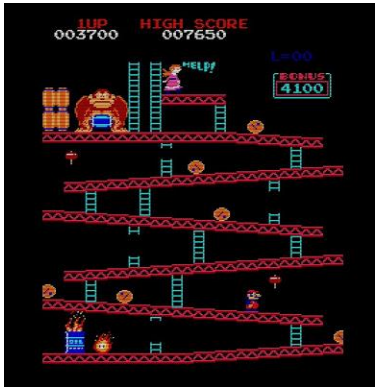
Collision Detection

CS434

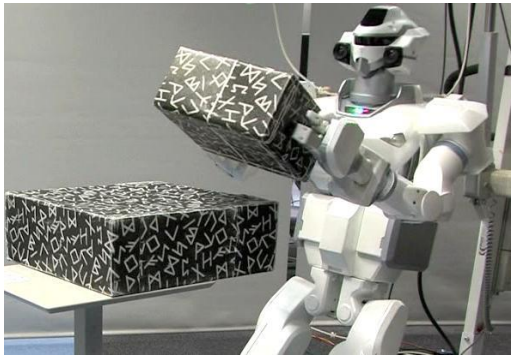
Daniel G. Aliaga
Department of Computer Science
Purdue University

Some Applications

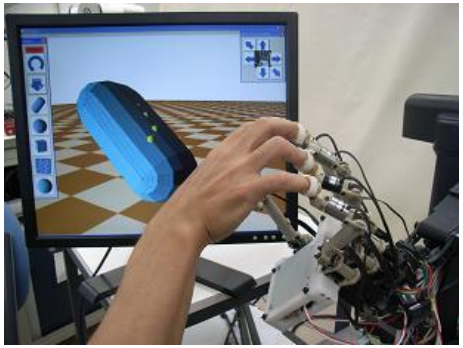
- Animations/games



- Robotic planning



- Haptics



Some collision detection approaches…

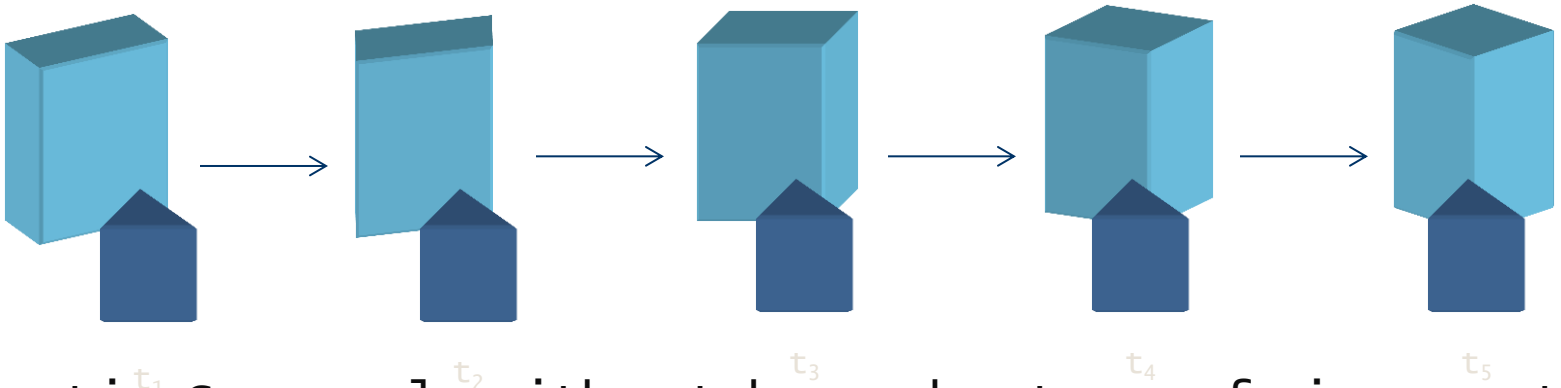
- Minkowski Sum 2D collisions
  - http://www.slideshare.net/crowdscontrol/minkowski-sum-on-2d-geometry
- http://gamma.cs.unc.edu/research/collision/
- "Efficient Collision Detection for Animation and Robotics", Lin's PhD thesis, 1993
  - Slides based on those of Adit Koolwal and Benson Limketkai

an overview of the lin-canny algorithm

The Lin-Canny algorithm:

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

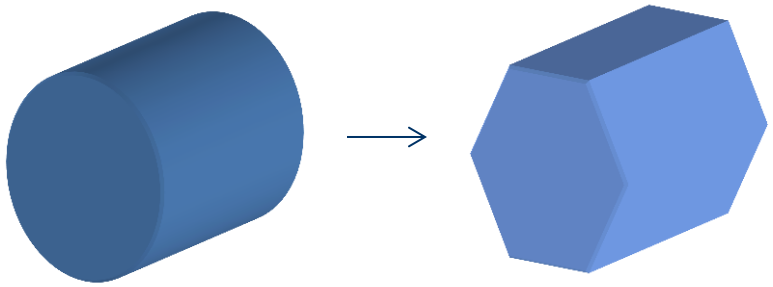
At each timestep, checks if the current closest feature pair is still the closest. If not, it finds the new closest pair.



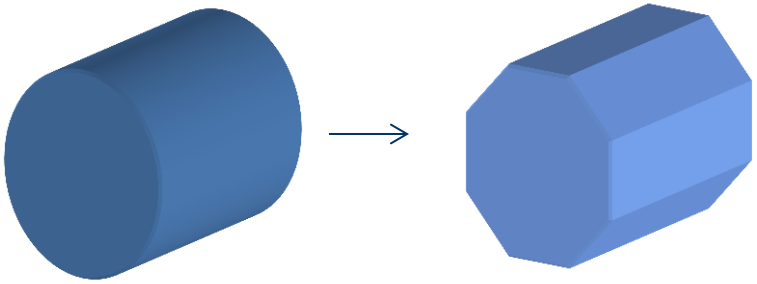
The Lin-Canny algorithm takes advantage of incremental motion because closest features change infrequently.

implementating lin-canny in 4 easy steps…

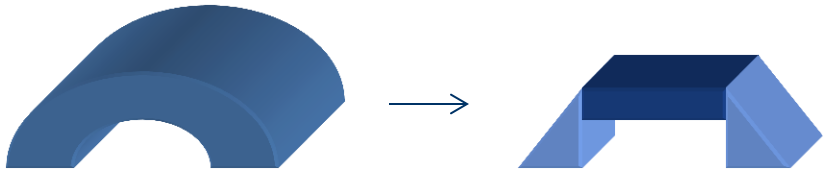
1. We first represent each object as a convex polyhedron, or a union of convex polyhedra.



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



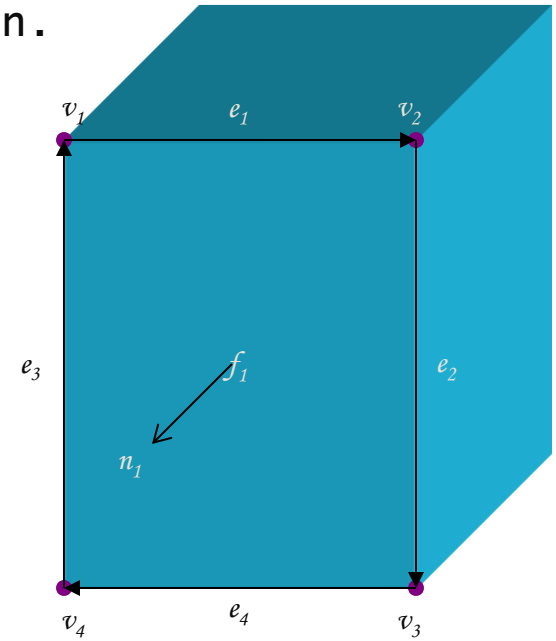
For non-convex objects we rely on subdivision into convex pieces, which can take up to quadratic time.



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

2. For each object we calculate the fields for each of its faces, edges, vertices, positions, and orientations.

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

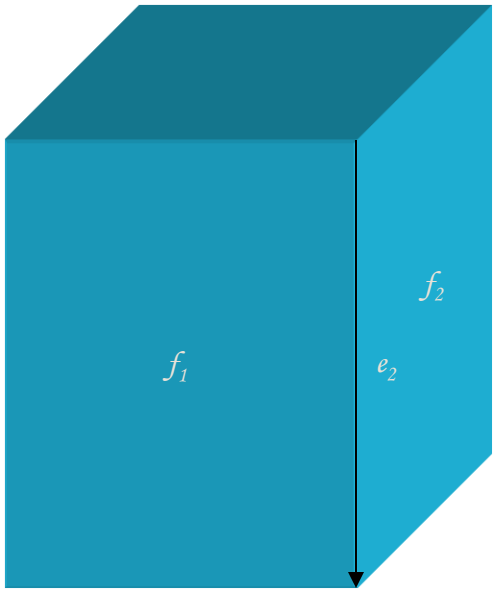


A FACE contains a list of VERTICES v_i which lie on its boundaries, and a list of EDGES e_i which bound the face.

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

2. For each object we calculate the fields for each of its faces, edges, vertices, positions, and orientations.

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



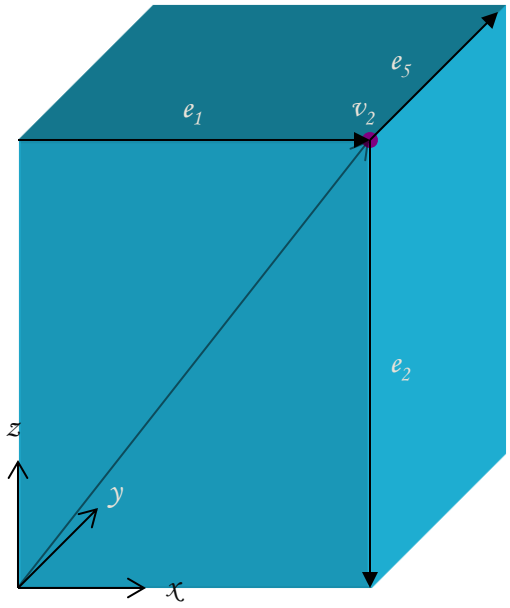
Each **EDGE** e_i is described by its head, tail, left face, and right face.

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 fields for each of its faces, edges, vertices, positions, and orientations.

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 intersect it.

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

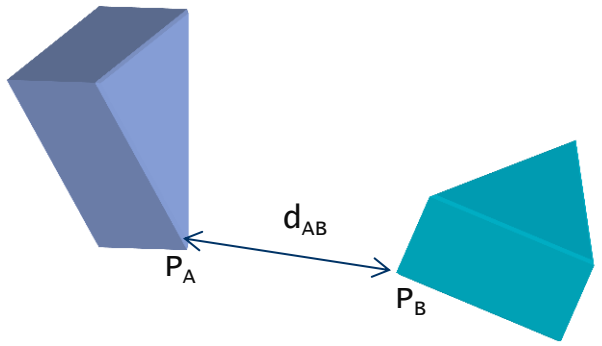
2. For each object we calculate the fields for each of its faces, edges, vertices, positions, and orientations.

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 $feature_A$

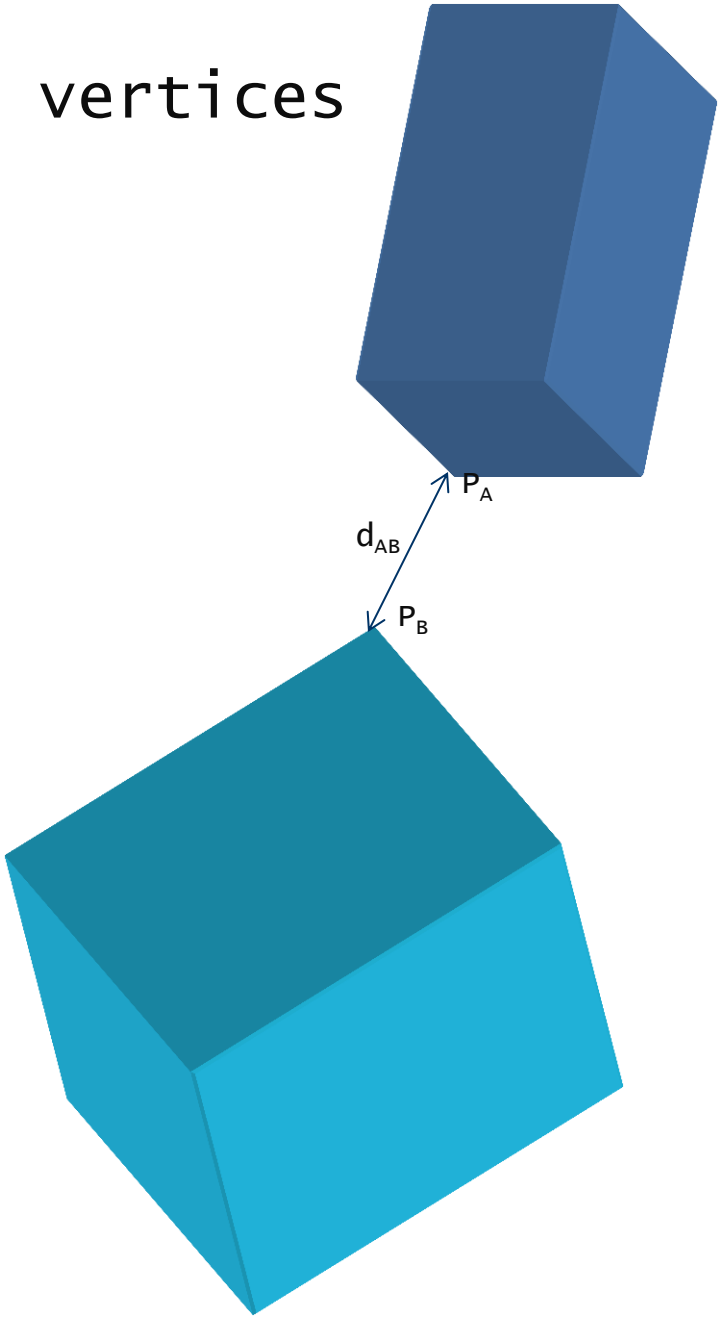
The distance d_{AB} is the Euclidean distance between P_A and P_B



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

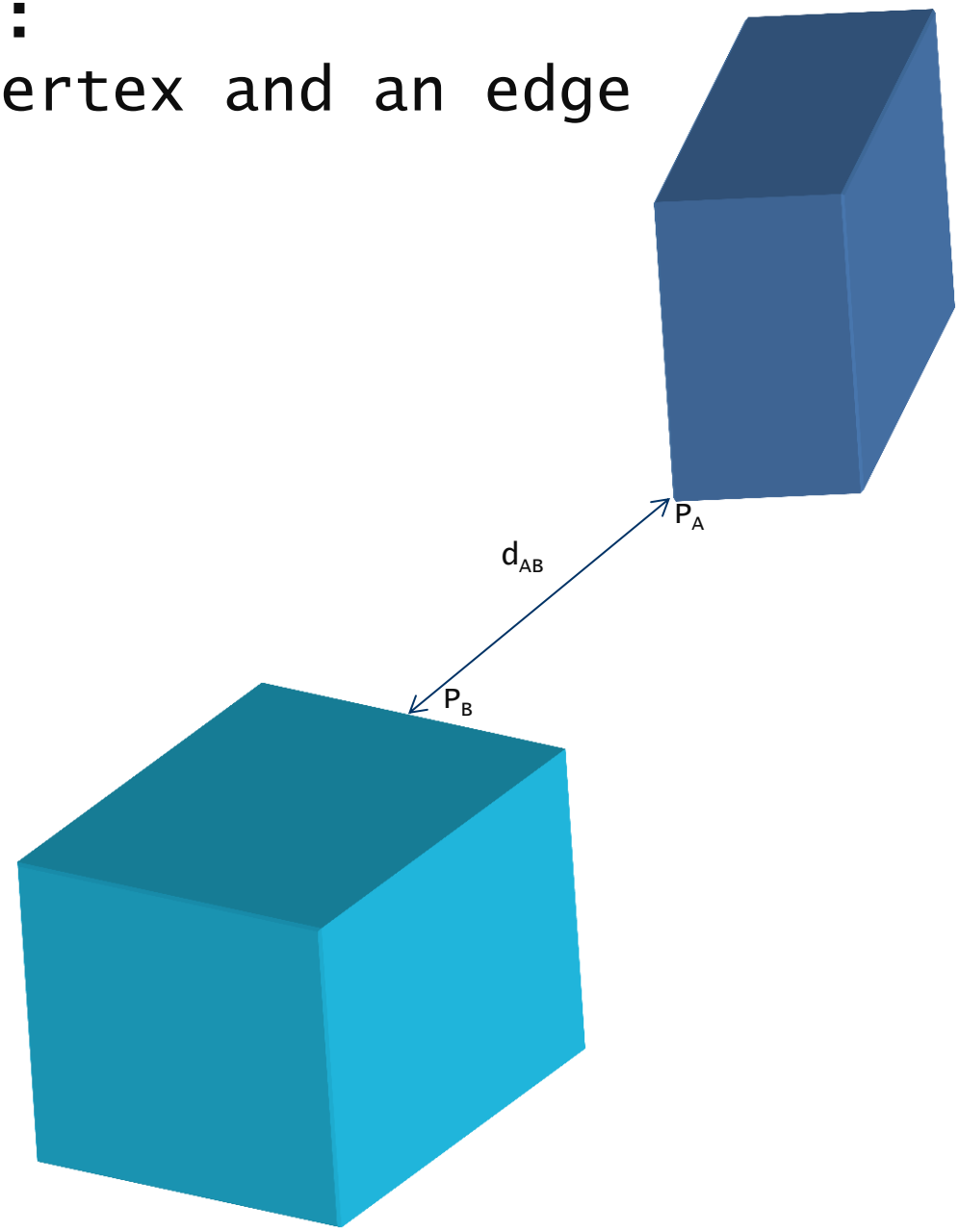
case one:

a pair of vertices

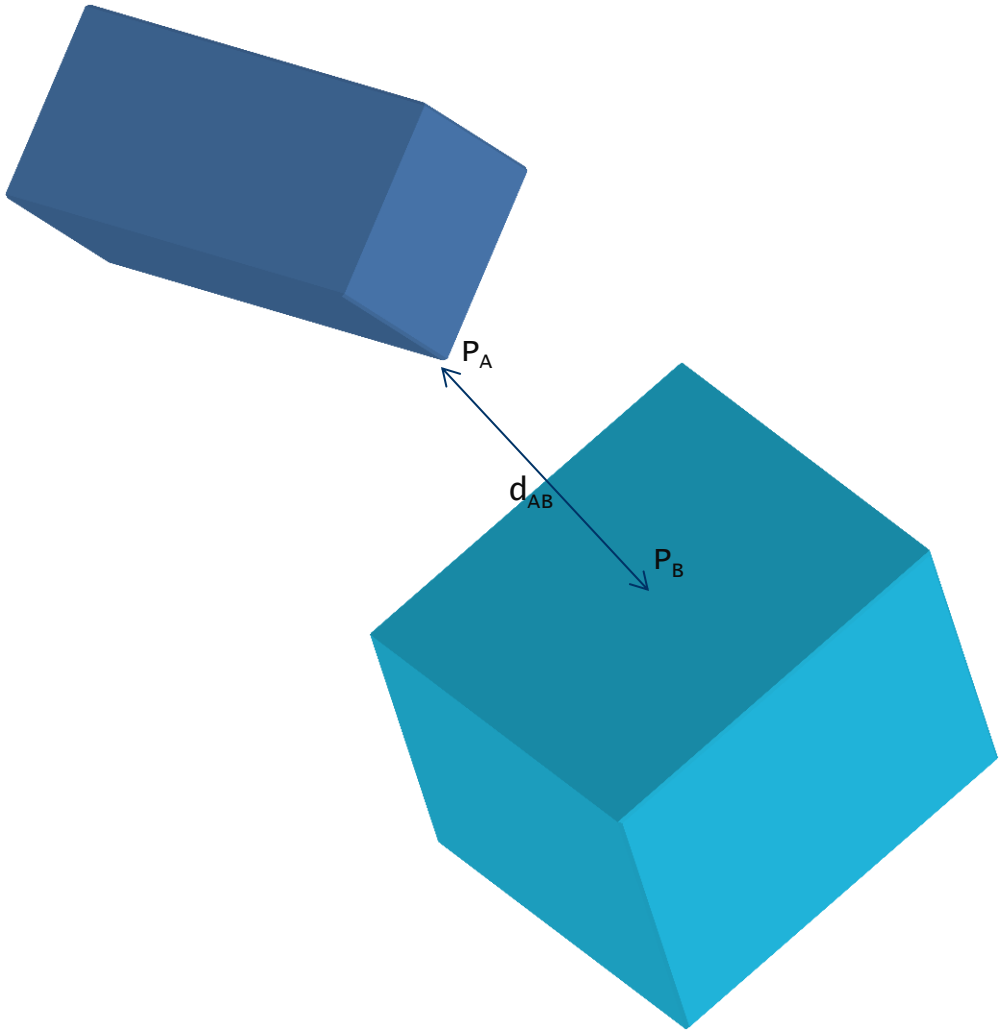


case two:

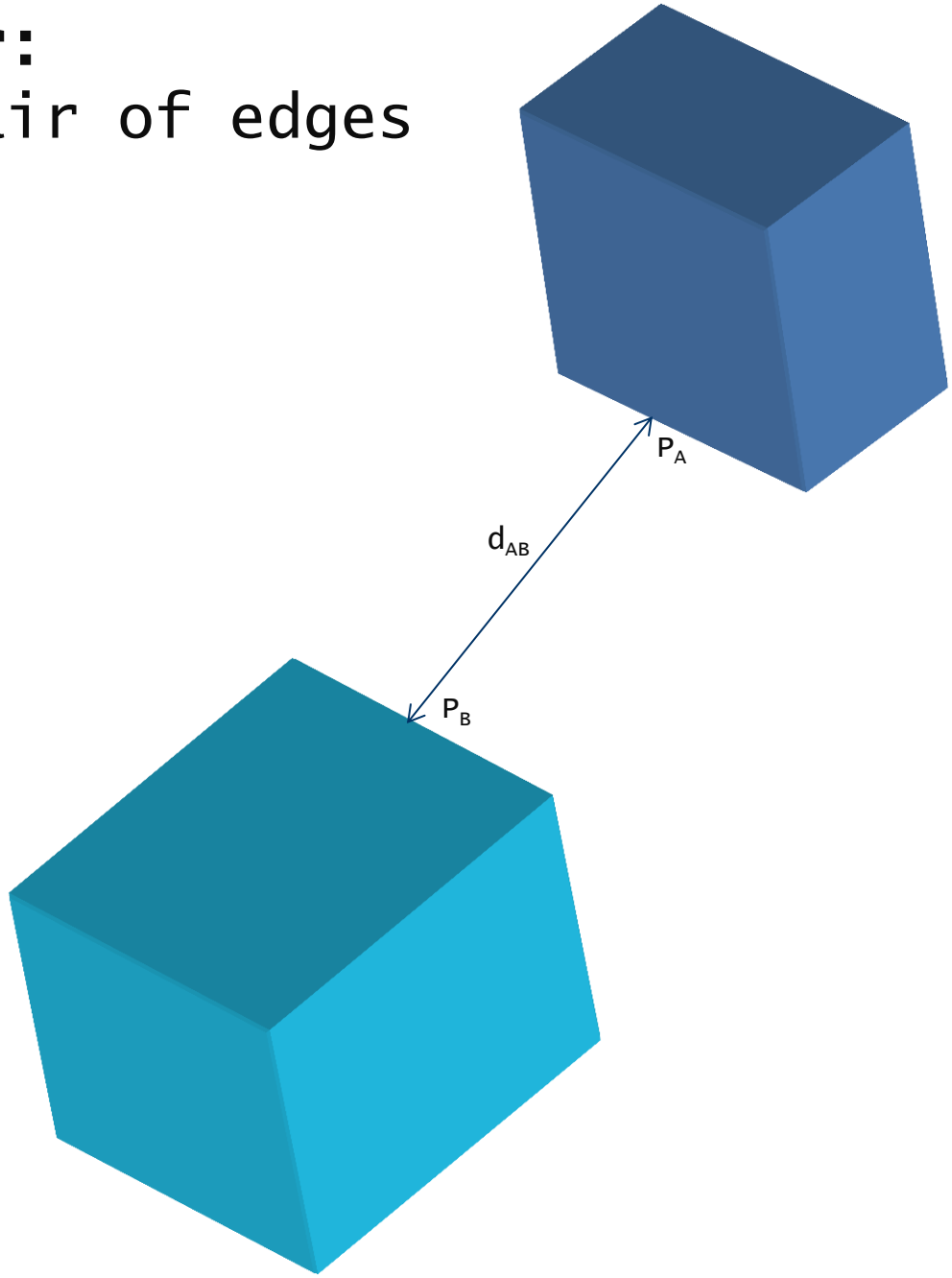
a vertex and an edge



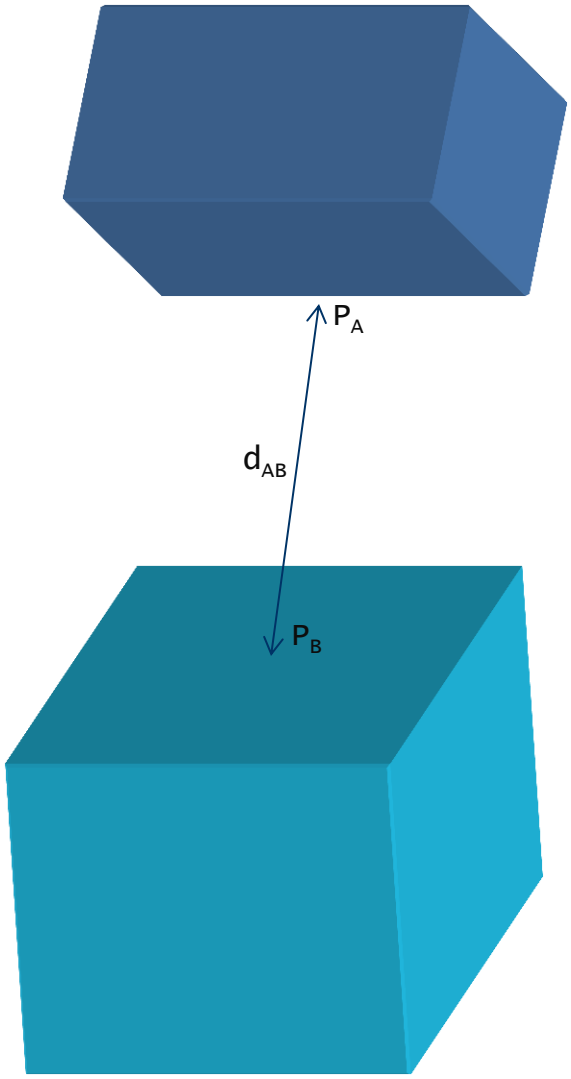
case three:
a vertex and a face



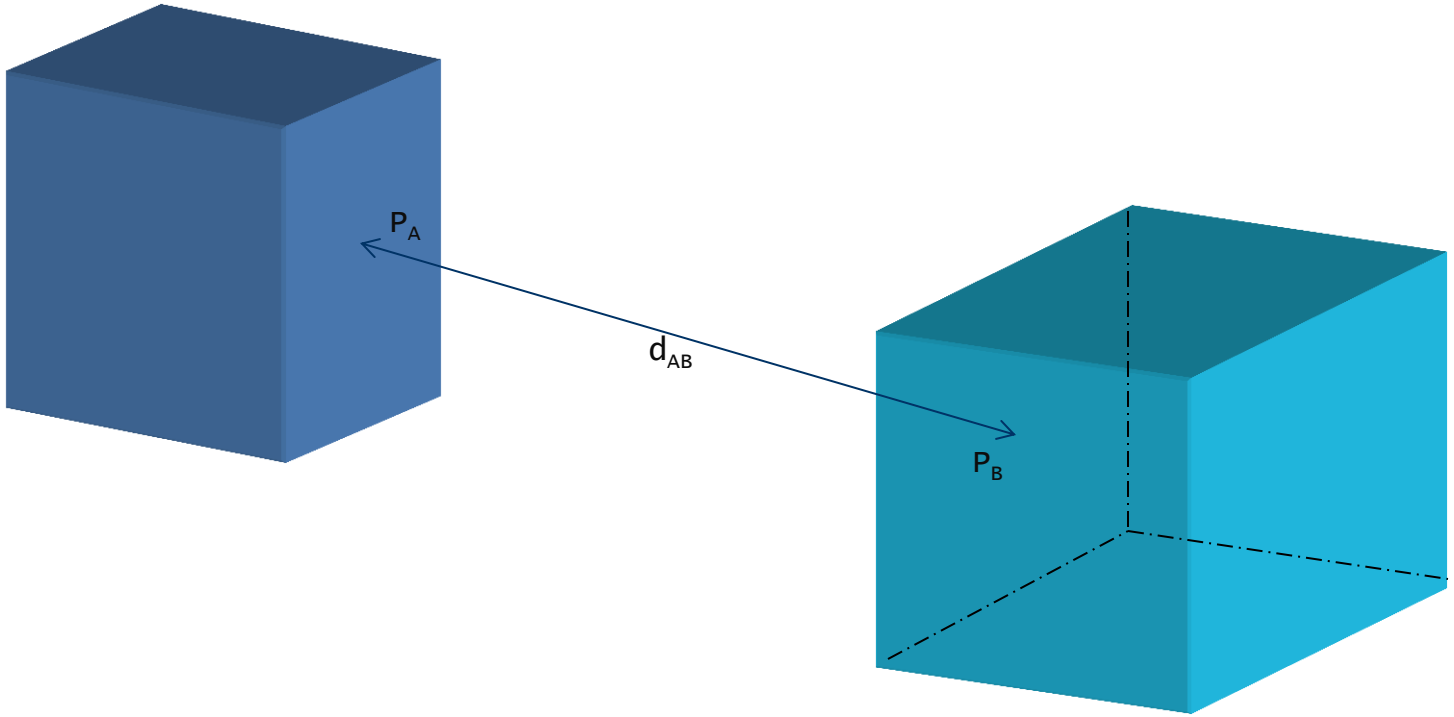
case four:
a pair of edges



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 fields for each of its faces, edges, vertices, positions, and orientations.

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

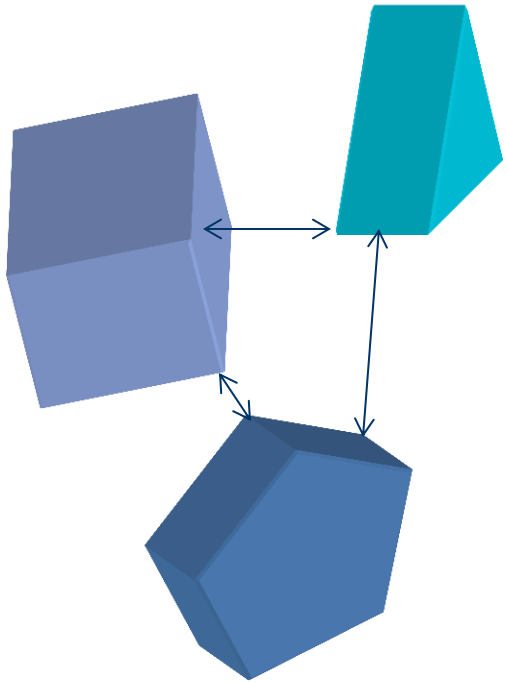
question: how can we tell if we've found a closest feature pair, or if we need to try a new pair?

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:

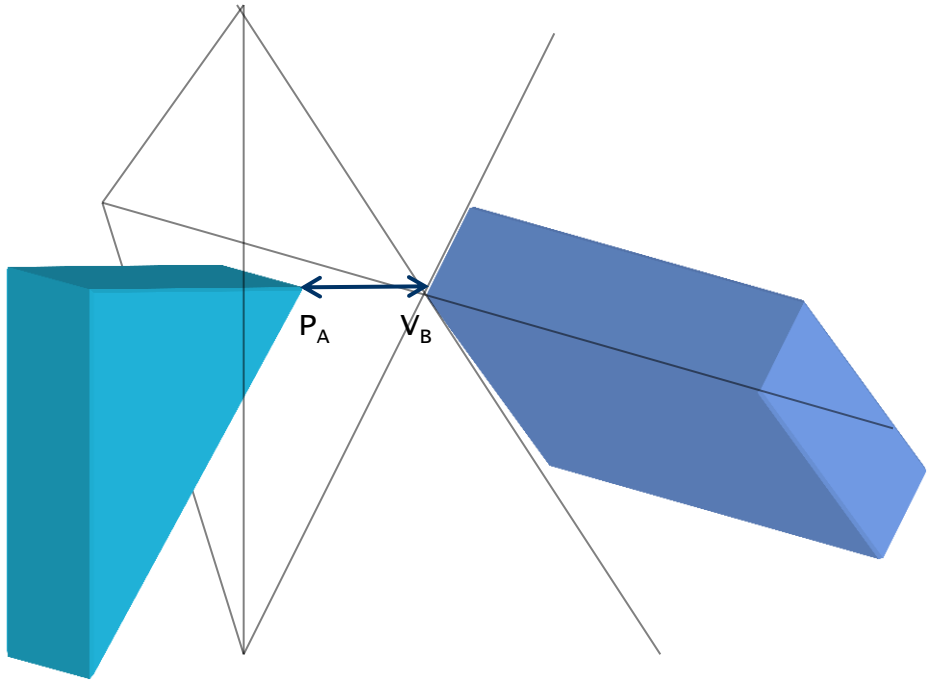
1. **Point-Vertex** (Vertex-Vertex)
2. **Point-Edge** (Vertex-Edge)
3. **Point-Face** (Vertex-Face)

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

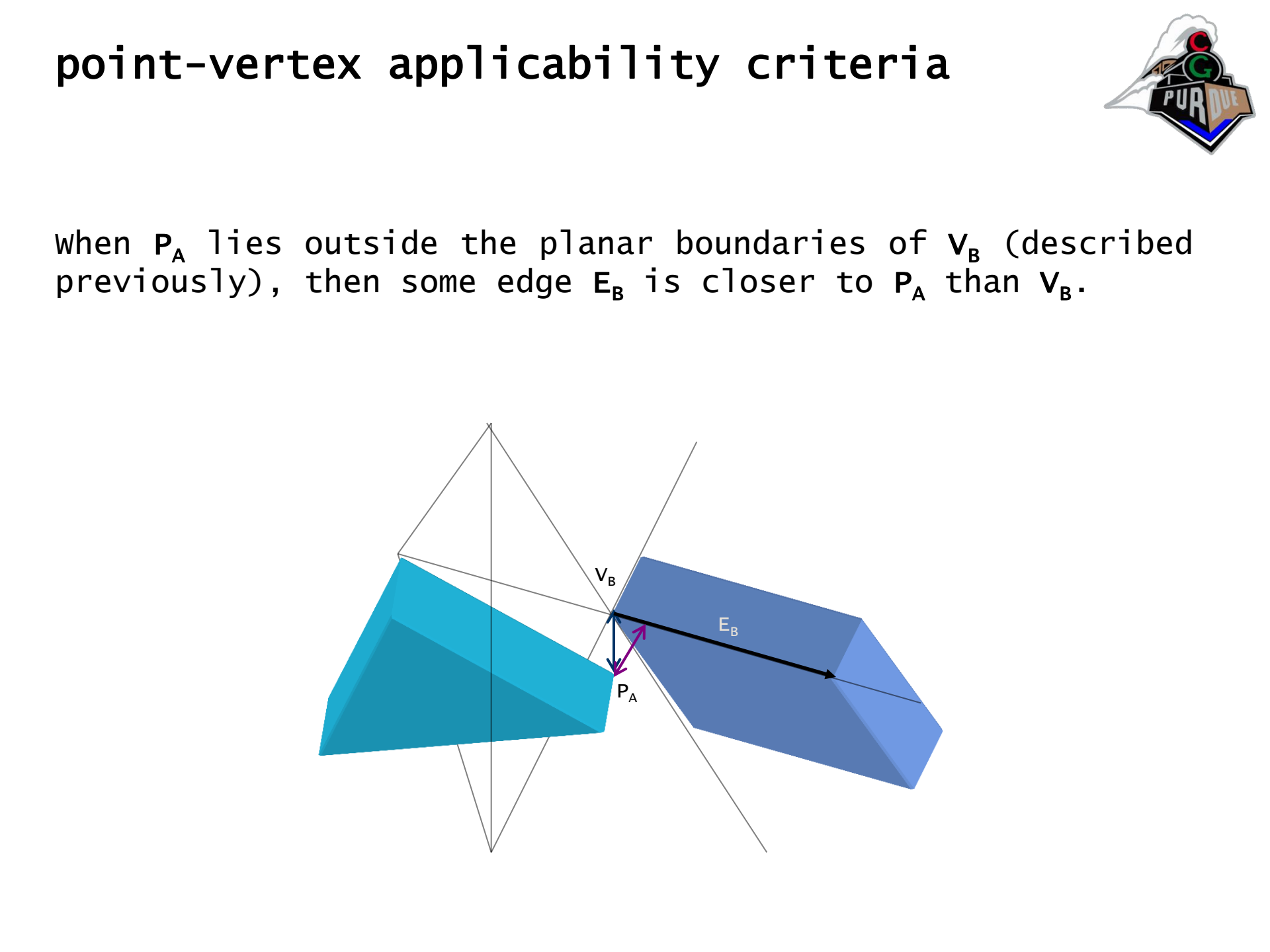


point-vertex applicability criteria

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



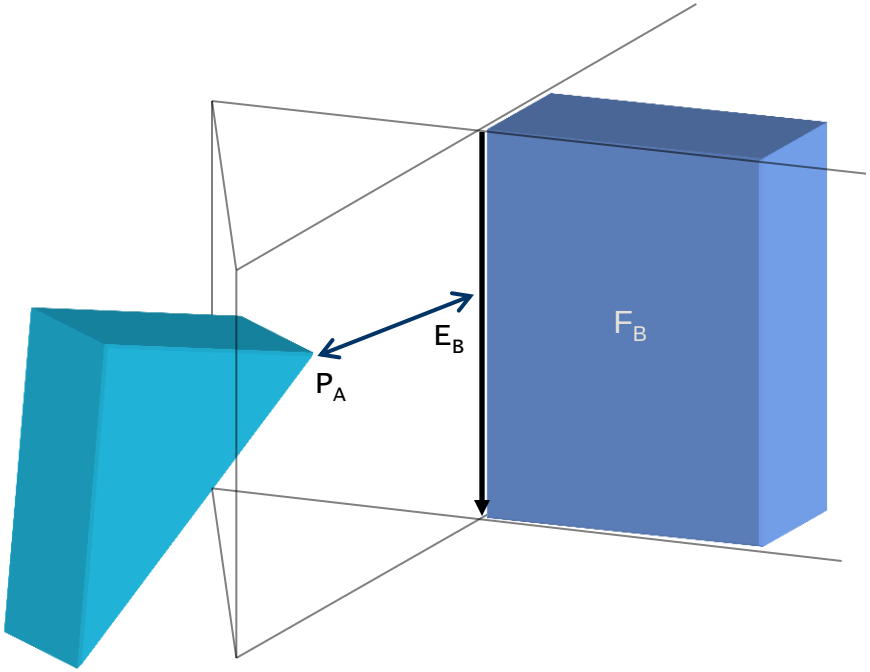
When P_A lies outside the planar boundaries of V_B (described previously), then some edge E_B is closer to P_A than V_B.

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:

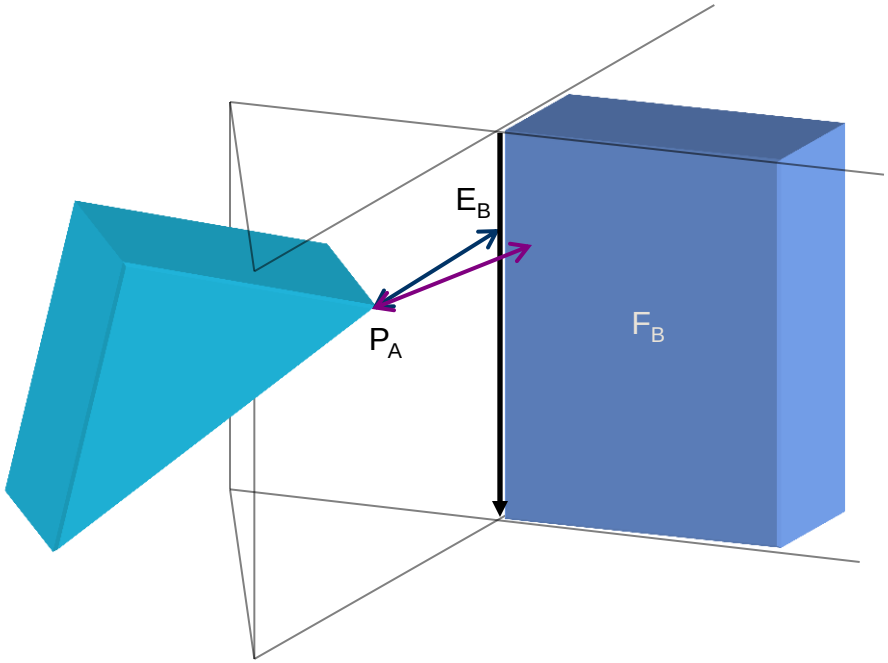
1. The two planes perpendicular to E_B, passing through its head and tail, *and*

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



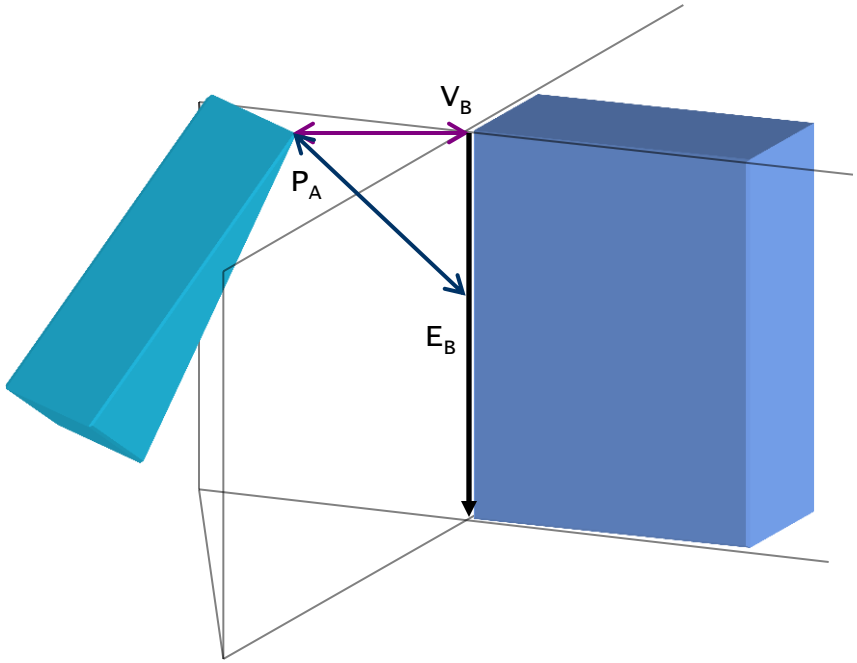
point-edge applicability criteria

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.



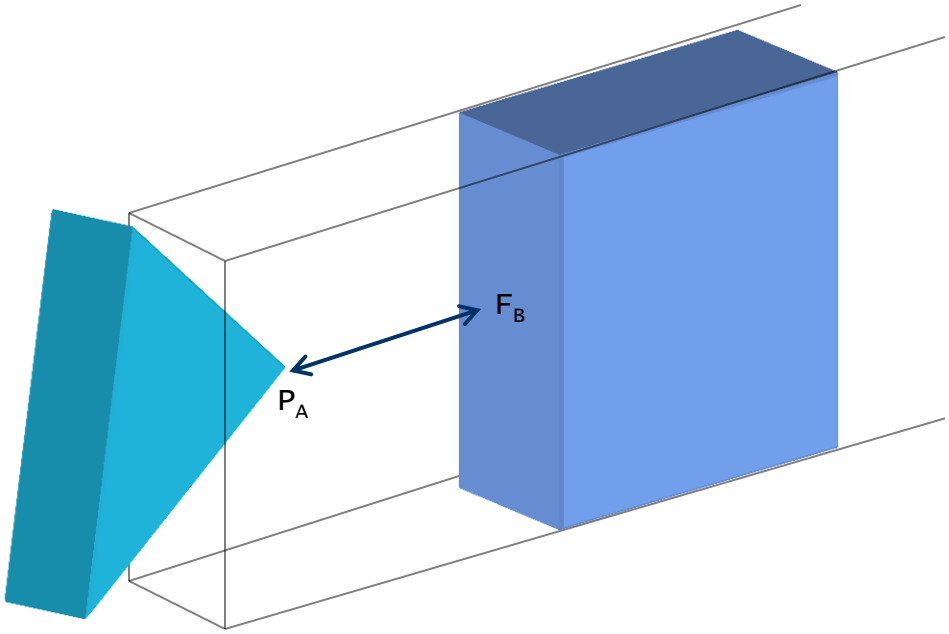
point-edge applicability criteria

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.



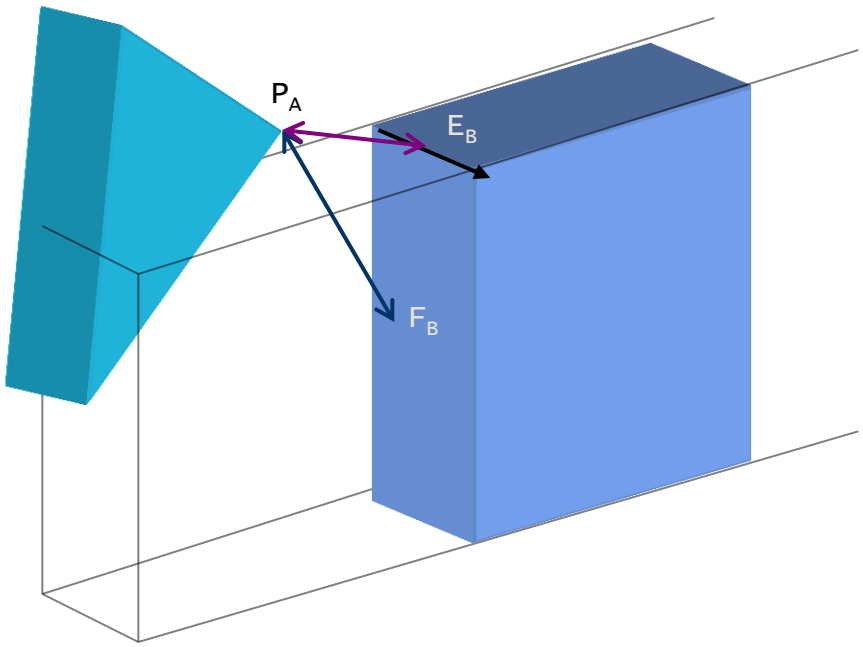
point-face applicability criteria

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.



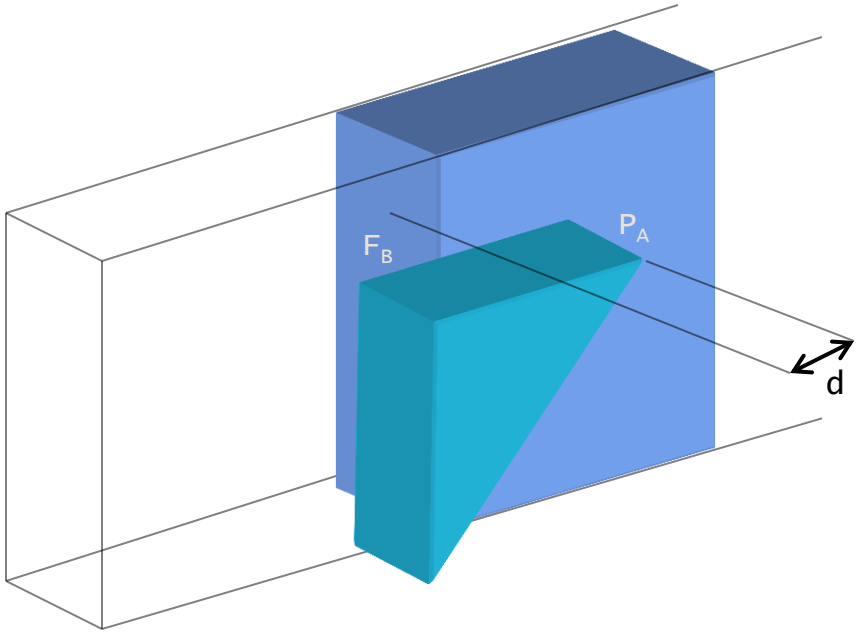
point-face applicability criteria

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.



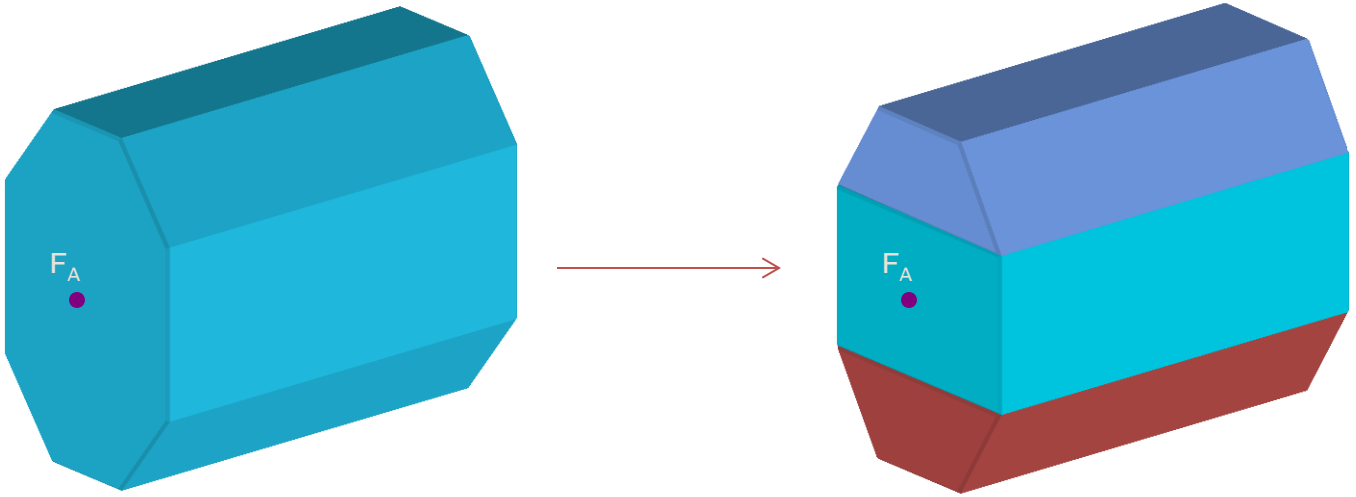
point-face applicability criteria

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.

conclusions

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 without any significant increase in running time.