CS**5643**

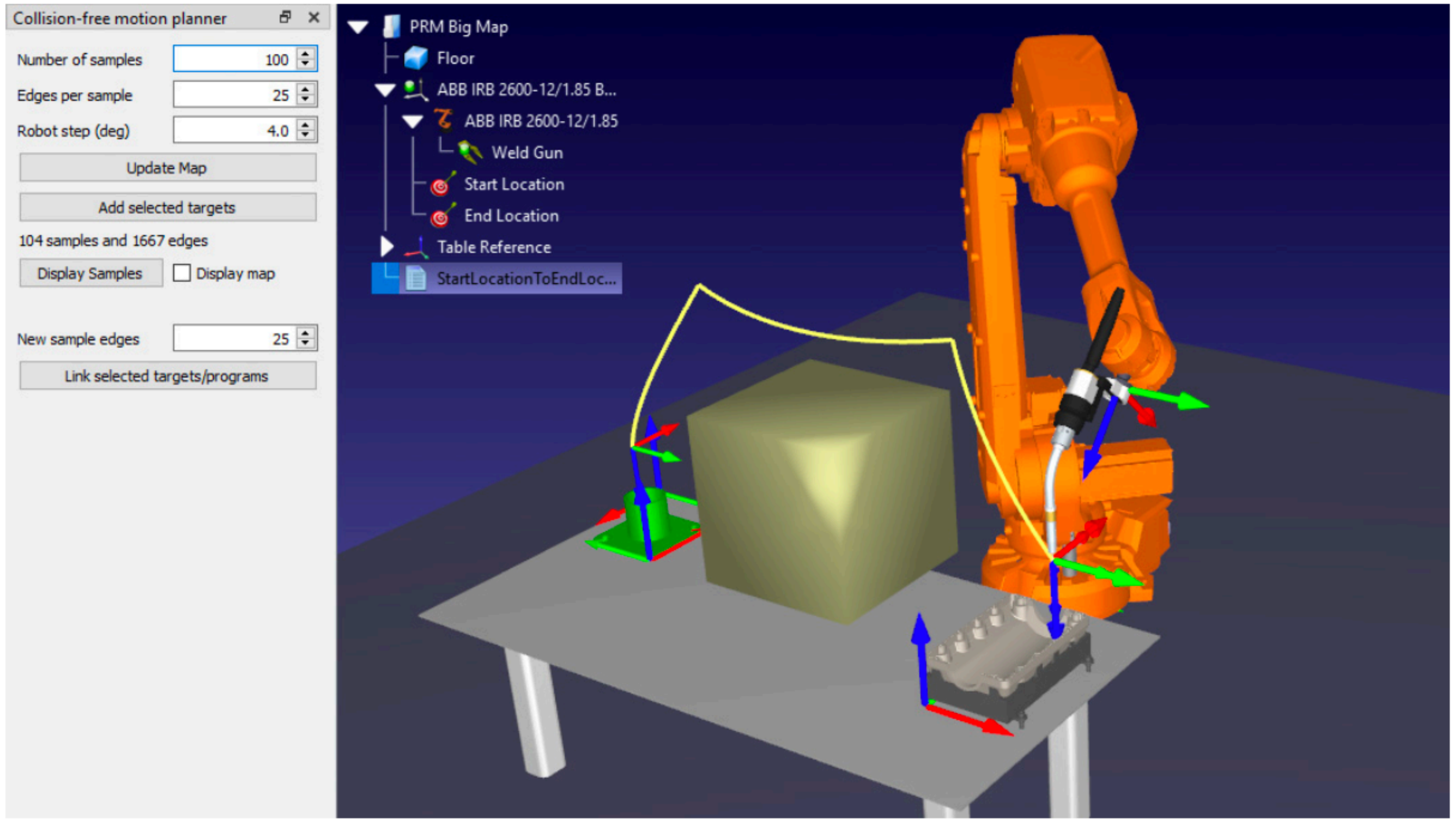
08 Collision detection

Steve Marschner
Cornell University
Spring 2023
(many images borrowed from Doug James's Stanford CS 248b slides)

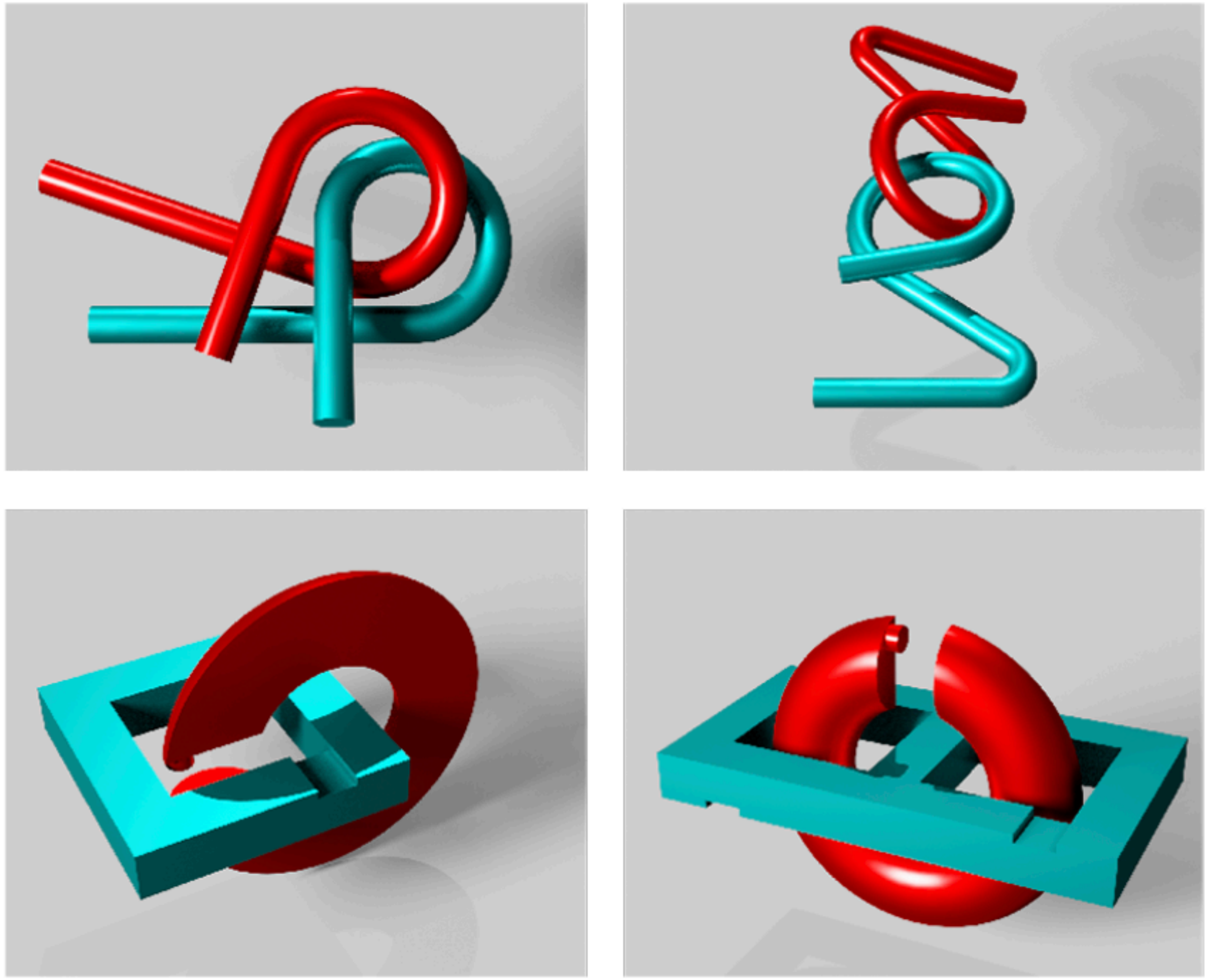
Collision detection

Goal: determine if two objects collide during a particular movement

- example: path planning for robotics or puzzles
- need to verify a particular motion path can execute with no collisions



https://robodk.com/blog/motion-planning-trend/



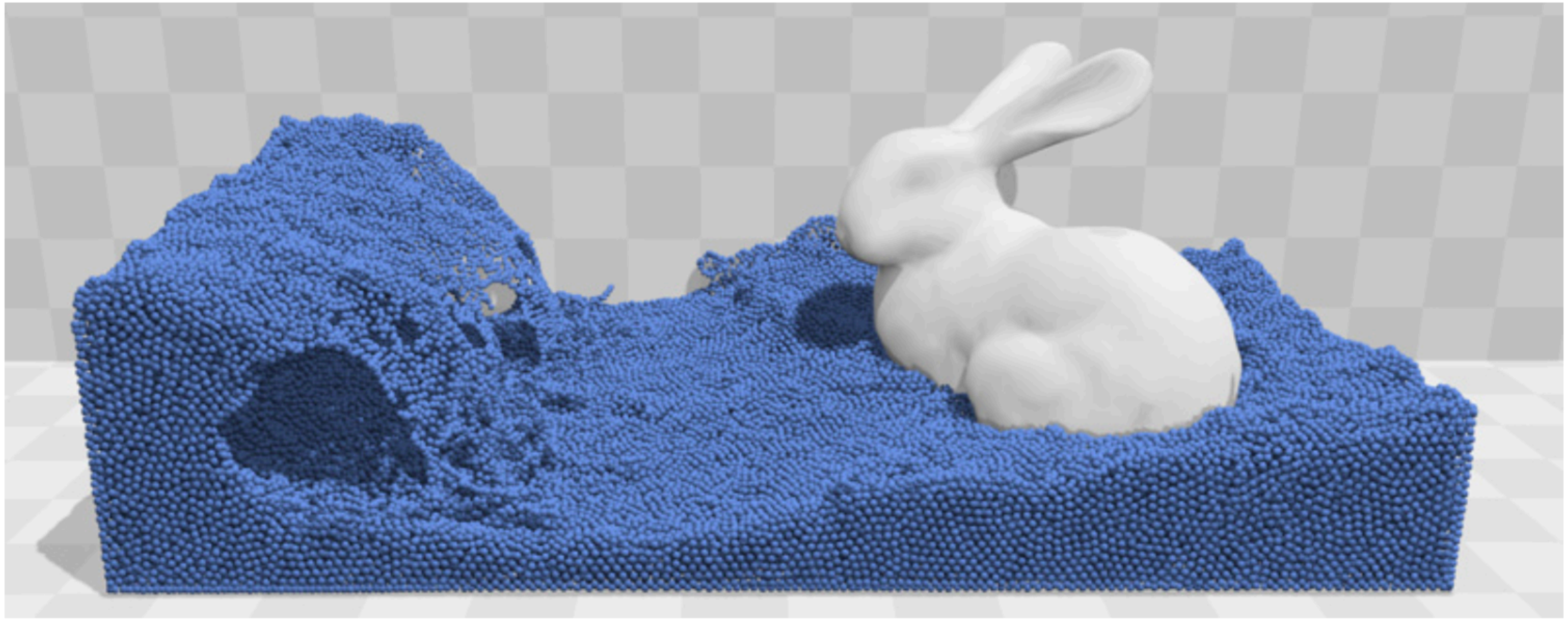
https://xinyazhang.gitlab.io/puzzletunneldiscovery/

slide borrowed from Doug James

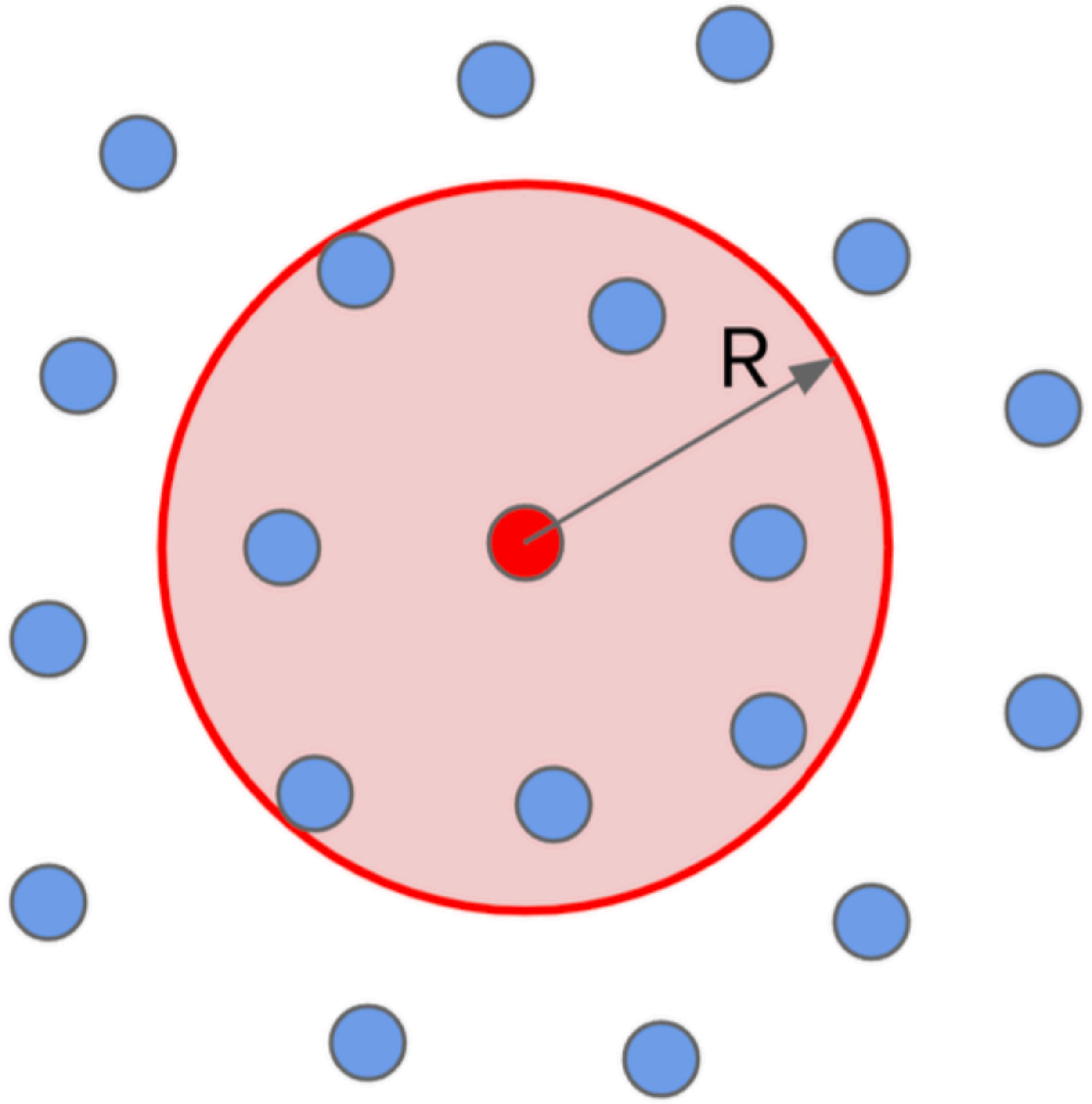
Proximity queries

Goal: detect when two objects approach within a threshold

- example: particle based fluid simulation
- each particle needs interacts with all particles closer than distance R



Position Based Fluids [Macklin and Mueller 2013]



slide borrowed from Doug James

Continuous vs. instantaneous collision detection

Version 1: "Are these two objects colliding right now?"

- instantaneous collision detection
- can miss collisions if you check once per frame

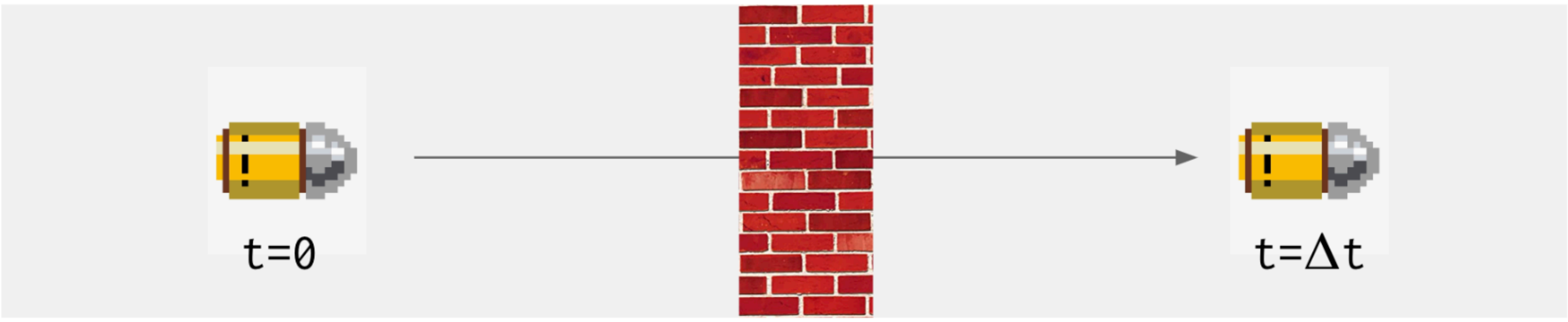
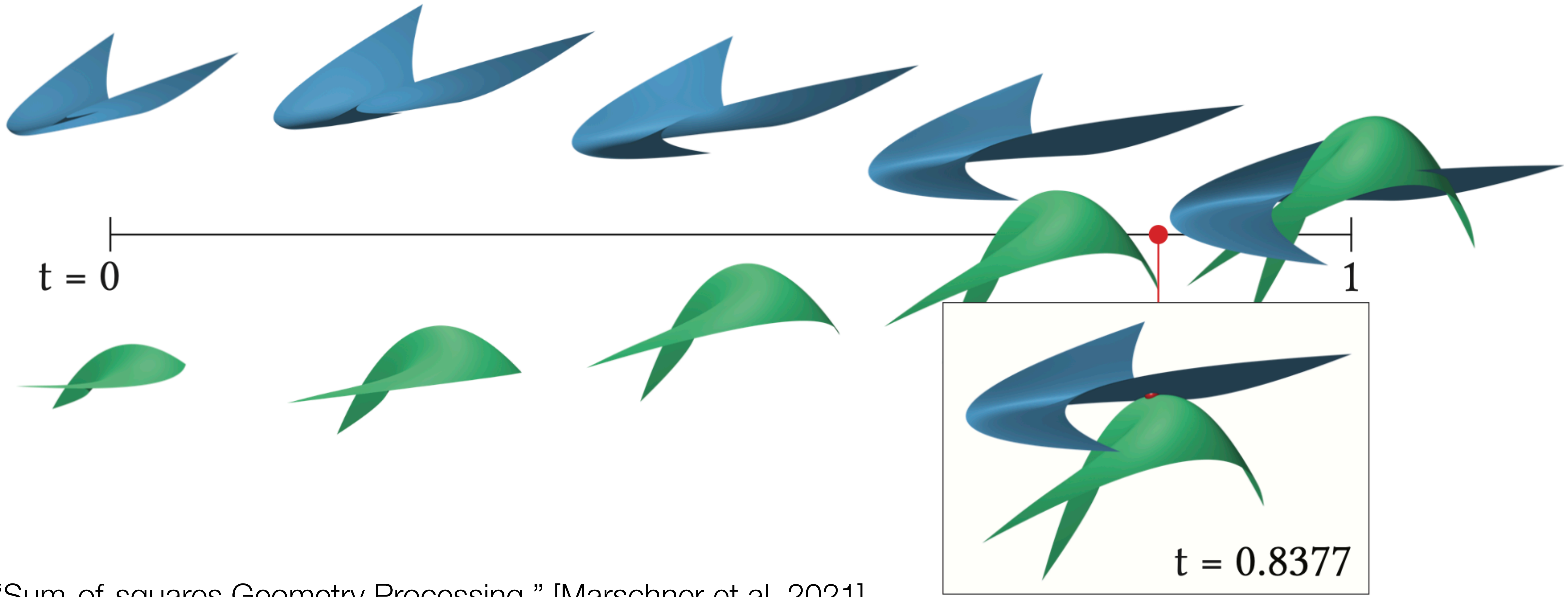


image borrowed from Doug James

Version 2: "If and when do these two moving objects collide?"

- continuous collision detection (CCD)
- can guarantee you don't miss collisions



"Sum-of-squares Geometry Processing." [Marschner et al. 2021]

Collision detection overview

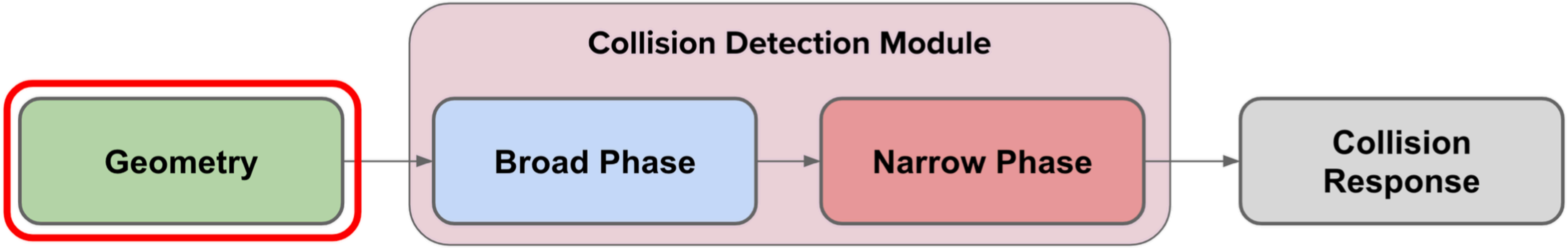
note: there's some disagreement between sources about where the boundary between "broad" and "narrow" goes…

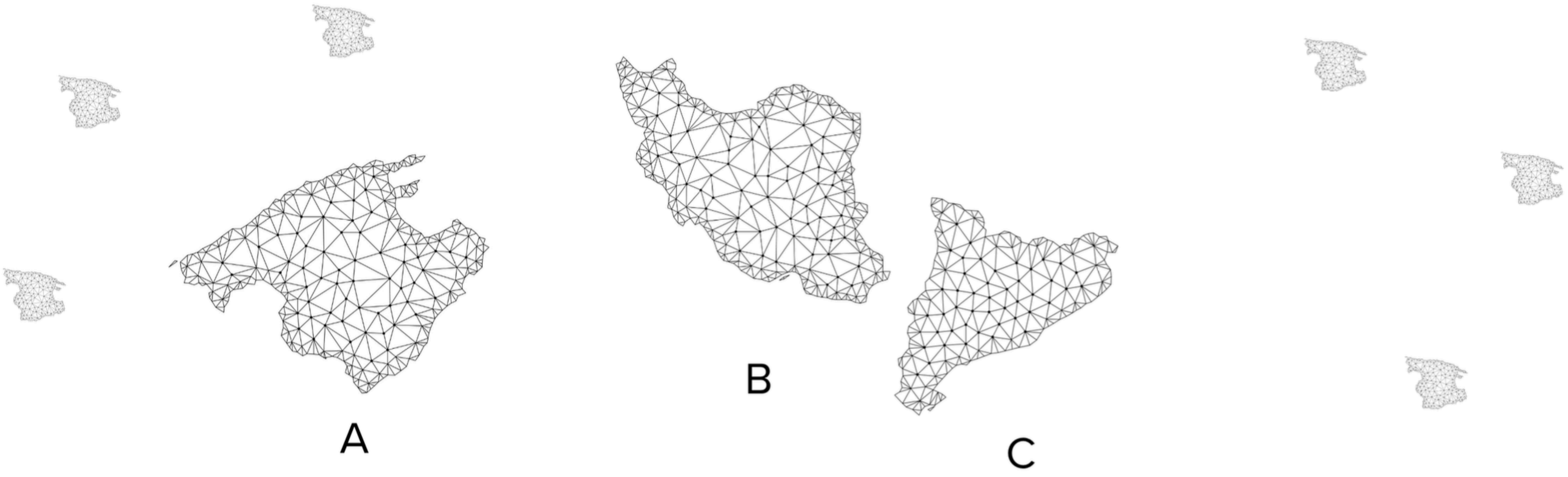
Narrow phase collision detection

- detects collisions between individual primitives
- produces definitive answers depending on the goals
    - yes/no for collision or proximity
    - time of collision
    - *k* nearest neighbors
- specific methods depend on primitive type (particles, lines, triangles, etc.)

Broad phase collision detection

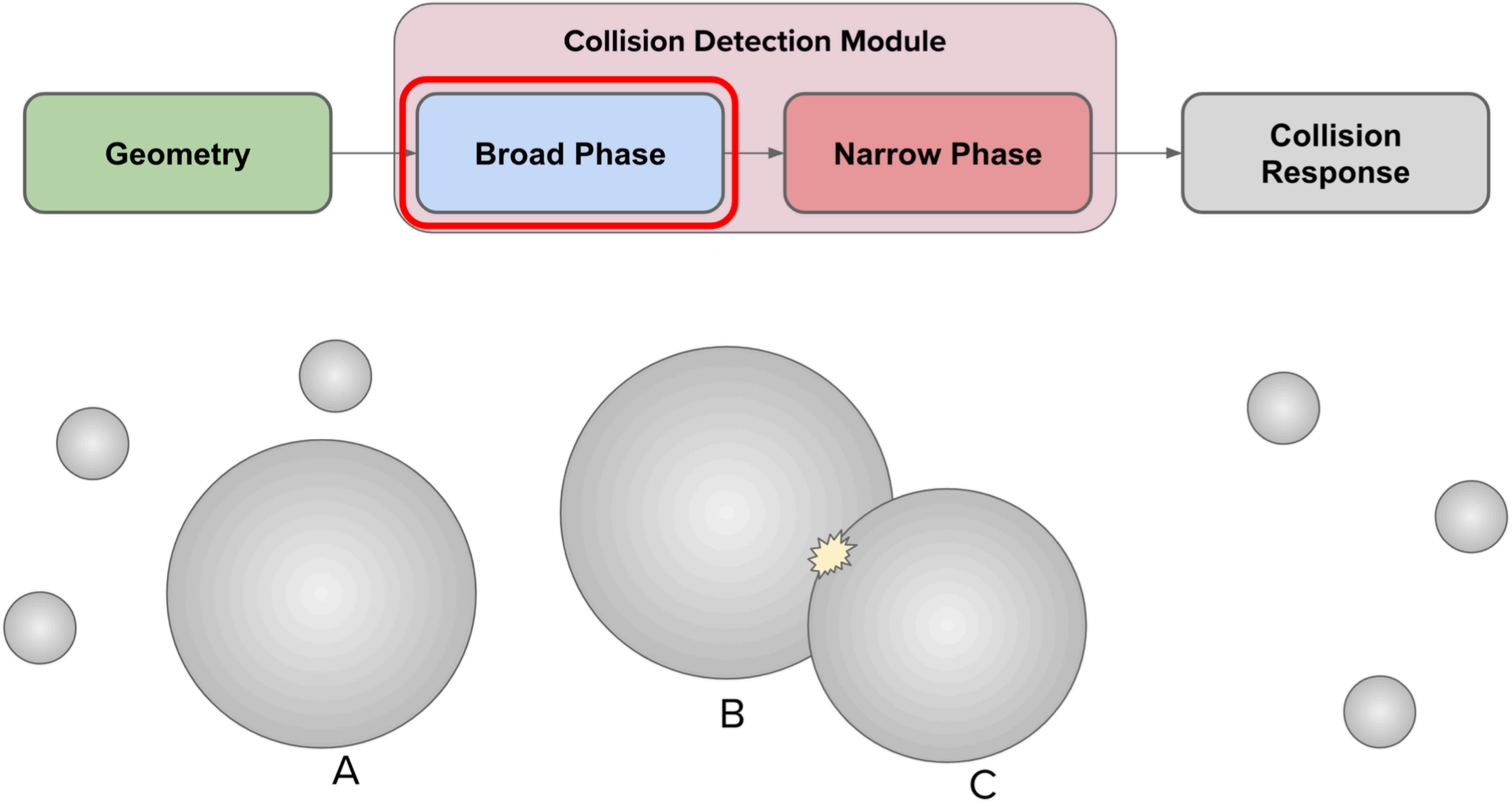
- conservatively eliminates potential collisions
- reduces the set of narrow-phase tests required
- uses various spatial data structures for efficiency
- specific methods depend on data structure (trees, grids, lists, etc.)





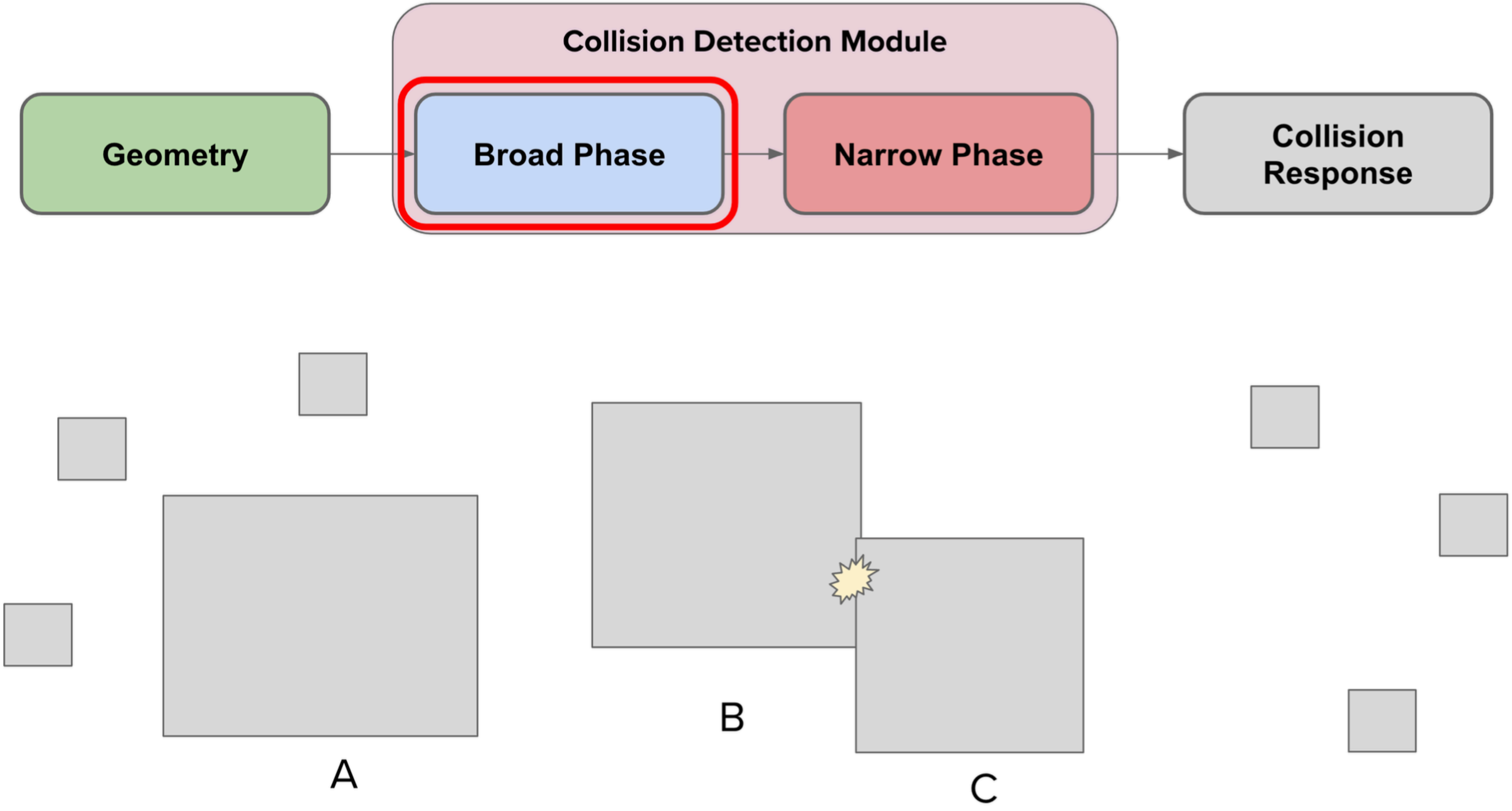
slide borrowed from Doug James

VectorStock.com images



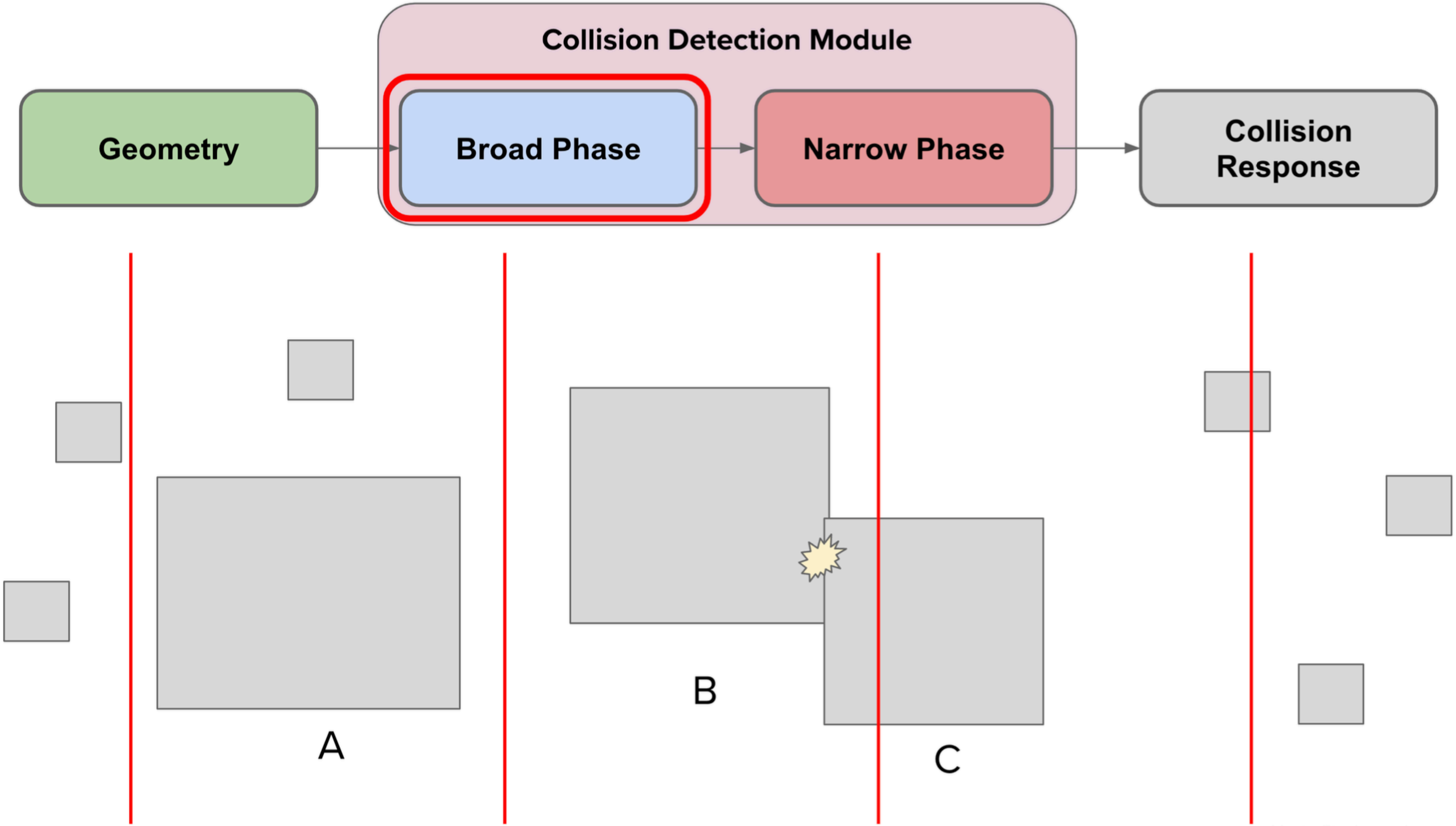
slide borrowed from Doug James

VectorStock.com images



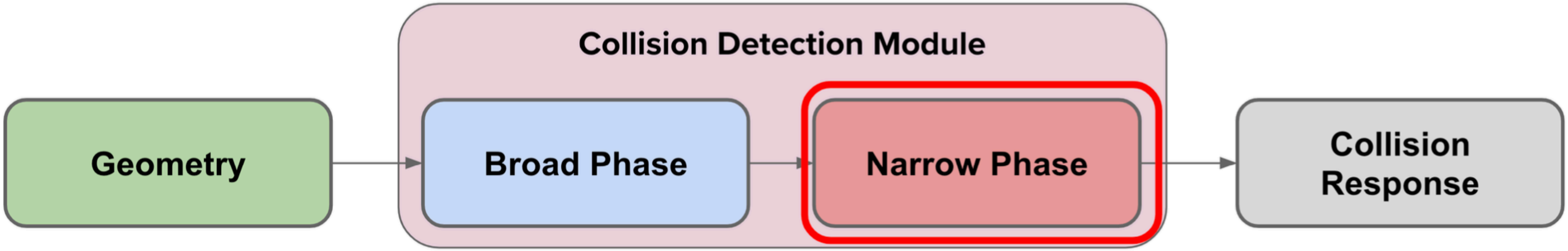
slide borrowed from Doug James

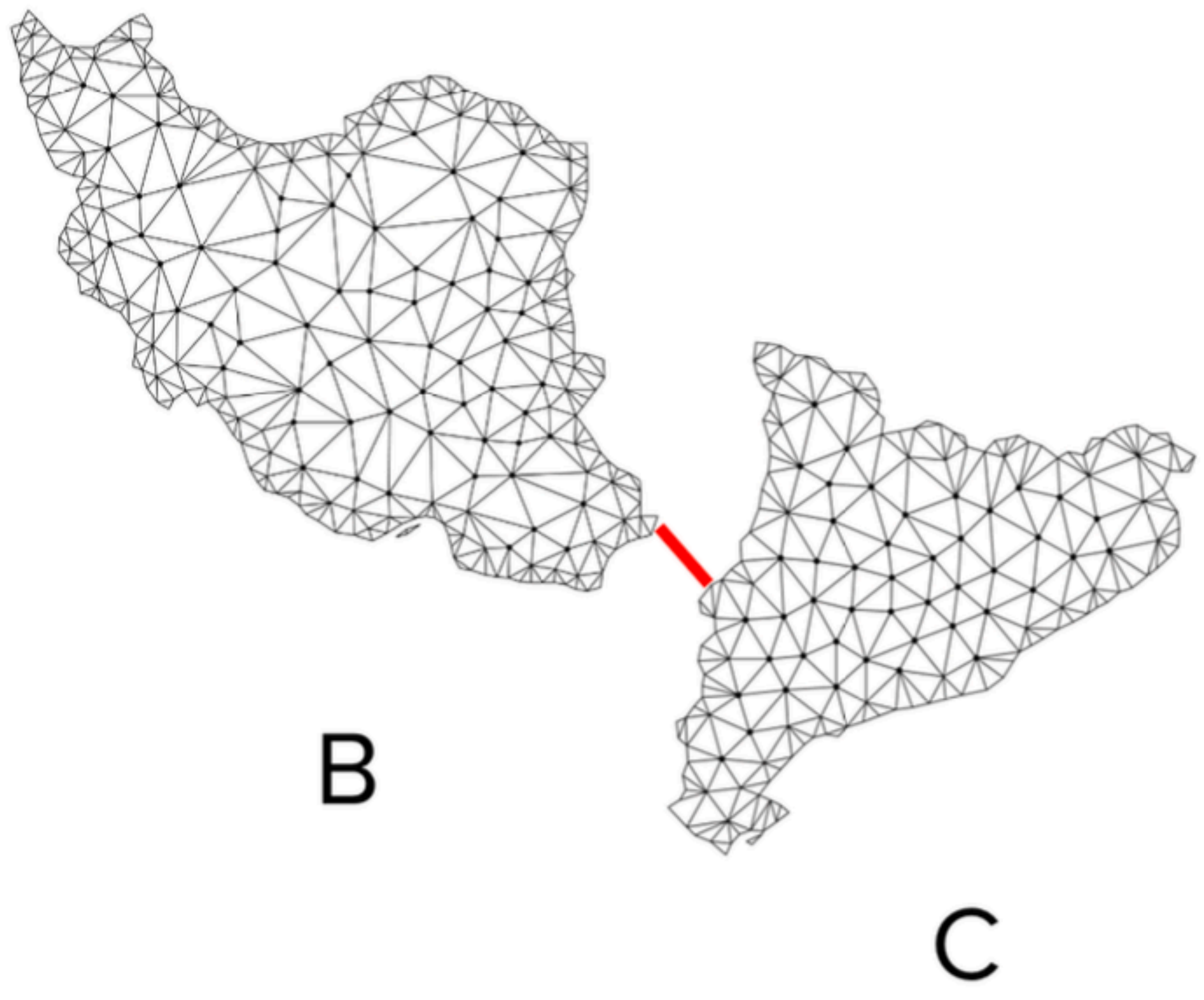
VectorStock.com images



slide borrowed from Doug James

VectorStock.com images





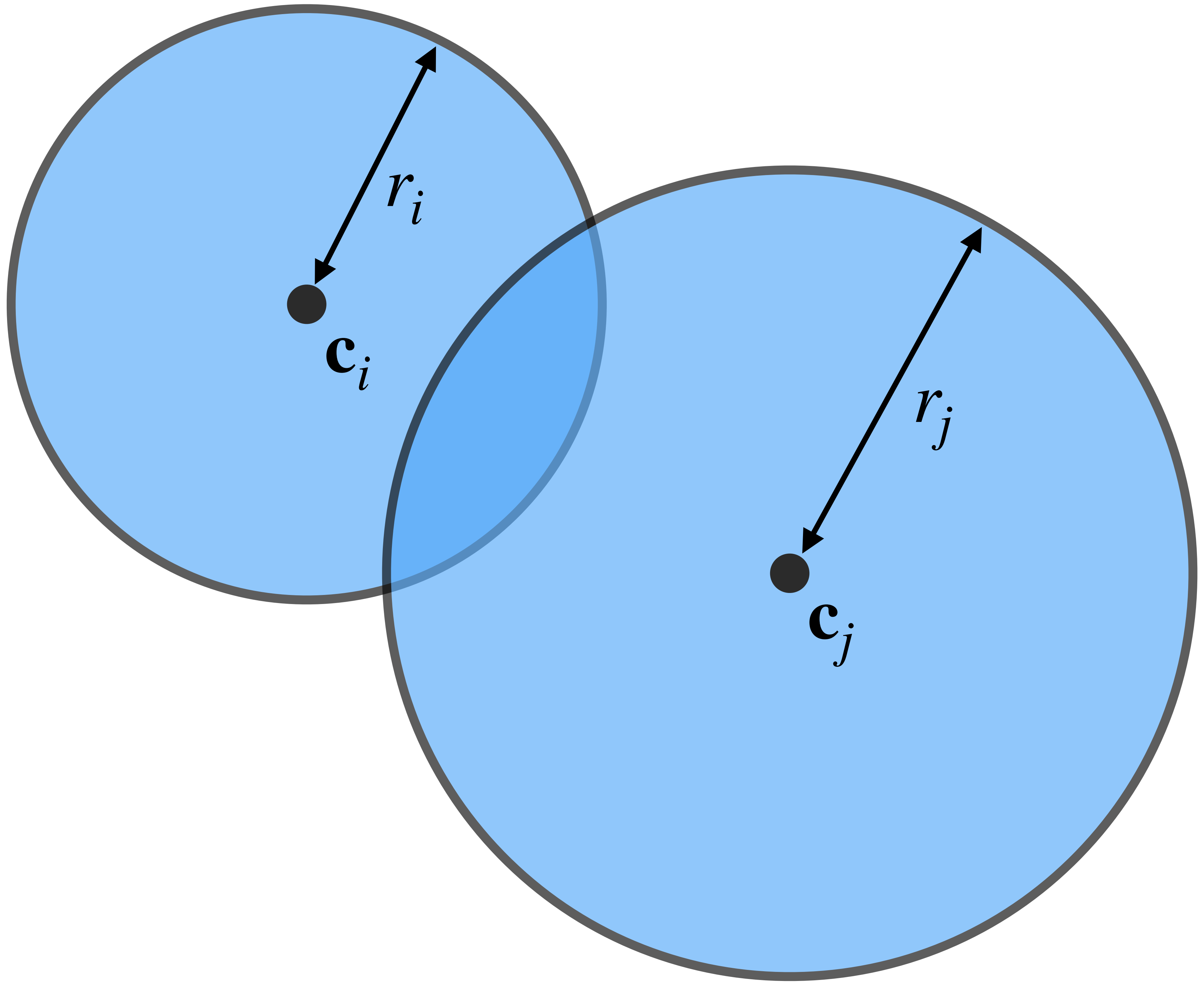
slide borrowed from Doug James

VectorStock.com images

Simple narrow-phase example

Colliding spheres

- example for now, will return to more interesting cases
- spheres or circles intersect if $\|\mathbf{c}_i - \mathbf{c}_j\|^2 < (r_i + r_j)^2$



Broad phase algorithm #0

Brute force loop over all pairs

- problem: $O(N^2)$

```
for i in range(N):
    for j in range(N):
        CheckCollision(i, j)
```

Avoiding N^2

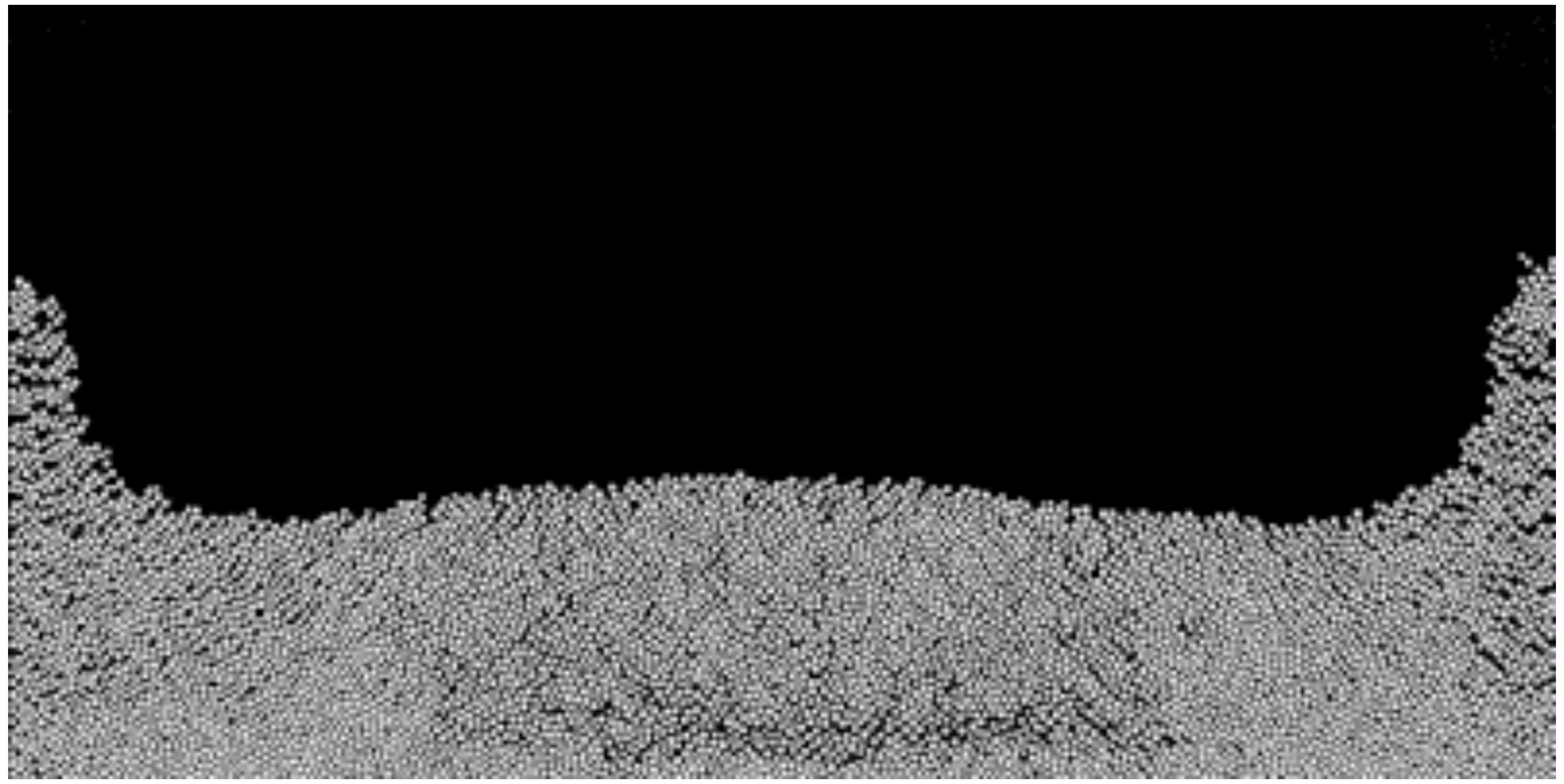
Sometimes there really are N^2 interactions

- have to deal with it
- reduce to $O(N)$ or $O(N \log N)$ by hierarchically approximating distant interactions
  - Fast Multipole Method (FMM)
  - Barnes-Hut approximation

In simulations usually only neighboring objects interact

- actual number of contacts is probably $O(N)$ for N objects
- goal is to efficiently search for “active contacts”

InsideHPC

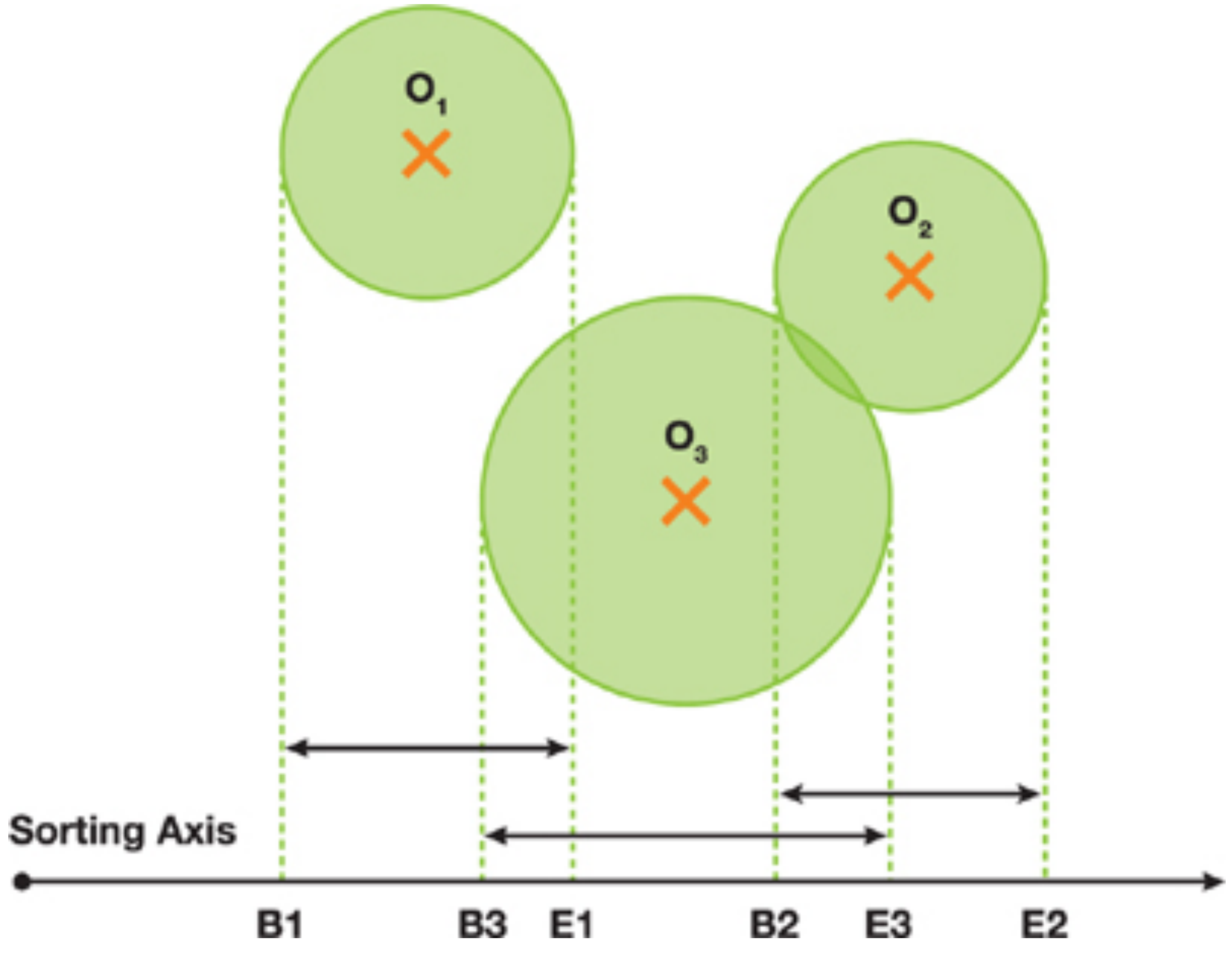


Taichi DEM demo

Collision detection by sort / sweep

Older idea: sort and sweep

- choose an axis (call it *x*) and project objects onto it
- put the min (begin) and max (end) *x* coordinates for each object into a big list
- sort the list
- traverse the list
  - begin object *i* -> add object *i* to active set
    check object *i* against active set
  - end object *i* -> remove object *i* from active set



Scott Le Grand, GPU Gems 3 Chapter 32

Problems

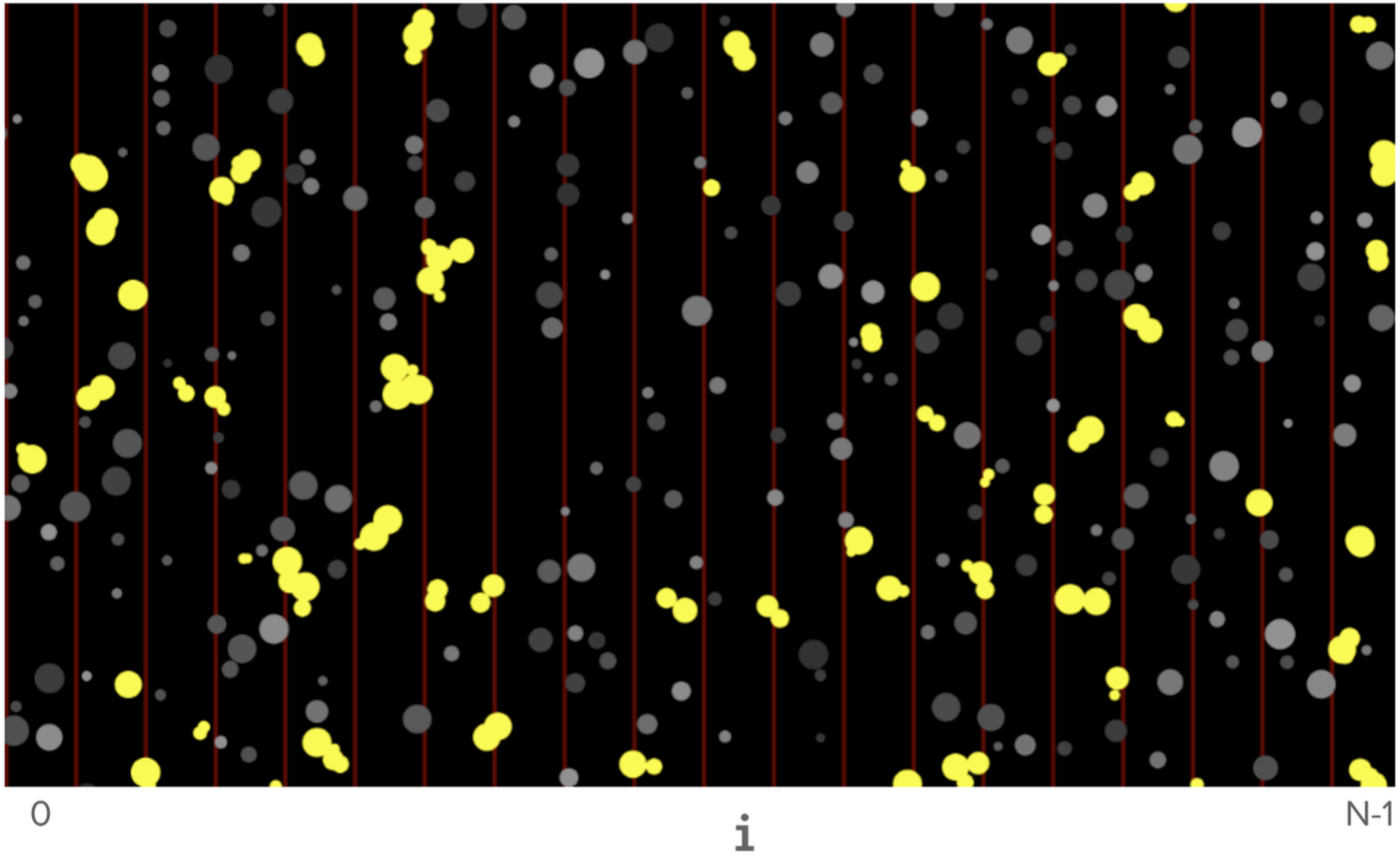
- sorting is not so parallel friendly
- what is the worst case for this? what is the time complexity for uniformly distributed objects?

Regular grid broad phase: 1D subdivision

Construction:

- Divide space into N bins of equal width, h
- Add each object to each bin that its bounding volume overlaps:
  - Use 1D overlap test

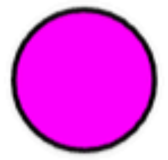
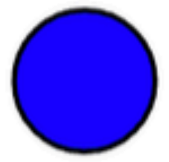
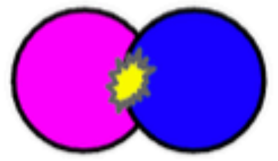
Cell Index, i: Given coordinate x, find containing cell `index(x)` using `Math.floor(x/h)` clamped to [0,N-1].

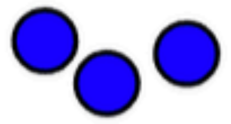


slide borrowed from Doug James

Regular grid broad phase: 1D subdivision

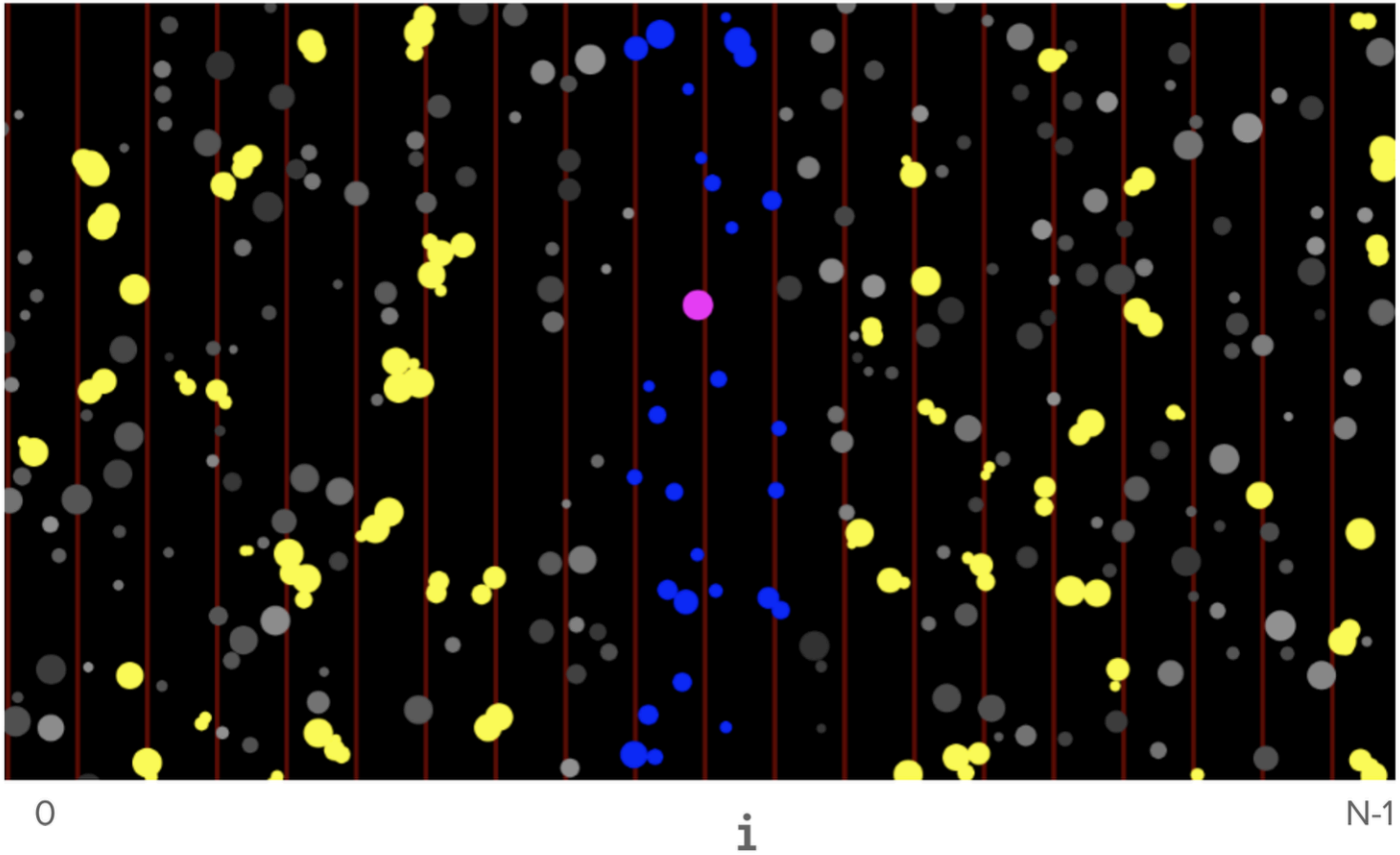
Overlap Testing:

- Given test bound 
- Find overlapping cells, and for each bound 
    - Do overlap test 
- Return overlapping results as a set.



Q: Can duplicate overlaps occur?

Weakness of 1D subdivision?



slide borrowed from Doug James

Regular grid broad phase: 2D subdivision

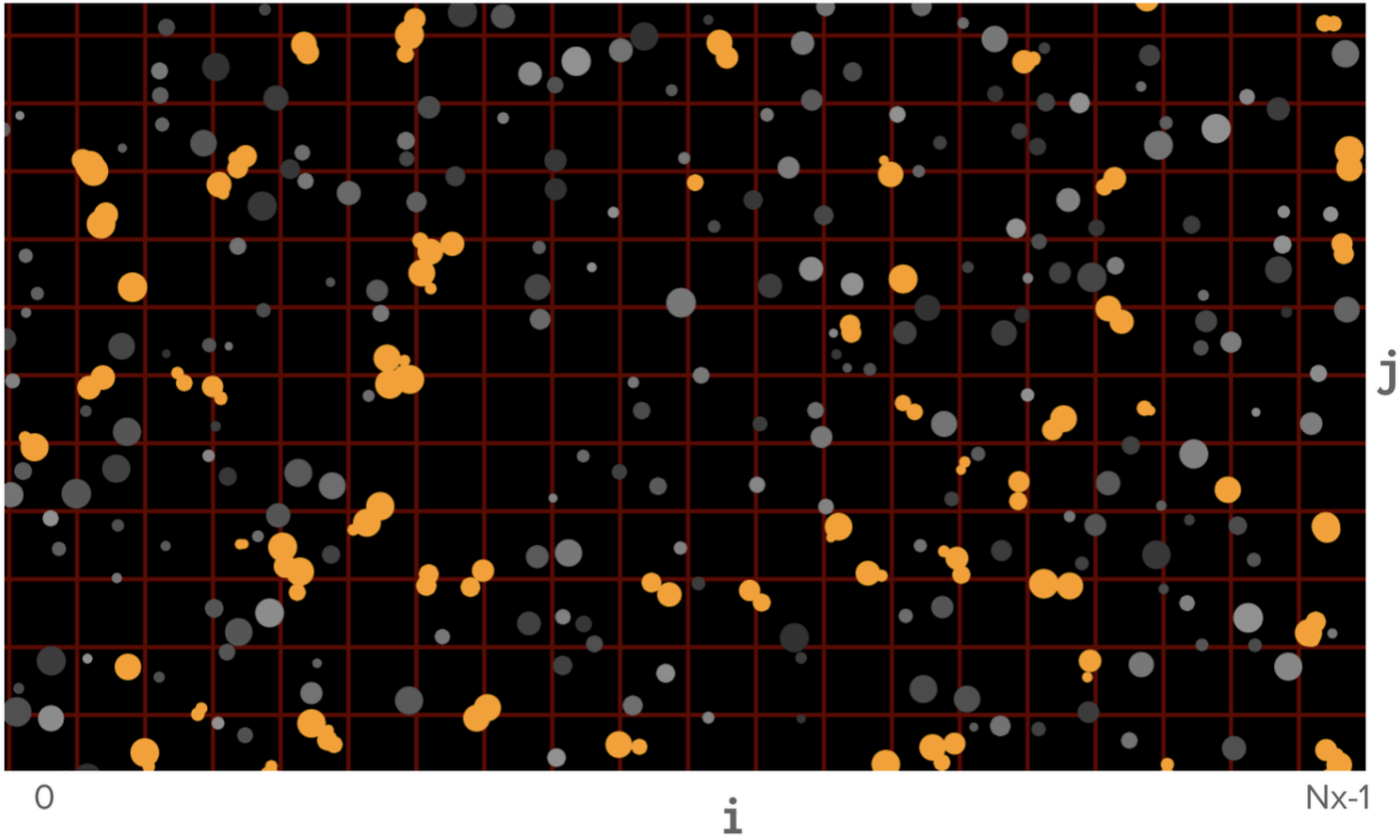
Construction:

- Divide space into Nx-by-Ny bins of constant width, h (or hx & hy)
- Add each object to each bin that its bounding volume overlaps:
  - Use 1D overlap tests

Cell Index (i,j): Given coords x & y,

`i = floor(x/h`$_x$`)` clamped to [0,Nx-1],

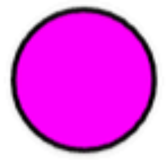
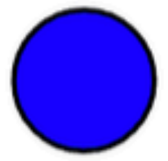
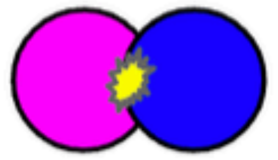
`J = floor(y/h`$_y$`)` clamped to [0,Ny-1].

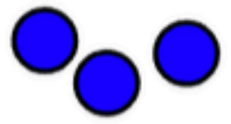


slide borrowed from Doug James

Regular grid broad phase: 2D subdivision

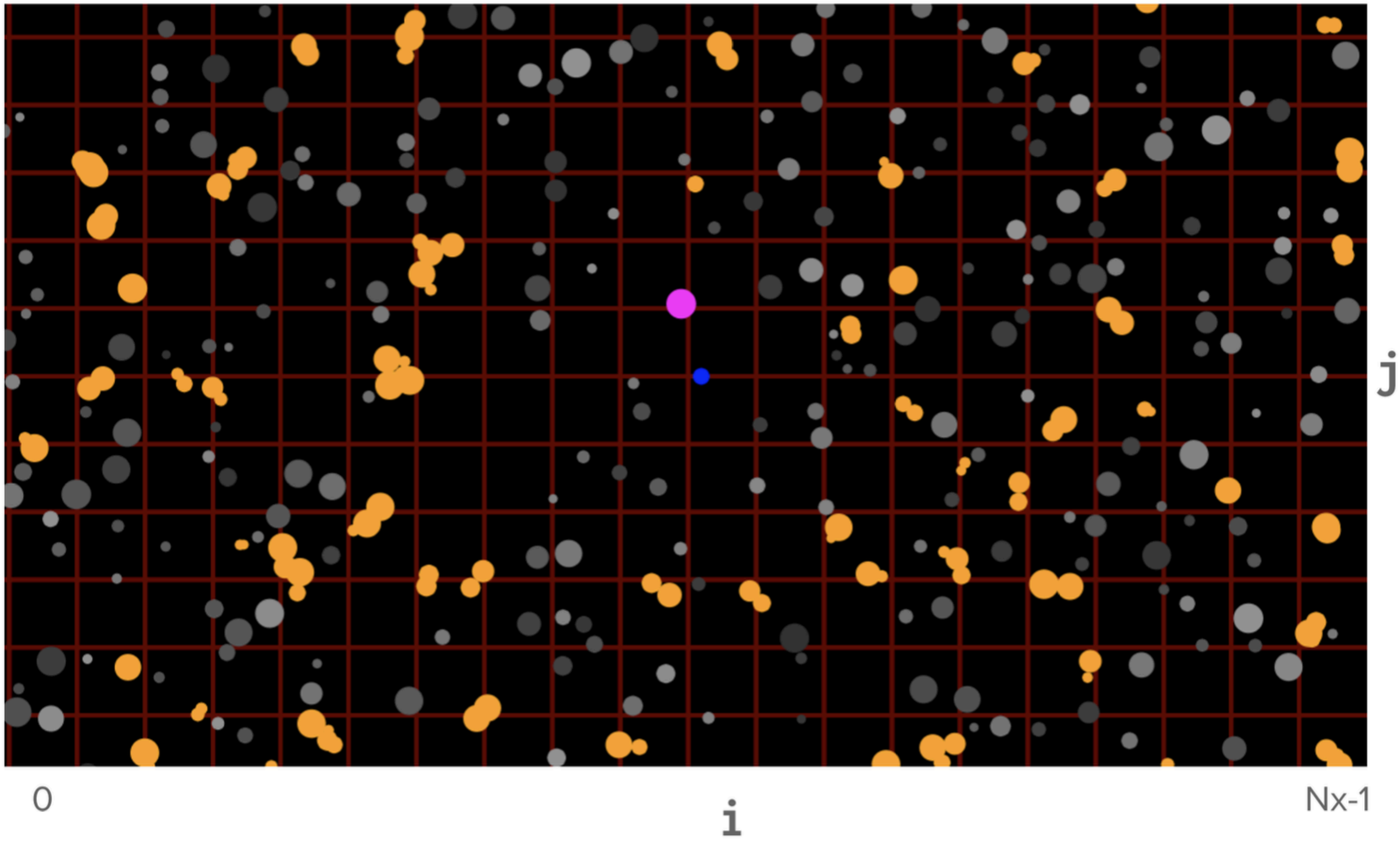
Overlap Testing:

- Given test bound 
- Find overlapping cells, and for each bound 
  - Do overlap test 
- Return overlapping results as a set.



Q: Can duplicate overlaps occur?

Weakness of 2D subdivision?



slide borrowed from Doug James

2D spatial subdivision

demo by Doug James

Advantages [demo]

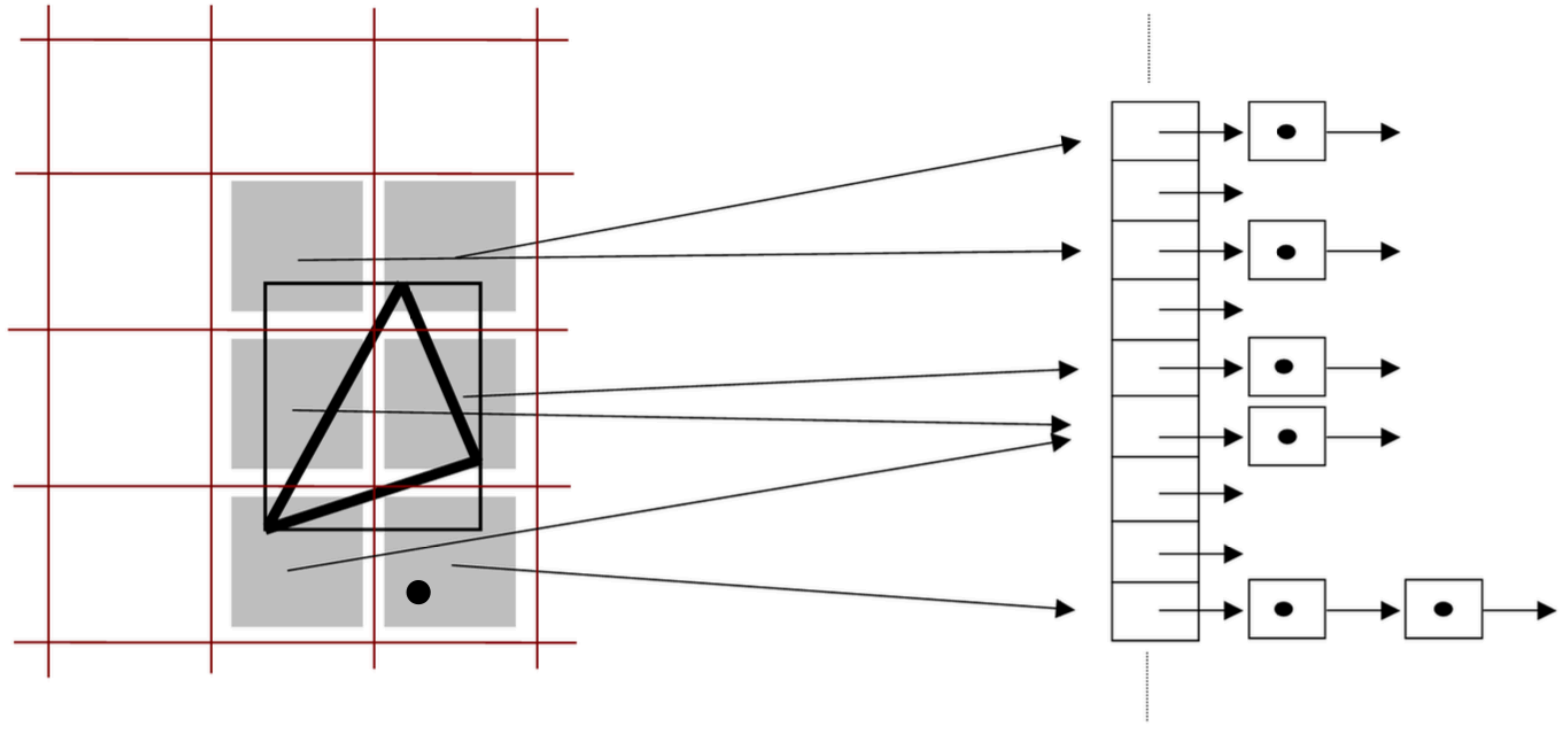
- often quite efficient; fairly simple to implement; reasonably parallel-friendly

Disadvantages

- large tables of possibly mostly empty particle lists; need to set grid dimensions up front
- what are the cases where it gets slow?

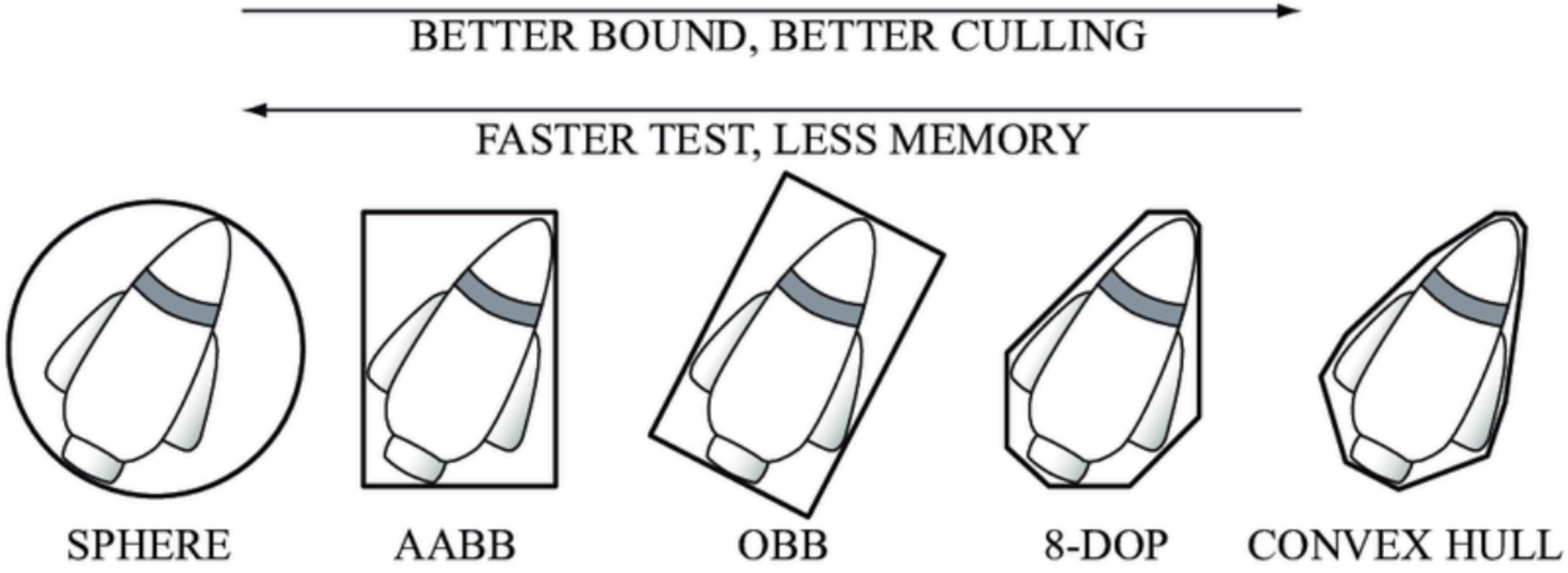
Variations

- spatial hashing: rather than grid[x,y],
  use table[hash(x,y)] for a suitable hash function
  - allows effectively unlimited grid; hash collisions
    just lead to some extra collision tests
- quadtrees, octrees: allow balancing cell occupancy
  when objects are nonuniformly distributed



Teschner et al 2003

Bounding volumes



[Ericsson 2004]

Simple idea to speed up collision checks

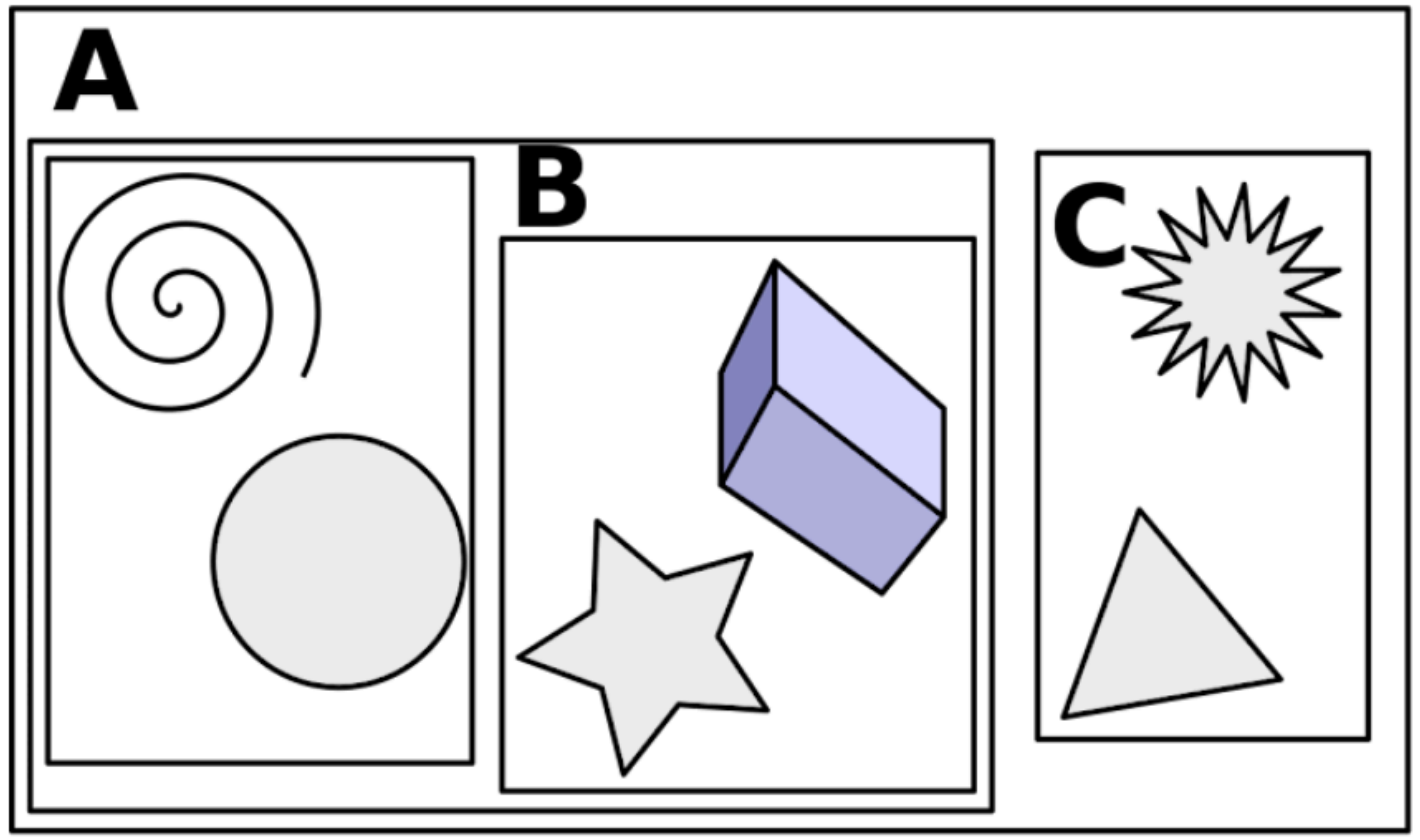
- first find a volume that contains (bounds) each object
- then when you want to test two objects for collision, first check whether their bounding volumes intersect
- no BV intersection → no collision, *guaranteed*!
- BV intersection → no guarantee, need to check for collisions
- for efficiency of intersection testing, BVs are always **convex**

Bounding volume hierarchies

Similar to those used for ray intersection

- can use any sort of bounding volume (BV)
- for any collision test, if the BV does not collide then the entire subtree can be skipped
- algorithms differ depending on query type
- to test against a simple obstacle for which a fast test is available, a simple traversal does the trick:

```
overlap(node, obstacle):
   if overlap_bv(node.bounds, obstacle):
      if node.is_leaf():
         return overlap_geom(node.geom, obstacle)
      else
         return overlap(node.left, obstacle) or
            overlap(node.right, obstacle)
   return false
```



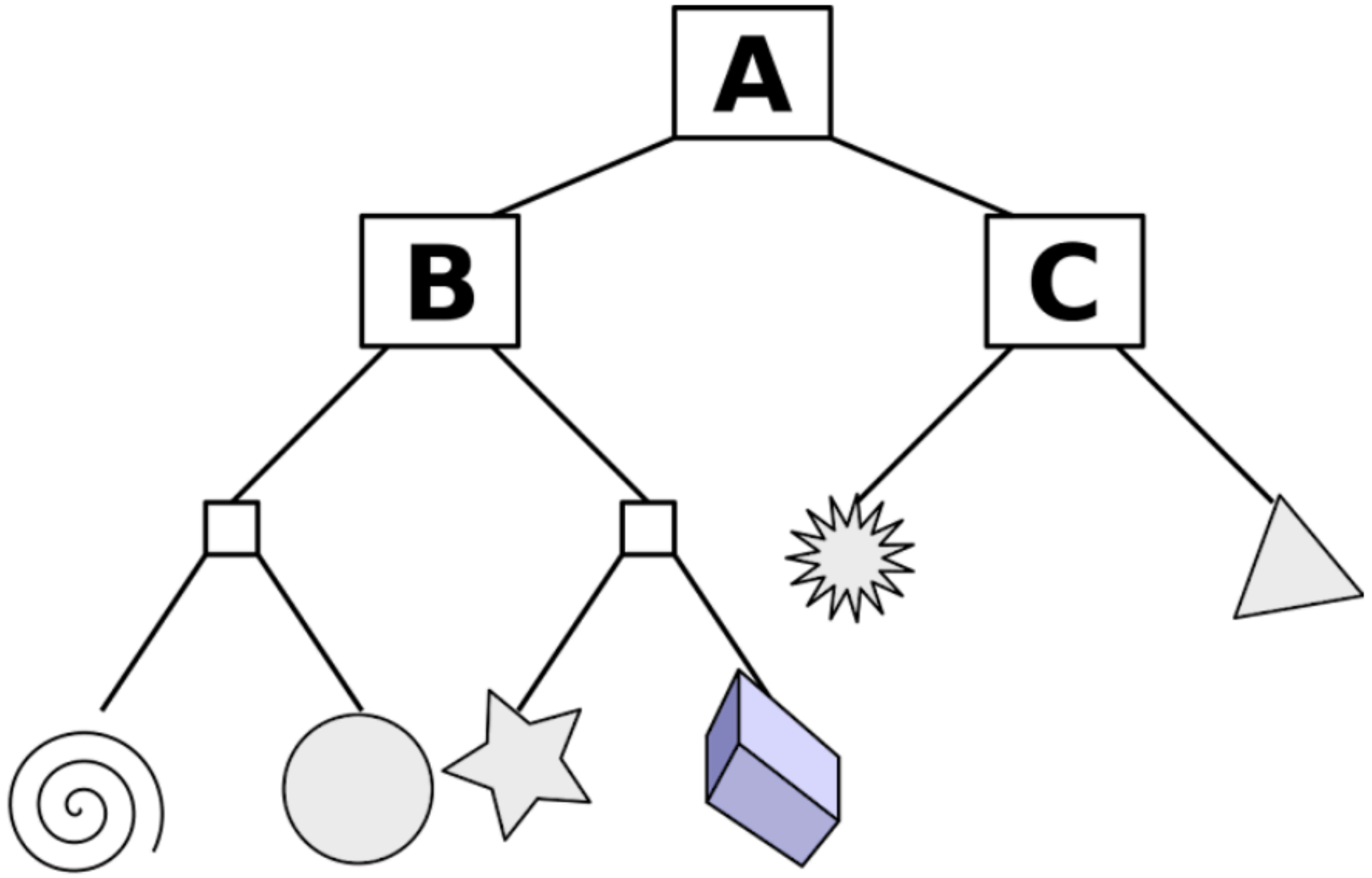
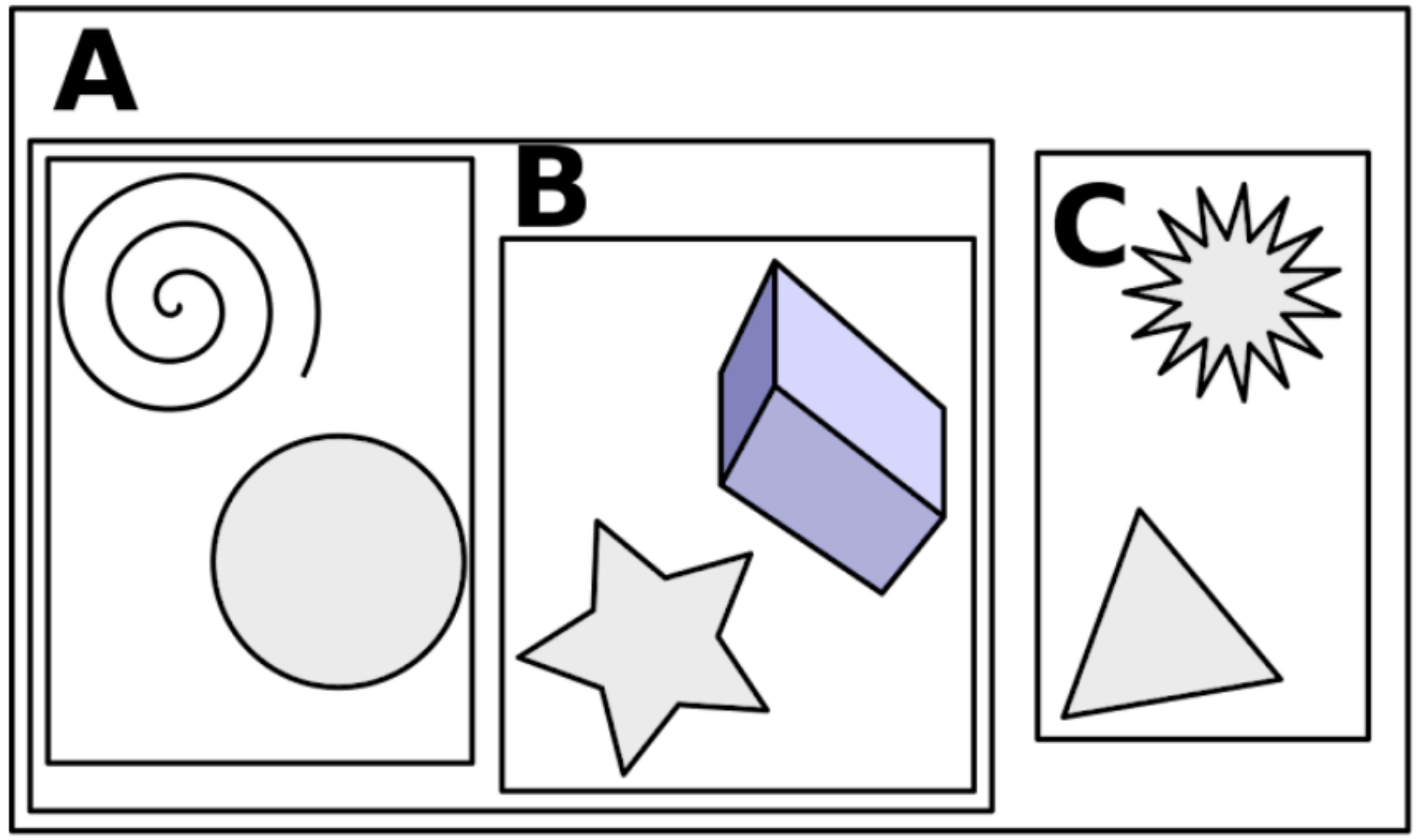


figure borrowed from Doug James

Bounding volume hierarchies

- to test against another complex object with its own BVH hierarchy, traverse trees in tandem:

```
overlap(node1, node2):
  if overlap_bv(node1.bounds, node2.bounds):
    if node1.is_leaf() and node2.is_leaf():
      return overlap_geom(node1.geom, node2.geom)
    if node1.is_leaf():
      return overlap(node1, node2.left) or
        overlap(node1, node2.right)
    if node2.is_leaf():
      return overlap(node1.left, node2) or
        overlap(node1.right, node2)
    if node2.long_axis() > node1.long_axis():
      return overlap(node1, node2.left) or
        overlap(node1, node2.right)
    else
      return overlap(node1.left, node2) or
        overlap(node1.right, node2)
  return false
```



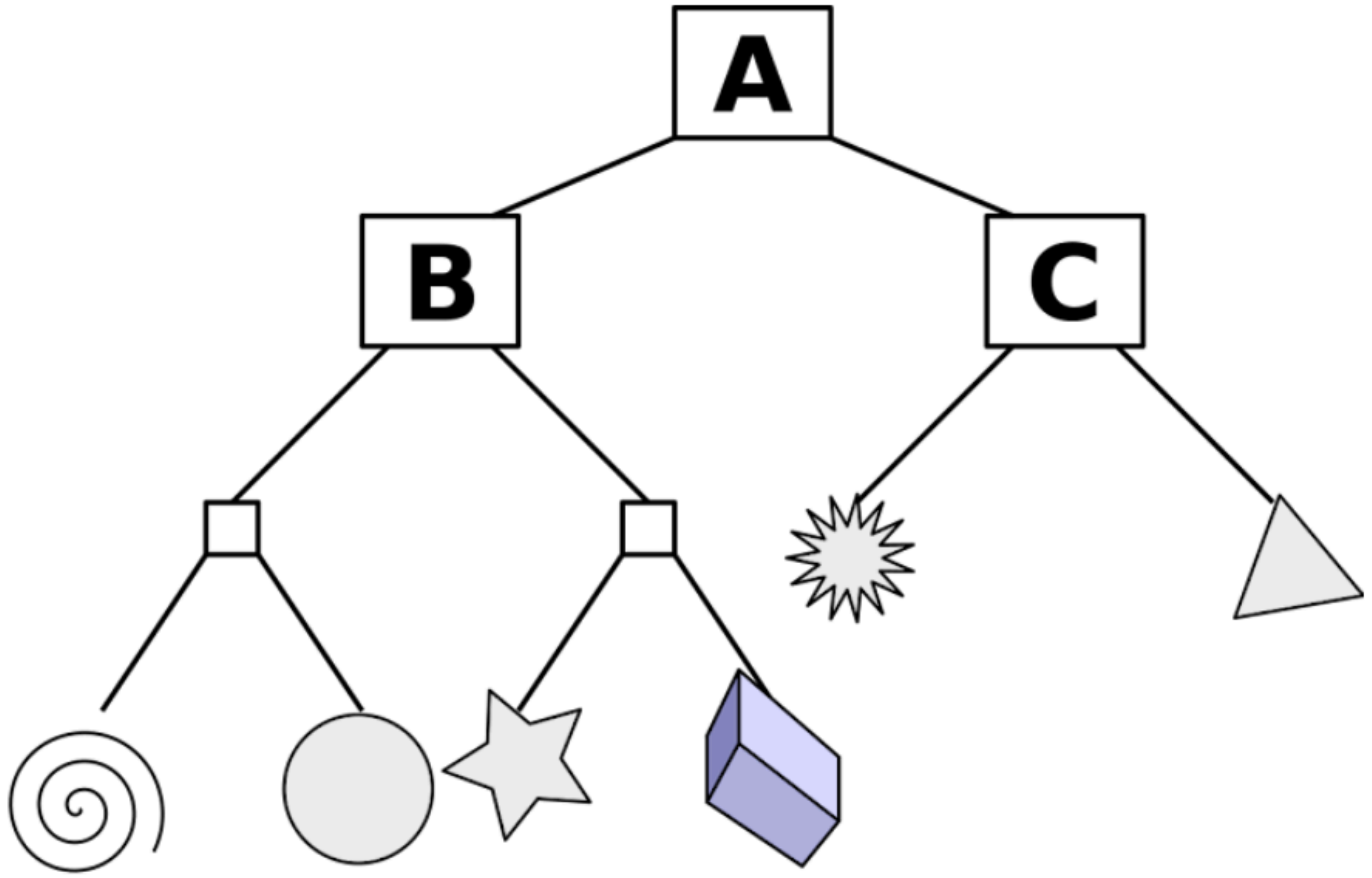
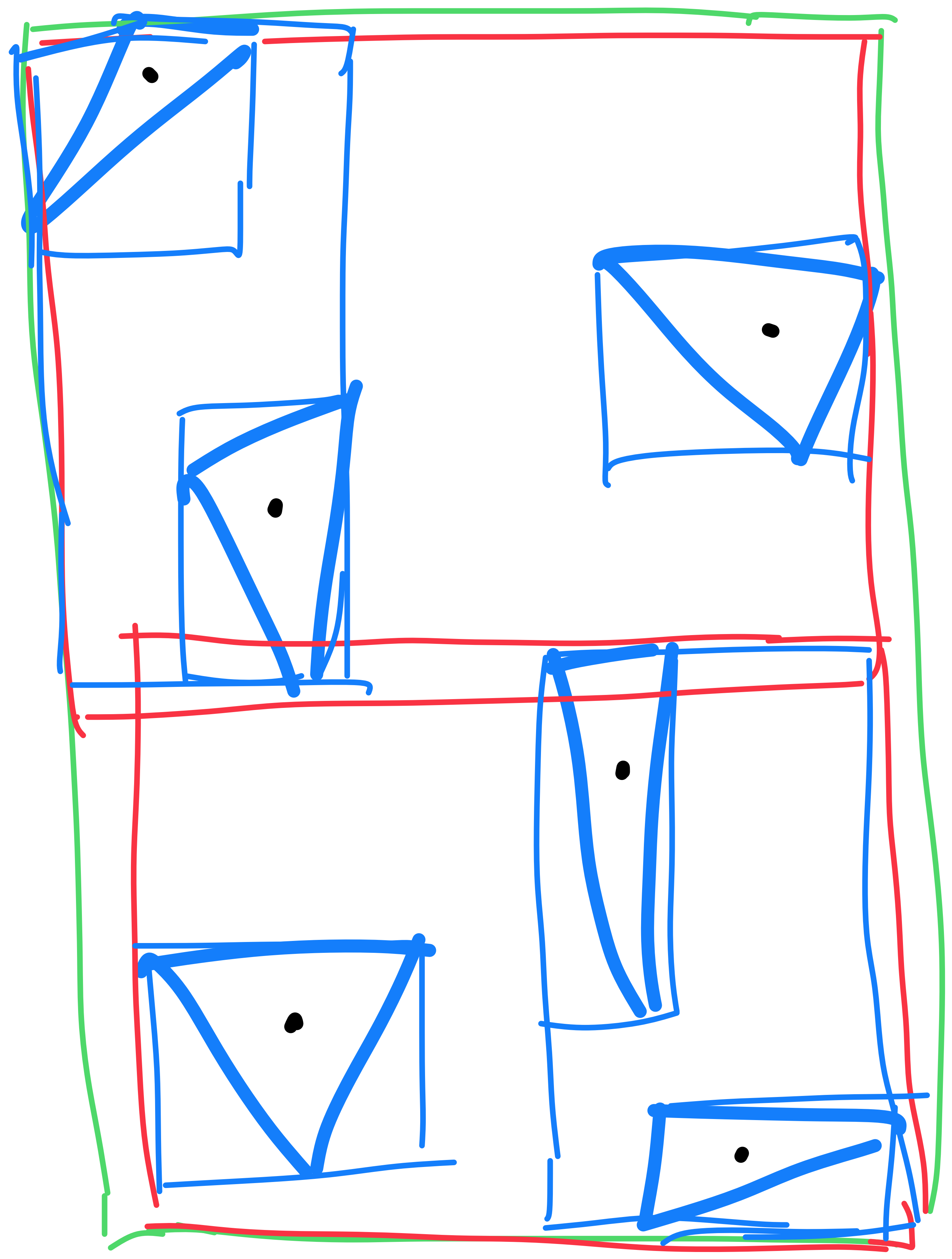


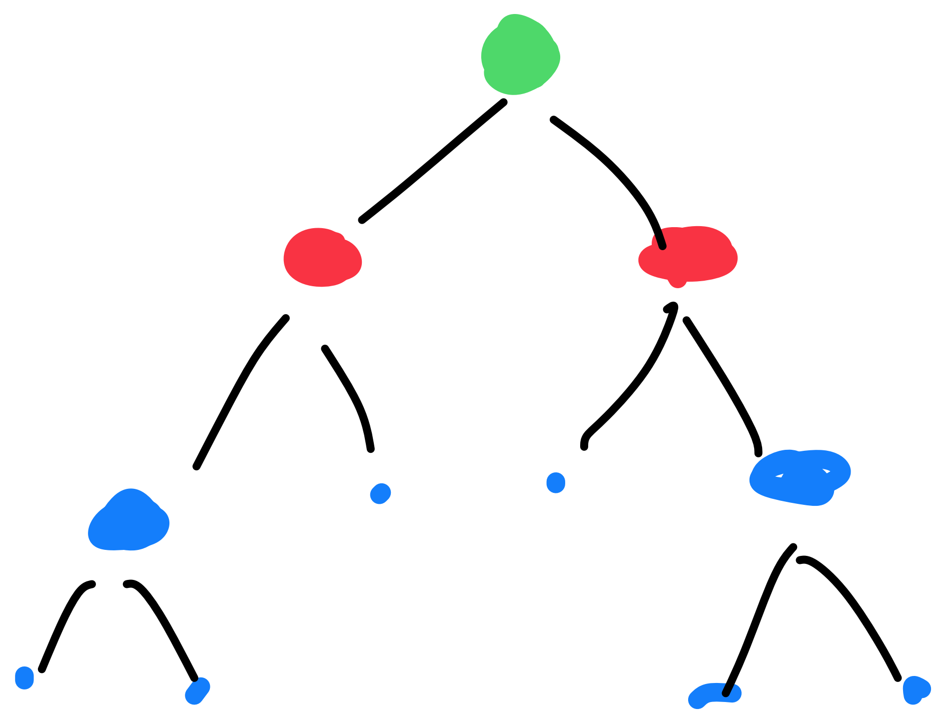
figure borrowed from Doug James

Building BVHs

Simplest way: top down splitting

- fit BV to all the geometry you have
- split geometry into two equal sized subsets
  - simple strategy: median split
  - choose axis along which to split
    (typically the longest BV axis)
  - split at median of projections of object
    centroids onto that axis
- recursively process the two halves

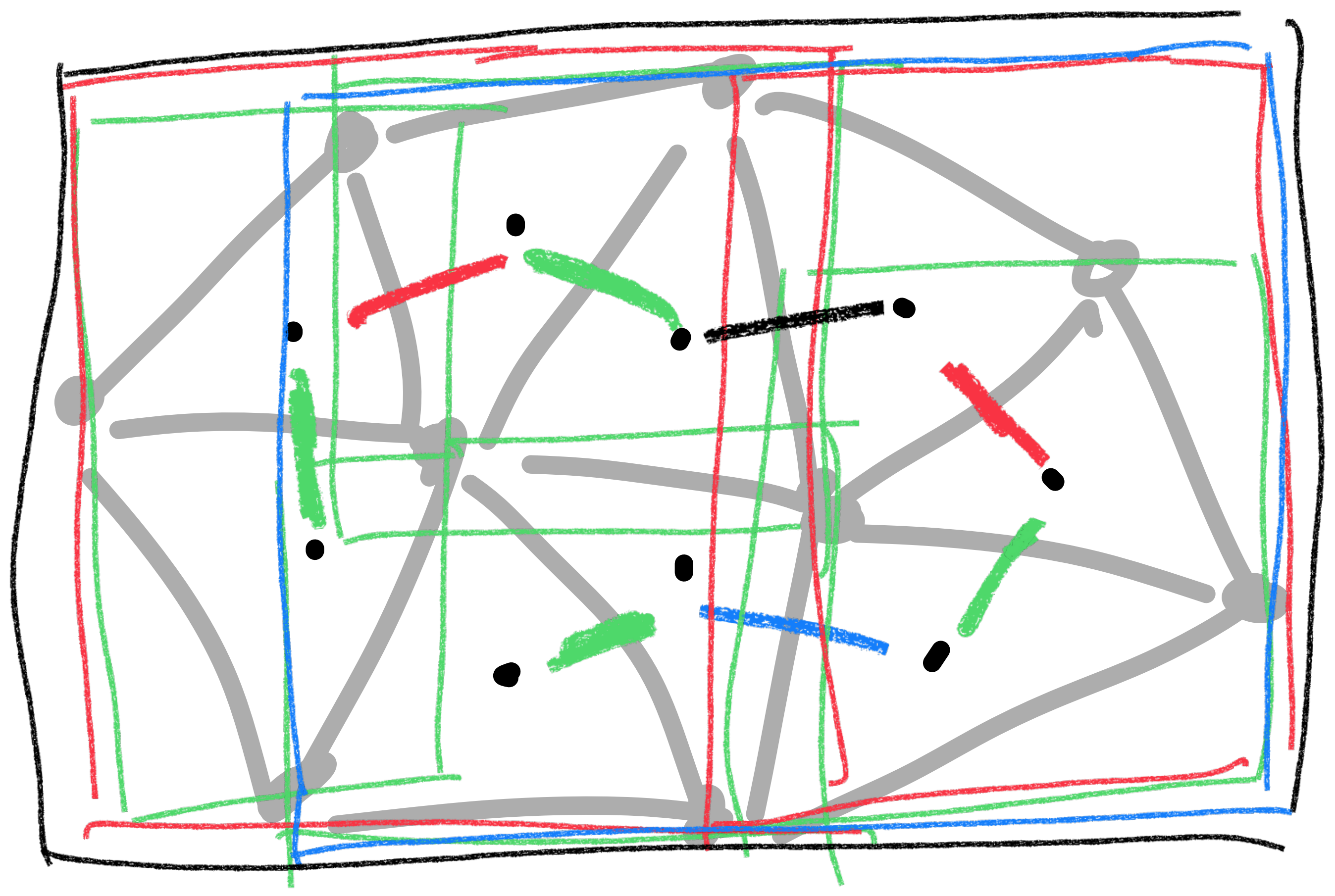


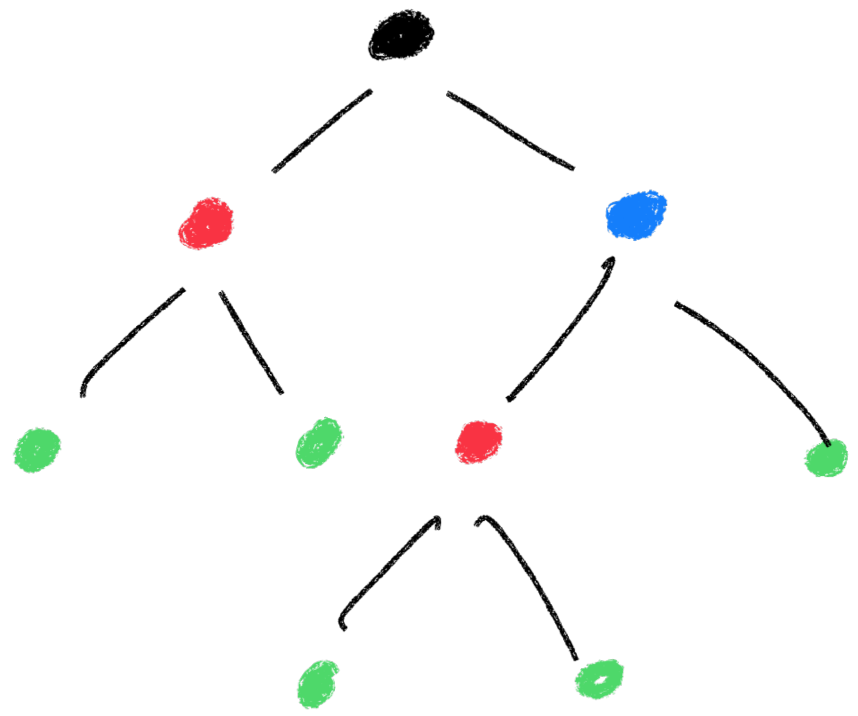


Building BVHs

Splitting according to mesh connectivity

- might want nodes to contain contiguous parts of objects
- leads to a bottom-up approach
  - build an adjacency graph of all primitives
  - repeatedly choose an edge with lowest "cost" and merge the two nodes
  - cost might be the volume of the resulting node or the height of the resulting subtree
- popular for deformables, produces trees likely to re-fit well (next slide)

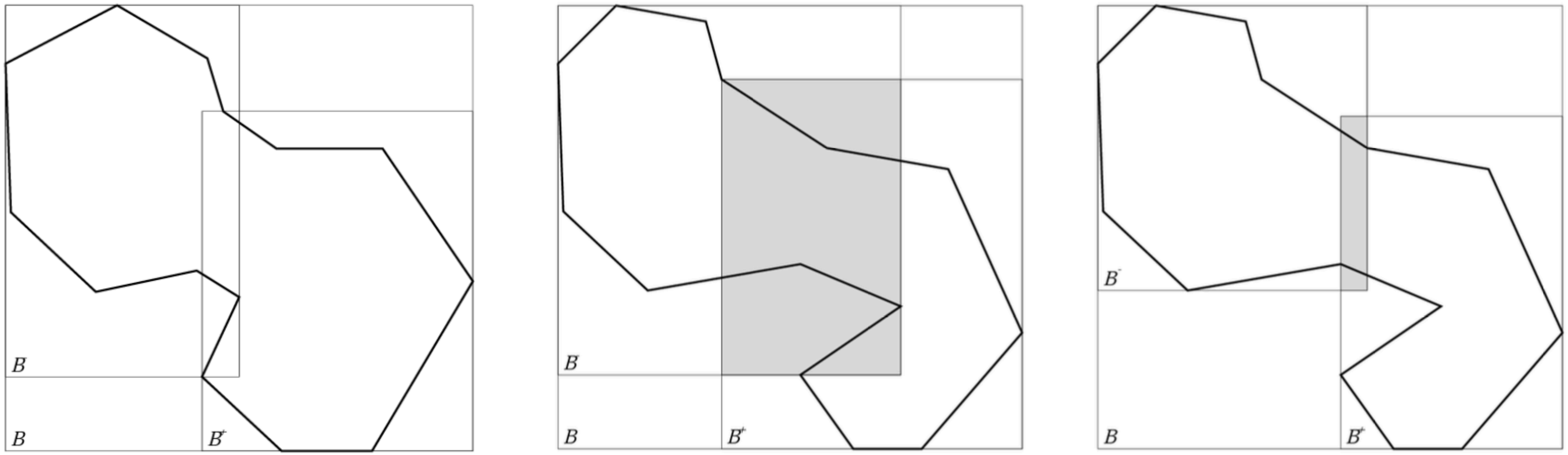




Updating BVH for deforming geometry

Geometry is different each frame—what to do?

- constructing a new tree from scratch every frame is expensive
- alternative: keep tree structure and re-fit bounds
  - simple bottom-up algorithm with reasonable memory access pattern
  - efficient for BVs that can efficiently bound their children
  - downside: can lead to increased overlap; mesh connectivity ameliorates this



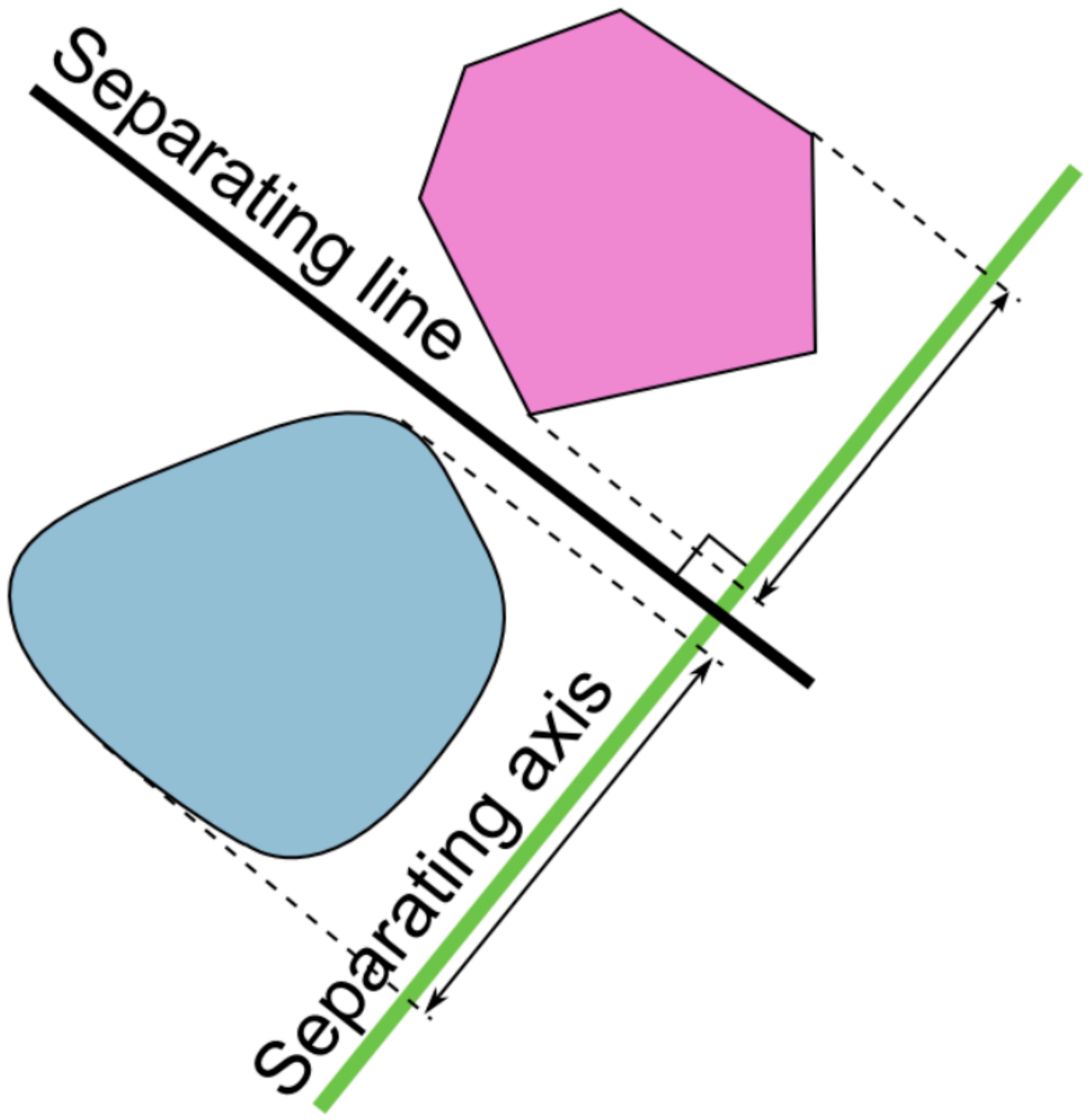
Undeformed (a) Refi tted (b) Rebuilt

[Gottschalk et al. 1996]

Finding collisions between convex polyhedra

An efficient strategy for fast BV intersection

- if the projections of two objects onto some axis are disjoint, the objects do not intersect and the axis is a *separating axis*
- if the objects do not intersect, a separating axis must exist
- for convex polygons in 2D or polyhedra in 3D, if there is no intersection then checking a finite list of potential separating axes suffices

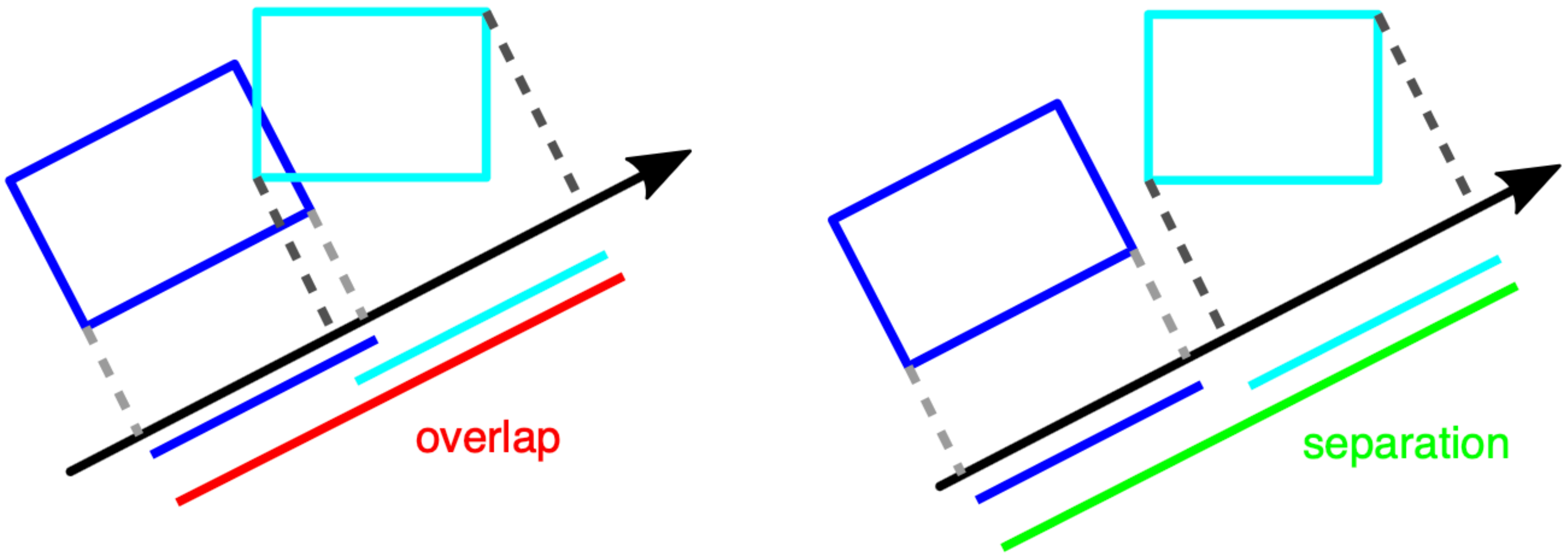


https://en.wikipedia.org/wiki/Hyperplane_separation_theorem

Examples

- 2 familiar tests for AABBs in 2D
- 4 tests for OBBs in 2D (4 distinct face normals)
- 15 tests for OBBs in 3D (6 face normals + 9 edge/edge normals)

E.g. separating axis approach for OBBs in 2D

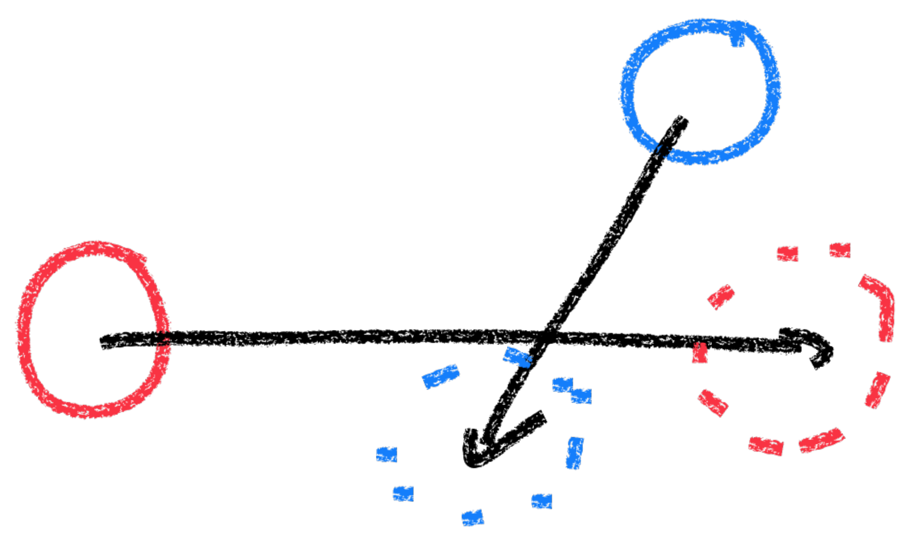


https://www.atoft.dev/posts/2020/04/12/implementing-3d-collision-resolution/

Continuous collision detection (CCD)

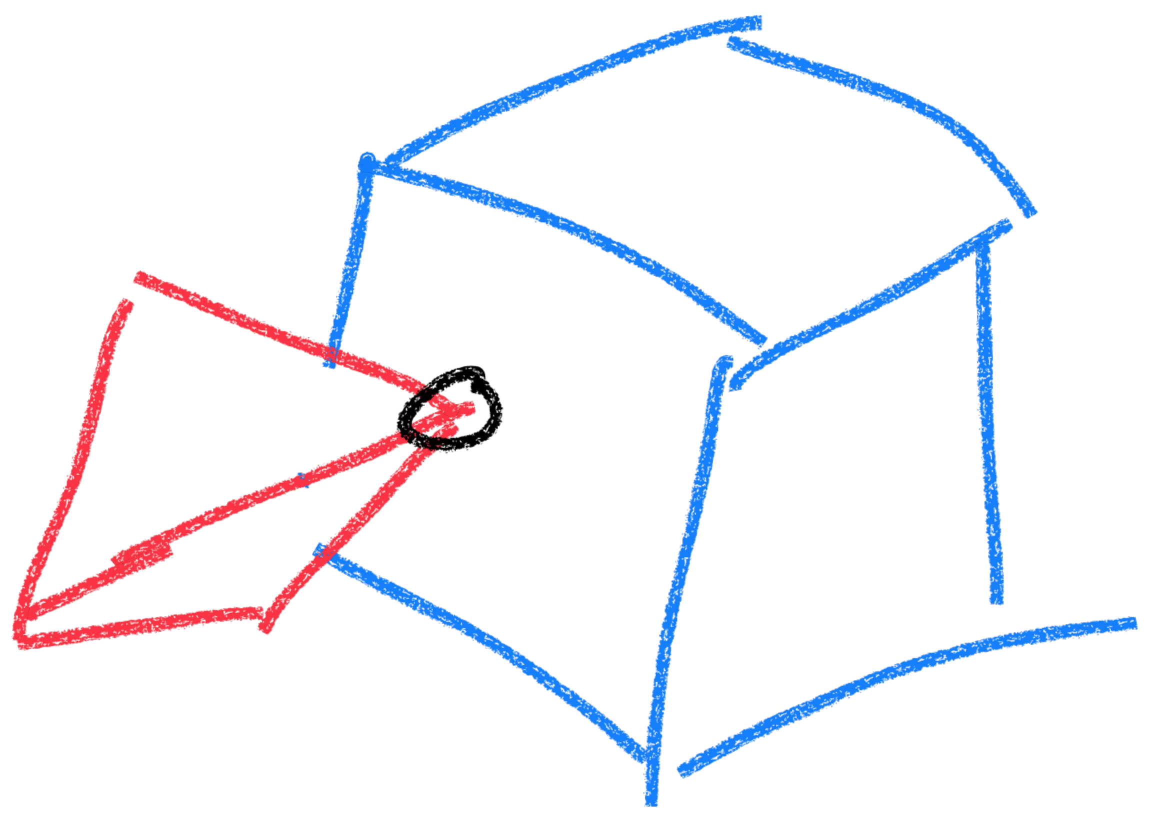
Given two moving primitives:

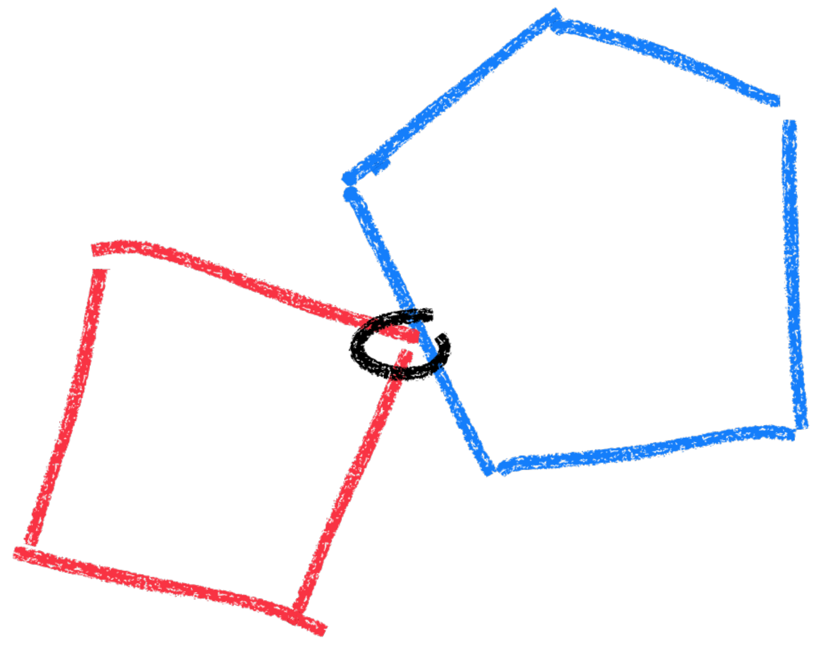
- do they collide in this time step?
- ...and if so, when and where?

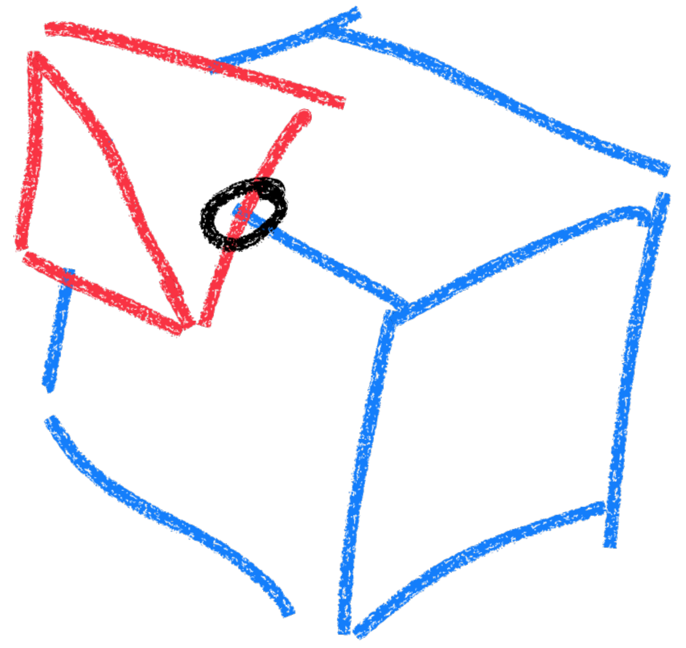


Common simplifications:

- limit to circles, spheres, triangles, line segments
- only allow for linear motion of vertices
- only consider non-degenerate cases
  - in 3D: vertex-face and edge-edge
  - in 2D: vertex-edge
- degenerate cases can be handled as an extreme case of one of these



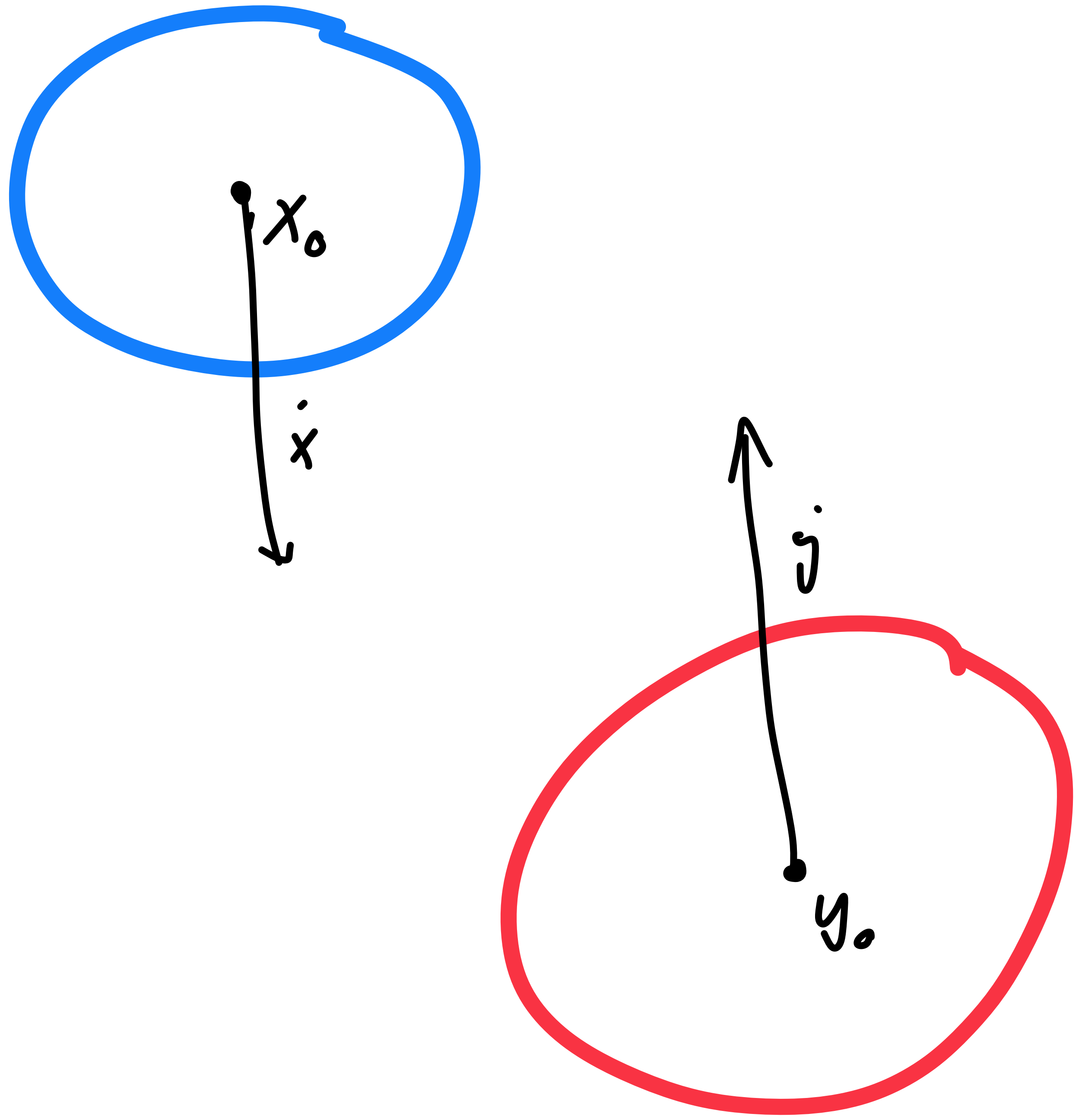




CCD for spheres

Given $\mathbf{x}_0, \dot{\mathbf{x}}, \mathbf{y}_0, \dot{\mathbf{y}}, r_x, r_y$

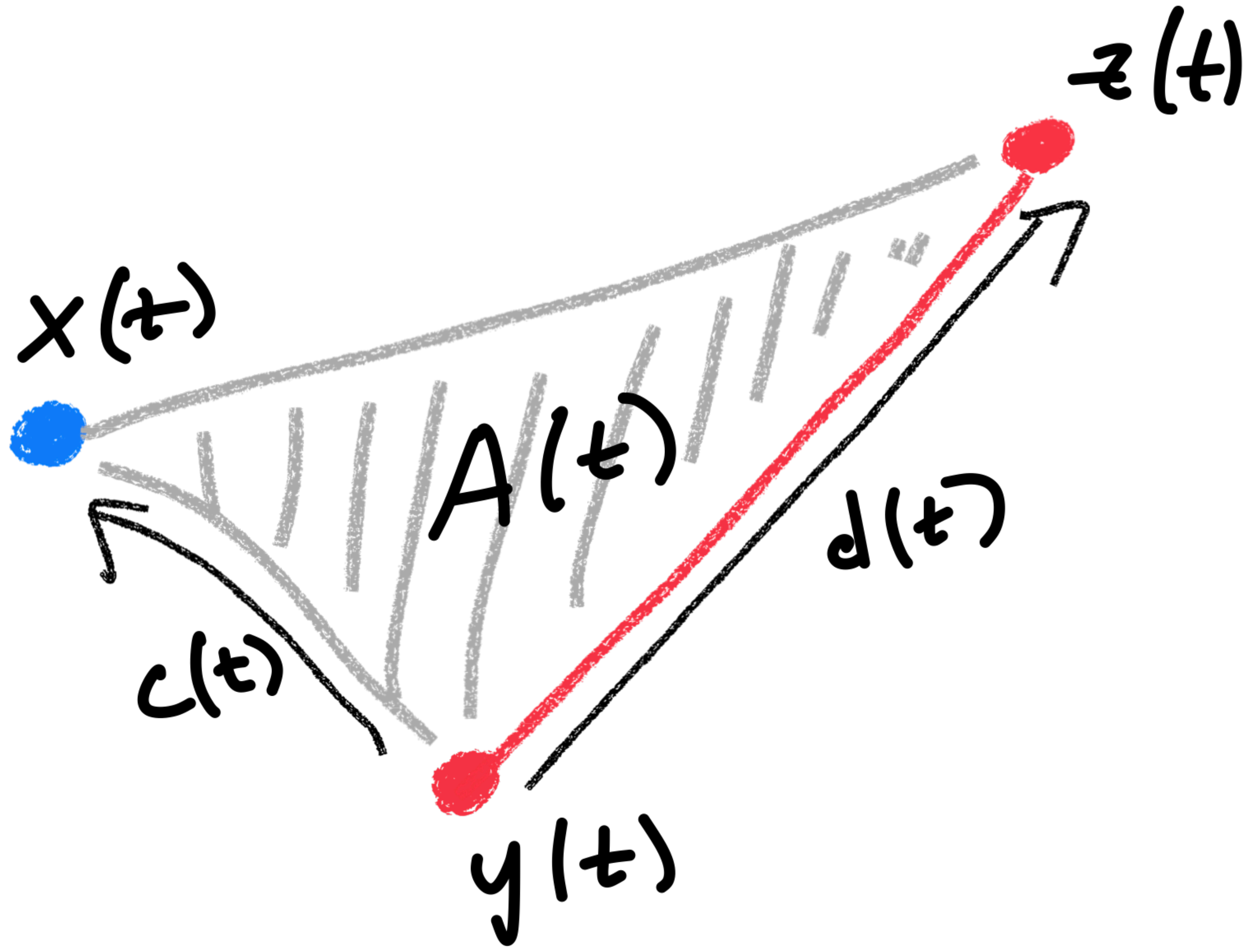
- is there a time $t \in (0,h]$ where the centers are at a distance $r_x + r_y$?
- positions are $\mathbf{x}(t) = \mathbf{x}_0 + t\dot{\mathbf{x}}$ and $\mathbf{y}(t) = \mathbf{y}_0 + t\dot{\mathbf{y}}$
- let $\mathbf{d}_0 = \mathbf{x}_0 - \mathbf{y}_0$; $\dot{\mathbf{d}} = \dot{\mathbf{x}} - \dot{\mathbf{y}}$; $R = r_x + r_y$
- difference is $\mathbf{d}(t) = \mathbf{d}_0 + t\dot{\mathbf{d}}$
- collision when $\|\mathbf{d}(t)\| = R$ or $(\mathbf{d}_0 + t\dot{\mathbf{d}}) \cdot (\mathbf{d}_0 + t\dot{\mathbf{d}}) = R^2$
- quadratic: $(\dot{\mathbf{d}} \cdot \dot{\mathbf{d}})t^2 + 2(\mathbf{d}_0 \cdot \dot{\mathbf{d}})t + (\mathbf{d}_0 \cdot \mathbf{d}_0 - R^2) = 0$
- there is a collision iff there is a root in $(0,h]$
- smallest root in $(0,h]$ is the collision time
- (déjà vu … remember ray-sphere intersection?)



CCD for line segments

The only nondegenerate case is vertex-edge

- vertex $\mathbf{x}(t)$ and edge endpoints $\mathbf{y}(t)$ and $\mathbf{z}(t)$
- given: $\mathbf{x}_0, \mathbf{y}_0, \mathbf{z}_0, \dot{\mathbf{x}}, \dot{\mathbf{y}}, \dot{\mathbf{z}}$
- collision occurs when $\{\mathbf{x}(t), \mathbf{y}(t), \mathbf{z}(t)\}$ are collinear and $\mathbf{x}$ is between $\mathbf{y}$ and $\mathbf{z}$
- simple collinearity test: area of triangle is zero
- triangle edges $\mathbf{c}(t) = \mathbf{x}(t) - \mathbf{y}(t) = \mathbf{c}_0 + t\dot{\mathbf{c}}$ and $\mathbf{d}(t) = \mathbf{z}(t) - \mathbf{y}(t) = \mathbf{d}_0 + t\dot{\mathbf{d}}$
- area $2A(t) = \mathbf{c}(t) \wedge \mathbf{d}(t)$, set to zero
- quadratic $(\dot{\mathbf{c}} \wedge \dot{\mathbf{d}})t^2 + (\mathbf{c}_0 \wedge \dot{\mathbf{d}} + \dot{\mathbf{c}} \wedge \mathbf{d}_0)t + (\mathbf{c}_0 \wedge \mathbf{d}_0) = 0$
- smallest root in $(0,h]$ for which $\mathbf{x}$ is between $\mathbf{y}$ and $\mathbf{z}$ (if any) is the collision time



$$
\begin{aligned}
\mathbf{v} \wedge \mathbf{w} &= (\mathbf{v} \times \mathbf{w})_z \\
&= v_x w_y - v_y w_x
\end{aligned}
$$

Robust quadratic formula

We all learned the quadratic formula in high school

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What they didn't tell us

- there are two equally reasonable quadratic formulas

$$t = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

- each one is inaccurate for certain cases (e.g. a or c near zero)
- if you just type in the familiar formula, you will sometimes get inaccurate collisions!

More stable procedure:

- compute $D = b^2 - 4ac$; if $D < 0$ there are no roots
- compute $r = -\frac{1}{2}\left(b + \mathrm{sign}(b)\sqrt{D}\right)$ (no subtraction, no cancellation!)
- roots are $t_1 = \frac{r}{a}$ and $t_2 = \frac{c}{r}$ (exercise: show that these are equal when $D = 0$)
- (see Numerical Recipes or other intro numerics textbooks)

CCD for triangle meshes

Here we have both edge-edge and point-face collisions

Analogous approach to 2D works

- both cases are actually the same (weird!)
- collision happens when the 4 involved vertices are coplanar, aka. volume of tetrahedron is zero
- points $\mathbf{w}(t), \mathbf{x}(t), \mathbf{y}(t), \mathbf{z}(t)$, velocities $\dot{\mathbf{w}}(t), \ldots, \dot{\mathbf{z}}(t)$
- think about tetrahedron edges $\mathbf{a} = \mathbf{x} - \mathbf{w}$, $\mathbf{b} = \mathbf{y} - \mathbf{w}$, $\mathbf{c} = \mathbf{z} - \mathbf{w}$
- $2V(t) = \det \begin{bmatrix} \mathbf{a}(t) & \mathbf{b}(t) & \mathbf{c}(t) \end{bmatrix} = \mathbf{a}(t) \cdot (\mathbf{b}(t) \times \mathbf{c}(t)) = 0$
- this is a cubic equation in t; collision time is the smallest root in $[0,h)$ for which the objects actually collide (vertex inside triangle, or line intersection inside edges)