# Game Physics

Game and Media Technology
Master Program - Utrecht University

Dr. Nicolas Pronost

# Collision detection

# The story so far

- We have rigid bodies moving in space according to forces applied on them
- We have seen when and how to apply gravity, drag *etc.*
- But reaction forces occur when a rigid body is in contact with another body
- So we need to be able to detect that event and to apply the correct reaction force
  - Collision detection
  - Collision solving

# Collisions and geometry

- Now is finally when we need the geometry of the object
  - A point (*e.g.* COM) is not enough anymore
  - We must know where the objects are in contact to apply the reaction force at that position

*CryEngine 3 (BeamNG)*

# Collision detection algorithm

- Collision detection occurs in three phases
  - Broad phase
    - disregard pairs of objects that cannot collide
    - ➢ model and space partitioning
  - Mid phase
    - determine potentially colliding primitives
    - ➢ movement bounds
  - Narrow phase
    - determine exact contact between two shapes
    - ➢ Gilbert-Johnson-Keerthi algorithm

# Broad phase

# Collisions and geometry

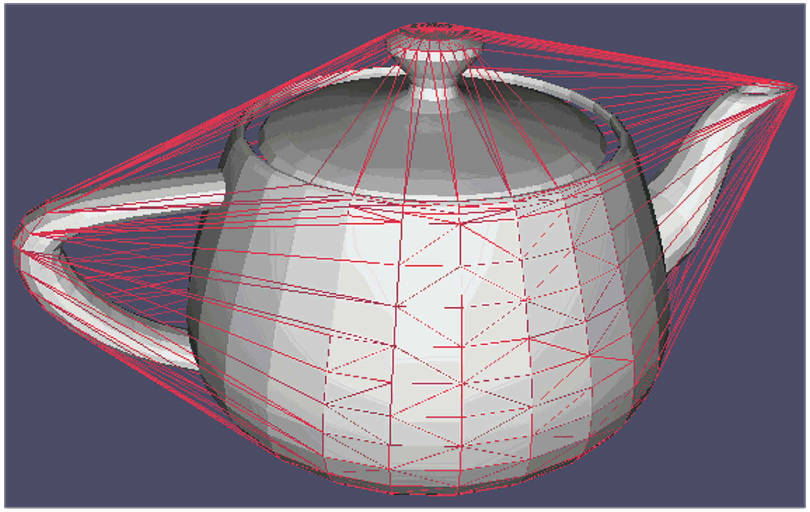
- Game physics engines use a simplification of the geometry
  - To compare ‘every vertex of every mesh’ at each frame is usually not possible in real-time
  - As primitive shapes are used to estimate the inertia, primitive shapes are also used to estimate the collisions
  - Collision shapes do not have to be the same as inertia shapes

# Model partitioning

- Technique used to quickly check complex objects using approximating bounding volumes
- A bounding volume has the following properties
  - It should fit as tight as possible the object
  - Overlap test with another volume should be fast
  - It should be described with little parameters
  - It should be fast to recalibrate under transformation
- What primitives to use so that collision checking is fast and accurate?

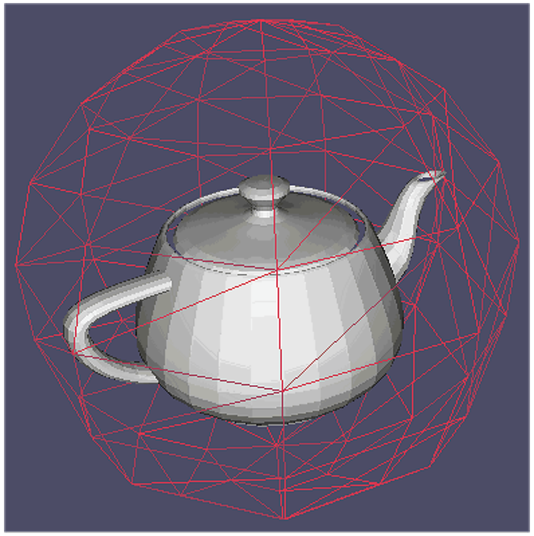
# Convex Hull

- Create the smallest convex surface/volume enclosing the object
  - Good representation of all convex objects
  - Create false positive collisions for concave objects
  - Can still be very complex, so costly detection



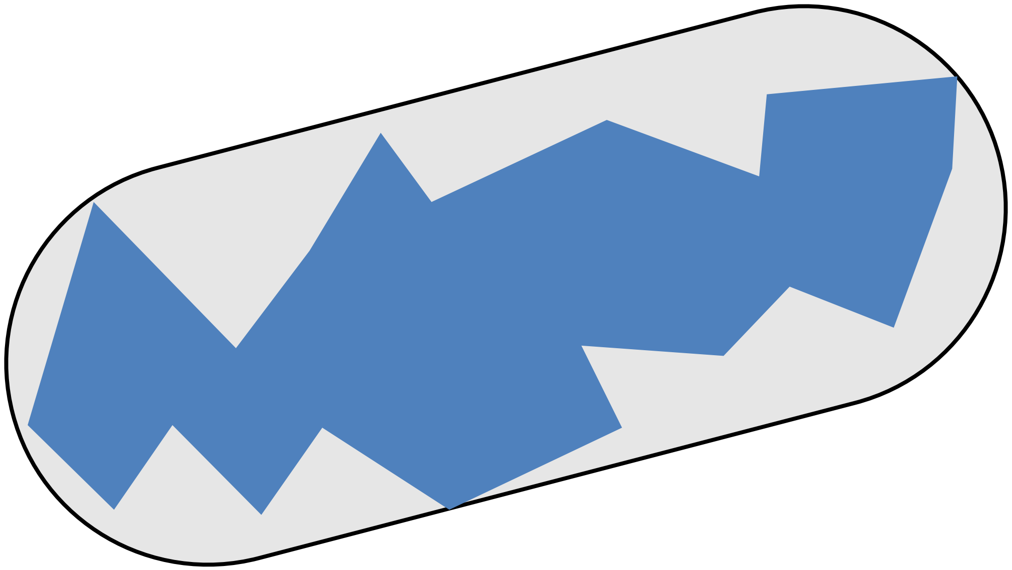
# Bounding Sphere

- Create the minimal sphere enclosing the object
  - Usually poor fit of the object (*e.g.* pipe), many false positive collisions
  - Stored in only 4 scalars, collision detection between spheres is very fast (11 prim. op.)
  - Trivial to update under rotation...



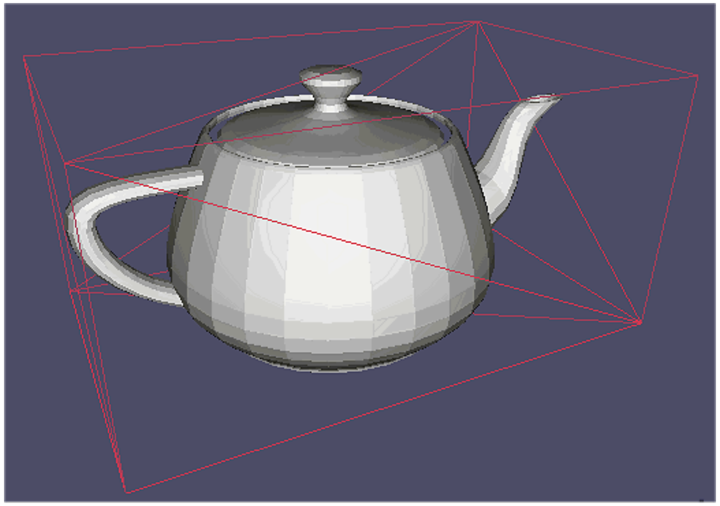
# Bounding Capsule

- The minimal swept bounding sphere enclosing the object
  - Better fit than bounding sphere
  - Collision detection still quite fast (bounding sphere with a distance to segment)



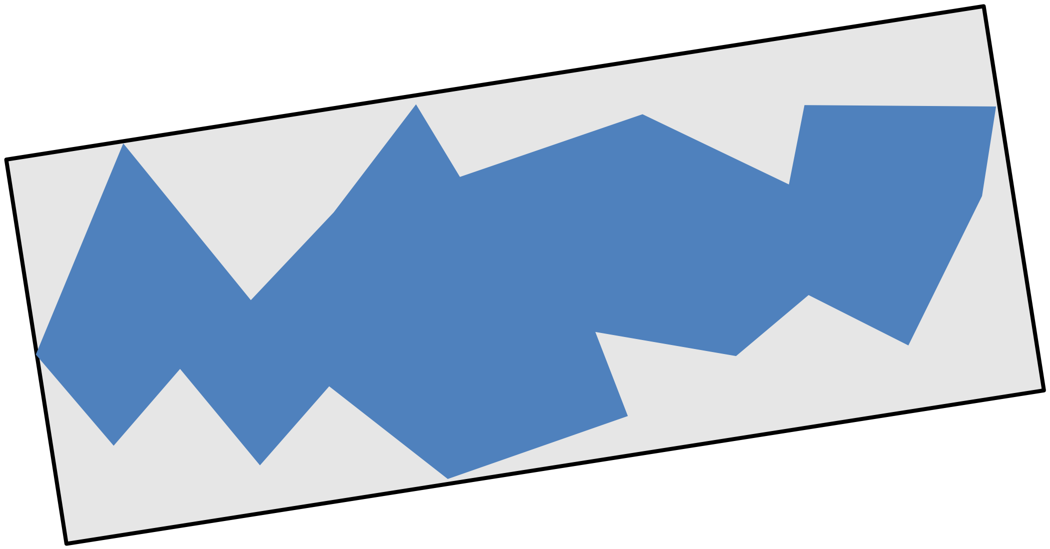
# Axis Aligned Bounding Box

- Create a box which dimensions are aligned with the axes of the world coordinate system
  - Usually poor fit of the object (*e.g.* diagonal box), many false positive collisions, recalculation after rotation
  - Stored in 6 scalars, collision detection between AABBs is very fast (6 prim. op.)



# Oriented Bounding Box

- The general minimal bounding box (no preferred orientation), abbreviated as OBB
  - Better fit than AABB, but worse than convex hull (*e.g.* triangle)
  - Stored in 9+6 scalars, collision detection slower than AABB (200 prim. op.), but much faster than convex hull
  - Similar to bounding capsule with sharp ends



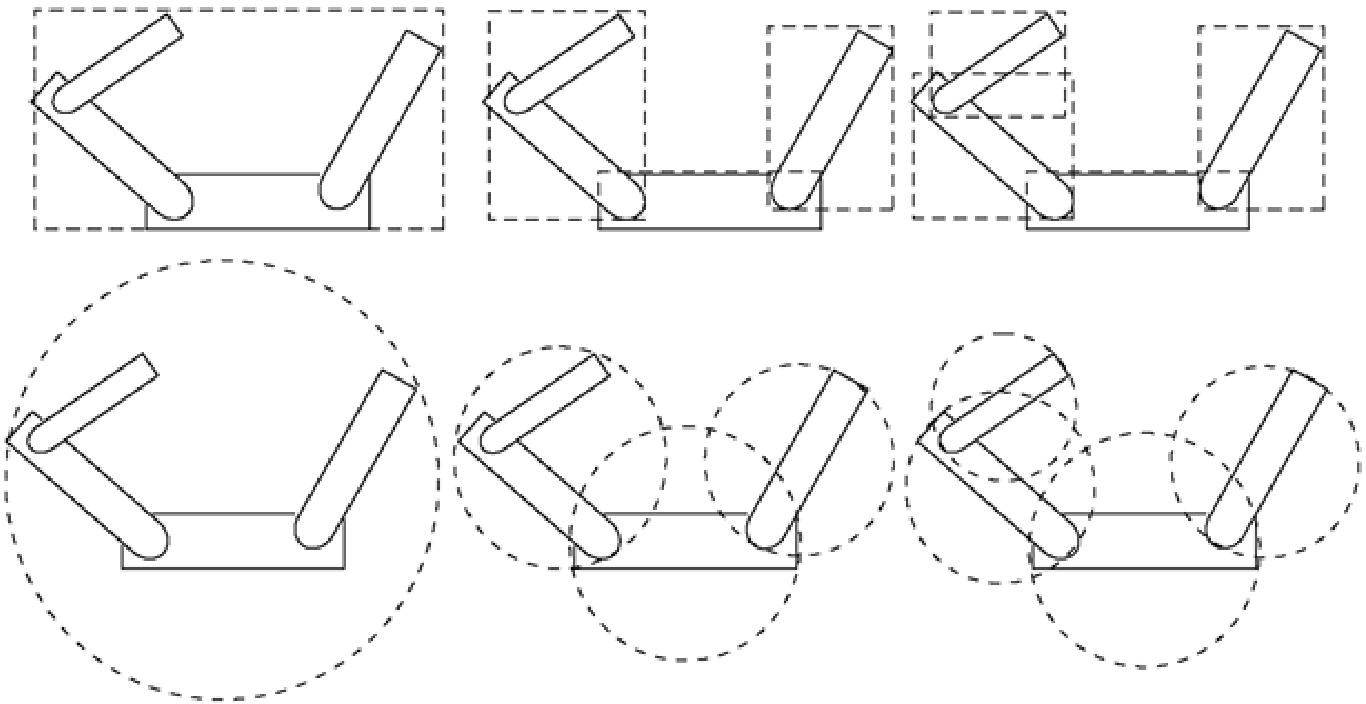
# Other primitives

- You can imagine using almost any primitive or combination of primitives
- As soon as the detection is faster than on the object itself there is an interest
  - Bounding cylinder
  - Bounding ellipsoid
  - *etc.*

# Bounding hierarchies

- Since one bounding volume can still creates many false positives, we build a hierarchy of volumes
- Called Bounding Volume Hierarchy (BVH)
- It has a tree structure with primitive volumes as leaves and enclosing volumes as nodes
- During collision detection, the hierarchies are traversed and child bounding volumes are checked only when necessary
  - children do not have to be examined if their parent volumes do not intersect

# Bounding hierarchies

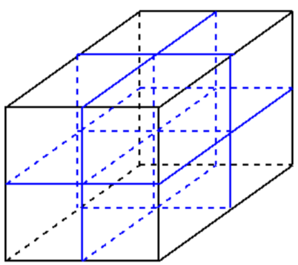


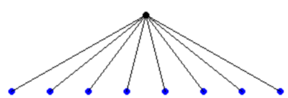
# Space partitioning

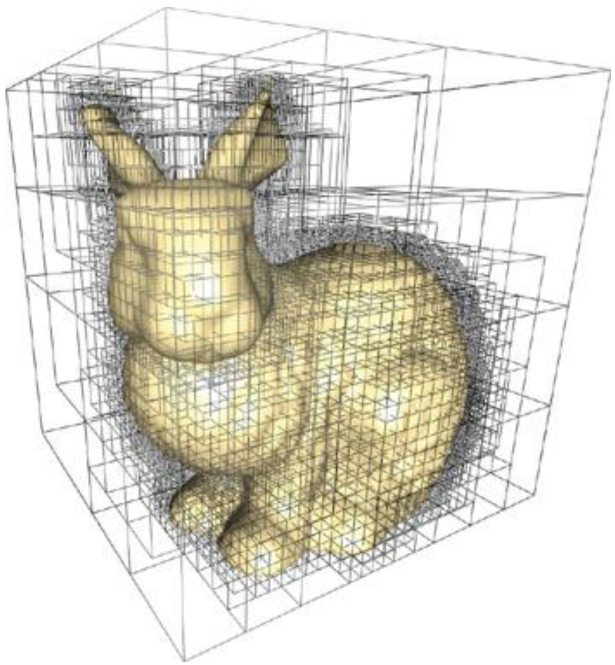
- Used to make a fast selection of which models to test for collision
- Based on the spatial configuration of the scene
- Associate together objects that are physically close to each other
- Only need to test collision with objects in the same partition
- Quickly disregards many unnecessary tests

# Octree

- An octree is a tree data structure in which each node has exactly eight children
- Partition the space in eight cubes (called octants) of equal volume along the dimensions of the space

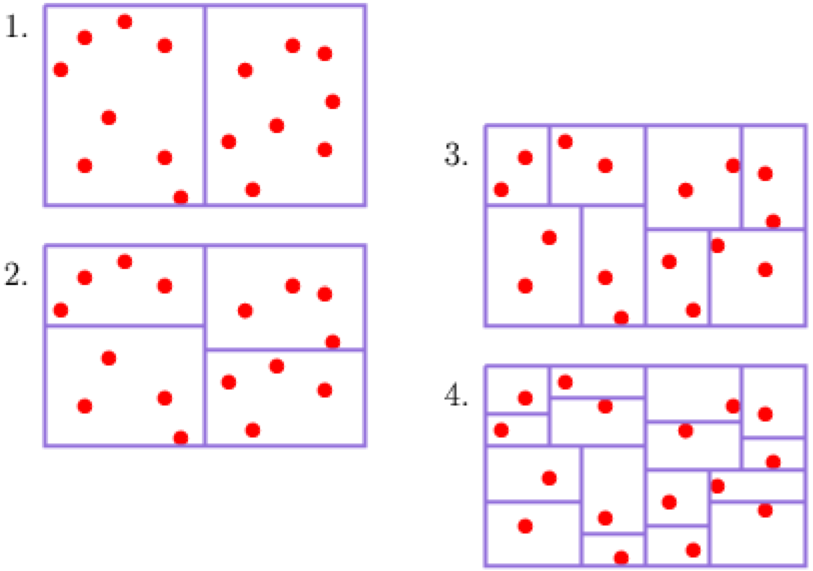






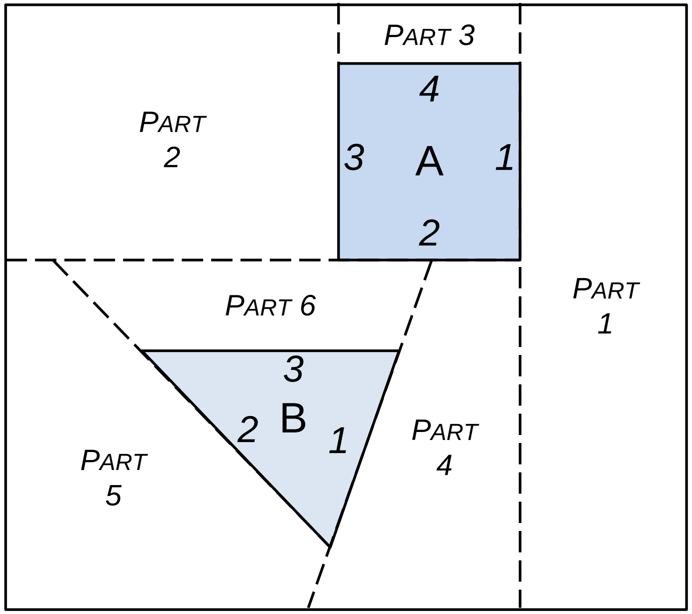
# Kd-tree

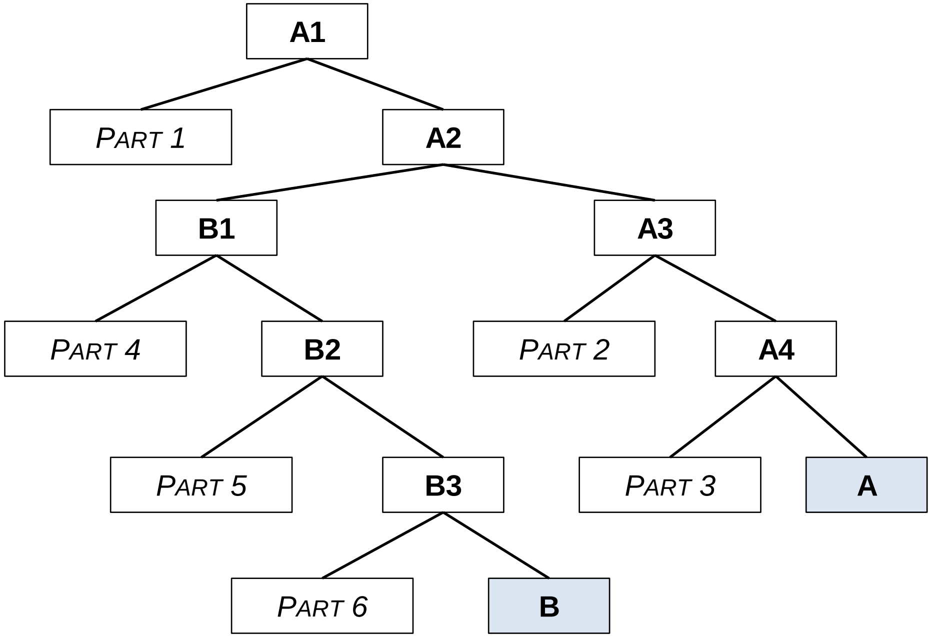
- A kd-tree (k-dimensional) is a binary tree where every node is alternately associated with one of the k-dimensions
- Usually the median hyperplane is chosen at each node



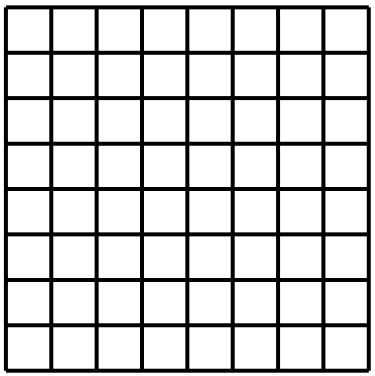
# Binary space partitioning

- Binary space partitioning (BSP) creates BSP trees
- Hyperplanes recursively partition space into two volumes but the planes can have any orientation
- Hyperplanes are usually defined by polygons in the scene

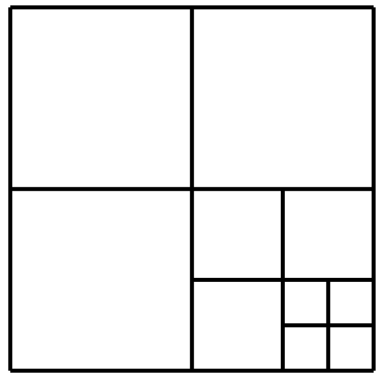


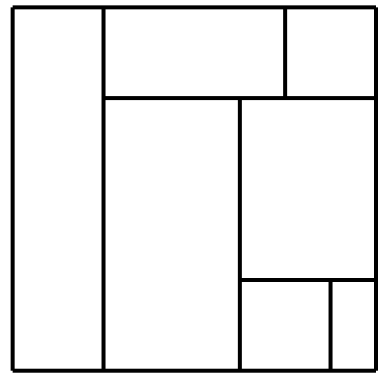
# Space partitioning summary



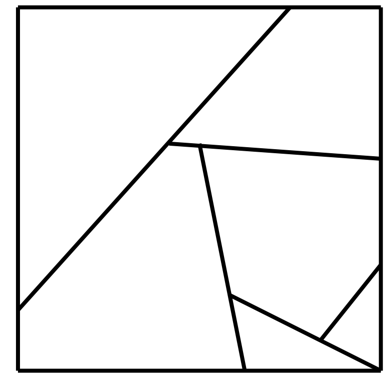
Uniform spatial subdivision



Quadtree
Octree



Kd-tree



BSP-tree

# Mid phase

# Collision between primitives

- You can imagine representing different objects with different primitives according to their original geometry
  - A simple convex object => convex hull
  - A spherical object like a ball => bounding sphere
  - A body part => bounding capsule
  - A box sliding on the floor => AABB
  - A box-like object that can translate and rotate => OBB
- Ideally you have to implement detection algorithms for every possible combination of primitives
  - Some are easier to implement than others

# Sphere-Sphere

- For two spheres $A$ and $B$ to intersect, the distance between their centers $c_A$ and $c_B$ should be smaller than the sum of their radii $r_A$ and $r_B$

$$A \cap B \neq \emptyset \Leftrightarrow \|c_A - c_B\| \leq r_A + r_B$$

- Distance between two non-intersecting spheres

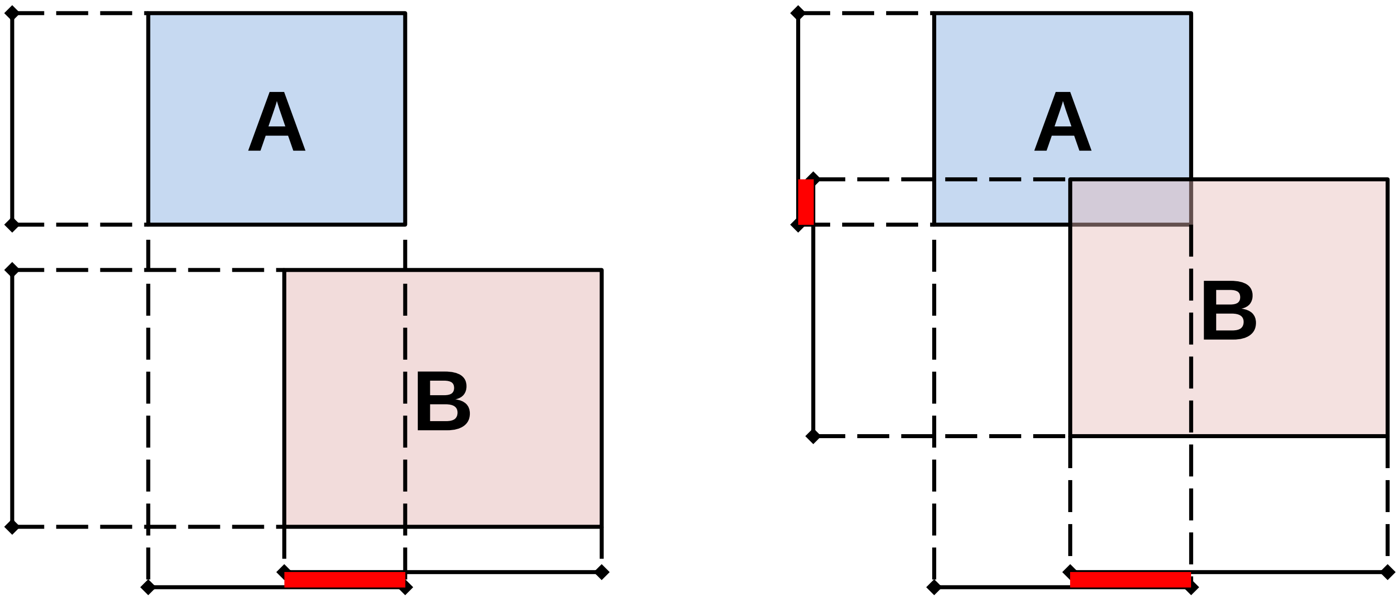
$$d(A, B) = \max(\|c_A - c_B\| - (r_A + r_B), 0)$$

- Penetration depth of two intersecting spheres

$$p(A, B) = \max(r_A + r_B - \|c_A - c_B\|, 0)$$

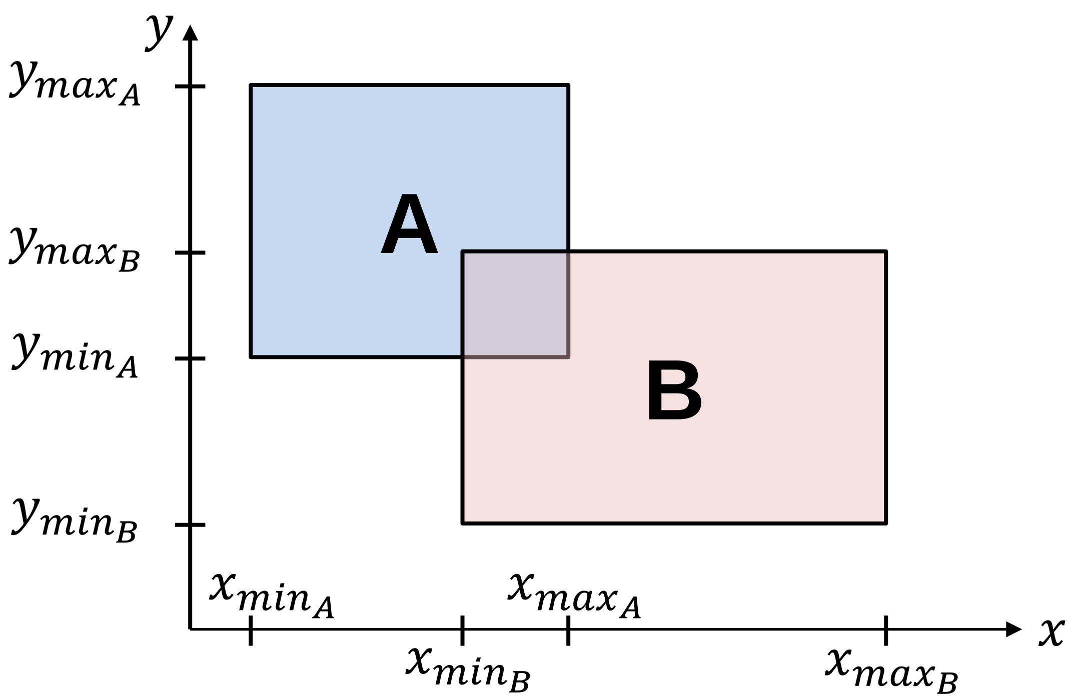
# AABB-AABB

- Project the boxes onto the axes, you will obtain two/three intervals per box, the two boxes collide if the intervals overlap



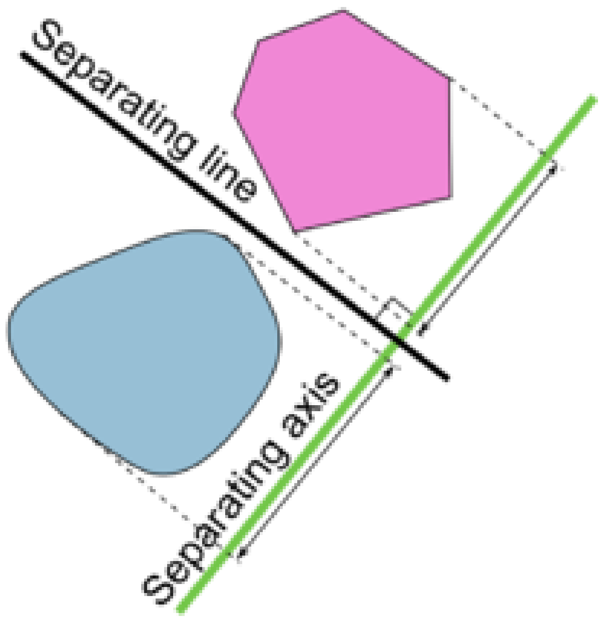
# AABB-AABB

$$A \cap B = \emptyset \Leftrightarrow$$
$$x_{max_A} < x_{min_B} \vee y_{max_A} < y_{min_B} \vee$$
$$x_{min_A} > x_{max_B} \vee y_{min_A} > y_{max_B}$$



# Separating Axis Theorem

- Given two convex shapes, if we can find an axis along which the projections of the two shapes do not overlap, then the shapes do not collide



# Separating Axis Theorem

- In 2D, each of these potential separating axes is perpendicular to one of the edges of each shape
  - We solve our 2D overlap query using a series of 1D queries
  - If we find an axis along which the objects do not overlap, we don't have to continue testing the rest of the axes, we know that the objects don't overlap
- As in a game it is more likely for two objects to **not** overlap, it speeds up calculations

# Separating Axis Theorem

- For AABB-AABB it is easy to apply as the possible separating axes on which we have to project the object are the main axes
- Equivalent to our previous collision checking of overlap of intervals

# Separating Axis Theorem

- For non-axis-aligned shapes, we have to project our objects on the axes perpendicular to the edges

Box-Polygon

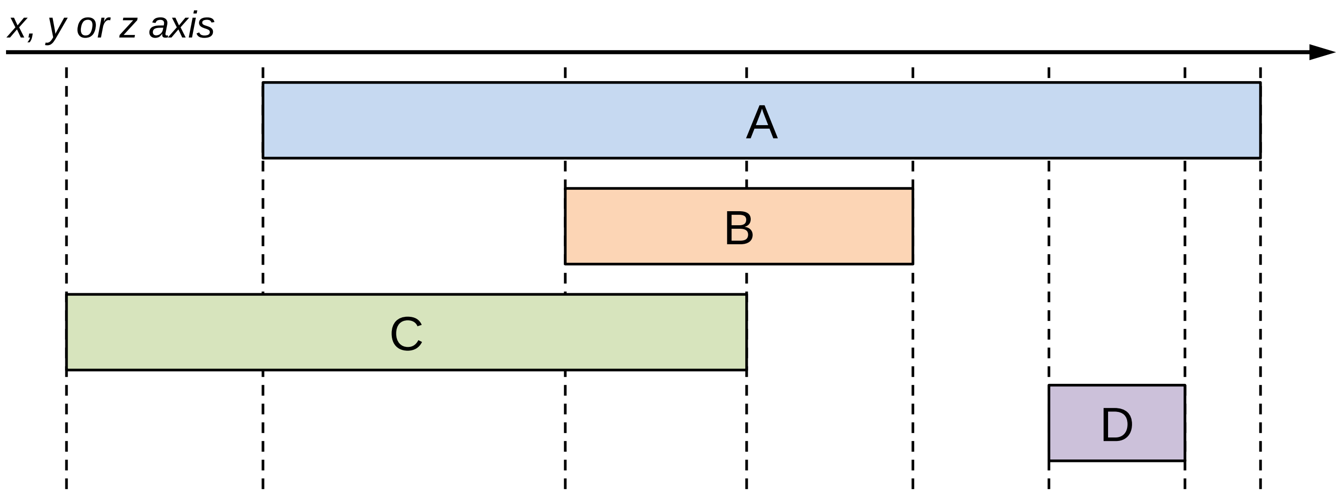
Box-Curve

Circle-Polygon

# Sweep and prune algorithm

- Several variants exist but all first sort then prune
- Objects are defined with their AABB
- 2 objects overlap if and only if their projections on the x, y and z coordinate axes overlap
  - The projections give 3 [min,max] intervals
  - The min and max are stored in 3 sorted structures
  - Scan the objects in increasing order of min
  - Detect possible overlapping pair when min of an object is smaller than max of another
  - Combine the three results (AND condition to overlap)

# Sweep and prune algorithm




`CurrentObjects [] [C] [C,A] [C,A,B] [A,B] [A] [A,D] [A] []`

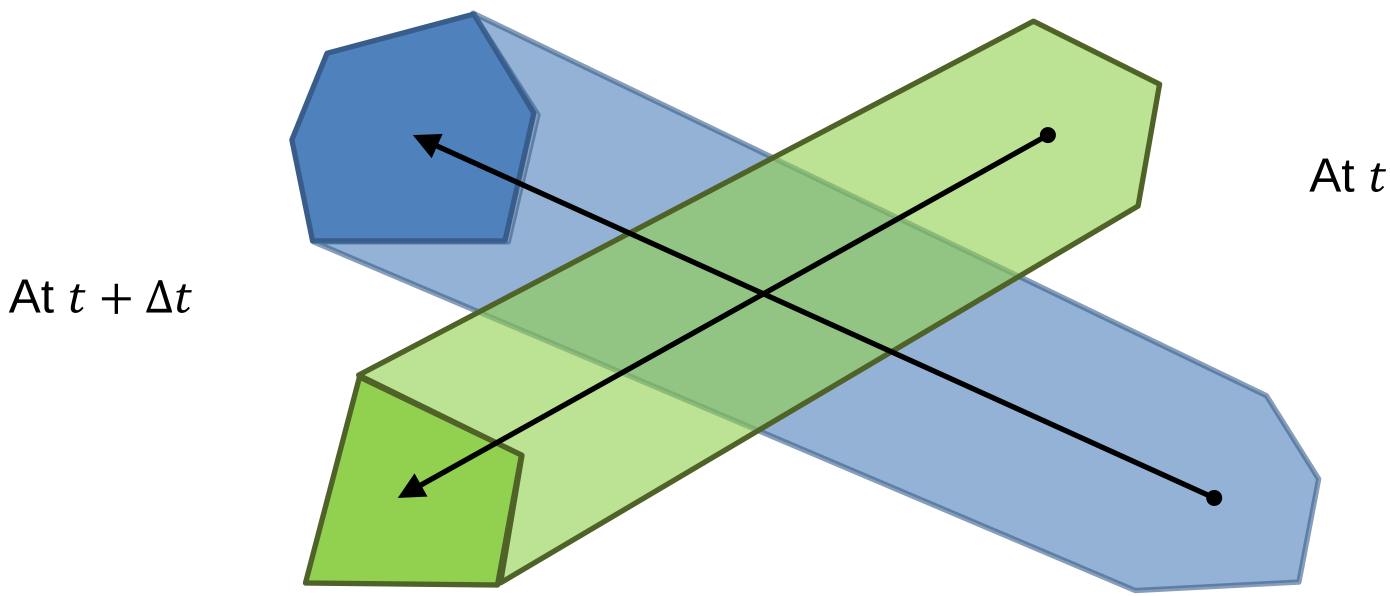
`CandidatePairs [CA] ∪ [BC,BA] ∪ [DA]`

=

`[CA,BC,BA,DA]`

# The time issue

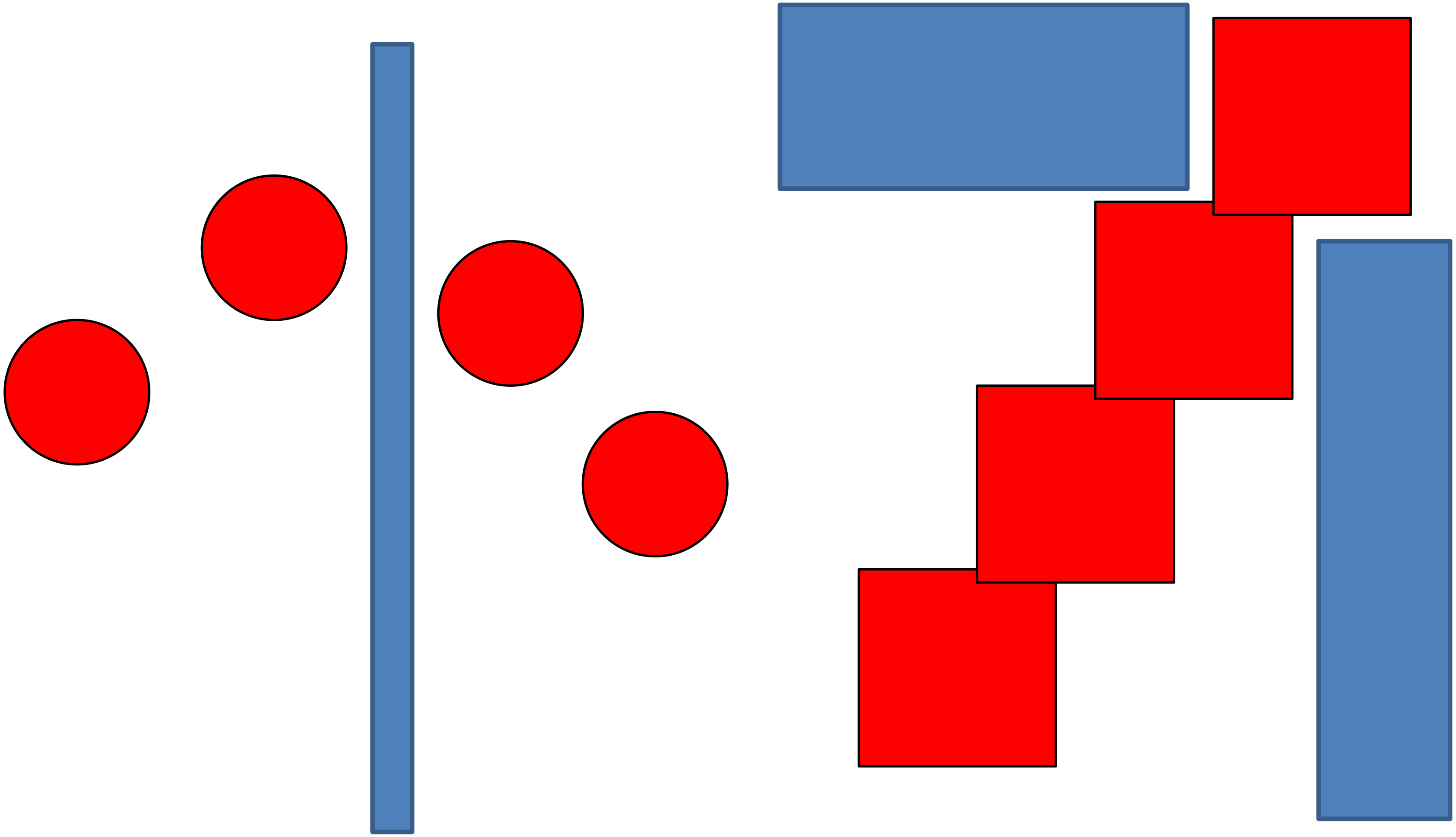
- Looking at uncorrelated sequences of positions is not enough
- Our objects are in motion and we need to know when and where they collide
  - as we want to react to the collision *e.g.* bouncing



# Tunneling

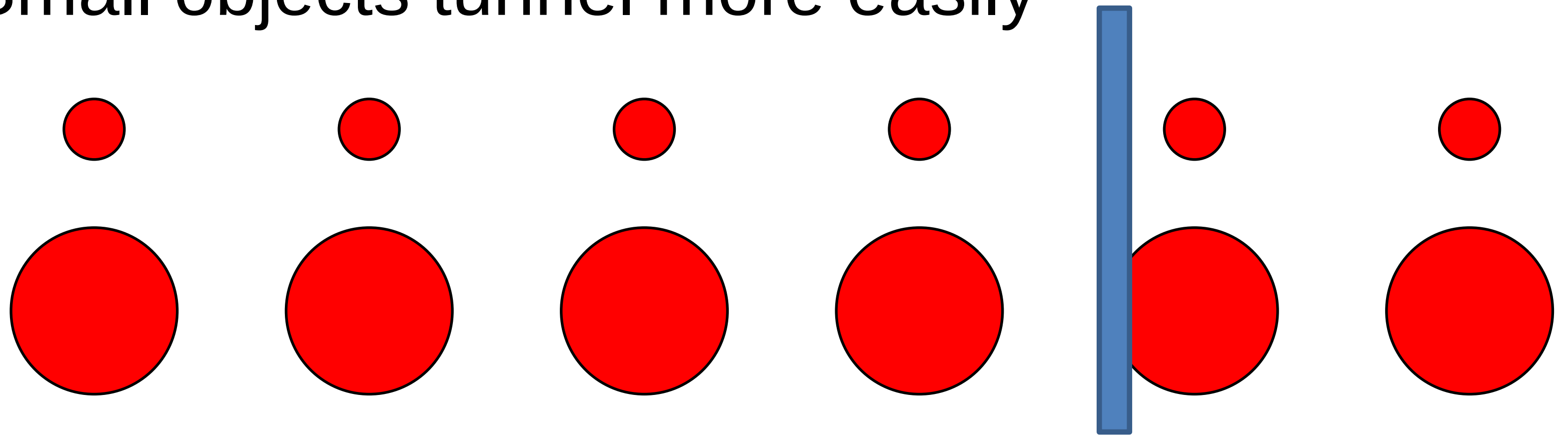
- Collision in-between steps can lead to tunneling
  - Objects pass through each other
    - They did not collide at $t$ and do not collide either at $t + \Delta t$
    - But they did collide somewhere in between
  - Lead to false negatives
- Tunneling is a serious issue in gameplay
  - Players getting to places they should not
  - Projectiles passing through characters and walls
  - Impossibility for the player to trigger actions on contact events

# Tunneling

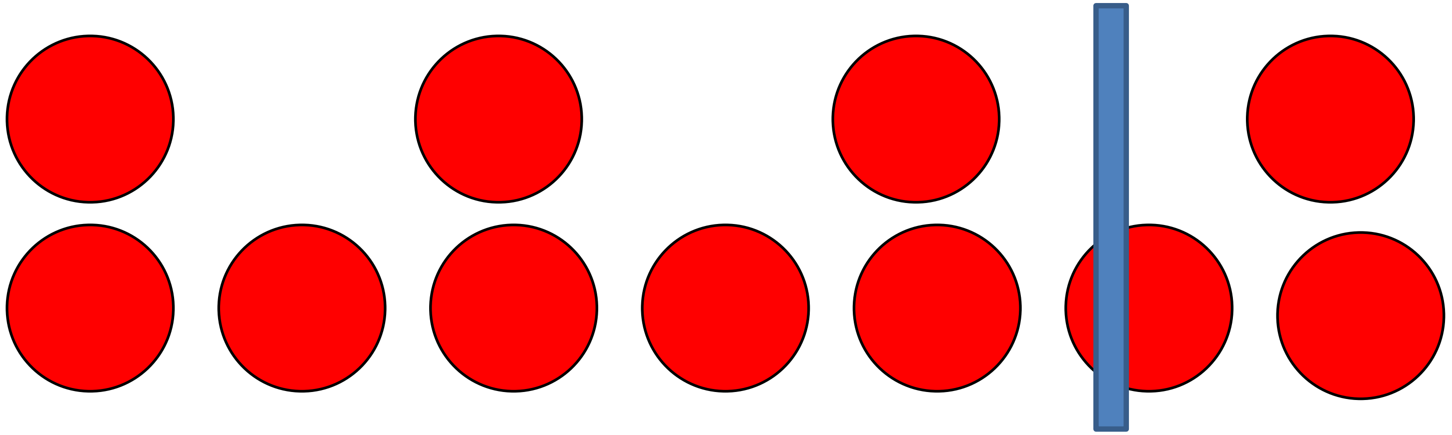


# Tunneling

- Small objects tunnel more easily



- Fast moving objects tunnel more easily



# Tunneling

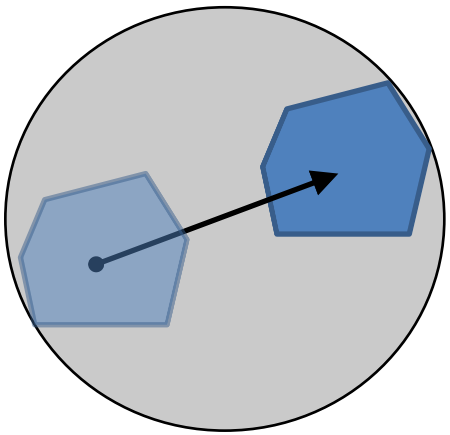
- Possible solutions
  - Minimum size requirement?
    - Fast object still tunnel
  - Maximum speed limit?
    - Small and fast objects not allowed (*e.g.* bullets...)
  - Smaller time step?
    - Essentially the same as speed limit
- We need another approach to the solution

# Movement bounds

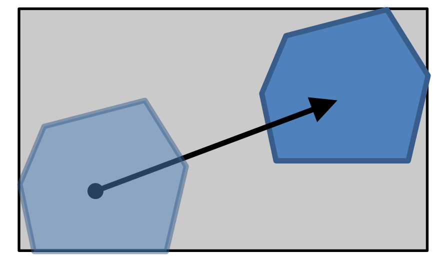
- Bounds enclosing the motion of the shape
  - In the time interval $\Delta t$, the linear motion of the shape is enclosed
  - Again, convex bounds are used, so the movement bounds are themselves primitive shapes

# Movement bounds

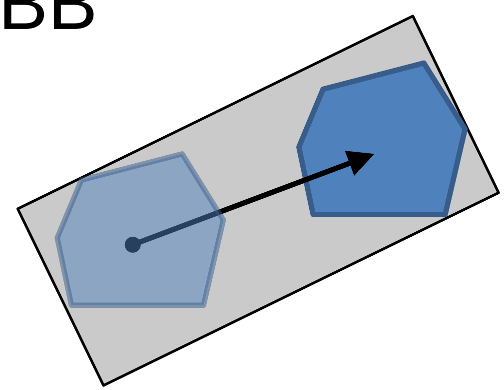
- Sphere



- AABB

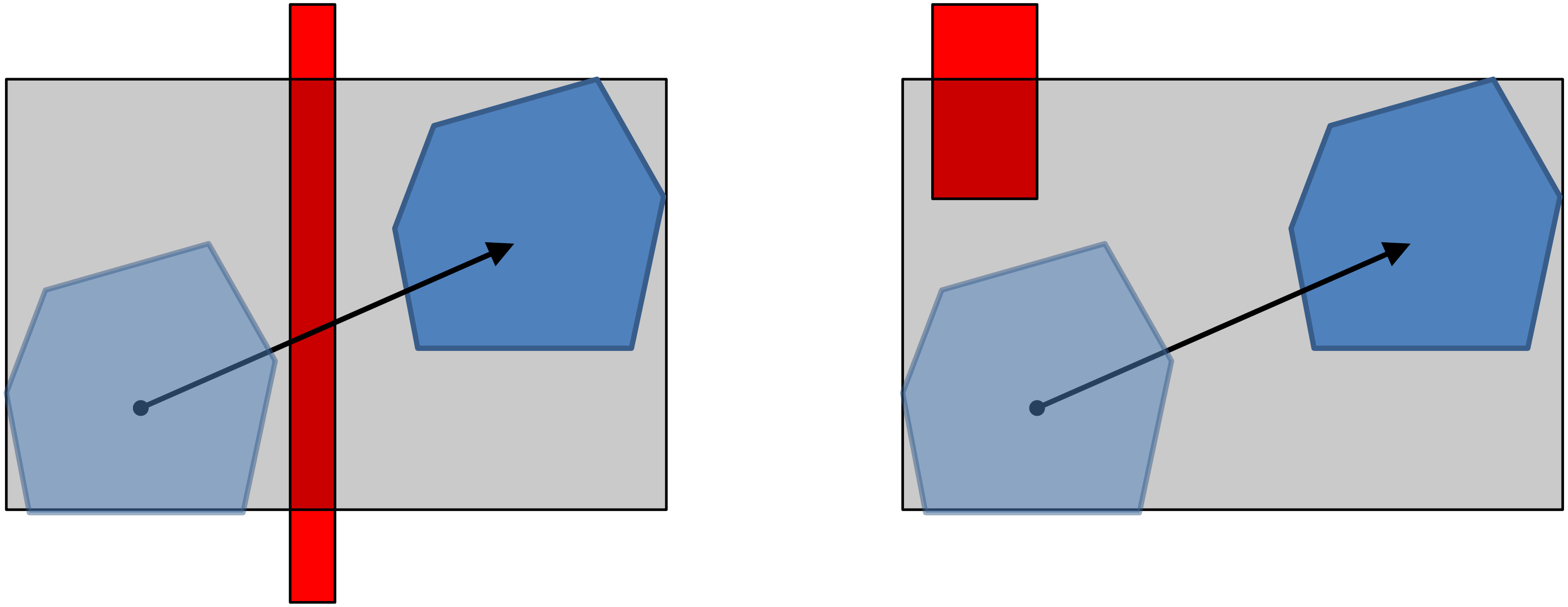


- OBB



# Movement bounds

- If movement bounds do not collide, there is no collision
- If movement bounds collide, there is possibly a collision

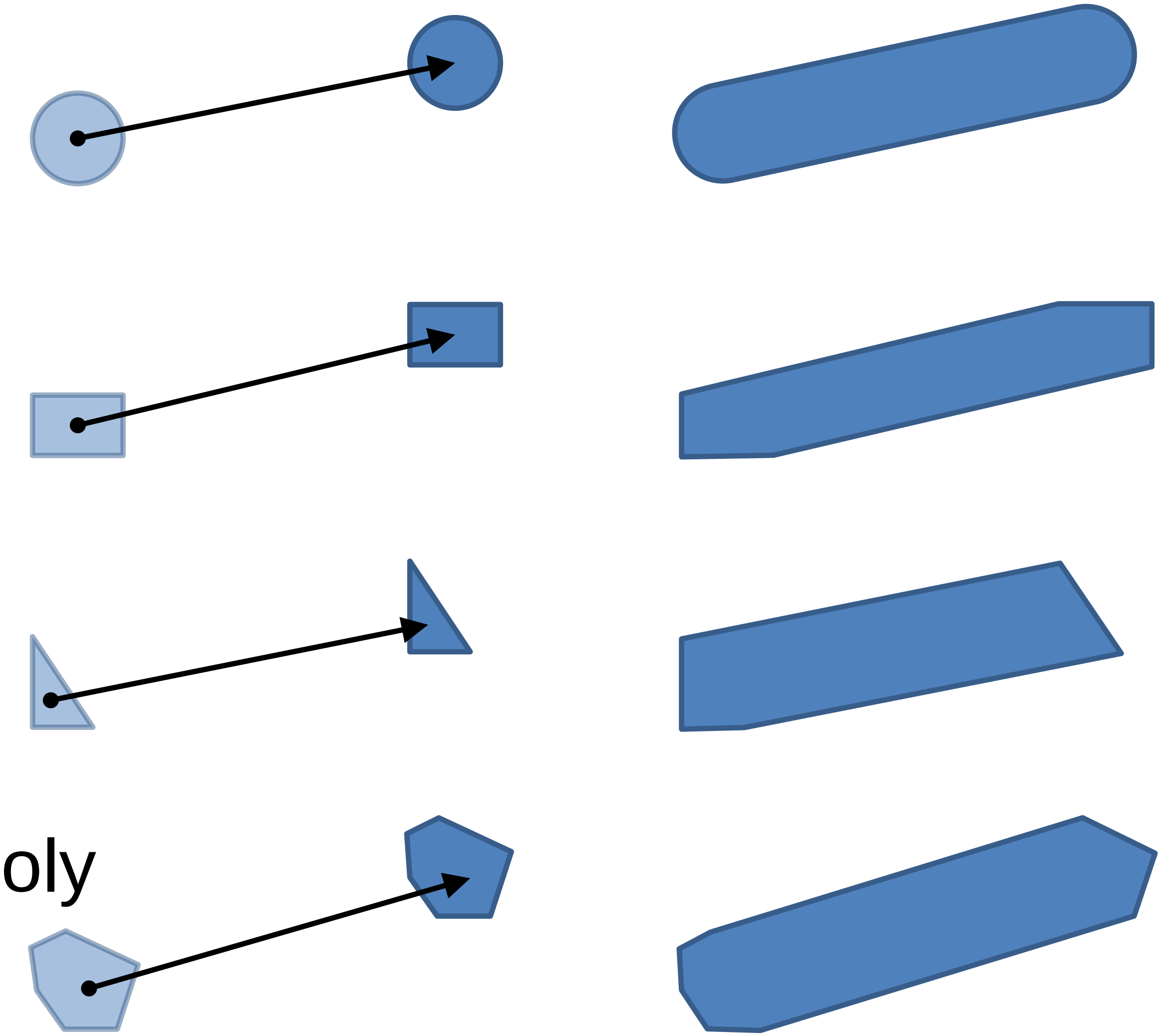


# Swept bounds

- As primitive based movement bounds do not have a really good fit, we can use swept bounds
  - More accurate, but more costly to calculate collisions
- A swept bound (or swept shape) is constructed from the union of all surfaces (volumes) of a shape under a transformation
  - we use the affine transformation from $t$ to $\mathrm{t} + \Delta t$

# Swept bounds

- Swept sphere
  - capsule
- Swept AABB
  - convex poly
- Swept triangle
  - convex poly
- Swept convex poly
  - convex poly



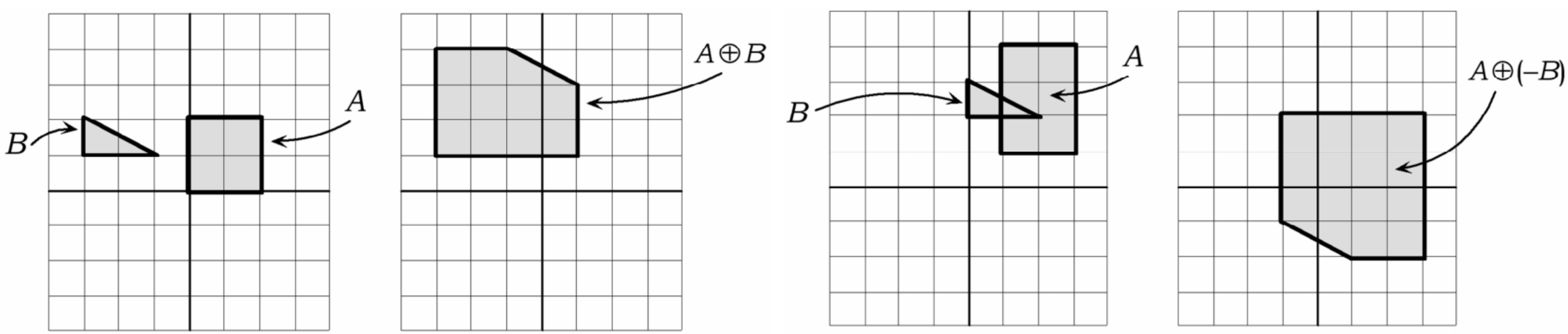
# Narrow phase

# GJK algorithm

- This algorithm effectively determines the intersection between polyhedra by computing the Euclidean distance between them
- Based on the property that the distance is the same as the shortest distance between their Minkowski difference and the origin
- Two new problems
  - Calculate the Minkowski difference between two objects
  - Calculate its distance to the origin (*i.e.* coordinate of the closest point to the origin)

# Minkowski difference

- The Minkowski difference $A \ominus B = A \oplus (-B)$ is obtained by adding $A$ to the reflection of $B$ about the origin
- Addition here means the swept bound of $B$ using $A$
- If $A$ and $B$ collide, $A \ominus B$ contains the origin

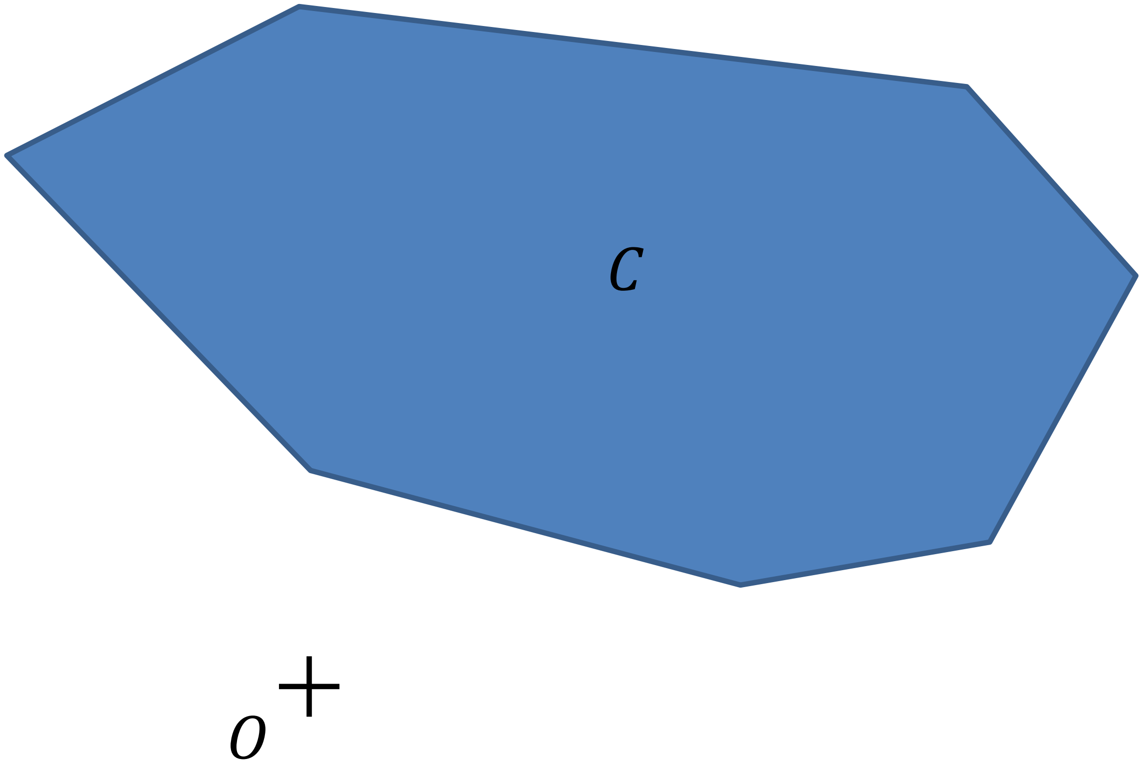


# GJK algorithm

- To calculate the shortest distance to the origin, the following algorithm is used
  1. Initialize the simplex set $Q$ with up to $d + 1$ points from the Minkowski difference object $C$
  2. If the origin is in the convex hull $CH(Q)$, then stop (collision detected)
  3. Compute the point $P$ of minimum norm of $CH(Q)$
  4. Reduce $Q$ to the smallest subset $Q'$ of $Q$ such that $P \in CH(Q')$
  5. Let $V = S_c(-P)$ be a supporting point in direction $-P$
  6. If $V$ is no more extreme than $P$ in direction $-P$, then return $||P||$
  7. Add $V$ to $Q$ and go to step 2

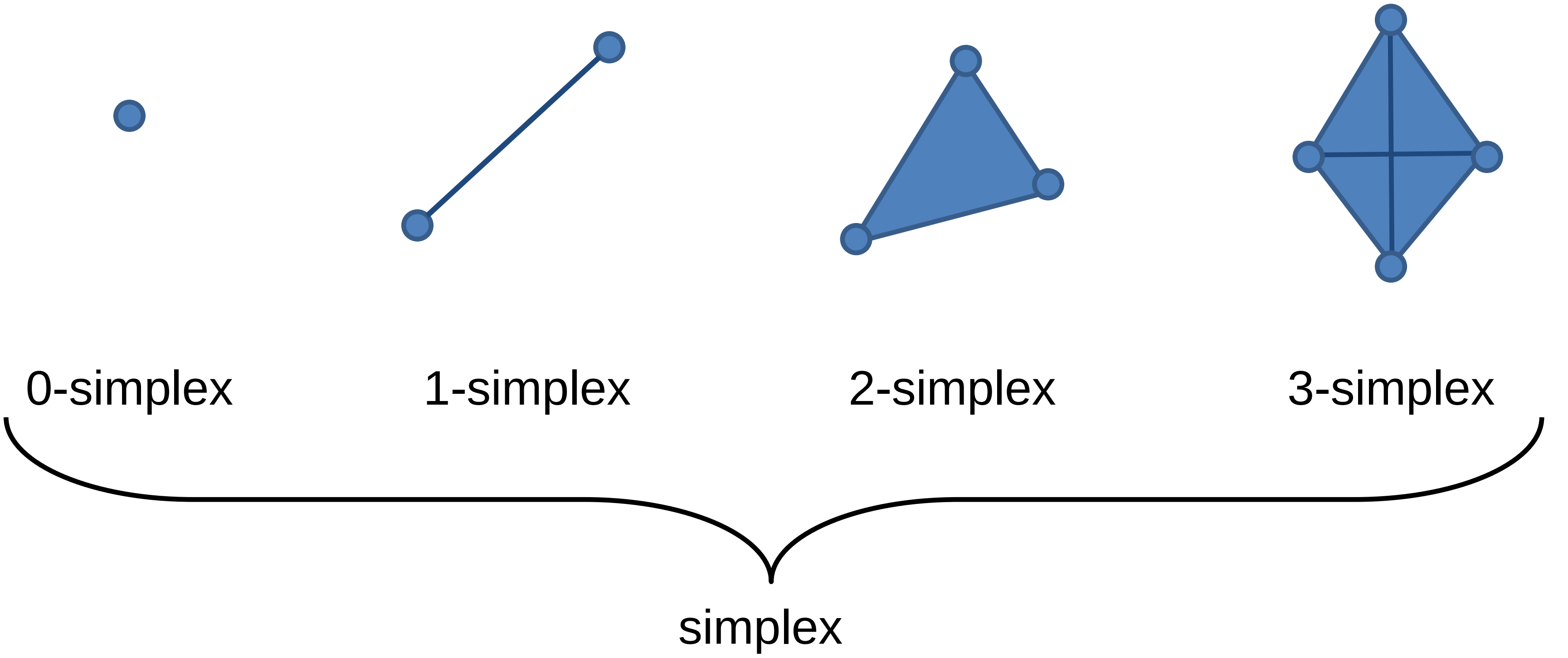
# GJK algorithm example

- Imagine the following Minkowski difference object $C$ and origin $O$



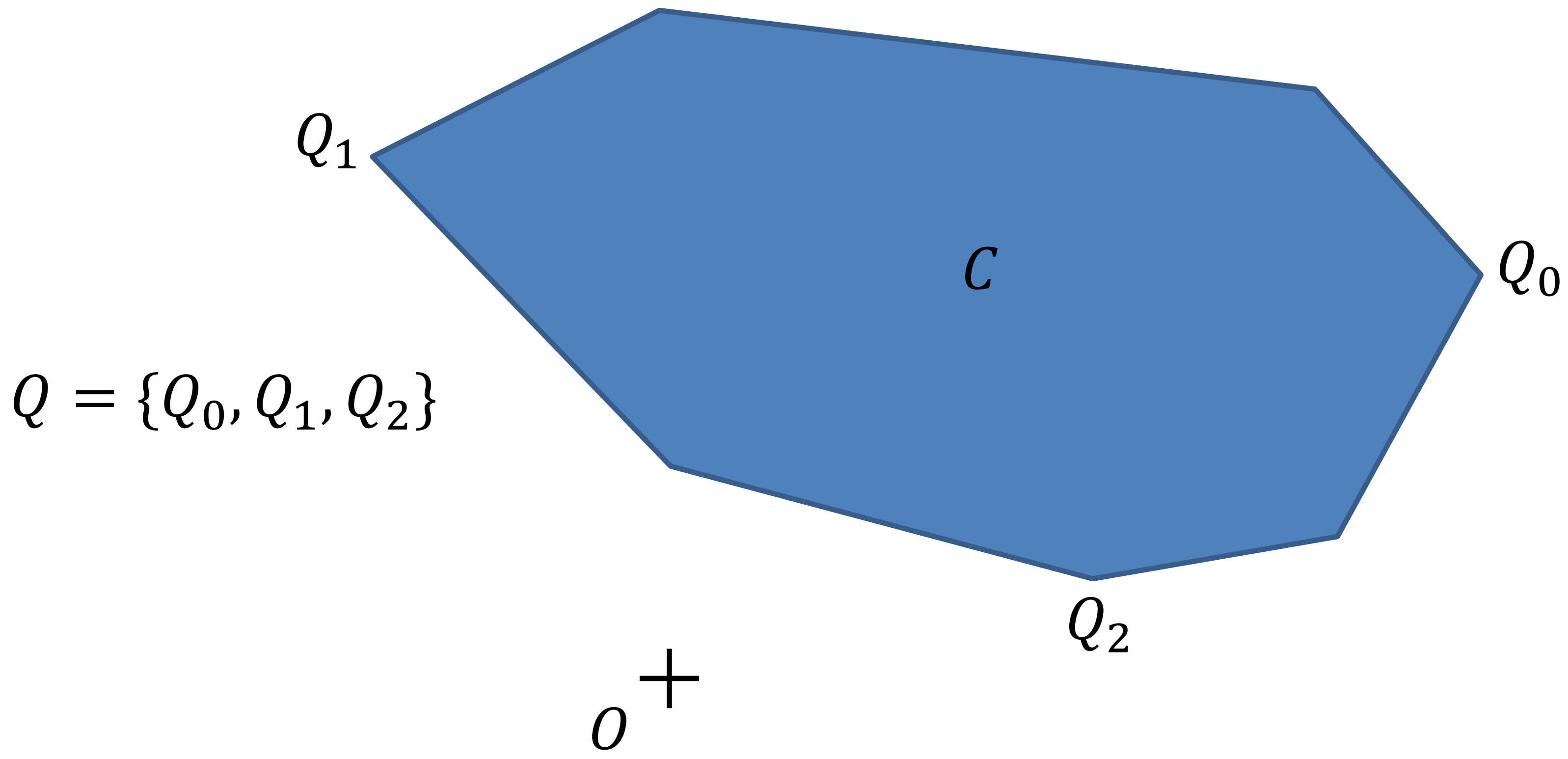
# GJK algorithm example

1. Initialize the simplex set $Q$ with up to $d$+1 points from the Minkowski difference object $C$



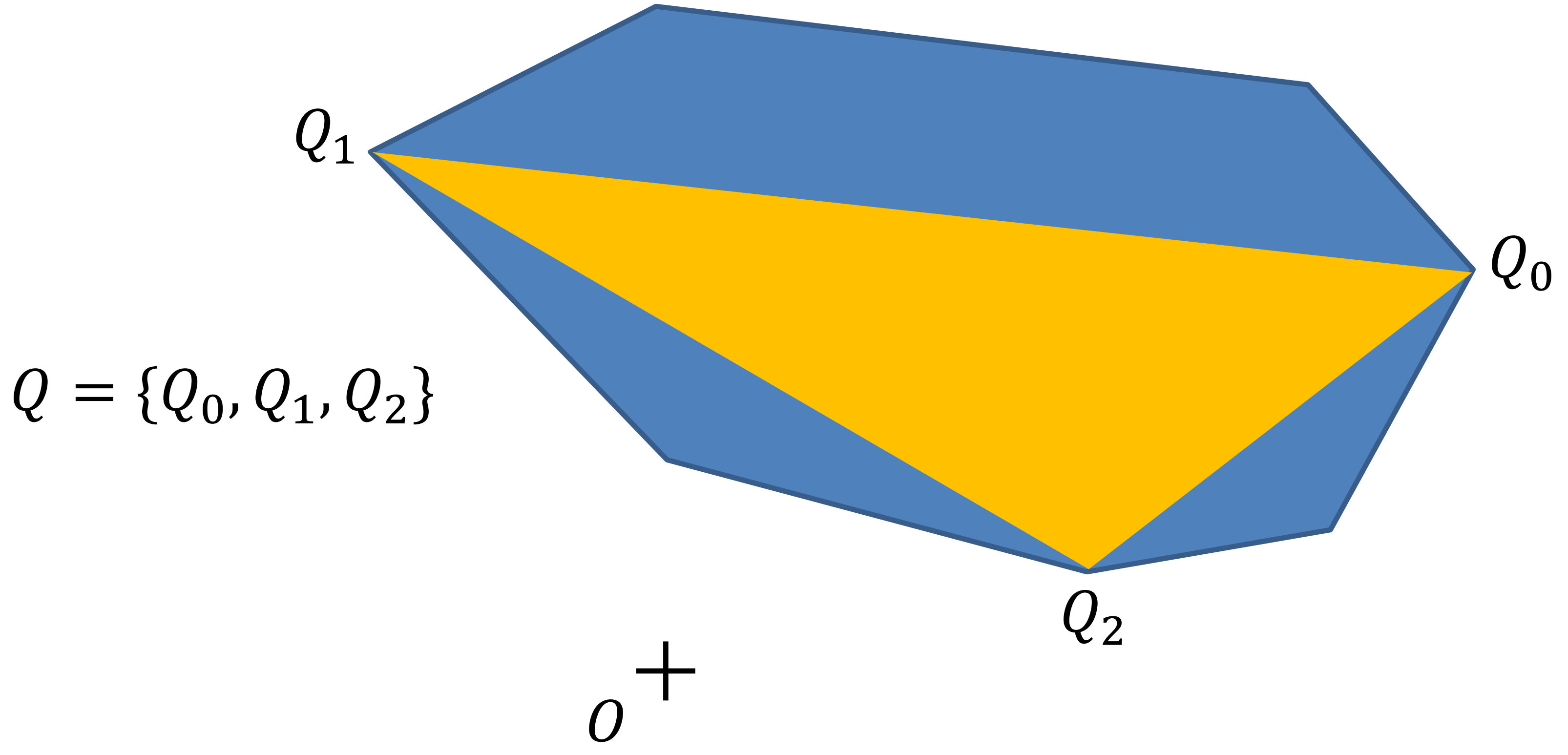
# GJK algorithm example

1. Initialize the simplex set $Q$ with up to $d$+1 points from the Minkowski difference object $C$



# GJK algorithm example

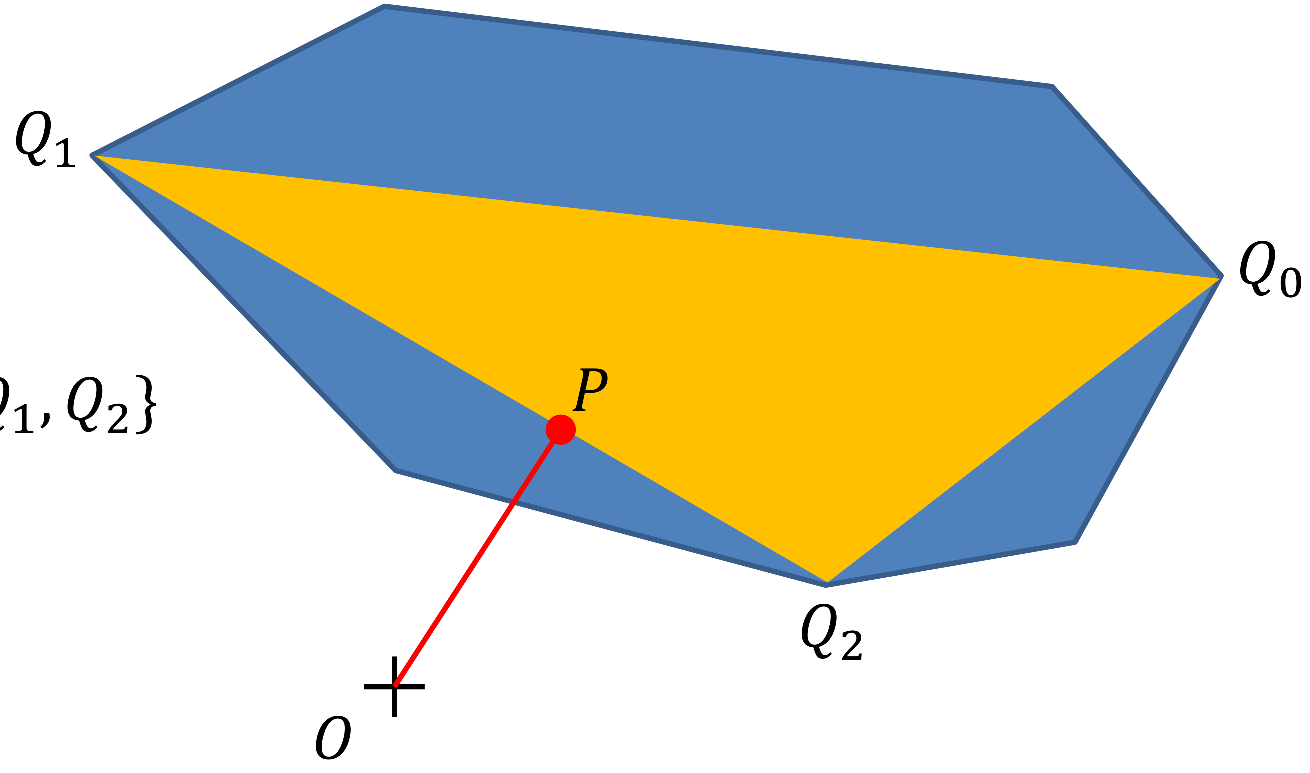
2. If the origin is in the convex hull $CH(Q)$, then stop (collision detected)



# GJK algorithm example

3. Compute the point $P$ of minimum norm of the convex hull $CH(Q)$

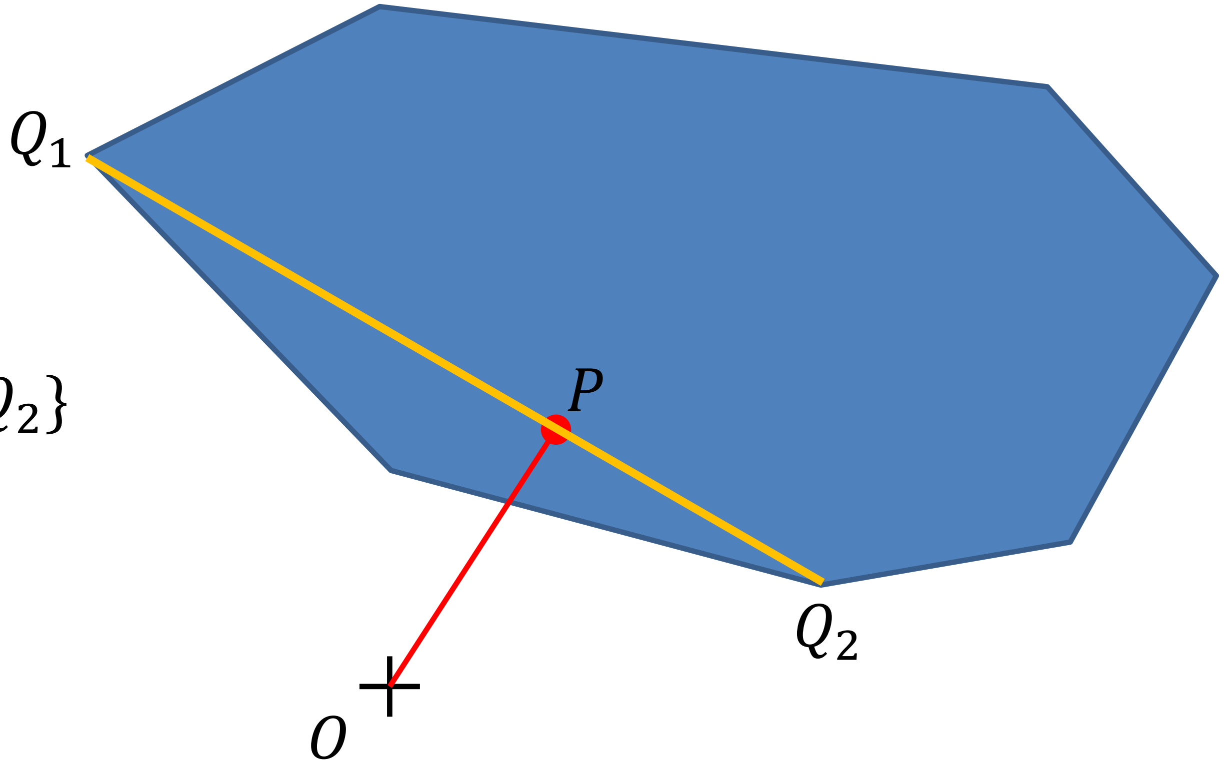
$Q = \{Q_0, Q_1, Q_2\}$



# GJK algorithm example

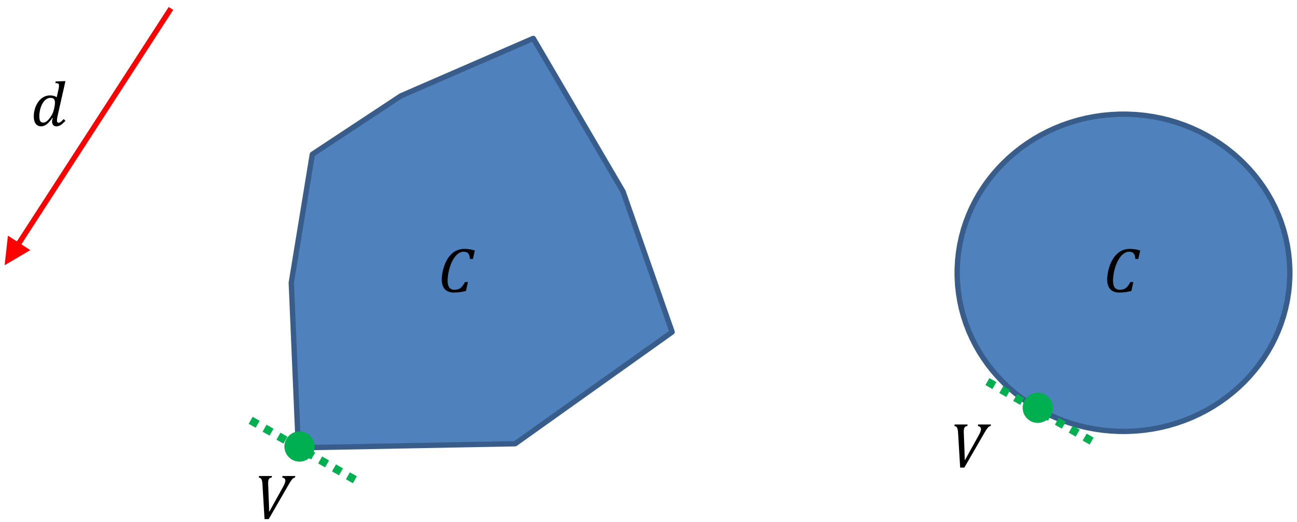
4. Reduce $Q$ to the smallest subset $Q'$ of $Q$ such that $P \in CH(Q')$

$$Q = \{Q_1, Q_2\}$$



# GJK algorithm example

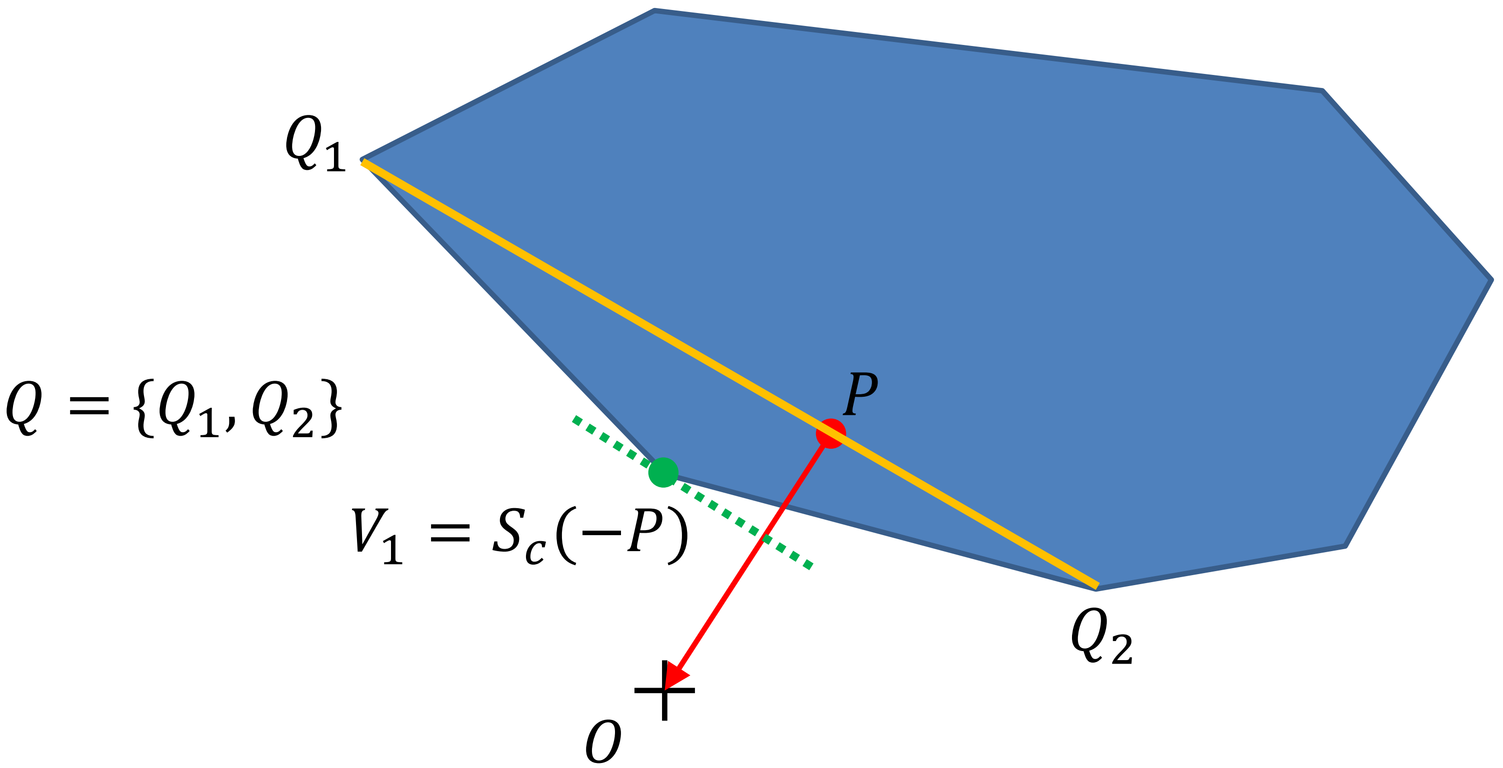
5. Let $V = S_c(-P)$ be a supporting point in direction $-P$




*Supporting point $V$ for a direction $d$ returned by support mapping function $S_c(d)$*

# GJK algorithm example

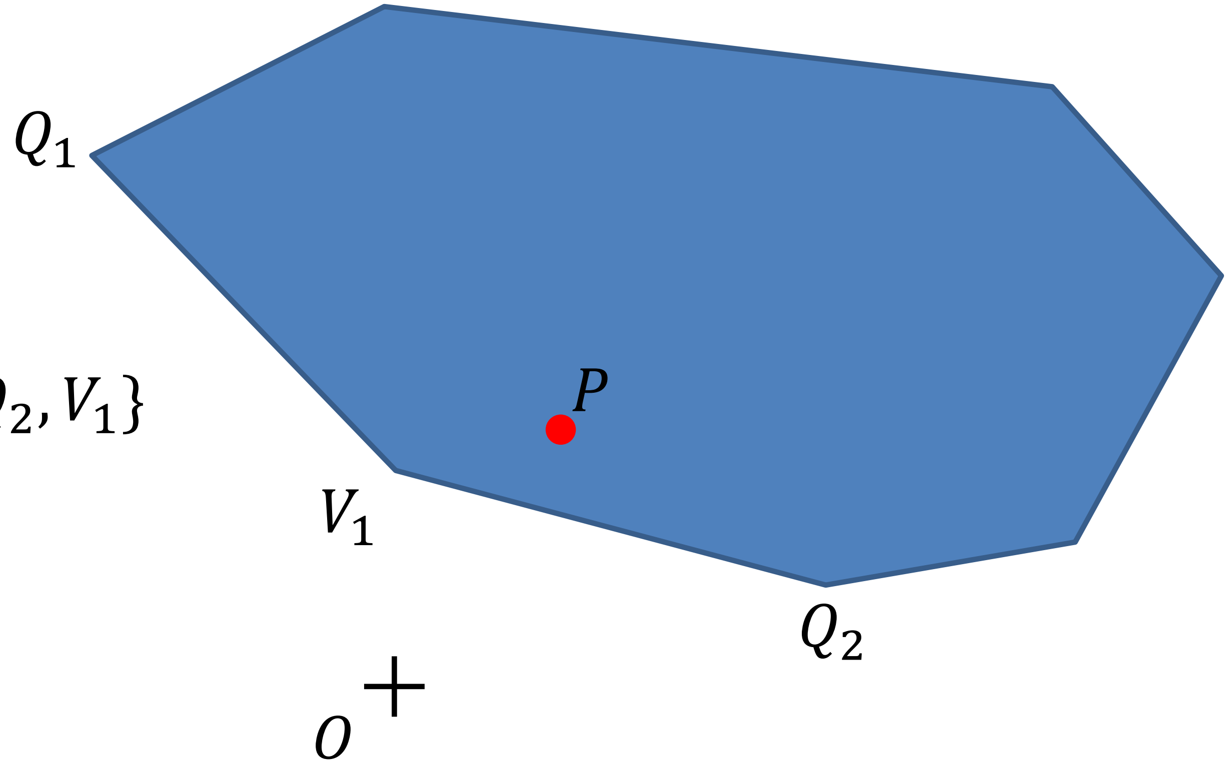
5. Let $V = S_c(-P)$ be a supporting point in direction $-P$. Let's call it $V_1$.



# GJK algorithm example

6. If $V$ is no more extreme than $P$ in direction $-P$, then return $||P||$

7. Add $V$ to $Q$ and go to step 2

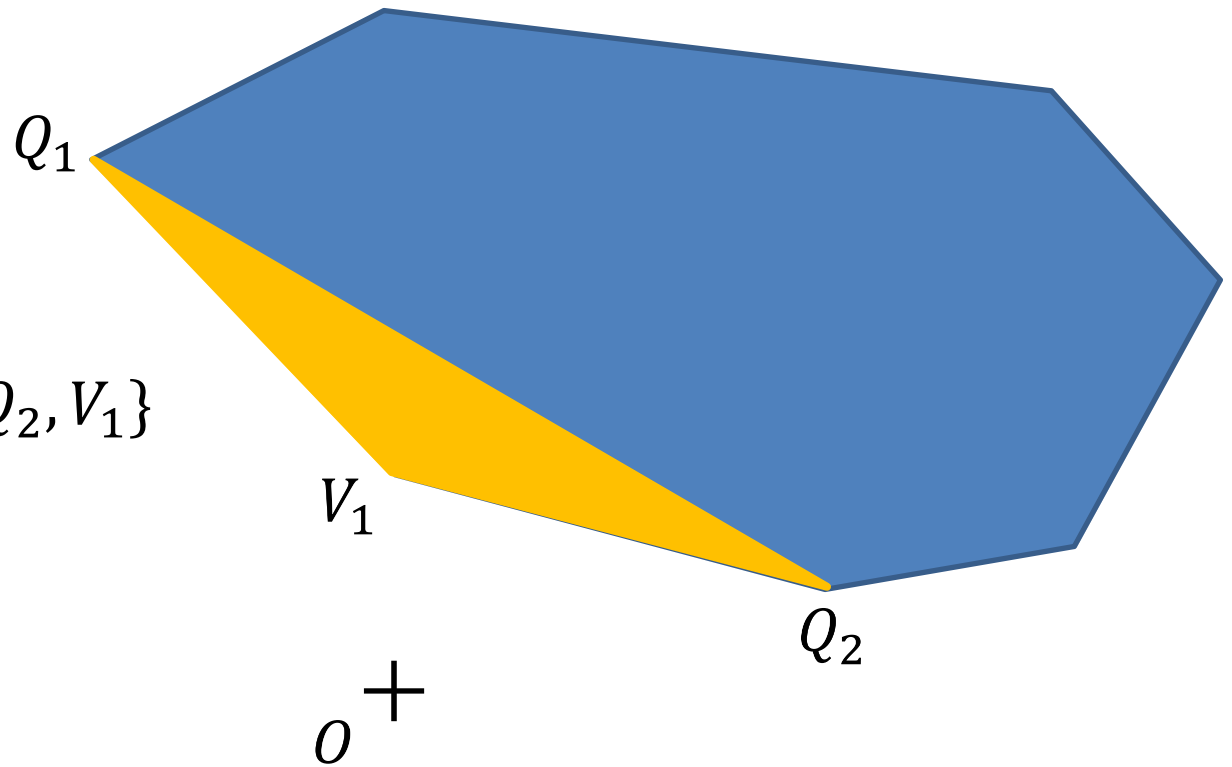



$Q = \{Q_1, Q_2, V_1\}$

# GJK algorithm example

2. If the origin is in the convex hull $CH(Q)$, then stop (collision detected)

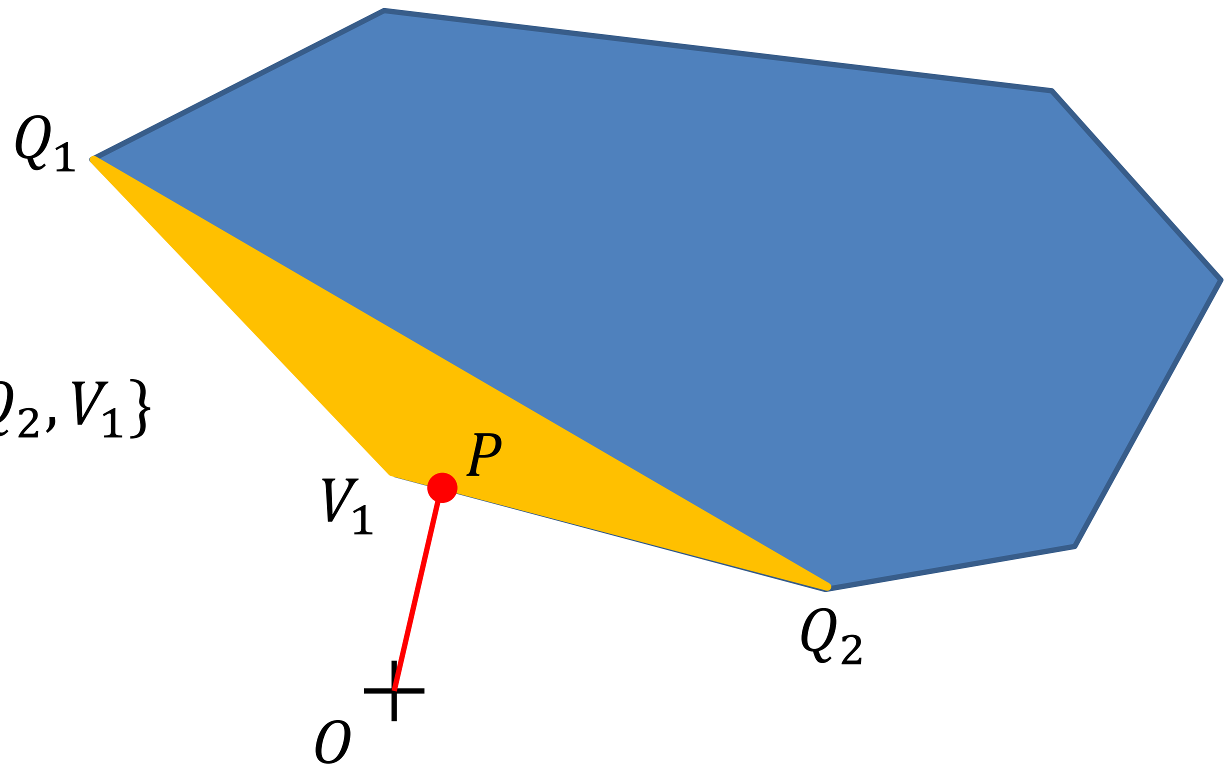
$Q = \{Q_1, Q_2, V_1\}$



# GJK algorithm example

3. Compute the point $P$ of minimum norm of the convex hull $CH(Q)$

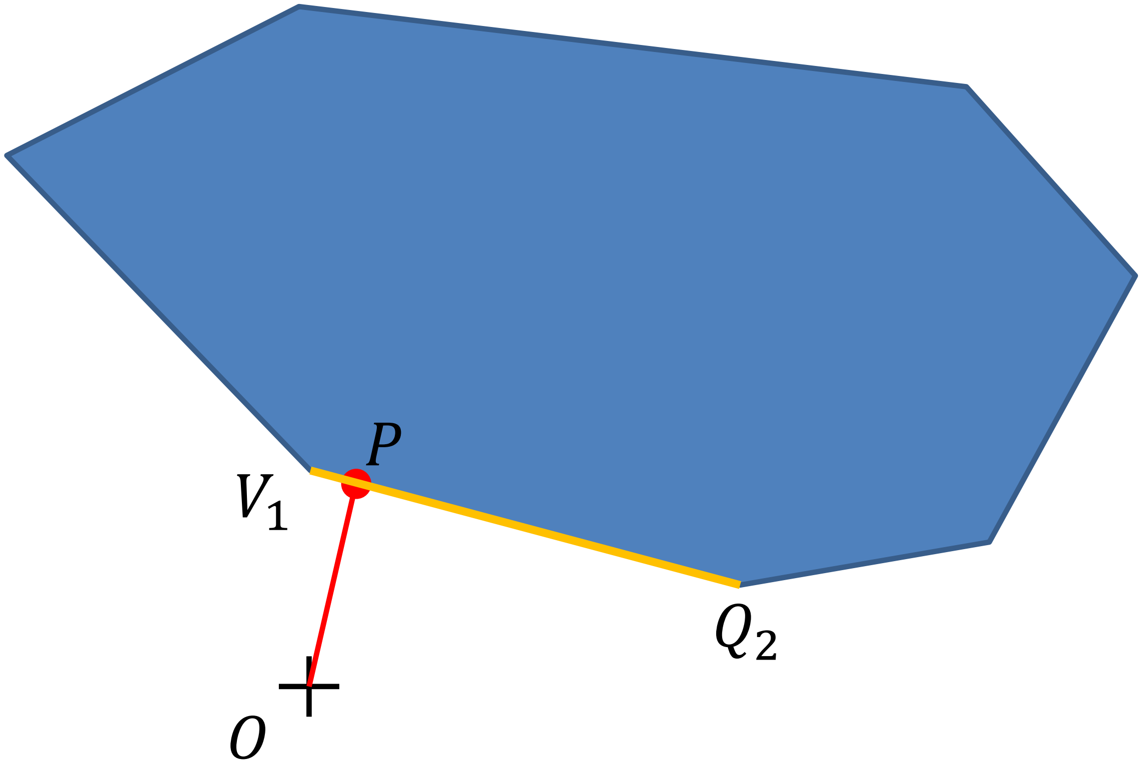
$Q = \{Q_1, Q_2, V_1\}$



# GJK algorithm example

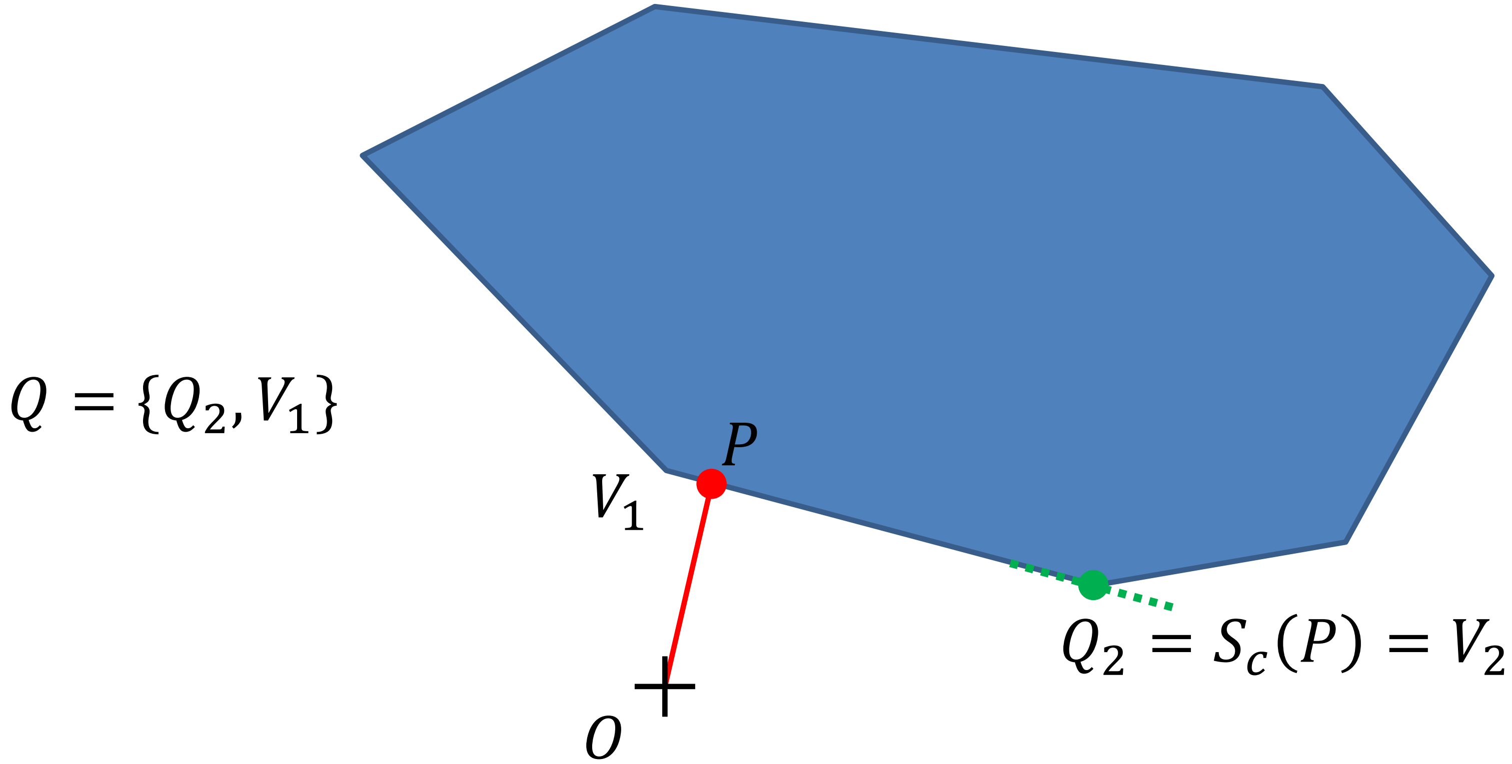
4. Reduce $Q$ to the smallest subset $Q'$ of $Q$ such that $P \in CH(Q')$

$Q = \{Q_2, V_1\}$



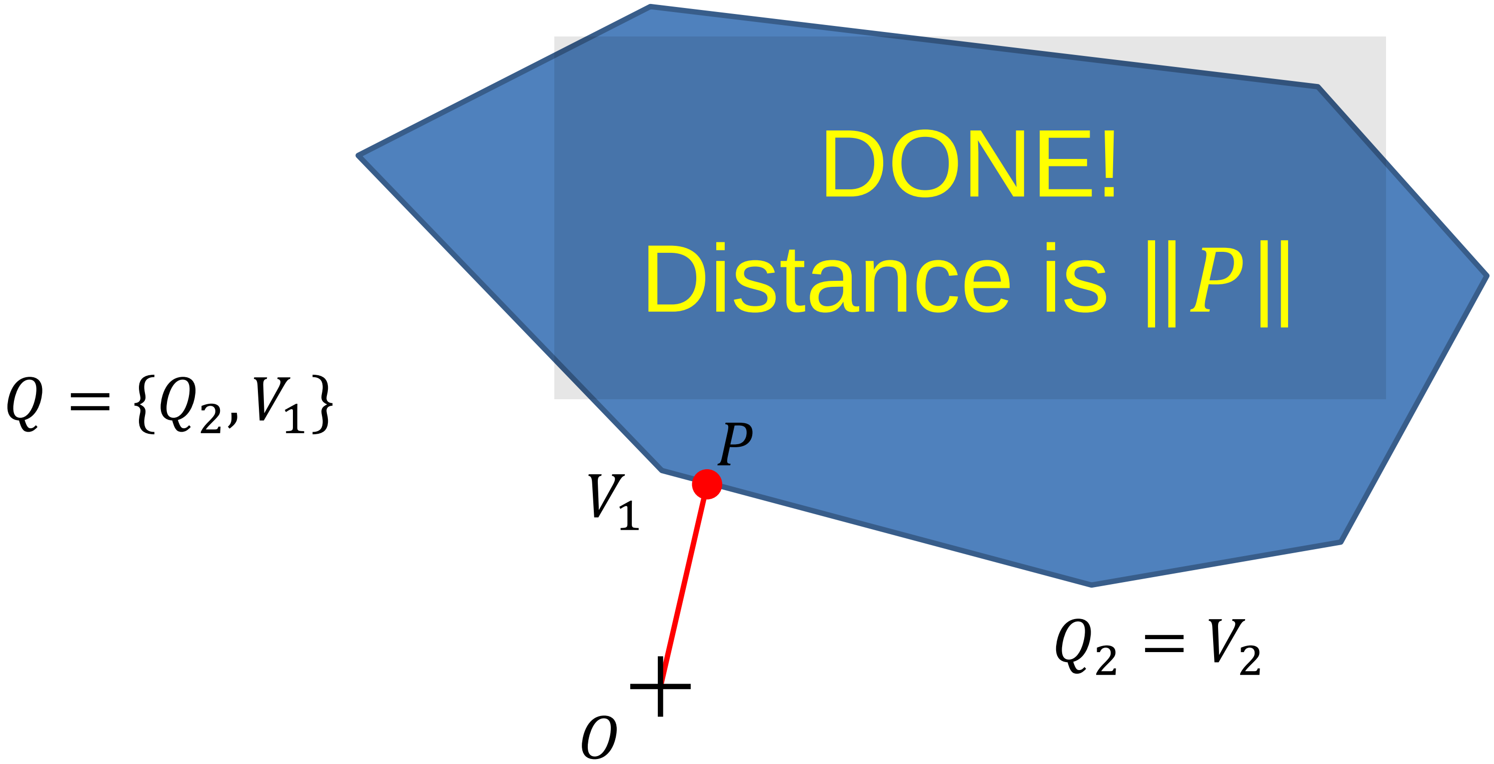
# GJK algorithm example

5. Let $V = S_c(-P)$ be a supporting point in direction $-P$. Let's call it $V_2$.



# GJK algorithm example

6. If $V$ is no more extreme than $P$ in direction $-P$, then return $||P||$



# Supporting point

- In step 5 we had to find the supporting point of $C$ in the direction $-P$
- It was intuitive an our example but how can we automatically calculate that point in any given situation?
  - we need the actual definition of a supporting point

# Supporting point

- A supporting point $V$ of a convex set $C$ in a direction $d$ is one of the most distant points along $d$
- In other words $V$ is a supporting point if

$$d \cdot V = \max\{d \cdot X : X \in C\}$$

  - that is, $V$ is a point for which $d \cdot V$ (its projection on $V$) is maximal
  - supporting points are sometimes called extreme points, and are not necessarily unique
  - for a polytope, one of the vertices can always be selected as a supporting point for a given direction

# Support mapping

- A support mapping $S_C(d)$ is a function that maps the direction $d$ into a supporting point of $C$
- For simple convex shapes, support mappings can be given in closed form
  - Sphere centered at $c$ of radius $r$

$$S_C(d) = c + r\frac{d}{||d||}$$

  - AABB centered at $c$ with size $2e_x \times 2e_y \times 2e_z$

$$S_C(d) = c + \left(sign(d_x)e_x, sign\left(d_y\right)e_y, sign(d_z)e_z\right)$$

    where $sign(\alpha) = -1$ if $\alpha < 0$ and 1 otherwise
  - Formulas exist for cylinder, cone *etc.*

# Support mapping

6.1

- Convex shapes of higher complexity require the support mapping function to determine a support point using numerical methods
- For a polytope of $n$ vertices, a supporting vertex is trivially found in $O(n)$ by searching over all vertices
- A greedy algorithm can be used to optimize the search by exploring the polytope through a simple hill-climbing algorithm (using the $d \cdot X_i$ values)
  - with extra optimizations we can design an algorithm in $O(\log n)$
  - we can also use frame coherency for determining the starting point, and then in practice we observe a performance almost insensitive to the complexity of the objects!

# Collision detection algorithm

- Remember the collision detection algorithm
  - Broad phase
    - disregard pairs of objects that cannot collide
    - ➢ model and space partitioning
  - Mid phase
    - determine potentially colliding primitives
    - ➢ movement bounds
  - Narrow phase
    - determine exact contact between two shapes
    - ➢ Gilbert-Johnson-Keerthi algorithm

# End of
# Collision detection

Next

Collision resolution