Geometric Primitives & Proximity Detection

Marq Singer
(marq@essentialmath.com)

Collisions

- Up to this point, objects just pass through each other
- Two parts to handling collisions
  - Collision detection – uses computational geometry techniques (useful in other ways, too)
  - Collision response – modifying physical simulation

Computational Geometry

- Algorithms for solving geometric problems
- Object intersections
- Object proximity
- Path planning

Distance Testing

- Useful for computing intersection between simple objects
- E.g. sphere intersection boils down to point-point distance test
- Just cover a few examples

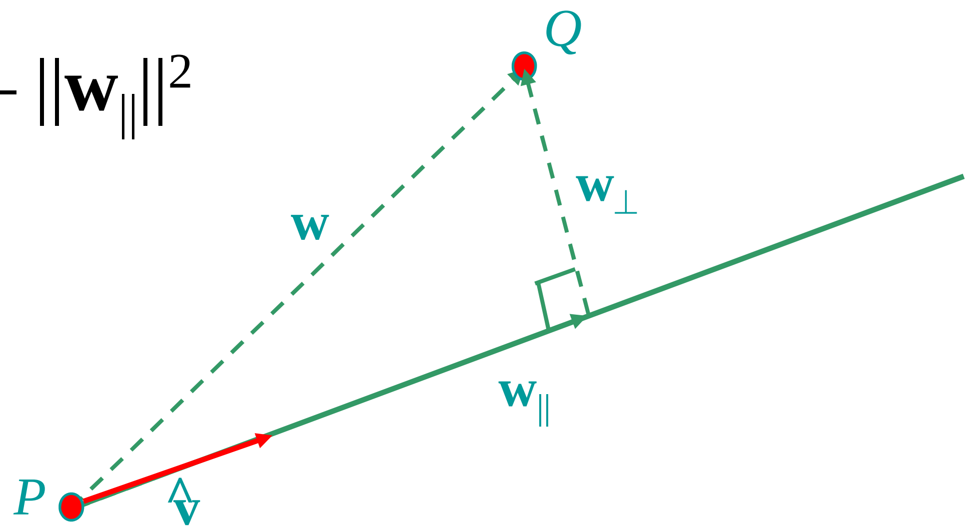
Point-Point Distance

» Compute length of vector between two points P_0 and P_1, or

$$\text{dist}(P_0, P_1) = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (z_0 - z_1)^2}$$

Line-Point Distance

- Line defined by point P and vector $\hat{\mathbf{v}}$
- Break vector $\mathbf{w} = Q - P$ into $\mathbf{w}_\perp$ and $\mathbf{w}_{||}$
- $\mathbf{w}_{||} = (\mathbf{w} \bullet \hat{\mathbf{v}})\, \hat{\mathbf{v}}$
- $||\mathbf{w}_\perp||^2 = ||\mathbf{w}||^2 - ||\mathbf{w}_{||}||^2$



Line-Point Distance

» Final formula:

$$\operatorname{dist}(P, \hat{\mathbf{v}}, Q) = \sqrt{\mathbf{w} \bullet (\mathbf{w} - \hat{\mathbf{v}})}$$

» If $\mathbf{v}$ isn't normalized:

$$\operatorname{dist}(P, \mathbf{v}, Q) = \sqrt{\mathbf{w} \bullet \left(\mathbf{w} - \frac{\mathbf{v}}{\mathbf{v} \bullet \mathbf{v}} \right)}$$

Line-Line Distance

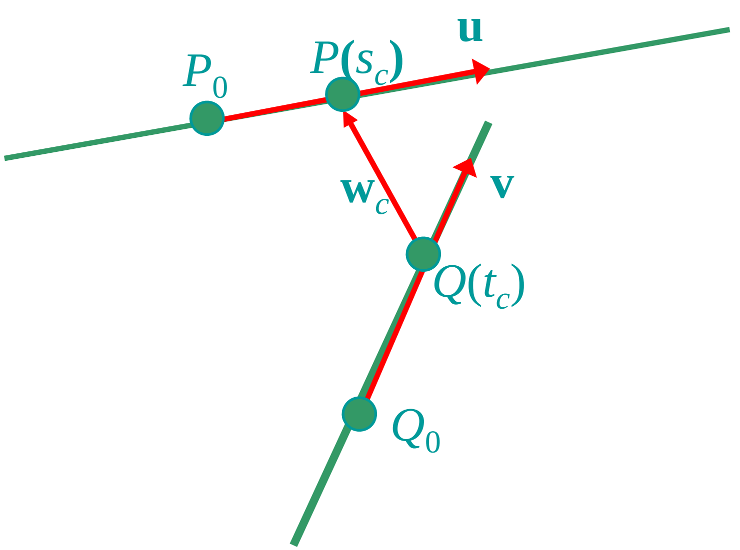
- From `http://www.geometryalgorithms.com`
- Vector $\mathbf{w}_c$ perpendicular to $\mathbf{u}$ and $\mathbf{v}$ or

$$\mathbf{w}_c = P(s_c) - Q(t_c)$$

$$\mathbf{u} \bullet \mathbf{w}_c = 0$$

$$\mathbf{v} \bullet \mathbf{w}_c = 0$$

- Two equations
- Two unknowns



Line-Line Distance

Final equations:

$$P(s_c) = \mathbf{P}_0 + (be - cd) \big/ (ac - b^2) \cdot \mathbf{u}$$

$$Q(t_c) = \mathbf{Q}_0 + (ae - bd) \big/ (ac - b^2) \cdot \mathbf{v}$$

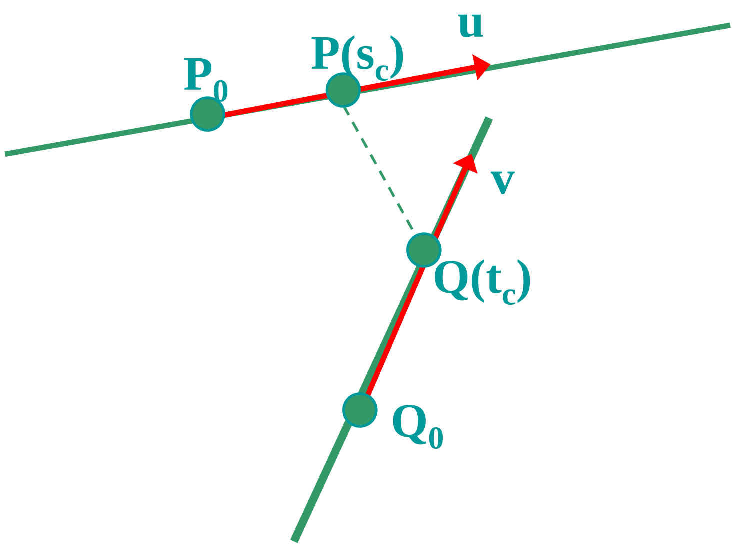
$$a = \mathbf{u} \bullet \mathbf{u}$$

$$b = \mathbf{u} \bullet \mathbf{v}$$

$$c = \mathbf{v} \bullet \mathbf{v}$$

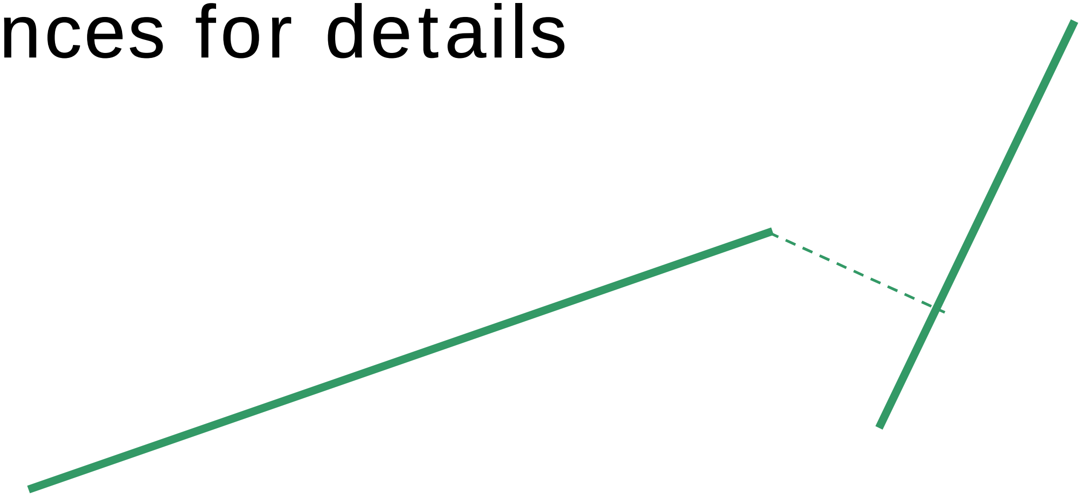
$$d = \mathbf{u} \bullet (\mathbf{P}_0 - \mathbf{Q}_0)$$

$$e = \mathbf{v} \bullet (\mathbf{P}_0 - \mathbf{Q}_0)$$



Segment-Segment Distance

- Determine closest point between *lines*
- If lies on both segments, done
- Otherwise clamp against nearest endpoint and recompute
- See references for details

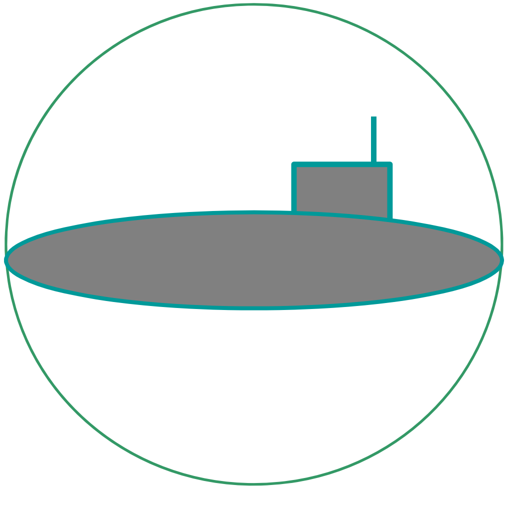


Bounding Objects

- Detecting intersections with complex objects expensive
- Provide simple object that surrounds them to cheaply cull out obvious cases
- Use for collision, rendering, picking
- Cover in increasing order of complexity

Bounding Sphere

- Tightest sphere that surrounds model
- For each point, compute distance from center, save max for radius

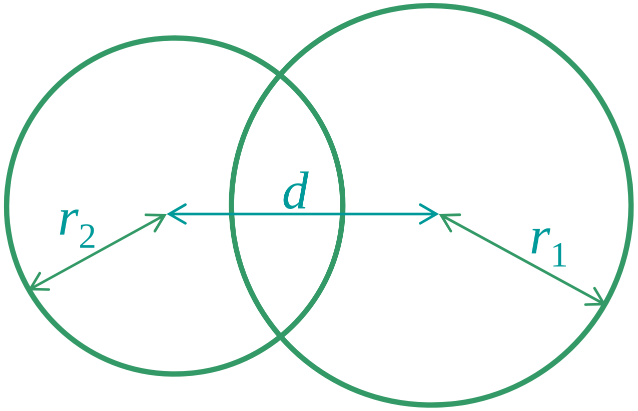


Bounding Sphere (Cont'd)

- What to use for center?
  - Local origin of model
  - Centroid (average of all points)
  - Center of bounding box
- Want a good fit to cull as much as possible
- Linear programming gives smallest fit

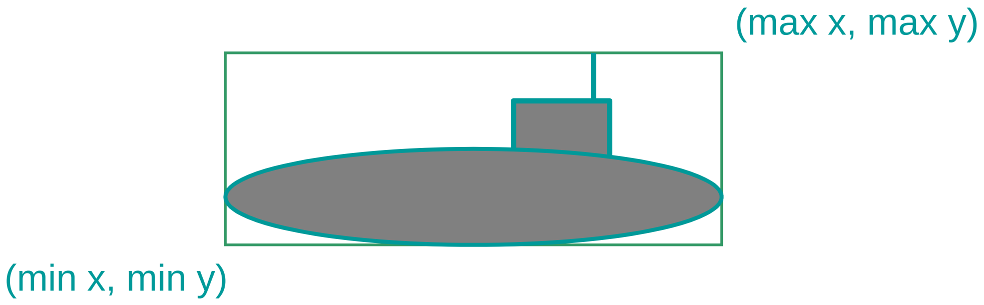
Sphere-Sphere Collision

- Compute distance d between centers
- If $d < r_1 + r_2$, colliding
- Note: d^2 is not necessarily $< r_1^2 + r_2^2$
  - want $d^2 < (r_1 + r_2)^2$



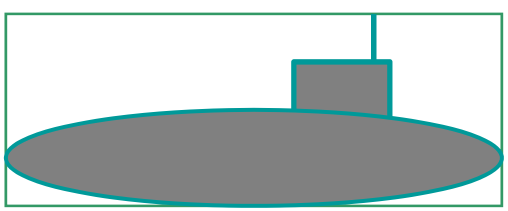
Bounding Box

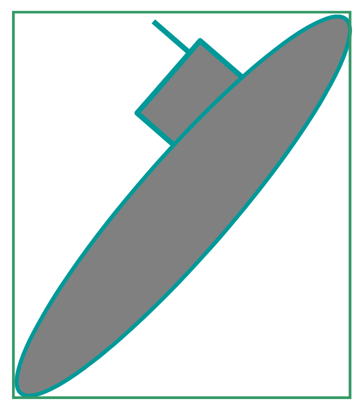
- Tightest box that surrounds model
- Compare points to min/max vertices
- If element less/greater, set element in min/max



Axis-Aligned Bounding Box

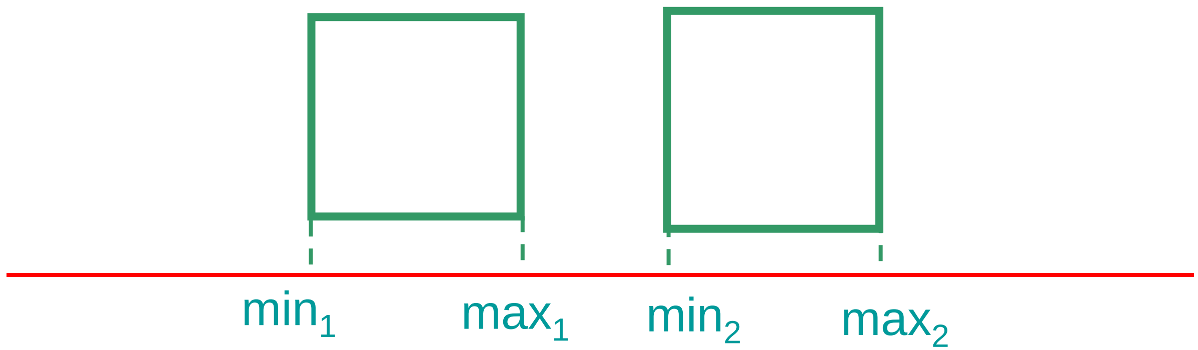
- Box edges aligned to world axes
- Recalc when object changes orientation
- Collision checks are cheaper though





Axis-Aligned Box-Box Collision

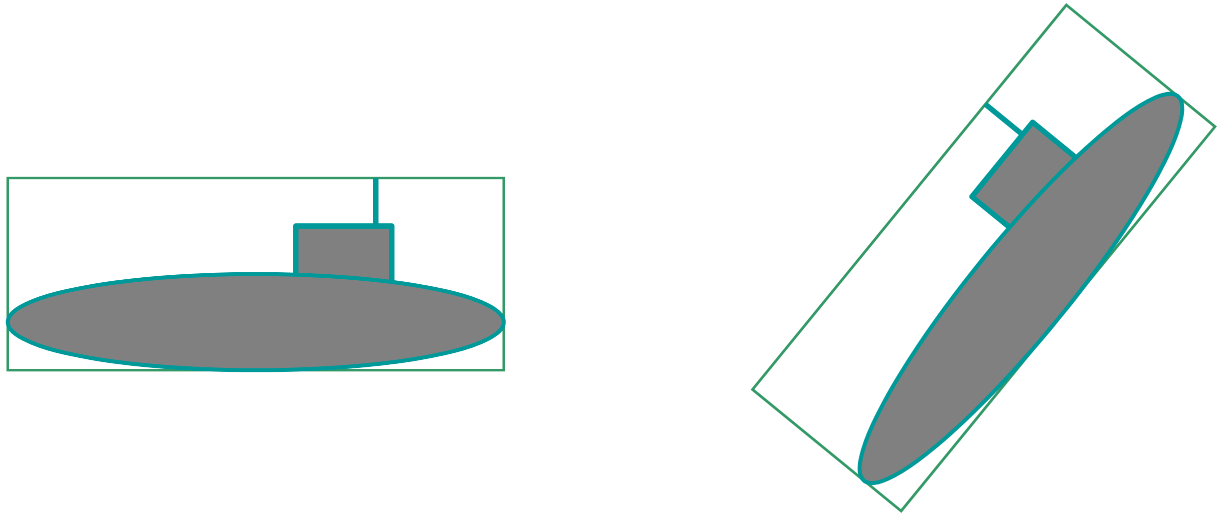
- Compare x values in min,max vertices
- If $min_2 > max_1$ or $min_1 > max_2$, no collision (separating plane)



- Otherwise check y and z directions

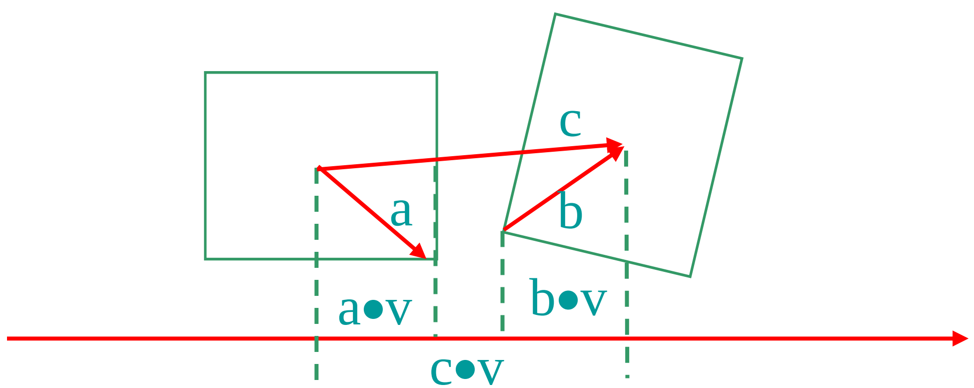
Object-Oriented Bounding Box

- Box edges aligned with **local** object coordinate system
- Much tighter, but collision calcs costly



OBB Collision

- Idea: determine if separating plane between boxes exists
- Project box extent onto plane vector, test against projection btwn centers



OBB Collision

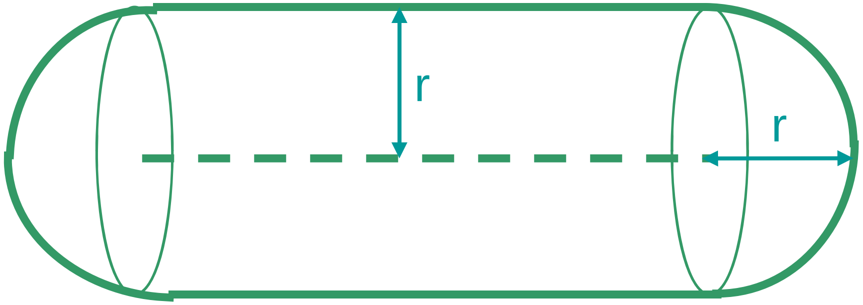
- To ensure maximum extents, take dot product using only absolute values

$$|a_x v_x| + |a_y v_y| + |a_z v_z|$$

- Check against axes for both boxes, plus cross products of all axes
- See Gottschalk for more details

Capsule

- Cylinder with hemispheres on ends
- One way to compute
  - Calc bounding box
  - Use long axis for length
  - Next largest width for radius

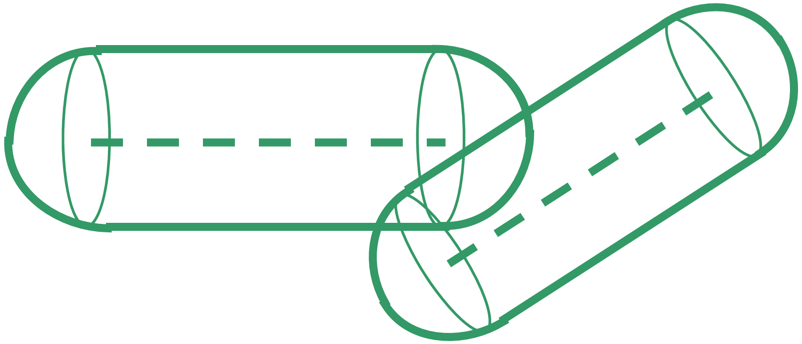


Capsule

- Compact
  - Only store radius, endpoints of line segment
- Oriented shape w/faster test than OBB
- Test path collision

Capsule-Capsule Collision

- Key: swept sphere axis is line segment with surrounding radius
- Compute distance between line segments
- If less than $r_1 + r_2$, collide



Caveat

- Math assumes infinite precision
- Floating point is *not to be trusted*
- Precision worse farther from 0
- Use epsilons
- Careful of operation order
- Re-use computed results
- More on floating point on website

Which To Use?

- As many as necessary
- Start with cheap tests, move up the list
  - Sphere
  - Swept Sphere
  - Box
- May not need them all

Recap

- Sphere -- cheap, not a good fit
- AABB -- still cheap, but must recalc and not a tight fit
- Swept Sphere -- oriented, cheaper than OBB but generally not as good a fit
- OBB -- somewhat costly, but a better fit

Collision Detection

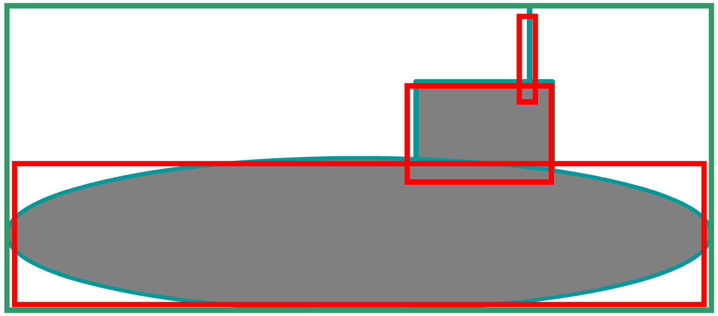
- Naïve: n^2 checks!
- Two part process
  - Broad phase
    - Cull out non-colliding pairs
  - Narrow phase
    - Determine penetration and contact points between pairs

Broad Phase

- Obvious steps
  - Only check each pair once
    - Flag object if collisions already checked
  - Only check moving objects
    - Check against other moving and static
  - Check rough bounding object first
    - AABB or sphere

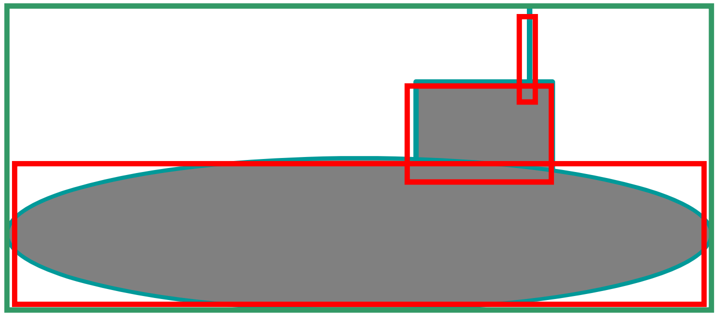
Hierarchical Systems

- Can break model into hierarchy and build bounds for each level of hierarchy
- Finer level of detection
- Test top level, cull out lots of lower levels



Hierarchical Systems

- Can use scene graph to maintain bounding information
- Propagate transforms down to children
- Propagate bound changes up to root

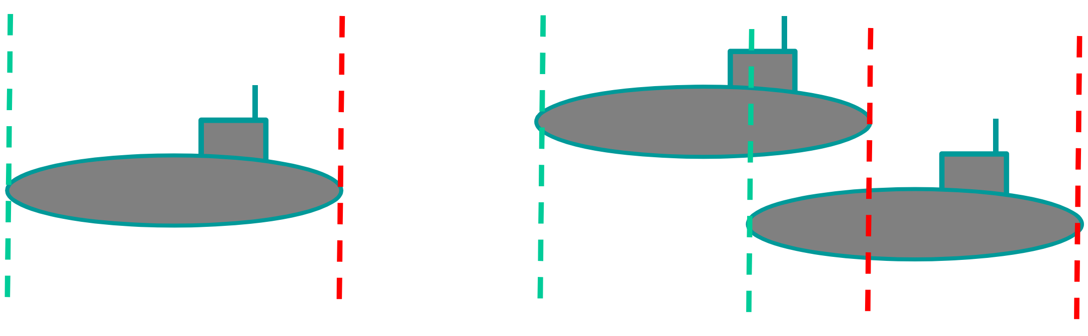


Spatial Subdivision

- Break world into separate areas
- Only check your area and neighbors
- Simplest: uniform
  - Slabs
  - Grid
  - Voxels

Sweep and Prune

- Store sorted x extents of objects
- Sweep from min x to max x
- As object min value comes up, make active, test against active objects
- Can extend to more dimensions



Spatial Subdivision

- Other methods:
  - Quadtrees, octrees
  - BSP trees, *k*d-trees
  - Room-portal
- Choice depends on your game type, rendering engine, memory available, etc.

Temporal Coherence

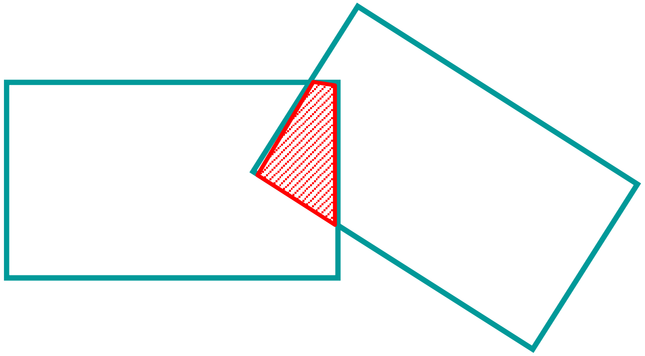
- Objects nearby generally stay nearby
- Check those first
- Can take memory to store information

Narrow Phase

- Have culled object pairs
- Need to find
  - Contact point
  - Normal
  - Penetration (if any)

Contact Region

- Two objects interpenetrate, have one (or more) regions



- A bit messy to deal with
- Many try to avoid interpenetration

Contact Features

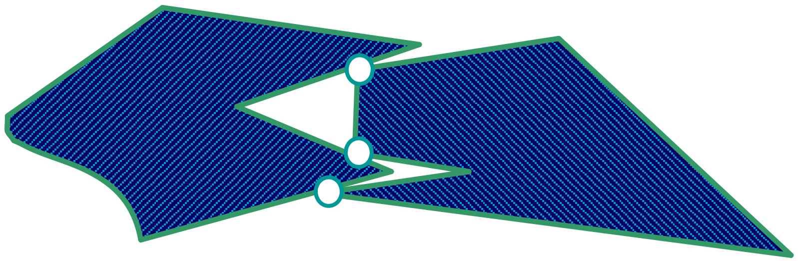
- Faceted objects collide at pair of contact features
- Only consider E-E and F-V pairs
- Infinite possibilities for normals for others
- Can generally convert to E-E and F-V
- Ex: V-V, pick neighboring face for one

Contact Features

- For E-E:
  - Point is intersection of edges
  - Normal is cross product of edge vectors
- For F-V:
  - Point is vertex location
  - Normal is face normal

Contact Points

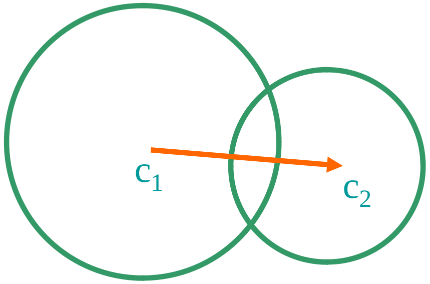
- Can have multiple contact points
  - Ex: two concave objects



- Store as part of collision detection
- Collate as part of collision resolution

Example: Spheres

- Difference between centers gives normal $\mathbf{n}$ (after you normalize)



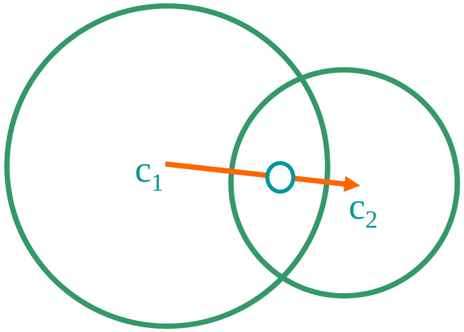
- Penetration distance p is

$$p = (r_1+r_2) - \|\mathbf{c}_2-\mathbf{c}_1\|$$

Example: Spheres

- Collision point: average of penetration distance along extended normal

$$\mathbf{v} = \tfrac{1}{2}(\mathbf{c}_1 + r_1\hat{n} + \mathbf{c}_2 - r_2\hat{n})$$



- If touching, where normal crosses sphere

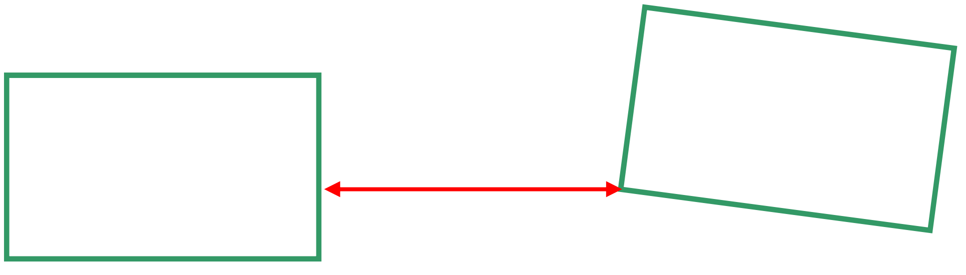
Lin-Canny

- For convex objects
- Easy to understand, hard to implement
- Closest features generally same from frame to frame
- Track between frames
- Modify by walking along object

Lin-Canny

- Frame 0

- Frame 1

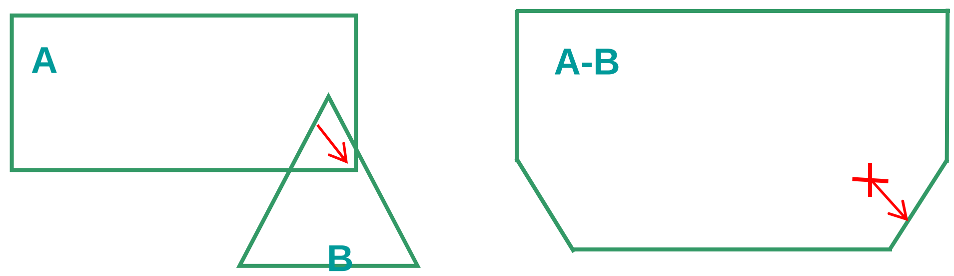


GJK

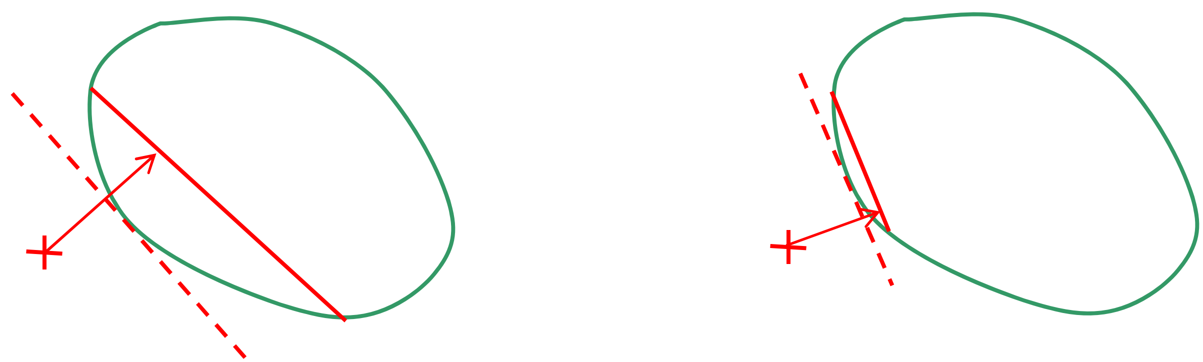
- For Convex Objects
- Hard to understand, easy to implement
- Finds point in Configuration Space Obstacle closest to origin. Corresponds to contact point
- Iteratively finds points by successive refinement of simplices

GJK

» CSO

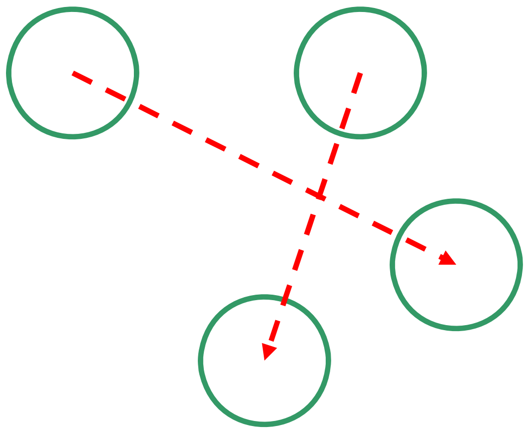


» Simplex Refinement



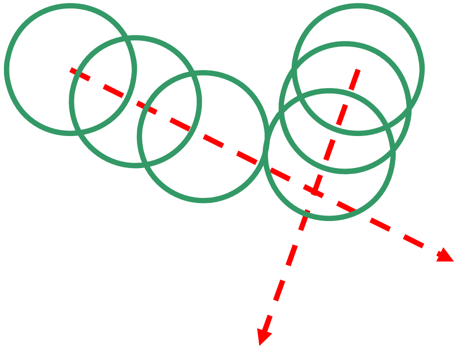
Missing Collision

» If time step is too large for object speed, two objects may pass right through each other without being detected (tunneling)



Missing Collision

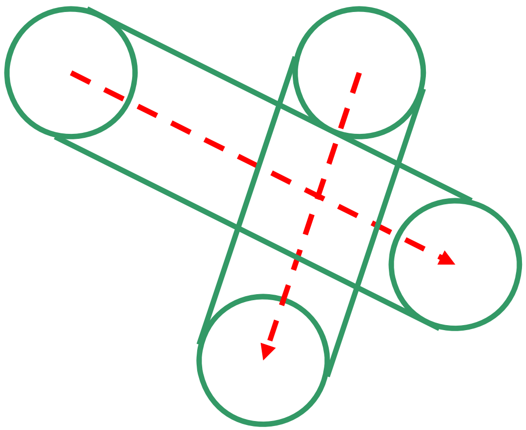
- One solution: slice time interval
- Simulate between slices



- Same problem, just reduced frequency

Missing Collision

- Another solution: use swept volumes



- If volumes collide, *may* collide in frame
- With more work can determine time-of-impact (TOI), if any

Recap

- Collision detection complex
- Combo of math and computing
- Break into two phases: broad and narrow
- Be careful of tunneling

References

- Preparata, Franco P. and Michael Ian Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.
- O'Rourke, Joseph, *Computational Geometry in C*, Cambridge University Press, New York, 1994.
- Eberly, David H., *3D Game Engine Design*, Morgan Kaufmann, San Francisco, 2001.
- Gottschalk, Stephan, Ming Lin and Dinesh Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection," *SIGGRAPH '96.*

References

- Van den Bergen, Gino, *Collision Detection in Interactive 3D Environments*, Morgan Kaufmann, San Francisco, 2003.
- Eberly, David H., *Game Physics*, Morgan Kaufmann, San Francisco, 2003.
- Ericson, Christer, *Real-Time Collision Detection*, Morgan Kaufmann, San Francisco, 2004.
- Van Verth, James M. and Lars M. Bishop, *Essential Mathematics For Games & Interactive Applications*, Morgan Kaufmann, San Francisco, 2004.