# GDC 09

learn
network
inspire

**www.GDConf.com**

Game Developers Conference®

**March 23-27, 2009** | Moscone Center, San Francisco

# Physics for Games Programmers: Collision Detection Crash Course

Gino van den Bergen
gino@dtecta.com

# Collision Detection

- Track which pairs of objects…

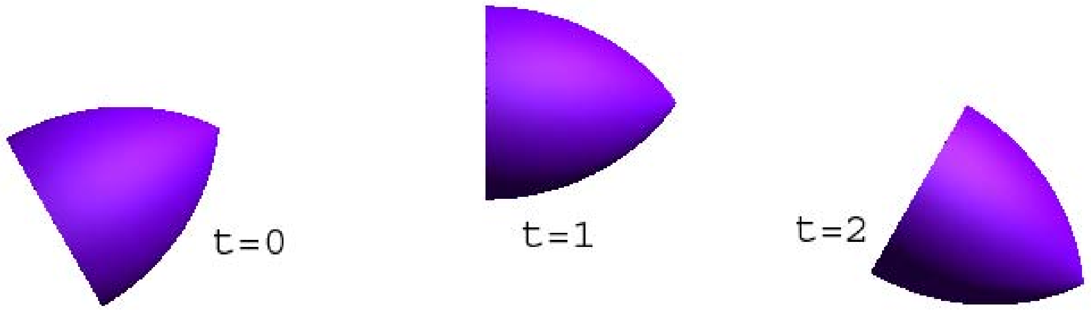
  are interpenetrating now, or rather

  will collide over the next frame if no counter action is taken.

- Compute data for response.

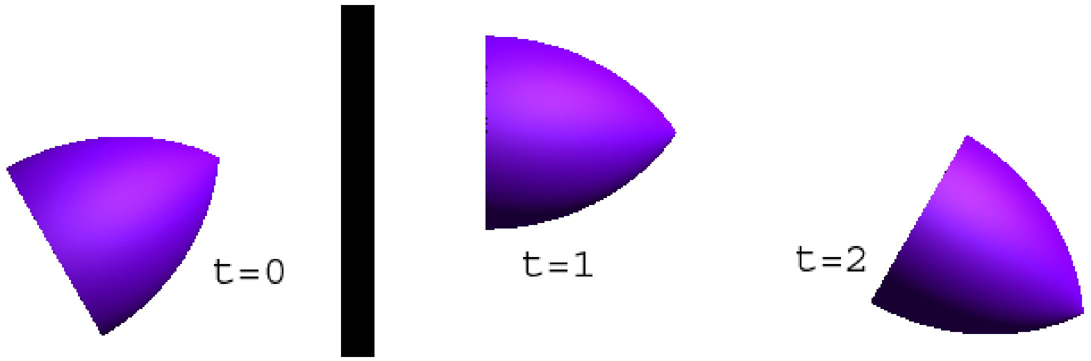
# The Problem

- » Object placements are computed for discrete moments in time.
- » Object trajectories are assumed to be continuous.



# The Problem (cont'd)

- If collisions are checked only for the sampled moments, some collisions are missed (tunneling).
- Humans easily spot such artifacts.



# The Fix

- Perform collision detection in continuous 4D space-time:

  Construct a plausible trajectory for each moving object.

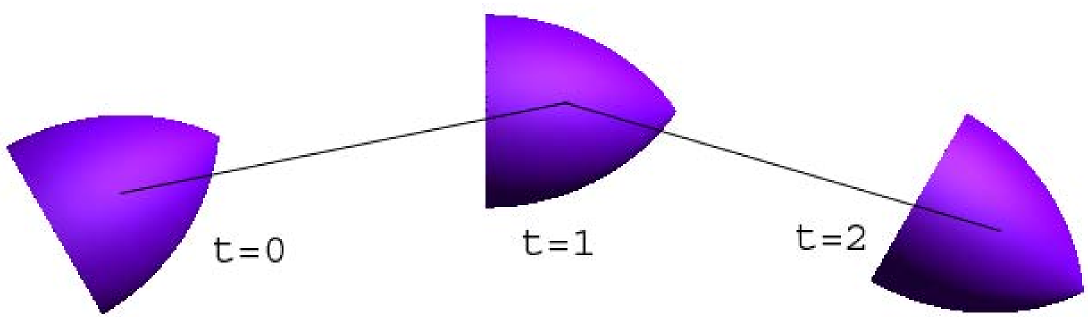
  Check for collisions along these trajectories.

# Plausible Trajectory?

- Use of physically correct trajectories in real-time 4D collision detection is something for the not-so-near future.
- In game development real-time constraints are met by cheating.
- We cheat by using simplified trajectories.

# Plausible Trajectory? (cont'd)

- » Limited to trajectories with piecewise constant linear velocities.
- » Angular velocities are ignored. Rotations are considered instantaneous.



# Physical Representation

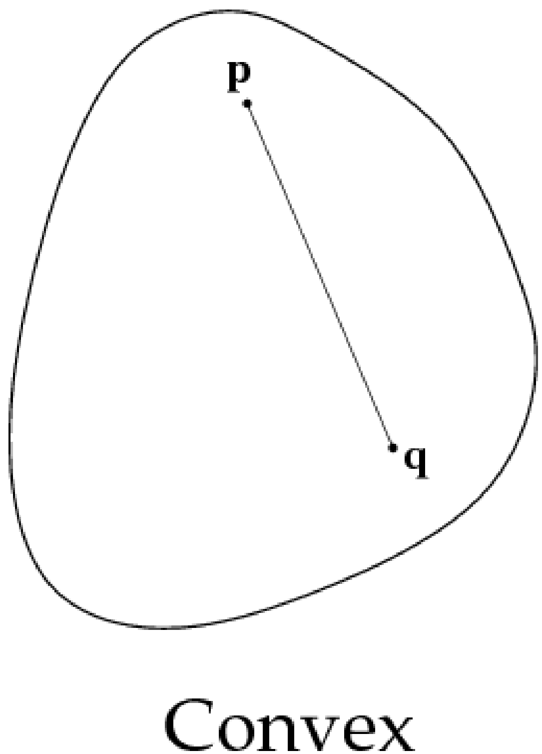
- Scenes may be composed of many independently moving objects.
- Objects may be composed of many primitives.
- Different types of primitives may be used (triangles, spheres, boxes, cylinders, capsules, andwhatnot).

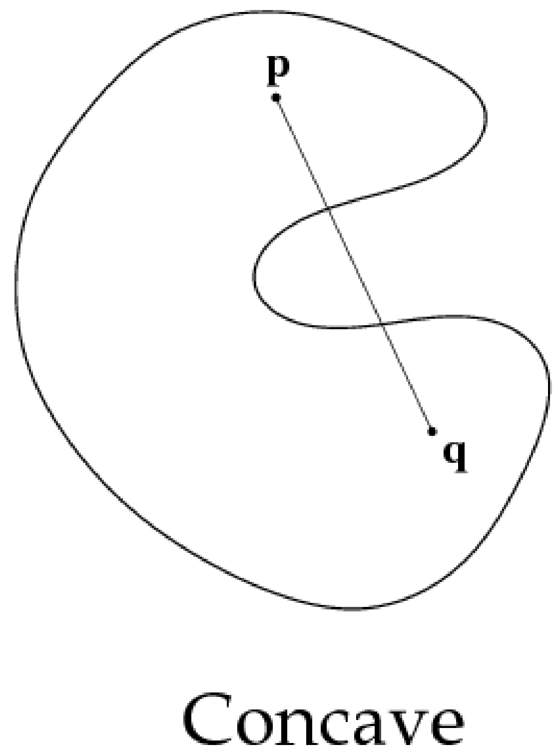
# Three Phases

» **Broad phase**: Determine all pairs of objects that potentially collide.
» **Mid phase**: Determine potentially colliding primitives of a pair of objects.
» **Narrow phase**: Determine contact between primitives and compute response data.

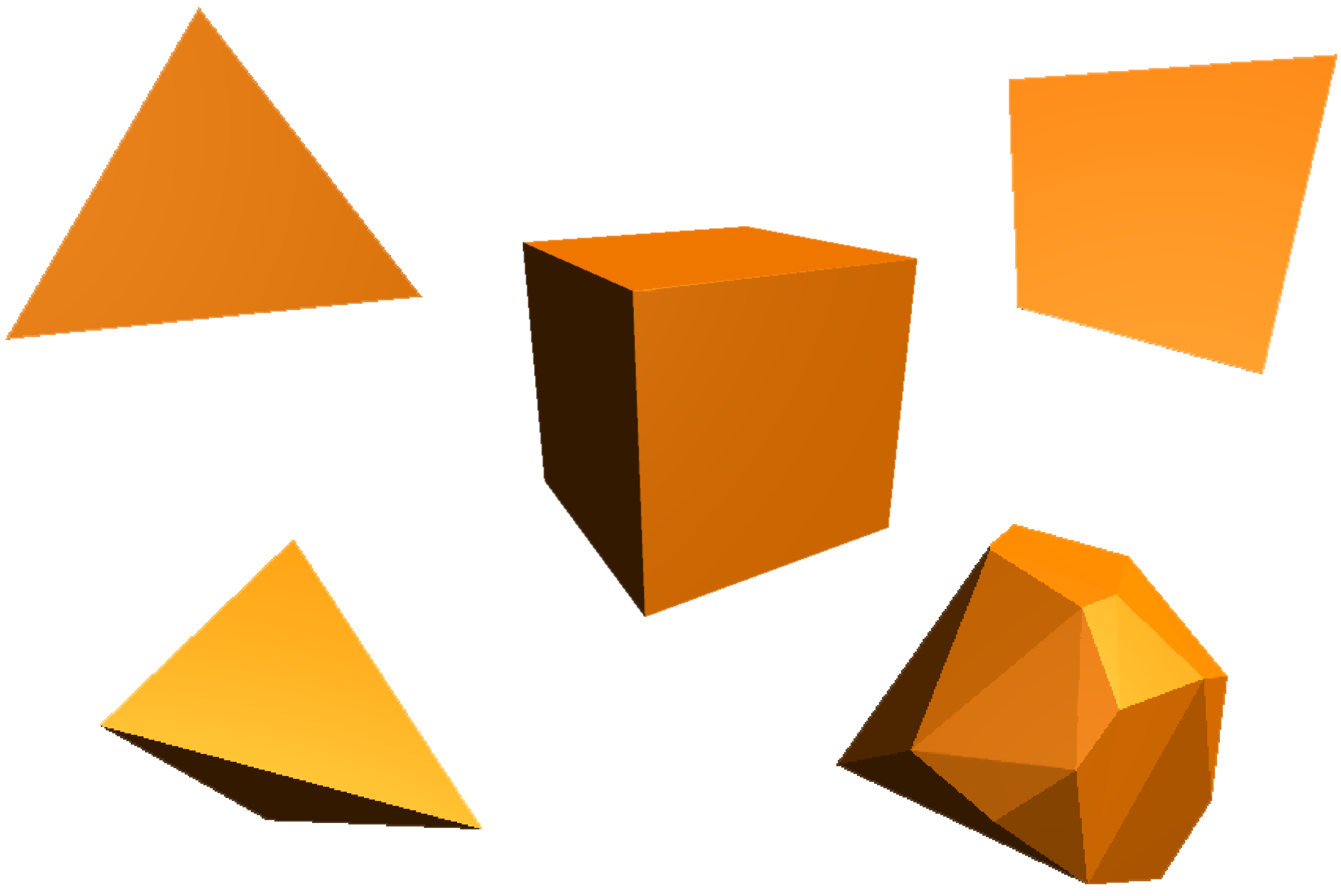
# Primitives

» Only convex shapes are considered

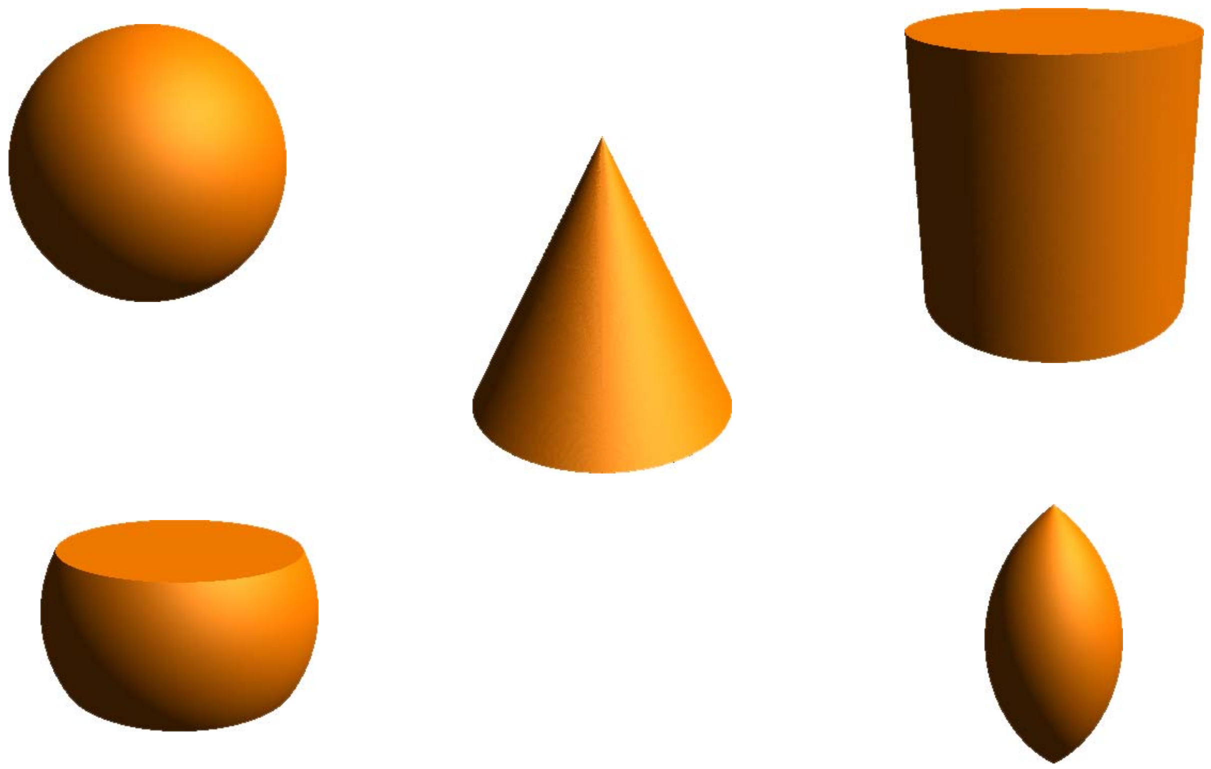


# Polytopes



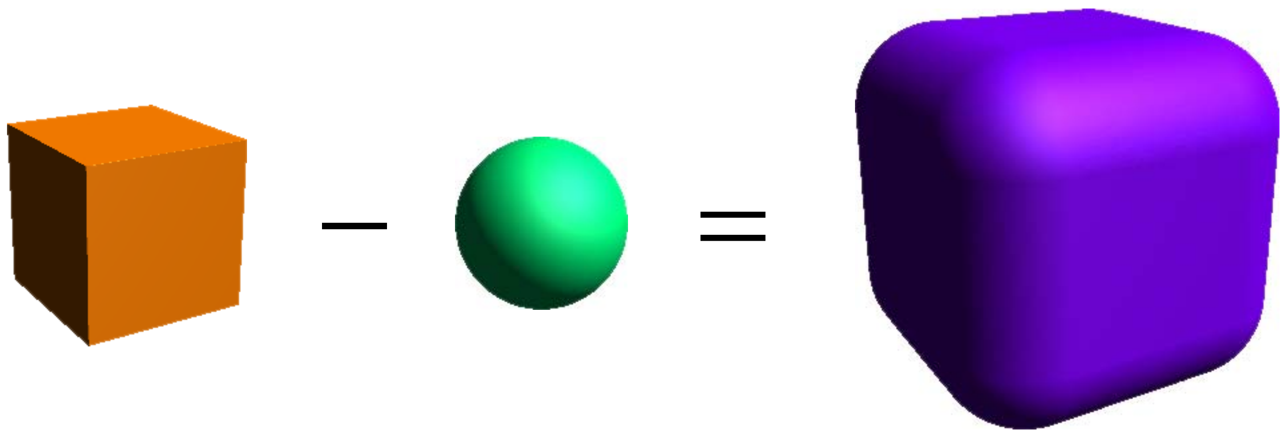
# Quadric Shapes



# Configuration Space

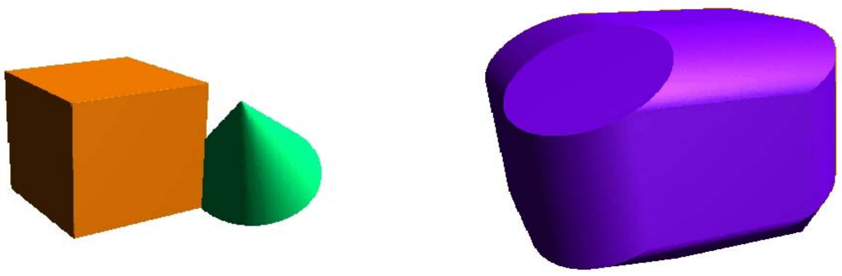
- The *configuration space obstacle* of objects $A$ and $B$ is the set of all vectors from a point of $B$ to a point of $A$.

$$A - B = \{\mathbf{a} - \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$$



# Translation

- Translation of $A$ and/or $B$ results in a translation of $A - B$.



# Rotation

- Rotation of $A$ and/or $B$ changes the shape of $A - B$.

# Configuration Space (cont'd)

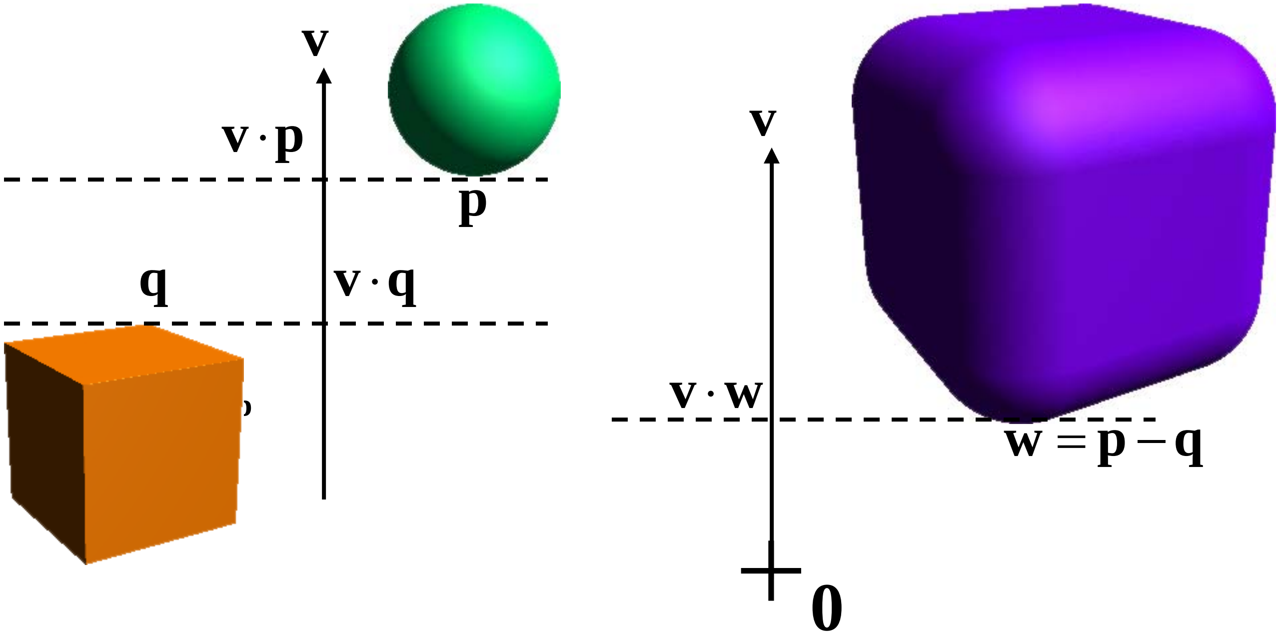
» $A$ and $B$ intersect: $A - B$ contains origin.

$$A \cap B \neq \emptyset \Leftrightarrow \mathbf{0} \in A - B$$

» Distance between $A$ and $B$: length of shortest vector in $A - B$.

$$d(A, B) = \min\left\{\|\mathbf{x}\| : \mathbf{x} \in A - B\right\}$$

# Separating Axis



# Separating Axis Theorem

- Any pair of non-intersecting polytopes has a separating axis that is orthogonal to:
- a face of either polytope, or
- an edge from each polytope.

# Separating Axis Theorem Proof (or at least a sketch)

- The CSO of non-intersecting polytopes is a polytope that does not contain the origin.
- The origin lies on the outside of at least one face of the CSO.
- A face of the CSO is either the CSO of a face and a vertex or the CSO of two edges.

# Separating Axis Method

- Test all face normals and all cross products of edge directions.
- If none of these vectors yields a separating axis then the polytopes must intersect.
- Given polytopes with resp. $f_1$ and $f_2$ faces and $e_1$ and $e_2$ edge directions, we need to test at most $f_1 + f_2 + e_1 \, e_2$ axes.

# Separating Axis Method

| Polytope 1 | Polytope 2 | #Axes |
|---|---|---|
| Line segment | Triangle | $0 + 1 + 3 = 4$ |
| Line segment | Box | $0 + 3 + 3 = 6$ |
| Triangle | Triangle | $1 + 1 + 9 = 11$ |
| Triangle | Box | $1 + 3 + 9 = 13$ |
| Box | Box | $3 + 3 + 9 = 15$ |

# GJK Algorithm

- An iterative method for computing the distance between convex objects.
- First publication in 1988 by Gilbert, Johnson, and Keerthi.
- Solves queries in configuration space.
- Uses an implicit object representation.

# GJK Algorithm: Workings

- Approximate the point of the CSO closest to the origin
- Generate a sequence of simplices inside the CSO, each simplex lying closer to the origin than its predecessor.
- A *simplex* is a point, a line segment, a triangle, or a tetrahedron.

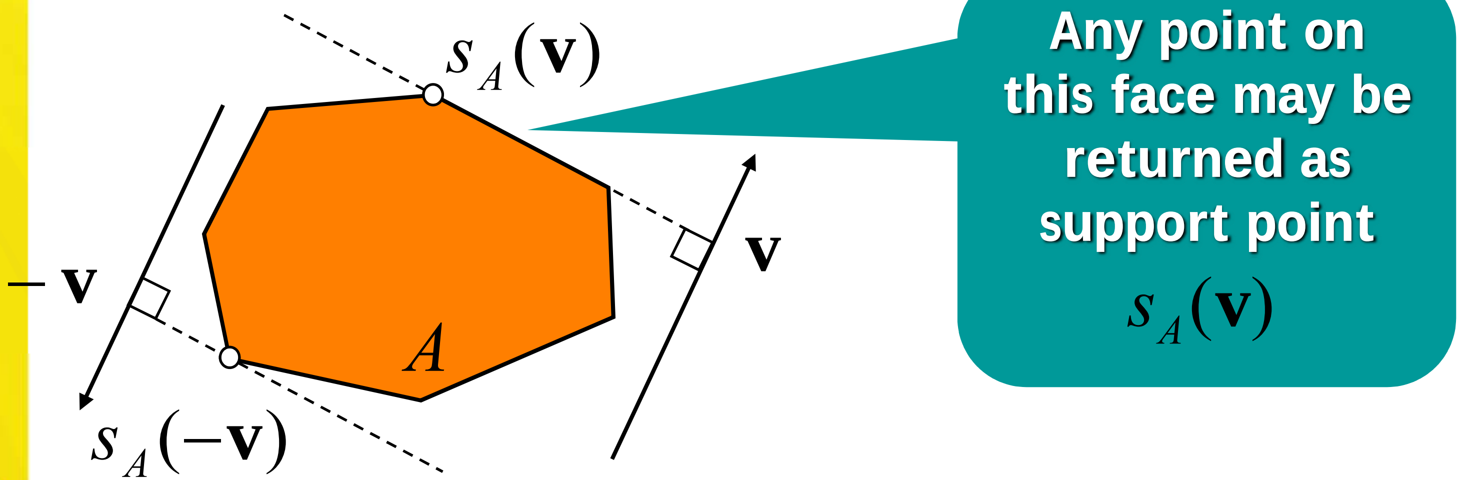
# GJK Algorithm: Workings (cont'd)

- Simplex vertices are computed using support mappings. (Definition follows.)
- Terminate as soon as the current simplex is close enough.
- In case of an intersection, the simplex contains the origin.

# Support Mappings

» A support mapping $s_A$ of an object $A$ maps vectors to points of $A$, such that

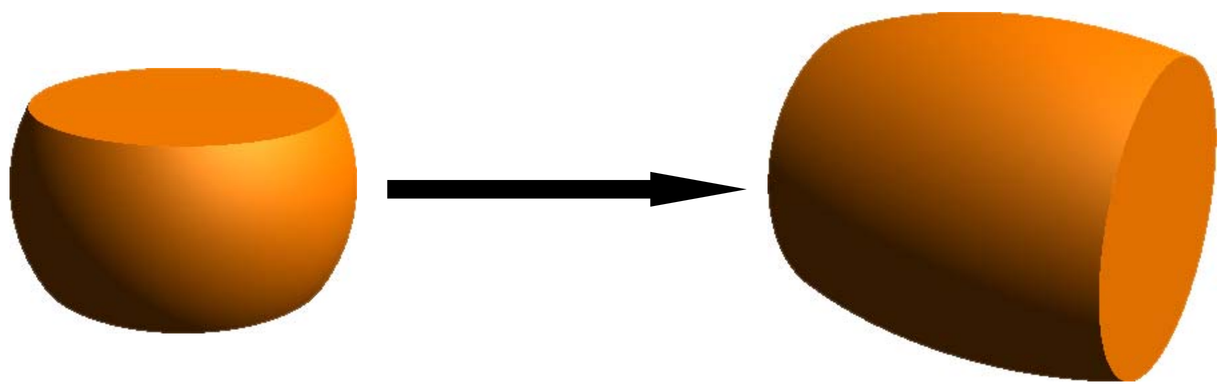
$$\mathbf{v} \cdot s_A(\mathbf{v}) = \max\left\{\mathbf{v} \cdot \mathbf{x} : \mathbf{x} \in A\right\}$$



# Affine Transformation

» Shapes can be translated, rotated, *and* scaled. For $\mathbf{T}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{c}$, we have

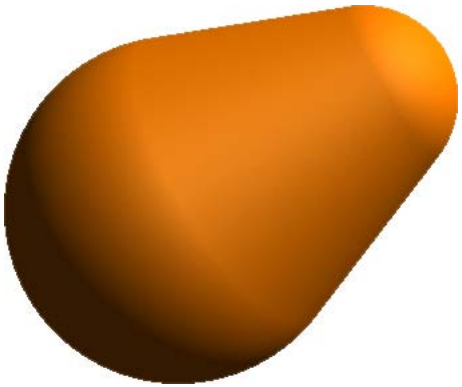
$$s_{\mathbf{T}(A)}(\mathbf{v}) = \mathbf{T}(s_A(\mathbf{B}^T\mathbf{v}))$$

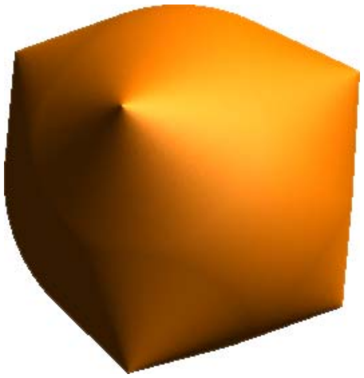


# Convex Hull

» Convex hulls of arbitrary convex shapes are readily available.

$$s_{\mathrm{conv}\{X_0,\ldots,X_{n-1}\}}(\mathbf{v}) = s_{\{s_{X_0}(\mathbf{v}),\ldots,s_{X_{n-1}}(\mathbf{v})\}}(\mathbf{v})$$



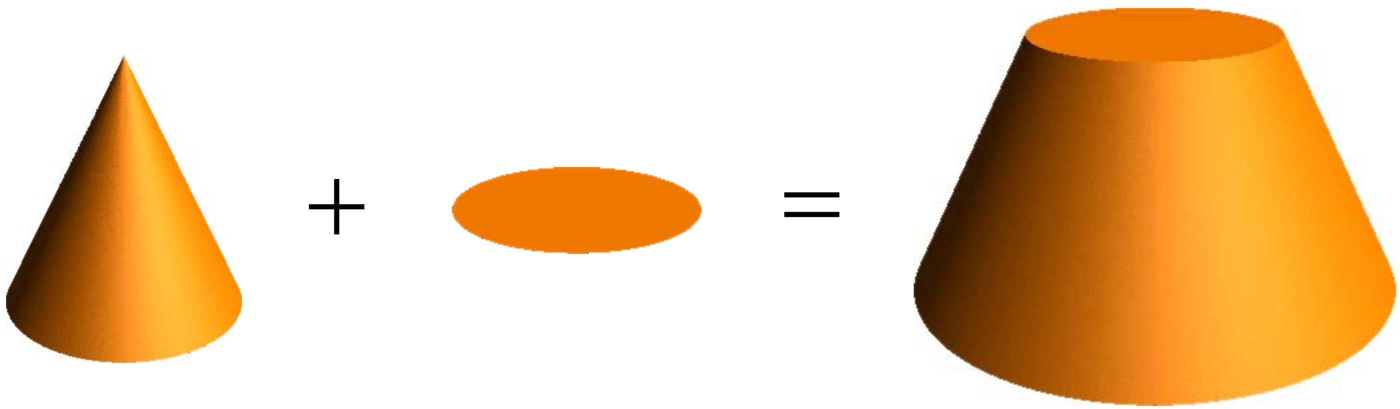


# Minkowski Sum

- Shapes can be fattened by Minkowski addition.

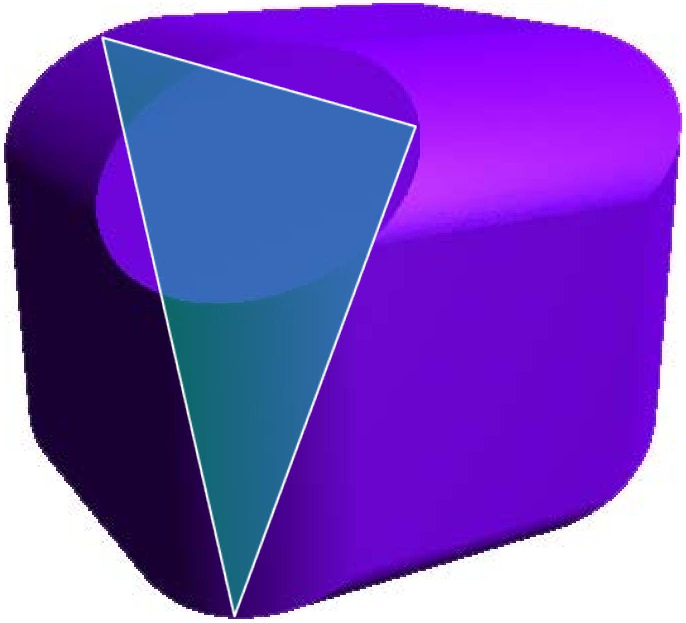
$$s_{A+B}(\mathbf{v}) = s_A(\mathbf{v}) + s_B(\mathbf{v})$$

$$s_{A-B}(\mathbf{v}) = s_A(\mathbf{v}) - s_B(-\mathbf{v})$$



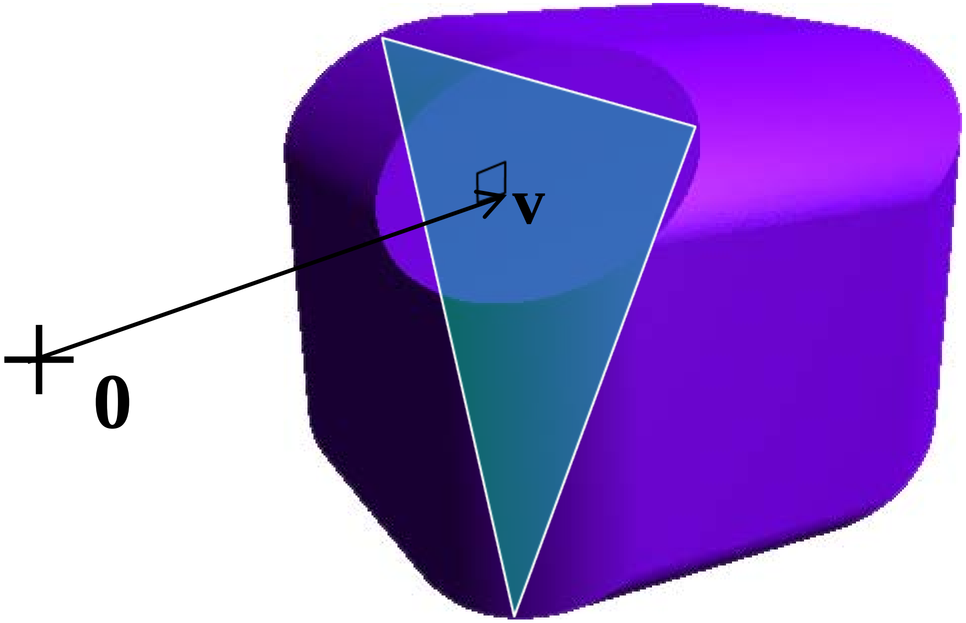
# Basic Steps (1/6)

» Suppose we have a simplex inside the CSO…



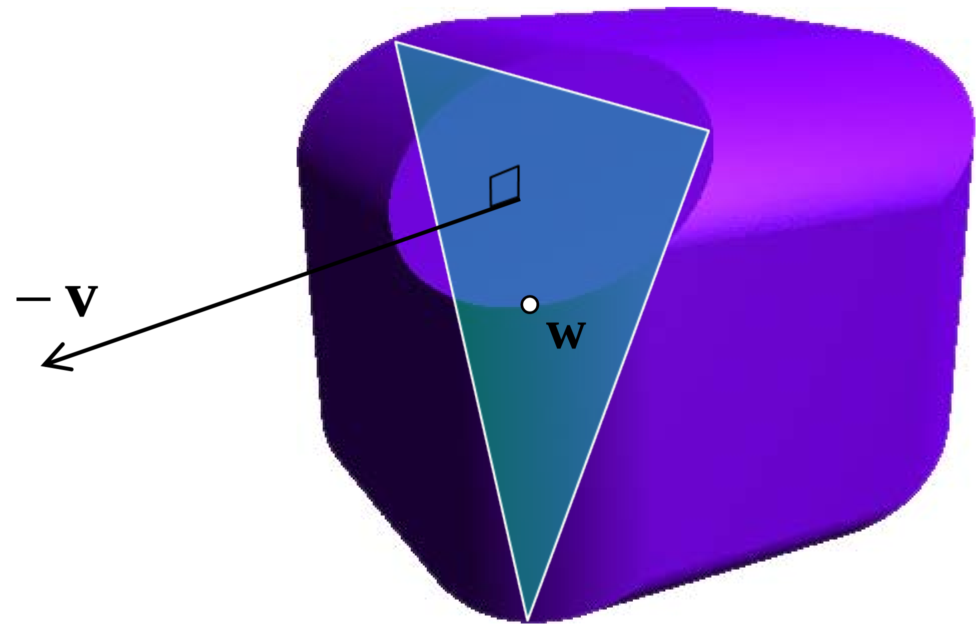
# Basic Steps (2/6)

» …and the point **v** of the simplex closest to the origin.



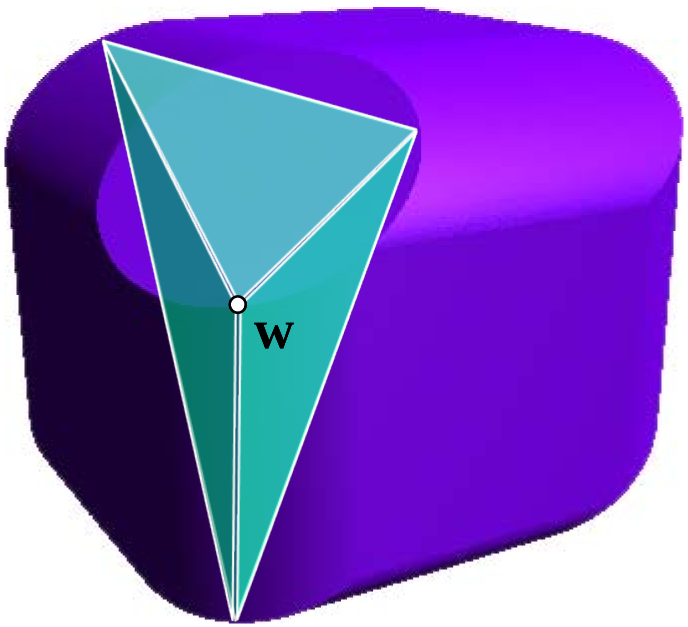
# Basic Steps (3/6)

» Compute support point $\mathbf{w} = s_{A\text{-}B}(-\mathbf{v})$.



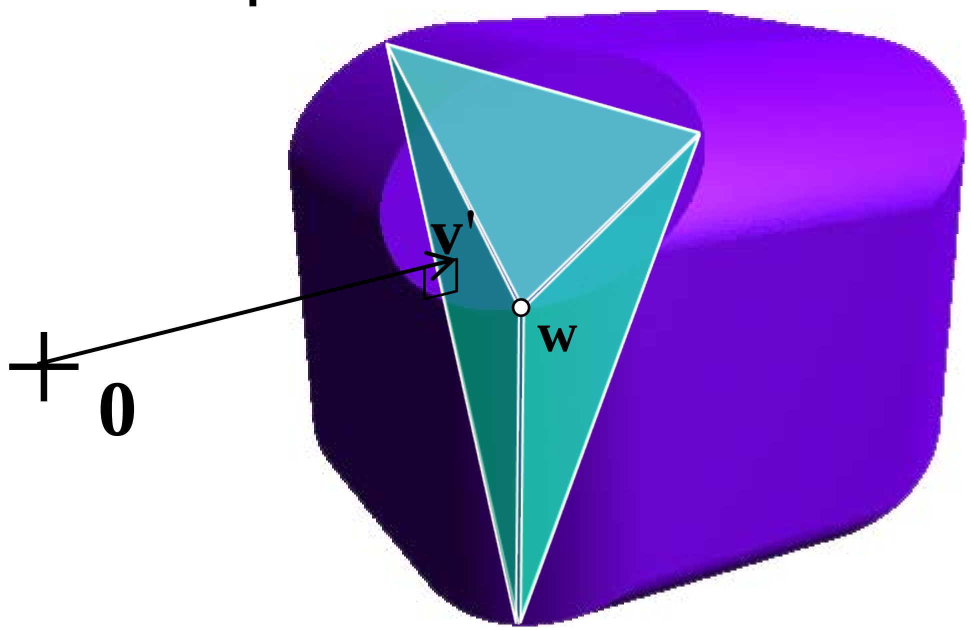
# Basic Steps (4/6)

» Add support point $\mathbf{w}$ to the current simplex.



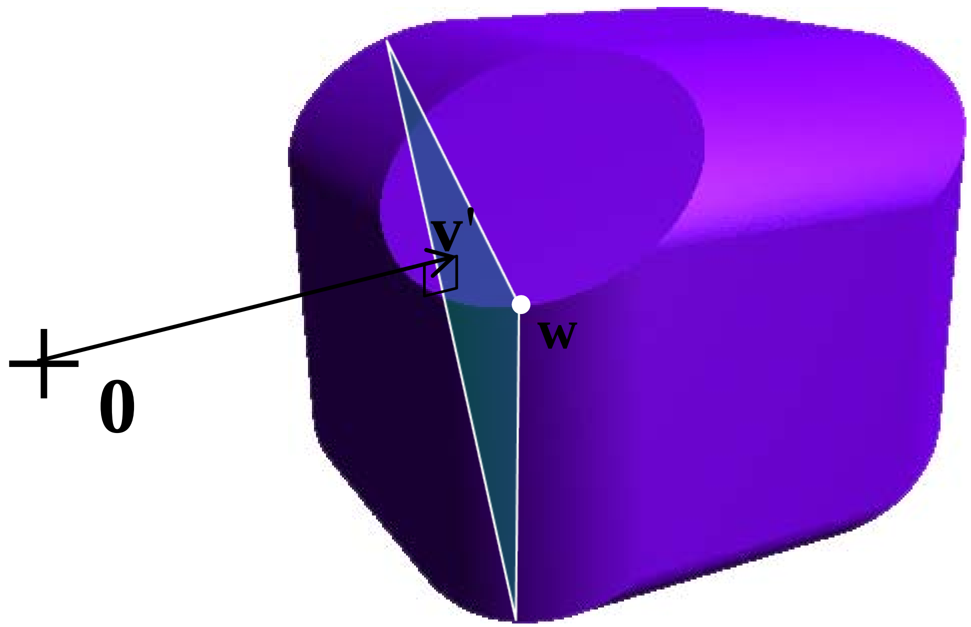
# Basic Steps (5/6)

» Compute the closest point **v'** of the new simplex.



# Basic Steps (6/6)

- Discard all vertices that do not contribute to **v'**.



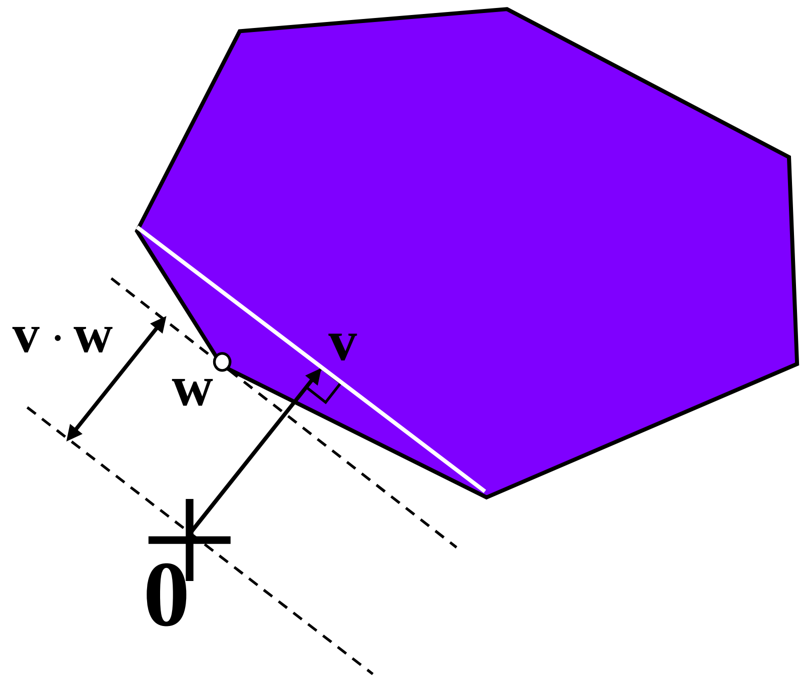
# Separating Axis

» If only an intersection test is needed then let GJK terminate as soon as the lower bound $\mathbf{v} \cdot \mathbf{w}$ becomes positive.

» For a positive lower bound $\mathbf{v} \cdot \mathbf{w}$, the vector $\mathbf{v}$ is a separating axis.

# Separating Axis (cont'd)

» The supporting plane through $\mathbf{w}$ separates the origin from the CSO.



# Separating Axes and Coherence

- Separating axes can be cached and reused as initial $\mathbf{v}$ in future tests on the same object pair.
- When the degree of frame coherence is high, the cached $\mathbf{v}$ is likely to be a separating axis in the new frame as well.
- An incremental version of GJK takes roughly one iteration per frame for smoothly moving objects.

# Shape Casting

» Find the earliest time two translated objects come in contact.

» Boils down to performing a ray cast in the objects' configuration space.

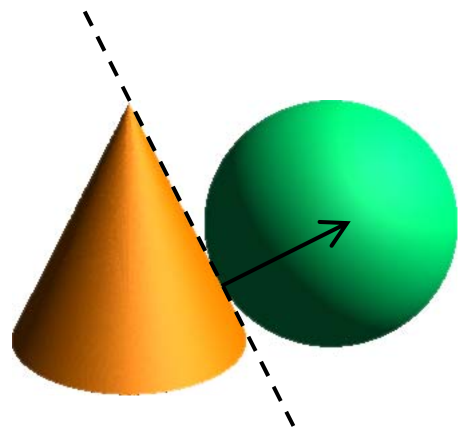
» For objects $A$ and $B$ being translated over respectively vectors $\mathbf{s}$ and $\mathbf{t}$, we perform a ray cast along the vector $\mathbf{r} = \mathbf{t} - \mathbf{s}$ onto $A - B$.

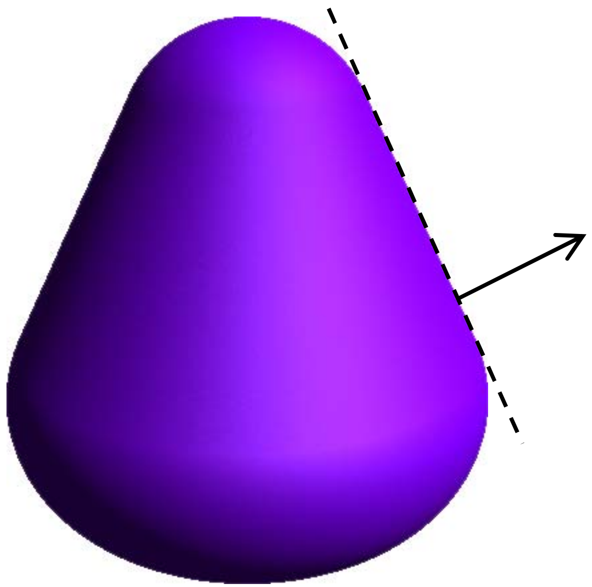
» The earliest time of contact is

$$\min\{ \lambda : \lambda\mathbf{r} \in A - B, 0 \le \lambda \le 1 \}$$

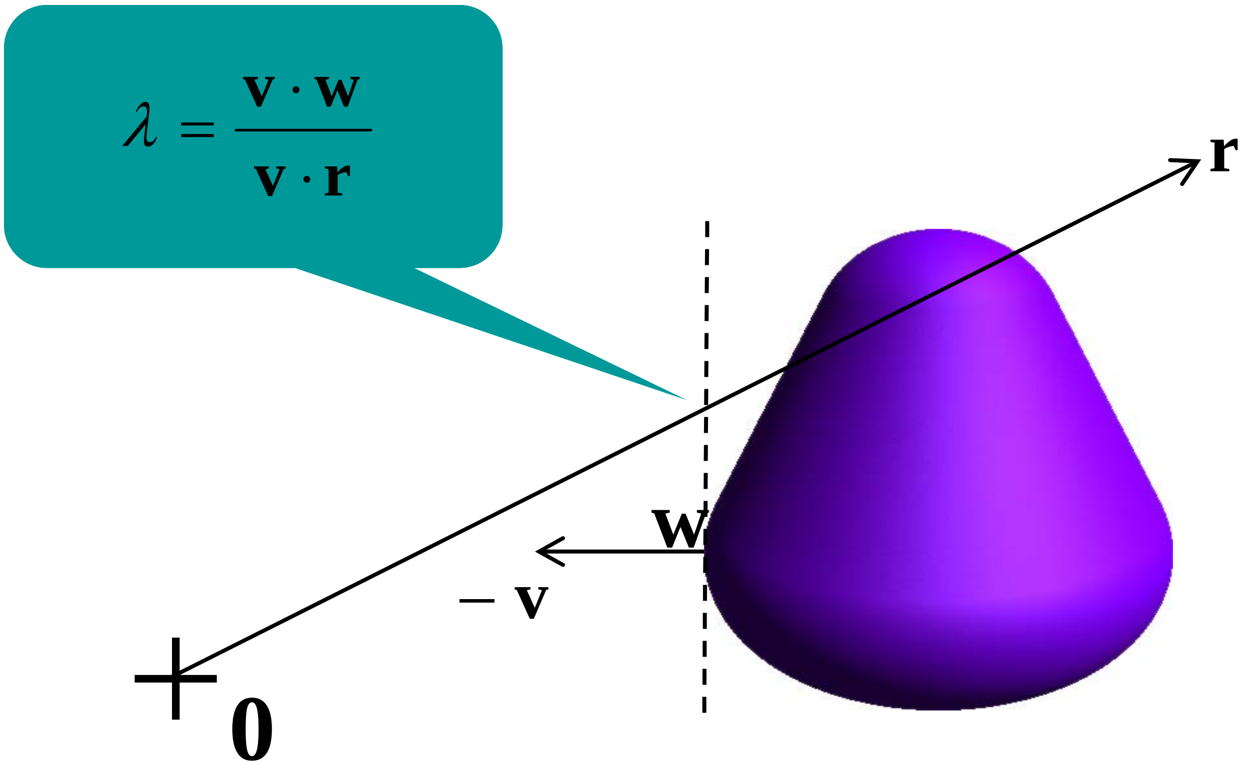
# Normals

» A normal at the hit point of the ray is normal to the contact plane.





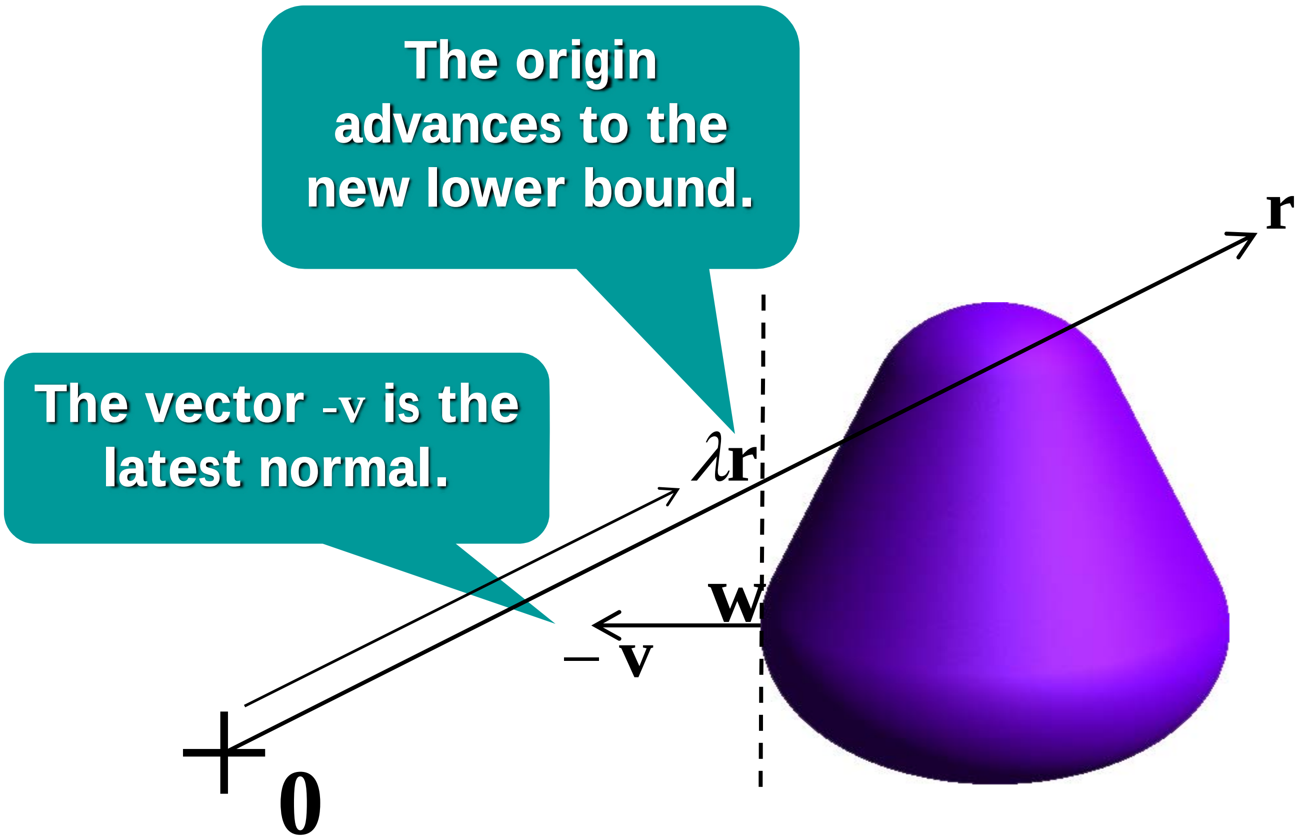
# Ray Clipping



# GJK Ray Cast

- Do a standard GJK iteration, and use the support planes as clipping planes.
- Each time the ray is clipped, the origin is "shifted" to $\lambda\mathbf{r}$,…
- …and the current simplex is set to the last-found support point.
- The vector $-\mathbf{v}$ that corresponds to the latest clipping plane is the normal at the hit point.

# GJK Ray Cast (cont'd)

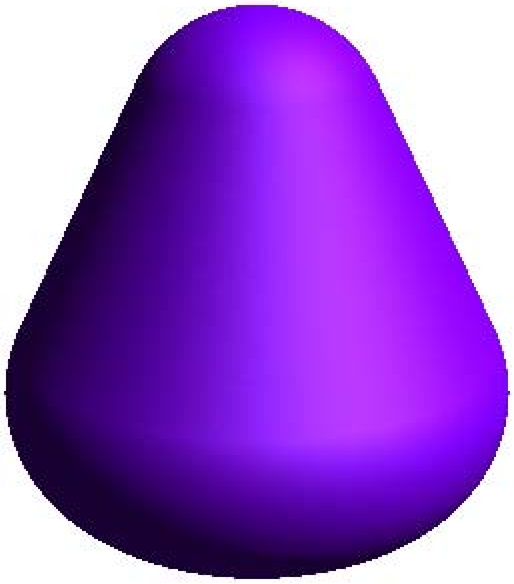


# Accuracy vs. Performance

- Accuracy can be traded for performance by tweaking the error tolerance $\varepsilon_{tol}$.
- A greater tolerance results in fewer iterations but less accurate hit points and normals.

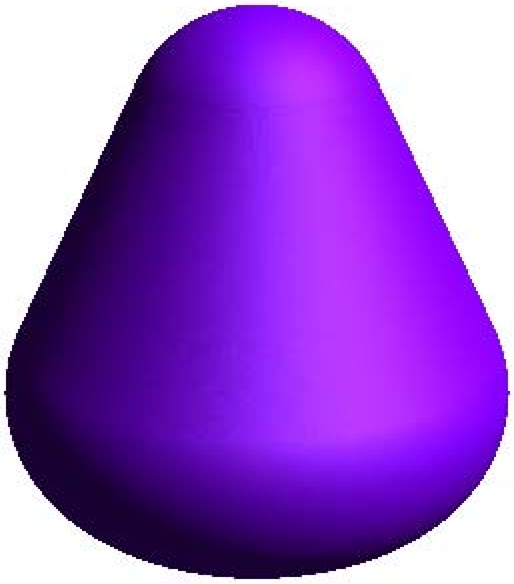
# Accuracy vs. Performance

» $\varepsilon_{tol} = 10^{-7}$, avg. time: 3.65 μs @ 2.6 GHz



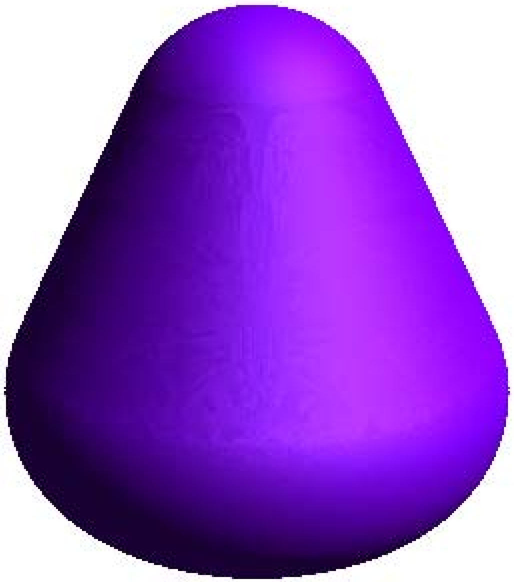
# Accuracy vs. Performance

» $\varepsilon_{tol} = 10^{-6}$, avg. time: 2.80 μs @ 2.6 GHz



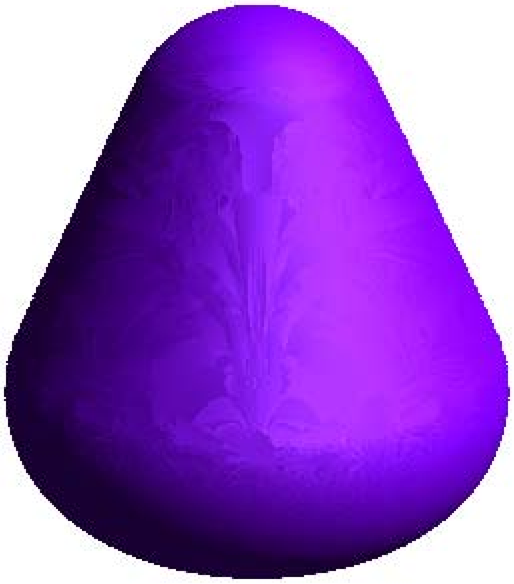
# Accuracy vs. Performance

» $\varepsilon_{tol} = 10^{-5}$, avg. time: 2.03 μs @ 2.6 GHz



# Accuracy vs. Performance

- $\varepsilon_{tol} = 10^{-4}$, avg. time: 1.43 μs @ 2.6 GHz



# Accuracy vs. Performance

» $\varepsilon_{tol} = 10^{-3}$, avg. time: 1.02 µs @ 2.6 GHz

# Accuracy vs. Performance

» $\varepsilon_{tol} = 10^{-2}$, avg. time: 0.77 μs @ 2.6 GHz

# Accuracy vs. Performance

- $\varepsilon_{tol} = 10^{-1}$, avg. time: 0.62 µs @ 2.6 GHz

# GJK Algorithm: Pros

- Extremely versatile:
  - Applicable to any combination of convex shape types.
  - Computes distances, common points, and separating axes.
  - Can be tailored for finding space-time collisions.
  - Allows a smooth trade-off between accuracy and speed.

# GJK Algorithm: Pros (cont'd)

- Performs well:
  - Exploits frame coherence.
  - Competitive with dedicated solutions for polytopes (Lin-Canny, V-Clip, SWIFT) .
- Despite its conceptual complexity, implementing GJK is not too difficult.
- Small code size.

# GJK Algorithm: Cons

- Difficult to grasp:
  - Concepts from linear algebra and convex analysis (determinants, Minkowski addition), take some time to get comfortable with.
  - Maintaining a “geometric” mental image of the workings of the algorithm is challenging and not very helpful.

# GJK Algorithm: Cons (cont'd)

- Suffers from numerical issues:
  - Termination is governed by predicates that rely on tolerances.
  - Despite the use of tolerances, certain “hacks” are needed in order to guarantee termination in all cases.
  - Using 32-bit floating-point numbers is doable but tricky.

# Mid Phase: BV Trees

- Used for objects composed of lots of primitives, such as triangle meshes.
- Aim is to quickly reject groups of primitives based on geometric locality.
- ‘Capture’ locality by constructing a hierarchy of bounding volumes.

# Bounding Volumes

- Should fit the model as tightly as possible.
- Overlap tests between volumes should be cheap.
- Should use a small amount of memory.
- Cost of computing the best-fit bounding volume should be low.

# Bounding Volumes

» Good bounding volume types are:
  - Spheres
  - Axis-aligned bounding boxes (AABBs)
  - Discrete-orientation polytopes (*k*-DOPs)
  - Oriented bounding boxes (OBBs)

# Bounding Volume Types

| | Fit | Test (ops) | Memory (scalars) | Best-fit Cost |
|---|---|---|---|---|
| Sphere | poor | $11$ | $4$ | high |
| AABB | fair | $\leq 6$ | $6$ | low |
| *k*-DOP | good | $\leq 2k$ | $2k$ | medium |
| OBB | excellent | $\leq 200$ | $15$ | high |

# Why AABBs?

- Offer a fair trade-off between storage space and fit.
- Overlap tests are fast.
- Allow for fast construction and update of bounding volumes.
- Storage requirements can be further reduced through compression.

# AABB Tree Construction

1. Compute the AABB of the set of primitives.
2. Split the set using the plane that cuts the longest axis of the AABB in two equal halves.
3. Primitives that straddle the plane are added to the dominant side. (AABBs of the two sets may overlap.)
4. Repeat from step 1 for the split sets.
5. Continue until all sets contain one primitive.

# Test Primitive vs. AABB Tree

- Compute the primitive's AABB in the AABB tree's local coordinate system.
- Recursively visit all nodes whose AABBs overlap the primitive's AABB.
- Test each visited leaf's primitive against the query primitive.

# Test Oriented AABB Trees

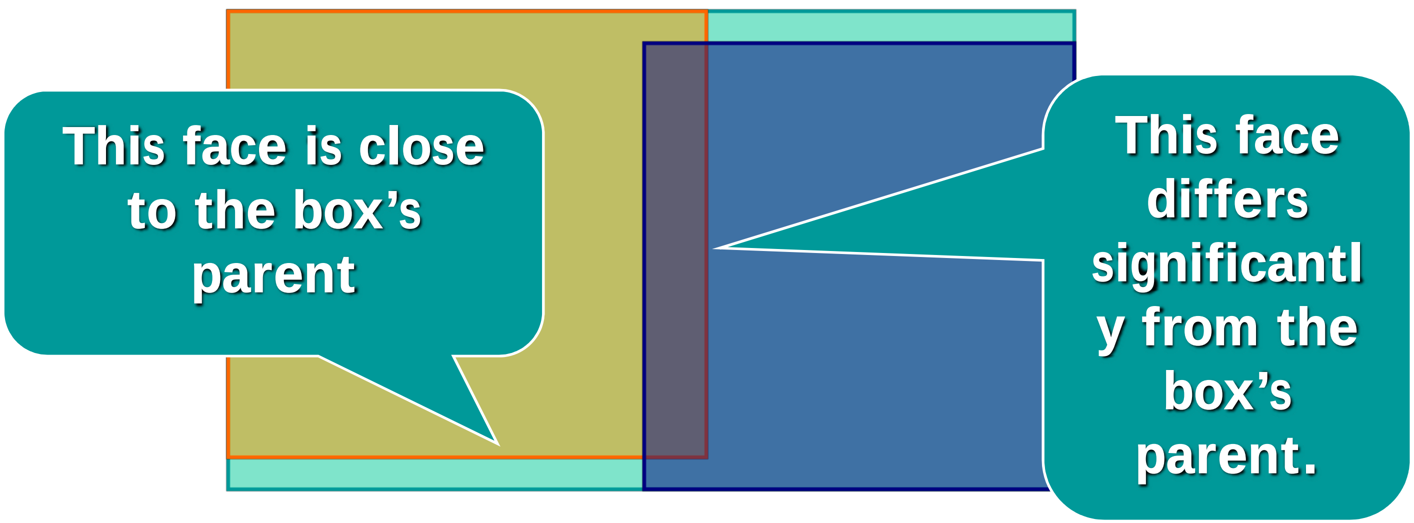
- Simultaneously descend in both trees.
- Requires an oriented-box test such as the SAT (lite).
- (SAT lite only tests the 6 face normals.)
- Always unfold the largest of the current two AABBs.
- If both sub-trees are leaves, perform a primitive-primitive test.

# Updating AABB Trees

- AABB trees can be updated rather than recomputed for deformable meshes.
- First recompute the AABBs of the leaves.
- Work your way back to the root: A parent's box is the AABB of the children's boxes.

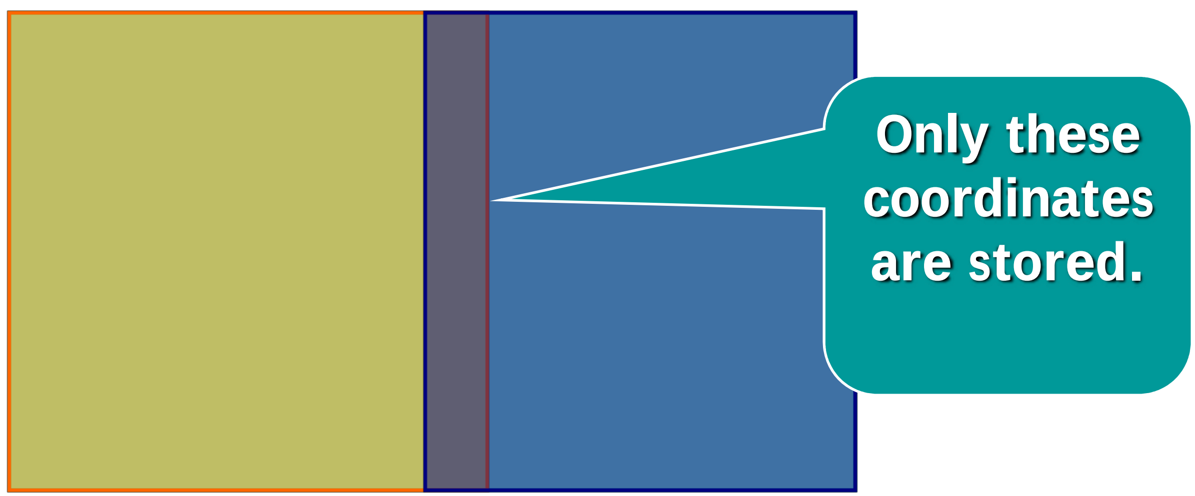
# Boxtree [Zachmann]

» Since the set of primitives is split along the longest axis, only one of each child's faces will differ significantly from its parent's.



# Boxtree [Zachmann]

- Store only the coordinate for the inner faces (similar to $k$-d tree.)
- The other coordinates are inherited from the parent box.



# Boxtree [Zachmann]

» Boxtrees have a few benefits over traditional AABB trees:
  - Smaller memory footprint.
  - Slightly faster build times.
  - Faster query times due to the fact that the number of axes in the SAT can be further reduced.

# Broad Phase

- Find all pairs of objects whose axis-aligned bounding boxes overlap.
- We can do better than the $O(n^2)$ test-all-pairs approach by exploiting two principles:
  - **Spatial sorting**: only nearby objects can collide.
  - **Temporal coherence**: the configuration of objects does not change a lot per frame.

# Uniform Grid

- The world is a box.
- Subdivide the box into uniform rectangular cells (voxels).
- Cells need not keep coordinates of their position.
- Position $(x, y, z)$ goes into cell

$$(i, j, k) = \left(\lfloor x / e_x \rfloor, \lfloor y / e_y \rfloor, \lfloor z / e_z \rfloor\right)$$

where $e_x$, $e_y$, $e_z$ are the cell dimensions.

# Uniform Grid (cont'd)

» Each cell maintains a set of objects. Two alternative strategies:

1. Add an object to all cells that overlap the object's bounding box. Overlapping boxes must occupy the same cell.
2. Add an object to the cell that contains the center of the box. Neighboring cells need to be visited for overlapping boxes, but cells contain fewer objects.

# Uniform Grid (cont'd)

- Grids work well for large number of objects of roughly equal size and density (e.g. fluids).
- For these cases, grids have $O(1)$ memory and query overhead.

# Spatial Hashing

- Same as uniform grid except that the world is unbounded.
- Cell ID $(i, j, k)$ is hashed to a bucket in a hash table.
- Neighboring cells can still be visited. Simply compute hashes for $(i\pm1, j\pm1, k\pm1)$.

# Spatial Hashing (cont'd)

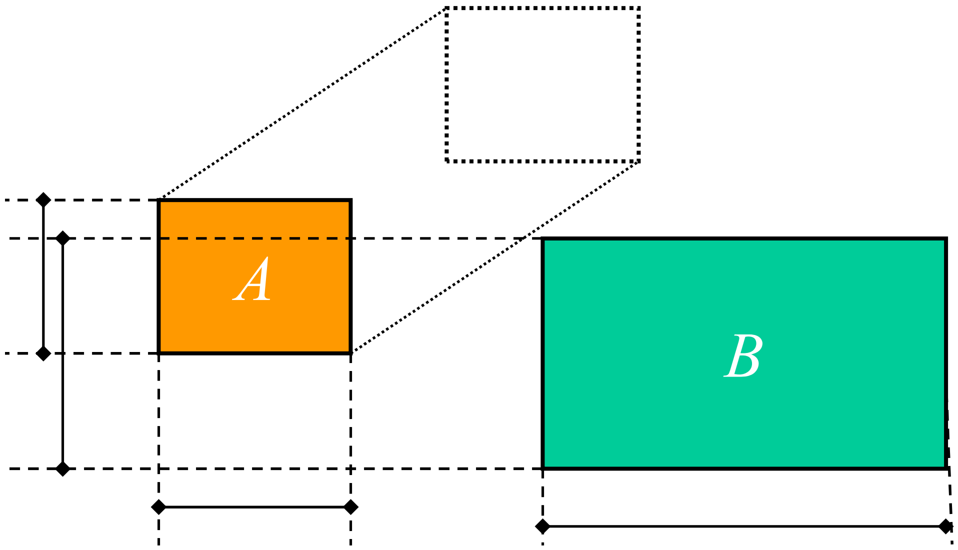
- As for grids, spatial hashing only works well for objects of roughly equal size and density.
- Multiple cells are hashed to the same bucket, so spatial coherence may not be that great.

# Sweep and Prune (1/3)

- For each world axis, maintain a sorted list of interval endpoints.
- Maintain also the set of overlapping box pairs.
- When a box moves, locally re-sort the lists by comparing and (if necessary) swapping the box endpoints with adjacent endpoints.

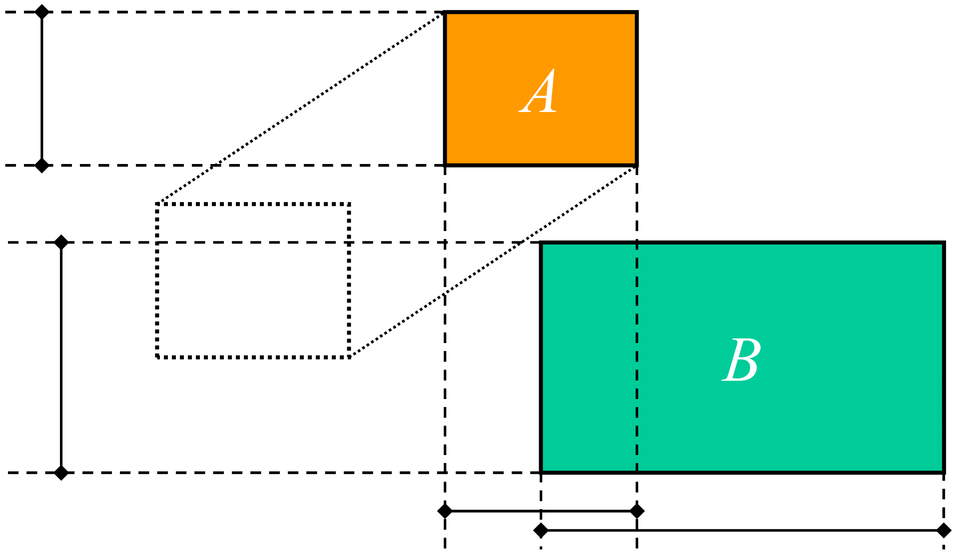
# Sweep and Prune (2/3)

Re-order endpoints of moving objects.



# Sweep and Prune (2/3)

Re-order endpoints of moving objects.



# Sweep and Prune (3/3)

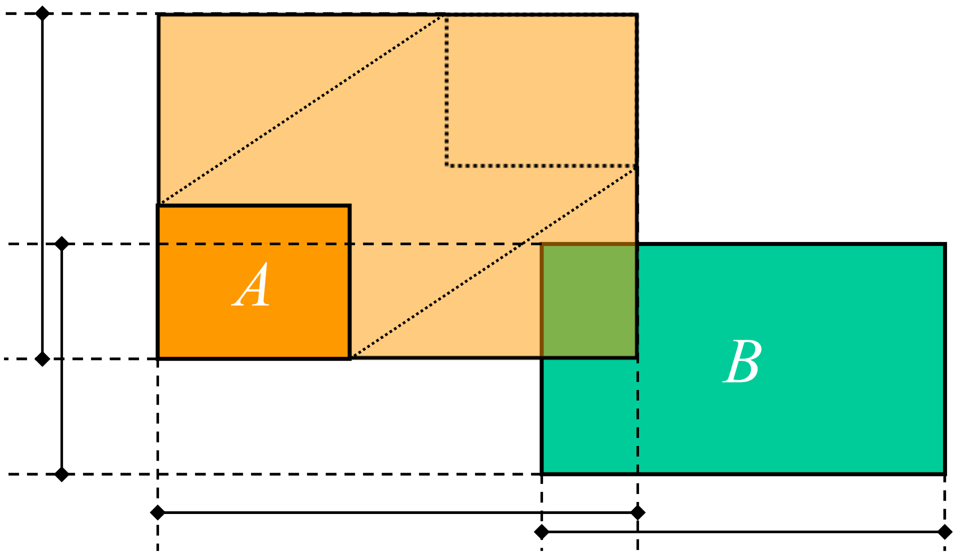
- When swapping "][" to "[]", the intervals start to overlap.
- When swapping "[]" to "][", the intervals cease to overlap.
- If the intervals on the other axes overlap, then the box pair starts or ceases to overlap.

# Adding Time: Encapsulation

- Enlarge AABBs of moving objects, such that they encapsulate the swept AABBs.
- Creates false positives: Encapsulating AABBs overlap where in space-time, the actual AABBs do not.

# Adding Time: Encapsulation (cont'd)

Encapsulation results in false positives.



# Adding Time: Queued Swaps

» Perform endpoint swaps in the proper order.

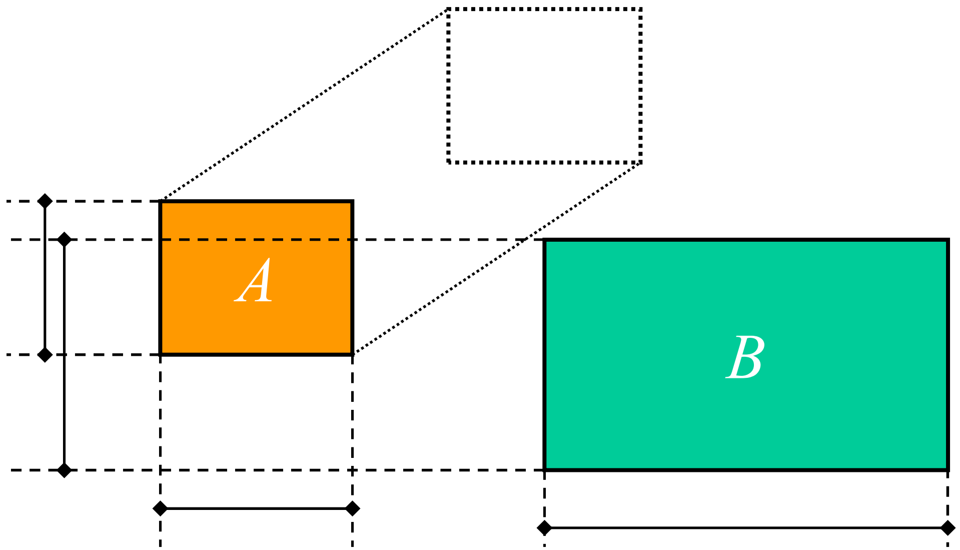
» Calculate swap times and prioritize swaps on earliest time.

» After each swap, re-evaluate swaps with new neighbors.

» Needs a priority queue that offers a *decrease-key* operation (as in Dijkstra's algorithm or A*).

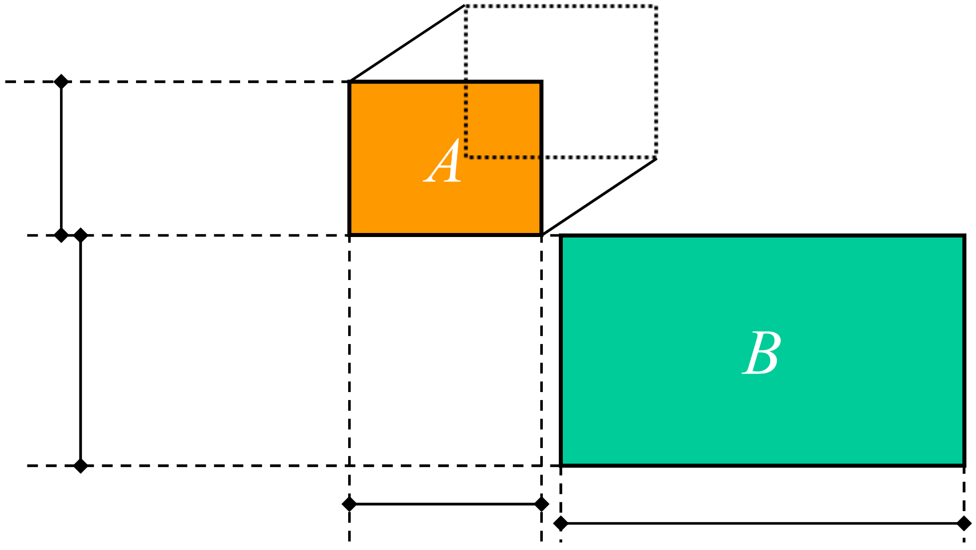
# Adding Time: Queued Swaps (cont'd)

Swap endpoints in the proper order.



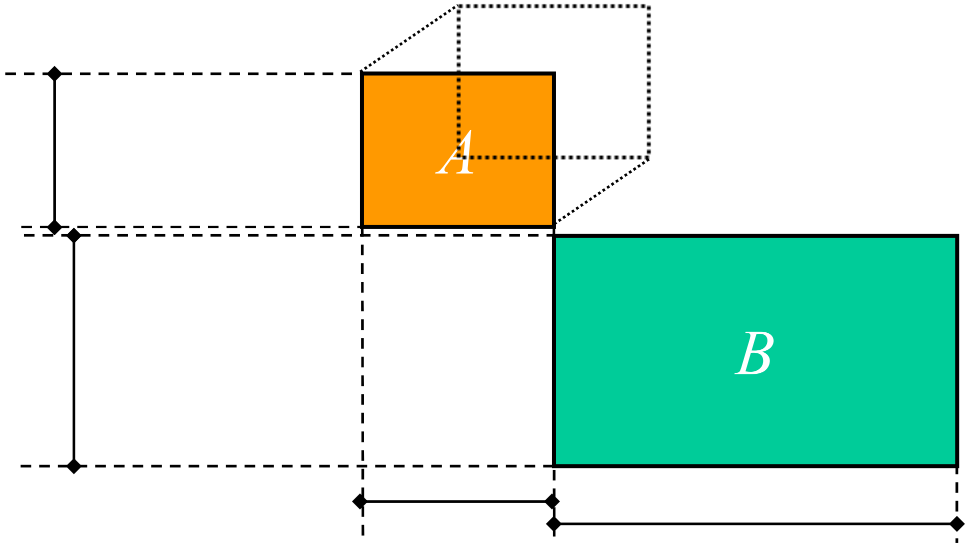
# Adding Time: Queued Swaps (cont'd)

Swap endpoints in the proper order.



# Adding Time: Queued Swaps (cont'd)

Swap endpoints in the proper order.



# Motion Coherence

- Space-time Sweep and Prune is often faster than the original version when many objects are moving in the same direction.
- Among a group of objects all having the same velocity vector not a single endpoint swap needs to be done.

# References


- Thomas H. Cormen et al. *Introduction to Algorithms, Second Edition*. MIT Press, 2001.
- E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):192-203, 1988.
- Gabriel Zachmann. *Minimal Hierarchical Collision Detection*. Proc. VRST, 2002.
- Gino van den Bergen. *Collision Detection in Interactive 3D Environments.* Morgan Kaufmann Publishers, 2004.

# Thank You!

- For papers and other information, check:

  http://www.dtecta.com