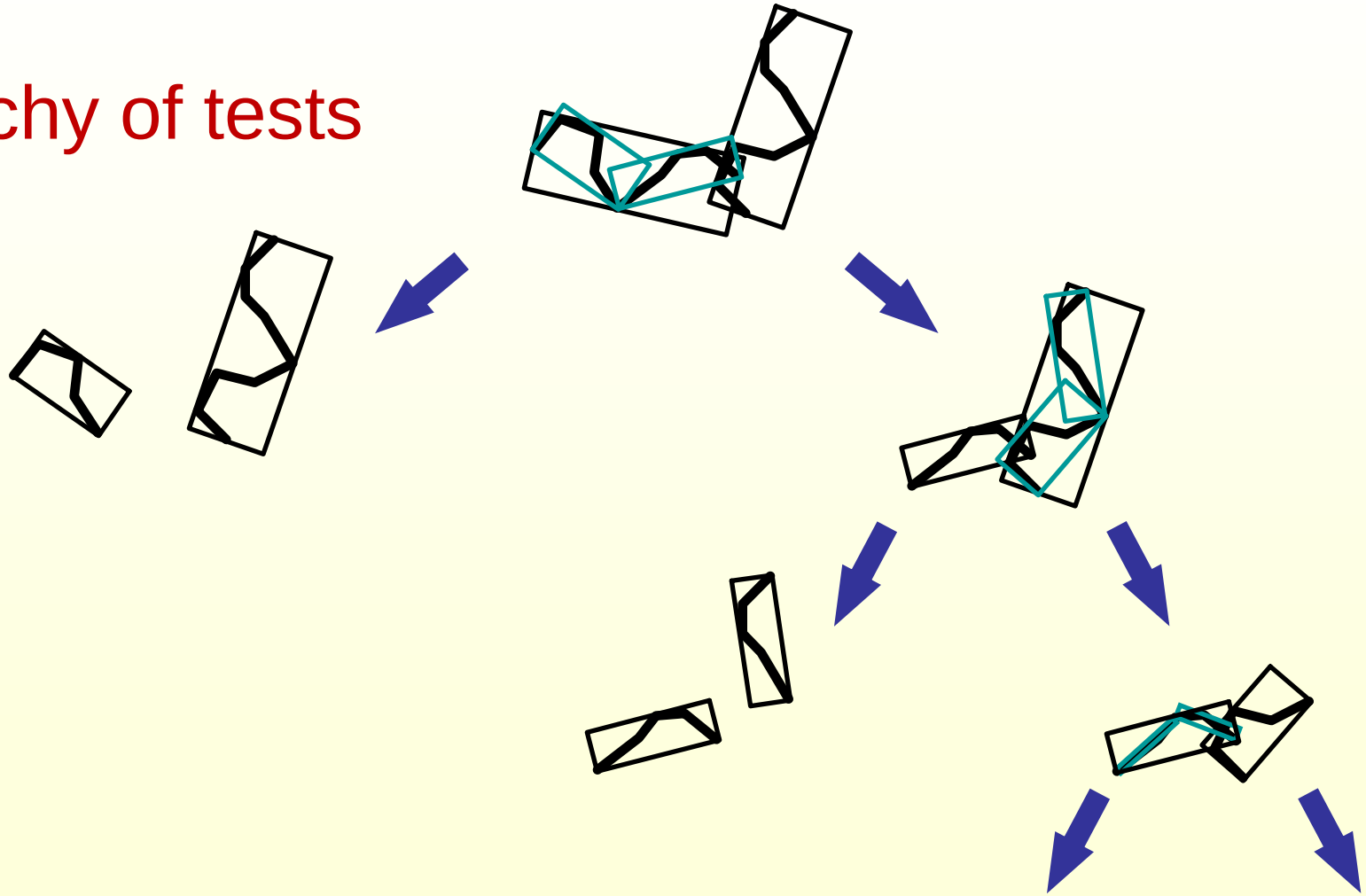
Overlapping bounding volumes:

- split one box into children
- test children against other box

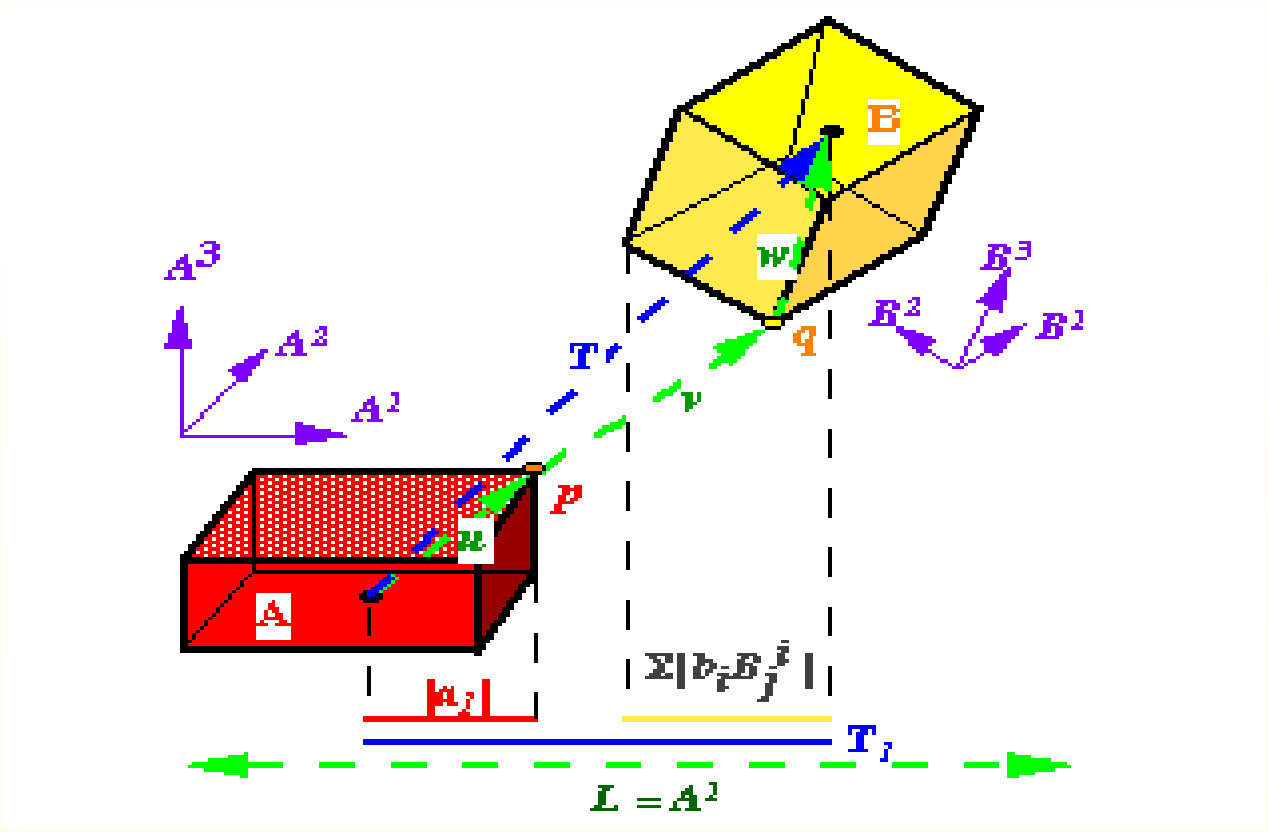


Tree Traversal

Hierarchy of tests



Separating Axis Theorem



- L is a separating axis for OBBs A & B, since A & B generate disjoint intervals under projection onto L

Separating Axis Theorem

Two polytopes A and B are disjoint iff there exists a separating axis which is:

perpendicular to a face from either

or

perpedicular to an edge from each

Implications of Theorem

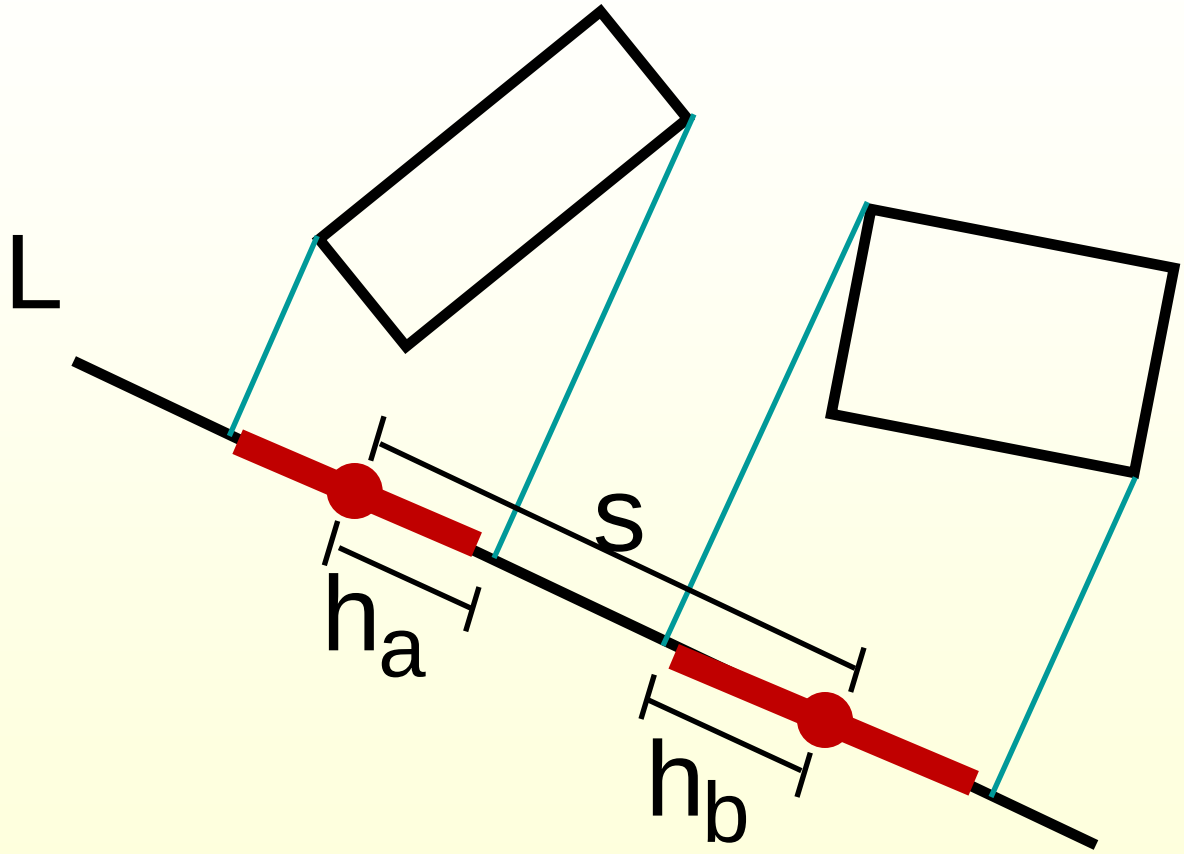
Given two generic polytopes, each with E edges and F faces, number of candidate axes to test is:

$$2F + E^2$$

OBBs have only E = 3 distinct edge directions, and only F = 3 distinct face normals. OBBs need at most 15 axis tests.

Because edge directions and normals each form orthogonal frames, the axis tests are rather simple.

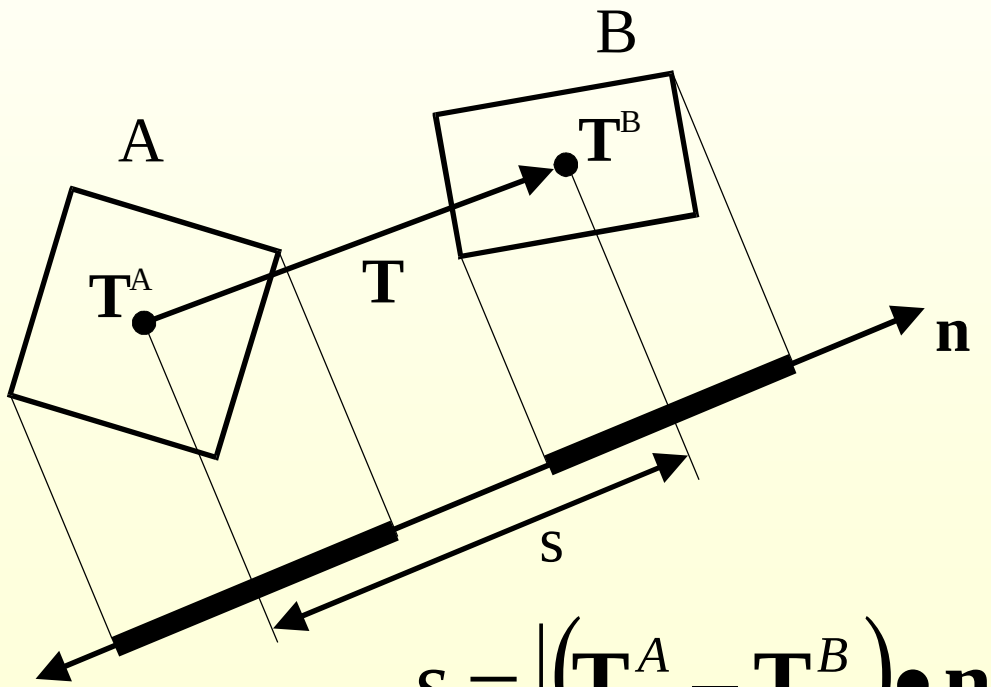
OBB Overlap Test: An Axis Test



L is a separating axis iff: $s > h_a + h_b$

OBB Overlap Test: Axis Test Details

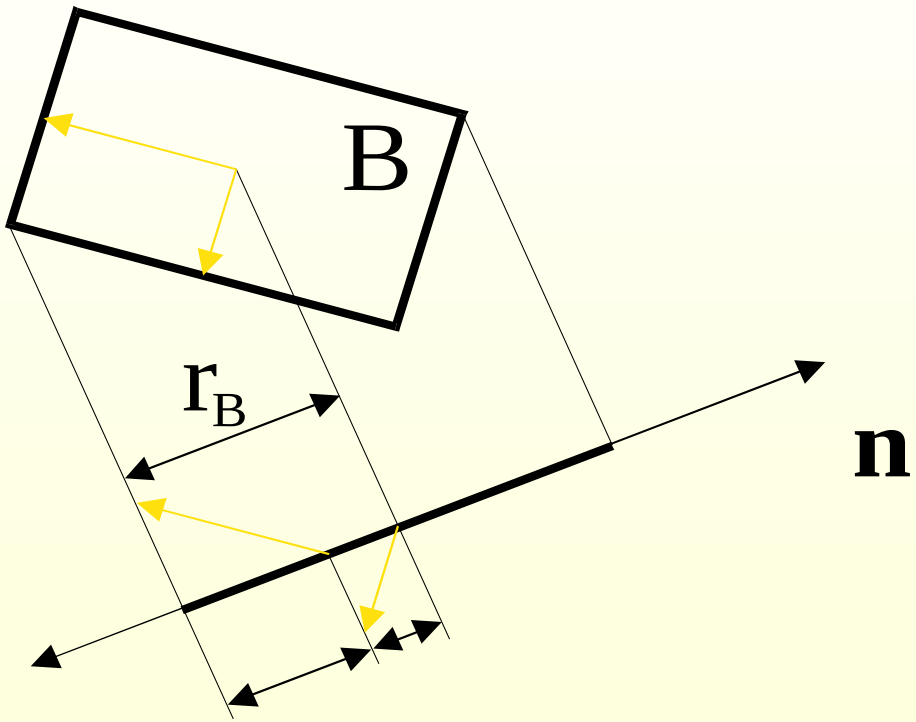
Box centers project to interval midpoints, so midpoint separation is length of vector **T**'s image.



$$s = \left| \left(\mathbf{T}^A - \mathbf{T}^B \right) \bullet \mathbf{n} \right|$$

OBB Overlap Test: Axis Test Details

- Half-length of interval is sum of box axis images.



$$r_B = b_1\left|\mathbf{R}_1^B \cdot \mathbf{n}\right| + b_2\left|\mathbf{R}_2^B \cdot \mathbf{n}\right| + b_3\left|\mathbf{R}_3^B \cdot \mathbf{n}\right|$$

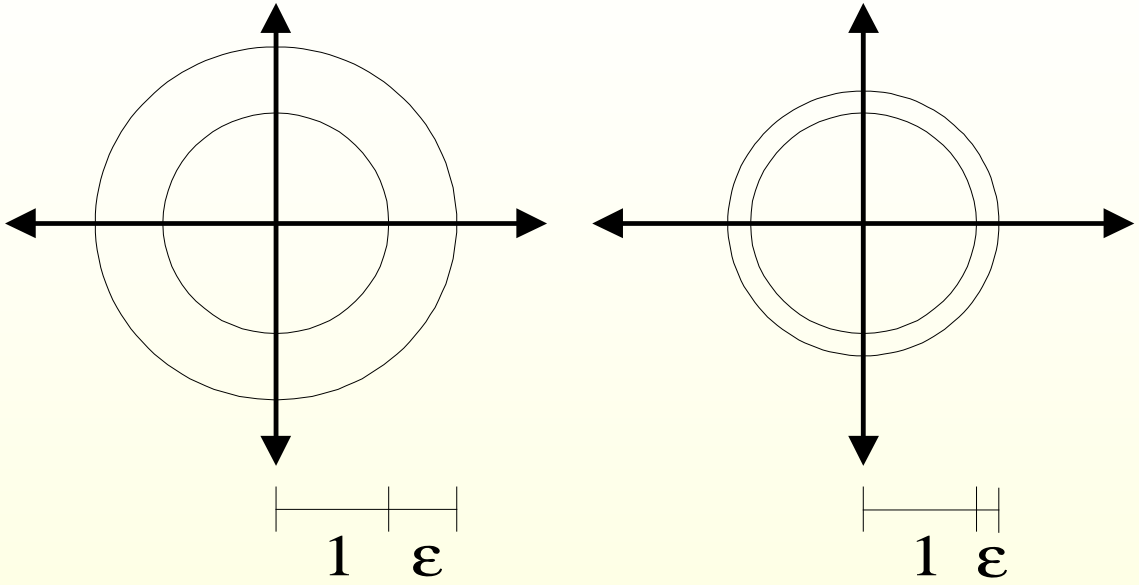
OBB Overlap Test

- Strengths of this overlap test:
  - 89 to 252 arithmetic operations per box overlap test
  - Simple guard against arithmetic error
  - No special cases for parallel/coincident faces, edges, or vertices
  - No special cases for degenerate boxes
  - No conditioning problems
  - Good candidate for micro-coding

Performance

Tens of thousands of collision checks per second for two three-dimensional objects each described by 500,000 triangles, on a 2-GHz PC

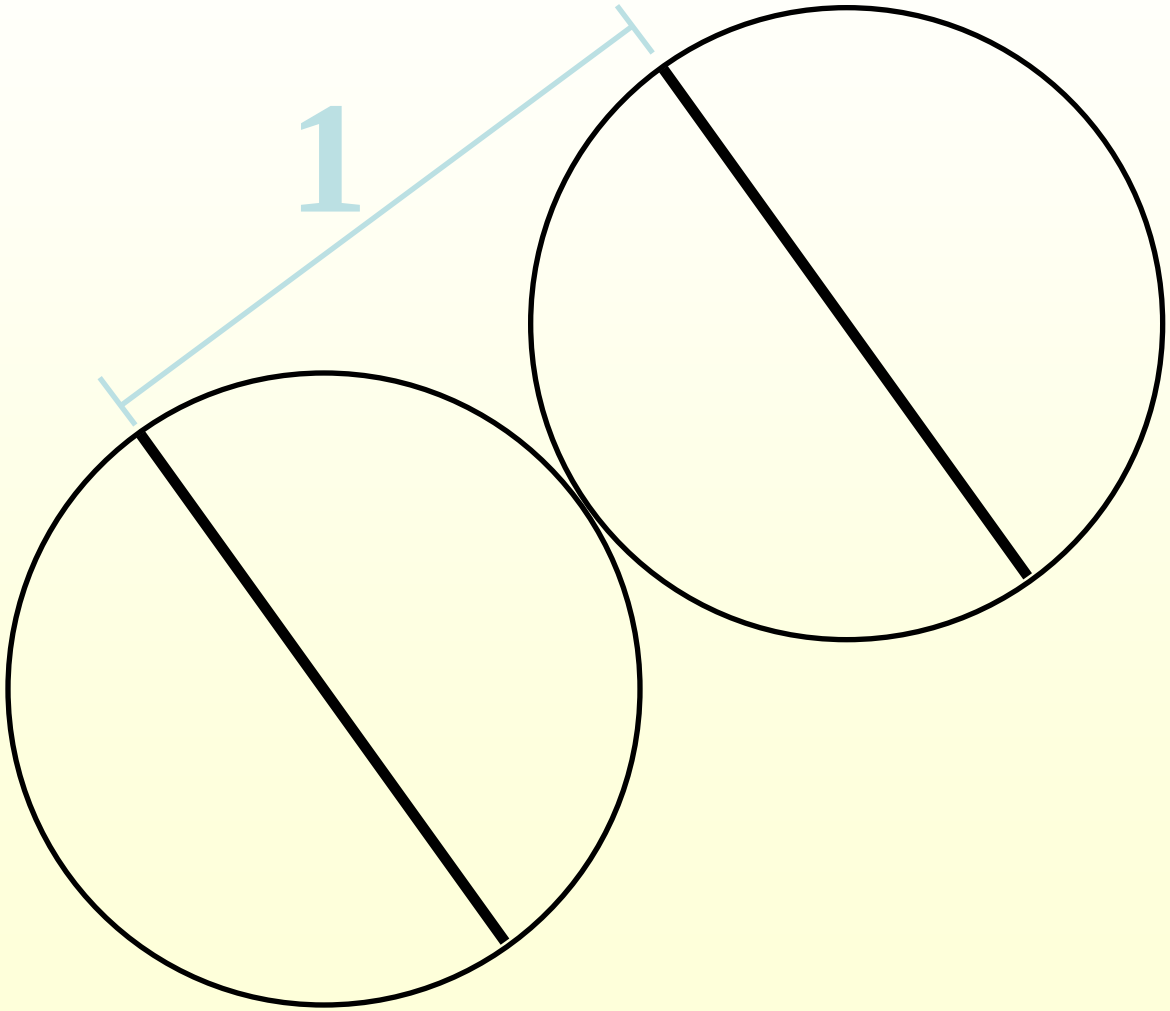
BVH Evaluation: Parallel Close Proximity



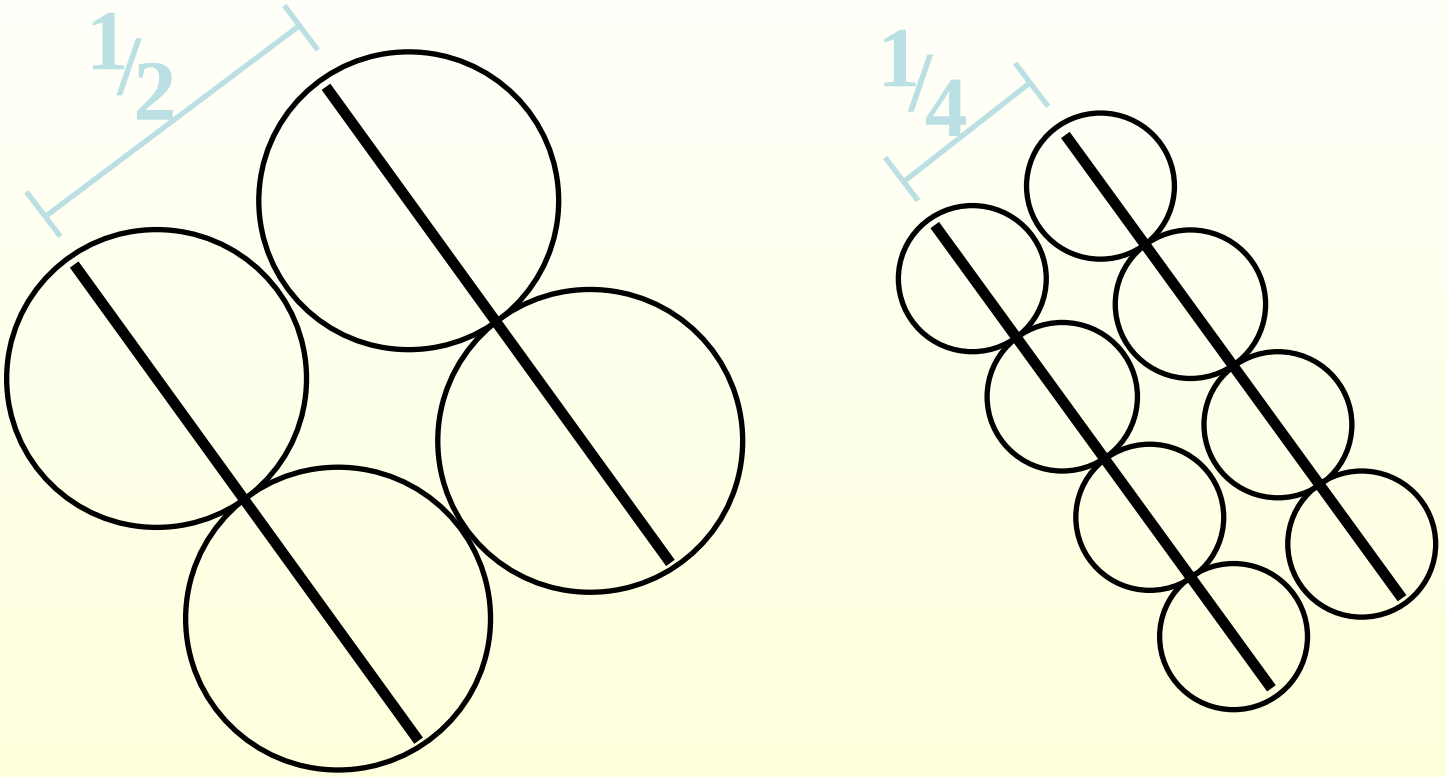
Two models are in *parallel close proximity* when every point on each model is a given fixed distance (e) from the other model.

Q: How does the number of BV tests increase as the gap size decreases?

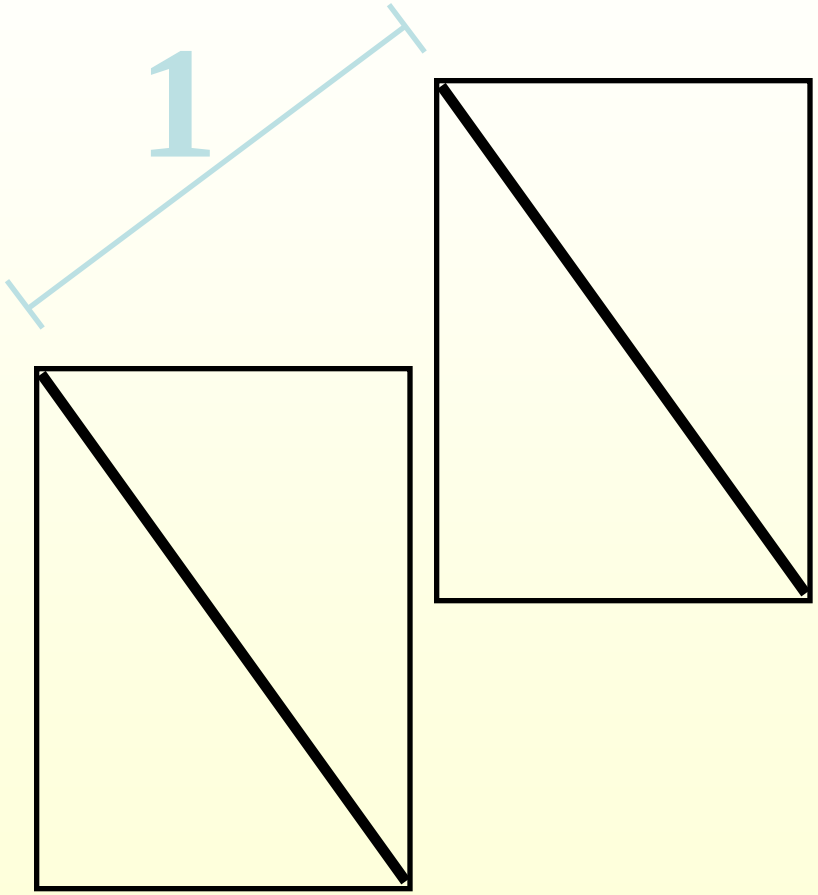
Parallel Close Proximity: Convergence



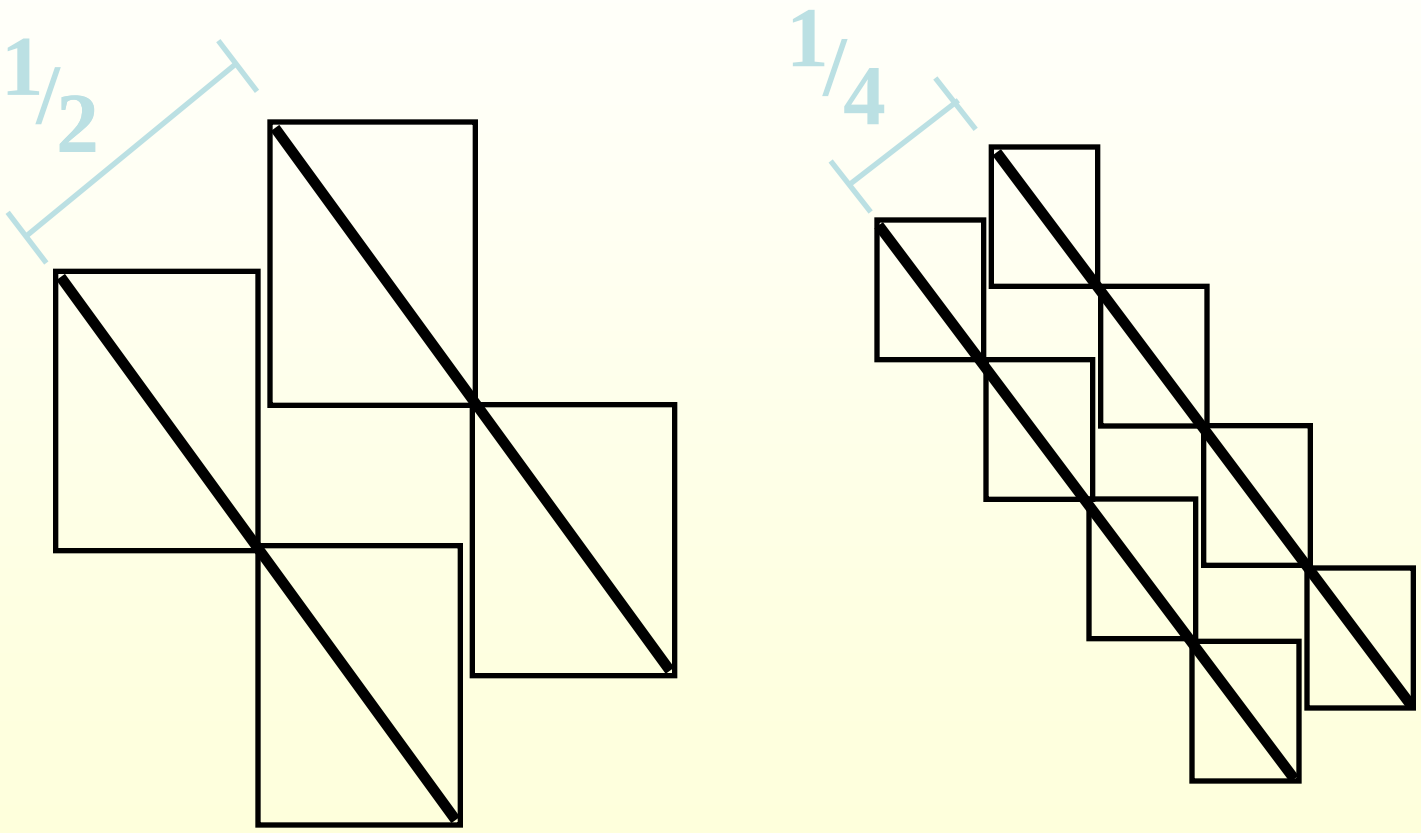
Parallel Close Proximity: Convergence



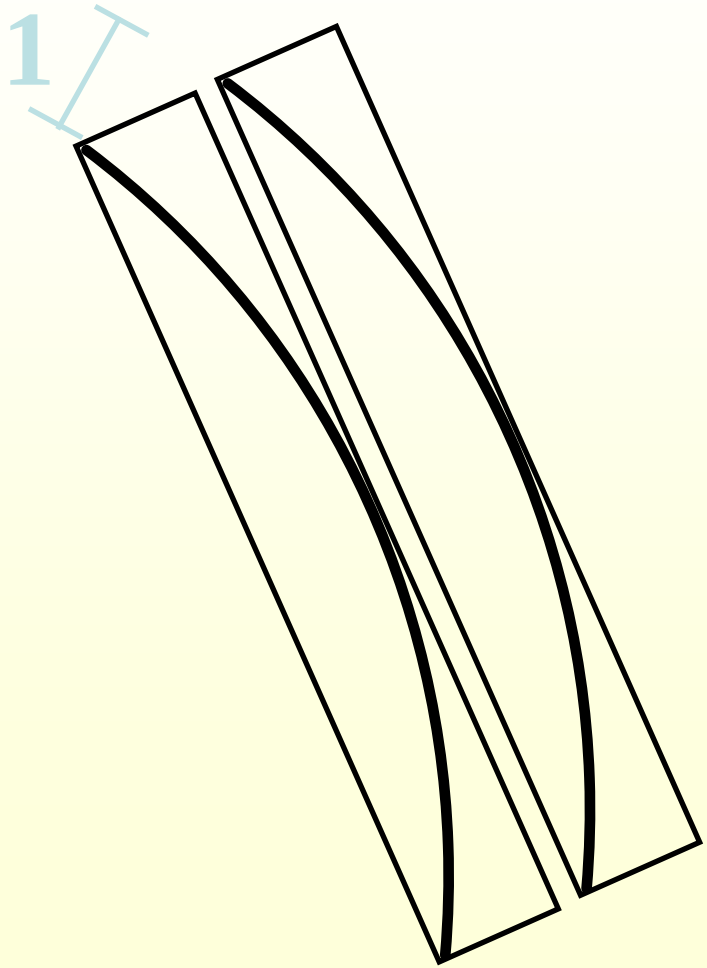
Parallel Close Proximity: Convergence



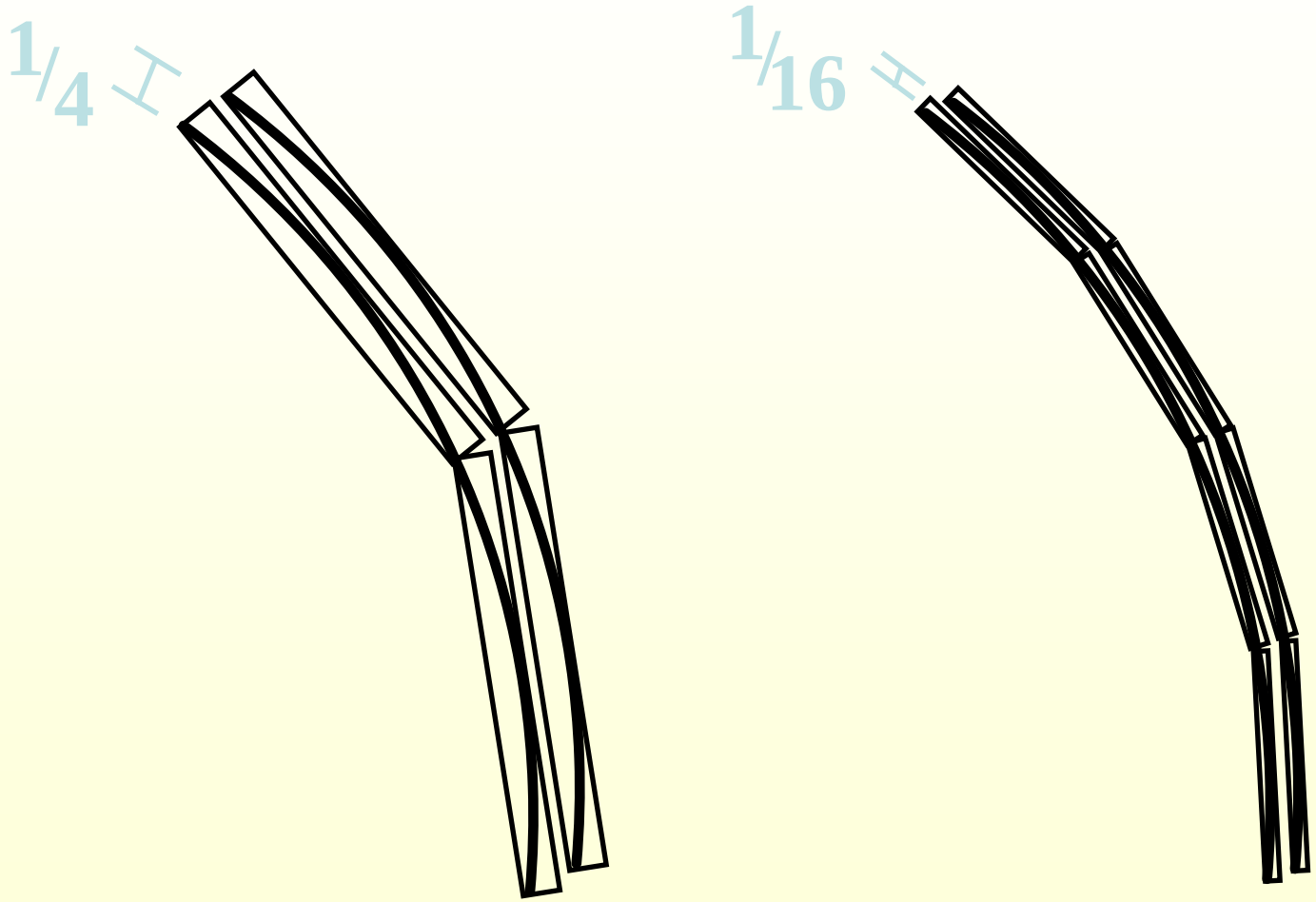
Parallel Close Proximity: Convergence



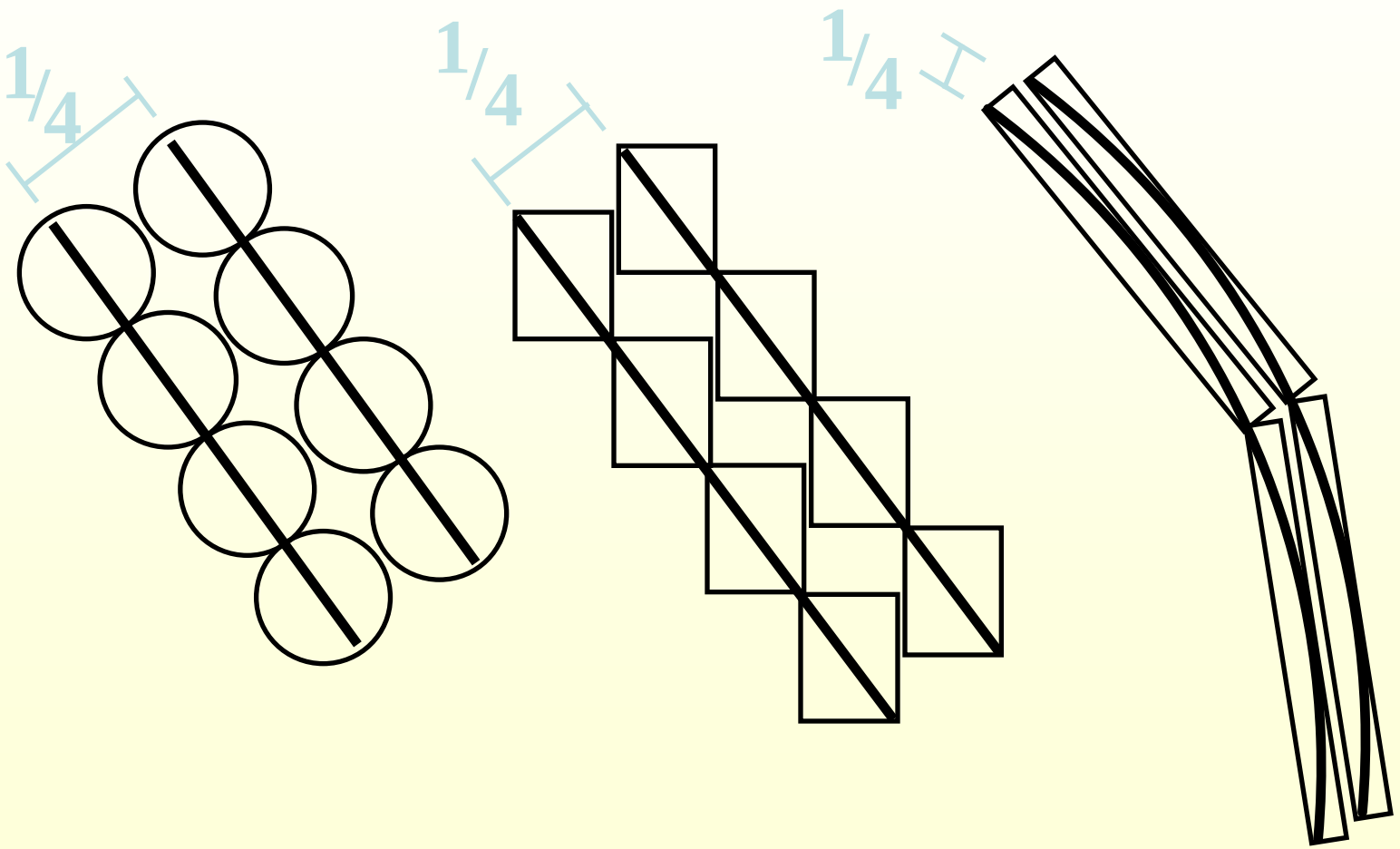
Parallel Close Proximity: Convergence



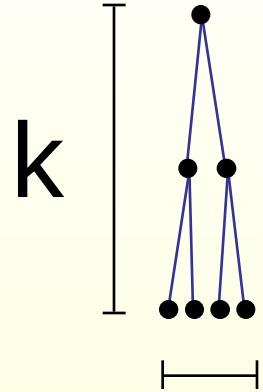
Parallel Close Proximity: Convergence



Parallel Close Proximity: Convergence

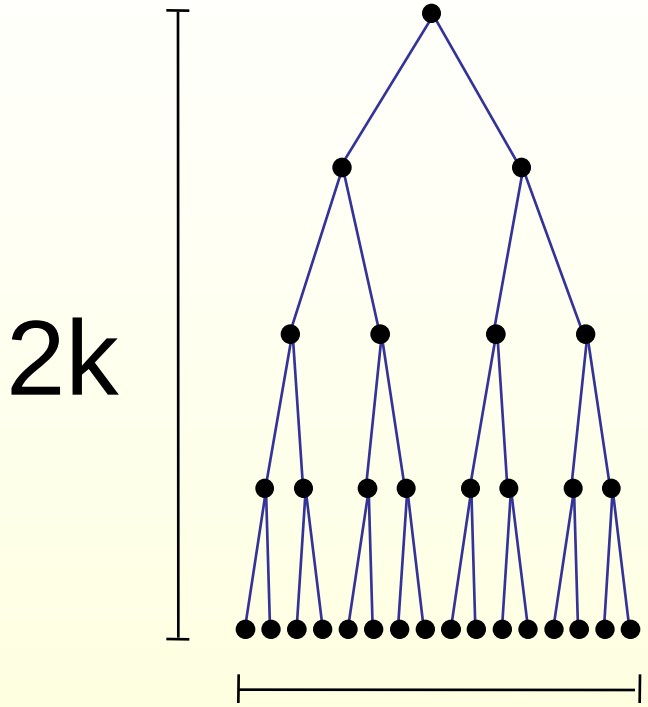


Performance: Overlap Tests



O(n)

OBBs



$O(n^2)$

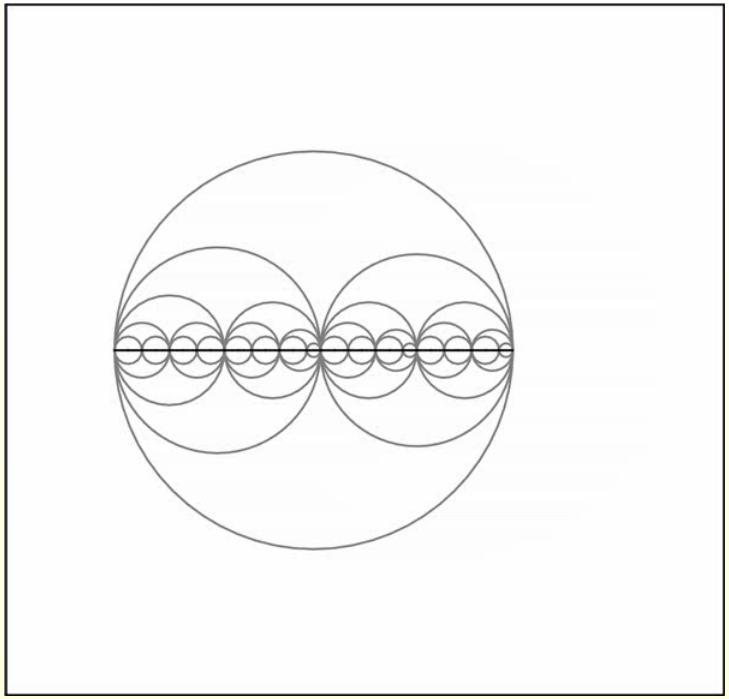
Spheres & AABBs

Additional Comments

- Bottom up construction of BVH is also quite common
- Collision detection is intimately connected with distance computation – most methods for intersection testing aim to establish a postive lower bound on the distance between two objects
- Distance computation is especially easy for convex objects

Deformable Objects

- This is a difficult case
- Bounding volume hierarchies can be adapted, but the recomputation is expensive



The End